

# LightDB 13.8-23.4 手册

Hundsun Technologies Inc

---

# LightDB 13.8-23.4 手册

Hundsun Technologies Inc

Copyright © 2022 Hundsun Technologies Inc

## 法律声明

LightDB Enterprise Postgres 版权所有 © 2022 恒生科技股份有限公司。保留所有权利。

---

---

# Table of Contents

前言	xxxv
1. 何为LightDB?	xxxv
2. 约定	xxxvi
3. 进一步的信息	xxxvi
I. 教程	1
1. 从头开始	3
1.1. 安装	3
1.2. 架构基础	3
1.3. 创建一个数据库	3
1.4. 访问数据库	5
2. SQL语言	7
2.1. 引言	7
2.2. 概念	7
2.3. 创建一个新表	7
2.4. 在表中增加行	8
2.5. 查询一个表	9
2.6. 在表之间连接	10
2.7. 聚集函数	12
2.8. 更新	14
2.9. 删除	14
3. 高级特性	16
3.1. 简介	16
3.2. 视图	16
3.3. 外键	16
3.4. 事务	17
3.5. 窗口函数	18
3.6. 继承	21
3.7. 小结	22
II. SQL 语言	23
4. SQL语法	31
4.1. 词法结构	31
4.2. 值表达式	39
4.3. 调用函数	54
5. 数据定义	57
5.1. 表基础	57
5.2. 默认值	58
5.3. 生成列	59
5.4. 约束	60
5.5. 系统列	67
5.6. 修改表	68
5.7. 权限	70
5.8. 行安全性策略	74
5.9. 模式	80
5.10. 继承	84
5.11. 表分区	87
5.12. 外部数据	98
5.13. 其他数据库对象	99
5.14. 依赖跟踪	99
5.15. LightDB DDL 规格检查	100
6. 数据操纵	101
6.1. 插入数据	101
6.2. 更新数据	102
6.3. 删除数据	103
6.4. 从修改的行中返回数据	103
6.5. LightDB DML规格检查	104

7. 查询	105
7.1. 概述	105
7.2. 表表达式	105
7.3. 选择列表	119
7.4. 组合查询	121
7.5. 行排序	122
7.6. LIMIT和OFFSET	123
7.7. VALUES列表	123
7.8. WITH查询（公共表表达式）	124
7.9. ROWNUM	130
8. 数据类型	131
8.1. 数字类型	132
8.2. 货币类型	137
8.3. 字符类型	138
8.4. 二进制数据类型	140
8.5. 日期/时间类型	141
8.6. 布尔类型	150
8.7. 枚举类型	151
8.8. 几何类型	153
8.9. 网络地址类型	155
8.10. 位串类型	157
8.11. 文本搜索类型	158
8.12. UUID类型	161
8.13. XML类型	161
8.14. JSON 类型	163
8.15. 数组	171
8.16. 组合类型	179
8.17. 范围类型	185
8.18. 域类型	190
8.19. 对象标识符类型	190
8.20. pg_lsn 类型	192
8.21. 伪类型	192
9. 函数和操作符	194
9.1. 逻辑操作符	194
9.2. 比较函数和操作符	194
9.3. 数学函数和操作符	198
9.4. 字符串函数和操作符	204
9.5. 二进制串函数和操作符	212
9.6. 位串函数和操作符	214
9.7. 模式匹配	216
9.8. 数据类型格式化函数	232
9.9. 时间/日期函数和操作符	239
9.10. 枚举支持函数	255
9.11. 几何函数和操作符	255
9.12. 网络地址函数和操作符	261
9.13. 文本搜索函数和操作符	264
9.14. UUID 函数	269
9.15. XML 函数	269
9.16. JSON 函数和操作符	283
9.17. 序列操作函数	297
9.18. 条件表达式	299
9.19. 数组函数和操作符	306
9.20. 范围函数和运算符	309
9.21. 聚集函数	310
9.22. 窗口函数	316
9.23. 子查询表达式	318
9.24. 行和数组比较	320
9.25. 集合返回函数	323

9.26.	系统信息函数和运算符	326
9.27.	系统管理函数	341
9.28.	触发器函数	353
9.29.	事件触发器函数	354
9.30.	统计信息函数	356
10.	类型转换	358
10.1.	概述	358
10.2.	操作符	359
10.3.	函数	362
10.4.	值存储	366
10.5.	UNION、CASE和相关结构	367
10.6.	SELECT的输出列	368
10.7.	类型兼容性	369
11.	索引	372
11.1.	简介	372
11.2.	索引类型	373
11.3.	多列索引	374
11.4.	索引和ORDER BY	375
11.5.	组合多个索引	375
11.6.	唯一索引	376
11.7.	表达式索引	376
11.8.	部分索引	377
11.9.	只用索引的扫描和覆盖索引	380
11.10.	操作符类和操作符族	382
11.11.	索引和排序规则	383
11.12.	检查索引使用	384
12.	全文搜索	385
12.1.	介绍	385
12.2.	表和索引	388
12.3.	控制文本搜索	390
12.4.	额外特性	397
12.5.	解析器	402
12.6.	词典	404
12.7.	配置例子	412
12.8.	测试和调试文本搜索	414
12.9.	GIN 和 GiST 索引类型	418
12.10.	ltsql支持	418
12.11.	限制	421
13.	并发控制	423
13.1.	介绍	423
13.2.	事务隔离	423
13.3.	显式锁定	428
13.4.	应用级别的数据完整性检查	432
13.5.	提醒	434
13.6.	锁定和索引	434
14.	性能提示	435
14.1.	使用EXPLAIN	435
14.2.	规划器使用的统计信息	446
14.3.	用显式JOIN子句控制规划器	450
14.4.	填充一个数据库	452
14.5.	非持久设置	454
15.	并行查询	455
15.1.	并行查询如何工作	455
15.2.	何时会用到并行查询?	456
15.3.	并行计划	456
15.4.	并行安全性	458
III.	服务器管理	460
16.	服务器设置和操作	466

16.1.	LightDB用户账户	466
16.2.	创建一个数据库集簇	466
16.3.	启动数据库服务器	468
16.4.	管理内核资源	471
16.5.	关闭服务器	478
16.6.	升级一个LightDB集簇	479
16.7.	阻止服务器欺骗	481
16.8.	加密选项	481
16.9.	用 SSL 进行安全的 TCP/IP 连接	482
16.10.	Secure TCP/IP Connections with GSSAPI Encryption	486
16.11.	使用SSH隧道的安全 TCP/IP 连接	486
16.12.	在Windows上注册事件日志	487
17.	服务器配置	488
17.1.	设置参数	488
17.2.	文件位置	491
17.3.	连接和认证	492
17.4.	资源消耗	498
17.5.	预写式日志	505
17.6.	复制	514
17.7.	查询规划	519
17.8.	错误报告和日志	526
17.9.	运行时统计数据	537
17.10.	自动清理	538
17.11.	客户端连接默认值	540
17.12.	锁管理	553
17.13.	版本和平台兼容性	553
17.14.	错误处理	555
17.15.	预置选项	555
17.16.	自定义选项	557
17.17.	开发者选项	557
17.18.	短选项	560
17.19.	Other Options	561
18.	客户端认证	563
18.1.	lt_hba.conf文件	563
18.2.	用户名映射	569
18.3.	认证方法	571
18.4.	信任认证	571
18.5.	口令认证	572
18.6.	GSSAPI 认证	572
18.7.	SSPI 认证	574
18.8.	Ident 认证	574
18.9.	Peer 认证	575
18.10.	LDAP 认证	575
18.11.	RADIUS 认证	578
18.12.	证书认证	579
18.13.	PAM 认证	579
18.14.	BSD 认证	580
18.15.	认证问题	580
19.	数据库角色	582
19.1.	数据库角色	582
19.2.	角色属性	583
19.3.	角色成员关系	584
19.4.	删除角色	585
19.5.	预定义角色	586
19.6.	函数和触发器安全性	587
20.	管理数据库	588
20.1.	概述	588
20.2.	创建一个数据库	588

20.3.	模板数据库	589
20.4.	数据库配置	590
20.5.	销毁一个数据库	590
20.6.	表空间	591
21.	本地化	593
21.1.	区域支持	593
21.2.	排序规则支持	595
21.3.	字符集支持	600
22.	日常数据库维护工作	610
22.1.	日常清理	610
22.2.	日常重建索引	616
22.3.	日志文件维护	617
23.	备份和恢复	619
23.1.	SQL转储	619
23.2.	文件系统级别备份	622
23.3.	连续归档和时间点恢复 (PITR)	623
24.	高可用、负载均衡和复制	633
24.1.	不同方案的比较	633
24.2.	日志传送后备服务器	635
24.3.	故障转移	643
24.4.	日志传送的替代方法	644
24.5.	热备	645
25.	监控数据库活动	652
25.1.	标准 Unix 工具	652
25.2.	统计收集器	653
25.3.	查看锁	680
25.4.	Progress Reporting	680
25.5.	动态追踪	687
26.	监控磁盘使用	695
26.1.	判断磁盘用量	695
26.2.	磁盘满失败	696
27.	可靠性和预写式日志	697
27.1.	可靠性	697
27.2.	预写式日志 (WAL)	698
27.3.	异步提交	699
27.4.	WAL配置	700
27.5.	WAL内部	702
28.	逻辑复制	704
28.1.	发布	704
28.2.	订阅	705
28.3.	冲突	706
28.4.	限制	706
28.5.	架构	707
28.6.	监控	707
28.7.	安全性	707
28.8.	配置设置	708
28.9.	快速设置	708
29.	即时编译 (JIT)	709
29.1.	什么是JIT编译?	709
29.2.	什么时候会用JIT?	709
29.3.	配置	710
29.4.	可扩展性	711
30.	回归测试	712
30.1.	运行测试	712
30.2.	测试评估	715
30.3.	变体比较文件	717
30.4.	TAP 测试	718
30.5.	测试覆盖检查	718

---

31. 透明数据加密 .....	720
IV. 客户端接口 .....	721
32. libpq - C 库 .....	726
32.1. 数据库连接控制函数 .....	726
32.2. 连接状态函数 .....	740
32.3. 命令执行函数 .....	745
32.4. 异步命令处理 .....	760
32.5. 一行一行地检索查询结果 .....	763
32.6. 取消进行中的查询 .....	764
32.7. 快速路径接口 .....	765
32.8. 异步提示 .....	766
32.9. COPY命令相关的函数 .....	767
32.10. 控制函数 .....	771
32.11. 杂项函数 .....	772
32.12. 通知处理 .....	775
32.13. 事件系统 .....	776
32.14. 环境变量 .....	782
32.15. 口令文件 .....	783
32.16. 连接服务文件 .....	784
32.17. 连接参数的 LDAP 查找 .....	784
32.18. SSL 支持 .....	785
32.19. 在线程化程序中的行为 .....	789
32.20. 编译 libpq 程序 .....	789
32.21. 例子程序 .....	790
33. 大对象 .....	801
33.1. 简介 .....	801
33.2. 实现特性 .....	801
33.3. 客户端接口 .....	801
33.4. 服务器端函数 .....	805
33.5. 例子程序 .....	806
34. ECPG - C中的嵌入式 SQL .....	812
34.1. 概念 .....	812
34.2. 管理数据库连接 .....	812
34.3. 运行 SQL 命令 .....	815
34.4. 使用主变量 .....	818
34.5. 动态 SQL .....	831
34.6. pgtypes 库 .....	834
34.7. 使用描述符区域 .....	846
34.8. 错误处理 .....	859
34.9. 预处理器指令 .....	865
34.10. 处理嵌入式 SQL 程序 .....	867
34.11. 库函数 .....	868
34.12. 大对象 .....	869
34.13. C++ 应用 .....	870
34.14. 嵌入式 SQL 命令 .....	874
34.15. Informix兼容模式 .....	896
34.16. Oracle 兼容模式 .....	910
34.17. 内部 .....	910
35. 信息模式 .....	913
35.1. 模式 .....	913
35.2. 数据类型 .....	913
35.3. information_schema_catalog_name .....	914
35.4. administrable_role_authorizations .....	914
35.5. applicable_roles .....	914
35.6. attributes .....	915
35.7. character_sets .....	917
35.8. check_constraint_routine_usage .....	917
35.9. check_constraints .....	918

---



---

35.10.	collations .....	918
35.11.	collation_character_set_applicability .....	919
35.12.	column_column_usage .....	919
35.13.	column_domain_usage .....	919
35.14.	column_options .....	920
35.15.	column_privileges .....	920
35.16.	column_udt_usage .....	921
35.17.	columns .....	921
35.18.	constraint_column_usage .....	924
35.19.	constraint_table_usage .....	925
35.20.	data_type_privileges .....	925
35.21.	domain_constraints .....	926
35.22.	domain_udt_usage .....	926
35.23.	domains .....	927
35.24.	element_types .....	928
35.25.	enabled_roles .....	930
35.26.	foreign_data_wrapper_options .....	931
35.27.	foreign_data_wrappers .....	931
35.28.	foreign_server_options .....	931
35.29.	foreign_servers .....	932
35.30.	foreign_table_options .....	932
35.31.	foreign_tables .....	933
35.32.	key_column_usage .....	933
35.33.	parameters .....	934
35.34.	referential_constraints .....	935
35.35.	role_column_grants .....	936
35.36.	role_routine_grants .....	936
35.37.	role_table_grants .....	937
35.38.	role_udt_grants .....	938
35.39.	role_usage_grants .....	938
35.40.	routine_privileges .....	939
35.41.	routines .....	939
35.42.	schemata .....	943
35.43.	sequences .....	944
35.44.	sql_features .....	945
35.45.	sql_implementation_info .....	945
35.46.	sql_parts .....	946
35.47.	sql_sizing .....	946
35.48.	table_constraints .....	946
35.49.	table_privileges .....	947
35.50.	tables .....	948
35.51.	transforms .....	948
35.52.	triggered_update_columns .....	949
35.53.	triggers .....	950
35.54.	udt_privileges .....	951
35.55.	usage_privileges .....	951
35.56.	user_defined_types .....	952
35.57.	user_mapping_options .....	954
35.58.	user_mappings .....	954
35.59.	view_column_usage .....	954
35.60.	view_routine_usage .....	955
35.61.	view_table_usage .....	955
35.62.	views .....	956
V.	服务器编程 .....	957
36.	扩展 SQL .....	963
36.1.	扩展性如何工作 .....	963
36.2.	LightDB类型系统 .....	963
36.3.	用户定义的函数 .....	965

---

36.4.	用户定义的过程	966
36.5.	查询语言 (SQL) 函数	966
36.6.	函数重载	981
36.7.	函数易变性分类	982
36.8.	过程语言函数	983
36.9.	内部函数	983
36.10.	C 语言函数	984
36.11.	函数优化信息	1003
36.12.	用户定义的聚集	1003
36.13.	用户定义的类型	1009
36.14.	用户定义的操作符	1013
36.15.	操作符优化信息	1014
36.16.	索引的接口扩展	1017
36.17.	打包相关对象到一个扩展中	1027
36.18.	扩展的构建基础设施	1034
37.	触发器	1039
37.1.	触发器行为概述	1039
37.2.	数据改变的可见性	1041
37.3.	用 C 编写触发器函数	1042
37.4.	一个完整的触发器实例	1044
38.	事件触发器	1048
38.1.	事件触发器行为总览	1048
38.2.	事件触发器触发矩阵	1048
38.3.	用 C 编写事件触发器函数	1052
38.4.	一个完整的事件触发器例子	1053
38.5.	一个表重写事件触发器例子	1054
39.	规则系统	1056
39.1.	查询树	1056
39.2.	视图和规则系统	1057
39.3.	物化视图	1064
39.4.	INSERT、UPDATE和DELETE上的规则	1066
39.5.	规则和权限	1077
39.6.	规则和命令状态	1078
39.7.	规则 vs 触发器	1079
40.	过程语言	1082
40.1.	安装过程语言	1082
41.	PL/pgSQL — SQL过程语言	1084
41.1.	综述	1084
41.2.	PL/pgSQL的结构	1085
41.3.	声明	1086
41.4.	表达式	1092
41.5.	基本语句	1092
41.6.	控制结构	1100
41.7.	游标	1114
41.8.	事务管理	1119
41.9.	错误和消息	1120
41.10.	触发器函数	1122
41.11.	PL/pgSQL的内部	1131
41.12.	PL/pgSQL开发提示	1134
41.13.	从Oracle PL/SQL 移植	1137
42.	PL/oraSQL — SQL过程化语言	1146
42.1.	概述	1146
42.2.	PL/oraSQL 的结构	1147
42.3.	声明	1151
42.4.	表达式	1154
42.5.	基本语句	1154
42.6.	控制结构	1158
42.7.	游标	1168

42.8.	集合	1189
42.9.	事务管理	1192
42.10.	软件包	1201
42.11.	表函数	1209
42.12.	错误处理	1210
42.13.	对象名称搜索规则	1217
42.14.	针对Oracle的增强功能	1218
43.	服务器编程接口	1220
43.1.	接口函数	1220
43.2.	接口支持函数	1253
43.3.	内存管理	1262
43.4.	事务管理	1272
43.5.	数据改变的可见性	1275
43.6.	例子	1275
44.	后台工作者进程	1279
45.	逻辑解码	1282
45.1.	逻辑解码的例子	1282
45.2.	逻辑解码概念	1284
45.3.	流复制协议接口	1285
45.4.	逻辑解码的 SQL 接口	1285
45.5.	与逻辑解码相关的系统目录	1286
45.6.	逻辑解码输出插件	1286
45.7.	逻辑解码输出写入器	1289
45.8.	逻辑解码的同步复制支持	1289
46.	复制进度追踪	1291
VI.	参考	1292
I.	SQL 命令	1298
	ABORT	1302
	ALTER AGGREGATE	1303
	ALTER COLLATION	1305
	ALTER CONVERSION	1307
	ALTER DATABASE	1308
	ALTER DEFAULT PRIVILEGES	1310
	ALTER DOMAIN	1313
	ALTER EVENT TRIGGER	1316
	ALTER EXTENSION	1317
	ALTER FOREIGN DATA WRAPPER	1320
	ALTER FOREIGN TABLE	1322
	ALTER FUNCTION	1327
	ALTER GROUP	1331
	ALTER INDEX	1333
	ALTER LANGUAGE	1336
	ALTER LARGE OBJECT	1337
	ALTER MATERIALIZED VIEW	1338
	ALTER OPERATOR	1340
	ALTER OPERATOR CLASS	1342
	ALTER OPERATOR FAMILY	1343
	ALTER POLICY	1347
	ALTER PROCEDURE	1349
	ALTER PUBLICATION	1352
	ALTER ROLE	1354
	ALTER ROUTINE	1358
	ALTER RULE	1359
	ALTER SCHEMA	1360
	ALTER SEQUENCE	1361
	ALTER SERVER	1364
	ALTER SESSION	1366
	ALTER STATISTICS	1367

---

ALTER SUBSCRIPTION .....	1368
ALTER SYSTEM .....	1370
ALTER TABLE .....	1372
ALTER TABLESPACE .....	1391
ALTER TEXT SEARCH CONFIGURATION .....	1393
ALTER TEXT SEARCH DICTIONARY .....	1395
ALTER TEXT SEARCH PARSER .....	1397
ALTER TEXT SEARCH TEMPLATE .....	1398
ALTER TRIGGER .....	1399
ALTER TYPE .....	1401
ALTER USER .....	1405
ALTER USER MAPPING .....	1406
ALTER VIEW .....	1407
ANALYZE .....	1409
BEGIN .....	1412
CALL .....	1414
CHECKPOINT .....	1415
CLOSE .....	1416
CLUSTER .....	1417
COMMENT .....	1419
COMMIT .....	1423
COMMIT PREPARED .....	1424
COPY .....	1425
CREATE ACCESS METHOD .....	1434
CREATE AGGREGATE .....	1435
CREATE CAST .....	1442
CREATE COLLATION .....	1446
CREATE CONVERSION .....	1448
CREATE DATABASE .....	1450
CREATE DOMAIN .....	1456
CREATE EVENT TRIGGER .....	1459
CREATE EXTENSION .....	1461
CREATE FOREIGN DATA WRAPPER .....	1463
CREATE FOREIGN TABLE .....	1465
CREATE FUNCTION .....	1469
CREATE GROUP .....	1476
CREATE INDEX .....	1477
CREATE LANGUAGE .....	1485
CREATE MATERIALIZED VIEW .....	1487
CREATE OPERATOR .....	1489
CREATE OPERATOR CLASS .....	1492
CREATE OPERATOR FAMILY .....	1495
CREATE POLICY .....	1496
CREATE PROCEDURE .....	1501
CREATE PUBLICATION .....	1504
CREATE ROLE .....	1506
CREATE RULE .....	1510
CREATE SCHEMA .....	1513
CREATE SEQUENCE .....	1515
CREATE SERVER .....	1519
CREATE STATISTICS .....	1521
CREATE SUBSCRIPTION .....	1523
CREATE TABLE .....	1526
CREATE TABLE AS .....	1555
CREATE TABLESPACE .....	1558
CREATE TEXT SEARCH CONFIGURATION .....	1560
CREATE TEXT SEARCH DICTIONARY .....	1561
CREATE TEXT SEARCH PARSER .....	1563

---

CREATE TEXT SEARCH TEMPLATE .....	1565
CREATE TRANSFORM .....	1566
CREATE TRIGGER .....	1568
CREATE TYPE .....	1574
CREATE USER .....	1585
CREATE USER MAPPING .....	1586
CREATE VIEW .....	1587
DEALLOCATE .....	1592
DECLARE .....	1593
DELETE .....	1596
DISCARD .....	1599
DO .....	1600
DROP ACCESS METHOD .....	1602
DROP AGGREGATE .....	1603
DROP CAST .....	1605
DROP COLLATION .....	1606
DROP CONVERSION .....	1607
DROP DATABASE .....	1608
DROP DOMAIN .....	1609
DROP EVENT TRIGGER .....	1610
DROP EXTENSION .....	1611
DROP FOREIGN DATA WRAPPER .....	1612
DROP FOREIGN TABLE .....	1613
DROP FUNCTION .....	1614
DROP GROUP .....	1616
DROP INDEX .....	1617
DROP LANGUAGE .....	1619
DROP MATERIALIZED VIEW .....	1620
DROP OPERATOR .....	1621
DROP OPERATOR CLASS .....	1623
DROP OPERATOR FAMILY .....	1625
DROP OWNED .....	1627
DROP POLICY .....	1628
DROP PROCEDURE .....	1629
DROP PUBLICATION .....	1631
DROP ROLE .....	1632
DROP ROUTINE .....	1633
DROP RULE .....	1634
DROP SCHEMA .....	1635
DROP SEQUENCE .....	1636
DROP SERVER .....	1637
DROP STATISTICS .....	1638
DROP SUBSCRIPTION .....	1639
DROP TABLE .....	1640
DROP TABLESPACE .....	1642
DROP TEXT SEARCH CONFIGURATION .....	1643
DROP TEXT SEARCH DICTIONARY .....	1644
DROP TEXT SEARCH PARSER .....	1645
DROP TEXT SEARCH TEMPLATE .....	1646
DROP TRANSFORM .....	1647
DROP TRIGGER .....	1648
DROP TYPE .....	1649
DROP USER .....	1650
DROP USER MAPPING .....	1651
DROP VIEW .....	1652
END .....	1653
EXECUTE .....	1654
EXPLAIN .....	1655

---

FETCH .....	1660
GRANT .....	1664
IMPORT FOREIGN SCHEMA .....	1669
INSERT .....	1671
LISTEN .....	1680
LOAD .....	1682
LOCK .....	1683
MOVE .....	1686
MERGE .....	1688
NOTIFY .....	1692
PREPARE .....	1694
PREPARE TRANSACTION .....	1697
REASSIGN OWNED .....	1699
REFRESH MATERIALIZED VIEW .....	1700
REINDEX .....	1702
RELEASE SAVEPOINT .....	1706
REPLACE INTO .....	1707
RESET .....	1710
REVOKE .....	1711
ROLLBACK .....	1715
ROLLBACK PREPARED .....	1716
ROLLBACK TO SAVEPOINT .....	1717
SAVEPOINT .....	1719
SECURITY LABEL .....	1721
SELECT .....	1724
SELECT INTO .....	1752
SET .....	1754
SET CONSTRAINTS .....	1757
SET ROLE .....	1758
SET SESSION AUTHORIZATION .....	1760
SET TRANSACTION .....	1762
SHOW .....	1765
START TRANSACTION .....	1769
TRUNCATE .....	1770
UNLISTEN .....	1772
UPDATE .....	1774
VACUUM .....	1779
VALUES .....	1783
II. LightDB 客户端应用 .....	1786
clusterdb .....	1787
createdb .....	1790
lt_createdb .....	1794
createuser .....	1797
dropdb .....	1801
dropuser .....	1804
ecpg .....	1807
executor .....	1809
lightdb_service.py .....	1810
lt_distributed_probackup.py .....	1815
lt_distributed_dump.py .....	1821
lt_distributed_restore.py .....	1832
ltdts_recvlogical .....	1839
ltdts_ora .....	1843
ltdts_mysql .....	1846
ltldr .....	1850
ltuldr .....	1853
lt_basebackup .....	1856
lt_config .....	1863

---

lt_dump	1866
lt_dumpall	1879
lt_isready	1886
lt_probackup	1888
lt_receivewal	1938
lt_recvlogical	1942
lt_restore	1946
lt_verifybackup	1955
ltbench	1958
ltcenter	1975
ltsql	1977
reindexdb	2017
vacuumdb	2021
III. LightDB 服务器应用	2027
initdb	2028
lt_initdb	2033
lt_patch	2038
lt_check	2040
lt_archivecleanup	2042
lt_checksums	2044
lt_controldata	2046
lt_ctl	2047
lt_resetwal	2052
lt_rewind	2055
lt_test_fsync	2059
lt_test_timing	2060
lt_upgrade	2063
lt_upgrade_check	2071
lt_waldump	2073
lightdb	2075
postmaster	2082
IV. LightDB 工具应用程序	2083
check_list	2084
iftop	2085
iotop	2086
keepalived	2087
linux-ftools	2088
ltloader	2089
lvs	2090
ntp	2091
ora2pg	2092
PostgREST	2093
tshark	2094
vmtouch	2095
VII. 内部	2096
47. LightDB内部概述	2102
47.1. 一个查询的路径	2102
47.2. 连接如何建立	2102
47.3. 分析器阶段	2103
47.4. LightDB规则系统	2103
47.5. 规划器/优化器	2104
47.6. 执行器	2105
48. 系统目录	2106
48.1. 概述	2106
48.2. lt_attri_ext	2108
48.3. lt_database_ext	2108
48.4. lt_global_temp_table_ext	2108
48.5. lt_package	2109

---

48.6.	lt_partition_ext .....	2109
48.7.	lt_proc_ext .....	2110
48.8.	lt_type_ext .....	2110
48.9.	lt_variable .....	2111
48.10.	pg_aggregate .....	2111
48.11.	pg_am .....	2113
48.12.	pg_amop .....	2113
48.13.	pg_amproc .....	2114
48.14.	pg_attrdef .....	2115
48.15.	pg_attribute .....	2115
48.16.	pg_authid .....	2117
48.17.	pg_auth_members .....	2118
48.18.	pg_cast .....	2118
48.19.	pg_class .....	2119
48.20.	pg_collation .....	2121
48.21.	pg_constraint .....	2122
48.22.	pg_conversion .....	2124
48.23.	pg_database .....	2124
48.24.	pg_db_role_setting .....	2125
48.25.	pg_default_acl .....	2126
48.26.	pg_depend .....	2126
48.27.	pg_description .....	2128
48.28.	pg_enum .....	2128
48.29.	pg_event_trigger .....	2129
48.30.	pg_extension .....	2129
48.31.	pg_foreign_data_wrapper .....	2130
48.32.	pg_foreign_server .....	2130
48.33.	pg_foreign_table .....	2131
48.34.	pg_index .....	2131
48.35.	pg_inherits .....	2132
48.36.	pg_init_privs .....	2133
48.37.	pg_language .....	2133
48.38.	pg_largeobject .....	2134
48.39.	pg_largeobject_metadata .....	2134
48.40.	pg_namespace .....	2135
48.41.	pg_opclass .....	2135
48.42.	pg_operator .....	2136
48.43.	pg_opfamily .....	2137
48.44.	pg_partitioned_table .....	2137
48.45.	pg_policy .....	2138
48.46.	pg_proc .....	2138
48.47.	pg_publication .....	2140
48.48.	pg_publication_rel .....	2141
48.49.	pg_range .....	2141
48.50.	pg_replication_origin .....	2142
48.51.	pg_rewrite .....	2142
48.52.	pg_seclabel .....	2143
48.53.	pg_sequence .....	2143
48.54.	pg_shdepend .....	2144
48.55.	pg_shdescription .....	2145
48.56.	pg_shseclabel .....	2145
48.57.	pg_statistic .....	2146
48.58.	pg_statistic_ext .....	2147
48.59.	pg_statistic_ext_data .....	2147
48.60.	pg_subscription .....	2148
48.61.	pg_subscription_rel .....	2149
48.62.	pg_tablespace .....	2149
48.63.	pg_transform .....	2149



48.64.	pg_trigger	2150
48.65.	pg_ts_config	2151
48.66.	pg_ts_config_map	2152
48.67.	pg_ts_dict	2152
48.68.	pg_ts_parser	2153
48.69.	pg_ts_template	2153
48.70.	pg_type	2154
48.71.	pg_user_mapping	2157
48.72.	系统视图	2157
48.73.	pg_available_extensions	2158
48.74.	pg_available_extension_versions	2158
48.75.	pg_config	2159
48.76.	pg_cursors	2159
48.77.	pg_file_settings	2160
48.78.	pg_group	2161
48.79.	pg_hba_file_rules	2161
48.80.	pg_indexes	2162
48.81.	pg_locks	2162
48.82.	pg_matviews	2164
48.83.	pg_policies	2165
48.84.	pg_prepared_statements	2165
48.85.	pg_prepared_xacts	2166
48.86.	pg_publication_tables	2166
48.87.	pg_replication_origin_status	2167
48.88.	pg_replication_slots	2167
48.89.	pg_roles	2168
48.90.	pg_rules	2169
48.91.	pg_seclabels	2169
48.92.	pg_sequences	2170
48.93.	pg_settings	2171
48.94.	pg_shadow	2173
48.95.	pg_shmem_allocations	2173
48.96.	pg_stats	2174
48.97.	pg_stats_ext	2175
48.98.	pg_tables	2176
48.99.	pg_timezone_abbrevs	2176
48.100.	pg_timezone_names	2177
48.101.	pg_user	2177
48.102.	pg_user_mappings	2178
48.103.	pg_views	2178
49.	前端/后端协议	2180
49.1.	概述	2180
49.2.	消息流	2181
49.3.	SASL认证	2193
49.4.	流复制协议	2194
49.5.	逻辑流复制协议	2201
49.6.	消息数据类型	2202
49.7.	消息格式	2202
49.8.	错误和通知消息域	2218
49.9.	逻辑复制消息格式	2220
49.10.	自协议2.0以来的变化总结	2224
50.	LightDB编码习惯	2226
50.1.	格式化	2226
50.2.	在服务器中报告错误	2226
50.3.	错误消息风格指导	2229
50.4.	其他编码习惯	2233
51.	本国语言支持	2235
51.1.	给翻译者	2235

51.2.	给编程者 .....	2237
52.	编写一个过程语言处理器 .....	2240
53.	编写一个外部数据包装器 .....	2243
53.1.	外部数据包装器函数 .....	2243
53.2.	外部数据包装器回调例程 .....	2243
53.3.	外部数据包装器助手函数 .....	2255
53.4.	外部数据包装器查询规划 .....	2257
53.5.	外部数据包装器中的行锁定 .....	2258
54.	编写一种表采样方法 .....	2260
54.1.	采样方法支持函数 .....	2260
55.	编写一个自定义扫描提供者 .....	2263
55.1.	创建自定义扫描路径 .....	2263
55.2.	创建自定义扫描计划 .....	2264
55.3.	执行自定义扫描 .....	2265
56.	遗传查询优化器 .....	2268
56.1.	将查询处理看成是一个复杂的优化问题 .....	2268
56.2.	遗传算法 .....	2268
56.3.	LightDB 中的遗传查询优化 (GEQO) .....	2269
56.4.	进一步阅读 .....	2270
57.	表访问方法接口定义 .....	2271
58.	索引访问方法接口定义 .....	2272
58.1.	索引的基本 API 结构 .....	2272
58.2.	索引访问方法函数 .....	2274
58.3.	索引扫描 .....	2279
58.4.	索引锁定考虑 .....	2280
58.5.	索引唯一性检查 .....	2281
58.6.	索引开销估计函数 .....	2282
59.	通用WAL 记录 .....	2285
60.	B-树索引 .....	2287
60.1.	简介 .....	2287
60.2.	B-树操作符类的行为 .....	2287
60.3.	B-树支持函数 .....	2288
60.4.	实现 .....	2290
61.	GiST 索引 .....	2293
61.1.	简介 .....	2293
61.2.	内建操作符类 .....	2293
61.3.	可扩展性 .....	2294
61.4.	实现 .....	2304
61.5.	示例 .....	2305
62.	GIN 索引 .....	2306
62.1.	简介 .....	2306
62.2.	内建操作符类 .....	2306
62.3.	可扩展性 .....	2306
62.4.	实现 .....	2309
62.5.	GIN 提示和技巧 .....	2310
62.6.	限制 .....	2311
62.7.	例子 .....	2311
63.	哈希索引 .....	2312
63.1.	概述 .....	2312
63.2.	实现 .....	2312
64.	数据库物理存储 .....	2314
64.1.	数据库文件布局 .....	2314
64.2.	TOAST .....	2316
64.3.	空闲空间映射 .....	2318
64.4.	可见性映射 .....	2318
64.5.	初始化分支 .....	2318
64.6.	数据库页面布局 .....	2319
65.	系统目录声明和初始内容 .....	2322

65.1.	系统目录声明规则	2322
65.2.	系统目录初始数据	2323
65.3.	BKI文件格式	2327
65.4.	BKI命令	2327
65.5.	自举BKI文件的结构	2328
65.6.	BKI例子	2329
66.	规划器如何使用统计信息	2330
66.1.	行估计例子	2330
66.2.	多变量统计例子	2335
66.3.	规划器统计和安全	2338
67.	备份清单格式	2339
67.1.	备份清单顶级(Top-level)对象	2339
67.2.	Backup Manifest File Object	2339
67.3.	Backup Manifest WAL Range Object	2340
VIII.	Oracle/MySQL兼容	2341
A.	Oracle兼容特性	2343
B.	MySQL兼容特性	2351
B.1.	MySQL兼容特性列表	2351
IX.	附录	2375
C.	LightDB错误代码	2384
D.	日期/时间支持	2393
D.1.	日期/时间输入解释	2393
D.2.	处理无效或不明确的时间戳	2394
D.3.	日期/时间关键词	2395
D.4.	日期/时间配置文件	2395
D.5.	POSIX 时区规范	2397
D.6.	单位的历史	2398
D.7.	儒略日期	2399
E.	SQL关键词	2401
F.	SQL 符合性	2436
F.1.	已支持特性	2437
F.2.	未支持特性	2448
F.3.	XML对于SQL的限制和符合性/XML	2457
G.	版本说明	2460
G.1.	版本发布 13.8-23.3	2460
G.2.	版本发布 13.8-23.2	2469
G.3.	版本发布 13.8-23.1	2478
G.4.	版本发布 13.8-22.4	2484
G.5.	版本发布 13.8-22.3	2489
G.6.	版本发布 13.3-22.2	2492
G.7.	版本发布 13.3-22.1	2494
G.8.	13.3-21.3版本发布	2497
G.9.	版本发布 13.3-21.2	2499
G.10.	版本 13.3-21.1	2500
H.	额外提供的模块	2502
H.1.	adminpack	2502
H.2.	amcheck	2503
H.3.	auth_delay	2506
H.4.	auto_explain	2507
H.5.	bloom	2509
H.6.	btree_gin	2512
H.7.	btree_gist	2513
H.8.	canopy	2514
H.9.	citext	2514
H.10.	cube	2516
H.11.	dblink	2520
H.12.	dict_int	2549
H.13.	dict_xsyn	2550

---

H. 14.	file_fdw .....	2551
H. 15.	fuzzystrmatch .....	2553
H. 16.	hstore .....	2556
H. 17.	intagg .....	2562
H. 18.	intarray .....	2563
H. 19.	isn .....	2566
H. 20.	lo .....	2569
H. 21.	lt_buffercache .....	2570
H. 22.	lt_bulkload (批量加载) .....	2572
H. 23.	lt_cheat_funcs .....	2582
H. 24.	ltcluster .....	2590
H. 25.	lt_cron .....	2590
H. 26.	lt_freespacemap .....	2594
H. 27.	lt_hint_plan .....	2595
H. 28.	lt_log_long_xact .....	2649
H. 29.	lt_ope .....	2651
H. 30.	lt_pathman .....	2655
H. 31.	lt_patch .....	2668
H. 32.	lt_prewarm .....	2669
H. 33.	lt_proctab .....	2671
H. 34.	lt_profile (轻量级开发工具配置文件) .....	2673
H. 35.	lt_qualstats .....	2723
H. 36.	lt_show_plans .....	2727
H. 37.	lt_sm .....	2729
H. 38.	lt_sql_inspect .....	2732
H. 39.	lt_standby_forward .....	2739
H. 40.	lt_stat_activity .....	2740
H. 41.	lt_stat_statements .....	2740
H. 42.	lt_tempfile .....	2746
H. 43.	lt_trgm .....	2747
H. 44.	lt_visibility .....	2752
H. 45.	ltaudit .....	2753
H. 46.	ltdcrypto .....	2759
H. 47.	ltdfce .....	2771
H. 48.	ltdfincore .....	2773
H. 49.	ltdtree .....	2778
H. 50.	ltdthrowlocks .....	2784
H. 51.	ltdstattuple .....	2785
H. 52.	ltdwalminer .....	2789
H. 53.	myfce .....	2789
H. 54.	mysql_fdw .....	2844
H. 55.	oracle_fdw .....	2847
H. 56.	orafce .....	2857
H. 57.	pageinspect .....	3026
H. 58.	passwordcheck .....	3033
H. 59.	postgres_fdw .....	3034
H. 60.	pldebugger .....	3039
H. 61.	rum .....	3040
H. 62.	seg .....	3044
H. 63.	sepgsql .....	3046
H. 64.	spi .....	3054
H. 65.	sslinfo .....	3055
H. 66.	system_stats .....	3056
H. 67.	tablefunc .....	3059
H. 68.	tcn .....	3068
H. 69.	test_decoding .....	3069
H. 70.	tsm_system_rows .....	3070
H. 71.	tsm_system_time .....	3070

---

---

H. 72.	tsvector2 .....	3071
H. 73.	unaccent .....	3072
H. 74.	uuid-oss .....	3073
H. 75.	wal2sql .....	3075
H. 76.	xml2 .....	3078
H. 77.	zhparser .....	3082
H. 78.	lt_t3sdk .....	3085
H. 79.	lt_http .....	3087
I.	额外提供的程序 .....	3093
I. 1.	客户端应用 .....	3093
I. 2.	服务器应用 .....	3100
J.	外部项目 .....	3104
J. 1.	客户端接口 .....	3104
J. 2.	管理工具 .....	3104
J. 3.	过程语言 .....	3104
J. 4.	扩展 .....	3105
K.	LightDB限制 .....	3106
L.	首字母缩写词 .....	3107
M.	术语表 .....	3113
N.	颜色支持 .....	3124
N. 1.	当使用颜色时 .....	3124
N. 2.	配置颜色 .....	3124
O.	LightDB JDBC .....	3125
P.	扩展的钩子列表 .....	3126
参考书目	.....	3128
Index	.....	3130

---

## List of Figures

56.1. 遗传算法的结构 .....	2269
62.1. GIN内部 .....	2309
64.1. 页面布局 .....	2320

---

## List of Tables

4.1.	反斜线转义序列	33
4.2.	操作符优先级（从高到低）	39
5.1.	ACL 权限缩写	73
5.2.	访问权限摘要	73
8.1.	数据类型	131
8.2.	数字类型	132
8.3.	货币类型	137
8.4.	字符类型	138
8.5.	特殊字符类型	139
8.6.	二进制数据类型	140
8.7.	bytea文字转义字节	140
8.8.	bytea输出转义字节	141
8.9.	日期/时间类型	141
8.10.	日期输入	143
8.11.	时间输入	144
8.12.	时区输入	144
8.13.	特殊日期/时间输入	145
8.14.	日期/时间输出风格	146
8.15.	日期顺序习惯	146
8.16.	ISO 8601 间隔单位缩写	148
8.17.	间隔输入	149
8.18.	间隔输出风格例子	150
8.19.	布尔数据类型	150
8.20.	几何类型	153
8.21.	网络地址类型	155
8.22.	cidr类型输入例子	156
8.23.	JSON 基本类型和相应的LightDB类型	164
8.24.	jsonpath 变量	170
8.25.	jsonpath Accessors	170
8.26.	对象标识符类型	191
8.27.	伪类型	192
9.1.	比较操作符	195
9.2.	比较谓词	195
9.3.	比较函数	198
9.4.	数学操作符	198
9.5.	数学函数	200
9.6.	随机函数	202
9.7.	三角函数	202
9.8.	双曲函数	204
9.9.	SQL字符串函数和操作符	205
9.10.	其他字符串函数	206
9.11.	SQL二进制串函数和操作符	212
9.12.	其他二进制串函数	212
9.13.	Text/Binary String Conversion Functions	214
9.14.	位串操作符	215
9.15.	位字符串函数	215
9.16.	正则表达式匹配操作符	219
9.17.	正则表达式原子	223
9.18.	正则表达式量词	224
9.19.	正则表达式约束	224
9.20.	正则表达式字符项逃逸	226
9.21.	正则表达式类缩写逃逸	227
9.22.	正则表达式约束逃逸	227
9.23.	正则表达式后引用	227
9.24.	ARE 嵌入选项字母	228

9.25.	格式化函数	232
9.26.	用于日期/时间格式化的模板模式	233
9.27.	用于日期/时间格式化的模板模式修饰语	234
9.28.	用于数字格式化的模板模式	237
9.29.	用于数字格式化的模板模式修饰语	238
9.30.	to_char例子	238
9.31.	日期/时间操作符	239
9.32.	日期/时间函数	240
9.33.	AT TIME ZONE变体	249
9.34.	nls_date_format/nls_timestamp_format格式化的模板模式	252
9.35.	枚举支持函数	255
9.36.	几何操作符	255
9.37.	几何函数	259
9.38.	几何类型转换函数	260
9.39.	IP地址操作符	261
9.40.	IP地址的函数	262
9.41.	MAC 地址函数	264
9.42.	文本搜索操作符	264
9.43.	文本搜索函数	265
9.44.	文本搜索调试函数	268
9.45.	json 和 jsonb 操作符	283
9.46.	附加的 jsonb 操作符	284
9.47.	JSON 创建函数	285
9.48.	JSON 处理函数	286
9.49.	jsonpath 操作符和方法	294
9.50.	jsonpath 过滤器表达式元素	296
9.51.	序列函数	298
9.52.	数组操作符	306
9.53.	数组函数	307
9.54.	范围操作符	309
9.55.	范围函数	310
9.56.	通用聚集函数	311
9.57.	用于统计的聚集函数	313
9.58.	有序集聚集函数	315
9.59.	假想集聚集函数	315
9.60.	分组操作	316
9.61.	通用窗口函数	317
9.62.	系列生成函数	323
9.63.	下标生成函数	324
9.64.	会话信息函数	326
9.65.	访问权限查询函数	329
9.66.	aclitem 操作符	330
9.67.	aclitem 函数	330
9.68.	模式可见性查询函数	331
9.69.	系统目录信息函数	332
9.70.	索引列属性	335
9.71.	索引性质	336
9.72.	索引访问方法性质	336
9.73.	对象信息和定位函数	336
9.74.	注释信息函数	337
9.75.	事务ID和快照信息功能	337
9.76.	快照组件	338
9.77.	已弃用的事务ID和快照信息功能	338
9.78.	已提交事务信息函数	339
9.79.	控制数据函数	339
9.80.	pg_control_checkpoint 输出列	339
9.81.	pg_control_system 输出列	340
9.82.	pg_control_init 输出列	340



9.83.	pg_control_recovery 输出列	340
9.84.	配置设置函数	341
9.85.	服务器信号函数	341
9.86.	备份控制函数	342
9.87.	恢复信息函数	344
9.88.	恢复控制函数	344
9.89.	快照同步函数	345
9.90.	复制管理函数	346
9.91.	数据库对象尺寸函数	348
9.92.	数据库对象位置函数	349
9.93.	排序规则管理函数	349
9.94.	分区信息函数	349
9.95.	索引维护函数	350
9.96.	通用文件访问函数	350
9.97.	咨询锁函数	352
9.98.	内置触发器函数	353
9.99.	表重写信息函数	356
12.1.	默认解析器的记号类型	402
13.1.	事务隔离级别	423
13.2.	冲突的锁模式	430
13.3.	冲突的行级锁	431
16.1.	System V IPC参数	471
16.2.	SSL 服务器文件用法	484
17.1.	synchronous_commit Modes	507
17.2.	消息严重级别	530
17.3.	短选项键	560
19.1.	预定义角色	586
21.1.	LightDB Character Sets	601
21.2.	内置客户端/服务器字符集转换	605
21.3.	所有内置字符集转换	606
24.1.	高可用、负载均衡和复制特性矩阵	635
25.1.	动态统计视图	654
25.2.	已收集统计信息的视图	655
25.3.	pg_stat_activity 视图	657
25.4.	等待事件类型	658
25.5.	Activity类型的等待事件	659
25.6.	BufferPin类型的等待事件	659
25.7.	Client类型的等待事件	660
25.8.	Extension类型的等待事件	660
25.9.	IO类型的等待事件	660
25.10.	IPC类型的等待事件	662
25.11.	Lock类型的等待事件	664
25.12.	LWLock类型的等待事件	664
25.13.	Timeout类型的等待事件	666
25.14.	pg_stat_replication 视图	667
25.15.	pg_stat_wal_receiver 视图	669
25.16.	pg_stat_subscription View	670
25.17.	pg_stat_ssl 视图	670
25.18.	pg_stat_gssapi 视图	671
25.19.	pg_stat_archiver 视图	671
25.20.	pg_stat_bgwriter View	672
25.21.	pg_stat_database 视图	672
25.22.	pg_stat_database_conflicts 视图	673
25.23.	pg_stat_all_tables 视图	674
25.24.	pg_stat_all_indexes 视图	675
25.25.	pg_statio_all_tables 视图	676
25.26.	pg_statio_all_indexes 视图	677
25.27.	pg_statio_all_sequences 视图	677

25.28.	pg_stat_user_functions 视图	678
25.29.	pg_stat_slru 视图	678
25.30.	Additional Statistics Functions	679
25.31.	Per-Backend Statistics Functions	680
25.32.	pg_stat_progress_analyze 视图	681
25.33.	ANALYZE Phases	681
25.34.	pg_stat_progress_create_index 视图	682
25.35.	CREATE INDEX 的阶段	683
25.36.	pg_stat_progress_vacuum 视图	683
25.37.	VACUUM的阶段	684
25.38.	pg_stat_progress_cluster 视图	685
25.39.	CLUSTER 和 VACUUM FULL 阶段	685
25.40.	pg_stat_progress_basebackup 视图	686
25.41.	基础备份阶段	686
25.42.	内建 DTrace 探针	687
25.43.	定义用在探针参数中的类型	692
32.1.	SSL 模式描述	787
32.2.	Libpq/客户端 SSL 文件用法	788
33.1.	面向 SQL 的大对象函数	805
34.1.	在 LightDB 数据类型和 C 变量类型之间映射	820
34.2.	PGTYPESdate_from_asc的合法输入格式	838
34.3.	PGTYPESdate_fmt_asc的合法输入格式	839
34.4.	rdefmtdate的合法输入格式	840
34.5.	PGTYPEStimestamp_from_asc的合法输入格式	841
35.1.	information_schema_catalog_name 列	914
35.2.	administrable_role_authorizations 列	914
35.3.	applicable_roles 列	914
35.4.	attributes 列	915
35.5.	character_sets 列	917
35.6.	check_constraint_routine_usage 列	918
35.7.	check_constraints 列	918
35.8.	collations 列	918
35.9.	collation_character_set_applicability 列	919
35.10.	column_column_usage 列	919
35.11.	column_domain_usage 列	920
35.12.	column_options 列	920
35.13.	column_privileges 列	921
35.14.	column_udt_usage 列	921
35.15.	columns 列	922
35.16.	constraint_column_usage 列	924
35.17.	constraint_table_usage 列	925
35.18.	data_type_privileges 列	925
35.19.	domain_constraints 列	926
35.20.	domain_udt_usage 列	926
35.21.	domains 列	927
35.22.	element_types 列	929
35.23.	enabled_roles 列	930
35.24.	foreign_data_wrapper_options 列	931
35.25.	foreign_data_wrappers 列	931
35.26.	foreign_server_options 列	931
35.27.	foreign_servers 列	932
35.28.	foreign_table_options 列	932
35.29.	foreign_tables 列	933
35.30.	key_column_usage 列	933
35.31.	parameters 列	934
35.32.	referential_constraints 列	935
35.33.	role_column_grants 列	936
35.34.	role_routine_grants 列	937

35.35.	role_table_grants 列	937
35.36.	role_udt_grants 列	938
35.37.	role_usage_grants 列	938
35.38.	routine_privileges 列	939
35.39.	routines 列	940
35.40.	schemata 列	944
35.41.	sequences 列	944
35.42.	sql_features 列	945
35.43.	sql_implementation_info 列	945
35.44.	sql_parts 列	946
35.45.	sql_sizing 列	946
35.46.	table_constraints 列	947
35.47.	table_privileges 列	947
35.48.	tables 列	948
35.49.	transforms 列	949
35.50.	triggered_update_columns 列	949
35.51.	triggers 列	950
35.52.	udt_privileges 列	951
35.53.	usage_privileges 列	952
35.54.	user_defined_types 列	952
35.55.	user_mapping_options 列	954
35.56.	user_mappings 列	954
35.57.	view_column_usage 列	955
35.58.	view_routine_usage 列	955
35.59.	view_table_usage 列	956
35.60.	views 列	956
36.1.	Polymorphic Types	964
36.2.	内建 SQL 类型等效的 C 类型	986
36.3.	B-树策略	1018
36.4.	哈希策略	1018
36.5.	GiST 二维“R-树”策略	1019
36.6.	GIN 数组策略	1019
36.7.	B-树支持函数	1020
36.8.	哈希支持函数	1020
36.9.	GIN 支持函数	1020
38.1.	支持事件触发器的命令标签	1049
41.1.	可用的诊断项	1099
41.2.	错误诊断项	1112
42.1.	集合方法	1192
269.	不同的语法兼容模式 LIGHTDB_SYNTAX_COMPATIBLE_TYPE 支持不同的模板:	1450
270.	按命令类型应用的策略	1498
271.	错误代码表	1811
272.	ltbench 自动变量	1966
273.	ltbench操作符	1968
274.	ltbench 函数	1969
48.1.	系统目录	2106
48.2.	lt_attri_ext Columns	2108
48.3.	lt_database_ext Columns	2108
48.4.	lt_global_temp_table_ext Columns	2108
48.5.	lt_package Columns	2109
48.6.	lt_partition_ext Columns	2109
48.7.	lt_proc_ext Columns	2110
48.8.	lt_type_ext Columns	2110
48.9.	lt_variable Columns	2111
48.10.	pg_aggregate Columns	2112
48.11.	pg_am Columns	2113
48.12.	pg_amop Columns	2113
48.13.	pg_amproc Columns	2114

---

48.14.	pg_attrdef Columns	2115
48.15.	pg_attribute Columns	2115
48.16.	pg_authid Columns	2117
48.17.	pg_auth_members Columns	2118
48.18.	pg_cast Columns	2119
48.19.	pg_class Columns	2119
48.20.	pg_collation Columns	2121
48.21.	pg_constraint Columns	2122
48.22.	pg_conversion Columns	2124
48.23.	pg_database Columns	2124
48.24.	pg_db_role_setting Columns	2125
48.25.	pg_default_acl Columns	2126
48.26.	pg_depend Columns	2126
48.27.	pg_description Columns	2128
48.28.	pg_enum Columns	2128
48.29.	pg_event_trigger Columns	2129
48.30.	pg_extension Columns	2129
48.31.	pg_foreign_data_wrapper Columns	2130
48.32.	pg_foreign_server Columns	2130
48.33.	pg_foreign_table Columns	2131
48.34.	pg_index Columns	2131
48.35.	pg_inherits Columns	2133
48.36.	pg_init_privs Columns	2133
48.37.	pg_language Columns	2133
48.38.	pg_largeobject Columns	2134
48.39.	pg_largeobject_metadata Columns	2135
48.40.	pg_namespace Columns	2135
48.41.	pg_opclass Columns	2135
48.42.	pg_operator Columns	2136
48.43.	pg_opfamily Columns	2137
48.44.	pg_partitioned_table Columns	2137
48.45.	pg_policy Columns	2138
48.46.	pg_proc Columns	2139
48.47.	pg_publication Columns	2141
48.48.	pg_publication_rel Columns	2141
48.49.	pg_range Columns	2141
48.50.	pg_replication_origin Columns	2142
48.51.	pg_rewrite Columns	2142
48.52.	pg_seclabel Columns	2143
48.53.	pg_sequence Columns	2143
48.54.	pg_shdepend Columns	2144
48.55.	pg_shdescription Columns	2145
48.56.	pg_shseclabel Columns	2145
48.57.	pg_statistic Columns	2146
48.58.	pg_statistic_ext Columns	2147
48.59.	pg_statistic_ext_data Columns	2148
48.60.	pg_subscription Columns	2148
48.61.	pg_subscription_rel Columns	2149
48.62.	pg_tablespace Columns	2149
48.63.	pg_transform Columns	2150
48.64.	pg_trigger Columns	2150
48.65.	pg_ts_config Columns	2151
48.66.	pg_ts_config_map Columns	2152
48.67.	pg_ts_dict Columns	2152
48.68.	pg_ts_parser Columns	2153
48.69.	pg_ts_template Columns	2153
48.70.	pg_type Columns	2154
48.71.	typcategory Codes	2156

48.72.	pg_user_mapping Columns	2157
48.73.	系统视图	2157
48.74.	pg_available_extensions Columns	2158
48.75.	pg_available_extension_versions Columns	2159
48.76.	pg_config Columns	2159
48.77.	pg_cursors Columns	2160
48.78.	pg_file_settings Columns	2160
48.79.	pg_group Columns	2161
48.80.	pg_hba_file_rules Columns	2161
48.81.	pg_indexes Columns	2162
48.82.	pg_locks Columns	2163
48.83.	pg_matviews Columns	2165
48.84.	pg_policies Columns	2165
48.85.	pg_prepared_statements Columns	2166
48.86.	pg_prepared_xacts Columns	2166
48.87.	pg_publication_tables Columns	2167
48.88.	pg_replication_origin_status Columns	2167
48.89.	pg_replication_slots Columns	2167
48.90.	pg_roles Columns	2168
48.91.	pg_rules Columns	2169
48.92.	pg_seclabels Columns	2170
48.93.	pg_sequences Columns	2170
48.94.	pg_settings Columns	2171
48.95.	pg_shadow Columns	2173
48.96.	pg_shmem_allocations Columns	2173
48.97.	pg_stats Columns	2174
48.98.	pg_stats_ext Columns	2175
48.99.	pg_tables Columns	2176
48.100.	pg_timezone_abbrevs Columns	2176
48.101.	pg_timezone_names Columns	2177
48.102.	pg_user Columns	2177
48.103.	pg_user_mappings Columns	2178
48.104.	pg_views Columns	2178
61.1.	内建GiST操作符类	2293
62.1.	内建GIN操作符类	2306
64.1.	LTDATA的内容	2314
64.2.	页面布局	2319
64.3.	PageHeaderData布局	2319
64.4.	HeapTupleHeaderData布局	2321
A.1.	对象类型	2343
A.2.	数据类型	2343
A.3.	函数	2344
A.4.	SQL语法	2346
A.5.	过程性语言	2347
A.6.	系统视图	2347
A.7.	字符集和字符序	2347
A.8.	数据库导入导出	2348
A.9.	事务一致性与隔离级别	2349
A.10.	优化器	2349
A.11.	优化器提示	2349
A.12.	高可用	2349
A.13.	备份恢复	2349
A.14.	安全特性	2350
A.15.	分区支持	2350
A.16.	并行执行	2350
C.1.	LightDB错误代码	2384
D.1.	月份名称	2395
D.2.	一周内每一天的名称	2395

D. 3. 日期/时间域修饰语 .....	2395
E. 1. SQL Key Words .....	2401
H. 1. adminpack Functions .....	2502
H. 2. 立方体外部表示 .....	2517
H. 3. 立方体操作符 .....	2517
H. 4. 立方体函数 .....	2518
H. 5. hstore 操作符 .....	2557
H. 6. hstore 函数 .....	2558
H. 7. intarray 函数 .....	2564
H. 8. intarray 操作符 .....	2564
H. 9. isn 数据类型 .....	2566
H. 10. isn 函数 .....	2568
H. 11. pg_buffercache 列 .....	2571
H. 12. hint table .....	2596
H. 13. lt_hint_plan的GUC参数 .....	2635
H. 14. 提示列表 .....	2645
H. 15. lt_patch 列 .....	2668
H. 16. servers Columns .....	2673
H. 17. samples Columns .....	2674
H. 18. sample_server_hardware Columns .....	2674
H. 19. sample_cluster_instance Columns .....	2675
H. 20. sample_kernel_settings Columns .....	2676
H. 21. sample_cpu_usage Columns .....	2676
H. 22. sample_db_memory_usage Columns .....	2677
H. 23. sample_settings Columns .....	2677
H. 24. sample_timings Columns .....	2677
H. 25. baselines Columns .....	2678
H. 26. bl_samples Columns .....	2678
H. 27. stmt_list Columns .....	2678
H. 28. sample_stat_database Columns .....	2679
H. 29. sample_statements Columns .....	2680
H. 30. sample_statements_total Columns .....	2682
H. 31. sample_wait_event_total Columns .....	2683
H. 32. tablespaces_list Columns .....	2683
H. 33. sample_stat_tablespaces Columns .....	2684
H. 34. tables_list Columns .....	2684
H. 35. sample_stat_tables Columns .....	2684
H. 36. sample_stat_tables_total Columns .....	2686
H. 37. indexes_list Columns .....	2687
H. 38. samples Columns .....	2688
H. 39. sample_stat_indexes_total Columns .....	2688
H. 40. funcs_list Columns .....	2689
H. 41. sample_stat_user_functions Columns .....	2689
H. 42. samples Columns .....	2690
H. 43. sample_stat_cluster Columns .....	2690
H. 44. sample_stat_archiver Columns .....	2691
H. 45. sample_stat_tables_failures Columns .....	2692
H. 46. sample_stat_indexes_failures Columns .....	2692
H. 47. lt_stat_backend_type_id_text Columns .....	2692
H. 48. lt_stat_cost_type_id_text Columns .....	2693
H. 49. lt_stat_wait_evt_id_text Columns .....	2693
H. 50. lt_stat_wait_evt_sub_id_text Columns .....	2693
H. 51. lt_stat_cost_1s Columns .....	2693
H. 52. lt_stat_cost_60s Columns .....	2694
H. 53. lt_stat_cost_600s Columns .....	2694
H. 54. pg_stat_statements 列 .....	2741
H. 55. lt_tempfile 函数 .....	2746
H. 56. lt_trgm函数 .....	2747

H. 57.	lt_trgm 运算符	2748
H. 58.	crypt() 支持的算法	2760
H. 59.	crypt() 的迭代次数	2761
H. 60.	哈希算法速度	2761
H. 61.	使用和不使用OpenSSL的功能概述	2769
H. 62.	函数	2771
H. 63.	运算符类型兼容性	2771
H. 64.	ltree 操作符	2779
H. 65.	ltree 函数	2780
H. 66.	pgrowlocks 输出列	2784
H. 67.	pgstattuple 输出列	2786
H. 68.	pgstattuple_approx 输出列	2789
H. 69.	数据类型	2790
H. 70.	数据转换	2790
H. 71.	数学函数	2790
H. 72.	字符串函数	2790
H. 73.	日期/时间函数	2791
H. 74.	JSON 函数	2791
H. 75.	条件表达式	2792
H. 76.	聚合函数	2792
H. 77.	系统信息函数和运算符	2792
H. 78.	杂项函数	2792
H. 79.	运算符	2792
H. 80.	算术运算符	2792
H. 81.	比较操作符	2793
H. 82.	date_format格式化的模板模式	2807
H. 83.	from_unixtime 格式化的模板模式	2810
H. 84.	str_to_date 格式化的模板模式	2812
H. 85.	数据类型	2857
H. 86.	SQL 查询	2858
H. 87.	数学函数	2858
H. 88.	字符串函数	2858
H. 89.	Date/Time Functions	2859
H. 90.	数据类型格式化函数	2860
H. 91.	条件表达式	2860
H. 92.	聚合函数	2860
H. 93.	返回内部信息的函数	2861
H. 94.	SQL运算符	2861
H. 95.	其他函数	2861
H. 96.	包	2862
H. 97.	系统视图	2862
H. 98.	数据类型	2867
H. 99.	函数	2867
H. 100.	函数	2867
H. 101.	运算符	2868
H. 102.	VARCHAR2语法	2869
H. 103.	NVARCHAR2语法	2870
H. 104.	日期语法	2871
H. 105.	RAW 语法	2871
H. 106.	PLS_INTEGER 语法	2872
H. 107.	BINARY_FLOAT 语法	2872
H. 108.	BINARY_DOUBLE 语法	2873
H. 109.	XMLType 语法	2873
H. 110.	表(t3)的组成	2890
H. 111.	NEXT_DAY 函数中设置 dayOfWk 的值	2907
H. 112.	ROUND 函数中设置格式模型	2908
H. 113.	DECODE可以转换的数据类型组合 (摘要)	2924
H. 114.	DECODE可转换的结果值和默认值日期/时间数据类型	2925

H. 115.	命名空间USERENV的预定义参数	2939
H. 116.	日期时间操作符	2941
H. 117.	DBMS_ALERT功能	2947
H. 118.	DBMS_ASSERT 特性	2951
H. 119.	DBMS_LOB 功能	2955
H. 120.	DBMS_OUTPUT 特点	2962
H. 121.	管道类型	2966
H. 122.	DBMS_PIPE 特点	2966
H. 123.	NEXT_ITEM_TYPE返回的值	2968
H. 124.	DBMS_RANDOM功能	2976
H. 125.	可指定的显示格式值	2978
H. 126.	DBMS_UTILITY功能	2979
H. 127.	可指定的显示格式值	2980
H. 128.	DBMS_JOB功能	2984
H. 129.	DBMS_LOCK功能	2986
H. 130.	DBMS_METADATA 功能	2987
H. 131.	DBMS_OBFUSCATION_TOOLKIT 功能特性	2987
H. 132.	DBMS_SNAPSHOT 功能特性	2988
H. 133.	UTL_FILE 功能特性	2988
H. 134.	UTL_RAW 特性	2995
H. 135.	UTL_URL Features	3000
H. 136.	UTL_ENCODE 特性	3001
H. 137.	ALL_SEQUENCES	3005
H. 138.	ALL_TAB_COLS	3006
H. 139.	ALL_TAB_COL_STATISTICS	3007
H. 140.	ALL_OBJECTS	3007
H. 141.	ALL_CATALOG	3008
H. 142.	DICTIONARY	3008
H. 143.	ALL_DEPENDENCIES	3008
H. 144.	ALL_SOURCE	3008
H. 145.	ALL_PROCEDURES	3009
H. 146.	ALL_TRIGGERS	3009
H. 147.	ALL_TRIGGER_COLS	3010
H. 148.	ALL_TYPES	3010
H. 149.	ALL_CONSTRAINTS	3011
H. 150.	ALL_CONS_COLUMNS	3012
H. 151.	ALL_VIEWS	3012
H. 152.	ALL_ALL_TABLES	3013
H. 153.	ALL_TAB_STATISTICS	3013
H. 154.	ALL_TAB_COMMENTS	3014
H. 155.	ALL_COL_COMMENTS	3014
H. 156.	ALL_TAB_MODIFICATIONS	3014
H. 157.	ALL_INDEXES	3015
H. 158.	DBA_INDEX_USAGE	3015
H. 159.	ALL_IND_COLUMNS	3016
H. 160.	ALL_IND_EXPRESSIONS	3016
H. 161.	ALL_IND_STATISTICS	3016
H. 162.	DBA_USERS	3017
H. 163.	ALL_USERS	3017
H. 164.	DBA_ROLES	3017
H. 165.	DBA_ROLE_PRIVS	3018
H. 166.	PRODUCT_COMPONENT_VERSION	3018
H. 167.	DBA_DATA_FILES	3018
H. 168.	DBA_JOBS	3019
H. 169.	DBA_JOBS_RUNNING	3020
H. 170.	ALL_MVIEWS	3020
H. 171.	ALL_MVIEW_COMMENTS	3021
H. 172.	DBA_TABLESPACES	3021



---

H. 173.	NLS_DATABASE_PARAMETERS .....	3022
H. 174.	DBA_SEGMENTS .....	3022
H. 175.	ALL_PART_TABLES .....	3022
H. 176.	ALL_TAB_PARTITIONS .....	3023
H. 177.	ALL_TAB_SUBPARTITIONS .....	3024
H. 178.	ALL_PART_KEY_COLUMNS .....	3024
H. 179.	ALL_SUBPART_KEY_COLUMNS .....	3024
H. 180.	ALL_IND_PARTITIONS .....	3025
H. 181.	ALL_PART_INDEXES .....	3025
H. 182.	COL .....	3026
H. 183.	seg外部表达 .....	3045
H. 184.	合法seg输入的例子 .....	3045
H. 185.	Seg GiST 操作符 .....	3046
H. 186.	Sepgsql 函数 .....	3052
H. 187.	tablefunc函数 .....	3060
H. 188.	connectby 参数 .....	3066
H. 189.	用于 UUID 产生的函数 .....	3074
H. 190.	返回 UUID 常量的函数 .....	3074
H. 191.	xml2 函数 .....	3079
H. 192.	xpath_table 参数 .....	3080
J. 1.	外部维护的客户端接口 .....	3104
J. 2.	外部维护的过程语言 .....	3105
K. 1.	LightDB限制 .....	3106
P. 1.	钩子列表 .....	3126

---

## List of Examples

8.1.	使用字符类型	139
8.2.	使用boolean类型	151
8.3.	使用位串类型	158
9.1.	转换 SQL/XML 输出到 HTML 的 XSLT 样式表	282
10.1.	平方根运算符类型解析	360
10.2.	字符串连接操作符类型决定	360
10.3.	绝对值与否定操作符类型决定	361
10.4.	数组包含操作符类型决定	361
10.5.	域类型上的自定义操作符	362
10.6.	圆整函数参数类型决定	364
10.7.	可变函数决定	364
10.8.	子串函数类型决定	365
10.9.	添加月份函数类型解析	365
10.10.	character存储类型转换	367
10.11.	联合中未指定类型的类型决定	367
10.12.	简单联合中的类型决定	368
10.13.	可换位联合中的类型决定	368
10.14.	嵌套合并中的类型决定	368
10.15.	修改或添加类型优先级	369
10.16.	自动应用显式转换	370
10.17.	类型优先级	370
10.18.	子字符串函数类型解析	370
10.19.	参数列表优先级	370
10.20.	插入和更新	371
11.1.	建立一个部分索引来排除公值	377
11.2.	建立一个部分索引来排除不感兴趣的值	378
11.3.	建立一个部分唯一索引	379
11.4.	不要使用部分索引代替分区	379
18.1.	示例 lt_hba.conf 项	568
18.2.	一个示例 lt_ident.conf 文件	570
32.1.	libpq 例子程序 1	791
32.2.	libpq例子程序 2	793
32.3.	libpq例子程序 3	796
33.1.	用libpq操作大对象的例子程序	806
34.1.	示例 SQLDA 程序	856
34.2.	访问大对象的 ECPG 程序	869
41.1.	在动态查询中引用值	1096
41.2.	UPDATE/INSERT的异常	1111
41.3.	一个 PL/pgSQL 触发器函数	1124
41.4.	一个用于审计的 PL/pgSQL 触发器函数	1125
41.5.	一个用于审计的 PL/pgSQL 视图触发器函数	1126
41.6.	一个 PL/pgSQL 用于维护汇总表的触发器函数	1127
41.7.	用传递表进行审计	1129
41.8.	一个 PL/pgSQL 事件触发器函数	1130
41.9.	从PL/SQL移植一个简单的函数到PL/pgSQL	1138
41.10.	从PL/SQL移植一个创建另一个函数的函数到PL/pgSQL	1139
41.11.	从PL/SQL移植一个带有字符串操作以及OUT参数的过程到PL/pgSQL	1140
41.12.	从PL/SQL移植一个过程到PL/pgSQL	1142
H.1.	为 LightDB CSV 日志创建一个外部表	2552

---

# 前言

本书是LightDB的官方文档。它是LightDB开发人员和其它志愿者并行编写到LightDB的开发中的。它描述了当前版本的LightDB官方支持的所有功能。

为了能够管理有关LightDB的大量信息，本书被组织成了几个部分。每个部分都是针对不同层次的用户，或者说针对具有不同阶段LightDB经验的用户：

- Part I 是一个给新用户的非正式介绍。
- Part II 记载了SQL查询语言环境，包括数据类型和函数，以及用户级别的性能调优。每个LightDB用户都应该阅读这些内容。
- Part III 描述服务器的安装和管理。每个运行LightDB服务器的人，不管是个人使用还是为别人维护，都应该阅读这部分内容。
- Part IV 描述LightDB客户端程序的编程接口。
- Part V 包含为高级用户准备的信息，比如服务器的扩展能力等。其中的内容包括用户定义数据类型和函数等。
- Part VI 包含有关SQL命令、客户端和服务端程序的参考信息。这部分以按照命令或程序排序的结构化信息支持其它部分。
- Part VII 包含可用于LightDB开发人员的各类信息。
- Part VIII 本章节包含了LightDB数据库对于已兼容支持的Oracle和MySQL的功能特性清单列表。

## 1. 何为LightDB?

LightDB是以加州大学伯克利分校计算机系开发的POSTGRES，版本 4.2<sup>1</sup>为基础的对象关系型数据库管理系统（ORDBMS）。POSTGRES 领先的许多概念在很久以后才出现在一些商业数据库系统中。

LightDB是最初的伯克利代码的开源继承者。它支持大部分SQL标准并且提供了许多现代特性：

- 复杂查询
- 外键
- 触发器
- 可更新视图
- 事务完整性
- 多版本并发控制

同样，LightDB可以用许多方法扩展，比如，通过增加新的：

- 数据类型
- 函数
- 操作符
- 聚集函数
- 索引方法
- 过程语言

并且，因为自由宽松的许可证，任何人都可以以任何目的免费使用、修改和分发LightDB，不管是私用、商用还是学术研究目的。

---

<sup>1</sup> <https://dsf.berkeley.edu/postgres.html>

## 2. 约定

下面的约定被用于命令的大纲：方括弧（[和]）表示可选的部分（在 Tcl 命令里，使用的是问号（?），就像通常的 Tcl 一样）。花括弧（{和}）和竖线（|）表示你必须选取一个候选。点（...）表示它前面的元素可以被重复。

如果能提高清晰度，那么 SQL 命令前面会放上提示符=>，而 shell 命令前面会放上提示符\$。不过，提示符通常不被显示。

一个管理员通常是一个负责安装和运行服务器的人员。一个用户可以是任何使用或者需要使用LightDB系统的任何部分的人员。我们不应该对这些术语的概念理解得太狭隘；这份文档集在系统管理过程方面没有固定的假设。

## 3. 进一步的信息

除了本文档（即本书），还有其他关于LightDB的资源：

维基

LightDB wiki<sup>2</sup>包含项目的FAQ<sup>3</sup>（常见问题）列表、TODO<sup>4</sup>列表以及更多主题的详细信息。

网站

LightDB网站<sup>5</sup>中有最新发布的信息，以及其他让你能够更高效使用LightDB的信息。

邮件列表

邮件列表是一个提问的好地方，也可以与其他用户分享经验，还可以用来联系开发者。详情请参考LightDB网站。

你自己！

LightDB是一个开源项目。就其本身而言，它依赖于用户社区提供不间断的支持。如果你刚开始使用LightDB，你将需要从其他人那里得到帮助，或者通过文档或者通过邮件列表。请考虑把你的知识贡献出来回馈社区。阅读邮件列表并且回答问题。如果你学到某些不在文档中的东西，请把它写下来并且贡献出来。如果你为代码增加了特性，也请把它们贡献出来。

---

<sup>2</sup> <https://wiki.postgresql.org>

<sup>3</sup> [https://wiki.postgresql.org/wiki/Frequently\\_Asked\\_Questions](https://wiki.postgresql.org/wiki/Frequently_Asked_Questions)

<sup>4</sup> <https://wiki.postgresql.org/wiki/ToDo>

<sup>5</sup> [www.hs.net/lightdb](http://www.hs.net/lightdb)

---

# Part I. 教程

欢迎来到LightDB教程。下面的几章将为那些新接触LightDB、关系数据库概念和 SQL 语言的读者给出一个简单介绍。我们只假定读者拥有关于如何使用计算机的一般知识。读者不需要特殊的 Unix 或编程经验。这一部分主要希望给你一些关于LightDB系统的重要方面的手把手的体验。我们并不准备把它写成一份能覆盖这些主题所有内容的文档。

当你完成了这份教程之后，你可能希望继续去阅读Part II来得到 SQL 语言更正式的知识，或者阅读Part III来了解如何开发LightDB应用程序。希望搭建并且管理自己的服务器的用户还应该阅读Part III

---

# Table of Contents

1. 从头开始 .....	3
1.1. 安装 .....	3
1.2. 架构基础 .....	3
1.3. 创建一个数据库 .....	3
1.4. 访问数据库 .....	5
2. SQL语言 .....	7
2.1. 引言 .....	7
2.2. 概念 .....	7
2.3. 创建一个新表 .....	7
2.4. 在表中增加行 .....	8
2.5. 查询一个表 .....	9
2.6. 在表之间连接 .....	10
2.7. 聚集函数 .....	12
2.8. 更新 .....	14
2.9. 删除 .....	14
3. 高级特性 .....	16
3.1. 简介 .....	16
3.2. 视图 .....	16
3.3. 外键 .....	16
3.4. 事务 .....	17
3.5. 窗口函数 .....	18
3.6. 继承 .....	21
3.7. 小结 .....	22

---

# Chapter 1. 从头开始

## 1.1. 安装

自然，在你能开始使用LightDB之前，你必须安装它。LightDB很有可能已经安装到你的节点上了，因为它可能包含在你的操作系统的发布里，或者是系统管理员已经安装了它。如果是这样的话，那么你应该从操作系统的文档或者你的系统管理员那里获取有关如何访问LightDB的信息。

如果你不清楚LightDB是否已经安装，或者不知道你能否用它（已经安装的）做自己的实验，那么你就可以自己安装。这么做并不难，并且是一次很好的练习。LightDB可以由任何非特权用户安装，并不需要超级用户（root）的权限。

如果你的站点管理员没有按照缺省的方式设置各项相关参数，那你还有点额外的活儿要干。比如，如果数据库服务器机器是一个远程的机器，那你就需要把LTHOST环境变量设置为数据库服务器的名字。环境变量LTPORT也可能需要设置。总而言之就是：如果你试着启动一个应用而该应用报告说不能与数据库建立联接时，你应该马上与你的数据库管理员联系，如果你就是管理员，那么你就要参考文档以确保你的环境变量得到正确的设置。如果你不理解随后的几段，那么先阅读下一节。

## 1.2. 架构基础

在我们继续之前，你应该先了解LightDB的系统架构。对LightDB的部件之间如何相互作用的理解将会使本节更易理解。

在数据库术语里，LightDB使用一种客户端/服务器的模型。一次LightDB会话由下列相关的进程（程序）组成：

- 一个服务器进程，它管理数据库文件、接受来自客户端应用与数据库的联接并且代表客户端在数据库上执行操作。该数据库服务器程序叫做lightdb。
- 那些需要执行数据库操作的用户的客户端（前端）应用。客户端应用可能本身就是多种多样的：可以是一个面向文本的工具，也可以是一个图形界面的应用，或者是一个通过访问数据库来显示网页的网页服务器，或者是一个特制的数据库管理工具。一些客户端应用是和LightDB发布一起提供的，但绝大部分是用户开发的。

和典型的客户端/服务器应用（C/S应用）一样，这些客户端和服务器可以在不同的主机上。这时它们通过TCP/IP网络联接通讯。你应该记住的是，在客户机上可以访问的文件未必能够在数据库服务器机器上访问（或者只能用不同的文件名进行访问）。

LightDB服务器可以处理来自客户端的多个并发请求。因此，它为每个连接启动（“forks”）一个新的进程。从这个时候开始，客户端和新服务器进程就不再经过最初的postgres进程的干涉进行通讯。因此，守护者服务器进程总是在运行并等待着客户端联接，而客户端和相关联的服务器进程则是起起停停（当然，这些对用户是透明的。我们介绍这些主要是为了内容的完整性）。

## 1.3. 创建一个数据库

看看你能否访问数据库服务器的第一个例子就是试着创建一个数据库。一台运行着的LightDB服务器可以管理许多数据库。通常我们会为每个项目和每个用户单独使用一个数据库。

你的站点管理员可能已经为你创建了可以使用的数据库。如果这样你就可以省略这一步，并且跳到下一节。

要创建一个新的数据库，在我们这个例子里叫mydb，你可以使用下面的命令：

```
$ createdb mydb
```

如果不产生任何响应则表示该步骤成功，你可以跳过本节的剩余部分。

如果你看到类似下面这样的信息：

```
createdb: command not found
```

那么就是LightDB没有安装好。或者是根本没安装，或者是你的shell搜索路径没有设置正确。尝试用绝对路径调用该命令试试：

```
$ /usr/local/lightdb-x/bin/createdb mydb
```

在你的站点上这个路径可能不一样。和你的站点管理员联系或者看看安装指导获取正确的位置。

另外一种响应可能是这样：

```
createdb: could not connect to database postgres: could not connect to server:
No such file or directory
Is the server running locally and accepting
connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

这意味着该服务器没有启动，或者在createdb期望去连接它的时候没有在监听。同样，你也要查看安装指导或者咨询管理员。

另外一个响应可能是这样：

```
createdb: could not connect to database postgres: FATAL: role "joe" does not
exist
```

在这里提到了你自己的登录名。如果管理员没有为你创建LightDB用户帐号，就会发生这些现象。（LightDB用户帐号和操作系统用户帐号是不同的。）如果你是管理员，参阅Chapter 19获取创建用户帐号的帮助。你需要变成安装LightDB的操作系统用户的身份（通常是 lightdb）才能创建第一个用户帐号。也有可能是赋予你的LightDB用户名和你的操作系统用户名不同；这种情况下，你需要使用-U选项或者使用LTUSER环境变量指定你的LightDB用户名。

如果你有个数据库用户帐号，但是没有创建数据库所需要的权限，那么你会看到下面的信息：

```
createdb: database creation failed: ERROR: permission denied to create database
```

并非所有用户都被许可创建新数据库。如果LightDB拒绝为你创建数据库，那么你需要让站点管理员赋予你创建数据库的权限。出现这种情况时请咨询你的站点管理员。如果你自己安装了LightDB，那么你应该以你启动数据库服务器的用户身份登录后参考手册完成权限的赋予工作。<sup>1</sup>

你还可以用其它名字创建数据库。LightDB允许你在一个站点上创建任意数量的数据库。数据库名必须是以字母开头并且小于 63 个字符长。一个方便的做法是创建和你当前用户名同

---

<sup>1</sup> 为什么这么做的解释：LightDB用户名是和操作系统用户账号分开的。如果你连接到一个数据库时，你可以选择以何种LightDB用户名进行联接；如果你不选择，那么缺省就是你的当前操作系统账号。如果这样，那么总有一个与操作系统用户同名的LightDB用户账号用于启动服务器，并且通常这个用户都有创建数据库的权限。如果你不想以该用户身份登录，那么你也可以在任何地方声明一个-U选项以选择一个用于连接的LightDB用户名。



名的数据库。许多工具假设该数据库名为缺省数据库名，所以这样可以节省你的敲键。要创建这样的数据库，只需要键入：

```
$ createdb
```

如果你想创建一个分布式数据库mydb，那么你就可以用lt\_createdb命令创建它：

```
$ lt_createdb mydb
```

对于这条命令而言，你必须在cn节点上执行，非cn节点上创建的还是普通数据库。

如果你再也不想使用你的数据库了，那么你可以删除它。比如，如果你是数据库mydb的所有人（创建人），那么你就可以用下面的命令删除它：

```
$ dropdb mydb
```

（对于这条命令而言，数据库名不是缺省的用户名，因此你就必须声明它）。这个动作将在物理上把所有与该数据库相关的文件都删除并且不可取消，因此做这中操作之前一定要考虑清楚。

更多关于createdb和dropdb的信息可以分别在createdb和dropdb中找到。

## 1.4. 访问数据库

一旦你创建了数据库，你就可以通过以下方式访问它：

- 运行LightDB的交互式终端程序，它被称为ltsql，它允许你交互地输入、编辑和执行SQL命令。
- 使用一种已有的图形化前端工具，比如pgAdmin或者带ODBC或JDBC支持的办公套件来创建和管理数据库。这种方法在这份教程中没有介绍。
- 使用多种绑定发行的语言中的一种写一个自定义的应用。这些可能性在Part IV中将有更深入的讨论。

你可能需要启动ltsql来试验本教程中的例子。你可以用下面的命令为mydb数据库激活它：

```
$ ltsql mydb
```

如果你不提供数据库名字，那么它的缺省值就是你的用户账号名字。在前面使用createdb的小节里你应该已经了解了这种方式。

在ltsql中，你将看到下面的欢迎信息：

```
ltsql (13.8-23.4)
compatible type: postgresql
Type "help" for help.
```

```
mydb=>
```

最后一行也可能是：

```
mydb=#
```

这个提示符意味着你是数据库超级用户，最可能出现在你自己安装了LightDB实例的情况下。作为超级用户意味着你不受访问控制的限制。对于本教程的目的而言，是否超级用户并不重要。

如果你启动ltsql时碰到了问题，那么请回到前面的小节。诊断createdb的方法和诊断ltsql的方法很类似，如果前者能运行那么后者也应该能运行。

ltsql打印出的最后一行是提示符，它表示ltsql正听着你说话，这个时候你就可以敲入 SQL 查询到一个ltsql维护的工作区中。试验一下下面的命令：

```
mydb=> SELECT version();
                version
-----
LightDB 13.8-23.4 on x86_64-pc-linux-gnu, compiled by gcc (Debian 4.9.2-10)
4.9.2, 64-bit
(1 row)

mydb=> SELECT current_date;
        date
-----
2016-01-07
(1 row)

mydb=> SELECT 2 + 2;
?column?
-----
         4
(1 row)
```

ltsql程序有一些不属于SQL命令的内部命令。它们以反斜线开头，“\”。欢迎信息中列出了一些这种命令。比如，你可以用下面的命令获取各种LightDB的SQL命令的帮助语法：

```
mydb=> \h
```

要退出ltsql，输入：

```
mydb=> \q
```

ltsql将会退出并且让你返回到命令行shell。（要获取更多有关内部命令的信息，你可以在ltsql提示符上键入\?。）ltsql的完整功能在ltsql中有文档说明。在这份文档里，我们将不会明确使用这些特性，但是你自己可以在需要的时候使用它们。

---

# Chapter 2. SQL语言

## 2.1. 引言

本章提供一个如何使用SQL执行简单操作的概述。本教程的目的只是给你一个介绍，并非完整的SQL教程。有许多关于SQL的书籍，包括[melt93]和[date97]。你还要知道有些LightDB语言特性是对标准的扩展。

在随后的例子里，我们假设你已经创建了名为mydb的数据库，就象在前面的章里面介绍的一样，并且已经能够启动ltsql。

本手册的例子也可以在LightDB源代码的目录src/tutorial/中找到（二进制LightDB发布中可能没有提供这些文件）。要使用这些文件，先进入该目录然后运行make:

```
$ cd .../src/tutorial
$ make
```

这样就创建了那些脚本并编译了包含用户定义函数和类型的 C 文件。接下来，要开始本教程，按照下面说的做:

```
$ ltsql -s mydb
```

```
...
```

```
mydb=> \i basics.sql
```

\i命令从指定的文件中读取命令。ltsql的-s选项把你置于单步模式，它在向服务器发送每个语句之前暂停。在本节使用的命令都在文件basics.sql中。

## 2.2. 概念

LightDB是一种关系型数据库管理系统（RDBMS）。这意味着它是一种用于管理存储在关系中的数据系统。关系实际上是表的数学术语。今天，把数据存储在表里的概念已经快成了固有的常识了，但是还有其它的一些方法用于组织数据库。在类 Unix 操作系统上的文件和目录就形成了一种层次数据库的例子。更现代的发展是面向对象数据库。

每个表都是一个命名的行集合。一个给定表的每一行由同一组的命名列组成，而且每一列都有一个特定的数据类型。虽然列在每行里的顺序是固定的，但一定要记住 SQL 并不对行在表中的顺序做任何保证（但你可以为了显示的目的对它们进行显式地排序）。

表被分组成数据库，一个由单个LightDB服务器实例管理的数据库集合组成一个数据库簇。

## 2.3. 创建一个新表

你可以通过指定表的名字和所有列的名字及其类型来创建表：

```
CREATE TABLE weather (
    city          varchar(80),
    temp_lo      int,          -- 最低温度
    temp_hi      int,          -- 最高温度
    prcp         real,        -- 湿度
    date         date
);
```

你可以在`ltsql`输入这些命令以及换行符。`ltsql`可以识别该命令直到分号才结束。

你可以在 SQL 命令中自由使用空白（即空格、制表符和换行符）。这就意味着你可以用和上面不同的对齐方式键入命令，或者将命令全部放在一行中。两个划线（“`--`”）引入注释。任何跟在它后面直到行尾的东西都会被忽略。SQL 是对关键字和标识符大小写不敏感的语言，只有在标识符用双引号包围时才能保留它们的大小写（上例没有这么做）。

`varchar(80)` 指定了一个可以存储最长 80 个字符的任意字符串的数据类型。`int` 是普通的整数类型。`real` 是一种用于存储单精度浮点数的类型。`date` 类型应该可以自解释（没错，类型为 `date` 的列名字也是 `date`。这么做可能比较方便或者容易让人混淆——你自己选择）。

LightDB 支持标准的 SQL 类型 `int`、`smallint`、`real`、`double precision`、`char(N)`、`varchar(N)`、`date`、`time`、`timestamp` 和 `interval`，还支持其他的通用功能的类型和丰富的几何类型。LightDB 中可以定制任意数量的用户定义数据类型。因而类型名并不是语法关键字，除了 SQL 标准要求支持的特例外。

第二个例子将保存城市和它们相关的地理位置：

```
CREATE TABLE cities (  
    name          varchar(80),  
    location      point  
);
```

类型 `point` 就是一种 LightDB 特有数据类型的例子。

最后，我们还要提到如果你不再需要某个表，或者你想以不同的形式重建它，那么你可以用下面的命令删除它：

```
DROP TABLE tablename;
```

## 2.4. 在表中增加行

`INSERT` 语句用于向表中添加行：

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

请注意所有数据类型都使用了相当明了的输入格式。那些不是简单数字值的常量通常必需用单引号（`'`）包围，就象在例子里一样。`date` 类型实际上对可接收的格式相当灵活，不过在本教程里，我们应该坚持使用这种清晰的格式。

`point` 类型要求一个坐标对作为输入，如下：

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

到目前为止使用的语法要求你记住列的顺序。一个可选的语法允许你明确地列出列：

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)  
VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

如果你需要，你可以用另外一个顺序列出列或者是忽略某些列，比如说，我们不知道降水量：

```
INSERT INTO weather (date, city, temp_hi, temp_lo)  
VALUES ('1994-11-29', 'Hayward', 54, 37);
```

许多开发人员认为明确列出列要比依赖隐含的顺序是更好的风格。

请输入上面显示的所有命令，这样你在随后的各节中才有可用的数据。

你还可以使用COPY从文本文件中装载大量数据。这种方式通常更快，因为COPY命令就是为这类应用优化的，只是比INSERT少一些灵活性。比如：

```
COPY weather FROM '/home/user/weather.txt';
```

这里源文件的文件名必须在运行后端进程的机器上是可用的，而不是在客户端上，因为后端进程将直接读取该文件。你可以在COPY中读到更多有关COPY命令的信息。

## 2.5. 查询一个表

要从一个表中检索数据就是查询这个表。SQL的SELECT语句就是做这个用途的。该语句分为选择列表（列出要返回的列）、表列表（列出从中检索数据的表）以及可选的条件（指定任意的限制）。比如，要检索表weather的所有行，键入：

```
SELECT * FROM weather;
```

这里\*是“所有列”的缩写。<sup>1</sup>因此相同的结果应该这样获得：

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

而输出应该是：

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0	1994-11-29
Hayward	37	54		1994-11-29

(3 rows)

你可以在选择列表中写任意表达式，而不仅仅是列的列表。比如，你可以：

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

这样应该得到：

city	temp_avg	date
San Francisco	48	1994-11-27
San Francisco	50	1994-11-29
Hayward	45	1994-11-29

(3 rows)

请注意这里的AS子句是如何给输出列重新命名的（AS子句是可选的）。

一个查询可以使用WHERE子句“修饰”，它指定需要哪些行。WHERE子句包含一个布尔（真值）表达式，只有那些使布尔表达式为真的行才会被返回。在条件中可以使用常用的布尔操作符（AND、OR和NOT）。比如，下面的查询检索旧金山的下雨天的天气：

<sup>1</sup> 虽然SELECT \*对于即席查询很有用，但我们普遍认为在生产代码中这是很糟糕的风格，因为给表增加一个列就改变了结果。

```
SELECT * FROM weather
  WHERE city = 'San Francisco' AND prcp > 0.0;
```

结果:

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27

(1 row)

你可以要求返回的查询结果是排好序的:

```
SELECT * FROM weather
  ORDER BY city;
```

city	temp_lo	temp_hi	prcp	date
Hayward	37	54		1994-11-29
San Francisco	43	57	0	1994-11-29
San Francisco	46	50	0.25	1994-11-27

在这个例子里, 排序的顺序并未完全被指定, 因此你可能看到属于旧金山的行被随机地排序。但是如果你使用下面的语句, 那么就总是会得到上面的结果:

```
SELECT * FROM weather
  ORDER BY city, temp_lo;
```

你可以要求在查询的结果中消除重复的行:

```
SELECT DISTINCT city
  FROM weather;
```

city
Hayward
San Francisco

(2 rows)

再次声明, 结果行的顺序可能变化。你可以组合使用DISTINCT和ORDER BY来保证获取一致的结果:<sup>2</sup>

```
SELECT DISTINCT city
  FROM weather
  ORDER BY city;
```

## 2.6. 在表之间连接

到目前为止, 我们的查询一次只访问一个表。查询可以一次访问多个表, 或者用这种方式访问一个表而同时处理该表的多个行。一次同时访问多张表(或者多个实例的同一张表)的查询叫join查询。它们将一个表中的行与另一个表中的行结合起来, 用一个表达式来指定哪

<sup>2</sup> 在一些数据库系统里, 包括老版本的LightDB, DISTINCT的实现自动对行进行排序, 因此ORDER BY是多余的。但是这一点并不是SQL 标准的要求, 并且目前的LightDB并不保证DISTINCT会导致行被排序。

些行将被配对。例如，返回所有天气记录以及相关的城市位置。数据库需要拿 weather表每行的city列和cities表所有行的name列进行比较，并选取那些在该值上相匹配的行。<sup>3</sup> 这个任务可以用下面的查询来实现：

```
SELECT *
  FROM weather, cities
 WHERE city = name;
```

city	temp_lo	temp_hi	prcp	date	name
San Francisco (-194, 53)	46	50	0.25	1994-11-27	San Francisco
San Francisco (-194, 53)	43	57	0	1994-11-29	San Francisco

(2 rows)

观察结果集的两个方面：

- 没有城市Hayward的结果行。这是因为在cities表里面没有Hayward的匹配行，所以连接忽略 weather表里的不匹配行。我们稍后将看到如何修补它。
- 有两个列包含城市名字。这是正确的，因为weather和cities表的列被串接在一起。不过，实际上我们不想要这些，因此你将可能希望明确列出输出列而不是使用\*：

```
SELECT city, temp_lo, temp_hi, prcp, date, location
  FROM weather, cities
 WHERE city = name;
```

练习：. 尝试确定在省略WHERE子句时，此查询的语义。

因为这些列的名字都不一样，所以规划器自动地找出它们属于哪个表。如果在两个表里有重名的列，你需要限定列名来说明你究竟想要哪一个，如：

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
  FROM weather, cities
 WHERE cities.name = weather.city;
```

人们广泛认为在一个连接查询中限定所有列名是一种好的风格，这样即使未来向其中一个表里添加重名列也不会导致查询失败。

到目前为止，这种类型的连接查询也可以用下面这样的形式写出来：

```
SELECT *
  FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

虽然这种语法不像上面那种常用，但我们在这里展示它，以帮助理解以下主题。

现在我们将看看如何能把Hayward记录找回来。我们想让查询干的事是扫描weather表，并且对每一行都找出匹配的cities表行。如果我们没有找到匹配的行，那么我们需要一些“空值”代替cities表的列。这种类型的查询叫外连接（我们在此之前看到的连接都是inner joins）。这样的命令看起来象这样：

<sup>3</sup> 这里只是一个概念上的模型。连接通常以比实际比较每个可能的行对更高效的方式执行，但这是用户看不到的。

```
SELECT *
  FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);
```

city location	temp_lo	temp_hi	prcp	date	name
Hayward	37	54		1994-11-29	
San Francisco (-194, 53)	46	50	0.25	1994-11-27	San Francisco
San Francisco (-194, 53)	43	57	0	1994-11-29	San Francisco

(3 rows)

这个查询是一个左外连接，因为在连接操作符左部的表中的行在输出中至少要出现一次，而在右部的表的行只有在能找到匹配的左部表行时才被输出。如果输出的左部表的行没有对应匹配的右部表的行，那么右部表行的列将填充空值（null）。

练习：. 还有右外连接和全外连接。试着找出来它们能干什么。

我们也可以把一个表和自己连接起来。这叫做自连接。比如，假设我们想找出那些在其它天气记录的温度范围之外的天气记录。这样我们就需要拿 weather表里每行的temp\_lo和temp\_hi列与weather表里其它行的temp\_lo和temp\_hi列进行比较。我们可以用下面的查询实现这个目标：

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
       W2.city, W2.temp_lo AS low, W2.temp_hi AS high
  FROM weather W1, weather W2
 WHERE W1.temp_lo < W2.temp_lo
       AND W1.temp_hi > W2.temp_hi;
```

city	low	high	city	low	high
San Francisco	43	57	San Francisco	46	50
Hayward	37	54	San Francisco	46	50

(2 rows)

在这里我们把weather表重新标记为w1和w2以区分连接的左部和右部。你还可以用这样的别名在其它查询里节约一些敲键，比如：

```
SELECT *
  FROM weather w, cities c
 WHERE w.city = c.name;
```

你以后会经常碰到这样的缩写的。

## 2.7. 聚集函数

和大多数其它关系数据库产品一样，LightDB支持聚集函数。一个聚集函数从多个输入行中计算出一个结果。比如，我们有在一个行集合上计算count（计数）、sum（和）、avg（均值）、max（最大值）和min（最小值）的函数。

比如，我们可以用下面的语句找出所有记录中最低温度中的最高温度：

```
SELECT max(temp_lo) FROM weather;
```



```

max
-----
46
(1 row)

```

如果我们想知道该读数发生在哪个城市，我们可以用：

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);    WRONG
```

不过这个方法不能运转，因为聚集max不能被用于WHERE子句中（存在这个限制是因为WHERE子句决定哪些行可以被聚集计算包括；因此显然它必需在聚集函数之前被计算）。不过，我们通常都可以用其它方法实现我们的目的；这里我们就可以使用子查询：

```
SELECT city FROM weather
   WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

```

      city
-----
San Francisco
(1 row)

```

这样做是 OK 的，因为子查询是一次独立的计算，它独立于外层的查询计算出自己的聚集。

聚集同样也常用于和GROUP BY子句组合。比如，我们可以获取每个城市观测到的最低温度的最高值：

```
SELECT city, max(temp_lo)
   FROM weather
   GROUP BY city;
```

```

      city      | max
-----+-----
Hayward         | 37
San Francisco  | 46
(2 rows)

```

这样给我们每个城市一个输出。每个聚集结果都是在匹配该城市的表行上面计算的。我们可以用HAVING 过滤这些被分组的行：

```
SELECT city, max(temp_lo)
   FROM weather
   GROUP BY city
   HAVING max(temp_lo) < 40;
```

```

      city      | max
-----+-----
Hayward         | 37
(1 row)

```

这样就只给出那些所有temp\_lo值曾都低于 40的城市。最后，如果我们只关心那些名字以“S”开头的城市，我们可以用：

```
SELECT city, max(temp_lo)
FROM weather
WHERE city LIKE 'S%'           -- ❶
GROUP BY city
HAVING max(temp_lo) < 40;
```

❶ LIKE操作符进行模式匹配，在Section 9. 里有解释。

理解聚集和SQL的WHERE以及HAVING子句之间的关系对我们非常重要。WHERE和HAVING的基本区别如下：WHERE在分组和聚集计算之前选取输入行（因此，它控制哪些行进入聚集计算），而HAVING在分组和聚集之后选取分组行。因此，WHERE子句不能包含聚集函数；因为试图用聚集函数判断哪些行应输入给聚集运算是没有意义的。相反，HAVING子句总是包含聚集函数（严格说来，你可以写不使用聚集的HAVING子句，但这样做很少有用。同样的条件用在WHERE阶段会更有效）。

在前面的例子里，我们可以在WHERE里应用城市名称限制，因为它不需要聚集。这样比放在HAVING里更加高效，因为可以避免那些未通过 WHERE检查的行参与到分组和聚集计算中。

## 2.8. 更新

你可以用UPDATE命令更新现有的行。假设你发现所有 11 月 28 日以后的温度读数都低了两度，那么你就可以用下面的方式改正数据：

```
UPDATE weather
SET temp_hi = temp_hi - 2, temp_lo = temp_lo - 2
WHERE date > '1994-11-28';
```

看看数据的新状态：

```
SELECT * FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0	1994-11-29
Hayward	35	52		1994-11-29

(3 rows)

## 2.9. 删除

数据行可以用DELETE命令从表中删除。假设你对Hayward的天气不再感兴趣，那么你可以用下面的方法把那些行从表中删除：

```
DELETE FROM weather WHERE city = 'Hayward';
```

所有属于Hayward的天气记录都被删除。

```
SELECT * FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0	1994-11-29

(2 rows)

我们用下面形式的语句的时候一定要小心

```
DELETE FROM tablename;
```

如果没有一个限制，DELETE将从指定表中删除所有行，把它清空。做这些之前系统不会请求你确认！

---

## Chapter 3. 高级特性

### 3.1. 简介

在之前的章节里我们已经涉及了使用SQL在LightDB中存储和访问数据的基础知识。现在我们将要讨论SQL中一些更高级的特性，这些特性有助于简化管理和防止数据丢失或损坏。最后，我们还将介绍一些LightDB扩展。

本章有时将引用Chapter 2中的例子并对其进行改变或改进以便于阅读本章。本章中的某些例子可以在教程目录的advanced.sql文件中找到。该文件也包含一些样例数据，在这里就不在赘述（查看Section 2.1了解如何使用该文件）。

### 3.2. 视图

回想一下Section 2.6中的查询。假设天气记录和城市位置的组合列表对我们的应用有用，但我们又不想每次需要使用它时都敲入整个查询。我们可以在该查询上创建一个视图，这会为该查询一个名字，我们可以像使用一个普通表一样来使用它：

```
CREATE VIEW myview AS
  SELECT name, temp_lo, temp_hi, prcp, date, location
     FROM weather, cities
     WHERE city = name;
```

```
SELECT * FROM myview;
```

对视图的使用是成就一个好的SQL数据库设计的关键方面。视图允许用户通过始终如一接口封装表的结构细节，这样可以避免表结构随着应用的进化而改变。

视图几乎可以用在任何可以使用表的地方。在其他视图基础上创建视图也并不少见。

### 3.3. 外键

回想第2章中的weather和cities表。考虑以下问题：我们希望确保在cities表中有相应项之前任何人都不能在weather表中插入行。这叫做维持数据的引用完整性。在过分简化的数据库系统中，可以通过先检查cities表中是否有匹配的记录存在，然后决定应该接受还是拒绝即将插入weather表的行。这种方法有一些问题且并不方便，于是LightDB可以为我们来解决：

新的表定义如下：

```
CREATE TABLE cities (
  name      varchar(80) primary key,
  location  point
);

CREATE TABLE weather (
  city      varchar(80) references cities(name),
  temp_lo   int,
  temp_hi   int,
  prcp      real,
  date      date
);
```

现在尝试插入一个非法的记录：

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ERROR: insert or update on table "weather" violates foreign key constraint
"weather_city_fkey"
DETAIL: Key (city)=(Berkeley) is not present in table "cities".
```

外键的行为可以很好地根据应用来调整。我们不会在这个教程里更深入地介绍，读者可以参考Chapter 5中的信息。正确使用外键无疑会提高数据库应用的质量，因此强烈建议用户学会如何使用它们。

## 3.4. 事务

事务是所有数据库系统的基础概念。事务最重要的一点是它将多个步骤捆绑成了一个单一的、要么全完成要么全不完成的操作。步骤之间的中间状态对于其他并发事务是不可见的，并且如果有某些错误发生导致事务不能完成，则其中任何一个步骤都不会对数据库造成影响。

例如，考虑一个保存着多个客户账户余额和支行总存款额的银行数据库。假设我们希望记录一笔从Alice的账户到Bob的账户的额度为100.00美元的转账。在最大程度地简化后，涉及到的SQL命令是：

```
UPDATE accounts SET balance = balance - 100.00
  WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
  WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
  WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
  WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

这些命令的细节在这里并不重要，关键点是为了完成这个相当简单的操作涉及到多个独立的更新。我们的银行职员希望确保这些更新要么全部发生，或者全部不发生。当然不能发生因为系统错误导致Bob收到100美元而Alice并未被扣款的情况。Alice当然也不希望自己被扣款而Bob没有收到钱。我们需要一种保障，当操作中途某些错误发生时已经执行的步骤不会产生效果。将这些更新组织成一个事务就可以给我们这种保障。一个事务被称为是原子的：从其他事务的角度来看，它要么整个发生要么完全不发生。

我们同样希望能保证一旦一个事务被数据库系统完成并认可，它就被永久地记录下来且即便其后发生崩溃也不会被丢失。例如，如果我们正在记录Bob的一次现金提款，我们当然不希望他刚走出银行大门，对他账户的扣款就消失。一个事务型数据库保证一个事务在被报告为完成之前它所做的所有更新都被记录在持久存储（即磁盘）。

事务型数据库的另一个重要性质与原子更新的概念紧密相关：当多个事务并发运行时，每一个都不能看到其他事务未完成的修改。例如，如果一个事务正忙着总计所有支行的余额，它不会只包括Alice的支行的扣款而不包括Bob的支行的存款，或者反之。所以事务的全做或全不做并不只体现在它们对数据库的持久影响，也体现在它们发生时的可见性。一个事务所做的更新在它完成之前对于其他事务是不可见的，而之后所有的更新将同时变得可见。

在LightDB中，开启一个事务需要将SQL命令用BEGIN和COMMIT命令包围起来。因此我们的银行事务看起来会是这样：

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
  WHERE name = 'Alice';
-- etc etc
```

COMMIT;

如果，在事务执行中我们并不想提交（或许是我们注意到Alice的余额不足），我们可以发出ROLLBACK命令而不是COMMIT命令，这样所有目前的更新将会被取消。

LightDB实际上将每一个SQL语句都作为一个事务来执行。如果我们没有发出BEGIN命令，则每个独立的语句都会被加上一个隐式的BEGIN以及（如果成功）COMMIT来包围它。一组被BEGIN和COMMIT包围的语句也被称为一个事务块。

### Note

某些客户端库会自动发出BEGIN和COMMIT命令，因此我们可能会在不被告知情的情况下得到事务块的效果。具体请查看所使用的接口文档。

也可以利用保存点来以更细的粒度来控制一个事务中的语句。保存点允许我们有选择性地放弃事务的一部分而提交剩下的部分。在使用SAVEPOINT定义一个保存点后，我们可以在必要时利用ROLLBACK TO回滚到该保存点。该事务中位于保存点和回滚点之间的数据库修改都会被放弃，但是早于该保存点的修改则会被保存。

在回滚到保存点之后，它的定义依然存在，因此我们可以多次回滚到它。反过来，如果确定不再需要回滚到特定的保存点，它可以被释放以便系统释放一些资源。记住不管是释放保存点还是回滚到保存点都会释放定义在该保存点之后的所有其他保存点。

所有这些都发生在一个事务块内，因此这些对于其他数据库会话都不可见。当提交整个事务块时，被提交的动作将作为一个单元变得对其他会话可见，而被回滚的动作则永远不会变得可见。

记住那个银行数据库，假设我们从Alice的账户扣款100美元，然后存款到Bob的账户，结果直到最后才发现我们应该存到Wally的账户。我们可以通过使用保存点来做这件事：

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
  WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
  WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
  WHERE name = 'Wally';
COMMIT;
```

当然，这个例子是被过度简化的，但是在一个事务块中使用保存点存在很多种控制可能性。此外，ROLLBACK TO是唯一的途径来重新控制一个由于错误被系统置为中断状态的事务块，而不是完全回滚它并重新启动。

## 3.5. 窗口函数

一个窗口函数在一系列与当前行有某种关联的表行上执行一种计算。这与一个聚集函数所完成的计算有可比之处。但是窗口函数并不会使多行被聚集成一个单独的输出行，这与通常的非窗口聚集函数不同。取而代之，行保留它们独立的标识。在这些现象背后，窗口函数可以访问的不仅仅是查询结果的当前行。

下面是一个例子用于展示如何将每一个员工的薪水与他/她所在部门的平均薪水进行比较：

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM
empsalary;
```

depname	empno	salary	avg
develop	11	5200	5020.0000000000000000
develop	7	4200	5020.0000000000000000
develop	9	4500	5020.0000000000000000
develop	8	6000	5020.0000000000000000
develop	10	5200	5020.0000000000000000
personnel	5	3500	3700.0000000000000000
personnel	2	3900	3700.0000000000000000
sales	3	4800	4866.6666666666666667
sales	1	5000	4866.6666666666666667
sales	4	4800	4866.6666666666666667

(10 rows)

最开始的三个输出列直接来自于表empsalary，并且表中每一行都有一个输出行。第四列表示对与当前行具有相同depname值的所有表行取得平均值（这实际和非窗口avg聚集函数是相同的函数，但是OVER子句使得它被当做一个窗口函数处理并在一个合适的窗口帧上计算。）。

一个窗口函数调用总是包含一个直接跟在窗口函数名及其参数之后的OVER子句。这使得它从句法上和一个普通函数或非窗口函数区分开来。OVER子句决定究竟查询中的哪些行被分离出来由窗口函数处理。OVER子句中的PARTITION BY子句指定了将具有相同PARTITION BY表达式值的行分到组或者分区。对于每一行，窗口函数都会在当前行同一分区的行上进行计算。

我们可以通过OVER上的ORDER BY控制窗口函数处理行的顺序（窗口的ORDER BY并不一定要符合行输出的顺序。）。下面是一个例子：

```
SELECT depname, empno, salary,
rank() OVER (PARTITION BY depname ORDER BY salary DESC) FROM empsalary;
```

depname	empno	salary	rank
develop	8	6000	1
develop	10	5200	2
develop	11	5200	2
develop	9	4500	4
develop	7	4200	5
personnel	2	3900	1
personnel	5	3500	2
sales	1	5000	1
sales	4	4800	2
sales	3	4800	2

(10 rows)

如上所示，rank函数在当前行的分区内按照ORDER BY子句的顺序为每一个可区分的ORDER BY值产生了一个数字等级。rank不需要显式的参数，因为它的行为完全决定于OVER子句。

一个窗口函数所考虑的行属于那些通过查询的FROM子句产生并通过WHERE、GROUP BY、HAVING过滤的“虚拟表”。例如，一个由于不满足WHERE条件被删除的行是不会被任何窗口函数所见的。在一个查询中可以包含多个窗口函数，每个窗口函数都可以用不同的OVER子句来按不同方式划分数据，但是它们都作用在由虚拟表定义的同一个人行集上。

我们已经看到如果行的顺序不重要时ORDER BY可以忽略。PARTITION BY同样也可以被忽略，在这种情况下会产生一个包含所有行的分区。

这里有一个与窗口函数相关的重要概念：对于每一行，在它的分区中的行集被称为它的窗口帧。一些窗口函数只作用在窗口帧中的行上，而不是整个分区。默认情况下，如果使用ORDER BY，则帧包括从分区开始到当前行的所有行，以及后续任何与当前行在ORDER BY子句上相等的行。如果ORDER BY被忽略，则默认帧包含整个分区中所有的行。<sup>1</sup> 下面是使用sum的例子：

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

salary	sum
5200	47100
5000	47100
3500	47100
4800	47100
3900	47100
4200	47100
4500	47100
4800	47100
6000	47100
5200	47100

(10 rows)

如上所示，由于在OVER子句中没有ORDER BY，窗口帧和分区一样，而如果缺少PARTITION BY则和整个表一样。换句话说，每个合计都会在整个表上进行，这样我们为每一个输出行得到的都是相同的结果。但是如果我们加上一个ORDER BY子句，我们会得到非常不同的结果：

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

salary	sum
3500	3500
3900	7400
4200	11600
4500	16100
4800	25700
4800	25700
5000	30700
5200	41100
5200	41100
6000	47100

(10 rows)

这里的合计是从第一个（最低的）薪水一直到当前行，包括任何与当前行相同的行（注意相同薪水行的结果）。

窗口函数只允许出现在查询的SELECT列表和ORDER BY子句中。它们不允许出现在其他地方，例如GROUP BY、HAVING和WHERE子句中。这是因为窗口函数的执行逻辑是在处理完这些子句之后。另外，窗口函数在非窗口聚集函数之后执行。这意味着可以在窗口函数的参数中包括一个聚集函数，但反过来不行。

如果需要在窗口计算执行后进行过滤或者分组，我们可以使用子查询。例如：

```
SELECT depname, empno, salary, enroll_date
```

<sup>1</sup> 还有些选项用于以其他方式定义窗口帧，但是这不包括在本教程内。详见Section 4.2.8



```

FROM
  (SELECT depname, empno, salary, enroll_date,
         rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
   FROM empsalary
  ) AS ss
WHERE pos < 3;

```

上述查询仅仅显示了内层查询中rank低于3的结果。

当一个查询涉及到多个窗口函数时，可以将每一个分别写在一个独立的OVER子句中。但如果多个函数要求同一个窗口行为时，这种做法是冗余的而且容易出错的。替代方案是，每一个窗口行为可以被放在一个命名的WINDOW子句中，然后在OVER中引用它。例如：

```

SELECT sum(salary) OVER w, avg(salary) OVER w
FROM empsalary
WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);

```

关于窗口函数的更多细节可以在Section 4.2.8 Section 9.22 Section 7.2.5以及SELECT参考页中找到。

## 3.6. 继承

继承是面向对象数据库中的概念。它展示了数据库设计的新的可能性。

让我们创建两个表：表cities和表capitals。自然地，首都也是城市，所以我们需要有某种方式能够在列举所有城市的时候也隐式地包含首都。如果真的聪明，我们会设计如下的模式：

```

CREATE TABLE capitals (
  name      text,
  population real,
  elevation int,    -- (in ft)
  state     char(2)
);

CREATE TABLE non_capitals (
  name      text,
  population real,
  elevation int    -- (in ft)
);

CREATE VIEW cities AS
  SELECT name, population, elevation FROM capitals
  UNION
  SELECT name, population, elevation FROM non_capitals;

```

这个模式对于查询而言工作正常，但是当我们需要更新一些行时它就变得不好用了。

更好的方案是：

```

CREATE TABLE cities (
  name      text,
  population real,
  elevation int    -- (in ft)
);

```

```
CREATE TABLE capitals (
  state      char(2) UNIQUE NOT NULL
) INHERITS (cities);
```

在这种情况下，一个capitals的行从它的父亲cities继承了所有列（name、population和elevation）。列name的类型是text，一种用于变长字符串的本地LightDB类型。capitals表有一个附加列，state，用于显示他们的州缩写。在LightDB中，一个表可以从0个或者多个表继承。

例如，如下查询可以寻找所有海拔500尺以上的城市名称，包括州首都：

```
SELECT name, elevation
FROM cities
WHERE elevation > 500;
```

返回为：

name	elevation
Las Vegas	2174
Mariposa	1953
Madison	845

(3 rows)

在另一方面，下面的查询可以查找所有海拔高于500尺且不是州首府的城市：

```
SELECT name, elevation
FROM ONLY cities
WHERE elevation > 500;
```

name	elevation
Las Vegas	2174
Mariposa	1953

(2 rows)

其中cities之前的ONLY用于指示查询只在cities表上进行而不会涉及到继承层次中位于cities之下的其他表。很多我们已经讨论过的命令 — SELECT、UPDATE 和DELETE — 都支持这个ONLY记号。

### Note

尽管继承很有用，但是它还未与唯一约束或外键集成，这也限制了它的可用性。更多详情见Section 5.10

## 3.7. 小结

LightDB中有很多特性在这个面向SQL新用户的教程中并未触及。有关这些特性的更多详情将在本书的后续部分进行讨论。

如果需要更多介绍材料，请访问 LightDB 官方网站<sup>2</sup>来获得更多资源链接。

<sup>2</sup> <https://www.lightdb.org>

---

## Part II. SQL 语言

这部份描述在LightDB中SQL语言的使用。我们从描述SQL的一般语法开始，然后解释如何创建保存数据的结构、如何填充数据库以及如何查询它。中间的部分列出了在SQL命令中可用的数据类型和函数。剩余的部分则留给对于调优数据性能的重要方面。

这部份的信息被组织成让一个新用户可以从头到尾跟随它来全面理解主题，而不需要多次参考后面的内容。这些章都是自包含的，这样高级用户可以根据他们的选择阅读单独的章。这一部分的信息被以一种叙事的风格展现。需要查看一个特定命令的完整描述的读者应该去看看Part VI

这一部分的阅读者应该知道如何连接到一个LightDB数据库并且发出SQL命令。我们鼓励不熟悉这些问题的读者先去阅读Part I SQL通常使用LightDB的交互式终端ltsql输入，但是其他具有相似功能的程序也可以被使用。

---

---

# Table of Contents

4. SQL语法	31
4.1. 词法结构	31
4.1.1. 标识符和关键词	31
4.1.2. 常量	33
4.1.3. 操作符	37
4.1.4. 特殊字符	38
4.1.5. 注释	38
4.1.6. 操作符优先级	38
4.2. 值表达式	39
4.2.1. 列引用	40
4.2.2. 位置参数	40
4.2.3. 下标	41
4.2.4. 域选择	41
4.2.5. 操作符调用	42
4.2.6. 函数调用	42
4.2.7. 聚集表达式	42
4.2.8. 窗口函数调用	44
4.2.9. 类型转换	46
4.2.10. 排序规则表达式	47
4.2.11. 标量子查询	48
4.2.12. 数组构造器	48
4.2.13. 行构造器	50
4.2.14. 用户变量	51
4.2.15. 表达式计算规则	53
4.3. 调用函数	54
4.3.1. 使用位置记号	54
4.3.2. 使用命名记号	55
4.3.3. 使用混合记号	55
5. 数据定义	57
5.1. 表基础	57
5.2. 默认值	58
5.3. 生成列	59
5.4. 约束	60
5.4.1. 检查约束	60
5.4.2. 非空约束	62
5.4.3. 唯一约束	63
5.4.4. 主键	64
5.4.5. 外键	64
5.4.6. 排他约束	67
5.5. 系统列	67
5.6. 修改表	68
5.6.1. 增加列	68
5.6.2. 移除列	69
5.6.3. 增加约束	69
5.6.4. 移除约束	69
5.6.5. 更改列的默认值	70
5.6.6. 修改列的数据类型	70
5.6.7. 重命名列	70
5.6.8. 重命名表	70
5.7. 权限	70
5.8. 行安全性策略	74
5.9. 模式	80
5.9.1. 创建模式	80
5.9.2. 公共模式	81
5.9.3. 模式搜索路径	81

5.9.4.	模式和权限	83
5.9.5.	系统目录模式	83
5.9.6.	使用方式	83
5.9.7.	可移植性	84
5.10.	继承	84
5.10.1.	警告	87
5.11.	表分区	87
5.11.1.	概述	87
5.11.2.	声明式划分	88
5.11.3.	使用继承的分区	92
5.11.4.	分区剪枝	96
5.11.5.	分区和约束排除	97
5.11.6.	声明分区最佳实践	98
5.12.	外部数据	98
5.13.	其他数据库对象	99
5.14.	依赖跟踪	99
5.15.	LightDB DDL 规格检查	100
6.	数据操纵	101
6.1.	插入数据	101
6.2.	更新数据	102
6.3.	删除数据	103
6.4.	从修改的行中返回数据	103
6.5.	LightDB DML规格检查	104
7.	查询	105
7.1.	概述	105
7.2.	表表达式	105
7.2.1.	FROM子句	105
7.2.2.	WHERE子句	113
7.2.3.	GROUP BY和HAVING子句	114
7.2.4.	GROUPING SETS、CUBE和ROLLUP	117
7.2.5.	窗口函数处理	119
7.3.	选择列表	119
7.3.1.	选择列表项	119
7.3.2.	列标签	120
7.3.3.	DISTINCT	120
7.4.	组合查询	121
7.5.	行排序	122
7.6.	LIMIT和OFFSET	123
7.7.	VALUES列表	123
7.8.	WITH查询（公共表表达式）	124
7.8.1.	WITH中的SELECT	124
7.8.2.	WITH中的数据修改语句	128
7.9.	ROWNUM	130
8.	数据类型	131
8.1.	数字类型	132
8.1.1.	整数类型	133
8.1.2.	任意精度数字	133
8.1.3.	浮点类型	135
8.1.4.	序数类型	136
8.2.	货币类型	137
8.3.	字符类型	138
8.4.	二进制数据类型	140
8.4.1.	bytea的十六进制格式	140
8.4.2.	bytea的转义格式	140
8.5.	日期/时间类型	141
8.5.1.	日期/时间输入	142
8.5.2.	日期/时间输出	146
8.5.3.	时区	147

8.5.4.	间隔输入	148
8.5.5.	间隔输出	150
8.6.	布尔类型	150
8.7.	枚举类型	151
8.7.1.	枚举类型的声明	151
8.7.2.	排序	152
8.7.3.	类型安全性	152
8.7.4.	实现细节	153
8.8.	几何类型	153
8.8.1.	点	153
8.8.2.	线	153
8.8.3.	线段	154
8.8.4.	方框	154
8.8.5.	路径	154
8.8.6.	多边形	155
8.8.7.	圆	155
8.9.	网络地址类型	155
8.9.1.	inet	155
8.9.2.	cidr	156
8.9.3.	inet vs. cidr	156
8.9.4.	macaddr	156
8.9.5.	macaddr8	157
8.10.	位串类型	157
8.11.	文本搜索类型	158
8.11.1.	tsvector	158
8.11.2.	tsquery	159
8.12.	UUID类型	161
8.13.	XML类型	161
8.13.1.	创建XML值	161
8.13.2.	编码处理	162
8.13.3.	访问XML值	163
8.14.	JSON 类型	163
8.14.1.	JSON 输入和输出语法	164
8.14.2.	设计 JSON 文档	165
8.14.3.	jsonb 包含和存在	165
8.14.4.	jsonb 索引	167
8.14.5.	jsonpath Type	169
8.15.	数组	171
8.15.1.	数组类型的定义	171
8.15.2.	数组值输入	171
8.15.3.	访问数组	173
8.15.4.	修改数组	175
8.15.5.	在数组中搜索	177
8.15.6.	数组输入和输出语法	179
8.16.	组合类型	179
8.16.1.	组合类型的声明	180
8.16.2.	构造组合值	180
8.16.3.	访问组合类型	181
8.16.4.	修改组合类型	182
8.16.5.	在查询中使用组合类型	182
8.16.6.	组合类型输入和输出语法	184
8.17.	范围类型	185
8.17.1.	内建范围类型和多范围类型	185
8.17.2.	例子	186
8.17.3.	包含和排除边界	186
8.17.4.	无限（无界）范围	186
8.17.5.	范围输入/输出	186
8.17.6.	构造范围和多范围	187

8. 17. 7.	离散范围类型	188
8. 17. 8.	定义新的范围类型	188
8. 17. 9.	索引	189
8. 17. 10.	范围上的约束	189
8. 18.	域类型	190
8. 19.	对象标识符类型	190
8. 20.	pg_lsn 类型	192
8. 21.	伪类型	192
9.	函数和操作符	194
9. 1.	逻辑操作符	194
9. 2.	比较函数和操作符	194
9. 3.	数学函数和操作符	198
9. 4.	字符串函数和操作符	204
9. 4. 1.	format	210
9. 5.	二进制串函数和操作符	212
9. 6.	位串函数和操作符	214
9. 7.	模式匹配	216
9. 7. 1.	LIKE	217
9. 7. 2.	SIMILAR TO正则表达式	218
9. 7. 3.	POSIX正则表达式	219
9. 8.	数据类型格式化函数	232
9. 9.	时间/日期函数和操作符	239
9. 9. 1.	EXTRACT, date_part	244
9. 9. 2.	date_trunc	248
9. 9. 3.	AT TIME ZONE	249
9. 9. 4.	当前日期/时间	250
9. 9. 5.	nls_date_format/nls_timestamp_format	252
9. 9. 6.	延时执行	254
9. 10.	枚举支持函数	255
9. 11.	几何函数和操作符	255
9. 12.	网络地址函数和操作符	261
9. 13.	文本搜索函数和操作符	264
9. 14.	UUID 函数	269
9. 15.	XML 函数	269
9. 15. 1.	产生 XML 内容	269
9. 15. 2.	XML 谓词	274
9. 15. 3.	处理 XML	275
9. 15. 4.	将表映射到 XML	279
9. 16.	JSON 函数和操作符	283
9. 16. 1.	处理和创建JSON数据	283
9. 16. 2.	SQL/JSON 路径语言	291
9. 17.	序列操作函数	297
9. 18.	条件表达式	299
9. 18. 1.	CASE	300
9. 18. 2.	COALESCE	301
9. 18. 3.	NULLIF	302
9. 18. 4.	GREATEST和LEAST	302
9. 18. 5.	NVL 和 NVL2	302
9. 18. 6.	DECODE	304
9. 19.	数组函数和操作符	306
9. 20.	范围函数和运算符	309
9. 21.	聚集函数	310
9. 22.	窗口函数	316
9. 23.	子查询表达式	318
9. 23. 1.	EXISTS	318
9. 23. 2.	IN	318
9. 23. 3.	NOT IN	319
9. 23. 4.	ANY/SOME	319

9.23.5.	ALL	320
9.23.6.	单一行比较	320
9.24.	行和数组比较	320
9.24.1.	IN	320
9.24.2.	NOT IN	321
9.24.3.	ANY/SOME (array)	321
9.24.4.	ALL (array)	321
9.24.5.	行构造器比较	322
9.24.6.	组合类型比较	322
9.25.	集合返回函数	323
9.26.	系统信息函数和运算符	326
9.26.1.	会话信息函数	326
9.26.2.	访问权限查询函数	329
9.26.3.	aclitem Operators	330
9.26.4.	架构可见性查询函数	331
9.26.5.	系统目录信息函数	332
9.26.6.	索引列属性	335
9.26.7.	索引性质	336
9.26.8.	索引访问方法性质	336
9.26.9.	对象信息和寻址函数	336
9.26.10.	注释信息函数	337
9.26.11.	事务 ID 和快照信息函数	337
9.26.12.	控制文件数据函数	339
9.27.	系统管理函数	341
9.27.1.	配置设定函数	341
9.27.2.	服务器信号函数	341
9.27.3.	备份控制函数	342
9.27.4.	恢复控制函数	344
9.27.5.	快照同步函数	345
9.27.6.	复制管理函数	345
9.27.7.	数据库对象管理函数	348
9.27.8.	索引维护函数	350
9.27.9.	通用文件访问函数	350
9.27.10.	咨询锁函数	352
9.28.	触发器函数	353
9.29.	事件触发器函数	354
9.29.1.	在命令结束处捕捉更改	354
9.29.2.	处理被 DDL 命令删除的对象	354
9.29.3.	处理表重写事件	356
9.30.	统计信息函数	356
9.30.1.	检查 MCV 列表	356
10.	类型转换	358
10.1.	概述	358
10.2.	操作符	359
10.3.	函数	362
10.4.	值存储	366
10.5.	UNION、CASE 和相关结构	367
10.6.	SELECT 的输出列	368
10.7.	类型兼容性	369
11.	索引	372
11.1.	简介	372
11.2.	索引类型	373
11.3.	多列索引	374
11.4.	索引和 ORDER BY	375
11.5.	组合多个索引	375
11.6.	唯一索引	376
11.7.	表达式索引	376
11.8.	部分索引	377



11.9.	只用索引的扫描和覆盖索引 .....	380
11.10.	操作符类和操作符族 .....	382
11.11.	索引和排序规则 .....	383
11.12.	检查索引使用 .....	384
12.	全文搜索 .....	385
12.1.	介绍 .....	385
12.1.1.	什么是一个文档? .....	385
12.1.2.	基本文本匹配 .....	386
12.1.3.	配置 .....	388
12.2.	表和索引 .....	388
12.2.1.	搜索一个表 .....	389
12.2.2.	创建索引 .....	389
12.3.	控制文本搜索 .....	390
12.3.1.	解析文档 .....	390
12.3.2.	解析查询 .....	391
12.3.3.	排名搜索结果 .....	394
12.3.4.	加亮结果 .....	396
12.4.	额外特性 .....	397
12.4.1.	操纵文档 .....	397
12.4.2.	操纵查询 .....	398
12.4.3.	用于自动更新的触发器 .....	400
12.4.4.	收集文档统计数据 .....	401
12.5.	解析器 .....	402
12.6.	词典 .....	404
12.6.1.	停用词 .....	405
12.6.2.	简单词典 .....	405
12.6.3.	同义词词典 .....	406
12.6.4.	分类词典 .....	408
12.6.5.	Ispell 词典 .....	410
12.6.6.	Snowball 词典 .....	412
12.7.	配置例子 .....	412
12.8.	测试和调试文本搜索 .....	414
12.8.1.	配置测试 .....	414
12.8.2.	解析器测试 .....	416
12.8.3.	词典测试 .....	417
12.9.	GIN 和 GiST 索引类型 .....	418
12.10.	ltsql支持 .....	418
12.11.	限制 .....	421
13.	并发控制 .....	423
13.1.	介绍 .....	423
13.2.	事务隔离 .....	423
13.2.1.	读已提交隔离级别 .....	424
13.2.2.	可重复读隔离级别 .....	425
13.2.3.	可序列化隔离级别 .....	426
13.3.	显式锁定 .....	428
13.3.1.	表级锁 .....	428
13.3.2.	行级锁 .....	430
13.3.3.	页级锁 .....	431
13.3.4.	死锁 .....	431
13.3.5.	咨询锁 .....	432
13.4.	应用级别的数据完整性检查 .....	432
13.4.1.	用可序列化事务来强制一致性 .....	433
13.4.2.	使用显式锁定强制一致性 .....	433
13.5.	提醒 .....	434
13.6.	锁定和索引 .....	434
14.	性能提示 .....	435
14.1.	使用EXPLAIN .....	435
14.1.1.	EXPLAIN基础 .....	435

---

14.1.2.	EXPLAIN ANALYZE .....	441
14.1.3.	警告 .....	445
14.2.	规划器使用的统计信息 .....	446
14.2.1.	单列统计信息 .....	446
14.2.2.	扩展统计信息 .....	448
14.3.	用显式JOIN子句控制规划器 .....	450
14.4.	填充一个数据库 .....	452
14.4.1.	禁用自动提交 .....	452
14.4.2.	使用COPY .....	452
14.4.3.	移除索引 .....	452
14.4.4.	移除外键约束 .....	453
14.4.5.	增加maintenance_work_mem .....	453
14.4.6.	增加max_wal_size .....	453
14.4.7.	禁用 WAL 归档和流复制 .....	453
14.4.8.	事后运行ANALYZE .....	453
14.4.9.	关于lt_dump的一些注记 .....	453
14.5.	非持久设置 .....	454
15.	并行查询 .....	455
15.1.	并行查询如何工作 .....	455
15.2.	何时会用到并行查询? .....	456
15.3.	并行计划 .....	456
15.3.1.	并行扫描 .....	457
15.3.2.	并行连接 .....	457
15.3.3.	并行聚集 .....	457
15.3.4.	并行Append .....	458
15.3.5.	并行计划小贴士 .....	458
15.4.	并行安全性 .....	458
15.4.1.	为函数和聚集加并行标签 .....	459

---

## Chapter 4. SQL语法

这一章描述了SQL的语法。它构成了理解后续具体介绍如何使用SQL定义和修改数据的章节的基础。

我们同时建议已经熟悉SQL的用户仔细阅读本章，因为本章包含一些在SQL数据库中实现得不一致的以及LightDB中特有的规则和概念。

### 4.1. 词法结构

SQL输入由一个命令序列组成。一个命令由一个记号的序列构成，并由一个分号（“;”）终结。输入流的末端也会标志一个命令的结束。具体哪些记号是合法的与具体命令的语法有关。

一个记号可以是一个关键词、一个标识符、一个带引号的标识符、一个literal（或常量）或者一个特殊字符符号。记号通常以空白（空格、制表符、新行）来分隔，但在无歧义时并不强制要求如此（唯一的例子是一个特殊字符紧挨着其他记号）。

例如，下面是一个（语法上）合法的SQL输入：

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

这是一个由三个命令组成的序列，每一行一个命令（尽管这不是必须地，在同一行中可以有超过一个命令，而且命令还可以被跨行分割）。

另外，注释也可以出现在SQL输入中。它们不是记号，它们和空白完全一样。

根据标识命令、操作符、参数的记号不同，SQL的语法不很一致。最前面的一些记号通常是命令名，因此在上面的例子中我们通常会说一个“SELECT”、一个“UPDATE”和一个“INSERT”命令。但是例如UPDATE命令总是要求一个SET记号出现在一个特定位置，而INSERT则要求一个VALUES来完成命令。每个命令的精确语法规则在Part V中介绍。

为了兼容Oracle数据库的使用习惯，LightDB支持双引号结合大写的列名来操作小写形式定义的列名。这个功能只在Oracle兼容模式下生效。

```
create table t1(id int);
insert into t1("ID") values(10), (20), (30);
select "ID" from t1;
update t1 set "ID"=25 where id = 20;
delete from t1 where "ID" = 25;
```

#### 4.1.1. 标识符和关键词

上例中的SELECT、UPDATE或VALUES记号是关键词的例子，即SQL语言中具有特定意义的词。记号MY\_TABLE和A则是标识符的例子。它们标识表、列或者其他数据库对象的名字，取决于使用它们的命令。因此它们有时也被简称为“名字”。关键词和标识符具有相同的词法结构，这意味着我们无法在没有语言知识的前提下区分一个标识符和关键词。一个关键词的完整列表可以在Appendix 中找到。

SQL标识符和关键词必须以一个字母（a-z，也可以是带变音符号的字母和非拉丁字母）或一个下划线（\_）开始。后续字符可以是字母、下划线（\_）、数字（0-9）或美元符号（\$）。注意根据SQL标准的字母规定，美元符号是不允许出现在标识符中的，因此它们的使用可能会降低应用的移植性。SQL标准不会定义包含数字或者以下划线开头或结尾的关键词，因此这种形式的标识符不会与未来可能的标准扩展冲突。

系统中一个标识符的长度不能超过 `NAMEDATALEN-1` 字节，在命令中可以写超过此长度的标识符，但是它们会被截断。默认情况下，`NAMEDATALEN` 的值为64，因此标识符的长度上限为63字节。如果这个限制有问题，可以在`src/include/pg_config_manual.h`中修改`NAMEDATALEN` 常量。

关键词和不被引号修饰的标识符是大小写不敏感的。因此：

```
UPDATE MY_TABLE SET A = 5;
```

可以等价地写成：

```
uPDaTE my_Table SeT a = 5;
```

一个常见的习惯是将关键词写成大写，而名称写成小写，例如：

```
UPDATE my_table SET a = 5;
```

这里还有第二种形式的标识符：受限标识符或被引号修饰的标识符。它是由双引号（`"`）包围的一个任意字符序列。一个受限标识符总是一个标识符而不会是一个关键字。因此`"select"`可以用于引用一个名为`"select"`的列或者表，而一个没有引号修饰的`select`则会被当作一个关键词，从而在本应使用表或列名的地方引起解析错误。在上例中使用受限标识符的例子如下：

```
UPDATE "my_table" SET "a" = 5;
```

受限标识符可以包含任何字符，除了代码为0的字符（如果要包含一个双引号，则写两个双引号）。这使得可以构建原本不被允许的表或列的名称，例如包含空格或花号的名字。但是长度限制依然有效。

引用标识符也使其区分大小写，而未引用的名称总是折叠成小写。例如，标识符`F00`、`foo`和`"foo"`在LightDB中被认为是相同的，但是`"Foo"`和`"F00"`与这三个不同，并且彼此不同。（在LightDB中，将不带引号的名称折叠为小写与SQL标准不兼容，SQL标准规定不带引号的名称应折叠为大写。因此，根据标准，`foo`应等同于`"F00"`而不是`"foo"`。如果您想编写可移植应用程序，建议您始终引用某个特定的名称，或者永远不要引用它。）

一种受限标识符的变体允许包括转义的用代码点标识的Unicode字符。这种变体以`U&`（大写或小写`U`跟上一个花号）开始，后面紧跟双引号修饰的名称，两者之间没有任何空白，如`U&"foo"`（注意这里与操作符`&`似乎有一些混淆，但是在`&`操作符周围使用空白避免了这个问题）。在引号内，Unicode字符可以以转义的形式指定：反斜线接上4位16进制代码点号码或者反斜线和加号接上6位16进制代码点号码。例如，标识符`"data"`可以写成：

```
U&"d\0061t\+000061"
```

下面的例子用斯拉夫语字母写出了俄语单词“slon”（大象）：

```
U&"\0441\043B\043E\043D"
```

如果希望使用其他转义字符来代替反斜线，可以在字符串后使用`UESCAPE`子句，例如：

```
U&"d!0061t!+000061" UESCAPE '!'
```

转义字符可以是除了16进制位、加号、单引号、双引号、空白字符之外的任意单个字符。请注意，转义字符在`UESCAPE`之后用单引号而不是双引号书写。

为了在标识符中包括转义字符本身，将其写两次即可。

4位或6位转义形式都可以被用来定义UTF-16代理对来组成代码点大于U+FFFF的字符，尽管6位形式的存在使得这种做法变得不必要（代理对并不被直接存储，而是绑定成一个单独的代码点）。

如果服务器编码不是UTF-8，则由其中一个转义序列标识的Unicode代码点转换为实际的服务器编码；如果不可能，则报告错误。

## 4.1.2. 常量

在LightDB中有三种隐式类型常量：字符串、位串和数字。常量也可以被指定显式类型，这可以使得它被更精确地展示以及更有效地处理。这些选择将会在后续小节中讨论。

### 4.1.2.1. 字符串常量

在SQL中，一个字符串常量是一个由单引号（'）包围的任意字符序列，例如' This is a string'。为了在一个字符串中包括一个单引号，可以写两个相连的单引号，例如'Dianne''s horse'。注意这和一个双引号（"）不同。

两个只由空白及至少一个换行分隔的字符串常量会被连接在一起，并且将作为一个写在一起的字符串常量来对待。例如：

```
SELECT 'foo'
'bar' ;
```

等同于：

```
SELECT 'foobar' ;
```

但是：

```
SELECT 'foo' 'bar' ;
```

则不是合法的语法（这种有些奇怪的行为是SQL指定的，LightDB遵循了该标准）。

### 4.1.2.2. C风格转义的字符串常量

LightDB也接受“转义”字符串常量，这也是SQL标准的一个扩展。一个转义字符串常量可以通过在开单引号前面写一个字母E（大写或小写形式）来指定，例如E'foo'（当一个转义字符串常量跨行时，只在第一个开引号之前写E）。在一个转义字符串内部，一个反斜线字符（\）会开始一个C风格的反斜线转义序列，在其中反斜线和后续字符的组合表示一个特殊的字节值（如Table 4.1中所示）。

Table 4.1. 反斜线转义序列

反斜线转义序列	解释
\b	退格
\f	换页
\n	换行
\r	回车
\t	制表符
\o, \oo, \ooo (o = 0 - 7)	八进制字节值

反斜线转义序列	解释
\xh, \xhh (h = 0-9, A-F)	十六进制字节值
\uxxxx, \Uxxxxxxx (x = 0-9, A-F)	16 或 32-位十六进制 Unicode 字符值

跟随在一个反斜线后面的任何其他字符被当做其字面意思。因此，要包括一个反斜线字符，请写两个反斜线 (\\)。在一个转义字符串中包括一个单引号除了普通方法''之外，还可以写成\`。

你要负责保证你创建的字节序列由服务器字符集编码中合法的字符组成，特别是在使用八进制或十六进制转义时。一个有用的替代方法是使用Unicode转义或替代的Unicode转义语法，如Section 4.1.2.8所述；然后服务器将检查字符转换是否可行。

### Caution

如果配置参数`standard_conforming_strings`为`off`，那么LightDB对常规字符串常量和转义字符串常量中的反斜线转义都识别。不过，从LightDB 9.1 开始，该参数的默认值为`on`，意味着只在转义字符串常量中识别反斜线转义。这种行为更兼容标准，但是可能打断依赖于历史行为（反斜线转义总是会被识别）的应用。作为一种变通，你可以设置该参数为`off`，但是最好迁移到符合新的行为。如果你需要使用一个反斜线转义来表示一个特殊字符，为该字符串常量写上一个`E`。

在`standard_conforming_strings`之外，配置参数`escape_string_warning`和`backslash_quote`也决定了如何对待字符串常量中的反斜线。

代码零的字符不能出现在一个字符串常量中。

Lightdb 23.3引入了一个新的数据库级别GUC参数`lightdb_ascii_zero_store_value`。如果数据库兼容模式为`Oracle`或`MySQL`，并且`lightdb_ascii_zero_store_value`为非零ASCII 码值，`E'\0'` 被替换为`lightdb_ascii_zero_store_value`。

更多关于GUC `lightdb_ascii_zero_store_value` 的信息，请参考 `lightdb_ascii_zero_store_value`

#### 4.1.2.3. 带有 Unicode 转义的字符串常量

LightDB也支持另一种类型的字符串转义语法，它允许用代码点指定任意 Unicode 字符。一个 Unicode 转义字符串常量开始于`U&`（大写或小写形式的字母 `U`，后跟花号），后面紧跟着开引号，之间没有任何空白，例如`U&'foo'`（注意这产生了与操作符`&`的混淆。在操作符周围使用空白来避免这个问题）。在引号内，Unicode 字符可以通过写一个后跟 4 位十六进制代码点编号或者一个前面有加号的 6 位十六进制代码点编号的反斜线来指定。例如，字符串`'data'`可以被写为

```
U&'d\0061t\+000061'
```

下面的例子用斯拉夫字母写出了俄语的单词“slon”（大象）：

```
U&' \0441\043B\043E\043D'
```

如果想要一个不是反斜线的转义字符，可以在字符串之后使用`UESCAPE`子句来指定，例如：

```
U&'d!0061t!+000061' UESCAPE '!'
```

转义字符可以是出一个十六进制位、加号、单引号、双引号或空白字符之外的任何单一字符。

要在一个字符串中包括一个表示其字面意思的转义字符，把它写两次。

4位或6位转义形式可用于指定UTF-16代理项对，以组成代码点大于U+FFFF的字符，尽管从技术上讲，6位形式的可用性使得这是不必要的（代理项对不是直接存储的，而是合并到单个代码点中。）

如果服务器编码不是 UTF-8，则由这些转义序列之一标识的 Unicode 代码点将转换为实际的服务器编码； 如果不可能，则会报告错误。

此外，字符串常量的 Unicode 转义语法仅在配置参数 `standard_conforming_strings` 开启时才有效。这是因为否则这种语法可能会混淆解析 SQL 语句的客户端，可能导致 SQL 注入和类似的安全问题。如果该参数设置为 `off`，则此语法将被拒绝并显示错误消息。

#### 4.1.2.4. 美元引用的字符串常量

虽然用于指定字符串常量的标准语法通常都很方便，但是当字符串中包含了太多单引号或反斜线时很难理解它，因为每一个都需要被双写。要在这种情形下允许可读性更好的查询，LightDB提供了另一种被称为“美元引用”的方式来书写字符串常量。一个美元引用的字符串常量由一个美元符号（\$）、一个可选的另个或更多字符的“标签”、另一个美元符号、一个构成字符串内容的任意字符序列、一个美元符号、开始这个美元引用的相同标签和一个美元符号组成。例如，这里有两种不同的方法使用美元引用指定字符串“Dianne’s horse”：

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

注意在美元引用字符串中，单引号可以在不被转义的情况下使用。事实上，在一个美元引用字符串中不需要对字符进行转义：字符串内容总是按其字面意思写出。反斜线不是特殊的，并且美元符号也不是特殊的，除非它们是匹配开标签的一个序列的一部分。

可以通过在每一个嵌套级别上选择不同的标签来嵌套美元引用字符串常量。这最常被用在编写函数定义上。例如：

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

这里，序列 `$q$[\t\r\n\v\\]$q$` 表示一个美元引用的文字串 `[\t\r\n\v\\]`，当该函数体被 LightDB 执行时它将被识别。但是因为该序列不匹配外层的美元引用的定界符 `$function$`，它只是一些在外层字符串所关注的常量中的字符而已。

一个美元引用字符串的标签（如果有）遵循一个未被引用标识符的相同规则，除了它不能包含一个美元符号之外。标签是大小写敏感的，因此 `$tag$string content$tag$` 是正确的，但是 `$TAG$string content$tag$` 不正确。

一个跟着一个关键词或标识符的美元引用字符串必须用空白与之分隔开，否则美元引用定界符可能会被作为前面标识符的一部分。

美元引用不是 SQL 标准的一部分，但是在书写复杂字符串文字方面，它常常是一种比兼容标准的单引号语法更方便的方法。当要表示的字符串常量位于其他常量中时它特别有用，这种情况常常在过程函数定义中出现。如果用单引号语法，上一个例子中的每个反斜线将必须被写成四个反斜线，这在解析原始字符串常量时会被缩减到两个反斜线，并且接着在函数执行期间重新解析内层字符串常量时变成一个。

### 4.1.2.5. Q-Quoted字符串常量

在表达式、条件、SQL函数和其他部分的SQL语句的语法中，每当字符串出现时，请使用文本字面值表示值。本参考使用术语文本字面值、字符字面值和字符串互换使用。文本、字符和字符串字面值始终用单引号括起来。如果语法使用术语char，则可以指定文本字面值或解析为字符数据的另一个表达式。当char出现在语法中时，不使用单引号。使用q引用符：

```
q' !name LIKE '%DBMS_%' !'
q' <'So,' she said, 'It's finished.'>'
q' {SELECT * FROM employees WHERE last_name = 'Smith';}'
nq' i ÿ1234 i'
q" name like '[' "
```

Lightdb数据库提供了在SQL和PL/pgSQL中指定自定义分隔符的能力。具体方法如下：在字符串文字前面添加字母“q”。然后输入单引号，后面跟随您的字符串文字的起始分隔符。然后输入您的字符串文字，而不必将单引号字符重复。当您输入完整的字符串文字后，请使用结束分隔符终止它，然后再加上一个单引号。换句话说，您将遵循这个一般格式：q' [your string here]'，其中“[”代表（当然可以用于）起始分隔符，而“]”代表结束分隔符。Lightdb自动识别“配对”的分隔符，例如[]，{}，（），和<>。如果您想使用其他字符作为起始分隔符，并且它没有“自然”的终止伙伴，则必须使用相同的字符作为起始和结束分隔符。最后，如果您选择一个分隔符字符，并且它在单引号标记之前立即出现在您的字符串中，Lightdb将无法识别并引发错误。

### 4.1.2.6. 位串常量

位串常量看起来像常规字符串常量在开引号之前（中间无空白）加了一个B（大写或小写形式），例如B'1001'。位串常量中允许的字符只有0和1。

作为一种选择，位串常量可以用十六进制记号法指定，使用一个前导X（大写或小写形式），例如X'1FF'。这种记号法等价于一个用四个二进制位取代每个十六进制位的位串常量。

两种形式的位串常量可以以常规字符串常量相同的方式跨行继续。美元引用不能被用在位串常量中。

### 4.1.2.7. 数字常量

在这些一般形式中可以接受数字常量：

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

其中digits是一个或多个十进制数字（0 到 9）。如果使用了小数点，在小数点前面或后面必须至少有一个数字。如果存在一个指数标记（e），在其后必须跟着至少一个数字。在该常量中不能嵌入任何空白或其他字符。注意任何前导的加号或减号并不实际被考虑为常量的一部分，它是一个应用到该常量的操作符。

这些是合法数字常量的例子：

```
42
3.5
4.
.001
5e2
1.925e-3
```



如果一个不包含小数点和指数的数字常量的值适合类型integer（32位），它首先被假定为类型integer。否则如果它的值适合类型bigint（64位），它被假定为类型bigint。再否则它会被取做类型numeric。包含小数点和/或指数的常量总是首先被假定为类型numeric。

一个数字常量初始指派的数据类型只是类型转换算法的一个开始点。在大部分情况中，常量将被根据上下文自动被强制到最合适的类型。必要时，你可以通过造型它来强制一个数字值被解释为一种指定数据类型。例如，你可以这样强制一个数字值被当做类型real（float4）：

```
REAL '1.23' -- string style
1.23::REAL -- LightDB (historical) style
```

这些实际上只是接下来要讨论的一般造型记号的特例。

### 4.1.2.8. 其他类型的常量

一种任意类型的一个常量可以使用下列记号中的任意一种输入：

```
type 'string'
'string'::type
CAST('string' AS type)
```

字符串常量的文本被传递到名为type的类型的输入转换例程中。其结果是指定类型的一个常量。如果对该常量的类型没有歧义（例如，当它被直接指派给一个表列时），显式类型造型可以被忽略，在那种情况下它会被自动强制。

字符串常量可以使用常规 SQL 记号或美元引用书写。

也可以使用一个类似函数的语法来指定一个类型强制：

```
typename('string')
```

但是并非所有类型名都可以用在这种方法中，详见Section 4.2.9

如Section 4.2.9中讨论的，::、CAST()以及函数调用语法也可以被用来指定任意表达式的运行时类型转换。要避免语法歧义，type 'string'语法只能被用来指定简单文字常量的类型。type 'string'语法上的另一个限制是它无法对数组类型工作，指定一个数组常量的类型可使用::或CAST()。

CAST()语法符合 SQL。type 'string'语法是该标准的一般化：SQL 指定这种语法只用于一些数据类型，但是LightDB允许它用于所有类型。带有::的语法是LightDB的历史用法，就像函数调用语法一样。

### 4.1.3. 操作符

一个操作符名是最多NAMEDATALEN-1（默认为 63）的一个字符序列，其中的字符来自下面的列表：

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

不过，在操作符名上有一些限制：

- -- and /\*不能在一个操作符名的任何地方出现，因为它们将被作为一段注释的开始。
- 一个多字符操作符名不能以+或-结尾，除非该名称也至少包含这些字符中的一个：

~ ! @ # % ^ & | ` ?

例如，@-是一个被允许的操作符名，但\*-不是。这些限制允许LightDB解析 SQL 兼容的查询而不需要在记号之间有空格。

当使用非 SQL 标准的操作符名时，你通常需要用空格分隔相邻的操作符来避免歧义。例如，如果你定义了一个名为@的前缀操作符，你不能写X\*@Y，你必须写X\* @Y来确保LightDB把它读作两个操作符名而不是一个。

#### 4.1.4. 特殊字符

一些不是数字字母的字符有一种不同于作为操作符的特殊含义。这些字符的详细用法可以在描述相应语法元素的地方找到。这一节只是为了告知它们的存在以及总结这些字符的目的。

- 跟随在一个美元符号 (\$) 后面的数字被用来表示在一个函数定义或一个预备语句中的位置参数。在其他上下文中该美元符号可以作为一个标识符或者一个美元引用字符串常量的一部分。
- 圆括号 (()) 具有它们通常的含义，用来分组表达式并且强制优先。在某些情况中，圆括号被要求作为一个特定 SQL 命令的固定语法的一部分。
- 方括号 ([]) 被用来选择一个数组中的元素。更多关于数组的信息见Section 8.15
- 逗号 (,) 被用在某些语法结构中来分割一个列表的元素。
- 分号 (;) 结束一个 SQL 命令。它不能出现在一个命令中间的任何位置，除了在一个字符串常量中或者一个被引用的标识符中。
- 冒号 (:) 被用来从数组中选择“切片” (见Section 8.15)。在某些 SQL 的“方言” (例如嵌入式 SQL) 中，冒号被用来作为变量名的前缀。
- 星号 (\*) 被用在某些上下文中标记一个表的所有域或者组合值。当它被用作一个聚集函数的参数时，它还有一种特殊的含义，即该聚集不要求任何显式参数。
- 句点 (.) 被用在数字常量中，并且被用来分割模式、表和列名。

#### 4.1.5. 注释

一段注释是以双横杠开始并且延伸到行结尾的一个字符序列，例如：

```
-- This is a standard SQL comment
```

另外，也可以使用 C 风格注释块：

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

这里该注释开始于/\*并且延伸到匹配出现的\*/。这些注释块可按照 SQL 标准中指定的方式嵌套，但和 C 中不同。这样我们可以注释掉一大段可能包含注释块的代码。

在进一步的语法分析前，注释会被从输入流中被移除并且实际被替换为空白。

#### 4.1.6. 操作符优先级

Table 4.显示了LightDB中操作符的优先级和结合性。大部分操作符具有相同的优先并且是左结合的。操作符的优先级和结合性被硬写在解析器中。如果您希望以不同于优先级规则所暗示的方式解析具有多个运算符的表达式，请添加括号。

Table 4.2. 操作符优先级（从高到低）

操作符/元素	结合性	描述
.	左	表/列名分隔符
::	左	LightDB-风格的类型转换
[ ]	左	数组元素选择
+ -	右	一元加、一元减
^	左	指数
* / %	左	乘、除、模
+ -	左	加、减
(任意其他操作符)	左	所有其他本地以及用户定义的操作符
BETWEEN IN LIKE ILIKE SIMILAR		范围包含、集合成员关系、字符串匹配
< > = <= >= <>		比较操作符
IS ISNULL NOTNULL		IS TRUE、IS FALSE、IS NULL、IS DISTINCT FROM等
NOT	右	逻辑否定
AND	左	逻辑合取
OR	左	逻辑析取

注意该操作符有限规则也适用于与上述内建操作符具有相同名称的用户定义的操作符。例如，如果你为某种自定义数据类型定义了一个“+”操作符，它将具有和内建的“+”操作符相同的优先级，不管你的操作符要做什么。

当一个模式限定的操作符名被用在OPERATOR语法中时，如下面的例子：

```
SELECT 3 OPERATOR(pg_catalog, +) 4;
```

OPERATOR结构被用来为“任意其他操作符”获得Table 4.2中默认的优先级。不管出现在OPERATOR()中的是哪个指定操作符，这都是真的。

### Note

版本 9.5 之前的LightDB使用的操作符优先级 规则略有不同。特别是，<=、>=和<>习惯于被当作普通操作符，IS 测试习惯于具有较高的优先级。并且在一些认为NOT比 BETWEEN优先级高的情况下，NOT BETWEEN 和相关的结构的行为不一致。为了更好地兼容 SQL 标准并且减少对 逻辑上等价的 结构不一致的处理，这些规则也得到了修改。在大部分情况下， 这些变化不会导致行为上的变化，或者可能会产生“no such operator” 错误，但可以通过增加圆括号解决。不过在一些极端情况中，查询可能在 没有被报告解析错误的情况下发生行为的改变。

## 4.2. 值表达式

值表达式被用于各种各样的环境中，例如在SELECT命令的目标列表中、作为INSERT或UPDATE中的新列值或者若干命令中的搜索条件。为了区别于一个表表达式（是一个表）的结果，一个值表达式的结果有时候被称为一个标量。值表达式因此也被称为标量表达式（或者甚至简称为表达式）。表达式语法允许使用算数、逻辑、集合和其他操作从原始部分计算值。

一个值表达式是下列之一：

- 一个常量或文字值
- 一个列引用
- 在一个函数定义体或预备语句中的一个位置参数引用
- 一个下标表达式
- 一个域选择表达式
- 一个操作符调用
- 一个函数调用
- 一个聚集表达式
- 一个窗口函数调用
- 一个类型转换
- 一个排序规则表达式
- 一个标量子查询
- 一个数组构造器
- 一个行构造器
- 另一个在圆括号（用来分组子表达式以及重载优先级）中的值表达式

在这个列表之外，还有一些结构可以被分类为一个表达式，但是它们不遵循任何一般语法规则。这些通常具有一个函数或操作符的语义并且在Chapter 9中的合适位置解释。一个例子是IS NULL子句。

我们已经在Section 4.1.1中讨论过常量。下面的小节会讨论剩下的选项。

### 4.2.1. 列引用

一个列可以以下面的形式被引用：

```
correlation.columnname
```

correlation是一个表（有可能以一个模式名限定）的名字，或者是在FROM子句中为一个表定义的别名。如果列名在当前索引所使用的表中都是唯一的，关联名称和分隔用的句点可以被忽略（另见Chapter 7）。

### 4.2.2. 位置参数

一个位置参数引用被用来指示一个由 SQL 语句外部提供的值。参数被用于 SQL 函数定义和预备查询中。某些客户端库还支持独立于 SQL 命令字符串来指定数据值，在这种情况下参数被用来引用那些线外数据值。一个参数引用的形式是：

```
$number
```

例如，考虑一个函数dept的定义：

```
CREATE FUNCTION dept(text) RETURNS dept
  AS $$ SELECT * FROM dept WHERE name = $1 $$
LANGUAGE SQL;
```

这里\$1引用函数被调用时第一个函数参数的值。

### 4.2.3. 下标

如果一个表达式得到了一个数组类型的值，那么可以抽取出该数组值的一个特定元素：

```
expression[subscript]
```

或者抽取出多个相邻元素（一个“数组切片”）：

```
expression[lower_subscript:upper_subscript]
```

（这里，方括号[ ]表示其字面意思）。每一个下标自身是一个表达式，它将四舍五入到最近的整数值。

通常，数组表达式必须被加上括号，但是当要被加下标的表达式只是一个列引用或位置参数时，括号可以被忽略。还有，当原始数组是多维时，多个下标可以被连接起来。例如：

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

最后一个例子中的圆括号是必需的。详见Section 8.15

### 4.2.4. 域选择

如果一个表达式得到一个组合类型（行类型）的值，那么可以抽取该行的指定域

```
expression.fieldname
```

通常行表达式必须被加上括号，但是当该表达式是仅从一个表引用或位置参数选择时，圆括号可以被忽略。例如：

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

（因此，一个被限定的列引用实际上只是域选择语法的一种特例）。一种重要的特例是从一个组合类型的表列中抽取一个域：

```
(compositecol).somefield
(mytable.compositecol).somefield
```

这里需要圆括号来显示compositecol是一个列名而不是一个表名，在第二种情况中则是显示mytable是一个表名而不是一个模式名。

你可以通过书写.\*来请求一个组合值的所有域：

(compositecol).\*

这种记法的行为根据上下文会有不同，详见Section 8.16.5

## 4.2.5. 操作符调用

运算符调用有三种可能的语法：

expression operator expression (二元中缀操作符)  
 operator expression (一元前缀操作符)  
 expression operator (一元后缀运算符)

其中operator记号遵循Section 4.1.的语法规则，或者是关键词AND、OR和NOT之一，或者是一个如下形式的受限定操作符名：

OPERATOR(schema.operatorname)

哪个特定操作符存在以及它们是一元的还是二元的取决于由系统或用户定义的那些操作符。Chapter 描述了内建操作符。

## 4.2.6. 函数调用

一个函数调用的语法是一个函数的名称（可能受限于一个模式名）后面跟上封闭于圆括号中的参数列表：

function\_name ([expression [, expression ... ]])

例如，下面会计算 2 的平方根：

sqrt(2)

当在一个某些用户不信任其他用户的数据库中发出查询时，在编写函数调用时应遵守Section 10.8中的安全防范措施。

内建函数的列表在Chapter 9中。其他函数可以由用户增加。

参数可以有选择地被附加名称。详见Section 4.3

### Note

一个采用单一组合类型参数的函数可以被有选择地称为域选择语法，并且反过来域选择可以被写成函数的风格。也就是说，记号col(table)和table.col是可以互换的。这种行为是非 SQL 标准的但是在LightDB中被提供，因为它允许函数的使用来模拟“计算域”。详见Section 8.16.5

## 4.2.7. 聚集表达式

一个聚集表达式表示在由一个查询选择的行上应用一个聚集函数。一个聚集函数将多个输入减少到一个单一输出值，例如对输入的求和或平均。一个聚集表达式的语法是下列之一：

aggregate\_name (expression [, ... ] [ order\_by\_clause ]) [ FILTER  
 ( WHERE filter\_clause ) ]

```

aggregate_name (ALL expression [ , ... ] [ order_by_clause ] ) [ FILTER
  ( WHERE filter_clause ) ]
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] ) [ FILTER
  ( WHERE filter_clause ) ]
aggregate_name ( * ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause )
  [ FILTER ( WHERE filter_clause ) ]

```

这里aggregate\_name是一个之前定义的聚集（可能带有一个模式名限定），并且expression是任意自身不包含聚集表达式的值表达式或一个窗口函数调用。可选的order\_by\_clause和filter\_clause描述如下。

第一种形式的聚集表达式为每一个输入行调用一次聚集。第二种形式和第一种相同，因为ALL是默认选项。第三种形式为输入行中表达式的每一个可区分值（或者对于多个表达式是值的可区分集合）调用一次聚集。第四种形式为每一个输入行调用一次聚集，因为没有特定的输入值被指定，它通常只对于count(\*)聚集函数有用。最后一种形式被用于有序集聚集函数，其描述如下。

大部分聚集函数忽略空输入，这样其中一个或多个表达式得到空值的行将被丢弃。除非另有说明，对于所有内建聚集都是这样。

例如，count(\*)得到输入行的总数。count(f1)得到输入行中f1为非空的数量，因为count忽略空值。而count(distinct f1)得到f1的非空可区分值的数量。

一般地，交给聚集函数的输入行是未排序的。在很多情况中这没有关系，例如不管接收到什么样的输入，min总是产生相同的结果。但是，某些聚集函数（例如array\_agg和string\_agg）依据输入行的排序产生结果。当使用这类聚集时，可选的order\_by\_clause可以被用来指定想要的顺序。order\_by\_clause与查询级别的ORDER BY子句（如Section 7. 所述）具有相同的语法，除非它的表达式总是仅有表达式并且不能是输出列名称或编号。例如：

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

在处理多参数聚集函数时，注意ORDER BY出现在所有聚集参数之后。例如，要这样写：

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

而不能这样写：

```
SELECT string_agg(a ORDER BY a, ',') FROM table; -- 不正确
```

后者在语法上是合法的，但是它表示用两个ORDER BY键来调用一个单一参数聚集函数（第二个是无用的，因为它是一个常量）。

如果在order\_by\_clause之外指定了DISTINCT，那么所有的ORDER BY表达式必须匹配聚集的常规参数。也就是说，你不能在DISTINCT列表没有包括的表达式上排序。

### Note

在一个聚集函数中指定DISTINCT以及ORDER BY的能力是一种LightDB扩展。

按照到目前为止的描述，如果一般目的和统计性聚集中排序是可选的，在要为其排序输入行时可以在该聚集的常规参数列表中放置ORDER BY。有一个聚集函数的子集叫做有序集聚集，它要求一个order\_by\_clause，通常是因为该聚集的计算只对其输入行的特定顺序有意义。有序集聚集的典型例子包括排名和百分位计算。按照上文的最后一种语法，对于一个

有序集聚集，`order_by_clause`被写在 `WITHIN GROUP (...)`中。`order_by_clause`中的表达式会像普通聚集参数一样对每一个输入行计算一次，按照每个 `order_by_clause`的要求排序并且交给该聚集函数作为输入参数（这和非 `WITHIN GROUP order_by_clause`的情况不同，在其中表达式的结果不会被作为聚集函数的参数）。如果有在 `WITHIN GROUP`之前的参数表达式，会把它们称为直接参数以便与列在 `order_by_clause`中的聚集参数相区分。与普通聚集参数不同，针对每次聚集调用只会计算一次直接参数，而不是为每一个输入行计算一次。这意味着只有那些变量被 `GROUP BY` 分组时，它们才能包含这些变量。这个限制同样适用于根本不在一个聚集表达式内部直接参数。直接参数通常被用于百分数之类的东西，它们只有作为每次聚集计算用一次的单一值才有意义。直接参数列表可以为空，在这种情况下，写成 `()` 而不是 `(*)`（实际上 `LightDB`接受两种拼写，但是只有第一种符合 `SQL` 标准）。

有序集聚集的调用例子：

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
percentile_cont
-----
          50489
```

这会从表 `households` 的 `income` 列得到第 50 个百分位或者中位的值。这里 0.5 是一个直接参数，对于百分位部分是一个在不同行之间变化的值的情况它没有意义。

如果指定了 `FILTER`，那么只有对 `filter_clause` 计算为真的输入行会被交给该聚集函数，其他行会被丢弃。例如：

```
SELECT
  count(*) AS unfiltered,
  count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
unfiltered | filtered
-----+-----
          10 |          4
(1 row)
```

预定义的聚集函数在 Section 9.2 中描述。其他聚集函数可以由用户增加。

一个聚集表达式只能出现在 `SELECT` 命令的结果列表或是 `HAVING` 子句中。在其他子句（如 `WHERE`）中禁止使用它，因为那些子句的计算在逻辑上是在聚集的结果被形成之前。

当一个聚集表达式出现在一个子查询中（见 Section 4.2.1 和 Section 9.23），聚集通常在该子查询的行上被计算。但是如果该聚集的参数（以及 `filter clause`，如果有）只包含外层变量则会产生一个异常：该聚集则属于最近的那个外层，并且会在那个查询的行上被计算。该聚集表达式从整体上则是对其所出现的子查询的一种外层引用，并且在那个子查询的任意一次计算中都作为一个常量。只出现在结果列表或 `HAVING` 子句的限制适用于该聚集所属的查询层次。

## 4.2.8. 窗口函数调用

一个窗口函数调用表示在一个查询选择的行的某个部分上应用一个聚集类的函数。和非窗口聚集函数调用不同，这不会被约束为将被选择的行分组为一个单一的输出行——在查询输出中每一个行仍保持独立。不过，窗口函数能够根据窗口函数调用的分组声明（`PARTITION BY` 列表）访问属于当前行所在分组中的所有行。一个窗口函数调用的语法是下列之一：

```
function_name ([expression [, expression ...]]) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ([expression [, expression ...]]) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition )
```



```
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition
)
```

其中window\_definition的语法是

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ]
[, ...] ]
[ frame_clause ]
```

可选的frame\_clause是下列之一

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

其中frame\_start和frame\_end可以是下面形式中的一种

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

而frame\_exclusion可以是下列之一

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

这里，expression表示任何自身不含有窗口函数调用的值表达式。

window\_name是对定义在查询的WINDOW子句中的一个命名窗口声明的引用。还可以使用在WINDOW子句中定义命名窗口的相同语法在圆括号内给定一个完整的window\_definition，详见SELECT参考页。值得指出的是，OVER wname并不严格地等价于OVER (wname ...)，后者表示复制并修改窗口定义，并且在被引用窗口声明包括一个帧子句时会被拒绝。

PARTITION BY选项将查询的行分组成为分区，窗口函数会独立地处理它们。PARTITION BY工作起来类似于一个查询级别的GROUP BY子句，不过它的表达式总是只是表达式并且不能是输出列的名称或编号。如果没有PARTITION BY，该查询产生的所有行被当作一个单一分区来处理。ORDER BY选项决定被窗口函数处理的一个分区中的行的顺序。它工作起来类似于一个查询级别的ORDER BY子句，但是同样不能使用输出列的名称或编号。如果没有ORDER BY，行将被以未指定的顺序被处理。

frame\_clause指定构成窗口帧的行集合，它是当前分区的一个子集，窗口函数将作用在该帧而不是整个分区。帧中的行集合会随着哪一行是当前行而变化。在RANGE、ROWS或者GROUPS模式中指定帧，在每一种情况下，帧的范围都是从frame\_start到frame\_end。如果frame\_end被省略，则末尾默认为CURRENT ROW。

UNBOUNDED PRECEDING的一个frame\_start表示该帧开始于分区的第一行，类似地UNBOUNDED FOLLOWING的一个frame\_end表示该帧结束于分区的最后一行。

在RANGE或GROUPS模式中，CURRENT ROW的一个frame\_start表示帧开始于当前行的第一个平行行（被窗口的ORDER BY子句排序为与当前行等效的行），而CURRENT ROW的一

一个frame\_end表示帧结束于当前行的最后一个平级行。在ROWS模式中，CURRENT ROW就表示当前行。

在offset PRECEDING以及offset FOLLOWING帧选项中，offset必须是一个不包含任何变量、聚集函数或者窗口函数的表达式。offset的含义取决于帧模式：

- 在ROWS模式中，offset必须得到一个非空、非负的整数，并且该选项表示帧开始于当前行之前或者之后指定数量的行。
- 在GROUPS模式中，offset也必须得到一个非空、非负的整数，并且该选项表示帧开始于当前行的平级组之前或者之后指定数量的平级组，这里平级组是在ORDER BY顺序中等效的行集合（要使用GROUPS模式，在窗口定义中就必须有一个ORDER BY子句）。
- 在RANGE模式中，这些选项要求ORDER BY子句正好指定一列。offset指定当前行中那一列的值与它在该帧中前面或后面的行中的列值的最大差值。offset表达式的数据类型会随着排序列的数据类型而变化。对于数字的排序列，它通常是与排序列相同的类型，但对于日期时间排序列它是一个interval。例如，如果排序列是类型date或者timestamp，我们可以写RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING。offset仍然要求是非空且非负，不过“非负”的含义取决于它的数据类型。

在任何一种情况下，到帧末尾的距离都受限于到分区末尾的距离，因此对于离分区末尾比较近的行来说，帧可能会包含比较少的行。

注意在ROWS以及GROUPS模式中，0 PRECEDING和0 FOLLOWING与CURRENT ROW等效。通常在RANGE模式中，这个结论也成立（只要有一种合适的、与数据类型相关的“零”的含义）。

frame\_exclusion选项允许当前行周围的行被排除在帧之外，即便根据帧的开始和结束选项应该把它们包括在帧中。EXCLUDE CURRENT ROW会把当前行排除在帧之外。EXCLUDE GROUP会把当前行以及它在顺序上的平级行都排除在帧之外。EXCLUDE TIES把当前行的任何平级行都从帧中排除，但不排除当前行本身。EXCLUDE NO OTHERS只是明确地指定不排除当前行或其平级行的这种默认行为。

默认的帧选项是RANGE UNBOUNDED PRECEDING，它和RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW相同。如果使用ORDER BY，这会把该帧设置为从分区开始一直到当前行的最后一个ORDER BY平级行的所有行。如果不使用ORDER BY，就意味着分区中所有的行都被包括在窗口帧中，因为所有行都成为了当前行的平级行。

限制是frame\_start不能是UNBOUNDED FOLLOWING、frame\_end不能是UNBOUNDED PRECEDING，并且在上述frame\_start和frame\_end选项的列表中frame\_end选择不能早于frame\_start选择出现——例如不允许RANGE BETWEEN CURRENT ROW AND offset PRECEDING，但允许ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING，虽然它不会选择任何行。

如果指定了FILTER，那么只有对filter\_clause计算为真的输入行会被交给该窗口函数，其他行会被丢弃。只有是聚集的窗口函数才接受FILTER。

内建的窗口函数在Table 9.6中介绍。用户可以加入其他窗口函数。此外，任何内建的或者用户定义的通用聚集或者统计性聚集都可以被用作窗口函数（有序集和假想集聚集当前不能被用作窗口函数）。

使用\*的语法被用来把参数较少的聚集函数当作窗口函数调用，例如count(\*) OVER (PARTITION BY x ORDER BY y)。星号(\*)通常不被用于窗口相关的函数。窗口相关的函数不允许在函数参数列表中使用DISTINCT或ORDER BY。

只有在SELECT列表和查询的ORDER BY子句中才允许窗口函数调用。

更多关于窗口函数的信息可以在Section 3.5 Section 9.2以及Section 7.2.5中找到。

## 4.2.9. 类型转换

一个类型造型指定从一种数据类型到另一种数据类型的转换。LightDB接受两种等价类型造型语法：

```
CAST ( expression AS type )
expression::type
```

CAST语法遵从 SQL，而用::的语法是LightDB的历史用法。

为了与MySQL兼容，在MySQL模式下（参见lightdb\_dblevel\_syntax\_compatible\_type），`cast(expr as char[(n)])` 等价于 `cast(expr as varchar[(n)])`。

当一个造型被应用到一种未知类型的值表达式上时，它表示一种运行时类型转换。只有已经定义了一种合适的类型转换操作时，该造型才会成功。注意这和常量的造型（如Section 4.1.2.8所示）使用不同。应用于一个未修饰串文字的造型表示一种类型到一个文字常量值的初始赋值，并且因此它将对任意类型都成功（如果该串文字的内容对于该数据类型的输入语法是可接受的）。

如果一个值表达式必须产生的类型没有歧义（例如当它被指派给一个表列），通常可以省略显式类型造型，在这种情况下系统会自动应用一个类型造型。但是，只有对在系统目录中被标记为“OK to apply implicitly”的造型才会执行自动造型。其他造型必须使用显式造型语法调用。这种限制是为了防止出人意料的转换被无声无息地应用。

还可以用像函数的语法来指定一次类型造型：

```
typename ( expression )
```

不过，这只对那些名字也作为函数名可用的类型有效。例如，`double precision`不能以这种方式使用，但是等效的`float8`可以。还有，如果名称`interval`、`time`和`timestamp`被用双引号引用，那么由于语法冲突的原因，它们只能以这种风格使用。因此，函数风格造型语法的使用会导致不一致性并且应该尽可能被避免。

#### Note

函数风格的语法事实上只是一次函数调用。当两种标准造型语法之一被用来做一次运行时转换时，它将在内部调用一个已注册的函数来执行该转换。简而言之，这些转换函数具有和它们的输出类型相同的名字，并且因此“函数风格的语法”无非是对底层转换函数的一次直接调用。显然，一个可移植的应用不应当依赖于它。详见CREATE CAST。

## 4.2.10. 排序规则表达式

COLLATE子句会重载一个表达式的排序规则。它被追加到它适用的表达式：

```
expr COLLATE collation
```

这里`collation`可能是一个受模式限定的标识符。COLLATE子句比操作符绑得更紧，必要时可以使用圆括号。

如果没有显式指定排序规则，数据库系统会从表达式所涉及的列中得到一个排序规则，如果该表达式没有涉及列，则会默认采用数据库的默认排序规则。

COLLATE子句的两种常见使用是重载ORDER BY子句中的排序顺序，例如：

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

以及重载具有区域敏感结果的函数或操作符调用的排序规则，例如：

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

注意在后一种情况中，COLLATE子句被附加到我们希望影响的操作符的一个输入参数上。COLLATE子句被附加到该操作符或函数调用的哪个参数上无关紧要，因为被操作符或函数应用的排序规则是考虑所有参数得来的，并且一个显式的COLLATE子句将重载所有其他参数的排序规则（不过，附加非匹配COLLATE子句到多于一个参数是一种错误。详见Section 21.2。因此，这会给出和前一个例子相同的结果：

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

但是这是一个错误：

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

因为它尝试把一个排序规则应用到>操作符的结果，而它的数据类型是非可排序数据类型boolean。

## 4.2.11. 标量子查询

一个标量子查询是一种圆括号内的普通SELECT查询，它刚好返回一行一列（关于书写查询可见Chapter 7。SELECT查询被执行并且该单一返回值被使用在周围的值表达式中。将一个返回超过一行或一列的查询作为一个标量子查询使用是一种错误（但是如果是一次特定执行期间该子查询没有返回行则不是错误，该标量结果被当做为空）。该子查询可以从周围的查询中引用变量，这些变量在该子查询的任何一次计算中都将作为常量。对于其他涉及子查询的表达式还可见Section 9.23

例如，下列语句会寻找每个州中最大的城市人口：

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
FROM states;
```

## 4.2.12. 数组构造器

一个数组构造器是一个能构建一个数组值并且将值用于它的成员元素的表达式。一个简单的数组构造器由关键词ARRAY、一个左方括号[、一个用于数组元素值的表达式列表（用逗号分隔）以及最后的一个右方括号]组成。例如：

```
SELECT ARRAY[1, 2, 3+4];
array
-----
{1, 2, 7}
(1 row)
```

默认情况下，数组元素类型是成员表达式的公共类型，使用和UNION或CASE结构（见Section 10.5 相同的规则决定。你可以通过显式将数组构造器造型为想要的类型来重载，例如：

```
SELECT ARRAY[1, 2, 22.7]::integer[];
array
```

```
-----
{1, 2, 23}
(1 row)
```

这和把每一个表达式单独地造型为数组元素类型的效果相同。关于造型的更多信息请见Section 4.2.9

多维数组值可以通过嵌套数组构造器来构建。在内层的构造器中，关键词ARRAY可以被忽略。例如，这些语句产生相同的结果：

```
SELECT ARRAY[ARRAY[1, 2], ARRAY[3, 4]];
      array
```

```
-----
{{1, 2}, {3, 4}}
(1 row)
```

```
SELECT ARRAY[[1, 2], [3, 4]];
      array
```

```
-----
{{1, 2}, {3, 4}}
(1 row)
```

因为多维数组必须是矩形的，处于同一层次的内层构造器必须产生相同维度的子数组。任何被应用于外层ARRAY构造器的造型会自动传播到所有的内层构造器。

多维数组构造器元素可以是任何得到一个正确种类数组的任何东西，而不仅仅是一个子-ARRAY结构。例如：

```
CREATE TABLE arr(f1 int[], f2 int[]);
```

```
INSERT INTO arr VALUES (ARRAY[[1, 2], [3, 4]], ARRAY[[5, 6], [7, 8]]);
```

```
SELECT ARRAY[f1, f2, '{9, 10}, {11, 12}']::int[] FROM arr;
      array
```

```
-----
{{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}, {{9, 10}, {11, 12}}}}
(1 row)
```

你可以构造一个空数组，但是因为无法得到一个无类型的数组，你必须显式地把你的空数组造型成想要的类型。例如：

```
SELECT ARRAY[]::integer[];
      array
```

```
-----
{}
(1 row)
```

也可以从一个子查询的结果构建一个数组。在这种形式中，数组构造器被写为关键词ARRAY后跟着一个加了圆括号（不是方括号）的子查询。例如：

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
      array
```

```
-----
{2011, 1954, 1948, 1952, 1951, 1244, 1950, 2005, 1949, 1953, 2006, 31, 2412}
(1 row)
```

```
SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
      array
-----
{{1, 2}, {2, 4}, {3, 6}, {4, 8}, {5, 10}}
(1 row)
```

子查询必须返回一个单一列。如果子查询的输出列是非数组类型，结果的一维数组将为该子查询结果中的每一行有一个元素，并且有一个与子查询的输出列匹配的元素类型。如果子查询的输出列是一种数组类型，结果将是同类型的一个数组，但是要高一个维度。在这种情况下，该子查询的所有行必须产生同样维度的数组，否则结果就不会是矩形形式。

用ARRAY构建的一个数组值的下标总是从一开始。更多关于数组的信息，请见Section 8.15

## 4.2.13. 行构造器

一个行构造器是能够构建一个行值（也称作一个组合类型）并用值作为其成员域的表达式。一个行构造器由关键词ROW、一个左圆括号、用于行的域值的零个或多个表达式（用逗号分隔）以及最后的一个右圆括号组成。例如：

```
SELECT ROW(1,2.5,'this is a test');
```

当在列表中有超过一个表达式时，关键词ROW是可选的。

一个行构造器可以包括语法rowvalue.\*，它将被扩展为该行值的元素的一个列表，就像在一个顶层SELECT列表（见Section 8.16.5 中使用.\*时发生的事情一样。例如，如果表t有列f1和f2，那么这些是相同的：

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

### Note

在LightDB 8.2 以前，.\*语法不会在行构造器中被扩展，这样写ROW(t.\*, 42)会创建一个有两个域的行，其第一个域是另一个行值。新的行为通常更有用。如果你需要嵌套行值的旧行为，写内层行值时不要使用.\*，例如ROW(t, 42)。

默认情况下，由一个ROW表达式创建的值是一种匿名记录类型。如果必要，它可以被造型为一种命名的组合类型 — 或者是一个表的行类型，或者是一种用CREATE TYPE AS创建的组类型。为了避免歧义，可能需要一个显式造型。例如：

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 不需要造型因为只有一个 getf1() 存在
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-----
      1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);
```

```
CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;
```

-- 现在我们需要一个造型来指示要调用哪个函数:

```
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR: function getf1(record) is not unique
```

```
SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
  getf1
-----
      1
(1 row)
```

```
SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
  getf1
-----
     11
(1 row)
```

行构造器可以被用来构建存储在一个组合类型表列中的组合值，或者被传递给一个接受组合参数的函数。还有，可以比较两个行值，或者用IS NULL或IS NOT NULL测试一个行，例如：

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');
```

```
SELECT ROW(table.*) IS NULL FROM table; -- detect all-null rows
```

详见Section 9.24 如Section 9.28中所讨论的，行构造器也可以被用来与子查询相连接。

## 4.2.14. 用户变量

在mysql模式（参见lightdb\_dblevel\_syntax\_compatible\_type）中，您可以在一个语句中将值存储在用户定义的变量中，并在另一个语句中稍后引用它。这使您可以将值从一个语句传递到另一个语句。

用户变量写成@var\_name的形式，其中变量名var\_name必须是标识符或关键字。如果您将其标识符用引号括起来，var\_name将被视为列名（例如，@"my-var"）。@var\_name和@var\_name是不同的。@var\_name是用户变量表达式，@ var\_name是列var\_name的绝对值。

```
lightdb@test=# select @c1 := 5;
  @c1
-----
     5
(1 row)
```

```
lightdb@test=# SELECT c1, @c1, @ c1 FROM t;
  c1 | @c1 | ?column?
-----+-----+-----
    0 |    5 |          0
    1 |    5 |          1
(2 rows)
```

```
lightdb@testmysql01=# SET @col = 'c1';
  @col
-----
   c1
(1 row)
```

```
lightdb@testmysql01=# SELECT @col from dual;
@col
-----
c1
(1 row)
```

```
lightdb@testmysql01=# select 'c2' into @col;
@col
-----
c2
(1 row)
```

```
lightdb@testmysql01=# SELECT @col from dual;
@col
-----
c2
(1 row)
```

用户定义的变量是会话特定的。一个客户端定义的用户变量不能被其他客户端看到或使用。

用户变量名称不区分大小写。名称最大长度为64个字符。

用户变量可以从一组有限的数据类型中分配一个值：整数、十进制数、浮点数、文本或空值。分配十进制数值和实数值时，不会保留值的精度或比例。不属于允许类型之一的类型的值会被转换为允许的类型。例如，具有JSON数据类型的值将被转换为字符串。

如果用户变量的值在结果集中被选中，则它将以字符串形式返回给客户端。

如果引用尚未初始化的变量，则它的值为NULL，类型为字符串。

在使用这种方式进行赋值时，赋值运算符必须是:=而不是=，因为后者在语句中视为比较运算符=。

```
lightdb@test=# select @t1:=1, @t2:=2, @t3:=4;
@t1 | @t2 | @t3
-----+-----+-----
  1 |   2 |   4
(1 row)
```

```
lightdb@test=# SELECT @t1, @t2, @t3, @t4 := @t1 + @t2 + @t3;
@t1 | @t2 | @t3 | @t4
-----+-----+-----+-----
  1 |   2 |   4 |   7
(1 row)
```

对于所有语句，您可能会得到您期望的结果，但这并不是保证的。在以下语句中，您可能认为LightDB首先评估@a，然后进行第二次赋值：

```
SELECT @a, @a:=@a+1, ...;
```

然而，涉及用户变量的表达式的计算顺序是未定义的。

在同一条语句中给变量赋值并读取值的另一个问题在于，变量的默认结果类型是基于语句开始时的类型。以下示例说明了这一点：



```
lightdb@test=# select @a:='test';
lightdb@test=# SELECT @a, (@a:=20) FROM tbl_name;
```

对于这个SELECT语句，LightDB向客户端报告列一是一个字符串，并将对@a的所有访问都转换为字符串，即使@a在第二行被设置为一个数字。在SELECT语句执行后，@a在下一条语句中被视为一个数字。

为避免这种行为可能带来的问题，要么不在同一条语句中对变量赋值和读取变量的值，要么在使用变量之前为其定义类型。

## 4.2.15. 表达式计算规则

子表达式的计算顺序没有被定义。特别地，一个操作符或函数的输入不必按照从左至右或其他任何固定顺序进行计算。

此外，如果一个表达式的结果可以通过只计算其一部分来决定，那么其他子表达式可能完全不需要被计算。例如，如果我们写：

```
SELECT true OR somefunc();
```

那么somefunc()将（可能）完全不被调用。如果我们写成下面这样也是一样：

```
SELECT somefunc() OR true;
```

注意这和一些编程语言中布尔操作符从左至右的“短路”不同。

因此，在复杂表达式中使用带有副作用的函数是不明智的。在WHERE和HAVING子句中依赖副作用或计算顺序尤其危险，因为在建立一个执行计划时这些子句会被广泛地重新处理。这些子句中布尔表达式（AND/OR/NOT的组合）可能会以布尔代数定律所允许的任何方式被重组。

当有必要强制计算顺序时，可以使用一个CASE结构（见Section 9.18）。例如，在一个WHERE子句中使用下面的方法尝试避免除零是不可靠的：

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

但是这是安全的：

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

一个以这种风格使用的CASE结构将使得优化尝试失败，因此只有必要时才这样做（在这个特别的例子中，最好通过写 $y > 1.5 * x$ 来回避这个问题）。

不过，CASE不是这类问题的万灵药。上述技术的一个限制是，它无法阻止常量子表达式的提早计算。如Section 36.7中所述，当查询被规划而不是被执行时，被标记成IMMUTABLE的函数和操作符可以被计算。因此

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

很可能导致一次除零失败，因为规划器尝试简化常量子表达式。即便是表中的每一行都有 $x > 0$ （这样运行时永远不会进入到ELSE分支）也是这样。

虽然这个特别的例子可能看起来愚蠢，没有明显涉及常量的情况可能会发生在函数内执行的查询中，因为函数参数的值和本地变量可以作为常量被插入到查询中用于规划目的。例如，在PL/pgSQL函数中，使用一个IF-THEN-ELSE语句来保护一种有风险的计算比把它嵌在一个CASE表达式中要安全得多。

另一个同类型的限制是，一个CASE无法阻止其所包含的聚集表达式的计算，因为在考虑SELECT列表或HAVING子句中的其他表达式之前，会先计算聚集表达式。例如，下面的查询会导致一个除零错误，虽然看起来好像已经这种情况加以了保护：

```
SELECT CASE WHEN min(employees) > 0
           THEN avg(expenses / employees)
           END
FROM departments;
```

min()和avg()聚集会在所有输入行上并行地计算，因此如果任何行有employees等于零，在有机会测试min()的结果之前，就会发生除零错误。取而代之的是，可以使用一个WHERE或FILTER子句来首先阻止有问题的输入行到达一个聚集函数。

## 4.3. 调用函数

LightDB允许带有命名参数的函数被使用位置或命名记号法调用。命名记号法对于有大量参数的函数特别有用，因为它让参数和实际参数之间的关联更明显和可靠。在位置记号法中，书写一个函数调用时，其参数值要按照它们在函数声明中被定义的顺序书写。在命名记号法中，参数根据名称匹配函数参数，并且可以以任何顺序书写。对于每一种记法，还要考虑函数参数类型的效果，这些在Section 10. 有介绍。

在任意一种记号法中，在函数声明中给出了默认值的参数根本不需要在调用中写出。但是在命名记号法中特别有用，因为任何参数的组合都可以被忽略。而在位置记号法中参数只能从右往左忽略。

LightDB也支持混合记号法，它组合了位置和命名记号法。在这种情况下，位置参数被首先写出并且命名参数出现在其后。

下列例子将展示所有三种记号法的用法：

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT
false)
RETURNS text
AS
$$
SELECT CASE
    WHEN $3 THEN UPPER($1 || ' ' || $2)
    ELSE LOWER($1 || ' ' || $2)
END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

函数concat\_lower\_or\_upper有两个强制参数，a和b。此外，有一个可选的参数uppercase，其默认值为false。a和b输入将被串接，并且根据uppercase参数被强制为大写或小写形式。这个函数的剩余细节对这里并不重要（详见Chapter 30）。

### 4.3.1. 使用位置记号

在LightDB中，位置记号法是给函数传递参数的传统机制。一个例子：

```
SELECT concat_lower_or_upper('Hello', 'World', true);
concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

所有参数被按照顺序指定。结果是大写形式，因为uppercase被指定为true。另一个例子：

```
SELECT concat_lower_or_upper('Hello', 'World');
concat_lower_or_upper
-----
hello world
(1 row)
```

这里，uppercase参数被忽略，因此它接收它的默认值false，并导致小写形式的输出。在位置记号法中，参数可以按照从右往左被忽略并且因此而得到默认值。

### 4.3.2. 使用命名记号

在命名记号法中，每一个参数名都用=> 指定来把它与参数表达式分隔开。例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
concat_lower_or_upper
-----
hello world
(1 row)
```

再次，参数uppercase被忽略，因此它被隐式地设置为false。使用命名记号法的一个优点是参数可以用任何顺序指定，例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

```
SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

为了向后兼容性，基于“:=”的旧语法仍被支持：

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

### 4.3.3. 使用混合记号

混合记号法组合了位置和命名记号法。不过，正如已经提到过的，命名参数不能超越位置参数。例如：

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

在上述查询中，参数a和b被以位置指定，而uppercase通过名字指定。在这个例子中，这只增加了一点文档。在一个具有大量带默认值参数的复杂函数中，命名的或混合的记号法可以节省大量的书写并且减少出错的机会。

Note

命名的和混合的调用记号法当前不能在调用聚集函数时使用（但是当聚集函数被用作窗口函数时它们可以被使用）。

---

## Chapter 5. 数据定义

本章包含了如何创建用来保存数据的数据库结构。在一个关系型数据库中，原始数据被存储在表中，因此本章的主要工作就是解释如何创建和修改表，以及哪些特性可以控制何种数据会被存储在表中。接着，我们讨论表如何被组织成模式，以及如何将权限分配给表。最后，我们将简短地介绍其他一些影响数据存储的特性，例如继承、表分区、视图、函数和触发器。

### 5.1. 表基础

关系型数据库中的一个表非常像纸上的一张表：它由行和列组成。列的数量和顺序是固定的，并且每一列拥有一个名字。行的数目是变化的，它反映了在一个给定时刻表中存储的数据量。SQL并不保证表中行的顺序。当一个表被读取时，表中的行将以非特定顺序出现，除非明确地指定需要排序。这些将在Chapter 介绍。此外，SQL不会为行分配唯一的标识符，因此在一个表中可能会存在一些完全相同的行。这是SQL之下的数学模型导致的结果，但并不是所期望的。稍后在本章中我们将看到如何处理这种问题。

每一列都有一个数据类型。数据类型约束着一组可以分配给列的可能值，并且它为列中存储的数据赋予了语义，这样它可以用于计算。例如，一个被声明为数字类型的列将不会接受任何文本串，而存储在这样一列中的数据可以用来进行数学计算。反过来，一个被声明为字符串类型的列将接受几乎任何一种的数据，它可以进行如字符串连接的操作但不允许进行数学计算。

LightDB包括了相当多的内建数据类型，可以适用于很多应用。用户也可以定义他们自己的数据类型。大部分内建数据类型有着显而易见的名称和语义，所以我们将它们的详细解释放在Chapter 8中。一些常用的数据类型是：用于整数的integer；可以用于分数的numeric；用于字符串的text，用于日期的date，用于一天内时间的time以及可以同时包含日期和时间的timestamp。

要创建一个表，我们要用到CREATE TABLE命令。在这个命令中 我们需要为新表至少指定一个名字、列的名字及数据类型。例如：

```
CREATE TABLE my_first_table (  
    first_column text,  
    second_column integer  
);
```

这将创建一个名为my\_first\_table的表，它拥有两个列。第一个列名为first\_column且数据类型为text；第二个列名为second\_column且数据类型为integer。表和列的名字遵循Section 4.1. 中解释的标识符语法。类型名称通常也是标识符，但是也有些例外。注意列的列表由逗号分隔并被圆括号包围。

当然，前面的例子是非常不自然的。通常，我们为表和列赋予的名称都会表明它们存储着什么类别的数据。因此让我们再看一个更现实的例子：

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric  
);
```

(numeric类型能够存储小数部分，典型的例子是金额。)

## Tip

当我们创建很多相关的表时，最好为表和列选择一致的命名模式。例如，一种选择是用单数或复数名词作为表名，每一种都受到一些理论家支持。

一个表能够拥有的列的数据是有限的，根据列的类型，这个限制介于250和1600之间。但是，极少会定义一个接近这个限制的表，即便有也是一个值的商榷的设计。

如果我们不再需要一个表，我们可以通过使用DROP TABLE命令来移除它。例如：

```
DROP TABLE my_first_table;
DROP TABLE products;
```

尝试移除一个不存在的表会引起错误。然而，在SQL脚本中在创建每个表之前无条件地尝试移除它的做法是很常见的，即使发生错误也会忽略之，因此这样的脚本可以在表存在和不存在时都工作得很好（如果你喜欢，可以使用DROP TABLE IF EXISTS变体来防止出现错误消息，但这并非标准SQL）。

如果我们需要修改一个已经存在的表，请参考本章稍后的Section 5.6

利用到目前为止所讨论的工具，我们可以创建一个全功能的表。本章的后续部分将集中于为表定义增加特性来保证数据完整性、安全性或方便。如果你希望现在就去填充你的表，你可以跳过这些直接去Chapter 6

## 5.2. 默认值

一个列可以被分配一个默认值。当一个新行被创建且没有为某些列指定值时，这些列将会被它们相应的默认值填充。一个数据操纵命令也可以显式地要求一个列被置为它的默认值，而不需要知道这个值到底是什么（数据操纵命令详见Chapter 6）。

如果没有显式指定默认值，则默认值是空值。这是合理的，因为空值表示未知数据。

在一个表定义中，默认值被列在列的数据类型之后。例如：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

默认值可以是一个表达式，它将在任何需要插入默认值的时候被实时计算（不是表创建时）。一个常见的例子是为一个timestamp列指定默认值为CURRENT\_TIMESTAMP，这样它将得到行被插入时的时间。另一个常见的例子是为每一行生成一个“序列号”。这在LightDB可以按照如下方式实现：

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

这里nextval()函数从一个序列对象Section 9.17。还有一种特别的速写：

```
CREATE TABLE products (
    product_no SERIAL,
```

```
); ...
```

SERIAL速写将在Section 8.1.进一步讨论。

特别是，用于默认值的表达式可以是引用其他列的表达式。有几个限制适用于引用其他列的表达式：

- 用于默认值的表达式所引用的列必须在当前列之前定义。
- 用于默认值的表达式不能引用系统列。

比如：

```
CREATE TABLE test (
    a integer,
    b integer default a*5,
);
```

## 5.3. 生成列

生成的列是一个特殊的列，它总是从其他列计算而来。因此说，它对于列就像视图对于表一样。生成列有两种：存储列和虚拟列。存储生成列在写入(插入或更新)时计算，并且像普通列一样占用存储空间。虚拟生成列不占用存储空间并且在读取时进行计算。如此看来，虚拟生成列类似于视图，存储生成列类似于物化视图(除了它总是自动更新之外)。LightDB目前只实现了存储生成列。

建立一个生成列，在 CREATE TABLE中使用 GENERATED ALWAYS AS 子句，例如：

```
CREATE TABLE people (
    ...,
    height_cm numeric,
    height_in numeric GENERATED ALWAYS AS (height_cm / 2.54) STORED
);
```

必须指定关键字 STORED 以选择存储类型的生成列。更多细节请参见 CREATE TABLE 。

生成列不能被直接写入。在INSERT 或 UPDATE 命令中，不能为生成列指定值，但是可以指定关键字DEFAULT。

考虑列缺省情况和生成列之间的差异。如果没有提供其他值，列缺省情况下在行被首次插入时计算一次；生成列则在行每次改变时进行更新，并且不能被取代。列缺省情况下不能引用表的其他列；生成表达式通常会这样做。列缺省情况下可以使用易失性函数，例如random()或引用当前时间函数；而对于生成列这是不允许的。

生成列和涉及生成列的表的定义有几个限制：

- 生成表达式只能使用不可变函数，并且不能使用子查询或以任何方式引用当前行以外的任何内容。
- 生成表达式不能引用另一个生成列。
- 生成表达式不能引用系统表，除了 tableoid。
- 生成列不能具有列默认或标识定义。
- 生成列不能是分区键的一部分。
- 外部表可以有生成列。更多细节请参见 CREATE FOREIGN TABLE 。

- 对于继承：
  - 如果父列是生成的列，则子列也必须也是使用相同的表达式生成的列。在子列的定义中，不再使用GENERATED子句，因为它将从父列复制过来。
  - 在进行多重继承的情况下，如果一个父列是生成的列，那么所有父列都必须是生成的列，并且具有相同的表达式。
  - 如果父列不是生成的列，子列可以定义是否为生成的列。

使用生成列的其他注意事项。

- 生成列保留着有别于其下层的基础列的访问权限。因此，可以对其进行排列以便于从生成列中读取特定的角色，而不是从下层基础列。
- 从概念上讲，生成列在BEFORE 触发器运行后更新。因此，BEFORE 触发器中的基础列所做的变更将反映在生成列中。但相反，不允许访问BEFORE 触发器中的生成列。

## 5.4. 约束

数据类型是一种限制能够存储在表中数据类别的方法。但是对于很多应用来说，它们提供的约束太粗糙。例如，一个包含产品价格的列应该只接受正值。但是没有任何一种标准数据类型只接受正值。另一个问题是我们可能需要根据其他列或行来约束一个列中的数据。例如，在一个包含产品信息的表中，对于每个产品编号应该只有一行。

到目前为止，SQL允许我们在列和表上定义约束。约束让我们能够根据我们的愿望来控制表中的数据。如果一个用户试图在一个列中保存违反一个约束的数据，一个错误会被抛出。即便是这个值来自于默认值定义，这个规则也同样适用。

### 5.4.1. 检查约束

一个检查约束是最普通的约束类型。它允许我们指定一个特定列中的值必须要满足一个布尔表达式。例如，为了要求正值的产品价格，我们可以使用：

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0)  
);
```

如你所见，约束定义就和默认值定义一样跟在数据类型之后。默认值和约束之间的顺序没有影响。一个检查约束有关键字CHECK以及其后的包围在圆括号中的表达式组成。检查约束表达式应该涉及到被约束的列，否则该约束也没什么实际意义。

我们也可以给与约束一个独立的名称。这会使得错误消息更为清晰，同时也允许我们在需要更改约束时能引用它。语法为：

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CONSTRAINT positive_price CHECK (price > 0)  
);
```

要指定一个命名的约束，请在约束名称标识符前使用关键词CONSTRAINT，然后把约束定义放在标识符之后（如果没有以这种方式指定一个约束名称，系统将会为我们选择一个）。

一个检查约束也可以引用多个列。例如我们存储一个普通价格和一个打折后的价格，而我们希望保证打折后的价格低于普通价格：



```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0),  
    discounted_price numeric CHECK (discounted_price > 0),  
    CHECK (price > discounted_price)  
);
```

前两个约束看起来很相似。第三个则使用了一种新语法。它并没有依附在一个特定的列，而是作为一个独立的项出现在逗号分隔的列列表中。列定义和这种约束定义可以以混合的顺序出现在列表中。

我们将前两个约束称为列约束，而第三个约束为表约束，因为它独立于任何一个列定义。列约束也可以写成表约束，但反过来不行，因为一个列约束只能引用它所依附的那一个列（LightDB并不强制要求这个规则，但是如果我们希望表定义能够在其他数据库系统中工作，那就应该遵循它）。上述例子也可以写成：

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric,  
    CHECK (price > 0),  
    discounted_price numeric,  
    CHECK (discounted_price > 0),  
    CHECK (price > discounted_price)  
);
```

甚至是：

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0),  
    discounted_price numeric,  
    CHECK (discounted_price > 0 AND price > discounted_price)  
);
```

这只是口味的问题。

表约束也可以用列约束相同的方法来指定名称：

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric,  
    CHECK (price > 0),  
    discounted_price numeric,  
    CHECK (discounted_price > 0),  
    CONSTRAINT valid_discount CHECK (price > discounted_price)  
);
```

需要注意的是，一个检查约束在其检查表达式值为真或空值时被满足。因为当任何操作数为空时大部分表达式将计算为空值，所以它们不会阻止被约束列中的空值。为了保证一个列不包含空值，可以使用下一节中的非空约束。

## Note

LightDB不支持引用表数据以外的要检查的新增或更新的行的CHECK约束。虽然违反此规则的CHECK约束在简单测试中看起来能工作，它不能保证数据库不会达到约束条件为假(false)的状态（由于涉及的其他行随后发生了更改）。这将导致数据库转储和重新加载失败。即使完整的数据库状态与约束一致，重新加载也可能失败，因为行未按照满足约束的顺序加载。如果可能的话，使用UNIQUE, EXCLUDE, 或 FOREIGN KEY约束以表示跨行和跨表限制。

如果你希望的是在插入行时的时候对其他行进行一次性检查，而不是持续维护的一致性保证，一个自定义的 trigger 可以用于实现这个功能。（此方法避免了转储/重新加载问题，因为lt\_dump不会重新安装触发器直到重新加载数据之后，因此不会在转储/重新加载期间强制执行检查。）

## Note

LightDB假定CHECK约束的条件是不可变的，也就是说，它们始终为同一输入行提供相同的结果。这个假设是仅在插入或更新行时，而不是在其他时间检查CHECK约束的原因。（上面关于不引用其他表数据的警告实际上是此限制的特殊情况。）

打破此假设的常见方法的一个示例是在CHECK表达式中引用用户定义的函数，然后更改该函数的行为。LightDB不会禁止那样，但它不会注意到现在表中是否有行违反了CHECK约束。这将导致后续数据库转储和重新加载失败。处理此类更改的建议方法是删除约束（使用ALTER TABLE），调整函数定义，然后重新添加约束，从而对所有表行进行重新检查。

## 5.4.2. 非空约束

一个非空约束仅仅指定一个列中不会有空值。语法例子：

```
CREATE TABLE products (
  product_no integer NOT NULL,
  name text NOT NULL,
  price numeric
);
```

一个非空约束总是被写成一个列约束。一个非空约束等价于创建一个检查约束CHECK (column\_name IS NOT NULL)，但在LightDB中创建一个显式的非空约束更高效。这种方式创建的非空约束的缺点是我们无法为它给予一个显式的名称。

当然，一个列可以有多个的约束，只需要将这些约束一个接一个写出：

```
CREATE TABLE products (
  product_no integer NOT NULL,
  name text NOT NULL,
  price numeric NOT NULL CHECK (price > 0)
);
```

约束的顺序没有关系，因为并不需要决定约束被检查的顺序。

NOT NULL约束有一个相反的情况：NULL约束。这并不意味着该列必须为空，进而肯定是无用的。相反，它仅仅选择了列可能为空的默认行为。SQL标准中并不存在NULL约束，因此它不能被用于可移植的应用中（LightDB中加入它是为了和某些其他数据库系统兼容）。但是某

些用户喜欢它，因为它使得在一个脚本文件中可以很容易的进行约束切换。例如，初始时我们可以：

```
CREATE TABLE products (  
    product_no integer NULL,  
    name text NULL,  
    price numeric NULL  
);
```

然后可以在需要的地方插入NOT关键词。

### Tip

在大部分数据库中多数列应该被标记为非空。

## 5.4.3. 唯一约束

唯一约束保证\在一列中或者一组列中保存的数据在表中所有行间是唯一的。写成一个列约束的语法是：

```
CREATE TABLE products (  
    product_no integer UNIQUE,  
    name text,  
    price numeric  
);
```

写成一个表约束的语法是：

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric,  
    UNIQUE (product_no)  
);
```

当写入表约束时。

要为一组列定义一个唯一约束，把它写作一个表级约束，列名用逗号分隔：

```
CREATE TABLE example (  
    a integer,  
    b integer,  
    c integer,  
    UNIQUE (a, c)  
);
```

这指定这些列的组合值在整个表的范围内是唯一的，但其中任意一列的值并不需要是（一般也不是）唯一的。

我们可以通常的方式为一个唯一索引命名：

```
CREATE TABLE products (  
    product_no integer CONSTRAINT must_be_different UNIQUE,
```

```

    name text,
    price numeric
);

```

增加一个唯一约束会在约束中列出的列或列组上自动创建一个唯一B-tree索引。只覆盖某些行的唯一性限制不能被写为一个唯一约束，但可以通过创建一个唯一的部分索引来强制这种限制。

通常，如果表中有超过一行在约束所包括列上的值相同，将会违反唯一约束。但是在这种比较中，两个空值被认为是不同的。这意味着即便存在一个唯一约束，也可以存储多个在至少一个被约束列中包含空值的行。这种行为符合SQL标准，但我们听说一些其他SQL数据库可能不遵循这个规则。所以在开发需要可移植的应用时应注意这一点。

#### 5.4.4. 主键

一个主键约束表示可以用作表中行的唯一标识符的一个列或者一组列。这要求那些值都是唯一的并且非空。因此，下面的两个表定义接受相同的数据：

```

CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);

```

```

CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

```

主键也可以包含多于一个列，其语法和唯一约束相似：

```

CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);

```

增加一个主键将自动在主键中列出的列或列组上创建一个唯一B-tree索引。并且会强制这些列被标记为NOT NULL。

一个表最多只能有一个主键（可以有任意数量的唯一和非空约束，它们可以达到和主键几乎一样的功能，但只能有一个被标识为主键）。关系数据库理论要求每一个表都要有一个主键。但LightDB中并未强制要求这一点，但是最好能够遵循它。

主键对于文档和客户端应用都是有用的。例如，一个允许修改行值的 GUI 应用可能需要知道一个表的主键，以便能唯一地标识行。如果定义了主键，数据库系统也有多种方法来利用主键。例如，主键定义了外键要引用的默认目标列。

#### 5.4.5. 外键

一个外键约束指定一列（或一组列）中的值必须匹配出现在另一个表中某些行的值。我们说这维持了两个关联表之间的引用完整性。

例如我们有一个使用过多次的产品表：

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);
```

让我们假设我们还有一个存储这些产品订单的表。我们希望保证订单表中只包含真正存在的产品的订单。因此我们在订单表中定义一个引用产品表的外键约束：

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products (product_no),  
    quantity integer  
);
```

现在就不可能创建包含不存在于产品表中的product\_no值（非空）的订单。

我们说在这种情况下，订单表是引用表而产品表是被引用表。相应地，也有引用和被引用列的说法。

我们也可以把上述命令简写为：

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products,  
    quantity integer  
);
```

因为如果缺少列的列表，则被引用表的主键将被用作被引用列。

你可以按常规方式为外键约束指定自己的名称。

一个外键也可以约束和引用一组列。照例，它需要被写成表约束的形式。下面是一个例子：

```
CREATE TABLE t1 (  
    a integer PRIMARY KEY,  
    b integer,  
    c integer,  
    FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)  
);
```

当然，被约束列的数量和类型应该匹配被引用列的数量和类型。

有时，外键约束的“其它表”是同一个表也是有用的；这称为自引用 外键。例如，如果希望表中的行表示树结构的节点，可以编写

```
CREATE TABLE tree (  
    node_id integer PRIMARY KEY,  
    parent_id integer REFERENCES tree,  
    name text,  
    ...  
);
```

顶级节点的parent\_id可以为NULL，而非NULL的parent\_id条目将被约束为引用表中的有效行。

一个表可以有超过一个的外键约束。这被用于实现表之间的多对多关系。例如我们有关于产品和订单的表，但我们现在希望一个订单能包含多种产品（这在上面的结构中是不允许的）。我们可以使用这种表结构：

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);  
  
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    shipping_address text,  
    ...  
);  
  
CREATE TABLE order_items (  
    product_no integer REFERENCES products,  
    order_id integer REFERENCES orders,  
    quantity integer,  
    PRIMARY KEY (product_no, order_id)  
);
```

注意在最后一个表中主键和外键之间有重叠。

我们知道外键不允许创建与任何产品都不相关的订单。但如果一个产品在一个引用它的订单创建之后被移除会发生什么？SQL允许我们处理这种情况。直观上，我们有几种选项：

- 不允许删除一个被引用的产品
- 同时也删除引用产品的订单
- 其他？

为了说明这些，让我们上面的多对多关系例子中实现下面的策略：当某人希望移除一个仍然被一个订单引用（通过order\_items）的产品时，我们阻止它。如果某人移除一个订单，订单项也同时被移除：

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);  
  
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    shipping_address text,  
    ...  
);  
  
CREATE TABLE order_items (  
    product_no integer REFERENCES products ON DELETE RESTRICT,  
    order_id integer REFERENCES orders ON DELETE CASCADE,  
    quantity integer,  
    PRIMARY KEY (product_no, order_id)  
);
```

限制删除或者级联删除是两种最常见的选项。RESTRICT阻止删除一个被引用的行。NO ACTION表示在约束被检查时如果有任何引用行存在，则会抛出一个错误，这是我们没有指定

任何东西时的默认行为（这两种选择的本质不同在于NO ACTION允许检查被推迟到事务的最后，而RESTRICT则不会）。CASCADE指定当一个被引用行被删除后，引用它的行也应该被自动删除。还有其他两种选项：SET NULL和SET DEFAULT。这些将导致在被引用行被删除后，引用行中的引用列被置为空值或它们的默认值。注意这些并不会是我们免于遵守任何约束。例如，如果一个动作指定了SET DEFAULT，但是默认值不满足外键约束，操作将会失败。

与ON DELETE相似，同样有ON UPDATE可以用在一个被引用列被修改（更新）的情况，可选的动作相同。在这种情况下，CASCADE意味着被引用列的更新值应该被复制到引用行中。

正常情况下，如果一个引用行的任意一个引用列都为空，则它不需要满足外键约束。如果在外键定义中加入了MATCH FULL，一个引用行只有在它的所有引用列为空时才不需要满足外键约束（因此空和非空值的混合肯定会导致MATCH FULL约束失败）。如果不希望引用行能够避开外键约束，将引用行声明为NOT NULL。

一个外键所引用的列必须是一个主键或者被唯一约束所限制。这意味着被引用列总是拥有一个索引（位于主键或唯一约束之下的索引），因此在其上进行的一个引用行是否匹配的检查将会很高效。由于从被引用表中DELETE一行或者UPDATE一个被引用列将要求对引用表进行扫描以得到匹配旧值的行，在引用列上建立合适的索引也会大有益处。由于这种做法并不是必须的，而且创建索引也有很多种选择，所以外键约束的定义并不会自动在引用列上创建索引。

更多关于更新和删除数据的信息请见Chapter 6 外键约束的语法描述请参考CREATE TABLE。

## 5.4.6. 排他约束

排他约束保证如果将任何两行的指定列或表达式使用指定操作符进行比较，至少其中一个操作符比较将会返回否或空值。语法是：

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

详见CREATE TABLE ... CONSTRAINT ... EXCLUDE。

增加一个排他约束将在约束声明所指定的类型上自动创建索引。

## 5.5. 系统列

每一个表都拥有一些由系统隐式定义的system columns。因此，这些列的名字不能像用户定义的列一样使用（注意这种限制与名称是否为关键词没有关系，即使用引号限定一个名称也无法绕过这种限制）。事实上用户不需要关心这些列，只需要知道它们存在即可。

tableoid

包含这一行的表的OID。该列是特别为从分区的表(见Section 5.11)或继承层次(见Section 5.10)中选择的查询而准备，因为如果没有它将很难知道一行来自于哪个表。tableoid可以与pg\_class的oid列进行连接来获得表的名称。

xmin

插入该行版本的事务身份（事务ID）。一个行版本是一个行的一个特别版本，对一个逻辑行的每一次更新都将创建一个新的行版本。

cmin

插入事务中的命令标识符（从0开始）。

xmax

删除事务的身份（事务ID），对于未删除的行版本为0。对于一个可见的行版本，该列值也可能为非零。这通常表示删除事务还没有提交，或者一个删除尝试被回滚。

cmax

删除事务中的命令标识符，或者为0。

ctid

行版本在其表中的物理位置。注意尽管ctid可以被用来非常快速地定位行版本，但是一个行的ctid会在被更新或者被VACUUM FULL移动时改变。因此，ctid不能作为一个长期行标识符。应使用主键来标识逻辑行。

rowid

作为Oracle的兼容字段，它与ctid函数一致。

当lightdb\_dblevel\_syntax\_compatible\_type为oracle时才生效。

事务标识符也是32位量。在一个历时长久的数据库中事务ID同样会绕回。但如果采取适当的维护过程，这不会是一个致命的问题，详见Chapter 22 但是，长期（超过10亿个事务）依赖事务ID的唯一性是不明智的。

命令标识符也是32位量。这对一个事务中包含的SQL命令设置了一个硬极限： $2^{32}$ （40亿）。在实践中，该限制并不是问题 — 注意该限制只是针对SQL命令的数目而不是被处理的行数。同样，只有真正修改了数据库内容的命令才会消耗一个命令标识符。

## 5.6. 修改表

当我们已经创建了一个表并意识到犯了一个错误或者应用需求发生改变时，我们可以移除表并重新创建它。但如果表中已经被填充数据或者被其他数据库对象引用（例如有一个外键约束），这种做法就显得很不方便。因此，LightDB提供了一族命令来对已有的表进行修改。注意这和修改表中所包含的数据是不同的，这里要做的是对表的定义或者说结构进行修改。

利用这些命令，我们可以：

- 增加列
- 移除列
- 增加约束
- 移除约束
- 修改默认值
- 修改列数据类型
- 重命名列
- 重命名表

所有这些动作都由ALTER TABLE命令执行，其参考页面中包含更详细的信息。

### 5.6.1. 增加列

要增加一个列，可以使用这样的命令：

```
ALTER TABLE products ADD COLUMN description text;
```

新列将被默认值所填充（如果没有指定DEFAULT子句，则会填充空值）。

#### Tip

从 LightDB 11开始，添加一个具有常量默认值的列不再意味着在执行ALTER TABLE 语句时需要更新表的每一行。相反，默认值将在下次访问该行时返回，并在表被重写时应用，从而使得ALTER TABLE即使在大表上也非常快。



但是，如果默认值是可变的（例如`clock_timestamp()`），则每一行需要被`ALTER TABLE`被执行时计算的值更新。为避免潜在的长时间的更新操作，特别是如果你想要用大多数非默认值填充列，那么最好添加没有默认值的列，再用`UPDATE`插入正确的值，然后按照下面所述添加任何期望的默认值。

也可以同时为列定义约束，语法：

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

事实上`CREATE TABLE`中关于一列的描述都可以应用在这里。记住不管怎样，默认值必须满足给定的约束，否则`ADD`将会失败。也可以先将新列正确地填充好，然后再增加约束（见后文）。

## 5.6.2. 移除列

为了移除一个列，使用如下的命令：

```
ALTER TABLE products DROP COLUMN description;
```

列中的数据将会消失。涉及到该列的表约束也会被移除。然而，如果该列被另一个表的外键所引用，LightDB不会安静地移除该约束。我们可以通过增加`CASCADE`来授权移除任何依赖于被删除列的所有东西：

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

关于这个操作背后的一般性机制请见Section 5.14

## 5.6.3. 增加约束

为了增加一个约束，可以使用表约束的语法，例如：

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES
  product_groups;
```

要增加一个不能写成表约束的非空约束，可使用语法：

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

该约束会立即被检查，所以表中的数据必须在约束被增加之前就已经符合约束。

## 5.6.4. 移除约束

为了移除一个约束首先需要知道它的名称。如果在创建时已经给它指定了名称，那么事情就变得很容易。否则约束的名称是由系统生成的，我们必须先找出这个名称。`ltsql`的命令`\d`表名将会对此有所帮助，其他接口也会提供方法来查看表的细节。因此命令是：

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

（如果处理的是自动生成的约束名称，如`$2`，别忘了用双引号使它变成一个合法的标识符。）

和移除一个列相似，如果需要移除一个被某些别的东西依赖的约束，也需要加上CASCADE。一个例子是一个外键约束依赖于被引用列上的一个唯一或者主键约束。

这对除了非空约束之外的所有约束类型都一样有效。为了移除一个非空约束可以用：

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

（回忆一下，非空约束是没有名称的，所以不能用第一种方式。）

### 5.6.5. 更改列的默认值

要为一个列设置一个新默认值，使用命令：

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

注意这不会影响任何表中已经存在的行，它只是为未来的INSERT命令改变了默认值。

要移除任何默认值，使用：

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

这等同于将默认值设置为空值。相应的，试图删除一个未被定义的默认值并不会引发错误，因为默认值已经被隐式地设置为空值。

### 5.6.6. 修改列的数据类型

为了将一个列转换为一种不同的数据类型，使用如下命令：

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

只有当列中的每一个项都能通过一个隐式造型转换为新的类型时该操作才能成功。如果需要一种更复杂的转换，应该加上一个USING子句来指定应该如何把旧值转换为新值。

LightDB将尝试把列的默认值转换为新类型，其他涉及到该列的任何约束也是一样。但是这些转换可能失败或者产生奇特的结果。因此最好在修改类型之前先删除该列上所有的约束，然后在修改完类型后重新加上相应修改过的约束。

### 5.6.7. 重命名列

要重命名一个列：

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

### 5.6.8. 重命名表

要重命名一个表：

```
ALTER TABLE products RENAME TO items;
```

## 5.7. 权限

一旦一个对象被创建，它会被分配一个所有者。所有者通常是执行创建语句的角色。对于大部分类型的对象，初始状态下只有所有者（或者超级用户）能够对该对象做任何事情。为了允许其他角色使用它，必须分配权限。

有多种不同的权

限：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、TRIGGER、CREATE、CONNECT、TEMPORARY、及USAGE。可以应用于一个特定对象的权限随着对象的类型（表、函数等）而不同。有关这些权限含义的更多详细信息请参阅下文。后续的章节将介绍如何使用这些权限。

修改或销毁一个对象的权利是作为该对象的所有者所固有的，不能授予或撤销其本身。

（但是，与所有特权一样，该权利可以由拥有角色的成员继承；请参阅Section 19.3）

一个对象可以通过该对象类型相应的ALTER命令来重新分配所有者，例如

```
ALTER TABLE table_name OWNER TO new_owner;
```

超级用户总是可以做到这点，普通角色只有同时是对象的当前所有者（或者是拥有角色的一个成员）以及新拥有角色的一个成员时才能做同样的事。

要分配权限，可以使用GRANT命令。例如，如果joe是一个已有角色，而accounts是一个已有表，更新该表的权限可以按如下方式授权：

```
GRANT UPDATE ON accounts TO joe;
```

用ALL取代特定权限会把与对象类型相关的所有权限全部授权。

一个特殊的名为PUBLIC的“角色”可以用来向系统中的每一个角色授予一个权限。同时，在数据库中有很多用户时可以设置“组”角色来帮助管理权限。详见Chapter 19

要撤销以前授予的特权，请使用恰当命名的REVOKE命令：

```
REVOKE ALL ON accounts FROM PUBLIC;
```

一般情况下，只有对象拥有者（或者超级用户）可以授予或撤销一个对象上的权限。但是在授予权限时使用“with grant option”来允许接收人将权限转授给其他人。如果后来授予选项被撤销，则所有从接收人那里获得的权限（直接或者通过授权链获得）都将被撤销。更多详情请见GRANT和REVOKE参考页。

对象的所有者可以选择撤销自己的普通特权，例如，令表对于自己和其他人只读。但是所有者总是被视为拥有所有的授予选项，所以他们总是可以重新授予自己的特权。

有效的权限如下：

SELECT

允许 SELECT 从任何列、或特定的列、表、视图、物化视图、或其他类似表格的对象。也允许使用COPY TO。还需要这个权限来引用UPDATE 或 DELETE中现有的列值。对于序列，这个权限还允许使用currval 函数。对于大对象，此权限允许读取对象。

INSERT

允许将新行的INSERT加入表、视图等等。可以在特定列上授予，在这种情况下INSERT命令中只有那些列可以被分配（其他列将因此而收到默认值）。还允许使用COPY FROM。

UPDATE

允许 UPDATE 更新任何列、或指定列、表、视图等等。（实际上，任何有效的UPDATE命令也需要SELECT权限，因为它必须引用表列来确定要更新的行，和/或计算列的新值。）SELECT ... FOR UPDATE和SELECT ... FOR SHARE除了SELECT权限外，还需要至少一列上的这个权限。对于序列，这个权限允许使用 nextval 和 setval 函数。对于大对象，此权限允许写入或截断对象。

DELETE

允许 DELETE 从表、视图等等中删除行。（实际上，任何有效的DELETE命令也需要SELECT权限，因为它必须引用表列来确定要删除的行。）

TRUNCATE

允许在表上TRUNCATE。

REFERENCES

允许创建引用表或表的特定列的外键约束。

TRIGGER

允许在表、视图等等上创建触发器。

CREATE

对于数据库，允许在数据库中创建新的模式和发布，并允许在数据库中安装受信任的扩展。

对于模式，允许在模式中创建新对象。要重命名现有对象，你必须拥有对象 and所包含模式的此权限。

对于表空间，允许在表空间中创建表、索引和临时文件，并允许创建将表空间作为默认表空间的数据库。

注意，取消该特权不会改变现有对象的存在或位置。

CONNECT

允许受让者连接到数据库。此权限在连接启动时进行检查(加之lt\_hba.conf施加的任何约束)。

TEMPORARY

允许在使用数据库时创建临时表。

EXECUTE

允许调用函数或过程，包括使用在函数之上实现的任何运算符。这是适用于函数和过程的唯一权限类型。

USAGE

对于程序语言，允许使用语言来创建该语言的函数。这是适用于过程语言的唯一权限类型。

对于模式，允许访问模式中包含的对象（假设对象自己的权限要求也已得到满足）。从本质上讲，这允许受让者“look up”模式中的对象。如果没有此权限，仍可以看到对象名称，例如通过查询系统目录。此外，在撤消此权限后，现有会话可能还具有以前执行过此查找的语句，因此这不是阻止对象访问的彻底安全的方法。

对于序列，允许使用currval 和 nextval 函数。

对于类型和域，允许在创建表、函数和其他模式对象时使用类型或域。（注意，此权限不控制类型的全部“usage”，例如查询中出现的类型的值。它仅防止创建依赖于类型的对象。此权限的主要目的是控制哪些用户可以对类型创建依赖项，这可能会防止所有者以后更改类型。）

对于外部数据包装器，允许使用外部数据包装器创建新服务器。

对于外部服务器，允许使用服务器创建外部表。受让者还可以创建、更改或删除与该服务器关联的自己的用户映射。

其他命令所需的权限罗列在相应命令的参考页上。

在创建对象时，LightDB默认将某些类型对象的权限授予PUBLIC。默认情况下，在表、表列、序列、外部数据包装器、外部服务器、大型对象、模式或表空间上，不向PUBLIC授予权限。对于其他类型的对象，授予PUBLIC的默认权限如下所示：针对数据库的CONNECT和TEMPORARY（创建临时表）权限；针对函数和程序的EXECUTE权限；以及针对语言和数据类型（包括域）的USAGE权限。当然，对象所有者可以REVOKE默认权限和特别授予的权限。（为了最大程度的安全性，在创建对象的同一事务中发出REVOKE；那么就没有其他用户能够使用该对象的窗口。）此外，可以使用ALTER DEFAULT PRIVILEGES命令取代这些默认权限设置。

Table 5. 显示了ACL（访问控制列表）值中用于这些权限类型的单字母缩写。你将在下面列出的 ltsql 命令的输出中，或者在查看系统目录的 ACL 列时看到这些字母。

Table 5.1. ACL 权限缩写

权限	缩写	适用对象类型
SELECT	r (“读”)	LARGE OBJECT, SEQUENCE, TABLE (and table-like objects), table column
INSERT	a (“增补”)	TABLE, table column
UPDATE	w (“写”)	LARGE OBJECT, SEQUENCE, TABLE, table column
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE, table column
TRIGGER	t	TABLE
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE

Table 5.2使用上面所示的缩写总结了每种类型 SQL 对象可用的权限。它还显示可用于检查每种对象类型的特权设置的 ltsql 命令。

Table 5.2. 访问权限摘要

对象类型	所有权限	默认 PUBLIC 权限	ltsql 命令
DATABASE	CTc	Tc	\l
DOMAIN	U	U	\dD+
FUNCTION or PROCEDURE	X	X	\df+
FOREIGN DATA WRAPPER	U	none	\dew+
FOREIGN SERVER	U	none	\des+
LANGUAGE	U	U	\dL+
LARGE OBJECT	rw	none	
SCHEMA	UC	none	\dn+
SEQUENCE	rwU	none	\dp

对象类型	所有权限	默认 PUBLIC 权限	ltsql 命令
TABLE (and table-like objects)	arwdDxt	none	\dp
Table column	arwx	none	\dp
TABLESPACE	C	none	\db+
TYPE	U	U	\dT+

已授予特定对象的权限显示为aclitem项的列表，其中每个aclitem项描述了特定授予者授予给一个被授与者的权限。例如，calvin=r\*w/hobbes 指明角色calvin具有SELECT (r) 权限和授予选项 (\*) 以及不可授予权限UPDATE (w)，均由角色hobbes授予。如果calvin对由其他授予人授予的同一对象也具有一些权限，那将显示为单独的aclitem条目。aclitem中的空受赠方字段代表PUBLIC。

例如，假设用户miriam创建了表mytable并且：

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

则 ltsql 的 \dp 命令将显示：

```
=> \dp mytable
                Access privileges
Schema | Name | Type | Access privileges | Column privileges |
Policies
-----+-----+-----+-----+-----+
public | mytable | table | miriam=arwdDxt/miriam+ | coll:              + |
        |         |      | =r/miriam              + | miriam_rw=rw/miriam |
        |         |      | admin=arw/miriam      |                    |
(1 row)
```

如果“Access privileges”列对于给定对象为空，则表示该对象具有默认权限（也就是说，它在相关系统目录中的权限条目为空）。默认权限始终包含所有者的所有权限，并且可以包括 PUBLIC 的一些权限，具体取决于对象类型，如上所述。对象上的第一个GRANT或REVOKE将实例化默认权限（例如，生成miriam\_arwdDxt/miriam），然后根据指定的请求修改它们。类似的，只有具有非默认特权的列的条目才显示在“Column privileges”中。（注意：为此目的，“default privileges”始终表示对象类型的内置缺省权限。其权限受ALTER DEFAULT PRIVILEGES 命令影响的对象将始终显示一个显式权限条目，其中包含 ALTER。）

注意所有者的隐式授予选项没有在访问权限显示中标记。仅当授予选项被显式授予给某人时才会出现\*。

## 5.8. 行安全性策略

除可以通过GRANT使用 SQL 标准的 特权系统之外，表还可以具有 行安全性策略，它针对每一个用户限制哪些行可以 被普通的查询返回或者可以被数据修改命令插入、更新或删除。这种 特性也被称为行级安全性。默认情况下，表不具有 任何策略，这样用户根据 SQL 特权系统具有对表的访问特权，对于 查询或更新来说其中所有的行都是平等的。

当在一个表上启用行安全性时（使用 ALTER TABLE ... ENABLE ROW LEVEL SECURITY），所有对该表选择行或者修改行的普通访问都必须被一条 行安全性策略所允许（不过，表的拥有者通常不服从行安全性策略）。如果 表上不存在策略，将使用一条默认的否定策略，即所有的行都不可见或者不能 被修改。应用在整个表上的操作不服从行安全性，例如TRUNCATE和 REFERENCES。

行安全性策略可以针对特定的命令、角色或者两者。一条策略可以被指定为 适用于ALL命令，或者SELECT、INSERT、UPDATE、或者DELETE。可以为一条给定策略分配多个角色，并且通常的角色成员关系和继承规则也适用。

要指定哪些行根据一条策略是可见的或者是可修改的，需要一个返回布尔结果的表达式。对于每一行，在计算任何来自用户查询的条件或函数之前，先会计算这个表达式（这条规则的唯一例外是leakproof函数，它们被保证不会泄露信息，优化器可能会选择在行安全性检查之前应用这类函数）。使该表达式不返回true的行将不会被处理。可以指定独立的表达式来单独控制哪些行可见以及哪些行被允许修改。策略表达式会作为查询的一部分运行并且带有运行该查询的用户的特权，但是安全性定义者函数可以被用来访问对调用用户不可用的数据。

具有BYPASSRLS属性的超级用户和角色在访问一个表时总是可以绕过行安全性系统。表所有者通常也能绕过行安全性，不过表所有者可以选择用ALTER TABLE ... FORCE ROW LEVEL SECURITY来服从行安全性。

启用和禁用行安全性以及向表增加策略是只有表所有者具有的特权。

策略的创建可以使用CREATE POLICY命令，策略的修改可以使用ALTER POLICY命令，而策略的删除可以使用DROP POLICY命令。要为一个给定表启用或者禁用行安全性，可以使用ALTER TABLE命令。

每一条策略都有名称并且可以为一个表定义多条策略。由于策略是表相关的，一个表的每一条策略都必须有一个唯一的名称。不同的表可以拥有相同名称的策略。

当多条策略适用于一个给定的查询时，会把它们用OR（对宽容性策略，默认的策略类型）或者AND（对限制性策略）组合在一起。这和给定角色拥有它作为成员的所有角色的特权的规则类似。宽容性策略和限制性策略在下文将会进一步讨论。

作为一个简单的例子，这里是如何在account关系上创建一条策略以允许只有managers角色的成员能访问行，并且只能访问它们账户的行：

```
CREATE TABLE accounts (manager text, company text, contact_email text);
```

```
ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;
```

```
CREATE POLICY account_managers ON accounts TO managers
    USING (manager = current_user);
```

上面的策略隐含地提供了一个与其该约束适用于被一个命令选择的行（这样一个经理不能SELECT、UPDATE或者DELETE属于其他经理的已有行）以及被一个命令修改的行（这样属于其他经理的行不能通过INSERT或者UPDATE创建）。

如果没有指定角色或者使用了特殊的用户名PUBLIC，则该策略适用于系统上所有的用户。要允许所有用户访问users表中属于他们自己的行，可以使用一条简单的策略：

```
CREATE POLICY user_policy ON users
    USING (user_name = current_user);
```

这个例子的效果和前一个类似。

为了对增加到表中的行使用与可见行不同的策略，可以组合多条策略。这一对策略将允许所有用户查看users表中的所有行，但只能修改他们自己的行：

```
CREATE POLICY user_sel_policy ON users
    FOR SELECT
    USING (true);
CREATE POLICY user_mod_policy ON users
    USING (user_name = current_user);
```

在一个SELECT命令中，这两条规则被用OR组合在一起，最终的效应就是所有的行都能被选择。在其他命令类型中，只有第二条策略适用，这样其效果就和以前相同。

也可以用ALTER TABLE命令禁用行安全性。禁用行安全性不会移除定义在表上的任何策略，它们只是被简单地忽略。然后该表中的所有行都是可见的并且可修改，服从于标准的SQL特权系统。

下面是一个较大的例子，它展示了这种特性如何被用于生产环境。表 passwd模拟了一个Unix 口令文件：

```
-- 简单的口令文件例子
CREATE TABLE passwd (
  user_name      text UNIQUE NOT NULL,
  pwhash         text,
  uid            int PRIMARY KEY,
  gid            int NOT NULL,
  real_name      text NOT NULL,
  home_phone     text,
  extra_info     text,
  home_dir       text NOT NULL,
  shell          text NOT NULL
);

CREATE ROLE admin; -- 管理员
CREATE ROLE bob;   -- 普通用户
CREATE ROLE alice; -- 普通用户

-- 填充表
INSERT INTO passwd VALUES
  ('admin', 'xxx', 0, 0, 'Admin', '111-222-3333', null, '/root', '/bin/dash');
INSERT INTO passwd VALUES
  ('bob', 'xxx', 1, 1, 'Bob', '123-456-7890', null, '/home/bob', '/bin/zsh');
INSERT INTO passwd VALUES
  ('alice', 'xxx', 2, 1, 'Alice', '098-765-4321', null, '/home/alice', '/bin/zsh');

-- 确保在表上启用行级安全性
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;

-- 创建策略
-- 管理员能看见所有行并且增加任意行
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK (true);
-- 普通用户可以看见所有行
CREATE POLICY all_view ON passwd FOR SELECT USING (true);
-- 普通用户可以更新它们自己的记录，但是限制普通用户可用的 shell
CREATE POLICY user_mod ON passwd FOR UPDATE
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash', '/bin/sh', '/bin/dash', '/bin/zsh', '/bin/tcsh')
  );

-- 允许管理员有所有普通权限
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;
-- 用户只在公共列上得到选择访问
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info, home_dir, shell)
  ON passwd TO public;
-- 允许用户更新特定行
```



```
GRANT UPDATE
    (pwhash, real_name, home_phone, extra_info, shell)
    ON passwd TO public;
```

对于任意安全性设置来说，重要的是测试并确保系统的行为符合预期。 使用上述的例子，下面展示了权限系统工作正确：

-- admin 可以看到所有的行和域

```
lightdb@postgres=> set role admin;
SET
lightdb@postgres=> table passwd;
 user_name | pwhash | uid | gid | real_name | home_phone | extra_info |
 home_dir  | shell
-----+-----+-----+-----+-----+-----+-----+-----
 admin    | xxx    | 0  | 0  | Admin    | 111-222-3333 |           | /root
          | /bin/dash
 bob      | xxx    | 1  | 1  | Bob      | 123-456-7890 |           | /home/
 bob      | /bin/zsh
 alice    | xxx    | 2  | 1  | Alice    | 098-765-4321 |           | /home/
 alice    | /bin/zsh
(3 rows)
```

-- 测试 Alice 能做什么

```
lightdb@postgres=> set role alice;
SET
lightdb@postgres=> table passwd;
ERROR: permission denied for relation passwd
lightdb@postgres=> select
 user_name, real_name, home_phone, extra_info, home_dir, shell from passwd;
 user_name | real_name | home_phone | extra_info | home_dir | shell
-----+-----+-----+-----+-----+-----
 admin    | Admin    | 111-222-3333 |           | /root    | /bin/dash
 bob      | Bob      | 123-456-7890 |           | /home/bob | /bin/zsh
 alice    | Alice    | 098-765-4321 |           | /home/alice | /bin/zsh
(3 rows)
```

```
lightdb@postgres=> update passwd set user_name = 'joe';
ERROR: permission denied for relation passwd
-- Alice 被允许更改她自己的 real_name, 但不能改其他的
lightdb@postgres=> update passwd set real_name = 'Alice Doe';
UPDATE 1
lightdb@postgres=> update passwd set real_name = 'John Doe' where user_name =
 'admin';
UPDATE 0
lightdb@postgres=> update passwd set shell = '/bin/xx';
ERROR: new row violates WITH CHECK OPTION for "passwd"
lightdb@postgres=> delete from passwd;
ERROR: permission denied for relation passwd
lightdb@postgres=> insert into passwd (user_name) values ('xxx');
ERROR: permission denied for relation passwd
-- Alice 可以更改她自己的口令; 行级安全性会悄悄地阻止更新其他行
lightdb@postgres=> update passwd set pwhash = 'abc';
UPDATE 1
```

目前为止所有构建的策略都是宽容性策略，也就是当多条策略都适用时会被适用“OR”布尔操作符组合在一起。而宽容性策略可以被用来仅允许在预计情况中对行的访问，这比将宽容性策略与限制性策略（记录必须通过这类策略并且它们会被“AND”布尔操作符组合起来）

组合在一起更简单。在上面的例子之上，我们增加一条限制性策略要求通过一个本地Unix套接字连接过来的管理员访问passwd表的记录：

```
CREATE POLICY admin_local_only ON passwd AS RESTRICTIVE TO admin
    USING (pg_catalog.inet_client_addr() IS NULL);
```

然后，由于这条限制性规则的存在，我们可以看到从网络连接进来的管理员将无法看到任何记录：

```
=> SELECT current_user;
current_user
-----
admin
(1 row)

=> select inet_client_addr();
inet_client_addr
-----
127.0.0.1
(1 row)

=> TABLE passwd;
user_name | pwhash | uid | gid | real_name | home_phone | extra_info | home_dir
| shell
-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)

=> UPDATE passwd set pwhash = NULL;
UPDATE 0
```

参照完整性检查（例如唯一或逐渐约束和外键引用）总是会绕过行级安全性以保证数据完整性得到维护。在开发模式和行级安全性时必须小心避免“隐通道”通过这类参照完整性检查泄露信息。

在某些环境中确保行安全性没有被应用很重要。例如，在做备份时，如果行安全性悄悄地导致某些行被从备份中忽略掉，这会是灾难性的。在这类情况下，你可以设置row\_security配置参数为 off。这本身不会绕过行安全性，它所做的是如果任何结果会被一条策略过滤掉，就会抛出一个错误。然后错误的原因就可以被找到并且修复。

在上面的例子中，策略表达式只考虑了要被访问的行之中的当前值。这是最简单并且表现最好的情况。如果可能，最好设计行安全性应用以这种方式工作。如果需要参考其他行或者其他表来做出策略的决定，可以在策略表达式中通过使用子-SELECT或者包含SELECT的函数来实现。不过要注意这类访问可能会导致竞争条件，在不小心的情况下这可能会导致信息泄露。作为一个例子，考虑下面的表设计：

```
-- 特权组的定义
CREATE TABLE groups (group_id int PRIMARY KEY,
                    group_name text NOT NULL);

INSERT INTO groups VALUES
    (1, 'low'),
    (2, 'medium'),
    (5, 'high');

GRANT ALL ON groups TO alice; -- alice 是管理员
GRANT SELECT ON groups TO public;
```

```

-- 用户的特权级别的定义
CREATE TABLE users (user_name text PRIMARY KEY,
                    group_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES
    ('alice', 5),
    ('bob', 2),
    ('mallory', 2);

GRANT ALL ON users TO alice;
GRANT SELECT ON users TO public;

-- 保存要被保护的信息的表
CREATE TABLE information (info text,
                          group_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES
    ('barely secret', 1),
    ('slightly secret', 2),
    ('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- 对于安全性 group_id 大于等于一行的 group_id 的用户,
-- 这一行应该是可见的/可更新的
CREATE POLICY fp_s ON information FOR SELECT
    USING (group_id <= (SELECT group_id FROM users WHERE user_name =
    current_user));
CREATE POLICY fp_u ON information FOR UPDATE
    USING (group_id <= (SELECT group_id FROM users WHERE user_name =
    current_user));

-- 我们只依赖于行级安全性来保护信息表
GRANT ALL ON information TO public;

现在假设alice希望更改“有一点点秘密” 的信息，但是觉得mallory不应该看到该行中的
新内容，因此她这样做：

BEGIN;
UPDATE users SET group_id = 1 WHERE user_name = 'mallory';
UPDATE information SET info = 'secret from mallory' WHERE group_id = 2;
COMMIT;

这看起来是安全的，没有窗口可供mallory看到“对 mallory 保密”的字符串。不过，这里
有一种竞争条件。如果mallory正在并行地做：

SELECT * FROM information WHERE group_id = 2 FOR UPDATE;

并且她的事务处于READ COMMITTED模式，她就可能看到“s对 mallory 保密”的东西。如果
她的事务在alice 做完之后就到达信息行，这就会发生。它会阻塞等待 alice的事务提交，
然后拜FOR UPDATE子句所赐 取得更新后的行内容。不过，对于来自users的隐式 SELECT，
它不会取得一个已更新的行，因为子-SELECT没有FOR UPDATE，相反 会使用查询开始时取得
的快照读取users行。因此，策略表达式会测试mallory的特权级别的旧值并且允许她看到
被更新的行。

有多种方法能解决这个问题。一种简单的答案是在行安全性策略中的 子-SELECT里使
用SELECT ... FOR SHARE。 不过，这要求在被引用表（这里是users）上授予 UPDATE特

```

权给受影响的用户，这可能不是我们想要的（但是另一条行安全性策略可能被应用来阻止它们实际使用这个特权，或者子-SELECT可能被嵌入到一个安全性定义者函数中）。还有，在被引用的表上过多并发地使用行共享锁可能会导致性能问题，特别是表更新比较频繁时。另一种解决方案（如果被引用表上的更新不频繁就可行）是在更新被引用表时对它取一个ACCESS EXCLUSIVE锁，这样就没有并发事务能够检查旧的行值了。或者我们可以在提交对被引用表的更新之后、在做依赖于新安全性情况的更改之前等待所有并发事务结束。

更多细节请见CREATE POLICY 和ALTER TABLE。

## 5.9. 模式

一个LightDB数据库集簇中包含一个或更多命名的数据库。角色和一些其他对象类型被整个集簇共享，连接到服务器的客户端只能访问单个数据库中的数据，在连接请求中指定的那一个。

### Note

一个集簇的用户并不必拥有访问集簇中每一个数据库的权限。角色名的共享意味着不可能在同一个集簇中出现重名的不同角色，例如两个数据库中都有叫joe的用户。但系统可以被配置为只允许joe访问某些数据库。

一个数据库包含一个或多个命名模式，模式中包含着表。模式还包含其他类型的命名对象，包括数据类型、函数和操作符。相同的对象名称可以被用于不同的模式中而不会出现冲突，例如schema1和myschema都可以包含名为mytable的表。和数据库不同，模式并不是被严格地隔离：一个用户可以访问他们所连接的数据库中的所有模式内的对象，只要他们有足够的权限。

下面是一些使用模式的原因：

- 允许多个用户使用一个数据库并且不会互相干扰。
- 将数据库对象组织成逻辑组以便更容易管理。
- 第三方应用的对象可以放在独立的模式中，这样它们就不会与其他对象的名称发生冲突。

模式类似于操作系统层的目录，但是模式不能嵌套。

### 5.9.1. 创建模式

要创建一个模式，可使用CREATE SCHEMA命令，并且给出选择的模式名称。例如：

```
CREATE SCHEMA myschema;
```

在一个模式中创建或访问对象，需要使用由模式名和表名构成的限定名，模式名和表名之间以点号分隔：

```
schema.table
```

在任何需要一个表名的地方都可以这样用，包括表修改命令和后续章节要讨论的数据访问命令（为了简洁我们在这里只谈到表，但是这种方式对其他类型的命名对象同样有效，例如类型和函数）。

事实上，还有更加通用的语法：

```
database.schema.table
```

也可以使用，但目前它只是在形式上与SQL标准兼容。如果我们写一个数据库名称，它必须是我们正在连接的数据库。

因此，如果要在一个新模式中创建一个表，可用：

```
CREATE TABLE myschema.mytable (  
    ...  
);
```

要删除一个为空的模式（其中的所有对象已经被删除），可用：

```
DROP SCHEMA myschema;
```

要删除一个模式以及其中包含的所有对象，可用：

```
DROP SCHEMA myschema CASCADE;
```

有关于此的更一般的机制请参见Section 5.14

我们常常希望创建一个由其他人所拥有的模式（因为这是将用户动作限制在良定义的名字空间中的方法之一）。其语法是：

```
CREATE SCHEMA schema_name AUTHORIZATION user_name;
```

我们甚至可以省略模式名称，在此种情况下模式名称将会使用用户名，参见Section 5.9.6

以pg\_开头的模式名被保留用于系统目的，所以不能被用户所创建。

## 5.9.2. 公共模式

在前面的小节中，我们创建的表都没有指定任何模式名称。默认情况下这些表（以及其他对象）会自动的被放入一个名为“public”的模式中。任何新数据库都包含这样一个模式。因此，下面的命令是等效的：

```
CREATE TABLE products (...);
```

以及：

```
CREATE TABLE public.products (...);
```

## 5.9.3. 模式搜索路径

限定名写起来很冗长，通常最好不要把一个特定模式名拉到应用中。因此，表名通常被使用非限定名来引用，它只由表名构成。系统将沿着一条搜索路径来决定该名称指的是哪个表，搜索路径是一个进行查看的模式列表。搜索路径中第一个匹配的表将被认为是所需要的。如果在搜索路径中没有任何匹配，即使在数据库的其他模式中存在匹配的表名也将会报告一个错误。

在不同模式中创建命名相同的对象的能力使得编写每次都准确引用相同对象的查询变得复杂。这也使得用户有可能更改其他用户查询的行为，不管是出于恶意还是无意。由于未经限

定的名称在查询中以及在LightDB内部的广泛使用，在search\_path中增加一个模式实际上是信任所有在该模式中具有CREATE特权的用户。在你运行一个普通查询时，如果恶意用户可以在搜索路径的模式中创建对象，那么他们将能够控制并执行任意SQL函数的对象，而这些事情就像是你在执行一样。

搜索路径中的第一个模式被称为当前模式。除了是第一个被搜索的模式外，如果CREATE TABLE命令没有指定模式名，它将是新创建表所在的模式。

要显示当前搜索路径，使用下面的命令：

```
SHOW search_path;
```

在默认设置下这将返回：

```
search_path
-----
"$user", public
```

第一个元素说明一个和当前用户同名的模式会被搜索。如果不存在这个模式，该项将被忽略。第二个元素指向我们已经见过的公共模式。

搜索路径中的第一个模式是创建新对象的默认存储位置。这就是默认情况下对象会被创建在公共模式中的原因。当对象在任何其他没有模式限定的环境中被引用（表修改、数据修改或查询命令）时，搜索路径将被遍历直到一个匹配对象被找到。因此，在默认配置中，任何非限定访问将只能指向公共模式。

要把新模式放在搜索路径中，我们可以使用：

```
SET search_path TO myschema, public;
```

（我们在这里省略了\$user，因为我们并不立即需要它）。然后我们可以删除该表而无需使用模式进行限定：

```
DROP TABLE mytable;
```

同样，由于myschema是路径中的第一个元素，新对象会被默认创建在其中。

我们也可以这样写：

```
SET search_path TO myschema;
```

这样我们在没有显式限时再也不必去访问公共模式了。公共模式没有什么特别之处，它只是默认存在而已，它也可以被删除。

请注意，无论lt\_catalog模式是否在search\_path中配置，默认情况下，它实际上都会包括lt\_catalog。当配置过Oracle/MySQL模式时，默认的它会放在lt\_catalog之前。

Note that when using ALTER DATABASE/ROLE to set search\_path, the Oracle/MySQL schema contained in it will not take effect, and LightDB will delete the Oracle/MySQL schema by default. 注意，当使用ALTER DATABASE/ROLE来设置search\_path时，如果包含Oracle/MySQL则不会生效。默认情况下，LightDB 会删除 Oracle/MySQL模式。

其他操作模式搜索路径的方法请见Section 9.26

搜索路径对于数据类型名称、函数名称和操作符名称的作用与表名一样。数据类型和函数名称可以使用和表名完全相同的限定方式。如果我们需要在一个表达式中写一个限定的操作符名称，我们必须写成一种特殊的形式：

```
OPERATOR(schema.operator)
```

这是为了避免句法歧义。例如：

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

实际上我们通常都会依赖于搜索路径来查找操作符，因此没有必要去写如此“丑陋”的东西。

## 5.9.4. 模式和权限

默认情况下，用户不能访问不属于他们的模式中的任何对象。要允许这种行为，模式的拥有者必须在该模式上授予USAGE权限。为了允许用户使用模式中的对象，可能还需要根据对象授予额外的权限。

一个用户也可以被允许在其他某人的模式中创建对象。要允许这种行为，模式上的CREATE权限必须被授予。注意在默认情况下，所有人都拥有在public模式上的CREATE和USAGE权限。这使得用户能够连接到一个给定数据库并在它的public模式中创建对象。回收这一特权的使用模式调用：

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

（第一个“public”是模式，第二个“public”指的是“每一个用户”。第一种是一个标识符，第二种是一个关键词，所以两者的大小写不同。请回想Section 4.1. 中的指导方针。）

## 5.9.5. 系统目录模式

除public和用户创建的模式之外，每一个数据库还包括一个pg\_catalog模式，它包含了系统表 and 所有内建的数据类型、函数以及操作符。pg\_catalog总是搜索路径的一个有效部分。如果没有在路径中显式地包括该模式，它将在路径中的模式之前被搜索。这保证了内建的名总是能被找到。然而，如果我们希望用用户定义的名称重载内建的名称，可以显式的将pg\_catalog放在搜索路径的末尾。

由于系统表名称以pg\_开头，最好还是避免使用这样的名称，以避免和未来新版本中可能出现的系统表名发生冲突。系统表将继续采用以pg\_开头的方式，这样它们不会与非限制的用户表名称冲突。

## 5.9.6. 使用方式

模式能够以多种方式组织数据。一个安全模式使用方式可以防止不受信任的用户更改其他用户的查询行为。当数据库没有使用安全模式使用方式时，希望安全地查询该数据库的用户将在每个会话开始时采取保护操作。具体的说，他们开始会话前会设置search\_path为空字符串，或者删除search\_path中非超级用户可写的模式。以下的一些使用方式在默认配置下可以很容易实现。

- 将普通用户约束在其私有的方案中。要实现这一点，执行REVOKE CREATE ON SCHEMA public FROM PUBLIC，并且为每一个用户创建一个用其用户名命名的模式。回想一下，默认搜索路径开头的\$user会解析为用户名。因此，如果每个用户都有单独的模式，则默认情况下他们访问自己的模式。在不受信任的用户已经登录的数据库中采用此使用方式后，请考虑检查公共模式中名字与模式pg\_catalog中对象相类似的对象。此方式是一种安全模

式的使用方式，除非不受信任的用户是数据库所有者或拥有CREATEROLE权限，在这种情况下将不存在没有安全模式使用方式。

- 从默认搜索路径中删除公共模式，通过修改lightdb.conf或执行ALTER ROLE ALL SET search\_path = "\$user"。每一个角色都仍然保留在公共模式中创建对象的能力，但是只有符合的名称才会选择到这些对象。虽然符合的的表引用是安全的，但是调用公共模式中的函数会是不安全或不可靠的。如果要在公共模式中创建函数或扩展，请改用第一个方式。否则，与第一个使用方式一样，这是安全的，除非不受信任的用户是数据库所有者或拥有CREATEROLE权限。
- 保持默认。所有用户都隐式地访问公共模式。这模拟了根本没有使用模式的情况，可以用于从无模式的世界平滑过渡。但是，这绝不是一个安全的使用方式。只有当数据库仅有单个用户或者少数相互信任的用户时，才可以接受。

对于任何一种模式，如果要安装共享的应用（所有人都要用的表、第三方提供的额外函数，等等），可把它们放在单独的模式中。记得授予适当的权限以允许其他用户访问它们。然后用户可以通过以模式名限定名称的方式来引用这些额外的对象，或者他们可以把额外的模式放在自己的搜索路径中。

### 5.9.7. 可移植性

在SQL标准中，不同用户拥有同一个模式中的对象的概念是不存在的。此外，某些实现不允许创建与拥有者名称不同名的模式。事实上，在那些仅实现了标准中基本模式支持的数据库中，模式和用户的概念是等同的。因此，很多用户认为限定名称实际上是由user\_name.table\_name组成的。如果我们为每一个用户都创建了一个模式，LightDB实际也是这样认为的。

同样，在SQL标准中也没有public模式的概念。为了最大限度的与标准一致，我们不应使用（甚至是删除）public模式。

当然，某些SQL数据库系统可能根本没有实现模式，或者提供（很可能是有限制地）允许跨数据库访问的命名空间。如果需要使用这样的系统，为了获得最好的可移植性，最好不要使用模式。

## 5.10. 继承

LightDB实现了表继承，这对数据库设计者来说是一种有用的工具（SQL:1999及其后的版本定义了一种类型继承特性，但和这里介绍的继承有很大的不同）。

让我们从一个例子开始：假设我们要为城市建立一个数据模型。每一个州有很多城市，但是只有一个首府。我们希望能够快速地检索任何特定州的首府城市。这可以通过创建两个表来实现：一个用于州首府，另一个用于不是首府的州城市。然而，当我们想要查看一个城市的数据（不管它是不是一个首府）时会发生什么？继承特性将有助于解决这个问题。我们可以将capitals表定义为继承自cities表：

```
CREATE TABLE cities (
    name          text,
    population    float,
    elevation     int    -- in feet
);
```

```
CREATE TABLE capitals (
    state        char(2)
) INHERITS (cities);
```

在这种情况下，capitals表继承了它的父表cities的所有列。州首府还有一个额外的列state用来表示它所属的州。



在LightDB中，一个表可以从0个或者多个其他表继承，而对一个表的查询则可以引用一个表的所有行或者该表的所有行加上它所有的后代表。默认情况是后一种行为。例如，下面的查询将查找所有高度高于500尺的城市的名称，包括州首府：

```
SELECT name, elevation
   FROM cities
  WHERE elevation > 500;
```

对于来自LightDB教程（见Section 2.1）的例子数据，它将返回：

name	elevation
Las Vegas	2174
Mariposa	1953
Madison	845

在另一方面，下面的查询将找到高度超过500尺且不是州首府的所有城市：

```
SELECT name, elevation
   FROM ONLY cities
  WHERE elevation > 500;
```

name	elevation
Las Vegas	2174
Mariposa	1953

这里的ONLY关键词指示查询只被应用于cities上，而其他在继承层次中位于cities之下的其他表都不会被该查询涉及。很多我们已经讨论过的命令（如SELECT、UPDATE和DELETE）都支持ONLY关键词。

我们也可以在表名后写上一个\*来显式地将后代表包括在查询范围内：

```
SELECT name, elevation
   FROM cities*
  WHERE elevation > 500;
```

写\*不是必需的，因为这种行为总是默认的。不过，为了兼容可以修改默认值的较老版本，现在仍然支持这种语法。

在某些情况下，我们可能希望知道一个特定行来自于哪个表。每个表中的系统列tableoid可以告诉我们行来自于哪个表：

```
SELECT c.tableoid, c.name, c.elevation
   FROM cities c
  WHERE c.elevation > 500;
```

将会返回：

tableoid	name	elevation
139793	Las Vegas	2174
139793	Mariposa	1953
139798	Madison	845

（如果重新生成这个结果，可能会得到不同的OID数字。）通过与pg\_class进行连接可以看到实际的表名：

```
SELECT p.relname, c.name, c.elevation
FROM cities c, pg_class p
WHERE c.elevation > 500 AND c.tableoid = p.oid;
```

将会返回：

relname	name	elevation
cities	Las Vegas	2174
cities	Mariposa	1953
capitals	Madison	845

另一种得到同样效果的方法是使用regclass别名类型，它将象征性地打印出表的OID：

```
SELECT c.tableoid::regclass, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

继承不会自动地将来自INSERT或COPY命令的数据传播到继承层次中的其他表中。在我们的例子中，下面的INSERT语句将会失败：

```
INSERT INTO cities (name, population, elevation, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

我们也许希望数据能被以某种方式被引入到capitals表中，但是这不会发生：INSERT总是向指定的表中插入。在某些情况下，可以通过使用一个规则（见Chapter 39 来将插入动作重定向。但是这对上面的情况并没有帮助，因为cities表根本就不包含state列，因而这个命令将在触发规则之前就被拒绝。

父表上的所有检查约束和非空约束都将自动被它的后代所继承，除非显式地指定了NO INHERIT子句。其他类型的约束（唯一、主键和外键约束）则不会被继承。

一个表可以从超过一个的父表继承，在这种情况下它拥有父表们所定义的列的并集。任何定义在子表上的列也会被加入到其中。如果在这个集合中出现重名列，那么这些列将被“合并”，这样在子表中只会会有一个这样的列。重名列能被合并的前提是这些列必须具有相同的数据类型，否则会导致错误。可继承的检查约束和非空约束会以类似的方式被合并。例如，如果合并成一个合并列的任一列定义被标记为非空，则该合并列会被标记为非空。如果检查约束的名称相同，则他们会被合并，但如果它们的条件不同则合并会失败。

表继承通常是在子表被创建时建立，使用CREATE TABLE语句的INHERITS子句。一个已经被创建的表也可以另外一种方式增加一个新的父亲关系，使用ALTER TABLE的INHERIT变体。要这样做，新的子表必须已经包括和父表相同名称和数据类型的列。子表还必须包括和父表相同的检查约束和检查表达式。相似地，一个继承链接也可以使用ALTER TABLE的 NO INHERIT变体从一个子表中移除。动态增加和移除继承链接可以用于实现表划分（见Section 5.11）。

一种创建一个未来将被用做子女的新表的方法是在CREATE TABLE中使用LIKE子句。这将创建一个和源表具有相同列的新表。如果源表上定义有任何CHECK约束，LIKE的INCLUDING CONSTRAINTS选项可以用来让新的子表也包含和父表相同的约束。

当有任何一个子表存在时，父表不能被删除。当子表的列或者检查约束继承于父表时，它们也不能被删除或修改。如果希望移除一个表和它的所有后代，一种简单的方法是使用CASCADE选项删除父表（见Section 5.14）。

ALTER TABLE将会把列的数据定义或检查约束上的任何变化沿着继承层次向下传播。同样，删除被其他表依赖的列只能使用CASCADE选项。ALTER TABLE对于重名列的合并和拒绝遵循与CREATE TABLE同样的规则。

继承的查询仅在附表上执行访问权限检查。例如，在cities表上授予UPDATE权限也隐含着通过cities访问时在capitals表中更新行的权限。这保留了数据（也）在父表中的样子。但是如果没有额外的授权，则不能直接更新capitals表。以类似的方式，父表的行安全性策略（见Section 5.8 适用于继承查询期间来自于子表的行。只有当子表在查询中被明确提到时，其策略（如果有）才会被应用，在那种情况下，附着在其父表上的任何策略都会被忽略。

外部表（见Section 5.12 也可以是继承层次中的一部分，即可以作为父表也可以作为子表，就像常规表一样。如果一个外部表是继承层次的一部分，那么任何不被该外部表支持的操作也不被整个层次所支持。

## 5.10.1. 警告

注意并非所有的SQL命令都能工作在继承层次上。用于数据查询、数据修改或模式修改（例如SELECT、UPDATE、DELETE、大部分ALTER TABLE的变体，但INSERT或ALTER TABLE ... RENAME不在此列）的命令会默认将子表包含在内并且支持ONLY记号来排除子表。负责数据库维护和调整的命令（如REINDEX、VACUUM）只工作在独立的、物理的表上并且不支持在继承层次上的递归。每个命令相应的行为请参见它们的参考页（SQL 命令）。

继承特性的一个严肃的限制是索引（包括唯一约束）和外键约束值应用在单个表上而非它们的继承子女。在外键约束的引用端和被引用端都是这样。因此，按照上面的例子：

- 如果我们声明cities.name为UNIQUE或者PRIMARY KEY，这将不会阻止capitals表中拥有和cities中城市同名的行。而且这些重复的行将会默认显示在cities的查询中。事实上，capitals在默认情况下是根本不能拥有唯一约束的，并且因此能够包含多个同名的行。我们可以为capitals增加一个唯一约束，但这无法阻止相对于cities的重复。
- 相似地，如果我们指定cities.name REFERENCES某个其他表，该约束不会自动地传播到capitals。在此种情况下，我们可以变通地在capitals上手工创建一个相同的REFERENCES约束。
- 指定另一个表的列REFERENCES cities(name)将允许其他表包含城市名称，但不会包含首府名称。这对于这个例子不是一个好的变通方案。

某些未为继承层次结构实现的功能是为声明性分区实现的。在决定使用旧继承进行分区是否对应用程序有用时，需要非常小心。

## 5.11. 表分区

LightDB支持基本的表划分。本小节介绍为何以及怎样把划分实现为数据库设计的一部分。

### 5.11.1. 概述

划分指的是将逻辑上的一个大表分成一些小的物理上的片。划分有很多益处：

- 在某些情况下查询性能能够显著提升，特别是当那些访问压力大的行在一个分区或者少数几个分区时。分区有效地替代了索引的上层树级别，使得索引中被大量使用的部分更有可能适合内存。
- 当查询或更新访问单个分区的很大一部分时，可以通过使用该分区的顺序扫描来提高性能，而不是使用索引，这将需要分散在整个表中的随机访问读取。
- 如果分区设计中考虑了使用模式，则可以通过添加或删除分区来完成批量加载和删除。使用DROP TABLE删除单个分区，或执行ALTER TABLE DETACH PARTITION操作，比批量操作快得多。这些命令还完全避免了批量DELETE所导致的VACUUM开销。

- 很少使用的数据可以被迁移到便宜且较慢的存储介质上。

当一个表非常大时，划分所带来的好处是非常值得的。一个表何种情况下会从划分获益取决于应用，一个经验法则是当表的尺寸超过了数据库服务器物理内存时，划分会为表带来好处。

LightDB对下列分区形式提供了内建支持：

#### 范围划分

表被根据一个关键列或一组列划分为“范围”，不同的分区的范围之间没有重叠。例如，我们可以根据日期范围划分，或者根据特定业务对象的标识符划分。每个范围的边界被理解为左边包含并且右边不包含。例如，如果一个分区的范围是从1到10，而下一个分区是从10到20，则10的值属于第二个分区，而不是第一个分区。

#### 列表划分

通过显式地列出每一个分区中出现的键值来划分表。

#### 哈希分区

通过为每个分区指定模数和余数来对表进行分区。每个分区所持有的行都满足：分区键的值除以其指定的模数将产生为其指定的余数。

如果你的应用需要使用上面所列之外的分区形式，可以使用诸如继承和UNION ALL视图之类的替代方法。这些方法很灵活，但是却缺少内建声明式分区的一些性能优势。

## 5.11.2. 声明式划分

LightDB 允许你声明一个表被划分为分区。被分割的表称为分区表。声明包括如上所述的分区方法，以及用作分区键的列或表达式列表。

分区表本身是一个“虚拟”表，没有自己的存储空间。相反，存储属于分区，这些分区是与分区表关联的普通表。每个分区存储由其分区边界定义的数据子集。插入到分区表中的所有行都将根据分区键列的值路由到适当的一个分区。如果一行不再满足其原始分区的分区边界，则更新该行的分区键将导致该行移动到另一个分区中。

分区本身可能被定义为分区表，从而导致子分区。尽管所有分区必须具有与其分区父级相同的列，但分区可能具有自己的与其他分区不同的索引、约束以及默认值。创建分区表及分区的更多细节请见CREATE TABLE。

不可能将常规表转换为分区表，反之亦然。但是，可以将现有的常规或分区表添加为分区表的分区，或从分区表中删除分区，将其转换为独立表；这可以简化和加快许多维护过程。有关ATTACH PARTITION and DETACH PARTITION 子命令的详细信息，请参见ALTER TABLE。

分区也可以是外部表，不过它们有一些普通表没有限制，详情请见CREATE FOREIGN TABLE。

### 5.11.2.1. 例子

假定我们正在为一个大型的冰激凌公司构建数据库。该公司每天测量最高温度以及每个区域的冰激凌销售情况。概念上，我们想要一个这样的表：

```
CREATE TABLE measurement (
  city_id      int not null,
  logdate     date not null,
  peaktemp    int,
  unitsales   int
```

);

我们知道大部分查询只会访问上周的、上月的或者上季度的数据，因为这个表的主要用途是为管理层准备在线报告。为了减少需要被存放的旧数据量，我们决定只保留最近3年的数据。在每个月的开始我们将去除掉最早的那个月的数据。在这种情况下我们可以使用分区技术来帮助满足对measurement表的所有不同需求。

要在这种情况下使用声明式分区，可采用下面的步骤：

1. 通过指定PARTITION BY子句把measurement表创建为分区表，该子句包括分区方法（这个例子中是RANGE）以及用作分区键的列列表。

```
CREATE TABLE measurement (
  city_id      int not null,
  logdate     date not null,
  peaktemp    int,
  unitsales   int
) PARTITION BY RANGE (logdate);
```

2. 创建分区。每个分区的定义必须指定与父分区的分区方法和分区键对应的边界。请注意，指定边界如果使得新分区的值与一个或多个现有分区中的值重叠将导致错误。

分区以普通LightDB表（或者可能是外部表）的方式创建。可以为每个分区单独指定表空间和存储参数。

在我们的示例中，每个分区应该保存一个月的数据，以满足一次删除一个月数据的要求。因此，这些命令可能看起来像：

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
  FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');
```

```
CREATE TABLE measurement_y2006m03 PARTITION OF measurement
  FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');
```

...

```
CREATE TABLE measurement_y2007m11 PARTITION OF measurement
  FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');
```

```
CREATE TABLE measurement_y2007m12 PARTITION OF measurement
  FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
  TABLESPACE fasttablespace;
```

```
CREATE TABLE measurement_y2008m01 PARTITION OF measurement
  FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
  WITH (parallel_workers = 4)
  TABLESPACE fasttablespace;
```

（回想一下，相邻分区可以共享一个边界值，因为范围上限被视为不包含的边界。）

如果你打算实现子分区，再次在创建分区的命令中指定PARTITION BY子句，例如：

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
  FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
  PARTITION BY RANGE (peaktemp);
```

在创建了measurement\_y2006m02的分区之后，任何被插入到measurement中且被映射到measurement\_y2006m02的数据（或者直接被插入到measurement\_y2006m02的数据，它被允许来满足这个分区的分区约束）将被基于peaktemp列进一步重定向

到measurement\_y2006m02的一个分区。指定的分区键可以与父亲的分区键重叠，不过在指定子分区的边界时要注意它接受的数据集合是分区自身边界允许的数据集合的一个子集，系统不会尝试检查事情情况是否如此。

将没有映射到任何现有分区的数据插入父表将导致错误；必须手动添加适当的分区。

不需要手动创建描述分区边界条件的表约束。此类约束将自动创建。

3. 在分区表的键列以及任何其它索引上创建索引。（键索引不是严格必需的，但在大多数情况下它是有用的。）会自动在每个分区上创建一个匹配的索引，你稍后创建或附加的任何分区也会有这样的索引。在分区表上声明的索引或唯一约束与分区表的方式相同：实际数据位于各个分区表的子索引中。

```
CREATE INDEX ON measurement (logdate);
```

4. 确保enable\_partition\_pruning配置参数在lightdb.conf中没有被禁用。如果被禁用，查询将不会按照想要的方式被优化。

在上面的例子中，我们会每个月创建一个新分区，因此写一个脚本来自动生成所需的DDL会更好。

### 5.11.2.2. 分区维护

通常在初始定义分区表时建立的分区并非保持静态不变。移除分区持有的旧数据并且为新数据周期性地增加新分区的需求比比皆是。分区的最大好处之一就是可以通过操纵分区结构来近乎瞬时地执行这类让人头痛的任务，而不是物理地去除大量数据。

移除旧数据最简单的选择是删除掉不再需要的分区：

```
DROP TABLE measurement_y2006m02;
```

这可以非常快地删除数百万行记录，因为它不需要逐个删除每个记录。不过要注意上面的命令需要在父表上拿到ACCESS EXCLUSIVE锁。

另一种通常更好的选项是把分区从分区表中移除，但是保留它作为一个表的访问权。有两种形式：

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
```

这允许在数据被丢弃之前对其执行进一步的操作。例如，这通常是使用COPY、lt\_dump或类似工具备份数据的有用时间。这可能也是将数据聚合为较小格式、执行其他数据操作或运行报告的有用时机。

类似地，我们可以增加一个新分区来处理新数据。我们可以在分区表中创建一个空分区，就像上面创建的初始分区那样：

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
TABLESPACE fasttablespace;
```

作为一种选择，有时在分区结构之外创建新表更方便，并在以后使其成为适当的分区。这允许在新数据出现在分区表中之前加载、检查和转换新数据。CREATE TABLE ... LIKE选项有助于避免冗长地重复父表的定义：

```
CREATE TABLE measurement_y2008m02
(LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
TABLESPACE fasttablespace;
```

```
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
```

```

CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );

```

ATTACH PARTITION命令要求在分区表上获取SHARE UPDATE EXCLUSIVE 锁。

在运行ATTACH PARTITION命令之前，建议在要附加的表上创建一个CHECK 约束，该约束与预期的分区约束匹配，如上所示。这样，系统将能够跳过验证隐式分区约束所需的扫描。如果没有CHECK约束，则将扫描表以验证分区约束，同时在该分区上保持ACCESS EXCLUSIVE锁。建议在ATTACH PARTITION完成后，删除现在冗余的CHECK约束。如果所附加的表本身是分区表，则其每个子分区将被递归锁定和扫描，直到遇到合适的CHECK约束或达到叶分区。

类似地，如果分区表具有DEFAULT分区，则建议创建CHECK约束，该约束不包括要附加的分区的约束。如果未执行此操作，则将扫描DEFAULT 分区，以验证其不包含应位于附加分区中的记录。此操作将在DEFAULT 分区上保持ACCESS EXCLUSIVE锁的同时执行。如果 DEFAULT分区本身就是一个分区表，那么它的每个分区将以与附加的表相同的方式进行递归检查，如上所述。

如上所述，可以在分区的表上创建索引，因此会自动将其应用于整个层次结构。 这非常便利，因为不仅现有分区将变为索引，而且将来创建的任何分区都将变为索引。 一个限制是，在创建这样一个分区索引时，不可能同时使用CONCURRENTLY限定符。 为了避免长时间锁，可以对分区表使用CREATE INDEX ON ONLY ;这样的索引被标记为无效，并且分区不会自动应用该索引。 分区上索引的创建可以分别使用CONCURRENTLY， 然后使用ALTER INDEX .. ATTACH PARTITIONattached到父索引。 一旦所有分区的索引附加到父索引，父索引将自动标记为有效。 例如：

```

CREATE INDEX measurement_usls_idx ON ONLY measurement (unitsales);

CREATE INDEX measurement_usls_200602_idx
    ON measurement_y2006m02 (unitsales);
ALTER INDEX measurement_usls_idx
    ATTACH PARTITION measurement_usls_200602_idx;
...

```

该技术也可以与UNIQUE 和PRIMARY KEY 约束一起试用；当创建约束时隐式创建索引。例如：

```

ALTER TABLE ONLY measurement ADD UNIQUE (city_id, logdate);

ALTER TABLE measurement_y2006m02 ADD UNIQUE (city_id, logdate);
ALTER INDEX measurement_city_id_logdate_key
    ATTACH PARTITION measurement_y2006m02_city_id_logdate_key;
...

```

### 5.11.2.3. 限制

分区表有下列限制：

- 分区表上的唯一约束（以及主键）必须包括所有分区键列。这种限制的存在是因为构成约束的各个索引只能直接在它们自己的分区内强制唯一性；因此，分区结构本身必须保证不同分区中不存在重复项。
- 无法创建跨越整个分区表的排它约束。只能单独对每个叶分区设置这样的约束。同样，这种限制源于无法实施跨分区限制。
- 在 INSERT 上的BEFORE ROW触发器无法更改新行的最终目标分区。

- 不允许在同一个分区树中混杂临时关系和持久关系。因此，如果分区表是持久的，则其分区也必须是持久的，反之亦然。在使用临时关系时，分区数的所有成员都必须来自于同一个会话。

各个分区悄悄地使用继承链接到它们的分区表。但是，不可能将继承的所有通用特性用于声明性分区的表或其分区，如下所述。值得注意的是，一个分区除了它所属的分区表之外，不能有任何父级，也不能同时从分区表和常规表继承。这意味着分区表及其分区永远不会与常规表共享继承层次结构。

各个分区悄悄地使用继承链接到它们的分区表。但是，不可能将继承的所有通用特性用于声明性分区的表或其分区，如下所述。值得注意的是，一个分区除了它所属的分区表之外，不能有任何父级，也不能同时从分区表和常规表继承。这意味着分区表及其分区永远不会与常规表共享继承层次结构。

- 分区不能包含父分区中不存在的列。使用CREATE TABLE创建分区时不可能指定列，也不可能事后使用ALTER TABLE向分区添加列。只有当表的列与父级完全匹配时，才可以使用ALTER TABLE... ATTACH PARTITION将表添加为分区。
- 分区表的CHECK和NOT NULL约束总是由其所有分区继承。不允许在分区表上创建标记为NO INHERIT的CHECK约束。如果父表中存在相同的约束，则不能在分区的列上删除NOT NULL约束。
- 只要不存在分区，就只支持使用ONLY在分区表上添加或删除约束。一旦存在分区，使用ONLY将导致错误。相反，可以添加和删除分区本身的约束（如果它们不在父表中）。
- 由于分区表本身没有任何数据，尝试在分区表上使用TRUNCATE，ONLY将始终返回错误。

### 5.11.3. 使用继承的分区

虽然内建的声明式分区适合于大部分常见的用例，但还是有一些场景需要更加灵活的方法。分区可以使用表继承来实现，这能够带来一些声明式分区不支持的特性，例如：

- 对声明式分区来说，分区必须具有和分区表正好相同的列集合，而在表继承中，子表可以有父表中没有出现过的额外列。
- 表继承允许多继承。
- 声明式分区仅支持范围、列表以及哈希分区，而表继承允许数据按照用户的选择来划分（不过注意，如果约束排除不能有效地剪枝子表，查询性能可能会很差）。
- 使用声明性分区时，某些操作需要比使用表继承时更强的锁。例如，从分区表中删除分区需要在父表上使用ACCESS EXCLUSIVE锁，而在常规继承的情况下，使用SHARE UPDATE EXCLUSIVE锁就足够了。

#### 5.11.3.1. 例子

这个示例构建了一个与上面的声明性分区示例等效的分区结构。使用以下步骤：

1. 创建“根”表，所有的“子”表都将从它继承。这个表将不包含数据。不要在这个表上定义任何检查约束，除非想让它们应用到所有的子表上。同样，在这个表上定义索引或者唯一约束也没有意义。对于我们的例子来说，根表是最初定义的measurement表。
2. 创建数个“子”表，每一个都从根表继承。通常，这些表将不会在从根表继承的列集合之外增加任何列。正如声明性分区那样，这些表就是普通的LightDB表（或者外部表）。

```
CREATE TABLE measurement_y2006m02 () INHERITS (measurement);
CREATE TABLE measurement_y2006m03 () INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 () INHERITS (measurement);
CREATE TABLE measurement_y2007m12 () INHERITS (measurement);
```



```
CREATE TABLE measurement_y2008m01 () INHERITS (measurement);
```

3. 为子表增加不重叠的表约束来定义每个分区允许的键值。

典型的例子是：

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
CHECK ( outletID >= 100 AND outletID < 200 )
```

确保约束能保证不同子表允许的键值之间没有重叠。设置范围约束的常见错误：

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

这是错误的，因为不清楚键值200属于哪一个子表。相反，范围应该以如下样式定义：

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
) INHERITS (measurement);
```

```
CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
) INHERITS (measurement);
```

...

```
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
) INHERITS (measurement);
```

```
CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
) INHERITS (measurement);
```

```
CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
) INHERITS (measurement);
```

4. 对于每个子表，在键列上创建一个索引，以及任何想要的其他索引。

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

5. 我们希望我们的应用能够使用INSERT INTO measurement ... 并且数据将被重定向到合适的分区表。我们可以通过为根表附加一个合适的触发器函数来实现这一点。如果数据将只被增加到最后一个分区，我们可以使用一个非常简单的触发器函数：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
```

```
LANGUAGE plpgsql;
```

完成函数创建后，我们创建一个调用该触发器函数的触发器：

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE FUNCTION measurement_insert_trigger();
```

我们必须在每个月重新定义触发器函数，这样它才会总是插入到当前的子表。而触发器的定义则不需要被更新。

我们也可能希望插入数据时服务器会自动地定位应该加入数据的子表。我们可以通过一个更复杂的触发器函数来实现之，例如：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND
        NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
        NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
        NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range. Fix the
measurement_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

触发器的定义和以前一样。注意每一个IF测试必须准确地匹配它的子表的CHECK约束。

当该函数比单月形式更加复杂时，并不需要频繁地更新它，因为可以在需要的时候提前加入分支。

### Note

在实践中，如果大部分插入都会进入最新的子表，最好先检查它。为了简洁，我们为触发器的检查采用了和本例中其他部分一致的顺序。

把插入重定向到一个合适的子表中的另一种不同方法是在根表上设置规则而不是触发器。例如：

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
```

```
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

规则的开销比触发器大很多，但是这种开销是每个查询只有一次，而不是每行一次，因此这种方法可能对批量插入的情况有优势。不过，在大部分情况下，触发器方法将提供更好的性能。

注意COPY会忽略规则。如果想要使用COPY插入数据，则需要拷贝到正确的子表而不是直接放在根表中。COPY会引发触发器，因此在使用触发器方法时可以正常使用它。

规则方法的另一个缺点是，如果规则集合无法覆盖插入日期，则没有简单的方法能够强制产生错误，数据将会无声无息地进入到根表中。

6. 确认constraint\_exclusion配置参数在lightdb.conf中没有被禁用，否则将会不必要地访问子表。

如我们所见，一个复杂的表层次可能需要大量的DDL。在上面的例子中，我们可能为每个月创建一个新的子表，因此编写一个脚本来自动生成所需要的DDL可能会更好。

### 5.11.3.2. 继承分区的维护

要快速移除旧数据，只需要简单地去掉不再需要的子表：

```
DROP TABLE measurement_y2006m02;
```

要从继承层次表中去掉子表，但还是把它当做一个表保留：

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

要增加一个新子表来处理新数据，可以像上面创建的原始子表那样创建一个空的子表：

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

或者，用户可能想要创建新子表并且在将它加入到表层次之前填充它。这可以允许数据在被父表上的查询可见之前对数据进行装载、检查以及转换。

```
CREATE TABLE measurement_y2008m02
    (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

### 5.11.3.3. 提醒

下面的提醒适用于用继承实现的分区：

- 没有自动的方法验证所有的CHECK约束之间是否互斥。编写代码来产生子表以及创建和修改相关对象比手写命令要更加安全。

- 索引和外键约束适用于单个表而不是其继承子级，因此它们有一些caveats 需要注意。
- 这里展示的模式假定行的键列值从不改变，或者说改变不足以让行移动到另一个分区。由于CHECK约束的存在，尝试那样做的UPDATE将会失败。如果需要处理那种情况，可以在子表上放置适当的更新触发器，但是那会使对结构的管理更加复杂。
- 如果使用手工的VACUUM或者ANALYZE命令，不要忘记需要在每个子表上单独运行它们。这样的命令：

```
ANALYZE measurement;
```

将只会处理根表。

- 带有ON CONFLICT子句的INSERT语句不太可能按照预期工作，因为只有在指定的目标关系而不是其子关系上发生唯一违背时才会采取ON CONFLICT行动。
- 将会需要触发器或者规则将行路由到想要的子表中，除非应用明确地知道分区的模式。编写触发器可能会很复杂，并且会比声明式分区在内部执行的元组路由慢很多。

## 5.11.4. 分区剪枝

分区剪枝是一种提升声明式分区表性能查询优化技术。例如：

```
SET enable_partition_pruning = on;           -- the default
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

如果没有分区剪枝，上面的查询将会扫描measurement表的每一个分区。如果启用了分区剪枝，规划器将会检查每个分区的定义并且检验该分区是否因为不包含符合查询WHERE子句的行而无需扫描。当规划器可以证实这一点时，它会把分区从查询计划中排除（剪枝）。

通过使用EXPLAIN命令和enable\_partition\_pruning配置参数，可以展示剪枝掉分区的计划与没有剪枝的计划之间的差别。对这种类型的表设置，一种典型的未优化计划是：

```
SET enable_partition_pruning = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
          QUERY PLAN
-----
Aggregate  (cost=188.76..188.77 rows=1 width=8)
  -> Append  (cost=0.00..181.05 rows=3085 width=0)
        -> Seq Scan on measurement_y2006m02  (cost=0.00..33.12 rows=617
width=0)
              Filter: (logdate >= '2008-01-01'::date)
        -> Seq Scan on measurement_y2006m03  (cost=0.00..33.12 rows=617
width=0)
              Filter: (logdate >= '2008-01-01'::date)
...
        -> Seq Scan on measurement_y2007m11  (cost=0.00..33.12 rows=617
width=0)
              Filter: (logdate >= '2008-01-01'::date)
        -> Seq Scan on measurement_y2007m12  (cost=0.00..33.12 rows=617
width=0)
              Filter: (logdate >= '2008-01-01'::date)
        -> Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617
width=0)
              Filter: (logdate >= '2008-01-01'::date)
```

某些或者全部的分区可能会使用索引扫描取代全表顺序扫描，但是这里的重点是根本不需要扫描较老的分区来回答这个查询。当我们启用分区剪枝时，我们会得到一个便宜很多的计划，而它能给出相同的答案：

```
SET enable_partition_pruning = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
          QUERY PLAN
-----
Aggregate  (cost=37.75..37.76 rows=1 width=8)
-> Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
    Filter: (logdate >= '2008-01-01'::date)
```

注意，分区剪枝仅由分区键隐式定义的约束所驱动，而不是由索引的存在驱动。因此，没有必要在键上定义索引。是否需要为一个给定分区创建索引取决于预期的查询扫描该分区时会扫描大部分还是小部分。后一种情况下索引的帮助会比前者大。

不仅在给定查询的规划期间可以执行分区剪枝，在其执行期间也能执行分区剪枝。这非常有用，因为如果子句中包含查询规划时值未知的表达式时，这可以剪枝掉更多的分区；例如在PREPARE语句中定义的参数会使用从子查询拿到的值，或者嵌套循环连接内侧关系上的参数化值。执行期间的分区剪枝可能在下列任何时刻执行：

- 在查询计划的初始化期间。对于执行的初始化阶段就已知值的参数，可以在这里执行分区剪枝。这个阶段中被剪枝掉的分区将不会显示在查询的EXPLAIN或EXPLAIN ANALYZE结果中。通过观察EXPLAIN输出的“Subplans Removed”属性，可以确定被剪枝掉的分区数。
- 在查询计划的实际执行期间。这里可以使用只有在实际查询执行时才能知道的值执行分区剪枝。这包括来自子查询的值以及来自执行时参数的值（例如来自于参数化嵌套循环连接的参数）。由于在查询执行期间这些参数的值可能会改变多次，所以只要分区剪枝使用到的执行参数发生改变，就会执行一次分区剪枝。要判断分区是否在这个阶段被剪枝，需要仔细地观察EXPLAIN ANALYZE输出中的loops属性。对应于不同分区的子计划可以具有不同的值，这取决于在执行期间每个分区被修剪的次数。如果每次都被剪枝，有些分区可能会显示为(never executed)。

可以使用enable\_partition\_pruning设置禁用分区剪枝。

## 5.11.5. 分区和约束排除

约束排除是一种与分区剪枝类似的查询优化技术。虽然它主要被用于使用传统继承方法实现的分区上，但它也可以被用于其他目的，包括用于声明式分区。

约束排除以非常类似于分区剪枝的方式工作，不过它使用每个表的CHECK约束——这也是它得名的原因——而分区剪枝使用表的分区边界，分区边界仅存在于声明式分区的情况中。另一点不同之处是约束排除仅在规划时应用，在执行时不会尝试移除分区。

由于约束排除使用CHECK约束，这导致它比分区剪枝要慢，但有时候可以被当作一种优点加以利用：因为甚至可以在声明式分区的表上（在分区边界之外）定义约束，约束排除可能可以从查询计划中消去额外的分区。

constraint\_exclusion的默认（也是推荐的）设置不是on也不是off，而是一种被称为partition的中间设置，这会导致该技术仅被应用于可能工作在继承分区表上的查询。on设置导致规划器检查所有查询中的CHECK约束，甚至是那些不太可能受益的简单查询。

下列提醒适用于约束排除：

- 约束排除仅适用于查询规划期间，和分区裁剪不同，在查询执行期间也可以应用。

- 只有查询的WHERE子句包含常量（或者外部提供的参数）时，约束排除才能有效果。例如，针对一个非不变函数（如CURRENT\_TIMESTAMP）的比较不能被优化，因为规划器不知道该函数的值在运行时会落到哪个子表中。
- 保持分区约束简单化，否则规划器可能无法验证哪些子表可能不需要被访问。如前面的例子所示，对列表分区使用简单的等值条件，对范围分区使用简单的范围测试。一种好的经验规则是分区约束应该仅包含分区列与常量使用B-树的可索引操作符的比较，因为只有B-树的可索引列才允许出现在分区键中。
- 约束排除期间会检查父表的所有子表上的所有约束，因此大量的子表很可能明显地增加查询规划时间。因此，传统的基于继承的分区可以很好地处理上百个子表，不要尝试使用上千个子表。

### 5.11.6. 声明分区最佳实践

应该谨慎地选择如何划分表，因为查询规划和执行的性能可能会受到不良设计的负面影响。

最重要的设计决策之一是选择对数据进行分区的列或者列的组合。通常最佳选择是按最常出现在分区表上执行的查询的WHERE子句中的列或列集合进行分区。与分区键匹配并兼容的WHERE子句项可用于裁剪不需要的分区。但是，你可能会被迫根据PRIMARY KEY或UNIQUE约束的要求做出其他决策。在规划分区策略时，删除不需要的数据也是需要考虑的一个因素。可以相当快地分离整个分区，因此采用这种方式设计分区策略可能是有益的，既把一次删除的所有数据都放在单个分区中。

选择表应该划分的分区的目标数量也是一个重要的决策。没有足够的分区可能意味着索引仍然太大，数据位置仍然较差，这可能导致缓存命中率很低。但是，将表划分为太多的分区也会导致问题。在查询规划和执行期间，过多的分区可能意味着查询计划时间较长，内存消耗也更高，见下面进一步的描述。在选择如何划分表时，考虑将来可能发生的更改也很重要。例如，如果您选择为每个客户提供一个分区，而您目前只有少量的大客户，那么，如果几年后您发现自己有大量的小客户，那么就要考虑这种影响。在这种情况下，最好选择按HASH分区并且选择合理数量的分区，而不是尝试按LIST进行分区，并希望客户数量的增长不会超出按数据分区的实际范围。

子分区有助于进一步划分那些预计会比其他分区更大的分区。另一种选择是使用分区键中有多列的范围分区。这两种情况都很容易导致分区数量过多，因此建议进行限制。

考虑查询计划和执行期间的分区开销也很重要。查询规划器通常能够很好地处理多达几千个分区的分区层次结构，前提是典型的查询允许查询规划器裁剪除了少量分区之外的所有分区。规划器执行分区修剪后保留更多分区时，规划时间会变长，内存消耗会更高。担心拥有大量分区的另一个原因是，服务器的内存消耗可能会随着时间的推移而显著增加，特别是如果许多会话接触大量分区。这是因为每个分区都需要将其元数据加载到接触它的每个会话的本地内存中。

对于数据仓库类型工作负载，使用比OLTP类型工作负载更多的分区数量很有意义。通常，在数据仓库中，查询计划时间不太值得关注，因为大多数处理时间都花在查询执行期间。对于这两种类型的工作负载，尽早做出正确的决策非常重要，因为重新分区大量数据可能会非常缓慢。模拟预期工作负载通常有利于优化分区策略。永远不要只是假设更多的分区比更少的分区更好，反之亦然。

## 5.12. 外部数据

LightDB实现了部分的SQL/MED规定，允许我们使用普通SQL查询来访问位于LightDB之外的数据。这种数据被称为外部数据（注意这种用法不要和外键混淆，后者是数据库中的一种约束）。

外部数据可以在一个外部数据包装器的帮助下被访问。一个外部数据包装器是一个库，它可以与一个外部数据源通讯，并隐藏连接到数据源和从它获取数据的细节。在contrib模块中有一些外部数据包装器，参见Appendix H 其他类型的外部数据包装器可以在第三方产品中

找到。如果这些现有的外部数据包装器都不能满足你的需要，可以自己编写一个，参见Chapter 53

要访问外部数据，我们需要建立一个外部服务器对象，它根据它所支持的外部数据包装器所使用的一组选项定义了如何连接到一个特定的外部数据源。接着我们需要创建一个或多个外部表，它们定义了外部数据的结构。一个外部表可以在查询中像一个普通表一样地使用，但是在LightDB服务器中外部表没有存储数据。不管使用什么外部数据包装器，LightDB会要求外部数据包装器从外部数据源获取数据，或者在更新命令的情况下传送数据到外部数据源。

访问远程数据可能需要在外部数据源的授权。这些信息通过一个用户映射提供，它基于当前的LightDB角色提供了附加的数据例如用户名和密码。

更多信息请见 CREATE FOREIGN DATA WRAPPER、CREATE SERVER、CREATE USER MAPPING、CREATE FOREIGN TABLE、以及 IMPORT FOREIGN SCHEMA。

## 5.13. 其他数据库对象

表是一个关系型数据库结构中的核心对象，因为它们承载了我们的数据。但是它们并不是数据库中的唯一一种对象。有很多其他种类的对象可以被创建来使得数据的使用和刮泥更加方便或高效。在本章中不会讨论它们，但是我们会给出一个列表：

- 视图
- 函数、过程和操作符
- 数据类型和域
- 触发器和重写规则

这些主题的详细信息请见Part V

## 5.14. 依赖跟踪

当我们创建一个涉及到很多具有外键约束、视图、触发器、函数等的表的复杂数据库结构时，我们隐式地创建了一张对象之间的依赖关系网。例如，具有一个外键约束的表依赖于它所引用的表。

为了保证整个数据库结构的完整性，LightDB确保我们无法删除仍然被其他对象依赖的对象。例如，尝试删除Section 5.4.5中的产品表会导致一个如下的错误消息，因为有订单表依赖于产品表：

```
DROP TABLE products;
```

```
ERROR: cannot drop table products because other objects depend on it
DETAIL: constraint orders_product_no_fkey on table orders depends on table
products
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

该错误消息包含了一个有用的提示：如果我们不想一个一个去删除所有的依赖对象，我们可以执行：

```
DROP TABLE products CASCADE;
```

这样所有的依赖对象将被移除，同样依赖于它们的任何对象也会被递归删除。在这种情况下，订单表不会被移除，但是它的外键约束会被移除。之所以在这里会停下，是因为没有什么依赖着外键约束（如果希望检查DROP ... CASCADE会干什么，运行不带CASCADE的DROP并阅读DETAIL输出）。

LightDB中的几乎所有DROP命令都支持CASCADE。当然，其本质的区别随着对象的类型而不同。我们也可以用RESTRICT代替CASCADE来获得默认行为，它将阻止删除任何被其他对象依赖的对象。

### Note

根据SQL标准，在DROP命令中指定RESTRICT或CASCADE是被要求的。但没有哪个数据库系统真正强制了这个规则，但是不同的系统中两种默认行为都是可能的。

如果一个DROP命令列出了多个对象，只有在存在指定对象构成的组之外的依赖关系时才需要CASCADE。例如，如果发出命令DROP TABLE tab1, tab2且存在从tab2到tab1的外键引用，那么就不需要CASCADE即可成功执行。

对于用户定义的函数，LightDB会追踪与函数外部可见性质相关的依赖性，例如它的参数和结果类型，但不追踪检查函数体才能知道的依赖性。例如，考虑这种情况：

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
                             'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
'SELECT note FROM my_colors WHERE color = $1'
LANGUAGE SQL;
```

(SQL语言函数的解释见Section 36.5。LightDB将会注意到get\_color\_note函数依赖于rainbow类型：删掉该类型会强制删除该函数，因为该函数的参数类型就无法定义了。但是LightDB不会认为get\_color\_note依赖于my\_colors表，因此即使该表被删除也不会删除这个函数。虽然这种方法有缺点，但是也有好处。如果该表丢失，这个函数在某种程度上仍然是有效的，但是执行它会导致错误。创建一个同名的新表将允许该函数重新有效。

## 5.15. LightDB DDL 规格检查

当您使用DDL语句时，LightDB将首先检查您的语句是否符合规范

详见 `lt_sql_inspect`。



---

## Chapter 6. 数据操纵

前面的章节讨论了如何创建表和其他结构来保存你的数据。现在是时候给表填充数据了。本章涉及如何插入、更新和删除表数据。在接下来的一章将最终解释如何把你丢失已久的数据从数据库中抽取出来。

### 6.1. 插入数据

当一个表被创建后，它不包含任何数据。在数据库发挥作用之前，首先要做的是插入数据。一次插入一行数据。你也可以在一个命令中插入多行，但不能插入不完整的行。即使只知道其中一些列的值，也必须创建完整的行。

要创建一个新行，使用INSERT命令。这条命令要求提供表的名称和其中列的值。例如，考虑Chapter 5 中的产品表：

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric  
);
```

一个插入一行的命令将是：

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

数据的值是按照这些列在表中出现的顺序列出的，并且用逗号分隔。通常，数据的值是文字（常量），但也允许使用标量表达式。

上面的语法的缺点是你必须知道表中列的顺序。要避免这个问题，你也可以显式地列出列。例如，下面的两条命令都有和上文那条命令一样的效果：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);  
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

许多用户认为明确列出列的名字是个好习惯。

如果你没有获得所有列的值，那么你可以省略其中的一些。在这种情况下，这些列将被填充为它们的缺省值。例如：

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');  
INSERT INTO products VALUES (1, 'Cheese');
```

第二种形式是LightDB的一个扩展。它从使用给出的值从左开始填充列，有多少个给出的列值就填充多少个列，其他列的将使用缺省值。

为了保持清晰，你也可以显式地要求缺省值，用于单个的列或者用于整个行：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);  
INSERT INTO products DEFAULT VALUES;
```

你可以在一个命令中插入多行：

```
INSERT INTO products (product_no, name, price) VALUES
```

```
(1, 'Cheese', 9.99),
(2, 'Bread', 1.99),
(3, 'Milk', 2.99);
```

也可以插入查询的结果（可能没有行、一行或多行）：

```
INSERT INTO products (product_no, name, price)
SELECT product_no, name, price FROM new_products
WHERE release_date = 'today';
```

这提供了用于计算要插入的行的SQL查询机制（Chapter 7 的全部功能。

### Tip

在一次性插入大量数据时，考虑使用COPY命令。它不如INSERT命令那么灵活，但是更高效。参考Section 14. 获取更多有关批量装载性能的信息。

## 6.2. 更新数据

修改已经存储在数据库中的数据的行为叫做更新。你可以更新单个行，也可以更新表中所有的行，还可以更新其中的一部分行。我们可以独立地更新每个列，而其他的列则不受影响。

要更新现有的行，使用UPDATE命令。这需要提供三部分信息：

1. 表的名字和要更新的列名
2. 列的新值
3. 要更新的是哪（些）行

我们在Chapter 5 说过，SQL 通常并不为行提供唯一标识符。因此我们无法总是直接指定需要更新哪一行。但是，我们可以通过指定一个被更新的行必须满足的条件。只有在表里面存在主键的时候（不管你声明它还是不声明它），我们才能可靠地通过选择一个匹配主键的条件来指定一个独立的行。图形化的数据库访问工具就靠这允许我们独立地更新某些行。

例如，这条命令把所有价格为5的产品的价格更新为10：

```
UPDATE products SET price = 10 WHERE price = 5;
```

这样做可能导致零行、一行或者更多行被更新。如果我们试图做一个不匹配任何行的更新，那也不算错误。

让我们仔细看看这个命令。首先是关键字UPDATE，然后跟着表名字。和平常一样，表名字也可以是用模式限定的，否则会从路径中查找它。然后是关键字SET，后面跟着列名、一个等号以及新的列值。新的列值可以是任意标量表达式，而不仅仅是常量。例如，如果你想把所有产品的价格提高 10%，你可以用：

```
UPDATE products SET price = price * 1.10;
```

如你所见，用于新值的表达式也可以引用行中现有的值。我们还忽略了WHERE子句。如果我们忽略了这个子句，那么就意味着表中的所有行都要被更新。如果出现了WHERE子句，那么只有匹配它后面的条件的行被更新。请注意在SET子句中的等号是一个赋值，而在WHERE子句中的等号是比较，不过这样并不会导致任何歧义。当然WHERE条件不一定非得是等值测试。许多其他操作符也都可以使用（参阅Chapter 9）。但是表达式必须得出一个布尔结果。

你还可以在一个UPDATE命令中更新更多的列，方法是在SET子句中列出更多赋值。例如：

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

## 6.3. 删除数据

到目前为止我们已经解释了如何向表中增加数据以及如何改变数据。剩下的是讨论如何删除不再需要的数据。和前面增加数据一样，你也只能从表中整行整行地删除数据。在前面一节里我们解释了 SQL 不提供直接访问单个行的方法。因此，删除行只能是通过指定被删除行必须匹配的条件进行。如果你在表上有一个主键，那么你可以指定准确的行。但是你也可以删除匹配条件的一组行，或者你可以一次从表中删除所有的行。

可以使用DELETE命令删除行，它的语法和UPDATE命令非常类似。例如，要从产品表中删除所有价格为 10 的产品，使用：

```
DELETE FROM products WHERE price = 10;
```

如果你只是写：

```
DELETE FROM products;
```

那么表中所有行都会被删除！程序员一定要注意。

## 6.4. 从修改的行中返回数据

有时在修改行的操作过程中获取数据很有用。INSERT、UPDATE和DELETE命令都有一个支持这个的可选的 RETURNING子句。使用RETURNING 可以避免执行额外的数据库查询来收集数据，并且在否则难以可靠地识别修改的行时尤其有用。

所允许的RETURNING子句的内容与SELECT命令的输出列表相同（请参阅Section 7.3）。它可以包含命令的目标表的列名，或者包含使用这些列的值表达式。一个常见的简写是RETURNING \*，它按顺序选择目标表的所有列。

在INSERT中，可用于RETURNING的数据是插入的行。这在琐碎的插入中并不是很有用，因为它只会重复客户端提供的数据。但依赖于计算出的默认值时可以非常方便。例如，当使用serial列来提供唯一标识符时，RETURNING可以返回分配给新行的ID：

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);
```

```
INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

RETURNING子句对于INSERT ... SELECT也非常有用。

在UPDATE中，可用于RETURNING的数据是被修改行的新内容。例如：

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

在DELETE中，可用于RETURNING的数据是删除行的内容。例如：

```
DELETE FROM products
  WHERE obsolescence_date = 'today'
```

RETURNING \*;

如果目标表上有触发器(Chapter 37), 可用于RETURNING 的数据是被触发器修改的行。因此, 检查由触发器计算的列是 RETURNING的另一个常见用例。

## 6.5. LightDB DML规格检查

当您使用DML语句时, LightDB将首先检查您的语句是否符合规范。

详见 `lt_sql_inspect`。

---

## Chapter 7. 查询

前面的章节解释了如何创建表、如何用数据填充它们 以及如何操纵那些数据。现在我们终于可以讨论如何从数据库中检索数据了。

### 7.1. 概述

从数据库中检索数据的过程或命令叫做查询。在 SQL 里SELECT命令用于指定查询。SELECT命令的一般语法是

```
[WITH with_queries] SELECT select_list FROM table_expression  
[sort_specification]
```

下面几个小节描述选择列表、表表达式和排序声明的细节。WITH查询等高级特性将在最后讨论。

一个简单类型的查询的形式：

```
SELECT * FROM table1;
```

假设有一个表叫做table1，这条命令将table1中检索所有行和所有用户定义的列（检索的方法取决于客户端应用。例如，ltsql程序将在屏幕上显示一个 ASCII 形式的表格，而客户端库将提供函数来从检索结果中抽取单个值）。选择列表声明\*意味着所有表表达式提供的列。一个选择列表也可以选择可用列的一个子集或者在使用它们之前对列进行计算。例如，如果table1有叫做a、b和c的列（可能还有其他），那么你可以用下面的查询：

```
SELECT a, b + c FROM table1;
```

（假设b和c都是数字数据类型）。参阅Section 7. 获取更多细节。

FROM table1是一种非常简单的表表达式：它只读取了一个表。通常，表表达式可以是基本表、连接和子查询组成的复杂结构。你也可以省略整个表表达式而把SELECT命令当做一个计算器：

```
SELECT 3 * 4;
```

如果选择列表里的表达式返回变化的结果，那么这就更有用了。例如，你可以用这种方法调用函数：

```
SELECT random();
```

### 7.2. 表表达式

表表达式计算一个表。该表表达式包含一个FROM子句，该子句后面可以根据需要选用WHERE、GROUP BY和HAVING子句。最简单的表表达式只是引用磁盘上的一个表，一个所谓的基本表，但是我们可以用更复杂的表表达式以多种方法修改或组合基本表。

表表达式里可选的WHERE、GROUP BY和HAVING子句指定一系列对源自FROM子句的表的转换操作。所有这些转换最后生成一个虚拟表，它提供行传递给选择列表计算查询的输出行。

#### 7.2.1. FROM子句

FROM子句从一个用逗号分隔的表引用列表中的一个或多个其它表中生成一个表。

```
FROM table_reference [, table_reference [, ...]]
```

表引用可以是一个表名字（可能有模式限定）或者是一个生成的表，例如子查询、一个JOIN结构或者这些东西的复杂组合。如果在FROM子句中引用了多于一个表，那么它们被交叉连接（即构造它们的行的笛卡尔积，见下文）。FROM列表的结果是一个中间的虚拟表，该表可以进行由WHERE、GROUP BY和HAVING子句指定的转换，并最后生成全局的表表达式结果。

如果一个表引用是一个简单的表名字并且它是表继承层次中的父表，那么该表引用将产生该表和它的后代表中的行，除非你在该表名字前面放上ONLY关键字。但是，这种引用只会产生出现在该命名表中的列 — 在子表中增加的列都会被忽略。

除了在表名前写ONLY，你可以在表名后面写上\*来显式地指定要包括所有的后代表。没有实际的理由再继续使用这种语法，因为搜索后代表在总是默认行为。不过，为了保持与旧版本的兼容性，仍然支持这种语法。

### 7.2.1.1. 连接表

一个连接表是根据特定的连接类型的规则从两个其它表（真实表或生成表）中派生的表。目前支持内连接、外连接和交叉连接。一个连接表的一般语法是：

```
T1 join_type T2 [ join_condition ]
```

所有类型的连接都可以被链在一起或者嵌套：T1和T2都可以是连接表。在JOIN子句周围可以使用圆括号来控制连接顺序。如果不使用圆括号，JOIN子句会从左至右嵌套。

#### 连接类型

##### 交叉连接

```
T1 CROSS JOIN T2
```

对来自于T1和T2的行的每一种可能的组合（即笛卡尔积），连接表将包含这样一行：它由所有T1里面的列后面跟着所有T2里面的列构成。如果两个表分别有 N 和 M 行，连接表将有  $N * M$  行。

FROM T1 CROSS JOIN T2等效于FROM T1 INNER JOIN T2 ON TRUE（见下文）。它也等效于FROM T1, T2。

#### Note

当多于两个表出现时，后一种等效并不严格成立，因为JOIN比逗号绑得更紧。例如FROM T1 CROSS JOIN T2 INNER JOIN T3 ON condition和FROM T1, T2 INNER JOIN T3 ON condition并不完全相同，因为第一种情况中的condition可以引用T1，但在第二种情况中却不行。

##### 条件连接

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
  ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column
  list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

INNER和OUTER对所有连接形式都是可选的。INNER是缺省；LEFT、RIGHT和FULL指示一个外连接。

连接条件在ON或USING子句中指定，或者用关键字NATURAL隐含地指定。连接条件决定来自两个源表中的哪些行是“匹配”的，这些我们将在后文详细解释。

可能的条件连接类型是：

#### INNER JOIN

对于 T1 的每一行 R1，生成的连接表都有一行对应 T2 中的每一个满足和 R1 的连接条件的行。

#### LEFT OUTER JOIN

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。因此，生成的连接表里为来自 T1 的每一行都至少包含一行。

#### RIGHT OUTER JOIN

首先，执行一次内连接。然后，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。因此，生成的连接表里为来自 T2 的每一行都至少包含一行。

#### FULL OUTER JOIN

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。同样，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。

ON子句是最常见的连接条件的形式：它接收一个和WHERE子句里用的一样的布尔值表达式。如果两个分别来自T1和T2的行在ON表达式上运算的结果为真，那么它们就算是匹配的行。

USING是个缩写符号，它允许你利用特殊的情况：连接的两端都具有相同的连接列名。它接受共享列名的一个逗号分隔列表，并且为其中每一个共享列构造一个包含等值比较的连接条件。例如用USING (a, b)连接T1和T2会产生连接条件ON T1.a = T2.a AND T1.b = T2.b。

更进一步，JOIN USING的输出会废除冗余列：不需要把匹配上的列都打印出来，因为它们必须具有相等的值。不过JOIN ON会先产生来自T1的所有列，后面跟上所有来自T2的列；而JOIN USING会先为列出的每一个列对产生一个输出列，然后先跟上来自T1的剩余列，最后跟上来自T2的剩余列。

最后，NATURAL是USING的缩写形式：它形成一个USING列表，该列表由那些在两个表里都出现了的列名组成。和USING一样，这些列只在输出表里出现一次。如果不存在公共列，NATURAL JOIN的行为将和JOIN ... ON TRUE一样产生交叉集连接。

#### Note

USING对于连接关系中的列改变是相当安全的，因为只有被列出的列会被组合成连接条件。NATURAL的风险更大，因为如果其中一个关系的模式改变会导致出现一个新的匹配列名，就会导致连接将新列也组合成连接条件。

为了解释这些问题，假设我们有一个表t1:

num	name
1	a
2	b
3	c

和t2:

num	value
1	xxx
3	yyy
5	zzz

然后用不同的连接方式可以获得各种结果:

=> SELECT \* FROM t1 CROSS JOIN t2;

num	name	num	value
1	a	1	xxx
1	a	3	yyy
1	a	5	zzz
2	b	1	xxx
2	b	3	yyy
2	b	5	zzz
3	c	1	xxx
3	c	3	yyy
3	c	5	zzz

(9 rows)

=> SELECT \* FROM t1 INNER JOIN t2 ON t1.num = t2.num;

num	name	num	value
1	a	1	xxx
3	c	3	yyy

(2 rows)

=> SELECT \* FROM t1 INNER JOIN t2 USING (num);

num	name	value
1	a	xxx
3	c	yyy

(2 rows)

=> SELECT \* FROM t1 NATURAL INNER JOIN t2;

num	name	value
1	a	xxx
3	c	yyy

(2 rows)

=> SELECT \* FROM t1 LEFT JOIN t2 ON t1.num = t2.num;

num	name	num	value
1	a	1	xxx
2	b		
3	c	3	yyy



(3 rows)

```
=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
```

num	name	value
1	a	xxx
2	b	
3	c	yyy

(3 rows)

```
=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
```

num	name	num	value
1	a	1	xxx
3	c	3	yyy
		5	zzz

(3 rows)

```
=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
```

num	name	num	value
1	a	1	xxx
2	b		
3	c	3	yyy
		5	zzz

(4 rows)

用ON指定的连接条件也可以包含与连接不直接相关的条件。这种功能可能对某些查询很有用，但是需要我们仔细想清楚。例如：

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
```

num	name	num	value
1	a	1	xxx
2	b		
3	c		

(3 rows)

注意把限制放在WHERE子句中会产生不同的结果：

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
```

num	name	num	value
1	a	1	xxx

(1 row)

这是因为放在ON子句中的一个约束在连接之前被处理，而放在WHERE子句中的一个约束是在连接之后被处理。这对内连接没有关系，但是对于外连接会带来麻烦。

## 7.2.1.2. 表和列别名

你可以给一个表或复杂的表引用指定一个临时的名字，用于剩下的查询中引用那些派生的表。这被叫做表别名。

要创建一个表别名，我们可以写：

```
FROM table_reference AS alias
```

或者

```
FROM table_reference alias
```

AS关键字是可选的。别名可以是任意标识符。

表别名的典型应用是给长表名赋予比较短的标识符，好让连接子句更易读。例如：

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON
    s.id = a.num;
```

到这里，别名成为当前查询的表引用的新名称——我们不再能够用该表最初的名字引用它了。因此，下面的用法是不合法的：

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;    -- 错误
```

表别名主要用于简化符号，但是当把一个表连接到它自身时必须使用别名，例如：

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id =
    child.mother_id;
```

此外，如果一个表引用是一个子查询，则必须要使用一个别名（见Section 7.2.1.3）。

圆括弧用于解决歧义。在下面的例子中，第一个语句将把别名b赋给my\_table的第二个实例，但是第二个语句把别名赋给连接的结果：

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

另外一种给表指定别名的形式是给表的列赋予临时名字，就像给表本身指定别名一样：

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

如果指定的列别名比表里实际的列少，那么剩下的列就没有被重命名。这种语法对于自连接或子查询特别有用。

如果用这些形式中的任何一种给一个JOIN子句的输出附加了一个别名，那么该别名就在JOIN的作用下隐去了其原始的名字。例如：

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

是合法 SQL，但是：

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

是不合法的：表别名a在别名c外面是看不到的。

### 7.2.1.3. 子查询

子查询指定了一个派生表，它必须被包围在圆括弧里并且必须被赋予一个表别名（参阅Section 7.2.1.2）。例如：

```
FROM (SELECT * FROM table1) AS alias_name
```

这个例子等效于FROM table1 AS alias\_name。更有趣的情况是在子查询里面有分组或聚集的时候，子查询不能被简化为一个简单的连接。

一个子查询也可以是一个VALUES列表：

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
      AS names(first, last)
```

再次的，这里要求一个表别名。为VALUES列表中的列分配别名是可选的，但是选择这样做是一个好习惯。更多信息可参见Section 7.7

### 7.2.1.4. 表函数

表函数是那些生成一个行集合的函数，这个集合可以由基本数据类型（标量类型）组成，也可以是由复合数据类型（表行）组成。它们的用法类似一个表、视图或者在查询的FROM子句里的子查询。表函数返回的列可以像一个表列、视图或者子查询那样被包含在SELECT、JOIN或WHERE子句里。

也可以使用ROWS FROM语法将平行列返回的结果组合成表函数；这种情况下结果行的数量是最大一个函数结果的数量，较小的结果会用空值来填充。

```
function_call [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ])]]
ROWS FROM( function_call [, ... ] ) [WITH ORDINALITY] [[AS] table_alias
[(column_alias [, ... ])]]
```

如果指定了WITH ORDINALITY子句，一个额外的 bigint类型的列将会被增加到函数的结果列中。这个列对函数结果集的行进行编号，编号从1开始（这是对SQL标准语法UNNEST... WITH ORDINALITY的一般化）。默认情况下，序数列被称为ordinality，但也可以通过使用一个AS子句给它分配一个不同的列名。

调用特殊的表函数UNNEST可以使用任意数量的数组参数，它会返回对应的列数，就好像在每一个参数上单独调用UNNEST (Section 9.19) 并且使用ROWS FROM结构把它们组合起来。

```
UNNEST( array_expression [, ... ] ) [WITH ORDINALITY] [[AS] table_alias
[(column_alias [, ... ])]]
```

如果没有指定table\_alias，该函数名将被用作表名。在ROWS FROM()结构的情况中，会使用第一个函数名。

如果没有提供列的别名，那么对于一个返回基数据类型的函数，列名也与该函数名相同。对于一个返回组合类型的函数，结果列会从该类型的属性得到名称。

例子：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
```

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM getfoo(1) AS t1;
```

```
SELECT * FROM foo
    WHERE foosubid IN (
```

```
SELECT foosubid
FROM getfoo(foo.fooid) z
WHERE z.fooid = foo.fooid
);
```

```
CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);
```

```
SELECT * FROM vw_getfoo;
```

参数的 有时候，定义一个能够根据它们被调用方式返回不同列集合的表函数是很有用的。为了支持这些，表函数可以被声明为返回没有OUT参数的伪类型record。如果在查询里使用这样的函数，那么我们必须要在查询中指定所预期的行结构，这样系统才知道如何分析和规划该查询。这种语法是这样的：

```
function_call [AS] alias (column_definition [, ... ])
function_call AS [alias] (column_definition [, ... ])
ROWS FROM( ... function_call AS (column_definition [, ... ]) [, ... ])
```

在没有使用ROWS FROM()语法时， column\_definition列表会取代无法附着在 FROM项上的列别名列表，列定义中的名称就起到列别名的作用。在使用ROWS FROM()语法时，可以为每一个成员函数单独附着一个 column\_definition列表；或者在只有一个成员函数并且没有WITH ORDINALITY子句的情况下，可以在 ROWS FROM()后面写一个 column\_definition列表来取代一个列别名列表。

考虑下面的例子：

```
SELECT *
FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
AS tl(proname name, prosrc text)
WHERE proname LIKE 'bytea%';
```

dblink函数（dblink模块的一部分）执行一个远程的查询。它被声明为返回record，因为它可能会被用于任何类型的查询。实际的列集必须在调用它的查询中指定，这样分析器才知道类似\*这样的东西应该扩展成什么样子。

此示例使用ROWS FROM：

```
SELECT *
FROM ROWS FROM
(
    json_to_recordset(' [{"a":40,"b":"foo"}, {"a":100,"b":"bar"} ]')
    AS (a INTEGER, b TEXT),
    generate_series(1, 3)
) AS x (p, q, s)
ORDER BY p;
```

p	q	s
40	foo	1
100	bar	2
		3

它将两个函数连接到一个FROM目标中。 json\_to\_recordset() 被指示返回两列，第一个integer 和第二个text。generate\_series()的结果直接使用。 ORDER BY子句将列值排序为整数。

### 7.2.1.5. LATERAL子查询

可以在出现于FROM中的子查询前放置关键词LATERAL。这允许它们引用前面的FROM项提供的列（如果没有LATERAL，每一个子查询将被独立计算，并且因此不能被其他FROM项交叉引用）。

出现在FROM中的表函数的前面也可以被放上关键词LATERAL，但对于函数该关键词是可选的，在任何情况下函数的参数都可以包含对前面的FROM项提供的列的引用。

一个LATERAL项可以出现在FROM列表顶层，或者出现在一个JOIN树中。在后一种情况下，如果它出现在JOIN的右部，那么它也可以引用在JOIN左部的任何项。

如果一个FROM项包含LATERAL交叉引用，计算过程如下：对于提供交叉引用列的FROM项的每一行，或者多个提供这些列的多个FROM项的行集合，LATERAL项将被使用该行或者行集中的列值进行计算。得到的结果行将和它们被计算出来的行进行正常的连接。对于来自这些列的源表的每一行或行集，该过程将重复。

LATERAL的一个简单例子：

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

这不是非常有用，因为它和一种更简单的形式得到的结果完全一样：

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

在必须要使用交叉引用列来计算那些即将要被连接的行时，LATERAL是最有用的。一种常用的应用是为一个返回集合的函数提供一个参数值。例如，假设vertices(polygon)返回一个多边形的顶点集合，我们可以这样标识存储在一个表中的多边形中靠近的顶点：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

这个查询也可以被写成：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

或者写成其他几种等价的公式（正如以上提到的，LATERAL关键词在这个例子中并不是必不可少的，但是我们在这里使用它是为了使表述更清晰）。

有时候也会很特别地把LEFT JOIN放在一个LATERAL子查询的前面，这样即使LATERAL子查询对源行不产生行，源行也会出现在结果中。例如，如果get\_product\_names()返回一个制造商制造的产品名字，但是某些制造商在我们的表中目前没有制造产品，我们可以找出哪些制造商是这样：

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

## 7.2.2. WHERE子句

WHERE子句的语法是

WHERE search\_condition

这里的search\_condition是任意返回一个boolean类型值的值表达式（参阅Section 4.2）。

在完成对FROM子句的处理之后，生成的虚拟表的每一行都会对根据搜索条件进行检查。如果该条件的结果是真，那么该行被保留在输出表中；否则（也就是说，如果结果是假或空）就把它抛弃。搜索条件通常至少要引用一些在FROM子句里生成的列；虽然这不是必须的，但如果不引用这些列，那么WHERE子句就没什么用了。

### Note

内连接的连接条件既可以写在WHERE子句也可以写在JOIN子句里。例如，这些表表达式是等效的：

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

和：

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

或者可能还有：

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

你想用哪个只是一个风格问题。FROM子句里的JOIN语法可能不那么容易移植到其它SQL数据库管理系统中。对于外部连接而言没有选择：它们必须在FROM子句中完成。外部连接的ON或USING子句不等于WHERE条件，因为它导致最终结果中行的增加（对那些不匹配的输入行）和减少。

这里是一些WHERE子句的例子：

```
SELECT ... FROM fdt WHERE c1 > 5
```

```
SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
```

```
SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
AND 100
```

```
SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

在上面的例子里，fdt是从FROM子句中派生的表。那些不符合WHERE子句的搜索条件的行会被从fdt中删除。请注意我们把标量子查询当做一个值表达式来用。和任何其它查询一样，子查询里可以使用复杂的表表达式。同时还请注意fdt在子查询中也被引用。只有在c1也是作为子查询输入表的生成表的列时，才必须把c1限定成fdt.c1。但限定列名字可以增加语句的清晰度，即使有时候不是必须的。这个例子展示了一个外层查询的列名范围如何扩展到它的内层查询。

## 7.2.3. GROUP BY和HAVING子句

在通过了WHERE过滤器之后，生成的输入表可以使用GROUP BY子句进行分组，然后用HAVING子句删除一些分组行。

```
SELECT select_list
      FROM ...
      [WHERE ...]
      GROUP BY grouping_column_reference [, grouping_column_reference]...
```

GROUP BY子句被用来把表中在所列出的列上具有相同值的行分组在一起。这些列的列出顺序并没有什么关系。其效果是把每组具有相同值的行组合为一个组行，它代表该组里的所有行。这样就可以删除输出里的重复和/或计算应用于这些组的聚集。例如：

```
=> SELECT * FROM test1;
```

```
x | y
---+---
a | 3
c | 2
b | 5
a | 1
(4 rows)
```

```
=> SELECT x FROM test1 GROUP BY x;
```

```
x
---
a
b
c
(3 rows)
```

在第二个查询里，我们不能写成SELECT \* FROM test1 GROUP BY x，因为列y里没有哪个值可以和每个组相关联起来。被分组的列可以在选择列表中引用是因为它们在每个组都有单一的值。

通常，如果一个表被分了组，那么没有在GROUP BY中列出的列都不能被引用，除非在聚集表达式中被引用。一个用聚集表达式的例子是：

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
```

```
x | sum
---+-----
a | 4
b | 5
c | 2
(3 rows)
```

这里的sum是一个聚集函数，它在整个组上计算出一个单一值。有关可用的聚集函数的更多信息可以在Section 9.21

### Tip

没有聚集表达式的分组实际上计算了一个列中可区分值的集合。我们也可以使用DISTINCT子句实现（参阅Section 7.3.3）。

这里是另外一个例子：它计算每种产品的总销售额（而不是所有产品的总销售额）：

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
   FROM products p LEFT JOIN sales s USING (product_id)
   GROUP BY product_id, p.name, p.price;
```

在这个例子里，列product\_id、p.name和p.price必须在GROUP BY子句里，因为它们都在查询的选择列表里被引用到（但见下文）。列s.units不必在GROUP BY列表里，因为它只是在一个聚集表达式（sum(...)）里使用，它代表一组产品的销售额。对于每种产品，这个查询都返回一个该产品的所有销售额的总和行。

如果产品表被建立起来，例如product\_id是主键，那么在上面的例子中用product\_id来分组就够了，因为名称和价格都是函数依赖于产品ID，并且关于为每个产品ID分组返回哪个名称和价格值就不会有歧义。

在严格的 SQL 里，GROUP BY只能对源表的列进行分组，但LightDB把这个扩展为也允许GROUP BY去根据选择列表中的列分组。也允许对值表达式进行分组，而不仅是简单的列名。

如果一个表已经用GROUP BY子句分了组，然后你又只对其中的某些组感兴趣，那么就可以用HAVING子句，它很象WHERE子句，用于从结果中删除一些组。其语法是：

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

在HAVING子句中的表达式可以引用分组的表达式和未分组的表达式（后者必须涉及一个聚集函数）。

例子：

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
```

x	sum
a	4
b	5

(2 rows)

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
```

x	sum
a	4
b	5

(2 rows)

再次，一个更现实的例子：

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
   FROM products p LEFT JOIN sales s USING (product_id)
   WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
   GROUP BY product_id, p.name, p.price, p.cost
   HAVING sum(p.price * s.units) > 5000;
```

在上面的例子里，WHERE子句用那些非分组的列选择数据行（表达式只是对那些最近四周发生的销售为真）。而HAVING子句限制输出为总销售收入超过 5000 的组。请注意聚集表达式不需要在查询中的所有地方都一样。

如果一个查询包含聚集函数调用，但是没有GROUP BY子句，分组仍然会发生：结果是一个单一行（或者根本就没有行，如果该单一行被HAVING所消除）。它包含一个HAVING子句时也是这样，即使没有任何聚集函数调用或者GROUP BY子句。



## 7.2.4. GROUPING SETS、CUBE和ROLLUP

使用分组集的概念可以实现比上述更加复杂的分组操作。由 FROM和WHERE子句选出的数据被按照每一个指定的分组集单独分组，按照简单GROUP BY子句对每一个分组计算 聚集，然后返回结果。例如：

```
=> SELECT * FROM items_sold;
```

brand	size	sales
Foo	L	10
Foo	M	20
Bar	M	15
Bar	L	5

(4 rows)

```
=> SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING SETS
((brand), (size), ());
```

brand	size	sum
Foo		30
Bar		20
	L	15
	M	35
		50

(5 rows)

GROUPING SETS的每一个子列表可以指定一个或者多个列或者表达式， 它们将按照直接出现在GROUP BY子句中同样的方式被解释。一个空的 分组集表示所有的行都要被聚集到一个单一分组（即使没有输入行存在也会被输出） 中，这就像前面所说的没有GROUP BY子句的聚集函数的情况一样。

对于分组列或表达式没有出现在其中的分组集的结果行，对分组列或表达式的引用会被空值所替代。要区分一个特定的输出行来自于哪个分组，请见 Table 9.60

LightDB 中提供了一种简化方法来指定两种常用类型的分组集。下面形式的子句

```
ROLLUP ( e1, e2, e3, ... )
```

表示给定的表达式列表及其所有前缀（包括空列表），因此它等效于

```
GROUPING SETS (
  ( e1, e2, e3, ... ),
  ...
  ( e1, e2 ),
  ( e1 ),
  ( )
)
```

这通常被用来分析历史数据，例如按部门、区和公司范围计算的总薪水。

下面形式的子句

```
CUBE ( e1, e2, ... )
```

表示给定的列表及其可能的子集（即幂集）。因此

CUBE ( a, b, c )

等效于

```
GROUPING SETS (
    ( a, b, c ),
    ( a, b   ),
    ( a,   c ),
    ( a     ),
    (   b, c ),
    (   b   ),
    (     c ),
    (     )
)
```

CUBE或ROLLUP子句中的元素可以是表达式或者圆括号中的元素子列表。在后一种情况中，对于生成分组集的目的来说，子列表被当做单一单元来对待。例如：

CUBE ( (a, b), (c, d) )

等效于

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b       ),
    (     c, d   ),
    (           )
)
```

并且

ROLLUP ( a, (b, c), d )

等效于

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c   ),
    ( a         ),
    (           )
)
```

CUBE和ROLLUP可以被直接用在 GROUP BY子句中，也可以被嵌套在一个 GROUPING SETS子句中。如果一个 GROUPING SETS子句被嵌套在另一个同类子句中，效果和把内层子句的所有元素直接写在外层子句中一样。

如果在一个GROUP BY子句中指定了多个分组项，那么最终的分组集列表是这些项的叉积。例如：

GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))

等效于

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),   (a, b, e),
    (a, c, d),   (a, c, e),
    (a, d),     (a, e)
)
```

### Note

在表达式中，结构(a, b)通常被识别为一个 a 行构造器。在 GROUP BY子句中，这不会在表达式的顶层应用，并且 (a, b)会按照上面所说的被解析为一个表达式的列表。如果出于某种原因你在分组表达式中需要一个行构造器，请使用 ROW(a, b)。

## 7.2.5. 窗口函数处理

如果查询包含任何窗口函数（见Section 3.5 Section 9.2和Section 4.2.8，这些函数将在任何分组、聚集和HAVING过滤被执行之后被计算。也就是说如果查询使用了任何聚集、GROUP BY或HAVING，则窗口函数看到的行是分组行而不是来自于FROM/WHERE的原始表行。

当多个窗口函数被使用，所有在窗口定义中有句法上等效的PARTITION BY和ORDER BY子句的窗口函数被保证在数据上的同一趟扫描中计算。因此它们将会看到相同的排序顺序，即使ORDER BY没有唯一地决定一个顺序。但是，对于具有不同PARTITION BY或ORDER BY定义的函数的计算没有这种保证（在这种情况下，在多个窗口函数计算之间通常要求一个排序步骤，并且并不保证保留行的顺序，即使它的ORDER BY把这些行视为等效的）。

目前，窗口函数总是要求排序好的数据，并且这样查询的输出总是被根据窗口函数的PARTITION BY/ORDER BY子句的一个或者另一个排序。但是，我们不推荐依赖于此。如果你希望确保结果以特定的方式排序，请显式使用顶层的ORDER BY子句。

## 7.3. 选择列表

如前面的小节说明的那样，在SELECT命令里的表表达式构造了一个中间的虚拟表，方法可能有组合表、视图、消除行、分组等等。这个表最后被选择列表传递下去处理。选择列表判断中间表的哪个列是实际输出。

### 7.3.1. 选择列表项

最简单的选择列表类型是\*，它发出表表达式生成的所有列。否则，一个选择列表是一个逗号分隔的值表达式的列表（和在Section 4.2定义的一样）。例如，它可能是一个列名的列表：

```
SELECT a, b, c FROM ...
```

列名字a、b和c要么是在FROM子句里引用的表中列的实际名字，要么是像Section 7.2.1.2解释的那样的别名。在选择列表里可用的名字空间和WHERE子句里的一样，除非你使用了分组，这时候它和HAVING子句一样。

如果超过一个表有同样的列名，那么你还必须给出表名字，如：

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

在使用多个表时，要求一个特定表的所有列也是有用的：

```
SELECT tbl1.*, tbl2.a FROM ...
```

更多有关`table_name.*`记号的内容请参考Section 8.16.5

如果将任意值表达式用于选择列表，那么它在概念上向返回的表中增加了一个新的虚拟列。值表达式为结果的每一行进行一次计算，对任何列引用替换行的值。不过选择列表中的这个表达式并非一定要引用来自FROM子句中表表达式里面的列，例如它也可以是任意常量算术表达式。

### 7.3.2. 列标签

选择列表中的项可以被赋予名字，用于进一步的处理。例如为了在一个ORDER BY子句中使用或者为了客户端应用显示。例如：

```
SELECT a AS value, b + c AS sum FROM ...
```

如果没有使用AS指定输出列名，那么系统会分配一个缺省的列名。对于简单的列引用，它 是被引用列的名字。对于函数调用，它是函数的名字。对于复杂表达式，系统会生成一个通用的名字。

只有在新列无法匹配任何LightDB关键词（见Appendix B 时，AS关键词是可选的。为了避免一个关键字的意外匹配，你可以使用双引号来修饰列名。例如，VALUE是一个关键字，所以下面的语句不会工作：

```
SELECT a value, b + c AS sum FROM ...
```

但是这个可以：

```
SELECT a "value", b + c AS sum FROM ...
```

为了防止未来可能的关键词增加，我们推荐总是写AS或者用双引号修饰输出列名。

#### Note

输出列的命名和在FROM子句里的命名是不一样的（参阅Section 7.2.1.2）。它实际上允许你对同一个列命名两次，但是在选择列表中分配的名字是要传递下去的名字。

### 7.3.3. DISTINCT

在处理完选择列表之后，结果表可以可选的删除重复行。我们可以直接在SELECT后面写上DISTINCT关键字来指定：

```
SELECT DISTINCT select_list ...
```

（如果不用DISTINCT你可以用ALL关键词来指定获得的所有行的缺省行为）。

显然，如果两行里至少有一个列有不同的值，那么我们认为它是可区分的。空值在这种比较中被认为是相同的。

另外，我们还可以用任意表达式来判断什么行可以被认为是可区分的：

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

这里expression是任意值表达式，它为所有行计算。如果一个行集合里所有表达式的值是一样的，那么我们认为它们是重复的并且因此只有第一行保留在输出中。请注意这里的一个集合的“第一行”是不可预料的，除非你在足够多的列上对该查询排了序，保证到达DISTINCT过滤器的行的顺序是唯一的（DISTINCT ON处理是发生在ORDER BY排序后面的）。

DISTINCT ON子句不是 SQL 标准的一部分，有时候有人认为它是一个糟糕的风格，因为它的结果是不可判定的。如果有选择的使用GROUP BY和在FROM中的子查询，那么我们可以避免使用这个构造，但是通常它是更方便的候选方法。

## 7.4. 组合查询

两个查询的结果可以用集合操作并、交、差进行组合。语法是

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

其中query1和query2是可以使用到目前为止讨论的任何功能的查询。

UNION有效地把query2的结果附加到query1的结果上（不过我们不能保证这就是这些行实际被返回的顺序）。此外，它将删除结果中所有重复的行，就象DISTINCT做的那样，除非你使用了UNION ALL。

INTERSECT返回那些同时存在于query1和query2的结果中的行，除非声明了INTERSECT ALL，否则所有重复行都被消除。

EXCEPT返回所有在query1的结果中但是不在query2的结果中的行（有时候这叫做两个查询的差）。同样的，除非声明了EXCEPT ALL，否则所有重复行都被消除。

为了计算两个查询的并、交、差，这两个查询必须是“并操作兼容的”，也就意味着它们都返回同样数量的列，并且对应的列有兼容的数据类型，如Section 10.5中描述的那样。

可以组合使用集合操作，例如

```
query1 UNION query2 EXCEPT query3
```

它等价于

```
(query1 UNION query2) EXCEPT query3
```

如此所示，您可以使用括号来控制评估顺序。没有括号，UNION和EXCEPT从左到右关联，但INTERSECT比这两个运算符的优先级更高。因此，

```
query1 UNION query2 INTERSECT query3
```

意思是

```
query1 UNION (query2 INTERSECT query3)
```

您还可以使用括号括起单个query。如果query需要使用以下部分中讨论的任何子句（例如LIMIT），这一点非常重要。如果没有括号，您将会得到语法错误，或者该子句将被理解为应用于集合操作的输出而不是其输入之一。例如，

```
SELECT a FROM b UNION SELECT x FROM y LIMIT 10
```

是被接受的，但它的意思是

```
(SELECT a FROM b UNION SELECT x FROM y) LIMIT 10
```

而不是

```
SELECT a FROM b UNION (SELECT x FROM y LIMIT 10)
```

## 7.5. 行排序

在一个查询生成一个输出表之后（在处理完选择列表之后），还可以选择性地对它进行排序。如果没有选择排序，那么行将以未指定的顺序返回。这时候的实际顺序将取决于扫描和连接计划类型以及行在磁盘上的顺序，但是肯定不能依赖这些东西。一种特定的顺序只能在显式地选择了排序步骤之后才能被保证。

ORDER BY子句指定了排序顺序：

```
SELECT select_list
      FROM table_expression
      ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
             [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

排序表达式可以是任何在查询的选择列表中合法的表达式。一个例子是：

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

当多于一个表达式被指定，后面的值将被用于排序那些在前面值上相等的行。每一个表达式后可以选择性地放置一个ASC或DESC关键词来设置排序方向为升序或降序。ASC顺序是默认值。升序会把较小的值放在前面，而“较小”则由<操作符定义。相似地，降序则由>操作符定义。<sup>1</sup>

NULLS FIRST和NULLS LAST选项将可以被用来决定在排序顺序中，空值是出现在非空值之前或者出现在非空值之后。默认情况下，排序时空值被认为比任何非空值都要大，即NULLS FIRST是DESC顺序的默认值，而不是NULLS LAST的默认值。

注意顺序选项是对每一个排序列独立考虑的。例如ORDER BY x, y DESC表示ORDER BY x ASC, y DESC，而和ORDER BY x DESC, y DESC不同。

一个sort\_expression也可以是列标签或者一个输出列的编号，如：

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

两者都根据第一个输出列排序。注意一个输出列的名字必须孤立，即它不能被用在一个表达式中 — 例如，这是不正确的：

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;           -- 错误
```

该限制是为了减少混淆。如果一个ORDER BY项是一个单一名字并且匹配一个输出列名或者一个表表达式的列，仍然会出现混淆。在这种情况下输出列将被使用。只有在你使用AS来重命名一个输出列来匹配某些其他表列的名字时，这才会导致混淆。

<sup>1</sup> 事实上，LightDB为表达式的数据类型使用默认B-tree操作符类来决定ASC和DESC的排序顺序。照惯例，数据类型将被建立，这样<和>操作符负责这个排序顺序，但是一个用户定义的数据类型的设计者可以选择做些不同的设置。

ORDER BY可以被应用于UNION、INTERSECT或EXCEPT组合的结果，但是在这种情况下它只被允许根据输出列名或编号排序，而不能根据表达式排序。

## 7.6. LIMIT和OFFSET

LIMIT和OFFSET允许你只检索查询剩余部分产生的行的一部分：

```
SELECT select_list
  FROM table_expression
  [ ORDER BY ... ]
  [ LIMIT { number | ALL } ] [ OFFSET number ]
```

```
SELECT select_list
  FROM table_expression
  [ ORDER BY ... ]
  [ LIMIT [ number, ] { number | ALL } ]
```

如果给出了一个限制计数，那么会返回数量不超过该限制的行（但可能更少些，因为查询本身可能生成的行数就比较少）。LIMIT ALL的效果和省略LIMIT子句一样，就像是LIMIT带有NULL 参数一样。

OFFSET说明在开始返回行之前忽略多少行。OFFSET 0的效果和省略OFFSET子句是一样的，并且LIMIT NULL的效果和省略LIMIT子句一样，就像是OFFSET带有 NULL 参数一样。

如果OFFSET和LIMIT都出现了，那么在返回LIMIT个行之前要先忽略OFFSET行。

如果使用LIMIT，那么用一个ORDER BY子句把结果行约束成一个唯一的顺序是很重要的。否则你就会拿到一个不可预料的该查询的行的子集。你要的可能是第十到第二行，但以什么顺序的第十到第二行？除非你指定了ORDER BY，否则顺序是不知道的。

查询优化器在生成查询计划时会考虑LIMIT，因此如果你给定LIMIT和OFFSET，那么你可能收到不同的规划（产生不同的行顺序）。因此，使用不同的LIMIT/OFFSET值选择查询结果的不同子集将生成不一致的结果，除非你用ORDER BY强制一个可预测的顺序。这并非bug，这是一个很自然的结果，因为SQL没有许诺把查询的结果按照任何特定的顺序发出，除非用了ORDER BY来约束顺序。

被OFFSET子句忽略的行仍然需要在服务器内部计算；因此，一个很大的OFFSET的效率可能还是不够高。

为了兼容MySQL，LightDB也支持LIMIT offset, row\_count语法。

## 7.7. VALUES列表

VALUES提供了一种生成“常量表”的方法，它可以被使用在一个查询中而不需要实际在磁盘上创建一个表。语法是：

```
VALUES ( expression [, ...] ) [, ...]
```

每一个被圆括号包围的表达式列表生成表中的一行。列表都必须具有相同数据的元素（即表中列的数目），并且在每个列表中对应的项必须具有可兼容的数据类型。分配给结果的每一列的实际数据类型使用和UNION相同的规则确定（参见Section 10.5）。

一个例子：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

将会返回一个有两列三行的表。它实际上等效于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

在默认情况下，LightDB将column1、column2等名字分配给一个VALUES表的列。这些列名不是由SQL标准指定的，并且不同的数据库系统的做法也不同，因此通常最好使用表别名列表来重写这些默认的名字，像这样：

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t
   (num, letter);
 num | letter
-----+-----
   1 | one
   2 | two
   3 | three
(3 rows)
```

在句法上，后面跟随着表达式列表的VALUES列表被视为和

```
SELECT select_list FROM table_expression
```

一样，并且可以出现在SELECT能出现的任何地方。例如，你可以把它用作UNION的一部分，或者附加一个sort\_specification (ORDER BY、LIMIT和/或OFFSET) 给它。VALUES最常见的用途是作为一个INSERT命令的数据源，以及作为一个子查询。

更多信息请见VALUES。

## 7.8. WITH查询（公共表表达式）

WITH提供了一种方式来书写在一个大型查询中使用的辅助语句。这些语句通常被称为公共表表达式或CTE，它们可以被看成是定义只在一个查询中存在的临时表。在WITH子句中的每一个辅助语句可以是一个SELECT、INSERT、UPDATE或DELETE，并且WITH子句本身也可以被附加到一个主语句，主语句也可以是SELECT、INSERT、UPDATE或DELETE。

### 7.8.1. WITH中的SELECT

WITH中SELECT的基本价值是将复杂的查询分解称为简单的部分。一个例子：

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
```



```
SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

它只显示在高销售区域每种产品的销售总额。WITH子句定义了两个辅助语句regional\_sales和top\_regions，其中regional\_sales的输出用在top\_regions中而top\_regions的输出用在主SELECT查询。这个例子可以不用WITH来书写，但是我们必须要用两层嵌套的子SELECT。使用这种方法要更简单些。

可选的RECURSIVE修饰符将WITH从单纯的句法便利变成了一种在标准SQL中不能完成的特性。通过使用RECURSIVE，一个WITH查询可以引用它自己的输出。一个非常简单的例子是计算从1到100的整数和的查询：

```
WITH RECURSIVE t(n) AS (
  VALUES (1)
  UNION ALL
  SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

一个递归WITH查询的通常形式总是一个非递归项，然后是UNION（或者UNION ALL），再然后是一个递归项，其中只有递归项能够包含对于查询自身输出的引用。这样一个查询可以被这样执行：

### 递归查询求值

1. 计算非递归项。对UNION（不对UNION ALL），抛弃重复行。把所有剩余的行包括在递归查询的结果中，并且也把它们放在一个临时的工作表中。
2. 只要工作表不为空，重复下列步骤：
  - a. 计算递归项，用当前工作表的内容替换递归自引用。对UNION（不是UNION ALL），抛弃重复行以及那些与之前结果行重复的行。将剩下的所有行包括在递归查询的结果中，并且也把它们放在一个临时的中间表中。
  - b. 用中间表的内容替换工作表的内容，然后清空中间表。

#### Note

严格来说，这个处理是迭代而不是递归，但是RECURSIVE是SQL标准委员会选择的术语。

在上面的例子中，工作表在每一步只有一个行，并且它在连续的步骤中取值从1到100。在第100步，由于WHERE子句导致没有输出，因此查询终止。

递归查询通常用于处理层次或者树状结构的数据。一个有用的例子是这个用于找到一个产品的直接或间接部件的查询，只要给定一个显示了直接包含关系的表：

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
  SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
  SELECT p.sub_part, p.part, p.quantity
  FROM included_parts pr, parts p
  WHERE p.part = pr.sub_part
)
```

```
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

在使用递归查询时，确保查询的递归部分最终将不返回元组非常重要，否则查询将会无限循环。在某些时候，使用UNION替代UNION ALL可以通过抛弃与之前输出行重复的行来达到这个目的。不过，经常有循环不涉及到完全重复的输出行：它可能只需要检查一个或几个域来看相同点之前是否达到过。处理这种情况的标准方法是计算一个已经访问过值的数组。例如，考虑下面这个使用link域搜索表graph的查询：

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
    UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

如果link关系包含环，这个查询将会循环。因为我们要求一个“depth”输出，仅仅将UNION ALL 改为UNION不会消除循环。反过来在我们顺着特定链接路径搜索时，我们需要识别我们是否再次到达了一个相同的行。我们可以向这个有循环倾向的查询增加两个列path和cycle：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
    ARRAY[g.id],
    false
    FROM graph g
    UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
    path || g.id,
    g.id = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

除了阻止环，数组值对于它们自己的工作显示到达任何特定行的“path”也有用。

在通常情况下如果需要检查多于一个域来识别一个环，请用行数组。例如，如果我们需要比较域f1和f2：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
    ARRAY[ROW(g.f1, g.f2)],
    false
    FROM graph g
    UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
    path || ROW(g.f1, g.f2),
    ROW(g.f1, g.f2) = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
```

```
SELECT * FROM search_graph;
```

Tip

在通常情况下只有一个域需要被检查来识别一个环，可以省略ROW()语法。这允许使用一个简单的数组而不是一个组合类型数组，可以获得效率。

Tip

递归查询计算算法使用宽度优先搜索顺序产生它的输出。你可以通过让外部查询ORDER BY一个以这种方法构建的“path”，用来以深度优先搜索顺序显示结果。

当你不确定查询是否可能循环时，一个测试查询的有用技巧是在父查询中放一个LIMIT。例如，这个查询没有LIMIT时会永远循环：

```
WITH RECURSIVE t(n) AS (
    SELECT 1
    UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

这会起作用，因为LightDB的实现只计算WITH查询中被父查询实际取到的行。不推荐在生产中使用这个技巧，因为其他系统可能以不同方式工作。同样，如果你让外层查询排序递归查询的结果或者把它们连接成某种其他表，这个技巧将不会起作用，因为在这些情况下外层查询通常将尝试取得WITH查询的所有输出。

WITH查询的一个有用的特性是在每一次父查询的执行中它们通常只被计算一次，即使它们被父查询或兄弟WITH查询引用了超过一次。因此，在多个地方需要的昂贵计算可以被放在一个WITH查询中来避免冗余工作。另一种可能的应用是阻止不希望的多个函数计算产生副作用。但是，从另一方面来看，优化器不能将来自父查询的约束下推到乘法引用WITH查询，因为当他应该只影响一个时它可能会影响所有使用WITH查询的输出的使用。乘法引用WITH查询通常将会被按照所写的方式计算，而不抑制父查询以后可能会抛弃的行（但是，如上所述，如果对查询的引用只请求有限数目的行，计算可能会提前停止）。

但是，如果 WITH 查询是非递归和边际效应无关的（就是说，它是一个SELECT包含没有可变函数），则它可以合并到父查询中，允许两个查询级别的联合优化。默认情况下，这发生在如果父查询仅引用 WITH查询一次的时候，而不是它引用WITH查询多于一次时。你可以超越控制这个决策，通过指定 MATERIALIZED 来强制分开计算 WITH 查询，或者通过指定 NOT MATERIALIZED来强制它被合并到父查询中。后一种选择存在重复计算WITH查询的风险，但它仍然能提供净节省，如果WITH查询的每个使用只需要WITH查询的完整输出的一小部分。

这些规则的一个简单示例是

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w WHERE key = 123;
```

这个 WITH 查询将被合并，生成相同的执行计划为

```
SELECT * FROM big_table WHERE key = 123;
```

特别是，如果在key上有一个索引，它可能只用于获取具有 key = 123的行。另一方面，在

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

WITH查询将被物化，生成一个big\_table的临时拷贝，然后与其自身一联合，这样将不能从索引上获得任何好处。如果写成下面的形式，这个查询将被执行得更有效率。

```
WITH w AS NOT MATERIALIZED (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

所以父查询的限制可以直接应用于big\_table的扫描。

一个NOT MATERIALIZED 可能不理想的例子为

```
WITH w AS (
    SELECT key, very_expensive_function(val) as f FROM some_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.f = w2.f;
```

在这里，WITH查询的物化确保very\_expensive\_function每个表行只计算一次，而不是两次。

以上的例子只展示了和SELECT一起使用的WITH，但是它可以被以相同的方式附加在INSERT、UPDATE或DELETE上。在每一种情况中，它实际上提供了可在主命令中引用的临时表。

## 7.8.2. WITH中的数据修改语句

你可以在WITH中使用数据修改语句（INSERT、UPDATE或DELETE）。这允许你在同一个查询中执行多个不同操作。一个例子：

```
WITH moved_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

这个查询实际上将products中的行移动到products\_log。WITH中的DELETE删除来自products的指定行，以RETURNING子句返回被删除的内容，然后主查询读该输出并将它插入到products\_log。

上述例子中好的一点是WITH子句被附加给INSERT，而没有附加给INSERT的子SELECT。这是必需的，因为WITH中的数据修改语句只能附加到顶层语句。不过，普通WITH的可见性规则仍适用，这样才能从子SELECT中引用WITH语句的输出。

正如上述例子所示，WITH中的数据修改语句通常具有RETURNING子句（见Section 6.4）。它是RETURNING子句的输出，不是数据修改语句的目标表，它形成了其他查询可以引用的临时表。如果一个WITH中的数据修改语句缺少一个RETURNING子句，则它不会形成临时表并且不能被其他查询引用。尽管如此，这样一个语句仍将被执行。一个不是特别有用的例子：

```
WITH t AS (  
    DELETE FROM foo  
)  
DELETE FROM bar;
```

这个例子将从表foo和bar中移除所有行。被报告给客户端的受影响行的数目可能只包括从bar中移除的行。

数据修改语句中不允许递归自引用。在某些情况中可以采取引用一个递归WITH的输出来解决这个限制，例如：

```
WITH RECURSIVE included_parts(sub_part, part) AS (  
    SELECT sub_part, part FROM parts WHERE part = 'our_product'  
    UNION ALL  
    SELECT p.sub_part, p.part  
    FROM included_parts pr, parts p  
    WHERE p.part = pr.sub_part  
)  
DELETE FROM parts  
    WHERE part IN (SELECT part FROM included_parts);
```

这个查询将会移除一个产品的所有直接或间接子部件。

WITH中的数据修改语句只被执行一次，并且总是能结束，而不管主查询是否读取它们所有（或者任何）的输出。注意这和WITH中SELECT的规则不同：正如前一小节所述，直到主查询要求SELECT的输出时，SELECT才会被执行。

WITH中的子语句和其他子语句以及主查询被并发执行。因此在使用WITH中的数据修改语句时，无法预知实际更新顺序。所有的语句都使用同一个snapshot执行（参见Chapter 13），因此它们不能“看见”目标表上另一个执行的效果。这减轻了行更新的实际顺序的不可预见性的影响，并且意味着RETURNING数据是在不同WITH子语句和主查询之间传达改变的唯一方法。其例子

```
WITH t AS (  
    UPDATE products SET price = price * 1.05  
    RETURNING *  
)  
SELECT * FROM products;
```

外层SELECT可以返回在UPDATE动作之前的原始价格，而在

```
WITH t AS (  
    UPDATE products SET price = price * 1.05  
    RETURNING *  
)  
SELECT * FROM t;
```

外部SELECT将返回更新过的数据。

在一个语句中试图两次更新同一行是不被支持的。只会发生一次修改，但是该办法不能很容易地（有时是不可能）可靠预测哪一个会被执行。这也应用于删除一个已经在同一个语句中

被更新过的行：只有更新被执行。因此你通常应该避免尝试在一个语句中两次修改同一行。尤其是避免编写可以作用于被主语句或兄弟子语句修改行的WITH子语句。这样一个语句的效果将是不可预测的。

当前，用作WITH中数据修改语句目标的任何表不能有条件规则、ALSO规则或扩展到多个语句的INSTEAD规则。

## 7.9. ROWNUM

ROWNUM是像Oracle的ROWNUM一样的伪列。伪列的行为类似于表列，但实际上并没有存储在表中。您可以从伪列中进行选择，但无法插入、更新或删除它们的值。伪列也类似于没有参数的函数。然而，没有参数的函数通常对结果集中的每一行返回相同的值，而伪列通常对每一行返回不同的值。

对于查询返回的每一行，ROWNUM伪列返回一个数字，指示LightDB从表或一组连接行中选择行的顺序。第一行选择的ROWNUM为1，第二行为2，以此类推。

您可以使用ROWNUM来限制查询返回的行数，就像这个例子中一样：

```
SELECT * from employees WHERE ROWNUM < 10;
```

如果在同一查询中，在ROWNUM之后有一个ORDER BY子句，那么行将按ORDER BY子句重新排序。

```
SELECT * FROM employees WHERE ROWNUM < 11 ORDER BY last_name;
```

如果您将ORDER BY子句嵌入子查询中，并将ROWNUM条件放置在顶层查询中，则可以强制在对行进行排序之后应用ROWNUM条件。例如，以下查询返回具有最小的10个员工编号的员工。这有时被称为top-N报告：

```
SELECT * FROM  
  (SELECT * FROM employees ORDER BY employee_id)  
  WHERE ROWNUM < 11;
```

在上面的例子中，ROWNUM值是顶层SELECT语句的值，因此它们是在子查询中按employee\_id排序后生成的。

测试ROWNUM值是否大于正整数的条件始终为false。例如，以下查询不返回任何行：

```
SELECT * FROM employees WHERE ROWNUM > 1;
```

获取的第一行被分配一个ROWNUM为1，并使条件为false。要获取的第二行现在是第一行，并且也被分配一个ROWNUM为1，并使条件为false。随后的所有行都无法满足条件，因此不返回任何行。

您也可以在UPDATE和DELETE语句中像在SELECT语句中一样使用ROWNUM。

```
UPDATE employees SET employee_id = 0 WHERE ROWNUM < 10;  
DELETE FROM employees WHERE ROWNUM < 10;
```

请注意，ROWNUM在canopy中不受支持。如果您在canopy中有时称之为top-N报告，请考虑改用limit。

## Chapter 8. 数据类型

LightDB有着丰富的本地数据类型可用。用户可以使用CREATE TYPE命令为 LightDB增加新的数据类型。

Table 8. 显示了所有内建的普通数据类型。大部分在“别名”列里列出的可选名字都是因历史原因 被LightDB在内部使用的名字。另外，还有一些内部使用的或者废弃的类型也可以用，但没有在这里列出。

Table 8.1. 数据类型

名字	别名	描述
bigint	int8	有符号的8字节整数
bigserial	serial8	自动增长的8字节整数
bit [ (n) ]		定长位串
bit varying [ (n) ]	varbit [ (n) ]	变长位串
boolean	bool	逻辑布尔值（真/假）
box		平面上的普通方框
bytea		二进制数据（“字节数组”）
character [ (n) ]	char [ (n) ]	定长字符串
character varying [ (n) ]	varchar [ (n) ]	变长字符串
cidr		IPv4或IPv6网络地址
circle		平面上的圆
date		日历日期（年、月、日）
double precision	float8	双精度浮点数（8字节）
inet		IPv4或IPv6主机地址
integer	int, int4	有符号4字节整数
interval [ fields ] [ (p) ]		时间段
json		文本 JSON 数据
jsonb		二进制 JSON 数据，已分解
line		平面上的无限长的线
lseg		平面上的线段
macaddr		MAC (Media Access Control) 地址
macaddr8		MAC (Media Access Control) 地址 (EUI-64格式)
money		货币数量
numeric [ (p, s) ]	decimal [ (p, s) ]	可选择精度的精确数字
path		平面上的几何路径
pg_lsn		LightDB日志序列号
pg_snapshot		用户级事务ID快照
point		平面上的几何点
polygon		平面上的封闭几何路径
real	float4	单精度浮点数（4字节）
smallint	int2	有符号2字节整数

名字	别名	描述
smallserial	serial2	自动增长的2字节整数
serial	serial4	自动增长的4字节整数
text		变长字符串
time [ (p) ] [ without time zone ]		一天中的时间（无时区）
time [ (p) ] with time zone	timetz	一天中的时间，包括时区
timestamp [ (p) ] [ without time zone ]		日期和时间（无时区）
timestamp [ (p) ] with time zone	timestampz	日期和时间，包括时区
tsquery		文本搜索查询
tsvector		文本搜索文档
txid_snapshot		用户级别事务ID快照(废弃; 参见 pg_snapshot)
uuid		通用唯一标识码
xml		XML数据

### 兼容性

下列类型（或者及其拼写）是SQL指定的：bigint、bit、bit varying、boolean、char、character varying、character、varchar、date、double precision、integer、interval、numeric、decimal、real、smallint、time（有时区或无时区）、timestamp（有时区或无时区）、xml。

每种数据类型都有一个由其输入和输出函数决定的外部表现形式。许多内建的类型有明显的格式。不过，许多类型要么是LightDB所特有的（例如几何路径），要么可能是有几种不同的格式（例如日期和时间类型）。有些输入和输出函数是不可逆的，即输出函数的结果和原始输入比较时可能丢失精度。

## 8.1. 数字类型

数字类型由2、4或8字节的整数以及4或8字节的浮点数和可选精度小数组成。Table 8.2列出了所有可用类型。

Table 8.2. 数字类型

名字	存储尺寸	描述	范围
smallint	2字节	小范围整数	-32768 to +32767
integer	4字节	整数的典型选择	-2147483648 to +2147483647
bigint	8字节	大范围整数	-9223372036854775808 to +9223372036854775807
decimal	可变	用户指定精度，精确	最高小数点前131072位，以及小数点后16383位
numeric	可变	用户指定精度，精确	最高小数点前131072位，以及小数点后16383位



名字	存储尺寸	描述	范围
real	4字节	可变精度，不精确	6位十进制精度
double precision	8字节	可变精度，不精确	15位十进制精度
smallserial	2字节	自动增加的小整数	1到32767
serial	4字节	自动增加的整数	1到2147483647
bigserial	8字节	自动增长的大整数	1到9223372036854775807

数字类型常量的语法在Section 4.1.1描述。数字类型有一整套对应的数学操作符和函数。相关信息请参考 Chapter 9 下面的几节详细描述这些类型。

### 8.1.1. 整数类型

类型smallint、integer和bigint存储各种范围的全部是数字的数，也就是没有小数部分的数字。试图存储超出范围以外的值将导致一个错误。

常用的类型是integer，因为它提供了在范围、存储空间和性能之间的最佳平衡。一般只有在磁盘空间紧张的时候才使用 smallint类型。而只有在integer的范围不够的时候才使用bigint。

SQL只声明了整数类型integer（或int）、smallint和bigint。类型int2、int4和int8都是扩展，也在许多其它SQL数据库系统中使用。

请注意，在MySQL兼容模式下，smallint可以插入和更新boolean（true/false），并使用istru/false，=true/false来判断。此行为与MySQL使用tinyint来存储boolean兼容。导入LightDB后，<type>tinyint</type>将转换为<type>smallint</type>。

请注意，Integer已支持Integer(n)类型，n的取值范围为1~19的整形。当n为1~9时，采用Int4类型；当n为10~19时，采用Int8类型；

### 8.1.2. 任意精度数字

类型numeric可以存储非常多位的数字。我们特别建议将它用于货币金额和其它要求计算准确的数量。numeric值的计算在可能的情况下会得到准确的结果，例如加法、减法、乘法。不过，numeric类型上的算术运算比整数类型或者下一节描述的浮点数类型要慢很多。

在随后的内容里，我们使用了下述术语：一个numeric的precision（精度）是整个数中有效位的总数，也就是小数点两边的位数。numeric的scale（小数位数）是小数部分的数字位数，也就是小数点右边的部分。因此数字 23.5141 的精度为6而小数位数为4。可以认为整数的小数位数为零。

numeric列的最大精度和最大小数位数都是可以配置的。要声明一个类型为numeric的列，你可以用下面的语法：

```
NUMERIC(precision, scale)
```

精度必须为正数，小数位数可以为零或者正数。另外：

```
NUMERIC(precision)
```

选择小数位数为 0 。如果使用

```
NUMERIC
```

在没有指定任何精度或小数位数的情况下，创建一个“无约束的数值”列，其中可以存储任意长度的数值，直到被实现所限制。此类列不会将输入值强制转换为任何特定的小数位数，

而具有声明小数位数的numeric列会将输入值强迫转换为该小数位数。（SQL标准要求默认小数位数为0，即强制为整数精度。我们发现这有点无用。如果你关心可移植性，请始终明确指定精度和小数位数。）

### Note

在NUMERIC类型声明中可以显式指定的最大精度为1000。一个无约束的NUMERIC列服从Table 8.2中描述的限制。

如果一个要存储的值的小数位数比列声明的小数位数高，那么系统将尝试圆整（四舍五入）该值到指定的分数位数。然后，如果小数点左边的位数超过了声明的精度减去声明的小数位数，那么抛出一个错误。

数字值在物理上是以不带任何前导或者后缀零的形式存储。因此，列上声明的精度和小数位数都是最大值，而不是固定分配的（在这个方面，numeric类型更类似于varchar(n)，而不像char(n)）。实际存储要求是每四个十进制位组用两个字节，再加上三到八个字节的开销。

除了普通的numeric值以外，numeric类型还有如下几个特殊值：

Infinity  
-Infinity  
NaN

这些是根据IEEE 754标准改编的，分别表示“infinity”，“negative infinity”，和“not-a-number”。在SQL命令中将它们作为常量写入时，必须在其周围加引号，例如UPDATE table SET x = '-Infinity'。输入时，以不区分大小写的方式识别这些字符串。无穷大值也可以拼写为inf和-inf。

无穷大值的行为符合数学期望。例如，Infinity加上任何有限值等于Infinity，就像Infinity加上Infinity一样；但是Infinity减去Infinity得到NaN（不是一个数字），因为它没有良好定义的解释。请注意，无穷大只能存储在无约束的numeric列中，因为它名义上超过了任何有限精度限制。

NaN（不是数字）值用于表示未定义的计算结果。通常，任何具有NaN输入的操作都会产生另一个NaN。唯一的例外是当操作的其他输入使得如果将NaN替换为任何有限或无限数值时将获得相同的输出，那么，该输出值也适用于NaN。（这个原理的一个例子是，NaN的零次方会得到1。）

### Note

在“不是一个数字”概念的大部分实现中，NaN被认为不等于任何其他数字值（包括NaN）。为了允许numeric值可以被排序和使用基于树的索引，LightDB把NaN值视为相等，并且比所有非NaN值都要大。

类型decimal和numeric是等效的。两种类型都是SQL标准的一部分。

在对值进行圆整时，numeric类型会圆到远离零的整数，而（在大部分机器上）real和double precision类型会圆到最近的偶数上。例如：

```
SELECT x,
       round(x::numeric) AS num_round,
       round(x::double precision) AS dbl_round
```

```
FROM generate_series(-3.5, 3.5, 1) as x;
  x | num_round | dbl_round
-----+-----+-----
-3.5 |         -4 |        -4
-2.5 |         -3 |        -2
-1.5 |         -2 |        -2
-0.5 |         -1 |         0
 0.5 |          1 |          0
 1.5 |          2 |          2
 2.5 |          3 |          2
 3.5 |          4 |          4
(8 rows)
```

### 8.1.3. 浮点类型

数据类型`real`和`double precision`是不精确的、变精度的数字类型。在所有当前支持的平台上，这些类型是IEEE标准 754 二进制浮点算术（分别对应单精度和双精度）的实现，一直到下层处理器、操作系统和支持它的编译器。

不准确意味着一些值不能准确地转换成内部格式并且是以近似的形式存储的，因此存储和检索一个值可能出现一些缺失。处理这些错误以及这些错误是如何在计算中传播的主题属于数学和计算机科学的一个完整的分支，我们不会在这里进一步讨论它，这里的讨论仅限于如下几点：

- 如果你要求准确的存储和计算（例如计算货币金额），应使用`numeric`类型。
- 如果你想用这些类型做任何重要的复杂计算，尤其是那些你对范围情况（无穷、下溢）严重依赖的事情，那你应该仔细评估你的实现。
- 用两个浮点数值进行等值比较不可能总是按照期望地进行。

在所有当前支持的平台上，`real`类型的范围是  $1E-37$  to  $1E+37$ ，精度至少是 6 位小数。`double precision`类型的范围是  $1E-307$  to  $1E+308$ ，精度至少是 15 位数字。太大或者太小的值都会导致错误。如果输入数字的精度太高，那么可能发生四舍五入。太接近零的数字，如果不能体现出与零的区别就会导致下溢错误。

默认情况下，浮点值以其最短精确的十进制表示的文本形式输出；所产生的十进制值与相同二进制精度的任何其他值表示相比，更接近于真实存储的二进制值。（但是，当前输出值永远不会精确地处于两个可表示的值之间，以免输入程序不能正确遵守舍近取整法则。）对于`float8`值，此值最多使用 17 个有效十进制数字，对于`float4`值，最多使用9个数字。

#### Note

生成这种最短精确的输出格式比历史的四舍五入的格式要快得多。

为了与LightDB的较旧版本生成的输出兼容，并允许降低输出精度，可以使用`extra_float_digits`参数选择四舍五入的十进制输出。将值设置为0将恢复以前的默认值，即将值四舍五入为6（对于`float4`）或15（对于`float8`）个有效的十进制数字。设置负值会进一步减少位数。例如-2会将输出分别舍入到4或13位数字。

设置`extra_float_digits`位任何大于 0 的值将选择最短精确格式。

#### Note

需要更精确值的应用需要设置`extra_float_digits`为3以获取更精确值。为了版本之间的最大兼容性，他们可以继续这样做。

除了普通的数字值之外，浮点类型还有几个特殊值：

Infinity  
-Infinity  
NaN

这些分别代表 IEEE 754 特殊值“infinity”、“negative infinity”以及“not-a-number”，如果在 SQL 命令里把这些数值当作常量写，你必须在它们周围放上单引号，例如UPDATE table SET x = '-Infinity'。在输入时，这些字符串是以大小写不敏感的方式识别的。无穷大值也可以拼写为inf和-inf。

#### Note

IEEE754指定NaN不应该与任何其他浮点值（包括NaN）相等。为了允许浮点值被排序或者在基于树的索引中使用，LightDB将NaN值视为相等，并且比所有非NaN值要更大。

LightDB还支持 SQL 标准表示法float和float(p)用于声明非精确的数字类型。在这里，p指定以二进制位表示的最低可接受精度。在选取real类型的时候，LightDB接受float(1)到float(24)，在选取double precision的时候，接受float(25)到float(53)。在允许范围之外的p值将导致一个错误。没有指定精度的float将被当作是double precision。

### 8.1.4. 序数类型

#### Note

这一节描述了LightDB特有的创建一个自增列的方法。另一种方法是使用SQL标准的标识列特性，它在CREATE TABLE中描述。

smallserial、serial和bigserial类型不是真正的类型，它们只是为了创建唯一标识符列而存在的方便符号（类似其它一些数据库中支持的AUTO\_INCREMENT属性）。在目前的实现中，下面一个语句：

```
CREATE TABLE tablename (
    colname SERIAL
);
```

等价于以下语句：

```
CREATE SEQUENCE tablename_colname_seq AS integer;
CREATE TABLE tablename (
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

因此，我们就创建了一个整数列并且把它的缺省值安排为从一个序列发生器取值。应用了一个NOT NULL约束以确保空值不会被插入（在大多数情况下你可能还希望附加一个UNIQUE或者PRIMARY KEY约束避免意外地插入重复的值，但这个不是自动发生的）。最后，该序列被标记为“属于”该列，这样当列或表被删除时该序列也会被删除。

## Note

因为smallserial、serial和bigserial是用序列实现的，所以即使没有删除过行，在出现在列中的序列值可能有“空洞”或者间隙。如果一个从序列中分配的值被用在一行中，即使该行最终没有被成功地插入到表中，该值也被“用掉”了。例如，当插入事务回滚时就会发生这种情况。更多信息参见Section 9.1中的nextval()。

要使用serial列插入序列的下一个数值到表中，请指定serial列应该被赋予其缺省值。我们可以通过在INSERT语句中把该列排除在列列表之外来实现，也可以通过使用DEFAULT关键字来实现。

类型名serial和serial4是等效的：两个都创建integer列。类型名bigserial和serial8也一样，只不过它们创建一个bigint列。如果你预计在表的生存期中使用的标识符数目超过 $2^{31}$ 个，那么你应该使用bigserial。类型名smallserial和serial2也以相同方式工作，只不过它们创建一个smallint列。

为一个serial列创建的序列在所属的列被删除的时候自动删除。你可以在不删除列的情况下删除序列，但是这会强制删除该列的默认值表达式。

## 8.2. 货币类型

money类型存储固定小数精度的货币数字，参阅Table 8.3 小数的精度由数据库的lc\_monetary设置决定。表中展示的范围假设有两个小数位。可接受的输入格式很多，包括整数和浮点数文字，以及常用的货币格式，如'\$1,000.00'。输出通常是最后一种形式，但和区域相关。

Table 8.3. 货币类型

名字	存储尺寸	描述	范围
money	8 bytes	货币额	-92233720368547758.08 到 +92233720368547758.07

由于这种数据类型的输出是区域敏感的，因此将money数据装入到一个具有不同lc\_monetary设置的数据库是不起作用的。为了避免这种问题，在恢复一个转储到一个新数据库中之前，应确保新数据库的lc\_monetary设置和被转储数据库的相同或者具有等效值。

数据类型numeric、int和bigint的值可以被造型成money。从数据类型real和double precision的转换可以通过先造型成numeric来实现，例如：

```
SELECT '12.34'::float8::numeric::money;
```

但是，我们不推荐这样做。浮点数不应该被用来处理货币，因为浮点数可能会有圆整错误。

一个money值可以在不损失精度的情况下被造型成numeric。转换到其他类型可能会丢失精度，并且必须采用两个阶段完成：

```
SELECT '52093.89'::money::numeric::float8;
```

一个money值被一个整数值除的除法结果会被截去分数部分。要得到圆整的结果，可以除以一个浮点值，或者在除法之前把money转换成numeric然后在除法之后转回money（如果要避免精度丢失的风险则后者更好）。当一个money值被另一个money值除时，结果是double precision（即一个纯数字，而不是金额），在除法中货币单位被约掉了。

## 8.3. 字符类型

Table 8.4. 字符类型

名字	描述
character varying(n [BYTE CHAR]), varchar(n [BYTE CHAR])	有限制的变长
character(n [BYTE CHAR]), char(n [BYTE CHAR])	定长, 空格填充
text	无限变长

Table 8. 显示了在LightDB里可用的一般用途的字符类型。

SQL定义了两种基本的字符类型：`character varying(n)`和`character(n)`，其中`n`是一个正整数。两种类型都可以存储最多`n`个字符长的串。试图存储更长的串到这些类型的列里会产生一个错误，除非超出长度的字符都是空白，这种情况下该串将被截断为最大长度（这个看上去有点怪异的例外是SQL标准要求）。如果要存储的串比声明的长度短，类型为`character`的值将会用空白填满；而类型为`character varying`的值将只是存储短些的串。

如果我们明确地把一个值造型成`character varying(n)`或者`character(n)`，那么超长的值将被截断成`n`个字符，而不会抛出错误（这也是SQL标准的要求）。

`varchar(n)`和`char(n)`的概念分别是`character varying(n)`和`character(n)`的别名。没有长度声明词的`character`等效于`character(1)`。如果不带长度说明词使用`character varying`，那么该类型接受任何长度的串。后者是一个LightDB的扩展。

另外，LightDB提供`text`类型，它可以存储任何长度的串。尽管类型`text`不是SQL标准，但是许多其它 SQL 数据库系统也有它。

类型`character`的值物理上都用空白填充到指定的长度`n`，并且以这种方式存储和显示。不过，拖尾的空白被当作是没有意义的，并且在比较两个 `character`类型值时不会考虑它们。在空白有意义的排序规则中，这种行为可能会产生意料之外的结果，例如`SELECT 'a'::CHAR(2) collate "C" < E'a\n'::CHAR(2)`会返回真（即便C区域会认为一个空格比新行更大）。当把一个`character`值转换成其他 字符串类型之一时，拖尾的空白会被移除。请注意，在`character varying`和`text`值里，结尾的空白语意上是有含义的，并且在使用模式匹配（如LIKE和正则表达式）时也会被考虑。

可以存储在这些数据类型中的字符由数据库字符集确定，该数据库字符集在创建数据库时选择。无论特定的字符集是什么，都无法存储代码为零的字符（有时称为NUL）。有关更多信息，请参阅Section 21.3

这些类型的存储需求是 4 字节加上实际的字串，如果是 `character` 的话再加上填充的字节。长的字串将会自动被系统压缩，因此在磁盘上的物理需求可能会更少些。长的数值也会存储在后台表里面，这样它们就不会干扰对短字段值的快速访问。不管怎样，允许存储的最长字串大概是 1 GB。（允许在数据类型声明中出现的 `n` 的最大值比这还小。修改这个行为没有甚么意义，因为在多字节编码下字符和字节的数目可能差别很大。如果你想存储没有特定上限的长字串，那么使用 `text` 或者没有长度声明词的 `character varying`，而不要选择一个任意长度限制。）一个短串（最长126字节）的存储要求是1个字节外加实际的串，该串在`character`情况下包含填充的空白。长一些的串在前面需要4个字节而不是1个字节。长串会被系统自动压缩，这样在磁盘上的物理需求可能会更少。非常长的值也会被存储在背景表中，这样它们不会干扰对较短的列值的快速访问。在任何情况下，能被存储的最长的字符串是1GB（数据类型定义中`n`能允许的最大值比这个值要小。修改它没有用处，因为对于多字节字符编码来说，字符的数量和字节数可能完全不同。如果你想要存储没有指定上限的长串，使用`text`或没有长度声明的`character varying`，而不是给出一个任意长度限制）。

为了兼容Oracle数据库语法，`char`数据类型支持指定 `BYTE` 或者 `CHAR` 关键字。这个特性仅是语法糖。

## Tip

这三种类型之间没有性能差别，只不过是在使用填充空白的类型的时候需要更多存储尺寸，以及在存储到一个有长度约束的列时需要少量额外CPU周期来检查长度。虽然在某些其它的数据库系统里，character(n)有一定的性能优势，但在LightDB里没有。事实上，character(n)通常是这三种类型之中最慢的一个，因为它需要额外的存储开销。在大多数情况下，应该使用text或者character varying。

请参考Section 4.1.2. 获取关于串文本的语法的信息，以及参阅Chapter 获取关于可用操作符和函数的信息。

## Example 8.1. 使用字符类型

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1
```

a	char_length
ok	2

```
CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good ');
INSERT INTO test2 VALUES ('too long');
ERROR: value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit truncation
SELECT b, char_length(b) FROM test2;
```

b	char_length
ok	2
good	5
too l	5

```
create table t1(name character(10 byte));
create table t2(name char(10 char));
create table t3(name varchar(10 char));
create table t4(name character varying(10 byte));
```

**1** 函数char\_length在Section 9. 中讨论。

在LightDB里另外还有两种定长字符类型，在Table 8. 显示。name类型只用于在内部系统目录中存储标识符并且不是给一般用户使用的。该类型长度当前定为 64 字节（63 可用字符加结束符）但在C源代码应该使用常量 NAMEDATALEN引用。这个长度是在编译的时候设置的（因而可以为特殊用途调整），缺省的最大长度在以后的版本可能会改变。类型“char”（注意引号）和 char(1)是不一样的，它只用了一个字节的存储空间。它在系统内部用于系统目录当做简化的枚举类型用。

Table 8.5. 特殊字符类型

名字	存储尺寸	描述
"char"	1字节	单字节内部类型

名字	存储尺寸	描述
name	64字节	用于对象名的内部类型

## 8.4. 二进制数据类型

bytea数据类型允许存储二进制串，参见Table 8.6

Table 8.6. 二进制数据类型

名字	存储尺寸	描述
bytea	1或4字节外加真正的二进制串	变长二进制串

二进制串是一个八位位组（或字节）的序列。二进制串和字符串的区别有两个：首先，二进制串明确允许存储零值的字节以及其它“不可打印的”字节（通常是位于十进制范围32到126之外的字节）。字符串不允许零字节，并且也不允许那些对于数据库的选定字符集编码是非法的任何其它字节值或者字节值序列。第二，对二进制串的操作会处理实际上的字节，而字符串的处理和取决于区域设置。简单说，二进制串适用于存储那些程序员认为是“裸字节”的数据，而字符串适合存储文本。

bytea类型支持两种用于输入和输出的格式：“十六进制”格式和LightDB的历史的“转义”格式。在输入时这两种格式总是会被接受。输出格式则取决于配置参数bytea\_output，其默认值为十六进制（注意十六进制格式是在LightDB 9.0中被引入的，早期的版本和某些工具无法理解它）。

SQL标准定义了一种不同的二进制串类型，叫做BLOB或者BINARY LARGE OBJECT。其输入格式和bytea不同，但是提供的函数和操作符大多一样。

### 8.4.1. bytea的十六进制格式

“十六进制”格式将二进制数据编码为每个字节2个十六进制位，最高有效位在前。整个串以序列\x开头（用以和转义格式区分）。在某些情景中，开头的反斜线可能需要通过双写来转义，详见(see Section 4.1.2.1)。作为输入，十六进制位可以是大写也可以是小写，在位对之间可以有空白（但是在位对内部以及开头的\x序列中不能有空白）。十六进制格式和很多外部应用及协议相兼容，并且其转换速度要比转义格式更快，因此人们更愿意用它。

例子：

```
SELECT '\xDEADBEEF';
```

### 8.4.2. bytea的转义格式

“转义”格式是bytea类型的传统LightDB格式。它采用将二进制串表示成ASCII字符序列的方法，而将那些无法用ASCII字符表示的字节转换成特殊的转义语句。从应用的角度来看，如果将字节表示为字符有意义，那么这种表示将很方便。但是在实际中，这常常是令人困扰的，因为它使二进制串和字符串之间的区别变得模糊，并且这种特别的转义机制也有点难于处理。因此这种格式可能会在大部分新应用中避免使用。

在转义模式下输入bytea值时，某些值的字节必须被转义，而所有的字节值都可以被转义。通常，要转义一个字节，需要把它转换成与它的三位八进制值，并且前导一个反斜线。反斜线本身（十进制字节值92）也可以用双写的反斜线表示。Table 8.显示了必须被转义的字符，并给出了可以使用的替代转义序列。

Table 8.7. bytea文字转义字节

十进制字节值	描述	转义输入表示	例子	十六进制表示
0	0字节	'\000'	'\000'::bytea	\x00



十进制字节值	描述	转义输入表示	例子	十六进制表示
39	单引号	''' 或 '\047'	''' ::bytea	\x27
92	反斜线	'\ ' 或 '\134'	'\ ' ::bytea	\x5c
0到31和127到255	“不可打印的”字节	'\xxx' (八进制值)	'\001' ::bytea	\x01

转义“不可打印的”字节的要求取决于区域设置。在某些实例中，你可以不理睬它们，让它们保持未转义的状态。

如Table 8.7所示，要求单引号必须写两次的原因对任何SQL命令中的字符串常量都是一样的。文字解析器消耗最外层的单引号，并缩减成对的单引号为一个普通数据字符。bytea输入函数看到的只是一个单引号，它将其视为普通数据字符。但是，bytea输入函数将反斜杠视为特殊字符，Table 8.7中显示的其他行为由该函数实现。

在某些情况下，反斜杠必须加倍，如上所示，因为通用的字符串文字解析器也会将一对反斜杠减少为一个数据字符；请参阅Section 4.1.2.1

Bytea字节默认被输出为hex格式。如果你把bytea\_output改为escape，“不可打印的”字节会被转换成与之等效的三位八进制值并且前置一个反斜线。大部分“可打印的”字节被输出为它们在客户端字符集中的标准表示形式，例如：

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124' ::bytea;
       bytea
-----
abc klm *\251T
```

十进制值为92（反斜线）的字节在输出时被双写。详情请见Table 8.8

Table 8.8. bytea输出转义字节

十进制字节值	描述	转义的输出表示	例子	输出结果
92	反斜线	\\	'\134' ::bytea	\\
0到31和127到255	“不可打印的”字节	\xxx (八进制值)	'\001' ::bytea	\001
32到126	“可打印的”字节	客户端字符集表示	'\176' ::bytea	~

根据你使用的LightDB前端，你在转义和未转义bytea串方面可能需要做额外的工作。例如，如果你的接口自动翻译换行和回车，你可能也不得不转义它们。

## 8.5. 日期/时间类型

LightDB支持SQL中所有的日期和时间类型，如Table 8.9所示。这些数据类型上可用的操作如Section 9.9所述。日期根据公历来计算，即使对于该历法被引入之前的年份也一样（见Section D.6）。

Table 8.9. 日期/时间类型

名字	存储尺寸	描述	最小值	最大值	解析度
timestamp [ (p) ]	8字节	包括日期和时间（无时区）	4713 BC	294276 AD	1微秒

名字	存储尺寸	描述	最小值	最大值	解析度
[ without time zone ]					
timestamp [ (p) ] with time zone	8字节	包括日期和时间，有时区	4713 BC	294276 AD	1微秒
date	4字节	日期（没有一天中的时间）	4713 BC	5874897 AD	1日
time [ (p) ] [ without time zone ]	8字节	一天中的时间（无日期）	00:00:00	24:00:00	1微秒
time [ (p) ] with time zone	12字节	仅仅是一天中的时间（没有日期），带有时区	00:00:00+1459	24:00:00-1459	1微秒
interval [ fields ] [ (p) ]	16字节	时间间隔	-178000000年	178000000年	1微秒

### Note

SQL要求只写timestamp等效于timestamp without time zone，并且LightDB鼓励这种行为。timestamptz被接受为timestamp with time zone的一种简写，这是一种LightDB的扩展。

time、timestamp和interval接受一个可选的精度值 p，这个精度值声明在秒域中小数点之后保留的位数。缺省情况下，在精度上没有明确的边界。p允许的范围是从 0 到 6。

interval类型有一个附加选项，它可以通过写下面之一的短语来限制存储的fields的集合：

YEAR  
MONTH  
DAY  
HOUR  
MINUTE  
SECOND  
YEAR TO MONTH  
DAY TO HOUR  
DAY TO MINUTE  
DAY TO SECOND  
HOUR TO MINUTE  
HOUR TO SECOND  
MINUTE TO SECOND

注意如果fields和p被指定，fields必须包括SECOND，因为精度只应用于秒。

类型time with time zone是 SQL 标准定义的，但是该定义显示出了一些会影响可用性的性质。在大多数情况下，date、time、timestamp without time zone和timestamp with time zone的组合就应该能提供任何应用所需的全范围的日期/时间功能。

## 8.5.1. 日期/时间输入

日期和时间的输入可以接受几乎任何合理的格式，包括 ISO 8601、SQL-兼容的、传统POSTGRES的和其他的形式。对于一些格式，日期输入里的日、月和年的顺序会让人混

淆，并且支持指定所预期的这些域的顺序。把DateStyle参数设置为MDY，就是选择“月一日一年”的解释，设置为DMY就是“日一月一年”，而YMD是“年一月一日”。

LightDB在处理日期/时间输入上比SQL标准要求的更灵活。参阅Appendix 获取关于日期/时间输入的准确的分析规则和可识别文本域，包括月份、星期几和时区。

请记住任何日期或者时间的文字输入需要由单引号包围，就象一个文本字符串一样。参考Section 4.1.2. 获取更多信息。SQL要求下面的语法

```
type [ (p) ] 'value'
```

其中p是一个可选的精度声明，它给出了在秒域中的小数位数。精度可以被指定给time、timestamp和interval类型，并且可以取从0到6的值。这允许前文所述的值。如果在一个常数声明中没有指定任何精度，它将默认取文字值的精度（但不能超过6位）。

### 8.5.1.1. 日期

Table 8.1显示了date类型可能的输入方式。

Table 8.10. 日期输入

例子	描述
1999-01-08	ISO 8601; 任何模式下的1月8日（推荐格式）
January 8, 1999	在任何datestyle输入模式下都无歧义
1/8/1999	MDY模式中的1月8日; DMY模式中的8月1日
1/18/1999	MDY模式中的1月18日; 在其他模式中被拒绝
01/02/03	MDY模式中的2003年1月2日; DMY模式中的2003年2月1日; YMD模式中的2001年2月3日
1999-Jan-08	任何模式下的1月8日
Jan-08-1999	任何模式下的1月8日
08-Jan-1999	任何模式下的1月8日
99-Jan-08	YMD模式中的1月8日, 否则错误
08-Jan-99	1月8日, 除了在YMD模式中错误
Jan-08-99	1月8日, 除了在YMD模式中错误
19990108	ISO 8601; 任何模式中的1999年1月8日
990108	ISO 8601; 任何模式中的1999年1月8日
1999.008	年和一年中的日子
J2451187	儒略日期
January 8, 99 BC	公元前99年

### 8.5.1.2. 时间

当日时间类型是time [ (p) ] without time zone和time [ (p) ] with time zone。只写time等效于time without time zone。

这些类型的有效输入由当日时间后面跟着可选的时区组成（参阅Table 8.1和Table 8.12）。如果在time without time zone的输入中指定了时区，那么它会被无声地忽略。你也可以指定一个日期但是它会被忽略，除非你使用了一个涉及到夏令时规则的时区，例如America/New\_York。在这种情况下，为了判断是应用了标准时间还是夏令时时间，要求指定该日期。适当的时区偏移被记录在time with time zone值中。

Table 8.11. 时间输入

例子	描述
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	和04:05一样，AM并不影响值
04:05 PM	和16:05一样，输入的小时必须为 ≤ 12
04:05:06.789-8	ISO 8601, 时区为UTC偏移
04:05:06-08:00	ISO 8601, 时区为UTC偏移
04:05-08:00	ISO 8601, 时区为UTC偏移
040506-08	ISO 8601, 时区为UTC偏移
040506+0730	ISO 8601, 以分数小时时区作为UTC偏移
040506+07:30:00	UTC偏移量指定为秒 (ISO 8601中不允许)
04:05:06 PST	缩写指定的时区
2003-04-12 04:05:06 America/New_York	全名指定的时区

Table 8.12. 时区输入

例子	描述
PST	缩写 (太平洋标准时间)
America/New_York	完整时区名
PST8PDT	POSIX风格的时区声明
-8:00:00	PST的UTC偏移
-8:00	PST的UTC偏移量 (ISO 8601扩展格式)
-800	PST的UTC偏移量 (ISO 8601基本格式)
-8	PST的UTC偏移量 (ISO 8601基本格式)
zulu	UTC的军方缩写
z	zulu的缩写形式 (也在ISO 8601中)

参考Section 8.5.可以了解如何指定时区。

### 8.5.1.3. 时间戳

时间戳类型的有效输入由一个日期和时间的串接组成，后面跟着一个可选的时区，一个可选的AD或者BC (另外，AD/BC可以出现在时区前面，但这个顺序并非最佳)。因此：

1999-01-08 04:05:06

和：

1999-01-08 04:05:06 -8:00

都是有效的值，它遵循ISO 8601 标准。另外，使用广泛的格式：

January 8 04:05:06 1999 PST

也被支持。

SQL标准通过“+”或者“-”符号的存在以及时间后面的时区偏移来区分timestamp without time zone和timestamp with time zone文字。因此，根据标准，

TIMESTAMP '2004-10-19 10:23:54'

是一个timestamp without time zone，而

TIMESTAMP '2004-10-19 10:23:54+02'

是一个timestamp with time zone。LightDB从来不会在确定文字串的类型之前检查其内容，因此会把上面两个都看做是 timestamp without time zone。因此要保证把上面的文字当作timestamp with time zone看待，就要给它正确的显式类型：

TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'

如果一个文字已被确定是timestamp without time zone，LightDB将不声不响忽略任何其中指出的时区。即，结果值是从输入值的日期/时间域衍生出来的，并且没有就时区进行调整。

对于timestamp with time zone，内部存储的值总是 UTC（全球统一时间，以前也叫格林威治时间GMT）。如果一个输入值有明确的时区声明，那么它将用该时区合适的偏移量转换成 UTC。如果在输入串里没有时区声明，那么它就被假设是在系统的TimeZone参数里的那个时区，然后使用这个 timezone时区的偏移转换成 UTC。

如果一个timestamp with time zone值被输出，那么它总是从 UTC 转换成当前的timezone时区，并且显示为该时区的本地时间。要看其它时区的时间，要么修改timezone，要么使用AT TIME ZONE构造（参阅Section 9.9.3）。

在timestamp without time zone和timestamp with time zone之间的转换通常假设timestamp without time zone值应该以timezone本地时间的形式接受或者写出。为该转换指定一个不同的可以用AT TIME ZONE。

### 8.5.1.4. 特殊值

为了方便，LightDB支持一些特殊日期/时间输入值，如Table 8.13所示。这些值中infinity和-infinity被在系统内部以特殊方式表示并且将被原封不动地显示。但是其他的仅仅只是概念上的速写，当被读到的时候会被转换为正常的日期/时间值（特殊地，now及相关串在被读到时立刻被转换到一个指定的时间值）。在作为常量在SQL命令中使用，所有这些值需要被包括在单引号内。

Table 8.13. 特殊日期/时间输入

输入串	合法类型	描述
epoch	date, timestamp	1970-01-01 00:00:00+00 (Unix系统时间0)
infinity	date, timestamp	比任何其他时间戳都晚
-infinity	date, timestamp	比任何其他时间戳都早
now	date, time, timestamp	当前事务的开始时间
today	date, timestamp	今日午夜 (00:00)

输入串	合法类型	描述
tomorrow	date, timestamp	明日午夜 (00:00)
yesterday	date, timestamp	昨日午夜 (00:00)
allballs	time	00:00:00.00 UTC

下列SQL-兼容的函数可以被用来为相应的数据类型获得当前时间值：  
CURRENT\_DATE、CURRENT\_TIME、CURRENT\_TIMESTAMP、LOCALTIME、LOCALTIMESTAMP。（参见Section 9.9.4。注意这些是SQL函数并且在数据输入串中不被识别。

### Caution

虽然输入字符串 `now`、`today`、`tomorrow`和`yesterday`都可以在交互式SQL命令中使用，但当命令被保存以待稍后执行时，它们可能会有令人惊讶的行为，例如在准备的语句、视图和函数定义中。字符串可以被转换为特定的时间值，该值在变得过时后长时间继续使用。在这种上下文中使用某一种SQL函数代替。例如，`CURRENT_DATE + 1`比`'tomorrow'::date`更安全。

## 8.5.2. 日期/时间输出

时间/日期类型的输出格式可以设成四种风格之一：`ISO 8601`、SQL (Ingres)、传统的POSTGRES (Unix的date格式)或`German`。缺省是ISO格式 (ISO标准要求使用`ISO 8601`格式。ISO输出格式的名字是历史偶然)。Table 8.1显示了每种输出风格的例子。`date`和`time`类型的输出通常只有日期或时间部分和例子中一致。不过，POSTGRES风格输出的是ISO格式的只有日期的值。

Table 8.14. 日期/时间输出风格

风格声明	描述	例子
ISO	ISO 8601, SQL标准	1997-12-17 07:37:16-08
SQL	传统风格	12/17/1997 07:37:16.00 PST
Postgres	原始风格	Wed Dec 17 07:37:16 1997 PST
German	地区风格	17.12.1997 07:37:16.00 PST

### Note

ISO 8601指定使用大写字母T来分隔日期和时间。LightDB在输入上接受这种格式，但是在输出时它采用一个空格而不是T，如上所示。和一些其他数据库系统一样，这是为了可读性以及与RFC 3339<sup>1</sup>的一致性。

SQL和POSTGRES风格中，如果DMY域顺序被指定，“日”将出现在“月”之前，否则“月”出现在“日”之前 (有关该设置如何影响输入值的解释，请参考Section 8.5.1)。Table 8.1给出了例子。

Table 8.15. 日期顺序习惯

datestyle设置	输入顺序	例子输出
SQL, DMY	日/月/年	17/12/1997 15:37:16.00 CET
SQL, MDY	月/日/年	12/17/1997 07:37:16.00 PST
Postgres, DMY	日/月/年	Wed 17 Dec 07:37:16 1997 PST

<sup>1</sup> <https://tools.ietf.org/html/rfc3339>

在ISO样式中，时区始终显示为与UTC之间的有符号数偏移，正号用于格林威治以东的区域。如果偏移量为整小时数，则偏移量将显示为 hh（仅小时），如果偏移量是整分钟数，则为hh:mm，否则为hh:mm:ss。（第三种情况在任何现代时区标准中都不可能实现，但如果是时区标准启用之前的时间戳则可能会出现。）在其它日期样式中，如果时区在当前时区中常用，则时区显示为字母缩写。否则，它将显示为ISO 8601基本格式的带符号数字偏移量（hh或hhmm）。

日期/时间风格可以由用户使用SET datestyle命令选取，在lightdb.conf配置文件里的参数DateStyle设置或者在服务器或客户端的LTDATESTYLE环境变量里设置。

格式化函数to\_char（见Section 9.8）也可以作为一个更灵活的方式来格式化日期/时间输出。

### 8.5.3. 时区

时区和时区习惯不仅仅受地球几何形状的影响，还受到政治决定的影响。到了19世纪，全球的时区变得稍微标准化了些，但是还是易于遭受随意的修改，部分是因为夏时制规则。LightDB使用广泛使用的IANA (Olson) 时区数据库来得到有关历史时区规则的信息。对于未来的时间，我们假设关于一个给定时区的最新已知规则将会一直持续到无穷远的未来。

LightDB努力在典型使用中与SQL标准的定义相兼容。但SQL标准在日期和时间类型和功能上有一些奇怪的混淆。两个显而易见的问题是：

- 尽管date类型与时区没有联系，而time类型却可以有。然而，现实世界的时区只有在与时间和日期都关联时才有意义，因为偏移（时差）可能因为实行类似夏时制这样的制度而在一年里有所变化。
- 缺省的时区会指定一个到UTC的数字常量偏移（时差）。因此，当跨DST边界做日期/时间算术时，我们根本不可能适应于夏时制时间。

为了克服这些困难，我们建议在使用时区的时候，使用那些同时包含日期和时间的日期/时间类型。我们不建议使用类型time with time zone（尽管LightDB出于遗留应用以及与SQL标准兼容性的考虑支持这个类型）。LightDB假设你用于任何类型的本地时区都只包含日期或时间。

在系统内部，所有时区相关的日期和时间都用UTC存储。它们在被显示给客户端之前会被转换成由TimeZone配置参数指定的本地时间。

LightDB允许你使用三种不同形式指定时区：

- 一个完整的时区名字，例如America/New\_York。能被识别的时区名字被列在pg\_timezone\_names视图中（参见Section 48.100）。LightDB用广泛使用的IANA时区数据来实现该目的，因此相同的时区名字也可以在其他软件中被识别。
- 一个时区缩写，例如PST。这样一种声明仅仅定义了到UTC的一个特定偏移，而不像完整时区名那样指出整套夏令时转换规则。能被识别的缩写被列在pg\_timezone\_abbrevs视图中（参见Section 48.99）。你不能将配置参数TimeZone或log\_timezone设置成一个时区缩写，但是你可以在日期/时间输入值和AT TIME\_ZONE操作符中使用时区缩写。
- 除了时区名和缩写，LightDB将接受POSIX-风格的时区规范，如同Section D.8所描述的。这个选项通常不优先用于指定时区，但是，如果没有合适的IANA时区条目，这可能是必要的。

简而言之，在缩写和全称之间是有不同的：缩写表示从UTC开始的一个特定偏移量，而很多全称表示一个本地夏令时规则并且因此具有两种可能的UTC偏移量。例如，2014-06-04 12:00 America/New\_York表示纽约本地时间的中午，这个特殊的日期是东部夏令时间（UTC-4）。因此2014-06-04 12:00 EDT 指定的是同一个时间点。但是2014-06-04 12:00 EST指定东部标准时间的中午（UTC-5），不管在那个日期夏令时是否生效。

更要命的是，某些行政区已经使用相同的时区缩写在不同的时间表示不同的 UTC 偏移量。例如，在莫斯科MSK在某些年份表示 UTC+3 而在另一些年份表示 UTC+4。LightDB 会根据在指定的日期它们到底表示什么（或者最近表示什么）来解释这种缩写。但是，正如上面的EST例子所示，这并不是必须和那一天的本地 标准时间相同。

在所有情况下，时区名及其缩写都是大小写不敏感的（这是对LightDB 8.2之前版本的一个修改，在这些版本中某些环境下时区名是大小写敏感的而在另外一些环境中却是大小写不敏感的）。

时区名和缩写都不是硬写在服务器中的，它们是从存储在安装目录下的.../share/timezone/和.../share/timezonesets/子目录中获取的（参见Section D.4）。

TimeZone配置参数可以在文件lightdb.conf中被设置，或者使用Chapter 1中描述的任何一种标准方法设置。同时也有一些特殊的方法来设置它：

- SQL命令SET TIME\_ZONE为会话设置时区。它是SET TIMEZONE TO的另一种拼写，它更加符合SQL的语法。
- libpq客户端使用PGTZ环境变量来通过连接发送一个SET TIME\_ZONE命令给服务器。

## 8.5.4. 间隔输入

interval值可以使用下列语法书写：

```
[@] quantity unit [quantity unit...] [direction]
```

其中quantity是一个数字（很可能是有符号的）； unit是毫秒、 millisecond、second、minute、hour、day、week、month、year、decade、century、millennium 或者缩写或者这些单位的复数； direction可以是ago或者为空。At符号(@)是一个可选的噪声。不同单位的数量通过合适的符号计数被隐式地添加。ago对所有域求反。如果IntervalStyle被设置为postgres\_verbose，该语法也被用于间隔输出。

日、小时、分钟和秒的数量可以不适用显式的单位标记指定。例如，'1 12:59:10' 被读作'1 day 12 hours 59 min 10 sec'。同样，一个年和月的组合可以使用一个横线指定，例如'200-10' 被读作'200年10个月'（这些较短的形式事实上是SQL标准唯一许可的形式，并且在IntervalStyle被设置为sql\_standard时用于输出）。

间隔值也可以被写成 ISO 8601 时间间隔，使用该标准4.4.3.2小节的“带标志符的格式”或者4.4.3.3小节的“替代格式”。带标志符的格式看起来像这样：

```
P quantity unit [ quantity unit ...] [ T [ quantity unit ...]]
```

该串必须以一个P开始，并且可以包括一个引入当日时间单位的T。可用的单位缩写在Table 8.16中给出。单位可以被忽略，并且可以以任何顺序指定，但是小于一天的单位必须出现在T之后。特别地，M的含义取决于它出现在T之前还是之后。

Table 8.16. ISO 8601 间隔单位缩写

缩写	含义
Y	年
M	月（在日期部分中）
W	周
D	日
H	小时
M	分钟（在时间部分中）
S	秒



如果使用替代格式:

P [ years-months-days ] [ T hours:minutes:seconds ]

串必须以P开始，并且一个T分隔间隔的日期和时间部分。其值按照类似于 ISO 8601日期的数字给出。

在用一個域声明书写一个间隔常量时，或者为一个用域声明定义的间隔列赋予一个串时，对于为标记的量的解释依赖于域。例如INTERVAL '1' YEAR被解读成1年，而INTERVAL '1'表示1秒。同样，域声明允许的最后一个有效域“右边”的域值会被无声地丢弃掉。例如书写INTERVAL '1 day 2:03:04' HOUR TO MINUTE将会导致丢弃秒域，而不是日域。

根据SQL标准，一个间隔值的所有域都必须由相同的符号，这样一个领头的负号将会应用到所有域；例如在间隔文字'-1 2:03:04'中的负号会被应用于日、小时、分钟和秒部分。LightDB允许域具有不同的符号，并且在习惯上认为以文本表示的每个域具有独立的符号，因此在这个例子中小时、分钟和秒部分被认为是正值。如果IntervalStyle被设置为sql\_standard，则一个领头的符号将被认为是应用于所有域（但是仅当没有额外符号出现）。否则将使用传统的LightDB解释。为了避免混淆，我们推荐在任何域为负值时为每一个域都附加一个显式的符号。

字段值可以有小数部分：例如，'1.5周'或'01:02:03.45'。但是，由于interval内部只存储三个整数单位（月，日，微秒），因此分数单位必须溢出到较小的单位。大于月的单位的小数部分被截断为整数个月，例如，'1.5年'变为'1年6个月'。周和日的小数部分计算为天数和微秒的整数，假设每月30天，每天24小时，例如，'1.75月'变为1月 22天12:00:00。只有秒才会在输出上显示为小数。

Table 8.1展示了一些有效interval输入的例子。

Table 8.17. 间隔输入

例子	描述
1-2	SQL标准格式：1年2个月
3 4:05:06	SQL标准格式：3日4小时5分钟6秒
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	传统Postgres格式：1年2个月3日4小时5分钟6秒钟
P1Y2M3DT4H5M6S	“带标志符的” ISO 8601 格式：含义同上
P0001-02-03T04:05:06	ISO 8601 的“替代格式”：含义同上

在内部，interval值被存储为months、days以及microseconds。之所以这样做是因为一个月中的天数是变化的，并且在涉及到夏令时调整时一天可以有23或者25个小时。months以及days域是整数，而microseconds字段可以存储带小数的秒。因为区间通常是从常量字符串或者timestamp减法创建而来，这种存储方法在大部分情况下都很好，但是也可能导致预料之外的结果：

```
SELECT EXTRACT(hours from '80 minutes'::interval);
date_part
-----
1
```

```
SELECT EXTRACT(days from '80 hours'::interval);
date_part
-----
0
```

函数justify\_days和justify\_hours可以用来调整溢出其正常范围之外的days和hours。

## 8.5.5. 间隔输出

间隔类型的输出格式可以被设置为四种风格之

一: `sql_standard`、`postgres`、`postgres_verbose`或`iso_8601`，设置方法使用`SET intervalstyle`命令。默认值为`postgres`格式。Table 8.1展示了每种输出风格的例子。

如果间隔值符合SQL标准的限制（仅年-月或仅日-时间，没有正负值部分的混合），`sql_standard`风格为间隔文字串产生符合SQL标准规范的输出。否则输出将看起来像一个标准的年-月文字串跟着一个日-时间文字串，并且带有显式添加的符号以区分混合符号的间隔。

当`DateStyle`参数被设置为ISO时，`postgres`风格的输出匹配LightDB 8.4版本以前的输出。

当`DateStyle`参数被设置为非ISO输出时，`postgres_verbose`风格的输出匹配LightDB 8.4版本以前的输出。

`iso_8601`风格的输出匹配在ISO 8601标准的4.4.3.2小节中描述的“带标志符的格式”。

Table 8.18. 间隔输出风格例子

风格声明	年-月间隔	日-时间间隔	混合间隔
<code>sql_standard</code>	1-2	3 4:05:06	-1-2 +3 -4:05:06
<code>postgres</code>	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
<code>postgres_verbose</code>	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
<code>iso_8601</code>	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

## 8.6. 布尔类型

LightDB提供标准的SQL类型`boolean`，参见Table 8.19 `boolean`可以有多个状态：“true（真）”、“false（假）”和第三种状态“unknown（未知）”，未知状态由SQL空值表示。

Table 8.19. 布尔数据类型

名字	存储字节	描述
<code>boolean</code>	1字节	状态为真或假

在SQL查询中，布尔常量可以表示为SQL关键字`TRUE`，`FALSE`，和`NULL`。

`boolean` 类型的数据类型输入函数接受这些字符串表示“真”状态：

```
true
yes
on
1
```

下面这些表示“假”状态：

```
false
no
off
0
```

这些字符串的唯一前缀也可以接受，例如 `t` 或 `n`。前端或尾部的空格将被忽略，并且大小写不敏感。

`boolean` 类型的数据类型输出函数总是发出 `t` 或 `f`，如 [Example 8.2](#) 所示。

### Example 8.2. 使用 `boolean` 类型

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
```

a	b
t	sic est
f	non est

```
SELECT * FROM test1 WHERE a;
```

a	b
t	sic est

在 SQL 查询中优先使用关键字 `TRUE` 和 `FALSE` 来写布尔常数 (SQL-兼容)。但是你也可以使用遵循 [Section 4.1.2.8](#) 中描述的通用字符串文字常量句法的字符串来表达，例如 `'yes'::boolean`。

注意语法分析程序会把 `TRUE` 和 `FALSE` 自动理解为 `boolean` 类型，但是不包括 `NULL`，因为它可以是任何类型的。因此在某些语境中你也许要将 `NULL` 转化为显示 `boolean` 类型，例如 `NULL::boolean`。反过来，上下文中的字符串文字布尔值也可以不转换，当语法分析程序能够断定文字必定为 `boolean` 类型时。

## 8.7. 枚举类型

枚举 (`enum`) 类型是由一个静态、值的有序集合构成的数据类型。它们等效于很多编程语言所支持的 `enum` 类型。枚举类型的一个例子可以是一周中的日期，或者一个数据的状态值集合。

### 8.7.1. 枚举类型的声明

枚举类型可以使用 `CREATE TYPE` 命令创建，例如：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

一旦被创建，枚举类型可以像很多其他类型一样在表和函数定义中使用：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
```

name	current_mood
Moe	happy

(1 row)

## 8.7.2. 排序

一个枚举类型的值的排序是该类型被创建时所列出的值的顺序。枚举类型的所有标准的比较操作符以及相关聚集函数都被支持。例如：

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
name | current_mood
-----+-----
Moe  | happy
Curly | ok
(2 rows)
```

```
SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
name | current_mood
-----+-----
Curly | ok
Moe  | happy
(2 rows)
```

```
SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
name
-----
Larry
(1 row)
```

## 8.7.3. 类型安全性

每一种枚举数据类型都是独立的并且不能和其他枚举类型相比较。看这样一个例子：

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

如果你确实需要做这样的事情，你可以写一个自定义的操作符或者在查询中加上显式造型：

```
SELECT person.name, holidays.num_weeks FROM person, holidays
WHERE person.current_mood::text = holidays.happiness::text;
name | num_weeks
-----+-----
Moe  |          4
(1 row)
```

## 8.7.4. 实现细节

枚举标签是大小写敏感的，因此 'happy' 与 'HAPPY' 是不同的。标签中的空格也是有意义的。

尽管枚举类型的主要目的是用于值的静态集合，但也有方法在现有枚举类型中增加新值和重命名值（见 ALTER TYPE）。不能从枚举类型中去除现有的值，也不能更改这些值的排序顺序，如果要那样做可以删除并且重建枚举类型。

一个枚举值在磁盘上占据4个字节。一个枚举值的文本标签的长度受限于 NAMEDATALEN 设置，该设置被编译在 LightDB 中，在标准编译下它表示最多63字节。

从内部枚举值到文本标签的翻译被保存在系统目录 pg\_enum 中。可以直接查询该目录。

## 8.8. 几何类型

几何数据类型表示二维的空间物体。Table 8.2展示了 LightDB 中可以用的几何类型。

Table 8.20. 几何类型

名字	存储尺寸	表示	描述
point	16字节	平面上的点	(x, y)
line	32字节	无限长的线	{A, B, C}
lseg	32字节	有限线段	((x1, y1), (x2, y2))
box	32字节	矩形框	((x1, y1), (x2, y2))
path	16+16n字节	封闭路径（类似于多边形）	((x1, y1), ...)
path	16+16n字节	开放路径	[(x1, y1), ...]
polygon	40+16n字节	多边形（类似于封闭路径）	((x1, y1), ...)
circle	24字节	圆	<(x, y), r>（中心点和半径）

我们有一系列丰富的函数和操作符可用来进行各种几何操作，如缩放、平移、旋转和计算相交等 它们在 Section 9.1 中解释。

### 8.8.1. 点

点是几何类型的基本二维构造块。用下面的语法描述 point 类型的值：

```
( x , y )
 x , y
```

其中 x 和 y 分别是坐标，都是浮点数。

点使用第一种语法输出。

### 8.8.2. 线

线由线性方程  $Ax + By + C = 0$  表示，其中 A 和 B 都不为零。类型 line 的值采用以下形式输入和输出：

{ A, B, C }

另外，还可以用下列任一形式输入：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1 , x2 , y2
```

其中 (x1, y1) 和 (x2, y2) 是线上不同的两点。

### 8.8.3. 线段

线段用一对线段的端点来表示。lseg类型的值用下面的语法声明：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1 , x2 , y2
```

其中(x1, y1) 和 (x2, y2) 是线段的端点。

线段使用第一种语法输出。

### 8.8.4. 方框

方框用其对角的点表示。box类型的值使用下面的语法指定：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1 , x2 , y2
```

其中(x1, y1) 和 (x2, y2) 是方框的对角点。

方框使用第二种语法输出。

在输入时可以提供任意两个对角，但是值将根据需要被按顺序记录为右上角和左下角。

### 8.8.5. 路径

路径由一系列连接的点组成。路径可能是开放的，也就是认为列表中第一个点和最后一个点没有被连接起来；也可能是封闭的，这时认为第一个和最后一个点被连接起来。

path类型的值用下面的语法声明：

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1 , ... , xn , yn )
x1 , y1 , ... , xn , yn
```

其中的点是组成路径的线段的端点。方括弧 ([]) 表示一个开放的路径，圆括弧 (()) 表示一个封闭的路径。如第三种到第五种语法所示，当最外面的圆括号被忽略时，路径将被假定为封闭。

路径的输出使用第一种或第二种语法。

## 8.8.6. 多边形

多边形由一系列点代表（多边形的顶点）。多边形和封闭路径很像，但是存储方式不一样而且有自己的一套支持例程。

polygon类型的值用下列语法声明：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1 , ... , xn , yn )
x1 , y1 , ... , xn , yn
```

其中的点是组成多边形边界的线段的端点。

多边形的输出使用第一种语法。

## 8.8.7. 圆

圆由一个圆心和半径代表。circle类型的值用下面的语法指定：

```
< ( x , y ) , r >
( ( x , y ) , r )
( x , y ) , r
x , y , r
```

其中(x, y)是圆心，而r是圆的半径。

圆的输出用第一种语法。

## 8.9. 网络地址类型

LightDB提供用于存储 IPv4、IPv6 和 MAC 地址的数据类型，如Table 8.2所示。用这些数据类型存储网络地址比用纯文本类型好，因为这些类型提供输入错误检查以及特殊的操作符和函数（见Section 9.12）

Table 8.21. 网络地址类型

名字	存储尺寸	描述
cidr	7或19字节	IPv4和IPv6网络
inet	7或19字节	IPv4和IPv6主机以及网络
macaddr	6字节	MAC地址
macaddr8	8 bytes	MAC地址（EUI-64格式）

在对inet或者cidr数据类型进行排序的时候，IPv4 地址将总是排在 IPv6 地址前面，包括那些封装或者是映射在 IPv6 地址里的 IPv4 地址，例如 ::10.2.3.4 或者 ::ffff::10.4.3.2。

### 8.9.1. inet

inet在一个数据域里保存一个 IPv4 或 IPv6 主机地址，以及一个可选的它的子网。子网由主机地址中表示的网络地址位数表示（“网络掩码”）。如果网络掩码为 32 并且地址是 IPv4 ，那么该值不表示任何子网，只是一台主机。在 IPv6 中地址长度是 128 位，因此 128 位指定一个唯一的主机地址。请注意如果你想只接受网络地址，你应该使用cidr类型而不是inet。

该类型的输入格式是地址/*y*，其中地址是一个 IPv4 或者 IPv6 地址，*y*是网络掩码的位数。如果/*y*部分省略，则网络掩码对 IPv4 取为 32，对 IPv6 取为 128，所以该值表示只有一台主机。在显示时，如果/*y*部分指定一个单台主机，它将被显示出来。

## 8.9.2. cidr

cidr类型保存一个 IPv4 或 IPv6 网络地址声明。其输入和输出遵循无类的互联网路由 (Classless Internet Domain Routing) 习惯。声明一个网络的格式是地址/*y*，其中address是最低位地址，表现为 IPv4 或 IPv6 网络地址而*y*是网络掩码的位数。如果省略*y*，那么掩码部分用旧的有类的网络编号系统进行计算，否则它将至少大到足以包括写在输入中的所有字节。声明一个在其指定的掩码右边置了位的网络地址会导致错误。

Table 8.2展示了一些例子。

Table 8.22. cidr类型输入例子

cidr输入	cidr输出	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba/64
2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

## 8.9.3. inet vs. cidr

inet和cidr类型之间的本质区别是inet接受右边有非零位的网络掩码，而cidr不接受。例如，192.168.0.1/24对inet是有效的，但对cidr是无效的。

### Tip

如果你不喜欢inet或cidr值的输出格式，可以尝试函数host、text和abbrev。

## 8.9.4. macaddr

macaddr类型存储 MAC 地址，也就是以太网卡硬件地址（尽管 MAC 地址还用于其它用途）。可以接受下列格式的输入：

```
'08:00:2b:01:02:03'
```



```
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

这些例子指定的都是同一个地址。对于位a到f，大小写都可以接受。输出总是使用展示的第一种形式。

IEEE Std 802-2001 指定第二种展示的形式（带有连字符）作为MAC地址的标准形式，并且指定第一种形式（带有分号）作为位翻转的记号，因此 08-00-2b-01-02-03 = 01:00:4D:08:04:0C。这种习惯目前已经被广泛地忽略，并且它只与废弃的网络协议（如令牌环）相关。LightDB 没有对位翻转做任何规定，并且所有可接受的格式都使用标准的LSB顺序。

剩下的五种输入格式不属于任何标准。

## 8.9.5. macaddr8

macaddr8类型以EUI-64格式存储MAC地址，例如以太网卡的硬件地址（尽管MAC地址也被用于其他目的）。这种类型可以接受6字节和8字节长度的MAC地址，并且将它们存储为8字节长度的格式。以6字节格式给出的MAC地址被存储为8字节长度格式的方式是把第4和第5字节分别设置为FF和FE。注意IPv6使用一种修改过的EUI-64格式，其中从EUI-48转换过来后的第7位应该被设置为一。函数macaddr8\_set7bit被用来做这种修改。一般而言，任何由16进制数（字节边界上）对构成的输入（可以由':'、'-'或者'.'统一地分隔）都会被接受。16进制数的数量必须是16（8字节）或者12（6字节）。前导和拖尾的空格会被忽略。下面是可以被接受的输入格式的例子：

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'
```

这些例子都指定相同的地址。数字a到f的大小写形式都被接受。输出总是以上面显示的第一种形式。

上述的后六种输入格式不属于任何标准。

要把EUI-48格式的传统48位MAC地址转换成修改版EUI-64格式（包括在IPv6地址中作为主机部分），可以使用下面的macaddr8\_set7bit：

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');
```

```
macaddr8_set7bit
-----
0a:00:2b:ff:fe:01:02:03
(1 row)
```

## 8.10. 位串类型

位串就是一串 1 和 0 的串。它们可以用于存储和可视化位掩码。我们有两种类型的 SQL 位类型：bit(n)和bit varying(n)，其中 n是一个正整数。

bit类型的数据必须准确匹配长度n； 试图存储短些或者长一些的位串都是错误的。bit varying数据是最长n的变长类型，更长的串会被拒绝。写一个没有长度的bit等效于bit(1)，没有长度的bit varying意味着没有长度限制。

### Note

如果我们显式地把一个位串值转换成bit(n)，那么它的右边将被截断或者在右边补齐零，直到刚好n位，而且不会抛出任何错误。类似地，如果我们显式地把一个位串数值转换成bit varying(n)，如果它超过了n位，那么它的右边将被截断。

请参考Section 4.1.2.6获取有关位串常量的语法的信息。还有一些位逻辑操作符和串操作函数可用，请见Section 9.6

### Example 8.3. 使用位串类型

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
```

```
ERROR: bit string length 2 does not match type bit(3)
```

```
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
```

a	b
101	00
100	101

一个位串值对于每8位的组需要一个字节，外加总共5个或8个字节，这取决于串的长度（但是长值可能被压缩或者移到线外，如Section 8.8中对字符串的解释一样）。

## 8.11. 文本搜索类型

LightDB提供两种数据类型，它们被设计用来支持全文搜索，全文搜索是一种在自然语言的文档集合中搜索以定位那些最匹配一个查询的文档的活动。tsvector类型表示一个为文本搜索优化的形式下的文档，tsquery类型表示一个文本查询。Chapter 11提供了对于这种功能的详细解释，并且Section 9.1总结了相关的函数和操作符。

### 8.11.1. tsvector

一个tsvector值是一个排序的可区分词位的列表，词位是被正规化合并了同一个词的不同变种的词（详见Chapter 12）。排序和去重是在输入期间自动完成的，如下例所示：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
          tsvector
-----
'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

要表示包含空白或标点的词位，将它们用引号包围：

```
SELECT $$the lexeme ' ' contains spaces$$::tsvector;
          tsvector
```

```
-----
' ' 'contains' 'lexeme' 'spaces' 'the'
```

（我们在这个例子中使用美元符号包围的串文字并且下一个用来避免在文字中包含双引号记号产生的混淆）。嵌入的引号和反斜线必须被双写：

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
          tsvector
```

```
-----
'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

可选的，整数位置可以被附加给词位：

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11
       rat:12'::tsvector;
          tsvector
```

```
-----
'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

一个位置通常表示源词在文档中的定位。位置信息可以被用于邻近排名。位置值可以从 1 到 16383，更大的数字会被 16383。对于相同的词位出现的重复位置将被丢弃。

具有位置的词位可以进一步地被标注一个权重，它可以是A、B、C或D。D是默认值并且因此在输出中不会显示：

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
```

```
-----
'a':1A 'cat':5 'fat':2B,4C
```

权重通常被用来反映文档结构，例如将主题词标记成与正文词不同。文本搜索排名函数可以为不同的权重标记器分配不同的优先级。

了解tsvector类型本身并不执行任何词正规化这一点很重要，它假定给它的词已经被恰当地为应用正规化过。例如，

```
SELECT 'The Fat Rats'::tsvector;
          tsvector
```

```
-----
'Fat' 'Rats' 'The'
```

对于大部分英语文本搜索应用，上面的词将会被认为是非正规化的，但是tsvector并不在乎这一点。原始文档文本通常应该经过to\_tsvector以恰当地为搜索正规化其中的词：

```
SELECT to_tsvector('english', 'The Fat Rats');
          to_tsvector
```

```
-----
'fat':2 'rat':3
```

再次地，详情请参阅Chapter 12

## 8.11.2. tsquery

一个tsquery值存储要用于搜索的词位，并且使用布尔操作符& (AND)、| (OR) 和! (NOT) 来组合它们，还有短语搜索操作符<-> (FOLLOWED BY)。也有一种 FOLLOWED BY 操作符的变体<N>，其中N是一个整数常量，它指定要搜索的两个词位之间的距离。<->等效于<1>。

圆括号可以被用来强制对操作符分组。如果没有圆括号，! (NOT) 的优先级最高，其次是<-> (FOLLOWED BY)，然后是& (AND)，最后是| (OR)。

这里有一些例子：

```
SELECT 'fat & rat'::tsquery;
      tsquery
-----
'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
      tsquery
-----
'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
      tsquery
-----
'fat' & 'rat' & !'cat'
```

可选地，一个tsquery中的词位可以被标注一个或多个权重字母，这将限制它们只能和具有那些权重之一的tsvector词位相匹配：

```
SELECT 'fat:ab & cat'::tsquery;
      tsquery
-----
'fat':AB & 'cat'
```

此外，一个tsquery中的词位可以被标注为\*来指定前缀匹配：

```
SELECT 'super:*'::tsquery;
      tsquery
-----
'super':*
```

这个查询将匹配一个tsvector中以“super”开头的任意词。

词位的引号规则和之前描述的tsvector中的词位相同；并且，正如tsvector，任何请求的词正规化必须在转换到tsquery类型之前完成。to\_tsquery函数可以方便地执行这种正规化：

```
SELECT to_tsquery('Fat:ab & Cats');
      to_tsquery
-----
'fat':AB & 'cat'
```

注意to\_tsquery将会以和其他词同样的方式处理前缀，这也意味着下面的比较会返回真：

```
SELECT to_tsvector('postgraduate') @@ to_tsquery('postgres:*');
?column?
-----
t
```

因为postgres会被处理成postgr:

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'postgres:*' );
 to_tsvector | to_tsquery
-----+-----
 'postgradu':1 | 'postgr':*
```

这会匹配postgraduate被处理后的形式。

## 8.12. UUID类型

数据类型uuid存储由RFC 4122<sup>2</sup>, ISO/IEC 9834-8:2005以及相关标准定义的通用唯一标识符 (UUID) (某些系统将这种数据类型引用为全局唯一标识符GUID)。这种标识符是一个128位的量, 它由一个精心选择的算法产生, 该算法能保证在已知空间中任何其他使用相同算法的人能够产生同一个标识符的可能性非常非常小。因此, 对于分布式系统, 这些标识符相比序列生成器而言提供了一种很好的唯一性保障, 序列生成器只能在一个数据库中保证唯一。

一个UUID被写成一个小写十六进制位的序列, 该序列被连字符分隔成多个组: 首先是一个8位组, 接下来是三个4位组, 最后是一个12位组。总共的32位 (十六进制位) 表示了128个二进制位。一个标准形式的UUID类似于:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

LightDB也接受另一种输入形式: 使用大写位、标准格式被花括号包围、忽略某些或者全部连字符、在任意4位组后面增加一个连字符。例如:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

输出总是采用标准形式。

对于如何在LightDB中生成UUID, 请参见 Section 9.14。

## 8.13. XML类型

xml数据类型可以被用来存储XML数据。它比直接在一个text域中存储XML数据的优势在于, 它会检查输入值的结构是不是良好, 并且有支持函数用于在其上执行类型安全的操作, 参见Section 9.15 使用这种数据类型要求在安装时用configure --with-libxml选项编译。

xml类型可以存储格式良好的遵循XML标准定义的“文档”, 以及“内容”片段, 它是通过引用更宽泛的“document node”<sup>3</sup> XQuery 和 XPath 数据模型来定义的。大致上说, 这意味着内容片段中可以有多于一个的顶层元素或字符节点。表达式xmlvalue IS DOCUMENT可以用来评估一个特定的xml值是一个完整文档或者仅仅是一个文档片段。

xml 数据类型的限制和兼容性说明可以在 Section F.9中找到。

### 8.13.1. 创建XML值

要从字符数据中生成一个xml类型的值, 可以使用函数xmlparse:

<sup>2</sup> <https://tools.ietf.org/html/rfc4122>

<sup>3</sup> <https://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/#DocumentNode>

```
XMLPARSE ( { DOCUMENT | CONTENT } value [WELLFORMED])
```

例子:

```
XMLPARSE (DOCUMENT ' <?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>' )
XMLPARSE (CONTENT ' abc<foo>bar</foo><bar>foo</bar>' )
XMLPARSE (DOCUMENT ' <?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>' WELLFORMED)
XMLPARSE (CONTENT ' abc<foo>bar</foo><bar>foo</bar>' WELLFORMED)
```

然而根据SQL标准这是唯一将字符串转换为XML值的方法，LightDB特有的语法:

```
xml ' <foo>bar</foo>'
' <foo>bar</foo>' ::xml
```

也可以被使用。

即便输入值指定了一个文档类型声明 (DTD)，xml类型也不根据DTD来验证输入值。目前也没有内建的支持用于根据其他XML模式语言 (如XML模式) 来进行验证。

作为一个逆操作，从xml产生一个字符串可以使用函数xmlserialize:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

type可以是 character、character varying或 text (或者其中之一的一个别名)。再次地，根据SQL标准，这也是在xml类型和字符类型间做转换的唯一方法，但是LightDB也允许你简单地造型这些值。

当一个字符串不是使用XMLPARSE造型成xml或者不是使用XMLSERIALIZE从xml造型得到，对于DOCUMENT和CONTENT两者的选择是根据“XML option” 会话配置参数决定的，它可以使用标准命令来设置:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

或者是更具有LightDB风格的语法

```
SET xmloption TO { DOCUMENT | CONTENT };
```

默认值是CONTENT，因此所有形式的XML数据都被允许。

兼容性: WELLFORMED 指令仅在 Oracle 模式时才有效。

## 8.13.2. 编码处理

在客户端、服务器以及其中流过的XML数据上处理多字符编码时必须要注意。在使用文本模式向服务器传递查询以及向客户端传递查询结果 (在普通模式) 时，LightDB将所有在客户端和服务器之间传递的字符数据转换为目标端的字符编码，参见Section 21.3 这也包括了表示XML值的串，正如上面的例子所述。这也通常意味着由于字符数据会在客户端和服务器之间传递时被转换成其他编码，包含在XML数据中的编码声明可能是无效的，因为内嵌的编码声明没有被改变。为了处理这种行为，包含在表示xml类型输入的字符串中包含的编码声明会被忽略，并且其内容被假定为当前服务器的编码。接着，为了正确处理，XML数据的字符串必须以当前客户端编码从客户端发出。客户端负责在把文档发送给服务器之前将它们转换为当前客户端编码，或者适当地调整客户端编码。在输出时，xml类型的值将不会有一个编码声明，并且客户端将会假设所有数据都是当前客户端编码。

在使用二进制模式传送查询参数给服务器以及传回查询结果给客户端时，不会执行编码转换，因此情况就有所不同。在这种情况下，XML数据中的编码声明将被注意到，并且如果缺少编码声明时该数据会被假定为UTF-8（由于XML标准的要求，注意LightDB不支持UTF-16）。在输出时，数据将会有有一个编码声明来指定客户端编码，除非客户端编码为UTF-8（这种情况下编码声明会被忽略）。

不用说，在LightDB中处理XML数据产生错误的可能性更小，并且在XML数据编码、客户端编码和服务器编码三者相同时效率更高。因为XML数据在内部是以UTF-8处理的，如果服务器编码也是UTF-8时，计算效率将会最高。

### Caution

当服务器编码不是UTF-8时，某些XML相关的函数可能在非ASCII数据上完全无法工作。尤其在`xmltable()`和`xpath()`上，这是一个已知的问题。

## 8.13.3. 访问XML值

xml数据类型有些不同寻常，因为它不提供任何比较操作符。这是因为对于XML数据不存在良定义的和通用的比较算法。这种状况造成的后果就是，你无法通过比较一个xml和一个搜索值来检索行。XML值因此通常应该伴随着一个独立键值域，如一个ID。另一种比较XML值的方案是将它们先转换为字符串，但注意字符串比较对于XML比较方法没有什么帮助。

由于没有可以用于xml数据类型的比较操作符，因此无法直接在这种类型上创建索引。如果需要在XML中快速的搜索，可能的解决方案包括将表达式造型为一个字符串类型然后索引之，或者在一个XPath表达式上索引。当然，实际的查询必须被调整为使用被索引的表达式。

LightDB中的文本搜索功能也可以被用来加速XML数据的全文搜索。但是，所需的预处理支持目前在LightDB发布中还不可用。

## 8.14. JSON 类型

根据RFC 7159<sup>4</sup>中的说明，JSON数据类型是用来存储JSON（JavaScript Object Notation）数据的。这种数据也可以被存储为text，但是JSON数据类型的优势在于能强制要求每个被存储的值符合JSON规则。也有很多JSON相关的函数和操作符可以用于存储在这些数据类型中的数据，见Section 9.16

LightDB提供存储JSON数据的两种类型：`json`和`jsonb`。为了实现这些数据类型高效的查询机制，LightDB还在Section 8.14.5中提供了`jsonpath`数据类型描述。

`json`和`jsonb`数据类型接受几乎完全相同的值集合作为输入。主要的实际区别之一是效率。`json`数据类型存储输入文本的精准拷贝，处理函数必须在每次执行时必须重新解析该数据。而`jsonb`数据被存储在一种分解好的二进制格式中，它在输入时要稍慢一些，因为需要做附加的转换。但是`jsonb`在处理时要快很多，因为不需要解析。`jsonb`也支持索引，这也是一个令人瞩目的优势。

由于`json`类型存储的是输入文本的准确拷贝，其中可能会保留在语法上不明显的、存在于记号之间的空格，还有JSON对象内部的键的顺序。还有，如果一个值中的JSON对象包含同一个键超过一次，所有的键/值对都会被保留（处理函数会把最后的值当作有效值）。相反，`jsonb`不保留空格、不保留对象键的顺序并且不保留重复的对象键。如果在输入中指定了重复的键，只有最后一个值会被保留。

通常，除非有特别特殊的需要（例如遗留的对象键顺序假设），大多数应用应该更愿意把JSON数据存储为`jsonb`。

<sup>4</sup> <https://tools.ietf.org/html/rfc7159>

RFC 7159 指定 JSON 字符串应以 UTF8 编码。因此 JSON 类型不可能严格遵守 JSON 规范，除非数据库编码是 UTF8。尝试直接包括数据库编码中无法表示的字符将会失败。反过来，能在数据库编码中表示但是不在 UTF8 中的字符是被允许的。

RFC 7159 允许 JSON 字符串包含 \uXXXX 所标记的 Unicode 转义序列。在 json 类型的输入函数中，不管数据库编码如何都允许 Unicode 转义，并且只检查语法正确性（即，跟在 \u 后面的四个十六进制位）。但是，jsonb 的输入函数更加严格：它不允许对无法在数据库编码中表示的字符进行 Unicode 转义。jsonb 类型也拒绝 \u0000（因为 LightDB 的 text 类型无法表示它），并且它坚持使用 Unicode 代理对来标记位于 Unicode 基本多语言平面之外的字符是正确的。合法的 Unicode 转义会被转换成等价的单个字符进行存储，这包括把代理对折叠成一个单一字符。

### Note

很多 Section 9.16 中描述的 JSON 处理函数将把 Unicode 转义转换成常规字符，并且将因此抛出和刚才所描述的同样类型的错误（即使它们的输入是类型 json 而不是 jsonb）。json 的输入函数不做这些检查是由来已久的，不过它确实允许将 JSON Unicode 转义简单的（不经处理）存储在一个不支持所表示字符的数据库编码中。

在把文本 JSON 输入转换成 jsonb 时，RFC 7159 描述的基本类型会被有效地映射到原生的 LightDB 类型（如 Table 8.23 所示）。因此，在合法 jsonb 数据的组成上有一些次要额外约束，它们不适合 json 类型和抽象意义上的 JSON，这些约束对应于有关哪些东西不能被底层数据类型表示的限制。尤其是，jsonb 将拒绝位于 LightDB numeric 数据类型范围之外的数字，而 json 则不会。这类实现定义的限制是 RFC 7159 所允许的。不过，实际上这类问题更可能发生在其他实现中，因为把 JSON 的 number 基本类型表示为 IEEE 754 双精度浮点是很常见的（这也是 RFC 7159 明确期待和允许的）。当在这类系统间使用 JSON 作为一种交换格式时，应该考虑丢失数字精度的风险。

相反地，如表中所述，有一些 JSON 基本类型输入格式上的次要限制并不适用于相应的 LightDB 类型。

Table 8.23. JSON 基本类型和相应的 LightDB 类型

JSON 基本类型	LightDB 类型	注释
string	text	\u0000 是不允许的，因为 Unicode 转义表示数据库编码中不可用的字符
number	numeric	不允许 NaN 和 infinity 值
boolean	boolean	只接受小写 true 和 false 拼写
null	(无)	SQL NULL 是一个不同的概念

## 8.14.1. JSON 输入和输出语法

RFC 7159 中定义了 JSON 数据类型的输入/输出语法。

下列都是合法的 json（或者 jsonb）表达式：

— 简单标量/基本值  
 — 基本值可以是数字、带引号的字符串、true、false 或者 null  
 SELECT '5'::json;

— 有零个或者更多元素的数组（元素不需要为同一类型）  
 SELECT '[1, 2, "foo", null]'::json;



```
-- 包含键值对的对象
-- 注意对象键必须总是带引号的字符串
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;
```

```
-- 数组和对象可以被任意嵌套
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

如前所述，当一个 JSON 值被输入并且接着不做任何附加处理就输出时，json 会输出和输入完全相同的文本，而 jsonb 则不会保留语义上没有意义的细节（例如空格）。例如，注意下面的不同：

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
           json
```

```
-----
{"bar": "baz", "balance": 7.77, "active":false}
(1 row)
```

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
           jsonb
```

```
-----
{"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

值得一提的一种语义上无意义的细节是，在 jsonb 中数据会被按照底层 numeric 类型的行为来打印。实际上，这意味着用 E 记号输入的数字被打印出来时就不会有该记号，例如：

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
           json           |           jsonb
```

```
-----+-----
{"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

不过，如这个例子所示，jsonb 将会保留拖尾的小数点后的零，即便这对于等值检查等目的来说是语义上无意义的。

对于可用于构造和处理 JSON 值的内置函数和运算符的列表，参见 Section 9.16

## 8.14.2. 设计 JSON 文档

将数据表示为 JSON 比传统关系数据模型要灵活得多，在需求不固定时这种优势更加令人感兴趣。在同一个应用里非常有可能有两种方法共存并且互补。不过，即便是在要求最大灵活性的应用中，我们还是推荐 JSON 文档有固定的结构。该结构通常是非强制的（尽管可能会强制一些业务规则），但是有一个可预测的结构会使书写概括一个表中的“文档”（数据）集合的查询更容易。

当被存储在表中时，JSON 数据也像其他数据类型一样服从相同的并发控制考虑。尽管存储大型文档是可行的，但是要记住任何更新都在整行上要求一个行级锁。为了在更新事务之间减少锁争夺，可考虑把 JSON 文档限制到一个可管理的尺寸。理想情况下，JSON 文档应该每个表示一个原子数据，业务规则命令不会进一步把它们划分成更小的可独立修改的数据。

## 8.14.3. jsonb 包含和存在

测试包含是 jsonb 的一种重要能力。对 json 类型没有平行的功能集。包含测试会测试一个 jsonb 文档是否被包含在另一个文档中。除了特别注解之外，这些例子都会返回真：

---

```
-- 简单的标量/基本值只包含相同的值:
SELECT 'foo'::jsonb @> 'foo'::jsonb;

-- 右边的数字被包含在左边的数组中:
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- 数组元素的顺序没有意义, 因此这个例子也返回真:
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- 重复的数组元素也没有关系:
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- 右边具有一个单一键值对的对象被包含在左边的对象中:
SELECT '{"product": "LightDB", "version": 9.4, "jsonb": true}'::jsonb @>
'{"version": 9.4}'::jsonb;

-- 右边的数组不会被认为包含在左边的数组中,
-- 即使其中嵌入了一个相似的数组:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb; -- 得到假

-- 但是如果同样也有嵌套, 包含就成立:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- 类似的, 这个例子也不会被认为是包含:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb; -- 得到假

-- 包含一个顶层键和一个空对象:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

一般原则是被包含的对象必须在结构和数据内容上匹配包含对象, 这种匹配 可以从包含对象中丢弃了不匹配的数组元素或者对象键值对之后成立。但 是记住做包含匹配时数组元素的顺序是没有意义的, 并且重复的数组元素实际也只会考虑一次。

结构必须匹配的一般原则有一种特殊情况, 一个数组可以包含一个基本值:

```
-- 这个数组包含基本字符串值:
SELECT '["foo", "bar"]'::jsonb @> "bar"::jsonb;

-- 反之不然, 下面的例子会报告“不包含”:
SELECT "bar"::jsonb @> ["bar"]::jsonb; -- 得到假
```

jsonb还有一个存在操作符, 它是包含的一种 变体: 它测试一个字符串(以一个text值的形式给出)是否出 现在jsonb值顶层的一个对象键或者数组元素中。除非特别注解, 下面这些例子返回真:

```
-- 字符串作为一个数组元素存在:
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- 字符串作为一个对象键存在:
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- 不考虑对象值:
SELECT '{"foo": "bar"}'::jsonb ? 'bar'; -- 得到假

-- 和包含一样, 存在必须在顶层匹配:
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- 得到假
```

---

-- 如果一个字符串匹配一个基本 JSON 字符串，它就被认为存在：  
 SELECT 'foo'::jsonb ? 'foo';

当涉及很多键或元素时，JSON 对象比数组更适合于做包含或存在测试，因为它们不像数组，进行搜索时会进行内部优化，并且不需要被线性搜索。

### Tip

由于 JSON 的包含是嵌套的，因此一个恰当的查询可以跳过对子对象的显式选择。例如，假设我们在顶层有一个doc列包含着对象，大部分对象包含着tags域，其中有子对象的数组。这个查询会找到其中出现了同时包含"term":"paris"和"term":"food"的子对象的项，而忽略任何位于tags数组之外的这类键：

```
SELECT doc->'site_name' FROM websites
WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
```

可以用下面的查询完成同样的事情：

```
SELECT doc->'site_name' FROM websites
WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
```

但是后一种方法灵活性较差，并且常常也效率更低。

在另一方面，JSON 的存在操作符不是嵌套的：它将只在 JSON 值的顶层查找指定的键或数组元素。

Section 9.16中记录了多个包含和存在操作符，以及所有其他 JSON 操作符和函数。

## 8.14.4. jsonb 索引

GIN 索引可以被用来有效地搜索在大量jsonb文档（数据）中出现 的键或者键值对。提供了两种 GIN “操作符类”，它们在性能和灵活性方面做出了不同的平衡。

jsonb的默认 GIN 操作符类支持使用@>、?、?&以及?|操作符的查询（这些操作符实现的详细语义请见Table 9.46）。使用这种操作符类创建一个索引的例子：

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

非默认的 GIN 操作符类jsonb\_path\_ops只支持索引 @>操作符。使用这种操作符类创建一个索引的例子：

```
CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

考虑这样一个例子：一个表存储了从一个第三方 Web 服务检索到的 JSON 文档，并且有一个模式定义。一个典型的文档：

```
{
  "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
  "name": "Angela Barton",
  "is_active": true,
  "company": "MagnaFone",
  "address": "178 Howard Place, Gulf, Washington, 702",
```

```

    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}

```

我们把这些文档存储在一个名为api的表的名为 jdoc的jsonb列中。如果在这个列上创建一个 GIN 索引，下面这样的查询就能利用该索引：

```

-- 寻找键 "company" 有值 "Magnafone" 的文档
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company":
    "Magnafone"}';

```

不过，该索引不能被用于下面这样的查询，因为尽管操作符? 是可索引的，但它不能直接被应用于被索引列jdoc：

```

-- 寻找这样的文档：其中的键 "tags" 包含键或数组元素 "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';

```

但是，通过适当地使用表达式索引，上述查询也能使用一个索引。 如果对"tags"键中的特定项的查询很常见，可能值得 定义一个这样的索引：

```

CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));

```

现在，WHERE 子句 jdoc -> 'tags' ? 'qui' 将被识别为可索引操作符?在索引表达式jdoc -> 'tags' 上的应用（更多有关表达式索引的信息可见Section 11.7）。

查询的另一种方法是利用包含关系，例如：

```

-- Find documents in which the key "tags" contains array element "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';

```

在jdoc列上创建一个简单的GIN索引可以支持此查询。但请注意，这样的索引将存储jdoc列中每个键和值的副本，而前面示例中的表达式索引仅存储在tags键下找到的数据。虽然简单索引方法更加灵活（因为它支持关于任何键的查询），但定向表达式索引可能比简单索引更小、更快速地进行搜索。

GIN索引还支持@?和@@运算符，用于执行jsonpath匹配。例如：

```

SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @? '$.tags[*] ? (@ ==
    "qui")';

```

```

SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @@ '$.tags[*] == "qui";

```

对于这些运算符，GIN索引从jsonpath模式中提取形式为 `accessors_chain = constant`的子句，并根据这些子句中提到的键和值进行索引搜索。访问器链可以包括.key、[]和[index]访问器。jsonb\_ops操作符类还支持.和.\*\*访问器，但jsonb\_path\_ops操作符类不支持。

尽管jsonb\_path\_ops操作符类只支持用 @>, @@和@?操作符的查询，但它比起默认的操作符类 jsonb\_ops有更客观的性能优势。一个 jsonb\_path\_ops索引通常也比一个相同数据上的

jsonb\_ops要小得多，并且搜索的专一性更好，特别是当查询包含频繁出现在该数据中的键时。因此，其上的搜索操作通常比使用默认操作符类的搜索表现更好。

jsonb\_ops和jsonb\_path\_ops GIN 索引之间的技术区别是前者为数据中的每一个键和值创建独立的索引项，<sup>5</sup>而后者值为该数据中的每个值创建索引项。基本上，每一个jsonb\_path\_ops索引项是其所对应的值和键的哈希。例如要索引{"foo": {"bar": "baz"}}，将创建一个单一的索引项，它把所有三个foo、bar、和baz合并到哈希值中。因此一个查找这个结构的包含查询可能导致极度详细的索引搜索。但是根本没有办法找到foo是否作为一个键出现。在另一方面，一个jsonb\_ops会创建三个索引项分别表示foo、bar和baz。那么要做同样的包含查询，它将会查找包含所有三个项的行。虽然GIN索引能够相当有效地执行这种AND搜索，它仍然不如等效的jsonb\_path\_ops搜索那样详细和快速（特别是如果有大量行包含三个索引项中的任意一个时）。

jsonb\_path\_ops方法的一个不足是它不会为不包含任何值的JSON结构创建索引项，例如{"a": {}}。如果需要搜索包含这样一种结构的文档，它将要求一次全索引扫描，那就非常慢。因此jsonb\_path\_ops不适合经常执行这类搜索的应用。

jsonb也支持btree和hash索引。这通常值用于检查完整JSON文档等值非常重要的场合。jsonb数据的btree顺序很少有人关系，但是为了完整性其顺序是：

对象 > 数组 > 布尔 > 数字 > 字符串 > 空值

带有 n 对的对象 > 带有 n - 1 对的对象

带有 n 个元素的数组 > 带有 n - 1 个元素的数组

具有相同数量对的对象这样比较：

key-1, value-1, key-2 ...

注意对象键被按照它们的存储顺序进行比较，特别是由于较短的键被存储在较长的键之前，这可能导致结果不直观，例如：

```
{ "aa": 1, "c": 1 } > { "b": 1, "d": 1 }
```

相似地，具有相同元素数量的数组按照以下顺序比较：

element-1, element-2 ...

基本JSON值的比较会使用底层LightDB数据类型相同的比较规则进行。字符串的比较会使用默认的数据库排序规则。

## 8.14.5. jsonpath Type

在LightDB中，jsonpath类型实现支持SQL/JSON路径语言以有效地查询JSON数据。它提供了已解析的SQL/JSON路径表达式的二进制表示，该表达式指定路径引擎从JSON数据中检索的项，以便使用SQL/JSON查询函数进行进一步处理。

SQL/JSON路径谓词和运算符的语义通常遵循SQL。同时，为了提供使用JSON数据的自然方法，SQL/JSON路径语法使用一些JavaScript约定：

- 小数点 (.) 用于成员访问。
- 方括号 ([]) 用于数组访问。

<sup>5</sup> 对于这种目的，术语“值”包括数组元素，尽管JSON的术语有时认为数组元素与对象内的值不同。

- 与从 1 开始的常规 SQL 数组不同，SQL/JSON 数组是 0 相对的。

SQL/JSON路径表达式通常以SQL字符串文字形式写入SQL查询中，因此它必须用单引号括起来，并且值中需要的任何单引号都必须是双引号(参见 Section 4.1.2.1)。某些形式的路径表达式需要其中的字符串文本。这些嵌入的字符串文本遵循JavaScript/ECMAScript约定：它们必须用双引号括起来，并且反斜杠转义可以用于表示其他难以输入的字符。特别是，在嵌入字符串文本中编写双引号的方法为`\`，并且要编写反斜杠本身，你必须写`\\`。包括在JSON字符串中识别的其他特殊的反斜杠序列：`\b`、`\f`、`\n`、`\r`、`\t`、`\v` 对于各种 ASCII 控制字符，以及由它的4个六位数编码点标识标识的 Unicode 字符`\uNNNN`。反斜杠语法还包括JSON 不允许的两个案例：`\xNN` 对于只用两个十六进制数字编写的字符代码，以及`\u{N...}` 对于用 1 到 6 个十六进制数字编写的字符代码。

路径表达式由一系列路径元素组成，可以是以下任何一种：

- JSON基本类型的路径文字:Unicode文本、数字、真、假或空。
- Path variables listed in Table 8.2中列出的路径变量。
- Table 8.2中列出的访问器运算符。
- Section 9.16.2.2中列出的jsonpath 运算符和方法。
- 括号，可用于提供筛选器表达式或定义路径计算的顺序。

有关使用jsonpath具有 SQL/JSON 查询函数的表达式的详细信息，参见Section 9.16.2

Table 8.24. jsonpath 变量

变量	描述
\$	表示被查询的 JSON 值的变量(context item)。
\$varname	命名变量。其值可以由参数vars多个JSON处理函数设置。详细信息请参见 Table 9.48
@	表示筛选器表达式中路径计算结果的变量。

Table 8.25. jsonpath Accessors

访问器运算符	描述
.key ."\$varname"	返回具有指定密钥的对象成员的成员访问器。如果键名称是以 \$ 开头的命名变量，或者不符合标识符的 JavaScript 规则，则必须将其括在双引号中以使其成为字符串文字。
.*	通配符成员访问器，该访问器返回位于当前对象顶层的所有成员的值。
.**	递归通配符成员访问器，它处理当前对象JSON层次结构的所有级别，并返回所有成员值，而不管它们的嵌套级别如何。这是 LightDB SQL/JSON 标准的扩展。
.**{level} .**{start_level to end_level}	与 .** 类似，但仅选择 JSON 层次结构的指定级别。嵌套级别指定为整数。零级别对应于当前对象。要访问最低嵌套级别，可以使用last关键字。这是 LightDB SQL/JSON 标准的扩展。
[subscript, ...]	数组元素访问器。subscript 能够以两种形式给出：index 或 start_index 到 end_index。第一个窗体按其索引返回单个数组元素。第二个窗体按索引范围返回数组切片，包括对应于提供的元素start_indexend_index。  指定的index可以是整数，也可以是返回单个数值的表达式，该数值将自动转换为整数。零索引对应于第一个数

访问器运算符	描述
	组元素。你还可以使用last 关键字来表示最后一个数组元素，这对于处理未知长度的数组很有用。
[*]	返回所有数组元素的通配符数组元素访问器。

## 8.15. 数组

LightDB允许一个表中的列定义为变长多维数组。可以创建任何内建或用户定义的基类、枚举类型、组合类型或者域的数组。

### 8.15.1. 数组类型的定义

为了展示数组类型的使用，我们创建这样一个表：

```
CREATE TABLE sal_emp (
  name          text,
  pay_by_quarter integer[],
  schedule      text[][]
);
```

如上所示，一个数组数据类型可以通过在数组元素的数据类型名称后面加上方括号（[]）来命名。上述命令将创建一个名为sal\_emp的表，它有一个类型为text的列（name），一个表示雇员的季度工资的一维integer类型数组（pay\_by\_quarter），以及一个表示雇员每周日程表的二维text类型数组（schedule）。

CREATE TABLE的语法允许指定数组的确切大小，例如：

```
CREATE TABLE tictactoe (
  squares integer[3][3]
);
```

然而，当前的实现忽略任何提供的数组尺寸限制，即其行为与未指定长度的数组相同。

当前的实现也不会强制所声明的维度数。一个特定元素类型的数组全部被当作是相同的类型，而不论其尺寸或维度数。因此，在CREATE TABLE中声明数组的尺寸或维度数仅仅只是文档而已，它并不影响运行时的行为。

另一种符合SQL标准的语法是使用关键词ARRAY，可以用来定义一维数组。pay\_by\_quarter可以这样定义：

```
pay_by_quarter integer ARRAY[4],
```

或者，不指定数组尺寸：

```
pay_by_quarter integer ARRAY,
```

但是和前面一样，LightDB在任何情况下都不会强制尺寸限制。

### 8.15.2. 数组值输入

要把一个数组值写成一个文字常数，将元素值用花括号包围并用逗号分隔（如果你懂C，这和初始化结构的C语法没什么两样）。在任意元素值周围可以使用双引号，并且在元素值包含逗号或花括号时必须这样做（更多细节如下所示）。因此，一个数组常量的一般格式如下：

```
'{ val1 delim val2 delim ... }'
```

这里delim是类型的定界符，记录在类型的pg\_type项中。在LightDB发行提供的标准数据类型中，所有的都使用一个逗号(,)，除了类型box使用一个分号(;)。每个val可以是数组元素类型的一个常量，也可以是一个子数组。一个数组常量的例子是：

```
'{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}'
```

该常量是一个二维的，3乘3数组，它由3个整数子数组构成。

要设置一个数组常量的一个元素为NULL，在该元素值处写NULL（任何NULL的大写或小写变体都有效）。如果你需要一个真正的字符串值“NULL”，你必须在它两边放上双引号。

（这些种类的数组常数实际是Section 4.1.2.8中讨论的一般类型常量的一种特殊形式。常数最初被当做一个字符串，然后被传给数组的输入转换例程。有必要时可能需要一个显式的类型指定。）

现在我们可以展示一些INSERT语句：

```
INSERT INTO sal_emp
VALUES ('Bill',
       '{10000, 10000, 10000, 10000}',
       '{{"meeting", "lunch"}, {"training", "presentation"}}');
```

```
INSERT INTO sal_emp
VALUES ('Carol',
       '{20000, 25000, 25000, 25000}',
       '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

前两个插入的结果看起来像这样：

```
SELECT * FROM sal_emp;
name | pay_by_quarter | schedule
-----|-----|-----
Bill | {10000,10000,10000,10000} | {{meeting, lunch}, {training, presentation}}
Carol | {20000,25000,25000,25000} | {{breakfast, consulting}, {meeting, lunch}}
(2 rows)
```

多维数组的每一维都必须有相匹配的长度。不匹配会造成错误，例如：

```
INSERT INTO sal_emp
VALUES ('Bill',
       '{10000, 10000, 10000, 10000}',
       '{{"meeting", "lunch"}, {"meeting"}}');
ERROR: multidimensional arrays must have array expressions with matching
dimensions
```

ARRAY构造器语法也可以被用于：

```
INSERT INTO sal_emp
VALUES ('Bill',
       ARRAY[10000, 10000, 10000, 10000],
       ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);
```



```
INSERT INTO sal_emp
VALUES ('Carol',
ARRAY[20000, 25000, 25000, 25000],
ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

注意数组元素是普通SQL常数或表达式，例如，字符串文字使用单引号而不是双引号包围，因为双引号可以出现在一个数组文字中。ARRAY构造器语法的详细讨论请见Section 4.2.12

### 8.15.3. 访问数组

现在，我们可以在该表上运行一些查询。首先，我们展示如何访问一个数组中的一个元素。下面的查询检索在第二季度工资发生变化的雇员的名字：

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];
```

```
name
-----
Carol
(1 row)
```

数组下标写在方括号内。默认情况下，LightDB为数组使用了一种从1开始的编号习惯，即一个具有n个元素的数组从array[1]开始，结束于array[n]。

下面的查询检索所有员工第三季度的工资：

```
SELECT pay_by_quarter[3] FROM sal_emp;
```

```
pay_by_quarter
-----
10000
25000
(2 rows)
```

我们也可以访问一个数组的任意矩形切片或者子数组。一个数组切片可以通过在一个或多个数组维度上指定下界:上界来定义例如，下面的查询检索Bill在本周头两天日程中的第一项：

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';
```

```
schedule
-----
{{meeting}, {training}}
(1 row)
```

如果任何维度被写成一个切片，即包含一个冒号，那么所有的维度都被看成是切片对待。其中任何只有一个数字（无冒号）的维度被视作是从1到指定的数字。例如，下面例子中的[2]被认为是[1:2]：

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';
```

```
schedule
-----
{{meeting, lunch}, {training, presentation}}
(1 row)
```

为了避免和非切片情况搞混，最好在所有的维度上都使用切片语法，例如[1:2][1:1]而不是[2][1:1]。

可以省略一个切片说明符的lower-bound或者 upper-bound（亦可两者都省略），缺失的边界会被数组下标的上下限所替代。例如：

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';
```

```

schedule
-----
  {{lunch}, {presentation}}
(1 row)
```

```
SELECT schedule[:] [1:1] FROM sal_emp WHERE name = 'Bill';
```

```

schedule
-----
  {{meeting}, {training}}
(1 row)
```

如果数组本身为空或者任何一个下标表达式为空，访问数组下标表达式将会返回空值。如果下标超过了数组边界，下标表达式也会返回空值（这种情况不会抛出错误）。例如，如果schedule目前具有的维度是[1:3][1:2]，那么引用schedule[3][3]将得到NULL。相似地，使用错误的下标号引用一个数组会得到空值而不是错误。

如果数组本身或者任何一个下标表达式为空，则一个数组切片表达式也会得到空值。但是，在其他情况例如选择一个完全位于当前数组边界之外的切片时，一个切片表达式会得到一个空（零维）数组而不是空值（由于历史原因，这并不符合非切片行为）。如果所请求的切片和数组边界重叠，那么它会被缩减为重叠的区域而不是返回空。

任何数组值的当前维度可以使用array\_dims函数获得：

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';
```

```

array_dims
-----
  [1:2][1:2]
(1 row)
```

array\_dims产生一个text结果，它便于人类阅读但是不便于程序读取。Dimensions can also be retrieved with 也可以通过array\_upper和array\_lower来获得维度，它们将分别返回一个指定数组的上界和下界：

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';
```

```

array_upper
-----
           2
(1 row)
```

array\_length将返回一个指定数组维度的长度：

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';
```

```

array_length
-----
```

```

                2
(1 row)

```

cardinality返回一个数组中在所有维度上的元素总数。这实际上是调用unnest将会得到的行数：

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';
```

```

cardinality
-----
                4
(1 row)

```

## 8.15.4. 修改数组

一个数组值可以被整个替换：

```
UPDATE sal_emp SET pay_by_quarter = '{25000, 25000, 27000, 27000}'
WHERE name = 'Carol';
```

或者使用ARRAY表达式语法：

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000, 25000, 27000, 27000]
WHERE name = 'Carol';
```

一个数组也可以在一个元素上被更新：

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
WHERE name = 'Bill';
```

或者在一个切片上被更新：

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000, 27000}'
WHERE name = 'Carol';
```

也可以使用省略lower-bound或者 upper-bound的切片语法，但是只能用于更新一个不是NULL 或者零维的数组值（否则无法替换现有的下标界线）。

一个已存储的数组值可以被通过为其还不存在的元素赋值来扩大之。任何位于之前已存在的元素和新元素之间的位置都将被空值填充。例如，如果数组myarray目前有4个元素，在用一个更新对myarray[6]赋值后它将有6个元素，其中myarray[5]为空值。目前，采用这种方式扩大数组只允许使用在一维数组上。

带下标的赋值方式允许创建下标不是从1开始的数组。例如，我们可以为myarray[-2:7]赋值来创建一个下标值从-2到7的数组。

新的数组值也可以通过串接操作符||构建：

```
SELECT ARRAY[1, 2] || ARRAY[3, 4];
?column?
-----
{1, 2, 3, 4}
(1 row)
```

```
SELECT ARRAY[5, 6] || ARRAY[[1, 2], [3, 4]];
      ?column?
-----
 {{5, 6}, {1, 2}, {3, 4}}
(1 row)
```

串接操作符允许把一个单独的元素加入到一个一维数组的开头或末尾。它也能接受两个N维数组，或者一个N维数组和一个N+1维数组。

当一个单独的元素被加入到一个一维数组的开头或末尾时，其结果是一个和数组操作数具有相同下界下标的新数组。例如：

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
      array_dims
-----
 [0:2]
(1 row)
```

```
SELECT array_dims(ARRAY[1, 2] || 3);
      array_dims
-----
 [1:3]
(1 row)
```

当两个具有相同维度的数组被串接时，其结果保留左操作数的外维度的下界下标。结果将是一个数组，它由左操作数的每一个元素以及紧接着的右操作数的每一个元素。例如：

```
SELECT array_dims(ARRAY[1, 2] || ARRAY[3, 4, 5]);
      array_dims
-----
 [1:5]
(1 row)
```

```
SELECT array_dims(ARRAY[[1, 2], [3, 4]] || ARRAY[[5, 6], [7, 8], [9, 0]]);
      array_dims
-----
 [1:5][1:2]
(1 row)
```

当一个N维数组被放在另一个N+1维数组的前面或者后面时，结果和上面的例子相似。每一个N维子数组实际上是N+1维数组外维度的一个元素。例如：

```
SELECT array_dims(ARRAY[1, 2] || ARRAY[[3, 4], [5, 6]]);
      array_dims
-----
 [1:3][1:2]
(1 row)
```

一个数组也可以通过使用函数array\_prepend、array\_append或array\_cat构建。前两个函数仅支持一维数组，但array\_cat支持多维数组。一些例子：

```
SELECT array_prepend(1, ARRAY[2, 3]);
      array_prepend
-----
 {1, 2, 3}
(1 row)
```

```

SELECT array_append(ARRAY[1, 2], 3);
      array_append
-----
      {1, 2, 3}
(1 row)

SELECT array_cat(ARRAY[1, 2], ARRAY[3, 4]);
      array_cat
-----
      {1, 2, 3, 4}
(1 row)

SELECT array_cat(ARRAY[[1, 2], [3, 4]], ARRAY[5, 6]);
      array_cat
-----
      {{1, 2}, {3, 4}, {5, 6}}
(1 row)

SELECT array_cat(ARRAY[5, 6], ARRAY[[1, 2], [3, 4]]);
      array_cat
-----
      {{5, 6}, {1, 2}, {3, 4}}

```

在简单的情况中，上面讨论的串接操作符比直接使用这些函数更好。不过，由于串接操作符需要服务于所有三种情况，所以它的负担比较重，在有些情况下使用这些函数之一有助于避免混淆。例如：

```

SELECT ARRAY[1, 2] || ' {3, 4}'; -- 没有指定类型的文字被当做一个数组
      ?column?
-----
      {1, 2, 3, 4}

SELECT ARRAY[1, 2] || '7'; -- 这个也是
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL; -- 未修饰的 NULL 也是如此
      ?column?
-----
      {1, 2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL); -- 这可能才是想要的意思
      array_append
-----
      {1, 2, NULL}

```

在上面的例子中，解析器看到在串接操作符的一边看到了一个整数数组，并且在另一边看到了一个未确定类型的常量。它用来决定该常量类型的启发式规则是假定它和该操作符的另一个输入具有相同的类型——在这种情况下是整数数组。因此串接操作符表示 `array_cat` 而不是 `array_append`。如果这样做是错误的选择，它可以通过将该常量造型成数组的元素类型来修复。但是显式地使用 `array_append` 可能是一种最好的方案。

## 8.15.5. 在数组中搜索

要在一个数组中搜索一个值，每一个值都必须被检查。这可以手动完成，但是我们必须知道数组的尺寸。例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                           pay_by_quarter[2] = 10000 OR
                           pay_by_quarter[3] = 10000 OR
                           pay_by_quarter[4] = 10000;
```

但是这对于大型数组来说太过冗长，且在数组尺寸未知时无法使用。一种可选的方法可见Section 9.24 上面的查询可以被替换为：

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

此外，我们还可以查找所有元素值都为10000的数组所在的行：

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

另外，generate\_subscripts函数也可以用来完成类似的查找。例如：

```
SELECT * FROM
  (SELECT pay_by_quarter,
         generate_subscripts(pay_by_quarter, 1) AS s
   FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

该函数的描述见Table 9.63

我们也可以使用&&操作符来搜索一个数组，它会检查左操作数是否与右操作数重叠。例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

该操作符和其他数组操作符的进一步描述请见Section 9.19 如Section 11.所述，它可以 使用一个合适的索引来提速。

你也可以使用array\_position和array\_positions在一个 数组中搜索特定值。前者返回值在 数组中第一次出现的位置的下标。后者返回一个数组， 其中有该值在数组中的所有出现位 置的下标。例如：

```
SELECT array_position(ARRAY['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'], 'mon');
 array_position
```

```
-----
                2
```

```
(1 row)
```

```
SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
```

```
-----
      {1, 4, 8}
```

```
(1 row)
```

### Tip

数组不是集合，在其中搜索指定数组元素可能是数据设计失误的表现。考虑使用一个独立的表来替代，其中每一行都对应于一个数组元素。这将更有利于搜索，并且对于大量元素的可扩展性更好。

## 8.15.6. 数组输入和输出语法

一个数组值的外部文本表现由根据数组元素类型的I/O转换规则解释的项构成，并在其上加上修饰用于指示数组结构。修饰包括数组值周围的花括号（{和}）以及相邻项之间的定界字符。定界字符通常是一个逗号（,），但是也可能是别的：它由数组元素类型的typedelim设置决定。在LightDB发行版提供的标准数据类型中，除了box类型使用分号（;）之外，其他都是用逗号。在一个多维数组中，每一个维度（行、平面、方体等）都有其自己的花括号层次，且同层的被花括号限定的相邻实体之间也必须有定界符。

如果元素值是空字符串、包含花括号、包含定界字符、包含双引号、包含反斜线、包含空白或者匹配词NULL，数组输出例程将在元素值周围放上双引号。嵌在元素值中的双引号以及反斜线将被反斜线转义。对于数字数据类型可以安全地假设双引号绝不会出现，但是对于文本数据类型我们必须准备好处理可能出现亦可能不出现的引号。

默认情况下，一个数组的一个维度的下界索引值被设置为1。要表示具有其他下界的数组，数组下标的范围应在填充数组内容之前被显式地指定好。这种修饰包括在每个数组维度上下界周围的方括号（[]），以及上下界之间的一个冒号（:）定界符。数组维度修饰后面要跟一个等号（=）。例如：

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
FROM (SELECT ' [1:1][-2:-1][3:5]={{ {1, 2, 3}, {4, 5, 6} }}'::int[] AS f1) AS ss;
```

```
e1 | e2
---+---
 1 |  6
(1 row)
```

只有当数组的维度中有一个或多个的下界不为1时，数组输出例程才会在结果中包括维度。

如果为一个元素给定的值是NULL（或者是任何变体），该元素将被设置为NULL。任何引号或反斜线的存在将阻止这种行为，而允许为元素值输入“NULL”的字面意思。为了向后兼容LightDB的8.2之前的版本，可将array\_nulls配置参数设置为off来阻止将NULL识别为NULL。

如前所示，在写一个数组值时我们可以在任何单独数组元素周围使用引号。如果元素值可能混淆数组值分析器时，我们必须这样做。例如，包含花括号、逗号（或者数据类型的定界符）、双引号、反斜线或首尾有空白的元素必须使用双引号。空字符串和匹配单词NULL的字符串也必须使用双引号。要把一个双引号或反斜线放在一个使用了双引号的数组元素值中，需要在它前面放一个反斜线。作为一种选择，我们可以免去使用引号而使用反斜线转义的方式来保护可能被认为是数组语法的所有数据字符。

我们可以在左括号前面或右括号后面增加空白。我们也可以在任何单独的项之前或之后加上空白。在所有这些情况中空白将被忽略。但是，在被使用了双引号的元素中的空白以及周围有其他非空白字符的空白不会被忽略。

### Tip

在SQL命令中写数组值时，ARRAY构造器语法（见Section 4.2.12）常常比数组文字语法要更容易使用。在ARRAY中，单独的元素值可以使用不属于数组成员时的方式来书写。

## 8.16. 组合类型

一个组合类型表示一行或一个记录的结构，它本质上就是一个域名和它们数据类型的列表。LightDB允许把组合类型用在很多能用简单类型的地方。例如，一个表的一列可以被声明为一种组合类型。

## 8.16.1. 组合类型的声明

这里有两个定义组合类型的简单例子：

```
CREATE TYPE complex AS (
    r      double precision,
    i      double precision
);

CREATE TYPE inventory_item AS (
    name          text,
    supplier_id   integer,
    price         numeric
);
```

该语法堪比CREATE TABLE，不过只能指定域名和类型，当前不能包括约束（例如NOT NULL）。注意AS关键词是必不可少的，如果没有它，系统将认为用户想要的是一种不同类型的CREATE TYPE命令，并且你将得到奇怪的语法错误。

定义了类型之后，我们可以用它们来创建表：

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

或函数：

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

只要你创建了一个表，也会自动创建一个组合类型来表示表的行类型，它具有和表一样的名称。例如，如果我们说：

```
CREATE TABLE inventory_item (
    name          text,
    supplier_id   integer REFERENCES suppliers,
    price         numeric CHECK (price > 0)
);
```

那么和上面所示相同的inventory\_item组合类型将成为一种副产品，并且可以按上面所说的进行使用。不过要注意当前实现的一个重要限制：因为没有约束与一个组合类型相关，显示在表定义中的约束不会应用于表外组合类型的值（要解决这个问题，可以在该组合类型上创建一个域，并且把想要的约束应用为这个域上的CHECK约束）。

## 8.16.2. 构造组合值

要把一个组合值写作一个文字常量，将该域值封闭在圆括号中并且用逗号分隔它们。你可以在任何域值周围放上双引号，并且如果该域值包含逗号或圆括号则必须这样做（更多细节见下文）。这样，一个组合常量的一般格式是下面这样的：



```
'( val1 , val2 , ... )'
```

一个例子是：

```
'("fuzzy dice", 42, 1.99)'
```

这将是上文定义的inventory\_item类型的一个合法值。要让一个域为 NULL，在列表中它的位置上根本不写字符。例如，这个常量指定其第三个域为 NULL：

```
'("fuzzy dice", 42,)'
```

如果你写一个空字符串而不是 NULL，写上两个引号：

```
'("", 42,)'
```

这里第一个域是一个非 NULL 空字符串，第三个是 NULL。

（这些常量实际上只是Section 4.1.2.8中讨论的一般类型常量的特殊类型。该常量最初被当做一个字符串并且被传递给组合类型输入转换例程。有必要用一次显式类型说明来告知要把该常量转换成何种类型。）。

ROW表达式也能被用来构建组合值。在大部分情况下，比起使用字符串语法，这相当简单易用，因为你不必担心多层引用。我们已经在上文用过这种方法：

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

只要在表达式中有多于一个域，ROW 关键词实际上就是可选的，因此这些可以被简化成：

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

Section 4.2.18中更加详细地讨论了ROW表达式语法。

### 8.16.3. 访问组合类型

要访问一个组合列的一个域，可以写成一个点和域的名称，更像从一个表名中选择一个域。事实上，它太像从一个表名中选择，这样我们不得不使用圆括号来避免让解析器混淆。例如，你可能尝试从例子表on\_hand中选取一些子域：

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

这不会有用了，因为名称item会被当成是一个表名，而不是on\_hand的一个列名。你必须写成这样：

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

或者你还需要使用表名（例如在一个多表查询中），像这样：

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

现在加上括号的对象就被正确地解释为对item列的引用，然后可以从中选出子域。

只要你从一个组合值中选择一个域，相似的语法问题就适用。例如，要从一个返回组合值的函数的结果中选取一个域，你需要这样写：

```
SELECT (my_func(...)).field FROM ...
```

如果没有额外的圆括号，这将生成一个语法错误。

特殊的域名称\*表示“所有的域”，Section 8.16.5中有进一步的解释。

## 8.16.4. 修改组合类型

这里有一些插入和更新组合列的正确语法的例子。首先，插入或者更新一整个列：

```
INSERT INTO mytab (complex_col) VALUES((1.1, 2.2));
```

```
UPDATE mytab SET complex_col = ROW(1.1, 2.2) WHERE ...;
```

第一个例子忽略ROW，第二个例子使用它，我们可以用两者之一完成。

我们能够更新一个组合列的单个子域：

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

注意这里我们不需要（事实上也不能）把圆括号放在正好出现在SET之后的列名周围，但是当在等号右边的表达式中引用同一列时确实需要圆括号。

并且我们也可以指定子域作为INSERT的目标：

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

如果我们没有为该列的所有子域提供值，剩下的子域将用空值填充。

## 8.16.5. 在查询中使用组合类型

对于查询中的组合类型有各种特殊的语法规则和行为。这些规则提供了有用的捷径，但是如果你不懂背后的逻辑就会被此困扰。

在LightDB中，查询中对一个表名（或别名）的引用实际上是对该表的当前行的组合值的引用。例如，如果我们有一个如上所示的表inventory\_item，我们可以写：

```
SELECT c FROM inventory_item c;
```

这个查询产生一个单一组合值列，所以我们会得到这样的输出：

```

      c
-----
("fuzzy dice", 42, 1.99)
(1 row)
```

不过要注意简单的名称会在表名之前先匹配到列名，因此这个例子可行的原因仅仅是因为在该查询的表中没有名为c的列。

普通的限定列名语法table\_name.column\_name可以理解为把字段选择应用在该表的当前行的组合值上（由于效率的原因，实际上不是以这种方式实现）。

当我们写

```
SELECT c.* FROM inventory_item c;
```

时，根据SQL标准，我们应该得到该表展开成列的内容：

name	supplier_id	price
fuzzy dice	42	1.99

(1 row)

就好像查询是

```
SELECT c.name, c.supplier_id, c.price FROM inventory_item c;
```

尽管如上所示，LightDB将对任何组合值表达式应用这种展开行为，但只要.\*所应用的值不是一个简单的表名，你就需要把该值写在圆括号内。例如，如果myfunc()是一个返回组合类型的函数，该组合类型由列a、b和c组成，那么这两个查询有相同的结果：

```
SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;
```

### Tip

LightDB实际上通过将第一种形式转换为第二种来处理列展开。因此，在这个例子中，用两种语法时对每行都会调用myfunc()三次。如果它是一个开销很大的函数，你可能希望避免这样做，所以可以用一个这样的查询：

```
SELECT m.* FROM some_table, LATERAL myfunc(x) AS m;
```

把该函数放在一个LATERAL FROM项中会防止它对每一行被调用超过一次。m.\*仍然会被展开为m.a, m.b, m.c，但现在那些变量只是对这个FROM项的输出的引用（这里关键词LATERAL是可选的，但我们在这里写上它是为了说明该函数从some\_table中得到x）。

当composite\_value.\*出现在一个SELECT输出列表的顶层中、INSERT/UPDATE/DELETE中的一个RETURNING列表中、一个VALUES子句中或者一个行构造器中时，该语法会导致这种类型的列展开。在所有其他上下文（包括被嵌入在那些结构之一中时）中，把.\*附加到一个组合值不会改变该值，因为它表示“所有的列”并且因此同一个组合值会被再次产生。例如，如果somefunc()接受一个组合值参数，这些查询是相同的：

```
SELECT somefunc(c.*) FROM inventory_item c;
SELECT somefunc(c) FROM inventory_item c;
```

在两种情况中，inventory\_item的当前行被传递给该函数作为一个单一的组合值参数。即使.\*在这类情况中什么也不做，使用它也是一种好的风格，因为它说清了一个组合值的目的是什么。特别地，解析器将会认为c.\*中的c是引用一个表名或别名，而不是一个列名，这样就不会出现混淆。而如果没有.\*，就弄不清楚c到底是表示一个表名还是一个列名，并且在有一个名为c的列时会优先选择按列名来解释。

另一个演示这些概念的例子是下面这些查询，它们表示相同的东西：

```
SELECT * FROM inventory_item c ORDER BY c;
SELECT * FROM inventory_item c ORDER BY c.*;
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

所有这些ORDER BY子句指定该行的组合值，导致根据Section 9.24.6中介绍的规则对行进行排序。不过，如果inventory\_item包含一个名为c的列，第一种情况会不同于其他情况，因为它表示仅按那一列排序。给定之前所示的列名，下面这些查询也等效于上面的那些查询：

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id, c.price);
SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id, c.price);
```

（最后一种情况使用了一个省略关键字ROW的行构造器）。

另一种与组合值相关的特殊语法行为是，我们可以使用函数记法来抽取一个组合值的字段。解释这种行为的简单方式是记法field(table)和table.field是可以互换的。例如，这些查询是等效的：

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

此外，如果我们有一个函数接受单一的组合类型参数，我们可以以任意一种记法来调用它。这些查询全都是等效的：

```
SELECT somefunc(c) FROM inventory_item c;
SELECT somefunc(c.*) FROM inventory_item c;
SELECT c.somefunc FROM inventory_item c;
```

这种函数记法和字段记法之间的等效性使得我们可以在组合类型上使用函数来实现“计算字段”。一个使用上述最后一种查询的应用不会直接意识到somefunc不是一个真实的表列。

### Tip

由于这种行为，让一个接受单一组合类型参数的函数与该组合类型的任意字段具有相同的名称是不明智的。出现歧义时，如果使用了字段名语法，则字段名解释将被选择，而如果使用的是函数调用语法则会选择函数解释。不过，LightDB在版本11之前总是选择字段名解释，除非该调用的语法要求它是一个函数调用。在老的版本中强制函数解释的一种方法是用方案限定函数名，也就是写成schema.func(compositevalue)。

## 8.16.6. 组合类型输入和输出语法

一个组合值的外部文本表达由根据域类型的 I/O 转换规则解释的项，外加指示组合结构的装饰组成。装饰由整个值周围的圆括号（(和)），外加相邻项之间的逗号（,）组成。圆括号之外的空格会被忽略，但是在圆括号之内空格会被当成域值的一部分，并且根据域数据类型的输入转换规则可能有意义，也可能没有意义。例如，在

```
' ( 42)'
```

中，如果域类型是整数则空格会被忽略，而如果是文本则空格不会被忽略。

如前所示，在写一个组合值时，你可以在任意域值周围写上双引号。如果不这样做会让域值迷惑组合值解析器，你就必须这么做。特别地，包含圆括号、逗号、双引号或反斜线的域必须用双引号引用。要把一个双引号或者反斜线放在一个被引用的组合域值中，需要在它前面

放上一个反斜线（还有，一个双引号引用的域值中的一对双引号被认为是表示一个双引号字符，这和 SQL 字符串中单引号的规则类似）。另一种办法是，你可以避免引用以及使用反斜线转义来保护所有可能被当作组合语法的数据字符。

一个全空的域值（在逗号或圆括号之间完全没有字符）表示一个 NULL。要写一个空字符串值而不是 NULL，可以写成""。

如果域值是空串或者包含圆括号、逗号、双引号、反斜线或空格，组合输出例程将在域值周围放上双引号（对空格这样处理并不是不可缺少的，但是可以提高可读性）。嵌入在域值中的双引号及反斜线将被双写。

### Note

记住你在一个 SQL 命令中写的东西将首先被解释为一个字符串，然后才会被解释为一个组合。这就让你所需要的反斜线数量翻倍（假定使用了转义字符串语法）。例如，要在组合值中插入一个含有一个双引号和一个反斜线的text域，你需要写成：

```
INSERT ... VALUES ('("\\"));
```

字符串处理器会移除一层反斜线，这样在组合值解析器那里看到的就会是("\").接着，字符串被交给text数据类型的输入例程并且变成\"（如果我们使用的数据类型的输入例程也会特别处理反斜线，例如bytea，在命令中我们可能需要八个反斜线用来在组合域中存储一个反斜线）。美元引用（见Section 4.1.2.4 可以被用来避免双写反斜线。

### Tip

当在 SQL 命令中书写组合值时，ROW构造器语法通常比组合文字语法更容易使用。在ROW中，单个域值可以按照平时不是组合值成员的写法来写。

## 8.17. 范围类型

范围类型是表达某种元素类型（称为范围的subtype）的一个值的范围的数据类型。例如，timestamp的范围可以被用来表达一个会议室被保留的时间范围。在这种情况下，数据类型是tsrange（“timestamp range”的简写）而timestamp是 subtype。subtype 必须具有一种总体的顺序，这样对于元素值是在一个范围值之内、之前或之后就是界线清楚的。

范围类型非常有用，因为它们可以表达一种单一范围值中的多个元素值，并且可以很清晰地表达诸如范围重叠等概念。用于时间安排的时间和日期范围是最清晰的例子；但是价格范围、一种仪器的量程等等也都有用。

### 8.17.1. 内建范围类型和多范围类型

LightDB 带有下列内建范围类型：

- int4range — integer的范围
- int8range — bigint的范围
- numrange — numeric的范围
- tsrange — 不带时区的 timestamp的范围

- `tstzrange` — 带时区的 `timestamp` 的范围
- `daterange` — `date` 的范围

此外，你可以定义自己的范围类型，详见 `CREATE TYPE`。

## 8.17.2. 例子

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- 包含
SELECT int4range(10, 20) @> 3;

-- 重叠
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- 抽取上界
SELECT upper(int8range(15, 25));

-- 计算交集
SELECT int4range(10, 20) * int4range(15, 25);

-- 范围为空吗?
SELECT isempty(numrange(1, 5));
```

范围类型上的操作符和函数的完整列表可见 [Table 9.5](#) 以及 [Table 9.55](#)

## 8.17.3. 包含和排除边界

每一个非空范围都有两个界限，下界和上界。这些值之间的所有点都被包括在范围内。一个包含界限意味着边界点本身也被包括在范围内，而一个排除边界意味着边界点不被包括在范围内。

在一个范围的文本形式中，一个包含下界被表达为 “[” 而一个排除下界被表达为 “(”。同样，一个包含上界被表达为 “]” 而一个排除上界被表达为 “)””（详见 [Section 8.17.5](#)）。

函数 `lower_inc` 和 `upper_inc` 分别测试一个范围值的上下界。

## 8.17.4. 无限（无界）范围

一个范围的下界可以被忽略，意味着所有小于上界的值都被包括在范围中，例如 `(, 3]`。同样，如果范围的上界被忽略，那么所有比上界大的值都被包括在范围中。如果上下界都被忽略，该元素类型的所有值都被认为在该范围中。规定缺失的包括界限自动转换为排除，例如，`[, ]` 转换为 `(, )`。你可以认为这些缺失值为 `+/- 无穷大`，但它们是特殊范围类型值，并且被视为超出任何范围元素类型的 `+/- 无穷大` 值。

具有“infinity”概念的元素类型可以用它们作为显式边界值。例如，在时间戳范围，`[today, infinity)` 不包括特殊的 `timestamp` 值 `infinity`，尽管 `[today, infinity]` 包括它，就好比 `[today, )` 和 `[today, ]`。

函数 `lower_inf` 和 `upper_inf` 分别测试一个范围的无限上下界。

## 8.17.5. 范围输入/输出

一个范围值的输入必须遵循下列模式之一：

```
(lower-bound, upper-bound)
(lower-bound, upper-bound]
[lower-bound, upper-bound)
[lower-bound, upper-bound]
empty
```

圆括号或方括号指示上下界是否为排除的或者包含的。注意最后一个模式是empty，它表示一个空范围（一个不包含点的范围）。

lower-bound可以是作为 subtype 的合法输入的一个字符串，或者是空表示没有下界。同样，upper-bound可以是作为 subtype 的合法输入的一个字符串，或者是空表示没有上界。

每个界限值可以使用“（双引号）字符引用。如果界限值包含圆括号、方括号、逗号、双引号或反斜线时，这样做是必须的，因为否则那些字符会被认作范围语法的一部分。要把一个双引号或反斜线放在一个被引用的界限值中，就在它前面放一个反斜线（还有，在一个双引号引用的界限值中的一对双引号表示一个双引号字符，这与 SQL 字符串中的单引号规则类似）。此外，你可以避免引用并且使用反斜线转义来保护所有数据字符，否则它们会被当做返回语法的一部分。还有，要写一个是空字符串的界限值，则可以写成“”，因为什么都不写表示一个无限界限。

范围值前后允许有空格，但是圆括号或方括号之间的任何空格会被当做上下界值的一部分（取决于元素类型，它可能是也可能不是有意义的）。

### Note

这些规则与组合类型文字中书写域值的规则非常相似。更多注解请见Section 8.16.6

例子：

```
-- 包括 3，不包括 7，并且包括 3 和 7 之间的所有点
SELECT '[3,7)'::int4range;
```

```
-- 既不包括 3 也不包括 7，但是包括之间的所有点
SELECT '(3,7)'::int4range;
```

```
-- 只包括单独一个点 4
SELECT '[4,4]'::int4range;
```

```
-- 不包括点（并且将被标准化为 '空'）
SELECT '[4,4)'::int4range;
```

## 8.17.6. 构造范围和多范围

每一种范围类型都有一个与其同名的构造器函数。使用构造器函数常常比写一个范围文字常数更方便，因为它避免了对界限值的额外引用。构造器函数接受两个或三个参数。两个参数的形式以标准的形式构造一个范围（下界是包含的，上界是排除的），而三个参数的形式按照第三个参数指定的界限形式构造一个范围。第三个参数必须是下列字符串之一：

“( )”、“[ ]”、“( ]”或者 “[ )”。例如：

```
-- 完整形式是：下界、上界以及指示界限包含性/排除性的文本参数。
SELECT numrange(1.0, 14.0, '(]');
```

```
-- 如果第三个参数被忽略，则假定为 '[)'
```

```
SELECT numrange(1.0, 14.0);
```

-- 尽管这里指定了 '()', 显示时该值将被转换成标准形式, 因为 int8range 是一种离散范围类型 (见下文)。

```
SELECT int8range(1, 14, '()');
```

-- 为一个界限使用 NULL 导致范围在那一边是无界的。

```
SELECT numrange(NULL, 2.2);
```

## 8.17.7. 离散范围类型

一种范围的元素类型具有一个良定义的“步长”，例如integer或date。在这些类型中，如果两个元素之间没有合法值，它们可以被说成是相邻。这与连续范围相反，连续范围中总是（或者几乎总是）可以在两个给定值之间标识其他元素值。例如，numeric类型之上的一个范围就是连续的，timestamp上的范围也是（尽管timestamp具有有限的精度，并且在理论上可以被当做离散的，最好认为它是连续的，因为通常并不关心它的步长）。

另一种考虑离散范围类型的方法是对每一个元素值都有一种清晰的“下一个”或“上一个”值。了解了这种思想之后，通过选择原来给定的下一个或上一个元素值来取代它，就可以在一个范围界限的包含和排除表达之间转换。例如，在一个整数范围类型中，[4, 8]和(3, 9)表示相同的值集合，但是对于 numeric 上的范围就不是这样。

一个离散范围类型应该具有一个正规化函数，它知道元素类型期望的步长。正规化函数负责把范围类型的相等值转换成具有相同的表达，特别是与包含或者排除界限一致。如果没有指定一个正规化函数，那么具有不同格式的范围将总是会被当作不等，即使它们实际上是表达相同的一组值。

内建的范围类型int4range、int8range和daterange都使用一种正规的形式，该形式包括下界并且排除上界，也就是[]。不过，用户定义的范围类型可以使用其他习惯。

## 8.17.8. 定义新的范围类型

用户可以定义他们自己的范围类型。这样做最常见的原因是为了使用内建范围类型中没有提供的 subtype 上的范围。例如，要创建一个 subtype float8的范围类型：

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);
```

```
SELECT '[1.234, 5.678]':floatrange;
```

因为float8没有有意义的“步长”，我们在这个例子中没有定义一个正规化函数。

定义自己的范围类型也允许你指定使用一个不同的子类型 B-树操作符类或者集合，以便更改排序顺序来决定哪些值会落入到给定的范围中。

如果 subtype 被认为是具有离散值而不是连续值，CREATE TYPE命令应当指定一个canonical函数。正规化函数接收一个输入的范围值，并且必须返回一个可能具有不同界限和格式的等价的范围值。对于两个表示相同值集合的范围（例如[1, 7]和[1, 8)），正规的输出必须一样。选择哪一种表达作为正规的没有关系，只要两个具有不同格式的等价值总是能被映射到具有相同格式的相同值就行。除了调整包含/排除界限格式外，假使期望的补偿比 subtype 能够存储的要大，一个正规化函数可能会舍入边界值。例如，一个timestamp之上的范围类型可能被定义为具有一个一小时的步长，这样正规化函数可能需要对不是一小时的倍数的界限进行舍入，或者可能直接抛出一个错误。

另外，任何打算要和 GiST 索引一起使用的范围类型应当定一个 subtype 差异或subtype\_diff函数（没有subtype\_diff时索引仍然能工作，但是可能效率不如提供了差异



函数时高)。subtype 差异函数采用两个 subtype 输入值，并且返回表示为一个float8值的差（即X减Y）。在我们上面的例子中，可以使用常规float8减法操作符之下的函数。但是对于任何其他 subtype，可能需要某种类型转换。还可能需要一些关于如何把差异表达为数字的创新型想法。为了最大的可扩展性，subtype\_diff函数应该同意选中的操作符类和排序规则所蕴含的排序顺序，也就是说，只要它的第一个参数根据排序顺序大于第二个参数，它的结果就应该是正值。

subtype\_diff函数的一个不那么过度简化的例子：

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;
```

```
CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);
```

```
SELECT '[11:10, 23:00]'::timerange;
```

更多关于创建范围类型的信息请参考CREATE TYPE。

## 8.17.9. 索引

可以为范围类型的表列创建 GiST 索引。在多范围类型的列上也可以创建GiST 索引。例如，要创建一个 GiST 索引：

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

范围类型上使用 GiST 索引可以加速涉及以下范围操作符的查询： &&, <@, @>, <<, >>, -|- , &<, 和 &>（详见Table 9.54）。

此外，B-树和哈希索引可以在范围类型的表列上创建。对于这些索引类型，基本上唯一有用的范围操作就是等值。使用相应的< 和 >操作符，对于范围值定义有一种 B-树排序顺序，但是该顺序相当任意并且在真实世界中通常不怎么有用。范围类型的 B-树和哈希支持主要是为了允许在查询内部进行排序和哈希，而不是创建真正的索引。

## 8.17.10. 范围上的约束

虽然UNIQUE是标量值的一种自然约束，它通常不适合于范围类型。反而，一种排除约束常常更加适合（见CREATE TABLE ... CONSTRAINT ... EXCLUDE）。排除约束允许在一个范围类型上说明诸如“non-overlapping”的约束。例如：

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

该约束将阻止任何重叠值同时存在于表中：

```
INSERT INTO reservation VALUES
(' [2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1
```

```
INSERT INTO reservation VALUES
(' [2010-01-01 14:45, 2010-01-01 15:45)');
```

```
ERROR: conflicting key value violates exclusion constraint
"reservation_during_excl"
DETAIL: Key (during)=(["2010-01-01 14:45:00", "2010-01-01 15:45:00"]) conflicts
with existing key (during)=(["2010-01-01 11:30:00", "2010-01-01 15:00:00"]).
```

你可以使用**btree\_gist**扩展来在纯标量数据类型上定义排除约束，然后把它和范围排除结合可以得到最大的灵活性。例如，安装**btree\_gist**之后，只有会议室号码相等时，下列约束将拒绝重叠的范围：

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR: conflicting key value violates exclusion constraint
"room_reservation_room_during_excl"
DETAIL: Key (room, during)=(123A, ["2010-01-01 14:30:00", "2010-01-01
15:30:00"]) conflicts
with existing key (room, during)=(123A, ["2010-01-01 14:00:00", "2010-01-01
15:00:00"]).

INSERT INTO room_reservation VALUES
('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

## 8.18. 域类型

域是一种用户定义的数据类型，它基于另一种底层类型。根据需要，它可以有约束来限制其有效值为底层类型所允许值的一个子集。如果没有约束，它的行为就和底层类型一样——例如，任何适用于底层类型的操作符或函数都对该域类型有效。底层类型可以是任何内建或者用户定义的基础类型、枚举类型、数组类型、组合类型、范围类型或者另一个域。

例如，我们可以在整数之上创建一个域，它只接受正整数：

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1); -- works
INSERT INTO mytable VALUES(-1); -- fails
```

当底层类型的一个操作符或函数适用于一个域值时，域会被自动向下造型为底层类型。因此，`mytable.id - 1`的结果会被认为是类型**integer**而不是**posint**。我们可以写成`(mytable.id - 1)::posint`来把结果转换回**posint**，这会导致域的约束被重新检查。在这种情况下，如果该表达式被应用于一个值为1的id就会错误。把底层类型的值赋给域类型的一个字段或者变量不需要写显式的造型，但是域的约束将会被检查。

更多信息请参考**CREATE DOMAIN**。

## 8.19. 对象标识符类型

对象标识符（OID）被LightDB用来在内部作为多个系统表的主键。类型oid表示一个对象标识符。也有多个oid的别名类型分别被命名为regsomething。Table 8.2显示了一个概览。

oid类型目前被实现为一个无符号4字节整数。因此，在大型数据库中它并不足以提供数据库范围内的唯一性，甚至在一些大型的表中也无法提供表范围内的唯一性。

oid类型本身除了比较之外只有很少的操作。不过，它可以被造型成整数，并且接着可以使用标准的整数操作符进行操纵（这样做时要注意有符号和无符号之间可能出现的混乱）。

OID的别名类型除了特定的输入和输出例程之外没有别的操作。这些例程可以接受并显示系统对象的符号名，而不是类型oid使用的原始数字值。别名类型使查找对象的OID值变得简单。例如，要检查与一个表mytable有关的pg\_attribute行，你可以写：

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

而不是：

```
SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

虽然从它本身看起来并没有那么糟，它仍然被过度简化了。如果有多个名为mytable的表存在于不同的模式中，就可能需要一个更复杂的子选择来选择右边的OID。regclass输入转换器会根据模式路径设置处理表查找，并且因此它会自动地完成这种“右边的事情”。类似地，对于一个数字OID的符号化显示可以很方便地通过将表OID造型成regclass来实现。

Table 8.26. 对象标识符类型

名字	引用	描述	值示例
oid	任意	数字形式的对象标识符	564182
regclass	pg_class	关系名字	pg_type
regcollation	pg_collation	排序规则名称	"POSIX"
regconfig	pg_ts_config	文本搜索配置	english
regdictionary	pg_ts_dict	文本搜索字典	simple
regnamespace	pg_namespace	命名空间名称	pg_catalog
regoper	pg_operator	操作符名字	+
regoperator	pg_operator	带参数类型的操作符	*(integer, integer) or -(NONE, integer)
regproc	pg_proc	函数名字	sum
regprocedure	pg_proc	函数与参数类型	sum(int4)
regrole	pg_authid	角色名	smithee
regtype	pg_type	数据类型名称	integer

按命名空间分组的对象的所有OID别名类型都接受模式限定名称，如果在当前搜索路径中找不到未经限定的对象，则将在输出中显示模式限定名称。例

如，myschema.mytable是regclass的可接受输入（如果存在这样的表）。该值可能输出为myschema.mytable，或仅为mytable，具体取决于当前搜索路径。regproc和regoper别名类型将只接受唯一（未重载）的输入名称，因此它们的用途有限；对于大多数用途，regprocedure或regoperator更合适。对于regoperator，通过为未使用的操作数写入NONE来标识一元操作符。

大多数OID别名类型的另一个特性是创建依赖项。如果这些类型中的一个常量出现在存储的表达式（如一个列的默认表达式或视图）中，它将创建对引用对象的依赖关系。例如，如果

列具有默认表达式`nextval('my_seq'::regclass)`，LightDB知道默认表达式取决于序列`my_seq`，因此系统只允许在首先删除默认表达式的情况下才能删除序列。`nextval('my_seq'::text)`的替代方法不会创建依赖项。（`regrole`是此属性的例外。存储表达式中不允许使用此类型的常量。）

### Note

OID别名类型并不完全遵循事务隔离规则。规划者还将它们视为简单的常数，这可能会导致次优规划。

另一种系统中使用的标识符类型是`xid`，或者称为事务（简称为`xact`）标识符。这是系统列`xmin`和`xmax`使用的数据类型。事务标识符是32位量。在某种情况下，使用64位变量`xid8`。不像`xid`值，`xid8`值严格单调地增加，并且不能在数据库集群的生命周期中重用。

系统使用的第三种标识符类型是`cid`，或者称为命令标识符。这是系统列`cmin`和`cmax`使用的数据类型。命令标识符也是32位量。

系统使用的最后一种标识符类型是`tid`，或者称为元组标识符（行标识符）。这是系统列`ctid`使用的数据类型。一个元组ID是一个（块号，块内元组索引）对，它标识了行在它的表中的物理位置。

（这些系统列在Section 5.5中有进一步的解释）。

## 8.20. pg\_lsn 类型

`pg_lsn`数据类型可以被用来存储 LSN（日志序列号）数据，LSN 是一个指向WAL中的位置的指针。这个类型是`XLogRecPtr`的一种表达并且是 LightDB的一种内部系统类型。

在内部，一个 LSN 是一个 64 位整数，表示在预写式日志流中的一个字节位置。它被打印成两个最高 8 位的十六进制数，中间用斜线分隔，例如`16/B374D848`。`pg_lsn`类型支持标准的比较操作符，如`=`和`>`。两个 LSN 可以用`-`操作符做减法，结果将是分隔两个预写式日志位置的字节数。此外，还可以分别使用`+(pg_lsn, numeric)`和`-(pg_lsn, numeric)`运算符在LSN中添加和减去字节数。请注意，计算过的LSN应在`pg_lsn`类型的范围内，即`0/0`和`FFFFFFFF/FFFFFFFF`之间。

## 8.21. 伪类型

LightDB类型系统包含了一些特殊目的的项，它们被统称为伪类型。一个伪类型不能被用作一个列的数据类型，但是它可以被用来定义一个函数的参数或者结果类型。每一种可用的伪类型都有其可以发挥作用的情况，这些情况的特点是一个函数的行为并不能符合于简单使用或者返回一种特定SQL数据类型的值。Table 8.2列出了现有的伪类型。

Table 8.27. 伪类型

名字	描述
<code>any</code>	表示一个函数可以接受任意输入数据类型。
<code>anyelement</code>	表示一个函数可以接受任意数据类型（参见Section 36.2.5）。
<code>anyarray</code>	表示一个函数可以接受任意数组数据类型（参见Section 36.2.5）。
<code>anynonarray</code>	表示一个函数可以接受任意非数组数据类型（参见Section 36.2.5）。
<code>anyenum</code>	表示一个函数可以接受任意枚举数据类型（参见Section 36.2.5和Section 8.7）。

名字	描述
anyrange	表示一个函数可以接受任意范围数据类型（参见Section 36.2.4和Section 8.17）。
anycompatible	指示函数接受任何数据类型，并自动将多个参数提升为通用数据类型（参见 Section 36.2.5）。
anycompatiblearray	指示函数接受任何数组数据类型，并自动将多个参数提升为通用数据类型（参见 Section 36.2.5）。
anycompatiblenonarray	指示函数接受任何非数组数据类型，并将多个参数自动提升为通用数据类型（参见 Section 36.2.5）。
anycompatiblerange	指示函数接受任何范围数据类型，并将多个参数自动提升为通用数据类型（参见 Section 36.2.5and Section 8.17）。
cstring	表示一个函数接受或者返回一个非空结尾的C字符串。
internal	表示一个函数接受或返回一个服务器内部数据类型。
language_handler	一个被声明为返回language_handler的过程语言调用处理器。
fdw_handler	一个被声明为返回fdw_handler的外部数据包装器处理器。
table_am_handler	一种表访问方法处理程序，声明要返回table_am_handler。
index_am_handler	一个被声明为返回index_am_handler索引访问方法处理器。
tsm_handler	一个被声明为返回tsm_handler的表采样方法处理器。
record	标识一个接收或者返回一个未指定的行类型的函数。
trigger	一个被声明为返回trigger的触发器函数。
event_trigger	一个被声明为返回event_trigger的事件触发器函数。
pg_ddl_command	标识一种对事件触发器可用的 DDL 命令的表达。
void	表示一个函数不返回值。
unknown	标识一种还未被解析的类型，例如一个未修饰的字符文本。

用C编写的函数（不管是内建的还是动态载入的）可以被声明为接受或返回这些为数据类型的任何一种。函数的作者应当保证当一个伪数据类型被用作一个参数类型时函数的行为是安全的。

用过程语言编写的函数只有在其实现语言允许的情况下才能使用伪类型。目前大部分过程语言都禁止使用伪类型作为一种参数类型，并且只允许使用void和record作为结果类型（如果函数被用于一个触发器或者事件触发器，trigger或者event\_trigger也被允许作为结果类型）。有些还支持使用多态伪类型的多态函数，上述展现和讨论的详细内容在Section 36.2.5中。

internal伪类型用于定义只在数据库系统内部调用的函数，这些函数不会被SQL直接调用。如果一个函数拥有至少一个internal类型的参数，则它不能从SQL中被调用。为了保持这种限制的类型安全性，遵循以下编码规则非常重要：不要创建任何被声明要返回internal的函数，除非它有至少一个internal参数。

---

## Chapter 9. 函数和操作符

LightDB为内建的数据类型提供了大量的函数和操作符。本章描述了其中的大部分内容，不过在手册的相关章节中还显示了更多的特殊用途函数。用户也可以定义它们自己的函数和操作符，如Part 所述。ltsql命令\df和\do可以分别被用于显示所有可用的函数和操作符的列表。

本章中用于描述函数或操作符的参数和结果数据类型的符号如下所示：

```
repeat ( text, integer ) → text
```

函数repeat接受一个文本和一个整型参数，并返回文本类型的结果。右箭头也用于指示举例的结果，因此：

```
repeat('Pg', 4) → PgPgPgPg
```

如果你关心移植性，那么请注意，我们在本章描述的大多数函数和操作符，除了最琐碎的算术和比较操作符以及一些做了明确标记的函数以外，都没有在SQL标准里声明。某些这种扩展的功能也出现在许多其它SQL数据库管理系统中，并且在很多情况下多个实现的这种功能是相互兼容的和一致的。

### 9.1. 逻辑操作符

常用的逻辑操作符有：

```
boolean AND boolean → boolean
```

```
boolean OR boolean → boolean
```

```
NOT boolean → boolean
```

SQL使用三值的逻辑系统，包括真、假和null，null表示“未知”。观察下面的真值表：

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

操作符AND和OR是可交换的，也就是说，你可以交换左右操作数而不影响结果。（但是，不能保证左操作数在右操作数之前计算。参见Section 4.2.1获取有关子表达式计算顺序的更多信息。）

### 9.2. 比较函数和操作符

常见的比较操作符都可用，如Table 9. 所示。

Table 9.1. 比较操作符

操作符	描述
<code>datatype &lt; datatype → boolean</code>	小于
<code>datatype &gt; datatype → boolean</code>	大于
<code>datatype &lt;= datatype → boolean</code>	小于等于
<code>datatype &gt;= datatype → boolean</code>	大于等于
<code>datatype = datatype → boolean</code>	等于
<code>datatype &lt;&gt; datatype → boolean</code>	不等于
<code>datatype != datatype → boolean</code>	不等于

## Note

`<>` 表示 “not equal” 的标准SQL符号。 `!=` 是一个别名，在解析的早期阶段被转换为 `<>`。因此，它不可能实现 `!=` 和 `<>` 操作符以做不同的事情。

这些比较操作符适用于所有具有自然排序的内置数据类型，包括数字、字符串、和日期/时间类型。此外，如果它们的组件数据类型具有可比性，则可以比较数组、复合类型和范围。

通常也可以比较相关数据类型的值；例如 `integer > bigint` 将起作用。这种排序的某些情况直接由 “cross-type” 比较操作符实现，但是，如果没有这种操作符，解析器将把不太通用的类型强制为更通用的类型，并应用后者的比较操作符。

如上所示，所有比较操作符都是二元操作符，返回 `boolean` 类型的值。因此，类似 `1 < 2 < 3` 的表达式是无效的。因为没有 `<` 操作符与 `3` 进行 `Boolean` 值比较。使用下面显示的 `BETWEEN` 谓词执行范围测试。

如Table 9.所示，也有一些比较谓词。它们的行为和操作符很像，但是具有 SQL 标准所要求的特殊语法。

Table 9.2. 比较谓词

谓词	描述 示例(s)
<code>datatype BETWEEN datatype AND datatype → boolean</code>	之间(包括范围端点)。 <code>2 BETWEEN 1 AND 3 → t</code> <code>2 BETWEEN 3 AND 1 → f</code>
<code>datatype NOT BETWEEN datatype AND datatype → boolean</code>	不在之间 (BETWEEN的否定)。 <code>2 NOT BETWEEN 1 AND 3 → f</code>
<code>datatype BETWEEN SYMMETRIC datatype AND datatype → boolean</code>	之间，在对两个端点值排序之后。 <code>2 BETWEEN SYMMETRIC 3 AND 1 → t</code>
<code>datatype NOT BETWEEN SYMMETRIC datatype AND datatype → boolean</code>	不在之间，在对两个端点值排序之后。 <code>2 NOT BETWEEN SYMMETRIC 3 AND 1 → f</code>
<code>datatype IS DISTINCT FROM datatype → boolean</code>	不相等，将空( <code>null</code> )视为可比值。 <code>1 IS DISTINCT FROM NULL → t</code> (而不是 <code>NULL</code> ) <code>NULL IS DISTINCT FROM NULL → f</code> (而不是 <code>NULL</code> )

谓词	描述 示例(s)
	datatype IS NOT DISTINCT FROM datatype → boolean 相等, 将空 (null) 视为可比值。 1 IS NOT DISTINCT FROM NULL → f (而不是 NULL) NULL IS NOT DISTINCT FROM NULL → t (而不是 NULL)
	datatype IS NULL → boolean 测试值是否为空。 1.5 IS NULL → f
	datatype IS NOT NULL → boolean 测试值是否不为空。 'null' IS NOT NULL → t
	datatype ISNULL → boolean 测试值是否为空 (非标准语法)。
	datatype NOTNULL → boolean 测试值是否不为空 (非标准语法)。
	boolean IS TRUE → boolean 测试布尔表达式是否为真。 true IS TRUE → t NULL::boolean IS TRUE → f (而不是 NULL)
	boolean IS NOT TRUE → boolean 测试布尔表达式是否为假或未知。 true IS NOT TRUE → f NULL::boolean IS NOT TRUE → t (而不是 NULL)
	boolean IS FALSE → boolean 测试布尔表达式是否为假。 true IS FALSE → f NULL::boolean IS FALSE → f (而不是 NULL)
	boolean IS NOT FALSE → boolean 测试布尔表达式是否为真或未知。 true IS NOT FALSE → t NULL::boolean IS NOT FALSE → t (而不是 NULL)
	boolean IS UNKNOWN → boolean 测试布尔表达式是否为未知。 true IS UNKNOWN → f NULL::boolean IS UNKNOWN → t (而不是 NULL)
	boolean IS NOT UNKNOWN → boolean 测试布尔表达式是否为真或假。 true IS NOT UNKNOWN → t NULL::boolean IS NOT UNKNOWN → f (而不是 NULL)

BETWEEN谓词可以简化范围测试:

a BETWEEN x AND y

等效于

a >= x AND a <= y

注意BETWEEN认为终点值是包含在范围内的。BETWEEN SYMMETRIC 就像BETWEEN, 除了没有要求AND的左边的参数小于或等于右边的参数。如果不是的话, 这两个参数将自动交换, 因此总是隐含一个非空范围。。



BETWEEN的各种变量都是以普通比较操作符的方式实现的，因此适用于任何可以比较的数据类型。

### Note

在BETWEEN语法中使用AND会与使用AND作为逻辑操作符产生歧义。为了解决这个问题，只允许有限的一组表达类型作为BETWEEN子句的第二个参数。如果您需要在BETWEEN中写一个更复杂的子表达式，在子表达式两边写上圆括号。

当有一个输入为空时，普通的比较操作符会得到空（表示“未知”），而不是真或假。例如，`7 = NULL`得到空，`7 <> NULL`也一样。如果这种行为不合适，可以使用`IS [ NOT ] DISTINCT FROM`谓词：

```
a IS DISTINCT FROM b
a IS NOT DISTINCT FROM b
```

对于非空输入，`IS DISTINCT FROM`和`<>`操作符一样。不过，如果两个输入都为空，它会返回假。而如果只有一个输入为空，它会返回真。类似地，`IS NOT DISTINCT FROM`对于非空输入的行为与`=`相同，但是当两个输入都为空时它返回真，并且当只有一个输入为空时返回假。因此，这些谓词实际上把空值当作一种普通数据值而不是“unknown”。

要检查一个值是否为空，使用下面的谓词：

```
expression IS NULL
expression IS NOT NULL
```

或者等效，但并不标准的谓词：

```
expression ISNULL
expression NOTNULL
```

不要写`expression = NULL`，因为NULL是不“等于”NULL的（控制代表一个未知的值，因此我们无法知道两个未知的数值是否相等）。

### Tip

有些应用可能要求表达式`expression = NULL`在`expression`得出空值时返回真。我们强烈建议这样的应用修改成遵循 SQL 标准。但是，如果这样修改不可能完成，那么我们可以使用配置变量`transform_null_equals`。如果打开它，LightDB将把`x = NULL`子句转换成`x IS NULL`。

如果`expression`是行值，那么当行表达式本身为非空值或者行的所有域为非空时`IS NULL`为真。由于这种行为，`IS NULL`和`IS NOT NULL`并不总是为行值表达式返回反转的结果，特别是，一个同时包含 `NULL` 和非空值的域将会对两种测试都返回假。在某些情况下，写成`row IS DISTINCT FROM NULL`或者`row IS NOT DISTINCT FROM NULL`会更好，它们只会检查整个行值是否为空而不需要在行的域上做额外的测试。

布尔值也可以使用下列谓词进行测试：

```
boolean_expression IS TRUE
boolean_expression IS NOT TRUE
```

```
boolean_expression IS FALSE
boolean_expression IS NOT FALSE
boolean_expression IS UNKNOWN
boolean_expression IS NOT UNKNOWN
```

这些谓词将总是返回真或假，从来不返回空值，即使操作数是空也如此。空值输入被当做逻辑值“未知”。请注意实际上IS UNKNOWN和IS NOT UNKNOWN分别与IS NULL和IS NOT NULL相同，只是输入表达式必须是布尔类型。

如Table 9.3所示，也有一些比较相关的函数可用。

Table 9.3. 比较函数

函数	描述 例子
<code>num_nonnulls ( VARIADIC "any" )</code>	<code>→ integer</code> 返回非空参数的数量。 <code>num_nonnulls(1, NULL, 2) → 2</code>
<code>num_nulls ( VARIADIC "any" )</code>	<code>→ integer</code> 返回空参数的数量。 <code>num_nulls(1, NULL, 2) → 1</code>

## 9.3. 数学函数和操作符

LightDB为很多类型提供了数学操作符。对于那些没有标准数学表达的类型（如日期/时间类型），我们将在后续小节中描述实际的行为。

Table 9.4 显示了可用于标准数字类型的数学操作符。除非另有说明，显示为可接受 `numeric_type` 的操作符对所有的 `smallint`、`integer`、`bigint`、`numeric`、`real` 和 `double precision`类型都可用。显示为可接受 `integral_type` 的操作符对 `smallint`、`integer` 和 `bigint`类型是可用的。除了特别说明之处，操作符的每种形式都返回与其参数相同的数据类型。涉及多个参数数据类型的调用，例如 `integer + numeric`，可通过使用这些列表中稍后出现的类型来解析。

Table 9.4. 数学操作符

操作符	描述 例子
<code>numeric_type + numeric_type</code>	<code>→ numeric_type</code> 加 <code>2 + 3 → 5</code>
<code>+ numeric_type</code>	<code>→ numeric_type</code> 一元加(无操作) <code>+ 3.5 → 3.5</code>
<code>numeric_type - numeric_type</code>	<code>→ numeric_type</code> 减 <code>2 - 3 → -1</code>
<code>- numeric_type</code>	<code>→ numeric_type</code> 否定 <code>- (-4) → 4</code>
<code>numeric_type * numeric_type</code>	<code>→ numeric_type</code> 乘 <code>2 * 3 → 6</code>
<code>numeric_type / numeric_type</code>	<code>→ numeric_type</code>

操作符	描述 例子
	除(对于整型, 除法将结果截断为零) $5.0 / 2 \rightarrow 2.5000000000000000$ $5 / 2 \rightarrow 2$ $(-5) / 2 \rightarrow -2$
	<code>numeric_type % numeric_type</code> $\rightarrow$ <code>numeric_type</code> 模(取余); 适用于 <code>smallint</code> , <code>integer</code> , <code>bigint</code> 和 <code>numeric</code> $5 \% 4 \rightarrow 1$
	<code>numeric ^ numeric</code> $\rightarrow$ <code>numeric</code> <code>double precision ^ double precision</code> $\rightarrow$ <code>double precision</code> 指数(不像典型的数学实践, 多次使用 <code>^</code> 将会从左到右关联) $2 ^ 3 \rightarrow 8$ 与典型的数学惯例不同, <code>^</code> 的多次使用默认情况下从左到右关联: $2 ^ 3 ^ 3 \rightarrow 512$ $2 ^ (3 ^ 3) \rightarrow 134217728$
	<code>sqrt</code> <code>double precision</code> $\rightarrow$ <code>double precision</code> 平方根 $\sqrt{25.0} \rightarrow 5$
	<code>cbrt</code> <code>double precision</code> $\rightarrow$ <code>double precision</code> 立方根 $\sqrt[3]{64.0} \rightarrow 4$
	<code>bigint !</code> $\rightarrow$ <code>numeric</code> 阶乘(已弃用, 使用 <code>factorial()</code> 代替) $5 ! \rightarrow 120$
	<code>!! bigint</code> $\rightarrow$ <code>numeric</code> 阶乘作为前缀操作符(已弃用, 使用 <code>factorial()</code> 代替) $!! 5 \rightarrow 120$
	<code>@ numeric_type</code> $\rightarrow$ <code>numeric_type</code> 绝对值 $@ -5.0 \rightarrow 5$
	<code>integral_type &amp; integral_type</code> $\rightarrow$ <code>integral_type</code> 按位与(AND) $91 \& 15 \rightarrow 11$
	<code>integral_type   integral_type</code> $\rightarrow$ <code>integral_type</code> 按位或(OR) $32   3 \rightarrow 35$
	<code>integral_type \# integral_type</code> $\rightarrow$ <code>integral_type</code> 按位异或(exclusive OR) $17 \# 5 \rightarrow 20$
	<code>~ integral_type</code> $\rightarrow$ <code>integral_type</code> 按位求反(NOT) $\sim 1 \rightarrow -2$
	<code>integral_type &lt;&lt; integer</code> $\rightarrow$ <code>integral_type</code> 按位左移 $1 \ll 4 \rightarrow 16$
	<code>integral_type &gt;&gt; integer</code> $\rightarrow$ <code>integral_type</code> 按位右移 $8 \gg 2 \rightarrow 2$

Table 9.5 显示了可用的数学函数。许多这样的函数以多种具有不同的参数类型的形式提供。除非注明, 任何给定形式的函数都返回与其参数相同的数据类型; 跨类型情况的解决方

法与上述对操作符的解释相同。使用double precision数据的函数大多是在主机系统的C库上实现的；因此，边界情况下的准确性和行为会因主机系统的区别而不同。

Table 9.5. 数学函数

函数	描述 例子
abs ( numeric_type ) → numeric_type	绝对值 abs(-17.4) → 17.4
cbirt ( double precision ) → double precision	立方根 cbirt(64.0) → 4
ceil ( numeric ) → numeric ceil ( double precision ) → double precision	大于或等于参数的最接近的整数 ceil(42.2) → 43 ceil(-42.8) → -42
ceiling ( numeric ) → numeric ceiling ( double precision ) → double precision	大于或等于参数的最接近的整数 (与 ceil 相同) ceiling(95.3) → 96
degrees ( double precision ) → double precision	将弧度转换为角度 degrees(0.5) → 28.64788975654116
div ( y numeric, x numeric ) → numeric	y/x 的整数商 (截断为零位) div(9,4) → 2
exp ( numeric ) → numeric exp ( double precision ) → double precision	指数 (e 的给定次方) exp(1.0) → 2.7182818284590452
factorial ( bigint ) → numeric	阶乘 factorial(5) → 120
floor ( numeric ) → numeric floor ( double precision ) → double precision	小于或等于参数的最接近整数 floor(42.8) → 42 floor(-42.8) → -43
gcd ( numeric_type, numeric_type ) → numeric_type	最大公约数 (能将两个输入数整除而无余数的最大正数)；如果两个输入为零则返回 0；适用于 integer, bigint, 和 numeric gcd(1071, 462) → 21
lcm ( numeric_type, numeric_type ) → numeric_type	最小公倍数(两个输入的整数倍的最小的严格正数)；如果任意一个输入值为零则返回0；适用于integer, bigint, 和 numeric lcm(1071, 462) → 23562
ln ( numeric ) → numeric ln ( double precision ) → double precision	自然对数 ln(2.0) → 0.6931471805599453

函数	描述 例子
	<code>log ( numeric ) → numeric</code> <code>log ( double precision ) → double precision</code> 以10为底的对数 <code>log(100) → 2</code>
	<code>log10 ( numeric ) → numeric</code> <code>log10 ( double precision ) → double precision</code> 以10为底的对数 (与 <code>log</code> 相同) <code>log10(1000) → 3</code>
	<code>log ( b numeric, x numeric ) → numeric</code> 以 <code>b</code> 为底的 <code>x</code> 的对数 <code>log(2.0, 64.0) → 6.0000000000</code>
	<code>min_scale ( numeric ) → integer</code> 精确表示所提供值所需的最小刻度 (小数位数) <code>min_scale(8.4100) → 2</code>
	<code>mod ( y numeric_type, x numeric_type ) → numeric_type</code> <code>y/x</code> 的余数; 适用于 <code>smallint</code> 、 <code>integer</code> 、 <code>bigint</code> 、和 <code>numeric</code> <code>mod(9, 4) → 1</code>
	<code>pi ( ) → double precision</code> $\pi$ 的近似值 <code>pi() → 3.141592653589793</code>
	<code>power ( a numeric, b numeric ) → numeric</code> <code>power ( a double precision, b double precision ) → double precision</code> <code>a</code> 的 <code>b</code> 次幂 <code>power(9, 3) → 729</code>
	<code>radians ( double precision ) → double precision</code> 将角度转换为弧度 <code>radians(45.0) → 0.7853981633974483</code>
	<code>round ( numeric ) → numeric</code> <code>round ( double precision ) → double precision</code> 四舍五入到最近的整数 <code>round(42.4) → 42</code>
	<code>round ( v numeric, s integer ) → numeric</code> 把 <code>v</code> 四舍五入到 <code>s</code> 位小数 <code>round(42.4382, 2) → 42.44</code>
	<code>scale ( numeric ) → integer</code> 参数的刻度 (小数点后的位数) <code>scale(8.4100) → 4</code>
	<code>sign ( numeric ) → numeric</code> <code>sign ( double precision ) → double precision</code> 参数的符号 (-1, 0, 或 +1) <code>sign(-8.4) → -1</code>
	<code>sqrt ( numeric ) → numeric</code> <code>sqrt ( double precision ) → double precision</code> 平方根 <code>sqrt(2) → 1.4142135623730951</code>
	<code>trim_scale ( numeric ) → numeric</code> 通过删除尾数部分的零来降低值的刻度 (小数位数) <code>trim_scale(8.4100) → 8.41</code>
	<code>trunc ( numeric ) → numeric</code>

函数	描述 例子
<code>trunc</code>	<code>( double precision ) → double precision</code> 截断整数 (向零靠近) <code>trunc(42.8) → 42</code> <code>trunc(-42.8) → -42</code>
<code>trunc</code>	<code>( v numeric, s integer ) → numeric</code> 截断 <code>v</code> 到 <code>s</code> 位小数位置的数字 <code>trunc(42.4382, 2) → 42.43</code>
<code>width_bucket</code>	<code>( operand numeric, low numeric, high numeric, count integer ) → integer</code> <code>width_bucket ( operand double precision, low double precision, high double precision, count integer ) → integer</code> 返回包含 <code>count</code> 等宽柱的柱状图中 <code>operand</code> 所在的柱的编号, 范围从 <code>low</code> 到 <code>high</code> 。超出该范围的输入则返回0或计数+1。 <code>width_bucket(5.35, 0.024, 10.06, 5) → 3</code>
<code>width_bucket</code>	<code>( operand anyelement, thresholds anyarray ) → integer</code> 返回一个柱号, 这个柱是在给定数组中 <code>operand</code> 将被分配的柱。对于一个低于第一个下界的输入返回0。 <code>operand</code> 和数组元素可以是具有标准比较操作符的任何类型。 <code>thresholds</code> 数组必须被排好序, 最小的排在最前面, 否则将会得到意想不到的结果。 <code>width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestampz[]) → 2</code>

Table 9. 展示了用于产生随机数的函数。

Table 9.6. 随机函数

函数	描述 例子
<code>random</code>	<code>( ) → double precision</code> 返回一个范围 $0.0 \leq x < 1.0$ 中的随机值 <code>random() → 0.897124072839091</code>
<code>setseed</code>	<code>( double precision ) → void</code> 为后续的 <code>random()</code> 调用设置种子; 参数必须在-1.0和1.0之间, 包括边界值 <code>setseed(0.12345)</code>

`random()`函数使用了一个简单的线性共轭算法。 它的速度很快, 但不适合于密码学应用; 关于更安全的替代方案, 请参阅 `lcrypto`模块。 如果`setseed()`被调用, 那么当前会话中的一系列后续`random()`调用的结果能够通过使用相同的参数重新发布`setseed()`来重复。

Table 9. 显示了可用的三角函数。 每一种这样的函数都有两个变体, 一个以弧度度量角, 另一个以角度度量角。

Table 9.7. 三角函数

函数	描述 例子
<code>acos</code>	<code>( double precision ) → double precision</code> 反余弦, 结果为弧度 <code>acos(1) → 0</code>
<code>acosd</code>	<code>( double precision ) → double precision</code> 反余弦, 结果为度数

函数	描述 例子
	<code>acosd(0.5) → 60</code>
<code>asin ( double precision ) → double precision</code>	反正弦, 结果为弧度 <code>asin(1) → 1.5707963267948966</code>
<code>asind ( double precision ) → double precision</code>	反正弦, 结果为度数 <code>asind(0.5) → 30</code>
<code>atan ( double precision ) → double precision</code>	反正切, 结果为弧度 <code>atan(1) → 0.7853981633974483</code>
<code>atand ( double precision ) → double precision</code>	反正切, 结果为度数 <code>atand(1) → 45</code>
<code>atan2 ( y double precision, x double precision ) → double precision</code>	<code>y/x</code> 的反正切, 结果为弧度 <code>atan2(1,0) → 1.5707963267948966</code>
<code>atan2d ( y double precision, x double precision ) → double precision</code>	<code>y/x</code> 的反正切, 结果为度数 <code>atan2d(1,0) → 90</code>
<code>cos ( double precision ) → double precision</code>	余弦, 参数为弧度 <code>cos(0) → 1</code>
<code>cosd ( double precision ) → double precision</code>	余弦, 参数为度数 <code>cosd(60) → 0.5</code>
<code>cot ( double precision ) → double precision</code>	余切, 参数为弧度 <code>cot(0.5) → 1.830487721712452</code>
<code>cotd ( double precision ) → double precision</code>	余切, 参数为度数 <code>cotd(45) → 1</code>
<code>sin ( double precision ) → double precision</code>	正弦, 参数为弧度 <code>sin(1) → 0.8414709848078965</code>
<code>sind ( double precision ) → double precision</code>	正弦, 参数为度数 <code>sind(30) → 0.5</code>
<code>tan ( double precision ) → double precision</code>	正切, 参数为弧度 <code>tan(1) → 1.5574077246549023</code>
<code>tand ( double precision ) → double precision</code>	正切, 参数为度数 <code>tand(45) → 1</code>

## Note

另一种使用以角度度量的角的方法是使用早前展示的单位转换函数 `radians()` 和 `degrees()`。不过，使用基于角度的三角函数更好，因为这类方法能避免 `sind(30)` 等特殊情况下的舍入偏差。

Table 9. 显示的是可用的双曲函数。

Table 9.8. 双曲函数

函数	描述 例子
<code>sinh ( double precision )</code>	<code>→ double precision</code> 双曲正弦 <code>sinh(1) → 1.1752011936438014</code>
<code>cosh ( double precision )</code>	<code>→ double precision</code> 双曲余弦 <code>cosh(0) → 1</code>
<code>tanh ( double precision )</code>	<code>→ double precision</code> 双曲切线 <code>tanh(1) → 0.7615941559557649</code>
<code>asinh ( double precision )</code>	<code>→ double precision</code> 反双曲正弦 <code>asinh(1) → 0.881373587019543</code>
<code>acosh ( double precision )</code>	<code>→ double precision</code> 反双曲余弦 <code>acosh(1) → 0</code>
<code>atanh ( double precision )</code>	<code>→ double precision</code> 反双曲切线 <code>atanh(0.5) → 0.5493061443340548</code>

## 9.4. 字符串函数和操作符

本节描述了用于检查和操作字符串值的函数和操作符。在这个环境中的串包括所有类型 `character`、`character varying` 和 `text` 的值。除非特别说明，这些函数和操作符声明为接受并返回 `text` 类型。他们将互换接受 `character varying` 参数。在应用函数或操作符之前 `character` 类型的值将被转换为 `text`，结果删除 `character` 值中的任何末尾空格。

SQL 定义了一些字符串函数，它们使用关键字，而不是逗号来分隔参数。详情请见 Table 9.9 LightDB 也提供了这些函数使用正常函数调用语法的版本（见 Table 9.10）。

## Note

由于存在从那些数据类型到 `text` 的隐式强制措施，在 LightDB 8.3 之前，这些函数也可以接受多种非字符串数据类型。这些强制措施在目前的版本中已经被删除，因为它们常常导致令人惊讶的行为。不过，字符串串接操作符 (`||`) 仍然接受非字符串输入，只要至少一个输入是一种字符串类型，如 Table 9.9 所示。对于其他情况，如果你需要复制之前的行为，可以为 `text` 插入一个显式强制措施。



Table 9.9. SQL字符串函数和操作符

函数/操作符 描述 例子
<p><code>text    text → text</code> 连接两个字符串。 <code>'Light'    'DB' → LightDB</code></p>
<p><code>text    anynonarray → text</code> <code>anynonarray    text → text</code> 将非字符串输入转换为文本，然后将两个字符串串联在一起。（非字符串输入不能为数组类型，因为这将在<code>  </code>操作符的数组中造成歧义。如果你想连接一个数组的文本相等的，请显式地将其转换为<code>text</code>。） <code>'Value: '    42 → Value: 42</code></p>
<p><code>expr1    expr2 → text</code> 以下数据类型可用于 <code>expr1</code> 和 <code>expr2</code>: <code>smallint</code>, <code>integer</code>, <code>bigint</code>, <code>real</code>, <code>double precision</code>, <code>numeric</code>, <code>time</code>, <code>time with time zone</code>, <code>timestamp</code>, <code>timestamp with time zone</code>, <code>date</code>. <code>1    42 → 142</code></p>
<p><code>text IS [NOT] [form] NORMALIZED → boolean</code> 检查字符串是否在指定的 Unicode 规范化表单中。可选的<code>form</code>关键词指定表单: <code>NFC</code> (默认的), <code>NFD</code>, <code>NFKC</code>, 或 <code>NFKD</code>。只有在服务器编码为UTF8时, 才能使用此表达式。请注意, 使用这个表达式检查规范化通常比规范化可能已经规范化的字符串要快。 <code>U&amp;'0061\0308bc' IS NFD NORMALIZED → t</code></p>
<p><code>bit_length ( text ) → integer</code> 返回字符串中的位数(8倍于<code>octet_length</code>)。 <code>bit_length('jose') → 32</code></p>
<p><code>char_length ( text ) → integer</code> <code>character_length ( text ) → integer</code> 返回字符串中的字符数。 <code>char_length('josé') → 4</code></p>
<p><code>lower ( text ) → text</code> 根据数据库的语言环境规则, 将字符串转换为全部小写。 <code>lower('TOM') → tom</code></p>
<p><code>normalize ( text [, form ] ) → text</code> 将字符串转换为指定的Unicode规范化形式。可选的<code>form</code>关键字指定了如下形式:<code>NFC</code> (the default), <code>NFD</code>, <code>NFKC</code>, 或<code>NFKD</code>。该函数只能在服务器编码为UTF8时使用。 <code>normalize(U&amp;'0061\0308bc', NFC) → U&amp;'00E4bc'</code></p>
<p><code>octet_length ( text ) → integer</code> 返回字符串的字节数。 <code>octet_length('josé') → 5</code> (if server encoding is UTF8)</p>
<p><code>octet_length ( character ) → integer</code> 返回字符串中的字节数。由于此版本的函数直接接受<code>character</code>类型, 它不会剥离尾随空格。 <code>octet_length('abc '::character(4)) → 4</code></p>
<p><code>overlay ( string text PLACING newsubstring text FROM start integer [ FOR count integer ] ) → text</code> 替换<code>string</code>从<code>start</code>字符开始的子串, 并用<code>newsubstring</code>扩展到<code>count</code>字符。如果省略了<code>count</code>, 则默认为<code>newsubstring</code>的长度。 <code>overlay('Txxxxas' placing 'hom' from 2 for 4) → Thomas</code></p>
<p><code>position ( substring text IN string text ) → integer</code></p>

函数/操作符
描述 例子 返回指定的substring在string起始索引，如果不存在则返回零。 position('om' in 'Thomas') → 3
substring ( string text [ FROM start integer ] [ FOR count integer ] ) → text text 如果已指定，提取string从start字符开始的子串，并且在count字符后停止。如果已指定的话。提供至少一个start和count中的至少一个。 substring('Thomas' from 2 for 3) → hom substring('Thomas' from 3) → omas substring('Thomas' for 2) → Th
substring ( string text FROM pattern text ) → text 提取匹配POSIX正则表达式的子字符串；参见 Section 9.7.3 substring('Thomas' from '...\$') → mas
substring ( string text FROM pattern text FOR escape text ) → text 提取匹配 SQL 正则表达式的字符串；参见 Section 9.7.2 substring('Thomas' from '%#o_a#"_' for '#') → oma
trim ( [ LEADING   TRAILING   BOTH ] [ characters text ] FROM string text ) → text text 从string的开始、末端或两端(默认为BOTH)移除仅包含characters(默认为空格)字符的最长字符串。 trim(both 'xyz' from 'yxTomxx') → Tom
trim ( [ LEADING   TRAILING   BOTH ] [ FROM ] string text [, characters text ] ) → text text 这是一个非标准的trim()语法。 trim(both from 'yxTomxx', 'xyz') → Tom
upper ( text ) → text text 根据数据库的定位规则，将字符串转换为所有大写。 upper('tom') → TOM

还有额外的串操作函数可以用，它们在Table 9.10中列出。它们有些在内部用于实现Table 9.9列出的SQL标准字符串函数。

Table 9.10. 其他字符串函数

函数
描述 例子 ascii ( text ) → integer 返回参数的第一个字符的数字代码。在UTF8编码中，返回该字符的Unicode代码点。在其他多字节编码中，该参数必须是一个ASCII字符。 ascii('x') → 120
btrim ( string text [, characters text ] ) → text text 从string的开头或结尾删除最长的只包含characters（默认是一个空格）的字符串 btrim('yxtrimyyx', 'xyz') → trim
chr ( integer ) → text text 返回给定代码的字符。在UTF8编码中该参数被视作一个Unicode代码点。在其他多字节编码中该参数必须指定一个ASCII字符。Lightdb 23.3引入了一个新的数据库级别GUC参数lightdb_ascii_zero_store_value。在Oracle或Mysql模式下，如果lightdb_ascii_zero_store_value设置成非零ASCII码值，chr(0)返回ASCII码值为lightdb_ascii_zero_store_value对应的字符，其它场景下（包含非

函数	描述 例子
	Oracle以及非Mysql模式) chr(0) 字符不被允许, 因为文本数据类型不能存储这种字符。 lightdb_ascii_zero_store_value 的信息, 请参考 lightdb_ascii_zero_store_value chr(65) → A chr(0) → \x14 (如果兼容模式是Oracle 或 Mysql, 并且 lightdb_ascii_zero_store_value 设置为20)
	concat ( val1 "any" [, val2 "any" [, ...] ] ) → text 连接所有参数的文本表示。空参数被忽略。 concat('abcde', 2, NULL, 22) → abcde222
	concat_ws ( sep text, val1 "any" [, val2 "any" [, ...] ] ) → text 用分隔符连接除第一个参数外的所有参数。第一个参数用作分隔符字符串, 不应为NULL。其他NULL参数将被忽略。 concat_ws(',', 'abcde', 2, NULL, 22) → abcde,2,22
	format ( formatstr text [, formatarg "any" [, ...] ] ) → text 根据格式字符串对参数进行格式化; 参见 Section 9.4.1 这个函数类似于C函数 sprintf。 format('Hello %s, %1\$s', 'World') → Hello World, World
	initcap ( text ) → text 将每个单词的第一个字母转换为大写, 其余字母转换为小写。单词是由非字母数字字符分隔的字母数字字符序列。 initcap('hi THOMAS') → Hi Thomas
	left ( string text, n integer ) → text 以字符串返回第一个 n 字符, 或在 n 为负时, 返回最后  n  个字符之外的全部字符。 left('abcde', 2) → ab
	length ( text ) → integer 返回字符串中的字符数。 length('jose') → 4
	lpad ( string text, length integer [, fill text ] ) → text 将string扩展为长度length, 通过前置字符fill (默认空格)。如果string已经超过length那么它将被截断 (在右侧)。 lpad('hi', 5, 'xy') → xyxhi
	ltrim ( string text [, characters text ] ) → text 从string开始删除包含characters (默认空格) 中仅包含字符的最长字符串。 ltrim('zzytest', 'xyz') → test
	md5 ( text ) → text 计算参数的 MD5 hash, 结果以十六进制形式写入。 md5('abc') → 900150983cd24fb0d6963f7d28e17f72
	parse_ident ( qualified_identifier text [, strict_mode boolean DEFAULT true ] ) → text[] 将qualified_identifier拆分为一个标识符数组, 删除单个标识符的任何引用。默认情况下, 最后一个标识符之后的额外字符被视为错误; 但是, 如果第二个参数为false, 则忽略这些额外的字符。(这种行为对于解析类似函数的对象的名称有作用。) 请注意, 此函数不会截断超长标识符。如果你想截断, 你可以把结果给到name[]。 parse_ident('"SomeSchema".someTable') → {SomeSchema, sometable}
	pg_client_encoding ( ) → name 返回当前客户端编码名称。 pg_client_encoding() → UTF8

函数	描述 例子
<code>quote_ident ( text )</code>	<code>→ text</code> 返回适合引用的给定字符串，作为SQL语句字符串中的标识符。只有在必要的情况下才添加引号(例如，如果字符串包含非标识符字符或将被大小写折叠)。嵌入的引号被适当地加双引号。参见Example 41.1 <code>quote_ident(' Foo bar') → "Foo bar"</code>
<code>quote_literal ( text )</code>	<code>→ text</code> 返回在SQL语句字符串中适当引用的给定字符串，用作字符串文字使用。嵌入式单引号和反斜线适当的翻倍(转双引号或双斜线)。请注意， <code>quote_literal</code> 返回无效输入；如果这个参数可能为空， <code>quote_nullable</code> 通常更合适。另请参阅Example 41.1 <code>quote_literal(E' O\ 'Reilly') → 'O' Reilly'</code>
<code>quote_literal ( anyelement )</code>	<code>→ text</code> 将给定的值转换为文本，然后将其作为字面量引用。内嵌的单引号和反斜杠被适当地翻倍。 <code>quote_literal(42.5) → '42.5'</code>
<code>quote_nullable ( text )</code>	<code>→ text</code> 返回在SQL语句字符串中适当引用的给定字符串文字；或者，如果参数为null，则返回NULL。内嵌的单引号和反斜杠被适当地翻倍。参见Example 41.1 <code>quote_nullable(NULL) → NULL</code>
<code>quote_nullable ( anyelement )</code>	<code>→ text</code> 将给定值转换为文本，然后将其作为字面量引用；或者，如果参数为null，则返回NULL。内嵌的单引号和反斜杠被适当地翻倍。 <code>quote_nullable(42.5) → '42.5'</code>
<code>regexp_match ( string text, pattern text [, flags text ] )</code>	<code>→ text[]</code> 返回从POSIX正则表达式到string的第一个匹配中捕获的子字符串；参见Section 9.7.3 <code>regexp_match(' foobarbequebaz', '(bar)(beque)') → {bar, beque}</code>
<code>regexp_matches ( string text, pattern text [, flags text ] )</code>	<code>→ setof text[]</code> 返回通过将POSIX正则表达式与string匹配而捕获的子字符串；参见Section 9.7.3 <code>regexp_matches(' foobarbequebaz', 'ba.', 'g') →</code>  <code>{bar}</code> <code>{baz}</code>
<code>regexp_replace ( string text, pattern text, replacement text [, flags text ] )</code>	<code>→ text</code> 替换匹配POSIX正则表达式的子字符串；参见 Section 9.7.3 <code>regexp_replace('Thomas', '[mN]a.', 'M') → ThM</code>
<code>regexp_split_to_array ( string text, pattern text [, flags text ] )</code>	<code>→ text[]</code> 使用POSIX正则表达式作为分隔符拆分string；参见 Section 9.7.3 <code>regexp_split_to_array('hello world', '\s+') → {hello, world}</code>
<code>regexp_split_to_table ( string text, pattern text [, flags text ] )</code>	<code>→ setof text</code> 使用POSIX正则表达式作为分隔符拆分string；参见 Section 9.7.3 <code>regexp_split_to_table('hello world', '\s+') →</code>  <code>hello</code> <code>world</code>
<code>repeat ( string text, number integer )</code>	<code>→ text</code>

函数	描述 例子
	重复string指定number的次数。 repeat('Pg', 4) → PgPgPgPg
	replace ( string text, from text, to text ) → text 将string 中当前的子串from替换为子串to。 replace('abcdefabcdef', 'cd', 'XX') → abXXefabXXef
	reverse ( text ) → text 颠倒字符串中字符的顺序。 reverse('abcde') → edcba
	right ( string text, n integer ) → text 返回字符串中的最后n个字符, 或者在n>为负时, 返回除了前面的 n 字符之外的所有字符。 right('abcde', 2) → de
	rpad ( string text, length integer [, fill text ] ) → text 扩展 string 到长度 length, 通过追加fill 字符(默认为空格). 如果string 已经比 length 长, 则截断它。 rpad('hi', 5, 'xy') → hixyx
	rtrim ( string text [, characters text ] ) → text 从string末尾删除包含characters (默认为空格) 中仅包含字符的最长字符串。 rtrim('testxxzx', 'xyz') → test
	split_part ( string text, delimiter text, n integer ) → text 在delimiter出现时拆分string, 并且返回第n个字段(从一计数)。 split_part('abc~@~def~@~ghi', '~@~', 2) → def
	strpos ( string text, substring text ) → integer 返回在string中指定的substring的起始索引, 如果不存在则为零。 (与(substring 在 string中的)位置相同, 但是请注意反转的参数顺序) strpos('high', 'ig') → 2
	substr ( string text, start integer [, count integer ] ) → text 提取string从start字符开始的子字符串, 并扩展count字符, 如果指定了的话。 (与 子字符串(string 从 start 开始计数 count)相同。) substr('alphabet', 3) → phabet substr('alphabet', 3, 2) → ph
	starts_with ( string text, prefix text ) → boolean 如果 string 以 prefix开始就返回真。 starts_with('alphabet', 'alph') → t
	to_ascii ( string text ) → text to_ascii ( string text, encoding name ) → text to_ascii ( string text, encoding integer ) → text 将string从另一个编码中转换为ASCII, 该编码可按名称或编号标识。 如果encoding被省略, 则假定数据库编码 (这在实践中是唯一有用的案例)。转换主要包括降音。转换仅支持来自 LATIN1、LATIN2、LATIN9、 和 WIN1250 的编码。(其他请参见 unaccent 模块, 更灵活的解决方案。) to_ascii('Karél') → Karel
	to_hex ( integer ) → text to_hex ( bigint ) → text 将数字转换为其相应的十六进制表示形式。 to_hex(2147483647) → 7fffffff
	translate ( string text, from text, to text ) → text 将string中与from集合中匹配的每个字符替换为to集合中相应的字符。 如果from长于to, from中出现的额外字符被删除。 translate('12345', '143', 'ax') → a2x5

`concat`、`concat_ws`和`format`函数是可变的，因此可以把要串接或格式化的值作为一个标记了VARIADIC关键字的数组进行传递（见Section 36.5.5）。数组的元素被当作函数的独立普通参数一样处理。如果可变数组参数为 `NULL`，`concat`和`concat_ws`返回 `NULL`，但`format`把 `NULL` 当作一个零元素数组。

还可以参阅Section 9.2中的`string_agg`，以及Table 9.18中的字符串和`bytea`类型之间转换的功能。

## 9.4.1. format

函数`format`根据一个格式字符串产生格式化的输出，其形式类似于 C 函数`sprintf`。

```
format(formatstr text [, formatarg "any" [, ...] ])
```

`formatstr`是一个格式字符串，它指定了结果应该如何被格式化。格式字符串中的文本被直接复制到结果中，除了使用格式说明符的地方。格式说明符在字符串中扮演着占位符的角色，它定义后续的函数参数如何被格式化及插入到结果中。每一个`formatarg`参数会被根据其数据类型的常规输出规则转换为文本，并接着根据格式说明符被格式化和插入到结果字符串中。

格式说明符由一个`%`字符开始并且有这样的形式

```
%[position][flags][width]type
```

其中的各组件域是：

`position`（可选）

一个形式为`n$`的字符串，其中`n`是要打印的参数的索引。索引 1 表示`formatstr`之后的第一个参数。如果`position`被忽略，默认会使用序列中的下一个参数。

`flags`（可选）

控制格式说明符的输出如何被格式化的附加选项。当前唯一支持的标志是一个负号（-），它将导致格式说明符的输出会被左对齐（left-justified）。除非`width`域也被指定，否则这个域不会产生任何效果。

`width`（可选）

指定用于显示格式说明符输出的最小字符数。输出将被在左部或右部（取决于-标志）用空格填充以保证充满该宽度。太小的宽度设置不会导致输出被截断，但是会被简单地忽略。宽度可以使用下列形式之一指定：一个正整数；一个星号（\*）表示使用下一个函数参数作为宽度；或者一个形式为`*n$`的字符串表示使用第`n`个函数参数作为宽度。

如果宽度来自于一个函数参数，则参数在被格式说明符的值使用之前就被消耗掉了。如果宽度参数是负值，结果会在长度为`abs(width)`的域中被左对齐（如果-标志被指定）。

`type`（必需）

格式转换的类型，用于产生格式说明符的输出。支持下面的类型：

- `s`将参数值格式化为一个简单字符串。一个控制被视为一个空字符串。
- `I`将参数值视作 SQL 标识符，并在必要时用双引号包围它。如果参数为空，将会是一个错误（等效于`quote_ident`）。
- `L`将参数值引用为 SQL 文字。一个空值将被显示为不带引号的字符串`NULL`（等效于`quote_nullable`）。

除了以上所述的格式说明符之外，要输出一个文字形式的%字符，可以使用特殊序列%%。

下面有一些基本的格式转换的例子：

```
SELECT format('Hello %s', 'World');
结果: Hello World
```

```
SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
结果: Testing one, two, three, %
```

```
SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\Reilly');
结果: INSERT INTO "Foo bar" VALUES('O'Reilly')
```

```
SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
结果: INSERT INTO locations VALUES(E'C:\\Program Files')
```

下面是使用width域和-标志的例子：

```
SELECT format('|%10s|', 'foo');
结果: |          foo|
```

```
SELECT format('|%-10s|', 'foo');
结果: |foo          |
```

```
SELECT format('|%*s|', 10, 'foo');
结果: |          foo|
```

```
SELECT format('|%*s|', -10, 'foo');
结果: |foo          |
```

```
SELECT format('|%-*s|', 10, 'foo');
结果: |foo          |
```

```
SELECT format('|%-*s|', -10, 'foo');
结果: |foo          |
```

这些例子展示了position域的例子：

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
结果: Testing three, two, one
```

```
SELECT format('|%*2$s|', 'foo', 10, 'bar');
结果: |          bar|
```

```
SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
结果: |          foo|
```

不同于标准的 C 函数printf，LightDB的format函数允许将带有或者不带有position域的格式说明符被混在同一个格式字符串中。一个不带有position域的格式说明符总是使用最后一个被消耗的参数的下一个参数。另外，format函数不要求所有函数参数都被用在格式字符串中。例如：

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
结果: Testing three, two, three
```

对于安全地构造动态 SQL 语句，%I和%L格式说明符特别有用。参见Example 41.1

## 9.5. 二进制串函数和操作符

本节描述那些检查和操作二进制字符串的函数和操作符，这是类型bytea的值。其中许多函数在用途和语法上都与上一节中描述的文本字符串函数等效。

SQL定义了一些使用关键字而不是逗号来分割参数的串函数。详情请见Table 9.11 LightDB也提供了这些函数使用常规函数调用语法的版本（参阅Table 9.12）。

Table 9.11. SQL二进制串函数和操作符

函数/操作符 描述 例子
bytea    bytea → bytea 连接两个二进制字符串。 '\x123456'::bytea    '\x789a00bcde'::bytea → \x123456789a00bcde
bit_length ( bytea ) → integer 返回二进制字符串中的位数（8 倍于 octet_length）。 bit_length('\x123456'::bytea) → 24
octet_length ( bytea ) → integer 返回二进制字符串中的字节数。 octet_length('\x123456'::bytea) → 3
overlay ( bytes bytea PLACING newsubstring bytea FROM start integer [ FOR count integer ] ) → bytea 将bytes的子字符串替换为newsubstring，该子字符串从start字节开始，并以count字节扩展。如果忽略了count，则默认为newsubstring的长度。 overlay('\x1234567890'::bytea placing '\002\003'::bytea from 2 for 3) → \x12020390
position ( substring bytea IN bytes bytea ) → integer 返回指定的substring在bytes内的起始索引，如果不存在，则为零。 position('\x5678'::bytea in '\x1234567890'::bytea) → 3
substring ( bytes bytea [ FROM start integer ] [ FOR count integer ] ) → bytea 提取bytes从start字节开始的子字符串，如果指定了，并且在count字节之后停止，如果指定了的话。至少提供start和count中的一个。 substring('\x1234567890'::bytea from 3 for 2) → \x5678
trim ( [ BOTH ] bytesremoved bytea FROM bytes bytea ) → bytea 从bytes的开始和结束处删除bytesremoved中只包含字节的最长字符串。 trim('\x9012'::bytea from '\x1234567890'::bytea) → \x345678
trim ( [ BOTH ] [ FROM ] bytesremoved bytea, bytesremoved bytea ) → bytea 这是trim()的非标准语法。 trim(both from '\x1234567890'::bytea, '\x9012'::bytea) → \x345678

还有一些二进制串处理函数可以使用，在Table 9.1列出。其中有一些是在内部使用，用于实现Table 9.1列出的SQL标准串函数。

Table 9.12. 其他二进制串函数

函数 描述 例子
btrim ( bytes bytea, bytesremoved bytea ) → bytea 从bytes的开始和结束处删除只包含bytesremoved中出现的字节的最长字符串



函数	描述 例子
	<code>btrim('\x1234567890'::bytea, '\x9012'::bytea) → \x345678</code>
<code>get_bit ( bytes bytea, n bigint ) → integer</code>	从二进制字符串中提取 n'th 位。 <code>get_bit('\x1234567890'::bytea, 30) → 1</code>
<code>get_byte ( bytes bytea, n integer ) → integer</code>	从二进制字符串中提取 n'th 字节。 <code>get_byte('\x1234567890'::bytea, 4) → 144</code>
<code>length ( bytea ) → integer</code>	返回二进制字符串中的字节数。 <code>length('\x1234567890'::bytea) → 5</code>
<code>length ( bytes bytea, encoding name ) → integer</code>	返回二进制字符串中的字符数，假设它是给定encoding中的文本。 <code>length('jose'::bytea, 'UTF8') → 4</code>
<code>md5 ( bytea ) → text</code>	计算二进制字符串的MD5 hash，结果以十六进制形式写入。 <code>md5('Th\000omas'::bytea) → 8ab2d3c9689aaf18b4958c334c82d8b1</code>
<code>set_bit ( bytes bytea, n bigint, newvalue integer ) → bytea</code>	设置二进制字符串中的n'th位为newvalue。 <code>set_bit('\x1234567890'::bytea, 30, 0) → \x1234563890</code>
<code>set_byte ( bytes bytea, n integer, newvalue integer ) → bytea</code>	设置二进制字符串中的 n'th 字节到 newvalue。 <code>set_byte('\x1234567890'::bytea, 4, 64) → \x1234567840</code>
<code>sha224 ( bytea ) → bytea</code>	计算二进制字符串的 SHA-224 hash。 <code>sha224('abc'::bytea) → \x23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7</code>
<code>sha256 ( bytea ) → bytea</code>	计算二进制字符串的 SHA-256 hash。 <code>sha256('abc'::bytea) → \xba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad</code>
<code>sha384 ( bytea ) → bytea</code>	计算二进制字符串的 SHA-384 hash。 <code>sha384('abc'::bytea) → \xcb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7</code>
<code>sha512 ( bytea ) → bytea</code>	计算二进制字符串的 SHA-512 hash。 <code>sha512('abc'::bytea) → \xddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9e64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f</code>
<code>substr ( bytes bytea, start integer [, count integer ] ) → bytea</code>	从start字节开始提取bytes的子字符串，并扩展为count字节，如果这是指定的。（与 <code>substring(bytes 从 start 到 count)</code> 相同。） <code>substr('\x1234567890'::bytea, 3, 2) → \x5678</code>

函数`get_byte`和`set_byte`把一个二进制串中的一个字节计数为字节 0。函数`get_bit`和`set_bit`在每一个字节中从右边起计数位；例如位 0 是第一个字节的最低有效位，而位 15 是第二个字节的最高有效位。

由于历史原因，函数`md5`返回的是一个十六进制编码的text值，而SHA-2函数返回类型`bytea`。可以使用函数`encode`和`decode`在两者之间转换。

例

如`encode(sha256('abc'), 'hex')`可以得到一个十六进制编码的文本表示，或者`decode(md5('abc'), 'hex')`得到一个bytea 值。

用于在不同字符集(编码)之间转换字符串的函数，以及用于以文本形式表示任意二进制数据的函数，在Table 9.13中显示。对于这些函数，类型为text的参数或结果表示为数据库的默认编码，而类型为bytea的参数或结果表示为由另一个参数命名的编码。

Table 9.13. Text/Binary String Conversion Functions

函数	描述 例子
<code>convert ( bytes bytea, src_encoding name, dest_encoding name )</code>	<code>→ bytea</code> 将表示编码src_encoding的文本的二进制字符串转换为编码dest_encoding的二进制字符串(适用的转换请参阅Section 21.3.4)。 <code>convert('text_in_utf8', 'UTF8', 'LATIN1') → \x746578745f696e5f75746638</code>
<code>convert_from ( bytes bytea, src_encoding name )</code>	<code>→ text</code> 将表示编码src_encoding的文本的二进制字符串转换为数据库编码中的text。(适用的转换请参阅 Section 21.3.4)。 <code>convert_from('text_in_utf8', 'UTF8') → text_in_utf8</code>
<code>convert_to ( string text, dest_encoding name )</code>	<code>→ bytea</code> 将text字符串(数据库编码)转换为编码dest_encoding中编码的二进制字符串。(适用的转换请参阅 Section 21.3.4)。 <code>convert_to('some_text', 'UTF8') → \x736f6d655f74657874</code>
<code>encode ( bytes bytea, format text )</code>	<code>→ text</code> 将二进制数据编码成文本表示;支持的format值为: base64, escape, hex。 <code>encode('123\000\001', 'base64') → MTIzAAE=</code>
<code>decode ( string text, format text )</code>	<code>→ bytea</code> 从文本表示中解码二进制数据;支持的format值与encode相同。 <code>decode('MTIzAAE=', 'base64') → \x3132330001</code>

encode 和 decode 函数支持以下文本格式:

base64

base64格式是RFC 2045 Section 6.8<sup>1</sup>。根据RFC, 编码的行被分割为76个字符。但是, 作为MIME CRLF行结束标记的替代, 只有换行符用于行结束。decode函数忽略回车、换行、空格和制表符。否则, 当decode被提供了无效的base64数据(包括结尾填充不正确)时。

escape

escape格式将零字节和设置了高位的字节转换为八进制转义序列(\nnn), 并将反斜杠加倍。其他字节值按字面意思表示。decode函数如果反斜杠后面没有第二个反斜杠或三个八进制数字将引发错误;它接受其他未变化的字节值。

hex

hex格式将每4位数据表示为一个十六进制数字, 从0到f, 首先写入每个字节的高阶数字。encode函数输出a-f的十六进制小写数字。因为数据的最小单位是8位, 所以encode总是返回偶数个字符。decode函数接受大写或小写的a-f字符。当decode给出无效的十六进制数据一时将引发一个错误, 包括给定奇数个字符时。

参见Section 9.2中的聚集函数string\_agg以及Section 33.中的大对象函数。

## 9.6. 位串函数和操作符

<sup>1</sup> <https://tools.ietf.org/html/rfc2045#section-6.8>

本节描述用于检查和操作位串的函数和操作符，也就是操作类型为bit和bit varying的值的函数和操作符。（虽然这些表中只提到了bit类型，但bit varying类型的值可以互换使用。）位字符串支持Table 9.14中显示的常用比较操作符，就像Table 9.1中显示的操作符。

Table 9.14. 位串操作符

操作符	描述 例子
bit    bit → bit	连接 B'10001'    B'011' → 10001011
bit & bit → bit	按位与（输入的长度必须相等） B'10001' & B'01101' → 00001
bit   bit → bit	按位或（输入的长度必须相等） B'10001'   B'01101' → 11101
bit # bit → bit	按位异或（输入的长度必须相等） B'10001' # B'01101' → 11100
~ bit → bit	按位求反 ~ B'10001' → 01110
bit << integer → bit	按位左移（字符串长度被保留） B'10001' << 3 → 01000
bit >> integer → bit	按位右移（字符串长度被保留） B'10001' >> 2 → 00100

一些可用于二进制字符串的函数也可用于位字符串，如Table 9.15中所示。

Table 9.15. 位字符串函数

函数	描述 例子
bit_length ( bit ) → integer	返回位字符串中的位数。 bit_length(B'10111') → 5
length ( bit ) → integer	返回位字符串中的位数。 length(B'10111') → 5
octet_length ( bit ) → integer	返回位字符串中的字节数。 octet_length(B'101111011') → 2
overlay ( bits bit PLACING newsubstring bit FROM start integer [ FOR count integer ] ) → bit	替换从start位开始的bits的子字符串，并将newsubstring扩展count位。如果count被省略，默认为newsubstring的长度。 overlay(B'010101010101010' placing B'11111' from 2 for 3) → 01111101010101010
position ( substring bit IN bits bit ) → integer	

函数	描述 例子
	按返回指定substring的起始索引，以bits为单位，如果不存在则返回0。 position(B'010' in B'000001101011') → 8
	substring ( bits bit [ FROM start integer ] [ FOR count integer ] ) → bit 如果指定了起始位，则提取从start位开始的bits的子字符串，如果指定了计数位，则在count位之后停止。 start和count至少提供一个。 substring(B'110010111111' from 3 for 2) → 00
	get_bit ( bits bit, n integer ) → integer 从位字符串中提取第n位；第一个(最左)位为第0位。 get_bit(B'1010101010101010', 6) → 1
	set_bit ( bits bit, n integer, newvalue integer ) → bit 将位字符串中的第n位设置为newvalue；第一个(最左)位是第0位。 set_bit(B'1010101010101010', 6, 0) → 1010100010101010

另外，我们可以在整数和bit之间来回转换。 将一个整数转换为bit(n)会复制最右边的n位。 将一个整数转换为比整数本身更宽的位字符串宽度将在左边进行符号扩展。一些例子：

```
44::bit(10)           0000101100
44::bit(3)           100
cast(-44 as bit(12)) 111111010100
'1110'::bit(4)::integer 14
```

请注意，如果只是转换为“bit”，意思是转换成bit(1)，因此只会转换整数的最低有效位。

## 9.7. 模式匹配

LightDB提供了三种独立的实现模式匹配的方法：SQL LIKE操作符、更近一些的SIMILAR TO操作符（SQL:1999 里添加进来的）和POSIX-风格的正则表达式。除了这些基本的“这个串匹配这个模式吗？”操作符外，还有一些函数可用于提取或替换匹配子串并在匹配位置分离一个串。

### Tip

如果你的模式匹配的要求超出了这些，请考虑用 Perl 或 Tcl 写一个用户定义的函数。

### Caution

虽然大部分的正则表达式搜索都能被很快地执行，但是正则表达式仍可能被人为地弄成需要任意长的时间和任意量的内存进行处理。要当心从不怀好意的来源接受正则表达式搜索模式。如果必须这样做，建议加上语句超时限制。

使用SIMILAR TO模式的搜索具有同样的安全性危险，因为SIMILAR TO提供了很多和 POSIX-风格正则表达式相同的能力。

LIKE搜索比其他两种选项简单得多，因此在使用 不怀好意的模式来源时要更安全些。

这三种类型的模式匹配算子都不支持非确定性拼贴。 如果需要的话，可以在表达式中应用不同的拼贴来绕过这个限制。

## 9.7.1. LIKE

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

如果该string匹配了提供的pattern，那么LIKE表达式返回真（和预期的一样，如果LIKE返回真，那么NOT LIKE表达式返回假，反之亦然。一个等效的表达式是NOT (string LIKE pattern)）。

如果pattern不包含百分号或者下划线，那么该模式只代表它本身的串；这时候LIKE的行为就象等号操作符。在pattern里的下划线（\_）代表（匹配）任何单个字符； 而一个百分号（%）匹配任何零或多个字符的序列。

一些例子：

```
'abc' LIKE 'abc'      true
'abc' LIKE 'a%'      true
'abc' LIKE '_b_'     true
'abc' LIKE 'c'       false
```

LIKE模式匹配总是覆盖整个串。因此，要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。

要匹配文本的下划线或者百分号，而不是匹配其它字符，在pattern里相应的字符必须前导逃逸字符。缺省的逃逸字符是反斜线，但是你可以用ESCAPE子句指定一个不同的逃逸字符。要匹配逃逸字符本身，写两个逃逸字符。

### Note

如果你关掉了standard\_conforming\_strings，你在文串常量中写的任何反斜线都需要被双写。详见Section 4.1.2.1

请注意反斜线在串文本里已经有特殊含义了，所以如果你写一个包含反斜线的模式常量，那你就需要在 SQL 语句里写两个反斜线。因此，写一个匹配单个反斜线的模式实际上要在语句里写四个反斜线。你可以通过用 ESCAPE 选择一个不同的逃逸字符 来避免这样；这样反斜线就不再是 LIKE 的特殊字符了。但仍然是字符文本分析器的特殊字符，所以你还是需要两个反斜线。）我们也可以通过写ESCAPE ''的方式不选择逃逸字符，这样可以有效地禁用逃逸机制，但是没有办法关闭下划线和百分号在模式中的特殊含义。

根据SQL标准，省略ESCAPE意味着没有转义字符（而不是默认为反斜杠），并且不允许使用零长度的ESCAPE值。因此，LightDB在这方面的行为有点不标准。

关键字ILIKE可以用于替换LIKE，它令该匹配根据活动区域成为大小写无关。这个不属于SQL标准而是一个LightDB扩展。

操作符~~等效于LIKE，而~~\*对应ILIKE。还有 !~~和!~~\*操作符分别代表NOT LIKE和NOT ILIKE。所有这些操作符都是LightDB特有的。你可能会在EXPLAIN输出和类似的地方看到这些操作符名称，因为解析器实际上将LIKE等翻译成这些运算符。

短语LIKE, ILIKE, NOT LIKE, 和 NOT ILIKE 在LightDB语法中通常被视为操作符；例如，它们可以用于expression operator的任何（subquery）构造，尽管这里不能包含ESCAPE子句。在某些晦涩难懂的情况下，可能需要用底层操作符名称替代。

还可参见前缀操作符`^@`和相应的`starts_with`函数，在需要简单匹配字符串开头的情况下比较有用。

## 9.7.2. SIMILAR TO正则表达式

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

`SIMILAR TO`操作符根据自己的模式是否匹配给定串而返回真或者假。它和`LIKE`非常类似，只不过它使用 SQL 标准定义的正则表达式理解模式。SQL 正则表达式是在`LIKE`标记和普通的 (POSIX) 正则表达式标记的奇怪的杂交。

类似`LIKE`，`SIMILAR TO`操作符只有在它的模式匹配整个串的时候才能成功；这一点和普通的正则表达式的行为不同，在普通的正则表达式里，模式匹配串的任意部分。和`LIKE`类似的地方还有，`SIMILAR TO`使用`_`和`%`作为分别代表任意单个字符和任意串的通配符（这些可以比得上 POSIX 正则表达式里的`.`和`*`）。

除了这些从`LIKE`借用的功能之外，`SIMILAR TO`支持下面这些从 POSIX 正则表达式借用的 模式匹配元字符：

- `|`表示选择（两个候选之一）。
- `*`表示重复前面的项零次或更多次。
- `+`表示重复前面的项一次或更多次。
- `?`表示重复前面的项零次或一次。
- `{m}`表示重复前面的项刚好`m`次。
- `{m,}`表示重复前面的项`m`次或更多次。
- `{m, n}`表示重复前面的项至少`m`次并且不超过`n`次。
- 可以使用圆括号`()`把多个项组合成一个逻辑项。
- 一个方括号表达式`[...]`声明一个字符类，就像 POSIX 正则表达式一样。

注意点号`.`不是`SIMILAR TO`的一个元字符。

与`LIKE`一样，反斜杠将禁用这些元字符的特殊含义。可以用 `ESCAPE` 来指定不同的转义字符，或者可以通过写 `ESCAPE ''` 来禁用转义功能。

根据SQL标准，省略`ESCAPE`意味着没有转义字符（而不是默认为反斜杠），并且不允许使用零长度的`ESCAPE`值。LightDB在这方面的行为有点不标准。

另一个非标准扩展是，在转义字符后面跟着一个字母或数字提供了对为POSIX正则表达式定义的转义序列的访问；参见下面的 Table 9.20 Table 9.21 和 Table 9.22。

一些例子：

```
'abc' SIMILAR TO 'abc'           true
'abc' SIMILAR TO 'a'             false
'abc' SIMILAR TO '%(b|d)%'      true
'abc' SIMILAR TO '(b|c)%'       false
'-abc-' SIMILAR TO '%\mabc\M%'  true
'xabcy' SIMILAR TO '%\mabc\M%'  false
```

带有三个参数的`substring`函数可以提取匹配SQL正则表达式模式的子字符串。该函数可以按照SQL99语法编写。

substring(string from pattern for escape-character)

或作为一个普通的三参数函数:

substring(string, pattern, escape-character)

与SIMILAR TO一样, 指定的模式必须与整个数据字符串匹配, 否则函数失败并返回空值。 为了表示匹配的数据子字符串的模式中, 模式中应该包含两个转义字符的出现, 并在后面加上一个双引号(")。 匹配成功后, 将返回与这些分隔符之间的模式部分匹配的文本。

转义-双引号分隔符实际上是 将子字符串的模式分成三个独立的 正则表达式; 例如, 竖条(|) 三节中的任何一节只影响到该节。 此外, 第一节和第三种正则表达式的定义是为了匹配最小的 尽可能多的文字, 而不是最大的文字, 当有歧义的时候, 就不应该是最大的文字。 关于有多少数据字符串符合哪种模式。 (在POSIX术语中, 第一和第三种正则表达式被强行规定为非贪婪)。

作为对SQL标准的扩展, LightDB只允许有一个转义双引号分隔符, 在这种情况下, 第三个正则表达式被视为空; 或者没有分隔符, 在这种情况下, 第一个和第三个正则表达式被视为空。

一些例子, 使用#"定界返回串:

```
substring('foobar' from '%"o_b#"' for '#') oob
substring('foobar' from #'"o_b#"' for '#') NULL
```

### 9.7.3. POSIX正则表达式

Table 9.16列出了所有可用于 POSIX 正则表达式模式匹配的操作符。

Table 9.16. 正则表达式匹配操作符

操作符	描述 例子
text ~ text → boolean	字符串匹配正则表达式, 大小写敏感 'thomas' ~ 't.*ma' → t
text ~* text → boolean	字符串匹配正则表达式, 大小写不敏感 'thomas' ~* 'T.*ma' → t
text !~ text → boolean	字符串不匹配正则表达式, 大小写敏感 'thomas' !~ 't.*max' → t
text !~* text → boolean	字符串不匹配正则表达式, 大小写不敏感 'thomas' !~* 'T.*ma' → f

POSIX正则表达式提供了比LIKE和SIMILAR TO操作符更强大的含义。许多 Unix 工具, 例如grep、sed或awk使用一种与我们这里描述的类似的模式匹配语言。

正则表达式是一个字符序列, 它是定义一个串集合 (一个正则集) 的缩写。 如果一个串是正则表达式描述的正则集中的一员时, 我们就说这个串匹配该正则表达式。 和LIKE一样, 模式字符准确地匹配串字符, 除非在正则表达式语言里有特殊字符 — 不过正则表达式用的特殊字符和LIKE用的不同。 和LIKE模式不一样的是, 正则表达式允许匹配串里的任何位置, 除非该正则表达式显式地挂接在串的开头或者结尾。

一些例子:

```
'abcd' ~ 'bc'      true
'abcd' ~ 'a.c'     true — 点号 (.) 匹配任意字符
'abcd' ~ 'a.*d'    true — * 重复前面的模式项
'abcd' ~ '(b|x)'   true — | 表示或 (OR), 括号用于分组
'abcd' ~ '^a'      true — ^ 锚定到字符串的开头
'abcd' ~ '^(b|c)' false — 除了锚定外, 将匹配
```

POSIX模式语言的详细描述见下文。

带两个参数的substring函数, 即substring(string from pattern), 提供了抽取一个匹配POSIX 正则表达式模式的子串的方法。如果没有匹配它返回空值, 否则就是文本中匹配模式的那部分。但是如果该模式包含任何圆括号, 那么将返回匹配第一对子表达式 (对应第一个左圆括号的) 的文本。如果你想在表达式里使用圆括号而又不想导致这个例外, 那么你可以在整个表达式外边放上一对圆括号。如果你需要在想抽取的子表达式前有圆括号, 参阅后文描述的非捕获性圆括号。

一些例子:

```
substring('foobar' from 'o.b')    oob
substring('foobar' from 'o(.)b')  o
```

regexp\_replace函数提供了将匹配 POSIX 正则表达式模式的子串替换为新文本的功能。它的语法是 regexp\_replace(source, pattern, replacement [, flags ])。如果没有匹配pattern, 那么返回不加修改的source串。如果有匹配, 则返回的source串里面的匹配子串将被replacement串替换掉。replacement串可以包含\n, 其中\n是 1 到 9, 表明源串里匹配模式里第n个圆括号子表达式的子串应该被插入, 并且它可以包含&表示应该插入匹配整个模式的子串。如果你需要放一个文字形式的反斜线在替换文本里, 那么写\\。flags参数是一个可选的文本串, 它包含另个或更多单字母标志, 这些标志可以改变函数的行为。标志i指定大小写无关的匹配, 而标志g指定替换每一个匹配的子串而不仅仅是第一个。支持的标志 (但不是g) 在Table 9.2中描述。

一些例子:

```
regexp_replace('foobarbaz', 'b..', 'X')
      fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
      fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
      fooXarYXazY
```

regexp\_match返回一个文本数组, 它包含一个POSIX正则表达式模式与一个字符串第一个匹配所得到的子串。其语法是regexp\_match(string, pattern [, flags ])。如果没有匹配, 则结果为NULL。如果找到一个匹配并且pattern不包含带括号的子表达式, 那么结果是一个单一元素的文本数组, 其中包含匹配整个模式的子串。如果找到一个匹配并且pattern含有带括号的子表达式, 那么结果是一个文本数组, 其中第n个元素是与pattern的第n个圆括号子表达式匹配的子串 (“非捕获” 圆括号不计入在内, 详见下文)。flags参数是一个可选的文本字符串, 它包含零个或者更多个可以改变该函数行为的单字母标志。所支持的标志在Table 9.2中介绍。

一些例子:

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
       regexp_match
-----
{barbeque}
```



```
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
   regexp_match
-----
 {bar, beque}
(1 row)
```

在通常情况下，人们只是想要的大整个匹配的子串或者NULL（没有匹配），可以写成这样

```
SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
   regexp_match
-----
 barbeque
(1 row)
```

`regexp_matches`函数返回一个文本数组的集合，其中包含着一个POSIX正则表达式模式与一个字符串匹配得到的子串。它和`regexp_match`具有相同的语法。如果没有匹配，这个函数不会返回行。如果有一个匹配并且给定了`g`标志，则返回一行。如果有`N`个匹配并且给定了`g`标志，则返回`N`行。每一个返回的行都是一个文本数组，其中含有整个匹配的子串或者匹配`pattern`的圆括号子表达式的子串，这和上面对`regexp_match`的介绍一样。`regexp_matches`接受Table 9.2中展示的所有标志，外加令它返回所有匹配而不仅仅是第一个匹配的`g`标志。

一些例子：

```
SELECT regexp_matches('foo', 'not there');
   regexp_matches
-----
(0 rows)
```

```
SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
   regexp_matches
-----
 {bar, beque}
 {bazil, barf}
(2 rows)
```

### Tip

在大部分情况下，`regexp_matches()`应该与`g`标志一起使用，因为如果只是想要第一个匹配，使用`regexp_match()`会更加简单高效。不过，`regexp_match()`仅存在于LightDB版本10以及更高的版本中。当在较老的版本中使用`regexp_matches()`时，一种常用的技巧是把`regexp_matches()`调用放在子选择中，例如：

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

如果有一个匹配，则这个语句会产生一个文本数组，否则返回NULL，这和`regexp_match()`的做法一样。如果没有子选择，这个查询对于没有匹配的表行根本不会产生输出，显然那不是想要的行为。

`regexp_split_to_table`把一个 POSIX 正则表达式模式当作一个定界符来分离一个串。它的语法形式是`regexp_split_to_table(string, pattern [, flags ])`。如果没有与`pattern`的匹配，该函数返回`string`。如果有至少有一个匹配，对每

一个匹配它都返回从上一个匹配的末尾（或者串的开头）到这次匹配开头之间的文本。当没有更多匹配时，它返回从上一次匹配的末尾到串末尾之间的文本。flags参数是一个可选的文本串，它包含零个或更多单字母标志，这些标识可以改变该函数的行为。regexp\_split\_to\_table能支持的标志在Table 9.2中描述。

regexp\_split\_to\_array函数的行为和regexp\_split\_to\_table相同，不过regexp\_split\_to\_array会把它的结果以一个text数组的形式返回。它的语法是regexp\_split\_to\_array(string, pattern [, flags ])。这些参数和regexp\_split\_to\_table的相同。

一些例子：

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog', '\s+') AS foo;
```

```
foo
-----
the
quick
brown
fox
jumps
over
the
lazy
dog
(9 rows)
```

```
SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+');
```

```
regexp_split_to_array
-----
{the, quick, brown, fox, jumps, over, the, lazy, dog}
(1 row)
```

```
SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo;
```

```
foo
-----
t
h
e
q
u
i
c
k
b
r
o
w
n
f
o
x
(16 rows)
```

正如上一个例子所示，正则表达式分离函数会忽略零长度的匹配，这种匹配发生在串的开头或结尾或者正好发生在前一个匹配之后。这和正则表达式匹配的严格定义是相悖的，后者

由`regexp_match`和`regexp_matches`实现，但是通常前者是实际中最常用的行为。其他软件系统如Perl也使用相似的定义。

### 9.7.3.1. 正则表达式细节

LightDB的正则表达式是使用 Henry Spencer 写的一个包来实现的。下面的正则表达式的大部分描述都是从他的手册页中逐字拷贝过来的。

正则表达式 (RE)，在POSIX 1003.2 中定义，它有两种形式：扩展的RE或者是ERE（大概地说就是那些在`egrep`里的），基本的RE或者是BRE（大概地说就是那些在`ed`里的）。LightDB支持两种形式，并且还实现了一些POSIX标准中没有但是在类似 Perl 或者 Tcl 这样的语言中得到广泛应用的一些扩展。使用了那些非POSIX扩展的RE叫高级RE，或者本文档里说的ARE。ARE 几乎完全是 ERE 的超集，但是 BRE 有几个符号上的不兼容（以及更多的限制）。我们首先描述 ARE 和 ERE 形式，描述那些只适用于 ARE 的特性，然后描述 BRE 的区别是什么。

#### Note

LightDB初始时总是推测一个正则表达式遵循 ARE 规则。但是，可以通过为 RE 模式预置一个`embedded option`来选择限制更多的 ERE 或 BRE 规则，如Section 9.7.3.1中所述。这对为期望准确的POSIX 1003.2 规则的应用提供兼容性很有用。

一个正则表达式被定义为一个或多个分支，它们之间被`|`分隔。只要能匹配其中一个分支的东西都能匹配正则表达式。

一个分支是一个或多个量化原子或者约束连接而成。一个原子匹配第一个，然后后面的原子匹配第二个，以此类推；一个空分支匹配空串。

一个量化原子是一个原子，后面可能跟着一个量词。没有量词的时候，它匹配一个原子，有量词的时候，它可以匹配若干个原子。一个原子可以在Table 9.17里面显示的任何可能。可能的量词和它们的含义在Table 9.18显示。

一个约束匹配一个空串，但只是在满足特定条件下才匹配。约束可以在能够使用原子的地方使用，只是它不能跟着量词。简单的约束在Table 9.19显示；更多的约束稍后描述。

Table 9.17. 正则表达式原子

原子	描述
<code>(re)</code>	（其中 <code>re</code> 是任何正则表达式） 匹配一个对 <code>re</code> 的匹配，匹配将为可能的报告被记下
<code>(?:re)</code>	同上，但是匹配不会为了报告而被记下（一个“非捕获”圆括号集）（只对 ARE）
<code>.</code>	匹配任意单个字符
<code>[chars]</code>	一个方括号表达式，匹配 <code>chars</code> 中的任意一个（详见Section 9.7.3.2）
<code>\k</code>	（其中 <code>k</code> 是一个非字母数字字符） 匹配一个被当作普通字符看待的特定字符，例如， <code>\</code> 匹配一个反斜线字符
<code>\c</code>	其中 <code>c</code> 是一个字母数字（可能跟着其它字符），它是一个逃逸，参阅Section 9.7.3.3（仅对 ARE；在 ERE 和 BRE 中，它匹配 <code>c</code> ）

原子	描述
{	如果后面跟着一个字符，而不是数字，那么就匹配左花括弧{；如果跟着一个数字，那么它是range的开始（见下文）
x	其中x是一个没有其它意义的单个字符，则匹配该字符

RE 不能以反斜线 (\) 结尾。

### Note

如果你关掉了`standard_conforming_strings`，任何你写在文字串常量中的反斜线都需要被双写。详见Section 4.1.2.1

Table 9.18. 正则表达式量词

量词	匹配
*	一个由原子的 0 次或更多次匹配组成的序列
+	一个由原子的 1 次或更多次匹配组成的序列
?	一个由原子的 0 次或 1 次匹配组成的序列
{m}	一个由原子的正好m次匹配组成的序列
{m, }	一个由原子的m次或更多次匹配组成的序列
{m, n}	一个由原子的从m次到n次（包括）匹配组成的序列；m不能超过n
*?	*的非贪婪版本
+?	+的非贪婪版本
??	?的非贪婪版本
{m}?	{m}的非贪婪版本
{m, }?	{m, }的非贪婪版本
{m, n}?	{m, n}的非贪婪版本

使用{...}的形式被称作范围。一个范围内的数字m和n都是无符号十进制整数，允许的数值从 0 到 255（包含）。

非贪婪的量词（只在 ARE 中可用）匹配对应的正常（贪婪）模式，区别是它寻找最少的匹配，而不是最多的匹配。详见Section 9.7.3.5

### Note

一个量词不能紧跟在另外一个量词后面，例如\*\*是非法的。量词不能作为表达式或者子表达式的开头，也不能跟在^或者|后面。

Table 9.19. 正则表达式约束

约束	描述
^	串开头的匹配
\$	串末尾的匹配

约束	描述
(?=re)	在匹配re的子串开始的任何点的positive lookahead匹配（只对 ARE）
(?!re)	在匹配re的子串开始的任何点的negative lookahead匹配（只对 ARE）
(?<=re)	只要有一个点上有一个子串匹配re端，positive lookbehind就在这个点上匹配（只对 ARE）
(?<!re)	只要有一个点上没有子串匹配re端，negative lookbehind就在这个点上匹配（只对 ARE）

Lookahead 和 lookbehind 约束不能包含后引用（参阅Section 9.7.3.3，并且其中的所有圆括号都被认为是非捕获的。

### 9.7.3.2. 方括号表达式

方括号表达式是一个包围在[]中的字符列表。它通常匹配列表中的任意单个字符（但见下文）。如果列表以^开头，它匹配任意单个不在该列表参与部分中的字符。如果该列表中两个字符用-隔开，那它就是那两个字符（包括在内）之间的所有字符范围的缩写，例如，在ASCII中[0-9]匹配任何十进制数字。两个范围共享一个端点是非法的，例如，a-c-e。范围与字符集关系密切，可移植的程序应该避免依靠它们。

想在列表中包含文本]，可以让它做列表的首字符（如果使用了^，需要放在其后）。想在列表中包含文本-，可以让它做列表的首字符或者尾字符，或者一个范围的第二个端点。想在列表把文本-当做范围的起点，把它用[.和.]包围起来，这样它就成为一个排序元素（见下文）。除了这些字符本身、一些用[的组合（见下段）以及逃逸（只在 ARE 中有效）以外，所有其它特殊字符在方括号表达式里都失去它们的特殊含义。特别是，在 ERE 和 BRE 规则下\不是特殊的，但在 ARE 里，它是特殊的（引入一个逃逸）。

在一个方括号表达式里，一个排序元素（一个字符、一个被当做一个单一字符排序的多字符序列或者一个表示上面两种情况的排序序列名称）包含在[.和.]里面的时候表示该排序元素的字符序列。该序列被当做该方括号列表的一个单一元素。这允许一个包含多字符排序元素的方括号表达式去匹配多于一个字符，例如，如果排序序列包含一个ch排序元素，那么 RE [[.ch.]]\*c匹配chchcc的头五个字符。

#### Note

LightDB当前不支持多字符排序元素。这些信息描述了将来可能有的行为。

在方括号表达式里，包围在[=和=]里的排序元素是一个等价类，代表等效于那一个的所有排序元素的字符序列，包括它本身（如果没有其它等效排序元素，那么就好象封装定界符是[.和.]）。例如，如果o和^是一个等价类的成员，那么[[=o=]]、[[=^=]]和[o^]都是同义的。一个等价类不能是一个范围的端点。

在方括号表达式里，在[:和:]里面封装的字符类的名字代表属于该类的所有字符的列表。字符类不能作为范围的端点使用。POSIX标准定义了这些字符类的名称：alnum（字符和数字），alpha（字符），blank（空格和制表符tab），cntrl（控制符），digit（位数），graph（空格除外可打印字符），lower（小写字母），print（包含空格可打印字符），punct（标点符号），space（空白），upper（大写字母），和xdigit（十六进制数）。对于7位ASCII字符集中的字符来说，这些标准字符类的行为在不同平台上一致。一个给定的非ASCII字符是否被认为属于这些类别中的一个，取决于正则表达式函数或运算符使用的collation（见Section 21.2，或者默认情况下取决于数据库的LC\_CTYPE locale设置（见Section 21.1）。非ASCII字符的分类在不同的平台上会有不同的分类，即使是在类似命名的locale中也是如此。（但C locale从不认为任何非ASCII字符属于上述任何

一类)。除了这些标准字符类之外，LightDB定义了ascii字符类，它完全包含7位ASCII字符集。

方括号表达式里有两个特例：方括号表达式`[:<:]`和`[:>:]`是约束，分别匹配一个单词开头和结束的空串。单词定义为一个单词字符序列，前面和后面都没有其它单词字符。单词字符是一个alnum字符（和如上所述POSIX字符类中定义的一样）或者一个下划线。这是一个扩展，兼容POSIX 1003.2，但那里并没有说明，而且在准备移植到其他系统里去的软件里一定要小心使用。通常下文描述的约束逃逸更好些（它们并非更标准，但是更容易键入）。

### 9.7.3.3. 正则表达式逃逸

逃逸是以`\`开头，后面跟着一个字母数字字符得特殊序列。逃逸有好几种变体：字符项、类缩写、约束逃逸以及后引用。在ARE里，如果一个`\`后面跟着一个字母数字，但是并未组成一个合法的逃逸，那么它是非法的。在ERE中没有逃逸：在方括号表达式之外，一个后面跟着字母数字字符的`\`只是表示该字符是一个普通的字符，而且在一个方括号表达式里，`\`是一个普通的字符（后者实际上在ERE和ARE不兼容）。

字符项逃逸用于便于我们在RE中声明那些不可打印的或其他习惯的字符。它们显示在Table 9.20中。

类缩写逃逸用来提供一些常用的字符类缩写。它们显示在Table 9.2中。

约束逃逸是一个约束，如果满足特定的条件，它匹配该空串。它们显示在Table 9.2中。

后引用`(\n)`匹配数字`n`指定的被前面的圆括号子表达式匹配的同个串（参阅Table 9.23）。例如，`([bc])\1`匹配`bb`或者`cc`，但是不匹配`bc`或者`cb`。RE中子表达式必须完全在后引用前面。子表达式以它们的前导圆括号的顺序编号。非捕获圆括号并不定义子表达式。

Table 9.20. 正则表达式字符项逃逸

逃逸	描述
<code>\a</code>	警告（响铃）字符，和C中一样
<code>\b</code>	退格，和C中一样
<code>\B</code>	反斜线（ <code>\</code> ）的同义词，用来减少双写反斜线
<code>\cX</code>	（其中X是任意字符）低序5位和X相同的字符，它的其他位都是零
<code>\e</code>	排序序列名为ESC的字符，如果无法做到该字符为八进制值033
<code>\f</code>	换页，和C中一样
<code>\n</code>	新行，和C中一样
<code>\r</code>	回车，和C中一样
<code>\t</code>	水平制表符，和C中一样
<code>\uwxyz</code>	（其中wxyz正好是四个十六进制位）十六进制值为0xwxyz的字符
<code>\Ustuvwxyz</code>	（其中stuvwxyz正好是八个十六进制位）十六进制值为0xstuvwxyz的字符
<code>\v</code>	垂直制表符，和C中一样
<code>\xhhh</code>	（其中hhh是十六进制位的任意序列）十六进制值为0xhhh的字符（一个单一字符，不管用了多少个十六进制位）

逃逸	描述
<code>\0</code>	值为0（空字节）的字符
<code>\xy</code>	（其中xy正好是两个八进制位，并且不是一个后引用）八进制值为0xy的字符
<code>\xyz</code>	（其中xyz正好是三个八进制位，并且不是一个后引用）八进制值为0xyz的字符

十六进制位是0-9、a-f和A-F。八进制位是0-7。

指定 ASCII 范围（0 - 127）之外的值的数字字符项转义的含义取决于数据库编码。当编码是 UTF-8 时，转义值等价于 Unicode 代码点，例如 `\u1234`表示字符U+1234。对于其他多字节编码，字符项转义通常只是指定该字符的字节值的串接。如果该转义值不对应数据库编码中的任何合法字符，将不会发生错误，但是它不会匹配任何数据。

字符项逃逸总是被当作普通字符。例如，`\135`是 ASCII 中的`]`，但`\135`并不终止一个方括号表达式。

Table 9.21. 正则表达式类缩写逃逸

逃逸	描述
<code>\d</code>	<code>[[[:digit:]]</code>
<code>\s</code>	<code>[[[:space:]]</code>
<code>\w</code>	<code>[[[:alnum:]]_]</code> （注意下划线是被包括的）
<code>\D</code>	<code>[^[:digit:]]</code>
<code>\S</code>	<code>[^[:space:]]</code>
<code>\W</code>	<code>[^[:alnum:]]_]</code> （注意下划线是被包括的）

在方括号表达式里，`\d`、`\s`和`\w`会失去它们的外层方括号，而`\D`、`\S`和`\W`是非法的（也就是说，例如`[a-c\d]`等效于`[a-c[:digit:]]`。同样`[a-c\D]`等效于`[a-c^[:digit:]]`的，也是非法的）。

Table 9.22. 正则表达式约束逃逸

逃逸	描述
<code>\A</code>	只在串开头匹配（与 <code>^</code> 的不同请参见Section 9.7.3.5）
<code>\m</code>	只在一个词的开头匹配
<code>\M</code>	只在一个词的末尾匹配
<code>\y</code>	只在一个词的开头或末尾匹配
<code>\Y</code>	只在一个词的不是开头或末尾的点上匹配
<code>\Z</code>	只在串的末尾匹配（与 <code>\$</code> 的不同请参见Section 9.7.3.5）

一个词被定义成在上面`[[[:<:]]`和`[[[:>:]]`中的声明。在方括号表达式里，约束逃逸是非法的。

Table 9.23. 正则表达式后引用

逃逸	描述
<code>\m</code>	（其中m是一个非零位）一个到第m个子表达式的后引用

逃逸	描述
<code>\mnn</code>	(其中m是一个非零位, 并且nn是一些更多的位, 并且十六进制值mnn不超过目前能看到的封闭捕获圆括号的数目) 一个到第mnn个子表达式的后引用

### Note

在八进制字符项逃逸和后引用之间有一个历史继承的歧义存在, 这个歧义是通过下面的启发式规则解决的, 像上面描述地那样。前导零总是表示这是一个八进制逃逸。而单个非零数字, 如果没有跟着任何其它位, 那么总是被认为是后引用。一个多位的非零开头的序列也被认为是后引用, 只要它出现在合适的子表达式后面 (也就是说, 在后引用的合法范围中的数), 否则就被认为是一个八进制。

## 9.7.3.4. 正则表达式元语法

除了上面描述的主要语法之外, 还有几种特殊形式和杂项语法。

如果一个 RE 以`***:`开头, 那么剩下的 RE 都被当作 ARE (这在LightDB中通常是无效的, 因为 RE 被假定为 ARE, 但是如果 ERE 或 BRE 模式通过`flags`参数被指定为一个正则表达式函数时, 它确实能产生效果)。如果一个 RE 以`***=`开头, 那么剩下的 RE 被当作一个文本串, 所有的字符都被认为是一个普通字符。

一个 ARE 可以以嵌入选项开头: 一个序列(`?xyz`) (这里的xyz是一个或多个字母字符) 声明影响剩余的 RE 的选项。这些选项覆盖任何前面判断的选项 — 特别地, 它们可以覆盖一个正则表达式操作符隐含的大小写敏感的行为, 或者覆盖`flags`参数中的正则表达式函数。可用的选项字母在Table 9.24中显示。注意这些同样的选项字母也被用在正则表达式函数的`flags`参数中。

Table 9.24. ARE 嵌入选项字母

选项	描述
b	RE的剩余部分是一个BRE
c	大小写敏感的匹配 (覆盖操作符类型)
e	RE的剩余部分是一个ERE
i	大小写不敏感的匹配 (见Section 9.7.3.5 (覆盖操作符类型))
m	n的历史原因的同义词
n	新行敏感的匹配 (见Section 9.7.3.5)
p	部分新行敏感的匹配 (见Section 9.7.3.5)
q	RE的剩余部分是一个文字 (“quoted”) 串, 全部是普通字符
s	非新行敏感的匹配 (默认)
t	紧语法 (默认, 见下文)
w	逆部分新行敏感 (“怪异”) 的匹配 (见Section 9.7.3.5)
x	扩展语法 (见下文)

嵌入选项在) 终止序列时发生作用。它们只在 ARE 的开始处起作用 (在任何可能存在的`***:`控制器后面)。



除了通常的（紧）RE 语法（这种情况下所有字符都有效），还有一种扩展语法，可以通过声明嵌入的x选项获得。在扩展语法里，RE 中的空白字符被忽略，就像那些在#和其后的新行（或 RE 的末尾）之间的字符一样。这样就允许我们给一个复杂的 RE 分段和注释。不过这个基本规则有三种例外：

- 空白字符或前置了\的#将被保留
- 方括号表达式里的空白或者#将被保留
- 在多字符符号里面不能出现空白和注释，例如(?:

为了这个目的，空白是空格、制表符、新行和任何属于空白字符类的字符。

最后，在 ARE 里，方括号表达式外面，序列(?#ttt)（其中ttt是任意不包含一个)的文本)是一个注释，它被完全忽略。同样，这样的东西是不允许出现在多字符符号的字符中间的，例如(?:。这种注释更像是一种历史产物而不是一种有用的设施，并且它们的使用已经被废弃；请使用扩展语法来替代。

如果声明了一个初始的\*\*\*=控制器，那么所有这些元语法扩展都不能使用，因为这样表示把用户输入当作一个文字串而不是 RE 对待。

### 9.7.3.5. 正则表达式匹配规则

在 RE 可以在给定串中匹配多于一个子串的情况下，RE 匹配串中最靠前的那个子串。如果 RE 可以匹配在那个位置开始的多个子串，要么是取最长的子串，要么是最短的，具体哪种，取决于 RE 是贪婪的还是非贪婪的。

一个 RE 是否贪婪取决于下面规则：

- 大多数原子以及所有约束，都没有贪婪属性（因为它们毕竟无法匹配个数变化的文本）。
- 在一个 RE 周围加上圆括号并不会改变其贪婪性。
- 带一个固定重复次数量词（{m}或者{m}?）的量化原子和原子自身具有同样的贪婪性（可能是没有）。
- 一个带其他普通的量词（包括{m,n}中m等于n的情况）的量化原子是贪婪的（首选最长匹配）。
- 一个带非贪婪量词（包括{m,n}?中m等于n的情况）的量化原子是非贪婪的（首选最短匹配）。
- 一个分支 — 也就是说，一个没有顶级|操作符的 RE — 和它里面的第一个有贪婪属性的量化原子有着同样的贪婪性。
- 一个由|操作符连接起来的两个或者更多分支组成的 RE 总是贪婪的。

上面的规则所描述的贪婪属性不仅仅适用于独立的量化原子，而且也适用于包含量化原子的分支和整个 RE。这里的意思是，匹配是按照分支或者整个 RE 作为一个整体匹配最长或者最短的可能子串。一旦整个匹配的长度确定，那么匹配任意特定子表达式的部分就基于该子表达式的贪婪属性进行判断，在 RE 里面靠前的子表达式的优先级高于靠后的子表达式。

一个相应的例子：

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
结果: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
```

结果: 1

在第一个例子里, RE `Y*` 作为整体是贪婪的, 因为 `Y*` 是贪婪的。它可以匹配从 `Y` 开始的东西, 并且它匹配从这个位置开始的最长的串, 也就是, `Y123`。输出是这里的圆括号包围的部分, 或者说是 `123`。在第二个例子里, RE `Y*?` 总体上是一个非贪婪的 RE, 因为 `Y*?` 是非贪婪的。它可以匹配从 `Y` 开始的最短的子串, 也就是说 `Y1`。子表达式 `[0-9]{1,3}` 是贪婪的, 但是它不能修改总体匹配长度的决定; 因此它被迫只匹配 `1`。

简而言之, 如果一个 RE 同时包含贪婪和非贪婪的子表达式, 那么总的匹配长度要么是尽可能长, 要么是尽可能短, 这取决于给整个 RE 赋予的属性。给子表达式赋予的属性只影响在这个匹配里, 各个子表达式之间相互允许“吃掉”的多少。

量词 `{1,1}` 和 `{1,1}?` 可以分别用于在一个子表达式 或者整个 RE 上强制贪婪或者非贪婪。当需要整个 RE 具有不同于从其元素中 推导出的贪婪属性时, 这很有用。例如, 假设我们尝试将一个包含一些数字的 字符串分隔成数字以及在它们之前和之后的部分, 我们可能会尝试这样做:

```
SELECT regexp_match(' abc01234xyz', '(.*) (\d+) (.*)');
Result: {abc0123, 4, xyz}
```

这不会有用了: 第一个 `.*` 是贪婪的, 因此它会“吃掉”尽可能多的字符而留下 `\d+` 去匹配在最后一个可能位置上的最后一个数字。我们可能会通过让它变成非贪婪来修复:

```
SELECT regexp_match(' abc01234xyz', '(.*?) (\d+) (.*)');
Result: {abc, 0, ""}
```

这也不会有用: 因为现在 RE 作为整体来说是非贪婪的, 因此它会尽快结束全部的匹配。我们可以通过强制 RE 整体是贪婪的来得到我们想要的:

```
SELECT regexp_match(' abc01234xyz', '(?:.*) (\d+) (.*) {1,1}');
Result: {abc, 01234, xyz}
```

独立于 RE 的组件的贪婪性之外控制 RE 的整体贪婪性为处理变长模式提供了很大的灵活性。

在决定更长或者更短的匹配时, 匹配长度是以字符衡量的, 而不是排序元素。一个空串会被认为比什么都不匹配长。例如: `bb*` 匹配 `abbbbc` 的中间三个字符; `(week|wee)(night|knights)` 匹配 `weeknights` 的所有十个字符; 而 `(.*)*` 匹配 `abc` 的时候, 圆括号包围的子表达式匹配所有三个字符; 当 `(a*)*` 被拿来匹配 `bc` 时, 整个 RE 和圆括号子表达式都匹配一个空串。

如果声明了大小写无关的匹配, 那么效果就好像所有大小写区别在字母表中消失了。如果在多个情况中一个字母以一个普通字符的形式出现在方括号表达式外面, 那么它实际上被转换成一个包含大小写的方括号表达式, 也就是说, `x` 变成 `[xX]`。如果它出现在一个方括号表达式里面, 那么它的所有大小写的同族都被加入方括号表达式中, 也就是说, `x` 变成 `[xX]`。当它出现在一个方括号表达式内时, 它的所有大小写副本都被加入到方括号表达式中, 例如, `[x]` 会变成 `[xX]`, 而 `[^x]` 会变成 `[^xX]`。

如果指定了新行敏感的匹配, `.` 和使用 `^` 的方括号表达式 将永远不会匹配新行字符 (这样, 匹配就绝对不会跨越新行, 除非 RE 显式地安排了这样的情况) 并且 `^` 和 `$` 除了分别匹配串开头和结尾之外, 还将分别匹配新行后面和前面的空串。但是 ARE 逃逸 `\A` 和 `\Z` 仍然只匹配串的开始和结尾。

如果指定了部分新行敏感的匹配, 那么它影响 `.` 和方括号表达式, 这个时候和新行敏感的匹配一样, 但是不影响 `^` 和 `$`。

如果指定了逆新行敏感匹配, 那么它影响 `^` 和 `$`, 其作用和在新行敏感的匹配里一样, 但是不影响 `.` 和方括号表达式。这个并不是很有用, 只是为了满足对称性而提供的。

### 9.7.3.6. 限制和兼容性

在这个实现里，对 RE 的长度没有特别的限制。但是，那些希望高移植性的程序应该避免使用长度超过 256 字节的 RE，因为 POSIX 兼容的实现可以拒绝接受这样的 RE。

ARE 实际上和 POSIX ERE 不兼容的唯一的特性是在方括号表达式里\并不失去它特殊的含义。所有其它 ARE 特性都使用在 POSIX ERE 里面是非法或者是未定义、未声明效果的语法；指示器的\*\*\*就是在 POSIX 的 BRE 和 ERE 之外的语法。

许多 ARE 扩展都是从 Perl 那里借来的（但是有些被做了修改来清理它们），以及一些 Perl 里没有出现的扩展。要注意的不兼容性包括\b、\B、对结尾的新行缺乏特别的处理、对那些被新行敏感匹配的东西附加的补齐方括号表达式、在 lookahead/lookbehind 约束里对圆括号和后引用的限制以及最长/最短 匹配（而不是第一匹配）的语义。

### 9.7.3.7. 基本正则表达式

BREs 在几个方面和 ERE 不太一样。在 BRE 中，|、+和?都是普通字符并且没有与它们功能等价的东西。范围的定界符是\{和\}，因为 {和} 本身是普通字符。嵌套的子表达式的圆括号是\（和\)，因为（和）自身是普通字符。除非在 RE 开头或者是圆括号子表达式开头，^都是一个普通字符。除非在 RE 结尾或者是圆括号子表达式的结尾，\$是一个普通字符。如果\*出现在 RE 开头或者是圆括号封装的子表达式开头（前面可能有^），那么它是个普通字符。最后，可以用单数字的后引用，\<和\>分别是[:<:] and [:>:]的同义词；在 BRE 中没有其它可用的逃逸。

### 9.7.3.8. 与XQuery的区别 (LIKE\_REGEX)

从SQL:2008开始，SQL标准中包含了一个LIKE\_REGEX操作符，它根据XQuery正则表达式标准执行模式匹配。LightDB还没有实现这个操作符，但是你可以使用regexp\_match()函数获得非常类似的行为，因为XQuery正则表达式非常接近于上面描述的ARE语法。

与现有的基于POSIX的正则表达式功能和XQuery正则表达式包括。

- 不支持XQuery字符类减法。这个功能的一个例子是使用下面的例子，只匹配英文辅音。`[a-z-[aeiou]]`。
- XQuery字符类速记c。不支持C、i和I。
- 不支持使用p{UnicodeProperty}或反过来的P{UnicodeProperty}的XQuery字符类元素。
- POSIX根据当前的locale来解释字符类，如/w（见Table 9.2）（你可以通过在操作符或函数中附加一个COLLATE子句来控制）。XQuery通过引用Unicode字符属性来指定这些类，因此只有遵循Unicode规则的locale才能获得等效的行为。
- SQL标准（而不是XQuery本身）试图满足更多的需求。“newline的变体”比POSIX的变体。上面描述的对新行敏感的匹配选项只考虑ASCII NL (n)是新行，但SQL会让我们把CR (r)、CRLF (r/n) (Windows风格的新行)，以及一些Unicode唯一的字符，如LINE SEPARATOR (U+2028)也视为新行。值得注意的是，.和s应该算作一个字符，而不是按照SQL的规定算作两个字符。
- 在Table 9.20中描述的字符输入转义中，XQuery只支持n、r和t。
- XQuery不支持[:name:]语法，不支持括号表达式中的字符类。
- XQuery没有 lookahead 或 lookbehind 约束，也没有任何在 Table 9.22
- Section 9.7.3.4中描述的metasyntax形式在XQuery中不存在。
- 由XQuery定义的正则表达式标志字母与POSIX的选项字母相关，但不一样。Table 9.24。虽然i和q选项的行为是一样的，但其他的选项却不一样。

- XQuery的s（允许点匹配换行）和m（允许 $\wedge$ 和 $\$$ 在换行处匹配）标志提供了与POSIX的n相同的行为。p和w标志，但它们与POSIX的s和m标志的行为不匹配。特别要注意的是，点匹配-newline是POSIX中的默认行为，但不是XQuery。
- XQuery的x（忽略模式中的空格）标志与POSIX的扩展模式标志明显不同。POSIX的x标志也允许#在模式中开始注释，并且POSIX不会忽略反斜线后的空格字符。

## 9.8. 数据类型格式化函数

LightDB格式化函数提供一套强大的工具用于把各种数据类型（日期/时间、整数、浮点、数字）转换成格式化的字符串以及反过来从格式化的字符串转换成指定的数据类型。Table 9.2列出了这些函数。这些函数都遵循一个公共的调用规范：第一个参数是待格式化的值，而第二个是一个定义输出或输入格式的模板。

Table 9.25. 格式化函数

函数	描述 例子
<code>to_char ( timestamp, text )</code>	<code>to_char ( timestamp, text )</code> → text
<code>to_char ( timestamp with time zone, text )</code>	<code>to_char ( timestamp with time zone, text )</code> → text 根据给定的格式将时间戳转换为字符串。 <code>to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS')</code> → 05:31:12
<code>to_char ( interval, text )</code>	<code>to_char ( interval, text )</code> → text 根据给定的格式将间隔转换为字符串。 <code>to_char(interval '15h 2m 12s', 'HH24:MI:SS')</code> → 15:02:12
<code>to_char ( numeric_type, text )</code>	<code>to_char ( numeric_type, text )</code> → text 根据给定的格式将数字转换为字符串；适用于 integer, bigint, numeric, real, double precision。 <code>to_char(125, '999')</code> → 125 <code>to_char(125.8::real, '999D9')</code> → 125.8 <code>to_char(-125.8, '999D99S')</code> → 125.80-
<code>to_date ( text, text )</code>	<code>to_date ( text, text )</code> → date 根据给定的格式将字符串转换为日期。 <code>to_date('05 Dec 2000', 'DD Mon YYYY')</code> → 2000-12-05
<code>to_number ( text, text )</code>	<code>to_number ( text, text )</code> → numeric 根据给定的格式将字符串转换为数字。 <code>to_number('12,454.8-', '99G999D9S')</code> → -12454.8
<code>to_timestamp ( text, text )</code>	<code>to_timestamp ( text, text )</code> → timestamp with time zone 根据给定的格式将字符串转换为时间戳。（也请参见Table 9.32中的 <code>to_timestamp(double precision)</code> 。） <code>to_timestamp('05 Dec 2000', 'DD Mon YYYY')</code> → 2000-12-05 00:00:00-05

### Tip

`to_timestamp`和`to_date`存在的目的是为了处理无法用简单造型转换的输入格式。对于大部分标准的日期/时间格式，简单地把源字符串造型成所需的数据类型是可以的，并且简单很多。类似地，对于标准的数字表示形式，`to_number`也是没有必要的。

在一个`to_char`输出模板串中，一些特定的模式可以被识别并且被替换成基于给定值的被恰当地格式化的数据。任何不属于模板模式的文本都简单地照字面拷贝。同样，在一个输入

模板串里（对其他函数），模板模式标识由输入数据串提供的值。如果在模板字符串中有不是模板模式的字符，输入数据字符串中的对应字符会被简单地跳过（不管它们是否等于模板字符串字符）。

Table 9.2展示了可以用于格式化日期和时间值的模版。

Table 9.26. 用于日期/时间格式化的模板模式

模式	描述
HH	一天中的小时 (01 - 12)
HH12	一天中的小时 (01 - 12)
HH24	一天中的小时 (00 - 23)
MI	分钟 (00 - 59)
SS	秒 (00 - 59)
MS	毫秒 (000 - 999)
US	微秒 (000000 - 999999)
FF1	十分之一秒 (0 - 9)
FF2	百分之一秒 (00 - 99)
FF3	毫秒 (000 - 999)
FF4	十分之一毫秒 (0000 - 9999)
FF5	百分之一毫秒 (00000 - 99999)
FF6	微秒 (000000 - 999999)
SSSS, SSSSS	午夜后的秒 (0 - 86399)
AM, am, PM or pm	正午指示器 (不带句号)
A. M. , a. m. , P. M. or p. m.	正午指示器 (带句号)
Y, YYYY	带逗号的年 (4 位或者更多位) with comma
YYYY	年 (4 位或者更多位)
YYY	年的最后 3 位数字
YY	年的最后 2 位数字
Y	年的最后 1 位数字
IYYYY	ISO 8601 周编号方式的年 (4 位或更多位)
IYY	ISO 8601 周编号方式的年的最后 3 位数字
IY	ISO 8601 周编号方式的年的最后 2 位数字
I	ISO 8601 周编号方式的年的最后 1 位数字
BC, bc, AD 或 ad	纪元指示器 (不带句号)
B. C. , b. c. , A. D. 或 a. d.	纪元指示器 (带句号)
MONTH	全大写形式的月名 (空格补齐到 9 字符)
Month	全首字母大写形式的月名 (空格补齐到 9 字符)
month	全小写形式的月名 (空格补齐到 9 字符)
MON	简写的大写形式的月名 (英文 3 字符, 本地化长度可变)
Mon	简写的首字母大写形式的月名 (英文 3 字符, 本地化长度可变)
mon	简写的小写形式的月名 (英文 3 字符, 本地化长度可变)

模式	描述
MM	月编号 (01 - 12)
DAY	全大写形式的日名 (空格补齐到 9 字符)
Day	全首字母大写形式的日名 (空格补齐到 9 字符)
day	全小写形式的日名 (空格补齐到 9 字符)
DY	简写的大写形式的日名 (英语 3 字符, 本地化长度可变)
Dy	简写的首字母大写形式的日名 (英语 3 字符, 本地化长度可变)
dy	简写的小写形式的日名 (英语 3 字符, 本地化长度可变)
DDD	一年中的日 (001 - 366)
IDDD	ISO 8601 周编号方式的年中的日 (001 - 371; 年的第 1 日时第一个 ISO 周的周一)
DD	月中的日 (01 - 31)
D	周中的日, 周日 (1) 到周六 (7)
ID	周中的 ISO 8601 日, 周一 (1) 到周日 (7)
W	月中的周 (1 - 5) (第一周从该月的第一天开始)
WW	年中的周数 (1 - 53) (第一周从该年的第一天开始)
IW	ISO 8601 周编号方式的年中的周数 (01 - 53; 新的一年的第一个周四在第一周)
CC	世纪 (2 位数) (21 世纪开始于 2001-01-01)
J	儒略日 (从午夜 UTC 的公元前 4714 年 11 月 24 日开始的整数日数)
Q	季度
RM	大写形式的罗马计数法的月 (I - XII; I=一月)
rm	小写形式的罗马计数法的月 (i - xii; i=一月)
TZ	大写形式的时区缩写 (仅在to_char中支持)
tz	小写形式的时区缩写 (仅在to_char中支持)
TZH	时区的小时
TZM	时区的分钟
OF	从UTC开始的时区偏移 (仅在to_char中支持)

修饰语可以被应用于模板模式来修改它们的行为。例如, FMMonth就是带着FM修饰语的Month模式。Table 9.2展示了可用于日期/时间格式化的修饰语模式。

Table 9.27. 用于日期/时间格式化的模板模式修饰语

修饰语	描述	例子
FM prefix	填充模式 (抑制前导零和填充的空格)	FMMonth

修饰语	描述	例子
TH suffix	大写形式的序数后缀	DDTH, e. g., 12TH
th suffix	小写形式的序数后缀	DDth, e. g., 12th
FX prefix	固定的格式化全局选项（见使用须知）	FX Month DD Day
TM prefix	翻译模式（基于lc_time使用本地化的日和月名）	TMMonth
SP suffix	拼写模式（未实现）	DDSP

日期/时间格式化的使用须知：

- FM抑制前导的零或尾随的空白，否则会把它们增加到输入从而把一个模式的输出变成固定宽度。在LightDB中，FM只修改下一个声明，而在Oracle中，FM影响所有随后的声明，并且重复的FM修饰语将触发填充模式开和关。
- 无论是否指定FM，TM抑制尾随的空格。
- to\_timestamp和to\_date忽略输入中的字母大小写；例如MON，Mon和mon都接受相同的字符串。当使用TM修饰符时，大小写折叠是根据函数的输入排序规则进行的（参见Section 21.2）。
- to\_timestamp和to\_date跳过了输入字符串开头和日期和时间值周围的多个空格，除非使用了FX选项。例如，to\_timestamp(' 2000 JUN', 'YYY-MON')和to\_timestamp('2000 - JUN', 'YYY-MON')都能工作，但to\_timestamp('2000 JUN', 'FXYYY-MON')返回一个错误，因为to\_timestamp只期望一个空格。FX必须指定为模板中的第一个项目。
- to\_timestamp和to\_date的模板字符串中的分隔符（一个空格或非字母/非数字字符）与输入字符串中的任何一个分隔符相匹配，或者被跳过，除非使用了FX选项。例如，to\_timestamp('2000JUN', 'YYY//MON')和to\_timestamp('2000/JUN', 'YYY/MON')可以工作，但to\_timestamp('2000/JUN', 'YYYY/MON')返回一个错误，因为输入字符串中的分隔符数量超过了模板中的分隔符数量。  
如果指定了FX，模板字符串中的分隔符正好与输入字符串中的一个字符匹配。但要注意的是，输入字符串中的字符不需要与模板字符串中的分隔符相同。例如，to\_timestamp('2000/JUN', 'FXYYYY MON')可以工作，但是to\_timestamp('2000/JUN', 'FXYYYY MON')返回错误，因为模板字符串中的第二个空格会消耗掉输入字符串中的字母J。
- TZH模板模式可以匹配一个有符号的数字。如果没有FX选项，减号可能是模糊的，可能被解释为分隔符。这种模棱两可的问题可以通过以下方式解决。如果模板字符串中TZH前的分隔符的数量小于输入字符串中减号前的分隔符数量，则减号被解释为TZH的一部分。否则，减号被认为是值之间的分隔符。例如，to\_timestamp('2000 -10', 'YYY TZH')与-10匹配，但to\_timestamp('2000 -10', 'YYYY TZH')匹配10到TZH。
- 在to\_char模板里可以有普通文本，并且它们会被照字面输出。你可以把一个子串放到双引号里强迫它被解释成一个文本，即使它里面包含模板模式也如此。例如，在'Hello Year YYYYY'中，YYYYY将被年份数据代替，但是Year中单独的Y不会。在to\_date、to\_number以及to\_timestamp中，文本和双引号字符串会导致跳过该字符串中所包含的字符数量，例如"XX"会跳过两个输入字符（不管它们是不是XX）。

### Tip

在LightDB 12之前，可以使用非字母或非数字字符跳过输入字符串中的任意文本。例如，to\_timestamp('2000y6m1d', 'yyyyy-MM-DD')以前是有效的。现在，你只能使用字母字符来实现这个目的。例如，to\_timestamp('2000y6m1d',

`'yyyymmdd')`和`to_timestamp('2000y6m1d', 'yyyy"y"MM"m"DD"d')`跳过`y`、`m`和`d`。

- 如果你在输出里有双引号，那么你必须在它们前面放反斜线，例如 `'\`YYYY Month\`'`。不然，在双引号字符串外面的反斜线就不是特殊的。在双引号字符串内，反斜线会导致下一个字符被取其字面形式，不管它是什么字符（但是这没有特殊效果，除非下一个字符是一个双引号或者另一个反斜线）。
- 在`to_timestamp`和`to_date`中，如果年份格式声明少于四位（如`YYY`）并且提供的年份少于四位，年份将被调整为最接近于 2020 年，例如95会变成 1995。
- 在`to_timestamp`和`to_date`中，负年份被视为表示BC。如果你同时写一个负年份和一个显式的BC字段，你又会得到AD。第0年的输入被视为公元前1年。
- 在`to_timestamp`和`to_date`中，在处理超过4位数的年份时，`YYYY`转换具有限制。你必须在`YYYY`后面使用一些非数字字符或者模板，否则年份总是被解释为 4 位数字。例如（对于 20000 年）：`to_date('200001131', 'YYYYMMDD')`将会被解释成一个 4 位数字的年份，而不是在年份后使用一个非数字分隔符，像`to_date('20000-1131', 'YYYY-MMDD')`或`to_date('20000Nov31', 'YYYYMonDD')`。
- 在`to_timestamp`和`to_date`中，`CC`（世纪）字段会被接受，但是如果有`YYY`、`YYYY`或者`Y`、`YY`字段则会忽略它。如果`CC`与`YY`或`Y`一起使用，则结果被计算为指定世纪中的那一年。如果指定了世纪但是没有指定年，则会假定为该世纪的第一年。
- 在`to_timestamp`和`to_date`中，工作日名称或编号（`DAY`、`D`以及相关的字段类型）会被接受，但会为了计算结果的目的而忽略。季度（`Q`）字段也是一样。
- 在`to_timestamp`和`to_date`中，一个 ISO 8601 周编号的日期（与一个格里高利日期相区别）可以用两种方法之一被指定为`to_timestamp`和`to_date`：
  - 年、周编号和工作日：例如`to_date('2006-42-4', 'IYYY-IW-ID')`返回日期2006-10-19。如果你忽略工作日，它被假定为 1（周一）。
  - 年和一年中的日：例如`to_date('2006-291', 'IYYY-IDDD')`也返回2006-10-19。

尝试使用一个混合了 ISO 8601 周编号和格里高利日期的域来输入一个日期是无意义的，并且将导致一个错误。在一个 ISO 周编号的年的环境下，一个“月”或“月中的日”的概念没有意义。在一个格里高利年的环境下，ISO 周没有意义。用户应当避免混合格里高利和 ISO 日期声明。

### Caution

虽然`to_date`将会拒绝混合使用格里高利和 ISO 周编号日期的域，`to_char`却不会，因为`YYYY-MM-DD (IYYY-IDDD)`这种输出格式也会有用。但是避免写类似`IYYY-MM-DD`的东西，那会得到在起始年附近令人惊讶的结果（详见Section 9.9.1）。

- 在`to_timestamp`中，毫秒（`MS`）和微秒（`US`）域都被用作小数点后的秒位。例如`to_timestamp('12.3', 'SS.MS')`不是 3 毫秒，而是 300，因为该转换把它看做  $12 + 0.3$  秒。这意味着对于格式`SS.MS`而言，输入值12.3、12.30和12.300指定了相同数目的毫秒。要得到三毫秒，你必须使用 12.003，转换会把它看做  $12 + 0.003 = 12.003$  秒。
- 下面是一个更复杂的例子：`to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US')`是 15 小时、12 分钟和 2 秒 + 20 毫秒 + 1230微秒 = 2.021230 秒。
- `to_char(..., 'ID')`的一周中日的编号匹配`extract(isodow from ...)`函数，但是`to_char(..., 'D')`不匹配`extract(dow from ...)`的日编号。



- `to_char(interval)` 格式化HH和HH12为显示在一个 12 小时的时钟上，即零小时和 36 小时输出为12，而HH24会输出完整的小时值，对于间隔它可以超过 23。

Table 9.2展示了可以用于格式化数字值的模版模式。

Table 9.28. 用于数字格式化的模板模式

模式	描述
9	数位（如果无意义可以被删除）
0	数位（即便没有意义也不会被删除）
. (period)	小数点
, (comma)	分组（千）分隔符
PR	尖括号内的负值
S	带符号的数字（使用区域）
L	货币符号（使用区域）
D	小数点（使用区域）
G	分组分隔符（使用区域）
MI	在指定位置的负号（如果数字 < 0）
PL	在指定位置的正号（如果数字 > 0）
SG	在指定位置的正/负号
RN	罗马数字（输入在 1 和 3999 之间）
TH or th	序数后缀
V	移动指定位数（参阅注解）
EEEE	科学记数的指数

数字格式化的用法须知：

- 0指定一个总是被打印的数位，即便它包含前导/拖尾的零。9也指定一个数位，但是如果它是前导零则会被空格替换，而如果是拖尾零并且指定了填充模式则它会被删除（对于`to_number()`来说，这两种模式字符等效）。
- 模式字符S、L、D以及G表示当前locale定义的负号、货币符号、小数点以及数字分隔符号（见`lc_monetary`和`lc_numeric`）。不管locale是什么，模式字符句号和逗号就表示小数点和数字分隔符。
- 对于`to_char()`的模式中的一个负号，如果没有明确的规定，将为该负号保留一列，并且它将被锚接到（出现在左边）那个数字。如果S正好出现在某个9的左边，它也将被锚接到那个数字。
- 使用SG、PL或MI格式化的符号并不挂在数字上面；例如，`to_char(-12, 'MI9999')`生成'- 12'而`to_char(-12, 'S9999')`生成'-12'（Oracle 里的实现不允许在9前面使用MI，而是要求9在MI前面。）
- TH不会转换小于零的数值，也不会转换小数。
- PL、SG和TH是LightDB扩展。
- 在`to_number`中，如果没有使用L或TH之类的非数据模板模式，相应数量的输入字符会被跳过，不管它们是否匹配模板模式，除非它们是数据字符（也就是数位、负号、小数点或者逗号）。例如，TH会跳过两个非数据字符。
- 带有`to_char`的V会把输入值乘上 $10^n$ ，其中n是跟在V后面的位数。带有`to_number`的V以类似的方式做除法。`to_char`和`to_number`不支持使用结合小数点的V（例如，不允许99.9V99）。

- EEEE（科学记数法）不能和任何其他格式化模式或修饰语（数字和小数点模式除外）组合在一起使用，并且必须位于格式化字符串的最后（例如9.99EEEE是一个合法的模式）。

某些修饰语可以被应用到任何模板来改变其行为。例如，FM99.99是带有FM修饰语的99.99模式。Table 9.29展示了用于数字格式化模式修饰语。

Table 9.29. 用于数字格式化的模板模式修饰语

修饰语	描述	例子
FM prefix	填充模式（抑制拖尾零和填充的空白）	FM99.99
TH suffix	大写序数后缀	999TH
th suffix	小写序数后缀	999th

Table 9.3展示了使用to\_char函数的例子。

Table 9.30. to\_char例子

表达式	结果
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(-0.1, 'FM90.99')	'-0.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'

表达式	结果
<code>to_char(485, 'L999')</code>	'DM 485'
<code>to_char(485, 'RN')</code>	' CDLXXXV'
<code>to_char(485, 'FMRN')</code>	'CDLXXXV'
<code>to_char(5.2, 'FMRN')</code>	'V'
<code>to_char(482, '999th')</code>	' 482nd'
<code>to_char(485, '"Good number:"999')</code>	'Good number: 485'
<code>to_char(485.8, 'Pre:"999" Post:" .999')</code>	'Pre: 485 Post: .800'
<code>to_char(12, '99V999')</code>	' 12000'
<code>to_char(12.4, '99V999')</code>	' 12400'
<code>to_char(12.45, '99V9')</code>	' 125'
<code>to_char(0.0004859, '9.99EEEE')</code>	' 4.86e-04'

## 9.9. 时间/日期函数和操作符

Table 9.3展示了可用于处理日期/时间值的函数，其细节在随后的小节中描述。Table 9.3演示了基本算术操作符（+、\*等）的行为。而与格式化相关的函数，可以参考Section 9.8 你应该很熟悉Section 8.5中的日期/时间数据类型的背景知识。

此外，对于日期/时间类型，您可以使用Table 9.4中列出的常规比较运算符。日期和时间戳（带或不带时区）都可以进行比较，而时间（带或不带时区）和间隔只能与相同数据类型的其他值进行比较。当将没有时区的时间戳与具有时区的时间戳进行比较时，前者的值被假定为在由TimeZone配置参数指定的时区中，并在进行比较之前被转换为UTC时间（后者已经在内部表示为UTC时间）。类似地，当将日期值与时间戳进行比较时，日期值被假定为在TimeZone时区中表示午夜时间。

所有下文描述的接受time或timestamp输入的函数和操作符实际上都有两种变体：一种接收time with time zone或timestamp with time zone，另外一种接受time without time zone或者timestamp without time zone。为了简化，这些变种没有被独立地展示。此外，+和\*操作符都是可交换的操作符对（例如，date + integer 和 integer + date）；我们只显示其中一个。

Table 9.31. 日期/时间操作符

操作符	描述 例子
<code>date + integer → date</code>	给日期加上天数 <code>date '2001-09-28' + 7 → 2001-10-05</code>
<code>date + interval → timestamp</code>	为日期添加时间间隔 <code>date '2001-09-28' + interval '1 hour' → 2001-09-28 01:00:00</code>
<code>date + time → timestamp</code>	在日期中添加一天中的时间 <code>date '2001-09-28' + time '03:00' → 2001-09-28 03:00:00</code>
<code>interval + interval → interval</code>	添加时间间隔 <code>interval '1 day' + interval '1 hour' → 1 day 01:00:00</code>
<code>timestamp + interval → timestamp</code>	在时间戳中添加一个时间间隔

操作符	描述 例子
	timestamp '2001-09-28 01:00' + interval '23 hours' → 2001-09-29 00:00:00
time + interval → time	为时间添加时间间隔 time '01:00' + interval '3 hours' → 04:00:00
- interval → interval	取否一个时间间隔 - interval '23 hours' → -23:00:00
date - date → integer	减去日期, 生成经过的天数 date '2001-10-01' - date '2001-09-28' → 3
date - integer → date	从日期中减去天数 date '2001-10-01' - 7 → 2001-09-24
date - interval → timestamp	从日期中减去时间间隔 date '2001-09-28' - interval '1 hour' → 2001-09-27 23:00:00
time - time → interval	减去时间 time '05:00' - time '03:00' → 02:00:00
time - interval → time	从时间中减去时间间隔 time '05:00' - interval '2 hours' → 03:00:00
timestamp - interval → timestamp	从时间戳中减去时间间隔 timestamp '2001-09-28 23:00' - interval '23 hours' → 2001-09-28 00:00:00
interval - interval → interval	减去时间间隔 interval '1 day' - interval '1 hour' → 1 day -01:00:00
timestamp - timestamp → interval	减去时间戳 (将24小时间隔转换为天, 类似于justify_hours()) timestamp '2001-09-29 03:00' - timestamp '2001-07-27 12:00' → 63 days 15:00:00
interval * double precision → interval	将时间间隔乘以数量 interval '1 second' * 900 → 00:15:00 interval '1 day' * 21 → 21 days interval '1 hour' * 3.5 → 03:30:00
interval / double precision → interval	用时间间隔除以数量 interval '1 hour' / 1.5 → 00:40:00

Table 9.32. 日期/时间函数

函数	描述 例子
age ( timestamp, timestamp ) → interval	减去参数, 生成一个使用年和月, 而不是只用日的“符号化”的结果

函数	描述 例子
	<code>age(timestamp '2001-04-10', timestamp '1957-06-13')</code> → 43 years 9 mons 27 days
<code>age ( timestamp )</code> → interval	从 <code>current_date</code> 减去参数(在午夜) <code>age(timestamp '1957-06-13')</code> → 62 years 6 mons 10 days
<code>clock_timestamp ( )</code> → timestamp with time zone	当前日期和时间 (在语句执行期间变化); 参见Section 9.9.4 <code>clock_timestamp()</code> → 2019-12-23 14:39:53.662522-05
<code>systimestamp</code> → timestamp with time zone	当前日期和时间 (当前语句的开始时间); 请参见Section 9.9.4 <code>systimestamp</code> → 2019-12-23 14:39:53.662522-05
<code>systimestamp ( integer )</code> → timestamp with time zone	当前日期和时间 (当前语句的开始时间); 请参见Section 9.9.4 <code>systimestamp(6)</code> → 2019-12-23 14:39:53.662522-05
<code>current_date</code> → date	当前日期; 参见 Section 9.9.4 <code>current_date</code> → 2019-12-23
<code>current_date ( )</code> → date	当前日期, 只在mysql模式下支持; 参见 Section 9.9.4 <code>current_date()</code> → 2019-12-23
<code>current_time</code> → time with time zone	一天中的当前时间; 参见 Section 9.9.4 <code>current_time</code> → 14:39:53.662522-05
<code>current_time ( integer )</code> → time with time zone	一天中的当前时间; 有限精度; 参见 Section 9.9.4 <code>current_time(2)</code> → 14:39:53.66-05
<code>current_timestamp</code> → timestamp with time zone	当前日期和时间 (当前事务的开始); 参见 Section 9.9.4 <code>current_timestamp</code> → 2019-12-23 14:39:53.662522-05
<code>current_timestamp ( integer )</code> → timestamp with time zone	当前日期和时间 (当前事务的开始); 有限精度; 参见 Section 9.9.4 <code>current_timestamp(0)</code> → 2019-12-23 14:39:53-05
<code>date_part ( text, timestamp )</code> → double precision	获取时间戳字段 (等同于 <code>extract</code> ); 参见 Section 9.9.1 <code>date_part('hour', timestamp '2001-02-16 20:38:40')</code> → 20
<code>date_part ( text, interval )</code> → double precision	获取时间间隔子字段 (等同于 <code>extract</code> ); 参见 Section 9.9.1 <code>date_part('month', interval '2 years 3 months')</code> → 3
<code>date_trunc ( text, timestamp )</code> → timestamp	截断到指定的精度; 参见 Section 9.9.2 <code>date_trunc('hour', timestamp '2001-02-16 20:38:40')</code> → 2001-02-16 20:00:00
<code>date_trunc ( text, timestamp with time zone, text )</code> → timestamp with time zone	在规定的时区中截断到指定的精度; 参见 Section 9.9.2 <code>date_trunc('day', timestamptz '2001-02-16 20:38:40+00', 'Australia/Sydney')</code> → 2001-02-16 13:00:00+00
<code>date_trunc ( text, interval )</code> → interval	截断到指定的精度; 参见 Section 9.9.2

函数	描述 例子
	<code>date_trunc('hour', interval '2 days 3 hours 40 minutes')</code> → 2 days 03:00:00
	<code>extract ( field from timestamp )</code> → double precision 获取时间戳子字段; 参见 Section 9.9.1 <code>extract(hour from timestamp '2001-02-16 20:38:40')</code> → 20
	<code>extract ( field from interval )</code> → double precision 获取时间间隔子字段; 参见 Section 9.9.1 <code>extract(month from interval '2 years 3 months')</code> → 3
	<code>isfinite ( date )</code> → boolean 测试有限日期 (不是+/-无限) <code>isfinite(date '2001-02-16')</code> → true
	<code>isfinite ( timestamp )</code> → boolean 测试有限时间戳 (不是+/-无限) <code>isfinite(timestamp 'infinity')</code> → false
	<code>isfinite ( interval )</code> → boolean 测试有限时间间隔 (当前总是为真) <code>isfinite(interval '4 hours')</code> → true
	<code>justify_days ( interval )</code> → interval 调整间隔, 使得30天时间周期表示为月 <code>justify_days(interval '35 days')</code> → 1 mon 5 days
	<code>justify_hours ( interval )</code> → interval 调整时间间隔, 使得24小时时间周期表示为日 <code>justify_hours(interval '27 hours')</code> → 1 day 03:00:00
	<code>justify_interval ( interval )</code> → interval 使用 <code>justify_days</code> 和 <code>justify_hours</code> 调整时间间隔; 通过额外的符号调整 <code>justify_interval(interval '1 mon -1 hour')</code> → 29 days 23:00:00
	<code>localtime</code> → time 一天中当前时间; 参见 Section 9.9.4 <code>localtime</code> → 14:39:53.662522
	<code>localtime ( integer )</code> → time 一天中的当前时间, 有限精度; 参见 Section 9.9.4 <code>localtime(0)</code> → 14:39:53
	<code>localtimestamp</code> → timestamp 当前日期和时间 (当前事务的开始); 参见 Section 9.9.4 <code>localtimestamp</code> → 2019-12-23 14:39:53.662522
	<code>localtimestamp ( integer )</code> → timestamp 当前日期和时间 (当前事务的开始); 有限精度; 参见 Section 9.9.4 <code>localtimestamp(2)</code> → 2019-12-23 14:39:53.66
	<code>sysdate</code> → timestamp 当前日期和时间 (当前语句的开始时间); 请参见Section 9.9.4 <code>sysdate</code> → 2019-12-23 14:39:53
	<code>make_date ( year int, month int, day int )</code> → date 从年、月和日字段创建日期 <code>make_date(2013, 7, 15)</code> → 2013-07-15
	<code>make_interval ( [ years int [, months int [, weeks int [, days int [, hours int [, mins int [, secs double precision ]]]]] ] )</code> → interval 从年、月、周、日、小时、分钟和秒字段创建时间间隔, 每个字段默认为0 <code>make_interval(days =&gt; 10)</code> → 10 days

函数	描述 例子
<code>make_time</code>	<code>( hour int, min int, sec double precision ) → time</code> 从小时、分钟和秒字段创建时间 <code>make_time(8, 15, 23.5) → 08:15:23.5</code>
<code>make_timestamp</code>	<code>( year int, month int, day int, hour int, min int, sec double precision ) → timestamp</code> 从年、月、日、小时、分钟和秒字段创建时间戳 <code>make_timestamp(2013, 7, 15, 8, 15, 23.5) → 2013-07-15 08:15:23.5</code>
<code>make_timestamptz</code>	<code>( year int, month int, day int, hour int, min int, sec double precision [, timezone text ] ) → timestamp with time zone</code> 从年、月、日、小时、分钟和秒字段结合时区创建时间戳；如果没有指定 <code>timezone</code> ，则使用当前时区 <code>make_timestamptz(2013, 7, 15, 8, 15, 23.5) → 2013-07-15 08:15:23.5+01</code>
<code>now</code>	<code>( ) → timestamp with time zone</code> 当前日期和时间(当前事务的开始)；参见 Section 9.9.4 <code>now() → 2019-12-23 14:39:53.662522-05</code>
<code>statement_timestamp</code>	<code>( ) → timestamp with time zone</code> 当前日期和时间(当前语句的开始)；参见 Section 9.9.4 <code>statement_timestamp() → 2019-12-23 14:39:53.662522-05</code>
<code>timeofday</code>	<code>( ) → text</code> 当前的日期和时间（类似 <code>clock_timestamp</code> ，但是采用 <code>text</code> 字符串）；参见 Section 9.9.4 <code>timeofday() → Mon Dec 23 14:39:53.662522 2019 EST</code>
<code>transaction_timestamp</code>	<code>( ) → timestamp with time zone</code> 当前日期和时间(当前事务的开始)；参见 Section 9.9.4 <code>transaction_timestamp() → 2019-12-23 14:39:53.662522-05</code>
<code>to_timestamp</code>	<code>( double precision ) → timestamp with time zone</code> 将Unix纪元转换为带时区的时间戳(从1970-01-01 00:00:00+00开始的秒) <code>to_timestamp(1284352323) → 2010-09-13 04:32:03+00</code>

除了这些函数以外，还支持 SQL 操作符 `OVERLAPS`：

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

这个表达式在两个时间域（用它们的端点定义）重叠的时候得到真，当它们不重叠时得到假。端点可以用一对日期、时间或者时间戳来指定；或者是用一个后面跟着一个间隔的日期、时间或时间戳来指定。当一对值被提供时，起点或终点都可以被写在前面，`OVERLAPS`会自动地把较早的值作为起点。每一个时间段被认为是表示半开的间隔 `start <= time < end`，除非 `start` 和 `end` 相等，这种情况下它表示单个时间实例。例如这表示两个只有一个共同端点的时间段不重叠。

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
```

结果: true

```
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
```

结果: false

```
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
```

结果: false

```
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
```

```
(DATE '2001-10-30', DATE '2001-10-31');
结果: true
```

当把一个interval值添加到timestamp with time zone上（或从中减去）时，天的部分会按照指定的天数增加或减少timestamp with time zone的日期，保持一天中相同的时间。对于横跨夏令时的变化（当会话的时区被设置为可识别DST的时区时），这意味着interval '1 day'并不一定等于interval '24 hours'。例如，当会话的时区设置为America/Denver时：

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '1 day';
Result: 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '24 hours';
Result: 2005-04-03 13:00:00-06
```

发生此情况是因为夏令时在America/Denver时区的2005-04-03 02:00:00的时间发生更改而跳过了一个小时。

注意age返回的月数域可能有歧义，因为不同的月份有不同的天数。LightDB的方法是当计算部分月数时，采用两个日期中较早的月。例如：age('2004-06-01', '2004-04-30')使用4月份得到1 mon 1 day，而用5月分会得到1 mon 2 days，因为5月有31天，而4月只有30天。

日期和时间戳的减法也可能会很复杂。执行减法的一种概念上很简单的方法是，使用EXTRACT(EPOCH FROM ...)把每个值都转换成秒数，然后执行减法，这样会得到两个值之间的秒数。这种方法将会适应每个月中天数、时区改变和夏令时调整。使用“-”操作符的日期或时间戳减法会返回值之间的天数（24小时）以及时/分/秒，也会做同样的调整。age函数会返回年、月、日以及时/分/秒，执行按域的减法，然后对负值域进行调整。下面的查询展示了这些方法的不同。例子中的结果由timezone = 'US/Eastern'产生，这使得两个使用的日期之间存在着夏令时的变化：

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
Result: 10537200
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
       / 60 / 60 / 24;
Result: 121.958333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';
Result: 121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01
12:00:00');
Result: 4 mons
```

## 9.9.1. EXTRACT, date\_part

```
EXTRACT(field FROM source)
```

extract函数从日期/时间值中抽取子域，例如年或者小时等。source必须是一个类型timestamp、time或interval的值表达式（类型为date的表达式将被造型为timestamp，并且因此也可以被同样使用）。field是一个标识符或者字符串，它指定从源值中抽取的域。extract函数返回类型为double precision的值。下列值是有效的域名字：

```
century
```

```
世纪
```

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
结果: 20
```



```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
结果: 21
```

第一个世纪从 0001-01-01 00:00:00 AD 开始，尽管那时候人们还不知道这是第一个世纪。这个定义适用于所有使用格里高利历法的国家。其中没有 0 世纪，我们直接从公元前 1 世纪到公元 1 世纪。如果你认为这个不合理，那么请把抱怨发给：罗马圣彼得教堂，梵蒂冈，教皇收。

#### day

对于timestamp值，是（月份）里的日域（1 - 31）；对于interval值，是日数

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
结果: 16
```

```
SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
结果: 40
```

#### decade

年份域除以10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
结果: 200
```

#### dow

一周中的日，从周日（0）到周六（6）

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
结果: 5
```

请注意，extract的一周中的日和to\_char(..., 'D')函数不同。

#### doy

一年的第几天（1 - 365/366）

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
结果: 47
```

#### epoch

对于timestamp with time zone值，是自 1970-01-01 00:00:00 UTC 以来的秒数（结果可能是负数）；对于date and timestamp值，是自本地时间 1970-01-01 00:00:00 以来的描述；对于interval值，它是时间间隔的总秒数。

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16
20:38:40.12-08');
结果: 982384720.12
```

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40.12');
结果: 982355920.12
```

```
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
结果: 442800
```

不能用to\_timestamp把一个 epoch 值转换回成时间戳：

```
SELECT to_timestamp(982384720.12);  
Result: 2001-02-17 04:38:40.12+00
```

请注意，将to\_timestamp应用于从date或timestamp值中提取的纪元时，可能会产生误导性的结果：结果将实际上假定原始值是以UTC时间给出的，但这可能不是实际情况。

#### hour

小时域（0 - 23）

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');  
结果: 20
```

#### isodow

一周中的日，从周一（1）到周日（7）

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');  
结果: 7
```

除了周日，这和dow相同。这符合ISO 8601 中一周中的日的编号。

#### isoyear

日期所落在的ISO 8601 周编号的年（不适用于间隔）

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');  
结果: 2005  
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');  
结果: 2006
```

每一个ISO 8601 周编号的年都开始于包含1月4日的那一周的周一，在早的1月或迟的12月中ISO年可能和格里高利年不同。更多信息见week域。

#### julian

与日期或时间戳对应的儒略日（不适用于间隔）。不是当地午夜的时间戳会得到一个小数值。有关更多信息，请参见Section D.7

```
SELECT EXTRACT(JULIAN FROM DATE '2006-01-01');  
Result: 2453737  
SELECT EXTRACT(JULIAN FROM TIMESTAMP '2006-01-01 12:00');  
Result: 2453737.5
```

#### microseconds

秒域，包括小数部分，乘以 1,000,000。请注意它包括全部的秒

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');  
结果: 28500000
```

#### millennium

千年

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');  
结果: 3
```

19xx的年份在第二个千年里。第三个千年从 2001 年 1 月 1 日开始。

milliseconds

秒域，包括小数部分，乘以 1000。请注意它包括完整的秒。

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');  
结果: 28500
```

minute

分钟域 (0 - 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');  
结果: 38
```

month

对于timestamp值，它是一年里的月份数 (1 - 12)；对于interval值，它是月的数目，然后对 12 取模 (0 - 11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');  
结果: 2
```

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');  
结果: 3
```

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');  
结果: 1
```

quarter

该天所在的该年的季度 (1 - 4)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');  
结果: 1
```

second

秒字段，包括任何小数秒。

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');  
结果: 40
```

```
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');  
结果: 28.5
```

timezone

与 UTC 的时区偏移，以秒记。正数对应 UTC 东边的时区，负数对应 UTC 西边的时区 (从技术上来看，LightDB不使用 UTC，因为其中不处理闰秒)。

timezone\_hour

时区偏移的小时部分。

timezone\_minute

时区偏移的分钟部分。

week

该天在所在的ISO 8601 周编号的年份里是第几周。根据定义，一年的第一周包含该年的 1 月 4 日并且 ISO 周从星期一开始。换句话说，一年的第一个星期四在第一周。

在 ISO 周编号系统中，早的 1 月的日期可能位于前一年的第五十二或者第五十三周，而迟的 12 月的日期可能位于下一年的第一周。例如，2005-01-01位于 2004 年的第五十三周，并且2006-01-01位于 2005 年的第五十二周，而2012-12-31位于 2013 年的第一周。我们推荐把isoyear域和week一起使用来得到一致的结果。

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果： 7

year

年份域。要记住这里没有 0 AD，所以从AD年里抽取BC年应该小心处理。

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
```

结果： 2001

### Note

当输入值为  $\pm\infty$  时，extract对于单调增的域（epoch、julian、year、isoyear、decade、century以及millennium）返回  $\pm\infty$ 。对于其他域返回 NULL。LightDB 9.6 之前的版本对所有输入无穷的情况都返回零。

extract函数主要的用途是做计算性处理。对于用于显示的日期/时间值格式化，参阅Section 9.8

在传统的Ingres上建模的date\_part函数等价于SQL标准函数extract：

```
date_part('field', source)
```

请注意这里的field参数必须是一个串值，而不是一个名字。有效的date\_part域名和extract相同。

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
```

结果： 16

```
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
```

结果： 4

## 9.9.2. date\_trunc

date\_trunc函数在概念上和用于数字的trunc函数类似。

```
date_trunc(field, source [, time_zone ])
```

source是类型timestamp或interval的值表达式（类型date和 time的值都分别被自动转换成timestamp, timestamp with time zone, 或者interval）。field选择对输入值选用什么样的精度进行截断。返回的值是timestamp, timestamp with time zone, 类型或者所有小于选定的精度的域都设置为零（或者一，对于日期和月份）的interval。

field的有效值是：

microseconds  
 milliseconds  
 second  
 minute  
 hour  
 day  
 week  
 month  
 quarter  
 year  
 decade  
 century  
 millennium

当输入值的类型为timestamp with time zone时。截断是针对特定时区进行的。例如，截断为day，产生的值是 是该区域的午夜。默认情况下，截断是在以下方面进行的 到当前的TimeZone设置，但在当前的 可以提供可选的time\_zone参数。以指定不同的时区。可以指定时区名称 Section 8.5. 中描述的任何一种方式。

当处理timestamp without time zone 或interval输入时，不能指定时区。这些总是按表面值来处理。

示例（假设当地时区为 America/New\_York）：

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
结果：2001-02-16 20:00:00
```

```
SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
结果：2001-01-01 00:00:00
```

```
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00');
Result: 2001-02-16 00:00:00-05
```

```
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00',
'Australia/Sydney');
Result: 2001-02-16 08:00:00-05
```

```
SELECT date_trunc('hour', INTERVAL '3 days 02:47:33');
Result: 3 days 02:00:00
```

### 9.9.3. AT TIME ZONE

AT TIME ZONE把时间戳without时区转换成时间戳with时区或者反过来，并且把time with time zone值转换成不同的时区。Table 9.3展示了它的变体。

Table 9.33. AT TIME ZONE变体

操作符	描述 例子
timestamp without time zone AT TIME ZONE zone → timestamp with time zone	将给定的时间戳without时区转换为时间戳with时区，假设给定的值在指定的时区内。 timestamp '2001-02-16 20:38:40' at time zone 'America/Denver' → 2001-02-17 03:38:40+00
timestamp with time zone AT TIME ZONE zone → timestamp without time zone	

操作符	描述 例子
	将给定的时间戳with时区转换为时间戳without时区，因为时间将出现在该时区中 timestamp with time zone '2001-02-16 20:38:40-05' at time zone 'America/Denver' → 2001-02-16 18:38:40
	time with time zone AT TIME ZONE zone → time with time zone 将给定的时间with时区转换为新的时区。由于没有提供日期，这将使用指定目的区域的当前活动UTC偏移量。 time with time zone '05:34:17-05' at time zone 'UTC' → 10:34:17+00

在这些表达式里，我们需要的时区zone可以指定为文本值（例如，'America/Los\_Angeles'）或者一个间隔（例如，INTERVAL '-08:00'）。在文本情况下，可用的时区名字可以用Section 8.5.8中描述的任何方式指定。时间区间只适用于与UTC有固定偏移量的区域，因此在实践中并不常见。

例子（假设当前 TimeZone 设置为 America/Los\_Angeles）：

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 19:38:40-08
```

```
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 18:38:40
```

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'America/Chicago';
Result: 2001-02-16 05:38:40
```

第一个例子给缺少时区的值加上了时区，并且显示了使用当前TimeZone设置的值。第二个例子把带有时区值的时间戳移动到指定的时区，并且返回不带时区的值。这允许存储和显示不同于当前TimeZone设置的值。第三个例子把东京时间转换成芝加哥时间。

函数timezone(zone, timestamp)等效于 SQL 兼容的结构timestamp AT TIME ZONE zone。

## 9.9.4. 当前日期/时间

LightDB提供了许多返回当前日期和时间的函数。这些 SQL 标准的函数全部都按照当前事务的开始时刻返回值：

```
CURRENT_DATE
CURRENT_DATE()
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
SYSDATE
SYSTIMESTAMP
SYSTIMESTAMP(precision)
```

CURRENT\_TIME、CURRENT\_TIMESTAMP和SYSTIMESTAMP传递带有时区的值；LOCALTIME和LOCALTIMESTAMP传递的值不带时区。

CURRENT\_TIME、CURRENT\_TIMESTAMP、LOCALTIME和 LOCALTIMESTAMP可以有选择地接受一个精度参数， 该精度导致结果的秒域被园整为指定小数位。如果没有精度参数，结果将被给予所能得到的全部精度。

CURRENT\_DATE() 和 CURRENT\_DATE 相同。

一些例子：

```
SELECT CURRENT_TIME;  
结果：14:39:53.662522-05
```

```
SELECT CURRENT_DATE;  
结果：2019-12-23
```

```
SELECT CURRENT_DATE();  
结果：2019-12-23
```

```
SELECT CURRENT_TIMESTAMP;  
结果：2019-12-23 14:39:53.662522-05
```

```
SELECT CURRENT_TIMESTAMP(2);  
结果：2019-12-23 14:39:53.66-05
```

```
SELECT LOCALTIMESTAMP;  
结果：2019-12-23 14:39:53.662522
```

```
SELECT SYSDATE;  
结果：2019-12-23 14:39:53
```

```
SELECT SYSTIMESTAMP;  
结果：2019-12-23 14:39:53.662522-05
```

```
SELECT SYSTIMESTAMP(6);  
结果：2019-12-23 14:39:53.662522-05
```

因为这些函数全部都按照当前事务的开始时刻返回结果，所以它们的值在事务运行的整个期间内都不改变。 我们认为这是一个特性：目的是为了允许一个事务在“当前”时间上有一致的概念， 这样在同一个事务里的多个修改可以保持同样的时间戳。

### Note

许多其它数据库系统可能会更频繁地推进这些值。

LightDB还提供了返回当前语句的开始时间以及 调用该函数时的实际当前时间的函数。这些非 SQL 标准的函数列表如下：

```
transaction_timestamp()  
statement_timestamp()  
clock_timestamp()  
timeofday()  
now()
```

transaction\_timestamp() 等价于CURRENT\_TIMESTAMP，但是其命名清楚地反映了它的返回值。statement\_timestamp() 返回当前语句的开始时刻（更准确的说是收到 客户端最后一条命令的时间）。statement\_timestamp() 和transaction\_timestamp() 在一个事务的第一条命

令期间返回值相同，但是在随后的命令中却不一定相同。 `clock_timestamp()` 返回真正的当前时间，因此它的值甚至在同一条 SQL 命令中都会变化。`timeofday()` 是一个有历史原因的 LightDB 函数。和 `clock_timestamp()` 相似，`timeofday()` 也返回真实的当前时间，但是它的结果是一个格式化的 text 串，而不是 timestamp with time zone 值。`now()` 是 LightDB 的一个传统，等效于 `transaction_timestamp()`。

所有日期/时间类型还接受特殊的文字值 `now`，用于指定当前的日期和时间（重申，被解释为当前事务的开始时刻）。因此，下面三个都返回相同的结果：

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now'; -- 但请参阅下面的提示
```

### Tip

当指定以后要计算的值时，不要使用第三种形式，例如在表列的 DEFAULT 子句中。系统将在分析这个常量的时候把 `now` 转换为一个 timestamp，这样需要默认值时就会得到创建表的时间！而前两种形式要到实际使用缺省值的时候才被计算，因为它们是函数调用。因此它们可以给出每次插入行的时刻。（参见 Section 8.5.1.4）

## 9.9.5. nls\_date\_format/nls\_timestamp\_format

仅在 Oracle 兼容模式下，LightDB 才能配置 `nls_date_format/nls_timestamp_format` 参数，设置 `sysdate` 和 `systimestamp` 的显示输出格式。

您必须先为 `nls_date_format/nls_timestamp_format` 变量设置 `sysdate/systimestamp` 格式。下面是使用 SET 语句进行设置的示例。

```
SET nls_date_format = 'YYYY/MM/DD HH24:MI:SS';
SET nls_timestamp_format = 'YYYY/MM/DD HH24:MI:SS.FF6TZH';
```

Table 9.3 展示了用于格式化 `sysdate` 和 `systimestamp` 值的模板模式。

Table 9.34. nls\_date\_format/nls\_timestamp\_format 格式化的模板模式

模式	描述
HH	小时 (01 - 12)
HH12	小时 (01 - 12)
HH24	小时 (00 - 23)
MI	分钟 (00 - 59)
SS	秒 (00 - 59)
MS	毫秒 (000 - 999)
US	微秒 (000000 - 999999)
FF1	十分之一秒 (0 - 9)
FF2	百分之一秒 (00 - 99)
FF3	毫秒 (000 - 999)
FF4	千分之一秒 (0000 - 9999)
FF5	万分之一秒 (00000 - 99999)
FF6	微秒 (000000 - 999999)



模式	描述
SSSS, SSSSS	午夜后的秒数 (0 - 86399)
AM, am, PM or pm	上午/下午指示符 (without periods)
A.M., a.m., P.M. or p.m.	上午/下午指示符 (with periods)
Y, YYYY	带逗号的年份 (4位或更多位数字)
YYYY	年份 (4位或更多位数字)
YYY	年份的最后3位
YY	年份的最后2位
Y	年份的最后1位
IYYYY	ISO 8601年度周数 (4位或更多位数字)
IYY	ISO 8601年度周数的最后3位
IY	ISO 8601年度周数的最后2位
I	ISO 8601年度周数的最后1位
BC, bc, AD or ad	纪元指示符 (不带句号)
B.C., b.c., A.D. or a.d.	纪元指示符 (带句号)
MONTH	全大写的月份名称 (空格填充至9个字符)
Month	首字母大写的月份名称 (空格填充至9个字符)
month	全小写的月份名称 (空格填充至9个字符)
MON	缩写的全大写月份名称 (英文为3个字符, 本地化长度可能有所不同)
Mon	缩写的首字母大写月份名称 (英文为3个字符, 本地化长度可能有所不同)
mon	缩写的全小写月份名称 (英文为3个字符, 本地化长度可能有所不同)
MM	月份数字 (01 - 12)
DAY	全大写的星期名称 (空格填充至9个字符)
Day	首字母大写的星期名称 (空格填充至9个字符)
day	全小写的星期名称 (空格填充至9个字符)
DY	缩写的全大写星期名称 (英文为3个字符, 本地化长度可能有所不同)
Dy	缩写的首字母大写星期名称 (英文为3个字符, 本地化长度可能有所不同)
dy	缩写的全小写星期名称 (英文为3个字符, 本地化长度可能有所不同)
DDD	年份中的第几天 (001 - 366)
IDDD	ISO 8601年度周数中的第几天 (001 - 371; 该年的第1天是第1周的周一)
DD	月份中的第几天 (01 - 31)
D	星期几, 周日为 1, 周六为 7
ID	ISO 8601日期格式中的星期几, 周一为 1, 周日为 7
W	月份中的第几周 (1 - 5; 第1周从本月的第1天开始)

模式	描述
WW	年份中的第几周（1 - 53；第1周从本年的第1天开始）
IW	ISO 8601年度周数中的第几周（01 - 53；本年的第一个星期四所在的周为第1周）
CC	世纪（2位数字）（21世纪从2001-01-01开始）
J	朱利安日期（自公元前4714年11月24日零时起的整数天数；参见 Section D.7）
Q	季度
RM	用大写罗马数字表示的月份（I - XII；其中I表示1月）
rm	用小写罗马数字表示的月份（i - xii；其中i表示1月）
TZ	时区缩写（全大写）
tz	时区缩写（全小写）
TZH	时区的小时数
TZM	时区的分钟数
OF	与UTC相差的小时和分钟数

### 9.9.6. 延时执行

下面的这些函数可以用于让服务器进程延时执行：

```
pg_sleep ( double precision )
pg_sleep_for ( interval )
pg_sleep_until ( timestamp with time zone )
```

`pg_sleep`使当前会话的进程休眠，直到过去给定的秒数。可以指定几分之一秒的延迟。`pg_sleep_for`是一个方便的函数，允许将睡眠时间指定为时间间隔。`pg_sleep_until`是一个方便的函数，用于需要特定的唤醒时间。例如：

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

#### Note

有效的休眠时间间隔精度是平台相关的，通常 0.01 秒是通用值。休眠延迟将至少持续指定的时长，也有可能由于服务器负荷而比指定的时间长。特别地，`pg_sleep_until`并不保证能刚好在指定的时刻被唤醒，但它不会在比指定时刻早的时候醒来。

#### Warning

请确保在调用`pg_sleep`或者其变体时，你的会话没有持有不必要的锁。否则其它会话可能必须等待你的休眠会话，因而减慢整个系统速度。

## 9.10. 枚举支持函数

对于枚举类型(在Section 8. 中描述)，有一些函数允许更清洁的编码，而不需要为一个枚举类型硬写特定的值。它们被列在Table 9.35中。本例假定一个枚举类型被创建为：

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue',
'purple');
```

Table 9.35. 枚举支持函数

函数	描述 例子
<code>enum_first ( anyenum ) → anyenum</code>	返回输入枚举类型的第一个值。 <code>enum_first(null::rainbow) → red</code>
<code>enum_last ( anyenum ) → anyenum</code>	返回输入枚举类型的最后一个值。 <code>enum_last(null::rainbow) → purple</code>
<code>enum_range ( anyenum ) → anyarray</code>	将输入枚举类型的所有值作为一个有序的数组返回。 <code>enum_range(null::rainbow) → {red, orange, yellow, green, blue, purple}</code>
<code>enum_range ( anyenum, anyenum ) → anyarray</code>	以一个数组返回在给定两个枚举值之间的范围。值必须来自相同的枚举类型。如果第一个参数为空，其结果将从枚举类型的第一个值开始。如果第二参数为空，其结果将以枚举类型的最后一个值结束。 <code>enum_range('orange'::rainbow, 'green'::rainbow) → {orange, yellow, green}</code> <code>enum_range(NULL, 'green'::rainbow) → {red, orange, yellow, green}</code> <code>enum_range('orange'::rainbow, NULL) → {orange, yellow, green, blue, purple}</code>

请注意，除了双参数形式的`enum_range`外，这些函数忽略传递给它们的具体值，它们只关心声明的数据类型。空值或类型的一个特定值可以通过，并得到相同的结果。这些函数更多地被用于一个表列或函数参数，而不是一个硬写的类型名，如例子中所使用。

## 9.11. 几何函数和操作符

几何类型`point`、`box`、`lseg`、`line`、`path`、`polygon`和`circle`有一大堆本地支持函数和操作符，如Table 9.36 Table 9.3和Table 9.38中所示。

Table 9.36. 几何操作符

操作符	描述 例子
<code>geometric_type + point → geometric_type</code>	将第二个 <code>point</code> 的坐标添加到第一个参数的每个点的坐标中，从而执行翻译。适用于 <code>point</code> 、 <code>box</code> 、 <code>path</code> 、 <code>circle</code> 。 <code>box '(1, 1), (0, 0)' + point '(2, 0)' → (3, 1), (2, 0)</code>
<code>path + path → path</code>	连接两个打开的路径(如果其中一个路径是关闭的，则返回NULL)。 <code>path '[(0, 0), (1, 1)]' + path '[(2, 2), (3, 3), (4, 4)]' → [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]</code>

操作符	描述 例子
<code>geometric_type - point</code>	<code>→ geometric_type</code> 从第一个参数的每个点的坐标中减去第二个point的坐标，从而执行翻译。适用于 <code>point</code> 、 <code>box</code> 、 <code>path</code> 、 <code>circle</code> 。 <code>box '(1,1), (0,0)' - point '(2,0)' → (-1,1), (-2,0)</code>
<code>geometric_type * point</code>	<code>→ geometric_type</code> 将第一个参数的每个点乘以第二个point (将点视为由实部和虚部表示的复数，并执行标准的复数乘法)。如果将第二个point解释为向量，这等价于将对象的大小和到原点的距离按向量的长度缩放，并以向量与x轴的夹角绕原点逆时针旋转。适用于 <code>point</code> 、 <code>box</code> 、 <sup>a</sup> <code>path</code> 、 <code>circle</code> 。 <code>path '((0,0), (1,0), (1,1))' * point '(3.0,0)' → ((0,0), (3,0), (3,3))</code> <code>path '((0,0), (1,0), (1,1))' * point (cosd(45), sind(45)) → ((0,0), (0.7071067811865475, 0.7071067811865475), (0, 1.414213562373095))</code>
<code>geometric_type / point</code>	<code>→ geometric_type</code> 将第一个参数的每个点除以第二个point (将点视为由实部和虚部表示的复数，并执行标准的复数除法)。如果将第二个point解释为向量，这等价于将物体的大小和到原点的距离按向量的长度向下缩放，并以向量与x轴的夹角围绕原点顺时针旋转。适用于 <code>point</code> 、 <code>box</code> 、 <sup>a</sup> <code>path</code> 、 <code>circle</code> 。 <code>path '((0,0), (1,0), (1,1))' / point '(2.0,0)' → ((0,0), (0.5,0), (0.5,0.5))</code> <code>path '((0,0), (1,0), (1,1))' / point (cosd(45), sind(45)) → ((0,0), (0.7071067811865476, -0.7071067811865476), (1.4142135623730951, 0))</code>
<code>@-@ geometric_type</code>	<code>→ double precision</code> 计算总长度。适用于 <code>lseg</code> 、 <code>path</code> 。 <code>@-@ path '[(0,0), (1,0), (1,1)]' → 2</code>
<code>@@ geometric_type</code>	<code>→ point</code> 计算中心点。适用于 <code>box</code> 、 <code>lseg</code> 、 <code>path</code> 、 <code>polygon</code> 、 <code>circle</code> 。 <code>@@ box '(2,2), (0,0)' → (1,1)</code>
<code># geometric_type</code>	<code>→ integer</code> 返回点的数量。适用于 <code>path</code> 、 <code>polygon</code> 。 <code># path '((1,0), (0,1), (-1,0))' → 3</code>
<code>geometric_type # geometric_type</code>	<code>→ point</code> 计算交点，如果没有则为NULL。适用于 <code>lseg</code> 、 <code>line</code> 。 <code>lseg '[(0,0), (1,1)]' # lseg '[(1,0), (0,1)]' → (0.5, 0.5)</code>
<code>box # box</code>	<code>→ box</code> 计算两个方框的交集，如果没有则为NULL。 <code>box '(2,2), (-1,-1)' # box '(1,1), (-2,-2)' → (1,1), (-1,-1)</code>
<code>geometric_type ## geometric_type</code>	<code>→ point</code> 计算第二个对象上离第一个对象最近的点。适用于这些类型对：( <code>point</code> , <code>box</code> )、( <code>point</code> , <code>lseg</code> )、( <code>point</code> , <code>line</code> )、( <code>lseg</code> , <code>box</code> )、( <code>lseg</code> , <code>lseg</code> )、( <code>line</code> , <code>lseg</code> )。 <code>point '(0,0)' ## lseg '[(2,0), (0,2)]' → (1,1)</code>
<code>geometric_type &lt;-&gt; geometric_type</code>	<code>→ double precision</code> 计算对象之间的距离。适用于所有七种几何类型，适用于point与另一种几何类型的所有组合，以及这些额外的类型对：( <code>box</code> , <code>lseg</code> )、( <code>lseg</code> , <code>line</code> )、( <code>polygon</code> , <code>circle</code> ) (以及易子情况下)。 <code>circle '&lt;(0,0),1&gt;' &lt;-&gt; circle '&lt;(5,0),1&gt;' → 3</code>
<code>geometric_type @&gt; geometric_type</code>	<code>→ boolean</code> 第一个对象包含第二个对象吗？适用于这些类型对：( <code>box</code> , <code>point</code> )、( <code>box</code> , <code>box</code> )、( <code>path</code> , <code>point</code> )、( <code>polygon</code> , <code>point</code> )、( <code>polygon</code> , <code>polygon</code> )、( <code>circle</code> , <code>point</code> )、( <code>circle</code> , <code>circle</code> )。 <code>circle '&lt;(0,0),2&gt;' @&gt; point '(1,1)' → t</code>

操作符	描述 例子
<code>geometric_type &lt;@ geometric_type → boolean</code>	第一个对象包含在第二个对象之中还是在第二个对象之上? 适用于这些类型对: (point, box), (point, lseg), (point, line), (point, path), (point, polygon), (point, circle), (box, box), (lseg, box), (lseg, line), (polygon, polygon), (circle, circle). point '(1,1)' <@ circle '<(0,0),2>' → t
<code>geometric_type &amp;&amp; geometric_type → boolean</code>	这些对象有重叠吗?(一个共同点使之成真。) 适用于 box, polygon, circle. box '(1,1), (0,0)' && box '(2,2), (0,0)' → t
<code>geometric_type &lt;&lt; geometric_type → boolean</code>	第一个对象完全位于第二个对象的左边吗? 适用于 point, box, polygon, circle. circle '<(0,0),1>' << circle '<(5,0),1>' → t
<code>geometric_type &gt;&gt; geometric_type → boolean</code>	第一个对象完全位于第二个对象的右边吗? 适用于 point, box, polygon, circle. circle '<(5,0),1>' >> circle '<(0,0),1>' → t
<code>geometric_type &amp;&lt; geometric_type → boolean</code>	第一个对象没有延伸到第二个对象的右侧吗? 适用于 box, polygon, circle. box '(1,1), (0,0)' &< box '(2,2), (0,0)' → t
<code>geometric_type &amp;&gt; geometric_type → boolean</code>	第一个对象没有延伸到第二个对象的左侧吗? 适用于 box, polygon, circle. box '(3,3), (0,0)' &> box '(2,2), (0,0)' → t
<code>geometric_type &lt;&lt;  geometric_type → boolean</code>	第一个对象是否确定位于第二个对象下面? 适用于 box, polygon, circle. box '(3,3), (0,0)' <<  box '(5,5), (3,4)' → t
<code>geometric_type  &gt;&gt; geometric_type → boolean</code>	第一个对象是否确定位于第二个对象上面? 适用于 box, polygon, circle. box '(5,5), (3,4)'  >> box '(3,3), (0,0)' → t
<code>geometric_type &amp;&lt;  geometric_type → boolean</code>	第一个对象是否没有扩展到第二个对象上面? 适用于 box, polygon, circle. box '(1,1), (0,0)' &<  box '(2,2), (0,0)' → t
<code>geometric_type  &amp;&gt; geometric_type → boolean</code>	第一个对象是否没有扩展到第二个对象下面? 适用于 box, polygon, circle. box '(3,3), (0,0)'  &> box '(2,2), (0,0)' → t
<code>box &lt;^ box → boolean</code>	第一个对象是否位于第二个对象下面(允许边缘相切)? box '((1,1), (0,0))' <^ box '((2,2), (1,1))' → t
<code>point &lt;^ point → boolean</code>	第一个对象是否确定位于第二个对象下面? (这个操作符命名错误; 它应该是 << .) point '(1,0)' <^ point '(1,1)' → t
<code>box &gt;^ box → boolean</code>	第一个对象是否位于第二个对象上面(允许边缘相切)? box '((2,2), (1,1))' >^ box '((1,1), (0,0))' → t
<code>point &gt;^ point → boolean</code>	第一个对象是否确定位于第二个对象上面? (这个操作符命名错误; 它应该是  >>.) point '(1,1)' >^ point '(1,0)' → t

操作符	描述	例子
<code>geometric_type ?# geometric_type</code>	<code>→ boolean</code>	这些对象是否相交? 适用于这些类型对: (box, box), (lseg, box), (lseg, lseg), (lseg, line), (line, box), (line, line), (path, path). <code>lseg ' [(-1,0), (1,0)]' ?# box ' (2,2), (-2,-2)' → t</code>
<code>?- line</code>	<code>→ boolean</code>	线是水平的? <code>?- lseg ' [(-1,0), (1,0)]' → t</code>
<code>point ?- point</code>	<code>→ boolean</code>	点是否水平对齐(即具有相同的y坐标)? <code>point ' (1,0)' ?- point ' (0,0)' → t</code>
<code>?  line</code>	<code>→ boolean</code>	线是纵向的? <code>?  lseg ' [(-1,0), (1,0)]' → f</code>
<code>point ?  point</code>	<code>→ boolean</code>	点是否垂直对齐(即具有相同的x坐标)? <code>point ' (0,1)' ?  point ' (0,0)' → t</code>
<code>line ?-  line</code>	<code>→ boolean</code>	线是垂直的? <code>lseg ' [(0,0), (0,1)]' ?-  lseg ' [(0,0), (1,0)]' → t</code>
<code>line ?   line</code>	<code>→ boolean</code>	线是平行的? <code>lseg ' [(-1,0), (1,0)]' ?   lseg ' [(-1,2), (1,2)]' → t</code>
<code>geometric_type ~# geometric_type</code>	<code>→ boolean</code>	这些对象是相同的吗? 适用于 point, box, polygon, circle. <code>polygon ' ((0,0), (1,1))' ~# polygon ' ((1,1), (0,0))' → t</code>

<sup>a</sup> “Rotating” 用这些操作符“旋转”一个盒子，只会移动它的角点:这个盒子仍然被认为有平行于轴的边。因此，盒子的大小并没有像真正的旋转那样得到保留。

### Caution

请注意“same as”操作符(`~#`)，表示point、box、polygon和circle类型的一般相等概念。这些类型中的某些还有一个`=`操作符，但是`=`只比较相同的面积。其它的标量比较操作符(`<=`等等)，在这些类型可用的地方，同样比较区域。

### Note

在LightDB之前，包含操作符`@>`和`<@`被分别称为`^`和`@`。这些名字仍然可以使用，但是已被废除并且最终将被移除。

Table 9.37. 几何函数

函数	描述 例子
area (geometric_type) → double precision	计算面积。适用于 box, path, circle。path 输入必须封闭, 否则返回NULL。 同样, 如果 path 是自交叉的, 结果可能是没有意义的。 area(box '(2, 2), (0, 0)') → 4
center (geometric_type) → point	计算中心点。适用于 box, circle。 center(box '(1, 2), (0, 0)') → (0.5, 1)
diagonal (box) → lseg	提取框的对角线作为线段 (与 lseg(box) 相同)。 diagonal(box '(1, 2), (0, 0)') → [(1, 2), (0, 0)]
diameter (circle) → double precision	计算圆的直径。 diameter(circle '<(0, 0), 2>') → 4
height (box) → double precision	计算框的垂直尺寸。 height(box '(1, 2), (0, 0)') → 2
isclosed (path) → boolean	路径是否封闭? isclosed(path '((0, 0), (1, 1), (2, 0))') → t
isopen (path) → boolean	路径是否开放? isopen(path '[(0, 0), (1, 1), (2, 0)]') → t
length (geometric_type) → double precision	计算总长度。适用于 lseg, path。 length(path '((-1, 0), (1, 0))') → 4
npoints (geometric_type) → integer	返回点的数量。适用于 path, polygon。 npoints(path '[(0, 0), (1, 1), (2, 0)]') → 3
pclose (path) → path	将路径转换为封闭形式。 pclose(path '[(0, 0), (1, 1), (2, 0)]') → ((0, 0), (1, 1), (2, 0))
popen (path) → path	将路径转换为开放形式。 popen(path '((0, 0), (1, 1), (2, 0))') → [(0, 0), (1, 1), (2, 0)]
radius (circle) → double precision	计算圆的半径。 radius(circle '<(0, 0), 2>') → 2
slope (point, point) → double precision	计算通过两点所画直线的斜率。 slope(point '(0, 0)', point '(2, 1)') → 0.5
width (box) → double precision	计算框的水平大小。 width(box '(1, 2), (0, 0)') → 1

Table 9.38. 几何类型转换函数

函数	描述 例子
<code>box ( circle )</code>	<code>→ box</code> 计算框中内刻的圆形。 <code>box(circle '&lt;(0,0),2&gt;')</code> $\rightarrow$ (1.414213562373095, 1.414213562373095), (-1.414213562373095, -1.414213562373095)
<code>box ( point )</code>	<code>→ box</code> 将点转换为空框。 <code>box(point '(1,0)')</code> $\rightarrow$ (1,0), (1,0)
<code>box ( point, point )</code>	<code>→ box</code> 将任意两个角点转换为框。 <code>box(point '(0,1)', point '(1,0)')</code> $\rightarrow$ (1,1), (0,0)
<code>box ( polygon )</code>	<code>→ box</code> 计算多边形的边界框。 <code>box(polygon '((0,0), (1,1), (2,0))')</code> $\rightarrow$ (2,1), (0,0)
<code>bound_box ( box, box )</code>	<code>→ box</code> 计算两个方框的边界框。 <code>bound_box(box '(1,1), (0,0)', box '(4,4), (3,3)')</code> $\rightarrow$ (4,4), (0,0)
<code>circle ( box )</code>	<code>→ circle</code> 计算最小的圆形包围框。 <code>circle(box '(1,1), (0,0)')</code> $\rightarrow$ <(0.5,0.5),0.7071067811865476>
<code>circle ( point, double precision )</code>	<code>→ circle</code> 从圆心和半径构造圆。 <code>circle(point '(0,0)', 2.0)</code> $\rightarrow$ <(0,0),2>
<code>circle ( polygon )</code>	<code>→ circle</code> 将多边形转换为圆。圆心是多边形各点位置的平均值，半径是多边形各点到圆心的平均距离。 <code>circle(polygon '((0,0), (1,3), (2,0))')</code> $\rightarrow$ <(1,1),1.6094757082487299>
<code>line ( point, point )</code>	<code>→ line</code> 将两个点转换成通过它们的直线。 <code>line(point '(-1,0)', point '(1,0)')</code> $\rightarrow$ {0,-1,0}
<code>lseg ( box )</code>	<code>→ lseg</code> 提取框的对角线作为线段。 <code>lseg(box '(1,0), (-1,0)')</code> $\rightarrow$ [(1,0), (-1,0)]
<code>lseg ( point, point )</code>	<code>→ lseg</code> 从两个端点构造线段。 <code>lseg(point '(-1,0)', point '(1,0)')</code> $\rightarrow$ [(-1,0), (1,0)]
<code>path ( polygon )</code>	<code>→ path</code> 将多边形转换为具有点的相同列表的封闭路径。 <code>path(polygon '((0,0), (1,1), (2,0))')</code> $\rightarrow$ ((0,0), (1,1), (2,0))
<code>point ( double precision, double precision )</code>	<code>→ point</code> 从它的坐标构造点。 <code>point(23.4, -44.5)</code> $\rightarrow$ (23.4, -44.5)
<code>point ( box )</code>	<code>→ point</code> 计算框的中心。 <code>point(box '(1,0), (-1,0)')</code> $\rightarrow$ (0,0)
<code>point ( circle )</code>	<code>→ point</code> 计算圆心。 <code>point(circle '&lt;(0,0),2&gt;')</code> $\rightarrow$ (0,0)



函数	描述 例子
<code>point ( lseg )</code>	<code>→ point</code> 计算线段的中心。 <code>point(lseg ' [(-1, 0), (1, 0)]' ) → (0, 0)</code>
<code>point ( polygon )</code>	<code>→ point</code> 计算多边形的中心（多边形的点位置的平均值）。 <code>point(polygon '(0, 0), (1, 1), (2, 0)') → (1, 0.3333333333333333)</code>
<code>polygon ( box )</code>	<code>→ polygon</code> 将框转换为4点多边形。 <code>polygon(box '(1, 1), (0, 0)') → ((0, 0), (0, 1), (1, 1), (1, 0))</code>
<code>polygon ( circle )</code>	<code>→ polygon</code> 将圆转换为12点多边形。 <code>polygon(circle '&lt;(0, 0), 2&gt;') → ((-2, 0), (-1.7320508075688774, 0.9999999999999999), (-1.0000000000000002, 1.7320508075688772), (-1.2246063538223773e-16, 2), (0.9999999999999996, 1.7320508075688774), (1.732050807568877, 1.0000000000000007), (2, 2.4492127076447545e-16), (1.7320508075688776, -0.9999999999999994), (1.0000000000000009, -1.7320508075688767), (3.673819061467132e-16, -2), (-0.9999999999999987, -1.732050807568878), (-1.7320508075688767, -1.0000000000000009))</code>
<code>polygon ( integer, circle )</code>	<code>→ polygon</code> 将圆转换为n点多边形。 <code>polygon(4, circle '&lt;(3, 0), 1&gt;') → ((2, 0), (3, 1), (4, 1.2246063538223773e-16), (3, -1))</code>
<code>polygon ( path )</code>	<code>→ polygon</code> 将封闭路径转换为具有点的相同列表的多边形。 <code>polygon(path '((0, 0), (1, 1), (2, 0))') → ((0, 0), (1, 1), (2, 0))</code>

我们可以把一个point的两个组成数字当作具有索引 0 和 1 的数组访问。例如，如果t.p是一个point列，那么SELECT p[0] FROM t检索 X 座标而 UPDATE t SET p[1] = ... 改变 Y 座标。同样，box或者lseg类型的值可以当作两个point值的数组值看待。

## 9.12. 网络地址函数和操作符

IP网络地址类型，cidr和inet，支持Table 9. 所示的常用比较操作符，以及Table 9.39和Table 9.40所示的专用操作符和函数。

任何cidr 值都可以隐式地转换到inet；因此，下面在inet上操作的操作符和函数也可以在cidr值上工作。（对于inet和cidr有单独的函数，这是因为这两种情况的行为应该是不同的。）此外，它允许将inet值转换为cidr。当这样做的时候，子网掩码右边的任何位都被静默地置零，以创建一个有效的cidr值。

Table 9.39. IP地址操作符

操作符	描述 例子
<code>inet &lt;&lt; inet</code>	<code>→ boolean</code> 子网是否严格包含在子网中?这个操作符和后续的几个操作符测试子网包含情况。它们只考虑两个地址的网络部分(忽略网络掩码右侧的任何位)，并确定一个网络与另一个网络相同或者是相同的子网。 <code>inet '192.168.1.5' &lt;&lt; inet '192.168.1/24' → t</code>

操作符	描述 例子
	inet '192.168.0.5' << inet '192.168.1/24' → f inet '192.168.1/24' << inet '192.168.1/24' → f
inet <=< inet	→ boolean 子网是否包含或等于子网? inet '192.168.1/24' <=< inet '192.168.1/24' → t
inet >> inet	→ boolean 子网是否严格包含子网? inet '192.168.1/24' >> inet '192.168.1.5' → t
inet >>= inet	→ boolean 子网是否包含或等于子网? inet '192.168.1/24' >>= inet '192.168.1/24' → t
inet && inet	→ boolean 其中一个子网包含或等于另一个子网? inet '192.168.1/24' && inet '192.168.1.80/28' → t inet '192.168.1/24' && inet '192.168.2.0/28' → f
~ inet	→ inet 计算位 NOT. ~ inet '192.168.1.6' → 63.87.254.249
inet & inet	→ inet 计算位 AND. inet '192.168.1.6' & inet '0.0.0.255' → 0.0.0.6
inet   inet	→ inet 计算位 OR. inet '192.168.1.6'   inet '0.0.0.255' → 192.168.1.255
inet + bigint	→ inet 向地址添加偏移量。 inet '192.168.1.6' + 25 → 192.168.1.31
bigint + inet	→ inet 向地址添加偏移量。 200 + inet '::ffff:fff0:1' → ::ffff:255.240.0.201
inet - bigint	→ inet 从地址中减去偏移量。 inet '192.168.1.43' - 36 → 192.168.1.7
inet - inet	→ bigint 计算两个地址的差值。 inet '192.168.1.43' - inet '192.168.1.19' → 24 inet ':::1' - inet '::ffff:1' → -4294901760

Table 9.40. IP地址的函数

函数	描述 例子
abbrev ( inet )	→ text 创建缩略的文本显示格式。(结果与inet输出函数产生的结果相同;它只是在与显式转换为text的结果比较时才被“abbreviated”，由于历史原因，它永远不会抑制子网掩码部分。) abbrev (inet '10.1.0.0/32') → 10.1.0.0
abbrev ( cidr )	→ text

函数	描述 例子
	创建缩写的文本显示格式。(缩写包括在子网掩码的右侧删除所有零字节;更多的例子请见 Table 8.22 ) abbrev(cidr '10.1.0.0/16') → 10.1/16
broadcast ( inet ) → inet	为地址的网络计算广播地址。 broadcast(inet '192.168.1.5/24') → 192.168.1.255/24
family ( inet ) → integer	返回地址的系列: 4 对应 IPv4, 6 对应 IPv6。 family(inet '::1') → 6
host ( inet ) → text	返回IP地址文本, 忽略子网掩码。 host(inet '192.168.1.0/24') → 192.168.1.0
hostmask ( inet ) → inet	为地址的网络计算主机掩码。 hostmask(inet '192.168.23.20/30') → 0.0.0.3
inet_merge ( inet, inet ) → cidr	计算包含两个给定网络的最小网络。 inet_merge(inet '192.168.1.5/24', inet '192.168.2.5/24') → 192.168.0.0/22
inet_same_family ( inet, inet ) → boolean	测试地址是否属于同一IP族。 inet_same_family(inet '192.168.1.5/24', inet '::1') → f
masklen ( inet ) → integer	以比特位返回子网掩码长度。 masklen(inet '192.168.1.5/24') → 24
netmask ( inet ) → inet	为地址的网络计算网络掩码。 netmask(inet '192.168.1.5/24') → 255.255.255.0
network ( inet ) → cidr	返回地址的网络部分, 将子网掩码右边的部分归零。(这相当于将值转换为cidr。) network(inet '192.168.1.5/24') → 192.168.1.0/24
set_masklen ( inet, integer ) → inet	设置inet值的子网掩码长度。地址部分不改变。 set_masklen(inet '192.168.1.5/24', 16) → 192.168.1.5/16
set_masklen ( cidr, integer ) → cidr	设置cidr值的子网掩码长度。新子网掩码右侧的地址位设置为零。 set_masklen(cidr '192.168.1.0/24', 16) → 192.168.0.0/16
text ( inet ) → text	以文本形式返回未缩写的IP地址和子网掩码长度。(这与显式转换为text的结果相同。) text(inet '192.168.1.5') → 192.168.1.5/32

## Tip

abbrev、host和text函数主要用于为IP地址提供另一种显示格式。

MAC地址类型， `macaddr` 和 `macaddr8`，支持Table 9.4中所示的常用比较操作符以及Table 9.4中所示的特殊函数。此外，它们支持位逻辑操作符 `~`、`&` 和 `|` (NOT, AND 和 OR)，就像上面对IP地址所示的那样。

Table 9.41. MAC 地址函数

函数	描述 例子
<code>trunc ( macaddr ) → macaddr</code>	将地址的最后3个字节设置为零。其余的前缀可以与特定的制造商关联(使用LightDB中没有包含的数据)。 <code>trunc (macaddr '12:34:56:78:90:ab') → 12:34:56:00:00:00</code>
<code>trunc ( macaddr8 ) → macaddr8</code>	将地址的最后5个字节设置为零。其余的前缀可以与特定的制造商关联(使用LightDB中没有包含的数据)。 <code>trunc (macaddr8 '12:34:56:78:90:ab:cd:ef') → 12:34:56:00:00:00:00:00</code>
<code>macaddr8_set7bit ( macaddr8 ) → macaddr8</code>	将地址的第7位设置为1，创建所谓的modified EUI-64，用于包含在IPv6地址中。 <code>macaddr8_set7bit (macaddr8 '00:34:56:ab:cd:ef') → 02:34:56:ff:fe:ab:cd:ef</code>

## 9.13. 文本搜索函数和操作符

Table 9.42 Table 9.43and Table 9.44总结了为全文搜索提供的函数和操作符。LightDB的文本搜索功能的详细解释可参考Chapter 12

Table 9.42. 文本搜索操作符

操作符	描述 例子
<code>tsvector @@ tsquery → boolean</code> <code>tsquery @@ tsvector → boolean</code>	<code>tsvector</code> 匹配 <code>tsquery</code> 吗?(参数可以按任意顺序给出。) <code>to_tsvector('fat cats ate rats') @@ to_tsquery('cat &amp; rat') → t</code>
<code>text @@ tsquery → boolean</code>	隐式调用 <code>to_tsvector()</code> 后的文本字符串匹配 <code>tsquery</code> 么? <code>'fat cats ate rats' @@ to_tsquery('cat &amp; rat') → t</code>
<code>tsvector @@@ tsquery → boolean</code> <code>tsquery @@@ tsvector → boolean</code>	这是@@已弃用的同义词。 <code>to_tsvector('fat cats ate rats') @@@ to_tsquery('cat &amp; rat') → t</code>
<code>tsvector    tsvector → tsvector</code>	连接两个 <code>tsvector</code> 。如果两个输入都包含词素位置，则相应地调整第二个输入的位置。 <code>'a:1 b:2'::tsvector    'c:1 d:2 b:3'::tsvector → 'a':1 'b':2,5 'c':3 'd':4</code>
<code>tsquery &amp;&amp; tsquery → tsquery</code>	ANDs两个 <code>tsquery</code> 一起，生成一个匹配两个输入查询的匹配文档的查询。 <code>'fat   rat'::tsquery &amp;&amp; 'cat'::tsquery → ('fat'   'rat') &amp; 'cat'</code>
<code>tsquery    tsquery → tsquery</code>	ORs两个 <code>tsquery</code> 一起，生成一个匹配两个输入查询的匹配文档的查询。 <code>'fat   rat'::tsquery    'cat'::tsquery → 'fat'   'rat'   'cat'</code>

操作符	描述 例子
!!	tsquery → tsquery 否定tsquery, 生成一个与输入查询不匹配的匹配文档的查询。 !! 'cat'::tsquery → '!cat'
<->	tsquery → tsquery 构造一个短语查询, 如果两个输入查询在连续的词素上匹配, 该查询将进行匹配。 to_tsquery('fat') <-> to_tsquery('rat') → 'fat' <-> 'rat'
@>	tsquery → boolean 第一个tsquery包含了第二个吗?(这只考虑出现在一个查询中的所有词素是否出现在另一个查询中, 忽略了组合操作符。) 'cat'::tsquery @> 'cat & rat'::tsquery → f
<@	tsquery → boolean 第一个tsquery包含在第二个中吗?(这只考虑出现在一个查询中的所有词素是否出现在另一个查询中, 而忽略了组合操作符。) 'cat'::tsquery <@ 'cat & rat'::tsquery → t 'cat'::tsquery <@ '!cat & rat'::tsquery → t

除了这些专用操作符之外, Table 9.1 中所示的常用比较操作符也适用于tsvector和tsquery类型。它们对于文本搜索不是很有用, 但是允许使用。例如, 建在这些类型列上的唯一索引。

Table 9.43. 文本搜索函数

函数	描述 例子
array_to_tsvector	( text[] ) → tsvector 将词素数组转换为tsvector。给定的字符串按原样使用, 不做进一步处理。 array_to_tsvector( {fat, cat, rat}::text[] ) → 'cat' 'fat' 'rat'
get_current_ts_config	( ) → regconfig 返回当前默认文本搜索配置的OID(由 default_text_search_config 所设定的)。 get_current_ts_config() → english
length	( tsvector ) → integer 返回tsvector中的词位数。 length('fat:2,4 cat:3 rat:5A'::tsvector) → 3
numnode	( tsquery ) → integer 返回tsquery中词位和操作符的数目。 numnode('fat & rat'   cat'::tsquery) → 5
plainto_tsquery	( [ config regconfig, ] query text ) → tsquery 将文本转换为tsquery, 根据指定的或默认配置对单词进行标准化。字符串中的任何标点符号都会被忽略(它不决定查询操作符)。结果查询匹配文本中包含所有非停止词的文档。 plainto_tsquery('english', 'The Fat Rats') → 'fat' & 'rat'
phraseto_tsquery	( [ config regconfig, ] query text ) → tsquery 将文本转换为tsquery, 根据指定的或默认配置对单词进行标准化。字符串中的任何标点符号都会被忽略(它不决定查询操作符)。结果查询匹配包含文本中所有非停止词的短语。 phraseto_tsquery('english', 'The Fat Rats') → 'fat' <-> 'rat' phraseto_tsquery('english', 'The Cat and Rats') → 'cat' <2> 'rat'
websearch_to_tsquery	( [ config regconfig, ] query text ) → tsquery

函数	描述 例子
	将文本转换为tsquery，根据指定的或默认配置对单词进行标准化。引用的单词序列被转换为短语测试。“or”一词被理解为产生OR操作符，而破折号产生NOT操作符；其他标点符号被忽略。这类似于一些常见的网络搜索工具的行为。 websearch_to_tsquery('english', 'fat rat or cat dog') → 'fat' <-> 'rat'   'cat' & 'dog'
querytree ( tsquery ) → text	生成tsquery的可转位部分的表示。结果为空或仅为T表示不可索引查询。 querytree('foo & ! bar'::tsquery) → 'foo'
setweight ( vector tsvector, weight "char" ) → tsvector	将指定的weight赋给vector的每个元素。 setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A') → 'cat':3A 'fat':2A, 4A 'rat':5A
setweight ( vector tsvector, weight "char", lexemes text[] ) → tsvector	将指定的weight赋给列在lexemes中的vector元素。 setweight('fat:2,4 cat:3 rat:5,6B'::tsvector, 'A', '{cat,rat}') → 'cat':3A 'fat':2,4 'rat':5A,6A
strip ( tsvector ) → tsvector	从tsvector中移除位置和权重。 strip('fat:2,4 cat:3 rat:5A'::tsvector) → 'cat' 'fat' 'rat'
to_tsquery ( [ config regconfig, ] query text ) → tsquery	将文本转换为tsquery，根据指定的或默认配置对单词进行标准化。单词必须由有效的tsquery操作符组合。 to_tsquery('english', 'The & Fat & Rats') → 'fat' & 'rat'
to_tsvector ( [ config regconfig, ] document text ) → tsvector	将文本转换为tsvector，根据指定的或默认配置对单词进行标准化。结果中包含位置信息。 to_tsvector('english', 'The Fat Rats') → 'fat':2 'rat':3
to_tsvector ( [ config regconfig, ] document json ) → tsvector to_tsvector ( [ config regconfig, ] document jsonb ) → tsvector	将JSON文档中的每个字符串值转换为tsvector，根据指定的或默认配置对单词进行标准化。然后将结果按文档顺序连接起来以产生输出。位置信息就像在每对字符串值之间存在一个停止词一样生成。（注意，当输入为jsonb时，JSON对象的字段的“document order”取决于实现；请观察这些例子中的差异。） to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::json) → 'dog':5 'fat':2 'rat':3 to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::jsonb) → 'dog':1 'fat':4 'rat':5
json_to_tsvector ( [ config regconfig, ] document json, filter jsonb ) → tsvector jsonb_to_tsvector ( [ config regconfig, ] document jsonb, filter jsonb ) → tsvector	选择filter请求的JSON文档中的每个项，并将每个项转换为tsvector，根据指定的或默认配置对单词进行标准化。然后将结果按文档顺序连接起来以产生输出。位置信息就像在每对选定的项目之间存在一个停止词一样生成。（注意，当输入为jsonb时，JSON对象字段的“document order”取决于实现。）filter必须是一个jsonb数组，其中包含0个或多个关键字：“string”（包括所有字符串值），“numeric”（包括所有数值），“boolean”（包括所有布尔值），“key”（包括所有键），或“all”（包括以上所有关键字）。作为一种特殊情况，该filter也可以是这些关键字之一的简单JSON值。 json_to_tsvector('english', '{"a": "The Fat Rats", "b": 123}'::json, ['string', 'numeric']) → '123':5 'fat':2 'rat':3

函数	描述 例子
	<code>json_to_tsvector('english', '{"cat": "The Fat Rats", "dog": 123} '::json, 'all')</code> → <code>'123':9 'cat':1 'dog':7 'fat':4 'rat':5</code>
<code>ts_delete ( vector tsvector, lexeme text )</code>	→ <code>tsvector</code> 从vector中删除任何出现的给定lexeme。 <code>ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')</code> → <code>'cat':3 'rat':5A</code>
<code>ts_delete ( vector tsvector, lexemes text[] )</code>	→ <code>tsvector</code> 从vector中删除lexemes中出现的任何词位。 <code>ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat'])</code> → <code>'cat':3</code>
<code>ts_filter ( vector tsvector, weights "char"[] )</code>	→ <code>tsvector</code> 只从vector中选择具有给定weights的元素。 <code>ts_filter('fat:2,4 cat:3b,7c rat:5A'::tsvector, '{a,b}')</code> → <code>'cat':3B 'rat':5A</code>
<code>ts_headline ( [ config regconfig, ] document text, query tsquery [, options text ] )</code>	→ <code>text</code> 以缩写形式显示document中query的匹配项，该匹配项必须是原始文本，而不是tsvector。在匹配查询之前，文档中的单词将根据指定的或默认的配置进行规范化。Section 12.3.4 中讨论了该函数的使用，还描述了可用的options。 <code>ts_headline('The fat cat ate the rat.', 'cat')</code> → <code>The fat &lt;b&gt;cat&lt;/b&gt; ate the rat.</code>
<code>ts_headline ( [ config regconfig, ] document json, query tsquery [, options text ] )</code>	→ <code>text</code>
<code>ts_headline ( [ config regconfig, ] document jsonb, query tsquery [, options text ] )</code>	→ <code>text</code> 以缩写形式显示匹配JSONdocument中字符串值中的query。更多细节请参阅Section 12.3.4 <code>ts_headline('{"cat":"raining cats and dogs"} '::jsonb, 'cat')</code> → <code>{"cat": "raining &lt;b&gt;cats&lt;/b&gt; and dogs"}</code>
<code>ts_rank ( [ weights real[], ] vector tsvector, query tsquery [, normalization integer ] )</code>	→ <code>real</code> 计算一个分数，显示vector与query的匹配程度。详情请参见Section 12.3.3 <code>ts_rank(to_tsvector('raining cats and dogs'), 'cat')</code> → <code>0.06079271</code>
<code>ts_rank_cd ( [ weights real[], ] vector tsvector, query tsquery [, normalization integer ] )</code>	→ <code>real</code> 使用覆盖密度算法计算一个分数，显示vector与query的匹配程度。详情参见Section 12.3.3 <code>ts_rank_cd(to_tsvector('raining cats and dogs'), 'cat')</code> → <code>0.1</code>
<code>ts_rewrite ( query tsquery, target tsquery, substitute tsquery )</code>	→ <code>tsquery</code> 在query中使用 substitute替换出现的target。详情参见 Section 12.4.2.1。 <code>ts_rewrite('a &amp; b'::tsquery, 'a'::tsquery, 'foo bar'::tsquery)</code> → <code>'b' &amp; ('foo'   'bar')</code>
<code>ts_rewrite ( query tsquery, select text )</code>	→ <code>tsquery</code> 根据目标替换部分query，并替换通过执行SELECT命令获得的查询。详情参见Section 12.4.2.1 <code>SELECT ts_rewrite('a &amp; b'::tsquery, 'SELECT t,s FROM aliases')</code> → <code>'b' &amp; ('foo'   'bar')</code>
<code>tsquery_phrase ( query1 tsquery, query2 tsquery )</code>	→ <code>tsquery</code> 构造一个短语查询，在连续的词位上搜索query1和query2的匹配项(与<->操作符相同)。 <code>tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'))</code> → <code>'fat' &lt;-&gt; 'cat'</code>

函数	描述 例子												
<code>tsquery_phrase</code>	<code>tsquery_phrase ( query1 tsquery, query2 tsquery, distance integer ) → tsquery</code> 构造一个短语查询，用于搜索query1和query2的匹配项，这些匹配项恰好出现在distance词位之间。 <code>tsquery_phrase(to_tsquery(' fat'), to_tsquery(' cat'), 10) → ' fat' &lt;10&gt; ' cat'</code>												
<code>tsvector_to_array</code>	<code>tsvector_to_array ( tsvector ) → text[]</code> 将tsvector转换为词位的数组。 <code>tsvector_to_array(' fat:2,4 cat:3 rat:5A'::tsvector) → {cat, fat, rat}</code>												
<code>unnest</code>	<code>unnest ( tsvector ) → setof record ( lexeme text, positions smallint[], weights text )</code> 将tsvector展开为一组行，每个行对应一个词位。 <code>select * from unnest(' cat:3 fat:2,4 rat:5A'::tsvector) →</code>												
	<table border="1"> <thead> <tr> <th>lexeme</th> <th>positions</th> <th>weights</th> </tr> </thead> <tbody> <tr> <td>cat</td> <td>{3}</td> <td>{D}</td> </tr> <tr> <td>fat</td> <td>{2,4}</td> <td>{D,D}</td> </tr> <tr> <td>rat</td> <td>{5}</td> <td>{A}</td> </tr> </tbody> </table>	lexeme	positions	weights	cat	{3}	{D}	fat	{2,4}	{D,D}	rat	{5}	{A}
lexeme	positions	weights											
cat	{3}	{D}											
fat	{2,4}	{D,D}											
rat	{5}	{A}											

## Note

所有接受一个可选的regconfig参数的文本搜索函数在该参数被忽略时，使用由default\_text\_search\_config指定的配置。

Table 9.4中的函数被单独列出，因为它们通常不被用于日常的文本搜索操作。它们主要有助于开发和调试新的文本搜索配置。

Table 9.44. 文本搜索调试函数

函数	描述 例子
<code>ts_debug</code>	<code>ts_debug ( [ config regconfig, ] document text ) → setof record ( alias text, description text, token text, dictionaries regdictionary[], dictionary regdictionary, lexemes text[] )</code> 根据指定的或默认的文本搜索配置从document中提取和标准化标记，并返回关于每个标记是如何处理的信息。详情请参见Section 12.8.1 <code>ts_debug('english', 'The Brightest supernovaes') → (asciiword, "Word, all ASCII", The, {english_stem}, english_stem, {}) ...</code>
<code>ts_lexize</code>	<code>ts_lexize ( dict regdictionary, token text ) → text[]</code> 如果字典知道输入标记，则返回替换词位数组；如果字典知道标记，但它是停止词，则返回空数组；如果它不是已知词，则返回NULL。详情参见Section 12.8.3 <code>ts_lexize('english_stem', 'stars') → {star}</code>
<code>ts_parse</code>	<code>ts_parse ( parser_name text, document text ) → setof record ( tokid integer, token text )</code> 使用命名的解析器从document中提取标记。详情参见Section 12.8.2 <code>ts_parse('default', 'foo - bar') → (1,foo) ...</code>
<code>ts_parse</code>	<code>ts_parse ( parser_oid oid, document text ) → setof record ( tokid integer, token text )</code>



函数	描述 例子
	使用OID指定的解析器从document中提取标记。 详情请参见 Section 12.8.2。 <code>ts_parse(3722, 'foo - bar') → (1, foo) ...</code>
	<code>ts_token_type ( parser_name text ) → setof record ( tokid integer, alias text, description text )</code> 返回一个表，该表描述命名解析器可以识别的每种类型的标记。详情请参见Section 12.8.2 <code>ts_token_type('default') → (1, asciiword, "Word, all ASCII") ...</code>
	<code>ts_token_type ( parser_oid oid ) → setof record ( tokid integer, alias text, description text )</code> 返回一个表，该表描述OID指定的解析器可以识别的每种标记类型。详情请参见Section 12.8.2 <code>ts_token_type(3722) → (1, asciiword, "Word, all ASCII") ...</code>
	<code>ts_stat ( sqlquery text [, weights text ] ) → setof record ( word text, ndoc integer, nentry integer )</code> 执行sqlquery，该查询必须返回单个tsvector列，并返回关于数据中包含的每个不同词位的统计信息。 详情请参见Section 12.4.4 <code>ts_stat('SELECT vector FROM apod') → (foo, 10, 15) ...</code>

## 9.14. UUID 函数

LightDB 包含一个函数来生成UUID:

```
gen_random_uuid () → uuid
```

此函数返回一个版本4(随机)的UUID。这是最常用的UUID类型，适用于大多数应用程序。

uuid-osspl模块提供了额外的功能，用于实现生成UUIDs的其他标准算法。

LightDB还为UUIDs提供了Table 9. 中所示的常用比较操作符。

## 9.15. XML 函数

本节中描述的函数以及类函数的表达式都在类型xml的值上操作。类型xml的详细信息请参见Section 8.13 用于在值和类型xml之间转换的类函数的表达式xmlparse和xmlserialize记录在这里，而不是在本节中。

使用大部分这些函数要求LightDB使用了configure --with-libxml进行编译。

### 9.15.1. 产生 XML 内容

有一组函数和类函数的表达式可以用来从 SQL 数据产生 XML 内容。它们特别适合于将查询结果格式化成 XML 文档以便于在客户端应用中处理。

#### 9.15.1.1. xmlcomment

```
xmlcomment ( text ) → xml
```

函数xmlcomment创建了一个 XML 值，它包含一个使用指定文本作为内容的 XML 注释。该文本不包含“—”或者也不会以一个“-”结尾，否则该结果的结构不是一个合法的 XML 注释。如果参数为空，结果也为空。

例子:

```
SELECT xmlcomment('hello');
```

```
xmlcomment
-----
<!--hello-->
```

### 9.15.1.2. xmlconcat

```
xmlconcat ( xml [, ...] ) → xml
```

函数xmlconcat将由单个 XML 值组成的列表串接成一个单独的值，这个值包含一个 XML 内容片断。空值会被忽略，只有当没有参数为非空时结果才为空。

例子:

```
SELECT xmlconcat(' <abc/>', ' <bar>foo</bar>');
```

```
xmlconcat
-----
<abc/><bar>foo</bar>
```

如果 XML 声明存在，它们会按照下面的方式被组合。如果所有的参数值都有相同的 XML 版本声明，该版本将被用在结果中，否则将不使用版本。如果所有参数值有独立声明值“yes”，那么该值将被用在结果中。如果所有参数值都有一个独立声明值并且至少有一个为“no”，则“no”被用在结果中。否则结果中将有独立声明。如果结果被决定要要求一个独立声明但是没有版本声明，将会使用一个版本 1.0 的版本声明，因为 XML 要求一个 XML 声明要包含一个版本声明。编码声明会被忽略并且在所有情况中都会被移除。

例子:

```
SELECT xmlconcat(' <?xml version="1.1"?><foo/>', ' <?xml version="1.1"
standalone="no"?><bar/>');
```

```
xmlconcat
-----
<?xml version="1.1"?><foo/><bar/>
```

### 9.15.1.3. xmlelement

```
xmlelement ( NAME name [, XMLATTRIBUTES ( attvalue [ AS attname ] [, ...] ) ]
[, content [, ...]] ) → xml
```

表达式xmlelement使用给定名称、属性和内容产生一个 XML 元素。语法中显示的name和attname项是简单的标识符，而不是值。 attvalue和content项是表达式，它们可以生成任何LightDB数据类型。 XMLATTRIBUTES的参数生成XML元素的属性；将content值连接起来形成其内容。

例子:

```
SELECT xmlelement(name foo);
```

```
xmlelement
```

```
-----
<foo/>
```

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar));
```

```
xmlelement
```

```
-----
<foo bar="xyz"/>
```

```
SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');
```

```
xmlelement
```

```
-----
<foo bar="2007-01-26">content</foo>
```

不是合法的 XML 名字的元素名和属性名将被逃逸，逃逸的方法是将违反的字符用序列 `_xHHHH_` 替换，其中 `HHHH` 是被替换字符的 Unicode 代码点的十六进制表示。例如：

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));
```

```
xmlelement
```

```
-----
<foo_x0024_bar a_x0026_b="xyz"/>
```

如果属性值是一个列引用，则不需要指定一个显式的属性名，在这种情况下列的名字将被默认用于属性的名字。在其他情况下，属性必须被给定一个显式名称。因此这个例子是合法的：

```
CREATE TABLE test (a xml, b xml);
```

```
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

但是下面这些不合法：

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
```

```
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

如果指定了元素内容，它们将被根据其数据类型格式化。如果内容本身也是类型 `xml`，就可以构建复杂的 XML 文档。例如：

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                  xmlelement(name abc),
                  xmlcomment('test'),
                  xmlelement(name xyz));
```

```
xmlelement
```

```
-----
<foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

其他类型的内容将被格式化为合法的 XML 字符数据。这意味着字符 `<`、`>`、`&` 和 `&` 将被转换为实体。二进制数据（数据类型 `bytea`）将被表示成 `base64` 或十六进制编码，具体取决于配置参数 `xmlbinary` 的设置。为了使 `LightDB` 的映射与 `SQL:2006` 及以后的 `SQL:2006` 中指定的映射保持一致，个别数据类型的特殊行为将不断发展，正如 `Section F.3.1` 中讨论的那样。

### 9.15.1.4. xmlforest

```
xmlforest ( content [ AS name ] [, ...] ) → xml
```

表达式xmlforest使用给定名称和内容产生一个元素的 XML 森林（序列）。对于xmlelement，每个name都必须是一个简单的标识符，而content表达式可以有任何数据类型。

例子：

```
SELECT xmlforest('abc' AS foo, 123 AS bar);
```

```
xmlforest
```

```
-----
<foo>abc</foo><bar>123</bar>
```

```
SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';
```

```
xmlforest
```

```
-----
<table_name>pg_authid</table_name><column_name>rolname</column_name>
<table_name>pg_authid</table_name><column_name>rolsuper</column_name>
...
```

如我们在第二个例子中所见，如果内容值是一个列引用，元素名称可以被忽略，这种情况下默认使用列名。否则，必须指定一个名字。

如上文xmlelement所示，非法 XML 名字的元素名会被逃逸。相似地，内容数据也会被逃逸来产生合法的 XML 内容，除非它已经是一个xml类型。

注意如果 XML 森林由多于一个元素组成，那么它不是合法的 XML 文档，因此在xmlelement中包装xmlforest表达式会有用处。

### 9.15.1.5. xmlpi

```
xmlpi ( NAME name [, content ] ) → xml
```

表达式xmlpi创建一个 XML 处理指令。对于xmlelement，name必须是一个简单的标识符，而content表达式可以有任何数据类型。如果存在，content不能包含字符序列?>。

例子：

```
SELECT xmlpi(name php, 'echo "hello world";');
```

```
xmlpi
```

```
-----
<?php echo "hello world";?>
```

### 9.15.1.6. xmlroot

```
xmlroot ( xml, VERSION {text|NO VALUE} [, STANDALONE {YES|NO|NO
VALUE} ] ) → xml
```

表达式xmlroot修改一个 XML 值的根结点的属性。如果指定了一个版本，它会替换根节点的值；如果指定了一个独立设置，它会替换根节点的独立声明中的值。

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
              version '1.0', standalone yes);

          xmlroot
-----
<?xml version="1.0" standalone="yes"?>
<content>abc</content>
```

### 9.15.1.7. xmlagg

```
xmlagg ( xml ) → xml
```

和这里描述的其他函数不同，函数xmlagg是一个聚集函数。它将聚集函数调用的输入值串接起来，非常像xmlconcat所做的事情，除了串接是跨行发生的而不是在单一行的多个表达式上发生。聚集表达式的更多信息请见Section 9.21

例子：

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
          xmlagg
-----
<foo>abc</foo><bar/>
```

为了决定串接的顺序，可以为聚集调用增加一个ORDER BY子句，如Section 4.2.4中所述。例如：

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
          xmlagg
-----
<bar/><foo>abc</foo>
```

我们推荐在以前的版本中使用下列非标准方法，并且它们在特定情况下仍然有用：

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
          xmlagg
-----
<bar/><foo>abc</foo>
```

以下案例演示与getclobval()一起使用的例子(仅限Oracle模式)：

```
SELECT xmlagg(xmlparse(content '1,2,3,')).getclobval() FROM dual;
getclobval
-----
1,2,3,
(1 row)
```

## 9.15.2. XML 谓词

这一节描述的表达式检查xml值的属性。

### 9.15.2.1. IS DOCUMENT

`xml IS DOCUMENT` → boolean

如果参数 XML 值是一个正确的 XML 文档，则IS DOCUMENT返回真，如果不是则返回假（即它是一个内容片断），或者是参数为空时返回空。文档和内容片断之间的区别请见Section 8.13

### 9.15.2.2. IS NOT DOCUMENT

`xml IS NOT DOCUMENT` → boolean

如果参数中的XML值是一个正确的XML文档，那么表达式IS NOT DOCUMENT返回假，否则返回真（也就是说它是一个内容片段），如果参数为空则返回空。

### 9.15.2.3. XMLEXISTS

`XMLEXISTS ( text PASSING [BY {REF|VALUE}] xml [BY {REF|VALUE}] )` → boolean

函数xmlexists评价一个XPath 1.0表达式(第一个参数)，以传递的XML值作为其上下文项。如果评价的结果产生一个空节点集，该函数返回false，如果产生任何其他值，则返回true。如果任何参数为空，则函数返回null。作为上下文项传递的非空值必须是一个XML文档，而不是内容片段或任何非XML值。

例子：

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY VALUE
'<towns><town>Toronto</town><town>Ottawa</town></towns>');
```

```
xmlexists
-----
t
(1 row)
```

BY REF和BY VALUE子句在LightDB中被接受，但在Section F.3.2中被忽略。

在SQL标准中，xmlexists函数评估XML查询语言中的表达式，但LightDB只允许使用XPath 1.0表达式，在Section F.3.2中讨论过。

### 9.15.2.4. xml\_is\_well\_formed

`xml_is_well_formed ( text )` → boolean

`xml_is_well_formed_document ( text )` → boolean

`xml_is_well_formed_content ( text )` → boolean

这些函数检查一个text串是不是一个良构的 XML，返回一个布尔结果。xml\_is\_well\_formed\_document检查一个良构的文档，而xml\_is\_well\_formed\_content检查良构的内容。如果xmloption配置参数被设置为DOCUMENT，xml\_is\_well\_formed会做第一个函数的工作；如果配置参数被设置

为CONTENT，`xml_is_well_formed`会做第二个函数的工作。这意味着`xml_is_well_formed`对于检查一个到类型xml的简单造型是否会成功非常有用，而其他两个函数对于检查XMLPARSE的对应变体是否会成功有用。

例子：

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
xml_is_well_formed
```

```
-----
f
(1 row)
```

```
SELECT xml_is_well_formed('<abc/>');
xml_is_well_formed
```

```
-----
t
(1 row)
```

```
SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
xml_is_well_formed
```

```
-----
t
(1 row)
```

```
SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</pg:foo>');
xml_is_well_formed_document
```

```
-----
t
(1 row)
```

```
SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</my:foo>');
xml_is_well_formed_document
```

```
-----
f
(1 row)
```

最后一个例子显示了这些检查也包括名字空间是否正确地匹配。

### 9.15.3. 处理 XML

要处理数据类型xml的值，LightDB 提供了函数`xpath`和`xpath_exists`，它们计算 XPath 1.0 表达式以及XMLTABLE表函数。

#### 9.15.3.1. xpath

```
xpath ( xpath text, xml xml [, nsarray text[] ] ) → xml[]
```

函数`xpath`根据 XML 值`xml`计算 XPath 1.0 表达式`xpath` (以文本形式给出)。它返回一个 XML 值的数组，该数组对应于该 XPath 表达式产生的结点集合。如果该 XPath 表达式返回一个标量值而不是一个结点集合，将会返回一个单一元素的数组。

第二个参数必须是一个良构的 XML 文档。特殊地，它必须有一个单一根结点元素。

该函数可选的第三个参数是一个名字空间映射的数组。这个数组应该是一个二维text数组，其第二轴长度等于2（即它应该是一个数组的数组，其中每一个都由刚好 2 个元素组成）。每个数组项的第一个元素是名字空间的名称（别名），第二个元素是名字空间的 URI。并不要求在这个数组中提供的别名和在 XML 文档本身中使用的那些名字空间相同（换句话说，在 XML 文档中和在xpath函数环境中，别名都是本地的）。

例子：

```
SELECT xpath('/my:a/text()', ' <my:a xmlns:my="http://example.com">test</my:a>',
            ARRAY[ARRAY['my', 'http://example.com']]);
```

```
  xpath
-----
 {test}
(1 row)
```

要处理默认（匿名）命名空间，做这样的事情：

```
SELECT xpath('//mydefns:b/text()', ' <a xmlns="http://example.com"><b>test</b></a>',
            ARRAY[ARRAY['mydefns', 'http://example.com']]);
```

```
  xpath
-----
 {test}
(1 row)
```

### 9.15.3.2. xpath\_exists

```
xpath_exists ( xpath text, xml xml [, nsarray text[] ] ) → boolean
```

函数xpath\_exists是xpath函数的一种特殊形式。这个函数不是返回满足 XPath 1.0 表达式的单一 XML 值，它返回一个布尔值表示查询是否被满足(具体来说，它是否产生了空节点集以外的任何值)。这个函数等价于标准的XMLEXISTS谓词，不过它还提供了对一个名字空间映射参数的支持。

例子：

```
SELECT xpath_exists('/my:a/text()', ' <my:a xmlns:my="http://example.com">test</my:a>',
            ARRAY[ARRAY['my', 'http://example.com']]);
```

```
  xpath_exists
-----
 t
(1 row)
```

### 9.15.3.3. xmltable

```
XMLTABLE (
  [ XMLNAMESPACES ( namespace_uri AS namespace_name [, ...] ), ]
  row_expression PASSING [BY {REF|VALUE}] document_expression [BY {REF|VALUE}]
  COLUMNS name { type [PATH column_expression] [DEFAULT default_expression]
  [NOT NULL | NULL]
```



```

        | FOR ORDINALITY }
    [, ...]
) → setof record

```

xmltable表达式基于给定的XML值产生一个表、一个抽取行的XPath过滤器以及一个列定义集合。虽然它在语法上类似于函数，但它只能作为一个表出现在查询的FROM子句中。

可选的XMLNAMESPACES子句是一个逗号分隔的名字空间定义列表。其中每个namespace\_uri是一个text表达式，每个namespace\_name是一个简单的标识符。它指定文档中使用的XML名字空间及其别名。当前不支持默认的名字空间说明。

所需的row\_expression参数是一个求值的XPath 1.0表达式(以text形式给出)，通过传递XML值document\_expression作为其上下文项，得到一组XML节点。这些节点就是xmltable转换为输出行的内容。如果document\_expression为空，或者row\_expression产生空节点集或节点集以外的任何值，则不会产生行。

document\_expression提供了上下文。row\_expression的项。它必须是一个格式良好的XML文档；不接受片段/森林。BY REF和BY VALUE子句如上文所讨论的那样，被接受但被忽略了，正如在Section F.3.2中所讨论的。

在SQL标准中，xmltable函数评估XML查询语言中的表达式。但LightDB只允许使用XPath 1.0的表达式，正如在Section F.3.1中所讨论的。

需要的COLUMNS子句指定将在输出表中生成的列。有关格式，请参阅上面的语法摘要。每个列都需要一个名称，作为一个数据类型(除非指定了FOR ORDINALITY，在这种情况下类型integer是隐式的)。路径、默认值以及为空性子句是可选的。

被标记为FOR ORDINALITY的列将按照从row\_expression的结果节点集中检索到的节点的顺序，从1开始，填充行号。最多只能有一个列被标记为FOR ORDINALITY。

### Note

XPath 1.0 并没有为节点集中的节点指定顺序，因此依赖特定结果顺序的代码将取决于实现。详情请参见 Section F.3.1.2

列的column\_expression是一个XPath 1.0表达式，它对每一行都要进行求值，并以row\_expression结果中的当前节点作为其上下文项，以找到列的值。如果没有给出column\_expression，那么列名被用作隐式路径。

如果一个列的XPath表达式返回一个非XML值(在XPath 1.0中仅限于string、boolean或double)，而该列的LightDB类型不是xml，那么该列将被设置为将值的字符串表示法分配给LightDB类型。(如果值是布尔值，如果输出列的类型类别是数字，那么它的字符串表示方式将被认为是1或0，否则true或false。)

如果一个列的XPath表达式返回一个非空的XML节点集，并且该列的LightDB类型是xml，那么如果该列是文档或内容形式的，那么该列将被精确地分配表达式结果。<sup>2</sup>

分配给xml输出列的非XML结果会产生内容，一个带有结果字符串值的单个文本节点。分配给任何其他类型的列的XML结果不能有一个以上的节点，否则会产生错误。如果正好有一个节点，则该列将被设置为将该节点的字符串值(如XPath 1.0 string函数定义的那样)分配给LightDB类型。

一个XML元素的字符串值是字符串值的协整，按文档的顺序。该元素中包含的所有文本节点及其子节点。字符串元素的值是一个没有下级文本节点的元素的值是一个空字符串(不

<sup>2</sup> 在顶层包含一个以上的元素节点的结果，或者在元素之外的非空格文本，就是内容形式的一个例子。一个XPath结果可以是这两种形式的，例如，如果它返回的是一个从包含它的元素中选择的属性节点。这样的结果将被放到内容形式中，每个不允许的节点都会被替换为它的字符串值，就像XPath 1.0string函数定义的那样。

是NULL)。任何xsi:nil属性都会被忽略。请注意，两个非文本之间的text()节点只用空格，而两个非文本元素，并且保留了text()上的前导白格。节点不被扁平化。XPath 1.0中的string函数可以参考XPath 1.0中的定义其他XML节点类型和非XML值的字符串值的规则。

这里介绍的转换规则并不完全是SQL标准中的转换规则，如Section F.3.1.8讨论的那样。

如果路径表达式为给定行返回一个空节点集（通常情况下，当它不匹配时），该列将被设置为NULL，除非指定了default\_expression；然后使用评价该表达式产生的值。

default\_expression，而不是在调用xmltable时立即被评价，而是在每次需要列的默认值时，都会被评价。如果表达式符合稳定或不可更改的条件，则可以跳过重复评价。这意味着，你可以在default\_expression中使用像nextval这样的不稳定函数。

列可能会被标记为NOT NULL。如果一个NOT NULL列的column\_expression不匹配任何东西并且没有DEFAULT或者default\_expression也计算为空，则会报告一个错误。

例子：

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
FROM xmldata,
XMLTABLE(' //ROWS/ROW'
          PASSING data
          COLUMNS id int PATH '@id',
                   ordinality FOR ORDINALITY,
                   "COUNTRY_NAME" text,
                   country_id text PATH 'COUNTRY_ID',
                   size_sq_km float PATH 'SIZE[@unit = "sq_km"]',
                   size_other text PATH
                     'concat(SIZE[@unit!="sq_km"], " ", SIZE[@unit!
="sq_km"]/@unit)',
                   premier_name text PATH 'PREMIER_NAME' DEFAULT 'not
specified');

id | ordinality | COUNTRY_NAME | country_id | size_sq_km | size_other |
-----+-----+-----+-----+-----+-----

```

1   specified	1   Australia	AU			not
5   Shinzo Abe	2   Japan	JP		145935 sq_mi	
6   specified	3   Singapore	SG		697	not

接下来的例子展示了多个text()节点的串接、列名用作XPath过滤器的用法以及对空格、XML注释和处理指令的处理：

```
CREATE TABLE xmlelements AS SELECT
xml $$
  <root>
    <element> Hello<!-- xyxyz -->2a2<?aaaaa?> <!--x--> bbb<x>xxx</x>CC </
element>
  </root>
$$ AS data;
```

```
SELECT xmltable.*
FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS element text);
      element
```

```
-----
Hello2a2  bbbxxxCC
```

下面的例子展示了如何使用XMLNAMESPACES子句指定用在XML文档以及XPath表达式中的名字空间列表：

```
WITH xmldata(data) AS (VALUES (
<example xmlns="http://example.com/myns" xmlns:B="http://example.com/b">
  <item foo="1" B:bar="2"/>
  <item foo="3" B:bar="4"/>
  <item foo="4" B:bar="5"/>
</example>'::xml)
)
SELECT xmltable.*
FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,
                             'http://example.com/b' AS "B"),
              '/x:example/x:item'
              PASSING (SELECT data FROM xmldata)
              COLUMNS foo int PATH '@foo',
                       bar int PATH '@B:bar');
```

```
foo | bar
```

```
-----+-----
  1 | 2
  3 | 4
  4 | 5
```

```
(3 rows)
```

## 9.15.4. 将表映射到 XML

下面的函数将会把关系表的内容映射成 XML 值。它们可以被看成是 XML 导出功能：

```
table_to_xml ( table regclass, nulls boolean,
               tableforest boolean, targetns text ) → xml
query_to_xml ( query text, nulls boolean,
               tableforest boolean, targetns text ) → xml
```

```
cursor_to_xml ( cursor refcursor, count integer, nulls boolean,
               tableforest boolean, targetns text ) → xml
```

table\_to\_xml映射由参数table传递的命名表的内容。regclass类型接受使用常见标记标识表的字符串，包括可选的模式限定和双引号。query\_to\_xml执行由参数query传递的查询并且映射结果集。cursor\_to\_xml从cursor指定的游标中取出指定数量的行。如果需要映射一个大型的表，我们推荐这种变体，因为每一个函数都是在内存中构建结果值的。

如果tableforest为假，则结果的 XML 文档看起来像这样：

```
<tablename>
  <row>
    <columnname1>data</columnname1>
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

如果tableforest为真，结果是一个看起来像这样的 XML 内容片断：

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

如果没有表名可用，在映射一个查询或一个游标时，在第一种格式中使用串table，在第二种格式中使用row。

这几种格式的选择由用户决定。第一种格式是一个正确的 XML 文档，它在很多应用中都很重要。如果结果值要被重组为一个文档，第二种格式在cursor\_to\_xml函数中更有用。前文讨论的产生 XML 内容的函数（特别是xmlelement）可以被用来把结果修改成符合用户的要求。

数据值会被以前文的函数xmlelement中描述的相同方法映射。

参数nulls决定空值是否会被包含在输出中。如果为真，列中的空值被表示为：

```
<columnname xsi:nil="true"/>
```

其中xsi是 XML 模式实例的 XML 名字空间前缀。一个合适的名字空间声明将被加入到结果值中。如果为假，包含空值的列将从输出中忽略掉。

参数targetns指定想要的结果的 XML 名字空间。如果没有想要的特定名字空间，将会传递一个空串。

下面的函数返回 XML 模式文档，这些文档描述上述对应函数所执行的映射：

```
table_to_xmlschema ( table regclass, nulls boolean,
```

```

        tableforest boolean, targetns text ) → xml
query_to_xmlschema ( query text, nulls boolean,
        tableforest boolean, targetns text ) → xml
cursor_to_xmlschema ( cursor refcursor, nulls boolean,
        tableforest boolean, targetns text ) → xml

```

最重要的是相同的参数被传递来获得匹配的 XML 数据映射和 XML 模式文档。

下面的函数产生 XML 数据映射和对应的 XML 模式，并把产生的结果链接在一起放在一个文档（或森林）中。在要求自包含和自描述的结果是它们非常有用：

```

table_to_xml_and_xmlschema ( table regclass, nulls boolean,
        tableforest boolean, targetns text ) → xml
query_to_xml_and_xmlschema ( query text, nulls boolean,
        tableforest boolean, targetns text ) → xml

```

另外，下面的函数可用于产生相似的整个模式或整个当前数据库的映射：

```

schema_to_xml ( schema name, nulls boolean,
        tableforest boolean, targetns text ) → xml
schema_to_xmlschema ( schema name, nulls boolean,
        tableforest boolean, targetns text ) → xml
schema_to_xml_and_xmlschema ( schema name, nulls boolean,
        tableforest boolean, targetns text ) → xml

```

```

database_to_xml ( nulls boolean,
        tableforest boolean, targetns text ) → xml
database_to_xmlschema ( nulls boolean,
        tableforest boolean, targetns text ) → xml
database_to_xml_and_xmlschema ( nulls boolean,
        tableforest boolean, targetns text ) → xml

```

这些函数会忽略当前用户不可读的表。数据库范围的函数还会忽略当前用户没有USAGE（查找）权限的模式。

请注意，这可能会产生大量数据，这些数据需要在内存中构建。当请求大型模式或数据库的内容映射时，可能值得考虑单独映射表，甚至可能通过游标。

一个模式内容映射的结果看起来像这样：

```

<schemaname>
table1-mapping
table2-mapping
...
</schemaname>

```

其中一个表映射的格式取决于上文解释的tableforest参数。

一个数据库内容映射的结果看起来像这样：

```

<dbname>
<schemaname>

```

```

...
</schema1name>

<schema2name>
...
</schema2name>

...

</dbname>

```

其中的模式映射如上所述。

作为一个使用这些函数产生的输出的例子，Example 9. 展示了一个 XSLT 样式表，它将 table\_to\_xml\_and\_xmlschema 的输出转换为一个包含表数据的扁平转印的 HTML 文档。以一种相似的方式，这些函数的结果可以被转换成其他基于 XML 的格式。

### Example 9.1. 转换 SQL/XML 输出到 HTML 的 XSLT 样式表

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
    doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
    indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
      select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
      select="$schema/xsd:complexType[@name=$tabletypename]/
xsd:sequence/xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/
xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </template>

```

```

        </tr>
      </xsl:for-each>
    </table>
  </body>
</html>
</xsl:template>

</xsl:stylesheet>

```

## 9.16. JSON 函数和操作符

本节描述:

- 用于处理和创建JSON数据的函数和运算器
- SQL/JSON路径语言

要了解有关SQL/JSON标准的更多信息, 请参阅[sqltr-19075-6]。有关LightDB中支持的JSON类型的详细信息, 见 Section 8.14。

### 9.16.1. 处理和创建JSON数据

Table 9.4展示了可以用于 JSON 数据类型 (见Section 8.14 的操作符。此外, Table 9.所示的常用比较操作符也适用于 jsonb, 但不适用于 json。比较操作符遵循 Section 8.14. 中的B树操作概要的排序规则。

Table 9.45. json 和 jsonb 操作符

操作符	描述 例子
json -> integer -> json jsonb -> integer -> jsonb	提取JSON数组的第n个元素 (数组元素从0开始索引, 但负整数从末尾开始计数)。 ' [{"a": "foo"}, {"b": "bar"}, {"c": "baz"} ] ':: json -> 2 -> {"c": "baz"}' ' [{"a": "foo"}, {"b": "bar"}, {"c": "baz"} ] ':: json -> -3 -> {"a": "foo"}
json -> text -> json jsonb -> text -> jsonb	用给定的键提取JSON对象字段。 ' {"a": {"b": "foo"}} ':: json -> 'a' -> {"b": "foo"}
json ->> integer -> text jsonb ->> integer -> text	提取JSON数组的第n个元素, 作为text。 ' [1, 2, 3] ':: json ->> 2 -> 3
json ->> text -> text jsonb ->> text -> text	用给定的键提取JSON对象字段, 作为text。 ' {"a": 1, "b": 2} ':: json ->> 'b' -> 2
json #> text[] -> json jsonb #> text[] -> jsonb	提取指定路径下的JSON子对象, 路径元素可以是字段键或数组索引。 ' {"a": {"b": ["foo", "bar"]}} ':: json #> ' {a, b, 1}' -> "bar"
json #>> text[] -> text jsonb #>> text[] -> text	将指定路径上的JSON子对象提取为text。 ' {"a": {"b": ["foo", "bar"]}} ':: json #>> ' {a, b, 1}' -> bar

## Note

如果JSON输入没有匹配请求的正确结构，字段/元素/路径提取操作符返回NULL，而不是失败；例如，如果不存在这样的键或数组元素。

还有一些操作符仅适用于jsonb，如表Table 9.46所示。第Section 8.14.描述了如何使用这些操作符来有效地搜索索引的jsonb数据。

Table 9.46. 附加的sonb 操作符

操作符	描述 例子
jsonb @> jsonb → boolean	第一个JSON值是否包含第二个?(请参见Section 8.14.以了解包含的详细信息。) '{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb → t
jsonb <@ jsonb → boolean	第二个JSON中是否包含第一个JSON值? '{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb → t
jsonb ? text → boolean	文本字符串是否作为JSON值中的顶级键或数组元素存在? '{"a":1, "b":2}'::jsonb ? 'b' → t '["a", "b", "c"]'::jsonb ? 'b' → t
jsonb ?  text[] → boolean	文本数组中的字符串是否作为顶级键或数组元素存在? '{"a":1, "b":2, "c":3}'::jsonb ?  array['b', 'd'] → t
jsonb ?& text[] → boolean	文本数组中的所有字符串都作为顶级键或数组元素存在吗? '["a", "b", "c"]'::jsonb ?& array['a', 'b'] → t
jsonb    jsonb → jsonb	连接两个jsonb值。连接两个数组将生成一个包含每个输入的所有元素的数组。连接两个对象将生成一个包含它们键的并集的对象，当存在重复的键时取第二个对象的值。所有其他情况都是通过非数组输入转换为单个元素数组，然后按照两个数组的方式进行处理。不递归操作：只有顶级数组或对象结构被合并。 '["a", "b"]'::jsonb    '['a", "d"]'::jsonb → ["a", "b", "a", "d"] '{"a": "b"}'::jsonb    '{"c": "d"}'::jsonb → {"a": "b", "c": "d"} '[1, 2]'::jsonb    '3'::jsonb → [1, 2, 3] '{"a": "b"}'::jsonb    '42'::jsonb → [{"a": "b"}, 42] 将一个数组附加到另一个数组作为单个条目时，请将其包装在一个额外的数组层中，例如： '[1, 2]'::jsonb    jsonb_build_array(' [3, 4]'::jsonb) → [1, 2, [3, 4]]
jsonb - text → jsonb	从JSON对象中删除键(以及它的值)，或从JSON数组中删除匹配的字符串值。 '{"a": "b", "c": "d"}'::jsonb - 'a' → {"c": "d"} '["a", "b", "c", "b"]'::jsonb - 'b' → ["a", "c"]
jsonb - text[] → jsonb	从左操作数中删除所有匹配的键或数组元素。 '{"a": "b", "c": "d"}'::jsonb - '{a, c}'::text[] → {}
jsonb - integer → jsonb	删除具有指定索引的数组元素(负整数从末尾计数)。如果JSON值不是数组，则抛出错误。 '["a", "b"]'::jsonb - 1 → ["a"]
jsonb #- text[] → jsonb	



操作符	描述 例子
	删除指定路径上的字段或数组元素，路径元素可以是字段键或数组索引。 '["a", {"b":1}]'::jsonb #- '{1,b}' → ["a", {}]
jsonb @? jsonpath → boolean	JSON路径是否为指定的JSON值返回任何项? '{"a":[1,2,3,4,5]}'::jsonb @? '\$.a[*] ? (@ > 2)' → t
jsonb @@ jsonpath → boolean	返回指定JSON值的JSON路径谓词检查的结果。只考虑结果的第一项。如果结果不是布尔值，则返回NULL。 '{"a":[1,2,3,4,5]}'::jsonb @@ '\$.a[*] > 2' → t

## Note

jsonpath操作符@? 和 @@抑制以下错误:缺少对象字段或数组元素,意外的JSON项目类型,日期时间和数字错误。还可以告诉以下描述的与jsonpath相关的函数来抑制这些类型的错误。在搜索不同结构的JSON文档集合时,此行为可能会有所帮助。

Table 9.47显示可用于构造json和jsonb值的函数。

Table 9.47. JSON 创建函数

函数	描述 例子
to_json ( anyelement ) → json to_jsonb ( anyelement ) → jsonb	将任何SQL值转换为json或jsonb。数组和组合递归地转换为数组和对象(多维数组在JSON中变成数组的数组)。否则,如果存在从SQL数据类型到json的类型转换,则造型函数将用于执行转换; <sup>a</sup> 否则,将生成一个标量json值。对于除数字、布尔值或空值之外的任何标量,将使用文本表示,并根据需要进行转义,使其成为有效的JSON字符串值。 to_json('Fred said "Hi."'::text) → "Fred said \"Hi.\"" to_jsonb(row(42, 'Fred said "Hi."'::text)) → {"f1": 42, "f2": "Fred said \"Hi.\""}
array_to_json ( anyarray [, boolean ] ) → json	将SQL数组转换为JSON数组。该行为与to_json相同,只是如果可选boolean参数为真,换行符将在顶级数组元素之间添加。 array_to_json('{{1,5},{99,100}}'::int[]) → [[1,5],[99,100]]
row_to_json ( record [, boolean ] ) → json	将SQL组合值转换为JSON对象。该行为与to_json相同,只是如果可选boolean参数为真,换行符将在顶级元素之间添加。 row_to_json(row(1,'foo')) → {"f1":1,"f2":"foo"}
json_build_array ( VARIADIC "any" ) → json jsonb_build_array ( VARIADIC "any" ) → jsonb	根据可变参数列表构建可能异构类型的JSON数组。每个参数都按照to_json或to_jsonb进行转换。 json_build_array(1, 2, 'foo', 4, 5) → [1, 2, "foo", 4, 5]
json_build_object ( VARIADIC "any" ) → json jsonb_build_object ( VARIADIC "any" ) → jsonb	根据可变参数列表构建一个JSON对象。按照惯例,参数列表由交替的键和值组成。关键参数强制转换为文本;值参数按照to_json或to_jsonb进行转换。

函数	描述 例子
	<code>json_build_object('foo', 1, 2, row(3,'bar')) → {"foo" : 1, "2" : {"f1":3, "f2":"bar"}}</code>
	<code>json_object ( text[] ) → json</code> <code>jsonb_object ( text[] ) → jsonb</code> 从文本数组构建JSON对象。该数组必须有两个维度，一个维度的成员数为偶数，在这种情况下，它们被视为交替的键/值对；另一个维度的成员数为二维，每个内部数组恰好有两个元素，它们被视为键/值对。所有值都转换为JSON字符串。 <code>json_object(' {a, 1, b, "def", c, 3.5}')</code> → <code>{"a" : "1", "b" : "def", "c" : "3.5"}</code> <code>json_object(' {{a, 1}, {b, "def"}, {c, 3.5}}')</code> → <code>{"a" : "1", "b" : "def", "c" : "3.5"}</code>
	<code>json_object ( keys text[], values text[] ) → json</code> <code>jsonb_object ( keys text[], values text[] ) → jsonb</code> 这种形式的 <code>json_object</code> 从单独的文本数组中成对地获取键和值。否则，它与单参数形式相同。 <code>json_object(' {a,b}', ' {1,2}')</code> → <code>{"a": "1", "b": "2"}</code>

<sup>a</sup> 例如，`hstore`扩展有一个从`hstore`到`json`的转换，这样通过`json`创建函数转换的`hstore`值将被表示为`json`对象，而不是原始字符串值

Table 9.48显示可用于处理`json`和`jsonb`值的函数。

Table 9.48. JSON 处理函数

函数	描述 例子				
	<code>json_array_elements ( json ) → setof json</code> <code>jsonb_array_elements ( jsonb ) → setof jsonb</code> 将顶级JSON数组展开为一组JSON值。 <code>select * from json_array_elements(' [1,true, [2,false]]')</code> → <table style="margin-left: 40px;"> <thead> <tr> <th>value</th> </tr> </thead> <tbody> <tr><td>1</td></tr> <tr><td>true</td></tr> <tr><td>[2, false]</td></tr> </tbody> </table>	value	1	true	[2, false]
value					
1					
true					
[2, false]					
	<code>json_array_elements_text ( json ) → setof text</code> <code>jsonb_array_elements_text ( jsonb ) → setof text</code> 将顶级JSON数组展开为一组文本值。 <code>select * from json_array_elements_text(' ["foo", "bar"]')</code> → <table style="margin-left: 40px;"> <thead> <tr> <th>value</th> </tr> </thead> <tbody> <tr><td>foo</td></tr> <tr><td>bar</td></tr> </tbody> </table>	value	foo	bar	
value					
foo					
bar					
	<code>json_array_length ( json ) → integer</code> <code>jsonb_array_length ( jsonb ) → integer</code> 返回顶级JSON数组中的元素数量。 <code>json_array_length(' [1, 2, 3, {"f1":1, "f2":[5,6]}, 4]')</code> → 5 <code>jsonb_array_length(' []')</code> → 0				
	<code>json_each ( json ) → setof record ( key text, value json )</code>				

函数						
<p>描述 例子</p> <p><code>jsonb_each ( jsonb ) → setof record ( key text, value jsonb )</code> 将顶级JSON对象展开为一组键/值对。 <code>select * from json_each('{"a":"foo", "b":"bar"}') →</code></p> <table border="1"> <thead> <tr> <th>key</th> <th>value</th> </tr> </thead> <tbody> <tr> <td>a</td> <td>"foo"</td> </tr> <tr> <td>b</td> <td>"bar"</td> </tr> </tbody> </table>	key	value	a	"foo"	b	"bar"
key	value					
a	"foo"					
b	"bar"					
<p><code>json_each_text ( json ) → setof record ( key text, value text )</code> <code>jsonb_each_text ( jsonb ) → setof record ( key text, value text )</code> 将顶级JSON对象扩展为一组键/值对。返回的值的类型为文本。 <code>select * from json_each_text('{"a":"foo", "b":"bar"}') →</code></p> <table border="1"> <thead> <tr> <th>key</th> <th>value</th> </tr> </thead> <tbody> <tr> <td>a</td> <td>foo</td> </tr> <tr> <td>b</td> <td>bar</td> </tr> </tbody> </table>	key	value	a	foo	b	bar
key	value					
a	foo					
b	bar					
<p><code>json_extract_path ( from_json json, VARIADIC path_elems text[] ) → json</code> <code>jsonb_extract_path ( from_json jsonb, VARIADIC path_elems text[] ) → jsonb</code> 在指定路径下提取JSON子对象。(这在功能上相当于#&gt;操作符,但在某些情况下,将路径写成可变参数列表会更方便。) <code>json_extract_path('{"f2":{"f3":1}, "f4":{"f5":99, "f6":"foo"}}', 'f4', 'f6') → "foo"</code></p>						
<p><code>json_extract_path_text ( from_json json, VARIADIC path_elems text[] ) → text</code> <code>jsonb_extract_path_text ( from_json jsonb, VARIADIC path_elems text[] ) → text</code> 将指定路径上的JSON子对象提取为文本。(这在功能上等同于#&gt;&gt;操作符。) <code>json_extract_path_text('{"f2":{"f3":1}, "f4":{"f5":99, "f6":"foo"}}', 'f4', 'f6') → foo</code></p>						
<p><code>json_object_keys ( json ) → setof text</code> <code>jsonb_object_keys ( jsonb ) → setof text</code> 返回顶级JSON对象中的键集合。 <code>select * from json_object_keys('{"f1":"abc", "f2":{"f3":"a", "f4":"b"}}') →</code></p> <table border="1"> <thead> <tr> <th>json_object_keys</th> </tr> </thead> <tbody> <tr> <td>f1</td> </tr> <tr> <td>f2</td> </tr> </tbody> </table>	json_object_keys	f1	f2			
json_object_keys						
f1						
f2						
<p><code>json_populate_record ( base anyelement, from_json json ) → anyelement</code> <code>jsonb_populate_record ( base anyelement, from_json jsonb ) → anyelement</code> 将顶级JSON对象扩展为具有基本参数的复合类型的行。JSON对象将被扫描,查找名称与输出行类型的列名匹配的字段,并将它们的值插入到输出的这些列中。(不对任何输出列名的字段将被忽略。)在典型的使用中,基本的值仅为NULL,这意味着任何不匹配任何对象字段的输出列都将被填充为空。但是,如果base不为NULL,那么它包含的值将用于不匹配的列。 要将JSON值转换为输出列的SQL类型,需要按次序应用以下规则:</p> <ul style="list-style-type: none"> <li>• 在所有情况下,JSON空值都会转换为SQL空值。</li> <li>• 如果输出列的类型是json或jsonb,则会精确地重制JSON值。</li> </ul>						

函数	描述 例子										
	<ul style="list-style-type: none"> <li>• 如果输出列是复合(行)类型，且JSON值是JSON对象，则该对象的字段将转换为输出行类型的列，通过这些规则的递归应用程序。</li> <li>• 同样，如果输出列是数组类型，而JSON值是JSON数组，则通过这些规则的递归应用程序将JSON数组的元素转换为输出数组的元素。</li> <li>• 否则，如果JSON值是字符串，则将字符串的内容提供给输入转换函数，用以确定列的数据类型。</li> <li>• 否则，JSON值的普通文本表示将被提供给输入转换函数，以确定列的数据类型。</li> </ul> <p>虽然下面的示例使用一个常量JSON值，典型的用法是在查询的FROM子句中从另一个表侧面地引用json或jsonb列。在FROM子句中编写json_populate_record是一种很好的实践，因为提取的所有列都可以使用，而不需要重复的函数调用。</p> <pre>create type subrowtype as (d int, e text); create type myrowtype as (a int, b text[], c subrowtype); select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}, "x": "foo"}')</pre> <table border="1" data-bbox="432 869 821 958"> <thead> <tr> <th>a</th> <th>b</th> <th>c</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>{2, "a b"}</td> <td>(4, "a b c")</td> </tr> </tbody> </table>	a	b	c	1	{2, "a b"}	(4, "a b c")				
a	b	c									
1	{2, "a b"}	(4, "a b c")									
	<pre>json_populate_recordset ( base anyelement, from_json json ) → setof anyelement jsonb_populate_recordset ( base anyelement, from_json jsonb ) → setof anyelement</pre> <p>将对象的顶级JSON数组展开为一组具有基本参数的复合类型的行。对于json[b]_populate_record，将如上所述处理JSON数组的每个元素。</p> <pre>create type twoints as (a int, b int); select * from json_populate_recordset(null::twoints, '[{"a":1,"b":2}, {"a":3,"b":4}]')</pre> <table border="1" data-bbox="432 1323 528 1451"> <thead> <tr> <th>a</th> <th>b</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>2</td> </tr> <tr> <td>3</td> <td>4</td> </tr> </tbody> </table>	a	b	1	2	3	4				
a	b										
1	2										
3	4										
	<pre>json_to_record ( json ) → record jsonb_to_record ( jsonb ) → record</pre> <p>将顶级JSON对象展开为具有由 AS子句定义的复合类型的行。（与所有返回record的函数一样，调用查询必须使用AS子句显式定义记录的结构。）输出记录由JSON对象的字段填充，与上面描述的json[b]_populate_record的方式相同。由于没有输入记录值，不匹配的列总是用空值填充。</p> <pre>create type myrowtype as (a int, b text); select * from json_to_record('{"a":1,"b":[1,2,3],"c": [1,2,3],"e":"bar","r":{"a": 123, "b": "a b c"}}') as x(a int, b text, c int[], d text, r myrowtype)</pre> <table border="1" data-bbox="432 1839 1010 1937"> <thead> <tr> <th>a</th> <th>b</th> <th>c</th> <th>d</th> <th>r</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>[1, 2, 3]</td> <td>{1, 2, 3}</td> <td></td> <td>(123, "a b c")</td> </tr> </tbody> </table>	a	b	c	d	r	1	[1, 2, 3]	{1, 2, 3}		(123, "a b c")
a	b	c	d	r							
1	[1, 2, 3]	{1, 2, 3}		(123, "a b c")							
	<pre>json_to_recordset ( json ) → setof record jsonb_to_recordset ( jsonb ) → setof record</pre>										

函数	描述 例子
	<p>将顶级JSON对象数组展开为一组由AS子句定义的复合类型的行。（与所有返回record的函数一样，调用查询必须使用AS子句显式定义记录的结构。）对于json[b]_populate_record，将如上所述处理JSON数组的每个元素。</p> <pre>select * from json_to_recordset(' [{"a":1,"b":"foo"}, {"a":2,"c":"bar"}]') as x(a int, b text) →</pre> <pre> a   b ---+----- 1   foo 2  </pre>
	<pre>jsonb_set ( target jsonb, path text[], new_value jsonb [, create_if_missing boolean ] ) → jsonb</pre> <p>返回target，将path指定的项替换为new_value，如果create_if_missing为真（此为默认值）并且path指定的项不存在，则添加new_value。路径中的所有前面步骤都必须存在，否则将不加改变地返回target。与面向路径操作符一样，负整数出现在JSON数组末尾的path计数中。如果最后一个路径步骤是超出范围的数组索引，并且create_if_missing为真，那么如果索引为负，新值将添加到数组的开头，如果索引为正，则添加到数组的结尾。</p> <pre>jsonb_set(' [{"f1":1,"f2":null},2,null,3]', '{0,f1}', '[2,3,4]', false) → [{"f1": [2, 3, 4], "f2": null}, 2, null, 3] jsonb_set(' [{"f1":1,"f2":null},2]', '{0,f3}', '[2,3,4]') → [{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]</pre>
	<pre>jsonb_set_lax ( target jsonb, path text[], new_value jsonb [, create_if_missing boolean [, null_value_treatment text ] ] ) → jsonb</pre> <p>如果new_value不为NULL，则行为与jsonb_set完全一样。否则，根据null_value_treatment的值，它必须是'raise_exception'，'use_json_null'，'delete_key'，或'return_target'。默认值为'use_json_null'。</p> <pre>jsonb_set_lax(' [{"f1":1,"f2":null},2,null,3]', '{0,f1}', null) → [{"f1":null,"f2":null},2,null,3] jsonb_set_lax(' [{"f1":99,"f2":null},2]', '{0,f3}', null, true, 'return_target') → [{"f1": 99, "f2": null}, 2]</pre>
	<pre>jsonb_insert ( target jsonb, path text[], new_value jsonb [, insert_after boolean ] ) → jsonb</pre> <p>返回插入new_value的target。如果path指派的项是一个数组元素，如果insert_after为假（此为默认值），则new_value将被插入到该项之前，如果insert_after为真则在该项之后。如果由path指派的项是一个对象字段，则只在对象不包含该键时才插入 new_value。路径中的所有前面步骤都必须存在，否则将不加改变地返回target。与面向路径操作符一样，负整数出现在JSON数组末尾的 path计数中。如果最后一个路径步骤是超出范围的数组下标，则如果下标为负，则将新值添加到数组的开头；如果下标为正，则将新值添加到数组的结尾。</p> <pre>jsonb_insert(' {"a": [0,1,2]}', '{a, 1}', '"new_value"') → {"a": [0, "new_value", 1, 2]} jsonb_insert(' {"a": [0,1,2]}', '{a, 1}', '"new_value"', true) → {"a": [0, 1, "new_value", 2]}</pre>
	<pre>json_strip_nulls ( json ) → json jsonb_strip_nulls ( jsonb ) → jsonb</pre> <p>从给定的JSON值中删除所有具有空值的对象字段，递归地。非对象字段的空值是未受影响的。</p> <pre>json_strip_nulls(' [{"f1":1, "f2":null}, 2, null, 3]') → [{"f1":1},2,null,3]</pre>
	<pre>jsonb_path_exists ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → boolean</pre>

函数	描述 例子
	<p>检查JSON路径是否返回指定JSON值的任何项。如果指定了vars参数，则它必须是一个JSON对象，并且它的字段提供要替换到jsonpath表达式中的名称值。如果指定了silent参数并为true，函数会抑制与@? 和 @@运算符相同的错误。</p> <pre>jsonb_path_exists('{"a":[1,2,3,4,5]}', '\$.a[*] ? (@ &gt;= \$min &amp;&amp; @ &lt;= \$max)', '{"min":2, "max":4}') → t</pre>
	<pre>jsonb_path_match ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → boolean</pre> <p>返回指定JSON值的JSON路径谓词检查的结果。只有结果的第一项被考虑在内。如果结果不是布尔值，则返回NULL。可选的vars和silent参数的作用与jsonb_path_exists相同。</p> <pre>jsonb_path_match('{"a":[1,2,3,4,5]}', 'exists(\$.a[*] ? (@ &gt;= \$min &amp;&amp; @ &lt;= \$max))', '{"min":2, "max":4}') → t</pre>
	<pre>jsonb_path_query ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → setof jsonb</pre> <p>为指定的JSON值返回由JSON路径返回的所有JSON项。可选的vars和silent参数的作用与jsonb_path_exists相同。</p> <pre>select * from jsonb_path_query('{"a":[1,2,3,4,5]}', '\$.a[*] ? (@ &gt;= \$min &amp;&amp; @ &lt;= \$max)', '{"min":2, "max":4}') →</pre> <pre> jsonb_path_query ----- 2 3 4 </pre>
	<pre>jsonb_path_query_array ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → jsonb</pre> <p>以JSON数组的形式返回由JSON路径为指定的JSON值返回的所有JSON项。可选的vars和silent参数的作用与jsonb_path_exists相同。</p> <pre>jsonb_path_query_array('{"a":[1,2,3,4,5]}', '\$.a[*] ? (@ &gt;= \$min &amp;&amp; @ &lt;= \$max)', '{"min":2, "max":4}') → [2, 3, 4]</pre>
	<pre>jsonb_path_query_first ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → jsonb</pre> <p>为指定的JSON值返回由JSON路径返回的第一个JSON项。如果没有结果则返回NULL。可选的vars和silent参数的作用与jsonb_path_exists相同。</p> <pre>jsonb_path_query_first('{"a":[1,2,3,4,5]}', '\$.a[*] ? (@ &gt;= \$min &amp;&amp; @ &lt;= \$max)', '{"min":2, "max":4}') → 2</pre>
	<pre>jsonb_path_exists_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → boolean</pre> <pre>jsonb_path_match_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → boolean</pre> <pre>jsonb_path_query_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → setof jsonb</pre> <pre>jsonb_path_query_array_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → jsonb</pre> <pre>jsonb_path_query_first_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ] ] ) → jsonb</pre> <p>这些函数与上面描述的没有_tz后缀的对应函数类似，除了这些函数支持需要时区感知转换的日期/时间值比较之外。下面的示例需要将只包含日期的值2015-08-02解释为带有时区的时间戳，因此结果依赖于当前TimeZone设置。由于这种依赖性，这些函数被标记为稳定的，这意味着不能在索引中使用这些函数。它们的对应项是不可改变的，因此可以用于索引；但是，如果要求他们进行这样的比较，他们就会抛出错误。</p>

函数	描述 例子
	<code>jsonb_path_exists_tz('["2015-08-01 12:00:00 -05"]', '\$[*] ? (@.datetime() &lt; "2015-08-02".datetime())') → t</code>
	<code>jsonb_pretty ( jsonb ) → text</code> 将给定的JSON值转换为精美打印的，缩进的文本。 <code>jsonb_pretty(' [{"f1":1,"f2":null}, 2]') →</code>  <pre> [   {     "f1": 1,     "f2": null   },   2 ] </pre>
	<code>json_typeof ( json ) → text</code> <code>jsonb_typeof ( jsonb ) → text</code> 以文本字符串形式返回顶级JSON值的类型。可能的类型有object, array, string, number, boolean, 和 null。(null的结果不应该与SQL NULL 混淆;参见示例。) <code>json_typeof(' -123.4') → number</code> <code>json_typeof(' null'::json) → null</code> <code>json_typeof(NULL::json) IS NULL → t</code>

参见 Section 9.21 聚合函数 `json_agg` 将聚合记录值为JSON，聚合函数 `json_object_agg` 将聚合成对的值为JSON对象，以及它们在 `jsonb` 中的相当的(函数)，`jsonb_agg` 和 `jsonb_object_agg`。

## 9.16.2. SQL/JSON 路径语言

SQL/JSON路径表达式指定了要从JSON数据中检索的项目，类似于SQL访问XML时使用的XPath表达式。在LightDB中，路径表达式作为 `jsonpath` 数据类型实现，可以使用Section 8.14.5中描述的任何元素。

JSON查询函数和操作符将提供的路径表达式传递给 `path engine` 进行评估。如果表达式与被查询的JSON数据匹配，则返回相应的JSON项或项集。路径表达式是用SQL/JSON路径语言编写的，也可以包括算术表达式和函数。

路径表达式由 `jsonpath` 数据类型允许的元素序列组成。路径表达式通常从左向右求值，但你可以使用圆括号来更改操作的顺序。如果计算成功，将生成一系列JSON项，并将计算结果返回到JSON查询函数，该函数将完成指定的计算。

要引用正在查询的JSON值(context item项)，在路径表达式中使用 `$` 变量。它后面可以跟着一个或多个 `accessor operators`，这些操作符在JSON结构中逐级向下检索上下文项的子项。后面的每个操作符处理前一个求值步骤的结果。

例如，假设你有一些你想要解析的来自GPS跟踪器的JSON数据，例如：

```

{
  "track": {
    "segments": [
      {
        "location": [ 47.763, 13.4034 ],
        "start time": "2018-10-14 10:05:14",
        "HR": 73
      }
    ]
  }
}

```

```

    },
    {
      "location": [ 47.706, 13.2635 ],
      "start time": "2018-10-14 10:39:21",
      "HR": 135
    }
  ]
}
}

```

为了检索可用的轨迹段，你需要使用 `.key` 访问操作符来向下浏览周边的JSON对象：

```
$.track.segments
```

要检索数组的内容，通常使用 `[*]` 操作符。例如，下面的路径将返回所有可用轨道段的位置坐标：

```
$.track.segments[*].location
```

要只返回第一个段的坐标，可以在 `[]` 访问操作符中指定相应的下标。重新调用相对于0的JSON数组索引：

```
$.track.segments[0].location
```

每个路径求值步骤的结果可以由Section 9.16.2中列出的一个或多个jsonpath操作符和方法来处理。每个方法名之前必须有一个点。例如，你可以得到一个数组的大小：

```
$.track.segments.size()
```

在路径表达式中使用jsonpath操作符和方法的更多示例见下面Section 9.16.2.2

在定义路径时，还可以使用一个或多个与SQL中的WHERE子句类似的filter expressions。过滤器表达式以问号开头，并在圆括号中提供条件：

```
? (condition)
```

过滤表达式必须在它们应该应用的路径求值步骤之后写入。该步骤的结果将被筛选，以只包括满足所提供条件的那些项。SQL/JSON定义了三值逻辑，因此条件可以是 `true`、`false`、或 `unknown`。`unknown`值发挥与SQL `NULL`相同的角色，可以使用 `is unknown`谓词进行测试。进一步的路径求值步骤只使用筛选器表达式返回 `true`的那些项。

可以在过滤表达式中使用的函数和操作符罗列在Table 9.5中。在一个过滤表达式中，`@`变量表示被过滤的值(也就是说，前面路径步骤的一个结果)。你可以在 `@`后面写访问操作符来检索组件项。

例如，假设你想要检索所有高于130的心率值。你可以使用下面的表达式来实现这一点：

```
$.track.segments[*].HR ? (@ > 130)
```

为了获得具有这些值的片段的开始时间，必须在返回开始时间之前过滤掉不相关的片段，所以过滤表达式应用于上一步，条件中使用的路径不同：



```
$.track.segments[*] ? (@.HR > 130). "start time"
```

如果需要，可以按顺序使用几个过滤器表达式。例如，下面的表达式选择所有包含有相关坐标和高心率值的位置的段的开始时间：

```
$.track.segments[*] ? (@.location[1] < 13.4) ? (@.HR > 130). "start time"
```

也允许在不同嵌套层级中使用过滤器表达式。下面的例子首先根据位置筛选所有的片段，然后返回这些片段的高心率值，如果适用的话：

```
$.track.segments[*] ? (@.location[1] < 13.4).HR ? (@ > 130)
```

你也可以在彼此之间嵌套过滤器表达式：

```
$.track ? (exists(@.segments[*] ? (@.HR > 130))).segments.size()
```

如果包含任何具有高心率值的片段，则该表达式返回曲目的大小，否则返回空序列。

LightDB的SQL/JSON路径语言的实现与SQL/JSON标准有以下偏差：

- 路径表达式可以是布尔谓词，尽管SQL/JSON标准只允许在过滤器中使用谓词。这是实现@操作符所必需的。例如，下面的jsonpath表达式在LightDB中是有效的：

```
$.track.segments[*].HR < 70
```

- 在解释like\_regex过滤器中使用的正则表达式模式方面有一些小的差异，如Section 9.16.2.4所述。

### 9.16.2.1. 严格的(Strict) 和 不严格的(Lax) 模式

当查询JSON数据时，路径表达式可能与实际的JSON数据结构不匹配。试图访问不存在的对象成员或数组元素会导致结构错误。SQL/JSON路径表达式有两种处理结构错误的模式：

- 不严格的(lax) (默认) — 路径引擎隐式地将查询的数据适配到指定的路径。任何剩余的结构错误都将被抑制并转换为空SQL/JSON序列。
- 严格的(strict) — 如果发生了结构错误，则会引发错误。

如果JSON数据不符合期望的模式，不严格的(lax)模式有助于匹配JSON文档结构和路径表达式。如果操作不匹配特定操作的要求，可以自动将其包装为SQL/JSON数组，也可以在执行该操作之前将其元素转换为SQL/JSON序列来解包装。此外，比较操作符会自动以lax模式打开它们的操作数，因此你可以开包即用的就能比较SQL/JSON数组。大小为1的数组被认为等于它的唯一元素。只有在以下情况下才不会自动展开：

- 路径表达式包含type()或size()方法，它们分别返回数组中的元素类型和数量。
- 查询的JSON数据包含嵌套的数组。在本例中，只有最外层的数组被打开，而所有内部数组保持不变。因此，隐式展开在每个路径求值步骤中只能向下进行一级。

例如，当查询上面列出的GPS数据时，当使用不严格的(lax)模式时，你可以从它存储了一组片段的事实中抽象出来：

```
lax $.track.segments.location
```

在严格的(strict)模式中, 指定的路径必须与查询的JSON文档的结构完全匹配才能返回SQL/JSON项, 因此使用该路径表达式会导致错误。 要得到与不严格的(lax)模式相同的结果, 你必须显式地打开segments数组:

```
strict $.track.segments[*].location
```

使用宽松模式时, `.**` 访问器可能导致意外的结果。例如, 以下查询会将每个 HR 值选择两次:

```
lax $.**.HR
```

这是因为 `.` 访问器同时选择了 `segments` 数组及其每个元素, 而使用宽松模式时 `.HR` 访问器会自动展开数组。为避免产生意外的结果, 我们建议仅在严格模式下使用 `.` 访问器。以下查询仅选择每个 HR 值一次:

```
strict $.**.HR
```

### 9.16.2.2. SQL/JSON 路径操作符和方法

Table 9.4显示了jsonpath中可用的操作符和方法。请注意, 虽然一元操作符和方法可以应用于由前一个路径步骤产生的多个值, 二元操作符(加法等)只能应用于单个值。

Table 9.49. jsonpath 操作符和方法

操作符/方法 描述 例子
number + number → number 加法 jsonb_path_query(' [2]', '\$[0] + 3') → 5
+ number → number 一元加号(无操作);与加法不同, 这个可以迭代多个值 jsonb_path_query_array('{ "x": [2, 3, 4] }', '+ \$.x') → [2, 3, 4]
number - number → number 减法 jsonb_path_query(' [2]', '7 - \${0}') → 5
- number → number 否定;与减法不同, 它可以迭代多个值 jsonb_path_query_array('{ "x": [2, 3, 4] }', '- \$.x') → [-2, -3, -4]
number * number → number 乘法 jsonb_path_query(' [4]', '2 * \${0}') → 8
number / number → number 除法 jsonb_path_query(' [8.5]', '\${0} / 2') → 4.2500000000000000
number % number → number 模数(余数) jsonb_path_query(' [32]', '\${0} % 10') → 2
value . type() → string JSON项的类型(参见 json_typeof) jsonb_path_query_array(' [1, "2", {}]', '\$[*].type()') → ["number", "string", "object"]

操作符/方法	描述	例子
<code>value . size()</code>	→ number JSON项的大小(数组元素的数量, 如果不是数组则为1)	<code>jsonb_path_query('{"m": [11, 15]}', '\$.m.size()') → 2</code>
<code>value . double()</code>	→ number 从JSON数字或字符串转换过来的近似浮点数	<code>jsonb_path_query('{"len": "1.9"}', '\$.len.double() * 2') → 3.8</code>
<code>number . ceiling()</code>	→ number 大于或等于给定数字的最接近的整数	<code>jsonb_path_query('{"h": 1.3}', '\$.h.ceiling()') → 2</code>
<code>number . floor()</code>	→ number 小于或等于给定数字的最近整数	<code>jsonb_path_query('{"h": 1.7}', '\$.h.floor()') → 1</code>
<code>number . abs()</code>	→ number 给定数字的绝对值	<code>jsonb_path_query('{"z": -0.3}', '\$.z.abs()') → 0.3</code>
<code>string . datetime()</code>	→ datetime_type (see note) 从字符串转换过来的日期/时间值	<code>jsonb_path_query(["2015-8-1", "2015-08-12"], '\$[*] ? (@.datetime() &lt; "2015-08-2".datetime())') → "2015-8-1"</code>
<code>string . datetime(template)</code>	→ datetime_type (see note) 使用指定的to_timestamp模板从字符串转换过来的日期/时间值	<code>jsonb_path_query_array(["12:30", "18:40"], '\$[*].datetime("HH24:MI")') → ["12:30:00", "18:40:00"]</code>
<code>object . keyvalue()</code>	→ array 对象的键值对, 表示为包含三个字段的对象数组:"key", "value", 和"id";"id"是键值对所归属对象的唯一标识符	<code>jsonb_path_query_array('{"x": "20", "y": 32}', '\$.keyvalue()') → [{"id": 0, "key": "x", "value": "20"}, {"id": 0, "key": "y", "value": 32}]</code>

### Note

`datetime()` 和 `datetime(template)` 方法的结果类型可以是 `date`, `timetz`, `time`, `timestamptz`, 或 `timestamp`。这两个方法都动态地确定它们的结果类型。

`datetime()` 方法依次尝试将其输入字符串与 `date`, `timetz`, `time`, `timestamptz`, 和 `timestamp` 的 ISO 格式进行匹配。它在第一个匹配格式时停止, 并发出相应的数据类型。

`datetime(template)` 方法根据所提供的模板字符串中使用的字段确定结果类型。

`datetime()` 和 `datetime(template)` 方法使用与 `to_timestamp` SQL 函数相同的解析规则 (see Section 9.8, 但有三个例外。首先, 这些方法不允许不匹配的模板模式。其次, 模板字符串中只允许以下分隔符: 减号、句点、solidus (斜杠)、逗号、撇号、分号、冒号和空格。第三, 模板字符串中的分隔符必须与输入字符串完全匹配。

如果需要比较不同的日期/时间类型, 则应用隐式转换。`date` 值可以转换为 `timestamp` 或 `timestamptz`, `timestamp` 可以转换为 `timestamptz`,

time可以转换为timetz。但是，除了第一个转换外，其他所有转换都依赖于当前TimeZone设置，因此只能在时区感知的jsonpath函数中执行。

Table 9.5显示了适用的过滤器表达式元素。

Table 9.50. jsonpath 过滤器表达式元素

谓词/值	描述	例子
value == value → boolean	相等比较(这个，和其他比较操作符，适用于所有JSON标量值)	jsonb_path_query_array('[1, "a", 1, 3]', '\$[*] ? (@ == 1)') → [1, 1] jsonb_path_query_array('[1, "a", 1, 3]', '\$[*] ? (@ == "a")') → ["a"]
value != value → boolean value <> value → boolean	不相等比较	jsonb_path_query_array('[1, 2, 1, 3]', '\$[*] ? (@ != 1)') → [2, 3] jsonb_path_query_array('["a", "b", "c"]', '\$[*] ? (@ <> "b")') → ["a", "c"]
value < value → boolean	小于比较	jsonb_path_query_array('[1, 2, 3]', '\$[*] ? (@ < 2)') → [1]
value <= value → boolean	小于或等于比较	jsonb_path_query_array('["a", "b", "c"]', '\$[*] ? (@ <= "b")') → ["a", "b"]
value > value → boolean	大于比较	jsonb_path_query_array('[1, 2, 3]', '\$[*] ? (@ > 2)') → [3]
value >= value → boolean	大于或等于比较	jsonb_path_query_array('[1, 2, 3]', '\$[*] ? (@ >= 2)') → [2, 3]
true → boolean	JSON常数 真	jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '\$[*] ? (@.parent == true)') → {"name": "Chris", "parent": true}
false → boolean	JSON常数 假	jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '\$[*] ? (@.parent == false)') → {"name": "John", "parent": false}
null → value	JSON常数null(注意，与SQL不同，与null比较可以正常工作)	jsonb_path_query('[{"name": "Mary", "job": null}, {"name": "Michael", "job": "driver"}]', '\$[*] ? (@.job == null) .name') → "Mary"
boolean && boolean → boolean	布尔 AND	jsonb_path_query('[1, 3, 7]', '\$[*] ? (@ > 1 && @ < 5)') → 3
boolean    boolean → boolean	布尔 OR	jsonb_path_query('[1, 3, 7]', '\$[*] ? (@ < 1    @ > 5)') → 7

谓词/值	描述	例子
!	boolean → boolean 布尔 NOT	<code>jsonb_path_query('[1, 3, 7]', '\$[*] ? (!(@ &lt; 5))') → 7</code>
boolean is unknown	boolean → boolean 测试布尔条件是否为 unknown。	<code>jsonb_path_query('[-1, 2, 7, "foo"]', '\$[*] ? ((@ &gt; 0) is unknown)') → "foo"</code>
string like regex string	[ flag string ] → boolean 测试第一个操作数是否与第二个操作数给出的正则表达式匹配，可选使用由一串flag字符描述的修改(参见Section 9.16.2.3)。	<code>jsonb_path_query_array(['abc', "abd", "aBdC", "abdacb", "babc"], '\$[*] ? (@ like_regex "^ab.*c")') → ["abc", "abdacb"]</code> <code>jsonb_path_query_array(['abc', "abd", "aBdC", "abdacb", "babc"], '\$[*] ? (@ like_regex "^ab.*c" flag "i")') → ["abc", "aBdC", "abdacb"]</code>
string starts with string	boolean → boolean 测试第二个操作数是否为第一个操作数的初始子串。	<code>jsonb_path_query(['John Smith', "Mary Stone", "Bob Johnson"], '\$[*] ? (@ starts with "John")') → "John Smith"</code>
exists ( path_expression )	boolean → boolean 测试路径表达式是否至少匹配一个SQL/JSON项。如果路径表达式会导致错误，则返回unknown;第二个例子使用这个方法避免在严格模式下出现无此键(no-such-key)错误。	<code>jsonb_path_query({'x': [1, 2], "y": [2, 4]}, 'strict \$.* ? (exists (@ ? (@[*] &gt; 2)))') → [2, 4]</code> <code>jsonb_path_query_array({'value': 41}, 'strict \$ ? (exists (@.name)) .name') → []</code>

### 9.16.2.3. SQL/JSON 正则表达式

SQL/JSON路径表达式允许通过like\_regex过滤器将文本匹配为正则表达式。例如，下面的SQL/JSON路径查询将不区分大小写地匹配以英语元音开头的数组中的所有字符串：

```
$[*] ? (@ like_regex "[aeiou]" flag "i")
```

可选的flag字符串可以包括一个或多个字符i用于不区分大小写的匹配，m允许^和\$在换行时匹配，s允许.匹配换行符，q引用整个模式(将行为简化为一个简单的子字符串匹配)。

SQL/JSON标准借用了来自LIKE\_REGEX操作符的正则表达式定义，其使用了XQuery标准。LightDB目前不支持LIKE\_REGEX操作符。因此，like\_regex过滤器是使用Section 9.7.8描述的POSIX正则表达式引擎来实现的。这导致了与标准SQL/JSON行为的各种细微差异，这在Section 9.7.3.8进行了分类。但是请注意，这里描述的标志字母不兼容并不适用于SQL/JSON，因为它将XQuery标志字母翻译为符合POSIX引擎的预期。

请记住，like\_regex的模式参数是一个JSON路径字符串文字，根据Section 8.14.8给出的规则编写。这特别意味着在正则表达式中要使用的任何反斜杠都必须加倍。例如，匹配只包含数字的字符串：

```
$.* ? (@ like_regex "\\d+$")
```

## 9.17. 序列操作函数

本节描述对sequence objects进行操作的函数，也称为序列生成器或序列。序列对象是使用CREATE SEQUENCE创建的特殊单行表。序列对象通常用于为表中的行生成惟一标识符。在Table 9.5中列出的序列函数，提供了简单的、多用户安全方法，用于从序列对象中获取连续的序列值。

Table 9.51. 序列函数

函数	描述
<code>nextval ( regclass ) → bigint</code>	<p>将序列对象推进到下一个值并返回该值。这是自动完成的:即使多个会话并发地执行nextval，每个会话也会安全地接收到不同的序列值。如果序列对象是用默认形参创建的，则连续的nextval调用将返回以1开始的连续值。其他行为可以通过在CREATE SEQUENCE命令中使用适当的参数获得。这个函数需要USAGE或UPDATE特权在序列上。</p>
<code>setval ( regclass, bigint [, boolean ] ) → bigint</code>	<p>设置序列对象的当前值，以及可选的它的is_called标志。双参数形式将序列的last_value字段设置为指定的值，并将其is_called字段设置为true，意味着下一个nextval将在返回值之前推进序列。currval将报告的值也设置为指定的值。在三参数形式中，is_called可以设置为true 或 false。true与双参数形式具有相同的效果。如果设置为false，下一个nextval将返回指定的值，序列推进从下面的nextval开始。而且，currval报告的值在这种情况下不会改变。例如，</p> <pre>SELECT setval('myseq', 42);           Next nextval will return 43 SELECT setval('myseq', 42, true);     Same as above SELECT setval('myseq', 42, false);    Next nextval will return 42</pre> <p>setval返回的结果就是它的第二个参数的值。这个函数在序列上需要UPDATE特权。</p>
<code>currval ( regclass ) → bigint</code>	<p>返回nextval在当前会话中为该序列最近获取的值。(如果在这个会话中没有为这个序列调用nextval会报告错误。)因为它返回的是一个会话本地值，所以它给出了一个可预测的答案，即自当前会话以来，其他会话是否执行了nextval。这个函数需要序列上的USAGE 或 SELECT特权。</p>
<code>lastval () → bigint</code>	<p>返回nextval在当前会话中最近返回的值。这个函数与currval相同，不同之处在于它没有使用序列名作为参数，而是引用当前会话中nextval最近应用到的序列。如果在当前会话中还没有调用nextval，那么调用lastval是一个错误。该函数在最后使用的序列上需要USAGE或SELECT特权。</p>

### Caution

为了避免阻塞从相同序列中获取数字的并发事务，nextval操作永远不会回滚；也就是说，一旦获取了一个值，它就会被认为是已使用的，并且不会再次返回。即使周围的事务随后中止，或者调用查询最终没有使用该值，也会出现这种情况。例如，带有ON CONFLICT子句的INSERT将计算要插入的元组，包括执行任何必需的nextval调用，在检测到任何可能导致它遵循ON CONFLICT规则的冲突之前。这种情况会在赋值序列中留下未使用的“holes”。因此，LightDB序列对象不能被用于获取“gapless”序列。

同样的，如果事务回滚，setval所做的任何序列状态更改都不会撤消。

如果数据库群集在提交包含 nextval 或 setval 调用的事务之前崩溃，则序列状态更改可能尚未传递到持久存储，因此不确定序列在群集重新启动后是否具有其原始或更新的状态。对于在数据库内使用序列来说，这是无害的，因为未

提交事务的其他影响也不会被看到。但是，如果您希望将序列值用于持久性的数据库外部用途，请确保在执行此操作之前已提交 `nextval` 调用。

序列函数所要操作的序列由 `regclass` 参数指定，该参数只是 `pg_class` 系统目录中序列的 OID。你不必手工查找 OID，不过，因为 `regclass` 数据类型的输入转换器将为您完成这项工作。只需将序列名用单引号括起来，这样它看起来就像一个文字常量。为了与处理普通 SQL 名称兼容，字符串将被转换为小写，除非它在序列名称周围包含双引号。因此：

```
nextval('foo')      序列上操作 foo
nextval('FOO')      序列上操作 foo
nextval('"Foo"')    序列上操作 Foo
```

如需要，序列名称可以是模式限定的：

```
nextval('myschema.foo')  操作 myschema.foo
nextval('"myschema".foo') 同上
nextval('foo')          在搜索路径中查找 foo
```

关于 `regclass` 的更多信息请参见 Section 8.19

### Note

在 LightDB 8.1 之前，序列函数的参数类型是 `text`，而不是 `regclass`，并且前文所述的从文本串到 OID 值的转换将在每次调用的时候发生。为了向后兼容，这个处理仍然存在，但是在内部实际上是通过在函数调用前隐式地将 `text` 转换成 `regclass` 实现的。

当你把一个序列函数的参数写成一个无修饰的字符串，那么它将变成类型为 `regclass` 的常量。因为这只是一个 OID，它将跟踪最初标识的序列，而不管后面是否改名、模式变化等等。这种“早期绑定”的行为通常是列默认值和视图中引用的序列所需要的。但是有时候你可能想要“延迟绑定”，其中序列的引用是在运行时解析的。要得到延迟绑定的行为，我们可以强制常量被存储为 `text` 常量，而不是 `regclass`：

```
nextval('foo'::text)    foo is looked up at runtime
```

当然，序列函数的参数也可以是表达式。如果它是一个文本表达式，那么隐式的转换将导致运行时的查找。

## 9.18. 条件表达式

本节描述在 LightDB 中可用的 SQL 兼容的条件表达式。

### Tip

如果你的需求超过这些条件表达式的能力，你可能会希望用一种更富表现力的编程语言写一个服务器端函数。

## Note

尽管COALESCE、GREATEST和LEAST在语法上类似于函数，但它们不是普通的函数，因此不能使用显式VARIADIC数组参数。

## 9.18.1. CASE

SQL CASE表达式是一种通用的条件表达式，类似于其它编程语言中的 if/else 语句：

```
CASE WHEN condition THEN result
      [WHEN ...]
      [ELSE result]
END
```

CASE子句可以用于任何表达式可以出现的地方。每一个condition是一个返回boolean结果的表达式。如果结果为真，那么CASE表达式的结果就是符合条件的result，并且剩下的CASE表达式不会被处理。如果条件的结果不为真，那么以相同方式搜寻任何随后的WHEN子句。如果没有WHEN condition为真，那么CASE表达式的值就是在ELSE子句里的result。如果省略了ELSE子句而且没有条件为真，结果为空。

例子：

```
SELECT * FROM test;
```

```
a
---
1
2
3
```

```
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
FROM test;
```

```
a | case
---+-----
1 | one
2 | two
3 | other
```

所有result表达式的数据类型都必须可以转换成单一的输出类型。参阅Section 10. 获取细节。

下面这个“简单”形式的CASE表达式是上述通用形式的一个变种：

```
CASE expression
      WHEN value THEN result
      [WHEN ...]
      [ELSE result]
```



END

第一个expression会被计算，然后与所有在WHEN子句中的每一个value对比，直到找到一个相等的。如果没有找到匹配的，则返回在ELSE子句中的result（或者控制）。这类似于C里的switch语句。

上面的例子可以用简单CASE语法来写：

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
            WHEN 2 THEN 'two'
            ELSE 'other'
       END
FROM test;
```

a	case
1	one
2	two
3	other

CASE表达式并不计算任何无助于判断结果的子表达式。例如，下面是一个可以避免被零除错误的方法：

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

在mysql兼容模式下(See `lightdb_dblevel_syntax_compatible_type`)，表达式的结果集可以是字符，数值，日期的任意组合，同时包含字符，数值和日期的种的任意两种以上时，返回值的类型为字符串。

```
SELECT CASE WHEN score >= 60 THEN score ELSE 'not pass' END AS score from
students;
```

### Note

如Section 4.2.1中所述，在有几种情况中一个表达式的子表达式 会被计算多次，因此“CASE只计算必要的表达式”这一原则并非不可打破。例如一个常量表达式1/0通常将会在规划时导致一次 除零错误，即便它位于一个执行时永远也不会进入的CASE分支时也是 如此。

## 9.18.2. COALESCE

COALESCE(value [, ...])

COALESCE函数返回它的第一个非空参数的值。当且仅当所有参数都为空时才会返回空。它常用于在为显示目的检索数据时用缺省值替换空值。例如：

```
SELECT COALESCE(description, short_description, '(none)') ...
```

如果description不为空，这将会返回它的值，否则如果short\_description非空则返回short\_description的值，如果前两个都为空则返回(none)。

所有参数都必须转换为一个公共数据类型，它将是结果的类型（详情请参见 Section 10.5）。

和CASE表达式一样，COALESCE将不会计算无助于判断结果的参数；也就是说，在第一个非空参数右边的参数不会被计算。这个 SQL 标准函数提供了类似于NVL和IFNULL的能力，它们被用在某些其他数据库系统中。

### 9.18.3. NULLIF

NULLIF(value1, value2)

当value1和value2相等时，NULLIF返回一个空值。否则它返回value1。这些可以用于执行前文给出的COALESCE例子的逆操作：

```
SELECT NULLIF(value, ' (none)') ...
```

在这个例子中，如果value是(none)，将返回空值，否则返回value的值。

这两个参数必须具有可比较的类型。具体来说，它们的比较与你写的 value1 = value2完全一样，因此必须有一个合适的=操作符可用。

结果的类型与第一个参数相同，但有一点细微的区别。实际上返回的是隐含 =操作符的第一个参数，在某些情况下，它将被提升以匹配第二个参数的类型。例如，NULLIF(1, 2.2) 生成 numeric，因为没有integer = numeric操作符，只有numeric = numeric。

### 9.18.4. GREATEST和LEAST

GREATEST(value [, ...])

LEAST(value [, ...])

GREATEST和LEAST函数从一个任意的数字表达式列表里选取最大或者最小的数值。这些表达式必须都可以转换为一个普通的数据类型，它将会是结果类型（参阅Section 10. 获取细节）。列表中的 NULL 数值将被忽略。只有所有表达式的结果都是 NULL 的时候，结果才会是 NULL。

请注意GREATEST和LEAST都不是 SQL 标准，但却是很常见的扩展。某些其他数据库让它们在任何参数为 NULL 时返回 NULL，而不是在所有参数都为 NULL 时才返回 NULL。

当 GUC 参数 lightdb\_dblevel\_syntax\_compatible\_type 设置为 oracle 时，GREATEST 和 LEAST 函数的行为遵循 Oracle 标准。在此模式下，如果任何参数求值为 NULL，则函数返回 NULL。另外，第一个参数用于确定返回类型。如果第一个参数是数值型，则 lightdb 确定具有最高数值优先级的参数，在比较之前隐式转换其余参数为该数据类型，并返回该数据类型。如果第一个参数不是数值型，则在比较之前将第一个参数之后的每个参数隐式转换为第一个参数的数据类型。

### 9.18.5. NVL 和 NVL2

NVL(expr1, expr2)

当 expr1 为 NULL 时，返回 expr2。当 expr1 不为 NULL 时，返回 expr1。expr1 和 expr2 的数据类型可以不同。我们支持以下不同的数据类型。

Name	Result data type	Argument data types
nvl	bit	bit, "any"
nvl	bytea	bytea, "any"
nvl	date	date, "any"
nvl	numeric	double precision, "any"
nvl	numeric	integer, "any"
nvl	numeric	numeric, "any"
nvl	text	text, "any"
nvl	timestamp without time zone	timestamp without time zone, "any"
nvl	timestamp with time zone	timestamp with time zone, "any"
nvl	time without time zone	time without time zone, "any"

上面给出的 NVL 示例:

```
lightdb@test=# select nvl(1.1, 'test'::text);
nvl
-----
1.1
(1 row)
```

在这个例子中, expr1 的数据类型是数字, 而 expr2 的数据类型是文本。因此, 调用函数 nvl(numeric, "any") 并返回数字数据类型。

在某些特殊情况下, 当 expr1 为 null 时返回的数据类型取决于数据源。当执行以下示例时, 根据隐式转换规则, null 实际上是未知类型, 因此调用的函数是 nvl(text, 'any'), 因此函数返回文本类型。

```
lightdb@test=# select pg_typeof(nvl(null, 1)), nvl(null, 1);
pg_typeof | nvl
-----+-----
text      | 1
(1 row)
```

在以下示例中, 因为数据库中的 null 字段是 int 类型, 所以该函数返回一个数字类型。

```
create table test_nvl (id int, func varchar(20));
insert into test_nvl values (null, 'nvl');

lightdb@test=# select pg_typeof(nvl(id, 1)), nvl(id, 1) from test;
pg_typeof | nvl
-----+-----
numeric   | 1
(1 row)
```

由于 ltrim 返回的数据类型是文本类型, 因此在下面的示例中实际调用的函数是 nvl(text, "any"), 因此该函数返回文本类型。

```
lightdb@test=# select pg_typeof(nvl(ltrim(func, 'nvl'), 1)), nvl(ltrim(func,
'nvl'), 1) from test;
```

```

pg_typeof | nvl
-----+-----
text      | 1
(1 row)

```

上述特性也适用于函数 `nvl2`、`decode` 和 `decode2`。

```
NVL2(expr, substitute1, substitute2)
```

`NVL2` 根据指定值是否为 `NULL` 返回替代值。当 `expr` 为 `NULL` 时，返回 `substitute2`。当 `expr` 不为 `NULL` 时，返回 `substitute1`。

`substitute1` 和 `substitute2` 的数据类型要相同，但 `expr` 可以不同。对于 `substitute1` 和 `substitute2`，我们支持以下数据类型。

Name	Result data type	Argument data types
<code>nvl2</code>	<code>bit</code>	<code>bit</code> , "any"
<code>nvl2</code>	<code>bytea</code>	<code>bytea</code> , "any"
<code>nvl2</code>	<code>date</code>	<code>date</code> , "any"
<code>nvl2</code>	<code>numeric</code>	double precision, "any"
<code>nvl2</code>	<code>numeric</code>	integer, "any"
<code>nvl2</code>	<code>numeric</code>	numeric, "any"
<code>nvl2</code>	<code>text</code>	<code>text</code> , "any"
<code>nvl2</code>	<code>timestamp without time zone</code>	<code>timestamp without time zone</code> , "any"
<code>nvl2</code>	<code>timestamp with time zone</code>	<code>timestamp with time zone</code> , "any"
<code>nvl2</code>	<code>time without time zone</code>	<code>time without time zone</code> , "any"

上面给出的 `NVL2` 示例：

```

lightdb@test=# select nvl2('2022-01-01'::date, 'ab'::text, 'cd'::text);
nvl2
-----
ab
(1 row)

```

在这个例子中，`expr` 的数据类型是日期，`substitute1` 和 `substitute2` 的数据类型是文本。因此，调用函数 `nvl2("any", text, "any")` 并返回文本数据类型。

## 9.18.6. DECODE

`DECODE`函数将`expr`和每个`search`逐个进行比较，若`expr`与某个`search`相等，则返回该`search`对应的`result`；若`expr`与所有`search`都不相等，则返回`default`。若没有指定`default`，则返回`null`。

```

DECODE(expr,
       search, result,
       [search, result],
       [search, result]...,
       [default])

```

DECODE函数（参数个数大于2）仅在oracle模式下使用，这意味着在使用decode函数前，需先将GUC参数lightdb\_dblevel\_syntax\_compatible\_type设置为oracle。

LightDB会自动将expr和每个search的数据类型转成第一个search的类型后再比较，若类型不能转换则报错；同时会自动将返回值的类型转成第一个result的类型，若类型不能转换则报错。若第一个result为null，则函数的返回值为TEXT类型。

```
lightdb@oracle=# create table t(a char(10),b varchar2(10));
```

```
CREATE TABLE
```

```
lightdb@oracle=# insert into t values('a','a');
```

```
INSERT 0 1
```

```
lightdb@oracle=# select decode(a,b,1,2) from t;
```

```
decode
```

```
-----
          2
(1 row)
```

```
lightdb@oracle=# select decode(b,a,1,2) from t;
```

```
decode
```

```
-----
          1
(1 row)
```

```
lightdb@oracle=# select decode(trim(a),b,1,2) from t;
```

```
decode
```

```
-----
          1
(1 row)
```

```
lightdb@oracle=# select pg_typeof(decode(1,2,'3',4));
```

```
pg_typeof
```

```
-----
text
(1 row)
```

search、result和default可以是复杂的表达式，LightDB会对result和default使用“短路”方式计算上述表达式的值，即每个search仅在与expr比较且比较结果为相等时才会计算result或default值，否则无需计算。该“短路”规则基于一个前提，那就是必须存在合适的类型转换规则，若不存在转换规则则直接报错。另外该“短路”规则对search不适用，也就是说一开始就要将所有的search的类型要全部转成第一个search的类型。

```
lightdb@oracle=# select decode(3,
```

```
1,1/0, --short-circuit on result, no division by
```

```
zero error
```

```
2,2/0, --short-circuit on result, no division by
```

```
zero error
```

```
3,300,
```

```
'x' --short-circuit on default, no type
```

```
conversion error
```

```
);
```

```
decode
```

```
-----
        300
(1 row)
```

```
SELECT DECODE(123,
```

```

123, 'Number Match',
'Not Match', 'Default') --'Not Match' cannot convert to int(type of 1st
search)
FROM DUAL;
ERROR: invalid input syntax for type numeric: "Not Match"
LINE 3: 'Not Match', 'Default') --'Not Match' cannot convert to in...

```

在DECODE函数中，LightDB在比较时会认为两个null是相等的，即当expr为null，某个search也为null时，LightDB认为expr与该search相等，所以函数会返回该search对应的result。

```

lightdb@oracle=# select decode(null,null,1,2);
decode
-----
1
(1 row)

```

## 9.19. 数组函数和操作符

Table 9.5显示了可以用于数组类型的专用的操作符。除此之外，Table 9.中所显示的常用比较运算符也适用于数组。比较操作符逐个元素的比较数组内容，使用默认的元素数据类型的B-tree比较函数，并根据第一个差值进行排序。多维数组的元素按照行序进行访问（最后的下标变化最快）。如果两个数组的内容相同但维数不等，那么维度信息中的第一个不同将决定排序顺序（这是对LightDB 8.2之前版本的修改：老版本认为内容相同的两个数组相等，即使它们的维数或下标范围并不相同）。

Table 9.52. 数组操作符

操作符	描述 例子
anyarray @> anyarray	→ boolean 第一个数组是否包含第二个数组，也就是说，出现在第二个数组中的每个元素是否等于第一个数组中的某个元素？（重复值不需要特殊处理，因此ARRAY[1] 和 ARRAY[1, 1]被认为包含对方。） ARRAY[1, 4, 3] @> ARRAY[3, 1, 3] → t
anyarray <@ anyarray	→ boolean 第一个数组包含在第二个数组中么？ ARRAY[2, 2, 7] <@ ARRAY[1, 7, 4, 2, 6] → t
anyarray && anyarray	→ boolean 这些数组有重叠么，也就是说，它们有共同的元素么？ ARRAY[1, 4, 3] && ARRAY[2, 1] → t
anyarray    anyarray	→ anyarray 连接两个数组。连接空(null)或空数组是一个无操作(no-op)；否则，数组必须具有相同的维度数(如第一个示例所示)，或者维度数相差一个(如第二个示例所示)。 ARRAY[1, 2, 3]    ARRAY[4, 5, 6, 7] → {1, 2, 3, 4, 5, 6, 7} ARRAY[1, 2, 3]    ARRAY[[4, 5, 6], [7, 8, 9]] → {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
anyelement    anyarray	→ anyarray 将元素连接到数组的前面(数组必须为空或一维的[one-dimensional])。 3    ARRAY[4, 5, 6] → {3, 4, 5, 6}
anyarray    anyelement	→ anyarray 将元素连接到数组的末尾(数组必须为空或一维的[one-dimensional])。 ARRAY[4, 5, 6]    7 → {4, 5, 6, 7}

参阅Section 8.1获取有关数组操作符行为的更多细节。有关哪些操作符支持被索引的操作，请参阅Section 11.2

Table 9.5展示了可以用于数组类型的函数。参阅Section 8.1获取更多信息以及使用这些函数的例子。

Table 9.53. 数组函数

函数	描述 例子
<code>array_append</code> ( anyarray, anyelement ) → anyarray	向一个数组的末端追加一个元素 (等同于 <code>anyarray    anyelement</code> 操作符)。 <code>array_append(ARRAY[1, 2], 3) → {1, 2, 3}</code>
<code>array_cat</code> ( anyarray, anyarray ) → anyarray	连接两个数组 (等同于 <code>anyarray    anyarray</code> 操作符)。 <code>array_cat(ARRAY[1, 2, 3], ARRAY[4, 5]) → {1, 2, 3, 4, 5}</code>
<code>array_dims</code> ( anyarray ) → text	返回数组维度的文本表示形式。 <code>array_dims(ARRAY[[1, 2, 3], [4, 5, 6]]) → [1:2][1:3]</code>
<code>array_fill</code> ( anyelement, integer[] [, integer[] ] ) → anyarray	返回一个包含给定值的拷贝的数组，其维数与第二个参数指定的长度相同。可选的第三个参数提供每个维度的下界值 (默认为全部为 1)。 <code>array_fill(11, ARRAY[2, 3]) → {{11, 11, 11}, {11, 11, 11}}</code> <code>array_fill(7, ARRAY[3], ARRAY[2]) → [2:4]={7, 7, 7}</code>
<code>array_length</code> ( anyarray, integer ) → integer	返回请求的数组维度的长度。 <code>array_length(array[1, 2, 3], 1) → 3</code> <code>array_length(array[]::int[], 1) → NULL</code> <code>array_length(array['text'], 2) → NULL</code>
<code>array_lower</code> ( anyarray, integer ) → integer	返回请求的数组维度的下界。 <code>array_lower(' [0:2]={1, 2, 3}'::integer[], 1) → 0</code>
<code>array_ndims</code> ( anyarray ) → integer	返回数组的维度数。 <code>array_ndims(ARRAY[[1, 2, 3], [4, 5, 6]]) → 2</code>
<code>array_position</code> ( anyarray, anyelement [, integer ] ) → integer	返回阵列中第二个参数中第一次发生的下标，如果它不存在，则返回NULL。如果给出了第三个参数，则搜索从该下标开始。数组必须是一维的。比较是使用IS NOT DISTINCT FROM语义进行的，所以可以搜索NULL。 <code>array_position(ARRAY['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'], 'mon') → 2</code>
<code>array_positions</code> ( anyarray, anyelement ) → integer[]	返回作为第一个参数的数组中第二个参数所有出现的下标的数组。数组必须是一维的。使用IS NOT DISTINCT FROM语义完成比较，所以可以搜索NULL。只有当数组为NULL时才返回NULL；如果在数组中没有找到该值，则返回空数组。 <code>array_positions(ARRAY['A', 'A', 'B', 'A'], 'A') → {1, 2, 4}</code>
<code>array_prepend</code> ( anyelement, anyarray ) → anyarray	在数组的开头添加一个元素 (等同于 <code>anyelement    anyarray</code> 操作符)。 <code>array_prepend(1, ARRAY[2, 3]) → {1, 2, 3}</code>
<code>array_remove</code> ( anyarray, anyelement ) → anyarray	从数组中移除与给定值相等的所有元素。数组必须是一维的。使用IS NOT DISTINCT FROM语义完成比较，所以可以删除NULL。 <code>array_remove(ARRAY[1, 2, 3, 2], 2) → {1, 3}</code>

函数	描述 例子
<code>array_replace</code>	<code>array_replace ( anyarray, anyelement, anyelement ) → anyarray</code> 将等于第二个参数的每个数组元素替换为第三个参数。 <code>array_replace(ARRAY[1, 2, 5, 4], 5, 3) → {1, 2, 3, 4}</code>
<code>array_to_string</code>	<code>array_to_string ( array anyarray, delimiter text [, null_string text ] ) → text</code> 将每个数组元素转换为它的文本表现形式，并连接由delimiter符字符串分隔的元素。如果给出了null_string且不为NULL，则NULL数组项由该字符串表示；否则，它们将被省略。 <code>array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*') → 1,2,3,*,5</code>
<code>array_upper</code>	<code>array_upper ( anyarray, integer ) → integer</code> 返回请求的数组维度的上界。 <code>array_upper(ARRAY[1, 8, 3, 7], 1) → 4</code>
<code>cardinality</code>	<code>cardinality ( anyarray ) → integer</code> 返回数组中元素的总数，如果数组为空则返回0。 <code>cardinality(ARRAY[[1, 2], [3, 4]]) → 4</code>
<code>string_to_array</code>	<code>string_to_array ( string text, delimiter text [, null_string text ] ) → text[]</code> 在出现分隔符时拆分字符串，并将剩余数据形成文本数组。如果分隔符为NULL，则字符串中的每个字符将成为数组中的单独元素。如果分隔符为空字符串，则该字符串视为单个字段。如果提供了null_string且不为NULL，那么匹配该字符串的字段将被转换为NULL项。 <code>string_to_array('xx~~yy~~zz', '~~', 'yy') → {xx, NULL, zz}</code>
<code>unnest</code>	<code>unnest ( anyarray ) → setof anyelement</code> 将数组展开为一组行。 <code>unnest(ARRAY[1, 2]) →</code>  1 2  <code>unnest(ARRAY[['foo', 'bar'], ['baz', 'quux']]) →</code>  foo bar baz quux
<code>unnest</code>	<code>unnest ( anyarray, anyarray [, ... ] ) → setof anyelement, anyelement [, ... ]</code> 将多个数组(可能是不同的数据类型)展开为一组行。如果数组的长度不完全相同，那么较短的数组将用NULL填充。这只在查询的FROM子句中允许；参见Section 7.2.1.4 <code>select * from unnest(ARRAY[1, 2], ARRAY['foo', 'bar', 'baz']) as x(a, b) →</code>  a   b ---+----- 1   foo 2   bar   baz



## Note

string\_to\_array的行为中有两点与LightDB 9.1之前的版本不同。首先，当输入串的长度为零时，它将返回一个空（无元素）数组而不是 NULL。其次，如果定界字符串为 NULL，该函数会将输入划分成独立字符，而不是像以前那样返回 NULL。

也可参见Section 9.2了解用于数组的聚集函数array\_agg。

## 9.20. 范围函数和运算符

范围类型的概述可参见 Section 8.17。

Table 9.54显示了范围类型的专用操作符。除此之外，Table 9.中所示的常用比较操作符也适用于范围类型。比较操作符首先按范围下界排序，只有当它们相等时才比较上界。这通常不会导致有用的总体排序，但提供的操作符允许在范围上构造惟一索引。

Table 9.54. 范围操作符

操作符	描述 例子
anyrange @> anyrange	→ boolean 第一个范围中包含第二个(范围)吗? int4range(2, 4) @> int4range(2, 3) → t
anyrange @> anyelement	→ boolean 范围是否包含元素? '[2011-01-01, 2011-03-01]'::tsrange @> '2011-01-10'::timestamp → t
anyrange <@ anyrange	→ boolean 第一个范围包含在第二个(范围)中吗? int4range(2, 4) <@ int4range(1, 7) → t
anyelement <@ anyrange	→ boolean 元素是否包含在范围内? 42 <@ int4range(1, 7) → f
anyrange && anyrange	→ boolean 范围是否重叠，也就是说，是否有相同的元素? int8range(3, 7) && int8range(4, 12) → t
anyrange << anyrange	→ boolean 第一个范围是否严格地在第二个(范围)的左侧? int8range(1, 10) << int8range(100, 110) → t
anyrange >> anyrange	→ boolean 第一个范围是否严格符合第二个(范围)? int8range(50, 60) >> int8range(20, 30) → t
anyrange &< anyrange	→ boolean 第一个范围是否没有扩展到第二个(范围)的右侧? int8range(1, 20) &< int8range(18, 20) → t
anyrange &> anyrange	→ boolean 第一个范围是否没有扩展到第二个(范围)的左侧? int8range(7, 20) &> int8range(5, 10) → t
anyrange - - anyrange	→ boolean 范围是相邻的么? numrange(1.1, 2.2) - - numrange(2.2, 3.3) → t

操作符	描述 例子
<code>anyrange + anyrange</code>	<code>→ anyrange</code> 计算范围的并集。范围必须重叠或相邻，这样的并集就是一个单一的范围(请参见 <code>range_merge()</code> )。 <code>numrange(5, 15) + numrange(10, 20) → [5, 20)</code>
<code>anyrange * anyrange</code>	<code>→ anyrange</code> 计算范围的交集。 <code>int8range(5, 15) * int8range(10, 20) → [10, 15)</code>
<code>anyrange - anyrange</code>	<code>→ anyrange</code> 计算范围的差异。第二个范围必须不能包含在第一个(范围)中，以使差异不是一个单一的范围。 <code>int8range(5, 15) - int8range(10, 20) → [5, 10)</code>

当涉及一个空范围时，左部/右部/相邻操作符总是返回假；即一个空范围被认为不在任何其他范围前面或者后面。

Table 9.55显示可用于范围类型的函数。

Table 9.55. 范围函数

函数	描述 例子
<code>lower ( anyrange )</code>	<code>→ anyelement</code> 提取范围的下界(如果范围为空或下界为无限，则为NULL)。 <code>lower(numrange(1.1, 2.2)) → 1.1</code>
<code>upper ( anyrange )</code>	<code>→ anyelement</code> 提取范围的上限(如果范围为空或上限为无限，则为NULL)。 <code>upper(numrange(1.1, 2.2)) → 2.2</code>
<code>isempty ( anyrange )</code>	<code>→ boolean</code> 范围为空吗? <code>isempty(numrange(1.1, 2.2)) → f</code>
<code>lower_inc ( anyrange )</code>	<code>→ boolean</code> 范围的下界是否包含在内? <code>lower_inc(numrange(1.1, 2.2)) → t</code>
<code>upper_inc ( anyrange )</code>	<code>→ boolean</code> 范围的上界是否包含在内? <code>upper_inc(numrange(1.1, 2.2)) → f</code>
<code>lower_inf ( anyrange )</code>	<code>→ boolean</code> 范围的下界是无限的吗? <code>lower_inf('(',')'::daterange) → t</code>
<code>upper_inf ( anyrange )</code>	<code>→ boolean</code> 范围的上界是无限的吗? <code>upper_inf('(',')'::daterange) → t</code>
<code>range_merge ( anyrange, anyrange )</code>	<code>→ anyrange</code> 计算包含两个给定范围的最小范围。 <code>range_merge('[1,2]'::int4range, '[3,4]'::int4range) → [1,4)</code>

`lower_inc`、`upper_inc`、`lower_inf`、和 `upper_inf` 函数对空范围(empty range)都返回假(false)。

## 9.21. 聚集函数

聚集函数从一个输入值的集合计算出一个单一值。内建的通用聚集函数在Table 9.56中列出，而统计性聚集是在Table 9.5中列出。内建的组内有序集聚集函数在Table 9.58中列出，而内建的组内假想集聚集在Table 9.59中列出。与聚集函数紧密相关的分组操作在Table 9.60中列出。Section 4.2.7中会解释针对聚集函数的特殊语法考虑。额外的介绍信息请参考Section 2.7

支持部分模式的聚合函数具备参与各种优化的条件，例如并行聚合。

Table 9.56. 通用聚集函数

函数 描述	部分模式
<code>array_agg ( anynonarray ) → anyarray</code> 将所有输入值，包括空值，收集到一个数组中。	No
<code>array_agg ( anyarray ) → anyarray</code> 将所有输入数组连接到一个更高维度的数组中。(输入必须都具有相同的维度，并且不能为空的(empty)或空值(null)。)	No
<code>avg ( smallint ) → numeric</code> <code>avg ( integer ) → numeric</code> <code>avg ( bigint ) → numeric</code> <code>avg ( numeric ) → numeric</code> <code>avg ( real ) → double precision</code> <code>avg ( double precision ) → double precision</code> <code>avg ( interval ) → interval</code> 计算所有非空输入值的平均值(算术平均值)。	Yes
<code>bit_and ( smallint ) → smallint</code> <code>bit_and ( integer ) → integer</code> <code>bit_and ( bigint ) → bigint</code> <code>bit_and ( bit ) → bit</code> 计算所有非空输入值的逐位AND。	Yes
<code>bit_or ( smallint ) → smallint</code> <code>bit_or ( integer ) → integer</code> <code>bit_or ( bigint ) → bigint</code> <code>bit_or ( bit ) → bit</code> 计算所有非空输入值的逐位OR。	Yes
<code>bool_and ( boolean ) → boolean</code> 如果全部非空输入值都为真则返回真，否则返回假。	Yes
<code>bool_or ( boolean ) → boolean</code> 如果任何非空输入值为真则返回真，否则返回假。	Yes
<code>count ( * ) → bigint</code> 计算输入行的数量。	Yes
<code>count ( "any" ) → bigint</code> 计算输入值不为空的输入行的数量。	Yes
<code>every ( boolean ) → boolean</code> 这是对应 <code>bool_and</code> 的SQL标准的等效物。	Yes
<code>json_agg ( anyelement ) → json</code> <code>jsonb_agg ( anyelement ) → jsonb</code> 收集所有输入值，包括空值，到一个JSON数组。根据 <code>to_json</code> 或 <code>to_jsonb</code> 将值转换为JSON。	No
<code>json_object_agg ( key "any", value "any" ) → json</code> <code>jsonb_object_agg ( key "any", value "any" ) → jsonb</code> 将所有键/值对收集到一个JSON对象中。关键参数强制转换为文本；值参数按照 <code>to_json</code> 或 <code>to_jsonb</code> 进行转换。值可以为空，但键不能(为空)。	No
<code>max ( see text ) → same as input type</code>	Yes

函数	描述	部分模式
	计算非空输入值的最大值。适用于任何数字、字符串、日期/时间或enum类型，以及inet, interval, money, oid, pg_lsn, tid和任何这些类型的数组。	
min ( see text )	→ same as input type 计算非空输入值的最小值。可用于任何数字、字符串、日期/时间或enum类型，以及inet, interval, money, oid, pg_lsn, tid和任何这些类型的数组。	Yes
string_agg ( value text, delimiter text )	→ text string_agg ( value bytea, delimiter bytea ) → bytea 连接非空输入值到字符串中。第一个值之后的每个值前面都有相应的分隔符(delimiter) (如果它不为空)。	No
sum ( smallint )	→ bigint sum ( integer ) → bigint sum ( bigint ) → numeric sum ( numeric ) → numeric sum ( real ) → real sum ( double precision ) → double precision sum ( interval ) → interval sum ( money ) → money 计算非空输入值的总和。	Yes
xmlagg ( xml )	→ xml 连接非空的XML输入值(参见Section 9.15.1.7)。	No

应该注意的是，除了count之外，这些函数在没有选择行时返回空值。特别地，行数的sum返回空(null)，而不是预期的零，array\_agg在没有输入行时返回空(null)而不是空数组。coalesce函数可以在必要时用零或空数组代替空(null)。

聚合函数 array\_agg, json\_agg, jsonb\_agg, json\_object\_agg, jsonb\_object\_agg, string\_agg, 和 xmlagg, 以及类似的用户定义的聚合函数，根据输入值的顺序产生富有意义的不同的结果值。默认情况下，这种排序是不指定的，但可以通过在聚合调用中写入ORDER BY子句来控制，如Section 4.2.所示。或者，从排序的子查询提供输入值通常也可以。例如：

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

注意，如果外部查询级别包含其他处理，例如关联，则此方法可能会失败，因为这可能导致子查询的输出在计算聚合之前重新排序。

### Note

布尔聚合 bool\_and 和 bool\_or 对应于标准SQL聚合 every 和 any 或 some。LightDB 支持 every, 但不支持 any 或 some, 因为标准语法中存在模糊性：

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

这里ANY可以被认为引入子查询，或者是聚合函数，如果子查询返回一行布尔值。因此，不能为这些聚合提供标准名称。

### Note

习惯使用其他SQL数据库管理系统的用户可能会对count聚合应用于整个表时的性能感到失望。一个类似下面的查询：

```
SELECT count(*) FROM sometable;
```

将需要与表大小成比例的工作:LightDB将需要扫描整个表或包含表中所有行的索引。

### Note

当您使用 SharedJDBC 框架时,它可能会解析聚合函数的参数名。但是,默认情况下,LightDB 输出函数调用列名没有参数。

```
select max(a) from agg_table;
max
-----
  10
(1 row)
```

如果您遇到了这个问题,可以打开开关 `lightdb_aggcolumn_rename`。当打开这个开关时,当您使用列引用调用 `count`、`avg`、`min`、`max`、`sum` 时,LightDB 将输出带有参数列表的列名。

```
set lightdb_aggcolumn_rename=on;
select max(a) from agg_table;
MAX(A)
-----
  10
(1 row)
```

这种重命名仅适用于简单的列引用,而不适用于函数调用、表达式等情况。

```
set lightdb_aggcolumn_rename=on;
select max(a + 1) from agg_table;
max
-----
  11
(1 row)
```

Table 9.5显示了统计分析中常用的聚合函数。(这些被分离出来仅仅是为了避免使更常用的聚合列表混乱。) 显示为接受`numeric_type`的函数可用于所有类型`smallint`, `integer`, `bigint`, `numeric`, `real`, 和 `double precision`。在描述中提及N时,它意味着所有输入表达式都非空的输入行数。在所有情况下,如果计算没有意义,则返回`null`,例如当N为0时。

Table 9.57. 用于统计的聚集函数

函数 描述	部分模式
<code>corr ( Y double precision, X double precision ) → double precision</code> 计算相关系数。	Yes
<code>covar_pop ( Y double precision, X double precision ) → double precision</code> 计算总体协方差。	Yes

函数	描述	部分模式
	<code>covar_samp ( Y double precision, X double precision ) → double precision</code> 计算样本协方差。	Yes
	<code>regr_avgx ( Y double precision, X double precision ) → double precision</code> 计算自变量的平均值, $\text{sum}(X)/N$ 。	Yes
	<code>regr_avgy ( Y double precision, X double precision ) → double precision</code> 计算因变量的平均值, $\text{sum}(Y)/N$ 。	Yes
	<code>regr_count ( Y double precision, X double precision ) → bigint</code> 计算两个输入都非空的行数。	Yes
	<code>regr_intercept ( Y double precision, X double precision ) → double precision</code> 计算由 (X, Y) 对决定的最小二乘拟合的线性方程的Y-截距。	Yes
	<code>regr_r2 ( Y double precision, X double precision ) → double precision</code> 计算相关系数的平方。	Yes
	<code>regr_slope ( Y double precision, X double precision ) → double precision</code> 计算由 (X, Y) 对决定的最小二乘拟合的线性方程的斜率。	Yes
	<code>regr_sxx ( Y double precision, X double precision ) → double precision</code> 计算自变量的“平方和” $\text{sum}(X^2) - \text{sum}(X)^2/N$ 。	Yes
	<code>regr_sxy ( Y double precision, X double precision ) → double precision</code> 计算独立变量乘以因变量的“sum of products”, $\text{sum}(X*Y) - \text{sum}(X) * \text{sum}(Y)/N$ 。	Yes
	<code>regr_syy ( Y double precision, X double precision ) → double precision</code> 计算因变量的“平方和”, $\text{sum}(Y^2) - \text{sum}(Y)^2/N$ 。	Yes
	<code>stddev ( numeric_type ) → double precision for real or double precision, otherwise numeric</code> 这是stddev_samp的一个历史别称。	Yes
	<code>stddev_pop ( numeric_type ) → double precision for real or double precision, otherwise numeric</code> 计算输入值的总体标准差。	Yes
	<code>stddev_samp ( numeric_type ) → double precision for real or double precision, otherwise numeric</code> 计算输入值的样本标准差。	Yes
	<code>variance ( numeric_type ) → double precision for real or double precision, otherwise numeric</code> 这是 var_samp 的一个历史别称。	Yes
	<code>var_pop ( numeric_type ) → double precision for real or double precision, otherwise numeric</code> 计算输入值的总体方差(总体标准差的平方)。	Yes
	<code>var_samp ( numeric_type ) → double precision for real or double precision, otherwise numeric</code> 计算输入值的样本方差(样本标准差的平方)。	Yes

Table 9.58显示了一些使用ordered-set aggregate语法的聚合函数。 这些函数有时被称为“inverse distribution”函数。 它们的聚合输入是通过ORDER BY引入的, 它们还可以

接受未聚合的 `direct argument`，但只计算一次。所有这些函数在其聚合的输入中都忽略空 (`null`) 值。对于使用 `fraction(fraction)` 参数的函数，分数值必须在 0 到 1 之间；否则将抛出一个错误。但是，空分数值简单地产生一个空结果。

Table 9.58. 有序集聚集函数

函数 描述	部分模式
<code>mode () WITHIN GROUP ( ORDER BY anyelement ) → anyelement</code> 计算 <code>mode</code> ，即聚合参数最频繁的值 (如果有多个相同频繁的值，第一个可以任意选择)。聚合参数必须是可排序类型。	No
<code>percentile_cont ( fraction double precision ) WITHIN GROUP ( ORDER BY double precision ) → double precision</code> <code>percentile_cont ( fraction double precision ) WITHIN GROUP ( ORDER BY interval ) → interval</code> 计算 <code>continuous percentile</code> ，该值对应于聚合参数值的有序集合中的指定分数 ( <code>fraction</code> )。如果需要，这将在相邻的输入项之间插入。	No
<code>percentile_cont ( fractions double precision[] ) WITHIN GROUP ( ORDER BY double precision ) → double precision[]</code> <code>percentile_cont ( fractions double precision[] ) WITHIN GROUP ( ORDER BY interval ) → interval[]</code> 计算多个连续的百分位数。结果是一个与分数 ( <code>fractions</code> ) 参数具有相同维数的数组，每个非空元素都被对应于该百分位的 (可能插值的) 值所替换。	No
<code>percentile_disc ( fraction double precision ) WITHIN GROUP ( ORDER BY anyelement ) → anyelement</code> 计算离散百分比 ( <code>discrete percentile</code> )，即聚合参数值的有序集合中的第一个值，该值在排序中的位置等于或超过指定的 <code>fraction</code> 。聚合参数必须是可排序类型。	No
<code>percentile_disc ( fractions double precision[] ) WITHIN GROUP ( ORDER BY anyelement ) → anyarray</code> 计算多个离散百分位数。结果是一个与 <code>fractions</code> 参数具有相同维数的数组，每个非空元素都被对应于该百分位的输入值替换。聚合参数必须是可排序类型。	No

列在 Table 9.59 中的每个 “`hypothetical-set`” 聚合都与 Section 9.2 中定义的同名窗口函数相关联。在每种情况下，聚合的结果都是相关的窗口函数将由 `args` 构造的 “`hypothetical`” 行返回的值，如果将这样的行添加到 `sorted_args` 表示的已排序行组中。对于这些函数中的每一个，`args` 中给出的直接参数列表必须与 `sorted_args` 中给出的聚合参数的数量和类型匹配。与大多数内置聚合不同，这些聚合不是严格的，也就是说它们不会删除包含空值的输入行。空值根据 `ORDER BY` 子句中指定的规则排序。

Table 9.59. 假想集聚集函数

函数 描述	部分模式
<code>rank ( args ) WITHIN GROUP ( ORDER BY sorted_args ) → bigint</code> 计算假设行的排名，包括间隔，就是说在它的对等组中第一行的行号。	No
<code>dense_rank ( args ) WITHIN GROUP ( ORDER BY sorted_args ) → bigint</code> 计算假设行的排名，没有间隔；这个功能有效地计数对等组。	No
<code>percent_rank ( args ) WITHIN GROUP ( ORDER BY sorted_args ) → double precision</code>	No

函数	描述	部分模式
	计算假设行的相关排行, 也就是 $(rank - 1) / (total\ rows - 1)$ 。取值范围为 0 到 1 (含)。	
	<code>cume_dist ( args ) WITHIN GROUP ( ORDER BY sorted_args ) → double precision</code> 计算累积分布, 也就是(前面或具有假设行的对等行数)/(总行数)。取值范围为 1/N 到 1。	No

Table 9.60. 分组操作

函数	描述
	<code>GROUPING ( group_by_expression(s) ) → integer</code> 返回一个位掩码以指示哪个GROUP BY表达式没有包含在当前分组集中。 比特位被分配给最右边的参数对应于最低有效位; 如果对应的表达式包含在生成当前结果行的分组集的分组条件中, 则每个位为0, 如果不包含则为1。

Table 9.60所示的分组操作与分组集(参见Section 7.2.4)共同使用, 以区分结果行。GROUPING函数的参数实际上并不求值, 但它们必须与相关查询级别的GROUP BY子句中给出的表达式完全匹配。例如:

```
=> SELECT * FROM items_sold;
```

make	model	sales
Foo	GT	10
Foo	Tour	20
Bar	City	15
Bar	Sport	5

(4 rows)

```
=> SELECT make, model, GROUPING(make,model), sum(sales) FROM items_sold GROUP BY ROLLUP(make,model);
```

make	model	grouping	sum
Foo	GT	0	10
Foo	Tour	0	20
Bar	City	0	15
Bar	Sport	0	5
Foo		1	30
Bar		1	20
		3	50

(7 rows)

在这里, 前四行中的grouping值0表明这些已经正常分组, 在两个分组列上。 值1表示model没有在倒数两行中分组, 值3表示无论是make还是model都没有在最后一行中分组(因此, 这是所有输入行的聚合)。

## 9.22. 窗口函数

Window functions提供了跨越与当前查询行相关的行集执行计算的能力。 该特性的介绍请参见Section 3.5 语法细节请参见Section 4.2.8

内置的窗口函数罗列在Table 9.6中。注意, 这些函数必须使用窗口函数语法来调用, 也就是说, 需要一个OVER子句。



除了这些函数之外，任何内置的或用户定义的普通聚合(例如非有序集或假设集聚合)都可以作为窗口函数使用;关于内置聚合的列表，参见Section 9.21 聚合函数只有在调用之后有一个OVER子句时才作为窗口函数;否则，它们充当普通的聚合，并为整个集合返回一行。

Table 9.61. 通用窗口函数

函数	描述
<code>row_number ()</code> → bigint	返回其分区内的当前行数，从1开始计数。
<code>rank ()</code> → bigint	返回当前行的排名，包含间隔;即对等组中第一行的row_number。
<code>dense_rank ()</code> → bigint	返回当前行的排名，不包括间隔;这个功能有效地计数对等组。
<code>percent_rank ()</code> → double precision	返回当前行的相对排名，即 $(rank - 1) / (\text{总的分区行数} - 1)$ 。因此，该值的范围从0到1(包含在内)。
<code>cume_dist ()</code> → double precision	返回累积分布，也就是(当前行之前或对等的分区行数)/(总的分区行数)。取值范围为1/N 到 1。
<code>ntile ( num_buckets integer )</code> → integer	返回一个从1到参数值的整数，并将分区划分为尽可能相等的值。
<code>lag ( value anyelement [, offset integer [, default anyelement ]])</code> → anyelement	返回分区中在当前行之前offset行的value;如果没有这样的行，则返回default(必须与value具有相同的类型)。offset和default都是针对当前行求值的。如果省略，offset默认为1，default为NULL。
<code>lead ( value anyelement [, offset integer [, default anyelement ]])</code> → anyelement	返回分区中在当前行之后offset行的value; 如果没有这样的行，则返回default(必须与value具有相同的类型)。offset和default都是针对当前行求值的。如果省略，offset默认为1，default为NULL。
<code>first_value ( value anyelement )</code> → anyelement	返回在窗口框架的第一行求得的value。
<code>last_value ( value anyelement )</code> → anyelement	返回在窗口框架的最后一行求得的value。
<code>nth_value ( value anyelement, n integer )</code> → anyelement	返回在窗口框架的第n行求得的value(从1开始计数);如果没有这样的行，则返回NULL。

在Table 9.6中列出的所有函数都依赖于相关窗口定义的ORDER BY子句指定的排序顺序。仅考虑ORDER BY列时不能区分的行被称为是同等行。定义的这四个排名函数(包括cume\_dist)，对于对等组的所有行的答案相同。

注意first\_value、last\_value和nth\_value只考虑“窗口帧”内的行，它默认情况下包含从分区的开始行直到当前行的最后一个同等行。这对last\_value可能不会给出有用的结果，有时对nth\_value也一样。你可以通过向OVER子句增加一个合适的帧声明(RANGE或GROUPS)来重定义帧。关于帧声明的更多信息请参考Section 4.2.8

当一个聚集函数被用作窗口函数时，它将在当前行的窗口帧内的行上聚集。一个使用ORDER BY和默认窗口帧定义的聚集产生一种“运行时求和”类型的行为，这可能是或者不是想要的结果。为了获取在整个分区上的聚集，忽略ORDER BY或者使用ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING。其它窗口帧声明可以用来获得其它的效果。

## Note

SQL 标准为 `lead`、`lag`、`first_value`、`last_value` 和 `nth_value` 定义了一个 `RESPECT NULLS` 或 `IGNORE NULLS` 选项。这在 LightDB 中没有实现：行为总是与标准的默认相同，即 `RESPECT NULLS`。同样，标准中用于 `nth_value` 的 `FROM FIRST` 或 `FROM LAST` 选项没有实现：只有支持默认的 `FROM FIRST` 行为（你可以通过反转 `ORDER BY` 的排序达到 `FROM LAST` 的结果）。

## 9.23. 子查询表达式

本节描述 LightDB 中可用的 SQL 兼容的子查询表达式。所有本节中成文的表达式都返回布尔值（真/假）结果。

### 9.23.1. EXISTS

`EXISTS (subquery)`

`EXISTS` 的参数是一个任意的 `SELECT` 语句，或者说子查询。系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，那么 `EXISTS` 的结果就为“真”；如果子查询没有返回行，那么 `EXISTS` 的结果是“假”。

子查询可以引用来自周围的查询的变量，这些变量在该子查询的任何一次计算中都起常量的作用。

这个子查询通常只是运行到能判断它是否可以返回至少一行为止，而不是等到全部结束。在这里写任何有副作用的子查询都是不明智的（例如调用序列函数）；这些副作用是否发生是很难判断的。

因为结果只取决于是否会返回行，而不取决于这些行的内容，所以这个子查询的输出列表通常是无关紧要的。一个常用的编码习惯是用 `EXISTS (SELECT 1 WHERE ...)` 的形式写所有的 `EXISTS` 测试。不过这条规则有例外，例如那些使用 `INTERSECT` 的子查询。

下面这个简单的例子类似在 `col2` 上的一次内联接，但是它为每个 `tab1` 的行生成最多一个输出，即使存在多个匹配 `tab2` 的行也如此：

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

### 9.23.2. IN

`expression IN (subquery)`

右边是一个圆括弧括起来的子查询，它必须正好只返回一个列。左边表达式将被计算并与子查询结果逐行进行比较。如果找到任何等于子查询行的情况，那么 `IN` 的结果就是“真”。如果没有找到相等行，那么结果是“假”（包括子查询没有返回任何行的情况）。

请注意如果左边表达式得到空值，或者没有相等的右边值，并且至少有一个右边行得到空值，那么 `IN` 结构的结果将是空值，而不是假。这个行为是遵照 SQL 处理空值的一般规则的。

和 `EXISTS` 一样，假设子查询将被完成运行完全是不明智的。

row\_constructor IN (subquery)

这种形式的IN的左手边是一个行构造器，如Section 4.2.1中所述。右手边是一个圆括弧子查询，它必须返回和左手边返回的行中表达式所构成的完全一样多的列。左手边表达式将被计算并与子查询结果逐行进行比较。如果找到任意相等的子查询行，则IN的结果为“真”。如果没有找到相等行，那么结果为“假”（包括子查询不返回行的情况）。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这两行不等；否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等，要么是空值，并且至少有一个空值，那么IN的结果是空值。

### 9.23.3. NOT IN

expression NOT IN (subquery)

右手边是一个用圆括弧包围的子查询，它必须返回正好一个列。左手边表达式将被计算并与子查询结果逐行进行比较。如果只找到不相等的子查询行（包括子查询不返回行的情况），那么NOT IN的结果是“真”。如果找到任何相等行，则结果为“假”。

请注意如果左手边表达式得到空值，或者没有相等的右手边值，并且至少有一个右手边行得到空值，那么NOT IN结构的结果将是空值，而不是真。这个行为是遵照 SQL 处理空值的一般规则的。

和EXISTS一样，假设子查询会完全结束是不明智的。

row\_constructor NOT IN (subquery)

这种形式的NOT IN的左手边是一个行构造器，如Section 4.2.1中所述。右手边是一个圆括弧子查询，它必须返回和左手边返回的行中表达式所构成的完全一样多的列。左手边表达式将被计算并与子查询结果逐行进行比较。如果找到不等于子查询行的行，则NOT IN的结果为“真”。如果找到相等行，那么结果为“假”（包括子查询不返回行的情况）。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这两行不等；否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等，要么是空值，并且至少有一个空值，那么NOT IN的结果是空值。

### 9.23.4. ANY/SOME

expression operator ANY (subquery)

expression operator SOME (subquery)

这种形式的右手边是一个圆括弧括起来的子查询，它必须返回正好一个列。左手边表达式将被计算并使用给出的操作符对子查询结果逐行进行比较。如果获得任何真值结果，那么ANY的结果就是“真”。如果没有找到真值结果，那么结果是“假”（包括子查询没有返回任何行的情况）。

SOME是ANY的同义词。IN等价于= ANY。

请注意如果没有任何成功并且至少有一个右手边行为该操作符结果生成空值，那么ANY结构的结果将是空值，而不是假。这个行为是遵照 SQL 处理空值布尔组合的一般规则制定的。

和EXISTS一样，假设子查询将被完全运行是不明智的。

row\_constructor operator ANY (subquery)

row\_constructor operator SOME (subquery)

这种形式的左手边是一个行构造器，如Section 4.2.1所述。右手边是一个圆括弧括起来的子查询，它必须返回和左手边列表给出的表达式一样多的列。左手边表达式将被计算并使用给出的操作符对子查询结果逐行进行比较。如果比较为任何子查询行返回真，则ANY的结果为“真”。如果比较对每一个子查询行都返回假，则结果为“假”（包括子查询不返回行的情况）。如果比较不对任何行返回真并且至少对一行返回 NULL，则结果为 NULL。

关于行构造器比较的详细含义请见Section 9.24.5

## 9.23.5. ALL

expression operator ALL (subquery)

ALL 的这种形式的右手边是一个圆括弧括起来的子查询，它必须只返回一行。左手边表达式将被计算并使用给出的操作符对子查询结果逐行进行比较。该操作符必须生成布尔结果。如果所有行得到真（包括子查询没有返回任何行的情况），ALL的结果就是“真”。如果没有存在任何假值结果，那么结果是“假”。如果比较为任何行都不返回假并且对至少一行返回 NULL，则结果为 NULL。

NOT IN等价于<> ALL。

和EXISTS一样，假设子查询将被完全运行是不明智的。

row\_constructor operator ALL (subquery)

ALL的这种形式的左手边是一个行构造器，如Section 4.2.1所述。右手边是一个圆括弧括起来的子查询，它必须返回和左手边行中表达式一样多的列。左手边表达式将被计算并使用给出的操作符对子查询结果逐行进行比较。如果对所有子查询行该比较都返回真，那么ALL的结果就是“真”（包括子查询没有返回任何行的情况）。如果对任何子查询行比较返回假，则结果为“假”。如果比较对任何子查询行都不返回假并且对至少一行返回 NULL，则结果为 NULL。

关于行构造器比较的详细含义请见Section 9.24.5

## 9.23.6. 单一行比较

row\_constructor operator (subquery)

左手边是一个行构造器，如Section 4.2.1所述。右手边是一个圆括弧括起来的子查询，该查询必须返回和左手边行中表达式数目完全一样的列。另外，该子查询不能返回超过一行的数量（如果它返回零行，那么结果就是空值）。左手边被计算并逐行与右手边的子查询结果行比较。

关于行构造器比较的详细含义请见Section 9.24.5

## 9.24. 行和数组比较

本节描述几个特殊的结构，用于在值的组之间进行多重比较。这些形式语法上和前面一节的子查询形式相关，但是不涉及子查询。这种形式涉及的数组子表达式是LightDB的扩展；其它的是SQL兼容的。所有本节记录的表达式形式都返回布尔（Boolean）结果（真/假）。

### 9.24.1. IN

expression IN (value [, ...])

右手边是一个圆括弧包围的标量列表。如果左手边表达式的结果等于任何右手边表达式中的一个，结果为“真”。它是下面形式的缩写

```
expression = value1
OR
expression = value2
OR
...
```

请注意如果左手边表达式得到空值，或者没有相等的右手边值并且至少有一个右手边的表达式得到空值，那么IN结构的结果将为空值，而不是假。这符合 SQL 处理空值的布尔组合的一般规则。

## 9.24.2. NOT IN

```
expression NOT IN (value [, ...])
```

右手边是一个圆括弧包围的标量列表。如果左手边表达式的结果不等于所有右手边表达式，结果为“真”。它是下面形式的缩写

```
expression <> value1
AND
expression <> value2
AND
...
```

请注意如果左手边表达式得到空值，或者没有相等的右手边值并且至少有一个右手边的表达式得到空值，那么NOT IN结构的结果将为空值，而不是我们可能天真地认为的真值。这符合 SQL 处理空值的布尔组合的一般规则。

### Tip

$x \text{ NOT IN } y$  在所有情况下都等效于  $\text{NOT } (x \text{ IN } y)$ 。但是，在处理空值的时候，用NOT IN比用IN更可能迷惑新手。最好尽可能用正逻辑来表达你的条件。

## 9.24.3. ANY/SOME (array)

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

右手边是一个圆括弧包围的表达式，它必须得到一个数组值。左手边表达式被计算并且使用给出的操作符对数组的每个元素进行比较，这个操作符必须得到布尔结果。如果得到了任何真值结果，那么ANY的结果是“真”。如果没有找到真值结果（包括数组只有零个元素的情况），那么结果是“假”。

如果数组表达式得到一个空数组，ANY的结果将为空值。如果左手边的表达式得到空值，ANY通常是空值（尽管一个非严格比较操作符可能得到一个不同的结果）。另外，如果右手边的数组包含任何空值元素或者没有得到真值比较结果，ANY的结果将是空值而不是假（再次，假设是一个严格的比较操作符）。这符合 SQL 对空值的布尔组合的一般规则。

SOME是ANY的同义词。

## 9.24.4. ALL (array)

```
expression operator ALL (array expression)
```

右边是一个圆括弧包围的表达式，它必须得到一个数组值。左边表达式将被计算并使用给出的操作符与数组的每个元素进行比较，这个操作符必须得到一个布尔结果。如果所有比较都得到真值结果，那么ALL的结果是“真”（包括数组只有零个元素的情况）。如果有任何假值结果，那么结果是“假”。

如果数组表达式得到一个空数组，ALL的结果将为空值。如果左手边的表达式得到空值，ALL通常是空值（尽管一个非严格比较操作符可能得到一个不同的结果）。另外，如果右手边的数组包含任何空值元素或者没有得到假值比较结果，ALL的结果将是空值而不是真（再次，假设是一个严格的比较操作符）。这符合 SQL 对空值的布尔组合的一般规则。

## 9.24.5. 行构造器比较

`row_constructor operator row_constructor`

每一边都是一个行构造器，如Section 4.2.1所述。两个行值必须具有相同数量的域。每一边被计算并且被逐行比较。当操作符是 `=`、`<>`、`<`、`<=`、`>`、`>=`时，允许进行行构造器比较。每一个行元素必须是具有一个默认 B 树操作符类的类型，否则尝试比较会产生一个错误。

### Note

Errors related to the number or types of elements might not occur if the comparison is resolved using earlier columns.

`=`和`<>`情况略有不同。如果两行的所有对应成员都是非空且相等则这两行被认为相等；如果任何对应成员是非空但是不相等则这两行不相等；否则行比较的结果为未知（空值）。

对于`<`、`<=`、`>`和`>=`情况，行元素被从左至右比较，在找到一处不等的或为空的元素对就立刻停下来。如果这一对元素都为空值，则行比较的结果为未知（空值）；否则这一对元素的比较结果决定行比较的结果。例如，`ROW(1, 2, NULL) < ROW(1, 3, 0)`得到真，而不是空值，因为第三对元素并没有被考虑。

### Note

在LightDB 8.2之前，`<`、`<=`、`>`和`>=`情况不是按照每个 SQL 声明来处理的。一个像`ROW(a, b) < ROW(c, d)`的比较会被实现为`a < c AND b < d`，而结果行为等价于`a < c OR (a = c AND b < d)`。

`row_constructor IS DISTINCT FROM row_constructor`

这个结构与`<>`行比较相似，但是它对于空值输入不会得到空值。任何空值被认为和任何非空值不相等（有区别），并且任意两个空值被认为相等（无区别）。因此结果将总是为真或为假，永远不会是空值。

`row_constructor IS NOT DISTINCT FROM row_constructor`

这个结构与`=`行比较相似，但是它对于空值输入不会得到空值。任何空值被认为和任何非空值不相等（有区别），并且任意两个空值被认为相等（无区别）。因此结果将总是为真或为假，永远不会是空值。

## 9.24.6. 组合类型比较

`record operator record`

SQL 规范要求结果依赖于比较两个 NULL 值或者一个 NULL 与一个非 NULL 时逐行比较返回 NULL。LightDB 只有在比较两个行构造器（如 Section 9.24.5 的结果或者比较一个行构造器与一个子查询的输出时）才这样做（如 Section 9.24.5 所述）。在其他比较两个组合类型值的环境中，两个 NULL 域值被认为相等，并且一个 NULL 被认为大于一个非 NULL。为了得到组合类型一致的排序和索引行为，这样做是必要的。

每一边都会被计算并且它们会被逐行比较。当操作符是 =、<>、<、<=、> 或者 >= 时或者具有与这些类似的语义时，允许组合类型的比较（更准确地说，如果一个操作符是一个 B 树操作符类的成员，或者是一个 B 树操作符类的 = 成员的否定词，它就可以是一个行比较操作符）。上述操作符的行为与用于行构造器（见 Section 9.24.5 的 IS [ NOT ] DISTINCT FROM 相同）。

为了支持包含无默认 B 树操作符类的元素的行匹配，为组合类型比较定义了下列操作符：`*`、`*<>`、`*<`、`*<=`、`*>` 以及 `*>=`。这些操作符比较两行的内部二进制表达。即使两行用相等操作符的比较为真，两行也可能具有不同的二进制表达。行在这些比较操作符之下的排序是决定性的，其他倒没什么意义。这些操作符在内部被用于物化视图并且可能对其其他如复制和 B-树复制（参见 Section 60.4.2 之类的特殊功能有用，但是它们并不打算用在书写查询这类普通用途中）。

## 9.25. 集合返回函数

本节描述那些可能返回多于一行的函数。目前这个类中被使用最广泛的是级数生成函数，如 Table 9.62 和 Table 9.63 所述。其他更特殊的集合返回函数在本手册的其他地方描述。组合多集合返回函数的方法可见 Section 7.2.1.4

Table 9.62. 系列生成函数

函数	描述
<code>generate_series ( start integer, stop integer [, step integer ] )</code>	<code>→ setof integer</code>
<code>generate_series ( start bigint, stop bigint [, step bigint ] )</code>	<code>→ setof bigint</code>
<code>generate_series ( start numeric, stop numeric [, step numeric ] )</code>	<code>→ setof numeric</code>
	从 start 到 stop 生成一系列的值，步长为 step。step 默认为 1。
<code>generate_series ( start timestamp, stop timestamp, step interval )</code>	<code>→ setof timestamp</code>
<code>generate_series ( start timestamp with time zone, stop timestamp with time zone, step interval )</code>	<code>→ setof timestamp with time zone</code>
	从 start 到 stop 生成一系列的值，步长为 step。

当 step 为正时，如果 start 大于 stop 则返回零行。相反，当 step 为负时，如果 start 小于 stop 则返回零行。如果任何输入为 NULL 也会返回零行。step 为零是一个错误。下面是一些例子：

```
SELECT * FROM generate_series(2,4);
generate_series
-----
                2
                3
                4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
generate_series
-----
                5
```

```

          3
          1
(3 rows)

SELECT * FROM generate_series(4, 3);
generate_series
-----
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
generate_series
-----
          1.1
          2.4
          3.7
(3 rows)

-- this example relies on the date-plus-integer operator:
SELECT current_date + s.a AS dates FROM generate_series(0, 14, 7) AS s(a);
dates
-----
2004-02-05
2004-02-12
2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
generate_series
-----
2008-03-01 00:00:00
2008-03-01 10:00:00
2008-03-01 20:00:00
2008-03-02 06:00:00
2008-03-02 16:00:00
2008-03-03 02:00:00
2008-03-03 12:00:00
2008-03-03 22:00:00
2008-03-04 08:00:00
(9 rows)

```

Table 9.63. 下标生成函数

函数	描述
<code>generate_subscripts ( array anyarray, dim integer ) → setof integer</code>	生成一个包含给定数组第dim维度的有效下标的序列。
<code>generate_subscripts ( array anyarray, dim integer, reverse boolean ) → setof integer</code>	生成一个包含给定数组第dim维度的有效下标的序列。当reverse为真时，以相反的顺序返回序列。

`generate_subscripts`是一个快捷函数，它为给定数组的指定维度生成一组合法的下标。对于不具有请求维度的数组返回零行，对于任何输入为NULL数组也返回零行。下面是一些例子：

```
-- basic usage:
```



```

SELECT generate_subscripts(' {NULL,1,NULL,2}'::int[], 1) AS s;
s
---
1
2
3
4
(4 rows)

-- presenting an array, the subscript and the subscripted
-- value requires a subquery:
SELECT * FROM arrays;
      a
-----
{-1,-2}
{100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
 array | subscript | value
-----+-----+-----
{-1,-2} |          1 |    -1
{-1,-2} |          2 |    -2
{100,200,300} |          1 |   100
{100,200,300} |          2 |   200
{100,200,300} |          3 |   300
(5 rows)

-- unnest a 2D array:
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
      from generate_subscripts($1,1) g1(i),
           generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
unnest2
-----
1
2
3
4
(4 rows)

```

当FROM子句中的函数以WITH ORDINALITY作为后缀时，将在函数的输出列上附加一个bigint列，该列从1开始，函数输出的每一行加1。这在 unnest() 等集合返回函数的情况下最有用。

```

-- set returning function WITH ORDINALITY:
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
      ls | n
-----+---
lt_serial | 1
lt_twophase | 2
lightdb.opts | 3

```

lt_notify		4
lightdb.conf		5
lt_tblspc		6
logfile		7
base		8
lightdb.pid		9
lt_ident.conf		10
global		11
lt_xact		12
lt_snapshots		13
lt_multixact		14
PG_VERSION		15
lt_wal		16
lt_hba.conf		17
lt_stat_tmp		18
lt_subtrans		19
(19 rows)		

## 9.26. 系统信息函数和运算符

Table 9.6展示了多个可以抽取会话和系统信息的函数。

除了本节列出的函数，还有一些与统计系统相关的函数也提供系统信息。详见Section 25.2.2

### 9.26.1. 会话信息函数

Table 9.64. 会话信息函数

函数	描述
current_catalog → name current_database () → name	返回当前数据库的名称。(在SQL标准中数据库被称为“catalogs”，因此current_catalog是该标准的拼写方式。)
current_query () → text	返回当前所执行查询的文本，由客户端提交的(可能包含一个以上的语句)。
current_role → name	这个等同于 current_user。
current_schema → name current_schema () → name	返回在搜索路径中的第一个模式的名称(如果搜索路径为空则返回空值)。这个模式将用于没有指定目标模式就创建的任何表或其他已命名对象。
current_schemas ( include_implicit boolean ) → name[]	返回当前在有效搜索路径中的所有模式的名称的数组，以优先级顺序。(当前 search_path设置中与已存在的、可搜索模式不相符的项将被省略。) 如果布尔参数为true，则类似pg_catalog的隐式搜索的系统模式将包含在结果中。
current_user → name	返回当前执行上下文的用户名。
inet_client_addr () → inet	返回当前客户端的IP地址，如果当前连接是通过Unix-域套接字则返回NULL
inet_client_port () → integer	返回当前客户端的IP端口号，如果当前连接是通过Unix-域套接字则返回NULL。
inet_server_addr () → inet	

函数	描述
	返回服务器接受当前连接的IP地址，如果当前连接是通过Unix-域套接字则返回NULL。
<code>inet_server_port ()</code> → integer	返回服务器接受当前连接的IP端口号，如果当前连接是通过Unix-域套接字则返回NULL。
<code>pg_backend_pid ()</code> → integer	返回附加到当前会话的服务器进程的进程ID。
<code>pg_blocking_pids ( integer )</code> → integer[]	返回阻止服务器进程的会话的进程ID数组，该进程ID与指定的进程ID一起获取锁定，如果没有这样的服务器进程或者没有被阻塞，则返回一个空数组。 如果一个服务器进程持有一个与被阻塞进程的锁请求冲突的锁(硬阻塞)，或者正在等待一个与被阻塞进程的锁请求冲突并且在等待队列中位于其前面的锁(软阻塞)，那么这个服务器进程就会阻塞另一个服务器进程。当使用并行查询时结果总是列出客户端可见的进程ID(即 <code>pg_backend_pid</code> 的结果)，即使实际的锁是由子工作进程持有或等待的。因此，结果中可能存在重复的pid。还要注意当准备好的事务持有冲突锁时，它将用零进程ID表示。 频繁调用这个函数可能会对数据库性能产生一些影响，因为它需要在短时间内独占访问锁管理器的共享状态。
<code>pg_conf_load_time ()</code> → timestamp with time zone	返回服务器配置文件最后加载的时间。如果当前会话当时是活跃的，那么这将是会话本身重新读取配置文件的时间(因此在不同的会话中读取会稍有不同)。否则，就是postmaster进程重新读取配置文件的时间。
<code>pg_current_logfile ( [ text ] )</code> → text	返回日志采集器当前使用的日志文件的路径名。该路径包括log_directory目录和单个日志文件名。如果日志采集器被禁用，结果为NULL。当存在多个日志文件时，每个文件的格式都不同，不带参数的 <code>pg_current_logfile</code> 将返回在有序列表中的第一种格式的文件路径:stderr, csvlog。如果没有日志文件具有任何这些格式，则返回NULL。要请求关于特定日志文件格式的信息，可以提供csvlog 或 stderr作为可选参数的值。如果在log_destination中没有配置需要的日志格式，则结果为NULL。结果反映了current_logfiles文件的内容。
<code>pg_my_temp_schema ()</code> → oid	返回当前会话的临时模式的OID，如果没有则返回0(因为它没有创建任何临时表)。
<code>pg_is_other_temp_schema ( oid )</code> → boolean	如果给定的OID是另一个会话的临时模式的OID则返回真。(这可能是有用的，例如，在目录显示中排除其他会话的临时表。)
<code>pg_jit_available ()</code> → boolean	如果JIT编译器扩展可用(参见Chapter 29，并且jit配置参数设置为on，则返回真。
<code>pg_listening_channels ()</code> → setof text	返回当前会话正在侦听的异步通知通道的名称集。
<code>pg_notification_queue_usage ()</code> → double precision	返回当前被等待处理的通知所占用的异步通知队列最大尺寸的分数(0 - 1)。更多信息请参见LISTEN 和 NOTIFY。
<code>pg_postmaster_start_time ()</code> → timestamp with time zone	返回服务器启动时的时间。
<code>pg_safe_snapshot_blocking_pids ( integer )</code> → integer[]	返回一个进程ID数组，该进程ID是阻塞服务器进程获取安全快照的会话的进程ID数组，如果没有这样的服务器进程或者没有阻塞，则返回一个空数组。

函数	描述
	运行SERIALIZABLE事务的会话会阻止SERIALIZABLE READ ONLY DEFERRABLE事务获取快照，直到后者确定可以安全地避免获取谓词锁。关于可序列化和可延迟事务的更多信息，请参见Section 13.2.3 频繁调用这个函数可能会对数据库性能产生一些影响，因为它需要在短时间内访问谓词锁管理器的共享状态。
<code>pg_trigger_depth ()</code>	<code>→ integer</code> 返回当前嵌套层次的LightDB触发器(如果没有调用则为 0，直接或间接，从一个触发器内部开始)。
<code>session_user</code>	<code>→ name</code> 返回会话用户名。
<code>user</code>	<code>→ name</code> 这个相当于 <code>current_user</code> 。
<code>version ()</code>	<code>→ text</code> 返回描述LightDB服务器的版本的字符串。你还可以从 <code>server_version</code> 中获得此信息，或者对于机器可读的版本，使用 <code>server_version_num</code> 。软件开发人员可以使用 <code>server_version_num</code> (从8.2起可用)或 <code>PQserverVersion</code> ，而不是解析文本版本。

### Note

`current_catalog`、`current_role`、`current_schema`、`current_user`、`session_user`和`user`在SQL里有特殊的语意状态：它们被调用时结尾不要跟着圆括号。在LightDB中，圆括号可以有选择性地被用于`current_schema`，但是不能和其他的一起用。

`session_user`通常是发起当前数据库连接的用户，不过超级用户可以用`SET SESSION AUTHORIZATION`修改这个设置。`current_user`是用于权限检查的用户标识。通常，它总是等于会话用户，但是可以被`SET ROLE`改变。它也会在函数执行的过程中随着属性`SECURITY DEFINER`的改变而改变。在Unix的说法里，那么会话用户是“真实用户”，而当前用户是“有效用户”。`current_role`以及`user`是`current_user`的同义词（SQL标准在`current_role`和`current_user`之间做了区分，但LightDB不区分，因为它把用户和角色统一成了一种实体）。

Table 9.6列出那些允许编程查询对象访问权限的函数。参阅Section 5. 获取更多有关权限的信息。在这些函数中，可以通过名称或OID (`pg_authid.oid`)指定被查询权限的用户，或者如果名称被指定为`public`，则检查`PUBLIC`伪角色的权限。同样，`user`参数可以完全省略，在这种情况下，假设为`current_user`。被查询的对象也可以通过名称或OID来指定。通过名称指定时，可以包含相关的模式名称。感兴趣的访问权限由一个文本字符串指定，它必须计算为对象类型的一个适当的权限关键字(例如，`SELECT`)。还可以将`WITH GRANT OPTION`添加到特权类型中，以测试该特权是否由授予选项持有。同样，可以用逗号分隔列出多个特权类型，在这种情况下，如果所列出的特权中有任何一个被持有，结果将为真。(特权字符串的大小写不重要，特权名之间允许有额外的空格，但在特权名中不允许。)一些例子：

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT
OPTION');
```

## 9.26.2. 访问权限查询函数

Table 9.65. 访问权限查询函数

函数	描述
<code>has_any_column_privilege</code> ( [ user name or oid, ] table text or oid, privilege text ) → boolean	用户是否对表的任何列有权限? 如果对整个表持有特权, 或者对至少一个列有列级的特权授予, 则会成功。允许的权限类型为SELECT, INSERT, UPDATE, 和 REFERENCES
<code>has_column_privilege</code> ( [ user name or oid, ] table text or oid, column text or smallint, privilege text ) → boolean	用户对指定的表列有特权么? 如果对整个表持有特权, 或者对列授予了列级别的特权, 则会成功。可以通过名称或属性编号 (pg_attribute.attnum) 指定列。允许的特权类型为SELECT, INSERT, UPDATE, 和 REFERENCES。
<code>has_database_privilege</code> ( [ user name or oid, ] database text or oid, privilege text ) → boolean	用户对数据库有特权吗? 允许的特权类型为CREATE, CONNECT, TEMPORARY, 和TEMP (相当于 TEMPORARY)。
<code>has_foreign_data_wrapper_privilege</code> ( [ user name or oid, ] fdw text or oid, privilege text ) → boolean	用户是否拥有外部数据包装的特权? 唯一允许的特权类型是USAGE。
<code>has_function_privilege</code> ( [ user name or oid, ] function text or oid, privilege text ) → boolean	用户对函数有特权吗? 唯一允许的特权类型是EXECUTE。 当通过名称而不是OID指定函数时, 允许的输入与regprocedure数据类型相同 (参见Section 8.19)。一个例子为:  <pre>SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');</pre>
<code>has_language_privilege</code> ( [ user name or oid, ] language text or oid, privilege text ) → boolean	用户对语言有特权吗? 唯一允许的特权类型是USAGE。
<code>has_schema_privilege</code> ( [ user name or oid, ] schema text or oid, privilege text ) → boolean	用户对模式有特权吗? 允许的特权类型是CREATE 和USAGE。
<code>has_sequence_privilege</code> ( [ user name or oid, ] sequence text or oid, privilege text ) → boolean	用户是否有顺序特权? 允许的特权类型有USAGE, SELECT, 和UPDATE。
<code>has_server_privilege</code> ( [ user name or oid, ] server text or oid, privilege text ) → boolean	用户是否对外部服务器有特权? 唯一允许的特权类型是USAGE。
<code>has_table_privilege</code> ( [ user name or oid, ] table text or oid, privilege text ) → boolean	用户对表有特权吗? 允许的特权类型有SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, 和 TRIGGER。
<code>has_tablespace_privilege</code> ( [ user name or oid, ] tablespace text or oid, privilege text ) → boolean	用户对表空间有特权吗? 唯一允许的特权类型是CREATE。
<code>has_type_privilege</code> ( [ user name or oid, ] type text or oid, privilege text ) → boolean	

函数	描述
	用户对数据类型有特权吗?唯一允许的特权类型是 USAGE。当通过名称而不是OID指定类型时, 允许的输入与regtype数据类型相同(参见Section 8.19)。
pg_has_role ( [ user name or oid, ] role text or oid, privilege text ) → boolean	用户对角色有特权么?允许的特权类型是MEMBER 和 USAGE。MEMBER表示角色中的直接或间接成员关系(即执行SET ROLE的权利), 而USAGE表示不执行SET ROLE情况下是否立即可用角色的特权。此函数不允许特殊情况下将user设置为public, 因为PUBLIC伪角色永远不能成为真实角色的成员。
row_security_active ( table text or oid ) → boolean	在当前用户和当前环境的上下文之中, 指定表的行级安全是活动的吗?

### 9.26.3. aclitem Operators

Table 9.66显示了aclitem类型的可用操作符, 它是访问权限的目录表示。有关如何读取访问权限值的信息, 请参阅 Section 5.7

Table 9.66. aclitem 操作符

操作符	描述 例子
aclitem = aclitem → boolean	aclitem相等吗?(注意, aclitem类型缺少比较操作符的通常集合;它只有相等。反之, aclitem数组只能进行相等比较。) 'calvin=r*w/hobbes'::aclitem = 'calvin=r*w*/hobbes'::aclitem → f
aclitem[] @> aclitem → boolean	数组是否包含指定的特权?(如果有一个数组条目与aclitem的被授权人和授予人相匹配, 并且至少具有特权的指定集, 则此选项为真。) '{calvin=r*w/hobbes, hobbes=r*w*/postgres}'::aclitem[] @> 'calvin=r*/hobbes'::aclitem → t
aclitem[] ~ aclitem → boolean	这是@>的已弃用别名。 '{calvin=r*w/hobbes, hobbes=r*w*/postgres}'::aclitem[] ~ 'calvin=r*/hobbes'::aclitem → t

Table 9.67显示了一些额外的函数来管理aclitem类型。

Table 9.67. aclitem 函数

函数	描述
acldefault ( type "char", ownerId oid ) → aclitem[]	构造一个aclitem数组, 该数组持有type类型对象的默认访问特权, 该对象属于OID为ownerId的角色。这表示当对象的ACL条目为空时所假定的访问特权。(默认访问特权在Section 5.7中描述。) type 参数必须是下列中的一个 'c' 对应 COLUMN, 'r' 对应 TABLE 和类表对象, 's' 对应 SEQUENCE, 'd' 对应 DATABASE, 'f' 对应 FUNCTION 或 PROCEDURE, 'l' 对应 LANGUAGE, 'L' 对应 LARGE OBJECT, 'n' 对应 SCHEMA, 't' 对应 TABLESPACE, 'F' 对应 FOREIGN DATA WRAPPER, 'S' 对应 FOREIGN SERVER, 或 'T' 对应 TYPE 或 DOMAIN。
aclxplode ( aclitem[] ) → setof record ( grantor oid, grantee oid, privilege_type text, is_grantable boolean )	

函数	描述
	以行集的形式返回aclitem数组。如果受让人是伪角色PUBLIC，则在grantee列中用0表示。每个被授予的特权都表示为SELECT, INSERT等。注意，每个特权被分割成单独的一行，因此在privilege_type列中只出现一个关键字。
makeaclitem ( grantee oid, grantor oid, privileges text, is_grantable boolean ) → aclitem	使用给定的属性构造 aclitem 。

## 9.26.4. 架构可见性查询函数

Table 9.6展示了决定是否一个特定对象在当前模式搜索路径中可见的函数。例如，如果一个表所在的模式在当前搜索路径中并且在它之前没有出现过相同的名字，这个表就被说是可见的。这等价于在语句中表可以被用名称引用但不加显式的模式限定。因此，要列出所有可见表的名字：

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

对于函数和操作符，如果路径前面没有相同名称and argument data type(s)的对象，那么搜索路径中的对象就是可见的。对于操作符类和操作符族，要考虑名称和关联的索引访问方法。

Table 9.68. 模式可见性查询函数

函数	描述
pg_collation_is_visible ( collation oid ) → boolean	排序规则在搜索路径中可见吗？
pg_conversion_is_visible ( conversion oid ) → boolean	转换在搜索路径中可见吗？
pg_function_is_visible ( function oid ) → boolean	函数在搜索路径中可见吗？（这也适用于过程和聚合。）
pg_opclass_is_visible ( opclass oid ) → boolean	操作符类在搜索路径中可见吗？
pg_operator_is_visible ( operator oid ) → boolean	操作符在搜索路径中可见吗？
pg_opfamily_is_visible ( opclass oid ) → boolean	操作符族在搜索路径中可见吗？
pg_statistics_obj_is_visible ( stat oid ) → boolean	统计对象在搜索路径中可见吗？
pg_table_is_visible ( table oid ) → boolean	表在搜索路径中可见吗？（这适用于所有类型的关系，包括视图、物化视图、索引、序列和外部表。）
pg_ts_config_is_visible ( config oid ) → boolean	文本搜索配置在搜索路径可见吗？
pg_ts_dict_is_visible ( dict oid ) → boolean	文本搜索字典在搜索路径可见吗？
pg_ts_parser_is_visible ( parser oid ) → boolean	文本搜索解析器在搜索路径中可见吗？
pg_ts_template_is_visible ( template oid ) → boolean	文本搜索模板在搜索路径可见吗？

函数	描述
<code>pg_type_is_visible ( type oid )</code>	<code>→ boolean</code> 类型(或域)在搜索路径中可见吗?

所有这些函数都要求用对象 OID 来标识将被检查的对象。如果你想用名称来测试一个对象，使用 OID 别名类型 (regclass、regtype、regprocedure、regoperator、regconfig或regdictionary) 将会很方便。例如：

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

注意以这种方式测试一个非模式限定的类型名没什么意义 — 如果该名称完全能被识别，它必须是可见的。

## 9.26.5. 系统目录信息函数

Table 9.69列出从系统目录中提取信息的函数。

Table 9.69. 系统目录信息函数

函数	描述
<code>format_type ( type oid, typemod integer )</code>	<code>→ text</code> 返回由其类型OID和可能的类型修饰符标识的数据类型的SQL名称。如果没有已知的类型修饰符，则传递NULL值给类型修饰符。
<code>pg_get_constraintdef ( constraint oid [, pretty boolean ] )</code>	<code>→ text</code> 重构为了约束的创建命令。(这是一个反编译的重构，而不是命令的原始文本。)
<code>pg_get_expr ( expr pg_node_tree, relation oid [, pretty boolean ] )</code>	<code>→ text</code> 反编译存储在系统目录中的表达式的内部形式，例如列的默认值。如果表达式可能包含变量，则指定它们所指向的关系的OID作为第二个参数；如果没有预期的变量，传递0就可以了。
<code>pg_get_functiondef ( func oid )</code>	<code>→ text</code> 重构为了函数或过程的创建命令。(这是一个反编译的重构，而不是命令的原始文本。) 结果是一个完整的CREATE OR REPLACE FUNCTION 或 CREATE OR REPLACE PROCEDURE语句。
<code>pg_get_function_arguments ( func oid )</code>	<code>→ text</code> 重新构造函数或过程的参数列表，以其在 CREATE FUNCTION里面需要出现的形式(包括默认值)。
<code>pg_get_function_identity_arguments ( func oid )</code>	<code>→ text</code> 重新构造标识函数或过程所需的参数列表，以其应出现在ALTER FUNCTION等命令中的形式。这个表单省略默认值。
<code>pg_get_function_result ( func oid )</code>	<code>→ text</code> 重构函数的RETURNS子句，以其需要出现在CREATE FUNCTION中的形式。对于过程，返回NULL。
<code>pg_get_indexdef ( index oid [, column integer, pretty boolean ] )</code>	<code>→ text</code> 重构针对索引的创建命令。(这是一个反编译的重构，而不是命令的原始文本。) 如果提供了column而且不为零，则只重构该列的定义。
<code>pg_get_keywords ()</code>	<code>→ setof record ( word text, catcode "char", catdesc text )</code> 返回一组描述服务器识别的SQL关键字的记录。word列包含关键字。 catcode列包含一个类别代码：U表示无保留关键字，C表示可以是列名的关键字，T表示可以是类型或函数名的关键字，或者R表示完全保留关键字。 catdesc列包含描述类别的可能本地化字符串。



函数	描述
<code>pg_get_ruledef ( rule oid [, pretty boolean ] ) → text</code>	重构针对规则的创建命令。(这是一个反编译的重构，而不是命令的原始文本。)
<code>pg_get_serial_sequence ( table text, column text ) → text</code>	返回与列相关联的序列名称，如果没有序列与该列相关联则返回NULL。如果列是标识列，则关联序列是在内部为该列创建的序列。对于使用一种串行类型 ( <code>serial</code> , <code>smallserial</code> , <code>bigserial</code> ) 创建的列，它是为该串行列定义创建的序列。在后一种情况下，可以使用 <code>ALTER SEQUENCE OWNED BY</code> 修改或删除关联。(这个函数可能应该被称为 <code>pg_get_owned_sequence</code> ; 它的当前名称反映了它在历史上曾与串行类型的列一起使用。) 第一个参数是具有可选模式的表名，第二个参数是列名。由于第一个参数可能包含模式名和表名，因此按照通常的SQL规则解析它，这意味着默认情况下它是小写的。第二个参数只是一个列名，按照字面来处理，因此保留了它的大小写。结果经过了适当的格式化，可以传递给序列函数 (参见Section 9.17)。典型的用法是读取序列的当前值以获取标识或串行列，示例如下：  <pre>SELECT currval(pg_get_serial_sequence('sometable', 'id'));</pre>
<code>pg_get_statisticsobjdef ( statobj oid ) → text</code>	重构针对扩展统计对象的创建命令。(这是一个反编译的重构，而不是命令的原始文本。)
<code>pg_get_triggerdef ( trigger oid [, pretty boolean ] ) → text</code>	重构针对触发器的创建命令。(这是一个反编译的重构，而不是命令的原始文本。)
<code>pg_get_userbyid ( role oid ) → name</code>	根据OID返回角色名。
<code>pg_get_viewdef ( view oid [, pretty boolean ] ) → text</code>	重构针对视图或物化视图的SELECT命令。(这是一个反编译的重构，而不是命令的原始文本。)
<code>pg_get_viewdef ( view oid, wrap_column integer ) → text</code>	重构针对视图或物化视图的底层SELECT命令。(这是一个反编译的重构，而不是命令的原始文本。) 在这种形式的函数中，总是启用美观打印，并对长行进行换行，以尽量使它们小于指定的列数。
<code>pg_get_viewdef ( view text [, pretty boolean ] ) → text</code>	根据视图的文本名称而不是它的OID，重构针对视图或物化视图的底层SELECT命令。(这是弃用; 请使用OID变体。)
<code>pg_index_column_has_property ( index regclass, column integer, property text ) → boolean</code>	测试一个索引列是否具有命名属性。Table 9.7列出了常用索引列属性。(注意，扩展访问方法可以为索引定义额外的属性名。) 如果属性名未知或不适用于特定对象，或者OID或列号不能识别有效的对象，则返回NULL。
<code>pg_index_has_property ( index regclass, property text ) → boolean</code>	测试一个索引是否具有命名属性。Table 9.7列出了常用的索引属性。(注意，扩展访问方法可以为索引定义额外的属性名。) 如果属性名未知或不适用于特定对象，或者OID不能识别有效的对象，则返回NULL。
<code>pg_indexam_has_property ( am oid, property text ) → boolean</code>	测试索引访问方法是否具有命名属性。访问方法属性如Table 9.7所示。如果属性名未知或不适用于特定对象，或者OID不能识别有效的对象，则返回NULL。
<code>pg_options_to_table ( options_array text[] ) → setof record ( option_name text, option_value text )</code>	返回源自 <code>pg_class.reloptions</code> 或 <code>pg_attribute.attoptions</code> 的值表示的存储选项集。

函数	描述
<code>pg_tablespace_databases ( tablespace oid ) → setof oid</code>	返回具有存储在指定表空间中的对象的数据库的OIDs集。如果这个函数返回了任何行，那么表空间就不是空的，且不能被删除。要识别填充表空间的特定对象，需要连接到由 <code>pg_tablespace_databases</code> 标识的数据库，并查询它们的 <code>pg_class</code> 目录。
<code>pg_tablespace_location ( tablespace oid ) → text</code>	返回表空间所在的文件系统路径。
<code>pg_typeof ( "any" ) → regtype</code>	<p>返回传递值给它的数据类型的OID。这对于故障排除或动态构造SQL查询很有帮助。函数声明为返回<code>regtype</code>，它是一个OID别名类型(see Section 8.19)；这意味着，为了比较，它与OID相同，但显示为类型名。</p> <p>例如：</p> <pre>SELECT pg_typeof(33);  pg_typeof -----  integer  SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);  typlen -----       4</pre>
<code>COLLATION FOR ( "any" ) → text</code>	<p>返回传递值给它的排序规则的名称。如果需要，该值会被引号括起来，并使用模式限定。如果没有为参数表达式派生排序规则，则返回NULL。如果参数不是可排序数据类型，则会引发错误。</p> <p>例如：</p> <pre>SELECT collation for (description) FROM pg_description LIMIT 1;  pg_collation_for -----  "default"  SELECT collation for ('foo' COLLATE "de_DE");  pg_collation_for -----  "de_DE"</pre>
<code>to_regclass ( text ) → regclass</code>	将文本关系名转换为它的OID。通过将字符串类型转换为 <code>regclass</code> 可以得到类似的结果(参见Section 8.19)；但是，如果没有找到名称，这个函数将返回NULL而不会抛出错误。与强制转换不同的是，它不接受数字OID作为输入。
<code>to_regcollation ( text ) → regcollation</code>	将文本排序规则名称转换为它的OID。通过将字符串类型转换为 <code>regcollation</code> (参见Section 8.19)可以得到类似的结果；但是，如果没有找到名称，这个函数将返回NULL而不会抛出错误。与强制转换不同的是，它不接受数字OID作为输入。
<code>to_renamespace ( text ) → regnamespace</code>	将文本模式名转换为它的OID。通过将字符串转换为 <code>regnamespace</code> 类型(参见Section 8.19)可以得到类似的结果；但是，如果没有找到名称，这个函数将返回NULL而不会抛出错误。与强制转换不同的是，它不接受数字OID作为输入。
<code>to_regoper ( text ) → regoper</code>	

函数	描述
	将文本操作符名称转换为它的OID。通过将字符串类型转换为regoper(参见Section 8.19)可以得到类似的结果;但是,如果找不到名称或名称有多义性,该函数将返回NULL而不会抛出错误。与强制转换不同的是,它不接受数字OID作为输入。
to_regoperator ( text ) → regoperator	将文本操作符名称(带有参数类型)转换为其OID。通过将字符串转换为regoperator类型(参见Section 8.19)可以得到类似的结果;但是,如果没有找到名称,这个函数将返回NULL而不会抛出错误。与强制转换不同的是,它不接受数字OID作为输入。
to_regproc ( text ) → regproc	将文本函数或过程名转换为其OID。通过将字符串转换为regproc类型(参见Section 8.19)可以得到类似的结果;但是,如果找不到名称或名称有多义性,该函数将返回NULL而不会抛出错误。与强制转换不同的是,它不接受数字OID作为输入。
to_regprocedure ( text ) → regprocedure	将文本函数或过程名(带有参数类型)转换为其OID。通过将字符串类型转换为regprocedure可以得到类似的结果(参见Section 8.19);但是,如果没有找到名称,这个函数将返回NULL而不会抛出错误。与强制转换不同的是,它不接受数字OID作为输入。
to_regrole ( text ) → regrole	将文本角色名转换为它的OID。通过将字符串类型转换为regrole可以得到类似的结果(参见Section 8.19);但是,如果没有找到名称,这个函数将返回NULL而不会抛出错误。与强制转换不同的是,它不接受数字OID作为输入。
to_regtype ( text ) → regtype	将文本类型名转换为它的OID。通过将字符串类型转换为regtype可以得到类似的结果(参见Section 8.19);但是,如果没有找到名称,这个函数将返回NULL而不会抛出错误。与强制转换不同的是,它不接受数字OID作为输入。

大多数重构(反编译)数据库对象的函数都有一个可选的 `pretty` 标志,如果为true,结果将被“pretty-printed”。`pretty` 美观打印会抑制不必要的圆括号,并为易读性增加空格。美观打印的格式可读性更好,但是默认格式更有可能被LightDB的未来版本以同样的方式解释;因此,避免为转储目的使用美观打印的输出。为`pretty`参数传递false会产生与省略参数相同的结果。

## 9.26.6. 索引列属性

Table 9.70. 索引列属性

名称	描述
asc	在向前扫描时列是按照升序排列吗?
desc	在向前扫描时列是按照降序排列吗?
nulls_first	在向前扫描时列排序会把空值排在前面吗?
nulls_last	在向前扫描时列排序会把空值排在最后吗?
orderable	列具有已定义的排序顺序吗?
distance_orderable	列能否通过一个“distance”操作符(例如ORDER BY col <-> constant)有序地扫描?
returnable	列值是否可以通过一次只用索引扫描返回?
search_array	列是否天然支持col = ANY(array)搜索?
search_nulls	列是否支持IS NULL和IS NOT NULL搜索?

## 9.26.7. 索引性质

Table 9.71. 索引性质

名称	描述
clusterable	索引是否可以用于CLUSTER命令?
index_scan	索引是否支持普通扫描(非位图)?
bitmap_scan	索引是否支持位图扫描?
backward_scan	在扫描中扫描方向能否被更改(为了支持游标上无需物化的FETCH BACKWARD)?

## 9.26.8. 索引访问方法性质

Table 9.72. 索引访问方法性质

名称	描述
can_order	访问方法是否支持ASC、DESC以及CREATE INDEX中的有关关键词?
can_unique	访问方法是否支持唯一索引?
can_multi_col	访问方法是否支持多列索引?
can_exclude	访问方法是否支持排除约束?
can_include	访问方法是否支持CREATE INDEX的INCLUDE子句?

## 9.26.9. 对象信息和寻址函数

Table 9.7列出了与数据库对象标识和定位有关的函数。

Table 9.73. 对象信息和定位函数

函数	描述
<code>pg_describe_object ( classid oid, objid oid, objsubid integer ) → text</code>	返回由目录OID、对象OID和子对象ID(例如表中的列号)标识的数据库对象的文本描述;当引用整个对象时,子对象ID为0)。这个描述是人类可读的,并且可以根据服务器配置进行翻译。这对于决定pg_depend目录中引用的对象的标识特别有用。
<code>pg_identify_object ( classid oid, objid oid, objsubid integer ) → record ( type text, schema text, name text, identity text )</code>	返回包含足够信息的行以唯一标识由目录OID、对象OID和子对象ID指定的数据库对象。这些信息是为了机器可读的,永远不会被翻译。type标识数据库对象的类型;schema是对象所属的模式名, NULL表示不属于模式的对象类型;name是对象的名称,如果有必要,用引号括起来,如果名称(随着模式名称,如果相关)足以唯一地标识对象,否则为NULL;identity是完整的对象标识,其精确格式依赖于对象类型,格式中的每个名称都是模式限定的,并在必要时用引号括起来。
<code>pg_identify_object_as_address ( classid oid, objid oid, objsubid integer ) → record ( type text, object_names text[], object_args text[] )</code>	返回包含足够信息的行以唯一标识由目录OID、对象OID和子对象ID指定的数据库对象。返回的信息独立于当前服务器,也就是说,它可以用于标识另一个服务器中具有相同名称的对象。type标识数据库对象的类型;object_names和object_args是文本数组,它们一起构成对对象的引用。这三个值可以传递给pg_get_object_address以获得对象的内部地址。

函数	描述
<code>pg_get_object_address ( type text, object_names text[], object_args text[] )</code> <code>→ record ( classid oid, objid oid, objsubid integer )</code>	返回包含足够信息的行以唯一标识由类型代码、对象名称和参数数组指定的数据库对象。返回的值将在系统目录中使用，例如 <code>pg_depend</code> ；它们可以传递给其他系统函数，比如 <code>pg_describe_object</code> 或 <code>pg_identify_object</code> 。 <code>classid</code> 是包含该对象的系统目录的OID； <code>objid</code> 是对象本身的OID， <code>objsubid</code> 是子对象的ID，如果没有则为零。这个函数是 <code>pg_identify_object_as_address</code> 的反向函数。

## 9.26.10. 注释信息函数

Table 9.7中展示的函数抽取注释，注释是由`COMMENT`命令在以前存储的。如果对指定参数找不到注释，则返回空值。

Table 9.74. 注释信息函数

函数	描述
<code>col_description ( table oid, column integer ) → text</code>	返回表列的注释，该注释由该表的OID和列号指定。(obj_description不能用于表的列，因为列没有自己的oid。)
<code>obj_description ( object oid, catalog name ) → text</code>	返回OID指定的数据库对象的注释和包含该对象的系统目录的名称。例如， <code>obj_description(123456, 'pg_class')</code> 将检索OID为123456的表的注释。
<code>obj_description ( object oid ) → text</code>	返回仅由其OID指定的数据库对象的注释。这个已被弃用(deprecated)因为无法保证oid在不同的系统目录中是唯一的；因此，可能会返回错误的注释。
<code>shobj_description ( object oid, catalog name ) → text</code>	返回共享数据库对象的注释，该对象由其OID和包含的系统编目的名称指定。这与 <code>obj_description</code> 类似，只是它用于检索共享对象(也就是数据库、角色和表空间)上的注释。有些系统编目对每个集群中的所有数据库都是全局的，其中对象的描述也全局存储。

## 9.26.11. 事务 ID 和快照信息函数

Table 9.7中展示的函数以一种可导出的形式提供了服务器事务信息。这些函数的主要用途是判断在两个快照之间哪些事务被提交。

Table 9.75. 事务ID和快照信息功能

函数	描述
<code>pg_current_xact_id () → xid8</code>	返回当前事务的ID。如果当前事务还没有一个ID(因为它还没有执行任何数据库更新)，它将分配一个新的事务。
<code>pg_current_xact_id_if_assigned () → xid8</code>	返回当前事务的ID，如果还没有分配ID则返回NULL。(如果事务可能是只读的，最好使用这种变体，以避免不必要地消耗XID。)
<code>pg_xact_status ( xid8 ) → text</code>	报告最近的事务的提交状态。如果事务为最近的，系统会保留事务的提交状态，则结果是 <code>in progress</code> 、 <code>committed</code> 或 <code>aborted</code> 。如果该事务的时间足够久，并且系统中没有对该事务的引用，而且提交状态信息已经被丢弃，则结果为NULL。应用可以使用此函数，例如，确定在进行 <code>COMMIT</code> 时，应用程序和数据库服务器

函数	描述
	断开连接后，它们的事务是已提交还是中止。注意，准备好的事务报告为in progress的事务；如果应用需要确定一个事务ID是否属于一个准备好的事务，则必须检查pg_prepared_xacts。
pg_current_snapshot () → pg_snapshot	返回当前snapshot，显示哪些事务IDs正在进行中的数据结构。
pg_snapshot_xip ( pg_snapshot ) → setof xid8	返回快照中包含的正在进行的事务IDs集。
pg_snapshot_xmax ( pg_snapshot ) → xid8	返回快照的xmax。
pg_snapshot_xmin ( pg_snapshot ) → xid8	返回快照的xmin。
pg_visible_in_snapshot ( xid8, pg_snapshot ) → boolean	根据此快照，给定的事务ID是否可见(visible) (也就是说，它是否在快照拍摄之前完成)? 注意，这个函数不会给出子事务ID的正确答案。

内部事务ID类型xid是32位宽的，可捆卷(wraps around)每40亿个事务。但是，Table 9.75中所示的函数使用的是64位类型的xid8，它在安装过程中不捆卷(wraps around)，如果需要，可以通过强制转换将其转换为xid。数据类型pg\_snapshot存储特定时刻事务ID可见性的信息。其组成如Table 9.76所描述。pg\_snapshot的文本表示形式是xmin:xmax:xip\_list。例如10:20:10,14,15表示xmin=10, xmax=20, xip\_list=10, 14, 15。

Table 9.76. 快照组件

名称	描述
xmin	仍然处于活动状态的最低事务ID。所有小于xmin的事务IDs要么提交且可见，要么回滚并死亡。
xmax	比最高完成的事务ID还高出一个值。所有大于或等于xmax的事务IDs到快照时还没有完成，因此不可见。
xip_list	快照时正在进行的事务。一个事务ID为xmin ≤ X < xmax且不在快照时已经完成的列表中，因此根据其提交状态，该事务ID要么是可见的，要么是死的。此列表不包括子事务的事务IDs。

在LightDB13以前的版本中，没有xid8类型，因此提供了这些函数的变体，使用bigint表示64位XID，并相应地提供不同的快照数据类型txid\_snapshot。这些旧的函数在它们的名字中有txid。它们仍然支持向后兼容性，但可能会从未来的版本中删除。参见Table 9.77

Table 9.77. 已弃用的事务ID和快照信息功能

函数	描述
txid_current () → bigint	参见 pg_current_xact_id()。
txid_current_if_assigned () → bigint	参见 pg_current_xact_id_if_assigned()。
txid_current_snapshot () → txid_snapshot	参见 pg_current_snapshot()。

函数	描述
<code>txid_snapshot_xip ( txid_snapshot )</code>	<code>→ setof bigint</code> 参见 <code>pg_snapshot_xip()</code> .
<code>txid_snapshot_xmax ( txid_snapshot )</code>	<code>→ bigint</code> 参见 <code>pg_snapshot_xmax()</code> .
<code>txid_snapshot_xmin ( txid_snapshot )</code>	<code>→ bigint</code> 参见 <code>pg_snapshot_xmin()</code> .
<code>txid_visible_in_snapshot ( bigint, txid_snapshot )</code>	<code>→ boolean</code> 参见 <code>pg_visible_in_snapshot()</code> .
<code>txid_status ( bigint )</code>	<code>→ text</code> 参见 <code>pg_xact_status()</code> .

Table 9.78中的函数提供了关于过去的事务何时被提交的信息。它们只在启用`track_commit_timestamp`配置选项时提供有用的数据，并且只针对在启用该选项后提交的事务。

Table 9.78. 已提交事务信息函数

函数	描述
<code>pg_xact_commit_timestamp ( xid )</code>	<code>→ timestamp with time zone</code> 返回事务的提交时间戳。
<code>pg_last_committed_xact ()</code>	<code>→ record ( xid xid, timestamp timestamp with time zone )</code> 返回最近提交的事务的事务ID和提交时间戳。

## 9.26.12. 控制文件数据函数

Table 9.79中所展示的函数能打印`initdb`期间初始化的信息，例如目录版本。它们也能显示有关预写式日志和检查点处理的信息。这些信息是集簇范围内的，不与任何特定的一个数据库相关。这些函数提供大致相同的信息，对于同一种来源，就像`lt_controldata`应用。

Table 9.79. 控制数据函数

函数	描述
<code>pg_control_checkpoint ()</code>	<code>→ record</code> 返回有关当前检查点状态的信息，如 Table 9.80所展示。
<code>pg_control_system ()</code>	<code>→ record</code> 返回有关当前控制文件状态的信息，如 Table 9.81所展示。
<code>pg_control_init ()</code>	<code>→ record</code> 返回有关集群初始化状态的信息，如 Table 9.82所展示。
<code>pg_control_recovery ()</code>	<code>→ record</code> 返回有关恢复状态的信息，如 Table 9.83所展示。

Table 9.80. `pg_control_checkpoint` 输出列

列名称	数据类型
<code>checkpoint_lsn</code>	<code>pg_lsn</code>
<code>redo_lsn</code>	<code>pg_lsn</code>
<code>redo_wal_file</code>	<code>text</code>

列名称	数据类型
timeline_id	integer
prev_timeline_id	integer
full_page_writes	boolean
next_xid	text
next_oid	oid
next_multixact_id	xid
next_multi_offset	xid
oldest_xid	xid
oldest_xid_dbid	oid
oldest_active_xid	xid
oldest_multi_xid	xid
oldest_multi_dbid	oid
oldest_commit_ts_xid	xid
newest_commit_ts_xid	xid
checkpoint_time	带时区的时间戳

Table 9.81. pg\_control\_system 输出列

列名称	数据类型
pg_control_version	integer
catalog_version_no	integer
system_identifier	bigint
pg_control_last_modified	timestamp with time zone

Table 9.82. pg\_control\_init 输出列

列名称	数据类型
max_data_alignment	integer
database_block_size	integer
blocks_per_segment	integer
wal_block_size	integer
bytes_per_wal_segment	integer
max_identifier_length	integer
max_index_columns	integer
max_toast_chunk_size	integer
large_object_chunk_size	integer
float8_pass_by_value	boolean
data_page_checksum_version	integer

Table 9.83. pg\_control\_recovery 输出列

列名称	数据类型
min_recovery_end_lsn	pg_lsn
min_recovery_end_timeline	integer



列名称	数据类型
backup_start_lsn	pg_lsn
backup_end_lsn	pg_lsn
end_of_backup_record_required	boolean

## 9.27. 系统管理函数

这一节描述的函数被用来控制和监视一个LightDB安装。

### 9.27.1. 配置设定函数

Table 9.8展示了那些可以用于查询以及修改运行时配置参数的函数。

Table 9.84. 配置设置函数

函数	描述 例子
<code>current_setting ( setting_name text [, missing_ok boolean ] )</code>	<code>→ text</code> 返回设置的setting_name的当前值。如果没有这样的设置，current_setting将抛出一个错误，除非missing_ok被提供并且为true。这个函数对应于SQL命令SHOW。 <code>current_setting('datestyle') → ISO, MDY</code>
<code>set_config ( setting_name text, new_value text, is_local boolean )</code>	<code>→ text</code> 将参数setting_name设置为new_value，并返回该值。如果is_local为true，新值将仅在当前事务中应用。如果您希望新值应用于当前会话的其余部分，请使用false代替。这个函数对应于SQL命令SET。 <code>set_config('log_statement_stats', 'off', false) → off</code>

### 9.27.2. 服务器信号函数

在Table 9.8中展示的函数向其它服务器进程发送控制信号。默认情况下这些函数只能被超级用户使用，但是如果需要，可以利用GRANT把访问特权授予给其他用户。

每个这样的函数如果成功则返回true，否则返回false。

Table 9.85. 服务器信号函数

函数	描述
<code>pg_cancel_backend ( pid integer )</code>	<code>→ boolean</code> 取消对后端进程具有指定进程ID的会话的当前查询。如果调用角色是后端被取消的角色的成员，或者调用角色被授予pg_signal_backend，也允许这样做，但是只有超级用户才能取消超级用户后端。
<code>pg_reload_conf ()</code>	<code>→ boolean</code> 导致LightDB服务器的所有进程重新加载其配置文件。（这是通过向postmaster进程发送SIGHUP信号来初始化的，然后这个进程又把SIGHUP发送给它的每个子进程。）
<code>pg_rotate_logfile ()</code>	<code>→ boolean</code> 通知日志文件管理器立即切换到一个新的输出文件。这仅在内置日志采集器运行时有效，因为否则没有日志文件管理器子进程。
<code>pg_terminate_backend ( pid integer )</code>	<code>→ boolean</code>

函数	描述
	终止后端进程具有指定进程ID的会话。 如果调用角色是后端被终止的角色的成员，或者调用角色被赋予pg_signal_backend，那么也允许这样做，但是只有超级用户才能终止超级用户后端。

pg\_cancel\_backend和pg\_terminate\_backend向由进程 ID 标识的后端进程发送信号（分别是SIGINT或SIGTERM）。 一个活动后端的进程 ID可以从pg\_stat\_activity视图的pid列中找到，或者通过在服务器上列出lightdb进程（在 Unix 上使用ps或者在Windows上使用任务管理器）得到。 一个活动后端的角色可以在pg\_stat\_activity视图的username列中找到。

### 9.27.3. 备份控制函数

Table 9.86中展示的函数可以辅助制作在线备份。 这些函数不能在恢复期间执行（非排他性的pg\_start\_backup，非排他性的pg\_stop\_backup，pg\_is\_in\_backup、pg\_backup\_start\_time和pg\_wal\_lsn\_diff除外）。

有关正确使用这些函数的详细信息，参见Section 23.3

Table 9.86. 备份控制函数

函数	描述
pg_create_restore_point ( name text ) → pg_lsn	在预写式日志中创建一个命名标记记录，稍后可以将其用作恢复目标，并返回相应的预写日志位置。 然后将给定的名称与 recovery_target_name一起使用，以指定进行恢复的点。 要避免创建多个名称相同的恢复点，因为恢复将在第一个名称与恢复目标匹配的恢复点停止。 默认情况下，该函数仅限超级用户使用，但可以授权给其他用户执行该函数。
pg_current_wal_flush_lsn () → pg_lsn	返回当前预写式日志刷新位置(参见下面的说明)。
pg_current_wal_insert_lsn () → pg_lsn	返回当前预式日志插入位置(参见下面的说明)。
pg_current_wal_lsn () → pg_lsn	返回当前预写式日志写位置(参见下面的说明)。
pg_start_backup ( label text [, fast boolean [, exclusive boolean ]] ) → pg_lsn	准备服务器开始在线备份。唯一需要的参数是用于备份的任意用户定义的标签。（通常，备份转储文件将存储在这个名称下。）如果可选的第二个参数被指定为true，它将指定尽可能快地执行pg_start_backup。这将强制产生一个即时检查点，这将导致I/O操作突增，从而降低并发执行的查询的速度。 第三个可选参数指定是执行排他或非排他备份(默认为排他备份)。 在排他模式下使用时，该函数将写一个备份标签文件(backup_label)，如果lt_tblspc/目录中有任何链接， 则将一个表空间映射文件(tablespace_map)写入数据库集群的数据目录，然后执行检查点，然后返回备份的开始写-提前日志位置。（用户可以忽略这个结果值，但在有用的情况下会提供它。）在非排他模式下使用时，这些文件的内容将由pg_stop_backup函数返回，并且应该由用户复制到备份区域。 默认情况下，该函数仅限超级用户使用，但可以授权其他用户EXECUTE该函数。
pg_stop_backup ( exclusive boolean [, wait_for_archive boolean ] ) → setof record ( lsn pg_lsn, labelfile text, spcmapfile text )	完成排他或非排他联机备份。exclusive参数必须与前面的pg_start_backup调用相匹配。 在排他备份中， pg_stop_backup删除备份标签文件，如果存在，则删除pg_start_backup创建的表空间映射文件。 在非排他备份中，这些文件的所需内容将作为函数结果的一部分返回，并且应该写入备份区域(不在数据目录)中的文件。

函数	描述
	<p>还有一个可选的boolean类型的第二个参数。如果为假，则该函数将在备份完成后立即返回，而无需等待WAL被归档。这种行为只有在独立监控WAL归档的备份软件中才有用。否则，使备份一致所需的WAL可能会丢失，从而使备份无效。默认情况下或当该参数为真时，<code>pg_stop_backup</code>将在启用归档时等待WAL被归档。（在备用状态下，这意味着只有当<code>archive_mode = always</code>时，它才会等待。如果主节点上的写活动很少，那么可以在主节点上运行<code>pg_switch_wal</code>来触发立即段切换。）</p> <p>当在主节点上执行时，这个函数还会在预写式日志归档区域中创建一个备份历史文件。历史文件包括给予<code>pg_start_backup</code>的标签，备份的开始和结束写前预写式日志的位置，以及备份的开始和结束时间。记录完结束位置后，当前的预写式日志插入点自动移到下一个预写式日志文件，以便结束的预写式日志文件可以立即归档，从而完成备份。</p> <p>该函数的结果是一条记录。<code>lsn</code>列保持备份的结束预写式日志位置(可以再忽略)。当结束排他备份时，第二和第三列为NULL；在非排他备份之后，它们保持标签和表空间映射文件所需的内容。</p> <p>默认情况下该函数仅限超级用户使用，但也可以授权其他用户执行该函数。</p>
<code>pg_stop_backup ()</code>	<p>→ <code>pg_lsn</code></p> <p>结束执行排他在线备份。这个简化版本等同于<code>pg_stop_backup(true, true)</code>，只是它只返回<code>pg_lsn</code>结果。</p> <p>默认情况下该函数仅限超级用户使用，但可以授权其他用户执行该函数。</p>
<code>pg_is_in_backup ()</code>	<p>→ <code>boolean</code></p> <p>如果正在进行联机排他备份则返回真。</p>
<code>pg_backup_start_time ()</code>	<p>→ <code>timestamp with time zone</code></p> <p>如果正在进行在线排他备份，则返回当前在线排他备份的开始时间，否则为NULL。</p>
<code>pg_switch_wal ()</code>	<p>→ <code>pg_lsn</code></p> <p>强制服务器切换到一个新的预写式日志文件，这允许对当前文件进行归档(假设你正在使用连续归档)。其结果是在刚刚完成的预写式日志文件中结束预写式日志位置加1。如果自从上次预写式日志切换以来没有提前写日志活动，<code>pg_switch_wal</code>将不做任何操作，并返回当前正在使用的提前写日志文件的起始位置。</p> <p>默认情况下该函数仅限超级用户使用，但可以授权其他用户执行该函数。</p>
<code>pg_walfile_name ( lsn pg_lsn )</code>	<p>→ <code>text</code></p> <p>将预写式日志位置转换为保持该位置的WAL文件的名称。</p>
<code>pg_walfile_name_offset ( lsn pg_lsn )</code>	<p>→ <code>record ( file_name text, file_offset integer )</code></p> <p>将预写式日志位置转换为WAL文件名和该文件中的字节偏移量。</p>
<code>pg_wal_lsn_diff ( lsn1 pg_lsn, lsn2 pg_lsn )</code>	<p>→ <code>numeric</code></p> <p>计算两个预写式日志位置之间的字节差。这可以与<code>pg_stat_replication</code>或Table 9.86中所示的一些函数一起使用，以获得复制延迟。</p>

`pg_current_wal_lsn` 显示当前预写式日志写位置，与上述函数所用的格式相同。类似地，`pg_current_wal_insert_lsn`显示当前预写式日志插入位置，`pg_current_wal_flush_lsn`显示当前预写式日志刷新位置。插入位置是预写式日志在任何时刻的“逻辑(logical)”结束，而写位置是已经从服务器内部缓冲区实际写入的内容的结束，而刷新位置是已知的要写入持久化存储的最后一个位置。写位置是可以从服务器外部检查的最后位置，如果你对归档部分完成的预写式日志文件感兴趣，那么它通常就是你想要的位置。插入和刷新位置主要用于服务器调试目的。这些都是只读操作，不需要超级用户权限。

你可以使用`pg_walfile_name_offset`从`pg_lsn`值中提取相应的预写式日志文件名称和字节偏移量。例如：

```
lightdb@postgres=# SELECT * FROM pg_walfile_name_offset(pg_stop_backup());
      file_name      | file_offset
-----|-----
00000001000000000000000D |      4039624
(1 row)
```

类似地，`pg_walfile_name`只提取预写式日志文件名称。当给定的预写式日志位置恰好位于预写式日志文件的边界时，这两个函数都会返回前一个预写式日志文件的名称。这通常是管理预写式日志归档行为所需的行为，因为前面的文件是当前需要归档的最后一个文件。

## 9.27.4. 恢复控制函数

Table 9.8中展示的函数提供有关后备服务器当前状态的信息。这些函数可以在恢复或普通运行过程中被执行。

Table 9.87. 恢复信息函数

函数	描述
<code>pg_is_in_recovery ()</code> → boolean	如果恢复仍在进行则返回真。
<code>pg_last_wal_receive_lsn ()</code> → pg_lsn	返回已接收并通过流复制同步到磁盘的最后一个预写式日志位置。当流复制正在进行时这将单调地增加。如果恢复已经完成，那么在恢复期间，接收到的最后一条WAL记录的位置将保持静态，并同步到磁盘。如果流复制已禁用，或者尚未启动，函数将返回NULL。
<code>pg_last_wal_replay_lsn ()</code> → pg_lsn	返回恢复期间重新播放的最后一个预写式日志位置。如果恢复仍在进行中这将会单调地增加。如果恢复已经完成，那么恢复期间应用的最后WAL记录的位置将保持静态。当服务器正常启动且没有恢复时，函数返回NULL。
<code>pg_last_xact_replay_timestamp ()</code> → timestamp with time zone	返回恢复期间重放的最后一个事务的时间戳。这是在主服务器上为该事务生成提交或中止WAL记录的时间。如果在恢复期间没有重放任何事务，该函数将返回NULL。否则，如果恢复仍在进行中，这将单调地增加。如果恢复已经完成，那么在恢复期间应用最后一个事务时，这将会保持静态。当服务器正常启动且没有恢复时，函数返回NULL。

控制恢复进度的功能如 Table 9.88所示。这些函数只能在恢复过程中执行。

Table 9.88. 恢复控制函数

函数	描述
<code>pg_is_wal_replay_paused ()</code> → boolean	如果恢复暂停了则返回真。
<code>pg_promote ( wait boolean DEFAULT true, wait_seconds integer DEFAULT 60 )</code> → boolean	将备用服务器提升为主服务器状态。当 <code>wait</code> 设置为 <code>true</code> (默认值)时，函数将等待直到升级完成或 <code>wait_seconds</code> 秒数已过，如果升级成功则返回 <code>true</code> ，否则返回 <code>false</code> 。如果 <code>wait</code> 设置为 <code>false</code> ，则该函数在向 <code>postmaster</code> 发送SIGUSR1信号以触发升级后立即返回 <code>true</code> 。 默认情况下这个函数仅限超级用户使用，但可以授权给其他用户执行该函数。
<code>pg_wal_replay_pause ()</code> → void	

函数	描述
	<p>暂停恢复。在恢复暂停时，不会应用进一步的数据库更改。如果热备是激活的，所有新查询将看到相同的一致数据库快照，并且在恢复继续之前不会生成进一步的查询冲突。</p> <p>默认情况下该函数仅限超级用户使用，但可以授权其他用户执行该函数。</p>
<code>pg_wal_replay_resume ()</code> → void	<p>如果暂停了，则重新启动恢复。</p> <p>默认情况下该函数仅限超级用户使用，但可以授权其他用户执行该函数。</p>

`pg_wal_replay_pause`和`pg_wal_replay_resume`不能在提升(promotion)进行时执行。如果在恢复暂停时触发了提升(promotion)，则暂停状态结束，升级继续进行。

如果禁用了流复制，则暂停状态可能会无限期地持续下去，不会出现问题。如果正在进行流复制，那么将继续接收WAL记录，这将最终填满可用磁盘空间，这取决于暂停持续时间、WAL生成速度和可用磁盘空间。

## 9.27.5. 快照同步函数

LightDB允许数据库会话同步它们的快照。一个快照决定对于正在使用该快照的事务哪些数据是可见的。当两个或者更多个会话需要看到数据库中的相同内容时，就需要同步快照。如果两个会话独立开始其事务，就总是有可能有某个第三事务在两个START TRANSACTION命令的执行之间提交，这样其中一个会话就可以看到该事务的效果而另一个则看不到。

为了解决这个问题，LightDB允许一个事务导出它正在使用的快照。只要导出的事务仍然保持打开，其他事务可以导入它的快照，并且因此可以保证它们可以看到和第一个事务看到的完全一样的数据库视图。但是注意这些事务中的任何一个对数据库所作的更改对其他事务仍然保持不可见，和未提交事务所作的修改一样。因此这些事务是针对以前存在的数据同步，而对由它们自己所作的更改则采取正常的动作。

如Table 9.89中所示，快照通过`pg_export_snapshot`函数导出，并且通过SET TRANSACTION命令导入。

Table 9.89. 快照同步函数

函数	描述
<code>pg_export_snapshot ()</code> → text	<p>保存事务的当前快照并返回text字符串以标识该快照。必须将此字符串传递(在数据库之外)给希望导入快照的客户端。快照仅在导出它的事务结束之前才可用于导入。</p> <p>如果需要的话，一个事务可以导出多个快照。请注意，这样做仅在READ COMMITTED事务中有效，因为在REPEATABLE READ和更高的隔离级别中，事务在它们的生命周期中使用相同的快照。一旦事务导出了快照，它就不能用PREPARE TRANSACTION进行准备。</p>

## 9.27.6. 复制管理函数

Table 9.90中展示的函数用于控制以及与复制特性交互。有关底层特性的信息请见Section 24.2.5 Section 24.2.6以及Chapter 46 复制原点函数的使用仅限于超级用户。复制槽的函数只限于超级用户和拥有REPLICATION权限的用户。

很多这些函数在复制协议中都有等价的命令，见Section 49.4

Section 9.27.3 Section 9.27.4和Section 9.27.5中描述的函数也与复制相关。

Table 9.90. 复制管理函数

函数	描述
<code>pg_create_physical_replication_slot</code>	<code>pg_create_physical_replication_slot ( slot_name name [, immediately_reserve boolean, temporary boolean ] ) → record ( slot_name name, lsn pg_lsn )</code> 创建一个新的名为 <code>slot_name</code> 的物理复制槽。第二个参数是可选的，当它为 <code>true</code> 时，立即为这个物理槽指定要被保留的LSN。否则该LSN会被保留在来自一个流复制客户端的第一个连接上。来自一个物理槽的流改变只可能出现在使用流复制协议时 — 见Section 49.4 当可选的第三参数 <code>temporary</code> 被设置为真时，指定那个槽不会被持久地存储在磁盘上并且仅对当前会话的使用有意义。临时槽也会在发生任何错误时被释放。这个函数对应于复制协议命令 <code>CREATE_REPLICATION_SLOT ... PHYSICAL</code> 。
<code>pg_drop_replication_slot</code>	<code>pg_drop_replication_slot ( slot_name name ) → void</code> 丢弃名为 <code>slot_name</code> 的物理或逻辑复制槽。和复制协议命令 <code>DROP_REPLICATION_SLOT</code> 相同。对于逻辑槽，在连接到在其中创建该槽的同一个数据库时，必须调用这个函数。
<code>pg_create_logical_replication_slot</code>	<code>pg_create_logical_replication_slot ( slot_name name, plugin name [, temporary boolean ] ) → record ( slot_name name, lsn pg_lsn )</code> 使用输出插件 <code>plugin</code> 创建一个名字为 <code>slot_name</code> 的新逻辑（解码）复制槽。当可选的第三参数 <code>temporary</code> 被设置为真时，指定那个槽不会被持久地存储在磁盘上并且仅对当前会话的使用有意义。临时槽也会在发生任何错误时被释放。对这个函数的调用与复制协议命令 <code>CREATE_REPLICATION_SLOT ... LOGICAL</code> 。
<code>pg_copy_physical_replication_slot</code>	<code>pg_copy_physical_replication_slot ( src_slot_name name, dst_slot_name name [, temporary boolean ] ) → record ( slot_name name, lsn pg_lsn )</code> 将一个名为 <code>src_slot_name</code> 的现有物理复制槽复制到一个名为 <code>dst_slot_name</code> 的物理复制槽。被复制的物理槽开始从与源槽相同的LSN开始保留WAL。 <code>temporary</code> 是可选的。如果省略了 <code>temporary</code> ，则使用与源槽相同的值。
<code>pg_copy_logical_replication_slot</code>	<code>pg_copy_logical_replication_slot ( src_slot_name name, dst_slot_name name [, temporary boolean [, plugin name ] ] ) → record ( slot_name name, lsn pg_lsn )</code> 复制一个名为 <code>src_slot_name</code> 的现有逻辑复制槽到一个名为 <code>dst_slot_name</code> 的逻辑复制槽，选择性的改变输出插件和持久性。被复制的逻辑槽从与源逻辑槽相同的LSN开始。 <code>temporary</code> 和 <code>plugin</code> 都是可选的。如果它们被省略了，使用与源逻辑槽相同的值。
<code>pg_logical_slot_get_changes</code>	<code>pg_logical_slot_get_changes ( slot_name name, upto_lsn pg_lsn, upto_nchanges integer, VARIADIC options text[] ) → setof record ( lsn pg_lsn, xid xid, data text )</code> 返回槽 <code>slot_name</code> 中的改变，从最后一次使用更改的点开始。如果 <code>upto_lsn</code> 和 <code>upto_nchanges</code> 为 <code>NULL</code> ，逻辑解码将一直继续到WAL的末尾。如果 <code>upto_lsn</code> 为非 <code>NULL</code> ，解码将只包括那些在指定LSN之前提交的事务。如果 <code>upto_nchanges</code> 为非 <code>NULL</code> ，解码将在其产生的行数超过指定值后停止。不过要注意，被返回的实际行数可能更大，因为对这个限制的检查只会在增加了解码每个新的提交事务产生的行之后进行。
<code>pg_logical_slot_peek_changes</code>	<code>pg_logical_slot_peek_changes ( slot_name name, upto_lsn pg_lsn, upto_nchanges integer, VARIADIC options text[] ) → setof record ( lsn pg_lsn, xid xid, data text )</code> 行为就像 <code>pg_logical_slot_get_changes()</code> 函数，不过改变不会被消费，即在未来的调用中还会返回这些改变。
<code>pg_logical_slot_get_binary_changes</code>	<code>pg_logical_slot_get_binary_changes ( slot_name name, upto_lsn pg_lsn, upto_nchanges integer, VARIADIC options text[] ) → setof record ( lsn pg_lsn, xid xid, data bytea )</code> 行为就像 <code>pg_logical_slot_get_changes()</code> 函数，不过改变会以 <code>bytea</code> 返回。

函数	描述
<code>pg_logical_slot_peek_binary_changes ( slot_name name, upto_lsn pg_lsn, upto_nchanges integer, VARIADIC options text[] )</code>	<code>→ setof record ( lsn pg_lsn, xid xid, data bytea )</code> 行为就像 <code>pg_logical_slot_peek_changes()</code> 函数，不过改变会以 <code>bytea</code> 返回。
<code>pg_replication_slot_advance ( slot_name name, upto_lsn pg_lsn )</code>	<code>→ record ( slot_name name, end_lsn pg_lsn )</code> 将复制槽的当前确认的位置提前到名为 <code>slot_name</code> 的复制槽的当前确认位置。该槽不会向后移动，也不会移动到当前插入位置之外。返回该槽的名称和它被推进到的真实位置。如果有任何进展，则在下一个检查点中写出更新后的槽位置信息。所以如果发生崩溃，该槽位可能会返回到之前的位置。
<code>pg_replication_origin_create ( node_name text )</code>	<code>→ oid</code> 用给定的外部名称创建一个复制源，并且返回分配给它的内部 ID。
<code>pg_replication_origin_drop ( node_name text )</code>	<code>→ void</code> 删除一个以前创建的复制源，包括任何相关的重放进度。
<code>pg_replication_origin_oid ( node_name text )</code>	<code>→ oid</code> 根据名称查找复制源并返回内部ID。如果没有发现这样的复制源，则抛出错误。
<code>pg_replication_origin_session_setup ( node_name text )</code>	<code>→ void</code> 将当前会话标记为从给定的原点回放，从而允许跟踪回放进度。只能在当前没有选择原点时使用。使用 <code>pg_replication_origin_session_reset</code> 命令来撤销。
<code>pg_replication_origin_session_reset ()</code>	<code>→ void</code> 取消 <code>pg_replication_origin_session_setup()</code> 的效果。
<code>pg_replication_origin_session_is_setup ()</code>	<code>→ boolean</code> 如果在当前会话中选择了复制源则返回真。
<code>pg_replication_origin_session_progress ( flush boolean )</code>	<code>→ pg_lsn</code> 返回当前会话中选择的复制源的重放位置。参数 <code>flush</code> 决定对应的本地事务是否被确保已经刷入磁盘。
<code>pg_replication_origin_xact_setup ( origin_lsn pg_lsn, origin_timestamp timestamp with time zone )</code>	<code>→ void</code> 将当前事务标记为重放在给定LSN和时间戳上提交的事务。只能在使用 <code>pg_replication_origin_session_setup</code> 选择复制源时调用。
<code>pg_replication_origin_xact_reset ()</code>	<code>→ void</code> 取消 <code>pg_replication_origin_xact_setup()</code> 的效果。
<code>pg_replication_origin_advance ( node_name text, lsn pg_lsn )</code>	<code>→ void</code> 将给定节点的复制进度设置为给定的位置。这主要用于设置初始位置，或在配置更改或类似的变更后设置新位置。请注意这个函数的不当使用可能会导致不一致的复制数据。
<code>pg_replication_origin_progress ( node_name text, flush boolean )</code>	<code>→ pg_lsn</code> 返回给定复制元重放位置。参数 <code>flush</code> 决定对应的本地事务是否被确保已经刷入磁盘。
<code>pg_logical_emit_message ( transactional boolean, prefix text, content text )</code> <code>pg_logical_emit_message ( transactional boolean, prefix text, content bytea )</code>	<code>→ pg_lsn</code> <code>→ pg_lsn</code> 发出逻辑解码消息。这可以被用来通过 WAL 向逻辑解码插件传递一般消息。 <code>transactional</code> 参数指定该消息是否应该是当前事务的一部分或者当逻辑解码读到该记录时该消息是否应该被立刻写入并且解码。 <code>prefix</code> 参数是逻辑解码插件可以用来识别它们感兴趣的消息的文本前缀。 <code>content</code> 是消息的文本，以文本或二进制形式给出。

## 9.27.7. 数据库对象管理函数

Table 9.9中所示的函数计算数据库对象的磁盘空间使用情况，或帮助表示使用结果。所有这些函数都返回以字节为单位的大小。如果将不代表已有对象的OID传递给这些函数之一，则返回NULL。

Table 9.91. 数据库对象尺寸函数

函数	描述
<code>pg_column_size ( "any" ) → integer</code>	显示用于存储任何单个数据值的字节数。如果直接应用于表的列值，则反映所做的任何压缩。
<code>pg_database_size ( name ) → bigint</code> <code>pg_database_size ( oid ) → bigint</code>	计算指定名称或OID的数据库所使用的总磁盘空间。要使用这个函数，你必须拥有CONNECT特权(默认授予)在指定的数据库上，或者是 <code>pg_read_all_stats</code> 角色的成员。
<code>pg_indexes_size ( regclass ) → bigint</code>	计算附加到指定表的索引所使用的总磁盘空间。
<code>pg_relation_size ( relation regclass [, fork text ] ) → bigint</code>	计算指定关系的一个“fork”所使用的磁盘空间。(注意在大多数情况下，使用更高级的函数 <code>pg_total_relation_size</code> 或 <code>pg_table_size</code> 更方便，它们将所有分叉(forks)的大小相加。)使用一个参数，这将返回关系的主数据分叉的大小。第二个参数可以用来指定要检查哪个分叉： <ul style="list-style-type: none"> <li>• <code>main</code> 返回关系的主数据分叉的大小。</li> <li>• <code>fsm</code> 返回与该关系关联的空闲空间映射(参见Section 64.3)的大小。</li> <li>• <code>vm</code> 返回与该关系相关联的可见性映射(参见Section 64.4)的大小。</li> <li>• <code>init</code> 返回初始化分叉的大小，如果有的话，与关系相关。</li> </ul>
<code>pg_size_bytes ( text ) → bigint</code>	将人可读的格式的大小(由 <code>pg_size_pretty</code> 返回)转换为字节。
<code>pg_size_pretty ( bigint ) → text</code> <code>pg_size_pretty ( numeric ) → text</code>	将以字节为单位的大小转换为更容易被人阅读的格式，使用大小单位(酌情采用字节、kB、MB、GB或TB等等)。注意，单位是2的幂，而不是10的幂，所以1kB是1024字节，1MB是 $1024^2 = 1048576$ 字节，依此类推。
<code>pg_table_size ( regclass ) → bigint</code>	计算指定表所使用的磁盘空间，不包括索引(但包括它的TOAST表，如果有的话，空闲空间映射，以及可见性映射)。
<code>pg_tablespace_size ( name ) → bigint</code> <code>pg_tablespace_size ( oid ) → bigint</code>	使用指定的名称或OID计算表空间中使用的总磁盘空间。要使用这个函数，必须在指定的表空间上拥有CREATE特权，或者是 <code>pg_read_all_stats</code> 角色的成员，除非它是当前数据库的默认表空间。
<code>pg_total_relation_size ( regclass ) → bigint</code>	计算指定表所使用的总磁盘空间，包括所有索引和TOAST数据。结果等价 <code>pg_table_size + pg_indexes_size</code> 。

上述操作表和索引的函数接受一个`regclass`参数，它是该表或索引在`pg_class`系统目录中的OID。你不必手工去查找该OID，因为`regclass`数据类型的输入转换器会为你代劳。只写包围在单引号内的表名，这样它看起来像一个文字常量。为了与普通SQL名称的处理相兼容，该字符串将被转换为小写形式，除非其中在表名周围包含双引号。

Table 9.9中展示的函数帮助标识数据库对象相关的磁盘文件。



Table 9.92. 数据库对象位置函数

函数	描述
<code>pg_relation_filenode ( relation regclass ) → oid</code>	返回当前分配给指定关系的“filenode”数字。文件节点是用于该关系的文件名称的基本组件(更多信息请参阅Section 64.1)。对于大多数关系,其结果与 <code>pg_class.relfilenode</code> 相同,但对于某些系统目录,relfilenode为0,并且必须使用这个函数来获得正确的值。如果传递的是一个没有存储的关系,例如一个视图,那么函数将返回NULL。
<code>pg_relation_filepath ( relation regclass ) → text</code>	返回关系的完整文件路径名称(相对于数据库集群的数据目录,关系的LTDATA)。
<code>pg_filenode_relation ( tablespace oid, filenode oid ) → regclass</code>	返回一个给定表空间OID和存储它的文件节点的关系的OID。这本质上是 <code>pg_relation_filepath</code> 的反向映射。对于数据库默认表空间中的关系,表空间可以指定为0。如果当前数据库中没有与给定值相关联的关系,则返回 NULL。

Table 9.93列出用于管理排序规则的函数。

Table 9.93. 排序规则管理函数

函数	描述
<code>pg_collation_actual_version ( oid ) → text</code>	返回当前安装在操作系统中的该排序规则对象的实际版本。如果这个版本与 <code>pg_collation.collversion</code> 中的值不同,则依赖于该排序规则的对象可能需要被重建。还可以参考ALTER COLLATION。
<code>pg_import_system_collations ( schema regnamespace ) → integer</code>	基于在操作系统中找到的所有区域环境(locales),加入排序规则到系统目录 <code>pg_collation</code> 中。这是 <code>initdb</code> 会用到的,更多细节请参考Section 21.2.2 如果后来在操作系统上安装了额外的区域环境,可以再次运行这个函数加入新区域环境的排序规则。匹配 <code>pg_collation</code> 中现有条目的区域环境将被跳过(但是这个函数不会移除以在操作系统中不再存在的区域环境为基础的排序规则对象)。 <code>schema</code> 参数通常是 <code>pg_catalog</code> ,但这不是一种要求,排序规则也可以被安装到其他的方案中。该函数返回其创建的新排序规则对象的数量。

Table 9.94列出提供有关分区表结构信息的函数。

Table 9.94. 分区信息函数

函数	描述
<code>pg_partition_tree ( regclass ) → setof record ( relid regclass, parentrelid regclass, isleaf boolean, level integer )</code>	列出给定分区表或分区索引的分区树中的表或索引,每行对应一个分区。提供的信息包括分区的OID、其直接父的OID、一个布尔值以告知分区是否是叶子,以及一个整数用来告诉分区在层次结构中的级别。对于输入表或索引,级别值为0,其直接子分区的为1,它们的分区为2,以此类推。如果关系不存在,或者不是分区或分区表,则不返回行。
<code>pg_partition_ancestors ( regclass ) → setof regclass</code>	列出给定分区的祖先关系,包括关系本身。如果关系不存在,或者不是分区或分区表,则不返回行。
<code>pg_partition_root ( regclass ) → regclass</code>	返回给定关系所属的分区树的最顶级父节点。如果关系不存在,或者不是分区或分区表,则返回NULL。

例如，要检查分区表measurement中包含的数据的总大小，可以使用以下查询：

```
SELECT pg_size_pretty(sum(pg_relation_size(releid))) AS total_size
FROM pg_partition_tree('measurement');
```

## 9.27.8. 索引维护函数

Table 9.95 显示了索引维护任务可以使用的函数。（注意，这些维护任务通常由自动清理（autovacuum）自动完成；只有在特殊情况下才需要使用这些函数。）这些函数在恢复过程中无法执行。这些函数的使用局限于超级用户和给定索引的所有者。

Table 9.95. 索引维护函数

函数	描述
<code>brin_summarize_new_values ( index regclass ) → integer</code>	扫描指定的BRIN索引以查找基表中当前没有被索引归纳的页面范围；对于任何这样的范围，它都通过扫描这些表页来创建一个新的摘要索引元组。返回插入到索引中的新页面范围摘要的数量。
<code>brin_summarize_range ( index regclass, blockNumber bigint ) → integer</code>	归纳(Summarizes)覆盖给定块的页面范围(如果还没有归纳的话)。这类类似于 <code>brin_summarize_new_values</code> ，只是它只处理覆盖给定表块数的页范围。
<code>brin_desummarize_range ( index regclass, blockNumber bigint ) → void</code>	删除归纳了覆盖给定表块的页面范围的BRIN索引元组，如果有的话。
<code>gin_clean_pending_list ( index regclass ) → bigint</code>	清理指定GIN索引的“pending”列表，通过移除里面的条目，以大批的方式，到主要的GIN数据结构。返回从挂起列表中删除的页数。如果参数是使用禁用 <code>fastupdate</code> 选项构建的GIN索引，则不会发生清理，结果为零，因为索引没有挂起的列表。关于挂起列表和 <code>fastupdate</code> 选项的详细信息，请参见Section 62.4.1和Section 62.5

## 9.27.9. 通用文件访问函数

Table 9.96中展示的函数提供了对数据库服务器所在机器上的文件的本地访问。只能访问数据库集簇目录以及`log_directory`中的文件，除非用户是超级用户或者被授予了角色`pg_read_server_files`。使用相对路径访问集簇目录里面的文件，以及匹配`log_directory`配置设置的路径访问日志文件。

注意在`pg_read_file()`或者相关函数上，向用户授予`EXECUTE`特权，以允许他们有能力读取服务器上该数据库服务器进程能读取的任何文件；这些函数会绕过所有的数据库内特权检查。这意味着，例如，具有这种访问的用户能够读取`pg_authid`表中存储着认证信息的内容，也能读取数据库中的任何表数据。因此，授予对这些函数的访问应该要很仔细地考虑。

这些函数中的一些函数有可选的`missing_ok`参数，可以指定当文件或目录不存在时的行为。如果为`true`，函数返回`NULL`或者空结果集，根据适合情况。如果为`false`，会产生一个错误。默认为`false`。

Table 9.96. 通用文件访问函数

函数	描述
<code>pg_ls_dir ( dirname text [, missing_ok boolean, include_dot_dirs boolean ] ) → setof text</code>	返回指定目录中所有文件(和目录以及其他指定文件)的名称。 <code>include_dot_dirs</code> 参数标示在结果集中是否包括“.”和“..”；默认为不包括

函数	描述
	<p>它们。要包括它们在missing_ok为 true时能够有用，以从不存在的目录中辨别一个空目录。</p> <p>这个函数默认限制为超级用户，但是其他用户可以被授予EXECUTE以运行此函数。</p>
pg_ls_logdir ()	<p>→ setof record ( name text, size bigint, modification timestamp with time zone )</p> <p>返回服务器的日志目录中的每个普通文件的名称、大小、和最后修改时间 (mtime)。文件名以一个点(dot)开始，目录和其他指定的文件不包括。</p> <p>这个函数默认限制在超级用户和pg_monitor角色的成员，但是其他用户可以被授予EXECUTE以运行此函数。</p>
pg_ls_waldir ()	<p>→ setof record ( name text, size bigint, modification timestamp with time zone )</p> <p>返回服务器的预写式日志 (WAL) 目录中的每个普通文件的名称、大小、和最后修改时间 (mtime)。文件名以一个点(dot)开始，目录和其他指定的文件不包括。</p> <p>这个函数默认限制在超级用户和pg_monitor角色的成员，但是其他用户可以被授予EXECUTE以运行此函数。</p>
pg_ls_archive_statusdir ()	<p>→ setof record ( name text, size bigint, modification timestamp with time zone )</p> <p>返回服务器的WAL归档状态目录(lt_wal/archive_status)中的每个普通文件的名称、大小、和最后修改时间 (mtime)。文件名以一个点(dot)开始，目录和其他指定的文件不包括。</p> <p>这个函数默认限制在超级用户和pg_monitor角色的成员，但是其他用户可以被授予EXECUTE以运行此函数。</p>
pg_ls_tmpdir ( [ tablespace oid ] )	<p>→ setof record ( name text, size bigint, modification timestamp with time zone )</p> <p>返回针对指定tablespace的临时文件目录中的每个普通文件的名称、大小、和最后修改时间 (mtime)。如果tablespace没有提供，pg_default 表空间为被检验的。文件名以一个点(dot)开始，目录和其他指定的文件不包括。</p> <p>这个函数默认限制在超级用户和pg_monitor角色的成员，但是其他用户可以被授予EXECUTE以运行此函数。</p>
pg_read_file ( filename text [, offset bigint, length bigint [, missing_ok boolean ] ] )	<p>→ text</p> <p>返回一个文本文件的全部或部分，开始于给定的字节offset，在最大的length字节返回（如果文件的结尾先达到了则减少）。如果offset为负，他于文件的结尾有关。如果offset和length 被省略，整个文件被返回。从文件中读的字节在服务器的编码中解释为字符串；如果它们在哪些编码中不是有效的则抛出错误。</p> <p>这个函数默认限制在超级用户，但是其他用户可以被授予EXECUTE以运行此函数。</p>
pg_read_binary_file ( filename text [, offset bigint, length bigint [, missing_ok boolean ] ] )	<p>→ bytea</p> <p>返回文件的全部或部分。这个函数与pg_read_file 是完全相同的，除了它可以读任意的二进制数据，返回结果为bytea而不是text；因此，没有编码检查会被执行。</p> <p>这个函数默认限制在超级用户，但是其他用户可以被授予EXECUTE以运行此函数。与 convert_from函数组合，这个函数能被用于以指定的编码读文本文件并转换到数据库的编码：</p> <pre>SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');</pre>
pg_stat_file ( filename text [, missing_ok boolean ] )	<p>→ record ( size bigint, access timestamp with time zone, modification timestamp with time zone, change timestamp with time zone, creation timestamp with time zone, isdir boolean )</p>

函数	描述
	返回一个记录，包含文件的大小、最后访问时间戳、最后修改时间戳，最后文件状态变更时间戳(仅在UNIX平台)、文件建立时间戳(仅Windows)，和一个标志旗如果它是一个目录。 这个函数默认限制在超级用户，但是其他用户可以被授予EXECUTE以运行此函数。

## 9.27.10. 咨询锁函数

Table 9.97中展示的函数管理咨询锁。有关正确使用这些函数的细节请参考Section 13.3.5

所有这些函数都打算用于锁定应用程序定义的资源，可以通过一个64位键值或两个32位键值来标识(注意这两个键空间不能重叠)。如果另一个会话已经在相同的资源标识符上持有冲突的锁，函数将等待直到资源变成可用，或者返回一个false结果，合适于函数的。锁可以是共享或排他的:共享锁不会与同一资源上的其他共享锁发生冲突，只会与排他锁发生冲突。锁可以在会话级(这样它们被保持直到释放或会话结束)或在事务级(这样它们被保持直到当前事务结束;没有手动释放的供应)。多个会话级锁请求堆栈，因此如果同一个资源标识符被锁定三次，那么必须有三个解锁请求来释放资源，在会话结束之前。

Table 9.97. 咨询锁函数

函数	描述
<code>pg_advisory_lock ( key bigint ) → void</code> <code>pg_advisory_lock ( key1 integer, key2 integer ) → void</code>	获取一个排他的会话级咨询锁，如有必要则等待。
<code>pg_advisory_lock_shared ( key bigint ) → void</code> <code>pg_advisory_lock_shared ( key1 integer, key2 integer ) → void</code>	获取一个共享的会话级咨询锁，如有必要则等待。
<code>pg_advisory_unlock ( key bigint ) → boolean</code> <code>pg_advisory_unlock ( key1 integer, key2 integer ) → boolean</code>	释放以前获取的排他会话级咨询锁。如果锁成功释放则返回true。如果锁没有被持有，则返回false，此外，服务器将报告一个SQL警告。
<code>pg_advisory_unlock_all () → void</code>	释放当前会话所持有的所有会话级咨询锁。(即使客户端异常断开连接，这个函数也会在会话结束时被隐式调用。)
<code>pg_advisory_unlock_shared ( key bigint ) → boolean</code> <code>pg_advisory_unlock_shared ( key1 integer, key2 integer ) → boolean</code>	释放以前获取的共享会话级咨询锁。如果锁成功释放则返回true。如果锁没有被持有，则返回false，此外，服务器将报告一个SQL警告。
<code>pg_advisory_xact_lock ( key bigint ) → void</code> <code>pg_advisory_xact_lock ( key1 integer, key2 integer ) → void</code>	获取一个排他的事务级咨询锁，如有必要则等待。
<code>pg_advisory_xact_lock_shared ( key bigint ) → void</code> <code>pg_advisory_xact_lock_shared ( key1 integer, key2 integer ) → void</code>	获取一个共享的事务级咨询锁，如有必要则等待。
<code>pg_try_advisory_lock ( key bigint ) → boolean</code> <code>pg_try_advisory_lock ( key1 integer, key2 integer ) → boolean</code>	获取一个排他的会话级咨询锁，如果适用。这将立即获得锁并返回true，或者如果不能立即获取锁则立即返回 false而无需等待。
<code>pg_try_advisory_lock_shared ( key bigint ) → boolean</code> <code>pg_try_advisory_lock_shared ( key1 integer, key2 integer ) → boolean</code>	

函数	描述
	获取一个共享的会话级咨询锁，如果适用。这将立即获得锁并返回true，或者如果不能立即获取锁则立即返回 false而无需等待。
pg_try_advisory_xact_lock ( key bigint ) → boolean pg_try_advisory_xact_lock ( key1 integer, key2 integer ) → boolean	获取一个排他的事务级咨询锁，如果适用。这将立即获得锁并返回true，或者如果不能立即获取锁则立即返回 false而无需等待。
pg_try_advisory_xact_lock_shared ( key bigint ) → boolean pg_try_advisory_xact_lock_shared ( key1 integer, key2 integer ) → boolean	获取一个共享的事务级咨询锁，如果适用。这将立即获得锁并返回true，或者如果不能立即获取锁则立即返回 false而无需等待。

## 9.28. 触发器函数

虽然很多触发器的使用都涉及到用户编写的触发器函数，但LightDB提供了一些可以直接在用户定义触发器中使用的内置触发器函数。这些总结在了Table 9.98中。（另外还有一些内置的触发器函数存在，它们实现了外键约束和延期索引约束。因为用户不需要直接使用它们，所以这里就不做论述了。）

有关创建触发器的更多信息，请参考CREATE TRIGGER。

Table 9.98. 内置触发器函数

函数	描述 Example Usage
suppress_redundant_updates_trigger ( ) → trigger	阻止不做事的更新操作。详见下文。 CREATE TRIGGER ... suppress_redundant_updates_trigger()
tsvector_update_trigger ( ) → trigger	自动从相关的纯文本文档列更新tsvector列。要使用的文本搜索配置是以名称指定为触发器参数。详情请参见Section 12.4.3 CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)
tsvector_update_trigger_column ( ) → trigger	自动从相关的纯文本文档列更新tsvector列。要使用的文本搜索配置取自表的regconfig列。详情请参见Section 12.4.3 CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, tsconfigcol, title, body)

suppress\_redundant\_updates\_trigger函数，在作为行级BEFORE UPDATE触发器应用时，将阻止任何没有实际更改行中数据的更新发生。这会覆盖那种始终执行物理行更新而无论数据是否已更改的常规行为。（这种常规的行为使更新运行得更快，因为不需要检查，而且在某些情况下也很有用。）

理想的情况下，你应该避免运行实际上并没有改变记录中数据的更新。冗余更新会花费大量不必要的时间，尤其是如果有大量索引要改变，并将最终不得不清理被死亡行占用的空间。但是，在客户端代码中检测这种情况并不总是容易的，甚至不可能做到。而写表达式来检测它们容易产生错误。作为替代，使用suppress\_redundant\_updates\_trigger可以跳过不改变数据的更新。但是，你需要小心使用它。触发器需要很短但不能忽略的时间来处理每条记录，所以如果受更新影响的大多数记录确实变化了，此触发器的使用将使更新比平均水平运行得更慢。

suppress\_redundant\_updates\_trigger函数可以像这样添加到表中：

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE FUNCTION suppress_redundant_updates_trigger();
```

在大多数情况下，你需要为每一行最后触发这个触发器，这样它就不会覆盖可能希望更改该行的其他触发器。请记住，触发器是按照名称顺序触发的，你将为此选择一个触发器名称，该名称位于表中可能存在的任何其他触发器的名称之后。（因此在示例中使用了“z”前缀。）

## 9.29. 事件触发器函数

LightDB提供了这些助手函数来从 事件触发器检索信息。

更多有关事件触发器的信息请见Chapter 38

### 9.29.1. 在命令结束处捕捉更改

```
pg_event_trigger_ddl_commands () → setof record
```

当在一个ddl\_command\_end事件触发器的函数中调用时，pg\_event\_trigger\_ddl\_commands返回被每一个用户动作 执行的DDL命令的列表。如果在其他任何环境中 调用这个函数，会发生错误。pg\_event\_trigger\_ddl\_commands为每一个被执行的基本 命令返回一行，某些只有一个单一 SQL 句子的命令可能会返回多于一行。这个函数返回下面的列：

名称	类型	描述
classid	oid	对象所属的目录的 OID
objid	oid	对象本身的 OID
objsubid	integer	对象的子-id（例如列的属性号）
command_tag	text	命令标签
object_type	text	对象的类型
schema_name	text	该对象所属的模式名称（如果有），如果没有则为NULL。没有引号。
object_identity	text	对象标识的文本表现形式，用模式限定。如果必要，出现在 该标识中的每一个标识符都会被引用。
in_extension	boolean	如果该命令是一个扩展脚本的一部分则为真
command	pg_ddl_command	以内部格式表达的该命令的一个完整表现形式。这不能被直接输出，但是可以把它传递给其他函数来得到有关于该命令不同部分的信息。

### 9.29.2. 处理被 DDL 命令删除的对象

```
pg_event_trigger_dropped_objects () → setof record
```

`pg_event_trigger_dropped_objects`返回被调用`sql_drop`事件的命令删除的所有对象的列表。如果调用在任何其他上下文中，会引发一个错误。这个函数返回以下列：

名称	类型	描述
<code>classid</code>	<code>oid</code>	对象所属的目录的 OID
<code>objid</code>	<code>oid</code>	对象本身的 OID
<code>objsubid</code>	<code>integer</code>	子对象ID（如列的属性号）
<code>original</code>	<code>boolean</code>	如果这是删除中的一个根对象则为真
<code>normal</code>	<code>boolean</code>	指示在依赖图中有一个普通依赖关系指向该对象的标志
<code>is_temporary</code>	<code>boolean</code>	如果该对象是一个临时对象则为真
<code>object_type</code>	<code>text</code>	对象的类型
<code>schema_name</code>	<code>text</code>	对象所属模式的名称（如果存在）；否则为NULL。不应用引用。
<code>object_name</code>	<code>text</code>	如果模式和名称的组合能被用于对象的一个唯一标识符，则是对象的名称；否则是NULL。不应用引用，并且名称不是模式限定的。
<code>object_identity</code>	<code>text</code>	对象身份的文本表现，模式限定的。每一个以及所有身份中出现的标识符在必要时加引号。
<code>address_names</code>	<code>text[]</code>	一个数组，它可以和 <code>object_type</code> 及 <code>address_args</code> 一起通过 <code>pg_get_object_address</code> 函数在一台包含有同类相同名称对象的远程服务器上重建该对象地址。
<code>address_args</code>	<code>text[]</code>	上述 <code>address_names</code> 的补充。

`pg_event_trigger_dropped_objects`可以被这样用在一个事件触发器中：

```
CREATE FUNCTION test_event_trigger_for_drops()
    RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % % % %',
            tg_tag,
            obj.object_type,
            obj.schema_name,
            obj.object_name,
            obj.object_identity;
    END LOOP;
END;
$$;
```

```
CREATE EVENT TRIGGER test_event_trigger_for_drops
  ON sql_drop
  EXECUTE FUNCTION test_event_trigger_for_drops();
```

### 9.29.3. 处理表重写事件

Table 9.99中所示的函数提供刚刚被调用过`table_rewrite`事件的表的信息。如果在任何其他环境中调用，会发生错误。

Table 9.99. 表重写信息函数

函数	描述
<code>pg_event_trigger_table_rewrite_oid () → oid</code>	返回将要重写的表的OID。
<code>pg_event_trigger_table_rewrite_reason () → integer</code>	返回解释重写原因的代码。代码的确切含义取决于版本。

这些函数可以在事件触发器中使用，就像这样：

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
  RETURNS event_trigger
  LANGUAGE plpgsql AS
  $$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
    pg_event_trigger_table_rewrite_oid()::regclass,
    pg_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
  ON table_rewrite
  EXECUTE FUNCTION test_event_trigger_table_rewrite_oid();
```

## 9.30. 统计信息函数

LightDB提供了一个函数来检查使用`CREATE STATISTICS`命令定义的复杂统计。

### 9.30.1. 检查MCV列表

```
pg_mcv_list_items ( pg_mcv_list ) → setof record
```

`pg_mcv_list_items`返回一组记录，描述存储在多列MCV列表中的所有项目。它返回以下列：

名称	类型	描述
<code>index</code>	<code>integer</code>	MCV 列表中的项目索引
<code>values</code>	<code>text[]</code>	存储在MCV项目中的值
<code>nulls</code>	<code>boolean[]</code>	标识NULL值的标志
<code>frequency</code>	<code>double precision</code>	MCV项目的频率
<code>base_frequency</code>	<code>double precision</code>	MCV项目的基本频率

`pg_mcv_list_items`函数可以这样使用：



```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
       pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts';
```

pg\_mcv\_list类型的值只能从pg\_statistic\_ext\_data.stxdmcv列中获取。

---

## Chapter 10. 类型转换

SQL语句可能（有意无意地）要求在同一表达式里混合不同的数据类型。LightDB在计算混合类型表达式方面有许多功能。

在大多数情况下，用户不需要明白类型转换机制的细节。但是，由LightDB进行的隐式类型转换会对查询的结果产生影响。必要时这些结果可以被使用显式类型转换来调整。

本章介绍LightDB类型转换的机制和习惯。关于特定的类型和允许的函数及操作符的进一步信息，请参考Chapter 8和Chapter 9的相关章节。

### 10.1. 概述

SQL是一种强类型语言。也就是说，每个数据项都有一个相关的数据类型，数据类型决定其行为和允许的用法。LightDB有一个可扩展的类型系统，该系统比其它SQL实现更具通用和灵活。因而，LightDB中大多数类型转换行为是由通用规则来管理的，而不是特定的启发式规则。这种做法允许使用混合类型表达式，即便是其中包含用户定义的类型。

LightDB扫描器/解析器只将词法元素分解成五个基本种类：整数、非整数数字、字符串、标识符、关键字。大多数非数字类型常量首先被分类为字符串。SQL语言定义允许将类型名指定为字符串，这个机制被LightDB用于保证解析器沿着正确的方向运行。例如，查询：

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";
```

```
label | value
-----+-----
Origin | (0,0)
(1 row)
```

有两个文字常量，类型分别为text和point。如果一个串文字没有指定类型，初始将被分配一个占位符类型unknown，该类型将在下文描述的后续阶段被解析。

在SQL解析器里，有四种基本的SQL结构要求独立的类型转换规则：

#### 函数调用

LightDB类型系统的大部分建立在一套丰富的函数上。函数可以有一个或多个参数。由于LightDB允许函数重载，所以函数名自身并不唯一地标识将要被调用的函数，解析器必须根据提供的参数类型选择正确的函数。

#### 操作符

LightDB允许带有前缀（单目）操作符的表达式，也允许中缀（两个参数）操作符。像函数一样，操作符也可以被重载，因此操作符的选择也有同样的问题。

#### 值存储

SQL INSERT和UPDATE语句将表达式的结果放入表中。语句中的表达式类型必须和目标列的类型一致（或者可以被转换为一致）。

#### UNION、CASE和相关结构

因为来自一个联合的SELECT语句中的所有查询结果必须在一个列集中显示，所以每个SELECT子句的结果类型必须能相互匹配并被转换成一个统一的集合。类似地，一个CASE结构的结果表达式必须被转换成一种公共的类型，这样CASE表达式作为整体才有一种已知的输出类型。其他一些结构，例如ARRAY[]和GREATEST和LEAST函数，同样需要确定几个子表达式的公共类型。

系统目录存储有关哪些数据类型之间存在哪种转换（或造型）以及如何执行这些转换的相关信息。额外的造型可以由用户通过CREATE CAST命令增加（这个通常和定义一种新的数据类型一起完成。内建的类型转换集已经经过了仔细的雕琢，最好不要去更改它们）。

解析器提供了一种额外的启发式规则，它允许在具有隐式造型的类型组中恰当造型行为的改进决定。数据类型被分为几个基本的类型分类，包括boolean、numeric、string、bitstring、datetime、timespan、geometric、network和用户自定义（可参阅Table 48.7中的列表；但需要注意的是也可以创建自定义的类型分类）。在每个分类中，可以有一个或多个首选类型，当存在类型选择时，这个是更好的选择。利用精心选择的首选类型和可用的隐式造型，我们可以确保有歧义的表达式（那些有多个候选解析方案的表达式）可以用一种有用的方式来处理。

所有类型转换规则都是建立在下面几个基本原则上的：

- 隐式转换决不能有意外的或不可预见的输出。
- 如果一个查询不需要隐式类型转换，解析器或执行器不应该有额外的开销。也就是说，如果一个查询是结构良好的并且类型已经匹配，则查询不应该在解析器里耗费额外的时间执行，也不会查询中引入不必要的隐式类型转换调用。
- 另外，如果一个查询通常要求为某个函数进行隐式类型转换，而用户定义了一个有正确参数类型的新函数，解析器应该使用新函数并不再做隐式转换来使用旧函数。

## 10.2. 操作符

被一个操作符表达式引用的特定操作符由下列过程决定。注意这个过程会被所涉及的操作符的优先级间接地影响，因为这将决定哪些子表达式被用作哪个操作符的输入。详见Section 4.1.6

### 操作符类型决定

1. 从系统目录pg\_operator中选出要考虑的操作符。如果使用了一个不带模式限定的操作符名（常见的情况），那么操作符被认为是那些在当前搜索路径中可见并有匹配的名字和参数个数的操作符（参见Section 5.9.9。如果给出一个被限定的操作符名，那么只考虑指定模式中的操作符。
  - (Optional) 如果搜索路径找到了多个有相同参数类型的操作符，那么只考虑最早出现在路径中的那一个。但是不同参数类型的操作符将被平等看待，而不管它们在路径中的位置如何。
2. 查找一个正好接受输入参数类型的操作符。如果找到一个（在一组被考虑的操作符中，可能只存在一个正好匹配的），则使用之。在通过限定名称（非典型）调用在一个允许不可信用户创建对象的方案中找到的任意操作符时，精确匹配的缺失会导致安全性危害<sup>1</sup>。在这样的情况下，应该造型参数以便强制一次精确匹配。
  - a. (Optional) 在兼容Oracle或MySQL类型（参见lightdb\_syntax\_compatible\_type）的情况下，如果二元运算符调用的一个参数是unknown类型，而另一个参数是数字类型（int2、int4、int8、float4、float8），则假定unknown类型为text类型。如果存在一个接受text类型和number类型精确匹配的运算符，则使用它。
  - b. (Optional) 如果一个二元操作符调用中的一个参数是unknown类型，则在本次检查中假设它与另一个参数类型相同。对于涉及两个unknown输入的调用或者带有一个unknown输入的前缀操作符，在这一步将永远找不到一个匹配。
  - c. (Optional) 如果一个二元操作符调用的其中一个参数是unknown类型而另一个是一种域类型，下一次检查会看看是否有一个操作符正好在两边都接受该域的基类型，如果有就使用它。

<sup>1</sup> 对非方案限定的名称，不会出现这种危害，因为包含允许不可信用户创建对象的方案的搜索路径不是一种安全的方案使用模式。

3. 寻找最优匹配。
  - a. 抛弃那些输入类型不匹配并且也不能被转换成匹配的候选操作符。 unknown文字被假定为可以为这个目的被转换为任何东西。如果只剩下一个候选操作符，则使用之，否则继续下一步。
  - b. 如果任何输入参数是一种域类型，对所有后续步骤都把它当做是该域的基类型。这确保在做有歧义的操作符解析时，域的举止像它们的基类型。
  - c. 遍历所有候选操作符，保留那些在输入类型上的匹配最准确的。如果没有一个操作符能准确匹配，则保留所有候选。如果只剩下一个候选操作符，则使用之，否则继续下一步。
  - d. 遍历所有候选操作符，保留那些在最多需要类型转换的位置上接受首选类型（属于输入数据类型的类型分类）的操作符。如果没有接受首选类型的操作符，则保留所有候选。如果只剩下一个候选操作符，则使用之，否则继续下一步。
  - e. 如果有任何输入参数是unknown类型，检查被剩余候选操作符在那些参数位置上接受的类型分类。在每一个位置，如果任何候选接受该分类，则选择string分类（这种对字符串的偏爱合适的，因为未知类型的文本确实像字符串）。否则，如果所有剩下的候选操作符都接受相同的类型分类，则选择该分类；否则抛出一个错误（因为在没有更多线索的条件下无法作出正确的推断）。现在抛弃不接受选定的类型分类的候选操作符。然后，如果任意候选操作符接受那个分类中的首选类型，则抛弃那些在该参数位置接受非首选类型的候选操作符。如果没有候选操作符能通过这些测试则保留全部候选者。如果只剩下一个候选者，则使用之；否则继续下一步。
  - f. 如果既有unknown参数也有已知类型的参数，并且所有已知类型参数具有相同的类型，则假定该unknown参数也是那种类型的，并且检查哪些候选操作符可以在该unknown参数的位置上接受那个类型。如果正好有一个候选者通过了这个测试，则使用之；否则失败。
  - g. 请参考Section 10. 中描述的类型兼容匹配方式。

下面是一些例子。

#### Example 10.1. 平方根运算符类型解析

在标准目录中只有一个被定义的平方根操作符（前缀|/），它接受一个类型为double precision的参数。在下面这个查询表达式中，扫描器会为该参数分配一个初始类型integer:

```
SELECT |/ 40 AS "square root of 40";
square root of 40
-----
6.324555320336759
(1 row)
```

因此，解析器在操作数上做了一个类型转换，该查询等价于:

```
SELECT |/ CAST(40 AS double precision) AS "square root of 40";
```

#### Example 10.2. 字符串连接操作符类型决定

一个类字符串的语法被用来处理字符串类型和处理复杂的扩展类型。未指定类型的字符串与可能的候选操作符匹配。

一个未指定参数的例子:

```
SELECT text 'abc' || 'def' AS "text and unknown";
```

```
text and unknown
-----
abcdef
(1 row)
```

在这种情况下，解析器查看是否有一个操作符的两个参数都使用text。既然有，那么它假设第二个参数应被解释为text类型。

下面是两个未指定类型的值的连接：

```
SELECT 'abc' || 'def' AS "unspecified";
```

```
unspecified
-----
abcdef
(1 row)
```

在这种情况下，没有对于使用哪种类型的初始提示，因为在查询中没有指定类型。因此，解析器查找所有的候选操作符并找到候选者同时接受字符串分类和位串分类的输入。因为字符串分类在可用时是首选的，该分类会被选中，并且接下来字符串的首选类型（text）会被用作解决未知类型文字的指定类型。

### Example 10.3. 绝对值与否定操作符类型决定

LightDB操作符目录中有几个对于前缀操作符@的条目，这些都现实了针对各种数字数据类型绝对值操作。其中之一用于float8类型，它是在数字分类中的首选类型。因此，LightDB将在遇到一个unknown输入时使用它：

```
SELECT @ '-4.5' AS "abs";
```

```
abs
-----
4.5
(1 row)
```

在这里，系统在应用所选操作符之前已经隐式地解决了将未知类型文字作为float8类型。我们可以验证我们使用的是float8而不是别的类型：

```
SELECT @ '-4.5e500' AS "abs";
```

```
ERROR: "-4.5e500" is out of range for type double precision
```

### Example 10.4. 数组包含操作符类型决定

这里是另一个决定带有一个已知和一个未知输入的操作符的例子：

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";
```

```
is subset
-----
t
(1 row)
```

LightDB操作符目录有一些条目用于中缀操作符<@，但是仅有的两个可以在左手边接受一个整数数组的是数组包含（anyarray <@ anyarray）和范围包含（anyelement <@

anyrange)。因为这些多态伪类型（见Section 8.2）中没有一个被认为是首选的，解析器不能以此为基础来解决歧义。不过，Step 3.f告诉它假定位置类型的文字和其他输入的类型相同，即整数数组。现在这两个操作符中只有一个可以匹配，因此数组包含被选择（如果选择范围包含，我们将得到一个错误，因为该字符串没有成为一个范围文字的正确格式）。

### Example 10.5. 域类型上的自定义操作符

用户有时会尝试声明只适用于一种域类型的操作符。这是可能的，但是远非它看起来那么有用，因为操作符解析规则被设计为选择适用于域的基类型的操作符。考虑这个例子：

```
CREATE DOMAIN mytext AS text CHECK(...);
CREATE FUNCTION mytext_eq_text (mytext, text) RETURNS boolean AS ...;
CREATE OPERATOR = (procedure=mytext_eq_text, leftarg=mytext, rightarg=text);
CREATE TABLE mytable (val mytext);
```

```
SELECT * FROM mytable WHERE val = 'foo';
```

这个查询将不会使用自定义操作符。解析器将首先看看是否有一个 `mytext = mytext` 操作符（Step 2.b），当然这里没有；然后它将会考虑该域的基类型 `text`，并且看看是否有一个 `text = text` 操作符（Step 2.c），这里也没有；因此它会把 `unknown`-类型文字解析为 `text` 并使用 `text = text` 操作符。让自定义操作符能被使用的唯一方法是显式地转换改变文字：

```
SELECT * FROM mytable WHERE val = text 'foo';
```

这样根据准确匹配规则会立即找到 `mytext = text` 操作符。如果到达最佳匹配规则，它们会积极地排斥域类型上的操作符。如果它们没有，这样一个操作符将创建太多歧义操作符失败，因为转换规则总是认为一个域可以和它的基类型相互转换，并且因此该域操作符在所有与该基类型上的一个类似命名的操作符相同的情况中都被认为可用。

## 10.3. 函数

被一个函数调用引用的特定函数使用下面的过程来决定。

### 函数类型决定

1. 从 `pg_proc` 系统目录中选择要被考虑的函数。如果使用一个非模式限定的函数名称，那么函数被认为是那些在当前搜索路径中可见并有匹配的名字和参数个数的函数（参见Section 5.9.3）。如果给出一个被限定的函数名，那么只考虑指定模式中的函数。
  - a. (Optional) 如果搜索路径发现多个参数类型相同的函数，那么只考虑最早在搜索路径中出现的那个。不同参数类型的函数被平等对待，不受在搜索路径中位置的影响。
  - b. (Optional) 如果使用一个 `VARIADIC` 数组参数声明一个函数，并且调用不使用关键字 `VARIADIC`，那么该函数就好像其数组参数被它的元素类型的一次或多次出现所替换，根据需去匹配调用。这样的扩展之后，函数可能会有和非可变函数相同的参数类型。在这种情况下，在搜索路径中出现比较早的函数将被使用，或者如果两个函数在相同的模式中时首选非可变的那一个。

在通过限定名称调用在一个允许不可信用户创建对象的方案中找到的可变函数时，会导致安全性危害<sup>2</sup>。恶意用户可以拿到控制权并且执行任意SQL函数（就好像你在执行它们一样）。将涉及 `VARIADIC` 关键词的调用替换掉就可以绕过这种危害。涉

<sup>2</sup> 对非方案限定的名称，不会出现这种危害，因为包含允许不可信用户创建对象的方案的搜索路径不是一种安全的方案使用模式。

及到VARIADIC “any”参数的调用通常没有等效的包含VARIADIC关键词的形式。为了安全地发出那些调用，函数的方案必须只允许可信用户创建对象。

- c. (Optional) 考虑使用有默认参数值的函数来匹配任何省略了零个或者多个可默认参数位置的调用。如果有超出一个的这种函数匹配一个调用，那么使用最早出现在搜索路径中的那个。如果同一个模式中在同一个非默认位置上有两个或者更多这样的函数（如果它们有不同的默认参数设置，这是可能的），系统将不能确定去选择哪一个，并且如果不能找到该调用更好的匹配，将会导致一个“有歧义的函数调用”错误。

在通过限定名称<sup>2</sup>调用在一个允许不可信用户创建对象的方案中找到的任意函数时，会导致可用性危害。恶意用户可以用一个已有函数的名称创建一个函数，复制该函数的参数并且追加新的具有默认值的参数。这会妨碍对原始函数的新调用。为了防止这种危害，应将函数放在仅允许可信用户创建对象的方案中。

2. 检查一个函数正好接受输入参数类型。如果存在一个（在所考虑的一组函数中只能有一个准确匹配），则使用之。在通过限定名称<sup>2</sup>调用在一个允许不可信用户创建对象的方案中找到的函数时，精确匹配的缺失会导致安全性危害。在这样的情况下，应该造型参数以便强制一次精确匹配（在该步骤中，涉及unknown的情况将永远找不到一个匹配）。
3. 如果没有发现准确匹配，那么查看函数调用是否作为一个特定的类型转换请求出现。如果函数调用仅有一个参数并且函数名和一些数据类型的（内部）名称相同，那么该情况将会发生。并且，该函数参数必须是一个未知类型的文字，或者是一个可以被二进制强制转换到命名数据类型的类型，或者是一个可以通过应用其I/O函数被转换为命名数据类型的类型（也就是，转换是转到标准字符串类型或者从标准字符串类型转来）。当满足这些条件的时候，函数调用被当做CAST声明的一种形式来对待。<sup>3</sup>
4. 查找最佳匹配。
  - a. 如果候选函数的输入类型不匹配并且不能通过转换（使用一个隐式转换）达到匹配，则丢弃它。为了这个目的，unknown文字被假定可被转换成任何东西。如果仅有一个候选项，则使用之；否则继续下一步。
  - b. 如果任何输入参数是一种域类型，在所有后续步骤中都把它当做该域的基类型。这确保在做有歧义的操作符解析时，域的举止像它们的基类型。
  - c. 遍历所有候选函数并保留那些最匹配输入类型的。如果没有准确匹配，则保留所有候选项。如果仅有一个候选项，则使用之；否则继续下一步。
  - d. 遍历所有候选函数并保留那些在最多要求类型转换的位置上接受首选类型（属于输入数据类型的类型分类）的候选项。如果没有接受首选类型的候选项，则保留所有候选项。如果仅有一个候选项，则使用之；否则继续下一步。
  - e. 如果任何输入参数是unknown，那么检查那些被剩余候选项在那些参数位置上接受的类型分类。在每一个位置上，如果任何候选项接受该分类则选择string分类（这个偏向于字符串是恰当的，因为一个未知类型文字看起来像字符）。否则，如果所有剩余的候选项接受相同的类型分类，那么选择那个分类；否则将失败，因为缺乏更多线索来推断出正确的选择。现在，丢弃不接受被选中类型分类的候选项。此外，如果任何候选项接受那个分类中的一个首选类型，则丢弃对该参数接受非首选类型的候选项。如果没有候选项能通过这些测试，则保留所有候选项。如果只剩下一个候选项，则使用之；否则继续下一步。
  - f. 如果既有unknown参数也有已知类型的参数，并且所有已知类型参数具有相同的类型，则假定该unknown参数也是那种类型的，并且检查哪些候选函数可以在该unknown参数的位置上接受那个类型。如果正好有一个候选者通过了这个测试，则使用之；否则失败。

<sup>3</sup> 这一步的原因是在没有一个实际的造型函数的情况下支持函数风格的造型声明。如果有一个造型函数，它被按惯例以其输出类型命名，并且不需要有特殊情况。更多信息请见CREATE CAST。

g. 请参考Section 10. 中描述的类型兼容匹配方式。

注意，对于操作符和函数类型决定来说“最优匹配”规则是完全相同的。下面是一些例子。

### Example 10.6. 圆整函数参数类型决定

只有一个带有两个参数的圆整函数；它采用第一个参数类型为numeric和第二个参数类型为integer。这样下面的查询自动将第一个类型为integer参数转换为numeric：

```
SELECT round(4, 4);
```

```
round
-----
4.0000
(1 row)
```

该查询实际上被解析器转换为：

```
SELECT round(CAST (4 AS numeric), 4);
```

因为包含小数点的数字常数初始会被分配类型numeric，下面的查询将不需要类型转换并因此可能会稍稍高效一些：

```
SELECT round(4.0, 4);
```

### Example 10.7. 可变函数决定

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

这个函数接受（但不要求）VARIADIC关键词。它能同时容忍integer以numeric参数：

```
SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
variadic_example | variadic_example | variadic_example
-----
1 | 1 | 1
(1 row)
```

不过，如果可以，第一个和第二个调用将更喜欢更明确的函数：

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION
```

```
CREATE FUNCTION public.variadic_example(int) RETURNS int
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION
```

```
SELECT public.variadic_example(0),
       public.variadic_example(0.0),
```



```

public.variadic_example(VARIADIC array[0.0]);
variadic_example | variadic_example | variadic_example
-----
3 | 2 | 1
(1 row)

```

如果给定默认的配置并且只有第一个函数存在，则第一个和第二个调用是不安全的。任何用户都可以通过创建第二个或者第三个函数来截断它们。通过精确匹配参数类型并且使用VARIADIC关键词，第三个调用是安全的。

### Example 10.8. 子串函数类型决定

有几个substr函数，其中一个用于text和integer类型。如果使用一个未指定类型的字符常量调用，那么系统选择接受一个首选分类string（也就是text类型）的参数的候选函数。

```
SELECT substr('1234', 3);
```

```

substr
-----
34
(1 row)

```

如果字符串被声明为类型varchar（如果它来自于一个表就会这样），那么解析器将尝试转换它为text：

```
SELECT substr(vchar '1234', 3);
```

```

substr
-----
34
(1 row)

```

解析器所作的转换：

```
SELECT substr(CAST (vchar '1234' AS text), 3);
```

#### Note

解析器从pg\_cast目录中知道text和varchar是二进制可兼容的，意思是其中一个可以被传递给接受另一种类型的函数而不需要做任何物理转换。因此，在这种情况下不会真正使用类型转换调用。

### Example 10.9. 添加月份函数类型解析。

orafce扩展提供了函数add\_months的两个版本，一个在模式pg\_catalog下，另一个在模式oracle下。

这两个函数之间存在一些细微的差异：`pg_catalog.add_months`接受date和int类型的参数，并返回date类型；`oracle.add_months`接受timestamp with time zone和int类型的参数，并返回timestamp类型。

```
CREATE FUNCTION pg_catalog.add_months(day date, value int)
RETURNS date ...
```

```
select pg_catalog.add_months('2021-12-23', 4);
add_months
-----
2022-04-23
(1 row)
```

```
CREATE FUNCTION oracle.add_months(TIMESTAMP WITH TIME ZONE, INTEGER)
RETURNS TIMESTAMP ...
```

```
select oracle.add_months('2021-12-23', 4);
add_months
-----
2022-04-23 00:00:00
(1 row)
```

当使用未知类型调用add\_months函数时，根据search\_path不同，会匹配到不同的函数。

当模式oracle不在search\_path中时，默认只在模式pg\_catalog中查找add\_months函数。

当模式oracle在search\_path中时，将有两个版本的add\_months函数可用。然而，oracle.add\_months参数类型timestamp with time zone是首选类型，因此将选择oracle.add\_months。

```
select add_months('2021-12-23', 4);
add_months
-----
2022-04-23
(1 row)

set search_path="$user",public,oracle;

select add_months('2021-12-23', 4);
add_months
-----
2022-04-23 00:00:00
(1 row)
```

## 10.4. 值存储

将被插入到一个表的值会按照下列步骤被转换到目标列的数据类型。

### 值存储类型转换

1. 检查一个与目标的准确匹配。
2. 否则，尝试转换表达式为目标类型。如果在两种类型之间的一个赋值造型已经被注册在pg\_cast目录（见CREATE CAST）中，这是可能的。或者，如果该表达式是一个未知类型的文字，则该文字串的内容将被提供给目标类型的输入转换例程。
3. 检查是否有一个用于目标类型的尺寸调整造型。尺寸调整造型是一个从该类型到其自身的造型。如果在pg\_cast目录中找到一个，那么把表达式存储到目标列中之前把它应用到表达式。这样一个造型的实现函数总是采用一个额外的integer类型的参数，它接收目标列的atttypmod值（通常是它被声明的长度，尽管对于不同数据类型atttypmod有不同的解释），并且它可能采用第三个boolean参数来说明造型是显式的还是隐式的。该造型函数负责应用任何长度相关的语义，例如尺寸检查或截断。

## Example 10.10. character存储类型转换

对于一个声明为character(20)的目标列，下面的语句展示了被存储的值如何被正确地调整尺寸：

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;
```

v	octet_length
abcdef	20

(1 row)

实际发生的事情是两个未知文字被默认决定为text，允许||操作符被决定为text连接。然后操作符的text结果被转换成bpchar（“空白填充字符”，character数据类型的内部名称）来匹配目标列类型（由于从text到bpchar的转换是二进制强制的，这个转换不会插入任何实际的函数调用）。最后，尺寸调整函数bpchar(bpchar, integer, boolean)被从系统目录中找到并应用到操作符的结果和存储的列长度上。这个类型相关的函数执行必要的长度检查并增加填充的空间。

## 10.5. UNION、CASE和相关结构

SQL UNION结构必须使可能不相似的类型匹配成为一个单一的结果集。该决定算法被独立地应用到一个联合查询的每个输出列。INTERSECT和EXCEPT采用和UNION相同的方法来决定不相似的类型。其他一些结构，包括CASE、ARRAY、VALUES，以及GREATEST和LEAST函数，使用相同的算法来使它们的组成表达式匹配并选择一种结果数据类型。

### UNION、CASE和相关结构的类型决定

1. 如果所有的输入为相同类型，并且不是unknown，那么就决定是该类型。
2. 如果任何输入是一种域类型，在所有后续步骤中都把它当做该域的基类型。<sup>4</sup>
3. 如果所有的输入为unknown类型，则决定为text（字符串分类的首选类型）类型。否则，为了剩余规则，unknown输入会被忽略。
4. 如果非未知输入不全是相同的类型分类，则失败。
5. 选择第一个非未知输入类型作为候选类型，然后从左到右考虑其他非未知输入类型。<sup>5</sup>如果候选类型可以隐式转换为其他类型，但反之不行，则选择其他类型作为新的候选类型。然后继续考虑剩余的输入。如果在此过程的任何阶段选择了首选类型，请停止考虑其他输入。
6. 将所有输入转换为最终候选类型。如果没有从给定输入类型到候选类型的隐式转换，则失败。

下面是一些例子。

### Example 10.11. 联合中未指定类型的类型决定

```
SELECT text 'a' AS "text" UNION SELECT 'b';
```

<sup>4</sup> 多少有些类似于对待用于操作符和函数的域输入的方式，这种行为允许一种域类型能通过一个UNION或相似的结构保留下来，只要用户小心地确保所有的输入都是（显式地或隐式地）准确类型。否则会使用该域的基类型。

<sup>5</sup> 出于历史原因，CASE将其ELSE子句（如果有的话）视为“第一个”输入，之后再考虑THEN子句。在所有其他情况下，“从左到右”表示表达式在查询文本中出现的顺序。

```

text
-----
a
b
(2 rows)

```

这里，未知类型文字' b' 将被决定为类型text。

#### Example 10.12. 简单联合中的类型决定

```
SELECT 1.2 AS "numeric" UNION SELECT 1;
```

```

numeric
-----
      1
     1.2
(2 rows)

```

文字1.2是numeric类型，且integer值1可以被隐式地造型为numeric，因此使用numeric类型。

#### Example 10.13. 可换位联合中的类型决定

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);
```

```

real
-----
      1
     2.2
(2 rows)

```

这里，由于类型real能被隐式地造型为integer，而integer可以被隐式地造型为real，联合结果类型被决定为real。

#### Example 10.14. 嵌套合并中的类型决定

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;
```

```
ERROR: UNION types text and integer cannot be matched
```

这个失败发生的原因是LightDB把多个UNION当作是成对操作的嵌套，也就是说上面的输入等同于：

```
(SELECT NULL UNION SELECT NULL) UNION SELECT 1;
```

根据上面给定的规则，内层的UNION被确定为类型text。然后外层的UNION的输入是类型text和integer，这就导致了上面看到的错误。通过确保最左边的UNION至少有一个输入类型为想要的结果类型，就可以修正这个问题。

INTERSECT和EXCEPT操作也被当作成对操作。不过，这一节中描述的其他结构会在一个决定步骤中考虑所有的输入。

## 10.6. SELECT的输出列

前面的小节中给出的规则将会导致对SQL查询中的所有表达式分配非unknown数据类型，不过作为SELECT命令的简单输出列出现的未指定类型的文本除外。例如，在

```
SELECT 'Hello World';
```

中没有标识该字符串应该取何种类型。在这种情况下，LightDB将会退而求其次将其类型决定为text。

当SELECT处于UNION（或者INTERSECT，或者EXCEPT）结构的一边或者出现在INSERT ... SELECT中时，这条规则就不适用了，因为在前面小节中给出的规则会优先。在第一种情况下未指定类型文本的类型将从UNION的另一边取得，而在第二种情况下未指定类型文本的类型将从目标列取得。

出于这样的目的，RETURNING列表采用和SELECT输出列表同样的方式对待。

## 10.7. 类型兼容性

LightDB 支持字符串和数字类型之间的特殊兼容性。当隐式转换规则不能找到合适的运算符或函数（Section 10.2 Section 10.3 时，将自动应用显式类型转换，并应用类型优先级来选择适当的运算符或函数。

这个特殊的兼容性特性包括9个类型，并默认按优先级排序。每个类型都支持隐式类型转换到其他类型。

以下是默认的9个类型（按优先级降序排列）：

```
numeric > double precision > real > bigint > integer > smallint > text > varchar
> char
```

您可以使用ALTER TYPE修改或添加类型的优先级，内置类型无法修改，请参阅ALTER TYPE。

### Note

优先级的值必须为正数。

#### Example 10.15. 修改或添加类型优先级

```
ALTER TYPE varchar2 set (priority=200)
```

#### 兼容性类型解析

1. 在遵循Section 10.7和Section 10.8中描述的规则之后，执行以下流程。
2. 忽略所有具有伪类型参数的候选项。
3. 所有的unknown-type将被视为text类型。
4. 运行所有候选项，并保留输入类型上显示转换最少的那些。如果只剩下一个候选项，则使用它；否则保留所有剩余的候选项并继续下一步操作。
5. 运行所有剩余的候选项，如果所有显式转换位置都具有相同的参数类型，则候选项的优先级将与参数类型的优先级相同。如果候选项的显式转换位置有多个参数类型，则优先级将低于所有类型。在运算符中，如果一个参数是数字，另一个参数是字符串，则数字类型的优先级将高于所有类型。

6. 运行所有剩余的候选项，保留那些最多位置是精确匹配的候选项。如果只剩下一个候选项，则使用它；否则继续下一步操作。
7. 运行所有剩余的候选项，从左到右比较候选项在相同位置上的参数类型优先级，保留具有最高优先级的候选项。

Example 10.16. 自动应用显式转换。

```
select |/'20'::text;

?column?
-----
4.47213595499958
(1 row)
```

在这个例子中，text类型不能隐式转换为double precision类型，但是根据类型兼容性规则，数字和字符串类型之间可以自动应用显式转换。

Example 10.17. 类型优先级

```
select ~ '20';

?column?
-----
-21
(1 row)
```

在这个例子中，输入类型是未知类型，因此将输入类型视为文本类型，并选择具有最高优先级参数的候选项，使用~ bigint。

如果输入值无法转换为bigint，将报告类型转换错误。

```
select ~ 'a';

ERROR: invalid input syntax for type bigint: "a"
LINE 1: select ~ 'a';
```

Example 10.18. 子字符串函数类型解析

如果使用integer类型的参数调用substr函数，LightDB 将尝试将其转换为text类型：

```
SELECT substr(1234, 3);
 substr
-----
34
(1 row)
```

Example 10.19. 参数列表优先级

```
create function testfun(text, integer)
  returns text as $$ select 'text, integer' $$ language sql;

create function testfun(integer, text)
  returns text as $$ select 'integer, text' $$ language sql;
```

```
select testfun('1'::text, '1'::text);
```

没有一个参数更适合，因此从左到右比较参数的优先级。现在第一个参数是整数和文本，整数优先于文本，使用testfun(integer, text)。

#### Example 10.20. 插入和更新

```
create table tb(name varchar, age int);

insert into tb(name,age) values(1223, '23'::varchar);
INSERT 0 1

update tb set name=1343,age='24'::text;
UPDATE 1

select * from tb;
 name | age
-----+-----
 1343 |  24
(1 row)
```

列类型和值类型不匹配，并且LightDB默认情况下没有值类型和列类型之间的隐式转换规则。但是，根据兼容性规则，LightDB可以在其上使用显式转换规则。

#### Note

类型兼容性适用于运算符、函数、插入和更新。不适用于SELECT UNION, CASE (Section 10.5)。

---

# Chapter 11. 索引

索引是提高数据库性能的常用途径。比起没有索引，使用索引可以让数据库服务器更快找到并获取特定行。但是索引同时也会增加数据库系统的日常管理负担，因此我们应该聪明地使用索引。

## 11.1. 简介

假设我们有一个如下的表：

```
CREATE TABLE test1 (  
    id integer,  
    content varchar  
);
```

而应用发出很多以下形式的查询：

```
SELECT content FROM test1 WHERE id = constant;
```

在没有事前准备的情况下，系统不得不扫描整个test1表，一行一行地去找到所有匹配的项。如果test1中有很多行但是只有一小部分行（可能是0或者1）需要被该查询返回，这显然是一种低效的方式。但是如果系统被指示维护一个在id列上的索引，它就能使用一种更有效的方式来定位匹配行。例如，它可能仅仅需要遍历一棵搜索树的几层而已。

类似的方法也被用于大部分非小说书籍中：经常被读者查找的术语和概念被收集在一个字母序索引中放在书籍的末尾。感兴趣的读者可以相对快地扫描索引并跳到合适的页而不需要阅读整本书来寻找感兴趣的材料。正如作者的任务是准备好读者可能会查找的术语一样，数据库程序员也需要预见哪些索引会有用。

正如前面讨论的，下列命令可以用来在id列上创建一个索引：

```
CREATE INDEX test1_id_index ON test1 (id);
```

索引的名字test1\_id\_index可以自由选择，但我们最好选择一个能让我们想起该索引用途的名字。

删除一个索引，使用DROP INDEX命令。索引可以随时被创建或删除。

一旦一个索引被创建，就不再需要进一步的干预：系统会在表更新时更新索引，而且会在它觉得使用索引比顺序扫描表效率更高时使用索引。但我们可能需要定期地运行ANALYZE命令来更新统计信息以便查询规划器能做出正确的决定。通过Chapter 11的信息可以了解如何找出一个索引是否被使用以及规划器在何时以及为什么会选择不使用索引。

索引也会对带有搜索条件的UPDATE和DELETE命令有帮助。此外索引还可以在连接搜索中使用。因此，一个定义在连接条件列上的索引可以显著地提高连接查询的速度。

在一个大表上创建一个索引会耗费很长的时间。默认情况下，LightDB允许在索引创建时并行地进行读（SELECT命令），但写（INSERT、UPDATE和DELETE）则会被阻塞直到索引创建完成。在生产环境中这通常是不可接受的。在创建索引时允许并行的写是可能的，但是有些警告需要注意，更多信息可以参考Building Indexes Concurrently。

一个索引被创建后，系统必须保持它与表同步。这增加了数据操作的负担。因此那些很少或从不在查询中使用的索引应该被移除。



## 11.2. 索引类型

LightDB提供了多种索引类型：B-tree、Hash、GiST、GIN。每一种索引类型使用了一种不同的算法来适应不同类型的查询。默认情况下，CREATE INDEX 命令创建适合于大部分情况的B-tree 索引。在索引类型名后面加上关键字USING，可以选择其他的索引类型，例如，创建一个HASH索引：

```
CREATE INDEX name ON table USING HASH (column);
```

B树可以处理可以排序的数据上的等值和范围查询。特别是，在涉及使用以下运算符之一进行比较的索引列时，LightDB查询规划器将考虑使用B树索引：

```
<
<=
=
>=
>
```

这些运算符的组合等效结构，如BETWEEN和IN，也可以使用B树索引搜索实现。此外，对索引列的IS NULL或IS NOT NULL条件可以与B树索引一起使用。

优化器也会将B-tree索引用于涉及到模式匹配操作符LIKE和~ 的查询，前提是如果模式是一个常量且被固定在字符串的开头—例如：col LIKE 'foo%' 或者col ~ '^foo'，但在col LIKE '%bar' 上则不会。但是，如果我们的数据库没有使用C区域设置，我们需要创建一个具有特殊操作符类的索引来支持模式匹配查询，参见下面的Section 11.10 同样可以将B-tree索引用于ILIKE和~\*，但仅当模式以非字母字符开始，即不受大小写转换影响的字符。

B-tree索引也可以用于检索排序数据。这并不会总是比简单扫描和排序更快，但是总是有用的。

哈希索引只能处理简单的等值比较。查询规划器将在涉及使用=运算符进行比较的索引列时，考虑使用哈希索引。以下命令用于创建哈希索引：

```
CREATE INDEX name ON table USING HASH (column);
```

GiST索引不是单一类型的索引，而是一个基础设施，可以在其中实现许多不同的索引策略。因此，可以使用GiST索引的特定运算符取决于索引策略（运算符类）。例如，LightDB的标准分发版包括用于几种二维几何数据类型的GiST运算符类，这些运算符支持使用以下运算符进行索引查询：

```
<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&
```

（有关这些运算符的含义，请参见Section 9.11 ） 标准分发版中包含的GiST运算符类在Table 61. 中有文档记录。许多其他GiST运算符类可在contrib集合或作为单独的项目中使用。有关更多信息，请参见Chapter 61

GiST索引还能够优化“最近邻”搜索，例如：

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

它将找到离给定目标点最近的10个位置。能够支持这种查询的能力同样取决于被使用的特定操作符类。在Table 61. 中，“Ordering Operators”列中列出了可以在这种方法中使用的操作符。

GIN索引是逆向索引，适用于包含多个组件值的数据值，例如数组。逆向索引包含每个组件值的单独条目，并且可以高效地处理测试特定组件值存在性的查询。

与GiST类似，GIN可以支持许多不同的用户定义索引策略，可以使用GIN索引的特定运算符取决于索引策略。例如，LightDB的标准分发版包括用于数组的GIN运算符类，这些运算符支持使用以下运算符进行索引查询：

```
<@
@>
=
&&
```

（有关这些运算符的含义，请参见Section 9.19）标准分发版中包含的GIN运算符类在Table 62. 中有文档记录。许多其他GIN运算符类可在contrib集合或作为单独的项目中使用。有关更多信息，请参见Chapter 62

## 11.3. 多列索引

一个索引可以定义在表的多个列上。例如，我们有这样一个表：

```
CREATE TABLE test2 (
    major int,
    minor int,
    name varchar
);
```

（即将我们的/dev目录保存在数据库中）而且我们经常会做如下形式的查询：

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

那么我们可以在major和minor上定义一个索引：

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

目前，只有 B-tree、GiST、GIN 索引类型支持多列索引。是否可以有多个关键列与INCLUDE列是否可以被添加到索引中无关。索引最多可以有32列，包括INCLUDE列。（该限制可以在源代码文件pg\_config\_manual.h中修改，但是修改后需要重新编译LightDB）。

一个B-tree索引可以用于条件中涉及到任意索引列子集的查询，但是当先导列（即最左边的那些列）上有约束条件时索引最为有效。确切的规则是：在先导列上的等值约束，加上第一个无等值约束的列上的不等值约束，将被用于限制索引被扫描的部分。在这些列右边的列上的约束将在索引中被检查，这样它们适当节约了对表的访问，但它们并未减小索引被扫描的部分。例如，在(a, b, c)上有一个索引并且给定一个查询条件WHERE a = 5 AND b >= 42 AND c < 77，对索引的扫描将从第一个具有a = 5和b = 42的项开始向上进行，直到最后一个具有a = 5的项。在扫描过程中，具有c >= 77的索引项将被跳过，但是它们还是会被扫描到。这个索引在原则上可以被用于在b和/

或c上有约束而在a上没有约束的查询，但是整个索引都不得被扫描，因此在大部分情况下规划器宁可使用一个顺序的表扫描来替代索引。

一个多列GiST索引可以用于条件中涉及到任意索引列子集的查询。在其余列上的条件将限制由索引返回的项，但是第一列上的条件是决定索引上扫描量的最重要因素。当第一列中具有很少的可区分值时，一个GiST索引将会相对比较低效，即便在其他列上有很多可区分值。

一个GIN索引可以用于条件中涉及到任意索引列子集的查询。与B-tree和GiST不同，GIN的搜索效率与查询条件中使用哪些索引列无关。

当然，要使索引起作用，查询条件中的列必须要使用适合于索引类型的操作符，使用其他操作符的子句将不会被考虑使用索引。

多列索引应该较少地使用。在绝大多数情况下，单列索引就足够了且能节约时间和空间。具有超过三个列的索引不太有用，除非该表的使用是极端程式化的。Section 11.4以及Section 11.9中有对不同索引配置优点的讨论。

## 11.4. 索引和ORDER BY

除了简单地查找查询要返回的行外，一个索引可能还需要将它们以指定的顺序传递。这使得查询中的ORDER BY不需要独立的排序步骤。在LightDB当前支持的索引类型中，只有B-tree可以产生排序后的输出，其他索引类型会把行以一种没有指定的且与实现相关的顺序返回。

规划器会考虑以两种方式来满足一个ORDER BY说明：扫描一个符合说明的可用索引，或者先以物理顺序扫描表然后再显式排序。对于一个需要扫描表的大部分的查询，一个显式的排序很可能比使用一个索引更快，因为其顺序访问模式使得它所需要的磁盘I/O更少。只有在少数行需要被取出时，索引才会更有用。一种重要的特殊情况是ORDER BY与LIMIT n联合使用：一个显式的排序将会处理所有的数据来确定最前面的n行，但如果有一个符合ORDER BY的索引，前n行将会被直接获取且根本不需要扫描剩下的数据。

默认情况下，B-tree索引将它的项以升序方式存储，并将空值放在最后(表TID被处理为其它相等条目之间的分线器列)。这意味着对列x上索引的一次前向扫描将产生满足ORDER BY x (或者更长的形式：ORDER BY x ASC NULLS LAST)的结果。索引也可以被后向扫描，产生满足ORDER BY x DESC (ORDER BY x DESC NULLS FIRST, NULLS FIRST是ORDER BY DESC的默认情况)。

我们可以在创建B-tree索引时通过ASC、DESC、NULLS FIRST和NULLS LAST选项来改变索引的排序，例如：

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

一个以升序存储且将空值前置的索引可以根据扫描方向来支持ORDER BY x ASC NULLS FIRST或ORDER BY x DESC NULLS LAST。

读者可能会疑惑为什么要麻烦地提供所有四个选项，因为两个选项连同可能的后向扫描可以覆盖所有ORDER BY的变体。在单列索引中这些选项确实有冗余，但是在多列索引中它们却很有用。考虑(x, y)上的一个两列索引：它可以通过前向扫描满足ORDER BY x, y, 或者通过后向扫描满足ORDER BY x DESC, y DESC。但是应用可能需要频繁地使用ORDER BY x ASC, y DESC。这样就没有办法从通常的索引中得到这种顺序，但是如果将索引定义为(x ASC, y DESC)或者(x DESC, y ASC)就可以产生这种排序。

显然，具有非默认排序的索引是相当专门的特性，但是有时它们会为特定查询提供巨大的速度提升。是否值得维护这样一个索引取决于我们会多频繁地使用需要特殊排序的查询。

## 11.5. 组合多个索引

只有查询子句中在索引列上使用了索引操作符类中的操作符并且通过AND连接时才能使用单一索引。例如，给定一个(a, b)上的索引，查询条件WHERE a = 5 AND b = 6可以使用该索引，而查询WHERE a = 5 OR b = 6不能直接使用该索引。

幸运的是，LightDB具有组合多个索引（包括多次使用同一个索引）的能力来处理那些不能用单个索引扫描实现的情况。系统能在多个索引扫描之间安排AND和OR条件。例如，WHERE x = 42 OR x = 47 OR x = 53 OR x = 99这样一个查询可以被分解成为四个独立的在x上索引扫描，每一个扫描使用其中一个条件。这些查询的结果将被“或”起来形成最后的结果。另一个例子是如果我们在x和y上都有独立的索引，WHERE x = 5 AND y = 6这样的查询的一种可能的实现方式就是分别使用两个索引配合相应的条件，然后将结果“与”起来得到最后的结果行。

为了组合多个索引，系统扫描每一个所需的索引并在内存中准备一个位图用于指示表中符合索引条件的行的位置。然后这些位图会被根据查询的需要“与”和“或”起来。最后，实际的表行将被访问并返回。表行将被以物理顺序访问，因为位图就是以这种顺序布局的。这意味着原始索引中的任何排序都会被丢失，并且如果存在一个ORDER BY子句就需要一个单独的排序步骤。由于这个原因以及每一个附加的索引都需要额外的时间，即使有额外的索引可用，规划器有时也会选择使用单一索引扫描。

在所有的应用（除了最简单的应用）中，可能会有多种有用的索引组合，数据库开发人员必须做出权衡以决定提供哪些索引。有时候多列索引最好，但是有时更好的选择是创建单独的索引并依赖于索引组合特性。例如，如果我们的查询中有时只涉及到列x，有时候只涉及到列y，还有时候会同时涉及到两列，我们可以选择在x和y上创建两个独立索引然后依赖索引组合来处理同时涉及到两列的查询。我们当然也可以创建一个(x, y)上的多列索引。当查询同时涉及到两列时，该索引会比组合索引效率更高，但是正如Section 11.6中讨论的，它在只涉及到y的查询中几乎完全无用，因此它不能是唯一的一个索引。一个多列索引和一个y上的独立索引的组合将会工作得很好。多列索引可以用于那些只涉及到x的查询，尽管它比x上的独立索引更大且更慢。最后一种选择是创建所有三个索引，但是这种选择最适合表经常被执行所有三种查询但是很少被更新的情况。如果其中一种查询要明显少于其他类型的查询，我们可能需要只为常见类型的查询创建两个索引。

## 11.6. 唯一索引

索引也可以被用来强制列值的唯一性，或者是多个列组合值的唯一性。

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

当前，只有B-tree能够被声明为唯一。

当一个索引被声明为唯一时，索引中不允许多个表行具有相同的索引值。空值被视为不相同。一个多列唯一索引将会拒绝在所有索引列上具有相同组合值的表行。

LightDB会自动为定义了一个唯一约束或主键的表创建一个唯一索引。该索引包含组成主键或唯一约束的所有列（可能是一个多列索引），它也是用于强制这些约束的机制。

### Note

不需要手工在唯一列上创建索引，如果那样做也只是重复了自动创建的索引而已。

## 11.7. 表达式索引

一个索引列并不一定是底层表的一个列，也可以是从表的一列或多列计算而来的一个函数或者标量表达式。这种特性对于根据计算结果快速获取表中内容是有用的。

例如，一种进行大小写不敏感比较的常用方法是使用lower函数：

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

这种查询可以利用一个建立在lower(col1)函数结果之上的索引：

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

如果我们将该索引声明为UNIQUE，它将阻止创建在col1值上只有大小写不同的行。

另外一个例子，如果我们经常进行如下的查询：

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

那么值得创建一个这样的索引：

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

正如第二个例子所示，CREATE INDEX命令的语法通常要求在被索引的表达式周围书写圆括号。而如第一个例子所示，当表达式只是一个函数调用时可以省略掉圆括号。

索引表达式的维护代价较为昂贵，因为在每一个行被插入或更新时都得为它重新计算相应的表达式。然而，索引表达式在进行索引搜索时却不需要重新计算，因为它们的结果已经被存储在索引中了。在上面两个例子中，系统将会发现查询的条件是WHERE indexedcolumn = 'constant'，因此查询的速度将等同于其他简单索引查询。因此，表达式索引对于检索速度远比插入和更新速度重要的情况非常有用。

## 11.8. 部分索引

一个部分索引是建立在表的一个子集上，而该子集则由一个条件表达式（被称为部分索引的谓词）定义。而索引中只包含那些符合该谓词的表行的项。部分索引是一种专门的特性，但在很多种情况下它们也很有用。

使用部分索引的一个主要原因是避免索引公值。由于搜索一个公值的查询（一个在所有表行中占比超过一定百分比的值）不会使用索引，所以完全没有理由将这些行保留在索引中。这可以减少索引的尺寸，同时也将加速使用索引的查询。它也将加速很多表更新操作，因为这种索引并不需要在所有情况下都被更新。Example 11. 展示了一种可能的应用：

### Example 11.1. 建立一个部分索引来排除公值

假设我们要在一个数据库中保存网页服务器访问日志。大部分访问都来自于我们组织内的IP地址，但是有些来自于其他地方（如使用拨号连接的员工）。如果我们主要通过IP搜索来自于外部的访问，我们就没有必要索引对应于我们组织内网的IP范围。

假设有这样一个表：

```
CREATE TABLE access_log (  
    url varchar,  
    client_ip inet,  
    ...  
);
```

用以下命令可以创建适用于我们的部分索引：

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
          client_ip < inet '192.168.100.255');
```

一个使用该索引的典型查询是：

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

此处查询的IP地址由部分索引覆盖。以下查询无法使用部分索引，因为它使用从索引中排除的IP地址：

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '192.168.100.23';
```

可以看到部分索引查询要求值能被预知，因此部分索引最适合于数据分布不会改变的情况。这样的索引也可以偶尔被重建来适应新的数据分布，但是这会增加维护负担。

Example 11. 展示了部分索引的另一个可能的用途：从索引中排除那些查询不感兴趣的值。这导致了上述相同的好处，但它防止了通过索引来访问“不感兴趣的”值，即便在这种情况下下一个索引扫描是有益的。显然，为这种场景建立部分索引需要很多考虑和实验。

#### Example 11.2. 建立一个部分索引来排除不感兴趣的值

如果我们有一个表包含已上账和未上账的订单，其中未上账的订单在整个表中占据一小部分且它们是最经常被访问的行。我们可以通过只在未上账的行上创建一个索引来提高性能。创建索引的命令如下：

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
WHERE billed is not true;
```

使用该索引的一个可能查询是：

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

然而，索引也可以用于完全不涉及order\_nr的查询，例如：

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

这并不如在amount列上部分索引有效，因为系统必须扫描整个索引。然而，如果有相对较少的未上账订单，使用这个部分索引来查找未上账订单将会更好。

注意这个查询将不会使用该索引：

```
SELECT * FROM orders WHERE order_nr = 3501;
```

订单3501可能在已上账订单或未上账订单中。

Example 11. 也显示索引列和谓词中使用的列并不需要匹配。LightDB支持使用任意谓词的部分索引，只要其中涉及的只有被索引表的列。然而，记住谓词必须匹配在将要受益于索引的查询中使用的条件。更准确地，只有当系统能识别查询的WHERE条件从数学上索引的谓词时，一个部分索引才能被用于一个查询。LightDB并不能给出一个精致的定理证明器来识别写成不同形式在数学上等价的表达式（一方面创建这种证明器极端困难，另一方面即便能创

建出来对于实用也过慢)。系统可以识别简单的不等蕴含，例如“ $x < 1$ ”蕴含“ $x < 2$ ”；否则谓词条件必须准确匹配查询的WHERE条件中的部分，或者索引将不会被识别为可用。匹配发生在查询规划期间而不是运行期间。因此，参数化查询子句无法配合一个部分索引工作。例如，对于参数的所有可能值来说，一个具有参数“ $x < ?$ ”的预备查询绝不会蕴含“ $x < 2$ ”。

部分索引的第三种可能的用途并不要求索引被用于查询。其思想是在一个表的子集上创建一个唯一索引，如Example 11.3所示。这对那些满足索引谓词的行强制了唯一性，而对那些不满足的行则没有影响。

### Example 11.3. 建立一个部分唯一索引

假设我们有一个描述测试结果的表。我们希望保证其中对于一个给定的主题和目标组合只有一个“成功”项，但其中可能会有任意多个“不成功”项。实现它的方式是：

```
CREATE TABLE tests (
  subject text,
  target text,
  success boolean,
  ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
WHERE success;
```

当有少数成功测试和很多不成功测试时这是一种特别有效的方法。通过创建具有IS NULL限制的惟一部分索引，也可以允许列中仅有一个空。

最后，一个部分索引也可以被用来重载系统的查询规划选择。同样，具有特殊分布的数据集可能导致系统在它并不需要索引的时候选择使用索引。在此种情况下可以被建立，这样它不会被那些无关的查询所用。通常，LightDB会对索引使用做出合理的选择（例如，它会在检索公值时避开索引，这样前面的例子只能节约索引尺寸，它并非是避免索引使用所必需的），非常不正确的规划选择则需要作为故障报告。

记住建立一个部分索引意味着我们知道的至少和查询规划器所知的一样多，尤其是我们知道什么时候一个索引会是有益的。构建这些知识需要经验和对于LightDB中索引工作方式的理解。在大部分情况下，一个部分索引相对于一个普通索引的优势很小。在某些情况下，它们会完全相反，例如Example 11.4

### Example 11.4. 不要使用部分索引代替分区

你可能想尝试创建一组巨大的、不重叠的部分索引，例如

```
CREATE INDEX mytable_cat_1 ON mytable (data) WHERE category = 1;
CREATE INDEX mytable_cat_2 ON mytable (data) WHERE category = 2;
CREATE INDEX mytable_cat_3 ON mytable (data) WHERE category = 3;
...
CREATE INDEX mytable_cat_N ON mytable (data) WHERE category = N;
```

这是个坏主意!几乎可以肯定，使用一个非部分索引会更好一些，声明如

```
CREATE INDEX mytable_cat_data ON mytable (category, data);
```

(将类别列放在前面，基于Section 11.3所述的原因。)虽然在这个更大的索引中进行搜索可能比在更小的索引中进行搜索要下降两倍以上，但这几乎肯定会比选择适当的部分索引中的一个所需的规划器的开销更便宜。问题的核心是系统不理解部分索引之间的关系，并将费力地测试每个索引，以确定它是否适用于当前查询。

如果你的表足够大，单个索引确实是一个坏主意，你应该考虑使用分区代替（参见Section 5.1）。通过这种机制，系统理解表和索引是不重叠的，就此而言可以获得更好的性能。

关于部分索引的更多信息可以在[ston89b]、[olson93]和[seshadri95]中找到。

## 11.9. 只用索引的扫描和覆盖索引

LightDB中的所有索引是二级索引，这意味着每个索引都是与表的主数据区（在LightDB术语称为表的堆中）分开存储。这意味着在普通索引扫描中，每行检索都需要从索引和堆中取数据。此外，虽然匹配给定的可索引WHERE条件的索引条目通常在一起靠近存储，但它们引用的表行可能在堆中的任何地方。因此索引扫描的堆访问部分涉及到对堆的大量随机访问，这可能很慢，特别是在传统旋转媒介上。如Section 11.5中所述，位图扫描尝试通过按排序的顺序进行堆访问来减少成本，但这远远不够）。

为了解决这种性能问题，LightDB支持只用索引的扫描，这类扫描可以仅用一个索引来回答查询而不产生任何堆访问。其基本思想是直接从每一个索引项中直接返回值，而不是去参考相关的堆项。在使用这种方法时有两个根本的限制：

1. 索引类型必须支持只用索引的扫描。B-树索引总是支持只用索引的扫描。GiST 索引只对某些操作符类支持只用索引的扫描。其他索引类型不支持这种扫描。底层的要求是索引必须在物理上存储或者可以重构出每一个索引项对应的原始数据值。GIN 索引是一个不支持只用索引的扫描的反例，因为它的每一个索引项通常只包含原始数据值的一部分。
2. 查询必须只引用存储在该索引中的列。例如，给定的索引建立在表的列x和y上，而该表还有一个列z，这些查询可以使用只用索引的扫描：

```
SELECT x, y FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND y < 42;
```

但是这些查询不能使用只用索引的查询：

```
SELECT x, z FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND z < 42;
```

（如下面所讨论的，表达式索引和部分索引会让这条规则更加复杂）。

如果符合这两个根本要求，那么该查询所要求的所有数据值都可以从索引得到，因此才可能使用只用索引的扫描。但是对LightDB中的任何表扫描还有一个额外的要求：必须验证每一个检索到的行对该查询的MVCC快照是“可见的”，如Chapter 1中讨论的那样。可见性信息并不存储在索引项中，只存储在堆项中。因此，乍一看似乎每一次行检索无论如何都会要求一次堆访问。如果表行最近被修改过，确实是这样。但是，对于很少更改的数据有一种方法可以解决这个问题。LightDB为表堆中的每一个页面跟踪是否其中所有的行的年龄都足够大，以至于对所有当前以及未来的事务都可见。这个信息存储在该表的可见性映射的一个位中。在找到一个候选索引项后，只用索引的扫描会检查对应堆页面的可见性映射位。如果该位被设置，那么这一行就是可见的并且该数据库可以直接被返回。如果该位没有被设置，那么就必须访问堆项以确定这一行是否可见，这种情况下相对于标准索引扫描就没有性能优势。即便是在成功的情况下，这种方法也是把对堆的访问换成了对可见性映射的访问。不过由于可见性映射比它所描述的堆要小四个数量级，所以访问可见性映射所需的物理I/O要少很多。在大部分情况下，可见性映射总是会被保留在内存中的缓冲中。

总之，虽然当两个根本要求满足时可以使用只用索引的扫描，但是只有该表的堆页面中有很大一部分的“所有都可见”映射位被设置时这种索引才有优势。不过，有很大一部分行不被更改的表是很常见的，这也让这一类扫描在实际中非常有用。

为了有效利用仅索引扫描功能，您可以选择创建一个覆盖索引，它是一个特别设计的索引，包含经常运行的特殊类型查询所需要的列。由于查询通常需要检索的列不仅仅是他们搜



索的列，LightDB允许您创建索引，这个索引中有些列只是“负荷”而不是搜索键的一部分。这可以通过添加INCLUDE来完成子句来列出了额外的列。例如，如果您通常可以运行这样的查询：

```
SELECT y FROM tab WHERE x = 'key' ;
```

加快此类查询的传统方法是仅在x上的索引。但是，一个索引定义为

```
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

可以将这些查询作为仅索引扫描处理，因为y可以从索引中获取而不需要访问堆。

因为列y不是搜索键的一部分，它不必是索引可以处理的数据类型；它只存储在索引中，不由索引机解释。另外，如果索引是唯一的索引，则

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

唯一性条件仅适用于x 列，而不是x和y的组合。（如果使用和在索引中设置的类似语法，一个INCLUDE子句可以写在UNIQUE和PRIMARY KEY约束中。）

保守地将非键负载列添加到索引是明智的，尤其是宽列。如果索引元组超过索引类型允许的最大大小，数据插入将失败。在任何情况下，非键列都将复制索引表中的数据并放大了索引的大小，从而有可能减慢搜索速度。请记住，除非一个表足够慢以至于仅索引扫描可能不必访问堆，否则没有什么理由在一个索引中包含负载列。无论如何，如果必须访问堆元组，从堆里获取列的值并不会带来更高的开销。其他限制是表达式不被作为包含的来支持。只有B树，GiST和SP-GiST 索引当前支持包含的列。

在 LightDB有INCLUDE特性之前，人们有时会通过写负载列作为普通索引列来制作覆盖索引。它这样写：

```
CREATE INDEX tab_x_y ON tab(x, y);
```

即使他们无意将y用作WHERE子句的一部分，只要额外的列是尾列就可以很好的工作。让它们成为前导字段是不明智的，原因在Section 11.8中有说明。但是，此方法不支持您希望索引在键列上实施唯一性。

Suffix truncation总是从B-Tree的上层移除非键列。作为有效负载列，它们从不用于指导索引扫描。当键列的其余前缀恰好足以描述最低B-Tree级别上的元组时，截断过程还会删除一个或多个尾随的键列。实际中，不使用INCLUDE子句覆盖索引通常会避免存储在上层有效负载的列。然而，显式地将有效负载列定义为非键列reliably使上层元组保持较小。

原则上，只用索引的扫描可以被用于表达式索引。例如，给定一个f(x)上的索引（x是一个表列），可以把

```
SELECT f(x) FROM tab WHERE f(x) < 1;
```

作为索引扫描的唯一扫描进行查询是非常有吸引力的，特别是当f()是一个计算开销很大的函数时。然而，LightDB的规划器目前对这种情况并不是很聪明。只有当查询所需的所有列都可以从索引中获取时，它才认为可以通过索引扫描进行查询。在这个例子中，除了在上下文f(x)中需要x之外，它不需要x，但规划器没有注意到这一点，并得出不能使用索引扫描的结论。如果索引扫描看起来非常值得，可以通过将x作为包含列添加来解决这个问题，例如：

```
CREATE INDEX tab_f_x ON tab (f(x)) INCLUDE (x);
```

另一个需要注意的问题是，如果目标是避免重新计算 $f(x)$ ，那么规划器不一定会将不在可索引的WHERE子句中使用 $f(x)$ 与索引列匹配。在上面显示的简单查询中，它通常会得出正确的结果，但在涉及连接的查询中不会。这些不足可能在未来版本的LightDB中得到解决。

部分索引也和只用索引的扫描之间有着有趣的关系。考虑Example 11.8所展示的部分索引：

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
WHERE success;
```

原则上，我们可以在这个索引上使用只用索引的扫描来满足查询

```
SELECT target FROM tests WHERE subject = 'some-subject' AND success;
```

但是有一个问题：WHERE子句引用的是不能作为索引结果列的success。尽管如此，还是可以使用只用索引的扫描，因为在运行时计划不需要重新检查WHERE子句的那个部分：在该索引中找到的所有项必定具有`success = true`，因此在计划中检查这个部分的需要并不明显。LightDB 9.6 和以后的版本将会识别这种情况，并且允许生成只用索引的扫描，但是旧版本无法这样做。

## 11.10. 操作符类和操作符族

一个索引定义可以为索引中的每一列都指定一个操作符类。

```
CREATE INDEX name ON table (column opclass [ ( opclass_options ) ] [sort
options] [, ...]);
```

操作符类标识该列上索引要使用的操作符。例如，一个int4类型上的B树索引会使用int4\_ops类，这个操作符类包括用于int4类型值的比较函数。实际上列的数据类型的默认操作符类通常就足够了。存在多个操作符类的原因是，对于某些数据类型可能会有多于一种的有意义的索引行为。例如，我们可能想要对一种复数数据类型按照绝对值排序或者按照实数部分排序。我们可以通过为该数据类型定义两个操作符类来实现，并且在创建一个索引时选择合适的类。操作符类会决定基本的排序顺序（可以通过增加排序选项COLLATE、ASC/DESC和/或 NULLS FIRST/NULLS LAST来修改）。

除了默认的操作符类，还有一些内建的操作符类：

- 操作符类text\_pattern\_ops、varchar\_pattern\_ops和bpchar\_pattern\_ops分别支持类型text、varchar和char上的B树索引。它们与默认操作符类的区别是值的比较是严格按照字符进行而不是根据区域相关的排序规则。这使得这些操作符类适合于当一个数据库没有使用标准“C”区域时被使用在涉及模式匹配表达式（LIKE或POSIX正则表达式）的查询中。一个例子是，你可以这样索引一个varchar列：

```
CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
```

注意如果你希望涉及到 $<$ 、 $<=$ 、 $>$ 或 $>=$ 比较的查询使用一个索引，你也应该创建一个使用默认操作符类的索引。这些查询不能使用xxx\_pattern\_ops操作符类（但是普通的等值比较可以使用这些操作符类）。可以在同一个列上创建多个使用不同操作符类的索引。如果你正在使用C区域，你并不需要xxx\_pattern\_ops操作符类，因为在C区域中的模式匹配查询可以用带有默认操作符类的索引。

下面的查询展示了所有已定义的操作符类：

```
SELECT am.amname AS index_method,
```

```

opc.opcname AS opclass_name,
opc.opcintype::regtype AS indexed_type,
opc.opcdefault AS is_default
FROM pg_am am, pg_opclass opc
WHERE opc.opcmethod = am.oid
ORDER BY index_method, opclass_name;

```

一个操作符类实际上只是一个更大的被称为操作符族的结构的一个子集。在多种数据类型具有相似行为的情况下，常常会定义跨数据类型的操作符并且允许索引使用它们。为了实现该目的，这些类型的操作符类必须被分组到同一个操作符族中。跨类型的操作符是该族的成员，但是并不与族内任意一个单独的类相关联。

前一个查询的扩展版本展示了每个操作符类所属的操作符族：

```

SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opf.opfname AS opfamily_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
FROM pg_am am, pg_opclass opc, pg_opfamily opf
WHERE opc.opcmethod = am.oid AND
       opc.opcfamily = opf.oid
ORDER BY index_method, opclass_name;

```

这个查询展示所有已定义的操作符族和每一个族中包含的所有操作符：

```

SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
FROM pg_am am, pg_opfamily opf, pg_amop amop
WHERE opf.opfmethod = am.oid AND
       amop.amopfamily = opf.oid
ORDER BY index_method, opfamily_name, opfamily_operator;

```

### Tip

ltsql has commands \dAc, \dAf, and \dAo, which provide slightly more sophisticated versions of these queries.

## 11.11. 索引和排序规则

一个索引在每一个索引列上只能支持一种排序规则。如果需要多种排序规则，你可能需要多个索引。

考虑这些语句：

```

CREATE TABLE testlc (
  id integer,
  content varchar COLLATE "x"
);

```

```

CREATE INDEX testlc_content_index ON testlc (content);

```

该索引自动使用下层列的排序规则。因此一个如下形式的查询：

```
SELECT * FROM testlc WHERE content > constant;
```

可以使用该索引，因为比较会默认使用列的排序规则。但是，这个索引无法加速涉及到某些其他排序规则的查询。因此对于下面形式的查询：

```
SELECT * FROM testlc WHERE content > constant COLLATE "y";
```

可以创建一个额外的支持“y”排序规则的索引，例如：

```
CREATE INDEX testlc_content_y_index ON testlc (content COLLATE "y");
```

## 11.12. 检查索引使用

尽管LightDB中的索引并不需要维护或调优，但是检查真实的查询负载实际使用了哪些索引仍然非常重要。检查一个独立查询的索引使用情况可以使用EXPLAIN命令，它应用于这种目的的内容在Section 14.4中有介绍。也可以在一个运行中的服务器上收集有关索引使用的总体统计情况，如Section 25.4所述。

很难明确地表达决定创建哪些索引的通用过程。在之前的小节中的例子里有一些典型的情况。通常需要大量的实验才能决定应该创建哪些索引。本小节剩余的部分将给出一些创建索引的提示：

- 总是先运行ANALYZE。这个命令会收集有关表中值分布情况的统计信息。估计一个查询将要返回的行数需要这些信息，而结果行数则被规划器用来为每一个可能的查询计划分配实际的代价。如果没有任何真实的统计信息，将会假定一些默认值，这几乎肯定是不准确的。在没有运行的情况下检查一个应用的索引使用情况是注定要失败的。详见Section 22.1.4和Section 22.1.6

- 使用真实数据进行实验。使用测试数据来建立索引将会告诉你测试数据需要什么样的索引，但这并不代表真实数据的需要。

使用非常小的测试数据集是特别致命的。在从100000行中选出1000行时可能会用到索引，但是从100行里选出1行是很难用到索引的，因为100行完全可能放入到一个磁盘页面中，而没有任何计划能够比得上从一个磁盘页面顺序获取的计划。

在创建测试数据时也要小心，特别是当应用还没有产生时通常是不可避免的。值非常相似、完全随机或以排好序的方式被插入都将使得统计信息倾斜于真实数据中的值分布。

- 如果索引没有被用到，强制使用它们将会对测试非常有用。有一些运行时参数可以关闭多种计划类型（参见Section 17.7.1）。例如，关闭顺序扫描（enable\_seqscan）以及嵌套循环连接（enable\_nestloop）将强制系统使用一种不同的计划。如果系统仍然选择一个顺序扫描或嵌套循环连接，则索引没有被使用的原因可能更加根本，例如查询条件不匹配索引（哪种查询能够使用哪种索引已经在前面的小节中解释过了）。
- 如果强制索引使用确实使用了索引，则有两种可能性：系统是正确并且索引确实不合适，或者查询计划的代价估计并没有反映真实情况。因此你应该对用索引的查询和不用索引的查询计时。此时EXPLAIN ANALYZE命令就能发挥作用了。
- 如果发现代价估计是错误的，也分为两种可能性。总代价是用每个计划节点的每行代价乘以计划节点的选择度估计来计算的。计划节点的代价估计可以通过运行时参数调整（如Section 17.7.1所述）。不准确的选择度估计可能是由于缺乏统计信息，可以通过调节统计信息收集参数（见ALTER TABLE）来改进。

如果你不能成功地把代价调整得更合适，那么你可能必须依靠显式地强制索引使用。你也可能希望联系LightDB开发者来检查该问题。

---

# Chapter 12. 全文搜索

## 12.1. 介绍

全文搜索（或者文本搜索）提供了确定满足一个查询的自然语言文档的能力，并可以选择将它们按照与查询的相关度排序。最常用的搜索类型是找到所有包含给定查询词的文档并按照它们与查询的相似性顺序返回它们。查询和相似性的概念非常灵活并且依赖于特定的应用。最简单的搜索认为查询是一组词而相似性是查询词在文档中的频度。

文本搜索操作符已经在数据库中存在很多年了。LightDB对文本数据类型提供了~、~\*、LIKE和ILIKE操作符，但是它们缺少现代信息系统所要求的很多基本属性：

- 即使对英语也缺乏语言的支持。正则表达式是不够的，因为它们不能很容易地处理派生词，例如satisfies和satisfy。你可能会错过包含satisfies的文档，尽管你可能想要在对于satisfy的搜索中找到它们。可以使用OR来搜索多个派生形式，但是这样做太啰嗦也容易出错（有些词可能有数千种派生）。
- 它们不提供对搜索结果的排序（排名），这使它们面对数以千计被找到的文档时变得无效。
- 它们很慢因为没有索引支持，因此它们必须为每次搜索处理所有的文档。

全文索引允许文档被预处理并且保存一个索引用于以后快速的搜索。预处理包括：

将文档解析成记号。标识出多种类型的记号是有所帮助的，例如数字、词、复杂的词、电子邮件地址，这样它们可以被以不同的方式处理。原则上记号分类取决于相关的应用，但是对于大部分目的都可以使用一套预定义的分类。LightDB使用一个解析器来执行这个步骤。其中提供了一个标准的解析器，并且为特定的需要也可以创建定制的解析器。

将记号转换成词位。和一个记号一样，一个词位是一个字符串，但是它已经被正规化，这样同一个词的不同形式被变成一样。例如，正规化几乎总是包括将大写字母转换成小写形式，并且经常涉及移除后缀（例如英语中的s或es）。这允许搜索找到同一个词的变体形式，而不需要冗长地输入所有可能的变体。此外，这个步骤通常会消除停用词，它们是那些太普通的词，它们对于搜索是无用的（简而言之，记号是文档文本的原始片段，而词位是那些被认为对索引和搜索有用的词）。LightDB使用词典来执行这个步骤。已经提供了多种标准词典，并且为特定的需要也可以创建定制的词典。

为搜索优化存储预处理好的文档。例如，每一个文档可以被表示为正规化的词位的一个有序数组。与词位一起，通常还想要存储用于近似排名的位置信息，这样一个包含查询词更“密集”区域的文档要比那些包含分散的查询词的文档有更高的排名。

词典允许对记号如何被正规化进行细粒度的控制。使用合适的词典，你可以：

- 定义不应该被索引的停用词。
- 使用Ispell把同义词映射到一个单一词。
- 使用一个分类词典把短语映射到一个单一词。
- 使用一个Ispell词典把一个词的不同变体映射到一种规范的形式。
- 使用Snowball词干分析器规则将一个词的不同变体映射到一种规范的形式。

我们提供了一种数据类型tsvector来存储预处理后的文档，还提供了一种类型tsquery来表示处理过的查询（Section 8.11）。有很多函数和操作符可以用于这些数据类型（Section 9.13，其中最重要的是匹配操作符@@，它在Section 12.1.2中介绍。全文搜索可以使用索引来加速（Section 12.9）。

### 12.1.1. 什么是一个文档？

一个document是在一个全文搜索系统中进行搜索的单元，例如，一篇杂志文章或电子邮件消息。文本搜索引擎必须能够解析文档并存储词位（关键词）与它们的父文档之间的关联。随后，这些关联会被用来搜索包含查询词的文档。

对于LightDB中的搜索，一个文档通常是一个数据库表中一行内的一个文本形式的域，或者可能是这类域的一个组合（连接），这些域可能存储在多个表或者是动态获取。换句话说，一个文档可能从用于索引的不同部分构建，并且它可能被作为一个整体存储在某个地方。例如：

```
SELECT title || ' ' || author || ' ' || abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;
```

```
SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS
document
FROM messages m, docs d
WHERE m.mid = d.did AND m.mid = 12;
```

### Note

实际上在这些例子查询中，`coalesce`应该被用来防止一个单一NULL属性导致整个文档的一个NULL结果。

另一种存储文档的可能性是作为文件系统中的简单文本文件。在这种情况下，数据库可以被用来存储全文索引并执行搜索，并且某些唯一标识符可以被用来从文件系统检索文档。但是，从数据库的外面检索文件要求超级用户权限或者特殊函数支持，因此这种方法通常不如把所有数据放在LightDB内部方便。另外，把所有东西放在数据库内部允许方便地访问文档元数据来协助索引和现实。

对于文本搜索目的，每一个文档必须被缩减成预处理后的`tsvector`格式。搜索和排名被整个在一个文档的`tsvector`表示上执行——只有当文档被选择来显示给用户时才需要检索原始文本。我们因此经常把`tsvector`说成是文档，但是当然它只是完整文档的一种紧凑表示。

## 12.1.2. 基本文本匹配

LightDB中的全文搜索基于匹配操作符`@@`，它在一个`tsvector`（文档）匹配一个`tsquery`（查询）时返回`true`。哪种数据类型写在前面没有影响：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat &
rat'::tsquery;
?column?
```

t

```
SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat
rat'::tsvector;
?column?
```

f

正如以上例子所建议的，一个`tsquery`并不只是一个未经处理的文本，顶多一个`tsvector`是这样。一个`tsquery`包含搜索术语，它们必须是已经正规化的词位，并且可以使用`AND`、`OR`、`NOT`以及`FOLLOWED BY`操作符结合多个术语（语法详见Section 8.11.2）。有几个函数`to_tsquery`、`plainto_tsquery`以及`phraseto_tsquery`可用于将用户书写的文本转换为正确的`tsquery`，它们会主要采用正则化出现在文本中的词的方法。相似地，`to_tsvector`被用来解析和正规化一个文档字符串。因此在实际上一个文本搜索匹配可能看起来更像：

```
SELECT to_tsvector(' fat cats ate fat rats') @@ to_tsquery(' fat & rat');
?column?
```

```
-----
t
```

注意如果这个匹配被写成下面这样它将不会成功:

```
SELECT ' fat cats ate fat rats'::tsvector @@ to_tsquery(' fat & rat');
?column?
```

```
-----
f
```

因为这里不会发生词rats的正规化。一个tsvector的元素是词位，它被假定为已经正规化好，因此rats不匹配rat。

@@操作符也支持text输出，它允许在简单情况下跳过从文本字符串到tsvector或tsquery的显式转换。可用的变体是:

```
tsvector @@ tsquery
tsquery  @@ tsvector
text     @@ tsquery
text     @@ text
```

前两种我们已经见过。形式text @@ tsquery等价于to\_tsvector(x) @@ y。形式text @@ text等价于to\_tsvector(x) @@ plainto\_tsquery(y)。

在tsquery中，& (AND) 操作符指定它的两个参数都必须出现在文档中才表示匹配。类似地，| (OR) 操作符指定至少一个参数必须出现，而! (NOT) 操作符指定它的参数不出现才能匹配。例如，查询fat & ! rat匹配包含fat但不包含rat的文档。

在<-> (FOLLOWED BY) tsquery操作符的帮助下搜索可能的短语，只有该操作符的参数的匹配是相邻的并且符合给定顺序时，该操作符才算是匹配。例如:

```
SELECT to_tsvector(' fatal error') @@ to_tsquery(' fatal <-> error');
?column?
```

```
-----
t
```

```
SELECT to_tsvector(' error is not fatal') @@ to_tsquery(' fatal <-> error');
?column?
```

```
-----
f
```

FOLLOWED BY 操作符还有一种更一般的版本，形式是<N>，其中N是一个表示匹配词位位置之间的差。<1>和<->相同，而<2>允许刚好一个其他词位出现在匹配之间，以此类推。当有些词是停用词时，phraseto\_tsquery函数利用这个操作符来构造一个能够匹配多词短语的tsquery。例如:

```
SELECT phraseto_tsquery(' cats ate rats');
phraseto_tsquery
```

```
-----
'cat' <-> 'ate' <-> 'rat'
```

```
SELECT phraseto_tsquery(' the cats ate the rats');
phraseto_tsquery
```

'cat' <-> 'ate' <2> 'rat'

一种有时候有用的特殊情况是，<0>可以被用来要求两个匹配同一个词的模式。

圆括号可以被用来控制tsquery操作符的嵌套。如果没有圆括号，|的计算优先级最低，然后从低到高依次是&、<->、!。

值得注意的是，当AND/OR/NOT操作符在一个FOLLOWED BY操作符的参数中时，它们表示与不在那些参数中时不同的含义，因为在FOLLOWED BY中匹配的准确位置是有意义的。例如，通常!x仅匹配在任何地方都不包含x的文档。但如果y不是紧接在一个x后面，!x <-> y就会匹配那个y，在文档中其他位置出现的x不会阻止匹配。另一个例子是，x & y通常仅要求x和y均出现在文档中的某处，但是(x & y) <-> z要求x和y在紧挨着z之前的同一个位置匹配。因此这个查询的行为会不同于x <-> z & y <-> z，它将匹配一个含有两个单独序列x z以及y z的文档（这个特定的查询一点用都没有，因为x和y不可能在同一个位置匹配，但是对于前缀匹配模式之类的更复杂的情况，这种形式的查询就会有有用武之地）。

### 12.1.3. 配置

前述的都是简单的文本搜索例子。正如前面所提到的，全文搜索功能包括做更多事情的能力：跳过索引特定词（停用词）、处理同义词并使用更高级的解析，例如基于空白之外的解析。这个功能由文本搜索配置控制。LightDB中有多种语言的预定义配置，并且你可以很容易地创建你自己的配置（ltsql的\dF命令显示所有可用的配置）。

在安装期间一个合适的配置将被选择并且default\_text\_search\_config也被相应地设置在lightdb.conf中。如果你正在对整个集簇使用相同的文本搜索配置，你可以使用在lightdb.conf中使用该值。要在集簇中使用不同的配置但是在任何一个数据库内部使用同一种配置，使用ALTER DATABASE ... SET。否则，你可以在每个会话中设置default\_text\_search\_config。

依赖一个配置的每一个文本搜索函数都有一个可选的regconfig参数，因此要使用的配置可以被显式指定。只有当这个参数被忽略时，default\_text\_search\_config才被使用。

我们为文本搜索解析器提供了两个配置。它们在lightdb.conf中相应地设置。您可以在会话期间修改此值：

- lightdb\_tsearch\_non\_stopwords 用于自定义非停用词（non-stop words），它与停用词（stop words）相反。
- lightdb\_tsearch\_word\_superpose 用于叠加使用停用词和非停用词的效果。该配置不支持中文文本解析。

为了让建立自定义文本搜索配置更容易，一个配置可以从更简单的数据库对象来建立。LightDB的文本搜索功能提供了四类配置相关的数据库对象：

- 文本搜索解析器将文档拆分成记号并分类每个记号（例如，作为词或者数字）。
- 文本搜索词典将记号转变成正规化的形式并拒绝停用词。
- 文本搜索模板提供位于词典底层的函数（一个词典简单地指定一个模板和一组用于模板的参数）。
- 文本搜索配置选择一个解析器和一组用于将解析器产生的记号正规化的词典。

文本搜索解析器和模板是从低层 C 函数构建而来，因此它要求 C 编程能力来开发新的解析器和模板，并且还需要超级用户权限来把它们安装到一个数据库中（在LightDB发布的contrib/区域中有一些附加的解析器和模板的例子）。由于词典和配置只是对底层解析器和模板的参数化和连接，不需要特殊的权限来创建一个新词典或配置。创建定制词典和配置的例子将在本章稍后的部分给出。

## 12.2. 表和索引

在前一节中的例子演示了使用简单常数字符串进行全文匹配。本节展示如何搜索表数据，以及可选择地使用索引。



## 12.2.1. 搜索一个表

可以在没有一个索引的情况下做一次全文搜索。一个简单的查询将打印每一个行的title, 这些行在其body域中包含词friend:

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

这还将还会找到相关的词例如friends和friendly, 因为这些都约减到同一个正规化的词位。

以上的查询指定要使用english配置来解析和正规化字符串。我们也可以忽略配置参数:

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

这个查询将使用由default\_text\_search\_config设置的配置。

一个更复杂的例子是选择 10 个最近的文档, 要求它们在title或body中包含create和table:

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

为了清晰, 我们忽略coalesce函数调用, 它可能需要被用来查找在这两个域之中包含NULL的行。

尽管这些查询可以在没有索引的情况下工作, 大部分应用会发现这种方法太慢了, 除了偶尔的临时搜索。实际使用文本搜索通常要求创建一个索引。

## 12.2.2. 创建索引

我们可以创建一个GIN索引 (Section 12.9 来加速文本搜索):

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', body));
```

注意这里使用了to\_tsvector的双参数版本。只有指定了一个配置名称的文本搜索函数可以被用在表达式索引 (Section 11.7 中。这是因为索引内容必须是没有被default\_text\_search\_config影响的。如果它们被影响, 索引内容可能会不一致因为不同的项可能包含被使用不同文本搜索配置创建的tsvector, 并且没有办法猜测哪个是哪个。也没有可能正确地转储和恢复这样的一个索引。

由于to\_tsvector的双参数版本被使用在上述的索引中, 只有一个使用了带有相同配置名的双参数版to\_tsvector的查询引用才能使用该索引。即, WHERE to\_tsvector('english', body) @@ 'a & b' 可以使用该索引, 但WHERE to\_tsvector(body) @@ 'a & b' 不能。这保证一个索引只能和创建索引项时所用的相同配置一起使用。

可以建立更复杂的表达式索引, 在其中配置名被另一个列指定, 例如:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector(config_name, body));
```

这里`config_name`是`pgweb`表中的一个列。这允许在同一个索引中有混合配置，同时记录哪个配置被用于每一个索引项。例如，如果文档集合包含不同语言的文档，这就可能会有用。同样，要使用索引的查询必须被措辞成匹配，例如`WHERE to_tsvector(config_name, body) @@ 'a & b'`。

索引甚至可以连接列：

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', title || ' '
|| body));
```

另一种方法是创建一个单独的`tsvector`列来保存`to_tsvector`的输出。若要使此列与其源数据保持自动更新，用存储生成的列。这个例子是`title`和`body`的连接，使用`coalesce`来保证当其他域为`NULL`时一个域仍然能留在索引中：

```
ALTER TABLE pgweb
  ADD COLUMN textsearchable_index_col tsvector
  GENERATED ALWAYS AS (to_tsvector('english', coalesce(title, ''))
  || ' ' || coalesce(body, '')) STORED;
```

然后我们创建一个GIN索引来加速搜索：

```
CREATE INDEX textsearch_idx ON pgweb USING GIN (textsearchable_index_col);
```

现在我们准备好执行一个快速的全文搜索了：

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

单独列方法相对于表达式索引的一个优势在于，它不必为了利用索引而在查询中显式地指定文本搜索配置。如上述例子所示，查询可以依赖`default_text_search_config`。另一个优势是搜索将会更快，因为它不必重做`to_tsvector`调用来验证索引匹配（在使用GiST索引时这一点比使用GIN索引时更重要；见Section 12.9。表达式索引方法更容易建立，但是它要求更少的磁盘空间，因为`tsvector`表示没有被显式地存储下来。

## 12.3. 控制文本搜索

要实现全文搜索必须要有一个从文档创建`tsvector`以及从用户查询创建`tsquery`的函数。而且我们需要一种有用的顺序返回结果，因此我们需要一个函数能够根据文档与查询的相关性比较文档。还有一点重要的是要能够很好地显示结果。LightDB对所有这些函数都提供了支持。

### 12.3.1. 解析文档

LightDB提供了函数`to_tsvector`将一个文档转换成`tsvector`数据类型。

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

`to_tsvector`把一个文本文档解析成记号，把记号缩减成词位，并且返回一个`tsvector`，它列出了词位以及词位在文档中的位置。文档被根据指定的或默认的文本搜索配置来处理。下面是一个简单例子：

```
SELECT to_tsvector('english', 'a fat cat sat on a mat - it ate a fat rats');
       to_tsvector
```

```
-----
'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

在上面这个例子中我们看到，作为结果的tsvector不包含词a、on或it，词rats变成了rat，并且标点符号-被忽略了。

to\_tsvector函数在内部调用了—个解析器，它把文档文本分解成记号并且为每一种记号分配一个类型。对于每一个记号，会去查询一个词典列表（Section 12.6，该列表会根据记号的类型而变化。第一个识别记号的词典产生一个或多个正规化的词位来表示该记号。例如，rats变成rat是因为一个词典识别到该词rats是rat的复数形式。一些词会被识别为停用词（Section 12.6.1），这将导致它们被忽略，因为它们出现得太频繁以至于在搜索中起不到作用。在我们的例子中有a、on和it是停用词。如果在列表中没有词典能识别该记号，那它也将被忽略。在这个例子中标点符号-就属于这种情况，因为事实上没有词典会给它分配记号类型（空间符号），即空间记号不会被索引。对于解析器、词典以及要素索引哪些记号类型是由所选择的文本搜索配置（Section 12.7决定的。可以在同一个数据库中有多种不同的配置，并且有用于很多种语言的预定义配置。在我们的例子中，我们使用用于英语的默认配置english。

函数setweight可以被用来对tsvector中的项标注一个给定的权重，这里一个权重可以是四个字母之一：A、B、C或D。这通常被用来标记来自文档不同部分的项，例如标题对正文。稍后，这种信息可以被用来排名搜索结果。

因为to\_tsvector(NULL)将返回NULL，不论何时一个域可能为空时，我们推荐使用coalesce。下面是我们推荐的从一个结构化文档创建一个tsvector的方法：

```
UPDATE tt SET ti =
  setweight(to_tsvector(coalesce(title,'')), 'A') ||
  setweight(to_tsvector(coalesce(keyword,'')), 'B') ||
  setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
  setweight(to_tsvector(coalesce(body,'')), 'D');
```

这里我们已经使用了setweight在完成的tsvector标注每一个词位的来源，并且接着将标注过的tsvector值用tsvector连接操作符||合并在一起（Section 12.4.给出了关于这些操作符的细节）。

## 12.3.2. 解析查询

LightDB提供了函数to\_tsquery、plainto\_tsquery、phraseto\_tsquery以及websearch\_to\_tsquery用来把一个查询转换成tsquery数据类型。to\_tsquery提供了比plainto\_tsquery和phraseto\_tsquery更多的特性，但是它对其输入要求更加严格。websearch\_to\_tsquery是to\_tsquery的一个简化版本，它使用一种可选择的语法，类似于Web搜索引擎使用的语法。

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

to\_tsquery从querytext创建一个tsquery值，该值由被tsquery操作符&（AND）、|（OR）、!（NOT）和<->（FOLLOWED BY）分隔的单个记号组成。这些操作符可以使用圆括号分组。换句话说，to\_tsquery的输入必须已经遵循tsquery输入的一般规则，如Section 8.11.所述。区别在于基本的tsquery输入把记号当作表面值，而to\_tsquery会使用指定的或者默认的配置把每一个记号正规化成一个词位，并且丢弃掉任何根据配置是停用词的记号。例如：

```
SELECT to_tsquery('english', 'The & Fat & Rats');
       to_tsquery
-----
'fat' & 'rat'
```

和在基本tsquery输入中一样，权重可以被附加到每一个词位来限制它只匹配属于那些权重的tsvector词位。例如：

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
       to_tsquery
-----
'fat' | 'rat':AB
```

同样，\*可以被附加到一个词位来指定前缀匹配：

```
SELECT to_tsquery('supern:*A & star:A*B');
       to_tsquery
-----
'supern':*A & 'star':*AB
```

这样一个词位将匹配一个tsvector中的任意以给定字符串开头的词。

to\_tsquery也能够接受单引号短语。当配置包括一个会在这种短语上触发的分类词典时就是它的主要用处。在下面的例子中，一个分类词典含规则supernovae stars : sn:

```
SELECT to_tsquery(''supernovae stars' & !crab');
       to_tsquery
-----
'sn' & !'crab'
```

在没有引号时，to\_tsquery将为那些没有被 AND、OR 或者 FOLLOWED BY 操作符分隔的记号产生一个语法错误。

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

plainto\_tsquery将未格式化的文本querytext转换成一个tsquery值。该文本被解析并被正规化，很像to\_tsvector，然后& (AND) 布尔操作符被插入到留下来的词之间。

例子：

```
SELECT plainto_tsquery('english', 'The Fat Rats');
       plainto_tsquery
-----
'fat' & 'rat'
```

注意plainto\_tsquery不会识其输入中的tsquery操作符、权重标签或前缀匹配标签：

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
       plainto_tsquery
-----
'fat' & 'rat' & 'c'
```

这里，所有输入的标点都被丢弃。

phraseto\_tsquery([ config regconfig, ] querytext text) returns tsquery

phraseto\_tsquery的行为很像plainto\_tsquery，不过前者会在留下来的词之间插入<-> (FOLLOWED BY) 操作符而不是& (AND) 操作符。还有，停用词也不是简单地丢弃掉，而是通过插入<N>操作符（而不是<->操作符）来解释。在搜索准确的词位序列时这个函数很有用，因为 FOLLOWED BY 操作符不只是检查所有词位的存在性，还会检查词位的顺序。

例子：

```
SELECT phraseto_tsquery('english', 'The Fat Rats');
      phraseto_tsquery
```

```
-----
'fat' <-> 'rat'
```

和plainto\_tsquery相似，phraseto\_tsquery函数不会识别其输入中的tsquery操作符、权重标签或者前缀匹配标签：

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
      phraseto_tsquery
```

```
-----
'fat' <-> 'rat' <-> 'c'
```

websearch\_to\_tsquery([ config regconfig, ] querytext text) returns tsquery

websearch\_to\_tsquery使用一种可供选择的语法从querytext创建一个tsquery值，这种语法中简单的未格式化文本是一个有效的查询。和plainto\_tsquery以及phraseto\_tsquery不同，它还识别特定的操作符。此外，这个函数绝不会报出语法错误，这就可以把原始的用户提供的输入用于搜索。支持下列语法：

- 无引号文本：不在引号中的文本将被转换成由&操作符分隔的词，就像被plainto\_tsquery处理过那样。
- “引号文本”：在引号中的文本将被转换成由<->操作符分隔的词，就像被phraseto\_tsquery处理过那样。
- OR: “or” 将转换为|运算符。
- -: 破折号将转换为!运算符。

忽略其他标点符号。因此，与 plainto\_tsquery 和 phraseto\_tsquery 一样，websearch\_to\_tsquery函数在其输入中将不会识别tsquery运算符、权重标签或前缀匹配标签。

示例：

```
SELECT websearch_to_tsquery('english', 'The fat rats');
      websearch_to_tsquery
```

```
-----
'fat' & 'rat'
(1 row)
```

```
SELECT websearch_to_tsquery('english', '"supernovae stars" -crab');
      websearch_to_tsquery
```

```
-----
'supernova' <-> 'star' & '!crab'
(1 row)
```

```

SELECT websearch_to_tsquery('english', '"sad cat" or "fat rat"');
      websearch_to_tsquery
-----
' sad' <-> ' cat' | ' fat' <-> ' rat'
(1 row)

SELECT websearch_to_tsquery('english', 'signal -"segmentation fault"');
      websearch_to_tsquery
-----
' signal' & !( ' segment' <-> ' fault' )
(1 row)

SELECT websearch_to_tsquery('english', '""')( dummy \\ query <->');
      websearch_to_tsquery
-----
' dummi' & ' queri'
(1 row)

```

### 12.3.3. 排名搜索结果

排名处理尝试度量文档和一个特定查询的接近程度，这样当有很多匹配时最相关的那些可以被先显示。LightDB提供了两种预定义的排名函数，它们考虑词法、临近性和结构信息；即，它们考虑查询词在文档中出现得有多频繁，文档中的词有多接近，以及词出现的文档部分有多重要。不过，相关性的概念是模糊的并且与应用非常相关。不同的应用可能要求额外的信息用于排名，例如，文档修改时间。内建的排名函数只是例子。你可以编写你自己的排名函数和/或把它们的结果与附加因素整合在一起以适应你的特定需求。

目前可用的两种排名函数是：

```
ts_rank([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4
```

基于向量的匹配词位的频率来排名向量。

```
ts_rank_cd([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4
```

这个函数为给定文档向量和查询计算覆盖密度排名，该方法在 Clarke、Cormack 和 Tudhope 于 1999 年在期刊 “Information Processing and Management” 上的文章 “Relevance Ranking for One to Three Term Queries” 文章中有描述。覆盖密度类似于 `ts_rank` 排名，不过它会考虑匹配词位相互之间的接近度。

这个函数要求词位的位置信息来执行其计算。因此它会忽略 `tsvector` 中任何 “被剥离的” 词位。如果在输入中有未被剥离的词位，结果将会是零（`strip` 函数和 `tsvector` 中的位置信息的更多内容请见 Section 12.4.1）。

对这两个函数，可选的权重参数提供了为词实例赋予更多或更少权重的能力，这种能力是依据它们被标注的情况的。权重数组指定每一类词应该得到多重的权重，按照如下的顺序：

```
{D-权重, C-权重, B-权重, A-权重}
```

如果没有提供权重，那么将使用这些默认值：

```
{0.1, 0.2, 0.4, 1.0}
```

通常权重被用来标记来自文档特别区域的词，如标题或一个初始的摘要，这样它们可以被认为比来自文档正文的词更重要或更不重要。

由于一个较长的文档有更多的机会包含一个查询术语，因此考虑文档的尺寸是合理的，例如一个一百个词的文档中有一个搜索词的五个实例而零一个一千个词的文档中有该搜索词的五个实例，则前者比后者更相关。两种排名函数都采用一个整数正规化选项，它指定文档长度是否影响其排名以及如何影响。该整数选项控制多个行为，因此它是一个位掩码：你可以使用|指定一个或多个行为（例如，2|4）。

- 0（默认值）忽略文档长度
- 1 用  $1 + \text{文档长度}$  的对数除排名
- 2 用文档长度除排名
- 4 用长度之间的平均调和距离除排名（只被ts\_rank\_cd实现）
- 8 用文档中唯一词的数量除排名
- 16 用  $1 + \text{文档中唯一词数量}$  的对数除排名
- 32 用排名 + 1 除排名

如果多于一个标志位被指定，转换将根据列出的顺序被应用。

值得注意的是排名函数并不使用任何全局信息，因此它不可能按照某些时候期望地产生一个公平的正规化，从 1% 或 100%。正规化选项 32 ( $\text{rank}/(\text{rank}+1)$ ) 可以被应用来缩放所有的排名到范围零到一，但是当然这只是一个外观上的改变；它不会影响搜索结果的顺序。

这里是一个例子，它只选择十个最高排名的匹配：

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
```

title	rank
Neutrinos in the Sun	3.1
The Sudbury Neutrino Detector	2.4
A MACHO View of Galactic Dark Matter	2.01317
Hot Gas and Dark Matter	1.91171
The Virgo Cluster: Hot Plasma and Dark Matter	1.90953
Rafting for Solar Neutrinos	1.9
NGC 4650A: Strange Galaxy and Dark Matter	1.85774
Hot Gas and Dark Matter	1.6123
Ice Fishing for Cosmic Neutrinos	1.6
Weak Lensing Distorts the Universe	0.818218

这是相同的例子使用正规化的排名：

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
```

title	rank
Neutrinos in the Sun	0.756097569485493
The Sudbury Neutrino Detector	0.705882361190954
A MACHO View of Galactic Dark Matter	0.668123210574724
Hot Gas and Dark Matter	0.65655958650282
The Virgo Cluster: Hot Plasma and Dark Matter	0.656301290640973
Rafting for Solar Neutrinos	0.655172410958162
NGC 4650A: Strange Galaxy and Dark Matter	0.650072921219637
Hot Gas and Dark Matter	0.617195790024749

Ice Fishing for Cosmic Neutrinos	0.615384618911517
Weak Lensing Distorts the Universe	0.450010798361481

排名可能会非常昂贵，因为它要求查询每一个匹配文档的tsvector，这可能会涉及很多I/O因而很慢。不幸的是，这几乎不可避免，因为实际查询常常导致巨大数目的匹配。

## 12.3.4. 加亮结果

要表示搜索结果，理想的方式是显示每一个文档的一个部分并且显示它是怎样与查询相关的。通常，搜索引擎显示文档片段时会对其中的搜索术语进行标记。LightDB提供了一个函数ts\_headline来实现这个功能。

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text
]) returns text
```

ts\_headline接受一个文档和一个查询，并且从该文档返回一个引用，在其中来自查询的术语会被加亮。被用来解析该文档的配置可以用config指定；如果config被忽略，将会使用default\_text\_search\_config配置。

如果一个options字符串被指定，它必须由一个逗号分隔的列表组成，列表中是一个或多个option=value对。可用的选项是：

- MaxWords、MinWords（整数）：这些数字决定了要输出的最长和最短标题。默认值为 35 和 15。
- ShortWord（整数）：此长度或更短的单词将被删除在标题的开头和结尾，除非它们是查询词。默认值为3将删除常见的英语冠词。
- HighlightAll（布尔值）：如果 true 将整个文档用作标题，忽略前面三个参数。默认值为 false。
- MaxFragments（整数）：要显示的最大文本片段数。默认值为零选择非基于片段的标题生成方法。大于零的值选择基于片段的标题生成（见下文）。
- StartSel、StopSel（strings）：用于分隔文档中出现的查询词的字符串，以将它们与其他摘录的单词区分开来。默认值为“<b>”和“</b>”，它可以适用于HTML输出。
- FragmentDelimiter（string）：当显示多个片段时，片段会被这个字符串分隔。默认值为“... ”。

这些选项名称不区分大小写。如果字符串值包含空格或逗号，则必须用双引号引起来。

在基于非片段的标题生成中，ts\_headline为给定的query查找匹配项，并选择一个要显示的匹配项，优先选择在允许标题长度内具有更多查询词的匹配项。在基于片段的标题生成中，ts\_headline定位查询匹配项，并将每个匹配项拆分为“fragments”，每个匹配项不超过MaxWords个词，首选具有更多查询词的片段，并且在可能的情况下“拉伸”片段以包括周围的词。因此，当查询匹配跨越文档的大部分时，或者当需要显示多个匹配时，基于片段的模式更有用。在任一模式下，如果无法识别查询匹配项，则将显示文档中前 MinWords 单词的单个片段。

例如：

```
SELECT ts_headline('english',
  'The most common type of search
  is to find all documents containing given query terms
  and return them in order of their similarity to the
  query.',
  to_tsquery('english', 'query & similarity'));
      ts_headline
-----
```



containing given `<b>query</b>` terms +  
and return them in order of their `<b>similarity</b>` to the+  
`<b>query</b>`.

```
SELECT ts_headline('english',
  'Search terms may occur
many times in a document,
requiring ranking of the search matches to decide which
occurrences to display in the result.',
  to_tsquery('english', 'search & term'),
  'MaxFragments=10, MaxWords=7, MinWords=3, StartSel=<<, StopSel=>>');
      ts_headline
```

```
<<Search>> <<terms>> may occur +
many times ... ranking of the <<search>> matches to decide
```

`ts_headline`使用原始文档，而不是一个`tsvector`摘要，因此它可能很慢并且应该被小心使用。

## 12.4. 额外特性

这一节描述在文本搜索中有一些额外的函数和操作符。

### 12.4.1. 操纵文档

Section 12.3. 展示了未经处理的文本文档如何被转换成`tsvector`值。LightDB也提供了用于操纵已经为`tsvector`形式的文档的函数和操作符。

```
tsvector || tsvector
```

`tsvector`连接操作符返回一个向量，它结合了作为参数给出的两个向量的词位和位置信息。位置和权重标签在连接期间被保留。出现在右手向量中的位置被使用左手向量中提到的最大位置进行偏移，这样结果几乎等于在两个原始文档字符串的连接上执行`to_tsvector`的结果（这种等价不是完全的，因为从左手参数的尾端移除的任何停用词将会影响结果，而如果文本连接被使用它们就影响了右手参数中的词位位置）。

使用向量形式的连接而不是在应用`to_tsvector`之前连接文本的一个优点是你可以使用不同配置来解析文档的不同小节。此外，因为`setweight`函数按照相同的方式标记给定向量的所有词位，如果你想把文档的不同部分标注不同的权重，你就有必要解析文本并且在连接之前做`setweight`。

```
setweight(vector tsvector, weight "char") returns tsvector
```

`setweight`返回输入向量的一个拷贝，其中每一个位置都被标注为给定的权重：A、B、C或D（D是新向量的默认值并且并不会被显示在输出上）。向量被连接时会保留这些标签，允许来自文档的不同部分的词被排名函数给予不同的权重。

注意权重标签是应用到位置而不是词位。如果输入向量已经被剥离了位置，则`setweight`什么也不会做。

```
length(vector tsvector) returns integer
```

返回存储在向量中的词位数。

```
strip(vector tsvector) returns tsvector
```

返回一个向量，其中列出了和给定向量相同的词位，不过没有任何位置或者权重信息。其结果通常比未被剥离的向量小很多，但是用处也小很多。和未被剥离的向量一样，相

关度排名在已剥离的向量上也不起作用。此外，<-> (FOLLOWED BY) tsquery操作符不会匹配已剥离的输入，因为它无法确定词位之间的距离。

Table 9.4中有tsvector相关函数的完整列表。

## 12.4.2. 操纵查询

Section 12.3展示了未经处理的文本形式的查询如何被转换成tsquery值。LightDB也提供了用于操纵已经是tsquery形式的查询的函数和操作符。

tsquery && tsquery

返回用 AND 结合的两个给定查询。

tsquery || tsquery

返回用 OR 结合的两个给定查询。

!! tsquery

返回一个给定查询的反 (NOT)。

tsquery <-> tsquery

返回一个查询，它用<-> (FOLLOWED BY) tsquery操作符搜索两个紧跟的匹配，第一个匹配符合第一个给定的查询而第二个匹配符合第二个给定的查询。例如：

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
       ?column?
-----
'fat' <-> ( 'cat' | 'rat' )
```

tsquery\_phrase(query1 tsquery, query2 tsquery [, distance integer ]) returns tsquery

返回一个查询，它使用<N> tsquery操作符搜索两个距离为distance个词位的匹配，第一个匹配符合第一个给定的查询而第二个匹配符合第二个给定的查询。例如：

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
       tsquery_phrase
-----
'fat' <10> 'cat'
```

numnode(query tsquery) returns integer

返回一个tsquery中的结点数（词位外加操作符）。要确定查询是否有意义或者是否只包含停用词时，这个函数有用，在前一种情况它返回 > 0，后一种情况返回 0。例子：

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE: query contains only stopword(s) or doesn't contain lexeme(s),
 ignored
 numnode
-----
0

SELECT numnode('foo & bar'::tsquery);
 numnode
```

`querytree(query tsquery)` returns text

返回一个tsquery中可以被用来搜索一个索引的部分。这个函数可用来检测不可被索引的查询，例如那些只包含停用词或者只有否定术语的查询。例如：

```
SELECT querytree(to_tsquery('defined'));
querytree
-----
'defin'

SELECT querytree(to_tsquery('!defined'));
querytree
-----
T
```

### 12.4.2.1. 查询重写

`ts_rewrite`函数族在一个给定的tsquery中搜索一个目标子查询的出现，并且将每一次出现替换成一个替补子查询。本质上这个操作就是一个tsquery版本的子串替换。一个目标和替补的组合可以被看成是一个查询重写规则。一个这类重写规则的集合可以是一个强大的搜索助手。例如，你可以使用同义词扩展搜索（如，new york、big apple、nyc、gotham），或者收缩搜索来将用户导向某些特点主题。在这个特性和分类词典（Section 12.6.4 有些功能重叠。但是，你可以随时修改一组重写规则而无需重索引，而更新一个分类词典则要求进行重索引才能生效。

`ts_rewrite (query tsquery, target tsquery, substitute tsquery)` returns tsquery

这种形式的`ts_rewrite`简单地应用一个单一重写规则：不管target出现在query中的那个地方，它都被substitute替代。例如：

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
ts_rewrite
-----
'b' & 'c'
```

`ts_rewrite (query tsquery, select text)` returns tsquery

这种形式的`ts_rewrite`接受一个开始query和一个 SQL `select`命令，它们以一个文本字符串的形式给出。`select`必须得到tsquery类型的两列。对于select结果的每一行，在当前query值中出现的第一列值（目标）被第二列值（替补）所替换。例如：

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
ts_rewrite
-----
'b' & 'c'
```

注意当多个重写规则被以这种方式应用时，应用的顺序很重要；因此在实际中你会要求源查询按某些排序键ORDER BY。

让我们考虑一个现实的天文学例子。我们将使用表驱动的重写规则扩展查询supernovae：

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
-----
ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' )
```

我们可以通过只更新表来改变重写规则：

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
-----
ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' & '!nebula' )
```

当有很多重写规则时，重写可能会很慢，因为它要为每一个可能的匹配检查每一条规则。要过滤掉明显不符合的规则，我们可以为tsquery类型使用包含操作符。在下面的例子中，我们只选择那些可能匹配原始查询的规则：

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');
-----
ts_rewrite
-----
'b' & 'c'
```

### 12.4.3. 用于自动更新的触发器

#### Note

本节中描述的方法已被使用存储生成的列所淘汰，如 Section 12.2.2 中所述。

当使用一个单独的列来存储你的文档的tsvector表示时，有必要创建一个触发器在文档内容列改变时更新tsvector列。两个内建触发器函数可以用于这个目的，或者你可以编写你自己的触发器函数。

```
tsvector_update_trigger(tsvector_column_name, config_name, text_column_name
[, ... ])
tsvector_update_trigger_column(tsvector_column_name,
config_column_name, text_column_name [, ... ])
```

这些触发器函数在CREATE TRIGGER命令中指定的参数控制下，自动从一个或多个文本列计算一个tsvector列。它们使用的一个例子是：

```
CREATE TABLE messages (
title      text,
```

```

        body      text,
        tsv       tsvector
    );

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title      |          body          |          tsv
-----+-----+-----
title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title      |          body
-----+-----
title here | the body text is here

```

在创建了这个触发器后，在title或body中的任何修改将自动地被反映到tsv中，不需要应用来操心同步的问题。

第一个触发器参数必须是要被更新的tsvector列的名字。第二个参数指定要被用来执行转换的文本搜索配置。对于tsvector\_update\_trigger，配置名被简单地用第二个触发器参数给出。如上所示，它必须是模式限定的，因此该触发器行为不会因为search\_path中的改变而改变。对于tsvector\_update\_trigger\_column，第二个触发器参数是另一个表列的名称，它必须是类型regconfig。这允许做一种逐行的配置选择。剩下的参数是文本列的名称（类型为text、varchar或char）。它们将按给定的顺序被包括在文档中。NULL 值将被跳过（但是其他列仍将被索引）。

这些内建触发器的一个限制是它们将所有输入列同样对待。要对列进行不同的处理——例如，使标题的权重和正文的不同——就需要编写一个自定义触发器。下面是用PL/pgSQL作为触发器语言的一个例子：

```

CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
    new.tsv :=
        setweight(to_tsvector('pg_catalog.english', coalesce(new.title, '')), 'A')
    ||
        setweight(to_tsvector('pg_catalog.english', coalesce(new.body, '')), 'D');
    return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION messages_trigger();

```

记住当在触发器内创建tsvector值时，显式地指定配置名非常重要，这样列的内容才不会被default\_text\_search\_config的改变所影响。如果不这样做很可能导致问题，例如在转储并重新载入之后搜索结果改变。

## 12.4.4. 收集文档统计数据

ts\_stat被用于检查你的配置以及寻找候选的停用词。

```
ts_stat(sqlquery text, [ weights text, ]
```

```
OUT word text, OUT ndoc integer,
OUT nentry integer) returns setof record
```

sqlquery是一个文本值，它包含一个必须返回单一tsvector列的 SQL 查询。ts\_stat执行该查询并返回有关包含在该tsvector数据中的每一个可区分词位（词）的统计数据。返回的列是：

- word text — 一个词位的值
- ndoc integer — 词出现过的文档（tsvector）的数量
- nentry integer — 词出现的总次数

如果提供了权重，只有具有其中之一权重的出现才被计算在内。

例如，要在一个文档集合中查找十个最频繁的词：

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

同样的要求，但是只计算以权重A或B出现的次数：

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

## 12.5. 解析器

文本搜索解析器负责把未处理的文档文本划分成记号并且标识每一个记号的类型，而可能的类型集合由解析器本身定义。注意一个解析器完全不会修改文本 — 它简单地标识看似有理的词边界。因为这种有限的视野，对于应用相关的自定义解析器的需求就没有自定义字典那么强烈。目前LightDB只提供了一种内建解析器，它已经被证实对很多种应用都适用。

内建解析器被称为pg\_catalog.default。它识别 23 种记号类型，如Table 12.1所示。

Table 12.1. 默认解析器的记号类型

别名	描述	例子
asciword	单词，所有 ASCII 字母	elephant
word	单词，所有字母	mañana
numword	单词，字母和数字	beta1
ascihwnd	带连字符的单词，所有 ASCII	up-to-date
hwnd	带连字符的单词，所有字母	lógico-matemática
numhwnd	带连字符的单词，字母和数字	postgresql-beta1
hwnd_asciipart	带连字符的单词部分，所有 ASCII	postgresql in the context postgresql-beta1
hwnd_part	带连字符的单词部分，所有字母	lógico or matemática in the context lógico-matemática
hwnd_numpart	带连字符的单词部分，字母和数字	beta1 in the context postgresql- beta1

别名	描述	例子
email	Email 地址	foo@example.com
protocol	协议头部	http://
url	URL	example.com/stuff/index.html
host	主机	example.com
url_path	URL 路径	/stuff/index.html, in the context of a URL
file	文件或路径名	/usr/local/foo.txt, if not within a URL
sfloat	科学记数法	-1.234e56
float	十进制记数法	-1.234
int	有符号整数	-1234
uint	无符号整数	1234
version	版本号	8.3.0
tag	XML 标签	<a href="dictionaries.html">
entity	XML 实体	&amp;
blank	空格符号	(其他不识别的任意空白或标点符号)

### Note

解析器的一个“字母”的概念由数据库的区域设置决定，具体是lc\_ctype。只包含基本 ASCII 字母的词被报告为一个单独的记号类型，因为有时可以用来区别它们。在大部分欧洲语言中，记号类型word和asciword应该被同样对待。

email不支持RFC 5322<sup>1</sup>定义的所有合法 email 字符。具体来说，对 email 用户名被支持的非字母数字字符只有句点、破折号和下划线。

解析器有可能从同一份文本得出相互覆盖的记号。例如，一个带连字符的词可能会被报告为一整个词或者多个部分：

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
-----
```

alias	description	token
numhword	Hyphenated word, letters and digits	foo-bar-beta1
hword_asciipart	Hyphenated word part, all ASCII	foo
blank	Space symbols	-
hword_asciipart	Hyphenated word part, all ASCII	bar
blank	Space symbols	-
hword_numpart	Hyphenated word part, letters and digits	beta1

这种行为是值得要的，因为它允许对整个复合词和每个部分进行搜索。这里是另一个例子：

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/
index.html');
-----
```

alias	description	token
-------	-------------	-------

<sup>1</sup> <https://tools.ietf.org/html/rfc5322>

protocol	Protocol head	http://
url	URL	example.com/stuff/index.html
host	Host	example.com
url_path	URL path	/stuff/index.html

## 12.6. 词典

词典被用来消除不被搜索考虑的词 (stop words)、并被用来正规化词这样同一个词的不同派生形式将会匹配。一个被成功地正规化的词被称为一个词位。除了提高搜索质量，正规化和移除停用词减小了文档的tsvector表示的尺寸，因而提高了性能。正规化不会总是有语言上的意义并且通常依赖于应用的语义。

一些正规化的例子：

- 语言学的 — Ispell 词典尝试将输入词缩减为一种正规化的形式；词干分析器词典移除词的结尾
- URL位置可以被规范化来得到等效的 URL 匹配：
  - `http://www.pgsql.ru/db/mw/index.html`
  - `http://www.pgsql.ru/db/mw/`
  - `http://www.pgsql.ru/db/./db/mw/index.html`
- 颜色名可以被它们的十六进制值替换，例如red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF
- 如果索引数字，我们可以移除某些小数位来缩减可能的数字的范围，因此如果只保留小数点后两位，例如3.14159265359、3.1415926、3.14在正规化后会变得相同。

一个词典是一个程序，它接受一个记号作为输入，并返回：

- 如果输入的记号对词典是已知的，则返回一个词位数组（注意一个记号可能产生多于一个词位）
- 一个TSL\_FILTER标志被设置的单一词位，用一个新记号来替换要被传递给后续字典的原始记号（做这件事的一个字典被称为一个过滤字典）
- 如果字典知道该记号但它是一个停用词，则返回一个空数组
- 如果字典不识别该输入记号，则返回NULL

LightDB为许多语言提供了预定义的字典。也有多种预定义模板可以被用于创建带自定义参数的新词典。每一种预定义词典模板在下面描述。如果没有合适的现有模板，可以创建新的；例子见LightDB发布的contrib/区域。

一个文本搜索配置把一个解析器和一组处理解析器输出记号的词典绑定在一起。对于每一中解析器能返回的记号类型，配置都指定了一个单独的词典列表。当该类型的一个记号被解析器找到时，每一个词典都被按照顺序查询，知道某个词典将其识别为一个已知词。如果它被标识为一个停用词或者没有一个词典识别它，它将被丢弃并且不会被索引和用于搜索。通常，第一个返回非NULL输出的词典决定结果，并且任何剩下的词典都不会被查找；但是一个过滤词典可以将给定词替换为一个被修改的词，它再被传递给后续的词典。

配置一个词典列表的通用规则是将最狭窄、最特定的词典放在第一位，然后是更加通用的词典，以一个非常通用的词典结尾，像一个Snowball词干分析器或什么都识别的simple。例如，对于一个天文学相关的搜索 (astro\_en 配置) 我们可以把记号类型asciiword (ASCII词) 绑定到一个天文学术语的分类词典、一个通用英语词典和一个Snowball英语词干分析器：

```
ALTER TEXT SEARCH CONFIGURATION astro_en
  ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

一个过滤词典可以被放置在列表中的任意位置，除了在最后，因为过滤词典放在最后就等于无用。过滤词典可用于部分正规化词来简化后续词典的工作。例如，一个过滤词典可以被用来从音标字母中移除重音符号，就像unaccent模块所做的。



## 12.6.1. 停用词

停用词是非常常用、在几乎每一个文档中出现并且没有任何区分度的词。因此，在全文搜索的环境中它们可以被忽略。例如，每一段英语文本都包含a和the等词，因此把它们存储在一个索引中是没有用处的。但是，停用词确实会影响在tsvector中的位置，这进而会影响排名：

```
SELECT to_tsvector('english', 'in the list of stop words');
       to_tsvector
-----
'list':3 'stop':5 'word':6
```

缺失的位置 1、2、4 是因为停用词。文档的排名计算在使用和不使用停用词的情况下是很不同的：

```
SELECT ts_rank_cd (to_tsvector('english', 'in the list of stop words'),
                  to_tsquery('list & stop'));
       ts_rank_cd
-----
          0.05
```

```
SELECT ts_rank_cd (to_tsvector('english', 'list stop words'), to_tsquery('list &
stop'));
       ts_rank_cd
-----
          0.1
```

如何对待停用词是由指定词典决定的。例如，ispell词典首先正规化词并且查看停用词列表，而Snowball词干分析器首先检查停用词的列表。这种不同行为的原因是一冲降低噪声的尝试。

## 12.6.2. 简单词典

simple词典模板的操作是将输入记号转换为小写形式并且根据一个停用词文件检查它。如果该记号在该文件中被找到，则返回一个空数组，导致该记号被丢弃。否则，该词的小写形式被返回作为正规化的词位。作为一种选择，该词典可以被配置为将非停用词报告为未识别，允许它们被传递给列表中的下一个词典。

下面是一个使用simple模板的词典定义的例子：

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

这里，english是一个停用词文件的基本名称。该文件的全名将是\$SHAREDIR/tsearch\_data/english.stop，其中\$SHAREDIR表示LightDB安装的共享数据目录，通常是/usr/local/share/postgresql（如果不确定，使用lt\_config --sharedir）。该文件格式是一个词的列表，每行一个。空行和尾部的空格都被忽略，并且大写也被折叠成小写，但是没有其他对该文件内容的处理。

现在我们能够测试我们的词典：

```
SELECT ts_lexize('public.simple_dict', 'YeS');
       ts_lexize
```

```

-----
{yes}

SELECT ts_lexize('public.simple_dict', 'The');
ts_lexize
-----
{}

```

如果没有在停用词文件中找到，我们也可以选择返回NULL而不是小写形式的词。这种行为可以通过设置词典的Accept参数为false来选择。继续该例子：

```

ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict', 'YeS');
ts_lexize
-----

```

```

SELECT ts_lexize('public.simple_dict', 'The');
ts_lexize
-----
{}

```

在使用默认值Accept = true，只有把一个simple词典放在词典列表的尾部才有用，因为它将不会传递任何记号给后续的词典。相反，Accept = false只有当至少有一个后续词典的情况下才有用。

### Caution

大部分类型的词典依赖于配置文件，例如停用词文件。这些文件必须被存储为UTF-8 编码。当它们被读入服务器时，如果存在不同，它们将被翻译成真实的数据库编码。

### Caution

通常，当一个词典配置文件第一次在数据库会话中使用时，数据库会话将只读取它一次。如果你修改了一个配置文件并且想强迫现有的会话取得新内容，可以在该词典上发出一个ALTER TEXT SEARCH DICTIONARY命令。这可以是一次“假”更新，它并不实际修改任何参数值。

## 12.6.3. 同义词词典

这个词典模板被用来创建用于同义词替换的词典。不支持短语（使用分类词典模板（Section 12.6.4 可以支持）。一个同义词词典可以被用来解决语言学问题，例如，阻止一个英语词干分析器词典把词“Paris”缩减成“pari”。在同义词词典中有一行Paris pari并把它放在english\_stem词典之前就足够了。例如：

```

SELECT * FROM ts_debug('english', 'Paris');
  alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (

```

```

    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
  alias | description | token | dictionaries | dictionary |
lexemes
-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym |
{paris}

```

synonym模板要求的唯一参数是SYNONYMS，它是其配置文件的基本名——上例中的my\_synonyms。该文件的完整名称将是\$SHAREDIR/tsearch\_data/my\_synonyms.syn（其中\$SHAREDIR表示LightDB安装的共享数据目录）。该文件格式是每行一个要被替换的词，后面跟着它的同义词，用空白分隔。空行和结尾的空格会被忽略。

synonym模板还有一个可选的参数CaseSensitive，其默认值为false。当CaseSensitive为false时，同义词文件中的词被折叠成小写，这和输入记号一样。当它为true时，词和记号将不会被折叠成小写，但是比较时就好像被折叠过一样。

一个星号(\*)可以被放置在配置文件中一个同义词的末尾。这表示该同义词是一个前缀。当项被用在to\_tsvector()中时，星号会被忽略；当它被用在to\_tsquery()中时，结果将是一个带有前缀匹配标记器（见Section 12.3.2的查询项。例如，假设我们在\$SHAREDIR/tsearch\_data/synonym\_sample.syn中有这些项：

```

postgres      pgsq1
postgresql    pgsq1
postgre pgsq1
google        googl
indices index*

```

那么我们将得到这些结果：

```

mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym,
      synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn', 'indices');
 ts_lexize
-----
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst', 'indices');
 to_tsvector
-----
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst', 'indices');
 to_tsquery
-----

```

```
'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
          tsvector
-----
'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst',
'indices');
?column?
-----
t
(1 row)
```

## 12.6.4. 分类词典

一个分类词典（有时被简写成TZ）是一个词的集合，其中包括了词与短语之间的联系，即广义词（BT）、狭义词（NT）、首选词、非首选词、相关词等。

基本上一个分类词典会用一个首选词替换所有非首选词，并且也可选择地保留原始术语用于索引。LightDB的分类词典的当前实现是同义词词典的一个扩展，并增加了短语支持。一个分类词典要求一个下列格式的配置文件：

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

其中冒号（:）符号扮演了一个短语及其替换之间的定界符。

一个分类词典使用一个子词典（在词典的配置中指定）在检查短语匹配之前正规化输入文本。只能选择一个子词典。如果子词典无法识别一个词，将报告一个错误。在这种情况下，你应该移除该词的使用或者让子词典学会这个词。你可以在一个被索引词的开头放上一个星号（\*）来跳过在其上应用子词典，但是所有采样词必须被子词典知道。

如果有多个短语匹配输入，则分类词典选择最长的那一个，并且使用最后的定义打破连结。

由子词典识别的特定停用词不能够被指定；改用?标记任何可以出现停用词的地方。例如，假定根据子词典a和the是停用词：

```
? one ? two : ssw
```

匹配a one the two和the one a two；两者都将被ssw替换。

由于一个分类词典具有识别短语的能力，它必须记住它的状态并与解析器交互。一个分类词典使用这些任务来检查它是否应当处理下一个词或者停止累积。分类词典必须被小心地配置。例如，如果分类词典被分配只处理asciword记号，则一个形如one 7的分类词典定义将不会工作，因为记号类型uint没有被分配给该分类词典。

### Caution

在索引期间要用到分类词典，因此分类词典参数中的任何变化都要求重索引。对于大多数其他索引类型，例如增加或移除停用词等小改动都不会强制重索引。

### 12.6.4.1. 分类词典配置

要定义一个新的分类词典，可使用thesaurus模板。例如：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

这里：

- thesaurus\_simple是新词典的名称
- mythesaurus是分类词典配置文件的基础名称（它的全名将是\$SHAREDIR/tsearch\_data/mythesaurus.ths，其中\$SHAREDIR表示安装的共享数据目录）。
- pg\_catalog.english\_stem是要用于分类词典正规化的子词典（这里是一个 Snowball 英语词干分析器）。注意子词典将拥有它自己的配置（例如停用词），但这里没有展示。

现在可以在配置中把分类词典thesaurus\_simple绑定到想要的记号类型上，例如：

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

### 12.6.4.2. 分类词典例子

考虑一个简单的天文学分类词典thesaurus\_astro，它包含一些天文学词组合：

```
supernovae stars : sn
crab nebulae : crab
```

下面我们创建一个词典并绑定一些记号类型到一个天文学分类词典以及英语词干分析器：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

现在我们可以看看它如何工作。ts\_lexize对于测试一个分类词典用处不大，因为它把它的输入看成是一个单一记号。我们可以用plainto\_tsquery和to\_tsvector，它们将把其输入字符串打断成多个记号：

```
SELECT plainto_tsquery('supernova star');
plainto_tsquery
-----
'sn'

SELECT to_tsvector('supernova star');
to_tsvector
-----
```

```
'sn':1
```

原则上，如果你对参数加了引号，你可以使用to\_tsquery：

```
SELECT to_tsquery('' supernova star'');
to_tsquery
-----
'sn'
```

注意在thesaurus\_astro中supernova star匹配supernovae stars，因为我们在分类词典定义中指定了english\_stem词干分析器。该词干分析器移除了e和s。

要和替补一样也索引原始短语，只要将它包含在定义的右手部分中：

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery(' supernova star');
plainto_tsquery
-----
'sn' & 'supernova' & 'star'
```

## 12.6.5. Ispell 词典

Ispell词典模板支持词法词典，它可以把一个词的很多不同语言学的形式正规化成相同的词位。例如，一个英语Ispell词典可以匹配搜索词bank的词尾变化和词形变化，例如banking、banked、banks、banks'和bank's。

标准的LightDB发布不包括任何Ispell配置文件。用于很多种语言的词典可以从Ispell<sup>2</sup>得到。此外，也支持一些更现代的词典文件格式 — MySpell<sup>3</sup> (00 < 2.0.1) 和Hunspell<sup>4</sup> (00 >= 2.0.2)。一个很大的词典列表在OpenOffice Wiki<sup>5</sup>上可以得到。

要创建一个Ispell词典，执行这三步：

- 下载词典配置文件。OpenOffice扩展文件的扩展名是.oxt。有必要抽取.aff和.dic文件，把扩展改为.affix和.dict。对于某些词典文件，还需要使用下面的命令把字符转换成UTF-8 编码（例如挪威语词典）：

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- 拷贝文件到\$SHAREDIR/tsearch\_data目录
- 用下面的命令把文件载入到 LightDB：

```
CREATE TEXT SEARCH DICTIONARY english_hunspell (
    TEMPLATE = ispell,
    DictFile = en_us,
    AffFile = en_us,
    Stopwords = english);
```

这里，DictFile、AffFile和StopWords指定词典、词缀和停用词文件的基础名称。停用词文件的格式和前面解释的simple词典类型相同。其他文件的格式在这里没有指定，但是也可以从上面提到的网站获得。

<sup>2</sup> <https://www.cs.hmc.edu/~geoff/ispell.html>

<sup>3</sup> <https://en.wikipedia.org/wiki/MySpell>

<sup>4</sup> <https://sourceforge.net/projects/hunspell/>

<sup>5</sup> <https://wiki.services.openoffice.org/wiki/Dictionaries>

Ispell 词典通常识别一个有限集合的词，这样它们后面应该跟着另一个更广义的词典；例如，一个 Snowball 词典，它可以识别所有东西。

Ispell的.affix文件具有下面的结构：

```

prefixes
flag *A:
    .          > RE      # As in enter > reenter
suffixes
flag T:
    E          > ST      # As in late > latest
    [^AEIOU]Y > -Y, IEST # As in dirty > dirtiest
    [AEIOU]Y  > EST     # As in gray > grayest
    [^EY]     > EST     # As in small > smallest

```

.dict文件具有下面的结构：

```

lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS

```

.dict文件的格式是：

```
basic_form/affix_class_name
```

在.affix文件中，每一个词缀标志以下面的格式描述：

```
condition > [-stripping_letters,] adding_affix
```

这里的条件具有和正则表达式相似的格式。它可以使用分组[...]和[^...]。例如，[AEIOU]Y表示词的最后一个字母是“y”并且倒数第二个字母是“a”、“e”、“i”、“o”或者“u”。[^EY]表示最后一个字母既不是“e”也不是“y”。

Ispell 词典支持划分复合词，这是一个有用的特性。注意词缀文件应该用compoundwords controlled语句指定一个特殊标志，它标记可以参与到复合格式中的词典词：

```
compoundwords controlled z
```

下面是挪威语的一些例子：

```

SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmasterassistent');
      {over, buljong, terning, pakk, mester, assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
      {sjokoladefabrikk, sjokolade, fabrikk}

```

MySpell格式是Hunspell格式的一个子集。Hunspell的.affix文件具有下面的结构：

```

PFX A Y 1
PFX A 0   re      .
SFX T N 4
SFX T 0   st      e
SFX T y   iest    [^aeiou]y

```

```
SFX T 0 est [aeiou]y
SFX T 0 est [^ey]
```

一个词缀类的第一行是头部。头部后面列出了词缀规则的域：

- 参数名（PFX 或者 SFX）
- 标志（词缀类的名称）
- 从该词的开始（前缀）或者结尾（后缀）剥离字符
- 增加词缀
- 和正则表达式格式类似的条件。

.dict文件看起来和Ispell的.dict文件相似：

```
larder/M
lardy/RT
large/RSPMYT
largehearted
```

### Note

MySpell 不支持复合词。Hunspell则对复合词有更好的支持。当前，LightDB 只实现了 Hunspell 中基本的复合词操作。

## 12.6.6. Snowball 词典

Snowball词典模板基于 Martin Porter 的一个项目，他是流行的英语 Porter 词干分析算法的发明者。Snowball 现在对许多语言提供词干分析算法（详见Snowball 站点<sup>6</sup>）。每一个算法懂得按照其语言中的拼写，如何缩减词的常见变体形式为一个基础或词干。一个 Snowball 词典要求一个language参数来标识要用哪种词干分析器，并且可以选择地指定一个stopword文件名来给出一个要被消除的词列表（LightDB的标准停用词列表也是由 Snowball 项目提供的）。例如，有一个内建的定义等效于

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

停用词文件格式和已经解释的一样。

一个Snowball词典识别所有的东西，不管它能不能简化该词，因此它应当被放置在词典列表的最后。把它放在任何其他词典前面是没有用处的，因为一个记号永远不会穿过它而进入到下一个词典。

## 12.7. 配置例子

一个文本搜索配置指定了将一个文档转换成一个tsvector所需的所有选项：用于把文本分解成记号的解析器，以及用于将每一个记号转换成词位的词典。每一次to\_tsvector或to\_tsquery的调用都需要一个文本搜索配置来执行其处理。配置参数default\_text\_search\_config指定了默认配置的名称，如果忽略了显式的配置参数，文本搜索函数将会使用它。它可以在lightdb.conf中设置，或者使用SET命令为一个单独的会话设置。

<sup>6</sup> <https://snowballstem.org/>



有一些预定义的文本搜索配置可用，并且你可以容易地创建自定义的配置。为了便于管理文本搜索对象，可以使用一组SQL命令，并且有多个`tsql`命令可以显示有关文本搜索对象（Section 12.10 的信息）。

作为一个例子，我们将创建一个配置`pg`，从复制内建的`english`配置开始：

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

我们将使用一个 `LightDB` 相关的同义词列表，并将它存储在`$SHAREDIR/tsearch_data/pg_dict.syn`中。文件内容看起来像：

```
postgres    pg
pgsql       pg
postgresql  pg
```

我们定义同义词词典如下：

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

接下来我们注册`Ispell`词典`english_ispell`，它有其自己的配置文件：

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

现在我们可以配置`pg`中建立词的映射：

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                    word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

我们选择不索引或搜索某些内建配置确实处理的记号类型：

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

现在我们可以测试我们的配置：

```
SELECT * FROM ts_debug(' public.pg', '
LightDB, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

下一个步骤是设置会话让它使用新配置，它被创建在`public`模式中：

```

=> \dF
    List of text search configurations
 Schema | Name | Description
-----+-----+-----
 public | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
-----
 public.pg

```

## 12.8. 测试和调试文本搜索

一个自定义文本搜索配置的行为很容易变得混乱。本节中描述的函数对于测试文本搜索对象有用。你可以测试一个完整的配置，或者独立测试解析器和词典。

### 12.8.1. 配置测试

函数`ts_debug`允许简单地测试一个文本搜索配置。

```

ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
         OUT dictionaries regdictionary[],
         OUT dictionary regdictionary,
         OUT lexemes text[])
returns setof record

```

`ts_debug`显示`document`的每一个记号的信息，记号由解析器产生并由配置的词典处理过。该函数使用由`config`指定的配置，如果该参数被忽略则使用`default_text_search_config`指定的配置。

`ts_debug`为解析器在文本中标识的每一个记号返回一行。被返回的列是：

- `alias text` — 记号类型的短名称
- `description text` — 记号类型的描述
- `token text` — 记号的文本
- `dictionaries regdictionary[]` — 配置为这种记号类型选择的词典
- `dictionary regdictionary` — 识别该记号的词典，如果没有词典能识别则为NULL
- `lexemes text[]` — 识别该记号的词典产生的词位，如果没有词典能识别则为NULL；一个空数组（{}）表示该记号被识别为一个停用词

这里是一个简单的例子：

```

SELECT * FROM ts_debug('english', 'a fat cat sat on a mat - it ate a fat
rats');
  alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | |
asciword | Word, all ASCII | fat | {english_stem} | english_stem | {fat}

```

blank	Space symbols		{}		
asciiword	Word, all ASCII	cat	{english_stem}	english_stem	{cat}
blank	Space symbols		{}		
asciiword	Word, all ASCII	sat	{english_stem}	english_stem	{sat}
blank	Space symbols		{}		
asciiword	Word, all ASCII	on	{english_stem}	english_stem	{}
blank	Space symbols		{}		
asciiword	Word, all ASCII	a	{english_stem}	english_stem	{}
blank	Space symbols		{}		
asciiword	Word, all ASCII	mat	{english_stem}	english_stem	{mat}
blank	Space symbols		{}		
blank	Space symbols	-	{}		
asciiword	Word, all ASCII	it	{english_stem}	english_stem	{}
blank	Space symbols		{}		
asciiword	Word, all ASCII	ate	{english_stem}	english_stem	{ate}
blank	Space symbols		{}		
asciiword	Word, all ASCII	a	{english_stem}	english_stem	{}
blank	Space symbols		{}		
asciiword	Word, all ASCII	fat	{english_stem}	english_stem	{fat}
blank	Space symbols		{}		
asciiword	Word, all ASCII	rats	{english_stem}	english_stem	{rat}

为了一个更广泛的示范，我们先为英语语言创建一个public.english配置和 Ispell 词典：

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );
```

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

```
ALTER TEXT SEARCH CONFIGURATION public.english
    ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;
```

```
SELECT * FROM ts_debug('public.english', 'The Brightest supernovaes');
  alias | description | token | dictionaries |
dictionary | lexemes
-----+-----+-----+-----+
+-----+-----+-----+-----+
asciiword | Word, all ASCII | The | {english_ispell,english_stem} |
english_ispell | {}
blank | Space symbols | | {} |
|
asciiword | Word, all ASCII | Brightest | {english_ispell,english_stem} |
english_ispell | {bright}
blank | Space symbols | | {} |
|
asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} |
english_stem | {supernova}
```

在这个例子中，词Brightest被解析器识别为一个ASCII 词（别名asciiword）。对于这种记号类型，词典列表是english\_ispell和english\_stem。该词被english\_ispell识别，这个词典将它缩减为名词bright。词supernovaes对于english\_ispell词典是未知的，因此它被传递给下一个词典，并且幸运地是，它被识别了（实际上，english\_stem是一个 Snowball 词典，它识别所有的东西；这也是为什么它被放置在词典列表的尾部）。

词The被english\_ispell词典识别为一个停用词（Section 12.6.1）并且将不会被索引。空格也被丢弃，因为该配置没有为它们提供词典。

你可以通过显式地指定你想看哪些列来缩减输出的宽度：

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english', 'The Brightest supernovaes');
  alias | token | dictionary | lexemes
-----+-----+-----+-----
asciword | The | english_ispell | {}
blank | | | |
asciword | Brightest | english_ispell | {bright}
blank | | | |
asciword | supernovaes | english_stem | {supernova}
```

## 12.8.2. 解析器测试

下列函数允许直接测试一个文本搜索解析器。

```
ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
         OUT tokid integer, OUT token text) returns setof record
```

ts\_parse解析给定的document并返回一系列记录，每一个记录对应一个由解析产生的记号。每一个记录包括一个tokid展示分配给记号的类型以及一个token展示记号的文本。例如：

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-----+-----
    22 | 123
    12 | 
    12 | -
     1 | a
    12 | 
     1 | number
```

```
ts_token_type(parser_name text, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
ts_token_type(parser_oid oid, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
```

ts\_token\_type返回一个表，该表描述指定解析器能够识别的每一种记号类型。对于每一种记号类型，该表给出了解析器用来标注该类型记号的整数tokid，还给出了在配置命令中命名该记号类型的alias，以及一个简短的description。例如：

```
SELECT * FROM ts_token_type('default');
 tokid | alias | description
-----+-----+-----
     1 | asciword | Word, all ASCII
     2 | word | Word, all letters
     3 | numword | Word, letters and digits
```

4		email		Email address
5		url		URL
6		host		Host
7		sfloat		Scientific notation
8		version		Version number
9		hword_numpart		Hyphenated word part, letters and digits
10		hword_part		Hyphenated word part, all letters
11		hword_asciipart		Hyphenated word part, all ASCII
12		blank		Space symbols
13		tag		XML tag
14		protocol		Protocol head
15		numhword		Hyphenated word, letters and digits
16		asciihword		Hyphenated word, all ASCII
17		hword		Hyphenated word, all letters
18		url_path		URL path
19		file		File or path name
20		float		Decimal notation
21		int		Signed integer
22		uint		Unsigned integer
23		entity		XML entity

### 12.8.3. 词典测试

`ts_lexize`函数帮助词典测试。

`ts_lexize(dict regdictionary, token text)` returns `text[]`

如果输入的`token`是该词典已知的，则`ts_lexize`返回一个词位数组；如果记号是词典已知的但是它是一个停用词，则返回一个空数组；或者如果它对词典是未知词，则返回NULL。

例子：

```
SELECT ts_lexize('english_stem', 'stars');
ts_lexize
-----
{star}
```

```
SELECT ts_lexize('english_stem', 'a');
ts_lexize
-----
{}
```

#### Note

`ts_lexize`函数期望一个单一记号而不是文本。下面的情况会让它搞混：

```
SELECT ts_lexize('thesaurus_astro', 'supernovae stars') is null;
?column?
-----
t
```

分类词典`thesaurus_astro`确实知道短语`supernovae stars`，但是`ts_lexize`会失败，因为它无法解析输入文本而把它当做一个单一记号。可以使用`plainto_tsquery`或`to_tsvector`来测试分类词典，例如：

```
SELECT plainto_tsquery('supernovae stars');
plainto_tsquery
-----
'sn'
```

## 12.9. GIN 和 GiST 索引类型

有两种索引可以用于加速全文搜索：GIN 和 GiST。请注意，索引对于全文搜索不是强制性的，但在经常搜索某个列的情况下，索引通常是可取的。

要创建这样的索引，请执行以下操作之一：

```
CREATE INDEX name ON table USING GIN (column);
```

创建一个基于 GIN（通用倒排索引）的索引。column 必须是 tsvector 类型。

```
CREATE INDEX name ON table USING GIST (column [ { DEFAULT | tsvector_ops } (siglen = number) ] );
```

创建一个基于 GiST（通用搜索树）的索引。column 可以是 tsvector 或 tsquery 类型。可选整数参数 siglen 确定以字节为单位的签名长度（详见下文）。

GIN 索引是更好的文本搜索索引类型。作为倒排索引，每个词（词位）在其中都有一个索引项，其中有压缩过的匹配位置的列表。多词搜索可以找到第一个匹配，然后使用该索引移除缺少额外词的行。GIN 索引只存储 tsvector 值的词（词位），并且不存储它们的权重标签。因此，在使用涉及权重的查询时需要一次在表行上的重新检查。

一个 GiST 索引是有损的，这表示索引可能产生假匹配，并且有必要检查真实的表行来消除这种假匹配（LightDB 在需要时会自动做这一步）。GiST 索引之所以是有损的，是因为每一个文档在索引中被表示为一个定长的签名。以字节为单位的签名长度由可选整数参数 siglen 的值决定。默认签名长度（未指定 siglen 时）为 124 字节，最大签名长度为 2024 字节。该签名通过哈希每一个词到一个 n 位串中的一个单位来产生，通过将所有这些位 OR 在一起产生一个 n 位的文档签名。当两个词哈希到同一个位位置时就会产生假匹配。如果查询中所有词都有匹配（真或假），则必须检索表行查看匹配是否正确。更长的签名导致更精确的搜索（扫描索引的一小部分和更少的堆页面），但代价是更大的索引。

GiST 索引可以被覆盖，例如使用 INCLUDE 子句。包含的列可以具有没有任何 GiST 操作符类的数据类型。包含的属性将非压缩存储。

有损性导致的性能下降归因于不必要的表记录（即被证实为假匹配的记录）获取。因为表记录的随机访问是较慢的，这限制了 GiST 索引的可用性。假匹配的可能性取决于几个因素，特别是唯一词的数量，因此推荐使用词典来缩减这个数量。

注意 GIN 索引的构件时间常常可以通过增加 maintenance\_work\_mem 来改进，而 GiST 索引的构件时间则与该参数无关。

对大集合分区并正确使用 GIN 和 GiST 索引允许实现带有在线更新的快速搜索。分区可以在数据库层面上使用表继承来完成，或者是通过将文档分布在服务器上并收集外部的搜索结果，例如通过外部数据访问。后者是可能的，因为排名函数只使用本地信息。

## 12.10. ltsql 支持

关于文本搜索配置对象的信息可以在 ltsql 中使用一组命令获得：

```
\dF {d, p, t} [+] [PATTERN]
```

可选的+能产生更多细节。

可选参数PATTERN可以是一个文本搜索对象的名称，可以是模式限定的。如果PATTERN被忽略，则所有可见对象的信息都将被显示。PATTERN可以是一个正则表达式并且可以为模式和对象名称提供独立的模式。下面的例子展示了这些特性：

```
=> \dF *fulltext*
      List of text search configurations
  Schema | Name          | Description
-----+-----+-----
  public | fulltext_cfg |
```

```
=> \dF *.fulltext*
      List of text search configurations
  Schema | Name          | Description
-----+-----+-----
  fulltext | fulltext_cfg |
  public   | fulltext_cfg |
```

可用的命令是：

```
\dF[+] [PATTERN]
```

列出文本搜索配置（加上+得到更多细节）。

```
=> \dF russian
      List of text search configurations
  Schema | Name | Description
-----+-----+-----
  pg_catalog | russian | configuration for russian language
```

```
=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
  Token | Dictionaries
-----+-----
  asciihword | english_stem
  asciiword  | english_stem
  email      | simple
  file       | simple
  float      | simple
  host       | simple
  hword      | russian_stem
  hword_asciipart | english_stem
  hword_numpart | simple
  hword_part  | russian_stem
  int        | simple
  numhword   | simple
  numword    | simple
  sfloat     | simple
  uint       | simple
  url        | simple
  url_path   | simple
  version    | simple
```

```
word | russian_stem
```

```
\dFd[+] [PATTERN]
```

列出文本搜索词典（加上+得到更多细节）。

```
=> \dFd
```

```

List of text search dictionaries
Schema | Name | Description
-----+-----+-----
pg_catalog | arabic_stem | snowball stemmer for arabic language
pg_catalog | danish_stem | snowball stemmer for danish language
pg_catalog | dutch_stem | snowball stemmer for dutch language
pg_catalog | english_stem | snowball stemmer for english language
pg_catalog | finnish_stem | snowball stemmer for finnish language
pg_catalog | french_stem | snowball stemmer for french language
pg_catalog | german_stem | snowball stemmer for german language
pg_catalog | greek_stem | snowball stemmer for greek language
pg_catalog | hungarian_stem | snowball stemmer for hungarian language
pg_catalog | indonesian_stem | snowball stemmer for indonesian language
pg_catalog | irish_stem | snowball stemmer for irish language
pg_catalog | italian_stem | snowball stemmer for italian language
pg_catalog | lithuanian_stem | snowball stemmer for lithuanian language
pg_catalog | nepali_stem | snowball stemmer for nepali language
pg_catalog | norwegian_stem | snowball stemmer for norwegian language
pg_catalog | portuguese_stem | snowball stemmer for portuguese language
pg_catalog | romanian_stem | snowball stemmer for romanian language
pg_catalog | russian_stem | snowball stemmer for russian language
pg_catalog | simple | simple dictionary: just lower case and check
for stopword
pg_catalog | spanish_stem | snowball stemmer for spanish language
pg_catalog | swedish_stem | snowball stemmer for swedish language
pg_catalog | tamil_stem | snowball stemmer for tamil language
pg_catalog | turkish_stem | snowball stemmer for turkish language

```

```
\dFp[+] [PATTERN]
```

列出文本搜索解析器（加上+得到更多细节）。

```
=> \dFp
```

```

List of text search parsers
Schema | Name | Description
-----+-----+-----
pg_catalog | default | default word parser
=> \dFp+
Text search parser "pg_catalog.default"
Method | Function | Description
-----+-----+-----
Start parse | prsd_start |
Get next token | prsd_nexttoken |
End parse | prsd_end |
Get headline | prsd_headline |
Get token types | prsd_lextype |

```

```

Token types for parser "pg_catalog.default"
Token name | Description

```



asciihword	Hyphenated word, all ASCII
asciiword	Word, all ASCII
blank	Space symbols
email	Email address
entity	XML entity
file	File or path name
float	Decimal notation
host	Host
hword	Hyphenated word, all letters
hword_asciipart	Hyphenated word part, all ASCII
hword_numpart	Hyphenated word part, letters and digits
hword_part	Hyphenated word part, all letters
int	Signed integer
numhword	Hyphenated word, letters and digits
numword	Word, letters and digits
protocol	Protocol head
sfloat	Scientific notation
tag	XML tag
uint	Unsigned integer
url	URL
url_path	URL path
version	Version number
word	Word, all letters

(23 rows)

`\dFt[+] [PATTERN]`

列出文本搜索模板（加上+得到更多细节）。

`=> \dFt`

List of text search templates		
Schema	Name	Description
pg_catalog	ispell	ispell dictionary
pg_catalog	simple	simple dictionary: just lower case and check for stopword
pg_catalog	snowball	snowball stemmer
pg_catalog	synonym	synonym dictionary: replace word by its synonym
pg_catalog	thesaurus	thesaurus dictionary: phrase by phrase substitution

## 12.11. 限制

LightDB的文本搜索特性的当前限制是：

- 每一个词位的长度必须小于 2K 字节
- 一个tsvector（词位 + 位置）的长度必须小于 1 兆字节
- 词位的数量必须小于  $2^{64}$
- tsvector中的位置值必须大于 0 并且小于 16,383
- $\langle N \rangle$  (FOLLOWED BY) tsquery操作符中的匹配距离不能超过 16,384
- 每个词位不超过 256 个位置
- 一个tsquery中结点（词位 + 操作符）的个数必须小于 32,768

为了对比，LightDB 8.1 的文档包含 10,441 个唯一词，总数 335,420 个词，并且最频繁的词“postgresql”在 655 个文档中被提到 6,127 次。

另一个例子 — LightDB的邮件列表归档在 461,020 条消息的 57,491,343 个词位中包含 910,989 个唯一词。

---

# Chapter 13. 并发控制

本章描述LightDB数据库系统在多个会话试图同时访问同一数据时的行为。这种情况的目标是为所有会话提供高效的访问，同时还要维护严格的数据完整性。每个数据库应用开发人员都应该熟悉本章讨论的话题。

## 13.1. 介绍

LightDB为开发者提供了一组丰富的工具来管理对数据的并发访问。在内部，数据一致性通过使用一种多版本模型（多版本并发控制，MVCC）来维护。这就意味着每个 SQL 语句看到的都只是一小段时间之前的数据快照（一个数据库版本），而不管底层数据的当前状态。这样可以保护语句不会看到可能由其他在相同数据行上执行更新的并发事务造成的不一致数据，为每一个数据库会话提供事务隔离。MVCC避免了传统的数据库系统的锁定方法，将锁争夺最小化来允许多用户环境中的合理性能。

使用MVCC并发控制模型而不是锁定的主要优点是在MVCC中，对查询（读）数据的锁请求与写数据的锁请求不冲突，所以读不会阻塞写，而写也从不阻塞读。甚至在通过使用革新的可序列化快照隔离（SSI）级别提供最严格的事务隔离级别时，LightDB也维持这个保证。

在LightDB里也有表和行级别的锁功能，用于那些通常不需要完整事务隔离并且想要显式管理特定冲突点的应用。不过，恰当地使用MVCC通常会提供比锁更好的性能。另外，由应用定义的咨询锁提供了一个获得不依赖于单一事务的锁的机制。

## 13.2. 事务隔离

SQL标准定义了四种隔离级别。最严格的是可序列化，在标准中用了一整段来定义它，其中说到一组可序列化事务的任意并发执行被保证效果和以某种顺序一个一个执行这些事务一样。其他三种级别使用并发事务之间交互产生的现象来定义，每一个级别中都要求必须不出现一种现象。注意由于可序列化的定义，在该级别上这些现象都不可能发生（这并不令人惊讶—如果事务的效果与每个时刻只运行一个的相同，你怎么可能看见由于交互产生的现象？）。

在各个级别上被禁止出现的现象是：

### 脏读

一个事务读取了另一个并行未提交事务写入的数据。

### 不可重复读

一个事务重新读取之前读取过的数据，发现该数据已经被另一个事务（在初始读之后提交）修改。

### 幻读

一个事务重新执行一个返回符合一个搜索条件的行集合的查询，发现满足条件的行集合因为另一个最近提交的事务而发生了改变。

### 序列化异常

成功提交一组事务的结果与这些事务所有可能的串行执行结果都不一致。

SQL 标准和 LightDB 实现的事务隔离级别在 Table 13.1 中描述。

Table 13.1. 事务隔离级别

隔离级别	脏读	不可重复读	幻读	序列化异常
读未提交	允许，但在 PG 中	可能	可能	可能

隔离级别	脏读	不可重复读	幻读	序列化异常
读已提交	不可能	可能	可能	可能
可重复读	不可能	不可能	允许，但不在 PG 中	可能
可序列化	不可能	不可能	不可能	不可能

在LightDB中，你可以请求四种标准事务隔离级别中的任意一种，但是内部只实现了三种不同的隔离级别，即 LightDB 的读未提交模式的行为和读已提交相同。这是因为把标准隔离级别映射到 LightDB 的多版本并发控制架构的唯一合理的方法。

该表格也显示 LightDB 的可重复读实现不允许幻读。而 SQL 标准允许更严格的行为：四种隔离级别只定义了哪种现象不能发生，但是没有定义哪种现象必须发生。可用的隔离级别的行为在下面的小节中详细描述。

要设置一个事务的事务隔离级别，使用SET TRANSACTION命令。

### Important

某些LightDB数据类型和函数关于事务的行为有特殊的规则。特别是，对一个序列的修改（以及用serial声明的一列的计数器）是立刻对所有其他事务可见的，并且在作出该修改的事务中断时也不会被回滚。  
见Section 9.1和Section 8.1.4

## 13.2.1. 读已提交隔离级别

读已提交是LightDB中的默认隔离级别。 当一个事务运行使用这个隔离级别时， 一个查询（没有FOR UPDATE/SHARE子句）只能看到查询开始之前已经被提交的数据， 而无法看到未提交的数据或在查询执行期间其它事务提交的数据。实际上，SELECT查询看到的是一个在查询开始运行的瞬间该数据库的一个快照。不过SELECT可以看见在它自身事务中之前执行的更新的效果，即使它们还没有被提交。还要注意的，即使在同一个事务里两个相邻的SELECT命令可能看到不同的数据， 因为其它事务可能会在第一个SELECT开始和第二个SELECT开始之间提交。

UPDATE、DELETE、SELECT FOR UPDATE和SELECT FOR SHARE命令在搜索目标行时的行为和SELECT一样： 它们将只找到在命令开始时已经被提交的行。 不过，在被找到时，这样的目标行可能已经被其它并发事务更新（或删除或锁住）。在这种情况下， 即将进行的更新将等待第一个更新事务提交或者回滚（如果它还在进行中）。 如果第一个更新事务回滚，那么它的作用将被忽略并且第二个事务可以继续更新最初发现的行。 如果第一个更新事务提交，若该行被第一个更新者删除，则第二个更新事务将忽略该行，否则第二个更新者将试图在该行的已被更新的版本上应用它的操作。该命令的搜索条件（WHERE子句）将被重新计算来看该行被更新的版本是否仍然符合搜索条件。如果符合，则第二个更新者使用该行的已更新版本继续其操作。在SELECT FOR UPDATE和SELECT FOR SHARE的情况下，这意味着把该行的已更新版本锁住并返回给客户端。

带有ON CONFLICT DO UPDATE子句的 INSERT行为类似。在读已提交模式，要插入的 每一行将被插入或者更新。除非有不相干的错误出现，这两种结果之一是肯定 会出现的。如果在另一个事务中发生冲突，并且其效果对于INSERT 还不可见，则UPDATE子句将会 影响那个行，即便那一行对于该命令来说没有惯常的可见版本。

带有ON CONFLICT DO NOTHING子句的 INSERT有可能因为另一个效果对 INSERT快照不可见的事务的结果无法让插入进行 下去。再一次，这只是读已提交模式中的情况。

因为上面的规则，正在更新的命令可能会看到一个不一致的快照： 它们可以看到并发更新命令在它尝试更新的相同行上的作用，但是却看不到那些命令对数据库里其它行的作用。这样的行为令读已提交模式不适合用于涉及复杂搜索条件的命令。不过，它对于更简单的情况是正确的。 例如，考虑用这样的命令更新银行余额：

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

如果两个这样的事务同时尝试修改帐号 12345 的余额，那我们很明显希望第二个事务从账户行的已更新版本上开始工作。因为每个命令只影响一个已经决定了的行，让它看到行的已更新版本不会导致任何麻烦的不一致性。

在读已提交模式中，更复杂的使用可能产生不符合需要的结果。例如：考虑一个在数据上操作的DELETE命令，它操作的数据正被另一个命令从它的限制条件中移除或者加入，例如，假定website是一个两行的表，两行的website.hits等于9和10：

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session: DELETE FROM website WHERE hits = 10;
COMMIT;
```

即便在UPDATE之前有一个website.hits = 10的行，DELETE将不会产生效果。这是因为更新之前的行值9被跳过，并且当UPDATE完成并且DELETE获得一个锁，新行值不再是10而是11，这再也不匹配条件了。

因为在读已提交模式中，每个命令都是从一个新的快照开始的，而这个快照包含在该时刻已提交的事务，因此同一事务中的后续命令将看到任何已提交的并行事务的效果。以上的焦点在于单个命令是否看到数据库的绝对一致的视图。

读已提交模式提供的部分事务隔离对于许多应用而言是足够的，并且这个模式速度快并且使用简单。不过，它不是对于所有情况都够用。做复杂查询和更新的应用可能需要比读已提交模式提供的更严格一致的数据库视图。

## 13.2.2. 可重复读隔离级别

可重复读隔离级别只看到在事务开始之前被提交的数据；它从来看不到未提交的数据或者并行事务在本事务执行期间提交的修改（不过，查询能够看见在它的事务中之前执行的更新，即使它们还没有被提交）。这是比SQL标准对此隔离级别所要求的更强的保证，并且阻止Table 13. 中描述的除了序列化异常之外的所有现象。如上面所提到的，这是标准特别允许的，标准只描述了每种隔离级别必须提供的最小保护。

这个级别与读已提交不同之处在于，一个可重复读事务中的查询可以看见在事务中第一个非事务控制语句开始时的一个快照，而不是事务中当前语句开始时的快照。因此，在一个单一事务中的后续SELECT命令看到的是相同的数据，即它们看不到其他事务在本事务启动后提交的修改。

使用这个级别的应用必须准备好由于序列化失败而重试事务。

UPDATE、DELETE、SELECT FOR UPDATE和SELECT FOR SHARE命令在搜索目标行时的行为和SELECT一样：它们将只找到在事务开始时已经被提交的行。不过，在被找到时，这样的目标行可能已经被其它并发事务更新（或删除或锁住）。在这种情况下，可重复读事务将等待第一个更新事务提交或者回滚（如果它还在进行中）。如果第一个更新事务回滚，那么它的作用将被忽略并且可重复读事务可以继续更新最初发现的行。但是如果第一个更新事务提交（并且实际更新或删除该行，而不是只锁住它），则可重复读事务将回滚并带有如下消息

```
ERROR: could not serialize access due to concurrent update
```

因为一个可重复读事务无法修改或者锁住被其他在可重复读事务开始之后的事务改变的行。

当一个应用接收到这个错误消息，它应该中断当前事务并且从开头重试整个事务。在第二次执行中，该事务将见到作为其初始数据库视图一部分的之前提交的改变，这样在使用行的新版本作为新事务更新的起点时就不会有逻辑冲突。

注意只有更新事务可能需要被重试；只读事务将永远不会有序列化冲突。

可重复读模式提供了一种严格的保证，在其中每一个事务看到数据库的一个完全稳定的视图。不过，这个视图并不需要总是和同一级别上并发事务的某些序列化（一次一个）执行保持一致。例如，即使这个级别上的一个只读事务可能看到一个控制记录被更新，这显示一个批处理已经被完成但是不能看见作为该批处理的逻辑组成部分的一个细节记录，因为它读取空值记录的一个较早的版本。如果不小心地使用显式锁来阻塞冲突事务，尝试运行在这个隔离级别的事务来强制业务规则不太可能正确地工作。

可重复读隔离级别是使用学术数据库文献和一些其他数据库产品中称为Snapshot Isolation的已知的技术来实现的。与使用传统锁技术并降低并发性的系统相比，可以观察到行为和性能方面的差异。一些其他系统甚至可以提供可重复读取和快照隔离作为具有不同行为的不同隔离级别。直到SQL标准开发出来之后，数据库研究人员才正式确定区分这两种技术的允许现象，并且超出了本手册的范围。全面的阐述，请参阅[berenson95]。

### Note

在LightDB版本 9.1 之前，一个对于可序列化事务隔离级别的请求会提供和这里描述的完全一样的行为。为了保持可序列化行为，现在应该请求可重复读。

## 13.2.3. 可序列化隔离级别

可序列化隔离级别提供了最严格的事务隔离。这个级别为所有已提交事务模拟序列事务执行；就好像事务被按照序列一个接着另一个被执行，而不是并行地执行。但是，和可重复读级别相似，使用这个级别的应用必须准备好因为序列化失败而重试事务。事实上，这个隔离级别完全像可重复读一样地工作，除了它会监视一些条件，这些条件可能导致一个可序列化事务的并发集合的执行产生的行为与这些事务所有可能的序列化（一次一个）执行不一致。这种监控不会引入超出可重复读之外的阻塞，但是监控会产生一些负荷，并且对那些可能导致序列化异常的条件的检测将触发一次序列化失败。

例如，考虑一个表mytab，它初始时包含：

class	value
1	10
1	20
2	100
2	200

假设可序列化事务 A 计算：

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

并且接着把结果（30）作为一个新行的value插入，新行的class = 2。同时，可序列化事务 B 计算：

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

并得到结果 300，它会将其与class = 1插入到一个新行中。然后两个事务都尝试提交。如果其中一个事务运行在可重复读隔离级别，两者都被允许提交；但是由于不同的执行顺序可能导致不同结果，使用可序列化事务将允许一个事务提交并且将回滚另一个并伴有这个消息：

```
ERROR: could not serialize access due to read/write dependencies among
transactions
```

这是因为，如果 A 在 B 之前执行，B 将计算得到合计值 330 而不是 300，而且相似地另一种顺序将导致 A 计算出一个不同的合计值。

当依赖可序列化事务来阻止异常时，重要的一点是任何从一个持久化用户表读出数据都不被认为是有效的，直到读它的事务已经成功提交为止。即便是对只读事务也是如此，除了在一个可推迟的只读事务中读取的数据是读出以后立刻有效的，因为这样的—个事务在开始读取任何数据之前会等待，直到它能获得一个快照保证来避免这种问题为止。在所有其他情况下，应用不能依靠在一个后来被中断的事务中读取的结果；相反，它们应当重试事务直到它成功。

要保证真正的可序列化，LightDB使用了谓词锁，这意味着它会保持锁，这些锁让它能够判断在它先运行的情况下，什么时候一个写操作会对一个并发事务中之前读取的结果产生影响。在LightDB中，这些锁并不导致任何阻塞，并且因此不会导致一个死锁。它们被用来标识和标志并发可序列化事务之间的依赖性，这些事务的组合可能导致序列化异常。相反，一个想要保证数据一致性的读已提交或可重复读事务可能需要拿走一个在整个表上的锁，这可能阻塞其他尝试使用该表的用户，或者它可能会使用不仅会阻塞其他事务还会导致磁盘访问的SELECT FOR UPDATE或SELECT FOR SHARE。

像大部分其他数据库系统，LightDB中的谓词锁基于被一个事务真正访问的数据。这些谓词锁将显示在pg\_locks系统视图中，它们的mode为SIReadLock。这种在一个查询执行期间获得的特别的锁将依赖于该查询所使用的计划，并且在事务过程中多个细粒度锁（如元组锁）可能和少量粗粒度锁（如页面锁）相结合来防止耗尽用于跟踪锁的内存。如果一个READ ONLY事务检测到不会有导致序列化异常的冲突发生，它可以在完成前释放其 SIRead 锁。事实上，READ ONLY事务将常常可以在启动时确立这一事实并避免拿到任何谓词锁。如果你显式地请求一个SERIALIZABLE READ ONLY DEFERRABLE事务，它将阻塞直到它能够确立这一事实（这是唯一一种可序列化事务阻塞但可重复读事务不阻塞的情况）。在另一方面，SIRead 锁常常需要被保持到事务提交之后，直到重叠的读写事务完成。

坚持使用可序列化事务可以简化开发。成功提交的并发可序列化事务的任意集合将得到和一次运行一个相同效果的这种保证意味着，如果你能证明一个单一事务在独自运行时能做正确的事情，则你可以相信它在任何混合的可序列化事务中也能做正确的事情，即使它不知道那些其他事务做了些什么，否则它将不会成功提交。重要的是使用这种技术的环境有一种普遍的方法来处理序列化失败（总是会返回一个 SQLSTATE 值 '40001'），因为它将很难准确地预计哪些事务可能为读/写依赖性做贡献并且需要被回滚来阻止序列化异常。读/写依赖性的监控会产生开销，如重启被序列化失败中止的事务，但是作为在该开销和显式锁及SELECT FOR UPDATE或SELECT FOR SHARE导致的阻塞之间的一种平衡，可序列化事务是在某些环境中最好性能的选择。

虽然LightDB的可序列化事务隔离级别只允许并发事务在能够证明有一种串行执行能够产生相同效果的前提下提交，但它却不能总是阻止在真正的串行执行中不会发生的错误产生。尤其是可能会看到由于可序列化事务重叠执行导致的唯一约束被违背的情况，这些情况即便在尝试插入键之前就显式地检查过该键不存在也会发生。避免这种问题的方法是，确保所有插入可能会冲突的键的可序列化事务首先显式地检查它们能不能那样做。例如，试想一个要求用户输入新键的应用，它会通过尝试查询用户给出的键来检查键是否已经存在，或者是通过选取现有最大的键并且加一来产生一个新键。如果某些可序列化事务不遵循这种协议而直接插入新键，则也可能会报告唯一约束被违背，即便在并发事务串行执行的情况下不会发生唯一约束被违背也是如此。

当依赖可序列化事务进行并发控制时，为了最佳性能应该考虑一下问题：

- 在可能时声明事务为READ ONLY。
- 控制活动连接的数量，如果需要使用一个连接池。这总是一个重要的性能考虑，但是在一个使用可序列化事务的繁忙系统中这尤为重要。
- 只在一个单一事务中放完整性目的所需要的东西。

- 不要让连接不必要地“闲置在事务中”。配置参数 `idle_in_transaction_session_timeout` 可以被用来自动断开拖延会话的连接。
- 在那些由于使用可序列化事务自动提供的保护的地方消除不再需要的显式锁、`SELECT FOR UPDATE`和`SELECT FOR SHARE`。
- 当系统因为谓词锁表内存短缺而被强制结合多个页面级谓词锁为一个单一的关系级谓词锁时，序列化失败的比例可能会上升。你可以通过增加 `max_pred_locks_per_transaction`、`max_pred_locks_per_relation`和`max_pred_locks_per_page`来避免这种情况。
- 一次顺序扫描将总是需要一个关系级谓词锁。这可能导致序列化失败的比例上升。通过缩减 `random_page_cost`和/或增加 `cpu_tuple_cost`来鼓励使用索引扫描将有助于此。一定要在事务回滚和重启数目的任何减少与查询执行时间的任何全面改变之间进行权衡。

可序列化隔离级别是使用学术数据库文献中称为可序列化快照隔离的技术实现的，通过添加序列化异常事务的检查的方式构建在快照隔离的基础之上。与使用传统锁技术的其他系统相比，可以观察到行为和性能方面的一些差异。详细信息请参阅[ports12]。

## 13.3. 显式锁定

LightDB提供了多种锁模式用于控制对表中数据的并发访问。这些模式可以用于在MVCC无法给出期望行为的情境中由应用控制的锁。同样，大多数LightDB命令会自动要求恰当的锁以保证被引用的表在命令的执行过程中不会以一种不兼容的方式删除或修改（例如，`TRUNCATE`无法安全地与同一表中的其他操作并发地执行，因此它在表上获得一个`ACCESS EXCLUSIVE`锁来强制这种行为）。

要检查在一个数据库服务器中当前未解除的锁列表，可以使用`pg_locks`系统视图。有关监控锁管理器子系统状态的更多信息，请参考Chapter 25

### 13.3.1. 表级锁

下面的列表显示了可用的锁模式和LightDB自动使用它们的场合。你也可以用`LOCK`命令显式获得这些锁。请记住所有这些锁模式都是表级锁，即使它们的名字包含“row”单词（这些名称是历史遗产）。在一定程度上，这些名字反应了每种锁模式的典型用法——但是语意却都是一样的。两种锁模式之间真正的区别是它们有着不同的冲突锁模式集合（参考Table 13.2）。两个事务在同一时刻不能在同一个表上持有属于相互冲突模式的锁（但是，一个事务决不会和自身冲突。例如，它可以在同一个表上获得`ACCESS EXCLUSIVE`锁然后接着获取`ACCESS SHARE`锁）。非冲突锁模式可以由许多事务同时持有。请特别注意有些锁模式是自冲突的（例如，在一个时刻`ACCESS EXCLUSIVE`锁不能被多于一个事务持有）而其他锁模式不是自冲突的（例如，`ACCESS SHARE`锁可以被多个事务持有）。

#### 表级锁模式

##### ACCESS SHARE

只与`ACCESS EXCLUSIVE`锁模式冲突。

`SELECT`命令在被引用的表上获得一个这种模式的锁。通常，任何只读取表而不修改它的查询都将获得这种锁模式。

##### ROW SHARE

与`EXCLUSIVE`和`ACCESS EXCLUSIVE`锁模式冲突。

`SELECT FOR UPDATE`和`SELECT FOR SHARE`命令在目标表上取得一个这种模式的锁（加上在被引用但没有选择`FOR UPDATE`/`FOR SHARE`的任何其他表上的`ACCESS SHARE`锁）。



## ROW EXCLUSIVE

与SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。

命令UPDATE、DELETE和INSERT在目标表上取得这种锁模式（加上在任何其他被引用表上的ACCESS SHARE锁）。通常，这种锁模式将被任何修改表中数据的命令取得。

## SHARE UPDATE EXCLUSIVE

与SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式保护一个表不受并发模式改变和VACUUM运行的影响。

由VACUUM（不带FULL）、ANALYZE、CREATE INDEX CONCURRENTLY、REINDEX CONCURRENTLY、CREATE STATISTICS以及某些ALTER INDEX 和 ALTER TABLE的变体获得（详细内容请参考这些命令的文档）。

## SHARE

与ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式保护一个表不受并发数据改变的影响。

由CREATE INDEX（不带CONCURRENTLY）取得。

## SHARE ROW EXCLUSIVE

与ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式保护一个表不受并发数据修改所影响，并且是自排他的，这样在一个时刻只能有一个会话持有它。

由CREATE TRIGGER和某些形式的 ALTER TABLE所获得。

## EXCLUSIVE

与ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式只允许并发的ACCESS SHARE锁，即只有来自于表的读操作可以与一个持有该锁模式的事务并行处理。

由REFRESH MATERIALIZED VIEW CONCURRENTLY获得。

## ACCESS EXCLUSIVE

与所有模式的锁冲突（ACCESS SHARE、ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE）。这种模式保证持有者是访问该表的唯一事务。

由ALTER TABLE、DROP TABLE、TRUNCATE、REINDEX、CLUSTER、VACUUM FULL和REFRESH MATERIALIZED VIEW（不带CONCURRENTLY）命令获取。很多形式的ALTER INDEX和ALTER TABLE也在这个层面上获得锁（见ALTER TABLE）。这也是未显式指定模式的LOCK TABLE命令的默认锁模式。

### Tip

只有一个ACCESS EXCLUSIVE锁阻塞一个SELECT（不带FOR UPDATE/SHARE）语句。

一旦被获取，一个锁通常将被持有直到事务结束。但是在建立保存点之后才获得锁，那么在回滚到这个保存点的时候将立即释放该锁。这与ROLLBACK取消保存点之后所有的影响的原则保持一致。同样的原则也适用于在PL/pgSQL异常块中获得的锁：一个跳出块的错误将释放在块中获得的锁。

Table 13.2. 冲突的锁模式

请求的锁模式	已存在的锁模式							
	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	ACCESS EXCL.
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCL.					X	X	X	X
SHARE UPDATE EXCL.				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCL.			X	X	X	X	X	X
EXCL.		X	X	X	X	X	X	X
ACCESS EXCL.	X	X	X	X	X	X	X	X

### 13.3.2. 行级锁

除了表级锁以外，还有行级锁，在下文列出了行级锁以及在哪些情境下LightDB会自动使用它们。行级锁的完整冲突表请见Table 13.3 注意一个事务可能会在相同的行上保持冲突的锁，甚至是在不同的子事务中。但是除此之外，两个事务永远不可能在相同的行上持有冲突的锁。行级锁不影响数据查询，它们只阻塞对同一行的写入者和加锁者。行级锁在事务结束时或保存点回滚的时候释放，就像表级锁一样。

#### 行级锁模式

##### FOR UPDATE

FOR UPDATE会导致由SELECT语句检索到的行被锁定，就好像它们要被更新。这可以阻止它们被其他事务锁定、修改或者删除，一直到当前事务结束。也就是说其他尝试UPDATE、DELETE、SELECT FOR UPDATE、SELECT FOR NO KEY UPDATE、SELECT FOR SHARE或者SELECT FOR KEY SHARE这些行的事务将被阻塞，直到当前事务结束。反过来，SELECT FOR UPDATE将等待已经在相同行上运行以上这些命令的并发事务，并且接着锁定并且返回被更新的行（或者没有行，因为行可能已被删除）。不过，在一个REPEATABLE READ或SERIALIZABLE事务中，如果一个要被锁定的行在事务开始后被更改，将会抛出一个错误。进一步的讨论请见Section 13.4

任何在一行上的DELETE命令也会获得FOR UPDATE锁模式，以及修改某些列的值的UPDATE也会获得该锁模式。当前UPDATE情况中被考虑的列集合是那些具有能用于外键的唯一索引的列（所以部分索引和表达式索引不被考虑），但是这种要求未来有可能会改变。

##### FOR NO KEY UPDATE

行为与FOR UPDATE类似，不过获得的锁较弱：这种锁将不会阻塞尝试在相同行上获得锁的SELECT FOR KEY SHARE命令。任何不获取FOR UPDATE锁的UPDATE也会获得这种锁模式。

##### FOR SHARE

行为与FOR NO KEY UPDATE类似，不过它在每个检索到的行上获得一个共享锁而不是排他锁。一个共享锁会阻塞其他事务在这些行上执行UPDATE、DELETE、SELECT FOR UPDATE或

者SELECT FOR NO KEY UPDATE，但是它不会阻止它们执行SELECT FOR SHARE或者SELECT FOR KEY SHARE。

FOR KEY SHARE

行为与FOR SHARE类似，不过锁较弱：SELECT FOR UPDATE会被阻塞，但是SELECT FOR NO KEY UPDATE不会被阻塞。一个键共享锁会阻塞其他事务执行修改键值的DELETE或者UPDATE，但不会阻塞其他UPDATE，也不会阻止SELECT FOR NO KEY UPDATE、SELECT FOR SHARE或者SELECT FOR KEY SHARE。

LightDB不会在内存里保存任何关于已修改行的信息，因此对一次锁定的行数没有限制。不过，锁住一行会导致一次磁盘写，例如，SELECT FOR UPDATE将修改选中的行以标记它们被锁住，并且因此会导致磁盘写入。

Table 13.3. 冲突的行级锁

要求的锁模式	当前的锁模式			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

### 13.3.3. 页级锁

除了表级别和行级别的锁以外，页面级别的共享/排他锁被用来控制对共享缓冲池中表页面的读/写。这些锁在行被抓取或者更新后马上被释放。应用开发者通常不需要关心页级锁，我们在这里提到它们只是为了完整。

### 13.3.4. 死锁

显式锁定的使用可能会增加死锁的可能性，死锁是指两个（或多个）事务相互持有对方想要的锁。例如，如果事务 1 在表 A 上获得一个排他锁，同时试图获取一个在表 B 上的排他锁，而事务 2 已经持有表 B 的排他锁，同时却正在请求表 A 上的一个排他锁，那么两个事务就都不能进行下去。LightDB能够自动检测到死锁情况并且会通过中断其中一个事务从而允许其它事务完成来解决这个问题（具体哪个事务会被中断是很难预测的，而且也不应该依靠这样的预测）。

要注意死锁也可能作为行级锁的结果而发生（并且因此，它们即使在没有使用显式锁定的情况下也会发生）。考虑如下情况，两个并发事务在修改一个表。第一个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

这样就在指定帐号的行上获得了一个行级锁。然后，第二个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

第一个UPDATE语句成功地在指定行上获得了一个行级锁，因此它成功更新了该行。但是第二个UPDATE语句发现它试图更新的行已经被锁住了，因此它等待持有该锁的事务结束。事务二现在就在等待事务一结束，然后再继续执行。现在，事务一执行：

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

事务一试图在指定行上获得一个行级锁，但是它得不到：事务二已经持有了这样的锁。所以它要等待事务二完成。因此，事务一被事务二阻塞，而事务二也被事务一阻塞：一个死锁。LightDB将检测这样的情况并中断其中一个事务。

防止死锁的最好方法通常是保证所有使用一个数据库的应用都以一致的顺序在多个对象上获得锁。在上面的例子里，如果两个事务以同样的顺序更新那些行，那么就不会发生死锁。我们也应该保证一个事务中在一个对象上获得的第一个锁是该对象需要的最严格的锁模式。如果我们无法提前验证这些，那么可以通过重试因死锁而中断的事务来即时处理死锁。

只要没有检测到死锁情况，寻求一个表级或行级锁的事务将无限等待冲突锁被释放。这意味着一个应用长时间保持事务开启不是什么好事（例如等待用户输入）。

### 13.3.5. 咨询锁

LightDB提供了一种方法创建由应用定义其含义的锁。这种锁被称为咨询锁，因为系统并不强迫其使用 — 而是由应用来保证其正确的使用。咨询锁可用于 MVCC 模型不适用的锁定策略。例如，咨询锁的一种常用用法是模拟所谓“平面文件”数据管理系统典型的悲观锁策略。虽然一个存储在表中的标志可以被用于相同目的，但咨询锁更快、可以避免表膨胀并且会由服务器在会话结束时自动清理。

有两种方法在LightDB中获取一个咨询锁：在会话级别或在事务级别。一旦在会话级别获得了咨询锁，它将被保持直到被显式释放或会话结束。不同于标准锁请求，会话级咨询锁请求不尊重事务语义：在一个后来被回滚的事务中得到的锁在回滚后仍然被保持，并且同样即使调用它的事务后来失败一个解锁也是有效的。一个锁在它所属的进程中可以被获取多次；对于每一个完成的锁请求必须有一个相应的解锁请求，直至锁被真正释放。在另一方面，事务级锁请求的行为更像普通锁请求：在事务结束时会自动释放它们，并且没有显式的解锁操作。这种行为通常比会话级别的行为更方便，因为它使用一个咨询锁的时间更短。对于同一咨询锁标识符的会话级别和事务级别的锁请求按照期望将彼此阻塞。如果一个会话已经持有了一个给定的咨询锁，由它发出的附加请求将总是成功，即使有其他会话在等待该锁；不管现有的锁和新请求是处在会话级别还是事务级别，这种说法都是真的。

和所有LightDB中的锁一样，当前被任何会话所持有的咨询锁的完整列表可以在pg\_locks系统视图中找到。

咨询锁和普通锁都被存储在一个共享内存池中，它的尺寸由max\_locks\_per\_transaction和max\_connections配置变量定义。必须当心不要耗尽这些内存，否则服务器将不能再授予任何锁。这对服务器可以授予的咨询锁数量设置了一个上限，根据服务器的配置不同，这个限制通常是数万到数十万。

在使用咨询锁方法的特定情况下，特别是查询中涉及显式排序和LIMIT子句时，由于 SQL 表达式被计算的顺序，必须小心控制锁的获取。例如：

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger!
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

在上述查询中，第二种形式是危险的，因为不能保证在锁定函数被执行之前应用LIMIT。这可能导致获得某些应用不期望的锁，并因此在会话结束之前无法释放。从应用的角度来看，这样的锁将被挂起，虽然它们仍然在pg\_locks中可见。

提供的操作咨询锁函数在Section 9.27.10中描述。

## 13.4. 应用级别的数据完整性检查

对于使用读已提交事务的数据完整性强制业务规则非常困难，因为对每一个语句数据视图都在变化，并且如果一个写冲突发生即使一个单一语句也不能把它自己限制到该语句的快照。

虽然一个可重复读事务在其执行期间有一个稳定的数据视图，在使用MVCC快照进行数据一致性检查时也有一个小问题，它涉及到被称为读/写冲突的东西。如果一个事务写数据并且一个并发事务尝试读相同的数据（不管是在写之前还是之后），它不能看到其他事务的工作。读取事务看起来是第一个执行的，不管哪个是第一个启动或者哪个是第一个提交。如果就此为止，则没有问题，但是如果读取者也写入被一个并发事务读取的数据，现在有一个事务好像是已经在前面提到的任何一个事务之前运行。如果看起来最后执行的事务实际上第一个提交，在这些事务的执行顺序图中很容易出现一个环。当这样一个环出现时，完整性检查在没有任何帮助的情况下将不会正确地工作。

正如Section 13.2.3中提到的，可序列化事务仅仅是可重复读事务增加了对读/写冲突的危险模式的非阻塞监控。当检测到一个可能导致表面的执行顺序中产生环的模式，涉及到的一个事务将被回滚来打破该环。

### 13.4.1. 用可序列化事务来强制一致性

如果可序列化事务隔离级别被用于所有需要一个一致数据视图的写入和读取，不需要其他的工作来保证一致性。在LightDB中，来自于其他环境的被编写成使用可序列化事务来保证一致性的软件应该“只工作”在这一点上。

当使用这种技术时，如果应用软件通过一个框架来自动重试由于序列化错误而回滚的事务，它将避免为应用程序员带来不必要的负担。把`default_transaction_isolation`设置为`serializable`可能是个好主意。通过触发器中的事务隔离级别检查来采取某些动作来保证没有其他事务隔离级别被使用（由于疏忽或者为了破坏完整性检查）也是明智的。

性能建议见Section 13.2.3

#### Warning

这个级别的使用可序列化事务的完整性保护还没有扩展到热备份模式（Section 24.5）。由于这个原因，那些使用热备份的系统可能想要在主数据库上使用可重复读和显式锁定。

### 13.4.2. 使用显式锁定强制一致性

当可以使用非可序列化写时，要保证一行的当前有效性并保护它不受并发更新的影响，我们必须使用`SELECT FOR UPDATE`、`SELECT FOR SHARE`或一个合适的`LOCK TABLE`语句（`SELECT FOR UPDATE`和`SELECT FOR SHARE`锁只针对并发更新返回行，而`LOCK TABLE`会锁住整个表）。当从其他环境移植应用到LightDB时需要考虑这些。

关于这些来自其他环境的转换还需要注意的是`SELECT FOR UPDATE`不保证一个并发事务将不会更新或删除一个被选中的行。要在LightDB中这样做，你必须真正地更新该行，即便没有值需要被改变。`SELECT FOR UPDATE`临时阻塞其他事务，让它们不能获取该相同的锁或者执行一个会影响被锁定行的`UPDATE`或`DELETE`，但是一旦正持有该所锁的事务提交或回滚，一个被阻塞的事务将继续执行冲突操作，除非当锁被持有时一个该行的实际`UPDATE`被执行。

在非可序列化MVCC环境下，全局有效性检查需要一些额外的考虑。例如，一个银行应用可能会希望检查一个表中的所有扣款总和等于另外一个表中的收款总和，同时两个表还会被更新。比较两个连续的在读已提交模式下不会可靠工作的`SELECT sum(...)`命令，因为第二个查询很可能会包含没有被第一个查询考虑的事务提交的结果。在一个单一的可重复读事务里进行两个求和则给出在可串行化事务开始之前提交的所有事务产生的准确结果——但有人可能会合理地质疑在结果被递交的时候，它们是否仍然相关。如果可重复读事务本身在尝试做一致性检查之前应用了某些变更，那么检查的有用性就更加值得讨论了，因为现在它包含了一些（但不是全部）事务开始后的变化。在这种情况下，一个小心的人可能希望锁住所有需要检查的表，这样才能获得一个无可置疑的当前现状的图像。一个`SHARE`模式（或者更高）的锁保证在被锁定表中除了当前事务所作的更改之外，没有未提交的更改。

还要注意如果某人正在依赖显式锁定来避免并发更改，那么他应该使用读已提交模式，或者是在可重复读模式里在执行命令之前小心地获取锁。在可重复读事务里获取的锁保证了不会有其它修改该表的事务正在运行，但是如果事务看到的快照在获取锁之前，那么它可能早于表中一些现在已经提交的更改。一个可重复读事务的快照实际上是在它的第一个查询或者数据修改命令（SELECT、INSERT、UPDATE或DELETE）开始的时候冻结的，因此我们可以在快照冻结之前显式地获取锁。

## 13.5. 提醒

一些 DDL 命令（当前只有TRUNCATE和ALTER TABLE的表重写形式）不是 MVCC 安全的。这意味着在截断或者重写提交之后，该表将对并发事务（如果它们使用的快照是在 DDL 命令提交前取得的）呈现出空表的形态。这只对没有在该 DDL 命令开始前访问所讨论的表的事务存在问题 — 任何在 DDL 命令开始前访问过该表的事务将持有至少一个 ACCESS SHARE 表锁，这将阻塞该 DDL 命令直到该事务完成。因此这些命令对于目标表上的连续查询将不会造成任何明显的表内容不一致，但是它们可能导致目标表内容和数据库中其他表内容之间的一致。

对于可序列化事务隔离级别的支持还没有被加入到热备复制目标（在Section 24.5中描述）中。当前在热备模式中支持的最严格的隔离级别是可重复读。虽然在主数据库上用可序列化事务执行所有持久化数据库写入将确保所有备用数据库将最终达到一个一致的状态，但是运行在备用数据库上的一个可重复读事务有时可能会看到一个短暂的、与主数据库上事务的任何串行执行都不一致的状态。

系统目录的内部访问不是使用当前事务的隔离级别来完成的。这意味着新创建的数据库对象，例如表，对于并发的可重复读和可序列化事务是可见的，即使它们包含的行并非如此。相比之下，显式检查系统目录的查询不会看到表示并发创建的数据库对象的行，在更高的隔离级别中。

## 13.6. 锁定和索引

尽管LightDB提供对表数据访问的非阻塞读/写，但并非LightDB中实现的每一个索引访问方法当前都能够提供非阻塞读/写访问。不同的索引类型按照下面方法操作：

### B-tree、GiST索引

短期的页面级共享/排他锁被用于读/写访问。每个索引行被取得或被插入后立即释放锁。这些索引类型提供了无死锁情况的最高并发性。

### Hash索引

Hash 桶级别的共享/排他锁被用于读/写访问。锁在整个 Hash 桶处理完成后释放。Hash 桶级锁比索引级的锁提供了更好的并发性但是可能产生死锁，因为锁持有的时间比一次索引操作时间长。

### GIN索引

短期的页面级共享/排他锁被用于读/写访问。锁在索引行被插入/抓取后立即释放。但要注意的是一个 GIN 索引值的插入通常导致对每行产生几个索引键的插入，因此 GIN 可能为了插入一个单一值而做大量的工作。

目前，B-tree 索引为并发应用提供了最好的性能。因为它还有比 Hash 索引更多的特性，在那些需要对标量数据进行索引的并发应用中，我们建议使用 B-tree 索引类型。在处理非标量类型数据的时候，B-tree 就没什么用了，应该使用 GiST、SP-GiST 或 GIN 索引替代。

---

# Chapter 14. 性能提示

查询性能可能受多种因素影响。其中一些因素可以由用户控制，而其它的则属于系统下层设计的基本原理。本章我们提供一些有关理解和调节LightDB性能提示。

## 14.1. 使用XPLAIN

LightDB为每个收到查询产生一个查询计划。选择正确的计划来匹配查询结构和数据的属性对于好的性能来说绝对是最关键的，因此系统包含了一个复杂的规划器来尝试选择好的计划。你可以使用EXPLAIN命令察看规划器为任何查询生成的查询计划。阅读查询计划是一门艺术，它要求一些经验来掌握，但是本节只试图覆盖一些基础。

本节中的例子都是从 9.3 开发源代码的回归测试数据库中抽取出来的，并且在此之前做过一次VACUUM ANALYZE。你应该能够在自己尝试这些例子时得到相似的结果，但是你的估计代价和行计数可能会小幅变化，因为ANALYZE的统计信息是随机采样而不是精确值，并且代价也与平台有某种程度的相关性。

这些例子使用EXPLAIN的默认“text”输出格式，这种格式紧凑并且便于人类阅读。如果你想把EXPLAIN的输出交给一个程序做进一步分析，你应该使用它的某种机器可读的输出格式（XML、JSON 或 YAML）。

### 14.1.1. EXPLAIN基础

查询计划的结构是一个计划结点的树。最底层的结点是扫描结点：它们从表中返回未经处理的行。不同的表访问模式有不同的扫描结点类型：顺序扫描、索引扫描、位图索引扫描。也还有不是表的行来源，例如VALUES子句和FROM中返回集合的函数，它们有自己的结点类型。如果查询需要连接、聚集、排序、或者在未经处理的行上的其它操作，那么就会在扫描结点之上有其它额外的结点来执行这些操作。并且，做这些操作通常都有多种方法，因此在这些位置也有可能出现不同的结点类型。EXPLAIN给计划树中每个结点都输出一行，显示基本的结点类型和计划器为该计划结点的执行所做的开销估计。第一行（最上层的结点）是对该计划的总执行开销的估计；计划器试图最小化的就是这个数字。

这里是一个简单的例子，只是用来显示输出看起来是什么样的：

```
EXPLAIN SELECT * FROM tenk1;
```

```
QUERY PLAN
```

```
-----  
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)
```

由于这个查询没有WHERE子句，它必须扫描表中的所有行，因此计划器只能选择使用一个简单的顺序扫描计划。被包含在圆括号中的数字是（从左至右）：

- 估计的启动开销。在输出阶段可以开始之前消耗的时间，例如在一个排序结点里执行排序的时间。
- 估计的总开销。这个估计值基于的假设是计划结点会被运行到完成，即所有可用的行都被检索。不过实际上一个结点的父结点可能很快停止读所有可用的行（见下面的LIMIT例子）。
- 这个计划结点输出行数的估计值。同样，也假定该结点能运行到完成。
- 预计这个计划结点输出的行平均宽度（以字节计算）。

开销是用规划器的开销参数（参见Section 17.7.2 所决定的捏造单位来衡量的。传统上以取磁盘页面为单位来度量开销；也就是seq\_page\_cost将被按照习惯设为1.0，其它开销参数将相对于它来设置。本节的例子都假定这些参数使用默认值。

有一点很重要：一个上层结点的开销包括它的所有子结点的开销。还有一点也很重要：这个开销只反映规划器关心的东西。特别是这个开销没有考虑结果行传递给客户端所花费的时间，这个时间可能是实际花费时间中的一个重要因素；但是它被规划器忽略了，因为它无法通过修改计划来改变（我们相信，每个正确的计划都将输出同样的行集）。

行数值有一些小技巧，因为它不是计划结点处理或扫描过的行数，而是该结点发出的行数。这通常比被扫描的行数少一些，因为有些被扫描的行会被应用于此结点上的任意WHERE子句条件过滤掉。理想中顶层的行估计会接近于查询实际返回、更新、删除的行数。

回到我们的例子：

```
EXPLAIN SELECT * FROM tenk1;
```

QUERY PLAN

```
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)
```

这些数字的产生非常直接。如果你执行：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

你会发现tenk1有358个磁盘页面和10000行。开销被计算为（页面读取数 \* seq\_page\_cost） + （扫描的行数 \* cpu\_tuple\_cost）。默认情况下，seq\_page\_cost是1.0，cpu\_tuple\_cost是0.01，因此估计的开销是  $(358 * 1.0) + (10000 * 0.01) = 458$ 。

现在让我们修改查询并增加一个WHERE条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;
```

QUERY PLAN

```
Seq Scan on tenk1 (cost=0.00..483.00 rows=7001 width=244)
  Filter: (unique1 < 7000)
```

请注意EXPLAIN输出显示WHERE子句被当做一个“过滤器”条件附加到顺序扫描计划结点。这意味着该计划结点为它扫描的每一行检查该条件，并且只输出通过该条件的行。因为WHERE子句的存在，估计的输出行数降低了。不过，扫描仍将必须访问所有 10000 行，因此开销没有被降低；实际上开销还有所上升（准确来说，上升了  $10000 * \text{cpu\_operator\_cost}$ ）以反映检查WHERE条件所花费的额外 CPU 时间。

这条查询实际选择的行数是 7000，但是估计的rows只是个近似值。如果你尝试重复这个试验，那么你很可能得到略有不同的估计。此外，这个估计会在每次ANALYZE命令之后改变，因为ANALYZE生成的统计数据是从该表中随机采样计算的。

现在，让我们把条件变得更严格：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;
```

QUERY PLAN

```
Bitmap Heap Scan on tenk1 (cost=5.07..229.20 rows=101 width=244)
  Recheck Cond: (unique1 < 100)
  -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=101 width=0)
      Index Cond: (unique1 < 100)
```

这里，规划器决定使用一个两步的计划：子计划结点访问一个索引来找出匹配索引条件的行的位置，然后上层计划结点实际地从表中取出那些行。独立地抓取行比顺序地读取它们的开



销高很多，但是不是所有的表页面都被访问，这么做实际上仍然比一次顺序扫描开销要少（使用两层计划的原因是因为上层规划结点把索引标识出来的行位置在读取之前按照物理位置排序，这样可以最小化单独抓取的开销。结点名称里面提到的“位图”是执行该排序的机制）。

现在让我们给WHERE子句增加另一个条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringul = 'xxx';
```

QUERY PLAN

```
-----
Bitmap Heap Scan on tenk1 (cost=5.04..229.43 rows=1 width=244)
  Recheck Cond: (unique1 < 100)
  Filter: (stringul = 'xxx'::name)
  -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=101 width=0)
      Index Cond: (unique1 < 100)
```

新增的条件stringul = 'xxx'减少了估计的输出行数，但是没有减少开销，因为我们仍然需要访问相同的行集合。请注意，stringul子句不能被应用为一个索引条件，因为这个索引只是在unique1列上。它被用来过滤从索引中检索出的行。因此开销实际上略微增加了一些以反映这个额外的检查。

在某些情况下规划器将更倾向于一个“simple”索引扫描计划：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;
```

QUERY PLAN

```
-----
Index Scan using tenk1_unique1 on tenk1 (cost=0.29..8.30 rows=1 width=244)
  Index Cond: (unique1 = 42)
```

在这类计划中，表行被按照索引顺序取得，这使得读取它们开销更高，但是其中有一些是对行位置排序的额外开销。你很多时候将在只取得一个单一行的查询中看到这种计划类型。它也经常被用于拥有匹配索引顺序的ORDER BY子句的查询中，因为那样就不需要额外的排序步骤来满足ORDER BY。在此示例中，添加ORDER BY unique1将使用相同的计划，因为索引已经隐式提供了请求的排序。

规划器可以通过多种方式实现ORDER BY子句。上面的例子表明，这样的排序子句可以隐式实现。规划器还可以添加一个明确的sort步骤：

```
EXPLAIN SELECT * FROM tenk1 ORDER BY unique1;
```

QUERY PLAN

```
-----
Sort (cost=1109.39..1134.39 rows=10000 width=244)
  Sort Key: unique1
  -> Seq Scan on tenk1 (cost=0.00..445.00 rows=10000 width=244)
```

如果计划的一部分保证对所需排序键的前缀进行排序，那么计划器可能会决定使用incremental sort步骤：

```
EXPLAIN SELECT * FROM tenk1 ORDER BY four, ten LIMIT 100;
```

QUERY PLAN

```
-----
Limit (cost=521.06..538.05 rows=100 width=244)
  -> Incremental Sort (cost=521.06..2220.95 rows=10000 width=244)
```

```

Sort Key: four, ten
Presorted Key: four
-> Index Scan using index_tenk1_on_four on tenk1 (cost=0.29..1510.08
rows=10000 width=244)

```

与常规排序相比，增量排序允许在对整个结果集进行排序之前返回元组，这尤其可以使用LIMIT查询进行优化。它还可以减少内存使用和将排序溢出到磁盘的可能性，但其代价是将结果集拆分为多个排序批次的开销增加。

如果在WHERE引用的多个行上有独立的索引，规划器可能会选择使用这些索引的一个 AND 或 OR 组合：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;
```

QUERY PLAN

```

-----
Bitmap Heap Scan on tenk1 (cost=25.08..60.21 rows=10 width=244)
  Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
  -> BitmapAnd (cost=25.08..25.08 rows=10 width=0)
      -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=101
width=0)
          Index Cond: (unique1 < 100)
      -> Bitmap Index Scan on tenk1_unique2 (cost=0.00..19.78 rows=999
width=0)
          Index Cond: (unique2 > 9000)

```

但是这要求访问两个索引，所以与只使用一个索引并把其他条件作为过滤器相比，它不一定能胜出。如果你变动涉及到的范围，你将看到计划也会相应改变。

下面是一个例子，它展示了LIMIT的效果：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;
```

QUERY PLAN

```

-----
Limit (cost=0.29..14.48 rows=2 width=244)
  -> Index Scan using tenk1_unique2 on tenk1 (cost=0.29..71.27 rows=10
width=244)
      Index Cond: (unique2 > 9000)
      Filter: (unique1 < 100)

```

这是和上面相同的查询，但是我们增加了一个LIMIT这样不是所有的行都需要被检索，并且规划器改变了它的决定。注意索引扫描结点的总开销和行计数显示出好像它会被运行到完成。但是，限制结点在检索到这些行的五分之一后就会停止，因此它的总开销只是索引扫描结点的五分之一，并且这是查询的实际估计开销。之所以用这个计划而不是在之前的计划上增加一个限制结点是因为限制无法避免在位图扫描上花费启动开销，因此总开销会是超过那种方法（25个单位）的某个值。

让我们尝试连接两个表，使用我们已经讨论过的列：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;
```

## QUERY PLAN

```

-----
Nested Loop (cost=4.65..118.62 rows=10 width=488)
  -> Bitmap Heap Scan on tenk1 t1 (cost=4.36..39.47 rows=10 width=244)
      Recheck Cond: (unique1 < 10)
      -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.36 rows=10
width=0)
          Index Cond: (unique1 < 10)
  -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.29..7.91 rows=1
width=244)
      Index Cond: (unique2 = t1.unique2)

```

在这个计划中，我们有一个嵌套循环连接结点，它有两个表扫描作为输入或子结点。该结点的摘要行的缩进反映了计划树的结构。连接的第一个（或“outer”）子结点是一个与前面见到的相似的位图扫描。它的开销和行计数与我们从SELECT ... WHERE unique1 < 10得到的相同，因为我们将WHERE子句unique1 < 10用在了那个结点上。t1.unique2 = t2.unique2子句现在还不相关，因此它不影响 outer 扫描的行计数。嵌套循环连接结点将为从 outer 子结点得到的每一行运行它的第二个（或“inner”）子结点。当前 outer 行的列值可以被插入 inner 扫描。这里，来自 outer 行的t1.unique2值是可用的，所以我们得到的计划和开销与前面见到的简单SELECT ... WHERE t2.unique2 = constant情况相似（估计的开销实际上比前面看到的略低，是因为在t2上的重复索引扫描会利用到高速缓存）。循环结点的开销则被以 outer 扫描的开销为基础设置，外加对每一个 outer 行都要进行一次 inner 扫描（10 \* 7.87），再加上用于连接处理一点 CPU 时间。

在这个例子里，连接的输出行计数等于两个扫描的行计数的乘积，但通常并不是所有的情况中都如此，因为可能有同时提及两个表的 额外WHERE子句，并且因此它只能被应用于连接点，而不能影响任何一个输入扫描。这里是一个例子：

```

EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

```

## QUERY PLAN

```

-----
Nested Loop (cost=4.65..49.46 rows=33 width=488)
  Join Filter: (t1.hundred < t2.hundred)
  -> Bitmap Heap Scan on tenk1 t1 (cost=4.36..39.47 rows=10 width=244)
      Recheck Cond: (unique1 < 10)
      -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.36 rows=10
width=0)
          Index Cond: (unique1 < 10)
  -> Materialize (cost=0.29..8.51 rows=10 width=244)
      -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.29..8.46
rows=10 width=244)
          Index Cond: (unique2 < 10)

```

条件t1.hundred < t2.hundred不能在tenk2\_unique2索引中被测试，因此它被应用在连接结点。这缩减了连接结点的估计输出行计数，但是没有改变任何输入扫描。

注意这里规划器选择了“物化”连接的 inner 关系，方法是在它的上方放了一个物化计划结点。这意味着t2索引扫描将只被做一次，即使嵌套循环连接结点需要读取其数据十次（每个来自 outer 关系的行都要读一次）。物化结点在读取数据时将它保存在内存中，然后在每一次后续执行时从内存返回数据。

在处理外连接时，你可能会看到连接计划结点同时附加有“连接过滤器”和普通“过滤器”条件。连接过滤器条件来自于外连接的ON子句，因此一个无法通过连接过滤器条件的行

也能够作为一个空值扩展的行被发出。但是一个普通过滤器条件被应用在外连接条件之后并且因此无条件移除行。在一个内连接中这两种过滤器类型没有语义区别。

如果我们把查询的选择度改变一点，我们可能得到一个非常不同的连接计划：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
-----
Hash Join (cost=230.47..713.98 rows=101 width=488)
  Hash Cond: (t2.unique2 = t1.unique2)
    -> Seq Scan on tenk2 t2 (cost=0.00..445.00 rows=10000 width=244)
    -> Hash (cost=229.20..229.20 rows=101 width=244)
        -> Bitmap Heap Scan on tenk1 t1 (cost=5.07..229.20 rows=101
width=244)
            Recheck Cond: (unique1 < 100)
            -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=101
width=0)
                Index Cond: (unique1 < 100)
```

这里规划器选择了使用一个哈希连接，在其中一个表的行被放入一个内存哈希表，在这之后其他表被扫描并且为每一行查找哈希表来寻找匹配。同样要注意缩进是如何反映计划结构的：tenk1上的位图扫描是哈希结点的输入，哈希结点会构造哈希表。然后哈希表会返回给哈希连接结点，哈希连接结点将从它的 outer 子计划读取行，并为每一个行搜索哈希表。

另一种可能的连接类型是一个归并连接，如下所示：

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
-----
Merge Join (cost=198.11..268.19 rows=10 width=488)
  Merge Cond: (t1.unique2 = t2.unique2)
    -> Index Scan using tenk1_unique2 on tenk1 t1 (cost=0.29..656.28 rows=101
width=244)
        Filter: (unique1 < 100)
    -> Sort (cost=197.83..200.33 rows=1000 width=244)
        Sort Key: t2.unique2
        -> Seq Scan on onek t2 (cost=0.00..148.00 rows=1000 width=244)
```

归并连接要求它的输入数据被按照连接键排序。在这个计划中，tenk1数据被使用一个索引扫描排序，以便能够按照正确的顺序来访问行。但是对于onek则更倾向于一个顺序扫描和排序，因为在那个表中有更多行需要被访问（对于很多行的排序，顺序扫描加排序常常比一个索引扫描好，因为索引扫描需要非顺序的磁盘访问）。

一种查看变体计划的方法是强制规划器丢弃它认为开销最低的任何策略，这可以使用Section 17.7. 中描述的启用/禁用标志实现（这是一个野蛮的工具，但是很有用。另见Section 14.3。例如，如果我们并不认同在前面的例子中顺序扫描加排序是处理表onek的最佳方法，我们可以尝试：

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
-----
Merge Join (cost=0.56..292.65 rows=10 width=488)
  Merge Cond: (t1.unique2 = t2.unique2)
    -> Index Scan using tenk1_unique2 on tenk1 t1 (cost=0.29..656.28 rows=101
        width=244)
        Filter: (unique1 < 100)
    -> Index Scan using onek_unique2 on onek t2 (cost=0.28..224.79 rows=1000
        width=244)
```

这显示规划器认为用索引扫描来排序onek的开销要比用顺序扫描加排序的方式高大约12%。当然，下一个问题是是否真的是这样。我们可以通过使用EXPLAIN ANALYZE来仔细研究一下，如下文所述。

## 14.1.2. EXPLAIN ANALYZE

可以通过使用EXPLAIN的ANALYZE选项来检查规划器估计值的准确性。通过使用这个选项，EXPLAIN会实际执行该查询，然后显示真实的行计数和在每个计划结点中累计的真实运行时间，还会有一个普通EXPLAIN显示的估计值。例如，我们可能得到这样一个结果：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
-----
Nested Loop (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377
rows=10 loops=1)
  -> Bitmap Heap Scan on tenk1 t1 (cost=4.36..39.47 rows=10 width=244)
      (actual time=0.057..0.121 rows=10 loops=1)
      Recheck Cond: (unique1 < 10)
      -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.36 rows=10
          width=0) (actual time=0.024..0.024 rows=10 loops=1)
          Index Cond: (unique1 < 10)
      -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.29..7.91 rows=1
          width=244) (actual time=0.021..0.022 rows=1 loops=10)
          Index Cond: (unique2 = t1.unique2)
Planning time: 0.181 ms
Execution time: 0.501 ms
```

注意“actual time”值是以毫秒计的真实时间，而cost估计值被以捏造的单位表示，因此它们不大可能匹配上。在这里面要查看的最重要的一点是估计的行计数是否合理地接近实际值。在这个例子中，估计值都是完全正确的，但是在实际中非常少见。

在某些查询计划中，可以多次执行一个子计划结点。例如，inner 索引扫描可能会因为上层嵌套循环计划中的每一个 outer 行而被执行一次。在这种情况下，loops值报告了执行该结点的总次数，并且 actual time 和行数值是这些执行的平均值。这是为了让这些数字能够与开销估计被显示的方式有可比性。将这些值乘上loops值可以得到在该结点中实际消耗的总时间。在上面的例子中，我们在执行tenk2的索引扫描上花费了总共 0.220 毫秒。

在某些情况中，EXPLAIN ANALYZE会显示计划结点执行时间和行计数之外的额外执行统计信息。例如，排序和哈希结点提供额外的信息：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;
```

QUERY PLAN

```
-----
Sort (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774
rows=100 loops=1)
  Sort Key: t1.fivethous
  Sort Method: quicksort Memory: 77kB
  -> Hash Join (cost=230.47..713.98 rows=101 width=488) (actual
time=0.711..7.427 rows=100 loops=1)
    Hash Cond: (t2.unique2 = t1.unique2)
    -> Seq Scan on tenk2 t2 (cost=0.00..445.00 rows=10000 width=244)
(actual time=0.007..2.583 rows=10000 loops=1)
    -> Hash (cost=229.20..229.20 rows=101 width=244) (actual
time=0.659..0.659 rows=100 loops=1)
      Buckets: 1024 Batches: 1 Memory Usage: 28kB
      -> Bitmap Heap Scan on tenk1 t1 (cost=5.07..229.20 rows=101
width=244) (actual time=0.080..0.526 rows=100 loops=1)
        Recheck Cond: (unique1 < 100)
        -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04
rows=101 width=0) (actual time=0.049..0.049 rows=100 loops=1)
          Index Cond: (unique1 < 100)
Planning time: 0.194 ms
Execution time: 8.008 ms
```

排序结点显示使用的排序方法（尤其是，排序是在内存中还是磁盘上进行）和需要的内存或磁盘空间量。哈希结点显示了哈希桶的数量和批数，以及被哈希表所使用的内存量的峰值（如果批数超过一，也将会涉及到磁盘空间使用，但是并没有被显示）。

另一种类型的额外信息是被一个过滤器条件移除的行数：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;
```

QUERY PLAN

```
-----
Seq Scan on tenk1 (cost=0.00..483.00 rows=7000 width=244) (actual
time=0.016..5.107 rows=7000 loops=1)
  Filter: (ten < 7)
  Rows Removed by Filter: 3000
Planning time: 0.083 ms
Execution time: 5.905 ms
```

这些值对于被应用在连接结点上的过滤器条件特别有价值。只有在至少有一个被扫描行或者在连接结点中一个可能的连接对被过滤器条件拒绝时，“Rows Removed”行才会出现。

一个与过滤器条件相似的情况出现在“有损”索引扫描中。例如，考虑这个查询，它搜索包含一个指定点的多边形：

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5, 2.0)';
```

## QUERY PLAN

```
-----
Seq Scan on polygon_tbl (cost=0.00..1.05 rows=1 width=32) (actual
time=0.044..0.044 rows=0 loops=1)
  Filter: (f1 @> '((0.5,2))'::polygon)
  Rows Removed by Filter: 4
Planning time: 0.040 ms
Execution time: 0.083 ms
```

规划器认为（非常正确）这个采样表太小不值得劳烦一次索引扫描，因此我们得到了一个普通的顺序扫描，其中的所有行都被过滤器条件拒绝。但是如果强制使得一次索引扫描可以被使用，我们看到：

```
SET enable_seqscan TO off;
```

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';
```

## QUERY PLAN

```
-----
Index Scan using gpolygonind on polygon_tbl (cost=0.13..8.15 rows=1 width=32)
(actual time=0.062..0.062 rows=0 loops=1)
  Index Cond: (f1 @> '((0.5,2))'::polygon)
  Rows Removed by Index Recheck: 1
Planning time: 0.034 ms
Execution time: 0.144 ms
```

这里我们可以看到索引返回一个候选行，然后它会被索引条件的重新检查拒绝。这是因为一个 GiST 索引对于多边形包含测试是“有损的”：它确实返回覆盖目标的多边形的行，然后我们必须对那些行上做精确的包含性测试。

EXPLAIN 有一个 BUFFERS 选项可以和 ANALYZE 一起使用来得到更多运行时统计信息：

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;
```

## QUERY PLAN

```
-----
Bitmap Heap Scan on tenk1 (cost=25.08..60.21 rows=10 width=244) (actual
time=0.323..0.342 rows=10 loops=1)
  Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
  Buffers: shared hit=15
  -> BitmapAnd (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309
rows=0 loops=1)
    Buffers: shared hit=7
    -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=101
width=0) (actual time=0.043..0.043 rows=100 loops=1)
      Index Cond: (unique1 < 100)
      Buffers: shared hit=2
    -> Bitmap Index Scan on tenk1_unique2 (cost=0.00..19.78 rows=999
width=0) (actual time=0.227..0.227 rows=999 loops=1)
      Index Cond: (unique2 > 9000)
      Buffers: shared hit=5
Planning time: 0.088 ms
```

Execution time: 0.423 ms

BUFFERS提供的数字帮助我们标识查询的哪些部分是对 I/O 最敏感的。

记住因为EXPLAIN ANALYZE实际运行查询，任何副作用都将照常发生，即使查询可能输出的任何结果被丢弃来支持打印EXPLAIN数据。如果你想要分析一个数据修改查询而不想改变你的表，你可以在分析完后回滚命令，例如：

BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

QUERY PLAN

```
-----
Update on tenk1 (cost=5.07..229.46 rows=101 width=250) (actual
time=14.628..14.628 rows=0 loops=1)
  -> Bitmap Heap Scan on tenk1 (cost=5.07..229.46 rows=101 width=250) (actual
time=0.101..0.439 rows=100 loops=1)
    Recheck Cond: (unique1 < 100)
    -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=101
width=0) (actual time=0.043..0.043 rows=100 loops=1)
        Index Cond: (unique1 < 100)
Planning time: 0.079 ms
Execution time: 14.727 ms
```

ROLLBACK;

正如在这个例子中所看到的，当查询是一个INSERT、UPDATE或DELETE命令时，应用表更改的实际工作由顶层插入、更新或删除计划结点完成。这个结点之下的计划结点执行定位旧行以及/或者计算新数据的工作。因此在上面，我们看到我们已经见过的位图表扫描，它的输出被交给一个更新结点，更新结点会存储被更新过的行。还有一点值得注意的是，尽管数据修改结点可能要可观的运行时间（这里，它消耗最大份额的时间），规划器当前并没有对开销估计增加任何东西来说明这些工作。这是因为这些工作对每一个正确的查询计划都得做，所以它不影响计划的选择。

当一个UPDATE或者DELETE命令影响继承层次时，输出可能像这样：

EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;

QUERY PLAN

```
-----
Update on parent (cost=0.00..24.53 rows=4 width=14)
  Update on parent
  Update on child1
  Update on child2
  Update on child3
  -> Seq Scan on parent (cost=0.00..0.00 rows=1 width=14)
    Filter: (f1 = 101)
  -> Index Scan using child1_f1_key on child1 (cost=0.15..8.17 rows=1
width=14)
    Index Cond: (f1 = 101)
  -> Index Scan using child2_f1_key on child2 (cost=0.15..8.17 rows=1
width=14)
    Index Cond: (f1 = 101)
  -> Index Scan using child3_f1_key on child3 (cost=0.15..8.17 rows=1
width=14)
```



Index Cond: (f1 = 101)

在这个例子中，更新节点需要考虑三个子表以及最初提到的父表。因此有四个输入 的扫描子计划，每一个对应于一个表。为清楚起见，在更新节点上标注了将被更新 的相关目标表，显示的顺序与相应的子计划相同。

EXPLAIN ANALYZE显示的 Planning time是从一个已解析的查询生成查询计划并进行优化 所花费的时间，其中不包括解析和重写。

EXPLAIN ANALYZE显示的Execution time包括执行器的启动和关闭时间，以及运行被触发的任何触发器的时间，但是它不包括解析、重写或规划的时间。如果有花在执行BEFORE执行器的时间，它将被包括在相关的插入、更新或删除节点的时间内；但是用来执行AFTER 触发器的时间没有被计算，因为AFTER触发器是在整个计划完成后被触发的。在每个触发器（BEFORE或AFTER）也被独立地显示。注意延迟约束触发器将不会被执行，直到事务结束，并且因此根本不会被EXPLAIN ANALYZE考虑。

### 14.1.3. 警告

在两种有效的方法中EXPLAIN ANALYZE所度量的运行时间可能偏离同一个查询的正常执行。首先，由于不会有输出行被递交给客户端，网络传输开销和 I/O 转换开销没有被包括在内。其次，由EXPLAIN ANALYZE所增加的度量开销可能会很可观，特别是在操作系统调用gettimeofday()很慢的机器上。你可以使用lt\_test\_timing工具来度量在你的系统上的计时开销。

EXPLAIN结果不应该被外推到与你实际测试的非常不同的情况。例如，一个很小的表上的结果不能被假定成适合大型表。规划器的开销估计不是线性的，并且因此它可能为一个更大或更小的表选择一个不同的计划。一个极端例子是，在一个只占据一个磁盘页面的表上，你将几乎总是得到一个顺序扫描计划，而不管索引是否可用。规划器认识到它在任何情况下都采用一次磁盘页面读取来处理该表，因此用额外的页面读取去查看一个索引是没有价值的（我们已经在前面的polygon\_tbl例子中见过）。

在一些情况中，实际的值和估计的值不会匹配得很好，但是这并非错误。一种这样的情况发生在计划结点的执行被LIMIT或类似的效果很快停止。例如，在我们之前用过的LIMIT查询中：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;
```

QUERY PLAN

```
-----
Limit (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2
loops=1)
  -> Index Scan using tenk1_unique2 on tenk1 (cost=0.29..72.42 rows=10
width=244) (actual time=0.174..0.244 rows=2 loops=1)
    Index Cond: (unique2 > 9000)
    Filter: (unique1 < 100)
    Rows Removed by Filter: 287
Planning time: 0.096 ms
Execution time: 0.336 ms
```

索引扫描结点的估计开销和行计数被显示成好像它会运行到完成。但是实际上限制结点在得到两个行之后就停止请求行，因此实际的行计数只有 2 并且运行时间远低于开销估计所建议的时间。这并非估计错误，这仅仅一种估计值和实际值显示方式上的不同。

归并连接也有类似的现象。如果一个归并连接用尽了一个输入并且其中的最后一个键值小于另一个输入中的下一个键值，它将停止读取另一个输入。在这种情况下，不会有更多的匹配并且因此不需要扫描第二个输入的剩余部分。这会导致不读取一个子结点的所有内容，其结

果就像在LIMIT中所提到的。另外，如果 `outer`（第一个）子结点包含带有重复键值的行，`inner`（第二个）子结点会被倒退并且被重新扫描来找能匹配那个键值的行。EXPLAIN ANALYZE会统计相同 `inner` 行的重复发出，就好像它们是真实的额外行。当有很多 `outer` 重复时，对 `inner` 子计划结点所报告的实际行计数会显著地大于实际在 `inner` 关系中的行数。

由于实现的限制，`BitmapAnd` 和 `BitmapOr` 结点总是报告它们的实际行计数为零。

通常，EXPLAIN将显示规划器生成的每个计划节点。但是，在某些情况下，执行器可以不执行某些节点，因为根据规划时不可用的参数值能确定这些节点无法产生任何行。（当前，这仅会在扫描分区表的Append或MergeAppend节点的子节点中发生。）发生这种情况时，将从EXPLAIN输出中省略这些计划节点，并显示Subplans Removed: N的标识。

如果使用`Explain ANALYZE`，则会显示所有节点。未执行的节点会标注为“未执行”。当存在许多这样的节点时，`explain`输出会变得无聊和分散注意力。

将`lightdb_explain_no_runtime_pruned_partition`设置为on，以消除从未执行的节点。例如，

```
SET enable_partition_pruning = on;
SET lightdb_explain_no_runtime_pruned_partition = on;
EXPLSIN (ANALYZE) select * from measurement where logdate < (select
  date('2006-04-01'));
```

#### QUERY PLAN

```
-----
Append (cost=0.01..144.85 rows=2468 width=16) (actual time=0.572..0.945 rows=8
loops=1)
  InitPlan 1 (returns $0)
    -> Result (cost=0.00..0.01 rows=1 width=4) (actual time=0.002..0.002
rows=1 loops=1)
      -> Seq Scan on measurement_y2006m02 measurement_1 (cost=0.00..33.12
rows=617 width=16) (actual time=0.520..0.523 rows=4 loops=1)
        Filter: (logdate < $0)
      -> Seq Scan on measurement_y2006m03 measurement_2 (cost=0.00..33.12
rows=617 width=16) (actual time=0.361..0.365 rows=4 loops=1)
        Filter: (logdate < $0)
```

表`measurement`有四个分区，其中两个在运行时被剪枝，因此未显示在查询计划中。有关更多信息，请参见Section 5.11.4

## 14.2. 规划器使用的统计信息

### 14.2.1. 单列统计信息

如我们在上一节所见，查询规划器需要估计一个查询要检索的行数，这样才能对查询计划做出好的选择。本节对系统用于这些估计的统计信息进行一个快速的介绍。

统计信息的一个部分就是每个表和索引中的项的总数，以及每个表和索引占用的磁盘块数。这些信息保存在`pg_class`表的`reltuples`和`relpages`列中。我们可以用类似下面的查询查看这些信息：

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

relname          | relkind | reltuples | relpages
```

tenk1	r	10000	358
tenk1_hundred	i	10000	30
tenk1_thous_tenthous	i	10000	30
tenk1_unique1	i	10000	30
tenk1_unique2	i	10000	30

(5 rows)

这里我们可以看到tenk1包含 10000 行，它的索引也有这么多行，但是索引远比表小得多（不奇怪）。

出于效率考虑，reltuples和relpages不是实时更新的，因此它们通常包含有些过时的值。它们被VACUUM、ANALYZE和几个 DDL 命令（例如CREATE INDEX）更新。一个不扫描全表的VACUUM或ANALYZE操作（常见情况）将以它扫描的部分为基础增量更新reltuples计数，这就导致了一个近似值。在任何情况中，规划器将缩放它在pg\_class中找到的值来匹配当前的物理表尺寸，这样得到一个较紧的近似。

大多数查询只是检索表中行的一部分，因为它们有限制要被检查的行的WHERE子句。因此规划器需要估算WHERE子句的选择度，即符合WHERE子句中每个条件的行的比例。用于这个任务的信息存储在pg\_statistic系统目录中。在pg\_statistic中的项由ANALYZE和VACUUM ANALYZE命令更新，并且总是近似值（即使刚刚更新完）。

除了直接查看pg\_statistic之外，手工检查统计信息的时候最好查看它的视图pg\_stats。pg\_stats被设计为更容易阅读。而且，pg\_stats是所有人都可以读取的，而pg\_statistic只能由超级用户读取（这样可以避免非授权用户从统计信息中获取一些其他人的表的内容的信息。pg\_stats视图被限制为只显示当前用户可读的表）。例如，我们可以：

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';
```

attname	inherited	n_distinct	most_common_vals
name	f	-0.363388	I- 580
			Ramp+
			I- 880
			Ramp+
			Sp Railroad
name	t	-0.284859	+
			I- 580
			Ramp
			I- 680
			Ramp+
name	t	-0.284859	I- 880
			Ramp+
			I- 580
			Ramp+
			I- 680
name	t	-0.284859	I- 580
			+
name	t	-0.284859	State Hwy 13
			Ramp

(2 rows)

注意，这两行显示的是相同的列，一个对应开始于road表（inherited=t）的完全继承层次，另一个只包括road表本身（inherited=f）。

ANALYZE在pg\_statistic中存储的信息量（特别是每个列的most\_common\_vals中的最大项数和histogram\_bounds数组）可以用ALTER TABLE SET STATISTICS命令为每一列设置，或者通过设置配置变量default\_statistics\_target进行全局设置。目前的默认限制是100个项。提升该限制可能会让规划器做出更准确的估计（特别是对那些有不规则数据分布的列），其代价是在pg\_statistic中消耗了更多空间，并且需要略微多一些的时间来计算估计数值。相比之下，比较低的限制可能更适合那些数据分布比较简单的列。

更多规划器对统计信息的使用可参阅Chapter 66

## 14.2.2. 扩展统计信息

常常可以看到由于查询子句中用到的多个列相互关联而运行着糟糕的执行计划的慢查询。规划器通常会假设多个条件是彼此独立的，这种假设在列值相互关联的情况下是不成立的。由于常规的统计信息天然的针对个体列的性质，它们无法捕捉到跨列关联的知识。不过，LightDB有能力计算多元统计信息，它能捕捉这类信息。

由于可能的列组合数非常巨大，所以不可能自动计算多元统计信息。可以创建扩展统计信息对象（更常被称为统计信息对象）来指示服务器获得跨感兴趣列集合的统计信息。

统计信息对象可以使用CREATE STATISTICS命令创建。这样一个对象的创建仅仅是创建了一个目录项来表示对统计信息有兴趣。实际的数据收集是由ANALYZE（或者是一个手工命令，或者是后台的自动分析）执行的。收集到的值可以在pg\_statistic\_ext\_data目录中看到。

ANALYZE基于它用来计算常规单列统计信息的表行样本来计算扩展统计信息。由于样本的尺寸会随着表或者表列的统计信息目标（如前一节所述）增大而增加，更大的统计信息目标通常将会导致更准确的扩展统计信息，同时也会导致更多花在计算扩展统计信息之上的时间。

下面的小节介绍当前支持的扩展统计信息类型。

### 14.2.2.1. 函数依赖

最简单的一类扩展统计信息跟踪函数依赖，这是在数据库范式定义中使用的概念。如果列a的值的知识足以决定列b的值，即不会有二个行具有相同的a值但是有不同的b值，我们就说列b函数依赖于列a。在一个完全规范化的数据库中，函数依赖应该仅存在于主键和超键上。不过，实际上很多数据集合会由于各种原因无法被完全规范化，常见的例子是为了性能而有意识地反规范化。即使在一个完全规范化的数据库中，也会有某些列之间的部分关联，这些可以表达成部分函数依赖。

函数依赖的存在直接影响了特定查询中估计的准确性。如果一个查询包含独立列和依赖列上的条件，依赖列上的条件不会进一步降低结果的尺寸。但是如果缺少函数依赖的知识，查询规划器将假定条件是独立的，导致对结果尺寸的低估。

要告知规划器有关函数依赖的信息，ANALYZE可以收集跨列依赖的测度。评估所有列组之间的依赖程度可能会昂贵到不可实现，因此数据收集被限制为针对那些在一个统计信息对象中一起出现的列组（用dependencies选项定义）。建议只对强相关的列组创建dependencies统计信息，以避免ANALYZE以及后期查询规划中不必要的开销。

这里是一个收集函数依赖统计信息的例子：

```
CREATE STATISTICS stts (dependencies) ON city, zip FROM zipcodes;
```

```
ANALYZE zipcodes;
```

```
SELECT stxname, stxkeys, stxddependencies
   FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts';
```

stxname	stxkeys	stxddependencies
stts	1 5	{ "1 => 5": 1.000000, "5 => 1": 0.423130 }

(1 row)

这里可以看到列1（邮编）完全决定列5（城市），因此系数为1.0，而城市仅决定42%的邮编，意味着有很多城市（58%）有多个邮编。

在为涉及函数依赖列的查询计算选择度时，规划器会使用依赖系数来调整针对条件的选择度估计，这样就不会产生低估。

### 14.2.2.1.1. 函数依赖的限制

当前只有在考虑简单等值条件（将列与常量值比较）和具有常量值的IN子句时，函数依赖才适用。不会使用它们来改进比较两个列或者比较列和表达式的等值条件的估计，也不会使用它们来改进范围子句、LIKE或者任何其他类型的条件。

在用函数依赖估计时，规划器假定在涉及的列上的条件是兼容的并且因此是冗余的。如果它们是不兼容的，正确的估计将是零行，但是那种可能性不会被考虑。例如，给定一个这样的查询

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';
```

规划器将会忽视city子句，因为它不改变选择度，这是正确的。不过，即便真地只有零行满足下面的查询，规划器也会做出同样的假设

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';
```

不过，函数依赖统计信息无法提供足够的信息来排除这种情况。

在很多实际情况中，这种假设通常是能满足的。例如，在应用程序中可能有一个GUI仅允许选择兼容的城市和邮编值用在查询中。但是如果不是这样，函数依赖可能就不是一个可行的选项。

### 14.2.2.2. 多元可区分值计数

单列统计信息存储每一列中可区分值的数量。在组合多个列（例如GROUP BY a, b）时，如果规划器只有单列统计数据，则对可区分值数量的估计常常会错误，导致选择不好的计划。

为了改进这种估计，ANALYZE可以为列组收集可区分值统计信息。和以前一样，为每一种可能的列组合做这件事情是不切实际的，因此只会为一起出现在一个统计信息对象（用ndistinct选项定义）中的列组收集数据。将会为列组中列出的列的每一种可能的组合都收集数据。

继续之前的例子，ZIP代码表中的可区分值计数可能像这样：

```
CREATE STATISTICS stts2 (ndistinct) ON city, state, zip FROM zipcodes;
```

```
ANALYZE zipcodes;
```

```
SELECT stxkeys AS k, stxnddistinct AS nd
FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
WHERE stxname = 'stts2';
-[ RECORD 1 ]-----
k | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}
(1 row)
```

这表示有三种列组合有33178个可区分值：ZIP代码和州、ZIP代码和城市、ZIP代码+城市+州（事实上对于表中给定的一个唯一的ZIP代码，它们本来就应该是一样的）。另一方面，城市和州的组合只有27435个可区分值。

建议只对实际用于分组的列组合以及分组数错误估计导致了糟糕计划的列组合创建ndistinct统计信息对象。否则，ANALYZE循环只会被浪费。

### 14.2.2.3. 多元MCV列表

为每列存储的另一种统计信息是频繁值列表。这样可以对单个列进行非常准确的估计，但是对于在多个列上具有条件的查询，可能会导致严重的错误估计。

为了改善这种估计，ANALYZE可以收集列组合上的MCV列表。与功能依赖和n-distinct系数类似，对每种可能的列分组进行此操作都是不切实际的。在这种情况下，甚至更是如此，因为MCV列表（与功能依赖性和n-distinct系数不同）存储了公共列值。因此，仅收集在使用mcv选项定义的统计对象中同时出现的那些列组的数据。

继续前面的示例，邮政编码表的MCV列表可能类似于以下内容（与更简单的统计信息不同，它需要一个函数来检查MCV内容）：

```
CREATE STATISTICS stts3 (mcv) ON city, state FROM zipcodes;
```

```
ANALYZE zipcodes;
```

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
        pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts3';
```

index	values	nulls	frequency	base_frequency
0	{Washington, DC}	{f, f}	0.003467	2.7e-05
1	{Apo, AE}	{f, f}	0.003067	1.9e-05
2	{Houston, TX}	{f, f}	0.002167	0.000133
3	{El Paso, TX}	{f, f}	0.002	0.000113
4	{New York, NY}	{f, f}	0.001967	0.000114
5	{Atlanta, GA}	{f, f}	0.001633	3.3e-05
6	{Sacramento, CA}	{f, f}	0.001433	7.8e-05
7	{Miami, FL}	{f, f}	0.0014	6e-05
8	{Dallas, TX}	{f, f}	0.001367	8.8e-05
9	{Chicago, IL}	{f, f}	0.001333	5.1e-05
...				

(99 rows)

这表明城市和州的最常见组合是华盛顿特区，实际频率（在样本中）约为0.35%。组合的基本频率（根据简单的每列频率计算）仅为0.0027%，导致两个数量级的低估。

建议仅在实际在条件中一起使用的列的组合上创建MCV统计对象，对于这些组合，错误估计组数会导致糟糕的执行计划。否则，只会浪费ANALYZE和规划时间。

## 14.3. 用显式JOIN子句控制规划器

我们可以在一定程度上用显式JOIN语法控制查询规划器。要明白为什么需要它，我们首先需要一些背景知识。

在一个简单的连接查询中，例如：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

规划器可以自由地按照任何顺序连接给定的表。例如，它可以生成一个使用WHERE条件a.id = b.id连接 A 到 B 的查询计划，然后用另外一个WHERE条件把 C 连接到这个连接表。或者它可以先连接 B 和 C 然后再连接 A 得到同样的结果。或者也可以连接 A 到 C 然后把结果与 B 连接——不过这么做效率不好，因为必须生成完整的 A 和 C 的迪卡尔积，而在WHERE子句中没有可用条件来优化该连接（LightDB执行器中的所有连接都发生在两个输入表之间，所以它必须以这些形式之一建立结果）。重要的一点是这些不同的连接可能性给出在语义等效的结果，但在执行开销上却可能有巨大的差别。因此，规划器会对它们进行探索并尝试找出最高效的查询计划。

当一个查询只涉及两个或三个表时，那么不需要考虑很多连接顺序。但是可能的连接顺序数随着表数目的增加成指数增长。当超过十个左右的表以后，实际上根本不可能对所有可能性做一次穷举搜索，甚至对六七个表都需要相当长的时间进行规划。当有太多的输入表

时，LightDB规划器将从穷举搜索切换为一种遗传概率搜索，它只需要考虑有限数量的可能性（切换的阈值用geqo\_threshold运行时参数设置）。遗传搜索用时更少，但是并不一定会找到最好的计划。

当查询涉及外连接时，规划器比处理普通（内）连接时拥有更小的自由度。例如，考虑：

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

尽管这个查询的约束表面上和前一个非常相似，但它们的语义却不同，因为如果 A 里有任何一行不能匹配 B 和 C 的连接表中的行，它也必须被输出。因此这里规划器对连接顺序没有什么选择：它必须先连接 B 到 C，然后把 A 连接到该结果上。相应地，这个查询比前面一个花在规划上的时间更少。在其它情况下，规划器就有可能确定多种连接顺序都是安全的。例如，给定：

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

将 A 首先连接到 B 或 C 都是有效的。当前，只有FULL JOIN完全约束连接顺序。大多数涉及LEFT JOIN或RIGHT JOIN的实际情况都在某种程度上可以被重新排列。

显式连接语法（INNER JOIN、CROSS JOIN或无修饰的JOIN）在语义上和FROM中列出输入关系是一样的，因此它不约束连接顺序。

即使大多数类型的JOIN并不完全约束连接顺序，但仍然可以指示LightDB查询规划器将所有JOIN子句当作有连接顺序约束来对待。例如，这里的三个查询在逻辑上是等效的：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

但如果我们告诉规划器遵循JOIN的顺序，那么第二个和第三个还是要比第一个花在规划上的时间少。这个效果对于只有三个表的连接而言是微不足道的，但对于数目众多的表，可能就是救命稻草了。

要强制规划器遵循显式JOIN的连接顺序，我们可以把运行时参数join\_collapse\_limit设置为 1（其它可能值在下文讨论）。

你不必为了缩短搜索时间来完全约束连接顺序，因为可以在一个普通FROM列表里使用JOIN操作符。例如，考虑：

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

如果设置join\_collapse\_limit = 1，那么这就强迫规划器先把 A 连接到 B，然后再连接到其它的表上，但并不约束它的选择。在这个例子中，可能的连接顺序的数目减少了 5 倍。

按照这种方法约束规划器的搜索是一个有用的技巧，不管是对减少规划时间还是对引导规划器生成好的查询计划。如果规划器按照默认选择了一个糟糕的连接顺序，你可以通过JOIN语法强迫它选择一个更好的顺序——假设你知道一个更好的顺序。我们推荐进行实验。

一个非常相近的影响规划时间的问题是把子查询压缩到它们的父查询中。例如，考虑：

```
SELECT *
FROM x, y,
  (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

这种情况可能在使用包含连接的视图时出现；该视图的SELECT规则将被插入到引用视图的地方，得到与上文非常相似的查询。通常，规划器会尝试把子查询压缩到父查询里，得到：

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

这样通常会生成一个比独立的子查询更好些的计划（例如，outer 的WHERE条件可能先把 X 连接到 A 上，这样就消除了 A 中的许多行，因此避免了形成子查询的全部逻辑输出）。但是同时，我们增加了规划的时间；在这里，我们用五路连接问题替代了两个独立的三路连接问题。这样的差别是巨大的，因为可能的计划数的是按照指数增长的。如果有超过from\_collapse\_limit个FROM项将会导致父查询，规划器将尝试通过停止提升子查询来避免卡在巨大的连接搜索问题中。你可以通过调高或调低这个运行时参数在规划时间和计划的质量之间取得平衡。

from\_collapse\_limit和join\_collapse\_limit的命名相似，因为它们做的几乎是同一件事：一个控制规划器何时将把子查询“平面化”，另外一个控制何时把显式连接平面化。通常，你要么把join\_collapse\_limit设置成和from\_collapse\_limit一样（这样显式连接和子查询的行为类似），要么把join\_collapse\_limit设置为 1（如果你想用显式连接控制连接顺序）。但是你可以把它们设置成不同的值，这样你就可以细粒度地调节规划时间和运行时间之间的平衡。

## 14.4. 填充一个数据库

第一次填充数据库时可能需要插入大量的数据。本节包含一些如何让这个处理尽可能高效的建议。

### 14.4.1. 禁用自动提交

在使用多个INSERT时，关闭自动提交并且只在最后做一次提交（在普通 SQL 中，这意味着在开始发出BEGIN并且在结束时发出COMMIT。某些客户端库可能背着你就做了这些，在这种情况下你需要确定在你需要做这些时该库确实帮你做了）。如果你允许每一个插入都被独立地提交，LightDB要为每一个被增加的行做很多工作。在一个事务中做所有插入的一个额外好处是：如果一个行的插入失败则所有之前插入的行都会被回滚，这样你不会被卡在部分载入的数据中。

### 14.4.2. 使用COPY

使用COPY在一条命令中装载所有记录，而不是一系列INSERT命令。COPY命令是为装载大量行而优化过的；它没INSERT那么灵活，但是在大量数据装载时导致的负荷也更少。因为COPY是单条命令，因此使用这种方法填充表时无须关闭自动提交。

如果你不能使用COPY，那么使用PREPARE来创建一个预备INSERT语句也有所帮助，然后根据需要EXECUTE多次。这样就避免了重复分析和规划INSERT的负荷。不同接口以不同的方式提供该功能，可参阅接口文档中的“预备语句”。

请注意，在载入大量行时，使用COPY几乎总是比使用INSERT快，即使使用了PREPARE并且把多个插入被成批地放入一个单一事务。

同样的事务中，COPY比更早的CREATE TABLE或TRUNCATE命令更快。在这种情况下，不需要写 WAL，因为在一个错误的情况下，包含新载入数据的文件不管怎样都将被移除。不过，只有当wal\_level设置为minimal（此时所有的命令必须写 WAL）时才会应用这种考虑。

### 14.4.3. 移除索引

如果你正在载入一个新创建的表，最快的方法是创建该表，用COPY批量载入该表的数据，然后创建表需要的任何索引。在已存在数据的表上创建索引要比在每一行被载入时增量地更新它更快。



如果你正在对现有表增加大量的数据，删除索引、载入表然后重新创建索引可能是最好的方案。当然，在缺少索引的期间，其它数据库用户的数据库性能将会下降。我们在删除唯一索引之前还需要仔细考虑清楚，因为唯一约束提供的错误检查在缺少索引的时候会丢失。

#### 14.4.4. 移除外键约束

和索引一样，“成批地”检查外键约束比一行行检查效率更高。因此，先删除外键约束、载入数据然后重建约束会很有用。同样，载入数据和约束缺失期间错误检查的丢失之间也存在平衡。

更重要的是，当你在已有外键约束的情况下向表中载入数据时，每个新行需要一个在服务器的待处理触发器事件（因为是一个触发器的触发会检查行的外键约束）列表的条目。载入数百万行会导致触发器事件队列溢出可用内存，造成不能接受的交换或者甚至是命令的彻底失败。因此在载入大量数据时，可能需要（而不仅仅是期望）删除并重新应用外键。如果临时移除约束不可接受，那唯一的其他办法可能就是就是将载入操作分解成更小的事务。

#### 14.4.5. 增加maintenance\_work\_mem

在载入大量数据时，临时增大maintenance\_work\_mem配置变量可以改进性能。这个参数也可以帮助加速CREATE INDEX命令和ALTER TABLE ADD FOREIGN KEY命令。它不会对COPY本身起很大作用，所以这个建议只有在你使用上面的一个或两个技巧时才有用。

#### 14.4.6. 增加max\_wal\_size

临时增大max\_wal\_size配置变量也可以让大量数据载入更快。这是因为向LightDB中载入大量的数据将导致检查点的发生比平常（由checkpoint\_timeout配置变量指定）更频繁。无论何时发生一个检查点时，所有脏页都必须被刷写到磁盘上。通过在批量数据载入时临时增加max\_wal\_size，所需的检查点数目可以被缩减。

#### 14.4.7. 禁用 WAL 归档和流复制

当使用 WAL 归档或流复制向一个安装中载入大量数据时，在录入结束后执行一次新的基础备份比处理大量的增量 WAL 数据更快。为了防止载入时记录增量 WAL，通过将wal\_level设置为minimal、将archive\_mode设置为off以及将max\_wal\_senders设置为零来禁用归档和流复制。但需要注意的是，修改这些设置需要重启服务，从而使先前进行的基本备份无法用于存档恢复或备用服务器，并可能导致数据丢失。

除了避免归档器或 WAL 发送者处理 WAL 数据的时间之外，这样做将实际上使某些命令更快，因为如果wal\_level是minimal并且当前子事务（或顶级事务）创建或截断了它们更改的表或索引，则它们根本不编写 WAL。（通过在最后执行一个fsync而不是写 WAL，它们能以更小代价保证崩溃安全）。

#### 14.4.8. 事后运行ANALYZE

不管什么时候你显著地改变了表中的数据分布后，我们都强烈推荐运行ANALYZE。这包括向表中批量载入大量数据。运行ANALYZE（或者VACUUM ANALYZE）保证规划器有表的最新统计信息。如果没有统计数据或者统计数据过时，那么规划器在查询规划时可能做出很差的决定，导致在任意表上的性能低下。需要注意的是，如果启用了 autovacuum 守护进程，它可能会自动运行ANALYZE；参阅Section 22.1.1和Section 22.1.6

#### 14.4.9. 关于lt\_dump的一些笔记

lt\_dump生成的转储脚本自动应用上面的若干个（但不是全部）技巧。要尽可能快地载入lt\_dump转储，你需要手工做一些额外的事情（请注意，这些要点适用于恢复一个转储，而不是创建它的时候。同样的要点也适用于使用ltsql载入一个文本转储或用lt\_restore从一个lt\_dump归档文件载入）。

默认情况下，`lt_dump`使用COPY，并且当它在生成一个完整的模式和数据转储时，它会很小心地先装载数据，然后创建索引和外键。因此在这种情况下，一些指导方针是被自动处理的。你需要做的是：

- 为`maintenance_work_mem`和`max_wal_size`设置适当的（即比正常值大的）值。
- 如果使用 `WAL` 归档或流复制，在转储时考虑禁用它们。在载入转储之前，可通过将`archive_mode`设置为`off`、将`wal_level`设置为`minimal`以及将`max_wal_senders`设置为零（在录入dump前）来实现禁用。之后，将它们设回正确的值并执行一次新的基础备份。
- 采用`lt_dump`和`lt_restore`的并行转储和恢复模式进行实验并且找出要使用的最佳并发任务数量。通过使用`-j`选项的并行转储和恢复应该能为你带来比串行模式高得多的性能。
- 考虑是否应该在一个单一事务中恢复整个转储。要这样做，将`-l`或`--single-transaction`命令行选项传递给`ltsql`或`lt_restore`。当使用这种模式时，即使是一个很小的错误也会回滚整个恢复，可能会丢弃已经处理了很多个小时的工作。根据数据间的相关性，可能手动清理更好。如果你使用一个单一事务并且关闭了 `WAL` 归档，`COPY`命令将运行得最快。
- 如果在数据库服务器上有多个 CPU 可用，可以考虑使用`lt_restore`的`--jobs`选项。这允许并行数据载入和索引创建。
- 之后运行ANALYZE。

一个只涉及数据的转储仍将使用COPY，但是它不会删除或重建索引，并且它通常不会触碰外键。<sup>1</sup> 因此当载入一个只有数据的转储时，如果你希望使用那些技术，你需要负责删除并重建索引和外键。在载入数据时增加`max_wal_size`仍然有用，但是不要去增加`maintenance_work_mem`；不如说在以后手工重建索引和外键时你已经做了这些。并且不要忘记在完成时执行ANALYZE，详见Section 22.1.4和Section 22.1.6

## 14.5. 非持久设置

持久性是数据库的一个保证已提交事务的记录的特性（即使是发生服务器崩溃或断电）。然而，持久性会明显增加数据库的负荷，因此如果你的站点不需要这个保证，LightDB可以被配置成运行更快。在这种情况下，你可以调整下列配置来提高性能。除了下面列出的，在数据库软件崩溃的情况下也能保证持久性。当这些设置被使用时，只有操作系统突然的崩溃会产生数据丢失或损坏的风险。

- 将数据库集簇的数据目录放在一个内存支持的文件系统上（即RAM磁盘）。这消除了所有的数据库磁盘 I/O，但将数据存储限制到可用的内存量（可能有交换区）。
- 关闭`fsync`；不需要将数据刷入磁盘。
- 关闭`synchronous_commit`；可能不需要在每次提交时强制把WAL写入磁盘。这种设置可能会在数据库崩溃时带来事务丢失的风险（但是没有数据破坏）。
- 关闭`full_page_writes`；不需要警惕部分页面写入。
- 增加`max_wal_size`和`checkpoint_timeout`；这会降低检查点的频率，但会增加`lt_wal`的存储要求。
- 创建不做日志的表 来避免WAL写入，不过这会让表在崩溃时不安全。

<sup>1</sup> 你可以通过使用`--disable-triggers`选项的方法获得禁用外键的效果 — 不过你要意识到这么做是消除（而不是推迟）外键验证。因此如果你使用该选项，就可能插入坏数据。

---

## Chapter 15. 并行查询

LightDB能设计出利用多 CPU 让查询更快的查询计划。这种特性被称为并行查询。由于现有实现的限制或者因为没有比连续查询计划更快的查询计划存在，很多查询并不能从并行查询获益。不过，对于那些可以从并行查询获益的查询来说，并行查询带来的速度提升是显著的。很多查询在使用并行查询时比之前快了超过两倍，有些查询是以前的四倍甚至更多的倍数。那些访问大量数据但只返回其中少数行给用户的查询最能从并行查询中获益。这一章介绍一些并行查询如何工作的细节以及哪些情况下可以使用并行查询，这样希望充分利用并行查询的用户可以理解他们能从并行查询得到什么。

### 15.1. 并行查询如何工作

当优化器判断对于某一个特定的查询，并行查询是最快的执行策略时，优化器将创建一个查询计划。该计划包括一个 Gather或者Gather Merge节点。下面是一个简单的例子：

```
EXPLAIN SELECT * FROM ltbench_accounts WHERE filler LIKE '%x%';
          QUERY PLAN
-----
Gather  (cost=1000.00..217018.43 rows=1 width=97)
  Workers Planned: 2
  -> Parallel Seq Scan on ltbench_accounts  (cost=0.00..216018.33 rows=1
width=97)
    Filter: (filler ~ '~' '%x%' ::text)
(4 rows)
```

在所有的情形下，Gather或Gather Merge节点都只有一个子计划，它是将被并行执行的计划的一部分。如果Gather或Gather Merge节点位于计划树的最顶层，那么整个查询将并行执行。如果它位于计划树的其他位置，那么只有查询中在它之下的那一部分会并行执行。在上面的例子中，查询只访问了一个表，因此除Gather节点本身之外只有一个计划节点。因为该计划节点是Gather节点的孩子节点，所以它会并行执行。

使用 EXPLAIN命令，你能看到规划器选择的工作者数量。当查询执行期间到达Gather节点时，实现用户会话的进程将会请求和规划器选中的工作者数量一样多的后台工作者进程。规划器将考虑使用的后台工作者的数量被限制为最多max\_parallel\_workers\_per\_gather个。任何时候能够存在的后台工作者进程的总数由max\_worker\_processes和max\_parallel\_workers限制。因此，一个并行查询可能会使用比规划中少的工作者来运行，甚至有可能根本不使用工作者。最优的计划可能取决于可用工作者的数量，因此这可能会导致不好的查询性能。如果这种情况经常发生，那么就应当考虑一下提高max\_worker\_processes和max\_parallel\_workers的值，这样更多的工作者可以同时运行；或者降低max\_parallel\_workers\_per\_gather，这样规划器会要求少一些的工作者。

为一个给定并行查询成功启动的后台工作者进程都将会执行计划的并行部分。这些工作者的领导者也将执行该计划，不过它还有一个额外的任务：它还必须读取所有由工作者产生的元组。当整个计划的并行部分只产生了少量元组时，领导者通常将表现为一个额外的加速查询执行的工作者。反过来，当计划的并行部分产生大量的元组时，领导者将几乎全用来读取由工作者产生的元组并且执行Gather或Gather Merge节点上层计划节点所要求的任何进一步处理。在这些情况下，领导者所作的执行并行部分的工作将会很少。

当计划的并行部分的顶层节点是Gather Merge而不是Gather时，它表示每个执行计划并行部分的进程会产生有序的元组，并且领导者执行一种保持顺序的合并。相反，Gather会以任何方便的顺序从工作者读取元组，这会破坏可能已经存在的排序顺序。

## 15.2. 何时会用到并行查询？

有几种设置会导致查询规划器在任何情况下都不生成并行查询计划。为了让并行查询计划能够被生成，必须配置好下列设置。

- `max_parallel_workers_per_gather` 必须被设置为大于零的值。这是一种特殊情况，更加普遍的原则是所用的工作者数量不能超过 `max_parallel_workers_per_gather` 所配置的数量。

此外，系统一定不能运行在单用户模式下。因为在单用户模式下，整个数据库系统运行在单个进程中，没有后台工作者进程可用。

如果下面的任一条件为真，即便对一个给定查询通常可以产生并行查询计划，规划器都不会为它产生并行查询计划：

- 查询要写任何数据或者锁定任何数据库行。如果一个查询在顶层或者 CTE 中包含了数据修改操作，那么不会为该查询产生并行计划。作为例外，以下创建新表并更新数据的命令，可以对查询的 SELECT 部分使用并行计划。
  - `CREATE TABLE ... AS`
  - `SELECT INTO`
  - `CREATE MATERIALIZED VIEW`
  - `REFRESH MATERIALIZED VIEW`
- 查询可能在执行过程中被暂停。只要在系统认为可能发生部分或者增量式执行，就不会产生并行计划。例如：用 `DECLARE CURSOR` 创建的游标将永远不会使用并行计划。类似地，一个 `FOR x IN query LOOP .. END LOOP` 形式的 PL/pgSQL 循环也永远不会使用并行计划，因为当并行查询进行时，并行查询系统无法验证循环中的代码执行起来是安全的。
- 使用了任何被标记为 `PARALLEL UNSAFE` 的函数的查询。大多数系统定义的函数都被标记为 `PARALLEL SAFE`，但是用户定义的函数默认被标记为 `PARALLEL UNSAFE`。参见 Section 15.4 中的讨论。
- 该查询运行在另一个已经存在的并行查询内部。例如，如果一个被并行查询调用的函数自己发出一个 SQL 查询，那么该查询将不会使用并行计划。这是当前实现的一个限制，但是或许不值得移除这个限制，因为它会导致单个查询使用大量的进程。

即使对于一个特定的查询已经产生了并行查询计划，在一些情况下执行时也不会并行执行该计划。如果发生这种情况，那么领导者将会自己执行该计划在 Gather 节点之下的部分，就好像 Gather 节点不存在一样。上述情况将在满足下面的任一条件时发生：

- 因为后台工作者进程的总数不能超过 `max_worker_processes`，导致不能得到后台工作者进程。
- 由于为并行查询目的启动的后台工作者数量不能超过 `max_parallel_workers` 这一限制而不能得到后台工作者。
- 客户端发送了一个执行消息，并且消息中要求取元组的数量不为零。执行消息可见扩展查询协议中的讨论。因为 `libpq` 当前没有提供方法来发送这种消息，所以这种情况只可能发生在不依赖 `libpq` 的客户端中。如果这种情况经常发生，那在它可能发生的会话中设置 `max_parallel_workers_per_gather` 为零是一个很好的主意，这样可以避免产生连续运行时次优的查询计划。

## 15.3. 并行计划

因为每个工作者只执行完成计划的并行部分，所以不可能简单地产生一个普通查询计划并使用多个工作者运行它。每个工作者都会产生输出结果集的一个完全拷贝，因而查询并不会比

普通查询运行得更快甚至还会产生不正确的结果。相反，计划的并行部分一定被查询优化器在内部当作一个部分计划，即它必须被构建出来，这样每一个执行该计划的进程将以无重复地方式产生输出行的一个子集，即保证每一个所需要的输出行正好只被一个合作进程生成。通常，这意味着该查询的驱动表上的扫描必须是一种可并行的扫描。

### 15.3.1. 并行扫描

当前支持下列可并行的表扫描。

- 在一个并行顺序扫描中，表块将在合作进程之间被划分。一次会分发一个块，这样对表的访问还是保持顺序方式。
- 在一个并行位图堆扫描中，一个进程被选为领导者。这个进程执行对一个或者多个索引的扫描并且构建出一个位图指示需要访问哪些表块。这些表块接着会在合作进程之间划分（和并行顺序扫描中一样）。换句话说，堆扫描以并行方式进行但底层的索引扫描不是并行。
- 在一个并行索引扫描或者并行只用索引的扫描中，合作进程轮流从索引读取数据。当前，并行索引扫描仅有B-树索引支持。每一个进程将认领一个索引块并且扫描和返回该索引块引用的所有元组，其他进程可以同时地从一个不同的索引块返回元组。并行B-树扫描的结果会以每个工作者进程内的顺序返回。

其他扫描类型（例如非B-树索引的扫描）可能会在未来支持并行扫描。

### 15.3.2. 并行连接

正如在非并行计划中那样，驱动表可能被使用嵌套循环、哈希连接或者归并连接连接到一个或者多个其他表。连接的内侧可以是任何类型的被规划器支持的非并行计划，假设它能够安全地在并行工作者中运行。根据连接类型，内侧还可以是一种并行计划。

- 在一个嵌套循环连接中，内侧总是非并行的。尽管它会被完全执行，如果内侧是一个索引扫描也会很高效，因为外侧元组以及在索引中查找值的循环会被划分到多个合作进程。
- 在一个归并连接中，内侧总是一个非并行计划并且因此会被完全执行。这可能是不太高效的，特别是在排序必须被执行时，因为在每一个合作进程中工作数据和结果数据是重复的。
- 在一个哈希连接（没有“并行”前缀）中，每个合作进程都会完全执行内侧以构建哈希表的相同拷贝。如果哈希表很大或者该计划开销很大，这种方式就很低效。在一个并行哈希连接中，内侧是一个并行哈希，它把构建共享哈希表的工作划分到多个合作进程。

### 15.3.3. 并行聚集

LightDB通过按两个阶段进行聚集来支持并行聚集。首先，每个参与到查询并行部分的进程执行一个聚集步骤，为该进程注意到的每个分组产生一个部分结果。这在计划中反映为一个Partial Aggregate节点。然后，部分结果通过Gather或者Gather Merge被传输到领导者。最后，领导者对来自所有工作者的结果进行重新聚集得到最终的结果。这在计划中反映为一个Finalize Aggregate节点。

因为Finalize Aggregate节点运行在领导者进程上，如果查询产生的分组数相对于其输入行数来说比较大，则查询规划器不会喜欢它。例如，在最坏的情况下，Finalize Aggregate节点看到的分组数可能与所有工作者进程在Partial Aggregate阶段看到的输入行数一样多。对于这类情况，使用并行聚集显然得不到性能收益。查询规划器会在规划过程中考虑这一点并且不太会在这种情况下选择并行聚集。

并行聚集并非在所有情况下都被支持。每一个聚集都必须是对并行安全的并且必须有一个组合函数。如果该聚集有一个类型为internal的转移状态，它必须有序列化和反序列化函数。更多细节请参考CREATE AGGREGATE。如果任何聚集函数调用包含DISTINCT或ORDER BY子句，

则不支持并行聚集。对于有序集聚集或者当查询涉及GROUPING SETS时，也不支持并行聚集。只有在查询中涉及的所有连接也是该计划并行部分的组成部分时，才能使用并行聚集。

### 15.3.4. 并行Append

只要当LightDB需要从多个源中整合行到一个单一结果集时，它会使用Append或MergeAppend计划节点。在实现UNION ALL或扫描分区表时常常会发生这种情况。就像这些节点可以被用在任何其他计划中一样，它们可以被用在并行计划中。不过，在并行计划中，规划器使用的是Parallel Append节点。

当一个Append节点被用在并行计划中时，每个进程将按照子计划出现的顺序执行子计划，这样所有的参与进程会合作执行第一个子计划直到它被完成，然后同时移动到第二个计划。而在使用Parallel Append时，执行器将把它的子计划尽可能均匀地散布在参与进程中，这样多个子计划会被同时执行。这避免了竞争，也避免了子计划在那些不执行它的进程中产生启动代价。

此外，和常规的Append节点不同（在并行计划中使用时仅有部分子计划），Parallel Append节点既可以有部分子计划也可以有非部分子计划。非部分子计划将仅被单个进程扫描，因为扫描它们不止一次会产生重复的结果。因此涉及到追加多个结果集的计划即使在没有有效的部分计划可用时，也能实现粗粒度的并行。例如，考虑一个针对分区表的查询，它只能通过使用一个不支持并行扫描的索引来实现。规划器可能会选择常规Index Scan计划的Parallel Append。每个索引扫描必须被单一的进程执行完，但不同的扫描可以由不同的进程同时执行。

enable\_parallel\_append可以被用来禁用这种特性。

### 15.3.5. 并行计划小贴士

如果我们想要一个查询能产生并行计划但事实上又没有产生，可以尝试减小parallel\_setup\_cost或者parallel\_tuple\_cost。当然，这个计划可能比规划器优先产生的顺序计划还要慢，但也不总是如此。如果将这些设置为很小的值（例如把它们设置为零）也不能得到并行计划，那就可能是有某种原因导致查询规划器无法为你的查询产生并行计划。可能的原因可见Section 15.3和Section 15.4

在执行一个并行计划时，可以用EXPLAIN (ANALYZE, VERBOSE)来显示每个计划节点在每个工作者上的统计信息。这些信息有助于确定是否所有的工作被均匀地分发到所有计划节点以及从总体上理解计划的性能特点。

## 15.4. 并行安全性

规划器把查询中涉及的操作分类成并行安全、并行受限或者并行不安全。并行安全的操作不会与并行查询的使用产生冲突。并行受限的操作不能在并行工作者中执行，但是能够在并行查询的领导者中执行。因此，并行受限的操作不能出现在Gather或者Gather Merge节点之下，但是能够出现在包含这类节点的计划的其他位置。并行不安全的操作不能在并行查询中执行，甚至不能在领导者中执行。当一个查询包含任何并行不安全操作时，并行查询对这个查询是完全被禁用的。

下面的操作总是并行受限的。

- 公共表表达式（CTE）的扫描。
- 临时表的扫描。
- 外部表的扫描，除非外部数据包装器有一个IsForeignScanParallelSafe API。
- InitPlan所挂接到的计划节点。
- 引用一个相关的SubPlan的计划节点。

## 15.4.1. 为函数和聚集加并行标签

规划器无法自动判定一个用户定义的函数或者聚集是并行安全、并行受限还是并行不安全，因为这需要预测函数可能执行的每一个操作。一般而言，这就相当于一个停机问题，因此是不可能的。甚至对于可以做到判定的简单函数我们也不会尝试，因为那会非常昂贵而且容易出错。相反，除非是被标记出来，所有用户定义的函数都被认为是并行不安全的。在使用CREATE FUNCTION或者ALTER FUNCTION时，可以通过指定PARALLEL SAFE、PARALLEL RESTRICTED或者PARALLEL UNSAFE来设置标记。在使用CREATE AGGREGATE时，PARALLEL选项可以被指定为SAFE、RESTRICTED或者 UNSAFE。

如果函数和聚集会写数据库、访问序列、改变事务状态（即便是临时改变，例如建立一个EXCEPTION块来捕捉错误的 PL/pgsql）或者对设置做持久化的更改，它们一定要被标记为PARALLEL UNSAFE。类似地，如果函数会访问临时表、客户端连接状态、游标、预备语句或者系统无法在工作者之间同步的后端本地状态，它们必须被标记为PARALLEL RESTRICTED。例如，setseed和 random由于后一种原因而是并行受限的。

一般而言，如果一个函数是受限或者不安全的却被标记为安全，或者它实际是不安全的却被标记为受限，把它用在并行查询中时可能会抛出错误或者产生错误的回答。如果 C 语言函数被错误标记，理论上它会展现出完全不明确的行为，因为系统中无法保护自身不受任意 C 代码的影响。但是，在最有可能的情况下，结果不会比其他任何函数更糟糕。如果有疑虑，最好还是标记函数为UNSAFE。

如果在并行工作者中执行的函数要求领导者没有持有的锁，例如读该查询中没有引用的表，那么工作者退出时会释放那些锁（而不是在事务结束时释放）。如果你写了一个这样做的函数并且这种不同的行为对你很重要，把这类函数标记为PARALLEL RESTRICTED以确保它们只在领导者中执行。

注意查询规划器不会为了获取一个更好的计划而考虑延迟计算并行受限的函数或者聚集。所以，如果一个被应用到特定表的WHERE子句是并行受限的，查询规划器就不会考虑对处于计划并行部分的表执行一次扫描。在一些情况中，可以（甚至效率更高）把对表的扫描包括在查询的并行部分并且延迟对WHERE子句的计算，这样它会出现在Gather节点之上。不过，规划器不会这样做。

---

## Part III. 服务器管理

这部份覆盖了LightDB数据库管理员感兴趣的`主题`。包括软件安装、搭建和配置一个服务器、管理用户和数据库以及维护任务。任何想要运行一个LightDB服务器的人（即使是用于个人用途而不是生产环境），应该熟悉这一部分覆盖的主题。

这部份的信息大致上按照一个新用户会阅读的顺序来安排。但是章节都是自组织的并且可以根据需要独立阅读。这一部分的信息也是按照叙事风格组织的。需要一个特定命令的完整描述的读者应该看看Part VI

最开始的几章主要是为了让读者理解时不需要先导知识，这样需要搭建自己的服务器的新用户可以从这部分开始他们的探索。这一部分剩下的内容是有关调优和管理，这些材料假定读者已经熟悉LightDB数据库系统的一般使用。我们鼓励读者阅读Part 和Part I以获得额外信息。

---



---

## Table of Contents

16. 服务器设置和操作 .....	466
16.1. LightDB用户账户 .....	466
16.2. 创建一个数据库集簇 .....	466
16.2.1. 二级文件系统的使用 .....	467
16.2.2. 文件系统 .....	467
16.3. 启动数据库服务器 .....	468
16.3.1. 服务器启动失败 .....	470
16.3.2. 客户端连接问题 .....	471
16.4. 管理内核资源 .....	471
16.4.1. 共享内存和信号量 .....	471
16.4.2. systemd RemoveIPC .....	475
16.4.3. 资源限制 .....	475
16.4.4. Linux 内存过量使用 .....	476
16.4.5. Linux 大页面 .....	477
16.5. 关闭服务器 .....	478
16.6. 升级一个LightDB集簇 .....	479
16.6.1. 通过lt_dumpall升级数据 .....	479
16.6.2. 通过lt_upgrade升级数据 .....	481
16.6.3. 通过复制升级数据 .....	481
16.7. 阻止服务器欺骗 .....	481
16.8. 加密选项 .....	481
16.9. 用 SSL 进行安全的 TCP/IP 连接 .....	482
16.9.1. Basic Setup .....	483
16.9.2. OpenSSL配置 .....	483
16.9.3. 使用客户端证书 .....	483
16.9.4. SSL 服务器文件用法 .....	484
16.9.5. 创建证书 .....	484
16.10. Secure TCP/IP Connections with GSSAPI Encryption .....	486
16.10.1. 基本设置 .....	486
16.11. 使用SSH隧道的安全 TCP/IP 连接 .....	486
16.12. 在Windows上注册事件日志 .....	487
17. 服务器配置 .....	488
17.1. 设置参数 .....	488
17.1.1. 参数名称和值 .....	488
17.1.2. 通过配置文件影响参数 .....	488
17.1.3. 通过SQL影响参数 .....	489
17.1.4. 通过 Shell 影响参数 .....	490
17.1.5. 管理配置文件内容 .....	490
17.2. 文件位置 .....	491
17.3. 连接和认证 .....	492
17.3.1. 连接设置 .....	492
17.3.2. 安全和认证 .....	495
17.3.3. SSL .....	496
17.4. 资源消耗 .....	498
17.4.1. 内存 .....	498
17.4.2. 磁盘 .....	501
17.4.3. 内核资源使用 .....	501
17.4.4. 基于代价的清理延迟 .....	502
17.4.5. 后台写入器 .....	502
17.4.6. 异步行为 .....	503
17.5. 预写式日志 .....	505
17.5.1. 设置 .....	505
17.5.2. 检查点 .....	509
17.5.3. 归档 .....	510
17.5.4. 归档恢复 .....	511

17.5.5.	恢复目标 .....	512
17.6.	复制 .....	514
17.6.1.	发送服务器 .....	514
17.6.2.	主服务器 .....	515
17.6.3.	后备服务器 .....	516
17.6.4.	订阅者 .....	519
17.7.	查询规划 .....	519
17.7.1.	规划器方法配制 .....	519
17.7.2.	规划器代价常量 .....	521
17.7.3.	遗传查询优化 .....	523
17.7.4.	其他规划器选项 .....	524
17.8.	错误报告和日志 .....	526
17.8.1.	在哪里做日志 .....	526
17.8.2.	什么时候记录日志 .....	529
17.8.3.	记录什么到日志 .....	531
17.8.4.	使用 CSV 格式的日志输出 .....	535
17.8.5.	进程标题 .....	536
17.9.	运行时统计数据 .....	537
17.9.1.	查询和索引统计收集器 .....	537
17.9.2.	统计监控 .....	537
17.10.	自动清理 .....	538
17.11.	客户端连接默认值 .....	540
17.11.1.	语句行为 .....	540
17.11.2.	区域和格式化 .....	546
17.11.3.	共享库预载入 .....	551
17.11.4.	其他默认值 .....	552
17.12.	锁管理 .....	553
17.13.	版本和平台兼容性 .....	553
17.13.1.	以前的 LightDB 版本 .....	553
17.13.2.	平台和客户端兼容性 .....	555
17.14.	错误处理 .....	555
17.15.	预置选项 .....	555
17.16.	自定义选项 .....	557
17.17.	开发者选项 .....	557
17.18.	短选项 .....	560
17.19.	Other Options .....	561
18.	客户端认证 .....	563
18.1.	lt_hba.conf 文件 .....	563
18.2.	用户名映射 .....	569
18.3.	认证方法 .....	571
18.4.	信任认证 .....	571
18.5.	口令认证 .....	572
18.6.	GSSAPI 认证 .....	572
18.7.	SSPI 认证 .....	574
18.8.	Ident 认证 .....	574
18.9.	Peer 认证 .....	575
18.10.	LDAP 认证 .....	575
18.11.	RADIUS 认证 .....	578
18.12.	证书认证 .....	579
18.13.	PAM 认证 .....	579
18.14.	BSD 认证 .....	580
18.15.	认证问题 .....	580
19.	数据库角色 .....	582
19.1.	数据库角色 .....	582
19.2.	角色属性 .....	583
19.3.	角色成员关系 .....	584
19.4.	删除角色 .....	585
19.5.	预定义角色 .....	586

19.6.	函数和触发器安全性	587
20.	管理数据库	588
20.1.	概述	588
20.2.	创建一个数据库	588
20.3.	模板数据库	589
20.4.	数据库配置	590
20.5.	销毁一个数据库	590
20.6.	表空间	591
21.	本地化	593
21.1.	区域支持	593
21.1.1.	概述	593
21.1.2.	行为	594
21.1.3.	问题	594
21.2.	排序规则支持	595
21.2.1.	概念	595
21.2.2.	管理排序规则	596
21.3.	字符集支持	600
21.3.1.	被支持的字符集	601
21.3.2.	设置字符集	603
21.3.3.	服务器和客户端之间的自动字符集转换	604
21.3.4.	可用的字符集转换	605
21.3.5.	进一步阅读	609
22.	日常数据库维护工作	610
22.1.	日常清理	610
22.1.1.	清理的基础知识	610
22.1.2.	恢复磁盘空间	610
22.1.3.	更新规划器统计信息	611
22.1.4.	更新可见性映射	612
22.1.5.	防止事务 ID 回卷失败	613
22.1.6.	自动清理后台进程	615
22.2.	日常重建索引	616
22.3.	日志文件维护	617
23.	备份和恢复	619
23.1.	SQL转储	619
23.1.1.	从转储中恢复	619
23.1.2.	使用lt_dumpall	620
23.1.3.	处理大型数据库	621
23.2.	文件系统级别备份	622
23.3.	连续归档和时间点恢复 (PITR)	623
23.3.1.	建立WAL归档	623
23.3.2.	制作一个基础备份	625
23.3.3.	使用低级API制作一个基础备份	626
23.3.4.	使用一个连续归档备份进行恢复	629
23.3.5.	时间线	630
23.3.6.	建议和例子	631
23.3.7.	警告	632
24.	高可用、负载均衡和复制	633
24.1.	不同方案的比较	633
24.2.	日志传送后备服务器	635
24.2.1.	规划	636
24.2.2.	后备服务器操作	636
24.2.3.	为后备服务器准备主控机	637
24.2.4.	建立一个后备服务器	637
24.2.5.	流复制	637
24.2.6.	复制槽	639
24.2.7.	级联复制	640
24.2.8.	同步复制	640
24.2.9.	在后备机上连续归档	643

24.3.	故障转移 .....	643
24.4.	日志传送的替代方法 .....	644
24.4.1.	实现 .....	645
24.4.2.	基于记录的日志传送 .....	645
24.5.	热备 .....	645
24.5.1.	用户概览 .....	646
24.5.2.	处理查询冲突 .....	647
24.5.3.	管理员概览 .....	649
24.5.4.	热备参数参考 .....	651
24.5.5.	警告 .....	651
25.	监控数据库活动 .....	652
25.1.	标准 Unix 工具 .....	652
25.2.	统计收集器 .....	653
25.2.1.	统计收集配置 .....	653
25.2.2.	查看统计信息 .....	654
25.2.3.	pg_stat_activity .....	657
25.2.4.	pg_stat_replication .....	667
25.2.5.	pg_stat_wal_receiver .....	668
25.2.6.	pg_stat_subscription .....	669
25.2.7.	pg_stat_ssl .....	670
25.2.8.	pg_stat_gssapi .....	671
25.2.9.	pg_stat_archiver .....	671
25.2.10.	pg_stat_bgwriter .....	671
25.2.11.	pg_stat_database .....	672
25.2.12.	pg_stat_database_conflicts .....	673
25.2.13.	pg_stat_all_tables .....	674
25.2.14.	pg_stat_all_indexes .....	675
25.2.15.	pg_statio_all_tables .....	676
25.2.16.	pg_statio_all_indexes .....	677
25.2.17.	pg_statio_all_sequences .....	677
25.2.18.	pg_stat_user_functions .....	677
25.2.19.	pg_stat_slru .....	678
25.2.20.	Statistics Functions .....	678
25.3.	查看锁 .....	680
25.4.	Progress Reporting .....	680
25.4.1.	ANALYZE Progress Reporting .....	681
25.4.2.	CREATE INDEX Progress Reporting .....	682
25.4.3.	VACUUM进度报告 .....	683
25.4.4.	CLUSTER进度报告 .....	685
25.4.5.	基础备份进度报告 .....	686
25.5.	动态追踪 .....	687
25.5.1.	动态追踪的编译 .....	687
25.5.2.	内建探针 .....	687
25.5.3.	使用探针 .....	692
25.5.4.	定义新探针 .....	693
26.	监控磁盘使用 .....	695
26.1.	判断磁盘用量 .....	695
26.2.	磁盘满失败 .....	696
27.	可靠性和预写式日志 .....	697
27.1.	可靠性 .....	697
27.2.	预写式日志 (WAL) .....	698
27.3.	异步提交 .....	699
27.4.	WAL配置 .....	700
27.5.	WAL内部 .....	702
28.	逻辑复制 .....	704
28.1.	发布 .....	704
28.2.	订阅 .....	705
28.2.1.	复制槽管理 .....	705

---

28.3.	冲突 .....	706
28.4.	限制 .....	706
28.5.	架构 .....	707
28.5.1.	初始快照 .....	707
28.6.	监控 .....	707
28.7.	安全性 .....	707
28.8.	配置设置 .....	708
28.9.	快速设置 .....	708
29.	即时编译 (JIT) .....	709
29.1.	什么是JIT编译? .....	709
29.1.1.	JIT加速的操作 .....	709
29.1.2.	内联 .....	709
29.1.3.	优化 .....	709
29.2.	什么时候会用JIT? .....	709
29.3.	配置 .....	710
29.4.	可扩展性 .....	711
29.4.1.	对扩展的内联支持 .....	711
29.4.2.	可插拔的JIT提供者 .....	711
30.	回归测试 .....	712
30.1.	运行测试 .....	712
30.1.1.	在一个临时安装上运行测试 .....	712
30.1.2.	在一个现有安装上运行测试 .....	712
30.1.3.	附加测试套件 .....	713
30.1.4.	区域和编码 .....	714
30.1.5.	额外测试 .....	715
30.1.6.	测试热备 .....	715
30.2.	测试评估 .....	715
30.2.1.	错误消息差异 .....	716
30.2.2.	区域差异 .....	716
30.2.3.	日期和时间差异 .....	716
30.2.4.	浮点差异 .....	716
30.2.5.	行序差异 .....	716
30.2.6.	栈深度不足 .....	717
30.2.7.	“随机”测试 .....	717
30.2.8.	配置参数 .....	717
30.3.	变体比较文件 .....	717
30.4.	TAP 测试 .....	718
30.5.	测试覆盖检查 .....	718
31.	透明数据加密 .....	720

---

## Chapter 16. 服务器设置和操作

本章讨论如何设置和运行数据库服务器，以及它与操作系统的交互。

本章中的说明假设您使用的是纯 `LightDB`，没有任何额外的基础设施，例如，您根据前面章节中的说明从源代码构建的副本。如果您使用的是预先打包或供应商提供的 `LightDB` 版本，那么打包者很可能已经根据您的系统约定为安装和启动数据库服务器做出了特殊规定。有关详细信息，请参阅包级文档。

### 16.1. LightDB用户账户

和对外部世界可访问的任何服务器守护进程一样，我们也建议在一个独立的用户账户下运行 `LightDB`。这个用户账户应该只拥有被该服务器管理的数据，并且应该不能被其他守护进程共享（例如，使用用户 `nobody` 是一个坏主意）。特别是，建议 `LightDB` 不为此用户帐户所有，以确保受感染的服务器进程无法修改这些可执行文件。

`LightDB` 的预打包版本通常会在包安装期间自动创建一个合适的用户帐户。

要在你的系统中增加一个 `Unix` 用户账户，查看一个命令 `useradd` 或 `adduser`。通常会用 `lightdb`（本书中也假定用这个账户），但是你可以使用另一个名称。

### 16.2. 创建一个数据库集簇

在你做任何事情之前，你必须在磁盘上初始化一个数据库存储区域。我们称之为一个数据库集簇（SQL标准使用的术语是目录集簇）。一个数据库集簇是被一个运行数据库服务器的单一实例所管理的多个数据库的集合。在初始化之后，一个数据库集簇将包含一个名为 `postgres` 的数据库，它表示被功能、用户和第三方应用所使用的默认数据库。数据库服务器本身并不要求 `postgres` 数据库存在。另一个在初始化过程中为每一个集簇创建的数据库被称为 `template1`。顾名思义，它将被用于创建后续数据库的模板；它不应该被用于实际工作（在集簇内创建新数据库的更多信息请见 [Chapter 20](#)）。

在文件系统术语中，一个数据库集簇是一个单一目录，所有数据都将被存储在其中。我们称它为数据目录或数据区域。在哪里存储你的数据完全由你选择。没有默认的位置，不过 `/usr/local/lightdb-x/data` 或 `/var/lib/lightdb-x/data` 位置比较流行。数据目录必须在使用前初始化，必须使用与 `LightDB` 一起安装的程序 `initdb`。

如果您使用的是 `LightDB` 的预打包版本，它可能有一个特定的约定来放置数据目录的位置，并且它还可能提供用于创建数据目录的脚本。在这种情况下，您应该优先使用该脚本而不是直接运行 `initdb`。有关详细信息，请参阅包级文档。

要手动初始化数据库集群，请运行 `initdb` 并使用 `-D` 选项指定所需的数据库集群文件系统位置，例如：

```
$ initdb -D /usr/local/lightdb-x/data
```

请注意，您必须在登录 `LightDB` 用户帐户时执行此命令，这在上一节中进行了描述。

#### Tip

作为 `-D` 选项的一种替换方案，你可以设置环境变量 `LTDATA`。

另一种替代方案是，你可以通过 `lt_ctl` 程序来运行 `initdb`：

```
$ lt_ctl -D /usr/local/lightdb-x/data initdb
```

如果你使用`lt_ctl`来启停服务器（见Section 16.3，这种方法可能更直观，以为这样`lt_ctl`将是你用来管理数据库服务器实例的唯一命令。

如果你指定的目录还不存在，`initdb`将尝试创建它。当然，如果`initdb`没有在父目录中的写权限，这将会失败。通常推荐让LightDB用户拥有数据目录及其父目录，这样就不存在上面的问题了。如果想要的父目录也不存在，你将需要先创建它，如果父目录不可写则使用`root`特权。因此，该过程可能像这样：

```
root# mkdir /usr/local/pgsql
root# chown lightdb /usr/local/pgsql
root# su lightdb
lightdb$ initdb -D /usr/local/lightdb-x/data
```

如果数据目录存在并且已经包含文件，`initdb`将拒绝运行。这可以避免无意中覆盖一个已有的安装。

因为数据目录包含所有存储在数据库里的数据，所以最重要的是保护这个目录不受未授权的访问。因此，`initdb`会回收禁止除LightDB用户，也可以选择组，之外所有用户的访问权限。当组访问启用时，是只读的。它允许相同组中未被授权的用户作为集簇属主，备份集簇数据或者执行其他只需要读访问权限的操作。

注意在现有集群启用或禁用组访问时，需要关闭集群，且重新启动LightDB之前设置所有的目录和文件到恰当的模式。否则，数据目录中会存在多种模式。集群仅可以被其属主访问，恰当的模式应该是，其目录设置为0700，普通文件设置为0600。允许集群被组可读，恰当的模式应该是，其目录设置为0750，普通文件设置为0640。

不过，虽然目录的内容是安全的，但默认的客户端认证设置允许任意本地用户连接到数据库甚至成为数据库超级用户。如果你不信任其他本地用户，我们建议你使用`initdb`的`-W`、`--pwprompt`或`--pwfile`选项之一给数据库超级用户赋予一个口令。还可以指定`-A md5`或`-A password`，这样就不会使用默认的`trust`身份认证。或者在执行`initdb`之后、第一次启动服务器之前修改生成的`lt_hba.conf`文件（另外一些可行的方法包括peer认证或者用文件系统权限限制连接。更多信息见Chapter 18）。

`initdb`同时也为数据库集簇初始化默认区域。通常，它将只是使用环境中的区域设置并且把它们应用于被初始化的数据库。可以为数据库指定一个不同的区域；有关于此的更多信息可以在Section 21.中找到。特定数据库集簇中使用的默认排序顺序是通过`initdb`设置的，虽然你可以创建使用不同排序顺序的新数据库，但在`initdb`创建的模板数据库中使用的顺序不能更改（除非删除并重建它们）。使用非C或POSIX的区域还会对性能造成影响。因此，第一次就正确地选择很重要。

`initdb`还为数据库集簇设置默认的字符集编码。通常字符集编码应该选择与区域设置匹配。详见Section 21.3

非C以及非POSIX区域对于字符集排序依赖于操作系统的排序规则库。这控制着索引中存储的键的排序。为此，通过快照恢复、二进制流复制、更换不同的操作系统或者升级操作系统都不能把一个集簇切换到一种不兼容的排序规则库版本。

## 16.2.1. 二级文件系统的使用

很多安装会在文件系统（卷）而不是机器的“根”卷上创建它们的数据库集簇。如果你选择这样做，我们不建议尝试使用二级卷的顶层目录（挂载点）作为数据目录。最好的做法是在LightDB用户拥有的挂载点目录中创建一个目录，然后在其中创建数据目录。这可以避免权限问题，特别是对于`lt_upgrade`这类操作，并且它也能在二级卷被断线后确保干净的失败。

## 16.2.2. 文件系统

通常来说，任何具有POSIX语义的文件系统都可用于LightDB。用户出于各种原因会喜欢不同的文件系统，包括供应商支持、性能和熟悉程度等。经验表明，在其他方面相同的情况

下，人们不应期望仅仅通过切换文件系统或进行较小的文件系统配置调整以造成明显的性能或行为更改。

### 16.2.2.1. NFS

可以用 NFS 文件系统来存储 LightDB 数据目录。LightDB 没有特殊对待 NFS 文件系统，这意味着它假定 NFS 的行为与本地连接的驱动器完全一样。LightDB 不使用任何已知在 NFS 上非标准的功能，例如文件锁定。

在 LightDB 使用 NFS 的唯一硬性要求是，文件系统需要使用 `hard` 选项挂载。使用 `hard` 选项，如果存在网络问题，进程可以无限期地“hang”，因此这个配置需要仔细的监视设置。`soft` 选项将在出现网络问题时中断系统调用，但 LightDB 不会重复系统呼叫以这种方式中断，因此任何此类中断都将导致报告 I/O 错误。

`sync` 挂载选项不是必须的。`async` 选项的特性是足够的，因为 LightDB 发出 `fsync` 调用，会在适当的时候刷新写缓存。（这与它在本地文件系统上的工作方式相类似。）但是，强烈建议在部署 NFS server 的系统上使用 `sync` 导出选项（主要是 Linux）。否则，NFS 客户端上的 `fsync` 或等效项实际上不能保证实现服务器上的永久存储，这可能会导致损坏，类似于使用参数 `fsync` 关闭。这些挂载和导出选项的默认值在供应商和版本之间有所不同，因此建议在任何情况下检查并明确指定它们，以避免任何歧义。

在某些情况下，可以通过 NFS 或较低级别的协议，例如 iSCSI，访问外部存储产品。在后一种情况下，存储显示为块设备，可以在其上创建任何可用的文件系统。这种方法可能会使 DBA 不必处理 NFS 的一些特性，但是在其他级别当然会出现管理远程存储的复杂性。

## 16.3. 启动数据库服务器

在任何人可以访问数据库前，你必须启动数据库服务器。数据库服务器程序是 `lightdb`

如果您使用的是 LightDB 的预打包版本，那么几乎可以肯定的是，它包含了根据操作系统的约定将服务器作为后台任务运行的规定。使用包的基础设施来启动服务器比自己弄清楚如何启动要简单得多。有关详细信息，请参阅包级文档。

手动启动服务器的基本方法是直接调用 `lightdb`，使用 `-D` 选项指定数据目录的位置，例如：

```
$ lightdb -D /usr/local/lightdb-x/data
```

这将把服务器放在前台运行。这个步骤同样必须以 LightDB 用户帐户登录来操作。如果没有 `-D` 选项，服务器将尝试使用环境变量 `LTDATA` 命名的目录。如果这个环境变量也没有提供则导致失败。

通常最好在后台启动 `lightdb`。要这样做，使用常用的 Unix shell 语法：

```
$ lightdb -D /usr/local/lightdb-x/data >logfile 2>&1 &
```

如上所示，把服务器的 `stdout` 和 `stderr` 输出存储到某个地方是非常重要的。这将对审计目的和诊断问题有所帮助（更深入的有关日志文件处理的讨论请见（Section 22.3）。

`lightdb` 还接受其它一些命令行选项。更多的信息请见 `lightdb` 参考页 和下面的 Chapter 17

这些 shell 语法很容易让人觉得无聊。因此我们提供了包装器程序 `lt_ctl` 以简化一些任务。例如：

```
lt_ctl start -l logfile
```

将在后台启动服务器并且把输出放到指定的日志文件中。`-D` 选项和 `lightdb` 中的一样。`lt_ctl` 还可以用于停止服务器。



通常，你会希望在计算机启动的时候启动数据库服务器。自动启动脚本是操作系统相关的。在 `contrib/start-scripts` 目录中有一些随 LightDB 分发的示例脚本。安装将需要 `root` 权限。

不同的系统在引导时有不同的启动守护进程的习惯。许多系统有一个文件 `/etc/rc.local` 或 `/etc/rc.d/rc.local`。其他的使用 `init.d` 或 `rc.d` 目录。不管你做什么，服务器必须由 LightDB 用户账户而不是 `root` 或任何其他用户启动。因此你可能应该在你的命令中使用 `su lightdb -c '...'` 这种形式。例如：

```
su lightdb -c 'lt_ctl start -D /usr/local/lightdb-x/data -l serverlog'
```

下面是一些更加与操作系统相关的建议（在每一种情况中要确保在我们展示通用值的地方使用正确的安装目录和用户名）。

- 对于 FreeBSD，找找 LightDB 源码发布中的文件 `contrib/start-scripts/freebsd`。
- 在 OpenBSD 上，把下面几行加到 `/etc/rc.local` 文件中：

```
if [ -x /usr/local/lightdb-x/bin/lt_ctl -a -x /usr/local/lightdb-x/bin/
lightdb ]; then
    su -l lightdb -c '/usr/local/lightdb-x/bin/lt_ctl start -s -l /var/
lightdb/log -D /usr/local/lightdb-x/data'
    echo -n ' lightdb'
fi
```

- 在 Linux 系统上将

```
/usr/local/lightdb-x/bin/lt_ctl start -l logfile -D /usr/local/lightdb-x/data
```

加入到 `/etc/rc.d/rc.local` 或 `/etc/rc.local` 中，还可以在 LightDB 的源码发布中找找文件 `contrib/start-scripts/linux`。

在使用 `systemd` 时，可以使用下面的服务单元文件（例如 `/etc/systemd/system/postgresql.service`）：

```
[Unit]
Description=LightDB database server
Documentation=man:postgres(1)

[Service]
Type=notify
User=lightdb
ExecStart=/usr/local/lightdb-x/bin/lightdb -D /usr/local/lightdb-x/data
ExecReload=/bin/kill -HUP $MAINPID
KillMode=mixed
KillSignal=SIGINT
TimeoutSec=infinity

[Install]
WantedBy=multi-user.target
```

使用 `Type=notify` 要求服务器的二进制文件使用 `configure --with-systemd` 编译。

要仔细地考虑超时设置。在写作这份文档时，`systemd` 的默认超时时长是 90 秒，并且将会杀死没有在这段时间内报告准备好的进程。但是 LightDB 服务器可能因为执行崩溃恢复而导致启动过程大大超过这个默认时间。建议的值是 0 禁用超时逻辑。

- 在NetBSD上，你可以根据爱好选择FreeBSD或Linux的启动脚本。
- 在Solaris上，创建一个名为/etc/init.d/lightdb的文件，其中包含下列行：

```
su - lightdb -c "/usr/local/lightdb-x/bin/lt_ctl start -l logfile -D /usr/local/lightdb-x/data"
```

然后在/etc/rc3.d中创建一个符号链接S99postgresql指向它。

当服务器在运行时，它的PID被保存在数据目录中的postmaster.pid文件。这样做可以防止多个服务器实例运行在同一个数据目录中，并且也可以被用来关闭服务器。

### 16.3.1. 服务器启动失败

有几个常见的原因会导致服务器启动失败。通过检查服务器日志或使用手工启动的方法（不做标准输出或标准错误的重定向），就可以看到出现什么错误消息。下面我们详细地解释一些最常见的错误消息。

```
LOG:  could not bind IPv4 address "127.0.0.1": Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few
seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

正如这个消息所说的，这表示：你试图在一个已经有服务器运行着的端口上再启动另一个服务器。不过，如果核心错误消息不是Address already in use或其变体，那就有可能是别的问题。例如，试图在一个被保留的端口上启动服务器会收到下面这样的消息：

```
$ lightdb -p 666
LOG:  could not bind IPv4 address "127.0.0.1": Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few
seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

像这样的消息：

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

可能意味着你的内核对共享内存区的限制小于LightDB试图创建的工作区域（本例中是4011376640字节）。这只有在您将shared\_memory\_type设置为sysv时才有可能发生。在这种情况下，你可以试着以小于正常数量的缓冲区（shared\_buffers）启动服务器，或者重新配置内核以增加共享内存允许的尺寸。当你试图在同一台机器上启动多个服务器，并且它们所需的总空间超过了内核的限制，也会报这个错。

一个这样的错误：

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

并不意味着你已经用光了磁盘空间。它的意思是你的内核对System V信号量的限制小于LightDB想创建的数量。和上面一样，你可以通过减少允许的连接数（max\_connections）来绕过这个限制，但最终你还是会希望提高内核的限制。

关于配置System V IPC功能的细节请见Section 16.4.1

## 16.3.2. 客户端连接问题

尽管可能在客户端出现的错误情况范围宽广而且是应用相关的，但的确有几种与服务器的启动方式直接相关。除了下面提到的几种错误之外的问题都应该在相应的客户端应用文档中。

```
ltsql: error: connection to server at "server.joe.com" (123.123.123.123), port
5432 failed: Connection refused
    Is the server running on that host and accepting TCP/IP connections?
```

这是常见的“I couldn’t find a server to talk to”失败。上面的情况看起来是发生在尝试 TCP/IP 通信时。常见的错误是忘记把服务器配置成允许 TCP/IP 连接。

另外，当试图通过 Unix 域套接字与本地服务器通信时，你会看到这个：

```
ltsql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: No
such file or directory
    Is the server running locally and accepting connections on that socket?
```

如果服务器确实在运行，检查客户端的套接字路径(这里是 `/tmp`)的理解与服务器的 `unix_socket_directories` 设置的一致性。

连接失败信息总是显示服务器地址或套接字路径名，有助于验证客户端是不是尝试连接到正确的位置。如果实际上没有服务器在那里监听，典型的核心错误消息将是 `Connection refused` 或 `No such file or directory` (值得注意的是这种环境中的 `Connection refused` 并不表示服务器得到了你的连接请求并拒绝了它。那种情况会产生一个不同的消息，如 Section 18.1 中所示)。其它像 `Connection timed out` 这样的消息可能表示更基础的问题，如缺少网络连接，或者有防火墙阻塞了连接。

## 16.4. 管理内核资源

LightDB 某些时候会耗尽操作系统的各种资源限制，当同一个系统上运行着多个拷贝的服务器或在一个非常大的安装中时尤其如此。本节解释了 LightDB 使用的内核资源以及你可以采取的用于解决内核资源消耗相关问题的步骤。

### 16.4.1. 共享内存和信号量

LightDB 需要操作系统提供进程间通信 (IPC) 特性，特别是共享内存和信号量。Unix 驱动的系统通常提供 “System V” IPC、“POSIX” IPC，或者两者都有。Windows 有它自己的这些功能的实现，这里不讨论。

默认情况下，LightDB 分配很少量的 System V 共享内存，和大量的匿名的 `mmap` 共享内存。或者，可以使用单个大型 System V 共享内存区域。(参见 `shared_memory_type`)。另外，在服务器启动时会创建大量信号量，这些信号量可以是 System V 或 POSIX 风格。目前，POSIX 信号量用于 Linux 和 FreeBSD 系统，而其他平台则使用 System V 信号量。

System V IPC 特性通常受系统范围分配限制的限制。当 LightDB 超出了这些限制之一时，服务器会拒绝启动并且留下一条有指导性的错误消息，其中描述了问题以及应该怎么做 (又见 Section 16.3.1)。相关的内核参数在不同系统之间的命名方式一致，Table 16.1 给出了一个概述。不过，设置它们的方法却多种多样。下面给出了对于某些平台的建议：

Table 16.1. System V IPC 参数

名称	描述	运行一个 LightDB 实例所需的值
SHMMAX	共享内存段的最大尺寸 (字节)	至少 1kB，但是默认值通常要高一些
SHMMIN	共享内存段的最小尺寸 (字节)	1

名称	描述	运行一个LightDB实例所需的值
SHMALL	可用共享内存的总量（字节或页面）	如果是字节，同SHMMAX；如果是页面，为 $\text{ceil}(\text{SHMMAX}/\text{PAGE\_SIZE})$ ，加上其他应用程序的空间
SHMSEG	每个进程的最大共享内存段数目	只需要 1 段，但是默认值高很多
SHMMNI	系统范围内的最大共享内存段数目	像SHMSEG外加其他应用的空间
SEMMNI	信号量标识符（即，集合）的最大数目	$\text{at least } \text{ceil}((\text{max\_connections} + \text{autovacuum\_max\_workers} + \text{max\_wal\_senders} + \text{max\_worker\_processes} + 5) / 16)$ plus room for other applications
SEMMNS	系统范围内的最大信号量数目	$\text{ceil}((\text{max\_connections} + \text{autovacuum\_max\_workers} + \text{max\_wal\_senders} + \text{max\_worker\_processes} + 5) / 16) * 17$ plus room for other applications
SEMMSL	每个集合中信号量的最大数目	至少 17
SEMMAP	信号量映射中的项数	见文本
SEMMVMX	信号量的最大值	至少 1000（默认值常常是 32767，如非必要不要更改）

LightDB要求少量字节的 System V 共享内存（在 64 位平台上通常是 48 字节）用于每一个服务器拷贝。在大多数现代操作系统上，这个量很容易得到。但是，如果你运行了很多个服务器副本，或者显式配置服务器以使用大量 System V 共享内存（参见 `shared_memory_type` 和 `dynamic_shared_memory_type`），可能需要增加SHMALL（系统范围内 System V 共享内存的总量）。注意在很多系统上SHMALL是以页面而不是字节来度量。

不太可能出问题的是共享内存段的最小尺寸（SHMMIN），对LightDB来说应该最多大约是 32 字节（通常只是1）。而系统范围（SHMMNI）或每个进程（SHMSEG）的最大共享内存段数目不太可能会导致问题，除非你的系统把它们设成零。

当使用System V信号量时，LightDB对每个允许的连接（`max_connections`）、每个允许的自动清理工作者进程（`autovacuum_max_workers`）和每个允许的后台进程（`max_worker_processes`）使用一个信号量，以16个为一个集合。每个这种集合还包含第 17 个信号量，其中存储一个“magic number”，以检测和其它应用使用的信号量集合的冲突。系统里的最大信号量数目是由SEMMNS设置的，因此这个值必须至少和`max_connections`加`autovacuum_max_workers`加`max_wal_senders`加`max_worker_processes`一样大，并且每 16 个连接外加工作者还要另外加一个（见Table 16. 中的公式）。参数SEMMNI 决定系统中同一时刻可以存在的信号量集合的数目限制。因此这个参数必须至少为 $\text{ceil}((\text{max\_connections} + \text{autovacuum\_max\_workers} + \text{max\_wal\_senders} + \text{max\_worker\_processes} + 5) / 16)$ 。降低允许的连接数目是一种临时的绕开失败（来自函数`semget`）的方法，通常使用让人混乱的措辞“No space left on device”。

在某些情况下可能还有必要增大SEMMAP，使之至少与SEMMNS相近。如果系统有这个参数（很多系统没有），这个参数定义信号量资源映射的尺寸，在其中每个连续的可用信号量块都需要一项。每当一个信号量集合被释放，那么它要么会被加入到该与被释放块相邻的一个现有项，或者它会被注册在一个新映射项中。如果映射被填满，被释放的信号量将丢失（直到重启）。因此信号量空间的碎片时间长了会导致可用的信号量比应有的信号量少。

与“semaphore undo”有关的其他各种设置，如SEMMNU和SEMUME 不会影响LightDB。

当使用POSIX信号量时，所需的信号量数量与System V相同，即每个允许的连接（`max_connections`）、允许的自动清理工作进程（`autovacuum_max_workers`）和允许的后台进程（`max_worker_processes`）一个信号量。在首选此选项的平台上，POSIX信号量的数量没有特定的内核限制。

## AIX

不再需要对这些参数（例如SHMMAX）做任何特殊的配置，这看起来就像是配置成允许所有内存都被用作共享内存。这是一种通常被用于其他数据库（DB/2）的配置。

但是，可能需要修改/etc/security/limits中的全局ulimit信息，默认的文件尺寸硬限制（fsize）和文件数量（nofiles）可能太低。

## FreeBSD

默认共享内存设置通常就足够了，除非您已将 `shared_memory_type` 设置为 `sysv`。在此平台上不使用 System V 信号量。

可以使用 `sysctl` 或 `loader` 接口来改变默认 IPC 配置。下列参数可以使用 `sysctl` 设置：

```
# sysctl kern.ipc.shmall=32768
# sysctl kern.ipc.shmmax=134217728
```

要让这些设置在重启之后也保持，请修改 `/etc/sysctl.conf`。

如果您已将 `shared_memory_type` 设置为 `sysv`，您可能还想配置您的内核以将 System V 共享内存锁定到 RAM 中并防止它被调出以进行交换。这可以使用 `sysctl` 设置 `kern.ipc.shm_use_phys` 来完成。

如果在 FreeBSD jail 中运行，您应该将它的 `sysvshm` 参数设置为 `new`，这样它就有自己独立的 System V 共享内存命名空间。（在 FreeBSD 11.0 之前，有必要从 jails 启用对主机 IPC 命名空间的共享访问，并采取措施避免冲突。）

## NetBSD

默认共享内存设置通常就足够了，除非您已将 `shared_memory_type` 设置为 `sysv`。作为 NetBSD 的默认设置，你总是会想调大 `kern.ipc.semni` 和 `kern.ipc.semns` 的值，因为他们实在太小了。

可以使用 `sysctl` 调整 IPC 参数，例如：

```
# sysctl -w kern.ipc.semni=100
```

要使这些设置在重新启动后保持不变，请修改 `/etc/sysctl.conf`。

如果已将 `shared_memory_type` 设置为 `sysv`，则可能还需要将内核配置为将 System V 共享内存锁定到 RAM 中，并防止将其调出以进行交换。这可以使用 `sysctl` 设置 `kern.ipc.shm_use_phys` 来完成。

## OpenBSD

默认共享内存设置通常就足够了，除非您已将 `shared_memory_type` 设置为 `sysv`。作为 OpenBSD 的默认配置，你总是会想调大 `kern.semni` 和 `kern.semns` 的值，因为他们实在太小了。

可以使用 `sysctl` 调整 IPC 参数，例如：

```
# sysctl kern.semni=100
```

要使这些设置在重新启动后保持不变，请修改 `/etc/sysctl.conf`。

## HP-UX

默认的设置可以满足正常的安装。

IPC参数可以在Kernel Configuration → Configurable Parameters下的System Administration Manager (SAM) 中被设置。当你完成时选择Create A New Kernel。

## Linux

默认的共享内存设置通常已经足够好了，除非您将shared\_memory\_type设置为sysv，即使这样，也只适用于低默认值的旧内核版本。System V信号量不在此平台上使用。

共享内存大小设置可以通过sysctl界面更改。例如，允许 16 GB：

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

要使这些设置在重新启动时保持不变，请参见/etc/sysctl.conf。

## macOS

默认共享内存和信号量设置通常就足够了，除非您已将 shared\_memory\_type 设置为 sysv。

在 macOS 中配置共享内存的推荐方法是创建一个名为/etc/sysctl.conf的文件，其中包含这样的变量赋值：

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

注意在某些 macOS 版本中，所有五个共享内存参数必须在/etc/sysctl.conf中设置，否则值将会被忽略。

SHMMAX 只能设置为 4096 的倍数。

在这个平台上，SHMALL以 4kB 的页面度量。

可以使用 sysctl 动态更改除 SHMMNI 之外的所有内容。但是最好还是通过/etc/sysctl.conf来设置你喜欢的值，这样重启之后这些值还能被保持。

## Solaris

### illumos

默认的共享内存和信号量设置对于大多数 LightDB 应用程序来说通常已经足够了。Solaris 现在将SHMMAX的默认值设置为系统 RAM的四分之一。要进一步调整这个设置，使用与lightdb用户有关的一个项目设置。例如，以root运行下列命令：

```
projadd -c "LightDB DB User" -K "project.max-shm-memory=(privileged,8GB,deny)" -U lightdb -G lightdb user.lightdb
```

这个命令增加user.lightdb项目并且将用于lightdb用户的最大共享内存设置为 8GB，并且在下次用户登录进来时或重启LightDB（不是重新载入）时生效。上述假定LightDB是由lightdb组中的lightdb用户所运行。不需要重新启动服务器。

对于将有巨大数量连接的数据库服务器，我们推荐的其他内核设置修改是：

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
```

```
project.max-msg-ids=(priv, 4096, deny)
```

此外，如果你正在在一个区中运行LightDB，你可能也需要提升该区的资源使用限制。更多关于projects 和prctl的信息请见System Administrator's Guide中的“Chapter2: Projects and Tasks”。

## 16.4.2. systemd RemoveIPC

如果正在使用systemd，则必须注意IPC资源（包括共享内存）不会被操作系统过早删除。从源代码安装LightDB时，这尤其值得关注。LightDB发布包的用户不太可能受到影响，因为lightdb用户通常是作为系统用户创建的。

控制当用户完全退出时是否移除IPC对象。系统用户免除。此设置在死板的systemd中默认为on，但某些操作系统分配默认为关闭。

当此设置打开时，典型的观察效果是用于并行查询执行的共享内存对象在明显随机的时间被删除，启用此设置时观察到的典型效果是用于并行查询执行的共享内存对象在明显随机的时间被删除，导致在尝试打开和删除它们时出现错误和警告，例如

```
WARNING: could not remove shared memory segment "/LightDB.1450751626": No such
file or directory
```

不同类型的IPC对象（共享内存与信号量，System V与POSIX）在systemd 中略有不同，因此可能会发现某些IPC资源不会像其他IPC资源一样被删除。但依靠这些微妙的差异是不可取的。

“注销用户”可能会作为维护工作的一部分发生，或者当管理员以 lightdb用户或类似名称登录时手动发生，所以通常难以防止。

什么是“系统用户”是由/etc/login.defs中的 SYS\_UID\_MAX设置在systemd编译时确定的。

打包和部署脚本应该小心，通过使用useradd -r、 adduser --system或等价物来创建lightdb用户作为系统用户。

或者，如果用户帐户创建不正确或无法更改，建议设置

```
RemoveIPC=no
```

在/etc/systemd/logind.conf或其他适当的配置文件中。

### Caution

至少要确保这两件事中的一件，否则LightDB服务器将非常不可靠。

## 16.4.3. 资源限制

Unix类操作系统强制了许多种资源限制，这些限制可能干扰你的LightDB服务器的操作。尤其重要的是对每个用户的进程数目的限制、每个进程打开文件数目的限制以及每个进程可用的内存的限制。这些限制中每个都有一个“硬”限制和一个“软”限制。实际使用的是软限制，但用户可以自己修改成最大为硬限制的数目。而硬限制只能由root用户修改。系统调用setrlimit负责设置这些参数。shell的内建命令ulimit (Bourne shells) 或limit (csh) 被用来从命令行控制资源限制。在BSD衍生的系统上，/etc/login.conf文件控制在登录期间设置的各种资源限制。详见操作系统文档。相关的参数是maxproc、openfiles和datasize。例如：

```
default:\
...
      :datasize-cur=256M:\
      :maxproc-cur=256:\
      :openfiles-cur=256:\
...
```

(-cur是软限制。增加-max可设置硬限制)。

内核也可以在某些资源上有系统范围的限制。

- 在Linux上，`/proc/sys/fs/file-max`决定内核可以支持打开的最大文件数。你可以通过往该文件写入一个不同的数值修改此值，或者通过在`/etc/sysctl.conf`中增加一个赋值来修改。每个进程的最大打开文件数限制是在编译内核的时候固定的；更多信息请见`/usr/src/linux/Documentation/proc.txt`。

LightDB服务器为每个连接都使用一个进程，所以你应该至少和允许的连接同样多的进程，再加上系统其它部分所需要的进程数目。通常这个并不是什么问题，但如果你在一台机器上运行多个服务器，资源使用可能会紧张。

打开文件的出厂默认限制通常设置为“socially friendly”的值，它允许许多用户在一台机器上共存，而不会导致不成比例的系统资源使用。如果你在一台机器上运行许多服务器，这也许就是你想要的，但是在专门的服务器上，你可能需要提高这个限制。

在另一方面，一些系统允许独立的进程打开非常多的文件；如果不止几个进程这么干，那系统范围的限制就很容易被超过。如果你发现这样的现象，并且不想修改系统范围的限制，你就可以设置LightDB的`max_files_per_process`配置参数来限制打开文件数的消耗。

## 16.4.4. Linux 内存过量使用

Linux上默认的虚拟内存行为对LightDB不是最优的。由于内核实现内存过量使用的方法，如果LightDB或其它进程的内存要求导致系统用光虚拟内存，那么内核可能会终止LightDB的`postmaster`进程（主管服务器进程）。

如果发生了这样的事情，你会看到像下面这样的内核消息（参考你的系统文档和配置，看看在哪里能看到这样的消息）：

```
Out of Memory: Killed process 12345 (lightdb).
```

这表明`lightdb`进程因为内存压力而被终止了。尽管现有的数据库连接将继续正常运转，但是新的连接将无法被接受。要想恢复，LightDB应该被重启。

一种避免这个问题的方法是在一台你确信其它进程不会耗尽内存的机器上运行LightDB。如果内存资源紧张，增加操作系统的交换空间可以帮助避免这个问题，因为内存不足（OOM）杀手（即终止进程这种行为）只有当物理内存和交换空间都被用尽时才会被调用。

如果LightDB本身是导致系统内存耗尽的原因，你可以通过改变你的配置来避免该问题。在某些情况中，降低内存相关的配置参数可能有所帮助，特别是`shared_buffers`、`work_mem`和`hash_mem_multiplier`。在其他情况中，允许太多连接到数据库服务器本身也可能导致该问题。在很多情况下，最好减小`max_connections`并且转而利用外部连接池软件。

可以修改内核的行为，这样它将不会“过量使用”内存。尽管此设置不会阻止OOM杀手<sup>1</sup>被调用，但它可以显著地降低其可能性并且将因此得到更鲁棒的系统行为。这可以通过用`sysctl`选择严格的过量使用模式来实现：

<sup>1</sup> <https://lwn.net/Articles/104179/>



```
sysctl -w vm.overcommit_memory=2
```

或者在/etc/sysctl.conf中放置一个等效的项。你可能还希望修改相关的设置vm.overcommit\_ratio。详细信息请参阅内核文档的<https://www.kernel.org/doc/Documentation/vm/overcommit-accounting>文件。

另一种方法，可以在改变或不改变vm.overcommit\_memory的情况下使用。它将 postmaster 进程的进程相关的OOM score adjustment值设置为-1000，从而保证它不会成为 OOM 杀手的目标。这样做最简单的方法是在 postmaster 的启动脚本中执行

```
echo -1000 > /proc/self/oom_score_adj
```

并且要在调用 postmaster 之前执行。请注意这个动作必须以 root 完成，否则它将不会产生效果。所以一个被 root 拥有的启动脚本是放置这个动作最容易的地方。如果这样做，你还应该在调用 postmaster 之前在启动脚本中设置这些环境变量：

```
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
export PG_OOM_ADJUST_VALUE=0
```

这些设置将导致 postmaster 子进程使用普通的值为零的 OOM score adjustment 运行，所以 OOM 杀手仍能在需要时把它们作为目标。如果你想要子进程用某些其他 OOM score adjustment 值运行，可以为PG\_OOM\_ADJUST\_VALUE使用其他的值（PG\_OOM\_ADJUST\_VALUE也能被省略，那时它会被默认为零）。如果你没有设置PG\_OOM\_ADJUST\_FILE，子进程将使用和 postmaster 相同的 OOM score adjustment 运行，这是不明智的，因为重点是确保 postmaster 具有优先的设置。

## 16.4.5. Linux 大页面

当LightDB使用大量连续的内存块时，使用大页面会减少开销，特别是在使用大shared\_buffers时。要在LightDB中使用此特性，您需要一个包含CONFIG\_HUGETLBFS=y和CONFIG\_HUGETLB\_PAGE=y的内核。您还必须配置操作系统以提供充足的期望大小的大页面。要估计所需的巨大页面的数量，请启动LightDB，而不启用巨大页面，并使用/proc和/sys文件系统来检查postmaster的匿名共享内存段大小以及系统的巨大页面大小。这可能看起来像：

```
$ head -1 $LTDATA/lightdb.pid
4170
$ pmap 4170 | awk '/rw-s/ && /zero/ {print $2}'
6490428K
$ grep ^Hugepagesize /proc/meminfo
Hugepagesize:      2048 kB
```

6490428 / 2048 大约得出3169.154，所以在这里例子里我们需要至少 3170 大页面。如果机器上的其他程序也需要大页面，更大的设置也许更适合。我们可以设置这个：

```
$ sysctl -w vm.nr_hugepages=3170
```

不要忘记将此设置添加到 /etc/sysctl.conf，以便重启后它能再次运用。

有时候内核在分片期间会无法立即分配想要数量的大页面，所以可能有必要重复该命令或者重新启动。（在重新启动之后，应立即将大部分机器的内存转换为大页面。）要验证巨大的页面对给出大小的分配情况，请使用：

```
$ grep Huge /proc/meminfo
```

可能还需要赋予数据库服务器的操作系统用户权限，让他能通过`sysctl`设置`vm.hugetlb_shm_group`以使用大页面，和/或赋予使用`ulimit -l`锁定内存的权限。

LightDB中大页面的默认行为是尽可能使用它们，用系统的默认大页面大小，并且在失败时转回到正常页面。要强制使用大页面，你可以在`lightdb.conf`中把`huge_pages`设置成`on`。注意此设置下如果没有足够的大页面可用，LightDB将会启动失败。

Linux大页面特性的详细描述可见<https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt>。

## 16.5. 关闭服务器

有几种关闭数据库服务器的方法。在后台，它们都简化为向主管 `lightdb` 进程发送信号。

如果您使用的是 LightDB 的预打包版本，并且您使用其规定来启动服务器，那么您还应该使用其规定来停止服务器。有关详细信息，请参阅包级文档。

直接管理服务器时，可以通过向 `lightdb` 进程发送不同的信号来控制关闭的类型：

### SIGTERM

这是智能关闭模式。在接收SIGTERM后，服务器将不允许新连接，但是会让现有的会话正常结束它们的工作。仅当所有的会话终止后它才关闭。如果服务器处在线备份模式，它将等待直到在线备份模式不再被激活。当在线备份模式被激活时，仍然允许新的连接，但是只能是超级用户的连接（这一例外允许超级用户连接来终止在线备份模式）。如果服务器在恢复时请求智能关闭，恢复和流复制只有在所有正常会话都终止后才停止。

### SIGINT

这是快速关闭模式。服务器不再允许新的连接，并向所有现有服务器进程发送SIGTERM，让它们中断当前事务并立刻退出。然后服务器等待所有服务器进程退出并最终关闭。如果服务处于在线备份模式，备份模式将被终止并致使备份无用。

### SIGQUIT

这是立即关闭模式。服务器将给所有子进程发送SIGQUIT并且等待它们终止。如果有任何进程没有在 5 秒内终止，它们将被发送SIGKILL。主管服务器进程将在所有子进程退出之后立刻退出，而无需做普通的数据库关闭处理。这将导致在下一次启动时（通过重放 WAL 日志）恢复。只在紧急时才推荐这种方式。

`lt_ctl`程序提供了一个发送这些信号关闭服务器的方便的接口。另外，你在非 Windows 系统上可以用`kill`直接发送这些信号。可以用`ps`程序或者从数据目录的`postmaster.pid`文件中找到`lightdb`进程的PID。例如，要做一次快速关闭：

```
$ kill -INT `head -1 /usr/local/lightdb-x/data/lightdb.pid`
```

### Important

最好不要使用SIGKILL关闭服务器。这样做将会阻止服务器释放共享内存和信号量。此外，使用SIGKILL杀掉`lightdb`进程时，`lightdb`不会有将信号传播到它的子进程，所以可能也必须手工杀掉单个的子进程。

要终止单个会话同时允许其他会话继续，使用`pg_terminate_backend()`（参阅Table 9.85 或发送SIGTERM信号到该会话相关的子进程。

## 16.6. 升级一个LightDB集簇

本节讨论如何把你的数据库数据从一个LightDB发行升级到一个更新的发行。

当前LightDB版本号由主要版本号和次要版本号组成。例如，在版本号10.1中，10是主要版本号，1是次要版本号，这意味着这将是主版本10的第一个次要版本。对于LightDB版本10.0之前的版本，版本号由三个数字组成，例如9.5.3。在这些情况下，主要版本由版本号的前两个数字组（例如9.5）组成，次要版本是第三个数字，例如3，这意味着这将是主要版本9.5的第三次要版本。

次要发行从来不改变内部存储格式并且总是向前并向后兼容同一主版本号中的次要发行。例如版本10.1与版本10.0和版本10.6兼容。类似的，例如9.5.3与9.5.0、9.5.1和9.5.6兼容。要在兼容的版本间升级，你只需要简单地在服务器关闭时替换可执行文件并重启服务器。数据目录则保持不变 — 次要升级就这么简单。

对于LightDB的主发行，内部数据存储格式常被改变，这使升级复杂化。传统的把数据移动到新主版本的方法是先转储然后重新载入到数据库，不过这可能会很慢。一种更快的方式是`lt_upgrade`。如下文所讨论的，复制方法也能被用于升级。（如果您使用的是LightDB的预打包版本，它可能会提供脚本来协助主要版本升级。有关详细信息，请参阅包级文档。）

新的主版本也通常会引入一些用户可见的不兼容性，因此可能需要应用程序编程上的改变。所有用户可见的更改都被列在发行注记（Appendix Q中，请特别注意标有“Migration”的小节。尽管你可能从一个主版本升级到另一个，而不用升级中间版本，你应该阅读全部中间版本的主要发行说明。

小心的用户在完全切换过去之前将希望在新版本上测试他们的客户端应用。因此，建立一个新旧版本的并存安装通常是一个好主意。在测试一个LightDB主要升级时，考虑下列可能的改变类别：

### 管理

用于管理员监控和控制服务器的功能在每一个主发行中经常会改变和增加。

### SQL

通常这包括新的SQL命令功能并且在行为上没有更改，除非在发行注记中有特别提到。

### 库 API

通常`libpq`等库值增加新功能，除非在发行注记中有特别提到。

### 系统目录

系统目录改变通常只影响数据库管理工具。

### 服务器 C-语言 API

这涉及到后端函数API中的改变，它使用C编程语言编写。这些改变影响引用服务器内部后端函数的代码。

### 16.6.1. 通过`lt_dumpall`升级数据

一种升级方法是从LightDB的一个主版本转储数据并将它重新载入到另一个主版本中 — 要这样做，你必须使用`lt_dumpall`这样的逻辑备份工具，文件系统级别的备份方法将不会有用（这也阻止你在一个不兼容版本的LightDB中使用一个数据目录，因此在一个数据目录上尝试启动一个错误的服务器版本不会造成很大的危害）。

我们推荐你从较新版本的LightDB中使用`lt_dump`和`lt_dumpall`程序，这样可以利用在这些程序中可能存在的改进。当前发行的转储程序可以读取任何7.0以上版本服务器中的数据。

这些指令假定你现有的安装位于/usr/local/pgsql目录，并且数据区域在/usr/local/lightdb-x/data。请用你的路径进行适当的替换。

1. 如果在创建一个备份，确认你的数据库没有在被更新。这不会影响备份的完整性，但是那些更改当然不会被包括在备份中。如果必要，编辑/usr/local/lightdb-x/data/lt\_hba.conf文件中的权限（或等效的方法）来不允许除你之外的任何人使用数据库。关于访问控制的额外信息请见Chapter 18

要备份你的数据库安装，键入：

```
lt_dumpall > outputfile
```

要制作备份，你可以使用你正在运行版本的lt\_dumpall命令，详见Section 23.1.2 但是，要得到最好的结果，试试使用LightDB 13.8-23.4 的lt\_dumpall命令，因为这个版本包含了对旧版本的缺陷修复和改进。虽然这个建议可能看起来很奇怪，因为你还没有安装新版本，但如果你计划平行地安装新版本，遵循这个建议是很明智的。在这种情况下，你可以正常完成安装并且稍后再来传输数据。这也将减少停机时间。

2. 关闭旧服务器：

```
lt_ctl stop
```

在那些自动启动LightDB的系统上，可能有一个启动文件将完成同样的事情。例如，在一个Red Hat Linux系统中，我们会发现这也能用：

```
/etc/rc.d/init.d/lightdb stop
```

关于启动和停止服务器的细节请见Chapter 16

3. 如果从备份恢复，重命名或删除旧的安装目录（如果它不是针对特定版本的）。重命名该目录是一个好主意，而不是删除它，因为如果你碰到问题并需要返回到它，它还存在。记住该目录可能消耗可观的磁盘空间。要重命名该目录，使用类似的命令：

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

（注意将该目录作为一个单一单元移动，这样相对路径可以保持不变）。

4. 安装新版本的LightDB
5. 如果需要，创建一个新的数据库集簇。记住你必须在登录到一个特殊的数据库用户账户（如果你在升级，你就已经有了这个账户）时执行这些命令。

```
/usr/local/lightdb-x/bin/initdb -D /usr/local/lightdb-x/data
```

6. 恢复你之前的lt\_hba.conf以及任何lightdb.conf修改。
7. 启动数据库服务器，也要使用特殊的数据库用户账户：

```
/usr/local/lightdb-x/bin/lightdb -D /usr/local/lightdb-x/data
```

8. 最后，参考以下从备份恢复你的数据：

```
/usr/local/lightdb-x/bin/ltsql -d postgres -f outputfile
```

使用 new ltsql.

通过在一个不同的目录中安装新的服务器并且并行地在不同的端口运行新旧两个服务器可以达到最低的停机时间。那么你可以这样用：

```
lt_dumpall -p 5432 | ltsql -d postgres -p 5433
```

来转移你的数据。

## 16.6.2. 通过lt\_upgrade升级数据

lt\_upgrade模块允许一个安装从一个 LightDB主版本“就地”升级成另一个主版本。升级可以在数分钟内被执行，特别是使用--link模式时。它 要求和上面的lt\_dumpall相似的步骤，例如启动/停止 服务器、运行initdb。lt\_upgrade 文档概述了所需的步骤。在升级前强烈建议使用lt\_upgrade\_check做升级前检查。

## 16.6.3. 通过复制升级数据

也可以用LightDB的已更新版本逻辑复制来创建一个后备服务器，逻辑复制支持在不同主版本的LightDB之间的复制。后备服务器可以在同一台计算机或者不同的计算机上。一旦它和主服务器（运行旧版本的LightDB）同步好，你可以切换主机并且将后备服务器作为主机，然后关闭旧的数据库实例。这样一种切换使得一次升级的停机时间只有数秒。

这种升级方法可以用内置的逻辑复制工具和外部的逻辑复制系统如pglogical, Slony, Londiste, 和Bucardo。

## 16.7. 阻止服务器欺骗

服务器在运行时，它不可能让恶意用户取代正常的数据库服务器。然而，当服务器关闭时，一个本地用户可以通过启动它们自己的服务器来欺骗正常的服务器。行骗的服务器可以读取客户端发送的密码和查询语句，但是不会返回任何数据，因为LTDATA这个目录是安全的（它有目录权限）。欺骗是可能的，因为任何用户都可以启动一个数据库服务器；客户端无法识别一个无效的服务器，除非它被专门配置。

一种阻止local连接欺骗的方法是使用一个 Unix 域套接字目录（unix\_socket\_directories），该目录只对一个被信任的本地用户有写权限。这可以防止恶意用户在该目录中创建自己的套接字文件。如果你担心有些应用程序可能仍然引用/tmp下的套接字文件并且因此容易受到欺骗，可在操作系统启动时创建一个符号链接/tmp/.s.PGSQL.5432指向一个被重定位的套接字文件。你也可能需要修改/tmp清除脚本防止删除这个符号链接。

local连接的另一个选项是对客户端使用requirepeer指定所需的连接到该套接字的服务器进程的拥有者。

要在TCP连接上防止欺骗，要么使用 SSL 证书，并且确保客户检查服务器的证书，或者使用GSSAPI加密（或者两者都用，如果它们位于单独的连接上）。

要防止 SSL 欺骗，服务器必须配置为仅接受hostssl连接（Section 18.1），并且有 SSL 密钥和证书文件（Section 16.9）。TCP 客户端连接必须使用sslmode=verify-ca或verify-full进行连接，并且安装有适当的根证书文件Section 32.18.1）。

要防止 GSSAPI 欺骗，必须将服务器配置为仅接受hostgssenc连接（Section 18.1）并对它们使用gss身份验证。TCP 客户端必须使用 gssencmode=require连接。

## 16.8. 加密选项

LightDB提供了几个不同级别的加密，并且在保护数据不会因为数据库服务器偷窃、不道德的管理员、不安全网络等因素而泄漏方面 提供很高的灵活性。加密可能也是保护一些诸如医疗记录或财务交易等敏感数据所要求的。

## 口令加密

数据库用户的口令都是以哈希(取决于password\_encryption配置)的方式存储, 所以管理员不能限定实际的口令赋予用户。如果 SCRAM 或者 MD5 加密算法被用于客户端认证, 那么未加密的口令甚至都不可能出现在服务器上, 因为客户端在通过网络发送口令之前, 就已经加密过。推荐使用SCRAM, 因为它是互联网标准而且相比于LightDB特定的MD5认证协议更安全。

## 指定列加密

ltpcrypto模块允许对特定域进行加密存储。这个功能只对某些敏感数据有用。客户端提供解密的密钥, 然后数据在服务器端解密并发送给客户端。

在数据被解密和在服务器与客户端之间传递时, 解密数据和解密密钥将会在服务器端存在短暂的一段时间。这就给那些能完全访问数据库服务器的人提供了一个短暂的截获密钥和数据的时间, 例如系统管理员。

## 数据分区加密

存储加密可以在文件系统层面或者块层面上执行。Linux 文件系统加密 选项包括 eCryptfs 和 EncFS, 而 FreeBSD 使用 PEFS。快层面或者全 盘加密选项包括 Linux 上的 dm-crypt + LUKS 以及 FreeBSD 上的 GEOM 模块 geli 及 gbde。很多其他操作系统也支持这个功能, 包括 Windows。

这个机制避免了在整个计算机或者驱动器被盗的情况下, 未加密的数据被从驱动器中读取。它无法防止在文件系统被挂 载时的攻击, 因为在挂载之后, 操作系统提供数据的解密视图。不过, 要想挂载该文件系统, 你需要有一些方法把加密密钥传递给操作 系统, 并且有时候这个密钥就存储在挂载该磁盘的主机上的某处。

## 跨网络加密数据

SSL 连接加密所有跨网络发送的数据: 口令、查询以及返回的数据。lt\_hba.conf文件允许管理员指定哪些主机可以使用 非加密连接(host), 以及哪些主机需要使用 SSL 加密的连接(hostssl)。客户端还可以指定它们只通过 SSL 连接到服务器。

GSSAPI 加密的连接加密通过网络发送的所有数据, 包括查询和返回的数据。(不会在网络中发送密码。) lt\_hba.conf 文件允许管理员指定哪些主机可以使用非加密连接(host)和哪些主机需要 GSSAPI-加密连接(hostgssenc。此外, 客户端还可以指定它们仅在GSSAPI-加密的连接上来连接服务器(gssencmode=require)。

Stunnel 或 SSH 也能够被用于加密传输。

## SSL 主机认证

客户端和主机都可以提供 SSL 证书给对方。这在两边都需要一些额外的配置, 但是这种方式提供了比仅使用口令更强的身份验证。它避免一个计算机伪装成服务器, 这个时长只要足够读取客户端发送的口令就行了。它还避免了“中间人”攻击, 在其中有一台计算机处于客户端和服务器之间并伪装成服务器读取和传递两者之间的所有数据。

## 客户端加密

如果服务器所在机器的系统管理员是不可信的, 那么客户端加密数据也是必要的。在这种情况下, 未加密的数据从来不会在数据库服务器上出现。数据在发送给服务器之前加密, 而数据库结果在能使用之前必须在客户端上解密。

# 16.9. 用 SSL 进行安全的 TCP/IP 连接

LightDB 有一个对使用 SSL 连接加密客户端/服务器通讯的本地支持, 它可以增加安全性。这个特性要求在客户端和服务器端都安装 OpenSSL 并且在编译 LightDB 的时候打开这个支持。

## 16.9.1. Basic Setup

当SSL支持被编译在LightDB中时，可以通过将lightdb.conf中的 `ssl` 设置为 `on` 让LightDB服务器带着SSL支持被启动。服务器在同一个 TCP 端口监听普通连接和SSL连接，并且将与任何正在连接的客户端协商是否使用SSL。默认情况下，这是客户端的选项，关于如何设置服务器来要求某些或者所有连接使用SSL请见Section 18.1

要SSL模式中启动服务器，包含服务器证书和私钥的文件必须存在。默认情况下，这些文件应该分别被命名为 `server.crt` 和 `server.key` 并且被放在服务器的数据目录中，但是可以通过配置参数 `ssl_cert_file` 和 `ssl_key_file` 指定其他名称和位置。

在 Unix 系统上，`server.key` 上的权限必须不允许所有人或组的任何访问，通过命令 `chmod 0600 server.key` 可以做到。或者，该文件可以由 `root` 所拥有并且具有组读访问（也就是0640权限）。这种设置适用于由操作系统管理证书和密钥文件的安装。用于运行LightDB服务器的用户应该被作为能够访问那些证书和密钥文件的组成员。

如果数据目录允许组读取访问，则证书文件可能需要位于数据目录之外，以符合上面概述的安全要求。通常，启用组访问权限是为了允许非特权用户备份数据库，在这种情况下，备份软件将无法读取证书文件，并且可能会出错。

如果私钥被一个密码保护着，服务器将提示要求这个密码，并且在它被输入前不会启动。默认情况下，使用密码禁用在重启服务器的情况下更改服务器的SSL配置的功能，并且参见 `ssl_passphrase_command_supports_reload`。此外，密码保护的私钥在Windows上根本无法使用。

`server.crt` 中的第一个证书必须是服务器的证书，因为它必须与服务器的私钥匹配。“intermediate”的证书颁发机构，也可以追加到文件。假设根证书和中间证书是使用 `v3_ca` 扩展名创建的，那么这样做避免了在客户端上存储中间证书的必要。（这将证书的CA的基本约束设置为 `true`。）这使得中间证书更容易到期。

无需将根证书添加到 `server.crt`。相反，客户端必须具有服务器证书链的根证书。

## 16.9.2. OpenSSL配置

LightDB读取系统范围的OpenSSL配置文件。默认情况下，该文件被命名为 `openssl.cnf` 并位于 `openssl version -d` 报告的目录中。通过将环境变量设置 `OPENSSL_CONF` 为所需配置文件的名称，可以覆盖此默认值。

OpenSSL支持各种强度不同的密码和身份验证算法。虽然许多密码可以在OpenSSL的配置文件中被指定，您可以通过修改lightdb.conf配置文件中指定专门针对数据库服务器使用密码的 `ssl_ciphers` 配置。

### Note

使用 `NULL-SHA` 或 `NULL-MD5` 可以得到身份验证但没有加密开销。不过，中间人能够读取和传递客户端和服务端之间的通信。此外，加密开销相比身份认证的开销是最小的。出于这些原因，我们建议不要使用 `NULL` 密码。

## 16.9.3. 使用客户端证书

要求客户端提供受信任的证书，把你信任的根证书颁发机构（CA）的证书放置在数据目录文件中。并且修改lightdb.conf中的参数 `ssl_ca_file` 到新的文件名，还要把认证选项 `clientcert=verify-ca` 或 `clientcert=verify-full` 加入到 `lt_hba.conf` 文件中合适的 `hostssl` 行上。然后将在 SSL 连接启动时从客户端请求该证书（一段对于如何在客户端设置证书的描述请见Section 32.18）。

对于具有 `clientcert=verify-ca` 的 `hostssl` 条目，服务器将验证客户端的证书是否由一个受信任的证书颁发机构签署的。如果指定了 `clientcert=verify-full`，则服务器不仅将验证证书链，还将检查用户名或其映射是否与所提供的证书的 `cn`（通用名称）相匹配。请注意，在使用 `cert` 身份验证方法时，要始终确保证书链验证（参见 Section 18.12）。

如果希望避免将链接到现有根证书的中间证书显示在 `ssl_ca_file` 文件中（假设根证书和中间证书是使用 `v3_ca` 扩展名创建的），则这些证书也可以显示在 `ssl_ca_file` 文件中。如果参数 `ssl_crl_file` 被设置，证书撤销列表（CRL）项也要被检查。

`clientcert` 认证选项适用于所有的认证方法，但仅适用于 `lt_hba.conf` 中用 `hostssl` 指定的行。当 `clientcert` 没有指定，服务器仅在客户端证书存在并且 CA 被配置的时候，通过客户端的 CA 文件来查证客户端证书。

有两种方法可以强制用户在登录时提供证书。

第一种方法是对 `lt_hba.conf` 文件中的 `hostssl` 条目使用 `cert` 身份验证方法，这样证书本身可以用于身份验证，同时提供 `ssl` 连接安全性。详细信息请参阅 Section 18.12（在使用 `cert` 身份验证方法时，不需要显式指定任何 `clientcert` 选项。）在这种情况下，证书中 `cn`（通用名称）将针对用户名或适用的映射进行检查。

第二种方法是将对 `hostssl` 条目的任何身份验证方法和客户端证书的验证相结合，通过将 `clientcert` 身份验证选项设置为 `verify-ca` 或 `verify-full`。前一个选项仅强制证书有效，而后者还确保证书中的 `cn`（通用名称）匹配用户名或适用的映射。

## 16.9.4. SSL 服务器文件用法

Table 16. 总结了与服务器上 SSL 配置有关的文件（显示的文件名是默认的名称。本地配置的名称可能会不同）。

Table 16.2. SSL 服务器文件用法

文件	内容	效果
<code>ssl_cert_file</code> ( <code>\$LTDATA/server.crt</code> )	服务器证书	发送给客户端来说明服务器的身份
<code>ssl_key_file</code> ( <code>\$LTDATA/server.key</code> )	服务器私钥	证明服务器证书是其所有者发送的，并不说明证书所有者是值得信任的
<code>ssl_ca_file</code>	可信的证书颁发机构	检查客户端证书是由一个可信的证书颁发机构签名的
<code>ssl_crl_file</code>	被证书授权机构撤销的证书	客户端证书不能出现在这个列表上

服务器在服务器启动时以及服务器配置重新加载时读取这些文件。在 Windows 系统上，只要为新客户端连接生成新的后端进程，它们也会重新读取。

如果在服务器启动时检测到这些文件中的错误，服务器将拒绝启动。但是，如果在配置重新加载过程中检测到错误，则会忽略这些文件，并继续使用旧的 SSL 配置。在 Windows 系统上，如果在后端启动时检测到这些文件中存在错误，则该后端将无法建立 SSL 连接。在所有这些情况下，错误情况都会在服务器日志中报告。

## 16.9.5. 创建证书

要为服务器创建一个有效期为 365 天的简单自签名证书，可以使用下面的 OpenSSL 命令，将 `dbhost.yourdomain.com` 替换为服务器的主机名：

```
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
```



```
-keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

然后执行：

```
chmod og-rwx server.key
```

如果文件的权限比这个更自由，服务器将拒绝该文件。要了解更多关于如何创建你的服务器私钥和证书的细节，请参考OpenSSL文档。

尽管可以使用自签名证书进行测试，但是在生产中应该使用由证书颁发机构（CA）（通常是企业范围的根CA）签名的证书。

要创建其身份可以被客户端验证的服务器证书，请首先创建一个证书签名请求（CSR）和一个公共/专用密钥文件：

```
openssl req -new -nodes -text -out root.csr \  
-keyout root.key -subj "/CN=root.yourdomain.com"  
chmod og-rwx root.key
```

然后，使用密钥对请求进行签名以创建根证书颁发机构（使用Linux上的默认OpenSSL配置文件位置）：

```
openssl x509 -req -in root.csr -text -days 3650 \  
-extfile /etc/ssl/openssl.cnf -extensions v3_ca \  
-signkey root.key -out root.crt
```

最后，创建由新的根证书颁发机构签名的服务器证书：

```
openssl req -new -nodes -text -out server.csr \  
-keyout server.key -subj "/CN=dbhost.yourdomain.com"  
chmod og-rwx server.key
```

```
openssl x509 -req -in server.csr -text -days 365 \  
-CA root.crt -CAkey root.key -CAcreateserial \  
-out server.crt
```

server.crt和server.key应该存储在服务器上，并且root.crt应该存储在客户端上，以便客户端可以验证服务器的叶证书已由其受信任的根证书签名。root.key应该离线存储以用于创建将来的证书。

也可以创建一个包括中间证书的信任链：

```
# root  
openssl req -new -nodes -text -out root.csr \  
-keyout root.key -subj "/CN=root.yourdomain.com"  
chmod og-rwx root.key  
openssl x509 -req -in root.csr -text -days 3650 \  
-extfile /etc/ssl/openssl.cnf -extensions v3_ca \  
-signkey root.key -out root.crt  
  
# intermediate  
openssl req -new -nodes -text -out intermediate.csr \  
-keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"  
chmod og-rwx intermediate.key  
openssl x509 -req -in intermediate.csr -text -days 1825 \  
-extfile /etc/ssl/openssl.cnf -extensions v3_ca \  
-out intermediate.crt
```

```

-CA root.crt -CAkey root.key -CAcreateserial \
-out intermediate.crt

# leaf
openssl req -new -nodes -text -out server.csr \
-keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
-CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
-out server.crt

```

server.crt和intermediate.crt应连接成一个证书文件包中并存储在服务器上。server.key还应该存储在服务器上。root.crt应将其存储在客户端上，以便客户端可以验证服务器的叶证书是否已由链接到其受信任根证书的证书链签名。root.key和intermediate.key应离线存储以用于创建将来的证书。

## 16.10. Secure TCP/IP Connections with GSSAPI Encryption

LightDB还具有原生支持使用 GSSAPI 加密客户端/服务器通信以提高安全性。支持要求在客户端和服务器系统上都安装 GSSAPI实现（例如 MIT Kerberos），并在编译时启用 LightDB 中的支持。

### 16.10.1. 基本设置

LightDB 服务器将侦听同一 TCP 端口上的普通连接和 GSSAPI-加密连接，并将与任何连接的客户端协商是否使用 GSSAPI 进行加密（和身份验证）。默认情况下，此决策由客户端决定（这意味着攻击者可以降级）；参见Section 18.1关于设置服务器对某些或所有连接要求使用 GSSAPI。

当使用GSSAPI加密时，一般也使用GSSAPI进行身份验证，因为底层机制将确定客户端和服务器的身份(根据GSSAPI实现)，在任何情况下。但这不是必需的；可以选择另一种LightDB认证方法执行额外的验证。

除了协商行为的配置外，GSSAPI加密不需要GSSAPI认证外的其他设置。（关于配置的更多信息，参见 Section 18.6）

## 16.11. 使用SSH隧道的安全 TCP/IP 连接

可以使用SSH来加密客户端和LightDB服务器之间的网络连接。如果处理得当，这将提供一个足够安全的网络连接，即使是对那些无 SSL 能力的客户端。

首先确认在LightDB服务器的同一台机器上正确运行着一个SSH服务器，并且你可以使用ssh作为某个用户登入；然后，您可以建立到远程服务器的安全隧道。安全隧道侦听本地端口并将所有流量转发到远程机器上的端口。发送到远程端口的流量可以到达其 localhost 地址，或者如果需要，可以到达不同的绑定地址；它似乎不是来自您的本地机器。此命令创建从客户端机器到远程机器 foo.com 的安全隧道：

```
ssh -L 63333:localhost:5432 joe@foo.com
```

-L 参数中的第一个数字 63333 是隧道的本地端口号；它可以是任何未使用的端口。（IANA 保留端口 49152 到 65535 供私人使用。）此后的名称或 IP 地址是您要连接到的远程绑定地址，即 localhost，这是默认值。第二个数字 5432 是隧道的远端，例如，您的数据库服务器正在使用的端口号。为了使用此隧道连接到数据库服务器，您需要连接到本地机器上的端口 63333：

```
ltsql -h localhost -p 63333 postgres
```

对于数据库服务器，它将把你看做是连接到localhost绑定地址的主机foo.com上的用户joe，并且它将使用为该用户到该绑定地址的连接配置的任何身份验证程序。注意服务器将不会认为连接是SSL加密的，因为事实上SSH服务器和LightDB服务器之间没有加密。这不应该造成任何额外的安全风险，因为它们在同一台机器上。

为了让隧道设置成功，你必须允许通过ssh作为joe@foo.com连接，就像你已经尝试使用ssh来创建一个终端会话。

你应当也已经设定好了端口转发：

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

但是数据库服务器则将会看到连接从它的foo.com绑定地址进来，它没有被默认设置listen\_addresses = 'localhost'所打开。这通常不是你想要的。

如果你必须通过某个登录主机“跳”到数据库服务器，一个可能的设置看起来像：

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

注意这种从shell.foo.com到db.foo.com的连接的方法将不会被SSH隧道加密。当网络被限制于各种方法时，SSH提供了相当多的配置可能性。详情请参考SSH的文档。

### Tip

一些其他的应用可以提供安全隧道，它们使用和刚刚描述的SSH概念上相似的过程。

## 16.12. 在Windows上注册事件日志

要为操作系统注册一个Windows事件日志库，发出这个命令：

```
regsvr32 postgresql_library_directory/pgevent.dll
```

这会创建被事件查看器使用的注册表项，默认事件源命名为LightDB。

要指定一个不同的事件源名称（见event\_source）。使用/n和/i选项：

```
regsvr32 /n /i:event_source_name postgresql_library_directory/pgevent.dll
```

要从操作系统反注册事件日志库，发出这个命令：

```
regsvr32 /u [/i:event_source_name] postgresql_library_directory/pgevent.dll
```

### Note

要启用数据库服务器中的事件日志，在lightdb.conf中修改log\_destination来包括eventlog。

---

# Chapter 17. 服务器配置

有很多配置参数可以影响数据库系统的行为。本章的第一节中我们将描述一下如何与配置参数交互。后续的小节将详细地讨论每一个参数。

## 17.1. 设置参数

### 17.1.1. 参数名称和值

所有参数名都是大小写不敏感的。每个参数都可以接受五种类型之一的值：布尔、字符串、整数、浮点数或枚举。该类型决定了设置该参数的语法：

- 布尔：值可以被写成 `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0`（都是大小写不敏感的）或者这些值的任何无歧义前缀。
- 字符串：通常值被包括在单引号内，值内部的任何单引号都需要被双写。不过，如果值是一个简单数字或者标识符，引号通常可以被省略。（与 SQL 关键字匹配的值需要在某些上下文中引用。）
- 数字（整数和浮点）：数字参数可以规定为惯用的整数和浮点格式；如果参数为整数类型，则小数值四舍五入到最接近的整数。证书参数还接受十六进制输入（以 `0x` 开头）和十进制输入（以 `0` 开头），但是这些格式不能有小数。不能使用千位分隔符。引号是不是必需的，除了十六进制输入。
- 带单位的数字：一些数字参数具有隐含单位，因为它们描述内存或时间量。单位可能是字节、千字节、块（通常是 8KB）、毫秒、秒或分钟。这些设置之一的一个未修饰的数字值将使用该设置的默认单位，默认单位可以通过引用 `pg_settings.unit` 来找到。为了方便，也可以显式地指定一个不同的单位，例如时间值可以是 `'120 ms'`，并且它们将被转换到参数的实际单位。要使用这个特性，注意值必须被写成一个字符串（带有引号）。单位名称是大小写敏感的，并且在数字值和单位之间可以有空白。
  - 可用的内存单位是 B（字节）、kB（千字节）、MB（兆字节）和 GB（吉字节）。内存单位的乘数是 1024 而不是 1000。
  - 可用的时间单位是 `us`（微秒）、`ms`（毫秒）、`s`（秒）、`min`（分钟）、`h`（小时）和 `d`（天）。  
如果一个单位指定了小数值，如果有下一个较小的单元，它将四舍五入为下一个较小单位的倍数。例如，30.1 GB 将被转换为 30822 MB 而不是 32319628902 B。如果参数为整数类型，则在进行任何单位转换之后，最后四舍五入到整数。
- 枚举：枚举类型的参数以与字符串参数相同的方式指定，但被限制到一组有限的值。这样一个参数可用的值可以在 `pg_settings.enumvals` 中找到。枚举参数值是大小写无关的。

### 17.1.2. 通过配置文件影响参数

设置这些参数最基本的方法是编辑 `lightdb.conf` 文件，它通常被保存在数据目录中（当数据库集簇目录被初始化时，一个默认的拷贝将会被安装在那里）。一个该文件的例子看起来是：

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '$user', public'
shared_buffers = 128MB
```

每一行指定一个参数。名称和值之间的等号是可选的。空白是无意义的（除了在一个引号引用的参数值内）并且空行被忽略。井号（#）指示该行的剩余部分是一个注释。非简单标识

符或者数字的参数值必须用单引号包围。要在参数值里嵌入单引号，要么写两个单引号（首选）或者在引号前放反斜线。如果文件包含相同参数的多个条目，则忽略除最后一个之外的所有条目。

以这种方式设定的参数为集簇提供了默认值。除非这些设置被覆盖，活动会话看到的就是这些设置。下面的小节描述了管理员或用户覆盖这些默认值的方法。

主服务器进程每次收到SIGHUP信号（最简单的方法是从命令行运行`lt_ctl reload`或调用SQL函数`pg_reload_conf()`来发送这个信号）后都会重新读取这个配置文件。主服务器进程还会把这个信号传播给所有正在运行的服务器进程，这样现有的会话也能采用新值（要等待它们完成当前正在执行的客户端命令之后才会发生）。另外，你可以直接向一个单一服务器进程发送该信号。有些参数只能在服务器启动时设置，在配置文件中对这些条目的修改将被忽略，直到下次服务器重启。配置文件中的非法参数设置也会在SIGHUP处理过程中被忽略（但是会记录日志）。

除`lightdb.conf`之外，LightDB数据目录还包含一个文件`lightdb.auto.conf`，它具有和`lightdb.conf`相同的格式但是原自动编辑，而不是手工编辑。这个文件保存了通过ALTER SYSTEM命令提供的设置。每当`lightdb.conf`被读取时这个文件会被自动读取，并且它的设置会以同样的方式生效。`lightdb.auto.conf`中的设置会覆盖`lightdb.conf`中的设置。

外部工具也可以修改`lightdb.auto.conf`。不建议在服务器运行时执行此操作，因为并发的ALTER SYSTEM可能会覆盖这些更改。这些工具可能只是简单地在末尾附加新的设置，或者它们可能删除重复的设置和/或注释（就像ALTER SYSTEM）。

系统视图`pg_file_settings`可以有助于对配置文件中的更改进行提前测试，或者在SIGHUP信号没有达到预期效果时用来诊断问题。

### 17.1.3. 通过SQL影响参数

LightDB提供了三个SQL命令来建立配置默认值。已经提到过的ALTER SYSTEM命令提供了一种改变全局默认值的从SQL可访问的方法；它在功效上等效于编辑`lightdb.conf`。此外，还有两个命令可以针对每个数据库或者每个角色设置默认值：

- ALTER DATABASE命令允许针对一个数据库覆盖其全局设置。
- ALTER ROLE命令允许用用户指定的值来覆盖全局设置和数据库设置。

只有当开始一个新的数据库会话时，用ALTER DATABASE和ALTER ROLE设置的值才会被应用。它们会覆盖从配置文件或服务器命令行获得的值，并且作为该会话后续的默认值。注意某些设置在服务器启动后不能被更改，并且因此不能被这些命令（或者下文列举的命令）设置。

以上3种修改参数的方式，都会在日志文件（位于`$LTDATA/log`中）中详细记录参数的修改情况：包括参数修改前的值，参数修改后的值，以及修改参数的时间、客户端程序名、修改参数的用户、修改参数使用的具体SQL语句。同时当执行ALTER DATABASE、ALTER DATABASE或ALTER ROLE时还会在客户端程序中提示参数修改前和修改后的值。

一旦一个客户端连接到数据库，LightDB会提供两个额外的SQL命令（以及等效的函数）用以影响会话本地的配置设置：

- SHOW命令允许察看任何参数的当前值。对应的SQL函数是`current_setting(setting_name text)`（参见Section 9.27.1）。
- SET命令允许修改对于一个会话可以本地设置的参数的当前值，它对其他会话没有影响。对应的SQL函数是`set_config(setting_name, new_value, is_local)`（参见Section 9.27.1）。

此外，系统视图`pg_settings`可以被用来查看和改变会话本地的值：

- 查询这个视图与使用SHOW ALL相似，但是可以提供更多细节。它也更加灵活，因为可以为它指定过滤条件或者把它与其他关系进行连接。

- 在这个视图上使用UPDATE并且指定更新setting 列，其效果等同于发出SET命令。例如，下面的命令

```
SET configuration_parameter TO DEFAULT;
```

等效于：

```
UPDATE pg_settings SET setting = reset_val WHERE name =
'configuration_parameter';
```

### 17.1.4. 通过 Shell 影响参数

除了在数据库或者角色层面上设置全局默认值或者进行覆盖，你还可以通过 shell 工具把设置 传递给LightDB。服务器和libpq 客户端库都能通过 shell 接受参数值。

- 在服务器启动期间，可以通过-c命令行参数把参数设置传递给 lightdb命令。例如：

```
lightdb -c log_connections=yes -c log_destination='syslog'
```

这种方式提供的设置会覆盖通过lightdb.conf或者 ALTER SYSTEM提供的设置，因此除了重启服务器之外无法从全局上改变它们。

- 当通过libpq启动一个客户端会话时，可以使用PGOPTIONS 环境变量指定参数设置。这种方式建立的设置构成了会话生存期间的默认值，但是不会影响 其他的会话。由于历史原因，LTOPTIONS的格式和启动 lightdb命令时用到的相似，特别是-c标志必须被指定。 例如：

```
env PGOPTIONS="-c geqo=off -c statement_timeout=5min" ltsql
```

通过 shell 或者其他方式，其他客户端和库可能提供它们自己的机制，以便允许用户在不直接 使用SQL命令的前提下修改会话设置。

### 17.1.5. 管理配置文件内容

LightDB提供了一些特性用于把复杂的 lightdb.conf文件分解成子文件。在管理多个具有相关但不完全相同 配置的服务器时，这些特性特别有用。

除了单个参数设置，lightdb.conf文件可以包含include指令，它指定要读入和处理的另一个文件，就好像该文件被插入到配置文件的这个点。这个特性允许一个配置文件被划分成物理上独立的部分。include指令看起来像：

```
include 'filename'
```

如果文件名不是一个绝对路径，它将作为包含引用配置文件的目录的相对位置。include可以被嵌套。

也有一个include\_if\_exists指令，它的作用和include指令一样，不过当被引用的文件不存在或者无法被读取时其行为不同。一个通常的include将认为这是一个错误情况，而include\_if\_exists仅仅记录一个消息并且继续处理引用配置文件。

lightdb.conf文件也可以包含include\_dir指令，它指定要被包含的配置文件的整个目录。它的用法类似：

```
include_dir 'directory'
```

非绝对目录名被当做包含引用配置文件的目录的相对路径。在该指定目录中，只有以后缀名 .conf 结尾的非目录文件才会被包括。以 . 字符开头的文件名也会被忽略，因为在某些平台上它们是隐藏文件。一个包括目录中的多个文件 被以文件名顺序处理（根据 C 区域规则排序，即数字在字母之前并且大写字母在小写字母 之前）。

包括文件或目录可以被用来在逻辑上分隔数据库配置的各个部分，而不是用一个很大的 lightdb.conf 文件。考虑一个有两台数据库服务器的公司，每一个都有不同的内存量。很可能配置的元素都会被共享，例如用于日志的参数。但是两者关于内存的参数将会不同。并且还可能会有服务器相关的自定义。一种管理这类情况的方法是将你的站点的自定义配置修改分成三个文件。你可以把下面的内容加入到你的 lightdb.conf 文件末尾来包括它们：

```
include 'shared.conf'
include 'memory.conf'
include 'server.conf'
```

所有的系统将会有相同的 shared.conf。每个有特定内存量的服务器可以共享相同的 memory.conf。你可能对所有 8GB 内存的服务器有一个，而对那些 16GB 内存的服务器有另一个。并且最后 server.conf 可以装有真正服务器相关的配置信息。

另一中可能性是创建一个配置文件目录并把这个信息放到其中的文件里。例如，一个 conf.d 目录可以在 lightdb.conf 的末尾被引用：

```
include_dir 'conf.d'
```

然后你可以这样命名 conf.d 目录中的文件：

```
00shared.conf
01memory.conf
02server.conf
```

这种命名习惯建立了这些文件将被载入的清晰顺序。这是很重要的，因为在服务器读取配置文件时，对于一个特定的参数只有最后碰到的一个设置才会被使用。在这个例子中，conf.d/02server.conf 设置的东西将会覆盖在 conf.d/01memory.conf 中相同参数的值。

你还可以使用这种配置目录方法，在命名文件时更有描述性：

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

这种形式的安排为每个配置文件变体给定了一个唯一的名称。当多个服务器把它们的配置全部存储在一个位置（例如在一个版本控制仓库中）时，这可以帮助消除歧义（在版本控制下存储数据库配置文件是另一个值得考虑的好方法）。

## 17.2. 文件位置

除了已经提到过的 lightdb.conf 文件之外，LightDB 还使用另外两个手工编辑的配置文件，它们控制客户端认证（其使用在 Chapter 18 中讨论）。默认情况下，所有三个配置文件都存放在数据库集簇的数据目录中。本节描述的参数允许配置文件放在别的地方（这么做可以简化管理，特别是如果配置文件被独立放置，可以很容易保证它得到恰当的备份）。

data\_directory (string)

指定用于数据存储的目录。这个选项只能在服务器启动时设置。

`config_file` (string)

指定主服务器配置文件（通常叫`lightdb.conf`）。这个参数只能在`lightdb`命令行上设置。

`hba_file` (string)

指定基于主机认证配置文件（通常叫`lt_hba.conf`）。这个参数只能在服务器启动的时候设置。

`ident_file` (string)

指定用于用户名称映射的配置文件（通常叫`lt_ident.conf`）。这个参数只能在服务器启动的时候设置。另见Section 18.2

`external_pid_file` (string)

指定可被服务器创建的用于管理程序的额外进程 ID (PID) 文件。这个参数只能在服务器启动的时候设置。

在默认安装中不会显式设置以上参数。相反，命令行参数`-D`或者环境变量`LTDATA`指定数据目录，并且上述配置文件都能在数据目录中找到。

如果你想把配置文件放在别的地方而不是数据目录中，那么`lightdb -D`命令行选项或者环境变量`LTDATA`必须指向包含配置文件的目录，并且`lightdb.conf`中（或者命令行上）的`data_directory`参数必须显示数据目录实际存放的地方。请注意，`data_directory`将覆盖`-D`和`LTDATA`指定的数据目录位置，但是不覆盖配置文件的位置。

如果你愿意，可以使用选项`config_file`、`hba_file`和/或`ident_file`单独指定配置文件名称和位置。`config_file`只能在`lightdb`命令行上指定，但是其他文件可以在主配置文件中设置。如果所有三个参数外加`data_directory`被显式地设置，则不必指定`-D`或`LTDATA`。

在设置任何这些参数时，相对路径将被解释为相对于`lightdb`启动路径的路径。

## 17.3. 连接和认证

### 17.3.1. 连接设置

`listen_addresses` (string)

指定服务器在哪些 TCP/IP 地址上监听客户端连接。值的形式是一个逗号分隔的主机名和/或数字 IP 地址列表。特殊项\*对应所有可用 IP 接口。项`0.0.0.0`允许监听所有 IPv4 地址并且`::`允许监听所有 IPv6 地址。如果列表为空，服务器将根本不会监听任何 IP 接口，在这种情况下只能使用 Unix 域套接字来连接它。默认值是`localhost`，它只允许建立本地 TCP/IP “环回”连接。虽然客户端认证（Chapter 18 允许细粒度地控制谁能访问服务器，`listen_addresses`控制哪些接口接受连接尝试，这能帮助在不安全网络接口上阻止重复的恶意连接请求。这个参数只能在服务器启动时设置。

`port` (integer)

服务器监听的 TCP 端口；默认是 5432。请注意服务器会同一个端口号监听所有的 IP 地址。这个参数只能在服务器启动时设置。

`max_connections` (integer)

决定数据库的最大并发连接数。默认值通常是 100 个连接，但是如果内核设置不支持（`initdb`时决定），可能会比这个数少。这个参数只能在服务器启动时设置。

当运行一个后备服务器时，你必须设置这个参数等于或大于主服务器上的参数。否则，后备服务器上可能无法允许查询。



`superuser_reserved_connections` (integer)

决定为LightDB超级用户连接而保留的连接“槽”数。同时活跃的并发连接最多`max_connections`个。任何时候，活跃的并发连接数最多为`max_connections`减去`superuser_reserved_connections`，新连接就只能由超级用户发起了，并且不会有新的复制连接被接受。

默认值是 3 连接。这个值必须小于`max_connections`。这个参数只能在服务器启动时设置。

`unix_socket_directories` (string)

指定服务器用于监听来自客户端应用的连接的 Unix 域套接字目录。通过列出用逗号分隔的多个目录可以建立多个套接字。项之间的空白被忽略，如果你需要在名字中包括空白或逗号，在目录名周围放上双引号。一个空值指定在任何 Unix 域套接字上都不监听，在这种情况下只能使用 TCP/IP 套接字来连接到服务器。

以@开头的值指定将要在抽象命名空间中创建Unix-domain套接字(当前在Linux和Windows上支持)。在这种情况下，该值不指定“directory”，而是指定一个前缀，实际的套接字名称以与文件系统名称空间相同的方式从该前缀计算。虽然抽象套接字名称前缀可以自由选择，由于它不是一个文件系统位置，但惯例上仍然使用类似文件系统的值，例如@/tmp。

默认值通常是/tmp，但是在编译时可以被改变。在windows上，默认值为空，意味着默认不建立UNIX-域嵌套。这个参数只能在服务器启动时设置。

除了套接字文件本身(名为.s.PGSQL.nnnn，其中nnnn是服务器的端口号)，一个名为.s.PGSQL.nnnn.lock的普通文件会在每一个`unix_socket_directories`目录中被创建。任何一个都不应该被手工移除。对于抽象命名空间，没有锁文件被建立。

`unix_socket_group` (string)

设置 Unix 域套接字的所属组(套接字的所属用户总是启动服务器的用户)。可以与选项`unix_socket_permissions`一起用于对 Unix域连接进行访问控制。默认是一个空字符串，表示服务器用户的默认组。这个参数只能在服务器启动时设置。

Windows 上不支持这个参数。所有设置会被忽略。同样，抽象命名空间中的套接字没有文件属主，所以在这种情况下，这个设置也会被忽略。

`unix_socket_permissions` (integer)

设置 Unix 域套接字的访问权限。Unix 域套接字使用普通的 Unix 文件系统权限集。这个参数值应该是数字的形式，也就是系统调用`chmod`和`umask`接受的形式(如果使用自定义的八进制格式，数字必须以一个0(零)开头)。

默认的权限是0777，意思是任何人都可以连接。合理的候选是0770(只有用户和同组的人可以访问，又见`unix_socket_group`)和0700(只有用户自己可以访问)(请注意，对于 Unix 域套接字，只有写权限有麻烦，因此没有对读取和执行权限的设置和收回)。

这个访问控制机制与Chapter 18中的用户认证没有关系。

这个参数只能在服务器启动时设置。

这个参数与完全忽略套接字权限的系统无关，尤其是自版本10以上的Solaris。在那些系统上，可以通过把`unix_socket_directories`指向一个把搜索权限限制给指定用户的目录来实现相似的效果。

抽象命名空间中的套接字没有文件权限，所以这种情况下这个设置也会被忽略。

`bonjour` (boolean)

通过Bonjour广告服务器的存在。默认值是关闭。这个参数只能在服务器启动时设置。

bonjour\_name (string)

指定Bonjour服务名称。空字符串'' (默认值) 表示使用计算机名。如果编译时没有打开Bonjour支持那么将忽略这个参数。这个参数只能在服务器启动时设置。

tcp\_keepalives\_idle (integer)

规定在操作系统向客户端发送一个TCP keepalive消息后无网络活动的时间总量。如果指定值时没有单位, 则以秒为单位。值0(默认值)表示选择操作系统默认值。指定不活动多少秒之后通过 TCP 向客户端发送一个 keepalive 消息。0 值表示使用默认值。这个参数只有在支持TCP\_KEEPIDLE或等效套接字选项的系统或 Windows 上才可以使用。在其他系统上, 它必须为零。在通过 Unix 域套接字连接的会话中, 这个参数被忽略并且总是读作零。

#### Note

在 Windows 上, 设定值为0将设置这个参数为 2 小时, 因为 Windows 不支持读取系统默认值。

tcp\_keepalives\_interval (integer)

规定未被客户端确认收到的TCP keepalive消息应重新传输的时间长度。如果指定值时没有单位, 则以秒为单位。值0(默认值)表示选择操作系统默认值。这个参数只有在支持TCP\_KEEPINTVL或等效套接字选项的系统或 Windows 上才可以使用。在其他系统上, 必须为零。在通过 Unix域套接字连接的会话中, 这个参数被忽略并总是被读作零。

#### Note

在 Windows 上, 设定值为0将设置这个参数为 1 秒, 因为 Windows 不支持读取系统默认值。

tcp\_keepalives\_count (integer)

指定服务器到客户端的连接被认为中断之前可以丢失的TCP keepalive消息的数量。值0(默认值)表示选择操作系统默认值。这个参数只有在支持TCP\_KEEPCNT或等效套接字选项的系统上才可以使用。在其他系统上, 必须为零。在通过 Unix 域套接字连接的会话中, 这个参数被忽略并总是被读作零。

#### Note

Windows 不支持该参数, 且必须为零。

tcp\_user\_timeout (integer)

指定传输的数据在TCP连接被强制关闭之前可以保持未确认状态的时间量。如果指定值时没有单位, 则以毫秒为单位。值0(默认值)表示选择操作系统默认值。这个参数只有在支持TCP\_USER\_TIMEOUT的系统上才被支持; 在其他系统上, 它必须为零。在通过 Unix-domain 套接字连接的会话中, 此参数将被忽略并且始终读取为零。

#### Note

在Windows上不支持该参数, 并且必须为零。

`client_connection_check_interval` (integer)

在运行查询时，设置检查客户端是否保持连接的可选检查的时间间隔。这个检查通过轮询套接字来执行，并且在内核报告该连接关闭时，允许长时间运行的查询可以尽快中止。

这个选项当前仅在支持非标的POLLRDHUP扩展到poll系统调用的系统上有效，包括Linux。

如果指定的值没有单位，则以毫秒为单位。默认值为0，代表禁用连接检查。没有连接检查，服务器将只在与套接字的下一次交互时检测连接的丢失，当它等待、接收或发送数据时。

为了让内核本身能够在包括网络故障在内的所有场景中，在已知的时间范围内可靠地检测丢失的TCP连接，它可能还需要调整操作系统的TCP保持连接设置，或者LightDB的`tcp_keepalives_idle`、`tcp_keepalives_interval` 和 `tcp_keepalives_count` 设置。

## 17.3.2. 安全和认证

`authentication_timeout` (integer)

允许完成客户端认证的最长时间。如果一个客户端没有在这段时间里完成认证协议，服务器将关闭连接。这样就避免了出问题的客户端无限制地占有一个连接。如果指定值时没有单位，则以秒为单位。默认值是 1分钟 (1m)。这个参数只能在服务器命令行上或者在`lightdb.conf`文件中设置。

`password_encryption` (enum)

当在CREATE ROLE或者ALTER ROLE中指定了口令时，这个参数决定用于加密该口令的算法。可能的值是 `scram-sha-256`，可以用SCRAM-SHA-256，和 `md5`加密密码，以MD5 哈希的方式存储密码。默认为 `scram-sha-256`。

注意老的客户端可能缺少对SCRAM认证机制的支持，因此无法使用用SCRAM-SHA-256加密的口令。详情请参考Section 18.5

`krb_server_keyfile` (string)

设置服务器的Kerberos密钥文件的位置。默认为`FILE:/usr/local/lightdb-x/etc/krb5.keytab`（其中目录部分是在构建时由`sysconfdir`指定的；用`lt_config --sysconfdir`来决定）。如果这个参数被设为空字符串，它将被忽略，并且系统依赖的默认值被应用。这个参数只能在`lightdb.conf`文件中或者服务器命令行上设置。详情请参考Section 18.6

`krb_caseins_users` (boolean)

设置是否应该以大小写不敏感的方式对待GSSAPI用户名。默认值是`off`（大小写敏感）。这个参数只能在`lightdb.conf`文件中或者服务器命令行上设置。

`db_user_namespace` (boolean)

这个参数启用针对每个数据库的用户名。这个参数默认是关掉的。这个参数只能在`lightdb.conf`文件中或者服务器命令行上设置。

如果这个参数为打开，应该把用户创建成`username@dbname`的形式。当一个连接客户端传来`username`时，`@`和数据库名会被追加到用户名并且服务器会查找这个与数据库相关的用户名。注意在SQL环境中用含有`@`的名称创建用户时，需要把用户名放在引号内。

在这个参数被启用时，仍然可以创建平常的全局用户。而在客户端中指定这种用户时只需要简单地追加`@`，例如`joe@`。在服务器查找该用户名之前，`@`会被剥离掉。

`db_user_namespace`会导致客户端和服务器的用户名表达形式不同。认证检查总是会以服务器的用户名表达形式来完成，因此认证方法必须针对服务器用户名而不是客户端用户

名来配置。由于md5方法在客户端和服务器两端都使用用户名作为salt，md5不能与db\_user\_namespace同时使用。

### Note

这种特性的目的是在找到完整的解决方案之前提供一种临时的措施。在找到完整解决方案时，这个选项将被去除。

## 17.3.3. SSL

有关设置SSL的更多信息请参考Section 16.9

ssl (boolean)

启用SSL连接。这个参数只能在lightdb.conf文件中或者服务器命令行上设置。默认值是off。

ssl\_ca\_file (string)

指定包含 SSL 服务器证书颁发机构 (CA) 的文件名。相对路径是相对于数据目录的。这个参数只能在lightdb.conf文件中或者服务器命令行上设置。默认值为空，表示没有载入CA文件，并且客户端证书验证没有被执行。

ssl\_cert\_file (string)

指定包含 SSL 服务器证书的文件名。相对路径是相对于数据目录的。这个参数只能在lightdb.conf文件中或者服务器命令行上设置。默认值是server.crt。

ssl\_crl\_file (string)

指定包含 SSL 服务器证书撤销列表 (CRL) 的文件名。其中的相对路径是相对于数据目录的。这个参数只能在lightdb.conf文件中或者服务器命令行上设置。默认值是空，表示没有载入CRL文件(除非设定了 ssl\_crl\_dir is set)。

ssl\_crl\_dir (string)

指定包含SSL服务器证书吊销列表(CRL)的目录的名称。相对路径与数据目录相关。这个参数只能在lightdb.conf文件中设置或者通过服务器命令行。默认值为空，意味着没有使用CRLs(除非设定了ssl\_crl\_file)。

这个目录需要用OpenSSL 命令 openssl rehash 或 c\_rehash来准备。 请参阅相应文档

当使用此设置时，在连接时会按需加载指定目录下的CRLs。新的CRLs可以添加到该目录中，并可以立即使用。这与ssl\_crl\_file不同，那个会导致文件中的CRL在服务器启动时间或重新加载配置时加载。两个设置可以一起使用。

ssl\_key\_file (string)

指定包含 SSL 服务器私钥的文件名。相对路径是相对于数据目录。这个参数只能在lightdb.conf文件中或者服务器命令行上设置。默认值是server.key。

ssl\_ciphers (string)

指定一个允许用于SSL连接的SSL密码套件列表。这个设置的语法和所支持的值列表可以参见OpenSSL包中的ciphers手册页。仅在使用 TLS 版本 1.2 及更低版本的连接才受影响。目前没有控制 TLS 版本 1.3 连接使用的密码选择的设置。默认值是HIGH:MEDIUM:+3DES:!aNULL。默认值通常是一种合理的选择，除非用户有特定的安全性需求。

这个参数只能在lightdb.conf文件中或者服务器命令行上设置。

默认值的解释：

HIGH

使用来自HIGH组的密码的密码组（例如 AES, Camellia, 3DES）

MEDIUM

使用来自MEDIUM组的密码的密码组（例如 RC4, SEED）

+3DES

OpenSSL 对HIGH的默认排序是有问题的，因为它认为 3DES 比AES128 更高。这是错误的，因为 3DES 提供的安全性比 AES128 低，并且它也更加慢。+3DES把它重新排序在所有其他HIGH和 MEDIUM密码之后。

!aNULL

禁用不做认证的匿名密码组。这类密码组容易收到MITM攻击，因此不应被使用。

可用的密码组细节可能会随着OpenSSL 版本变化。可使用命令 `openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL'` 来查看当前安装的OpenSSL版本的实际细节。注意这个列表是根据服务器密钥类型在运行时过滤过的。

`ssl_prefer_server_ciphers` (boolean)

指定是否使用服务器的 SSL 密码首选项，而不是用客户端的。这个参数只能在lightdb.conf文件中或者服务器命令行上设置。默认值是 on。

老的LightDB版本没有这个设置并且总是使用客户端的首选项。这个设置主要用于与那些版本的向后兼容性。使用服务器的首选项通常会更好，因为服务器更可能会被合适地配置。

`ssl_ecdh_curve` (string)

指定用在ECDH密钥交换中的曲线名称。它需要被所有连接的客户端支持。它不需要与服务椭圆曲线密钥使用的曲线相同。这个参数只能在lightdb.conf文件中或者服务器命令行上设置。默认值是prime256v1。

OpenSSL最常见的曲线名称是：prime256v1 (NIST P-256)、secp384r1 (NIST P-384)、secp521r1 (NIST P-521)。openssl eparam -list\_curves命令可以显示可用曲线的完整列表。不过并不是所有的都在TLS中可用。

`ssl_min_protocol_version` (enum)

设置要使用的最小SSL/TLS协议版本。当前的可用版本包括：TLSv1, TLSv1.1, TLSv1.2, TLSv1.3。旧版本的 OpenSSL 库不支持所有值；如果选择了不支持的设置将会引发错误。TLS 1.0之前的协议版本，也就是SSL 版本 2 and 3，总是禁用的。

默认为TLSv1.2，在本文撰写时的行业最佳实践。

这个参数只能在lightdb.conf文件中或通过服务器命令行进行设置。

`ssl_max_protocol_version` (enum)

设定要使用的最大SSL/TLS协议版本。有效的版本为 `ssl_min_protocol_version`，添加一个空字符串，允许任何协议版本。默认为允许任何版本。设置最大协议版本主要用于测试，或者某个组件在与较新的协议配合工作时出现了问题。

此参数只能在lightdb.conf文件或服务器命令行中设置。

`ssl_dh_params_file` (string)

指定含有用于SSL密码的所谓临时DH家族的Diffie-Hellman参数的文件名。默认值为空，这种情况下将使用内置的默认DH参数。使用自定义的DH参数可以降低攻击者破解众所周知的内置DH参数的风险。可以用命令`openssl dhparam -out dhparams.pem 2048`创建自己的DH参数文件。

这个参数只能在 `lightdb.conf` 文件中或服务器命令行上进行设置。

`ssl_passphrase_command` (string)

设置当需要一个密码（例如一个私钥）来解密SSL文件时会调用的一个外部命令。默认情况下，这个参数为空，表示使用内建的提示机制。

该命令必须将密码打印到标准输出并且以代码0退出。在该参数值中，`%p`被替换为一个提示字符串（要得到文字`%`，应该写成`%%`）。注意该提示字符串将可能含有空格，因此要确保加上适当的引号。如果输出的末尾有单一的新行，它会被剥离掉。

该命令实际上并不一定要提示用户输入一个密码。它可以从文件中读取密码、从钥匙链得到密码等等。确保选中的机制足够安全是用户的责任。

这个参数只能在 `lightdb.conf` 文件中或服务器命令行上进行设置。

`ssl_passphrase_command_supports_reload` (boolean)

这个参数决定在配置重载期间如果一个密钥文件需要口令时，是否也调用`ssl_passphrase_command`设置的密码命令。如果这个参数为`off`（默认），那么在重载期间将忽略`ssl_passphrase_command`，如果在此期间需要密码则SSL配置将不会被重载。对于要求一个TTY（当服务器正在运行时可能是不可用的）来进行提示的命令，这种设置是合适的。例如，如果密码是从一个文件中得到的，将这个参数设置为`on`可能是合适的。

这个参数只能在`lightdb.conf`文件中或者服务器命令行上设置。

## 17.4. 资源消耗

### 17.4.1. 内存

`shared_buffers` (integer)

设置数据库服务器将使用的共享内存缓冲区量。默认通常是 128 兆字节（128MB），但是如果你的内核设置不支持（在`initdb`时决定），那么可以会更少。这个设置必须至少为 128 千字节。不过为了更好的性能，通常会使用明显高于最小值的设置。如果指定值时没有单位，则以块为单位，即`BLCKSZ`字节，通常为8kB。（`BLCKSZ` 的非默认值改变最小值。）此参数只能在服务器启动时设置。

如果有一个专用的 1GB 或更多内存的数据库服务器，一个合理的`shared_buffers`开始值是系统内存的 25%。即使更大的`shared_buffers`有效，也会造成一些工作负载，但因为LightDB同样依赖操作系统的高速缓冲区，将`shared_buffers`设置为超过40%的RAM不太可能比一个小点值工作得更好。为了能把对写大量新的或改变的数据的处理分布在一个较长的时间段内，`shared_buffers`更大的设置通常要求对`max_wal_size`也做相应增加。

如果系统内存小于 1GB，一个较小的 RAM 百分数是合适的，这样可以为操作系统留下足够的空间。

`huge_pages` (enum)

控制是否为主共享内存区域请求巨型页。有效值是`try`（默认）、`on`以及`off`。如果`huge_pages`被设置为`try`，则服务器将尝试请求巨型页，但是如果失败会退回到默认的

方式。如果为on，请求巨型页失败将使得服务器无法启动。如果为off，则不会请求巨型页。

当前，只有Linux和Windows上支持这个设置。在其他系统上这个参数被设置为try时，它会被忽略。在Linux中，它只在shared\_memory\_type设置为mmap（默认）的时候被支持。

巨型页面的使用会导致更小的页面表以及花费在内存管理上的CPU时间更少，从而提高性能。更多有关Linux上使用巨型页面的细节请见Section 16.4.5

巨型页在Windows上被称为大页面。要使用大页面，需要为运行LightDB的Windows用户账号分配“Lock pages in memory”的用户权限。可以使用Windows的组策略工具（gpedit.msc）来分配用户权限“Lock pages in memory”。为了在命令窗口以单进程（而不是Windows服务）的方式启动数据库服务器，命令窗口必须以管理员身份运行或者禁用用户访问控制（UAC）。当UAC被启用时，普通的命令窗口会在启动时收回用户权限“Lock pages in memory”。

注意这种设置仅影响主共享内存区域。Linux、FreeBSD以及Illumos之类的操作系统也能作为普通内存分配自动使用巨型页（也被称为“超级”页或者“大”页面），而不需要来自LightDB的显式请求。在Linux上，这被称为“transparent huge pages”（THP，透明巨型页）。已知这种特性对某些Linux版本上的某些用户会导致LightDB的性能退化，因此当前并不鼓励使用它（与huge\_pages的显式使用不同）。

#### lt\_ring\_buffers (integer)

该参数设置了每个数据库会话中用于环形缓冲区的最大内存量。当运行需要访问大量页（超过shared\_buffers总页数的四分之一）仅一次的顺序扫描查询时，使用一种称为bulk-read的特殊访问策略。只有通过这种扫描访问过的页面不太可能在不久的将来再次需要，因此不会清空整个缓冲高速缓存，而是分配由lt\_ring\_buffers指定的小型缓冲环，并将这些缓冲区用于整个扫描。这还意味着由这样的语句引起的大部分写入流量将由后端进程本身完成，而不是推迟至其他进程处理。

如果未指定单位而设置此值，则将其视为块，即BLCKSZ字节，通常为8kB。默认值为256千字节（256kB）。（如果BLCKSZ不是8kB，则默认值会按比例缩放。）此设置可以在单个会话中更改但请注意，环形缓冲区的实际大小是由min(shared\_buffers / 8, lt\_ring\_buffers)确定的，以确保环形缓冲区不是共享缓冲区的过度分数。

#### temp\_buffers (integer)

为每个数据库会话设置用于临时缓冲区的最大内存。这些是仅用于访问临时表的会话本地缓冲。如果指定值时没有单位，则以块为单位，即BLCKSZ字节，通常为8kB。默认为8兆字节（8MB）。（如果BLCKSZ不是8kB，则默认值按比例缩放。）这个设置可以在独立的会话内部被改变，但是只有在会话第一次使用临时表之前才能改变；在会话中随后企图改变该值是无效的。

一个会话将按照temp\_buffers给出的限制根据需要分配临时缓冲区。如果在一个并不需要大量临时缓冲区的会话里设置一个大的数值，其开销只是一个缓冲区描述符，或者说temp\_buffers每增加一则增加大概64字节。不过，如果一个缓冲区被实际使用，那么它就会额外消耗8192字节（或者BLCKSZ字节）。

#### max\_prepared\_transactions (integer)

设置可以同时处于“prepared”状态的事务的最大数目（见PREPARE TRANSACTION）。把这个参数设置为零（这是默认设置）将禁用预备事务特性。这个参数只能在服务器启动时设置。

如果你不打算使用预备事务，可以把这个参数设置为零来防止意外创建预备事务。如果你正在使用预备事务，你将希望把max\_prepared\_transactions至少设置为max\_connections一样大，因此每一个会话可以有一个预备事务待处理。

当运行一个后备服务器时，这个参数必须至少与主服务器上的一样大。否则，后备服务器上将不会执行查询。

`work_mem` (integer)

设置在写入临时磁盘文件之前查询操作(例如排序或哈希表)可使用的基础最大内存容量。如果指定值时没有单位,则以千字节为单位。默认值是4兆字节(4MB)。注意对于一个复杂查询,可能会并行运行好几个排序或者哈希操作;每个操作通常都会被允许使用这个参数指定的内存量,然后才会开始写数据到临时文件。同样,几个正在运行的会话可能并发进行这样的操作。因此被使用的总内存可能是`work_mem`值的好几倍,在选择这个值时一定要记住这一点。ORDER BY、DISTINCT和归并连接都要用到排序操作。哈希连接、基于哈希的聚集,结果缓存节点以及基于哈希的IN子查询处理中都要用到哈希表。

与等效的基于排序的操作相比,基于哈希的操作通常对内存可用性更敏感。可用于哈希表的内存通过将`work_mem`乘`hash_mem_multiplier`来计算。这使得基于哈希的操作可以使用超出通常`work_mem`基本量的内存。

`lightdb_keep_temp_file` (boolean)

当查询操作使用的基本最大内存量是不够的时,允许保留临时文件。

`lt_list_temp_file`可用于列出临时文件,例如:选择`lt_list_temp_file ('*')`。

`lt_remove_temp_file`可用于删除临时文件,例如:选择`lt_remove_temp_file ('*')`。

注意: 's' 用于当前会话, '\*' 用于所有会话。

`hash_mem_multiplier` (浮点)

用于计算基于哈希的操作可以使用的最大内存量。最终限制通过将`work_mem`乘以`hash_mem_multiplier`来决定。默认值为 1.0,这使得基于hash的操作与基于排序的操作一样,都取决于简单的`work_mem`最大值。。

考虑在经常发生溢出的环境中增加`hash_mem_multiplier`,尤其是在简单增加 `work_mem`会导致内存压力(内存压力通常以间歇性的内存退出错误的形式出现)。设置为 1.5 或 2.0 对混合作负载有效。在 2.0 - 8.0 或更多范围内的较高设置在已`work_mem`已经达到40MB 或更多的环境中可能有效。

`maintenance_work_mem` (integer)

指定在维护性操作(例如VACUUM、CREATE INDEX和ALTER TABLE ADD FOREIGN KEY)中使用的最大的内存量。如果指定值时没有单位,则以千字节为单位,其默认值是64 兆字节(64MB)。因为在一个数据库会话中,一个时刻只有一个这样的操作可以被执行,并且一个数据库安装通常不会有太多这样的操作并发执行,把这个数值设置得比`work_mem`大很多是安全的。更大的设置可以改进清理和恢复数据库转储的性能。

注意当自动清理运行时,可能会分配最多达这个内存的`autovacuum_max_workers`倍,因此要小心不要把该默认值设置得太高。通过独立地设置`autovacuum_work_mem`可能会对控制这种情况有所帮助。

注意,对于死亡元组标识的集合,VACUUM最多只能使用1GB的内存。

`autovacuum_work_mem` (integer)

指定每个自动清理工作者进程能使用的最大内存量。如果指定值时没有单位,则以千字节为单位。其默认值为 -1,表示转而使用 `maintenance_work_mem`的值。当运行在其他上下文环境中时,这个设置对VACUUM的行为没有影响。这个参数只能在`lightdb.conf`中或者服务器命令行上设置。

对于死亡元组标识的集合, `autovacuum`最大只能使用1GB内存,所以设置`autovacuum_work_mem`到一个更高的值,对于`autovacuum`在扫描表时所能收集的死亡元组数量是无效的。

`logical_decoding_work_mem` (integer)

指定逻辑解码要使用的最大内存量,在将某些解码的更改写入本地磁盘之前。这将限制逻辑流复制连接使用的内存量。它默认为 64 兆字节(64MB)。由于每个复制连接仅使



用此大小的单个缓冲区，并且安装通常不会同时具有多个此类连接（受 `max_wal_senders` 的限制），因此将此值设置得明显高于 `work_mem` 是安全的，从而减少写入磁盘的解码更改数量。

`max_stack_depth` (integer)

指定服务器的执行堆栈的最大安全深度。这个参数的理想设置是由内核强制的实际栈尺寸限制（`ulimit -s` 所设置的或者本地等价物），减去大约一兆字节的安全边缘。需要这个安全边缘是因为在服务器中并非所有例程都检查栈深度，只是在关键的可能递归的例程中才进行检查。如果指定值时没有单位，则以千字节为单位。默认设置是两兆字节（2MB），这个值相对比较小并且不可能导致崩溃。但是，这个值可能太小了，以至于无法执行复杂的函数。只有超级用户可以修改这个设置。

把 `max_stack_depth` 参数设置得高于实际的内核限制将意味着一个失控的递归函数可能会导致一个独立的后端进程崩溃。在 LightDB 能够检测内核限制的平台，服务器将不允许把这个参数设置为一个不安全的值。不过，并非所有平台都能提供该信息，所以我们还是建议你在选择值时要小心。

`shared_memory_type` (enum)

指定服务器应用于主共享内存区域的共享内存实现，包括 LightDB 的共享缓冲区和其他共享数据。可能的值为 `mmap`（对使用 `mmap` 分配的匿名共享内存）、`sysv`（通过 `shmget` 分配的 System V 共享内存）和 `windows`（Windows 共享内存）。并非在所有平台上都支持全部值；第一个被支持的选项是该平台的默认选项。`sysv` 选项不是任何平台的默认选项，通常不建议使用，因为它通常需要非默认的内核设置来允许大量的地址分配（参见 Section 16.4.1）。

`dynamic_shared_memory_type` (enum)

指定服务器应该使用的动态共享内存实现。可能的值是 `posix`（用于使用 `shm_open` 分配的 POSIX 共享内存）、`sysv`（用于通过 `shmget` 分配的 System V 共享内存）、`windows`（用于 Windows 共享内存）和 `mmap`（使用存储在数据目录中的内存映射文件模拟共享内存）。并非所有平台上都支持所有值，平台上第一个支持的选项就是其默认值。在任何平台上 `mmap` 选项都不是默认值，通常不鼓励使用它，因为操作系统会反复地把修改过的页面写回到磁盘上，从而增加了系统的 I/O 负载。不过当 `lt_dynshmem` 目录被存储在一个 RAM 盘时或者没有其他共享内存功能可用时，它还是有用的。

## 17.4.2. 磁盘

`temp_file_limit` (integer)

指定一个进程能用于临时文件（如排序和哈希临时文件，或者用于保持游标的存储文件）的最大磁盘空间量。一个试图超过这个限制的事务将被取消。如果指定值时没有单位，则以千字节为单位。`-1`（默认值）意味着没有限制。只有超级用户能够修改这个设置。

这个设置约束着一个给定 LightDB 进程在任何瞬间所使用的的所有临时文件的总空间。应该注意的是，与在查询执行中在幕后使用的临时文件相反，显式临时表所用的磁盘空间不被这个设置所限制。

## 17.4.3. 内核资源使用

`max_files_per_process` (integer)

设置每个服务器子进程允许同时打开的最大文件数目。默认是 1000 个文件。如果内核强制一个安全的针对每个进程的限制，那么就不用操心这个设置。但是在一些平台上（特别是大多数 BSD 系统），如果很多进程都尝试打开很多文件，内核将允许独立进程打开比系统真正可以支持的数目大得多得文件数。如果你发现自己看到了“Too many open files”这样的失败，可尝试减小这个设置。这个参数只能在服务器启动时设置。

## 17.4.4. 基于代价的清理延迟

在VACUUM和ANALYZE命令的执行过程中，系统维持着一个内部计数器来跟踪各种被执行的I/O操作的估算开销。当累计的代价达到一个限制（由vacuum\_cost\_limit指定），执行这些操作的进程将按照vacuum\_cost\_delay所指定的休眠一小段时间。然后它将重置计数器并继续执行。

这个特性的出发点是允许管理员降低这些命令对并发的数据库活动产生的I/O影响。在很多情况下，VACUUM和ANALYZE等维护命令能否快速完成并不重要，而非常重要是这些命令不会对系统执行其他数据库操作的能力产生显著的影响。基于代价的清理延迟提供了一种方式让管理员能够保证这一点。

对于手动发出的VACUUM命令，该特性默认被禁用。要启用它，只要把vacuum\_cost\_delay变量设为一个非零值。

vacuum\_cost\_delay (floating point)

当超出开销限制时进程将要休眠的时间量。如果指定值时没有单位，则以毫秒为单位。其默认值为0，这将禁用基于代价的清理延迟特性。正值将启用基于代价的清理。

在使用基于代价的清理时，vacuum\_cost\_delay的合适值通常很小，也许是小于1毫秒。虽然vacuum\_cost\_delay可以被设置为毫秒级别的值，但是在较老的平台上可能无法准确地测量这种延迟。在这样的平台上，增加VACUUM的节流资源消耗在1ms以上，需要改变其他的清理开销参数。尽管如此，你应该保持vacuum\_cost\_delay在平台能持续测量的情况下尽可能小；大延迟没有帮助。

vacuum\_cost\_page\_hit (integer)

清理一个在共享缓存中找到的缓冲区的估计代价。它表示锁住缓冲池、查找共享哈希表和扫描页内容的代价。默认值为1。

vacuum\_cost\_page\_miss (integer)

清理一个必须从磁盘上读取的缓冲区的代价。它表示锁住缓冲池、查找共享哈希表、从磁盘读取需要的块以及扫描其内容的代价。默认值为2。

vacuum\_cost\_page\_dirty (integer)

当清理修改一个之前干净的块时需要花费的估计代价。它表示再次把脏块刷出到磁盘所需要的额外I/O。默认值为20。

vacuum\_cost\_limit (integer)

将导致清理进程休眠的累计代价。默认值为200。

### Note

有些操作会保持关键性的锁，这样可以尽快完成。基于代价的清理延迟在这类操作期间不会发生。因此有可能代价会累计至大大超过指定的限制。为了防止这种情况下的无意义的长时间延迟，实际延迟的计算方式是 $\text{vacuum\_cost\_delay} * \text{accumulated\_balance} / \text{vacuum\_cost\_limit}$ ，且最大值是 $\text{vacuum\_cost\_delay} * 4$ 。

## 17.4.5. 后台写入器

有一个独立的服务器进程，叫做后台写入器，它的功能就是发出写“脏”（新的或修改过的）共享缓冲区的命令。当干净的共享缓存数量出现不足时，后台写入器写入一些

脏缓存到文件系统，并标记为干净。不过，后台写入器确实会增加 I/O 的总负荷，因为虽然在每个检查点间隔中一个重复弄脏的页面可能只会写出一次，但在同一个间隔中后台写入器可能会把它写出好几次。在这一小节讨论的参数可以被用于调节本地需求的行为。

`bgwriter_delay` (integer)

指定后台写入器活动轮次之间的延迟。在每个轮次中，写入器都会为一定数量的脏缓冲区发出写操作（可以用下面的参数控制）。然后它就休眠 `bgwriter_delay` 的时长，然后重复动作。当缓冲池中没有脏缓冲区时，不管 `bgwriter_delay`，它都会进入更长的休眠。如果指定值时没有单位，则以毫秒为单位。默认值是 200 毫秒 (200ms)。注意在许多系统上，休眠延迟的有效解析度是 10 毫秒；因此，为 `bgwriter_delay` 设置一个不是 10 的倍数的值与把它设置为下一个更高的 10 的倍数是一样的效果。这个选项只能在服务器命令行上或者在 `lightdb.conf` 文件中设置。

`bgwriter_lru_maxpages` (integer)

在每个轮次中，不超过这么多个缓冲区将被后台写入器写出。把这个参数设置为零可禁用后台写出（注意被一个独立、专用辅助进程管理的检查点不受影响）。默认值是 100 个缓冲区。这个参数只能在 `lightdb.conf` 文件中或在服务器命令行上设置。

`bgwriter_lru_multiplier` (floating point)

每一轮次要写的脏缓冲区的数目基于最近几个轮次中服务器进程需要的新缓冲区的数目。最近所需的平均值乘以 `bgwriter_lru_multiplier` 可以估算下一轮次中将会需要的缓冲区数目。脏缓冲区将被写出直到有很多干净可重用的缓冲区（然而，每一轮次中写出的缓冲区数不超过 `bgwriter_lru_maxpages`）。因此，设置为 1.0 表示一种“刚好好的”策略，这种策略会写出正好符合预测值的数目的缓冲区。更大的值可以为需求高峰提供某种缓冲，而更小的值则需要服务进程来处理一些写出操作。默认值是 2.0。这个参数只能在 `lightdb.conf` 文件中或在服务器命令行上设置。

`bgwriter_flush_after` (integer)

只要后台写入的数据超过这个数量，尝试强制 OS 把这些写发送到底层存储上。这样做将限制内核页缓存中脏数据的量，降低了在检查点末尾发出一个 `fsync` 时或者 OS 在后台大批量写回数据时卡住的可能性。那常常会导致大幅度压缩的事务延迟，但是也有一些情况（特别是负载超过 `shared_buffers` 但小于 OS 页面高速缓存）的性能会降低。这种设置可能会在某些平台上没有效果。如果指定值时没有单位，则以块为单位，即为 `BLCKSZ` 字节，通常为 8kB。合法的范围在 0（禁用受控写回）和 2MB 之间。Linux 上的默认值是 512kB，其他平台上是 0（如果 `BLCKSZ` 不是 8kB，则默认值和最大值会按比例缩放至这个值）。这个参数只能在 `lightdb.conf` 文件中或者服务器命令行上设置。

较小的 `bgwriter_lru_maxpages` 和 `bgwriter_lru_multiplier` 可以降低由后台写入器造成的额外 I/O 开销。但更可能的是，服务器进程将必须自己发出写入操作，这会延迟交互式查询。

## 17.4.6. 异步行为

`effective_io_concurrency` (integer)

设置 LightDB 可以同时被执行的并发磁盘 I/O 操作的数量。调高这个值，可以增加任何单个 LightDB 会话试图并行发起的 I/O 操作的数目。允许的范围是 1 到 1000，或 0 表示禁用异步 I/O 请求。当前这个设置仅影响位图堆扫描。

对于磁盘驱动器，这个设置的一个很好的出发点是组成一个被用于该数据库的 RAID 0 条带或 RAID 1 镜像的独立驱动器数量（对 RAID 5 而言，校验驱动器不计入）。但是，如果数据库经常忙于在并发会话中发出的多个查询，较低的值可能足以使磁盘阵列繁忙。比保持磁盘繁忙所需的值更高的值只会造成额外的 CPU 开销。SSD 以及其他基于内存的存储常常能处理很多并发请求，因此它们的最佳值可能是数百。

异步 I/O 依赖于一个有效的`posix_fadvise`函数（一些操作系统可能没有）。如果不存在这个函数，将这个参数设置为除 0 之外的任何东西将导致错误。在一些操作系统上（如Solaris）虽然提供了这个函数，但它不会做任何事情。

在支持的系统上默认值为 1，否则为 0。对于一个特定表空间中的表，可以通过设定该表空间的同名参数（见ALTER TABLESPACE）可以覆盖这个值。

`maintenance_io_concurrency` (integer)

与`effective_io_concurrency`相似，但用于支持许多客户端会话完成的维护工作。

在支持的系统上的默认值为 10，否则为 0。对于特定表空间中的表，这个值可以被覆盖，通过设置同名的表空间参数（参见 ALTER TABLESPACE）。

`max_worker_processes` (integer)

设置系统能够支持的后台进程的最大数量。这个参数只能在服务器启动时设置。默认值为 8。

在运行一个后备服务器时，你必须把这个参数设置为等于或者高于主控服务器上的值。否则，后备服务器上可能不会允许查询。

在更改这个值时，考虑也

对`max_parallel_workers`、`max_parallel_maintenance_workers`以及`max_parallel_workers_per_gather`进行调整。

`max_parallel_workers_per_gather` (integer)

设置单个Gather或者Gather Merge节点能够开始的工作者的最大数量。并行工作者会从`max_worker_processes`建立的进程池中取得，数量由`max_parallel_workers`限制。注意所要求的工作者数量在运行时可能实际无法被满足。如果这种事情发生，该计划将会以比预期更少的工作者运行，这可能会不太高效。默认值是2。把这个值设置为0将会禁用并行查询执行。

注意并行查询可能消耗比非并行查询更多的资源，因为每一个工作者进程时一个完全独立的进程，它对系统产生的影响大致和一个额外的用户会话相同。在为这个设置选择值时，以及配置其他控制资源利用的设置（例如`work_mem`）时，应该把这个因素考虑在内。`work_mem`之类的资源限制会被独立地应用于每一个工作者，这意味着所有进程的总资源利用可能会比单个进程时高得多。例如，一个使用 4 个工作者的并行查询使用的CPU 时间、内存、I/O 带宽可能是不使用工作者时的 5 倍之多。

并行查询的更多信息请见Chapter 15

`max_parallel_maintenance_workers` (integer)

设置单一工具性命令能够启动的并行工作者的最大数目。当前，支持使用并行工作者的工具性命令是CREATE INDEX，并且只有在构建B-树索引时才能并行，并且 VACUUM 没有 FULL选项。并行工作者从由`max_worker_processes`创建的进程池中取出，数量由`max_parallel_workers`控制。注意实际在运行时所请求数量的工作者可能不可用。如果发生这种情况，工具性操作将使用比预期数量少的工作者运行。默认值为2。将这个值设置为0可以禁用工具性命令对并行工作者的使用。

注意并行工具性命令不应该消耗比同等数量非并行操作更多的内存。这种策略与并行查询不同，并行查询的资源限制通常是应用在每个工作者进程上。并行工具性命令把资源限制`maintenance_work_mem`当作对整个工具性命令的限制，而不管其中用到了多少个并行工作者进程。不过，并行工具性命令实际上可能仍会消耗更多的CPU资源和I/O带宽。

`max_parallel_workers` (integer)

设置系统为并行操作所支持的工作者的最大数量。默认值为8。在增加或者减小这个值时，也要考虑对`max_parallel_maintenance_workers`以及`max_parallel_workers_per_gather`进行调整。此外，要注意将这个值设置得大

于`max_worker_processes`将不会产生效果，因为并行工作者进程都是从`max_worker_processes`所建立的工作者进程池中取出来的。

`backend_flush_after` (integer)

每当单个后端写入的数据量超过此数量时，请尝试强制操作系统向底层存储发出这些写入。这样做将限制内核页面缓存中脏数据的数量，从而降低在检查点结束时发出`fsync`时，或者当操作系统在后台以更大的批数写回数据时，暂停的可能性。这通常会大大降低事务延迟，但也有一些情况，尤其是工作负载大于`shared_buffers`，但小于操作系统的页面缓存，性能可能会降低。此设置可能对某些平台没有影响。如果指定该值时没有单位，则将其作为块，即`BLCKSZ`字节，通常为8kB。有效范围介于禁用强制写回的0和2MB之间。默认值为0，即没有强制写回。（如果`BLCKSZ`不是8kB，则最大值与其成比例缩放。）

`old_snapshot_threshold` (integer)

设置在使用快照时，一个快照可以被使用而没有发生“snapshot too old”错误的风险的最小时间。超过此阈值时间的死数据将允许被清除。这可以有助于阻止长时间使用的快照造成的快照膨胀。为了阻止由于本来对该快照可见的数据被清理导致的不正确结果，当快照比这个阈值更旧并且该快照被用来读取一个该快照建立以来被修改过的页面时，将会产生一个错误。

如果指定值时没有单位，则以分钟为单位。值 `-1`（默认值）禁用此功能，实际上将快照的时限设置为无穷大。这个参数只能在服务器启动时设置。

对于生产工作有用的值的范围可能从几个小时到几天不等。只允许小的值（例如0或者1min）是因为它们有时可能对测试有用。虽然允许高达60d的设置，但是请注意很多负载情况下，很短的时间帧里就可能发生极大的膨胀或者事务 ID 回卷。

当这个特性被启用时，关系末尾的被清出的空间不能被释放给操作系统，因为那可能会移除用于检测“snapshot too old”情况所需的信息。所有分配给关系的空间还将与该关系关联在一起便于重用，除非它们被显式地释放（例如，用`VACUUM FULL`）。

这个设置不会尝试保证在任何特殊情况下都会生成错误。事实上，如果（例如）可以从一个已经物化了一个结果集的游标中生成正确的结果，即便被引用表中的底层行已经被清理掉也不会生成错误。某些表不能被过早地安全清除，并且因此将不受这个设置的影响，例如系统目录。对于这些表，这个设置将不能降低膨胀，也不能降低在扫描时产生“snapshot too old”错误的可能性。

## 17.5. 预写式日志

参阅Section 27. 获取调节这些设置的额外信息。

### 17.5.1. 设置

`wal_level` (enum)

`wal_level`决定多少信息写入到 WAL 中。默认值是`replica`，它会写入足够的信息以支持 WAL 归档和复制，包括在后备服务器上运行只读查询。`minimal`会去掉除从崩溃或者立即关机中进行恢复所需的信息之外的所有记录。最后，`logical`会增加支持逻辑解码所需的信息。每个层次包括所有更低层次记录的信息。这个参数只能在服务器启动时设置。

在`minimal`级别中，对于创建或重写永久关系的事务的其余部分，不会记录任何信息。这可以使那些操作更快（参见Section 14.4.7）。初始化这种优化的操作包括：

```
ALTER ... SET TABLESPACE
CLUSTER
CREATE TABLE
REFRESH MATERIALIZED VIEW (without CONCURRENTLY)
REINDEX
```

## TRUNCATE

但最少的 `WAL` 不会包括足够的信息来从基础备份和 `WAL` 日志中重建数据，因此，要启用 `WAL` 归档 (`archive_mode`) 和流复制，必须使用 `replica` 或更高级别。注意改变 `wal_level` 为 `minimal` 导致归档恢复和备用服务器之前的基本备份失效，也可能导致数据丢失。

在 `logical` 层，与 `replica` 相同的信息会被记录，外加上 允许从 `WAL` 抽取逻辑修改集所需的信息。使用级别 `logical` 将增加 `WAL` 容量，特别是如果为了 `REPLICA IDENTITY FULL` 配置了很多表并且执行了很多 `UPDATE` 和 `DELETE` 语句时。

在 9.6 之前的版本中，这个参数也允许值 `archive` 和 `hot_standby`。现在仍然接受这些值，但是它们会被映射到 `replica`。

## fsync (boolean)

如果打开这个参数，LightDB服务器将尝试确保更新被物理地写入到磁盘，做法是发出 `fsync()` 系统调用或者使用多种等价的方法（见 `wal_sync_method`）。这保证了数据库集簇在一次操作系统或者硬件崩溃后能恢复到一个一致的状态。

虽然关闭 `fsync` 常常可以得到性能上的收益，但当发生断电或系统崩溃时可能造成不可恢复的数据损坏。因此，只有在能很容易地从外部数据中重建整个数据库时才建议关闭 `fsync`。

能安全关闭 `fsync` 的环境的例子包括从一个备份文件中初始加载一个新数据库集簇、使用一个数据库集簇来在数据库被删掉并重建之后处理一批数据，或者一个被经常重建但并不用于失效备援的只读数据库克隆。单独的高质量硬件不足以成为关闭 `fsync` 的理由。

当把 `fsync` 从关闭改成打开时，为了可靠的恢复，需要强制在内核中的所有被修改的缓冲区进入持久化存储。这可以在多个时机来完成：在集簇被关闭时或在 `fsync` 因为运行 `initdb --sync-only` 而打开时、运行 `sync` 时、卸载文件系统时或者重启服务器时。

在很多情况下，为不重要的事务关闭 `synchronous_commit` 可以提供很多关闭 `fsync` 的潜在性能收益，并不会有的同时，关闭 `fsync` 可以提供很多潜在的性能优势，而不会有伴随着的数据损坏风险。

`fsync` 只能在 `lightdb.conf` 文件中或在服务器命令行上设置。如果你关闭这个参数，请也考虑关闭 `full_page_writes`。

## synchronous\_commit (enum)

指定数据库服务器返回 “success” 指示给客户端之前，必须要完成多少 `WAL` 处理。合法的值为 `remote_apply`, `on` (默认值), `remote_write`, `local`, 和 `off`。

如果 `synchronous_standby_names` 为空，则唯一有意义的设置为 `on` 和 `off`；`remote_apply`, `remote_write` 和 `local` 都提供与 `on` 相同的本地同步级别。所有非 `off` 模式的本地行为都是等待 `WAL` 的本地刷新到磁盘。在 `off` 模式，无需等待，因此在向客户端报告成功和以后保证事务安全防止服务器崩溃之间可能会出现延迟。当设置为 `off` 时，在向客户端报告成功和真正保证事务不会被服务器崩溃威胁之间会有延迟（最大的延迟是 `wal_writer_delay` 的三倍）。不同于 `fsync`，将这个参数设置为 `off` 不会产生数据库不一致性的风险：一个操作系统或数据库崩溃可能会造成一些最近据说已提交的事务丢失，但数据库状态是一致的，就像这些事务已经被干净地中止。因此，当性能比完全确保事务的持久性更重要时，关闭 `synchronous_commit` 可以作为一个有效的代替手段。更多讨论见 Section 27.3

如果 `synchronous_standby_names` 为非空，`synchronous_commit` 也控制是否事务提交将等待它们的 `WAL` 记录在后备服务器上被处理。

当设置为 `remote_apply` 时，提交将等待，直到来自当前同步备用服务器的答复显示他们已收到事务的提交记录并应用了它，以便它变得对备用服务器上的查询可见，并写入备用服务器上的持久存储。这将导致比以前的设置更大的提交延迟，因为它等待 `WAL`

重放(replay)。当设置为on时,提交将等待,直到来自于当前同步的后备服务器的回复显示它们已经收到了事务的提交记录并将其刷入了磁盘。这保证事务将不会被丢失,除非主服务器和所有同步后备都遭受到了数据库存储损坏的问题。当这个参数被设置为remote\_write时,提交将等待,直到来自当前的同步后备的回复指示它们已经收到了该事务的提交记录并且已经把该记录写到它们的文件系统,这种设置保证数据得以保存,在LightDB的后备服务器实例崩溃时,但是不能保证后备服务器遭受操作系统级别崩溃时数据能被保持,因为数据不一定必须要在后备机上达到持久存储。设置local会导致提交等待本地刷写到磁盘,而不是复制。在使用同步复制时这通常是不可取的,但是为了完整性提供了这个选项。

这个参数可以随时被修改;任何一个事务的行为由其提交时生效的设置决定。因此,可以同步提交一些事务,同时异步提交其他事务。例如,当默认是相反时,实现一个单一多语句事务的异步提交,在事务中发出SET LOCAL synchronous\_commit TO OFF。

Table 17.1概括了 synchronous\_commit 设置的能力。

Table 17.1. synchronous\_commit Modes

synchronous_commit setting	local durable commit	standby durable commit after PG crash	standby durable commit after OS crash	standby query consistency
remote_apply	•	•	•	•
on	•	•	•	
remote_write	•	•		
local	•			
off				

wal\_sync\_method (enum)

用来向强制 WAL 更新到磁盘的方法。如果fsync是关闭的,那么这个设置就不相关,因为 WAL 文件更新将根本不会被强制。可能的值是:

- open\_datasync (用open()选项O\_DSYNC写 WAL 文件)
- fdatasync (在每次提交时调用fdatasync())
- fsync (在每次提交时调用fsync())
- fsync\_writethrough (在每次提交时调用fsync(), 强制任何磁盘写高速缓存的直写)
- open\_sync (用open()选项O\_SYNC写 WAL 文件)

open\_\* 选项也可以使用O\_DIRECT(如果可用)。不是在所有平台上都能使用所有这些选择。默认值是列表中第一个被平台支持的那个,不过fdatasync是Linux和FreeBSD中的默认值。默认值不一定是理想的;有可能需要修改这个设置或系统配置的其他方面来创建一个崩溃-安全的配置,或达到最佳性能。这些方面在Section 27.中讨论。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

full\_page\_writes (boolean)

当这个参数为打开时,LightDB服务器在一个检查点之后的页面的第一次修改期间将每个页面的全部内容写到 WAL 中。这么做是因为在操作系统崩溃期间正在处理的一次页写入可能只有部分完成,从而导致在一个磁盘页面中混合有新旧数据。在崩溃后的恢复期间,通常存储在 WAL 中的行级改变数据不足以完全恢复这样一个页面。存储完整的页面映像可以保证页面被正确存储,但代价是增加了必须被写入 WAL 的数据量(因为 WAL 重放总是从一个检查点开始,所以在检查点后每个页面的第一次改变时这样做就够了。因此,一种减小全页面写开销的方法是增加检查点间隔参数值)。

把这个参数关闭会加快正常操作，但是在系统失败后可能导致不可恢复的数据损坏，或者静默的数据损坏。其风险类似于关闭`fsync`，但是风险较小。并且只有在可关闭`fsync`的情况下才应该关闭它。

关闭这个选项并不影响用于时间点恢复（PITR）的 WAL 归档使用（见Section 23.3）。

这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。默认值是`on`。

`wal_log_hints` (boolean)

当这个参数为`on`时，LightDB服务器一个检查点之后页面被第一次修改期间把该磁盘页面的整个内容都写入 WAL，即使对所谓的提示位做非关键修改也会这样做。

如果启用了数据校验和，提示位更新总是会被 WAL 记录并且这个设置会被忽略。你可以使用这个设置测试如果你的数据库启用了数据校验和，会有多少额外的 WAL 记录发生。

这个参数只能在服务器启动时设置。默认值是`off`。

`wal_compression` (boolean)

当这个参数为`on`时，如果`full_page_writes` 为打开或者处于基础备份期间，LightDB服务器 会压缩写入到 WAL 中的完整页面镜像。压缩页面镜像将在 WAL 重放时 被解压。默认值为`off`。只有超级用户可以更改这个设置。

打开这个参数可以减小 WAL 所占的空间且无需承受不可恢复的数据损坏风险，但是代价是需要额外的 CPU 开销以便在 WAL 记录期间进行压缩以及在 WAL 重放时解压。

`wal_init_zero` (boolean)

如果设置为`on`（默认值），此选项会导致新的 WAL 文件被零填充。在某些文件系统中，这可确保在我们需要写入 WAL 记录之前分配空间。但是，Copy-On-Write (COW) 文件系统可能不会从此技术中受益，因此可以选择跳过不必要的工作。如果设置为`off`，则在创建文件时仅写入最终字节，以便其具有预期大小。

`wal_recycle` (boolean)

如果设置为 `on`（默认值），此选项通过重命名来回收 WAL 文件，从而避免创建新文件。在 COW 文件系统中，创建新文件系统可能更快，因此提供了禁用此行为的选项。

`wal_buffers` (integer)

用于还未写入磁盘的 WAL 数据的共享内存量。默认值 `-1` 选择等于`shared_buffers`的  $1/32$  的尺寸（大约3%），但是不小于64kB也不大于 WAL 段的尺寸（通常为）。如果自动的选择太大或太小可以手工设置该值，但是任何小于32kB的正值都将被当作32kB。如果指定值时没有单位，则以WAL块作为单位，即为 `XLOG_BLCKSZ` 字节，通常为8kB。这个参数只能在服务器启动时设置。

在每次事务提交时，WAL 缓冲区的内容被写出到磁盘，因此极大的值不可能提供显著的收益。不过，把这个值设置为几个兆字节可以在一个繁忙的服务器（其中很多客户端会在同一时间提交）上提高写性能。由默认设置 `-1` 选择的自动调节将在大部分情况下得到合理的结果。

`wal_writer_delay` (integer)

指定 WAL 写入器刷写 WAL 的频繁程度，以时间为单位。在刷写WAL之后，写入器将根据`wal_writer_delay`所给出的时间长度进行睡眠，除非被一个异步提交的事务提前唤醒。如果最近的刷写发生在 `wal_writer_delay` 之前，并且小于 `wal_writer_flush_after` WAL的值产生之后，那么WAL只会被写入操作系统，而不会被刷写到磁盘。如果指定值时没有单位，则以毫秒作为单位。默认值是 `200` 毫秒（200ms）。注意在很多系统上，有效的睡眠延迟粒度是 `10` 毫秒，把`wal_writer_delay`设置为一个不是 `10` 的倍数的值，其效果和把它设置为大于该值的下一个 `10` 的倍数产生的效果相同。这个参数只能在`lightdb.conf`文件中或者服务器命令行上设置。



wal\_writer\_flush\_after (integer)

指定 WAL 写入器刷写 WAL 的频繁程度，以卷为单位。如果最近的刷写发生在 wal\_writer\_delay 之前，并且小于 wal\_writer\_flush\_after WAL 的值产生之后，那么 WAL 只会被写入操作系统，而不会被刷写到磁盘。如果 wal\_writer\_flush\_after 被设置为 0，则 WAL 数据总是会被立即刷写。如果指定值时没有单位，则以 WAL 块作为单位，即为 XLOG\_BLCKSZ 字节，通常为 8kB。默认是 1MB。这个参数只能在 lightdb.conf 文件中或者服务器命令行上设置。

wal\_skip\_threshold (integer)

当 wal\_level 为 minimal，并且在创建或重写永久关系之后提交事务时，此设置将确定如何保留新数据。如果数据小于此设置，将其写入 WAL 日志；否则，使用受影响文件的 fsync。根据存储的属性，如果此类提交减慢了并发事务，提高或降低此值可能会有所帮助。如果指定此值时没有单位，则视为千字节。默认为两兆字节（2MB）。

commit\_delay (integer)

在一次 WAL 刷写被发起之前，commit\_delay 增加一个时间延迟。如果系统负载足够高，使得在一个给定间隔内有额外的事务准备好提交，那么通过允许更多事务通过一个单次 WAL 刷写来提交能够提高组提交的吞吐量。但是，它也把每次 WAL 刷写的潜伏期增加到了最多 commit\_delay。因为如果没有其他事务准备好提交，就会浪费一次延迟，只有在当一次刷写将要被发起时有至少 commit\_siblings 个其他活动事务时，才会执行一次延迟。另外，如果 fsync 被禁用，则将不会执行任何延迟。如果指定值时没有单位，则以微秒作为单位。默认的 commit\_delay 是零（无延迟）。只有超级用户才能修改这个设置。

在 LightDB 的 9.3 发布之前，commit\_delay 的行为不同并且效果更差：它只影响提交，而不是所有 WAL 刷写，并且即使在 WAL 刷写马上就要完成时也会等待一整个配置的延迟。从 LightDB 9.3 中开始，第一个准备好刷写的进程会等待配置的间隔，而后续的进程只等到领先者完成刷写操作。

commit\_siblings (integer)

在执行 commit\_delay 延迟时，要求的并发活动事务的最小数目。大一些的值会导致在延迟间隔期间更可能有至少另外一个事务准备好提交。默认值是五个事务。

## 17.5.2. 检查点

checkpoint\_timeout (integer)

自动 WAL 检查点之间的最长时间。如果指定值时没有单位，则以秒为单位。合理的范围在 30 秒到 1 天之间。默认是 5 分钟（5min）。增加这个参数的值会增加崩溃恢复所需的时间。这个参数只能在 lightdb.conf 文件中或在服务器命令行上设置。

checkpoint\_completion\_target (floating point)

指定检查点完成的目标，作为检查点之间总时间的一部分。默认是 0.9，这将把检查点分布在几乎所有可用的时间间隔上，提供公平一致的 I/O 负载，同时也为检查点完成开销留下了一些时间。减少此参数是不被推荐的，因为这会导致检查点完成得更快。这个造成处于在检查点和下一个计划检查点之间较少 I/O 之后的检查点会有更高的 I/O 比例。这个参数只能在 lightdb.conf 文件中或在服务器命令行上设置。

checkpoint\_flush\_after (integer)

当执行检查点时写入的数据量超过此数量时，就尝试强制 OS 把这些写发送到底层存储。这样做将会限制内核页面高速缓存中的脏数据数量，降低在检查点末尾发出 fsync 或者 OS 在后台大批量写回数据时被卡住的可能性。那常常会导致大幅度压缩的事务延迟，但是也有一些情况（特别是负载超过 shared\_buffers 但小于 OS 页面高速缓存）的性能会降低。这

种设置可能会在某些平台上没有效果。如果指定值时没有单位，则以块为单位，即为BLCKSZ 字节，通常为8kB。合法的范围在0（禁用强制写回）和2MB之间。Linux 上的默认值是256kB，其他平台上是0（如果BLCKSZ不是8kB，则默认值和最大值会按比例缩放到它）。这个参数只能在lightdb.conf文件中或者服务器命令行上设置。

checkpoint\_warning (integer)

如果由于填充WAL段文件导致的检查点之间的间隔低于这个参数表示的时间量，那么就向服务器日志写一个消息（它建议增加max\_wal\_size的值）。如果指定值时没有单位，则以秒为单位。默认值是 30 秒（30s）。零则关闭警告。如果checkpoint\_timeout低于checkpoint\_warning，则不会有警告产生。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

max\_wal\_size (integer)

在自动 WAL 检查点之间允许 WAL 增长到的最大尺寸。这是一个软限制，在特殊的情况下 WAL 尺寸可能会超过max\_wal\_size，例如在重度负荷下、archive\_command失败或者高的 wal\_keep\_size设置。如果指定值时没有单位，则以兆字节为单位。默认为 1 GB。增加这个参数可能导致崩溃恢复所需的时间。这个参数只能在lightdb.conf 或者服务器命令行中设置。

min\_wal\_size (integer)

只要 WAL 磁盘用量保持在这个设置之下，在检查点时旧的 WAL 文件总是被回收以便未来使用，而不是直接被删除。这可以被用来确保有足够的 WAL 空间被保留来应付 WAL 使用的高峰，例如运行大型的批处理任务。如果指定值时没有单位，则以兆字节为单位。默认是 80 MB。这个参数只能在lightdb.conf 或者服务器命令行中设置。

### 17.5.3. 归档

archive\_mode (enum)

当启用archive\_mode时，可以通过设置 archive\_command命令将完成的 WAL 段发送到归档存储。除用于禁用的off之外，还有两种模式： on和always。在普通操作期间，这两种模式之间没有区别，但是当设置为always时，WAL 归档器在归档恢复 或者后备模式下也会被启用。在always模式下，所有从归档恢复 的或者用流复制传来的文件将被（再次）归档。详见 Section 24.2.9

archive\_mode和archive\_command是独立的变量，这样可以在不影响归档模式的前提下修改archive\_command。这个参数只能在服务器启动时设置。当wal\_level被设置为minimal时，archive\_mode不能被启用。

archive\_command (string)

本地 shell 命令被执行来归档一个完成的 WAL 文件段。字符串中的任何%p被替换成要被归档的文件的文件名，而%f只被文件名替换（路径名是相对于服务器的工作目录，即集簇的数据目录）。如果要在命令里嵌入一个真正的%字符，可以使用%。有一点很重要，该命令只在成功时返回一个零作为退出状态。更多信息请见Section 23.3.1

这个参数只能在lightdb.conf文件中或在服务器命令行上设置。除非服务器启动时启用了archive\_mode，否则它会被忽略。如果archive\_mode被启用时，archive\_command是一个空字符串（默认），WAL 归档会被临时禁用，但服务器仍会继续累计 WAL 段文件，期待着一个命令被提供。将archive\_command设置为一个只返回真但不做任何事的命令（例如/bin/true或 Windows 上的REM）实际上会禁用归档，也会打破归档恢复所需的 WAL 文件链，因此只有在极少数情况下才能用。

archive\_command不能包含通常用于存档清理的字符串'xargs -i rm'。如果包含它，LightDB将启动失败。

lightdb\_archive\_dir (string)

配置存档路径以清理存档文件，它必须与archive\_command中使用的路径相同。 档案目录只能是本地目录，而不是另一个主机上的目录。 不建议将其与备份一起使用。它可能会清理备份所需的存档文件。 与备份一起使用时，您可以设置两个存档目录，一个用于高可用性，一个用于备份。

此参数只能在服务器启动时设置。 在服务器启动时，只有 archive\_mode 和 archive\_command 为启用时才有效。 当 archive\_mode 和 archive\_command 已启用时，如果 lightdb\_archive\_dir 是一个空字符串（默认），WAL 禁止存档清理。

如果lightdb\_archive\_retention\_size不为0， 必须设置lightdb\_archive\_dir， 否则LightDB会启动失败。

如果设置了lightdb\_archive\_dir且archive\_command中包含'\$lightdb\_archive\_dir/%f'，同时archive\_command中包含'cp %p'，LightDB会做校验。 \$lightdb\_archive\_dir就表示 lightdb\_archive\_retention\_size中的字符串

lightdb\_archive\_retention\_size (integer)

清理早于 (checkpoint file - lightdb\_archive\_retention\_size) 的存档文件。 建议配置超过10（最大10000），具体取决于磁盘容量和主备之间的同步速率。

此参数只能在服务器启动时设置。 在服务器启动时，只有 lightdb\_archive\_dir 为启用时才有效。 当 lightdb\_archive\_dir 已启用时，如果 lightdb\_archive\_retention\_size 为0（默认值），WAL 禁止存档清理。

archive\_timeout (integer)

archive\_command仅在已完成的 WAL 段上调用。因此，如果你的服务器只产生很少的WAL 流量（或产生流量的周期很长），那么在事务完成和它被安全地记录到归档存储之间将有一个很长的延迟。为了限制未归档数据存在的时间，你可以设置archive\_timeout来强制服务器来周期性地切换到一个新的 WAL 段文件。 当这个参数被设置为大于零时，只要从上次段文件切换后过了参数所设置的时间量，并且已经有过任何数据库活动（包括一个单一检查点），服务器将切换到一个新的段文件（如果没有数据库活动则会跳过检查点）。 注意，由于强制切换而提早关闭的被归档文件仍然与完整的归档文件长度相同。因此，使用非常短的archive\_timeout是不明智的 — 它将占用巨大的归档存储。一分钟左右的archive\_timeout设置通常比较合理。如果你希望数据能被更快地从主服务器上复制下来，你应该考虑使用流复制而不是归档。如果指定值时没有单位，则以秒为单位。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

## 17.5.4. 归档恢复

本节描述了仅在恢复期间适用的设置。如果您希望执行任何后续恢复，则必须重置它们。

“Recovery” 涵盖使用服务器作为备用服务器或用于执行目标恢复。 通常情况，备用模式用于提供高可用性和/或读可扩展性，而目标恢复用于从数据丢失中恢复。

若要在备用模式下启动服务器，在数据目录中建立名为standby.signal的文件。 服务器将会进入恢复状态并且在到达归档WAL末尾时不会停止恢复，但将保持尝试继续恢复，通过连接到primary\_conninfo设置指定的发送服务器和/或用restore\_command获取新的WAL分段。对于这种模式，来自本节的参数和Section 17.6.3是值得关注的。 Section 17.5.5中的参数也会被应用，但通常在这种模式下没用。

要启动服务器为目标恢复模式，需在数据目录中建立名为recovery.signal的文件。 如果同时创建了standby.signal 和 recovery.signal 文件，则优先使用备用模式。 目标恢复模式在归档的WAL全部回放或到达recovery\_target时结束。 在这种模式下，将使用来自本节和Section 17.5.5的参数。

restore\_command (string)

用于获取 WAL 文件系列的一个已归档段的本地 shell 命令。这个参数是归档恢复所必需的，但是对于流复制是可选的。在该字符串中的任何%f会被替换为从归档中获得的文件的名称，并且任何%p会被在服务器上的复制目标路径名替换（该路径名是相对于当前工作目录的，即集簇的数据目录）。任何%r会被包含上一个可用重启点的文件的名称所替换。在那些必须被保留用于使得一次恢复变成可重启的文件中，这个文件是其中最早的一个，因此这个信息可以被用来把归档截断为支持从当前恢复重启所需的最小值。%r通常只被温备配置（见Section 24.2 所使用。要嵌入一个真正的%字符，需要写成%%。

很重要的一点是，该命令只有在成功时才返回一个为零的退出状态。该命令将会被询问不存在于归档中的文件名，当这样被询问时它必须返回非零。例子：

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"' # Windows
```

一个例外是如果该命令被一个信号（不是SIGTERM，它是数据库服务器关闭的一部分）或者一个 shell 错误（例如命令未找到）终止，则恢复将会中止并且服务器将不会启动。

这个参数只能在lightdb.conf文件中或通过服务器命令行进行设置。

archive\_cleanup\_command (string)

这个可选参数指定了一个 shell 命令，它将在每一个重启点被执行。archive\_cleanup\_command的目的是提供一种清除不再被后备服务器需要的旧的已归档 WAL 文件的机制。任何%r会被替换为包含最后一个可用重启点的文件的名称。那是使一次恢复变成可重启的所必须被保留的最早的文件，并且因此比%r更早的所有文件可以被安全地移除。这个信息可以被用来把归档截断为支持从当前恢复重启所需的最小值。对于单一后备配置，lt\_archivecleanup模块常常被用在archive\_cleanup\_command中，例如：

```
archive_cleanup_command = 'lt_archivecleanup /mnt/server/archivedir %r'
```

但是注意，如果多个后备服务器正在从同一个归档目录中恢复，你将需要保证只有当任意服务器都不再需要 WAL 文件时才会删除它们。archive\_cleanup\_command通常被用于一种温后备配置（见Section 24.2 中。要在该命令中嵌入一个真正的%字符，需要写成%%。

如果该命令返回一个非零退出状态，则将会写出一个警告日志消息。一个例外是如果该命令被一个信号或者一个 shell 错误（例如命令未找到）终止，则会抛出一个致命错误。

这个参数只能在 lightdb.conf 文件中设置或通过服务器命令行的方式。

recovery\_end\_command (string)

这个参数指定了一个将只在恢复末尾被执行一次的 shell 命令。这个参数是可选的。recovery\_end\_command的目的是为复制或恢复之后的清除提供一种机制。与archive\_cleanup\_command中相似，任何%r会被替换为包含最后一个可用重启点的文件的名称。

如果该命令返回一个非零退出状态，则一个警告日志消息将被写出并且不管怎样该数据库将继续启动。一个例外是如果该命令被一个信号或者 shell 错误（例如命令未找到）中止，该数据库将不会继续启动。

这个参数只能在 lightdb.conf 文件中设置或通过服务器命令行的方式。

## 17.5.5. 恢复目标

默认情况下，恢复将会一直恢复到 WAL 日志的末尾。下面的参数可以被用来指定一个更早的停止点。

在`recovery_target`、`recovery_target_lsn`、`recovery_target_name`、`recovery_target_time`和`recovery_target_xid`最多只能使用一个，如果在配置文件中使用了多个，将会产生一个错误。这个参数只能在服务器启动时设置。

`recovery_target = 'immediate'`

这个参数指定恢复应该在达到一个一致状态后尽快结束，即尽早结束。在从一个在线备份中恢复时，这意味着备份结束的那个点。

在技术上，这是一个字符串参数，但是`'immediate'`是目前唯一允许的值。

`recovery_target_name (string)`

这个参数指定 (`pg_create_restore_point()`所创建) 的已命名的恢复点，恢复将进入该恢复点。

`recovery_target_time (timestamp)`

此参数指定恢复将执行的时间戳。精确的停止点还受到`recovery_target_inclusive`得影响。

此参数的值是一个被`timestamp with time zone`数据类型接受的相同格式的时间戳，只不过你不能使用时区缩写（除非`timezone_abbreviations`变量在配置文件中已提前设置）。首选样式是使用 UTC 的数字偏移量，或者你可以写一个完整时区名称，例如 `Europe/Helsinki` 而不是 `EEST`。

`recovery_target_xid (string)`

这个参数指定恢复将进入的事务 ID。记住虽然事务 ID 是在事务开始时顺序分配的，但是事务可能以不同的数字顺序完成。那些在指定事务之前（也可以包括该事务）提交的事务将被恢复。精确的停止点也受到`recovery_target_inclusive`的影响。

`recovery_target_lsn (pg_lsn)`

此参数指定恢复将继续进行的预写日志位置的LSN。精确的停靠点也受到`recovery_target_inclusive`的影响。使用系统数据类型`pg_lsn`解析此参数。

下列选项进一步指定恢复目标，并且影响到达目标时会发生什么：

`recovery_target_inclusive (boolean)`

指定我们是否仅在指定的恢复目标之后停止 (`on`)，或者仅在恢复目标之前停止 (`off`)。适用于`recovery_target_lsn`、`recovery_target_time`或者`recovery_target_xid`被指定的情况。这个设置分别控制事务是否有准确的目标WAL位置 (LSN)、提交时间或事务ID将被包括在该恢复中。默认值为`on`。

`recovery_target_timeline (string)`

指定恢复到一个特定的时间线中。该值可以是数字时间线 ID 或特殊值。值`current`沿着与执行基本备份时相同的时间线恢复。值`latest`将恢复到归档中能找到的最新的时间线，这在一个备用服务器中 useful。`latest`是默认的。

你通常只需要在复杂的重恢复情况下设置这个参数，在这种情况下你需要返回到一个状态，该状态本身是在一次时间点恢复之后到达的。相关讨论见Section 23.3.5

`recovery_target_action (enum)`

指定在达到恢复目标时服务器应该立刻采取的动作。默认动作是`pause`，这表示恢复将会被暂停。`promote`表示恢复处理将会结束并且服务器将开始接受连接。最后，`shutdown`将在达到恢复目标之后停止服务器。

使用`pause`设置的目的是：如果这个恢复目标就是恢复最想要的位置，就允许对数据库执行查询。暂停的状态可以使用`pg_wal_replay_resume()`（见Table 9.88 继续，这会让

恢复终结。如果这个恢复目标不是想要的停止点，那么关闭服务器，将恢复目标设置改为一个稍后的目标并且重启以继续恢复。

要让实例在想要的重放点那里准备好，`shutdown`设置可以派上用场。该实例将仍能重放更多 WAL 记录（并且事实上将不得不重放从下一次它被启动后最后一个检查点以来的 WAL 记录）。

注意由于在`recovery_target_action`被设置为`shutdown`时，`recovery.signal`将不会被移除，任何后续的启动都将会以立刻关闭为终结，除非该配置被改变或者`recovery.signal`文件被手工移除。

如果没有设置恢复目标，这个设置没有效果。如果没有启用`hot_standby`，`pause`设置的动作将与`shutdown`一样。如果在升级期间达到恢复目标，`pause`的设置将与`promote`的行为相同。

在任何情况下，如果已配置了恢复目标，但归档恢复在达到目标之前结束，则服务器将关闭，并出现致命错误。

## 17.6. 复制

这些设置控制内建流复制特性（见Section 24.2.5 的行为。服务器可以是主控服务器或后备服务器。主控机能发送数据，而后备机总是被复制数据的接收者。当使用级联复制（见Section 24.2.7 时，后备服务器也可以是发送者，同时也是接收者。这些参数主要用于发送服务器和后备服务器，尽管某些只在主服务器上有意义。如果有必要，设置可以在集群中变化而不出问题。

### 17.6.1. 发送服务器

这些参数可以在任何发送复制数据给一个或多个后备服务器的服务器上设置。主控机总是一个发送服务器，因此这些参数总是要在主控机上设置。这些参数的角色和含义不会在一个后备机变成主控机后改变。

`max_wal_senders` (integer)

指定来自后备服务器或流式基础备份客户端的并发连接的最大数量（即同时运行 WAL 发送进程的最大数）。默认值是10。值0意味着禁用复制。突然的流客户端断开连接可能留下一个孤立连接槽（知道达到超时），因此这个参数应该设置得略高于最大客户端连接数，这样断开连接的客户端可以立刻重新连接。这个参数只能在服务器启动时被设置。此外，`wal_level`必须设置为`replica`或更高级别以允许来自后备服务器的连接。

当运行备用服务器时，你必须将此参数设置为与主服务器上相同的或更高的值。否则，备用服务器中将不允许查询。

`max_replication_slots` (integer)

指定服务器可以支持的复制槽（见Section 24.2.6 最大数量。默认值为10。这个参数只能在服务器启动时设置。将它设置为一个比当前已有复制槽要少的值会阻碍服务器启动。此外，要允许使用复制槽，`wal_level`必须被设置为`replica`或更高。

在订阅端，指定多少复制源头（参见 Chapter 46 可以被同时跟踪，有效的限制了服务器上可以建立多少逻辑复制订阅。设置它到一个低与当前跟踪复制源（体现在`pg_replication_origin_status`，不是`pg_replication_origin`）的值将阻止服务器启动。

`wal_keep_size` (integer)

指定在后备服务器需要为流复制获取日志段文件的情况下，`lt_wal`目录下所能保留的过去日志文件段的最小尺寸。如果一个连接到发送服务器的后备服务器落后了超过`wal_keep_size`兆字节，发送服务器可以移除一个后备机仍然需要的 WAL 段，在这种

情况下复制连接将被中断。最终结果是下行连接也将最终失败（不过，如果在使用 WAL 归档，后备服务器可以通过从归档获取段来恢复）。

只设置`lt_wal`中保留的文件段的最小尺寸；系统可能需要为 WAL 归档或从一个检查点恢复保留更多段。如果`wal_keep_size`为零（默认值），更多的空间来存放WAL归档或从一个检查点恢复。如果`wal_keep_segments`是零（缺省），系统不会为后备目的保留任何多余的段，因此后备服务器可用的旧 WAL 段的数量是一个上个检查点位置和 WAL 归档状态的函数。如果此值被指定时没有单位，则视为兆字节。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

`max_slot_wal_keep_size` (integer)

指定`replication_slots`所允许的在检查点时保留在`lt_wal`目录中的WAL文件的最大尺寸。如果`max_slot_wal_keep_size`为`-1`（默认值），复制槽可能会保留无限数量的WAL文件。否则，如果复制槽的`restart_lsn`比当前LSN滞后超过给定的大小，由于删除了所需的WAL文件，使用插槽的备用服务器可能无法继续复制。你可以在`pg_replication_slots`中查看复制槽有效的WAL。

`wal_sender_timeout` (integer)

中断那些停止活动超过这个时间量的复制连接。这对发送服务器检测一个备用服务器崩溃或网络中断有用。如果指定值时没有单位，则以毫秒为单位。默认值是60秒。值0将禁用超时机制。

对于跨越多个地理位置分布的集群，每个位置使用不同的值可以为集群管理带来更大的灵活性。对于具有低延迟网络连接的备用服务器，较小的值有助于更快的故障检测；对于位于远端位置的具有高延迟网络连接的备用服务器，较大的值有助于更好地判断备用服务器的健康状况。

`track_commit_timestamp` (boolean)

记录事务的提交时间。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。默认值是`off`。

## 17.6.2. 主服务器

这些参数可以在发送复制数据给一个或多个后备服务器的主控/主要服务器上设置。注意除了这些参数之外，在主控服务器上必须设置合适的`wal_level`，并且也启用可选的WAL归档（见Section 17.5.3。这些参数值与后备服务器无关，尽管你可能希望为了准备好一个后备机转变成主控机来设置这些参数。

`synchronous_standby_names` (string)

如Section 24.2.所述，这个参数指定一个支持同步复制的后备服务器的列表。可能会有一个或者多个活动的同步后备服务器，在这些后备服务器确认收到它们的数据之后，等待提交的事务将被允许继续下去。同步后备服务器是那些名字出现在这个列表前面，并且当前已连接并且正在实时流传输数据（如`pg_stat_replication`视图中`streaming`的状态所示）的服务器。指定多于一台同步后备可以得到非常高的可用性并且能防止数据丢失。

用于这一目的的后备服务器的名称是其`application_name`设置，它在后备服务器的连接信息中设置。在物理复制后备的情况下，这应该被设置在`primary_conninfo`设置中，如果设置了，默认是`cluster_name`的设置，否则为`walreceiver`。对于逻辑复制，可以在订阅的连接信息中设置。对于其他复制流消费者，请参考其文档。

这个参数使用下面的语法之一来指定一个后备服务器列表：

```
[FIRST] num_sync ( standby_name [, ...] )
ANY num_sync ( standby_name [, ...] )
standby_name [, ...]
```

其中num\_sync是事务需要等待其回复的同步后备服务器的数量，standby\_name是一个后备服务器的名称。FIRST以及ANY指定从所列服务器中选取同步后备的方法。

关键词FIRST加上num\_sync指定一种基于优先的同步复制，并且会让事务提交等待，直到它们的WAL记录被复制到基于优先级选择的num\_sync台同步后备上为止。例如，设置FIRST 3 (s1, s2, s3, s4)将导致每次提交都等待来自三台较高优先级的后备机的答复，这三台后备机将从后备服务器s1、s2、s3以及s4中选出。在该列表中出现较早的后备服务器将被给予较高的优先级，并且将被考虑为同步后备。列表中出现的其他后备服务器表示潜在的同步后备。如果当前的任何同步后备因为某种原因断开连接，它将立刻被下一个最高优先级的后备服务器替代。关键词FIRST是可选的。

关键词ANY加上num\_sync指定一种基于规定数量的同步复制，并且会让事务提交等待，直到它们的WAL记录被复制到所列后备服务器中的至少num\_sync台上为止。例如，设置ANY 3 (s1, s2, s3, s4)将导致每次提交会在收到s1、s2、s3以及s4中任意三台后备服务器的回答后立刻继续下去。

FIRST和ANY是大小写不敏感的。如果这些关键词被用作后备服务器的名字，其standby\_name必须被放在双引号内。

LightDB版本 9.6 之前使用过第三种语法，目前也仍然支持。它和FIRST和num\_sync等于1的第一种语法相同。例如，FIRST 1 (s1, s2)和s1, s2具有相同的含义：s1或者s2会被选中作为同步后备服务器。

特殊项\*匹配任意后备名称。

没有机制强制后备服务器名称的唯一性。在出现重复的情况下，匹配的后备之一将被认为是较高优先级，不过无法弄清到底是哪一个。

### Note

每一个standby\_name都应该具有合法 SQL 标识符的形式，除非它是\*。如果必要你可以使用双引号。但是注意在比较standby\_name和后备机应用程序名称时是大小写不敏感的（不管有没有双引号）。

如果这里没有指定同步后备机名称，那么同步复制不能被启用并且事务提交将不会等待复制。这是默认的配置。即便当同步复制被启用时，个体事务也可以被配置为不等待复制，做法是将synchronous\_commit参数设置为local或off。

这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

vacuum\_defer\_cleanup\_age (integer)

指定VACUUM和HOT更新在清除死亡行版本之前，应该推迟多久（以事务数量计）。默认值是零个事务，表示死亡行版本将被尽可能快地清除，即当它们不再对任何打开的事务可见时尽快清除。在一个支持热后备服务器的主服务器上，你可能希望把这个参数设置为一个非零值，如Section 24.5中所述。这允许后备机上的查询有更多时间来完成而不会由于先前的行清除产生冲突。但是，由于该值是用在主服务器上发生的写事务的数目衡量的，很难预测对后备机查询可用的附加时间到底是多少。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

你也可以考虑设置后备服务器上的hot\_standby\_feedback作为使用这个参数的一种替代方案。

这无法阻止已经达到old\_snapshot\_threshold所指定年龄的死亡行被清除。

## 17.6.3. 后备服务器

这些设置控制一个standby server接收复制数据的行为。它们在主服务器的值是无关系的。



`primary_conninfo` (string)

指定备用服务器用来连接发送服务器的连接字符串。这个字符串的格式在Section 32.1.中描述。如果在这个字符串中有任何选项未被指定，那么将检查相应的环境变量（见Section 32.14）。如果环境变量也没有被设置，则使用默认值。

连接字符串应当指定发送服务器的主机名（或地址），以及端口号（如果它和备用服务器的默认端口不同）。还要指定对应于发送服务器上合适权限角色的用户名（见Section 24.2.5.1）。如果发送服务器要求口令认证，还需要提供一个口令。它可以在`primary_conninfo`字符串中提供，或者在后备服务器（使用`replication`作为数据库名）的一个单独`~/.pgpass`文件中提供。不要在`primary_conninfo`字符串中指定一个数据库名。

这个参数只能在 `lightdb.conf` 文件或服务器命令行上设置。如果在 WAL 接收器进程运行时更改此参数，则该进程将发出关机信号，并预期使用新设置重新启动（除非`primary_conninfo`为空字符串）。如果服务器没有处于备用模式，则此设置无效。

`primary_slot_name` (string)

有选择地指定通过流复制连接到发送服务器时使用一个现有的复制槽来控制上游节点上的资源移除（见Section 24.2.6）。这个参数只能在 `lightdb.conf` 文件或服务器命令行上设置。如果在 WAL 接收器进程运行时更改此参数，则该进程将发出关机信号，并预期使用新设置重新启动（除非`primary_conninfo`为空字符串）。如果没有设置`primary_conninfo`或服务器没有处于备用模式，则此设置无效。

`promote_trigger_file` (string)

指定一个触发器文件，该文件的存在会结束后备机中的恢复。即使这个值没有被设置，你也能够使用`lt_ctl promote`或调用`pg_promote()`来提升后备机。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

`hot_standby` (boolean)

指定在恢复期间，你是否能够连接并运行查询，如Section 24.5中所述。默认值是`on`。这个参数只能在服务器启动时设置。它只在归档恢复期间或后备机模式下才有效。

`max_standby_archive_delay` (integer)

当热后备机处于活动状态时，这个参数决定取消那些与即将应用的 WAL 项冲突的后备机查询之前，后备服务器应该等待多久，如Section 24.5.2中所述。当 WAL 数据被从 WAL 归档（并且因此不是当前的 WAL）时，`max_standby_archive_delay`可以应用。如果指定值时没有单位，则以毫秒为单位。默认值是 30 秒。值 `-1` 允许后备机一直等到冲突查询结束。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

注意，`max_standby_archive_delay`与取消之前一个查询能够运行的最长时间不同；它表示应用任何一个 WAL 段数据能够被允许的最长总时间。因此，如果一个查询早于 WAL 段导致了显著的延迟，后续冲突查询将只有更少的时间。

`max_standby_streaming_delay` (integer)

当热后备机处于活动状态时，这个参数决定取消那些与即将应用的 WAL 项冲突的后备机查询之前，后备服务器应该等待多久，如Section 24.5.2中所述。当 WAL 数据正在通过流复制被接收时，`max_standby_streaming_delay`可以应用。如果指定值时没有单位，则以毫秒为单位。默认值是 30 秒。值 `-1` 允许后备机一直等到冲突查询结束。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

注意，`max_standby_streaming_delay`与取消之前一个查询能够运行的最长时间不同；它表示在从主服务器接收到 WAL 数据并立刻应用它能够被允许的最长总时间。因此，如果一个查询导致了显著的延迟，后续冲突查询将只有更少的时间，直到后备服务器再次赶上进度。

wal\_receiver\_create\_temp\_slot (boolean)

指定在未配置永久复制槽时，WAL 接收器进程是否应在远程实例上创建临时复制槽（使用 primary\_slot\_name）。默认为“关闭”。此参数只能在lightdb.conf文件或服务器命令行中设置。如果在 WAL 接收器进程运行时更改此参数，则该进程将发出关机的信号，并预期使用新设置重新启动。

wal\_receiver\_status\_interval (integer)

指定在后备机上的 WAL 接收器进程向主服务器或上游后备机发送有关复制进度的信息的最小频度，它可以使用pg\_stat\_replication视图看到。后备机将报告它已经写入的上一个预写式日志位置、它已经刷到磁盘的上一个位置以及它已经应用的最后一个位置。这个参数的值是报告之间的最大时间量。每次写入或刷出位置改变时会发送状态更新，或者按这个参数的指定的频度发送，如果设置为非零值。在其他情况下发送更新时忽略此参数；例如，当现有的WAL处理完成或当synchronous\_commit设置为remote\_apply时。因此，应用位置可能比真实位置略微滞后。如果指定值时没有单位，则以秒为单位。默认值是 10 秒。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

hot\_standby\_feedback (boolean)

指定一个热后备机是否将会向主服务器或上游后备机发送有关于后备机上当前正被执行的查询的反馈。这个参数可以被用来排除由于记录清除导致的查询取消，但是可能导致在主服务器上用于某些负载的数据库膨胀。反馈消息的发送频度不会高于每个wal\_receiver\_status\_interval周期发送一次。默认值是off。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

如果使用级联复制，反馈将被向上游传递直到它最后到达主服务器。后备机在接收到反馈之后除了传递给上游不会做任何其他操作。

这个设置不会覆盖主服务器上的old\_snapshot\_threshold的行为，后备服务器上一个超过了主服务器年龄阈值的快照可能会变得不可用，导致后备服务器上事务的取消。这是因为old\_snapshot\_threshold是为了对死亡行能够存在的时间给出一个绝对限制，不然就会因为一个后备服务器的配置而被违背。

wal\_receiver\_timeout (integer)

中止处于非活动状态超过这个时间量的复制链接。这对于正在接收的后备服务器检测主服务器崩溃或网络断开有用。如果指定值时没有单位，则以毫秒为单位。默认值是 60 秒。这个参数只能在lightdb.conf文件中或者服务器命令行中设置。

wal\_retrieve\_retry\_interval (integer)

指定当从任何来源（流复制、本地lt\_wal或者 WAL 归档）都得不到 WAL 数据时，后备服务器应该等待多久才去重新尝试获取 WAL 数据。如果指定值时没有单位，则以毫秒为单位。默认值是 5 秒。这个参数只能在lightdb.conf文件或者服务器命令行中设置。

这个参数对恢复中的节点需要为新 WAL 数据可用等待多少时间的配置有用。例如，在归档恢复中，通过减小这个参数的值可以让恢复更积极地检测新的 WAL 日志文件。在一个 WAL 活动较低的系统上，增加这个参数的值可以减少访问 WAL 归档所必需的请求数量，这对于例如云环境是有用的，在其中对于基础设施的访问时间也是被考虑的。

recovery\_min\_apply\_delay (integer)

默认情况下，后备服务器会尽快恢复来自于发送服务器的 WAL 记录。有一份数据的延迟拷贝是有用的，它能提供机会纠正数据丢失错误。这个参数允许你将恢复延迟一段指定的时间量。例如，如果你设置这个参数为5min，对于一个事务提交，只有当后备机上的系统时钟超过主服务器报告的提交时间至少 5分钟时，后备机才会重放该事务。如果指定值时没有单位，则以毫秒为单位。默认为0，不增加延迟。

有可能服务器之间的复制延迟会超过这个参数的值，在这种情况下则不会增加延迟。注意延迟是根据主服务器上写 WAL 的时间戳以及后备机上的当前时间来计算。由于网络延迟或者级联复制配置导致的传输延迟可能会显著地减少实际等待时间。如果主服务器

和后备机上的系统时钟不同步，这会导致恢复比预期的更早应用记录。但这不是一个主要问题，因为这个参数有用的设置比服务器之间的典型事件偏差要大得多。

只有在事务提交的 WAL 记录上才会发生延迟。其他记录还是会被尽可能快地重放，这不会成为问题，因为 MVCC 可见性规则确保了在对应的提交记录被应用之前它们的效果不会被看到。

一旦恢复中的数据库已经达到一致状态，延迟就会产生，直到后备机被提升或者触发。在那之后，后备机将会结束恢复并且不再等待。

这个参数的目的是和流复制部署一起使用，但是，如果指定了该参数，除了崩溃恢复之外所有的情况下都会遵守它。使用这个特性也会让 `hot_standby_feedback` 被延迟，这可能导致主服务器的膨胀，两者一起使用时要小心。

### Warning

当 `synchronous_commit` 被设置为 `remote_apply` 时，同步复制会受到这个设置的影响，每一个 COMMIT 都需要等待被应用。

这个参数只能在 `lightdb.conf` 文件中或在服务器命令行中设置。

## 17.6.4. 订阅者

这些设置控制逻辑复制订阅者的行为。它们在发布者上的值与此无关。

注意，配置参数 `wal_receiver_timeout`、`wal_receiver_status_interval` 以及 `wal_retrieve_retry_interval` 也影响逻辑复制工作者。

`max_logical_replication_workers` (int)

指定逻辑复制工作者的最大数目。这同时包括应用工作者和表同步工作者。

逻辑复制工作者是从 `max_worker_processes` 定义的池中取出的。

默认值是4。这个参数只能在服务器启动时被设置。

`max_sync_workers_per_subscription` (integer)

每个订阅的同步工作者的最大数目。这个参数控制订阅初始化期间或者新表增加时的初始数据拷贝的并行度。

当前，每个表只能有一个同步工作者。

同步工作者是从 `max_logical_replication_workers` 定义的池中取出的。

默认值为2。这个参数只能在 `lightdb.conf` 文件中或在服务器命令行上设置。

## 17.7. 查询规划

### 17.7.1. 规划器方法配制

这些配置参数提供了影响查询优化器选择查询规划的原始方法。如果优化器为特定的查询选择的缺省规划并不是最优，那么我们就可以通过使用这些配置参数强制优化器选择一个更好的规划来 temporary 解决这个问题。不过，永久地关闭这些设置几乎从不是个好主意。更好的改善优化器选择规划的方法包括调节 Section 18.6.2、更频繁运行 ANALYZE、增大配置参数 `default_statistics_target` 的值、使用 `ALTER TABLE SET STATISTICS` 为某个字段增加收集的统计信息。这些配置参数影响查询优化器选择查询计划的暴力方法。如果优化器为一个特定查询选择的默认计划不是最优的，一种临时解决方案

是使用这些配置参数之一来强制优化器选择一个不同的计划。提高优化器选择的计划质量的更好的方式包括调整规划器的代价常数（见Section 17.7.2、手工运行ANALYZE、增加default\_statistics\_target配置参数的值以及使用ALTER TABLE SET STATISTICS增加为特定列收集的统计信息量。

enable\_bitmapscan (boolean)

允许或禁止查询规划器使用位图扫描计划类型。默认值是on。

enable\_gathermerge (boolean)

启用或者禁用查询规划器对收集归并计划类型的使用。默认值是on。

enable\_hashagg (boolean)

允许或禁用查询规划器使用哈希聚集计划类型。默认值是on。

lightdb\_enable\_sorted\_group (boolean)

启用或禁用查询规划器对排序聚合或分组（排序）计划类型的使用。默认值是on。

enable\_hashjoin (boolean)

允许或禁止查询规划器使用哈希连接计划类型。默认值是on。

enable\_incremental\_sort (boolean)

启用或禁用查询规划器对增量排序步骤的使用。默认为on。

enable\_indexscan (boolean)

允许或禁止查询规划器使用索引扫描计划类型。默认值是on。

enable\_indexonlyscan (boolean)

允许或禁止查询规划器使用只用索引扫描计划类型（见Section 11.9）。默认值是on。

enable\_material (boolean)

允许或者禁止查询规划器使用物化。它不可能完全禁用物化，但是关闭这个变量将阻止规划器插入物化节点，除非为了保证正确性。默认值是on。

enable\_mergejoin (boolean)

允许或禁止查询规划器使用归并连接计划类型。默认值是on。

enable\_nestloop (boolean)

允许或禁止查询规划器使用嵌套循环连接计划。它不可能完全禁止嵌套循环连接，但是关闭这个变量将使得规划器尽可能优先使用其他方法。默认值是on。

enable\_parallel\_append (boolean)

允许或禁止查询规划器使用并行追加计划类型。默认值是on。

enable\_parallel\_hash (boolean)

允许或禁止查询规划器对并行哈希使用哈希连接计划类型。如果哈希连接计划也没有启用，这个参数没有效果。默认值是on。

enable\_partition\_pruning (boolean)

允许或者禁止查询规划器从查询计划中消除一个分区表的分区。这也控制着规划器产生允许执行器在查询执行期间移除（忽略）分区的查询计划的能力。默认值是on。详情请参考Section 5.11.4

`enable_partitionwise_join` (boolean)

允许或者禁止查询规划器使用面向分区的连接，这使得分区表之间的连接以连接匹配的分区的方式来执行。面向分区的连接当前只适用于连接条件包括所有分区键的情况，连接条件必须是相同的数据类型并且子分区集合要1对1匹配。由于面向分区的连接规划在规划期间会使用可观的CPU时间和内存，所以默认值为off。

`enable_partitionwise_aggregate` (boolean)

允许或者禁止查询规划器使用面向分区的分组或聚集，这使得在分区表上的分组或聚集可以在每个分区上分别执行。如果GROUP BY子句不包括分区键，只有部分聚集能够以基于每个分区的方式执行，并且finalization必须最后执行。由于面向分区的分组或聚集在规划期间会使用可观的CPU时间和内存，所以默认值为off。

`enable_seqscan` (boolean)

允许或禁止查询规划器使用顺序扫描计划类型。它不可能完全禁止顺序扫描，但是关闭这个变量将使得规划器尽可能优先使用其他方法。默认值是on。

`enable_sort` (boolean)

允许或禁止查询规划器使用显式排序步骤。它不可能完全禁止显式排序，但是关闭这个变量将使得规划器尽可能优先使用其他方法。默认值是on。

`enable_tidscan` (boolean)

允许或禁止查询规划器使用TID扫描计划类型。默认值是on。

## 17.7.2. 规划器代价常量

这一节中描述的代价变量可以按照任意尺度衡量。我们只关心它们的相对值，将它们以相同的因子缩放不会影响规划器的选择。默认情况下，这些代价变量是基于顺序页面获取的代价的，即`seq_page_cost`被设置为1.0并且其他代价变量都参考它来设置。不过你可以使用你喜欢的不同尺度，例如在一个特定机器上的真实执行时间。

### Note

不幸的是，没有一种良定义的方法来决定代价变量的理想值。它们最好被作为一个特定安装将接收到的查询的平均值来对待。这意味着基于少量的实验来改变它们是有风险的。

`seq_page_cost` (floating point)

设置规划器计算一次顺序磁盘页面抓取的开销。默认值是1.0。通过设置同名的表空间参数，这个值可以重写为一个特定的表空间。参阅ALTER TABLESPACE。设置规划器对一系列顺序磁盘页面获取中的一次代价估计。默认值是1.0。通过把表和索引放在一个特殊的表空间（要设置该表空间的同名参数）中可以覆盖这个值（见ALTER TABLESPACE）。

`random_page_cost` (floating point)

设置规划器对一次非顺序获取磁盘页面的代价估计。默认值是4.0。通过把表和索引放在一个特殊的表空间（要设置该表空间的同名参数）中可以覆盖这个值（见ALTER TABLESPACE）。

减少这个值（相对于`seq_page_cost`）将导致系统更倾向于索引扫描；提高它将让索引扫描看起来相对更昂贵。你可以一起提高或降低两个值来改变磁盘 I/O 代价相对于 CPU 代价的重要性，后者由下列参数描述。

对磁盘存储的随机访问通常比顺序访问要贵不止四倍。但是，由于对磁盘的大部分随机访问（例如被索引的读取）都被假定在高速缓冲中进行，所以使用了一个较低的默认值（4.0）。默认值可以被想成把随机访问建模为比顺序访问慢 40 倍，而期望 90% 的随机读取会被缓存。

如果你相信 90% 的缓冲率对你的负载是一个不正确的假设，你可以增加 `random_page_cost` 来更好的反映随机存储读取的真正代价。相应地，如果你的数据可以完全放在高速缓存中（例如当数据库小于服务器总内存时），降低 `random_page_cost` 可能是合适的。为具有很低的随机读取代价的存储（例如固态驱动器）采用较低的 `random_page_cost` 值可能更好，例如 1.1。

### Tip

虽然允许你将 `random_page_cost` 设置的比 `seq_page_cost` 小，但是物理上的实际情况并不受此影响。然而当所有数据库都位于内存中时，两者设置为相等是非常合理的，因为在此情况下，乱序抓取并不比顺序抓取开销更大。同样，在缓冲率很高的数据库上，你应当相对于 CPU 开销同时降低这两个值，因为获取内存中的页比通常情况下的开销小许多。尽管系统可以是你把 `random_page_cost` 设置得小于 `seq_page_cost`，但是实际上没有意义。不过，如果数据库被整个缓存在 RAM 中，将它们设置为相等是有意义的，因为在那种情况中不按顺序访问页面是没有惩罚值的。同样，在一个高度缓存化的数据库中，你应该相对于 CPU 参数降低这两个值，因为获取一个已经在 RAM 中的页面的代价要远小于通常情况下的代价。

`cpu_tuple_cost` (floating point)

设置规划器对一次查询中处理每一行的代价估计。默认值是 0.01。

`cpu_index_tuple_cost` (floating point)

设置规划器对一次索引扫描中处理每一个索引项的代价估计。默认值是 0.005。

`cpu_operator_cost` (floating point)

设置规划器对于一次查询中处理每个操作符或函数的代价估计。默认值是 0.0025。

`parallel_setup_cost` (floating point)

设置规划器对启动并行工作者进程的代价估计。默认是 1000。

`parallel_tuple_cost` (floating point)

设置规划器对于从一个并行工作者进程传递一个元组给另一个进程的代价估计。默认是 0.1。

`min_parallel_table_scan_size` (integer)

为必须扫描的表数据量设置一个最小值，扫描的表数据量超过这一个值才会考虑使用并行扫描。对于并行顺序扫描，被扫描的表数据量总是等于表的尺寸，但是在使用索引时，被扫描的表数据量通常会更小。如果指定值时没有单位，则以块为单位，即BLCKSZ字节，通常为8kB。默认值是8兆字节（8MB）。

`min_parallel_index_scan_size` (integer)

为必须扫描的索引数据量设置一个最小值，扫描的索引数据量超过这一个值时才会考虑使用并行扫描。注意并行索引扫描通常并不会触及整个索引，它是规划器认为该扫描会实际用到的相关页面的数量。这个参数还用于决定特定的索引是否参与并行vacuum。参见VACUUM。如果指定值时没有单位，则以块为单位，即BLCKSZ字节，通常为8kB。默认值是512千字节（512kB）。

`effective_cache_size` (integer)

设置规划器对一个单一查询可用的有效磁盘缓冲区尺寸的假设。这个参数会被考虑在使用一个索引的代价估计中，更高的数值会使得索引扫描更可能被使用，更低的数值会使得顺序扫描更可能被使用。在设置这个参数时，你还应该考虑LightDB的共享缓冲区以及将被用于LightDB数据文件的内核磁盘缓冲区，尽管有些数据可能在两个地方都存在。另外，还要考虑预计在不同表上的并发查询数目，因为它们必须共享可用的空间。这个参数对LightDB分配的共享内存尺寸没有影响，它也不会保留内核磁盘缓冲，它只用于估计的目的。系统也不会假设在查询之间数据会保留在磁盘缓冲中。如果指定值没有单位，则以块为单位，即BLCKSZ字节，通常为8kB。默认值是 4吉字节（4GB）。（如果BLCKSZ不是8kB，默认值会按比例缩放它。）

`jit_above_cost` (floating point)

设置激活JIT编译的查询代价，如果查询代价超过这个值就会激活JIT编译（如果启用了JIT，见Chapter 29。执行JIT会消耗一些规划时间，但是能够加速查询执行。将这个值设置为-1会禁用JIT编译。默认值是100000。

`jit_inline_above_cost` (floating point)

设置JIT编译尝试内联函数和操作符的查询代价阈值，如果查询代价超过这个值，JIT编译就会尝试内联。内联会增加规划时间，但是可以提高执行速度。将这个参数设置成小于`jit_above_cost`是没有意义的。将这个参数设置为-1会禁用内联。默认值是500000。

`jit_optimize_above_cost` (floating point)

设置JIT编译应用优化的查询代价阈值，如果查询代价超过这个值，JIT编译就会应用开销较大的优化。这类优化会增加规划时间，但是更能够改进执行速度。将这个参数设置成小于`jit_above_cost`是没有意义的，并且将它设置成大于`jit_inline_above_cost`也未必有益。将这个参数设置为-1会禁用开销较大的优化。默认值是500000。

### 17.7.3. 遗传查询优化

GEQO是一个使用探索式搜索来执行查询规划的算法。它可以降低负载查询的规划时间。同时，GEQO的检索是随机的，因此它的规划可能会不可确定。更多信息参阅Chapter 50。遗传查询规划器（GEQO）是一种使用启发式搜索来进行查询规划的算法。它可以降低对于复杂查询（连接很多表的查询）的规划时间，但是代价是它产生的计划有时候要差于使用穷举搜索算法找到的计划。详见Chapter 56

`geqo` (boolean)

允许或禁止遗传查询优化。默认是启用。在生产环境中通常最好不要关闭它。`geqo_threshold`变量提供了对 GEQO 更细粒度的空值。

`geqo_threshold` (integer)

只有当涉及的FROM项数量至少有这么多个的时候，才使用遗传查询优化（注意一个FULL OUTER JOIN只被计为一个FROM项）。默认值是 12。对于更简单的查询，通常会使用普通的穷举搜索规划器，但是对于有很多表的查询穷举搜索会花很长时间，通常比执行一个次优的计划带来的惩罚值还要长。因此，在查询尺寸上的一个阈值是管理 GEQO 使用的一种方便的方法。

`geqo_effort` (integer)

控制 GEQO 里规划时间和查询规划的有效性之间的平衡。这个变量必须是一个范围从 1 到 10 的整数。缺省值是 5。大的数值增加花在进行查询规划上面的时间，但是也很可能会提高选中更有效的查询规划的几率。控制 GEQO 中规划时间和查询计划质量之间的折中。这个变量必须是位于 1 到 10 之间的一个整数。默认值是 5。更大的值会增加花在查询规划上的时间，但是同时也增加了选择一个高效查询计划的可能性。

`geqo_effort`实际并不直接做任何事情；它只是被用来计算其他影响 GEQO 行为的变量（如下所述）的默认值。如果你愿意，你可以手工设置其他参数。

`geqo_pool_size` (integer)

控制 GEQO 使用的池尺寸，它就是遗传种群中的个体数目。它必须至少为 2，且有用的值通常在 100 到 1000 之间。如果它被设置为零（默认设置）则会基于`geqo_effort`和查询中表的数量选择一个合适的值。

`geqo_generations` (integer)

控制 GEQO 使用的子代数。子代的意思是算法的迭代次数。它必须至少是 1，有用的值范围和池大小相同。如果设置为零（缺省），那么将基于 `geqo_pool_size` 选取合适的值。控制 GEQO 使用的代数，也是算法的迭代次数。它必须至少为 1，并且有用值的范围和池尺寸相同。如果它被设置为零（默认设置）则会基于`geqo_pool_size`选择一个合适的值。

`geqo_selection_bias` (floating point)

控制 GEQO 使用的选择偏好。选择偏好是种群中的选择压力。值可以是 1.5 到 2.0 之间，后者是默认值。

`geqo_seed` (floating point)

控制 GEQO 使用的随机数生成器的初始值，随机数生成器用于在连接顺序搜索空间中选择随机路径。该值可以从 0（默认值）到 1。变化该值会改变被探索的连接路径集合，并且可能导致找到一个更好或更差的路径。

## 17.7.4. 其他规划器选项

`default_statistics_target` (integer)

为没有通过ALTER TABLE SET STATISTICS设置列相关目标的表列设置默认统计目标。更大的值增加了需要做ANALYZE的时间，但是可能会改善规划器的估计质量。默认值是 100。有关LightDB查询规划器使用的统计信息的更多内容，请参考Section 14.2

`lightdb_enable_indexautoanalyze` (integer)

启用或禁用在创建索引后立即自动收集统计信息。默认启用。

`constraint_exclusion` (enum)

控制查询规划器对表约束的使用，以优化查询。`constraint_exclusion`的允许值是on（对所有表检查约束）、off（从不检查约束）和partition（只对继承的子表和UNION ALL子查询检查约束）。partition是默认设置。它通常与传统的继承树一起使用来提高性能。

当对一个特定表允许这个参数，规划器比较查询条件和表的CHECK约束，并且忽略那些条件违反约束的表扫描。例如：

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

在启用约束排除时，这个SELECT将完全不会扫描child1000，从而提高性能。

目前，约束排除只在通过继承树实现表分区的情况中被默认启用。为所有表启用它会增加额外的规划开销，特别是在简单查询上并且不会产生任何好处。如果没有用传统继承



树分区的表时，最好是完全关闭它。（注意分区表的等效特性是由单独的参数控制的，`enable_partition_pruning`。）

更多关于使用约束排除实现分区的信息请参阅Section 5.11.5

`cursor_tuple_fraction` (floating point)

设置规划器对将被检索的一个游标的行的比例的估计。默认值是 0.1。更小的值使得规划器偏向为游标使用“快速开始”计划，它将很快地检索前几行但是可能需要很长时间来获取所有行。更大的值强调总的估计时间。最大设置为 1.0，游标将和普通查询完全一样地被规划，只考虑总估计时间并且不考虑前几行会被多快地返回。

`from_collapse_limit` (integer)

如果生成的FROM列表不超过这么多项，规划器将把子查询融合到上层查询。较小的值可以减少规划时间，但是可能会生成较差的查询计划。默认值是 8。详见Section 14.3

将这个值设置为`geqo_threshold`或更大，可能触发使用 GEQO 规划器，从而产生非最优计划。见Section 17.7.3

`jit` (boolean)

决定如果可用（见Chapter 29，LightDB是否可以使用JIT编译。默认值是on。

`join_collapse_limit` (integer)

如果得出的列表中不超过这么多项，那么规划器将把显式JOIN（除了FULL JOIN）结构重写到 FROM项列表中。较小的值可减少规划时间，但是可能会生成差些的查询计划。

默认情况下，这个变量被设置成和`from_collapse_limit`相同，这样适合大多数使用。把它设置为 1 可避免任何显式JOIN的重排序。因此查询中指定的显式连接顺序就是关系被连接的实际顺序。因为查询规划器并不是总能选取最优的连接顺序，高级用户可以选择暂时把这个变量设置为 1，然后显式地指定他们想要的连接顺序。更多信息请见Section 14.3

将这个值设置为`geqo_threshold`或更大，可能触发使用 GEQO 规划器，从而产生非最优计划。见Section 17.7.3

`parallel_leader_participation` (boolean)

允许leader进程在Gather 和 Gather Merge节点下执行查询计划，而不是等待worker进程。默认值是on。设置该值为off可以降低workers因为leader读取元组的速度不够快而被阻塞的可能性，但在第一个元组生成之前需要leader进程等待worker进程启动。leader能帮助或阻碍性能的程度取决于计划类型，workers的数量和查询持续时间。

`force_parallel_mode` (enum)

允许应用并行查询作为测试目的，即使在有预期性能好处的情况下。`force_parallel_mode` 的允许值为off（仅在期望提高性能时使用并行模式），on（对被认为是安全的所有查询强制并行查询），以及regress（类似on，但是有额外的行为变化在下面说明）。

更具体地说，将该值设置为on的将在看起来安全的任何查询计划的顶部添加一个Gather节点，以便查询在并行worker内部运行。即使当并行worker不可用或不能使用时，例如启动子事务之类的操作将被禁止，在并行查询上下文中将被禁止，除非规划器认为这会导致查询失败。如果设置此选项时发生失败或意外结果，查询使用的一些函数可能需要被标记为PARALLEL UNSAFE（或者，可能是PARALLEL RESTRICTED）。

设置该值为regress与设置为on 具有相同的所有效果，加上一些附加效果，为了便于自动回归测试。通常，来自并行worker的消息包含一个上下文行表示之，但是regress的设置会抑制这一行，因此输出与非并行执行时相同。此外，通过此设置隐藏在EXPLAIN输出中，Gather节点添加到计划，以便输出匹配如果将此设置off将获得的结果。

plan\_cache\_mode (enum)

准备语句（显式准备或隐式生成的，例如 PL/pgSQL）可以使用自定义或通用计划执行。使用其特定的参数值集为每个执行重新生成自定义计划，而通用计划不依赖于参数值，并且可以在执行中重复使用。因此，使用通用计划可以节省计划时间，但如果理想计划严重依赖参数值，则通用计划可能效率低下。这些选项之间的选择通常是自动进行的，但可以通过plan\_cache\_mode覆盖它。允许的值为 auto（默认的）、force\_custom\_plan 和 force\_generic\_plan。这个设置是在执行缓存计划时考虑，而不是在准备计划时考虑。更多信息请参阅 PREPARE。

## 17.8. 错误报告和日志

### 17.8.1. 在哪里做日志

log\_destination (string)

LightDB支持多种方法来记录服务器消息，包括stderr、csvlog和syslog。在 Windows 上还支持eventlog。设置这个参数为一个由想要的日志目的地的列表，之间用逗号分隔。默认值是只记录到stderr。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

如果csvlog被包括在log\_destination中，日志项会以“逗号分隔值”（CSV）格式被输出，这样可以很方便地把日志载入到程序中。详见Section 17.8.4 要产生 CSV 格式的日志输出，必须启用logging\_collector。

当包括有stderr或csvlog时，会创建文件current\_logfiles来记录当前正在被日志收集器使用的日志文件的位置以及相关的日志目的地。这提供了一种查找实例当前使用的日志的便利手段。这里是该文件内容的一个例子：

```
stderr log/lightdb.log
csvlog log/lightdb.csv
```

当由于轮转效应创建一个新的日志文件时以及log\_destination被重载时，current\_logfiles文件会被重建。当log\_destination中不包括stderr和csvlog时以及当日志收集器被禁用时，这个文件会被删除。

#### Note

在大多数 Unix 系统上，你将需要修改系统的syslog守护进程的配置来使用log\_destination的syslog选项。LightDB可以在syslog设备LOCAL0到LOCAL7中记录（见syslog\_facility），但是大部分平台上的默认syslog配置会丢弃所有这种消息。你将需要增加这样的内容：

```
local0.* /var/log/lightdb
```

到syslog守护进程的配置文件来让它工作。

在 Windows 上，当你使用log\_destination的eventlog选项时，你应该在操作系统中注册一个事件源极其库，这样 Windows 事件查看器能够清楚地显示事件日志消息。详见Section 16.12

logging\_collector (boolean)

这个参数启用日志收集器，它是一个捕捉被发送到stderr的日志消息的后台进程，并且它会将这些消息重定向到日志文件中。这种方法比记录到syslog通常更有用，因为某些

类型的消息不会在syslog输出中出现（一个常见的例子是动态链接器错误消息；另一个例子是由archive\_command等脚本产生的错误消息）。这个参数只能在服务器启动时设置。

### Note

也可以不使用日志收集器而把日志记录到stderr，日志消息将只会去到服务器的stderr被定向到的位置。不过，那种方法只适合于低日志量，因为它没有提供方法来轮转日志文件。还有，在某些不使用日志收集器的平台上可能会导致丢失或者混淆日志输出，因为多个进程并发写入同一个日志文件时会覆盖彼此的输出。

### Note

日志收集器被设计成从来不会丢失消息。这意味着在极高的负载下，如果服务器进程试图在收集器已经落后时发送更多的日志消息，那么它会被阻塞。相反，syslog倾向于在无法写入消息时丢掉消息，这意味着在这样的情况下它可能会无法记录某些消息，但是它不会阻塞系统的其他部分。

#### log\_directory (string)

当logging\_collector被启用时，这个参数决定日志文件将在哪个目录下创建。它可以被指定为一个绝对路径，也可以被指定为一个相对于集簇数据目录的相对路径。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。默认是log。

#### log\_filename (string)

当logging\_collector被启用时，这个参数设置被创建的日志文件的文件名。该值被视为一种strftime模式，因此%转义可以被用来指定根据时间变化的文件名（注意如果有任何时区独立的%转义，计算将在由log\_timezone指定的时区中完成）。被支持的%转义和开放组织的strftime<sup>1</sup>说明中列举的类似。注意系统的strftime不会被直接使用，因此平台相关（非标准）的扩展无法工作。默认是lightdb-%Y-%m-%d\_%H%M%S.log。

如果你不使用转义来指定一个文件名，你应该计划使用一个日志轮转工具来避免最终填满整个磁盘。在 8.4 发行之之前，如果不存在%转义，LightDB将追加新日志文件创建时间的纪元，但是现在已经不再这样做了。

如果在log\_destination中启用了 CSV 格式输出，.csv将会被追加到时间戳日志文件名中来创建 CSV 格式输出（如果log\_filename以.log结尾，该后缀会被替换）。

这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

#### log\_file\_mode (integer)

在 Unix 系统上，当logging\_collector被启用时，这个参数设置日志文件的权限（在微软 Windows 上这个参数将被忽略）。这个参数值应当是一个数字形式的模式，它可以被chmod和umask系统调用接受（要使用通常的十进制格式，该数字必须以一个0（零）开始）。

默认的权限是0600，表示只有服务器拥有者才能读取或写入日志文件。其他常用的设置是0640，它允许拥有者的组成员读取文件。不过要注意你需要修改log\_directory为将文件存储在集簇数据目录之外的某个位置，才能利用这个设置。在任何情况下，让日志文件变成任何人都可读是不明智的，因为日志文件中可能包含敏感数据。

这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

<sup>1</sup> <https://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html>

`log_rotation_age` (integer)

当`logging_collector`被启用时，这个参数决定使用一个单个日志文件的最大时间量，之后将创立一个新的日志文件。如果指定值时没有单位，则以分钟为单位。默认为24小时。将这个参数设置为零将禁用基于时间的新日志文件创建。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

`lightdb_log_retention_age` (integer)

当启用`logging_collector`参数时，此参数确定保留志文件的最长时间，超过此时间的日志将在切换日志文件时被清除。如果没有指定单位，则将其作为分钟处理。默认值为0。将其设置为零以禁用基于时间的删除日志文件功能。此参数只能在`lightdb.conf`文件或服务器命令行中设置。

`log_rotation_size` (integer)

当`logging_collector`被启用时，这个参数决定一个个体日志文件的最大尺寸。当这些数据量被发送到一个日志文件后，将创建一个新的日志文件。如果指定值的时候没有单位，则以千字节为单位。默认值是10兆字节。设置为零时将禁用基于大小创建新的日志文件。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

`log_truncate_on_rotation` (boolean)

当`logging_collector`被启用时，这个参数将导致LightDB截断（覆盖而不是追加）任何已有的同名日志文件。不过，截断只在一个新文件由于基于时间的轮转被打开时发生，在服务器启动或基于尺寸的轮转时不会发生。如果被关闭，在所有情况下以前存在的文件将被追加。例如，使用这个设置和一个类似`lightdb-%H.log`的`log_filename`将导致产生24个每小时的日志文件，并且循环地覆盖它们。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

例子：要保留7天的日志，每天的一个日志文件被命令为`server_log.Mon`、`server_log.Tue`等等，并且自动用本周的日志覆盖上一周的日志。可以这样做：将`log_filename`设置为`server_log.%a`、将`log_truncate_on_rotation`设置为`on`并且将`log_rotation_age`设置为1440。

例子：要保留24小时的日志，每小时一个日志文件，但是在日志文件尺寸超过1GB时轮转。可以这样做：将`log_filename`设置为`server_log.%H%M`、将`log_truncate_on_rotation`设置为`on`、将`log_rotation_age`设置为60并且将`log_rotation_size`设置为1000000。在`log_filename`中包括%M允许发生任何尺寸驱动的轮转来选择一个不同于每个小时的初始文件名的新文件名。

`syslog_facility` (enum)

当启用了向syslog记录时，这个参数决定要使用的syslog“设备”。你可以在`LOCAL0`、`LOCAL1`、`LOCAL2`、`LOCAL3`、`LOCAL4`、`LOCAL5`、`LOCAL6`、`LOCAL7`中选择，默认值是`LOCAL0`。还请参阅系统的syslog守护进程的文档。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

`syslog_ident` (string)

当启用了向syslog记录时，这个参数决定用来标识syslog中的LightDB消息的程序名。默认值是`postgres`。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

`syslog_sequence_numbers` (boolean)

当日志被记录到syslog并且这个设置为`on`（默认）时，每一个消息会被加上一个增长的序号作为前缀（例如[2]）。这种行为避开了很多syslog实现默认采用的“——上一个消息重复N次——”形式。在现代syslog实现中，抑制重复消息是可以配置的（例如`rsyslog`中的`$RepeatedMsgReduction`），因此这个参数可能不是必需的。此外，如果你真的想抑制重复消息，你可以把这个参数设置为`off`。

这个参数只能在lightdb.conf文件或者服务器命令行上设置。

syslog\_split\_messages (boolean)

当启用把日志记录到syslog时，这个参数决定消息如何送达 syslog。当设置为 on（默认）时，消息会被分成行，并且长的行也会被划分以便能够放到 1024 字节中，这是传统 syslog 实现一种典型的尺寸限制。当设置为 off 时，LightDB 服务器日志消息会被原样送达 syslog 服务，而处理可能的大体量消息的任务由 syslog 服务负责。

如果 syslog 最终被记录到一个文本文件中，那么两种设置的效果是一样的，但最好设置为 on，因为大部分 syslog 实现要么不能处理大型消息，要么需要做特殊的配置以处理大型消息。但是如果 syslog 最终写入到某种其他媒介，有必要让消息保持逻辑上的整体性（也更加有用）。

这个参数只能在lightdb.conf文件或者服务器命令行上设置。

event\_source (string)

当启用了向事件日志记录时，这个参数决定用来标识日志中LightDB消息的程序名。默认值是LightDB。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

## 17.8.2. 什么时候记录日志

log\_min\_messages (enum)

控制哪些消息级别 被写入到服务器日志。有效值是DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、LOG、FATAL和PANIC。每个级别都包括以后的所有级别。级别越靠后，被发送的消息越少。默认值是WARNING。注意LOG在这里有与client\_min\_messages中不同的排名。只有超级用户可以改变这个设置。

log\_min\_error\_statement (enum)

控制哪些导致一个错误情况的 SQL 语句被记录在服务器日志中。任何指定 严重级别 或更高级别的消息的当前 SQL 语句将被包括在日志项中。有效值是DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、LOG、FATAL和PANIC。默认值是ERROR，它表示导致错误、日志消息、致命错误或恐慌错误的语句将被记录在日志中。要有效地关闭记录失败语句，将这个参数设置为PANIC。只有超级用户可以改变这个设置。

log\_min\_duration\_statement (integer)

如果语句运行至少指定的时间量，将导致记录每一个这种完成的语句的持续时间。例如，如果你设置它为250ms，那么所有运行 250ms 或更久的 SQL 语句将被记录。启用这个参数可以有助于追踪应用中未优化的查询。如果指定值时没有单位，则以毫秒为单位。将这个参数设置为零将打印所有语句的执行时间。设置为 -1（默认值）将停止记录语句持续时间。只有超级用户可以改变这个设置。

这会覆盖log\_min\_duration\_sample，意味着持续时间超过此设置的查询不进行抽样，并且始终被记录。

对于使用扩展查询协议的客户端，解析、绑定和执行步骤的持续时间将被独立记录。

### Note

当把这个选项和log\_statement一起使用时，已经被log\_statement记录的语句文本不会在持续时间日志消息中重复。如果你没有使用syslog，我们推荐你使用log\_line\_prefix记录 PID 或会话 ID，这样你可以使用进程 ID 或会话 ID 把语句消息链接到后来的持续时间消息。

log\_min\_duration\_sample (integer)

允许对最少运行了指定时间的已完成语句的持续时间进行抽样。这将生成与log\_min\_duration\_statement 相同的日志条目，但仅适用于已执行语句的子集，采样率由 log\_statement\_sample\_rate 控制。例如，如果将它设置为 100ms则所有运行100ms 或更长的 SQL 语句都将被采样。当流量过高而无法记录所有查询时，启用此参数会很有帮助。如果指定此值时没有单位，则以毫秒为单位。设置为零将采样所有语句持续时间。-1（默认值）禁用采样语句持续时间。只有超级用户可以更改此设置。

此设置的优先级低于 log\_min\_duration\_statement，意味着持续时间超过 log\_min\_duration\_statement 的语句不被采样，并且始终被记录。

log\_min\_duration\_statement的其他注释也适用于此设置。

log\_statement\_sample\_rate (floating point)

确定持续时间超过 log\_min\_duration\_sample的语句的部分。采样是随机的，例如0.5意味着从统计学上讲，任何给定语句都有1/2的概率将被记录。默认值为 1.0，意味着要记录所有采样语句。设置为零将禁用采样语句持续时间日志，与将log\_min\_duration\_sample设为 -1 相同。只有超级用户可以更改此设置。

log\_transaction\_sample\_rate (floating point)

设置所有语句都被记录的交易的的部分，以及由于其他原因而记录的语句。它适用于每个新事务，而不考虑其语句的持续时间。采样是随机的，例如 0.1意味着从统计学上讲，任何给定事务都有十分之一的机会将被记录。log\_transaction\_sample\_rate 有助于构建一个事务示例。默认值为 0，表示不记录任何额外事务的语句。设置这个为1将记录所有交易的所有语句。只有超级用户可以更改此设置。

**Note**

就像所有的语句-日志选项一样，这个选项可能会增加大量开销。

Table 17. 解释了LightDB所使用的消息严重级别。如果日志输出被发送到syslog或 Windows的eventlog，严重级别会按照表中所示进行转换。

Table 17.2. 消息严重级别

严重性	用法	syslog	eventlog
DEBUG1 .. DEBUG5	为开发者提供连续的更详细的信息。	DEBUG	INFORMATION
INFO	提供用户隐式要求的信息，例如来自VACUUM VERBOSE的输出。	INFO	INFORMATION
NOTICE	提供可能对用户有用的信息，例如长标识符截断提示。	NOTICE	INFORMATION
WARNING	提供可能出现的问题的警告，例如在一个事务块外COMMIT。	NOTICE	WARNING
ERROR	报告一个导致当前命令中断的错误。	WARNING	ERROR
LOG	报告管理员可能感兴趣的信息，例如检查点活动。	INFO	INFORMATION
FATAL	报告一个导致当前会话中断的错误。	ERR	ERROR
PANIC	报告一个导致所有数据库会话中断的错误。	CRIT	ERROR

### 17.8.3. 记录什么到日志

`application_name` (string)

`application_name`可以是任意小于NAMEDATALEN个字符（标准编译中是 64 个字符）的字符串。这通常由一个应用通过到服务器的连接设置。该名称将被显示在`pg_stat_activity`视图中并被包括在 `CSV` 日志项中。它也会被通过`log_line_prefix`包括在普通日志项中。只有可打印 ASCII 字符能被使用在`application_name`之中。其他字符将被替换为问号(?)。

`debug_print_parse` (boolean)

`debug_print_rewritten` (boolean)

`debug_print_plan` (boolean)

这个参数启用发出各种调试输出。当设置时，会打印生成的解析树，查询重写输出，或执行的每个查询的执行计划。这些信息是在LOG 信息级别发出，因此默认的，它们会出现在服务器日志中，但不会发送给客户端。可以通过 `client_min_messages`和/或`log_min_messages` 来设置。这些参数缺省是off。这些参数将会让多种调试输出被发出。当被设置时，它们为每一个被执行的查询打印结果分析树、查询重写器输出或执行计划。这些消息在LOG消息级别上被发出，因此默认情况下它们将出现在服务器日志中但不会被发送到客户端。你可以通过调整`client_min_messages`和/或`log_min_messages`来改变这种情况。这些参数默认是关闭的。

`debug_pretty_print` (boolean)

当被设置时，`debug_pretty_print`会缩进由`debug_print_parse`、`debug_print_rewritten`或 `debug_print_plan`产生的输出。这将导致比关闭参数时使用的“紧凑”模式可读性更强但是更长的输出。它默认是打开的。

`log_checkpoints` (boolean)

导致检查点和重启点被记录在服务器日志中。一些统计信息也被包括在日志消息中，包括写入缓冲区的数据和写它们所花的时间。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。默认值是关闭。

`log_connections` (boolean)

导致每一次尝试对服务器的连接被记录，客户端认证(如需要)和认证的成功完成也会被记录。只有超级用户能在会话开始时更改这个参数，在会话中它不能被更改。默认为off。

#### Note

某些客户端程序（例如`ltsql`）在要求密码时会尝试连接两次，因此重复的“收到连接”消息并不一定表示一个错误。

`log_disconnections` (boolean)

导致会话终止被记录。日志输出提供的信息类似于 `log_connections`，不过还外加会话的持续时间。只有超级用户能在会话开始时更改这个参数，在会话中它不能被更改。默认为off。

`log_duration` (boolean)

导致每一个完成的语句的持续时间被记录。默认值是off。只有超级用户可以改变这个设置。

对于使用扩展查询协议的客户端，解析、绑定和执行步骤的持续时间将被独立记录。

## Note

启用log\_duration和设置log\_min\_duration\_statement为零之间的区别是，超过log\_min\_duration\_statement强制查询的文本被记录，但这个选项不会。因此，如果log\_duration为on并且log\_min\_duration\_statement为正值，所有持续时间都将被记录，但是只有超过阈值的语句才会被记录查询文本。这种行为有助于在高负载安装中收集统计信息。

## log\_error\_verbosity (enum)

控制为每一个被记录的消息要写入到服务器日志的细节量。有效值是TERSE、DEFAULT和VERBOSE，每一个都为显示的消息增加更多域。TERSE排除记录DETAIL、HINT、QUERY和CONTEXT错误信息。VERBOSE输出包括SQLSTATE错误码（见Appendix O）以及产生错误的源代码文件名、函数名和行号。只有超级用户能够更改这个设置。

## log\_hostname (boolean)

默认情况下，连接日志消息只显示连接主机的 IP 地址。打开这个参数将导致也记录主机名。注意根据你的主机名解析设置，这可能会导致很微小的性能损失。这个参数只能在lightdb.conf文件中或在服务器命令行上设置。

## log\_line\_prefix (string)

这是一个printf风格的字符串，它在每个日志行的开头输出。%字符开始“转义序列”，它将被按照下文描述的替换成状态信息。未识别的转义被忽略。其他字符被直接复制到日志行。某些转义只被会话进程识别并且被主服务器进程等后台进程当作空。通过指定一个在%之后和该选项之前的数字可以让状态信息左对齐或右对齐。负值将导致在右边用空格填充状态信息已达到最小宽度，而正值则在左边填充。填充对于日志文件的人类可读性大有帮助。

这个参数只能在lightdb.conf文件中或在服务器命令行上设置。默认值是'%m [%p] '，它记录时间戳和进程ID。

转义	效果	只限会话
%a	应用名	是
%u	用户名	是
%d	数据库名	是
%r	远程主机名或 IP 地址，以及远程端口	是
%h	远程主机名或 IP 地址	是
%b	后端类型	否
%p	进程 ID	否
%P	并行组leader的进程ID，如果该进程是一个并行查询worker	no
%t	无毫秒的时间戳	否
%m	带毫秒的时间戳	否
%n	带毫秒的时间戳（作为Unix 时代）	no
%i	命令标签：会话当前命令的类型	是



转义	效果	只限会话
%e	SQLSTATE 错误代码	否
%c	会话 ID: 见下文	否
%l	对每个会话或进程的日志行号, 从 1 开始	否
%s	进程开始的时间戳	否
%v	虚拟事务 ID (backendID/localXID)	否
%x	事务 ID (如果未分配则为 0)	否
%q	不产生输出, 但是告诉非会话进程在字符串的这一点停止; 会话进程忽略	否
%Q	当前查询的查询标识符。查询标识符默认是不计算的, 所以这个部分将是零, 除非compute_query_id 参数被激活或者配置了计算查询标识符的第三方模块	yes
%%	纯文字 %	否

后端类型对应视图 `pg_stat_activity` 中的 `backend_type` 列。但是其他类型可能会出现在日志中而不显示在该视图中。

`%c` 转义打印一个准唯一的会话标识符, 它由两个 4 字节的十六进制数 (不带先导零) 组成, 以点号分隔。这些数字是进程启动时间和进程 ID, 因此 `%c` 也可以被用作保存打印这些项的方式的空间。例如, 要从 `pg_stat_activity` 生成会话标识符, 使用这个查询:

```
SELECT to_hex(trunc(EXTRACT(EPOCH FROM backend_start))::integer) || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

### Tip

如果你为 `log_line_prefix` 设置了非空值, 你通常应该让它的最后一个字符为空格, 这样用以提供和日志行的剩余部分的视觉区别。也可以使用标点符号。

### Tip

syslog 产生自己的时间戳和进程 ID 信息, 因此如果你记录到 syslog 你可能不希望包括哪些转义。

### Tip

在包括仅在会话 (后端) 上下文中可用的信息 (如用户名或者数据库名) 时, `%q` 转义很有用。例如:

```
log_line_prefix = '%m [%p] %q%u@%d/%a '
```

### Note

对于log\_statement输出的行，%Q 总是报告零标识符， 因为log\_statement在标识符能被计算之前生成输出，包括无效标识符不能计算的无效语句。

### log\_lock\_waits (boolean)

控制当一个会话为获得一个锁等到超过deadlock\_timeout时，是否要产生一个日志消息。这有助于决定是否所等待造成了性能低下。默认值是off。只有超级用户可以更改这个设置。

### log\_recovery\_conflict\_waits (boolean)

控制启动过程等待时间超过针对恢复冲突的deadlock\_timeout的时候是否产生日志消息。这对于决定恢复冲突是否会阻止恢复应用WAL有所帮助。

默认为off。这个参数只能在lightdb.conf文件中或服务器命令行中设置。

### log\_parameter\_max\_length (integer)

如果大于零，则使用非错误语句日志消息记录的每个绑定参数值都被裁剪为这么多字节。零禁用非错误语句日志的绑定参数日志。-1（默认值）允许绑定参数被完整登录。如果指定此值时没有单位，则将其作为字节。只有超级用户可以更改此设置。

此设置仅影响作为结果打印的日志消息log\_statement, log\_duration, 和相关的设置。该设置的非零值会增加一些开销，特别是当参数以二进制形式发送时，因为需要转换为文本。

### log\_parameter\_max\_length\_on\_error (integer)

如果大于零，则错误消息中报告的每个绑定参数值都将裁剪为这么多字节。零（默认值）禁止在错误消息中包含绑定参数。-1允许打印完整绑定参数。如果指定此值时没有单位，则将其作为字节。

该设置的非零值会增加开销，由于LightDB需要在每条语句的开始处将参数值的文本表示存储在内存中，无论最终是否会发生错误。当绑定参数以二进制形式发送时，开销比以文本形式发送时更大，因为前者需要数据转换，而后者只需要复制字符串。

### log\_statement (enum)

控制哪些 SQL 语句被记录。有效值是 none (off)、ddl、mod和 all (所有语句)。ddl记录所有数据定义语句，例如CREATE、ALTER和 DROP语句。mod记录所有ddl语句，外加数据修改语句例如INSERT, UPDATE、DELETE、TRUNCATE, 和COPY FROM。如果PREPARE、EXECUTE和 EXPLAIN ANALYZE包含合适类型的命令，它们也会被记录。对于使用扩展查询协议的客户端，当收到一个执行消息时会产生日志并且会包括绑定参数的值（任何内嵌的单引号会被双写）。

默认值是none。只有超级用户可以改变这个设置。

### Note

即使使用log\_statement = all设置，包含简单语法错误的语句也不会被记录。这是因为只有在完成基本语法解析并确定了语句类型之后才会发出日

志消息。在扩展查询协议的情况下，在执行阶段之前（即在解析分析或规划期间）出错的语句也不会被记录。将`log_min_error_statement`设置为`ERROR`（或更低）来记录这种语句。

`log_replication_commands` (boolean)

导致每一个复制命令都被记录在服务器日志中。关于复制命令的详细信息请见 Section 49.4 默认值是`off`。只有超级用户可以更改这个设置。

`log_temp_files` (integer)

控制记录临时文件名和尺寸。临时文件可以被创建用来排序、哈希和存储临时查询结果。如果启用这个设置，当每一个临时文件被删除时都会产生一个日志项。一个零值记录所有临时文件信息，而正值只记录尺寸大于或等于指定数据量的文件。如果指定值时没有单位，则以千字节为单位。默认设置为 `-1`，它禁用这种记录。只有超级用户可以更改这个设置。

`log_timezone` (string)

设置在服务器日志中写入的时间戳的时区。和`TimeZone`不同，这个值是集群范围的，因此所有会话将报告一致的时间戳。内建默认值是`GMT`，但是通常会被在`lightdb.conf`中覆盖。`initdb`将安装一个对应于其系统环境的设置。详见 Section 8.5.3 这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

## 17.8.4. 使用 CSV 格式的日志输出

在`log_destination`列表中包括`csvlog`提供了一种便捷方式将日志文件导入到一个数据库表。这个选项发出逗号分隔值（CSV）格式的日志行，包括这些列：带毫秒的时间戳、用户名、数据库名、进程 ID、客户端主机:端口号、会话 ID、每个会话的行号、命令标签、会话开始时间、虚拟事务 ID、普通事务 ID、错误严重性、SQLSTATE 代码、错误消息、错误消息详情、提示、导致错误的内部查询（如果有）、错误位置所在的字符计数、错误上下文、导致错误的用户查询（如果有且被`log_min_error_statement`启用）、错误位置所在的字符计数、在 LightDB 源代码中错误的位置（如果`log_error_verbosity`被设置为`verbose`）以及应用名，后端类型，并行组`leader`的进程ID和查询ID。下面是一个定义用来存储 CSV 格式日志输出的样表：

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
```

```

context text,
query text,
query_pos integer,
location text,
application_name text,
backend_type text,
leader_pid integer,
query_id bigint,
PRIMARY KEY (session_id, session_line_num)
);

```

It is also possible to access the file as a foreign table, using the supplied `file_fdw` module.

使用COPY FROM命令将一个日志文件导入到这个表中：

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

也可以作为外部表访问该文件，使用提供的 `file_fdw` 模块。

你可以做一些事情来简化导入 CSV 日志文件：

1. 设置`log_filename`和`log_rotation_age`为你的日志文件提供一种一致的、可预测的命名空间。这让你预测文件名会是怎样以及知道什么时候一个个体日志文件完成并且因此准备好被导入。
2. 将`log_rotation_size`设置为 0 来禁用基于尺寸的日志轮转，因为它使得日志文件名难以预测。
3. 将`log_truncate_on_rotation`设置为on，这样在同一个文件中旧日志数据不会与新数据混杂。
4. 上述表定义包括一个主键声明。这有助于避免意外地两次导入相同的信息。COPY命令一次提交所有它导入的数据，因此任何错误将导致整个导入失败。如果你导入一个部分完成的日志文件并且稍后当它完全完成后再次导入，主键违背将导致导入失败。请等到日志完成且被关闭之后再导入。这个过程也可以避免意外地导入部分完成的行，这种行也将导致COPY失败。

## 17.8.5. 进程标题

这些设置控制服务器进程的进程标题如何被修改。进程标题通常可以用ps或者 Windows 上的进程浏览器等程序来查看。详见Section 25.1

`cluster_name` (string)

为不同目的设置标识这个数据库集群(实例)的名称。此集群名称出现在该集群中所有服务器进程的进程标题中。此外，它还是备用连接的默认应用名称。(参见 `synchronous_standby_names`.)

这个名称可以是任何长度不超过NAMEDATALEN个字符（在标准编译中是 64字符）的任何字符串。只有可打印的 ASCII 字符能被用在`cluster_name`值中。其他字符将被替换为问号(?)。如果这个参数被设置为空字符串''（也是默认值），将不会显示名称。这个参数只能在服务器启动时设置。

`update_process_title` (boolean)

启用进程标题更新，每次服务器接收到一个新的 SQL 命令时都更新进程标题。在大部分平台上这个设置默认为on，但是由于 Windows 上更新进程标题的开销更大，所以在 Windows 这个设置默认为off。只有超级用户能更改这个设置。

## 17.9. 运行时统计数据

### 17.9.1. 查询和索引统计收集器

这些参数控制服务器范围的统计数据收集特性。当统计收集被启用时，被产生的数据可以通过`lt_stat`和`pg_statio`系统视图族访问。详见Chapter 25

`track_activities` (boolean)

启用对每个会话的当前执行命令的信息收集，还有它的标识符和命令开始执行的时间。这个参数默认为打开。注意即使被启用，这些信息也不是对所有用户可见，只有超级用户和拥有报告信息的会话的用户可见，因此它不会表现为一个安全风险。只有超级用户可以更改这个设置。

`track_activity_query_size` (integer)

为每个活动会话指定存储当前执行命令的文本所保留的内存量，它们被用于`pg_stat_activity.query`域。如果指定值时没有单位，则以字节为单位。默认值是1024字节。这个参数只能在服务器启动时被设置。

`track_counts` (boolean)

启用在数据库活动上的统计收集。这个参数默认为打开，因为自动清理守护进程需要被收集的信息。只有超级用户可以更改这个设置。

`track_io_timing` (boolean)

启用对系统 I/O 调用的计时。这个参数默认为关闭，因为它将重复地向操作系统查询当前时间，这会在某些平台上导致显著的负荷。你可以使用`lt_test_timing`工具来度量你的系统中计时的开销。I/O 计时信息被显示在 `pg_stat_database`中，当`BUFFERS`选项被使用时的`EXPLAIN`输出中，通过`autovacuum`对`auto-vacuums` 和 `auto-analyzes`，当`log_autovacuum_min_duration` 被`lt_stat_statements`设置时。只有超级用户可以更改这个设置。

`track_functions` (enum)

启用跟踪函数调用计数和用时。指定`pl`只跟踪过程语言函数，指定`all`还会跟踪 SQL 和 C 语言函数。默认值是`none`，它禁用函数统计跟踪。只有超级用户可以更改这个设置。

#### Note

简单到足以被“内联”到调用查询中的 SQL 语言函数不会被跟踪，而不管这个设置。

`stats_temp_directory` (string)

设置存储临时统计数据的目录。这可以是一个相对于数据目录的路径或一个绝对路径。默认值是`lt_stat_tmp`。在一个基于 RAM 的文件系统上指明这个参数将降低物理 I/O 需求，并且提高性能。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

### 17.9.2. 统计监控

`compute_query_id` (enum)

启用查询标识符的内核中计算。查询标识符可以显示在`pg_stat_activity`视图中，使用`EXPLAIN`，或者在日志中发出，如果通过`log_line_prefix`参数配置，。`lt_stat_statements`扩展还需要一个查询标识符以计算。注意，如果内核中查询标识符计算方法不可接受，也可以使用外部模块。在这种情况下，必须始终禁用核内计

算。有效的值为off(总是禁用)，on(总是启用)和auto，这让像lt\_stat\_statements的模块自动启用它。默认为auto。

### Note

确认只有一个查询标识符被计算和显示，如果一个查询标识符已经被计算，扩展计算标识符将抛出一个错误。

```
log_statement_stats (boolean)
log_parser_stats (boolean)
log_planner_stats (boolean)
log_executor_stats (boolean)
```

对每个查询，向服务器日志里输出相应模块的性能统计。这是一种粗糙的分析工具。类似于 Unix 的getrusage() 系统功能。log\_statement\_stats报告总的语句统计，而其它的报告针对每个模块的统计。log\_statement\_stats不能和其它任何针对每个模块统计的选项一起启用。所有这些选项都是默认禁用的。只有超级用户可以更改这个设置。

## 17.10. 自动清理

这些设置控制autovacuum特性的行为。详情请参考 Section 22.1.6 注意很多这些设置可以被针对每个表的设置所覆盖，参见Storage Parameters。

```
autovacuum (boolean)
```

控制服务器是否运行自动清理启动器后台进程。默认为开启，不过要自动清理正常工作还需要启用track\_counts。该参数只能在lightdb.conf文件或服务器命令行中设置，不过，通过更改表存储参数可以为表禁用自动清理。

注意即使该参数被禁用，系统也会在需要防止事务ID回卷时发起清理进程。详情请见Section 22.1.5

```
log_autovacuum_min_duration (integer)
```

如果autovacuum执行的每个操作至少运行了指定的时间，该操作会被记录。设置这个为零记录所有autovacuum操作。-1(默认的)禁用日志记录autovacuum操作。如果指定此值时不带单位，则以毫秒为单位。例如，如果将其设置为250ms，那么所有运行250毫秒或更长时间的自动vacuums和分析都将被记录。此外，当此参数设置为-1以外的任何值时，如果由于锁冲突或并发删除关系而跳过autovacuum操作，则会记录一条消息。启用此参数有助于跟踪autovacuum活动。该参数只能在lightdb.conf文件或在服务器命令行中设置；但是可以通过更改表存储参数来覆盖单独表的设置。

```
autovacuum_max_workers (integer)
```

指定能同时运行的自动清理进程（除了自动清理启动器之外）的最大数量。默认值为3。该参数只能在服务器启动时设置。

```
autovacuum_naptime (integer)
```

指定自动清理在任意给定数据库上运行的最小延迟。在每一轮中后台进程检查数据库并根据需要为数据库中的表发出VACUUM和ANALYZE命令。如果指定值时没有单位，则以秒为单位。默认值为1分钟(1min)。该参数只能在lightdb.conf文件或在服务器命令行上设置。

```
autovacuum_vacuum_threshold (integer)
```

指定能在一个表上触发VACUUM的被更新或被删除元组的最小数量。默认值为50个元组。该参数只能在lightdb.conf文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

`autovacuum_vacuum_insert_threshold` (integer)

指定在任何一个表中触发 VACUUM 所需要插入的元组数。默认值为 1000 个元组。如果指定了 -1，则 autovacuum 不会根据插入数量在任何表上触发 VACUUM 操作。此参数只能在 `lightdb.conf` 文件或服务器命令行中设置；但可以通过更改表存储参数来覆盖单个表的设置。

`autovacuum_analyze_threshold` (integer)

指定能在一个表上触发 ANALYZE 的被插入、被更新或被删除元组的最小数量。默认值为 50 个元组。该参数只能在 `lightdb.conf` 文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

`autovacuum_vacuum_scale_factor` (floating point)

指定一个表尺寸的分数，在决定是否触发 VACUUM 时将它加到 `autovacuum_vacuum_threshold` 上。默认值为 0.2（表尺寸的 20%）。该参数只能在 `lightdb.conf` 文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

`autovacuum_vacuum_insert_scale_factor` (floating point)

指定一个要添加到 `autovacuum_vacuum_insert_threshold` 中的表的大的比例，在决定是否触发 VACUUM 时。默认值为 0.2（表大小的 20%）。此参数只能在 `lightdb.conf` 文件或服务器命令行中设置；但可以通过更改表存储参数来覆盖单个表的设置。

`autovacuum_analyze_scale_factor` (floating point)

指定一个表尺寸的分数，在决定是否触发 ANALYZE 时将它加到 `autovacuum_analyze_threshold` 上。默认值为 0.1（表尺寸的 10%）。该参数只能在 `lightdb.conf` 文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

`autovacuum_freeze_max_age` (integer)

指定在一个 VACUUM 操作被强制执行来防止表中事务 ID 回卷之前，一个表的 `pg_class.relFrozenxid` 域能保持的最大年龄（事务的）。注意即便自动清理被禁用，系统也将发起自动清理进程来阻止回卷。

清理也允许从 `lt_xact` 子目录中移除旧文件，这也是为什么默认值被设置为较低的 2 亿事务。该参数只能在服务器启动时设置，但是对于个别表可以通过修改表存储参数来降低该设置。详见 Section 22.1.5

`autovacuum_multixact_freeze_max_age` (integer)

指定在一个 VACUUM 操作被强制执行来防止表中多事务 ID 回卷之前，一个表的 `pg_class.relminmxid` 域能保持的最大年龄（多事务的）。注意即便自动清理被禁用，系统也将发起自动清理进程来阻止回卷。

清理多事务也允许从 `lt_multixact/members` 和 `lt_multixact/offsets` 子目录中移除旧文件，这也是为什么默认值被设置为较低的 4 亿事务。该参数只能在服务器启动时设置，但是对于个别表可以通过修改表存储参数来降低该设置。详见 Section 22.1.5.1

`autovacuum_vacuum_cost_delay` (floating point)

指定用于自动 VACUUM 操作中的代价延迟值。如果指定 -1，则使用 `vacuum_cost_delay` 值。如果指定值时没有单位，则以毫秒为单位。默认值为 2 毫秒。该参数只能在 `lightdb.conf` 文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

`autovacuum_vacuum_cost_limit` (integer)

指定用于自动VACUUM操作中的代价限制值。如果指定-1（默认值），则使用`vacuum_cost_limit`值。注意该值被按比例地分配到运行中的自动清理工作者上（如果有多个），因此每一个工作者的限制值之和不会超过这个变量中的值。该参数只能在`lightdb.conf`文件或在服务器命令中设置。但是对个别表可以通过修改表存储参数来覆盖该设置。

## 17.11. 客户端连接默认值

### 17.11.1. 语句行为

`client_min_messages` (enum)

控制被发送给客户端的消息级别。有效值是DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、LOG、NOTICE、WARNING、ERROR。每个级别都包括其后的所有级别。级别越靠后，被发送的消息越少。默认值是NOTICE。注意LOG在这里有与`log_min_messages`中不同的排名。

INFO 级别的消息总是被发送到客户端。

`search_path` (string)

这个变量指定当一个对象（表、数据类型、函数等）被用一个无模式限定的简单名称引用时，用于进行搜索该对象的模式顺序。当在不同模式中有同名对象时，将使用第一个在搜索路径中被找到的对象。一个不属于搜索路径中任何一个模式的对象只能通过用限定名（带点号）指定包含它的模式来引用。

`search_path`的值必需是一个逗号分隔的模式名列表。任何不是一个已有模式的名称，或者是一个用户不具有USAGE权限的模式，将被安静地忽略。

如果列表项之一是特殊名`$user`，则具有SESSION\_USER返回的名字的模式将取代它（如果有这样一个模式并且该用户有该模式的USAGE权限；如果没有，`$user`会被忽略）。

如果列表项之一是特殊名`$lt_syntax_compatible_type`，并且`lightdb_dblevel_syntax_compatible_type`不是off，则具有`lt_syntax_compatible_type`返回的名字的模式将取代它（如果有这样一个模式并且该用户有该模式的USAGE权限）。（如果`lightdb_dblevel_syntax_compatible_type`是off，`$lt_syntax_compatible_type`会被忽略）

系统目录模式`pg_catalog`总是被搜索，不管它是否在搜索路径中被提及。如果它在路径中被提及，那么它将被按照路径指定的顺序搜索。如果`pg_catalog`不在路径中，则它将在任何路径项之前被搜索。

同样，当前会话的临时表模式`pg_temp_nnn`也总是被搜索（如果存在）。它可以在路径中通过使用别名`pg_temp`显式列出。如果在路径中没有列出，那么会首先对它进行搜索（甚至是在`pg_catalog`之前）。然而，临时模式只被用来搜索关系（表、视图、序列等）和数据类型名。它从不用于搜索函数或操作符名。

当对象创建时没有指定一个特定目标模式，它们将被放置在`search_path`中第一个合法模式中。如果搜索路径为空将报告一个错误。

这个参数的缺省值是`"$user", public`。这种设置支持一个数据库（其中没有用户拥有私有模式，并且所有人共享使用`public`）、每个用户私有模式及其组合的共享使用。其它效果可以通过全局或者针对每个用户修改默认搜索路径设置获得。

更多有关方案处理的信息，请参考Section 5.9 特别地，只有当数据库只有一个用户或者有少数的相互信任的用户时，默认配置是合适的。



搜索路径的当前有效值可以通过SQL函数`current_schemas`检查（见Section 9.26）。它和检查`search_path`的值不太一样，因为`current_schemas`显示出现在`search_path`中的项如何被解析。

`mysql`和`oracle`是特殊的模式，只能包含在指定的`lightdb_dblevel_syntax_compatible_type`中。

`row_security` (boolean)

这个变量控制是否以抛出一个错误来代替应用一条行安全性策略。在设置为`on`时，策略正常应用。在设置为`off`时，只要有至少一条策略被应用则查询就会失败。默认为`on`。受限的行可见性会导致不正确的结果时，可将其改成`off`。例如，`lt_dump`默认会做这种更改。这个变量对能绕过每一条行安全性策略的角色（即超级用户和具有`BYPASSRLS`）属性的角色没有效果。

更多关于行安全性策略的信息请见`CREATE POLICY`。

`default_table_access_method` (string)

如果`CREATE`命令没有显式指定访问方法，则此参数指定在创建表或物化视图时使用的默认表访问方法，或者在使用`SELECT ... INTO`时，不允许指定表访问方法。默认为`heap`。

`default_tablespace` (string)

这个变量指定当一个`CREATE`命令没有显式指定一个表空间时，创建对象（表和索引）的默认表空间。

该值要么是一个表空间的名字，要么是一个指定使用当前数据库默认表空间的空字符串。如果该值和任何现有表空间的名字都不匹配，LightDB将自动使用当前数据库的默认表空间。如果指定了一个非默认的表空间，用户必须对它有`CREATE`权限，否则创建企图将失败。

这个变量不被用于临时表，对临时表会使用`temp_tablespace`。

当创建数据库时也会使用这个变量。默认情况下，一个新数据库会从它的模板数据库继承其表空间设置。

当一个分区表建立时如果这个参数被设置为空字符串以外的值，分区表的表空间将被设置为该值，将被用做未来建立分区的默认表空间，即使`default_tablespace`已经改变。

有关表空间的更多的信息，请见Section 20.6

`temp_tablespace` (string)

这个变量指定当一个`CREATE`命令没有显式指定一个表空间时，创建临时对象（临时表和临时表上的索引）的默认表空间。用于排序大型数据集的临时文件也被创建在这些表空间中。

该值是一个表空间名字的列表。当列表中有多个名称时，每次一个临时对象被创建时LightDB随机选择列表中的一个成员。例外是在一个事务中，连续创建的临时对象被放置在表中的连续表空间中。如果列表被选中元素是一个空字符串，LightDB将自动使用当前数据库的默认表空间。

当`temp_tablespace`被交互式地设置时，指定一个不存在的表空间是一种错误，类似于为用户指定一个不具有`CREATE`权限的表空间。不过，当使用一个之前设置的值时，不存在的表空间会被忽略，就像用户缺少`CREATE`权限的表空间一样。特殊地，使用一个在`lightdb.conf`中设置的值时，这条规则起效。

默认值是一个空字符串，它使得所有临时对象被创建在当前数据库的默认表空间中。

参阅`default_tablespace`。

`check_function_bodies` (boolean)

这个参数通常为打开。当设置为off时，它禁用CREATE FUNCTION和CREATE PROCEDURE期间对例程体字符串的验证。禁用验证避免了验证处理的副作用，特别避免了如向前引用导致的伪肯定。在代表其他用户载入函数之前设置这个参数为off；`lt_dump`会自动这样做。

`default_transaction_isolation` (enum)

每个 SQL 事务都有一个隔离级别，可以是“读未提交”、“读已提交”、“可重复读”或者“可序列化”。这个参数控制每个新事务的默认隔离级别。默认是“读已提交”。

更多信息请参阅Chapter 1和SET TRANSACTION。

`default_transaction_read_only` (boolean)

一个只读的 SQL 事务不能修改非临时表。这个参数控制每个新事务的默认只读状态。默认是off（读/写）。

更多信息请参考SET TRANSACTION。

`default_transaction_deferrable` (boolean)

当运行在可序列化隔离级别时，一个可延迟只读 SQL 事务可以在它被允许继续之前延迟一段时间。但是，一旦它开始执行就不会产生任何用来保证可序列化性的负荷；因此序列化代码将没有任何理由因为并发更新而强制它中断，使得这个选项适合于长时间运行的只读事务。

这个参数控制每个新事务的默认可延迟状态。目前它对读写事务或者那些操作在低于可序列化隔离级别上的事务无效。默认值是off。

详情请参阅SET TRANSACTION。

`transaction_isolation` (enum)

此参数反映当前事务的隔离级别。在每个事务开始时，它被设置为`default_transaction_isolation`的当前值。任何后续更改它的尝试都相当于SET TRANSACTION命令。

`transaction_read_only` (boolean)

此参数反映当前事务的只读状态。在每个事务的开始，它被设置为`default_transaction_read_only`的当前值。任何后续尝试更改它的都等同于SET TRANSACTION命令。

`transaction_deferrable` (boolean)

此参数反映当前事务的可延迟性状态。在每个事务的开始，它被设置为`default_transaction_deferrable`的当前值。任何后续尝试更改它的都等同于SET TRANSACTION命令。

`session_replication_role` (enum)

为当前会话控制复制相关的触发器和规则的触发。需要超级用户权限才能设置这个变量，并且会导致丢弃任何之前缓存下来的查询计划。可能的值有origin（默认）、replica和local。

这个设置的预期用途是由逻辑复制系统在应用所复制的更改时将它设置为replica。其效果将是触发器和规则（没有对其默认配置做修改）在复制机上将不会被触发。更多信息请参考ALTER TABLE的子句ENABLE TRIGGER以及ENABLE RULE。

LightDB在内部会把设置origin和local同样对待。第三方复制系统可能会把这两个值用于其内部目的，例如把local用来标出一个不应复制其更改的会话。

因为外键被实现为触发器，将这个参数设置为`replica`还会禁用所有的外键检查，如果使用不当可能会让数据处于一种不一致的状态。

`statement_timeout` (integer)

中止任何使用了超过指定时间量的语句。如果`log_min_error_statement`被设置为`ERROR`或更低，语句如果超时也会被记录。如果指定值时没有单位，则以毫秒为单位。一个零值（默认）将禁用超时。

超时从命令到达服务器的时间开始计算，一直到它被服务器完成为止。如果单个简单查询消息中出现多个 SQL 语句，超时将分别应用于每个语句。（LightDB 13 之前的版本通常将超时应用于整个查询字符串。在扩展查询协议中，当任何与查询相关的消息（解析、绑定、执行、描述）到达时，超时就开始运行，并且可以通过执行完成或同步消息来取消超时。

我们不推荐在`lightdb.conf`中设置`statement_timeout`，因为它会影响所有会话。

`lock_timeout` (integer)

如果任何语句在试图获取表、索引、行或其他数据库对象上的锁时等到超过指定的时间量，该语句将被中止。该时间限制独立地应用于每一次锁获取尝试。该限制会应用到显式锁定请求（如`LOCK TABLE`或不带`NOWAIT`的`SELECT FOR UPDATE`）和隐式获得的锁。如果指定值时没有单位，则以毫秒为单位。一个零值（默认）将禁用超时。

与`statement_timeout`不同，这个超时只在等待锁时发生。注意如果`statement_timeout`为非零，设置`lock_timeout`为相同或更大的值没有意义，因为事务超时将总是第一个被触发。如果`log_min_error_statement`被设置为`ERROR`或更低，超时的语句将被记录。

我们不推荐在`lightdb.conf`中设置`lock_timeout`，因为它会影响所有会话。

`idle_in_transaction_session_timeout` (integer)

终止任何已经闲置（这是指，等待客户端查询）超过这个参数所指定时间量的打开事务的会话。如果这个值指定时没有单位，它被设为毫秒。零值（默认）禁用超时。

此选项可以用于确保空闲会话不会在不合理的时间内持有锁。即使没有持有重要的锁的时候，打开的事务也会防止清理最近死亡的可能只对这个事务可见的元组；所以长时间保持空闲会导致表膨胀。详见Section 22.1

`vacuum_freeze_table_age` (integer)

当表的`pg_class.relfrozensid`域达到该设置指定的年龄时，`VACUUM`会执行一次激进的扫描。激进的扫描与常规`VACUUM`的不同在于它会访问每一个可能包含未冻结 `XID` 或者 `MXID` 的页面，而不只是那些可能包含死亡元组的页面。默认值是 1.5 亿个事务。尽管用户可以把这个值设置为从 0 到 20 亿，`VACUUM`会悄悄地将有效值设置为`autovacuum_freeze_max_age`值的95%，因此在表上启动一次反回卷自动清理之前有机会进行一次定期手动`VACUUM`。更多信息请见Section 22.1.5

`vacuum_freeze_min_age` (integer)

指定`VACUUM`在扫描表时用来决定是否冻结行版本的切断年龄（以事务计）。默认值是 5 千万个事务。尽管用户可以将这个值设置为从 0 到 10 亿，`VACUUM`会悄悄地将有效值设置为`autovacuum_freeze_max_age`值的一半，这样在强制执行的自动清理之间不会有过短的时间间隔。更多信息请见Section 22.1.5

`vacuum_multixact_freeze_table_age` (integer)

如果表的`pg_class.relminmxid`域超过了这个设置指定的年龄，`VACUUM`会执行一次激进的扫描。激进的扫描与常规`VACUUM`的区别在于它会访问每一个可能包含未冻结 `XID` 或

者 `MXID` 的页面，而不是只扫描那些可能包含死亡元组的页面。默认值是 `1.5` 亿个组合事务。尽管用户可以把这个值设置为从 `0` 到 `20` 亿，`VACUUM` 会悄悄地将有效值设置为 `autovacuum_multixact_freeze_max_age` 值的 `95%`，因此在表上启动一次反回卷自动清理之前有机会进行一次定期手动 `VACUUM`。更多信息请见 [Section 22.1.5.1](#)

`vacuum_multixact_freeze_min_age` (integer)

指定 `VACUUM` 在扫描表时用来决定是否把组合事务 `ID` 替换为一个更新的事务 `ID` 或组合事务 `ID` 的切断年龄（以组合事务计）。默认值是 `5` 千万个组合事务。尽管用户可以将这个值设置为从 `0` 到 `10` 亿，`VACUUM` 会悄悄地将有效值设置为 `autovacuum_multixact_freeze_max_age` 值的一半，这样在强制执行的自动清理之间不会有过短的时间间隔。更多信息请见 [Section 22.1.5.1](#)

`bytea_output` (enum)

设置 `bytea` 类型值的输出格式。有效值是 `hex`（默认）和 `escape`（传统的 `LightDB` 格式）。详见 [Section 8.4](#) 不管这个设置的值如何，`bytea` 类型总是接受这两种格式的输出。

`xmlbinary` (enum)

设置二进制值如何被编码为 XML。例如，这适用于通过 `xmlelement` 函数或 `xmlforest` 函数将 `bytea` 值转换到 XML 值。可能的值有 `base64` 和 `hex`，它们都是用 XML 模式标准定义的。默认值是 `base64`。更多关于 XML 相关函数的信息可参阅 [Section 9.15](#)

这里的实际选择都是根据爱好做出的，只受客户端应用中可能存在的限制的约束。两种方法都支持所有可能的值，尽管十六进制编码将比 `base64` 编码更大。

`xmloption` (enum)

当在 XML 和字符串值之间进行转换时，无论设置 `DOCUMENT` 或 `CONTENT` 都是隐式的。可参阅 [Section 8.13](#)。有效值是 `DOCUMENT` 和 `CONTENT`。缺省值是 `CONTENT`。当在 XML 和字符串值之间进行转换时，设置 `DOCUMENT` 或 `CONTENT` 都是隐式的。详见 [Section 8.13](#) 有效值是 `DOCUMENT` 和 `CONTENT`。默认值是 `CONTENT`。

根据 SQL 标准，设置这个选项的命令是：

```
SET XML OPTION { DOCUMENT | CONTENT };
```

这种语法在 `LightDB` 也可用。

`gin_pending_list_limit` (integer)

设置 `fastupdate` 被启用时可以使用的 `GIN` 索引的待处理列表的最大尺寸。如果该列表增长到超过这个最大尺寸，会通过批量将其中的项移入索引的主 `GIN` 数据结构来清理列表。如果指定值时没有单位，则以千字节为单位。默认值是四兆字节（`4MB`）。可以通过更改索引的存储参数来为个别 `GIN` 索引覆盖这个设置。更多信息请见 [Section 62.4](#) 和 [Section 62.5](#)

`lightdb_syntax_compatible_type` (enum)

有效值为 `Oracle`、`MySQL`、`off`。默认值为 `off`。设置 `create database` 命令的默认值。例如，当值为 `oracle` 时，“创建数据库测试”等效于“使用 `lightdb_syntax_compatible_type oracle` 创建数据库测试”。

`lightdb_dblevel_syntax_compatible_type` (enum)

报告当前数据库的语法兼容类型。它取决于创建数据库时 `lightdb_syntax_compatible_type` 的值。

当该值设置为Oracle时，以下功能会有所不同：

- 当运算符为“/”且两个操作数的数据类型均为整数时，返回值类型为数字。
- 当表达式的类型为文本，varchar(n)， varchar2(n)， nvarchar2(n)或char(n)时，空字符串被认为是NULL。
- 当表达式的类型为char(n)时，转换为text或varchar(n)而不删除尾部空格。
- 空字符串可以转换为整数类型、浮点类型、数字、日期、时间、带时区的时间戳和不带时区的时戳，值为NULL。
- 支持rowid
- 支持ROWNUM
- PLPGSQL中使用的Sysdate和Systimestamp将在不同的语句中获得不同的值，否则获得相同的值。
- search\_path可以包括oracle。当search\_path包含oracle时，首先调用oracle模式的函数。
- transform\_null\_equals无效

当该值设置为MySQL时，以下功能会有所不同：

- search\_path可以包括mysql。当search\_path包含mysql时，首先调用mysql模式的函数。
- 支持用户变量。请参见Section 4.2.14
- transform\_null\_equals无效
- 与mysql相同，integer类型的值作为布尔值使用。例如，“select\*from test where 1”相当于“select\*form test where true”。
- 在insert和update语句中，可以使用varchar, char(n)和text给timestamp with time zone, datetime类型的属性赋值
- 支持日期类型（仅包含date, datetime and timestamp with time zone）的表达式和字符类型值表达式比较
- CAST(expr AS char[(n)]) (参考Section 4.2.9)接受任何类型的表达式，并生成VARCHAR数据类型的字符串。
- case表达式(see Section 9.18.)结果集合支持更多类型组合。

lightdb\_mysql\_lower\_case\_table\_names (boolean)

在Mysql兼容模式下，是否开启表名全小写特性。这个参数的值为true或者false，默认值是true。true: 开启 false: 不开启

lightdb\_ascii\_zero\_store\_value (整形)

此值由lightdb\_ascii\_zero\_store\_value决定，数据库创建的时候设定。Oracle模式或Mysql模式下此值才有意义。

Lightdb 23.3 引入了一个新的GUC参数 lightdb\_ascii\_zero\_store\_value，此值取值范围为0到32，包含0，32，数据库创建时指定，创建成功后不能被修改。

当兼容模式被设置为Oracle或Mysql，并且 lightdb\_ascii\_zero\_store\_value 设置为非零ASCII码值，ASCII 码值 E'\0' 以及 CHR(0) 的值等于 ASCII 码值 lightdb\_ascii\_zero\_store\_value，如果我们通过JDBC的绑定参数与服务端交互，文本

类型的参数中ASCII码值 '\0' 由 ASCII 码值 `lightdb_ascii_zero_store_value` 在服务器端被替换, Libpq同样场景下, 值替换发生在Libpq这一端。也请注意, 如果ASCII码值 '\0' 由 ASCII 码值 `lightdb_ascii_zero_store_value` 替换, 替换后此值在系统中的行为与 ASCII 码值 `lightdb_ascii_zero_store_value` 相同, 比如, 比较操作, `concat` 函数调用, `length`函数调用, 等其它接受字符输入参数的函数。

`lightdb_convert_chinese_char` (boolean)

在Oracle兼容模式下, 将SQL语句中的中文逗号和中文空格, 转换成英文逗号和英文空格。这个参数的值为true或者false, 默认值是true。true: 开启 false: 不开启

`lightdb_ignore_template_compatible_type_mismatch` (boolean)

该参数从 Lightdb 23.4 开始引入, 用于决定如何处理 CREATE DATABASE 中 TEMPLATE 选项和 LIGHTDB\_SYNTAX\_COMPATIBLE\_TYPE 选项不兼容的情况。当两个选项不兼容时, 如果该 GUC 参数的值为 false, 则提示错误, 否则忽略 TEMPLATE 选项。这个参数的值为true或者false, 默认值是false。

## 17.11.2. 区域和格式化

`DateStyle` (string)

设置日期和时间值的显示格式, 以及解释有歧义的时间输入值的规则。由于历史原因, 这个变量包含两个独立的部分: 输出格式声明 (ISO、Postgres、SQL或German)、输入/输出的年/月/日顺序 (DMY、MDY或YMD)。这些可以被独立设置或者一起设置。关键字Euro和European是DMY的同义词; 关键字US、NonEuro和NonEuropean是MDY的同义词。详见Section 8.5 内建默认值是ISO, MDY, 但是initdb将用对应于选中的lc\_time区域行为的设置初始化配置文件。

`IntervalStyle` (enum)

设置间隔值的显示格式。值`sql_standard`将产生匹配 SQL 标准间隔文本的输出。当`DateStyle`参数被设置为ISO时, 值`postgres` (默认) 将产生匹配LightDB发行 8.4 之前的输出。当`DateStyle`参数被设置为非ISO输出时, 值`postgres_verbose`会产生匹配 LightDB发行 8.4 之前的输出。值`iso_8601`会产生匹配在 ISO 8601 的 4.4.3.2 节中定义的“带标志符格式”的时间间隔的输出。

`IntervalStyle`参数也可以影响对有歧义的时间间隔的解释。详见Section 8.5.4

`TimeZone` (string)

设置用于显示和解释时间戳的时区。内建默认值是GMT, 但是它通常会在`lightdb.conf`中被覆盖; `initdb`将安装一个对应于其系统环境的设置。详见Section 8.5.3

`timezone_abbreviations` (string)

设置服务器接受的日期时间输入中使用的时区缩写集合。默认值为'Default', 这个集合在全世界大多数地方都能工作。也还有'Australia'和'India', 以及可能为一种特定安装定义的其他集合。详见Section D.4

`extra_float_digits` (integer)

这个参数调整用于文本输出浮点值的位数, 包括float4, float8以及几何数据类型。

如果值为1(默认值)或更高, 浮点值被输出为最短-精度格式; 参见Section 8.1.3 实际生成的位数只取决于输出的值, 而不取决于此参数的值。float8 值最多需要 17 位数字, float4值最多需要9位数字。这种格式既快速又精确, 在正确读取时精确地保留了原始的二进制浮点值。为了历史兼容性, 允许的值最大为3。

如果值为零或负, 则输出四舍五入为给定的十进制精度。使用的精度是根据此参数的值减小的类型 (FLT\_DIG或DBL\_DIG, 视情况而定) 的标准位数。(例如, 指定 -1 将导

致float4 值输出四舍五入为 5 位有效数字，而float8值四舍五入为 14 位。) 此格式较慢，不会保留二进制浮点值的所有位，但可能令人更易于阅读。

### Note

此参数的含义，以及其默认值，在 LightDB 12 中发生了变化；参见 Section 8.1.3以便进一步讨论。

#### client\_encoding (string)

设置客户端编码（字符集）。默认使用数据库编码。LightDB服务器所支持的字符集在Section 21.3. 中描述。

#### lc\_messages (string)

设置消息显示的语言。可接受的值是系统相关的；详见Section 21.1 如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

在一些系统上，这个区域分类并不存在。仍然可以设置这个变量，只是不会有任何效果。同样，所期望语言的翻译消息也可能不存在。在这种情况下，你将仍然继续看到英文消息。

只有超级用户可以改变这个设置。因为它同时影响发送到服务器日志和客户端的消息。一个不正确的值可能会降低服务器日志的可读性。

#### lc\_monetary (string)

设置用于格式化货币量的区域，例如用to\_char函数族。可接受的值是系统相关的；详见Section 21.1 如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

#### lc\_numeric (string)

设置用于格式化数字的区域，例如用to\_char函数族。可接受的值是系统相关的；详见Section 21.1 如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

#### lc\_time (string)

设置用于格式化日期和时间的区域，例如用to\_char函数族。可接受的值是系统相关的；详见Section 21.1 如果这个变量被设置为空字符串（默认），那么该值将以一种系统相关的方式从服务器的执行环境中继承。

#### default\_text\_search\_config (string)

选择被那些没有显式参数指定配置的文本搜索函数变体使用的文本搜索配置。详见Chapter 12 内建默认值是pg\_catalog.simple，但是如果能够标识一个匹配区域的配置，initdb将用对应于选中的lc\_ctype区域的设置初始化配置文件。

#### lightdb\_tsearch\_non\_stopwords (string)

LightDB全文搜索有许多内置的停止词。由停止词连接的单词将通过to\_tsvector函数拆分为多个独立的单词，例如，“2022-04-10”将拆分为“2022”、“04”和“10”。lightdb\_tsearch\_non\_stopwords用于自定义不间断词，这与停止词正好相反，即不间断词被认为是单词的一个组成部分，因此不再被to\_tsvector函数拆分为多个独立的单词。还是上面的例子，如果你将“-”设置为不间断的单词，你只会得到“2022-04-10”的结果。以下字符可以用作LightDB中的非停止单词。从以下字符中自由选择，以提高其准确性和易于使用全文SERACH。受支持的非停止词列表：

```
! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~
```

详见Chapter 12

内置的默认值是''，您可以在 `lightdb.conf` 中更新`lightdb_tsearch_non_stopwords`值，需要重启后生效。您还可以在会话期间更新`lightdb_tsearch_non_stopwords`，可以无需重新启动服务器。从非停止词列表中选择任意字符组合，例如：

```
lightdb@postgres=# set lightdb_tsearch_non_stopwords = '@-:~';
SET
lightdb@postgres=# select to_tsvector('this is from lt_hs_tab table
logged at 2020-12-21 00:22:32, username=zjh&pwd=balabala www.hs.net');
to_tsvector
```

```
-----
'00:22:32':9 '2020-12-21':8 'log':6 'lt_hs_tab':4 'pwd=balabala':11
'tabl':5 'username=zjh':10 'www.hs.net':12
(1 row)
```

```
lightdb@postgres=# set lightdb_tsearch_non_stopwords = '';
SET
lightdb@postgres=# select to_tsvector('this is from lt_hs_tab table
logged at 2020-12-21 00:22:32, username=zjh&pwd=balabala www.hs.net');

to_tsvector
```

```
-----
'-12':11 '-21':12 '00':13 '2020':10 '22':14 '32':15 'balabala':19
'hs':5 'log':8 'lt':4 'pwd':18 'tab':6 'tabl':7 'usernam':16 'www.hs.net':20
'zjh':17
(1 row)
```

如果`lightdb_tsearch_non_stopwords` parameter是无效值会报错。比如：

```
lightdb@postgres=# set lightdb_tsearch_non_stopwords = '1@';
ERROR:  invalid value for parameter "lightdb_tsearch_non_stopwords":
"1@"
DETAIL:  lightdb_tsearch_non_stopwords not support '1@' as non
stopword.
```

`lightdb_tsearch_word_superpose` (boolean)

`lightdb_tsearch_word_superpose`用于叠加使用停止词和非停止词的效果。例如，如果将`lightdb_tsearch_word_superpose`设置为on，并将`lightdb_tsearch_non_stopwords`设置为'-，则在使用`to_tsvector("2022-04-10")`时，将同时获得“2022”、“04”、“10”和“2022-04-10”两种结果。详见Chapter 12

内置的默认值是off，你可以在`lightdb.conf`中更新`lightdb_tsearch_word_superpose`的值，需要重启后生效。你也可以在会话期间更新`lightdb_tsearch_word_superpose`则无需重启。不支持中文文字。比如：



```

lightdb@postgres=# set lightdb_tsearch_word_superpose = off;
SET
lightdb@postgres=# select to_tsvector('this is from lt_hs_tab table
logged at 2020-12-21 00:22:32, username=zjh&pwd=balabala www.hs.net');

to_tsvector
-----
'-12':11 '-21':12 '00':13 '2020':10 '22':14 '32':15 'balabala':19
'hs':5 'log':8 'lt':4 'pwd':18 'tab':6 'tabl':7 'usernam':16 'www.hs.net':20
'zjh':17
(1 row)

lightdb@postgres=# set lightdb_tsearch_word_superpose = on;
SET
lightdb@postgres=# select to_tsvector('this is from lt_hs_tab table
logged at 2020-12-21 00:22:32, username=zjh&pwd=balabala www.hs.net');

to_tsvector
-----
'-12':33 '-21':34 '00':13,35 '12':11 '2020':10,32 '21':12 '22':14,36
'32':15,37 'balabala':19,41 'hs':5,21,27 'log':8,30 'lt':4,26 'net':22
'pwd':18,40 'tab':6,28 'tabl':7,29 'usernam':16
,38 'www':20 'www.hs.net':42 'zjh':17,39
(1 row)

```

lightdb\_encryption\_key\_command (string)

用于存储透明数据加密的密钥路径，该路径可确保每次重新启动数据后再次读取TDE密钥。

详见Chapter 31

When Transparent Data Encryption is not used, this parameter is null. Only when Transparent Data Encryption is configured, this parameter stores the tde key. 如果未使用透明数据加密，则该参数为NULL。只有配置后该参数才保存TDE密钥。设置示例如下：

```

show lightdb_encryption_key_command;
lightdb_encryption_key_command
-----
/home/lightdb/stage/provide_key.sh
(1 row)

```

nls\_date\_format (string)

只有在Oracle兼容模式下，lightdb才可以配置nls\_date\_format来设置sysdate 和 timestamp的显示输出格式

详见 Section 9.9

必须提前为`nls_date_format`变量设置系统日期格式。 设置示例如下：

```
SET nls_date_format = 'YYYY/MM/DD HH24:MI:SS';
```

```
select sysdate;
       sysdate
-----
2022/09/15 10:03:29
(1 row)
```

```
select to_char(sysdate);
       to_char
-----
2022/09/15 10:08:46
(1 row)
```

`nls_timestamp_format` (string)

只有在Oracle兼容模式下，`lightdb`才可以配置`nls_timestamp_format`来设置`systemtimestamp` 和 `timestampz`的显示输出格式

详见 Section 9.9

您必须提前为`nls_timestamp_format`变量设置系统时间戳格式。 设置示例如下：

```
SET nls_timestamp_format = 'YYYY/MM/DD HH24:MI:SS.FF6TZH';
```

```
select systemtimestamp;
       systemtimestamp
-----
2022/09/15 10:05:56.884814+08
(1 row)
```

`lightdb_only_full_group_by` (boolean)

仅在MySQL兼容模式下，此参数用于控制mysql的SELECT DISTINCT...ORDER BY语法是否与ONLY\_FULL\_GROUP\_BY以外的mysql语法兼容。 此参数默认值为on。

默认情况下，`lightdb`将报告SELECT DISTINCT...ORDER BY语法中不一致列的错误：

```
select distinct a from test_distinct_order_tbl order by d;
ERROR: for SELECT DISTINCT, ORDER BY expressions must appear in select list
LINE 1: select distinct a from test_distinct_order_tbl order by d;
```

只有在MySQL兼容模式和`lightdb_only_full_group_by`设置off时，上述SQL才能正常执行：

```
set lightdb_only_full_group_by = off;
SET

select distinct a from test_distinct_order_tbl order by d;
       a
-----
```

```

3
1
2
(3 rows)

```

请注意，这种行为只是一种与MySQL兼容的非标准语法。它用于确保LightDB中的执行不会报告错误，但不能保证结果与MySQL完全一致。不建议使用此功能。请使用标准语法格式。

### 17.11.3. 共享库预载入

为了载入附加的功能或者达到提高性能的目的，可用多个设置来预先载入共享库到服务器中。例如，`'$libdir/mylib'` 设置可能会导致`mylib.so`（或者某些平台上的`mylib.sl`）从安装的标准库目录被预装载。这些设置之间的区别在于生效的时间以及改变它们所需的特权。

可以用这个方法预装载LightDB的过程语言库，通常是使用`'$libdir/p1XXX'` 语法，其中的XXX是`pgsql`、`plorasql`。

只有特别为与LightDB一起使用设计的共享库才能以这种方式载入。每一个LightDB支持的库都有一个“魔法块”，它会被检查以保证兼容性。由于这个原因，非LightDB无法以这种方式被载入。你可能可以使用操作系统的工具（如`LD_PRELOAD`）载入它。

总之，请参考特定模块的文档来用推荐的方法载入它。

`local_preload_libraries` (string)

这个变量指定一个或者多个要在连接开始时预载入的共享库。它包含一个由逗号分隔的库名列表，其中每个名称都会按LOAD命令的方式解析。项之间的空格会被忽略，如果需要在库名中包含空格或者逗号，请把库名放在双引号内。这个参数值只在连接开始时生效。后续的更改不会有任何效果。如果一个指定的库没有找到，连接尝试将会失败。

任何用户都能设置这个选项。正因为如此，能被这样载入的库被严格限制为出现于安装的标准库目录中`plugins`子目录下的共享库（保证只有“安全的”库被安装到这里是数据库管理员的责任）。`local_preload_libraries`中的项可以显式指定这个目录，例如`'$libdir/plugins/mylib'`，或者只是指定库的名称 — `mylib` 和 `'$libdir/plugins/mylib'`的效果是相同的。

这个特性的目的是允许非特权用户在特定的会话中载入正在调试的或者性能度量库，而无需一个显式的LOAD命令。为了这个目的，通常通过使用客户端的`PGOPTIONS`环境变量或者 `ALTER ROLE SET`来设置这个参数。

不过，除非一个模块被特别设计成由非超级用户以这种方式使用，通常不推荐使用这个设置。应该看看 `session_preload_libraries`。

`session_preload_libraries` (string)

这个变量指定一个或者多个要在连接开始时预载入的共享库。只有超级用户能够更改这个设置。它包含一个由逗号分隔的库名列表，其中每个名称都会按LOAD命令的方式解析。项之间的空格会被忽略，如果需要在库名中包含空格或者逗号，请把库名放在双引号内。这个参数只在连接开始时起效。后续的改变没有效果。如果指定的库没有找到，连接尝试将会失败。只有超级用户能够更改这个设置。

这个特性的意图是允许在特定会话中载入调试用的或者测量性能的库，而不需要显式的给出一个LOAD命令。例如，通过用`ALTER ROLE SET`设置这个参数可以为一个给定用户名下的所有会话启用`auto_explain`。还有，无需重启服务器就能更改这个参数（但是只有新会话启动时才会生效），这样可以以这种方式更容易地增加新模块，即便它们会应用到所有会话。

和`shared_preload_libraries`不同，相对于在库被第一次使用时载入它，在会话开始时载入库并没有什么性能优势。不过，当使用连接池时这样做还是有一些优势。

`shared_preload_libraries` (string)

这个变量指定一个或者多个要在服务器启动时预载入的共享库。它包含一个由逗号分隔的库名列表，其中每个名称都会按LOAD命令的方式解析。项之间的空格会被忽略，如果需要在库名中包含空格或者逗号，请把库名放在双引号内。这个参数只能在服务器启动时设置。如果指定的库没有找到，服务器将无法启动。

有些库需要执行只能在postmaster启动时发生的特定操作，例如分配共享内存、保留轻量级锁或者启动后台工作者。这些库必须通过这个参数在服务器启动时载入。每个库的详情请见文档。

其他库也能被预载入。通过预载入一个共享库，当该库被第一次使用时就可以避免库的启动时间。不过，启动每个新服务器进程的时间可能会略有增加，即使该进程从不使用该库。因此，推荐只把这个参数用于那些要在大多数会话中使用的库上。还有，改变这个参数要求重启服务器，因此对于短期的调试任务来说这不是好的选择，应该转用`session_preload_libraries`。

### Note

在 Windows 主机上，在服务器启动时预载入一个库并不会减少启动每个新服务器进程所需的时间；每一个服务器进程将会重新载入预载入的库。不过，对于那些要在postmaster启动时执行操作的库来说，Windows 主机上的 `shared_preload_libraries`任然有用。

`jit_provider` (string)

这个变量是要被使用的JIT提供者库的名称（见Section 29.4.2。默认是`llvmjit`。这个参数只能在服务器启动时设置。

如果这个变量被设置为一个不存在的库，JIT将不可用，但是也不会发生错误。这种特性允许在主LightDB包之外单独安装JIT支持。

## 17.11.4. 其他默认值

`dynamic_library_path` (string)

如果需要打开一个可以动态装载的模块并且在CREATE FUNCTION或LOAD命令中指定的文件名没有目录部分（即名字中不包含斜线），那么系统将搜索这个路径以查找所需的文件。

`dynamic_library_path`的值必须是一个冒号分隔（或者在 Windows 上以分号分隔）的绝对目录路径的列表。如果一个列表元素以特殊字符串开始，`$libdir`会被替换为LightDB包中已编译好的库目录。这里是LightDB发布提供的模块被安装的位置（使用`lt_config --pkglibdir`来找到这个目录的名字）。例如：

```
dynamic_library_path = '/usr/local/lib/lightdb:/home/my_project/lib:$libdir'
```

这个参数的默认值是'`$libdir`'。如果该值被设置为一个空字符串，则关闭自动路径搜索。

这个参数可以在运行时由超级用户修改，但是这样修改的设置只能保持到这个客户端连接的结尾，因此这个方法应该保留给开发目的。我们建议在`lightdb.conf`配置文件中设置这个参数。

`gin_fuzzy_search_limit` (integer)

GIN 索引返回的集合尺寸的软上限。详见Section 62.5

## 17.12. 锁管理

`deadlock_timeout` (integer)

这是进行死锁检测之前在一个锁上等待的时间量。死锁检测相对昂贵，因此服务器不会在每次等待锁时都运行这个它。我们乐观地假设在生产应用中死锁是不常出现的，并且只在开始检测死锁之前等待一会儿。增加这个值就减少了浪费在无用的死锁检测上的时间，但是减慢了报告真正死锁错误的速度。如果指定值时没有单位，则以毫秒为单位。默认是 1 秒 (1s)，这可能是实际中你想要的最小值。在一个高负载的服务器上，你可能需要增大它。这个值的理想设置应该超过你通常的事务时间，这样就可以减少在锁释放之前就开始死锁检查的机会。只有超级用户可以更改这个设置。

当`log_lock_waits`被设置时，这个参数还可以决定发出关于锁等待的日志之前等待的时间量。如果你想调查锁延迟，你可能希望设置一个比正常的`deadlock_timeout`小的值。

`max_locks_per_transaction` (integer)

共享锁表跟踪在`max_locks_per_transaction * (max_connections + max_prepared_transactions)` 个对象（如表）上的锁。因此，在任何一个时刻，只有不超过这么多个可区分对象能够被锁住。这个参数控制为每个事务分配的对象锁的平均数量。个体事务可以锁住更多对象，数量可以和锁表中能容纳的所有事务的锁一样多。这不是能被锁住的行数，那个值是没有限制的。默认值 64 已经被历史证明是足够的，但是如果你有需要在一个事务中使用很多不同表的查询（例如查询一个有很多子表的父表），你可能需要提高这个值。这个参数只能在服务器启动时设置。

当运行一个后备服务器时，你必须设置这个参数为大于等于主服务器上的值。否则，后备服务器上将不允许查询。

`max_pred_locks_per_transaction` (integer)

共享谓词锁表跟踪在`max_pred_locks_per_transaction * (max_connections + max_prepared_transactions)` 个对象（如表）上的锁。因此，在任何一个时刻，只有不超过这么多个可区分对象能够被锁住。这个参数控制为每个事务分配的对象锁的平均数量。个体事务可以锁住更多对象，数量可以和锁表中能容纳的所有事务的锁一样多。这不是能被锁住的行数，那个值是没有限制的。默认值 64 已经在测试中被证明通常是足够的，但是如果你有需要在一个可序列化事务中使用很多不同表的查询（例如查询一个有很多子表的父表），你可能需要提高这个值。这个参数只能在服务器启动时设置。

`max_pred_locks_per_relation` (integer)

这个参数控制在谓词锁被提升为覆盖整个关系之前，该谓词锁能够在单个关系上锁住多少页面或元组。大于等于零的值表示一种绝对限制，而负值表示用`max_pred_locks_per_transaction`除以这个设置的绝对值。默认值为-2，它将保持以往版本的LightDB中的行为。这个参数只能在`lightdb.conf`文件中或者服务器命令行上设置。

`max_pred_locks_per_page` (integer)

这个参数控制在谓词锁被提升为覆盖整个页面之前，该谓词锁能在单一页面上锁住多少行。默认值是2。这个参数只能在`lightdb.conf`文件中或者服务器命令行上设置。

## 17.13. 版本和平台兼容性

### 17.13.1. 以前的 LightDB 版本

`array_nulls` (boolean)

这个参数控制数组输入解析器是否把未用引号的NULL识别为一个空数组元素。默认为on，允许输入包含空值的数组值。但是LightDB 8.2 之前的版本不支持数组中的空

值，并且因此将把NULL当作指定一个值为字符串“NULL”的正常数组元素。对于那些要求旧行为的应用的向后兼容性，这个变量可以被设置为off。

注意即使这个变量为off也能够创建包含空值的数组值。

backslash\_quote (enum)

这个参数控制字符串文本中的单引号是否能够用\来表示。首选的 SQL 标准的方法是将其双写（''），但是LightDB在历史上也接受\。不过使用\容易导致安全风险，因为在某些客户端字符集编码中，有多字节字符的最后一个字节在数值上等价于 ASCII 的\。如果客户端代码没有做到正确转义，那么将会导致 SQL 注入攻击。如果服务器拒绝看起来带有被反斜线转义的单引号的查询，那么就可以避免这种风险。backslash\_quote的可用值是on（总是允许\）、off（总是拒绝）以及safe\_encoding（只有客户端编码不允许在多字节字符中存在 ASCII \时允许）。safe\_encoding是默认设置。

注意在符合标准的字符串文本中，\就表示\。这个参数只影响不符合标准的文本的处理，包括转义字符串语法（E'...'）。

escape\_string\_warning (boolean)

打开时，如果在普通字符串文本中（'...'语法）出现了一个反斜线（\）并且standard\_conforming\_strings为关闭，那么就会发出一个警告。默认值是on。

希望使用反斜线作为转义符的应用应该被修改来使用转义字符串语法（E'...'），因为在 SQL 标准中普通字符串的默认行为是将反斜线视作一个普通字符。这个变量可以被启用来帮助定位需要被更改的代码。

lo\_compat\_privileges (boolean)

默认是off。只有超级用户可以更改这个设置。

设置这个参数不会禁用所有与大对象相关的安全检查 — 除了那些在LightDB 9.0中已经修改了的默认行为。

operator\_precedence\_warning (boolean)

启用该选项后，解析器将对任何可能由于运算符优先级的更改而在LightDB9.4之后更改了含义的构造发出警告。这对于审核应用程序以查看优先级更改是否破坏了任何内容非常有用；但它并不意味着在生产中一直处于打开状态，因为它会警告一些完全有效、符合标准的SQL代码。默认值为off。

See Section 4.1.6 for more information.

quote\_all\_identifiers (boolean)

当数据库产生 SQL 时，强制所有标识符被引号包围，即使它们（当前）不是关键字。这将影响EXPLAIN的输出以及pg\_get\_viewdef等函数的结果。另请参阅lt\_dump和lt\_dumpall的--quote-all-identifiers选项。

standard\_conforming\_strings (boolean)

控制普通字符串文本（'...'）是否按照 SQL 标准把反斜线当普通文本。从LightDB 9.1 开始，默认值为on（之前的发行中默认值为off）。应用可以检查这个参数来判断字符串文本如何被处理。这个参数的存在也可以被当做转义字符串语法（E'...'）被支持的标志。如果一个应用希望反斜线被当做转义字符，应该使用转义字符串语法（Section 4.1.2.2）。

synchronize\_seqscans (boolean)

它允许对大型表的顺序扫描与其他扫描同步，因此并发扫描可以在几乎相同的时刻读取相同的块，这样可以分担 I/O 负载。当启用这个参数时，一个扫描可能会从表的中间开始并且之后“绕回”到开头以覆盖所有的行，这样可以与已在进行中的扫描活动同步。对于没有ORDER BY子句的查询来，这样的扫描会在返回行的顺序中造成不可预料的改

变。将这个参数设置为off以保证 8.3 之前的行为（顺序扫描总是从表的起始处开始）。默认值是on。

## 17.13.2. 平台和客户端兼容性

`transform_null_equals` (boolean)

当打开且`lightdb_dblevel_syntax_compatible_type`关闭时，形为`expr = NULL`（或`NULL = expr`）的表达式将被当做`expr IS NULL`，也就是说，如果`expr`得出空值则返回真，否则返回假。正确的 SQL 标准兼容的`expr = NULL`行为总是返回空（未知）。这个参数默认为on。

不过，在Microsoft Access里的过滤表单生成的查询似乎使用`expr = NULL`来测试空值，因此，如果你使用这个接口访问数据库，你可能想把这个选项打开。因为`expr = NULL`形式的表达式总是返回空值（使用 SQL 标准解释）。它们不是非常有用并且在普通应用中也不常见，在应用中也不常见，因此这个选项实际上没有什么危害。但是新用户常常对涉及空值的表达式的语义上感到困惑，因此这个选项默认为关闭。

请注意这个选项只影响`= NULL`形式，而不影响其它比较操作符或者其它与一些涉及等值操作符的表达式在计算上等效的其他表达式（例如IN）。因此，这个选项不是劣质程序的一般修复。

相关信息请见Section 9.2

## 17.14. 错误处理

`exit_on_error` (boolean)

如果为开(on)，任何错误将中止当前会话。默认情况下，这个值被设置为关(off)，这样只有 FATAL 错误（致命）将中止会话。

`restart_after_crash` (boolean)

当被设置为开(on)，LightDB将在一次后端崩溃后自动重新初始化。让这个值设置为开(on)通常是将数据库可用性最大化的最佳方法。但是在某些环境中，例如LightDB被集群软件调用时，禁用重启可能很有用，这样集群软件可以得到控制并且采取它认为适当的行动。

这个参数只能在`lightdb.conf`文件中或服务器命令行中设置。

`data_sync_retry` (boolean)

如果设置为关(off)，LightDB在将修改的数据文件刷新到文件系统失败时，将引发PANIC级错误。这样会导致数据库服务器崩溃。这个参数只能在服务器启动时被设置。

在某些操作系统上，回写失败后，内核页面缓存中的数据状态未知。在某些情况下，它可能已被完全遗忘，因此重试不安全；第二次尝试可能报告为成功，而事实上数据已丢失。在此类情形下，避免数据丢失的唯一方法是在报告任何故障后从WAL中恢复，最好是在调查了故障的根本原因并更换了任何有故障的硬件之后。

如果设置为开(on)，LightDB将报告错误，但会继续运行，以便可以在以后的检查点中重试数据刷新操作。仅在调查操作系统假如回写失败时对缓冲数据的处理方式的情况下，才将其设置为开(on)。

## 17.15. 预置选项

下列“参数”是只读的。同样，它们被排除在`lightdb.conf`文件例子之外。这些选项报告特定应用可能感兴趣的多种LightDB行为，特别是管理前端相关的行为。所有这些都是LightDB被编译或者它被安装时决定的。

block\_size (integer)

报告一个磁盘块的大小。它由编译服务器时BLCKSZ的值确定。默认值是 8192 字节。有些配置变量的含义（例如shared\_buffers）会被block\_size影响。详见Section 17.4

data\_checksums (boolean)

报告对这个集簇是否启用了数据校验码。详见data\_checksums。

data\_directory\_mode (integer)

在Unix系统上，这个参数报告启动时的数据目录(由data\_directory定义)的权限。（在Microsoft Windows上这个参数将总是显示0700。）更多信息请参考group access。

debug\_assertions (boolean)

报告编译LightDB时是否启用了断言。如果LightDB被编译时定义了宏USE\_ASSERT\_CHECKING is defined when LightDB（例如通过 configure选项 --enable-cassert定义），那么会报告已启用。默认情况下 LightDB编译时没有用断言。

integer\_datetimes (boolean)

报告LightDB是否在编译时打开了 64 位整数日期和时间。从LightDB 10起，这个值总是on。

lc\_collate (string)

报告文本数据排序使用的区域。详见Section 21.1 该值是在数据库被创建时确定的。

lc\_ctype (string)

报告决定字符分类的区域。详见Section 21.1 该值是在数据库被创建时决定的。通常它和lc\_collate一样，但是可以为特殊应用设置成不同的值。

max\_function\_args (integer)

报告函数参数的最大数量。它由编译服务器时的FUNC\_MAX\_ARGS值决定的。默认值是 100 个参数。

max\_identifier\_length (integer)

报告标识符的最大长度。它由编译服务器时的NAMEDATALEN值减一决定。NAMEDATALEN的默认值是 64 ；因此max\_identifier\_length的默认值是 63，但是在使用多字节编码时可以少于 63 个字符。

max\_index\_keys (integer)

报告索引键的最大数目。它由编译服务器时的INDEX\_MAX\_KEYS值决定。默认值是 32 个键。

segment\_size (integer)

报告一个文件段中可以存储的块（页）的数量。由编译服务器时的RELSEG\_SIZE值决定。一个段文件的最大尺寸（以字节计）等于segment\_size乘以block\_size，默认是 1GB。

server\_encoding (string)

报告数据库的编码（字符集）。这是在数据库被创建时决定的。通常，客户端只需要关心client\_encoding的值。

server\_version (string)

报告服务器版本数值。它是由编译服务器时的PG\_VERSION值决定的。



`server_version_num` (integer)

报告服务器版本数值的整数值。它是由编译服务器时的PG\_VERSION\_NUM值决定的。

`ssl_library` (string)

报告此LightDB服务器已构建的 SSL 库的名称（即使此实例当前未配置或使用 SSL），例如OpenSSL，或一个空字符串(如果没有)。

`wal_block_size` (integer)

报告一个 WAL 磁盘块的大小。由编译服务器时的XLOG\_BLCKSZ值决定。默认是 8192 字节。

`wal_segment_size` (integer)

报告 WAL 段文件的大小。默认是 16MB。详见Section 27.4

## 17.16. 自定义选项

这个特性被设计用来由附加模块向LightDB添加通常不为系统知道的参数（例如过程语言）。这允许使用标准方法配制扩展模块。

自定义选项有两部分名称：一个扩展名，然后是一个句点，再然后是正确的参数名，就像SQL中的合格名称。一个例子是`plpgsql.variable_conflict`。

因为自定义选项可能需要在没有载入相关扩展模块的进程中设置，LightDB将接收任意两部分参数名的设置。这种变量被认为是占位符并且在定义它们的模块被载入之前不会有实际功能。当一个扩展模块被载入，它将加入它的变量定义、根据那些定义转换任何占位符值并且对以其扩展名开始的任意未识别占位符发出警告。

## 17.17. 开发者选项

下面的参数目的是用在开发测试上，并且永远不能用于生产数据库。但是，它们中的一些能够用于帮助恢复严重损坏的数据库。同样，它们被从例子`lightdb.conf`文件中排除。请注意许多这些参数要求特殊的源代码编译标志才能工作。

`allow_in_place_tablespace` (boolean)

当向CREATE TABLESPACE命令提供空位置字符串时，允许将表空间创建为`lt_tblspc`内的目录。这是为了允许测试主服务器和备用服务器在同一台计算机上运行的复制场景。这样的目录可能会混淆备份工具，因为备份工具希望在该位置只找到符号链接。只有超级用户才能更改此设置。

`allow_system_table_mods` (boolean)

允许对系统表结构的修改，以及对系统表的某些其他危险操作。这是不允许的，即使是对超级用户。不理智地使用此设置可能导致无法恢复的数据丢失或严重损坏数据库系统。只有超级用户才能更改此设置。

`backtrace_functions` (string)

这个参数包含一个以逗号分隔的C函数名称列表。如果出现错误，并且发生错误的内部C函数的名称与列表中的值相匹配，那么将向服务器日志中写入一个回溯跟踪信息和错误消息。这可以用来调试源代码的特定区域。

回溯支持并非在所有平台上都可以使用，并且回溯的质量取决于编译选项。

该参数只能由超级用户设置。

`ignore_system_indexes` (boolean)

读取系统表时忽略系统索引（但是修改系统表时依然同时更新索引）。这在从被破坏的系统索引中恢复数据的时有用。这个参数在会话开始之后不能被更改。

`post_auth_delay` (integer)

执行身份验证过程后启动新服务器进程时延迟的时间量。这是为了给开发者们一个机会在一个服务器进程上附加一个调试器。如果指定值时没有单位，则以秒为单位。0值(默认值)禁用延迟。这个参数在会话开始之后不能被更改。

`pre_auth_delay` (integer)

在新服务器进程分叉后，在进行身份验证过程之前，延迟的时间量。这是为了给开发者们一个机会在一个服务器进程上附加一个调试器来跟踪认证过程中的不当行为。如果指定值时没有单位，则以秒为单位。0值(默认值)禁用延迟。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

`trace_notify` (boolean)

为LISTEN和NOTIFY命令生成大量调试输出。`client_min_messages`和`log_min_messages`必须是DEBUG1或者更低才能把这种输出分别发送到客户端或者服务器日志。

`trace_recovery_messages` (enum)

启用记录与恢复有关的调试输出，否则无法记录。这个参数允许用户覆盖`log_min_messages`的正常设置，但只用于指定的消息。这个参数的目的是用来调试热后备。有效值包括DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1和LOG。默认值LOG完全不会影响日志决定。其他值会记录相关级别或更高级别的与恢复相关的调试消息，就好像它们具有LOG优先级一样；对于`log_min_messages`的通用设置，这会无条件的将消息发送给服务器日志。这个参数只能在`lightdb.conf`文件中或在服务器命令行上设置。

`trace_sort` (boolean)

如果打开，发出在排序操作中的资源使用的相关信息。只有在编译LightDB时定义了TRACE\_SORT宏，这个参数才可用（不过，当前在默认情况下就定义了TRACE\_SORT）。

`trace_locks` (boolean)

如果开启，发出锁使用情况的信息。被转储信息中包括锁操作的类型、锁的类型和 被锁或被解锁对象的唯一标识符。同样包括的还有已经授予这个对象的锁类型的位掩码和 等待这个对象的锁类型的位掩码。对每一种锁类型，已授权锁和等待锁的计数也会被一起转储。一个日志文件输出的例子如下：

```
LOG: LockAcquire: new: lock(0xb7acd844) id(24688, 24696, 0, 0, 0, 1)
      grantMask(0) req(0, 0, 0, 0, 0, 0)=0 grant(0, 0, 0, 0, 0, 0)=0
      wait(0) type(AccessShareLock)
LOG: GrantLock: lock(0xb7acd844) id(24688, 24696, 0, 0, 0, 1)
      grantMask(2) req(1, 0, 0, 0, 0, 0)=1 grant(1, 0, 0, 0, 0, 0)=1
      wait(0) type(AccessShareLock)
LOG: UnGrantLock: updated: lock(0xb7acd844) id(24688, 24696, 0, 0, 0, 1)
      grantMask(0) req(0, 0, 0, 0, 0, 0)=0 grant(0, 0, 0, 0, 0, 0)=0
      wait(0) type(AccessShareLock)
LOG: CleanUpLock: deleting: lock(0xb7acd844) id(24688, 24696, 0, 0, 0, 1)
      grantMask(0) req(0, 0, 0, 0, 0, 0)=0 grant(0, 0, 0, 0, 0, 0)=0
      wait(0) type(INVALID)
```

被转储结构的详细信息可以在`src/include/storage/lock.h`中找到。

只有在编译LightDB时定义了LOCK\_DEBUG宏，这个参数才可用。

`trace_lwlocks` (boolean)

如果开启，发出轻量级锁的使用信息。轻量级锁主要是为了提供对共享内存数据结构的互斥访问。

只有在编译LightDB时定义了LOCK\_DEBUG宏，这个参数才可用。

`trace_userlocks` (boolean)

如果开启，发出关于用户锁使用的信息。与`trace_locks`的输出一样，但只用于咨询锁。

只有在编译LightDB时定义了LOCK\_DEBUG宏，这个参数才可用。

`trace_lock_oidmin` (integer)

如果设置，不会跟踪小于这个OID的锁（用于避免在系统表上的输出）。

只有在编译LightDB时定义了LOCK\_DEBUG宏，这个参数才可用。

`trace_lock_table` (integer)

无条件地跟踪此表（OID）上的锁。

只有在编译LightDB时定义了LOCK\_DEBUG宏，这个参数才可用。

`debug_deadlocks` (boolean)

如果设置，当死锁超时发生时，转储所有当前锁的信息。

只有在编译LightDB时定义了LOCK\_DEBUG宏，这个参数才可用。

`log_btree_build_stats` (boolean)

如果设置，会记录B树操作上的系统资源使用情况统计（内存和CPU）。

只有在编译LightDB时定义了BTREE\_BUILD\_STATS宏，这个参数才可用。

`wal_consistency_checking` (string)

这个参数被设计用来检查WAL重做例程中的缺陷。当这个参数被启用时，被修改的任何缓冲区的全页映像及其WAL记录都被加入到记录中。如果该记录后来被重放，系统将首先应用每个记录然后测试该记录修改的缓冲区是否符合存储的映像。在某些情况下（例如提示位），小的变动是可以接受的，并且会被忽略。任何预期之外的差别都将导致致命错误，最后中止恢复。

这个设置的默认值是空字符串，它将禁用这一特性。它可以被设置为`all`以检查所有记录，或者被设置为一个逗号分隔的资源管理器列表用以检查那些资源管理器产生的记录。当前，支持的资源管理器是`heap`、`heap2`、`btree`、`hash`、`gist`、`gin`、`sequence`以及`generic`。只有超级用户可以更改这一设置。

`wal_debug` (boolean)

如果被打开，WAL相关的调试输出将被发出。只有在编译LightDB时定义了WAL\_DEBUG宏的情况下，这个参数才可用。

`ignore_checksum_failure` (boolean)

只有当`data checksums`被启用时才有效。

在读取过程中检测到一次校验码失败通常会导致LightDB报告一个错误。设置`ignore_checksum_failure`为打开会导致系统忽略失败（但是仍然报告一个警告），并且继续执行。这种行为可能导致崩溃、传播或隐藏损坏或者其他严重的问题。但是，它

允许你绕过错误并且在块头部仍然健全的情况下从表中检索未损坏的元组。如果头部被损坏，即便这个选项被启用系统也将报告一个错误。默认设置是off，并且只能被超级用户改变。

zero\_damaged\_pages (boolean)

检测到一个损坏的页面头部通常会导致LightDB报告一个错误，并且中止当前事务。把zero\_damaged\_pages设置为打开会让系统报告一个警告、把损坏的页面填充零，然后继续处理。这种行为会毁掉数据，即被损坏页面上的所有行。但是它允许你绕开错误并且从可能存在表中的任何未损坏页面中检索行。如果由于一次硬件或软件错误而发生毁坏，这种方法可用于恢复数据。通常你不应该把它设置为打开，除非你已经彻底放弃从表的损坏页面中恢复数据。被填充零的页面不会被强制到磁盘上，因此我们推荐在再次关闭这个参数之前先重建表或索引。默认的设置是off，并且只有超级用户可以改变它。

ignore\_invalid\_pages (boolean)

如果设置为 off（默认值），则在恢复期间检测到引用无效页面的 WAL 记录会导致LightDB 引发 PANIC 级别错误，中止恢复。将ignore\_invalid\_pages设置为on会导致系统忽略 WAL 记录中的无效页面引用（但仍报告警告），并继续恢复。此行为可能会导致崩溃、数据丢失、传播或隐藏腐败，或其他严重问题。但是，它可能允许你通过 PANIC 级错误，完成恢复，并启动服务器。参数只能在服务器启动时设置。它仅在恢复期间或待机模式下生效。

jit\_debugging\_support (boolean)

如果LLVM有所需要的功能，用GDB注册所生成的函数。这会让调试更加容易。默认设置是off。这个参数只能在服务器启动时设置。

jit\_dump\_bitcode (boolean)

把生成的LLVM IR写出到文件系统，写到data\_directory中。只有在做JIT内部实现工作时，这个参数才能派上用场。默认设置是off。这个参数只能由超级用户修改。

jit\_expressions (boolean)

当JIT编译被激活时（见Section 29.2，确定表达式是否用JIT编译。默认值是on。

jit\_profiling\_support (boolean)

如果LLVM有所需的功能，发出需要的数据以允许perf对JIT生成的函数画像。这会写出文件到~/debug/jit/中，如果需要，由用户负责对其执行清除。默认设置是off。这个参数只能在服务器启动时设置。

jit\_tuple\_deforming (boolean)

当JIT编译被激活时（见Section 29.2，确定元组拆解是否被JIT编译。默认值是on。

## 17.18. 短选项

为了方便起见，系统中还为一些参数提供了单字母的命令行选项开关。它们在Table 17.3中描述。其中一些选项是由于历史原因而存在，并且它们作为一个单字母选项存在并不表示它们会被大量使用。

Table 17.3. 短选项键

短选项	等效于
-B x	shared_buffers = x
-d x	log_min_messages = DEBUGx
-e	datestyle = euro

短选项	等效于
-fb, -fh, -fi, -fm, -fn, -fo, -fs, -ft	enable_bitmapscan = off, enable_hashjoin = off, enable_indexscan = off, enable_mergejoin = off, enable_nestloop = off, enable_indexonlyscan = off, enable_seqscan = off, enable_tidscan = off
-F	fsync = off
-h x	listen_addresses = x
-i	listen_addresses = '*'
-k x	unix_socket_directories = x
-l	ssl = on
-N x	max_connections = x
-O	allow_system_table_mods = on
-p x	port = x
-P	ignore_system_indexes = on
-s	log_statement_stats = on
-S x	work_mem = x
-tpa, -tpl, -te	log_parser_stats = on, log_planner_stats = on, log_executor_stats = on
-W x	post_auth_delay = x

## 17.19. Other Options

lightdb\_enable\_default\_tablespace (integer)

启用或禁用创建数据库的默认表空间lt\_user，默认off。

lightdb\_external\_virtual\_ip (string)

配置外部虚拟IP。如果外部虚拟IP存在，在备自动切换成为主之前，它会先确定本地是否具有此IP地址。如果没有，它就不会自动处理。 详见 ltcluster<sup>2</sup>

lt\_statement\_rollback\_enable (integer)

在事务块中打开或关闭语句级回滚。默认关闭。此函数在Oracle或MySQL模式下生效。启用了语句级回滚，性能损失约为5%。最好不要在事务块中设置此开关，这可能会导致意外错误。使用lt\_dump时，也需要关闭。

lightdb\_aborted\_rollback (boolean)

lightdb\_aborted\_rollback默认禁用。启用lightdb\_aborted\_rollback后，报告错误并在事务发生错误时回滚所有语句。lt\_statement\_rollback\_enable参数打开时，您无法设置lightdb\_aborted\_rollback，它将无法执行。反之亦然。

lightdb\_sql\_mode (String)

lightdb\_sql\_mode指定了一些由逗号隔开的SQL执行规则。支持以下规则：  
no\_zero\_date, uppercase\_identifier。

可以在各个会话中更改此设置。但是不建议在会话中设置'uppercase\_identifier'，应该通过在lightdb.conf内使能该参数且尽可能不要改动它。更多关于'uppercase\_identifier'规则可参见于create\_table.sgml的'column\_name'章节及select.sgml的'SELECT List'章节，并在'Examples'内列出一些示例指导如何使用它。

<sup>2</sup> <http://www.light-pg.com/docs/ltcluster/13.8-23.4/index.html>

`lightdb_oracle_sql_mode` (String)

`lightdb_oracle_sql_mode`是一个可在会话中配置的用户级参数。设置为'`show_identifier_uppercase`'时，执行一条`select`语句时默认显示大写列名。更多关于该功能的信息可以参考`create_table.sgml`内的'列名'章节及`select.sgml`的'`SELECT List`'章节，并配有一些示例作为指导如何使用它。

`lightdb_cursor_after_commit` (String)

添加GUC参数`lightdb_cursor_after_commit`，当参数值为 `on`，事务提交后，可以继续操作该游标。当参数值为 `off`，事务提交后，不可以继续操作该游标。这个参数的默认值为`off`。

---

## Chapter 18. 客户端认证

当一个客户端应用连接一个数据库服务器时，它将指定以哪个LightDB 数据库用户名连接，就像我们以一个特定用户登录一台 Unix 计算机一样。在 SQL 环境中，活动的数据库用户名决定对数据库对象的访问权限 — 详见Chapter 19 因此，它本质上是哪些数据库用户可以连接。

### Note

如Chapter 19中所释，LightDB实际上以“角色”来进行权限管理。在本章中，我们用数据库用户表示“拥有LOGIN权限的角色”。

认证是数据库服务器建立客户端身份的过程，并且服务器决定客户端应用（或者运行客户端应用的用户）是否被允许以请求的数据库用户名来连接。

LightDB提供多种不同的客户端认证方式。被用来认证一个特定客户端连接的方法可以基于（客户端）主机地址、数据库和用户来选择。

LightDB数据库用户名在逻辑上是和服务器运行的操作系统中的用户名相互独立的。如果一个特定服务器的所有用户在那台服务器的机器上也有帐号，那么分配与操作系统用户名匹配的数据库用户名是有意义的。不过，一个接受远程连接的服务器可能有许多没有本地操作系统帐号的用户，并且在这种情况下数据库用户和操作系统用户名之间不必有任何联系。

### 18.1. lt\_hba.conf文件

客户端认证是由一个配置文件（通常名为lt\_hba.conf并被存放在数据库集簇目录中）控制（HBA表示基于主机的认证）。在initdb初始化数据目录时，它会安装一个默认的lt\_hba.conf文件。不过我们也可以把认证配置文件放在其它地方；参阅hba\_file配置参数。

lt\_hba.conf文件的常用格式是一组记录，每行一条。空白行将被忽略，#注释字符后面的任何文本也被忽略。记录可以延续到下一行并以反斜线结束该行。（反斜线不是特定的，除了于行尾。）一条记录由若干用空格和/或制表符分隔的域组成。如果域值用双引号包围，那么它可以包含空白。在数据库、用户或地址域中引用一个关键字（例如，all或replication）将使该词失去其特殊含义，并且只是匹配一个有该名字的数据库、用户或主机。反斜线行的延续在引用文本或注释中也可以使用。

每条记录指定一种连接类型、一个客户端 IP 地址范围（如果和连接类型相关）、一个数据库名、一个用户名以及对匹配这些参数的连接使用的认证方法。第一条匹配连接类型、客户端地址、连接请求的数据库和用户名的记录将被用于执行认证。这个过程没有“落空”或者“后备”的说法：如果选择了一条记录而且认证失败，那么将不再考虑后面的记录。如果没有匹配的记录，那么访问将被拒绝。

记录可以有多种格式：

```
local      database user auth-method [auth-options]
host       database user address  auth-method [auth-options]
hostssl    database user address  auth-method [auth-options]
hostnossl  database user address  auth-method [auth-options]
hostgssenc database user address  auth-method [auth-options]
hostnogssenc database user address  auth-method [auth-options]
host       database user IP-address IP-mask  auth-method [auth-
options]
hostssl    database user IP-address IP-mask  auth-method [auth-
options]
```

hostnoss1	database	user	IP-address	IP-mask	auth-method	[auth- options]
hostgssenc	database	user	IP-address	IP-mask	auth-method	[auth- options]
hostnogssenc	database	user	IP-address	IP-mask	auth-method	[auth- options]

各个域的含义如下：

local

这条记录匹配企图使用 Unix 域套接字的连接。如果没有这种类型的记录，就不允许 Unix 域套接字连接。

host

这条记录匹配企图使用 TCP/IP 建立的连接。host记录匹配SSL和非SSL的连接尝试，此外还有GSSAPI 加密的或非GSSAPI 加密的连接尝试。

### Note

除非服务器带着合适的listen\_addresses配置参数值启动，否则将不可能进行远程的 TCP/IP 连接，因为默认的行为是只监听在本地环回地址localhost上的 TCP/IP 连接。

hostssl

这条记录匹配企图使用 TCP/IP 建立的连接，但必须是使用SSL加密的连接。

要使用这个选项，编译服务器的时候必须打开SSL支持。此外，在服务器启动的时候必须通过设置ssl配置参数（详见Section 16.9 打开SSL。否则，hostssl记录会被忽略，并且会记录一个警告说它无法匹配任何连接。

hostnoss1

这条记录的行为与hostssl相反；它只匹配那些在 TCP/IP上不使用SSL的连接企图。

hostgssenc

这条记录匹配企图使用TCP/IP建立的连接，但仅当使用GSSAPI加密建立连接时。

要使用这个选项，服务器必须具备GSSAPI支持。否则，除了记录无法匹配任何连接的警告以外，hostgssenc记录将被忽略。

hostnogssenc

这个记录类型具有与hostgssenc相反的表现；它仅匹配通过不使用GSSAPI加密的TCP/IP进行的连接尝试。

database

指定记录所匹配的数据库名称。值all指定该记录匹配所有数据库。值 sameuser指定如果被请求的数据库和请求的用户同名，则匹配。值samerole指定请求的用户必须是一个与数据库同名的角色中的成员（samegroup是一个已经废弃了，但目前仍然被接受的samerole同义词）。对于一个用于samerole目的的角色，超级用户不会被考虑为其中的成员，除非它们是该角色的显式成员（直接或间接），而不是由于超级用户的原因。值replication指定如果一个物理复制连接被请求则该记录匹配，不过，它不匹配逻辑复制连接。注意，物理复制连接不指定任何特定的数据库而逻辑复制连接需要指定。在其它情况里，这就是一个特定的LightDB数据库名字。可以通过用逗号分隔的方法指定多个数据库，也可以通过在文件名前面放@来指定一个包含数据库名的文件。



## user

指定这条记录匹配哪些数据库用户名。值all指定它匹配所有用户。否则，它要么是一个特定数据库用户的名字或者是一个有前导+的组名称（回想一下，在LightDB里，用户和组没有真正的区别，+实际表示“匹配这个角色的任何直接或间接成员角色”，而没有+记号的名字只匹配指定的角色）。出于这个目的，如果超级用户显式的是一个角色的成员（直接或间接），那么超级用户将只被认为是该角色的一个成员而不是作为一个超级用户。多个用户名可以通过用逗号分隔的方法提供。一个包含用户名的文件可以通过在文件名前面加上@来指定。

## address

指定这个记录匹配的客户端机器地址。这个域可以包含一个主机名、一个 IP 地址范围或下文提到的特殊关键字之一。

一个 IP 地址范围以该范围的开始地址的标准数字记号指定，然后是一个斜线 (/) 和一个CIDR掩码长度。掩码长度表示客户端 IP 地址必须匹配的高序二进制位数。在给定的 IP 地址中，这个长度的右边的二进制位必须为零。在 IP 地址、/和 CIDR 掩码长度之间不能有空白。

这种方法指定一个 IPv4 地址范围的典型例子是：172.20.143.89/32用于一个主机，172.20.143.0/24用于一个小型网络，10.6.0.0/16用于一个大型网络。一个单主机的 IPv6 地址范围看起来像这样：::1/128 (IPv6 回环地址)，一个小型网络的 IPv6 地址范围则类似于：fe80::7a31:c1ff:0000:0000/96。0.0.0.0/0表示所有 IPv4 地址，并且::0/0表示所有 IPv6 地址。要指定一个单一主机，IPv4 用一个长度为 32 的 CIDR 掩码或者 IPv6 用长度为 128 的 CIDR 掩码。在一个网络地址中，不要省略结尾的零。

一个以 IPv4 格式给出的项将只匹配 IPv4 连接并且一个以 IPv6 格式给出的项将只匹配 IPv6 连接，即使对应的地址在 IPv4-in-IPv6 范围内。请注意如果系统的 C 库不支持 IPv6 地址，那么 IPv6 格式中的项将被拒绝。

你也可以写all来匹配任何 IP 地址、写samehost来匹配任何本服务器自身的 IP 地址或者写samenet来匹配本服务器直接连接到的任意子网的任意地址。

若果指定了一个主机名（任何除 IP 地址单位或特殊关键字之外的都被作为主机名处理），该名称会与客户端的 IP 地址的反向名字解析（例如使用 DNS 时的反向 DNS 查找）结果进行比较。主机名比较是大小写敏感的。如果匹配上，那么将在主机名上执行一次正向名字解析（例如正向 DNS 查找）来检查它解析到的任何地址是否等于客户端的 IP 地址。如果两个方向都匹配，则该项被认为匹配（lt\_hba.conf中使用的主机名应该是客户端 IP 地址的地址到名字解析返回的结果，否则该行将不会匹配。某些主机名数据库允许将一个 IP 地址关联多个主机名，但是当被要求解析一个 IP 地址时，操作系统将只返回一个主机名）。

一个以点号 (.) 开始的主机名声明匹配实际主机名的后缀。因此.example.com将匹配foo.example.com（但不匹配example.com）。

当主机名在lt\_hba.conf中被指定时，你应该保证名字解析很快。建立一个类似nsd的本地名字解析缓存是一种不错的选择。另外，你可能希望启用配置参数log\_hostname来在日志中查看客户端的主机名而不是 IP 地址。

这个域不适用于local记录。

## Note

用户有时候会疑惑为什么这样处理的主机名看起来很复杂，因为需要两次名字解析（包括一次客户端 IP 地址的反向查找）。在客户端的反向 DNS 项没有建立或者得到某些意料之外的主机名的情况下，这种方式会让该特性的使用变得复杂。这样做主要是为了效率：通过这种方式，一次连接尝试要求最多两次解析器查找，一次逆向以及一次正向。如果有一个解析器对于该地址有问题，这仅仅是客户端的问题。一种假想的替代实现是只做前向查找，这种方法不得不在每

一次连接 尝试期间解析`lt_hba.conf`中提到的每一个主机名。如果列出了很多 名称，这就会很慢。并且如果主机名之一有解析器问题，它会变成所有人的问题。

另外，一次反向查找也是实现后缀匹配特性所需的，因为需要知道实际的客户端主机名来与模式进行匹配。

注意这种行为与其他流行的基于主机名的访问控制实现相一致，例如 Apache HTTP Server 和 TCP Wrappers。

IP-address

IP-mask

这两个域可以被用作IP-address/ mask-length记号法的替代方案。和指定掩码长度不同，实际的掩码被指定在一个单独的列中。例如，255.0.0.0表示 IPv4 CIDR 掩码长度 8，而255.255.255.255表示 CIDR 掩码长度 32。

这些域不适用于local记录。

auth-method

指定当一个连接匹配这个记录时，要使用的认证方法。下面对可能的选择做了概述，详见Section 18.3

trust

无条件地允许连接。这种方法允许任何可以与LightDB数据库服务器连接的用户以他们期望的任意LightDB数据库用户身份登入，而不需要口令或者其他任何认证。详见Section 18.4

reject

无条件地拒绝连接。这有助于从一个组中“过滤出”特定主机，例如一个reject行可以阻塞一个特定的主机连接，而后面一行允许一个特定网络中的其余主机进行连接。

scram-sha-256

执行SCRAM-SHA-256认证来验证用户的口令。详见Section 18.5

md5

执行SCRAM-SHA-256或MD5认证来验证用户的口令。详见Section 18.5

password

要求客户端提供一个未加密的口令进行认证。因为口令是以明文形式在网络上发送的，所以我们不应该在不可信的网络上使用这种方式。详见Section 18.5

gss

用 GSSAPI 认证用户。只对 TCP/IP 连接可用。详见Section 18.6它可以与 GSSAPI 加密一起结合使用。

sspi

用 SSPI 来认证用户。只在 Windows 上可用。详见Section 18.7

ident

通过联系客户端的 ident 服务器获取客户端的操作系统名，并且检查它是否匹配被请求的数据库用户名。Ident 认证只能在 TCIP/IP 连接上使用。当为本地连接指定这种认证方式时，将用 peer 认证来替代。详见Section 18.8

peer

从操作系统获得客户端的操作系统用户，并且检查它是否匹配被请求的数据库用户名。这只对本地连接可用。详见Section 18.9

ldap

使用LDAP服务器认证。详见Section 18.10

radius

用 RADIUS 服务器认证。详见Section 18.11

cert

使用 SSL 客户端证书认证。详见Section 18.12

pam

使用操作系统提供的可插入认证模块服务（PAM）认证。详见Section 18.13

bsd

使用由操作系统提供的 BSD 认证服务进行认证。详见Section 18.14

auth-options

在auth-method域的后面，可以是形如name=value的域，它们指定认证方法的选项。关于哪些认证方法可以用哪些选项的细节请见下文。

除了下文列出的与方法相关的选项之外，还有一个与方法无关的认证选项clientcert，它可以在任何hostssl记录中指定。这个选项可以被设置到verify-ca或verify-full。这两个选项都需要客户端提供有效的(受信任的)SSL证书，并且verify-full额外强制证书中的cn(通用名称)匹配用户名或适用的映射。这种行为类似于cert认证方法(参见 Section 18.12，但是允许客户端证书的验证与任何支持hostssl登记的认证方法相搭配。

在使用客户端证书任何的记录上(即一个使用cert认证方法或一个使用clientcert选项)，你可以使用clientname选项指定要匹配的客户端证书凭证的哪一部分。这个选项可以用两个值中的一个。如果你指定clientname=CN，这是默认值，则用户名将与证书的Common Name (CN)进行匹配。如果你指定clientname=DN，则用户名将与证书的整个Distinguished Name (DN)进行匹配。这个选项最好与用户名映射一起使用。DN在RFC 2253<sup>1</sup>格式的完成完成比较 要在这种格式中查看客户端证书的DN，请执行

```
openssl x509 -in myclient.crt -noout --subject -nameopt RFC2253 | sed "s/
^subject=/"
```

使用这个选项时要小心，特别是使用正则表达式匹配DN的时候。

用@结构包括的文件被读作一个名字列表，它们可以用空白或者逗号分隔。注释用#引入，就像在lt\_hba.conf中那样，并且允许嵌套@结构。除非跟在@后面的文件名是一个绝对路径，文件名都被认为是相对于包含引用文件的目录。

因为每一次连接尝试都会顺序地检查lt\_hba.conf记录，所以这些记录的顺序是非常关键的。通常，靠前的记录有比较严的连接匹配参数和比较弱的认证方法，而靠后的记录有比较松的匹配参数和比较强的认证方法。例如，我们希望对本地 TCP/IP 连接使用trust认证，而对远程 TCP/IP 连接要求口令。在这种情况下为来自于 127.0.0.1 的连接指定trust认证的记录将出现在为一个更宽范围的客户端 IP 地址指定口令认证的记录前面。

在启动以及主服务器进程收到SIGHUP信号时，lt\_hba.conf文件会被读取。如果你在活动的系统上编辑了该文件，你将需要通知 postmaster (使用lt\_ctl reload, 调用SQL函数pg\_reload\_conf(), 或使用kill -HUP) 使其重新读取该文件。

<sup>1</sup> <https://tools.ietf.org/html/rfc2253>

## Note

前面的说明在Microsoft Windows上不为真：在Windows上，lt\_hba.conf文件中的任何更改会立即被应用到后续的新连接上。

系统视图pg\_hba\_file\_rules有助于预先测试对lt\_hba.conf文件的更改，该视图也可以在该文件的装载没有产生预期效果时用于诊断问题。该视图中带有非空error域的行就表示该文件对对应行中存在问题。

## Tip

要连接到一个特定数据库，一个用户必须不仅要通过lt\_hba.conf检查，还必须要有该数据库上的CONNECT权限。如果你希望限制哪些用户能够连接到哪些数据库，授予/撤销CONNECT权限通常比在lt\_hba.conf项中设置规则简单。

Example 18. 中展示了lt\_hba.conf项的一些例子。不同认证方法的详情请见下一节。

## Example 18.1. 示例t\_hba.conf 项

```
# 允许本地系统上的任何用户
# 通过 Unix 域套接字以任意
# 数据库用户名连接到任意数据库（本地连接的默认值）。
#
# TYPE DATABASE          USER          ADDRESS          METHOD
local  all                all           trust

# 相同的规则，但是使用本地环回 TCP/IP 连接。
#
# TYPE DATABASE          USER          ADDRESS          METHOD
host   all                all           127.0.0.1/32    trust

# 和前一例相同，但是使用了一个独立的掩码列
#
# TYPE DATABASE          USER          IP-ADDRESS      IP-MASK          METHOD
host   all                all           127.0.0.1       255.255.255.255 trust

# IPv6 上相同的规则
#
# TYPE DATABASE          USER          ADDRESS          METHOD
host   all                all           ::1/128         trust

# 使用主机名的相同规则（通常同时覆盖 IPv4 和 IPv6）。
#
# TYPE DATABASE          USER          ADDRESS          METHOD
host   all                all           localhost       trust

# 允许来自任意具有 IP 地址192.168.93.x 的主机上任意
# 用户以 ident 为该连接所报告的不同用户名连接到
# 数据库 "postgres"（通常是操作系统用户名）。
#
# TYPE DATABASE          USER          ADDRESS          METHOD
```

```

host    postgres    all                192.168.93.0/24    ident

# 如果用户的口令被正确提供, 允许来自主机 192.168.12.10
# 的任意用户连接到数据库 "postgres"。
#
# TYPE DATABASE      USER      ADDRESS      METHOD
host    postgres    all        192.168.12.10/32  scram-sha-256

# 如果用户的口令被正确提供, 允许 example.com 中主机上
# 的任意用户连接到任意数据库。
#
# 为大部分用户要求SCRAM认证, 但是用户'mike'是个例外,
# 他使用的是不支持SCRAM认证的旧客户端。
#
# TYPE DATABASE      USER      ADDRESS      METHOD
host    all        mike       .example.com  md5
host    all        all        .example.com  scram-sha-256

# 如果没有前面的 "host" 行, 这三行
# 将拒绝所有来自 192.168.54.1的连接 (因为那些项将首先被匹配),
# 但是允许来自互联网其他任何地方的
# GSSAPI-encrypted连接。零掩码导致主机IP 地址中的所有位都不会被考虑,
# 因此它匹配任意主机。未加密GSSAPI连接
# (which "跳转"到第三行是因为"hostgssenc" 仅匹配加密的 GSSAPI 连接) 是被允许的,
# 但只能来自192.168.12.10.
#
# TYPE DATABASE      USER      ADDRESS      METHOD
host    all        all        192.168.54.1/32  reject
hostgssenc all      all        0.0.0.0/0      gss
host    all        all        192.168.12.10/32 gss

# 允许来自 192.168.x.x 主机的用户连接到任意数据库, 如果它们能够
# 通过 ident 检查。例如, 假设 ident说用户是 "bryanh" 并且他要求以
# LightDB 用户 "guest1" 连接, 如果在 lt_ident.conf 有一个映射
# "omicron" 的选项说 "bryanh" 被允许以 "guest1" 连接, 则该连接将被允许。
#
# TYPE DATABASE      USER      ADDRESS      METHOD
host    all        all        192.168.0.0/16   ident
map=omicron

# 如果这些是本地连接的唯一三行, 它们将允许本地用户只连接到它们
# 自己的数据库 (与其数据库用户名同名的数据库), 不过管理员和角
# 色 "support" 的成员除外 (它们可以连接到所有数据库)。文件
# $LTDATA/admins 包含一个管理员名字的列表。在所有情况下都要求口令。
#
# TYPE DATABASE      USER      ADDRESS      METHOD
local  sameuser  all        md5
local  all      @admins   md5
local  all      +support  md5

# 上面的最后两行可以被整合为一行:
local  all      @admins,+support  md5

# 数据库列也可以用列表和文件名:
local  db1,db2,@demodbs all      md5

```

## 18.2. 用户名映射

当使用像 Ident 或者 GSSAPI 之类的外部认证系统时，发起连接的操作系统用户名可能不同于要被使用的数据库用户（角色）。在这种情况下，一个用户名映射可被用来把操作系统用户名映射到数据库用户。要使用用户名映射，在 `lt_hba.conf` 的选项域指定 `map=map-name`。此选项支持所有接收外部用户名的认证方法。由于不同的连接可能需要不同的映射，在 `lt_hba.conf` 中的 `map-name` 参数中指定要被使用的映射名，用以指示哪个映射用于每个个体连接。

用户名映射定义在 `ident` 映射文件中，默认情况下它被命名为 `lt_ident.conf` 并被存储在集群的数据目录中（不过，可以把该映射文件放在其他地方，见 `ident_file` 配置参数）。`ident` 映射文件包含的行的一般格式：

```
map-name system-username database-username
```

在 `lt_hba.conf` 中同样的方式处理注释，空白和行延续。 `map-name` 是一个任意名称，它将被用于在 `lt_hba.conf` 中引用该映射。 其他两个域指定一个操作系统用户名和一个匹配的数据库用户名。 相同的 `map-name` 可以被反复地用在同一个映射中指定多个用户映射。

对于一个给定操作系统用户可以对应多少个数据库用户没有限制，反之亦然。因此，一个映射中的项应该被看成意味着“这个操作系统用户被允许作为这个数据库用户连接”，而不是按时它们等价。如果有任何映射项把从外部认证系统获得的用户名和用户要求的数据库用户名配对，该连接将被允许。

如果 `system-username` 域以一个斜线（/）开始，域的剩余部分被当做一个正则表达式（LightDB的正则表达式语法详见Section 9.7.3.1）。正则表达式可以包括一个单一的捕获，或圆括号子表达式，然后它可以在 `database-username` 域中以 `\1`（反斜线一）被引用。这允许在单个行中多个用户名的映射，这特别有助于简单的语法替换。例如，这些项

```
mymap /^(.*)@mydomain\.com$ \1
mymap /^(.*)@otherdomain\.com$ guest
```

将为用户移除以 `@mydomain.com` 结束的系统用户名的域部分，以及允许系统名以 `@otherdomain.com` 结束的任意用户作为 `guest` 登入。

### Tip

记住在默认情况下，一个正则表达式可以只匹配字符串的一部分。如上例所示，使用 `^` 和 `$` 来强制匹配整个系统用户名通常是明智的。

在启动以及主服务器进程收到 `SIGHUP` 信号时，`lt_ident.conf` 文件会被读取。如果你在活动的系统上编辑了该文件，你将需要通知 `postmaster`（使用 `lt_ctl reload`，调用 SQL 函数 `pg_reload_conf()`，或用 `kill -HUP`）重新读取文件。

Example 18.2 展示了一个可以联合 `lt_hba.conf` 文件（Example 18.1 使用的 `lt_ident.conf` 文件。在这个例子中，对于任何登入到 `192.168` 网络上的一台机器的用户，如果该用户没有操作系统用户名 `bryanh`、`ann` 或 `robert`，则他不会被授予访问权限。只有当 Unix 用户 `robert` 尝试作为 LightDB 用户 `bob`（而不是作为 `robert` 或其他人）连接时，他才被允许访问。`ann` 只被允许作为 `ann` 连接。用户 `bryanh` 被允许以 `bryanh` 或者 `guest1` 连接。

Example 18.2. 一个示例 `lt_ident.conf` 文件

```
# MAPNAME          SYSTEM-USERNAME          PG-USERNAME

omicron           bryanh                   bryanh
omicron           ann                      ann
# bob 在这些机器上有用户名 robert
```

```

omicron      robert      bob
# bryanh 也可以作为 guest1 连接
omicron      bryanh      guest1

```

## 18.3. 认证方法

LightDB 为认证用户提供不同的方法:

- Trust authentication, 简单的信任用户声称的身份。
- Password authentication, 需要用户提供密码。
- GSSAPI authentication, 依靠GSSAPI兼容的安全库, 常常用于访问认证服务器, 例如 Kerberos或微软活动目录服务器。
- SSPI authentication, 用windows规定的类似于GSSAPI的协议。
- Ident authentication, 依靠客户机器上的 “Identification Protocol” (RFC 1413<sup>2</sup>) 服务, (在本地Unix-socket 连接, 这个按对等认证处理)。
- Peer authentication, 依靠操作系统工具来识别本地连接另一端的进程。这种方法不支持远程连接。
- LDAP authentication, 依靠LDAP认证服务器。
- RADIUS authentication, 依靠RADIUS认证服务器。
- Certificate authentication, 需要SSL连接和通过SSL证书检查的认证用户。
- PAM authentication, 依靠PAM库(Pluggable Authentication Modules, 可插拔认证模块)。
- BSD authentication, 依靠BSD认证框架(当前仅在OpenBSD上应用)。

对等身份验证通常适用于本地连接, 信任认证在某些情况下也许是比较适合的。密码认证是远程连接的常见选择。所有其它的选项都需要某种外部安全基础架构(通常是认证服务器或颁发 SSL 证书的证书颁发机构。), 或用于某些特定平台。

以下各节将更详细地介绍这些身份认证方法。

## 18.4. 信任认证

当trust认证被指定时, LightDB假设任何可以连接到服务器的人都被授权使用他们指定的任何数据库用户名(即使是超级用户)访问数据库。当然, 在database和 user列中设置的限制仍然适用。只有当在操作系统层对进入服务器的连接有足够保护时, 才应该使用这种方法。

trust认证对于单用户工作站的本地连接是非常合适和方便的。通常它本身不适用于一台多用户机器。不过, 只要你利用文件系统权限限制了对服务器的 Unix 域套接字文件的访问, 即使在多用户机器上, 你也可以使用trust。要做这些限制, 你可以设置Section 17.8<sup>2</sup>中描述的unix\_socket\_permissions配置参数(可能还有unix\_socket\_group)。或者你可以设置unix\_socket\_directories配置参数来把 Unix 域套接字文件放在一个经过恰当限制的目录中。

设置文件系统权限只能有助于 Unix 套接字连接。本地 TCP/IP 连接不会被文件系统权限限制。因此, 如果你想利用文件系统权限来控制本地安全, 那么从lt\_hba.conf中移除host ... 127.0.0.1 ... 行, 或者把它改为一个非trust认证方法。

<sup>2</sup> <https://tools.ietf.org/html/rfc1413>

如果通过指定trust的lt\_hba.conf行让你信任每一个被允许连接到服务器的机器上的用户，trust认证只适合TCP/IP连接。为任何不是来自localhost（127.0.0.1）的TCP/IP连接使用trust很少是合理的。

## 18.5. 口令认证

有几种基于口令的认证方法。这些方法的过程类似，但是区别在于用户口令如何被存放在服务器上以及客户端提供的口令如何通过连接发送。

### scram-sha-256

方法scram-sha-256按照RFC 7677<sup>3</sup>中的描述执行SCRAM-SHA-256认证。它使用的是一种挑战-响应的方案，可以防止在不可信连接上对口令的嗅探并且支持在服务器上以一种加密哈希的方式存放口令，因此被认为是安全的。

这是当前提供的方法中最安全的一种，但是旧的客户端库不支持这种方法。

### md5

方法md5使用一种自定义的安全性较低的挑战-响应机制。它能防止口令嗅探并且防止口令在服务器上以明文存储，但是无法保护攻击者想办法从服务器上窃取了口令哈希的情况。此外，现在认为MD5哈希算法对于确定攻击已经不再安全。

md5方法不能与db\_user\_namespace特性一起使用。

为了简化从md5方法到较新的SCRAM方法的转变，如果在lt\_hba.conf中指定了md5但是用户在服务器上的口令是为SCRAM（见下文）加密的，则将自动选择基于SCRAM的认证。

### password

方法password以明文形式发送口令，因此它对于口令“嗅探”攻击很脆弱。如果可能应该尽量避免使用它。不过，如果连接被SSL加密保护着，那么可以安全地使用password（不过如果依靠SSL，SSL证书认证可能是更好的选择）。

LightDB数据库口令独立于操作系统用户口令。每个数据库用户的口令被存储在pg\_authid系统目录中。口令可以用SQL命令CREATE USER和ALTER ROLE管理，例如CREATE ROLE foo WITH LOGIN PASSWORD 'secret'或者ltsql的\password命令。如果没有为一个用户设置口令，那么存储的口令为空并且对该用户的口令认证总会失败。

不同的基于口令的认证方法的可用性取决于用户的口令在服务器上是如何被加密（或者更准确地说是哈希）的。这由设置口令时的配置参数password\_encryption控制。如果口令使用scram-sha-256设置加密，那么它可以被用于认证方法scram-sha-256和password（但后一种情况中口令将以明文传输）。如上所释，在这种情况下，指定的认证方法md5将自动切换到使用scram-sha-256方法。如果口令使用md5设置加密，那么它仅能用于md5和password认证方法说明（同样，后一种情况中口令以明文传输）。（之前的LightDB发行版支持在服务器上存储明文口令。现在已经不可能了）。要检查当前存储的口令哈希，可以参考系统目录pg\_authid。

要把现有的安装从md5升级到scram-sha-256，可以在确保所有在用的客户端已经足以支持SCRAM之后，在lightdb.conf中设置password\_encryption = 'scram-sha-256'，然后让所有用户设置新口令并且在lt\_hba.conf中将认证方法说明改为scram-sha-256。

## 18.6. GSSAPI 认证

GSSAPI是用于RFC 2743<sup>4</sup>中定义的安全认证的一个工业标准协议。LightDB支持GSSAPI用于认证，通讯加密或两者都做。GSSAPI为支持它的系统提供自动认证（单点登录）。认证本身是安全的。如果使用了GSSAPI加密或SSL加密，数据库连接发送的数据将被加密，否则将不加密。

<sup>3</sup> <https://tools.ietf.org/html/rfc7677>

<sup>4</sup> <https://tools.ietf.org/html/rfc2743>



当GSSAPI使用Kerberos的时候，它使用一个标准服务主体(身份验证)名称，以servicename/hostname@realm的格式。特定安装使用的主体名称(principal name)没有被以任何方式编码在LightDB服务器中；而是被指定在keytab文件中，服务器读取该文件以决定它的身份。如果在keytab文件中罗列着多个主体，服务器将接受其中任何一个。服务器的领域名称(realm name)是在服务器可访问的Kerberos配置文件中指定的优先领域。

当连接时，客户端必须知道它打算连接的服务器的主体的名称。主体名称的servicename部分通常是postgres，但是其他值可以通过libpq的krbsrvname连接参数来选择。hostname部分是libpq告知要连接的全限制的主机名称。领域名称是在客户端可访问的Kerberos配置文件中指定的优先领域。

客户端也可以有主体名称作为它自己的身份(并且它必须拥有针对这个主体的有效的标签[ticket])。要使用GSSAPI做身份验证，客户端的主体必须与LightDB数据库用户名关联。lt\_ident.conf配置文件可以用于映射主体到用户名；例如，pgusername@realm可能会被映射到pgusername。或者，你可以使用完整的username@realm当事人作为LightDB中的角色而无需任何映射。

LightDB也支持映射客户端主体到用户名，通过从主体中剥离领域(realm)的方式。这种方法是为了向后兼容性，并且我们强烈反对使用它，因为这样就无法区分具有相同用户名却来自不同realm的不同用户了。要启用这种方法，可将include\_realm设置为0。对于简单的单realm安装，这样做并且设置krb\_realm参数(这会检查principal的realm是否正好匹配krb\_realm中的参数)仍然是安全的。但比起在lt\_ident.conf中指定一个显式映射来说，这种方法的能力较低。

服务器的keytab文件的位置是由krb\_server\_keyfile配置参数指定的。出于安全原因，它建议针对LightDB使用独立的keytab而不是允许服务器读取系统的keytab文件。确保你的服务器的keytab文件是对LightDB服务器账号可读的(并且最好是只读的，不可写)(参见Section 16.1)

keytab文件用Kerberos软件生成；详见Kerberos文档。下面展示了用MIT兼容的Kerberos 5实现的kadmin来做这个的例子：

```
kadmin% addprinc -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

GSSAPI验证方法支持下列身份验证选项：

include\_realm

如果设置为0，在通过用户名映射之前(Section 18.2，来自自己认证用户principal的realm名称会被剥离掉。我们不鼓励这样做，这种方法主要是为了向后兼容性而存在的，因为它在多realm环境中是不安全的(除非也使用krb\_realm)。推荐用户让include\_realm设置为默认值(1)并且在lt\_ident.conf中提供一条显式的映射来把principal名称转换成LightDB用户名。

map

允许从客户端主体到数据库用户名之间的映射。详见Section 18.2 对于一个GSSAPI/Kerberos原则，例如username@EXAMPLE.COM(或者更不常见的username/hostbased@EXAMPLE.COM)，用于映射的用户名会是username@EXAMPLE.COM(或者username/hostbased@EXAMPLE.COM，相应地)，除非include\_realm已经被设置为0，在那种情况下username(或者username/hostbased)是映射时被视作系统用户名的东西。

krb\_realm

Sets the realm to match user principal names against. If this parameter is set, only users of that realm will be accepted. If it is not set, users of any realm can connect, subject to whatever user name mapping is done.

在这些设置之外，对于不同的`lt_hba.conf`项可能有所不同，还有服务器范围的`krb_caseins_users`配置参数。如果设置为真，客户端主体匹配用户映射条目是大小写不敏感的。如果设置了`krb_realm`，也不区分大小写。

## 18.7. SSPI 认证

SSPI是一种用于带单点登录的安全认证的Windows技术。LightDB在`negotiate`模式中将使用SSPI，它在可能的情况下使用Kerberos并在其他情况下自动降回到NTLM。只有在服务器和客户端都运行着Windows时，SSPI才能工作。或者在非Windows平台上GSSAPI可用时，SSPI也能工作。

当使用Kerberos认证时，SSPI和GSSAPI的工作方式相同，详见Section 18.6

下列被支持的配置选项用于SSPI：

### `include_realm`

如果设置为0，在通过用户名映射之前（Section 18.2，来自自己认证用户principal的realm名称会被剥离掉。我们不鼓励这样做，这种方法主要是为了向后兼容性而存在的，因为它在多realm环境中是不安全的（除非也使用`krb_realm`）。推荐用户让`include_realm`设置为默认值（1）并且在`lt_ident.conf`中提供一条显式的映射来把principal名称转换成LightDB用户名。

### `compat_realm`

如果被设置为1，该域的SAM兼容名称（也被称为NetBIOS名称）被用于`include_realm`选项。这是默认值。如果被设置为0，会使用来自Kerberos用户主名的真实realm名称。

不要禁用这个选项，除非你的服务器运行在一个域账号（这包括一个域成员系统上的虚拟服务账号）下并且所有通过SSPI认证的所有客户端也在使用域账号，否则认证将会失败。

### `upn_username`

如果这个选项和`compat_realm`一起被启用，来自Kerberos UPN的用户名会被用于认证。如果它被禁用（默认），会使用SAM兼容的用户名。默认情况下，对于新用户账号这两种名称是一样的。

注意如果没有显式指定用户名，libpq会使用SAM兼容的名称。如果你使用的是libpq或者基于它的驱动，你应该让这个选项保持禁用或者在连接字符串中显式指定用户名。

### `map`

允许在系统和数据库用户名之间的映射。详见Section 18.2 对于一个GSSAPI/Kerberos原则，例如`username@EXAMPLE.COM`（或者更不常见的`username/hostbased@EXAMPLE.COM`），用于映射的用户名会是`username@EXAMPLE.COM`（或者`username/hostbased@EXAMPLE.COM`，相应地），除非`include_realm`已经被设置为0，在那种情况下`username`（或者`username/hostbased`）是映射时被视作系统用户名的东西。

### `krb_realm`

设置领域为对用户principal名进行匹配的范围。如果这个参数被设置，只有那个领域的用户将被接受。如果它没有被设置，任何领域的用户都能连接，服从任何已完成的用户名映射。

## 18.8. Ident 认证

ident 认证方法通过从一个 ident 服务器获得客户端的操作系统用户名并且用它作为被允许的数据库用户名（和可选的用户名映射）来工作。它只在 TCP/IP 连接上支持。

### Note

当为一个本地（非 TCP/IP）连接指定 ident 时，将实际使用 peer 认证（见Section 18.9）。

下列被支持的配置选项用于ident:

map

允许系统和数据库用户名之间的映射。详见Section 18.2

“Identification Protocol (标识协议)”在 RFC 1413<sup>5</sup>中描述。实际上每个类 Unix 操作系统都带着一个默认监听 TCP 113 端口的 ident 服务器。ident 服务器的基本功能是回答类似这样的问题：“哪个用户从你的端口X发起了连接并且连到了我的端口Y?”。因为当一个物理连接被建立后，LightDB既知道X也知道Y，所以它可以询问尝试连接的客户端主机上的 ident 服务器并且在理论上可以判断任意给定连接的操作系统用户。

这个过程的缺点是它依赖于客户端的完整性：如果客户端机器不可信或者被攻破，攻击者可能在 113 端口上运行任何程序并且返回他们选择的任何用户。因此这种认证方法只适用于封闭的网络，这样的网络中的每台客户端机器都处于严密的控制下并且数据库和操作系统管理员操作时可以方便地联系。换句话说，你必须信任运行 ident 服务器的机器。注意这样的警告：

标识协议的本意不是作为一种认证或访问控制协议。

—RFC 1413

有些 ident 服务器有一个非标准的选项，它导致返回的用户名是被加密的，使用的是只有原机器管理员知道的一个密钥。当与LightDB配合使用 ident 服务器时，一定不要使用这个选项，因为LightDB没有任何方法对返回的字符串进行解密以获取实际的用户名。

## 18.9. Peer 认证

Peer 认证方法通过从内核获得客户端的操作系统用户名并把它用作被允许的数据库用户名（和可选的用户名映射）来工作。这种方法只在本地连接上支持。

下列被支持的配置选项用于peer:

map

允许在系统和数据库用户名之间的映射。详见Section 18.2

Peer 认证只在提供getpeereid()函数、SO\_PEERCREDS套接字参数或相似机制的操作系统上可用。这些 OS 当前包括Linux、大部分的BSD包括OS X以及Solaris。

## 18.10. LDAP 认证

这种认证方法操作起来类似于password，只不过它使用 LDAP 作为密码验证方法。LDAP 只被用于验证用户名/口令对。因此，在使用 LDAP 进行认证之前，用户必须已经存在于数据库中。

<sup>5</sup> <https://tools.ietf.org/html/rfc1413>

LDAP 认证可以在两种模式下操作。在第一种模式中（我们将称之为简单绑定模式），服务器将绑定到构造为prefix username suffix的可区分名称。通常，prefix参数被用于指定 cn=或者一个活动目录环境中的DOMAIN\。suffix被用来指定非活动目录环境中的DN的剩余部分。

在第二种模式中（我们将称之为搜索与绑定模式），服务器首先用一个固定的用户名和密码（用ldapbinddn和ldapbindpasswd指定）绑定到 LDAP 目录，并为试图登入该数据库的用户执行一次搜索。如果没有配置用户名和密码，将尝试一次匿名绑定到目录。搜索将在位于ldapbasedn的子树上被执行，并将尝试做一次ldapsearchattribute中指定属性的精确匹配。一旦在这次搜索中找到用户，服务器断开并且作为这个用户重新绑定到目录，使用由客户端指定的口令来验证登录是正确的。这种模式与在其他软件中的 LDAP 认证所使用的相同，例如 Apache mod\_authnz\_ldap 和 pam\_ldap。这种方法允许位于目录中用户对象的更大灵活性，但是会导致建立两个到 LDAP 服务器的独立连接。

下列配置选项被用于两种模式：

ldapservers

要连接的LDAP服务器的名称或IP地址。可以指定多个服务器，用空格分隔。

ldapport

要连接的LDAP服务器的端口号。如果没有指定端口，LDAP库的默认端口设置将被使用。

ldapscheme

设置为ldaps可以使用LDAPS。这是一种非标准的在SSL之上使用LDAP的方法，在有一些LDAP服务器实现上可以支持。其他选择还可以参考ldaptls选项。

ldaptls

设置为1以使LightDB和LDAP服务器之间的连接使用TLS加密。这会按照RFC 4513<sup>6</sup>使用StartTLS操作。其他选择还可以参考ldapscheme选项。

注意使用ldapscheme或ldaptls仅会加密LightDB服务器和LDAP服务器之间的通信。LightDB服务器和LightDB客户端之间的连接仍是未加密的，除非也在其上使用SSL。

下列选项只被用于简单绑定模式：

ldapprefix

当做简单绑定认证时，前置到用户名形成要用于绑定的DN的字符串。

ldapsuffix

当做简单绑定认证时，前置到用户名形成要用于绑定的DN的字符串。

下列选项只被用于搜索与绑定模式：

ldapbasedn

当做搜索与绑定认证时，开始搜索用户的根DN。

ldapbinddn

当做搜索与绑定认证时，用户要绑定到目录开始执行搜索的DN。

ldapbindpasswd

当做搜索与绑定认证时，用户用于绑定到目录开始执行搜索的口令。

<sup>6</sup> <https://tools.ietf.org/html/rfc4513>

### ldapsearchattribute

当做搜索与绑定认证时，在搜索中用来与用户名匹配的属性。如果没有指定属性，将会使用uid属性。

### ldapsearchfilter

在做search+bind认证时使用的搜索过滤器。\$username的出现将被替换为用户名。这允许比ldapsearchattribute更加灵活的搜索过滤器。

### ldapurl

一个RFC 4516<sup>7</sup> LDAP URL。这是一种用更紧凑和标准的形式书写某些其他LDAP选项的可选方法。格式是

```
ldap[s]://host[:port]/basedn[?[attribute][?[scope][?[filter]]]]
```

scope必须是base、one、sub之一，通常是最后一个（默认是base，但它在这个应用中通常没啥用）。attribute可以指定一个属性，在这种情况下它被用作ldapsearchattribute的一个值。如果attribute为空，那么filter可以被用作ldapsearchfilter的一个值。

URL模式ldaps选择LDAPS方法来在SSL上建立LDAP连接，等效于使用ldapscheme=ldaps。要使用StartTLS操作加密LDAP连接，可以用普通的URL模式ldap并且在ldapurl之外指定ldaptls选项。

对于非匿名绑定，ldapbinddn和ldapbindpasswd必须被指定为独立选项。

LDAP URL 当前只支持OpenLDAP，而不支持 Windows。

将简单绑定的选项中混合用于搜索与绑定的选项是一种错误。

在使用search+bind模式时，可以用ldapsearchattribute指定的单个属性执行搜索，或者使用ldapsearchfilter指定的自定义搜索过滤器执行搜索。指定ldapsearchattribute=foo等效于指定ldapsearchfilter="(foo=\$username)"。如果两个选项都没有被指定，则默认为ldapsearchattribute=uid。

如果LightDB与OpenLDAP一起编译为LDAP客户端库，ldapservice 设置可能被省略。在这种情况下，主机名和端口列表通过RFC 2782<sup>8</sup> DNS SRV记录来进行查询。查询名称\_ldap.\_tcp.DOMAIN，DOMAIN是从ldapbasedn中摘取的。

这里是一个简单绑定 LDAP 配置的例子：

```
host ... ldap ldapservice=ldap.example.net ldapprefix="cn=" ldapsuffix="",
dc=example, dc=net"
```

当请求一个作为数据库用户someuser到数据库服务器的连接时，LightDB 将尝试使用cn=someuser, dc=example, dc=net和客户端提供的口令来绑定到 LDAP 服务器。如果那个连接成功，将被授予数据库访问。

这里是一个搜索与绑定配置的例子：

```
host ... ldap ldapservice=ldap.example.net ldapbasedn="dc=example, dc=net"
ldapsearchattribute=uid
```

<sup>7</sup> <https://tools.ietf.org/html/rfc4516>

<sup>8</sup> <https://tools.ietf.org/html/rfc2782>

当请求一个作为数据库用户someuser到数据库服务器的连接时，LightDB 将尝试匿名绑定（因为没有指定ldapbinddn）到 LDAP 服务器，在指定的基础 DN 下执行一次对于(uid=someuser)的搜索。如果找到一个项，则它将尝试使用找到的信息和客户端提供的口令进行绑定。如果第二个连接成功，将被授予数据库访问。

这里是被写成一个 URL 的相同搜索与绑定配置：

```
host ... ldap ldapurl="ldap://ldap.example.net/dc=example,dc=net?uid?sub"
```

一些支持根据 LDAP 认证的其他软件使用相同的 URL 格式，因此很容易共享该配置。

这里是一个search+bind配置的例子，它使用ldapsearchfilter而不是ldapsearchattribute来允许用用户ID或电子邮件地址进行认证：

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net"
  ldapsearchfilter="(|(uid=$username)(mail=$username))"
```

这是一个search+bind配置的例子，它使用DNS SRV discovery来查找域名example.net的LDAP服务的主机名和端口。

```
host ... ldap ldapbasedn="dc=example,dc=net"
```

### Tip

如例子中所示，由于 LDAP 通常使用逗号和空格来分割一个 DN 的不同部分，在配置 LDAP 选项时通常有必要使用双引号包围的参数值。

## 18.11. RADIUS 认证

这种认证方法的操作类似于password，不过它使用 RADIUS 作为密码验证方式。RADIUS 只被用于验证 用户名/密码对。因此，在 RADIUS 能被用于认证之前，用户必须已经存在于数据库中。

当使用 RADIUS 认证时，一个访问请求消息将被发送到配置好的 RADIUS 服务器。这一请求将是Authenticate Only类型，并且包含参数user name、password（加密的）和NAS Identifier。该请求将使用一个与服务器共享的密钥加密。RADIUS 服务器将对这个请求响应Access Accept或者Access Reject。不支持RADIUS accounting。

可以指定多个RADIUS服务器，这种情况下将会依次尝试它们。如果从一台服务器接收到否定响应，则认证失败。如果没有接收到响应，则将会尝试列表中的下一台服务器。要指定多台服务器，可用双引号括住列表并用逗号将服务器名称分开。如果指定了多台服务器，其他RADIUS选项也可以用逗号分隔的列表给出，用来为每台服务器应用个别的值。它们也可以指定为单个值，这种情况下该值将被应用到所有的服务器。

下列被支持的配置选项用于 RADIUS：

radiusservers

连接到 RADIUS 服务器的DNS名称或IP地址。此参数是必需的。

radiussecrets

和 RADIUS 服务器秘密交谈时会用到共享密钥。这在 LightDB 和 RADIUS 服务器之间必须有完全相同的值。我们推荐用一个至少 16 个字符的字符串。这个参数是必需的。

## Note

如果LightDB编译为支持OpenSSL，所用的加密向量将只是强密码。在其他情况下，到 RADIUS 服务器的传输应该被视为应该被视为被混淆的、不安全的。如有必要，应采用外部安全措施。

## radiusports

连接 RADIUS 服务器的端口号。如果没有指定端口，则使用默认RADIUS端口1812。

## radiusidentifiers

在 RADIUS 请求中字符串被用作NAS Identifier。这个参数可以被用于识别用户尝试连接哪些数据库集群，可以被用于 RADIUS 服务器上的策略匹配。如果没有指定标识符，默认使用postgresql。

如果RADIUS参数值中需要有逗号或者空格，可以通过双引号括住该值来完成，但这样做是比较繁琐的因为需要两层双引号。将空格放到RADIUS秘密字符串的一个示例为：

```
host ... radius radiusservers="server1,server2" radiussecrets=""secret
one","secret two""
```

## 18.12. 证书认证

这种认证方法使用 SSL 客户端证书执行认证。因此，它只适用于 SSL 连接。当使用这种认证方法时，服务器将要求客户端提供一个有效的、可信的证书。不会有密码提示将被发送到客户端。证书的cn（通用名）属性将与被请求的数据库用户名进行比较，并且如果匹配将允许登录。用户名映射可以被用来允许cn与数据库用户名不同。

下列被支持的配置选项用于 SSL 证书认证：

## map

允许在系统和数据库用户名之间的映射。详见Section 18.2

将 clientcert 选项与 cert验证一起使用是多余的，因为 cert身份验证是经过clientcert=verify-full的有效的 trust 验证。

## 18.13. PAM 认证

这种认证方法操作起来类似password，只不过它使用 PAM（插入式验证模块）作为认证机制。默认的 PAM 服务名是postgresql。PAM 只被用于验证用户名/口令对并且可以有选择地验证已连接的远程主机名或 IP 地址。因此，在使用 PAM 进行认证之前，用户必须已经存在于数据库中。有关 PAM 的更多信息，请阅读 Linux-PAM 页面<sup>9</sup>。

下列被支持的配置选项用于 PAM：

## pamservice

PAM服务名称。

## pam\_use\_hostname

判断是否通过PAM\_RHOST项把远程 IP 地址或者主机名提供给 PAM 模块。默认情况下会使用 IP 地址。把这个选项设置为 1 可以使用解析过的主机名。主机名解析可能导致登

<sup>9</sup> <https://www.kernel.org/pub/linux/libs/pam/>

录延迟（大部分的 PAM 配置不使用这些信息，因此只有使用为利用这种信息而特别创建的 PAM 配置时才需要考虑这个设置）。

### Note

如果 PAM 被设置为读取/etc/shadow，认证将会失败，因为 LightDB 服务器是由一个非 root 用户启动。然而，当 PAM 被配置为使用 LDAP 或其他认证验证方法时这就不是一个问题。

## 18.14. BSD 认证

这种认证方法操作起来类似于password，不过它使用 BSD 认证来验证口令。BSD 认证只被用来验证用户名/口令对。因此，在 BSD 认证可以被用于认证之前，用户的角色必须已经存在于数据库中。BSD 认证框架当前只在 OpenBSD 上可用。

LightDB中的 BSD 认证使用auth-postgresql登录类型，如果login.conf中定义了postgresql登录分类，就会用它来认证。默认情况下这种登录分类不存在，LightDB将使用默认的登录分类。

### Note

要使用 BSD 认证，LightDB 用户账号（也就是运行服务器的操作系统用户）必须首先被加入到auth组中。在 OpenBSD 系统上默认存在auth组。

## 18.15. 认证问题

认证失败以及相关的问题通常由类似下面的错误信息显示：

```
FATAL: no lt_hba.conf entry for host "123.123.123.123", user "andym", database "testdb"
```

这条消息最可能出现的情况是你成功地联系了服务器，但它不愿意和你说话。就像消息本身所建议的，服务器拒绝了连接请求，因为它没有在其lt\_hba.conf配置文件里找到匹配项。

```
FATAL: password authentication failed for user "andym"
```

这样的消息表示你联系了服务器，并且它也愿意和你交谈，但是你必须通过lt\_hba.conf文件中指定的认证方法。检查你提供的口令，或者如果错误消息提到了 Kerberos 或 ident 认证类型，检查那些软件。

```
FATAL: user "andym" does not exist
```

指示的数据库用户没有被找到。

```
FATAL: database "testdb" does not exist
```

你试图连接的数据库不存在。请注意如果你没有声明数据库名，默认会用数据库用户名作为数据库名，这可能正确也可能不正确。



Tip

服务器日志可能包含比报告给客户端的更多的有关认证失败的信息。如果你为失败的原因而困惑，那么请检查服务器日志。

---

## Chapter 19. 数据库角色

LightDB使用角色的概念管理数据库访问权限。一个角色可以被看成是一个数据库用户或者是一个数据库用户组，这取决于角色被怎样设置。角色可以拥有数据库对象（例如，表和函数）并且能够把那些对象上的权限赋予给其他角色来控制谁能访问哪些对象。此外，还可以把一个角色中的成员资格授予给另一个角色，这样允许成员角色使用被赋予给另一个角色的权限。

角色的概念把“用户”和“组”的概念都包括在内。在LightDB版本 8.1 之前，用户和组是完全不同的两种实体，但是现在只有角色。任意角色都可以扮演用户、组或者两者。

本章描述如何创建和管理角色。更多角色权限在多个数据库对象上的效果可以在Section 5. 中找到。

### 19.1. 数据库角色

数据库角色在概念上已经完全与操作系统用户独立开来。事实上可能维护一个对应关系会比较方便，但是这并非必需。数据库角色在一个数据库集簇安装范围内是全局的（而不是独立数据库内）。要创建一个角色，可使用CREATE ROLE SQL 命令：

```
CREATE ROLE name;
```

name遵循 SQL 标识符的规则：或是未经装饰没有特殊字符，或是用双引号包围（实际上，你将总是给该命令要加上额外选项，例如LOGIN。更多细节可见下文）。要移除一个已有的角色，使用相似的DROP ROLE命令：

```
DROP ROLE name;
```

为了方便，createuser和dropuser程序被提供作为这些 SQL 命令的包装器，它们可以从shell 命令行调用：

```
createuser name  
dropuser name
```

要决定现有角色的集合，检查pg\_roles系统目录，例如：

```
SELECT rolname FROM pg_roles;
```

ltsql程序的\du元命令也可以用来列出现有角色。

为了引导数据库系统，一个刚刚被初始化好的系统总是包含一个预定义角色。这个角色总是“superuser”，并且默认情况下（除非在运行initdb时修改）它的名字和初始化数据库集簇的操作系统用户相同。习惯上，这个角色将被命名为postgres。为了创建更多角色，你首先必须以初始角色的身份连接。

每一个到数据库服务器的连接都是使用某个特定角色名建立的，并且这个角色决定发起连接的命令的初始访问权限。要使用一个特定数据库连接的角色名由客户端指示，该客户端以一种应用相关的风格发起连接请求。例如，ltsql程序使用-U命令行选项来指定要以哪个角色连接。很多应用假定该名字默认是当前操作系统用户（包括createuser和ltsql）。因此在角色和操作系统用户之间维护一个名字对应关系通常是很方便的。

一个给定客户端连接能够用来连接的数据库角色的集合由该客户端的认证设置决定，这些在Chapter 18中有解释（因此，一个客户端不止限于以匹配其操作系统用户的角色连接，就像一个人的登录名不需要匹配她的真实名字一样）。因为角色身份决定一个已连接客户端可用的权限集合，在设置一个多用户环境时要小心地配置权限。

## 19.2. 角色属性

一个数据库角色可以有一些属性，它们定义角色的权限并且与客户端认证系统交互。

### login privilege

只有具有LOGIN属性的角色才能被用于一个数据库连接的初始角色名称。一个带有LOGIN属性的角色可以被认为和一个“数据库用户”相同。要创建一个带有登录权限的角色，使用两者之一：

```
CREATE ROLE name LOGIN;  
CREATE USER name;
```

（CREATE USER和CREATE ROLE等效，除了CREATE USER默认假定有LOGIN，而CREATE ROLE不这样认为）。

### superuser status

一个数据库超级用户会绕过所有权限检查，除了登入的权利。这是一个危险的权限并且应该小心使用，最好用一个不是超级用户的角色来完成你的大部分工作。要创建一个新数据库超级用户，使用CREATE ROLE name SUPERUSER。你必须作为一个超级用户来完成这些。

### database creation

一个角色必须被显式给予权限才能创建数据库（除了超级用户，因为它们会绕过所有权限检查）。要创建这样一个角色，使用CREATE ROLE name CREATEDB。

### role creation

一个角色必须被显式给予权限才能创建更多角色（除了超级用户，因为它们会绕过所有权限检查）。要创建这样一个角色，使用CREATE ROLE name CREATEROLE。一个带有CREATEROLE权限的角色也可以修改和删除其他角色，还可以授予或回收角色中的成员关系。然而，要创建、修改、删除或修改一个超级用户角色的成员关系，需要以超级用户的身份操作。CREATEROLE不足以完成这一切。

### initiating replication

一个角色必须被显式给予权限才能发起流复制（除了超级用户，因为它们会绕过所有权限检查）。一个被用于流复制的角色必须也具有LOGIN权限。要创建这样一个角色，使用CREATE ROLE name REPLICATION LOGIN。

### password

只有当客户端认证方法要求用户在连接数据库时提供一个口令时，一个口令才有意义。password和md5认证方法使用口令。数据库口令与操作系统命令独立。在角色创建时指定一个口令：CREATE ROLE name PASSWORD 'string'。

在创建后可以用ALTER ROLE修改一个角色属性。CREATE ROLE和ALTER ROLE命令的细节可见参考页。

### Tip

一个好习惯是创建一个具有CREATEDB和CREATEROLE权限的角色，而不是创建一个超级用户，并且然后用这个角色来完成对数据库和角色的例行管理。这种方法避免了在非必要时作为超级用户操作任务的风险。

对于Chapter 1中描述的运行时配置设置，一个角色也可以有角色相关的默认值。例如，如果出于某些原因你希望在每次连接时禁用索引扫描（提示：不是好主意），你可以使用：

```
ALTER ROLE myname SET enable_indexscan TO off;
```

这将保存设置（但是不会立刻设置它）。在这个角色的后续连接中，它就表现得像在会话开始之前执行过SET enable\_indexscan TO off。你也可以在会话期间改变该设置，它将只是作为默认值。要移除一个角色相关的默认设置，使用ALTER ROLE rolename RESET varname。注意附加到没有LOGIN权限的角色的角色相关默认值相当无用，因为它们从不会被调用。

## 19.3. 角色成员关系

把用户分组在一起来便于管理权限常常很方便：那样，权限可以被授予一整个组或从一整个组回收。在LightDB中通过创建一个表示组的角色来实现，并且然后将在该组角色中的成员关系授予给单独的用户角色。

要建立一个组角色，首先创建该角色：

```
CREATE ROLE name;
```

通常被用作一个组的角色不需要有LOGIN属性，不过如果你希望你也可以设置它。

一旦组角色存在，你可以使用GRANT 和 REVOKE commands:命令增加和移除成员：

```
GRANT group_role TO role1, ... ;  
REVOKE group_role FROM role1, ... ;
```

你也可以为其他组角色授予成员关系（因为组角色和非组角色之间其实没有任何区别）。数据库将不会让你设置环状的成员关系。另外，不允许把一个角色中的成员关系授予给PUBLIC。

组角色的成员可以以两种方式使用角色的权限。第一，一个组的每一个成员可以显式地做SET ROLE来临时“成为”组角色。在这种状态中，数据库会话可以访问组角色而不是原始登录角色的权限，并且任何被创建的数据库对象被认为属于组角色而不是登录角色。第二，有INHERIT属性的成员角色自动地具有它们所属角色的权限，包括任何组角色继承得到的权限。作为一个例子，假设我们已经有：

```
CREATE ROLE joe LOGIN INHERIT;  
CREATE ROLE admin NOINHERIT;  
CREATE ROLE wheel NOINHERIT;  
GRANT admin TO joe;  
GRANT wheel TO admin;
```

在作为角色joe连接后，一个数据库会话将立即拥有直接授予给joe的权限，外加任何授予给admin的权限，因为joe“继承了” admin的权限。然而，授予给wheel的权限不可用，因为即使joe是wheel的一个间接成员，但是该成员关系是通过带NOINHERIT属性的admin得到的。在：

```
SET ROLE admin;
```

之后，该会话将只拥有授予给admin的权限，但是没有授予给joe的权限。在执行：

```
SET ROLE wheel;
```

之后，该会话将只拥有授予给wheel的权限，但是没有授予给joe或admin的权限。初始的权限状态可以使用下面命令之一恢复：

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

#### Note

SET ROLE命令总是允许选择原始登录角色的直接或间接组角色。因此，在上面的例子中，在成为wheel之前不必先成为admin。

#### Note

在 SQL 标准中，用户和角色之间的区别很清楚，并且用户不会自动继承权限而角色会继承。这种行为在LightDB中也可以实现：为要用作 SQL 角色的角色给予INHERIT属性，而为要用作 SQL 用户的角色给予NOINHERIT属性。不过，为了向后兼容 8.1 以前的发布（在其中用户总是拥有它们所在组的权限），LightDB默认给所有的角色INHERIT属性。

角色属性LOGIN、SUPERUSER、CREATEDB和CREATEROLE可以被认为是一种特殊权限，但是它们从来不会像数据库对象上的普通权限那样被继承。要使用这些属性，你必须实际SET ROLE到一个有这些属性之一的特定角色。继续上述例子，我们可以选择授予CREATEDB和CREATEROLE给admin角色。然后一个以joe角色连接的会话将不会立即有这些权限，只有在执行了SET ROLE admin之后才会拥有。

要销毁一个组角色，使用DROP ROLE：

```
DROP ROLE name;
```

任何在该组角色中的成员关系会被自动撤销（但是成员角色不会受到影响）。

## 19.4. 删除角色

由于角色可以拥有数据库对象并且能持有访问其他对象的特权，删除一个角色常常并非一次DROP ROLE就能解决。任何被该用户所拥有的对象必须首先被删除或者转移给其他拥有者，并且任何已被授予给该角色的权限必须被收回。

对象的拥有关系可以使用REASSIGN OWNED命令一次转移出去，例如：

```
ALTER TABLE bobs_table OWNER TO alice;
```

此外，REASSIGN OWNED命令可以被用来把要被删除的角色所拥有的所有对象的拥有关系转移给另一个角色。由于 REASSIGN OWNED不能访问其他数据库中的对象，有必要在每一个包含该角色所拥有对象的数据库中运行该命令（注意第一个这样的 REASSIGN OWNED将更改任何在数据库间共享的该角色拥有的对象的拥有关系，即数据库或者表空间）。

一旦任何有价值的对象已经被转移给新的拥有者，任何由被删除角色拥有的剩余对象就可以用DROP OWNED命令删除。再次，由于这个命令不能访问其他数据库中的对象，有必要在每一个包含该角色所拥有对象的数据库中运行该命令。还有，DROP OWNED将不会删除整个数据库或者表空间，因此如果该角色拥有任何还没有被转移给新拥有者的数据库或者表空间，有必要手工删除它们。

DROP OWNED也会注意移除为不属于目标角色的对象授予给目标角色的任何特权。因为REASSIGN OWNED不会触碰这类对象，通常有必要运行REASSIGN OWNED和 DROP OWNED（按照这个顺序！）以完全地移除要被删除对象的从属物。

总之，移除曾经拥有过对象的角色方法是：

```
REASSIGN OWNED BY doomed_role TO successor_role;
DROP OWNED BY doomed_role;
-- 在集簇中的每一个数据库中重复上述命令
DROP ROLE doomed_role;
```

如果不是所有的拥有对象都被转移给了同一个后继拥有者，最好手工处理异常然后执行上述步骤直到结束。

如果在依赖对象还存在时尝试了DROP ROLE，它将发出消息标识哪些对象需要被重新授予或者删除。

## 19.5. 预定义角色

LightDB提供了一组预定义角色，它们提供对特定的、通常需要的、需要特权的功能和信息的访问。管理员（包括具有CREATEROLE权限的角色）可以把这些角色GRANT给其环境中的用户或者其他角色，让这些用户能够访问指定的功能和信息。

Table 19. 中描述了预定义的角色。注意由于额外功能的增加，每一种角色相关的权限可能会在未来被改变。管理员应该关注发行注册中提到的这方面的变化。

Table 19.1. 预定义角色

角色	允许的访问
pg_read_all_settings	读取所有配置变量，甚至是那些通常只对超级用户可见的变量。
pg_read_all_stats	读取所有的pg_stat_*视图并且使用与扩展相关的各种统计信息，甚至是那些通常只对超级用户可见的信息。
pg_stat_scan_tables	执行可能会在表上取得ACCESS SHARE锁的监控函数（可能会持锁很长时间）。
pg_monitor	读取/执行各种不同的监控视图和函数。这个角色是pg_read_all_settings, pg_read_all_stats和pg_stat_scan_tables的成员。
pg_signal_backend	发信号到其他后端以取消查询或中止它的会话。
pg_read_server_files	允许使用COPY以及其他文件访问函数从服务器上该数据库可访问的任意位置读取文件。
pg_write_server_files	允许使用COPY以及其他文件访问函数在服务器上该数据库可访问的任意位置中写入文件。
pg_execute_server_program	允许用运行该数据库的用户执行数据库服务器上的程序来配合COPY和其他允许执行服务器端程序的函数。

pg\_monitor、pg\_read\_all\_settings、pg\_read\_all\_stats和pg\_stat\_scan\_tables角色的目的是允许管理员能为监控数据库服务器的目的很容易地配置角色。它们授予一组常用的特权，这些特权允许角色读取各种有用的配置设置、统计信息以及通常仅限于超级用户的其他系统信息。

pg\_signal\_backend角色想要允许管理员启用受信任的、但是非超级用户的、发送信号给其他后端的角色。当前，此角色允许发送信号以取消另一个后端上的查询或终止其会话。不过授予此角色的用户不能向属于超级用户的后端发送信号。参见Section 9.27.2

`pg_read_server_files`、`pg_write_server_files`以及`pg_execute_server_program`角色的目的是允许管理员有一些可信但不是超级用户的角色来访问文件以及以运行数据库的用户在数据库服务器上运行程序。由于这些角色能够访问服务器文件系统上的任何文件，因此在直接访问文件时它们会绕过任何数据库级别的权限检查并且它们可以被用来得到超级用户级别的访问，因此在把这些角色授予给用户时应当特别小心。

在授予这些角色时应当非常小心，以确保它们只被用在需要的地方，并且要理解这些角色会授予对特权信息的访问。

管理员可以用GRANT命令把对这些角色的访问授予给用户，例如：

```
GRANT pg_signal_backend TO admin_user;
```

## 19.6. 函数和触发器安全性

函数、触发器以及行级安全性策略允许用户在后端服务器中插入代码，其他用户不会注意到这些代码的执行。因此，这些机制允许用户相对容易地为其他人设置“特洛伊木马”。最强的保护是严格控制哪些人能定义对象。如果做不到，则编写查询时应该只引用具有可信任拥有者的对象。可以从`search_path`中去除`public`方案以及任何其他允许不可信用户创建对象的方案。

在后端服务器进程中运行的函数带有数据库服务器守护进程的操作系统权限。如果用于函数的编程语言允许非检查的内存访问，它就可能改变服务器的内部数据结构。因此，在很多其他事情中，这些函数可能绕开任何系统访问控制。允许这种访问的函数语言被认为是“不可信的”，并且LightDB只允许超级用户创建用这些语言编写的函数。

---

## Chapter 20. 管理数据库

每个正在运行的LightDB服务器实例都管理着一个或多个数据库。因此，在组织SQL对象（“数据库对象”）的层次中，数据库位于最顶层。本章描述数据库的属性，以及如何创建、管理、删除它们。

### 20.1. 概述

少量的对象，例如角色、数据库和表空间名，是在集群级别定义并存储在pg\_global表空间之中的。集群内部有多个数据库，相互之间彼此隔离，但可以访问集群级对象。每个数据库内部都有多个模式，它们包含表和函数等对象。因此，完整的层级结构为：集群、数据库、模式、表（或一些其他类型的对象，如函数）。

当连接到数据库服务器时，客户端必须在它的连接请求中指定数据库名称。每个连接不可能访问多于一个数据库。但是，客户端可以对同一个数据库打开多个连接，或可以打开不同的数据库。数据库级别的安全有两个组成部分：访问控制（参见Section 18.1，在连接级进行管理，还有授权控制（参见Section 5.7，通过授权系统进行管理。外部数据包装器（参见postgres\_fdw）允许一个数据库中的对象作为其他数据库或集群中的对象的代理。旧的dblink模块（参见dblink）提供了类似的功能。默认情况下，所有用户可以使用所有连接方法连接所有的数据库。

如果一个LightDB服务器集群计划包含不相关的项目或用户，在很大程度上，彼此之间是不知道的，那么建议将它们放到单独的数据库中，并且调整相应的授权和访问控制。如果项目或用户是相互关联的，因此应该能够互相使用彼此的资源，它们将被放在同一个数据库中，但可能被放入相互独立的模式中；这提供了具有名称空间隔离和授权控制的模块化结构。关于模式管理的更多信息在Section 5.9中。

虽然可以在单个集群中创建多个数据库，但建议仔细考虑好处是否大于风险和限制。特别是，共享WAL（参见Chapter 27对备份和恢复选项的影响。从用户的角度来看，集群中的各个数据库是隔离的，但从数据库管理员的角度来看，它们是紧密绑定的。

数据库是使用CREATE DATABASE（见Section 20.2，并且用DROP DATABASE命令删除（见Section 20.5。要确定现有数据库的集合，可以检查系统目录pg\_database，例如

```
SELECT datname FROM pg_database;
```

ltsql程序的\l命令和-l命令行选项也可以用来列出已有的数据库。

#### Note

SQL标准把数据库称作“目录”，不过实际上没有区别。

### 20.2. 创建一个数据库

为了创建一个数据库，LightDB服务器必须启动并运行（见Section 16.3。

数据库用 SQL 命令CREATE DATABASE创建：

```
CREATE DATABASE name;
```

其中name遵循SQL标识符的一般规则。当前角色自动成为该新数据库的拥有者。以后删除这个数据库也是该拥有者的特权（同时还会删除其中的所有对象，即使那些对象有不同的拥有者）。



创建数据库是一个受限的操作。如何授权请见Section 19.2

因为你需要连接到数据库服务器来执行CREATE DATABASE命令，那么还有一个问题是任意给定站点的第一个数据库是怎样创建的？第一个数据库总是由initdb命令在初始化数据存储区域时创建的（见Section 16.2）。这个数据库被称为postgres。因此要创建第一个“普通”数据库时，你可以连接到postgres。

在数据库集簇初始化期间也会创建第二个数据库template1。当在集簇中创建一个新数据库时，实际上就是克隆了template1。这就意味着你对template1所做的任何修改都会体现在所有随后创建的数据库中。因此应避免在template1中创建对象，除非你想把它们传播到每一个新创建的数据库中。详见Section 20.3

为了方便，你还可以用一个程序来创建新数据库：createdb。

```
createdb dbname
```

createdb没什么神奇的。它连接到postgres数据库并且发出CREATE DATABASE命令，和前面介绍的完全一样。createdb参考页包含了调用细节。注意不带任何参数的createdb将创建一个使用当前用户名的数据库。

### Note

Chapter 19包含有关如何限制谁能连接到一个给定数据库的信息。

有时候你想为其他人创建一个数据库，并且使其成为新数据库的拥有者，这样他们就可以自己配置和管理这个数据库。要实现这个目标，使用下列命令之一：用于SQL环境的

```
CREATE DATABASE dbname OWNER rolename;
```

或者用于shell的

```
createdb -O rolename dbname
```

只有超级用户才被允许为其他人（即为一个你不是其成员的角色）创建一个数据库。

## 20.3. 模板数据库

CREATE DATABASE实际上通过拷贝一个已有数据库进行工作。默认情况下，它拷贝名为template1的标准系统数据库。所以该数据库是创建新数据库的“模板”。如果你为template1数据库增加对象，这些对象将被拷贝到后续创建的用户数据库中。这种行为允许对数据库中标准对象集合的站点本地修改。例如，如果你把过程语言PL/Perl安装到template1中，那么你在创建用户数据库后不需要额外的操作就可以使用该语言。

系统里还有名为template0的第二个标准系统数据库。这个数据库包含和template1初始内容一样的数据，也就是说，只包含你的LightDB版本预定义的标准对象。在数据库集簇被初始化之后，不应该对template0做任何修改。通过指示CREATE DATABASE使用template0取代template1进行拷贝，你可以创建一个“原始的”用户数据库（其中不存在用户定义的对象，并且系统对象没有被改变），它不会包含任何template1中的站点本地附加物。这一点在恢复一个lt\_dump转储时非常方便：转储脚本应该在一个原始的数据库中恢复以确保我们重建被转储数据库的正确内容，而不和任何现在可能已经被加入到template1中的附加物相冲突。

另一个从template0而不是template1复制的常见原因是，可以在复制template0时指定新的编码和区域设置，而一个template1的副本必须使用和它相同的设置。这是因为template1可能包含编码相关或区域相关的数据，而template0中没有。

要通过拷贝template0来创建一个数据库，使用：SQL 环境中的

```
CREATE DATABASE dbname TEMPLATE template0;
```

或者 shell 中的

```
createdb -T template0 dbname
```

可以创建额外的模板数据库，并且实际上可以通过将集群中任意数据库指定为CREATE DATABASE的模板来从该数据库拷贝。不过，我们必需明白，这个功能并不是设计作为一般性的“COPY DATABASE”功能。主要的限制是当源数据库被拷贝时，不能有其他会话连接到它。如果在CREATE DATABASE开始时存在任何其它连接，那么该命令将会失败。在拷贝操作期间，到源数据库的新连接将被阻止。

对于每一个数据库在pg\_database中存在两个有用的标志：

datistemplate和datallowconn列。datistemplate可以被设置来指示该数据库是不是要作为CREATE DATABASE的模板。如果设置了这个标志，那么该数据库可以被任何有CREATEDB权限的用户克隆；如果没有被设置，那么只有超级用户和该数据库的拥有者可以克隆它。如果datallowconn为假，那么将不允许与该数据库建立任何新的连接（但已有的会话不会因为把该标志设置为假而被中止）。template0通常被标记为datallowconn = false来阻止对它的修改。template0和template1通常总是被标记为datistemplate = true。

### Note

除了template1是CREATE DATABASE的默认源数据库名之外，template1和template0没有任何特殊的状态。例如，我们可以删除template1然后从template0重新创建它而不会有任何不良效果。如果我们不小心在template1中增加了一堆垃圾，那么我们会建议做这样的操作（要删除template1，它必须有pg\_database.datistemplate = false）。

当数据库集群被初始化时，也会创建postgres数据库。这个数据库用于做为用户和应用连接的默认数据库。它只是 template1的一个拷贝，需要时可以删除并重建。

## 20.4. 数据库配置

回顾一下Chapter 17 LightDB服务器提供了大量的运行时配置变量。你可以为其中的许多设置数据库相关的默认值。

例如，如果由于某种原因，你想禁用指定数据库上的GEQO优化器，正常情况下你不得不对所有数据库禁用它，或者确保每个连接的客户端小心地发出了SET geqo TO off。要令这个设置在一个特定数据库中成为默认值，你可以执行下面的命令：

```
ALTER DATABASE mydb SET geqo TO off;
```

这样将保存该设置（但不是立即设置它）。在后续建立的到该数据库的连接中它将表现得像在会话开始后马上调用SET geqo TO off;。注意用户仍然可以在该会话中更改这个设置，它只是默认值。要撤消这样的设置，使用ALTER DATABASE dbname RESET varname。

## 20.5. 销毁一个数据库

数据库用DROP DATABASE命令删除：

```
DROP DATABASE name;
```

只有数据库的拥有者或者超级用户才可以删除数据库。删除数据库会移除其中包括的所有对象。数据库的删除不能被撤销。

你不能在与目标数据库连接时执行DROP DATABASE命令。不过，你可以连接到任何其它数据库，包括 template1数据库。template1也是你删除一个给定集簇中最后一个用户数据库的唯一选项。

为了方便，有一个在 shell 程序可以删除数据库，dropdb:

```
dropdb dbname
```

(和createdb不同，删除当前用户名的数据库不是默认动作)。

## 20.6. 表空间

LightDB中的表空间允许数据库管理员在文件系统中定义用来存放表示数据库对象的文件的位置。一旦被创建，表空间就可以在创建数据库对象时通过名称引用。

通过使用表空间，管理员可以控制一个LightDB安装的磁盘布局。这么做至少有两个用处。首先，如果初始化集簇所在的分区或者卷用光了空间，而又不能在逻辑上扩展或者做别的什么操作，那么表空间可以被创建在一个不同的分区上，直到系统可以被重新配置。

其次，表空间允许管理员根据数据库对象的使用模式来优化性能。例如，一个很频繁使用的索引可以被放在非常快并且非常可靠的磁盘上，如一种非常贵的固态设备。同时，一个很少使用的或者对性能要求不高的存储归档数据的表可以存储在一个便宜但比较慢的磁盘系统上。

### Warning

即便是位于主要的 LightDB 数据目录之外，表空间也是数据库集簇的一部分并且不能被视作数据文件的一个自治集合。它们依赖于包含在主数据目录中的元数据，并且因此不能被附加到一个不同的数据库集簇或者单独备份。类似地，如果丢失一个表空间（文件删除、磁盘失效等），数据库集簇可能会变成不可读或者无法启动。把一个表空间放在一个临时文件系统（如一个内存虚拟盘）上会带来整个集簇的可靠性风险。

要定义一个表空间，使用CREATE TABLESPACE命令，例如：

```
CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';
```

这个位置必须是一个已有的空目录，并且属于LightDB操作系统用户。所有后续在该表空间中创建的对象都将被存放在这个目录下的文件中。该位置不能放在可移动或者瞬时存储上，因为如果表空间丢失会导致集簇无法工作。

### Note

通常在每个逻辑文件系统上创建多于一个表空间没有什么意义，因为你无法控制在一个逻辑文件系统中特定文件的位置。不过，LightDB不强制任何这样的限制，并且事实上它不会注意你的系统上的文件系统边界。它只是在你告诉它要使用的目录中存储文件。

表空间的创建本身必须作为一个数据库超级用户完成，但在创建完之后之后你可以允许普通数据库用户来使用它。要这样做，给数据库普通用户授予表空间上的CREATE权限。

表、索引和整个数据库都可以被分配到特定的表空间。想这么做，在给定表空间上有CREATE权限的用户必须把表空间的名字以一个参数的形式传递给相关的命令。例如，下面的命令在表空间space1中创建一个表：

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

另外，还可以使用default\_tablespace参数：

```
SET default_tablespace = space1;  
CREATE TABLE foo(i int);
```

当default\_tablespace被设置为非空字符串，那么它就为没有显式TABLESPACE子句的CREATE TABLE和CREATE INDEX命令提供一个隐式TABLESPACE子句。

还有一个temp\_tablespaces参数，它决定临时表和索引的位置，以及用于大数据集排序等目的的临时文件的位置。这可以是一个表空间名的列表，而不是只有一个。因此，与临时对象有关的负载可以散布在多个表空间上。每次要创建一个临时对象时，将从列表中随机取一个成员来存放它。

与一个数据库相关联的表空间用来存储该数据库的系统目录。此外，如果没有给出TABLESPACE子句并且没有在default\_tablespace或temp\_tablespaces（如适用）中指定其他选择，它还是在该数据库中创建的表、索引和临时文件的默认表空间。如果一个数据库被创建时没有指定表空间，它会使用其模板数据库相同的表空间。

当初始化数据库集簇时，会自动创建两个表空间。pg\_global表空间被用于共享系统目录。pg\_default表空间是template1和template0数据库的默认表空间（并且，因此也将是所有其他数据库的默认表空间，除非被一个CREATE DATABASE中的TABLESPACE子句覆盖）。

表空间一旦被创建，就可以被任何数据库使用，前提是请求的用户具有足够的权限。这也意味着，一个表空间只有在所有使用它的数据库中所有对象都被删除掉之后才可以被删掉。

要删除一个空的表空间，使用DROP TABLESPACE命令。

要确定现有表空间的集合，可检查pg\_tablespace 系统目录，例如

```
SELECT spcname FROM pg_tablespace;
```

ltsql程序的\db元命令也可以用来列出现有的表空间。

LightDB使用符号连接来简化表空间的实现。这就意味着表空间只能在支持符号连接的系统上使用。

\$LTDATA/lt\_tblspc目录包含指向集簇中定义的每个非内建表空间的符号连接。尽管我们不推荐，但还是可以通过手工重定义这些连接来调整表空间布局。在服务器运行时，绝不要这样做。注意在 LightDB 9.1 及更早的版本中，你将还需要用新位置更新pg\_tablespace目录（如果你不更新，lt\_dump将继续输出旧的表空间位置）。

---

# Chapter 21. 本地化

本章从管理员的角度描述可用的本地化特性。LightDB支持两种本地化方法：

- 利用操作系统的区域（locale）特性，提供对区域相关的排序顺序、数字格式、翻译过的信息和其它方面。这种方法在Section 21.1和Section 21.2中。
- 提供一些不同的字符集来支持存储所有种类语言的文本，并提供在客户端和服务端之间的字符集转换。这种方法在Section 21.3中。

## 21.1. 区域支持

区域支持指的是应用遵守文化偏好的问题，包括字母表、排序、数字格式等。LightDB使用服务器操作系统提供的标准 ISO C 和POSIX的区域机制。更多的信息请参考你的系统的文档。

### 21.1.1. 概述

区域支持是在使用initdb创建一个数据库集簇时自动被初始化的。默认情况下，initdb将按照它的执行环境的区域设置初始化数据库集簇；因此如果你的系统已经设置为你的数据库集簇想要使用的区域，那么你就没有什么可干的。如果你想使用其它的区域（或者你还不知道你的系统设置的区域是什么），那么你可以用--locale选项准确地告诉initdb你要用哪一个区域。比如：

```
initdb --locale=sv_SE
```

这个Unix系统上的例子把区域设置为瑞典（SE）瑞典语（sv）。其他的可能性包括 en\_US（美国英语）和fr\_CA（加拿大法语）。如果有多于一种字符集可以用于区域，那么声明可以采用如下的形式：language\_territory.codeset。例如fr\_BE.UTF-8表示在比利时（BE）讲的法语（fr），使用一个UTF-8字符集编码。

在你的系统上有哪些区域可用取决于操作系统提供商提供了什么以及安装了什么。在大部分Unix系统上，命令locale -a将会提供一个所有可用区域的列表。Windows使用一些更繁琐的区域名，例如German\_Germany或者Swedish\_Sweden.1252，但是其原则是相同的。

有时候，把几种区域规则混合起来也很有用，比如，使用英语排序规则而用西班牙语消息。为了支持这些，我们有一套区域子类用于控制本地化规则的某些方面：

LC_COLLATE	字符串排序顺序
LC_CTYPE	字符分类（什么是一个字符？它的大写形式是否等效？）
LC_MESSAGES	消息使用的语言Language of messages
LC_MONETARY	货币数量使用的格式
LC_NUMERIC	数字的格式
LC_TIME	日期和时间的格式

这些类名转换成initdb的选项名来覆盖某个特定分类的区域选择。比如，要把区域设置为加拿大法语，但使用 U.S. 规则格式化货币，可以使用initdb --locale=fr\_CA --lc-monetary=en\_US。

如果你想让系统表现得象没有区域支持，那么使用特殊的区域名C或者等效的POSIX。

一些区域分类的值必需在数据库被创建时的就被固定。你可以为不同的数据库使用不同的设置，但是一旦一个数据库被创建，你就不能在数据库上修改这些区域分类的值。LC\_COLLATE和LC\_CTYPE就是这样的分类。它们影响索引的排序顺序，因此它们必需保持固定，否则在文本列上的索引将会崩溃（但是你可以使用排序规则放松这种限制，讨论

见Section 21.2。这些分类的默认值在initdb运行时被确定，并且这些值在新数据库被创建时使用，除非在CREATE DATABASE命令中特别指定。

其它区域分类可以在任何时候被更改，更改的方式是设置与区域分类同名的服务器配置参数（详见Section 17.11.2。被initdb选中的值实际上只是被写入到配置文件lightdb.conf中作为服务器启动时的默认值。如果你将这些赋值从lightdb.conf中除去，那么服务器将会从其执行环境中继承该设置。

请注意服务器的区域行为是由它看到的环境变量决定的，而不是由任何客户端的环境变量影响的。因此，我们要在启动服务器之前认真地设置好这些变量。这样带来的一种后果是如果客户端和服务器设置成不同的区域，那么消息可能以不同的语言呈现，实际情况取决于它们的起源地。

### Note

在我们谈到从执行环境继承区域的时候，我们的意思是在大多数操作系统上的下列动作：对于一个给定的区域分类，比如排序规则，按照下面的顺序评估这些环境变量，直到找到一个被设置了的：LC\_ALL、LC\_COLLATE（或者对应于相应分类的变量）、LANG。如果这些环境变量一个都没有被设置，那么将区域缺省设置为C。

一些消息本地化库也查看环境变量LANGUAGE，它覆盖所有其它用于设置消息语言的区域设置。如果有疑问，请参考你的操作系统的文档，特别是有关gettext的文档。

要允许消息被翻译成用户喜欢的语言，编译时必需打开NLS（configure --enable-nls）。所有其他区域支持都会被自动编译。

## 21.1.2. 行为

区域设置特别影响下面的 SQL 特性：

- 在文本数据上使用ORDER BY或标准比较操作符的查询中的排序顺序
- 函数upper、lower和initcap
- 模式匹配操作符（LIKE、SIMILAR TO和POSIX风格的正则表达式）；区域影响大小写不敏感匹配和通过字符类正则表达式的字符分类
- to\_char函数家族
- 为LIKE子句使用索引的能力

LightDB中使用非C或非POSIX区域的缺点是性能影响。它降低了字符处理的速度并且阻止了在LIKE中对普通索引的使用。因此，只能在真正需要的时候才使用它。

作为允许LightDB在非C区域下为LIKE子句使用索引，有好几种自定义操作符类可用。这些操作符类允许创建一个执行严格按字符比较的索引。详见Section 11.10 另一种方法是创建使用C排序规则的索引，如Section 21.2所讨论的。

## 21.1.3. 问题

如果根据上面解释区域支持仍然不能运转，检查一下操作系统的区域支持是否被正确配置。要检查系统中安装了哪些区域，你可以使用命令locale -a（如果你的操作系统提供了该命令）。

请检查LightDB确实正在使用你认为它该用的区域设置。LC\_COLLATE和LC\_CTYPE设置都是在数据库创建时决定的，并且在除了创建数据库之外的操作中都不能被更改。其它的区域设置包括LC\_MESSAGES和LC\_MONETARY都是由服务器启动的环境决定的，但是可以在运行时修改。你可以用SHOW命令检查活跃的区域设置。

源代码目录的src/test/locale中包含LightDB的区域支持的测试套件。

那些通过分析错误消息来处理服务器端错误的客户端应用很明显会有问题，因为服务器来的消息可能会是以不同语言表示的。我们建议这类应用的开发人员改用错误代码机制。

维护消息翻译目录需要许多志愿者的坚持不懈的努力，他们希望LightDB以他们的语言说话。如果你的语言表示的消息目前还不可用或者没有完全翻译完成，那么我们很感谢你的协助。如果你想帮忙，那么请参考Chapter 5或者向开发者邮递列表发邮件。

## 21.2. 排序规则支持

排序规则特性允许指定每一列甚至每一个操作的数据的排序顺序和字符分类行为。这放松了数据库的LC\_COLLATE和LC\_CTYPE设置自创建以后就不能更改这一限制。

### 21.2.1. 概念

在概念上，一种可排序数据类型的每一种表达式都有一个排序规则（内建的可排序数据类型是text、varchar和char。用户定义的基础类型也可以被标记为可排序的，并且在一种可排序数据类型上的域也是可排序的）。如果该表达式是一个列引用，该表达式的排序规则就是列所定义的排序规则。如果该表达式是一个常量，排序规则就是该常量数据类型的默认排序规则。更复杂表达式的排序规则根据其输入的排序规则得来，如下所述：

一个表达式的排序规则可以是“默认”排序规则，它表示数据库的区域设置。一个表达式的排序规则也可能是不确定的。在这种情况下，排序操作和其他需要知道排序规则的操作会失败。

当数据库系统必须要执行一次排序或者字符分类时，它使用输入表达式的排序规则。这会在使用例如ORDER BY子句以及函数或操作符调用（如<）时发生。应用于ORDER BY子句的排序规则就是排序键的排序规则。应用于函数或操作符调用的排序规则从它们的参数得来，具体如下文所述。除比较操作符之外，在大小写字母之间转换的函数会考虑排序规则，例如lower、upper和initcap。模式匹配操作符和to\_char及相关函数也会考虑排序规则。

对于一个函数或操作符调用，其排序规则通过检查在执行指定操作时参数的排序规则来获得。如果该函数或操作符调用的结果是一种可排序的数据类型，万一有外围表达式要求函数或操作符表达式的排序规则，在解析时结果的排序规则也会被用作函数或操作符表达式的排序规则。

一个表达式的排序规则派生可以是显式或隐式。该区别会影响多个不同的排序规则出现在同一个表达式中时如何组合它们。当使用一个COLLATE子句时，将发生显式排序规则派生。所有其他排序规则派生都是隐式的。当多个排序规则需要被组合时（例如在一个函数调用中），将使用下面的规则：

1. 如果任何一个输入表达式具有一个显式排序规则派生，则在输入表达式之间的所有显式派生的排序规则必须相同，否则将产生一个错误。如果任何一个显式派生的排序规则存在，它就是排序规则组合的结果。
2. 否则，所有输入表达式必须具有相同的隐式排序规则派生或默认排序规则。如果任何一个非默认排序规则存在，它就是排序规则组合的结果。否则，结果是默认排序规则。
3. 如果在输入表达式之间存在冲突的非默认隐式排序规则，则组合被认为是具有不确定排序规则。这并非一种错误情况，除非被调用的特定函数要求提供排序规则的知识。如果它确实这样做，运行时将发生一个错误。

例如，考虑这个表定义：

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
);
```

然后在

```
SELECT a < 'foo' FROM test1;
```

中，<比较被根据de\_DE规则执行，因为表达式组合了一个隐式派生的排序规则和默认排序规则。但是在

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

中，比较被使用fr\_FR规则执行，因为显式排序规则派生重载了隐式排序规则。更进一步，给定

```
SELECT a < b FROM test1;
```

解析器不能确定要应用哪个排序规则，因为a列和b列具有冲突的隐式排序规则。由于<操作符不需要知道到底使用哪一个排序规则，这将会导致一个错误。该错误可以通过在一个输入表达式上附加一个显式排序规则说明符来解决，因此：

```
SELECT a < b COLLATE "de_DE" FROM test1;
```

或者等效的

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

在另一方面，结构相似的情况

```
SELECT a || b FROM test1;
```

不会导致一个错误，因为||操作符不关心排序规则：不管排序规则怎样它的结果都相同。

如果一个函数或操作符发送一个具有可排序数据类型的结果，分配给该函数或操作符的组合输入表达式的排序规则也被考虑应用在函数或操作符的结果。因此，在

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

中排序将根据de\_DE规则完成。但这个查询：

```
SELECT * FROM test1 ORDER BY a || b;
```

会导致一个错误，因为即使||操作符不需要知道排序规则，但ORDER BY子句需要。按照以前，冲突可以通过使用一个显式排序规则说明符来解决：

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

## 21.2.2. 管理排序规则

排序规则是SQL模式对象，它将SQL名称映射到操作系统中安装的库提供的语言环境。排序规则定义中有一个提供程序，它指定哪个库提供语言环境数据。一个标准的提供者名称是libc，它使用操作系统C库提供的语言环境。这些是操作系统提供的大多数工具使用的语言环境。另一个提供者是icu，它使用外部ICU库。只有在构建LightDB时配置了对ICU的支持，才能使用ICU区域设置。

libc提供的一个排序规则对象映射到LC\_COLLATE 和LC\_CTYPE设置的组合，如setlocale()系统库调用所接受的。（正如其名字所说的，一个排序规则的主要目的是设置LC\_COLLATE，



它控制排序顺序。但是在实际中LC\_CTYPE设置与LC\_COLLATE 不同是很少有必要的，因此通过一个概念来收集这些信息比为了设置每一个表达式的 LC\_CTYPE而创建另一种架构要更加方便。此外， 一个libc排序规则是和一个字符集编码（见Section 21.3 绑定在一起的。相同的排序规则名字可能存在于不同的编码中。

由icu提供的排序规则对象映射到由ICU库提供的指定整理器。 ICU不支持单独的“collate”和“ctype”设置， 所以它们总是相同的。此外， ICU排序规则与编码无关，因此在数据库中总是只有一个给定名称的ICU排序规则。

### 21.2.2.1. 标准的排序规则

在所有的平台上，名为default、C和POSIX的排序规则都可用。附加的排序规则是否可用取决于操作系统的支持。default排序规则选择在数据库创建时指定的LC\_COLLATE和LC\_CTYPE值。C和POSIX排序规则都指定了“传统的C”行为，在其中只有ASCII字母“A”到“Z”被视为字母，并且排序严格地按照字符编码的字节值完成。

此外，SQL标准排序规则名称ucs\_basic可用于编码UTF8。 它相当于C，并按Unicode代码点排序。

### 21.2.2.2. 预定义的排序规则

如果操作系统支持在一个程序中使用多个区域（newlocale和相关函数）， 或者配置了ICU支持，那么在一个数据集簇被初始化时，initdb 将以它在操作系统中能找到的所有区域为基础在系统目录pg\_collation 中填充排序规则。

要检查当前可用的语言环境，请在ltsql中使用查询 `SELECT * FROM pg_collation`或命令 `\dos +`。

#### 21.2.2.2.1. libc 排序规则

例如，操作系统可能会提供一个名为de\_DE.utf8的区域。initdb则会创建一个用于编码UTF8的名为de\_DE.utf8的排序规则，在其中LC\_COLLATE和LC\_CTYPE都被设置为de\_DE.utf8。它也会创建一个具有去掉名称的.utf8标签的排序规则。这样你也可以使用名字de\_DE来使用该排序规则，这写起来更简单并且使得名字更加独立于编码。不过要注意，最初的排序规则名称的集合是平台依赖的。

由libc提供的默认排序规则直接映射到操作系统中安装的语言环境， 可以使用命令 `locale -a`列出。如果所需的libc 排序规则与LC\_COLLATE和LC\_CTYPE的值不同， 或者在数据库系统初始化之后， 操作系统中安装了新的语言环境，可以使用CREATE COLLATION 命令创建新的排序规则。新的操作系统语言环境也可以使用 `pg_import_system_collations()` 函数集中导入。

在任何特定的数据库中，只有使用数据库编码的排序规则是令人感兴趣的。其他pg\_collation中的项会被忽略。 因此，一个如de\_DE的被剥离的排序规则名在一个给定数据库中可以被认为是唯一的，即使它在全局上并不唯一。 我们推荐使用被剥离的排序规则名，因为在你决定要更改到另一个数据库编码时需要做的事情更少。 但是要注意default、C和POSIX排序规则在使用时可以不考虑数据库编码。

LightDB在碰到具有相同属性的不同排序规则对象时会认为它们是不兼容的。因此对于例子：

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

将会得到一个错误，即使C和POSIX排序规则具有相同的行为。因此，我们不推荐混合使用被剥离的和非被剥离的排序规则名。

#### 21.2.2.2.2. ICU 排序规则

对于ICU，枚举所有可能的语言环境名称并不明智。 ICU为语言环境使用特定的命名系统，但命名语言环境的方法多于实际上不同的语言环境。 initdb使用ICU API提取一组不同的语

言环境以填充初始排序规则集合。由ICU提供的排序规则是在SQL环境中创建的，名称采用BCP 47语言标记格式，并附有一个“专用”扩展名-x-icu，以将它们与libc语言环境区分开来。

以下是可能创建的一些排序规则的示例：

de-x-icu

德语排序规则，默认变体

de-AT-x-icu

奥地利的德语排序规则，默认变体

(也就是说de-DE-x-icu 或de-CH-x-icu，但是这种写法，相当于 de-x-icu。)

und-x-icu (for “undefined”)

ICU “root” 排序规则。使用它获取合理的语言无关的排序顺序

一些（不常用的）编码不受ICU支持。当数据库编码是其中之一时，忽略pg\_collation中的ICU排序规则项。试图使用其中一个将会抛出一个类似“collation “de-x-icu” for encoding “WIN874” does not exist”的错误。

### 21.2.2.3. 创建新的排序规则对象

如果标准和预定义的排序规则不够用，用户可以使用SQL命令 CREATE COLLATION创建自己的排序规则对象。

与所有预定的对象一样，标准和预定义的排序规则在模式 pg\_catalog中。用户定义的排序规则应该在用户模式中创建。这也确保它们由lt\_dump保存。

#### 21.2.2.3.1. libc 排序规则

可以像这样创建新的libc排序规则：

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');
```

该命令中locale子句可接受的确切值取决于操作系统。在类Unix系统上，命令locale -a将显示一个列表。

由于预定义的libc排序规则已经包含了数据库实例初始化时在操作系统中定义的所有排序规则，因此通常不需要手动创建新排序规则。如果需要不同的命名系统（在这种情况下，另请参阅Section 21.2.2.3.3，或者操作系统已经升级以提供新的区域设置定义（在这种情况下，另请参阅pg\_import\_system\_collations()），可能需要手动创建。

#### 21.2.2.3.2. ICU 排序规则

ICU允许自定义超出由initdb预加载的基本语言+国家/地区集的排序规则。鼓励用户定义他们自己的排序规则对象，利用这些条件来满足他们排序行为的需求。请参阅<https://unicode-org.github.io/icu/userguide/locale/>和<https://unicode-org.github.io/icu/userguide/collation/api.html>获取有关ICU区域设置命名的信息。可接受的名称和属性集取决于特定的ICU版本。

这里有些例子：

```
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale = 'de-u-co-phonebk');
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale = 'de@collation=phonebook');
```

德语排序规则和电话簿排序规则类型

第一个例子使用“语言标签”根据 BCP 47选择了ICU区域设置。第二个示例使用传统的ICU特定区域设置语法。第一种风格是首选，但它不受旧版ICU支持。

请注意，您可以在SQL环境中任意指定排序规则对象的名称。在这个例子中，我们遵循预定义排序规则使用的命名风格，而这种风格又遵循BCP 47，但这对于用户定义的排序规则不是必需的。

```
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale = 'und-u-co-emoji');
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale = '@collation=emoji');
```

根据Unicode技术标准#51，使用表情符号排序规则类型的根排序规则

观察传统ICU区域命名系统中的方式，根区域设置由空字符串选择。

```
CREATE COLLATION latinlast (provider = icu, locale = 'en-u-kr-grek-latn');
CREATE COLLATION latinlast (provider = icu, locale = 'en@colReorder=grek-latn');
```

在拉丁字母之前对希腊字母进行排序。（默认为拉丁语在希腊语之前。）

```
CREATE COLLATION upperfirst (provider = icu, locale = 'en-u-kf-upper');
CREATE COLLATION upperfirst (provider = icu, locale = 'en@colCaseFirst=upper');
```

在小写字母前面排列大写字母。（默认是小写字母在前面。）

```
CREATE COLLATION special (provider = icu, locale = 'en-u-kf-upper-kr-grek-latn');
CREATE COLLATION special (provider = icu, locale = 'en@colCaseFirst=upper;colReorder=grek-latn');
```

结合上述两个选项。

```
CREATE COLLATION numeric (provider = icu, locale = 'en-u-kn-true');
CREATE COLLATION numeric (provider = icu, locale = 'en@colNumeric=yes');
```

数字排序，按数字值排序数字序列，例如：A-21 < A-123（也称为自然排序）。

参阅Unicode 技术标准 #35<sup>1</sup> 和BCP 47<sup>2</sup>获取详细信息。可能的排序规则类型（co子标签）列表可以在 CLDR 仓库<sup>3</sup>中找到。

请注意，虽然此系统允许创建“忽略大小写”或“忽略重音符”或类似（使用ks键）的排序规则，但为了使这些排序规则真正以大小写敏感或重音敏感方式工作，它们也需要在CREATE COLLATION中定义为非确定性的；见Section 21.2.2.4 否则，根据排序规则比较相等但按照字节不相等的任何字符串将根据其字节值进行排序。

### Note

根据设计，ICU几乎可以接受任何字符串作为区域名称，并使用其文档中描述的后备程序将其与最接近的区域设置相匹配。因此，如果使用给定ICU安装实际上不支持的功能组合排序规范，则不会有直接反馈。因此建议创建应用程序级别的测试用例，以检查排序规则定义是否满足需求。

#### 21.2.2.3.3. 复制排序规则

也可以使用命令CREATE COLLATION 从现有的排序规则创建新的排序规则，这对于能够在应用程序中使用与操作系统无关的排序规则名称、创建兼容性名称或以更易读的名称使用ICU提供的排序规则很有帮助。例如：

<sup>1</sup> <https://www.unicode.org/reports/tr35/tr35-collation.html>

<sup>2</sup> <https://tools.ietf.org/html/bcp47>

<sup>3</sup> <https://github.com/unicode-org/clldr/blob/master/common/bcp47/collation.xml>

```
CREATE COLLATION german FROM "de_DE";
CREATE COLLATION french FROM "fr-x-icu";
```

### 21.2.2.4. 非确定性排序规则

排序规则要么是确定性的，要么是非确定性的。确定性排序规则使用确定性比较方法，也就意味着字符串要认为是相等的话，只有在它们是由相同的字节顺序组成的时候。非确定性比较可以确定字符串相等，即使它们由不同字节组成。典型的情形包括大小写敏感比较、重音敏感比较和不同Unicode规范形式的字符串的比较。非敏感比较的实际是由排序规则提供程序来实现的；确定性标志仅确定使用按字节比较时关系是否要断开。也可参见Unicode技术标准10<sup>4</sup>获得更多术语信息。

创建非确定性排序规则时，指定属性deterministic = false以CREATE COLLATION，例如：

```
CREATE COLLATION ndcoll (provider = icu, locale = 'und', deterministic = false);
```

该例子以非确定性方式使用标准的Unicode排序规则。特别地，这将允许不同规范形式的字符串能正确地比较。更有趣的例子是使用上述的ICU定制化功能。例如：

```
CREATE COLLATION case_insensitive (provider = icu, locale = 'und-u-ks-level2',
    deterministic = false);
CREATE COLLATION ignore_accents (provider = icu, locale = 'und-u-ks-level1-kc-true',
    deterministic = false);
```

所有标准和预定义的排序规则是确定性的，所有用户自定义的排序规则缺省是确定性的。尽管非确定性规则提供了更“正确”的行为，尤其是考虑到Unicode和其许多特殊情况的全部能力时，也有一些缺点。首先，使用它们会导致性能下降。特别请注意，B-tree不能将重复数据消除与使用非确定性排序的索引一起使用。此外，用非确定性规则排序时某些操作就不可能做，例如模式匹配。因此，仅在特别需要它们的情况下才使用。

#### Tip

要处理不同 Unicode 规范化格式中的文本，还有一个选项，使用函数/表达式normalize 和 is normalized预处理或检查字符串，而不是使用非确定性排序规则。每种方法都有不同的权衡。

## 21.3. 字符集支持

LightDB里面的字符集支持你能够以各种字符集存储文本，包括单字节字符集，比如 ISO 8859 系列，以及多字节字符集，比如EUC（扩展 Unix 编码 Extended Unix Code）、UTF-8 和 Mule 内部编码。所有被支持的字符集都可以被客户透明地使用，但少数只能在服务器上使用（即作为一种服务器方编码）。默认的字符集是在使用 initdb初始化你的LightDB数据库集簇时选择的。在你创建一个数据库时可以重载它，因此你可能会有多个数据库并且每一个使用不同的字符集。

但是，一个重要的限制是每个数据库的字符集必须和数据库的LC\_CTYPE（字符分类）和LC\_COLLATE（字符串排序顺序）设置兼容。对于 C或POSIX环境，任何字符集都是允许的，但是对于其他libc提供的环境只有一种字符集可以正确工作（不过，在Windows上UTF-8编码可以和任何环境配合使用）。如果您配置了ICU支持，则ICU提供的区域设置可用于大多数服务器端编码，但不能用于所有服务器端编码。

<sup>4</sup> <https://www.unicode.org/reports/tr10>

## 21.3.1. 被支持的字符集

Table 21. 显示了LightDB中可用的字符集。

Table 21.1. LightDB Character Sets

名称	描述	语言	是否服务器端?	ICU?	字节/字符	别名
BIG5	Big Five	繁体中文	否	否	1 - 2	WIN950, Windows950
EUC_CN	扩展UNIX编码-中国	简体中文	是	是	1 - 3	
EUC_JP	扩展UNIX编码-日本	日文	是	是	1 - 3	
EUC_JIS_2004	扩展UNIX编码-日本, JIS X 0213	日文	是	否	1 - 3	
EUC_KR	扩展UNIX编码-韩国	韩文	是	是	1 - 3	
EUC_TW	扩展UNIX编码-台湾	繁体中文, 台湾话	是	是	1 - 3	
GB18030	国家标准	中文	是	否	1 - 4	
GBK	扩展国家标准	简体中文	是	否	1 - 2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	拉丁语/西里尔语	是	是	1	
ISO_8859_6	ISO 8859-6, ECMA 114	拉丁语/阿拉伯语	是	是	1	
ISO_8859_7	ISO 8859-7, ECMA 118	拉丁语/希腊语	是	是	1	
ISO_8859_8	ISO 8859-8, ECMA 121	拉丁语/希伯来语	是	是	1	
JOHAB	JOHAB	韩语	否	否	1 - 3	
KOI8R	KOI8-R	西里尔语 (俄语)	是	是	1	KOI8
KOI8U	KOI8-U	西里尔语 (乌克兰语)	是	是	1	
LATIN1	ISO 8859-1, ECMA 94	西欧	是	是	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	中欧	是	是	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	南欧	是	是	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	北欧	是	是	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	土耳其语	是	是	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	日耳曼语	是	是	1	ISO885910

名称	描述	语言	是否服务器端?	ICU?	字节/字符	别名
LATIN7	ISO 8859-13	波罗的海	是	是	1	ISO885913
LATIN8	ISO 8859-14	凯尔特语	是	是	1	ISO885914
LATIN9	ISO 8859-15	带欧罗巴和口音的LATIN1	是	是	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	罗马尼亚语	是	否	1	ISO885916
MULE_INTERNAL	Mule内部编码	多语种编辑器	是	否	1 - 4	
SJIS	Shift JIS	日语	否	否	1 - 2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	日语	否	否	1 - 2	
SQL_ASCII	未指定 (见文本)	任意	是	否	1	
UHC	统一韩语编码	韩语	否	否	1 - 2	WIN949, Windows949
UTF8	Unicode, 8-bit	所有	是	是	1 - 4	Unicode
WIN866	Windows CP866	西里尔语	是	是	1	ALT
WIN874	Windows CP874	泰语	是	否	1	
WIN1250	Windows CP1250	中欧	是	是	1	
WIN1251	Windows CP1251	西里尔语	是	是	1	WIN
WIN1252	Windows CP1252	西欧	是	是	1	
WIN1253	Windows CP1253	希腊语	是	是	1	
WIN1254	Windows CP1254	土耳其语	是	是	1	
WIN1255	Windows CP1255	希伯来语	是	是	1	
WIN1256	Windows CP1256	阿拉伯语	是	是	1	
WIN1257	Windows CP1257	波罗的海	是	是	1	
WIN1258	Windows CP1258	越南语	是	是	1	ABC, TCVN, TCVN5712, VSCII

并非所有的客户端API都支持上面列出的字符集。比如，LightDB的JDBC驱动就不支持MULE\_INTERNAL、LATIN6、LATIN8和LATIN10。

SQL\_ASCII设置与其他设置表现得相当不同。如果服务器字符集是SQL\_ASCII，服务器把字节值0 - 127根据ASCII标准解释，而字节值128 - 255则当作无法解析的字符。如果设置为SQL\_ASCII，就不会有编码转换。因此，这个设置基本不是用来声明所使用的指定编码，因为这个声明会忽略编码。在大多数情况下，如果你使用了任何非ASCII数据，那么使用SQL\_ASCII设置都是不明智的，因为LightDB将无法帮助你转换或者校验非ASCII字符。

LightDB支持的GB18030标准是2022版。如果在GBK/GB18030编码中含有0x40-0x7E这些特殊值，则不可以使用这些汉字去命名database、schema、relation和column。

## 21.3.2. 设置字符集

initdb为一个LightDB集簇定义缺省的字符集（编码）。比如：

```
initdb -E EUC_JP
```

把缺省字符集设置为EUC\_JP（用于日文的扩展Unix 编码）。如果你喜欢用长选项字符串，你可以用--encoding代替-E。如果没有给出-E或者--encoding选项，initdb会尝试基于指定的或者默认的区域判断要使用的合适编码。

你可以在数据库创建时指定一个非默认编码，提供的编码应和选择的区域兼容：

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr
korean
```

将创建一个使用EUC\_KR字符集和ko\_KR区域名为korean的数据库。另外一种实现方法是使用SQL命令：

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr'
LC_CTYPE='ko_KR.euckr' TEMPLATE=template0;
```

注意上述命令指定拷贝template0数据库。在拷贝任何其他数据库时，不能更改从源数据库得来的编码和区域设置，因为这可能会导致破坏数据。详见Section 20.3

数据库的编码存储在系统目录pg\_database中。你可以使用ltsql -l选项或者\l命令来看。

```
$ ltsql -l
```

List of databases					
Name	Owner	Encoding	Collation	Ctype	Access Privileges
cloaledb	hlinnaka	SQL_ASCII	C	C	
englishdb	hlinnaka	UTF8	en_GB.UTF8	en_GB.UTF8	
japanese	hlinnaka	UTF8	ja_JP.UTF8	ja_JP.UTF8	
korean	hlinnaka	EUC_KR	ko_KR.euckr	ko_KR.euckr	
postgres	hlinnaka	UTF8	fi_FI.UTF8	fi_FI.UTF8	
template0	hlinnaka	UTF8	fi_FI.UTF8	fi_FI.UTF8	{=c/
hlinnaka, hlinnaka=CTc/hlinnaka}					
templatel	hlinnaka	UTF8	fi_FI.UTF8	fi_FI.UTF8	{=c/
hlinnaka, hlinnaka=CTc/hlinnaka}					

(7 rows)

## Important

在大部分现代操作系统上，LightDB可以判断LC\_CTYPE设置意味着哪一种字符集，并且它强制只有匹配的数据库编码被使用。在老的系统上你需要自己负责确保所使用的编码就是你所选择的区域所期望的。在这里的一个错误很可能导致区域依赖的操作产生奇怪的行为，例如排序。

即使LC\_CTYPE不是C或POSIX时，LightDB将允许超级用户使用SQL\_ASCII编码创建数据库。正如前文所述，SQL\_ASCII并不强制存储在数据库中的数据具有任何特定的编码，并且这样这种选择存在着区域依赖的不正当行为的风险。使用这种设置组合的做法已经被废弃，并且在某天将被完全禁止。

### 21.3.3. 服务器和客户端之间的自动字符集转换

LightDB支持一些编码在服务器和前端之间的自动编码集转换, 用于许多字符集的组合 (Section 21.3.4显示了一些)。

要想启用自动字符集转换功能，你必须告诉LightDB你想在客户端使用的字符集（编码）。你可以用好几种方法来完成：

- 用ltsql里的\encoding命令。 \encoding允许你动态修改客户端编码。比如，把编码改变为SJIS，键入：

```
\encoding SJIS
```

- libpq (Section 32.10 中提供函数控制客户端编码。
- 使用SET client\_encoding TO。 可以使用这个SQL命令设置客户端编码：

```
SET CLIENT_ENCODING TO 'value';
```

你还可以把标准SQL语法里的SET NAMES用于这个目的：

```
SET NAMES 'value';
```

要查询当前客户端编码：

```
SHOW client_encoding;
```

要返回到缺省编码：

```
RESET client_encoding;
```

- 使用PGCLIENTENCODING。如果在客户端的环境里定义了PGCLIENTENCODING环境变量，那么在与服务器进行了连接后将自动选择客户端编码（这个设置随后可以用上文提到的任何其他方法重载）。
- 使用client\_encoding配置变量。如果client\_encoding变量被设置，那么在与服务器建立了连接之后，这个客户端编码将备自动选定（这个设置随后可以用上文提到的其他方法重载）。

假如无法进行一个特定字符的转换 — 假如你选的服务器编码是EUC\_JP而客户端是LATIN1，那么有些日文字符不能转换成LATIN1 — 将会报告一个错误。



如果客户端字符集定义成了SQL\_ASCII，那么编码转换会被禁用，不管服务器的字符集是什么都一样。（但是，如果服务器的字符集不是SQL\_ASCII，服务器仍将检查传入数据是否对该编码有效；所以最终的效果是客户端字符集和服务器的字符集是一样的。。和服务器一样，除非你的工作环境全部是ASCII数据，否则使用SQL\_ASCII是不明智的。

### 21.3.4. 可用的字符集转换

LightDB 允许在pg\_conversion系统目录中列出的转换函数里面的任何两个字符集之间进行转换。LightDB 带了一些预定义的转换，概括在Table 21.2中，并在Table 21.3中显示更详细的信息。你可以使用SQL命令CREATE CONVERSION建立一个新的转换。（要用于自动的客户端/服务器转换，该字符集组合的转换必须标记为“default”。

Table 21.2. 内置客户端/服务器字符集转换

服务器字符集	可用的客户端字符集
BIG5	不支持作为服务器编码
EUC_CN	EUC_CN, MULE_INTERNAL, UTF8
EUC_JP	EUC_JP, MULE_INTERNAL, SJIS, UTF8
EUC_JIS_2004	EUC_JIS_2004, SHIFT_JIS_2004, UTF8
EUC_KR	EUC_KR, MULE_INTERNAL, UTF8
EUC_TW	EUC_TW, BIG5, MULE_INTERNAL, UTF8
GB18030	GB18030, UTF8
GBK	GBK, UTF8
ISO_8859_5	ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	ISO_8859_6, UTF8
ISO_8859_7	ISO_8859_7, UTF8
ISO_8859_8	ISO_8859_8, UTF8
JOHAB	不支持作为服务器编码
KOI8R	KOI8R, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
KOI8U	KOI8U, UTF8
LATIN1	LATIN1, MULE_INTERNAL, UTF8
LATIN2	LATIN2, MULE_INTERNAL, UTF8, WIN1250
LATIN3	LATIN3, MULE_INTERNAL, UTF8
LATIN4	LATIN4, MULE_INTERNAL, UTF8
LATIN5	LATIN5, UTF8
LATIN6	LATIN6, UTF8
LATIN7	LATIN7, UTF8
LATIN8	LATIN8, UTF8
LATIN9	LATIN9, UTF8
LATIN10	LATIN10, UTF8
MULE_INTERNAL	MULE_INTERNAL, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8R, LATIN1 至 LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	不支持作为服务器编码
SHIFT_JIS_2004	不支持作为服务器编码
SQL_ASCII	任何（没有转换将被执行）
UHC	不支持作为服务器编码

服务器字符集	可用的客户端字符集
UTF8	全部支持的编码
WIN866	WIN866, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN1251
WIN874	WIN874, UTF8
WIN1250	WIN1250, LATIN2, MULE_INTERNAL, UTF8
WIN1251	WIN1251, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866
WIN1252	WIN1252, UTF8
WIN1253	WIN1253, UTF8
WIN1254	WIN1254, UTF8
WIN1255	WIN1255, UTF8
WIN1256	WIN1256, UTF8
WIN1257	WIN1257, UTF8
WIN1258	WIN1258, UTF8

Table 21.3. 所有内置字符集转换

Conversion Name <sup>a</sup>	源编码	目标编码
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL

## 本地化

Conversion Name <sup>a</sup>	源编码	目标编码
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
windows_1258_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN

## 本地化

Conversion Name <sup>a</sup>	源编码	目标编码
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_windows_1258	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL

Conversion Name <sup>a</sup>	源编码	目标编码
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

<sup>a</sup> 转换名称遵循标准命名模式：源编码的正式名称，所有非字母数字字符都替换为下划线，后跟\_to\_，后跟类似处理目标编码名称。因此，这些名称有时会偏离Table 21. 中显示的自定义编码名。

### 21.3.5. 进一步阅读

下面是学习各种类型的编码系统的好资源。

CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing

包含对EUC\_JP、EUC\_CN、EUC\_KR、EUC\_TW的详细解释。

<https://www.unicode.org/>

Unicode联盟的网站。

RFC 3629<sup>5</sup>

UTF-8（8-bit UCS/Unicode转换格式）在这里定义。

<sup>5</sup> <https://tools.ietf.org/html/rfc3629>

---

## Chapter 22. 日常数据库维护工作

和任何数据库软件一样，LightDB需要定期执行特定的任务来达到最优的性能。这里讨论的任务是必需的，但它们本质上是重复性的并且可以很容易使用cron脚本或Windows的任务计划程序等标准工具来自动进行。建立合适的脚本并检查它们是否成功运行是数据库管理员的职责。

一个显而易见的维护任务是定期创建数据的后备拷贝。如果没有一个最近的备份，你就不可能在灾难（磁盘失败、或在、错误地删除一个关键表等）后进行恢复。LightDB中的备份和恢复机制在Chapter 2中有详细的介绍。

另一种主要类型的维护任务是周期性地“清理”数据库。该活动在Section 22. 中讨论。与之相关，更新将被查询规划器使用的统计信息的活动将在Section 22. 1. 中讨论。

另一项需要周期性考虑的任务是日志文件管理。这在Section 22. 中讨论。

check\_postgres<sup>1</sup>可用于检测数据库的健康并报告异常情况。check\_postgres与Nagios和MRTG整合在一起，但也可以被单独运行。

相对于其他数据库管理系统，LightDB的维护量较低。但是，适当对这些任务加以注意将大有助于愉快和高效地使用该系统。

### 22. 1. 日常清理

LightDB数据库要求周期性的清理维护。对于很多安装，让自动清理守护进程来执行清理已经足够，如Section 22. 1. 所述。你可能需要调整其中描述的自动清理参数来获得最佳结果。某些数据库管理员会希望使用手动管理的VACUUM命令来对后台进程的活动进行补充或者替换，这通常使用cron或任务计划程序脚本来执行。要正确地设置手动管理的清理，最重要的是理解接下来几小节中讨论的问题。依赖自动清理的管理员最好也能略读该内容以帮助他们理解和调整自动清理。

#### 22. 1. 1. 清理的基础知识

LightDB的 VACUUM命令出于几个原因必须定期处理每一个表：

1. 恢复或重用被已更新或已删除行所占用的磁盘空间。
2. 更新被LightDB查询规划器使用的数据统计信息。
3. 更新可见性映射，它可以加速只用索引的扫描。
4. 保护老旧数据不会由于事务ID回卷或多事务ID回卷而丢失。

正如后续小节中解释的，每一个原因都将指示以不同的频率和范围执行VACUUM操作。

有两种VACUUM的变体：标准VACUUM和VACUUM FULL。VACUUM FULL可以收回更多磁盘空间但是运行起来更慢。另外，标准形式的VACUUM可以和生产数据库操作并行运行

（SELECT、INSERT、UPDATE和DELETE等命令将继续正常工作，但在清理期间你无法使用ALTER TABLE等命令来更新表的定义）。VACUUM FULL要求在其工作的表上得到一个ACCESS EXCLUSIVE 锁，因此无法和对此表的其他使用并行。因此，通常管理员应该努力使用标准VACUUM并且避免VACUUM FULL。

VACUUM会产生大量I/O流量，这将导致其他活动会话性能变差。可以调整一些配置参数来后台清理活动造成的性能冲击 — 参阅Section 17. 4. 4

#### 22. 1. 2. 恢复磁盘空间

---

<sup>1</sup> [https://bucardo.org/check\\_postgres/](https://bucardo.org/check_postgres/)

在LightDB中，一次行的UPDATE或DELETE不会立即移除该行的旧版本。这种方法对于从多版本并发控制（MVCC，见Chapter 13 获益是必需的：当旧版本仍可能对其他事务可见时，它不能被删除。但是最后，任何事务都不会再对一个过时的或者被删除的行版本感兴趣。它所占用的空间必须被回收来用于新行，这样可避免磁盘空间需求的无限制增长。这通过运行VACUUM完成。

VACUUM的标准形式移除表和索引中的死亡行版本并将该空间标记为可在未来重用。不过，它将不会把该空间交还给操作系统，除非在特殊的情况中表尾部的一个或多个页面变成完全空闲并且能够很容易地得到一个排他表锁。相反，VACUUM FULL通过把死亡空间之外的内容写成一个完整的新版本表文件来主动紧缩表。这将最小化表的尺寸，但是要花费较长的时间。它也需要额外的磁盘空间用于表的新副本，直到操作完成。

例行清理的一般目标是多做标准的VACUUM来避免需要VACUUM FULL。自动清理守护进程尝试这样工作，并且实际上永远不会发出VACUUM FULL。在这种方法中，其思想不是让表保持它们的最小尺寸，而是保持磁盘空间使用的稳定状态：每个表占用的空间等于其最小尺寸外加清理之间将使用的空间量。尽管VACUUM FULL可被用来把一个表收缩回它的最小尺寸并将该磁盘空间交还给操作系统，但是如果该表将在未来再次增长这样就没什么意义。因此，对于维护频繁被更新的表，适度运行标准VACUUM运行比少量运行VACUUM FULL要更好。

一些管理员更喜欢自己计划清理，例如在晚上负载低时做所有的工作。根据一个固定日程来做清理的难点在于，如果一个表有一次预期之外的更新活动尖峰，它可能膨胀得真正需要VACUUM FULL来回收空间。使用自动清理守护进程可以减轻这个问题，因为守护进程会根据更新活动动态规划清理操作。除非你的负载是完全可以预估的，完全禁用守护进程是不理智的。一种可能的折中方案是设置守护进程的参数，这样它将只对异常的大量更新活动做出反应，因而保证事情不会失控，而在负载正常时采用有计划的VACUUM来做批量工作。

对于那些不使用自动清理的用户，一种典型的方法是计划一个数据库范围的VACUUM，该操作每天在低使用量时段执行一次，并根据需要辅以在重度更新表上的更频繁的清理（一些有着极高更新率的安装会每几分钟清理一次它们的最繁忙的表）。如果你在一个集群中有多个数据库，别忘记VACUUM每一个，你会用得上vacuumdb程序。

### Tip

当一个表因为大量更新或删除活动而包含大量死亡行版本时，纯粹的VACUUM可能不能令人满意。如果你有这样一个表并且你需要回收它占用的过量磁盘空间，你将需要使用VACUUM FULL，或者CLUSTER，或者ALTER TABLE的表重写变体之一。这些命令重写该表的一整个新拷贝并且为它构建新索引。所有这些选项都要求ACCESS EXCLUSIVE 锁。注意它们也临时使用大约等于该表尺寸的额外磁盘空间，因为直到新表和索引完成之前旧表和索引都不能被释放。

### Tip

如果你有一个表，它的整个内容会被周期性删除，考虑用TRUNCATE而不是先用DELETE再用VACUUM。TRUNCATE会立刻移除该表的整个内容，而不需要一次后续的VACUUM或VACUUM FULL来回收现在未被使用的磁盘空间。其缺点是违背严格的 MVCC 语义。

## 22.1.3. 更新规划器统计信息

LightDB查询规划器依赖于有关表内容的统计信息来为查询产生好的计划。这些统计信息由ANALYZE命令收集，它除了直接被调用之外还可以作为VACUUM的一个可选步骤被调用。拥有适度准确的统计信息很重要，否则差的计划可能降低数据库性能。

自动清理守护进程如果被启用，当一个表的内容被改变得足够多时，它将自动发出ANALYZE命令。不过，管理员可能更喜欢依靠手动的ANALYZE操作，特别是如果知道一个表

上的更新活动将不会影响“感兴趣的”列的统计信息时。守护进程严格地按照一个被插入或更新行数的函数来计划ANALYZE，它不知道那是否将导致有意义的统计信息改变。

在分区和继承子表中更改的元组不会触发父表上的分析操作。如果父表为空或很少更改，则可能永远不会由自动清理程序处理，并且将不会收集继承树的统计信息。需要手动在父表上运行ANALYZE，以保持统计信息最新。

正如用于空间恢复的清理一样，频繁更新统计信息对重度更新的表更加有用。但即使对于一个重度更新的表，如果该数据的统计分布没有很大改变，也没有必要更新统计信息。一个简单的经验法则是考虑表中列的最大和最小值改变了多少。例如，一个包含行被更新时间的timestamp列将在行被增加和更新时有一直增加的最大值；这样一列将可能需要更频繁的统计更新，而一个包含一个网站上被访问页面 URL 的列则不需要。URL 列可以经常被更改，但是其值的统计分布的变化相对很慢。

可以在指定表上运行ANALYZE甚至在表的指定列上运行，因此如果你的应用需要，可以更加频繁地更新某些统计。但实际上，通常只分析整个数据库是最好的，因为它是一种很快的操作。ANALYZE对一个表的行使用一种统计的随机采样，而不是读取每一个单一行。

### Tip

尽管对每列的ANALYZE频度调整可能不是非常富有成效，你可能会发现值得为每列调整被ANALYZE收集统计信息的详细程度。经常在WHERE中被用到的列以及数据分布非常不规则的列可能需要比其他列更细粒度的数据直方图。见ALTER TABLE SET STATISTICS，或者使用default\_statistics\_target配置参数改变数据库范围的默认值。

还有，默认情况下关于函数的选择度的可用信息是有限的。但是，如果你创建一个统计对象或者使用函数的表达式索引，关于该函数的有用的统计信息将被收集，这些信息能够大大提高使用该表达式索引的查询计划的质量。

### Tip

自动清理守护进程不会为外部表发出ANALYZE命令，因为无法确定一个合适的频度。如果你的查询需要外部表的统计信息来正确地进行规划，比较好的方式是按照一个合适的时间表在那些表上手工运行ANALYZE命令。

### Tip

自动清理程序不会为分区表发出ANALYZE命令。只有当父表本身发生更改时，继承父表才会被分析 - 对子表的更改不会触发父表上的自动分析。如果您的查询需要父表的统计信息进行正确的规划，必须定期在这些表上运行手动的ANALYZE以保持统计信息最新。

## 22.1.4. 更新可见性映射

清理机制为每一个表维护着一个可见性映射，它被用来跟踪哪些页面只包含对所有活动事务（以及所有未来的事务，直到该页面被再次修改）可见的元组。这样做有两个目的。第一，清理本身可以在下一次运行时跳过这样的页面，因为其中没有什么需要被清除。

第二，这允许LightDB回答一些只用索引的查询，而不需要引用底层表。因为LightDB的索引不包含元组的可见性信息，一次普通的索引扫描会为每一个匹配的索引项获取堆元组，用来



检查它是否能被当前事务所见。另一方面，一次只用索引的扫描会首先检查可见性映射。如果它了解到在该页面上的所有元组都是可见的，堆获取就可以被跳过。这对大数据集很有用，因为可见性映射可以防止磁盘访问。可见性映射比堆小很多，因此即使堆非常大，可见性映射也可以很容易地被缓存起来。

## 22.1.5. 防止事务 ID 回卷失败

LightDB的 MVCC 事务语义依赖于能够比较事务 ID (XID) 数字：如果一个行版本的插入 XID 大于当前事务的 XID，它就是“属于未来的”并且不应该对当前事务可见。但是因为事务 ID 的尺寸有限 (32位)，一个长时间 (超过 40 亿个事务) 运行的集簇会遭受到事务 ID 回卷问题：XID 计数器回卷到 0，并且本来属于过去的事务突然间就变成了属于未来 — 这意味着它们的输出变成不可见。简而言之，灾难性的数据丢失 (实际上数据仍然在那里，但是如果你不能得到它也无济于事)。为了避免发生这种情况，有必要至少每 20 亿个事务就清理每个数据库中的每个表。

周期性的清理能够解决该问题的原因是，VACUUM会把行标记为冻结，这表示它们是被一个在足够远的过去提交的事务所插入，这样从 MVCC 的角度来看，效果就是该插入事务对所有当前和未来事务来说当然都是可见的。LightDB保留了一个特殊的 XID (FrozenTransactionId)，这个 XID 并不遵循普通 XID 的比较规则并且总是被认为比任何普通 XID 要老。普通 XID 使用模- $2^{32}$ 算法来比较。这意味着对于每一个普通 XID 都有 20 亿个 XID “更老”并且有 20 亿个“更新”，另一种解释的方法是普通 XID 空间是没有端点的环。因此，一旦一个行版本创建时被分配了一个特定的普通 XID，该行版本将成为接下来 20 亿个事务的“过去” (与我们谈论的具体哪个普通 XID 无关)。如果在 20 亿个事务之后该行版本仍然存在，它将突然变得好像在未来。要阻止这一切发生，被冻结行版本会被看成其插入 XID 为FrozenTransactionId，这样它们对所有普通事务来说都是“在过去”，而不管回卷问题。并且这样的行版本将一直有效直到被删除，不管它有多旧。

vacuum\_freeze\_min\_age控制在其行版本被冻结前一个 XID 值应该有多老。如果被冻结的行将很快会被再次修改，增加这个设置可以避免不必要的工作。但是减少这个设置会增加在表必须再次被清理之前能够流逝的事务数。

VACUUM使用可见性映射来确定必须扫描哪些表页面。通常，即使这些页面可能仍然具有旧的XID值的行版本，VACUUM也会跳过没有任何死行版本的页面。因此，普通的VACUUM不会总是冻结表中的每个旧行版本。定期地，VACUUM将执行一次主动清理，仅跳过那些既不包含死行又不包含任何未冻结的XID或MXID值的页面。vacuum\_freeze\_table\_age控制VACUUM何时执行此操作：如果自上次扫描以来经过的事务数大于vacuum\_freeze\_table\_age减去vacuum\_freeze\_min\_age，则扫描所有可见但未全部冻结的页面。将vacuum\_freeze\_table\_age设置为0会强制VACUUM对所有扫描都使用此更积极的策略。

一个表能保持不被清理的最长时间是 20 亿个事务减去VACUUM上次扫描全表时的vacuum\_freeze\_min\_age值。如果它超过该时间没有被清理，可能会导致数据丢失。要保证这不会发生，将在任何包含比autovacuum\_freeze\_max\_age配置参数所指定的年龄更老的XID 的未冻结行的表上调用自动清理 (即使自动清理被禁用也会发生)。

这意味着如果一个表没有被清理，大约每autovacuum\_freeze\_max\_age减去vacuum\_freeze\_min\_age事务就会在该表上调用一次自动清理。对那些为了空间回收目的而被正常清理的表，这是无关紧要的。然而，对静态表 (包括接收插入但没有更新或删除的表) 就没有为空间回收而清理的需要，因此尝试在非常大的静态表上强制自动清理的间隔最大化会非常有用。显然我们可以通过增加autovacuum\_freeze\_max\_age或减少vacuum\_freeze\_min\_age来实现此目的。

vacuum\_freeze\_table\_age的实际最大值是  $0.95 * \text{autovacuum\_freeze\_max\_age}$ ，高于它的设置将被上限到最大值。一个高于autovacuum\_freeze\_max\_age的值没有意义，因为不管怎样在那个点上都会触发一次防回卷自动清理，并且 0.95 的乘数为在防回卷自动清理发生之前运行一次手动VACUUM留出了一些空间。作为一种经验法则，vacuum\_freeze\_table\_age应当被设置成一个低于autovacuum\_freeze\_max\_age的值，留出一个足够的空间让一次被正常调度的VACUUM或一次被正常删除和更新活动触发的自动清理可以在这个窗口中被运行。将它设置得太接近可能导致防回卷自动清理，即使该表最近因为回收空间的目的被清理过，而较低的值将导致更频繁的全表扫描。

增加autovacuum\_freeze\_max\_age（以及和它一起的vacuum\_freeze\_table\_age）的唯一不足是数据库簇的lt\_xact和pg\_commit\_ts子目录将占据更多空间，因为它必须存储所有向后autovacuum\_freeze\_max\_age范围内的所有事务的提交状态和（如果启用了track\_commit\_timestamp）时间戳。提交状态为每个事务使用两个二进制位，因此如果autovacuum\_freeze\_max\_age被设置为它的最大允许值 20 亿，lt\_xact将会增长到大约 0.5 吉字节，pg\_commit\_ts大约20GB。如果这对于你的总数据库尺寸是微小的，我们推荐设置autovacuum\_freeze\_max\_age为它的最大允许值。否则，基于你想要允许lt\_xact和pg\_commit\_ts使用的存储空间大小来设置它（默认情况下 2 亿个事务大约等于lt\_xact的 50 MB存储空间，pg\_commit\_ts的2GB的存储空间）。

减小vacuum\_freeze\_min\_age的一个不足之处是它可能导致VACUUM做无用的工作：如果该行在被替换成FrozenXID之后很快就被修改（导致该行获得一个新的 XID），那么冻结一个行版本就是浪费时间。因此该设置应该足够大，这样直到行不再可能被修改之前，它们都不会被冻结。

为了跟踪一个数据库中最老的未冻结 XID 的年龄，VACUUM在系统表pg\_class和pg\_database中存储 XID 的统计信息。特别地，一个表的pg\_class行的relfrozenxid列包含被该表的上一次全表VACUUM所用的冻结截止 XID。该表中所有被有比这个截断 XID 老的普通 XID 的事务插入的行 都确保被冻结。相似地，一个数据库的pg\_database行的datfrozenxid列是出现在该数据库中的未冻结 XID 的下界——它只是数据库中每一个表的relfrozenxid值的最小值。一种检查这些信息的方便方法是执行这样的查询：

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');
```

```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

age列度量从该截断 XID 到当前事务 XID 的事务数。

VACUUM通常只扫描从上次清理后备修改过的页面，但是只有当全表被扫描时relfrozenxid才能被推进。当relfrozenxid比vacuum\_freeze\_table\_age个事务还老时、当VACUUM的FREEZE选项被使用时或当所有页面正好要求清理来移除死亡行版本时，全表将被扫描。当VACUUM扫描全表时，在它被完成后，age(relfrozenxid)应该比被使用的vacuum\_freeze\_min\_age设置略大（比在VACUUM开始后开始的事务数多）。如果在autovacuum\_freeze\_max\_age被达到之前没有全表扫描VACUUM在该表上被发出，将很快为该表强制一次自动清理。

如果出于某种原因自动清理无法从一个表中清除旧的 XID，当数据库的最旧 XID 和回卷点之间达到 4 千万个事务时，系统将开始发出这样的警告消息：

```
WARNING: database "mydb" must be vacuumed within 10985967 transactions
HINT: To avoid a database shutdown, execute a database-wide VACUUM in that
database.
```

（如该示意所建议的，一次手动的VACUUM应该会修复该问题；但是注意该次VACUUM必须由一个超级用户来执行，否则它将无法处理系统目录并且因而不能推进数据库的datfrozenxid）。如果这些警告被忽略，一旦距离回卷点只剩下 3 百万个事务时，该系统将会关闭并且拒绝开始任何新的事务：

```
ERROR: database is not accepting commands to avoid wraparound data loss in
database "mydb"
HINT: Stop the postmaster and vacuum that database in single-user mode.
```

1百万个事务的安全裕量是为了让管理员在没有数据丢失的情况下通过手动执行所需的VACUUM命令来恢复。然而，由于系统一旦进入安全关闭模式后将不会执行命令，唯一的方法是停止服务器并以单用户模式启动服务器以执行VACUUM。关闭模式在单用户模式下不起作用。有关使用单用户模式的详细信息，请参见lightdb参考页面。

### 22.1.5.1. 多事务和回卷

Multixact ID被用来支持被多个事务锁定的行。由于在一个元组头部 只有有限的空间可以用来存储锁信息，所以只要有多于一个事务并发地锁住一个行， 锁信息将使用一个“多个事务 ID”（或简称多事务 ID）来编码。任何特定 多事务 ID 中包括的事务 ID 的信息被独立地存储在lt\_multixact子 目录中，并且只有多事务 ID 出现在元组头部的xmax域中。和事务 ID 类似，多事务 ID 也是用一个 32 位计数器实现，并且也采用了相似的存储，这些都要 求仔细的年龄管理、存储清除和回卷处理。在每个多事务中都有一个独立的存储区域 保存成员列表，它也使用一个 32 位计数器并且也应被管理。

在一次VACUUM表扫描（部分或者全部）期间，任何比 vacuum\_multixact\_freeze\_min\_age 要老的多事务 ID 会被替换为一个不同的值，该值可以是零值、 一个单一事务 ID 或者一个更新的多事务 ID。 对于每一个表，pg\_class.relminmxid 存储了在该表任意元组中仍然存在的最老可能多事务 ID。如果这个值比 vacuum\_multixact\_freeze\_table\_age老， 将强制一次全表扫描。可以在 pg\_class.relminmxid 上使用mxid\_age() 来找到它的年龄。

全表VACUUM扫描（不管是什么导致它们）将为表推进该值。 最后，当所有数据库中的所有表被扫描并且它们的最老多事务值被推进， 较老的多事务的磁盘存储可以被移除。

作为一种安全设备，对任何多事务年龄 （详见 Section 22.1.5.1） 超过 autovacuum\_multixact\_freeze\_max\_age的表， 都将发生一次全表清理扫描。当多事务成员占用的存储超过 2GB 时，从那些具有最老多事务年龄的表开始，全表清理扫描也将逐步在所有表上进行。即使自动清理被 在名义上被禁用，也会发生这两种主动扫描。

### 22.1.6. 自动清理后台进程

LightDB有一个可选的但是被高度推荐的特性autovacuum，它的目的是自动执行VACUUM和ANALYZE命令。当它被启用时，自动清理会检查被大量插入、更新或删除元组的表。这些检查会利用统计信息收集功能，因此除非track\_counts被设置为true，自动清理不能被使用。在默认配置下，自动清理是被启用的并且相关配置参数已被正确配置。

“自动清理后台进程”实际上由多个进程组成。有一个称为 自动清理启动器的常驻后台进程， 它负责为所有数据库启动自动清理工作者进程。 启动器将把工作散布在一段时间上，它每隔 autovacuum\_naptime秒尝试在每个数据库中启动一个工作者 （因此，如果安装中有N个数据库，则每 autovacuum\_naptime/N秒将启动一个新的工作者）。 在同一时间只允许最多autovacuum\_max\_workers 个工作者进程运行。如果有超过autovacuum\_max\_workers个数据库需要被处理，下一个数据库将在第一个工作者结束后马上被处理。 每一个工作者进程将检查其数据库中的每一个表并且在需要时执行 VACUUM和/或ANALYZE。 可以设置log\_autovacuum\_min\_duration 来监控自动清理工作者的活动。

如果在一小段时间内多个大型表都变得可以被清理，所有的自动清理工作者可能都会被占用用来在一段长的时间内清理这些表。这将会造成其他的表和数据库无法被清理，直到一个工作者变得可用。对于一个数据库中的工作者数量并没有限制，但是工作者确实会试图避免重复已经被其他工作者完成的工作。注意运行着的工作者的数量不会被计入max\_connections或superuser\_reserved\_connections限制。

relfrozenxid值比autovacuum\_freeze\_max\_age事务年龄更大的表总是会被清理（本页表示这些表的冻结最大年龄被通过表的存储参数修改过，参见后文）。否则，如果从上次VACUUM以来失效的元组数超过“清理阈值”，表也会被清理。清理阈值定义为：

清理阈值 = 清理基本阈值 + 清理缩放系数 \* 元组数

其中清理基本阈值为autovacuum\_vacuum\_threshold， 清理缩放系数为autovacuum\_vacuum\_scale\_factor， 元组数为pg\_class.reltuples。

如果自上次清理以来插入的元组数量超过了定义的插入阈值，表也会被清理，该阈值定义为：

清理插入阈值 = 清理基础插入阈值 + 清理插入缩放系数 \* 元组数

清理插入基础阈值为`autovacuum_vacuum_insert_threshold`，清理插入缩放系数为`autovacuum_vacuum_insert_scale_factor`。这样的清理可以允许部分的表被标识为`all visible`，并且也可以允许元组被冻结，可以减小后续清理的工作需要。对于可以接收INSERT操作但是不能或几乎不能UPDATE/DELETE操作的表，可能会从降低表的`autovacuum_freeze_min_age`中受益，因为这可能允许元组在早期清理中被冻结。废弃元组的数量和插入元组的数量可从统计收集器中获得；它是一个半精确的计数，由每个UPDATE、DELETE 和 INSERT 操作进行更新。（它只是半精确的，因为一些信息可能会在重负载情况下丢失。）如果表的`relfrozenxid`值大于`vacuum_freeze_table_age`事务老的，执行一个主动的清理来冻结旧的元组，并推进`relfrozenxid`；否则，只有上次清理以后修改过的页面被扫描。

对于分析，也使用了一个相似的阈值：

分析阈值 = 分析基本阈值 + 分析缩放系数 \* 元组数

该阈值将与自从上次ANALYZE以来被插入、更新或删除的元组数进行比较。

临时表不能被自动清理访问。因此，临时表的清理和分析操作必须通过会话期间的SQL命令来执行。

自动清理程序无法访问临时表。因此，应通过会话SQL命令执行适当的清理和分析操作。

默认的阈值和缩放系数都取自于`lightdb.conf`，但是可以为每一个表重写它们(和许多其他自动清理控制参数)，详情参见Storage Parameters。如果一个设置已经通过一个表的存储参数修改，那么在处理该表时使用该值，否则使用全局设置。全局设置请参阅Section 17.10

当多个工作者运行时，在所有运行着的工作者之间自动清理代价延迟参数（参阅Section 17.4.4）是“平衡的”，这样不管实际运行的工作者数量是多少，对于系统的总体 I/O 影响总是相同的。不过，任何正在处理已经设置了每表`autovacuum_vacuum_cost_delay`或`autovacuum_vacuum_cost_limit`存储参数的表的工作者不会被考虑在均衡算法中。

`autovacuum`工作进程通常不会阻止其他命令。如果某个进程尝试获取与`autovacuum`持有的`SHARE UPDATE EXCLUSIVE`锁冲突的锁，则锁获取将中断该`autovacuum`。有关冲突的锁定模式，请参见Table 13.2 但是，如果`autovacuum`正在运行以防止事务ID回卷（即在`pg_stat_activity`视图中的`autovacuum`查询名以`(to prevent wraparound)`结尾），则`autovacuum`不会被自动中断。

### Warning

定期运行需要获取与`SHARE UPDATE EXCLUSIVE`锁冲突的锁的命令（例如ANALYZE）可能会让`autovacuum`始终无法完成。

## 22.2. 日常重建索引

在某些情况下值得周期性地使用REINDEX命令或一系列独立重构步骤来重建索引。

已经完全变成空的B树索引页面被收回重用。但是，还是有一种低效的空间利用的可能性：如果一个页面上除少量索引键之外的全部键被删除，该页面仍然被分配。因此，在这种每个

范围中大部分但不是全部键最终被删除的使用模式中，可以看到空间的使用是很差的。对于这样的使用模式，推荐使用定期重索引。

对于非B树索引可能的膨胀还没有很好地定量分析。在使用非B树索引时定期监控索引的物理尺寸是个好主意。

还有，对于B树索引，一个新建立的索引比更新了多次的索引访问起来要略快，因为在新建立的索引上，逻辑上相邻的页面通常物理上也相邻（这样的考虑目前并不适用于非B树索引）。仅仅为了提高访问速度也值得定期重索引。

REINDEX在所有情况下都可以安全和容易地使用。默认情况下，此命令需要一个ACCESS EXCLUSIVE锁，因此通常最好使用CONCURRENTLY选项执行它，该选项仅需要获取SHARE UPDATE EXCLUSIVE锁。

## 22.3. 日志文件维护

把数据库服务器的日志输出保存在一个地方是个好主意，而不是仅仅通过/dev/null丢弃它们。在进行问题诊断的时候，日志输出是非常宝贵的。不过，日志输出可能很庞大（特别是在比较高的调试级别上），因此你不会希望无休止地保存它们。你需要轮转日志文件，这样在一段合理的时间后会开始新的日志文件并且移除旧的。

如果你简单地把lightdb的stderr定向到一个文件中，你会得到日志输出，但是截断该日志文件的唯一方法是停止并重起服务器。这样做对于开发环境中使用的LightDB可能是可接受的，但是你肯定不想在生产环境上这么干。

一个更好的办法是把服务器的stderr输出发送到某种日志轮转程序里。我们有一个内建的日志轮转程序，你可以通过在lightdb.conf里设置配置参数logging\_collector为true的办法启用它。该程序的控制参数在Section 17.8.里描述。你也可以使用这种方法把日志数据捕捉成机器可读的CSV（逗号分隔值）格式。

另外，如果你已经使用的其他服务器软件中有一个外部日志轮转程序，你可能更喜欢使用它。比如，包含在Apache发布里的rotatelog工具就可以用于LightDB。要做到这一点，方法之一是把服务器的stderr用管道重定向到要用的程序。如果你用lt\_ctl启动服务器，那么stderr已经重定向到stdout，因此你只需要一个管道命令，比如：

```
lt_ctl start | rotatelog /var/log/pgsql_log 86400
```

您可以通过设置logrotate来收集由LightDB内置日志收集器生成的日志文件来组合这些方法。在这种情况下，日志收集器定义日志文件的名称和位置，而logrotate则定期归档这些文件。启动日志轮转时，logrotate必须确保应用程序将进一步的输出发送到新文件。这通常是通过postrotate脚本完成的，该脚本向应用程序发送SIGHUP信号，使其重新打开日志文件。在LightDB中，您可以使用logrotate选项运行lt\_ctl。服务器收到此命令后，服务器将切换到新的日志文件或重新打开现有文件，具体取决于日志记录配置（请参见Section 17.8.1）。

### Note

使用静态日志文件名时，如果达到最大打开文件数限制或发生文件表溢出，则服务器可能无法重新打开日志文件。在这种情况下，日志消息将发送到旧的日志文件，直到成功进行日志轮转为止。如果将logrotate配置为压缩日志文件并将其删除，则服务器可能会丢失此时间范围内记录的消息。为避免此问题，可以将日志收集器配置为动态分配日志文件名，并使用prerotate脚本忽略打开的日志文件。

另外一种生产级的管理日志输出的方法就是把它们发送给syslog，让syslog处理文件轮转。要利用这个工具，我们需要设置lightdb.conf里的log\_destination配置参数设置

为syslog（记录syslog日志）。然后在你想强迫syslog守护进程开始写入一个新日志文件的时候，你就可以发送一个 SIGHUP信号给它。如果你想自动进行日志轮转，可以配置logrotate程序处理来自syslog的日志文件。

不过，在很多系统上，syslog不是非常可靠，特别是在面对大量日志消息的情况下；它可能在你最需要那些消息的时候截断或者丢弃它们。另外，在Linux，syslog会把每个消息刷写到磁盘上，这将导致很差的性能（你可以在syslog配置文件里面的文件名开头使用一个“-”来禁用这种行为）。

请注意上面描述的所有解决方案关注的是在可配置的间隔上开始一个新的日志文件，但它们并没有处理对旧的、不再需要的日志文件的删除。你可能还需要设置一个批处理任务来定期地删除旧日志文件。另一种可能的的方法是配置日志轮转程序，让它循环地覆盖旧的日志文件。

pgBadger<sup>2</sup>是一个外部项目，它可以进行日志文件的深度分析。check\_postgres<sup>3</sup>可在重要消息出现在日志文件中时向Nagios提供警告，也可以探测很多其他的特别情况。

---

<sup>2</sup> <https://pgbadger.darold.net/>

<sup>3</sup> [https://bucardo.org/check\\_postgres/](https://bucardo.org/check_postgres/)

---

## Chapter 23. 备份和恢复

由于包含着有价值的数 据，LightDB数据库应当被定期地备份。虽然过程相当简单，但清晰地理解其底层技术和假设是非常重要的。

有三种不同的基本方法来备份LightDB数据：

- SQL转储
- 文件系统级备份
- 连续归档

每一种都有其优缺点，在下面的小节中将分别讨论。

### 23.1. SQL转储

SQL转储方法的思想是创建一个由SQL命令组成的文件，当把这个文件回馈给服务器时，服务器将利用其中的SQL命令重建与转储时状态一样的数据库。LightDB为此提供了工具`lt_dump`。这个工具的基本用法是：

```
lt_dump dbname > dumpfile
```

正如你所见，`lt_dump`把结果输出到标准输出。我们后面将看到这样做有什么用处。尽管上述命令会创建一个文本文件，`lt_dump`可以用其他格式创建文件以支持并行和细粒度的对象恢复控制。

`lt_dump`是一个普通的LightDB客户端应用（尽管是个相当聪明的东西）。这就意味着你可以在任何可以访问该数据库的远端主机上进行备份工作。但是请记住，`lt_dump`不会以任何特殊权限运行。具体说来，就是它必须要有你想备份的表的读权限，因此为了备份整个数据库你几乎总是必须以数据库超级用户来运行它（如果你没有足够的特权来备份整个数据库，你仍然可以使用诸如`-n schema`或`-t table`选项来备份该数据库中你能够访问的部分）。

要声明`lt_dump`连接哪个数据库服务器，使用命令行选项`-h host`和`-p port`。默认主机是本地主机或你的`LTHOST`环境变量指定的主机。类似地，默认端口是环境变量`LTPORT`或（如果`LTPORT`不存在）内建的默认值。（服务器通常有相同的默认值，所以还算方便。）

和任何其他LightDB客户端应用一样，`lt_dump`默认使用与当前操作系统用户名同名的数据库用户名进行连接。要使用其他名字，要么声明`-U`选项，要么设置环境变量`LTUSER`。请注意`lt_dump`的连接也要通过客户认证机制（在Chapter 18描述）。

`lt_dump`对于其他备份方法的一个重要优势是，`lt_dump`的输出可以很容易地在新版本的LightDB中载入，而文件级备份和连续归档都是极度的服务器版本限定的。`lt_dump`也是唯一可以将一个数据库传送到一个不同机器架构上的方法，例如从一个32位服务器到一个64位服务器。

由`lt_dump`创建的备份在内部是一致的，也就是说，转储表现了`lt_dump`开始运行时刻的数据库快照，且在`lt_dump`运行过程中发生的更新将不会被转储。`lt_dump`工作的时候并不阻塞其他的对数据库的操作。（但是会阻塞那些需要排它锁的操作，比如大部分形式的ALTER TABLE）

#### 23.1.1. 从转储中恢复

`lt_dump`生成的文本文件可以由`ltsql`程序读取。从转储中恢复的常用命令是：

```
ltsql dbname < dumpfile
```

其中dumpfile就是lt\_dump命令的输出文件。这条命令不会创建数据库dbname，你必须在执行ltsql前自己从template0创建（例如，用命令createdb -T template0 dbname）。ltsql支持类似lt\_dump的选项用以指定要连接的数据库服务器和要使用的用户名。参阅ltsql的手册获取更多信息。非文本文件转储可以使用lt\_restore工具来恢复。

在开始恢复之前，转储库中对象的拥有者以及在其上被授予了权限的用户必须已经存在。如果它们不存在，那么恢复过程将无法将对象创建成具有原来的所属关系以及权限（有时候这就是你所需要的，但通常不是）。

默认情况下，ltsql脚本在遇到一个SQL错误后会继续执行。你也许希望在遇到一个SQL错误后让ltsql退出，那么可以设置ON\_ERROR\_STOP变量来运行ltsql，这将使ltsql在遇到SQL错误后退出并返回状态3：

```
ltsql --set ON_ERROR_STOP=on dbname < dumpfile
```

不管怎样，你将只能得到一个部分恢复的数据库。作为另一种选择，你可以指定让整个恢复作为一个单独的事务运行，这样恢复要么完全完成要么完全回滚。这种模式可以通过向ltsql传递-l或--single-transaction命令行选项来指定。在使用这种模式时，注意即使是很小的一个错误也会导致运行了数小时的恢复被回滚。但是，这仍然比在一个部分恢复后手工清理复杂的数据库要更好。

lt\_dump和ltsql读写管道的能力使得直接从一个服务器转储一个数据库到另一个服务器成为可能，例如：

```
lt_dump -h host1 dbname | ltsql -h host2 dbname
```

### Important

lt\_dump产生的转储是相对于template0。这意味着在template1中加入的任何语言、过程等都会被lt\_dump转储。结果是，如果在恢复时使用的是一个自定义的template1，你必须从template0创建一个空的数据库，正如上面的例子所示。

一旦完成恢复，在每个数据库上运行ANALYZE是明智的举动，这样优化器就有有用的统计数据了，更多信息参见Section 22.1.1和Section 22.1.6 更多关于如何有效地向LightDB里装载大量数据的建议，请参考Section 14.4

## 23.1.2. 使用lt\_dumpall

lt\_dump每次只转储一个数据库，而且它不会转储关于角色或表空间（因为它们是集簇范围的）的信息。为了支持方便地转储一个数据库集簇的全部内容，提供了lt\_dumpall程序。lt\_dumpall备份一个给定集簇中的每一个数据库，并且也保留了集簇范围的数据，如角色和表空间定义。该命令的基本用法是：

```
lt_dumpall > dumpfile
```

转储的结果可以使用ltsql恢复：

```
ltsql -f dumpfile postgres
```

（实际上，你可以指定恢复到任何已有数据库名，但是如果你正在将转储载入到一个空集簇中则通常要用（postgres）。在恢复一个lt\_dumpall转储时常常需要具有数据库超级用户访



问权限，因为它需要恢复角色和表空间信息。如果你在使用表空间，请确保转储中的表空间路径适合于新的安装。

`lt_dumpall`工作时发出命令重新创建角色、表空间和空数据库，接着为每一个数据库运行 `lt_dump`。这意味着每个数据库自身是一致的，但是不同数据库的快照并不同步。

集簇范围的数据可以使用 `lt_dumpall` 的 `--globals-only` 选项来单独转储。如果在单个数据库上运行 `lt_dump` 命令，上述做法对于完全备份整个集簇是必需的。

### 23.1.3. 处理大型数据库

在一些具有最大文件尺寸限制的操作系统上创建大型的 `lt_dump` 输出文件可能会出现问题。幸运的是，`lt_dump` 可以写出到标准输出，因此你可以使用标准 Unix 工具来处理这种潜在的问题。有几种可能的方法：

使用压缩转储。你可以使用你喜欢的压缩程序，例如 `gzip`：

```
lt_dump dbname | gzip > filename.gz
```

恢复：

```
gunzip -c filename.gz | ltsql dbname
```

或者：

```
cat filename.gz | gunzip | ltsql dbname
```

使用 `split`。`split` 命令允许你将输出分割成较小的文件以便能够适应底层文件系统的尺寸要求。例如，让每一块的大小为 2 吉字节：

```
lt_dump dbname | split -b 2G - filename
```

恢复：

```
cat filename* | ltsql dbname
```

如果使用 GNU `split`，可能会把它和 `gzip` 一起使用：

```
lt_dump dbname | split -b 2G --filter='gzip > $FILE.gz'
```

它可以用 `zcat` 恢复。

使用 `lt_dump` 的自定义转储格式。如果 LightDB 所在的系统上安装了 `zlib` 压缩库，自定义转储格式将在写出数据到输出文件时对其压缩。这将产生和使用 `gzip` 时差不多大小的转储文件，但是这种方式的一个优势是其中的表可以被有选择地恢复。下面的命令使用自定义转储格式来转储一个数据库：

```
lt_dump -Fc dbname > filename
```

自定义格式的转储不是 `ltsql` 的脚本，只能通过 `lt_restore` 恢复，例如：

```
lt_restore -d dbname filename
```

详情请参阅lt\_dump和lt\_restore。

对于非常大型的数据库，你可能需要将split配合其他两种方法之一进行使用。

使用lt\_dump的并行转储特性。为了加快转储一个大型数据库的速度，你可以使用lt\_dump的并行模式。它将同时转储多个表。你可以使用-j参数控制并行度。并行转储只支持“目录”归档格式。

```
lt_dump -j num -F d -f out.dir dbname
```

你可以使用lt\_restore -j来以并行方式恢复一个转储。它只能适合于“自定义”归档或者“目录”归档，但不管归档是否由lt\_dump -j创建。

## 23.2. 文件系统级别备份

另外一种备份策略是直接复制LightDB用于存储数据库中数据的文件，Section 16. 解释了这些文件的位置。你可以采用任何你喜欢的方式进行文件系统备份，例如：

```
tar -cf backup.tar /usr/local/lightdb-x/data
```

但是这种方法有两个限制，使得这种方法不实用，或者说至少比lt\_dump方法差：

1. 为了得到一个可用的备份，数据库服务器必须被关闭。例如阻止所有连接的半路措施是不起作用的（部分原因是tar和类似工具无法得到文件系统状态的一个原子的快照，还有服务器内部缓冲的原因）。关于停止服务器的信息可以在Section 16. 中找到。不用说，在恢复数据之前你也需要关闭服务器。
2. 如果你已经深入地了解了数据库的文件系统布局的细节，你可能会感兴趣尝试通过相应的文件或目录来备份或恢复特定的表或数据库。这种方法也不会起作用，因为包含在这些文件中的信息只有配合提交日志文件（lt\_xact/\*）才有用，提交日志文件包含了所有事务的提交状态。一个表文件只有和这些信息一起才有用。当然也不可能只恢复一个表及相关的lt\_xact数据，因为这会导致数据库集簇中所有其他表变得无用。因此文件系统备份值适合于完整地备份或恢复整个数据库集簇。

另一种文件系统备份方法是创建一个数据目录的“一致快照”，如果文件系统支持此功能（并且你相信它的实现正确）。典型的过程是创建一个包含数据库的卷的“冻结快照”，然后从该快照复制整个数据目录（如上，不能是部分复制）到备份设备，最后释放冻结快照。即使在数据库服务器运行时，这种方式也有效。但是，以这种方式创建的备份保存的文件看起来就像数据库没有被正确关闭时的状态。因此，当你从备份数据上启动数据库服务器时，它会认为上一次的服务器实例崩溃了并尝试重放WAL日志。这不是问题，只是需要注意（当然WAL文件必须要包括在备份中）。你可以在拍摄快照之前执行一次CHECKPOINT以便节省恢复时间。

如果你的数据库跨越多个文件系统，可能没有任何方式可以对所有卷获得完全同步的冻结快照。例如，如果你的数据文件和WAL日志放置在不同的磁盘上，或者表空间在不同的文件系统中，可能没有办法使用快照备份，因为快照必须是同步的。在这些情况下，一定要仔细阅读你的文件系统文档以了解其对一致快照技术的支持。

如果没有可能获得同步快照，一种选择是将数据库服务器关闭足够长的时间以建立所有的冻结快照。另一种选择是执行一次连续归档基础备份（Section 23.3.2，因为这种备份对于备份期间发生的文件系统改变是免疫的。这要求在备份过程中允许连续归档，恢复时使用连续归档恢复（Section 23.3.4）。

还有一种选择是使用rsync来执行一次文件系统备份。其做法是先在数据库服务器运行时执行rsync，然后关闭数据库服务器足够长时间来做一次rsync --checksum（--checksum是必

需的，因为rsync的文件修改时间粒度只能精确到秒）。第二次rsync会比第一次快，因为它只需要传送相对很少的数据，由于服务器是停止的，所以最终结果将是一致的。这种方法允许在最小停机时间内执行一次文件系统备份。

注意一个文件系统备份通常会比一个SQL转储体积更大（例如lt\_dump不需要转储索引的内容，而是转储用于重建索引的命令）。但是，做一次文件系统备份可能更快。

## 23.3. 连续归档和时间点恢复（PITR）

在任何时间，LightDB在数据集簇目录的lt\_wal/子目录下都保持有一个预写式日志（WAL）。这个日志存在的目的是为了保证崩溃后的安全：如果系统崩溃，可以“重放”从最后一次检查点以来的日志项来恢复数据库的一致性。该日志的存在也使得第三种备份数据库的策略变得可能：我们可以把一个文件系统级别的备份和WAL文件的备份结合起来。当需要恢复时，我们先恢复文件系统备份，然后从备份的WAL文件中重放来把系统带到一个当前状态。这种方法比之前的方法管理起来更复杂，但是有其显著的优点：

- 我们不需要一个完美的一致性的文件系统备份作为开始点。备份中的任何内部不一致性将通过日志重放（这和崩溃恢复期间发生的并无显著不同）来修正。因此我们不需要文件系统快照功能，只需要tar或一个类似的归档工具。
- 由于我们可以结合一个无穷长的WAL文件序列用于重放，可以通过简单地归档WAL文件来达到连续备份。这对于大型数据库特别有用，因为在其中不方便频繁地进行完全备份。
- 并不需要一直重放WAL项一直到最后。我们可以在任何点停止重放，并得到一个数据库在当时的一致快照。这样，该技术支持时间点恢复：在得到你的基础备份以后，可以将数据库恢复到它在其后任何时间的状态。
- 如果我们连续地将一系列WAL文件输送给另一台已经载入了相同基础备份文件的机器，我们就得到了一个热后备系统：在任何时间点我们都能提出第二台机器，它差不多是数据库的当前副本。

### Note

lt\_dump和lt\_dumpall不会产生文件系统级别的备份，并且不能用于连续归档方案。这类转储是逻辑的并且不包含足够的信息用于WAL重放。

就简单的文件系统备份技术来说，这种方法只能支持整个数据库集簇的恢复，却无法支持其中一个子集的恢复。另外，它需要大量的归档存储：一个基础备份的体积可能很庞大，并且一个繁忙的系统将会产生大量需要被归档的WAL流量。尽管如此，在很多需要高可靠性的情况下，它是首选的备份技术。

要使用连续归档（也被很多数据库厂商称为“在线备份”）成功地恢复，你需要一个从基础备份时间开始的连续的归档WAL文件序列。为了开始，在你建立第一个基础备份之前，你应该建立并测试用于归档WAL文件的过程。对应地，我们首先讨论归档WAL文件的机制。

### 23.3.1. 建立WAL归档

抽象地说，一个运行中的LightDB系统产生一个无穷长的WAL记录序列。系统从物理上将这个序列划分成WAL段文件，通常是每个16MB（段尺寸在initdb期间可修改）。段文件会被分配一个数字名称以便反映它在整个抽象WAL序列中的位置。在没有使用WAL归档时，系统通常只创建少量段文件，并且通过重命名不再使用的段文件为更高的段编号来“回收”它们。系统假设内容位于最后一个检查点之前的段文件是无用的且可以被回收。

在归档WAL数据时，我们需要在每一段被填满时捕捉其内容，并且在段文件被回收重用之前保存该数据。依靠应用和可用的硬件，有很多不同的方法来“保存数据”：我们可以将段

文件拷贝到一个已挂载的位于另一台机器上的NFS目录，或者将它们写出到一个磁带驱动器（确保你有办法标识每个文件的原始文件名），或者将它们批量烧录到CD上，或者其他什么方法。为了向数据库管理员提供灵活性，LightDB不对如何归档做任何假设。取而代之的是，LightDB让管理员声明一个shell命令来拷贝一个完整的段文件到它需要去的地方。该命令可以简单得就是一个cp，或者它可以调用一个复杂的 shell 脚本 — 所有都由你决定。

要启用WAL归档，需设置wal\_level配置参数为replica或更高，设置archive\_mode为on，并且使用archive\_command配置参数指定一个shell命令。实际上，这些设置总是被放置在lightdb.conf文件中。在archive\_command中，%p会被将要归档的文件路径所替代，而%f只会被文件名所替代（路径名是相对于当前工作目录而言的，即集簇的数据目录）。如果你需要在命令中嵌入一个真正的%字符，可以使用%%。最简单的命令类似于：

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/
archivedir/%f' # Unix
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"' # Windows
```

它将把 WAL 段拷贝到目录/mnt/server/archivedir（这个只是一个例子，并非我们建议的方法，可能不能在所有系统上都正确运行）。在%p和%f参数被替换之后，实际被执行的命令看起来可能是：

```
test ! -f /mnt/server/archivedir/00000001000000A9000000065 && cp
lt_wal/00000001000000A9000000065 /mnt/server/archivedir/00000001000000A9000000065
```

对每一个将要被归档的新文件都会生成一个类似的命令。

归档命令将在运行LightDB服务器的同一个用户的权限下执行。因为被归档的一系列WAL 文件实际上包含你的数据库里的所有东西，所以你应该确保自己的归档数据不会被别人窥探；比如，归档到一个没有组或者全局读权限的目录里。

有一点很重要：当且仅当归档命令成功时，它才返回零退出。在得到一个零值结果之后，LightDB将假设该文件已经成功归档，因此它稍后将被删除或者被新的数据覆盖。但是，一个非零值告诉LightDB该文件没有被归档；因此它会周期性的重试直到成功。

当归档命令被一个信号(不是作为服务器关闭的一部分所用的SIGTERM)或者是退出状态超过125(例如命令没有发现的)的脚本的一个错误所终止时，归档进程会退出并且被postmaster重新启动。当这种情况下，pg\_stat\_archiver 中不会报告故障。

归档命令通常应该被设计成拒绝覆盖已经存在的归档文件。这是一个非常重要的安全特性，可以在管理员操作失误（比如把两个不同的服务器的输出发送到同一个归档目录）的时候保持你的归档的完整性。

我们建议你首先要测试你准备使用到归档命令，以保证它实际上不会覆盖现有的文件，并且在这种情况下它返回非零状态。以上Unix中的命令例子通过包含一个独立的test步骤来保证这一点。在某些Unix平台上，cp具有诸如-i的开关，可用来更简洁地完成这一切，但是在没有验证返回的退出状态正确之前你不能依赖它们（特别地，GNU的cp在使用-i时将已存在的目标文件返回状态零，这并不是我们所期望的行为）。

在设计你的归档环境时，请考虑一下如果归档命令不停失败会发生什么情况，因为有些情况要求操作者的干涉，或者是归档空间不够了。例如，如果你往磁带上写，但是没有自动换带机，那么就有可能发生这种情况；如果磁带满了，除非换磁带，否则任何事也做不了。你应该确保任何错误情况或者任何要求操作员干涉的情况都会被正确报告，这样才能迅速解决这些问题。否则lt\_wal/目录会不停地被WAL段文件填充，直到问题解决（如果包含lt\_wal/的文件系统被填满，LightDB将会做一次致命关闭。不会有未提交事务被丢失，但是数据库将会保持离线直到你释放一部分空间）。

归档命令的速度并不要紧，只要它能跟上你的服务器生成 WAL 数据的平均速度即可。即使归档进程稍微落后，正常的操作也会继续进行。如果归档进程慢很多，就会增加灾难发生的

时候丢失的数据量。这同时也意味着`lt_wal/`目录包含大量未归档的段文件，并且可能最后超出了可用磁盘空间。我们建议你监控归档进程，确保它是按照你的期望运转的。

在写自己的归档命令的时候，你应该假设被归档的文件名最长为64个字符并且可以包含ASCII字母、数字以及点的任意组合。我们不需要保持原始的相对路径（%p），但是有必要保持文件名（%f）。

请注意尽管WAL归档允许你恢复任何对你的LightDB数据库中数据所做的修改，但它不会恢复对配置文件的修改（即`lightdb.conf`、`lt_hba.conf`以及`lt_ident.conf`），因为这些文件都是手工编辑的，而不是通过SQL操作来编辑的。所以你可能会需要把你的配置文件放在一个日常文件系统备份过程可处理的位置。如何重定位配置文件请参阅Section 17.2

归档命令只会为完成的WAL段调用。因此如果你的服务器产生了一点点WAL流量（或者在产生时有宽松的周期），从一个事务完成到它被安全地记录在归档存储中之间将会有较长的延迟。要为未归档数据设置一个年龄限制，你可以设置`archive_timeout`来强制要求服务器按照其设定的频度切换到一个新的WAL段。注意由于强制切换而被归档的文件还是具有和完全归档的文件相同的长度。因此设置一个很短的`archive_timeout`是很不明智的——它会膨胀你的归档存储。将`archive_timeout`设置为1分钟左右通常是合理的。

同样，如果你希望确保一个刚刚完成的事务能被尽快归档，可以使用`pg_switch_wal`进行一次手动段切换。其他与WAL管理相关的使用函数在Table 9.86中列出。

如Section 14.4所述，当`wal_level`为`minimal`时，一些SQL命令被优化为避免记录WAL日志。在这些语句的其中之一执行过程中如果打开了归档或流复制，WAL中将不会包含足够的信息用于归档恢（崩溃恢复不受影响）。出于这个原因，`wal_level`只能在服务器启动时修改。但是，`archive_command`可以通过重载配置文件来修改。如果你希望暂时停止归档，一种方式是将`archive_command`设置为空串（''）。这将导致WAL文件积累在`lt_wal/`中，直到一个可用的`archive_command`被重新建立。

## 23.3.2. 制作一个基础备份

执行一次基础备份最简单的方法是使用`lt_basebackup`工具。它将会以普通文件或一个tar归档的方式创建一个基础备份。如果需要比`lt_basebackup`更高的灵活性，你也可以使用低级API（见Section 23.3.3来制作一个基础备份。

没有必要关心创建一个基础备份所需的时间。但是，如果你正常地运行停用了`full_page_writes`的服务器，你可能会注意到备份运行时的性能下降，因为`full_page_writes`在备份模式期间会被实际强制实施。

要使用备份，你将需要保留所有在文件系统备份期间及之后生成的WAL段文件。为了便于你做这些，基础备份过程会创建一个备份历史文件，它将被立刻存储到WAL归档区域。该文件以文件系统备份中你需要的第一个WAL段文件命名。例如，如果开始的WAL文件是`0000000100001234000055CD`，则备份历史文件将被命名为`0000000100001234000055CD.007C9330.backup`。（文件名的第二部分表明WAL文件中的一个准确位置，一般可以被忽略）。一旦你已经安全地归档了文件系统备份和在备份过程中被使用的WAL段文件（如备份历史文件中所指定的），所有名字在数字上低于备份历史文件中记录值的已归档WAL段对于恢复文件系统备份就不再需要了，并且可以被删除。但是你应该考虑保持多个备份集以绝对保证你能恢复你的数据。

备份历史文件是一个很小的文本文件。它包含你指定给`lt_basebackup`的标签字符串，以及备份的起止时间以及起止WAL段。如果你使用该标签来标识相关转储文件，则已归档的历史文件足以说明需要哪个转储文件进行恢复。

由于你不得不保存最后一次基础备份之后的所有归档WAL文件，基础备份之间的间隔通常应该根据你希望在归档WAL文件上花费的存储空间来设定。你也应该考虑你准备花多长时间来进行恢复，如果需要恢复——系统将不得不重放所有那些WAL段，如果这些WAL段覆盖了最后一次基础备份以后的很长时间，重放过程将会花费一些时间。

### 23.3.3. 使用低级API制作一个基础备份

使用低级API制作一个基础备份的过程比`lt_basebackup`方法要包含更多的步骤，但相对要更简单。很重要的一点是，这些步骤要按照顺序执行，并且在执行下一步之前要验证上一步是否成功。

可以用非排他或者排他的方法来制作低级基础备份。我们推荐非排他方法，而排他的方法已经被废弃并且最终将被去除。

#### 23.3.3.1. 制作一个非排他低级备份

非排他低级备份允许其他并发备份运行（既包括那些使用同样的 备份 API 开始的备份，也包括那些使用 `lt_basebackup`开始的备份）。

1. 确保WAL归档被启用且正在工作。
2. 作为一个具有运行 `pg_start_backup` 权利的用户（超级用户，或者被授予在该 函数上 EXECUTE 的用户）连接到服务器（不在乎是哪个数据库）并且发出命令：

```
SELECT pg_start_backup('label', false, false);
```

其中`label`是用来唯一标识这次备份操作的任意字符串。调用 `pg_start_backup`的连接必须被保持到备份结束，否则备份 将被自动中止。

默认情况下，`pg_start_backup`可能需要较长的时间完成。这是因为它会执行一个检查点，并且该检查点所需要的 I/O 将会分散到一段 显著的时间上，默认情况下是你的检查点间隔（见配置参数 `checkpoint_completion_target`）的一半。这通常 是你所想要的，因为它可以最小化对查询处理的影响。如果你想要尽可能快地 开始备份，请把第二个参数改成`true`，这将会发出一个立即的检查点并且使用尽可能多的I/O。

第三个参数为`false`会告诉`pg_start_backup` 开始一次非排他基础备份。

3. 使用任何趁手的文件系统备份工具（例如`tar`或者 `cpio`，不是`lt_dump` 或者`lt_dumpall`）执行备份。当你做这些 时，不需要也不值得停止正常的数据库操作。在这类备份期间要考虑的事情 请见小节Section 23.3.3.3
4. 在同一个连接中，发出命令：

```
SELECT * FROM pg_stop_backup(false, true);
```

这会终止备份模式。在主控机上，它还执行一次自动切换到下一个WAL段。在后备机上，它无法自动切换WAL段，因此用户可能希望在主控机上运行`pg_switch_wal`来执行一次手工切换。要做切换的原因是让在备份期间写入的最后一个WAL段文件能准备好被归档。

`pg_stop_backup`将返回一个具有三个值的行。这些域的 第二个应该被写入到该备份根目录中名为`backup_label`的 文件。第三个域应该被写入到一个名为`tablespace_map` 的文件，除非该域为空。这些文件对该备份正常工作来说是至关重要的，必须逐个字节的写入而不能被随意修改，可能需要采用二进制的方式打开文件。

5. 一旦备份期间活动的WAL段文件被归档，备份就完成了。由 `pg_stop_backup`的第一个返回值标识的文件是构成一个 完整备份文件集合所需的最后一个段。在主控机上，如果`archive_mode`被启用并且`wait_for_archive`参数为`true`，在最后一个段被归档之前`pg_stop_backup`都不会返回。在后备机上，为了让`pg_stop_backup`等待，`archive_mode`必须为`always`。从你已经配置好`archive_command`之后这些文件的 归档就会自动发生。在大部分情况下，这些归档会很快发生，但是建议你监 控你的归档系统确保没有延迟。如果归档进程由于归档命令的失败而落后， 它将会持续重试知道归档成功并且备份完成。如果你希望对 `pg_stop_backup`的执行给出一个时间限制，可以设置

一个合适的`statement_timeout`值，但要注意如果 `pg_stop_backup`因此而中止会致使备份可能失效。

如果备份进程监控并且确保备份所需的所有WAL段文件都被成功地归档，那么`wait_for_archive`参数（默认为`true`）可以被设置为`false`，以便`pg_stop_backup`在停止备份记录被写入到WAL后立即返回。默认情况下，`pg_stop_backup`将会等待，直至所有WAL都被归档，这种等待会花一段时间。这个选项必须被小心地使用：如果WAL归档没有被正确的监控，则备份可能没有包括所有的WAL文件并且因此将变得不完整和不可恢复。

### 23.3.3.2. 制作一个排他低级备份

#### Note

排他式备份方法已过时，应避免使用。在LightDB 9.6 之前，这是唯一可用的低级方法，但是现在 推荐所有用户升级他们的脚本来使用非排他备份。

一个排他备份的处理绝大部分都和非排他备份相同，但是在一些关键步骤上不同。这种备份只能在主控机上制作，并且不允许并发备份。而且，由于它像如下所述的那样创建一个备份标签文件，所以会阻止崩溃后主服务器的自动重启。另一方面，从备份或后备中删除该文件是常见的错误，会导致严重的数据损坏。如果需要用此方法，可使用下述步骤。

1. 确保 WAL 归档被启用且正常工作。
2. 作为一个具有运行 `pg_start_backup` 权利的用户（超级用户，或者被授予在该 函数上 EXECUTE 的用户）连接到服务器（不在乎是哪个数据库）并且发出命令：

```
SELECT pg_start_backup('label');
```

这里`label`是任何你希望用来唯一标识这个备份操作的字符串。 `pg_start_backup`在集簇目录中创建一个关于备份信息的 备份标签文件，也被称为`backup_label`，其中包括了开始时间和标签字符串。该函数也会在集簇目录中创建一个 名为`tablespace_map`的表空间映射文件，如果在`lt_tblspc/`中有一个或者多个表空间符号链接存在，该文件会包含它们的信息。如果你需要从备份中恢复，这两个文件对于备份的 完整性都至关重要。

默认情况下，`pg_start_backup`会花费很长时间来完成。这是因为它会执行一个检查点，而检查点所需要的I/O在相当一段时间内将会被传播，默认情况下这段时间是内部检查点间隔的一半（参见配置参数`checkpoint_completion_target`）。这通常是你所希望的，因为它能将对查询处理的影响最小化。如果你要尽快开始备份，可使用：

```
SELECT pg_start_backup('label', true);
```

这会使检查点尽可能快地被完成。

3. 使用任何方便的文件系统备份工具执行备份，例如`tar` 或`cpio`（不是`lt_dump` 或`lt_dumpall`）。在此期间，不需要也 不值得停止正常的数据库操作。在备份期间要考虑的事情可见 Section 23.3.3.3小节。

如上所述，如果服务器在备份期间崩溃，它可能无法重启，直至从`LTDATA`目录中手工地移除`backup_label`文件。注意到在恢复备份时永远不要删除`backup_label`文件非常重要，因为这会导致损坏。使用此方法时，关于何时适合删除该文件的困惑是导致数据损坏的常见原因；一定要确认仅在现有的主服务器上删除该文件，并且从未在构建后备或恢复备份时删除，即使你正在构建后备服务器，而后该后备服务器会提升为新的主服务器。

4. 再次以具有运行 `pg_stop_backup` 权利的用户（超级用户，或者已经被授予 该函数上 EXECUTE 的用户）连接到数据库并且发出命令：

```
SELECT pg_stop_backup();
```

这个函数将终止备份模式，并且执行一个自动切换到下一个WAL段。进行切换的原因是将在备份期间生成的最新WAL段文件安排为可归档。

- 一旦备份期间活动的WAL段文件被归档，你的工作就完成了。 `pg_stop_backup`的结果所标识的文件是构成一个完整备份 文件组所需的最新段。如果`archive_mode`被启用，直到最新段被归档`pg_stop_backup`都不会返回。由于你已经配置了 `archive_command`，这些文件的归档过程会自动发生。在 大部分情况下这会很快发生，但还是建议你监控你的归档系统来确保不会有 延迟。如果归档处理由于归档命令的错误而延迟，它会保持重试直到归档成功和备份完成。

当使用排他式备份时，绝对要确保备份最后阶段`pg_stop_backup`成功执行。即使备份本身失败，例如由于缺少磁盘空间，调用`pg_stop_backup`失败，这样就不确定服务器是否留在备份模式下，导致未来的备份失败，增加了`backup_label`存在期间重启失败的风险。

### 23.3.3.3. 备份数据目录

如果被拷贝的文件在拷贝过程中发生变化，某些文件系统备份工具会发出警告或错误。在建立一个活动数据库的基础备份时，这种情况是正常的，并非一个错误。然而，你需要确保你能够把它们和真正的错误区分开。例如，某些版本的`rsync`为“消失的源文件”返回一个独立的退出码，且你可以编写一个驱动脚本来将该退出码接受为一种非错误情况。同样，如果一个文件在被`tar`复制的过程中被截断，某些版本的GNU `tar`会返回一个与致命错误无法区分的错误代码。幸运的是，如果一个文件在备份期间被改变，版本为1.16及其后的GNU `tar`将会退出并返回1，而对于其他错误返回2。在版本1.23及其后的GNU `tar`中，你可以使用警告选项`--warning=no-file-changed --warning=no-file-removed`来隐藏相关的警告消息。

确认你的备份包含数据库集簇目录（例如`/usr/local/lightdb-x/data`）下的所有文件。如果你使用了不在此目录下的表空间，注意也把它们包括在内（并且确保你的备份将符号链接归档为链接，否则恢复过程将破坏你的表空间）。

不过，你应当从备份中忽略集簇的`lt_wal/`子目录中的文件。这种微小的调整是值得的，因为它降低了恢复时的错误风险。如果`lt_wal/`是一个指向位于集簇目录之外其他地方的符号链接就很容易安排了，这是一种出于性能原因的常见设置。你可能也希望排除`postmaster.pid`和`lightdb.opts`，它们记录了关于`postmaster`运行的信息，但与最终使用这个备份的`postmaster`无关（这些文件可能会使`lt_ctl`搞混淆）。

从备份中忽略集簇的`lt_replslot/`子目录中的文件通常也是个好主意，这样 主控机上存在的复制槽不会成为备份的一部分。否则，后续用该备份创建一个后备机可能会导致该 后备机上的WAL文件被无限期保留，并且在启用了热后备反馈的情况下可能导致主控机膨胀，因为使用 那些复制槽的客户端将继续连接到主控机（而不是后备机）并且继续更新其上的槽。即使该备份是要被 用来创建一个新的主控机，拷贝复制槽也不是特别有用，因为这些槽的内容在新主控机上线时很可能已 经过时。

目录`lt_dynshmem/`、`lt_notify/`、`lt_serial/`、`lt_snapshots/`、`lt_stat_tmp/`和`lt_subtrans/`的内容（但不是这些目录本身）可以从备份中省略，因为它们在`postmaster`启动时会被初始化。如果`stats_temp_directory`被设置并且位于数据目录中，则该目录的内容也可以被省略。

任何以`pgsql_tmp`开始的文件或目录都可以从备份中省略。这些文件在`postmaster`启动时会被移除，而目录将被根据需要重建。

只要找到名为`lt_internal.init`的文件，它就可以从备份中省略。这些文件包含关系缓冲数据，它们在恢复时总是会被重建。

备份标签文件包含你指定给 `pg_start_backup`的标签字符串，以及 `pg_start_backup`被运行的时刻和起始WAL文件的名字。 在发生混乱的情况下就可以在备份文件中查看并准确地决定



该转储文件来自于哪个备份会话。表空间映射文件包括存在于目录`lt_tblspc/`中的符号链接名称以及每一个符号链接的完整路径。这些文件不仅是为了供参考，它们的存在和内容对于系统恢复过程的正确操作是至关重要。

在服务器停止时也可以创建一个备份。在这种情况下，你显然不能使用`pg_start_backup`或`pg_stop_backup`，并且因此你只能依靠你自己的策略来跟踪哪个备份是哪个，以及相关WAL文件应该走回到什么程度。通常最好遵循上面的连续归档过程。

### 23.3.4. 使用一个连续归档备份进行恢复

好，现在最坏的情况发生了，你需要从你的备份进行恢复。这里是其过程：

1. 如果服务器仍在运行，停止它。
2. 如果你具有足够的空间，将整个集簇数据目录和表空间复制到一个临时位置，稍后你将用到它们。注意这种预防措施将要求在你的系统上有足够的空闲空间来保留现有数据库的两个拷贝。如果你没有足够的空间，你至少要保存集簇的`lt_wal`子目录的内容，因为它可能包含在系统垮掉之前还未被归档的日志。
3. 移除所有位于集簇数据目录和正在使用的表空间根目录下的文件和子目录。
4. 从你的文件系统备份中恢复数据库文件。注意它们要使用正确的所有权恢复（数据库系统用户，不是`root`！）并且使用正确的权限。如果你在使用表空间，你应该验证`lt_tblspc/`中的符号链接被正确地恢复。
5. 移除`lt_wal/`中的任何文件，这些是来自于文件系统备份而不是当前日志，因此可以被忽略。如果你根本没有归档`lt_wal/`，那么以正确的权限重建它。注意如果以前它是一个符号链接，请确保你也以同样的方式重建它。
6. 如果你有在第2步中保存的未归档WAL段文件，把它们拷贝到`lt_wal/`（最好是拷贝而不是移动它们，这样如果在开始恢复后出现问题你任然有未修改的文件）。
7. 设定`lightdb.conf`（见Section 17.5.4中的恢复配置设置，并且在集簇数据目录中创建一个`recovery.signal`文件。你可能还想临时修改`lt_hba.conf`来阻止普通用户在成功恢复之前连接。
8. 启动服务器。服务器将会进入到恢复模式并且进而根据需要读取归档WAL文件。恢复可能因为一个外部错误而被终止，可以简单地重新启动服务器，这样它将继续恢复。恢复过程结束后，服务器将删除`recovery.signal`（为了阻止以后意外地重新进入恢复模式），并且开始正常数据库操作。
9. 检查数据库的内容来确保你已经恢复到了期望的状态。如果没有，返回到第1步。如果一切正常，通过恢复`lt_hba.conf`为正常来允许用户连接。

所有这些的关键部分是设定一个恢复配置，它描述你希望如何恢复以及恢复要运行到什么程度。你绝对必须指定的是`restore_command`，它告诉LightDB如何获取归档WAL文件段。与`archive_command`相似，这也是一个shell命令字符串。它可以包含`%f`（将被期望的日志文件名替换）和`%p`（将被日志文件被拷贝的目标路径名替换）。（路径名是相对于当前工作目录的，即集簇的数据目录）。如果你需要在命令中嵌入一个真正的`%`字符，可以写成`%%`。最简单的命令类似于：

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

它将从目录`/mnt/server/archivedir`中拷贝之前归档的WAL段。当然，你可以使用更复杂的，甚至是一个要求操作者装载合适磁带的shell脚本。

重要的是命令在失败时返回非零退出状态。该命令将被调用来请求不在归档中的文件，在这种情况下它应该返回非零值。这不是一种错误情况。一种例外是该命令被一个信号（除了

被用作数据库服务器关闭动作一部分的SIGTERM) 终止或者被shell的错误 (例如命令未找到) 终止, 那样恢复将中止并且服务器将不会启动。

并非所有被请求的文件都是WAL段文件, 你也许还会请求一些具有.history后缀的文件。还要注意的, %p路径的基本名字将会和%f 不同, 但不要期望它们可以互换。

归档中找不到的WAL段可以在lt\_wal/中看到, 这使得可以使用最近未归档的段。但是, 在归档中可用的段将会被优先于lt\_wal/中的文件被使用。

通常, 恢复将会处理完所有可用的WAL段, 从而将数据库恢复到当前时间点 (或者尽可能接近给定的可用WAL段)。因此, 一个正常的恢复将会以一个“文件未找到”消息结束, 错误消息的准确文本取决于你选择的restore\_command。你也可能在恢复的开始看到一个针对名称类似于00000001.history文件的错误消息。这也是正常的并且不表示在简单恢复情况中的问题, 对此的讨论见Section 23.3.5

如果你希望恢复到之前的某个时间点 (例如, 恢复到幼稚的DBA丢弃了你主要的交易表之前), 只需要指定要求的停止点。你可以使用日期/时间、命名恢复点或一个指定事务ID的结束时间来定义停止点 (也被称为“恢复目标”)。在这种写法中, 只有日期/时间和命名恢复点选项非常有用, 因为没有工具可以帮助你准确地确定要用哪个事务ID。

### Note

停止点必须位于基础备份的完成时间之后, 即pg\_stop\_backup的完成时间。在备份过程中你不能使用基础备份来恢复 (要恢复到这个时间, 你必须回到你之前的基础备份并且从这里开始前滚)。

如果恢复找到被破坏的WAL数据, 恢复将会停止于该点并且服务器不会启动。在这种情况下, 恢复进程需要从开头重新开始运行, 并指定一个在损坏点之前的“恢复目标”以便恢复能够正常完成。如果恢复由于一个外部原因失败, 例如一个系统崩溃或者WAL归档变为不可访问, 则那次恢复可以被简单地重启并且它将会从几乎是上次失败的地方继续。恢复重启工作起来很像普通操作时的检查点: 服务器周期性地强制把它的所有状态写到磁盘中, 然后更新lt\_control文件来说明已经处理过的WAL数据, 这样它们就不会被再次扫描。

## 23.3.5. 时间线

将数据库恢复到一个之前的时间点的能力带来了一些复杂性, 这和有关时间旅行和平行宇宙的科幻小说有些相似。例如, 在数据库的最初历史中, 假设你在周二晚上5:15时丢弃了一个关键表, 但是一直到周三中午才意识到你的错误。不用苦恼, 你取出你的备份, 恢复到周二晚上5:14的时间点, 并上线运行。在数据库宇宙的这个历史中, 你从没有丢弃该表。但是假设你后来意识到这并非一个好主意, 并且想回到最初历史中周三早上的某个时间。你没法这样做, 在你的数据库在线运行期间, 它重写了某些WAL段文件, 而这些文件本来可以将你引向你希望回到的时间。因此, 为了避免出现这种状况, 你需要将完成时间点恢复后生成的WAL记录序列与初始数据库历史中产生的WAL记录序列区分开来。

要解决这个问题, LightDB有一个时间线概念。无论何时当一次归档恢复完成, 一个新的时间线被创建来标识恢复之后生成的WAL记录序列。时间线ID号是WAL段文件名的一部分, 因此一个新的时间线不会重写由之前的时间线生成的WAL数据。实际上可以归档很多不同的时间线。虽然这可能看起来是一个无用的特性, 但是它常常扮演救命稻草的角色。考虑到你不太确定需要恢复到哪个时间点的情况, 你可能不得不做多次时间点恢复尝试和错误, 直到最终找到从旧历史中分支出去的最佳位置。如果没有时间线, 该处理将会很快生成一堆不可管理的混乱。而有了时间线, 你可以恢复到任何之前的状态, 包括早先被你放弃的时间线分支中的状态。

每次当一个新的时间线被创建, LightDB会创建一个“时间线历史”文件, 它显示了新时间线是什么时候从哪个时间线分支出来的。系统在从一个包含多个时间线的归档中恢复时, 这些历史文件对于允许系统选取正确的WAL段文件非常必要。因此, 和WAL段文件相似, 它们也要被归档到WAL归档区域。历史文件是很小的文本文件, 因此将它们无限期地保存起来的代

价很小，而且也是很合适的（而段文件都很大）。如果你喜欢，你可以在一个历史文件中增加注释来记录如何和为什么要创建该时间线。当你由于试验的结果拥有了一大堆错综复杂的不同时间线时，这种注释将会特别有价值。

恢复的默认行为是沿着归档中所发现的最新的时间线进行恢复。如果你希望恢复到执行基础备份时的当前时间线或者某个指定的子时间线（即，你希望回到在一次恢复尝试后产生的某个状态），你需要指定`current`或者在`recovery_target_timeline`中指定目标时间线ID。你不能恢复到早于该基础备份之前分支出去的时间线。

## 23.3.6. 建议和例子

这里将给出一些配置连续归档的建议。

### 23.3.6.1. 单机热备份

可以使用LightDB的备份功能来产生单机热备份。这些备份不能被用于时间点恢复，然而备份和恢复时要比使用`lt_dump`转储更快（它们也比`lt_dump`转储更大，所以在某些情况下速度优势可能会被否定）。

在基础备份的帮助下，产生一个单机热备份最简单的方式是使用`lt_basebackup`工具。如果你在调用它时使用了`-X`参数，使用该备份所需的所有事务日志将会被自动包含在该备份中，并且恢复该备份也不需要特殊的动作。

如果在复制备份文件时需要更多灵活性，也可以使用一个较低层的处理来创建单机热备份。要为低层 单机热备份做准备，确保`wal_level`被设置为`replica`或更高，`archive_mode`设置为`on`，并且设置一个`archive_command`，该命令只当一个开关文件存在时执行归档。例如：

```
archive_command = 'test ! -f /var/lib/lightdb-x/backup_in_progress || (test ! -f /var/lib/lightdb-x/archive/%f && cp %p /var/lib/lightdb-x/archive/%f)'
```

该命令在`/var/lib/lightdb-x/backup_in_progress`存在时执行归档，否则会安静地返回0值退出状态（让LightDB能回收不需要的WAL文件）。

通过这样的准备，可以使用一个如下所示的脚本来建立备份：

```
touch /var/lib/lightdb-x/backup_in_progress
ltsql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/lightdb-x/backup.tar /var/lib/lightdb-x/data/
ltsql -c "select pg_stop_backup();"
rm /var/lib/lightdb-x/backup_in_progress
tar -rf /var/lib/lightdb-x/backup.tar /var/lib/lightdb-x/archive/
```

开关文件`/var/lib/lightdb-x/backup_in_progress`首先被创建，这使对于未完成WAL文件的归档操作发生。备份完成之后开关文件会被删除。归档的WAL文件则被加入到备份中，这样基础备份和所有需要的WAL文件都是同一个tar文件的组成部分。请记住在你的备份脚本中加入错误处理。

### 23.3.6.2. 压缩的归档日志

如果担心归档存储的尺寸，你可以使用`gzip`来压缩归档文件：

```
archive_command = 'gzip < %p > /mnt/server/archivedir/%f.gz'
```

那么在恢复时你将需要使用`gunzip`：

```
restore_command = 'gunzip < /mnt/server/archivedir/%f.gz > %p'
```

### 23.3.6.3. archive\_command脚本

很多人选择使用脚本来定义他们的archive\_command，这样他们的lightdb.conf项看起来非常简单：

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

任何时候如果你希望在归档处理中使用多个命令，明智的方法是使用一个独立的脚本文件。这可以使脚本更为复杂，它可以使用一种流行的脚本语言来编写，例如bash或perl。

需要在一个脚本内解决的需求例子包括：

- 将数据拷贝到安全的场外数据存储
- 批处理WAL文件，这样它们可以每三小时被传输一次，而不是一次一个
- 与其他备份和恢复软件交互
- 与监控软件交互以报告错误

#### Tip

在使用一个archive\_command脚本时，最好启用logging\_collector。任何从该脚本被写到stderr的消息将出现在数据库服务器日志中，这允许在复杂配置失败后能更容易被诊断。

### 23.3.7. 警告

在编写此文档时，连续归档技术存在一些限制。这可能会在未来的发布中被修复：

- 如果一个CREATE DATABASE命令在基础备份时被执行，然后在基础备份进行时CREATE DATABASE所复制的模板数据库被修改，恢复中可能会导致这些修改也被传播到已创建的数据库中。这当然是我们不希望的。为了避免这种风险，最好不要在创建基础备份时修改任何模板数据库。
- CREATE TABLESPACE命令会WAL以其字面绝对路径记录，并且因此将在重放时以相同的绝对路径来创建表空间。当日志在一台不同的机器上被重放时，这可能也不是我们希望的。即使日志在同一台机器上被重放也是危险的，就算是恢复到一个新的数据目录重放过程也会覆盖原来表空间的内容。为了避免这种潜在的陷阱，最佳做法是在创建或丢弃表空间后创建一个新的基础备份。

还需要注意的是，默认的WAL格式相当庞大，因为它包括了很多磁盘页快照。这些页快照被设计用于支持崩溃恢复，因为我们可能需要修复断裂的磁盘页。依靠你的系统硬件和软件，页断裂的风险可能会小到可以忽略，在此种情况下你可以通过使用full\_page\_writes参数关闭页快照来显著降低归档日志的总容量（在这样做之前阅读Chapter 2中的注解和警告）。关闭页快照并不会阻止使用日志进行PITR操作。一个未来的开发点是通过移除不需要的页拷贝来压缩归档的WAL数据，即使full\_page\_writes为on。同时，管理员可能希望通过尽可能增大检查点间隔参数来减少WAL中包含的页快照数量。

---

## Chapter 24. 高可用、负载均衡和复制

数据库服务器可以一起工作，这样如果主要的服务器失效则允许一个第二服务器快速接手它的任务（高可用性），或者可以允许多个计算机提供相同的数据（负载均衡）。理想情况下，数据库服务器能够无缝地一起工作。提供静态网页服务的网页服务器可以非常容易地通过把网页请求均衡到多个机器来组合。事实上，只读的数据库服务器也可以相对容易地组合起来。不幸的是，大部分数据库服务器收到的请求是读/写混合的，并且读/写服务器更难于组合。这是因为尽管只读数据只需要在每台服务器上放置一次，但对于任意服务器的一次写动作却必须被传播给所有的服务器，这样才能保证未来对于那些服务器的读请求能返回一致的结果。

这种同步问题是服务器一起工作的最根本的困难。因为没有单一解决方案能够消除该同步问题对所有用例的影响。有多种解决方案，每一种方案都以一种不同的方式提出了这个问题，并且对于一种特定的负载最小化了该问题所产生的影响。

某些方案通过只允许一台服务器修改数据来处理同步。能修改数据的服务器被称为读/写、主控或主要服务器。跟踪主控机中改变的服务器被称为后备或次级服务器。如果一台后备服务器只有被提升为一台主控服务器后才能被连接，它被称为一台温后备服务器，而一台总是能够接受连接并且提供只读查询的后备服务器被称为一台热后备服务器。

某些方案是同步的，即一个数据修改事务只有到所有服务器都提交了该事务之后才被认为是提交成功。这保证了一次故障转移不会丢失任何数据并且所有负载均衡的服务器将返回一致的结果（不管哪台服务器被查询）。相反，异步的方案允许在一次提交和它被传播到其他服务器之间有一些延迟，这产生了切换到一个备份服务器时丢失某些事务的可能性，并且负载均衡的服务器可能会返回略微陈旧的结果。当同步通信可能很慢时，可以使用异步通信。

方案也可以按照它们的粒度进行分类。某些方案只能处理一整个数据库服务器，而其他的允许在每个表或每个数据库的级别上进行控制。

在任何选择中，都必须考虑性能。通常在功能和性能之间都存在着权衡。例如，在一个低速网络上的一种完全同步的方案可能使性能减少超过一半，而一种异步的方案产生的性能影响可能是最小的。

本节的剩余部分勾勒了多种故障转移、复制和负载均衡方案。

### 24.1. 不同方案的比较

#### 共享磁盘故障转移

共享磁盘故障转移避免了只使用一份数据库拷贝带来的同步开销。它使用一个由多个服务器共享的单一磁盘阵列。如果主数据库服务器失效，后备服务器则可以挂载并启动数据库，就好像它从一次数据库崩溃中恢复过来了。这是一种快速的故障转移，并且不存在数据丢失。

共享硬件功能在网络存储设备中很常见。也可以使用一个网络文件系统，但是要注意的是该文件系统应具有完全的POSIX行为（见Section 16.2.2.1）。这种方法的一个重大限制是如果共享磁盘阵列失效或损坏，主要和后备服务器都会变得无法工作。另一个问题是在主要服务器运行时，后备服务器永远不能访问共享存储。

#### 文件系统（块设备）复制

共享硬件功能的一种修改版本是文件系统复制，在其中对一个文件系统的改变会被镜像到位于另一台计算机上的一个文件系统。唯一的限制是该镜像过程必须能保证后备服务器有一份该文件系统的一致的拷贝——特别是对后备服务器的写入必须按照主控机上相同的顺序进行。DRBD是用于Linux的一种流行的文件系统复制方案。

## 预写式日志传送

温备和热备服务器能够通过读取一个预写式日志（WAL）记录的流来保持为当前状态。如果主服务器失效，后备服务器拥有主服务器的几乎所有数据，并且能够快速地被变成新的主数据库服务器。这可以是同步的或异步的，并且只能用于整个数据库服务器。

可以使用基于文件的日志传送（Section 24.2）、流复制（见Section 24.2.5）或两者的组合来实现一个后备服务器。关于热备的信息可见Section 24.5

## 逻辑复制

逻辑复制允许数据库服务器发送数据更新流给另一台服务器。LightDB逻辑复制从WAL构建出逻辑数据更新流。逻辑复制允许您逐个表复制数据更改。此外，发布数据更新的服务器可以同时订阅其他服务器的更改，从而允许数据在多个方向流动。有关逻辑复制的更多信息，请参考 Chapter 28 通过逻辑解码接口（Chapter 45，第三方扩展也能提供类似的功能。

## 基于触发器的主-备复制

基于触发器的复制通常会将修改数据的查询发送到指定的主服务器。它在逐个表的基础上工作，主服务器（通常）将数据更改异步发送到备用服务器。主服务器运行时，备用服务器可以响应查询，并执行本地数据修改或写入操作。这种形式的复制通常用于减轻大数据分析型平台或者数据仓库查询负荷。

Slony-I是这种复制类型的一个例子。它使用表粒度，并且支持多个后备服务器。因为它会异步更新后备服务器（批量），在故障转移时可能会有数据丢失。

## 基于SQL的复制中间件

通过基于SQL的复制中间件，一个程序拦截每一个 SQL 查询并把它发送给一个或所有服务器。每一个服务器独立地操作。读写查询必须被发送给所有服务器，这样每一个服务器都能接收到任何修改。但只读查询可以被只发送给一个服务器，这样允许读负载在服务器之间分布。

如果查询未经修改发送，则函数的random()随机值和CURRENT\_TIMESTAMP函数的当前时间和序列值可能因不同服务器而异。因为每个服务器独立运行，并且它发送 SQL 查询而没有真正的更改数据。如果这是不可接受的，那么中间件或应用程序必须从单一服务器源确定此类值，并将结果用于写入查询。还必须注意确保所有服务器在提交或中止事务时都是相同的。这将涉及使用 两阶段提交（PREPARE TRANSACTION和COMMIT PREPARED）。Pgpool-II和Continuent Tungsten就是这种复制的例子。

## 异步多主控机复制

对于不会被定期连接或通讯链路较慢的服务器，如笔记本或远程服务器，保持服务器间的数据一致是一个挑战。通过使用异步的多主控机复制，每一个服务器独立工作并且定期与其他服务器通信来确定冲突的事务。这些冲突可以由用户或冲突解决规则来解决。Bucardo 是这种复制类型的一个例子。

## 同步多主控机复制

在同步多主控机复制中，每一个服务器能够接受写请求，并且在每一个事务提交之前，被修改的数据会被从原始服务器传送给每一个其他服务器。繁重的写活动可能导致过多的锁定和提交延迟，进而导致很差的性能。读请求可以被发送给任意服务器。某些实现使用共享磁盘来减少通信负荷。同步多主控机复制主要对于读负载最好，尽管它的大优点是任意服务器都能接受写请求 — 没有必要在主服务器和后备服务器之间划分负载，并且因为数据修改被从一个服务器发送到另一个服务器，不会有非确定函数（如random()）的问题。

LightDB不提供这种复制类型，尽管在应用代码或中间件中可以使用LightDB的两阶段提交（PREPARE TRANSACTION和COMMIT PREPARED）来实现这种复制。

Table 24. 总结了上述多种方案的能力。

Table 24.1. 高可用、负载均衡和复制特性矩阵

特性	共享磁盘	文件系统复制	预写式日志传送	逻辑复制	基于触发器的复制	SQL复制中间件	异步多主控机复制	同步多主控机复制
常用的示例	NAS	DRBD	内建流复制	内建逻辑复制, pglogical	diste, Slon, logical	pool-II	Bucardo	
通信方法	共享磁盘	磁盘块	WAL	逻辑解码	表行	SQL	表行	表行和行锁
不要求特殊硬件		•	•	•	•	•	•	•
允许多个主控机服务器				•		•	•	•
无主服务器负载	•		•	•		•		
不等待多个服务器	•		with sync off	with sync off	•		•	
主控机失效将永不丢失数据	•	•	with sync on	with sync on		•		•
复制体接受只读查询			with hot	•	•	•	•	•
每个表粒度				•	•		•	•
不需要冲突解决	•	•	•		•	•		•

有一些方案不适合上述的类别:

#### 数据分区

数据分区将表分开成数据集。每个集合只能被一个服务器修改。例如，数据可以根据办公室划分，如伦敦和巴黎，每一个办公室有一个服务器。如果查询有必要组合伦敦和巴黎的数据，一个应用可以查询两个服务器，或者可以使用主/备复制来在每一台服务器上保持其他办公室数据的一个只读拷贝。

#### 多服务器并行查询执行

上述的很多方案允许多个服务器来处理多个查询，但是没有一个允许一个单一查询使用多个服务器来更快完成。这种方案允许多个服务器在一个单一查询上并发工作。这通常通过把数据在服务器之间划分并且让每一个服务器执行该查询中属于它的部分，然后将结果返回给一个中心服务器，由它整合结果并发回给用户。这也可以使用PL/Proxy工具集来实现这种方案。

还需要注意的是由于LightDB是开源的并且易于扩展，许多公司已经采用LightDB并创建了具有独特的故障转移、复制和负载均衡功能的商业闭源解决方案。这里不讨论这些。

## 24.2. 日志传送后备服务器

连续归档可以被用来创建一个高可用性（HA）集群配置，其中有一个或多个后备服务器随时准备在主服务器失效时接管操作。这种能力被广泛地称为温备或日志传送。

主服务器和后备服务器一起工作来提供这种能力，但这些服务器只是松散地组织在一起。主服务器在连续归档模式下操作，而每一个后备服务器在连续恢复模式下操作并且持续从主服务器读取 WAL 文件。要启用这种能力不需要对数据库表做任何改动，因此它相对于其他复制方案降低了管理开销。这种配置对主服务器的性能影响也相对较低。

直接从一个数据库服务器移动 WAL 记录到另一台服务器通常被描述为日志传送。LightDB 通过一次一文件（WAL 段）的 WAL 记录传输实现了基于文件的日志传送。不管 WAL 文件（16 MB）要被送到一个临近的系统、同一站点的另一个系统或是在地球遥远的另一端的一个系统上，它都可以在任何距离上被简单和便宜地传送。这种技术所需的带宽取根据主服务器的事务率而变化。基于记录的日志传送具有更细的粒度并且能够在网络连接上以流的方式增量传递 WAL 的改变（见Section 24.2.5）。

需要注意的是日志传送是异步的，即 WAL 记录是在事务提交后才被传送。正因为如此，在一个窗口期内如果主服务器发生灾难性的失效则会导致数据丢失，还没有被传送的事务将会被丢失。基于文件的日志传送中这个数据丢失窗口的尺寸可以通过使用参数 `archive_timeout` 进行限制，它可以被设置成低至数秒。但是这样低的设置大体上会增加文件传送所需的带宽。流复制（见Section 24.2.5 允许更小的数据丢失窗口）。

这种配置的恢复性能是足够好的，后备服务器在被激活后通常只有片刻就可以到达完全可用。因此，这被称为一种提供高可用性的温备配置。从一个已归档的基础备份恢复一个服务器并且前滚将需要较长时间，因此该技术只提供了灾难恢复的一种方案，而不适合于高可用性。一台后备服务器也可以被用于只读查询，在这种情况下它被称为一台热备服务器。更多信息请见Section 24.5

## 24.2.1. 规划

创建主服务器和后备服务器通常是明智的，因此它们可以尽可能相似，至少从数据库服务器的角度来看是这样。特别地，与表空间相关的路径名将被未经修改地传递，因此如果该特性被使用，主、备服务器必须对表空间具有完全相同的挂载路径。记住如果 `CREATE TABLESPACE` 在主服务器上被执行，在命令被执行前，它所需要的任何新挂载点必须在主服务器和所有后备服务器上先创建好。硬件不需要完全相同，但是经验显示，在应用和系统的生命期内维护两个相同的系统比维护两个不相似的系统更容易。在任何情况下硬件架构必须相同——从一个 32 位系统传送到一个 64 位系统将不会工作。

通常，不能在两个运行着不同主版本 LightDB 的服务器之间传送日志。LightDB 全球开发组的策略是不在次版本升级中改变磁盘格式，因此在主服务器和后备服务器上运行不同次版本将会成功地工作。不过，在这方面并没有提供正式的支持，因此我们建议让主备服务器上运行的版本尽可能相同。当升级到一个新的次版本时，最安全的策略是先升级后备服务器——一个新的次版本发行更可能兼容从前一个次版本读取 WAL 文件。

## 24.2.2. 后备服务器操作

服务器启动时，数据目录中存在 `standby.signal` 文件，服务器进入 standby 模式。

在后备模式中，服务器持续地应用从主控服务器接收到的 WAL。后备服务器可以从一个 WAL 归档 (`restore_command`) 或者通过一个 TCP 连接直接从主控机（流复制）读取 WAL。后备服务器将也尝试恢复在后备集簇的 `lt_wal` 目录中找到的 WAL。那通常在一次数据库重启后发生，那时后备机将在重启之前重播从主控机流过来的 WAL，但是你也可以在任何时候手动拷贝文件到 `lt_wal` 让它们被重播。

在启动时，后备机通过恢复归档位置所有可用的 WAL 来开始，这称为 `restore_command`。一旦它到达那里可用的 WAL 的末尾并且 `restore_command` 失败，它会尝试恢复 `lt_wal` 目录中可用的任何 WAL。如果那也失败并且流复制已被配置，后备机会尝试连接到主服务器并且从在归档或 `lt_wal` 中找到的最后一个可用记录开始流式传送 WAL。如果那失败并且没有配置流复制，或者该连接后来断开，后备机会返回到步骤 1 并且尝试再次从归档里的文件恢复。这种尝试归档、`lt_wal` 和流复制的循环会一直重复知道服务器停止或者一个触发器文件触发了故障转移。



当`lt_ctl promote`被运行，`pg_promote()`被调用，或一个触发器文件被找到（`promote_trigger_file`），后备模式会退出并且服务器会切换到普通操作。在故障转移之前，在归档或`lt_wal`中立即可用的任何 WAL 将被恢复，但不会尝试连接到主控机。

### 24.2.3. 为后备服务器准备主控机

如Section 23.8所述，在主服务器上设置连续归档到一个后备服务器可访问的归档目录。即使主服务器垮掉该归档位置也应当是后备服务器可访问的，即它应当位于后备服务器本身或者另一个可信赖的服务器，而不是位于主控服务器上。

如果你想要使用流复制，在主服务器上设置认证来允许来自后备服务器的复制连接。即创建一个角色并且在`lt_hba.conf`中提供一个或多个数据库域被设置为`replication`的项。还要保证在主服务器的配置文件中`max_wal_senders`被设置为足够大的值。如果要使用复制槽，请确保`max_replication_slots`也被设置得足够高。

按Section 23.3所述取得一个基础备份来引导后备服务器。

### 24.2.4. 建立一个后备服务器

要建立后备服务器，恢复从主服务器取得的基础备份（Section 23.3.4）。在后备服务器的集簇数据目录中创建一个文件`standby.signal`。将`restore_command`设置为一个从 WAL 归档中复制文件的简单命令。如果你计划为了高可用性目的建立多个后备服务器，确认`recovery_target_timeline`被设置为`latest`（默认）来使得该后备服务器遵循发生在故障转移到另一个后备服务器之后发生的时间线改变。

#### Note

请勿使用`lt_standby`或类似工具与此处所描述的内置备用模式一起使用。如果文件不存在，`restore_command`应该立即返回；如果需要，服务器将重新尝试该命令。有关使用`lt_standby`等工具的方法，请参见Section 24.4

如果你想要使用流复制，在`primary_conninfo`中填入一个 `libpq` 连接字符串，其中包括主机名（或 IP 地址）和连接到主服务器所需的任何附加细节。如果主服务器需要一个口令用于认证，口令也应该被指定在`primary_conninfo`中。

如果你正在为高性能目的建立后备服务器，像主服务器一样建立 WAL 归档、连接和认证，因为在故障转移后该后备服务器将作为一个主服务器工作。

如果你正在使用一个 WAL 归档，可以使用`archive_cleanup_command`参数来移除后备服务器不再需要的文件，这样可以最小化 WAL 归档的尺寸。`lt_archivecleanup`工具被特别设计为在典型单一后备配置下与`archive_cleanup_command`共同使用，见`lt_archivecleanup`。不过要注意，如果你正在为备份目的使用归档，有一些文件即使后备服务器不再需要你也需要保留它们，它们被用来从至少最后一个基础备份恢复。

配置的一个简单例子是：

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass
options='' -c wal_sender_timeout=5000''
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'lt_archivecleanup /path/to/archive %r'
```

你可以有任意数量的后备服务器，但是如果你使用流复制，确保你在主服务器上 将`max_wal_senders`设置得足够高，这样可以允许它们能同时连接。

### 24.2.5. 流复制

流复制允许一台后备服务器比使用基于文件的日志传送更能保持为最新的状态。后备服务器连接到主服务器，主服务器则在 WAL 记录产生时即将它们以流式传送给后备服务器而不必等到 WAL 文件被填充。

默认情况下流复制是异步的（见Section 24.2.8，在这种情况下主服务器上提交一个事务与该变化在后备服务器上变得可见之间存在短暂的延迟。不过这种延迟比基于文件的日志传送方式中要小得多，在后备服务器的能力足以跟得上负载的前提下延迟通常低于一秒。在流复制中，不需要archive\_timeout来缩减数据丢失窗口。

如果你使用的流复制没有基于文件的连续归档，该服务器可能在后备机收到 WAL 段之前回收这些旧的 WAL 段。如果发生这种情况，后备机将需要重新从一个新的基础备份初始化。通过设置wal\_keep\_size为一个足够高的值来确保旧的 WAL 段不会被太早重用或者为后备机配置一个复制槽，可以避免发生这种情况。如果设置了一个后备机可以访问的 WAL 归档，就不需要这些解决方案，因为该归档可以为后备机保留足够的段，后备机总是可以使用该归档来追赶主控机。

要使用流复制，按Section 24.2所述建立一个基于文件的日志传送后备服务器。将一个基于文件日志传送后备服务器转变成流复制后备服务器的步骤是把recovery.conf文件中的设置以指向主服务器。设置主服务器上的listen\_addresses和认证选项（见lt\_hba.conf），这样后备服务器可以连接到主服务器上的伪数据库replication（见Section 24.2.5.1）。

在支持 keepalive 套接字选项的系统上，设置tcp\_keepalives\_idle、tcp\_keepalives\_interval和 tcp\_keepalives\_count有助于主服务器迅速地注意到一个断开的连接。

设置来自后备服务器的并发连接的最大数目（详见max\_wal\_senders）。

当后备服务器被启动并且primary\_conninfo被正确设置，后备服务器将在重放完归档中所有可用的 WAL 文件之后连接到主服务器。如果连接被成功建立，你将在后备服务器中看到一个 walreceiver，并且在主服务器中有一个相应的 walsender 进程。

### 24.2.5.1. 认证

设置好用于复制的访问权限非常重要，这样只有受信的用户可以读取 WAL 流，因为很容易从 WAL 流中抽出需要特权才能访问的信息。后备服务器必须作为一个具有REPLICATION特权的账户或一个超级用户来向主服务器认证。推荐为复制创建一个专用的具有REPLICATION和LOGIN特权的用户账户。虽然REPLICATION特权给出了非常高的权限，但它不允许用户修改主系统上的任何数据，而SUPERUSER特权则可以。

复制的客户端认证由一个在database域中指定replication的lt\_hba.conf记录控制。例如，如果后备服务器运行在主机 IP 192.168.1.100并且用于复制的账户名为foo，管理员可以在主服务器上向lt\_hba.conf文件增加下列行：

```
# 允许来自 192.168.1.100 的用户 "foo" 在提供了正确的口令时作为一个
# 复制后备机连接到主控机。
#
# TYPE DATABASE USER ADDRESS METHOD
host replication foo 192.168.1.100/32 md5
```

主服务器的主机名和端口号、连接用户名和口令在primary\_conninfo中指定。在后备服务器上还可以在~/.pgpass文件中设置口令（在database域中指定replication）。例如，如果主服务器运行在主机 IP 192.168.1.50、端口5432上，并且口令为foopass，管理员可以在后备服务器的lightdb.conf文件中增加下列行：

```
# 后备机要连接到的主控机运行在主机 192.168.1.50 上，
# 端口号是 5432，连接所用的用户名是 "foo"，口令是 "foopass"。
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

## 24.2.5.2. 监控

流复制的一个重要健康指标是在主服务器上产生但还没有在后备服务器上应用的 WAL 记录数。你可以通过比较主服务器上的当前 WAL 写位置和后备服务器接收到的最后一个 WAL 位置来计算这个滞后量。这些位置分别可以用主服务器上的 `pg_current_wal_lsn` 和后备服务器上的 `pg_last_wal_receive_lsn` 来检索（详见 Table 9.86 和 Table 9.87）。后备服务器的最后 WAL 接收位置也被显示在 WAL 接收者进程的进程状态中，即使用 `ps` 命令显示的状态（详见 Section 25.1）。

你可以通过 `pg_stat_replication` 视图检索 WAL 发送者进程的列表。`pg_current_wal_lsn` 与 `sent_lsn` 域之间的巨大差异表示主服务器承受着巨大的负载，而 `sent_lsn` 和后备服务器上 `pg_last_wal_receive_lsn` 之间的差异可能表示网络延迟或者后备服务器正承受着巨大的负载。

在一台热后备上，WAL 接收者进程的状态可以通过 `pg_stat_wal_receiver` 视图检索到。`pg_last_wal_replay_lsn` 和该视图的 `flushed_lsn` 的差别表示 WAL 的接收速度大于它被重放的速度。

## 24.2.6. 复制槽

复制槽提供了一种自动化的方法来确保主控机在所有的后备机收到 WAL 段之前不会移除它们，并且主控机也不会移除可能导致恢复冲突的行，即使后备机断开也是如此。

作为复制槽的替代，也可以使用 `wal_keep_size` 阻止移除旧的 WAL 段，或者使用 `archive_command` 把段保存在一个归档中。不过，这些方法常常会导致保留的 WAL 段比需要的更多，而复制槽只保留已知所需要的段。另一方面，复制槽可以保留很多的 WAL 段以至于它们填满了分配给 `lt_wal` 的空间；`max_slot_wal_keep_size` 限制复制槽所保留的 WAL 文件的大小。

类似地，`hot_standby_feedback` 和 `vacuum_defer_cleanup_age` 保护了相关行不被 `vacuum` 移除，但是前者在后备机断开期间无法提供保护，而后者则需要被设置为一个很高的值已提供足够的保护。复制槽克服了这些缺点。

### 24.2.6.1. 查询和操纵复制槽

每个复制槽都有一个名字，名字可以包含小写字母、数字和下划线字符。

已有的复制槽和它们的状态可以在 `pg_replication_slots` 视图中看到。

槽可以通过流复制协议（见 Section 49.4）或者 SQL 函数（见 Section 9.27.6）创建并且移除。

### 24.2.6.2. 配置实例

你可以这样创建一个复制槽：

```
lightdb@postgres=# SELECT * FROM
pg_create_physical_replication_slot('node_a_slot');
 slot_name | lsn
-----+-----
node_a_slot |

lightdb@postgres=# SELECT slot_name, slot_type, active FROM
pg_replication_slots;
 slot_name | slot_type | active
-----+-----+-----
node_a_slot | physical | f
(1 row)
```

要配置后备机使用这个槽，在后备机中应该配置`primary_slot_name`。这里是一个简单的例子：

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'  
primary_slot_name = 'node_a_slot'
```

## 24.2.7. 级联复制

级联复制特性允许一台后备服务器接收复制连接并且把 WAL 记录流式传送给其他后备服务器，就像一个转发器一样。这可以被用来减小对于主控机的直接连接数并且使得站点间的带宽开销最小化。

一台同时扮演着接收者和发送者角色的后备服务器被称为一台级联后备服务器。“更直接”（通过更少的级联后备服务器）连接到主控机的后备服务器被称为上游服务器，而那些离得更远的后备服务器被称为下游服务器。级联复制并没有对下游服务器的数量或布置设定限制。

一台级联后备服务器不仅仅发送从主控机接收到的 WAL 记录，还要发送那些从归档中恢复的记录。因此即使某些上游连接中的复制连接被中断，只要还有新的 WAL 记录可用，下游的流复制都会继续下去。

级联复制目前是异步的。同步复制（见Section 24.2.8 设置当前对级联复制无影响。

不管在什么样的级联布置中，热备反馈都会向上游传播。

如果一台上游后备服务器被提升为新的主控机，且下游服务器的`recovery_target_timeline`被设置成`'latest'`（默认），下游服务器将继续从新的主控机得到流。

要使用级联复制，要建立级联后备服务器让它能够接受复制连接（即设置`max_wal_senders`和`hot_standby`，并且配置基于主机的认证）。你还将需要设置下游后备服务器中的`primary_conninfo`指向级联后备服务器。

## 24.2.8. 同步复制

LightDB流复制默认是异步的。如果主服务器崩溃，则某些已被提交的事务可能还没有被复制到后备服务器，这会导致数据丢失。数据的丢失量与故障转移时的复制延迟成比例。

同步复制能够保证一个事务的所有修改都能被传送到一台或者多台同步后备服务器。这扩大了由一次事务提交所提供的标准持久化级别。在计算机科学理论中这种保护级别被称为 `2-safe` 复制。而当`synchronous_commit`被设置为`remote_write`时，则是 `group-1-safe`（`group-safe` 和 `1-safe`）。

在请求同步复制时，一个写事务的每次提交将一直等待，直到收到一个确认表明该提交在主服务器和后备服务器上都被写入到磁盘上的预写式日志中。数据会被丢失的唯一可能性是主服务器和后备服务器在同一时间都崩溃。这可以提供更高级别的持久性，尽管只有系统管理员要关系两台服务器的放置和管理。等待确认提高了用户对于修改不会丢失的信心，但是同时也不必要地增加了对请求事务的响应时间。最小等待时间是在主服务器和后备服务器之间的来回时间。

只读事务和事务回滚不需要等待后备服务器的回复。子事务提交也不需要等待后备服务器的响应，只有顶层提交才需要等待。长时间运行的动作（如数据载入或索引构建）不会等待最后的提交消息。所有两阶段提交动作要求提交等待，包括预备和提交。

同步后备可以是物理复制后备或者是逻辑复制订阅者。它还可以是任何其他物理或者逻辑 WAL 复制流的消费者，它懂得如何发送恰当的反馈消息。除内建的物理和逻辑复制系统之外，还包括`pg_receivewal`和`pg_recvlogical`之类的特殊程序，以及一些第三方复制系统和定制程序。同步复制支持的细节请查看相应的文档。

### 24.2.8.1. 基本配置

一旦流复制已经被配置，配置同步复制就只需要一个额外的配置步骤：

`synchronous_standby_names`必须被设置为一个非空值。`synchronous_commit`也必须被设置为on，但由于这是默认值，通常不需要改变（见Section 17.5.和Section 17.6.2）。这样的配置将导致每一次提交都等待确认消息，以保证后备服务器已经将提交记录写入到持久化存储中。`synchronous_commit`可以由个体用户设置，因此它可以在配置文件中配置、可以为特定用户或数据库配置或者由应用动态配置，这样可以在一种每事务基础上控制持久性保证。

在一个提交记录已经在主服务器上被写入到磁盘后，WAL 记录接着被发送到后备服务器。每次一批新的 WAL 数据被写入到磁盘后，后备服务器会发送回复消息，除非在后备服务器上`wal_receiver_status_interval`被设置为零。如果`synchronous_commit`被设置为`remote_apply`，当提交记录被重放时后备服务器会发送回应消息，这会让该事务变得可见。如果根据主服务器的`synchronous_standby_names`设置选中该后备服务器作为一个同步后备，将会根据来自该后备服务器和其他同步后备的回应消息来决定何时释放正在等待确认提交记录被收到的事务。这些参数允许管理员指定哪些后备服务器应该是同步后备。注意同步复制的配置主要在主控机上。命名的后备服务器必须直接连接到主控机，主控机对使用级联复制的下游后备服务器一无所知。

将`synchronous_commit`设置为`remote_write`将导致每次提交都等待后备服务器已经接收提交记录并将它写出到其自身所在的操作系统的确认，但并非等待数据都被刷出到后备服务器上的磁盘。这种设置提供了比on要弱一点的持久性保障：在一次操作系统崩溃事件中后备服务器可能丢失数据，尽管它不是一次LightDB崩溃。不过，在实际中它是一种有用的设置，因为它可以减少事务的响应时间。只有当主服务器和后备服务器都崩溃并且主服务器的数据库同时被损坏的情况下，数据丢失才会发生。

把`synchronous_commit`设置为`remote_apply`将导致每一次提交都会等待，直到当前的同步后备服务器报告说它们已经重放了该事务，这样就会使该事务对用户查询可见。在简单的情况下，这为带有因果一致性的负载均衡留出了余地。

如果请求一次快速关闭，用户将停止等待。不过，在使用异步复制时，在所有未解决的 WAL 记录被传输到当前连接的后备服务器之前，服务器将不会完全关闭。

### 24.2.8.2. 多个同步后备

同步复制支持一个或者更多个同步后备服务器，事务将会等待，直到所有同步后备服务器都确认收到了它们的数据为止。事务必须等待其回复的同步后备的数量

由`synchronous_standby_names`指定。这个参数还指定一个后备服务器名称及方法（FIRST和ANY）的列表来从列出的后备中选取同步后备。

方法FIRST指定一种基于优先的同步复制并且让事务提交等待，直到它们的WAL记录被复制到基于优先级选中的所要求数量的同步后备上为止。在列表中出现较早的后备被给予较高的优先级，并且将被考虑为同步后备。其他在这个列表中位置靠后的后备服务器表示可能的同步后备。如果任何当前的同步后备由于任何原因断开连接，它将立刻被下一个最高优先级的后备所替代。

基于优先的多同步后备的`synchronous_standby_names`示例为：

```
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

在这个例子中，如果有四个后备服务器s1、s2、s3和s4在运行，两个后备服务器s1和s2将被选中为同步后备，因为它们出现在后备服务器名称列表的前部。s3是一个潜在的同步后备，当s1或s2中的任何一个失效，它就会取而代之。s4则是一个异步后备因为它的名字不在列表中。

方法ANY指定一种基于规定数量的同步复制并且让事务提交等待，直到它们的WAL记录至少被复制到列表中所要求数量的同步后备上为止。

`synchronous_standby_names`的基于规定数量的多同步后备的例子：

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

在这个例子中，如果有四台后备服务器s1、s2、s3以及s4正在运行，事务提交将会等待来自至少其中任意两台后备服务器的回复。s4是一台异步后备，因为它的名字不在该列表中。

后备服务器的同步状态可以使用`pg_stat_replication`视图查看。

### 24.2.8.3. 性能规划

同步复制通常要求仔细地规划和放置后备服务器来保证应用能令人满意地工作。等待并不利用系统资源，但是事务锁会持续保持直到传输被确认。其结果是，不小心使用同步复制将由于响应时间增加以及较高的争用率而降低数据库应用的性能。

LightDB允许应用开发者通过复制来指定所要求的持久性级别。这可以为整个系统指定，不过它也能够为特定的用户或连接指定，甚至还可以为单个事务指定。

例如，一个应用的载荷的组成可能是这样：10%的改变是重要的客户详情，而90%的改变是不太重要的数据，即使它们丢失业务也比较容易容忍（例如用户间的聊天消息）。

通过在应用级别（在主服务器上）指定的同步复制选项，我们可以为大部分重要的改变提供同步复制，并且不会拖慢整体的载荷。应用级别选项是使高性能应用享受同步复制的一种重要和实用的工具。

你应该认为网络带宽必须比WAL数据的产生率高。

### 24.2.8.4. 高可用性规划

当`synchronous_commit`被设置为`on`、`remote_apply`或者`remote_write`时，`synchronous_standby_names`指定产生的事务提交要等待其回应的同步后备的数量和名称。如果任一同步后备崩溃，这类事务提交可能无法完成。

高可用的最佳方案是确保有所要求数量的同步后备。这可以通过使用`synchronous_standby_names`指定多个潜在后备服务器来实现。

在基于优先的同步复制中，出现在该列表前部的后备服务器将被用作同步后备。后面的后备服务器将在当前同步后备服务器失效时取而代之。

在基于规定数量的同步复制中，所有出现在该列表中的后备服务器都将被用作同步后备的候选。即使其中的一个失效，其他后备仍将继续担任候选同步后备的角色。

当一台后备服务器第一次附加到主服务器时，它将处于一种还没有正确同步的状态。这被描述为追赶模式。一旦后备服务器和主服务器之间的迟滞第一次变成零，我们就来到了实时的流式状态。在后备服务器被创建之后的很长时间内可能都是追赶模式。如果后备服务器被关闭，则追赶周期将被增加，增加量由后备服务器被关闭的时间长度决定。只有当后备服务器到达流式状态后，它才能成为一台同步后备。这种状态可以使用`pg_stat_replication`视图查看。

如果在提交正在等待确认时主服务器重启，那些正在等待的事务将在主数据库恢复时被标记为完全提交。没有办法确认所有后备服务器已经收到了在主服务器崩溃时所有还未处理的WAL数据。某些事务可能不会在后备服务器上显示为已提交，即使它们在主服务器上显示为已提交。我们提供的保证是：在WAL数据已经被所有后备服务器安全地收到之前，应用将不会收到一个事务成功提交的显式确认。

如果实在无法保持所要求数量的同步后备，那么应该减少`synchronous_standby_names`中指定的事务提交应该等待其回应的同步后备的数量（或者禁用），并且在主服务器上重载配置文件。

如果主服务器与剩下的后备服务器是隔离的，你应当故障转移到那些其他剩余后备服务器中的最佳候选者上。

如果在事务等待时你需要重建一台后备服务器，确保命令 `pg_start_backup()` 和 `pg_stop_backup()` 被运行在一个 `synchronous_commit = off` 的会话中，否则那些请求将永远等待后备服务器出现。

## 24.2.9. 在后备机上连续归档

当在一个后备机上使用连续归档时，有两种不同的情景：WAL 归档在主服务器和后备机之间共享，或者后备机有自己的 WAL 归档。当后备机拥有其自身的 WAL 归档时，将 `archive_mode` 设置为 `always`，后备机将在收到每个 WAL 段时调用归档命令，不管它是从归档恢复还是使用流复制恢复。共享归档可以类似地处理，但是 `archive_command` 必须测试要被归档的文件是否已经存在，以及现有的文件是否有相同的内容。这要求 `archive_command` 中有更多处理，因为它必须当心不要覆盖具有不同内容的已有文件，但是如果完全相同的文件被归档两次时应返回成功。并且如果两个服务器尝试同时归档同一个文件，所有这些都必须在没有竞争情况的前提下完成。

如果 `archive_mode` 被设置为 `on`，归档器在恢复或者后备模式中无法启用。如果后备服务器被提升，它将在被提升后开始归档，但是它将不会归档任何不是它自身产生的 WAL 或时间线历史文件。要在归档中得到完整的一系列 WAL 文件，你必须确保所有 WAL 在到达后备机之前都被归档。对于基于文件的日志传输来说天然就是这样，因为后备机只能恢复在归档中找到的文件，而启用了流复制时则不是这样。当一台服务器不在恢复模式中时，在 `on` 和 `always` 模式之间没有差别。

## 24.3. 故障转移

如果主服务器失效，则后备服务器应该开始故障转移过程。

如果后备服务器失效，则不会有故障转移发生。如果后备服务器可以被重启（即使晚一点），由于可重启恢复的优势，那么恢复处理也能被立即重启。如果后备服务器不能被重启，则一个全新的后备服务器实例应该被创建。

如果主服务器失效并且后备服务器成为了新的主服务器，那么接下来旧的主服务器重启后，你必须有一种机制来通知旧的主服务器不再成为主服务器。有些时候这被称为 STONITH (Shoot The Other Node In The Head, 关闭其他节点)，这对于避免出现两个系统都认为它们是主服务器的情况非常必要，那种情况将导致混乱并且最终导致数据丢失。

很多故障转移系统仅使用两个系统，主系统和后备系统，它们由某种心跳机制连接来持续验证两者之间的连接性和主系统的可用性。也可能会使用第三个系统（称为目击者服务器）来防止某些不当故障转移的情况，但是除非非常小心地建立它并且经过了严格地测试，额外的复杂度可能会使该工作得不偿失。

LightDB 并不提供在主服务器上标识失败并且通知后备数据库服务器所需的系统软件。现在已有很多这样的工具并且很好地与成功的故障转移所需的操作系统功能整合在一起，例如 IP 地址迁移。

一旦发生到后备服务器的故障转移，就只有单一的一台服务器在操作。这被称为一种退化状态。之前的后备服务器现在是主服务器，但之前的主服务器处于关闭并且可能一直保持关闭。要回到正常的操作，一个后备服务器必须被重建，要么在之前的主系统起来时使用它重建，要么使用第三台（可能是全新的）服务器来重建。在大型集簇上，`lt_rewind` 功能可以被用来加速这个过程。一旦完成，主服务器和后备服务器可以被认为是互换了角色。某些人选择使用第三台服务器来为新的主服务器提供备份，直到新的后备服务器被重建，不过显然这会使得系统配置和操作处理更复杂。

因此，从主服务器切换到后备服务器可以很快，但是要求一些时间来重新准备故障转移集群。从主服务器到后备服务器的常规切换是有用的，因为它允许每个系统有常规的关闭时间

来进行维护。这也可以作为一种对故障转移机制的测试，以保证在你需要它时它真地能够工作。我们推荐写一些管理过程来做这些事情。

要触发一台日志传送后备服务器的故障转移，运行`lt_ctl promote`，调用`pg_promote()`，或者创建一个触发器文件，其文件名和路径由`promote_trigger_file`设置指定。如果你正在规划使用`lt_ctl promote`或调用`pg_promote()`以进行故障转移，`promote_trigger_file`就不是必要的。如果你正在建立只用于从主服务器分流只读查询而不是高可用性目的的报告服务器，你不需要提升它。

LightDB的高可用性默认使用`keepalived`提供虚拟IP，客户端连接虚拟IP而不是物理IP。当发生故障转移时，虚拟IP应该跟随正确的主服务器。然而，在特殊情况下，例如维护期间，您需要禁用自动故障转移，不希望虚拟IP移动。在这种情况下，您可以运行`ltcluster service pause`（详见`ltcluster`<sup>1</sup>）以禁用自动故障转移。然后在主服务器上创建一个`$LTDATA/primary.maintain`文件，以防止虚拟IP自动移动。维护完成后，调用`ltcluster service unpause`以启用自动故障转移并删除`$LTDATA/primary.maintain`文件。

在`ltcluster`自动将备用服务器提升为主服务器后，将创建文件`$LTDATA/standby.promotion`以指示备用服务器已被提升。如果出现多个主服务器（例如，在备用服务器被提升后，旧的主服务器恢复正常），则`ltclusterd`守护程序将保留具有`$LTDATA/standby.promotion`文件的数据库实例，并停止没有此文件的主实例，以避免由多个主服务器引起的数据不一致风险。

## 24.4. 日志传送的替代方法

在前面的章节中所描述的内置备用模式的替代方法是使用一个轮询归档位置的`restore_command`。在8.4及以下版本中，这是唯一可用的选项。有关此的参考实现，请参见`lt_standby`模块。

请注意，在此模式下，服务器将一次应用一个WAL文件，因此如果您将备用服务器用于查询（请参见热备模式），主服务器中的操作与在备用服务器中可见之间存在一定的延迟，该延迟对应于填充WAL文件所需的时间。`archive_timeout`可用于缩短该延迟。还请注意，您不能将流复制与此方法结合使用。

在主服务器和备用服务器上发生的操作是正常的连续归档和恢复任务。两个数据库服务器之间唯一的联系点是它们共享的WAL文件归档：主服务器将WAL写入归档，备用服务器从归档中读取。必须注意确保来自不同主服务器的WAL归档不会混合在一起或混淆。如果归档仅用于备用操作，则归档无需很大。

使这两个松散耦合的服务器协同工作的魔法只是在备用服务器上使用的`restore_command`，当请求下一个WAL文件时，它会等待从主服务器可用。正常的恢复处理将从WAL归档中请求文件，如果文件不可用，则报告失败。对于备用处理，下一个WAL文件通常不可用，因此备用必须等待其出现。对于以`.history`结尾的文件，无需等待，必须返回非零返回代码。等待的`restore_command`可以编写为一个自定义脚本，该脚本在轮询下一个WAL文件的存在后循环。还必须有某种触发故障转移的方式，这应该中断`restore_command`，中止循环并向备用服务器返回文件未找到错误。这将结束恢复，然后备用服务器将作为普通服务器启动。

适当的`restore_command`的伪代码如下：

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);          /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
```

<sup>1</sup> <http://www.light-pg.com/docs/ltcluster/current/index.html>



```
CopyWALFileForRecovery();
```

在`lt_standby`模块中提供了一个可用的等待`restore_command`示例。它应该被用作参考，以了解如何正确实现上述逻辑。它也可以根据需要进行扩展，以支持特定的配置和环境。

触发故障转移的方法是规划和设计的重要部分。一个潜在的选项是`restore_command`命令。它对于每个WAL文件执行一次，但运行`restore_command`的进程会为每个文件创建和终止，因此没有守护进程或服务进程，无法使用信号或信号处理程序。因此，`restore_command`不适合触发故障转移。可以使用简单的超时机制，特别是与主服务器上已知的`archive_timeout`设置一起使用时。但是，这种方法有点容易出错，因为网络问题或繁忙的主服务器可能足以启动故障转移。如果可以安排，通知机制例如显式创建触发器文件是理想的。

## 24.4.1. 实现

使用这种替代方法配置备用服务器的简要步骤如下。有关每个步骤的详细信息，请参见前面的各节。

1. 尽可能相同地设置主系统和备用系统，包括两个相同版本的LightDB副本。
2. 从主服务器设置连续归档到备用服务器上的一个WAL归档目录。确保在主服务器上适当设置`archive_mode`、`archive_command`和`archive_timeout`（请参见Section 23.3.1）。
3. 对主服务器进行基本备份（请参见Section 23.3.2），并将此数据加载到备用服务器上。
4. 使用先前描述的等待`restore_command`（请参见Section 23.3.4）从本地WAL归档开始在备用服务器上恢复。

恢复将WAL归档视为只读，因此一旦将WAL文件复制到备用系统中，它就可以同时被复制到磁带中，就像备用数据库服务器正在读取它一样。因此，在进行高可用性备用服务器运行的同时，可以将文件存储为长期灾难恢复目的。

为了测试目的，可以在同一系统上运行主服务器和备用服务器。这不会提供任何有价值的服务器鲁棒性改进，也不能被描述为高可用性。

## 24.4.2. 基于记录的日志传送

使用这种替代方法也可以实现基于记录的日志传送，但这需要定制开发，并且更改仍将只在完整的WAL文件已被传送后对热备查询可见。

外部程序可以调用`pg_walfile_name_offset()`函数（请参见Section 9.27）来查找当前WAL的结束位置所在的文件名和确切的字节偏移量。然后它可以直接访问WAL文件，并将上次已知的WAL结束位置到当前结束位置之间的数据复制到备用服务器。使用这种方法，数据丢失的窗口是复制程序的轮询周期时间，这可以非常小，从而避免了强制归档部分使用的段文件所浪费的带宽。请注意，备用服务器的`restore_command`脚本只能处理整个WAL文件，因此增量复制的数据通常不会提供给备用服务器使用。它只有在主服务器死机时才有用—然后在允许它启动之前，将上次的一部分WAL文件提供给备用服务器。正确实现此过程需要`restore_command`脚本与数据复制程序的协作。

从LightDB 9.0版本开始，您可以使用流复制（请参见Section 24.2.5）以更少的工作实现相同的效益。

## 24.5. 热备

术语热备用来描述处于归档恢复或后备模式中的服务器连接到服务器并运行只读查询的能力。这有助于复制目的以及以高精度恢复一个备份到一个期望的状态。术语热备也指服务器从恢复转移到正常操作而用户能继续运行查询并且保持其连接打开的能力。

在热备模式中运行查询与正常查询操作相似，尽管如下所述存在一些用法和管理上的区别。

## 24.5.1. 用户概览

当`hot_standby`参数在一台后备服务器上被设置为真时，一旦恢复将系统带到一个一致的状态它将开始接受连接。所有这些连接都被限制为只读，甚至临时表都不能被写入。

后备服务器上的数据需要一些时间从主服务器到达后备服务器，因此在主服务器和后备服务器之间将有一段可以度量的延迟。近乎同时在主服务器和后备服务器上运行相同的查询可能因此而返回不同的结果。我们说后备服务器上的数据与主服务器是最终一致的。一旦一个事务的提交记录在后备服务器上被重播，那个事务所作的修改将对后备服务器上所有新取得的快照可见。快照可以在每个查询或每个事务的开始时取得，这取决于当前的事务隔离级别。详见Section 13.2

在热备期间开始的事务可能发出下列命令：

- 查询访问：SELECT、COPY TO
- 游标命令：DECLARE、FETCH、CLOSE
- 设置：SHOW、SET、RESET
- 事务管理命令：
  - BEGIN、END、ABORT、START TRANSACTION
  - SAVEPOINT、RELEASE、ROLLBACK TO SAVEPOINT
  - EXCEPTION块或其他内部子事务
- LOCK TABLE，不过只在下列模式之一中明确发出：ACCESS SHARE、ROW SHARE 或 ROW EXCLUSIVE.
- 计划和资源：PREPARE、EXECUTE、DEALLOCATE、DISCARD
- 插件和扩展：LOAD
- UNLISTEN

在热备期间开始的事务将不会被分配一个事务 ID 并且不能被写入到系统的预写式日志。因此，下列动作将产生错误消息：

- 数据操纵语言 (DML)：INSERT、UPDATE、DELETE、COPY FROM、TRUNCATE。注意不允许在恢复期间导致一个触发器被执行的动作。这个限制甚至被应用到临时表，因为不分配事务 ID 表行就不能被读或写，而当前不可能在一个热备环境中分配事务 ID。
- 数据定义语言 (DDL)：CREATE、DROP、ALTER、COMMENT。这个限制甚至被应用到临时表，因为执行这些操作会要求更新系统目录表。
- SELECT ... FOR SHARE | UPDATE，因为不更新底层数据文件就无法取得行锁。
- SELECT语句上的能产生 DML 命令的规则。
- 显式请求一个高于ROW EXCLUSIVE MODE的模式的LOCK。
- 默认短形式的LOCK，因为它请求ACCESS EXCLUSIVE MODE。
- 显式设置非只读状态的事务管理命令：
  - BEGIN READ WRITE, START TRANSACTION READ WRITE

- SET TRANSACTION READ WRITE, SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
- SET transaction\_read\_only = off
- 两阶段提交命令: PREPARE TRANSACTION、COMMIT PREPARED、ROLLBACK PREPARED, 因为即使只读事务也需要在准备阶段（两阶段提交中的第一个阶段）写 WAL。
- 序列更新 : nextval()、setval()
- LISTEN, NOTIFY

在正常操作中，“只读”事务被允许使用LISTEN和NOTIFY，因此热备会话在比普通只读会话更紧一点的限制下操作。这些限制中的某些可能会在一个未来的发行中被放松。

在热备期间，参数transaction\_read\_only总是为真并且不可以被改变。但是只要不尝试修改数据库，热备期间的连接工作起来更像其他数据库连接。如果发生故障转移或切换，该数据库将切换到正常处理模式。当服务器改变模式时会话将保持连接。一旦热备结束，它可以发起读写事务（即使是一个在热备期间启动的会话）。

用户可以通过SHOW in\_hot\_standby来检查hot standby会话是否是活跃的（在服务器版本14之前该参数in\_hot\_standby不存在。对于更早版本的服务器，可行的替代方法是SHOW transaction\_read\_only。）此外，还有一些函数（Table 9.87 允许用户访问有关备用服务器的信息。它们允许您编写程序来识别数据库当前的状态。用于监控恢复进度，或者您可以编写复杂的程序将数据库恢复到特定状态。

## 24.5.2. 处理查询冲突

主服务器和后备服务器在多方面都松散地连接在一起。主服务器上的动作将在后备服务器上产生效果。结果是在它们之间有潜在的副作用或冲突。最容易理解的冲突是性能：如果在主服务器上发生一次大数据量的载入，那么着将在后备服务器上产生一个相似的 WAL 记录流，因而后备服务器查询可能要竞争系统资源（例如 I/O）。

随着热备发生的还可能还有其他类型的冲突。对于可能需要被取消的查询和（某些情况中）解决它们的已断开会话来说，这些冲突是硬冲突。用户可以用几种方式来处理这种冲突。冲突情况包括：

- 在主服务器上取得了访问排他锁（包括显式LOCK命令和多种DDL动作）与后备查询中的表访问冲突。
- 在主服务器上删除一个表空间与使用该表空间存储临时工作文件的后备查询冲突。
- 在主服务器上删除一个数据库与在后备服务器上连接到该数据库的会话冲突。
- 从 WAL 清除记录的应用与快照仍能“看见”任意要被移除的行的后备事务冲突。
- 从 WAL 清除记录的应用与在后备服务器上访问该目标页的查询冲突，不管要被移除的数据是否为可见。

在主服务器上，这些情况仅仅会导致等待；并且用户可以选择取消这些冲突动作中间的一个。但是，在后备服务器上则没有选择：已被 WAL 记录的动作已经在主服务器上发生，那么后备服务器不能在应用它时失败。此外，允许 WAL 应用无限等待是非常不可取的，因为后备服务器的状态将变得逐渐远远落后于主服务器的状态。因此，提供了一种机制来强制性地取消与要被应用的 WAL 记录冲突的后备查询。

该问题情形的一个例子是主服务器上的一位管理员在一个表上运行DROP TABLE，而该表正在后备服务器上被查询。如果DROP TABLE被应用在后备服务器上，很明显该后备查询不能继续。如果这种情况在主服务器上发生，DROP TABLE将等待直到其他查询结束。但是当DROP TABLE被运行在主服务器上，主服务器没有关于运行在后备服务器上查询的信息，因

此它不会等待任何这样的后备查询。WAL 改变记录在后备查询还在运行时来到后备服务器上，导致一个冲突。后备服务器必须要么延迟 WAL 记录的应用（还有它们之后的任何事情），要么取消冲突查询这样DROP TABLE可以被应用。

当一个冲突查询很短时，我们通常期望能延迟 WAL 应用一小会儿让它完成；但是在 WAL 应用中一段长的延迟通常是不受欢迎的。因此取消机制有参数，`max_standby_archive_delay`和`max_standby_streaming_delay`，它们定义了WAL应用中的最大允许延迟。当应用任何新收到的WAL数据花费了超过相关延迟设置值时，冲突查询将被取消。设立两个参数是为了对从一个归档读取WAL数据（即来自一个基础备份的初始恢复或者“追赶”一个已经落后很远的后备服务器）和通过流复制读取WAL数据的两种情况指定不同的延迟值。

在一台后备服务器上这主要是为了该可用性而存在，最好把延迟参数设置得比较短，这样服务器不会由于后备查询导致的延迟落后主服务器太远。但是，如果该后备服务器是位了执行长时间运行的查询，则一个较高甚至无限的延迟值更好。但是记住一个长时间运行的查询延迟了WAL记录的应用，它可能导致后备服务器上的其他会话无法看到主服务器上最近的变化。

一旦`max_standby_archive_delay`或`max_standby_streaming_delay`指定的延迟被超越，冲突查询将被取消。这通常仅导致一个取消错误，尽管在重放一个DROP DATABASE的情况下整个冲突会话都将被中断。另外，如果冲突发生在一个被空闲事务持有的锁上，该冲突会话会被中断（这种行为可能在未来被改变）。

被取消的查询可能会立即被重试（当然是在开始一个新的事务后）。因为查询取消依赖于WAL记录被重放的本质，如果一个被取消的查询被再次执行，它可能会很好地成功完成。

记住延迟参数是从WAL数据被后备服务器收到后流逝的时间。因此，留给后备服务器上任何一个查询的宽限期从不会超过延迟参数，并且如果后备服务器已经由于等待之前的查询完成而落后或者因为过重的更新负载而无法跟上主服务器，宽限期可能会更少。

在后备查询和WAL重播之间发生冲突的最常见原因是“过早清除”。正常地，LightDB允许在没有事务需要看到旧行版本时对它们进行清除，这样可以保证根据MVCC规则的正确的数据可见性。不过，这个规则只能被应用于执行在主控机上的事务。因此有可能主控机上的清除会移除对一个后备服务器事务还可见的行版本。

有经验的用户应当注意行版本清除和行版本冻结都可能与后备查询冲突。即便在一个没有被更新或被删除行的表上运行一次手工VACUUM FREEZE也可能导致冲突。

用户应当清楚，主服务器上被正常和重度更新的表将快速地导致后备服务器上长时间运行的查询被取消。在这样的情况下，`max_standby_archive_delay`或`max_standby_streaming_delay`的有限制设置可以被视作`statement_timeout`设置。

如果发现后备查询取消的数量不可接受，还是有补救的可能。第一种选项是设置参数`hot_standby_feedback`，它阻止VACUUM移除最近死亡的元组并且因此清除冲突不会产生。如果你这样做，你应当注意这将使主服务器上的死亡元组清除被延迟，这可能会导致不希望发生的表膨胀。不过，清除的情况不会比在主服务器上直接运行后备查询时更糟，并且你仍然能够享受将执行分流到后备服务器的好处。如果后备服务器频繁地连接和断开，你可能想要做些调整来处理无法提供`hot_standby_feedback`反馈的时期。例如，考虑增加`max_standby_archive_delay`，这样在断开连接的期间查询就不会快速地被WAL归档文件中的冲突取消。你也应该考虑增加`max_standby_streaming_delay`来避免重新连接后新到达的流WAL项导致的快速取消。

另一个选项是增加主服务器上的`vacuum_defer_cleanup_age`，这样死亡行不会像平常那么快地被清理。这将允许在后备服务器上的查询能在被取消前有更多时间执行，并且不需要设置一个很高的`max_standby_streaming_delay`。但是，这种方法很难保证任何指定的执行时间窗口，因为`vacuum_defer_cleanup_age`是用主服务器上被执行的事务数来衡量的。

查询取消的数量和原因可以使用后备服务器上的`pg_stat_database_conflicts`系统视图查看。`pg_stat_database`系统视图也包含汇总信息。

### 24.5.3. 管理员概览

如果`hot_standby`在`lightdb.conf`中被设置为`on`并且存在一个`standby.signal`文件，服务器将运行在热备模式。但是，可能需要一些时间来允许热备连接，因为在服务器完成足够的恢复来为查询提供一个一致的状态之前，它将不会接受连接。在此期间，尝试连接的客户端将被一个错误消息拒绝。要确认服务器已经可连接，要么循环地从应用尝试连接，要么在服务器日志中查找这些消息：

```
LOG: entering standby mode
```

```
... then some time later ...
```

```
LOG: consistent recovery state reached
```

```
LOG: database system is ready to accept read only connections
```

在主服务器上，一旦创建一个检查点，一致性信息就被记录下来。当读取在特定时段（当在主服务器上`wal_level`没有被设置为`replica`或者`logical`的期间）产生的 WAL 时无法启用热备。在同时存在这些条件时，到达一个一致状态也会被延迟：

- 一个写事务有超过 64 个子事务
- 生存时间非常长的写事务

如果你正在运行基于文件的日志传送（“温备”），你可能需要等到下一个 WAL 文件到达，这可能和主服务器上的`archive_timeout`设置一样长。

设置几个参数可确定用于跟踪事务ID、锁和预备事务的共享内存大小。备用服务器上的设置必须大于或等于主服务器上的设置，以确保在恢复过程中不会耗尽共享内存。例如，如果主数据库正在执行预备事务，而备用数据库没有获取共享内存来跟踪预备事务，则备用数据库将无法继续恢复，直到配置更改。受影响的参数是：

- `max_connections`
- `max_prepared_transactions`
- `max_locks_per_transaction`
- `max_wal_senders`
- `max_worker_processes`

确保这不是问题的可靠方法是使备用数据库上的这些参数的值等于或大于主数据库上的值。因此，如果您想增加这些值，您应该先更改备用服务器上的设置，然后再更改主服务器上的设置。相反，如果要减小这些值，则应先更改主服务器上的设置，然后再更改备用服务器上的设置。请记住，当一个备用数据库被提升时，它会成为后续备用数据库所需参数设置的新基准。因此，最好在所有备用服务器上保持这些设置相同，这样在切换/故障转移期间就不会出现问题。

管理员为`max_standby_archive_delay`和`max_standby_streaming_delay`选择适当的设置很重要。最好的选择取决于业务的优先级。例如如果服务器主要的任务是作为高可用服务器，那么你将想要低延迟设置，甚至是零（尽管它是一个非常激进的设置）。如果后备服务器的任务是作为一个用于决策支持查询的额外服务器，那么将其最大延迟值设置为很多小时甚至-1（表示无限等待）可能都是可以接受的。

在主服务器上写出的事务状态“hint bits”是不被 WAL 记录的，因此后备服务器上的数据将可能重新写出该提示。这样，即使所有用户都是只读的，后备服务器仍将执行磁盘写操作；但数据值本身并没有发生改变。用户将仍写出大的排序临时文件并且重新生成 `relcache` 信息文件，这样在热备模式中数据库没有哪个部分是真正只读的。还要注意使用`dblink`模块写

到远程数据库以及其他使用 PL 函数位于数据库之外的操作仍将可用，即使该事务是本地只读的。

在恢复模式期间，下列类型的管理命令是不被接受的：

- 数据定义语言（DDL）：e.g., CREATE INDEX
- 特权和所有权：GRANT, REVOKE,
- 维护命令：ANALYZE, VACUUM, CLUSTER, REINDEX

注意这些命令中的某些实际上在主服务器上的“只读”模式事务期间是被允许的。

结果是，你无法创建只存在于后备服务器上的额外索引以及统计信息。如果需要这些管理命令，它们应该在主服务器上被执行，并且最后那些改变将被传播到后备服务器。

`pg_cancel_backend()` 和 `pg_terminate_backend()` 将在用户后端上工作，而不是执行恢复的 Startup 进程。`pg_stat_activity` 不会为 Startup 进程显示一个项，也不会把恢复事务显示为活动。结果是在恢复期间 `pg_prepared_xacts` 总是为空。如果你希望解决不能确定的预备事务，查看主服务器上的 `pg_prepared_xacts` 并且发出命令来解决那里的事务或者在恢复结束后来解决它们。

和平常一样，`pg_locks` 将显示被后端持有的锁。`pg_locks` 也会显示一个由 Startup 进程管理的虚拟事务，它拥有被恢复重播的事务所持有的所有 `AccessExclusiveLocks`。注意 Startup 进程不请求锁来做数据库更改，并且因此对于 Startup 进程除 `AccessExclusiveLocks` 之外的锁不显示在 `pg_locks` 中，它们仅被假定存在。

Nagios 的插件 `check_pgsql` 将可以工作，因为它检查的简单信息是存在的。`check_postgres` 监控脚本也将能工作，尽管某些被报告的值可能给出不同或者混乱的结果。例如，上一次清理时间将不会被维护，因为在后备服务器上不会发生清理。在主服务器上运行的清理仍会把它们的改变发送给后备服务器。

WAL 文件控制命令在恢复期间将不会工作，如 `pg_start_backup`、`pg_switch_wal` 等。

可动态载入的模块可以工作，包括 `pg_stat_statements`。

咨询锁在恢复期间工作正常，包括死锁检测。注意咨询锁从来都不会被 WAL 记录，因此在主服务器或后备服务器上一个咨询锁不可能与 WAL 重播发生冲突。也不可能会在主服务器上获得一个咨询锁并且在后备服务器上开始一个相似的咨询锁。咨询锁只与它们被取得的那个服务器相关。

基于触发器的复制系统（如 Slony、Londiste 和 Bucardo）将根本不会运行在后备服务器上，然而只要改变不被发送到要被应用的后备服务器，它们将在主服务器上运行得很好。WAL 重播不是基于触发器的，因此你不能用后备服务器接替任何需要额外数据库写操作或依赖触发器使用的系统。

新的 OID 不能被分配，然而某些 UUID 生成器仍然能工作，只要它们不依赖于向数据库写新的状态。

当前，在只读事务期间不允许创建临时表，因此在某些情况中现有的脚本将不会正确运行。这个限制可能会在稍后的发行中被放松。这既是一个 SQL 标准符合问题也是一个技术问题。

只有在表空间为空时 `DROP TABLESPACE` 才能成功。某些后备服务器用户可能正在通过他们的 `temp_tablespaces` 参数使用该表空间。如果在该表空间中有临时文件，所有活动查询将被取消来保证临时文件被移除，这样该表空间可以被移除并且 WAL 重播可以继续。

在主服务器上运行 `DROP DATABASE` 或 `ALTER DATABASE ... SET TABLESPACE` 将产生一个 WAL 项，它将导致所有连接到后备服务器上那个数据库的用户被强制地断开连接。这个动作会立即发生，不管 `max_standby_streaming_delay` 的设置是什么。注意 `ALTER DATABASE ... RENAME` 不会断开用户，这在大部分情况中不会有提示，然而如果它依赖某种基于数据库名的方法，在某些情况中会导致程序混乱。

在普通（非恢复）模式中，如果你为具有登录能力的角色发出DROP USER或DROP ROLE，而该用户仍然连接着，则对已连接用户不会发生任何事情 - 他们保持连接。但是用户不能重新连接。这种行为也适用于恢复，因此在主服务器上的一次DROP USER不会使后备服务器上的用户断开。

在恢复期间统计收集器是活动的。所有扫描、读、阻塞、索引使用等将在后备服务器上被正常的记录。被重播的动作将不会重复它们在主服务器上的效果，因此重播一个插入将不会导致pg\_stat\_user\_tables的Inserts列上的递增。在恢复的开始stats文件会被删除，因此来自主服务器和后备服务器的stats将不同；这被认为是一种特性而不是缺陷。

在恢复期间自动清理不是活动的。它将在恢复末尾正常启动。

检查点进程和后台写入进程在恢复期间是活动状态的。检查点进程将执行重启点（与主服务器上的检查点相似），后台写入进程将执行正常的块清理活动。这可以包括存储在后备服务器上的提示位信息的更新。在恢复期间，CHECKPOINT命令会被接受，然而它会执行一个重启点而不是一个新的检查点。

## 24.5.4. 热备参数参考

多个参数已经在Section 24.5.3和Section 24.5.4中提到过。

在主服务器上，可以使用参数wal\_level和vacuum\_defer\_cleanup\_age。在主服务器上设置max\_standby\_archive\_delay和max\_standby\_streaming\_delay不会产生效果。

在主服务器上，可以使用参数hot\_standby、max\_standby\_archive\_delay和max\_standby\_streaming\_delay。只要服务器保持在后备模式vacuum\_defer\_cleanup\_age就没有效果，然而当后备服务器变成主服务器时它将变得相关。

## 24.5.5. 警告

热备有一些限制。这些限制很可能在未来的发行中被修复：

- 在能够取得快照之前，需要正在运行的事务的完整知识。使用大量事务（目前指超过64个）的事务将延迟只读连接的启动，直到最长的运行着的写事务完成。如果发生这种情况，说明消息将被发送到服务器日志。
- 主服务器上的每一个检查点将产生用于后备查询的可用启动点。如果后备服务器在主机处于关闭状态时被关闭，就没有办法在主服务器启动之前重新进入热后备，因此它在WAL日志中产生一个进一步启动点。这种情况在它可能发生的大部分常见情况中不是一个问题。通常，如果主服务器被关闭并且不再可用，这可能是由于某种严重错误要求后备服务器被转变成为一个新的主服务器来操作。并且在主服务器被故意关闭的情况下，协调保证后备服务器平滑地过渡为新的主服务器也是一种标准过程。
- 在恢复尾声，由预备事务持有的AccessExclusiveLocks将要求两倍的正常锁表项。如果你计划运行大量并发的通常要求AccessExclusiveLocks的预备事务，或者你计划运行一个需要很多AccessExclusiveLocks的大型事务，我们建议你为max\_locks\_per\_transaction选择一个更大的值，也许是主服务器上该参数值的两倍。如果你的max\_prepared\_transactions设置为0，你根本不需要考虑这个问题。
- 可序列化事务隔离级别目前在热备中不可用（详见Section 13.2.3和Section 13.4.1）。尝试在热备模式中将一个事务设置为可序列化隔离级别将产生一个错误。

---

## Chapter 25. 监控数据库活动

一个数据库管理员常常会疑惑，“系统现在正在做什么？”这一章会讨论如何搞清楚这个问题。

一些工具可以用来监控数据库活动并且分析性能。这一章的大部分都致力于描述LightDB的统计收集器，但是我们也不能忽视常规的 Unix 监控程序，如ps、top、iostat和vmstat。另外，一旦我们发现了一个性能差的查询，可能需要LightDB的EXPLAIN命令来进行进一步的调查。Section 14. 会讨论EXPLAIN以及其他用来理解个体查询行为的方法。

### 25.1. 标准 Unix 工具

在大部分 Unix 平台上，LightDB会修改由ps报告的命令标题，这样个体服务器进程可以被标识。一个显示样例是

```
$ ps auxww | grep ^lightdb
lightdb 15551 0.0 0.1 57536 7132 pts/0 S 18:02 0:00 lightdb -i
lightdb 15554 0.0 0.0 57536 1184 ? Ss 18:02 0:00 lightdb:
background writer
lightdb 15555 0.0 0.0 57536 916 ? Ss 18:02 0:00 lightdb:
checkpointer
lightdb 15556 0.0 0.0 57536 916 ? Ss 18:02 0:00 lightdb:
walwriter
lightdb 15557 0.0 0.0 58504 2244 ? Ss 18:02 0:00 lightdb:
autovacuum launcher
lightdb 15558 0.0 0.0 17512 1068 ? Ss 18:02 0:00 lightdb: stats
collector
lightdb 15582 0.0 0.0 58772 3080 ? Ss 18:04 0:00 lightdb: joe
runbug 127.0.0.1 idle
lightdb 15606 0.0 0.0 58772 3052 ? Ss 18:07 0:00 lightdb: tgl
regression [local] SELECT waiting
lightdb 15610 0.0 0.0 58772 3056 ? Ss 18:07 0:00 lightdb: tgl
regression [local] idle in transaction
```

(ps的调用方式随不同的平台而变，但是显示的细节都差不多。这个例子来自于一个最近的Linux 系统)。列在这里的第一个进程是主服务器进程。为它显示的命令参数是当它被启动时使用的那些。接下来的五个进程是由主进程自动启动的后台工作者进程（如果你已经设置系统为不启动统计收集器，“统计收集器”进程将不会出现；同样“自动清理发动”进程也可以被禁用）。剩余的每一个进程都是一个处理一个客户端连接的服务器进程。每个这种进程都会把它的命令行显示设置为这种形式

```
lightdb: user database host activity
```

在该客户端连接的生命期中，用户、数据库以及（客户端）主机项保持不变，但是活动指示器会改变。活动可以是闲置（即等待一个客户端命令）、在事务中闲置（在一个BEGIN块里等待客户端）或者一个命令类型名，例如SELECT。还有，如果服务器进程正在等待一个其它会话持有的锁，等待中会被追加到上述信息中。在上面的例子中，我们可以推断：进程 15606 正在等待进程 15610 完成其事务并且因此释放一些锁（进程 15610 必定是阻塞者，因为没有其他活动会话。在更复杂的情况中，可能需要查看pg\_locks系统视图来决定谁阻塞了谁）。

如果配置了cluster\_name，则集簇的名字 也将会显示在ps的输出中：



```
$ ltsql -c 'SHOW cluster_name'
cluster_name
-----
server1
(1 row)

$ ps aux|grep server1
lightdb 27093 0.0 0.0 30096 2752 ?        Ss  11:34   0:00 lightdb:
server1: background writer
...
```

如果你已经关闭了update\_process\_title，那么活动指示器将不会被更新，进程标题仅在新进程被启动的时候设置一次。在某些平台上这样做可以为每个命令节省可观的开销，但在其它平台上却不明显。

### Tip

Solaris需要特别的处理。你必需使用/usr/ucb/ps而不是/bin/ps。你还必需使用两个w标志，而不是一个。另外，你对lightdb命令的最初调用必须用一个比服务器进程提供的短的ps状态显示。如果你没有满足全部三个要求，每个服务器进程的ps输出将是原始的lightdb命令行。 command line.

## 25.2. 统计收集器

LightDB的统计收集器是一个支持收集和报告服务器活动信息的子系统。目前，这个收集器可以对表和索引的访问计数，计数可以按磁盘块和个体行来进行。它还跟踪每个表中的总行数、每个表的清理和分析动作的信息。它也统计调用用户定义函数的次数以及在每次调用中花费的总时间。

LightDB也支持报告有关系统正在干什么的动态信息，例如当前正在被其他服务器进程执行的命令以及系统中存在哪些其他连接。这个功能是独立于收集器进程存在的。

### 25.2.1. 统计收集配置

因为统计收集给查询执行增加了一些负荷，系统可以被配置为收集或不收集信息。这由配置参数控制，它们通常在lightdb.conf中设置（关于设置配置参数的细节请见Chapter 17）。

参数track\_activities允许监控当前被任意服务器进程执行的命令。

参数track\_counts控制是否收集关于表和索引访问的统计信息。

参数track\_functions启用对用户定义函数使用的跟踪。

参数track\_io\_timing启用对块读写次数的监控。

通常这些参数被设置在lightdb.conf中，这样它们会应用于所有服务器进程，但是可以在单个会话中使用SET命令打开或关闭它们（为了阻止普通用户对管理员隐藏他们的活动，只有超级用户被允许使用SET来改变这些参数）。

统计收集器通过临时文件将收集到的信息传送给其他LightDB进程。这些文件被存储在名字由stats\_temp\_directory参数指定的目录中，默认是lt\_stat\_tmp。为了得到更好的性能，stats\_temp\_directory可以被指向一个基于RAM的文件系统来降低物理I/O需求。当服务器被干净地关闭时，一份统计数据的永久拷贝被存储在lt\_stat子目录中，这样在服务器重启后统计信息能被保持。当在服务器启动时执行恢复时（例如立即关闭、服务器崩溃以及时间点恢复之后），所有统计计数器会被重置。

## 25.2.2. 查看统计信息

Table 25. 中列出了一些预定义视图 可以用来显示系统的当前状态。 Table 25. 中列出了另一些视图可以 显示统计收集的结果。你也可以使用底层统计函数（在 Section 25.2.2中讨论）来建立自定义的视图。

在使用统计信息监控收集到的数据时，你必须了解这些信息并非实时更新的。每个独立的服务器进程只在进入闲置状态之前才向收集器传送新的统计计数；因此正在进行的查询或事务并不影响显示出来的总数。同样，收集器本身也最多每PGSTAT\_STAT\_INTERVAL毫秒（缺省为 500ms，除非在编译服务器的时候修改过）发送一次新的报告。因此显示的信息总是落后于实际活动。但是由track\_activities收集的当前查询信息总是最新的。

另一个重点是当一个服务器进程被要求显示任何这些统计信息时，它首先取得收集器进程最近发出的报告并且接着为所有统计视图和函数使用这个快照，直到它的当前事务的结尾。因此只要你继续当前事务，统计数据将会一直显示静态信息。相似地，当任何关于所有会话的当前查询的信息在一个事务中第一次被请求时，这样的信息将被收集。并且在整个事务期间将显示相同的信息。这是一种特性而非缺陷，因为它允许你在该统计信息上执行多个查询并且关联结果而不用担心那些数字会在你不知情的情况下改变。但是如果你希望用每个查询都看到新结果，要确保在任何事务块之外做那些查询。或者，你可以调用pg\_stat\_clear\_snapshot()，那将丢弃当前事务的统计快照（如果有）。下一次对统计性信息的使用将导致获取一个新的快照。

一个事务也可以在视

图pg\_stat\_xact\_all\_tables、pg\_stat\_xact\_sys\_tables、pg\_stat\_xact\_user\_tables和pg\_stat\_xact\_user中看到它自己的统计信息（还没有被传送给收集器）。这些数字并不像上面所述的那样行动，相反它们在事务期间持续被更新。

Table 25. 中显示的动态统计视图中的一些信息是有安全限制的。普通用户只能看到关于他们自己的会话的所有信息（属于他们是成员的角色会话）。在关于其他会话的行中，许多列将为空。但是，请注意，一个会话的存在和它的一般属性，例如会话用户和数据库，对所有用户都是可见的。超级用户和内置角色pg\_read\_all\_stats的成员（参见Section 19.5 可以看到所有会话的所有信息。

Table 25.1. 动态统计视图

视图名称	描述
pg_stat_activity	每个服务器进程一行，显示与那个进程的当前活动相关的信息，例如状态和当前查询。详见pg_stat_activity。
pg_stat_replication	每一个 WAL 发送进程一行，显示有关到该发送进程连接的后备服务器的复制的统计信息。详见pg_stat_replication。
pg_stat_wal_receiver	只有一行，显示来自 WAL 接收器所连接服务器的有关该接收器的统计信息。详见pg_stat_wal_receiver。
pg_stat_subscription	每个订阅至少一行，显示有关该订阅的工作者的信息。详见 pg_stat_subscription。
pg_stat_ssl	每个连接（常规的或者复制）一行，显示在这个连接上使用的SSL的信息。详见pg_stat_ssl。
pg_stat_gssapi	每个连接（常规和复制）有一行，显示关于GSSAPI验证和加密的信息。详情请参阅pg_stat_gssapi。
pg_stat_progress_analyze	每个运行ANALYZE的后端(包括自动清理工作者进程)的行，显示当前进度。参见Section 25.4.1

视图名称	描述
pg_stat_progress_create_index	每个后台运行CREATE INDEX或REINDEX的后端都有一行，显示当前的进度。参见Section 25.4.2
pg_stat_progress_vacuum	每个运行着VACUUM的后端（包括autovacuum工作者进程）一行，显示当前的进度。详见Section 25.4.3
pg_stat_progress_cluster	每个运行着CLUSTER或VACUUM FULL的后端一行，显示当前进度。参见 Section 25.4.4
pg_stat_progress_basebackup	每一个WAL发送者进程的行显示一个基础备份，显示当前进度。参见Section 25.4.5

Table 25.2. 已收集统计信息的视图

视图名称	描述
pg_stat_archiver	只有一行，显示有关 WAL 归档进程活动的统计信息。详见 pg_stat_archiver。
pg_stat_bgwriter	只有一行，显示有关后台写进程的活动的统计信息。详见 pg_stat_bgwriter。
pg_stat_database	每个数据库一行，显示数据库范围的统计信息。详见 pg_stat_database。
pg_stat_database_conflicts	每个数据库一行，显示数据库范围的统计信息， 这些信息的内容是关于由于与后备服务器的恢复过程 发生冲突而被取消的查询。详见 pg_stat_database_conflicts。
pg_stat_all_tables	当前数据库中每个表一行，显示有关访问指定表的统计信息。详见 pg_stat_all_tables。
pg_stat_sys_tables	和pg_stat_all_tables一样，但只显示系统表。
pg_stat_user_tables	和pg_stat_all_tables一样，但只显示用户表。
pg_stat_xact_all_tables	和pg_stat_all_tables相似，但计数动作只在当前事务内发生（还没有被包括在pg_stat_all_tables和相关视图中）。用于生存和死亡行数量的列以及清理和分析动作在此视图中不出现。
pg_stat_xact_sys_tables	和pg_stat_xact_all_tables一样，但只显示系统表。
pg_stat_xact_user_tables	和pg_stat_xact_all_tables一样，但只显示用户表。
pg_stat_all_indexes	当前数据库中的每个索引一行，显示：表OID、索引OID、模式名、表名、索引名、使用了该索引的索引扫描总数、索引扫描返回的索引记录数、使用该索引的简单索引扫描抓取的活表(livetable)中数据行数。当前数据库中的每个索引一行，显示与访问指定索引有关的统计信息。详见 pg_stat_all_indexes。
pg_stat_sys_indexes	和pg_stat_all_indexes一样，但只显示系统表上的索引。

视图名称	描述
pg_stat_user_indexes	和pg_stat_all_indexes一样，但只显示用户表上的索引。
pg_statio_all_tables	当前数据库中每个表一行(包括TOAST表)，显示：表OID、模式名、表名、从该表中读取的磁盘块总数、缓冲区命中次数、该表上所有索引的磁盘块读取总数、该表上所有索引的缓冲区命中总数、在该表的辅助TOAST表(如果存在)上的磁盘块读取总数、在该表的辅助TOAST表(如果存在)上的缓冲区命中总数、TOAST表的索引的磁盘块读取总数、TOAST表的索引的缓冲区命中总数。当前数据库中的每个表一行，显示有关在指定表上 I/O 的统计信息。详见 pg_statio_all_tables。
pg_statio_sys_tables	和pg_statio_all_tables一样，但只显示系统表。
pg_statio_user_tables	和pg_statio_all_tables一样，但只显示用户表。
pg_statio_all_indexes	当前数据库中每个索引一行，显示：表OID、索引OID、模式名、表名、索引名、该索引的磁盘块读取总数、该索引的缓冲区命中总数。当前数据库中的每个索引一行，显示与指定索引上的 I/O 有关的统计信息。详见 pg_statio_all_indexes。
pg_statio_sys_indexes	和pg_statio_all_indexes一样，但只显示系统表上的索引。
pg_statio_user_indexes	和pg_statio_all_indexes一样，但只显示用户表上的索引。
pg_statio_all_sequences	当前数据库中每个序列对象一行，显示：序列OID、模式名、序列名、序列的磁盘读取总数、序列的缓冲区命中总数。当前数据库中的每个序列一行，显示与指定序列上的 I/O 有关的统计信息。详见 pg_statio_all_sequences。
pg_statio_sys_sequences	和pg_statio_all_sequences一样，但只显示系统序列（目前没有定义系统序列，因此这个视图总是为空）。
pg_statio_user_sequences	和pg_statio_all_sequences一样，但只显示用户序列。
pg_stat_user_functions	对于所有跟踪功能，函数的OID，模式，名称，数量 通话总时间，和自我的时间。自我时间是 在函数本身所花费的时间量，总时间包括 它调用函数所花费的时间。时间值以毫秒为单位。每一个被跟踪的函数一行，显示与执行该函数有关的统计信息。详见 pg_stat_user_functions。
pg_stat_xact_user_functions	和pg_stat_user_functions相似，但是只统计在当前事务期间的调用（还没有被包括在pg_stat_user_functions中）。
pg_stat_slru	每个SLRU一行，显示操作的统计信息。参见 pg_stat_slru for details.

针对每个索引的统计信息对于判断哪个索引正被使用以及它们的效果特别有用。

pg\_statio 系列视图主要用于判断缓冲区的效果。当实际磁盘读取数远小于缓冲区命中时，这个缓冲能满足大部分读请求而无需进行内核调用。但是，这些统计信息并没有给出所有的事情：由于LightDB处理磁盘 I/O 的方式，不在LightDB缓冲区中的数据库仍然驻留在内核的 I/O 缓存中，并且因此可以被再次读取而不需要物理磁盘读取。我们建议希望了解LightDB I/O 行为更多细节的用户将LightDB统计收集器和操作系统中允许观察内核处理 I/O 的工具一起使用。

### 25.2.3. pg\_stat\_activity

pg\_stat\_activity视图每个服务器进程将有一行，显示与该进程当前活动相关的信息。

Table 25.3. pg\_stat\_activity 视图

列类型	描述
datid oid	这个后端连接到的数据库的OID
datname name	这个后端连接到的数据库的名称
pid integer	这个后端的进程 ID
leader_pid integer	并行组组长的进程ID，如果该进程是并行查询工作者。如果该进程是一个并行组的组长或不参与并行查询，则为NULL。
usesysid oid	登录到这个后端的用户的 OID
username name	登录到这个后端的用户的 OID
application_name text	连接到这个后端的应用的名称
client_addr inet	连接到这个后端的客户端的 IP 地址。如果这个字段为空，它表示客户端通过服务器机器上的一个 Unix 套接字连接或者这是一个内部进程，如自动清理。
client_hostname text	已连接的客户端的主机名，由client_addr的反向 DNS 查找报告。这个字段将只对 IP 连接非空，并且只有 log_hostname被启用时才会非空。
client_port integer	客户端用于与此后端通信的TCP端口号，如果使用Unix套接字，则为-1。如果该字段为空，它表示这是一个内部服务器进程。
backend_start timestamp with time zone	这个进程被启动的时间。对客户端后端来说，这就是客户端连接到服务器的时间。
xact_start timestamp with time zone	这个进程的当前事务被启动的时间，如果没有活动事务则为空。如果当前查询是它的第一个事务，这一列等于query_start列。
query_start timestamp with time zone	当前活动查询被开始的时间，如果state不是active，则为上一个查询开始的时间
state_change timestamp with time zone	state上一次被改变的时间
wait_event_type text	后端等待的事件类型，如果有的话;否则NULL。参见Table 25.4

列类型	描述
wait_event text	如果后端当前正在等待，则等待事件名称，否则为NULL。参见Table 25.5到Table 25.13
state text	这个后端的当前总体状态。可能的值为： <ul style="list-style-type: none"> <li>• active: 后端正在执行一个查询。</li> <li>• idle: 后端正在等待一个新的客户端命令。</li> <li>• idle in transaction: 后端在一个事务中，但是当前没有正在执行一个查询。</li> <li>• idle in transaction (aborted): 这个状态与 idle in transaction相似，除了在该事务中的一个语句导致了一个错误。</li> <li>• fastpath function call: 后端正在执行一个 fast-path 函数。</li> <li>• disabled: 如果在这个后端中track_activities被禁用，则报告这个状态。</li> </ul>
backend_xid xid	这个后端的顶层事务标识符，如果存在。
backend_xmin xid	当前后端的xmin范围。
query text	这个后端最近查询的文本。如果state为active，这个字段显示当前正在执行的查询。在所有其他状态下，它显示上一个被执行的查询。默认情况下，查询文本会被截断至1024个字节，这个值可以通过参数track_activity_query_size更改。
backend_type text	当前后端的类型。可能的类型为 autovacuum launcher, autovacuum worker, logical replication launcher, logical replication worker, parallel worker, background writer, client backend, checkpoint, startup, walreceiver, walsender and walwriter. 除此以外，由扩展注册的后台Worker可能有额外的类型。

**Note**

wait\_event和state列是独立的。如果一个后端处于active状态，它可能是也可能不是某个事件上的waiting。如果状态是active并且wait\_event为非空，它意味着一个查询正在被执行，但是它被阻塞在系统中某处。

Table 25.4. 等待事件类型

等待事件类型	描述
Activity	服务器进程空闲。此事件类型表示在其主处理循环中等待活动的进程。wait_event将识别特定的等待点;参见Table 25.5
BufferPin	服务器进程正在等待对数据缓冲的独占访问。如果另一个进程持有一个打开的游标，该游标最后一次从相关缓冲区读取数据，则缓冲区销等待可能是漫长的。参见Table 25.6
Client	服务器进程正在等待连接到用户应用程序的套接字上的活动。因此，服务器预计发生一

等待事件类型	描述
	些独立于其内部进程的事情。wait_event将识别特定的等待点；参见Table 25.7
Extension	服务器进程正在等待扩展模块定义的某个条件。参见Table 25.8
I/O	服务器进程正在等待一个I/O操作完成。wait_event将识别特定的等待点；参见Table 25.9
IPC	服务器进程正在等待与另一个服务器进程进行交互。wait_event将识别特定的等待点；参见Table 25.10
Lock	服务器进程正在等待一个重量级锁。重量级锁，也称为锁管理器锁或简单锁，主要保护表等SQL可见对象。然而，它们也用于确保某些内部操作的互斥，例如关系扩展。wait_event将识别等待的锁的类型；参见Table 25.11
LWLock	服务器进程正在等待一个轻量级锁。大多数这样的锁保护共享内存中的特定数据结构。wait_event将包含标识轻量级锁用途的名称。（有些锁有特定的名称；其他锁是一组锁的一部分，每个锁具有类似的目的。）参见Table 25.12
Timeout	服务器进程正在等待超时过期。wait_event将识别特定的等待点；参见Table 25.13

Table 25.5. Activity类型的等待事件

Activity 等待事件	描述
ArchiverMain	在归档进程的主循环中等待。
AutoVacuumMain	在自动清理启动过程的主循环中等待。
BgWriterHibernate	在后台写进程中等待，休眠状态。
BgWriterMain	在后台写进程主循环中等待。
CheckpointMain	在校验指针进程的主循环中等待。
LogicalApplyMain	在逻辑复制应用进程的主循环中等待。
LogicalLauncherMain	在逻辑复制启动器进程的主循环中等待。
PgStatMain	在统计收集器进程的主循环中等待。
RecoveryWalStream	流恢复期间，在启动进程主循环等待WAL到达。
SysLoggerMain	在syslogger进程的主循环中等待。
WalReceiverMain	在WAL接收器进程的主循环中等待。
WalSenderMain	在WAL发送者进程的主循环中等待。
WalWriterMain	在WAL写入进程的主循环中等待。

Table 25.6. BufferPin类型的等待事件

BufferPin 等待事件	描述
BufferPin	等待获得缓冲区上的独占锁。

Table 25.7. Client类型的等待事件

Client 等待事件	描述
ClientRead	等待从客户端读取数据。
ClientWrite	等待写入数据到客户端。
GSSOpenServer	在建立GSSAPI会话时等待从客户端读取数据。
LibPQWalReceiverConnect	在WAL接收器等待与远程服务器建立连接。
LibPQWalReceiverReceive	在WAL接收器中等待从远程服务器接收数据。
SSLOpenServer	在尝试连接时等待SSL。
WalReceiverWaitStart	等待启动进程发送用于流复制的初始数据。
WalSenderWaitForWAL	在WAL发送器进程中等待WAL被刷新。
WalSenderWriteData	在WAL发送器进程中处理WAL接收器的回复时，等待任何活动。

Table 25.8. Extension类型的等待事件

Extension 等待事件	描述
Extension	在扩展中等待。

Table 25.9. IO类型的等待事件

IO 等待事件	描述
BufFileRead	等待从缓冲文件中读取。
BufFileWrite	等待对缓冲文件的写入。
ControlFileRead	等待读取lt_control文件。
ControlFileSync	等待lt_control文件到达持久存储。
ControlFileSyncUpdate	等待更新lt_control文件以达到持久存储。
ControlFileWrite	等待写入lt_control文件。
ControlFileWriteUpdate	等待写入更新lt_control文件。
CopyFileRead	在文件复制操作期间等待读取。
CopyFileWrite	在文件拷贝操作期间等待写入。
DSMFillZeroWrite	等待用零填充动态共享内存备份(backing)文件。
DataFileExtend	等待关系数据文件被扩展。
DataFileFlush	等待关系数据文件达到持久存储。
DataFileImmediateSync	等待关系数据文件到持久存储的立即同步。
DataFilePrefetch	等待关系数据文件的异步预取。
DataFileRead	等待对关系数据文件的读取。
DataFileSync	等待对关系数据文件的更改达到持久存储。
DataFileTruncate	等待关系数据文件被截断。
DataFileWrite	等待对关系数据文件的写入。
LockFileAddToDataDirRead	在向数据目录锁文件中添加一行时等待读取。
LockFileAddToDataDirSync	等待数据到达持久存储，同时向数据目录锁文件添加一行。



I/O 等待事件	描述
LockFileAddToDataDirWrite	在向数据目录锁文件中添加一行时等待写操作。
LockFileCreateRead	创建数据目录锁文件时等待读取。
LockFileCreateSync	在创建数据目录锁文件时等待数据到达持久存储。
LockFileCreateWrite	在创建数据目录锁文件时等待写操作。
LockFileReCheckDataDirRead	在重新检查数据目录锁文件期间等待读取。
LogicalRewriteCheckpointSync	等待逻辑重写映射到检查点到达持久存储。
LogicalRewriteMappingSync	在逻辑重写期间等待映射数据到达持久存储。
LogicalRewriteMappingWrite	在逻辑重写期间等待映射数据的写入。
LogicalRewriteSync	等待逻辑重写映射到达持久存储。
LogicalRewriteTruncate	等待在逻辑重写期间截断映射数据。
LogicalRewriteWrite	等待逻辑重写映射的写入。
RelationMapRead	等待关系映射文件的读取。
RelationMapSync	等待关系映射文件到达持久存储。
RelationMapWrite	等待对关系映射文件的写入。
ReorderBufferRead	在重新排序缓冲区管理期间等待读取。
ReorderBufferWrite	在重新排序缓冲区管理期间等待写操作。
ReorderLogicalMappingRead	在重新排序缓冲区管理期间等待读取逻辑映射。
ReplicationSlotRead	等待从复制槽位控制文件读取。
ReplicationSlotRestoreSync	等待复制槽控制文件到达持久存储，同时将其恢复到内存中。
ReplicationSlotSync	等待复制槽控制文件到达持久存储。
ReplicationSlotWrite	等待对复制槽控制文件的写入。
SLRUFlushSync	在检查点或数据库关闭期间等待SLRU数据到达持久存储。
SLRURead	等待读取SLRU页面。
SLRUSync	在写页面后等待SLRU数据到达持久存储。
SLRUWrite	等待SLRU页面的写入。
SnapbuildRead	等待读取序列化的历史目录快照。
SnapbuildSync	等待序列化历史目录快照到达持久存储。
SnapbuildWrite	等待串行历史目录快照的写入。
TimelineHistoryFileSync	等待通过流复制接收的时间线历史文件到达持久存储。
TimelineHistoryFileWrite	等待通过流复制接收的时间线历史文件的写入。
TimelineHistoryRead	等待读取时间线历史文件。
TimelineHistorySync	等待新创建的时间线历史文件到达持久存储。
TimelineHistoryWrite	等待写入新创建的时间线历史文件。
TwophaseFileRead	等待读取两阶段状态文件。

IO 等待事件	描述
TwophaseFileSync	等待两阶段状态文件到达持久存储。
TwophaseFileWrite	等待对两阶段状态文件的写入。
WALBootstrapSync	在引导过程中等待WAL达到持久存储。
WALBootstrapWrite	在引导过程中等待WAL页面的写入。
WALCopyRead	通过复制一个已有WAL段来创建一个新的WAL段时等待读取。
WALCopySync	等待通过复制一个已有WAL段到持久存储来创建一个新的WAL段。
WALCopyWrite	通过复制一个已有WAL段来创建一个新的WAL段时等待写入。
WALInitSync	等待一个新初始化的WAL文件到持久存储。
WALInitWrite	在初始化一个新的WAL文件时等待写入。
WALRead	等待WAL文件的读取。
WALSenderTimelineHistoryRead	在walsender时间线命令期间等待从时间线历史文件读取。
WALSync	等待WAL文件到达持久存储。
WALSyncMethodAssign	等待数据到达持久存储，同时分配一个新的WAL同步方法。
WALWrite	等待写入WAL文件。

Table 25.10. IPC类型的等待事件

IPC 等待事件	描述
BackupWaitWalArchive	等待备份所需的WAL文件成功存档。
BgWorkerShutdown	等待后台工作者关闭。
BgWorkerStartup	等待后台工作者启动。
BtreePage	正等待继续并行B-树扫描所需的页号变得可用。
CheckpointDone	等待检查点完成。
CheckpointStart	等待检查点开始。
ExecuteGather	在执行Gather 计划节点时，等待子进程的活动。
HashBatchAllocate	等待一个选定的并行哈希参与者分配哈希表。
HashBatchElect	等待选择一个并行哈希参与者来分配哈希表。
HashBatchLoad	等待其他并行哈希参与者完成哈希表的加载。
HashBuildAllocate	等待一个选定的并行哈希参与者分配初始哈希表。
HashBuildElect	等待选择一个并行哈希参与者来分配初始哈希表。
HashBuildHashInner	等待其他并行哈希参与者完成内部关系的散列。
HashBuildHashOuter	等待其他Parallel 哈希参与者完成对外部关系的分区。

IPC 等待事件	描述
HashGrowBatchesAllocate	等待选定的并行哈希参与者分配更多批处理。
HashGrowBatchesDecide	等待选择一个并行哈希参与者来决定未来的批处理增长。
HashGrowBatchesElect	等待选择一个Parallel 哈希参与者来分配更多批处理。
HashGrowBatchesFinish	Waiting for an elected Parallel Hash participant to decide on future batch growth.
HashGrowBatchesRepartition	等待一个选定的并行哈希参与者决定未来的批处理增长。
HashGrowBucketsAllocate	等待选定的并行哈希参与者完成更多bucket 的分配。
HashGrowBucketsElect	等待选择一个并行哈希参与者来分配更多的buckets。
HashGrowBucketsReinsert	等待其他Parallel 哈希参与者完成将元组插入到新buckets中。
LogicalSyncData	等待逻辑复制远程服务器发送用于初始表同步的数据。
LogicalSyncStateChange	等待逻辑复制远程服务器更改状态。
MessageQueueInternal	等待另一个进程附加到共享消息队列。
MessageQueuePutMessage	等待将协议消息写入共享消息队列。
MessageQueueReceive	等待从共享消息队列接收字节。
MessageQueueSend	等待将字节发送到共享消息队列。
ParallelBitmapScan	等待并行位图扫描被初始化。
ParallelCreateIndexScan	等待并行CREATE INDEX 工作者完成堆扫描。
ParallelFinish	等待并行工作人员完成计算。
ProcArrayGroupUpdate	等待组领导在并行操作结束时清除事务ID。
ProcSignalBarrier	等待屏障事件被所有后端处理。
Promote	等待备用系统提升。
RecoveryConflictSnapshot	等待vacuum清理的恢复冲突解决。
RecoveryConflictTablespace	等待恢复冲突解决删除表空间。
RecoveryPause	等待恢复继续进行。
ReplicationOriginDrop	等待复制源变为非活动状态，以便可以删除它。
ReplicationSlotDrop	等待复制槽变为非活动状态，以便可以删除它。
SafeSnapshot	等待获取READ ONLY DEFERRABLE事务的有效快照。
SyncRep	在同步复制期间等待远程服务器的确认。
XactGroupUpdate	等待分组组长在并行操作结束时更新事务状态。

Table 25.11. Lock类型的等待事件

Lock 等待事件	描述
advisory	等待获得一个建议用户锁。
extend	等待扩展一个关系。
frozenid	等待升级 pg_database.datfrozenxid 和 pg_database.datminmxid。
object	等待获取非关系数据库对象上的锁。
page	等待获取一个关系页面上的锁。
relation	等待获得一个关系的锁。
spectoken	等待获取推测的插入锁。
transactionid	等待事务完成。
tuple	等待获取元组上的锁。
userlock	等待获取用户锁。
virtualxid	等待获取虚拟事务ID锁。

Table 25.12. LWLock类型的等待事件

LWLock 等待事件	描述
AddinShmemInit	等待管理共享内存中的扩展空间分配。
AutoFile	等待更新lightdb.auto.conf文件。
Autovacuum	等待读取或更新自动清理工作者的当前状态。
AutovacuumSchedule	等待确保选择为自动清理的表仍然需要清理。
BackgroundWorker	等待读取或更新后台工作者状态。
BtreeVacuum	等待读取或更新b-树索引的清理相关信息。
BufferContent	等待访问内存中的数据页。
BufferIO	等待数据页上的I/O。
BufferMapping	等待将数据块与缓冲池中的缓冲区关联。
Checkpoint	等待开始一个检查点。
CheckpointInterComm	等待管理fsync请求。
CommitTs	等待读取或更新事务提交时间戳的最后一个值集。
CommitTsBuffer	在提交时间戳SLRU缓冲区上等待I/O。
CommitTsSLRU	等待访问提交时间戳SLRU缓存。
ControlFile	等待读取或更新lt_control文件或创建一个新的WAL文件。
DynamicSharedMemoryControl	等待读取或更新动态共享内存分配信息。
LockFastPath	等待读取或更新进程的快速路径锁信息。
LockManager	等待读取或更新关于“heavyweight”锁。
LogicalRepWorker	等待读取或更新逻辑复制工作器的状态。
MultiXactGen	等待读取或更新共享的multixact状态。
MultiXactMemberBuffer	在multixact成员SLRU缓冲区上等待I/O。
MultiXactMemberSLRU	等待访问multixact成员SLRU缓存。

LWLock 等待事件	描述
MultiXactOffsetBuffer	在multixact 偏移 SLRU缓冲区上等待I/O。
MultiXactOffsetSLRU	等待访问multixact 偏移 SLRU缓存。
MultiXactTruncation	等待读取或截断multixact信息。
NotifyBuffer	在NOTIFY 消息 SLRU缓冲区上等待I/O。
NotifyQueue	等待读取或更新NOTIFY 消息。
NotifyQueueTail	等待NOTIFY消息存储上的更新限制。
NotifySLRU	等待访问NOTIFY消息SLRU缓存。
OldGen	等待分配一个新的OID。
OldSnapshotTimeMap	等待读取或更新旧的快照控制信息。
ParallelAppend	在并行附加计划执行期间等待选择下一个子计划。
ParallelHashJoin	在并行哈希连接计划执行期间等待同步工作器。
ParallelQueryDSA	等待并行查询动态共享内存分配。
PerSessionDSA	等待并行查询动态共享内存分配。
PerSessionRecordType	等待访问有关复合类型的并行查询信息。
PerSessionRecordTypmod	等待访问有关标识匿名记录类型的类型修饰符的并行查询信息。
PerXactPredicateList	在并行查询期间等待访问当前可序列化事务持有的谓词锁列表。
PredicateLockManager	等待访问可序列化事务使用的谓词锁信息。
ProcArray	等待访问每个进程共享的数据结构(通常情况, 是获取快照或报告会话的事务ID)。
RelationMapping	等待读取或更新pg_filenode.map文件(用于跟踪某些系统目录的文件节点分配)。
RelCacheInit	等待读取或更新lt_internal.init关系缓存初始化文件。
ReplicationOrigin	等待创建、删除或使用复制源。
ReplicationOriginState	等待读取或更新一个复制源的进度。
ReplicationSlotAllocation	等待分配或释放复制槽。
ReplicationSlotControl	等待读取或更新复制槽状态。
ReplicationSlotIO	在复制槽位上等待I/O。
SerialBuffer	在可串行事务冲突的SLRU缓冲区上等待I/O。
SerializableFinishedList	等待访问已完成的可序列化事务列表。
SerializablePredicateList	等待访问可序列化事务持有的谓词锁列表。
SerializableXactHash	等待读取或更新关于可序列化事务的信息。
SerialSLRU	等待访问可序列化事务冲突SLRU缓存。
SharedTidBitmap	在并行位图索引扫描期间等待访问共享的TID位图。
SharedTupleStore	在并行查询期间等待访问共享元组存储。
ShmemIndex	等待在共享内存中找到或分配空间。
SInvalRead	等待从共享目录失效队列中检索消息。
SInvalWrite	等待向共享编目失效队列添加消息。

LWLock 等待事件	描述
SubtransBuffer	在子事务SLRU缓冲区上等待I/O。
SubtransSLRU	等待访问子事务SLRU缓存。
SyncRep	等待读取或更新有关同步复制状态的信息。
SyncScan	等待选择同步表扫描的起始位置。
TablespaceCreate	等待创建或删除表空间。
TwoPhaseState	等待读取或更新已准备事务的状态。
WALBufMapping	等待在WAL缓冲区中替换一个页面。
WALInsert	等待将WAL数据插入内存缓冲区。
WALWrite	等待WAL缓冲区写入磁盘。
WrapLimitsVacuum	等待更新事务id和multixact消费的限制。
XactBuffer	在事务状态的SLRU缓冲区上等待I/O。
XactSLRU	等待访问事务状态的SLRU缓存。
XactTruncation	等待执行pg_xact_status或更新它可用的最早的事务ID。
XidGen	等待分配新的事务ID。
LTNameLock	等待获取用于表、包或schema的创建或重命名的锁信息

Note

扩展可以将LWLock类型添加到Table 25.13所示的列表中。在某些情况下，扩展名分配的名称不能在所有服务器进程中使用；因此LWLock等待事件可能只被报告为“extension”，而不是扩展名分配的名称。

Table 25.13. Timeout类型的等待事件

Timeout 等待事件	描述
BaseBackupThrottle	当有限流活动时基础备份期间等待。
PgSleep	由于调用pg_sleep或同类函数而等待。
RecoveryApplyDelay	由于延迟设置，在恢复期间等待应用WAL。
RecoveryRetrieveRetryInterval	当WAL数据无法从任何来源(lt_wal, 存档或流)获得时，在恢复期间等待。
RegisterSyncRequest	等待向检查点进程发送同步请求，因为请求队列已满。
VacuumDelay	在一个基于代价的清理延迟点。

下面的例子展示了如何查看等待事件：

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE wait_event
is NOT NULL;
 pid | wait_event_type | wait_event
-----+-----+-----
 2540 | Lock            | relation
 6644 | LWLock         | ProcArray
(2 rows)
```

## 25.2.4. pg\_stat\_replication

pg\_stat\_replication视图将在每个WAL发送方进程中包含一行，显示关于复制到发送方连接的备用服务器的统计信息。只有直接连接的备用设备被列出；没有关于下游备用服务器的信息。

Table 25.14. pg\_stat\_replication 视图

列类型	描述
pid integer	一个 WAL 发送进程的进程 ID
usesysid oid	登录到这个 WAL 发送进程的用户的 OID
username name	登录到这个 WAL 发送进程的用户的名称
application_name text	连接到这个 WAL 发送进程的应用的名称
client_addr inet	连接到这个 WAL 发送进程的客户端的 IP 地址。如果这个域为空，它表示该客户端通过服务器机器上的一个 Unix 套接字连接。
client_hostname text	连接上的客户端的主机名，由一次对 client_addr 的逆向 DNS 查找报告。这个域将只对 IP 连接非空，并且只有在 log_hostname 被启用时非空。
client_port integer	客户端用来与这个 WAL 发送进程通讯的 TCP 端口号，如果使用 Unix 套接字则为 -1
backend_start timestamp with time zone	这个进程开始的时间，即客户端是何时连接到这个 WAL 发送进程的。
backend_xmin xid	由 hot_standby_feedback 报告的这个后备机的 xmin 水平线。
state text	当前的 WAL 发送进程状态。可能的值是： <ul style="list-style-type: none"> <li>• startup: 这个 WAL 发送器正在启动。</li> <li>• catchup: 这个 WAL 发送者连接的备用服务器正在赶上主服务器。</li> <li>• streaming: 在其连接的备用服务器赶上主服务器之后，这个 WAL 发送方正在流化变化。</li> <li>• backup: 这个 WAL 发送器正在发送一个备份。</li> <li>• stopping: 这个 WAL 发送器正在停止。</li> </ul>
sent_lsn pg_lsn	在这个连接上发送的最后一个预写式日志的位置
write_lsn pg_lsn	被这个后备服务器写入到磁盘的最后一个预写式日志的位置
flush_lsn pg_lsn	被这个后备服务器刷入到磁盘的最后一个预写式日志的位置
replay_lsn pg_lsn	被重放到这个后备服务器上的数据库中的最后一个预写式日志的位置
write_lag interval	

列类型	描述
	从本地刷新近期的WAL与接收到此备用服务器已写入WAL的通知(但尚未刷新或应用它)之间的时间经过。 如果将此服务器配置为同步备用服务器, 则可以使用此参数来衡量在提交时synchronous_commit级别remote_write所导致的延迟。
flush_lag interval	在本地刷写近期的WAL与接收到后备服务器已经写入并且刷写它(但还没有应用)的通知之间流逝的时间。 如果这台服务器被配置为一个同步后备, 这可以用来计量在提交时synchronous_commit的级别on所导致的延迟。
replay_lag interval	在本地刷写近期的WAL与接收到后备服务器已经写入它、刷写它并且应用它的通知之间流逝的时间。 如果这台服务器被配置为一个同步后备, 这可以用来计量在提交时synchronous_commit的级别remote_apply所导致的延迟。
sync_priority integer	在基于优先的同步复制中, 这台后备服务器被选为同步后备的优先级。在基于规定数量的同步复制中, 这个值没有效果。
sync_state text	这一台后备服务器的同步状态。 可能的值是: <ul style="list-style-type: none"> <li>• async: 这台后备服务器是异步的。</li> <li>• potential: 这台后备服务器现在是异步的, 但可能在当前的同步后备失效时变成同步的。</li> <li>• sync: 这台后备服务器是同步的。</li> <li>• quorum: 这台后备服务器被当做规定数量后备服务器的候选。</li> </ul>
reply_time	带时区的时间戳 从备用服务器收到的最后一条回复信息的发送时间

pg\_stat\_replication视图中报告的滞后时间近期的WAL被写入、刷写并且重放以及发送器知道这一切所花的时间的度量。如果远程服务器被配置为一台同步后备, 这些时间表示由每一种同步提交级别所带来(或者是可能带来)的提交延迟。对于一台异步后备, replay\_lag列是最近的事务变得对查询可见的延迟时间的近似值。如果后备服务器已经完全追上了发送服务器并且没有WAL活动, 在短时间内将继续显示最近测到的滞后时间, 再然后就会显示为NULL。

对于物理复制会自动测量滞后时间。逻辑解码插件可能会选择性地发出跟踪消息, 如果它们没有这样做, 跟踪机制将把滞后显示为NULL。

Note

报告的滞后时间并非按照当前的重放速率该后备还有多久才能追上发送服务器的预测。在新的WAL被生成期间, 这样一种系统将显示类似的时间, 但是当发送器变为闲置时会显示不同的值。特别是当后备服务器完全追上时, pg\_stat\_replication显示的是写入、刷写及重放最近报告的WAL位置所花的时间而不是一些用户可能预期的零。这种做法与为近期的写事务测量同步提交和事务可见性延迟的目的是一致的。为了降低用户预期一种不同的滞后模型带来的混淆, 在一个完全重放完的闲置系统上, lag列会在一段比较短的时间后回复成NULL。监控系统应该选择将这种情况表示为缺失数据、零或者继续显示最近的已知值。

## 25.2.5. pg\_stat\_wal\_receiver



pg\_stat\_wal\_receiver事务只包含一行，它显示了从 WAL 接收器所连接的服务器得到的有关该接收器的统计信息。

Table 25.15. pg\_stat\_wal\_receiver 视图

列类型	描述
pid integer	WAL接收器进程的进程ID
status text	WAL接收进程的活动状态
receive_start_lsn pg_lsn	WAL接收器启动时使用的第一个写前日志位置
receive_start_tli integer	WAL接收器启动时使用的第一个时间线数字
written_lsn pg_lsn	已经接收并写入磁盘的最后一个预写式日志位置，但没有刷入。这不能用于数据完整性检查。
flushed_lsn pg_lsn	已经接收并刷入到磁盘的最后一个预写式日志位置，该字段的初始值是启动WAL接收器时使用的第一个日志位置
received_tli integer	接收并刷入到磁盘的最后一个预写式日志位置的时间线数字，该字段的初始值为启动WAL接收器时使用的第一个日志位置的时间线数字
last_msg_send_time timestamp with time zone	从源头WAL发送器收到的最后一条信息的发送时间
last_msg_receipt_time timestamp with time zone	从源头WAL发送器收到的最后一条信息的接收时间
latest_end_lsn pg_lsn	向源头WAL发送器报告的最后的预写式日志位置
latest_end_time timestamp with time zone	向源头WAL发送方报告的最后一次写前日志位置的时间
slot_name text	这个WAL接收器使用的复制槽的名称
sender_host text	这个WAL接收器连接到的LightDB实例的主机。这可以是主机名、IP地址，或者目录路径，如果连接是通过Unix套接字进行的。（路径的情况可以区分，因为它总是以/开头的绝对路径。）
sender_port integer	这个WAL接收器连接的LightDB实例的端口号。
conninfo text	这个WAL接收器使用的连接字符串，对安全敏感的字段进行了模糊处理。

## 25.2.6. pg\_stat\_subscription

每一个订阅的主工作者都在pg\_stat\_subscription视图中有一行（如果工作者没有运行则PID为空），处理被订阅表的初始数据拷贝操作的工作者还会有额外的行。

Table 25.16. pg\_stat\_subscription View

列类型	描述
subid oid	订阅的OID
subname name	订阅的名称
pid integer	订阅工作者进程的进程ID
relid oid	工作器正在同步的关系的OID;Null用于主应用工作器
received_lsn pg_lsn	接收到的最后一个预写式日志位置, 该字段的初始值为0
last_msg_send_time timestamp with time zone	从WAL发送器收到的最后一条信息的发送时间
last_msg_receipt_time timestamp with time zone	从WAL发送器收到的最后一条信息的接收时间
latest_end_lsn pg_lsn	向WAL发送器报告的最后预写式日志位置
latest_end_time timestamp with time zone	向WAL发送器报告的最后一次预写式日志位置的时间

## 25.2.7. pg\_stat\_ssl

pg\_stat\_ssl视图将为每一个后端或者 WAL 发送进程包含一行, 用来显示这个连接上的 SSL 使用情况。 可以把它与pg\_stat\_activity或者pg\_stat\_replication通过pid列连接来得到更多有关该连接的细节。

Table 25.17. pg\_stat\_ssl 视图

列类型	描述
pid integer	后端或WAL发送器进程ID
ssl boolean	如果在此连接上使用SSL, 则为真
version text	使用SSL的版本, 如果此连接上没有使用SSL则为NULL
cipher text	正在使用的SSL密码的名称, 如果此连接上没有使用SSL则为NULL
bits integer	使用的加密算法中的位数, 如果此连接上没有使用SSL则为NULL
compression boolean	如果使用SSL压缩则为真, 否则为假, 如果此连接未使用SSL则为NULL
client_dn text	区别名称(DN, Distinguished Name)字段与使用的客户端证书, 如果没有提供客户端证书或在此连接上没有使用SSL, 则为NULL。 如果DN字段长于NAMEDATALEN(标准构建中为64个字符), 则该字段将被截断。
client_serial numeric	

列类型	描述
	客户端证书的序列号，如果没有提供客户端证书或在此连接上没有使用SSL，则为NULL。证书序列号和证书颁发者的组合唯一标识一个证书(除非颁发者错误地重用序列号)。
issuer_dn text	客户端证书颁发者的区别名称(DN, Distinguished Name)，如果没有提供客户端证书或在此连接上没有使用SSL，则为NULL。该字段像client_dn一样被截断。

## 25.2.8. pg\_stat\_gssapi

pg\_stat\_gssapi视图将包含每一个后端一个行，显示该连接上的GSSAPI使用情况。它可以加入到pg\_stat\_activity或pg\_stat\_replication上的pid列，获取更多关于连接的详细信息。

Table 25.18. pg\_stat\_gssapi 视图

列类型	描述
pid integer	后端进程ID
gss_authenticated boolean	如果此连接使用了GSSAPI身份验证，则为True
principal text	用于验证此连接的主体，如果未使用GSSAPI对此连接进行身份验证，则为NULL。如果主体长度超过NAMEDATALEN(标准构建中为64个字符)，则该字段被截断。
encrypted boolean	如果在此连接上使用了GSSAPI加密，则为真

## 25.2.9. pg\_stat\_archiver

pg\_stat\_archiver视图总是有一行，其中包含关于集群的存档进程的数据。

Table 25.19. pg\_stat\_archiver 视图

列类型	描述
archived_count bigint	已成功存档的WAL文件数
last_archived_wal text	最后一个成功存档的WAL文件的名称
last_archived_time timestamp with time zone	最后一次成功存档操作的时间
failed_count bigint	记录WAL文件归档失败次数
last_failed_wal text	最后一次失败的存档操作的WAL文件的名称
last_failed_time timestamp with time zone	上次存档操作失败的时间
stats_reset timestamp with time zone	这些统计数据最后一次重置的时间

## 25.2.10. pg\_stat\_bgwriter

pg\_stat\_bgwriter视图始终只有一行，其中包含集群的全局数据。

Table 25.20. pg\_stat\_bgwriter View

列类型	描述
checkpoints_timed bigint	已执行的预定检查点数
checkpoints_req bigint	请求已执行的检查点数
checkpoint_write_time double precision	检查点处理中将文件写入磁盘的部分所花费的总时间，以毫秒为单位
checkpoint_sync_time double precision	检查点处理中将文件同步到磁盘的部分所花费的总时间，以毫秒为单位
buffers_checkpoint bigint	检查点期间写入的缓冲区数
buffers_clean bigint	后台写入器写入的缓冲区数
maxwritten_clean bigint	后台写入器因为写入太多缓冲区而停止清理扫描的次数
buffers_backend bigint	后端直接写入的缓冲区数
buffers_backend_fsync bigint	后端必须执行自己的fsync调用的次数(通常后台写入器处理这些，即使后端执行自己的写入)
buffers_alloc bigint	分配的缓冲区数
stats_reset timestamp with time zone	这些统计数据最后一次重置的时间

## 25.2.11. pg\_stat\_database

pg\_stat\_database视图将包含一行用于集群中的每个数据库，加一行用于共享对象，显示数据库范围的统计信息。

Table 25.21. pg\_stat\_database 视图

列类型	描述
datid oid	该数据库的OID，属于共享关系的对象为0
datname name	这个数据库的名称，或者共享对象为NULL。
numbackends integer	当前连接到此数据库的后端数，对于共享对象则为NULL。这是该视图中唯一返回反映当前状态的值的列；所有其他列返回自上次重置以来累积的值。
xact_commit bigint	此数据库中已提交的事务数
xact_rollback bigint	该数据库中已回滚的事务数
blks_read bigint	在该数据库中读取的磁盘块数

列类型	描述
blks_hit bigint	在缓存中发现磁盘块的次数，因此读取不是必需的(这只包括在LightDB缓存中，而不是在操作系统的文件系统缓存中)
tup_returned bigint	这个数据库中查询返回的行数
tup_fetched bigint	这个数据库中查询获取的行数
tup_inserted bigint	查询在该数据库中插入的行数
tup_updated bigint	这个数据库中查询更新的行数
tup_deleted bigint	这个数据库中查询删除的行数
conflicts bigint	由于与此数据库中的恢复冲突而取消的查询数。(冲突只发生在备用服务器上;详情请参见 pg_stat_database_conflicts。)
temp_files bigint	这个数据库中查询创建的临时文件的数量。所有临时文件都将被计数，而不顾及临时文件为什么被创建(例如，排序或散列)，也不考虑log_temp_files设置。
temp_bytes bigint	这个数据库中的查询写入临时文件的数据总量。所有临时文件都将被计数，而不考虑临时文件为什么被创建，也不考虑log_temp_files设置。
deadlocks bigint	在此数据库中检测到的死锁数
checksum_failures bigint	在此数据库(或共享对象)中检测到的数据页校验码失败数，如果没有启用数据校验码则为NULL。
checksum_last_failure timestamp with time zone	在此数据库(或共享对象)中检测到最后一个数据页校验码失败的时间，如果没有启用数据校验码则为NULL。
blk_read_time double precision	在这个数据库中通过后端读取数据文件块所花费的时间，以毫秒为单位(如果启用了track_io_timing，否则为零)
blk_write_time double precision	在这个数据库中通过后端写数据文件块所花费的时间，以毫秒为单位(如果启用了track_io_timing，否则为零)
stats_reset timestamp with time zone	这些统计数据最后一次重置的时间

## 25.2.12. pg\_stat\_database\_conflicts

pg\_stat\_database\_conflicts视图为每一个数据库包含一行，与后备服务器上的恢复过程冲突而被取消的查询的统计信息。器上的信息，因为冲突不会发生在主服务器上。

用来显示数据库范围内由于这个视图将只包含后备服务器

Table 25.22. pg\_stat\_database\_conflicts 视图

列类型	描述
datid oid	

列类型	描述
	数据库的OID
datname name	数据库的名称
confl_tablespace bigint	这个数据库中由于删除表空间而取消的查询的数量
confl_lock bigint	此数据库中由于锁定超时而取消的查询数
confl_snapshot bigint	此数据库中由于旧快照而取消的查询数
confl_bufferpin bigint	此数据库中由于固定缓冲区而取消的查询数
confl_deadlock bigint	此数据库中由于死锁而取消的查询数

## 25.2.13. pg\_stat\_all\_tables

pg\_stat\_all\_tables视图将为当前数据库中的每一个表（包括 TOAST 表）包含一行，该行显示与对该表的访问相关的统计信息。 pg\_stat\_user\_tables和pg\_stat\_sys\_tables视图包含相同的信息，但是被过滤得分别只显示用户和系统表。

Table 25.23. pg\_stat\_all\_tables 视图

列类型	描述
relid oid	表的OID
schemaname name	该表所在的模式的名称
relname name	这个表的名称
seq_scan bigint	在此表上启动的顺序扫描数
seq_tup_read bigint	连续扫描获取的实时行数
idx_scan bigint	对这个表发起的索引扫描数
idx_tup_fetch bigint	索引扫描获取的实时行数
n_tup_ins bigint	插入的行数
n_tup_upd bigint	更新的行数(包括HOT更新的行)
n_tup_del bigint	删除的行数
n_tup_hot_upd bigint	HOT更新的行数(即，不需要单独的索引更新)
n_live_tup bigint	活的行的估计数量

列类型	描述
n_dead_tup bigint	僵死行的估计数量
n_mod_since_analyze bigint	自上次分析此表以来修改的行的估计数量
n_ins_since_vacuum bigint	自上次清空此表以来插入的行的估计数量
last_vacuum timestamp with time zone	最后一次手动清理这个表(不包括VACUUM FULL)
last_autovacuum timestamp with time zone	这个表最后一次被自动清理守护进程清理的时间
last_analyze timestamp with time zone	上一次手动分析这个表
last_autoanalyze timestamp with time zone	自动清理守护进程最后一次分析这个表
vacuum_count bigint	这个表被手动清理的次数(VACUUM FULL不计数)
autovacuum_count bigint	这个表被autovacuum守护进程清理的次数
analyze_count bigint	手动分析这个表的次数
autoanalyze_count bigint	这个表被autovacuum守护进程分析的次数

## 25.2.14. pg\_stat\_all\_indexes

pg\_stat\_all\_indexes视图将为当前数据库中的每个索引包含一行，该行显示关于对该索引访问的统计信息。pg\_stat\_user\_indexes和pg\_stat\_sys\_indexes视图包含相同的信息，但是被过滤得只分别显示用户和系统索引。

Table 25.24. pg\_stat\_all\_indexes 视图

列类型	描述
relid oid	对于此索引的表的OID
indexrelid oid	这个索引的OID
schemaname name	这个索引所在的模式名称
relname name	这个索引的表的名称
indexrelname name	这个索引的名称
idx_scan bigint	在这个索引上开启的索引扫描的数量
idx_tup_read bigint	扫描此索引返回的索引项数
idx_tup_fetch bigint	

列类型	描述
	使用此索引进行简单索引扫描获取的活动表行数

索引可以被简单索引扫描、“位图”索引扫描以及优化器使用。在一次位图扫描中，多个索引的输出可以被通过 AND 或 OR 规则组合，因此当使用一次位图扫描时难以将取得的个体堆行与特定的索引关联起来。因此，一次位图扫描会增加它使用的索引的pg\_stat\_all\_indexes.idx\_tup\_read计数，并且为每个表增加pg\_stat\_all\_tables.idx\_tup\_fetch计数，但是它不影响pg\_stat\_all\_indexes.idx\_tup\_fetch。如果所提供的常量值不在优化器统计信息记录的范围之内，优化器也会访问索引来检查，因为优化器统计信息可能已经“不新鲜”了。

Note

即使不用位图扫描，idx\_tup\_read和idx\_tup\_fetch计数也可能不同，因为idx\_tup\_read统计从该索引取得的索引项而idx\_tup\_fetch统计从表取得的活着的行。如果使用该索引取得了任何死亡行或还未提交的行，或者如果通过一次只用索引扫描的方式避免了任何堆获取，后者将较小。

## 25.2.15. pg\_statio\_all\_tables

pg\_statio\_all\_tables视图将为当前数据库中的每个表（包括 TOAST 表）包含一行，该行显示指定表上有关 I/O 的统计信息。pg\_statio\_user\_tables和pg\_statio\_sys\_tables视图包含相同的信息，但是被过滤得分别只显示用户表和系统表。

Table 25.25. pg\_statio\_all\_tables 视图

列类型	描述
relid oid	表的OID
schemaname name	该表所在的模式名
relname name	这个表的名称
heap_blks_read bigint	从该表中读取的磁盘块的数量
heap_blks_hit bigint	该表中的缓冲区命中数
idx_blks_read bigint	从这个表上所有索引读取的磁盘块数
idx_blks_hit bigint	这个表上所有索引中的缓冲区命中数
toast_blks_read bigint	从这个表的TOAST表中读取的磁盘块的数量(如果有的话)
toast_blks_hit bigint	这个表的TOAST表中的缓冲区命中数(如果有的话)
tidx_blks_read bigint	从这个表的TOAST表索引中读取的磁盘块的数量(如果有的话)
tidx_blks_hit bigint	



列类型	描述
	这个表的TOAST表索引中的缓冲区命中数(如果有的话)

## 25.2.16. pg\_statio\_all\_indexes

pg\_statio\_all\_indexes视图将为当前数据库中的每个索引包含一行，该行显示指定索引上有关 I/O 的统计信息。pg\_statio\_user\_indexes和pg\_statio\_sys\_indexes视图包含相同的信息，但是被过滤得分别只显示用户索引和系统索引。

Table 25.26. pg\_statio\_all\_indexes 视图

列类型	描述
relid oid	对这个索引的表的OID
indexrelid oid	这个索引的OID
schemaname name	索引所在的模式名称
relname name	此索引的表的名称
indexrelname name	这个索引的名称
idx_blks_read bigint	从此索引中读取的磁盘块的数量
idx_blks_hit bigint	此索引中的缓冲区命中数

## 25.2.17. pg\_statio\_all\_sequences

pg\_statio\_all\_sequences视图将为当前数据库中的每个序列包含一行，该行显示在指定序列上有关 I/O 的统计信息。

Table 25.27. pg\_statio\_all\_sequences 视图

列类型	描述
relid oid	序列的OID
schemaname name	此序列所在的模式的名称
relname name	此序列的名称
blks_read bigint	从这个序列中读取的磁盘块的数量
blks_hit bigint	在此序列中的缓冲区命中数

## 25.2.18. pg\_stat\_user\_functions

pg\_stat\_user\_functions视图将为每一个被追踪的函数包含一行，该行显示有关该函数执行的统计信息。track\_functions参数控制到底哪些函数被跟踪。

Table 25.28. pg\_stat\_user\_functions 视图

列类型	描述
funcid oid	函数的OID
schemaname name	这个函数所在的模式的名称
funcname name	这个函数的名称
calls bigint	这个函数已经被调用的次数
total_time double precision	在这个函数以及它所调用的其他函数中花费的总时间，以毫秒计
self_time double precision	在这个函数本身花费的总时间，不包括被它调用的其他函数，以毫秒计

## 25.2.19. pg\_stat\_slru

LightDB通过SLRU(simple least-recently-used, 简单的最近-最少-使用)缓存访问某些磁盘上的信息。 pg\_stat\_slru视图将为每个被跟踪的SLRU缓存包含一行，显示关于访问缓存页面的统计信息。

Table 25.29. pg\_stat\_slru 视图

列类型	描述
name text	SLRU的名称
blks_zeroed bigint	初始化期间被置零的块数
blks_hit bigint	已经在SLRU中的磁盘块被发现的次数，因此不需要读取(这只包括SLRU中的命中，而不是操作系统的文件系统缓存)
blks_read bigint	为这个SLRU读取的磁盘块数
blks_written bigint	为这个SLRU写入的磁盘块数
blks_exists bigint	为这个SLRU检查是否存在的块数
flushes bigint	此SLRU的脏数据刷新数
truncates bigint	这个SLRU的截断数
stats_reset timestamp with time zone	这些统计数据最后一次重置的时间

## 25.2.20. Statistics Functions

其他查看统计信息的方法是直接使用查询，这些查询使用上述标准视图用到的底层统计信息访问函数。 如要了解如函数名等细节，可参考标准视图的定义（例如，在ltsql中你可以发出\d+ pg\_stat\_activity）。 针对每一个数据库统计信息的访问函数把一个数据库 OID 作

为参数来标识要报告哪个数据库。而针对每个表和每个索引的函数要求表或索引 OID。针对每个函数统计信息的函数用一个函数 OID。注意只有在当前数据库中的表、索引和函数才能被这些函数看到。

更多统计集合的函数列在 Table 25.30中。

Table 25.30. Additional Statistics Functions

函数	描述
<code>pg_backend_pid () → integer</code>	返回附加到当前会话的服务器进程的进程ID。
<code>pg_stat_get_activity ( integer ) → setof record</code>	使用指定的进程ID返回有关后端信息的记录，如果指定了NULL，则返回系统中每个活动后端的一条记录。返回的字段是 <code>pg_stat_activity</code> 视图中字段的子集。
<code>pg_stat_get_snapshot_timestamp () → timestamp with time zone</code>	返回当前统计快照的时间戳。
<code>pg_stat_clear_snapshot () → void</code>	丢弃当前的统计快照。
<code>pg_stat_reset () → void</code>	将当前数据库的所有统计计数器重置为零。 默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。
<code>pg_stat_reset_shared ( text ) → void</code>	根据参数的不同，将一些集群范围的统计计数器重置为零。参数可以是 <code>bgwriter</code> 来重置 <code>pg_stat_bgwriter</code> 视图中显示的所有计数器，或者 <code>archiver</code> 来重置 <code>pg_stat_archiver</code> 视图中显示的所有计数器。 默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。
<code>pg_stat_reset_single_table_counters ( oid ) → void</code>	将当前数据库中单个表或索引的统计信息重置为零。 默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。
<code>pg_stat_reset_single_function_counters ( oid ) → void</code>	将当前数据库中单个函数的统计信息重置为零。 默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。
<code>pg_stat_reset_slru ( text ) → void</code>	将单个SLRU缓存或集群中所有SLRU的统计信息重置为零。如果该参数为NULL，则所有SLRU缓存的 <code>pg_stat_slru</code> 视图中显示的计数器将被重置。 参数可以是 <code>CommitTs</code> 、 <code>MultiXactMember</code> 、 <code>MultiXactOffset</code> 、 <code>Notify</code> 、 <code>Serial</code> 、 <code>Subtrans</code> 、或 <code>Xact</code> 中的一个，以便只重置该条目的计数器。如果参数是 <code>other</code> (或实际上，任何无法识别的名称)，那么所有其他SLRU缓存的计数器，如扩展定义的缓存，将被重置。 默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。

`pg_stat_get_activity`是`pg_stat_activity`视图的底层函数，它返回一个行集合，其中包含有关每个后端进程所有可用的信息。有时只获得该信息的一个子集可能会更方便。在那些情况中，可以使用一组更老的针对每个后端的统计访问函数，这些显示在Table 25.3中。这些访问函数使用一个后端 ID 号，范围从 1 到当前活动后端数目。函数`pg_stat_get_backend_idset`提供了一种方便的方法为每个活动后端产生一行来调用这些函数。例如，要显示PID以及所有后端当前的查询：

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
```

```
pg_stat_get_backend_activity(s.backendid) AS query
FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Table 25.31. Per-Backend Statistics Functions

函数	描述
<code>pg_stat_get_backend_idset ()</code>	<code>→ setof integer</code> 返回当前活动后端ID号的集合(从1到活动后端数)。
<code>pg_stat_get_backend_activity ( integer )</code>	<code>→ text</code> 返回此后端最近查询的文本。
<code>pg_stat_get_backend_activity_start ( integer )</code>	<code>→ timestamp with time zone</code> 返回后端最近一次查询开始的时间。
<code>pg_stat_get_backend_client_addr ( integer )</code>	<code>→ inet</code> 返回连接到此后端的客户端的IP地址。
<code>pg_stat_get_backend_client_port ( integer )</code>	<code>→ integer</code> 返回客户端用于通信的TCP端口号。
<code>pg_stat_get_backend_dbid ( integer )</code>	<code>→ oid</code> 返回此后端连接的数据库的OID。
<code>pg_stat_get_backend_pid ( integer )</code>	<code>→ integer</code> 返回此后端进程ID。
<code>pg_stat_get_backend_start ( integer )</code>	<code>→ timestamp with time zone</code> 返回该进程开始的时间。
<code>pg_stat_get_backend_userid ( integer )</code>	<code>→ oid</code> 返回登录到此后端的用户的OID。
<code>pg_stat_get_backend_wait_event_type ( integer )</code>	<code>→ text</code> 如果后端当前正在等待, 返回等待事件类型名称, 否则返回NULL。 详情请参见Table 25.4
<code>pg_stat_get_backend_wait_event ( integer )</code>	<code>→ text</code> 如果后端当前正在等待, 则返回等待事件名称, 否则为NULL。 参见 Table 25.5 到 Table 25.13
<code>pg_stat_get_backend_xact_start ( integer )</code>	<code>→ timestamp with time zone</code> 返回后端当前事务开始的时间。

## 25.3. 查看锁

监控数据库活动的另外一个有用的工具是`pg_locks`系统表。这样就允许数据库管理员查看在锁管理器里面未解决的锁的信息。例如, 这个功能可以被用于:

- 查看当前所有未解决的锁、在一个特定数据库中的关系上所有的锁、在一个特定关系上所有的锁, 或者由一个特定LightDB会话持有的所有的锁。
- 判断当前数据库中带有最多未授予锁的关系(它很可能是数据库客户端的竞争源)。
- 判断锁竞争给数据库总体性能带来的影响, 以及锁竞争随着整个数据库流量的变化范围。

`pg_locks`视图的细节在Section 48.8中。更多有关LightDB的锁和管理并发性的信息, 请参考Chapter 13

## 25.4. Progress Reporting

LightDB具有在命令执行过程中报告某些命令进度的能力。目前, 支持进度报告的命令只有ANALYZE, CLUSTER, CREATE INDEX, VACUUM, 和 BASE\_BACKUP(例如 `lt_basebackup` 发出的进行基础备份的复制命令。)。未来可能还会扩展。

## 25.4.1. ANALYZE Progress Reporting

每当ANALYZE运行时，pg\_stat\_progress\_analyze视图将包含当前运行该命令的每个后端的一行。下面的表描述了将要报告的信息，并提供了关于如何解释它们的信息。

Table 25.32. pg\_stat\_progress\_analyze 视图

列类型	描述
pid integer	后端的进程ID。
datid oid	后端连接到的数据库的OID。
datname name	后端连接到的数据库的名称。
relid oid	被分析的表的OID。
phase text	当前处理阶段。参见 Table 25.33
sample_blks_total bigint	将被采样的堆块的总数。
sample_blks_scanned bigint	扫描的堆块数量。
ext_stats_total bigint	扩展统计信息的数量。
ext_stats_computed bigint	已经计算的扩展统计的数量。此计数器仅在 computing extended statistics阶段增进。
child_tables_total bigint	子表的数量。
child_tables_done bigint	扫描的子表数。此计数器只有在acquiring inherited sample rows阶段才会增进。
current_child_table_relid oid	当前正在扫描的子表的OID。此字段仅在acquiring inherited sample rows时有效。

Table 25.33. ANALYZE Phases

阶段	描述
initializing	命令正在准备开始扫描堆。这个阶段预计会非常短暂。
acquiring sample rows	该命令当前正在扫描relid给出的表以获得示例行。
acquiring inherited sample rows	该命令当前正在扫描子表以获得示例行。列child_tables_total, child_tables_done, 和current_child_table_relid包含此阶段的进度信息。
computing statistics	该命令从表扫描期间获得的样例行计算统计信息。
computing extended statistics	该命令从表扫描期间获得的样例行计算扩展统计信息。
finalizing analyze	该命令在更新pg_class。当此阶段完成时，ANALYZE 将结束。

Note

注意当在分区表上运行ANALYZE时，它的所有分区也会被递归分析，如在ANALYZE中曾提到过。在这种情况下，首先报告父表的ANALYZE进度，收集它的继承统计信息，然后是每个分区的(继承统计信息)。

## 25.4.2. CREATE INDEX Progress Reporting

每当运行CREATE INDEX或REINDEX时，pg\_stat\_progress\_create\_index视图将包含当前正在创建索引的每个后端的一行。下面的表描述了将要报告的信息，并提供了关于如何解释它的信息。

Table 25.34. pg\_stat\_progress\_create\_index 视图

列类型	描述
pid integer	后端的进程ID。
datid oid	后端连接到的数据库的OID。
datname name	后端连接到的数据库的名称。
relid oid	正在创建索引的表的OID。
index_relid oid	正在创建或重建索引的OID。在非并发 CREATE INDEX的时候，此为 0。
command text	在运行的命令：CREATE INDEX, CREATE INDEX CONCURRENTLY, REINDEX, 或 REINDEX CONCURRENTLY.
phase text	索引创建的当前处理阶段。参见 Table 25.35
lockers_total bigint	在适用的情况下，需要等待的储物柜总数
lockers_done bigint	已经等待的储物柜数量。
current_locker_pid bigint	目前正在等待的储物柜的进程ID。
blocks_total bigint	本阶段要处理的区块总数。
blocks_done bigint	当前阶段已经处理的区块数量。
tuples_total bigint	当前阶段要处理的元组总数。
tuples_done bigint	在当前阶段已经处理的元组数量。
partitions_total bigint	在分区表上创建索引时，该列被设置为要在其上创建索引的分区总数。
partitions_done bigint	当在分区表上创建索引时，该列被设置为在其上完成索引的分区数。

Table 25.35. CREATE INDEX 的阶段

阶段	描述
初始化	CREATE INDEX或REINDEX正在准备创建索引。这个阶段预计会非常短暂。
构建前等待读写器	CREATE INDEX CONCURRENTLY或REINDEX CONCURRENTLY正在等待有可能看到表的写锁的事务完成。当不在并发模式时，这个阶段会被跳过。lockers_total、lockers_done 和 current_locker_pid 列包含了这个阶段的进度信息。
新建索引	索引是由访问方法专用代码建立的。在这一阶段，支持进度报告的访问方法填写自己的进度数据，子阶段在这一栏中表示。通常情况下，blocks_total和blocks_done将包含进度数据，也可能包含tuples_total和tuples_done。
在验证前等待读写器	CREATE INDEX CONCURRENTLY或REINDEX CONCURRENTLY正在等待有可能写入表的事务完成写锁的事务。当不在并发模式时，这个阶段会被跳过。lockers_total、lockers_done 和 current_locker_pid 列包含了这个阶段的进度信息。
索引验证：扫描索引	CREATE INDEX CONCURRENTLY正在扫描索引，搜索需要验证的图元组。如果不是在并发模式下，这个阶段会被跳过。列 blocks_total（设置为索引的总大小）和 blocks_done包含了这个阶段的进度信息。
指数验证：排序元组	CREATE INDEX CONCURRENTLY正在对索引扫描阶段的输出进行排序。
索引验证：扫描表	CREATE INDEX CONCURRENTLY正在扫描表，以验证前两个阶段收集的索引图元。当不在并发模式时，这个阶段被跳过。blocks_total列（设置为表的总大小）和blocks_done列包含这个阶段的进度信息。
等待旧照	CREATE INDEX CONCURRENTLY或REINDEX CONCURRENTLY正在等待可能看到表的事务释放快照。当不处于并发模式时，这个阶段会被跳过。lockers_total、lockers_done 和 current_locker_pid 列包含了这个阶段的进度信息。
标记 dead之前等待readers	REINDEX CONCURRENTLY等待表上有读锁的事务完成后，再将旧索引标记为死索引。当不在并发模式时，这个阶段被跳过。lockers_total、lockers_done 和 current_locker_pid 列包含了这个阶段的进度信息。
在 dropping之前等待 readers	REINDEX CONCURRENTLY等待表上有读锁的事务完成后，再丢弃旧索引。当不在并发模式时，这个阶段被跳过。列 lockers_total、lockers_done 和 current_locker_pid包含了这个阶段的进度信息。

### 25.4.3. VACUUM进度报告

只要VACUUM正在运行，每一个当前正在清理的后端（包括autovacuum工作者进程）在pg\_stat\_progress\_vacuum视图中都会有一行。下面的表描述了将被报告的信息并且提供了如何解释它们的信息。VACUUM FULL命令的进度是通过pg\_stat\_progress\_cluster报告的，因为VACUUM FULL和CLUSTER都是重写表，而普通的VACUUM只是原地修改表。见Section 25.4.4

Table 25.36. pg\_stat\_progress\_vacuum 视图

列类型	描述
pid integer	

列类型	描述
	后端的进程ID。
datid oid	这个后端连接的数据库的OID。
datname name	这个后端连接的数据库的名称。
relid oid	被vacuum的表的OID。
phase text	vacuum的当前处理阶段。参见 Table 25.37
heap_blks_total bigint	该表中堆块的总数。这个数字在扫描开始时报告，之后增加的块将不会（并且不需要）被这个VACUUM访问。
heap_blks_scanned bigint	被扫描的堆块数量。由于visibility map被用来优化扫描，一些块将被跳过而不做检查，被跳过的块会被包括在这个总数中，因此当清理完成时这个数字最终将会等于heap_blks_total。仅当处于扫描堆阶段时这个计数器才会前进。
heap_blks_vacuumed bigint	被清理的堆块数量。除非表没有索引，这个计数器仅在处于清理堆阶段时才会前进。不包含死亡元组的块会被跳过，因此这个计数器可能有时会向前跳跃一个比较大的增量。
index_vacuum_count bigint	已完成的索引清理周期数。
max_dead_tuples bigint	在需要执行一个索引清理周期之前我们可以存储的死亡元组数，取决于maintenance_work_mem。
num_dead_tuples bigint	从上一个索引清理周期以来收集的死亡元组数。

Table 25.37. VACUUM的阶段

阶段	描述
初始化	VACUUM正在准备开始扫描堆。这个阶段应该很简短。
扫描堆	VACUUM正在扫描堆。如果需要，它将会对每个页面进行修建以及碎片整理，并且可能会执行冻结动作。heap_blks_scanned列可以用来监控扫描的进度。
清理索引	VACUUM当前正在清理索引。如果一个表拥有索引，那么每次清理时这个阶段会在堆扫描完成后至少发生一次。如果maintenance_work_mem不足以存放找到的死亡元组，则每次清理时会多次清理索引。
清理堆	VACUUM当前正在清理堆。清理堆与扫描堆不是同一个概念，清理堆发生在每一次清理索引的实例之后。如果heap_blks_scanned小于heap_blks_total，系统将在这个阶段完成之后回去扫描堆；否则，系统将在这个阶段完成后开始清理索引。
清除索引	VACUUM当前正在清除索引。这个阶段发生在堆被完全扫描并且对堆和索引的所有清理都已经完成以后。
截断堆	VACUUM正在截断堆，以便把关系尾部的空页面返还给操作系统。这个阶段发生在清除完索引之后。



阶段	描述
执行最后的清除	VACUUM在执行最终的清除。在这个阶段中，VACUUM将清理空闲空间映射、更新pg_class中的统计信息并且将统计信息报告给统计收集器。当这个阶段完成时，VACUUM也就结束了。

## 25.4.4. CLUSTER进度报告

每当CLUSTER或VACUUM FULL运行时，pg\_stat\_progress\_cluster视图将包含当前正在运行的每一个后台的记录。下面的表格描述了将被报告的信息，并提供了关于如何解释这些信息的信息。

Table 25.38. pg\_stat\_progress\_cluster 视图

列类型	描述
pid integer	后台的进程ID。
datid oid	该后端连接的数据库的OID。
datname name	与此后端连接的数据库的名称。
relid oid	被集群的表的OID。
command text	正在运行的命令。CLUSTER或VACUUM FULL。
phase text	当前处理阶段。参见Table 25.39
cluster_index_relid oid	如果正在使用索引对表进行扫描，这就是正在使用的索引的OID；否则为0。
heap_tuples_scanned bigint	扫描的堆元组数。这个计数器只有在阶段为seq scanning heap, index scanning heap 或 writing new heap时才会增进。
heap_tuples_written bigint	写入的堆元组的数量。这个计数器只有在阶段为seq scanning heap, index scanning heap 或 writing new heap时才会前进。
heap_blks_total bigint	表中的堆块总数。这个数字是在seq scanning heap的开始时报告的。
heap_blks_scanned bigint	扫描的堆块数量。这个计数器只有在阶段为seq scanning heap时才会增进。
index_rebuild_count bigint	重建的索引数。该计数器仅在重建索引阶段时才会增进。

Table 25.39. CLUSTER 和 VACUUM FULL 阶段

阶段	描述
初始化	该命令准备开始扫描堆栈。这个阶段预计会非常短暂。
seq扫描堆	该命令目前采用顺序扫描的方式对表进行扫描。
索引扫描堆	CLUSTER目前正在使用索引扫描表。
元组排序	CLUSTER目前正在对元组进行排序。
新写入堆	CLUSTER目前正在编写新的堆。

阶段	描述
交换关系文件	目前，该命令正在将新建的文件调换到位。
重建索引	该命令目前正在重建一个索引。
清理	该命令正在执行最后的清理工作。 当此阶段完成后，CLUSTER或VACUUM FULL将结束。

## 25.4.5. 基础备份进度报告

每当像lt\_basebackup这样的应用程序进行基本备份时， pg\_stat\_progress\_basebackup视图将包含当前运行BASE\_BACKUP复制命令和流备份的每个WAL发送进程的一行。 下面的表描述了将要报告的信息，并提供了关于如何解释它的信息。

Table 25.40. pg\_stat\_progress\_basebackup 视图

列类型	描述
pid integer	WAL发送方进程ID。
phase text	目前的处理阶段。 参见 Table 25.41
backup_total bigint	将被流传输的数据总量。这是在streaming database files阶段开始时的估计和报告。 注意，这只是一个近似值，因为在streaming database files阶段，数据库可能会改变，而WAL日志可能会在稍后的备份中包含。 一旦流数据量超过了估计的总大小，该值始终与backup_streamed相同。 如果在lt_basebackup中禁用估算(也就是说，指定了--no-estimate-size选项)，这为NULL。
backup_streamed bigint	数据流的总量。这个计数器只在streaming database files阶段或transferring wal files时增进。
tablespaces_total bigint	要流传输的表空间总数。
tablespaces_streamed bigint	流传输的表空间数。此计数器仅在streaming database files阶段增进。

Table 25.41. 基础备份阶段

阶段	描述
initializing	WAL发送器进程正在准备开始备份。这个阶段预计会非常短暂。
waiting for checkpoint to finish	WAL发送器进程目前正在执行pg_start_backup以准备进行基础备份，并等待启动备份检查点完成。
estimating backup size	WAL发送程序目前正在估计将作为基础备份流传输的数据库文件的总量。
streaming database files	WAL发送器当前正在流数据库文件作为基础备份。
waiting for wal archiving to finish	WAL发送方进程目前正在执行pg_stop_backup以完成备份，并等待基础备份所需的所有WAL文件成功存档。 如果在lt_basebackup中指定了--wal-method=none或--wal-method=stream，则备份将在此阶段完成后结束。
transferring wal files	WAL发送器进程正在传输备份过程中产生的所有WAL日志。 如果lt_basebackup中指定了--wal-method=fetch，则该阶段发生在waiting for wal archiving to finish阶段之后。当此阶段完成时备份将结束。

## 25.5. 动态追踪

LightDB提供了功能来支持数据库服务器的动态追踪。这样就允许在代码中的特 定点上调用外部工具来追踪执行过程。

一些探针或追踪点已经被插入在源代码中。这些探针的目的是被数据库开发者和管理员使用。默认情况下，探针不被编译到LightDB中；用户需要显式地告诉配置脚本使得探针可用。

目前，在写本文当时DTrace<sup>1</sup>已被支持，它在 Solaris、macOS、FreeBSD、NetBSD 和 Oracle Linux 上可用。Linux 的SystemTap<sup>2</sup>项目提供了一种可用的 DTrace 等价物。支持其他动态追踪工具在理论上可以通过改变src/include/utlils/probes.h中的宏定义实现。

### 25.5.1. 动态追踪的编译

默认情况下，探针是不可用的，因此你将需要显式地告诉配置脚本让探针在LightDB中可用。要包括 DTrace 支持，在配置时指定--enable-dtrace。

### 25.5.2. 内建探针

如Table 25.4所示，源代码中提供了一些标准探针。Table 25.4显式了在探针中使用的类型。当然，可以增加更多探针来增强LightDB的可观测性。

Table 25.42. 内建 DTrace 探针

名称	参数	描述
transaction-start	(LocalTransactionId)	在一个新事务开始时触发的探针。arg0 是事务 ID。
transaction-commit	(LocalTransactionId)	在一个事务成功完成时触发的探针。arg0 是事务 ID。
transaction-abort	(LocalTransactionId)	当一个事务失败完成时触发的探针。arg0 是事务 ID。
query-start	(const char *)	当一个查询的处理被开始时触发的探针。arg0 是查询字符串。
query-done	(const char *)	当一个查询的处理完成时触发的探针。arg0 是查询字符串。
query-parse-start	(const char *)	当一个查询的解析被开始时触发的探针。arg0 是查询字符串。
query-parse-done	(const char *)	当一个查询的解析完成时触发的探针。arg0 是查询字符串。
query-rewrite-start	(const char *)	当一个查询的改写被开始时触发的探针。arg0 是查询字符串。
query-rewrite-done	(const char *)	当一个查询的改写完成时触发的探针。arg0 是查询字符串。
query-plan-start	()	当一个查询的规划被开始时触发的探针。
query-plan-done	()	当一个查询的规划完成时触发的探针。
query-execute-start	()	当一个查询的执行被开始时触发的探针。

<sup>1</sup> <https://en.wikipedia.org/wiki/DTrace>

<sup>2</sup> <https://sourceware.org/systemtap/>

名称	参数	描述
query-execute-done	()	当一个查询的执行完成时触发的探针。
statement-status	(const char *)	任何时候当服务器进程更新它的pg_stat_activity.status时触发的探针。arg0 是新的状态字符串。
checkpoint-start	(int)	当一个检查点被开始时触发的探针。arg0 传递位标志来区分不同的检查点类型，例如关闭 (shutdown)、立即 (immediate) 或强制 (force)。
checkpoint-done	(int, int, int, int, int)	当一个检查点完成时触发的探针 (检查点处理过程中序列中列出的下一个触发的探针)。arg0 是要写的缓冲区数量。arg1 是缓冲区的总数。arg2、arg3 和 arg4 分别包含了增加、删除和循环回收的 WAL 文件的数量。
clog-checkpoint-start	(bool)	当一个检查点的 CLOG 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
clog-checkpoint-done	(bool)	当一个检查点的 CLOG 部分完成时触发的探针。arg0 的含义与clog-checkpoint-start中相同。
subtrans-checkpoint-start	(bool)	当一个检查点的 SUBTRANS 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
subtrans-checkpoint-done	(bool)	当一个检查点的 SUBTRANS 部分完成时触发的探针。arg0 的含义与subtrans-checkpoint-start中相同。
multixact-checkpoint-start	(bool)	当一个检查点的 MultiXact 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
multixact-checkpoint-done	(bool)	当一个检查点的 MultiXact 部分完成时触发的探针。arg0 的含义与multixact-checkpoint-start中相同。
buffer-checkpoint-start	(int)	当一个检查点的写缓冲区部分被开始时触发的探针。arg0 传递位标志来区分不同的检查点类型，例如关闭 (shutdown)、立即 (immediate) 或强制 (force)。
buffer-sync-start	(int, int)	当我们在检查点期间开始写脏缓冲区时 (在标识哪些缓冲区必须被写之后) 触发的探针。arg0 是

名称	参数	描述
		缓冲区总数, arg1 是当前为脏并且需要被写的缓冲区数量。
buffer-sync-written	(int)	在检查点期间当每个缓冲区被写完之后触发的探针。arg0 是缓冲区的 ID。
buffer-sync-done	(int, int, int)	当所有脏缓冲区被写之后触发的探针。arg0 是缓冲区总数。arg1 是检查点进程实际写的缓冲区数量。arg2 是期望写的数目 (buffer-sync-start的 arg1); arg1 和 arg2 的任何的不同反映在该检查点期间有其他进程刷写了缓冲区。
buffer-checkpoint-sync-start	()	在脏缓冲区被写入到内核之后并且在开始发出 fsync 请求之前触发的探针。
buffer-checkpoint-done	()	当同步缓冲区到磁盘完成时触发的探针。
twophase-checkpoint-start	()	当一个检查点的两阶段部分被开始时触发的探针。
twophase-checkpoint-done	()	当一个检查点的两阶段部分完成时触发的探针。
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	当一次缓冲区读被开始时触发的探针。arg0 和 arg1 包含该页的分叉号和块号 (如果这是一次关系扩展请求, arg1 为 -1)。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对于一个本地缓冲区, arg5 是创建临时关系的后端的 ID; 对于一个共享缓冲区, arg5 是 InvalidBackendId (-1)。arg6 为真表示一次关系扩展请求, 为假表示正常读。
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	当一次缓冲区读完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号 (如果这是一次关系扩展请求, arg1 现在包含新增加块的块号)。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对于一个本地缓冲区, arg5 是创建临时关系的后端的 ID; 对于一个共享缓冲区, arg5 是 InvalidBackendId (-1)。arg6 为真表示一次关系扩展请求, 为假表示正常读。arg7 为真表示在池中找到该缓冲区, 为假表示没有找到。
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	在发出对一个共享缓冲区的任意写请求之前触发的探针。arg0 和 arg1 包含该页的分叉号和块

名称	参数	描述
		号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个写请求完成时触发的探针（注意这只反映传递数据给内核的时间，它通常并没有实际地被写入到磁盘）。参数和buffer-flush-start的相同。
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个服务器进程开始写一个脏缓冲区时触发的探针（如果这经常发生，表示shared_buffers太小，或需要调整后写器器的控制参数）。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一次脏缓冲区写完成时触发的探针。参数与buffer-write-dirty-start相同。
wal-buffer-write-dirty-start	()	当一个服务器进程因为没有可用 WAL 缓冲区空间开始写一个脏 WAL 缓冲区时触发的探针（如果这经常发生，表示wal_buffers太小）。
wal-buffer-write-dirty-done	()	当一次脏 WAL 缓冲区完成时触发的探针。
wal-insert	(unsigned char, unsigned char)	当一个 WAL 记录被插入时触发的探针。arg0 是该记录的资源管理者 (rmid)。arg1 包含 info 标志。
wal-switch	()	当请求一次 WAL 段切换时触发的探针。
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始从一个关系读取一块时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId (-1)。
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一次块读取完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId (-1)。arg6 是实际读取的字节数，而 arg7 是请求读取的字节数（如果两者不同就意味着麻烦）。

名称	参数	描述
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始向一个关系中写入一个块时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对于一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是InvalidBackendId (-1)。
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一个块写操作完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和arg4 包含表空间、数据库和关系 OID来标识该关系。对于一个本地缓冲区，arg5 是创建临时关系的后端 ID；对于一个共享缓冲区，arg5 是InvalidBackendId (-1)。arg6 是实际写的字节数，而 arg7 是要求写的字节数（如果这两者不同，则意味着麻烦）。
sort-start	(int, bool, int, int, bool, int)	当一次排序操作开始时触发的探针。arg0 指示是堆排序、索引排序或数据排序。arg1 为真表示唯一值强制。arg2 是键列的数目。arg3 是允许使用的工作内存数（以千字节计）。如果要求随机访问排序结果，那么 arg4 为真。arg5为0时表示串行，为1时表示并行工作者，为2时表示并行领袖。
sort-done	(bool, long)	当一次排序完成时触发的探针。arg0 为真表示外排序，为假表示内排序。arg1 是用于一次外排序的磁盘块的数目，或用于一次内排序的以千字节计的内存。
lwlock-acquire	(char *, LWLockMode)	当成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片 (Tranche)。arg1 所请求的锁模式，是排他或共享。
lwlock-release	(char *)	当一个 LWLock 被释放时（但是注意还没有唤醒任何一个被释放的等待者）触发的探针。arg0 是该 LWLock 所在的切片 (Tranche)。
lwlock-wait-start	(char *, LWLockMode)	当一个 LWLock不是当即可用并且一个服务器进程因此开始等待该锁变为可用时触发的探针。arg0 是该 LWLock 所在的切片 (Tranche)。arg1 请求的锁模式，是排他或共享。

名称	参数	描述
lwlock-wait-done	(char *, LWLockMode)	当一个进程从对一个 LWLock 的等待中被释放时（它实际还没有得到该锁）时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lwlock-condacquire	(char *, LWLockMode)	当调用者指定无需等待而成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lwlock-condacquire-fail	(char *, LWLockMode)	当调用者指定无需等待而没有成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr 锁）的请求由于锁不可用开始等待时触发的探针。arg0 到 arg3 是标识被锁定对象的标签域。arg4 指示被锁对象的类型。arg5 表示被请求的锁类型。
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr 锁）的请求结束等待时（即已经得到锁）触发的探针。参数与lock-wait-start一样。
deadlock-found	()	当死锁检测器发现死锁时触发的探针。

Table 25.43. 定义用在探针参数中的类型

类型	定义
LocalTransactionId	unsigned int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	unsigned char

### 25.5.3. 使用探针

下面的例子展示了一个分析系统中事务计数的 DTrace 脚本，可以用来代替一次性能测试之前和之后的pg\_stat\_database快照：

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
    @start["Start"] = count();
}
```



```

        self->ts = timestamp;
    }

postgresql$1:::transaction-abort
{
    @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
    @commit["Commit"] = count();
    @time["Total time (ns)"] = sum(timestamp - self->ts);
    self->ts=0;
}

```

当被执行时，该例子 D 脚本给出这样的输出：

```

# ./txn_count.d `pgrep -n lightdb` or ./txn_count.d <PID>
^C

Start                71
Commit               70
Total time (ns)     2312105013

```

### Note

SystemTap 为追踪脚本使用一个不同于 DTrace 的标记，但是底层的探针是兼容的。值得注意的是，在这样写的时候，SystemTap 脚本必须使用双下划线代替连字符来引用探针名。在未来的 SystemTap 发行中这很可能会被修复。

你应该记住，DTrace 脚本需要细心地编写和调试，否则被收集的追踪信息可能会毫无意义。在大部分发现问题的情况中，它就是发生问题的部件，而不是底层系统。当讨论使用动态追踪发现的信息时，一定要附上使用的脚本以便其也被检查和讨论。

## 25.5.4. 定义新探针

开发者可以在代码中任意位置定义新的探针，当然这要重新编译之后才能生效。下面是插入新探针的步骤：

1. 决定探针名称以及探针可用的数据
2. 把该探针定义加入到src/backend/utils/probes.d
3. 如果pg\_trace.h还不存在于包含该探针点的模块中，包括它，并且在源代码中期望的位置插入TRACE\_POSTGRESQL探针宏
4. 重新编译并验证新探针是可用的

例子：. 这里是一个如何增加一个探针来用事务 ID 追踪所有新事务的例子。

1. 决定探针将被命名为transaction\_start并且需要一个LocalTransactionId类型的参数
2. 将该探针定义加入到src/backend/utils/probes.d:

```
probe transaction__start(LocalTransactionId);
```

注意探针名字中双下划线的使用。在一个使用探针的 DTrace 脚本中，双下划线需要被替换为一个连字符，因此，对用户而言transaction-start是文档名。

3. 在编译时，transaction\_\_start被转换成一个宏调用TRACE\_POSTGRESQL\_TRANSACTION\_START（注意这里是单下划线），可以通过包括头文件pg\_trace.h获得。将宏调用加入到源代码中的合适位置。在这种情况下，看起来类似：

```
TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
```

4. 在重新编译和运行新的二进制文件之后，通过运行下面的 DTrace 命令来检查新增的探针是否可用。你应该看到类似下面的输出：

```
# dtrace -ln transaction-start
   ID   PROVIDER      MODULE      FUNCTION NAME
18705 postgresql49878 postgres StartTransactionCommand transaction-
start
18755 postgresql49877 postgres StartTransactionCommand transaction-
start
18805 postgresql49876 postgres StartTransactionCommand transaction-
start
18855 postgresql49875 postgres StartTransactionCommand transaction-
start
18986 postgresql49873 postgres StartTransactionCommand transaction-
start
```

向C代码中添加追踪宏时，有一些事情需要注意：

- 需要小心的是，为探针参数指定的数据类型要匹配宏中使用的变量的数据类型，否则会发生编译错误。
- 在大多数平台上，如果用--enable-dtrace编译了LightDB，无论何时当控制经过一个追踪宏时，都会评估该宏的参数，即使没有进行追踪也会这样做。通常不需要担心你是否只在报告一些局部变量的值。但是要注意不要将开销大的函数调用放入参数中。如果你需要这样做，考虑通过检查追踪是否真的被启用来保护该宏：

```
if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
    TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
```

每个追踪宏有一个对应的ENABLED宏。

---

## Chapter 26. 监控磁盘使用

本章讨论如何监控LightDB数据库系统的磁盘使用情况。

### 26.1. 判断磁盘用量

每个表都有一个主要的堆磁盘文件，大多数数据都存储在其中。如果一个表有着可能会很宽（尺寸大）的列，则另外还有一个TOAST文件与这个表相关联，它用于存储因为太宽而不能存储在主表里面的值（参阅Section 64.2）。如果有这个附属文件，那么TOAST表上会有一个可用的索引。当然，同时还可能有索引和基表关联。每个表和索引都存放在单独的磁盘文件里——如果文件超过 1G 字节，甚至可能多于一个文件。这些文件的命名原则在Section 64.4中描述。

你可以以三种方式监视磁盘空间：使用Table 9.9中列出的SQL函数、使用oid2name模块或者人工观察系统目录。SQL函数是最容易使用的方法，同时也是我们通常推荐的方法。本节剩余的部分将展示如何通过观察系统目录来监视磁盘空间。

在一个最近清理过或者分析过的数据库上使用ltsql，你可以发出查询来查看任意表的磁盘用量：

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname =
'customer';
```

```
pg_relation_filepath | relpages
-----+-----
base/16384/16806    |         60
(1 row)
```

每个页通常都是 8K 字节（记住，relpages只会由VACUUM、ANALYZE和少数几个 DDL 命令如CREATE INDEX所更新）。如果你想直接检查表的磁盘文件，那么文件路径名应该有用。

要显示TOAST表使用的空间，我们可以使用一个类似下面这样的查询：

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
            FROM pg_index
            WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;
```

```
relname          | relpages
-----+-----
pg_toast_16806   |         0
pg_toast_16806_index |         1
```

你也可以很容易地显示索引的尺寸：

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
```

```
c.oid = i.indrelid AND
c2.oid = i.indexrelid
ORDER BY c2.relname;
```

relname	relpages
customer_id_index	26

我们很容易用下面的信息找出最大的表和索引：

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;
```

relname	relpages
bigtable	3290
customer	3144

## 26.2. 磁盘满失败

一个数据库管理员最重要的磁盘监控任务就是确保磁盘不会写满。一个写满了的数据磁盘可能不会导致数据的崩溃，但它肯定会让系统变得不可用。如果保存 WAL 文件的磁盘变满，会发生数据库服务器致命错误并且可能发生关闭。

如果你不能通过删除一些其他的東西来释放一些磁盘空间，那么你可以通过使用表空间把一些数据库文件移动到其他文件系统上去。参阅Section 20. 获取更多信息。

### Tip

有些文件系统在快满的时候性能会急剧恶化，因此不要等到磁盘完全满的时候才采取行动。

如果你的系统支持每用户的磁盘份额，那么数据库将自然地受制于用户所处的服务器给他的份额限制。超过份额的负面影响和完全用光磁盘是完全一样的。

---

# Chapter 27. 可靠性和预写式日志

本章解释预写式日志如何用于获得有效的、可靠的操作。

## 27.1. 可靠性

可靠性是任何严肃的数据库系统的重要属性，LightDB尽一切可能来保证可靠的操作。可靠的操作的一个方面是，被一个提交事务记录的所有数据应该被存储在一个非易失的区域，这样就不会因为失去电力、操作系统失败以及硬件失败（当然，除了非易失区域自身失效之外）等原因导致的数据丢失。向计算机的永久存储（磁盘驱动器或者等效的设备）成功写入数据通常可以满足这个要求。实际上，即使计算机受到致命损坏，只要磁盘驱动器幸存下来，那么它们就可以被移动到另外一台具有类似硬件的计算机上，而所有已经提交的事务将保持原状。

周期地强制数据进入磁盘盘片看上去像一件简单的操作，但实际上并不是。因为磁盘驱动器比内存和CPU要慢很多，在计算机的主存和磁盘盘片之间存在多层的高速缓存。首先，有操作系统的高速缓存，它缓冲常用的磁盘块并且组合对磁盘的写入。幸运的是，所有操作系统都给予应用一种强制从高速缓存写入磁盘的方法，LightDB则使用了那个特性（参阅wal\_sync\_method参数调节如何完成之）。

然后，在磁盘驱动器的控制器上可能还有一个高速缓存；这在RAID控制卡上是特别常见的。有些高速缓存是直写式的，即写入动作在到达的时候就立刻写入到磁盘上。其它是回写式的，即发送给驱动器的数据在稍后的某个时间写入驱动器。这样的高速缓存可能会称为可靠性灾难，因为磁盘控制器高速缓存的内存是易失性的，在发生电力失败的情况下会丢失其内容。好一些的控制卡有后备电池单元（BBU），即这种卡上面有电池可以在系统电力失败的情况下提供电力。在电力恢复之后，这些数据将会被写入磁盘驱动器。

最后，大多数磁盘驱动器都有高速缓存。有些是直写的，有些是回写的，和磁盘控制器一样，回写的磁盘高速缓存也存在数据丢失的问题。消费级别的IDE和SATA驱动器尤其可能包含回写式高速缓存，在掉电的情况下很容易丢失数据。很多固态驱动器（SSD）也具有易失性回写式高速缓存。

这些高速缓存通常可以被禁用，但是不同的操作系统和驱动器类型有不同的做法：

- 在Linux上，可以使用hdparm -I查询IDE和SATA驱动器，如果在Write cache之后有一个\*则表示写高速缓存被启用。可以用hdparm -W 0来关闭写高速缓存。可以使用sdparm<sup>1</sup>查询SCSI驱动器。使用sdparm --get=WCE来检查写高速缓存是否被启用，而sdparm --clear=WCE可以用来禁用它。
- 在FreeBSD上，IDE驱动器可以使用atacontrol查询，而写高速缓存可以用/boot/loader.conf中的hw.ata.wc=0关闭。SCSI驱动器可以使用camcontrol identify查询，而写高速缓存的查询和更改都可以使用sdparm。
- 在Solaris上，磁盘的写高速缓存被format -e控制（Solaris的ZFS文件系统对于开启的磁盘写高速缓存是安全的，因为它会发出它自己的磁盘高速缓存刷写命令）。
- 在Windows上，如果wal\_sync\_method是open\_datasync（默认值），写高速缓存可以通过取消选中My Computer\Open\disk drive\Properties\Hardware\Properties\Policies\Enable write caching on the disk禁用。另一种方法可以通过设置wal\_sync\_method为fsync或fsync\_writethrough来阻止写高速缓存。
- 在macOS上，通过设置wal\_sync\_method为fsync\_writethrough可以阻止写高速缓存。

最近的SATA驱动器（遵循ATAPI-6及更新标准）提供了一个驱动器高速缓存刷写命令（FLUSH CACHE EXT），而SCSI驱动器有一个存在很长时间的类似命令SYNCHRONIZE CACHE。这些命令

---

<sup>1</sup> <http://sg.danny.cz/sg/sdparm.html>

对于LightDB并不能直接访问，但某些文件系统（例如ZFS、ext4）可以使用它们将数据刷写到回写式驱动器的盘片上。不幸的是，这些文件系统在和后备电池单元（BBU）一起工作时的表现要略差。在这种设置下，同步命令强制所有来自控制器高速缓存的数据到磁盘，消除了BBU的很多好处。你可以运行`lt_test_fsync`程序来看你是否被影响。如果你被影响了，BBU带来的性能好处可以通过关闭文件系统的写障碍或者重新配置磁盘控制器来重新获得。如果写障碍被关闭，请确认电池是否保持有效，一个有问题的电池可能会导致数据丢失。但愿文件系统和磁盘控制器设计师们将最终解决这种次优行为。

在操作系统向存储硬件发出一个写请求的时候，它没有什么好办法来保证数据真正到达非易失的存储区域。实际上，确保所有存储部件都保证数据和文件系统元数据的完整性是管理人员的责任。避免使用那些没有电池作为后备的写高速缓存的磁盘控制器。在驱动器级别，如果驱动器不能保证在关闭（掉电）之前写入数据，那么关闭回写高速缓冲。如果你在使用SSD，注意很多SSD默认都没有兑现高速缓存刷写命令。你可以使用`diskchecker.pl`<sup>2</sup>来测试可靠的I/O子系统行为。

另外一个数据丢失的风险来自磁盘盘片写操作自身。磁盘盘片会被分割为扇区，通常每个扇区512字节。每次物理读写都对整个扇区进行操作。当一个写操作到达磁盘的时候，它可能是512字节（LightDB通常一次写8192字节或者16个扇区）的某个倍数，而写入处理可能因为电力失效在任何时候失败，这意味着某些512字节的扇区写入了，而有些没有。为了避免这样的失效，LightDB在修改磁盘上的实际页面之前，周期地把整个页面的映像写入永久WAL存储。这么做之后，在崩溃恢复的时候，LightDB可以从WAL恢复部分写入的页面。如果你的文件系统阻止部分页面写入（如ZFS），你可以通过关闭`full_page_writes`参数来关闭这种页映像。后备电池单元（BBU）磁盘控制器不阻止部分页面写入，除非它们保证数据都是以整页（8kB）写入到BBU。

LightDB也能防止由于硬件错误或者介质失败超时在存储设备上造成的各种数据损坏，例如读/写垃圾数据。

- WAL文件中的每一个记录都被一个CRC-32（32位）校验码所保护，这让我们可以判断记录内容是否正确。CRC值在我们写入每一个WAL记录时设置，并且在崩溃恢复、归档恢复和复制时检查。
- 目前数据页并没有默认地被校验，但是WAL记录中记录的整页映像将被保护。关于启用数据校验的内容详见`initdb`。
- 诸如`lt_xact`、`lt_subtrans`、`lt_multixact`、`lt_serial`、`lt_notify`、`lt_stat`、`lt_snapshots`等内部数据结构既没有被直接校验，其页面也没有被整页写保护。但是，这些数据结构是持久的话，WAL记录被写入，它允许最近的修改能在崩溃恢复时被准确重建且这些WAL记录被按照以上讨论的方式保护着。
- `lt_twophase`中的单个状态文件被CRC-32保护。
- 用在大型SQL查询中排序的临时数据库文件、物化和中间结果目前没有被校验，对于这些文件的改变也不会导致写入WAL记录。

LightDB无法避免可更正内存错误，它假定你会操作由工业标准纠错码（ECC）或更好方案保护的RAM。

## 27.2. 预写式日志（WAL）

预写式日志（WAL）是保证数据完整性的一种标准方法。对其详尽的描述几乎可以在所有（如果不是全部）有关事务处理的书中找到。简单来说，WAL的中心概念是数据文件（存储着表和索引）的修改必须在这些动作被日志记录之后才被写入，即在描述这些改变的日志记录被刷到持久存储以后。如果我们遵循这种过程，我们不需要在每个事务提交时刷写数据页面到磁盘，因为我们知道在发生崩溃时可以使用日志来恢复数据库：任何还没有被应用到数据页面的改变可以根据其日志记录重做（这是前滚恢复，也被称为REDO）。

<sup>2</sup> <https://brad.livejournal.com/2116715.html>

## Tip

因为WAL在崩溃后恢复数据库文件内容，不需要日志化文件系统作为数据文件或WAL文件的可靠存储。实际上，日志会降低性能，特别是如果日志导致文件系统数据被刷写到磁盘。幸运的是，日志期间的数据刷写常常可以在文件系统挂载选项中被禁用，例如在Linux ext3文件系统中可以使用data=writeback。在崩溃后日志化文件系统确实可以提高启动速度。

使用WAL可以显著降低磁盘的写次数，因为只有日志文件需要被刷出到磁盘以保证事务被提交，而被事务改变的每一个数据文件则不必被刷出。日志文件被按照顺序写入，因此同步日志的代价要远低于刷写数据页面的代价。在处理很多影响数据存储不同部分的小事务的服务器上这一点尤其明显。此外，当服务器在处理很多小的并行事务时，日志文件的一个fsync可以提交很多事务。

WAL也使得在线备份和时间点恢复能被支持，如Section 23.所述。通过归档WAL数据，我们可以支持回转到被可用WAL数据覆盖的任何时间：我们简单地安装数据库的一个较早的物理备份，并且重放WAL日志一直到所期望的时间。另外，该物理备份不需要是数据库状态的一个一致的快照 — 如果它的制作经过了一段时间，则重放这一段时间的WAL日志将会修复任何内部不一致性。

## 27.3. 异步提交

异步提交是一个允许事务能更快完成的选项，代价是在数据库崩溃时最近的事务会丢失。在很多应用中这是一个可接受的交换。

如前一节所述，事务提交通常是同步的：服务器等到事务的WAL记录被刷写到持久存储之后才向客户端返回成功指示。因此客户端可以确保那些报告已被提交的事务确会被保存，即便随后马上发生了一次服务器崩溃。但是，对于短事务来说这种延迟是其总执行时间的主要部分。选择异步提交模式意味着服务器将在事务被逻辑上提交后立刻返回成功，而此时由它生成的WAL记录还没有被真正地写到磁盘上。这将为小型事务的生产力产生显著地提升。

异步提交会带来数据丢失的风险。在向客户端报告事务完成到事务真正被提交（即能保证服务器崩溃时它也不会丢失）之间有一个短的时间窗口。因此如果客户端将会做一些要求其事务被记住的外部动作，就不应该用异步提交。例如，一个银行肯定不会使用异步提交事务来记录一台ATM的现金分发。但是在很多情境中不需要这种强的保证，例如事件日志。

使用异步提交带来的风险是数据丢失，而不是数据损坏。如果数据库可能崩溃，它会通过重放WAL到被刷写的最后一个记录来进行恢复。数据库将因此被恢复到一个自身一致状态，但是任何还没有被刷写到磁盘的事务将不会反映在该状态中。因此其影响就是丢失了最后的少量事务。由于事务按照提交顺序被重放，所以不会出现任何不一致性 — 例如一个事务B按照前面一个事务A的效果来进行修改，则不会出现A的效果丢失而B的效果被保留的情况。

用户可以选择每一个事务的提交模式，这样可以有同步提交和异步提交的事务并行运行。这允许我们灵活地在性能和事务持久性之间进行权衡。提交模式由用户可设置的参数synchronous\_commit控制，它可以任意使用任何一种修改配置参数的方法进行设置。一个事务真正使用的提交模式取决于当事务提交开始时synchronous\_commit的值。

特定的实用命令，如DROP TABLE，被强制按照同步提交而不考虑synchronous\_commit的设定。这是为了确保服务器文件系统和数据库逻辑状态之间的一致性。支持两阶段提交的命令页总是同步提交的，如PREPARE TRANSACTION。

如果数据库在异步提交和事务WAL记录写入之间的风险窗口期间崩溃，在该事务期间所作的修改将丢失。风险窗口的持续时间是有限制的，因为一个后台进程（“WAL写进程”）每wal\_writer\_delay毫秒会把未写入的WAL记录刷写到磁盘。风险窗口实际的最大持续时间是wal\_writer\_delay的3倍，因为WAL写进程被设计成倾向于在忙时一次写入所有页面。

### Caution

一个立刻关闭等同于一次服务器崩溃，因此也将会导致未刷写的异步提交丢失。

异步提交提供的行为与配置`fsync = off`不同。`fsync`是一个服务器范围的设置，它将会影响所有事务的行为。它禁用了LightDB中所有尝试同步写入到数据库不同部分的逻辑，并且因此一次系统崩溃（即，一个硬件或操作系统崩溃，不是LightDB本身的失败）可能造成数据库状态的任意损坏。在很多情境中，带来大部分性能提升的异步提交可以通过关闭`fsync`来获得，而且不会带来数据损坏的风险。

`commit_delay`也看起来很像异步提交，但它实际上是一种同步提交方法（事实上，`commit_delay`在异步提交时被忽略）。`commit_delay`会使事务在刷写WAL到磁盘之前有一个延迟，它期望由一个这样的事务所执行的刷写能够也服务于其他同时提交的事务。该设置可以被看成是一种时间窗口，在其期间事务可以参与到一次单一的刷写中，这种方式用于在多个事务之间摊销刷写的开销。

## 27.4. WAL配置

有几个WAL相关的配置参数会影响数据库性能。本节将解释它们的使用。关于服务器配置参数的设置的一般信息请参考Chapter 17

检查点是在事务序列中的点，这种点保证被更新的堆和索引数据文件的所有信息在该检查点之前已被写入。在检查点时刻，所有脏数据页被刷写到磁盘，并且一个特殊的检查点记录被写入到日志文件（修改记录之前已经被刷写到WAL文件）。在崩溃时，崩溃恢复过程检查最新的检查点记录用来决定从日志中的哪一点（称为重做记录）开始REDO操作。在这一点之前对数据文件所做的任何修改都已经被保证位于磁盘之上。因此，完成一个检查点后位于包含重做记录的日志段之前的日志段就不再需要了，可以将其回收或删除（当WAL归档工作时，日志段在被回收或删除之前必须被归档）。

检查点对于刷写所有脏数据页到磁盘的要求可能会导致可观的I/O负载。出于这一原因，检查点活动是被有所限制的，这样I/O在检查点开始时开始并且能在下一个检查点将要开始之间完成，这使得检查点期间的性能下降被最小化。

服务器的检查点进程常常自动地执行一个检查点。检查点在每`checkpoint_timeout`秒开始，或者在快要超过`max_wal_size`时开始。默认的设置分别是5分钟和1GB。如果从前一个检查点以来没有WAL被写入，则即使过了`checkpoint_timeout`新的检查点也会被跳过（如果正在使用WAL归档并且你想对文件被归档频率设置一个较低的限制来约束潜在的数据丢失，你应该调整`archive_timeout`参数而不是检查点参数）。也可以使用SQL命令 `CHECKPOINT`来强制一个检查点。

降低`checkpoint_timeout`和/或`max_wal_size`会导致检查点更频繁地发生。这使得崩溃后恢复更快，因为需要重做的工作更少。但是，我们必须在这点和增多的刷写脏数据页开销之间做出平衡。如果`full_page_writes`被设置（默认情况），则还有一个因素需要考虑。为了确保数据页一致性，在每个检查点之后对一个数据页的第一次修改将导致整个页面内容被日志记录。在这情况下，一个较小的检查点间隔会增加输出到WAL日志的容量，这让使用较小间隔的效果打了折扣并且将导致更多的磁盘I/O。

检查点的代价相对比较昂贵，首先是因为它们要求写出所有当前为脏的缓冲区，正如以上讨论的，第二个原因是它们会导致额外的WAL流量。因此比较明智的做法是将检查点参数设置得足够高，这样检查点就不会过于频繁地发生。你可以设置`checkpoint_warning`参数作为对于你的检查点参数的一种简单完整性检查。如果检查点的发生时间间隔比`checkpoint_warning`秒还要接近，一个消息将会被发送到服务器日志来推荐你增加`max_wal_size`。偶尔出现的这样的消息并不会导致警报，但是如果它出现得太频繁，那么就应该增加检查点控制参数。如果你没有把`max_wal_size`设置得足够高，那么在进行如大型COPY传输等批量操作的时候可能会导致出现大量类似的警告消息。



为了避免大批页面写入对I/O系统产生的冲击，一个检查点中对脏缓冲区的写出操作被散布到一段时间上。这个时间段由`checkpoint_completion_target`控制(使用`checkpoint_timeout`配置)，它用检查点间隔的一个分数表示。I/O率将被调整，以便能按照要求完成检查点：当`checkpoint_timeout`给定的秒数已经过去，或者`max_wal_size`被超过之前会发生检查点，以先达到的为准。在默认值0.9的情况下，LightDB可以被预期在下一个预定检查点之前完成每个检查点(大约是上一个检查点持续时间的90%)。这将尽可能地分散I/O，使得检查点I/O负载在整个检查点间隔内是一致的。这个缺点是延长检查点影响恢复时间，因为需要保留更多的WAL段，为了在可能的恢复中使用。关注恢复所需时间的用户可能希望减少`checkpoint_timeout`，以便检查点更频繁地出现，但仍然将I/O分散到检查点间隔。或者，`checkpoint_completion_target`可以被缩减，但这将导致更密集I/O时间(检查点期间)和更少I/O时间(检查点完成后，但在下一个计划的检查点之前)，因此不推荐。虽然`checkpoint_completion_target`可以设置为1.0，通常建议将其设置为不高于0.9(默认值)，因为检查点除了写入脏缓冲区之外还包括一些其他活动。1.0的设置极有可能导致检查点不能按时被完成，这可能由于所需的WAL段数量意外变化导致性能损失。

在Linux和POSIX平台上，`checkpoint_flush_after`允许强制OS超过一个可配置的字节数后将检查点写入的页面刷入磁盘。否则，这些页面可能会被保留在OS的页面缓存中，当检查点结束发出`fsync`时就会导致大量刷写形成延迟。这个设置通常有助于减小事务延迟，但是它也可能对性能带来负面影响，尤其是对于超过`shared_buffers`但小于OS页面缓存的负载来说更是如此。

`lt_wal`目录中的WAL段文件数量取决于`min_wal_size`、`max_wal_size`以及在之前的检查点周期中产生的WAL数量。当旧的日志段文件不再被需要时，它们将被移除或者被再利用(也就是被重命名变成数列中未来的段)。如果由于日志输出率的短期峰值导致超过`max_wal_size`，不需要的段文件将被移除直到系统回到这个限制以下。低于该限制时，系统会再利用足够的WAL文件来覆盖直到下一个检查点之前的需要。这种需要是基于之前的检查点周期中使用的WAL文件数量的移动平均数估算出来的。如果实际用量超过估计值，移动平均数会立即增加，因此它能在一定程度上适应峰值用量而不是平均用量。`min_wal_size`对回收给未来使用的WAL文件的量设置了一个最小值，这个参数指定数量的WAL将总是被回收给未来使用，即便系统很闲并且WAL用量估计建议只需要一点点WAL时也是如此。

独立于`max_wal_size`之外，始终保留最新的`wal_keep_size`兆字节的WAL文件和一个额外的WAL文件。还有，如果使用了WAL归档，旧的段在被归档之前不能被移除或者再利用。如果WAL归档无法跟上产生WAL的步伐，或者如果`archive_command`重复失败，旧的WAL文件将累积在`lt_wal`中，直到该情况被解决。一个使用了复制槽的较慢或者失败的后备服务器也会带来同样的效果(见Section 24.2.6)。

在归档恢复模式或后备模式，服务器周期性地执行重启点。和正常操作时的检查点相似：服务器强制它所有的状态到磁盘，更新`lt_control`来指示已被处理的WAL数据不需要被再次扫描，并且接着回收`lt_wal`中的任何旧日志段文件。重启点的执行频率不能高于主机上检查点的执行频率，因为重启点只有在检查点记录处才能被执行。如果从最后一个重启点之后过去了至少`checkpoint_timeout`秒或者WAL尺寸快要达到`max_wal_size`，则会到达一个检查点，这时会触发一个重启点。不过，因为对于何时可以执行一个重启点有限制，在恢复期间`max_wal_size`常常被超过，最多会超过一个检查点周期期间的WAL(不管怎样，`max_wal_size`从来不是一个硬限制，因此你应该总是应该留出充足的净空来避免耗尽磁盘空间)。

有两个常用的内部WAL函数：`XLogInsertRecord`和`XLogFlush`。`XLogInsertRecord`用于向共享内存中的WAL缓冲区里放置一个新记录。如果没有空间存放新记录，那么`XLogInsertRecord`就不得不写出(向内核缓存里写)一些填满了的WAL缓冲区。这并非我们所期望的，因为`XLogInsertRecord`用于每次数据库底层修改(比如，记录插入)时都要在受影响的数据页上持有一个排它锁，因为该操作需要越快越好。但糟糕的是，写WAL缓冲可能还会强制创建新的日志段，这花的时间甚至更多。通常，WAL缓冲区应该由一个`XLogFlush`请求来写和刷出，在大部分时候它都是发生在事务提交的时候以确保事务记录被刷写到永久存储。在那些日志输出量比较大的系统上，`XLogFlush`请求可能不够频繁，这样就避免`XLogInsert`进行写操作。在这样的系统上，我们应该通过修改配置参数`wal_buffers`的值来增加WAL缓冲区的数量。如果设置了`full_page_writes`并且系统相当繁忙，把`wal_buffers`设置得更高一些将有助于在紧随每个检查点之后的时间段里得到平滑的响应时间。

`commit_delay`定义了一个组提交领导者进程在XLogFlush中要求一个锁之后将会休眠的微妙数，而组提交追随者都排队等候在领导者之后。这样的延迟可以允许其它服务器进程把它们提交的记录追加到WAL缓冲区中，这样所有的这些记录将会被领导者的最终同步操作刷出。如果`fsync`被禁用或者当前处于活跃事务中的会话数少于`commit_siblings`，休眠将不会发生；这样就避免了在其它事务不会很快提交的情况下进行休眠。 请注意在某些平台上，休眠要求的单位是十毫秒，所以任何介于 1 和 10000 微妙之间的非零`commit_delay`设置的作用都是一样的。 还要注意在某些平台上，休眠操作的时间会比该参数所请求的要略长一点。

由于`commit_delay`的目的是允许每次刷写操作的开销能够在并发提交的事务之间进行分摊（可能会以事务延迟为代价），在能够明智地选择该设置之前有必要对代价进行量化。代价越高，在一定程度上`commit_delay`对于提高事务吞吐量的效果就越好。`lt_test_fsync`程序可以被用来衡量一次WAL刷写操作需要的平均微妙数。该程序报告的一次8kB写操作后的刷出所用的平均时间的一半常常是`commit_delay`最有效的设置，因此在优化一种特定工作负荷时，该值被推荐为起始点。当WAL日志被存储在高延迟的旋转磁盘上时，调节`commit_delay`特别有效，即使在具有非常快同步时间的存储介质上也能得到很显著的收益，例如固态驱动器或具有电池后备写高速缓存的RAID阵列。但是这应该在一个具有代表性的工作负荷下进行明确地测试。较高的`commit_siblings`值应该用在这种情况下，反之较小的`commit_siblings`值通常对高延迟介质有用。注意过高的`commit_delay`设置也很有可能增加事务延迟甚至整个事务吞吐量都会受到影响。

当`commit_delay`被设置为0（默认值），仍然有可能出现组提交的形式，但是组中的成员只能是那些在前一个刷写操作发生过程窗口中需要刷写它们提交记录的会话。在较高的客户端数量时很可能发生“gangway effect”，因此即使`commit_delay`为0，组提交的效果也很显著，并且显式地设置`commit_delay`将会没有作用。设置`commit_delay`只有在两种情况下有帮助：（1）有一些并发提交的事务，以及（2）吞吐量在某种程度上被提交率限制。但是在高旋转延迟的设备上，即使少到只有两个客户端，该设置也能有效提高事务吞吐量。

`wal_sync_method`参数决定LightDB如何请求内核强制将WAL更新到磁盘。只要满足可靠性，那么除了`fsync_writethrough`所有选项应该都是一样的，`fsync_writethrough`可以在某些时候强制磁盘高速缓存的刷写，而其他选项不能这样做。不过，哪种选项最快则可能和平台密切相关。 你可以使用`lt_test_fsync`程序来测试不同选项的速度。请注意如果你关闭了`fsync`，那么这个参数就无所谓了。

启用`wal_debug`配置参数（前提是LightDB编译的时候打开了这个支持） 将导致每次XLogInsertRecord和XLogFlush WAL调用都被记录到服务器日志。这个选项以后可能会被更通用的机制取代。

## 27.5. WAL内部

WAL是自动被启用的。除了确保满足WAL日志存放所需要的磁盘空间以及一些必要的调优外（参阅Section 27.4，管理员无需执行任何操作。

当每个新记录被写入时，WAL记录被追加到WAL日志中。 插入位置由日志序列号（LSN）描述，该日志序列号是日志中的字节偏移量， 随每个新记录单调递增。LSN值作为数据类型`pg_lsn`返回。 值可以进行比较以计算分离它们的WAL数据量，因此它们用于衡量复制和恢复的进度。

WAL日志被存放在数据目录的`lt_wal`目录里，它是作为一个文件段的集合存储的，通常每个段16MB大小（不过这个大小可以通过`initdb`配置选项`--with-wal-segsize`来修改）。 每个段分割成多个页，通常每个页为8K（该尺寸可以通过`--with-wal-blocksize`配置选项来修改）。 日志记录头部在`access/xlogrecord.h`里描述；日志内容取决于它记录的事件类型。段文件的名字是不断增长的数字，从000000010000000000000001开始。 目前这些数字不能回卷，不过要把所有可用的数字都用光也需要非常非常长的时间。

日志被放置在和主数据库文件不同的另外一个磁盘上会更好。你可以通过把`lt_wal`目录移动到另外一个位置（当然在此期间服务器应当被关闭），然后在原来的位置上创建一个指向新位置的符号链接来实现重定位日志。

WAL的目的是确保在数据库记录被修改之前先写了日志，但是这可能会被那些谎称向内核写成功的破坏，这时候它们实际上只是缓冲了数据而并未把数据存储到磁盘上。这种情况下的电源失效仍然可能导致不可恢复的数据崩溃。管理员应该确保保存LightDB的WAL日志文件的磁盘不会做这种谎报（参见Section 27.1）。

在完成一个检查点并且刷写了日志文件之后，检查点的位置被保存在文件`lt_control`里。因此在恢复的开始，服务器首先读取`lt_control`，然后读取检查点记录；接着它通过从检查点记录里标识的日志位置开始向前扫描执行 REDO操作。因为数据页的所有内容都保存在检查点之后的第一个页面修改的日志里（假设`full_page_writes`没有被禁用），所以自检查点以来的所有变化的页都将被恢复到一个一致的状态。

为了处理`lt_control`被损坏的情况，我们应该支持对于现有日志段反向扫描的功能 — 从最新到最老 — 这样才能找到最后的检查点。但这些目前还没有被实现。`lt_control`很小（比一个磁盘页小），因此它不会出现页断裂问题，并且到目前为止还没有发现仅仅由于无法读取`lt_control`本身导致数据库失败的报告。因此，尽管这在理论上是一个薄弱环节，但是`lt_control`看起来似乎并不是实际会发生的问题。

---

## Chapter 28. 逻辑复制

逻辑复制是一种基于数据对象的复制标识（通常是主键）复制数据对象及其更改的方法。我们使用术语“逻辑”来与物理复制加以区分，后者使用准确的块地址以及逐字节的复制方式。LightDB两种机制都支持，请见Chapter 24 逻辑复制允许在数据复制和安全性上更细粒度的控制。

逻辑复制使用一种发布和订阅模型，其中有一个或者更多订阅者订阅一个发布者节点上的一个或者更多发布。订阅者从它们所订阅的发布拉取数据并且可能后续重新发布这些数据以允许级联复制或者更复杂的配置。

一个表的逻辑复制通常开始于对发布者服务器上的数据取得一个快照并且将快照拷贝给订阅者。一旦这项工作完成，发布者上的更改会被实时发送给订阅者。订阅者以与发布者相同的顺序应用那些数据，这样在一个订阅中能够保证发布的事务一致性。这种数据复制的方法有时候也被称为事务性复制。

逻辑复制的典型用法是：

- 在一个数据库或者一个数据库的子集中发生更改时，把增量的改变发送给订阅者。
- 在更改到达订阅者时引发触发器。
- 把多个数据库联合到单一数据库中（例如用于分析目的）。
- 在LightDB的不同主版本之间进行复制。
- 在不同平台上（例如Linux到Windows）的LightDB实例之间进行复制。
- 将复制数据的访问给予不同的用户组。
- 在多个数据库间共享数据库的一个子集。

订阅者数据库的行为与任何其他LightDB实例相同，并且可以被用作其他数据库的发布者，只需要定义它自己的发布。当订阅者被应用当作只读时，单一的订阅中不会有冲突。在另一方面，如果应用或者对相同表集合的订阅者执行了其他的写动作，冲突可能会发生。

### 28.1. 发布

发布可以被定义在任何物理复制的主服务器上。定义有发布的节点被称为发布者。发布是从一个表或者一组表生成的改变的集合，也可以被描述为更改集合或者复制集合。每个发布都只存在于一个数据库中。

发布与模式不同，不会影响表的访问方式。如果需要，每个表都可以被加入到多个发布。当前，发布只能包含表。对象必须被明确地加入到发布，除非发布是用ALL TABLES创建的。

Publication可以选择把它们产生的更改限制为INSERT、UPDATE、DELETE以及TRUNCATE的任意组合，类似于触发器如何被特定事件类型触发的方式。默认情况下，所有操作类型都会被复制。

为了能够复制UPDATE和DELETE操作，被发布的表必须配置有一个“复制标识”，这样在订阅者那一端才能标识对于更新或删除合适的行。默认情况下，复制标识就是主键（如果有主键）。也可以在复制标识上设置另一个唯一索引（有特定的额外要求）。如果表没有合适的键，那么可以设置成复制标识“full”，它表示整个行都成为那个键。不过，这样做效率很低，只有在没有其他方案的情况下才应该使用。如果在发布者端设置了“full”之外的复制标识，在订阅者端也必须设置一个复制标识，它应该由相同的或者少一些的列组成。如何设置复制标识的细节请参考REPLICA IDENTITY。如果在复制UPDATE或DELETE操作的发布中加入了没有复制标识的表，那么订阅者上后续的UPDATE或DELETE操作将导致错误。不管有没有复制标识，INSERT操作都能继续下去。

每一个发布都可以由多个订阅者。

Publication通过使用CREATE PUBLICATION命令创建并且可以在之后使用相应的命令进行修改或者删除。

表可以使用ALTER PUBLICATION动态地增加或者移除。ADD TABLE以及DROP TABLE操作都是事务性的，因此一旦该事务提交，该表将以正确的快照开始或者停止复制。

## 28.2. 订阅

订阅是逻辑复制的下游端。订阅被定义在其中的节点被称为订阅者。一个订阅会定义到另一个数据库的连接以及它想要订阅的发布集合（一个或者多个）。

订阅者数据库的行为与任何其他LightDB实例相同，并且可以被用作其他数据库的发布者，只需要定义它自己的发布。

如果需要，一个订阅者节点可以有多个订阅。可以在一对发布者-订阅者之间定义多个订阅，在这种情况下要确保被订阅的发布对象不会重叠。

每一个订阅都将通过一个复制槽（见Section 24.2.6 接收更改。预先存在的表数据的初始数据同步过程可能会要求额外的复制槽，并且在数据同步结束后删除。

逻辑复制订阅可以是同步复制（见Section 24.2.8 的后备服务器。后备名称默认是该订阅的名称。可以在订阅的连接信息中用application\_name指定一个可供选择的名称。

如果当前用户是一个超级用户，则订阅会被lt\_dump转储。否则订阅会被跳过并且写出一个警告，因为非超级用户不能从pg\_subscription目录中读取所有的订阅信息。

可以使用CREATE SUBSCRIPTION增加订阅，并且使用ALTER SUBSCRIPTION在任何时刻停止/继续订阅，还可以使用DROP SUBSCRIPTION删除订阅。

在一个订阅被删除并且重建时，同步信息会丢失。这意味着数据必须被重新同步。

模式定义不会被复制，并且被发布的表必须在订阅者上存在。只有常规表可以成为复制的目标。例如，不能复制视图。

表在发布者和订阅者之间使用完全限定的表名进行匹配。不支持复制到订阅者上命名不同的表。

表的列也通过名称匹配。订阅表中的列顺序不需要与发表表中的顺序一样。列的数据类型也不需要一样，只要可以将数据的文本表示形式转换为目标类型即可。例如，您可以从integer类型的列复制到bigint类型的列。目标表还可以具有发表表中不存在的额外列。额外列都将使用目标表的定义中指定的默认值填充。

### 28.2.1. 复制槽管理

如前所述，每个（活动的）订阅都从远程（发布端）的复制槽接收更改。通常，当使用CREATE SUBSCRIPTION创建订阅时，远程复制槽会自动创建，并在使用DROP SUBSCRIPTION删除订阅时自动删除。然而，在某些情况下，单独操作订阅和底层复制槽可能是有用或必要的。以下是一些情况：

- 在创建一个订阅时，复制槽已经存在。在这种情况下，可以使用create\_slot = false选项创建订阅并关联到现有的槽。
- 在创建一个订阅时，远程主机不可达或者处于一种不明状态。在这种情况下，可以使用connect = false选项创建订阅。那么远程主机将根本不会被联系。这是lt\_dump所使用的方式。这样，在订阅可以被激活之前，必须手工创建远程复制槽。

- 在删除一个订阅时，复制槽应该被保留。当订阅者数据库正在被移动到一台不同的主机并且将从那里再被激活时，这种行为很有用。在这种情况下，可以在尝试删除该订阅之前，使用ALTER SUBSCRIPTION将复制槽解除关联。
- 在删除一个订阅时，远程主机不可达。在这种情况下，可以在尝试删除该订阅之前，使用ALTER SUBSCRIPTION将复制槽解除关联。如果远程数据库实例不再存在，那么不需要进一步的行动。不过，如果远程数据库实例只是不可达，那么复制槽应该被手动删除（以及任何剩余的表同步槽）。否则它将会继续保留WAL并且最终可能会导致磁盘被填满。这种情况应该要仔细地研究。

## 28.3. 冲突

逻辑复制的行为类似于正常的DML操作，即便数据在订阅者节点本地被修改，逻辑复制也会根据收到的更改来更新数据。如果流入的数据违背了任何约束，复制将停止。这种情况被称为一个冲突。在复制UPDATE或DELETE操作时，缺失的数据将不会产生冲突并且这类操作将简单地跳过。

冲突将会产生错误并且停止复制，它必须由用户手工解决。在订阅者的服务器日志中可以找到有关冲突的详细情况。

通过更改订阅者上的数据（这样它就不会与到来的数据发生冲突）或者跳过与已有数据冲突的事务可以解决这种冲突。通过调用pg\_replication\_origin\_advance()函数可以跳过该事务，函数的参数是对应于该订阅名称的node\_name以及一个位置。复制源头的当前位置可以在pg\_replication\_origin\_status系统视图中看到。

## 28.4. 限制

逻辑复制当前有下列限制或者缺失的功能。这些可能在未来的发行中解决。

- 数据库模式和DDL命令不会被复制。初始模式可以手工使用lt\_dump --schema-only进行拷贝。后续的模式改变需要手工保持同步（不过值得注意的是，模式其实不需要在两端保持绝对相同）。当一个活跃的数据库中模式定义改变时，逻辑复制是鲁棒的：当模式在发布者上发生改变并且被复制的数据开始到达订阅者但却不适合表模式时，复制将报错，直至模式被更新。在很多情况下，可以通过先对订阅者应用额外的模式更改来避免间歇性的错误。
- 序列数据不被复制。后台由序列支撑的serial或者标识列中的数据当然将被作为表的一部分复制，但是序列本身在订阅者上仍将显示开始值。如果订阅者被用作一个只读数据库，那么这通常不会是什么问题。不过，如果订阅者数据库预期有某种转换或者容错，那么序列需要被更新到最后的值，要么通过从发布者拷贝当前数据的防范（也许使用lt\_dump），要么从表本身决定一个足够高的值。
- 支持TRUNCATE命令的复制，但是在截断由外键连接在一起的表群体时必须要小心。在复制截断动作时，订阅者将截断与发布者上被截断的相同的表群体，这些表或者被明确指定或者通过CASCADE隐含地收集而来，然后还要减去不属于该订阅的表。如果所有受影响的表都属于同一个订阅，这会正确地工作。但是如果订阅者上要被截断的某些表有外键链接到不属于同一订阅的表，那么在订阅者上该截断动作的应用将会失败。
- 大对象（见Chapter 33）不会被复制。没有办法可以解决这个问题，除非把数据存储在普通表中。
- 只有表支持复制，包括分区表。试图复制其他类型的关系，例如视图、物化视图或外部表，将会导致错误。
- 在分区表之间进行复制时，实际的复制来源，缺省情况下，源自发布者上的叶子分区，因此发布者上的分区也必须作为有效的目标表存在于订阅者上。（它们可以是叶分区本身，也可以是进一步子分区段，甚至可以是独立的表。）发布还可以指定使用已分区根表的标识和模式来复制更改，而不是使用实际产生更改的各个叶分区的标识和模式（参见CREATE PUBLICATION）。

## 28.5. 架构

逻辑复制从拷贝发布者数据库上的数据库快照开始。拷贝一旦完成，发布者上的更改会在它们发生时实时传送给订阅者。订阅者按照数据在发布者上被提交的顺序应用数据，这样任意单一订阅中的发布的事务一致性才能得到保证。

逻辑复制被构建在一种类似于物理流复制（见Section 24.2.5 的架构上。它由“walsender”和“apply”进程实现。walsender进程开始对WAL的逻辑解码（在Chapter 45中描述）并且载入标准逻辑解码插件（pgoutput）。该插件把从WAL中读取的更改转换成逻辑复制协议（见Section 49.5 并且根据发布说明过滤数据。然后数据会被连续地使用流复制协议传输到应用工作者，应用工作者会把数据映射到本地表并且以正确的事务顺序应用它们接收到的更改。

订阅者数据库上的应用进程总是将session\_replication\_role设置为replica运行，这会产生触发器和约束上通常的效果。

逻辑复制应用进程当前只会引发行触发器，而不会引发语句触发器。不过，初始的表同步是以类似一个COPY命令的方式实现的，因此会引发INSERT的行触发器和语句触发器。

### 28.5.1. 初始快照

获取现有订阅表中初始数据的快照并且以一种特殊类型的应用进程的并行实例进行拷贝。这种进程将创建自己的复制槽并且拷贝现有的数据。复制完成后，表内容对其他后端可见。一旦现有的数据被拷贝完，worker进程会进入到同步模式，主应用进程会流式更新在使用标准逻辑复制拷贝初始数据期间发生的任意改变，这会确保表变为已同步的状态。在此同步阶段，应用和提交更改的顺序与它们在发布者发生的顺序相同。一旦同步完成，该表的复制的控制权会被交回给主应用进程，其中复制会照常继续。

## 28.6. 监控

因为逻辑复制是基于与物理流复制相似的架构的，一个发布节点上的监控也类似于对物理复制主节点（见Section 24.2.5.2 的监控。

有关订阅的监控信息在pg\_stat\_subscription中可以看到。每一个订阅工作者在这个视图都有一行。一个订阅能有零个或者多个活跃订阅工作者取决于它的状态。

通常，对于一个已启用的订阅会有单一的应用进程运行。一个被禁用的订阅或者崩溃的订阅在这个视图中不会有行存在。如果有任何表的数据同步正在进行，对正在被同步的表会有额外的工作者。

## 28.7. 安全性

能够修改订阅者侧模式的用户可以作为超级用户执行任意代码。将这些表上的所有权和TRIGGER特权限制为超级用户信任的角色。此外，如果不受信任的用户可以创建表，则只使用显式列出表的发布。也就是说，只有当超级用户信任允许在发布者或订阅者上创建非临时表的所有用户时，才创建FOR ALL TABLES的订阅。

用于复制连接的角色必须具有REPLICATION属性（或者是超级用户）。如果角色缺少SUPERUSER 和 BYPASSRLS，发布者的行安全策略可以执行。如果角色不信任全部表所有者，则在连接字符串中包含options=-crow\_security=off;如果表所有者随后添加行安全策略，则该设置将导致复制中断，而不是执行该策略。角色的访问权限必须在lt\_hba.conf中配置，并且必须具有LOGIN属性。

为了能够拷贝初始表数据，用于复制连接的角色必须在被发布的表上具有SELECT特权（或者是一个超级用户）。

要创建发布，用户必须在数据库中有CREATE特权。

要把表加入到一个发布，用户必须在该表上有拥有权。要创建一个自动发布所有表的发布，用户必须是一个超级用户。

要创建订阅，用户必须是一个超级用户。

订阅的应用过程将在本地数据库上以超级用户的特权运行。

特权检查仅在复制连接开始时被执行一次。在从发布者读到每一个更改记录时不会重新检查特权，在每一个更改被应用时也不会重新检查特权。

## 28.8. 配置设置

逻辑复制要求设置一些配置选项。

在发布者端，`wal_level`必须被设置为`logical`，而`max_replication_slots`中设置的值必须至少是预期要连接的订阅数加上保留给表同步的连接数。`max_wal_senders`应该至少被设置为`max_replication_slots`加上同时连接的物理复制体的数量。

订阅者必须配置`max_replication_slots`。它必须设置为至少是订阅者数，加上一些用于表同步的预留。`max_logical_replication_workers`必须至少被设置为订阅数加上保留给表同步的连接数。此外，可能需要调整`max_worker_processes`以容纳复制工作者，至少为 $(\text{max\_logical\_replication\_workers} + 1)$ 。注意，一些扩展和并行查询也会从`max_worker_processes`中取得工作者槽。

## 28.9. 快速设置

首先在`lightdb.conf`中设置配置选项：

```
wal_level = logical
```

对于一个基础设置来说，其他所需的设置使用默认值就足够了。

需要调整`lt_hba.conf`以允许复制（这里的值取决于实际的网络配置以及用于连接的用户）：

```
host      all      repuser    0.0.0.0/0      md5
```

然后在发布者数据库上：

```
CREATE PUBLICATION mypub FOR TABLE users, departments;
```

并且在订阅者数据库上：

```
CREATE SUBSCRIPTION mysub CONNECTION 'dbname=foo host=bar user=repuser'
PUBLICATION mypub;
```

上面的语句将开始复制过程，它会同步表`users`以及`departments`的初始表内容，然后开始复制对那些表的增量更改。



---

## Chapter 29. 即时编译 (JIT)

这一章解释什么是即时编译以及如何在LightDB中配置即时编译。

### 29.1. 什么是JIT编译?

即时 (Just-In-Time, JIT) 编译是将某种形式的解释程序计算转变成原生程序的过程, 并且这一过程是在运行时完成的。例如, 与使用能够计算任意SQL表达式的通用代码来计算一个特定的SQL谓词 (如WHERE a.col = 3) 不同, 可以产生一个专门针对该表达式的函数并且可以由CPU原生执行, 从而得到加速。

LightDB内建支持用LLVM<sup>1</sup>执行JIT编译。

进一步的细节请参考src/backend/jit/README。

#### 29.1.1. JIT加速的操作

当前, LightDB的JIT实现支持对表达式计算以及元组拆解的加速。未来可能有更多其他操作采用这种技术加速。

表达式计算被用来计算WHERE子句、目标列表、聚集以及投影。通过为每一种情况生成专门的代码来实现加速。

元组拆解是把一个磁盘上的元组 (见Section 64.6.1) 转换成其在内存中表示的过程。通过创建一个专门针对该表布局和被抽取的列数的函数来实现加速。

#### 29.1.2. 内联

LightDB有很好的扩展性并且允许定义新的数据类型、函数、操作符以及其他数据库对象, 见Chapter 36 实际上, 内建对象都使用近乎完全相同的机制来实现。这种可扩展性隐含了一些开销, 例如函数调用带来的开销 (见Section 36.3)。为了降低这类开销, JIT编译可以把小函数的函数体内联到使用它们的表达式中。这种方式可以优化掉可观的开销。

#### 29.1.3. 优化

LLVM支持对生成的代码进行优化。一些优化代价很低, 以至于可以在每次使用JIT时都执行, 而另一些优化则只有在运行时间较长的查询中才能获益。更多有关优化的细节请参考<https://llvm.org/docs/Passes.html#transform-passes>。

### 29.2. 什么时候会用JIT?

JIT编译主要可以让长时间运行的CPU密集型的查询受益。对于短查询, 执行JIT编译增加的开销常常比它节省的时间还要多。

为了判断是否应该使用JIT编译, 会用到一个查询的总的估计代价 (见Chapter 6和Section 17.7.2)。查询的估计代价将与jit\_above\_cost的设置进行比较。如果代价更高, JIT编译将被执行。然后需要两个进一步的决定。首先, 如果估计代价超过jit\_inline\_above\_cost的设置, 该查询中使用的短函数和操作符都将被内联。其次, 如果估计代价超过jit\_optimize\_above\_cost的设置, 会应用昂贵的优化来改进产生的代码。这些选项中的每一种都会增加JIT编译的开销, 但是可以可观地降低查询执行时间。

这些基于代价的决定将在规划时做出, 而不是在执行时做出。这意味着, 在使用预备语句并且使用了一个一般性的计划时 (见PREPARE), 配置参数的值实际上是在预备时控制这些决定, 而不是由执行时的设置来决定。

---

<sup>1</sup> <https://llvm.org/>

## Note

如果jit被设置为off或者没有JIT实现可用（例如因为服务器没有用--with-llvm编译），即便基于上述原则能带来很大的好处，JIT也不会被执行。把jit设置成off对规划时和执行时都有影响。

EXPLAIN可以被用来看看是否使用了JIT。例如，这是一个没有使用JIT的查询：

```
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                QUERY PLAN
-----
Aggregate (cost=16.27..16.29 rows=1 width=8) (actual time=0.303..0.303 rows=1
 loops=1)
  -> Seq Scan on pg_class (cost=0.00..15.42 rows=342 width=4) (actual
 time=0.017..0.111 rows=356 loops=1)
 Planning Time: 0.116 ms
 Execution Time: 0.365 ms
(4 rows)
```

看看给出的计划代价，不使用JIT是非常合理的，JIT的代价会比可能得到的节省更高。调整代价限制会导致用到JIT：

```
=# SET jit_above_cost = 10;
SET
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                QUERY PLAN
-----
Aggregate (cost=16.27..16.29 rows=1 width=8) (actual time=6.049..6.049 rows=1
 loops=1)
  -> Seq Scan on pg_class (cost=0.00..15.42 rows=342 width=4) (actual
 time=0.019..0.052 rows=356 loops=1)
 Planning Time: 0.133 ms
 JIT:
  Functions: 3
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 1.259 ms, Inlining 0.000 ms, Optimization 0.797 ms,
 Emission 5.048 ms, Total 7.104 ms
 Execution Time: 7.416 ms
```

如这里所看到的，JIT被用到了，但是内联和昂贵的优化没有被用到。如果jit\_inline\_above\_cost或者jit\_optimize\_above\_cost也被降低，这种情况会被改变。

## 29.3. 配置

配置变量jit决定启用或者禁用JIT编译。如果它被启用，配置变量jit\_above\_cost、jit\_inline\_above\_cost以及jit\_optimize\_above\_cost判断是否要为一个查询执行JIT编译以及在执行中花费多大的努力。

jit\_provider决定使用哪一种JIT实现。很少需要改变这一设置。见Section 29.4.2

如Section 17.1中所述，对于开发和调试目的，还有一些额外的配置参数存在。

## 29.4. 可扩展性

### 29.4.1. 对扩展的内联支持

LightDB的JIT实现可以内联C以及internal类型的函数体，还有基于这类函数的操作符。为了能对扩展中的函数这样做，需要让那些函数的定义可用。在使用PGXS对一个已经编译有LLVM JIT支持的服务器构建一个扩展时，相关的文件将被自动构建并且安装。

相关的文件必须被安装在\$pkglibdir/bitcode/\$extension/中并且对它们的一份概要必须被安装在\$pkglibdir/bitcode/\$extension.index.bc中，其中\$pkglibdir是lt\_config --pkglibdir返回的目录里，而\$extension是扩展的共享库的基础名称。

#### Note

对于编译在LightDB本身中的函数，其bitcode被安装在\$pkglibdir/bitcode/postgres。

### 29.4.2. 可插拔的JIT提供者

LightDB提供一种基于LLVM的JIT实现。JIT提供者的接口是可插拔的，可以无需重编译就能改变提供者（尽管当前构建过程仅提供了对LLVM的内联支持数据）。活跃的提供者通过设置jit\_provider来选择。

#### 29.4.2.1. JIT提供者接口

JIT提供者需要通过动态装载其共享库来载入。正常的搜索路径被用来定位该库。为了提供所要求的JIT提供者回调并且表示该库实际上是一个JIT提供者，它需要提供一个名为\_PG\_jit\_provider\_init的C函数。会有一个结构被传入这个函数，在函数中应该用回调函数指针来填充该结构：

```
struct JitProviderCallbacks
{
    JitProviderResetAfterErrorCB reset_after_error;
    JitProviderReleaseContextCB release_context;
    JitProviderCompileExprCB compile_expr;
};

extern void _PG_jit_provider_init(JitProviderCallbacks *cb);
```

---

## Chapter 30. 回归测试

回归测试是LightDB中对于 SQL 实现的一组综合测试集。它们测试标准 SQL 操作以及LightDB的扩展能力。

### 30.1. 运行测试

回归测试可以在一个已经安装并运行的服务器上运行，或者在编译树中的一个临时安装上运行。此外，还有运行该测试的“并行”和“顺序”模式。顺序方法单独运行每一个测试脚本，而并行方法则开启多个服务器进程来并行地运行多组测试。并行测试能够发现进程间通信和锁定是否工作正确。

#### 30.1.1. 在一个临时安装上运行测试

要在编译之后且在安装之前运行并行回归测试，可在顶层目录中键入：

```
make check
```

（或者你可以切换到src/test/regress并且在那里运行该命令）。最后你应该看到这样的信息：

```
=====  
All 193 tests passed.  
=====
```

或者关于哪些测试失败的提示。见下面的Section 30.2来确定一个“失败”是否表示一个严重的问题。

因为这种测试方法运行一个临时服务器，如果你作为根用户进行了编译，它将无法工作，因为服务器无法用 root 启动。我们推荐的过程是不要作为 root 编译，或者在完成安装后执行测试。

如果你已经配置LightDB安装到一个已经存在有旧的LightDB安装的位置，并且你在安装新版本前执行了make check，你可能会发现测试会因为新程序尝试使用已经安装的共享库而失败（典型特征是抱怨未定义的符号）。如果你希望在覆盖旧安装之前运行测试，你将需要使用configure --disable-rpath编译。但是我们不推荐为最终安装使用这个选项。

并行回归测试会在你的用户 ID 下启动相当多的进程。当前，最大并发量是二十个并行测试脚本，这意味着四十个进程：对每一个测试脚本有一个服务器进程和一个ltsql进程。因此如果你的系统对每个用户的进程数有强制限制，确保这个限制至少是五十，否则你将在并行测试中失败。如果你没有权利提升该限制，你可以通过设置MAX\_CONNECTIONS参数来降低并发度。例如：

```
make MAX_CONNECTIONS=10 check
```

会并发运行不超过十个测试。

#### 30.1.2. 在一个现有安装上运行测试

要在安装后运行测试，初始化一个数据目录并且按照Chapter 16解释的启动服务器，然后输入：

```
make installcheck
```

或者进行一次并行测试：

```
make installcheck-parallel
```

该测试将期望联系在本地主机和默认端口号上的服务器（除非通过LTHOST和LTPORT环境变量覆盖）。该测试将在一个名为regression的数据库中运行，任何以该名称存在的数据库将被删除。

该测试还将短暂地创建一些集簇范围内的对象，例如角色、表空间和订阅。这些对象的名称都会以regress\_开始。在实际具有以这种方式全局对象命名的安装中使用installcheck模式时要格外小心。

### 30.1.3. 附加测试套件

make check和make installcheck命令只运行“核心的”回归测试，这只测试LightDB服务器的内建功能。源代码发布包含许多额外的测试套件，它们中的大部分用于测试附加功能，例如可选的过程语言。

要运行将被编译模块的所有测试套件（包括核心测试），在编译树的顶端输入这些命令之一：

```
make check-world
make installcheck-world
```

这些命令分别在临时服务器或已经安装好的服务器上运行测试（与之前介绍的make check和make installcheck类似）。其他的考虑与之前为每种方法解释的相同。注意make check-world为每一个受测模块建立一个独立的实例（临时数据目录），因此它比起make installcheck-world需要更多的时间和磁盘空间。

在具有多个 CPU 内核且没有严格操作系统限制的现代计算机上，你可以通过并行化使操作速度大大加快。大多数 LightDB 开发人员实际用于运行所有测试的方法类似于这样

```
make check-world -j8 >/dev/null
```

通过-j限制接近或略高于可用内核数。当你只是想验证成功时，放弃stdout消除那些没有意义的闲杂信息。（如果发生故障，stderr消息通常足以确定要查找的位置。）

你也可以通过在编译树适当的子目录中输入make check或make installcheck来运行个体的测试套件。记住make installcheck假设你已经安装了相关模块，而不仅仅是核心服务器。

可以以这种方法调用的额外测试包括：

- 可选过程语言的回归测试。这些位于src/pl之下。
- contrib模块的回归测试，位于contrib。不是所有的contrib模块都有测试。
- ECPG 接口库的回归测试，位于src/interfaces/ecpg/test。
- 位于src/test/authentication的核心支持的身份验证方法的测试。（更多身份验证相关的测试请参阅下文。）
- 并发会话行为的压力测试，位于src/test/isolation。
- 崩溃恢复和物理复制的测试，位于 src/test/recovery。

- 逻辑复制的测试，位于 `src/test/subscription`。
- 客户端的测试程序在 `src/bin` 下。

在使用 `installcheck` 模式时，这些测试将建立并毁掉名字包括 `regression` 的测试数据库，例如 `pl_regression` 或 `contrib_regression`。请注意使用 `installcheck` 模式，对于安装时以这种方式命名的任何非测试数据库。

其中一些辅助测试套件使用的 TAP 基础结构解释在 Section 30.4 只有在 LightDB 被使用选项 `--enable-tap-tests` 配置时，基于 TAP 的测试才能被运行。推荐在开发时使用这种方式，但如果沒有合适的 Perl 安装可用也可以忽略。

一些测试套件默认不会被运行，因为它们在多用户系统上运行不安全或者它们需要特殊的软件。可以通过设置 `make` 或环境变量为空格分隔的列表来决定额外运行哪些测试套件，例如：

```
make check-world PG_TEST_EXTRA='kerberos ldap ssl'
```

当前支持下列值：

`kerberos`

运行 `src/test/kerberos` 下的测试套件。这要求一个 MIT Kerberos 安装并且打开 TCP/IP 监听端口。

`ldap`

运行 `src/test/ldap` 下的测试套件。这要求一个 OpenLDAP 安装并且打开 TCP/IP 监听端口。

`ssl`

运行 `src/test/ssl` 下的测试套件。这会打开 TCP/IP 监听端口。

即便在 `PG_TEST_EXTRA` 中提到了当前编译配置不支持的特性，针对它们的测试也不会被运行。

此外，在 `src/test/modules` 中的测试，将由 `make check-world` 而不是 `make installcheck-world` 运行。这是因为它们安装非生产扩展，或者具有其他被认为不适合生产安装的副作用。如果你愿意，你可以在这些子目录中使用 `make install` 和 `make installcheck`，但不建议使用非测试服务器执行此操作。

### 30.1.4. 区域和编码

默认情况下，测试使用的临时安装将使用在当前环境中定义的区域和由 `initdb` 决定的相应数据库编码。通过设置适当的环境变量来测试不同的区域是有用的，例如：

```
make check LANG=C
make check LC_COLLATE=en_US.utf8 LC_CTYPE=fr_CA.utf8
```

由于实现的原因，为此目的设置 `LC_ALL` 不能工作，所有其他区域相关的环境变量都可以工作。

在对一个现有安装测试时，区域由现有数据库集簇决定并且不能为测试而独立设置。

你也可以通过设置变量 `ENCODING` 来显式地选择数据库编码，例如：

```
make check LANG=C ENCODING=EUC_JP
```

这样设置数据库编码通常只对区域为 `C` 有意义；否则编码将自动从区域选择，并且指定一个不匹配区域的编码将会导致错误。

不管测试是针对临时安装还是已有安装，数据库编码都可以被设置，然而在后一种情况中它必须与安装的区域相兼容。

### 30.1.5. 额外测试

核心回归测试套件包含一些默认情况下不被运行的测试文件，因为它们可能平台相关的或者需要很长时间来运行。你可以通过设置变量EXTRA\_TESTS来运行这些或者其他额外测试文件。例如，要运行numeric\_big测试：

```
make check EXTRA_TESTS=numeric_big
```

### 30.1.6. 测试热备

源代码发布中还包含有用于热备的静态行为的回归测试。这些测试需要一个运行着的主服务器和一个运行着的后备服务器，并且后备服务器正从主服务器接受新的 WAL 改变（使用基于文件的日志传送或流复制）。那些服务器不是自动创建的，这里也没有关于建立复制的文档。请查阅本文档中的相关章节。

要运行热备测试，首先在主服务器上创建一个名为regression的数据库：

```
ltsql -h primary -c "CREATE DATABASE regression"
```

接下来，在主服务器上的 regression 数据库中运行准备脚本src/test/regress/sql/hs\_primary\_setup.sql，例如：

```
ltsql -h primary -f src/test/regress/sql/hs_primary_setup.sql regression
```

允许这些改变传播到后备服务器。

为受测后备服务器安排默认数据库连接（例如通过设置LTHOST和LTPORT环境变量）。最后，在 regression 目录中运行make standbycheck：

```
cd src/test/regress
make standbycheck
```

在主服务器上也可以使用src/test/regress/sql/hs\_primary\_extremes.sql脚本生成某些极限行为来允许测试后备服务器的行为。

## 30.2. 测试评估

一些正确安装的并且全功能的LightDB安装可能会在这些回归测试中的某些上“失败”，其原因是平台相关的因素，例如可变浮点表示和消息措辞。这些测试目前采用diff命令来比较测试输出和在参考系统上产生的输出，这样测试的结果对小的系统差异也很敏感。当一个测试被报告为“失败”时，请总是检查实际结果和期望结果之间的差异，你可能会发现该差异其实并不明显。不管怎样，我们将努力维护在所有被支持平台上的准确的参考文件，以期待所有的测试都能通过。

回归测试的实际输出在src/test/regress/results目录中的文件内。测试脚本会使用diff来把每一个输出文件与存储在src/test/regress/expected目录中的参考输出进行比较。任何差异都被保存在src/test/regress/regression.diffs中便于你的观察（当运行一个除核心测试之外的测试套件时，这些文件当然会出现在相关子目录中，而不是src/test/regress）。

如果你不喜欢被默认使用的diff选项，请设置环境变量PG\_REGRESS\_DIFF\_OPTS，例如PG\_REGRESS\_DIFF\_OPTS='-c'（或者如果你愿意，你可以自己运行diff）。

如果由于某种原因一个特定的平台对一个给定测试产生了“失败”，而对输出的检查却说明该结果是合法的，你可以增加一个新的比较文件来让失败报告在未来的测试运行中保持沉默。详见Section 30.3

### 30.2.1. 错误消息差异

某些回归测试涉及到故意的非法输入值。错误消息可能来自LightDB代码或主机平台系统例程。在后一种情况中，消息会随着平台而变化，但是会反映相似的信息。这些消息中的差异将导致一次“失败的”回归测试，这可以通过检查来确认。

### 30.2.2. 区域差异

如果你在一台使用除 C 之外的排序规则顺序区域初始化的服务器上运行测试，那么可能会出现由于排序顺序和后续失败产生的差异。回归测试套件被设置为可以处理这种问题，方法是提供替代的结果文件来处理大量的区域。

要在使用临时安装方法时在一种不同的区域中运行测试，可在make命令行上传递适当的区域相关的环境变量，例如：

```
make check LANG=de_DE.utf8
```

（回归测试驱动器会取消LC\_ALL设置，因此使用这个变量选择区域是不起作用的）。要不使用区域，要么取消所有区域相关的环境变量设置（或把它们设置为C），要么使用下列特殊调用：

```
make check NO_LOCALE=1
```

当对一个现有安装运行测试时，区域设置由现有安装决定。要改变它，通过向initdb传递合适的选项来使用不同的区域初始化数据库集簇。

通常，我们建议对将要在生产环境中使用的区域设置运行回归测试，因为这样可以测试即将真正被用在生产环境中的与区域和编码相关的代码。根据操作系统环境，你可能会得到失败，但是那样你将至少知道在真实应用运行时会得到什么样的与区域相关的行为。

### 30.2.3. 日期和时间差异

大部分的日期和时间结果依赖于时区环境。参考文件是用时区PST8PDT（伯克利，加利福尼亚）生成的，并且如果测试不是运行在该时区设置中显然会出现失败。回归测试驱动器会设置环境变量PGTZ为 PST8PDT，这通常能保证正确的结果。

### 30.2.4. 浮点差异

某些测试涉及到从表列中计算 64 位浮点数（双精度）。我们已经发现了涉及到双精度列的数学函数的结果中的差异。float8和geometry测试容易在不同平台之间产生小的差异，甚至对不同的编译器优化设置也可能产生差异。这些差异通常位于小数点右边的 10 个位置，决定这些差异的实际意义需要人类眼球比较。

某些系统显示负零为-0，而其他的只显示0。

某些系统标志来自pow()和exp()的错误的机制不同于当前LightDB代码所期望的机制。

### 30.2.5. 行序差异

你可能看到这样一些差异：一组相同的行在输出中的顺序与参考文件中的顺序不同。严格来说，在大部分情况下这不是缺陷。大部分回归测试脚本没有为每一个单独的SELECT使用一个ORDER BY，并且因此它们的结果行顺序根据 SQL 规范是非良定义的。实际上，因为我们考



虑的是由相同的软件在相同的数据上执行相同的查询，我们通常会在所有平台上得到相同的结果顺序，所以缺少ORDER BY不是一个问题。但是，某些查询确实会在不同平台上产生不同的顺序。当对一个已经安装的服务器运行测试时，顺序差异可能由非 C 区域设置或非默认参数设置导致，例如work\_mem的自定义值或规划器代价参数。

因此，如果你看到一个顺序差异，没有什么可担心的，除非查询确实有一个ORDER BY而结果却违反了。但是，不管怎样请报告它，这样我们可以为特定的查询加上一个ORDER BY来在未来的发布中消除虚假的“失败”。

你可能好奇为什么我们不对所有回归测试查询进行显式排序来一次性解决这个问题。其原因是那可能会降低回归测试的有用性，因为它们已经倾向于测试产生有序结果的查询计划类型而排除了那些无法产生有序结果的计划类型。

### 30.2.6. 栈深度不足

如果错误测试导致了在select infinite\_recurse()命令上的一次服务器崩溃，它意味着平台对进程栈尺寸的限制低于max\_stack\_depth参数所指定的值。这可以通过在一个更高的栈尺寸限制（对max\_stack\_depth的默认值，我们推荐 4 MB）下运行该服务器来修复。如果你不能这样做，一种可替代的方案是减小max\_stack\_depth的值。

在支持getrlimit()的平台上，服务器应该自动选择一个max\_stack\_depth的安全值。所以除非你已经手工覆盖了该设置，这类失败就是一个可报告的缺陷。

### 30.2.7. “随机”测试

随机测试脚本用来产生随机结果。在非常罕见的情况下，这会导致回归测试失败。输入：

```
diff results/random.out expected/random.out
```

应当产生一行或少数几行差异。你不需要担心，除非随机测试重复地失败。

### 30.2.8. 配置参数

当对一个现有安装运行测试时，某些非默认参数设置可能导致测试失败。例如，改变enable\_seqscan或enable\_indexscan等参数可能导致计划改变，然后影响使用EXPLAIN的测试的结果。

## 30.3. 变体比较文件

因为某些测试生来就会产生依赖环境的结果，我们提供了方法来指定替代的“预期”结果文件。每一个回归测试可以有多个比较文件来展示在不同平台上的可能结果。有两种独立的机制来决定为每一个测试使用哪个比较文件。

第一种机制允许为指定平台选择比较文件。这是一个映射文件src/test/regress/resultmap，它定义了为每一个平台使用哪个比较文件。要为一个特定平台消除虚假的测试“失败”，你可以首先选择或创建一个变体结果文件，然后在resultmap文件中增加一行。

在该映射文件中的每一行的形式为：

```
testname:output:platformpattern=comparisonfilename
```

测试名只是该特定回归测试模块的名称。输出值指定要检查哪个输出文件。对于标准回归测试，这总是out。该值对应于输出文件的文件扩展。平台模式是一个 Unix 工具expr风格的模式（即在开头带有一个隐式^锚的正则表达式）。它被与config.guess打印出的平台名称进行匹配。比较文件名是替补的结果比较文件的基础名。

例如：某些系统缺少工作`strtouf`函数，我们的解决方法在`float4`回归测试中会导致舍入误差。因此，我们提供一个变体比较文件`float4-misrounded-input.out`，其中包括了在这些系统上的期望结果。要在HP-UX 10平台上屏蔽这种虚假的“failure”消息，`resultmap`包括：

```
float4:out:hppa.*-hp-hpux10.*=float4-misrounded-input.out
```

这将触发`config.guess`的输出匹配 `hppa.*-hp-hpux10.*`的任何机器。`resultmap`中的其他行为为其他平台选择适用的变体比较文件。

第二种变体比较文件的选择机制更加自动：它简单地在多个提供的比较文件中采用“最佳匹配”。回归测试驱动器脚本对一个测试考虑两种标准比较文件，`testname.out`以及名为`testname_digit.out`的变体文件（其中`digit`是任何单一数字0-9）。如果任一这种文件是一个完全匹配，测试被认为是通过的。否则，产生最短区别的文件被用来创建失败报告（如果`resultmap`包括特定测试的一个项，那么基础`testname`是`resultmap`中给定的替补名称）。

例如，对于`char`测试，比较文件`char.out`包含在C和POSIX区域中期望的结果，而文件`char_1.out`包含在其他很多区域中的排序结果。

最佳匹配机制被设计为与区域依赖的结果协同工作，但是它可以被用在任何测试结果无法从平台名很容易地预测的情况中。这种机制的一个限制是测试驱动器不能说出哪个变体对当前环境是真正“正确的”，它将只是选择看起来工作得最好的变体。因此对你认为在所有上下文中具有同等合法性的变体结果使用这种机制才是最安全的。

## 30.4. TAP 测试

很多测试，特别是`src/bin`下面的客户端程序测试使用 Perl 的 TAP 工具并且用Perl测试程序`prove`运行。你可以通过 设置`make`变量`PROVE_FLAGS` 向`prove`传递命令行选项，例如：

```
make -C src/bin check PROVE_FLAGS='--timer'
```

详见`prove`的手册页。

`make`变量`PROVE_TESTS`可被用来定义一个空格分隔的列表，其中是调用`prove`来运行的指定测试子集的路径，这些测试子集将取代默认的`t/*.pl`，并且这些路径是相对于`Makefile`的。例如：

```
make check PROVE_TESTS='t/001_test1.pl t/003_test3.pl'
```

TAP测试需要 Perl 模块`IPC::Run`。这个模块可以从 CPAN 或者一个操作系统包得到。它们还需要为`LightDB`配置`--enable-tap-tests`选项。

一般来说，如果你执行`make installcheck`，TAP测试将测试之前安装的安装树中的可执行文件；或者如果你执行`make check`，将从当前源构建新的本地安装树。在这两种情况下，他们将初始化本地实例（数据目录），并在其中暂时运行服务器。其中一些测试运行多个服务器。因此，这些测试可能相当耗费资源。

重要的是要认识到TAP测试将启动测试服务器，即使你说`make installcheck`；这与传统的非TAP 测试基础架构不同，在这种情况下它期望使用已经运行的测试服务器。某些 `LightDB`子目录包含传统样式和 TAP 样式测试，这意味着`make installcheck`将产生来自临时服务器和已运行测试服务器的混合结果。

## 30.5. 测试覆盖检查

`LightDB` 源代码可以使用覆盖测试指令编译，因此可以检查哪些部分的代码被回归测试或任何其他测试套件所覆盖。当前使用 GCC 编译时支持该特性，并且它需要`gcov`和`lcov`程序。

一个典型的工作流程看起来是：

```
./configure --enable-coverage ... OTHER OPTIONS ...  
make  
make check # 或其他测试套件  
make coverage-html
```

然后将你的 HTML 浏览器指向coverage/index.html。

如果没有lcov或者更喜欢文本输出而不是HTML报告，还可以运行

```
make coverage
```

来取代make coverage-html，它将为每个与测试相关的源文件产生.gcov输出文件（make coverage和make coverage-html将覆盖彼此的文件，所以把它们混合在一起可能会导致混乱）。

要在测试运行之间重置执行计数，请运行：

```
make coverage-clean
```

---

## Chapter 31. 透明数据加密

TDE（透明数据加密）是将组成数据库集群的所有文件以加密格式（静态数据加密）安全地存储在磁盘上，然后在从磁盘读取块时对其进行解密。数据在内存中是未加密的。

TDE具有以下特点：

- 从应用程序的角度来看，加密是透明的。
- 它使用单个密钥来加密整个集群。

在启动期间，服务器可以访问初始化数据库的密钥，并通过特殊的配置参数提供加密密钥，该参数指定了实现特殊安全要求的自定义密钥设置命令。TDE在内部使用行业标准的128位AES-CTR加密算法来加密/解密磁盘上的数据块。

使用`lt_initdb`通过`-K`选项传递TDE密钥。在创建数据库实例之前，您必须编写一些代码以确保数据库可以在启动和实例创建期间读取密钥。注意，该密钥除数据库服务器外不可见。该密钥必须是32字节的十六进制字符串，否则在初始化实例时将报告错误：

```
fatal: encryption key is too short, should be a 32 character hex key
```

这是一个示例：

```
cat /somewhere/provide_key.sh
#!/bin/sh
echo 882fb7c12e80280fd664c69d2d636913
```

您需要的只是一个将密钥打印到标准输出的程序，并确保LightDB能够执行此程序：

```
chmod +x /somewhere/provide_key.sh
```

创建一个数据库实例：

```
lt_initdb -p 5432 -D ./data -K /somewhere/provide_key.sh
```

您可以使用`lt_controldata`确定当前实例是否已使用透明加密。在清单的末尾，您可以找到加密详细信息。

`lightdb_encryption_key_command`的GUC参数将存在于`lightdb.conf`配置文件中，确保在每次数据重新启动后将再次读取TDE密钥。

请注意，我们目前不支持对现有集群进行原地加密。您需要将其备份并重新加载到加密实例中，或使用逻辑复制在线执行迁移。

---

## Part IV. 客户端接口

这一部分描述和LightDB一起发布的客户端编程接口。这些章中的每一个都能被独立阅读。注意，还有很多用于客户端程序的其他编程接口是被独立发布的并且包含它们自己的文档（Appendix 列出了一些很流行的）。这部份的读者应该熟悉使用SQL命令来操纵和查询数据库（见Part I），以及熟悉接口所使用的编程语言。

---

---

## Table of Contents

32.	libpq - C 库	726
32.1.	数据库连接控制函数	726
32.1.1.	连接字符串	732
32.1.2.	参数关键词	734
32.2.	连接状态函数	740
32.3.	命令执行函数	745
32.3.1.	主要函数	745
32.3.2.	检索查询结果信息	752
32.3.3.	检索其他结果信息	756
32.3.4.	用于包含在 SQL 命令中的转移串	757
32.4.	异步命令处理	760
32.5.	一行一行地检索查询结果	763
32.6.	取消进行中的查询	764
32.7.	快速路径接口	765
32.8.	异步提示	766
32.9.	COPY命令相关的函数	767
32.9.1.	用于发送COPY数据的函数	767
32.9.2.	用于接收COPY数据的函数	768
32.9.3.	用于COPY的废弃函数	769
32.10.	控制函数	771
32.11.	杂项函数	772
32.12.	通知处理	775
32.13.	事件系统	776
32.13.1.	事件类型	776
32.13.2.	事件回调函数	778
32.13.3.	事件支持函数	779
32.13.4.	事件实例	780
32.14.	环境变量	782
32.15.	口令文件	783
32.16.	连接服务文件	784
32.17.	连接参数的 LDAP 查找	784
32.18.	SSL 支持	785
32.18.1.	服务器证书的客户验证	785
32.18.2.	客户端证书	786
32.18.3.	不同模式中提供的保护	787
32.18.4.	SSL 客户端文件使用	788
32.18.5.	SSL 库初始化	788
32.19.	在线程化程序中的行为	789
32.20.	编译 libpq 程序	789
32.21.	例子程序	790
33.	大对象	801
33.1.	简介	801
33.2.	实现特性	801
33.3.	客户端接口	801
33.3.1.	创建一个大对象	801
33.3.2.	导入一个大对象	802
33.3.3.	导出一个大对象	802
33.3.4.	打开一个现有的大对象	803
33.3.5.	向一个大对象写入数据	803
33.3.6.	从一个大对象读取数据	803
33.3.7.	在一个大对象中查找	803
33.3.8.	获取一个大对象的查找位置	804
33.3.9.	截断一个大对象	804
33.3.10.	关闭一个大对象描述符	805
33.3.11.	移除一个大对象	805

---

33.4.	服务器端函数	805
33.5.	例子程序	806
34.	ECPG — C中的嵌入式 SQL	812
34.1.	概念	812
34.2.	管理数据库连接	812
34.2.1.	连接到数据库服务器	812
34.2.2.	选择一个连接	814
34.2.3.	关闭一个连接	815
34.3.	运行 SQL 命令	815
34.3.1.	执行 SQL 语句	815
34.3.2.	使用游标	816
34.3.3.	管理事务	816
34.3.4.	预备语句	817
34.4.	使用主变量	818
34.4.1.	概述	818
34.4.2.	声明小节	818
34.4.3.	检索查询结果	819
34.4.4.	类型映射	820
34.4.5.	处理非简单 SQL 数据类型	826
34.4.6.	指示符	831
34.5.	动态 SQL	831
34.5.1.	执行没有结果集的语句	831
34.5.2.	执行一个有输入参数的语句	832
34.5.3.	执行一个有结果集的语句	832
34.5.4.	执行一个带输入输出参数的 Do Statement	833
34.6.	pgtypes 库	834
34.6.1.	字符串	834
34.6.2.	numeric类型	834
34.6.3.	日期类型	837
34.6.4.	时间戳类型	840
34.6.5.	区间类型	844
34.6.6.	decimal类型	845
34.6.7.	pgtypeslib 的 errno 值	845
34.6.8.	pgtypeslib 的特殊常量	846
34.7.	使用描述符区域	846
34.7.1.	命名 SQL 描述符区域	847
34.7.2.	SQLDA 描述符区域	849
34.8.	错误处理	859
34.8.1.	设置回调	859
34.8.2.	sqlca	860
34.8.3.	SQLSTATE 与 SQLCODE	862
34.9.	预处理器指令	865
34.9.1.	包括文件	865
34.9.2.	define 和 undef 指令	866
34.9.3.	ifdef, ifndef, elif, else, 和 endif 指令	867
34.10.	处理嵌入式 SQL 程序	867
34.11.	库函数	868
34.12.	大对象	869
34.13.	C++ 应用	870
34.13.1.	主变量的可见范围	871
34.13.2.	使用外部 C 模块的 C++ 应用开发	872
34.14.	嵌入式 SQL 命令	874
34.15.	Informix兼容模式	896
34.15.1.	附加类型	896
34.15.2.	附加的/缺少的 嵌入式 SQL 语句	897
34.15.3.	Informix-兼容的 SQLDA 描述符区域	897
34.15.4.	附加函数	900
34.15.5.	额外的常量	909

---

34.16.	Oracle 兼容模式 .....	910
34.17.	内部 .....	910
35.	信息模式 .....	913
35.1.	模式 .....	913
35.2.	数据类型 .....	913
35.3.	information_schema_catalog_name .....	914
35.4.	administrable_role_authorizations .....	914
35.5.	applicable_roles .....	914
35.6.	attributes .....	915
35.7.	character_sets .....	917
35.8.	check_constraint_routine_usage .....	917
35.9.	check_constraints .....	918
35.10.	collations .....	918
35.11.	collation_character_set_applicability .....	919
35.12.	column_column_usage .....	919
35.13.	column_domain_usage .....	919
35.14.	column_options .....	920
35.15.	column_privileges .....	920
35.16.	column_udt_usage .....	921
35.17.	columns .....	921
35.18.	constraint_column_usage .....	924
35.19.	constraint_table_usage .....	925
35.20.	data_type_privileges .....	925
35.21.	domain_constraints .....	926
35.22.	domain_udt_usage .....	926
35.23.	domains .....	927
35.24.	element_types .....	928
35.25.	enabled_roles .....	930
35.26.	foreign_data_wrapper_options .....	931
35.27.	foreign_data_wrappers .....	931
35.28.	foreign_server_options .....	931
35.29.	foreign_servers .....	932
35.30.	foreign_table_options .....	932
35.31.	foreign_tables .....	933
35.32.	key_column_usage .....	933
35.33.	parameters .....	934
35.34.	referential_constraints .....	935
35.35.	role_column_grants .....	936
35.36.	role_routine_grants .....	936
35.37.	role_table_grants .....	937
35.38.	role_udt_grants .....	938
35.39.	role_usage_grants .....	938
35.40.	routine_privileges .....	939
35.41.	routines .....	939
35.42.	schemata .....	943
35.43.	sequences .....	944
35.44.	sql_features .....	945
35.45.	sql_implementation_info .....	945
35.46.	sql_parts .....	946
35.47.	sql_sizing .....	946
35.48.	table_constraints .....	946
35.49.	table_privileges .....	947
35.50.	tables .....	948
35.51.	transforms .....	948
35.52.	triggered_update_columns .....	949
35.53.	triggers .....	950
35.54.	udt_privileges .....	951
35.55.	usage_privileges .....	951



35.56.	user_defined_types .....	952
35.57.	user_mapping_options .....	954
35.58.	user_mappings .....	954
35.59.	view_column_usage .....	954
35.60.	view_routine_usage .....	955
35.61.	view_table_usage .....	955
35.62.	views .....	956

---

## Chapter 32. libpq - C 库

libpq是应用程序员使用LightDB的C接口。libpq是一个库函数的集合，它们允许客户端程序传递查询给LightDB后端服务器并且接收这些查询的结果。

libpq也是很多其他LightDB应用接口的底层引擎，包括为 C++、Perl、Python、Tcl 和 ECPG编写的接口。如果你使用那些包，某些方面的libpq行为将会对你很重要。特别是，Section 32.14 Section 32.1和Section 32.1描述了任何使用libpq的应用的用户可见的行为。

在本章的末尾 (Section 32.2) 包括了一些短程序来展示如何编写使用libpq的应用。在源代码发布的src/test/examples目录中还有一些完整的libpq应用的例子。

使用libpq的客户端程序必须包括头文件libpq-fe.h并必须与libpq库链接在一起。

### 32.1. 数据库连接控制函数

下列函数会建立到一个LightDB后端服务器的连接。一个应用程序可以在一个时刻打开多个后端连接（原因之一就是为了解访问多个数据库）。每个连接用一个PGconn对象表示，它从函数PQconnectdb、PQconnectdbParams，或PQsetdbLogin得到。注意这些函数将总是返回一个非空的对象指针，除非正好没有内存来分配PGconn对象。在通过该连接对象发送查询之前，应该调用PQstatus函数来检查返回值以确定是否得到了一个成功的连接。

#### Warning

如果不可信用户能够访问一个没有采用安全方案使用模式的数据库，开始每个会话时应该从search\_path中移除公共可写的方案。我们可以把参数关键词options设置为-csearch\_path=。也可以在连接后发出PQexec(conn, "SELECT pg\_catalog.set\_config('search\_path', '', false)")。这种考虑并非专门针对libpq，它适用于每一种用来执行任意SQL命令的接口。

#### Warning

在 Unix 上，复制一个拥有打开 libpq 连接的进程可能导致不可以预料的结果，因为父进程和子进程会共享相同的套接字和操作系统资源。出于这个原因，我们不推荐这样的用法，尽管从子进程执行一个exec来载入新的可执行代码是安全的。

#### PQconnectdbParams

开启一个到数据库服务器的新连接。

```
PGconn *PQconnectdbParams(const char * const *keywords,  
                          const char * const *values,  
                          int expand_dbname);
```

这个函数使用从两个以NULL结尾的数组中取得的参数打开一个新的数据库连接。第一个数组keywords被定义为一个字符串数组，每一个元素是一个关键词。第二个数组values给出了每个关键词的值。和下面的PQsetdbLogin不同，可以在不改变函数签名的情况下扩展参数集合，因此使用这个函数（或者与之相似的非阻塞的PQconnectStartParams和PQconnectPoll）对于新应用编程要更好。

当前能被识别的参数关键词被列举在Section 32.1.中。

当`expand_dbname`为非零时，`dbname`关键词的值被允许识别为一个连接字符串。只有`dbname`的第一次出现会按这种方式扩展，任何后续`dbname`值会被当做一个普通数据库名处理。有关可能的连接字符串格式的详情可见Section 32.1.1

被传递的数组可以为空，这样就会使用所有默认参数。也可以只包含一个或几个参数设置。两个参数数组应该在长度上匹配。对于参数数组的处理将会停止于`keywords`数组中第一个非-NULL元素。

如果任何一个参数是NULL或者一个空字符串，那么将会检查相应的环境变量（见Section 32.14）。如果该环境变量也没有被设置，那么将使用内建的默认值。

通常，关键词的处理是从这些数组的头部开始并且以索引顺序进行。这样做的效果就是，当关键词有重复时，只会保留最后一个被处理的值。因此，通过小心地放置关键词`dbname`，可以决定什么会被一个`conninfo`字符串所重载，以及什么不会被重载。

### PQconnectdb

开启一个到数据库服务器的新连接。

```
PGconn *PQconnectdb(const char *conninfo);
```

这个函数使用从字符串`conninfo`中得到的参数开启一个新的数据库连接。

被传递的字符串可以为空，这样将会使用所有的默认参数。也可以包含由空格分隔的一个或多个参数设置，还可以包含一个URI。详见Section 32.1.1

### PQsetdbLogin

开启一个到数据库服务器的新连接。

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

这是`PQconnectdb`的带有固定参数集合的前辈。它具有相同的功能，不过其中缺失的参数将总是采用默认值。对任意一个固定参数写NULL或一个空字符串将会使它采用默认值。

如果`dbName`包含一个`=`符号或者具有一个合法的连接URI前缀，它会被当作一个`conninfo`字符串，就好像它已经被传递给了`PQconnectdb`，并且剩余的参数则被应用为指定给`PQconnectdbParams`。

### PQsetdb

开启一个到数据库服务器的新连接。

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

这是一个调用`PQsetdbLogin`的宏，其中为`login`和`pwd`参数使用空指针。提供它是为了向后兼容非常老的程序。

PQconnectStartParams  
 PQconnectStart  
 PQconnectPoll

以非阻塞的方式建立一个到数据库服务器的连接。

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                             int expand_dbname);
```

```
PGconn *PQconnectStart(const char *conninfo);
```

```
PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

这三个函数被用来开启一个到数据库服务器的连接，这样你的应用的执行线程不会因为远程的I/O而被阻塞。这种方法的重点在于等待 I/O 完成可能在应用的主循环中发生，而不是在PQconnectdbParams 或 PQconnectdb中，并且因此应用能够把这种操作和其他动作并行处理。

在PQconnectStartParams中，数据库连接使用从keywords和values数组中取得的参数创建，并且被expand\_dbname控制，这和之前描述的PQconnectdbParams相同。

在PQconnectStart中，数据库连接使用从字符串conninfo中取得的参数创建，这和之前描述的PQconnectdb相同。

只要满足一些限制，PQconnectStartParams或PQconnectStart或PQconnectPoll都不会阻塞：

- hostaddr和host参数必须被合适地使用，以防止做DNS查询。详见Section 32.1. 中这些参数的文档。
- 如果你调用PQtrace，确保你追踪的该流对象不会阻塞。
- 如后文所述，你必须要确保在调用PQconnectPoll之前，套接字处于合适的状态。

要开始无阻塞的连接请求，可调用PQconnectStart或者PQconnectStartParams。如果结果为空，则libpq无法分配一个新的PGconn结构。否则，一个有效的PGconn指针会被返回（不过还没有表示一个到数据库的有效连接）。接下来调用PQstatus(conn)。如果结果是CONNECTION\_BAD，则连接尝试已经失败，通常是因为有无效的连接参数。

如果PQconnectStart或PQconnectStartParams成功，下一个阶段是轮询libpq，这样它能够继续进行连接序列。使用PQsocket(conn)来获得该数据库连接底层的套接字描述符（警告：不要假定在PQconnectPoll调用之间套接字会保持相同）。这样循环：如果PQconnectPoll(conn)上一次返回PGRES\_POLLING\_READING，等到该套接字准备好读取（按照select()、poll()或类似的系统函数所指示的）。则再次调用PQconnectPoll(conn)。反之，如果PQconnectPoll(conn)上一次返回PGRES\_POLLING\_WRITING，等到该套接字准备好写入，则再次调用PQconnectPoll(conn)。在第一次迭代时，即如果你还没有调用PQconnectPoll，行为就像是它上次返回了PGRES\_POLLING\_WRITING。持续这个循环直到PQconnectPoll(conn)返回PGRES\_POLLING\_FAILED指示连接过程已经失败，或者返回PGRES\_POLLING\_OK指示连接已经被成功地建立。

在连接期间的任意时刻，该连接的状态可以通过调用PQstatus来检查。如果这个调用返回CONNECTION\_BAD，那么连接过程已经失败。如果该调用返回CONNECTION\_OK，则该连接已经准备好。如前所述，这些状态同样都可以从PQconnectPoll的返回值检测。在一个异步连接过程中（也只有在这个过程中）也可能出现其他状态。这些状态指示该连接过程的当前阶段，并且可能有助于为用户提供反馈。这些状态是：

CONNECTION\_STARTED

等待连接被建立。

CONNECTION\_MADE

连接 OK，等待发送。

CONNECTION\_AWAITING\_RESPONSE

等待来自服务器的一个回应。

CONNECTION\_AUTH\_OK

收到认证，等待后端启动结束。

CONNECTION\_SSL\_STARTUP

协商 SSL 加密。

CONNECTION\_SETENV

协商环境驱动的参数设置。

CONNECTION\_CHECK\_WRITABLE

检查连接是否能够处理写事务。

CONNECTION\_CONSUME

消费连接上的任何剩下的响应消息。

注意，尽管这些常数将被保留（为了维护兼容性），一个应用永远不应该依赖这些状态按照特定顺序出现，或者根本就不依赖它们，或者不依赖状态总是这些文档中所说的值。一个应用可能做些这样的事情：

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connecting...";
        break;

    case CONNECTION_MADE:
        feedback = "Connected to server...";
        break;

    .
    .
    .
    default:
        feedback = "Connecting...";
}
```

在使用PQconnectPoll时，连接参数connect\_timeout会被忽略：判断是否超时是应用的责任。否则，PQconnectStart后面跟着PQconnectPoll循环等效于PQconnectdb。

注意当PQconnectStart或PQconnectStartParams返回一个非空的指针时，你必须在用完它之后调用PQfinish来处理那些结构和任何相关的内存块。即使连接尝试失败或被放弃时也必须完成这些工作。

PQconndefaults

返回默认连接选项。

```
PQconninfoOption *PQconndefaults(void);
```

```

typedef struct
{
    char    *keyword; /* 该选项的关键词 */
    char    *envvar;  /* 依赖的环境变量名 */
    char    *compiled; /* 依赖的内建默认值 */
    char    *val;     /* 选项的当前值, 或者 NULL */
    char    *label;   /* 连接对话框中域的标签 */
    char    *dispchar; /* 指示如何在一个连接对话框中显示这个域。值是:
                        ""          显示输入的值
                        "*"         口令域 - 隐藏值
                        "D"        调试选项 - 默认不显示 */
    int     dispsize; /* 用于对话框的以字符计的域尺寸 */
} PQconninfoOption;

```

返回一个连接选项数组。这可以用来确定用于连接服务器的所有可能的PQconnectdb选项和它们的当前缺省值。返回值指向一个PQconninfoOption结构的数组，该数组以一个包含空keyword指针的条目结束。如果无法分配内存，则返回该空指针。注意当前缺省值（val域）将依赖于环境变量和其他上下文。一个缺失或者无效的服务文件将会被无声地忽略掉。调用者必须把连接选项当作只读对待。

在处理完选项数组后，把它交给PQconninfoFree释放。如果没有这么做，每次调用PQconndefaults都会导致一小部分内存泄漏。

#### PQconninfo

返回被一个活动连接使用的连接选项。

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

返回一个连接选项数组。这可以用来确定用于连接服务器的所有可能的PQconnectdb选项和它们的当前缺省值。返回值指向一个PQconninfoOption结构的数组，该数组以一个包含空keyword指针的条目结束。上述所有对于PQconndefaults的注解也适用于PQconninfo的结果。

#### PQconninfoParse

返回从提供的连接字符串中解析到的连接选项。

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

解析一个连接字符串并且将结果选项作为一个数组返回，或者在连接字符串有问题时返回NULL。这个函数可以用来抽取所提供的连接字符串中的PQconnectdb选项。返回值指向一个PQconninfoOption结构的数组，该数组以一个包含空keyword指针的条目结束。

所有合法选项将出现在结果数组中，但是任何在连接字符串中没有出现的选项的PQconninfoOption的val会被设置为NULL，默认值不会被插入。

如果errmsg不是NULL，那么成功时\*errmsg会被设置为NULL，否则设置为被malloc过的错误字符串以说明该问题（也可以将\*errmsg设置为NULL并且函数返回NULL，这表示一种内存耗尽的情况）。

在处理完选项数组后，把它交给PQconninfoFree释放。如果没有这么做，每次调用PQconninfoParse都会导致一小部分内存泄漏。反过来，如果发生一个错误并且errmsg不是NULL，确保使用PQfreemem释放错误字符串。

#### PQfinish

关闭与服务器的连接。同时释放PGconn对象使用的内存。

```
void PQfinish(PGconn *conn);
```

注意，即使与服务器的连接尝试失败（由PQstatus指示），应用也应当调用PQfinish来释放PGconn对象使用的内存。不能在调用PQfinish之后再使用PGconn指针。

### PQreset

重置与服务器的通讯通道。

```
void PQreset(PGconn *conn);
```

此函数将关闭与服务器的连接，并且使用所有之前使用过的参数尝试重新建立与同一个服务器的连接。这可能有助于在工作连接丢失后的错误恢复。

### PQresetStart

### PQresetPoll

以非阻塞方式重置与服务器的通讯通道。

```
int PQresetStart(PGconn *conn);
```

```
PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

这些函数将关闭与服务器的连接，并且使用所有之前使用过的参数尝试重新建立与同一个服务器的连接。这可能有助于在工作连接丢失后的错误恢复。它们和上面的PQreset的不同在于它们工作在非阻塞方式。这些函数受到PQconnectStartParams、PQconnectStart和PQconnectPoll相同的限制。

要发起一次连接重置，调用PQresetStart。如果它返回 0，那么重置失败。如果返回 1，用与使用PQresetPoll建立连接的相同方法使用PQconnectPoll重置连接。

### PQpingParams

PQpingParams报告服务器的状态。它接受与PQconnectdbParams相同的连接参数，如上所述。获得服务器状态不需要提供正确的用户名、口令或数据库名。不过，如果提供了不正确的值，服务器将记录一次失败的连接尝试。

```
PGPing PQpingParams(const char * const *keywords,  
                   const char * const *values,  
                   int expand_dbname);
```

该函数返回下列值之一：

### PQPING\_OK

服务器正在运行，并且看起来可以接受连接。

### PQPING\_REJECT

服务器正在运行，但是处于一种不允许连接的状态（启动、关闭或崩溃恢复）。

### PQPING\_NO\_RESPONSE

无法联系到服务器。这可能表示服务器没有运行，或者给定的连接参数中有些错误（例如，错误的端口号），或者有一个网络连接问题（例如，一个防火墙阻断了连接请求）。

## PQPING\_NO\_ATTEMPT

没有尝试联系服务器，因为提供的参数显然不正确，或者有一些客户端问题（例如，内存用完）。

## PQping

PQping报告服务器的状态。它接受与PQconnectdb相同的连接参数。获得服务器状态不需要提供正确的用户名、口令或数据库名。不过，如果提供了不正确的值，服务器将记录一次失败的连接尝试。

```
PGPing PQping(const char *conninfo);
```

返回值和PQpingParams相同。

## PQsetSSLKeyPassHook\_OpenSSL

PQsetSSLKeyPassHook\_OpenSSL 令一个应用取代 libpq的 default handling of encrypted client certificate key files，使用sslpassword 或交互提示方式。

```
void PQsetSSLKeyPassHook_OpenSSL(PQsslKeyPassHook_OpenSSL_type hook);
```

应用程序传递一个指针到带签名的回调函数：

```
int callback_fn(char *buf, int size, PGconn *conn);
```

libpq将调用而不是（instead of）它的默认PQdefaultSSLKeyPassHook\_OpenSSL处理程序。回调函数应该确定密钥的密码并将其复制到大小为size的结果缓冲区buf中。buf中的字符串必须以空终止符(null-terminated)。回调函数必须返回存储在buf中的密码的长度，不包括空终止符。如果失败，回调函数应该设置buf[0] = '\0' 并返回0。参见libpq源代码中的PQdefaultSSLKeyPassHook\_OpenSSL以获得示例。

如果用户指定了一个显式的密钥位置，那么当调用回调时，它的路径将在conn->sslkey中。如果使用默认密钥路径，则这将为空。对于作为引擎说明符的密钥，由引擎实现决定它们是使用OpenSSL密码回调还是定义自己的处理方式。

应用回调可能会选择将未处理的案例委派给PQdefaultSSLKeyPassHook\_OpenSSL，或者先调用它，并且如果它返回0时再尝试其他方法，或者完全覆盖它。

除了例外、longjmp(...)等，回调务必不可逃避正常的流控制。它必须正常返回。

## PQgetSSLKeyPassHook\_OpenSSL

PQgetSSLKeyPassHook\_OpenSSL返回当前客户端证书密钥密码钩子，如果没有设置则返回NULL。

```
PQsslKeyPassHook_OpenSSL_type PQgetSSLKeyPassHook_OpenSSL(void);
```

## 32.1.1. 连接字符串

几个libpq函数会解析一个用户指定的字符串来获得连接参数。这些字符串有两种被接受的格式：纯关键词 = 值字符串以及URI。URI通常遵循RFC 3986<sup>1</sup>，除非像下文进一步描述的那样允许多主机连接字符串。

<sup>1</sup> <https://tools.ietf.org/html/rfc3986>



### 32.1.1.1. 关键词/值连接字符串

在第一种格式中，每一个参数设置的形式都是关键词 = 值。等号周围的空白是可选的。要写一个空值或一个包含空白的值，将它用单引号包围，例如关键词 = 'a value'。值里面的单引号和反斜线必须用一个反斜线转义，即\'和\\。

例子：

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

能被识别的参数关键词在Section 32.1.2中列出。

### 32.1.1.2. 连接 URI

一个连接URI的一般形式是：

```
postgresql://[userspec@[hostspec] [/dbname] [?paramspec]
```

其中 userspec 是：

```
user[:password]
```

而 hostspec 是：

```
[host][:port][,...]
```

而 paramspec 是：

```
name=value[&...]
```

URI模式标志符可以是postgresql://或postgres://。每一个URI部分都是可选的。下列例子展示了合法的URI语法：

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
postgresql://host1:123,host2:456/somedb?
target_session_attrs=any&application_name=myapp
```

URI的层次部分的组件可以作为参数给出。例如：

```
postgresql:///mydb?host=localhost&port=5433
```

所有命名参数必须与 Section 32.1.2中列出的关键字匹配，但为了与 JDBC 连接 URI 兼容，ssl=true 的实例被翻译为 sslmode=require。

URI连接需要和percent-encoding<sup>2</sup>一同编码，如果它包括包括在任意部分带有特殊含义的符号。这是一个示例，等号(=)被%3D和带%20的空格所替换：

```
postgresql://user@localhost:5433/mydb?options=-c%20synchronous_commit%3Doff
```

<sup>2</sup> <https://tools.ietf.org/html/rfc3986#section-2.1>

主机部分可能是主机名或一个 IP 地址。要指定一个 IPv6 主机地址，将它封闭在方括号中：

```
postgresql://[2001:db8::1234]/database
```

主机组件会被按照参数host对应的描述来解释。特别地，如果主机部分是空或看起来像一个绝对路径名称，将使用一个 Unix 域套接字连接，否则将启动一个 TCP/IP 连接。不过要注意，斜线是 URI 层次部分中的一个保留字符。因此，要指定一个非标准的 Unix 域套接字目录，要么忽略 URI 中的主机说明并且指定该主机为一个参数，要么在 URI 的主机部分用百分号编码路径：

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

可以在一个URI中指定多个主机，每一个都有一个可选的端口。一个形式为postgresql://host1:port1,host2:port2,host3:port3/的URI等效于host=host1,host2,host3port=port1,port2,port3形式的连接字符串。每一个主机都将被尝试，直到成功地建立一个连接。

### 32.1.1.3. 指定多个主机

可以指定多个要连接的主机，这样它们会按给定的顺序被尝试。在键/值格式中，host、hostaddr和port选项都接受逗号分隔的值列表。在指定的每一个选项都必须给出相同数量的元素，这样第一个hostaddr对应于第一个主机名，第二个hostaddr对应于第二个主机名，以此类推。不过，如果仅指定一个port，它将被应用于所有的主机。

在连接URI格式中，在URI的host部分我们可以列出多个由逗号分隔的host:port对。

不管是哪一种格式，单一的主机名可以被翻译成多个网络地址。常见的例子是一个主机同时具有IPv4和IPv6地址。

当多个主机被指定时或者单个主机名被翻译成多个地址时，所有的主机和地址都将按照顺序被尝试，直至遇到一个成功的。如果没有主机可以到达，则连接失败。如果成功地建立一个连接但是认证失败，也不会尝试列表中剩下的主机。

如果使用了口令文件，可以为不同的主机使用不同的口令。所有其他连接选项对列表中的每一台主机都是相同的，例如不能为不同的主机指定不同的用户名。

## 32.1.2. 参数关键词

当前被识别的参数关键词是：

host

要连接的主机名。如果一个主机名看起来像一个绝对路径名称，则表示一个 Unix 域通信而不是 TCP/IP 通信，其值是存储套接字文件的目录名。（在Unix上，绝对路径名称以斜杠开始。在Windows上，以驱动器符号开始的路径名称也可以被识别。）当host没有指定或者为空时的默认行为是连接到一个/tmp（或者LightDB编译时指定的任何套接字目录）中的 Unix 域套接字。在Windows和没有 Unix 域套接字的机器上，默认是连接到localhost。

也接受由逗号分隔的主机名列表，这种情况下将依次尝试列表中的主机名，列表中的空项会选择上述的默认行为。详情请参考Section 32.1.1.3

hostaddr

要连接的机器的数字 IP 地址。它应该是标准的 IPv4 地址格式，例如172.28.40.9。如果你的机器支持 IPv6，你也可以使用那些地址。当为这个参数指定一个非空字符串时，

总是会使用 TCP/IP 通信。如果这些参数没有被指定，会查询host的值并找出相应的IP地址；或者，如果host指定了一个IP地址，那个值将被直接使用。

使用hostaddr允许应用能避免一次主机名查找，这对于具有时间约束的应用可能非常重要。不过，GSSAPI 或 SSPI 认证方法以及verify-full SSL 证书验证还是要求一个主机名。使用的是下列规则：

- 如果指定了host但没有hostaddr，则会发生主机名查找（在使用PQconnectPoll时，PQconnectPoll第一次考虑这个主机名时会发生查找，可能会导致PQconnectPoll阻塞可观的时间）。
- 如果hostaddr被指定且没有host，hostaddr的值给出了服务器的网络地址。如果认证方法要求一个主机名则连接尝试将会失败。
- 如果host和hostaddr都被指定，hostaddr的值给出服务器的网络地址。host的值将被忽略，除非认证方法要求它，在那种情况下它将被用作主机名。注意如果host不是网络地址hostaddr上的服务器名，认证很可能会失败。此外，当host和hostaddr都被指定时，host被用来在口令文件中标识该连接（见Section 32.15）。

逗号分隔的hostaddr值列表也被接受，这种情况下将依次尝试列表中的主机。列表中的空项会导致对应的主机名被使用，或者如果对应的主机名也为空时使用默认主机名。详情请参考Section 32.1.1.3

如果没有一个主机名或主机地址，libpq将尝试使用一个本地 Unix 域套接字连接，或者在Windows和没有 Unix 域套接字的机器上尝试连接到localhost。

#### port

在服务器主机上要连接的端口号，或者用于 Unix 域连接的套接字文件名扩展。如果在host或hostaddr参数中给出了多个主机，这个参数可以指定一个逗号分隔的端口列表，列表长度与主机列表相同，或者这个参数可以指定一个单一的端口号用于所有的主机。空字符串或者逗号分隔列表中的空项指定的是LightDB编译时建立的默认端口号。

#### dbname

数据库名。默认和用户名相同。在一般的环境下，会为扩展格式检查该值，详见Section 32.1.1

#### user

LightDB 要作为哪个用户连接。默认与运行着该应用的用户的操作系统名相同。

#### password

服务器要求口令认证时要使用的口令。

#### passfile

指定用于存放口令的文件名（见Section 32.15）。默认是~/.pgpass，或者是Microsoft Windows上的%APPDATA%\postgresql\pgpass.conf（如果该文件不存在也不会报错）。

#### channel\_binding

这个选项控制客户端对通道绑定的使用。设置require意味着连接必须使用通道绑定，prefer意味着客户端将在可用时选择通道绑定，而disable阻止使用通道绑定。默认为prefer，如果LightDB编译时带有支持SSL；否则默认为disable。

通道绑定是服务器向客户端验证其自身的方法。它只支持使用SCRAM认证方法与LightDB 11或更高版本服务器的SSL连接。

`connect_timeout`

连接时的最大等待时间，以秒为单位（写作一个十进制整数，例如10）。零、负值或者不指定表示无限等待。允许的最小超时是2秒，因此值1会被解释为2。这个超时单独应用于每个主机名或者IP地址。例如，如果指定两个主机并且`connect_timeout`为5，每个主机都会在5秒内没有建立起连接的情况下超时，因此花费在等待连接上的总时间可能高达10秒。

`client_encoding`

为连接设置`client_encoding`配置参数。除了被相应服务器选项所接受的值，你还能使用`auto`从客户端的当前区域（Unix 系统上的`LC_CTYPE`环境变量）决定正确的编码。

`options`

指定在连接开始时发送给服务器的命令行选项。例如，设置这个参数为`-c geqo=off`会把会话的`geqo`参数值设置为`off`。这个字符串中的空格被认为是命令行参数的分隔符，除非用一个反斜线（`\`）对它转义，用`\\`可以表示一个字面意义上的反斜线。可用选项的详细讨论请参考Chapter 17

`application_name`

为`application_name`配置参数指定一个值。

`fallback_application_name`

为`application_name`配置参数指定一个后补值。如果通过一个连接参数或`PGAPPNAME`环境变量没有为`application_name`给定一个值，将使用这个值。在希望设置一个默认应用名但不希望它被用户覆盖的一般工具程序中指定一个后补值很有用。

`keepalives`

控制是否使用客户端的 TCP 保持存活机制。默认值是 1，表示打开。但是如果不要保持存活机制，你可以改成 0 表示关闭。对于通过一个 Unix 域套接字建立连接会忽略这个参数。

`keepalives_idle`

控制非活动多少秒之后 TCP 应该向服务器发送一个存活消息。零值表示使用系统默认值。对于一个通过 Unix 域套接字建立连接将忽略这个参数，如果保持存活机制被禁用也将忽略这个参数。它只被`TCP_KEEPIDLE`或等效的套接字选项可用的系统以及 Windows支持，在其他系统上，它没有效果。

`keepalives_interval`

控制一个 TCP 存活消息没有被服务器认可多少秒之后应该被重传。零值表示使用系统默认值。对于一个通过 Unix 域套接字建立连接将忽略这个参数，如果保持存活机制被禁用也将忽略这个参数。它只被`TCP_KEEPINTVL`或等效的套接字选项可用的系统以及 Windows支持，在其他系统上，它没有效果。

`keepalives_count`

控制该客户端到服务器的连接被认为死亡之前可以丢失的 TCP 存活消息数量。零值表示使用系统默认值。对于一个通过 Unix 域套接字建立连接将忽略这个参数，如果保持存活机制被禁用也将忽略这个参数。它只被`TCP_KEEPCNT`或等效的套接字选项可用的系统以及 Windows支持，在其他系统上，它没有效果。

`tcp_user_timeout`

在强制关闭连接之前，控制传输数据的毫秒数可能保持未确认。 值为零使用系统默认值。对于通过 Unix 域套接字进行的连接，将忽略此参数。 它仅在`TCP_USER_TIMEOUT`可用的系统上支持；对其他系统，它没有任何影响。

tty

被忽略（之前，这指定向哪里发送服务器调试输出）。

replication

这个选项决定是否该连接应该使用复制协议而不是普通协议。这是LightDB的复制连接以及lt\_basebackup之类的工具在内部使用的协议，但也可以被第三方应用使用。关于复制协议的介绍可以参考Section 49.4

支持下列值，大小写无关：

true, on, yes, 1

连接进入到物理复制模式。

database

连接进入到逻辑复制模式，连接到dbname参数中指定的数据库。

false, off, no, 0

该连接是一个常规连接，这是默认行为。

在物理或者逻辑复制模式中，仅能使用简单查询协议。

gssencmode

此选项确定是否与服务器协商安全GSS TCP/IP 连接是否或具有什么优先级。有三种模式：

disable

仅尝试非GSSAPI加密的连接。

prefer (default)

如果存在GSSAPI凭据（即凭据缓存中），先尝试GSSAPI加密连接；如果失败或没有凭据，则尝试非GSSAPI加密连接。当LightDB已使用GSSAPI支持编译时，这是默认值。

require

仅尝试GSSAPI加密连接。

对于Unix 域套接字通信，gssencmode被忽略。如果LightDB是在没有 GSSAPI 支持的情况下编译的，使用require选项将导致错误，而prefer将被接受，但libpq实际上不会尝试GSSAPI加密连接。

sslmode

这个选项决定一个SSL TCP/IP连接是否将与服务器协商，或者决定以何种优先级协商。有六种模式：

disable

只尝试非SSL连接

allow

首先尝试非SSL连接，如果失败再尝试SSL连接

prefer (默认)

首先尝试SSL连接，如果失败再尝试非SSL连接

**require**

只尝试SSL连接。如果存在一个根 CA 文件，以verify-ca被指定的相同方式验证该证书

**verify-ca**

只尝试SSL连接，并且验证服务器证书是由一个可信的证书机构颁发的（CA）

**verify-full**

只尝试SSL连接，验证服务器证书是由一个可信的CA颁发并且请求的服务器主机名匹配证书中的主机名

这些选项如何工作的详细描述见Section 32.18

对于 Unix 域套接字通信，sslmode会被忽略。如果LightDB被编译为不带 SSL 支持，使用选项require、verify-ca或verify-full将导致错误，而选项allow和prefer将会被接受但是libpq将不会真正尝试SSL连接。

请注意，如果可能使用 GSSAPI 加密，则优先使用它而不考虑 sslmode 的值。如果要在具有工作的 GSSAPI 基础设施（例如 Kerberos 服务器）的环境中强制使用 SSL 加密，请将 gssencmode 设置为 disable。

**requiressl**

为了支持sslmode模式，这个选项已被废弃。

如果设置为 1，则要求一个到服务器的SSL连接（这等效于sslmode require）。如果服务器不接受SSL连接，libpq则将拒绝连接。如果设置为 0（默认），libpq将与该服务器协商连接类型（等效于sslmode prefer）。只有LightDB被编译为带有 SSL 支持，这个选项才可用。

**sslcompression**

如果设置为1，通过SSL连接发送的数据将被压缩。如果设置为0，压缩将被禁用。默认值为0。如果建立的连接没有使用SSL，则这个参数会被忽略。

现如今SSL压缩被认为是不安全的，因此已经不再推荐使用。OpenSSL 1.1.0默认禁用压缩，并且很多操作系统发行版在前面的版本中也将其禁用，因此如果服务器不接受压缩，将这个参数设置为on不会有任何效果。

如果安全性不是主要考虑的问题，在网络是瓶颈的情况下，压缩能够改进吞吐量。如果CPU性能是受限的因素，禁用压缩能提高响应时间和吞吐量。

**sslcert**

这个参数指定客户端 SSL 证书的文件名，它替换默认的~/.postgresql/postgresql.crt。如果没有建立 SSL 连接，这个参数会被忽略。

**sslkey**

这个参数指定用于客户端证书的密钥位置。它能指定一个会被用来替代默认的~/.postgresql/postgresql.key的文件名，或者它能够指定一个从外部“引擎”（引擎是OpenSSL的可载入模块）得到的密钥。一个外部引擎说明应该由一个冒号分隔的引擎名称以及一个引擎相关的关键标识符组成。如果没有建立 SSL 连接，这个参数会被忽略。

**sslpassword**

此参数指定在sslkey中指定的密钥的密码，在交互式密码短语输入不实用时，允许客户端证书私钥以加密形式存储在磁盘上。

指定该参数为任何非空值，阻止Enter PEM pass phrase: 提示OpenSSL将默认发出，在向libpq提供加密的客户端证书密钥时。

如果密钥未加密则忽略此参数。该参数对OpenSSL引擎指定的密钥没有影响，除非引擎使用OpenSSL密码回调机制进行提示。

没有与此选项相等的环境变量，也没有在.pgpass中查找它的工具。它可以在服务文件连接定义中使用。有更复杂用途的用户应该考虑使用openssl引擎和工具，例如PKCS#11或USB加密卸载设备。

#### sslrootcert

这个参数指定一个包含 SSL 证书机构 (CA) 证书的文件名称。如果该文件存在，服务器的证书将被验证是由这些机构之一签发。默认值是`~/postgresql/root.crt`。

#### sslcr1

这个参数指定 SSL 证书撤销列表 (CRL) 的文件名。列在这个文件中的证书如果存在，在尝试认证该服务器证书时会被拒绝。默认值是`~/postgresql/root.crl`。

#### requirepeer

这个参数指定服务器的操作系统用户，例如`requirepeer=postgres`。当建立一个 Unix 域套接字连接时，如果设置了这个参数，客户端在连接开始时检查服务器进程是运行在指定的用户名之下。如果发现不是，该连接会被一个错误中断。这个参数能被用来提供与 TCP/IP 连接上 SSL 证书相似的服务器认证（注意，如果 Unix 域套接字在/tmp或另一个公共可写的位置，任何用户能开始一个在那里监听的服务器。使用这个参数来保证你连接的是一个由可信用户运行的服务器）。这个选项只在实现了peer认证方法的平台上受支持，见Section 18.9

#### ssl\_min\_protocol\_version

此参数规定了允许连接的最小SSL/TLS协议版本。有效值为TLSv1, TLSv1.1, TLSv1.2 和 TLSv1.3。所支持的协议取决于所使用的OpenSSL版本，旧版本不支持最新的协议版本。如果没有指定，则默认为TLSv1.2，在撰写本文时的行业最佳实践。

#### ssl\_max\_protocol\_version

此参数指定允许连接的最大SSL/TLS协议版本。有效值为TLSv1, TLSv1.1, TLSv1.2 和 TLSv1.3。所支持的协议取决于所使用的OpenSSL版本，旧版本不支持最新的协议版本。如果未设置，这个参数会被忽略，并且连接将使用后端定义的最大绑定。设置最大协议版本主要用于测试或某些组件在使用新协议时出现问题。

#### krbsrvname

当用 GSSAPI 认证时，要使用的 Kerberos 服务名。为了让 Kerberos 认证成功，这必须匹配在服务器配置中指定的服务名（另见Section 18.6）。

#### gsslib

用于GSSAPI身份验证的GSS库。目前，除了在包含GSSAPI和SSPI支持的Windows构建中，这一点被忽略了。在这种情况下，将此设置为gssapi使 libpq 使用 GSSAPI 库进行身份验证，而不是默认 SSPI。

#### service

用于附加参数的服务名。它指定保持附加连接参数的pg\_service.conf中的一个服务名。这允许应用只指定一个服务名，这样连接参数能被集中维护。见Section 32.16

#### target\_session\_attrs

如果这个参数被设置为read-write，只有默认接受读写事务的连接才是可接受的。在任何成功的连接上将发出查询SHOW transaction\_read\_only，如果它返回on则连接将被关

闭。如果在连接字符串中指定了多个主机，只要连接尝试失败就会尝试剩余的服务器。这个参数的默认值any认为所有连接都可接受。

## 32.2. 连接状态函数

这些函数可以被用来询问一个已有数据库连接对象的状态。

### Tip

libpq应用程序员应该小心地维护PGconn抽象。使用下面描述的访问函数来理解PGconn的内容。我们不推荐使用libpq-int.h引用内部的PGconn域，因为它们可能在未来改变。

下列函数返回一个连接所建立的参数值。这些值在连接的生命期中是固定的。如果使用的是多主机连接字符串，如果使用同一个PGconn对象建立新连接，PQhost, PQport, 和 PQpass可能会改变。其他值在PGconn对象的一生中都是固定的。

#### PQdb

返回该连接的数据库名。

```
char *PQdb(const PGconn *conn);
```

#### PQuser

返回该连接的用户名。

```
char *PQuser(const PGconn *conn);
```

#### PQpass

返回该连接的口令。

```
char *PQpass(const PGconn *conn);
```

PQpass将返回连接参数中指定的口令，如果连接参数中没有口令并且能从口令文件中得到口令，则它将返回得到的口令。在后一种情况中，如果连接参数中指定了多个主机，在连接被建立之前都不能依赖PQpass的结果。连接的状态可以用函数PQstatus检查。

#### PQhost

返回活跃连接的服务器主机名。可能是主机名、IP 地址或者一个目录路径（如果通过 Unix 套接字连接，路径的情况很容易区分，因为路径总是一个绝对路径，以/开始）。

```
char *PQhost(const PGconn *conn);
```

如果连接参数同时指定了host和hostaddr，则PQhost将返回host信息。如果仅指定了hostaddr，则返回它。如果在连接参数中指定了多个主机，PQhost返回实际连接到的主机。

如果conn参数是NULL，则PQhost返回NULL。否则，如果有一个错误产生主机信息（或许是连接没有被完全建立或者有什么错误），它会返回一个空字符串。

如果在连接参数中指定了多个主机，则在连接建立之前都不能依赖于PQhost的结果。连接的状态可以用函数PQstatus检查。



### PQhostaddr

返回活动连接的服务器 IP 地址。这可以是主机名解析出的地址，也可以是通过hostaddr参数提供的 IP 地址。

```
char *PQhostaddr(const PGconn *conn);
```

如果conn 参数为 NULL则 PQhostaddr 返回 NULL。否则，如果生成主机信息时出现错误（如果连接尚未完全建立或出现错误），则返回一个空字符串。

### PQport

返回活跃连接的端口。

```
char *PQport(const PGconn *conn);
```

如果在连接参数中指定了多个端口，PQport返回实际连接到的端口。

如果conn参数是NULL，则PQport返回NULL。否则，如果有一个错误产生端口信息（或许是连接没有被完全建立或者有什么错误），它会返回一个空字符串。

如果在连接参数中指定了多个端口，则在连接建立之前都不能依赖于PQport的结果。连接的状态可以用函数PQstatus检查。

### PQtty

返回该连接的调试TTY（这已被废弃，因为服务器不再关心TTY设置，但这个函数保持了向后兼容）。

```
char *PQtty(const PGconn *conn);
```

### PQoptions

返回被传递给连接请求的命令行选项。

```
char *PQoptions(const PGconn *conn);
```

下列函数返回会随着在PGconn对象上执行的操作改变的状态数据。

### PQstatus

返回该连接的状态。

```
ConnStatusType PQstatus(const PGconn *conn);
```

该状态可以是一系列值之一。不过，其中只有两个在一个异步连接过程之外可见：CONNECTION\_OK和CONNECTION\_BAD。一个到数据库的完好连接的状态为CONNECTION\_OK。一个失败的连接尝试则由状态CONNECTION\_BAD表示。通常，一个 OK 状态将一直保持到PQfinish，但是一次通信失败可能导致该状态过早地改变为CONNECTION\_BAD。在那种情况下，该应用可以通过调用PQreset尝试恢复。

关于其他可能会被返回的状态代码，请见PQconnectStartParams、PQconnectStart和PQconnectPoll的条目。

### PQtransactionStatus

返回服务器的当前事务内状态。

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

该状态可能是PQTRANS\_IDLE（当前空闲）、PQTRANS\_ACTIVE（一个命令运行中）、PQTRANS\_INTRANS（空闲，处于一个合法的事务块中）或者PQTRANS\_INERROR（空闲，处于一个失败的事务块中）。如果该连接损坏，将会报告PQTRANS\_UNKNOWN。只有当一个查询已经被发送给服务器并且还没有完成时，才会报告PQTRANS\_ACTIVE。

### PQparameterStatus

查找服务器的一个当前参数设置。

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

某一参数值会被服务器在连接开始或值改变时自动报告。PQparameterStatus可以被用来询问这些设置。它为已知的参数返回当前值，为未知的参数返回NULL。

当前版本报告的参数包括 server\_version、 server\_encoding、 client\_encoding、 application\_name、 is\_superuser、 session\_authorization、 DateStyle、 IntervalStyle、 TimeZone、 integer\_datetimes 和 standard\_conforming\_strings。请注意， server\_version、 server\_encoding 和 integer\_datetimes 在启动后无法更改。

3.0 之前协议的服务器不报告参数设置，但是libpq包括获得server\_version以及client\_encoding值的逻辑。我们鼓励应用使用PQparameterStatus而不是ad hoc代码来确定这些值（不过注意在一个 3.0 之前的连接上，连接开始后通过SET改变client\_encoding不会被PQparameterStatus反映）。对于server\_version（另见PQserverVersion），它以一种数字形式返回信息，这样更容易进行比较。

如果没有为standard\_conforming\_strings报告值，应用能假设它是off，也就是说反斜线会被视为字符串中的转义。还有，这个参数的存在可以被作为转义字符串语法（E'...'）被接受的指示。

尽管被返回的指针被声明成const，它事实上指向与PGconn结构相关的可变存储。假定该指针在存储之间保持有效是不明智的。

### PQprotocolVersion

询问所使用的 前端/后端协议。

```
int PQprotocolVersion(const PGconn *conn);
```

应用可能希望用这个函数来确定某些特性是否被支持。当前，可能值是 2（2.0 协议）、3（3.0 协议）或零（连接损坏）。协议版本在连接启动完成后将不会改变，但是理论上在连接重置期间是可以改变的。当与LightDB 7.4 或以后的服务器通信时通常使用 3.0 协议，7.4 之前的服务器只支持协议 2.0（协议 1.0 已被废弃并且不再被libpq支持）。

### PQserverVersion

返回一个表示服务器版本的整数。

```
int PQserverVersion(const PGconn *conn);
```

应用可能会使用这个函数来判断它们连接到的数据库服务器的版本。结果通过将服务器的主版本号乘以10000再加上次版本号形成。例如，版本10.1将被返回为100001，而版本11.0将被返回为110000。如果连接无效则返回零。

在主版本10之前，LightDB采用一种由三个部分组成的版本号，其中前两部分共同表示主版本。对于那些版本，PQserverVersion为每个部分使用两个数字，例如版本9.1.5将被返回为90105，而版本9.2.0将被返回为90200。

因此，出于判断特性兼容性的目的，应用应该将PQserverVersion的结果除以100而不是10000来判断逻辑的主版本号。在所有的发行序列中，只有最后两个数字在次发行（问题修正发行）之间不同。

#### PQlightdbserverVersion

返回表示 LightDB 服务器版本的整数。

```
int PQlightdbserverVersion(const PGconn *conn);
```

应用程序可以使用此函数确定它们连接的数据库服务器的 LightDB 版本。结果是通过将服务器的主版本号乘以10000并将次版本号加上100来形成的。例如，版本22.3将返回为220300，版本23.0将返回为230000。如果连接不良，则返回零。

#### PQerrorMessage

返回连接上的一个操作最近产生的错误消息。

```
char *PQerrorMessage(const PGconn *conn);
```

几乎所有的libpq在失败时都会为PQerrorMessage设置一个消息。注意按照libpq习惯，一个非空PQerrorMessage结果由多行构成，并且将包括一个尾部新行。调用者不应该直接释放结果。当相关的PGconn句柄被传递给PQfinish时，它将被释放。在PGconn结构上的多个操作之间，不能指望结果字符串会保持不变。

#### PQsocket

获得服务器连接套接字的文件描述符号。一个合法的描述符将会大于等于零。结果为-1表示当前没有打开服务器连接（在普通操作期间这不会改变，但是在连接设置或重置期间可能改变）。

```
int PQsocket(const PGconn *conn);
```

#### PQbackendPID

返回处理这个连接的后端进程的进程ID（PID）。

```
int PQbackendPID(const PGconn *conn);
```

后端PID有助于调试目的并且可用于与NOTIFY消息（它包括发出提示的后端进程的PID）进行比较。注意PID属于一个在数据库服务器主机上执行的进程，而不是本地主机进程！

#### PQconnectionNeedsPassword

如果连接认证方法要求一个口令但没有可用的口令，返回真（1）。否则返回假（0）。

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

这个函数可以在连接尝试失败后被应用于决定是否向用户提示要求一个口令。

#### PQconnectionUsedPassword

如果连接认证方法使用一个口令，返回真（1）。否则返回假（0）。

```
int PQconnectionUsedPassword(const PGconn *conn);
```

这个函数能在一次连接尝试失败或成功后用于检测该服务器是否要求一个口令。

下面的函数返回与 SSL 有关的信息。这些信息在连接建立后通常不会改变。

PQsslInUse

如果该连接使用了 SSL 则返回真（1），否则返回假（0）。

```
int PQsslInUse(const PGconn *conn);
```

PQsslAttribute

返回关于该连接的有关 SSL 的信息。

```
const char *PQsslAttribute(const PGconn *conn, const char *attribute_name);
```

可用的属性列表取决于使用的 SSL 库以及连接类型。如果一个属性不可用，会返回 NULL。

下列属性通常是可用的：

library

正在使用的 SSL 实现的名称（当前只实现了“OpenSSL”）

protocol

正在使用的 SSL/TLS 版本。常见的值有“TLSv1”、“TLSv1.1”以及“TLSv1.2”，但是如果一种实现使用了其他协议，可能返回其他字符串。

key\_bits

加密算法所使用的密钥位数。

cipher

所使用的加密套件的简短名称，例如“DHE-RSA-DES-CBC3-SHA”。这些名称与每一种 SSL 实现相关。

compression

如果正在使用 SSL 压缩，则返回压缩算法的名称。如果使用了压缩但是算法未知则返回“on”。如果没有使用压缩，则返回“off”。

PQsslAttributeNames

返回一个可用的 SSL 名称的数组。该数组以一个 NULL 指针终结。

```
const char * const * PQsslAttributeNames(const PGconn *conn);
```

PQsslStruct

返回一个描述该连接的 SSL 实现相关的对象指针。

```
void *PQsslStruct(const PGconn *conn, const char *struct_name);
```

可用的结构，这些结构取决于使用的 SSL 实现。对于 OpenSSL，有一个结构可用，其名称为“OpenSSL”。它返回一个指向该 OpenSSL SSL 结构的指针。要使用这个函数，可以用下面的代码：

```
#include <libpq-fe.h>
#include <openssl/ssl.h>

...

SSL *ssl;

dbconn = PQconnectdb(...);
...

ssl = PQsslStruct(dbconn, "OpenSSL");
if (ssl)
{
    /* 使用 OpenSSL 函数来访问 ssl */
}
```

这个结构可以被用来验证加密级别、检查服务器证书等等。有关这个结构的信息请参考 OpenSSL 的文档。

#### PQgetssl

返回连接中使用的 SSL 结构，如果没有使用 SSL 则返回空。

```
void *PQgetssl(const PGconn *conn);
```

这个函数等效于 PQsslStruct(conn, "OpenSSL")。在新的应用中不应该使用它，因为被返回的结构是 OpenSSL 专用的，如果使用了另一种 SSL 实现就不再可用。要检查一个连接是否使用 SSL，应该调用 PQsslInUse，而要得到有关连接的更多细节应该使用 PQsslAttribute。

## 32.3. 命令执行函数

一旦到一个数据库服务器的连接被成功建立，这里描述的函数可以被用来执行 SQL 查询和命令。

### 32.3.1. 主要函数

#### PQexec

提交一个命令给服务器并且等待结果。

```
PGresult *PQexec(PGconn *conn, const char *command);
```

返回一个 PGresult 指针或者可能是一个空指针。除了内存不足的情况或者由于严重错误无法将命令发送给服务器之外，一般都会返回一个非空指针。PQresultStatus 函数应当被调用来检查返回值是否代表错误（包括空指针的值，它会返回 PGRES\_FATAL\_ERROR）。用 PQerrorMessage 可得到关于那些错误的详细信息。

命令字符串可以包括多个 SQL 命令（用分号分隔）。在一次 PQexec 调用中被发送的多个查询会在一个事务中处理，除非其中有显式的 BEGIN/COMMIT 命令将该查询字符串划分成多个事务（服务器如何处理多查询字符串的更多细节请参考 Section 49.2.2.1）。但是注意，返回

的PGresult结构只描述该字符串中被执行的最后一个命令的结果。 如果一个命令失败，该字符串的处理会在它那里停止并且返回的PGresult会描述错误情况。

### PQexecParams

提交一个命令给服务器并且等待结果，它可以在 SQL 命令文本之外独立地传递参数。

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

PQexecParams与PQexec相似，但是提供了额外的功能：参数值可以与命令字符串分开指定，并且可以以文本或二进制格式请求查询结果。 PQexecParams只在 3.0 协议及其后的连接中被支持，当使用 2.0 协议时它会失败。

该函数的参数是：

conn

要在其中发送命令的连接对象。

command

要执行的 SQL 命令字符串。如果使用了参数，它们在该命令字符串中被引用为\$1、\$2等。

nParams

提供的参数数量。它是数组paramTypes[]、paramValues[]、paramLengths[]和paramFormats[]的长度（当nParams为零时，数组指针可以是NULL）。

paramTypes[]

通过 OID 指定要赋予给参数符号的数据类型。如果paramTypes为NULL或者该数组中任何特定元素为零，服务器会用对待未知类型文字串的方式为参数符号推测一种数据类型。

paramValues[]

指定参数的实际值。这个数组中的一个空指针表示对应的参数为空，否则该指针指向一个以零终止的文本字符串（用于文本格式）或者以服务器所期待格式的二进制数据（用于二进制格式）。

paramLengths[]

指定二进制格式参数的实际数据长度。它对空参数和文本格式参数被忽略。当没有二进制参数时，该数组指针可以为空。

paramFormats[]

指定参数是否为文本（在参数相应的数组项中放一个零）或二进制（在参数相应的数组项中放一个一）。如果该数组指针为空，那么所有参数都会被假定为文本串。

以二进制格式传递的值要求后端所期待的内部表示形式的知识。例如，整数必须以网络字节序被传递。传递numeric值要求关于服务器存储格式的知识，正如src/

backend/utils/adt/numeric.c::numeric\_send() 以及src/backend/utils/adt/numeric.c::numeric\_recv() 中所实现的。

Lightdb 23.3 引入了一个新的数据库级别 GUC 参数 `lightdb_ascii_zero_store_value`。此参数取值范围为 0 到 32，包含 0，32。如果数据库兼容模式为 Oracle 或 MySQL，并且 `lightdb_ascii_zero_store_value` 为非零 ASCII 码值，当参数类型为文本类型的时候，`'\0'` 被替换为 ASCII 码值 `lightdb_ascii_zero_store_value`。

更多关于 GUC `lightdb_ascii_zero_store_value` 的信息，请参考 `lightdb_ascii_zero_store_value`

#### resultFormat

指定零来得到文本格式的结果，或者指定一来得到二进制格式的结果（目前没有规定要求以不同格式得到不同的结果列，尽管在底层协议中这是可以实现的）。

PQexecParams 相对于 PQexec 的主要优点是参数值可以从命令串中分离，因此避免了冗长的书写、容易发生错误的引用以及转义。

和 PQexec 不同，PQexecParams 至多允许在给定串中出现一个 SQL 命令（其中可以有分号，但是不能有超过一个非空命令）。这是底层协议的一个限制，但是有助于抵抗 SQL 注入攻击。

#### Tip

通过 `OID` 指定参数类型很罗嗦，特别是如果你不愿意将特定的 `OID` 值硬编码到你的程序中时。不过，即使服务器本身也无法确定参数的类型，你可以避免这样做，或者选择一种与你想要的不同的类型。在 SQL 命令文本中，附加一个显式造型给参数符号来表示你将发送什么样的数据类型。例如：

```
SELECT * FROM mytable WHERE x = $1::bigint;
```

这强制参数 `$1` 被当作 `bigint`，而默认情况下它将被赋予与 `x` 相同的类型。当以二进制格式发送参数值时，我们强烈推荐以这种方式或通过指定一个数字类型的 `OID` 来强制参数类型决定。因为二进制格式比文本格式具有更少的冗余，并且因此服务器将不会有更多机会为你检测一个类型匹配错误。

#### PQprepare

提交一个请求用给定参数创建一个预备语句并且等待完成。

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

PQprepare 创建一个后面会由 PQexecPrepared 执行的预备语句。这个特性允许命令被反复执行而无需每次都进行解析和规划，详见 PREPARE。PQprepare 只在协议 3.0 及之后的连接中被支持，当使用协议 2.0 时它将失败。

该函数从 query 串创建一个名为 stmtName 的预备语句，该串必须包含一个单一 SQL 命令。stmtName 可以是 "" 来创建一个未命名语句，在这种情况下任何已存在未命名语句将被自动替换。如果语句名称已经在当前会话中被定义，则是一种错误。如果使用了任何参数，它们在查询中以 \$1、\$2 等引用。nParams 是参数的个数，其类型在数

组paramTypes[]中被预先指定（当nParams为零时，该数组指针可以是NULL）。paramTypes[]通过OID指定要赋予给参数符号的数据类型。如果paramTypes是NULL或者该数组中任何特定元素为零，服务器会用对待未知类型文字串的方式为参数符号推测一种数据类型。还有，查询能够使用编号高于nParams的参数符号，它们的数据类型也会被自动推测（找出推测出的数据类型的方法见PQdescribePrepared）。

正如PQexec一样，结果通常是一个PGresult对象，其内容代表服务器端成功或失败。一个空结果表示内存不足或者根本无法发送命令。关于错误的更多信息请见PQerrorMessage。

用于PQexecPrepared的预备语句也能通过执行SQL PREPARE语句来创建。还有，尽管没有libpq函数来删除一个预备语句，SQL DEALLOCATE语句可被用于此目的。

### PQexecPrepared

发送一个请求来用给定参数执行一个预备语句，并且等待结果。

```
PGresult *PQexecPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

PQexecPrepared像PQexecParams，但是要执行的命令是用之前准备的语句的名字指定，而不是指定一个查询串。这个特性允许将被重复使用的命令只被解析和规划一次，而不是在每次被执行时都被解析和规划。这个语句必须之前在当前会话中已经被准备好。PQexecPrepared仅被协议3.0及之后的连接支持，当使用协议2.0时它会失败。

参数和PQexecParams相同，除了给定的是一个预备语句的名称而不是一个查询语句，以及不存在paramTypes[]参数（因为预备语句的参数类型已经在它被创建时决定好了）。

### PQdescribePrepared

提交一个请求来获得有关指定预备语句的信息，并且等待完成。

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

PQdescribePrepared允许一个应用获得有关一个之前预备好的语句的信息。PQdescribePrepared仅被协议3.0及之后的连接支持，当使用协议2.0时它会失败。

stmtName可以用""或者NULL来引用未命名语句，否则它必须是一个现有预备语句的名字。如果成功，一个PGresult以及状态PGRES\_COMMAND\_OK会被返回。函数PQnparams和PQparamtype可以被应用到这个PGresult来得到关于该预备语句参数的额外信息，而函数PQnfields, PQfname, PQftype等提供该语句结果列（如果有）的信息。

### PQdescribePortal

提交一个请求来得到有关指定入口的信息，并且等待完成。

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

PQdescribePortal允许一个应用获得有关一个之前被创建的入口的信息（libpq不提供对入口任何直接的访问，但是你可以使用这个函数来观察一个通过DECLARE CURSOR SQL命令创建的游标的属性）。PQdescribePortal仅被协议3.0及之后的连接支持，当使用协议2.0时它会失败。



portalName可以用""或者NULL来引用未命名入口，否则它必须是一个现有入口的名字。如果成功，一个PGresult和状态PGRES\_COMMAND\_OK会被返回。函数PQnfields, PQfname, PQftype等可以被应用到PGresult来获得有关该入口结果列（如果有）的信息。

PGresult结构封装了由服务器返回的结果。libpq应用程序员应该小心地维护PGresult的抽象。使用下面的存储器函数来得到PGresult的内容。避免直接引用PGresult结构的域，因为它们可能在未来更改。

### PQresultStatus

返回该命令的结果状态。

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus能返回下列值之一：

PGRES\_EMPTY\_QUERY

发送给服务器的字符串为空。

PGRES\_COMMAND\_OK

一个不返回数据的命令成功完成。

PGRES\_TUPLES\_OK

一个返回数据的命令（例如SELECT或者SHOW）成功完成。

PGRES\_COPY\_OUT

从服务器复制出数据的传输开始。

PGRES\_COPY\_IN

复制数据到服务器的传输开始。

PGRES\_BAD\_RESPONSE

无法理解服务器的响应。

PGRES\_NONFATAL\_ERROR

发生了一次非致命错误（一个提示或警告）。

PGRES\_FATAL\_ERROR

发生了一次致命错误。

PGRES\_COPY\_BOTH

向服务器复制数据/从服务器复制数据的传输开始。这个特性当前只被用于流复制，因此这个状态应该不会在普通应用中出现。

PGRES\_SINGLE\_TUPLE

PGresult包含来自于当前命令的一个单一结果元组。这个状态只在查询选择了单一行模式时发生（见Section 32.5）。

如果结果状态是PGRES\_TUPLES\_OK或者PGRES\_SINGLE\_TUPLE，那么下面所描述的函数能被用来检索该查询所返回的行。注意，一个恰好检索零行的SELECT命令仍然会显

示PGRES\_TUPLES\_OK。PGRES\_COMMAND\_OK用于从不返回行的命令（不带RETURNING子句的INSERT或者UPDATE等）。一个PGRES\_EMPTY\_QUERY可能表示客户端软件中的一个缺陷。

一个状态为PGRES\_NONFATAL\_ERROR的结果将不会被PQexec或者其他查询执行函数直接返回，这类结果将被传递给提示处理器（见 Section 32.12）。

### PQresStatus

将PQresultStatus返回的枚举转换成描述状态编码的字符串常量。调用者不应该释放结果。

```
char *PQresStatus(ExecStatusType status);
```

### PQresultErrorMessage

返回与该命令相关的错误消息，如果有错误则会返回一个空字符串。

```
char *PQresultErrorMessage(const PGresult *res);
```

如果有一个错误，被返回的字符串将包含一个收尾的新行。调用者不应该直接释放结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

紧跟着一个PQexec 或 PQgetResult调用，PQerrorMessage（在连接上）将返回与PQresultErrorMessage相同的字符串（在结果上）。不过，一个PGresult将保持它的错误消息直到被销毁，而连接的错误消息将在后续操作被执行时被更改。当你想要知道与一个特定PGresult相关的状态，使用PQresultErrorMessage。而当你想要知道连接上最后一个操作的状态，使用PQerrorMessage。

### PQresultVerboseErrorMessage

返回与PGresult对象相关的错误消息的重新格式化的版本。

```
char *PQresultVerboseErrorMessage(const PGresult *res,
                                  PGVerbosity verbosity,
                                  PGContextVisibility show_context);
```

在有些情况下，客户端可能希望得到之前报告过的错误的更加详尽的版本。如果在产生给定PGresult的连接上 verbosity 设置有效，PQresultVerboseErrorMessage会通过计算已经被PQresultErrorMessage产生过的消息来满足这种需求。如果PGresult不是一个错误结果，则会报告“PGresult is not an error result”。返回的字符串包括一个新行作为结尾。

和大部分从PGresult中提取数据的其他函数不同，这个函数的结果是一个全新分配的字符串。调用者在不需要这个字符串以后，必须使用PQfreemem()释放它。

如果内存不足，可能会返回 NULL。

### PQresultErrorField

返回一个错误报告的一个域。

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

fieldcode是一个错误域标识符，见下列符号。如果PGresult不是一个错误或者警告结果或者不包括指定域，会返回NULL。域通常不包括一个收尾的新行。调用者不应该直接释放结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

下列域代码可用：

#### PG\_DIAG\_SEVERITY

严重性。域的内容是ERROR、FATAL或PANIC（在一个错误消息中）。或者是WARNING、NOTICE、DEBUG、INFO或LOG（在一个提示消息中）。或者是其中之一的一个本地化翻译。总是存在。

#### PG\_DIAG\_SEVERITY\_NONLOCALIZED

域的内容是ERROR、FATAL或PANIC（在一个错误消息中）。或者是WARNING、NOTICE、DEBUG、INFO或LOG（在一个提示消息中）。这和PG\_DIAG\_SEVERITY域相同，不过内容不会被本地化。只存在于LightDB 9.6 版本以后产生的报告中。

#### PG\_DIAG\_SQLSTATE

用于错误的 SQLSTATE 代码。SQLSTATE 代码标识了已经发生的错误的类型，它可以被前端应用用来执行特定操作（例如错误处理）来响应一个特定数据库错误。一个可能的 SQLSTATE 代码列表可见Appendix C 这个域无法被本地化，并且总是存在。

#### PG\_DIAG\_MESSAGE\_PRIMARY

主要的人类可读的错误消息（通常是一行）。总是存在。

#### PG\_DIAG\_MESSAGE\_DETAIL

细节：一个可选的次级错误消息，它携带了关于问题的等多细节。可能有多行。

#### PG\_DIAG\_MESSAGE\_HINT

提示：一个关于如何处理该问题的可选建议。它与细节的区别在于它提供了建议（可能不合适）而不是铁的事实。可能有多行。

#### PG\_DIAG\_STATEMENT\_POSITION

包含一个十进制整数的字符串，它表示一个错误游标位置，该位置是原始语句字符串的索引。第一个字符的索引是 1，位置以字符计算而不是以及字节计算。

#### PG\_DIAG\_INTERNAL\_POSITION

这被定义为与PG\_DIAG\_STATEMENT\_POSITION域相同，但是它被用在游标位置引用一个内部产生的命令而不是客户端提交的命令时。当这个域出现时，PG\_DIAG\_INTERNAL\_QUERY域将总是出现。

#### PG\_DIAG\_INTERNAL\_QUERY

一个失败的内部产生的命令的文本。例如，这可能是由一个 PL/pgSQL 函数发出的 SQL 查询。

#### PG\_DIAG\_CONTEXT

指示错误发生的环境。当前这包括活动过程语言函数的调用栈追踪以及内部生成的查询。追踪是每行一项，最近的排在最前面。

#### PG\_DIAG\_SCHEMA\_NAME

如果错误与某个特定的数据库对象相关，这里是包含该对象的模式名（如果有）。

#### PG\_DIAG\_TABLE\_NAME

如果错误与某个特定表相关，这里是该表的名字（该表的模式参考模式名域）。

PG\_DIAG\_COLUMN\_NAME

如果错误与一个特定表列相关，这里是该表列的名字（参考模式和表名域来标识该表）。

PG\_DIAG\_DATATYPE\_NAME

如果错误与一个特定数据类型相关，这里是该数据了行的名字（该数据类型的模式名参考模式名域）。

PG\_DIAG\_CONSTRAINT\_NAME

如果错误与一个特定约束相关，这里是该约束的名字。相关的表或域参考上面列出的域（为了这个目的，索引也被视作约束，即使它们不是用约束语法创建的）。

PG\_DIAG\_SOURCE\_FILE

报告错误的源代码所在的文件名。

PG\_DIAG\_SOURCE\_LINE

报告错误的源代码行号。

PG\_DIAG\_SOURCE\_FUNCTION

报告错误的源代码函数的名字。

### Note

用于模式名、表名、列名、数据类型名和约束名的域只提供给有限的错误类型，见Appendix C。不要假定任何这些域的存在保证另一个域的存在。核心错误源会遵守上面提到的内在联系，但是用户定义的函数可能以其他方式使用这些域。同样地，不要假定这些域代表当前数据库中同类的对象。

客户端负责格式化显示信息来迎合它的需要，特别是根据需要打断长的行。出现在错误消息域中的新行字符应该被当作分段而不是换行。

libpq内部产生的错误将有严重和主要消息，但是通常没有其他域。3.0 协议之前的服务器返回的错误将包括严重和主要消息，并且有时候还有细节消息，但是没有其他域。

注意错误与只从PGresult对象中有效，对PGconn对象无效。没有PQerrorField函数。

PQclear

Frees the storage associated with a 释放与一个PGresult相关的存储。每一个命令结果不再需要时应该用PQclear释放。

```
void PQclear(PGresult *res);
```

你可以按照需要保留PGresult对象，当你发出一个新命令时它也不会消失，甚至关闭连接时也不会消失。要去掉它，你必须调用PQclear。没有这样做将会导致在应用中的内存泄露。

## 32.3.2. 检索查询结果信息

这些函数被用来从一个代表成功查询结果（也就是状态为PGRES\_TUPLES\_OK或者PGRES\_SINGLE\_TUPLE）的PGresult对象中抽取信息。它们也可以被用来从一个成功的

**Describe 操作中抽取信息：**一个 Describe 的结果具有和该查询被实际执行所提供的完全相同的列信息，但是它没有行。对于其他状态值的对象，这些函数会认为结果具有零行和零列。

#### PQntuples

返回查询结果中的行（元组）数（注意，PGresult对象被限制为不超过INT\_MAX行，因此一个int结果就足够了）。

```
int PQntuples(const PGresult *res);
```

#### PQnfields

返回查询结果中每一行的列（域）数。

```
int PQnfields(const PGresult *res);
```

#### PQfname

返回与给定列号相关联的列名。列号从 0 开始。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

```
char *PQfname(const PGresult *res,
              int column_number);
```

如果列号超出范围，将返回NULL。

#### PQfnumber

返回与给定列名相关联的列号。

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

如果给定的名字不匹配任何列，将返回 -1。

给定的名称被视作一个 SQL 命令中的一个标识符，也就是说，除非被双引号引用，它是小写形式的。例如，给定一个 SQL 命令：

```
SELECT 1 AS F00, 2 AS "BAR";
```

我们将得到结果：

PQfname(res, 0)	foo
PQfname(res, 1)	BAR
PQfnumber(res, "F00")	0
PQfnumber(res, "foo")	0
PQfnumber(res, "BAR")	-1
PQfnumber(res, "\"BAR\"")	1

#### PQftable

返回给定列从中取出的表的 OID。列号从 0 开始。

```
Oid PQftable(const PGresult *res,  
             int column_number);
```

如果列号超出范围或者指定的列不是对一个表列的简单引用或者在使用 3.0 协议时，返回 InvalidOid。你可以查询系统表 pg\_class 来确定究竟是哪个表被引用。

当你包括 libpq 头文件，类型 oid 以及常数 InvalidOid 将被定义。它们将都是某种整数类型。

#### PQftablecol

返回构成指定查询结果列的列（在其表中）的列号。查询结果列号从 0 开始，但是表列具有非零编号。

```
int PQftablecol(const PGresult *res,  
               int column_number);
```

如果列号超出范围或者指定的列不是对一个表列的简单引用或者在使用 3.0 协议时，返回零。

#### PQfformat

返回指示给定列格式的格式编码。列号从 0 开始。

```
int PQfformat(const PGresult *res,  
             int column_number);
```

格式代码零指示文本数据表示，而格式代码一表示二进制表示（其他代码被保留用于未来的定义）。

#### PQftype

返回与给定列号相关联的数据类型。被返回的整数是该类型的内部 OID 号。列号从 0 开始。

```
Oid PQftype(const PGresult *res,  
           int column_number);
```

你可以查询系统表 pg\_type 来得到多个数据类型的名字和属性。内建数据类型的 OID 被定义在源代码树中的文件 src/include/catalog/pg\_type\_d.h 中。

#### PQfmod

返回与给定列号相关联的列的修饰符类型。列号从 0 开始。

```
int PQfmod(const PGresult *res,  
          int column_number);
```

修饰符值的解释是与类型相关的，它们通常指示精度或尺寸限制。值 -1 被用来指示“没有信息可用”。大部分的数据类型不适用修饰符，在那种情况中值总是 -1。

#### PQfsize

返回与给定列号相关的列的尺寸（以字节计）。列号从 0 开始。

```
int PQfsize(const PGresult *res,
```

```
int column_number);
```

PQfsize返回在一个数据库行中为这个列分配的空间，换句话说就是服务器对该数据类型的内部表示的尺寸（因此，它对客户端并不是真地非常有用）。一个负值指示该数据类型是变长的。

### PQbinaryTuples

如果PGresult包含二进制数据，返回 1。如果包含的是文本数据，返回 0。

```
int PQbinaryTuples(const PGresult *res);
```

这个函数已经被废弃（除了与COPY一起使用），因为一个单一PGresult可以在某些列中包含文本数据而且在另一些列中包含二进制数据。PQfformat要更好。只有结果的所有列是二进制（格式 1）时PQbinaryTuples才返回 1。

### PQgetvalue

返回一个PGresult的一行的单一域值。行和列号从 0 开始。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

对于文本格式的数据，PQgetvalue返回的值是该域值的一种空值结束的字符串表示。对于二进制格式的数据，该值是由该数据类型的typsend和typreceive函数决定的二进制表示（在这种情况下该值实际上也跟随着一个零字节，但是这通常没有用处，因为该值很可能包含嵌入的空）。

如果该域值为空，则返回一个空串。关于区分空值和空字符串值请见PQgetisnull。

PQgetvalue返回的指针指向作为PGresult结构一部分的存储。我们不应该修改它指向的数据，并且如果要在超过PGresult结构本身的生命期之外使用它，我们必须显式地把该数据拷贝到其他存储中。

### PQgetisnull

测试一个域是否为空值。行号和列号从 0 开始。

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

如果该域是空，这个函数返回 1。如果它包含一个非空值，则返回 0（注意PQgetvalue将为一个空域返回一个空串，不是一个空指针）。

### PQgetlength

返回一个域值的真实长度，以字节计。行号和列号从 0 开始。

```
int PQgetlength(const PGresult *res,
                 int row_number,
                 int column_number);
```

这是特定数据值的真实数据长度，也就是PQgetvalue指向的对象的尺寸。对于文本数据格式，这和strlen()相同。对于二进制格式这是基本信息。注意我们不应该依赖于PQfsize来得到真值的数据长度。

## PQnparams

返回一个预备语句的参数数量。

```
int PQnparams(const PGresult *res);
```

只有在查看PQdescribePrepared的结果时，这个函数才有用。对于其他类型的查询，它将返回零。

## PQparamtype

返回所指示的语句参数的数据类型。参数号从 0 开始。

```
Oid PQparamtype(const PGresult *res, int param_number);
```

只有在查看PQdescribePrepared的结果时，这个函数才有用。对于其他类型的查询，它将返回零。

## PQprint

将所有的行打印到指定的输出流，以及有选择地将列名打印到指定的输出流。

```
void PQprint(FILE *fout,          /* 输出流 */
             const PGresult *res,
             const PQprintOpt *po);
typedef struct
{
    pqbool header;      /* 打印输出域标题和行计数 */
    pqbool align;      /* 填充对齐域 */
    pqbool standard;   /* 旧的格式 */
    pqbool html3;      /* 输出 HTML 表格 */
    pqbool expanded;   /* 扩展表格 */
    pqbool pager;      /* 如果必要为输出使用页 */
    char *fieldSep;    /* 域分隔符 */
    char *tableOpt;    /* 用于 HTML 表格元素的属性 */
    char *caption;     /* HTML 表格标题 */
    char **fieldName; /* 替换域名称的空终止数组 */
} PQprintOpt;
```

这个函数以前被ltsql用来打印查询结果，但是现在不是这样了。注意它假定所有的数据都是文本格式。

### 32.3.3. 检索其他结果信息

这些函数被用来从PGresult对象中抽取其他信息。

## PQcmdStatus

返回来自于产生PGresult的 SQL 命令的命令状态标签。

```
char *PQcmdStatus(PGresult *res);
```

通常这就是该命令的名称，但是它可能包括额外数据，例如已被处理的行数。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

## PQcmdTuples

返回受该 SQL 命令影响的行数。



```
char *PQcmdTuples(PGresult *res);
```

这个函数返回一个字符串，其中包含着产生PGresult的SQL语句影响的行数。这个只能被用于下列情况：执行一个SELECT、CREATE TABLE AS、INSERT、UPDATE、DELETE、MOVE、FETCH或COPY语句，或者一个包含INSERT、UPDATE或DELETE语句的预备查询的EXECUTE。如果产生PGresult的命令是其他什么东西，PQcmdTuples会返回一个空串。调用者不应该直接释放该结果。它将在相关的PGresult句柄被传递给PQclear之后被释放。

PQoidValue

如果该SQL命令是一个正好将一行插入到具有OID的表的INSERT，或者是一个包含合适INSERT语句的预备查询的EXECUTE，这个函数返回被插入行的OID。否则，这个函数返回InvalidOid。如果被INSERT语句影响的表不包含OID，这个函数也将返回InvalidOid。

```
Oid PQoidValue(const PGresult *res);
```

PQoidStatus

这个函数已经被PQoidValue取代，并且不是线程安全的。它返回包含被插入行的OID的一个字符串，而PQoidValue返回OID值。

```
char *PQoidStatus(const PGresult *res);
```

### 32.3.4. 用于包含在 SQL 命令中的转移串

PQescapeLiteral

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

为了让一个串能用在SQL命令中，PQescapeLiteral可对它进行转义。当在SQL命令中插入一个数据值作为文字常量时，这个函数很有用。一些字符（例如引号和反斜线）必须被转义以防止它们被SQL解析器解释成特殊的意思。PQescapeLiteral执行这种操作。

PQescapeLiteral返回一个str参数的已被转义版本，该版本被放在用malloc()分配的内存中。当该结果不再被需要时，这个内存应该用PQfreemem()释放。一个终止的零字节不是必须的，并且不应该被计入length（如果在length字节被处理之前找到一个终止字节，PQescapeLiteral会停止在零，该行为更像strncpy）。返回串中的所有特殊字符都被替换掉，这样它们能被LightDB字符串解析器正确地处理。还会加上一个终止零字节。包括在结果串中的LightDB字符串必须用单引号包围。

发生错误时，PQescapeLiteral返回NULL并且一个合适的消息会被存储在conn对象中。

#### Tip

在处理从一个非可信源接收到的串时，做正确的转义特别重要。否则就会有安全性风险：你容易受到“SQL注入”攻击，其中可能会有预期之外的SQL语句会被喂给你的数据库。

注意，当一个数据值被作为PQexecParams或其兄弟例程中的一个独立参数传递时，没有必要做转义而且做转义也不正确。

## PQescapeIdentifier

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

PQescapeIdentifier转义一个要用作 SQL 标识符的字符串，例如表名、列名或函数名。当一个用户提供的标识符可能包含被 SQL 解析器解释为标识符一部分的特殊字符时，或者当该标识符可能包含大小写形式应该被保留的大写字符时，这个函数很有用。

PQescapeIdentifier返回一个str参数的已被转义为 SQL 标识符的版本，该版本被放在用malloc()分配的内存中。当该结果不再被需要时，这个内存应该用PQfreemem()释放。一个终止的零字节不是必须的，并且不应该被计入length（如果在length字节被处理之前找到一个终止字节，PQescapeIdentifier会停止在零，该行为更像strncpy）。返回串中的所有特殊字符都被替换掉，这样它们能被作为一个 SQL 标识符正确地处理。还会加上一个终止零字节。返回串也将被双引号包围。

发生错误时，PQescapeIdentifier返回NULL并且一个合适的消息会被存储在conn对象中。

## Tip

与字符串一样，要阻止 SQL 注入攻击，当从一个不可信的来源接收到 SQL 标识符时，它们必须被转义。

## PQescapeStringConn

```
size_t PQescapeStringConn(PGconn *conn,
                           char *to, const char *from, size_t length,
                           int *error);
```

PQescapeStringConn转义字符串，它很像PQescapeLiteral。与PQescapeLiteral不同的是，调用者负责提供一个合适尺寸的缓冲区。此外，PQescapeStringConn不产生必须包围LightDB字符串的单引号。它们应该在结果要插入的 SQL 命令中提供。参数from指向要被转义的串的第一个字符，并且length参数给出了这个串中的字节数。一个终止的零字节不是必须的，并且不应该被计入length（如果在length字节被处理之前找到一个终止字节，PQescapeStringConn会停止在零，该行为更像strncpy）。to应当指向一个缓冲区，它能够保持至少比length值的两倍还要多至少一个字节，否则该行为是未被定义的。如果to和from串重叠，行为也是未被定义的。

如果error参数不是NULL，那么成功时\*error被设置为零，错误时设置为非零。当前唯一可能的错误情况涉及源串中非法的多字节编码。错误时仍然会产生输出串，但是可以预期服务器将认为它是畸形的并且拒绝它。在发生错误时，一个合适的消息被存储在conn对象中，不管error是不是NULL。

PQescapeStringConn返回写到to的字节数，不包括终止的零字节。

## PQescapeString

PQescapeString是一个更老的被废弃的PQescapeStringConn版本。

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

PQescapeStringConn和PQescapeString之间的唯一区别是不需要PGconn或error参数。正因为如此，它不能基于连接属性（例如字符编码）调整它的行为并且因此它可能给出错误的结果。还有，它没有方法报告错误情况。

PQescapeString可以在一次只使用一个LightDB连接的客户端程序中安全地使用（在这种情况下它可以“在现象后面”找出它需要知道的东西）。在其他环境中它是一个安全性灾难并且应该用PQescapeStringConn来避免。

### PQescapeByteaConn

把要用于一个 SQL 命令的二进制数据用类型bytea转义。和PQescapeStringConn一样，只有在将数据直接插入到一个 SQL 命令串时才使用它。

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                const unsigned char *from,
                                size_t from_length,
                                size_t *to_length);
```

当某些字节值被用作一个SQL语句中的bytea文字的一部分时，它们必须被转义。PQescapeByteaConn转义使用十六进制编码或反斜线转义的字节。详见Section 8.4

from参数指向要被转义的串的第一个字节，并且from\_length参数给出这个二进制串中的字节数（一个终止的零字节是不需要的也是不被计算的）。to\_length参数指向一个将保持生成的已转义串长度的变量。这个结果串长度包括结果的终止零字节。

PQescapeByteaConn返回一个from参数的已被转义为二进制串的版本，该版本被放在用malloc()分配的内存中。当该结果不再被需要时，这个内存应该用PQfreemem()释放。返回串中的所有特殊字符都被替换掉，这样它们能被LightDB的字符串解析器以及bytea输入函数正确地处理。还会加上一个终止零字节。不是结果串一部分的LightDB字符串必须被单引号包围。

在发生错误时，将返回一个空指针，并且一个合适的错误消息被存储在conn对象中。当前，唯一可能的错误是没有足够的内存用于结果串。

### PQescapeBytea

PQescapeBytea是一个更老的被废弃的PQescapeByteaConn版本。

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

与PQescapeByteaConn的唯一区别是PQescapeBytea不用一个PGconn参数。正因为这样，PQescapeBytea只能在一次只使用一个LightDB连接的客户端程序中安全地使用（在这种情况下它可以“在现象后面”找出它需要知道的东西）。如果在有多个数据库连接的程序中使用，它可能给出错误的结果（在那种情况下使用PQescapeByteaConn）。

### PQunescapeBytea

将二进制数据的一个字符串表示转换成二进制数据 — 它是PQescapeBytea的逆向函数。当检索文本格式的bytea数据时，需要这个函数，但检索二进制个事时则不需要它。

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

from参数指向一个字符串，例如PQgetvalue被应用到一个bytea列上所返回的。PQunescapeBytea把这个串表示转换成它的二进制表示。它返回一个指向用malloc()分配的缓冲区的指针，在错误时返回NULL，并且把缓冲区的尺寸放在to\_length中。当结果不再需要时，它必须使用PQfreemem释放。

这种转换并不完全是PQescapeBytea的逆函数，因为当从PQgetvalue接收到字符串时，我们并不能期待它被“转义”。特别地这意味着不需要考虑字符串引用，并且因此也不需要一个参数。

## 32.4. 异步命令处理

PQexec函数对于在普通的同步应用中提交命令是足以胜任的。不过，它的一些缺点可能对某些用户很重要：

- PQexec会等待命令完成。该应用可能有其他的工作要做（例如维护用户界面），这时它不希望阻塞等待回应。
- 因为客户端应用的执行在它等待结果时会被挂起，对于应用来说很难决定要不要尝试取消正在进行的命令（这可以在一个信号处理器中完成，但别无他法）。
- PQexec只能返回一个PGresult结构。如果提交的命令串包含多个SQL命令，除了最后一个PGresult之外都会被PQexec丢弃。
- PQexec总是收集命令的整个结果，把它缓存在一个单一的PGresult中。虽然这简化了应用的错误处理逻辑，它对于包含很多行的结果并不现实。

不想受到这些限制的应用可以改用构建PQexec的底层函数：PQsendQuery以及PQgetResult。还有 PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared, 和 PQsendDescribePortal, 它们可以与 PQgetResult 一起使用来分别复制 PQexecParams, PQprepare, PQexecPrepared, PQdescribePrepared, 和 PQdescribePortal的函数。

### PQsendQuery

向服务器提交一个命令而不等待结果。如果该命令被成功发送则返回 1，否则返回 0（此时，可以用PQerrorMessage获取关于失败的信息）。

```
int PQsendQuery(PGconn *conn, const char *command);
```

在成功调用PQsendQuery后，调用PQgetResult一次或者多次来获取结果。在PQsendQuery返回一个空指针之前，都不能再次调用PQgetResult，返回的空指针指示该命令已经完成。

### PQsendQueryParams

向服务器提交一个命令和单独的参数，而不等待结果。

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

这个函数等效于PQsendQuery，不过查询参数可以独立于查询字符串分开指定。该函数的参数处理和PQexecParams一样。和PQexecParams类似，它不能在 2.0 协议的连接上工作，并且它只允许在查询字符串中有一条命令。

### PQsendPrepare

发送一个请求用给定参数创建一个预备语句，而不等待完成。

```
int PQsendPrepare(PGconn *conn,
                 const char *stmtName,
                 const char *query,
                 int nParams,
                 const Oid *paramTypes);
```

这个函数是PQprepare的异步版本：如果它能发送这个请求，则返回 1；如果不能，则返回 0。在成功调用之后，调用PQgetResult判断服务器是否成功创建了预备语句。这个函数的参数的处理和PQprepare一样。和PQprepare类似，它不能在 2.0 协议的连接上工作。

#### PQsendQueryPrepared

发送一个请求用给定参数执行一个预备语句，而不等待结果。

```
int PQsendQueryPrepared(PGconn *conn,
                       const char *stmtName,
                       int nParams,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

这个函数与PQsendQueryParams类似，但是要执行的命令是通过一个之前已经命名的预备语句指定，而不是一个给出的查询字符串。该函数的参数处理和PQexecPrepared一样。和PQexecPrepared类似，它不能在 2.0 协议的连接上工作。

#### PQsendDescribePrepared

发送一个请求获得指定的预备语句的信息，但不等待完成。

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

这个函数是PQdescribePrepared的一个异步版本：如果它能够发送请求，则返回 1；否则，返回 0。在一次成功的调用后，调用PQgetResult来得到结果。该函数的参数处理和PQdescribePrepared一样。和PQdescribePrepared类似，它不能在 2.0 协议的连接上工作。

#### PQsendDescribePortal

提交一个请求来获得关于指定入口的信息，但不等待完成。

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

这个函数是PQdescribePortal的一个异步版本：如果它能够发送请求，则返回 1；否则，返回 0。在一次成功的调用后，调用PQgetResult来得到结果。该函数的参数处理和PQdescribePortal一样。和PQdescribePortal类似，它不能在 2.0 协议的连接上工作。

#### PQgetResult

等待来自于一个之前的 PQsendQuery, PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared, 或 PQsendDescribePortal调用的结果，并且返回它。当该命令完成并且没有更多结果时，将返回一个空指针。

```
PGresult *PQgetResult(PGconn *conn);
```

PQgetResult必须被反复调用直到它返回一个空指针，空指针表示该命令完成（如果在没有命令活动时被调用，PQgetResult将立即返回一个空指针）。每一个来自PQgetResult的非空结果应该使用之前描述的同一个PGresult访问器处理。不要忘记在处理完之后释放PQclear的每一个结果对象。注意，只有一个命令是活动的并且PQconsumeInput 还没有读取必要的响应数据时，PQgetResult将会阻塞。

### Note

即使当PQresultStatus指出一个致命错误时，PQgetResult也应当被调用直到它返回一个空指针，以允许libpq完全处理该错误信息。

使用PQsendQuery和PQgetResult解决了PQexec的一个问题：如果一个命令字符串包含多个SQL命令，这些命令的结果可以被个别地获得（顺便说一句：这样就允许一种简单的重叠处理形式，客户端可以处理一个命令的结果，而同时服务器可以继续处理同一命令字符串中后面的查询）。

可以被PQsendQuery和PQgetResult获得的另一种常常想要的特性是一次从大型结果中检索一行。这会在Section 32.5中讨论。

就其本身而言，调用PQgetResult将仍会导致客户端阻塞，直到服务器完成下一个SQL命令。可以通过正确使用两个函数来避免这种情况：

#### PQconsumeInput

如果有来自服务器的输入可用，则使用之。

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput 通常返回 1 表明“没有错误”，而返回 0 表明有某种麻烦发生（此时可以用PQerrorMessage）。注意该结果并不表明是否真正收集了任何输入数据。在调用PQconsumeInput 之后，应用可以检查PQisBusy和/或PQnotifies来看看它们的状态是否改变。

即使应用还不准备处理一个结果或通知，PQconsumeInput 也可以被调用。这个函数将读取可用的数据并且把它保存在一个缓冲区中，从而导致一个select()的读准备好指示消失。因此应用可以使用PQconsumeInput 立即清除select()条件，并且在空闲时再检查结果。

#### PQisBusy

如果一个命令繁忙则返回 1，也就是说PQgetResult会阻塞等待输入。返回 0 表示可以调用PQgetResult而不用担心阻塞。

```
int PQisBusy(PGconn *conn);
```

PQisBusy本身将不会尝试从服务器读取数据，因此必须先调用PQconsumeInput，否则繁忙状态将永远不会结束。

一个使用这些函数的典型应用将有一个主循环，在主循环中会使用select()或poll()等待所有它必须响应的情况。其中之一将是来自服务器的输入可用，对select()来说意味着PQsocket标识的文件描述符上有可读的数据。当主循环检测到输入准备好时，它将调用PQconsumeInput 读取输入。然后它可以调用PQisBusy，如果PQisBusy返回假(0)则接着调用PQgetResult。它还可以调用PQnotifies检测NOTIFY消息（见Section 32.8）。

一个使用PQsendQuery/PQgetResult的客户端也可以尝试取消一个正在被服务器处理的命令，见Section 32.6 但是，不管PQcancel的返回值是什么，应用都必须继续使

用PQgetResult进行正常的结果读取序列。一次成功的取消只会导致命令比不取消时更快终止。

通过使用上述函数，我们可以避免在等待来自数据库服务器的输入时被阻塞。不过，在应用发送输出给服务器时还是可能出现阻塞。这种情况比较少见，但是如果发送非常长的 SQL 命令或者数据值时确实可能发生（不过，最有可能是应用通过COPY IN发送数据时）。为了避免这种可能性并且实现完全地非阻塞数据库操作，可以使用下列附加函数。

#### PQsetnonblocking

把连接的状态设置为非阻塞。

```
int PQsetnonblocking(PGconn *conn, int arg);
```

如果arg为 1，把连接状态设置为非阻塞；如果arg为 0，把连接状态设置为阻塞。如果OK 返回 0，如果错误返回 -1。

在非阻塞状态，调用 PQsendQuery, PQputline, PQputnbytes, PQputCopyData, 和 PQendcopy将不会阻塞，但是如果它们需要被再次调用则会返回一个错误。

注意PQexec不会遵循任何非阻塞模式；如果调用PQexec，那么它的行为总是阻塞的。

#### PQisnonblocking

返回数据库连接的阻塞状态。

```
int PQisnonblocking(const PGconn *conn);
```

如果连接被设置为非阻塞状态，返回 1，如果是阻塞状态返回 0。

#### PQflush

尝试把任何正在排队的输出数据刷到服务器。如果成功（或者发送队列为空）返回 0，如果因某种原因失败则返回 -1，或者如果还无法把发送队列中的所有数据都发送出去，则返回 1（这种情况只在连接为非阻塞时候才会发生）。

```
int PQflush(PGconn *conn);
```

在一个非阻塞连接上发送任何命令或者数据之后，要调用PQflush。 如果它返回 1，就要等待套接字变成读准备好或写准备好。如果它变为写准备好，应再次调用PQflush。 如果它变为读准备好，则应先调用PQconsumeInput，然后再调用PQflush。 一直重复直到PQflush返回 0（有必要检查读准备好并且用PQconsumeInput 耗尽输入，因为服务器可能阻塞给我们发送数据的尝试，例如 NOTICE 消息，并且在读它的数据之前它都不会读我们的数据）。一旦PQflush返回 0，应等待套接字变成读准备好并且接着按照上文所述读取响应。

## 32.5. 一行一行地检索查询结果

通常，libpq会收集一个 SQL 命令的整个结果并且把它作为单个PGresult返回给应用。这对于返回大量行的命令是行不通的。对于这类情况，应用可以使用PQsendQuery和PQgetResult的单行模式。在这种模式中，结果行以一次一行的方式被返回给应用。

要进入到单行模式，需要在成功调用PQsendQuery(或兄弟函数)之后立即调用PQsetSingleRowMode。这种模式选择只对当前正在执行的查询有效。然后反复调用PQgetResult，直到它返回空，如Section 32.4中所示。如果该查询返回行，它们会作为单个的PGresult对象返回，它们看起来都像普通的查询结果，只不过其状态代码

是PGRES\_SINGLE\_TUPLE而非PGRES\_TUPLES\_OK。 在最后一行之后或者紧接着该查询返回零行之后，一个状态为PGRES\_TUPLES\_OK的零行对象会被返回，这就是代表不会有更多行的信号（但是注意仍然有必要继续调用PQgetResult直到它返回空）。 所有这些PGresult对象将包含相同的行描述数据（列名、类型等等），这些数据和普通一个查询的PGresult对象的不同。 每一个对象都应该按常规用PQclear释放。

#### PQsetSingleRowMode

为当前正在执行的查询选择单行模式。

```
int PQsetSingleRowMode(PGconn *conn);
```

这个函数只能在调用PQsendQuery或一个其兄弟函数之后立刻调用，并且要在任何连接上的其他操作之前调用，例如PQconsumeInput 或PQgetResult。 如果在正确的时间被调用，该函数会为当前查询激活单行模式并且返回 1。 否则模式会保持不变并且该函数返回 0。 在任何情况下，当前查询结束之后模式都会恢复到正常。

#### Caution

在处理一个查询时，服务器可能返回一些行并且接着遇到一个错误导致查询被中断。通常，libpq会丢弃掉这样的行并且至报告错误。但是在单行模式中，那些行（错误之前返回的行）已经被返回给应用。因此，应用将看到一些PGRES\_SINGLE\_TUPLE PGresult对象并且然后看到一个PGRES\_FATAL\_ERROR对象。为了得到正确的事务行为，如果查询最终失败，应用必须被设计为丢弃或者撤销使用之前处理的行完成的事情。

## 32.6. 取消进行中的查询

一个客户端应用可以使用本节描述的函数请求取消一个仍在被服务器处理的命令。

#### PQgetCancel

创建一个数据结构，这个数据结构包含取消一个通过特定数据库连接发出的命令所需要的信息。

```
PGcancel *PQgetCancel(PGconn *conn);
```

PQgetCancel 创建一个PGcancel 对象，给出一个 PGconn 连接对象。他将返回NULL，如果给出的conn 为 NULL 或是一个无效的连接。 The PGcancel 对象是一个不透明结构，不应该被应用所直接访问。它只能够传递到 PQcancel 或 PQfreeCancel。

#### PQfreeCancel

释放一个由PQgetCancel创建的数据结构。

```
void PQfreeCancel(PGcancel *cancel);
```

PQfreeCancel释放一个之前由PQgetCancel创建的数据对象。

#### PQcancel

要求服务器放弃当前命令的处理。



```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

如果取消请求成功发送，则返回值为 1，否则为 0。如果不成功，则errbuf会被填充一个解释性的错误消息。errbuf必须是一个尺寸为errbufsize的字符数组（推荐尺寸为 256 字节）。，解释为何不成功。errbuf必须是一个大小为errbufsize的 char 数组（建议大小为 256 字节）。

不过，成功的发送并不保证请求会有任何效果。如果取消有效，那么当前的命令将提前终止并且返回一个错误结果。如果取消失败（也就是说，因为服务器已经完成命令的处理），那么就根本不会有可见的结果。

如果errbuf是信号处理器中的一个局部变量，PQcancel可以从一个信号处理器中安全地调用。在PGcancel有关的范围内，PQcancel都是只读的，因此也可以在一个从操纵PGconn对象的线程中独立出来的线程中调用它。

### PQrequestCancel

PQrequestCancel是PQcancel的一个被废弃的变体。

```
int PQrequestCancel(PGconn *conn);
```

要求服务器放弃当前命令的处理。它直接在PGconn对象上进行操作，并且如果失败，就会在PGconn对象里存储错误消息（因此可以用PQerrorMessage检索出来）。尽管功能相同，这个方法在多线程程序和信号处理器里会带来危险，因为它可能覆盖PGconn的错误消息，进而将当前连接上正在处理的操作搞乱。

## 32.7. 快速路径接口

LightDB提供一种快速路径接口来向服务器发送简单的函数调用。

### Tip

这个接口在某种程度上已被废弃，因为我们可以通过创建一个定义该函数调用的预备语句来达到类似或者更强大的功能。然后，用参数和结果的二进制传输执行该语句，从而取代快速函数调用。

函数PQfn请求通过快速路径接口执行服务器函数。

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);
```

```
typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
};
```

```
} PQArgBlock;
```

`fnid`参数是要被执行的函数的 `OID`。`args`和`nargs`定义了要传递给函数的参数；它们必须匹配已声明的函数参数列表。当一个参数结构的`isint`域为真时，`u.integer`值被以指定长度（必须是 1、2 或者 4 字节）整数的形式发送给服务器；这时候会发生恰当的字节交换。当`isint`为假时，`*u.ptr`中指定数量的字节将不做任何处理被发送出去；这些数据必须是服务器预期的用于该函数参数数据类型的二进制传输的格式（由于历史原因`u.ptr`被声明为类型`int *`，其实把它考虑成`void *`会更好）。`result_buf`是放置该函数返回值的缓冲区。调用者必须已经分配了足够的空间来存储返回值（这里没有检查！）。实际的结果长度将被放在`result_len`指向的整数中返回。如果预期结果是 2 或 4 字节整数，把`result_is_int`设为 1；否则设为 0。把`result_is_int`设为 1 导致`libpq`在必要时对值进行交换字节，这样它就作为对客户机器正确的`int`值被传输，注意对任一种允许的结果大小都会传递一个 4 字节到`*result_buf`。当`result_is_int`是 0 时，服务器发送的二进制格式字节将不做修改直接返回（在这种情况下，把`result_buf`考虑为类型`void *`更好）。

`PQfn`总是返回一个有效的`PGresult`指针。在使用结果之前应该检查结果状态。当结果不再使用后，调用者有义务使用`PQclear`释放`PGresult`。

要将 `NULL` 参数传递给函数，请将该参数结构的 `len` 字段设置为 `-1`；此时 `isint` 和 `u` 字段无关紧要。（但这仅适用于协议 3.0 及更高版本的连接。）

如果函数返回 `NULL`，则将 `*result_len` 设置为 `-1`，并且不修改 `*result_buf`。（这仅适用于协议 3.0 及更高版本的连接；在协议 2.0 中，`*result_len` 和 `*result_buf` 都不会被修改。）

注意我们没办法处理空参数、空结果，也没办法在使用这个接口时处理集值结果。

## 32.8. 异步提示

LightDB通过`LISTEN`和`NOTIFY`命令提供了异步通知。一个客户端会话用`LISTEN`命令在一个特定的通知频道中注册它感兴趣的通知（也可以用`UNLISTEN`命令停止监听）。当任何会话执行一个带有特定频道名的`NOTIFY`命令时，所有正在监听该频道的会话会被异步通知。可以传递一个“载荷”字符串来与监听者沟通附加的数据。

`libpq`应用把`LISTEN`、`UNLISTEN`和`NOTIFY`命令作为通常的 `SQL` 命令提交。随后通过调用`PQnotifies`来检测`NOTIFY`消息的到达。

函数`PQnotifies`来自服务器的未处理通知消息列表中返回下一个通知。如果没有待处理的信息则返回一个空指针。一旦`PQnotifies`返回一个通知，该通知会被认为已处理并且将从通知列表中删除。

```
PGnotify *PQnotifies(PGconn *conn);
```

```
typedef struct pgNotify
{
    char *relname;           /* notification channel name */
    int be_pid;             /* process ID of notifying server process */
    char *extra;            /* notification payload string */
} PGnotify;
```

在处理完`PQnotifies`返回的`PGnotify`对象后，别忘了用`PQfreemem`把它释放。释放`PGnotify`指针就足够了；`relname`和`extra`域并不代表独立分配的内存（这些域的名称是历史性的，尤其是频道名称与关系名称没有什么联系）。

Example 32. 给出了一个例子程序展示异步通知的使用。

`PQnotifies`实际上并不从服务器读取数据；它只是返回被另一个`libpq`函数之前吸收的消息。在以前的`libpq`版本中，及时收到`NOTIFY`消息的唯一方法是持续地提交命令，即使是空

命令也可以，并且在每次PQexec后检查PQnotifies。虽然这个方法还能用，但是由于太过浪费处理能力已被废弃。

当你没有可用的命令提交时，一种更好的检查NOTIFY消息的方法是调用PQconsumeInput，然后检查PQnotifies。你可以使用select()来等待服务器数据到达，这样在无事做时可以不浪费CPU能力（参考PQsocket来获得用于select()的文件描述符）。注意不管是用PQsendQuery/PQgetResult提交命令还是简单地使用PQexec，这种方法都能正常工作。不过，你应该记住在每次PQgetResult或PQexec之后检查PQnotifies，看看在命令的处理过程中是否有通知到达。

## 32.9. COPY命令相关的函数

LightDB中的COPY命令有用于libpq的对网络连接读出或者写入的选项。这一节描述的函数允许应用通过提供或者消耗已拷贝的数据来充分利用这个功能。

整个处理是应用首先通过PQexec或者一个等效的函数发出SQL COPY命令。对这个命令的响应（如果命令无误）将是一个状态代码是PGRES\_COPY\_OUT或者PGRES\_COPY\_IN（取决于指定的拷贝方向）的PGresult对象。应用然后就应该使用这一节的函数接收或者传送数据行。在数据传输结束之后，另外一个PGresult对象会被返回以表明传输的成功或者失败。它的状态将是：PGRES\_COMMAND\_OK表示成功，PGRES\_FATAL\_ERROR表示发生了一些问题。此时我们可以通过PQexec发出进一步的SQL命令（在COPY操作的处理过程中，不能用同一个连接执行其它SQL命令）。

如果一个COPY命令是通过PQexec在一个可能包含额外命令的字符串中发出的，那么应用在完成COPY序列之后必须继续用PQgetResult取得结果。只有在PQgetResult返回NULL时，我们才能确信PQexec的命令字符串已经处理完毕，并且可以安全地发出更多命令。

这一节的函数应该只在从PQexec或PQgetResult获得了PGRES\_COPY\_OUT或PGRES\_COPY\_IN结果状态的后执行。

一个承载了这些状态值之一的PGresult对象携带了正在开始的COPY操作的一些额外数据。这些额外的数据可以用于那些与带查询结果的连接一起使用的函数：

**PQnfields**

返回要拷贝的列（域）的个数。

**PQbinaryTuples**

0 表示整体拷贝格式都是文本（行用新行分隔，列用分隔字符分隔等等）。1 表示整体拷贝格式都是二进制。详见COPY。

**PQfformat**

返回与拷贝操的每列相关的格式代码（0 是文本，1 是二进制）。当整体拷贝格式是文本时，那么每列的格式代码将总是零，但是二进制格式可以同时支持文本和二进制列（不过，就目前的COPY实现而言，二进制拷贝中只会出现二进制列；所以目前每列的格式总是匹配总体格式）。

### Note

这些额外的数据值只在使用协议 3.0 时可用。在使用协议 2.0 时，所有这些函数都返回 0。

### 32.9.1. 用于发送COPY数据的函数

这些函数用于在COPY FROM STDIN期间发送数据。如果在连接不是COPY\_IN状态，调用它们会失败。

### PQputCopyData

在COPY\_IN状态中向服务器发送数据。

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

传输指定buffer中长度为nbytes的COPY数据到服务器。如果数据被放在队列中，结果是 1；如果因为缓冲区满而无法被放在队列中（只可能发生在连接是非阻塞模式时），那么结果是零；如果发生错误，结果为 -1（如果返回值为 -1，那么使用PQerrorMessage检索细节。如果值是零，那么等待写准备好然后重试）。

应用可以把COPY数据流划分成任意方便的大小放到缓冲区中。在发送时，缓冲区载荷的边界没有什么语意。数据流的内容必须匹配COPY命令预期的数据格式；详见COPY。

### PQputCopyEnd

在COPY\_IN状态中向服务器发送数据结束的指示。

```
int PQputCopyEnd(PGconn *conn,
                  const char *errmsg);
```

如果errmsg是NULL，则成功结束COPY\_IN操作。如果errmsg不是NULL则COPY被强制失败，errmsg指向的字符串是错误消息（不过，我们不应假定这个准确的错误信息将会从服务器传回，因为服务器可能已经因为其自身原因导致COPY失败。还要注意的是在使用 3.0 协议之前的连接时，强制失败的选项是不能用的）。

如果终止消息被发送，则结果为 1；在非阻塞模式中，结果为 1 也可能只表示终止消息被成功地放在了发送队列中（在非阻塞模式中，要确认数据确实被发送出去，你应该接着等待写准备好并且调用PQflush，重复这些直到返回零）。零表示该函数由于缓冲区满而无法将该终止消息放在队列中，这只会发生在非阻塞模式中（在这种情况下，等待写准备好并且再次尝试PQputCopyEnd调用）。如果发生系统错误，则返回 -1，可以使用PQerrorMessage检索详情。

在成功调用PQputCopyEnd之后，调用PQgetResult获取COPY命令的最终结果状态。我们可以用平常的方法来等待这个结果可用。然后返回到正常的操作。

## 32.9.2. 用于接收COPY数据的函数

这些函数用于在COPY TO STDOUT的过程中接收数据。如果连接不在COPY\_OUT状态，那么调用它们将会失败。

### PQgetCopyData

在COPY\_OUT状态下从服务器接收数据。

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

在一个COPY期间尝试从服务器获取另外一行数据。数据总是以每次一个数据行的方式被返回；如果只有一个部分行可用，那么它不会被返回。成功返回一个数据行涉及到分配一块内存来保存该数据。buffer参数必须为非NULL。\*buffer被设置为指向分配到的内存的指针，或者是在没有返回缓冲区的情况下指向NULL。一个非NULL的结果缓冲区在不需要时必须用PQfreemem释放。

在成功返回一行之后，返回的值就是该数据行里数据的字节数（将是大于零）。被返回的字符串总是空终止的，虽然这可能只是对文本COPY有用。一个零结果表示该COPY仍然在处理中，但是还没有可用的行（只在async为真时才可能）。一个 -1 结果表示COPY已经完成。-2 结果表示发生了错误（参考PQerrorMessage获取原因）。

当async为真时（非零），PQgetCopyData将不会阻塞等待输入；如果COPY仍在处理过程中并且没有可用的完整行，那么它将返回零（在这种情况下等待读准备好，然后在再次调用PQgetCopyData之前，调用PQconsumeInput）。当async为假（零）时，PQgetCopyData将阻塞，直到数据可用或者操作完成。

在PQgetCopyData返回 -1 之后，调用PQgetResult获取COPY命令的最后结果状态。我们可以用平常的方法来等待这个结果可用。然后返回到正常的操作。

### 32.9.3. 用于COPY的废弃函数

这些函数代表了以前的处理COPY的方法。尽管它们还能用，但是现在已经被废弃，因为它们的错误处理很糟糕、检测结束数据的方法也不方便，并且缺少对二进制或非阻塞传输的支持。

#### PQgetline

读取一个以新行终止的字符行到（由服务器传输）到一个长度为length的字符串缓冲区。

```
int PQgetline(PGconn *conn,
             char *buffer,
             int length);
```

这个函数拷贝最多length-1 个字符到该缓冲区中，并且把终止的新行转换成一个零字节。PQgetline在输入结束时返回EOF，如果整行都被读取则返回 0，如果缓冲区填满了而还没有遇到结束的新行则返回 1。

注意，应用必须检查是否一个新行包含两个字符\，这表明服务器已经完成了COPY命令的结果发送。如果应用可能收到超过length-1 字符长的行，我们就应该确保正确识别\行（例如，不要把一个长数据行的结束当作一个终止行）。

#### PQgetlineAsync

不阻塞地读取一行COPY数据（由服务器传输）到一个缓冲区中。

```
int PQgetlineAsync(PGconn *conn,
                 char *buffer,
                 int bufsize);
```

这个函数类似于PQgetline，但是可以被用于那些必须异步读取COPY数据的应用，也就是不阻塞的应用。在发出了COPY命令并得到了PGRES\_COPY\_OUT响应之后，应用应该调用PQconsumeInput 和 PQgetlineAsync直到检测到结束数据的信号。

不像PQgetline，这个函数负责检测结束数据。

在每次调用时，如果libpq的输入缓冲区中有一个完整的数据行可用，PQgetlineAsync都将返回数据。否则，在剩余行到达之前不会返回数据。如果识别到拷贝数据结束的标志，此函数返回 -1；如果没有可用数据则返回 0；或者是给出一个正数给出被返回的字节数。如果返回 -1，调用者下一步必须调用PQendcopy，然后回到正常处理。

返回的数据将不超过一个数据行的范围。如果可能，每次将返回一个完整行。但如果调用者提供的缓冲区太小不足以容下服务器发送的行，那么将返回部分行。对于文本数

据，这可以通过测试返回的最后一个字节是否\n来检测（在二进制COPY中，需要对COPY数据格式进行实际的分析，以便做相同的判断）。被返回的字符串不是空结尾的（如果你想增加一个终止空，确保传递一个比实际可用空间少一字节的bufsize）。

### PQputline

向服务器发送一个空终止的字符串。如果 OK 则返回 0；如果不能发送字符串则返回EOF。

```
int PQputline(PGconn *conn,
              const char *string);
```

一系列PQputline调用发送的COPY数据流和PQgetlineAsync返回的数据具有相同的格式，只是应用不需要每次PQputline调用中发送刚好一个数据行；在每次调用中发送多行或者部分行都是可以的。

#### Note

在LightDB协议 3.0 之前，应用必须显式地发送两个字符\.作为最后一行来指示服务器已经完成发送COPY数据。虽然这么做仍然有效，但是它已经被废弃并且\的特殊含义可能在将来的版本中删除。在发送完实际数据之后，调用PQendcopy就足够了。

### PQputnbytes

向服务器发送一个非空终止的字符串。如果 OK 则返回 0，如果不能发送字符串则返回EOF。

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

这个函数类似PQputline，除了数据缓冲区不需要是空终止，因为要发送的字节数是直接指定的。在发送二进制数据时使用这个过程。

### PQendcopy

与服务器同步。

```
int PQendcopy(PGconn *conn);
```

这个函数等待服务器完成拷贝。当最后一个字符串已经用PQputline发送给服务器时或者当最后一个字符串已经用PQgetline从服务器接收到时，就会发出这个函数。这个函数必须被发出，否则服务器将会和客户端“不同步”。从这个函数返回后，服务器就已经准备好接收下一个 SQL 命令了。函数成功完成时返回值为 0，否则返回非零值（如果返回值为非零值，用PQerrorMessage检索详情）。

在使用PQgetResult时，应用应该通过反复调用PQgetline并且在看到终止行后调用PQendcopy来响应PGRES\_COPY\_OUT结果。然后它应该返回到PQgetResult循环直到PQgetResult返回一个空指针。类似地，PGRES\_COPY\_IN结果会用一系列PQputline加上之后的PQendcopy来处理，然后返回到PQgetResult循环。这样的安排将保证嵌入到一系列SQL命令中的COPY命令将被正确执行。

旧的应用很可能会通过PQexec提交一个COPY命令并且假定事务在PQendcopy之后完成。只有在COPY是命令字符串中唯一的SQL命令时才能正确工作。

## 32.10. 控制函数

这些函数控制libpq行为各种各样的细节。

### PQclientEncoding

返回客户端编码。

```
int PQclientEncoding(const PGconn *conn);
```

请注意，它返回的是编码 ID，而不是一个符号字符串，如EUC\_JP。如果不成功，它会返回 -1。要把一个编码 ID 转换为为一个编码名称，可以用：

```
char *pg_encoding_to_char(int encoding_id);
```

### PQsetClientEncoding

设置客户端编码。

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

conn是一个到服务器的连接，而encoding是你想使用的编码。如果函数成功地设置编码，则返回 0，否则返回 -1。这个连接的当前编码可以使用PQclientEncoding确定。

### PQsetErrorVerbosity

决定PQerrorMessage 和 PQresultErrorMessage返回的消息的细节程度。

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE,
    PQERRORS_SQLSTATE
} PGVerbosity;
```

```
PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

PQsetErrorVerbosity设置细节模式，并返回该连接的前一个设置。在TERSE模式下，返回的消息只包括严重性、主要文本以及位置；这些东西通常放在一个单一行中。DEFAULT模式生成的消息包括上面的信息加上任何细节、提示或者上下文域（这些可能跨越多行）。VERBOSE模式包括所有可以可用的域。修改细节模式不会影响来自已有PGresult对象中的可用消息。只有随后创建的PGresult对象才受到影响。SQLSTATE模式仅包括错误严重性和SQLSTATE错误代码，如果其中之一是可用的（如果没有，输出类似于TERSE模式）。

更改详细程度设置不会影响已存在的PGresult对象的可用的消息，只会影响随后创建的对象。（如果想要用不同的详细程度打印之前的错误，请见PQresultVerboseErrorMessage）

### PQsetErrorContextVisibility

决定如何处理PQerrorMessage 和 PQresultErrorMessage返回的消息中的CONTEXT域。

```
typedef enum
{
```

```

    PQSHOW_CONTEXT_NEVER,
    PQSHOW_CONTEXT_ERRORS,
    PQSHOW_CONTEXT_ALWAYS
} PGContextVisibility;

```

```

PGContextVisibility PQsetErrorContextVisibility(PGconn *conn,
    PGContextVisibility show_context);

```

`PQsetErrorContextVisibility` 设置上下文显示模式，返回该连接上之前的设置。这个模式控制消息中是否包括 `CONTEXT` 域。 `NEVER` 模式不会包括 `CONTEXT`，而 `ALWAYS` 则尽可能地包括这个域。在 `ERRORS` 模式（默认）中，只在错误消息中包括 `CONTEXT` 域，而在通知和警告消息中不会包括。（但是，如果详细程度设置为 `TERSE` 或 `SQLSTATE`，则无论上下文显示模式如何，都会省略 `CONTEXT` 字段。）

更改这个模式不会影响从已经存在的 `PGresult` 对象项中得到的消息，只会影响后续创建的 `PGresult` 对象（如果想要用不同的详细程度打印之前的错误，请见 `PQresultVerboseErrorMessage`）。

### PQtrace

启用对客户端/服务器通讯的跟踪，把跟踪信息输出到一个调试文件流中。

```
void PQtrace(PGconn *conn, FILE *stream);
```

#### Note

在 Windows 上，如果 `libpq` 库和应用使用了不同的标志编译，那么这个函数调用会导致应用崩溃，因为 `FILE` 指针的内部表达是不一样的。特别是多线程/单线程、发布/调试 以及静态/动态标志应该是库和所有使用库的应用都一致。

### PQuntrace

禁用 `PQtrace` 打开的跟踪。

```
void PQuntrace(PGconn *conn);
```

## 32.11. 杂项函数

一如往常，总有一些函数不适合放在任何其他地方。

### PQfreemem

释放 `libpq` 分配的内存。

```
void PQfreemem(void *ptr);
```

释放 `libpq` 分配的内存，尤其是 `PQescapeByteaConn`, `PQescapeBytea`, `PQunescapeBytea`, 和 `PQnotifies` 分配的内存。特别重要的是，在微软 Windows 上使用这个函数，而不是 `free()`。这是因为只有 `DLL` 和应用的当多线程/单线程、发布/调试以及静态/动态标志相同时，才能在一个 `DLL` 中分配内存并且在应用中释放它。在非微软 Windows 平台上，这个函数与标准库函数 `free()` 相同。

### PQconninfoFree

释放 `PQconnndefaults` 或 `PQconninfoParse` 分配的数据结构。



```
void PQconninfoFree(PQconninfoOption *connOptions);
```

一个简单的PQfreemem不会做这些，因为数组包含对子字符串的引用。

### PQencryptPasswordConn

准备一个LightDB口令的加密形式。

```
char *PQencryptPasswordConn(PGconn *conn, const char *passwd, const char
 *user, const char *algorithm);
```

这个函数旨在用于那些希望发送类似于ALTER USER joe PASSWORD 'pwd' 命令的客户端应用。不在这样一个命令中发送原始的明文密码是一个好习惯，因为它可能被暴露在命令日志、活动显示等等中。相反，在发送之前使用这个函数可以将口令转换为加密的形式。

passwd和user参数是明文口令以及用户的SQL名称。 algorithm指定用来加密口令的加密算法。 当前支持的算法是md5和scram-sha-256（on和off也被接受作为md5的别名，用于与较老的服务器版本兼容）。 注意，对scram-sha-256支持是在LightDB版本10中引入的，并且在老的服务器版本上无法工作。 如果algorithm是NULL，这个函数将向服务器查询password\_encryption设置的当前值。 这种行为可能会阻塞当前事务，并且当前事务被中止或者连接正忙于执行另一个查询时会失败。 如果希望为服务器使用默认的算法但避免阻塞，应在调用PQencryptPasswordConn之前查询你自己的password\_encryption，并且将该值作为algorithm传入。

返回值是一个由malloc分配的字符串。调用者可以假设该字符串不含有需要转义的任何特殊字符。 在处理完它之后，用PQfreemem释放结果。发生错误时，返回的是NULL，并且适当的消息会被存储在连接对象中。

### PQencryptPassword

准备一个LightDB口令的md5加密形式。

```
char *PQencryptPassword(const char *passwd, const char *user);
```

PQencryptPassword是PQencryptPasswordConn的一个较老的已经被废弃的版本。其差别是PQencryptPassword不要求一个连接对象，并且总是用md5作为加密算法。

### PQmakeEmptyPGresult

用给定的状态，构造一个空PGresult对象。

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

这是libpq内部用于分配并初始化一个空PGresult对象的函数。 如果不能分配内存，那么这个函数返回NULL。 它也是可以对外使用的，因为一些应用认为它可以用于产生结果对象（特别是带有错误状态的对象）本身。 如果conn非空，并且status表示一个错误，那么指定连接的当前错误消息会被复制到PGresult中。 如果conn非空，那么连接中的任何已注册事件过程也会被复制到PGresult中（它们不会获得PGEVT\_RESULTCREATE调用，但会看到PQfireResultCreateEvents）。 注意在该对象上最终应该调用PQclear，正如对libpq本身返回的PGresult对象所作的那样。

### PQfireResultCreateEvents

为每一个在PGresult对象中注册的事件过程触发一个PGEVT\_RESULTCREATE事件（见Section 32.13）。成功时返回非0，如果任何事件过程失败则返回0。

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

conn参数被传送给事件过程，但不会被直接使用。如果事件过程不使用它，则会返回NULL。

已经接收到这个对象的PGEVT\_RESULTCREATE或PGEVT\_RESULTCOPY事件的事件过程不会被再次触发。

这个函数与PQmakeEmptyPGresult分开的主要原因是调用事件过程之前创建一个PGresult并且填充它常常是合适的。

### PQcopyResult

为一个PGresult对象创建一个拷贝。这个拷贝不会以任何方式链接到源结果，并且当该拷贝不再需要时，必须调用PQclear进行清理。如果函数失败，返回NULL。

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

这个函数的意图并非制作一个准确的拷贝。返回的结果总是会被放入PGRES\_TUPLES\_OK状态，并且不会拷贝来源中的任何错误消息（不过它确实会拷贝命令状态字符串）。flags参数决定还要拷贝些什么。它通常是几个标志的按位OR。PG\_COPYRES\_ATTRS指定复制源结果的属性（列定义）。PG\_COPYRES\_TUPLES指定复制源结果的元组（这也意味着复制属性）。PG\_COPYRES\_NOTICEHOOKS指定复制源结果的提醒钩子。PG\_COPYRES\_EVENTS指定复制源结果的事件（但是不会复制与源结果相关的实例数据）。

### PQsetResultAttrs

设置PGresult对象的属性。

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc
    *attDescs);
```

提供的attDescs被复制到结果中。如果attDescs指针为NULL或numAttributes小于1，那么请求将被忽略并且函数成功。如果res已经包含属性，那么函数会失败。如果函数失败，返回值是 0。如果函数成功，返回值是非 0。

### PQsetvalue

设置一个PGresult对象的一个元组域值。

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int
    len);
```

这个函数将自动按需增加结果的内置元组数组。但是，tup\_num参数必须小于等于PQntuples，意味着这个函数对元组数组一次只能增加一个元组。但已存在的任意元组中的任意域可以以任意顺序进行调整。如果field\_num的一个值已经存在，它会被覆盖。如果len是 -1，或value是NULL，该域值会被设置为一个SQL空值。value会被复制到结果的私有存储中，因此函数返回后就不再需要了。如果函数失败，返回值是 0。如果函数成功，返回值会是非 0。

### PQresultAlloc

为一个PGresult对象分配附属存储。

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

当res被清除时，这个函数分配的内存也会被释放掉。如果函数失败，返回值是NULL。结果被保证为按照数据的任意类型充分地对齐，正如malloc所作的。

#### PQresultMemorySize

检索为PGresult对象分配的字节数。

```
size_t PQresultMemorySize(const PGresult *res);
```

此值是与PGresult对象关联的所有malloc请求的总和，就是说，PQclear将释放的所有空间。此信息可用于管理内存消耗。

#### PQlibVersion

返回所使用的libpq版本。

```
int PQlibVersion(void);
```

在运行时，这个函数的结果可以被用来决定在当前已载入的 libpq 版本中特定的功能是否可用。例如，这个函数可以被用来决定哪些选项可以被用于PQconnectdb。

结果通过将库的主版本号乘以10000再加上次版本号形成。例如，版本10.1将被返回为100001，而版本11.0将被返回为110000。

在主版本10之前，LightDB采用一种由三个部分组成的版本号，其中前两部分共同表示主版本。对于那些版本，PQlibVersion为每个部分使用两个数字，例如版本9.1.5将被返回为90105，而版本9.2.0将被返回为90200。

因此，出于判断特性兼容性的目的，应用应该将PQlibVersion的结果除以100而不是10000来判断逻辑的主版本号。在所有的发行序列中，只有最后两个数字在次发行（问题修正发行）之间不同。

#### Note

这个函数出现于LightDB版本 9.1，因此它不能被用来在早期的版本中检测所需的功能，因为调用它将会创建一个对版本9.1及其后版本的链接依赖。

## 32.12. 通知处理

服务器产生的通知和警告消息不会被查询执行函数返回，因为它们不代表查询失败。它们可以被传递给一个通知处理函数，并且在处理器返回后执行会继续正常进行。默认的处理函数会把消息打印在stderr上，但是应用可以通过提供它自己的处理函数来重载这种行为。

由于历史原因，通知处理有两个级别，称为通知接收器和通知处理器。通知接收器的默认行为是格式化通知并且将一个字符串传递给通知处理器来打印。不过，如果一个应用选择提供自己的通知接收器，它通常会忽略通知处理器层并且在通知接收器中完成所有工作。

函数 `PQsetNoticeReceiver` 设置或检查连接对象的当前通知接收器。同样，`PQsetNoticeProcessor` 设置或检查当前的通知处理器。

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);
```

```
PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                   PQnoticeReceiver proc,
```

```

void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                    PQnoticeProcessor proc,
                    void *arg);

```

这些函数中的每一个会返回之前的通知接收器或处理器函数指针，并且设置新值。如果你提供了一个空函数指针，将不会采取任何动作，只会返回当前指针。

当接收到一个服务器产生的或者libpq内部产生的通知或警告消息，通知接收器函数会被调用。它会以一种PGRES\_NONFATAL\_ERROR PGresult的形式传递该消息（这允许接收器使用PQresultErrorField抽取个别的域，或者使用PQresultErrorMessage或者PQresultVerboseErrorMessage得到一个完整的预格式化的消息）。被传递给PQsetNoticeReceiver的同一个空指针也被传递（必要时，这个指针可以被用来访问应用相关的状态）。

默认的通知接收器会简单地抽取消息（使用PQresultErrorMessage）并且将它传递给通知处理器。

通知处理器负责处理一个以文本形式给出的通知或警告消息。该消息的字符串文本（包括一个收尾的新行）被传递给通知处理器，外加一个同时被传递给PQsetNoticeProcessor的空指针（必要时，这个指针可以被用来访问应用相关的状态）。

默认的通知处理器很简单：

```

static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}

```

一旦你设定了一个通知接收器或处理器，你应该期待只要PGconn对象或者从它构造出的PGresult对象存在，该函数就应该能被调用。在一个PGresult创建时，PGconn的当前通知处理指针被复制到PGresult中，以备类似PQgetvalue的函数使用。

## 32.13. 事件系统

libpq的事件系统被设计为通知已注册的事件处理器它感兴趣的libpq事件，例如PGconn以及PGresult对象的创建和毁灭。一种主要的使用情况是这允许应用将自己的数据与一个PGconn或者PGresult关联在一起，并且确保那些数据在适当的时候被释放。

每一个已注册的事件处理器与两部分数据相关，对于libpq它们只是透明的void \*指针。当事件处理器被注册到一个PGconn时，会有一个应用提供的转移指针。该转移指针在PGconn及其产生的所有PGresult的生命期内都不会改变。因此，如果使用它，它必须指向长期存在的数据。此外，还有一个instance data指针，它在每一个PGconn和PGresult中都开始于NULL。这个指针可以使用 PQinstanceData, PQsetInstanceData, PQresultInstanceData和 PQsetResultInstanceData函数操纵。注意和转移指针不同，一个PGconn的实例数据不会被从它创建的PGresult自动继承。libpq不知道转移和实例数据指针指向的是什么（如果有），并且将不会尝试释放它们 — 那是事件处理器的责任。

### 32.13.1. 事件类型

枚举PQEventId命名了事件系统处理的事件类型。它的所有值的名称都以PQEVT开始。对于每一种事件类型，都有一个相应的事件信息结构用来承载传递给事件处理器的参数。事件类型是：

## PGEVT\_REGISTER

当PQregisterEventProc被调用时，注册事件会发生。这是一个初始化每一个事件过程都可能需要的instanceData的最佳时机。每个连接的每个事件处理器只会触发一个注册事件。如果该事件过程失败，注册会被中止。

```
typedef struct
{
    PGconn *conn;
} PGEvtRegister;
```

当收到一个PGEVT\_REGISTER事件时，evtInfo指针应该被造型为PGEvtRegister \*。这个结构包含一个状态应该为CONNECTION\_OK的PGconn，保证在得到一个良好的PGconn之后能马上调用PQregisterEventProc。当返回一个失败代码时，所有的清理都必须被执行而不会发送PGEVT\_CONNDESTROY事件。

## PGEVT\_CONNRESET

连接重置事件在PQreset或PQresetPoll完成时被触发。在两种情况中，只有重置成功才会触发该事件。如果事件过程失败，整个连接重置将失败，PGconn会被置为CONNECTION\_BAD状态并且PQresetPoll将返回PGRES\_POLLING\_FAILED。

```
typedef struct
{
    PGconn *conn;
} PGEvtConnReset;
```

当收到一个PGEVT\_CONNRESET事件时，evtInfo指针应该被造型为PGEvtConnReset \*。尽管所包含的PGconn刚被重置，所有的事件数据还是保持不变。这个事件应该被用来重置/重载/重新查询任何相关的instanceData。注意即使事件过程无法处理PGEVT\_CONNRESET，它仍将在连接被关闭时接收到一个PGEVT\_CONNDESTROY事件。

## PGEVT\_CONNDESTROY

为了响应PQfinish，连接销毁事件会被触发。由于 libpq 没有能力管理事件数据，事件过程有责任正确地清理它的事件数据。清理失败将会导致内存泄露。

```
typedef struct
{
    PGconn *conn;
} PGEvtConnDestroy;
```

当接收到一个PGEVT\_CONNDESTROY事件时，evtInfo指针应该被造型为PGEvtConnDestroy \*。这个事件在PQfinish执行任何其他清理之前被触发。该事件过程的返回值被忽略，因为没有办法指示一个来自PQfinish的失败。还有，一个事件过程失败不该中断对不需要的内存的清理。

## PGEVT\_RESULTCREATE

为了响应任何生成一个结果的查询执行函数，结果创建事件会被触发。这些函数包括PQgetResult。这个事件只有在结果被成功地创建之后才会被触发。

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
```

```
} PGEventResultCreate;
```

当接收到一个PGEVT\_RESULTCREATE事件时，`evtInfo`指针应该被造型为`PGEventResultCreate *`。`conn`是用来产生结果的连接。这是初始化任何需要与结果关联的`instanceData`的理想位置。如果该事件过程失败，结果将被清除并且失败将会被传播。该事件过程不能尝试自己`PQclear`结果对象。当返回一个失败代码时，所有清理必须被执行而不会发送PGEVT\_RESULTDESTROY事件。

PGEVT\_RESULTCOPY

为了响应`PQcopyResult`，结果复制事件会被触发。这个事件只会在复制完成后才被触发。只有成功地处理了PGEVT\_RESULTCREATE和PGEVT\_RESULTCOPY事件的事件过程才会收到PGEVT\_RESULTCOPY事件。

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

当收到一个PGEVT\_RESULTCOPY事件时，`evtInfo`指针应该被造型为`PGEventResultCopy *`。`src`结果是要被复制的，而`dest`结果则是复制的目的地。这个事件可以被用来提供`instanceData`的一份深度副本，因为`PQcopyResult`没法这样做。如果该事件过程失败，整个复制操作将失败并且`dest`结果将被清除。当返回一个失败代码时，所有清理必须被执行而不会为目标结果发送PGEVT\_RESULTDESTROY事件。

PGEVT\_RESULTDESTROY

为了响应`PQclear`，结果销毁事件会被触发。由于 `libpq` 没有能力管理事件数据，事件过程有责任正确地清理它的事件数据。清理失败将会导致内存泄露。

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

当接收到一个PGEVT\_RESULTDESTROY事件时，`evtInfo`指针应该被造型为`PGEventResultDestroy *`。这个事件在`PQclear`执行任何其他清理之前被触发。该事件过程的返回值被忽略，因为没有办法指示来自`PQclear`的失败。还有，一个事件过程失败不该中断不需要的内存的清理过程。

## 32.13.2. 事件回调函数

PGEventProc

`PGEventProc`是到一个事件过程的指针的 `typedef`，也就是从 `libpq` 接收事件的用户回调函数。一个事件过程的原型必须是

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

`evtId`指示发生了哪一个PGEVT事件。`evtInfo`指针必须被造型为合适的结构类型才能获得关于事件的进一步信息。当事件过程已被注册时，`passThrough`参数是提供给`PQregisterEventProc`的指针。如果成功，该函数应该返回非零值，失败则返回零。

在任何一个`PGconn`中，一个特定事件过程只能被注册一次。这是因为该过程的地址被用作查找键来标识相关的实例数据。

### Caution

在 Windows 上，函数能够有两个不同的地址：一个对 DLL 之外可见而另一个对 DLL 之内可见。我们应当小心只有其中之一会被用于 libpq 的事件过程函数，否则将会产生混淆。编写代码的最简单规则是将所有的事件过程声明为 `static`。如果过程的地址必须对它自己的源代码文件之外可见，提供一个单独的函数来返回该地址。

## 32.13.3. 事件支持函数

### PQregisterEventProc

为 libpq 注册一个事件回调过程。

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                      const char *name, void *passThrough);
```

在每一个你想要接收事件的 PGconn 上必须注册一个事件过程。和内存不同，没有限制说一个连接上能注册多少个事件过程。如果该函数成功，它会返回一个非零值。如果它失败，则会返回零。

当一个 libpq 事件被触发时，proc 参数将被调用。它的内存地址也被用来查找 instanceData。name 参数被用来在错误消息中引用该事件过程。这个值不能是 NULL 或一个零长度串。名字串被复制到 PGconn 中，因此传递进来的东西不需要长期存在。当一个事件发生时，passThrough 指针被传递给 proc。这个参数可以是 NULL。

### PQsetInstanceData

设置连接 conn 的用于过程 proc 的 instanceData 为 data。它在成功时返回非零值，失败时返回零（只有 proc 没有被正确地注册在 conn 中，才可能会失败）。

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

### PQinstanceData

返回连接 conn 的与过程 proc 相关的 instanceData，如果没有则返回 NULL。

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

### PQresultSetInstanceData

把结果的用于 proc 的 instanceData 设置为 data。成功返回非零，失败返回零（只有 proc 没有被正确地注册在 conn 中，才可能会失败）。

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

请注意，data 表示的任何存储都不会由 PQresultMemorySize 考虑，除非使用 PQresultAlloc 分配。（这样做是值得推荐的，因为它消除了销毁结果时显式释放此类存储的需要。）

### PQresultInstanceData

返回结果的与过程 proc 相关的 instanceData，如果没有则返回 NULL。

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

## 32.13.4. 事件实例

这里是一个管理与 libpq 连接和结果相关的私有数据的例子的框架。

```
/* 要求 libpq 事件的头文件（注意：包括 libpq-fe.h） */
#include <libpq-events.h>

/* instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn =
        PQconnectdb("dbname=postgres options=-csearch_path=");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    /* 在任何应该接收事件的连接上调用一次。
     * 发送一个 PGEVT_REGISTER 给 myEventProc。
     */
    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
    }

    /* conn 的 instanceData 可用 */
    data = PQinstanceData(conn, myEventProc);

    /* 发送一个 PGEVT_RESULTCREATE 给 myEventProc */
    res = PQexec(conn, "SELECT 1 + 1");

    /* 结果的 instanceData 可用 */
    data = PQresultInstanceData(res, myEventProc);

    /* 如果使用了 PG_COPYRES_EVENTS, 发送一个 PGEVT_RESULTCOPY 给 myEventProc */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);
}
```



```
/* 如果在 PQcopyResult 调用时使用了 PG_COPYRES_EVENTS, 结果的 instanceData
可用。*/
data = PQresultInstanceData(res_copy, myEventProc);

/* 两个清除都发送一个 PGEVT_RESULTDESTROY 给 myEventProc */
PQclear(res);
PQclear(res_copy);

/* 发送一个 PGEVT_CONNDESTROY 给 myEventProc */
PQfinish(conn);

return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            /* 将应用相关的数据与连接关联起来 */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
                memset(data, 0, sizeof(mydata));
            break;
        }

        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            /* 因为连接正在被销毁, 释放示例数据 */
            if (data)
                free_mydata(data);
            break;
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn, myEventProc);
            mydata *res_data = dup_mydata(conn_data);

            /* 把应用相关的数据与结果 (从 conn 复制过来) 关联起来 */
            PQsetResultInstanceData(e->result, myEventProc, res_data);
        }
    }
}
```

```

        break;
    }

    case PGEVT_RESULTCOPY:
    {
        PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
        mydata *src_data = PQresultInstanceData(e->src, myEventProc);
        mydata *dest_data = dup_mydata(src_data);

        /* 把应用相关的数据与结果（从一个结果复制过来）关联起来 */
        PQsetResultInstanceData(e->dest, myEventProc, dest_data);
        break;
    }

    case PGEVT_RESULTDESTROY:
    {
        PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
        mydata *data = PQresultInstanceData(e->result, myEventProc);

        /* 因为结果正在被销毁，释放实例数据 */
        if (data)
            free_mydata(data);
        break;
    }

    /* 未知事件 ID，只返回true。 */
    default:
        break;
}

return true; /* 事件处理成功 */
}

```

## 32.14. 环境变量

下列环境变量能被用于选择默认的连接参数值，如果调用代码没有直接指定值，它们将被用于PQconnectdb, PQsetdbLogin 和 PQsetdb。例如，这些有助于防止数据库连接信息被硬编码到简单的客户端应用中。

- LTHOST的行为和host连接参数相同。
- LTHOSTADDR的行为和hostaddr连接参数相同。可以设置它来替代或者作为LTHOST的补充来防止 DNS 查找负担。
- LTPORT的行为和port连接参数相同。
- LTDATABASE的行为和dbname连接参数相同。
- LTUSER的行为和user连接参数相同。
- LTPASSWORD的行为和password连接参数相同。出于安全原因，我们不推荐使用这个环境变量，因为某些操作系统允许非根用户通过ps看到进程的环境变量。可以考虑使用一个口令文件（见Section 32.15）。
- LTPASSFILE的行为和passfile连接参数相同。
- LTCHANNELBINDING 的行为和 channel\_binding 连接参数相同。
- LTSERVICE的行为和service连接参数相同。

- `LTSERVICEFILE`指定针对每个用户的连接服务文件名。如果没有设置，默认为`~/pg_service.conf`（见Section 32.10）。
- `LTOPTIONS`的行为和`options`连接参数相同。
- `LTAPPNAME`的行为和`application_name`连接参数相同。
- `LTSSLMODE`的行为和`sslmode`连接参数相同。
- `LTREQUIRESSL`的行为和`requiressl`连接参数相同。为了支持`PGSSLMODE`变量，这个环境变量已被废弃。同时设置两个变量会抑制这一个的效果。
- `LTSSLCOMPRESSION`的行为和`sslcompression`连接参数相同。
- `LTSSLCERT`的行为和`sslcert`连接参数相同。
- `LTSSLKEY`的行为和`sslkey`连接参数相同。
- `LTSSLROOTCERT`的行为和`sslrootcert`连接参数相同。
- `LTSSLCRL`的行为和`sslcr1`连接参数相同。
- `LTREQUIREPEER`的行为和`requirepeer`连接参数相同。
- `LTSSLMINPROTOCOLVERSION` 的行为和 `ssl_min_protocol_version` 连接参数相同。
- `LTSSLMAXPROTOCOLVERSION` 行为和 `ssl_min_protocol_version` 连接参数相同。
- `LTGSSENCMODE` 的行为和 `gssencmode`连接参数相同。
- `LTKRBSRVNAME`的行为和`krbsrvname`连接参数相同。
- `LTGSSLIB`的行为和`gsslib`连接参数相同。
- `LTCONNECT_TIMEOUT`的行为和`connect_timeout`连接参数相同。
- `LTCLIENTENCODING`的行为和`client_encoding`连接参数相同。
- `LTTARGETSESSIONATTRS`的行为和`target_session_attrs`连接参数相同。

下面的环境变量可用来为每一个LightDB会话指定默认行为（为每一个用户或每一个数据库设置默认行为的方法还可见`ALTER ROLE`和`ALTER DATABASE`命令）。

- `LTDATESTYLE`设置日期/时间表示的默认风格（等同于`SET datestyle TO ...`）。
- `LTTZ`设置默认的时区（等同于`SET timezone TO ...`）。
- `LTGEQO`为遗传查询优化器设置默认模式（等同于`SET geqo TO ...`）。

这些环境变量的正确值可参考SQL 命令 `SET`。

下面的环境变量决定libpq的内部行为，它们会覆盖编译在程序中的默认值。

- `LTSYSCONFDIR`设置包含`pg_service.conf`文件以及未来版本中可能出现的其他系统范围配置文件的目录。
- `LTLOCALEDIR`设置包含用于消息本地化的`locale`文件的目录。

## 32.15. 口令文件

一个用户主目录中的`.pgpass`文件能够包含在连接需要时使用的口令（并且其他情况不会指定口令）。在微软的 Windows 上该文件被命名为`%APPDATA%\postgresql\pgpass.conf`（其中`%APPDATA%`指的是用户配置中的应用数据子目录）。另外，可以使用连接参数`passfile`或者环境变量`LTPASSFILE`指定一个口令文件。

这个文件应该包含下列格式的行：

```
hostname:port:database:username:password
```

（你可以向该文件增加一个提醒：把上面的行复制到该文件并且在前面加上#）。前四个域的每一个都可以是文字值或者匹配任何东西的\*。第一个匹配当前连接参数的行中的口令域将被使用（因此，在使用通配符时把更特殊的项放在前面）。如果一个条目需要包含:或者\，用\对该字符转义。如果指定了host连接参数，主机名字段会被匹配到host，否则如果指定了hostaddr参数则匹配到hostaddr，如果两者都没有给出，则会搜索主机名localhost。当连接是一个Unix域套接字连接并且host参数匹配libpq的默认套接字目录路径时，也会搜索主机名localhost。在一台后备服务器上，值为replication的数据库字段匹配连接到主服务器的里复制连接。否则数据库字段的用途有限，因为用户对同一个集簇中的所有数据库都有相同的口令。

在 Unix 系统上，口令文件上的权限必须不允许所有人或组内访问，可以用 `chmod 0600 ~/.pgpass` 这样的命令实现。如果权限没有这么严格，该文件将被忽略。在微软 Windows 上，该文件被假定存储在一个安全的目录中，因此不会进行特别的权限检查。

## 32.16. 连接服务文件

连接服务文件允许 libpq 连接参数与一个单一服务名称关联。那个服务名称可以被一个 libpq 连接指定，与其相关的设置将被使用。这允许在不重新编译 libpq 应用的前提下修改连接参数。服务名称也可以被使用PGSERVICE环境变量来指定。

连接服务文件可以是每个用户都有一个的服务文件，它位于 `~/.pg_service.conf` 或者环境变量 `PGSERVICEFILE` 指定的位置。它也可以是一个系统范围的文件，位于 `lt_config --sysconfdir/pg_service.conf` 的或者环境变量 `PGSYSCONFDIR` 指定的目录。如果相同名称的服务定义存在于用户和系统文件中，用户文件将优先考虑。

该文件使用一种“INI 文件”格式，其中小节名是服务名并且参数是连接参数。列表见Section 32.1.2 例如：

```
# comment
[mydb]
host=somehost
port=5433
user=admin
```

`share/pg_service.conf.sample` 中提供了一个例子文件。

## 32.17. 连接参数的 LDAP 查找

如果libpq已经在编译时打开了 LDAP 支持（configure的选项`--with-ldap`），就可以通过LDAP 从一个中央服务器检索host或dbname之类的连接参数。这样做的好处是如果一个数据库的连接参数改变，不需要在所有的客户端机器上更新连接信息。

LDAP 连接参数查找使用连接服务文件 `pg_service.conf`（见Section 32.16）。`pg_service.conf` 中一个以 `ldap://` 开始的行将被识别为一个 LDAP URL 并且将执行一个 LDAP 查询。结果必须是一个 `keyword = value` 对列表，它将被用来设置连接选项。URL 必须遵循 RFC 1959 并且是形式

```
ldap://[hostname[:port]]/search_base?attribute?search_scope?filter
```

其中hostname默认为localhost并且port默认为 389。

一次成功的 LDAP 查找后，`pg_service.conf` 的处理被终止。但是如果联系不上 LDAP 则会继续处理 `pg_service.conf`。这就提供了后手，可以加入更多指向不同 LDAP 服务器的 LDAP

URL 行、经典的keyword = value对或者默认连接选项。如果你宁愿在这种情况下得到一个错误消息，在该 LDAP URL 之后增加一个语法错误的行。

一个和 LDIF 文件一起创建的 LDAP 条目实例

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:device
cn:mydatabase
description:host=dbserver.mycompany.com
description:port=5439
description:dbname=mydb
description:user=mydb_user
description:sslmode=require
```

可以用下面的 LDAP URL 查询：

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?description?one?(cn=mydatabase)
```

你也可以将常规的服务文件条目和 LDAP 查找混合。pg\_service.conf中一节的完整例子：

```
# 只有主机和端口存储在LDAP中，显式指定dbname和user。
[customerdb]
dbname=customer
user=appuser
ldap://ldap.acme.com/cn=dbserver,cn=hosts?pgconnectinfo?base?(objectclass=*)
```

## 32.18. SSL 支持

LightDB本地支持使用SSL连接加密客户端/服务器通信以提高安全性。关于服务器端的SSL功能详见Section 16.9

libpq读取系统范围的OpenSSL配置文件。默认情况下，这个文件被命名为openssl.cnf并且位于openssl version -d所报告的目录中。可以通过设置环境变量OPENSSL\_CONF把这个默认值覆盖为想要的配置文件的名称。

### 32.18.1. 服务器证书的客户端验证

默认情况下，LightDB将不会执行服务器证书的任何验证。这意味着可以在不被客户端知晓的情况下伪造服务器身份（例如通过修改一个 DNS 记录或者接管服务器的 IP 地址）。为了阻止哄骗，客户端必须能够通过一条信任链验证服务器的身份。信任链可以这样建立：在一台计算机上放置一个根（自签名的）证书机构（CA）的证书并且在另一台计算机上放置一个由根证书签发的叶子证书。还可以使用一种“中间”证书，它由根证书签发并且可以签发叶子证书。

为了允许客户端验证服务器的身份，在客户端上放置一份根证书并且在服务器上放置由根证书签发的叶子证书。为了允许服务器验证客户端的身份，在服务器上放置一份根证书并且在客户端上放置由根证书签发的叶子证书。也可以使用一个或者更多个中间证书（通常与叶子证书存在一起）来将叶子证书链接到根证书。

一旦信任链被建立起来，客户端有两种方法验证服务器发过来的叶子证书。如果参数sslmode被设置为verify-ca，libpq将通过检查该证书是否链接到存储在客户端上的根证书来验证服务器。如果sslmode被设置为verify-full，libpq还将验证服务器的主机名匹配

存储在服务器证书中的名称。如果服务器证书无法被验证，则SSL连接将失败。在大部分对安全性很敏感的环境中，推荐使用verify-full。

在verify-full模式中，主机名被拿来与证书的主体别名属性匹配，或者在不存在类型dNSName的主体别名时与通用名称属性匹配。如果证书的名称属性以一个星号(\*)开始，这个星号将被视作一个通配符，它将匹配所有除了句点(.)之外的字符。这意味着该证书将不会匹配子域。如果连接是使用一个IP地址而不是一个主机名创建的，该IP地址将被匹配（不做任何DNS查找）。

要允许服务器证书验证，必须将一个或者更多个根证书放置在用户主目录下的~/.postgresql/root.crt文件中（在Microsoft Windows上该文件名为%APPDATA%\postgresql\root.crt）。如果需要把服务器发来的证书链链接到存储在客户端的根证书，还应该将中间证书加到该文件中。

如果文件~/.postgresql/root.crl存在（微软Windows上的%APPDATA%\postgresql\root.crl），证书撤销列表（CRL）项也会被检查。

根证书文件和CRL的位置可以通过设置连接参数sslrootcert和sslerl或环境变量PGSSLROOTCERT和PGSSLCRL改变。

### Note

为了与LightDB的早期版本达到向后兼容，如果存在一个根CA文件，sslmode=require的行为将与verify-ca相同，即服务器证书根据CA验证。我们鼓励依赖这种行为，并且需要证书验证的应用应该总是使用verify-ca或者verify-full。

## 32.18.2. 客户端证书

如果服务器尝试通过请求客户端的叶子证书来验证客户端的身份，libpq将发送用户主目录下文件~/.postgresql/postgresql.crt中存储的证书。该证书必须链接到该服务器信任的根证书。也必须存在一个匹配的私钥文件~/.postgresql/postgresql.key。该私钥文件不能允许全部用户或者组用户的任何访问，可以通过命令chmod 0600 ~/.postgresql/postgresql.key实现。在微软Windows上这些文件被命名为%APPDATA%\postgresql\postgresql.crt和%APPDATA%\postgresql\postgresql.key，不会有特别的权限检查，因为该目录已经被假定为安全。证书和密钥文件的位置可以使用连接参数sslcert和sslkey或者环境变量PGSSLCERT和PGSSLKEY覆盖。

在Unix系统上，私钥文件的权限必须禁止世界或组访问；可以通过以下命令实现：chmod 0600 ~/.postgresql/postgresql.key。或者，该文件可以归属于root，具有组读取权限（即，0640权限）。该设置适用于由操作系统管理证书和密钥文件的安装。然后，libpq的用户应该成为能够访问这些证书和密钥文件的组的成员。（在Microsoft Windows上，没有文件权限检查，因为%APPDATA%\postgresql目录被认为是安全的。）

postgresql.crt中的第一个证书必须是客户端的证书，因为它必须匹配客户端的私钥。可以选择将“中间”证书追加到该文件——这样做避免了在服务器上存放中间证书的要求（ssl\_ca\_file）。

证书和密钥可能是PEM或ASN.1 DER格式。

密钥可以以明文存储，也可以使用OpenSSL支持的任何算法（例如AES-128）使用密码进行加密。如果密钥是加密存储的，那么可以在sslpassword连接选项中提供密码。如果提供了加密密钥，而且sslpassword选项不存在或为空，如果TTY可用，那么OpenSSL将交互式地以Enter PEM pass phrase:提示输入密码。应用程序可以越过客户端证书提示和sslpassword参数的处理，通过提供它们自己的密钥密码回调；参见PQsetSSLKeyPassHook\_OpenSSL。

创建证书的指令请参考Section 16.9.5

### 32.18.3. 不同模式中提供的保护

sslmode参数的不同值提供了不同级别的保护。SSL 能够针对三类攻击提供保护：

#### 窃听

如果一个第三方能够检查客户端和服务端之间的网络流量，它能读取连接信息（包括用户名和口令）以及被传递的数据。SSL使用加密来阻止这种攻击。

#### 中间人（MITM）

如果一个第三方能对客户端和服务端之间传送的数据进行修改，它就能假装是服务器并且因此能看见并且修改数据，即使这些数据已被加密。然后第三方可以将连接信息和数据转送给原来的服务器，使得它不可能检测到攻击。这样做的通常途径包括 DNS 污染和地址劫持，借此客户端被重定向到一个不同的服务器。还有几种其他的攻击方式能够完成这种攻击。SSL使用证书验证让客户端认证服务器，就可以阻止这种攻击。

#### 模仿

如果一个第三方能假装是一个授权的客户端，它能够简单地访问它本不能访问的数据。通常这可以由不安全的口令管理所致。SSL使用客户端证书来确保只有持有合法证书的客户端才能访问服务器，这样就能阻止这种攻击。

对于一个已知的SSL-secured连接，在连接被建立之前，SSL 使用必须被配置在客户端和服务端之上。如果只在服务端上配置，客户端在知道服务端要求高安全性之前可能会结束发送敏感信息（例如口令）。在 libpq 中，要确保连接安全，可以设置sslmode参数为verify-full或verify-ca并且为系统提供一个根证书用来验证。这类似于使用一个https URL进行加密网页浏览。

一旦服务端已经被认证，客户端可以传递敏感数据。这意味着直到这一点，客户端都不需要知道是否证书将被用于认证，这样只需要在服务端配置中指定就比较安全。

所有SSL选项都带来了加密和密钥交换的负荷，因此必须在性能和安全性之间做出平衡。Table 32.1. 不同sslmode值所保护的冒险，以及它们是怎样看待安全性和负荷的。

Table 32.1. SSL 模式描述

sslmode	窃听保护	MITM保护	声明
disable	No	No	我不关心安全性，并且我不想为加密增加负荷。
allow	可能	No	我不关心安全性，但如果服务器坚持，我将承担加密带来的负荷。
prefer	可能	No	我不关心安全性，但如果服务器支持，我希望承担加密带来的负荷。
require	Yes	No	我想要对数据加密，并且我接受因此带来的负荷。我信任该网络会保证我总是连接到想要连接的服务器。
verify-ca	Yes	Depends on CA policy	我想要对数据加密，并且我接受因此带来的负荷。我想要确保我连接到的是我信任的服务器。
verify-full	Yes	Yes	我想要对数据加密，并且我接受因此带来的负荷。我想要确保我连接到的是我信任的服务器，并且就是我指定的那一个。

verify-ca和verify-full之间的区别取决于根CA的策略。如果使用了一个公共CA，verify-ca允许连接到那些可能已经被其他人注册到该CA的服务器。在这种情况下，总是应该使

用verify-full。如果使用了一个本地CA或者甚至是一个自签名的证书，使用verify-ca常常就可以提供足够的保护。

sslmode的默认值是prefer。如表中所示，这在安全性的角度来说没有意义，并且它只承诺可能的性能负荷。提供它作为默认值只是为了向后兼容，并且我们不推荐在安全部署中使用它。

## 32.18.4. SSL 客户端文件使用

Table 32. 总结了与客户端 SSL 设置相关的文件。

Table 32.2. Libpq/客户端 SSL 文件用法

文件	内容	效果
~/.postgresql/postgresql.crt	客户端证书	发送到服务器
~/.postgresql/postgresql.key	客户端私钥	证明客户端证书是由拥有者发送；不代表证书拥有者可信
~/.postgresql/root.crt	可信的证书机构	检查服务器证书是由一个可信的证书机构签发
~/.postgresql/root.crl	被证书机构撤销的证书	服务器证书不能在这个列表上

## 32.18.5. SSL 库初始化

如果你的应用初始化libssl或libcrypto库以及带有SSL支持的libpq，你应该调用PQinitOpenSSL来告诉libpq：libssl或libcrypto库已经被你的应用初始化，这样libpq将不会也去初始化那些库。

PQinitOpenSSL

允许应用选择要初始化哪个安全性库。

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

当do\_ssl是非零时，libpq将在第一次打开数据库连接前初始化OpenSSL库。当do\_crypto是非零时，libcrypto库将被初始化。默认情况下（如果没有调用PQinitOpenSSL），两个库都会被初始化。当SSL支持没有被编译时，这个函数也存在但是什么也不做。

如果你的应用使用并且初始化OpenSSL或者它的底层libcrypto库，你必须在第一次打开数据库连接前以合适的非零参数调用这个函数。同时要确保在打开一个数据库连接前已经完成了初始化。

PQinitSSL

允许应用选择要初始化哪个安全性库。

```
void PQinitSSL(int do_ssl);
```

这个函数等效于PQinitOpenSSL(do\_ssl, do\_ssl)。这对于要么初始化OpenSSL以及libcrypto要么都不初始化的应用足够用了。

PQinitSSL从LightDB 8.0 就存在了，而PQinitOpenSSL直到LightDB 8.4 才被加入，因此PQinitSSL可能对那些需要与旧版本libpq一起工作的应用来说更合适。



## 32.19. 在线程化程序中的行为

libpq默认是可再入的并且是线程安全的。你可能需要使用特殊的编译器命令行选项来编译你的应用代码。参考你的系统文档来了解如何编译启用线程的应用，或者在src/Makefile.global中查找PTHREAD\_CFLAGS和PTHREAD\_LIBS。这个函数允许查询libpq的线程安全状态：

PQisthreadsafe

返回libpq库的线程安全状态。

```
int PQisthreadsafe();
```

如果libpq是线程安全的则返回 1，否则返回 0。

一个线程限制是不允许两个线程同时尝试操纵同一个PGconn对象。特别是你不能从不同的线程通过同一个连接对象发出并发的命令（如果你需要运行并发命令，请使用多个连接）。

PGresult对象在创建后通常是只读的，并且因此可以在线程之间自由地被传递。但是，如果你使用任何Section 32.1或Section 32.1中描述的PGresult修改函数，你需要负责避免在同一个PGresult上的并发操作。

被废弃的函数PQrequestCancel以及PQoidStatus不是线程安全的并且不应当在多线程程序中使用。PQrequestCancel可以被替换为PQcancel。PQoidStatus可以被替换为PQoidValue。

如果你在应用中使用 Kerberos（除了在libpq中之外），你将需要对 Kerberos 调用加锁，因为 Kerberos 函数不是线程安全的。参考libpq源代码中的PQregisterThreadLock函数，那里有在libpq和应用之间做合作锁定的方法。

## 32.20. 编译 libpq 程序

要编译（即编译并且链接）一个使用libpq的程序，你需要做下列所有的事情：

- 包括libpq-fe.h头文件：

```
#include <libpq-fe.h>
```

如果你无法这样做，那么你通常会从你的编译器得到像这样的错误消息：

```
foo.c: In function `main':
foo.c:34: `PGconn' undeclared (first use in this function)
foo.c:35: `PGresult' undeclared (first use in this function)
foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
foo.c:68: `GRES_COMMAND_OK' undeclared (first use in this function)
foo.c:95: `GRES_TUPLES_OK' undeclared (first use in this function)
```

- 通过为你的编译器提供-Idirectory选项，向你的编译器指出LightDB头文件安装在哪里（在某些情况下编译器默认将查看该目录，因此你可以忽略这个选项）。例如你的编译命令行可能看起来像：

```
cc -c -I/usr/local/lightdb-x/include testprog.c
```

如果你在使用 makefile，那么把该选项加到CPPFLAGS变量中：

```
CPPFLAGS += -I/usr/local/lightdb-x/include
```

如果你的程序可能由其他用户编译，那么你不应该像那样硬编码目录位置。你可以运行工具 `lt_config` 在本地系统上找出头文件在哪里：

```
$ lt_config --includedir
/usr/local/include
```

如果你安装了 `pkg-config`，你可以运行：

```
$ pkg-config --cflags libpq
-I/usr/local/include
```

注意这将在路径前面包括 `-I`。

无法为编译器指定正确的选项将导致一个错误消息，例如：

```
testlibpq.c:8:22: libpq-fe.h: No such file or directory
```

- 当链接最终的程序时，指定选项 `-lpq`，这样 `libpq` 库会被编译进去，也可以用选项 `-Ldirectory` 向编译器指出 `libpq` 库所在的位置（再次，编译器将默认搜索某些目录）。为了最大的可移植性，将 `-L` 选项放在 `-lpq` 选项前面。例如：

```
cc -o testprog testprog1.o testprog2.o -L/usr/local/lightdb-x/lib -lpq
```

你也可以使用 `lt_config` 找出库目录：

```
$ lt_config --libdir
/usr/local/lightdb-x/lib
```

或者再次使用 `pkg-config`：

```
$ pkg-config --libs libpq
-L/usr/local/lightdb-x/lib -lpq
```

再次提示这会打印出全部的选项，而不仅仅是路径。

指出这一部分问题的错误消息可能看起来像：

```
testlibpq.o: In function `main':
testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
testlibpq.o(.text+0x71): undefined reference to `PQstatus'
testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
```

This means you forgot `-lpq`.

```
/usr/bin/ld: cannot find -lpq
```

这意味着你忘记了 `-L` 选项或者没有指定正确的目录。

## 32.21. 例子程序

这些例子和其他例子可以在源代码发布的 `src/test/examples` 目录中找到。

## Example 32.1. libpq 例子程序 1

```
/*
 * src/test/examples/testlibpq.c
 *
 *
 * testlibpq.c
 *
 * 测试 libpq (LightDB 前端库) 的 C 版本。
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int        nFields;
    int        i,
              j;

    /*
     * 如果用户在命令行上提供了一个参数，将它用作连接信息串。
     * 否则默认用设置 dbname=postgres 并且为所有其他链接参数使用环境变量或默认
     值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* 建立到数据库的一个连接 */
    conn = PQconnectdb(conninfo);

    /* 检查后端连接是否成功建立 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置总是安全的搜索路径，这样恶意用户就无法取得控制。 */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
```

```

{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * 任何时候不再需要 PGresult 时, 应该 PQclear 它来避免内存泄露
 */
PQclear(res);

/*
 * 我们的测试案例这里涉及使用一个游标, 对它我们必须用在一个事务块内。
 * 我们可以在一个单一的 "select * from pg_database" 的 PQexec() 中做整个事
情,
 * 但是作为一个好的例子它太琐碎。
 */

/* 开始一个事务块 */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

/*
 * 从 pg_database 取得行, 它是数据库的系统目录
 */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/* 首先, 打印出属性名 */
nFields = PQnfields(res);
for (i = 0; i < nFields; i++)
    printf("%-15s", PQfname(res, i));
printf("\n\n");

/* 接下来, 打印出行 */
for (i = 0; i < PQntuples(res); i++)
{
    for (j = 0; j < nFields; j++)

```

```

        printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* 关闭入口，我们不需要考虑检查错误 */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* 结束事务 */
    res = PQexec(conn, "END");
    PQclear(res);

    /* 关闭到数据库的连接并且清理 */
    PQfinish(conn);

    return 0;
}

```

### Example 32.2. libpq例子程序 2

```

/*
 * src/test/examples/testlibpq2.c
 *
 *
 * testlibpq2.c
 * 测试异步通知接口
 *
 * 开始这个程序，然后在另一个窗口的 ltsql 中做
 * NOTIFY TBL2;
 * 重复四次来让这个程序退出。
 *
 * 或者，如果你想要得到奇妙的事情，尝试：
 * 用下列命令填充一个数据库
 * （在 src/test/examples/testlibpq2.sql 中提供）
 *
 * CREATE SCHEMA TESTLIBPQ2;
 * SET search_path = TESTLIBPQ2;
 * CREATE TABLE TBL1 (i int4);
 * CREATE TABLE TBL2 (i int4);
 * CREATE RULE r1 AS ON INSERT TO TBL1 DO
 * (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * 开始这个程序，然后从psql做下面的操作四次：
 *
 * INSERT INTO TESTLIBPQ2.TBL1 VALUES (10);
 */

#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#ifdef HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int        nnotifies;

    /*
     * 用过用户在命令行上提供了一个参数，将它用作连接信息串。
     * 否则默认用设置 dbname=postgres 并且为所有其他链接参数使用环境变量或默认
     值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* 建立一个到数据库的连接 */
    conn = PQconnectdb(conninfo);

    /* 检查后端连接是否成功建立 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置总是安全的搜索路径，这样恶意用户就无法取得控制。 */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * 任何时候不再需要 PGresult 时，应该 PQclear 它来避免内存泄露
     */
}
```

```
    */
    PQclear(res);

    /*
     * 发出 LISTEN 命令启用来自规则的 NOTIFY 的通知。
     */
    res = PQexec(conn, "LISTEN TBL2");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    /* 在接收到四个通知后退出。 */
    nnotifies = 0;
    while (nnotifies < 4)
    {
        /*
         * 休眠到在连接上发生某些事情。我们使用 select(2) 来等待输入，但是你也可以使用 poll() 或相似的设施。
         */
        int      sock;
        fd_set   input_mask;

        sock = PQsocket(conn);

        if (sock < 0)
            break;          /* 不应该发生 */

        FD_ZERO(&input_mask);
        FD_SET(sock, &input_mask);

        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
            fprintf(stderr, "select() failed: %s\n", strerror(errno));
            exit_nicely(conn);
        }

        /* 现在检查输入 */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                    "ASYNC NOTIFY of '%s' received from backend PID %d\n",
                    notify->relname, notify->be_pid);
            PQfreemem(notify);
            nnotifies++;
            PQconsumeInput(conn);
        }
    }

    fprintf(stderr, "Done. \n");

    /* 关闭到数据库的连接并且清理 */
    PQfinish(conn);
```

```

    return 0;
}

```

### Example 32.3. libpq例子程序 3

```

/*
 * src/test/examples/testlibpq3.c
 *
 *
 * testlibpq3.c
 *     测试线外参数和二进制 I/O。
 *
 * 在运行之前，使用下列命令填充一个数据库（在 src/test/examples/testlibpq3.sql
 * 中提供）
 *
 * CREATE SCHEMA testlibpq3;
 * SET search_path = testlibpq3;
 * SET standard_conforming_strings = ON;
 * CREATE TABLE test1 (i int4, t text, b bytea);
 * INSERT INTO test1 values (1, 'joe''s place', '\000\001\002\003\004');
 * INSERT INTO test1 values (2, 'ho there', '\004\003\002\001\000');
 *
 * 期待的输出是:
 *
 * tuple 0: got
 * i = (4 bytes) 1
 * t = (11 bytes) 'joe''s place'
 * b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 * i = (4 bytes) 2
 * t = (8 bytes) 'ho there'
 * b = (5 bytes) \004\003\002\001\000
 */

#ifdef WIN32
#include <windows.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
}

```



```
    exit(1);
}

/*
 * 这个函数打印一个查询结果，该结果以二进制格式从上面的注释中定义的表中取得。
 * 我们把它分离出来是因为 main() 函数需要使用它两次。
 */
static void
show_binary_results(PGresult *res)
{
    int        i,
              j;
    int        i_fnum,
              t_fnum,
              b_fnum;

    /* 使用 PQfnumber 来避免假定结果中域的顺序 */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char    *iptr;
        char    *tptr;
        char    *bptr;
        int     blen;
        int     ival;

        /* 得到域值（我们忽略它们为空值的可能性！） */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         * INT4 的二进制表示是按照网络字节序的，我们最好强制为本地字节序。
         */
        ival = ntohl(*(uint32_t *) iptr);

        /*
         * TEXT 的二进制表示是文本，并且因为 libpq 会为其追加一个零字节，它将工
         * 作得和 C 字符串一样好。
         */
        /*
         * BYTEA 的二进制表示是一堆字节，其中可能包含嵌入的空值，因此我们必须注
         * 意域长度。
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
              PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
              PQgetlength(res, i, t_fnum), tptr);
        printf(" b = (%d bytes) ", blen);
        for (j = 0; j < blen; j++)
            printf("\\%03o", bptr[j]);
        printf("\n\n");
    }
}
```

```

}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int        paramLengths[1];
    int        paramFormats[1];
    uint32_t   binaryIntVal;

    /*
     * 如果用户在命令行上提供了一个参数，将它用作连接信息串。
     * 否则默认用设置 dbname=postgres 并且为所有其他链接参数使用环境变量或默认
     值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* 建立一个到数据库的连接 */
    conn = PQconnectdb(conninfo);

    /* 检查后端连接是否成功被建立 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置总是安全的搜索路径，这样恶意用户就无法取得控制。 */
    res = PQexec(conn, "SET search_path = testlibpq3");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    /*
     * 这个程序的要点在于用线外参数展示 PQexecParams() 的使用，以及数据的二进制
     传输。
     *
     * 第一个例子将参数作为文本传输，但是以二进制格式接收结果。
     * 通过使用线外参数，我们能够避免使用繁杂的引用和转义，即便数据是文本。
     * 注意我们怎么才能对参数值中的引号不做任何事情。
     */

    /* 这里是我们的线外参数值 */
    paramValues[0] = "joe's place";

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE t = $1",

```

```
        1,          /* 一个参数 */
        NULL,       /* 让后端推导参数类型 */
        paramValues,
        NULL,       /* 因为文本不需要参数长度 */
        NULL,       /* 对所有文本参数的默认值 */
        1);        /* 要求二进制结果 */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);

PQclear(res);

/*
 * 在第二个例子中，我们以二进制形式传输一个整数参数，并且再次以二进制形式接收结果。
 *
 * 尽管我们告诉 PQexecParams 我们让后端推导参数类型，我们实际上通过在查询文本中造型参数符号来强制该决定。
 * 在发送二进制参数时，这是一种好的安全测度。
 */

/* 将整数值 "2" 转换为网络字节序 */
binaryIntVal = htonl((uint32_t) 2);

/* 为 PQexecParams 设置参数数组 */
paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal);
paramFormats[0] = 1;          /* binary */

res = PQexecParams(conn,
    "SELECT * FROM test1 WHERE i = $1::int4",
    1,          /* 一个参数 */
    NULL,       /* 让后端推导参数类型 */
    paramValues,
    paramLengths,
    paramFormats,
    1);        /* 要求二进制结果 */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);

PQclear(res);

/* 关闭到数据库的连接并清理 */
PQfinish(conn);
```

```
    return 0;  
}
```

---

## Chapter 33. 大对象

LightDB具有大对象的功能，它提供了对于存储在一个特殊大对象结构中的用户数据的流式访问。对于那些大得无法以一个整体处理的数据值，流式访问非常有用。

本章介绍了LightDB中大对象数据的实现、编程和查询语言接口。在本章中我们使用名为libpq的C库来作为例子，但是绝大部分LightDB支持的本地编程接口也具有相同的功能。其他接口可能会在内部使用大对象接口来提供对大值的一般支持，但这里就不再描述。

### 33. 1. 简介

所有的大对象都存在一个名为pg\_largeobject的系统表中。每一个大对象还在系统表pg\_largeobject\_metadata中有一个对应的项。大对象可以通过类似于标准文件操作的读/写API来进行创建、修改和删除。

LightDB也支持一种称为“TOAST”的存储系统，它自动把大于一个数据库页的值存储到一个二级存储区域，每一个表都有专属的二级存储区域。这使得大对象功能显得有些陈旧。但是大对象功能仍然有一个优势是能够支持高达4TB的值，而TOAST域只能支持最大1GB。此外，读写一个大对象的片段会很高效率，但是大部分在TOAST域上的操作都将把整个值作为一个单元进行读或写。

### 33. 2. 实现特性

大对象的实现将大对象分解成很多“数据块”并且将这些数据块存储在数据库的行中。一个B-tree索引用来保证在进行随机访问读写时能够根据数据块号快速地搜索到正确的数据块。

为一个大对象存储的数据块并不需要是连续的。例如，如果一个应用打开了一个新的大对象，移动到偏移量1000000并写了一些字节，这并不会导致分配1000000字节的存储，只有覆盖写入字节范围的数据块需要被分配。而一个读操作将会把现有最后的数据块之前还未分配的位置读出为0。这和Unix文件系统中“稀疏”文件的一般行为相对应。

自LightDB 9.0起，大对象可以有一个拥有者和一组访问权限，它们可以用GRANT和REVOKE管理。读一个大对象需要SELECT权限，而写或者截断一个大对象则需要UPDATE权限。只有大对象的拥有者（或者一个数据库超级用户）可以创建大对象、注释大对象或修改大对象的拥有者。要调整这些行为以兼容以前的发行，请见lo\_compat\_privileges的运行时参数。

### 33. 3. 客户端接口

本节描述LightDB的libpq客户端接口为访问大对象所提供的功能。LightDB的大对象接口按照Unix文件系统的接口建模，也有相似的open、read、write、lseek等。

所有使用这些函数对大对象的操作都必须发生在一个SQL事务块中，因为大对象文件描述符只在事务期间有效。

在执行任何一个这种函数期间如果发生一个错误，该函数将会返回一个其他的不可能值，典型的是0或-1。一个关于该错误的消息亦会被保存在连接对象中，可以通过PQerrorMessage检索到。

使用这些函数的客户端应用应该包括头文件libpq/libpq-fs.h并链接libpq库。

#### 33. 3. 1. 创建一个对象

函数

```
Oid lo_creat(PGconn *conn, int mode);
```

创建一个新的大对象。其返回值是分配给这个新大对象的OID或者InvalidOid (0) 表示失败。mode自LightDB 8.1就不再使用且会被忽略。但是，为了和以前的发行兼容，它最好被设置为INV\_READ、INV\_WRITE或INV\_READ | INV\_WRITE（这些符号常量定义在头文件libpq/libpq-fs.h中）。

一个例子：

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

函数

```
Oid lo_create(PGconn *conn, Oid loobjId);
```

也创建一个新的大对象。分配给该大对象的OID可以通过loobjId指定，如果这样做，该OID已经被某个大对象使用时会产生错误。如果loobjId是InvalidOid (0)，则lo\_create会分配一个未使用的OID（这时和lo\_creat的行为相同）。返回值是分配给新大对象的OID或InvalidOid (0) 表示发生错误。

lo\_create在从LightDB 8.1开始的版本中是新的，如果该函数在旧服务器版本上运行，它将失败并返回InvalidOid。

一个例子：

```
inv_oid = lo_create(conn, desired_oid);
```

### 33.3.2. 导入一个大对象

要将一个操作系统文件导入成一个大对象，调用：

```
Oid lo_import(PGconn *conn, const char *filename);
```

filename指定了要导入为大对象的操作系统文件名。返回值是分配给新大对象的OID或InvalidOid (0) 表示发生错误。注意该文件是被客户端接口库而不是服务器所读取，因此它必须存在于客户端文件系统中并且对于客户端应用是可读的。

函数

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid loobjId);
```

也可以导入一个新大对象。分配给新大对象的OID可以用loobjId指定，如果这样做，该OID已经被某个大对象使用时会产生错误。如果loobjId是InvalidOid (0)，则lo\_import\_with\_oid会分配一个未使用的OID（这和lo\_import的行为相同）。返回值是分配给新大对象的OID或InvalidOid (0) 表示发生错误。

lo\_import\_with\_oid在从LightDB 8.1开始的版本中是新的并且在内部使用了lo\_create（在8.1中也是新的），如果该函数在旧服务器版本上运行，它将失败并返回InvalidOid。

### 33.3.3. 导出一个大对象

要把一个大对象导出到一个操作系统文件，调用：

```
int lo_export(PGconn *conn, Oid loobjId, const char *filename);
```

loobjId参数指定要导出的大对象的OID，filename参数指定操作系统文件名。注意该文件是被客户端接口库而不是服务器写入。成功返回1，错误返回-1。

### 33.3.4. 打开一个现有的大对象

要打开一个现有的大对象进行读写，调用：

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

lobjId参数指定要打开的大对象的OID。mode位控制着打开对象是为了只读（INV\_READ）、只写（INV\_WRITE）或者读写（这些符号常量定义在头文件libpq/libpq-fs.h中）。lo\_open返回一个（非负）大对象描述符以便后面用于lo\_read、lo\_write、lo\_lseek、lo\_lseek64、lo\_tell、lo\_tell64、lo\_truncate、lo\_truncate64以及lo\_close。该描述符只在当前事务期间有效。如果打开错误将会返回-1。

服务器目前并不区分模式INV\_WRITE和INV\_READ | INV\_WRITE：在两种情况中都允许从描述符读取。但是在这些模式和单独的INV\_READ之间有明显的区别：使用INV\_READ我们不能向描述符写入，从中读取的数据则反映了该大对象在活动事务快照时刻的内容（该快照在lo\_open被执行时创建），而不管之后被该事务或其他事务写入的内容。从一个以INV\_WRITE模式打开的描述符读取的数据所有其他已提交事务以及当前事务所作的写入。这与普通SQL命令 SELECT的REPEATABLE READ和READ COMMITTED事务模式之间的区别相似。

如果大对象的SELECT特权不可用，或者如果在指定了INV\_WRITE时UPDATE特权不可用，则lo\_open将会失败（在LightDB 11之前，这些特权的检查是在使用该描述符的第一次实际读取或写入时进行）。这些特权检查可以用lo\_compat\_privileges运行时参数禁用。

一个例子：

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

### 33.3.5. 向一个大对象写入数据

函数

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

从buf（大小必须是len）中写出len字节到大对象描述符fd。参数fd必须是已经由前面的lo\_open返回的大对象描述符。函数将返回实际写入的字节数（在当前的实现中，除非出错，返回的字节数总是等于len）。在出错时，返回值为-1。

尽管参数len被声明为类型size\_t，该函数会拒绝超过INT\_MAX的长度值。在实际中，被传送的数据最好是每块最多数兆字节。

### 33.3.6. 从一个大对象读取数据

函数

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

从大对象描述符fd中读取最多len字节到buf（大小必须是len）中。参数fd必须是已经由前面的lo\_open返回的大对象描述符。实际读出的字节数将被返回，如果先到达了大对象的末尾返回值可能会小于len。出错时返回值为-1。

尽管参数len被声明为类型size\_t，该函数会拒绝超过INT\_MAX的长度值。在实际中，被传送的数据最好是每块最多数兆字节。

### 33.3.7. 在一个大对象中查找

要改变一个大对象描述符的当前读或写位置，调用：

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

该函数将大对象文件描述符fd的当前位置指针移动到由offset指定的新位置。whence的可用值是SEEK\_SET（从对象开头定位）、SEEK\_CUR（从当前位置定位）以及SEEK\_END（从对象末尾定位）。返回值是新位置的指针，或者是-1表示出错。

在处理可能超过2GB大小的大对象时，换用

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int whence);
```

该函数的行为和lo\_lseek相同，但是它能接受一个超过2GB的offset并/或传送一个超过2GB的结果。注意如果新位置的指针超过2GB，lo\_lseek会失败。

lo\_lseek64是从LightDB 9.3开始增加的新函数。如果该函数在一个旧服务器版本上执行，将会失败并返回-1。

### 33.3.8. 获取一个大对象的查找位置

要得到一个对象描述符的当前读或写位置，调用：

```
int lo_tell(PGconn *conn, int fd);
```

如果出现错误，返回值是-1。

在处理可能超过2GB大小的大对象时，换用：

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

该函数和lo\_tell的行为相同，但是它能传递超过2GB的结果。注意如果当前读/写位置超过2GB，lo\_tell将会失败。

lo\_tell64是从LightDB 9.3开始新增的函数。如果该函数在旧服务器版本上运行，将会失败并返回-1。

### 33.3.9. 截断一个大对象

要将一个大对象截断成一个给定长度，调用：

```
int lo_truncate(PGconn *conn, int fd, size_t len);
```

该函数将大对象描述符fd截断为长度len。参数fd必须是已经由前面的lo\_open返回的大对象描述符。如果len超过了大对象的当前长度，大对象将会被使用空字节（'\0'）扩展到指定长度。成功时lo\_truncate返回0，失败时返回值为-1。

描述fd的读/写位置不变。

尽管参数len被声明为类型size\_t，lo\_truncate会拒绝超过INT\_MAX的长度值。

在处理可能超过2GB大小的大对象时，换用：

```
int lo_truncate64(PGconn *conn, int fd, pg_int64 len);
```

该函数和lo\_truncate的行为相同，但它能够接受超过2GB的len值。

lo\_truncate64是从LightDB 8.3开始新的函数，如果该函数运行在一个旧服务器版本上，它将失败并返回-1。



lo\_truncate64是从LightDB 9.3开始新的函数，如果该函数运行在一个旧服务器版本上，它将失败并返回-1。

### 33.3.10. 关闭一个大对象描述符

要关闭一个大对象描述符，调用：

```
int lo_close(PGconn *conn, int fd);
```

其中fd是由lo\_open返回的大对象描述符。成功时，lo\_close返回0，失败时返回-1。

在事务末尾仍然保持打开的任何大对象描述符都会自动被关闭。

### 33.3.11. 移除一个大对象

要从数据库中移除一个大对象，调用：

```
int lo_unlink(PGconn *conn, Oid loObjId);
```

loObjId参数指定要移除的大对象的OID。成功时返回1，失败时返回-1。

## 33.4. 服务器端函数

Table 33. 中列出了为从 SQL 操纵大对象定制的服务器端函数。

Table 33.1. 面向 SQL 的大对象函数

函数	描述 例子
lo_from_bytea ( loid oid, data bytea ) → oid	创建一个大对象并在其中存储数据。如果loid为0，则系统将选择一个空闲OID，否则该OID已被使用(如果某个大对象已经拥有该OID，则会出现错误)。在成功时，将返回大对象的OID。 lo_from_bytea(0, '\xfffff00') → 24528
lo_put ( loid oid, offset bigint, data bytea ) → void	在大对象中写入从给定偏移量开始的数据;如有必要，可以把大对象扩大。 lo_put(24528, 1, '\xaa') →
lo_get ( loid oid [, offset bigint, length integer ] ) → bytea	提取大对象的内容，或在其中的子字符串。 lo_get(24528, 0, 3) → \xffaaff

之前描述过的每个客户端函数都有一个相应的服务器端函数。实际上，多半客户端函数都是等效的服务器端函数的简单接口。这些可以从 SQL 命令方便调用的函数是：lo\_creat、lo\_create、lo\_unlink、lo\_import以及 lo\_export。下面是使用它们的例子：

```
CREATE TABLE image (
    name          text,
    raster        oid
);

SELECT lo_creat(-1);      -- 返回新的空大对象的OID

SELECT lo_create(43213); -- 尝试创建OID为43213的大对象

SELECT lo_unlink(173454); -- 删除OID为173454的大对象
```

```

INSERT INTO image (name, raster)
  VALUES ('beautiful image', lo_import('/etc/motd'));

INSERT INTO image (name, raster) -- 和上面相同, 但是指定了使用的OID
  VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
  WHERE name = 'beautiful image';

```

服务器端的`lo_import`和`lo_export`函数具有和它们的客户端同类大不相同的行为。这两个函数从服务器的文件系统中读和写文件, 使用的是数据库所有者的权限。因此, 默认情况下它们的使用被限制于超级用户。相反, 客户端的导入和导出函数读写的是客户端的文件系统, 使用的是客户端程序的权限。除了读取或写入所请求的大对象的特权之外, 客户端函数不要任何数据库特权。

### Caution

可以把服务器端的`lo_import`和`lo_export`函数GRANT给非超级用户, 但需要仔细地考虑安全因素。有这类特权的恶意用户可以很容易地利用它们成为超级用户(例如通过重写服务器配置文件), 或者攻击该服务器文件系统的其他部分而无需获得数据库超级用户特权。因此对具有这类特权的角色访问必须受到和超级用户角色一样的仔细保护。尽管如此, 如果某些例行任务需要使用服务器端的`lo_import`或者`lo_export`, 使用具有这类特权的角色比使用具有完整超级用户特权的角色更加安全, 因为那样会减小意外错误造成的损伤风险。

函数`lo_read`和`lo_write`的功能也可以在服务器端调用, 但是在服务器端的名称与客户端接口不同: 它们的名称中不包含下划线。我们必须以`loread`和`lowrite`调用这些函数。

## 33.5. 例子程序

Example 33. 是一个展示`libpq`中大对象接口如何使用的例子程序。部分程序被注释但仍保留在代码中, 用户可以利用之。该程序可以在源代码的`src/test/examples/testlo.c`中找到。

Example 33.1. 用`libpq`操作大对象的例子程序

```

/*-----
 *
 * testlo.c
 * 测试通过libpq实用大对象
 *
 * Portions Copyright (c) 1996-2021, Hundsun Technologies Inc.
 *
 *
 * IDENTIFICATION
 * src/test/examples/testlo.c
 *
 *-----
 */
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>

```

```
#include <fcntl.h>
#include <unistd.h>

#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile -
 *   把文件 "in_filename" 作为一个大对象 "lobjOid" 载入到数据库
 *
 */
static Oid
importFile(PGconn *conn, char *filename)
{
    Oid      lobjId;
    int      lobj_fd;
    char     buf[BUFSIZE];
    int      nbytes,
            tmp;
    int      fd;

    /*
     * 打开要读入的文件
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {
        /* error */
        fprintf(stderr, "cannot open unix file \"%s\"\n", filename);
    }

    /*
     * 创建大对象
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * 从该 Unix 文件读取并写入到大对象
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading \"%s\"", filename);
    }

    close(fd);
    lo_close(conn, lobj_fd);

    return lobjId;
}

static void
```

```
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int      lobj_fd;
    char     *buf;
    int      nbytes;
    int      nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = '\0';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
        if (nbytes <= 0)
            break;          /* no more data? */
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

static void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int      lobj_fd;
    char     *buf;
    int      nbytes;
    int      nwritten;
    int      i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = '\0';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
        if (nbytes <= 0)
        {
            fprintf(stderr, "\nWRITE FAILED!\n");
            break;
        }
    }
}
```

```
    }
}
free(buf);
fprintf(stderr, "\n");
lo_close(conn, lobj_fd);
}

/*
 * exportFile -
 *   把大对象 "lobj0id" 导出成文件 "out_filename"
 *
 */
static void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int      lobj_fd;
    char     buf[BUFSIZE];
    int      nbytes,
            tmp;
    int      fd;

    /*
     * 打开大对象
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    /*
     * 打开要写入的文件
     */
    fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    if (fd < 0)
    {
        /* 错误 */
        fprintf(stderr, "cannot open unix file \"%s\"",
                filename);
    }

    /*
     * 从大对象读入并写出到 Unix 文件
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing \"%s\"",
                    filename);
        }
    }

    lo_close(conn, lobj_fd);
    close(fd);

    return;
}
```

```
static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char      *in_filename,
              *out_filename;
    char      *database;
    Oid       lobjOid;
    PGconn    *conn;
    PGresult  *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * 设置连接
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* 检查看看后端连接是否成功建立 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* 设置总是安全的搜索路径，这样恶意用户就无法取得控制权。 */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "begin");
    PQclear(res);
    printf("importing file \"%s\" ... \n", in_filename);
    /* lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    if (lobjOid == 0)
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
}
```

---

```
else
{
    printf("\tas large object %u.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's\n");
    overwrite(conn, lobjOid, 1000, 1000);

    printf("exporting large object to file \"%s\" ... \n", out_filename);
/*
    exportFile(conn, lobjOid, out_filename); */
    if (lo_export(conn, lobjOid, out_filename) < 0)
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
}

res = PQexec(conn, "end");
PQclear(res);
PQfinish(conn);
return 0;
}
```

---

## Chapter 34. ECPG — C中的嵌入式 SQL

这一章描述了用于LightDB的嵌入式SQL包。它由 Linus Tolke (<linus@epact.se>) 和 Michael Meskes (<meskes@postgresql.org>) 编写。最初它是为了与C一起工作而编写的。它也能与C++配合，但是它还不识别所有的C++结构。

这份文档还远没有完成。但是因为这个结构是标准化的，额外的信息可以在有关 SQL 的很多资源中找到。

### 34.1. 概念

一个嵌入式 SQL 程序由一种普通编程语言编写的代码（在这里是 C）和位于特殊标记的小节中的 SQL 命令混合组成。要构建该程序，源代码 (\*.pgc) 首先会通过嵌入式 SQL 预处理器，它会将源代码转换成一个普通 C 程序 (\*.c)，并且后来它能够被一个 C 编译器所处理（编译和链接详见Section 34.10。转换过的 ECPG 应用会通过嵌入式 SQL 库 (ecpglib) 调用 libpq 库中的函数，并且与 LightDB 服务器使用普通的前端/后端协议通信。

嵌入式SQL在为 C 代码处理SQL命令方面比起其他方法来具有优势。首先，它会搞定向你的C程序变量传递或者读取信息时的繁文缛节。其次，程序中的 SQL 代码在编译时就会被检查以保证语法正确性。第三，C 中的嵌入式SQL是在SQL标准中指定的并且受到很多其他SQL数据库系统的支持。LightDB实现被设计为尽可能匹配这个标准，并且通常可以相对容易地把它为其他 SQL 数据库编写的SQL程序移植到LightDB。

正如已经指出的，为嵌入式SQL接口编写的程序是插入了用于执行数据库相关动作的特殊代码的普通的 C 程序。这种特殊代码总是具有这样的形式：

```
EXEC SQL ...;
```

这些语句在语法上取代了一个 C 语句。取决于特定的语句，它们可以出现在全局层面或者是一个函数中。嵌入式 SQL语句遵循普通SQL代码的大小写敏感性规则，而不是 C 的大小写敏感性规则。它们也允许嵌套的 C 风格注释（SQL 标准的一部分）。不过，程序的 C 部分遵循 C 的标准不接受嵌套注释。

LightDB 从 23.3版本开始支持在EXEC SQL中使用如下三种注释：'-- comment'， '// comment' 和 '/\* comment \*/'。在 23.3 版本之前只支持/\*\*/注释。

'-- comment' 和 '// comment' 注释在转换成 C 程序后会被删除，而 '/\* comment \*/' 注释会被移动到 ECPGdo 函数的前面，做为函数调用的注释。

下列小节解释了所有嵌入式 SQL 语句。

### 34.2. 管理数据库连接

这一节描述如何打开、关闭以及切换数据库连接。

#### 34.2.1. 连接到数据库服务器

我们可以使用下列语句连接到一个数据库：

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

target可以用下列方法指定：

- dbname[@hostname][:port]
- tcp:postgresql://hostname[:port][/?options]



- `unix:postgresql://hostname[:port][/dbname][?options]`
- 一个包含上述形式之一的 SQL 字符串
- 到一个包含上述形式之一（参见例子）的字符变量的引用
- DEFAULT

如果你用字面（也就是不通过一个变量引用）指定连接目标并且你没有引用该值，那么将会应用普通 SQL 的大小写不敏感性规则。在那种情况中，你也能够按照需要单独将个体参数放置在双引号中。实际上，使用一个（单引号引用）的字符串或一个变量引用出错的可能性更小。连接目标DEFAULT会以默认用户名发起一个到默认数据库的连接。在那种情况中不能指定单独的用户名或连接名。

也有不同的方法来指定用户名：

- `username`
- `username/password`
- `username IDENTIFIED BY password`
- `username USING password`

如上所述，参数username以及password可以是一个 SQL 标识符、一个 SQL 字符串或者一个对字符变量的引用。

如果连接目标包含任何options，这些由keyword=value组成的规范，由表示“和”的符号(&)分隔。允许的关键字与libpq识别的关键字相同(参见Section 32.1.2)。在任何keyword或value之前的空格将被忽略，但其中或之后则不会。请注意，无法将&写入value。

connection-name被用来在一个程序中处理多个连接。如果一个程序只使用一个连接，它可以被忽略。最近被打开的连接将成为当前连接，当一个 SQL 语句要被执行时，将默认使用它（见这一章稍后的部分）。

如果不可信用户能够访问一个没有采用安全方案使用模式的数据库，开始每个会话时应该从search\_path中移除公共可写的方案。例如，把options=-c search\_path=加到options中或者在连接后发出EXEC SQL SELECT pg\_catalog.set\_config('search\_path', '', false);。这种考虑并非专门针对ECPG，它适用于每一种用来执行任意SQL命令的接口。

这里有一些CONNECT语句的例子：

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;
```

```
EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john;
```

```
EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
...
EXEC SQL CONNECT TO :target USER :user USING :passwd;
/* 或者 EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

最后一种形式利用被上文成为字符变量引用的变体。你将在后面的小节中看到当你把 C 变量前放上一个冒号时，它们是怎样被用于 SQL 语句的。

注意连接目标的格式没有在 SQL 标准中说明。因此如果你想要开发可移植的应用，你可能想要使用某种基于上述最后一个例子的方法来把连接目标字符串封装在某个地方。

## 34.2.2. 选择一个连接

嵌入式 SQL 程序中的 SQL 语句默认是在当前连接（也就是最近打开的那一个）上执行的。如果一个应用需要管理多个连接，那么有两种方法来处理这种需求。

第一个选项是显式地为每一个 SQL 语句选择一个连接，例如：

```
EXEC SQL AT connection-name SELECT ...;
```

如果应用需要以混合的顺序使用多个连接，这个选项特别合适。

如果你的应用使用多个线程执行，它们不能并发地共享一个连接。你必须显式地控制对连接的访问（使用互斥量）或者为每个线程使用一个连接。

第二个选项是执行一个语句来切换当前的连接。该语句是：

```
EXEC SQL SET CONNECTION connection-name;
```

如果很多语句要被在同一个连接上执行，这个选项特别方便。

这里有一个管理多个数据库连接的例子程序：

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
    COMMIT;
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
    COMMIT;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
    COMMIT;

    /* 这个查询将在最近打开的数据库 "testdb3" 中执行 */
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb3)\n", dbname);

    /* 使用 "AT" 在 "testdb2" 中运行一个查询 */
    EXEC SQL AT con2 SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb2)\n", dbname);

    /* 切换当前连接到 "testdb1" */
    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb1)\n", dbname);
}
```

```
EXEC SQL DISCONNECT ALL;
return 0;
}
```

这个例子将产生这样的输出：

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

### 34.2.3. 关闭一个连接

要关闭一个连接，使用下列语句：

```
EXEC SQL DISCONNECT [connection];
```

connection可以用下列方法指定：

- connection-name
- DEFAULT
- CURRENT
- ALL

如果没有指定连接名，当前连接将被关闭。

在一个应用中总是显式地从它打开的每一个连接断开是一种好的风格。

## 34.3. 运行 SQL 命令

任何 SQL 命令都可以在一个嵌入式 SQL 应用中被运行。下面是一些在嵌入式 SQL 应用中运行 SQL 命令的例子。

### 34.3.1. 执行 SQL 语句

创建一个表：

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

插入行：

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

删除行：

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

更新：

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

返回一个单一结果行的SELECT语句也可以直接使用EXEC SQL执行。要处理有多行的结果集，一个应用必须使用一个游标，可参考下面的Section 34.3.2(作为一种特殊情况，一个应用可以一次取出多行到一个数组主变量中，参考Section 34.4.4.3)。

单行选择:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

还有，一个配置参数可以用SHOW命令检索:

```
EXEC SQL SHOW search_path INTO :var;
```

:something形式的记号是主变量，即它们指向 C 程序中的变量。它们在Section 34.4中解释。

### 34.3.2. 使用游标

要检索一个保持多行的结果集，一个应用必须声明一个游标并且从该游标中取得每一行。使用一个游标的步骤如下：声明一个游标、打开它、从该游标取得一行、重复并且最终关闭它。

使用游标选择:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

有关声明游标的更多细节，可参考DECLARE；FETCH命令的细节则可以参考FETCH。

#### Note

ECPG DECLARE命令实际上不会导致一个语句被发送到 LightDB 后端。在OPEN命令被执行时，游标会在后端被打开（使用后端的DECLARE命令）。

### 34.3.3. 管理事务

在默认模式中，只有当EXEC SQL COMMIT被发出时才会提交命令。嵌入式 SQL 接口也通过ecpg的-t命令行选项（见ecpg）或者通过EXEC SQL SET AUTOCOMMIT TO ON语句支持事务的自动提交（类似于ltsql的默认行为）。在自动提交模式中，除非位于一个显式事务块内，每一个命令都会被自动提交。这种模式可以使用EXEC SQL SET AUTOCOMMIT TO OFF显式地关闭。

可以使用下列事务管理命令:

```
EXEC SQL COMMIT
```

提交一个进行中的事务。

```
EXEC SQL ROLLBACK
```

回滚一个进行中的事务。

```
EXEC SQL PREPARE TRANSACTION transaction_id
```

为两阶段提交准备当前事务。

```
EXEC SQL COMMIT PREPARED transaction_id
```

提交一个处于准备好状态的事务。

```
EXEC SQL ROLLBACK PREPARED transaction_id
```

回滚一个处于准备好状态的事务。

```
EXEC SQL SET AUTOCOMMIT TO ON
```

启用自动提交模式。

```
EXEC SQL SET AUTOCOMMIT TO OFF
```

禁用自动提交模式。这是默认值。

### 34.3.4. 预备语句

当传递给 SQL 语句的值在编译时未知或者同一个语句要被使用多次时，那么预备语句就有用武之地了。

语句使用命令PREPARE进行预备。对于还未知的值，使用占位符“?”：

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?";
```

如果一个语句返回一个单一行，应用可以在PREPARE之后调用EXECUTE来执行该语句，同时要用一个USING子句为占位符提供真实的值：

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

如果一个语句返回多行，应用可以使用一个基于该预备语句声明的游标。要绑定输入参数，该游标必须用一个USING子句打开：

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;
```

```
/* 当到达结果集末尾时，跳出 while 循环 */
EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

```
EXEC SQL OPEN foo_bar USING 100;
```

```
...
```

```
while (1)
```

```
{
```

```
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
```

```
    ...
```

```
}
```

```
EXEC SQL CLOSE foo_bar;
```

当你不再需要该预备语句时，你应该释放它：

```
EXEC SQL DEALLOCATE PREPARE name;
```

更多有关PREPARE的细节，可参考PREPARE。关于使用占位符和输入参数的细节，可参考Section 34.5

## 34.4. 使用主变量

在Section 34.3中，你了解了如何从一个嵌入式 SQL 程序执行 SQL 语句。某些那种语句只使用固定值并且没有提供方法来插入用户提供的值到语句中或者让程序处理查询返回的值。那种语句在实际应用中其实没有什么用处。这一节详细解释了如何使用一种简单的机制（主变量）在 C 程序和嵌入式 SQL 语句之间传递数据。在一个嵌入式 SQL 程序中，我们认为 SQL 语句是 C 程序代码中的客人，而 C 代码是主语言。因此 C 程序的变量被称为主变量。

另一种在 LightDB 后端和 ECPG 应用之间交换值的方式是使用 SQL 描述符，它在Section 34.4中介绍。

### 34.4.1. 概述

在嵌入式 SQL 中进行 C 程序和 SQL 语句间的数据传递特别简单。我们不需要让程序把数据粘贴到语句（这会导致很多复杂性，例如正确地引用值），我们可以简单地在 SQL 语句中写 C 变量的名称，只要在它前面放上一个冒号。例如：

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

这个语句引用了两个 C 变量（名为v1和v2）并且还使用了一个常规的 SQL 字符串来说明你没有被限制于使用某一种数据。

这种在 SQL 语句中插入 C 变量的风格可以用在 SQL 语句中每一个应该出现值表达式的地方。

### 34.4.2. 声明小节

要从程序传递数据给数据库（例如作为一个查询的参数）或者从数据库传数据回程序，用于包含这些数据的 C 变量必须在特别标记的节中被声明，这样嵌入式 SQL 预处理器才会注意它们。

这个节开始于：

```
EXEC SQL BEGIN DECLARE SECTION;
```

并且结束于：

```
EXEC SQL END DECLARE SECTION;
```

在这两行之间，必须是正常的 C 变量声明，例如：

```
int    x = 4;
char  foo[16], bar[16];
```

如你所见，你可以选择为变量赋一个初始值。变量的可见范围由定义它的节在程序中的位置决定。你也可以使用下面的语法声明变量，这种语法将会隐式地创建一个声明节：

```
EXEC SQL int i = 4;
```

你可以按照你的意愿在一个程序中放上多个声明节。

这些声明也会作为 C 变量被重复在输出文件中，因此无需再次声明它们。不准备在 SQL 命令中使用的变量可以正常地在这些特殊节之外声明。

一个结构或联合的定义也必须被列在一个DECLARE节中。否则预处理器无法处理这些类型，因为它不知道它们的定义。

### 34.4.3. 检索查询结果

现在你应该能够把程序产生的数据传递到一个 SQL 命令中了。但是怎么检索一个查询的结果呢？为此，嵌入式 SQL 提供了常规命令SELECT和FETCH的特殊变体。这些命令有一个特殊的INTO子句，它指定被检索到的值要被存储在哪些主变量中。SELECT被用于只返回单一行的查询，而FETCH被用于使用一个游标返回多行的查询。

这里是一个例子：

```
/*
 * 假定有这个表：
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

那么INTO子句出现在选择列表和FROM子句之间。选择列表中的元素数量必须和INTO后面列表（也被称为目标列表）的元素数量相等。

这里有一个使用命令FETCH的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

...

do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

这里INTO子句出现在所有正常子句的后面。

## 34.4.4. 类型映射

当 ECPG 应用在 LightDB 服务器和 C 应用之间交换值时（例如从服务器检索查询结果时或者用输入参数执行 SQL 语句时），值需要在 LightDB 数据类型和主语言变量类型（具体来说是 C 语言数据类型）之间转换。ECPG 的要点之一就是它会在大多数情况下自动搞定这种转换。

在这方面有两类数据类型：一些简单 LightDB 数据类型（例如integer和text）可以被应用直接读取和写入。其他 LightDB 数据类型（例如timestamp和numeric）只能通过特殊库函数访问，见Section 34.4.4.2

Table 34.展示了哪种 LightDB 数据类型对应于哪一种 C 数据类型。当你希望发送或接收一种给定 LightDB 数据类型的值时，你应该在声明节中声明一个具有相应 C 数据类型的 C 变量。

Table 34.1. 在 LightDB 数据类型和 C 变量类型之间映射

LightDB 数据类型	主变量类型
smallint	short
integer	int
bigint	long long int
decimal	decimal <sup>a</sup>
numeric	numeric <sup>a</sup>
real	float
double precision	double
smallserial	short
serial	int
bigserial	long long int
oid	unsigned int
character(n), varchar(n), text	char[n+1], VARCHAR[n+1]
name	char[NAMEDATALEN]
timestamp	timestamp <sup>a</sup>
interval	interval <sup>a</sup>
date	date <sup>a</sup>
boolean	bool <sup>b</sup>
bytea	char *, bytea[n]

<sup>a</sup>这种类型只能通过特殊的库函数访问，见Section 34.4.4.2

<sup>b</sup>如果不是本地化类型，则声明在ecpglib.h中

### 34.4.4.1. 处理字符串

要处理 SQL 字符串数据类型（例如varchar以及text），有两种可能的方式来声明主变量。

一种方式是使用char[]（一个char字符串），这是在 C 中处理字符数据最常见的方式。

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

注意你必须自己照看长度。如果你把这个主变量用作一个查询的目标变量并且该查询返回超过 49 个字符的字符串，那么将会发生缓冲区溢出。



另一种方式是使用VARCHAR类型，它是 ECPG 提供的一种特殊类型。在一个VARCHAR类型数组上的定义会被转变成一个命名的struct。这样一个声明：

```
VARCHAR var[180];
```

会被转变成：

```
struct varchar_var { int len; char arr[180]; } var;
```

成员arr容纳包含一个终止零字节的字符串。因此，要在一个VARCHAR主变量中存储一个字符串，该主变量必须被声明为具有包括零字节终止符的长度。成员len保存存储在arr中的字符串的长度，不包括终止零字节。当一个主变量被用做一个查询的输入时，如果strlen(arr)和len不同，将使用短的那一个。

VARCHAR可以被写成大写或小写形式，但是不能大小写混合。

char和VARCHAR主变量也可以保存其他 SQL 类型的值，它们将被存储为字符串形式。

### 34.4.4.2. 访问特殊数据类型

ECPG 包含一些特殊类型帮助你容易地与来自 LightDB 服务器的一些特殊数据类型交互。特别地，它已经实现了对于numeric、decimal、date、timestamp以及interval类型的支持。这些数据类型无法有效地被映射到原始的主变量类型（例如int、long long int或者char[]），因为它们有一种复杂的内部结构。应用通过声明特殊类型的主变量以及使用pgtypes 库中的函数来处理这些类型。pgtypes 库（在Section 34.6中详细描述）包含了处理这些类型的基本函数，这样你不需要仅仅为了给一个时间戳增加一个时段而发送一个查询给 SQL 服务器。

下面的小节描述了这些特殊数据类型。关于 pgtypes 库函数的更多细节，请参考Section 34.6

#### 34.4.4.2.1. timestamp, date

这里有一种在 ECPG 主应用中处理timestamp变量的模式。

首先，程序必须包括用于timestamp类型的头文件：

```
#include <pgtypes_timestamp.h>
```

接着，在声明节中声明一个主变量为类型timestamp：

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

并且在读入一个值到该主变量中之后，使用 pgtypes 库函数处理它。在下面的例子中，timestamp值被PGTYPEStimestamp\_to\_asc()函数转变成文本（ASCII）形式：

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

这个例子将展示像下面形式的一些结果：

```
ts = 2010-06-27 18:03:56.949343
```

另外，DATE 类型可以用相同的方式处理。程序必须包括pgtypes\_date.h，声明一个主变量为日期类型并且将一个 DATE 值使用PGTYPESdate\_to\_asc()函数转变成一种文本形式。关于pgtypes 库函数的更多细节，请参考Section 34.6

#### 34.4.4.2.2. interval

对interval类型的处理也类似于timestamp和date类型。不过，必须显式为一个interval类型分配内存。换句话说，该变量的内存空间必须在堆内存中分配，而不是在栈内存中分配。

这里是一个例子程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    interval *in;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config(' search_path', '', false); EXEC SQL
COMMIT;

    in = PGTYPESinterval_new();
EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGTYPESinterval_to_asc(in));
    PGTYPESinterval_free(in);

    EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
    return 0;
}
```

#### 34.4.4.2.3. numeric, decimal

numeric和decimal类型的处理类似于interval类型：需要定义一个指针、在堆上分配一些内存空间并且使用 pgtypes 库函数访问该变量。关于 pgtypes 库函数的更多细节，请参考Section 34.6

pgtypes 库没有特别为decimal类型提供函数。一个应用必须使用一个 pgtypes 库函数把它转变成一个numeric变量以便进一步处理。

这里是一个处理numeric和decimal类型变量的例子程序。

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    numeric *num;
```

```

    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb;
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
COMMIT;

num = PGTYPEStnumeric_new();
dec = PGTYPEStdecimal_new();

EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;

printf("numeric = %s\n", PGTYPEStnumeric_to_asc(num, 0));
printf("numeric = %s\n", PGTYPEStnumeric_to_asc(num, 1));
printf("numeric = %s\n", PGTYPEStnumeric_to_asc(num, 2));

/* 将一个decimal转变成numeric以显示一个decimal值。 */
num2 = PGTYPEStnumeric_new();
PGTYPEStnumeric_from_decimal(dec, num2);

printf("decimal = %s\n", PGTYPEStnumeric_to_asc(num2, 0));
printf("decimal = %s\n", PGTYPEStnumeric_to_asc(num2, 1));
printf("decimal = %s\n", PGTYPEStnumeric_to_asc(num2, 2));

PGTYPEStnumeric_free(num2);
PGTYPEStdecimal_free(dec);
PGTYPEStnumeric_free(num);

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
return 0;
}

```

#### 34.4.4.2.4. bytea

bytea类型的处理与VARCHAR相似。类型bytea的数组上的定义被转换为每个变量的命名结构。声明类似于：

```
bytea var[180];
```

会被转变成：

```
struct bytea_var { int len; char arr[180]; } var;
```

成员 arr 承载二进制格式数据。不像VARCHAR，它还可以作为数据的一部分处理 '\0'。数据往/来转换为十六进制格式，并通过 ecpglib 发送/接收。

#### Note

bytea 变量只有在 bytea\_output 被设置为 hex时才能够使用。

#### 34.4.4.3. 非简单类型的主变量

你也可以把数组、typedefs、结构和指针用作主变量。

### 34.4.4.3.1. 数组

将数组用作主变量有两种情况。第一种如Section 34.4.4. 所述，是一种将一些文本字符串存储在char[]或VARCHAR[]中的方法。第二种是不用一个游标从一个查询结果中检索多行。如果没有一个数组，要处理由多个行组成的查询结果，我们需要使用一个游标以及FETCH命令。但是使用数组主变量，多个行可以被一次收取。该数组的长度必须被定义成足以容纳所有的行，否则很可能会发生一次缓冲区溢出。

下面的例子扫描pg\_database系统表并且显示所有可用数据库的 OID 和名称：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

    memset(dbname, 0, sizeof(char)* 16 * 8);
    memset(dbid, 0, sizeof(int) * 8);

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
COMMIT;

    /* 一次检索多行到数组中。 */
    EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM pg_database;

    for (i = 0; i < 8; i++)
        printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

这个例子显示下面的结果（确切的值取决于本地环境）。

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

### 34.4.4.3.2. 结构

一个成员名称匹配查询结果列名的结构可以被用来一次检索多列。该结构使得我们能够在一个单一主变量中处理多列值。

下面的例子从pg\_database系统表以及使用pg\_database\_size()函数检索可用数据库的OID、名称和尺寸。在这个例子中，一个成员名匹配SELECT结果的每一列的结构变量dbinfo\_t被用来检索结果行，而不需要把多个主变量放在FETCH语句中。

```

EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
        int oid;
        char datname[65];
        long long int size;
    } dbinfo_t;

    dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid)
AS size FROM pg_database;
    EXEC SQL OPEN cur1;

    /* 在达到结果集末尾时, 跳出 while 循环 */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* 将多列取到一个结构中。 */
        EXEC SQL FETCH FROM cur1 INTO :dbval;

        /* 打印该结构的成员。 */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname,
dbval.size);
    }

    EXEC SQL CLOSE cur1;

```

这个例子会显示下列结果（确切的值取决于本地环境）。

```

oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
oid=313780, datname=testdb, size=8183012

```

结构主变量将列尽数“吸收”成结构的域。额外的列可以被分配给其他主变量。例如，上面的程序也可以使用结构外部的size变量重新构造：

```

EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
        int oid;
        char datname[65];
    } dbinfo_t;

    dbinfo_t dbval;
    long long int size;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid)
AS size FROM pg_database;

```

```

EXEC SQL OPEN cur1;

/* 在达到结果集末尾时, 跳出 while 循环 */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将多列取到一个结构中。 */
    EXEC SQL FETCH FROM cur1 INTO :dbval, :size;

    /* 打印该结构的成员。 */
    printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname,
size);
}

EXEC SQL CLOSE cur1;

```

### 34.4.4.3.3. Typedefs

使用typedef关键词可以把新类型映射到已经存在的类型。

```

EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;

```

注意你也可以使用:

```
EXEC SQL TYPE serial_t IS long;
```

这种声明不需要位于一个声明节之中。

### 34.4.4.3.4. 指针

你可以声明最常见类型的指针。不过注意, 你不能使用指针作为不带自动分配内存的查询的目标变量。关于自动分配内存的详情请参考Section 34.7

```

EXEC SQL BEGIN DECLARE SECTION;
    int *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;

```

## 34.4.5. 处理非简单 SQL 数据类型

这一节包含关于如何处理 ECPG 应用中非标量以及用户定义的 SQL 级别数据类型。注意这和上一节中描述的简单类型主变量的处理有所不同。

### 34.4.5.1. 数组

ECPG 中不直接支持 SQL 级别的多维数组。一维 SQL 数组可以被映射到 C 数组主机变量, 反之亦然。不过, 在创建一个语句时, ecpg并不知道列的类型, 因此它无法检查一个 C 数组否是一个 SQL 数组的输入。在处理一个 SQL 语句的输出时, ecpg 有必需的信息并且进而检查是否两者都是 数组。

如果一个查询个别地访问一个数组的元素, 那么这可以避免使用 ECPG 中的数组。然后, 应该使用一个能被映射到该元素类型的类型的主变量。例如, 如果一个列类型是integer数

组，可以使用一个类型int的主变量。还有如果元素类型是varchar或text，可以使用一个类型char[]或VARCHAR[]的主变量。

这里是一个例子。假定有下面的表：

```
CREATE TABLE t3 (  
    ii integer[]  
);  
  
testdb=> SELECT * FROM t3;  
    ii  
-----  
    {1,2,3,4,5}  
(1 row)
```

下面的例子程序检索数组的第四个元素并且把它存储到一个类型为int的主变量中：

```
EXEC SQL BEGIN DECLARE SECTION;  
int ii;  
EXEC SQL END DECLARE SECTION;  
  
EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;  
EXEC SQL OPEN cur1;  
  
EXEC SQL WHENEVER NOT FOUND DO BREAK;  
  
while (1)  
{  
    EXEC SQL FETCH FROM cur1 INTO :ii ;  
    printf("ii=%d\n", ii);  
}  
  
EXEC SQL CLOSE cur1;
```

这个例子会显示下面的结果：

```
ii=4
```

要把多个数组元素映射到一个数组类型主变量中的多个元素，数组列的每一个元素以及主变量数组的每一个元素都必须被单独管理，例如：

```
EXEC SQL BEGIN DECLARE SECTION;  
int ii_a[8];  
EXEC SQL END DECLARE SECTION;  
  
EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;  
EXEC SQL OPEN cur1;  
  
EXEC SQL WHENEVER NOT FOUND DO BREAK;  
  
while (1)  
{  
    EXEC SQL FETCH FROM cur1 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];  
    ...  
}
```

注意

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 错误 */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
    ...
}
```

在这种情况下不会正确工作，因为你无法把一个数组类型列直接映射到一个数组主变量。

另一种变通方案是在类型char[]或VARCHAR[]的主变量中存储数组的外部字符串表达。关于这种表达的详情请见Section 8.15.2 注意这意味着该数组无法作为一个主程序中的数组被自然地访问（没有解析文本表达的进一步处理）。

#### 34.4.5.2. 组合类型

ECPG 中并不直接支持组合类型，但是有一种可能的简单变通方案。可用的变通方案和上述用于数组的方案相似：要么单独访问每一个属性或者使用外部字符串表达。

对于下列例子，假定有下面的类型和表：

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'LightDB') );
```

最显而易见的解决方案是单独访问每一个属性。下面的程序通过单独选择类型comp\_t的每一个属性从例子表中检索数据：

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;

/* 将组合类型列的每一个元素放在 SELECT 列表中。 */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM
t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将组合类型列的每一个元素取到主变量中。 */
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}
```



```
EXEC SQL CLOSE cur1;
```

为了加强这个例子，在FETCH命令中存储值的主变量可以被集中在一个结构中。结构形式的主变量的详情可见Section 34.4.4.3.2 要切换到结构形式，该例子可以被改成下面的样子。两个主变量intval和textval变成comp\_t结构的成员，并且该结构在FETCH命令中指定。

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;

/* 将组合类型列的每一个元素放在 SELECT 列表中。 */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM
t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将 SELECT 列表中的所有值放入一个结构。 */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}

EXEC SQL CLOSE cur1;
```

尽管在FETCH命令中使用了一个结构，SELECT子句中的属性名还是要一个一个指定。可以通过使用一个\*来要求该组合类型值的所有属性来改进。

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* 将 SELECT 列表中的所有值放入一个结构。 */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}
...
```

通过这种方法，即便 ECPG 不理解组合类型本身，组合类型也能够几乎无缝地被映射到结构。

最后，也可以在类型char[]或VARCHAR[]的主变量中把组合类型值存储成它们的外部字符串表达。但是如果使用那种方法，就不太可能从主程序中访问该值的各个域了。

### 34.4.5.3. 用户定义的基础类型

ECPG 并不直接支持新的用户定义的基本类型。你可以使用外部字符串表达以及类型 `char[]` 或 `VARCHAR[]` 的主变量，并且这种方案事实上对很多类型都是合适和足够的。

这里有一个使用来自 Section 36.1 中例子里的数据类型 `complex` 的例子。该类型的外部字符串表达是 `(%f,%f)`，它被定义在函数 `complex_in()` 以及 Section 36.1 中的 `complex_out()` 函数内。下面的例子把复杂类型值 `(1, 1)` 和 `(3, 3)` 插入到列 `a` 和 `b`，并且之后把它们从表中选择出来。

```
EXEC SQL BEGIN DECLARE SECTION;
    varchar a[64];
    varchar b[64];
EXEC SQL END DECLARE SECTION;

EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :a, :b;
    printf("a=%s, b=%s\n", a.arr, b.arr);
}

EXEC SQL CLOSE cur1;
```

这个例子会显示下列结果：

```
a=(1,1), b=(3,3)
```

另一种变通方案是避免在 ECPG 中直接使用用户定义的类型，而是创建一个在用户定义的类型和 ECPG 能处理的简单类型之间转换的函数或者造型。不过要注意，在类型系统中引入类型造型（特别是隐式造型）要非常小心。

例如，

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0)' + $2 * complex '(0,1)' $$;
```

在这个定义之后，下面的语句

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;
```

```
EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b),
      create_complex(:c, :d));
```

具有和

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)');
```

相同的效果。

## 34.4.6. 指示符

上述例子并没有处理空值。事实上，如果检索的例子从数据库取到了一个空值，它们将会产生一个错误。要能够向数据库传递空值或者从数据库检索空值，你需要对每一个包含数据的主变量追加一个次要主变量说明。这个次要主变量被称为指示符并且包含一个说明数据是否为空的标志，如果为空真正的主变量中的值就应该被忽略。这里有一个能正确处理检索空值的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION;
```

...

```
EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

如果值不为空，指示符变量val\_ind将为零；否则它将为负值。

指示符有另一种功能：如果指示符值为正，它表示值不为空，但是当它被存储在主变量中时已被截断。

如果参数-r no\_indicator被传递给预处理器ecpg，它会工作在“无指示符”模式。在无指示符模式中，如果没有指定指示符变量，对于字符串类型空值被标志（在输入和输出上）为空串，对于整数类型空值被标志为类型的最低可能值（例如，int的是INT\_MIN）。

## 34.5. 动态 SQL

在很多情况中，一个应用必须要执行的特定 SQL 语句在编写该应用时就已知。不过在某些情况中，SQL 语句在运行时构造或者由一个外部来源提供。这样你就不能直接把 SQL 语句嵌入到 C 源代码，不过有一种功能允许你调用在一个字符串变量中提供的任意 SQL 语句。

### 34.5.1. 执行没有结果集的语句

执行一个任意 SQL 语句的最简单方法是使用命令EXECUTE IMMEDIATE。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL EXECUTE IMMEDIATE :stmt;
```

EXECUTE IMMEDIATE可以被用于不返回结果集的 SQL 语句（例如 DDL、INSERT、UPDATE、DELETE）。你不能用这种方法执行检索数据的语句（例如SELECT）。下一节将描述如何执行这一种语句。

### 34.5.2. 执行一个有输入参数的语句

执行任意 SQL 语句的一种更强大的方法是准备它们一次并且在每次需要时执行该预备语句。也可以准备一个一般化的语句，然后通过替换参数执行它的特定版本。在准备语句时，在你想要稍后替换参数的地方写上问号。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?)";
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL PREPARE mystmt FROM :stmt;
...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

当你不再需要预备语句时，你应该释放它：

```
EXEC SQL DEALLOCATE PREPARE name;
```

### 34.5.3. 执行一个有结果集的语句

要执行一个只有单一结果行的 SQL 语句，可以使用EXECUTE。要保存结果，在其中增加一个INTO子句。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL PREPARE mystmt FROM :stmt;
...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

一个EXECUTE命令可以有一个INTO子句、一个USING子句，可以同时有这两个子句，也可以不带这两个子句。

如果一个查询被期望返回多于一个结果行，应该如下列例子所示使用一个游标（关于游标详见Section 34.3.2）。

```
EXEC SQL BEGIN DECLARE SECTION;
char dname[128];
char datname[128];
char *stmt = "SELECT u.username as dname, d.datname "
            " FROM pg_database d, pg_user u "
            " WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
COMMIT;

EXEC SQL PREPARE stmt1 FROM :stmt;
```

```

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname, :datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;

```

### 34.5.4. 执行一个带输入输出参数的 Do Statement

目前带输入输出参数的 do statement 只支持 language 为 plorasql。

```

EXEC SQL BEGIN DECLARE SECTION;
int id = 1;
int iReturnCode = -1;
EXEC SQL END DECLARE SECTION;

EXEC SQL create table t1(
    id integer,
    val text);

EXEC SQL insert into t1 values
(1, 'a'),
(2, null),
(4, 'd');

EXEC SQL
Do $$
begin
    begin
        update t1 set val = 'C' where id = :id;

        :iReturnCode = 0;
        :id = :id + 1;

        if SQL%NOTFOUND then
            :iReturnCode = 123;
        end if;

        exception
            when others then
                :iReturnCode = 456;
        end;
    end;
end;
$$ Language plorasql;

```

每个以 ':' 开头的变量都被识别为一个绑定变量，并且为输入输出模式。

## 34.6. pgtypes 库

pgtypes 库将LightDB数据库类型映射到 C 中等价的类型以便在 C 程序中使用。它还提供在 C 中对这些类型进行基本计算的函数，即不依赖LightDB服务器进行计算。请看下面的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
    date date1;
    timestamp ts1, tsout;
    interval iv1;
    char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
PGTYPESchar_free(out);
```

### 34.6.1. 字符串

PGTYPESnumeric\_to\_asc之类的一些函数返回一个新分配的字符串的指针。这些结果应该用PGTYPESchar\_free而不是free释放（这只在Windows上很重要，因为Windows上的内存分配和释放有时候需要由同一个库完成）。

### 34.6.2. numeric类型

numeric类型用来完成对任意精度的计算。LightDB服务器中等效的类型请见Section 8.1 因为要用于任意精度，这种变量需要能够动态地扩展和收缩。这也是为什么你只能用PGTYPESnumeric\_new和PGTYPESnumeric\_free函数在堆上创建numeric变量。decimal类型与numeric类型相似但是在精度上有限制，decimal类型可以在堆上创建也可以在栈上创建。

下列函数可以用于numeric类型：

PGTYPESnumeric\_new

请求一个指向新分配的numeric变量的指针。

```
numeric *PGTYPESnumeric_new(void);
```

PGTYPESnumeric\_free

释放一个numeric类型，释放它所有的内存。

```
void PGTYPESnumeric_free(numeric *var);
```

PGTYPESnumeric\_from\_asc

从字符串记号中解析一个numeric类型。

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

例如，可用的格式是： -2、 .794、 +3.44、 592.49E07或者 -32.84e-4。  
如果值能被成功地解析，将返回一个有效的指针，否则返回 NULL 指针。目

前 ECPG 总是解析整个字符串并且因此当前不支持把第一个非法字符的地址存储在\*endpstr中。你可以安全地把endpstr设置为 NULL。

#### PGTYPEStnumeric\_to\_asc

返回由malloc分配的字符串的指针，它包含numeric类型num的字符串表达。

```
char *PGTYPEStnumeric_to_asc(numeric *num, int dscale);
```

numeric值将被使用dscale小数位打印，必要时会圆整。结果必须用PGTYPEStchar\_free()释放。

#### PGTYPEStnumeric\_add

把两个numeric变量相加放到第三个numeric变量中。

```
int PGTYPEStnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

该函数把变量var1和var2相加放到结果变量result中。成功时该函数返回 0，出错时返回 -1。

#### PGTYPEStnumeric\_sub

把两个numeric变量相减并且把结果返回到第三个numeric变量。

```
int PGTYPEStnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

该函数把变量var2从变量var1中减除。该操作的结果被存储在变量result中。成功时该函数返回 0，出错时返回 -1。

#### PGTYPEStnumeric\_mul

把两个numeric变量相乘并且把结果返回到第三个numeric变量。

```
int PGTYPEStnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

该函数把变量var1和var2相乘。该操作的结果被存储在变量result中。成功时该函数返回 0，出错时返回 -1。

#### PGTYPEStnumeric\_div

把两个numeric变量相除并且把结果返回到第三个numeric变量。

```
int PGTYPEStnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

该函数用变量var2除变量var1。该操作的结果被存储在变量result中。成功时该函数返回 0，出错时返回 -1。

#### PGTYPEStnumeric\_cmp

比较两个numeric变量。

```
int PGTYPEStnumeric_cmp(numeric *var1, numeric *var2)
```

这个函数比较两个numeric变量。错误时会返回INT\_MAX。成功时，该函数返回三种可能结果之一：

- var1大于var2则返回 1

- 如果var1小于var2则返回 -1
- 如果var1和var2相等则返回 0

#### PGTYPEsnumeric\_from\_int

把一个整数变量转换成一个numeric变量。

```
int PGTYPEsnumeric_from_int(signed int int_val, numeric *var);
```

这个函数接受一个有符号整型变量并且把它存储在numeric变量var中。成功时返回 0，失败时返回 -1。

#### PGTYPEsnumeric\_from\_long

把一个长整型变量转换成一个numeric变量。

```
int PGTYPEsnumeric_from_long(signed long int long_val, numeric *var);
```

这个函数接受一个有符号长整型变量并且把它存储在numeric变量var中。成功时返回 0，失败时返回 -1。

#### PGTYPEsnumeric\_copy

把一个numeric变量复制到另一个中。

```
int PGTYPEsnumeric_copy(numeric *src, numeric *dst);
```

这个函数把src指向的变量的值复制到dst指向的变量中。成功时该函数返回 0，出错时返回 -1。

#### PGTYPEsnumeric\_from\_double

把一个双精度类型的变量转换成一个numeric变量。

```
int PGTYPEsnumeric_from_double(double d, numeric *dst);
```

这个函数接受一个双精度类型的变量并且把结果存储在dst指向的变量中。成功时该函数返回 0，出错时返回 -1。

#### PGTYPEsnumeric\_to\_double

将一个numeric类型的变量转换成双精度。

```
int PGTYPEsnumeric_to_double(numeric *nv, double *dp)
```

这个函数将nv指向的变量中的numeric值转换成dp指向的双精度变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES\_NUM\_OVERFLOW。

#### PGTYPEsnumeric\_to\_int

将一个numeric类型的变量转换成整数。

```
int PGTYPEsnumeric_to_int(numeric *nv, int *ip);
```

该函数将nv指向的变量的numeric值转换成ip指向的整数变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES\_NUM\_OVERFLOW。



## PGTYPESnumeric\_to\_long

将一个numeric类型的变量转换成长整型。

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

该函数将nv指向的变量的numeric值转换成lp指向的长整型变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES\_NUM\_OVERFLOW。

## PGTYPESnumeric\_to\_decimal

将一个numeric类型的变量转换成decimal。

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

该函数将nv指向的变量的numeric值转换成ip指向的decimal变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。溢出时，全局变量errno将被额外地设置成PGTYPES\_NUM\_OVERFLOW。

## PGTYPESnumeric\_from\_decimal

将一个decimal类型的变量转换成numeric。

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

该函数将nv指向的变量的decimal值转换成ip指向的numeric变量。成功时该函数返回 0，出错时返回 -1（包括溢出）。因为decimal类型被实现为numeric类型的一个有限的版本，在这个转换上不会发生溢出。

### 34.6.3. 日期类型

C 中的日期类型允许你的程序处理 SQL 日期类型的数据。LightDB服务器的等效类型可见Section 8.5

下列函数可以被用于日期类型：

## PGTYPESdate\_from\_timestamp

从一个时间戳中抽取日期部分。

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

该函数接收一个时间戳作为它的唯一参数并且从这个时间戳返回抽取的日期部分。

## PGTYPESdate\_from\_asc

从日期的文本表达解析一个日期。

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

该函数接收一个 C 的字符串str以及一个指向 C 字符串的指针endptr。当前 ECPG 总是解析完整的字符串并且因此当前不支持将第一个非法字符的地址存储在\*endptr中。你可以安全地把endptr设置为 NULL。

注意该函数总是假定格式按照 MDY 格式化并且当前在 ECPG 中没有变体可以改变这种格式。

Table 34. 展示了所有允许的输入格式。

Table 34.2. PGTYPEStime\_from\_asc的合法输入格式

输入	结果
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	年以及积日
J2451187	儒略日
January 8, 99 BC	公元前 99 年

## PGTYPEStime\_to\_asc

返回一个日期变量的文本表达。

```
char *PGTYPEStime_to_asc(date dDate);
```

该函数接收日期dDate作为它的唯一参数。它将以形式1999-01-18输出该日期，即以YYYY-MM-DD格式输出。结果必须用PGTYPEStime\_free()释放。

## PGTYPEStime\_julmdy

从一个日期类型变量中抽取日、月和年的值。

```
void PGTYPEStime_julmdy(date d, int *mdy);
```

该函数接收日期d以及一个指向有 3 个整数值的数组mdy的指针。变量名就表明了顺序：mdy[0]将被设置为包含月份，mdy[1]将被设置为日的值，而mdy[2]将包含年。

## PGTYPEStime\_mdyjul

从一个由 3 个整数构成的数组创建一个日期值，3 个整数分别指定日、月和年。

```
void PGTYPEStime_mdyjul(int *mdy, date *jdate);
```

这个函数接收 3 个整数 (mdy) 组成的数组作为其第一个参数，其第二个参数是一个指向日期类型变量的指针，它被用来保存操作的结果。

## PGTYPEStime\_dayofweek

为一个日期值返回表示它是星期几的数字。

```
int PGTYPEdate_dayofweek(date d);
```

这个函数接收日期变量d作为它唯一的参数并且返回一个整数说明这个日期是星期几。

- 0 - 星期日
- 1 - 星期一
- 2 - 星期二
- 3 - 星期三
- 4 - 星期四
- 5 - 星期五
- 6 - 星期六

```
PGTYPEdate_today
```

得到当前日期。

```
void PGTYPEdate_today(date *d);
```

该函数接收一个指向一个日期变量（d）的指针并且把该参数设置为当前日期。

```
PGTYPEdate_fmt_asc
```

使用一个格式掩码将一个日期类型的变量转换成它的文本表达。

```
int PGTYPEdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

该函数接收要转换的日期（dDate）、格式掩码（fmtstring）以及将要保存日期的文本表达的字符串（outbuf）。

成功时，返回 0；如果发生错误，则返回一个负值。

下面是你可以使用的域指示符：

- dd - 一个月中的第几天。
- mm - 一年中的第几个月。
- yy - 两位数的年份。
- yyyy - 四位数的年份。
- ddd - 星期几的名称（简写）。
- mmm - 月份的名称（简写）。

所有其他字符会被原封不动地复制到输出字符串中。

Table 34. 指出了一些可能的格式。这将给你一些线索如何使用这个函数。所有输出都是基于同一个日期：1959年11月23日。

Table 34.3. PGTYPEdate\_fmt\_asc的合法输入格式

格式	结果
mmddy	112359
ddmmy	231159

格式	结果
yymmdd	591123
yy/mm/dd	59/11/23
yy mm dd	59 11 23
yy. mm. dd	59. 11. 23
. mm. yyyy. dd.	. 11. 1959. 23.
mmm. dd, yyyy	Nov. 23, 1959
mmm dd yyyy	Nov 23 1959
yyyy dd mm	1959 23 11
ddd, mmm. dd, yyyy	Mon, Nov. 23, 1959
(ddd) mmm. dd, yyyy	(Mon) Nov. 23, 1959

## PGTYPESdate\_defmt\_asc

使用一个格式掩码把一个 C 的 char\*子返回串转换成一个日期类型的值。

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

该函数接收一个用来保存操作结果的指向日期值的指针 (d)、用于解析日期的格式掩码 (fmt) 以及包含日期文本表达的 C char\* 串 (str)。该函数期望文本表达匹配格式掩码。不过你不需要字符串和格式掩码的一一映射。该函数只分析相继顺序并且查找表示年份位置的数字yy或者yyyy、表示月份位置的mm以及表示日位置的dd。

Table 34. 给出了一些可能的格式。这将给你一些线索如何使用这个函数。

Table 34.4. rdefmtdate的合法输入格式

格式	字符串	结果
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13
mmm. dd. yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03
mmm. dd. yyyy	041269	1969-04-12
yy/mm/dd	在 2525 年的七月二十八日, 人类还将存在	2525-07-28
dd-mm-yy	也是 2525 年七月的二十八日	2525-07-28
mmm. dd. yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm. dd. yyyy	oct 28 1975	1975-10-28
mmddy	Nov 14th, 1985	1985-11-14

### 34.6.4. 时间戳类型

C 中的时间戳类型允许你的程序处理 SQL 时间戳类型的数据。LightDB服务器的等效类型可见Section 8.5

下列函数可用于时间戳类型：

#### PGTYPEStimestamp\_from\_asc

从文本表达解析一个时间戳并放到一个时间戳变量中。

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

这个函数接收一个要解析的字符串（str）以及一个 C char\* 的指针（endptr）。当前 ECPG 总是解析完整的字符串并且因此当前不支持将第一个非法字符的地址存储在\*endptr中。你可以安全地把endptr设置为 NULL。

成功时该函数返回解析到的时间戳。错误时，会返回PGTYPEStimestamp\_invalid并且errno会被设置为PGTYPEStimestamp\_invalid\_TS\_BAD\_TIMESTAMP。关于这个值的重要提示请见PGTYPEStimestamp\_invalid。

通常，该输入字符串能够包含一个允许的日期说明、一个空格字符和一个允许的时间说明的任意组合。注意 ECPG 不支持时区。它能够解析时区但是不会应用任何计算（例如 LightDB服务器所作的事情）。时区指示符会被无声无息地丢弃。

Table 34. 包含输入字符串的一些例子。

Table 34.5. PGTYPEStimestamp\_from\_asc的合法输入格式

输入	结果
1999-01-08 04:05:06	1999-01-08 04:05:06
January 8 04:05:06 1999 PST	1999-01-08 04:05:06
1999-Jan-08 04:05:06.789-8	1999-01-08 04:05:06.789 （忽略了时区指示符）
J2451187 04:05-08:00	1999-01-08 04:05:00 （忽略了时区指示符）

#### PGTYPEStimestamp\_to\_asc

将一个日期转换成一个 C char\* 字符串。

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

该函数接收时间戳tstamp作为它的唯一参数并且返回一个分配好的包含该时间戳文本表达的字符串。结果必须用PGTYPEStimestamp\_free()释放。

#### PGTYPEStimestamp\_current

检索当前的时间戳。

```
void PGTYPEStimestamp_current(timestamp *ts);
```

该函数检索当前的时间戳并且将它保存在ts指向的时间戳变量。

#### PGTYPEStimestamp\_fmt\_asc

使用一个格式掩码将一个时间戳变量转换成一个 C char\* 。

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

该函数接收一个指向时间戳的指针作为它的第一个参数（ts）、一个指向输出缓冲区的指针（output）、为输出缓冲区分配的最大长度（str\_len）以及用于转换的格式掩码（fmtstr）。

成功时，该函数返回 0；如果有错误发生，则返回一个负值。

你可以为格式掩码使用下列格式指示符。格式指示符就是用在libc的strftime函数中的那一些。任何非格式指示符将被复制到输出缓冲区。

- %A - 被完整的星期几名称的本国表达所替换。
- %a - 被简写星期几名称的本国表达所替换。
- %B - 被完整的月份名称的本国表达所替换。
- %b - 被简写月份名称的本国表达所替换。
- %C - 被十进制数（年份/100）所替换，单一数字会被前置一个零。
- %c - 被时间和日期的本国表达所替换。
- %D - 等效于%m/%d/%y。
- %d - 被十进制数（01 - 31）的日所替换。
- %E\* %O\* - POSIX 区域扩展。序列 %Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy 被假定提供可供选择的表达。

此外还实现了%OB来表达可供选择的月份名称（单独使用，不用提过的日）。

- %e - 被十进制数（1 - 31）的日所替换，单一数字被前置一个空格。
- %F - 等效于%Y-%m-%d。
- %G - 被替换为一个带有世纪的十进制数年份。这个年份是包含这一周大部分的年份（星期一作为这一周的第一天）。
- %g - 被替换为与%G中相同的年份，但是作为一个不带世纪的十进制数（00 - 99）。
- %H - 被替换为一个十进制数的小时（24 小时制，00 - 23）。
- %h - 和%b相同。
- %I - 被替换为一个十进制数的小时（12 小时制，01 - 12）。
- %j - 被替换为一个十进制数的积日（001 - 366）。
- %k - 被替换为一个十进制数的小时（24 小时制，0 - 23），单一数字被前置一个空白。
- %l - 被替换为一个十进制数的小时（12 小时制，1 - 12），单一数字被前置一个空白。
- %M - 被替换为一个十进制数的分钟（00 - 59）。
- %m - 被替换为一个十进制数的月份（01 - 12）。
- %n - 被替换为一个新行。
- %O\* - 和%E\*相同。
- %p - 根据情况被替换为“午前”或“午后”的本国表达。
- %R - 等效于%H:%M。

- %r - 等效于%I:%M:%S%p。
- %S - 被替换为十进制数的秒（00 - 60）。
- %s - 被替换为从 UTC 新纪元以来的秒数。
- %T - 等效于%H:%M:%S
- %t - 被替换为一个制表符。
- %U - 被替换为十进制数的周数（周日作为一周的第一天，00 - 53）。
- %u - 被替换为十进制数的星期几（周一作为一周的第一天，1 - 7）。
- %V - 被替换为十进制数的周数（周一作为一周的第一天，01 - 53）。如果包含 1 月 1 日的周在新一年中有 4 天或更多天，那么它是第一周。否则它是前一年的最后一周，并且下一周是第一周。
- %v - 等效于%e-%b-%Y。
- %W - 被替换为十进制数的周数（周一作为一周的第一天，00 - 53）。
- %w - 被替换为十进制数的星期几（0 - 6，周日作为一周的第一天）。
- %X - 被替换为时间的本国表达。
- %x - 被替换为日期的本国表达。
- %Y - 被替换为十进制数的带世纪的年份。
- %y - 被替换为十进制数的不带世纪的年份（00 - 99）。
- %Z - 被替换为时区名称。
- %z - 被替换为相对于 UTC 的时区偏移；一个前导的加号表示 UTC 东部，一个负号表示 UTC 西部，接着是分别有两个数字的小时和分钟并且它们之间没有定界符（RFC 822 日期头部的一般形式）。
- %+ - 被替换为日期和时间的本国表达。
- %-\* - GNU libc 扩展。在执行数值输出时不做任何填充。
- \$\_\* - GNU libc 扩展。显式地指定用空格填充。
- %0\* - GNU libc 扩展。显式地指定用零填充。
- %% - 被替换为%。

#### PGTYPEStimestamp\_sub

从一个时间戳中减去另一个时间戳并且把结果保存在一个区间类型的变量中。

```
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

该函数将从ts1指向的时间戳变量中减去ts2指向的时间戳变量，并且将把结果存储在iv指向的区间变量中。

成功时，该函数返回 0；发生错误时则返回一个负值。

#### PGTYPEStimestamp\_defmt\_asc

用一个格式掩码从时间戳的文本表达解析其值。

```
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

该函数接收一个放在变量str中的时间戳文本表达以及放在变量fmt中的要使用的格式掩码。结果将被存放在d指向的变量中。

如果格式掩码fmt是NULL，该函数将回退到使用默认的格式掩码%Y-%m-%d %H:%M:%S。

这是PGTYPEStimestamp\_fmt\_asc的逆函数。可能的格式掩码项可以参考那个函数的文档。

PGTYPEStimestamp\_add\_interval

把一个interval变量加到一个时间戳变量上。

```
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

该函数接收一个指向时间戳变量的指针tin以及一个指向interval变量的指针span。它把interval加到时间戳上，然后将结果时间戳保存在tout指向的变量中。

成功时该函数返回0，如果发生错误则返回一个负值。

PGTYPEStimestamp\_sub\_interval

从一个时间戳变量中减去一个interval变量。

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

该函数从tin指向的时间戳变量中减去span指向的interval变量，然后把结果保存在tout指向的变量中。

成功时该函数返回0，如果发生错误则返回一个负值。

### 34.6.5. 区间类型

C 中的区间类型允许你的程序处理 SQL 区间类型的数据。LightDB服务器的等效类型可见Section 8.5

下列函数可以用于区间类型：

PGTYPEStimestamp\_new

返回一个指向新分配的区间变量的指针。

```
interval *PGTYPEStimestamp_new(void);
```

PGTYPEStimestamp\_free

释放先前分配的区间变量的内存。

```
void PGTYPEStimestamp_free(interval *intvl);
```

PGTYPEStimestamp\_from\_asc

从文本表达解析一个区间。



```
interval *PGTYPEInterval_from_asc(char *str, char **endptr);
```

该函数解析输入字符串`str`并且返回一个已分配的区间变量的指针。目前 ECPG 总是解析整个字符串并且因此当前不支持把第一个非法字符的地址存储在`*endptr`中。你可以安全地把`endptr`设置为 `NULL`。

PGTYPEInterval\_to\_asc

将一个区间类型的变量转换成它的文本表达。

```
char *PGTYPEInterval_to_asc(interval *span);
```

该函数将`span`指向的区间变量转换成一个 `C char*`。输出看起来像这个例子： `@ 1 day 12 hours 59 mins 10 secs`。结果必须用`PGTYPEChar_free()`释放。

PGTYPEInterval\_copy

复制一个区间类型的变量。

```
int PGTYPEInterval_copy(interval *intvlsrc, interval *intvldest);
```

该函数将`intvlsrc`指向的区间变量复制到`intvldest`指向的区间变量。注意你需要现为目标变量分配好内存。

### 34.6.6. decimal类型

`decimal`类型和`numeric`类型相似。不过，它被限制为最大精度是 30 个有效位。与`numeric`类型只能在堆上创建相反，`decimal`类型既可以在栈上也可以在堆上创建（使用函数`PGTYPEdecimal_new` 和`PGTYPEdecimal_free`）。在Section 34.15中描述的Informix兼容模式中有很多其它函数可以处理`decimal`类型。

下列函数可以被用于`decimal`类型并且不仅被包含于`libcompat`库中。

PGTYPEdecimal\_new

要求一个指向新分配的`decimal`变量的指针。

```
decimal *PGTYPEdecimal_new(void);
```

PGTYPEdecimal\_free

释放一个`decimal`类型，释放它的所有内存。

```
void PGTYPEdecimal_free(decimal *var);
```

### 34.6.7. pgtypeslib 的 errno 值

PGTYPES\_NUM\_BAD\_NUMERIC

一个参数应该包含一个`numeric`变量（或者指向一个`numeric`变量），但是实际上它的内存表达非法。

PGTYPES\_NUM\_OVERFLOW

发生一次溢出。由于`numeric`类型可以处理几乎任何精度，将一个`numeric`变量转换成其他类型可能导致溢出。

**PGTYPES\_NUM\_UNDERFLOW**

发生一次下溢。由于numeric类型可以处理几乎任何精度，将一个numeric变量转换成其他类型可能导致下溢。

**PGTYPES\_NUM\_DIVIDE\_ZERO**

尝试了一次除零。

**PGTYPES\_DATE\_BAD\_DATE**

一个非法的日期字符串被传给了PGTYPESdate\_from\_asc函数。

**PGTYPES\_DATE\_ERR\_EARGS**

非法参数被传给了PGTYPESdate\_defmt\_asc函数。

**PGTYPES\_DATE\_ERR\_ENOSHORTDATE**

PGTYPESdate\_defmt\_asc函数在输入字符串中发现了一个非法记号。

**PGTYPES\_INTVL\_BAD\_INTERVAL**

一个非法的区间字符串被传给了PGTYPESinterval\_from\_asc函数，或者一个非法的区间值被传给了PGTYPESinterval\_to\_asc函数。

**PGTYPES\_DATE\_ERR\_ENOTDMY**

在PGTYPESdate\_defmt\_asc函数中有日/月/年不匹配的赋值。

**PGTYPES\_DATE\_BAD\_DAY**

PGTYPESdate\_defmt\_asc函数发现了月中的一个非法日值。

**PGTYPES\_DATE\_BAD\_MONTH**

PGTYPESdate\_defmt\_asc函数发现了一个非法的月值。

**PGTYPES\_TS\_BAD\_TIMESTAMP**

一个非法的时间戳字符串被传给了PGTYPEStimestamp\_from\_asc函数，或者一个非法的时间戳值被传给了PGTYPEStimestamp\_to\_asc函数。

**PGTYPES\_TS\_ERR\_EINFTIME**

在一个无法处理无限时间戳值的环境中遇到了这样一个值。

## 34.6.8. pgtypeslib 的特殊常量

**PGTYPESInvalidTimestamp**

表示一个非法时间戳的时间戳类型值。在解析错误时，函数PGTYPEStimestamp\_from\_asc会返回这个值。注意由于timestamp数据类型的内部表达，PGTYPESInvalidTimestamp在同时也是一个合法的时间戳。它被设置为1899-12-31 23:59:59。为了检测到错误，确认你的应用在每次调用PGTYPEStimestamp\_from\_asc后不仅仅测试PGTYPESInvalidTimestamp，还应该测试errno != 0。

## 34.7. 使用描述符区域

一个 SQL 描述符区域是一种处理SELECT、FETCH或者DESCRIBE语句结果的高级方法。一个SQL 描述符区域把数据中一行的数据及元数据项组合到一个数据结构中。在执行动态 SQL 语句时（结果行的性质无法提前预知），元数据特别有用。LightDB 提供两种方法来使用描述符区域：命名 SQL 描述符区域和 C 结构 SQLDA。

### 34.7.1. 命名 SQL 描述符区域

一个命名 SQL 描述符区域由一个头部以及一个或多个条目描述符区域构成，头部包含与整个描述符相关的信息，而条目描述符区域则描述结果行中的每一列。

在使用 SQL 描述符区域之前，需要先分配一个：

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

identifier 会作为该描述符区域的“变量名”。当不再需要该描述符时，应当释放它：

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

要使用一个描述符区域，把它指定为INTO子句的存储目标（而不是列出主变量）：

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

如果结果集为空，该描述符区域仍然会包含查询的元数据，即域的名称。

对于还没有执行的预备查询，DESCRIBE可以被用来得到其结果集的元数据：

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

在 LightDB 9.0 之前，SQL关键词是可选的，因此使用DESCRIPTOR和SQL DESCRIPTOR都会产生命名 SQL 描述符区域。现在该关键词是强制性的，省略SQL关键词会产生 SQLDA 描述符区域（见Section 34.7.2）。

在DESCRIBE和FETCH语句中，INTO和USING关键词的使用相似：它们产生结果集以及一个描述符区域中的元数据。

现在我们如何从描述符区域得到数据呢？你可以把描述符区域看成是一个具有命名域的结构。要从头部检索一个域的值并且把它存储到一个主变量中，可使用下面的命令：

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

当前，只定义了一个头部域：COUNT，它告诉我们有多少个条目描述符区域（也就是，结果中包含多少列）。主变量需要是一个整数类型。要从条目描述符区域中得到一个域，可使用下面的命令：

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

num可以是一个字面整数或者包含一个整数的主变量。可能的域有：

CARDINALITY （整数）

    结果集中的行数

DATA

    实际的数据项（因此，这个域的数据类型取决于查询）

DATETIME\_INTERVAL\_CODE (整数)

当TYPE是9时， DATETIME\_INTERVAL\_CODE将具有以下值之一： 1 表示 DATE， 2 表示 TIME， 3 表示 TIMESTAMP， 4 表示 TIME WITH TIME ZONE， 5 表示 TIMESTAMP WITH TIME ZONE。

DATETIME\_INTERVAL\_PRECISION (整数)

没有实现

INDICATOR (整数)

指示符 (表示一个空值或者一个值截断)

KEY\_MEMBER (整数)

没有实现

LENGTH (整数)

以字符计的数据长度

NAME (string)

列名

NULLABLE (整数)

没有实现

OCTET\_LENGTH (整数)

以字节计的数据字符表达的长度

PRECISION (整数)

精度 (用于类型numeric)

RETURNED\_LENGTH (整数)

以字符计的数据长度

RETURNED\_OCTET\_LENGTH (整数)

以字节计的数据字符表达的长度

SCALE (整数)

比例 (用于类型numeric)

TYPE (整数)

列的数据类型的数字编码

在EXECUTE、DECLARE以及OPEN语句中，INTO和USING关键词的效果不同。也可以手工建立一个描述符区域来为一个查询或者游标提供输入参数，并且USING SQL DESCRIPTOR name是用来传递输入参数给参数化查询的方法。建立一个命名 SQL 描述符区域的语句如下：

```
EXEC SQL SET DESCRIPTOR name VALUE num field = :hostvar;
```

LightDB 支持在一个FETCH语句中检索多于一个记录并且在这种情况下把主变量假定为一个数组来存储数据。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

## 34.7.2. SQLDA 描述符区域

SQLDA 描述符区域是一个 C 语言结构，它也能被用来得到一个查询的结果集和元数据。一个结构存储一个来自结果集的记录。

```
EXEC SQL include sqlda.h;
sqlda_t      *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

注意SQL关键词被省略了。Section 34.7. 中关于INTO和USING关键词用例的段落一定条件下也适用于这里。在一个DESCRIBE语句中，如果使用了INTO关键词，则DESCRIPTOR关键词可以完全被省略：

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

使用 SQLDA 的程序的一般流程是：

1. 准备一个查询，并且为它声明一个游标。
2. 为结果行声明一个 SQLDA 。
3. 为输入参数声明一个 SQLDA，并且初始化它们（内存分配、参数设置）。
4. 用输入 SQLDA 打开一个游标。
5. 从游标中取得行，并且把它们存储到一个输出 SQLDA。
6. 从输出 SQLDA 读取值到主变量中（必要时使用转换）。
7. 关闭游标。
8. 关闭为输入 SQLDA 分配的内存区域。

### 34.7.2.1. SQLDA 数据结构

SQLDA 使用三种数据结构类型：sqlda\_t、sqlvar\_t以及struct sqlname。

#### Tip

LightDB 的 SQLDA 与 IBM DB2 Universal 数据库中相似的数据结构，因此一些 DB2 的 SQLDA 的技术信息有助于更好地理解 LightDB 的 SQLDA。

#### 34.7.2.1.1. sqlda\_t 结构

结构类型sqlda\_t是实际 SQLDA 的类型。它保存一个记录。并且两个或者更多个sqlda\_t结构能够以desc\_next域中的指针连接成一个链表，这样可以表示一个有序的行集合。因此，当两个或多个行被取得时，应用可以通过沿着每一个sqlda\_t节点中的desc\_next指针读取它们。

sqllda\_t的定义是:

```
struct sqllda_struct
{
    char          sqldaid[8];
    long          sqldabc;
    short         sqln;
    short         sqld;
    struct sqllda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqllda_struct sqllda_t;
```

域的含义是:

sqldaid

它包含一个字符串"SQLDA "。

sqldabc

它包含已分配空间的尺寸（以字节计）。

sqln

当它被传递给使用USING关键词的OPEN、DECLARE或者EXECUTE语句时，它包含用于一个参数化查询实例的输入参数的数目。在它被用作SELECT、EXECUTE或FETCH语句的输出时，它的值和sqld一样

sqld

它包含一个结果集中的域的数量。

desc\_next

如果查询返回不止一个记录，会返回多个链接在一起的 SQLDA 结构，并且desc\_next保存一个指向下一个项的指针。

sqlvar

这是结果集中列的数组。

#### 34.7.2.1.2. sqlvar\_t 结构

结构类型sqlvar\_t保存一个列值和元数据（例如类型和长度）。该类型的定义是:

```
struct sqlvar_struct
{
    short         sqltype;
    short         sqllen;
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

各个域的含义是:

sqltype

包含该域的类型标识符。值可以参考ecpgtype.h中的enum ECPGttype。

sqlllen

包含域的二进制长度，例如ECPGt\_int是 4 字节。

sqldata

指向数据。数据的格式在Section 34.4.4中描述。

sqlind

指向空指示符。0 表示非空，-1 表示空。

sqlname

域的名称。

### 34.7.2.1.3. struct sqlname 结构

一个struct sqlname结构保持一个列名。它被用作sqlvar\_t结构的一个成员。该结构的定义是：

```
#define NAMEDATALEN 64

struct sqlname
{
    short          length;
    char          data[NAMEDATALEN];
};
```

各个域的含义是：

length

包含域名称的长度。

data

包含实际的域名称。

### 34.7.2.2. 使用一个 SQLDA 检索一个结果集

通过一个 SQLDA 检索一个查询结果集的一般步骤是：

1. 声明一个sqllda\_t结构来接收结果集。
2. 执行 FETCH/EXECUTE/DESCRIBE 命令来处理一个指定已声明 SQLDA 的查询。
3. 通过查看sqllda\_t结构的成员sqln来检查结果集中记录的数量。
4. 从sqllda\_t结构的成员sqlvar[0]、sqlvar[1]等中得到每一列的值。
5. 沿着sqllda\_t结构的成员desc\_next指针到达下一行 (sqllda\_t)。
6. 根据你的需要重复上述步骤。

这里是一个通过 SQLDA 检索结果集的例子。

首先，声明一个sqllda\_t结构来接收结果集。

```
sqllda_t *sqllda1;
```

接下来，指定一个命令中的 SQLDA。这是一个FETCH命令的例子。

```
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqllda1;
```

运行一个循环顺着链表来检索行。

```
sqllda_t *cur_sqllda;
```

```
for (cur_sqllda = sqllda1;
     cur_sqllda != NULL;
     cur_sqllda = cur_sqllda->desc_next)
{
    ...
}
```

在循环内部，运行另一个循环来检索行中每一列的数据（sqlvar\_t结构）。

```
for (i = 0; i < cur_sqllda->sqlld; i++)
{
    sqlvar_t v = cur_sqllda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqlllen = v.sqlllen;
    ...
}
```

要得到一列的值，应检查sqlvar\_t结构的成员sqltype的值。然后，根据列类型切换到一种合适的方法从sqlvar域中复制数据到一个主变量。

```
char var_buf[1024];
```

```
switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqlllen ? sizeof(var_buf) -
1 : sqlllen));
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqlllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...
}
```

### 34.7.2.3. 使用一个 SQLDA 传递查询参数

使用一个 SQLDA 传递输入参数给一个预备查询的一般步骤是：

1. 创建一个预备查询（预备语句）。
2. 声明一个 sqllda\_t 结构作为输入 SQLDA。



3. 为输入 SQLDA 分配内存区域（作为 sqllda\_t 结构）。
4. 在分配好的内存中设置（复制）输入值。
5. 打开一个说明了输入 SQLDA 的游标。

这里是一个例子。

首先，创建一个预备语句。

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE
  d.oid = s.datid AND (d.datname = ? OR d.oid = ?)";
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL PREPARE stmt1 FROM :query;
```

接下来为一个 SQLDA 分配内存，并且在 sqllda\_t 结构的 sqln 成员变量中设置输入参数的数量。当预备查询要求两个或多个输入参数时，应用必须分配额外的内存空间，空间的大小为 (参数数目 - 1) \* sizeof(sqlvar\_t)。这里的例子展示了为两个输入参数分配内存空间。

```
sqllda_t *sqllda2;
```

```
sqllda2 = (sqllda_t *) malloc(sizeof(sqllda_t) + sizeof(sqlvar_t));
memset(sqllda2, 0, sizeof(sqllda_t) + sizeof(sqlvar_t));
```

```
sqllda2->sqln = 2; /* 输入变量的数目 */
```

内存分配之后，把参数值存储到 sqlvar[] 数组（当 SQLDA 在接收结果集时，这也是用来检索列值的数组）。在这个例子中，输入参数是“postgres”（字符串类型）和 1（整数类型）。

```
sqllda2->sqlvar[0].sqltype = ECPGt_char;
sqllda2->sqlvar[0].sqldata = "postgres";
sqllda2->sqlvar[0].sqlllen = 8;
```

```
int intval = 1;
sqllda2->sqlvar[1].sqltype = ECPGt_int;
sqllda2->sqlvar[1].sqldata = (char *) &intval;
sqllda2->sqlvar[1].sqlllen = sizeof(intval);
```

通过打开一个游标并且说明之前已经建立好的 SQLDA，输入参数被传递给预备语句。

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqllda2;
```

最后，用完输入 SQLDA 后必须显式地释放已分配的内存空间，这与用于接收查询结果的 SQLDA 不同。

```
free(sqllda2);
```

#### 34.7.2.4. 一个使用 SQLDA 的应用例子

这里是一个例子程序，它描述了如何按照输入参数的指定从系统目录中取得数据库的访问统计。

这个应用在数据库 OID 上连接两个系统表 (pg\_database 和 pg\_stat\_database)，并且还取得和显示通过两个输入参数（一个数据库 postgres 和 OID 1）检索到的数据库统计。

首先，为输入和输出分别声明一个 SQLDA。

```
EXEC SQL include sqlda.h;
```

```
sqlda_t *sqlda1; /* 一个输出描述符 */
sqlda_t *sqlda2; /* 一个输入描述符 */
```

接下来，连接到数据库，准备一个语句并且为预备语句声明一个游标。

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s
    WHERE d.oid=s.datid AND ( d.datname=? OR d.oid=? )";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
    COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

然后，为输入参数在输入 SQLDA 中放入一些值。为输入 SQLDA 分配内存，并且在sqln中设置输入参数的数目。在sqlvar结构的sqltype、sqldata和sqlllen中存入类型、值和值长度。

```
/* 为输入参数创建 SQLDA 结构。 */
sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
sqlda2->sqln = 2; /* 输入变量的数量 */

sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqlllen = 8;

intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *)&intval;
sqlda2->sqlvar[1].sqlllen = sizeof(intval);
```

设置完输入 SQLDA 之后，用输入 SQLDA 打开一个游标。

```
/* 用输入参数打开一个游标。 */
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

从打开的游标中取行到输出 SQLDA 中（通常，你不得不在循环中反复调用FETCH来取出结果集中的所有行）。

```
while (1)
{
    sqlda_t *cur_sqlda;

    /* 分配描述符给游标 */
    EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

再后，沿着sqllda\_t结构的链表从 SQLDA 中检索取得的记录。

```
for (cur_sqllda = sqllda1 ;
     cur_sqllda != NULL ;
     cur_sqllda = cur_sqllda->desc_next)
{
    ...
}
```

读取第一个记录中的每一列。列的数量被存储在sqlld中，第一列的实际数据被存储在sqlvar[0]中，两者都是sqllda\_t结构的成员。

```
/* 打印一行中的每一列。 */
for (i = 0; i < sqllda1->sqlld; i++)
{
    sqlvar_t v = sqllda1->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen = v.sqllen;

    strncpy(name_buf, v.sqlname.data, v.sqlname.length);
    name_buf[v.sqlname.length] = '\0';
}
```

现在，列数据已经被存在了变量v中。把每个数据复制到主变量中，列的类型可以查看。

```
switch (v.sqltype) {
    int intval;
    double doubleval;
    unsigned long long int longlongval;

    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ?
sizeof(var_buf)-1 : sqllen));
        break;

    case ECPGt_int: /* 整数 */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...

    default:
        ...
}

printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
}
```

处理所有记录后关闭游标，并且从数据库断开连接。

```
EXEC SQL CLOSE cur1;
EXEC SQL COMMIT;

EXEC SQL DISCONNECT ALL;
```

整个程序显示在Example 34. 中。

## Example 34.1. 示例 SQLDA 程序

```

#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* 用于输出的描述符 */
sqlda_t *sqlda2; /* 用于输入的描述符 */

EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s
WHERE d.oid=s.datid AND ( d.datname=? OR d.oid=? )";

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;

    /* 为一个输入参数创建一个 SQLDA 结构 */
    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* 输入变量的数量 */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqlllen = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *) &intval;
    sqlda2->sqlvar[1].sqlllen = sizeof(intval);

    /* 用输入参数打开一个游标。 */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;

    while (1)
    {
        sqlda_t *cur_sqlda;

        /* 给游标分配描述符 */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
    }
}

```

```

for (cur_sqlda = sqlda1 ;
     cur_sqlda != NULL ;
     cur_sqlda = cur_sqlda->desc_next)
{
    int i;
    char name_buf[1024];
    char var_buf[1024];

    /* 打印一行中的每一列。 */
    for (i=0 ; i<cur_sqlda->sqld ; i++)
    {
        sqlvar_t v = cur_sqlda->sqlvar[i];
        char *sqldata = v.sqldata;
        short sqllen = v.sqllen;

        strncpy(name_buf, v.sqlname.data, v.sqlname.length);
        name_buf[v.sqlname.length] = '\0';

        switch (v.sqltype)
        {
            case ECPGt_char:
                memset(&var_buf, 0, sizeof(var_buf));
                memcpy(&var_buf, sqldata, (sizeof(var_buf)<=sqllen ?
sizeof(var_buf)-1 : sqllen) );
                break;

            case ECPGt_int: /* 整数 */
                memcpy(&intval, sqldata, sqllen);
                snprintf(var_buf, sizeof(var_buf), "%d", intval);
                break;

            case ECPGt_long_long: /* 大整数 */
                memcpy(&longlongval, sqldata, sqllen);
                snprintf(var_buf, sizeof(var_buf), "%lld", longlongval);
                break;

            default:
                {
                    int i;
                    memset(var_buf, 0, sizeof(var_buf));
                    for (i = 0; i < sqllen; i++)
                    {
                        char tmpbuf[16];
                        snprintf(tmpbuf, sizeof(tmpbuf), "%02x ", (unsigned
char) sqldata[i]);
                        strcat(var_buf, tmpbuf, sizeof(var_buf));
                    }
                }
                break;
        }

        printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
    }

    printf("\n");
}
}

```

```
EXEC SQL CLOSE curl;
EXEC SQL COMMIT;

EXEC SQL DISCONNECT ALL;

return 0;
}
```

这个例子的输出应该看起来类似下面的结果（一些数字会变化）。

```
oid = 1 (type: 1)
datname = templat1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
datallowconn = t (type: 1)
datconlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig = (type: 1)
datacl = {=c/uptime,uptime=Ctc/uptime} (type: 1)
datid = 1 (type: 1)
datname = templat1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
datallowconn = t (type: 1)
datconlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig = (type: 1)
datacl = (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
```

```
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)
```

## 34.8. 错误处理

这一节描述在一个嵌入式 SQL 程序中如何处理异常情况和警告。有两种非互斥的工具可以用于这个目的。

- 可以使用WHENEVER命令配置回调来处理警告和错误情况。
- 可以从sqlca变量中获得错误或警告的详细信息。

### 34.8.1. 设置回调

一种捕捉错误和警告的简单方法是设置一个特殊的动作，只要一个特定情况发生就执行该动作。通常是这样：

```
EXEC SQL WHENEVER condition action;
```

condition可以是下列之一：

SQLERROR

只要在 SQL 语句执行期间发生一个错误就调用指定的动作。

SQLWARNING

只要在 SQL 语句执行期间发生一个警告就调用指定的动作。

NOT FOUND

只要一个 SQL 语句检索或者影响零行就调用指定的动作（这种情况不是一个错误，但是你可能需要特别地处理它）。

action可以是下列之一：

CONTINUE

这实际上表示该情况被忽略。这是默认值。

```
GOTO label
GO TO label
```

调到指定的标签（使用一个 C goto语句）。

SQLPRINT

把一个消息打印到标准错误。对于简单程序或原型开发中这很有用。消息的细节无法配置。

STOP

调用exit(1)终止程序。

DO BREAK

执行 C 语句break。只应被用在循环或switch语句中。

DO CONTINUE

执行C语句continue。这应该只被用在循环语句中。如果被执行，将导致控制流返回到循环的顶层。

```
CALL name (args)
DO name (args)
```

用指定参数调用指定的C函数（这种用法不同于正常LightDB语法中CALL和DO的含义）。

SQL 标准只提供动作CONTINUE和GOTO（以及GO TO）。

这里有一个可能会用在简单程序中的例子。当一个警告发生时它打印一个简单消息，而发生一个错误时它会中止程序：

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

语句EXEC SQL WHENEVER是 SQL 预处理器的一个指令，而不是一个 C 语句。不管 C 程序的控制流程如何，该语句设置的错误或警告动作适用于所有位于处理程序设置点之后的嵌入式 SQL 语句，除非在第一个EXEC SQL WHENEVER和导致情况的 SQL 语句之间为同一个情况设置了不同的动作。因此下面的两个 C 程序都不会得到预期的效果：

```
/*
 * 错误
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}

/*
 * 错误
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

### 34.8.2. sqlca

为了更强大的错误处理，嵌入式 SQL 接口提供了一个名为sqlca（SQL 通讯区域）的全局变量，它具有下面的结构：

```
struct
```



```

{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;

```

（在一个多线程程序中，每一个线程会自动得到它自己的sqlca副本。这和对于标准 C 全局变量errno的处理相似。）

sqlca覆盖了警告和错误。如果执行一个语句时发生了多个警告或错误，那么sqlca将只包含关于最后一条的信息。

如果在上一个SQL语句中没有产生错误，sqlca.sqlcode将为 0 并且sqlca.sqlstate将为“00000”。如果发生一个警告或错误，则sqlca.sqlcode将为负并且sqlca.sqlstate将不为“00000”。一个正的sqlca.sqlcode表示一种无害的情况，例如上一个查询返回零行。sqlcode和sqlstate是两种不同的错误代码模式，详见下文。

如果上一个 SQL 语句成功，那么sqlca.sqlerrd[1]包含被处理行的 OID（如果可用），并且sqlca.sqlerrd[2]包含被处理或被返回的行数（如果适用于该命令）。

在发生一个错误或警告的情况下，sqlca.sqlerrm.sqlerrmc将包含一个描述该错误的字符串。域sqlca.sqlerrm.sqlerrml包含存储在sqlca.sqlerrm.sqlerrmc中错误消息的长度（strlen()的结果，对于一个 C 程序员来说并不感兴趣）。注意一些消息可能太长不能适应定长的sqlerrmc数组，它们将被截断。

在发生一个警告的情况下，sqlca.sqlwarn[2]被设置为W（在所有其他情况中，它被设置为不同于W的东西）。如果sqlca.sqlwarn[1]被设置为W，那么一个值被存储在一个主变量中时会被截断。如果任意其他元素被设置为指示一个警告，sqlca.sqlwarn[0]会被设置为W。

域sqlcaid、sqlabc、sqlerrp以及sqlerrd的剩余元素还有sqlwarn当前不包含有用的信息。

SQL 标准中没有定义sqlca结构，但是在一些其他的 SQL 数据系统中都有实现。在核心上这些定义都相似，但是如果你想要编写可移植的应用，那么你应该仔细研究不同的实现。

这里有一个整合使用WHENEVER和sqlca的例子，当一个错误发生时打印出sqlca的内容。在安装一个更“用户友好”的错误处理器之前，这可能对调试或开发原型应用有用。

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();
```

```

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n", sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n", sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n",
        sqlca.sqlerrd[0], sqlca.sqlerrd[1], sqlca.sqlerrd[2],

```

```

sqlca.sqlerrd[3], sqlca.sqlerrd[4], sqlca.sqlerrd[5]);
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n", sqlca.sqlwarn[0],
sqlca.sqlwarn[1], sqlca.sqlwarn[2],
                                                                    sqlca.sqlwarn[3],
sqlca.sqlwarn[4], sqlca.sqlwarn[5],
                                                                    sqlca.sqlwarn[6],
sqlca.sqlwarn[7]);
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "=====\n");
}

```

结果看起来像（这里的错误是一个拼写错误的表名）：

```

==== sqlca ====
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
=====

```

### 34.8.3. SQLSTATE 与 SQLCODE

域`sqlca.sqlstate`以及`sqlca.sqlcode`是提供错误代码的两种不同模式。两种都源自于 SQL 标准，但是在标准的 SQL-92 版本中`SQLCODE`已经被标记为弃用并且在后面的版本中被删除。因此，强烈建议新应用使用`SQLSTATE`。

`SQLSTATE`是一个五字符数组。这五个字符包含数字或大写字母，它表示多种错误或警告情况的代码。`SQLSTATE`具有一种层次模式：前两个字符表示情况的总体分类，后三个字符表示总体情况的子类。代码`00000`表示一种成功的状态。SQL 标准中的大部分都有对应的`SQLSTATE`代码。LightDB服务器本地支持`SQLSTATE`错误代码，因此通过在所有应用中自始至终使用这种错误代码模式可以实现高度的一致性。进一步的信息请见Appendix C

被弃用的错误代码模式`SQLCODE`是一个简单的整数。值为 0 表示成功，一个正值表示带附加信息的成功，一个负值表示一个错误。SQL 标准只定义了正值 +100，它表示上一个命令返回或者影响了零行，并且没有特定的负值。因此，这种模式只能实现很可怜的可移植性并且不具有层次性的代码分配。历史上，LightDB的嵌入式 SQL 处理器已经分配了一些特定的`SQLCODE`值供它使用，它们的数字值和符号名称被列在下文。记住这些对其他 SQL 实现不是可移植的。为了简化移植应用到`SQLSTATE`模式，对应的`SQLSTATE`也被列出。不过，在两种模式之间没有一对一或者一对多的映射（事实上是多对多），因此在每一种情况下你都应该参考Appendix C中列出的全局`SQLSTATE`。

这些是已分配的`SQLCODE`值：

0 (ECPG\_NO\_ERROR)

表示没有错误（`SQLSTATE 00000`）。

100 (ECPG\_NOT\_FOUND)

这是一种无害情况，它表示上一个命令检索或者处理了零行，或者你到达了游标的末尾（`SQLSTATE 02000`）。

在一个循环中处理一个游标时，你可以使用这个代码作为一种方法来检测何时中止该循环，像这样：

```
while (1)
{
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}
```

但是WHENEVER NOT FOUND DO BREAK实际上会在内部这样做，因此显式地把它写出来通常没有什么好处。

-12 (ECPG\_OUT\_OF\_MEMORY)

表示你的虚拟内存已被耗尽。数字值被定义为-ENOMEM (SQLSTATE YE001)。

-200 (ECPG\_UNSUPPORTED)

表示预处理器已经产生了一些该库不知道的东西。也许你正在运行一个不兼容版本的预处理和库 (SQLSTATE YE002)。

-201 (ECPG\_TOO\_MANY\_ARGUMENTS)

这表示命令指定了超过该命令预期数量的主变量 (SQLSTATE 07001 或 07002)。

-202 (ECPG\_TOO\_FEW\_ARGUMENTS)

这表示命令指定的主变量数量低于该命令的预期 (SQLSTATE 07001 或 07002)

-203 (ECPG\_TOO\_MANY\_MATCHES)

这意味着一个查询已经返回了多个行，但是该语句只准备存储一个结果行 (例如，因为指定的变量不是数组) (SQLSTATE 21000)。

-204 (ECPG\_INT\_FORMAT)

主变量是类型int而数据库中的数据是一种不同的类型并且含有一个不能被解释为int的值。该库使用strtol()进行这种转换 (SQLSTATE 42804)。

-205 (ECPG\_UINT\_FORMAT)

主变量是类型unsigned int而数据库中的数据是一种不同的类型并且含有一个不能被解释为unsigned int的值。该库使用strtoul()进行这种转换 (SQLSTATE 42804)。

-206 (ECPG\_FLOAT\_FORMAT)

主变量是类型float而数据库中的数据是另一种类型并且含有一个不能被解释为float的值。该库使用strtod()进行这种转换 (SQLSTATE 42804)。

-207 (ECPG\_NUMERIC\_FORMAT)

主变量是类型numeric而数据库中的数据是另一种类型并且含有一个不能被解释为numeric的值 (SQLSTATE 42804)。

-208 (ECPG\_INTERVAL\_FORMAT)

主变量是类型interval而数据库中的数据是另一种类型并且含有一个不能被解释为interval的值 (SQLSTATE 42804)。

-209 (ECPG\_DATE\_FORMAT)

主变量是类型date而数据库中的数据是另一种类型并且含有一个不能被解释为date的值 (SQLSTATE 42804)。

## -210 (ECPG\_TIMESTAMP\_FORMAT)

主变量是类型timestamp而数据库中的数据是另一种类型并且含有一个不能被解释为timestamp的值 (SQLSTATE 42804)。

## -211 (ECPG\_CONVERT\_BOOL)

这表示主变量是类型bool而数据库中的数据既不是't'也不是'f' (SQLSTATE 42804)。

## -212 (ECPG\_EMPTY)

发送给LightDB服务器的语句是空的 (通常在一个嵌入式 SQL 程序中不会发生, 因此它可能指向一个内部错误) (SQLSTATE YE002)。

## -213 (ECPG\_MISSING\_INDICATOR)

返回了一个空值并且没有提供空值指示符 (SQLSTATE 22002)。

## -214 (ECPG\_NO\_ARRAY)

在要求一个数组的地方使用了一个普通变量 (SQLSTATE 42804)。

## -215 (ECPG\_DATA\_NOT\_ARRAY)

在一个要求数组值的地方数据库返回了一个普通变量 (SQLSTATE 42804)。

## -216 (ECPG\_ARRAY\_INSERT)

该值不能被插入到数组 (SQLSTATE 42804)。

## -220 (ECPG\_NO\_CONN)

程序尝试访问一个不存在的连接 (SQLSTATE 08003)。

## -221 (ECPG\_NOT\_CONN)

程序尝试访问一个存在的连接但是它没有打开 (这是一个内部错误) (SQLSTATE YE002)。

## -230 (ECPG\_INVALID\_STMT)

你尝试使用的语句还没有被准备好 (SQLSTATE 26000)。

## -239 (ECPG\_INFORMIX\_DUPLICATE\_KEY)

重复键错误, 违背唯一约束 (Informix 兼容模式) (SQLSTATE 23505)。

## -240 (ECPG\_UNKNOWN\_DESCRIPTOR)

没有找到指定的描述符。你尝试使用的语句还没有被准备好 (SQLSTATE 33000)。

## -241 (ECPG\_INVALID\_DESCRIPTOR\_INDEX)

指定的描述符超出范围 (SQLSTATE 07009)。

## -242 (ECPG\_UNKNOWN\_DESCRIPTOR\_ITEM)

请求了一个非法的描述符 (这是一个内部错误) (SQLSTATE YE002)。

## -243 (ECPG\_VAR\_NOT\_NUMERIC)

在执行一个动态语句期间, 数据库返回了一个numeric值而主变量不是numeric的 (SQLSTATE 07006)。

-244 (ECPG\_VAR\_NOT\_CHAR)

在执行一个动态语句期间，数据库返回了一个非numeric值而主变量是numeric的 (SQLSTATE 07006)。

-284 (ECPG\_INFORMIX\_SUBSELECT\_NOT\_ONE)

子查询的结果不是单一行 (Informix 兼容模式) (SQLSTATE 21000)。

-400 (ECPG\_PGSQL)

LightDB服务器导致了某个错误。该消息包含来自LightDB服务器的错误消息。

-401 (ECPG\_TRANS)

LightDB服务器通知我们不能启动、提交或回滚事务 (SQLSTATE 08007)。

-402 (ECPG\_CONNECT)

到数据库的连接尝试没有成功 (SQLSTATE 08001)。

-403 (ECPG\_DUPLICATE\_KEY)

重复键错误，违背唯一约束 (SQLSTATE 23505)。

-404 (ECPG\_SUBSELECT\_NOT\_ONE)

子查询的结果不是单一行 (SQLSTATE 21000)。

-602 (ECPG\_WARNING\_UNKNOWN\_PORTAL)

指定了一个非法的游标名 (SQLSTATE 34000)。

-603 (ECPG\_WARNING\_IN\_TRANSACTION)

事务正在进行 (SQLSTATE 25001)。

-604 (ECPG\_WARNING\_NO\_TRANSACTION)

没有活动 (正在进行) 的事务 (SQLSTATE 25P01)。

-605 (ECPG\_WARNING\_PORTAL\_EXISTS)

指定了一个现有的游标名 (SQLSTATE 42P03)。

## 34.9. 预处理器指令

一些预处理器指令可以用来改变ecpg预处理器解析和处理一个文件的方式。

### 34.9.1. 包括文件

要包括一个外部文件到你的嵌入式 SQL 程序中，可以用：

```
EXEC SQL INCLUDE filename;  
EXEC SQL INCLUDE <filename>;  
EXEC SQL INCLUDE "filename";
```

嵌入式 SQL 预处理器将查找一个名为filename.h的文件，处理它并且把它包括在结果 C 输出中。这样，被包括文件中的嵌入式 SQL 语句会被正确地处理。

ecpg预处理器将以下列顺序在几个目录中搜索一个文件：

- 当前目录
- /usr/local/include
- LightDB 的包括目录，在编译时定义（例如/usr/local/lightdb-x/include）
- /usr/include

但是当使用EXEC SQL INCLUDE "filename"时，只有当前目录会被搜索。

在每一个目录中，预处理器将首先按给定的文件名搜索，如果没有找到将会追加.h到文件名并且重试（除非指定的文件名已经具有该后缀）。

注意EXEC SQL INCLUDE不同于：

```
#include <filename.h>
```

因为这个文件不服从 SQL 命令预处理。自然地，你可以继续使用 C 的#include指令来包括其他头文件。

### Note

包括文件名是大小写敏感的，即使EXEC SQL INCLUDE命令的剩余部分遵守通常的 SQL 大小写敏感规则。

## 34.9.2. define 和 undef 指令

与 C 中我们熟知的指令#define相似，嵌入式 SQL 具有类似的概念：

```
EXEC SQL DEFINE name;
EXEC SQL DEFINE name value;
```

因此你可以定义一个名称：

```
EXEC SQL DEFINE HAVE_FEATURE;
```

并且你也可以定义常量：

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

使用undef来移除一个之前的定义：

```
EXEC SQL UNDEF MYNUMBER;
```

当然在你的嵌入式 SQL 程序中你可以继续使用 C 版本的#define和#undef。区别在于你定义的值会在哪里被计算。如果你使用EXEC SQL DEFINE，那么ecpg预处理器会计算这些定义并且替换值。例如，如果你写：

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tb1 SET col = MYNUMBER;
```

那么ecpg将已经做过替换并且你的C编译器将永远不会看见名为MYNUMBER的任何名称或标识符。注意你不能把#define用于一个将要在一个嵌入式SQL查询中使用的常量，因为在这种情况下嵌入式SQL预编译器不能看到这个声明。

### 34.9.3. ifdef, ifndef, elif, else, 和 endif 指令

你可以使用下列指定来有条件地编译代码小节：

```
EXEC SQL ifdef name;
```

检查一个name，如果已经用EXEC SQL define name定义了name就处理接下来的行。

```
EXEC SQL ifndef name;
```

检查一个name，如果没有用EXEC SQL define name定义name就处理接下来的行。

```
EXEC SQL elif name;
```

在EXEC SQL ifdef name或者EXEC SQL ifndef name命令之后开始一个可选的备选小节。可以出现任意数量的elif小节。跟在elif后面的行将被处理，如果name已经被定义并且在相同的ifdef/ifndef...endif结构中前面的小节没有被处理。

```
EXEC SQL else;
```

开始一个可选的，最终可以替代的小节在EXEC SQL ifdef name 或 EXEC SQL ifndef name命令之后。后续的行将被处理，如果在相同的ifdef/ifndef...endif结构中前面的小节已经被处理。

```
EXEC SQL endif;
```

结束一个ifdef/ifndef...endif结构。后续的行正常处理。

ifdef/ifndef...endif 结构可以被嵌套，可达127层深。

此示例将完全编译三个SET TIMEZONE命令之一：

```
EXEC SQL ifdef TZVAR;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL endif;
```

## 34.10. 处理嵌入式 SQL 程序

现在你已经对如何构造嵌入式SQL C程序有所了解，你可能希望知道如何编译它们。在编译之前，你需要让该文件通过嵌入式SQL C预处理器，它会把你用到的SQL转换成特殊的函数调用。在编译之后，你必须链接一个包含所需函数的特殊库。这些函数从参数中取得信息、使用libpq执行SQL命令并且把结果放在指定的参数中用来输出。

该预处理器程序被称作ecpg并且被包括在一个正常的LightDB安装中。嵌入式SQL程序通常带有扩展名.pgc。如果你有一个程序文件prog1.pgc，你可以调用下面的命令对它进行预处理：

```
ecpg prog1.pgc
```

这将创建一个文件prog1.c。如果你的输入文件不遵循建议的命名模式，你可以用-o选项显式地指定输出文件。

预处理过的文件可以被正常地编译，例如：

```
cc -c prog1.c
```

产生的 C 源文件从LightDB安装中包括头文件，因此如果你把LightDB安装在一个不被默认搜索的位置，你必须在编译命令行中增加一个选项（例如-I/usr/local/lightdb-x/include）。

要链接一个嵌入式 SQL 程序，你需要包括libecpg库，像这样：

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

再次，你可能不得不在命令行中增加类似-L/usr/local/lightdb-x/lib的选项。

你可以使用lt\_config 或者pkg-config 加上包名libecpg来得到你的安装路径。

如果你使用make来管理一个大工程的构建过程，把下面的隐式规则包括在你的 makefile 中将会很方便：

```
ECPG = ecpg
```

```
%.c: %.pgc
    $(ECPG) $<
```

ecpg命令的完整语法可见ecpg。

ecpg库默认是线程安全的。不过，你可能需要使用一些线程命令行选项来编译你的客户端代码。

## 34.11. 库函数

libecpg库主要包含用于实现嵌入式 SQL 命令所表达功能的“隐藏”函数。但是也有一些可以被直接调用的函数。但是注意这会让你的代码不可移植。

- 如果调用时第一个参数非零，ECPGdebug(int on, FILE \*stream)会打开调试日志。调试日志在流上完成。该日志包含所有插入了输入变量的SQL语句，以及来自于LightDB服务器的结果。在你的SQL语句中查找错误时这会非常有用。

### Note

在 Windows 上，如果ecpg库和应用使用不同标志编译的，这个函数调用将会是应用崩溃，因为FILE指针的内部表达不同。特别地，库和使用库的应用应该使用相同的多线程/单线程、发行/调试以及静态/动态标志。

- ECPGget\_PGconn(const char \*connection\_name) 返回由给定名称标识的库数据库连接句柄。如果connection\_name被设置为NULL，当前连接句柄将被返回。如果无法定位到连接句柄，该函数返回NULL。如果需要，返回的连接句柄可以被用来调用任何其他来自于libpq的函数。

### Note

直接使用libpq例程来操纵ecpg中建立的数据库连接句柄是一种糟糕的做法。



- ECPGtransactionStatus(const char \*connection\_name)返回由connection\_name标识的给定连接的当前事务状态。关于返回的状态代码请参考Section 32.2 和 libpq的 PQtransactionStatus。
- 如果你连接到了一个数据库，ECPGstatus(int lineno, const char\* connection\_name)会返回真；否则返回假。如果使用的是一个单一连接，connection\_name可以为NULL。

## 34.12. 大对象

ECPG 并不直接支持大对象，在调用ECPGget\_PGconn()函数获得所需的PGconn对象后，ECPG应用能通过 libpq 大对象函数操纵大对象（不过，对ECPGget\_PGconn()函数的使用以及直接接触PGconn对象都必须非常小心，并且最好不要与其他 ECPG 数据库访问调用混合在一起）。

更多关于ECPGget\_PGconn()的细节可见Section 34.11 大对象函数接口的相关信息可见Chapter 33

大对象函数必须在一个事务块中被调用，因此当自动提交关闭时，必须显式地发出BEGIN命令。

Example 34. 给出了一个例子程序，它展示了在一个 ECPG 应用中如何创建、写入和读取一个大对象。

### Example 34.2. 访问大对象的 ECPG 程序

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn      *conn;
    Oid         loid;
    int         fd;
    char        buf[256];
    int         buflen = 256;
    char        buf2[256];
    int         rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
    COMMIT;

    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);

    /* 创建 */
    loid = lo_create(conn, 0);
    if (loid < 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));
```

```
printf("loid = %d\n", loid);

/* 写入测试 */
fd = lo_open(conn, loid, INV_READ|INV_WRITE);
if (fd < 0)
    printf("lo_open() failed: %s", PQerrorMessage(conn));

printf("fd = %d\n", fd);

rc = lo_write(conn, fd, buf, buflen);
if (rc < 0)
    printf("lo_write() failed\n");

rc = lo_close(conn, fd);
if (rc < 0)
    printf("lo_close() failed: %s", PQerrorMessage(conn));

/* 读取测试 */
fd = lo_open(conn, loid, INV_READ);
if (fd < 0)
    printf("lo_open() failed: %s", PQerrorMessage(conn));

printf("fd = %d\n", fd);

rc = lo_read(conn, fd, buf2, buflen);
if (rc < 0)
    printf("lo_read() failed\n");

rc = lo_close(conn, fd);
if (rc < 0)
    printf("lo_close() failed: %s", PQerrorMessage(conn));

/* 检查 */
rc = memcmp(buf, buf2, buflen);
printf("memcmp() = %d\n", rc);

/* 清理 */
rc = lo_unlink(conn, loid);
if (rc < 0)
    printf("lo_unlink() failed: %s", PQerrorMessage(conn));

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
return 0;
}
```

## 34.13. C++ 应用

ECPG 对于 C++ 应用提供了有限的支持。这一节描述了一些忠告。

ecpg 预处理器采用一个用 C (或者类似 C 的东西) 和嵌入式 SQL 命令编写的输入文件, 把嵌入式 SQL 命令转换成 C 语言块, 并且最终产生一个 .c 文件。在 C++ 下使用时, 因此它们应该能在 C++ 中无缝地使用。

不过, 通常 ecpg 预处理器只理解 C, 它无法处理 C++ 语言的特殊语法和保留词。因此, 一些写在 C++ 应用代码中的使用了 C++ 特定复杂特性的嵌入式 SQL 代码可能无法被正确地预处理或者无法按预期工作。

使用 C++ 应用中嵌入式 SQL 代码的安全方法是把 ECPG 调用隐藏在一个 C 模块中，C++ 应用代码会调用它来访问数据库，还要把它和剩余的 C++ 代码链接起来。详见Section 34.13.2

### 34.13.1. 主变量的可见范围

ecpg预处理器能理解 C 中变量的可见范围。在 C 语言中，这是相当简单的，因为变量的可见范围是基于它们的代码块的。不过在 C++ 中，引用类成员变量的代码块是不同于定义它的代码块的，因此ecpg预处理器将无法理解类成员变量的可见范围。

例如，在下面的情况中，ecpg预处理器无法为test方法中的变量dbname找到任何声明，因此将发生一个错误。

```
class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
    COMMIT;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~TestCpp()
{
    EXEC SQL DISCONNECT ALL;
}
```

这段代码将导致一个这样的错误：

```
ecpg test_cpp.pgc
test_cpp.pgc:28: ERROR: variable "dbname" is not declared
```

为了避免这种可见性问题，可以修改test方法来把一个本地变量用作中间存储。但是这种方法只是一种比较差的变通方案，因为它让代码变得丑陋并且降低了性能。

```
void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;
```

```

EXEC SQL SELECT current_database() INTO :tmp;
strcpy(dbname, tmp, sizeof(tmp));

printf("current_database = %s\n", dbname);
}

```

### 34.13.2. 使用外部 C 模块的 C++ 应用开发

如果你理解了 C++ 中 `ecpg` 预处理器的这些技术限制，你可能已经知道在链接阶段把 C 对象和 C++ 对象链接起来让 C++ 应用能使用 ECPG 特性比直接在 C++ 代码中写一些嵌入式 SQL 命令要更好。这一节用一个简单的例子描述了一种将嵌入式 SQL 命令从 C++ 应用代码中独立出去的方法。在这个例子中，应用由 C++ 实现，而 C 和 ECPG 被用来连接到 LightDB 服务器。

需要创建三种文件：一个 C 文件 (\*.pgc)、一个头文件和一个 C++ 文件：

`test_mod.pgc`

一个执行嵌入在 C 中的 SQL 命令的子例程模块。它将被预处理器转换成 `test_mod.c`。

```

#include "test_mod.h"
#include <stdio.h>

void
db_connect()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
    COMMIT;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}

```

`test_mod.h`

包含 C 模块 (`test_mod.pgc`) 中函数定义的头文件。它会被 `test_cpp.cpp` 包括。这个文件必须在声明周围有一个 `extern "C"` 块，因为它将被链接到 C++ 模块。

```

#ifdef __cplusplus
extern "C" {
#endif

```

```
void db_connect();
void db_test();
void db_disconnect();
```

```
#ifdef __cplusplus
}
#endif
```

test\_cpp.cpp

应用的主代码，包括main例程以及这个例子中的一个 C++ 类。

```
#include "test_mod.h"
```

```
class TestCpp
{
public:
    TestCpp();
    void test();
    ~TestCpp();
};
```

```
TestCpp::TestCpp()
{
    db_connect();
}
```

```
void
TestCpp::test()
{
    db_test();
}
```

```
TestCpp::~TestCpp()
{
    db_disconnect();
}
```

```
int
main(void)
{
    TestCpp *t = new TestCpp();

    t->test();
    return 0;
}
```

要构建该应用，按以下步骤处理。通过运行ecpg将test\_mod.pgc转换为test\_mod.c，并且用C编译器将test\_mod.c编译成test\_mod.o：

```
ecpg -o test_mod.c test_mod.pgc
cc -c test_mod.c -o test_mod.o
```

接着，用 C++ 编译器把test\_cpp.cpp编译成test\_cpp.o：

```
c++ -c test_cpp.cpp -o test_cpp.o
```

最后，使用 C++ 编译器链接这些对象文件（test\_cpp.o和test\_mod.o）成为一个可执行文件：

```
c++ test_cpp.o test_mod.o -lecpG -o test_cpp
```

## 34.14. 嵌入式 SQL 命令

这一节描述嵌入式 SQL 所有特定的 SQL 命令。SQL 命令中的 SQL 命令也能被用于嵌入式 SQL，如果有例外会特别说明。

## ALLOCATE DESCRIPTOR

ALLOCATE DESCRIPTOR — 分配一个 SQL 描述符区域

### Synopsis

```
ALLOCATE DESCRIPTOR name
```

### 描述

ALLOCATE DESCRIPTOR分配一个新的命名 SQL 描述符区域，它将被用来在 LightDB 服务器和主程序之间交换数据。

以后可以使用DEALLOCATE DESCRIPTOR命令释放描述符区域。

### 参数

name

SQL 描述符的名称，大小写敏感。这可以是一个 SQL 标识符或者一个主变量。

### 例子

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

### 兼容性

SQL 标准中说明了ALLOCATE DESCRIPTOR。

### 参见

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

## CONNECT

CONNECT — 建立一个数据库连接

## Synopsis

```
CONNECT TO connection_target [ AS connection_name ] [ USER connection_user ]
CONNECT TO DEFAULT
CONNECT connection_user
DATABASE connection_target
```

## 描述

CONNECT命令在客户端和 LightDB 服务器之间建立一个连接。

## 参数

connection\_target

connection\_target以下列形式之一指定连接的目标服务器。

```
[ database_name ] [ @host ] [ :port ]
```

通过 TCP/IP 连接

```
unix:postgresql://host [ :port ] / [ database_name ] [ ?connection_option ]
```

通过 Unix 域套接字

```
tcp:postgresql://host [ :port ] / [ database_name ] [ ?connection_option ]
```

通过 TCP/IP 连接

SQL string constant

包含上述形式之一的一个值

host variable

类型char[]或VARCHAR[]的主变量，它包含上述形式之一的一个值

connection\_name

用于该连接的一个可选标识符，这样可以在其他命令中引用它。这可以是一个 SQL 标识符或者一个主变量。

connection\_user

用于数据库连接的用户名。

使用 user\_name/password、user\_name IDENTIFIED BY password或者 user\_name USING password之一，这个参数也能指定用户名和口令。

用户名和口令可以是 SQL 标识符、字符串常量或者主变量。

DEFAULT

按 libpq 的定义使用所有默认连接参数。



## 例子

这里是一些指定连接参数的变体:

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS main USER
connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER
connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser
IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser
IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER
connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser
IDENTIFIED BY "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING
"connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14
USER connectuser;
```

这里是一个展示使用主变量指定连接参数的例子程序:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    char *dbname    = "testdb";    /* 数据库名 */
    char *user      = "testuser";  /* 连接用户名 */
    char *connection = "tcp:postgresql://localhost:5432/testdb";
                                /* 连接字符串 */
    char ver[256];    /* 存储版本字符串的缓冲区 */
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;
```

```
printf("version: %s\n", ver);

EXEC SQL CONNECT TO :connection USER :user;
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
COMMIT;
EXEC SQL SELECT version() INTO :ver;
EXEC SQL DISCONNECT;

printf("version: %s\n", ver);

return 0;
}
```

## 兼容性

SQL 标准中说明了CONNECT，但是连接参数的格式是与实现相关的。

## 参见

DISCONNECT, SET CONNECTION

## DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR — 释放一个 SQL 描述符区域

### Synopsis

```
DEALLOCATE DESCRIPTOR name
```

### 描述

DEALLOCATE DESCRIPTOR释放一个命名的 SQL 描述符区域。

### 参数

name

要被释放的描述符的名称。它是大小写敏感的。这可以是一个 SQL 标识符或者一个主变量。

### 例子

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

### 兼容性

SQL 标准说明了DEALLOCATE DESCRIPTOR。

### 参见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

## DECLARE

DECLARE — 定义一个游标

## Synopsis

```
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH
| WITHOUT } HOLD ] FOR prepared_name
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH
| WITHOUT } HOLD ] FOR query
```

## 描述

DECLARE声明一个游标用来在一个预备语句的结果集上迭代。这个命令与直接的 SQL 命令DECLARE在语义上有一点点区别：后者会执行一个查询并且准备结果集用于检索，而这个嵌入式 SQL 命令仅仅声明一个名称作为“循环变量”用于在一个查询的结果集上迭代，实际的执行在游标被OPEN命令打开时才发生。

## 参数

cursor\_name

一个游标名称，大小写敏感。这可以是一个 SQL 标识符或者一个主变量。

prepared\_name

一个预备查询的名称，可以是一个 SQL 标识符或者一个主变量。

query

一个提供游标要返回的行的SELECT或者VALUES命令。

游标选项的含义请见DECLARE。

## 例子

为一个查询声明一个游标的例子：

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE cur1 CURSOR FOR SELECT version();
```

为一个预备语句声明一个游标的例子：

```
EXEC SQL PREPARE stmt1 AS SELECT version();
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

## 兼容性

SQL 标准中说明了DECLARE。

## 参见

OPEN, CLOSE, DECLARE

## DESCRIBE

DESCRIBE — 得到有关一个预备语句或结果集的信息

## Synopsis

```
DESCRIBE [ OUTPUT ] prepared_name USING [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO sqllda_name
```

## 描述

DESCRIBE检索被一个预备语句所含的结果列的元信息，而不会实际取得一行。

## 参数

prepared\_name

一个预备语句的名称。这可以是一个 SQL 标识符或者一个主变量。

descriptor\_name

一个描述符名称。它是大小写敏感的。它可以是一个 SQL 标识符或者一个主变量。

sqllda\_name

一个 SQLDA 变量的名称。

## 例子

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

## 兼容性

SQL 标准中说明了DESCRIBE。

## 参见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

## DISCONNECT

DISCONNECT — 终止一个数据库连接

### Synopsis

```
DISCONNECT connection_name  
DISCONNECT [ CURRENT ]  
DISCONNECT DEFAULT  
DISCONNECT ALL
```

### 描述

DISCONNECT关闭一个（或者所有）到数据库的连接。

### 参数

connection\_name

一个由CONNECT命令建立的数据库连接名称。

CURRENT

关闭“当前的”连接，它可以是最近打开的连接或者是由SET CONNECTION命令设置的连接。如果没有参数被传给DISCONNECT命令，这将是默认值。

DEFAULT

关闭默认连接。

ALL

关闭所有打开的连接。

### 例子

```
int  
main(void)  
{  
    EXEC SQL CONNECT TO testdb AS DEFAULT USER testuser;  
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;  
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;  
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;  
  
    EXEC SQL DISCONNECT CURRENT; /* 关闭 con3      */  
    EXEC SQL DISCONNECT DEFAULT; /* 关闭 DEFAULT  */  
    EXEC SQL DISCONNECT ALL;     /* 关闭 con2 以及 con1 */  
  
    return 0;  
}
```

### 兼容性

SQL 标准中说明了DISCONNECT。

### 参见

CONNECT, SET CONNECTION

## EXECUTE IMMEDIATE

EXECUTE IMMEDIATE — 动态地准备和执行一个语句

### Synopsis

```
EXECUTE IMMEDIATE string
```

### 描述

EXECUTE IMMEDIATE立刻预备并且执行一个动态指定的 SQL 语句，不检索结果行。

### 参数

string

包含要被执行的 SQL 语句的一个 C 字符串或者是一个主变量。

### 例子

这里是一个用EXECUTE IMMEDIATE和一个名为command的主变量执行INSERT语句的例子：

```
printf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: 'r1'',  
1, 'f')");  
EXEC SQL EXECUTE IMMEDIATE :command;
```

### 兼容性

SQL 标准中说明了EXECUTE IMMEDIATE。

## GET DESCRIPTOR

GET DESCRIPTOR — 从一个 SQL 描述符区域得到信息

### Synopsis

```
GET DESCRIPTOR descriptor_name :cvariable = descriptor_header_item [, ... ]
GET DESCRIPTOR descriptor_name VALUE column_number :cvariable = descriptor_item
[, ... ]
```

### 描述

GET DESCRIPTOR从一个 SQL 描述符区域检索关于一个查询结果集的信息并且把它存储在主变量中。在使用这个命令把信息传输到主语言变量之前，一个描述符区域通常是用FETCH或SELECT填充的。

这个命令有两种形式：第一种形式检索描述符的“头部”项，它适用于全面地查看结果集。一种例子是行计数。第二种形式要求列号作为附加参数，它检索有关一个特定列的信息。其例子是查看列名和实际列值。

### 参数

descriptor\_name

一个描述符名称。

descriptor\_header\_item

一个标识要检索哪一个头部信息项的记号。当前只支持用于得到结果集中列数的COUNT。

column\_number

要检索其信息的列号。计数从 1 开始。

descriptor\_item

一个标识要检索哪一个有关一列信息的项的记号。被支持的项可见Section 34.7.1

cvariable

接收从描述符区域检索到的数据的主变量。

### 例子

检索一个结果集中列数的例子：

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

检索第一列中数据长度的例子：

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length =
RETURNED_OCTET_LENGTH;
```

把第二列的数据体检索成一个字符串的例子：



```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

这里是执行SELECT current\_database();并且显示列数、列数据长度和列数据的完整过程的例子:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int d_count;
    char d_data[1024];
    int d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;

    /* 描述、打开一个游标，并且分配一个描述符给该游标 */
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

    /* 得到全部列的数量 */
    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count          = %d\n", d_count);

    /* 得到一个返回列的长度 */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length =
RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n", d_returned_octet_length);

    /* 将返回的列取出成一个字符串 */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
    printf("d_data          = %s\n", d_data);

    /* 关闭 */
    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

当该例子被执行时，结果看起来是:

```
d_count          = 1
d_returned_octet_length = 6
d_data          = testdb
```

## 兼容性

SQL 标准中说明了GET DESCRIPTOR。

## 参见

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

## OPEN

OPEN — 打开一个动态游标

## Synopsis

```
OPEN cursor_name  
OPEN cursor_name USING value [, ... ]  
OPEN cursor_name USING SQL DESCRIPTOR descriptor_name
```

## 描述

OPEN打开一个游标并且可选地绑定实际值到游标声明中的占位符。该游标必须之前用DECLARE命令声明。OPEN的执行会导致查询开始在服务器上执行。

## 参数

cursor\_name

要被打开的游标的名称。这可以是一个 SQL 标识符或者一个主变量。

value

要被绑定到游标中一个占位符的值。这可以是一个 SQL 常量、一个主变量或者一个带有指示符的主变量。

descriptor\_name

包含要绑定到游标中占位符的值的描述符的名称。这可以是一个 SQL 标识符或者一个主变量。

## 例子

```
EXEC SQL OPEN a;  
EXEC SQL OPEN d USING 1, 'test';  
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;  
EXEC SQL OPEN :curname1;
```

## 兼容性

SQL 标准中说明了OPEN。

## 参见

DECLARE, CLOSE

## PREPARE

PREPARE — 准备一个语句用于执行

## Synopsis

```
PREPARE name FROM string
```

## 描述

PREPARE将一个作为字符串动态指定的语句准备好执行。这不同于直接的 SQL 语句PREPARE（也可以用于嵌入式程序）。EXECUTE命令被用来执行两种类型的预备语句。

## 参数

prepared\_name

预备查询的一个标识符。

string

包含一个可预备语句的一个 C 字符串或一个主变量，可预备语句是 SELECT、INSERT、UPDATE 或者 DELETE 之一。

## 例子

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";
```

```
EXEC SQL ALLOCATE DESCRIPTOR outdesc;
```

```
EXEC SQL PREPARE foo FROM :stmt;
```

```
EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

## 兼容性

SQL 标准中说明了PREPARE。

## 参见

EXECUTE

## SET AUTOCOMMIT

SET AUTOCOMMIT — 设置当前会话的自动提交行为

### Synopsis

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

### 描述

SET AUTOCOMMIT设置当前数据库会话的自动提交行为。默认情况下，嵌入式 SQL 程序不在自动提交模式中，因此需要显式地发出COMMIT。这个命令可以把会话改成自动提交模式，这样每一个单独的语句都会被隐式提交。

### 兼容性

SET AUTOCOMMIT是 LightDB ECPG 的扩展。

## SET CONNECTION

SET CONNECTION — 选择一个数据库连接

### Synopsis

```
SET CONNECTION [ TO | = ] connection_name
```

### 描述

SET CONNECTION设置“当前的”数据库连接，除非被覆盖，所有命令都会使用这个连接。

### 参数

`connection_name`

一个由CONNECT命令建立的数据库连接名。

DEFAULT

设置该连接为默认连接。

### 例子

```
EXEC SQL SET CONNECTION TO con2;  
EXEC SQL SET CONNECTION = con1;
```

### 兼容性

SQL 标准中说明了SET CONNECTION。

### 参见

CONNECT, DISCONNECT

## SET DESCRIPTOR

SET DESCRIPTOR — 在一个 SQL 描述符区域中设置信息

### Synopsis

```
SET DESCRIPTOR descriptor_name descriptor_header_item = value [, ... ]
SET DESCRIPTOR descriptor_name VALUE number descriptor_item = value [, ...]
```

### 描述

SET DESCRIPTOR用值填充一个 SQL 描述符区域。然后该描述符区域通常会被用来在一个预备查询执行中绑定参数。

这个命令由两种形式：第一种形式适用于描述符“头部”，它独立于特定的数据。第二种形式为由数字标识的特定数据赋值。

### 参数

descriptor\_name

一个描述符名称。

descriptor\_header\_item

一个标识要设置哪个头部信息项的记号。当前只有设置描述符项数量的COUNT被支持。

number

要设置的描述符项的编号。计数从 1 开始。

descriptor\_item

一个标识在描述符中要设置哪个信息项的记号。受支持的项的列表可见Section 34.7.1

value

一个要存储在描述符项中的值。这可以是一个 SQL 标识符或者一个主变量。

### 例子

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

### 兼容性

SQL 标准中说明了SET DESCRIPTOR。

### 参见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

## TYPE

TYPE — 定义一种新数据类型

## Synopsis

```
TYPE type_name IS ctype
```

## 描述

TYPE命令定义一个新的 C 类型。它等效于把一个typedef放在声明节中。

只有使用选项-c运行ecpg时才能识别这个命令。

## 参数

type\_name

新类型的名称。这必须是一个合法的 C 类型名。

ctype

一个 C 类型说明。

## 例子

```
EXEC SQL TYPE customer IS
  struct
  {
    varchar name[50];
    int     phone;
  };
```

```
EXEC SQL TYPE cust_ind IS
  struct ind
  {
    short  name_ind;
    short  phone_ind;
  };
```

```
EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

这里是一个使用EXEC SQL TYPE的例子程序：

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;
```

```
EXEC SQL TYPE tt IS
  struct
  {
    varchar v[256];
    int     i;
```



```
};

EXEC SQL TYPE tt_ind IS
  struct ind {
    short   v_ind;
    short   i_ind;
  };

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
  tt t;
  tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT TO testdb AS con1;
  EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
  COMMIT;

  EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

  printf("t.v = %s\n", t.v.arr);
  printf("t.i = %d\n", t.i);

  printf("t_ind.v_ind = %d\n", t_ind.v_ind);
  printf("t_ind.i_ind = %d\n", t_ind.i_ind);

  EXEC SQL DISCONNECT con1;

  return 0;
}
```

这个程序的输出看起来像：

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

## 兼容性

TYPE命令是一种 LightDB 扩展。

## VAR

VAR — 定义一个变量

## Synopsis

```
VAR varname IS ctype
```

## 描述

VAR命令分配一个新的 C 数据类型给一个主变量。主变量必须之前在一个声明节中声明过。

## 参数

varname

一个 C 变量名。

ctype

一个 C 类型说明。

## 例子

```
Exec sql begin declare section;  
short a;  
exec sql end declare section;  
EXEC SQL VAR a IS int;
```

## 兼容性

VAR命令是一个 LightDB 扩展。

## WHENEVER

WHENEVER — 指定一个要在一个 SQL 语句导致发生一个特定类别的情况时要采取的动作

## Synopsis

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action
```

## 描述

定义一个行为，它会在 SQL 执行结果的特殊情况（行未找到、SQL 警告或错误）中被调用。

## 参数

参数描述见Section 34.8.1

## 例子

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER NOT FOUND DO CONTINUE;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

一个典型的应用是使用WHENEVER NOT FOUND BREAK来处理通过结果集的循环：

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
    COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;

    /* 当到达结果集末尾时，跳出循环 */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;
```

```

EXEC SQL DEALLOCATE DESCRIPTOR d;
EXEC SQL DISCONNECT ALL;

return 0;
}

```

## 兼容性

SQL 标准中说明了WHENEVER，但是大部分动作是 LightDB 扩展。

## 34.15. Informix兼容模式

ecpg可以运行在一种所谓的Informix 兼容模式中。如果这种模式被激活，它的行为就好像它是一个用于Informix E/SQL 的Informix预编译器。一般而言，这将允许你使用美元符号替代EXEC SQL来引入嵌入式 SQL 命令：

```

$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;

```

### Note

在\$之间不能有任何空白以及下列之一的预处理器指令：include、define、ifdef等。否则，预处理器将把记号解析成一个主变量。

有两种兼容性模式：INFORMIX、INFORMIX\_SE

在链接使用这种兼容性模式的程序时，要记得链接上和 ECPG 一起发布的libcompat。

除了之前解释过的语法糖，Informix兼容性模式从 E/SQL 中移植了一些用于输入、输出和数据转换的函数以及嵌入式 SQL 语句到 ECPG 中。

Informix兼容性模式与 ECPG 的 pgtypeslib 库紧密连接。pgtypeslib 把 SQL 数据类型映射到 C 主程序中的数据类型并且大部分Informix兼容性模式的附加函数允许我们在那些 C 主程序类型上操作。不过注意兼容性的范围被有所限制。它并不是想尝试复制Informix的行为。它允许你做或多或少的相同操作并且给你具有相同名称和相同基本行为的函数，但是此刻如果你使用Informix，其中并没有唾手可得的替代品。此外，一些数据类型也不同。例如，LightDB的日期时间和区间类型不理解范围（例如YEAR TO MINUTE），因此你也无法在 ECPG 中找到支持。

### 34.15.1. 附加类型

用于存储右切边字符串数据的 Informix-特殊的“string”伪类型现在在 Informix 模式中不用typedef就能支持。事实上，在 Informix 模式中，ECPG 拒绝处理包含typedef sometype string;的源文件。

```

EXEC SQL BEGIN DECLARE SECTION;
string userid; /* 这个变量将包含切边过的数据 */
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;

```

## 34.15.2. 附加的/缺少的 嵌入式 SQL 语句

CLOSE DATABASE

这个语句关闭当前连接。事实上，这是 ECPG 的 DISCONNECT CURRENT 语句的同义词：

```
$CLOSE DATABASE;           /* 关闭当前连接 */
EXEC SQL CLOSE DATABASE;
```

FREE cursor\_name

由于 ECPG 和 Informix ESQL/C 在工作方式上的区别（一个是纯语法转换而另一个依赖于底层的运行时库），在 ECPG 中没有 FREE cursor\_name 语句。这是因为在 ECPG 中，DECLARE CURSOR 不会翻译成一个运行时库中使用游标名的函数调用。这意味着在 ECPG 运行时库中不会有 SQL 游标的运行时登记，SQL 游标只登记在 LightDB 服务器中。

FREE statement\_name

FREE statement\_name 是 DEALLOCATE PREPARE statement\_name 的同义词。

## 34.15.3. Informix-兼容的 SQLDA 描述符区域

Informix-兼容模式支持一种与 Section 34.7.2 中所述不同的结构。如下：

```
struct sqlvar_compat
{
    short  sqltype;
    int    sqllen;
    char   *sqldata;
    short  *sqlind;
    char   *sqlname;
    char   *sqlformat;
    short  sqlitype;
    short  sqlilen;
    char   *sqlidata;
    int    sqlxid;
    char   *sqltypename;
    short  sqltypelen;
    short  sqlownerlen;
    short  sqlsourcetype;
    char   *sqlownername;
    int    sqlsourceid;
    char   *sqlilongdata;
    int    sqlflags;
    void   *sqlreserved;
};

struct sqlda_compat
{
    short  sqld;
    struct sqlvar_compat *sqlvar;
    char   desc_name[19];
    short  desc_occ;
    struct sqlda_compat *desc_next;
    void   *reserved;
};
```

```
};
```

```
typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqllda_compat    sqllda_t;
```

全局属性是:

sqllda

SQLDA描述符中域的数量。

sqlvar

每一个域属性的指针。

desc\_name

未使用，用零字节填充。

desc\_occ

已分配结构的尺寸。

desc\_next

如果结果集包含多于一个记录，这个域是下一个 SQLDA 结构的指针。

reserved

未使用的指针，包含 NULL。为 Informix-兼容性而保留。

对每一个域的属性如下，它们被存储在sqlvar数组中:

sqltype

域的类型。可以使用的常量定义在sqltypes.h中。

sqlllen

域数据的长度。

sqldata

域数据的指针。该指针是char \*类型，它所指向的数据是二进制个事。例子:

```
int intval;
```

```
switch (sqldata->sqlvar[i].sqltype)
{
    case SQLINTEGER:
        intval = *(int *)sqldata->sqlvar[i].sqldata;
        break;
    ...
}
```

sqlind

NULL 指示符的指针。如果是由 DESCRIBE 或 FETCH 返回，那么它总是一个有效的指针。如果被用作EXECUTE ... USING sqllda;的输入，那么 NULL-指针值意味着这个域的值是非-NULL 的。否则必须正确地设置一个有效的指针和sqlitype。例子:

```
if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
    printf("value is NULL\n");
```

sqlname

域的名称。以 0 终止的字符串。

sqlformat

在 Informix 中保留，是该域的PQfformat的值。

sqlitype

NULL 指示符数据的类型。当从服务器返回数据时，它总是 SQLSMINT。当SQLDA被用于一个参数化查询时，数据要根据设置的类型对待。

sqlilen

NULL 指示符数据的长度。

sqlxid

该域的扩展类型，PQftype的结果。

sqltypename

sqltypelen

sqlownerlen

sqlsourcetype

sqlownername

sqlsourceid

sqlflags

sqlreserved

未使用。

sqlilongdata

如果sqlilen大于 32kB，它等于sqldata。

例子:

```
EXEC SQL INCLUDE sqlda.h;
```

```
sqlda_t      *sqlda; /* 这不需要在嵌入式 DECLARE SECTION 下 */
```

```
EXEC SQL BEGIN DECLARE SECTION;
char *prep_stmt = "select * from table1";
int i;
EXEC SQL END DECLARE SECTION;
```

...

```
EXEC SQL PREPARE mystmt FROM :prep_stmt;
```

```
EXEC SQL DESCRIBE mystmt INTO sqlda;
```

```
printf("# of fields: %d\n", sqlda->sqld);
for (i = 0; i < sqlda->sqld; i++)
    printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);
```

```

EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
EXEC SQL OPEN mycursor;
EXEC SQL WHENEVER NOT FOUND GOTO out;

while (1)
{
    EXEC SQL FETCH mycursor USING sqllda;
}

EXEC SQL CLOSE mycursor;

free(sqllda); /* 主结构完全被 free(), sqllda 和 sqllda->sqlvar 在一个已分配区域
中 */

```

更多信息可见sqllda.h头部和src/interfaces/ecpg/test/compat\_informix/sqllda.pgc回归测试。

## 34.15.4. 附加函数

decadd

将两个decimal类型值相加。

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

该函数接收第一个类型为 decimal 的操作数的指针 (arg1)、第二个类型为 decimal 的操作数的指针 (arg2) 以及将包含和的 decimal 值的指针 (sum)。成功时该函数返回 0。溢出时返回ECPG\_INFORMIX\_NUM\_OVERFLOW, 下溢时返回ECPG\_INFORMIX\_NUM\_UNDERFLOW。其他失败会返回 -1 并且errno会被设置为相应的pgtypeslib 中的errno编号。

deccmp

比较两个 decimal 变量。

```
int deccmp(decimal *arg1, decimal *arg2);
```

该函数接收第一个 decimal 值的指针 (arg1)、第二个 decimal 值的指针 (arg2) 并且返回一个整数值说明哪一个值更大。

- 1, 如果arg1指向的值大于var2指向的值
- -1, 如果arg1指向的值小于var2指向的值
- 0, 如果arg1指向的值与arg2指向的值相等

deccopy

拷贝一个 decimal 值。

```
void deccopy(decimal *src, decimal *target);
```

该函数接收要拷贝的 decimal 值的指针作为第一个参数 (src) 以及一个类型为 decimal 的目标结构的指针作为第二个参数 (target)。

deccvasc

把一个值从 ASCII 表达转换成一个 decimal 类型。



```
int deccvasc(char *cp, int len, decimal *np);
```

该函数接收一个包含要转换的数字的字符串表达的字符串指针 (cp) 及其长度 (len)。np是一个用来保存操作结果的 decimal 值的指针。

例如, 可用的格式有: -2、.794、+3.44、592.49E07或者 -32.84e-4。

成功时该函数返回 0。如果发生溢出或者下溢, 分别返回 ECPG\_INFORMIX\_NUM\_OVERFLOW或者 ECPG\_INFORMIX\_NUM\_UNDERFLOW。如果 ASCII 表达无法被解析, 将返回ECPG\_INFORMIX\_BAD\_NUMERIC。如果解析指数时发生问题则返回ECPG\_INFORMIX\_BAD\_EXPONENT。

deccvdbl

将一个 double 值转换成一个 decimal 值。

```
int deccvdbl(double dbl, decimal *np);
```

该函数接收要被转换的 double 变量作为第一个参数 (dbl)。该函数接收一个 decimal 变量的指针作为第二个参数 (np), 它被用来保存操作的结果。

该函数在成功时返回 0, 在转换失败时返回一个负值。

deccvint

将一个 int 值转换成 decimal 值。

```
int deccvint(int in, decimal *np);
```

该函数接收要被转换的 int 变量作为第一个参数 (in)。该函数接收一个 decimal 变量的指针作为第二个参数 (np), 它被用来保存操作的结果。

该函数在成功时返回 0, 在转换失败时返回一个负值。

deccvlong

将一个 long 值转换成 decimal 值。

```
int deccvlong(long lng, decimal *np);
```

该函数接收要被转换的 long 变量作为第一个参数 (lng)。该函数接收一个 decimal 变量的指针作为第二个参数 (np), 它被用来保存操作的结果。

该函数在成功时返回 0, 在转换失败时返回一个负值。

decdiv

用两个 decimal 类型的变量做除法。

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

该函数接收两个变量的指针作为第一个 (n1) 和第二个 (n2) 操作数并且结算n1/n2。result是一个指向保存操作结果的变量的指针。

成功时返回 0, 如果除法失败则返回一个负值。如果发生溢出或下溢, 该函数分别返回 ECPG\_INFORMIX\_NUM\_OVERFLOW或者 ECPG\_INFORMIX\_NUM\_UNDERFLOW。如果发现一次除零尝试, 该函数返回 ECPG\_INFORMIX\_DIVIDE\_ZERO。

## decmul

将两个 decimal 值相乘。

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

该函数接收两个变量的指针作为第一个 (n1) 和第二个 (n2) 操作数并且结算 $n1*n2$ 。result是一个指向保存操作结果的变量的指针。

成功时返回 0, 如果乘法失败则返回一个负值。如果发生溢出或下溢, 该函数分别返回 ECPG\_INFORMIX\_NUM\_OVERFLOW或者 ECPG\_INFORMIX\_NUM\_UNDERFLOW。

## decsub

从一个 decimal 值中减去另一个。

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

该函数接收两个变量的指针作为第一个 (n1) 和第二个 (n2) 操作数并且结算 $n1-n2$ 。result是一个指向保存操作结果的变量的指针。

成功时返回 0, 如果减法失败则返回一个负值。如果发生溢出或下溢, 该函数分别返回 ECPG\_INFORMIX\_NUM\_OVERFLOW或者 ECPG\_INFORMIX\_NUM\_UNDERFLOW。

## dectoasc

将一个 decimal 变量转换成它的 ASCII 表达放在一个 C char\* 字符串中。

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

该函数接收一个要被转换成文本表达的 decimal 类型变量的指针 (np)。cp是应保存操作结果的缓冲区。参数right指定小数点右边应该有多少位保留在输出中。结果将被圆整到所指定数量的十进制位。将right设置为 -1 表示输出中应该包括所有可用的十进制位。如果输出缓冲区的长度 (由len指定) 不足以保存包含拖尾零字节的文本表达, 结果中将只保存一个单一的\*字符并且返回 -1。

如果缓冲区cp太小该函数返回 -1; 如果内存耗尽, 则返回 ECPG\_INFORMIX\_OUT\_OF\_MEMORY。

## dectodbl

将一个 decimal 类型变量转换成一个 double 类型变量。

```
int dectodbl(decimal *np, double *dbl);
```

该函数接收一个要转换的 decimal 值的指针 (np) 以及一个保存操作结果的 double 变量的指针 (dbl)。

该函数在成功时返回 0, 在转换失败时返回一个负值。

## dectoint

将一个 decimal 类型变量转换成一个整数类型变量。

```
int dectoint(decimal *np, int *ip);
```

该函数接收一个要转换的 decimal 值的指针 (np) 以及一个保存操作结果的整数变量的指针 (ip)。

该函数在成功时返回 0，在转换失败时返回一个负值。如果发生溢出，会返回 ECPG\_INFORMIX\_NUM\_OVERFLOW。

注意 ECPG 实现与 Informix 实现不同。Informix 限制一个整数的范围是从 -32767 到 32767，而 ECPG 实现中的限制取决于架构 (-INT\_MAX .. INT\_MAX)。

#### dectolong

将一个 decimal 类型变量转换成一个长整型变量。

```
int dectolong(decimal *np, long *lngp);
```

该函数接收一个要转换的 decimal 值的指针 (np) 以及一个保存操作结果的长整型变量的指针 (lngp)。

该函数在成功时返回 0，在转换失败时返回一个负值。如果发生溢出，会返回 ECPG\_INFORMIX\_NUM\_OVERFLOW。

注意 ECPG 实现与 Informix 实现不同。Informix 限制一个整数的范围是从 -2,147,483,647 到 2,147,483,647，而 ECPG 实现中的限制取决于架构 (-LONG\_MAX .. LONG\_MAX)。

#### rdatestr

将一个日期转换成一个 C char\* 字符串。

```
int rdatestr(date d, char *str);
```

该函数接收两个参数，第一个是要转换的日期 (d)，第二个是目标字符串的指针。输出格式总是 yyyy-mm-dd，因此你需要为该字符串分配至少 11 个字节（包括零字节终止符）。

成功时该函数返回 0，如果发生错误则返回一个负值。

注意 ECPG 实现与 Informix 实现不同。在 Informix 中，该格式可能受到环境变量设置的影响。而在 ECPG 中，你不能改变输出格式。

#### rstrdate

解析一个日期的文本表达。

```
int rstrdate(char *str, date *d);
```

该函数接收要转换的日期的文本表达 (str) 以及一个日期类型变量的指针 (d)。这个函数不允许你指定一个格式掩码。它使用 Informix 的默认格式掩码 mm/dd/yyyy。在内部，这个函数用 rdefmtdate 的方式实现。因此，rstrdate 不会更快，并且如果可以选择，你应该选用允许你显式指定格式掩码的 rdefmtdate。

该返回与 rdefmtdate 相同的值。

#### rtoday

得到当前日期。

```
void rtoday(date *d);
```

该函数接收一个日期变量的指针 (d)，它会把该变量设置为当前日期。

在内部这个函数使用 PGTYPE\_SDATE\_today 函数。

### rjulmdy

从一个日期类型变量中抽取日、月、年的值。

```
int rjulmdy(date d, short mdy[3]);
```

该函数接收日期d和由 3 个短整型值构成的数组的指针mdy。该变量名指定了顺序：mdy[0]将被设置为包含月的编号，mdy[1]将被设置为日的值，而mdy[2]将包含年。

当前该函数总是返回 0。

在内部该函数使用PGTYPESdate\_julmdy函数。

### rdefmtdate

使用一个格式掩码把一个字符串转换成一个日期类型的值。

```
int rdefmtdate(date *d, char *fmt, char *str);
```

该函数接收一个用于保存操作结果的日期值的指针（d）、要用来解析日期的格式掩码（fmt）以及包含日期文本表达的 C char\* 字符串（str）。该文本表达应该匹配格式掩码。不过，你不需要具有从该字符串到格式掩码的一一映射。该函数将分析顺序并且寻找表示年的位置的数字yy或yyyy、表示月的位置的mm以及表示日的位置的dd。

该函数返回下列值：

- 0 - 该函数成功终止。
- ECPG\_INFORMIX\_ENOSHORTDATE - 该日期不包含日、月、年之间的定界符。在这种情况下，输入字符串必须是正好 6 个或 8 个字节，但实际上却不是。
- ECPG\_INFORMIX\_ENOTDMY - 格式字符串没有正确地指示年、月、日的顺序。
- ECPG\_INFORMIX\_BAD\_DAY - 输入字符串不含一个合法的日。
- ECPG\_INFORMIX\_BAD\_MONTH - 输入字符串不含一个合法的月。
- ECPG\_INFORMIX\_BAD\_YEAR - 输入字符串不含一个合法的年。

在内部这个函数被实现为使用PGTYPESdate\_defmt\_asc函数。示例输入表可以在那里找到。

### rfmtdate

使用一个格式掩码将一个日期类型变量转换成它的文本表达。

```
int rfmtdate(date d, char *fmt, char *str);
```

该函数接收要转换的日期（d）、格式掩码（fmt）以及将保存日期的文本表达的字符串（str）。

成功时该函数返回 0，如果发生错误则返回一个负值。

在内部这个函数使用PGTYPESdate\_fmt\_asc函数，例子请参考该函数。

### rmdyjul

从由 3 个短整型组成的数组创建一个日期值，它指定了该日期的日、月、年。

```
int rmdyjul(short mdy[3], date *d);
```

该函数接收一个由 3 个短整型构成的数组 (mdy) 以及一个用来保存操作结构的日期类型变量的指针。

当前该函数总是返回 0。

在内部这个函数被实现为使用PGTYPEdate\_mdyjul。

rdayofweek

为一个日期值返回一个表示它是星期几的数字。

```
int rdayofweek(date d);
```

该函数接收日期变量d作为它的唯一参数并且返回一个整数指示这一天是星期几。

- 0 - 周日
- 1 - 周一
- 2 - 周二
- 3 - 周三
- 4 - 周四
- 5 - 周五
- 6 - 周六

在内部这个函数被实现为使用函数PGTYPEdate\_dayofweek。

dtcurrent

检索当前的时间戳。

```
void dtcurrent(timestamp *ts);
```

该函数检索当前时间戳并且把它保存在ts指向的时间戳变量中。

dtcvasc

把一个时间戳从它的文本表达解析到一个时间戳变量中。

```
int dtcvasc(char *str, timestamp *ts);
```

该函数接收要解析的字符串 (str) 以及一个指向保存操作结果的时间戳变量的指针 (ts)。

成功时该函数返回 0, 如果发生错误则返回一个负值。

在内部这个函数使用PGTYPEtimestamp\_from\_asc函数。一个输入示例的表格可以参考该函数的文档。

dtcvfmtasc

使用一个格式掩码把一个时间戳从它的文本表达解析到一个时间戳变量中。

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

该函数接收要解析的字符串（inbuf）、要使用的格式掩码（fmtstr）以及一个指向保存操作结果的时间戳变量的指针（dtvalue）。

这个函数通过PGTYPEStimestamp\_defmt\_asc函数实现。可以使用的格式说明符的列表可以参考该函数的文档。

成功时该函数返回 0，如果发生错误则返回一个负值。

#### dtsub

从一个时间戳中减去另一个并且返回一个区间类型变量。

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

该函数将从ts1指向的时间戳变量中减去ts2指向的时间戳变量，并且将把结果存储在iv指向的区间变量中。

成功时该函数返回 0，如果发生错误则返回一个负值。

#### dttoasc

将一个时间戳变量转换成一个 C char\* 字符串。

```
int dttoasc(timestamp *ts, char *output);
```

该函数接收一个要转换的时间戳变量的指针（ts）以及用于保存操作结果的字符串（output）。它根据 SQL 标准把ts转换成它的文本表达，形式为YYYY-MM-DD HH:MM:SS。

成功时该函数返回 0，如果发生错误则返回一个负值。

#### dttofmtasc

使用一个格式掩码将一个时间戳变量转换成一个 C char\*。

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

该函数接收一个要转换的时间戳的指针（ts）、一个输出缓冲区的指针（output）、已经为输出缓冲区分配的最大长度（str\_len）以及用于转换的格式掩码（fmtstr）。

成功时该函数返回 0，如果发生错误则返回一个负值。

在内部，这个函数使用PGTYPEStimestamp\_fmt\_asc函数。可以使用的格式说明符的列表可以参考该函数的文档。

#### intoasc

将一个区间变量转换成一个 C char\* 字符串。

```
int intoasc(interval *i, char *str);
```

该函数接收一个要转换的区间变量的指针（i）以及要保持该操作结果的字符串（str）。它根据 SQL 标准把i转换成它的文本表达，形式为YYYY-MM-DD HH:MM:SS。

成功时该函数返回 0，如果发生错误则返回一个负值。

#### rfmtlong

用一个格式掩码将一个长整型值转换成它的文本表达。

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

该函数接收长整型值`lng_val`、格式掩码`fmt`以及输出缓冲区的指针`outbuf`。它根据格式掩码将长整型值转换成文本表达。

格式掩码可以由下列格式说明字符构成：

- \*（星） - 如果这个位置可以为空白，否则用一个星号填充。
- &（花号） - 如果这个位置可以为空白，否则用一个零填充。
- # - 把前导零转变成空白。
- < - 左对齐字符串中的数字。
- ,（逗号） - 将有四个或者更多数位的数字份组成用逗号分隔的 3 数位组。
- .（点） - 这个字符分隔数字的整数部分和小数部分。
- -（负） - 如果该数字是一个负值则负号会出现。
- +（加） - 如果该数字是一个正值则加号会出现。
- (- - 这会替换负数前面的负号。负号将不会出现。
- ) - 这个字符替换负号并且被打印在负值的后面。
- \$ - 货币符号。

`rupshift`

把一个字符串转换成大写形式。

```
void rupshift(char *str);
```

该函数接收一个字符串的指针并且把每一个小写形式的字符变成大写形式。

`byleng`

返回一个字符串的字符数，其中不含拖尾的空白。

```
int byleng(char *str, int len);
```

该函数期待一个定长字符串作为它的第一个参数（`str`）并且把它的长度作为第二个参数（`len`）。该函数会返回有效字符的数量，也就是字符串不含拖尾空白的长度。

`ldchar`

复制一个定长字符串到一个空终止的字符串。

```
void ldchar(char *src, int len, char *dest);
```

该函数接收要被复制的定长字符串（`src`）、它的长度（`len`）以及目标内存的指针（`dest`）。注意你需要为`dest`指向的字符串保留至少`len+1`个字节。该函数复制至多`len`个字节到新的位置（如果源字符串有拖尾的空格）并且增加空终止符。

`rgetmsg`

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

这个函数存在，但是目前还没有实现！

rtpalign

```
int rtpalign(int offset, int type);
```

这个函数存在，但是目前还没有实现！

rtpmsize

```
int rtpmsize(int type, int len);
```

这个函数存在，但是目前还没有实现！

rtpwidth

```
int rtpwidth(int sqltype, int sqllen);
```

这个函数存在，但是目前还没有实现！

rsetnull

设置一个变量为 NULL。

```
int rsetnull(int t, char *ptr);
```

该函数接收一个表示变量类型的整数以及一个被造型成 C char\* 指针的变量本身的指针。

存在下列类型：

- CCHARTYPE - 用于类型char或者char\*的一个变量
- CSHORTTYPE - 用于类型short int的一个变量
- CINTTYPE - 用于类型int的一个变量
- CBOOLTYPE - 用于类型boolean的一个变量
- CFLOATTYPE - 用于类型float的一个变量
- CLONGTYPE - 用于类型long的一个变量
- CDOUBLETYPE - 用于类型double的一个变量
- CDECIMALTYPE - 用于类型decimal的一个变量
- CDATETYPE - 用于类型date的一个变量
- CDTIMETYPE - 用于类型timestamp的一个变量

这里是一个调用这个函数的例子：

```
$char c[] = "abc";  
$short s = 17;  
$int i = -74874;
```



```
rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

risnull

测试一个变量是否为 NULL。

```
int risnull(int t, char *ptr);
```

该函数接收要测试的变量的类型 (t) 以及一个指向该变量的指针 (ptr)。注意后者需要被造型为一个 char\*。可能的变量类型请见函数rsetnull。

这里是一个如何使用这个函数的例子：

```
$char c[] = "abc";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

### 34.15.5. 额外的常量

注意这里所有的常量都描述错误并且它们都被定义为表示负值。在每一种不同常量的描述中，你还可以找到在当前实现中该常量表示的值。不过你不应该依赖于这个数字。但是你可以相信所有的这些常量都是被定义为表示负值。

ECPG\_INFORMIX\_NUM\_OVERFLOW

如果在一次计算中发生了溢出，函数会返回这个值。在内部它被定义为 -1200 (Informix定义)。

ECPG\_INFORMIX\_NUM\_UNDERFLOW

如果在一次计算中发生了下溢，函数会返回这个值。在内部它被定义为 -1201 (Informix定义)。

ECPG\_INFORMIX\_DIVIDE\_ZERO

如果发现尝试除零，函数会返回这个值。在内部它被定义为 -1202 (Informix定义)。

ECPG\_INFORMIX\_BAD\_YEAR

如果在解析一个日期时为年找到了一个坏的值，函数会返回这个值。在内部它被定义为 -1204 (Informix定义)。

ECPG\_INFORMIX\_BAD\_MONTH

如果在解析一个日期时为月找到了一个坏的值，函数会返回这个值。在内部它被定义为 -1205 (Informix定义)。

ECPG\_INFORMIX\_BAD\_DAY

如果在解析一个日期时为日找到了一个坏的值，函数会返回这个值。在内部它被定义为 -1206 (Informix定义)。

**ECPG\_INFORMIX\_ENOSHORTDATE**

如果一个解析例程需要一个短日期表示但是却没有得到正确长度的日期自如穿，函数会返回这个值。在内部它被定义为 -1209 (Informix定义)。

**ECPG\_INFORMIX\_DATE\_CONVERT**

如果在日期格式化时产生了一个错误，函数会返回这个值。在内部它被定义为 -1210 (Informix定义)。

**ECPG\_INFORMIX\_OUT\_OF\_MEMORY**

如果在操作时内存被耗尽，函数会返回这个值。在内部它被定义为 -1211 (Informix定义)。

**ECPG\_INFORMIX\_ENOTDMY**

如果一个解析例程被假定为得到一个格式掩码 (如mmddy) 但是列出的域并不是全部正确，函数会返回这个值。在内部它被定义为 -1212 (Informix定义)。

**ECPG\_INFORMIX\_BAD\_NUMERIC**

如果一个解析例程因为一个numeric值的文本表达包含错误而不能解析它或者一个例程因为至少一个numeric变量非法而无法完成一次涉及numeric变量的计算，函数会返回这个值。在内部它被定义为 -1213 (Informix定义)。

**ECPG\_INFORMIX\_BAD\_EXPONENT**

如果一个解析例程不能解析一个指数，函数会返回这个值。在内部它被定义为 -1216 (Informix定义)。

**ECPG\_INFORMIX\_BAD\_DATE**

如果一个解析例程不能解析一个日期，函数会返回这个值。在内部它被定义为 -1218 (Informix定义)。

**ECPG\_INFORMIX\_EXTRA\_CHARS**

如果一个解析例程被传递了它不能解析的额外字符，函数会返回这个值。在内部它被定义为 -1264 (Informix定义)。

## 34.16. Oracle 兼容模式

ecpg 可以在所谓的 Oracle compatibility mode 下运行。如果该模式处于活动状态，则它会尝试表现得像 Oracle 的 Pro\*C 一样。

具体来说，此模式通过三种方式更改 ecpg：

- 填充字符数组，用于接收具有指定长度的尾随空格的字符串类型
- 零字节终止这些字符数组，并在发生截断时设置指示符变量
- 当字符数组接收到空字符串类型时，将null指示符设置为-1

## 34.17. 内部

这一节解释ECPG在内部如何工作。这些信息有时有助于用户理解如何使用ECPG。

ecpg写到输出的头四行是固定行。两行是注释，两行是与库接口必须的包括行。然后预处理器会从文件读取并且写输出。通常它会把所有东西回显在输出上。

当它看见一个EXEC SQL语句时，它会干预并且改变它。命令开始于EXEC SQL 并且结束于;。之间的任何东西都被视作一个SQL语句，并且会被解析进行变量替换。

当一个符号开始于一个冒号(:)时，变量替换会发生。有该名称的变量会被在之前声明于EXEC SQL DECLARE小节中的变量中搜索。

该库中最重要的函数是ECPGdo，它负责执行大部分命令。它采用可变数量的参数。可以很容易地增加到最多 50 个左右的参数，并且我们希望在任何平台上这都不会成为问题。

参数是：

一个行号

这是原始行的行号，只用于错误消息。

一个字符串

这是要被发出的SQL命令。它会被输入变量修改，即在编译时不知道但是要在命令中被输入的变量。其中变量应该去到包含?的字符串中。

输入变量

每一个输入参数导致十个参数被创建（见下文）。

ECPGt\_EOIT

一个说明没有更多输入变量的enum。

输出变量

每一个输出变量导致十个参数被创建（见下文）。这些变量由该函数填充。

ECPGt\_EORT

一个说明没有更多变量的enum。

对于每一个作为SQL命令一部分的变量，该函数得到十个参数：

1. 作为一个特殊符号的类型。
2. 一个值的指针或者一个指针的指针。
3. 如果变量是一个char或者varchar，这是它的尺寸。
4. 数组中元素的数量（用于数组获取）。
5. 数组中下一个元素的偏移量（用于数组获取）。
6. 作为一个特别符号的指示符变量的类型。
7. 一个指示符变量的指针。
8. 0
9. 指示符数组中的元素数量（用于数组获取）。
10. 到指示符数组中下一个元素的偏移量（用于数组取得）。

注意并非所有 SQL 命令都被以这种方式对待。例如，一个打开游标语句：

```
EXEC SQL OPEN cursor;
```

不会被复制到输出。反而，游标的DECLARE命令被用在OPEN命令的位置上，因为它事实上会打开该游标。

这里有一个完整的例子，它描述了一个文件foo.pgc的预处理器输出（对预处理器的每一个特定版本细节可能不同）：

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

会被翻译成：

```
/* 由 ecpg (2.6.0) 处理 */
/* 这两个头文件由预处理器增加 */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* 声明节开始 */

#line 1 "foo.pgc"

    int index;
    int result;
/* 声明节结束 */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?    ",
        ECPGt_int,&(index), 1L, 1L, sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result), 1L, 1L, sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

（这里的缩进是为了可读性而添加的，并非预处理器做的处理）。

---

## Chapter 35. 信息模式

信息模式由一组视图构成，它们包含定义在当前数据库中对象的信息。信息模式以 SQL 标准定义，因此能够被移植并且保持稳定——系统目录则不同，它们是与LightDB相关的并且是为了实现的考虑而建模的。不过，信息模式视图不包含与LightDB-相关特性有关的信息。要咨询那些信息你需要查询系统目录或其他LightDB-相关视图。

### Note

当在数据库中查询约束信息时，一个期望返回一行的标准兼容的查询可能返回多行。这是因为 SQL 标准要求约束名在一个模式中唯一，但是LightDB并不强制这种限制。LightDB自动产生的约束名避免在相同的模式中重复，但是用户能够指定这种重复的名称。

这个问题可能在查询信息模式视图时出现，例如 `check_constraint_routine_usage`、`check_constraints`、`domain_constraints`和 `referential_constraints`。一些其他视图也有相似的问题，但是它们包含了表名来帮助区分重复行，例如 `constraint_column_usage`、`constraint_table_usage`、`table_constraints`。

### 35.1. 模式

信息模式本身是一个名为 `information_schema` 的模式。这个模式自动存在于所有数据库中。这个模式的拥有者是簇中的初始数据库用户，并且该用户自然地拥有这个模式上的所有特权，包括删除它的能力（但是这样节省的空间是很小的）。

默认情况下，信息模式不在模式搜索路径中，因此你需要使用限定名访问其中的所有对象。因为信息模式中的某些对象的名称是可能出现在用户应用中的一般名称，如果你想把该信息模式放在路径中，你应该小心。

### 35.2. 数据类型

信息模式视图的列使用定义在信息模式中的特殊数据类型。它们被定义为普通内建类型之上的简单域。你不应在信息模式之外使用这些类型进行工作，但是如果你的应用从信息模式中进行选择，那你的应用就必须准备好面对它们。

这些类型是：

`cardinal_number`

一种非负整数。

`character_data`

一种字符串（没有指定最大长度）。

`sql_identifier`

一种字符串。这种类型被用于 SQL 标识符，类型 `character_data` 被用于任何其他类型的文本数据。

`time_stamp`

在类型 `timestamp with time zone` 之上的一个域。

yes\_or\_no

一种字符串域，它包含YES或NO。这被用来在信息模式中表示布尔（真/假）（信息模式是在类型boolean被加到 SQL 标准之前被发明的，因此这个惯例是用来使信息模式向后兼容）。

信息模式中的每一列都是这五种类型之一。

### 35.3. information\_schema\_catalog\_name

information\_schema\_catalog\_name是一个表，它总是包含一行和一列，其中包含了当前数据库（SQL 术语中的当前目录）的名字。

Table 35.1. information\_schema\_catalog\_name 列

列类型	描述
catalog_name sql_identifier	包含这个信息模式的数据库名称

### 35.4. administrable\_role\_authorizations

视图administrable\_role\_authorizations标识当前用户对其有管理选项的所有角色。

Table 35.2. administrable\_role\_authorizations 列

列类型	描述
grantee sql_identifier	授予此角色成员的角色名称(可以是当前用户，或在嵌套角色成员的情况下不同的角色。)
role_name sql_identifier	一个角色的名称
is_grantable yes_or_no	总是 YES

### 35.5. applicable\_roles

视图applicable\_roles当前用户可以使用其特权的所有角色。这意味着有某种角色授权链从当前用户到讨论中的角色。当前用户本身也是一个可应用的角色。可应用的角色集合通常被用于权限检查。

Table 35.3. applicable\_roles 列

列类型	描述
grantee sql_identifier	授予此角色成员的角色名称(可以是当前用户，或在嵌套角色成员的情况下不同的角色。)
role_name sql_identifier	一个角色的名称
is_grantable yes_or_no	

列类型	描述
	如果被授权人在角色上有管理选项则为YES，如果没有则为NO

## 35.6. attributes

视图attributes包含数据库中定义的组合数据类型的属性的有关信息（注意该视图并不给出有关表列的信息，表列有时候在 LightDB 上下文环境中也被称为属性）。只有当前用户能够访问（由于是拥有者获得的权限或是在类型上有某些特权）的那些属性会被显示。

Table 35.4. attributes 列

列类型	描述
udt_catalog sql_identifier	包含该数据类型的数据库名称（总是当前数据库）
udt_schema sql_identifier	包含该数据类型的模式名称
udt_name sql_identifier	数据类型名称
attribute_name sql_identifier	属性名称
ordinal_position cardinal_number	属性在该数据类型内部的顺序位置（从 1 开始计算）
attribute_default character_data	该属性的默认表达式
is_nullable yes_or_no	如果该属性是可能为空的，值为YES，否则为NO。
data_type character_data	如果该属性是一个内建类型，此列值为该属性的数据类型；如果该属性是某种数组，此列值为ARRAY（在这种情况下，见视图element_types）；其他情况，此列值为USER-DEFINED（在这种情况下，该类型在attribute_udt_name和相关列中标识）。
character_maximum_length cardinal_number	如果data_type标识一个字符或位串类型，这里是声明的最大长度；如果没有声明最大长度，则对于所有其他数据类型为空。
character_octet_length cardinal_number	如果data_type标识一个字符类型，这里是一个数据的最大可能长度（以字节计）；对其他所有数据类型为空。最大字节长度取决于声明的字符最大长度（见上文）和服务器的编码。
character_set_catalog sql_identifier	应用于一个LightDB中不可用的特性
character_set_schema sql_identifier	应用于一个LightDB中不可用的特性
character_set_name sql_identifier	应用于一个LightDB中不可用的特性
collation_catalog sql_identifier	包含此属性排序规则的数据库名（总是当前数据库），如果默认或者该属性的数据类型不可排序则为空
collation_schema sql_identifier	包含此属性排序规则的模式名，如果默认或者该属性的数据类型不可排序则为空

列类型	描述
<code>collation_name sql_identifier</code>	该属性排序规则的名称，如果默认或者该属性的数据类型不可排序则为空
<code>numeric_precision cardinal_number</code>	如果 <code>data_type</code> 标识一种数字类型，这列包含这个属性类型的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列 <code>numeric_precision_radix</code> 中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
<code>numeric_precision_radix cardinal_number</code>	如果 <code>data_type</code> 标识一种数字类型，这一列指示 <code>numeric_precision</code> 和 <code>numeric_scale</code> 列中的值是基于什么来表示。该值为 2 或 10。对于所有其他数据类型，这一列为空。
<code>numeric_scale cardinal_number</code>	如果 <code>data_type</code> 标识一种准确数字类型，这列包含这个属性类型的（声明的或隐式的）比例。比例指示了有效位数。它可以按照列 <code>numeric_precision_radix</code> 中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
<code>datetime_precision cardinal_number</code>	如果 <code>data_type</code> 标识一种日期、时间、时间戳或时间间隔类型，这一列包含这个属性类型的（声明的或隐式的）分数秒的精度，也就是秒值的小数点后的十进制位数。对于所有其他数据类型，这一列为空。
<code>interval_type character_data</code>	如果 <code>data_type</code> 标识一种时间间隔类型，这一列包含时间间隔为这个属性包括哪些域的声明，例如YEAR TO MONTH、DAY TO SECOND等等。如果没有指定域限制（也就是该时间间隔接受所有域），并且对于所有其他数据类型，这个域为空。
<code>interval_precision cardinal_number</code>	应用于一个LightDB中不可用的特性（关于时间间隔类型属性的分数秒精度可见 <code>datetime_precision</code> ）
<code>attribute_udt_catalog sql_identifier</code>	属性数据类型被定义的数据库名（总是当前数据库）
<code>attribute_udt_schema sql_identifier</code>	属性数据类型被定义的模式名
<code>attribute_udt_name sql_identifier</code>	属性数据类型的名称
<code>scope_catalog sql_identifier</code>	应用于一个LightDB中不可用的特性
<code>scope_schema sql_identifier</code>	应用于一个LightDB中不可用的特性
<code>scope_name sql_identifier</code>	应用于一个LightDB中不可用的特性
<code>maximum_cardinality cardinal_number</code>	总是空，因为数组在LightDB中总是有无限制的最大势
<code>dtd_identifier sql_identifier</code>	该列的数据类型描述符的一个标识符，在从属于该表的数据类型标识符之中唯一。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
<code>is_derived_reference_attribute yes_or_no</code>	应用于一个LightDB中不可用的特性

关于某些列的详情，参见Section 35.1之下的一个相似结构的视图。



## 35.7. character\_sets

视图character\_sets标识当前数据库中可用的字符集。因为 LightDB 不支持在同一个数据库中有多个字符集，这个视图只显示一个字符集，它就是数据库编码。

注意下列术语在 SQL 标准中是怎样使用的：

字元集 (character repertoire)

字符的一个抽象集合，例如UNICODE、UCS或LATIN1。它不作为一个 SQL 对象显示，但是在这个视图中可见。

字符编码形式 (character encoding form)

某种字元集的一种编码。大部分较老的字元集只使用一种编码形式，并且因此它们没有独立的名字（例如LATIN1就是一种适用于LATIN1字元集的编码形式）。但是 Unicode 就有几种编码形式如UTF8、UTF16等等（并非全部被 LightDB 支持）。编码形式不作为一个 SQL 对象显示，但是在这个视图中可见。

字符集 (character set)

一个标识一种字元集、一种字符编码以及一种默认排序规则的命名 SQL 对象。一个预定义的字符集通常具有和一种编码形式相同的名字，但是用户可以定义其他名字。例如，字符集UTF8通常标识字元集UCS、编码形式UTF8以及某种默认排序规则。

你可以把 LightDB 中的一种“编码”想成一个字符集或是一种字符编码形式。它们将具有相同的名字，并且在一个数据库中只能用其中一个。

Table 35.5. character\_sets 列

列类型	描述
character_set_catalog sql_identifier	当前字符集并未被实现为模式对象，因此这一列为空。
character_set_schema sql_identifier	当前字符集并未被实现为模式对象，因此这一列为空。
character_set_name sql_identifier	该字符集的名字，当前实现为显示该数据库编码的名字
character_repertoire sql_identifier	字元集，如果编码为UTF8则显示UCS，否则只显示编码名称
form_of_use sql_identifier	字符编码形式，与数据库编码相同
default_collate_catalog sql_identifier	包含该默认排序规则的数据库名（如果任意排序规则被标识，总是当前数据库）
default_collate_schema sql_identifier	包含该默认排序规则的模式名
default_collate_name sql_identifier	默认排序规则的名字。该默认排序规则被标识为匹配当前数据库的COLLATE和CTYPE设置的排序规则。如果没有那种排序规则，那么这一列和相关模式以及目录列为空。

## 35.8. check\_constraint\_routine\_usage

视图check\_constraint\_routine\_usage标识由检查约束使用的例程(函数和过程)。只有那些例程显示为当前启用的角色所拥有。

Table 35.6. check\_constraint\_routine\_usage 列

列类型	描述
constraint_catalog sql_identifier	包含约束的数据库名称(总是当前数据库)
constraint_schema sql_identifier	包含约束的模式名称
constraint_name sql_identifier	约束的名称
specific_catalog sql_identifier	包含该函数的数据库名称(总是当前数据库)
specific_schema sql_identifier	包含函数的模式名称
specific_name sql_identifier	函数的“特定名称”。 详情请参见 Section 35.41

## 35.9. check\_constraints

视图check\_constraints包含所有检查约束，不管是定义在一个表上的还是定义在一个域上的，它们被一个当前启用的角色所拥有（表或域的拥有者就是约束的拥有者）。

Table 35.7. check\_constraints 列

列类型	描述
constraint_catalog sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema sql_identifier	包含该约束的模式名称
constraint_name sql_identifier	约束名称
check_clause character_data	该检查约束的检查表达式

## 35.10. collations

视图collations包含在当前数据库中可用的排序规则。

Table 35.8. collations 列

列类型	描述
collation_catalog sql_identifier	包含该排序规则的数据库名称（总是当前数据库）
collation_schema sql_identifier	包含该排序规则的模式名称
collation_name sql_identifier	默认排序规则的名称
pad_attribute character_data	总是NO PAD（另一种选择PAD SPACE没有被 LightDB 支持）

## 35.11. collation\_character\_set\_applicability

视图collation\_character\_set\_applicability标识可用的排序规则适用于哪些字符集。在LightDB中，每个数据库中只有一种字符集（解释见Section 35.7，因此这个视图没有提供很有用的信息。

Table 35.9. collation\_character\_set\_applicability 列

列类型	描述
collation_catalog sql_identifier	包含该排序规则的数据库名称（总是当前数据库）
collation_schema sql_identifier	包含该排序规则的模式名称
collation_name sql_identifier	默认排序规则的名称
character_set_catalog sql_identifier	字符集当前还未被实现为模式对象，所以这一列为空
character_set_schema sql_identifier	字符集当前还未被实现为模式对象，所以这一列为空
character_set_name sql_identifier	字符集名称

## 35.12. column\_column\_usage

视图column\_column\_usage标识依赖于同一表中的另一个基本列生成的所有列。只包含当前启用的角色所拥有的表。

Table 35.10. column\_column\_usage 列

列类型	描述
table_catalog sql_identifier	包含表的数据库的名称(总是当前数据库)
table_schema sql_identifier	包含表的模式的名称
table_name sql_identifier	表的名称
column_name sql_identifier	生成列所依赖的基本列的名称
dependent_column sql_identifier	生成的列的名称

## 35.13. column\_domain\_usage

视图column\_domain\_usage标识所有使用定义在当前数据库中并且被一个当前启用的角色拥有的域的列（表列或视图列）。

Table 35.11. column\_domain\_usage 列

列类型	描述
domain_catalog sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema sql_identifier	包含该域的模式名称
domain_name sql_identifier	域名称
table_catalog sql_identifier	包含该表的数据库名（总是当前数据库）
table_schema sql_identifier	包含该表的模式名称
table_name sql_identifier	表名称
column_name sql_identifier	列名称

## 35.14. column\_options

视图column\_options包含为当前数据库中外部表列定义的所有选项。只有当前用户能够访问（作为拥有者或具有某些特权）的那些外部表列才被显示。

Table 35.12. column\_options 列

列类型	描述
table_catalog sql_identifier	包含该外部表的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该外部表的模式名称
table_name sql_identifier	外部表名称
column_name sql_identifier	列名称
option_name sql_identifier	一个选项名称
option_value character_data	该选项的值

## 35.15. column\_privileges

视图column\_privileges标识所有授予给一个当前启用的角色或者被一个当前启用的角色授予的特权。对每一个列、授予者、被授予者的组合只有一行。

如果一个特权被授予在一整个表上，它在这个视图中被显示为在每一列上授予，但是只有可用于列粒度的特权类型才会这样：SELECT、INSERT、UPDATE、REFERENCES。

Table 35.13. column\_privileges 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称
table_catalog sql_identifier	包含该列的表所在的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该列的表所在的模式名称
table_name sql_identifier	包含该列的表名称
column_name sql_identifier	列名称
privilege_type character_data	特权类型：SELECT、INSERT、UPDATE或 REFERENCES
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO

## 35.16. column\_udt\_usage

视图column\_udt\_usage标识所有使用被一个当前启用的角色拥有的数据类型的列。注意在LightDB中，内建数据类型的行为和用户定义的类型相似，因此它们也被包括在这里。详见Section 35.17

Table 35.14. column\_udt\_usage 列

列类型	描述
udt_catalog sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的数据库名称（总是当前数据库）
udt_schema sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的模式名称
udt_name sql_identifier	该列数据类型（如果适用，底层的域类型）的名称
table_catalog sql_identifier	包含该表的数据库名（总是当前数据库）
table_schema sql_identifier	包含该表的模式名
table_name sql_identifier	表名称
column_name sql_identifier	列名称

## 35.17. columns

视图columns包含数据库中有关所有表列（或视图列）的信息。系统列（ctid等）不被包括在内。只有那些当前用户能够访问（作为拥有者或具有某些特权）的列才被显示。

Table 35.15. columns 列

列类型	描述
table_catalog sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该表的模式名称
table_name sql_identifier	表名称
column_name sql_identifier	列名称
ordinal_position cardinal_number	该列在表内的顺序位置（从 1 开始计）
column_default character_data	该列的默认表达式
is_nullable yes_or_no	如果该列可以为空，则为YES，否则为NO。一个非空约束是让一列成为不能为空的方法，但还有其他方法。
data_type character_data	如果该列的数据类型是一种内建类型，则为该列的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况下该类型被标识在udt_name和相关列中）。如果该列基于一个域，这一列引用该域底层的类型（该列被标识在domain_name和相关列中）。
character_maximum_length cardinal_number	如果data_type标识一种字符或位串类型，这里是声明的最大长度；如果没有声明最大长度或者所有其他数据类型，这里为空。
character_octet_length cardinal_number	如果data_type标识一个字符类型，这里是一个数据的最大可能长度（以字节计）；对其他所有数据类型为空。最大字节长度取决于声明的字符最大长度（见上文）和服务端编码。
numeric_precision cardinal_number	如果data_type标识一种数字类型，这列包含这个属性类型的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
numeric_precision_radix cardinal_number	如果data_type标识一种数字类型，这一列指示numeric_precision和numeric_scale列中的值是基于什么来表示。该值为 2 或 10。对于所有其他数据类型，这一列为空。
numeric_scale cardinal_number	如果data_type标识一种准确数字类型，这列包含这个属性类型的（声明的或隐式的）比例。比例指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
datetime_precision cardinal_number	如果data_type标识一种日期、时间、时间戳或时间间隔类型，这一列包含这个属性类型的（声明的或隐式的）分数秒的精度，也就是秒值的小数点后的十进制位数。对于所有其他数据类型，这一列为空。
interval_type character_data	

列类型	描述
	如果data_type标识一种时间间隔类型，这一列包含时间间隔为这个属性包括哪些域的声明，例如YEAR TO MONTH、DAY TO SECOND等等。如果没有指定域限制（也就是该时间间隔接受所有域），并且对于所有其他数据类型，这个域为空。
interval_precision cardinal_number	应用于一个LightDB中不可用的特性（关于时间间隔类型属性的分数秒精度可见datetime_precision）
character_set_catalog sql_identifier	应用于一个LightDB中不可用的特性
character_set_schema sql_identifier	应用于一个LightDB中不可用的特性
character_set_name sql_identifier	应用于一个LightDB中不可用的特性
collation_catalog sql_identifier	包含此属性排序规则的数据库名（总是当前数据库），如果默认或者该属性的数据类型不可排序则为空
collation_schema sql_identifier	该列的包含排序规则的模式名，如果默认或者该列的数据类型不可排序则为空
collation_name sql_identifier	该列的排序规则的名称，如果默认或者该列的数据类型不可排序则为空
domain_catalog sql_identifier	如果该列有一个域类型，这里是该域所在的数据库名（总是当前数据库），否则为空。
domain_schema sql_identifier	如果该列有一个域类型，这里是该域所在的模式名，否则为空。
domain_name sql_identifier	如果该列有一个域类型，这里是该域的名称，否则为空。
udt_catalog sql_identifier	该列数据类型（如果适用，底层的域类型）被定义的数据库名（总是当前数据库）
udt_schema sql_identifier	该列数据类型（如果适用，域的底层类型）被定义的模式名称
udt_name sql_identifier	该列数据类型（如果适用，域的底层类型）的名称
scope_catalog sql_identifier	应用于一个LightDB中不可用的特性
scope_schema sql_identifier	应用于一个LightDB中不可用的特性
scope_name sql_identifier	应用于一个LightDB中不可用的特性
maximum_cardinality cardinal_number	总是空，因为数组在LightDB中总是有无限制的最大势
dtd_identifier sql_identifier	该列的数据类型描述符的一个标识符，在从属于该表的数据类型标识符之中唯一。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
is_self_referencing yes_or_no	应用于一个LightDB中不可用的特性
is_identity yes_or_no	

列类型	描述
	如果该列是一个标识列，则为YES，否则为NO。
identity_generation character_data	如果该列是一个标识列，则为ALWAYS或者BY DEFAULT，它反映该列的定义。
identity_start character_data	如果该列是一个标识列，则是内部序列的起始值，否则为空。
identity_increment character_data	如果该列是一个标识列，则是内部序列的增量，否则为空。
identity_maximum character_data	如果该列是一个标识列，则是内部序列的最大值，否则为空。
identity_minimum character_data	如果该列是一个标识列，则是内部序列的最小值，否则为空。
identity_cycle yes_or_no	如果该列是一个标识列，则YES和NO分别表示内部序列可循环和不可循环，否则为空。
is_generated character_data	如果列是生成的列，则ALWAYS，否则 NEVER。
generation_expression character_data	如果列是生成的列，则生成表达式，否则为空。
is_updatable yes_or_no	如果该列是可更新的，则为YES，否则为NO（基表中的列总是可更新的，视图中的列则不一定）

因为在 SQL 中有多种方式定义数据类型，而LightDB还包含额外的方式来定义数据类型，它们在信息模式中的表示可能有点困难。列data\_type应该标识列的底层内建类型。在LightDB中，这表示定义在系统目录模式pg\_catalog中的类型。如果应用能够特别地（例如以不同方式格式化数字类型或使用精度列中的数据）处理众所周知的内建类型，这列可能会有用。列udt\_name、udt\_schema和udt\_catalog总是标识列的底层数据类型，即使该列是基于一个域的（因为LightDB对待内建类型和用户定义类型的方式是一样的，内建类型也出现在这里。这是 SQL 标准的一种扩展）。如果一个应用想要根据该类型以不同的方式处理数据，就应该使用这些列，因为在那种情况下即使该列真地基于一个域也没有关系。如果该列是基于一个域，该域的标识被存储在列domain\_name、domain\_schema和domain\_catalog。如果你想要把列和它们相关的数据类型配对并且把域视作单独的类型，你可以写coalesce(domain\_name, udt\_name)等等。

## 35.18. constraint\_column\_usage

视图constraint\_column\_usage标识在当前数据库中被某个约束使用的所有列。只有包含在被一个当前启用的角色拥有的表中的那些列才被显示。对于一个检查约束，这个视图标识被用在该检查约束中的列。对于一个外键约束，这个视图标识外键引用的列。对于一个唯一或主键约束，这个视图标识被约束的列。

Table 35.16.constraint\_column\_usage 列

列类型	描述
table_catalog sql_identifier	包含被某个约束使用的列的表所在的数据库名称（总是当前数据库）
table_schema sql_identifier	包含被某个约束使用的列的表所在的模式名称
table_name sql_identifier	包含被某个约束使用的列的表名称



列类型	描述
column_name sql_identifier	包含被某个约束使用的列名称
constraint_catalog sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema sql_identifier	包含该约束的模式名称
constraint_name sql_identifier	约束名称

## 35.19. constraint\_table\_usage

视图constraint\_table\_usage标识在当前数据库中被某个约束使用的所有表（这与视图table\_constraints不同，它标识哪些表约束定义在哪些表上）。对于一个外键约束，这个视图标识该外键引用的表。对于一个唯一或主键约束，这个视图仅标识该约束属于的表。检查约束和非空约束不被包括在这个视图中。

Table 35.17. constraint\_table\_usage 列

列类型	描述
table_catalog sql_identifier	包含被某个约束使用的表的数据库名称（总是当前数据库）
table_schema sql_identifier	包含被某个约束使用的表的模式名称
table_name sql_identifier	包含被某个约束使用的表名称
constraint_catalog sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema sql_identifier	包含该约束的模式名称
constraint_name sql_identifier	约束名称

## 35.20. data\_type\_privileges

视图data\_type\_privileges标识当前用户能够访问（作为被描述对象的拥有者或者具有其上的某种特权）的所有数据类型描述符。只要一个数据类型被用在一个表列、一个域或一个函数（作为参数或返回类型）就会生成一个数据类型描述符并且在那个实例中存储一些有关该数据类型如何被使用的信息（例如，声明的最大长度，如果适用）。每一个数据类型描述符被赋予一个任意的标识符，它在被赋予给一个对象（表、域、函数）的数据类型描述符中唯一。这个视图对于应用可能没什么用，但是它被用于定义信息模式中的一些其他视图。

Table 35.18. data\_type\_privileges 列

列类型	描述
object_catalog sql_identifier	包含该被描述对象的数据库名称（总是当前数据库）
object_schema sql_identifier	包含该被描述对象的模式名称

列类型	描述
object_name sql_identifier	该描述对象的名称
object_type character_data	被描述对象的类型：TABLE（从属于表的一列的数据类型描述符）、DOMAIN（从属于域的数据类型描述符）、ROUTINE（从属于函数的一个参数或返回数据类型的数据类型描述符）。
dtd_identifier sql_identifier	数据类型描述符的标识符，它在同一对象的数据类型描述符之间唯一。

## 35.21. domain\_constraints

视图domain\_constraints包含所有属于当前数据库中定义的域的约束。只有当前用户能访问的那些域才被显示（作为拥有者或具有某些特权）。

Table 35.19. domain\_constraints 列

列类型	描述
constraint_catalog sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema sql_identifier	包含该约束的模式名称
constraint_name sql_identifier	约束名称
domain_catalog sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema sql_identifier	包含该域的模式名称
domain_name sql_identifier	该域的名称
is_deferrable yes_or_no	如果该约束是可延迟的，则为YES，否则为NO
initially_deferred yes_or_no	如果该约束是可延迟的且初始就被延迟，则为YES，否则为NO

## 35.22. domain\_udt\_usage

视图domain\_udt\_usage标识所有基于被一个当前启用的角色拥有的数据类型域的域。注意在LightDB中，内建数据类型的行为类似于用户定义的类型，因此它们也被包括在这里。

Table 35.20. domain\_udt\_usage 列

列类型	描述
udt_catalog sql_identifier	该域数据类型被定义的数据库名称（总是当前数据库）
udt_schema sql_identifier	该域数据类型被定义的模式名称
udt_name sql_identifier	该域数据类型的名称

列类型	描述
domain_catalog sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema sql_identifier	包含该域的模式名称
domain_name sql_identifier	该域的名称

## 35.23. domains

视图domains包含定义在当前数据库中的所有域。只有当前用户能够访问（作为拥有者或具有某些特权）的域才被显示。

Table 35.21. domains 列

列类型	描述
domain_catalog sql_identifier	包含该域的数据库名称（总是当前数据库）
domain_schema sql_identifier	包含该域的模式名称
domain_name sql_identifier	该域的名称
data_type character_data	该域的数据类型如果是一种内建类型，这里是该域的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在udt_name和相关列中）。
character_maximum_length cardinal_number	如果该域有一个字符或位串类型，这里是声明的最大长度；如果没有声明最大长度，则对于所有其他数据类型为空。
character_octet_length cardinal_number	如果该域有一个字符类型，这里是一个数据的最大可能长度（以字节计）；对其他所有数据类型为空。最大字节长度取决于声明的字符最大长度（见上文）和服务器编码。
character_set_catalog sql_identifier	应用于一个LightDB中不可用的特性
character_set_schema sql_identifier	应用于一个LightDB中不可用的特性
character_set_name sql_identifier	应用于一个LightDB中不可用的特性
collation_catalog sql_identifier	包含此域排序规则的数据库名（总是当前数据库），如果默认或者该域的数据类型不可排序则为空
collation_schema sql_identifier	包含此域排序规则的模式名，如果默认或者该域的数据类型不可排序则为空
collation_name sql_identifier	该域排序规则的名称，如果默认或者该域的数据类型不可排序则为空
numeric_precision cardinal_number	如果该域有一种数字类型，这列包含这个域类型的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为

列类型	描述
	十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
numeric_precision radix cardinal_number	如果该域有一种数字类型，这一列指示numeric_precision和numeric_scale列中的值是基于什么来表示。该值为 2 或 10。对于所有其他数据类型，这一列为空。
numeric_scale cardinal_number	如果该域有一种准确数字类型，这列包含这个域类型的（声明的或隐式的）比例。比例指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。对于所有其他数据类型，这一列为空。
datetime_precision cardinal_number	如果data_type标识一种日期、时间、时间戳或时间间隔类型，这一列包含这个域类型的（声明的或隐式的）分数秒的精度，也就是秒值的小数点后的十进制位数。对于所有其他数据类型，这一列为空。
interval_type character_data	如果data_type标识一种时间间隔类型，这一列包含时间间隔为这个域包括哪些域的声明，例如YEAR TO MONTH、DAY TO SECOND等等。如果没有指定域限制（也就是该时间间隔接受所有域），并且对于所有其他数据类型，这个域为空。
interval_precision cardinal_number	应用于一个LightDB中不可用的特性（关于时间间隔类型域的分数秒精度可见datetime_precision）
domain_default character_data	该域的默认表达式
udt_catalog sql_identifier	该域数据类型被定义的数据库名称（总是当前数据库）
udt_schema sql_identifier	该域数据类型被定义的模式名称
udt_name sql_identifier	该域数据类型的名称
scope_catalog sql_identifier	应用于一个LightDB中不可用的特性
scope_schema sql_identifier	应用于一个LightDB中不可用的特性
scope_name sql_identifier	应用于一个LightDB中不可用的特性
maximum_cardinality cardinal_number	总是空，因为数组在LightDB中总是有无限制的最大势
dtd_identifier sql_identifier	该域的数据类型描述符的一个标识符，在从属于该域的数据类型标识符之中唯一（这不重要，因为一个域只包含一个数据类型描述符）。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。

## 35.24. element\_types

视图element\_types包含数组元素的数据类型描述符。当一个表列、组合类型属性、域、函数参数或函数返回值被定义为一种数组类型，相应的信息模式视图只在列data\_type中包

含ARRAY。要获得该数组元素类型的信息，你可以连接该相应的视图和这个视图。例如，要显示一个表的列及其数据类型和数组元素类型，你可以：

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
    ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE',
    c.dtd_identifier)
    = (e.object_catalog, e.object_schema, e.object_name, e.object_type,
    e.collection_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

这个视图只包括当前用户能够访问（作为所有者或具有某些特权）的对象。

Table 35.22. element\_types 列

列类型	描述
object_catalog sql_identifier	包含使用被描述的数组的对象的数据库名称（总是当前数据库）
object_schema sql_identifier	包含使用被描述的数组的对象的模式名称
object_name sql_identifier	使用被描述的模式的对象名称
object_type character_data	使用被描述的数组的对象的类型：TABLE（被一个表列使用的数组）、USER-DEFINED TYPE（被组合类型的一个属性使用的数组）、DOMAIN（被域使用的数组）、ROUTINE（被函数的一个参数或返回数据类型使用的数组）。
collection_type_identifier sql_identifier	被描述的数组的数据类型描述符的标识符。使用这个去与其他信息模式视图的dtd_identifier列连接。
data_type character_data	如果数组元素的数据类型是内建类型，这里是数组元素的数据类型，否则为USER-DEFINED（在那种情况下，该类型被标识在udt_name和相关列中）。
character_maximum_length cardinal_number	总是为空，因为这种信息不适用于LightDB中的数组元素数据类型
character_octet_length cardinal_number	总是为空，因为这种信息不适用于LightDB中的数组元素数据类型
character_set_catalog sql_identifier	应用于一个LightDB中不可用的特性
character_set_schema sql_identifier	应用于一个LightDB中不可用的特性
character_set_name sql_identifier	应用于一个LightDB中不可用的特性
collation_catalog sql_identifier	包含元素类型排序规则的数据库名称（总是当前数据库），如果默认或该元素的数据类型是不可排序的则为空
collation_schema sql_identifier	包含元素类型排序规则的模式名称，如果默认或该元素的数据类型是不可排序的则为空
collation_name sql_identifier	元素类型的排序规则名，如果默认或该元素的数据类型是不可排序的则为空

列类型	描述
numeric_precision cardinal_number	总是为空，因为这种信息不适用于LightDB中的数组元素数据类型
numeric_precision_radix cardinal_number	总是为空，因为这种信息不适用于LightDB中的数组元素数据类型
numeric_scale cardinal_number	总是为空，因为这种信息不适用于LightDB中的数组元素数据类型
datetime_precision cardinal_number	总是为空，因为这种信息不适用于LightDB中的数组元素数据类型
interval_type character_data	总是为空，因为这种信息不适用于LightDB中的数组元素数据类型
interval_precision cardinal_number	总是为空，因为这种信息不适用于LightDB中的数组元素数据类型
domain_default character_data	还未被实现
udt_catalog sql_identifier	元素的数据类型所在的数据库名称（总是当前数据库）
udt_schema sql_identifier	定义元素的数据类型的模式的名称
udt_name sql_identifier	元素的数据类型名
scope_catalog sql_identifier	应用于一个LightDB中不可用的特性
scope_schema sql_identifier	应用于一个LightDB中不可用的特性
scope_name sql_identifier	应用于一个LightDB中不可用的特性
maximum_cardinality cardinal_number	总是空，因为数组在LightDB中总是有无限制的最大势
dtd_identifier sql_identifier	该元素的数据类型描述符的标识符。当前没有用。

## 35.25. enabled\_roles

视图enabled\_roles标识当前“已被启用的角色”。已被启用的角色被递归地定义为：当前用户以及被授予给具有自动继承的已被启用角色的所有角色。换句话说，就是当前用户是其直接或间接成员的所有角色。

为了权限检查，“可应用角色”的集合被应用，它会比已被启用角色的集合包含的角色范围更宽。因此通常使用视图applicable\_roles要更好，applicable\_roles视图的详情请见Section 35.5

Table 35.23. enabled\_roles 列

列类型	描述
role_name sql_identifier	角色的名称

## 35.26. foreign\_data\_wrapper\_options

视图foreign\_data\_wrapper\_options包含为当前数据库中外部数据包装器定义的所有选项。只有那些当前用户能够访问（作为拥有者或具有某些特权）的外部数据包装器被显示。

Table 35.24. foreign\_data\_wrapper\_options 列

列类型	描述
foreign_data_wrapper_catalog sql_identifier	该外部数据包装器所在的数据库名称（总是当前数据库）
foreign_data_wrapper_name sql_identifier	该外部数据包装器的名称
option_name sql_identifier	一个选项的名称
option_value character_data	该选项的值

## 35.27. foreign\_data\_wrappers

视图foreign\_data\_wrappers包含定义在当前数据库中的所有外部数据包装器。只有那些当前用户能够访问（作为拥有者或具有某些特权）的外部数据包装器才会被显示。

Table 35.25. foreign\_data\_wrappers 列

列类型	描述
foreign_data_wrapper_catalog sql_identifier	包含该外部数据包装器的数据库名称（总是当前数据库）
foreign_data_wrapper_name sql_identifier	外部数据包装器的名称
authorization_identifier sql_identifier	外部服务器拥有者的名称
library_name character_data	实现这个外部数据包装器的库文件名称
foreign_data_wrapper_language character_data	用于实现这个外部数据包装器的语言

## 35.28. foreign\_server\_options

视图foreign\_server\_options包含为当前数据库中外部服务器定义的所有选项。只有那些当前用户能够访问（作为拥有者或具有某些特权）的外部服务器才会被显示。

Table 35.26. foreign\_server\_options 列

列类型	描述
foreign_server_catalog sql_identifier	该外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name sql_identifier	

列类型	描述
	该外部服务器的名称
option_name sql_identifier	一个选项的名称
option_value character_data	该选项的值

## 35.29. foreign\_servers

视图foreign\_servers包含当前数据库中定义的所有外部服务器。只有那些当前用户能够访问（作为所有者或具有某些特权）的外部服务器才会被显示。

Table 35.27. foreign\_servers 列

列类型	描述
foreign_server_catalog sql_identifier	该外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name sql_identifier	该外部服务器的名称
foreign_data_wrapper_catalog sql_identifier	包含被该外部服务器使用的外部数据包装器的数据库名称（总是当前数据库）
foreign_data_wrapper_name sql_identifier	被该外部服务器所使用的外部数据包装器的名称
foreign_server_type character_data	外部服务器类型信息，如果在创建时指定过
foreign_server_version character_data	外部服务器版本信息，如果在创建时指定过
authorization_identifier sql_identifier	该外部服务器的拥有者的名称

## 35.30. foreign\_table\_options

视图foreign\_table\_options包含为当前数据库中外部表定义的所有选项。只有那些当前用户能够访问（作为所有者或具有某些特权）的外部表才会被显示。

Table 35.28. foreign\_table\_options 列

列类型	描述
foreign_table_catalog sql_identifier	包含该外部表的数据库名称（总是当前数据库）
foreign_table_schema sql_identifier	包含该外部表的模式名称
foreign_table_name sql_identifier	外部表的名称
option_name sql_identifier	一个选项的名称
option_value character_data	



列类型	描述
	该选项的值

## 35.31. foreign\_tables

视图foreign\_tables包含定义在当前数据库中的所有外部表。只有那些当前用户能够访问（作为所有者或具有某些特权）的外部表才会被显示。

Table 35.29. foreign\_tables 列

列类型	描述
foreign_table_catalog sql_identifier	该外部表所在的数据库名称（总是当前数据库）
foreign_table_schema sql_identifier	包含该外部表的模式名称
foreign_table_name sql_identifier	该外部表的名称
foreign_server_catalog sql_identifier	该外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name sql_identifier	外部服务器的名称

## 35.32. key\_column\_usage

视图key\_column\_usage标识当前数据库中所有被某种唯一、主键或外键约束限制的列。检查约束不被包括在这个视图中。只有那些当前用户能够访问的列才会被显示（作为所有者或具有某些特权）。

Table 35.30. key\_column\_usage 列

列类型	描述
constraint_catalog sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema sql_identifier	包含该约束的模式名称
constraint_name sql_identifier	约束的名称
table_catalog sql_identifier	包含被这个约束限制的列的表所在的数据库名称（总是当前数据库）
table_schema sql_identifier	包含被这个约束限制的列的表所在的模式名称
table_name sql_identifier	包含被这个约束限制的列的表的名称
column_name sql_identifier	被这个约束限制的列的名称
ordinal_position cardinal_number	该列在约束键中的顺序位置（从 1 开始计数）
position_in_unique_constraint cardinal_number	

列类型	描述
	对于一个外键约束，被引用行在其唯一约束中的顺序位置（从 1 开始计数）；对于其他约束为空

## 35.33. parameters

视图parameters包含当前数据库中所有函数的参数的有关信息。只有那些当前用户能够访问（作为拥有者或具有某些特权）的函数才会被显示。

Table 35.31. parameters 列

列类型	描述
specific_catalog sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema sql_identifier	包含该函数的模式名称
specific_name sql_identifier	该函数的“专用名”。更多信息参见Section 35.41
ordinal_position cardinal_number	该参数在函数参数列表中的顺序位置（从 1 开始计数）
parameter_mode character_data	IN表示输入参数，OUT表示输出参数，INOUT表示输入/输出参数。
is_result yes_or_no	应用于一个LightDB中不可用的特性
as_locator yes_or_no	应用于一个LightDB中不可用的特性
parameter_name sql_identifier	参数名称，如果参数没有名称则为空
data_type character_data	该参数的数据类型如果是一种内建类型，这里是该参数的数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为USER-DEFINED（此种情况中，该类型被标识在udt_name和相关列中）。
character_maximum_length cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
character_octet_length cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
character_set_catalog sql_identifier	应用于一个LightDB中不可用的特性
character_set_schema sql_identifier	应用于一个LightDB中不可用的特性
character_set_name sql_identifier	应用于一个LightDB中不可用的特性
collation_catalog sql_identifier	总是为空，因为这种信息不适用于LightDB中的返回数据类型
collation_schema sql_identifier	总是为空，因为这种信息不适用于LightDB中的返回数据类型
collation_name sql_identifier	总是为空，因为这种信息不适用于LightDB中的返回数据类型

列类型	描述
numeric_precision cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
numeric_precision_radix cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
numeric_scale cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
datetime_precision cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
interval_type character_data	总是为空，因为这种信息不适用于LightDB中的返回数据类型
interval_precision cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
udt_catalog sql_identifier	该参数的数据类型所在的数据库名称（总是当前数据库）
udt_schema sql_identifier	该参数的数据类型所在的模式名称
udt_name sql_identifier	该参数的数据类型的名称
scope_catalog sql_identifier	应用于一个LightDB中不可用的特性
scope_schema sql_identifier	应用于一个LightDB中不可用的特性
scope_name sql_identifier	应用于一个LightDB中不可用的特性
maximum_cardinality cardinal_number	总是空，因为数组在LightDB中总是有无限制的最大势
dtd_identifier sql_identifier	该参数的数据类型描述符的一个标识符，在从属于该函数的数据类型标识符之中唯一（这不重要，因为一个域只包含一个数据类型描述符）。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
parameter_default character_data	该参数的默认表达式，如果没有或者该函数不被一个当前启用的角色拥有则为空值。

## 35.34. referential\_constraints

视图referential\_constraints包含当前数据库中的所有引用（外键）约束。只有那些当前用户具有其引用表上写权限（作为拥有者或具有某些除SELECT之外的特权）的约束才会被显示。

Table 35.32. referential\_constraints 列

列类型	描述
constraint_catalog sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema sql_identifier	包含该约束的模式名称

列类型	描述
constraint_name sql_identifier	约束的名称
unique_constraint_catalog sql_identifier	包含该外键约束所引用的唯一或主键约束的数据库名称（总是当前数据库）
unique_constraint_schema sql_identifier	包含该外键约束所引用的唯一或主键约束的模式名称
unique_constraint_name sql_identifier	包含该外键约束所引用的唯一或主键约束的名称
match_option character_data	外键约束的匹配选项： FULL、PARTIAL或NONE。
update_rule character_data	外键约束的更新规则： CASCADE、SET NULL、SET DEFAULT、RESTRICT或 NO ACTION。
delete_rule character_data	外键约束的删除规则： CASCADE、SET NULL、SET DEFAULT、RESTRICT或 NO ACTION。

## 35.35. role\_column\_grants

视图role\_column\_grants标识所有在列上授予的特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在column\_privileges中找到。这个视图和column\_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当前用户获得其访问权限的列。

Table 35.33.role\_column\_grants 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称
table_catalog sql_identifier	包含该列的表所在的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该列的表所在的模式名称
table_name sql_identifier	包含该列的表名称
column_name sql_identifier	列名称
privilege_type character_data	特权类型： SELECT、INSERT、UPDATE或 REFERENCES
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO

## 35.36. role\_routine\_grants

视图role\_routine\_grants标识所有在函数上授予的特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在routine\_privileges中找到。这个视图

和routine\_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当前用户获得其访问权限的函数。

Table 35.34. role\_routine\_grants 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称
specific_catalog sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema sql_identifier	包含该函数的模式名称
specific_name sql_identifier	该函数的“专用名”。详见Section 35.41
routine_catalog sql_identifier	包含该函数的数据库名称（总是当前数据库）
routine_schema sql_identifier	包含该函数的模式名称
routine_name sql_identifier	该函数的名称（在重载的情况下可能会重复）
privilege_type character_data	总是为EXECUTE（函数唯一的特权类型）
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO

## 35.37. role\_table\_grants

视图role\_table\_grants标识所有在表或视图上授予的特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在table\_privileges中找到。这个视图和table\_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当前用户获得其访问权限的表。

Table 35.35. role\_table\_grants 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称
table_catalog sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该表的模式名称
table_name sql_identifier	表名称
privilege_type character_data	该特权的类型：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES或TRIGGER

列类型	描述
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO
with_hierarchy yes_or_no	在 SQL 标准中，WITH HIERARCHY OPTION是一个独立的（子）特权，它允许在表继承层级上的特定操作。在 LightDB 中，这被包括在SELECT特权中，因此这一列在特权为SELECT时显示YES，其他时候显示NO。

## 35.38. role\_udt\_grants

视图role\_udt\_grants标识所有在用户定义类型上授予的USAGE特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在udt\_privileges中找到。这个视图和udt\_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当前用户获得其访问权限的对象。因为数据类型在 LightDB 中并没有真正的特权，而是只有一个给PUBLIC的隐式授予，这个视图为空。

Table 35.36.role\_udt\_grants 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称
udt_catalog sql_identifier	包含该类型的数据库名称（总是当前数据库）
udt_schema sql_identifier	包含该类型的模式名称
udt_name sql_identifier	类型的名称
privilege_type character_data	总是TYPE USAGE
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO

## 35.39. role\_usage\_grants

视图role\_usage\_grants标识所有在多种对象上授予的USAGE特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。更多信息可以在usage\_privileges中找到。这个视图和usage\_privileges之间的唯一实质性区别是：这个视图忽略那些以授予给PUBLIC的方式使当前用户获得其访问权限的对象。

Table 35.37.role\_usage\_grants 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称

列类型	描述
object_catalog sql_identifier	包含该对象的数据库名称（总是当前数据库）
object_schema sql_identifier	如果适用，则为包含该对象的模式名称，否则为一个空字符串
object_name sql_identifier	对象的名称
object_type character_data	COLLATION或DOMAIN或FOREIGN DATA WRAPPER或FOREIGN SERVER或SEQUENCE
privilege_type character_data	总是USAGE
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO

## 35.40. routine\_privileges

视图routine\_privileges标识所有在函数上授予的特权，其授予者或被授予者是一个当前已被启用的角色。对于每一种函数、授予者和被授予者的组合，这里都有一行。

Table 35.38.routine\_privileges 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称
specific_catalog sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema sql_identifier	包含该函数的模式名称
specific_name sql_identifier	该函数的“专用名”。详见Section 35.41。
routine_catalog sql_identifier	包含该函数的数据库名称（总是当前数据库）
routine_schema sql_identifier	包含该函数的模式名称
routine_name sql_identifier	该函数的名称（在重载的情况下可能重复）
privilege_type character_data	总是EXECUTE（函数唯一的特权类型）
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO

## 35.41. routines

视图routines包含当前数据库中所有的函数和过程。只有那些当前用户能够访问（作为拥有者或具有某些特权）的函数和过程才会被显示。

Table 35.39.routines 列

列类型	描述
specific_catalog sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema sql_identifier	包含该函数的模式名称
specific_name sql_identifier	该函数的“专用名”。这是一个在模式中唯一标识该函数的名称，即使该函数真正的名称已经被重载。专用名的格式尚未被定义，它应当仅被用来与指定例程名称的其他实例进行比较。
routine_catalog sql_identifier	包含该函数的数据库名称（总是当前数据库）
routine_schema sql_identifier	包含该函数的模式名称
routine_name sql_identifier	该函数的名称（在重载的情况下可能重复）
routine_type character_data	FUNCTION表示是一个函数，PROCEDURE表示是一个过程
module_catalog sql_identifier	应用于一个LightDB中不可用的特性
module_schema sql_identifier	应用于一个LightDB中不可用的特性
module_name sql_identifier	应用于一个LightDB中不可用的特性
udt_catalog sql_identifier	应用于一个LightDB中不可用的特性
udt_schema sql_identifier	应用于一个LightDB中不可用的特性
udt_name sql_identifier	应用于一个LightDB中不可用的特性
data_type character_data	该函数的返回数据类型如果是一种内建类型，这里是该数据类型；如果是某种数组（此种情况见视图element_types），则为ARRAY；否则为用户-DEFINED（此种情况中，该类型被标识在type_udt_name和相关列中）。如果是过程则此项为空。
character_maximum_length cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
character_octet_length cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
character_set_catalog sql_identifier	应用于一个LightDB中不可用的特性
character_set_schema sql_identifier	应用于一个LightDB中不可用的特性
character_set_name sql_identifier	应用于一个LightDB中不可用的特性
collation_catalog sql_identifier	总是为空，因为这种信息不适用于LightDB中的返回数据类型
collation_schema sql_identifier	



列类型	描述
	总是为空，因为这种信息不适用于LightDB中的返回数据类型
collation_name sql_identifier	总是为空，因为这种信息不适用于LightDB中的返回数据类型
numeric_precision cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
numeric_precision_radix cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
numeric_scale cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
datetime_precision cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
interval_type character_data	总是为空，因为这种信息不适用于LightDB中的返回数据类型
interval_precision cardinal_number	总是为空，因为这种信息不适用于LightDB中的返回数据类型
type_udt_catalog sql_identifier	该函数的返回数据类型所在的数据库名（总是当前数据库）。如果是过程则此项为空。
type_udt_schema sql_identifier	该函数的返回数据类型所在的模式名。如果是过程则此项为空。
type_udt_name sql_identifier	该函数的返回数据类型的名字。如果是过程则此项为空。
scope_catalog sql_identifier	应用于一个LightDB中不可用的特性
scope_schema sql_identifier	应用于一个LightDB中不可用的特性
scope_name sql_identifier	应用于一个LightDB中不可用的特性
maximum_cardinality cardinal_number	总是空，因为数组在LightDB中总是有无限制的最大势
dtd_identifier sql_identifier	该函数返回数据类型的数据类型描述符的一个标识符，在从属于该函数的数据类型标识符之中唯一。这主要用于与这类标识符的其他实例进行连接（该标识符的指定格式没有被定义并且不保证在未来的版本中保持相同）。
routine_body character_data	如果该函数是一个 SQL 函数，则为SQL，否则为EXTERNAL。
routine_definition character_data	该函数的源文本（如果该函数不属于一个当前已被启用的角色，则为空）。（根据 SQL 标准，只有routine_body为SQL时这一列才适用。但是在LightDB中，它将会包含该函数被创建时所指定的任何源文本。）
external_name character_data	如果这个函数是一个 C 函数，则为该函数的外部名称（链接符号），否则为空（这会产生和显示在routine_definition中相同的值）。
external_language character_data	该函数所用的语言
parameter_style character_data	总是GENERAL（SQL 标准定义了其他参数风格，但在LightDB中不可用）

列类型	描述
is_deterministic yes_or_no	如果该函数被声明为不变（在 SQL 标准中被称为确定性的），则为YES，否则为NO（你不能通过该信息模式查询在LightDB中可用的其他易变级别）。
sql_data_access character_data	总是MODIFIES，表示该函数可能修改 SQL 数据。这种信息对LightDB没有用处。
is_null_call yes_or_no	如果该函数在任一参数为空时自动返回空值，则为YES，否则为NO。如果是过程则此项为空。
sql_path character_data	应用于一个LightDB中不可用的特性
schema_level_routine yes_or_no	总是YES（反例是一个用户定义类型的方法，这是在LightDB不可用的一种特性）。
max_dynamic_result_sets cardinal_number	应用于一个LightDB中不可用的特性
is_user_defined_cast yes_or_no	应用于一个LightDB中不可用的特性
is_implicitly_invocable yes_or_no	应用于一个LightDB中不可用的特性
security_type character_data	如果该函数以当前用户的特权运行，则为INVOKER；如果该函数以定义它的用户的特权运行，则为DEFINER。
to_sql_specific_catalog sql_identifier	应用于一个LightDB中不可用的特性
to_sql_specific_schema sql_identifier	应用于一个LightDB中不可用的特性
to_sql_specific_name sql_identifier	应用于一个LightDB中不可用的特性
as_locator yes_or_no	应用于一个LightDB中不可用的特性
created time_stamp	应用于一个LightDB中不可用的特性
last_altered time_stamp	应用于一个LightDB中不可用的特性
new_savepoint_level yes_or_no	应用于一个LightDB中不可用的特性
is_udt_dependent yes_or_no	当前总是NO。另一个选项YES应用于一个LightDB中不可用的特性。
result_cast_from_data_type character_data	应用于一个LightDB中不可用的特性
result_cast_as_locator yes_or_no	应用于一个LightDB中不可用的特性
result_cast_char_max_length cardinal_number	应用于一个LightDB中不可用的特性
result_cast_char_octet_length cardinal_number	应用于一个LightDB中不可用的特性
result_cast_char_set_catalog sql_identifier	

列类型	描述
	应用于一个LightDB中不可用的特性
result_cast_char_set_schema sql_identifier	应用于一个LightDB中不可用的特性
result_cast_char_set_name sql_identifier	应用于一个LightDB中不可用的特性
result_cast_collation_catalog sql_identifier	应用于一个LightDB中不可用的特性
result_cast_collation_schema sql_identifier	应用于一个LightDB中不可用的特性
result_cast_collation_name sql_identifier	应用于一个LightDB中不可用的特性
result_cast_numeric_precision cardinal_number	应用于一个LightDB中不可用的特性
result_cast_numeric_precision_radix cardinal_number	应用于一个LightDB中不可用的特性
result_cast_numeric_scale cardinal_number	应用于一个LightDB中不可用的特性
result_cast_datetime_precision cardinal_number	应用于一个LightDB中不可用的特性
result_cast_interval_type character_data	应用于一个LightDB中不可用的特性
result_cast_interval_precision cardinal_number	应用于一个LightDB中不可用的特性
result_cast_type_udt_catalog sql_identifier	应用于一个LightDB中不可用的特性
result_cast_type_udt_schema sql_identifier	应用于一个LightDB中不可用的特性
result_cast_type_udt_name sql_identifier	应用于一个LightDB中不可用的特性
result_cast_scope_catalog sql_identifier	应用于一个LightDB中不可用的特性
result_cast_scope_schema sql_identifier	应用于一个LightDB中不可用的特性
result_cast_scope_name sql_identifier	应用于一个LightDB中不可用的特性
result_cast_maximum_cardinality cardinal_number	应用于一个LightDB中不可用的特性
result_cast_dtd_identifier sql_identifier	应用于一个LightDB中不可用的特性

## 35.42. schemata

视图schemata包含当前数据库中被当前用户（作为属主或具有某些特权）可访问的所有模式。

Table 35.40. schemata 列

列类型	描述
catalog_name sql_identifier	该模式所在的数据库名称（总是当前数据库）
schema_name sql_identifier	模式的名称
schema_owner sql_identifier	模式拥有者的名称
default_character_set_catalog sql_identifier	应用于一个LightDB中不可用的特性
default_character_set_schema sql_identifier	应用于一个LightDB中不可用的特性
default_character_set_name sql_identifier	应用于一个LightDB中不可用的特性
sql_path character_data	应用于一个LightDB中不可用的特性

## 35.43. sequences

视图sequences包含所有定义在当前数据库中的序列。只有那些当前用户能够访问（作为拥有者或具有某些特权）的序列才会被显示。

Table 35.41. sequences 列

列类型	描述
sequence_catalog sql_identifier	包含该序列的数据库名称（总是当前数据库）
sequence_schema sql_identifier	包含该序列的模式名称
sequence_name sql_identifier	序列的名称
data_type character_data	序列的数据类型
numeric_precision cardinal_number	这列包含这个序列数据类型（见上文）的（声明的或隐式的）精度。精度指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。
numeric_precision_radix cardinal_number	这一列指示numeric_precision和numeric_scale列中的值是基于什么来表示。该值为 2 或 10。
numeric_scale cardinal_number	这列包含这个序列数据类型（见上文）的（声明的或隐式的）比例。比例指示了有效位数。它可以按照列numeric_precision_radix中指定的被表示为十进制（基于 10）或二进制（基于 2）。
start_value character_data	该序列的开始值
minimum_value character_data	该序列的最小值

列类型	描述
maximum_value character_data	该序列的最大值
increment character_data	该序列的增量
cycle_option yes_or_no	如果该序列会循环, 则为YES, 否则为NO

注意依照 SQL 标准, 开始值、最小值、最大值和增量值被作为字符串返回。

## 35.44. sql\_features

表sql\_features包含的信息指示了哪些 SQL 标准中定义的正式特性被LightDB所支持。这和Appendix 中的信息一样。这里你也能找到一些额外的背景信息。

Table 35.42. sql\_features 列

列类型	描述
feature_id character_data	特性的标识符字符串
feature_name character_data	特性的描述性名称
sub_feature_id character_data	该子特性的标识符字符串, 或者如果不是一个子特性则为一个长度为零的字符串
sub_feature_name character_data	该子特性的描述性名称, 或者如果不是一个子特性则为一个长度为零的字符串
is_supported yes_or_no	如果该特性被当前版本的LightDB完全支持, 则为YES, 否则为NO
is_verified_by character_data	总是为空, 因为LightDB开发组没有对特性的一致性执行正式的测试
comments character_data	可能会是关于该特性被支持状态的一段注释

## 35.45. sql\_implementation\_info

表sql\_implementation\_info包含的信息指示剩下的由 SQL 标准实现定义的多个方面。这类信息主要用来在 ODBC 接口的情境中使用; 其它接口的用户可能将发现这类信息用处不大。由于这个原因, 个体实现信息项没有在这里描述, 你将会在 ODBC 接口的描述中找到它们。

Table 35.43. sql\_implementation\_info 列

列类型	描述
implementation_info_id character_data	实现信息项的标识符字符串
implementation_info_name character_data	实现信息项的描述性名称
integer_value cardinal_number	该实现信息项的值, 如果该值被包含在character_value列中则为空 character_value

列类型	描述
character_value character_data	该实现信息项的值，如果该值被包含在integer_value列中则为空 integer_value
comments character_data	可能是从属于该实现信息项的一段注释

## 35.46. sql\_parts

表sql\_parts包含的信息指示哪些定义在 SQL 标准中的部分被LightDB支持。

Table 35.44. sql\_parts 列

列类型	描述
feature_id character_data	包含该部分编号的一个标识符字符串
feature_name character_data	该部分的描述性名称
is_supported yes_or_no	如果当前版本的LightDB完全支持该部分，则为YES，否则为NO
is_verified_by character_data	总是为空，因为LightDB开发组没有对特性的一致性执行正式的测试
comments character_data	可能会是关于该部分被支持状态的一段注释

## 35.47. sql\_sizing

表sql\_sizing包含有关LightDB中多种尺寸限制和最大值的信息。这类信息主要用来在 ODBC 接口的情境中使用；其它接口的用户可能将发现这类信息用处不大。由于这个原因，个体实现信息项没有在这里描述，你将会在 ODBC 接口的描述中找到它们。

Table 35.45. sql\_sizing 列

列类型	描述
sizing_id cardinal_number	尺寸项的标识符
sizing_name character_data	尺寸项的描述性名称
supported_value cardinal_number	尺寸项的值，如果尺寸是不受限制或不能确定的则为 0，如果尺寸项适用的特性不受支持则为空
comments character_data	可能是从属于尺寸项的一段注释

## 35.48. table\_constraints

视图table\_constraints包含属于特定表的所有约束，这些表要满足的条件是：当前用户拥有表或者是当前用户在表上具有某种除SELECT之外的特权。

Table 35.46. table\_constraints 列

列类型	描述
constraint_catalog sql_identifier	包含该约束的数据库名称（总是当前数据库）
constraint_schema sql_identifier	包含该约束的模式名称
constraint_name sql_identifier	约束名称
table_catalog sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该表的模式名称
table_name sql_identifier	表名称
constraint_type character_data	该约束的类型：CHECK、 FOREIGN KEY、 PRIMARY KEY或UNIQUE
is_deferrable yes_or_no	如果该约束是可延迟的，则为YES，否则为NO
initially_deferred yes_or_no	如果该约束是可延迟的并且是初始已被延迟，则为YES，否则为NO
enforced yes_or_no	适用于一种LightDB中不可用的特性（当前总是YES）

## 35.49. table\_privileges

视图table\_privileges标识在表或视图上所有被授予的特权，这些特权必须是被一个当前已被启用角色授出或者被授予给一个当前已被启用角色。对每一个表、授予者和被授予者的组合都有一行。

Table 35.47. table\_privileges 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称
table_catalog sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该表的模式名
table_name sql_identifier	表名称
privilege_type character_data	特权类型：SELECT、 INSERT、 UPDATE、 DELETE、 TRUNCATE、 REFERENCES或TRIGGER
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO

列类型	描述
with_hierarchy yes_or_no	在 SQL 标准中，WITH HIERARCHY OPTION是一个独立的（子）特权，它允许在表继承层级上的特定操作。在 LightDB 中，这被包括在SELECT特权中，因此这一列在特权为SELECT时显示YES，其他时候显示NO。

## 35.50. tables

视图tables包含定义在当前数据库中的所有表和视图。只有那些当前用户能够访问（作为拥有者或具有某些特权）的表和视图才会被显示。

Table 35.48. tables 列

列类型	描述
table_catalog sql_identifier	包含该表的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该表的模式名称
table_name sql_identifier	表名称
table_type character_data	该表的类型：BASE TABLE表示一个持久的基本表（常见表类型），VIEW表示一个视图，FOREIGN表示一个外部表，LOCAL TEMPORARY表示一个临时表
self_referencing_column_name sql_identifier	应用于一个LightDB中不可用的特性
reference_generation character_data	应用于一个LightDB中不可用的特性
user_defined_type_catalog sql_identifier	如果该表是一个有类型的表，则是包含其底层数据类型的数据库名称（总是当前数据库），否则为空。
user_defined_type_schema sql_identifier	如果该表是一个有类型的表，则是包含其底层数据类型的模式名，否则为空。
user_defined_type_name sql_identifier	如果该表是一个有类型的表，则是其底层数据类型的名称，否则为空。
is_insertable_into yes_or_no	如果该表能够被插入，则为YES，否则为NO（基本表总是能被插入，而视图则不一定）。
is_typed yes_or_no	如果该表是一个有类型的表，则为YES，否则为NO
commit_action character_data	还未被实现

## 35.51. transforms

视图transforms包含定义在当前数据库中的转换的信息。更准确来说，包含在转换中的每一个函数（“FROM SQL”或者“TO SQL”函数）在其中都有一行。



Table 35.49. transforms 列

列类型	描述
udt_catalog sql_identifier	包含该转换所适用类型的数据库的名称（总是当前数据库）
udt_schema sql_identifier	包含该转换所适用类型的模式的名称
udt_name sql_identifier	该转换所适用类型的名称
specific_catalog sql_identifier	包含该函数的数据库的名称（总是当前数据库）
specific_schema sql_identifier	包含该函数的模式的名称
specific_name sql_identifier	该函数的“专用名”。更多信息参见Section 35.41
group_name sql_identifier	SQL 标准允许在“组”中定义转换，并且在运行时选择一个组。LightDB 不支持这种做法，转换是与一种语言相关的。作为一种折衷，这个域包含该转换所适用的语言。
transform_type character_data	FROM SQL或者TO SQL

## 35.52. triggered\_update\_columns

对于当前数据库中指定一个列列表（如UPDATE OF column1, column2）的触发器，视图triggered\_update\_columns标识这些列。没有指定一个列列表的触发器不被包括在这个视图中。只有那些当前用户拥有或具有某种除SELECT之外特权的列才会被显示。

Table 35.50. triggered\_update\_columns 列

列类型	描述
trigger_catalog sql_identifier	包含该触发器的数据库名称（总是当前数据库）
trigger_schema sql_identifier	包含该触发器的模式名称
trigger_name sql_identifier	触发器的名称
event_object_catalog sql_identifier	包含触发器所在的表的数据库名称（总是当前数据库）
event_object_schema sql_identifier	包含触发器所在的表的模式名称
event_object_table sql_identifier	触发器所在的表的名称
event_object_column sql_identifier	触发器所在的列的名称

## 35.53. triggers

视图triggers包含所有定义在当前数据库中表和视图上的触发器，并且只显示当前用户拥有的触发器或者是当前用户在其上具有某种除SELECT之外特权的触发器。

Table 35.51. triggers 列

列类型	描述
trigger_catalog sql_identifier	包含该触发器的数据库名称（总是当前数据库）
trigger_schema sql_identifier	包含该触发器的模式名称
trigger_name sql_identifier	触发器的名称
event_manipulation character_data	触发该触发器的事件（INSERT、UPDATE或DELETE）
event_object_catalog sql_identifier	包含触发器所在的表的数据库名称（总是当前数据库）
event_object_schema sql_identifier	包含该触发器所在的表的模式名称
event_object_table sql_identifier	该触发器所在的表的名称
action_order cardinal_number	同一个表上具有相同event_manipulation、action_timing和action_orientation的触发器之间的触发顺序。在LightDB中，触发器按照名称顺序被触发，因此这一列会反映这种规则。
action_condition character_data	触发器的WHEN条件，如果没有则为空（如果该表不被一个当前已启用角色拥有也是为空）
action_statement character_data	触发器执行的语句（当前总是 EXECUTE FUNCTION function(...)）
action_orientation character_data	标识触发器是对每个被处理的行触发一次还是为每个语句触发一次（ROW或STATEMENT）
action_timing character_data	触发器在什么时候触发（BEFORE、AFTER或INSTEAD OF）
action_reference_old_table sql_identifier	“旧”传递表的名称，如果没有则为空
action_reference_new_table sql_identifier	“新”传递表的名称，如果没有则为空
action_reference_old_row sql_identifier	应用于一个LightDB中不可用的特性
action_reference_new_row sql_identifier	应用于一个LightDB中不可用的特性
created time_stamp	应用于一个LightDB中不可用的特性

LightDB中的触发器有两点与 SQL 标准不兼容，这会影响到该信息模式中的表示。第一，在LightDB中触发器的名字是局限于每个表的，而不是独立于模式对象。因此可能在一个模式

中会有重复的触发器名称，只要它们属于不同的表（trigger\_catalog和trigger\_schema才真正标识了触发器被定义在哪个表上）。第二，在LightDB中触发器可以被定义为在多个事件上触发（例如ON INSERT OR UPDATE），而在 SQL 标准中只允许一个。如果一个触发器被定义为在多个事件上触发，它在信息模式中被表示为多行，每一行对应于一类事件。作为这两个问题的结果，视图triggers的主键实际上是(trigger\_catalog, trigger\_schema, event\_object\_table, trigger\_name, event\_manipulation)，而不是(trigger\_catalog, trigger\_schema, trigger\_name)（这是 SQL 标准指定的）。尽管如此，如果你以符合 SQL 标准（在模式中触发器名称唯一并且每个触发器只能有一种事件类型）的方式定义你的触发器，这将不会影响你。

Note

在LightDB 9.1 之前，这个视图的列 action\_timing、action\_reference\_old\_table、action\_reference\_new\_table、action\_reference\_old\_row和 action\_reference\_new\_row 分别被命名为 condition\_timing、condition\_reference\_old\_table、condition\_reference\_new\_table、condition\_reference\_old\_row和 condition\_reference\_new\_row。那也是它们在 SQL:1999 标准中的命名。新的命名遵循 SQL:2003 及其后的版本。

## 35.54. udt\_privileges

视图udt\_privileges标识所有在用户定义类型上授予的 USAGE特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。对每一个类型、授予者和被授予者的组合都有一行。这个视图只显示组合类型（原因见下面的 Section 35.50 。 域特权见Section 35.55

Table 35.52. udt\_privileges 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称
udt_catalog sql_identifier	包含该类型的数据库名称（总是当前数据库）
udt_schema sql_identifier	包含该类型的模式名称
udt_name sql_identifier	类型的名称
privilege_type character_data	总是TYPE USAGE
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO

## 35.55. usage\_privileges

视图usage\_privileges标识所有在多种对象上授予的USAGE特权，这些特权的授予者或者被授予者是一个当前已被启用的角色。在LightDB中，这当前适用于排序规则、域、外部数据包装器、外部服务器和序列。对每一个对象、授予者和被授予者都有一行。

由于在LightDB中排序规则并没有真正的特权，这个视图对所有排序规则显示由所有者授予给PUBLIC的隐式非可授予的USAGE特权。但是对其他对象类型则显示真实的特权。

在 LightDB 中，序列也支持除USAGE之外的SELECT和UPDATE特权。这些是非标准的并且因此在该信息模式中不可见。

Table 35.53. usage\_privileges 列

列类型	描述
grantor sql_identifier	授予该特权的角色名称
grantee sql_identifier	被授予该特权的角色名称
object_catalog sql_identifier	包含该对象的数据库名称（总是当前数据库）
object_schema sql_identifier	如果适用，则是包含该对象的模式名称，否则为一个空字符串
object_name sql_identifier	对象的名称
object_type character_data	COLLATION或DOMAIN或FOREIGN DATA WRAPPER或FOREIGN SERVER或SEQUENCE
privilege_type character_data	总是USAGE
is_grantable yes_or_no	如果该特权是可授予的，则为YES，否则为NO

## 35.56. user\_defined\_types

视图user\_defined\_types目前包含定义在当前数据库中的所有组合类型。只有那些当前用户能够访问（作为所有者或具有某些特权）的类型才会被显示。

SQL 知道两种用户定义类型：结构类型（在LightDB中也被称为组合类型）以及独特类型（在LightDB没有实现）。要经得起未来的考验，请使用列user\_defined\_type\_category来区分它们。其他用户定义类型如基类型和枚举（都是LightDB的扩展）不会被显示在这里。对于域，请见Section 35.23

Table 35.54. user\_defined\_types 列

列类型	描述
user_defined_type_catalog sql_identifier	包含该类型的数据库名称（总是当前数据库）
user_defined_type_schema sql_identifier	包含该类型的模式名称
user_defined_type_name sql_identifier	类型的名称
user_defined_type_category character_data	当前总是STRUCTURED
is_instantiable yes_or_no	应用于一个LightDB中不可用的特性
is_final yes_or_no	应用于一个LightDB中不可用的特性

列类型	描述
ordering_form character_data	应用于一个LightDB中不可用的特性
ordering_category character_data	应用于一个LightDB中不可用的特性
ordering_routine_catalog sql_identifier	应用于一个LightDB中不可用的特性
ordering_routine_schema sql_identifier	应用于一个LightDB中不可用的特性
ordering_routine_name sql_identifier	应用于一个LightDB中不可用的特性
reference_type character_data	应用于一个LightDB中不可用的特性
data_type character_data	应用于一个LightDB中不可用的特性
character_maximum_length cardinal_number	应用于一个LightDB中不可用的特性
character_octet_length cardinal_number	应用于一个LightDB中不可用的特性
character_set_catalog sql_identifier	应用于一个LightDB中不可用的特性
character_set_schema sql_identifier	应用于一个LightDB中不可用的特性
character_set_name sql_identifier	应用于一个LightDB中不可用的特性
collation_catalog sql_identifier	应用于一个LightDB中不可用的特性
collation_schema sql_identifier	应用于一个LightDB中不可用的特性
collation_name sql_identifier	应用于一个LightDB中不可用的特性
numeric_precision cardinal_number	应用于一个LightDB中不可用的特性
numeric_precision_radix cardinal_number	应用于一个LightDB中不可用的特性
numeric_scale cardinal_number	应用于一个LightDB中不可用的特性
datetime_precision cardinal_number	应用于一个LightDB中不可用的特性
interval_type character_data	应用于一个LightDB中不可用的特性
interval_precision cardinal_number	应用于一个LightDB中不可用的特性
source_dtd_identifier sql_identifier	应用于一个LightDB中不可用的特性
ref_dtd_identifier sql_identifier	应用于一个LightDB中不可用的特性

## 35.57. user\_mapping\_options

视图user\_mapping\_options包含在当前数据库中为用户映射定义的所有选项。只有那些当前用户能够访问其相应外部服务器（作为拥有者或具有某些特权）的用户映射才会被显示。

Table 35.55. user\_mapping\_options 列

列类型	描述
authorization_identifier sql_identifier	被映射的用户名称，如果映射是公共的则为PUBLIC
foreign_server_catalog sql_identifier	这个映射所使用的外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name sql_identifier	这个映射所使用的外部服务器的名称
option_name sql_identifier	一个选项的名称
option_value character_data	选项的值。除非当前用户是被映射的用户或者映射是PUBLIC的并且当前用户是服务器拥有者或者超级用户，这一列将显示为空。这样做的目的是保护作为用户映射选项存储的口令信息。

## 35.58. user\_mappings

视图user\_mappings包含定义在当前数据库中的所有用户映射。只有当前用户能够访问其对应外部服务器（作为拥有者或具有某些特权）的用户映射才会被显示。

Table 35.56. user\_mappings 列

列类型	描述
authorization_identifier sql_identifier	被映射的用户名称，如果映射是公共的则为PUBLIC
foreign_server_catalog sql_identifier	这个映射所使用的外部服务器所在的数据库名称（总是当前数据库）
foreign_server_name sql_identifier	这个映射所使用的外部服务器的名称

## 35.59. view\_column\_usage

视图view\_column\_usage标识被使用在一个视图的查询表达式（定义该视图的SELECT语句）中的所有列。只有当包含一列的表被一个当前已被启用角色拥有时，该列才会被包括在这个视图中。

### Note

系统表列不被包括。在某个时候这应该会被修复。

Table 35.57. view\_column\_usage 列

列类型	描述
view_catalog sql_identifier	包含该视图的数据库名称（总是当前数据库）
view_schema sql_identifier	包含该视图的模式名称
view_name sql_identifier	视图的名称
table_catalog sql_identifier	被视图所使用的列所属表的数据库名称（总是当前数据库）
table_schema sql_identifier	被视图所使用的列所属表的模式名称
table_name sql_identifier	被视图所使用的列所属表的名称
column_name sql_identifier	被该视图所使用的列名称

## 35.60. view\_routine\_usage

视图view\_routine\_usage标识被使用在一个视图的查询表达式（定义该视图的SELECT语句）中的所有例程（函数和过程）。只有被一个当前已被启用角色拥有的例程才会被包括在这个视图中。

Table 35.58. view\_routine\_usage 列

列类型	描述
table_catalog sql_identifier	包含该视图的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该视图的模式名称
table_name sql_identifier	视图的名称
specific_catalog sql_identifier	包含该函数的数据库名称（总是当前数据库）
specific_schema sql_identifier	包含该函数的模式名称
specific_name sql_identifier	该函数的“专用名”。详情请参见Section 35.41

## 35.61. view\_table\_usage

视图view\_table\_usage标识被使用在一个视图的查询表达式（定义该视图的SELECT语句）中的所有表。只有被一个当前已被启用角色拥有的表才会被包括在这个视图中。

### Note

系统表没有被包括。这应当会在某个时候被修复。

Table 35.59.view\_table\_usage 列

列类型	描述
view_catalog sql_identifier	包含该视图的数据库名称（总是当前数据库）
view_schema sql_identifier	包含该视图的模式名称
view_name sql_identifier	视图的名称
table_catalog sql_identifier	包含被该视图所使用的表的数据库名称（总是当前数据库）
table_schema sql_identifier	包含被该视图所使用的表的模式名称
table_name sql_identifier	包含被该视图所使用的表的名称

## 35.62. views

视图views包含定义在当前数据库中的所有视图。只有当前用户能够访问（作为所有者或具有某些特权）的视图才会被显示。

Table 35.60.views 列

列类型	描述
table_catalog sql_identifier	包含该视图的数据库名称（总是当前数据库）
table_schema sql_identifier	包含该视图的模式名称
table_name sql_identifier	视图的名称
view_definition character_data	定义视图的查询表达式（如果该视图不被一个当前已被启用角色拥有则为空）
check_option character_data	CASCADED 或 LOCAL 如果视图上有CHECK OPTION 定义, 如果没有则为NONE。
is_updatable yes_or_no	如果该视图是可更新的（允许UPDATE和DELETE），则为YES，否则为NO
is_insertable_into yes_or_no	如果该视图是可插入的（允许INSERT），则为YES，否则为NO
is_trigger_updatable yes_or_no	如果该视图上有一个INSTEAD OF UPDATE触发器，则为YES，否则为NO
is_trigger_deletable yes_or_no	如果该视图上有一个INSTEAD OF DELETE触发器，则为YES，否则为NO
is_trigger_insertable_into yes_or_no	如果该视图上有一个INSTEAD OF INSERT触发器，则为YES，否则为NO



---

## Part V. 服务器编程

这部分关于使用用户定义的函数、数据类型、触发器等扩展服务器功能。这些是高级主题，读者应该在理解了有关LightDB的所有其他用户文档之后才阅读这些主题。这一部分的后面一些章节描述LightDB发布中可用的服务器端编程语言，以及与服务器端编程语言相关的一般性问题。在深入研究服务器端编程语言的材料之前，请至少阅读Chapter 36中前几节（覆盖函数）。

---

---

## Table of Contents

36. 扩展 SQL .....	963
36.1. 扩展性如何工作 .....	963
36.2. LightDB类型系统 .....	963
36.2.1. 基础类型 .....	963
36.2.2. 容器类型 .....	963
36.2.3. 域 .....	964
36.2.4. 伪类型 .....	964
36.2.5. 多态类型 .....	964
36.3. 用户定义的函数 .....	965
36.4. 用户定义的过程 .....	966
36.5. 查询语言 (SQL) 函数 .....	966
36.5.1. SQL函数的参数 .....	967
36.5.2. 基本类型上的SQL .....	968
36.5.3. 组合类型上的SQL函数 .....	969
36.5.4. 带有输出参数的SQL函数 .....	972
36.5.5. 带有可变数量参数的SQL函数 .....	973
36.5.6. 带有参数默认值的SQL函数 .....	974
36.5.7. SQL 函数作为表来源 .....	974
36.5.8. 返回集合的SQL函数 .....	975
36.5.9. 返回TABLE的SQL函数 .....	978
36.5.10. 多态SQL函数 .....	979
36.5.11. 带有排序规则的SQL函数 .....	981
36.6. 函数重载 .....	981
36.7. 函数易变性分类 .....	982
36.8. 过程语言函数 .....	983
36.9. 内部函数 .....	983
36.10. C 语言函数 .....	984
36.10.1. 动态载入 .....	984
36.10.2. C 语言函数中的基本类型 .....	985
36.10.3. 版本 1 的调用约定 .....	987
36.10.4. 编写代码 .....	990
36.10.5. 编译和链接动态载入的函数 .....	991
36.10.6. 组合类型参数 .....	993
36.10.7. 返回行 (组合类型) .....	994
36.10.8. 返回集合 .....	995
36.10.9. 多态参数和返回类型 .....	1000
36.10.10. 共享内存和 LWLock .....	1002
36.10.11. 把 C++ 用于可扩展性 .....	1002
36.11. 函数优化信息 .....	1003
36.12. 用户定义的聚集 .....	1003
36.12.1. 移动聚集模式 .....	1005
36.12.2. 多态和可变聚集 .....	1006
36.12.3. 有序集聚集 .....	1007
36.12.4. 部分聚集 .....	1008
36.12.5. 聚集的支持函数 .....	1009
36.13. 用户定义的类型 .....	1009
36.13.1. TOAST 考量 .....	1012
36.14. 用户定义的操作符 .....	1013
36.15. 操作符优化信息 .....	1014
36.15.1. COMMUTATOR .....	1014
36.15.2. NEGATOR .....	1015
36.15.3. RESTRICT .....	1015
36.15.4. JOIN .....	1016
36.15.5. HASHES .....	1016
36.15.6. MERGES .....	1017

36.16.	索引的接口扩展 .....	1017
36.16.1.	索引方法和操作符类 .....	1018
36.16.2.	索引方法策略 .....	1018
36.16.3.	索引方法支持例程 .....	1019
36.16.4.	一个例子 .....	1020
36.16.5.	操作符类和操作符族 .....	1022
36.16.6.	操作符类上的系统依赖 .....	1025
36.16.7.	排序操作符 .....	1026
36.16.8.	操作符类的特性 .....	1027
36.17.	打包相关对象到一个扩展中 .....	1027
36.17.1.	扩展文件 .....	1028
36.17.2.	扩展可再定位性 .....	1030
36.17.3.	扩展配置表 .....	1031
36.17.4.	扩展更新 .....	1031
36.17.5.	用更新脚本安装扩展 .....	1032
36.17.6.	Security Considerations for Extensions .....	1032
36.17.7.	扩展实例 .....	1033
36.18.	扩展的构建基础设施 .....	1034
37.	触发器 .....	1039
37.1.	触发器行为概述 .....	1039
37.2.	数据改变的可见性 .....	1041
37.3.	用 C 编写触发器函数 .....	1042
37.4.	一个完整的触发器实例 .....	1044
38.	事件触发器 .....	1048
38.1.	事件触发器行为总览 .....	1048
38.2.	事件触发器触发矩阵 .....	1048
38.3.	用 C 编写事件触发器函数 .....	1052
38.4.	一个完整的事件触发器例子 .....	1053
38.5.	一个表重写事件触发器例子 .....	1054
39.	规则系统 .....	1056
39.1.	查询树 .....	1056
39.2.	视图和规则系统 .....	1057
39.2.1.	SELECT规则如何工作 .....	1058
39.2.2.	非SELECT语句中的视图规则 .....	1062
39.2.3.	LightDB中视图的能力 .....	1063
39.2.4.	更新一个视图 .....	1063
39.3.	物化视图 .....	1064
39.4.	INSERT、UPDATE和DELETE上的规则 .....	1066
39.4.1.	更新规则如何工作 .....	1067
39.4.2.	与视图合作 .....	1071
39.5.	规则和权限 .....	1077
39.6.	规则和命令状态 .....	1078
39.7.	规则 vs 触发器 .....	1079
40.	过程语言 .....	1082
40.1.	安装过程语言 .....	1082
41.	PL/pgSQL — SQL过程语言 .....	1084
41.1.	综述 .....	1084
41.1.1.	使用PL/pgSQL的优点 .....	1084
41.1.2.	支持的参数和结果数据类型 .....	1084
41.2.	PL/pgSQL的结构 .....	1085
41.3.	声明 .....	1086
41.3.1.	声明函数参数 .....	1087
41.3.2.	ALIAS .....	1089
41.3.3.	复制类型 .....	1090
41.3.4.	行类型 .....	1090
41.3.5.	记录类型 .....	1091
41.3.6.	PL/pgSQL变量的排序规则 .....	1091
41.4.	表达式 .....	1092

41.5.	基本语句	1092
41.5.1.	赋值	1093
41.5.2.	执行没有结果的命令	1093
41.5.3.	执行一个有单行结果的命令	1094
41.5.4.	执行动态命令	1095
41.5.5.	获得结果状态	1098
41.5.6.	什么也不做	1099
41.6.	控制结构	1100
41.6.1.	从一个函数返回	1100
41.6.2.	从过程中返回	1102
41.6.3.	调用存储过程	1102
41.6.4.	条件	1103
41.6.5.	简单循环	1106
41.6.6.	通过查询结果循环	1108
41.6.7.	通过数组循环	1109
41.6.8.	俘获错误	1110
41.6.9.	获得执行位置信息	1113
41.7.	游标	1114
41.7.1.	声明游标变量	1114
41.7.2.	打开游标	1114
41.7.3.	使用游标	1116
41.7.4.	通过一个游标的结果循环	1119
41.8.	事务管理	1119
41.9.	错误和消息	1120
41.9.1.	报告错误和消息	1120
41.9.2.	检查断言	1122
41.10.	触发器函数	1122
41.10.1.	数据改变的触发器	1123
41.10.2.	事件触发器	1130
41.11.	PL/pgSQL的内部	1131
41.11.1.	变量替换	1131
41.11.2.	计划缓存	1132
41.12.	PL/pgSQL开发提示	1134
41.12.1.	处理引号	1134
41.12.2.	额外的编译时和运行时检查	1136
41.13.	从Oracle PL/SQL 移植	1137
41.13.1.	移植示例	1138
41.13.2.	其他要关注的事项	1143
41.13.3.	附录	1143
42.	PL/oraSQL — SQL过程化语言	1146
42.1.	概述	1146
42.1.1.	使用 PL/oraSQL 的优势	1146
42.1.2.	支持的参数和结果数据类型	1146
42.2.	PL/oraSQL 的结构	1147
42.2.1.	匿名块	1147
42.2.2.	注释	1150
42.3.	声明	1151
42.3.1.	声明函数参数	1152
42.3.2.	复制类型	1153
42.3.3.	行类型	1153
42.3.4.	记录类型	1153
42.4.	表达式	1154
42.5.	基本语句	1154
42.5.1.	赋值	1154
42.5.2.	在PLoraSQL中执行DML	1154
42.5.3.	执行一个单行结果的查询	1155
42.5.4.	执行动态命令	1156
42.5.5.	完全不做任何操作	1158

42.6.	控制结构	1158
42.6.1.	从函数中返回	1159
42.6.2.	从过程返回	1159
42.6.3.	调用存储过程	1159
42.6.4.	条件语句	1159
42.6.5.	简单循环	1162
42.6.6.	GOTO语句	1164
42.6.7.	遍历查询结果	1167
42.6.8.	捕获错误	1168
42.7.	游标	1168
42.7.1.	隐式游标	1169
42.7.2.	显式游标	1171
42.7.3.	处理查询结果集	1180
42.7.4.	游标变量	1183
42.8.	集合	1189
42.8.1.	关联数组	1190
42.8.2.	嵌套表	1190
42.8.3.	可变数组	1191
42.8.4.	集合方法	1192
42.9.	事务管理	1192
42.9.1.	使用事务操作游标	1193
42.9.2.	自治事务	1201
42.10.	软件包	1201
42.10.1.	包概述	1201
42.10.2.	创建包	1202
42.10.3.	创建和访问包	1205
42.10.4.	示例	1207
42.10.5.	限制	1208
42.11.	表函数	1209
42.12.	错误处理	1210
42.12.1.	异常处理概述	1210
42.12.2.	预定义异常	1210
42.12.3.	显式抛出异常	1216
42.12.4.	支持DBMS_UTILITY内置包format_error_backtrace函数	1216
42.12.5.	支持 SQLCODE	1217
42.13.	对象名称搜索规则	1217
42.14.	针对Oracle的增强功能	1218
43.	服务器编程接口	1220
43.1.	接口函数	1220
43.2.	接口支持函数	1253
43.3.	内存管理	1262
43.4.	事务管理	1272
43.5.	数据改变的可见性	1275
43.6.	例子	1275
44.	后台工作者进程	1279
45.	逻辑解码	1282
45.1.	逻辑解码的例子	1282
45.2.	逻辑解码概念	1284
45.2.1.	逻辑解码	1284
45.2.2.	复制槽	1284
45.2.3.	输出插件	1285
45.2.4.	导出快照	1285
45.3.	流复制协议接口	1285
45.4.	逻辑解码的 SQL 接口	1285
45.5.	与逻辑解码相关的系统目录	1286
45.6.	逻辑解码输出插件	1286
45.6.1.	初始化函数	1286
45.6.2.	能力	1286

45.6.3.	输出模式 .....	1286
45.6.4.	输出插件回调 .....	1287
45.6.5.	用于产生输出的函数 .....	1289
45.7.	逻辑解码输出写入器 .....	1289
45.8.	逻辑解码的同步复制支持 .....	1289
45.8.1.	总览 .....	1289
45.8.2.	注意事项 .....	1290
46.	复制进度追踪 .....	1291

---

## Chapter 36. 扩展 SQL

在下面的小节中，我们将讨论如何通过增加各种元素来扩展LightDB SQL 查询语言：

- 函数（从Section 36.3开始）
- 聚集（从Section 36.12开始）
- 数据类型（从Section 36.13开始）
- 操作符（从Section 36.14开始）
- 用于索引的操作符类（从Section 36.16开始）
- 相关对象的包（从Section 36.17开始）

### 36.1. 扩展性如何工作

LightDB是可扩展的，因为它的操作是目录驱动的。如果你熟悉标准的关系型数据库系统，你会知道它们把有关数据库、表、列等的信息存储在众所周知的系统目录中（某些系统称之为数据目录）。目录对于用户来说好像其他的表一样，但是DBMS把自己的内部信息记录在其中。LightDB和标准关系型数据库系统的一个关键不同是LightDB在其目录中存储更多信息：不只是有关表和列的信息，还有关于数据类型、函数、访问方法等的信息。这些表可以被用户修改，并且因为LightDB的操作是基于这些表的，所以LightDB可以被用户扩展。通过比较，传统数据库系统只能通过源代码中改变硬编码的过程或者载入由DBMS提供者特殊编写的模块进行扩展。

此外，LightDB服务器能够通过动态载入把用户编写的代码结合到它自身中。也就是，用户能够指定一个实现了一个新类型或函数的对象代码文件（例如一个共享库），并且LightDB将按照要求载入它。把用SQL编写的代码加入到服务器会更加轻而易举。这种“即时”修改其操作的能力让LightDB独特地适合新应用和存储结构的快速原型设计。

### 36.2. LightDB类型系统

LightDB数据类型被划分为基础类型、容器类型、域和伪类型。

#### 36.2.1. 基础类型

基础类型是那些被实现在SQL语言层面之下的类型（通常用一种底层语言，如C），例如integer。它们通常对应于常说的抽象数据类型。LightDB只能通过由用户提供的函数在这类类型上操作，并且只能理解到用户描述这种类型行为的程度。Chapter 8中描述了内建的基础类型。

枚举（enum）类型可以被认为是基础类型的一个子类。主要区别是它们可以使用SQL命令创建，不需要用到底层的编程。更多信息请参考Section 8.7

#### 36.2.2. 容器类型

LightDB有三种“容器”类型，它们是包含多个其他类型值的类型。它们是数组、组合以及范围。

数组可以保存全部是同种类型的多个值。为每一种基本类型、组合类型、范围类型以及域类型都会自动创建一个数组类型。但是没有数组的数组。就类型系统的认知而言，多维数组就和一维数组一样。更多信息请参考Section 8.15

只要用户创建一个表，就会创建组合类型或者行类型。也可以使用CREATE TYPE来定义一个没有关联表的“stand-alone”组合类型。一个组合类型只是一个具有相关域名称的类型列表。一个组合类型的值是一个行或者域值记录。用户可以访问来自SQL查询的组成域。更多信息请参考Section 8.16

范围类型可以保存同种类型的两个值，它们是该范围的上下界。范围类型是用户创建的，不过也存在一些内建的范围类型。更多信息请参考Section 8.17

### 36.2.3. 域

一个域是基于一种特定底层类型的，并且出于很多目的它可以与其底层类型互换。不过，一个域能够具有约束来限制它的合法值于其底层基础类型允许值的一个子集。可以使用SQL命令CREATE DOMAIN创建域。更多信息请参考Section 8.18

### 36.2.4. 伪类型

有一些用于特殊目的“伪类型”。伪类型不能作为表列或者容器类型的组件出现，但是它们能被用于声明函数的参数和结果类型。这在类型系统中提供了一种机制来标识函数的特殊分类。Table 8.2列出了现有的伪类型。

### 36.2.5. 多态类型

特别让人感兴趣的一些伪类型是polymorphic types，用于声明polymorphic functions。这种强大的特性允许单个函数定义对许多不同的数据类型进行操作，具体数据类型由特定调用中实际传递给它的数据类型确定。多态类型显示在Table 36.1中。它们的一些用例出现在Section 36.5.1里面。

Table 36.1. Polymorphic Types

名称	家族	描述
anyelement	Simple	表示函数接受任何数据类型
anyarray	Simple	表示函数接受任何数组数据类型
anynonarray	Simple	表示函数接受任何非数组数据类型
anyenum	Simple	表示函数接受任何enum数据类型(参见 Section 8.7)
anyrange	Simple	表示函数接受任何范围数据类型(参见 Section 8.17)
anycompatible	Common	表示函数接受任何数据类型，并自动将多个参数提升为公共数据类型
anycompatiblearray	Common	表示函数接受任何数组数据类型，并自动将多个参数提升为公共数据类型
anycompatiblenonarray	Common	表示函数接受任何非数组数据类型，并自动将多个参数提升为公共数据类型
anycompatiblerange	Common	表示函数接受任何范围数据类型，并自动将多个参数提升为公共数据类型

多态参数和结果是相互关联的，并且它们在解析调用多态函数的查询时被决定到一种特定的数据类型。当存在多个多态参数时，输入值的实际数据类型必须匹配如下所述。如果函数的结果类型是多态的，或者它具有多态类型的输出参数，则这些结果的类型是根据下面所描述的多态输入的实际类型推导出来的。

对于多态类型的“simple”家族，匹配和推导规则是这样工作的：

每一个被声明为anyelement的位置（参数或返回值）被允许具有任意特定的实际数据类型，但是在任何给定的查询中它们必须全部是相同的实际类型。每一个被声明为anyarray的位置可以有任意数组数据类型，但是相似地，它们必须全部具有相同类型。并且类似地，被声明为anyrange的位置必须是全部是相同的范围类型。此外，如果有位置被声明为anyarray并且其他位置被声明为anyelement，anyarray位置中的实际数组类型必须是一个数组，该数组的元素都是出现在anyelement位置的同一种类型。相似地，如果有位置被声



明为`anyrange`并且其他位置被声明为`anyelement`或`anyarray`，`anyrange`位置的实际范围类型必须是一个范围，该范围的子类型是出现在`anyelement`位置的同一种类型，并且与`anyarray`位置的元素类型相同。`anynonarray`被当做和`anyelement`相同，但是增加了额外的约束要求实际类型不能是一种数组类型。`anyenum`被当做和`anyelement`相同，但是增加了额外的约束要求实际类型必须是一种枚举类型。

因此，当使用一种多态类型声明了多于一个参数位置，有效效果是只有实际参数类型的某些组合才被允许。例如，一个被声明为`equal(anyelement, anyelement)`的函数将要求任意两个输入值，只要它们是同一种数据类型。

当一个函数的返回值被声明为多态类型时，必须至少有一个参数位置也是多态的，并且为多态参数提供的实际数据类型确定该调用的实际结果类型。例如，如果还没有一种数组下标机制，我们可以定义一个函数来实现下标：`subscript(anyarray, integer) returns anyelement`。这个声明约束了实际的第一个参数是一种数组类型，并且允许解析器从实际的第一个参数类型推断正确的结果类型。另一个例子是一个被声明为`f(anyarray) returns anyenum`的函数将只接受枚举类型的数组。

在大多数情况下，解析器可以从同一家族中不同多态类型的参数推断出多态结果类型的实际数据类型；例如，`anyarray`可以从`anyelement`推断出来，反之亦然。一个例外是，类型`anyrange`的多态结果需要类型`anyrange`的参数；它不能从`anyarray`或`anyelement`参数中推断出来。这是因为可能有多个范围类型具有相同的子类型。

注意`anynonarray`和`anyenum`并不表示独立的类型变量，它们是和`anyelement`相同的类型，只是有一个额外的约束。例如，将一个函数声明为`f(anyelement, anyenum)`等效于把它声明为`f(anyenum, anyenum)`：两种实际参数必须是相同的枚举类型。

对于多态类型的“common”家族，匹配和推断规则工作与“simple”家族相同，有一个主要区别：参数的实际类型不需要相同，只要可以隐式地将它们转换为单一的公共类型即可。`common`类型的选择遵循与`UNION`和相关结构相同的规则（参见 Section 10.5）。`common`类型的选择考虑了`anycompatible`和`anycompatiblenonarray`输入的实际类型，`anycompatiblearray`输入的数组元素类型，以及`anycompatiblerange`输入中的范围子类型。如果存在`anycompatiblenonarray`，那么通用类型必须是非数组类型。一旦识别了`common`类型，`anycompatible`和`anycompatiblenonarray`位置中的参数将自动转换到该类型，并且`anycompatiblearray`位置的参数将自动转换到该类型的数组类型。

因为不可能选择只知道子类型的范围类型，使用`anycompatiblerange`需要使用该类型声明的所有参数具有相同的实际范围类型，这种类型的子类型与所选的公共类型一致，这样就不需要对范围值进行强制转换。与`anyrange`一样，使用`anycompatiblerange`作为函数结果类型需要有一个`anycompatiblerange`参数。

注意，没有`anycompatibleenum`类型。这样的类型不是很有用，因为通常没有任何隐式的枚举类型的强制转换，这意味着无法为不同的枚举输入解析通用类型。

“simple”和“common”多态家族代表两组独立的类型变量。考虑例如

```
CREATE FUNCTION myfunc(a anyelement, b anyelement,
                      c anycompatible, d anycompatible)
RETURNS anycompatible AS ...
```

在此函数的实际调用中，前两个输入必须具有完全相同的类型。最后两个输入必须提升为一个通用类型，但是这个类型不需要与前两个输入的类型有任何关系。结果将具有最后两个输入的共同类型。

一个可变函数（可以有可变数量的参数，如Section 36.5.5中所述）能够是多态的：这可以通过声明其最后一个参数为`VARIADIC anyarray`或`VARIADIC anycompatiblearray`来实现。为了匹配和决定实际结果类型的参数，这样一种函数的行为和写了合适数量的`anynonarray`或`anycompatiblenonarray`参数是一样的。

## 36.3. 用户定义的函数

LightDB提供四种函数：

- 查询语言函数（用SQL编写的函数）（Section 36.5
- 过程语言函数（例如，用PL/pgSQL或PL/Tcl编写的函数）（Section 36.8
- 内部函数（Section 36.9
- C 语言函数（Section 36.10

每一类函数可以采用基本类型、组合类型或者它们的组合作为参数。此外，每一类函数可以返回一个基本类型或一个组合类型。函数也能被定义成返回基本类型或组合类型值的集合。

很多类函数可以接受或者返回特定的伪类型（例如，多态类型），但是可用的功能会变化。详情可以参考每一种函数的描述。

定义SQL函数最容易，因此我们将从讨论SQL函数开始。大部分SQL函数的概念也能用到其他类型的函数上。

在这一章中，查看CREATE FUNCTION命令的参考页有助于更好地理解例子。本章中的一些例子可以在LightDB源代码发布的src/tutorial目录中的funcs.sql和funcs.c中找到。

## 36.4. 用户定义的过程

过程是一个类似于函数的数据库对象。关键差异是：

- 过程是由 CREATE PROCEDURE 命令定义，而不是 CREATE FUNCTION。
- 过程不返回函数值；因此CREATE PROCEDURE 缺少 RETURNS 子句。然而，过程可以通过输出参数返回数据到它们的调用器。
- 当函数可以作为查询或DML命令的一部分调用时，过程用CALL命令孤立的调用。
- 在执行期间过程可以提交或回滚事务(而后自动开始一个新事物)，只要调用CALL命令不是显式事务块的一部分。函数做不了这个。
- 某些函数属性，例如严格性，没有应用于过程。 这些属性控制如何在查询中使用函数，与过程不相关。

后面章节中关于如何定义用户定义的函数的解释同样适用于过程，除了上面指出的那些点。

函数和过程一起构成了例程。有ALTER ROUTINE 和 DROP ROUTINE这样的命令可以操作函数和过程而不需要知道它们是哪一种。不过，要注意没有CREATE ROUTINE命令。

## 36.5. 查询语言（SQL）函数

SQL 函数执行一个由任意 SQL 语句构成的列表，返回列表中最后一个查询的结果。在简单（非集合）的情况下，最后一个查询的结果的第一行将被返回（记住一个多行结果的“第一行”不是良定义的，除非你使用ORDER BY）。如果最后一个查询正好根本不返回行，将会返回空值。

或者，一个 SQL 函数可以通过指定函数的返回类型为SETOF sometype被声明为返回一个集合（也就是多个行），或者等效地声明它为RETURNS TABLE(columns)。在这种情况下，最后一个查询的结果的所有行会被返回。下文将给出进一步的细节。

一个 SQL 函数的主体必须是一个由分号分隔的 SQL 语句的列表。最后一个语句之后的分号是可选的。除非函数被声明为返回void，最后一个语句必须是一个SELECT或者一个带有RETURNING子句的INSERT、UPDATE或者DELETE。

SQL语言中的任何命令集合都能被打包在一起并且被定义成一个函数。除了SELECT查询，命令可以包括数据修改查询（INSERT、UPDATE以及DELETE）和其他 SQL 命令（你不能在SQL函

数中使用事务控制命令，例如COMMIT、SAVEPOINT，以及一些工具命令，例如VACUUM）。不过，最后一个命令必须是一个SELECT或者带有一个RETURNING子句，该命令必须返回符合函数返回类型的数据。或者，如果你想要定义一个执行动作但是不返回有用的值的函数，你可以把它定义为返回void。例如，这个函数从emp表中移除具有负值薪水的行：

```
CREATE FUNCTION clean_emp() RETURNS void AS '
DELETE FROM emp
WHERE salary < 0;
' LANGUAGE SQL;
```

```
SELECT clean_emp();
```

```
clean_emp
-----
```

```
(1 row)
```

### Note

在被执行前，SQL 函数的整个主体都要被解析。虽然 SQL 函数可以包含修改系统目录的命令（如CREATE TABLE），但这类命令的效果对于该函数中后续命令的解析分析不可见。例如，如果把CREATE TABLE foo (...); INSERT INTO foo VALUES (...);打包到一个 SQL 函数中是得不到预期效果的，因为在解析INSERT命令时foo还不存在。在这类情况下，推荐使用PL/pgSQL而不是 SQL 函数。

CREATE FUNCTION命令的语法要求函数体被写作一个字符串常量。使用用于字符串常量的美元引用通常最方便（见Section 4.1.2.4）。你过你选择使用常规的单引号引用的字符串常量语法，你必须在函数体中双写单引号（'）和反斜线（\）（假定转义字符串语法）（见Section 4.1.2.1）。

## 36.5.1. SQL函数的参数

一个 SQL 函数的参数可以在函数体中用名称或编号引用。下面会有两种方法的例子。

要使用一个名称，将函数参数声明为带有一个名称，然后在函数体中只写该名称。如果参数名称与函数内当前 SQL 命令中的任意列名相同，列名将优先。如果不想这样，可以用函数本身的名称来限定参数名，也就是function\_name.argument\_name（如果这会与一个被限定的列名冲突，照例还是列名赢得优先。你可以通过为 SQL 命令中的表选择一个不同的别名来避免这种混淆）。

在更旧的数字方法中，参数可以用语法\$n引用：\$1指的是第一个输入参数，\$2指的是第二个，以此类推。不管特定的参数是否使用名称声明，这种方法都有效。

如果一个参数是一种组合类型，那么点号记法（如 argname.fieldname 或\$1.fieldname）也可以被用来访问该参数的属性。同样，你可能需要用函数的名称来限定参数的名称以避免歧义。

SQL 函数参数只能被用做数据值而不能作为标识符。例如这是合理的：

```
INSERT INTO mytable VALUES ($1);
```

但这样就不行：

```
INSERT INTO $1 VALUES (42);
```

## Note

使用名称来引用 SQL 函数参数的能力是在LightDB 9.2 中加入的。要在老的服务器中使用的函数必须使用\$n记法。

## 36.5.2. 基本类型上的SQL

最简单的SQL函数没有参数并且简单地返回一个基本类型，例如integer：

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;
```

```
-- Alternative syntax for string literal:
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;
```

```
SELECT one();
```

```
one
-----
1
```

注意我们为该函数的结果在函数体内定义了一个列别名（名为result），但是这个列别名在函数以外是不可见的。因此，结果被标记为one而不是result。

定义用基本类型作为参数的SQL函数也很容易：

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;
```

```
SELECT add_em(1, 2) AS answer;
```

```
answer
-----
3
```

我们也能省掉参数的名称而使用数字：

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;
```

```
SELECT add_em(1, 2) AS answer;
```

```
answer
-----
3
```

这里是一个更有用的函数，它可以被用来借记一个银行账号：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
UPDATE bank
    SET balance = balance - debit
    WHERE accountno = tf1.accountno;
SELECT 1;
$$ LANGUAGE SQL;
```

一个用户可以这样执行这个函数来从账户 17 中借记 \$100.00:

```
SELECT tf1(17, 100.0);
```

在这个例子中，我们为第一个参数选择了名称accountno，但是这和表bank中的一个列名相同。在UPDATE命令中，accountno引用列bank.accountno，因此tf1.accountno必须被用来引用该参数。我们当然可以通过为该参数使用一个不同的名称来避免这样的问题。

实际上我们可能喜欢从该函数得到一个更有用的结果而不是一个常数 1，因此一个更可能的定义是：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
UPDATE bank
    SET balance = balance - debit
    WHERE accountno = tf1.accountno;
SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

它会调整余额并且返回新的余额。同样的事情也可以用一个使用RETURNING的命令实现：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
UPDATE bank
    SET balance = balance - debit
    WHERE accountno = tf1.accountno
RETURNING balance;
$$ LANGUAGE SQL;
```

如果SQL函数中的最后一个SELECT或RETURNING子句没有准确返回函数声明的结果类型，LightDB将自动将值转换为所需的类型，如果可以使用隐式或赋值转换。否则，您必须编写显式强制转换。例如，假设我们想要之前的add\_em函数返回类型float8。写就足够了

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
SELECT $1 + $2;
$$ LANGUAGE SQL;
```

因为integer总和可以隐式转换为float8。（有关强制转换的更多信息，请参阅Chapter 10或CREATE CAST。）

### 36.5.3. 组合类型上的SQL函数

在编写使用组合类型参数的函数时，我们必须不仅指定我们想要哪些参数，还要指定参数的期望属性（域）。例如，假定emp是一个包含雇员数据的表，并且因此它也是该表每一行的组合类型的名称。这里是一个函数double\_salary，它计算某个人的双倍薪水：

```
CREATE TABLE emp (
    name      text,
    salary    numeric,
    age       integer,
```

```

    cubicle    point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, ' (2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ^= point ' (2,1)';

```

```

name | dream
-----+-----
Bill | 8400

```

注意语法\$1.salary的使用是要选择参数行值的一个域。 还要注意调用的SELECT命令是如何使用table\_name.\*来选择一个表的整个当前行作为一个组合值的。该表行也可以只用表名来引用:

```

SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ^= point ' (2,1)';

```

但这种用法已被废弃因为它很容易让人搞混（关于表行的组合值的这两种记法的详细情况请见Section 8.16.5 。

有时候实时构建一个组合参数很方便。这可以用ROW结构完成。 例如，我们可以调整被传递给函数的数据:

```

SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;

```

也可以构建一个返回组合类型的函数。这是一个返回单一emp行的函数例子:

```

CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
           1000.0 AS salary,
           25 AS age,
           point ' (2,2)' AS cubicle;
$$ LANGUAGE SQL;

```

在这个例子中，我们为每一个属性指定了一个常量值，但是可以用任何计算来替换这些常量。

有关定义函数有两件重要的事情:

- 查询中的选择列表顺序必须与列在复合类型中出现的顺序完全相同。（正如我们上面所做的那样，命名列与系统无关。）
- 我们必须确保每个表达式的类型都可以转换为复合类型的相应列的类型。 否则我们会得到这样的错误:

```

ERROR: return type mismatch in function declared to return emp
DETAIL: Final statement returns text instead of point at column 4.

```

与基本类型的情况一样，系统不会自动插入显式转换，只会插入隐式或赋值转换。

定义同样的函数的一种不同的方法是：

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

这里我们写了一个只返回正确组合类型的单一列的SELECT。在这种情况下这种写法实际并非更好，但是它在一些情况下比较方便——例如，我们需要通过调用另一个返回所期望的组合值的函数来计算结果。另一个例子是，如果我们试图编写一个函数，它返回一个复合类型的域，而不是一个普通的复合类型，总是有必要把它写成返回单个列，因为没有办法导致整行结果。

我们可以直接调用这个函数或者在一个值表达式中使用它：

```
SELECT new_emp();

           new_emp
-----
(None, 1000.0, 25, "(2,2)")
```

或者把它当做一个表函数调用：

```
SELECT * FROM new_emp();

 name | salary | age | cubicle
-----+-----+-----+-----
None | 1000.0 | 25  | (2,2)
```

第二种方式在Section 36.5. 中有更完全的描述。

当你使用一个返回组合类型的函数时，你可能只想要其结果中的一个域（属性）。你可以这样做：

```
SELECT (new_emp()).name;

 name
-----
None
```

额外的圆括号是必须的，它用于避免解析器被搞混。如果你不写这些括号，会这样：

```
SELECT new_emp().name;
ERROR: syntax error at or near "."
LINE 1: SELECT new_emp().name;
                        ^
```

另一个选项是使用函数记号来抽取一个属性：

```
SELECT name(new_emp());

 name
```

```
-----
None
```

如Section 8.16.5中所说，字段记法和函数记法是等效的。

另一种使用返回组合类型的函数的方法是把结果传递给另一个接收正确行类型作为输入的函数：

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;
```

```
SELECT getname(new_emp());
    getname
```

```
-----
None
(1 row)
```

### 36.5.4. 带有输出参数的SQL函数

一种描述一个函数的结果的替代方法是定义它的输出参数，例如：

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;
```

```
SELECT add_em(3,7);
    add_em
```

```
-----
    10
(1 row)
```

这和Section 36.5.2中展示的add\_em版本没有本质上的不同。输出参数的真正价值是它们提供了一种方便的方法来定义返回多个列的函数。例如，

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;
```

```
SELECT * FROM sum_n_product(11,42);
    sum | product
```

```
-----+-----
    53 |    462
(1 row)
```

这里实际发生的是我们为该函数的结果创建了一个匿名的组合类型。上述例子具有与下面相同的最终结果

```
CREATE TYPE sum_prod AS (sum int, product int);
```

```
CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

但是不必单独定义组合类型常常很方便。注意输出参数的名称并非只是装饰，而且决定了匿名组合类型的列名（如果你为一个输出参数忽略了名称，系统将自行选择一个名称）。



在从 SQL 调用这样一个函数时，输出参数不会被包括在调用参数列表中。这是因为LightDB只考虑输入参数来定义函数的调用签名。这也意味着在为诸如删除函数等目的引用该函数时只有输入参数有关系。我们可以用下面的命令之一删除上述函数

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

参数可以被标记为IN（默认）、OUT、INOUT或者VARIADIC。一个INOUT参数既作为一个输入参数（调用参数列表的一部分）又作为一个输出参数（结果记录类型的一部分）。VARIADIC参数是输入参数，但被按照下文所述特殊对待。

### 36.5.5. 带有可变数量参数的SQL函数

只要“可选的”参数都是相同的数据类型，SQL函数可以被声明为接受可变数量的参数。可选的参数将被作为一个数组传递给该函数。声明该函数时要把最后一个参数标记为VARIADIC，这个参数必须被声明为一个数组类型，例如：

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

```
SELECT mleast(10, -1, 5, 4.4);
 mleast
-----
      -1
(1 row)
```

实际上，所有位于或者超过VARIADIC位置的实参会被收集成一个一位数组，就好像你写了：

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- 不起作用
```

但是你实际无法这样写 `SELECT mleast(ARRAY[10, -1, 5, 4.4]);` 或者说至少它将无法匹配这个函数定义。一个被标记为VARIADIC的参数匹配其元素类型的一次或者多次出现，而不是它自身类型的出现。

有时候能够传递一个已经构造好的数组给 `variadic` 函数是有用的，特别是当一个 `variadic` 函数想要把它的数组参数传递给另一个函数时这会特别方便。此外，这是在一个允许不可信用户创建对象的方案中调用一个variadic函数的唯一安全的方式，见Section 10.3 你可以通过在调用中指定VARIADIC来做到这一点：

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

这会阻止该函数的 `variadic` 参数扩展成它的元素结构，从而允许数组参数数值正常匹配。VARIADIC只能被附着在函数调用的最后一个实参上。

在调用中指定VARIADIC也是将空数组传递给 `variadic` 函数的唯一方式，例如：

```
SELECT mleast(VARIADIC ARRAY[]::numeric[]);
```

简单地写成SELECT mleast()是没有作用的，因为一个 `variadic` 参数必须匹配至少一个实参（如果想允许这类调用，你可以定义第二个没有参数且也叫mleast的函数）。

从一个 `variadic` 参数产生的数组元素参数会被当做自己不具有名称。这意味着不能使用命名参数调用 `variadic` 函数（Section 4.3，除非你指定了 VARIADIC。例如下面的调用是可以工作的：

```
SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);
```

但这些就不行：

```
SELECT mleast(arr => 10);
SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);
```

### 36.5.6. 带有参数默认值的SQL函数

函数可以被声明为对一些或者所有输入参数具有默认值。只要调用函数时没有给出足够的实参，就会插入默认值来弥补缺失的实参。由于参数只能从实参列表的尾部开始被省略，在一个有默认值的参数之后的所有参数都不得不也具有默认值（尽管使用命名参数记法可以允许放松这种限制，这种限制仍然会被强制以便位置参数记法能工作）。不管你是否使用它，这种能力都要求在某些用户不信任其他用户的数据中调用函数时做一些预防措施，见Section 10.3

例如：

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;
```

```
SELECT foo(10, 20, 30);
foo
-----
 60
(1 row)
```

```
SELECT foo(10, 20);
foo
-----
 33
(1 row)
```

```
SELECT foo(10);
foo
-----
 15
(1 row)
```

```
SELECT foo(); -- 因为第一个参数没有默认值，所以会失败
ERROR: function foo() does not exist
```

=符号也可以用来替代关键词 DEFAULT。

### 36.5.7. SQL 函数作为表来源

所有的 SQL 函数都可以被用在查询的FROM子句中，但是对于返回组合类型的函数特别有用。如果函数被定义为返回一种基本类型，该表函数会产生一个单列表。如果该函数被定义为返回一种组合类型，该表函数会为该组合类型的每一个属性产生一列。

这里是一个例子：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');
```

```
CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

```
SELECT *, upper(fooname) FROM getfoo(1) AS t1;
```

```
fooid | foosubid | fooname | upper
-----+-----+-----+-----
     1 |         1 | Joe     | JOE
(1 row)
```

正如例子所示，我们可以把函数结果的列当作常规表的列来使用。

注意我们只从函数得到了一行。这是因为我们没有使用SETOF。这会在下一节中介绍。

### 36.5.8. 返回集合的SQL函数

当一个 SQL 函数被声明为返回SETOF sometype时，该函数的 最后一个查询会被执行完，并且它输出的每一行都会被 作为结果集的一个元素返回。

在FROM子句中调用函数时通常会使用这种特性。在这种情况下，该函数返回的每一行都变成查询所见的表的一行。例如，假设 表foo具有和上文一样的内容，并且我们做了以下动作：

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM getfoo(1) AS t1;
```

那么我们会得到：

```
fooid | foosubid | fooname
-----+-----+-----
     1 |         1 | Joe
     1 |         2 | Ed
(2 rows)
```

也可以返回多个带有由输出参数定义的列的行，像这样：

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);
```

```
CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

```
SELECT * FROM sum_n_product_with_tab(10);
```

```

sum | product
-----+-----
 11 |      10
 13 |      30
 15 |      50
 17 |      70
(4 rows)

```

这里的关键点是必须写上RETURNS SETOF record来指示 该函数返回多行而不是一行。如果只有一个输出参数，则写上该参数的 类型而不是record。

通过多次调用集合返回函数来构建查询的结果非常有用，每次调用的参数 来自于一个表或者子查询的连续行。做这种事情最好的方法是使用 Section 7.2.1.5中描述的LATERAL关键词。这里是一个使用集合返回函数枚举树结构中元素的例子：

```

SELECT * FROM nodes;
  name | parent
-----+-----
  Top  |
 Child1 | Top
 Child2 | Top
 Child3 | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

```

```

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

```

```

SELECT * FROM listchildren('Top');
 listchildren
-----
 Child1
 Child2
 Child3
(3 rows)

```

```

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;
  name | child
-----+-----
  Top  | Child1
  Top  | Child2
  Top  | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)

```

这个例子和我们使用的简单连接的效果没什么不同，但是在更复杂的 计算中，把一些工作放在函数中会是一种很方便的选项。

返回集合的函数也能在查询的选择列表中调用。对于该查询本身产生的每一行都会调用集合返回函数，并且会从该函数的结果集中的每一个元素生成一个输出行。之前的例子也可以用这样的查询实现：

```

SELECT listchildren('Top');
 listchildren

```

```
-----
Child1
Child2
Child3
(3 rows)
```

```
SELECT name, listchildren(name) FROM nodes;
```

```
  name | listchildren
-----+-----
  Top  | Child1
  Top  | Child2
  Top  | Child3
Child1 | SubChild1
Child1 | SubChild2
(5 rows)
```

在最后一个SELECT中，注意对于Child2、Child3等没有出现输出行。这是因为listchildren 对这些参数返回空集，因此没有产生结果行。这和使用LATERAL 语法时，我们从与该函数结果的内连接得到的行为是一样的。

LightDB中，写在查询的选择列表中的集合返回函数的行为几乎和写在LATERAL FROM子句项中的集合返回函数完全一样。例如：

```
SELECT x, generate_series(1,5) AS g FROM tab;
```

几乎等效于

```
SELECT x, g FROM tab, LATERAL generate_series(1,5) AS g;
```

这会是完全一样的，除了在这个特别的例子中，规划器会选择把g放在嵌套循环连接的外侧，因为g对tab没有实际的横向依赖。那会导致一种不同的输出行顺序。选择列表中的集合返回函数总是会被计算，就好像它们在FROM子句剩余部分的嵌套循环连接的内侧一样，因此在考虑来自FROM子句的下一行之前，这些函数会运行到完成。

如果在查询的选择列表中有不止一个集合返回函数，则行为类似于把那些函数放到一个单一的LATERAL ROWS FROM( ... ) FROM子句项中的行为。对于来自底层查询的每一行，都有一个用到每个函数首个结果的输出行，然后是一个使用每个函数第二个结果的输出行，以此类推。如果某些集合返回函数产生的输出比其他函数少，会用空值代替缺失的数据，因此为一个底层行形成的总行数等于产生最多输出的集合返回函数的输出行数。因此集合返回函数会“步调一致”地运行直到它们的输出被耗尽，然后用下一个底层行继续执行。

集合返回函数可以被嵌套在一个选择列表中，不过在FROM子句项中不允许这样做。在这种情况下，嵌套的每一层会被单独对待，就像它是一个单独的LATERAL ROWS FROM( ... )项一样。例如，在

```
SELECT srf1(srf2(x), srf3(y)), srf4(srf5(z)) FROM tab;
```

中，集合返回函数srf2、srf3和srf5将为tab的每一行步调一致地运行，然后会对较低层的函数产生的每一行以步调一致的形式应用srf1和srf4。

在CASE或COALESCE这样的条件计算结构中，不能使用集合返回函数。例如，考虑

```
SELECT x, CASE WHEN x > 0 THEN generate_series(1, 5) ELSE 0 END FROM tab;
```

看起来这个语句应该产生满足x > 0的输入行的五次重复，以及不满足的行的一次重复。但实际上，由于在CASE表达时被计算前，generate\_series(1, 5)会被运行在一个隐式

的LATERAL FROM项中，它会为每个输入行产生五次重复。为了减少混乱，这类情况会产生一个解析时错误。

### Note

如果函数的最后一个命令是带有RETURNING的 INSERT、UPDATE或者 DELETE，该命令将总是会被执行完，即使函数没有用 SETOF定义或者调用查询不要求取出所有结果行也是如此。RETURNING子句产生的多余的行会被悄无声息地丢掉，但是在命令的目标表上的修改仍然会发生（而且在从该函数返回前就会全部完成）。

### Note

在LightDB 10之前，把多个集合返回函数放在同一个选择列表中的行为并不容易察觉，除非它们总是产生同等的行数。否则，你得到的输出行数将会是各集合返回函数产生的行数的最小公倍数。此外，嵌套的集合返回函数不会按照上述的方式工作。相反，一个集合返回函数只能有最多一个集合返回参数，集合返回函数的每一次嵌套会被独立运行。此外，条件执行（CASE等中的集合返回函数）以前是被允许的，但是会让事情更加复杂。在编写需要在较老的LightDB版本中工作的查询时，推荐使用LATERAL语法，因为这种语法能够在不同的版本间提供一致的结果。如果有一个依赖于集合返回函数的条件执行，那么可能可以通过把条件测试移到一个自定义集合返回函数中来修正该问题。例如，

```
SELECT x, CASE WHEN y > 0 THEN generate_series(1, z) ELSE 5 END FROM
tab;
```

可以变成

```
CREATE FUNCTION case_generate_series(cond bool, start int, fin int,
els int)
RETURNS SETOF int AS $$
BEGIN
IF cond THEN
RETURN QUERY SELECT generate_series(start, fin);
ELSE
RETURN QUERY SELECT els;
END IF;
END$$ LANGUAGE plpgsql;
```

```
SELECT x, case_generate_series(y > 0, 1, z, 5) FROM tab;
```

这种表达形式将在所有版本的LightDB中以相同的方式工作。

## 36.5.9. 返回TABLE的SQL函数

还有另一种方法可以把函数声明为返回一个集合，即使用 RETURNS TABLE(columns)语法。这等效于使用一个或者多个OUT参数外加把函数标记为返回 SETOF record（或者是SETOF单个输出参数的类型）。这种写法是在最近的 SQL 标准中指定的，因此可能比使用 SETOF的移植性更好。

例如，前面的求和并且相乘的例子也可以这样做：

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

不允许把显式的OUT或者INOUT参数用于 RETURNS TABLE记法 — 必须把所有输出列放在TABLE列表中。

### 36.5.10. 多态SQL函数

SQL函数可以声明为接受和返回Section 36.2.5中描述的多态类型。这是一个多态函数make\_array，它从两个任意数据类型元素构建一个数组：

```
CREATE FUNCTION make_array(anelement, anelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
```

```
SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
intarray | textarray
-----+-----
{1,2}   | {a,b}
(1 row)
```

注意类型造型'a'::text的使用是为了指定该参数的类型是text。如果该参数只是一个字符串这就是必须的，因为否则它会被当作unknown类型，并且unknown的数组也不是一种合法的类型。如果没有改类型造型，将得到这样的错误：

```
ERROR: could not determine polymorphic type because input has type unknown
```

使用上述声明的make\_array，您必须提供两个数据类型完全相同的参数；系统不会尝试解决任何类型差异。因此，例如，这并不工作：

```
SELECT make_array(1, 2.5) AS numericarray;
ERROR: function make_array(integer, numeric) does not exist
```

另一种方法是使用“common”多态类型系列，它允许系统尝试识别合适的公共类型：

```
CREATE FUNCTION make_array2(anycompatible, anycompatible)
RETURNS anycompatiblearray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
```

```
SELECT make_array2(1, 2.5) AS numericarray;
numericarray
-----
{1,2.5}
(1 row)
```

因为当所有输入都是未知类型时，通用类型解析规则默认选择类型text，这也适用：

```
SELECT make_array2('a', 'b') AS textarray;
textarray
-----
```

```
{a, b}
(1 row)
```

允许具有多态参数和固定的返回类型，但是反过来不行。例如：

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;
```

```
SELECT is_greater(1, 2);
 is_greater
-----
f
(1 row)
```

```
CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR: cannot determine result data type
DETAIL: A result of type anyelement requires at least one input of type
anyelement, anyarray, anynonarray, anyenum, or anyrange.
```

多态化可以用在具有输出参数的函数上。例如：

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;
```

```
SELECT * FROM dup(22);
 f2 | f3
----+-----
 22 | {22,22}
(1 row)
```

多态化也可以用在 variadic 函数上。例如：

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

```
SELECT anyleast(10, -1, 5, 4);
 anyleast
-----
      -1
(1 row)
```

```
SELECT anyleast('abc'::text, 'def');
 anyleast
-----
abc
(1 row)
```

```
CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;
```

```
SELECT concat_values('|', 1, 4, 2);
```



```
concat_values
-----
1|4|2
(1 row)
```

## 36.5.11. 带有排序规则的SQL函数

当一个 SQL 函数具有一个或者更多可排序数据类型的参数时，按照 Section 21.2 所述，对每一次函数调用都会根据分配给实参的排序规则为其确定一个排序规则。如果成功地确定（即在参数之间没有隐式排序规则的冲突），那么所有的可排序参数都被认为隐式地具有该排序规则。这将会影响函数中对排序敏感的操作的行为。例如，使用上述的 `anyleast` 函数时，

```
SELECT anyleast('abc'::text, 'ABC');
```

的结果将依赖于数据库的默认排序规则。在 C 区域中，结果将是 ABC，但是在很多其他区域中它将是 abc。可以在任意参数上增加一个 `COLLATE` 子句来强制要使用的排序规则，例如：

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

此外，如果你希望一个函数用一个特定的排序规则工作而不管用什么排序规则调用它，可以根据需要在函数定义中插入 `COLLATE` 子句。这种版本的 `anyleast` 将总是使用 `en_US` 区域来比较字符串：

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

但是注意如果应用到不可排序数据类型上，这将会抛出一个错误。

如果在实参之间无法确定共同的排序规则，那么 SQL 函数会把它的参数当作拥有其数据类型的默认排序规则（通常是数据库的默认排序规则，但是域类型的参数可能会不同）。

可排序参数的行为可以被想成是多态的一种受限形式，只对于文本数据类型有效。

## 36.6. 函数重载

可以用同样的 SQL 名称定义多于一个函数，只要它们的参数不同即可。换句话说，函数名可以被重载。不管你是否使用它，这种能力都要求在某些用户不信任其他用户的数据中调用函数时做一些预防措施，见 Section 10.3 当一个查询被执行时，服务器将从数据类型和所提供的参数个数来决定要调用哪个函数。重载也可用来模拟具有可变参数个数（最大个数有限）的函数。

在创建一个重载函数家族时，应该小心不要创建歧义。例如，给定函数：

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

对于 `test(1, 1.5)` 这样的输入就无法立刻清楚地知道应该调用哪个函数。当前实现的解决规则在 Chapter 10 中有描述，但是设计一个依赖于这种行为的系统是不明智的。

一个具有单个组合类型参数的函数通常不应与该类型的任何属性（域）重名。回想一下，`attribute(table)` 被认为等效于 `table.attribute`。在出现“一个组合类型上的函数”与“组合类型的一个属性”的情况下，将总是使用属性。可以通过用模式限定该函数

名（即 `schema.func(table)`）来覆盖这种选择，但是最好 不要选择有冲突的名称以避免此类问题。

另一种可能的冲突在于 `variadic` 和非 `variadic` 函数之间。例如，可以创建 `foo(numeric)` 和 `foo(VARIADIC numeric[])`。在这种情况下，对于提供了一个数字参数的调用（例如 `foo(10.1)`）就不清楚应该匹配哪一个函数。规则是使用在搜索路径中出现得较早的函数，或者当两者都在同一个模式中时优先使用非 `variadic` 的那一个函数。

在重载 C 语言函数时有一个额外的约束：重载函数家族中的每一个函数的 C 名称必须与其他所有函数的 C 名称不同，不管是内部的 还是动态载入的。如果这条规则被违背，该行为将不可移植。你可能会得到一个运行时链接器错误，或者这些函数之一将被调用（通常是内部的那一个）。SQL `CREATE FUNCTION` 命令的 AS 子句的另一种形式 可以把 SQL 函数名和 C 源代码中的函数名分离。例如：

```
CREATE FUNCTION test(int) RETURNS int
  AS 'filename', 'test_larg'
  LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
  AS 'filename', 'test_2arg'
  LANGUAGE C;
```

这里的 C 函数名称反映了很多种可能的习惯之一。

## 36.7. 函数易变性分类

每一个函数都有一个易变性分类，可能是 `VOLATILE`、`STABLE` 或者 `IMMUTABLE`。如果 `CREATE FUNCTION` 命令没有指定一个分类，则默认是 `VOLATILE`。易变性分类是给优化器的关于该函数行为的一种承诺：

- 一个 `VOLATILE` 函数可以做任何事情，包括修改数据库。在 使用相同的参数连续调用时，它能返回不同的结果。优化器不会对这类函数的行为做任何假定。在每一行需要 `volatile` 函数值时，一个使用 `volatile` 函数的查询都会重新计算该函数。
- 一个 `STABLE` 函数不能修改数据库并且被确保对一个语句中 的所有行用给定的相同参数返回相同的结果。这种分类允许优化器把该函数的多个调用优化成一个调用。特别是，在一个索引扫描条件中使用包含 这样一个函数的表达式是安全的（因为一次索引扫描只会计算一次比较值， 而不是为每一行都计算一次，在一个索引扫描条件中不能使用 `VOLATILE` 函数）。
- 一个 `IMMUTABLE` 函数不能修改数据库并且被确保用相同的参数 永远返回相同的结果。这种分类允许优化器在一个查询用常量参数调用该函数 时提前计算该函数。例如，一个 `SELECT ... WHERE x = 2 + 2` 这样的查询可以被简化为 `SELECT ... WHERE x = 4`，因为整数加法操作符底层的函数被 标记为 `IMMUTABLE`。

为了最好的优化结果，你应该把函数标记为对它们合法的易变性分类中最严格 的那种。

任何带有副作用的函数必须被标记为 `VOLATILE`， 这样对它的调用就不能被优化掉。甚至如果一个函数的值在一个查询中会 变化，即使它没有副作用也需要被标记为 `VOLATILE`。这样的例子有 `random()`、`currval()`、`timeofday()` 等。

另一种重要的例子是 `current_timestamp` 家族的函数有资格 被标记为 `STABLE`，因为它们的值在一个事务中不会改变。

在考虑先规划然后立即执行的简单交互式查询时，在 `STABLE` 和 `IMMUTABLE` 分类间的区别相对较小：一个函数是在规划时只 执行一次还是在查询执行开始期间只执行一次没有太大关系。但是如果计划 被保存下来然后在后面被重用，区别就大了。如果在不允许过早把一个函数 变成规划期间的一个常数时把它标记为 `IMMUTABLE`，会导致 在后续重用该计划时用到一个陈旧的值。当使用预备语句或者使用会缓存计 划的函数语言（PL/pgSQL）时，这就会是一种灾难。

对于用 SQL 或者其他任何标准过程语言编写的函数，还有第二种由易变性分类 决定的特性，即由调用该函数的 SQL 命令所作的数据库修改的可见性。VOLATILE函数将看到这些更改，STABLE 或者IMMUTABLE函数则看不到。这种行为使用 MVCC 的快照 行为（见Chapter 13 实现：STABLE和 IMMUTABLE函数使用一个在调用查询开始时建立的快照，而 VOLATILE函数在它们执行的每一个查询的开始都获得一个新鲜 的快照。

### Note

用 C 编写的函数按照它们自己需要的方式管理快照，但是通常最好 让 C 函数也按照上面的方式来。

由于这种快照行为，一个只包含SELECT命令的函数可以被 安全地标记为STABLE，即便它选择的表可能正在被并发查询所 修改。LightDB将使用为调用查询所 建立的快照来执行STABLE函数中的所有命令，因此它将在整个 查询期间看到一种数据库的固定视图。

对IMMUTABLE函数中的SELECT使用了相同 的快照行为。通常在一个IMMUTABLE函数中从数据库表选择是 不明智的，因为如果表内容变化就会破坏不变性。不过，LightDB不会强制不让你这样做。

一种常见的错误是当一个函数的结果依赖于一个配置参数时把它标记为 IMMUTABLE。例如，一个操纵时间戳的函数有可能结果 依赖于TimeZone设置。为了安全起见，这类 函数应该被标记为STABLE。

### Note

LightDB要求STABLE 和IMMUTABLE函数中不包含非SELECT 的 SQL 命令以阻止数据库修改（这也不是完全万无一失，因为这类函数还可以 调用修改数据库的VOLATILE函数。如果那样做，你将发现 该STABLE或IMMUTABLE函数不会发现由被调 用函数所作的数据库改变，因为它们对它的快照不可见）。

## 36.8. 过程语言函数

LightDB允许用除 SQL 和 C 之外 的语言编写用户定义的函数。这些语言通常被称为过程语言 (PL)。过程语言并不内建在LightDB服务器中，它们通过可装载模块提供。更多信息请见Chapter 40以及接下来的 章节。

## 36.9. 内部函数

内部函数由 C 编写并且已经被静态链接到LightDB 服务器中。该函数定义的“主体”指定该函数的 C 语言名称，它不必和声明 SQL 函数所用的名称一样（为了向后兼容性的原因，也接受空 主体，那时会认为 C 语言函数名与 SQL 函数名相同）。

通常，所有存在于服务器中的内部函数都在数据库集簇的初始化（见 Section 16.2 期间被声明，但是用户可以使用 CREATE FUNCTION为一个内部函数创建 额外的别名。在CREATE FUNCTION中用 语言名internal来声明内部函数。例如，要为 sqrt函数创建一个别名：

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
AS 'dsqrt'
LANGUAGE internal
STRICT;
```

（大部分内部函数应该被声明为“严格”）。

## Note

上述场景中并非所有“预定义”的函数都是“内部”函数。有些预定义的函数由 SQL 编写。

## 36.10. C 语言函数

用户定义的函数可以用 C 编写（或者可以与 C 兼容的语言，例如 C++）。这类函数被编译成动态载入对象（也被称为共享库）并且由服务器在需要时载入。动态载入是把“C语言”函数和“内部”函数区分开的特性——两者真正的编码习惯实际上是一样的（因此，标准的内部函数库是用户定义的 C 函数很好的源代码实例）。

当前仅有一种调用约定被用于 C 函数（“版本1”）。如下文所示，为函数编写一个 `PG_FUNCTION_INFO_V1()` 宏就能指示对该调用约定的支持。

### 36.10.1. 动态载入

在一个会话中第一次调用一个特定可载入对象文件中的用户定义函数时，动态载入器会把那个对象文件载入到内存以便该函数被调用。因此用户定义的 C 函数的 `CREATE FUNCTION` 必须为该函数指定两块信息：可载入对象文件的名称，以及要在该对象文件中调用的特定函数的 C 名称（链接符号）。如果没有显式指定 C 名称，则它被假定为和 SQL 函数名相同。

下面的算法被用来基于 `CREATE FUNCTION` 命令中给定的名称来定位共享对象文件：

1. 如果名称是一个绝对路径，则载入给定的文件。
2. 如果该名称以字符串 `$libdir` 开始，那么这一部分会被 LightDB 包的库目录名（在编译时确定）替换。
3. 如果该名称不包含目录部分，会在配置变量 `dynamic_library_path` 指定的路径中搜索该文件。
4. 否则（在该路径中没找到该文件，或者它包含一个非绝对目录），动态载入器将尝试接受给定的名称，这大部分会导致失败（依赖当前工作目录是不可靠的）。

如果这个序列不起作用，会把平台相关的共享库文件名扩展（通常是 `.so`）追加到给定的名称并且再次尝试上述的过程。如果还是失败，则载入失败。

我们推荐相对于 `$libdir` 或者通过动态库路径来定位共享库。如果升级版本时新的安装在一个不同的位置，则可以简化升级过程。`$libdir` 实际表示的目录可以用命令 `lt_config --pkglibdir` 来找到。

用于运行 LightDB 服务器的用户 ID 必须能够通过要载入文件的路径。常见的错误是把文件或更高层的目录变得对 `postgres` 用户不可读或者不可执行。

在任何情况下，`CREATE FUNCTION` 命令中给定的文件名会被原封不动地记录在系统目录中，这样如果需要再次载入该文件则会应用同样的过程。

## Note

LightDB 不会自动编译 C 函数。在从 `CREATE FUNCTION` 命令中引用对象文件之前，它必须先被编译好。更多信息请见 Section 36.10.5

为了确保动态载入对象文件不会被载入到一个不兼容的服务器，LightDB 会检查该文件是否包含一个带有合适内容的“magic block”。这允许服务器检测到明显的不兼容，例如为不

同LightDB主版本编译的代码。要包括一个 magic block，在写上包括头文件fmgr.h的语句之后，在该模块的源文件之一（并且只能在其中一个）中写上这些：

```
PG_MODULE_MAGIC;
```

在被第一次使用后，动态载入对象文件会留在内存中。在同一个会话中 对该函数未来的调用将只会消耗很小的负荷进行符号表查找。如果需要 重新载入一个对象文件（例如重新编译以后），需要开始一个新的会话。

可以选择让一个动态载入文件包含初始化和终止化函数。如果文件包含一个 名为 `_PG_init` 的函数，则文件被载入后会立刻调用该函数。 该函数不接受参数并且应该返回 `void`。如果文件包括一个名为 `_PG_fini` 的函数，则在卸载该文件之前会立即调用该函数。同样地，该函数不接受参数并且应该返回 `void`。注意将只在卸载文件的过程中会调用 `_PG_fini`，进程结束时不会调用它（当前，卸载被 禁用并且从不发生，但是未来可能会改变）。

## 36.10.2. C 语言函数中的基本类型

要了解如何编写 C 语言函数，你需要了解 LightDB如何在内部表达基本数据类型 以及如何与函数传递它们。在内部，LightDB把一个基本类型认为是“一团内存”。在类型上定义的用户定义函数说明了 LightDB在该类型上操作的方式。也就是说，LightDB将只负责把数据存在磁盘以及从磁盘检索数据，而使用你的用户定义函数来输入、处理和输出该数据。

基本类型可以有三种内部格式之一：

- 传值，定长
- 传引用，定长
- 串引用，变长

传值类型在长度上只能是 1、2 或 4 字节（如果你的机器上 `sizeof(Datum)` 是 8，则还有 8 字节）。你应当小心地 定义你的类型以便它们在所有的架构上都是相同的尺寸（字节）。例如，`long` 类型很危险，因为它在某些机器上是 4 字节但在 另外一些机器上是 8 字节，而 `int` 类型在大部分 Unix 机器 上都是 4 字节。在 Unix 机器上 `int4` 类型一种合理的实现可能是：

```
/* 4 字节整数，传值 */
typedef int int4;
```

（实际的 LightDB C 代码会把这种类型称为 `int32`，因为 C 中的习惯是 `intXX` 表示 XX 位。注意 因此还有尺寸为 1 字节的 C 类型 `int8`。SQL 类型 `int8` 在 C 中被称为 `int64`。另见 Table 36.2 。

在另一方面，任何尺寸的定长类型可以用传引用的方法传递。例如，这里有一种 LightDB类型的实现示例：

```
/* 16 字节结构，传引用 */
typedef struct
{
    double x, y;
} Point;
```

在LightDB函数中传进或传出这种 类型时，只能使用指向这种类型的指针。要返回这样一种类型的值，用 `palloc`分配正确的内存量，然后填充分配好的内存， 并且返回一个指向该内存的指针（还有，如果只想返回与具有相同数据类型的 一个输入参数相同的值，可以跳过额外的 `palloc` 并且返回 指向该输入值的指针）。

最后，所有变长类型必须也以引用的方式传递。所有变长类型必须用一个正好 4 字节的不透明长度域开始，该域会由 SET\_VARSIZE 设置，绝不要直接设置该域！所有要被存储在该类型中的数据必须在内存中接着该长度域的后面存储。长度域包含该结构的总长度，也就是包括长度域本身的尺寸。

另一个重点是要避免在数据类型值中留下未被初始化的位。例如，要注意把可能存在于结构中的任何对齐填充字节置零。如果不这样做，你的数据类型逻辑等价常量可能会被规划器认为是不等的，进而导致低效的（不过还是正确的）计划。

**Warning**

绝不要修改通过引用传递的输入值的内容。如果这样做很可能会破坏磁盘上的数据，因为给出的指针可能直接指向一个磁盘缓冲区。这条规则唯一的例外在 Section 36.1 中有解释。

例如，我们可以这样定义类型 text：

```
typedef struct {
    int32 length;
    char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

[FLEXIBLE\_ARRAY\_MEMBER] 记号表示数据部分的实际长度不由该声明指定。

在操纵变长字节时，我们必须小心地分配正确数量的内存并且正确地设置长度域。例如，如果我们想在一个 text 结构中存储 40 字节，我们可以使用这样的代码片段：

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...

```

VARHDRSZ 和 sizeof(int32) 一样，但是用宏 VARHDRSZ 来引用变长类型的载荷的尺寸被认为是比较好的风格。还有，必须使用 SET\_VARSIZE 宏来设置长度域，而不是用简单的赋值来设置。

Table 36. 指定在编写使用一种 LightDB 内建类型的 C 语言函数时，哪一种 C 类型对应于哪一种 SQL 类型。“定义文件”列给出了要得到该类型定义需要包括的头文件（实际的定义可能在一个由列举文件包括的不同文件中。推荐用户坚持使用已定义的接口）。注意在任何源文件中应该总是首先包括 postgres.h，因为它声明了很多你需要的东西。

Table 36.2. 内建 SQL 类型等效的 C 类型

SQL 类型	C 类型	定义文件
boolean	bool	postgres.h (可能是编译器内建)
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(编译器内建)

SQL 类型	C 类型	定义文件
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
float4 (real)	float4	postgres.h
float8 (double precision)	float8	postgres.h
int2 (smallint)	int16	postgres.h
int4 (integer)	int32	postgres.h
int8 (bigint)	int64	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
numeric	Numeric	utils/numeric.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	RegProcedure	postgres.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp	datatype/timestamp.h
timestamp with time zone	TimestampTz	datatype/timestamp.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

现在我们已经复习了基本类型所有可能的结构，现在可以展示一些 真实函数的例子了。

### 36.10.3. 版本 1 的调用约定

版本-1 的调用规范依赖于宏来降低传参数和结果的复杂度。版本-1 函数的 C 声明总是：

```
Datum funcname(PG_FUNCTION_ARGS)
```

此外，宏调用：

```
PG_FUNCTION_INFO_V1(funcname);
```

必须出现在同一个源文件中（按惯例会正好写在该函数本身之前）。这种宏调用不是 internal 语言函数所需要的，因为 LightDB 会假定所有内部函数都使用 版本-1 规范。不过，对于动态载入函数是必需的。

在版本-1 函数中，每一个实参都使用对应于该参数数据类型的 PG\_GETARG\_xxx() 宏取得。（在非严格的函数中，需要使用 PG\_ARGISNULL() 对参数是否为空提前做检查；见下文。）结

果要用对应于返回类型的PG\_RETURN\_xxx()宏返回。PG\_GETARG\_xxx()的参数是要取得的函数参数的编号，从零开始计。PG\_RETURN\_xxx()的参数是实际要返回的值。

这里是一些使用版本-1调用约定的例子：

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

PG_MODULE_MAGIC;

/* 传值 */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32    arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* 传引用，定长 */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* FLOAT8 的宏隐藏了它的传引用本质。 */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* 这里，Point 的传引用本质没有被掩盖。 */
    Point    *pointx = PG_GETARG_POINT_P(0);
    Point    *pointy = PG_GETARG_POINT_P(1);
    Point    *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* 传引用，变长 */

PG_FUNCTION_INFO_V1(copytext);
```



```

Datum
copytext(PG_FUNCTION_ARGS)
{
    text      *t = PG_GETARG_TEXT_PP(0);

    /*
     * VARSIZE_ANY_EXHDR是该结构的尺寸（以字节为单位）减去其头部的
     * VARHDRSZ或VARHDRSZ_SHORT。用一个完整长度的头部构建该拷贝。
     */
    text      *new_t = (text *) palloc(VARSIZE_ANY_EXHDR(t) + VARHDRSZ);
    SET_VARSIZE(new_t, VARSIZE_ANY_EXHDR(t) + VARHDRSZ);

    /*
     * VARDATA是指向新结构的数据区域的指针。来源可以是一个短数据，
     * 所以要通过VARDATA_ANY检索它的数据。
     */
    memcpy((void *) VARDATA(new_t), /* 目标 */
           (void *) VARDATA_ANY(t), /* 源头 */
           VARSIZE_ANY_EXHDR(t)); /* 多少字节 */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

```

```

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_PP(0);
    text *arg2 = PG_GETARG_TEXT_PP(1);
    int32 arg1_size = VARSIZE_ANY_EXHDR(arg1);
    int32 arg2_size = VARSIZE_ANY_EXHDR(arg2);
    int32 new_text_size = arg1_size + arg2_size + VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA_ANY(arg1), arg1_size);
    memcpy(VARDATA(new_text) + arg1_size, VARDATA_ANY(arg2), arg2_size);
    PG_RETURN_TEXT_P(new_text);
}

```

假定上述代码已经准备在文件funcs.c中并且被编译成一个共享对象，我们可以用这样的命令在LightDB中定义函数：

```

CREATE FUNCTION add_one(integer) RETURNS integer
AS 'DIRECTORY/funcs', 'add_one'
LANGUAGE C STRICT;

```

-- 注意SQL函数名“add\_one”的重载

```

CREATE FUNCTION add_one(double precision) RETURNS double precision
AS 'DIRECTORY/funcs', 'add_one_float8'
LANGUAGE C STRICT;

```

```

CREATE FUNCTION makepoint(point, point) RETURNS point
AS 'DIRECTORY/funcs', 'makepoint'
LANGUAGE C STRICT;

```

```
CREATE FUNCTION copytext(text) RETURNS text
AS 'DIRECTORY/funcs', 'copytext'
LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
AS 'DIRECTORY/funcs', 'concat_text'
LANGUAGE C STRICT;
```

这里，DIRECTORY表示共享库文件的目录（例如LightDB的教程目录，它包含这一节中用到的例子的代码）。（更好的风格是先把DIRECTORY放入搜索路径，在AS子句中只使用'funcs'。在任何情况下，我们可以为一个共享库省略系统相关的扩展名，通常是.so）。

注意我们已经把函数指定为“strict”，这意味着如果有任何输入值为空，系统应该自动假定得到空结果。通过这种做法，我们避免在函数代码中检查空值输入。如果不这样做，我们必须使用PG\_ARGISNULL()明确地检查空值输入。

宏PG\_ARGISNULL(n)允许一个函数测试是否每一个输入为空（当然，只需要在没有声明为“strict”的函数中这样做）。和PG\_GETARG\_xxx()宏一样，输入参数也是从零开始计数。注意应该在验证了一个参数不是空之后才执行PG\_GETARG\_xxx()。要返回一个空结果，应执行PG\_RETURN\_NULL()，它对严格的以及非严格的函数都有用。

乍一看，与使用普通的C调用约定相比，版本1编码约定似乎只是毫无意义的愚民政策。然而，它们确实允许我们处理NULLable 参数/返回值，以及“toasted”（压缩或离线）值。

在版本-1接口中提供的其他选项是PG\_GETARG\_xxx()宏的两个变种。其中的第一种是PG\_GETARG\_xxx\_COPY()，它确保返回的指定参数的拷贝可以被安全地写入（通常的宏有时会返回一个指向表中物理存储的值，它不能被写入。使用PG\_GETARG\_xxx\_COPY()宏可以保证得到一个可写的结果）。第二种变种PG\_GETARG\_xxx\_SLICE()宏有三个参数。第一个是函数参数的编号（如上文）。第二个和第三个是要被返回的段的偏移量和长度。偏移量从零开始计算，而负值的长度则表示要求返回该值的剩余部分。当大型值的存储类型为“external”时，这些宏提供了访问这些大型值的更有效的方法（列的存储类型可以使用ALTER TABLE tablename ALTER COLUMN colname SET STORAGE storagetype来指定。storagetype取plain、external、extended或者main）。

最后，版本-1 的函数调用规范可以返回集合结果（Section 36.10.8、实现触发器函数（Chapter 37 和过程语言调用处理器（Chapter 52。更多细节可见源代码发布中的src/backend/utils/fmgr/README）。

## 36.10.4. 编写代码

在开始更高级的话题之前，我们应该讨论一下用于 LightDB C 语言函数的编码规则。虽然可以把不是 C 编写的函数载入到 LightDB中，这通常是很困难的，因为其他语言（例如 C++、FORTRAN 或者 Pascal）通常不会遵循和 C 相同的调用规范。也就是说，其他语言不会以同样的方式在函数之间传递参数以及返回值。由于这个原因，我们会假定你的 C 语言函数确实是用 C 编写的。

编写和编译 C 函数的基本规则如下：

- 使用lt\_config --includedir-server 找出LightDB服务器头文件安装在 系统的哪个位置。
- 编译并且链接你的代码（这样它就能被动态载入到 LightDB中）总是 要求特殊的标志。对特定的操作系统的做法详见 Section 36.10.5
- 记住为你的共享库按Section 36.10.4中所述 定义一个“magic block”。
- 在分配内存时，使用 LightDB函数 palloc和 pfree，而不是使用对应的 C 库函数 malloc和free。在每个事务结束时会自动释放通过palloc 分配的内存，以免内存泄露。
- 总是要使用memset把你的结构中的字节置零（或者 最开始就用palloc0分配它们）。即使你对结构中的 每个域都赋值，也可能有对齐填充（结构中的空洞）包含着垃圾值。如果

不这样做，很难支持哈希索引或哈希连接，因为你必须选出数据结构中有意义的位进行哈希计算。规划器有时也依赖于用按位相等来比较常量，因此如果逻辑等价的值不是按位相等的会导致出现不想要的规划结果。

- 大部分的内部LightDB类型都声明在postgres.h中，不过函数管理器接口（PG\_FUNCTION\_ARGS等）在fmgr.h中，因此你将需要包括至少这两个文件。为了移植性，最好在包括任何其他系统或者用户头文件之前，先包括postgres.h。包括postgres.h也将会为你包括elog.h和palloc.h。
- 对象文件中定义的符号名不能相互冲突或者与LightDB服务器可执行程序中定义的符号冲突。如果出现有关于此的错误消息，你将必须重命名你的函数或者变量。

## 36.10.5. 编译和链接动态载入的函数

在使用C编写的LightDB扩展函数之前，必须以一种特殊的方式编译并且链接它们，以便产生一个能被服务器动态载入的文件。简而言之，需要创建一个共享库。

超出本节所含内容之外的信息请参考你的操作系统文档，特别是C编译器（cc）和链接编辑器（ld）的手册页。另外，LightDB源代码的contrib目录中包含了一些可以工作的例子。但是，如果你依靠这些例子，也会使你的模块依赖于LightDB源码的可用性。

创建共享库通常与链接可执行文件相似：首先源文件被编译成对象文件，然后对象文件被链接起来。对象文件需要被创建为独立位置代码（PIC），，这在概念上意味着当它们被可执行文件载入时，它们可以被放置在内存中的任意位置（用于可执行文件的对象文件通常不会以那种方式编译）。链接一个共享库的命令会包含特殊标志来把它与链接一个可执行文件区分开（至少在理论上——在某些系统上实际上很丑陋）。

在下列例子中，我们假定你的源代码在一个文件foo.c中，并且我们将创建一个共享库foo.so。除非特别注明，中间的对象文件将被称为foo.o。一个共享库能包含多于一个对象文件，但是我们在这里只使用一个。

### FreeBSD

用来创建PIC的编译器标志是-fPIC。要创建共享库，编译器标志是-shared。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

这适用于FreeBSD从3.0开始的版本。

### HP-UX

该系统编译器创建PIC的编译器标志是+z。当使用GCC自己的-fPIC时。用于共享库的链接器标志是-b。因此：

```
cc +z -c foo.c
```

或者：

```
gcc -fPIC -c foo.c
```

并且然后：

```
ld -b -o foo.sl foo.o
```

和大部分其他系统不同，HP-UX为共享库使用扩展.sl。

## Linux

创建PIC的编译器标志是`-fpic`。创建一个共享库的编译器标志是`-shared`。一个完整的例子看起来像：

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

## macOS

这里是一个例子。它假定安装了开发者工具。

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

## NetBSD

创建PIC的编译器标志是`-fPIC`。对于ELF系统，带着标志`-shared`的编译器被用来链接共享库。在旧的非ELF系统上，`ld -Bshareable`会被使用。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

## OpenBSD

创建PIC的编译器标志是`-fPIC`。`ld -Bshareable`被用来链接共享库。

```
gcc -fPIC -c foo.c
ld -Bshareable -o foo.so foo.o
```

## Solaris

创建PIC的编译器标志是`-KPIC`（用于Sun编译器）以及`-fPIC`（用于GCC）。要链接共享库，编译器选项对几种编译器都是`-G`或者是对GCC使用`-shared`。

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fPIC -c foo.c
gcc -G -o foo.so foo.o
```

**Tip**

如果这对你来说太复杂，你应该考虑使用 GNU Libtool<sup>1</sup>，它会用一个统一的接口隐藏平台差异。

结果的共享库文件接着就可以被载入到LightDB。当把文件名指定给`CREATE FUNCTION`命令时，必须把共享库文件的名称给它，而不是中间的对象文件。注意系统的标准共享库扩展

<sup>1</sup> <https://www.gnu.org/software/libtool/>

(通常是 .so 或者 .sl) 在 CREATE FUNCTION 命令中可以被忽略, 并且通常为了最好的可移植性应该被忽略。

服务器会期望在哪里寻找共享库文件请参考 Section 36.10.1

## 36.10.6. 组合类型参数

组合类型没有像 C 结构那样的固定布局。组合类型的实例可能包含空值域。此外, 继承层次中的组合类型可能具有和同一继承层次中其他成员不同的域。因此, LightDB 提供了函数接口来访问 C 的组合类型的域。

假设我们想要写一个函数来回答查询:

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
FROM emp
WHERE name = 'Bill' OR name = 'Sam';
```

如果使用版本-1的调用规范, 我们可以定义 c\_overpaid 为:

```
#include "postgres.h"
#include "executor/executor.h" /* 用于 GetAttributeByName() */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32          limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    /* 另外, 我们可能更想对空 salary 用 PG_RETURN_NULL() 。*/

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

GetAttributeByName 是返回指定行的属性的 LightDB 系统函数。它有三个参数: 类型为 HeapTupleHeader 的传入参数、想要访问的函数名 以及一个说明该属性是否为空的返回参数。GetAttributeByName 返回一个 Datum 值, 可以把它用合适的 DatumGetXXX() 宏转换成正确的数据类型。注意如果空值标志被设置, 那么返回值是没有意义的, 所以在对结果做任何事情之前应该先检查空值标志。

也有 GetAttributeByNum 函数, 它可以用目标属性的属性号而不是属性名来选择目标属性。

下面的命令声明 SQL 中的 c\_overpaid:

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
AS 'DIRECTORY/funcs', 'c_overpaid'
```

```
LANGUAGE C STRICT;
```

注意我们用了STRICT，这样我们不需要检查输入参数是否 为 NULL。

### 36.10.7. 返回行（组合类型）

要从一个 C 语言函数中返回一个行或者组合类型值，你可以使用一种 特殊的 API，它提供的宏和函数隐藏了大部分的构建组合数据类型的 复杂性。要使用这种 API，源文件中必须包括：

```
#include "funcapi.h"
```

有两种方式可以构建一个组合数据值（以后就叫一个“元组”）： 可以从一个 Datum 值的数组构造，或者从一个 C 字符串（可被传递给该元组 各列的数据类型的输入转换函数）的数组构造。在两种情况下，都首先需要为 该元组的结构获得或者构造一个TupleDesc描述符。在处 理 Datum 时，需要把该TupleDesc传递给 BlessTupleDesc，接着为每一行调用 heap\_form\_tuple。在处理 C 字符串时，需要把该 TupleDesc传递给 TupleDescGetAttInMetadata，接着为每一行调用 BuildTupleFromCStrings。对于返回一个元组集合的函数， 这些设置步骤可以在第一次调用该函数时一次性完成。

有一些助手函数可以用来设置所需的TupleDesc。在大部分 返回组合值的函数中推荐的方式是调用：

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

传递给调用函数本身的同一个fcinfo结构（这当然要求使用的 是版本-1 的调用规范）。resultTypeId可以被指定为 NULL或者一个本地变量的地址以接收该函数的结果类型 OID。 resultTupleDesc应该是一个本地 TupleDesc变量的地址。检查结果是不是 TYPEFUNC\_COMPOSITE，如果是则 resultTupleDesc已经被用所需的 TupleDesc填充（如果不是，你可以报告一个错误，并且 返回“function returning record called in context that cannot accept type record”字样的消息）。

#### Tip

get\_call\_result\_type能够解析一个多态函数结果的实际类型， 因此不仅在返回组合类型的函数中，在返回标量多态结果的函数中它也是非常 有用的。resultTypeId输出主要用于返回多态标量的函数。

#### Note

get\_call\_result\_type有一个兄弟 get\_expr\_result\_type，它被用来解析被表示为一棵表达式 树的函数调用的输出类型。在尝试确定来自函数外部的结果类型时可以用它。 也有一个get\_func\_result\_type，当只有函数的OID 可用时 可以用它。不过这些函数无法处理被声明为返回record的函数，并且get\_func\_result\_type无法解析多态类型，因此你 应该优先使用get\_call\_result\_type。

比较老的，现在已经被废弃的获得TupleDesc的函数是：

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

它可以为一个提到的关系的行类型得到TupleDesc， 还有：

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

可以基于一个类型 OID 得到TupleDesc。这可以被用来 为一种基础或者组合类型获得TupleDesc。不过，对于 返回record的函数它不起作用，并且它无法解析多态类型。

一旦有了一个TupleDesc，如果计划处理 Datum可以调用：

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

如果计划处理 C 字符串，可调用：

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

如果正在编写一个返回集合的函数，你可以把这些函数的结果保存在 FuncCallContext结构中 — 分别使用 tuple\_desc或者attinmeta域。

在处理 Datum 时，使用

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

来用 Datum 形式的用户数据构建一个HeapTuple。

在处理 C 字符串时，使用

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

来用 C 字符串形式的用户数据构建一个HeapTuple。 values是一个 C 字符串数组，每一个元素是返回行 的一个属性。每一个 C 字符串应该是该属性数据类型的输入函数所期望 的格式。为了对一个属性返回空值，values数组中对应的指针应该被设置为NULL。对于你返回的每一行都将 再次调用这个函数。

一旦已经构建了一个要从函数中返回的元组，它必须被转换成一个 Datum。使用

```
HeapTupleGetDatum(HeapTuple tuple)
```

可把一个HeapTuple转换成合法的 Datum。如果你 只想返回一行，那么这个Datum可以被直接返回，在一个 集合返回函数中它也可以被当做当前的返回值。

下一节中会有一个例子。

## 36.10.8. 返回集合

C 语言函数有两个返回集合（多行）的选项。在一种称为ValuePerCall 模式的方法中，一个集合返回函数被重复调用（每次传递相同的参数），并在每次调用时返回一个新行， 直到没有更多行要返回并且 通过返回 NULL 来表示这一点。因此，集合返回函数（SRF）必须在调用之间保存足够的状态以记住它在做什么并在每次调用时返回正确的下一项。 在另一种称为Materialize模式的方法中，SRF 填充并返回一个包含其整个结果的 tuplestore 对象；那么整个结果只发生一次调用，不需要调用间状态。

使用 ValuePerCall 模式时，重要的是要记住查询不能保证运行完成； 也就是说，由于诸如LIMIT之类的选项， 执行程序可能会在获取所有行之前停止调用 set-returning 函

数。这意味着在最后一次调用中执行清理活动是不安全的，因为这可能永远不会发生。对于需要访问外部资源（例如文件描述符）的函数，建议使用 Materialize 模式。

本节的其余部分记录了一组使用 ValuePerCall 模式的 SRF 常用（尽管不是必须使用）的帮助程序宏。有关 Materialize 模式的其他详细信息可以在 src/backend/utils/fmgr/README 中找到。此外，LightDB 源代码分发中的 contrib 模块包含许多使用 ValuePerCall 和 Materialize 模式的 SRF 示例。

要使用此处描述的 ValuePerCall 支持宏，请包含 funcapi.h。这些宏与结构 FuncCallContext 一起使用，该结构包含需要跨调用保存的状态。在调用 SRF 中，fcinfo->flinfo->fn\_extra 用于在调用之间保存指向 FuncCallContext 的指针。宏在第一次使用时自动填充该字段，并期望在后续使用中找到相同的指针。

```
typedef struct FuncCallContext
{
    /*
     * 本次调用以前已经被调用过多少次
     *
     * SRF_FIRSTCALL_INIT() 会为你把 call_cntr 初始化为 0，
     * 并且在每次调用 SRF_RETURN_NEXT() 时增加。
     */
    uint64 call_cntr;

    /*
     * 可选：最大调用次数
     *
     * 这里的 max_calls 只是为了方便，设置它是可选的。
     * 如果没有设置，你必须提供替代的方法来了解函数什么时候做完。
     */
    uint64 max_calls;

    /*
     * 可选：指向用户提供的上下文信息的指针
     *
     * user_fctx 是一个指向你自己的数据的指针，它用来在函数的多次
     * 调用之间保存任意的上下文信息。
     */
    void *user_fctx;

    /*
     * 可选：指向包含属性类型输入元数据的结构的指针
     *
     * attinmeta 被用在返回元组（即组合数据类型）时，在返回基本数据类型
     * 时不会使用。只有想用 BuildTupleFromCStrings() 创建返回元组时才需要它。
     */
    AttInMetadata *attinmeta;

    /*
     * 用于保存必须在多次调用间都存在的结构的内存上下文
     *
     * SRF_FIRSTCALL_INIT() 会为你设置 multi_call_memory_ctx，并且由
     * SRF_RETURN_DONE() 来清理。对于任何需要在 SRF 的多次调用间都
     * 存在的内存来说，它是最合适的内存上下文。
     */
    MemoryContext multi_call_memory_ctx;

    /*
     * 可选：指向包含元组描述的结构体的指针

```



```

*
* tuple_desc 被用在返回元组（即组合数据类型）时，并且只有在用
* heap_form_tuple() 而不是 BuildTupleFromCStrings() 构建元组时才需要它。
* 注意这里存储的 TupleDesc 指针通常已经被先运行过 BlessTupleDesc()。
*/
TupleDesc tuple_desc;

} FuncCallContext;

```

使用此基础结构的SRF将使用的宏是：

SRF\_IS\_FIRSTCALL()

来判断你的函数是否是第一次被调用。在第一次调用时（只能在第一次调用时）使用：

SRF\_FIRSTCALL\_INIT()

使用它来确定您的函数是第一次还是随后被调用。在第一次调用时（仅），调用：

SRF\_PERCALL\_SETUP()

设置使用FuncCallContext。

如果您的函数有数据要在当前调用中返回，请使用：

SRF\_RETURN\_NEXT(funcctx, result)

把它返回给调用者（result必须是类型Datum，可以是一个单一值或者按上文所述准备好的元组）。最后，当函数完成了数据返回后，可使用：

SRF\_RETURN\_DONE(funcctx)

来清理并且结束SRF。

SRF被调用时的当前内存上下文被称作一个瞬时上下文，在两次调用之间会清除它。这意味着你不必对用palloc分配的所有东西调用pfree，它们将自动被释放。不过，如果你想要分配任何需要在多次调用间都存在的数据结构，需要把它们放在其他地方。对于任何需要在SRF结束运行之前都存在的来说，multi\_call\_memory\_ctx引用的内存上下文是一个合适的位置。在大多数情况下，这意味着您应该在首次调用设置时切换到multi\_call\_memory\_ctx。使用funcctx->user\_fctx来保存指向任何此类交叉调用数据结构的指针。（您在multi\_call\_memory\_ctx中分配的数据将在查询结束时自动消失，因此也无需手动释放该数据。）

### Warning

虽然函数的实参在多次调用之间保持不变，但如果在瞬时上下文中反 TOAST 了参数（通常由 PG\_GETARG\_xxx 宏完成），那么被反 TOAST 的拷贝将在每次循环中被释放。相应地，如果你把这些值的引用保存在user\_fctx中，你也必须在反 TOAST 之后把它们拷贝到 multi\_call\_memory\_ctx中，或者确保你只在那个上下文中反 TOAST 这些值。

一个完整的伪代码例子：

```

Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext *funcctx;
    Datum            result;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* 这里是一次性设置代码: */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* 这里是每一次都要做的设置代码: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* 这里只是一种测试是否执行完的方法: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* 这里返回另一个项: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* 这里已经完成了项的返回, 所以只报告事实. */
        /* (不要将清理代码放在这里的。) */
        SRF_RETURN_DONE(funcctx);
    }
}

```

一个返回组合类型的简单SRF的完整例子:

```

PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext *funcctx;
    int              call_cntr;
    int              max_calls;
    TupleDesc        tupdesc;
    AttInMetadata    *attinmeta;

    /* 只在第一次函数调用时做的事情 */

```

```
if (SRF_IS_FIRSTCALL())
{
    MemoryContext    oldcontext;

    /* 创建一个函数上下文，让它在多次调用间都保持存在 */
    funcctx = SRF_FIRSTCALL_INIT();

    /* 切换到适合多次函数调用的内存上下文 */
    oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

    /* 要返回的元组总数 */
    funcctx->max_calls = PG_GETARG_UINT32(0);

    /* 为结果类型构造一个元组描述符 */
    if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                 errmsg("function returning record called in context "
                        "that cannot accept type record")));

    /*
     * 生成后面需要用来从原始 C 字符串产生元组的属性元数据
     */
    attinmeta = TupleDescGetAttInMetadata(tupdesc);
    funcctx->attinmeta = attinmeta;

    MemoryContextSwitchTo(oldcontext);
}

/* 在每一次函数调用都要完成的事情 */
funcctx = SRF_PERCALL_SETUP();

call_cntr = funcctx->call_cntr;
max_calls = funcctx->max_calls;
attinmeta = funcctx->attinmeta;

if (call_cntr < max_calls)    /* 如果还有要发送的 */
{
    char        **values;
    HeapTuple    tuple;
    Datum        result;

    /*
     * 为构建返回元组准备一个值数组。这应该是一个 C
     * 字符串数组，之后类型输入函数会处理它。
     */
    values = (char **) palloc(3 * sizeof(char *));
    values[0] = (char *) palloc(16 * sizeof(char));
    values[1] = (char *) palloc(16 * sizeof(char));
    values[2] = (char *) palloc(16 * sizeof(char));

    snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
    snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
    snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

    /* 构建一个元组 */
    tuple = BuildTupleFromCStrings(attinmeta, values);
}
```

```

    /* 把元组变成 datum */
    result = HeapTupleGetDatum(tuple);

    /* 清理（实际并不必要） */
    pfree(values[0]);
    pfree(values[1]);
    pfree(values[2]);
    pfree(values);

    SRF_RETURN_NEXT(funcctx, result);
}
else /* 如果没有要发送的 */
{
    SRF_RETURN_DONE(funcctx);
}
}

```

在 SQL 中声明这个函数的一种方法是：

```

CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;

```

一种不同的方法是使用 OUT 参数：

```

CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;

```

注意在这种方法中，函数的输出类型在形式上是一种匿名的 record 类型。

### 36.10.9. 多态参数和返回类型

可以声明 C 语言函数来接受和返回 Section 36.2.5 中描述的多态类型。当函数参数或者返回类型被定义为多态类型时，函数的编写者无法提前知道会用什么数据类型调用该函数或者该函数需要返回什么数据类型。在 `fmgr.h` 中提供了两种例程来允许版本-1 的 C 函数发现其参数的实际数据类型以及它要返回的类型。这些例程被称为 `get_fn_expr_rettype(FmgrInfo *flinfo)` 和 `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`。它们返回结果或者参数的类型的 OID，或者当该信息不可用时返回 `InvalidOid`。结构 `flinfo` 通常被当做 `fcinfo->flinfo` 访问。参数 `argnum` 则是从零开始计。`get_call_result_type` 也可被用作 `get_fn_expr_rettype` 的一种替代品。还有 `get_fn_expr_variadic`，它可以被用来找出 `variadic` 参数是否已经被合并到了一个数组中。这主要用于 VARIADIC "any" 函数，因为对于接收普通数组类型的 `variadic` 函数来说总是会发生这类合并。

例如，假设我们想要写一个接收一个任意类型元素并且返回一个该类型的一维数组的函数：

```

PG_FUNCTION_INFO_V1(make_array);

```

```

Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType *result;
    Oid        element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum      element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* 得到提供的元素，小心它为 NULL 的情况 */
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* 只有一个维度 */
    ndims = 1;
    /* 和一个元素 */
    dims[0] = 1;
    /* 且下界是 1 */
    lbs[0] = 1;

    /* 得到该元素类型所需的信息 */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* 现在构建数组 */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                               element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}

```

下面的命令在 SQL 中声明函数make\_array:

```

CREATE FUNCTION make_array(anyelement) RETURNS anyarray
AS 'DIRECTORY/funcs', 'make_array'
LANGUAGE C IMMUTABLE;

```

有一种只对 C 语言函数可用的多态变体：它们可以被声明为接受类型为“any”的参数（注意这种类型名必须用双引号引用，因为它也是一个 SQL 保留字）。这和anyelement相似，不过它不约束不同的“any”参数为同一种类型，它们也不会帮助确定函数的结果类型。C 语言函数也能声明它的第一个参数为 VARIADIC “any”。这可以匹配一个或者多个任意类型的实参（不需要是同一种类型）。这些参数不会像普通 variadic 函数那样被收集到一个数组中，它们将被单独传递给该函数。使用这种特性时，必须用PG\_NARGS()宏以及上述方法来判断实参的个数和类型。还有，这种函数的用户可能希望在他们的函数调用中使用 VARIADIC关键词，以期让该函数将数组元素作为单独的参数对待。如果想要这样，在使用get\_fn\_expr\_variadic检测被标记为VARIADIC的实参之后，函数本身必须实现这种行为。

## 36.10.10. 共享内存和 LWLock

外接程序可以在服务器启动时保留 LWLock 和共享内存。必须通过在 `shared_preload_libraries` 中指定外接程序的共享库来预先载入它。从 `_PG_init` 函数中调用

```
void RequestAddinShmemSpace(int size)
```

可以保留共享内存。

通过从 `_PG_init` 中调用

```
void RequestNamedLWLockTranche(const char *tranche_name, int num_lwlocks)
```

可以保留 LWLock。这将确保一个名为 `tranche_name` 的 LWLock 数组可用，该数组的长度为 `num_lwlocks`。使用 `GetNamedLWLockTranche` 可得到该数组的指针。

为了避免可能的竞争情况，在连接并且初始化共享内存时，每一个后端应该使用 `LWLockAddinShmemInitLock`，如下所示：

```
static mystruct *ptr = NULL;

if (!ptr)
{
    bool    found;

    LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
    ptr = ShmemInitStruct("my struct name", size, &found);
    if (!found)
    {
        initialize contents of shmem area;
        acquire any requested LWLocks using:
        ptr->locks = GetNamedLWLockTranche("my tranche name");
    }
    LWLockRelease(AddinShmemInitLock);
}
```

## 36.10.11. 把 C++ 用于可扩展性

尽管 LightDB 后端是用 C 编写的，只要遵循下面的指导方针也可以用 C++ 编写扩展：

- 所有被后端访问的函数必须对后端呈现一种 C 接口，然后这些 C 函数调用 C++ 函数。例如，对后端访问的函数要求 `extern C` 链接。对需要在后端和 C++ 代码之间作为指针传递的任何函数也要这样做。
- 使用合适的释放方法释放内存。例如，大部分后端内存是通过 `palloc()` 分配的，所以应使用 `pfree()` 来释放。在这种情况下使用 C++ 的 `delete` 会失败。
- 防止异常传播到 C 代码中（在所有 `extern C` 函数的顶层使用一个捕捉全部异常的块）。即使 C++ 代码不会显式地抛出任何异常也需要这样做，因为类似内存不足等事件仍会抛出异常。任何异常都必须被捕捉并且用适当的错误传回给 C 接口。如果可能，用 `-fno-exceptions` 来编译 C++ 以完全消灭异常。在这种情况下，你必须在 C++ 代码中检查失败，例如检查 `new()` 返回的 `NULL`。
- 如果从 C++ 代码调用后端函数，确定 C++ 调用栈值包含传统 C 风格的数据结构（POD）。这是必要的，因为后端错误会产生远距离的 `longjmp()`，它无法正确的退回具有非 POD 对象的 C++ 调用栈。

总之，最好把 C++ 代码放在与后端交互的extern C函数之后，并且避免异常、内存和调用栈泄露。

## 36.11. 函数优化信息

默认情况下，函数只是一个“black box”，数据库系统对它的行为了解得很少。不过，这意味着使用函数的查询执行效率可能会低于它们的能力。可以提供额外的知识帮助计划器优化函数调用。

一些基本事实可以通过CREATE FUNCTION命令中提供的声明性注释来提供。这里面最重要的是函数的volatility category (IMMUTABLE、STABLE或VOLATILE)；在定义函数时，要始终小心地正确指定这个。并行安全属性(PARALLEL UNSAFE、PARALLEL RESTRICTED或PARALLEL SAFE)也必须被指定，如果你希望在并行查询中使用该函数。指定函数的估算执行开销也会有作用，和/或集返回函数估计返回的行数。不过，指定这两个事实的声明方式只允许指定常数值，而这通常是不够的。

也可以将一个planner support function附加到SQL-可调用函数（称为target function），从而提供关于目标函数的知识，该函数过于复杂而无法以声明方式表示。计划器支持函数必须写在 C 中（尽管它们的目标函数可以不是），所以这是一个高级功能，相对很少有人会使用。

计划器支持函数必须具有SQL签名

```
supportfn(internal) returns internal
```

当建立目标函数时，它通过指定SUPPORT子句附加到它的目标函数。

计划器支持函数的 API 的详细信息可以在 LightDB源代码中的src/include/nodes/supportnodes.h文件中找到。这里我们提供了计划器支持函数的概述。支持函数的可能请求集合是可扩展的，所以在将来的版本中可能会有更多(功能)。

在规划期间，根据指定函数的特性，一些函数调用可以进行简化。例如，int4mul(n, 1)可以被简化为n。这种类型的转换可以通过计划器支持函数执行，通过它实现SupportRequestSimplify请求类型。对于在查询解析树中找到其目标函数的每个实例，将调用支持函数。如果它发现特定的调用可以简化成某种其他窗体，它可以构建并返回表示该表达式的解析树。这将为基于函数的操作符自动工作，非常一在刚才的示例中，n \* 1也将简化为n。（但注意这只是一个例子；这个特殊的优化实际上不是标准的LightDB执行）。我们不保证LightDB在支持函数能够简化的情况下，永远不会调用目标函数。确保简化表达式与目标函数的实际执行之间严格等效。

对于返回 boolean 的目标函数，估计使用该函数的 WHERE 子句将选择的行的比重通常会有用。这可以通过实现SupportRequestSelectivity请求类型的支持函数来完成。

如果目标函数的运行时间高度依赖于其输入，提供非固定开销估算可能很有用。这可以通过实现SupportRequestCost请求类型的支持函数来完成。

对于返回集的目标函数，为提供要返回的行数的非常量估计通常很有用。这可以通过实现SupportRequestRows请求类型的支持函数来完成。

对于返回 boolean的目标函数，可以将WHERE中出现的函数调用转换为一个可索引操作符子句或多个子句。转换的子句可能与函数的条件完全相同，或者它们可能比较弱态一些（也就是说，它们可能接受函数条件所不接受的一些值）。在后一种情况下，索引条件被称作lossy；它仍然可用于扫描索引，但必须为索引返回的每一行执行函数调用，以看它是否真的通过 WHERE条件或没有。要建立这样的条件，支持函数必须实现SupportRequestIndexCondition需求类型。

## 36.12. 用户定义的聚集

LightDB中的聚集函数用状态值和状态转换函数定义。也就是，一个聚集操作使用一个状态值，它在每一个后续输入行被处理时被更新。要定义一个新的聚集函数，我们要为状态值选择一种数据类型、一个状态的初始值和一个状态转换函数。状态转换函数接收前一个状态值和该聚集当前行的输入值，并且返回一个新的状态值。万一该聚集的预期结果与需要保存在运行状态之中的数据不同，还能指定一个最终函数。最终函数接受结束状态值并且返回作为聚集结果的任何东西。原则上，转换函数和最终函数只是也可以在聚集环境之外使用的普通函数（实际上，通常出于性能的原因，会创建特殊的只能作为聚集的一部分工作的转换函数）。

因此，除了该聚集的用户所见的参数和结果数据类型之外，还有一种可能不同于参数和结果状态的内部状态值数据类型。

如果我们定义一个聚集但不使用一个最终函数，我们就得到了一个从每一行的列值计算一个运行函数的聚集。sum是这类聚集的一个例子。sum从零开始，并且总是把当前行的值加到它的运行总和上。例如，如果我们希望让一个sum聚集能工作在复数数据类型上，我们只需要该数据类型的加法函数。聚集定义是：

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = ' (0,0)'
);
```

我们可以这样使用：

```
SELECT sum(a) FROM test_complex;
```

```
sum
-----
(34, 53.9)
```

（注意我们依赖于函数重载：有多于一个名为sum的聚集，但是LightDB能够找出哪种 sum 适用于一个类型为complex的列）。

如果没有非空输入值，上述的sum定义将返回零（初始状态值）。也许我们想要在这种情况下返回空 — SQL 标准期望sum以这种方式行事。我们可以通过忽略initcond阶段简单地做到这一点，这样初始状态值就为空。通常这表示sfunc将需要检查一个空状态值输入。但是对于sum和一些其他简单聚集（如max和min），把第一个非空输入值插入到状态变量中并且接着在第二个非空输入值上开始应用转换函数就足够了。如果初始状态值为空并且转换函数被标记为“strict”（即不为空输入调用），LightDB会自动这样做。

“strict”转换函数的另一点默认行为是只要碰到了一个空输入值，之前的状态值就保持不变。因此，空值会被忽略。如果你需要某些其他用于空输入的行为，不要把你的转换函数声明为 strict，而是把它编码为测试空输入并且做所需要的事情。

avg（均值）是一种更复杂的聚集例子。它要求两份运行状态：输入的总和以及输入的计数。最终结果通过将这些量相除而得到。均值是使用一个数组作为状态值的典型实现。例如，内建的avg(float8)实现看起来像：

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = ' {0,0,0}'
);
```



## Note

(float8\_accum要求一个三元素的数组，而不只是两个元素，因为它累积平方和以及输入的总和以及计数。因此它也可以被用于其他聚集函数以及avg)。

SQL 中的聚集函数调用允许用DISTINCT和ORDER BY选项控制以什么顺序把行传递给该聚集的转换函数。这些选项的实现不需要该聚集的支持函数关心。

进一步的细节可见CREATE AGGREGATE命令。

### 36.12.1. 移动聚集模式

聚集函数可以选择性地支持移动聚集模式，这种模式允许很大程度上提高在具有移动帧起点的窗口中执行的聚集函数的速度（有关把聚集函数用作窗口函数请见Section 3.4和Section 4.2.8）。基本思想是在通常的“前向”转换函数之外，聚集提供一个逆向转换函数，该函数允许当行退出窗口帧时从聚集的运行状态值中移除它们的值。例如一个sum聚集使用加法作为前向转换函数，它可以使用减法作为逆向转换函数。如果没有一个逆向转换函数，每一次帧起点移动时，窗口函数机制必须重新从头计算该聚集，这会导致运行时间与输入行的数量乘以平均帧长度成比例。如果有一个逆向转换函数，运行时间只与输入行的数量成比例。

当前状态值和包含在当前状态中最早的行的聚集输入值被传递给逆向转换函数。它必须重新构建出如果给定的输入行不再被聚集（只聚集其后的行）时状态值会是什么样。这有时要求前向转换函数保存比普通聚集模式下更多的状态。因此，移动聚集模式使用一种完全不同于普通模式的实现：它有自己的状态数据类型、自己的前向转换函数以及自己的状态函数（如果需要）。如果不需要额外的状态，这些可以和普通模式的数据类型和函数相同。

作为一个例子，我们可以把上面给定的sum聚集扩展成支持移动聚集模式：

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)',
    msfunc = complex_add,
    minvfunc = complex_sub,
    mstype = complex,
    minitcond = '(0,0)'
);
```

名称以m开始的参数定义移动聚集实现。除了逆向转换函数minvfunc，它们都对应于没有m的普通聚集参数。

用于移动聚集模式的前向转换函数不允许返回空值作为新状态值。如果逆向转换函数返回空值，这被当作一种指示，它表明该逆向函数无法为这个特定输入逆转状态计算，因此该聚集计算将根据当前的帧开始位置重新从头计算。这种习惯允许移动聚集模式被用在一些不适合逆转运行状态值的少数情况下。逆向转换函数在这些情况下可以“撒手不管”，然后在它能够工作的大部分情况中再出来干活。例如，一个浮点数的聚集可能会在必须从运行状态值中移除一个NaN（不是一个数字）输入时撒手不管。

在编写移动聚集支持函数时，很重要的是确保逆向转换函数能够准确地重构正确的状态值。否则会导致用不用移动聚集模式时结果中产生用户可见的差别。为一个聚集增加一个逆向转换函数的例子最初看起来很简单，但是却无法满足float4或者float8输入上的sum的要求。sum(float8)的一种未经考虑的定义可以是

```
CREATE AGGREGATE unsafe_sum (float8)
(
```

```

    stype = float8,
    sfunc = float8pl,
    mstype = float8,
    msfunc = float8pl,
    minvfunc = float8mi
);

```

但是，这个聚集可能给出与没有逆向转换函数时很不同的结果。例如，考虑

```

SELECT
  unsafe_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
FROM (VALUES (1, 1.0e20::float8),
           (2, 1.0::float8)) AS v (n, x);

```

这个查询返回0作为它的第二个结果，而不是我们期待的1。其原因是浮点值的有限精度：把1加到1e20还是会得到1e20，因此从中减去1e20会得到0而不是1。这是对于浮点计算的一种一般性限制，而不是LightDB的限制。

## 36.12.2. 多态和可变聚集

聚集函数可以使用多态状态转换函数或最终函数，这样同样的函数能被用来实现多个聚集。关于多态函数的解释可参见Section 36.2.5 更进一步，聚集函数本身可以被指定为具有多态输入类型和状态类型，允许一个聚集函数服务于多种输入数据类型。这里是一个多态聚集的例子：

```

CREATE AGGREGATE array_accum (anyelement)
(
  sfunc = array_append,
  stype = anyarray,
  initcond = '{}'
);

```

这里，每一次给定聚集调用的实际状态类型是把实际输入类型作为元素的数组类型。该聚集的行为是串接所有输入成一个该类型的数组（注意：内建的聚集array\_agg提供了相似的功能，但是具有比上述定义更好的性能）。

这里是使用两种不同实际数据类型作为参数的输出：

```

SELECT attrelid::regclass, array_accum(attname)
FROM pg_attribute
WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
GROUP BY attrelid;

```

attrelid	array_accum
pg_tablespace	{spcname, spcowner, spcacl, spcoptions}

(1 row)

```

SELECT attrelid::regclass, array_accum(atttypid::regtype)
FROM pg_attribute
WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
GROUP BY attrelid;

```

attrelid	array_accum
pg_tablespace	{name, oid, aclitem[], text[]}

(1 row)

如上述例子所示，通常一个具有多态结果类型的聚集函数有一个多态状态类型。这是必须的，因为否则就无法有意义地声明最终函数：它会需要有一个多态结果类型但是不能有多态参数类型，CREATE FUNCTION将当场拒绝那些无法从调用中推断结果类型的函数。但是使用一个多态状态类型有时并不方便。最常见的情况是，聚集支持函数使用 C 编写并且状态类型应该被声明为internal，因为在 SQL 层面上没有与它等效的类型。为了表述这种情况，可以声明最终函数为接受额外的匹配该聚集输入参数的“dummy”参数。这种假参数总是被传递为空值，因为当最终函数被调用时没有特定的值可用。它们的唯一用途是允许一个多态最终函数的结果类型被连接到该聚集的输入类型。例如，内建聚集array\_agg的定义等效于：

```
CREATE FUNCTION array_agg_transfn(internal, anynonarray)
  RETURNS internal ...;
CREATE FUNCTION array_agg_finalfn(internal, anynonarray)
  RETURNS anyarray ...;

CREATE AGGREGATE array_agg (anynonarray)
(
  sfunc = array_agg_transfn,
  stype = internal,
  finalfunc = array_agg_finalfn,
  finalfunc_extra
);
```

这里，finalfunc\_extra选项指定该最终函数接收除了状态值之外，还接收对应于该聚集输入参数的额外假参数。额外的anynonarray参数允许array\_agg\_finalfn的声明成为合法。

与常规函数的习惯大致相同，可以通过把一个聚集函数的最后一个参数声明为一个VARIADIC数组，这样可以让该函数接受可变数量的参数（见Section 36.5.5）。该聚集的转换函数也必须有相同的数组类型作为它们的最后一个参数。通常这类转换函数也会被标上VARIADIC，但这不被严格要求。

### Note

可变聚集最容易被误用的情况是与ORDER BY选项（见Section 4.2.7）一起使用，因为解析器无法在这样一种组合中是否给出了错误的实际参数数量。要记住在ORDER BY右侧的任何东西都是一个排序键，而不是一个聚集的参数。例如，在

```
SELECT myaggregate(a ORDER BY a, b, c) FROM ...
```

中，解析器将认为看到的是一个聚集函数参数和三个排序键。但是，用户可能想要的是

```
SELECT myaggregate(a, b, c ORDER BY a) FROM ...
```

如果myaggregate是可变的，两种调用都是合法的。

出于相同的原因，在创建具有相同名称以及不同数量的常规参数的聚集函数时一定要三思而后行。

## 36.12.3. 有序集聚集

目前为止我们已经描述的聚集都是“普通”聚集。LightDB还支持有序集聚集，它和普通聚集在两个关键点上相区别。首先，除了对每个输入行都要计算一次的普通聚集参数之外，一个有序集聚集可以有“直接”参数，这类参数针对每次聚集操作只计算一次。其次，用于普通聚集参数的语法需要显式地为它们指定一个排序顺序。一个有序集聚集通常被用来实现一

种依赖于特定行序的计算（例如排名或者百分位数），因此排序是任何调用都要求的。例如，`percentile_disc`的内建定义等效于：

```
CREATE FUNCTION ordered_set_transition(internal, anyelement)
  RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement)
  RETURNS anyelement ...;

CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement)
(
  sfunc = ordered_set_transition,
  stype = internal,
  finalfunc = percentile_disc_final,
  finalfunc_extra
);
```

这个聚集接受一个float8直接参数（百分位数分数）以及一个可以是任意可排序数据类型的聚集输入。它可以用来得到一个家庭收入的中位数：

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_disc
-----
          50489
```

这里0.5是一个直接参数，它对于要作为一个在行之间变化的百分位数分数没有意义。

和普通聚集的情况不同，用于有序集聚集的输入行排序不是在幕后完成的，而是由该聚集的支持函数负责完成。典型的实现方法是在该聚集的状态值中保持对于一个“`tuplesort`”对象的引用，把到来的行输入给该对象，然后完成排序并且在最终函数中读出该数据。这种设计允许最终函数能够执行特殊操作，例如把附加的“假想”行注入到被排序的数据中。虽然用由PL/pgSQL或另一种 PL 语言编写的支持函数通常能够实现普通聚集，但是有序集聚集通常必须用 C 编写，因为它们的状态值无法用任何 SQL 数据类型来定义（在上面的例子中，注意状态值被声明为类型`internal`——这很典型）。此外，由于最终函数会执行排序，后面就不能继续通过再次执行转移函数增加输入行。这意味着最终函数不是`READ_ONLY`的，它必须在`CREATE AGGREGATE`中被声明为`READ_WRITE`，或者在可以有额外的最终函数调用利用已经排序好的状态时声明为`SHAREABLE`。

用于一个有序集聚集的状态转移函数接收当前状态值外加对于每一行的聚集输入值，并且返回更新后的状态值。这和普通聚集的定义相同，但是注意没有提供直接参数（如果有）。最终函数接收最后的状态值、直接参数（如果有）的值以及对应于聚集输入的空值（如果指定了`finalfunc_extra`）。正如普通聚集，只有聚集是多态时`finalfunc_extra`才真正有用，那时就需要额外的假参数把最终函数的结果类型连接到该聚集的输入类型。

当前，有序集聚集不能被用做窗口函数，并且因此没有必要让它们支持移动聚集模式。

## 36.12.4. 部分聚集

可选地，一个聚集函数可以支持部分聚集。部分聚集的思想是在输入数据的不同子集上独立的运行该聚集的状态转移函数，然后把从这些子集得到的状态值组合起来产生最终的状态值，这样得到的状态值与在单次聚集操作中扫描所有输入得到的状态值相同。这种模式可以被用来进行并行聚集，用不同的工作者进程扫描表的不同部分。每一个工作者产生一个部分状态值，最后把这些部分状态值组合产生最终状态值（在未来，这种模式可能也会被用于组合在本地表和远程表上的聚集，但目前还未实现）。

为了支持部分聚集，聚集定义必须提供一个组合函数，这个函数接收两个该聚集的状态类型（表示在输入行的两个不同子集上得到的聚集结果）并且产生一个该状态类型的新值，该结果表示组合哪些聚集结果后的状态。至于来自两个集合的输入行的相对顺序则并没有指定。这意味着通常不可能为对输入行顺序敏感的聚集定义出可用的组合函数。

作为简单的例子，通过指定组合函数为与其转移函数中相同的“两者中较大者”和“两者中较小者”比较函数，MAX和MIN聚集可以支持部分聚集。SUM聚集则只需要一个额外的函数作为组合函数（同样，组合函数与其转移函数相同，除非状态值的宽度比输入数据类型更宽）。

组合函数很像一个把状态类型值而不是底层输入类型值作为其第二个参数的转移函数。尤其是处理空值和严格函数的规则是相似的。此外，如果聚集定义指定了非空的initcond，记住那不仅会被作为每一次部分聚集运行的初始状态，还会被作为组合函数的初始状态，对每一个部分结果都会调用组合函数将部分结果组合到该初始状态中。

如果聚集的状态类型被声明为internal，则组合函数应负责在用于聚集状态值的内存上下文中分配其结果。这意味着当第一个输入为NULL时，不能简单地返回第二个输入，因为那个值将会在错误的上下文中并且将不具有足够的寿命。

当聚集的状态类型被声明为internal时，通常聚集定义提供序列化函数和反序列化函数也是合适的，这两个函数允许这样一种状态值被从一个进程复制到另一个进程。如果没有这些函数就无法执行并行聚集，并且未来的本地/远程聚集之类的应用也可能无法工作。

一个序列化函数必须接收一个单一的internal类型参数并且返回一个bytea类型的结果，它表示把状态值打包成一个平面化的字节串。反过来，反序列化函数是上述转换的逆变换。反序列化函数必须接收两个类型为bytea和internal的参数，并且返回类型为internal的结果（第二个参数没有被使用并且总是为零，它的存在是由于类型安全性的原因）。反序列化函数的结果应该直接在当前内存上下文中分配，这与组合函数的结果不同，因为它不需要长期存在。

还有一点值得提示的是关于要被并行执行的聚集，聚集本身必须被标记上PARALLEL SAFE。其支持函数上的并行安全性标记不会被参考。

## 36.12.5. 聚集的支持函数

用 C 编写的函数能够通过调用AggCheckCallContext检测它是作为聚集支持函数调用的，例如：

```
if (AggCheckCallContext(fcinfo, NULL))
```

检查这个区别的原因是当它为真时，第一个输入必须是一个临时状态值并且可以因此安全地被就地修改而不是分配一个新的副本。例子可见int8inc()（虽然聚集的转移函数总是被允许就地修改转移值，但不鼓励聚集的最终函数这样做。如果最终函数要这样做，必须在创建聚集时声明这种行为。更多细节请参考CREATE AGGREGATE）。

AggCheckCallContext的第二个参数可以被用来检索保存有聚集状态值的内存上下文。这对希望把“扩展”对象（见Section 36.13.1）用作状态值的转移函数有用。在第一次调用时，转移函数应该返回一个扩展对象，其内存上下文是聚集状态上下文的一个子节点，然后在后续的调用中都保持返回同一个扩展对象。示例请见array\_append()（array\_append()不是任意内建聚集的转移函数，但其编写的目的就是在被用作一种自定义聚集的转移函数时表现得有效）。

另一种可用于由 C 编写的聚集函数的支持例程是AggGetAggref，它返回定义该聚集调用的Aggref解析节点。这主要对有序聚集有用，它能检查Aggref的子结构来找出它们本应实现的排序顺序。在LightDB源代码的orderedsetaggs.c中可以找到例子。

## 36.13. 用户定义的类型

如Section 36.2中所述，LightDB能够被扩展成支持新的数据类型。这一节描述了如何定义新的基本类型，它们是被定义在SQL语言层面之下的数据类型。创建一种新的基本类型要求使用低层语言（通常是 C）实现在该类型上操作的函数。

这一节中的例子可以在源代码src/tutorial目录下的complex.sql和complex.c中找到。运行这些例子的指令可以在该目录的README文件中找到。

一种用户定义的类型必须总是具有输入和输出函数。这些函数决定该类型如何出现在字符串中（用于用户输入或者对用户的输出）以及如何在内存中组织该类型。输入函数采用一个空终止的字符串作为它的参数并且返回该类型的内部（内存）表达。输出函数采用该类型的内部表达作为参数并且返回一个空终止的字符串。如果我们想要对该类型做更多事情而不是只存储它，我们必须提供为我们想要的任何操作提供额外的实现函数。

假设我们想要定义一种类型`complex`，它表示复数。一种在内存中表达复数的自然的方法是下面的 C 结构：

```
typedef struct Complex {
    double    x;
    double    y;
} Complex;
```

我们将需要让它成为一种传引用类型，因为它没办法放到一个单一的Datum值中。

至于该类型的外部字符串表达，我们选择了一种字符串形式的(x, y)。

输入和输出函数通常并不难编写，特别是输出函数。但是在定义类型的外部字符串表达时，记住你必须最终为该表达编写一个完整并且鲁棒的解析器作为你的输入函数。例如：

```
PG_FUNCTION_INFO_V1(complex_in);
```

```
Datum
complex_in(PG_FUNCTION_ARGS)
{
    char        *str = PG_GETARG_CSTRING(0);
    double      x,
               y;
    Complex     *result;

    if (sscanf(str, "(%lf,%lf)", &x, &y) != 2)
        ereport(ERROR,
                (errmsg("invalid input syntax for type %s: \"%s\"",
                       "complex", str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

输出函数可以简单地写作：

```
PG_FUNCTION_INFO_V1(complex_out);
```

```
Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex     *complex = (Complex *) PG_GETARG_POINTER(0);
    char        *result;

    result = psprintf("(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

你应当让输入和输出函数互为彼此的逆函数。如果不这样做，当你需要把数据转储到一个文件并且以后将它重新读入时会遇到很严重的问题。在涉及到浮点数时这是一个特别常见的问题。

可选地，一种用户定义的类型可以提供二进制输入和输出例程。二进制 I/O 通常比文本 I/O 更快但是可移植性更差。与文本 I/O 一样，定义准确的外部二进制表达是你需要负责的工作。大部分的内建数据类型都尝试提供一种不依赖机器的二进制表达。对于 `complex`，我们的工作将建立在为类型 `float8` 提供的二进制 I/O 转换器上：

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

一旦我们编写了 I/O 函数并且把它们编译到了一个共享库中，我们就可以在 SQL 中定义 `complex` 类型。首先我们把它声明为一种 `shell` 类型：

```
CREATE TYPE complex;
```

这个语句的作用是为要定义的类型创建了一个占位符，这样允许我们在定义其 I/O 函数时引用该类型。现在我们可以定义 I/O 函数：

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

```
CREATE FUNCTION complex_recv(internal)
  RETURNS complex
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT;
```

```
CREATE FUNCTION complex_send(complex)
  RETURNS bytea
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT;
```

最后，我们可以提供该数据类型的完整定义：

```
CREATE TYPE complex (
  internallength = 16,
  input = complex_in,
  output = complex_out,
  receive = complex_recv,
  send = complex_send,
  alignment = double
);
```

在定义了一种新的基本类型后，LightDB会自动提供对这种类型的数组支持。数组类型通常具有和基本类型相同的名称以及一个前置的下划线字符（\_）。

一旦数据类型存在，我们就能够声明额外的函数来提供在该数据类型上有用的操作。然后可以在函数之上定义操作符，并且如果需要，可以创建操作符类来支持对该数据类型进行索引。这些额外的内容会在下面的小节中讨论。

如果数据类型的内部表达是可变长的，则内部表达必须遵循可变长数据的标准布局：头四个字节必须是一个char[4]域，它从来不会被直接访问（通常被称为v1\_len\_）。你必须使用SET\_VARSIZE()宏在这个域中存储整个数据的尺寸（包括长度域本身），并且使用VARSIZE()来检索它（这些宏之所以存在，是因为长度域可能会根据平台来进行解码）。

更多细节请见CREATE TYPE命令的描述。

### 36.13.1. TOAST 考量

如果你的数据类型值的尺寸（内部形式）是可变的，更适合让该数据类型变成可TOAST的（见Section 64.2）。即便值总是很小不会被压缩或者线外存储你也应该这样做，因为TOAST也能通过减少头部负荷来为小数据减少空间。

为了支持TOAST存储，在该数据类型上操作的C函数必须总是要使用PG\_DETOAST\_DATUM解包任何交给它们的被TOAST过的值（习惯上这些细节都通过定义类型相关的GETARG\_DATATYPE\_P宏隐藏起来）。然后，在运行CREATE TYPE命令时，指定内部长度为variable并且选择某个不是plain的适当的存储选项。

如果数据对齐无关紧要（不管是为一个特定函数或者因为数据类型指定了字节对齐），那么有可能避免PG\_DETOAST\_DATUM的一些开销。你可以转而使用PG\_DETOAST\_DATUM\_PACKED（习惯上通过定义一个GETARG\_DATATYPE\_PP宏隐藏）并且使用宏VARSIZE\_ANY\_EXHDR以及VARDATA\_ANY来访问一个可能包装过的数据。此外，即使数据类型定义指定了一种对齐方式，这些宏返回的数据也不是对齐过的。如果对齐对你很重要，你必须使用常规的PG\_DETOAST\_DATUM接口。

#### Note

老的代码经常声明v1\_len\_为一个int32域而不是char[4]。只要结构定义含有其他具有至少int32对齐的域，这就是OK的。但是在使用可能未对齐的数据



时，使用这样一种结构定义就是危险的，编译器可能会把它当作一个授权来假定数据实际上已经被对齐，在对于对齐很严格的架构上会导致核心转储。

TOAST支持带来的另一个特性是能够拥有一种扩展内存中数据表达，它比存储在磁盘上的格式使用起来更方便。常规的或者“扁平的” varlena 存储格式最终只是一堆字节，它不能包含指针，因为它可能会被复制到内存中的其他位置。对于复杂数据类型，扁平格式使用起来可能代价更高，因此LightDB提供了一种方式把扁平格式“扩展”成更适合计算的一种表达，然后在该数据类型的函数之间传递这种在内存中的格式。

要使用扩展存储，数据类型必须遵循src/include/utils/expandeddatum.h 中给定的规则定义一种扩展的格式，并且提供函数把扁平的 varlena 值“扩展”到该格式以及从该格式“扁平化”回常规的 varlena 表达。然后确保所有该数据类型的 C 函数都能接受这两种表达（可能通过一接收到其中一种就立刻转换成另一种来做到）。这不要求一次性修改所有该数据类型的现有函数，因为标准的PG\_DETOAST\_DATUM宏可以把扩展输入转换成常规扁平格式。因此，现有的用于扁平 varlena 格式的函数仍然能够用于扩展输入（虽然效率略低）。它不需要被转换，直到需要提高性能。

直到如何对付扩展表达的 C 函数通常分为两类：只能处理扩展格式的，以及能同时处理扩展或扁平 varlena 输入的。前者更容易编写，但是可能总体效率较低，因为由单个函数将一种扁平输入转换为扩展的形式开销可能会超过在扩展格式上操作所节省的开销。在只需要处理扩展格式时，可以把扁平输入到扩展形式的转换隐藏在一个参数获取宏中，这样该函数就显得不比处理传统 varlena 输入的函数更复杂了。要处理两种类型的输入，需要编写一个参数获取函数来反 TOAST 外部、短头部以及压缩的 varlena 输入，但不需要处理扩展输入。这样一个函数可以被定义为返回一个指向由扁平 varlena 格式和扩展格式组成的联合的指针。调用者可以使用 VARATT\_IS\_EXPANDED\_HEADER() 宏来判断它们接收到的是哪种格式。

TOAST机制不仅允许把常规 varlena 值同扩展值区分开来，还能区分指向扩展值的“read-write”和“read-only”指针。只需要检查扩展值或者只会以安全的并且非语义可见的方式更改扩展值的 C 函数不需要关心它们收到的是哪种类型的指针。如果收到一个读写指针，要为输入值产生一个修改版本的 C 函数将被允许就地修改该扩展输入值，但是如果它们收到的是一个只读指针则不能修改，在这种情况下它们不得不先复制该值产生一个用于修改的新值。构建了新扩展值的 C 函数应该总是返回一个指向该值的读写指针。还有，如果一个就地修改读写扩展值的 C 函数中途失败，它应该负责让该值处于一种正常的状态。

有关使用扩展值的例子，请见标准数组这种基础结构，特别是 src/backend/utils/adt/array\_expanded.c。

## 36.14. 用户定义的操作符

对于一个完成实际工作的底层函数的调用来说，每一个操作符都是“语法糖”，因此在创建操作符之前你必须先创建底层函数。不过，一个操作符不只是语法糖，因为它携带了额外的信息来帮助查询规划器优化使用该操作符的查询。下一节将致力于解释这些额外信息。

LightDB支持前缀和中缀操作符。操作符可以被重载，也就是说相同的操作符名称可以被用于具有不同操作数数量和类型的操作符。在执行一个查询时，系统会根据提供的操作数的数量和类型决定要调用的操作符。

这里有一个创建用于对两个复数做加法的操作符的例子。我们假设我们已经创建了类型complex（见Section 36.13 的定义）。首先我们需要一个函数做这个加法，然后我们可以定义该操作符：

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;
```

```
CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    function = complex_add,
    commutator = +
);
```

现在我们可以执行一个这样的查询：

```
SELECT (a + b) AS c FROM test_complex;
```

```

      c
-----
(5.2, 6.05)
(133.42, 144.95)
```

这里我们已经展示了如何创建一个二元操作符。要创建前缀操作符，只要忽略leftarg。在CREATE OPERATOR中只要求procedure子句和参数子句。例子中展示的commutator子句是一个可选的子句，它被用作一个查询优化器使用的提示。有关commutator以及其他优化器提示的细节出现在下一小节中。

## 36.15. 操作符优化信息

一个LightDB的操作符定义能够包括几种可选的子句，它们可以把有关操作符行为的有用的事情告诉系统。只要合适就应该提供这些子句，因为它们能够为使用该操作符的查询带来可观的速度提升。但是如果你提供了它们，你必须确保它们是正确的！不正确地使用一个优化子句可能导致很慢的查询、错误的输出或者其他不好的事情。如果你没有把握你可以总是省去优化子句，这样做的唯一后果是查询会比正常的速度慢。

在LightDB的未来版本中可能会增加更多的优化子句。这里描述的优化子句都是版本13.8-23.4能理解的。

还可以将计划器支持的函数附加到作为操作符基础的函数中，从而提供另一种向系统讲述操作符行为的方法。更多信息参见Section 36.11

### 36.15.1. COMMUTATOR

如果提供了COMMUTATOR子句，它指定一个操作符作为被定义的操作符的交换子。如果对于所有可能输入的  $x$ 、 $y$  值， $(x \ A \ y)$  等于  $(y \ B \ x)$ ，我们可以说操作符  $A$  是操作符  $B$  的交换子。注意， $B$  也是  $A$  的交换子。例如，用于一种特定数据类型的操作符  $<$  和  $>$  通常互为交换子，并且操作符  $+$  通常和它本身是交换的。但是操作符  $-$  通常不能与任何东西交换。

一个可交换操作符的左操作数类型与其交换子的右操作数类型相同，反之亦然。因此要查找交换子，只需要给LightDB该交换子操作符的名称即可，并且在COMMUTATOR子句中也只需要提供它的名称。

为将要在索引和连接子句中使用的操作符提供交换子信息是很关键的，因为这允许查询优化器把这样一个子句“翻转”成不同计划类型所需的形式。例如，考虑一个这样的 WHERE 子句  $tab1.x = tab2.y$ ，其中  $tab1.x$  和  $tab2.y$  是一种用户定义的类型，并且假设  $tab2.y$  被索引。除非优化器能决定如何把该子句翻转成  $tab2.y = tab1.x$ ，否则它无法产生一个索引扫描，因为索引扫描机制期望看到被索引列出现在被给出的操作符的左边。LightDB将无法简单地假定有一个可用的变换  $=$  操作符的创建者必须指定它是合法的（通过为该操作符标记交换子信息）。

在你定义一个子交换的操作符时，你这样做就行了。自拟定义一堆交换的操作符时，事情有一点棘手：如何在没有定义第二个操作符时完成第一个操作符的定义？因为第一个操作符需要第二个操作符作为其交换子。对这个问题有两种解决方案：

- 一种方法是忽略你定义的第一个操作符的COMMUTATOR子句，并且然后在第二个操作符的定义中提供第一个操作符作为交换子。由于LightDB知道交换的操作符是成对出现的，当它看到第二个定义时它将自动回去并且填上第一个定义中缺失的COMMUTATOR子句。
- 另一种更直接的方法是就在两个定义中包括COMMUTATOR子句。当LightDB处理第一个定义并且意识到COMMUTATOR引用了一个不存在的操作符时，系统将为那个操作符在系统目录中创建一个虚拟项。这个虚拟项只有操作符名称、左右操作数类型和结果类型的数据，因为这些是LightDB在此时能够推断出来的所有东西。第一个操作符的目录项将会链接到这个虚拟项。稍后，当你定义第二个操作符时，系统用来自第二个定义的额外信息更新那个虚拟项。如果你尝试在虚拟操作符还未被填充之前使用它，你将只会得到一个错误消息。

## 36.15.2. NEGATOR

如果提供了NEGATOR子句，它指定一个操作符是正在被定义的操作符的求反器。如果操作符 A 和 B 都返回布尔结果并且对于所有可能的  $x, y$  输入都有  $(x \ A \ y)$  等于  $\text{NOT} (x \ B \ y)$ ，那么我们可以说 A 是 B 的求反器。注意 B 也是 A 的求反器。例如， $<$  和  $>=$  就是大部分数据类型的一对求反器。一个操作符不可能是它自身的求反器。

与交换子不同，一对一元操作符可以合法地被标记为对方的求反器；这意味着对于所有  $x$  有  $(A \ x)$  等于  $\text{NOT} (B \ x)$ 。

一个操作符的求反器必须具有和被定义的操作符相同的左或右操作数类型，因此正如COMMUTATOR一样，NEGATOR子句中只需要给出操作符的名称即可。

提供一个求反器对查询优化器非常有帮助，因为它允许 $\text{NOT} (x = y)$ 这样的表达式被简化为 $x <> y$ 。这可能比你想象的更多地发生，因为NOT操作可能会被作为其他调整的结果被插入。

求反器对的定义可以使用与定义交换子对相同的方法来完成。

## 36.15.3. RESTRICT

如果提供了RESTRICT子句，它为该操作符指定一个限制选择度估计函数（注意这是一个函数名而不是一个操作符名）。RESTRICT子句只对返回boolean的二元操作符有意义。一个限制选择度估计器背后的思想是猜测一个表中有多大比例的行对于当前的操作符和一个特定的常数值将会满足一个

column OP constant

形式的WHERE子句条件。这能通过告知优化器具有这种形式的WHERE子句将会消除掉多少行来协助它的工作（你可能会好奇，如果常数位于左部会发生什么？好吧，COMMUTATOR就是干这个的）。

编写一个新的限制选择度估算函数已经超出了本章的范围，但是幸运地是你通常可以将系统的一个标准估算器用于很多你自己的操作符。标准的限制估算器有：

```
eqsel用于=
neqsel用于<>
scalartsel用于<
scalarlesel用于<=
scalargtsel用于>
scalargesel用于>=
```

你能经常成功地为具有非常高或者非常低选择度的操作符使用eqsel或neqsel，即使它们实际上并非相等或不相等。例如，近似相等几何操作符使用eqsel的前提是假定它们通常只匹配表中的一小部分项。

你可以使用scalartsel、scalarlesel、scalargtsel以及scalargesel来比较被转换为数字标量进行范围比较具有意义的数据类型。如果可能，增加一种能被src/backend/utls/adt/selfuncs.c中的函数convert\_to\_scalar()所理解的数据类型（最后，这个函数应该被通

过pg\_type系统目录的一列所标识的针对每个数据类型的函数所替换，但是那还没有发生）。如果你没有这样做，还是能工作，但是优化器的估计将不会达到最好的效果。

另一个有用的内置选择性估计函数是matchingsel，如果为输入数据类型收集标准 MCV 和/或直方图统计信息，它几乎适用于任何二元运算符。它的默认估计值设置为eqsel中使用的默认估计值的两倍，使其最适合于比等式更不严格的比较运算符。（或者您可以调用底层的generic\_restriction\_selectivity函数，提供不同的默认估计。）

有一些额外的选择度估算函数是为src/backend/utils/adt/geo\_selfuncs.c中的几何操作符设计的：areasel、positionsel和contsel。在写这份材料时，这些还只是存根，但是你可能想要使用它们（或者甚至改进它们）。

## 36.15.4. JOIN

如果提供了JOIN子句，表示用于该操作符的一个连接选择度估计函数（注意这是一个函数名而不是一个操作符名）。JOIN子句只对返回boolean的二元操作符有意义。一个连接选择度估算器背后的思想是猜测一对表中有多大比例的行对于当前的操作符将会满足一个

```
table1.column1 OP table2.column2
```

形式的WHERE子句条件。和RESTRICT子句一样，这通过让优化器知道哪种连接序列需要做的工作最少来极大地帮助优化器。

一如既往，这一章将不会尝试解释如何编写一个连接选择度估算函数，而只是建议你在适当的时候使用一种标准估算器：

```
eqjoinsel用于=
neqjoinsel用于<>
scalarltjoinsel用于<
scalarelejoinsel用于<=
scalargtjoinsel用于>
scalargejoinsel用于>=
matchingjoinsel用于通用匹配运算符
areajoinsel用于基于 2D 区域比较
positionjoinsel用于基于 2D 位置比较
contjoinsel用于基于 2D 包含比较
```

## 36.15.5. HASHES

如果存在HASHES子句，它告诉系统它被许可为基于这个操作符的一个连接使用哈希连接方法。HASHES只对返回boolean的二元操作符有意义，并且实际上该操作符必须必须表达某种数据类型或数据类型对的相等。

哈希连接之下的假设是连接操作符只能对哈希到相同哈希码的左右值返回真。如果两个值被放到不同的哈希桶中，连接将根本不会比较它们，这隐式地假定该连接操作符的结果必须是假。因此，为不表示某种形式相等的操作符指定HASHES是没有意义的。在大部分情况下，只有为在两端都是相同数据类型的操作符支持哈希才有意义。不过，有时可以为两种或更多数据类型设计兼容的哈希函数，也就是说，对于“相等”的值（即使具有不同的表达）会产生相同哈希码的函数。例如，在哈希不同宽度的证书时，安排这个属性相当简单。

要被标记为HASHES，连接操作符必须出现在一个哈希索引操作符族中。当你创建该操作符时这不会被强制，因为要引用的操作符族当然不可能已经存在。但是如果这样的操作符族不存在，尝试在哈希连接中使用该操作符将在运行时失败。系统需要用该操作符族来为操作符的输入数据类型寻找数据类型相关的哈希函数。当然，在创建操作符族之前，你还必须创建合适的哈希函数。

在准备一个哈希函数时应当慎重，因为有一些方法是依赖于机器的，这样它可能无法做正确的事情。例如，如果你的数据类型是一个结构，其中可能有无用的填充位，你不能简单地把整个结构传递给hash\_any（除非你编写你自己的操作符和函数按照推荐的策略来保证未被使

用的位总是为零)。另一个例子是在符合IEEE浮点标准的机器上，负数零和正数零是不同的值（不同的位模式），但是它们被定义为相等。如果一个浮点值可能包含负数零，那么需要额外的步骤来保证它产生的哈希值与正数零产生的相同。

一个可哈希连接的操作符必须拥有一个出现在同一操作符族中的交换子（如果两个操作数数据类型相同，那么就是它自身，否则是一个相关的相等操作符）。如果情况不是这样，在使用该操作符时，可能会发生规划器错误。此外，一个支持多种数据类型的哈希操作符族为数据类型的每一种组合都提供相等操作符是一个好主意（但是并不被严格要求），这会带来更好的优化。

#### Note

一个可哈希连接的操作符底层的函数必须被标记为可交换或者稳定。如果它是不稳定的，系统将永远不会为一个哈希连接尝试使用该操作符。

#### Note

如果一个可哈希连接的操作符有一个被标记为 `strict` 的底层函数，该函数必须也是 `complete`：也就是对于任意两个非空输入它应当返回真或假，从不会返回空。如果没有遵守这个规则，IN操作的哈希优化可能产生错误的结果（特别是，当依据标准的正确答案可能是空时，IN可能会返回假，或者它会产生一个错误来抱怨它没有准备会收到一个空结果）。

## 36.15.6. MERGES

如果存在MERGES子句，它告诉系统它被许可为基于这个操作符的一个连接使用归并连接方法。MERGES只对返回boolean的二元操作符有意义，并且实际上该操作符必须必须表达某种数据类型或数据类型对的相等。

归并连接的思想是排序左右手表并且接着并行扫描它们。这样，两种数据类型必须能够被完全排序，并且该连接操作符必须只为落在排序顺序上“同一位置”的值对返回成功。实际上这意味着该连接操作符必须和相等的行为一样。但是只要两种不同的数据类型在逻辑上是兼容的，就能对它们使用归并连接。例如，`smallint-versus-integer`相等操作符就是可归并连接的。我们只需要将两种数据类型变成逻辑上兼容的序列的排序操作符。

要被标记为MERGES，该连接操作符必须作为一个btree索引操作符的相等成员出现。当你创建该操作符时，这不是强制的，因为要引用的操作符族当然可能还不存在。但是除非能找到一个匹配的操作符族，否则该操作符将不会被实际用于归并连接。MERGES标志因此扮演着一一种对于规划器的提示，表示值得去寻找一个匹配的操作符族。

一个可归并连接的操作符必须拥有一个出现在同一操作符族中的交换子（如果两个操作数数据类型相同，那么就是它自身，否则是一个相关的相等操作符）。如果情况不是这样，在使用该操作符时，可能会发生规划器错误。此外，一个支持多种数据类型的btree操作符族为数据类型的每一种组合都提供相等操作符是一个好主意（但是并不被严格要求），这会带来更好的优化。

#### Note

一个可归并连接的操作符底层的函数必须被标记为可交换或者稳定。如果它是不稳定的，系统将永远不会为一个归并连接尝试使用该操作符。

## 36.16. 索引的接口扩展

迄今为止已经描述的过程让我们能够定义新的类型、新的函数以及新的操作符。但是，我们还不能在一种新数据类型的列上定义索引。要做这件事情，我们必须为新数据类型定义一个操作符类。在这一小节稍后的部分，我们将用一个例子阐述这部份内容：一个用于 B-树索引方法的操作符类，它以绝对值的升序存储和排序复数。

操作符类可以被分成操作符族来体现语义兼容的类之间的联系。当只涉及到一种单一数据类型时，一个操作符类就足矣。因此我们将先把重点放在这种情况上，然后再回到操作符族。

### 36.16.1. 索引方法和操作符类

pg\_am表为每一种索引方法都包含一行（内部被称为访问方法）。LightDB中内建了对表常规访问的支持，但是所有的索引方法则是在pg\_am中描述。可以通过编写必要的代码并且在pg\_am中创建一项来增加一种新的索引访问方法——但这超出了本章的范围（见Chapter 58）。

一个索引方法的例程并不直接了解它将要操作的数据类型。而是由一个操作符类标识索引方法用来操作一种特定数据类型的一组操作。之所以被称为操作符类是因为它们指定的一件事情就是可以被用于一个索引的WHERE子句操作符集合（即，能被转换成一个索引扫描条件）。一个操作符类也能指定一些索引方法内部操作所需的支持函数，这些过程不能直接对应于能用于索引的任何WHERE子句操作符。

可以为相同的数据类型和索引方法定义多个操作符类。通过这种方式，可以为一种数据类型定义多个索引语义集合。例如，一个B-树索引要求在它要操作的每一种数据类型上都定义一个排序顺序。对一种复数数据类型来说，拥有一个可以根据复数绝对值排序的 B-树操作符类和另一个可以根据实数部分排序的操作符类可能会有用。典型地，其中一个操作符类将被认为是最常用的并且将被标记为那种数据类型和索引方法的默认操作符类。

相同的操作符类名称可以被用于多个不同的索引方法（例如，B-树和哈希索引方法都有名为int4\_ops的操作符类）。但是每一个这样的类都是一个独立实体并且必须被单独定义。

### 36.16.2. 索引方法策略

与一个操作符类关联的操作符通过“策略号”标识，它被用来标识每个操作符在其操作符类中的语义。例如，B-树在键上施行了一种严格的顺序（较小到较大），因此“小于”和“大于等于”这样的操作符就是 B-树所感兴趣的。因为LightDB允许用户定义操作符，LightDB不能看着一个操作符（如<和>=）的名字并且说出它是哪一种比较。取而代之的是，索引方法定义了一个“策略”集合，它们可以被看成是广义的操作符。每一个操作符类会说明对于一种特定的数据类型究竟是哪个实际的操作符对应于每一种策略以及该索引语义的解释。

B-树索引方法定义了五种策略，如Table 36.3所示。

Table 36.3. B-树策略

操作	策略号
小于	1
小于等于	2
等于	3
大于等于	4
大于	5

哈希索引只支持等值比较，因此它们只使用一种策略，如Table 36.4所示。

Table 36.4. 哈希策略

操作	策略号
等于	1

GiST 索引更加灵活：它们根本没有一个固定的策略集合。取而代之的是，每一个特定 GiST 操作符类的“consistency”支持例程会负责解释策略号。例如，一些内建的 GiST 索引操作符类索引二维几何对象，它们提供Table 36.5中所示的“R-树”策略。其中四个是真正的二维测试（重叠、相同、包含、被包含），其中四个只考虑 X 方向，其他四个提供 Y 方向上的相同测试。

Table 36.5. GiST 二维“R-树”策略

操作	策略号
左参数严格地位于右参数的左边	1
左参数不会延伸到右参数的右边	2
重叠	3
左参数不会延伸到右参数的左边	4
左参数严格地位于右参数的右边	5
相同	6
包含	7
被包含	8
不会延伸到高于	9
严格低于	10
严格高于	11
不会延伸到低于	12

GIN 没有一个固定的策略集合。取而代之的是，每一个操作符类的支持例程负责根据该操作符类的定义解释策略号。例如，被内建操作符类用于数组的策略号如Table 36.6所示。

Table 36.6. GIN 数组策略

操作	策略号
重叠	1
包含	2
被包含	3
等于	4

注意上文列出的所有操作符都返回布尔值。实际上，所有作为索引方法搜索操作符定义的操作符必须返回类型boolean，因为它们必须出现在一个WHERE子句的顶层来与一个索引一起使用（某些索引访问方法还支持排序操作符，它们通常不返回布尔值，这种特性在Section 36.16.中讨论）。

### 36.16.3. 索引方法支持例程

对于系统来说只有策略信息通常不足以断定如何使用一种索引。实际上，为了能工作，索引方法还要求额外的支持例程。例如，B-树索引方法必须能比较两个键并且决定其中一个是否大于、等于或小于另一个。类似地，哈希索引方法必须能够为键值计算哈希码。这些操作并不对应 SQL 命令的条件中使用的操作符。它们是索引方法在内部使用的管理例程。

与策略一样，操作符类会标识哪些函数应该为一种给定的数据类型扮演这些角色以及相应的语义解释。索引方法定义它需要的函数集合，而操作符类则会通过为函数分配由索引方法说明的“支持函数号”来标识正确的函数。

此外，一些 opclass 允许用户指定控制其行为的参数。每个内置索引访问方法都有一个可选的 options支持函数，它定义了一组特定于 opclass 的参数。

如Table 36.所示，B-树要求一个比较支持函数，并且允许在操作符类作者的选项中提供四个额外的支持函数。这些支持函数的要求在Section 60.中会进一步解释。

Table 36.7. B-树支持函数

函数	支持号
比较两个键并且返回一个小于零、等于零或大于零的整数，它表示第一个键小于、等于或者大于第二个键。	1
返回C可调用的排序支持函数的地址（可选）。	2
将一个测试值与一个基础值加上/减去一个偏移量的结果进行比较，根据比较的结果返回真或假（可选）	3
确定使用运算符类应用 btree 重复数据删除优化的索引是否安全（可选）	4
定义一组特定于此运算符类的选项（可选）	5

如Table 36.8所示，哈希索引要求一个支持函数，并且允许在操作符类作者的选项中提供两个额外的支持函数。

Table 36.8. 哈希支持函数

函数	支持号
为一个键计算32位哈希值	1
给定一个64位salt，计算一个键的64位哈希值。如果salt为0，结果的低32位必须匹配会由函数1计算出来的值（可选）	2
定义一组特定于此运算符类的选项（可选）	3

如Table 36.9所示，GIN 索引有七个支持函数，其中四个是可选的（详见Chapter 62）。

Table 36.9. GIN 支持函数

函数	描述	支持号
compare	比较两个键并且返回一个小于零、等于零或大于零的整数，它表示第一个键小于、等于或者大于第二个键	1
extractValue	从一个要被索引的值中抽取键	2
extractQuery	从一个查询条件中抽取键	3
consistent	判断值是否匹配查询条件（布尔变体）（如果支持函数 6 存在则是可选的）	4
comparePartial	比较来自查询的部分键和来自索引的键，并且返回一个小于零、等于零或大于零的整数，表示 GIN 是否应该忽略该索引项、把该项当做一个匹配或者停止索引扫描（可选）	5
triConsistent	判断值是否匹配查询条件（三元变体）（如果支持函数 4 存在则是可选的）	6
options	定义一组特定于此运算符类的选项（可选）	7

和搜索操作符不同，支持函数返回特定索引方法所期望的数据类型，例如在 B 树的比较函数中是一个有符号整数。每个支持函数的参数数量和类型也取决于索引方法。对于 B 树和哈希，比较和哈希支持函数和包括在操作符类中的操作符接收一样的输入数据类型，但是大部分 GiST、GIN 支持函数则不是这样。

### 36.16.4. 一个例子

现在我们已经看过了基本思想，这里是创建一个新操作符类的例子（可以在源代码的src/tutorial/complex.c和src/tutorial/complex.sql中找到这个例子）。该操作符类封装了以



绝对值顺序排序复数的操作符，因此我们为它取名为`complex_abs_ops`。首先，我们需要一个操作符集合。定义操作符的过程已经在Section 36.1中讨论过。对于一个 B-树上的操作符类，我们需要的操作符有：

- 绝对值小于（策略 1）
- 绝对值小于等于（策略 2）
- 绝对值等于（策略 3）
- 绝对值大于等于（策略 4）
- 绝对值大于（策略 5）

定义一个比较操作符的相关集合最不容易出错的方式是，先编写 B-树比较支持函数，然后编写该支持函数的包装器函数。这降低了极端情况下得到不一致结果的几率。遵照这种方法，我们首先编写：

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double    amag = Mag(a),
             bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

现在小于函数看起来像这样：

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

其他四个函数的区别只在于它们如何比较内部函数的结果与 0。

接下来我们基于这些函数声明 SQL 的函数和操作符：

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'filename', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
    leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
    commutator = >, negator = >=,
    restrict = scalarltsel, join = scalarltjoinsel
);
```

指定正确的交换子和求反器操作符很重要，合适的限制和连接选择度函数也是一样，否则优化器将无法有效地利用索引。

其他值得注意的事情：

- 只能有一个操作符被命名为=且两个操作数都为类型complex。在这种要求下，我们对于complex没有任何其他操作符=。但是如果我们是在构建一种实际的数据类型，我们可能想让=成为复数的普通等值操作（不是绝对值的相等）。这样，我们需要为complex\_abs\_eq使用某种其他的操作符名称。
- 尽管LightDB能够处理具有相同 SQL 名称的函数（只要它们具有不同的参数数据类型），但 C 只能处理具有给定名称一个全局函数。因此，我们不能简单地把 C 函数命名为abs\_eq之类的东西。通常，在 C 函数名中包括数据类型的名称是一种好习惯，这样就不会与其他数据类型的函数发生冲突。
- 我们可以让函数也具有abs\_eq这样的 SQL 名称，而依靠LightDB通过参数数据类型来区分它和其他同名 SQL 函数。为了保持例子的简洁，我们这里让 C 级别和 SQL 级别的函数具有相同的名称。

下一步是注册 B-树所要求的支持例程。实现支持例程的 C 代码例子在包含操作符函数的同一文件中。我们这样来声明该函数：

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

现在我们已经有了所需的操作符和支持例程，就可以最终创建操作符类：

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
    OPERATOR          1          < ,
    OPERATOR          2          <= ,
    OPERATOR          3          = ,
    OPERATOR          4          >= ,
    OPERATOR          5          > ,
    FUNCTION          1          complex_abs_cmp(complex, complex);
```

做好了！现在应该可以在complex列上创建并且使用 B-树索引了。

我们可以把操作符项写得更繁琐，像这样：

```
OPERATOR          1          < (complex, complex) ,
```

但是当操作符操作的数据类型和正在定义的操作符类所服务的数据类型相同时可以不用这么做。

上述例子假定这个新操作符类是complex数据类型的默认 B-树操作符类。如果不是这样，只需要省去关键词DEFAULT。

## 36.16.5. 操作符类和操作符族

到目前为止，我们暗地里假设一个操作符类只处理一种数据类型。虽然在一个特定的索引列中必定只有一种数据类型，但是把被索引列与一种不同数据类型的值比较的索引操作通常也很有用。还有，如果与一种操作符类相关的扩数据类型操作符有用，通常情况是其他数据类型也有其自身相关的操作符类。在相关的类之间建立起明确的联系会很有用，因为这可以帮助规划器进行 SQL 查询优化（尤其是对于 B-树操作符类，因为规划器包含了大量有关如何使用它们的知识）。

为了处理这些需求，LightDB使用了操作符族的概念。一个操作符族包含一个或者多个操作符类，并且也能包含属于该族整体而不属于该族中任何单一类的可索引操作符和相应的支持函数。我们说这样的操作符和函数是“松散地”存在于该族中，而不是被绑定在一个特定的类中。通常每个操作符类包含单一数据类型的操作符，而跨数据类型操作符则松散地存在于操作符族中。

一个操作符族中的所有操作符和函数必须具有兼容的语义，其中的兼容性要求由索引方法设定。你可能因此而奇怪为什么要这么麻烦地把族的特定子集单另出来成为操作符类，并且实际上（由于很多原因）这种划分与操作符之间没有什么直接的关联，只有操作符族才是实际的分组。定义操作符类的原因是，它们指定了特定索引对操作符族的依赖程度。如果一个索引使用着一个操作符类，那么不删除该索引是不能删除该操作符类的——但是操作符族的其他部分（即其他操作符类和松散的操作符）可以被删除。因此，一个操作符类应该包含一个索引在特定数据类型上正常工作所需的最小操作符和函数集合，而相关但不关键的操作符可以作为操作符族的松散成员被加入。

例如，LightDB有一个内建的 B-树操作符族 `integer_ops`，它包括分别用于类型 `bigint` (`int8`)、`integer` (`int4`) 和 `smallint` (`int2`) 列上索引的操作符类 `int8_ops`、`int4_ops` 以及 `int2_ops`。这个族也包含跨数据类型比较操作符，它们允许对这些类型中的任意两种进行比较，这样可以通过一种类型的比较值来搜索另一种类型之上的索引。这个族可以用这些定义来重现：

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
-- 标准 int8 比较
OPERATOR 1 < ,
OPERATOR 2 <= ,
OPERATOR 3 = ,
OPERATOR 4 >= ,
OPERATOR 5 > ,
FUNCTION 1 btint8cmp(int8, int8) ,
FUNCTION 2 btint8sortsupport(internal) ,
FUNCTION 3 in_range(int8, int8, int8, boolean, boolean) ,
FUNCTION 4 btequalimage(oid) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
-- 标准 int4 比较
OPERATOR 1 < ,
OPERATOR 2 <= ,
OPERATOR 3 = ,
OPERATOR 4 >= ,
OPERATOR 5 > ,
FUNCTION 1 btint4cmp(int4, int4) ,
FUNCTION 2 btint4sortsupport(internal) ,
FUNCTION 3 in_range(int4, int4, int4, boolean, boolean) ,
FUNCTION 4 btequalimage(oid) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
-- 标准 int2 比较
OPERATOR 1 < ,
OPERATOR 2 <= ,
OPERATOR 3 = ,
OPERATOR 4 >= ,
OPERATOR 5 > ,
FUNCTION 1 btint2cmp(int2, int2) ,
```

```
FUNCTION 2 btint2sortsupport(internal) ,
FUNCTION 3 in_range(int2, int2, int2, boolean, boolean) ,
FUNCTION 4 btequalimage(oid) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
-- 跨类型比较 int8 vs int2
OPERATOR 1 < (int8, int2) ,
OPERATOR 2 <= (int8, int2) ,
OPERATOR 3 = (int8, int2) ,
OPERATOR 4 >= (int8, int2) ,
OPERATOR 5 > (int8, int2) ,
FUNCTION 1 btint82cmp(int8, int2) ,

-- 跨类型比较 int8 vs int4
OPERATOR 1 < (int8, int4) ,
OPERATOR 2 <= (int8, int4) ,
OPERATOR 3 = (int8, int4) ,
OPERATOR 4 >= (int8, int4) ,
OPERATOR 5 > (int8, int4) ,
FUNCTION 1 btint84cmp(int8, int4) ,

-- 跨类型比较 int4 vs int2
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
OPERATOR 4 >= (int4, int2) ,
OPERATOR 5 > (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,

-- 跨类型比较 int4 vs int8
OPERATOR 1 < (int4, int8) ,
OPERATOR 2 <= (int4, int8) ,
OPERATOR 3 = (int4, int8) ,
OPERATOR 4 >= (int4, int8) ,
OPERATOR 5 > (int4, int8) ,
FUNCTION 1 btint48cmp(int4, int8) ,

-- 跨类型比较 int2 vs int8
OPERATOR 1 < (int2, int8) ,
OPERATOR 2 <= (int2, int8) ,
OPERATOR 3 = (int2, int8) ,
OPERATOR 4 >= (int2, int8) ,
OPERATOR 5 > (int2, int8) ,
FUNCTION 1 btint28cmp(int2, int8) ,

-- 跨类型比较 int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ,

-- 跨类型的in_range函数
FUNCTION 3 in_range(int4, int4, int8, boolean, boolean) ,
FUNCTION 3 in_range(int4, int4, int2, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int8, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int4, boolean, boolean) ;
```

注意这种定义“重载”了操作符策略和支持函数号：每一个编号在该族中出现多次。只要一个特定编号的每一个实例都有可区分的输入数据类型，就允许这样做。输入类型等于操作符类输入类型的实例是该操作符类的主要操作符和支持函数，并且在大部分情况下应该被声明为该操作符类的一部分而不是作为操作符族的松散成员存在。

如Section 60. 中的细节所述，在一个 B-树操作符族中，所有该族中的操作符必须以兼容的方式排序。对该族中的每一个操作符都必须有一个与该操作符具有相同的两个输入数据类型的支持函数。我们推荐让操作符族保持完整，即对每一种数据类型的组合都应该包括所有的操作符。每个操作符类只应该包括非跨类型操作符和用于其数据类型的支持函数。

为了构建一个多数据类型的哈希操作符族，必须为该族支持的每一种数据类型创建相兼容的哈希支持函数。这里的兼容性是指这些函数对于任意两个被该族中等值操作符认为相等的值会保证返回相同的哈希码，即便这些值具有不同的类型时也是如此。当这些类型具有不同的物理表示时，这通常难以实现，但是在某些情况下是可以做到的。此外，将该操作符族中一种数据类型的值通过隐式或者二进制强制造型转换成该族中另一种数据类型时，不应该改变所计算出的哈希值。注意每种数据类型只有一个支持函数，而不是每个等值操作符一个。我们推荐让操作符族保持完整，即对每一种数据类型的组合提供一个等值操作符。每个操作符类只应该包括非跨类型等值操作符和用于其数据类型的支持函数。

GiST 和 GIN 索引没有任何明显的跨数据类型操作的概念。它们所支持的操作符集合就是一个给定操作符类能够处理的主要支持函数。

### Note

在LightDB 8.3 之前，没有操作符族的概念，并且因此要在索引中使用的任何跨数据类型操作符必须被直接绑定到该索引的操作符类中。虽然这种方法仍然有效，但是已被废弃，因为它会让索引的依赖过于广泛，还因为当两种数据类型都在同一操作符族中有操作符时规划器可以更有效地处理跨数据类型比较。

## 36. 16. 6. 操作符类上的系统依赖

LightDB使用操作符类来以更多方式推断操作符的属性，而不仅仅是它们是否能被用于索引。因此，即便不准备对你的数据类型的列建立索引，也可能想要创建操作符类。

特别地，ORDER BY和DISTINCT等 SQL 特性要求对值的比较和排序。为了在用户定义的数据类型上实现这些特性，LightDB会为数据类型查找默认 B-树操作符类。这个操作符类的“equals”成员定义了用于GROUP BY和DISTINCT的值的等值概念，而该操作符类施加的排序顺序定义了默认的ORDER BY顺序。

如果一种数据类型没有默认的 B-树操作符类，系统将查找默认的哈希操作符类。但由于这类操作符类只提供等值，所以它只能支持分组而不能支持排序。

在一种数据类型没有默认操作符类时，如果尝试对该数据类型使用这些 SQL 特性，你将得到类似“could not identify an ordering operator”（无法标识排序操作符）的错误。

### Note

在版本 7.4 以前的LightDB中，排序和分组操作将隐式地使用名为=、<以及>的操作符。新的依赖于默认操作符类的行为避免了对具有特定名字的操作符行为作出任何假设。

通过在一个USING选项中指定一个非默认B-树操作符类的小于操作符，可以使用该操作符进行排序，例如

```
SELECT * FROM mytable ORDER BY somecol USING ~<~;
```

或者，在USING中指定该操作符类的大于操作符可以选择升序的排序。

用户定义类型的数组的比较还依赖于该类型的默认B-树操作符类所定义的语义。如果没有默认的B-树操作符类，但有一个默认的哈希操作符类，则支持数组的相等比较，但不支持顺序的比较。

另一种要求更多数据类型相关知识的SQL特性是窗口函数（见Section 4.2.8 的RANGE offset PRECEDING/FOLLOWING选项。对于这样的一个查询

```
SELECT sum(x) OVER (ORDER BY x RANGE BETWEEN 5 PRECEDING AND 10 FOLLOWING)
FROM mytable;
```

不足以了解如何用x进行排序，数据库还必须理解如何对当前行的x值“减5”或者“加10”以标识当前窗口帧的边界。把得到的边界与其他行的x值用B-树操作符类提供的比较操作符（定义了ORDER BY顺序）进行比较是可能的——但是加和减操作符并不是该操作符类的一部分，因此应该用哪些操作符呢？硬编码的选择是不切实际的，因为不同的排序顺序（不同的B-树操作符）可能需要不同的行为。因此，一个B-树操作符类可以指定一个in\_range支持函数，它封装有对排序顺序有意义的加和减行为。如果有多种数据类型可以用作RANGE子句中的偏移量，甚至可以提供多个in\_range支持函数。如果与窗口的ORDER BY子句关联的B-树操作符类没有一个匹配的in\_range支持函数，则不支持RANGE offset PRECEDING/FOLLOWING选项。

另一个要点是，出现在一个哈希操作符族中的操作符是哈希连接、哈希聚集和相关优化的候选。这些情况下哈希操作符族就是至关重要的，因为它标识了要使用的哈希函数。

## 36.16.7. 排序操作符

有些索引访问方法（当前只有 GiST和SP-GiST）支持排序操作符的概念。到目前为止我们所讨论的都是搜索操作符。搜索索引时，会用搜索操作符来寻找所有满足 WHERE indexed\_column operator constant 的行。注意被返回的匹配行的顺序是没有任何保证的。相反，一个排序操作符并不限制能被返回的行集合，而是决定它们的顺序。扫描索引时，会使用排序操作符来以 ORDER BY indexed\_column operator constant 所表示的顺序返回行。这样定义排序操作符的原因是，如果该操作符能度量距离，它就能支持最近邻搜索。例如，这样的一个查询

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

寻找离一个给定目标点最近的十个位置。位置列上的 GiST 索引可以有效地完成这个查询，因为<->是一个排序操作符。

搜索操作符必须返回布尔结果，排序操作符通常返回某种其他类型，例如浮点、数字或者距离。这种类型通常不同于被索引的数据类型。为了避免硬编码有关不同数据类型行为的假设，需要定义一个排序操作符来提名一个 B-树操作符族指定结果数据类型的排序顺序。正如我们在前一节介绍的，B-树操作符族定义了LightDB的顺序概念，因此这是一种自然的表达。由于点<->操作符返回float8，可以在一个操作符类创建命令中这样指定它：

```
OPERATOR 15 <-> (point, point) FOR ORDER BY float_ops
```

其中float\_ops是包括float8上操作的内置操作符族。这种声明说明该索引能够以<->操作符的递增顺序返回行。

## 36.16.8. 操作符类的特性

有两个操作符类的特性我们还没有讨论，主要是因为它们对于最常用的索引方法不太有用。

通常，把一个操作符声明为一个操作符类（或操作符族）的成员意味着该索引方法能够使用该操作符准确地检索满足WHERE条件的行集。例如：

```
SELECT * FROM table WHERE integer_column < 4;
```

恰好可以被该整数列上一个 B-树索引满足。但是也有情况下索引只是作为匹配行的非精确向导。例如，如果一个 GiST 索引只存储几何对象的边界框，那么它无法精确地满足测试非矩形对象（如多边形）之间相交的WHERE条件。但是我们可以使用该索引来寻找边界框与目标对象的边界框相交的对象，并且只在通过该索引找到的对象上做精确的相交测试。如果适用于这种场景，该索引被称为对该操作符是“有损的”。有损索引搜索通过在一行可能满足或者不满足该查询条件时返回一个recheck标志来实现。核心系统将接着在检索到的行上测试原始查询条件来看它是否应该被作为一个合法匹配返回。如果索引被保证能返回所有所需的行外加一些额外的行，这种方法就能有效，因为那些额外的行可以通过执行原始的操作符调用来消除。支持有损搜索的索引方法（当前有 GiST、SP-GiST 和 GIN）允许个别操作符类的支持函数设置 recheck 标志，因此这也是一种操作符类的重要特性。

再次考虑在索引中只存储复杂对象（如多边形）的边界框的情况。在这种情况下，把整个多边形存储在索引项中没有很大价值 — 我们也可以只存储一个更简单的box类型对象。这种情况通过CREATE OPERATOR CLASS中的STORAGE选项表示：

```
CREATE OPERATOR CLASS polygon_ops
  DEFAULT FOR TYPE polygon USING gist AS
  ...
  STORAGE box;
```

当前，只有 GiST、GIN 索引方法支持不同于列数据类型的STORAGE类型。在使用STORAGE时，GiST 的支持例程compress和decompress必须处理数据类型转换。在 GIN 中，STORAGE类型标识“key”值的类型，它通常不同于被索引列的类型 — 例如，一个用于整数数组列的操作符类可能具有整数键值。GIN 的支持例程extractValue和extractQuery负责从被索引值中抽取键。

## 36.17. 打包相关对象到一个扩展中

一个对LightDB有用的扩展通常包括多个 SQL 对象，例如，一种新的数据类型将需要新函数、新操作符以及可能的新的索引操作符类。将所有这些对象收集到一个单一包中有助于简化数据库管理。LightDB称这样一个包为一个扩展。要定义一个扩展，你至少需要一个包含创建该扩展的对象的SQL命令的脚本文件以及一个指定扩展本身的一些基本属性的控制文件。如果扩展包括 C 代码，通常还有一个 C 代码编译而成的共享库文件。一旦你有了这些文件，一个简单的CREATE EXTENSION命令可以把这些对象载入到你的数据库。

使用一个扩展而不是只运行SQL脚本载入一堆“松散”对象到数据库的主要优点是，LightDB将能理解该扩展的对象是一起的。你可以用一个单一的DROP EXTENSION命令删除所有的对象（不用维护一个单独的“卸载”脚本）。甚至更有用的一点是，lt\_dump知道它不应该转储该扩展中的个体成员对象 — 它将只在转储中包括一个CREATE EXTENSION命令。这大大简化了迁移到一个包含不同于旧版扩展中对象的新版扩展的工作。不过，注意在把这样一个转储载入到一个新数据库时，该扩展的控制、脚本和其他文件必须可用。

LightDB不会让你删除包含在一个扩展中的个体对象，除非删除整个扩展。还有，虽然你能够改变一个扩展的成员对象的定义（例如，通过CREATE OR REPLACE FUNCTION改变一个函数），记住被修改后的定义将不会被lt\_dump转储。这种改变通常只有在你并发地在扩展脚本文件中做出相同更改时才有意义（但是对于包含配置数据的表有特殊的规定，

见Section 36.17.3。在生产环境中，通常更好的方式是创建一个扩展更新脚本来执行对扩展中成员对象的更改。

扩展脚本可能会设置扩展中所含对象的特权，使用GRANT和REVOKE语句。每一个对象的最终特权集合（如果设置了任何特权）将被存储在pg\_init\_privs系统目录中。使用lt\_dump时，CREATE EXTENSION命令将被包括在转储中，后面会跟着必要的GRANT和REVOKE语句集合来将对象的特权设置成取得该转储时的样子。

LightDB当前不支持扩展脚本发出CREATE POLICY或者SECURITY LABEL语句。这些东西的设置应该在扩展被创建好之后来进行。所有在扩展对象上创建的 RLS 策略和安全标签都将被包括在lt\_dump创建的转储中。

扩展机制也对打包调整一个扩展中所含 SQL 对象定义的修改脚本有规定。例如，如果一个扩展的 1.1 版本比 1.0 版本增加了一个函数并且更改了另一个函数的函数体，该扩展的作者可以提供更新脚本来做这两个更改。那么ALTER EXTENSION UPDATE命令可以被用来应用这些更改并且跟踪在给定数据库中实际安装的是该扩展的哪个版本。

能作为一个扩展的成员的 SQL 对象的种类如ALTER EXTENSION所示。尤其是数据库集簇范围的对象（例如数据库、角色和表空间）不能作为扩展成员，因为一个扩展只在一个数据库范围内可见（尽管一个扩展脚本并没有被禁止创建这些对象，但是这样做将无法把它们作为扩展的一部分来跟踪）。还要注意虽然一个表可以是一个扩展的成员，它的扶助对象（例如索引）不会被直接认为是该扩展的成员。另一个重点是模式可以属于扩展，但是反过来不行：一个扩展本身有一个不被限定的名称并且不存在于任何模式“中”。不过，扩展的成员对象只要对象类型合适就可以属于模式。一个扩展拥有包含其成员对象的模式可能合适也可能不合适。

如果一个扩展的脚本创建任何临时对象（例如临时表），在当前会话的剩余部分会把它们当作扩展的成员，但是在会话结束会自动删除它们，这和其他任何临时对象是一样的。对于不删除整个扩展就不能删除扩展的成员对象的规则来说，这是一种例外。

## 36.17.1. 扩展文件

CREATE EXTENSION命令依赖每一个扩展都有的控制文件，控制文件必须被命名为扩展的名称加上一个后缀.control，并且必须被放在安装的SHAREDIR/extension目录中。还必须至少有一个SQL脚本文件，它遵循命名模式extension--version.sql（例如，foo--1.0.sql表示扩展foo的1.0版本）。默认情况下，脚本文件也被放置在SHAREDIR/extension目录中，但是控制文件中可以为脚本文件指定一个不同的目录。

一个扩展控制文件的格式与lightdb.conf文件相同，即是一个parameter\_name = value赋值的列表，每行一个。允许空行和#引入的注释。注意对任何不是单一词或数字的值加上引号。

一个控制文件可以设置下列参数：

directory (string)

包含扩展的SQL脚本文件的目录。除非给出一个绝对路径，这个目录名是相对于安装的SHAREDIR目录。默认行为等效于指定directory = 'extension'。

default\_version (string)

该扩展的默认版本（就是如果在CREATE EXTENSION中没有指定版本时将会被安装的那个）。尽管可以忽略这个参数，但如果没有出现VERSION选项时那将会导致CREATE EXTENSION失败，因此你通常不会希望这样做。

comment (string)

一个关于该扩展的注释（任意字符串）。该注释会在初始创建扩展时应用，但是扩展更新时不会引用该注释（因为可能会覆盖用户增加的注释）。扩展的注释也可以通过在脚本文件中写上COMMENT命令来设置。



encoding (string)

该脚本文件使用的字符集编码。当脚本文件包含任何非 ASCII 字符时，可以指定这个参数。否则文件都会被假定为数据库编码。

module\_pathname (string)

这个参数的值将被用于替换脚本文件中每一次出现的MODULE\_PATHNAME。如果设置，将不会进行替换。通常，这会被设置为\$libdir/shared\_library\_name并且接着MODULE\_PATHNAME被用在CREATE FUNCTION命令中进行 C-语言函数的创建，因此该脚本文件不必把共享库的名称硬编码在其中。

requires (string)

这个扩展依赖的其他扩展名的一个列表，例如requires = 'foo, bar'。被依赖的扩展必须先于这个扩展安装。

superuser (boolean)

如果这个参数为true（默认情况），只有超级用户能够创建该扩展或者将它更新到一个新版本。（另请参见下面的 trusted）如果被设置为false，只需要用来执行安装中命令或者更新脚本的特权。如果任何脚本命令需要超级用户权限，这将通常被设置为true。（这样的命令总之都会失败，但提前给出错误对用户更友好。）

trusted (boolean)

这个参数，如果设置为 true（非默认值），则允许某些非超级用户安装superuser已设置为 true 的扩展。具体地说，对于在当前数据库上具有CREATE特权的任何人，都允许安装。当执行CREATE EXTENSION的用户不是超级用户，但允许通过此参数安装时，则此安装或更新脚本作为引导超级用户运行，而不是作为调用用户。这个参数是不相干的，如果 superuser 为 false。通常，对于可能允许访问其他超级用户功能的扩展这将不被设为真(true)，例如文件系统访问。此外，标记一个可信的扩展需要大量额外的努力，以安全的编写扩展的安装和更新脚本；参见Section 36.17.6

relocatable (boolean)

如果一个扩展可能在初始创建之后将其所含的对象移动到一个不同的模式中，它就是relocatable。默认值是false，即该扩展是不可重定位的。详见Section 36.17.2

schema (string)

这个参数只能为非可重定位扩展设置。它强制扩展被载入到给定的模式而非其他模式中。只有在初始创建一个扩展时才会参考schema参数，扩展更新时则不会参考这个参数。详见Section 36.17.2

除了主要控制文件extension.control，一个扩展还可以有二级控制文件，它们以extension--version.control的风格命名。如果提供了二级控制文件，它们必须被放置在脚本文件的目录中。二级控制文件遵循主要控制文件相同的格式。在安装或更新该扩展的版本时，一个二级控制文件中设置的任何参数将覆盖主要控制文件中的设置。不过，参数directory以及default\_version不能在二级控制文件中设置。

一个扩展的SQL脚本文件能够包含任何 SQL 命令，除了事务控制命令（BEGIN、COMMIT等）以及不能在一个事务块中执行的命令（如VACUUM）。这是因为脚本文件会被隐式地在一个事务块中被执行。

一个扩展的SQL脚本文件也能包含以\echo开始的行，它将被扩展机制忽略（当作注释）。如果脚本文件被送给ltsql而不是由CREATE EXTENSION载入（见Section 36.17.中的示例脚本），这种机制通常被用来抛出错误。如果没有这种功能，用户可能会意外地把该扩展的内容作为“松散的”对象而不是一整个扩展载入，这样的状态恢复起来比较麻烦。

如果扩展脚本包含字符串 `@extowner@`，该字符串将替换为调用 `CREATE EXTENSION` 或 `ALTER EXTENSION` 的用户的名称（适当引用）。通常，此特性会被标记为受信任的扩展使用，将所选对象的所有权分配给调用用户，而不是引导超级用户。（但是，在这样做时应该小心谨慎。例如，将C语言函数的所有权分配给非超级用户将为该用户创建一个权限升级路径。）

尽管脚本文件可以包含指定编码允许的任何字符，但是控制文件应该只包含纯 ASCII 字符，因为LightDB没有办法知道一个控制文件是什么编码。实际上，如果你想在扩展的注释中使用非 ASCII 字符只有一个问题。推荐的方法是不使用控制文件的 `comment` 参数，而是使用脚本文件中的 `COMMENT ON EXTENSION` 来设置注释。

## 36.17.2. 扩展可再定位性

用户常常希望把扩展中包含的对象载入到一个与扩展的作者所设想的不一样的模式中。对于这种可重定位性，有三种支持的级别：

- 一个完全可重定位的扩展能在任何时候被移动到另一个模式中，即使在它被载入到一个数据库中之后。这种移动通过 `ALTER EXTENSION SET SCHEMA` 命令完成，该命令会自动地把所有成员对象重命名到新的模式中。通常，只有扩展不包含任何对其所在模式的内部假设时才可能这样做。还有，该扩展的对象必须全部在同一个模式中（忽略那些不属于任何模式的对象，例如过程语言）。要让一个扩展变成完全可定位，在它的控制文件中设置 `relocatable = true`。
- 一个扩展可能在安装过程中是可重定位的，但是安装完后就不再可重定位。典型的情况是扩展的脚本文件需要显式地引用目标模式，例如为 SQL 函数设置 `search_path` 属性。对于这样一种扩展，在其控制文件中设置 `relocatable = false`，并且使用 `@extschema@` 在脚本文件中引用目标模式。在脚本被执行前，所有这个字符串的出现都将被替换为实际的目标模式名。用户可以使用 `CREATE EXTENSION` 的 `SCHEMA` 选项设置目标模式名。
- 如果扩展根本就不支持重定位，在它的控制文件中设置 `relocatable = false`，并且还设置 `schema` 为想要的目标模式的名称。这将阻止使用 `CREATE EXTENSION` 的 `SCHEMA` 选项修改目标模式，除非它指定的是和控制文件中相同的模式。如果该扩展包括关于模式名的内部假设且模式名不能使用 `@extschema@` 的方法替换，这种选择通常是必须的。`@extschema@` 替换机制在这种情况下也是可用的，不过由于模式名已经被控制文件所决定，它的使用受到了很大的限制。

在所有情况下，脚本文件将被用 `search_path` 执行，它最初会被设置为指向目标模式，也就是说 `CREATE EXTENSION` 做的也是等效的工作：

```
SET LOCAL search_path TO @extschema@, pg_temp;
```

这允许由这个脚本文件创建的对象进入到目标模式中。如果脚本文件希望，它可以改变 `search_path`，但这种用法通常是不受欢迎的。在 `CREATE EXTENSION` 结束后，`search_path` 会被恢复到之前的设置。

如果控制文件中给出了 `schema` 参数，目标模式就由该参数决定，否则目标模式由 `CREATE EXTENSION` 的 `SCHEMA` 选项给出，如果以上两者都没有给出则会用当前默认的对象创建模式（在调用者 `search_path` 中的第一个）。当使用扩展文件的 `schema` 参数时，如果目标模式还不存在将创建它，但是在另外两种情况下它必须已经存在。

如果在控制文件中的 `requires` 中列举了任何先导扩展，它们的目标模式会被追加到 `search_path` 的初始设置中，遵循新扩展的目标模式。这允许新扩展的脚本文件能够看到它们的对象。

为了安全，`pg_temp` 在所有情况下都会自动添加到 `search_path` 的末尾。

尽管一个不可重定位的扩展能够包含散布在多个模式中的对象，通常还是值得将意图用于外部使用的所有对象放置在一个模式中，这被认为是该扩展的目标模式。这样一种安排可以在依赖的扩展创建过程中方便地与 `search_path` 的默认设置一起工作。

### 36.17.3. 扩展配置表

某些扩展包括配置表，其中包含可以由用户在扩展安装后增加或修改的数据。通常，如果一个表是一个扩展的一部分，该表的定义和内容都不会被`lt_dump`转储。但是对于一个配置表来说并不希望是这样的行为，任何用户做出的数据修改都需要被包括在转储中，否则该扩展在重载之后的行为将和转储之前不同。

要解决这个问题，一个扩展的脚本文件可以把一个它创建的表或者序列标记为配置关系，这将导致`lt_dump`把该表或者序列的内容（而不是它的定义）包括在转储中。要这样做，在创建表或序列之后调用函数`pg_extension_config_dump(regclass, text)`，例如

```
CREATE TABLE my_config (key text, value text);
CREATE SEQUENCE my_config_seq;

SELECT pg_catalog.pg_extension_config_dump('my_config', '');
SELECT pg_catalog.pg_extension_config_dump('my_config_seq', '');
```

可以用这种方法标记任意数量的表或者序列。与`serial`或者`bigserial`列相关联的序列也可以被标记。

当`pg_extension_config_dump`的第二个参数是一个空字符串时，该表的全部内容都会被`lt_dump`转储。这通常只有在表被扩展脚本创建为初始为空时才正确。如果在表中混合有初始数据和用户提供的数据，`pg_extension_config_dump`的第二个参数提供了一种WHERE条件来选择要被转储的数据。例如，你可能会做

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT
standard_entry');
```

并且确保只有扩展脚本创建的行中`standard_entry`才为真。

对于序列，`pg_extension_config_dump`的第二个参数没有影响。

更复杂的情况（例如用户可能会修改初始提供的数据）可以通过在配置表上创建触发器来处理，触发器将负责保证被修改的行会被正确地标记。

你可通过再次调用`pg_extension_config_dump`来修改与一个配置表相关的过滤条件（这通常对于一个扩展更新脚本有用）。将一个表标记为不再是一个配置表的方法是用`ALTER EXTENSION ... DROP TABLE`将它与扩展脱离开。

注意这些表之间的外键关系将会指导这些表被`lt_dump`转储的顺序。特别地，`lt_dump`将尝试先转储被引用的表再转储引用表。由于外键关系是在`CREATE EXTENSION`时间（先于数据被载入到表中）建立的，环状依赖还没有建立。当环状依赖存在时，数据将仍然被转储，但是该转储无法被直接恢复并且必须要用户的介入。

与`serial`或者`bigserial`列相关联的序列需要被直接标记以转储它们的状态。只标记它们的父关系不足以转储它们的状态。

### 36.17.4. 扩展更新

扩展机制的一个优点是它提供了方便的方法来管理那些定义扩展中对象的 SQL 命令的更新。这是通过为扩展的安装脚本的每一个发行版本关联一个版本名称或者版本号实现的。此外，如果你希望用户能够动态地把他们的数据库从一个版本更新到下一个版本，你应该提供更新脚本来做必要的更改。更新脚本的名称遵循`extension--old_version--target_version.sql`模式（例如，`foo--1.0--1.1.sql`包含着把扩展`foo`的版本1.0修改成版本1.1的命令）。

假定有一个合适的更新脚本可用，命令ALTER EXTENSION UPDATE将把一个已安装的扩展更新到指定的新版本。更新脚本运行在与CREATE EXTENSION提供给安装脚本相同的环境中：特别是search\_path会按照相同的方式设置，并且该脚本创建的任何新对象会被自动地加入到扩展中。此外，如果脚本选择删除扩展的成员对象，它们会自动与扩展解除关联。

如果一个扩展具有二级控制文件，用于更新脚本的控制参数是那些与新目标版本相关的参数。

ALTER EXTENSION能够执行更新脚本的序列来实现一个要求的更新。例如，如果只有foo--1.0--1.1.sql和foo--1.1--2.0.sql可用，当前安装了1.0版本并且要求更新到版本2.0，ALTER EXTENSION将依次应用它们。

LightDB并不假定任何有关版本名称的性质：例如，它不知道1.1是否跟在1.0后面。它只是匹配可用的版本名称并且遵照要求应用最少更新脚本的路径进行（一个版本名称实际上可以是不含--或者前导或后缀-的字符串）。

有时提供“降级”脚本也有用，例如foo--1.1--1.0.sql允许把版本1.1相关的改变恢复原状。如果你这样做，要当心降级脚本被意外应用的可能性，因为它会得到一个较短的路径。危险的情况是，有一个跳过几个版本的“快速路径”更新脚本还有一个降级到该快速路径开始点的降级脚本。先应用降级然后再应用快速路径可能比一次升级一个版本需要更少的步骤。如果降级版本删除了任何不可替代的对象，这将会得到意想不到的结果。

要检查意料之外的更新路径，可使用这个命令：

```
SELECT * FROM pg_extension_update_paths('extension_name');
```

这会为指定的扩展显示已知的每一个可区分的版本名对，每一个版本名对还带有一个从源版本到目标版本的更新路径序列，如果没有可用的更新路径则这部份信息为NULL。该路径显示为用--分隔符的文本形式。如果你更喜欢数组格式，可以使用regexp\_split\_to\_array(path, '--')。

## 36.17.5. 用更新脚本安装扩展

一个已经存在一段时间的扩展可能存在多个版本，作者将需要为它们编写更新脚本。例如，如果你已经发布了扩展foo的版本1.0、1.1和1.2，就应该有更新脚本foo--1.0--1.1.sql和foo--1.1--1.2.sql。在LightDB 10之前，还有必要创建新的脚本文件foo--1.1.sql和foo--1.2.sql，它们直接构建比较新的扩展版本，或者新的版本无法被直接安装，而是通过先安装1.0然后更新。那种方式是无聊的重复性工作，但是现在它是不必要的了，因为CREATE EXTENSION能够自动遵循更新链。例如，如果只有脚本文件foo--1.0.sql、foo--1.0--1.1.sql和foo--1.1--1.2.sql可用，那么安装版本1.2的请求会通过按顺序运行上述三个脚本来实现。这种处理和先安装1.0然后更新到1.2是一样的（和ALTER EXTENSION UPDATE一样，如果有多条路径可用则优先选择最短的）。按这种风格安排扩展的脚本文件可以减少生产小更新所需的维护工作量。

如果以这种风格维护的扩展中使用了二级（版本相关的）控制文件，记住每个版本都需要一个控制文件，即使它没有单独的安装脚本，因为该控制文件将决定如何执行到这个版本的隐式更新。例如，如果foo--1.0.control指定有requires = 'bar'，但foo的其他控制文件没有这样做，在从1.0更新到另一个版本时，该扩展对bar的依赖将被删除。

## 36.17.6. Security Considerations for Extensions

广泛分布的扩展应该很少考虑它们所占用的数据库。因此，以一种不会被基于搜索路径的攻击所破坏的安全风格编写扩展提供的函数是合适的。

将superuser设置为真的扩展还必须考虑在其安装和更新脚本中执行的操作的安全危害。对于恶意用户来说，创建特洛伊木马对象并不非常困难，这些木马对象会影响到后续粗心编写的扩展脚本的执行，从而允许该用户获得超级用户特权。

如果扩展标记为 `trusted`，则安装用户可以选择其安装模式，安装用户可能会有意使用不安全的模式，希望获得超级用户权限。因此，从安全角度来看，受信任的扩展是非常易受攻击的，并且必须仔细检查其所有脚本命令，以确保不会出现危害。

关于安全地编写函数的建议在下面的Section 36.17.6.1中，关于安全编写安装脚本的建议在Section 36.17.6.2中提供。

### 36.17.6.1. Security Considerations for Extension Functions

扩展提供的SQL-language和PL-language 函数在执行时面临基于搜索路径的攻击风险，因为解析这些函数发生在执行时，而不是创建时。

CREATE FUNCTION参考页包含有关安全地编写SECURITY DEFINER函数的建议。将这些技术应用于扩展提供的任何函数是一种很好的做法，因为该函数可能由具有高权限的用户调用。

如果无法将`search_path`设置为仅包含安全模式，假设每个无资格的名称都可以解析为恶意用户定义的对象。小心隐式依赖于`search_path` 的构造；例如，IN和CASE expression WHEN总是使用搜索路径选择操作符。在它们的位置，使用OPERATOR(schema.=) ANY 和 CASE WHEN expression。

通用扩展通常不应假定它已安装到安全模式中，这意味着即使是对其自身对象的模式限定引用也不是完全无风险的。例如，如果扩展定义了一个函数`myschema.myfunc(bigint)`，则诸如`myschema.myfunc(42)`此类的调用可以被恶意函数`myschema.myfunc(integer)`捕获。请注意，函数和操作符形参的数据类型与声明的实参类型完全匹配，必要时可使用显式强制转换。

### 36.17.6.2. Security Considerations for Extension Scripts

编写扩展安装或更新脚本以防止脚本执行时发生基于搜索路径的攻击。如果脚本中的对象引用可以解析到脚本作者预期之外的其他对象，则可能会立即发生妥协，或者在稍后使用错误定义的扩展对象时发生妥协。

DDL 命令例如 CREATE FUNCTION 和 CREATE OPERATOR CLASS 通常是安全的，但要注意任何具有通用表达式作为组件的命令。例如，CREATE VIEW 需要经过审查，就像 CREATE FUNCTION中的 DEFAULT 表达式一样。

有时，扩展脚本可能需要执行通用 SQL，例如，通过DDL无法进行目录调整。小心地使用安全操作执行此类`search_path`；不要相信由 CREATE/ALTER EXTENSION 提供的路径是安全的。最佳做法是暂时设置`search_path` 到 `'pg_catalog, pg_temp'`，并根据需要显式插入对扩展安装架构的引用。（这种做法可能也有助于创建视图。）示例可以在LightDB源代码分发版本的contrib模块中找到。

交叉扩展引用极难做到完全安全，部分由于不确定另一个扩展位于哪个模式中。如果两个扩展都安装在同一模式中，则危险会减少，因为在安装时间中无法将恶意对象置于引用的扩展`search_path`。但是，目前没有任何机制要求这样做。目前，最佳做法是，如果扩展依赖于另一个扩展，则不要标记它受信任的扩展，除非该扩展始终安装在`pg_catalog`中。

## 36.17.7. 扩展实例

这里是一个只用SQL的扩展的完整例子，一个两个元素的组合类型，它可以在它的槽（命名为“k”和“v”）中存储任何类型的值。非文本值会被自动强制为文本进行存储。

脚本文件`pair--1.0.sql`看起来像这样：

```
-- 如果脚本是由 ltsql 而不是 CREATE EXTENSION 执行，则报错
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );
```

```

CREATE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, FUNCTION = pair);

-- "SET search_path" 容易操作，但限定名称更好。
CREATE FUNCTION lower(pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;
SET search_path = pg_temp;

CREATE FUNCTION pair_concat(pair, pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW($1.k OPERATOR(pg_catalog.||) $2.k,
               $1.v OPERATOR(pg_catalog.||) $2.v)::@extschema@.pair;';

```

控制文件pair.control看起来像这样：

```

# pair 扩展
comment = 'A key/value pair data type'
default_version = '1.0'
# cannot be relocatable because of use of @extschema@
relocatable = false

```

虽然你几乎不会需要一个 `makefile` 来安装这两个文件到正确的目录，你还是可以使用一个Makefile：

```

EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = lt_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)

```

这个 `makefile` 依赖于PGXS，它在Section 36.18中描述。命令`make install`将把控制和脚本文件安装到`lt_config`报告的正确的目录中。

一旦文件被安装，使用`CREATE EXTENSION`命令就可以把对象载入到任何特定的数据库中。

## 36.18. 扩展的构建基础设施

如果你正在考虑发布你的LightDB扩展模块，为它们建立一个可移植的构建系统实在是相当困难。因此LightDB安装为扩展提供了一种被称为PGXS构建基础设施，因此简单的扩展模块能够在已经安装的服务器上简单地编译。PGXS主要是为了包括 C 代码的扩展而设计，不过它也能用于纯 SQL 的扩展。注意PGXS并不想成为一种用于构建任何与LightDB交互的软件的通用构建系统框架。它只是简单地把简单服务器扩展模块的公共构建规则自动化。对于更复杂的包，你可能需要编写你自己的构建系统。

要把PGXS基础设施用于你的扩展，你必须编写一个简单的 `makefile`。在这个 `makefile` 中，你需要设置一些变量并且把它们包括在全局的PGXS `makefile` 中。这里有一个例子，它构建一个名为`isbn_issn`的扩展模块，其中包括一个含有 C 代码的共享库、一个扩展控制文件、一个 SQL 脚本、一个包括文件（仅当其他模块可能需要通过调用而不是SQL访问这个扩展的函数时才需要）以及一个文档文件：

```

MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn
HEADERS_isbn_issn = isbn_issn.h

PG_CONFIG = lt_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)

```

最后三行应该总是相同的。在这个文件的前面部分，你要对变量赋值或者增加自定义的make规则。

设置这三个变量之一来指定要构建什么：

MODULES

要从源文件构建的具有相同词干的共享库对象的列表（不要在这个列表中包括库后缀）

MODULE\_big

一个要从多个源文件中构建的共享库（在OBS中列出对象文件）

PROGRAM

一个要构建的可执行程序（在OBS中列出对象文件）

还可以设置下列变量：

EXTENSION

扩展名称；你必须为每一个名称提供一个extension.control文件，它将被安装到prefix/share/extension中

MODULEDIR

subdirectory of prefix/share的子目录，DATA 和 DOCS 文件会被安装到其中（如果没有设置，设置了EXTENSION时默认为extension，没有设置EXTENSION时默认为contrib）

DATA

要安装到prefix/share/\$MODULEDIR中的随机文件

DATA\_built

要安装到prefix/share/\$MODULEDIR中的随机文件，它们需要先被构建

DATA\_TSEARCH

要安装到prefix/share/tsearch\_data中的随机文件

DOCS

要安装到prefix/doc/\$MODULEDIR中的随机文件

HEADERS

HEADERS\_built

要（构建并且）安装在prefix/include/server/\$MODULEDIR/\$MODULE\_big下面的文件。

和DATA\_built不同，HEADERS\_built中的文件不会被clean目标移除，如果想要移除它们，把它们也加入到EXTRA\_CLEAN或者增加自己的规则来做这件事。

HEADERS\_\$MODULE  
HEADERS\_built\_\$MODULE

要安装（如果指定了构建则在构建之后安装）在prefix/include/server/\$MODULEDIR/\$MODULE之下的文件，这里\$MODULE必须是一个在MODULES or MODULE\_big中用到的模块名。

和DATA\_built不同，HEADERS\_built\_\$MODULE中的文件不会被clean目标移除，如果想要移除它们，把它们也加入到EXTRA\_CLEAN或者增加自己的规则来做这件事。

可以为同一个模块同时使用这两个变量或者两者的任意组合，除非你在MODULES列表中有两个模块名称仅有前缀built\_上的区别，因为那样会导致歧义。在那种情况下（还好不太可能），应该仅使用HEADERS\_built\_\$MODULE变量。

SCRIPTS

要安装到prefix/bin中的脚本文件（非二进制）

SCRIPTS\_built

要安装到prefix/bin中的脚本文件（非二进制），它们需要先被构建

REGRESS

回归测试案例（不带后缀）的列表，见下文

REGRESS\_OPTS

要传递给pg\_regress的附加开关

ISOLATION

隔离测试用例列表，请参阅下文了解更多详细信息。

ISOLATION\_OPTS

要传递给pg\_isolation\_regress的附加开关

TAP\_TESTS

是否需要运行 TAP 测试的开关定义，请参阅下文

NO\_INSTALLCHECK

不定义installcheck目标，如果测试要求特殊的配置就会很有用，或者不使用pg\_regress

EXTRA\_CLEAN

要在make clean中移除的额外文件

PG\_CPPFLAGS

将被加到CPPFLAGS前面

PG\_CFLAGS

将被加到CFLAGS后面

PG\_CXXFLAGS

将被加到CXXFLAGS后面



PG\_LDFLAGS

将被加到LDFLAGS前面

PG\_LIBS

将被加到PROGRAM链接行

SHLIB\_LINK

将被加到MODULE\_big链接行

PG\_CONFIG

要在其中构建的LightDB安装的lt\_config程序的路径（通常只用在你的PATH中的第一个lt\_config）

把这个 makefile 作为Makefile放在保存你扩展的目录中。然后你可以执行make进行编译，并且接着make install来安装你的模块。默认情况下，该模块会为在你的PATH中找到的第一个lt\_config程序所对应的LightDB安装编译和安装。你可以通过在 makefile 中或者make命令行中设置PG\_CONFIG指向另一个lt\_config程序来使用一个不同的安装。

如果你想保持编译目录独立，你也可以在你的扩展所属的源代码树之外的目录中运行make。这个过程也被称为一个 VPATH 编译。下面是做法：

```
mkdir build_dir
cd build_dir
make -f /path/to/extension/source/tree/Makefile
make -f /path/to/extension/source/tree/Makefile install
```

此外，你可以以对核心代码所作的方式一样为 VPATH 设置一个目录。一种方式是使用核心脚本 config/prep\_buildtree。一旦这样做，你可以这样设置 make变量VPATH：

```
make VPATH=/path/to/extension/source/tree
make VPATH=/path/to/extension/source/tree install
```

这个过程可以在很多种目录布局下工作。

列举在REGRESS变量中的脚本会被用来对你的扩展进行回归测试，回归测试可以在做完make install之后用make installcheck调用。要让这能够工作，你必须已经有一个运行着的LightDB服务器。列举在REGRESS中的脚本文件必须在你扩展目录的名为sql/的子目录中出现。这些文件必须带有扩展.sql，但扩展不能被包括在 makefile 的REGRESS列表中。对每一个测试还应该在名为expected/的子目录中有一个包含预期输出的文件，它具有和脚本文件相同的词干并带有扩展.out。make installcheck会用ltsql执行每一个测试脚本，并且将得到结果输出与相应的预期输出比较。任何区别都将以diff -c格式写入到文件regression.diffs中。注意尝试运行一个不带预期文件的测试将被报告为“故障”，因此确保你拥有所有的预期文件。

ISOLATION变量中列出的脚本用于测试强调与模块并发会话的行为，可以在make install之后通过make installcheck 调用。要实现这个工作，你必须有一个正在运行的LightDB服务器。ISOLATION中列出的脚本文件必须显示在扩展名目录中名为 specs/的子目录中。这些文件必须具备扩展名.spec，并且不得包含在 makefile 中的ISOLATION列表中。对于每个测试，在名为expected/的子目录中还应该有一个包含预期输出的文件，并且具有相同的词干和扩展名.out。make installcheck执行每个测试脚本，并将结果输出与匹配的预期文件进行比较。任何差异都将以diff -c的格式写入到output\_iso/regression.diffs文件中。请注意，尝试运行缺少其预期文件的测试将会报告“trouble”，因此请确保你具有全部的预期文件。

TAP\_TESTS 启用TAP测试。每个运行中的数据都存在于名为 tmp\_check/的子目录中。更多详细信息，请参阅Section 30.4

Tip

创建预期文件最简单的方法是创建空文件，然后做一次测试运行（这当然将报告区别）。检查在results/目录中找到的实际结果文件（对于 REGRESS中的测试），或output\_iso/results/ 目录（对于ISOLATION）中的测试，如果它们符合你的预期则把它们复制到expected/中。

---

## Chapter 37. 触发器

本章提供有关编写触发器函数的一般信息。触发器函数可以使用大多数可用的过程性语言编写，包括 PL/pgSQL (Chapter 41), PL/oraSQL (Chapter 42)。阅读本章后，您应该查阅您喜欢的过程性语言的章节，以了解在其中编写触发器的特定于语言的详细信息。

也可以使用C编写一个触发器函数，尽管大部分人觉得使用一种过程性语言更简单。当前不能用纯SQL函数语言来编写触发器函数。

### 37.1. 触发器行为概述

一个触发器声明了当执行一种特定类型的操作时数据库应该自动执行一个特殊的函数。触发器可以被附加到表（分区的或者不分区的）、视图和外部表。

在表和外部表上，触发器可以被定义为在 INSERT、UPDATE或 DELETE操作之前或之后被执行，可以为每个SQL语句被执行一次或者为每个修改的行 被执行一次。UPDATE 触发器可以进一步地设置为只针对UPDATE 语句的SET子句的特定列出发。触发器也可以被 TRUNCATE语句触发。如果一个触发器事件发生，触发器函数会在适当的事件被调用来处理该事件。

在视图上，触发器可以被定义来取代INSERT、UPDATE或 DELETE操作的执行。这种INSTEAD OF触发器对视图中需要被修改的每一行触发一次。触发器函数的职责是对视图的底层基本表执行必要的修改，并且在合适的时候返回被修改的行以便显示在视图中。视图上的触发器也可以被定义为对每个SQL语句执行一次，在INSERT\UPDATE或DELETE操作之前或之后。不过，只有在该视图上还有一个INSTEAD OF触发器时，上述那些触发器才会被触发。否则，以该视图为目标的任何语句都必须被重写成一个影响其底层基表的语句，然后附着在那些基表上的触发器将会被引发。

触发器函数必须在触发器本身被创建之前被定义好。触发器函数必须被定义成一个没有参数的函数，并且返回类型为trigger（触发器函数通过一个特殊传递的TriggerData结构作为其输入，而不是以普通函数参数的形式）。

一旦一个合适的触发器函数被创建，就可以使用CREATE TRIGGER建立触发器。同一个触发器函数可以被用于多个触发器。

LightDB同时提供每行的触发器和每语句的触发器。对于一个每行的触发器，对于触发触发器的语句所修改的每一行都会调用一次触发器函数。相反，一个每语句的触发器对于其触发语句只被调用一次，而不管该语句影响了多少行。特别地，一个不影响任何行的语句仍然会导致任何可用每语句的触发器的执行。这两类触发器有时也分别被称作行级触发器和语句级触发器。TRUNCATE上的触发器只能被定义在语句级。

触发器也可以根据它们是否在操作之前、之后触发，或者被触发来取代操作来分类。它们分别指BEFORE触发器、AFTER 触发器以及INSTEAD OF触发器。语句级BEFORE触发器在语句开始做任何事情之前被触发，而语句级AFTER触发器则在语句做完所有事情之后被触发。这些触发器类型可以被定义在表、视图或外部表上。行级BEFORE触发器在每一个行被操作之前被触发，而行级AFTER触发器在语句结束之后被触发（但在任何语句级AFTER触发器之前）。这些触发器类型只能被定义在表和外部表上，但不能定义在视图上。INSTEAD OF触发器只能被定义在视图上，并且只能定义在行级，当视图中的每一行被标识为需要被操作时，它们会立即触发。

一个AFTER触发器的执行可以推迟到事务的最后，而不是语句的最后，如果它被定义为constraint trigger。在所有情况下，触发器作为触发它的语句的同一个事务的一部分来执行，所以无论语句或触发器导致错误，两者的后果都将会回滚。

一个以继承或者分区层次中父表为目标的语句不会导致受影响的子表的语句级触发器被引发，只有父表的语句级触发器会被引发。不过，受影响的子表的行级触发器将被引发。

如果一个INSERT包含ON CONFLICT DO UPDATE子句并且引用了EXCLUDED列，有可能所有行级BEFORE INSERT触发器和所有行级 BEFORE UPDATE触发器的效果可能会以一种对于 被更

新行最终状态透明的方式被应用。不过，对于要执行的两种集合的行级 BEFORE 触发器都不需要有 EXCLUDED 列引用。当同时有行级 BEFORE INSERT 和 BEFORE UPDATE 触发器影响被插入/更新的行时（如果在两者不幂等时修改或多或少地等价，这仍可能是有问题的），应该考虑可能出现的意料之外的结果。注意在指定了 ON CONFLICT DO UPDATE 时，不管有没有行被 UPDATE 影响（并且不管是否采用了其他 UPDATE 路径），语句级 UPDATE 都将被执行。一个带有 ON CONFLICT DO UPDATE 子句的 INSERT 将首先执行语句级 BEFORE INSERT，然后执行语句级 BEFORE UPDATE 触发器，接着是语句级 AFTER UPDATE 触发器，最后是语句级 AFTER INSERT 触发器。

如果一个分区表上的 UPDATE 导致一行移动到另一个分区，它将被从原始分区 DELETE 掉然后再 INSERT 到新分区中。在这种情况下，原始分区上所有的行级 BEFORE UPDATE 触发器和所有行级 BEFORE DELETE 触发器会被引发。然后目标分区上所有的行级 BEFORE INSERT 触发器会被引发。当所有这些触发器都影响被移动的行时，应该对令人惊讶的结果有心理准备。至于 AFTER ROW 触发器，AFTER DELETE 和 AFTER INSERT 触发器会被应用，但 AFTER UPDATE 触发器不会被应用，因为 UPDATE 已经被转换成了一个 DELETE 和一个 INSERT。对于语句级触发器，即便发生行移动，DELETE 和 INSERT 触发器也都不会被引发，只有 UPDATE 语句中用到的目标表上的 UPDATE 触发器将被引发。

被语句级触发器调用的触发器函数应该总是返回 NULL。根据行级触发器的选择，被其调用的触发器函数可以返回一个表行（类型 HeapTuple 的一个值）给执行器。在一个操作前触发的行级触发器有下列选择：

- 它可以返回 NULL 来跳过对当前行的操作。这指示执行器不要执行调用触发器的行级操作（对一个特定表行的插入、修改或删除）。
- 仅对行级 INSERT 和 UPDATE 触发器来说，被返回的行称为将要被插入的行或者替代将被更新的行。这允许触发器函数修改将要被插入或更新的行。

一个无意导致任何这些行为的行级 BEFORE 触发器必须小心地它的结果，使之和被传入的行一样（即，INSERT 和 UPDATE 触发器的 NEW 行，DELETE 触发器的 OLD 行）。

一个行级 INSTEAD OF 触发器可以返回 NULL 来指示它没有修改任何来自于视图底层基表的数据，也可以返回被传入的视图行（INSERT 和 UPDATE 操作的 NEW 行，或者 DELETE 操作的 OLD 行）。一个非空返回值被用于标志触发器在视图中执行了必须的数据修改。这将会导致被命令修改的行计数被增加。仅对于 INSERT 和 UPDATE 操作，触发器可能会在返回 NEW 行之前对其进行修改。这将会改变 INSERT RETURNING 或 UPDATE RETURNING 返回的数据，并在视图无法正确地显示提供给它的相同数据时有用。

对于在一个操作之后触发的行级触发器，返回值会被忽略，因此它们可以返回 NULL。

一些情况适用于生成的列。存储生成的列在 BEFORE 触发器之后和 AFTER 触发器之前计算。因此，生成的值可以在 AFTER 触发器中检查。在 BEFORE 触发器中，OLD 行包含旧的生成的值，正如人们所期待的，但 NEW 行尚未包含新的生成值并且不应访问。在 C 语言界面中，此时列的内容还没有被定义；在 BEFORE 触发器中，高级别编程语言应阻止访问 NEW 行中存储生成的列。在 BEFORE 触发器中更改到生成列的值将被忽略并覆盖。

如果为同一个关系上的同一事件定义了超过一个触发器，它们将按照其名称的字母表顺序被触发。在 BEFORE 和 INSTEAD OF 触发器的情况下，每一个触发器返回的可能被修改的行将成为下一个触发器的输入。如果任何一个 BEFORE 或 INSTEAD OF 触发器返回 NULL，该操作将在该行上被禁用并且对于该行不会触发后续的触发器。

一个触发器定义也能指定一个布尔的 WHEN 条件，它将被测试来看该触发器是否应该被触发。在行级触发器中，WHEN 条件可以检查该行的旧列值和/或新列值（语句级触发器也有 WHEN 条件，但是该特性对它们不太有用）。在一个 BEFORE 触发器中，WHEN 条件只是在该函数被或者将被执行前计算，因此使用 WHEN 条件与在该触发器函数的开始测试相同的条件没有本质区别。不过，在一个 AFTER 触发器中，WHEN 条件只是在行更新发生之后被计算，并且它决定在语句的末尾一个事件是否被排队来触发该触发器。因此当一个 AFTER 触发器的 WHEN 不返回真时，在语句的末尾没有必要将一个事件进行排队，也没有必要重新取出该行。如果触发器只对少数行触发，这可以使得修改很多行的语句明显加快。INSTEAD OF 触发器不支持 WHEN 条件。

通常，行级BEFORE被用来检查或修改即将被插入或更新的数据。例如，一个BEFORE触发器可以被用来把当前时间插入到一个timestamp列中，或者检查该行的两个元素之间是否一致。行级AFTER触发器大多数被用来将更新传播到其他表，或者针对其他表进行一致性检查。进行这种工作分工的原因是，一个AFTER触发器可以肯定它看到的是该行的最终值，而一个BEFORE触发器则不能，因为还可能有其他BEFORE触发器在它之后触发。如果你不知道让一个触发器是BEFORE或AFTER，则BEFORE形式更加有效，因为关于该操作的信息直到语句的末尾都不需要被保存。

如果一个触发器函数执行 SQL 命令，则这些命令可能会再次引发触发器。这就是所谓的级联触发器。对于级联的层数没有直接的限制。级联有可能会对同一个触发器的递归调用。例如，一个INSERT触发器可能执行一个向同一个表插入一个额外行的命令，这就导致该INSERT触发器被再次引发。所以在这种情形下，触发器程序员应该负责避免无限递归。

在定义一个触发器时，可以为它指定参数。在触发器定义中包括参数的目的是允许具有相似需求的不同触发器调用同一个函数。例如，可能有一个一般性的触发器函数，它需要两个列名作为参数，一个放当前用户而另一个放当前时间戳。在正确编写的情况下，这个触发器函数应该独立于它所触发的表。因此同一个函数可以被用于具有适当列的任意表上的INSERT事件，这样做的用途之一是可以自动追踪一个交易表中记录的创建。如果被定义成一个UPDATE触发器，它也可以被用来追踪最新的更新事件。

每一种支持触发器的编程语言都有自己的方法来让触发器输入数据对触发器函数可用。这种输入数据包括触发器事件的类型（如INSERT或UPDATE）以及被列在CREATE TRIGGER中的任何参数。对于一个行级触发器，输入数据还包括用于INSERT和UPDATE触发器的NEW行，和/或用于UPDATE和DELETE触发器的OLD行。语句级触发器当前没有任何方法检查被语句修改的单个行。

默认情况下，语句级触发器没有办法检查该语句修改的行。但是AFTER STATEMENT触发器可以请求创建传递表，这样可以让受影响的行集合对该触发器可用。AFTER ROW触发器也可以请求传递表，这样它们可以看到表中的整个变化，同时也能看到当前引发它们的个体行中的变化。检查传递表的方法仍是取决于使用的编程语言，但是通常的方法让传递表变得像触发器函数内部发出的SQL命令能够访问的只读临时表一样。

## 37.2. 数据改变的可见性

如果你在你的触发器函数中执行 SQL 命令，并且这些命令会访问触发器所在的表，那么你需要注意数据可见性规则。因为这些规则决定了这些 SQL 命令是否将能看见引发触发器的数据改变。简单地：

- 语句级触发器遵循简单的可见性规则：一个语句所作的改变对于语句级 BEFORE触发器都不可见，而所有修改对于语句级 AFTER触发器都是可见的。
- 导致触发器被引发的数据更改（插入、更新或删除）自然对于在一个行级BEFORE触发器中执行的 SQL 命令不可见，因为它还没有发生。
- 但是，在一个行级BEFORE触发器中执行的 SQL 命令将会看见之前在同一个外层命令中所作的数据更改的效果。这里需要小心，因为这些更改时间的顺序通常是不可预测的，一个影响多行的 SQL 命令可能以任何顺序访问这些行。
- 类似地，一个行级INSTEAD OF触发器将会看见之前在同一个外层命令中INSTEAD OF触发器引发所作的更改。
- 当一个行级AFTER触发器被引发时，所有由外层命令所作的更改已经完成，并且对于该被调用的触发器函数是可见的。

如果你的触发器函数使用任何一种标准过程语言编写的，那么只有在该函数被声明为VOLATILE时上述陈述才适用。被声明为STABLE或IMMUTABLE的函数在任何情况下将不能看到由调用命令所作出的更改。

有关数据可见性规则的更多信息可见Section 43.5 Section 37.4中的例子包含了对这些规则的示范。

## 37.3. 用 C 编写触发器函数

这一节描述了一个触发器函数的接口的低层细节。只有用 C 编写触发器函数时才需要这些信息。如果你使用一种更高层的语言，那么这些细节就不需要你来处理。在大部分情况下，你应该优先考虑使用一种过程语言。每一种过程语言的文档阐述了如何使用那种语言编写一个触发器。

触发器函数必须使用“版本 1”函数管理器接口。

当一个函数被触发器管理器调用时，不会给它传递任何常规的参数，但是会有一个“context”指针传递给它，该指针指向一个TriggerData结构。C 函数可以通过执行一个宏来检查它们是否是从触发器管理器被调用：

```
CALLED_AS_TRIGGER(fcinfo)
```

它会展开成为：

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

如果这返回真，那么将fcinfo->context造型成类型TriggerData \*并且利用所指向的TriggerData结构就是安全的。该函数不能修改该TriggerData结构或者它指向的任何数据。

struct TriggerData被定义在commands/trigger.h中：

```
typedef struct TriggerData
{
    NodeTag          type;
    TriggerEvent     tg_event;
    Relation         tg_relation;
    HeapTuple        tg_trigtuple;
    HeapTuple        tg_newtuple;
    Trigger          *tg_trigger;
    TupleTableSlot  *tg_trigslot;
    TupleTableSlot  *tg_newslot;
    Tuplestorestate *tg_oldtable;
    Tuplestorestate *tg_newtable;
    const Bitmapset *tg_updatedcols;
} TriggerData;
```

其中的成员被定义如下：

type

总是T\_TriggerData.

tg\_event

描述该函数是为什么事件被调用的。你可以使用下列宏来检查tg\_event：

```
TRIGGER_FIRED_BEFORE(tg_event)
```

如果该触发器在操作前被引发则返回真。

```
TRIGGER_FIRED_AFTER(tg_event)
```

如果该触发器在操作后被引发则返回真。

TRIGGER\_FIRED\_INSTEAD(tg\_event)

如果该触发器被引发替代操作则返回真。

TRIGGER\_FIRED\_FOR\_ROW(tg\_event)

如果该触发器为一个行级事件而引发则返回真。

TRIGGER\_FIRED\_FOR\_STATEMENT(tg\_event)

如果该触发器为一个语句级事件而引发则返回真。

TRIGGER\_FIRED\_BY\_INSERT(tg\_event)

如果该触发器由一个INSERT命令引发则返回真。

TRIGGER\_FIRED\_BY\_UPDATE(tg\_event)

如果该触发器由一个UPDATE命令引发则返回真。

TRIGGER\_FIRED\_BY\_DELETE(tg\_event)

如果该触发器由一个DELETE命令引发则返回真。

TRIGGER\_FIRED\_BY\_TRUNCATE(tg\_event)

如果该触发器由一个TRUNCATE命令引发则返回真。

tg\_relation

一个结构指针，该结构描述该触发器为其引发的关系。关于这个结构的细节请参考utils/rel.h。最有趣的东西是tg\_relation->rd\_att（该关系元组的描述符）和tg\_relation->rd\_rel->relname（关系名称，该类型不是char\*而是NameData。如果你需要该名称的一个拷贝，可使用SPI\_getrelname(tg\_relation)来得到一个char\*）。

tg\_trigtuple

一个该触发器为其引发的行的指针。这是被插入、更新或删除的行。如果这个触发器是为一个INSERT或DELETE而引发，在你不想把该行替换成另一行（在INSERT的情况中）或不想跳过该操作时你应该从该函数中返回它。对于外部表上的触发器，此中的系统列值未被指定。

tg\_newtuple

如果该触发器为一个UPDATE而引发，则是一个指向该行新版本的指针。如果是为一个INSERT或DELETE而引发，则是NULL。如果事件是一个UPDATE并且你并不想用一个新的行替换这个行或者不想跳过该操作时，你必须从函数中返回它。对于外部表上的触发器，此中的系统列值未被指定。

tg\_trigger

一个指向类型为Trigger的结构的指针，定义在utils/reltrigger.h中：

```
typedef struct Trigger
{
    Oid          tgoid;
    char        *tgname;
    Oid          tgfoid;
    int16       tgtype;
    char        tgenabled;
```

```

    bool        tgisinternal;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16       *tgattr;
    char        **tgargs;
    char        *tgqual;
    char        *tgoldtable;
    char        *tgnewtable;
} Trigger;

```

其中tgname是该触发器的名称，tgnargs是tgargs中参数的数量，而tgargs是一个指向CREATE TRIGGER语句中指定的参数的指针数组。其他成员只用于内部用途。

tg\_trigtuplebuf

包含tg\_trigtuple的插槽。或者一个NULL指针，如果没有这样的元组的话。

tg\_newtuplebuf

包含tg\_trigtuple的插槽。或者一个NULL指针，如果没有这样的元组的话。

tg\_oldtable

一个指向Tuplestorestate类型的结构的指针，该结构包含格式由tg\_relation指定的零行或者多行。如果没有OLD TABLE传递关系，则为NULL指针。

tg\_newtable

一个指向Tuplestorestate类型的结构的指针，该结构包含格式由tg\_relation指定的零行或者多行。如果没有NEW TABLE传递关系，则为NULL指针。

tg\_updatedcols

对于 UPDATE 触发器，位图集指示由触发命令更新的列。通用触发器函数可以使用它来优化操作，而不必处理未更改的列。

例如，要确定属性编号为 attnum（基于 1）的列是否是此位图集的成员，请调用 bms\_is\_member(attnum - FirstLowInvalidHeapAttributeNumber, trigdata->tg\_updatedcols)。

对于UPDATE触发器以外的触发器，这将是NULL。

为了允许通过SPI发出的查询引用传递表，请参考SPI\_register\_trigger\_data。

一个触发器函数必须返回一个HeapTuple指针或一个NULL指针（不是一个 SQL 空值，也就是不会设置isNull为真）。如果你不希望修改正在被操作的行，要小心地根据情况返回tg\_trigtuple或tg\_newtuple。

## 37.4. 一个完整的触发器实例

这里有一个用 C 编写的触发器函数的非常简单的例子（用过程语言编写的触发器的例子可以在过程语言的文档中找到）。

如果该命令试图向列x中插入一个空值，函数trigf报告表ttest中的行数并且跳过实际的操作（这样该触发器会作为一个非空约束但不会中止事务）。



首先，表定义：

```
CREATE TABLE ttest (  
    x integer  
);
```

这是该触发器函数的源代码：

```
#include "postgres.h"  
#include "fmgr.h"  
#include "executor/spi.h" /* this is what you need to work with SPI */  
#include "commands/trigger.h" /* ... triggers ... */  
#include "utils/rel.h" /* ... and relations */  
  
PG_MODULE_MAGIC;  
  
PG_FUNCTION_INFO_V1(trigf);  
  
Datum  
trigf(PG_FUNCTION_ARGS)  
{  
    TriggerData *trigdata = (TriggerData *) fcinfo->context;  
    TupleDesc tupdesc;  
    HeapTuple rettuple;  
    char *when;  
    bool checknull = false;  
    bool isnull;  
    int ret, i;  
  
    /* make sure it's called as a trigger at all */  
    if (!CALLED_AS_TRIGGER(fcinfo))  
        elog(ERROR, "trigf: not called by trigger manager");  
  
    /* tuple to return to executor */  
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))  
        rettuple = trigdata->tg_newtuple;  
    else  
        rettuple = trigdata->tg_trigtuple;  
  
    /* check for null values */  
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)  
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))  
        checknull = true;  
  
    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))  
        when = "before";  
    else  
        when = "after";  
  
    tupdesc = trigdata->tg_relation->rd_att;  
  
    /* connect to SPI manager */  
    if ((ret = SPI_connect()) < 0)  
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);  
  
    /* get number of rows in table */  
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);
```

```

if (ret < 0)
    elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

/* count(*) returns int8, so be careful to convert */
i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                SPI_tuptable->tupdesc,
                                1,
                                &isnull));

elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

SPI_finish();

if (checknull)
{
    SPI_getbinval(rettuple, tupdesc, 1, &isnull);
    if (isnull)
        rettuple = NULL;
}

return PointerGetDatum(rettuple);
}

```

在你编译了该源代码（见Section 36.10.5 之后，声明该函数和触发器：

```

CREATE FUNCTION trigf() RETURNS trigger
AS 'filename'
LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
FOR EACH ROW EXECUTE FUNCTION trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
FOR EACH ROW EXECUTE FUNCTION trigf();

```

现在你可以测试该触发器的操作：

```

=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
      ^^^^^^^^

remember what we said about visibility.

INSERT 167793 1
vac=> SELECT * FROM ttest;

```

```

x
---
1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                     ~~~~~

remember what we said about visibility.

INSERT 167794 1
=> SELECT * FROM ttest;
x
---
1
2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
x
---
1
4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                     ~~~~~

remember what we said about visibility.

DELETE 2
=> SELECT * FROM ttest;
x
---
(0 rows)

```

在src/test/regress/regress.c和spi中有更多复杂的例子。

---

## Chapter 38. 事件触发器

为了对Chapter 3中讨论的触发器机制加以补充，LightDB也提供了事件触发器。和常规触发器（附着在一个表上并且只捕捉DML事件）不同，事件触发器对一个特定数据库来说是全局的，并且可以捕捉DDL事件。

和常规触发器相似，可以用任何包括了事件触发器支持的过程语言或者C编写事件触发器，但是不能用纯SQL编写。

### 38.1. 事件触发器行为总览

只要与一个事件触发器相关的事件在事件触发器所在的数据库中发生，该事件触发器就会被引发。当前支持的事件是 `ddl_command_start`、`ddl_command_end`、`table_rewrite`和`sql_drop`。未来的发行版中可能会增加对更多事件的支持。

`ddl_command_start`事件就在CREATE、ALTER、DROP、SECURITY LABEL、COMMENT、GRANT或者REVOKE命令的执行之前发生。在事件触发器引发前不会做受影响对象是否存在的检查。不过，一个例外是，这个事件不会为目标是共享对象——数据库、角色以及表空间——的DDL命令发生，也不会为目标是事件触发器的DDL命令发生。事件触发器机制不支持这些对象类型。`ddl_command_start`也会在SELECT INTO命令的执行之前发生，因为这等价于CREATE TABLE AS。

`ddl_command_end`事件就在同一组命令的执行之后发生。为了得到发生的DDL操作的更多细节，可以从 `ddl_command_end`事件触发器代码中使用集合返回函数 `pg_event_trigger_ddl_commands()`（见Section 9.29）。注意该触发器是在那些动作已经发生之后（但是在事务提交前）引发，并且因此系统目录会被读作已更改。

`sql_drop`事件为任何删除数据库对象的操作在 `ddl_command_end`事件触发器之前发生。要列出已经被删除的对象，可以从`sql_drop`事件触发器代码中使用集合返回函数 `pg_event_trigger_dropped_objects()`见Section 9.29。注意该触发器是在对象已经从系统目录删除以后执行，因此不能再查看它们。

`table_rewrite`事件在表被命令ALTER TABLE和ALTER TYPE的某些动作重写之前发生。虽然其他控制语句（例如CLUSTER和VACUUM）也可以用来重写表，但是它们不会触发`table_rewrite`事件。

不能在一个中止的事务中执行事件触发器（其他函数也一样）。因此，如果一个DDL命令出现错误失败，将不会执行任何相关的 `ddl_command_end`触发器。反过来，如果一个`ddl_command_start`触发器出现错误失败，将不会引发进一步的事件触发器，并且不会尝试执行该命令本身。类似地，如果一个 `ddl_command_end`触发器出现错误失败，DDL命令的效果将被回滚，就像其他包含事务中止的情况中那样。

Section 38.2中有事件触发器机制所支持的完整命令列表。

事件触发器通过命令CREATE EVENT TRIGGER创建。为了创建一个事件触发器，你必须首先创建一个有特殊返回类型 `event_trigger`的函数。这个函数不一定需要返回一个值，该返回类型仅仅是作为一种信号表示该函数要被作为一个事件触发器调用。

如果对于一个特定的事件定义了多于一个事件触发器，它们将按照触发器名称的字母表顺序被引发。

一个触发器定义也可以指定一个WHEN条件，这样事件触发器（例如`ddl_command_start`触发器）就可以只对用户希望介入的特定命令触发。这类触发器的通常用法是用于限制用户可能执行的DDL操作的范围。

### 38.2. 事件触发器触发矩阵

Table 38.列出了所有命令的事件触发器支持情况。

Table 38.1. 支持事件触发器的命令标签

命令标签	ddl_ command_ start	ddl_ command_ end	sql_drop	table_ rewrite	注释
ALTER AGGREGATE	X	X	-	-	
ALTER COLLATION	X	X	-	-	
ALTER CONVERSION	X	X	-	-	
ALTER DOMAIN	X	X	-	-	
ALTER DEFAULT PRIVILEGES	X	X	-	-	
ALTER EXTENSION	X	X	-	-	
ALTER FOREIGN DATA WRAPPER	X	X	-	-	
ALTER FOREIGN TABLE	X	X	X	-	
ALTER FUNCTION	X	X	-	-	
ALTER LANGUAGE	X	X	-	-	
ALTER LARGE OBJECT	X	X	-	-	
ALTER MATERIALIZED VIEW	X	X	-	-	
ALTER OPERATOR	X	X	-	-	
ALTER OPERATOR CLASS	X	X	-	-	
ALTER OPERATOR FAMILY	X	X	-	-	
ALTER POLICY	X	X	-	-	
ALTER PROCEDURE	X	X	-	-	
ALTER PUBLICATION	X	X	-	-	
ALTER ROUTINE	X	X	-	-	
ALTER SCHEMA	X	X	-	-	
ALTER SEQUENCE	X	X	-	-	
ALTER SERVER	X	X	-	-	
ALTER STATISTICS	X	X	-	-	
ALTER SUBSCRIPTION	X	X	-	-	
ALTER TABLE	X	X	X	X	
ALTER TEXT SEARCH CONFIGURATION	X	X	-	-	
ALTER TEXT SEARCH DICTIONARY	X	X	-	-	
ALTER TEXT SEARCH PARSER	X	X	-	-	
ALTER TEXT SEARCH TEMPLATE	X	X	-	-	
ALTER TRIGGER	X	X	-	-	
ALTER TYPE	X	X	-	X	
ALTER USER MAPPING	X	X	-	-	
ALTER VIEW	X	X	-	-	

事件触发器

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注释
COMMENT	X	X	-	-	仅对本地对象
CREATE ACCESS METHOD	X	X	-	-	
CREATE AGGREGATE	X	X	-	-	
CREATE CAST	X	X	-	-	
CREATE COLLATION	X	X	-	-	
CREATE CONVERSION	X	X	-	-	
CREATE DOMAIN	X	X	-	-	
CREATE EXTENSION	X	X	-	-	
CREATE FOREIGN DATA WRAPPER	X	X	-	-	
CREATE FOREIGN TABLE	X	X	-	-	
CREATE FUNCTION	X	X	-	-	
CREATE INDEX	X	X	-	-	
CREATE LANGUAGE	X	X	-	-	
CREATE MATERIALIZED VIEW	X	X	-	-	
CREATE OPERATOR	X	X	-	-	
CREATE OPERATOR CLASS	X	X	-	-	
CREATE OPERATOR FAMILY	X	X	-	-	
CREATE POLICY	X	X	-	-	
CREATE PROCEDURE	X	X	-	-	
CREATE PUBLICATION	X	X	-	-	
CREATE RULE	X	X	-	-	
CREATE SCHEMA	X	X	-	-	
CREATE SEQUENCE	X	X	-	-	
CREATE SERVER	X	X	-	-	
CREATE STATISTICS	X	X	-	-	
CREATE SUBSCRIPTION	X	X	-	-	
CREATE TABLE	X	X	-	-	
CREATE TABLE AS	X	X	-	-	
CREATE TEXT SEARCH CONFIGURATION	X	X	-	-	
CREATE TEXT SEARCH DICTIONARY	X	X	-	-	
CREATE TEXT SEARCH PARSER	X	X	-	-	
CREATE TEXT SEARCH TEMPLATE	X	X	-	-	
CREATE TRIGGER	X	X	-	-	
CREATE TYPE	X	X	-	-	

命令标签	ddl_ command_ start	ddl_ command_ end	sql_drop	table_ rewrite	注释
CREATE USER MAPPING	X	X	-	-	
CREATE VIEW	X	X	-	-	
DROP ACCESS METHOD	X	X	X	-	
DROP AGGREGATE	X	X	X	-	
DROP CAST	X	X	X	-	
DROP COLLATION	X	X	X	-	
DROP CONVERSION	X	X	X	-	
DROP DOMAIN	X	X	X	-	
DROP EXTENSION	X	X	X	-	
DROP FOREIGN DATA WRAPPER	X	X	X	-	
DROP FOREIGN TABLE	X	X	X	-	
DROP FUNCTION	X	X	X	-	
DROP INDEX	X	X	X	-	
DROP LANGUAGE	X	X	X	-	
DROP MATERIALIZED VIEW	X	X	X	-	
DROP OPERATOR	X	X	X	-	
DROP OPERATOR CLASS	X	X	X	-	
DROP OPERATOR FAMILY	X	X	X	-	
DROP OWNED	X	X	X	-	
DROP POLICY	X	X	X	-	
DROP PROCEDURE	X	X	X	-	
DROP PUBLICATION	X	X	X	-	
DROP ROUTINE	X	X	X	-	
DROP RULE	X	X	X	-	
DROP SCHEMA	X	X	X	-	
DROP SEQUENCE	X	X	X	-	
DROP SERVER	X	X	X	-	
DROP STATISTICS	X	X	X	-	
DROP SUBSCRIPTION	X	X	X	-	
DROP TABLE	X	X	X	-	
DROP TEXT SEARCH CONFIGURATION	X	X	X	-	
DROP TEXT SEARCH DICTIONARY	X	X	X	-	
DROP TEXT SEARCH PARSER	X	X	X	-	
DROP TEXT SEARCH TEMPLATE	X	X	X	-	
DROP TRIGGER	X	X	X	-	
DROP TYPE	X	X	X	-	

命令标签	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注释
DROP USER MAPPING	X	X	X	-	
DROP VIEW	X	X	X	-	
GRANT	X	X	-	-	只对本地对象
IMPORT FOREIGN SCHEMA	X	X	-	-	
REFRESH MATERIALIZED VIEW	X	X	-	-	
REVOKE	X	X	-	-	只对本地对象
SECURITY LABEL	X	X	-	-	只对本地对象
SELECT INTO	X	X	-	-	

### 38.3. 用 C 编写事件触发器函数

这一节描述了事件触发器函数接口的低层细节。只有在用 C 编写事件触发器函数时才需要用到这里的信息。如果使用更高层的语言，那么这些细节已经被处理好了。在大部分情况下都应该优先考虑使用过程语言来编写你的事件触发器。每一种过程语言的文档都解释了如何用它们编写事件触发器。

事件触发器函数必须使用“版本 1”的函数管理器接口。

当一个函数被事件触发器管理器调用时，向它传递的并不是普通参数，而是一个指向 EventTriggerData 结构的“context”指针。C 函数可以通过执行以下宏来检查它是否被事件触发器管理器调用：

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

这个宏会被扩展为：

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, EventTriggerData))
```

如果这个宏返回真，那么就可以安全地把 `fcinfo->context` 造型为类型 `EventTriggerData *` 并且使用所指向的 `EventTriggerData` 结构。函数不能修改 `EventTriggerData` 结构以及它指向的任何内容。

`struct EventTriggerData` 在 `commands/event_trigger.h` 中定义：

```
typedef struct EventTriggerData
{
    NodeTag    type;
    const char *event;      /* 事件名称 */
    Node       *parsetree; /* 解析树 */
    CommandTag tag;        /* 命令标签 */
} EventTriggerData;
```

其中的成员定义如下：

```
type
```

总是 `T_EventTriggerData`。



event

描述要为其调用这个函数的事件，可以是“ddl\_command\_start”、“ddl\_command\_end”、“sql\_drop”、“table\_rewrite”之一。这些事件的含义请见Section 38.1

parsetree

该命令的解析树的指针。其细节可以参考 LightDB 的源代码。解析树结构可能会在未经通知的情况下改变。

tag

与事件触发器的事件相关联的命令标签，例如“CREATE FUNCTION”。

一个事件触发器函数必须返回一个NULL指针（不是一个 SQL 空值，也就是不要把 isNull 设置为真）。

## 38.4. 一个完整的事件触发器例子

这里是一个用 C 编写的事件触发器函数的简单例子（用过程语言编写的触发器例子可以在过程语言的文档中找到）。

函数nodd1在每一次被调用时抛出一个异常。事件触发器定义把该函数和ddl\_command\_start事件关联在了一起。其效果就是所有 DDL 命令（除Section 38.中提到的例外）都被阻止运行。

这是该触发器函数的源代码：

```
#include "postgres.h"
#include "commands/event_trigger.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(nodd1);

Datum
nodd1(PG_FUNCTION_ARGS)
{
    EventTriggerData *trigdata;

    if (!CALLED_AS_EVENT_TRIGGER(fcinfo)) /* internal error */
        elog(ERROR, "not fired by event trigger manager");

    trigdata = (EventTriggerData *) fcinfo->context;

    ereport(ERROR,
            (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
             errmsg("command \"%s\" denied", trigdata->tag)));

    PG_RETURN_NULL();
}
```

在编译了源代码（见Section 36.10.5 后，声明函数和触发器：

```
CREATE FUNCTION nodd1() RETURNS event_trigger
AS 'nodd1' LANGUAGE C;
```

```
CREATE EVENT TRIGGER nodd1 ON ddl_command_start
EXECUTE FUNCTION nodd1();
```

现在你可以测试该触发器的操作：

```
=# \dy
                                List of event triggers
 Name |          Event          | Owner | Enabled | Function | Tags
-----+-----+-----+-----+-----+-----
 nodd1 | ddl_command_start      | dim   | enabled | nodd1    |
(1 row)
```

```
=# CREATE TABLE foo(id serial);
ERROR:  command "CREATE TABLE" denied
```

在这种情况下，为了在需要时能运行某些 DDL 命令，你必须删除该事件触发器 或者禁用它。只在一个事务期间禁用该触发器会比较方便：

```
BEGIN;
ALTER EVENT TRIGGER nodd1 DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER nodd1 ENABLE;
COMMIT;
```

（回忆一下，事件触发器本身上的 DDL 命令不受事件触发器影响）。

## 38.5. 一个表重写事件触发器例子

得益于table\_rewrite事件的存在，我们可以实现一种只允许在 维护窗口中重写的表重写策略。

这里是实现这种策略的例子。

```
CREATE OR REPLACE FUNCTION no_rewrite()
  RETURNS event_trigger
  LANGUAGE plpgsql AS
$$
---
--- 实现本地表重写策略：
---  public.foo 不允许重写，其他表只允许在 1am 和 6am 之间重写，
---  且前提是它们拥有不超过 100 块
---
DECLARE
  table_oid oid := pg_event_trigger_table_rewrite_oid();
  current_hour integer := extract('hour' from current_time);
  pages integer;
  max_pages integer := 100;
BEGIN
  IF pg_event_trigger_table_rewrite_oid() = 'public.foo'::regclass
  THEN
    RAISE EXCEPTION 'you''re not allowed to rewrite the table %',
      table_oid::regclass;
  END IF;

  SELECT INTO pages relpages FROM pg_class WHERE oid = table_oid;
```

```
IF pages > max_pages
THEN
    RAISE EXCEPTION 'rewrites only allowed for table with less than %
pages',
                    max_pages;
END IF;

IF current_hour NOT BETWEEN 1 AND 6
THEN
    RAISE EXCEPTION 'rewrites only allowed between 1am and 6am';
END IF;
END;
$$;

CREATE EVENT TRIGGER no_rewrite_allowed
    ON table_rewrite
    EXECUTE FUNCTION no_rewrite();
```

---

# Chapter 39. 规则系统

本章讨论LightDB中的规则系统。产生规则系统的概念很简单，但是在实际使用的时候会碰到很多细节问题。

某些其它数据库系统定义活动的数据库规则，通常是存储过程和触发器。在LightDB中，这些东西可以通过函数和触发器来实现。

规则系统（更准确地说是查询重写规则系统）与存储过程和触发器完全不同。它把查询修改为需要考虑规则，并且然后把修改过的查询传递给查询规划器进行规划和执行。它非常大，并且可以被用于许多东西如查询语言过程、视图和版本。这个规则系统的理论基础和能力也在[ston90b]和[ong90]中讨论。

## 39.1. 查询树

要了解规则系统是如何工作的，必须要知道它什么时候被调用以及它的输入和结果是什么。

规则系统位于解析器和规划器之间。它采用解析器的输出（即一个查询树）和用户定义的重写规则（也是查询树，不过带有一些额外信息），并且常见零个或者更多个查询树作为结果。因此它的输入和输出总是那些规划器自身就能产生的东西，并且因此它看到的任何东西都可以被表示成一个SQL语句。

那么什么是一个查询树？它是一个SQL语句的一种内部表示，其中用于创建它的每一个单独的部分都被独立存储。如果你设置了配置参数`debug_print_parse`、`debug_print_rewritten`或`debug_print_plan`，这些查询树可以被显示在服务器日志中。规则动作也被做为查询树存储在系统目录`pg_rewrite`中。它们没有被格式化为日志输出的形式，但是它们包含完全相同的信息。

阅读一棵未加工的查询树需要一些经验。但是由于查询树的SQL表示形式足以用来理解规则系统，本章将不会教授如何阅读查询树。

在阅读本章中查询树的SQL表现形式时，读者需要能够知道语句被分解成了哪些部分并且能在查询树结构中标识它们。一棵查询树的部分有：

### 命令类型

这是一个简单的值来说明是哪一种命令（SELECT、INSERT、UPDATE、DELETE）产生了该查询树。

### 范围表

范围表是被使用在该查询中的关系的列表。在一个SELECT语句中，范围表是在关键词FROM后面给出的关系。

每一个范围表项标识一个表或视图，并且说明在该查询的其他部分要以哪个名称调用它。在查询树中，范围表项被使用编号而不是名称来引用，因此在一个SQL语句中出现重复的名字也没有关系。在规则的范围表被合并以后可能会发生这种情况。本章中的例子将不会有这种情况。

### 结果关系

这是一个指向范围表的索引，它标识了该查询的结果应该去哪个关系。

SELECT查询没有结果关系（特殊情况SELECT INTO几乎等于CREATE TABLE后面跟上INSERT ... SELECT，并且不在这里单独讨论）。

对于INSERT、UPDATE和DELETE命令，结果关系是修改要进行的表（或视图！）。

## 目标列表

目标列表是一个表达式的列表，它定义了查询的结果。在一个SELECT的情况下，这些表达式会构建出该查询最终的输出。它们对应于关键字SELECT和FROM之间的表达式（\*是一个关系所有列名的缩写。解析器会把它扩展成独立的列，因此规则系统永远见不到它）。

DELETE命令不需要一个目标列表，因为它们不产生任何结果。相反，规划器会向空的目标列表中加入一个特殊的CTID项来允许执行器找到要被删除的行（当结果关系是一个普通表时才加入CTID。如果结果关系是一个视图，则会被规则系统加入一个整行变量，如Section 39.2. 所述）。

对于INSERT命令，目标列表描述了将要进入到结果关系中的新行。它由VALUES子句中的表达式或来自INSERT ... SELECT中SELECT子句的表达式构成。重写处理的第一步会为那些没有被原始命令赋值但有默认值的列增加目标列项。任何剩余的列（既没有给定值也没有默认值）将被规划器用一个常量空值表达式填充。

对于UPDATE命令，目标列表描述要替换旧行的新行。在规则系统中，它只包含来自命令的SET column = expression部分的表达式。规划器将处理缺失的列，做法是为它们插入表达式，这种表达式会把旧行的值复制到新行。正如DELETE一样，会增加一个CTID或整行变量，这样执行器能够标识要被更新的旧行。

目标列表中的每一个项所包含的表达式可以是一个常量值、一个指向范围表中关系的列的变量、一个参数或一个由函数调用、常量、变量、操作符等构成的表达式树。

## 条件

查询的条件是一个表达式，它很像包含在目标列表项中的表达式。这个表达式的结果值是一个布尔值，它说明对最终结果行的操作（INSERT、UPDATE、DELETE或SELECT）是否应该被执行。它对应于一个SQL语句的WHERE子句。

## 连接树

查询的连接树展示了FROM子句的结构。对于一个SELECT ... FROM a, b, c这样的简单查询，连接树就是FROM项的一个列表，因为我们被允许以任何顺序连接它们。但是当JOIN表达式（特别是外连接）被使用时，我们必须按照连接显示的顺序来连接。在这种情况下，连接树展示了JOIN表达式的结构。与特定JOIN子句（来自ON或USING）相关的限制被存储为附加到那些连接树节点的条件表达式。我们发现把顶层WHERE表达式存储为附加到顶层连接树项的一个条件也很方便。这样实际上连接树表达了一个SELECT的FROM和WHERE子句。

## 其他

查询树的其他部分（如ORDER BY子句）在这里并不受到关注。规则系统在应用规则时会替换这里的某些项，但是这些与规则系统的基础没有什么关系。

## 39.2. 视图和规则系统

LightDB中的视图是通过规则系统来实现的。事实上，下面的命令

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

与下面两个命令相比没有不同：

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

因为这就是CREATE VIEW命令在内部所作的。这样做有一些副作用。其中之一就是在LightDB系统目录中的视图信息与表的信息完全一样。所以对于解析器来说，表和视图之间完全没有区别。它们是同样的事物：关系。

## 39.2.1. SELECT规则如何工作

规则ON SELECT被应用于所有查询作为最后一步，即使给出的是一条INSERT、UPDATE或DELETE命令。而且它们与其他命令类型上的规则有着不同的语义，它们会就地修改查询树而不是创建一个新的查询树。因此我们首先描述SELECT规则。

目前，一个ON SELECT规则中只能有一个动作，而且它必须是一个无条件的INSTEAD的SELECT动作。这个限制是为了令规则足够安全，以便普通用户也可以打开它们，并且它限制ON SELECT规则使之行为类似视图。

本章的例子是两个连接视图，它们做一些运算并且某些更多视图会轮流使用它们。最前面的两个视图之一后面将利用对INSERT、UPDATE和DELETE操作增加规则的方法被自定义，这样最终结果将是一个视图，它表现得像一个具有魔力的真正的表。这个例子不适合于作为简单易懂的例子，它可能会让本章更难懂。但是用一个覆盖所有关键点的例子来一步一步讨论要比举很多例子搞乱思维好。

在前两个规则系统描述中我们需要真实表是：

```
CREATE TABLE shoe_data (
    shoename    text,           -- 主键
    sh_avail    integer,       -- 可用的双数
    slcolor     text,         -- 首选的鞋带颜色
    slminlen    real,         -- 最小鞋带长度
    slmaxlen    real,         -- 最大鞋带长度
    slunit      text         -- 长度单位
);

CREATE TABLE shoelace_data (
    sl_name     text,         -- 主键
    sl_avail    integer,     -- 可用的双数
    sl_color    text,       -- 鞋带颜色
    sl_len      real,       -- 鞋带长度
    sl_unit     text        -- 长度单位
);

CREATE TABLE unit (
    un_name     text,       -- 主键
    un_fact     real        -- 转换到厘米的参数
);
```

如你所见，它们表示鞋店的数据。

视图被创建为：

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
```

```

FROM shoe_data sh, unit un
WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
SELECT s.sl_name,
       s.sl_avail,
       s.sl_color,
       s.sl_len,
       s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
SELECT rsh.shoename,
       rsh.sh_avail,
       rsl.sl_name,
       rsl.sl_avail,
       least(rsh.sh_avail, rsl.sl_avail) AS total_avail
FROM shoe rsh, shoelace rsl
WHERE rsl.sl_color = rsh.slcolor
      AND rsl.sl_len_cm >= rsh.slminlen_cm
      AND rsl.sl_len_cm <= rsh.slmaxlen_cm;

```

创建shoelace视图的CREATE VIEW命令（也是最简单的一个）将创建一个shoelace关系和一个pg\_rewrite项，这个pg\_rewrite项说明有一个重写规则，只要一个查询的范围表中引用了关系shoelace，就必须应用它。该规则没有规则条件（稍后和非SELECT规则一起讨论，因为目前的SELECT规则不能有规则条件）并且它是INSTEAD规则。要注意规则条件与查询条件不一样。我们的规则的动作有一个查询条件。该规则的动作是一个查询树，这个查询是视图创建命令中的SELECT语句的一个拷贝。

### Note

你在pg\_rewrite项中看到的两个额外的用于NEW和OLD的范围表项不是SELECT规则感兴趣的东西。

现在我们填充unit、shoe\_data和shoelace\_data，并且在视图上运行一个简单的查询：

```

INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('s11', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('s12', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('s13', 0, 'black', 35.0, 'inch');
INSERT INTO shoelace_data VALUES ('s14', 8, 'black', 40.0, 'inch');
INSERT INTO shoelace_data VALUES ('s15', 4, 'brown', 1.0, 'm');
INSERT INTO shoelace_data VALUES ('s16', 0, 'brown', 0.9, 'm');
INSERT INTO shoelace_data VALUES ('s17', 7, 'brown', 60, 'cm');
INSERT INTO shoelace_data VALUES ('s18', 1, 'brown', 40, 'inch');

```

```
SELECT * FROM shoelace;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s11	5	black	80	cm	80
s12	6	black	100	cm	100
s17	7	brown	60	cm	60
s13	0	black	35	inch	88.9
s14	8	black	40	inch	101.6
s18	1	brown	40	inch	101.6
s15	4	brown	1	m	100
s16	0	brown	0.9	m	90

(8 rows)

这是你可以在我们的视图上做的最简单的SELECT，所以我们用这次机会来解释视图规则的基本要素。SELECT \* FROM shoelace会被解析器解释并生成下面的查询树：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace;
```

然后这将被交给规则系统。规则系统遍历范围表，检查有没有可用于任何关系的规则。在为shoelace（到目前为止的唯一一个）处理范围表时，它会发现查询树里有\_RETURN规则：

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace old, shoelace new,
     shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;
```

要扩展该视图，重写器简单地创建一个子查询范围表项，它包含规则的动作的查询树，然后用这个范围表记录取代原来引用视图的那个。作为结果的重写后的查询树几乎与你键入的那个一样：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM (SELECT s.sl_name,
            s.sl_avail,
            s.sl_color,
            s.sl_len,
            s.sl_unit,
            s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
      WHERE s.sl_unit = u.un_name) shoelace;
```

不过有一个区别：子查询的范围表有两个额外的项shoelace old和shoelace new。这些项并不直接参与到查询中，因为它们没有被子查询的连接树或者目标列表引用。重写器用它们存储最初出现在引用视图的范围表项中表达的访问权限检查信息。以这种方式，执行器仍然会检查该用户是否有访问视图的正确权限，尽管在重写后的查询中没有对视图的直接使用。

这是被应用的第一个规则。规则系统将检查顶层查询里剩下的范围表项（本例中没有了），并且它将递归的检查增加的子查询中的范围表项，看看其中有没有引用视图的（不过这样不会扩展old或new — 否则我们会得到无限递归！）。在这个例子中，没有用



于shoelace\_data或unit的重写规则，所以重写结束并且上面得到的就是给规划器的最终结果。

现在我们想写一个查询，它找出目前在店里哪些鞋子有匹配的（颜色和长度）鞋带并且完全匹配的鞋带双数大于等于二。

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;
```

shoename	sh_avail	sl_name	sl_avail	total_avail
sh1	2	sl1	5	2
sh3	4	sl7	7	4

(2 rows)

这个词解析器的输出是查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM shoe_ready shoe_ready
WHERE shoe_ready.total_avail >= 2;
```

第一个被应用的规则将是用于shoe\_ready的规则并且它会导致查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
            rsh.sh_avail,
            rsl.sl_name,
            rsl.sl_avail,
            least(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
      WHERE rsl.sl_color = rsh.slcolor
            AND rsl.sl_len_cm >= rsh.slminlen_cm
            AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail >= 2;
```

相似地，用于shoe和shoelace的规则被替换到子查询的范围表中，得到一个三层的最终查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
            rsh.sh_avail,
            rsl.sl_name,
            rsl.sl_avail,
            least(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM (SELECT sh.shoename,
                  sh.sh_avail,
                  sh.slcolor,
                  sh.slminlen,
                  sh.slminlen * un.un_fact AS slminlen_cm,
                  sh.slmaxlen,
```

```

        sh.slmaxlen * un.un_fact AS slmaxlen_cm,
        sh.slunit
    FROM shoe_data sh, unit un
    WHERE sh.slunit = un.un_name) rsh,
    (SELECT s.sl_name,
        s.sl_avail,
        s.sl_color,
        s.sl_len,
        s.sl_unit,
        s.sl_len * u.un_fact AS sl_len_cm
    FROM shoelace_data s, unit u
    WHERE s.sl_unit = u.un_name) rsl
    WHERE rsl.sl_color = rsh.slcolor
        AND rsl.sl_len_cm >= rsh.slminlen_cm
        AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
    WHERE shoe_ready.total_avail > 2;

```

这看起来是低效的，但是计划器会通过“pulling up”子查询将其折叠成一个单层查询树，然后它会计划联接，就像我们手动写出来一样。因此，折叠查询树是重写系统本身不必关心的一种优化。

### 39.2.2. $\neq$ SELECT语句中的视图规则

有两个查询树的细节在上面的视图规则的描述中没有涉及。它们是命令类型和结果关系。实际上，视图规则不需要命令类型，但是结果关系可能会影响查询重写器工作的方式，因为如果结果关系是一个视图，我们需要采取特殊的措施。

一个SELECT的查询树和其它命令的查询树之间很少的几处不同。显然，它们有不同的命令类型并且对于SELECT之外的命令，结果关系指向结果将进入的范围表项。其它所有东西都完全相同。所以如果有两个表t1和t2分别有列a和b，下面两个语句的查询树：

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

```
UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

几乎是一样的。特别是：

- 范围表包含表t1和t2的项。
- 目标列表包含一个变量，该变量指向表t2的范围表项的列b。
- 条件表达式比较两个范围表项的列a以寻找相等。
- 连接树展示了t1和t2之间的一次简单连接。

结果是，两个查询树生成相似的执行计划：它们都是两个表的连接。对于UPDATE语句，规划器把t1缺失的列加到目标列并且最终查询树读起来是：

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

因此在连接上运行的执行器将产生完全相同的结果集：

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

但是在UPDATE中有个小问题：执行器计划中执行连接的部分不关心连接的结果的含义。它只是产生一个行的结果集。一个是SELECT命令而另一个是由执行器中的更高层处理的UPDATE命

令，在那里执行器知道这是一个UPDATE，并且它知道这个结果应该进入表t1。但是这里的哪些行必须被新行替换呢？

要解决这个问题，在UPDATE和DELETE语句的目标列表里面增加了另外一个项：当前元组ID（CTID）。这是一个系统列，它包含行所在的文件块编号和在块中的位置。在已知表的情况下，CTID可以被用来检索要被更新的t1的原始行。在添加CTID到目标列之后，该查询实际看起来像：

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

现在，另一个LightDB的细节进入到这个阶段了。表中的旧行还没有被覆盖，这就是为什么ROLLBACK很快的原因。在一个UPDATE中，新的结果行被插入到表中（在剥除CTID之后），并且把CTID指向的旧行的行头部中的cmax和xmax项设置为当前命令计数器和当前事务ID。这样旧的行就被隐藏起来，并且在事务提交之后 vacuum 清理器就可以最终移除死亡的行。

知道了所有这些，我们就可以用完全相同的方式简单地把视图规则应用到任意命令中。没有任何区别。

### 39.2.3. LightDB中视图的能力

上文演示了规则系统如何把视图定义整合到原始的查询树中。在第二个例子中，一个来自于一个视图的简单SELECT创建了一个四表连接（unit以不同的名字被用了两次）的最终查询树。

用规则系统实现视图的好处是，规划器拥有关于哪些表必须被扫描、这些表之间的联系、来自于视图的限制性条件、一个单一查询树中原始查询的条件等所有信息。当原始查询已经是一个视图上的连接时仍然是这样。规划器必须决定执行查询的最优路径，而且规划器拥有越多信息，该决定就越好。并且LightDB中实现的规则系统保证这些信息是此时能获得的有关该查询的所有信息。

### 39.2.4. 更新一个视图

如果视图是INSERT、UPDATE或DELETE的目标关系会怎样？使用上文所述的替换将给出一个查询树，其中的结果关系指向一个子查询范围表项，这样无法工作。不过，LightDB中有几种方法来支持更新视图。根据用户体验复杂性的顺序，它们是：自动在底层表中替换视图、执行用户定义的触发器、或依据用户定义规则重写查询。这些选项将在下面讨论。

如果子查询从一个单一基本关系选择并且该关系足够简单，重写器会自动地把该子查询替换成底层的基本关系，这样INSERT、UPDATE或DELETE会被以适当的方式应用到该基本关系。其中“足够简单”的视图被称为自动可更新。有关这种可以被自动更新的视图类别的详细信息，请见CREATE VIEW。

或者，该操作可以被定义在视图上的一个用户提供的INSTEAD OF触发器处理。（参见CREATE TRIGGER）在这种情况下重写工作有一点点不同。对于INSERT，重写器对视图什么也不做，让它作为查询的结果关系。对于UPDATE和DELETE，仍有必要扩展该视图查询来产生命令将尝试更新或删除的“旧”行。因此该视图被按照通常的方式扩展，但是另一个未被扩展的范围表项会被增加到查询来表示该视图会尽其所能作为结果关系。

现在出现的问题是如何标识在视图中要被更新的行。回忆一下，当结果关系是一个表时，一个特殊的CTID项会被加入到目标列表来标识要被更新的行的物理位置。如果结果关系是一个视图这就行不通，因为一个视图根本就没有CTID，它的行没有实际的物理位置。对于一个UPDATE或DELETE操作，一个特殊的whoIerow项会被增加到目标列表中，它会扩展来包括来自该视图的所有列。执行器使用这个值来提供“旧”行给INSTEAD OF触发器。现在就轮到触发器来基于新旧行值来找出要更新什么了。

另外一种可能性是让用户定义INSTEAD规则，这种规则指定对视图上的INSERT\UPDATE和DELETE命令的替代动作。这些规则将重写该命令，通常是重写成一个更新一个或多个表（而不是视图）的命令。这是Section 39. 的主题。

注意规则会首先被计算，然后在原始查询被规划和执行之前重写它。因此，如果一个视图上同时有INSTEAD OF触发器和INSERT、UPDATE或DELETE规则，那么首先会计算规则，然后根据其结果决定是否执行触发器，触发器可能完全都不会被使用。

Automatic rewriting of an 在一个简单视图上的INSERT、UPDATE或DELETE查询的自动重写总是在最后尝试。因此，如果一个视图有规则或触发器，它们将重载自动可更新视图的默认行为。

如果对该视图没有INSTEAD规则或INSTEAD OF触发器，并且重写器不能自动地把该查询重写成一个底层基本关系上的更新，将会抛出一个错误，因为执行器不能更新一个这样的视图。

### 39.3. 物化视图

LightDB中的物化视图像视图一样使用了规则系统，但是以一种类表的形式保留了结果。在物化视图：

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

和视图：

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

之间的主要区别是物化视图不能直接被更新，并且用于创建物化视图的查询的存储方式和视图查询的存储方式完全相同，因此要为物化视图生成新鲜的数据：

```
REFRESH MATERIALIZED VIEW mymatview;
```

The information about a materialized view in the 有关一个LightDB系统目录中的物化视图的信息和一个表或视图的信息完全相同。因此对于解析器，一个物化视图就是一个关系，就像一个表或一个视图。当一个物化视图被一个查询引用时，数据直接从物化视图中返回，如同表一样；规则只被用来填充物化视图。

虽然对物化视图中存储的数据的访问常常要快于直接访问底层表或通过一个视图访问，但是数据并不总是最新的；但是某些时候并不需要当前数据。考虑一个记录销售情况的表：

```
CREATE TABLE invoice (
    invoice_no integer PRIMARY KEY,
    seller_no integer, -- 销售员的 ID
    invoice_date date, -- 销售日期
    invoice_amt numeric(13,2) -- 销售量
);
```

如果人们想快速绘制历史销售数据，他们可能希望汇总，并且他们可能并不关心当前日期的不完整数据：

```
CREATE MATERIALIZED VIEW sales_summary AS
SELECT
    seller_no,
    invoice_date,
    sum(invoice_amt)::numeric(13,2) as sales_amt
FROM invoice
WHERE invoice_date < CURRENT_DATE
GROUP BY
    seller_no,
```

```

    invoice_date
ORDER BY
    seller_no,
    invoice_date;

```

```

CREATE UNIQUE INDEX sales_summary_seller
ON sales_summary (seller_no, invoice_date);

```

这个物化视图可能对在为销售员创建的控制面板上显示一个图表非常有用。可以用一个计划任务在每晚使用这个 SQL 语句更新该统计信息：

```
REFRESH MATERIALIZED VIEW sales_summary;
```

物化视图的另一种使用是允许通过一个外部数据包装器对来自一个远程系统的数据进行更快的访问。下面有一个使用 `file_fdw` 的简单例子，但是由于本地系统上可以使用高速缓存，因此比起访问一个远程系统的性能差异可能会比这里所展示的更大。注意鉴于 `file_fdw` 不支持索引，我们也使用这种能力来在物化视图上放置索引。这种优势可能不适用于其他种类的外部数据访问。

建立：

```

CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE words (word text NOT NULL)
    SERVER local_file
    OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION lt_trgm;
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;

```

现在让我们对一个词进行拼写检查。直接使用 `file_fdw`：

```
SELECT count(*) FROM words WHERE word = 'caterpiler';
```

```

count
-----
      0
(1 row)

```

通过 `EXPLAIN ANALYZE`，我们可以看到：

```

Aggregate (cost=21763.99..21764.00 rows=1 width=0) (actual
time=188.180..188.181 rows=1 loops=1)
  -> Foreign Scan on words (cost=0.00..21761.41 rows=1032 width=0) (actual
time=188.177..188.177 rows=0 loops=1)
    Filter: (word = 'caterpiler'::text)
    Rows Removed by Filter: 479829
    Foreign File: /usr/share/dict/words
    Foreign File Size: 4953699
Planning time: 0.118 ms
Execution time: 188.273 ms

```

如果使用物化视图，该查询会快很多：

```

Aggregate (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1
loops=1)
  -> Index Only Scan using wrd_word on wrd (cost=0.42..4.44 rows=1 width=0)
(actual time=0.039..0.039 rows=0 loops=1)
      Index Cond: (word = 'caterpiler'::text)
      Heap Fetches: 0
Planning time: 0.164 ms
Execution time: 0.117 ms

```

不管哪种方式，单词都是被拼错的，因此让我们看看什么是我们可能想要的。再次使用file\_fdw 和 lt\_trgm:

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;
```

```

      word
-----
cater
caterpillar
Caterpillar
caterpillars
caterpillar's
Caterpillar's
caterer
caterer's
caters
catered
(10 rows)

```

```

Limit (cost=11583.61..11583.64 rows=10 width=32) (actual
time=1431.591..1431.594 rows=10 loops=1)
  -> Sort (cost=11583.61..11804.76 rows=88459 width=32) (actual
time=1431.589..1431.591 rows=10 loops=1)
      Sort Key: ((word <-> 'caterpiler'::text))
      Sort Method: top-N heapsort  Memory: 25kB
      -> Foreign Scan on words (cost=0.00..9672.05 rows=88459 width=32)
(actual time=0.057..1286.455 rows=479829 loops=1)
          Foreign File: /usr/share/dict/words
          Foreign File Size: 4953699
Planning time: 0.128 ms
Execution time: 1431.679 ms

```

使用物化视图:

```

Limit (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257 rows=10
loops=1)
  -> Index Scan using wrd_trgm on wrd (cost=0.29..37020.87 rows=479829
width=10) (actual time=187.219..188.252 rows=10 loops=1)
      Order By: (word <-> 'caterpiler'::text)
Planning time: 0.196 ms
Execution time: 198.640 ms

```

如果你能够忍受定期把远程数据更新到本地数据库，其性能收益可能是巨大的。

## 39.4. INSERT、UPDATE和DELETE上的规则

定义在INSERT、UPDATE和DELETE上的规则与前一节描述的视图规则有明显的不同。首先，它们的CREATE RULE命令允许更多：

- 它们可以没有动作。
- 它们可以有多个动作。
- 它们可以是INSTEAD或ALSO（缺省）。
- 伪关系NEW和OLD变得有用了。
- 它们可以有规则条件。

第二，它们不是就地修改查询树，而是创建零个或多个新查询树并且可能把原始的那个查询树扔掉。

### Caution

在很多情况下，由INSERT/UPDATE/DELETE上的规则执行的任务用触发器能做得更好。触发器在记法上要更复杂些，但是它们的语义理解起来更简单些。当原始查询包含不稳定函数时，规则容易产生令人惊讶的结果：在执行规则的过程中不稳定函数的执行次数可能比预期中的更多。

还有，有些情况根本无法用这些类型的规则支持，典型的是在原始查询中包括WITH子句以及在UPDATE查询的SET列表中包括多个赋值的子SELECT。这是因为把这些结构复制到一个规则查询中可能导致子查询的多次计算，这与查询作者表达的意图相悖。

## 39.4.1. 更新规则如何工作

记住以下语法：

```
CREATE [ OR REPLACE ] RULE name AS ON event
  TO table [ WHERE condition ]
  DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

在随后的内容中，更新规则表示定义在INSERT、UPDATE或DELETE上的规则。

如果查询树的结果关系和命令类型等于CREATE RULE命令中给出的对象和事件，规则系统就会应用更新规则。对于更新规则，规则系统会创建一个查询树列表。一开始该查询树列表是空的。更新规则中可以有零个（NOTHING关键字）、一个或多个动作。为简单起见，我们先看一个只有一个动作的规则。这个规则可以有条件或者没有条件，并且它可以是INSTEAD或ALSO（缺省）。

什么是规则条件？它是一个限制，告诉规则动作什么时候做、什么时候不做。这个条件只能引用NEW和/或OLD伪关系，它们基本上代表作为对象给定的关系（但是有着特殊含义）。

所以，对这个单动作的规则生成下面的查询树，我们有三种情况。

没有条件，有ALSO或INSTEAD

来自规则动作的查询树，在其上增加原始查询树的条件

给出了条件，有ALSO

来自规则动作的查询树，在其上加入规则条件和原始查询树的条件

给出了条件，有INSTEAD

来自规则动作的查询树，在其上加入规则条件和原始查询树的条件；以及带有反规则条件的原始查询树

最后，如果规则是ALSO，那么未修改的原始查询树也被加入到列表。因为只有合格的INSTEAD规则已经被加入到原始查询树中，对于单动作的规则，我们将结束于一个或两个输出查询树。

对于ON INSERT规则，原始查询（如果没有被INSTEAD取代）是在任何规则增加的动作之前完成的。这样就允许动作看到被插入的行。但是对ON UPDATE 和ON DELETE规则，原始查询是在规则增加的动作之后完成的。这样就确保动作可以看到将要更新或者将要删除的行；否则，动作可能什么也不做，因为它们无法发现符合它们要求的行。

从规则动作生成的查询树会被再次丢给重写系统，并且可能有更多规则被应用而得到更多或更少的查询树。所以一个规则的动作必须有一种不同的命令类型或者和规则所在的关系不同的另一个结果关系。否则这样的递归处理就会没完没了（规则的递归展开会被检测到，并当作一个错误报告）。

在pg\_rewrite系统目录中的动作中的查询树只是模板。因为它们可以引用NEW和OLD的范围表项，在使用它们之前必须做一些替换。对于任何NEW的引用，都要先在原始查询的目标列表中搜索对应的项。如果找到，该项的表达式将会替换该引用。否则NEW和OLD的含义一样（对于UPDATE）或者被替换成一个空值（对于INSERT）。任何对OLD的引用都用结果关系的范围表项的引用替换。

在系统完成应用更新规则后，它再应用视图规则到生成的查询树上。视图无法插入新的更新动作，所以没有必要向视图重写的输出应用更新规则。

### 39.4.1.1. 第一个规则循序渐进

假设我们想要跟踪shoelace\_data关系中的sl\_avail列。所以我们建立一个日志表和一条规则，这条规则每次在shoelace\_data上执行UPDATE时有条件地写入一个日志项。

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- 改变的鞋带
    sl_avail   integer,      -- 新的可用值
    log_who    text,          -- 谁做的
    log_when   timestamp     -- 何时做的
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
WHERE NEW.sl_avail <> OLD.sl_avail
DO INSERT INTO shoelace_log VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    current_user,
    current_timestamp
);
```

现在有人做：

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl1';
```

然后看看日志表：

```
SELECT * FROM shoelace_log;
```



```

sl_name | sl_avail | log_who | log_when
-----+-----+-----+-----
sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)

```

这就是我们所期望的。在后台发生的的事情如下。解析器创建查询树：

```

UPDATE shoelace_data SET sl_avail = 6
FROM shoelace_data shoelace_data
WHERE shoelace_data.sl_name = 'sl7';

```

这是一个带有规则条件表达式的ON UPDATE规则log\_shoelace，条件是：

```
NEW.sl_avail <> OLD.sl_avail
```

它的动作是：

```

INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old;

```

（这看起来有点奇怪，因为你通常不能写INSERT ... VALUES ... FROM。这里的FROM子句只是表示查询树里有用于new和old的范围表项。这些东西是必需的，这样它们就可以被INSERT命令的查询树中的变量引用）。

该规则是一个有条件的ALSO规则，所以规则系统必须返回两个查询树：更改过的规则动作和原始查询树。在第 1 步里，原始查询的范围表被集成到规则动作的查询树中。得到：

```

INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
shoelace_data shoelace_data;

```

第 2 步把规则条件增加进去，所以结果集被限制为sl\_avail改变了的行：

```

INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
shoelace_data shoelace_data
WHERE new.sl_avail <> old.sl_avail;

```

（这看起来更奇怪，因为INSERT ... VALUES也没有WHERE子句，但是规划器和执行器处理它没有任何难度。不管怎样，它们需要为INSERT ... SELECT支持这种相同功能）。

第 3 步把原始查询树的条件加进去，把结果集进一步限制成只有被初始查询树改变的行：

```

INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
shoelace_data shoelace_data

```

```
WHERE new.sl_avail <> old.sl_avail
AND shoelace_data.sl_name = 's17';
```

第 4 步把NEW引用替换为来自原始查询树的目标列表项或来自结果关系的相匹配的变量引用:

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE 6 <> old.sl_avail
AND shoelace_data.sl_name = 's17';
```

第 5 步, 用结果关系引用把OLD引用替换掉:

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE 6 <> shoelace_data.sl_avail
AND shoelace_data.sl_name = 's17';
```

这就完成了。因为规则是ALSO, 我们还要输出原始查询树。简而言之, 从规则系统输出的是一个包含两个查询树的列表, 它们与下面语句相对应:

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data
WHERE 6 <> shoelace_data.sl_avail
AND shoelace_data.sl_name = 's17';
```

```
UPDATE shoelace_data SET sl_avail = 6
WHERE sl_name = 's17';
```

这些会按照这个顺序被执行, 并且这也正是规则要做的事情。

做的替换和追加的条件用于确保对于下面这样的原始查询不会有日志记录被写入:

```
UPDATE shoelace_data SET sl_color = 'green'
WHERE sl_name = 's17';
```

在这种情况下, 原始查询树不包含sl\_avail的目标列表项, 因此NEW.sl\_avail将被shoelace\_data.sl\_avail代替。所以, 规则生成的额外命令是:

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, shoelace_data.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data
WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
AND shoelace_data.sl_name = 's17';
```

并且条件将永远不可能为真。

如果原始查询修改多个行，这也能争产工作。所以如果某人发出命令：

```
UPDATE shoelace_data SET sl_avail = 0
WHERE sl_color = 'black';
```

实际上有四行（s11、s12、s13和s14）被更新。但s13已经是sl\_avail = 0。在这种情况下，原始查询树的条件不同并且导致规则产生额外的查询树：

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
FROM shoelace_data
WHERE 0 <> shoelace_data.sl_avail
AND shoelace_data.sl_color = 'black';
```

这个查询树将肯定插入三个新的日志项。这也是完全正确的。

到这里我们就能明白为什么原始查询树最后执行非常重要。如果UPDATE先被执行，则所有的行都已经被设为零，所以记日志的INSERT将无法找到任何符合0 shoelace\_data.sl\_avail的行。 <>

## 39.4.2. 与视图合作

要保护一个视图关系不被INSERT、UPDATE或DELETE，一种简单的方法是让那些查询树被丢掉。因此我们可以创建规则：

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
DO INSTEAD NOTHING;
```

如果现在某人尝试对视图关系shoe做任何这些操作，规则系统将应用这些规则。因为这些规则没有动作而且是INSTEAD，作为的查询树列表将是空的并且整个查询将变得什么也不做，因为经过规则系统处理后没有什么东西剩下来被优化或执行了。

一个更好的使用规则系统的方法是创建一些规则，这些规则把查询树重写成在一个在真实表上进行正确的操作的查询树。要在视图shoelace上做这件事，我们创建下列规则：

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
INSERT INTO shoelace_data VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    NEW.sl_color,
    NEW.sl_len,
    NEW.sl_unit
);

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
DO INSTEAD
UPDATE shoelace_data
SET sl_name = NEW.sl_name,
    sl_avail = NEW.sl_avail,
```

```

        sl_color = NEW.sl_color,
        sl_len = NEW.sl_len,
        sl_unit = NEW.sl_unit
WHERE sl_name = OLD.sl_name;

```

```

CREATE RULE shoelace_del AS ON DELETE TO shoelace
DO INSTEAD
DELETE FROM shoelace_data
WHERE sl_name = OLD.sl_name;

```

如果你要在视图上支持RETURNING查询，你需要让规则包含RETURNING子句来计算视图行。这对于基于单个表的视图来说通常非常简单，但是对于连接视图（如shoelace）就有点冗长了。对于插入的一个例子：

```

CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
INSERT INTO shoelace_data VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    NEW.sl_color,
    NEW.sl_len,
    NEW.sl_unit
)
RETURNING
shoelace_data.*,
(SELECT shoelace_data.sl_len * u.un_fact
FROM unit u WHERE shoelace_data.sl_unit = u.un_name);

```

注意，这个规则同时支持该视图上的INSERT和INSERT RETURNING查询 — 对于INSERT会简单地忽略RETURNING子句。

现在假设有时一包鞋带抵达了商店，并且随着它有一个大的清单。但是你不希望每次都手工更新shoelace视图。取而代之的是我们建立两个小表：一个用来从清单向其中插入东西，另一个则用了一个特殊的技巧。这些东西的创建命令如下：

```

CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

```

```

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

```

```

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
DO INSTEAD
UPDATE shoelace
SET sl_avail = sl_avail + NEW.ok_quant
WHERE sl_name = NEW.ok_name;

```

现在你可以用来自清单的数据填充表shoelace\_arrive：

```

SELECT * FROM shoelace_arrive;

arr_name | arr_quant

```

```

-----+-----
s13      |          10
s16      |          20
s18      |          20
(3 rows)

```

快速地看一看当前的数据:

```
SELECT * FROM shoelace;
```

```

sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----+-----+-----+-----+-----+-----
s11     |         5 | black    |      80 | cm      |         80
s12     |         6 | black    |     100 | cm      |        100
s17     |         6 | brown    |      60 | cm      |         60
s13     |         0 | black    |      35 | inch    |        88.9
s14     |         8 | black    |      40 | inch    |       101.6
s18     |         1 | brown    |      40 | inch    |       101.6
s15     |         4 | brown    |       1 | m       |        100
s16     |         0 | brown    |     0.9 | m       |         90
(8 rows)

```

现在把到的货鞋带移到:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

并检查结果:

```
SELECT * FROM shoelace ORDER BY sl_name;
```

```

sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----+-----+-----+-----+-----+-----
s11     |         5 | black    |      80 | cm      |         80
s12     |         6 | black    |     100 | cm      |        100
s17     |         6 | brown    |      60 | cm      |         60
s14     |         8 | black    |      40 | inch    |       101.6
s13     |        10 | black    |      35 | inch    |        88.9
s18     |        21 | brown    |      40 | inch    |       101.6
s15     |         4 | brown    |       1 | m       |        100
s16     |        20 | brown    |     0.9 | m       |         90
(8 rows)

```

```
SELECT * FROM shoelace_log;
```

```

sl_name | sl_avail | log_who | log_when
-----+-----+-----+-----
s17     |         6 | A1      | Tue Oct 20 19:14:45 1998 MET DST
s13     |        10 | A1      | Tue Oct 20 19:25:16 1998 MET DST
s16     |        20 | A1      | Tue Oct 20 19:25:16 1998 MET DST
s18     |        21 | A1      | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)

```

从一个INSERT ... SELECT到这些结果经过了很长的过程。并且该查询树转换的描述将出现在本章的最后。首先，这里是解析器的输出:

```
INSERT INTO shoelace_ok
```

```
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

现在应用第一条规则shoelace\_ok\_ins被应用并且把这个输出转换成:

```
UPDATE shoelace
SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace
WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

并且丢掉shoelace\_ok上的INSERT。这个被重写后的查询被再次传递给规则系统，并且第二个被应用的规则shoelace\_upd会产生:

```
UPDATE shoelace_data
SET sl_name = shoelace.sl_name,
sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
sl_color = shoelace.sl_color,
sl_len = shoelace.sl_len,
sl_unit = shoelace.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace, shoelace old,
shoelace new, shoelace_data shoelace_data
WHERE shoelace.sl_name = shoelace_arrive.arr_name
AND shoelace_data.sl_name = shoelace.sl_name;
```

同样这是一个INSTEAD规则并且前一个查询树会被丢弃掉。注意这个查询仍然使用视图shoelace。但是规则系统还没有完成这一步，所以它会继续并在其上应用\_RETURN规则，并且我们得到:

```
UPDATE shoelace_data
SET sl_name = s.sl_name,
sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
sl_color = s.sl_color,
sl_len = s.sl_len,
sl_unit = s.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace, shoelace old,
shoelace new, shoelace_data shoelace_data,
shoelace old, shoelace new,
shoelace_data s, unit u
WHERE s.sl_name = shoelace_arrive.arr_name
AND shoelace_data.sl_name = s.sl_name;
```

最后，规则log\_shoelace被应用，生成额外的查询树:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
s.sl_avail + shoelace_arrive.arr_quant,
current_user,
current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
```

```

shoelace shoelace, shoelace old,
shoelace new, shoelace_data shoelace_data,
shoelace old, shoelace new,
shoelace_data s, unit u,
shoelace_data old, shoelace_data new
shoelace_log shoelace_log
WHERE s.sl_name = shoelace_arrive.arr_name
      AND shoelace_data.sl_name = s.sl_name
      AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;

```

完成这些之后，规则系统用完了所有的规则并且返回生成的查询树。

所以我们结束于两个最终查询树，它们等效于SQL语句：

```

INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
     shoelace_data s
WHERE s.sl_name = shoelace_arrive.arr_name
      AND shoelace_data.sl_name = s.sl_name
      AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive,
     shoelace_data shoelace_data,
     shoelace_data s
WHERE s.sl_name = shoelace_arrive.sl_name
      AND shoelace_data.sl_name = s.sl_name;

```

结果是从一个关系来的数据插入了到另一个中，改变成第三个上的更新，改变成更新第四个外加做日志，在第五个中的最后更新缩减为两个查询。

有一个小细节有点丑陋。看看那两个查询，我们会发现shoelace\_data关系在范围表中出现了两次而实际上绝对可以缩为出现一次。规划器不会处理它，因此INSERT的规则系统输出的执行规划会是

```

Nested Loop
-> Merge Join
    -> Seq Scan
        -> Sort
            -> Seq Scan on s
    -> Seq Scan
        -> Sort
            -> Seq Scan on shoelace_arrive
-> Seq Scan on shoelace_data

```

在省略额外的范围表项后会得到

```

Merge Join
-> Seq Scan
    -> Sort
        -> Seq Scan on s
-> Seq Scan

```

```
-> Sort
    -> Seq Scan on shoelace_arrive
```

这在日志表中生成完全一样的项。因此，规则系统导致了shoelace\_data表上的一次绝对不必要的扫描。并且同样的冗余扫描会在UPDATE中进行。但是要把这些全部实现实在是一项很困难的工作。

现在我们对LightDB规则系统及其能力做最后一个演示。假设你向你的数据库中添加一些有特别颜色的鞋带：

```
INSERT INTO shoelace VALUES ('s19', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('s110', 1000, 'magenta', 40.0, 'inch', 0.0);
```

我们想要建立一个视图来检查哪些shoelace项在颜色上不配任何鞋子。适用的视图是：

```
CREATE VIEW shoelace_mismatch AS
  SELECT * FROM shoelace WHERE NOT EXISTS
    (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

它的输出是：

```
SELECT * FROM shoelace_mismatch;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s19	0	pink	35	inch	88.9
s110	1000	magenta	40	inch	101.6

现在我们想建立它，这样没有库存的不匹配的鞋带都会被从数据库中删除。为了对LightDB有点难度，我们不直接删除它们。而是我们再创建一个视图：

```
CREATE VIEW shoelace_can_delete AS
  SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

然后用下面方法：

```
DELETE FROM shoelace WHERE EXISTS
  (SELECT * FROM shoelace_can_delete
   WHERE sl_name = shoelace.sl_name);
```

The results are:

```
SELECT * FROM shoelace;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s11	5	black	80	cm	80
s12	6	black	100	cm	100
s17	6	brown	60	cm	60
s14	8	black	40	inch	101.6
s13	10	black	35	inch	88.9
s18	21	brown	40	inch	101.6
s110	1000	magenta	40	inch	101.6
s15	4	brown	1	m	100



```
s16      |      20 | brown  |      0.9 | m      |      90
(9 rows)
```

对一个视图上的DELETE，这个命令带有一个总共使用了四个嵌套/连接视图的子查询条件，这四个视图之一本身有一个包含一个视图的子查询条件，该条件计算使用的视图列；这个命令被重写成了一个查询树，该查询树从一个真正的表里面把需要删除的数据删除。

在现实世界里只有很少的情况需要上面的这样的构造。但这些东西能运转肯定让你感觉不错。

## 39.5. 规则和权限

由于LightDB规则系统对查询的重写，会访问没有在原始查询中指定的表/视图。使用更新规则时，这可能包括对表的写权限。

重写规则并不拥有一个独立的所有者。关系（表或视图）的所有者自动成为为其所定义的重写规则的所有者。LightDB规则系统改变了默认的访问控制系统的行为。由于规则被使用的关系会按照规则所有者的权限来检查，而不是调用规则的用户。这表示用户只需要在其查询中显式指定的表/视图上的所需权限。

例如：某用户有一个电话号码列表，其中一些是私人的，另外的一些是办公室助理需要的。该用户可以构建下面的东西：

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO assistant;
```

除了该用户以外（还有数据库超级用户）没有人可以访问phone\_data表。但因为GRANT的原因，助理可以在phone\_number视图上运行SELECT。规则系统将把phone\_number上的SELECT重写为phone\_data上的SELECT。因为该用户是phone\_number的所有者，因此也是规则的所有者，对phone\_data的读访问现在被根据该用户的权限检查，并且该查询被允许。同时也要检查访问phone\_number的权限，但这是针对调用用户进行的，所以除了用户自己和助理外没有人可以使用它。

权限检查是按规则逐条进行的。所以此时助理是唯一的一个可以看到公共电话号码的人。但助理可以建立另一个视图并且赋予该视图公共权限。这样，任何人都可以通过助理的视图看到phone\_number数据。助理不能做的事情是创建一个直接访问phone\_data的视图（实际上助理是可以的，但没有任何作用，因为每次访问都会因通不过权限检查而被否定）。而且该用户一旦注意到助理开放了他的phone\_number视图，该用户还可以收回助理的访问权限。立刻，所有对助理视图的访问将会失败。

有人可能会认为这种逐条规则的检查是一个安全漏洞，但事实上不是。如果这样做不能奏效，助理将必须建立一个与phone\_number有相同列的表并且每天拷贝一次数据进去。那么这是助理自己的数据因而助理可以为每一个想要访问的人授权。一个GRANT意味着“我信任你”。如果某个你信任的人做了上面的事情，那么是时候认为信任已经结束并且要使用REVOKE。

需要注意的是，虽然视图可以用前文展示的技术来隐藏特定列的内容，它们不能可靠地在不可见行上隐藏数据，除非标志被设置。例如，下面的视图是不安全的：

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

这个视图看起来是安全的，因为规则系统会把任何phone\_number上的SELECT重写为phone\_data上的SELECT，并且增加限制使得只有phone 不以 412 开头的项才被处理。但是如

果用户可以创建自己的函数，那就不难让规划器在NOT LIKE表达式之前先执行用户自定义函数。例如：

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END;
$$ LANGUAGE plpgsql COST 0.000000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

phone\_data表中的每一个人和电话号码会被打印成一个NOTICE，因为规划器会选择在执行NOT LIKE之前先执行tricky，因为前者的开销大。即使禁止用户自定义一个新函数，内置函数也可以用在类似的攻击中（例如，大部分造型函数会在它们产生的错误信息中包含它们的输入值）。

类似的考虑应用于更新规则。在前一节例子中，例子数据库中表的所有者可以把shoelace视图上的SELECT、INSERT、UPDATE和DELETE权限授予其他人，但对shoelace\_log只有SELECT权限。写日志项的规则动作将仍然可以被成功地执行，并且其它用户可以看到日志项。但他们不能创建伪造的项，并且他们也不能操纵或移除现有的项。在这种情况下，不可能通过让规划器改变操作的顺序来推翻规则，因为引用shoelace\_log的唯一规则是无限制的INSERT。在更复杂的情景中，这可能不正确。

当需要对一个视图提供行级安全时，security\_barrier属性应该被应用到该视图。这会阻止恶意选择的函数和操作符通过行被传递，直到视图完成其工作。例如，如果前文所示的视图被创建成这样，它就是安全的：

```
CREATE VIEW phone_number WITH (security_barrier) AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

Views created with the 使用security\_barrier创建的视图的性能会远差于没有使用该选项的视图。通常，没有办法来避免这种现状：如果最快的候选计划可能在安全性上折衷，它就必须被拒绝。出于该原因，这个选在在默认情况下是没有启用的。

当处理没有副作用的函数时，查询规划器有更多的灵活性。这类函数被称为LEAKPROOF，并且包括很多简单常用的操作符，例如很多等于操作符。查询规划器可以安全地允许这类函数在查询执行过程中的任何点被计算，因为在用户不可见的行上调用它们将不会泄露关于不可见行的任何信息。更进一步，不接收参数或者不从安全屏障视图得到任何参数的函数不必被标记为LEAKPROOF以便被下推，因为它们从来不会从该视图接收数据。相反，一个可能会基于接收到的参数值抛出错误的函数（例如在溢出或被零除事件中抛出错误的函数）不是防泄漏的，并且如果它被应用在安全性视图的行过滤器之前，它可能会提供有关不可见行的有效信息。

有一点很重要的是理解：即使一个视图使用security\_barrier选项创建，它也只在不可见元组不会被传递给可能不安全的函数的前提下才是安全的。用户可能也有其他方式来推断不可见数据：例如，他们可以使用EXPLAIN看到查询计划，或者针对视图来测量查询的运行时间。一个恶意攻击者可能有能力推断有关不可见数据的总量，或者甚至得到有关数据分布的某些信息或最常用值（因为这些东西可以影响计划的运行时间；或者甚至计划的选择，因为它们也被反映在优化器的统计数据中）。如果这类“隐通道”攻击很重要，那么授予任何到该数据的访问都可能是不明智的。

## 39.6. 规则和命令状态

LightDB服务器为它收到的每个命令返回一个命令状态字符串，例如INSERT 149592 1。没有涉及规则时这很简单，但是查询被规则重写时会发生什么呢？

规则对命令状态的影响如下：

- 如果没有查询的无条件INSTEAD规则，那么原始给出的查询将会被执行，并且它的命令状态将像平常一样被返回（但是请注意如果存在任何有条件INSTEAD规则，那么它们的反条件将被加到原始查询中。这样可能会减少它处理的行数，并且报告的状态将受影响）。
- 如果有查询的任何无条件INSTEAD规则，那么原始查询将完全不被执行。在这种情况下，服务器将返回由服务器将返回由INSTEAD规则（有条件的或无条件的）插入的最后一条和原始查询命令类型（INSERT、UPDATE或DELETE）相同的查询的命令状态。如果任何规则添加的查询都不符合这些要求，那么返回的命令状态显示原始查询类型并且行计数和 OID 域为零。

通过为任何想要的INSTEAD规则指定在活动规则中排名最后的规则名，程序员可以确保该规则都是在第二种情况里设置命令状态的规则，因为它会被最后一个应用。

## 39.7. 规则 vs 触发器

许多触发器可以干的事情同样也可以用LightDB规则系统来实现。目前不能用规则来实现的东西之一是某些约束，特别是外键。可以放置一个合格的规则在一列上，这个规则在列的值没有出现在另一个表中时把命令重写成NOTHING。但是这样做数据就会被不声不响地丢弃，因此也不是一个好主意。如果要求检查值的有效性，并且在出现无效值的情况下应该生成一个错误消息，这种需求就必须要用触发器来完成。

在本章中，我们关注于使用规则来更新视图。本章中所有的更新规则的例子都可以使用视图上的INSTEAD OF触发器来实现。编写这类触发器通常比编写规则要容易，特别是在要求使用复杂逻辑来执行更新的情况下。

对于两者都可实现的情况，哪个更好取决于对数据库的使用。触发器为每一个受影响的行都执行一次。规则修改查询树或生成一个额外的查询。所以如果在一个语句中影响到很多行，一个发出额外查询的规则通常可能会比一个触发器快，因为触发器对每一个行都要被调用，并且每次被调用时都需要重新判断要做什么样的操作。不过，触发器方法从概念上要远比规则方法简单，并且很容易让新人上手。

下面我们展示一个例子，该例子说明了在同种情况下两种选择的比较。这里有两个表：

```
CREATE TABLE computer (
    hostname      text,    -- 被索引
    manufacturer  text    -- 被索引
);
```

```
CREATE TABLE software (
    software      text,    -- 被索引
    hostname      text    -- 被索引
);
```

两个表都有数千行，并且在hostname上的索引是唯一的。规则或触发器应该实现一个约束，该约束从software中删除引用已删除计算机的行。触发器可以用下面这条命令：

```
DELETE FROM software WHERE hostname = $1;
```

因为触发器会为每一个从computer中删除的独立行调用一次，那么它可以准备并且保存这个命令的规划，把hostname作为参数传入。规则应该被写为：

```
CREATE RULE computer_del AS ON DELETE TO computer
DO DELETE FROM software WHERE hostname = OLD.hostname;
```

现在看看不同类型的删除。在这种情况下：

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

表computer被使用索引（快速）扫描，并且由触发器发出的命令也将使用一个索引扫描（同样快速）。来自规则的额外查询应该是：

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
AND software.hostname = computer.hostname;
```

由于已经建立了合适的索引，规划器将创建一个规划

Nestloop

```
-> Index Scan using comp_hostidx on computer
-> Index Scan using soft_hostidx on software
```

所以在触发器和规则的实现之间没有太多的速度差别。

在接下来的删除中，我们想要去掉所有 2000 个hostname以old开头的计算机。有两个命令可以来做这件事。一个是：

```
DELETE FROM computer WHERE hostname >= 'old'
AND hostname < 'ole'
```

被规则增加的命令将是：

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname <
'ole'
AND software.hostname = computer.hostname;
```

计划是：

Hash Join

```
-> Seq Scan on software
-> Hash
-> Index Scan using comp_hostidx on computer
```

另一个可能的命令是：

```
DELETE FROM computer WHERE hostname ~ '^old';
```

它会为规划增加的命令产生下面的执行计划：

Nestloop

```
-> Index Scan using comp_hostidx on computer
-> Index Scan using soft_hostidx on software
```

这表明，当有多个条件表达式被使用AND组合在一起时，规划器不能认识到表computer中hostname上的条件也可以被用于一个software上的索引扫描，而在该命令的正则表达式版本中正是这样做的。触发器将为要被删除的 2000 个旧计算机中的每一个调用，并且会导致在computer上的一次索引扫描和software上的 2000 次索引扫描。采用规则的实现将会使用两个使用索引的命令来完成。并且在顺序扫描情况下规则是否仍将更快是取决于software表的总体大小的。即使所有的索引块都将很快地进入高速缓存，通过 SPI 管理器执行来自触发器的 2000 个命令也要花不少时间。

我们要看的最后一个命令是：

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

同样，这也会导致很多行被从computer中删除。所以触发器同样会通过执行器运行很多命令。规则生成的命令将会是：

```
DELETE FROM software WHERE computer.manufacturer = 'bim'  
AND software.hostname = computer.hostname;
```

这个命令的计划又将是在两个索引扫描上的嵌套循环，只不过使用了computer上的另一个索引：

Nestloop

```
-> Index Scan using comp_manufidx on computer  
-> Index Scan using soft_hostidx on software
```

在任何这些情况之一，来自规则系统的额外命令都或多或少与命令中影响的行数无关。

概括来说，规则只有在其动作导致了大而且糟糕的条件连接时才会明显地慢于触发器，这种情况下规划器将没有什么办法。

---

## Chapter 40. 过程语言

LightDB允许使用除了 SQL 和 C 之外的其他语言编写用户定义的函数。这些其他的语言通常被称作过程语言 (PL)。对于一个用过程语言编写的函数，数据库服务器没有关于如何解释该函数的源文本的内建知识。因此，这个任务被交给一个了解语言细节的特殊处理器。该处理器能够自己处理所有的解析、语法分析、执行工作，或者它可以作为一种LightDB和编程语言既有实现之间的“粘合剂”。就像任何其他 C 函数一样，处理器本身是一个编译到共享对象并且按需载入的 C 语言函数。

在标准的 LightDB 发布版中，目前有两种可用的过程性语言：PL/pgSQL (Chapter 41)，PL/oraSQL (Chapter 42)。还有其他可用的过程性语言，它们未包含在核心发行版中。Appendix J提供了有关查找它们的信息。此外，用户可以定义其他语言；如何开发新的过程性语言的基础知识在 Chapter 52中介绍。

### 40.1. 安装过程语言

在每一个要使用过程语言的数据库中都必须“安装”相应的过程语言。不过安装在数据库template1中的过程语言会被后续创建的数据库自动继承，因为template1中与过程语言相关的项会被CREATE DATABASE复制。因此，数据库管理员可以决定在哪些数据库中可以使用哪些语言，并且按照选择让一些语言默认可用。

对于标准发布所提供的语言，只需要执行CREATE EXTENSION language\_name来把该语言安装在当前数据库中。下文所述的手工过程主要是为了安装没有被包装成扩展的语言。

#### 手工安装过程语言

安装一个过程语言到一个数据库中包括五个步骤，且必须由一个数据库超级用户来执行。在大部分情况下，所需的 SQL 命令应该被打包成一个“扩展”的安装脚本，这样可以用CREATE EXTENSION来执行它们。

1. 用于语言处理器的共享对象必须被编译并安装到一个合适的库目录中。这和编译和安装常规的用户定义 C 函数一样，参见Section 36.10.5 通常，语言处理器将依赖于一个实际提供编程语言引擎的外部库，如果是这样，那些外部库也应该被安装。
2. 处理器必须用下面的命令声明

```
CREATE FUNCTION handler_function_name()  
    RETURNS language_handler  
    AS 'path-to-shared-object'  
    LANGUAGE C;
```

特殊的返回类型language\_handler告诉数据库系统，这个函数不返回已定义的SQL数据类型，并且不能直接在SQL语句中使用。

3. (Optional) 可选地，语言处理器能提供一个“内联”处理器函数来执行用这种语言编写的匿名代码块 (DO命令)。如果该语言提供了一个内联函数，用类似下面的命令声明它

```
CREATE FUNCTION inline_function_name(internal)  
    RETURNS void  
    AS 'path-to-shared-object'  
    LANGUAGE C;
```

4. (Optional) 可选地，语言处理器能提供一个“验证器”函数用来检查一个函数定义的正确性而无需实际执行它。如果验证器函数存在，它将被CREATE FUNCTION调用。如果该语言提供了一个验证器函数，用类似下面的命令声明它

```
CREATE FUNCTION validator_function_name(oid)
  RETURNS void
  AS 'path-to-shared-object'
  LANGUAGE C STRICT;
```

5. 最后，PL 必须用下面的命令声明

```
CREATE [TRUSTED] LANGUAGE language_name
  HANDLER handler_function_name
  [INLINE inline_function_name]
  [VALIDATOR validator_function_name] ;
```

可选关键字 `TRUSTED` 指定该语言不授予用户无法访问的数据权限。受信任的语言是为普通数据库用户（没有超级用户特权）设计的，允许他们安全地创建函数和过程。由于 PL 函数在数据库服务器内部执行，因此应该只为不允许访问数据库服务器内部或文件系统的语言提供 `TRUSTED` 标志。语言 `PL/pgSQL`、`PL/oraSQL` 被认为是受信任的。

在一个默认的LightDB安装中，用于PL/pgSQL语言的处理器会被编译并且安装到“library”目录，此外PL/pgSQL语言本身会被安装在所有的数据库中。如果配置了Tcl支持，用于PL/Tcl和PL/TclU的处理器会被编译并且安装到库目录中，但语言本身默认不会被安装在任何数据库中。同样地，PL/Perl和PL/PerlU处理器在配置了Perl支持时被编译和安装，并且PL/PythonU处理器在配置了Python支持时被安装，但是这些语言默认都不会被安装。

---

# Chapter 41. PL/pgSQL — SQL过程语言

## 41.1. 综述

PL/pgSQL是一种用于LightDB数据库系统的可载入的过程语言。PL/pgSQL的设计目标是创建一种这样的可载入过程语言

- 可以被用来创建函数，存储过程，和触发器过程，
- 对SQL语言增加控制结构，
- 可以执行复杂计算，
- 继承所有用户定义类型、函数、存储过程和操作符，
- 可以被定义为受服务器信任，
- 便于使用。

用PL/pgSQL创建的函数可以被用在任何可以使用内建函数的地方。例如，可以创建复杂条件的计算函数并且后面用它们来定义操作符或把它们用于索引表达式。

在LightDB 9.0 和以后的版本中，PL/pgSQL是默认被安装的。但是它仍然是一种可载入模块，因此特别关注安全性的管理员可以选择移除它。

### 41.1.1. 使用PL/pgSQL的优点

SQL被LightDB和大多数其他关系数据库用作查询语言。它是可移植的并且容易学习。但是每一个SQL语句必须由数据库服务器单独执行。

这意味着你的客户端应用必须发送每一个查询到数据库服务器、等待它被处理、接收并处理结果、做一些计算，然后发送更多查询给服务器。如果你的客户端和数据库服务器不在同一台机器上，所有这些会引起进程间通信并且将带来网络负担。

通过PL/pgSQL，你可以将一整块计算和一系列查询分组在数据库服务器内部，这样就有了一种过程语言的能力并且使 SQL 更易用，但是节省了相当多的客户端/服务器通信开销。

- 客户端和服务端之间的额外往返通信被消除
- 客户端不需要的中间结果不必被整理或者在服务器和客户端之间传送
- 多轮的查询解析可以被避免

与不使用存储函数的应用相比，这能够导致可观的性能提升。

还有，通过PL/pgSQL你可以使用 SQL 中所有的数据类型、操作符和函数。

### 41.1.2. 支持的参数和结果数据类型

PL/pgSQL编写的函数可以接受服务器支持的任何标量或数组数据类型作为参数，并且它们能够返回任何这些类型的结果。它们也能接受或返回任何用名称指定的组合类型（行类型）。还可以声明一个PL/pgSQL函数为接受record，这表示任意组合类型都将作为输入，或者声明为返回record，表示结果是一种行类型，它的列由调用查询中的说明确定（如Section 7.2.1.中所讨论）。

PL/pgSQL函数可以通过使用VARIADIC标记被声明为接受数量不定的参数。如Section 36.5.中所讨论的，它的工作方式和 SQL 函数一样。



PL/pgSQL函数也可以声明为接受和返回 Section 36.2.5 中描述的多态类型，从而允许函数处理的实际数据类型发生变化 从呼叫到呼叫。示例出现在Section 41.3. 中。

PL/pgSQL函数还能够被声明为返回一个任意（可作为一个单一实例返回的）数据类型的“集合”（或表）。这样的函数通过为结果集的每个期望元素执行RETURN NEXT来产生输出，或者通过使用RETURN QUERY来输出一个查询计算的结果。

最后，如果一个PL/pgSQL函数没有可用的返回值，它可以被声明为返回void（另外一种选择是，在那种情况下它可以被写作一个过程）。

PL/pgSQL函数也能够被声明为用输出参数代替返回类型的一个显式说明。这没有为该语言增加任何基础功能，但是它常常很方便，特别是对于要返回多个值的情况。RETURNS TABLE符号也可以被用来替代RETURNS SETOF。

在Section 41.3. 和Section 41.6. 中有详细的例子。

## 41.2. PL/pgSQL的结构

通过执行CREATE FUNCTION命令，以PL/pgSQL写成的函数可以被定义到服务器中。这种命令通常看起来是这样：

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

就目前CREATE FUNCTION所关心的来说，函数体就是简单的一个字符串。通常在写函数体时，使用美元符号引用（见Section 4.1.2.4 通常比使用普通单引号语法更有帮助。如果没有美元引用，函数体中的任何单引号或者反斜线必须通过双写来转义。这一章中几乎所有的例子都在其函数体中使用美元符号引用。

PL/pgSQL是一种块结构的语言。一个函数体的完整文本必须是一个块。一个块被定义为：

```
[ <<label>> ]
[ DECLARE
  declarations ]
BEGIN
  statements
END [ label ];
```

在一个块中的每一个声明和每一个语句都由一个分号终止。如上所示，出现在另一个块中的块必须有一个分号在END之后。不过最后一个结束函数体的END不需要一个分号。

### Tip

一种常见的错误是直接在BEGIN之后写一个分号。这是不正确的并且将会导致一个语法错误。

如果你想要标识一个块以便在一个EXIT语句中使用或者标识在该块中声明的变量名，那么label是你唯一需要的。如果一个标签在END之后被给定，它必须匹配在块开始处的标签。

所有的关键词都是大小写无关的。除非被双引号引用，标识符会被隐式地转换为小写形式，就像它们在普通 SQL 命令中。

PL/pgSQL代码中的注释和普通 SQL 中的一样。一个双连字符（--）开始一段注释，它延伸到该行的末尾。一个/\*开始一段块注释，它会延伸到匹配\*/出现的位置。块注释可以嵌套。

一个块的语句节中的任何语句可以是一个子块。子块可以被用来逻辑分组或者将变量局部化为语句的一个小组。在子块的持续期间，在一个子块中声明的变量会掩盖外层块中相同名称的变量。但是如果你用块的标签限定外层变量的名字，你仍然可以访问它们。例如：

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity; -- Prints 30
    quantity := 50;
    --
    -- 创建一个子块
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity; -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity; -- Prints
50
    END;

    RAISE NOTICE 'Quantity here is %', quantity; -- Prints 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

### Note

在任何PL/pgSQL函数体的外部确实有一个隐藏的“外层块”包围着。这个块提供了该函数参数（如果有）的声明，以及某些诸如FOUND之类特殊变量（见Section 41.5.5）。外层块被标上函数的名称，这意味着参数和特殊变量可以用该函数的名称限定。

重要的是不要把PL/pgSQL中用来分组语句的BEGIN/END与用于事务控制的同名 SQL 命令弄混。PL/pgSQL的BEGIN/END只用于分组，它们不会开始或结束一个事务。有关PL/pgSQL中管理事务的信息，请参考Section 41.8 此外，一个包含EXCEPTION子句的块实际上会形成一个子事务，它可以被回滚而不影响外层事务。详见Section 41.6.8

## 41.3. 声明

在一个块中使用的所有变量必须在该块的声明小节中声明（唯一的例外是在一个整数范围上迭代的FOR循环变量会被自动声明为一个整数变量，并且相似地在一个游标结果上迭代的FOR循环变量会被自动地声明为一个记录变量）。

PL/pgSQL变量可以是任意 SQL 数据类型，例如integer、varchar和char。

这里是变量声明的一些例子：

```
user_id integer;
quantity numeric(5);
url varchar;
```

```
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

一个变量声明的一般语法是：

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | :=
| = } expression ];
```

如果给定DEFAULT子句，它会指定进入该块时分配给该变量的初始值。如果没有给出DEFAULT子句，则该变量被初始化为SQL空值。CONSTANT选项阻止该变量在初始化之后被赋值，这样它的值在块的持续期内保持不变。COLLATE选项指定用于该变量的一个排序规则（见Section 41.3.6）。如果指定了NOT NULL，对该变量赋值为空值会导致一个运行时错误。所有被声明为NOT NULL的变量必须被指定一个非空默认值。等号(=)可以被用来代替PL/SQL-兼容的:=。

一个变量的默认值会在每次进入该块时被计算并且赋值给该变量（不是每次函数调用只计算一次）。因此，例如将now()赋值给类型为timestamp的一个变量将会导致该变量具有当前函数调用的时间，而不是该函数被预编译的时间。

例子：

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

### 41.3.1. 声明函数参数

传递给函数的参数被命名为标识符\$1、\$2等等。可选地，能够为\$n参数名声明别名来增加可读性。不管是别名还是数字标识符都能用来引用参数值。

有两种方式来创建一个别名。比较好的方式是在CREATE FUNCTION命令中为参数给定一个名称。例如：

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

另一种方式是显式地使用声明语法声明一个别名。

```
name ALIAS FOR $n;
```

使用这种风格的同一个例子看起来是：

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

## Note

这两个例子并非完全等效。在第一种情况中，subtotal可以被引用为sales\_tax.subtotal，但在第二种情况中它不能这样引用（如果我们为内层块附加了一个标签，subtotal则可以用那个标签限定）。

更多一些例子：

```
CREATE FUNCTION instr(vchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- 这里是一些使用 v_string 和 index 的计算
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION concat_selected_fields(in_t sometable) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

当一个PL/pgSQL函数被声明为带有输出参数，输出参数可以用普通输入参数相同的方式被给定\$n名称以及可选的别名。一个输出参数实际上是一个最初为 NULL 的变量，它应当在函数的执行期间被赋值。该参数的最终值就是要被返回的东西。例如，sales-tax 例子也可以用这种方式来做：

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

注意我们忽略了RETURNS real — 我们也可以包括它，但是那将是冗余。

当返回多个值时，输出参数最有用。一个小例子是：

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

如Section 36.5.4中所讨论的，这实际上为该函数的结果创建了一个匿名记录类型。如果给定了个RETURNS子句，它必须RETURNS record。

声明一个PL/pgSQL函数的另一种方式是用RETURNS TABLE，例如：

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
```

```

RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;

```

这和声明一个或多个OUT参数并且指定RETURNS SETOF sometype完全等效。

当PL/pgSQL函数的返回类型被声明为多态类型（参见Section 36.2.5 时，一个特殊的参数 \$0 已创建。它的数据类型是函数的实际返回类型，从实际输入类型推导出来。\$0被初始化为空并且不能被该函数修改，因此它能够被用来保持可能需要的返回值，不过这不是必须的。\$0也可以被给定一个别名。例如，这个函数工作在任何具有一个+操作符的数据类型上：

```

CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;

```

通过声明一个或多个输出参数为多态类型可以得到同样的效果。在这种情况下，不使用特殊的\$0参数，输出参数本身就用作相同的目的。例如：

```

CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;

```

在实践中，使用anycompatible类型系列声明多态函数可能更有用，以便将输入参数自动提升为公共类型。例如：

```

CREATE FUNCTION add_three_values(v1 anycompatible, v2 anycompatible, v3
    anycompatible)
RETURNS anycompatible AS $$
BEGIN
    RETURN v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;

```

在此示例中，调用如

```
SELECT add_three_values(1, 2, 4.7);
```

将工作，自动将整数输入提升为数字。使用anyelement的函数需要您手动将三个输入转换为相同的类型。

## 41.3.2. ALIAS

```
newname ALIAS FOR oldname;
```

ALIAS语法比前一节中建议的更一般化：你可以为任意变量声明一个别名，而不只是函数参数。其主要实际用途是为预先决定了名称的变量分配一个不同的名称，例如在一个触发器过程中的NEW或OLD。

例子：

```
DECLARE
    prior ALIAS FOR old;
    updated ALIAS FOR new;
```

因为ALIAS创造了两种不同的方式来命名相同的对象，如果对其使用不加限制就会导致混淆。最好只把它用来覆盖预先决定的名称。

### 41.3.3. 复制类型

```
variable%TYPE
```

%TYPE提供了一个变量或表列的数据类型。你可以用它来声明将保持数据库值的变量。例如，如果你在users中有一个名为user\_id的列。要定义一个与users.user\_id具有相同数据类型的变量：

```
user_id users.user_id%TYPE;
```

通过使用%TYPE，你不需要知道你要引用的结构的实际数据类型，而且最重要地，如果被引用项的数据类型在未来被改变（例如你把user\_id的类型从integer改为real），你不需要改变你的函数定义。

%TYPE在多态函数中特别有价值，因为内部变量所需的数据类型能在两次调用时改变。可以把%TYPE应用在函数的参数或结果占位符上来创建合适的变量。

### 41.3.4. 行类型

```
name table_name%ROWTYPE;
name composite_type_name;
```

一个组合类型的变量被称为一个行变量（或行类型变量）。这样一个变量可以保持一个SELECT或FOR查询结果的一整行，前提是查询的列集合匹配该变量被声明的类型。该行值的各个域可以使用通常的点号标记访问，例如rowvar.field。

通过使用table\_name%ROWTYPE标记，一个行变量可以被声明为具有和一个现有表或视图的行相同的类型。它也可以通过给定一个组合类型名称来声明（因为每一个表都有一个相关联的具有相同名称的组合类型，所以在LightDB中实际上写不写%ROWTYPE都没有关系。但是带有%ROWTYPE的形式可移植性更好）。

一个函数的参数可以是组合类型（完整的表行）。在这种情况下，相应的标识符\$n将是一个行变量，并且可以从中选择域，例如\$1.user\_id。

这里是一个使用组合类型的例子。table1和table2是已有的表，它们至少有以下提到的域：

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
```

```

SELECT * INTO t2_row FROM table2 WHERE ... ;
RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;

```

### 41.3.5. 记录类型

name RECORD;

记录变量和行类型变量类似，但是它们没有预定义的结构。它们采用在一个SELECT或FOR命令期间为其赋值的行的真实行结构。一个记录变量的子结构能在每次它被赋值时改变。这样的结果是直到一个记录变量第一次被赋值之前，它都没有子结构，并且任何尝试访问其中一个域都会导致一个运行时错误。

注意RECORD并非一个真正的数据类型，只是一个占位符。我们也应该认识到当一个PL/pgSQL函数被声明为返回类型record，这与一个记录变量并不是完全相同的概念，即便这样一个函数可能会用一个记录变量来保持其结果。在两种情况下，编写函数时都不知道真实的行结构，但是对于一个返回record的函数，当调用查询被解析时就已经决定了真正的结构，而一个行变量能够随时改变它的行结构。

### 41.3.6. PL/pgSQL变量的排序规则

当一个PL/pgSQL函数有一个或多个可排序数据类型的参数时，为每一次函数调用都会基于赋值给实参的排序规则来确定出一个排序规则，如Section 21.2中所述。如果一个排序规则被成功地确定（即在参数之间隐式排序规则没有冲突），那么所有的可排序参数会被当做隐式具有那个排序规则。这将在函数中影响行为受到排序规则影响的操作。例如，考虑

```

CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;

```

less\_than的第一次使用将会采用text\_field\_1和text\_field\_2共同的排序规则进行比较，而第二次使用将采用C排序规则。

此外，被确定的排序规则也被假定为任何可排序数据类型本地变量的排序规则。因此，当这个函数被写为以下形式时，它工作将不会有什么不同

```

CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;

```

如果没有可排序数据类型的参数，或者不能为它们确定共同的排序规则，那么参数和本地变量会使用它们数据类型的默认排序规则（通常是数据库的默认排序规则，但是可能不同于域类型的变量）。

通过在一个可排序数据类型的本地变量的声明中包括COLLATE选项，可以为它指定一个不同的排序规则，例如

```
DECLARE
    local_a text COLLATE "en_US";
```

这个选项会覆盖根据上述规则被给予该变量的排序规则。

还有，如果一个函数想要强制在一个特定操作中使用一个特定排序规则，当然可以在该函数内部写一个显式的COLLATE子句。例如：

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

这会覆盖表达式中使用的表列、参数或本地变量相关的排序规则，就像在纯 SQL 命令中发生的一样。

## 41.4. 表达式

PL/pgSQL语句中用到的所有表达式会被服务器的主SQL执行器处理。例如，当你写一个这样的PL/pgSQL语句时

```
IF expression THEN ...
```

PL/pgSQL将通过给主 SQL 引擎发送一个查询

```
SELECT expression
```

来计算该表达式。如Section 41.11.中所详细讨论的，在构造该SELECT命令时，PL/pgSQL变量名的每一次出现会被查询参数所替换。这允许SELECT的查询计划仅被准备一次并且被重用于之后的对于该变量不同值的计算。因此，在一个表达式第一次被使用时实际发生的本质上是一个PREPARE命令。例如，如果我们已经声明了两个整数变量x和y，并且我们写了

```
IF x < y THEN ...
```

在现象之后发生的等效于

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

并且然后为每一次IF语句的执行，这个预备语句都会被EXECUTE，执行时使用变量的当前值作为参数值。通常这些细节对于一个PL/pgSQL用户并不重要，但是在尝试诊断一个问题时了解它们很有用。更多信息可见Section 41.11.2

## 41.5. 基本语句

在本节及以下节中，我们描述了PL/pgSQL明确理解的所有语句类型。任何未被识别为这些语句类型之一的内容都被认为是SQL命令，并被发送到主数据库引擎执行，如Section 41.5.和Section 41.5.所述。



## 41.5.1. 赋值

为一个PL/pgSQL变量赋一个值可以被写为：

```
variable { := | = } expression;
```

正如以前所解释的，这样一个语句中的表达式被以一个 SQL SELECT命令被发送到主数据库引擎的方式计算。该表达式必须得到一个单一值（如果该变量是一个行或记录变量，它可能是一个行值）。该目标变量可以是一个简单变量（可以选择用一个块名限定）、一个行或记录变量的域或是一个简单变量或域的数组的元素或者数组的切片。等号（=）可以被用来代替 PL/SQL-兼容的 :=。

如果该表达式的结果数据类型不匹配变量的数据类型，该值将被强制为变量的类型，就好像做了赋值造型一样（见Section 10.4）。如果没有用于所涉及到的数据类型的赋值造型可用，PL/pgSQL解释器将尝试以文本的方式转换结果值，也就是在应用结果类型的输出函数之后再应用变量类型的输入函数。注意如果结果值的字符串形式无法被输入函数所接受，这可能会导致由输入函数产生的运行时错误。

例子：

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

## 41.5.2. 执行没有结果的命令

对于任何不返回行的 SQL 命令，例如 INSERT 没有 RETURNING 子句，您可以在 PL/pgSQL 函数中执行该命令，只需编写该命令即可。

在命令文本中出现的任何 PL/pgSQL 变量名称都被视为参数，然后该变量的当前值在运行时作为参数值提供。这与前面描述的表达式处理完全相同；有关详细信息，请参阅 Section 41.11.1

在以这种方式执行 SQL 命令时，PL/pgSQL 可以缓存并重用命令的执行计划，如 Section 41.11.2中所述。

有时候需要计算表达式或 SELECT 查询，但是丢弃结果，例如在调用具有副作用但没有有用结果值的函数时。在 PL/pgSQL 中，可以使用 PERFORM 语句来实现这一点：

```
PERFORM query;
```

这会执行query并且丢弃掉结果。以写一个SQL SELECT命令相同的方式写该query，并且将初始的关键词SELECT替换为PERFORM。对于WITH查询，使用PERFORM并且接着把该查询放在圆括号中（在这种情况下，该查询只能返回一行）。PL/pgSQL变量将被替换到该查询中，正如上面描述的查询，并且计划被以相同的方式被缓存。还有，如果该查询产生至少一行，特殊变量FOUND会被设置为真，而如果它不产生行则设置为假（见Section 41.5.5）。

### Note

我们可能期望直接写SELECT能实现这个结果，但是当前唯一被接受的方式是PERFORM。一个能返回行的 SQL 命令（例如SELECT）将被当成一个错误拒绝，除非它像下一节中讨论的有一个INTO子句。

一个例子：

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

### 41.5.3. 执行一个有单一行结果的命令

一个产生单一行（可能有多个列）的 SQL 命令的结果可以被赋值给一个记录变量、行类型变量或标量变量列表。这通过书写基础 SQL 命令并增加一个INTO子句来达成。例如：

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

其中target可以是一个记录变量、一个行变量或一个有逗号分隔的简单变量和记录/行域列表。PL/pgSQL变量将被替换到该命令的剩余部分中（即除了INTO子句所有的），正如之前描述的，计划以同样的方式缓存。这对SELECT、带有RETURNING的INSERT/UPDATE/DELETE以及返回行集的某些工具命令（例如EXPLAIN）。除了INTO子句，SQL命令和它在PL/pgSQL之外的写法一样。

#### Tip

注意带INTO的SELECT的这种解释和LightDB常规的SELECT INTO命令有很大的不同，后者的INTO目标是一个新创建的表。如果你想要在一个PL/pgSQL函数中从一个SELECT的结果创建一个表，请使用语法CREATE TABLE ... AS SELECT。

如果一个行变量或一个变量列表被用作目标，该命令的结果列必须完全匹配该结果的结构，包括数量和数据类型，否则会发生一个运行时错误。当一个记录变量是目标时，它会自动地把自身配置成命令的结果列组成的行类型。

INTO子句几乎可以出现在 SQL 命令中的任何位置。通常它被写成刚好在SELECT命令中的select\_expressions列表之前或之后，或者在其他命令类型的命令最后。我们推荐你遵循这种惯例，以防PL/pgSQL的解析器在未来的版本中变得更严格。

如果STRICT没有在INTO子句中被指定，那么target将被设置为该命令返回的第一个行，或者在该命令不返回行时设置为空（注意除非使用了ORDER BY，否则“第一行”的界定并不清楚）。第一行之后的任何结果行都会被抛弃。你可以检查特殊的FOUND变量（见Section 41.5.5 来确定是否返回了一行：

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

如果指定了STRICT选项，该命令必须刚好返回一行或者将会报告一个运行时错误，该错误可能是NO\_DATA\_FOUND（没有行）或TOO\_MANY\_ROWS（多于一行）。如果你希望捕捉该错误，可以使用一个异常块，例如：

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE EXCEPTION 'employee % not found', myname;
    WHEN TOO_MANY_ROWS THEN
        RAISE EXCEPTION 'employee % not unique', myname;
```

END;

成功执行一个带STRICT的命令总是会将FOUND置为真。

对于带有RETURNING的INSERT/UPDATE/DELETE，即使没有指定STRICT，PL/pgSQL也会针对多于一个返回行的情况报告一个错误。这是因为没有类似于ORDER BY的选项可以用来决定应该返回哪个被影响的行。

如果为该函数启用了If print\_strict\_params，那么当因为 STRICT的要求没有被满足而抛出一个错误时，该错误消息 的DETAIL将包括传递给该命令的参数信息。可以通过设置 plpgsql.print\_strict\_params为所有函数更改 print\_strict\_params设置，但是只有修改后被编译的函数 才会生效。也可以使用一个编译器选项来为一个函数启用它，例如：

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END;
$$ LANGUAGE plpgsql;
```

失败时，这个函数会产生一个这样的错误消息

```
ERROR: query returned no rows
DETAIL: parameters: $1 = 'nosuchuser'
CONTEXT: PL/pgSQL function get_userid(text) line 6 at SQL statement
```

### Note

STRICT选项匹配 Oracle PL/SQL 的SELECT INTO和相关语句的行为。

要处理需要从 SQL 查询中处理多个结果行的情况，请参阅 Section 41.6.6

## 41.5.4. 执行动态命令

很多时候你将想要在PL/pgSQL函数中产生动态命令，也就是每次执行中会涉及到不同表或不同数据类型的命令。PL/pgSQL通常对于命令所做的缓存计划尝试（如Section 41.11.2中讨论）在这种情境下无法工作。要处理这一类问题，需要提供EXECUTE语句：

```
EXECUTE [IMMEDIATE] command-string [ INTO [STRICT] target ] [ USING expression
[, ... ] ];
```

其中command-string是一个能得到一个包含要被执行命令字符串（类型text）的表达式。可选的target是一个记录变量、一个行变量或者一个逗号分隔的简单变量以及记录/行域的列表，该命令的结果将存储在其中。可选的USING表达式提供要被插入到该命令中的值。

在计算得到的命令字符串中，不会做PL/pgSQL变量的替换。任何所需的变量值必须在命令字符串被构造时被插入其中，或者你可以使用下面描述的参数。

还有，对于通过EXECUTE执行的命令不会有计划被缓存。该命令反而在每次运行时都会被做计划。因此，该命令字符串可以在执行不同表和列上动作的函数中被动态创建。

INTO子句指定一个返回行的 SQL 命令的结果应该被赋值到哪里。如果提供了一个行或变量列表，它必须完全匹配查询命令结果的结构（当一个记录变量被提供时，它会自动把它自己配置为匹配结果结构）。如果返回多个行，只有第一个行会被赋值给INTO变量。如果没有返回行，NULL 会被赋值给INTO变量。如果没有指定INTO变量，该查询结果会被抛弃。

如果给出了STRICT选项，除非该查询刚好产生一行，否则将会报告一个错误。

命令字符串可以使用参数值，它们在命令中用\$1、\$2等引用。这些符号引用在USING子句中提供的值。这种方法常常更适用于把数据值作为文本插入到命令字符串中：它避免了将该值转换为文本以及转换回来的运行时负荷，并且它更不容易被 SQL 注入攻击，因为不需要引用或转义。一个例子是：

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
      INTO c
      USING checked_user, checked_date;
```

需要注意的是，参数符号只能用于数据值 — 如果想要使用动态决定的表名或列名，你必须将它们以文本形式插入到命令字符串中。例如，如果前面的那个查询需要在一个动态选择的表上执行，你可以这么做：

```
EXECUTE 'SELECT count(*) FROM '
      || quote_ident(tabname)
      || ' WHERE inserted_by = $1 AND inserted <= $2'
      INTO c
      USING checked_user, checked_date;
```

一种更干净的方法是使用format()的 %I规范，插入自带引号的表名或者列名：

```
EXECUTE format('SELECT count(*) FROM %I '
      'WHERE inserted_by = $1 AND inserted <= $2', tabname)
      INTO c
      USING checked_user, checked_date;
```

另一个关于参数符号的限制是，它们只能在可优化的命令SELECT、INSERT、UPDATE和DELETE中有效。在另一种语句类型（通常被称为实用语句）中，即使值是数据值，你也必须将它们以文本形式插入。

在上面第一个例子中，带有一个简单的常量命令字符串和一些USING参数的EXECUTE命令在功能上等效于直接用PL/pgSQL写的命令，并且允许自动发生PL/pgSQL变量替换。重要的不同之处在于，EXECUTE会在每一次执行时根据当前的参数值重新规划该命令，而PL/pgSQL则是创建一个通用计划并且将其缓存以便重用。在最佳计划强依赖于参数值的情况下，使用EXECUTE来明确地保证不会选择一个通用计划是很有帮助的。

EXECUTE目前不支持SELECT INTO。但是可以执行一个纯的SELECT命令并且指定INTO作为EXECUTE本身的一部分。

为了兼容Oracle数据库语法，新增支持EXECUTE IMMEDIATE语法，这个特性仅是语法糖。

### Note

PL/pgSQL中的EXECUTE语句与EXECUTE LightDB服务器支持的 SQL 语句无关。服务器的EXECUTE语句不能直接在PL/pgSQL函数中使用（并且也没有必要）。

Example 41.1. 在动态查询中引用值

在使用动态命令时经常不得不处理单引号的转义。我们推荐在函数体中使用美元符号引用来引用固定的文本（如果你没有使用美元符号界定的老代码，请参考Section 41.12. 中的概述，这样在把上述代码转换成更合理的模式时会省力些）。

动态值需要被小心地处理，因为它们可能包含引号字符。一个使用 `format()` 的例子（这假设你用美元符号引用了函数体，因此引号不需要被双写）：

```
EXECUTE format('UPDATE tbl SET %I = $1 '
              'WHERE key = $2', colname) USING newvalue, keyvalue;
```

还可以直接调用引用函数：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ', = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

这个例子展示了`quote_ident`和`quote_literal`函数的使用（见Section 9.4）。为了安全，在进行一个动态查询中的插入之前，包含列或表标识符的表达式应该通过`quote_ident`被传递。如果表达式包含在被构造出的命令中应该是字符串的值时，它应该通过`quote_literal`被传递。这些函数采取适当的步骤来分别返回被封闭在双引号或单引号中的文本，其中任何嵌入的特殊字符都会被正确地转义。

因为`quote_literal`被标记为`STRICT`，当用一个空参数调用时，它总是会返回空。在上面的例子中，如果`newvalue`或`keyvalue`为空，整个动态查询字符串会变成空，导致从`EXECUTE`得到一个错误。可以通过使用`quote_nullable`函数来避免这种问题，它工作起来和`quote_literal`相同，除了用空参数调用时会返回一个字符串`NULL`。例如：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ', = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

如果正在处理的参数值可能为空，那么通常应该用`quote_nullable`来代替`quote_literal`。

通常，必须小心地确保查询中的空值不会递送意料之外的结果。例如如果`keyvalue`为空，下面的`WHERE`子句

```
'WHERE key = ' || quote_nullable(keyvalue)
```

永远不会成功，因为在`=`操作符中使用空操作数得到的结果总是为空。如果想让空和一个普通键值一样工作，你应该将上面的命令重写成

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

（目前，`IS NOT DISTINCT FROM`的处理效率不如`=`，因此只有在非常必要时才这样做。关于空和`IS DISTINCT`的详细信息请见Section 9.2）。

请注意美元符号引用只对引用固定文本有用。尝试写出下面这个例子是一个非常糟糕的主意：

```
EXECUTE 'UPDATE tbl SET '
      || quote_ident(colname)
      || ' = $$'
      || newvalue
      || '$$ WHERE key = '
      || quote_literal(keyvalue);
```

因为如果newvalue的内容碰巧含有\$\$，那么这段代码就会出问题。同样的缺点可能适用于你选择的任何其他美元符号引用定界符。因此，要想安全地引用事先不知道的文本，必须恰当地使用quote\_literal、quote\_nullable或quote\_ident。

动态 SQL 语句也可以使用format（见Section 9.4.1）函数来安全地构造。例如：

```
EXECUTE format('UPDATE tbl SET %I = %L '
      'WHERE key = %L', colname, newvalue, keyvalue);
```

%I等效于quote\_ident并且 %L等效于quote\_nullable。format函数可以和 USING子句一起使用：

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
      USING newvalue, keyvalue;
```

这种形式更好，因为变量被以它们天然的数据类型格式处理，而不是无条件地把它们转换成文本并且通过%L引用它们。这也效率更高。

动态命令和EXECUTE的一个更大的例子可以在Example 41.10中找到，它会构建并且执行一个CREATE FUNCTION命令来定义一个新的函数。

EXECUTE IMMEDIATE使用样例。

```
-- create table
CREATE TABLE test_employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50)
);
INSERT INTO test_employees (first_name) VALUES ('John'), ('Jane'), ('Alice'),
('Bob'), ('David');

-- create function
CREATE OR REPLACE FUNCTION get_employee_name_func(p_employee_id INTEGER)
RETURNS VARCHAR AS $$
DECLARE
    sql_query TEXT;
    emp_name VARCHAR;
BEGIN
    sql_query := 'SELECT first_name FROM test_employees WHERE employee_id = $1';
    EXECUTE IMMEDIATE sql_query INTO emp_name USING p_employee_id;

    RAISE NOTICE 'Employee name is: %', emp_name;
    RETURN emp_name;
END;
$$ LANGUAGE plpgsql;
```

## 41.5.5. 获得结果状态

有好几种方法可以判断一条命令的效果。第一种方法是使用GET DIAGNOSTICS命令，其形式如下：

```
GET [ CURRENT ] DIAGNOSTICS variable { = | := } item [ , ... ];
```

这条命令允许检索系统状态指示符。CURRENT是一个噪声词（另见Section 41.6.8. 中的GET STACKED DIAGNOSTICS）。每个item是一个关键字，它标识一个要被赋予给指定变量的状态值（变量应具有正确的数据类型来接收状态值）。Table 41. 中展示了当前可用的状态项。冒号等号（:=）可以被用来取代 SQL 标准的=符号。例如：

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

Table 41.1. 可用的诊断项

名称	类型	描述
ROW_COUNT	bigint	最近的SQL命令处理的行数
PG_CONTEXT	text	描述当前调用栈的文本行 (见Section 41.6.9

第二种判断命令效果的方法是检查一个名为FOUND的boolean类型的特殊变量。在每一次PL/pgSQL函数调用时，FOUND开始都为假。它的值会被下面的每一种类型的语句设置：

- 如果一个SELECT INTO语句赋值了一行，它将把FOUND设置为真，如果没有返回行则将之设置为假。
- 如果一个PERFORM语句生成（并且抛弃）一行或多行，它将把FOUND设置为真，如果没有产生行则将之设置为假。
- 如果UPDATE、INSERT以及DELETE语句影响了至少一行，它们会把FOUND设置为真，如果没有影响行则将之设置为假。
- 如果一个FETCH语句返回了一行，它将把FOUND设置为真，如果没有返回行则将之设置为假。
- 如果一个MOVE语句成功地重定位了游标，它将会把FOUND设置为真，否则设置为假。
- 如果一个FOR或FOREACH语句迭代了一次或多次，它将会把FOUND设置为真，否则设置为假。当循环退出时，FOUND用这种方式设置：在循环执行中，尽管FOUND可能被循环体中的其他语句的执行所改变，但它不会被循环语句修改。
- 如果查询返回至少一行，RETURN QUERY和RETURN QUERY EXECUTE语句会把FOUND设为真，如果没有返回行则设置为假。

其他的PL/pgSQL语句不会改变FOUND的状态。尤其需要注意的一点是：EXECUTE会修改GET DIAGNOSTICS的输出，但不会修改FOUND的输出。

FOUND是每个PL/pgSQL函数的局部变量；任何对它的修改只影响当前的函数。

## 41.5.6. 什么也不做

有时一个什么也不做的占位语句也很有用。例如，它能够指示 if/then/else 链中故意留出的空分支。可以使用NULL语句达到这个目的：

```
NULL;
```

例如，下面的两段代码是等价的：

```

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL; -- 忽略错误
END;

```

```

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN -- 忽略错误
END;

```

究竟使用哪一种取决于各人的喜好。

### Note

在 Oracle 的 PL/SQL 中，不允许出现空语句列表，并且因此在这种情况下必须使用 NULL 语句。而 PL/pgSQL 允许你什么也不写。

## 41.6. 控制结构

控制结构可能是 PL/pgSQL 中最有用的（以及最重要）的部分了。利用 PL/pgSQL 的控制结构，你可以以非常灵活而且强大的方法操纵 LightDB 的数据。

### 41.6.1. 从一个函数返回

有两个命令让我们能够从函数中返回数据：RETURN 和 RETURN NEXT。

#### 41.6.1.1. RETURN

```
RETURN expression;
```

带有一个表达式的 RETURN 用于终止函数并把 expression 的值返回给调用者。这种形式被用于不返回集合的 PL/pgSQL 函数。

如果一个函数返回一个标量类型，表达式的结果将被自动转换成函数的返回类型。但是要返回一个复合（行）值，你必须写一个正好产生所需列集合的表达式。这可能需要使用显式造型。

如果你声明带输出参数的函数，那么就只需要写不带表达式的 RETURN。输出参数变量的当前值将被返回。

如果你声明函数返回 void，一个 RETURN 语句可以被用来提前退出函数；但是不要在 RETURN 后面写一个表达式。

一个函数的返回值不能是未定义。如果控制到达了函数最顶层的块而没有碰到一个 RETURN 语句，那么会发生一个运行时错误。不过，这个限制不适用于带输出参数的函数以及返回 void 的函数。在这些情况中，如果顶层的块结束，将自动执行一个 RETURN 语句。

一些例子：



```

-- 返回一个标量类型的函数
RETURN 1 + 2;
RETURN scalar_var;

-- 返回一个组合类型的函数
RETURN composite_type_var;
RETURN (1, 2, 'three'::text); -- 必须把列造型成正确的类型

```

#### 41.6.1.2. RETURN NEXT以及RETURN QUERY

```

RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];

```

当一个PL/pgSQL函数被声明为返回SETOF sometype，那么遵循的过程则略有不同。在这种情况下，要返回的个体项被用一个RETURN NEXT或者RETURN QUERY命令的序列指定，并且接着会用一个不带参数的最终RETURN命令来指示这个函数已经完成执行。RETURN NEXT可以被用于标量和复合数据类型；对于复合类型，将返回一个完整的结果“表”。RETURN QUERY将执行一个查询的结果追加到一个函数的结果集中。在一个单一的返回集合的函数中，RETURN NEXT和RETURN QUERY可以被随意地混合，这样它们的结果将被串接起来。

RETURN NEXT和RETURN QUERY实际上不会从函数中返回 — 它们简单地向函数的结果集中追加零或多行。然后会继续执行PL/pgSQL函数中的下一条语句。随着后继的RETURN NEXT和RETURN QUERY命令的执行，结果集就建立起来了。最后一个RETURN（应该没有参数）会导致控制退出该函数（或者你可以让控制到达函数的结尾）。

RETURN QUERY有一种变体RETURN QUERY EXECUTE，它可以动态指定要被执行的查询。可以通过USING向计算出的查询字符串插入参数表达式，这和EXECUTE命令中的方式相同。

如果你声明函数带有输出参数，只需要写不带表达式的RETURN NEXT。在每一次执行时，输出参数变量的当前值将被保存下来用于最终返回为结果的一行。注意为了创建一个带有输出参数的集合返回函数，在有多个输出参数时，你必须声明函数为返回SETOF record；或者如果只有一个类型为sometype的输出参数时，声明函数为SETOF sometype。

下面是一个使用RETURN NEXT的函数例子：

```

CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- 这里可以做一些处理
        RETURN NEXT r; -- 返回 SELECT 的当前行
    END LOOP;
    RETURN;
END;
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();

```

这里是一个使用RETURN QUERY的函数的例子：

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                  FROM flight
                  WHERE flightdate >= $1
                  AND flightdate < ($1 + 1);

    -- 因为执行还未结束，我们可以检查是否有行被返回
    -- 如果没有就抛出异常。
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No flight at %.', $1;
    END IF;

    RETURN;
END;
$BODY$
LANGUAGE plpgsql;
```

-- 返回可用的航班或者在没有可用航班时抛出异常。  
SELECT \* FROM get\_available\_flightid(CURRENT\_DATE);

### Note

如上所述，目前RETURN NEXT和RETURN QUERY的实现在从函数返回之前会把整个结果集都保存起来。这意味着如果一个PL/pgSQL函数生成一个非常大的结果集，性能可能会很差：数据将被写到磁盘上以避免内存耗尽，但是函数本身在整个结果集都生成之前不会退出。将来的PL/pgSQL版本可能会允许用户定义没有这种限制的集合返回函数。目前，数据开始被写入到磁盘的时机由配置变量work\_mem控制。拥有足够内存来存储大型结果集的管理员可以考虑增大这个参数。

## 41.6.2. 从过程中返回

过程没有返回值。因此，过程的结束可以不用RETURN语句。 如果想用一个RETURN语句提前退出代码，只需写一个没有表达式的RETURN。

如果过程有输出参数，那么输出参数最终的值会被返回给调用者。

## 41.6.3. 调用存储过程

PL/pgSQL函数，存储过程或DO块可以使用CALL调用存储过程。输出参数的处理方式与纯SQL中CALL的工作方式不同。存储过程的每个OUT 或者 INOUT参数必须和CALL语句中的变量对应，并且无论存储过程返回什么，都会在返回后赋值给该变量。例如：

```
CREATE PROCEDURE triple(INOUT x int)
LANGUAGE plpgsql
AS $$
BEGIN
    x := x * 3;
END;
```

```

$$;

DO $$
DECLARE myvar int := 5;
BEGIN
    CALL triple(myvar);
    RAISE NOTICE 'myvar = %', myvar; -- prints 15
END;
$$;

```

## 41.6.4. 条件

IF和CASE语句让你可以根据某种条件执行二选其一的命令。PL/pgSQL有三种形式的IF:

- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

以及两种形式的CASE:

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

### 41.6.4.1. IF-THEN

```

IF boolean-expression THEN
    statements
END IF;

```

IF-THEN语句是IF的最简单形式。如果条件为真，在THEN和END IF之间的语句将被执行。否则，将忽略它们。

例子:

```

IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;

```

### 41.6.4.2. IF-THEN-ELSE

```

IF boolean-expression THEN
    statements
ELSE
    statements
END IF;

```

IF-THEN-ELSE语句对IF-THEN进行了增加，它让你能够指定一组在条件不为真时应该被执行的语句（注意这也包括条件为 NULL 的情况）。

例子:

```

IF parentid IS NULL OR parentid = ''

```

```

THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;

IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;

```

### 41.6.4.3. IF-THEN-ELSIF

```

IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...
]
]
[ ELSE
    statements ]
END IF;

```

有时会有多于两种选择。IF-THEN-ELSIF则提供了一个简便的方法来检查多个条件。IF条件会被一个接一个测试，直到找到第一个为真的。然后执行相关语句，然后控制会被交给END IF之后的下一个语句（后续的任何IF条件不会被测试）。如果没有一个IF条件为真，那么ELSE块（如果有）将被执行。

这里有一个例子：

```

IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- 嗯，唯一的其他可能性是数字为空
    result := 'NULL';
END IF;

```

关键词ELSIF也可以被拼写成ELSEIF。

另一个可以完成相同任务的方法是嵌套IF-THEN-ELSE语句，如下例：

```

IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    
```

```

    END IF;
END IF;

```

不过，这种方法需要为每个IF都写一个匹配的END IF，因此当有很多选择时，这种方法比使用ELSIF要麻烦得多。

#### 41.6.4.4. 简单CASE

```

CASE search-expression
  WHEN expression [, expression [ ... ]] THEN
    statements
  [ WHEN expression [, expression [ ... ]] THEN
    statements
    ... ]
  [ ELSE
    statements ]
END CASE;

```

CASE的简单形式提供了基于操作数等值判断的有条件执行。search-expression会被计算（一次）并且一个接一个地与WHEN子句中的每个expression比较。如果找到一个匹配，那么相应的statements会被执行，并且接着控制会被交给END CASE之后的下一个语句（后续的WHEN表达式不会被计算）。如果没有找到匹配，ELSE 语句会被执行。但是如果ELSE不存在，将会抛出一个CASE\_NOT\_FOUND异常。

这里是一个简单的例子：

```

CASE x
  WHEN 1, 2 THEN
    msg := 'one or two';
  ELSE
    msg := 'other value than one or two';
END CASE;

```

#### 41.6.4.5. 搜索CASE

```

CASE
  WHEN boolean-expression THEN
    statements
  [ WHEN boolean-expression THEN
    statements
    ... ]
  [ ELSE
    statements ]
END CASE;

```

CASE的搜索形式基于布尔表达式真假的有条件执行。每一个WHEN子句的boolean-expression会被依次计算，直到找到一个得到真的。然后相应的statements会被执行，并且接下来控制会被传递给END CASE之后的下一个语句（后续的WHEN表达式不会被计算）。如果没有找到为真的结果，ELSE statements会被执行。但是如果ELSE不存在，那么将会抛出一个CASE\_NOT\_FOUND异常。

这里是一个例子：

```

CASE
  WHEN x BETWEEN 0 AND 10 THEN

```

```

        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;

```

这种形式的CASE整体上等价于IF-THEN-ELSIF，不同之处在于CASE到达一个被忽略的ELSE子句时会导致一个错误而不是什么也不做。

## 41.6.5. 简单循环

使用LOOP、EXIT、CONTINUE、WHILE、FOR和FOREACH语句，你可以安排PL/pgSQL重复一系列命令。

### 41.6.5.1. LOOP

```

[ <<label>> ]
LOOP
    statements
END LOOP [ label ];

```

LOOP定义一个无条件的循环，它会无限重复直到被EXIT或RETURN语句终止。可选的label可以被EXIT和CONTINUE语句用在嵌套循环中指定这些语句引用的是哪一层循环。

### 41.6.5.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

如果没有给出label，那么最内层的循环会被终止，然后跟在END LOOP后面的语句会被执行。如果给出了label，那么它必须是当前或者更高层的嵌套循环或者语句块的标签。然后该命名循环或块就会被终止，并且控制会转移到该循环/块相应的END之后的语句上。

如果指定了WHEN，只有boolean-expression为真时才会发生循环退出。否则，控制会转移到EXIT之后的语句。

EXIT可以被用在所有类型的循环中，它并不限于在无条件循环中使用。

在和BEGIN块一起使用时，EXIT会把控制交给块结束后的下一个语句。需要注意的是，一个标签必须被用于这个目的；一个没有被标记的EXIT永远无法被认为与一个BEGIN块匹配（这种状况从LightDB 8.4 之前的发布就已经开始改变。这可能允许一个未被标记的EXIT匹配一个BEGIN块）。

例子：

```

LOOP
    -- 一些计算
    IF count > 0 THEN
        EXIT; -- 退出循环
    END IF;
END LOOP;

LOOP
    -- 一些计算
    EXIT WHEN count > 0; -- 和前一个例子相同的结果
END LOOP;

```

```

<<ablock>>
BEGIN
  -- 一些计算
  IF stocks > 100000 THEN
    EXIT ablock; -- 导致从 BEGIN 块中退出
  END IF;
  -- 当stocks > 100000时，这里的计算将被跳过
END;

```

### 41.6.5.3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

如果没有给出label，最内层循环的下次迭代会开始。也就是，循环体中剩余的所有语句将被跳过，并且控制会返回到循环控制表达式（如果有）来决定是否需要另一次循环迭代。如果label存在，它指定应该继续执行的循环的标签。

如果指定了WHEN，该循环的下次迭代只有在boolean-expression为真时才会开始。否则，控制会传递给CONTINUE后面的语句。

CONTINUE可以被用在所有类型的循环中，它并不限于在无条件循环中使用。

例子：

```

LOOP
  -- 一些计算
  EXIT WHEN count > 100;
  CONTINUE WHEN count < 50;
  -- 一些用于 count IN [50 .. 100] 的计算
END LOOP;

```

### 41.6.5.4. WHILE

```

[ <<label>> ]
WHILE boolean-expression LOOP
  statements
END LOOP [ label ];

```

只要boolean-expression被计算为真，WHILE语句就会重复一个语句序列。在每次进入到循环体之前都会检查该表达式。

例如：

```

WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
  -- 这里是一些计算
END LOOP;

WHILE NOT done LOOP
  -- 这里是一些计算
END LOOP;

```

### 41.6.5.5. FOR（整型变体）

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

这种形式的FOR会创建一个在一个整数范围上迭代的循环。变量name会自动定义为类型integer并且只在循环内存在（任何该变量名的现有定义在此循环内都将被忽略）。给出范围上下界的两个表达式在进入循环的时候计算一次。如果没有指定BY子句，迭代步长为1，否则步长是BY中指定的值，该值也只在循环进入时计算一次。如果指定了REVERSE，那么在每次迭代后步长值会被减除而不是增加。

整数FOR循环的一些例子：

```
FOR i IN 1..10 LOOP
    -- 我在循环中将取值 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
END LOOP;
```

```
FOR i IN REVERSE 10..1 LOOP
    -- 我在循环中将取值 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
END LOOP;
```

```
FOR i IN REVERSE 10..1 BY 2 LOOP
    -- 我在循环中将取值 10, 8, 6, 4, 2
END LOOP;
```

如果下界大于上界（或者在REVERSE情况下是小于），循环体根本不会被执行。而且不会抛出任何错误。

如果一个label被附加到FOR循环，那么整数循环变量可以用一个使用那个label的限定名引用。

## 41.6.6. 通过查询结果循环

使用一种不同类型的FOR循环，你可以通过一个查询的结果进行迭代并且操纵相应的数据。语法是：

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

target是一个记录变量、行变量或者逗号分隔的标量变量列表。target被连续不断被赋予来自query的每一行，并且循环体将为每一行执行一次。下面是一个例子：

```
CREATE FUNCTION refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing all materialized views...';

    FOR mviews IN
        SELECT n.nspname AS mv_schema,
               c.relname AS mv_name,
               pg_catalog.pg_get_userbyid(c.relowner) AS owner
        FROM pg_catalog.pg_class c
```



```

LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
WHERE c.relkind = 'm'
ORDER BY 1
LOOP

-- Now "mviews" has one record with information about the materialized
view

RAISE NOTICE 'Refreshing materialized view %.% (owner: %)...',
quote_ident(mviews.mv_schema),
quote_ident(mviews.mv_name),
quote_ident(mviews.owner);
EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.mv_schema,
mviews.mv_name);
END LOOP;

RAISE NOTICE 'Done refreshing materialized views.';
RETURN 1;
END;
$$ LANGUAGE plpgsql;

```

如果循环被一个EXIT语句终止，那么在循环之后你仍然可以访问最后被赋予的行值。

在这类FOR语句中使用的query可以是任何返回行给调用者的 SQL 命令：最常见的是SELECT，但你也可以使用带有RETURNING子句的INSERT、UPDATE或DELETE。一些EXPLAIN之类的功能性命令也可以用在這裡。

PL/pgSQL变量会被查询参数替换，并且如Section 41.11.1和Section 41.11.2中详细讨论的，查询计划会被缓存以用于可能的重用。

FOR-IN-EXECUTE语句是在行上迭代的另一种方式：

```

[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
statements
END LOOP [ label ];

```

这个例子类似前面的形式，只不过源查询被指定为一个字符串表达式，在每次进入FOR循环时都会计算它并且重新规划。这允许程序员在一个预先规划好了的命令的速度和一个动态命令的灵活性之间进行选择，就像一个纯EXECUTE语句那样。在使用EXECUTE时，可以通过USING将参数值插入到动态命令中。

另一种指定要对其结果迭代的查询的方式是将其声明为一个游标。这会在Section 41.7.4中描述。

## 41.6.7. 通过数组循环

FOREACH循环很像一个FOR循环，但不是通过一个 SQL 查询返回的行进行迭代，它通过一个数组值的元素来迭代（通常，FOREACH意味着通过一个组合值表达式的部件迭代；用于通过除数组之外组合类型进行循环的变体可能会在未来被加入）。在一个数组上循环的FOREACH语句是：

```

[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
statements
END LOOP [ label ];

```

如果没有SLICE，或者如果没有指定SLICE 0，循环会通过计算expression得到的数组的个体元素进行迭代。target变量被逐一赋予每一个元素值，并且循环体会为每一个元素执行。这里是一个通过整数数组的元素循环的例子：

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
    s int8 := 0;
    x int;
BEGIN
    FOREACH x IN ARRAY $1
    LOOP
        s := s + x;
    END LOOP;
    RETURN s;
END;
$$ LANGUAGE plpgsql;
```

元素会被按照存储顺序访问，而不管数组的维度数。尽管target通常只是一个单一变量，当通过一个组合值（记录）的数组循环时，它可以是一个变量列表。在那种情况下，对每一个数组元素，变量会被从组合值的连续列赋值。

通过一个正SLICE值，FOREACH通过数组的切片而不是单一元素迭代。SLICE值必须是一个不大于数组维度数的整数常量。target变量必须是一个数组，并且它接收数组值的连续切片，其中每一个切片都有SLICE指定的维度数。这里是一个通过一维切片迭代的例子：

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
    x int[];
BEGIN
    FOREACH x SLICE 1 IN ARRAY $1
    LOOP
        RAISE NOTICE 'row = %', x;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]);

NOTICE: row = {1, 2, 3}
NOTICE: row = {4, 5, 6}
NOTICE: row = {7, 8, 9}
NOTICE: row = {10, 11, 12}
```

## 41.6.8. 俘获错误

默认情况下，PL/pgSQL函数中发生的任何错误都会中止函数和周围事务的执行。你可以使用一个带有EXCEPTION子句的BEGIN块俘获错误并且从中恢复。其语法是BEGIN块通常的语法的一个扩展：

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
```

```

    handler_statements
  [ WHEN condition [ OR condition ... ] THEN
    handler_statements
    ... ]
END;
```

如果没有发生错误，这种形式的块只是简单地执行所有statements，并且接着控制转到END之后的下一个语句。但是如果在statements内发生了一个错误，则会放弃对statements的进一步处理，然后控制会转到EXCEPTION列表。系统会在列表中寻找匹配所发生错误的第一个condition。如果找到一个匹配，则执行对应的handler\_statements，并且接着把控制转到END之后的下一个语句。如果没有找到匹配，该错误就会传播出去，就好像根本没有EXCEPTION一样：错误可以被一个带有EXCEPTION的闭合块捕捉，如果没有EXCEPTION则中止该函数的处理。

condition的名字可以是Appendix C中显示的任何名字。一个分类名匹配其中所有的错误。特殊的条件名OTHERS匹配除了QUERY\_CANCELED和ASSERT\_FAILURE之外的所有错误类型（虽然通常并不明智，还是可以用名字捕获这两种错误类型）。条件名是大小写无关的。一个错误条件也可以通过SQLSTATE代码指定，例如以下是等价的：

```

WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

如果在选中的handler\_statements内发生了新的错误，那么它不能被这个EXCEPTION子句捕获，而是被传播出去。一个外层的EXCEPTION子句可以捕获它。

当一个错误被EXCEPTION捕获时，PL/pgSQL函数的局部变量会保持错误发生时的值，但是该块中所有对持久数据库状态的改变都会被回滚。例如，考虑这个片段：

```

INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
  UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
  x := x + 1;
  y := x / 0;
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'caught division_by_zero';
    RETURN x;
END;
```

当控制到达对y赋值的地方时，它会带着一个division\_by\_zero错误失败。这个错误将被EXCEPTION子句捕获。而在RETURN语句中返回的值将是x增加过后的值。但是UPDATE命令的效果将已经被回滚。不过，在该块之前的INSERT将不会被回滚，因此最终的结果是数据库包含Tom Jones但不包含Joe Jones。

### Tip

进入和退出一个包含EXCEPTION子句的块要比不包含EXCEPTION的块开销大的多。因此，只在必要的时候使用EXCEPTION。

#### Example 41.2. UPDATE/INSERT的异常

这个例子使用异常处理来酌情执行UPDATE或INSERT。我们推荐应用使用带有ON CONFLICT DO UPDATE的INSERT而不是真正使用这种模式。下面的例子主要是为了展示PL/pgSQL如何控制流程：

```

CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- 首先尝试更新
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- 不在这里，那么尝试插入该键
        -- 如果其他某人并发地插入同一个键，
        -- 我们可能得到一个唯一键失败
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- 什么也不做，并且循环再次尝试 UPDATE
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');

```

这段代码假定unique\_violation错误是INSERT造成，并且不是由该表上一个触发器函数中的INSERT导致。如果在该表上有多于一个唯一索引，也可能发生不正确的行为，因为不管哪个索引导致该错误它都将重试该操作。通过接下来要讨论的特性来检查被捕获的错误是否为所预期的会更安全。

#### 41.6.8.1. 得到有关一个错误的信息

异常处理器经常被用来标识发生的特定错误。有两种方法来得到PL/pgSQL中当前异常的信息：特殊变量和GET STACKED DIAGNOSTICS命令。

在一个异常处理器内，特殊变量SQLSTATE包含了对应于被抛出异常的错误代码（可能的错误代码列表见表 C.1）。特殊变量SQLERRM包含与该异常相关的错误消息。这些变量在异常处理器外是未定义的。

在一个异常处理器内，我们也可以使用GET STACKED DIAGNOSTICS命令检索有关当前异常的信息，该命令的形式为：

```
GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];
```

每个item是一个关键词，它标识一个被赋予给指定变量（应该具有接收该值的正确数据类型）的状态值。Table 41.2显示了当前可用的状态项。

Table 41.2. 错误诊断项

名称	类型	描述
RETURNED_SQLSTATE	text	该异常的 SQLSTATE 错误代码
COLUMN_NAME	text	与异常相关的列名
CONSTRAINT_NAME	text	与异常相关的约束名

名称	类型	描述
PG_DATATYPE_NAME	text	与异常相关的数据类型名
MESSAGE_TEXT	text	该异常的主要消息的文本
TABLE_NAME	text	与异常相关的表名
SCHEMA_NAME	text	与异常相关的模式名
PG_EXCEPTION_DETAIL	text	该异常的详细消息文本（如果有）
PG_EXCEPTION_HINT	text	该异常的提示消息文本（如果有）
PG_EXCEPTION_CONTEXT	text	描述产生异常时调用栈的文本行（见Section 41.6.9

如果异常没有为一个项设置值，将返回一个空字符串。

这里是一个例子：

```

DECLARE
    text_var1 text;
    text_var2 text;
    text_var3 text;
BEGIN
    -- 某些可能导致异常的处理
    ...
EXCEPTION WHEN OTHERS THEN
    GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                           text_var2 = PG_EXCEPTION_DETAIL,
                           text_var3 = PG_EXCEPTION_HINT;
END;
```

## 41.6.9. 获得执行位置信息

GET DIAGNOSTICS（之前在Section 41.5.5中描述）命令检索有关当前执行状态的信息（反之上文讨论的GET STACKED DIAGNOSTICS命令会把有关执行状态的信息报告成一个以前的错误）。它的PG\_CONTEXT状态项可用于标识当前执行位置。状态项PG\_CONTEXT将返回一个文本字符串，其中有描述该调用栈的多行文本。第一行会指向当前函数以及当前正在执行GET DIAGNOSTICS的命令。第二行及其后的行表示调用栈中更上层的调用函数。例如：

```

CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
BEGIN
    RETURN inner_func();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
    stack text;
BEGIN
    GET DIAGNOSTICS stack = PG_CONTEXT;
    RAISE NOTICE E'--- Call Stack ---\n%', stack;
    RETURN 1;
END;
$$ LANGUAGE plpgsql;

SELECT outer_func();
```

```

NOTICE: --- Call Stack ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT: PL/pgSQL function outer_func() line 3 at RETURN
outer_func
-----
          1
(1 row)

```

GET STACKED DIAGNOSTICS ... PG\_EXCEPTION\_CONTEXT返回同类的栈跟踪，但是它描述检测到错误的位置而不是当前位置。

## 41.7. 游标

和一次执行整个查询不同，可以建立一个游标来封装该查询，并且接着一次读取该查询结果的一些行。这样做的原因之一是在结果中包含大量行时避免内存不足（不过，PL/pgSQL用户通常不需要担心这些，因为FOR循环在内部会自动使用一个游标来避免内存问题）。一种更有趣的用法是返回一个函数已经创建的游标的引用，允许调用者读取行。这提供了一种有效的方法从函数中返回大型行集。

### 41.7.1. 声明游标变量

所有在PL/pgSQL中对游标的访问都会通过游标变量，它总是特殊的数据类型refcursor。创建游标变量的一种方法是把它声明为一个类型为refcursor的变量。另外一种方法是使用游标声明语法，通常是：

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

（为了对Oracle的兼容性，可以用IS替代FOR）。如果指定了SCROLL，那么游标可以反向滚动；如果指定了NO SCROLL，那么反向取的动作会被拒绝；如果二者都没有被指定，那么能否进行反向取就取决于查询。如果指定了arguments，那么它是一个逗号分隔的name datatype对的列表，它们定义在给定查询中要被参数值替换的名称。实际用于替换这些名字的值将在游标被打开之后指定。

一些例子：

```

DECLARE
  curs1 refcursor;
  curs2 CURSOR FOR SELECT * FROM tenk1;
  curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;

```

所有这三个变量都是refcursor类型，但是第一个可以用于任何查询，而第二个已经被绑定了一个完全指定的查询，并且最后一个被绑定了一个参数化查询。（游标被打开时，key将被一个整数参数值替换）。变量curs1被称为未绑定，因为它没有被绑定到任何特定查询。

如果游标查询FOR UPDATE/SHARE使用，则该SCROLL选项不可用。此外，最好用于涉及可变函数的查询NO SCROLL。SCROLL的实现假定重新加载查询的输出将返回一致的结果，而易失性函数并非如此。

### 41.7.2. 打开游标

在一个游标可以被用来检索行之前，它必需先被打开（这是和SQL命令DECLARE CURSOR等效的操作）。PL/pgSQL有三种形式的OPEN命令，其中两种用于未绑定游标变量，另外一种用于已绑定的游标变量。

## Note

可以通过Section 41.7.4中描述的FOR语句在不显式打开游标的情况下使用已绑定的游标变量。

## 41.7.2.1. OPEN FOR query

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

该游标变量被打开并且被给定要执行的查询。游标不能是已经打开的，并且它必需已经被声明为一个未绑定的游标变量（也就是声明为一个简单的refcursor变量）。该查询必须是一条SELECT或者其它返回行的东西（例如EXPLAIN）。该查询会按照其它PL/pgSQL中的SQL命令同等的方式对待：先代换PL/pgSQL变量名，并且执行计划会被缓存用于可能的重用。当一个PL/pgSQL变量被替换到游标查询中时，替换的值是在OPEN时它所具有的值。对该变量后续的改变不会影响游标的行为。对于一个已经绑定的游标，SCROLL和NO SCROLL选项具有相同的含义。

一个例子：

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

## 41.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
    [ USING expression [, ... ] ];
```

打开游标变量并且执行指定的查询。该游标不能是已打开的，并且必须已经被声明为一个未绑定的游标变量（也就是声明为一个简单的refcursor变量）。该查询以和EXECUTE中相同的方式被指定为一个字符串表达式。照例，这提供了灵活性，因此查询计划可以在两次运行之间变化（见Section 41.11.2，并且它也意味着在该命令字符串上还没有完成变量替换。正如EXECUTE，可以通过format()和USING将参数值插入到动态命令中。SCROLL和NO SCROLL选项具有和已绑定游标相同的含义。

一个例子：

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1', tablename) USING
    keyvalue;
```

在这个例子中，表名被通过format()插入到查询中。col1的比较值被通过一个USING参数插入，所以它不需要引用。

## 41.7.2.3. 打开一个已绑定的游标

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ... ] ) ];
```

这种形式的OPEN被用于打开一个游标变量，它的查询是在声明时绑定的。该游标不能是已经打开的。当且仅当该游标被声明为接收参数时，才必需出现一个实际参数值表达式的列表。这些值将被替换到命令中。

一个已绑定游标的查询计划总是被认为是可缓存的，在这种情况下没有EXECUTE的等效形式。注意SCROLL和NO SCROLL不能在OPEN中指定，因为游标的滚动行为已经被确定。

使用位置或命名记号可以传递参数值。在位置记号中，所有参数都必须按照顺序指定。在命名记号中，每一个参数的名字被使用:=指定以将它和参数表达式分隔开。类似于Section 4.8中描述的调用函数，也允许混合位置和命名记号。

例子（这些例子使用上面例子中的游标声明）：

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

因为在一个已绑定游标的查询上已经完成了变量替换，实际有两种方式将值传到游标中：给OPEN一个显式参数，或者在查询中隐式引用一个PL/pgSQL变量。不过，只有在已绑定游标之前声明的变量才将会被替换到游标中。在两种情况下，要被传递的值都是在OPEN时确定的。例如，得到上例中curs3相同效果的另一种方式是

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

### 41.7.3. 使用游标

一旦一个游标已经被打开，那么就可以用这里描述的语句操作它。

这些操作不需要发生在打开该游标开始操作的同一个函数中。你可以从一个函数返回一个refcursor值，并且让调用者在该游标上操作（在内部，refcursor值只是一个包含该游标活动查询的所谓入口的字符串名称。这个名字可以被传递、赋予给其它refcursor变量等等，而不用担心扰乱入口）。

所有入口会在事务的结尾被隐式地关闭。因此一个refcursor值只能在该事务结束前用于引用一个打开的游标。

#### 41.7.3.1. FETCH

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

就像SELECT INTO一样，FETCH从游标中检索下一行到目标中，目标可以是一个行变量、记录变量或者逗号分隔的简单变量列表。如果没有下一行，目标会被设置为 NULL。与SELECT INTO一样，可以检查特殊变量FOUND来看一行是否被获得。

direction子句可以是 SQL FETCH命令中允许的任何变体，除了那些能够取得多于一行的。即它可以是 NEXT、 PRIOR、 FIRST、 LAST、 ABSOLUTE count、 RELATIVE count、 FORWARD或者 BACKWARD。省略direction和指定NEXT是一样的。在使用count的形式中，count可以是任意的整数值表达式（与SQL命令FETCH不一样，FETCH仅允许整数常量）。除非游标被使用SCROLL选项声明或打开，否则要求反向移动的direction值很可能会失败。

cursor必须是一个引用已打开游标入口的refcursor变量名。

例子：

```
FETCH curs1 INTO rowvar;
```



```
FETCH curs2 INTO foo, bar, baz;  
FETCH LAST FROM curs3 INTO x, y;  
FETCH RELATIVE -2 FROM curs4 INTO x;
```

### 41.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

MOVE重新定位一个游标而不检索任何数据。MOVE的工作方式与FETCH命令很相似，但是MOVE只是重新定位游标并且不返回至移动到的行。与SELECT INTO一样，可以检查特殊变量FOUND来看要移动到的行是否存在。

例子：

```
MOVE curs1;  
MOVE LAST FROM curs3;  
MOVE RELATIVE -2 FROM curs4;  
MOVE FORWARD 2 FROM curs4;
```

### 41.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;  
DELETE FROM table WHERE CURRENT OF cursor;
```

当一个游标被定位到一个表行上时，使用该游标标识该行就可以对它进行更新或删除。对于游标的查询可以是什么是有限制的（尤其是不能有分组），并且最好在游标中使用FOR UPDATE。详见DECLARE参考页。

一个例子：

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

### 41.7.3.4. CLOSE

```
CLOSE cursor;
```

CLOSE关闭一个已打开游标的底层入口。这样就可以在事务结束之前释放资源，或者释放掉该游标变量以便再次打开。

一个例子：

```
CLOSE curs1;
```

### 41.7.3.5. 返回游标

PL/pgSQL函数可以向调用者返回游标。这对于返回多行或多列（特别是巨大的结果集）很有用。要想这么做，该函数打开游标并且把该游标的名字返回给调用者（或者简单的使用调用者指定的或已知的入口名打开游标）。调用者接着可以从游标中取得行。游标可以由调用者关闭，或者是在事务关闭时自行关闭。

用于一个游标的入口名可以由编程者指定或者自动生成。要指定一个入口名，只需要在打开refcursor变量之前简单地为其赋予一个字符串。OPEN将把refcursor变量的字符串值用作

底层入口的名字。不过，如果`refcursor`变量为空，`OPEN`会自动生成一个与任何现有入口不冲突的名称，并且将它赋予给`refcursor`变量。

### Note

一个已绑定的游标变量被初始化为表示其名称的字符串值，因此入口的名字和游标变量名相同，除非程序员在打开游标之前通过赋值覆盖了这个名字。但是一个未绑定的游标变量最初默认为空值，因此它会收到一个自动生成的唯一名字，除非被覆盖。

下面的例子显示了一个调用者提供游标名字的方法：

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funcursor');
FETCH ALL IN funcursor;
COMMIT;
```

下面的例子使用了自动游标名生成：

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- 需要在一个事务中使用游标。
BEGIN;
SELECT reffunc2();
```

```

    reffunc2
-----
<unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

下面的例子展示了从一个函数中返回多个游标的一种方法：

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
```

```

    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

```

-- 需要在一个事务中使用游标。  
BEGIN;

```
SELECT * FROM myfunc('a', 'b');
```

```

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;

```

## 41.7.4. 通过一个游标的结果循环

有一种FOR语句的变体，它允许通过游标返回的行进行迭代。语法是：

```

[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value
  [, ...] ) ] LOOP
    statements
END LOOP [ label ];

```

该游标变量必须在声明时已经被绑定到某个查询，并且它不能已经被打开。FOR语句会自动打开游标，并且在退出循环时自动关闭游标。当且仅当游标被声明要使用参数时，才必须出现一个实际参数值表达式的列表。这些值会被替换到查询中，采用OPEN期间的方式（见Section 41.7.2.3）。

变量recordvar会被自动定义为record类型，并且只存在于循环内部（循环中该变量名任何已有定义都会被忽略）。每一个由游标返回的行都会被陆续地赋值给这个记录变量并且执行循环体。

## 41.8. 事务管理

在由CALL命令调用的过程中以及匿名代码块（DO命令）中，可以用命令COMMIT和ROLLBACK结束事务。在一个事务被使用这些命令结束后，一个新的事务会被自动开始，因此没有单独的START TRANSACTION命令（注意BEGIN和END在PL/pgSQL中有不同的含义）。

这里是一个简单的例子：

```

CREATE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END;

```

```
$$;

CALL transaction_test1();
```

新事务开始时具有默认事务特征，如事务隔离级别。在循环中提交事务的情况下，可能需要以与前一个事务相同的特征来自动启动新事务。命令COMMIT AND CHAIN和ROLLBACK AND CHAIN可以完成此操作。

只有在从顶层调用的CALL或DO中才能进行事务控制，在没有任何其他中间命令的嵌套CALL或DO调用中也能进行事务控制。例如，如果调用栈是CALL proc1() → CALL proc2() → CALL proc3()，那么第二个和第三个过程可以执行事务控制动作。但是如果调用栈是CALL proc1() → SELECT func2() → CALL proc3()，则最后一个过程不能做事务控制，因为中间有SELECT。

对于游标循环有特殊的考虑。看看这个例子：

```
CREATE PROCEDURE transaction_test2()
LANGUAGE plpgsql
AS $$
DECLARE
    r RECORD;
BEGIN
    FOR r IN SELECT * FROM test2 ORDER BY x LOOP
        INSERT INTO test1 (a) VALUES (r.x);
        COMMIT;
    END LOOP;
END;
$$;

CALL transaction_test2();
```

通常，游标会在事务提交时被自动关闭。但是，一个作为循环的组成部分创建的游标会自动被第一个COMMIT或ROLLBACK转变成一个可保持游标。这意味着该游标在第一个COMMIT或ROLLBACK处会被完全计算出来，而不是逐行被计算。该游标在循环后仍会被自动删除，因此这通常对用户是不可见的。

有非只读命令（UPDATE ... RETURNING）驱动的游标循环中不允许有事务命令。

事务在一个具有异常处理部分的块中不能被结束。

## 41.9. 错误和消息

### 41.9.1. 报告错误和消息

使用RAISE语句报告消息以及抛出错误。

```
RAISE [ level ] 'format' [, expression [, ... ] ] [ USING option = expression
[, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ] ;
RAISE ;
```

level选项指定了错误的严重性。允许的级别有DEBUG、LOG、INFO、NOTICE、WARNING以及EXCEPTION，默认级别是EXCEPTION。EXCEPTION会抛出一个错误（通常会中止当前事

务)。其他级别仅仅是产生不同优先级的消息。不管一个特定优先级的消息是被报告给客户端、还是写到服务器日志、亦或是二者同时都做，这都由log\_min\_messages和client\_min\_messages配置变量控制。详见Chapter 17

如果有level，在它后面可以写一个format（它必须是一个简单字符串而不是表达式）。该格式字符串指定要被报告的错误消息文本。在格式字符串后面可以跟上可选的要被插入到该消息的参数表达式。在格式字符串中，%会被下一个可选参数的值所替换。写%%可以发出一个字面的%。参数的数量必须匹配格式字符串中%占位符的数量，否则在函数编译期间就会发生错误。

在这个例子中，v\_job\_id的值将替换字符串中的%:

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

通过写一个后面跟着option = expression项的USING，可以为错误报告附加一些额外信息。每一个expression可以是任意字符串值的表达式。允许的option关键词是:

MESSAGE

设置错误消息文本。这个选项可以被用于在USING之前包括一个格式字符串的RAISE形式。

DETAIL

提供一个错误的细节消息。

HINT

提供一个提示消息。

ERRCODE

指定要报告的错误代码（SQLSTATE），可以用Appendix C中所示的条件名，或者直接作为一个五字符的SQLSTATE代码。

COLUMN

CONSTRAINT

DATATYPE

TABLE

SCHEMA

提供一个相关对象的名称。

这个例子将用给定的错误消息和提示中止事务:

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
  USING HINT = 'Please check your user ID';
```

这两个例子展示了设置SQLSTATE的两种等价的方法:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

还有第二种RAISE语法，在其中主要参数是要被报告的条件名或SQLSTATE，例如:

```
RAISE division_by_zero;
```

```
RAISE SQLSTATE '22012';
```

在这种语法中，USING能被用来提供一个自定义的错误消息、细节或提示。另一种做前面的例子的方法是

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

仍有另一种变体是写RAISE USING或者RAISE level USING并且把所有其他东西都放在USING列表中。

RAISE的最后一种变体根本没有参数。这种形式只能被用在一个BEGIN块的EXCEPTION子句中，它导致当前正在被处理的错误被重新抛出。

### Note

在LightDB 9.1 之前，没有参数的RAISE被解释为重新抛出来自包含活动异常处理器的块的错误。因此一个嵌套在那个处理器中的EXCEPTION子句无法捕捉它，即使RAISE位于嵌套EXCEPTION子句的块中也是这样。这种行为很奇怪，也并不兼容 Oracle 的 PL/SQL。

如果在一个RAISE EXCEPTION命令中没有指定条件名以及 SQLSTATE，默认是使用ERRCODE\_RAISE\_EXCEPTION (P0001)。如果没有指定消息文本，默认是使用条件名或SQLSTATE 作为消息文本。

### Note

当用 SQLSTATE 代码指定一个错误代码时，你不会受到预定义错误代码的限制，而是可以选择任何由五位以及大写 ASCII 字母构成的错误代码，只有00000不能使用。我们推荐尽量避免抛出以三个零结尾的错误代码，因为这些是分类代码并且只能用来捕获整个类别。

## 41.9.2. 检查断言

ASSERT语句是一种向 PL/pgSQL函数中插入调试检查的方便方法。

```
ASSERT condition [ , message ];
```

condition是一个布尔 表达式，它被期望总是计算为真。如果确实如此， ASSERT语句不会再做什么。但如果结果是假 或者空，那么将发生一个ASSERT\_FAILURE异常（如果在计算condition时发生错误，它会被报告为一个普通错误）。

如果提供了可选的message， 它是一个结果（如果非空）被用来替换默认错误消息文本“assertion failed”的表达式（如果 condition失败）。 message表达式在 断言成功的普通情况下不会被计算。

通过配置参数plpgsql.check\_asserts可以启用或者禁用断言测试， 这个参数接受布尔值且默认为on。如果这个参数为off， 则ASSERT语句什么也不做。

注意ASSERT是为了检测程序的 bug，而不是 报告普通的错误情况。如果要报告普通错误，请使用前面介绍的 RAISE语句。

## 41.10. 触发器函数

PL/pgSQL可以被用来在数据更改或者数据库事件上定义触发器函数。触发器函数用CREATE FUNCTION命令创建，它被声明为一个没有参数并且返回类型为trigger（对于数据更改触发器）或者event\_trigger（对于数据库事件触发器）的函数。名为PG\_something的特殊局部变量将被自动创建用以描述触发该调用的条件。

## 41.10.1. 数据改变的触发器

一个数据更改触发器被声明为一个没有参数并且返回类型为trigger的函数。注意，如下所述，即便该函数准备接收一些在CREATE TRIGGER中指定的参数 — 这类参数通过TG\_ARGV传递，也必须把它声明为没有参数。

当一个PL/pgSQL函数当做触发器调用时，在顶层块会自动创建一些特殊变量。它们是：

NEW

数据类型是RECORD；该变量为行级触发器中的INSERT/UPDATE操作保持新数据行。在语句级别的触发器以及DELETE操作，这个变量是null。

OLD

数据类型是RECORD；该变量为行级触发器中的UPDATE/DELETE操作保持新数据行。在语句级别的触发器以及INSERT操作，这个变量是null。

TG\_NAME

数据类型是name；该变量包含实际触发的触发器名。

TG\_WHEN

数据类型是text；是值为BEFORE、AFTER或INSTEAD OF的一个字符串，取决于触发器的定义。

TG\_LEVEL

数据类型是text；是值为ROW或STATEMENT的一个字符串，取决于触发器的定义。

TG\_OP

数据类型是text；是值为INSERT、UPDATE、DELETE或TRUNCATE的一个字符串，它说明触发器是为哪个操作引发。

TG\_RELID

数据类型是oid；是导致触发器调用的表的对象 ID。

TG\_RELNAME

数据类型是name；是导致触发器调用的表的名称。现在已经被废弃，并且可能在未来的一个发行中消失。使用TG\_TABLE\_NAME替代。

TG\_TABLE\_NAME

数据类型是name；是导致触发器调用的表的名称。

TG\_TABLE\_SCHEMA

数据类型是name；是导致触发器调用的表所在的模式名。

TG\_NARGS

数据类型是integer；在CREATE TRIGGER语句中给触发器函数的参数数量。

TG\_ARGV[]

数据类型是text数组；来自CREATE TRIGGER语句的参数。索引从 0 开始记数。非法索引（小于 0 或者大于等于tg\_nargs）会导致返回一个空值。

一个触发器函数必须返回NULL或者是一个与触发器为之引发的表结构完全相同的记录/行值。

BEFORE引发的行级触发器可以返回一个空来告诉触发器管理器跳过对该行剩下的操作（即后续的触发器将不再被引发，并且不会对该行发生INSERT/UPDATE/DELETE）。如果返回了一个非空值，那么对该行值会继续操作。返回不同于原始NEW的行值将修改将要被插入或更新的行。因此，如果该触发器函数想要触发动作正常成功而不修改行值，NEW（或者另一个相等的值）必须被返回。要修改将被存储的行，可以直接在NEW中替换单一值并且返回修改后的NEW，或者构建一个全新的记录/行来返回。在一个DELETE上的前触发器情况下，返回值没有直接效果，但是它必须为非空以允许触发器动作继续下去。注意NEW在DELETE触发器中是空值，因此返回它通常没有意义。在DELETE中的常用方法是返回OLD。

INSTEAD OF触发器（总是行级触发器，并且可能只被用于视图）能够返回空来表示它们没有执行任何更新，并且对该行剩余的操作可以被跳过（即后续的触发器不会被引发，并且该行不会被计入外围INSERT/UPDATE/DELETE的行影响状态中）。否则一个非空值应该被返回用以表示该触发器执行了所请求的操作。对于INSERT 和UPDATE操作，返回值应该是NEW，触发器函数可能对它进行了修改来支持INSERT RETURNING和UPDATE RETURNING（这也将影响被传递给任何后续触发器的行值，或者被传递给带有ON CONFLICT DO UPDATE的INSERT语句中一个特殊的EXCLUDED别名引用）。对于DELETE操作，返回值应该是OLD。

一个AFTER行级触发器或一个BEFORE或AFTER语句级触发器的返回值总是会被忽略，它可能也是空。不过，任何这些类型的触发器可能仍会通过抛出一个错误来中止整个操作。

Example 41. 展示了PL/pgSQL中一个触发器函数的例子。

### Example 41.3. 一个 PL/pgSQL 触发器函数

这个例子触发器保证：任何时候一个行在表中被插入或更新时，当前用户名和时间也会被标记在该行中。并且它会检查给出了一个雇员的姓名以及薪水是一个正值。

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
BEGIN
    -- 检查给出了 empname 以及 salary
    IF NEW.empname IS NULL THEN
        RAISE EXCEPTION 'empname cannot be null';
    END IF;
    IF NEW.salary IS NULL THEN
        RAISE EXCEPTION '% cannot have null salary', NEW.empname;
    END IF;

    -- 谁会倒贴钱为我们工作?
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
    END IF;
```



```

-- 记住谁在什么时候改变了工资单
NEW.last_date := current_timestamp;
NEW.last_user := current_user;
RETURN NEW;
END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW EXECUTE FUNCTION emp_stamp();

```

另一种记录对表的改变的方法涉及到创建一个新表来为每一个发生的插入、更新或删除保持一行。这种方法可以被认为是对一个表的改变的审计。Example 41. 展示了PL/pgSQL中一个审计触发器函数的例子。

#### Example 41.4. 一个用于审计的 PL/pgSQL 触发器函数

这个例子触发器保证了在emp表上的任何插入、更新或删除一行的动作都被记录（即审计）在emp\_audit表中。当前时间和用户名会被记录到行中，还有在其上执行的操作类型。

```

CREATE TABLE emp (
    empname      text NOT NULL,
    salary       integer
);

CREATE TABLE emp_audit(
    operation     char(1)  NOT NULL,
    stamp        timestamp NOT NULL,
    userid       text     NOT NULL,
    empname      text     NOT NULL,
    salary       integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --
    -- 在 emp_audit 中创建一行来反映 emp 上执行的动作，
    -- 使用特殊变量 TG_OP 来得到操作。
    --
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
    END IF;
    RETURN NULL; -- 因为这是一个 AFTER 触发器，结果被忽略
END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW EXECUTE FUNCTION process_emp_audit();

```

前一个例子的一种变体使用一个视图将主表连接到审计表来展示每一项最后被修改是什么时候。这种方法还是记录了对于表修改的完整审查跟踪，但是也提供了审查跟踪的一个简化视图，只为每一个项显示从审查跟踪生成的最后修改时间戳。Example 41. 展示了在PL/pgSQL中一个视图上审计触发器的例子。

## Example 41.5. 一个用于审计的 PL/pgSQL 视图触发器函数

这个例子在视图上使用了一个触发器让它变得可更新，并且确保视图中一行的任何插入、更新或删除被记录（即审计）在emp\_audit表中。当前时间和用户名会被与执行的操作类型一起记录，并且该视图会显示每一行的最后修改时间。

```

CREATE TABLE emp (
    empname      text PRIMARY KEY,
    salary       integer
);

CREATE TABLE emp_audit(
    operation    char(1) NOT NULL,
    userid      text NOT NULL,
    empname     text NOT NULL,
    salary      integer,
    stamp       timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
    FROM emp e
    LEFT JOIN emp_audit ea ON ea.empname = e.empname
    GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
BEGIN
    --
    -- 执行 emp 上所要求的操作，并且在 emp_audit 中创建一行来反映对 emp 的改
    变。
    --
    IF (TG_OP = 'DELETE') THEN
        DELETE FROM emp WHERE empname = OLD.empname;
        IF NOT FOUND THEN RETURN NULL; END IF;

        OLD.last_updated = now();
        INSERT INTO emp_audit VALUES('D', user, OLD.*);
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
        IF NOT FOUND THEN RETURN NULL; END IF;

        NEW.last_updated = now();
        INSERT INTO emp_audit VALUES('U', user, NEW.*);
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp VALUES(NEW.empname, NEW.salary);

        NEW.last_updated = now();
        INSERT INTO emp_audit VALUES('I', user, NEW.*);
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;

```

```
CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
FOR EACH ROW EXECUTE FUNCTION update_emp_view();
```

触发器的一种用法是维护一个表的另一个汇总表。作为结果的汇总表可以用来在特定查询中替代原始表 — 通常会大量减少运行时间。这种技术常用于数据仓库中，在其中被度量或被观察数据的表（称为事实表）可能会极度大。Example 41.6展示了PL/pgSQL中一个为数据仓库事实表维护汇总表的触发器函数的例子。

#### Example 41.6. 一个 PL/pgSQL 用于维护汇总表的触发器函数

这里详述的模式有一部分是基于 Ralph Kimball 所作的The Data Warehouse Toolkit中的Grocery Store例子。

```
--
-- 主表 - 时间维度和销售事实。
--
CREATE TABLE time_dimension (
    time_key          integer NOT NULL,
    day_of_week       integer NOT NULL,
    day_of_month      integer NOT NULL,
    month             integer NOT NULL,
    quarter           integer NOT NULL,
    year              integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key          integer NOT NULL,
    product_key       integer NOT NULL,
    store_key         integer NOT NULL,
    amount_sold       numeric(12,2) NOT NULL,
    units_sold        integer NOT NULL,
    amount_cost       numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- 汇总表 - 按时间汇总销售
--
CREATE TABLE sales_summary_bytime (
    time_key          integer NOT NULL,
    amount_sold       numeric(15,2) NOT NULL,
    units_sold        numeric(12) NOT NULL,
    amount_cost       numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- 在 UPDATE、INSERT、DELETE 时修改汇总列的函数和触发器。
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
```

```
BEGIN

-- 算出增量/减量数。
IF (TG_OP = 'DELETE') THEN

    delta_time_key = OLD.time_key;
    delta_amount_sold = -1 * OLD.amount_sold;
    delta_units_sold = -1 * OLD.units_sold;
    delta_amount_cost = -1 * OLD.amount_cost;

ELSIF (TG_OP = 'UPDATE') THEN

    -- 禁止更改 the time_key 的更新-
    -- (可能不会太麻烦, 因为大部分的更改是用 DELETE + INSERT 完成的)。
    IF ( OLD.time_key != NEW.time_key) THEN
        RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
            OLD.time_key,
NEW.time_key;
    END IF;

    delta_time_key = OLD.time_key;
    delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
    delta_units_sold = NEW.units_sold - OLD.units_sold;
    delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

ELSIF (TG_OP = 'INSERT') THEN

    delta_time_key = NEW.time_key;
    delta_amount_sold = NEW.amount_sold;
    delta_units_sold = NEW.units_sold;
    delta_amount_cost = NEW.amount_cost;

END IF;

-- 插入或更新带有新值的汇总行。
<<insert_update>>
LOOP
    UPDATE sales_summary_bytime
        SET amount_sold = amount_sold + delta_amount_sold,
            units_sold = units_sold + delta_units_sold,
            amount_cost = amount_cost + delta_amount_cost
        WHERE time_key = delta_time_key;

    EXIT insert_update WHEN found;

BEGIN
    INSERT INTO sales_summary_bytime (
        time_key,
        amount_sold,
        units_sold,
        amount_cost)
        VALUES (
            delta_time_key,
            delta_amount_sold,
            delta_units_sold,
            delta_amount_cost
        );
```

```

        EXIT insert_update;

    EXCEPTION
        WHEN UNIQUE_VIOLATION THEN
            -- 什么也不做
        END;
    END LOOP insert_update;

    RETURN NULL;

END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE FUNCTION maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1, 1, 1, 10, 3, 15);
INSERT INTO sales_fact VALUES(1, 2, 1, 20, 5, 35);
INSERT INTO sales_fact VALUES(2, 2, 1, 40, 15, 135);
INSERT INTO sales_fact VALUES(2, 3, 1, 10, 1, 13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;

```

AFTER也可以利用传递表来观察被触发语句更改的整个行集合。CREATE TRIGGER命令会为一个或者两个传递表分配名字，然后函数可以引用那些名字，就好像它们是只读的临时表一样。Example 41. 展示了一个例子。

#### Example 41.7. 用传递表进行审计

这个例子产生和Example 41. 相同的结果，但并未使用一个为每一行都触发的触发器，而是在把相关信息收集到一个传递表中之后用了一个只为每个语句引发一次的触发器。当调用语句修改了很多行时，这种方法明显比行触发器方法快。注意我们必须为每一种事件建立一个单独的触发器声明，因为每种情况的REFERENCING子句必须不同。但是这并不能阻止我们使用单一的触发器函数（实际上，使用三个单独的函数会更好，因为可以避免在TG\_OP上的运行时测试）。

```

CREATE TABLE emp (
    empname      text NOT NULL,
    salary       integer
);

CREATE TABLE emp_audit(
    operation    char(1) NOT NULL,
    stamp       timestamp NOT NULL,
    userid      text NOT NULL,
    empname     text NOT NULL,
    salary      integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --

```

```

-- 在emp_audit中创建行来反映在emp上执行的操作,
-- 利用特殊变量TG_OP来区分操作。
--
IF (TG_OP = 'DELETE') THEN
    INSERT INTO emp_audit
        SELECT 'D', now(), user, o.* FROM old_table o;
ELSIF (TG_OP = 'UPDATE') THEN
    INSERT INTO emp_audit
        SELECT 'U', now(), user, n.* FROM new_table n;
ELSIF (TG_OP = 'INSERT') THEN
    INSERT INTO emp_audit
        SELECT 'I', now(), user, n.* FROM new_table n;
END IF;
RETURN NULL; -- 由于这是一个AFTER触发器, 所以结果被忽略
END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit_ins
    AFTER INSERT ON emp
    REFERENCING NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_upd
    AFTER UPDATE ON emp
    REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_del
    AFTER DELETE ON emp
    REFERENCING OLD TABLE AS old_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();

```

## 41.10.2. 事件触发器

PL/pgSQL可以被用来定义事件触发器。LightDB要求一个可以作为事件触发器调用的函数必须被声明为没有参数并且返回类型为event\_trigger。

当一个PL/pgSQL函数被作为一个事件触发器调用, 在顶层块中会自动创建一些特殊变量。它们是:

TG\_EVENT

数据类型是text; 它是一个表示引发触发器的事件的字符串。

TG\_TAG

数据类型是text; 它是一个变量, 包含了该触发器为之引发的命令标签。

Example 41.7展示了PL/pgSQL中一个事件触发器函数的例子。

Example 41.8. 一个 PL/pgSQL 事件触发器函数

这个例子触发器在受支持命令每一次被执行时会简单地抛出一个NOTICE消息。

```

CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE FUNCTION snitch();

```

## 41.11. PL/pgSQL的内部

这一节讨论了一些PL/pgSQL用户应该知道的一些重要的实现细节。

### 41.11.1. 变量替换

一个PL/pgSQL函数中的 SQL 语句和表达式能够引用该函数的变量和参数。在现象背后，PL/pgSQL会为这些引用替换查询参数。只有在语法上允许的地方才会替换参数。作为一种极端情况，考虑这个编程风格糟糕的例子：

```
INSERT INTO foo (foo) VALUES (foo);
```

foo的第一次出现在语法上必须是一个表名，因此它将不会被替换，即使该函数有一个名为foo的变量。第二次出现必须是该表的一列的名称，因此它也将不会被替换。同样，第三位应该是一个函数名。因此它也不被替换。只有最后一个位置是PL/pgSQL函数中变量引用的候选者。

#### Note

LightDB 9.0 之前的版本会尝试在所有三种情况下替换变量，导致语法错误。

因为变量名在语法上与表列的名字没什么区别，在也引用表的语句中会有歧义：一个给定的名字意味着一个表列或一个变量？让我们把前一个例子改成：

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

这里，dest和src必须是表名，并且col必须是dest的一列，但是foo和bar可能该函数的变量或者src的列。

默认情况下，如果一个 SQL 语句中的名称可能引用一个变量或者一个表列，PL/pgSQL将报告一个错误。修复这种问题的方法很多：你可以重命名变量或列来，或者对有歧义的引用加以限定，或者告诉PL/pgSQL要引用哪种解释。

最简单的解决方案是重命名变量或列。一种常用的编码规则是为PL/pgSQL变量使用一种不同于列名的命名习惯。例如，如果你将函数变量统一地命名为v\_something，而你的列名不会开始于v\_，就不会发生冲突。

另外你可以限定有歧义的引用让它们变清晰。在上面的例子中，src.foo将是对表列的一种无歧义的引用。要创建对一个变量的无歧义引用，在一个被标记的块中声明它并且使用块的标签（见Section 41.2）。例如

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

这里block.foo表示变量，即使在src中有一个列foo。函数参数以及诸如FOUND的特殊变量，都能通过函数的名称被限定，因为它们被隐式地声明在一个带有该函数名称的外层块中。

有时候在一个大型的PL/pgSQL代码体中修复所有的有歧义引用是不现实的。在这种情况下，你可以指定PL/pgSQL应该将有歧义的引用作为变量（这与PL/pgSQL在 LightDB 9.0 之前的行为兼容）或表列（这与某些其他系统兼容，例如Oracle）解决。

要在系统范围内改变这种行为，将配置参数`plpgsql.variable_conflict`设置为`error`、`use_variable`或者`use_column`（这里`error`是出厂设置）之一。这个参数会影响PL/pgSQL函数中语句的后续编译，但是不会影响在当前会话中已经编译过的语句。因为改变这个设置能够导致PL/pgSQL函数中行为的意想不到的改变，所以只能由一个超级用户来更改它。

你也可以对逐个函数设置该行为，做法是在函数文本的开始插入这些特殊命令之一：

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

这些命令只影响它们所属的函数，并且会覆盖`plpgsql.variable_conflict`的设置。一个例子是：

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
            WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

在UPDATE命令中，`curtime`、`comment`以及`id`将引用该函数的变量和参数，不管`users`有没有这些名称的列。注意，我们不得不在WHERE子句中对`users.id`的引用加以限定，以便让它引用表列。但是我们不需要在UPDATE列表中把对`comment`的引用限定为一个目标，因为语法上那必须是`users`的一列。我们可以用下面的方式写一个相同的不依赖于`variable_conflict`设置的函数：

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment =
            stamp_user.comment
            WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

被交给EXECUTE或其变体的命令字符串中不会发生变量替换。如果你需要插入一个变化值到这样一个命令中，在构建该字符串值时就这样做，或者使用USING，如Section 41.5.4中所阐明的。

当前变量替换只能在SELECT、INSERT、UPDATE和DELETE还有包含下列之一的命令（比如EXPLAIN和CREATE TABLE ... AS SELECT）中有效，因为主SQL引擎只允许查询参数在这些命令中。要在其他语句类型（通常被称为实用语句）中使用一个非常量名称或值，你必须将实用语句构建为一个字符串并且EXECUTE它。

## 41.11.2. 计划缓存

在函数被第一次调用时（在每个会话中），PL/pgSQL解释器解析函数的源文本并且产生一个内部的二进制指令树。该指令树完全翻译了PL/pgSQL语句结构，但是该函数中使用的SQL表达式以及SQL命令并没有被立即翻译。



作为该函数中每一个表达式和第一次被执行的SQL命令，PL/pgSQL解释器使用SPI管理器的SPI\_prepare函数解析并且分析该命令来创建一个预备语句。对于那个表达式或命令的后续访问将会重用该预备语句。因此，一个带有很少被访问的条件性代码路径的函数将永远不会发生分析那些在当前会话中永远不被执行的命令的开销。一个缺点是在一个特定表达式或命令中的错误将不能被检测到，直到函数的该部分在执行时被到达（不重要的语法错误在初始的解析中就会被检测到，但是任何更深层次的东西将只有在执行时才能检测到）。

PL/pgSQL（或者更准确地说是 SPI 管理器）能进一步尝试缓冲与任何特定预备语句相关的执行计划。如果没有使用一个已缓存的计划，那么每次访问该语句时都会生成一个新的执行计划，并且当前的参数值（也就是PL/pgSQL的变量值）可以被用来优化被选中的计划。如果该语句没有参数，或者要被执行很多次，SPI 管理器将考虑创建一个不依赖特定参数值的一般计划并且将其缓存用于重用。通常只有在执行计划对其中引用的PL/pgSQL变量值不那么敏感时，才会这样做。如果这样做，每一次生成的计划就是纯利。关于预备语句的行为请详见PREPARE。

由于PL/pgSQL保存预备语句并且有时候以这种方式保存执行计划，直接出现在一个PL/pgSQL函数中的 SQL 命令必须在每次执行时引用相同的表和列。也就是说，你不能在一个 SQL 命令中把一个参数用作表或列的名字。要绕过这种限制，你可以构建PL/pgSQL EXECUTE使用的动态命令，但是会付出在每次执行时需要执行新解析分析以及构建新执行计划的代价。

记录变量的易变天性在这个关系中带来了另一个问题。当一个记录变量的域被用在表达式或语句中时，域的数据类型不能在该函数的调用之间改变，因为每一个表达式被分析时都将使用第一次到达该表达式时存在的数据类型。必要时，可以用EXECUTE来绕过这个问题。

如果同一个函数被用作一个服务于多个表的触发器，PL/pgSQL会为每一个这样的表独立地准备并缓存语句——也就是对每一种触发器函数和表的组合都会有一个缓存，而不是每个函数一个缓存。这减轻了数据类型变化带来的问题。例如，一个触发器函数将能够成功地使用一个名为key的列工作，即使该列正好在不同的表中有不同的类型。

同样，具有多态参数类型的函数也会为它们已经被调用的每一种实参类型组合都保留一个独立的缓存，这样数据类型差异不会导致意想不到的失败。

语句缓存有时可能在解释时间敏感的值时产生令人惊讶的效果。例如这两个函数做的事情就有区别：

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
BEGIN
    INSERT INTO logtable VALUES (logtxt, 'now');
END;
$$ LANGUAGE plpgsql;
```

以及：

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
DECLARE
    curtime timestamp;
BEGIN
    curtime := 'now';
    INSERT INTO logtable VALUES (logtxt, curtime);
END;
$$ LANGUAGE plpgsql;
```

在logfunc1中，LightDB的主解析器在分析INSERT时就知道字符串'now'应该被解释为timestamp，因为logtable的目标列是这种类型。因此，在INSERT被分析时'now'将被转换为一个timestamp常量，并且在该会话的生命周期内被用于所有对logfunc1的调用。不用说，这不是程序员想要的。一个更好的主意是使用now()或current\_timestamp函数。

在logfunc2中，LightDB的主解析器不知道'now'应该变成什么类型并且因此返回一个text类型的包含字符串now的数据值。在确定对本地变量curtime的赋值期间，PL/pgSQL解释器通过

调用用于转换的`textout`以及`timestamp_in`函数将这个字符串造型为`timestamp`类型。因此，计算出来的时间戳会按照程序员的期待在每次执行时更新。虽然这正好符合预期，但是它的效率很糟糕，因此使用`now()`函数仍然是一种更好的方案。

## 41.12. PL/pgSQL开发提示

在PL/pgSQL中进行开发的一种好方法是使用你自己选择的文本编辑器来创建函数，并且在另一个窗口中使用`ltsql`来载入并且测试那些函数。如果你正在这样做，使用`CREATE OR REPLACE FUNCTION`来编写函数是一个好主意。用那种方式你只需要重载该文件来更新函数的定义。例如：

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
    ....
$$ LANGUAGE plpgsql;
```

在运行`ltsql`期间，你可以用下面的命令载入或者重载这样一个函数定义文件：

```
\i filename.sql
```

并且接着立即发出 SQL 命令来测试该函数。

另一种在PL/pgSQL中开发的方式是用一个 GUI 数据库访问工具，它能方便对过程语言的开发。这种工具的一个例子是`pgAdmin`。这些工具通常提供方便的特性，例如转义单引号以及便于重新创建和调试函数。

### 41.12.1. 处理引号

一个PL/pgSQL函数的代码在一个`CREATE FUNCTION`中被指定为一个字符串。如果你用通常的方式把该字符串写在单引号中间，那么该函数体中的任何单引号都必须被双写；同样任何反斜线也必须被双写（假定使用了转义字符串语法）。双写引号最多有点冗长，并且在更复杂的情况下代码会变得完全无法理解，因为你很容易发现你需要半打或者更多相邻的引号。我们推荐你转而把函数体写成一个“美元引用”的字符串（见Section 4.1.2.4）。在美元引用方法中，你从不需要双写任何引号。但是要注意为你需要的每一层嵌套选择一个不同的美元引用定界符。例如，你可能把`CREATE FUNCTION`命令写成：

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
    ....
$PROC$ LANGUAGE plpgsql;
```

在这里面，你可以在 SQL 命令中为简单字符串使用引号并且用`$$`来界定被你组装成字符串的 SQL 命令片段。如果你需要引用包括`$$`的文本，你可以使用`Q$$`等等。

下列图表展示了在写没有美元引用的引号时需要做什么。在将之前用美元引用的代码翻译成更容易理解的代码时，它们会有所帮助。

#### 1 个引号

用来开始和结束函数体，例如：

```
CREATE FUNCTION foo() RETURNS integer AS '
    ....
' LANGUAGE plpgsql;
```

在一个单引号引用的函数体中的任何位置，引号必须成对出现。



```
if v_... like '...' then return '...'; end if;
```

在美元引用方法中，这会变成：

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
|| referrer_keys.key_string || $$'
then return '$$ || referrer_keys.referrer_type
|| $$'; end if;$$;
```

这里我们假定我们只需要把单引号放在a\_output中，因为在使用前它将被再引用。

## 41.12.2. 额外的编译时和运行时检查

为了辅助用户在一些简单但常见的问题产生危害之前找到它们，PL/pgSQL提供了额外的检查。当被启用时，根据配置，它们可以在一个函数的编译期间被用来发出WARNING或者ERROR。一个已经收到了WARNING的函数可以被继续执行而不会产生进一步的消息，因此建议你在一个单独的开发环境中进行测试。

根据需要设置 plpgsql.extra\_warnings 或 plpgsql.extra\_errors，适当情况下，在开发和/或测试环境中可以设置为“all”。

这些附加检查通过配置变量启用，plpgsql.extra\_warnings用于警告，plpgsql.extra\_errors用于错误。两者都可以设置为逗号分隔的检查列表，“none”或“all”。默认值为“none”。当前可用的检查列表包括：

shadowed\_variables

检查声明是否遮盖了以前定义的变量

strict\_multi\_assignment

某些PL/PgSQL命令允许一次将值分配给多个变量，例如SELECT INTO。通常，目标变量的数量和源变量的数量应匹配，尽管PL/PgSQL将使用NULL来处理缺失的值和被忽略的额外变量。启用此检查将导致PL/PgSQL在目标变量数和源变量数不同时引发WARNING或ERROR。

too\_many\_rows

启用此检查将导致PL/PgSQL检查在使用INTO子句时给定查询是否返回多行。由于INTO语句只会使用一行，让查询返回多行通常会效率低下和/或不确定性，因此很可能出现错误。

下面的示例显示了plpgsql.extra\_warnings 设置为shadowed\_variables的效果：

```
SET plpgsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
DECLARE
f1 int;
BEGIN
RETURN f1;
END;
$$ LANGUAGE plpgsql;
WARNING: variable "f1" shadows a previously defined variable
LINE 3: f1 int;
      ^
CREATE FUNCTION
```

下面的示例显示了将plpgsql.extra\_warnings 设置为strict\_multi\_assignment:

```
SET plpgsql.extra_warnings TO 'strict_multi_assignment';

CREATE OR REPLACE FUNCTION public.foo()
  RETURNS void
  LANGUAGE plpgsql
AS $$
DECLARE
  x int;
  y int;
BEGIN
  SELECT 1 INTO x, y;
  SELECT 1, 2 INTO x, y;
  SELECT 1, 2, 3 INTO x, y;
END;
$$;

SELECT foo();
WARNING: number of source and target fields in assignment does not match
DETAIL: strict_multi_assignment check of extra_warnings is active.
HINT: Make sure the query returns the exact list of columns.
WARNING: number of source and target fields in assignment does not match
DETAIL: strict_multi_assignment check of extra_warnings is active.
HINT: Make sure the query returns the exact list of columns.

foo
----

(1 row)
```

## 41.13. 从Oracle PL/SQL 移植

这一节解释了LightDB的PL/pgSQL语言和 Oracle 的PL/SQL语言之间的差别，用以帮助那些从Oracle®向LightDB移植应用的人。

PL/pgSQL与 PL/SQL 在许多方面都非常类似。它是一种块结构的、命令式的语言并且所有变量必须先被声明。赋值、循环和条件则很类似。在从PL/SQL向PL/pgSQL移植时必须记住一些事情:

- 如果一个 SQL 命令中使用的名字可能是一个表的列名或者是对一个函数中变量的引用，那么PL/SQL会将它当作一个列名。默认情况下，PL/pgSQL会抛出名称模糊的错误，你可以指定plpgsql.variable\_conflict=use\_column来改变这一行为以匹配PL/SQL。如Section 41.11. 中所述。通常最好是首先避免这种歧义，但如果不得不移植依赖于该行为的大量代码，那么设置variable\_conflict将是最好的方案。
- 在LightDB中，函数体必须写成字符串文本。因此你需要使用美元符引用或者转义函数体中的单引号（见Section 41.12. 1）。
- 数据类型名称常常需要翻译。例如，在 Oracle 中字符串值通常被声明为类型varchar2，这并非 SQL 标准类型。在LightDB中则要使用类型varchar或者text来替代。类似地，要把类型number替换成numeric，或者在适当的时候使用某种其他数字数据类型。
- 应该用模式把函数组织成不同的分组，而不是用包。
- 因为没有包，所以也没有包级别的变量。这一点有时候挺讨厌。你可以在临时表里保存会话级别的状态。

- 带有REVERSE的整数FOR循环的工作方式不同：PL/SQL中是从第二个数向第一个数倒数，而PL/pgSQL是从第一个数向第二个数倒数，因此在移植时需要交换循环边界。不幸的是这种不兼容性是不太可能改变的（见Section 41.6.5.5）。
- 查询上的FOR循环（不是游标）的工作方式同样不同：目标变量必须已经被声明，而PL/SQL总是会隐式地声明它们。但是这样做的优点是在退出循环后，变量值仍然可以访问。
- 在使用游标变量方面，存在一些记法差异。

自从 LightDB 22.2 版本开始，LightDB 支持 Oracle PL/SQL 过程，用户可以直接从 Oracle 数据库导出 PL/SQL 代码，然后在 LightDB 数据库中重新执行。请参阅 Chapter 42

### 41.13.1. 移植示例

Example 41.8 展示了如何从PL/SQL移植一个简单的函数到PL/pgSQL中。

Example 41.9. 从PL/SQL移植一个简单的函数到PL/pgSQL

这里有一个Oracle PL/SQL函数：

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar2,
                                                  v_version varchar2)
RETURN varchar2 IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

让我们过一遍这个函数并且看看与PL/pgSQL相比有什么样的不同：

- 类型名称varchar2被改成了varchar或者text。在这一节的例子中，我们将使用varchar，但如果不需要特定的字符串长度限制，text常常是更好的选择。
- 在函数原型中（不是函数体中）的RETURN关键字在LightDB中变成了RETURNS。还有，IS变成了AS，并且你还需要增加一个LANGUAGE子句，因为PL/pgSQL并非唯一可用的函数语言。
- 在LightDB中，函数体被认为是一个字符串，所以你需要使用引号或者美元符号包围它。这代替了Oracle 方法中的用于终止的/。
- 在LightDB中没有show errors命令，并且也不需要这个命令，因为错误是自动报告的。

这个函数被移植到LightDB后看起来会是这样：

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Example 41.10展示了如何移植一个会创建另一个函数的函数，以及如何处理引号问题。

Example 41.10. 从PL/SQL移植一个创建另一个函数的函数到PL/pgSQL

下面的过程从一个SELECT语句抓取行，并且为了效率而构建一个带有IF语句中结果的大型函数。

这是 Oracle 版本：

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN
    VARCHAR2,
                                v_domain IN VARCHAR2, v_ur1 IN VARCHAR2) RETURN VARCHAR2 IS
    BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
            ' IF v_' || referrer_key.kind
            || ' LIKE ''' || referrer_key.key_string
            || ''' THEN RETURN ''' || referrer_key.referrer_type
            || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

这里是LightDB的版本：

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc() AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
            ' IF v_' || referrer_key.kind
            || ' LIKE ' || quote_literal(referrer_key.key_string)
            || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
            || '; END IF;';
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';
```

```

func_cmd :=
  'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                    v_domain varchar,
                                                    v_url varchar)

  RETURNS varchar AS '
  || quote_literal(func_body)
  || ' LANGUAGE plpgsql;' ;

EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;

```

请注意函数体是如何被单独构建并且通过`quote_literal`被传递以双写其中的任何引号。需要这个技术是因为无法安全地使用美元引用定义新函数：我们不确定从`referrer_key.key_string`域中来的什么字符串会被插入（我们这里假定`referrer_key.kind`可以确信总是为`host`、`domain`或者`url`，但是`referrer_key.key_string`可能是任何东西，特别是它可能包含美元符号）。这个函数实际上是在 Oracle 的原版上的改进，因为当`referrer_key.key_string`或者`referrer_key.referrer_type`包含引号时，它将不会生成坏掉的代码。

Example 41.1展示了如何移植一个带有OUT参数和字符串处理的函数。LightDB没有内建的`instr`函数，但是你可以用其它函数的组合来创建一个。在Section 41.13. 中有一个`instr`的PL/pgSQL实现，你可以用它让你的移植变得更容易。

Example 41.11. 从PL/SQL移植一个带有字符串操作以及OUT参数的过程到PL/pgSQL

下面的Oracle PL/SQL 过程被用来解析一个 URL 并且返回一些元素（主机、路径和查询）。

这是 Oracle 版本：

```

CREATE OR REPLACE PROCEDURE cs_parse_url(
  v_url IN VARCHAR2,
  v_host OUT VARCHAR2, -- 这将被传回去
  v_path OUT VARCHAR2, -- 这个也是
  v_query OUT VARCHAR2) -- 还有这个
IS
  a_pos1 INTEGER;
  a_pos2 INTEGER;
BEGIN
  v_host := NULL;
  v_path := NULL;
  v_query := NULL;
  a_pos1 := instr(v_url, '/');

  IF a_pos1 = 0 THEN
    RETURN;
  END IF;
  a_pos2 := instr(v_url, '/', a_pos1 + 2);
  IF a_pos2 = 0 THEN
    v_host := substr(v_url, a_pos1 + 2);
    v_path := '/';
    RETURN;
  END IF;

  v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
  a_pos1 := instr(v_url, '?', a_pos2 + 1);

```



```

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;

```

这里是一种到PL/pgSQL的可能翻译:

```

CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR, -- 这将被传递回去
    v_path OUT VARCHAR, -- 这个也是
    v_query OUT VARCHAR) -- 以及这个
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;

```

这个函数可以这样使用:

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Example 41.1展示了如何移植一个使用了多种 Oracle 特性的过程。

## Example 41.12. 从PL/SQL移植一个过程到PL/pgSQL

Oracle 版本:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS
    NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- 释放锁
        raise_application_error(-20000,
            'Unable to create a new job: a job is currently running.');
```

END IF;

```
DELETE FROM cs_active_job;
INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
EXCEPTION
    WHEN dup_val_on_index THEN NULL; -- 如果已经存在也不用担心
END;
COMMIT;
END;
/
show errors
```

这是我们如何将这个过程移植到PL/pgSQL:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id integer) AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS
    NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- 释放锁
        RAISE EXCEPTION 'Unable to create a new job: a job is currently
running'; -- ❶
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN -- ❷
            -- 如果已经存在不用担心
    END;
```

```

COMMIT;
END;
$$ LANGUAGE plpgsql;

```

- ❶ RAISE的语法与 Oracle 的语句相当不同，尽管基本的形式RAISE exception\_name工作起来是相似的。
- ❷ PL/pgSQL所支持的异常名称不同于 Oracle。内建的异常名称集合要更大（见Appendix C）。目前没有办法声明用户定义的异常名称，尽管你能够抛出用户选择的 SQLSTATE 值。

## 41.13.2. 其他要关注的事项

这一节解释了在移植 Oracle PL/SQL函数到LightDB中时要关注的一些其他问题。

### 41.13.2.1. 异常后隐式回滚

在PL/pgSQL，当一个异常被EXCEPTION子句捕获之后，从该块的BEGIN以来的所有数据库改变都会被自动回滚。也就是，该行为等效于你在 Oracle 中用下面的代码得到的效果：

```

BEGIN
    SAVEPOINT s1;
    ... 代码 ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... 代码 ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... 代码 ...
END;

```

如果你正在翻译一个使用这种风格的SAVEPOINT以及ROLLBACK TO的 Oracle 过程，你的工作比较简单：只要忽略掉SAVEPOINT以及ROLLBACK TO。如果你的 Oracle 过程是以不同的方法使用SAVEPOINT以及ROLLBACK TO，那么就要真正地动一番脑筋了。

### 41.13.2.2. EXECUTE

PL/pgSQL的EXECUTE与PL/SQL中的工作相似，但是必须要记住按照Section 41.5.4中所述地使用quote\_literal以及quote\_ident。EXECUTE 'SELECT \* FROM \$1';类型的结构将无法可靠地工作除非你使用这些函数。

### 41.13.2.3. 优化 PL/pgSQL 函数

LightDB提供了两种函数创建修饰符来优化执行：“volatility”（对于给定的相同参数，函数是否总是返回相同的结果）以及“strictness”（如果任何参数为空，函数是否返回空）。详见CREATE FUNCTION参考页。

在利用这些优化属性时，你的CREATE FUNCTION语句应该看起来像这样：

```

CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

```

## 41.13.3. 附录

这一节包含了一组 Oracle 兼容的instr函数代码，你可以用它来简化你的移植工作。

```
--  
-- instr 函数模仿 Oracle 的对应函数  
-- 语法: instr(string1, string2 [, n [, m]])  
-- 其中 [] 表示可选参数。  
--  
-- 从第n个字符开始搜索string1, 要求找到string2的第m次出现。  
-- 如果n为负, 则从后向前搜索, 从string1的末尾开始的第abs(n)个字符开始。  
-- 如果没有传n, 假定它为1 (从第1个字符开始搜索)。  
-- 如果没有传m, 假定它为1 (找第1次出现)。  
-- 在string1中返回string2的开始索引, 如果没有找到string2则为0。  
--
```

```
CREATE FUNCTION instr(vchar, vchar) RETURNS integer AS $$  
BEGIN  
    RETURN instr($1, $2, 1);  
END;  
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

```
CREATE FUNCTION instr(string vchar, string_to_search_for vchar,  
                      beg_index integer)  
RETURNS integer AS $$  
DECLARE  
    pos integer NOT NULL DEFAULT 0;  
    temp_str vchar;  
    beg integer;  
    length integer;  
    ss_length integer;  
BEGIN  
    IF beg_index > 0 THEN  
        temp_str := substring(string FROM beg_index);  
        pos := position(string_to_search_for IN temp_str);  
  
        IF pos = 0 THEN  
            RETURN 0;  
        ELSE  
            RETURN pos + beg_index - 1;  
        END IF;  
    ELSIF beg_index < 0 THEN  
        ss_length := char_length(string_to_search_for);  
        length := char_length(string);  
        beg := length + 1 + beg_index;  
  
        WHILE beg > 0 LOOP  
            temp_str := substring(string FROM beg FOR ss_length);  
            IF string_to_search_for = temp_str THEN  
                RETURN beg;  
            END IF;  
  
            beg := beg - 1;  
        END LOOP;  
  
        RETURN 0;  
    ELSE  
        RETURN 0;  
    END IF;
```

```
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                    beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF occur_index <= 0 THEN
        RAISE 'argument ''%'' is out of range', occur_index
        USING ERRCODE = '22003';
    END IF;

    IF beg_index > 0 THEN
        beg := beg_index - 1;
        FOR i IN 1..occur_index LOOP
            temp_str := substring(string FROM beg + 1);
            pos := position(string_to_search_for IN temp_str);
            IF pos = 0 THEN
                RETURN 0;
            END IF;
            beg := beg + pos;
        END LOOP;

        RETURN beg;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                occur_number := occur_number + 1;
                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;
        END IF;

        beg := beg - 1;
    END LOOP;

    RETURN 0;
ELSE
    RETURN 0;
END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

---

# Chapter 42. PL/oraSQL — SQL过程化语言

## 42.1. 概述

PL/oraSQL 是一个可加载的过程化语言，用于 LightDB 数据库系统。它用于与 Oracle PL/SQL 应用程序兼容。PL/oraSQL 的设计目标是创建一个可加载的过程化语言，该语言具有以下特性：

- 可用于创建函数、过程和触发器，
- 添加控制结构到 SQL 语言，
- 可执行复杂计算，
- 继承所有用户定义的类型、函数、过程和操作符，
- 可定义为服务器信任的对象，
- 容易使用。

使用 PL/oraSQL 创建的函数可以在任何内置函数可用的地方使用。例如，可以创建复杂的条件计算函数，然后将它们用于定义操作符或在索引表达式中使用。

在 LightDB 22.2 及更高版本中，默认安装了 PL/oraSQL。但是，它仍然是一个可加载的模块，因此特别是注重安全的管理员可以选择删除它。

尽管大多数功能默认情况下都已启用，但某些功能（例如调用特定于 Oracle 的函数或 Oracle 内置包）也需要 LightDB 在 Oracle 模式下运行。

### 42.1.1. 使用 PL/oraSQL 的优势

SQL 是 LightDB 和大多数其他关系型数据库所使用的查询语言。它是可移植且易学习的。但是，每个 SQL 语句必须由数据库服务器单独执行。

这意味着您的客户端应用程序必须将每个查询发送到数据库服务器，等待其被处理，接收和处理结果，执行某些计算，然后发送进一步的查询到服务器。所有这些都产生进程间通信，并且如果您的客户端位于与数据库服务器不同的计算机上，还会产生网络开销。

使用 PL/oraSQL，您可以将一组计算和一系列查询放在数据库服务器内部，从而具备了过程化语言的功能以及 SQL 的易用性，但是可以大大节省客户端/服务器通信开销。

- 兼容 Oracle 的 PL/SQL 存储过程语言
- 客户端和服务器之间的额外往返被消除
- 客户端不需要的中间结果不必在服务器和客户端之间进行编组或传输
- 可以避免多次查询解析

与不使用存储函数的应用程序相比，这可以显著提高性能。

此外，使用 PL/oraSQL，您可以使用 SQL 的所有数据类型、操作符和函数。

### 42.1.2. 支持的参数和结果数据类型

使用 PL/oraSQL 编写的函数可以接受服务器支持的任何标量或数组数据类型作为参数，并且它们可以返回任何这些类型的结果。它们还可以通过名称指定接受或返回任何复合类型（行类型）。还可以将 PL/oraSQL 函数声明为接受 record，这意味着任何复合类型都将作为

输入，或者声明为返回 `record`，这意味着结果是一个行类型，其列由调用查询中的规范确定。

## 42.2. PL/oraSQL 的结构

PL/oraSQL 是一种块结构语言。函数、过程、包、触发器或匿名块都包含一个块结构，我们称所有包含在匿名块中的部分为块结构。

### 42.2.1. 匿名块

定义匿名块如下：

```
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
    Exception-handling part ]
END;
/
```

块内每个声明和语句都以分号结尾。出现在另一个块内的块必须在 `END` 后面加上分号，如上所示；然而，结束函数体的最终 `END` 也需要一个分号，其中 `/` 是 `plorasql` 低层块结构的结束字符。

#### Tip

一个常见的错误是在 `BEGIN` 后立即写一个分号。这是不正确的，会导致语法错误。匿名块目前不支持使用 `<<label>>`。

例如，使用匿名块输出“quantity”的值：

```
DECLARE
    quantity integer := 80;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Quantity here is '||quantity); -- Prints 80
END;
/
```

在使用 `DBMS_OUTPUT.PUT_LINE` 函数之前，需要打开打印权限。

```
select dbms_output.serveroutput(true);
```

当然，此函数只能在 Oracle 兼容模式下使用。

从 LightDB 23.4 版本开始，匿名块通过在 `libpq` 使用时支持绑定变量。需要注意的是，使用绑定参数时客户端需要传递参数类型。绑定参数的数量限制为65535。

下面是通过 `libpq` 执行使用绑定参数的匿名块的用例：

```
/*
 * src/test/examples/lt_testlibpq.c
 *
 *
```

```
* lt_testlibpq.c
* this test program shows to use LIBPQ to exec dostmt with bind param
*
*/
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn      *conn;
    PGresult     *res;
    const char *paramValues[10];
    Oid paramTypes[10];
    int  nFields;
    int  i,
        j;

    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    conn = PQconnectdb(conninfo);

    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "create table test_dostmt(key1 int, key2 text);");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    PQclear(res);

    res = PQexec(conn, "insert into test_dostmt values(1,'a');");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    }
}
```



```

        PQclear(res);
        exit_nicely(conn);
    }

    PQclear(res);

    paramTypes[0] = 23;
    paramValues[0] = "1";

    printf("update test_dostmt key2 to 'test_dostmt'\n");
    res = PQexecParams(conn, "begin\
        begin\
            update test_dostmt set key2 = 'test_dostmt' where key1
= $1 ;\
        end;\
    end;", 1, paramTypes, paramValues, NULL, NULL, 0);

    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    PQclear(res);

    printf("get test_dostmt key2\n");
    res = PQexec(conn, "select key2 from test_dostmt where key1=1;");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    nFields = PQnfields(res);

    /* print out the instances */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%s: %s", PQfname(res, j), PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    printf("update test_dostmt key2 to 'test_dostmt_new'\n");
    paramTypes[0] = 23;
    paramValues[0] = "1";
    paramTypes[1] = 25;
    paramValues[1] = "test_dostmt1";
    res = PQexecParams(conn, "\
        declare\
            id int := $1; \
            val text := $2; \
        begin\

```

```
        begin\  
            val = val || '_new'; \  
            update test_dostmt set key2 = val where key1 = id ;\  
        end;\  
end;”, 2, paramTypes, paramValues, NULL, NULL, 0);  
  
if (PQresultStatus(res) != PGRES_COMMAND_OK)  
{  
    fprintf(stderr, “SET failed: %s”, PQerrorMessage(conn));  
    PQclear(res);  
    exit_nicely(conn);  
}  
  
PQclear(res);  
  
printf(“get test_dostmt key2 \n”);  
res = PQexec(conn, “select key2 from test_dostmt where key1=1;”);  
if (PQresultStatus(res) != PGRES_TUPLES_OK)  
{  
    fprintf(stderr, “SET failed: %s”, PQerrorMessage(conn));  
    PQclear(res);  
    exit_nicely(conn);  
}  
  
nFields = PQnfields(res);  
/* print out the instances */  
for (i = 0; i < PQntuples(res); i++)  
{  
    for (j = 0; j < nFields; j++)  
        printf(“%s: %s”, PQfname(res, j), PQgetvalue(res, i, j));  
    printf(“\n”);  
}  
  
PQclear(res);  
  
res = PQexec(conn, “drop table test_dostmt;”);  
if (PQresultStatus(res) != PGRES_COMMAND_OK)  
{  
    fprintf(stderr, “SET failed: %s”, PQerrorMessage(conn));  
    PQclear(res);  
    exit_nicely(conn);  
}  
  
PQclear(res);  
  
PQfinish(conn);  
  
return 0;  
}
```

## 42.2.2. 注释

PL/oraSQL 编译器会忽略注释。它们的目的是帮助其他应用程序开发人员理解您的源文本。通常，您使用注释来描述每个代码段的目的和用途。您还可以通过将它们变成注释来禁用过时或未完成的代码片段。

### 42.2.2.1. 单行注释

单行注释以 `--` 开始，延伸到行末。这个例子有三个单行注释。

```
DECLARE
  howmany    NUMBER;
  num_tables NUMBER;
BEGIN
  -- Begin processing
  SELECT COUNT(*) INTO howmany
  FROM USER_OBJECTS
  WHERE OBJECT_TYPE = 'TABLE'; -- Check number of tables
  num_tables := howmany;      -- Compute another value
END;
/
```

### 42.2.2.2. 多行注释

多行注释以 `/*` 开始，以 `*/` 结束，可以跨多行。

```
BEGIN
  /*
  IF 2 + 2 = 4 THEN
    some_condition := TRUE;
  END IF;
  */
  NULL;
END;
/
```

## 42.3. 声明

在块中使用的所有变量都必须在块的声明部分中声明。（唯一的例外是，迭代整数值范围的 FOR 循环的循环变量自动声明为整数变量，类似地，迭代游标结果的 FOR 循环的循环变量自动声明为记录变量。）

PL/oraSQL 变量可以具有任何 SQL 数据类型，例如 `integer`、`varchar` 和 `char`。

这里是一些变量声明的示例：

```
user_id integer;
quantity numeric(5);
url varchar(64);
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

变量声明的一般语法如下：

```
name [ CONSTANT ] type [ NOT NULL ] { DEFAULT | := } expression;
```

如果给出了 `DEFAULT` 子句，则指定在输入块时分配给变量的初始值。如果未给出 `DEFAULT` 子句，则会将变量初始化为 SQL 空值。`CONSTANT` 选项防止在初始化后对变量进行分配，因此其值将在块的持续时间内保持不变。如果指定了 `NOT NULL`，则将空值分配给它会导致运行时错误。所有声明为 `NOT NULL` 的变量都必须指定一个非空默认值。

示例:

```
quantity integer DEFAULT 32;
url varchar(64) := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

### 42.3.1. 声明函数参数

声明的函数参数名的数据类型可以是 SQL 参数类型或 PLoraSQL 参数类型。并且您可以使用 in、 out 和 inout来修改参数。例如:

```
CREATE FUNCTION sales_tax(subtotal in real) RETURN real IS
BEGIN
    RETURN subtotal * 0.06;
END;
/
```

如果匿名块需要函数返回值时, 使用inout修饰参数, return返回值必须要有具体的返回值, 不然返回的就是一个record。例如:

```
create table nested_tab(id int, name varchar2(100), job varchar2(100), age int);
insert into nested_tab values (2, 'sdfs', 'cvxx', 14);
```

```
create or replace function nested_func(id1 inout int, name1 inout varchar2(100),
    job1 inout varchar2(100), age1 inout int)
return int as
    id2 int;
    name2 varchar2(100);
    job2 varchar2(100);
    age2 int;
begin
    select * into id2, name2, job1, age1 from nested_tab where age = 14;
    id1 := id2;
    name1 := name2;
    job1 := job2;
    age1 := age2;
    begin
        id1 := 45;
        name1 := 'name';
        job1 := 'job';
        age1 := 45;
    end;
    return 0;
end;
/
```

```
create or replace function nested_func12(id1 inout int, name1 inout
    varchar2(100), job1 inout varchar2(100), age1 inout int)
return int as
    retcode int;
begin
    begin
        retcode := nested_func(id1, name1, job1, age1);
    end;
    return 1;
end;
```

```

end;
/
declare
  id2 int;
  name2 varchar2(100);
  job2  varchar2(100);
  age2  int;
  retcode int;
begin
  begin
  retcode := nested_func12(id2, name2, job2, age2);
  end;
  dbms_output.put_line(id2 || name2 || job2 || age2);
end;
/

```

### 42.3.2. 复制类型

```
variable%TYPE
```

`%TYPE` 提供变量或表列的数据类型。您可以使用它来声明将保存数据库值的变量。例如，假设在您的 `users` 表中有一个名为 `user_id` 的列。要声明一个与 `users.user_id` 具有相同数据类型的变量，您可以编写：

```
user_id users.user_id%TYPE;
```

通过使用 `%TYPE`，您不需要知道所引用的结构的数据类型，最重要的是，如果所引用的项目的数据类型在将来发生更改（例如：将 `user_id` 的类型从 `integer` 更改为 `real`），您可能不需要更改函数定义。

在多态函数中，`%TYPE` 特别有价值，因为内部变量所需的数据类型可能会从一次调用到下一次调用发生变化。可以通过将 `%TYPE` 应用于函数的参数或结果占位符来创建适当的变量。

`%TYPE` 也可以用来声明包类型变量。要声明一个数据类型与 `package.variable` 相同的变量，您可以这样写：

```
var package.variable%TYPE;
```

### 42.3.3. 行类型

```
name table_name%ROWTYPE;
```

复合类型的变量称为 **行变量**（或 **行类型变量**）。这样的变量可以保存 `SELECT` 或 `FOR` 查询结果的整行，只要该查询的列集与变量的声明类型匹配即可。访问行值的各个字段使用通常的点标记法，例如 `rowvar.field`。

可以使用 `table_name%ROWTYPE` 符号将行变量声明为具有与现有表或视图的行相同类型的变量；或者可以通过给出复合类型的名称来声明它。（由于每个表都有与其名称相同的关联复合类型，因此实际上在 `LightDB` 中无论您是否编写 `%ROWTYPE` 都无关紧要。但是带有 `%ROWTYPE` 的形式更具可移植性。）

### 42.3.4. 记录类型

```
name RECORD;
```

记录变量类似于行类型变量，但它们没有预定义的结构。它们在 `SELECT` 或 `FOR` 命令期间分配的行上采用实际的行结构。记录变量的子结构可能每次分配给它时都会发生变化。这样的结果是，直到首次分配给记录变量之前，它没有子结构，任何尝试访问其中的字段都会引发运行时错误。

请注意，`RECORD` 不是真正的数据类型，只是一个占位符。还应该意识到，当声明一个返回类型为 `record` 的函数时，这与记录变量的概念并不完全相同，尽管这样的函数可能使用记录变量来保存其结果。在两种情况下，当编写函数时，实际的行结构是未知的，但对于返回 `record` 的函数，在解析调用查询时确定实际结构，而记录变量可以在运行时更改其行结构。

## 42.4. 表达式

PL/oraSQL 语句中使用的所有表达式都使用服务器的主 SQL 执行程序进行处理。例如，当您编写一个 PL/oraSQL 语句时，例如：

```
IF expression THEN ...
```

例如，如果我们声明了两个整数变量 `x` 和 `y`，并编写以下代码：

```
IF x < y THEN ...
```

## 42.5. 基本语句

在本节和接下来的几节中，我们描述了所有由 PL/oraSQL 显式理解的语句类型。任何不被识别为这些语句类型之一的内容都被认为是 SQL 命令，并被发送到主数据库引擎执行。

### 42.5.1. 赋值

一个给 PL/oraSQL 变量赋值的语句写作如下：

```
variable := expression;
```

如之前所述，这条语句中的表达式通过发送 SQL `SELECT` 命令到主数据库引擎进行评估。表达式必须产生一个单一的值（如果变量是行或记录变量，则可能是一行值）。目标变量可以是一个简单的变量（可选地带有块名称），行或记录变量的字段，或者是一个简单变量或字段的数组元素。

示例：

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

### 42.5.2. 在 PLoraSQL 中执行 DML

PLoraSQL 在匿名块、函数、存储过程，包中支持 DML，DML 包括 `INSERT`，`UPDATE`，and `DELETE` 语句。如果需要使用 `SELECT` 语句，必须加上 `into` 子句，否则必然报错。例子如下：

```
create table dml_table(id int, name varchar2(100), job varchar2(100), age int);
insert into dml_table values (1, 'asda', 'gfdgd', 12);
insert into dml_table values (2, 'sdfsd', 'cvxvx', 14);
insert into dml_table values (3, 'uyiy', 'mmbv', 16);
```

```

create or replace function support_dml return int as
begin
update dml_table set id = 4 where name = 'uyiy';
return 12;
end;
/
select support_dml() from dual;

declare
retcode int := 1;
begin
retcode := support_dml();
dbms_output.put_line(retcode);
end;
/

```

### 42.5.3. 执行一个单行结果的查询

产生单行结果（可能是多列）的SQL命令的结果可以赋值给一个记录变量、行类型变量或标量变量列表。这可以通过编写基本SQL命令并添加一个INTO子句来完成。例如：

```

SELECT select_expressions INTO target FROM ...;
INSERT ... RETURNING expressions INTO target;
UPDATE ... RETURNING expressions INTO target;
DELETE ... RETURNING expressions INTO target;

```

其中target可以是一个记录变量，一个行变量，或者是一个逗号分隔的简单变量和记录/行字段的列表。PL/oraSQL变量将被替换到查询的剩余部分中，并且计划被缓存，就像上面描述的那样，对于不返回行的命令。这适用于带有RETURNING的SELECT、INSERT/UPDATE/DELETE以及返回行集结果的实用命令（例如EXPLAIN）。除了INTO子句之外，SQL命令与在PL/oraSQL外部编写的命令相同。

#### Tip

注意，这种带有INTO的SELECT的解释与LightDB的常规SELECT INTO命令截然不同，其中INTO目标是一个新创建的表。如果你想从PL/oraSQL函数内部的SELECT结果创建一个表，请使用语法CREATE TABLE ... AS SELECT。

如果行或变量列表被用作目标，则查询的结果列必须与目标的结构在数量和数据类型上完全匹配，否则将发生运行时错误。当记录变量是目标时，它会自动将自己配置为查询结果列的行类型。

INTO子句可以出现在SQL命令的几乎任何位置。通常，它要么在SELECT命令中的select\_expressions列表之前或之后，要么在其他命令类型的命令的末尾。建议你遵循这个惯例，以防PL/oraSQL解析器在未来版本中变得更加严格。

如果INTO子句中未指定STRICT，则target将被设置为查询返回的第一行，如果查询未返回任何行，则设置为nulls。（请注意，“第一行”未定义，除非你使用了ORDER BY。）第一行后的任何结果行都将被丢弃。您可以检查特殊的SQL%FOUND变量来确定是否返回了一行：

```

DROP TABLE t1;
DROP SEQUENCE t1_seq;

CREATE TABLE t1 (

```

```

    id NUMBER(10),
    description VARCHAR2(50),
    CONSTRAINT t1_pk PRIMARY KEY (id)
);

CREATE SEQUENCE t1_seq;

INSERT INTO t1 VALUES (t1_seq.nextval, 'ONE');
INSERT INTO t1 VALUES (t1_seq.nextval, 'TWO');
INSERT INTO t1 VALUES (t1_seq.nextval, 'THREE');

```

RETURNING INTO子句允许我们返回受DML语句影响的行的列值。返回的数据可以是单列。例如，当我们使用序列插入数据来生成我们的主键值时，我们可以如下返回主键值：

```

DECLARE
    l_id t1.id%TYPE;
BEGIN
    INSERT INTO t1 VALUES (t1_seq.nextval, 'FOUR')
    RETURNING id INTO l_id;
    COMMIT;

    DBMS_OUTPUT.put_line(' ID=' || l_id);
END;
/

```

例如，语法也适用于更新和删除语句。

```

DECLARE
    l_id t1.id%TYPE;
BEGIN
    UPDATE t1
    SET    description = description
    WHERE description = 'FOUR'
    RETURNING id INTO l_id;

    DBMS_OUTPUT.put_line(' UPDATE ID=' || l_id);

    DELETE FROM t1
    WHERE description = 'FOUR'
    RETURNING id INTO l_id;

    DBMS_OUTPUT.put_line(' DELETE ID=' || l_id);

    COMMIT;
END;
/

```

## 42.5.4. 执行动态命令

通常情况下，你会想在PL/oraSQL函数内部生成动态命令，即每次执行时都涉及不同的表或不同的数据类型。PL/oraSQL的正常尝试缓存命令计划在这种情况下无法工作。为了处理这种问题，提供了EXECUTE IMMEDIATE语句：

```
EXECUTE IMMEDIATE command-string [ INTO target ] [ USING expression [, ... ] ];
```

其中command-string是一个表达式，产生一个包含要执行的命令的字符串（类型为varchar）。可选的target是一个记录变量、行变量或由简单变量和记录/行字段组成的逗号分隔列表，用于存储命令的结果。可选的USING表达式提供要插入命令的值。



在计算的命令字符串上不执行PL/oraSQL变量的替换。任何必需的变量值必须在构建命令字符串时插入；或者你可以使用下面描述的参数。

此外，通过EXECUTE IMMEDIATE执行的命令没有计划缓存。相反，每次运行语句时，命令都会被计划。因此，命令字符串可以在函数内部动态创建，以执行对不同表和列的操作。

INTO子句指定SQL命令返回的行应分配到哪里。如果提供了行或变量列表，则它必须与查询结果的结构完全匹配（当使用记录变量时，它将自动配置自身以匹配结果结构）。如果返回多行，则仅将第一行分配给INTO变量。如果没有返回行，则将NULL分配给INTO变量。如果没有指定INTO子句，则丢弃查询结果。

如果提供了STRICT选项，除非查询产生恰好一行，否则将报告错误。

命令字符串可以使用参数值，在命令中引用为:inserted\_by、:inserted等。这些符号指的是在USING子句中提供的值。这种方法通常比将数据值作为varchar插入命令字符串更可取：它避免了将值转换为varchar并返回的运行时开销，并且由于不需要引用或转义，它更不容易受到SQL注入攻击的影响。例如：

```
EXECUTE IMMEDIATE 'SELECT count(*) FROM mytable WHERE inserted_by = :inserted_by
AND inserted <= :inserted'
  INTO c
  USING checked_user, checked_date;
```

简单常量命令字符串和一些USING参数的EXECUTE IMMEDIATE（如上面的第一个示例）在功能上等同于直接在PL/oraSQL中编写命令，并允许自动替换PL/oraSQL变量。重要的区别在于，EXECUTE IMMEDIATE将在每次执行时重新计划命令，生成一个特定于当前参数值的计划；而PL/oraSQL可能会创建一个通用计划并缓存它以供重用。在最佳计划强烈依赖于参数值的情况下，使用EXECUTE IMMEDIATE可以帮助确保不选择通用计划。

EXECUTE IMMEDIATE中目前不支持SELECT INTO；相反，执行一个简单的SELECT命令，并将INTO作为EXECUTE IMMEDIATE本身的一部分指定。

using 节点所使用的变量前面可以添加变量类型 in 和 out。in 输入的变量类型表示不可以被更改，out 输入的变量类型表示可以被更改。 例如：

```
CREATE OR REPLACE PROCEDURE TargetProcedure(
  p_input1 IN VARCHAR2,
  p_input2 IN NUMBER,
  p_output1 OUT VARCHAR2,
  p_output2 OUT NUMBER
) AS
BEGIN
  DBMS_Output.PUT_LINE('TargetProcedure begin-> p_input1 '||' is: '||
p_input1);
  DBMS_Output.PUT_LINE('TargetProcedure begin-> p_input2 '||' is: '||
p_input2);
  DBMS_Output.PUT_LINE('TargetProcedure begin-> p_output1 '||' is: '||
p_output1);
  DBMS_Output.PUT_LINE('TargetProcedure begin-> p_output2 '||' is: '||
p_output2);
  p_output1 := 'Response from TargetProcedure: ' || p_input1;
  p_output2 := p_input2 * 10;
  DBMS_Output.PUT_LINE('TargetProcedure end-> p_input1 '||' is: '||p_input1);
  DBMS_Output.PUT_LINE('TargetProcedure end-> p_input2 '||' is: '||p_input2);
  DBMS_Output.PUT_LINE('TargetProcedure end-> p_output1 '||' is: '||
p_output1);
  DBMS_Output.PUT_LINE('TargetProcedure end-> p_output2 '||' is: '||
p_output2);
```

```

END TargetProcedure;
/
CREATE OR REPLACE PROCEDURE MainProcedure(
    main_input1 IN VARCHAR2,
    main_input2 IN NUMBER,
    main_output1 OUT VARCHAR2,
    main_output2 OUT NUMBER
) AS
BEGIN
    main_output2 := 0;
    main_output1 := 'Operation successful';
    -- Use EXECUTE IMMEDIATE to execute dynamic SQL and pass parameters
    DBMS_Output.PUT_LINE('MainProcedure begin-> main_input1 '||' is: '||
main_input1);
    DBMS_Output.PUT_LINE('MainProcedure begin-> main_input2 '||' is: '||
main_input2);
    DBMS_Output.PUT_LINE('MainProcedure begin-> main_output1 '||' is: '||
main_output1);
    DBMS_Output.PUT_LINE('MainProcedure begin-> main_output2 '||' is: '||
main_output2);
    EXECUTE IMMEDIATE 'BEGIN
TargetProcedure(:p_input1, :p_input2, :p_output1, :p_output2); END;'
    USING IN main_input1, IN main_input2, OUT main_output1, OUT main_output2;
    DBMS_Output.PUT_LINE('MainProcedure end-> main_input1 '||' is: '||
main_input1);
    DBMS_Output.PUT_LINE('MainProcedure end-> main_input2 '||' is: '||
main_input2);
    DBMS_Output.PUT_LINE('MainProcedure end-> main_output1 '||' is: '||
main_output1);
    DBMS_Output.PUT_LINE('MainProcedure end-> main_output2 '||' is: '||
main_output2);
END MainProcedure;
/
call MainProcedure('xiaoming',123,'xiaohong',234);

```

### 42.5.5. 完全不做任何操作

有时候，一个不做任何事情的占位符语句是有用的。例如，它可以表示if/then/else语句中的一个分支是故意空的。为此，使用NULL语句：

```
NULL;
```

例如，以下两个代码片段是等效的：

```

declare
y int := 10;
x int := 10;
BEGIN
    NULL;
END;
/

```

## 42.6. 控制结构

控制结构可能是PL/oraSQL中最有用（也是最重要）的部分。使用PL/oraSQL的控制结构，可以以非常灵活和强大的方式操纵LightDB数据。

## 42.6.1. 从函数中返回

有一个命令可以让你从函数中返回数据：RETURN。

### 42.6.1.1. RETURN

```
RETURN expression;
```

使用带有表达式的RETURN终止函数并将表达式的值返回给调用者。这种形式用于不返回集合的PL/oraSQL函数。

如果使用输出参数声明了函数，则只需编写没有表达式的RETURN。输出参数变量的当前值将被返回。当前不支持混合使用Oracle输出参数和返回子句。

函数的返回值不能未定义。如果控制到达函数的顶层块的末尾而没有命中RETURN语句，则会发生运行时错误。在这些情况下，如果顶层块完成，则会自动执行RETURN语句。

一些例子：

```
-- 返回一个标量类型的函数
RETURN 1 + 2;
RETURN scalar_var;
```

## 42.6.2. 从过程返回

过程没有返回值。因此，过程可以在没有RETURN语句的情况下结束。如果您希望使用RETURN语句提前退出代码，则只需编写没有表达式的RETURN语句。

存储过程参数使用out进行修饰，您可以使用只有RETURN语句但没有返回值来结束存储过程。

如果存储过程具有输出参数，则输出参数变量的最终值将返回给调用方。

## 42.6.3. 调用存储过程

PL/oraSQL函数、过程或匿名块可以调用存储过程。例如：

```
CREATE PROCEDURE triple(x in int, y out int)
AS
BEGIN
    y := x * 3;
END;
/

DECLARE
    myvar int := 5;
    myresult int := 10;
BEGIN
    triple(myvar, myresult);
    DBMS_OUTPUT.PUT_LINE('myresult = ' || myresult); -- prints 15
END;
/
```

## 42.6.4. 条件语句

IF和CASE语句可让您根据某些条件执行替代命令。 PL/oraSQL有三种IF形式：

- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

和两种CASE形式:

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

#### 42.6.4.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN语句是IF语句的最简单形式。如果条件为真，则执行THEN和END IF之间的语句。否则，它们将被跳过。

示例:

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

#### 42.6.4.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

IF-THEN-ELSE语句通过让您指定一组备选语句，以便在条件不为真时执行，增强了IF-THEN语句。（请注意，这包括条件评估为NULL的情况。）

示例:

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

#### 42.6.4.3. IF-THEN-ELSIF

```

IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
]
[ ELSIF boolean-expression THEN
    statements
    ...
]
[ ELSE
    statements ]
END IF;

```

有时会有超过两个的备选方案。IF-THEN-ELSIF提供了一种方便的方法，逐个检查几个备选方案。IF条件逐个进行测试，直到找到第一个为真为止。然后执行相关的语句，之后控制传递到END IF之后的下一条语句。（任何后续的IF条件都不会被测试。）如果没有任何IF条件为真，则执行ELSE块（如果有的话）。

下面是一个例子：

```

IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- 嗯，唯一的其他可能性是number为null
    result := 'NULL';
END IF;

```

关键字ELSIF也可以拼写为ELSEIF。

完成相同任务的另一种方法是嵌套IF-THEN-ELSE语句，如下面的示例所示：

```

IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;

```

然而，这种方法需要为每个IF编写一个匹配的END IF，所以当有许多备选方案时，它比使用ELSIF更加麻烦。

#### 42.6.4.4. 简单的CASE

```

CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
        statements
    [ WHEN expression [, expression [ ... ]] THEN
        statements
    ... ]
    [ ELSE
        statements ]
END CASE;

```

简单的CASE形式提供了基于操作数的相等性条件执行。search-expression被评估（一次）并逐个与WHEN子句中的每个expression进行比较。如果找到匹配项，则执行相应的statements，然后控制传递到 END CASE之后的下一条语句。（后续的WHEN表达式不会被评估。）如果找不到匹配项，则执行ELSE statements；但是如果找不到ELSE，则会引发CASE\_NOT\_FOUND异常。

下面是一个简单的例子：

```
CASE x
  WHEN 1, 2 THEN
    msg := 'one or two';
  ELSE
    msg := 'other value than one or two';
END CASE;
```

#### 42.6.4.5. 搜索型CASE

```
CASE
  WHEN boolean-expression THEN
    statements
  [ WHEN boolean-expression THEN
    statements
  ... ]
  [ ELSE
    statements ]
END CASE;
```

CASE 的搜索形式根据布尔表达式的真值提供条件执行。每个 WHEN 子句的 boolean-expression 会依次被评估，直到找到一个产生 true 的表达式。然后执行相应的 statements，然后控制权转移到 END CASE 后的下一条语句。（后续的 WHEN 表达式不会被评估。）如果找不到真实结果，则会执行 ELSE statements；但是如果找不到 ELSE，则会引发 CASE\_NOT\_FOUND 异常。

下面是一个例子：

```
CASE
  WHEN x BETWEEN 0 AND 10 THEN
    msg := 'value is between zero and ten';
  WHEN x BETWEEN 11 AND 20 THEN
    msg := 'value is between eleven and twenty';
END CASE;
```

这种形式的 CASE 和 IF-THEN-ELSIF 完全等价，唯一的区别是遗漏了 ELSE 子句会导致错误，而不是什么都不做。

### 42.6.5. 简单循环

使用 LOOP、EXIT、CONTINUE、WHILE、FOR 和 FOREACH 语句，您可以安排 PL/oraSQL 函数重复执行一系列命令。

#### 42.6.5.1. LOOP

```
[ <<label>> ]
LOOP
  statements
END LOOP [ label ];
```

LOOP 定义了一个无条件循环，直到被 EXIT 或 RETURN 语句终止。可选的 label 可以被嵌套循环内的 EXIT 和 CONTINUE 语句使用，以指定这些语句所引用的循环。

### Tip

标签名称不能是保留字 label。

#### 42.6.5.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

如果没有给出 label，则最内层的循环将被终止，然后执行 END LOOP 后的语句。如果给出了 label，则它必须是当前循环或某个外层嵌套循环或块的标签。然后，命名的循环或块将被终止，并继续执行循环/块对应的 END 语句之后的语句。

如果指定了 WHEN，则只有在 boolean-expression 为 true 时才会退出循环。否则，控制权传递到 EXIT 后的语句。

EXIT 可以与所有类型的循环一起使用，它不仅限于与无条件循环一起使用。

例如：

```
LOOP
  -- some computations
  IF count > 0 THEN
    EXIT; -- 退出循环
  END IF;
END LOOP;

LOOP
  -- some computations
  EXIT WHEN count > 0; -- 与前面的例子产生相同的结果
END LOOP;
```

#### 42.6.5.3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

如果没有给出 标签，则开始内层循环的下一次迭代。也就是说，跳过循环体中剩余的所有语句，控制返回循环控制表达式（如果有的话），以确定是否需要进行一次循环迭代。如果有 标签，则指定将继续执行的循环的标签。

如果指定了 WHEN，那么仅当 布尔表达式 为真时才开始循环的下次迭代。否则，控制传递到 CONTINUE 之后的语句。

CONTINUE 可以与所有类型的循环一起使用，它不仅仅局限于无条件循环的使用。

示例：

```
LOOP
  -- 一些计算
  EXIT WHEN count > 100;
  CONTINUE WHEN count < 50;
  -- 一些计算，用于计数在 [50 .. 100] 范围内的情况
```

```
END LOOP;
```

#### 42.6.5.4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

WHILE 语句会重复执行一系列语句，只要 boolean-expression 评估为 true。该表达式在每次进入循环体之前进行检查。

例如：

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- 在这里进行一些计算
END LOOP;
```

```
WHILE NOT done LOOP
    -- 在这里进行一些计算
END LOOP;
```

#### 42.6.5.5. FOR (整数变量)

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

这种形式的FOR创建一个循环，遍历整数值的范围。变量name会自动定义为integer类型，仅在循环内存在（循环内会忽略任何现有的该变量的定义）。指定范围的两个表达式在进入循环时只会计算一次。如果没有指定BY子句，则迭代步长为1，否则为BY子句中指定的值，该值也会在循环进入时计算一次。如果指定了REVERSE，则在每次迭代后，步长值会被减去而不是加上。

一些整数型FOR循环的例子：

```
FOR i IN 1..10 LOOP
    -- i在循环内将取值1, 2, 3, 4, 5, 6, 7, 8, 9, 10
END LOOP;
```

```
FOR i IN REVERSE 10..1 LOOP
    -- i在循环内将取值10, 9, 8, 7, 6, 5, 4, 3, 2, 1
END LOOP;
```

```
FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i在循环内将取值10, 8, 6, 4, 2
END LOOP;
```

如果下限大于上限（或者在REVERSE情况下，小于上限），则根本不执行循环体。不会引发任何错误。

如果FOR循环附有一个label，则可以使用该label来限定名称，从而引用整数型循环变量。

#### 42.6.6. GOTO语句



GOTO语句无条件地使控制权转到一个标签处。该标签在其作用域内必须是唯一的，并且必须位于一条可执行语句或一个PL/oraSQL块之前。当程序运行时，GOTO语句将控制权转到一个带标签的语句或块处。

语法如下：

```
GOTO labelName;
```

#### 42.6.6.1. GOTO语句的限制

- GOTO语句无法将控制权转入IF语句、CASE语句、LOOP语句或子块中。
- GOTO语句无法将控制权从一个IF语句子句传到另一个子句，或者从一个CASE语句的WHEN子句传到另一个子句。
- GOTO语句无法将控制权传出子程序。
- GOTO语句无法将控制权传入异常处理程序。
- GOTO语句无法将控制权从异常处理程序传回当前块。
- LightDB的关键词无法作为标签名。

#### 42.6.6.2. 例子

该示例可以在ltsql下运行。进入ltsql，执行下列命令。

```
create database test_oracle lightdb_syntax_compatible_type oracle;
\c test_oracle
select dbms_output.serveroutput(true);
```

##### 42.6.6.2.1. 示例1 GOTO到语句

```
DECLARE
  p VARCHAR2(30);
  n PLS_INTEGER := 37;
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n % j = 0 THEN
      p := ' is not a prime number';
      GOTO print_now;
    END IF;
  END LOOP;

  p := ' is a prime number';

  <<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

结果：

```
37 is a prime number
```

##### 42.6.6.2.2. 示例2 GOTO到NULL

```

DECLARE
  done BOOLEAN;
  j     int;
BEGIN
  done := true;
  FOR i IN 1..50 LOOP
    IF done THEN
      j := 100;
      GOTO end_loop;
    END IF;
  END LOOP;
  <<end_loop>>
  NULL;
  DBMS_OUTPUT.PUT_LINE(j);
END;
/

```

结果:  
100

#### 42.6.6.2.3. 示例3 GOTO到块

```

BEGIN
  DBMS_OUTPUT.PUT_LINE(' out block 1 ');
  goto testlabel;
  DBMS_OUTPUT.PUT_LINE(' out block 2 ');
  <<testlabel>>
  BEGIN
    DBMS_OUTPUT.PUT_LINE(' in block 1 ');
    goto testlabel2;
    DBMS_OUTPUT.PUT_LINE(' in block 2 ');
  END;
  DBMS_OUTPUT.PUT_LINE(' out block 3 ');
  <<testlabel2>>
  DBMS_OUTPUT.PUT_LINE(' out block 4 ');
END;
/

```

结果:  
out block 1  
in block 1  
out block 4

#### 42.6.6.2.4. 示例4 语句无法将控制权传到IF语句中

GOTO语句将控制权转入IF语句，导致出现错误。

```

DECLARE
  done BOOLEAN := TRUE;
BEGIN
  GOTO update_row;

  IF done THEN
    <<update_row>>
  NULL;

```

```

    END IF;
END;
/

```

结果:

```

ERROR: illegal GOTO statement; this GOTO cannot branch to label
"update_row"
CONTEXT: PL/oraSQL function inline_code_block

```

## 42.6.7. 遍历查询结果

使用不同类型的FOR循环，可以遍历查询结果并相应地操作该数据。语法如下：

```

[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];

```

target是一个记录变量、行变量或逗号分隔的标量变量列表。target依次分配query生成的每一行，并为每一行执行循环体。以下是一个例子：

```

create table refresh_mview (mv_schema varchar(100), mv_name varchar(100), owner
    varchar(100));
insert into refresh_mview values ('asd', 'sdf', 'dfg');
CREATE OR REPLACE FUNCTION refresh_mviews(id int) RETURN integer IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Refreshing all materialized views...');

    FOR mviews IN (SELECT mv_schema, mv_name, owner FROM refresh_mview)
    LOOP
        -- Now "mviews" has one record with information about the materialized
        view
        DBMS_OUTPUT.PUT_LINE('Refreshing materialized view ' ||
            mviews.mv_schema || '.' ||
                mviews.mv_name || ' (owner: %)...' || mviews.owner);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Done refreshing materialized views. ');
    RETURN id;
END;
/

```

如果循环由EXIT语句终止，则最后分配的行值在循环后仍然可访问。

在这种类型的FOR语句中使用的query可以是任何返回行的SQL命令：SELECT是最常见的情况，但您也可以使用带有RETURNING子句的INSERT、UPDATE或DELETE。一些实用程序命令，如EXPLAIN也可以使用。

PL/oraSQL变量会被替换为查询文本，并且查询计划会被缓存以便可能重复使用。

FOR-IN-EXECUTE-IMMEDIATE语句是迭代行的另一种方式：

```

[ <<label>> ]
FOR target IN EXECUTE IMMEDIATE text_expression [ USING expression [, ... ] ]
LOOP
    statements

```

```
END LOOP [ label ];
```

这类似于前面的形式，但源查询指定为字符串表达式，在进入FOR循环的每个条目上进行评估和重新规划。这允许程序员选择预安排查询的速度或动态查询的灵活性，就像普通的EXECUTE IMMEDIATE语句一样。与EXECUTE IMMEDIATE一样，可以通过USING将参数值插入动态命令中。

另一种指定应遍历其结果的查询的方法是将其声明为游标。这在Section 42.7.3. 中有描述。

## 42.6.8. 捕获错误

默认情况下，在PL/oraSQL函数中发生任何错误都会中止函数的执行以及周围的事务。可以通过使用带有EXCEPTION子句的BEGIN块来捕获错误并从中恢复。语法是BEGIN块的正常语法的扩展形式：

```
[ DECLARE
  declarations ]
BEGIN
  statements
EXCEPTION
  WHEN condition [ OR condition ... ] THEN
    handler_statements
  [ WHEN condition [ OR condition ... ] THEN
    handler_statements
  ... ]
END;
```

如果没有错误发生，此块形式只会执行所有statements，然后控制权转移到END后的下一条语句。但是，如果在statements中发生错误，则进一步处理statements被放弃，并且控制权转移到EXCEPTION列表。该列表会查找与发生的错误匹配的第一个condition。如果找到匹配项，则执行相应的handler\_statements，然后控制权传递到END后的下一条语句。如果找不到匹配项，则错误会像根本没有EXCEPTION子句一样传播：错误可以由带有EXCEPTION 的封闭块捕获，如果没有，则会中止函数的处理。

特殊的条件名称OTHERS匹配每种错误类型。条件名称不区分大小写。

### Tip

包含EXCEPTION子句的块进入和退出的成本要比没有该子句的块高得多。因此，在没有必要的情况下不要使用EXCEPTION。

## 42.7. 游标

可以设置一个封装查询的游标，然后一次读取几行查询结果，而不是立即执行整个查询。这样做的一个原因是避免在结果包含大量行时出现内存溢出。(然而，PL/oraSQL用户通常不需要担心这个问题，因为FOR循环在内部自动使用游标以避免内存问题。)更有趣的用法是返回一个函数创建的游标的引用，允许调用者读取行。这提供了一种从函数返回大型行集的有效方式。

游标是指向存储有关处理特定SELECT或DML语句的信息的私有SQL区域的指针。

由PL/oraSQL构建和管理的游标是隐式游标。您构建和管理的游标是显式游标。您可以从其属性获取有关任何会话游标的信息(您可以在过程语句中引用这些属性，但不能在SQL语句中引用)。

## 42.7.1. 隐式游标

隐式游标是由PL/oraSQL构建和管理的会话游标。每次运行SELECT或DML语句时，PL/oraSQL都会打开一个隐式游标。您无法控制隐式游标，但可以从其属性获取信息。

与之关联的语句运行后，隐式游标会关闭；但是，其属性值仍然可用，直到运行另一个SELECT或DML语句。

最近运行的SELECT或DML语句可能处于不同的作用域中。为了保存属性值以供以后使用，请立即将其分配给本地变量。否则，其他操作，例如子程序调用，可能会在您测试之前更改属性的值。

### 42.7.1.1. SQL%ISOPEN属性：游标是否打开？

SQL%ISOPEN总是返回FALSE，因为一个隐式游标在关联的语句执行完后总是会关闭。

### 42.7.1.2. SQL%FOUND属性：是否有任何行受到影响？

SQL%FOUND返回：

- 如果没有SELECT或DML语句运行，则返回NULL，
- 如果SELECT语句返回了一个或多个行或DML语句影响了一个或多个行，则返回TRUE，
- 否则返回FALSE。

使用 SQL%FOUND 来确定 DELETE 语句是否影响了任何行。例如：

```
create table employees(salary int, job_id varchar(100), employee_id int,
  last_name varchar(100), first_name varchar(100), department_id int, hire_date
  date);
create table departments(department_id int, department_name varchar(100),
  location_id int, staff int);
create table locations(country_id varchar(100),city varchar(100),location_id
  int);
create table jobs(job_title varchar(100), job_id varchar(100));

insert into employees values (12308, 'SA_REP', 100, 'huaxiang', 'zhao', 2345,
  TO_DATE('31-DEC-2005', 'DD-MON-YYYY'));
insert into employees values (12301, 'AD_FG', 12301, 'huahai', 'zhao', 2345,
  TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
insert into employees values (12302, 'SA_REP', 12302, 'huaduo', 'zhao', 2345,
  TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
insert into employees values (12302, 'S[HT]_CLERK', 12302, 'huashuai', 'zhao',
  2345, TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
insert into employees values (12302, 'AD_FG', 100, 'huashuai', 'zhao', 2345,
  TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
insert into employees values (12304, 'AD_FG', 100, 'huashuai', 'zhao', 2345,
  TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
insert into employees values (12305, 'AD_FG', 100, 'huashuai', 'zhao', 2345,
  TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
insert into employees values (12306, 'AD_FG', 100, 'huashuai', 'zhao', 2345,
  TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
insert into employees values (12307, 'AD_FG', 100, 'huashuai', 'zhao', 2345,
  TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
insert into employees values (12308, 'AD_MAN', 100, 'huaxiang', 'zhao', 2345,
  TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
```

```

insert into departments values (110, 'asd', 1700, 2345);
insert into departments values (2345, 'asd', 1800, 2345);
insert into departments values (110, 'asd', 1900, 2345);
insert into departments values (110, 'AD_FG', 12302, 2345);

insert into locations values ('JP', 'Japan', 1700);
insert into locations values ('Toronto', 'Canada', 1800);
insert into locations values ('WH', 'Whitehorse', 1900);

insert into jobs values ('sdadf', 'AD_FG');

DROP TABLE dept_temp;
CREATE TABLE dept_temp AS
  SELECT * FROM departments;

DECLARE
  dept_no NUMBER := 12;
BEGIN
  DELETE FROM dept_temp
  WHERE department_id = dept_no;

  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('Delete succeeded for department number ' || dept_no);
  ELSE
    DBMS_OUTPUT.PUT_LINE ('No department number ' || dept_no);
  END IF;
END;
/

```

#### 42.7.1.3. SQL%NOTFOUND Attribute: Were Any Rows Affected?

SQL%NOTFOUND（逻辑上与 SQL%FOUND 相反）返回：

- 如果没有运行 SELECT 或 DML 语句，则返回 NULL。
- 如果 SELECT 语句返回了一行或多行，或者 DML 语句影响了一行或多行，则返回 FALSE。

#### 42.7.1.4. SQL%ROWCOUNT 属性：有多少行受到影响？

SQL%ROWCOUNT 返回：

- 如果没有运行 SELECT 或 DML 语句，则返回 NULL。
- 否则，返回 SELECT 语句返回的行数或被 DML 语句影响的行数（一个 INTEGER 类型）。

使用 SQL%ROWCOUNT 来确定被删除的行数。例如：

```

DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT * FROM employees;

DECLARE
  mgr_no int := 123005;
  rowcount int := 0;
BEGIN
  DELETE FROM employees_temp WHERE employee_id = mgr_no;
  rowcount := SQL%ROWCOUNT;
  DBMS_OUTPUT.PUT_LINE ('Number of employees deleted: ' || rowcount);

```

```
END;
/
```

## 42.7.2. 显式游标

显式游标是一种会话游标，由您构造和管理。您必须声明和定义一个显式游标，给它起一个名称并将其与一个查询关联（通常，查询会返回多行）。然后，您可以通过以下两种方式之一处理查询结果集：

- Open 显式游标，从结果集中 fetch 行，并 close 显式游标。
- 在游标 FOR LOOP 语句中使用显式游标。

与 隐式游标 不同，您可以通过名称引用 显式游标 或游标变量。因此，显式游标或游标变量被称为命名游标。

### 42.7.2.1. 声明和定义显式游标

您可以先声明显式游标，然后在同一块、子程序或包中稍后定义它，也可以同时声明和定义它。

显式游标声明，仅声明游标，其语法如下：

```
CURSOR cursor_name parameter_list RETURN return_type;
```

显式游标定义 的语法如下：

```
CURSOR cursor_name parameter_list RETURN return_type IS select_statement;
```

要实际使用显式游标，可以直接定义它而不声明它。

在下面的示例中，声明并定义了三个显式游标：

```
DECLARE
  CURSOR c1 RETURN departments%ROWTYPE;    -- 声明 c1

  CURSOR c2 IS                               -- 声明并定义 c2
    SELECT employee_id, job_id, salary FROM employees
    WHERE salary > 2000;

  CURSOR c1 RETURN departments%ROWTYPE IS   -- 定义 c1,
    SELECT * FROM departments              -- 重复返回类型
    WHERE department_id = 110;

  CURSOR c3 RETURN locations%ROWTYPE;      -- 声明 c3

  CURSOR c3 IS                               -- 定义 c3,
    SELECT * FROM locations                -- 省略返回类型
    WHERE country_id = 'JP';
BEGIN
  NULL;
END;
/
```

### 42.7.2.2. 打开和关闭显式游标

声明并定义了显式游标后，可以使用OPEN语句打开它。

使用CLOSE语句关闭打开的显式游标，从而允许重用其资源。关闭游标后，无法从其结果集中获取记录或引用其属性。

### 42.7.2.3. 使用显式游标获取数据

打开显式游标后，可以使用FETCH语句获取查询结果集的行。返回一行的FETCH语句的基本语法如下：

```
FETCH cursor_name INTO into_clause;
```

into\_clause 是变量列表或单个记录变量。对于查询返回的每一列，变量列表或记录必须有一个相应的类型兼容变量或字段。在声明用于FETCH语句的变量和记录时，%TYPE和%ROWTYPE属性非常有用。

FETCH语句检索结果集的当前行，将该行的列值存储到变量或记录中，并将游标移动到下一行。

通常，您在LOOP语句中使用FETCH语句，当FETCH语句运行完所有行时退出循环。要检测此退出条件，请使用游标属性%NOTFOUND。当FETCH语句返回没有行时，PL/oraSQL不会引发异常。

使用FETCH和%NOTFOUND在LOOP语句中一次一行地检索两个显式游标的结果集。第一个FETCH语句将列值检索到变量中，第二个FETCH语句将列值检索到记录中。变量和记录分别使用%TYPE和%ROWTYPE声明。例如：

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
    ORDER BY last_name;

  v_lastname employees.last_name%TYPE; -- variable for last_name
  v_jobid employees.job_id%TYPE;      -- variable for job_id

  CURSOR c2 IS
    SELECT * FROM employees
    WHERE REGEXP_LIKE (job_id, '[ACADFIMKSA]_M[ANGR]')
    ORDER BY job_id;

  v_employees employees%ROWTYPE; -- record variable for row of table

BEGIN
  OPEN c1;
  LOOP -- Fetches 2 columns into variables
    FETCH c1 INTO v_lastname, v_jobid;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (RPAD(v_lastname, 25, ' ') || v_jobid);
  END LOOP;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE ('-----');

  OPEN c2;
  LOOP -- Fetches entire row into the v_employees record
    FETCH c2 INTO v_employees;
    EXIT WHEN c2%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (RPAD(v_employees.last_name, 25, ' ') ||
v_employees.job_id);
  END LOOP;
```



```

    CLOSE c2;
END;
/

```

这个操作通过使用五个FETCH语句，将结果集的前五行数据分别装载到五个记录变量中。这些记录变量使用%ROWTYPE进行声明。例如：

```

DECLARE
    CURSOR c IS
        SELECT e.job_id, j.job_title
        FROM employees e, jobs j
        WHERE e.job_id = j.job_id AND e.employee_id = 100
        ORDER BY last_name;

    -- Record variables for rows of cursor result set:

    job1 c%ROWTYPE;
    job2 c%ROWTYPE;
    job3 c%ROWTYPE;
    job4 c%ROWTYPE;
    job5 c%ROWTYPE;

BEGIN
    OPEN c;
    FETCH c INTO job1; -- fetches first row
    FETCH c INTO job2; -- fetches second row
    FETCH c INTO job3; -- fetches third row
    FETCH c INTO job4; -- fetches fourth row
    FETCH c INTO job5; -- fetches fifth row
    CLOSE c;

    DBMS_OUTPUT.PUT_LINE (job1.job_title || ' (' || job1.job_id || ')');
    DBMS_OUTPUT.PUT_LINE (job2.job_title || ' (' || job2.job_id || ')');
    DBMS_OUTPUT.PUT_LINE (job3.job_title || ' (' || job3.job_id || ')');
    DBMS_OUTPUT.PUT_LINE (job4.job_title || ' (' || job4.job_id || ')');
    DBMS_OUTPUT.PUT_LINE (job5.job_title || ' (' || job5.job_id || ')');
END;
/

```

#### 42.7.2.4. 显式游标查询中的变量

显式游标查询可以引用其作用域内的任何变量。当您打开显式游标时，PL/oraSQL会评估查询中的任何变量，并在识别结果集时使用这些值。后来更改变量的值不会改变结果集。

显式游标查询引用变量factor。当游标打开时，factor的值为2。因此，sal\_multiple始终为sal的两倍，尽管在每次检索后都会增加factor的值。例如：

```

DECLARE
    sal          employees.salary%TYPE;
    sal_multiple employees.salary%TYPE;
    factor       INTEGER := 2;

    CURSOR c1 IS
        SELECT salary, salary*factor FROM employees
        WHERE job_id LIKE 'AD_%';

BEGIN

```

```

OPEN c1; -- PL/SQL evaluates factor

LOOP
  FETCH c1 INTO sal, sal_multiple;
  EXIT WHEN c1%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE ('factor = ' || factor);
  DBMS_OUTPUT.PUT_LINE ('sal          = ' || sal);
  DBMS_OUTPUT.PUT_LINE ('sal_multiple = ' || sal_multiple);
  factor := factor + 1; -- Does not affect sal_multiple
END LOOP;

CLOSE c1;
END;
/

```

显式游标查询中的变量-结果集更改。例如：

```

DECLARE
  sal          employees.salary%TYPE;
  sal_multiple employees.salary%TYPE;
  factor       INTEGER := 2;

CURSOR c1 IS
  SELECT salary, salary*factor FROM employees
  WHERE job_id LIKE 'AD_%';

BEGIN
  DBMS_OUTPUT.PUT_LINE ('factor = ' || factor);
  OPEN c1; -- PL/SQL evaluates factor
  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE ('sal_multiple = ' || sal_multiple);
  END LOOP;
  CLOSE c1;

  factor := factor + 1;

  DBMS_OUTPUT.PUT_LINE ('factor = ' || factor);
  OPEN c1; -- PL/SQL evaluates factor
  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE ('sal_multiple = ' || sal_multiple);
  END LOOP;
  CLOSE c1;
END;
/

```

#### 42.7.2.5. 显式游标查询需要列别名时

当显式游标查询包括虚拟列（表达式）时，如果符合以下任一条件，则该列必须有别名：

使用CLOSE语句可以关闭一个开启的显式游标，从而允许其资源被重用。关闭游标之后，您不能再从其结果集中获取记录或引用其属性。如果尝试这样做，PL/oraSQL会引发预定义的异常INVALID\_CURSOR。

- 如果您使用的游标是用%ROWTYPE声明的记录变量，则可以使用该游标将数据装载到该记录变量中。
- 如果您想在程序中引用虚拟列，则可以使用该游标。

显式游标中的虚拟列需要为上述两个原因都提供别名。例如：

```

DECLARE
  CURSOR c1 IS
    SELECT employee_id,
           (salary * .05) raise
    FROM employees
    WHERE job_id LIKE '%_MAN'
    ORDER BY employee_id;
  emp_rec c1%ROWTYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('Raise for employee #' || emp_rec.employee_id || ' is
  $' || emp_rec.raise);
  END LOOP;
  CLOSE c1;
END;
/

```

#### 42.7.2.6. 接受参数的显式游标

您可以创建具有形式参数的显式游标，然后每次打开游标时传递不同的实际参数。在游标查询中，您可以在可以使用常量的任何位置使用形式游标参数。在游标查询之外，您无法引用形式游标参数。

创建一个显式游标，其两个形式参数表示一个工作和其最高工资。当使用指定的工作和最高工资打开游标时，游标查询会选择那些薪资过高的职员（对于每个这样的职员，查询会选择其名字、姓氏和超支金额）。接下来，该示例创建一个打印游标查询结果集的过程（关于过程的信息，请参阅PL/SQL子程序）。最后，该示例使用一组实际参数打开游标，打印结果集，关闭游标，然后使用不同的实际参数打开游标，打印结果集并关闭游标。例如：

```

DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER) IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    ORDER BY salary;

  last_name_ employees.last_name%TYPE;
  first_name_ employees.first_name%TYPE;
  overpayment_ employees.salary%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('-----');
  DBMS_OUTPUT.PUT_LINE ('Overpaid Stock Clerks:');
  DBMS_OUTPUT.PUT_LINE ('-----');
  OPEN c('ST_CLERK', 5000);
  LOOP

```

```

    FETCH c INTO last_name_, first_name_, overpayment_;
    EXIT WHEN c%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (last_name_ || ',' || first_name_ || ', (by' ||
overpayment_ || ')');
    END LOOP;
    CLOSE c;

    DBMS_OUTPUT.PUT_LINE ('-----');
    DBMS_OUTPUT.PUT_LINE ('Overpaid Sales Representatives:');
    DBMS_OUTPUT.PUT_LINE ('-----');
    OPEN c('SA_REP', 10000);
    LOOP
        FETCH c INTO last_name_, first_name_, overpayment_;
        EXIT WHEN c%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (last_name_ || ',' || first_name_ || ', (by' ||
overpayment_ || ')');
        END LOOP;
        CLOSE c;
    END;
/

```

#### 42.7.2.7. 具有默认值的形式游标参数

当您创建一个具有形式参数的显式游标时，您可以为它们指定默认值。当形式参数具有默认值时，相应的实际参数是可选的。如果您在不指定实际参数的情况下打开游标，那么形式参数具有默认值。

```
parameter_name [IN] data_type [:= | default expression]
```

应用场景包括存储过程、函数、匿名块和包。与Oracle相比，有一些差异。例如，在 data\_type 中，Oracle 支持在游标参数字符串中声明 char，但是 lightdb 的 char 长度默认值为 1，因此对于长度超过 1 的字符串，您需要指定 char 长度，char(100)。但对于可变长度字符类型，如 varchar、varchar2 等，使用方式与 Oracle 相同。并且不支持 Oracle 类型：string、long、raw、long raw、rowid、timestamp with local time 等。在包中不支持由 constant 定义的常量类型。

与 Oracle 相比，Expressions 也有一些差异。

- 布尔表达式
  - 包支持返回布尔类型、布尔表达式和布尔字面量的函数（支持 NULL，Oracle 不支持），
  - 函数、存储过程和匿名块支持返回布尔类型、布尔常量、布尔变量、布尔表达式和布尔字面量的函数（支持 NULL，Oracle 不支持）。
- 字符表达式
  - 包支持返回 characer 类型和 characer 字面量的函数，
  - 函数、存储过程和匿名块支持返回 characer 类型、characer 常量、characer 变量、characer 表达式和 characer 字面量的函数。
- 日期表达式
  - 包支持返回 date 类型和 date 字面量的函数，
  - 函数、存储过程和匿名块支持返回 date 类型、date 常量、date 变量、date 表达式和 date 字面量的函数。

- 数值表达式
  - 包中不支持：  
collection、named\_cursor、numeric\_constant、numeric\_variable、placeholder、SQL  
% cursor attribute、\*\*（不支持的运算符），
  - 函数、存储过程和匿名块中不支持：collection、named\_cursor、placeholder、SQL%  
cursor attribute、\*\*（不支持的运算符）。
- Case 表达式
  - 包、函数、存储过程和匿名块中支持 search\_case\_expression 和  
simple\_case\_expression。

创建一个显式游标，其形式参数代表位置ID。参数的默认值是公司总部的位置ID。例如：

```

DECLARE
  CURSOR c (location NUMBER DEFAULT 1700) IS
    SELECT d.department_name,
           e.last_name manager,
           l.city
    FROM departments d, employees e, locations l
    WHERE l.location_id = location
          AND l.location_id = d.location_id
          AND d.department_id = e.department_id
    ORDER BY d.department_id;
  dept_name departments.department_name%TYPE;
  mgr_name employees.last_name%TYPE;
  city_name locations.city%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('DEPARTMENTS AT HEADQUARTERS:');
  DBMS_OUTPUT.PUT_LINE ('-----');
  OPEN c;
  LOOP
    FETCH c INTO dept_name, mgr_name, city_name;
    EXIT WHEN c%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (dept_name || ' (Manager: ' || mgr_name || ')');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('-----');
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE ('DEPARTMENTS IN CANADA:');
  DBMS_OUTPUT.PUT_LINE ('-----');
  OPEN c(1800); -- Toronto
  LOOP
    FETCH c INTO dept_name, mgr_name, city_name;
    EXIT WHEN c%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (dept_name || ' (Manager: ' || mgr_name || ')');
  END LOOP;
  CLOSE c;
  OPEN c(1900); -- Whitehorse
  LOOP
    FETCH c INTO dept_name, mgr_name, city_name;
    EXIT WHEN c%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (dept_name || ' (Manager: ' || mgr_name || ')');
  END LOOP;
  CLOSE c;
END;

```

/

如果你向游标添加形式参数，并为这些参数指定了默认值，那么您无需更改对游标的现有引用。例如：

```

DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER,
            hired DATE DEFAULT TO_DATE('31-DEC-1999', 'DD-MON-YYYY')) IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    AND hire_date > hired
    ORDER BY salary;
  last_name_ employees.last_name%TYPE;
  first_name_ employees.first_name%TYPE;
  overpayment_ employees.salary%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('-----');
  DBMS_OUTPUT.PUT_LINE ('Overpaid Sales Representatives Hired After 2004:');
  DBMS_OUTPUT.PUT_LINE ('-----');
  OPEN c('SA_REP', 10000);
  LOOP
    FETCH c INTO last_name_, first_name_, overpayment_;
    EXIT WHEN c%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (last_name_ || ', (by ' || overpayment_ || ')');
  END LOOP;
  CLOSE c;
END;
/

```

#### 42.7.2.8. 显式游标属性

显式游标属性的语法是cursor\_name，紧跟着属性（例如，c1%ISOPEN）。

使用CLOSE语句关闭一个打开的显式游标，从而允许其资源被重用。关闭游标之后，您就无法从其结果集中提取记录或引用其属性。如果尝试这样做，PL/oraSQL会引发预定义的异常INVALID\_CURSOR。

##### 42.7.2.8.1. %ISOPEN属性：游标是否打开？

如果显式游标处于打开状态，%ISOPEN会返回true，否则返回false。

仅当显式游标c1未打开时，才打开它；仅当它已打开时，才关闭它。例如：

```

DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11;

  the_name employees.last_name%TYPE;
  the_salary employees.salary%TYPE;
BEGIN
  IF NOT c1%ISOPEN THEN
    OPEN c1;
  END IF;

```

```

    FETCH c1 INTO the_name, the_salary;

    IF c1%ISOPEN THEN
        CLOSE c1;
    END IF;
END;
/

```

要使用ROWNUM, lightdb\_dblevel\_syntax\_compatible\_type的值必须为oracle。

#### 42.7.2.8.2. %FOUND属性：是否有行受影响？

%FOUND 返回：

- 在显式游标打开但在第一次提取之前返回NULL，
- 如果最近一次从显式游标提取了一行，则返回TRUE，
- 否则返回FALSE。

循环遍历结果集，打印每个提取的行，并在没有更多行可提取时退出。例如：

```

DECLARE
    CURSOR c1 IS
        SELECT last_name, salary FROM employees
        WHERE ROWNUM < 11
        ORDER BY last_name;

    my_ename    employees.last_name%TYPE;
    my_salary   employees.salary%TYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO my_ename, my_salary;
        IF c1%FOUND THEN -- fetch succeeded
            DBMS_OUTPUT.PUT_LINE ('Name = ' || my_ename || ' salary = ' || my_salary);
        ELSE -- fetch failed
            EXIT;
        END IF;
    END LOOP;
END;
/

```

#### 42.7.2.8.3. %NOTFOUND属性：是否有行受影响？

%NOTFOUND (the logical opposite of %FOUND) returns:

- 在显式游标打开但在第一次提取之前返回NULL，
- 如果最近一次从显式游标提取了一行，则返回FALSE，
- 否则返回TRUE。

当FETCH未返回行时，%NOTFOUND用于退出循环。例如：

```

DECLARE
    CURSOR c1 IS
        SELECT last_name FROM employees

```

```

WHERE ROWNUM < 11
ORDER BY last_name;
rowcount int;
name_1 employees.last_name%TYPE;
BEGIN
OPEN c1;
LOOP
  FETCH c1 INTO name_1;
  EXIT WHEN c1%NOTFOUND;
  rowcount := c1%ROWCOUNT;
  DBMS_OUTPUT.PUT_LINE (rowcount || '. ' || name_1);
  IF c1%ROWCOUNT = 5 THEN
    DBMS_OUTPUT.PUT_LINE ('--- Fetched 5th row ---');
  END IF;
END LOOP;
CLOSE c1;
END;
/

```

#### 42.7.2.8.4. %ROWCOUNT属性：有多少行受到影响？

%ROWCOUNT returns:

- 在显式游标打开但在第一次提取之前返回零，
- 否则，返回提取的行数（一个INTEGER）。

对提取的行进行编号并打印，并在提取第五行后打印一条消息。例如：

```

DECLARE
CURSOR c1 IS
  SELECT last_name FROM employees
  WHERE ROWNUM < 11
  ORDER BY last_name;

name_1 employees.last_name%TYPE;
rowcount int;
BEGIN
OPEN c1;
LOOP
  FETCH c1 INTO name_1;
  EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
  rowcount := c1%ROWCOUNT;
  DBMS_OUTPUT.PUT_LINE (rowcount || '. ' || name_1);
  IF c1%ROWCOUNT = 5 THEN
    DBMS_OUTPUT.PUT_LINE ('--- Fetched 5th row ---');
  END IF;
END LOOP;
CLOSE c1;
END;
/

```

### 42.7.3. 处理查询结果集

在PL/oraSQL中，与传统的数据库编程一样，您使用游标来处理查询结果集。然而，在PL/oraSQL中，您可以使用隐式游标或显式游标。

前者需要的代码较少，但后者更加灵活。例如，显式游标可以接受参数。



以下 PL/oraSQL 语句使用 PL/oraSQL 为您定义和管理的隐式游标：

- SELECT INTO。
- 隐式游标 FOR LOOP。

以下 PL/oraSQL 语句使用显式游标：

- 显式游标 FOR LOOP。
- OPEN、FETCH 和 CLOSE。

### 42.7.3.1. 使用SELECT INTO语句处理查询结果集

使用隐式游标，SELECT INTO语句从一个或多个数据库表中检索值（就像SQL的SELECT语句一样）并将它们存储在变量中（这是SQL的SELECT语句所不具备的）。

#### 42.7.3.1.1. 处理单行结果集

如果您希望查询仅返回一行，则使用SELECT INTO语句将该行的值存储在一个或多个标量变量，或一个记录变量中。

如果查询可能返回多行，但您只关心第n行，则可以使用WHERE ROWNUM=n子句将结果集限制为该行。

#### 42.7.3.1.2. 处理大型多行结果集

如果您必须将大量的表数据分配给变量，LightDB建议使用带有BULK COLLECT子句的SELECT INTO语句。

### 42.7.3.2. 使用游标FOR LOOP语句处理查询结果集

游标FOR LOOP语句允许您运行SELECT语句，然后立即循环遍历结果集中的行。

这个语句可以使用隐式或显式游标（但不能使用游标变量）。

如果您只在游标的 FOR LOOP 语句中使用SELECT语句，则应该在游标的 FOR LOOP 语句内部指定SELECT语句。这称为隐式游标的FOR LOOP语句。例如：

```
BEGIN
  FOR item IN (
    select last_name, job_id
    FROM employees
    WHERE job_id LIKE '%CLERK%'
    AND employee_id > 120
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Name = ' || item.last_name || ', Job = ' ||
item.job_id);
  END LOOP;
END;
/
```

显式游标FOR LOOP语句。例如：

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE job_id LIKE '%CLERK%' AND employee_id > 120
```

```

        ORDER BY last_name;
BEGIN
    FOR item IN c1
    LOOP
        DBMS_OUTPUT.PUT_LINE ('Name = ' || item.last_name || ', Job = ' ||
            item.job_id);
    END LOOP;
END;
/

```

传递参数给显式游标FOR LOOP语句。例如：

```

DECLARE
    CURSOR c1 (job VARCHAR2, max_wage NUMBER) IS
        SELECT * FROM employees
        WHERE job_id = job
        AND salary > max_wage;
BEGIN
    FOR person IN c1('ST_CLERK', 3000)
    LOOP
        -- 处理数据记录
        DBMS_OUTPUT.PUT_LINE ('Name = ' || person.last_name || ', salary = ' ||
            person.salary || ', Job id= ' || person.job_id);
    END LOOP;
END;
/

```

游标FOR LOOP引用虚拟列。例如：

```

BEGIN
    FOR item IN (
        SELECT first_name || ' ' || last_name AS full_name,
               salary * 10 AS dream_salary
        FROM employees
        WHERE ROWNUM < 6
        ORDER BY dream_salary DESC, last_name ASC
    ) LOOP
        DBMS_OUTPUT.PUT_LINE (item.full_name || ' dreams of making ' ||
            item.dream_salary);
    END LOOP;
END;
/

```

### 42.7.3.3. 使用子查询处理查询结果集

如果你通过循环遍历查询结果集并为每行运行另一个查询来处理结果集，那么你可以通过将第二个查询从循环内部删除并将其作为第一个查询的子查询来提高性能。

虽然普通的子查询对每个表进行评估，但相关子查询对每一行进行评估。

父查询中FROM子句中的子查询。例如：

```

DECLARE
    CURSOR c1 IS
        SELECT t1.department_id, t1.department_name, t2.staff
        FROM departments t1,

```

```

        ( SELECT department_id, COUNT(*) AS staff
          FROM employees
          GROUP BY department_id
        ) t2
WHERE (t1.department_id = t2.department_id) AND t2.staff > 5
ORDER BY t2.staff;

BEGIN
  FOR dept IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Department = ' || dept.department_name || ', staff =
  ' || dept.staff);
  END LOOP;
END;
/

```

相关子查询。例如：

```

DECLARE
  CURSOR c1 IS
    SELECT department_id, last_name, salary
    FROM employees t
    WHERE salary >= ( SELECT AVG(salary)
                      FROM employees
                      WHERE t.department_id = department_id
                    )
    ORDER BY department_id, last_name;
BEGIN
  FOR person IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Making above-average salary = ' ||
  person.last_name);
  END LOOP;
END;
/

```

## 42.7.4. 游标变量

cursor变量类似于显式游标，但具有以下特点：

- 它不限于一个查询。 您可以为一个查询打开一个游标变量，处理结果集，然后将游标变量用于另一个查询。
- 您可以为它赋值。
- 您可以在表达式中使用它。
- 它可以是子程序参数。 您可以使用游标变量在子程序之间传递查询结果集。
- 它可以是宿主变量。 您可以使用游标变量在PL/oraSQL存储的子程序和它们的客户端之间传递查询结果集。
- 它无法接受参数。 您不能向游标变量传递参数，但可以将整个查询传递给它。 查询可以包括变量。

### 42.7.4.1. 创建游标变量

要创建游标变量，请声明一个预定义类型 `SYS_REFCURSOR` 的变量， 或者定义一个 `REF CURSOR` 类型，然后声明一个该类型的变量。

REF CURSOR 类型定义的基本语法如下：

```
TYPE type_name IS REF CURSOR RETURN return_type%TYPE
```

游标变量声明。例如：

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  TYPE genericcurtyp IS REF CURSOR;

  cursor1 empcurtyp;
  cursor2 genericcurtyp;
  my_cursor SYS_REFCURSOR;

  TYPE deptcurtyp IS REF CURSOR RETURN departments%ROWTYPE;
  dept_cv deptcurtyp; -- strong cursor variable
BEGIN
  NULL;
END;
/
```

具有用户定义返回类型的游标变量。例如：

```
DECLARE
  TYPE EmpRecTyp IS RECORD (
    employee_id NUMBER,
    last_name VARCHAR2(25),
    salary NUMBER(8,2));

  TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
  emp_cv EmpCurTyp;
BEGIN
  NULL;
END;
/
```

使用 CREATE DOMAIN 创建的游标类型的变量。例如：

```
CREATE DOMAIN mycursor AS refcursor;
SELECT dbms_output.serveroutput(true);
DECLARE
  employee_cursor mycursor;

  v_employee_name VARCHAR2(50);
BEGIN
  OPEN employee_cursor for SELECT first_name FROM employees_curs WHERE
  employee_id = 1500;

  LOOP
    FETCH employee_cursor INTO v_employee_name;
    EXIT WHEN employee_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_employee_name);
  END LOOP;
```

```

    CLOSE employee_cursor;
END;
/

```

使用sys\_refcursor作为参数变量。例如：

```

CREATE TABLE customer_address
(
    ca_address_sk          integer          not null,
    ca_address_id         char(16)         not null,
    ca_street_number      char(10)         ,
    ca_street_name        varchar(60)      ,
    ca_street_type        char(15)        ,
    ca_suite_number       char(10)        ,
    ca_city               varchar(60)      ,
    ca_county             varchar(30)      ,
    ca_state              char(2)         ,
    ca_zip                char(10)        ,
    ca_country            varchar(20)      ,
    ca_gmt_offset         decimal(5,2)    ,
    ca_location_type      char(20)        ,
    PRIMARY KEY (ca_address_sk)
);

DECLARE
    i_id          integer := 1;
    address_id    char(16);
    street_number char(10);
    street_name   varchar(60) ;
    street_type   char(15);
    suite_number  char(10);
    city          varchar(60);
    county        varchar(30);
    state         char(2);
    zip           char(10);
    country       varchar(20);
    gmt_offset    decimal(5,2);
    location_type char(20);

BEGIN
    -- insert
    address_id := 'address_id';
    street_number := '3588' ;
    street_name := 'street_name';
    street_type := 'street_type';
    suite_number := '1';
    city := 'Hang Zhou';
    county := 'Bin Jiang';
    state := 'CN';
    zip := '000000';
    country := 'China';
    gmt_offset := 1.02;
    location_type := 'location_type';

    INSERT INTO customer_address(ca_address_sk, ca_address_id, ca_street_number,
    ca_street_name, ca_street_type, ca_suite_number, ca_city, ca_county, ca_state,
    ca_zip, ca_country, ca_gmt_offset, ca_location_type)

```

```
VALUES (i_id, address_id, street_number, street_name, street_type,
suite_number, city, county, state, zip, country, gmt_offset, location_type);
END;
/

CREATE OR REPLACE PROCEDURE getCustomerAddress(cur OUT sys_refcursor) IS
BEGIN
    OPEN cur FOR SELECT * FROM customer_address;
END;
/

DECLARE
    ad sys_refcursor;
    res customer_address%rowtype;
BEGIN
    getCustomerAddress(ad);

    LOOP
        FETCH ad INTO res;
        EXIT WHEN ad%NOTFOUND;
        dbms_output.put_line('ca_address_sk:' || res.ca_address_sk);
        dbms_output.put_line('ca_address_id:' || res.ca_address_id);
        dbms_output.put_line('ca_street_number:' || res.ca_street_number);
        dbms_output.put_line('ca_street_name:' || res.ca_street_name);
        dbms_output.put_line('ca_street_type:' || res.ca_street_type);
        dbms_output.put_line('ca_suite_number:' || res.ca_suite_number);
        dbms_output.put_line('ca_city:' || res.ca_city);
        dbms_output.put_line('ca_county:' || res.ca_county);
        dbms_output.put_line('ca_state:' || res.ca_state);
        dbms_output.put_line('ca_zip:' || res.ca_zip);
        dbms_output.put_line('ca_country:' || res.ca_country);
        dbms_output.put_line('ca_gmt_offset:' || res.ca_gmt_offset);
        dbms_output.put_line('ca_location_type:' || res.ca_location_type);
    END LOOP;
END;
/
```

#### 42.7.4.2. 打开和关闭游标变量

您可以使用 OPEN FOR 语句打开游标变量，该语句执行以下操作：

- 将游标变量与查询关联（通常情况下，查询返回多行）。 查询可以包含绑定变量的占位符，在 OPEN FOR 语句的 USING 子句中指定其值。
- 分配数据库资源以处理查询。
- 处理查询，即：
  - 识别结果集。 如果查询引用变量，则它们的值会影响结果集。
  - 如果查询具有 FOR UPDATE 子句，则锁定结果集中的行。
- 将游标定位在结果集的第一行之前。

#### 42.7.4.3. 使用游标变量提取数据

打开游标变量后，您可以使用 FETCH 语句获取查询结果集的行。 游标变量的返回类型必须与 FETCH 语句的 into\_clause 兼容。

第一个iOPEN FOR语句包含查询本身。第二个OPEN FORi语句引用一个变量，其值是一个查询。例如：

```

DECLARE
  cv SYS_REFCURSOR; -- cursor variable

  v_lastname employees.last_name%TYPE; -- variable for last_name
  v_jobid    employees.job_id%TYPE;    -- variable for job_id

  query_2 VARCHAR2(200) :=
    'SELECT * FROM employees
     WHERE REGEXP_LIKE (job_id, '' [ACADFIMKSA]_M[ANGR]'')
     ORDER BY job_id';

  v_employees employees%ROWTYPE; -- record variable row of table

BEGIN
  OPEN cv FOR
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
    ORDER BY last_name;

  LOOP -- Fetches 2 columns into variables
    FETCH cv INTO v_lastname, v_jobid;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (RPAD(v_lastname, 25, ' ') || v_jobid);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('-----');

  CLOSE CV;

  OPEN cv FOR query_2;

  LOOP -- Fetches entire row into the v_employees record
    FETCH cv INTO v_employees;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (RPAD(v_employees.last_name, 25, ' ') ||
v_employees.job_id);
  END LOOP;

  CLOSE cv;
END;
/

```

#### 42.7.4.4. 给光标变量赋值

您可以将一个PL/oraSQL光标变量的值赋给另一个PL/oraSQL光标变量或主机光标变量。

```
target_cursor_variable := source_cursor_variable;
```

如果source\_cursor\_variable是打开的，在赋值后，target\_cursor\_variable也将是打开的。这两个光标变量指向相同的SQL工作区。

如果source\_cursor\_variable没有打开，在赋值后打开target\_cursor\_variable并不会打开source\_cursor\_variable。

从光标变量中提取数据到集合中。例如：

```

DECLARE
  TYPE empcurtyp IS REF CURSOR;
  emp_cv empcurtyp;
  emp_cf empcurtyp;
  item employees%rowtype;
BEGIN
  OPEN emp_cv FOR SELECT * FROM employees WHERE job_id = 'SA_REP' ORDER BY
  salary DESC;
  OPEN emp_cf FOR SELECT * FROM employees;
  emp_cf := emp_cv;
  LOOP
    FETCH emp_cf INTO item;
    DBMS_OUTPUT.PUT_LINE ('Name = ' || item.last_name || ', salary = ' ||
  item.salary);
    EXIT WHEN emp_cf%notfound;
  END LOOP;
  CLOSE emp_cf;
END;
/

```

#### 42.7.4.5. 光标变量查询中的变量

与光标变量相关联的查询可以引用其作用域内的任何变量。

当您使用OPEN FOR语句打开光标变量时，PL/oraSQL将评估查询中的任何变量，并在识别结果集时使用这些值。稍后更改变量的值不会更改结果集。

从光标变量中提取数据到集合中。例如：

```

DECLARE
  sal          employees.salary%TYPE;
  sal_multiple employees.salary%TYPE;
  factor       INTEGER := 2;

  cv SYS_REFCURSOR;

BEGIN
  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%'; -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('factor = ' || factor);
    DBMS_OUTPUT.PUT_LINE ('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE ('sal_multiple = ' || sal_multiple);
    factor := factor + 1; -- Does not affect sal_multiple
  END LOOP;

  CLOSE cv;
END;
/

```



从光标变量中提取数据到集合中。例如：

```

DECLARE
    sal            employees.salary%TYPE;
    sal_multiple  employees.salary%TYPE;
    factor        INTEGER := 2;

    cv SYS_REFCURSOR;

BEGIN
    DBMS_OUTPUT.PUT_LINE (' factor = ' || factor);

    OPEN cv FOR
        SELECT salary, salary*factor
        FROM employees
        WHERE job_id LIKE 'AD_%'; -- PL/SQL evaluates factor

    LOOP
        FETCH cv INTO sal, sal_multiple;
        EXIT WHEN cv%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (' sal            = ' || sal);
        DBMS_OUTPUT.PUT_LINE (' sal_multiple = ' || sal_multiple);
    END LOOP;

    factor := factor + 1;

    DBMS_OUTPUT.PUT_LINE (' factor = ' || factor);

    CLOSE cv;

    OPEN cv FOR
        SELECT salary, salary*factor
        FROM employees
        WHERE job_id LIKE 'AD_%'; -- PL/SQL evaluates factor

    LOOP
        FETCH cv INTO sal, sal_multiple;
        EXIT WHEN cv%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (' sal            = ' || sal);
        DBMS_OUTPUT.PUT_LINE (' sal_multiple = ' || sal_multiple);
    END LOOP;

    CLOSE cv;
END;
/

```

#### 42.7.4.6. 光标变量属性

光标变量具有与显式光标相同的属性（请参见显式光标属性）。光标变量属性的值的语法是光标变量名称后紧跟属性（例如，cv%ISOPEN）。如果光标变量没有打开，则引用%ISOPEN以外的任何属性都会引发预定义的INVALID\_CURSOR异常。

## 42.8. 集合

在集合中，内部组件始终具有相同的数据类型，并称为元素。您可以通过索引访问集合变量的每个元素，语法为：variable\_name(index)。要创建集合变量，您可以定义集合类型，然后创建该类型的变量，或者使用%TYPE。

在PL/oraSQL块中定义的集合类型是本地类型。它仅在块中可用，并且仅在块位于独立的或包子程序中时存储在数据库中。

在模式级别定义的集合类型是独立类型。您可以使用“CREATE TYPE语句”创建它。它存储在数据库中，直到您使用“DROP TYPE语句”将其删除。

在包规范中定义的集合类型是公共项。您可以通过使用包名称（package\_name.type\_name）来限定它从包外引用。它存储在数据库中，直到您删除包。

## 42.8.1. 关联数组

关联数组是一组键值对。每个键是唯一的索引，用于使用语法variable\_name(index)定位关联值。

索引的数据类型可以是字符串类型（VARCHAR）或INTEGER。索引按排序顺序存储，而不是创建顺序。

例如，定义了一个按字符串索引的关联数组类型，声明了该类型的变量，将变量填充到三个元素，更改了一个元素的值，并打印了这些值（按排序顺序，而不是创建顺序）。

```

DECLARE
  -- Associative array indexed by integer:
  TYPE population IS TABLE OF varchar -- Associative array type
    INDEX BY integer;                -- indexed by string

  city_population population;          -- Associative array variable
  i integer;

BEGIN
  -- Add elements (key-value pairs) to associative array:
  city_population('2') := 'Smallville';
  city_population('5') := 'Midland';
  city_population('3') := 'Megalopolis';

  FOR i IN 1..6 LOOP
    IF city_population.EXISTS(i) THEN
      dbms_output.put_line ('city_population(' || i || ') ' || ' exists');
    ELSE
      dbms_output.put_line ('city_population(' || i || ') ' || ' does not
exist');
    END IF;
  END LOOP;

END;
/

```

## 42.8.2. 嵌套表

在数据库中，嵌套表是一种列类型，它以没有特定顺序的方式存储未指定数量的行。

当您将数据库中的嵌套表值检索到PL/oraSQL嵌套表变量中时，PL/oraSQL会为行分配连续的索引，从1开始。使用这些索引，您可以访问嵌套表变量的各个行。语法是variable\_name(index)。随着您将嵌套表从数据库中存储和检索，嵌套表的索引和行顺序可能不会保持稳定。

随着您添加或删除元素，嵌套表变量占用的内存量可以动态增加或减少。

示例：本地类型的嵌套表

```

DECLARE
TYPE Roster IS TABLE OF VARCHAR(15); -- nested table type

-- nested table variable initialized with constructor:
names2 Roster := Roster('D Caruso', 'J Hamil', 'D Piro', 'R Singh');

BEGIN
  DBMS_OUTPUT.PUT_LINE('Initial Values:');
  FOR i IN names2.FIRST .. names2.LAST LOOP -- For first to last element
    DBMS_OUTPUT.PUT_LINE(names2(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('-----');
  names2(5) := 'P Perez'; -- add one element
  FOR i IN names2.FIRST .. names2.LAST LOOP -- For first to last element
    DBMS_OUTPUT.PUT_LINE(names2(i));
  END LOOP;
END;
/

```

示例：独立类型的嵌套表

```

CREATE TYPE names2 IS TABLE OF VARCHAR(15);

DECLARE
  nt names2 ;
BEGIN
  nt(1) = 'D Caruso';
  nt(2) = 'J Hamil';
  nt(3) = 'R Singh';
  DBMS_OUTPUT.PUT_LINE(nt(2));
END;
/

```

### 42.8.3. 可变数组

可变大小数组是一个数组，其元素数量可以从零（空）到声明的最大大小不等，在这里超过也支持。

要访问可变数组的元素，使用语法variable\_name(index)。在数据库中存储和检索可变数组时，其索引和元素顺序将保持稳定。

随着您添加或删除元素，可变数组变量占用的内存量可以动态增加或减少。

目前只支持本地类型的可变数组，不支持全局可变数组。

示例：本地类型的可变数组

```

DECLARE
  TYPE Roster IS VARRAY(10) OF VARCHAR(15); -- varray type

  -- varray variable initialized with constructor:
  names2 Roster := Roster('D Caruso', 'J Hamil', 'D Piro', 'R Singh');

BEGIN
  DBMS_OUTPUT.PUT_LINE('Initial Values:');
  FOR i IN names2.FIRST .. names2.LAST LOOP -- For first to last element

```

```

        DBMS_OUTPUT.PUT_LINE(names2(i));
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('-----');
    names2.extend;
    names2(5) := 'P Perez'; -- add one element
    FOR i IN names2.FIRST .. names2.LAST LOOP -- For first to last element
        DBMS_OUTPUT.PUT_LINE(names2(i));
    END LOOP;
END;
/

```

## 42.8.4. 集合方法

集合方法是一个PL/oraSQL子程序，可以是返回有关集合信息的函数，也可以是操作集合的过程。集合方法使集合易于使用，使您的应用程序易于维护。

Table 42. 显示了集合方法。

Table 42.1. 集合方法

方法	描述
DELETE	从集合中删除元素。
EXTEND	将元素添加到嵌套表的末尾。
EXISTS	如果嵌套表的指定元素存在，则返回TRUE。
FIRST	返回集合中的第一个索引。
LAST	返回集合中的最后一个索引。
COUNT	返回集合中的元素数量。

集合方法调用的基本语法如下：

```
collection_name.method
```

## 42.9. 事务管理

在由CALL命令调用的过程中，以及在匿名代码块中，可以使用COMMIT和ROLLBACK命令结束事务。使用这些命令结束事务后，会自动启动一个新事务，因此没有单独的START TRANSACTION命令。（注意，PL/oraSQL中BEGIN和END具有不同的含义。）

以下是一个简单的示例：

```

CREATE OR REPLACE PROCEDURE transaction_test1(b int) IS
BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i < b THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END;
/

CALL transaction_test1(3);

```

## 42.9.1. 使用事务操作游标

本文主要介绍了游标操作和事务操作（提交和回滚）的组合。主要分为以下五类：

- 功能
  - 在函数中调用函数：不支持。
  - 函数调用过程：不支持。
  - 函数包中的函数和存储过程：
    - 传递函数：不支持；
    - 包调整过程：不支持。
- 存储过程
  - 在存储过程中调用函数：不支持。
  - 存储过程调用过程：支持。
  - 在存储过程中调用包中的函数和存储过程：
    - 传递函数：不支持；
    - 包调整过程：不支持。
- 匿名块
  - 在匿名块中调用函数：不支持。
  - 匿名块中间调整过程：支持。
  - 在匿名块中调用包中的函数和存储过程：
    - 传递函数：不支持；
    - 包调整过程：不支持。
- 包中的函数
  - 包函数调用函数：不支持。
  - 包函数调用存储过程：不支持。
  - 包函数调用包中的函数和存储过程：
    - 传递函数：不支持；
    - 包调整过程：不支持。
- 包中的存储过程
  - 包存储过程调用者：不支持。
  - 包存储过程调用存储过程：支持。
  - 包存储过程传递函数和包中的存储过程：
    - 传递函数：不支持；
    - 包调整过程：不支持。

## Tip

要使用游标操作和事务操作（提交和回滚），并且 `lightdb_dblevel_syntax_compatible_type` 的值为 `oracle`，需要将 `search_path` 设置为以下内容： `SET search_path TO public, oracle;`

## Tip

在匿名块中不支持与存储过程相同的方式调用函数，例如：

```
create or replace function func return int as
begin
  dbms_output.put_line ('This is just a test');
end;
/

begin
  func();
  dbms_output.put_line ('This is just a test');
end;
/
```

## Tip

游标声明和定义的包说明：

- 在声明包时，不能直接在包中声明游标；
- 在定义包体时，不能在包体的全局区域直接定义游标变量。

## Tip

目前无法同时使用存储过程和插件 `auto_explain`。

示例：调用存储过程

```
--preconditional
-- the value of lightdb_dblevel_syntax_compatible_type must be oracle
create extension IF NOT EXISTS orafce;
SET search_path TO public, oracle;

create table ltbench_accounts(id int, abalance int);

--insert
declare
  irevert int := 9999;
  icount int := 1000;
begin
  loop
```

```
        execute immediate 'insert into ltbench_accounts values(:er,:rt)' using
irevert, icount;
        icount := icount - 1;
        irevert := irevert -1;
        exit when icount < 1;
    end loop;
end;
/

--commit、rollback
create or replace procedure tt_try_proc as
    CURSOR c1 IS
        SELECT * FROM ltbench_accounts;

    CURSOR c2 IS
        SELECT * FROM ltbench_accounts;

    emp_rec ltbench_accounts%ROWTYPE;
BEGIN
    OPEN c2;
    OPEN c1;
    UPDATE ltbench_accounts SET id = 25 WHERE abalance = 12;
    rollback;
    UPDATE ltbench_accounts SET id = 17 WHERE abalance = 13;
    commit;
    LOOP
        FETCH c1 INTO emp_rec;
        EXIT WHEN c1%NOTFOUND;
        dbms_output.put_line ('emp_rec.id = ' || emp_rec.id || ' emp_rec.abalance = '
|| emp_rec.abalance);
        UPDATE ltbench_accounts SET id = 23 WHERE abalance = 12;
        rollback;
    END LOOP;
    LOOP
        FETCH c2 INTO emp_rec;
        EXIT WHEN c2%NOTFOUND;
        dbms_output.put_line ('emp_rec.id = ' || emp_rec.id || ' emp_rec.abalance = '
|| emp_rec.abalance);
        UPDATE ltbench_accounts SET id = 18 WHERE abalance = 13;
        commit;
    END LOOP;
    commit;
    rollback;
END;
/

--call procedure in procedure
create or replace procedure tt_try_dproc as
begin
    tt_try_proc();
    dbms_output.put_line ('This is just a test');
end;
/

call tt_try_dproc();
```

示例：在存储过程中调用存储在包中的存储过程

```
--preconditional
create table test(x int, y varchar(100));
insert into test values(1, 'One');
insert into test values(2, 'Two');
insert into test values(3, 'Three');

--create package
create or replace package pkg is
  CURSOR c1 IS SELECT x,y FROM test;
  function c_open return int;
  function c_fetch return int;
  procedure c_open_fetch;
end;
/

--create package boby
create or replace package body pkg IS
  function c_open return int as
  begin
    OPEN c1;
    return 0;
  end;

--The fetch in the package, the cursor will not be cleared
  function c_fetch return int AS
  v_x test.x%TYPE;
  v_y test.y%TYPE;
  begin
    LOOP
      FETCH c1 INTO v_x, v_y;
      EXIT WHEN c1%NOTFOUND;
      dbms_output.put_line( v_x || '.' || v_y);
    END LOOP;
    --CLOSE c1;
    return 0;
  end;

  procedure c_open_fetch as
  v_x test.x%TYPE;
  v_y test.y%TYPE;
  BEGIN
    OPEN c1;
    COMMIT;
    LOOP
      FETCH c1 INTO v_x, v_y;
      EXIT WHEN c1%NOTFOUND;
      dbms_output.put_line( v_x || '.' || v_y);
      IF c1%FOUND THEN
        UPDATE test SET x = 4 WHERE y = 'Two';
        ROLLBACK;
      END IF;
    END LOOP;
    CLOSE c1;
  end;

begin
```



```
    dbms_output.put_line('initializer');
    UPDATE test SET x = 5 WHERE y = 'Two';
    COMMIT;
end;
/

--procedure of package called by procedure
create or replace procedure tt_try_proc as
begin
    pkg.c_open_fetch();
    dbms_output.put_line ('This is just a test');
end;
/

call tt_try_proc();
```

示例：匿名块中间调优过程

```
begin
    tt_try_proc();
end;
/
```

示例：在匿名块中调用包中的存储过程

```
--procedure of package called by anonymous block
begin
    pkg.c_open_fetch();
    dbms_output.put_line ('This is just a test');
end;
/
```

示例：包存储过程调用存储过程

```
--create package
create or replace package pkg_proc is
    CURSOR c1 IS SELECT x,y FROM test;
    procedure c_proc;
end;
/

--create package body
create or replace package body pkg_proc IS
    procedure c_proc as
        v_x test.x%TYPE;
        v_y test.y%TYPE;
    BEGIN
        OPEN c1;
        COMMIT;
        LOOP
```

```

        tt_try_proc();
        FETCH c1 INTO v_x, v_y;
        EXIT WHEN c1%NOTFOUND;
        dbms_output.put_line( v_x || '.' || v_y);
        IF c1%FOUND THEN
            UPDATE test SET x = 4 WHERE y = 'Two';
            ROLLBACK;
        END IF;
    END LOOP;
    CLOSE c1;
end;

begin
    dbms_output.put_line('initializer');
    UPDATE test SET x = 5 WHERE y = 'Two';
    COMMIT;
end;
/

call pkg_proc.c_proc();

```

示例：包存储过程在包中传递存储过程

```

--procedure of package called by procedure of package
--create package of pkg_proc
create or replace package pkg_proc is
    CURSOR c1 IS SELECT x,y FROM test;
    procedure c_proc;
end;
/

--create package body of pkg_proc
create or replace package body pkg_proc IS
    procedure c_proc as
        v_x test.x%TYPE;
        v_y test.y%TYPE;
    BEGIN
        OPEN c1;
        COMMIT;
        LOOP
            FETCH c1 INTO v_x, v_y;
            EXIT WHEN c1%NOTFOUND;
            dbms_output.put_line( v_x || '.' || v_y);
            IF c1%FOUND THEN
                UPDATE test SET x = 4 WHERE y = 'Two';
                ROLLBACK;
            END IF;
        END LOOP;
        CLOSE c1;
    end;

begin
    dbms_output.put_line('initializer');

```

```
UPDATE test SET x = 5 WHERE y = 'Two';
COMMIT;
end;
/

--create package of pkg_call_proc
create or replace package pkg_call_proc is
  CURSOR c1 IS SELECT x,y FROM test;
  procedure c_call_proc;
end;
/

--create package body of pkg_call_proc
create or replace package body pkg_call_proc IS
  procedure c_call_proc as
    v_x test.x%TYPE;
    v_y test.y%TYPE;
  BEGIN
    OPEN c1;
    COMMIT;
    LOOP
      pkg_proc.c_proc();
      FETCH c1 INTO v_x, v_y;
      EXIT WHEN c1%NOTFOUND;
      dbms_output.put_line( v_x || '.' || v_y);
      IF c1%FOUND THEN
        UPDATE test SET x = 4 WHERE y = 'Two';
        ROLLBACK;
      END IF;
    END LOOP;
    CLOSE c1;
  end;

begin
  dbms_output.put_line('initializer');
  UPDATE test SET x = 5 WHERE y = 'Two';
  COMMIT;
end;
/

call pkg_call_proc.c_call_proc();
```

示例：声明包时，不能直接在包中声明游标

```
--Declare cursors in packages using sys_refcursor
--error
create or replace package pkg is
  c1 sys_refcursor;
  function c_open return int;
  function c_fetch return int;
  procedure c_open_fetch;
end;
/
```

示例：在定义包体时，不能直接在包体的全局区域定义游标变量

```
--在包体内使用sys_refcursor声明游标变量
create or replace package pkg is
  function c_open return int;
  function c_fetch return int;
  procedure c_open_fetch;
end;
/
```

--错误示例

```
create or replace package body pkg IS
  c1 sys_refcursor;
  function c_open return int as
  begin
    OPEN c1 for SELECT x,y FROM test;
    return 0;
  end;

  function c_fetch return int AS
  v_x test.x%TYPE;
  v_y test.y%TYPE;
  begin
    LOOP
      FETCH c1 INTO v_x, v_y;
      EXIT WHEN c1%NOTFOUND;
      dbms_output.put_line( v_x || '.' || v_y);
    END LOOP;
    --CLOSE c1;
    return 0;
  end;

  procedure c_open_fetch as
  v_x test.x%TYPE;
  v_y test.y%TYPE;
  BEGIN
    OPEN c1 for SELECT x,y FROM test;
    COMMIT;
    LOOP
      FETCH c1 INTO v_x, v_y;
      EXIT WHEN c1%NOTFOUND;
      dbms_output.put_line( v_x || '.' || v_y);
      IF c1%FOUND THEN
        UPDATE test SET x = 4 WHERE y = 'Two';
        ROLLBACK;
      END IF;
    END LOOP;
    CLOSE c1;
  end;

  begin
    dbms_output.put_line('initializer');
    UPDATE test SET x = 5 WHERE y = 'Two';
    COMMIT;
  end;
```

/

## 42.9.2. 自治事务

AUTONOMOUS\_TRANSACTION语法改变事务中子程序的执行方式。通过此语法，子程序的提交，撤销可以独立于主事务。

此功能从版本23.2开始支持，在Oracle模式下，此语法可应用于函数和事务，当前仅支持语法兼容，无实际功能。

语法

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

示例如下：

```
CREATE OR REPLACE PROCEDURE example_pragma_autonomous_transaction_procedure()
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  NULL;
END;
/
```

```
CREATE OR REPLACE FUNCTION example_pragma_autonomous_transaction_func()
RETURN void
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  NULL;
END;
/
```

## 42.10. 软件包

本节介绍 LightDB 的“Oracle 风格包”（Oracle Style Package）。按照定义，一个包是一组打包在一起的对象或者一组对象。在数据库领域，这意味着一个命名的模式对象，其中包含逻辑上分组的一组过程、函数、变量、游标、用户定义的记录类型和引用记录。

### 42.10.1. 包概述

#### 42.10.1.1. 包的必要性

与其他编程语言中类似的结构一样，使用 SQL 包有很多好处。在本节中，我们将涵盖其中的一些。

- 可靠性和可重用性 – 包提供了创建封装代码的模块化对象的能力。这使得整体设计和实现变得更简单。封装允许通过公开包接口而不是其实现细节即包体来实现，从而使得模块更加简单易懂、易于维护和使用。此外，封装还允许应用程序和用户引用一致的接口，而不用担心其包含的内容。它还可以防止用户基于代码实现做出任何决策，因为它们永远不会暴露给用户。

- 易用性 - 在 LightDB 中创建一致的函数接口可以帮助简化应用程序开发，因为可以在不包括其正文的情况下编译包。在开发阶段之外，包允许用户管理整个包的访问控制，而不是单个对象。如果包含许多模式对象，则这是非常有价值的。
- 性能 - 包被加载到内存中进行维护，因此利用最少的 I/O 资源。重新编译仅限于更改的对象，而依赖对象不会重新编译。
- 附加功能 - 除了性能和易用性之外，包还为变量和游标提供了会话级持久性。这意味着变量和游标的生命周期与数据库会话相同，并在会话结束时被销毁。

## 42.10.1.2. 包组成部分

包由两个组成部分组成：包规范和包体。

包规范 - 包中要从外部使用的任何对象都在包规范部分中指定。这是我们在早期部分中提到的公共可访问接口。它不包含它们的定义或实现，即函数和过程。它只定义了它们的头部而没有定义正文。变量可以初始化。下面是可以列在包规范中的对象列表：

- 函数
- 过程
- 游标
- 类型
- 变量
- 记录类型
- 关联数组
- 嵌套表

包体 - 包体包含包的所有实现代码，包括公共接口和私有对象。如果规范不包含任何子程序或游标，则包体是可选的。

包体必须包含规范中声明的子程序的定义，并且相应的定义必须匹配。

包体可以包含其自己的子程序和类型声明，以及任何未在规范中指定的内部对象。然后，这些对象被视为私有对象。私有对象无法从包外部访问。

除了子程序定义之外，它还可以可选地包含一个初始化块，用于初始化规范中声明的变量，仅在会话中第一次调用包时执行一次。

## 42.10.2. 创建包

### 42.10.2.1. 创建包

CREATE PACKAGE —— 定义一个新的包规范。

语法

```
CREATE [ OR REPLACE ] PACKAGE [schema.]package_name [IS | AS]
    item_list[, item_list ...]
END [package_name];
```

```

item_list:
[
  function_declaration |
  procedure_declaration |
  type_definition |
  cursor_declaration |
  item_declaration
]

function_declaration:
  FUNCTION function_name [(parameter_declaration[, ...])] RETURN datatype;

procedure_declaration:
  PROCEDURE procedure_name [(parameter_declaration[, ...])]

type_definition:
  record_type_definition |
  ref_cursor_type_definition

cursor_declaration:
  CURSOR name [(cur_param_decl[, ...])] RETURN rowtype;

item_declaration:
  cursor_declaration |
  cursor_variable_declaration |
  record_variable_declaration |
  variable_declaration

record_type_definition:
  TYPE record_type IS RECORD ( variable_declaration [,
  variable_declaration]... );

ref_cursor_type_definition:
  TYPE type IS REF CURSOR [ RETURN type%ROWTYPE ];

cursor_variable_declaration:
  curvar curtype;

record_variable_declaration:
  recvar { record_type | rowtype_attribute | record_type%TYPE };

variable_declaration:
  varname datatype [ [ NOT NULL ] := expr ]

parameter_declaration:
  parameter_name [IN] datatype [[:= | DEFAULT] expr]

```

### 描述

CREATE PACKAGE用于创建包规范，其中包含公共声明。包规范中声明的项可以从包中的任何位置以及同一数据库中的任何其他子程序中访问。

CREATE PACKAGE用于定义一个新的包。CREATE OR REPLACE PACKAGE既可以创建一个新的包，也可以替换现有的定义。

如果包含模式名称，则在指定的模式中创建包。否则，它将在当前模式中创建。新包的名称在模式内必须是唯一的。

当使用CREATE OR REPLACE PACKAGE替换现有包时，包的所有权和权限不会更改。所有其他包属性都分配了指定或隐含在命令中的值。只有所有者和拥有角色的成员才允许替换包。

用户创建包后成为包的所有者。

## 42.10.2.2. 创建包体

CREATE PACKAGE BODY —— 定义一个新的包定义。

语法

```
CREATE [ OR REPLACE ] PACKAGE BODY [schema.]package_name [IS | AS]
  [item_list[, item_list ...]] |
  item_list_2 [, item_list_2 ...]
  [initialize_section]
END [package_name];

initialize_section:
  BEGIN statement[, ...]

item_list:
[
  function_declaration |
  procedure_declaration |
  type_definition |
  cursor_declaration |
  item_declaration
]

item_list_2:
[
  function_declaration
  function_definition
  procedure_declaration
  procedure_definition
  cursor_definition
]

function_definition:
  FUNCTION function_name [(parameter_declaration[, ...])] RETURN datatype [IS
| AS]
  [declare_section] body;

procedure_definition:
  PROCEDURE procedure_name [(parameter_declaration[, ...])] [IS | AS]
  [declare_section] body;

cursor_definition:
  CURSOR name [(cur_param_decl[, ...])] RETURN rowtype IS select_statement;

body:
  BEGIN statement[, ...] END [name];

statement:
  [LABEL] pl_statments[, ...];
```



## 描述

CREATE PACKAGE BODY 用于定义包的包体。CREATE OR REPLACE PACKAGE BODY 可以创建新的包体，或替换现有的包体定义。要创建包体，必须先创建包规范。包体包含通过 CREATE PACKAGE 创建的包规范中声明的每个游标和子程序的实现。仅当包规范中列出它们的规范时，包体中定义的对象才可以被外部访问。对于仅在包体中定义而未包含在包规范中的所有对象，它们成为包的私有成员，并且无法在包外部访问。包和其包体必须在同一架构中创建。

### 42.10.2.3. 参数

package\_name 是要创建的包的名称（可选模式限定符）。

item\_list 是包含在包中的项目列表。

procedure\_declaration 表示过程签名，即 procedure\_name(argument\_list)。procedure\_declaration 可以出现在包规范和包体中。在包规范中列出的过程声明使该过程成为公共过程，并且可以从包外部访问。而在包体中声明的过程被视为前向声明，并成为包的私有成员。

procedure\_definition 是包过程的具体实现/定义。procedure\_definition 只能在包体中提供。过程访问限定符由过程声明确定，而在包体中定义的过程，如果没有相应的声明，则自动成为包的私有成员。

function\_declaration 表示函数签名及其返回类型，即 function\_name(argument\_list) RETURN return\_type;。

function\_declaration 可以出现在包规范和包体中。在包规范中列出的函数声明使该函数成为公共函数，并且可以从包外部访问。而在包体中声明的函数被视为前向声明，并成为包的私有成员。

function\_definition 是包函数的具体实现/定义。function\_definition 只能在包体中提供。函数访问限定符由函数声明确定，而在包体中定义的函数，如果没有相应的声明，则自动成为包的私有成员。

type\_definition 可以是 RECORD 或 CURSOR 类型定义。

cursor\_declaration 表示带有参数和返回类型为所需 ROWTYPE 的 CURSOR 声明。

item\_declaration 允许声明：游标、游标变量、记录变量、变量。

parameter\_declaration 定义声明参数的语法。如果指定了关键字“IN”，则表示这是一个输入参数。对于输入参数，DEFAULT 关键字后跟表达式（或值）。

declare\_section 包含所有对于函数或过程局部的元素，可以在其体内被引用。

body 由 plorasql 语言支持的 SQL 语句或 PL 控制结构组成。

## 42.10.3. 创建和访问包

### 42.10.3.1. 创建包

在之前的章节中，我们已经了解了规定包结构的语法。在本章中，我们将进一步了解包的构造过程以及如何访问其公共元素。

包创建后，LightDB 会编译它并报告任何可能发现的问题。一旦包成功编译，它就可以用于操作了。

### 42.10.3.2. 访问包元素

在会话中首次引用包时，将实例化和初始化该包。在此过程中按相同顺序执行以下操作：

- 给公共变量分配初始值
- 执行包的初始化程序块

有几种方法可以访问包元素：

- 包函数可以像其他函数一样在 SELECT 语句或其他 PL 块中使用
- 可以直接使用 CALL 调用包过程或从其他 PL 块调用
- 可以在 PL 块中使用包名称限定符直接读取和写入包变量。
- 可以使用点符号表示法进行直接访问：可以按以下方式访问元素：

- `package_name.func('foo');`

- `package_name.proc('foo');`

- `package_name.variable;`

如果元素是函数，则可以从 PL 块内或 SELECT 语句中使用这些语句。

- SQL CALL 语句：另一种方法是使用 CALL 语句。CALL 语句在类型或包中执行独立过程。

- `CALL package_name.proc('foo');`

### 42.10.3.3. 了解可见性范围

PL/oraSQL 块中声明的变量的作用域仅限于该块。如果它有嵌套块，则对于嵌套块来说，它将是一个全局变量。

同样地，如果两个块声明了相同名称的变量，那么在嵌套块中，其自己声明的变量可见，父级的变量则不可见。要访问父级变量，必须完全限定该变量。

请看以下代码片段。

示例：可见性和限定变量名称

```
-- blk_1
DECLARE
  x INT;
  y INT;
BEGIN
  -- both blk_1.x and blk_1.y are visible
  -- blk_2
  DECLARE
    x INT;
    z INT;
  BEGIN
    -- blk_2.x, y and z are visible
    -- to access blk_1.x it has to be a qualified name. blk_1.x := 0; NULL;
    NULL;
  END;
  -- both x and y are visible
END;
/
```

上面的示例展示了当嵌套包含一个与变量同名的变量时，必须完全限定变量名的方法。

变量名限定有助于解决以下情况中引入的可能混淆：

- 包和嵌套包变量：未经限定，嵌套包优先。
- 包变量和列名称：未经限定，列名称优先。
- 函数或过程变量和包变量：未经限定，包变量优先。

## 42.10.4. 示例

创建一个名为 example 的包。

```
CREATE TABLE test(x INT, y VARCHAR2(100));
create table rectype(a int,b varchar2(100));
INSERT INTO test VALUES (1, 'One');
INSERT INTO test VALUES (2, 'Two');
INSERT INTO test VALUES (3, 'Three');
```

```
CREATE OR REPLACE PACKAGE example
AS
  -- 声明类型和游标:
  CURSOR curtype RETURN rectype%rowtype;

  rec rectype%rowtype;

  -- 声明子程序:
  FUNCTION somefunc (v int) RETURN NUMBER;

  -- 重载前面的子程序:
  PROCEDURE xfunc (emp_id NUMBER);
  PROCEDURE xfunc (emp_email VARCHAR2);
END;
/
```

创建一个名为 example 的包体。

```
CREATE OR REPLACE PACKAGE BODY example
AS
  nelems NUMBER; -- 这个包中的变量

  -- 定义在包规范中声明的游标:
  CURSOR curtype RETURN rectype%rowtype IS SELECT x, y FROM test ORDER BY x;
  -- 定义在包规范中声明的子程序:
  FUNCTION somefunc (v int) RETURN NUMBER
  IS
    id NUMBER := 0;
  BEGIN
    OPEN curtype;
    FETCH curtype INTO rec;
    RETURN rec.a;
  END;

  PROCEDURE xfunc (emp_id NUMBER) IS
  BEGIN
    NULL;
  END;

  PROCEDURE xfunc (emp_email VARCHAR2) IS
  BEGIN
```

```

        NULL;
    END;

END;
/

```

## 42.10.5. 限制

记录类型被支持作为包变量，但是它们只能在包元素内使用，即包函数/过程可以利用它们。它们不能在包外部访问。

不建议将包类型用于建表和函数/过程语句（在包外部），因为这可能会导致一些问题。

同一 schema 下不能存在相同名字的表和包。与 oracle 相同。

```

lightdb@postgres=# create table t1(key1 int);
CREATE TABLE
lightdb@postgres=# create package t1 IS
        key1 real;
        end;
/
ERROR:  relation "t1" already exists
HINT:  A relation has the same name, so you must use a name that doesn't
conflict with any existing relation.
lightdb@postgres=# drop table t1;
DROP TABLE
lightdb@postgres=# create package t1 IS
        key1 real;
        end;
/
CREATE PACKAGE
lightdb@postgres=# create table t1(key1 int);
ERROR:  package "t1" already exists
HINT:  A package has the same name, so you must use a name that doesn't conflict
with any existing package.
lightdb@postgres=#

```

schema 和 table 不能有相同的名字。

```

lightdb@postgres=# create package t1 IS
        key1 real;
        end;
/
CREATE PACKAGE
lightdb@postgres=# create schema t1;
ERROR:  package name "t1" exists, schema name is forbid same with package name.
lightdb@postgres=# drop package t1;
DROP PACKAGE
lightdb@postgres=# create schema t1;
CREATE SCHEMA
lightdb@postgres=# create package t1 IS
        key1 real;
        end;
/
ERROR:  schema name "t1" exists, package name is forbid same with schema name.

```

```
lightdb@postgres=#
```

## 42.11. 表函数

表函数可以将数组数据进行行转列操作，参数类型可以是数组或者嵌套表，或者是函数返回值(数组/嵌套表)。也可以查询语句中使用。为了兼容Oracle数据库的table函数的功能，当from子句中使用table函数，并且table函数返回的数据是单列时，将列名修改为column\_value。这个功能在Oracle兼容模式下生效。

数组例子:

```
select table(array[1,2,3]) as a;
```

嵌套表例子:

```
create type kk is table of int;
select table(kk(1,2,3)) as a;
```

函数返回值的例子:

```
create table t1(a int, b int);
insert into t1 (values (1,1), (2,2), (3,3));
```

```
create or replace package pkg
as
  type nt is table of t1%rowtype;
  res nt;
  function myfunc() return nt;
end;
/
```

```
create or replace package body pkg
as
  function myfunc() return nt
  is
  begin
    res(1) := ROW(1,1);
    res(2) := ROW(2,2);
    res(3) := ROW(3,3);
    return res;
  end;
end;
/
```

```
select * from table(pkg.myfunc());
```

查询语句使用例子:

```
create type kk is table of int;
create table t100 (a int, b kk);
insert into t100 values(1, kk(1, 2));
insert into t100 values(2, kk(3, 4, 5));
select a, b_table from t100, table(b) b_table;
```

## 42.12. 错误处理

本章讲解如何处理 PLoraSQL 编译时警告和 PLoraSQL 运行时错误。后者被称为异常。

### 42.12.1. 异常处理概述

异常（PLoraSQL 运行时错误）可能由于设计缺陷、编码错误、硬件故障等多种原因而引起。您无法预见所有可能的异常，但可以编写异常处理程序，使您的程序能够在出现异常时继续运行。

任何 PLoraSQL 块都可以有异常处理部分，它可以有一个或多个异常处理程序。例如，异常处理部分可以具有以下语法：

```
EXCEPTION
  WHEN ex_name_1 THEN statements_1           -- 异常处理程序
  WHEN ex_name_2 OR ex_name_3 THEN statements_2 -- 异常处理程序
  WHEN OTHERS THEN statements_3             -- 异常处理程序
END;
```

当块的可执行部分中引发异常时，可执行部分停止并将控件转移到异常处理部分。如果引发 `ex_name_1`，则运行 `statements_1`。如果引发 `ex_name_2` 或 `ex_name_3`，则运行 `statements_2`。如果引发其他异常，则运行 `statements_3`。

在异常处理程序运行后，控制转移到封闭块的下一个语句。如果没有封闭块，则：

- 如果异常处理程序在子程序中，则控制返回到调用者，在调用语句之后
- 如果异常处理程序在匿名块中，则控制转移到主机环境

#### 42.12.1.1. 异常类别

异常类别包括：

- 内部定义 运行时系统隐式（自动）引发内部定义的异常
- 预定义 预定义异常是内部定义的异常，由 PLoraSQL 赋予名称
- 用户定义 您可以在任何 PLoraSQL 匿名块、子程序或包的声明部分中声明自己的异常

#### 42.12.1.2. 异常处理程序的优点

使用异常处理程序进行错误处理使程序更易编写和理解，并减少未处理异常的可能性。如果没有异常处理程序，您必须在可能发生错误的每个地方检查每个可能的错误，然后处理它。特别是如果错误不能立即检测到，很容易忽略可能的错误或可能发生错误的地方。错误处理代码分散在整个程序中。有了异常处理程序，您不需要知道每个可能的错误或可能发生错误的地方。您只需要在可能发生错误的每个块中包含一个异常处理部分。在异常处理部分中，您可以包括针对特定和未知错误的异常处理程序。如果块中的任何位置（包括子块内部）发生错误，则异常处理程序将处理它。错误处理代码隔离在块的异常处理部分中。

目前 PLoraSQL 仅支持预定义错误。

### 42.12.2. 预定义异常

预定义异常是具有预定义名称的内部定义异常。运行时系统隐式（自动）引发预定义异常。由于预定义异常具有名称，因此可以专门为它们编写异常处理程序。

目前支持 7 种预定义错误：

- INVALID\_CURSOR
- CASE\_NOT\_FOUND
- DUP\_VAL\_ON\_INDEX
- NOT\_DATA\_FOUND
- TOO\_MANY\_ROWS
- ZERO\_DIVIDE
- LOGIN\_DENIED

#### 42.12.2.1. INVALID\_CURSOR（无效游标）

INVALID\_CURSOR 主要针对两种情况：显式游标和游标变量。

##### 42.12.2.1.1. 显式游标

- 使用 CLOSE 语句关闭打开的显式游标，从而允许重用其资源。关闭游标后，不能从其结果集中获取记录或引用其属性。如果尝试这样做，则 PLoraSQL 引发预定义的异常 INVALID\_CURSOR。
- 如果显式游标未打开，则除 %ISOPEN 之外的任何属性引用都会引发预定义的异常 INVALID\_CURSOR。

例如，关闭游标变量后，不能从其结果集中获取记录或引用其属性。

```
--preconditional
create table employees(id int, name varchar2(100), job varchar2(100));
insert into employees values (22091401, 'sunli', 'actress');
insert into employees values (22091402, 'dengchao', 'actor');
insert into employees values (22091403, 'lishizhen', 'doctor');
insert into employees values (22091404, 'zhangsanfeng', 'martial artist');

create or replace function invalid_cursor_f return int as
  cursor c1 is select * from employees;
  emp_cur employees%rowtype;
begin
  open c1;
  loop
    fetch c1 into emp_cur;
    exit when c1%notfound;
    close c1;
  end loop;
  close c1;
EXCEPTION
  WHEN invalid_cursor THEN
    DBMS_OUTPUT.PUT_LINE('invalid cursor');
    return 1;
end;
/

select invalid_cursor_f();
```

例如，如果显式游标未打开，则除 `%ISOPEN` 之外的任何属性引用都会引发预定义的异常 `INVALID_CURSOR`。

```
create or replace function invalid_cursor_f return int as
  cursor c1 is select * from employees;
  emp_cur employees%rowtype;
  ora_found boolean := false;
begin
  ora_found := c1%found;
EXCEPTION
  WHEN invalid_cursor THEN
    DBMS_OUTPUT.PUT_LINE('invalid cursor');
  return 1;
end;
/

select invalid_cursor_f();
```

### Tip

使用 PLoraSQL 进行错误处理，且 `lightdb_dblevel_syntax_compatible_type` 的值为 `oracle`。

#### 42.12.2.1.2. 游标变量

- 当你不再需要一个游标变量时，用 `CLOSE` 语句关闭它，从而允许其资源被重用。在关闭游标变量之后，你不能从其结果集中获取记录或引用其属性。如果你尝试这样做，PLoraSQL 会引发预定义的异常 `INVALID_CURSOR`。
- 游标变量具有与显式游标相同的属性。游标变量属性的值语法为 `cursor_variable_name` 紧接着 `attribute`（例如，`cv%ISOPEN`）。如果一个游标变量没有打开，引用除 `%ISOPEN` 以外的任何属性都会引发预定义的异常 `INVALID_CURSOR`。

例如，在关闭游标变量之后，你不能从其结果集中获取记录或引用其属性。

```
create or replace function invalid_cursor_f return int as
  c1 sys_refcursor;
  emp_cur employees%rowtype;
begin
  open c1 for select * from employees;
  loop
    fetch c1 into emp_cur;
    exit when c1%notfound;
    close c1;
  end loop;
  close c1;
EXCEPTION
  WHEN invalid_cursor THEN
    DBMS_OUTPUT.PUT_LINE('invalid cursor');
  return 1;
end;
/

select invalid_cursor_f();
```



例如，如果一个游标变量没有打开，引用除%ISOPEN以外的任何属性都会引发预定义的异常INVALID\_CURSOR。

```
create or replace function invalid_cursor_f return int as
  c1 sys_refcursor;
  emp_cur employees%rowtype;
  ora_isopen boolean := false;
begin
  ora_isopen := c1%isopen;
  return 1;
exception
  when INVALID_CURSOR then
    dbms_output.put_line('this is a normal error');
end;
/

select invalid_cursor_f();
```

#### 42.12.2.2. CASE\_NOT\_FOUND

case语句，如果when子句中没有包含必要的条件分支，就会触发CASE\_NOT\_FOUND异常。简单情况下有simple\_case\_statement，搜索情况下有search\_case\_statement。

例如，简单情况下的simple\_case\_statement。

```
create or replace function case_not_found_f return int as
  grade CHAR(1);
BEGIN
  grade := 'G';

  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
  END CASE;
EXCEPTION
  WHEN CASE_NOT_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No such grade');
  return 1;
END;
/

select case_not_found_f();
```

例如，搜索情况下的search\_case\_statement。

```
create or replace function case_not_found_f return int as
  grade CHAR(1);
BEGIN
  grade := 'G';
```

```

CASE
  WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
  WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
  WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
  WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
  WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
END CASE;
EXCEPTION
  WHEN CASE_NOT_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No such grade');
    return 1;
END;
/

select case_not_found_f();

```

#### 42.12.2.3. DUP\_VAL\_ON\_INDEX

当向唯一索引对应的列插入重复值时触发。

例如：

```

--preconditional
CREATE TABLE emp_name AS SELECT id, name FROM employees;

CREATE UNIQUE INDEX empname_ix ON emp_name (id);

create or replace procedure dup_val_on_index_p as
  emp_id      NUMBER(10);
  emp_name    VARCHAR2(100);
BEGIN
  SELECT id, name INTO emp_id, emp_name FROM employees WHERE id = 22091401;

  INSERT INTO emp_name (id, name) VALUES (emp_id, emp_name);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Inserts were rolled back');
END;
/

call dup_val_on_index_p();

```

#### 42.12.2.4. NOT\_DATA\_FOUND

当执行select into时没有返回任何行时触发。

例如：

```

-- no_data_found
-- select into
-- number of record selected is zero
create or replace function no_data_found_f return int as

```

```

    v_emp int;
begin
    select id into v_emp from employees where id = 22091501;
EXCEPTION
    WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('No data found');
    return 1;
end;
/

select no_data_found_f();

```

#### 42.12.2.5. TOO\_MANY\_ROWS

当执行select into语句时，如果返回超过一行，则会触发TOO\_MANY\_ROWS异常。

例如：

```

create or replace function too_many_rows_f return int as
    v_emp int;
begin
    select id into v_emp from employees;

EXCEPTION
    WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('No data found');
    WHEN too_many_rows THEN
        DBMS_OUTPUT.PUT_LINE('Too Many Rows');

    return 1;
end;
/

select too_many_rows_f();

```

#### 42.12.2.6. 登录被拒绝

你可以使用raise来抛出预定义的异常，并在异常后处理问题。（在plorasql中不会触发，但在LightDB中会触发）

例如：

```

create or replace function login_denied_f return int as
begin
    raise login_denied;
exception
    when login_denied then
        dbms_output.put_line('this is a login_denied message.. good luck the next
time');
    return 1;
end;
/

select login_denied_f();

```

### 42.12.2.7. 零除错误

当除数为0时，被除数不为0的数字执行除法的事件就会触发该异常。

例如：

```
create or replace function zero_divide_f return int as
  a int := 10;
  b int := 0;
begin
  a := a / b;
  exception
  when zero_divide then
    dbms_output.put_line('this is a normal error');
  return 1;
end;
/

select zero_divide_f();
```

#### Tip

1. 错误处理不支持处理问题摘要：
  - 匿名块中的动态SQL包含提交和回滚的匿名块，不支持；
  - 存储过程中的动态SQL包含提交和回滚的匿名块，不支持；
  - 包中的存储过程以及匿名块中的动态SQL包含提交和回滚，不支持。
2. 在包中，函数、存储过程和匿名块的执行顺序有问题；
3. 目前plorasql不支持保存点。

### 42.12.3. 显式抛出异常

为了显式抛出异常，请使用RAISE语句或RAISE\_APPLICATION\_ERROR过程。

你只能从存储子程序或方法中调用RAISE\_APPLICATION\_ERROR过程。通常，你调用这个过程来抛出一个用户定义的异常，并将它的错误代码和错误消息返回给调用者。

调用RAISE\_APPLICATION\_ERROR时，请使用以下语法：

```
RAISE_APPLICATION_ERROR (error_code, message[, {TRUE | FALSE}]);
```

错误代码是-20000..-20999范围内的整数。如果指定TRUE，PL/oraSQL将不会打印消息。否则，PL/oraSQL将打印消息。默认为FALSE。

### 42.12.4. 支持DBMS\_UTILITY内置包 format\_error\_backtrace函数

DBMS\_UTILITY.format\_error\_backtrace 函数是记录异常时必须调用的关键子程序。它返回一个字符串，将错误追溯到引发错误的行！追溯函数总是只追溯到最近引发的异常。错误允许返回的最大长度为10000。

使用DBMS\_UTILITY.format\_error\_backtrace函数的示例:

```
--预处理条件
create table test_error_stack(id number primary key);
insert into test_error_stack values (1);
create table t_errlog(operate_error_msg varchar(1000));

create or replace procedure pr_error_code_test1 is
    v_backtrace    varchar(1000);
    v_error        varchar(1000);
begin
    INSERT INTO test_error_stack values (2);
    INSERT INTO test_error_stack values (1);
    commit;
    exception when others then
        insert into t_errlog(operate_error_msg) select
to_char(dbms_utility.format_error_backtrace);
        commit;
        rollback;
    end pr_error_code_test1;
/

call pr_error_code_test1();
select * from t_errlog;

--仅使用函数调用
select to_char(dbms_utility.format_error_backtrace)
```

## 42.12.5. 支持 SQLCODE

支持在抛出异常时获取整数错误代码。该整数的错误代码可以转换为长度为5的字符串错误代码。转换函数 SQLCODE\_TO\_SQLSTATE 在 ora9ce。长度为5的字符串错误码在 Appendix C

例如:

```
declare
    a text;
    b int;
begin
    raise exception 'sqlcode test';
exception
    when others then
        a := sqlstate;
        b := sqlcode;
        raise notice 'sqlstate: %, sqlcode: %, sqlcode: %', a, b, sqlcode;
end;
/
select SQLCODE_TO_SQLSTATE(16777248);
```

## 42.13. 对象名称搜索规则

1. 包中的变量和列名称相同时，优先列名

```

create table test(a int);
insert into test values(1);

CREATE OR REPLACE function GET_AVG_PRICE()
  RETURN text IS

  a int;
BEGIN
  -- 列名      包变量
  select a into a      from test limit 1;
  dbms_output.put_line(a);

  return a || ' ';
END GET_AVG_PRICE;
/

```

2. 包名和表名有重复时，优先表名(23.3版本禁止重复)

```

create table tn (id int);

create package tn IS
id text;
end;
/

CREATE OR REPLACE function abc()
  RETURN text IS

  a tn.id%type; -- 引用表 类型为int
  v text;
BEGIN
  select pg_typeof(a) into v ;
  dbms_output.put_line(v);

  return v;
END abc;
/

```

另外包名和schema名称不允许重复，所以不存在冲突的情况。

## 42.14. 针对Oracle的增强功能

1. 函数功能增强：支持回滚和提交。

```

create table table_func_rbk(a int, b int);
insert into table_func_rbk values(100,100);
insert into table_func_rbk values(200,200);
select * from table_func_rbk order by a;

create or replace function func_rollback1(a int) return int

```

```
is
begin
  insert into table_func_rbk values(300,300);
  commit;
  insert into table_func_rbk values(400,400);
  rollback;
  update table_func_rbk set a = 400 where a = 300;
  rollback;
  delete from table_func_rbk where a = 100;
  rollback;
  return a;
end;
/
select func_rollback1(100);
select * from table_func_rbk order by a;

create or replace function func_rollback1(a int) return int
is
begin
  insert into table_func_rbk values('asdasda');
exception
when others then
  insert into table_func_rbk values(300,300);
  commit;
  insert into table_func_rbk values(400,400);
  rollback;
  update table_func_rbk set a = 400 where a = 300;
  rollback;
  delete from table_func_rbk where a = 100;
  rollback;
  return a;
end;
/
select func_rollback1(100);
select * from table_func_rbk order by a;

drop table table_func_rbk;
drop function func_rollback1;
```

---

## Chapter 43. 服务器编程接口

服务器编程接口 (SPI) 给予用户定义C函数编写者在其函数或程序内运行SQL命令的能力。SPI是一组接口函数，它们可以简化对解析器、规划器和执行器的访问。SPI也做一些内存管理。

### Note

可用的过程语言提供了多种方法从函数中执行 SQL 命令。大部分这些设施都是基于 SPI 的，因此这个文档也对那些语言的用户有用。

注意如果一个通过 SPI 调用的命令失败，那么控制将会返回到你的C函数中。当然啦，你的C函数所在的事务或者子事务将被回滚（这可能看起来令人惊讶，因为据文档所说 SPI 函数大多数都有错误返回约定。但是那些约定只适用于在 SPI 函数本身内部检测到的错误）。通过在可能失败的 SPI 调用周围建立自己的子事务可以在错误之后恢复控制。

SPI成功时返回一个非负结果（要么通过一个返回的整数值，要么如下所述放在全局变量SPI\_result中）。错误时，将会返回一个负结果或者NULL。

使用 SPI 的源代码文件必须包括头文件executor/spi.h。

### 43.1. 接口函数



## SPI\_connect

SPI\_connect, SPI\_connect\_ext — 连接一个C函数到 SPI 管理器

## Synopsis

```
int SPI_connect(void)
```

```
int SPI_connect_ext(int options)
```

## 描述

SPI\_connect从一个C函数调用中打开一个到 SPI 管理器的连接。如果你想要通过 SPI 执行命令，你必须调用这个函数。有一些功能性 SPI 函数可以从未连接的C函数中调用。

SPI\_connect\_ext会做同样的事情，但是有一个允许传递选项标志的参数。当前有下列选项值可用：

SPI\_OPT\_NONATOMIC

设置SPI连接为nonatomic，这表示允许事务控制调用SPI\_commit、SPI\_rollback以及SPI\_start\_transaction。否则，调用这些函数将立即导致错误。

SPI\_connect() 等效于SPI\_connect\_ext(0)。

## 返回值

SPI\_OK\_CONNECT

成功时

SPI\_ERROR\_CONNECT

错误时

## SPI\_finish

SPI\_finish — 将一个C函数从 SPI 管理器断开

### Synopsis

```
int SPI_finish(void)
```

### 描述

SPI\_finish关闭一个到 SPI 管理器的现有连接。你必须在完成你的C函数的当前调用中所需的 SPI 操作之后必须调用这个函数。不过，如果你通过elog(ERROR)中断了事务，你无须担心这个函数的调用。在那种情况下，SPI 将自己自动进行清理。

### 返回值

SPI\_OK\_FINISH

如果正确地断开连接

SPI\_ERROR\_UNCONNECTED

如果从一个未连接的C函数中调用

## SPI\_execute

SPI\_execute — 执行一个命令

### Synopsis

```
int SPI_execute(const char * command, bool read_only, long count)
```

### 描述

SPI\_execute执行指定的 SQL 命令以获得count行。如果read\_only为true，该命令必须是只读的，并且执行开销也会有所降低。

只能从一个已连接的C函数中调用这个函数。

如果count为零，那么该命令会为其所适用的所有行执行。如果count大于零，那么会检索不超过count行，当到达该计数时执行会停止，这很像为查询增加一个LIMIT子句。例如：

```
SPI_execute("SELECT * FROM foo", true, 5);
```

会从表中检索至多 5 行。注意这样一个限制只有当命令真正返回行时才有效。例如：

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

插入所有来自于bar的行，而忽略count参数。不过，通过

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

将插入至多 5 行，因为在第五个RETURNING结果行被检索到后执行就会停止。

你可以在一个字符串中传递多个命令，SPI\_execute会返回最后一个被执行的命令的结果。count限制单独适用于每一个命令（即便只有最后一个结果会被实际返回）。该限制 不适用于由规则产生的任何隐藏命令。

当read\_only是false时，SPI\_execute增加命令计数器并且在执行字符串中每一个命令之前计算一个新的snapshot。如果当前事务隔离级别是SERIALIZABLE或REPEATABLE READ，该快照并不会实际改变。但是在READ COMMITTED模式中，快照更新允许每个命令看到来自其他会话中新近已提交事务的结果。当命令正在修改数据库时，这对一致性行为非常重要。

当read\_only是true时，SPI\_execute不更新快照或者命令计数器，并且它只允许纯SELECT命令出现在命令字符串中。这些命令被使用之前为周围查询建立的快照来执行。这种执行模式要比读/写模式更快，因为消除了每个命令跟新快照的开销。它也允许建立真正stable的函数：因为连续执行将会使用同一个快照，因此结果不会有改变。

在一个使用 SPI 的单一函数中混合只读和读写命令通常是不明智的，这样可能会导致非常令人困惑的行为，因为只读查询将看不到任何由读写查询完成的数据库更新结果。

被执行的（最后一个）命令的实际行数使用全局变量SPI\_processed返回。如果该函数的返回值是SPI\_OK\_SELECT、SPI\_OK\_INSERT\_RETURNING、SPI\_OK\_DELETE\_RETURNING或者SPI\_OK\_UPDATE\_RETURNING，那么你可以使用全局指针SPITupleTable \*SPI\_tuptable来访问结果行。某些工具命令（例如EXPLAIN）也返回行集合，并且在这些情况中SPI\_tuptable也会包含该结果。某些工具命令（COPY、CREATE TABLE AS）不返回一个行集合，因此SPI\_tuptable为 NULL，但是它们仍然会在SPI\_processed中返回被处理的行数。

结构SPITupleTable被定义为:

```
typedef struct SPITupleTable
{
    /* Public members */
    TupleDesc tupdesc; /* tuple descriptor */
    HeapTuple *vals; /* array of tuples */
    uint64 numvals; /* number of valid tuples */

    /* Private members, not intended for external callers */
    uint64 allocated; /* allocated length of vals array */
    MemoryContext tuptabctx; /* memory context of result table */
    slist_node next; /* link for internal bookkeeping */
    SubTransactionId subid; /* subxact in which tuptable was created */
} SPITupleTable;
```

SPI 调用者可以使用字段tupdesc、vals和numvals；其余字段是内部的。vals 是一个指向行的指针数组。行数由 numvals 给出（由于某些历史原因，这个计数也在 SPI\_processed 中返回）。tupdesc 是一个行描述符，您可以将其传递给处理行的 SPI 函数。

SPI\_finish释放在当前的C函数中已分配的所有SPITupleTable。如果你已经用完了一个结果表，你可以通过调用SPI\_freetuptable提早释放它。

## 参数

const char \* command

包含要执行命令的字符串

bool read\_only

对只读执行为true

long count

要返回的最大行数，或者用0表示没有限制

## 返回值

如果命令的执行成功，那么将会返回下列（非负）值之一：

SPI\_OK\_SELECT

如果执行了一个SELECT（但不是SELECT INTO）

SPI\_OK\_SELINTO

如果执行了一个SELECT INTO

SPI\_OK\_INSERT

如果执行了一个INSERT

SPI\_OK\_DELETE

如果执行了一个DELETE

SPI\_OK\_UPDATE

如果执行了一个UPDATE

SPI\_OK\_INSERT\_RETURNING

如果执行了一个INSERT RETURNING

SPI\_OK\_DELETE\_RETURNING

如果执行了一个DELETE RETURNING

SPI\_OK\_UPDATE\_RETURNING

如果执行了一个UPDATE RETURNING

SPI\_OK\_UTILITY

如果执行了一个工具命令（例如CREATE TABLE）

SPI\_OK\_REWRITTEN

如果该命令被一个规则重写成了另一类命令（例如UPDATE变成了一个INSERT）

发生错误时，将会返回下列负值之一：

SPI\_ERROR\_ARGUMENT

如果command为NULL或者count小于 0

SPI\_ERROR\_COPY

如果尝试COPY TO stdout或者COPY FROM stdin

SPI\_ERROR\_TRANSACTION

如果尝试了一个事务操纵命令（BEGIN、COMMIT、ROLLBACK、SAVEPOINT、PREPARE TRANSACTION、COMMIT PREPARED、ROLLBACK PREPARED或者其他变体）

SPI\_ERROR\_OPUNKNOWN

如果命令类型位置（不应该会发生）

SPI\_ERROR\_UNCONNECTED

如果从未连接的C函数中调用

## 注解

所有 SPI 查询执行函数都会设置SPI\_processed和SPI\_tuptable（只是指针，而不是结构的内容）。如果你需要在以后访问SPI\_execute或另一个查询执行函数的结果表，请将这两个全局变量保存到本地的C函数变量中。

## SPI\_exec

SPI\_exec — 执行一个读/写命令

## Synopsis

```
int SPI_exec(const char * command, long count)
```

## 描述

SPI\_exec和 SPI\_execute相同，但后者的 read\_only参数的值总是取 false。

## Arguments

const char \* command

命令字符串

long count

最大返回行数，或0表示没有限制。

## 返回值

返回值同SPI\_execute一样。

## SPI\_execute\_with\_args

SPI\_execute\_with\_args — 用线外参数执行一个命令

### Synopsis

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

### 描述

SPI\_execute\_with\_args执行一个可能包括 对外部提供的参数引用的命令。命令文本用 \$n引用一个参数，并且调用 会为每一个这种符号指定数据类型和值。read\_only和 count的解释与 SPI\_execute中相同。

相对于SPI\_execute，这个例程的主要优点是数据值可以被插入到命令中而无需冗长的引用/转义，并且因此 减少了 SQL 注入攻击的风险。

可以通过在SPI\_prepare后面跟上 SPI\_execute\_plan达到相似的结果。但是，使用这个函数时查询计划总是被定制成提供的指定参数值。对于一次性的查询执行，这个函数应该更好。如果同样的命令需要用很多不同的参数执行，两种方法都可能会更快，这取决于重新做规划的代价与 定制计划带来的好处之间的对比。

### 参数

const char \* command

命令字符串

int nargs

输入参数的数量（\$1、\$2等等）。

Oid \* argtypes

一个长度为nargs的数组，包含参数的数据类型的OID

Datum \* values

一个长度为nargs的数组，包含实际的参数值

const char \* nulls

一个长度为nargs的数组，描述哪些参数为空值

如果nulls为NULL，那么SPI\_execute\_with\_args会假设没有参数 为空值。否则，如果对应的参数值为非空，nulls 数组的每一个项都应该是' '；如果对应参数值为空，nulls数组的项应为'n'（在后 面的情况中，对应的values项中的值没有 关系）。注意nulls不是一个文本字符串，它只是一个数组：它不需要一个'\0'终止符。

bool read\_only

对只读执行是true

long count

要返回的最大行数，0表示没有限制

## 返回值

该返回值和SPI\_execute一样。

如果成功SPI\_execute会设置 SPI\_processed和 SPI\_tuptable。



## SPI\_prepare

SPI\_prepare — 准备一个语句，但不执行它

## Synopsis

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

## 描述

SPI\_prepare为指定的命令创建并且返回一个 预备语句，但是并不执行该命令。该预备语句会在稍后使用 SPI\_execute\_plan重复执行。

当相同的或者相似的命令要被重复执行时，通常来说只执行一次解析分析是有利的，并且更有利的是重用该命令的执行计划。SPI\_prepare把一个命令字符串转换成一个预备语句，它包装了解析分析的结果。如果发现为每一次执行都生成一个 定制计划没有帮助，该预备语句也提供了一个地方缓存执行计划。

一个预备命令可以被一般化为在一个普通命令中应该出现常量的地方写上参数（\$1、\$2等等）。参数的实际值在 SPI\_execute\_plan被调用时指定。这让该预备 语句可以比没有参数的形式用户与更广泛的情况。

SPI\_prepare返回的语句只能在当前的C过程调用 中使用，因为SPI\_finish会释放为这样一个语句 分配的内存。但是可以使用函数SPI\_keepplan 或SPI\_saveplan把该语句保存更久。

## 参数

const char \* command

命令字符串

int nargs

输入参数（\$1、\$2等等）的数量

Oid \* argtypes

一个数组指针，它指向的数组包含参数的数据类型 OID

## 返回值

SPI\_prepare返回一个指向SPIPlan 的非空指针，它是一个表示一个预备语句的不透明结构。发生错误时， 将会返回NULL，并且 SPI\_result将被设置为一个也被 SPI\_execute使用的错误码，不过当 command为NULL、 或者nargs小于零、或者nargs大于 零但是argtypes为NULL 时它会被设置为SPI\_ERROR\_ARGUMENT。

## 注解

如果没有定义参数，在第一次使用SPI\_execute\_plan 时将会创建一个一般的计划，并且把它用于所有的后续执行。如果有参数， SPI\_execute\_plan的前几次使用将根据提供的参数值 产生定制计划。在使用同一个预备语句足够多次后， SPI\_execute\_plan将构建一个一般计划，并且如果它 并不比定制计划昂贵太多， SPI\_execute\_plan将开始使用一般计划来取代每次都进行重新规划。如果这种默认的行为不合适，你可以 通过传递CURSOR\_OPT\_GENERIC\_PLAN或 CURSOR\_OPT\_CUSTOM\_PLAN标志给 SPI\_prepare\_cursor，以分别强制使用一般或者定制 计划。

尽管一个预备语句的要点是避免对语句的重复解析分析以及规划，只要语句中用到的数据库对象从上一次使用该预备语句以来经历过定义性（DDL）改变，LightDB将会强制重新分析和重新规划该语句。还有，如果search\_path的值从一个改变成下一个，该语句将会使用新的search\_path进行重新解析（后一种行为是从LightDB 9.3开始的新行为）。更多有关预备语句行为的信息请见PREPARE。

这个函数只能从一个已连接的C函数调用。

SPIPlanPtr被声明为spi.h中的一种不透明结构类型的指针。尝试直接访问其内容是不明智的，因为那会让你的代码更有可能会在未来版本的LightDB中崩溃。

SPIPlanPtr这个名字多少有点历史原因，因为该数据结构不再需要包含一个执行计划。

## SPI\_prepare\_cursor

`SPI_prepare_cursor` — 预备一个语句，但是不执行它

## Synopsis

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,  
                              Oid * argtypes, int cursorOptions)
```

## 描述

`SPI_prepare_cursor`和 `SPI_prepare`一样，不过它也允许说明规划器的“游标选项”参数。这是一个位掩码，它的值如 `nodes/parsenodes.h`中 `DeclareCursorStmt`的`options`域所示。`SPI_prepare`总是把该游标选项取做零。

## 参数

`const char * command`

命令字符串

`int nargs`

输入参数（\$1、\$2等等）的数量

`Oid * argtypes`

一个数组指针，它指向的数组包含参数的数据类型 OID

`int cursorOptions`

整数形式的游标选项位掩码，零会导致默认行为

## 返回值

`SPI_prepare_cursor`具有和 `SPI_prepare`一样的返回习惯。

## 注解

在`cursorOptions`设置的有用的位包括 `CURSOR_OPT_SCROLL`、`CURSOR_OPT_NO_SCROLL`、`CURSOR_OPT_FAST_PLAN`、`CURSOR_OPT_GENERIC_PLAN`以及 `CURSOR_OPT_CUSTOM_PLAN`。注意`CURSOR_OPT_HOLD`被特别地忽略。

## SPI\_prepare\_params

SPI\_prepare\_params — 预备一个语句，但是不执行它

## Synopsis

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

## 描述

SPI\_prepare\_params为指定的命令创建并返回一个预备语句，但是不执行该命令。这个函数等效于 SPI\_prepare\_cursor，此外调用者可以指定解析器钩子函数来控制外部参数引用的解析。

## 参数

const char \* command

命令字符串

ParserSetupHook parserSetup

解析器钩子设置函数

void \* parserSetupArg

用于parserSetup的转嫁参数

int cursorOptions

整数形式的游标选项位掩码，零会导致默认行为

## 返回值

SPI\_prepare\_params具有和 SPI\_prepare相同的返回习惯。

## SPI\_getargcount

SPI\_getargcount — 返回一个由SPI\_prepare准备好的语句所需的参数数量

### Synopsis

```
int SPI_getargcount(SPIPlanPtr plan)
```

### 描述

SPI\_getargcount返回执行一个由 SPI\_prepare准备好的语句所需的参数数量。

### 参数

SPIPlanPtr plan

预备语句（由SPI\_prepare返回）

### 返回值

plan所期望的参数计数。如果该 plan为NULL或者无效, SPI\_result会被设置为SPI\_ERROR\_ARGUMENT 并且返回 -1。

## SPI\_getargtypeid

SPI\_getargtypeid — 为由SPI\_prepare 准备好的一个语句的一个参数返回其数据类型 OID

### Synopsis

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

### 描述

SPI\_getargtypeid返回由 SPI\_prepare准备好的一个语句的 第argIndex个参数的类型的OID。 第一个参数的索引为零。

### 参数

SPIPlanPtr plan

预备语句（由SPI\_prepare返回）

int argIndex

参数的索引，从零开始

### 返回值

给定索引处的参数的类型 OID。如果该 plan为NULL或者无效， 或者argIndex小于零或者小于为 plan声明的参数数量， SPI\_result会被设置为 SPI\_ERROR\_ARGUMENT并且将会返回InvalidOid。

## SPI\_is\_cursor\_plan

`SPI_is_cursor_plan` — 如果一个由`SPI_prepare`预备好的语句可以用于`SPI_cursor_open`则返回 `true`

## Synopsis

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

## 描述

如果一个由`SPI_prepare`预备好的语句可以被作为一个参数传递给`SPI_cursor_open`，`SPI_is_cursor_plan`会返回`true`。否则返回`false`。原则是该 `plan`表示一个单一命令并且这个命令向其调用者 返回元组。例如，只要不含`INTO`子句，`SELECT` 就被允许，而只有包含一个`RETURNING`子句时才允许 `UPDATE`。

## 参数

`SPIPlanPtr plan`

预备语句（由`SPI_prepare`返回）

## 返回值

如果该`plan`能产生一个游标则返回 `true`，否则返回`false` 并且把`SPI_result`设置为零。如果不可能决定答案（例如，如果`plan`为 `NULL`或无效，或者在没有连接到 `SPI` 时调用），那么`SPI_result`会被设置为一个合适的错误码 并且返回`false`。

## SPI\_execute\_plan

SPI\_execute\_plan — 执行一个由SPI\_prepare预备好的语句

## Synopsis

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                    bool read_only, long count)
```

## 描述

SPI\_execute\_plan执行一个由 SPI\_prepare或其同类方法准备好的语句。 read\_only和 and count的解释和 SPI\_execute中相同。

## 参数

SPIPlanPtr plan

预备语句（由SPI\_prepare返回）

Datum \* values

一个实际参数值的数组。必须和语句的参数数量等长。

const char \* nulls

一个描述哪些参数为空值的数组。必须和语句的参数数量等长。

如果nulls为NULL，那么SPI\_execute\_plan会假设没有参数 为空值。否则，如果对应的参数值为非空， nulls 数组的每一个项都应该是' '；如果对应参数值为空， nulls数组的项应为'n'（在后 面的情况中，对应的values项中的值没有 关系）。注意nulls不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

bool read\_only

true表示只读执行

long count

要返回的行的最大数量，或者用0表示没有限制

## 返回值

返回值和SPI\_execute相同， 还有下列额外可能的错误（负值）结果：

SPI\_ERROR\_ARGUMENT

如果plan为NULL 或者非法，或者count小于 0

SPI\_ERROR\_PARAM

如果values为NULL但是 plan被准备时用了一些参数

成功时，就像在SPI\_execute中会设置 SPI\_processed和 SPI\_tuptable。



## SPI\_execute\_plan\_with\_paramlist

SPI\_execute\_plan\_with\_paramlist — 执行一个由SPI\_prepare准备好的语句

### Synopsis

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                   ParamListInfo params,
                                   bool read_only,
                                   long count)
```

### 描述

SPI\_execute\_plan\_with\_paramlist执行一个由 SPI\_prepare准备好的语句。这个函数与 SPI\_execute\_plan等效，不过被传递给该查询的参数值的信息以不同的方式呈现。ParamListInfo表现形式更方便于把这种格式的值向下传递。它也支持通过ParamListInfo中指定的钩子函数动态设置参数。

这个函数现在废弃了，由SPI\_execute\_plan\_extended代替。

### 参数

SPIPlanPtr plan

预备语句（由SPI\_prepare返回）

ParamListInfo params

包含参数类型和值的数据结构，如果没有则为 NULL

bool read\_only

true表示只读执行

long count

要返回的行的最大数量，或者用0表示没有限制

### 返回值

返回值和SPI\_execute\_plan相同。

成功时，在SPI\_execute\_plan中会设置 SPI\_processed和 SPI\_tuptable。

## SPI\_execp

SPI\_execp — 以读/写模式执行一个语句

## Synopsis

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

## 描述

SPI\_execp与 SPI\_execute\_plan相同，不过后者的 read\_only参数总是取false。

## 参数

SPIPlanPtr plan

预备语句（由SPI\_prepare返回）

Datum \* values

实际参数值的数组。长度必须等于该语句的参数数量。

const char \* nulls

描述哪些参数为空值的数据。长度必须等于该语句的参数数量。

如果nulls为NULL，那么SPI\_execp会假设没有参数 为空值。否则，如果对应的参数值为非空， nulls 数组的每一个项都应该是' '；如果对应参数值为空， nulls数组的项应为'n'（在后 面的情况中，对应的values项中的值没有 关系）。注意nulls不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

long count

要返回的行的最大数量，或者用0表示没有限制

## 返回值

见SPI\_execute\_plan。

成功时，就像在SPI\_execute中会设置 SPI\_processed和 SPI\_tuptable。

## SPI\_cursor\_open

SPI\_cursor\_open — 使用由SPI\_prepare创建的 语句建立一个游标

## Synopsis

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

## 描述

SPI\_cursor\_open建立一个游标（在内部是一个 portal），该游标将执行由SPI\_prepare准备好 的一个语句。参数具有和SPI\_execute\_plan的 相应参数相同的含义。

使用一个游标而不是直接执行该语句有两个好处。首先，可以一次只取出 一些结果行，避免为返回很多行的查询过度使用内存。其次，一个 portal 可以比当前的C函数生存更长时间（事实上，它可以生存到当前事务结束）。把 portal 的名称返回给该C函数的调用者提供了一种将一个行集合返回为结 果的方法。

被传入的参数数据将被复制到游标的 portal 中，因此在该游标仍然存在时 可以释放掉被传入的参数数据。

## 参数

const char \* name

portal 的名字，或者设置成NULL 让系统选择一个名称

SPIPlanPtr plan

预备语句（由SPI\_prepare返回）

Datum \* values

实际参数值的数组。长度必须等于该语句的参数数量。

const char \* nulls

描述哪些参数是空值的数据。长度必须等于该语句的参数数量。

如果nulls为NULL，那么SPI\_cursor\_open会假设没有参数 为空值。否则，如果对应的参数值为非空， nulls 数组的每一项都应该是' '；如果对应参数值为空， nulls数组的项应为'n'（在后 面的情况中，对应的values项中的值没有 关系）。注意nulls不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

bool read\_only

true表示只读执行

## 返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定， 任何错误都将通过elog报告。

## SPI\_cursor\_open\_with\_args

SPI\_cursor\_open\_with\_args — 使用一个查询和参数建立一个游标

### Synopsis

```
Portal SPI_cursor_open_with_args(const char *name,
                                const char *command,
                                int nargs, Oid *argtypes,
                                Datum *values, const char *nulls,
                                bool read_only, int cursorOptions)
```

### 描述

SPI\_cursor\_open\_with\_args建立一个将 执行指定查询的游标（在内部是一个 portal）。大部分参数具有和 SPI\_prepare\_cursor 和SPI\_cursor\_open中相应参数相同的含义。

对于一次性的查询执行，这个函数应该比 SPI\_prepare\_cursor加上其后的 SPI\_cursor\_open更好。如果相同的命令 要被用很多不同的参数执行，哪种方法更快就要取决于重做计划的 代价与定制计划带来的好处之间谁更有利。

被传入的参数数据将被复制到游标的 portal 中，因此在该游标仍然存在时 可以释放掉被传入的参数数据。

### 参数

const char \* name

portal 的名字，或者设置成NULL 让系统选择一个名称

const char \* command

命令字符串

int nargs

输入参数的数量（\$1、\$2等等）

Oid \* argtypes

一个长度为nargs的数组，它包含参数的 数据类型的OID

Datum \* values

一个长度为nargs的数组，它包含实际的参数值

const char \* nulls

一个长度为nargs的数组，它描述哪些参数为空值

如果nulls为NULL，那么SPI\_cursor\_open\_with\_args会假设没有参数 为空值。否则，如果对应的参数值为非空， nulls 数组的每一个项都应该是' '；如果对应参数值为空， nulls数组的项应为'n'（在后 面的情况中，对应的values项中的值没有 关系）。注意nulls不是一个文本字符串， 它只是一个数组：它不需要一个'\0'终止符。

bool read\_only

true表示只读执行

`int cursorOptions`

光标选项的整数型位掩码，为零会产生默认行为

## 返回值

指向包含该游标的 `portal` 的指针。注意这里没有错误返回约定，任何错误都将通过 `eLog` 报告。

## SPI\_cursor\_open\_with\_paramlist

SPI\_cursor\_open\_with\_paramlist — 使用参数建立一个游标

### Synopsis

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

### 描述

SPI\_cursor\_open\_with\_paramlist建立一个游标（在内部是一个 portal），它将执行一个由 SPI\_prepare准备好的语句。这个函数等效于 SPI\_cursor\_open，不过被传递给该查询的参数值的信息以不同的方式呈现。ParamListInfo表现形式更方便于把这种格式的值向下传递。它也支持通过 ParamListInfo中指定的钩子函数动态设置参数。

被传入的参数数据将被复制到游标的 portal 中，因此在该游标仍然存在时 可以释放掉被传入的参数数据。

### 参数

const char \* name

portal 的名字，或者设置成NULL 让系统选择一个名称

SPIPlanPtr plan

预备语句（由SPI\_prepare返回）

ParamListInfo params

包含参数类型和值的数据结构，如果没有就为 NULL

bool read\_only

true表示只读执行

### 返回值

指向包含该游标的 portal 的指针。注意这里没有错误返回约定，任何错误都将通过elog报告。

## SPI\_cursor\_find

SPI\_cursor\_find — 用名称查找一个现有的游标

## Synopsis

```
Portal SPI_cursor_find(const char * name)
```

## 描述

SPI\_cursor\_find用名称查找一个现有的 portal。这主要被用于解析由其他某个函数返回的一个油表名称。

## 参数

const char \* name

该 portal 的名称

## 返回值

带有指定名称的 portal 的指针，如果没有找到就是 NULL

## SPI\_cursor\_fetch

SPI\_cursor\_fetch — 从一个游标取出一些行

### Synopsis

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

### 描述

SPI\_cursor\_fetch从一个游标取得一些行。这等效于 SQL 命令FETCH的一个子集（更多功能见SPI\_scroll\_cursor\_fetch）。

### 参数

Portal portal

包含该游标的 portal

bool forward

为真表示向前获取，为假表示向后获取

long count

要取得的最大行数

### 返回值

成功时，就像在SPI\_execute中会设置 SPI\_processed和 SPI\_tuptable。

### 注解

如果该游标的计划不是用CURSOR\_OPT\_SCROLL 选项创建的，向后获取会失败。



## SPI\_cursor\_move

SPI\_cursor\_move — 移动一个游标

### Synopsis

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

### 描述

SPI\_cursor\_move跳过一个游标中的一些行。这等效于 SQL 命令MOVE的一个子集（更多的功能 请见SPI\_scroll\_cursor\_move）。

### 参数

Portal portal

包含该游标的 portal

bool forward

为真表示前移，为假表示后移

long count

要移动的最大行数

### 注解

如果该游标的计划不是用CURSOR\_OPT\_SCROLL 选项创建的，向后移动会失败。

## SPI\_scroll\_cursor\_fetch

SPI\_scroll\_cursor\_fetch — 从一个游标取出一些行

### Synopsis

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,  
                             long count)
```

### 描述

SPI\_scroll\_cursor\_fetch从一个游标中取出一些行。这等效于 SQL 命令FETCH。

### 参数

Portal portal

包含该游标的 portal

FetchDirection direction

FETCH\_FORWARD、FETCH\_BACKWARD、FETCH\_ABSOLUTE或者 FETCH\_RELATIVE之一

long count

FETCH\_FORWARD或 FETCH\_BACKWARD方式中要取出的行数； FETCH\_ABSOLUTE方式中要取出的绝对行号； FETCH\_RELATIVE方式中要取出的相对行号

### 返回值

成功时，就像在SPI\_execute中会设置 SPI\_processed和 SPI\_tuptable。

### 注解

参数direction和 count的详细解释请见 SQL FETCH命令。

如果该游标的计划不是用CURSOR\_OPT\_SCROLL 选项创建的，除FETCH\_FORWARD之外的方向值会失败。

## SPI\_scroll\_cursor\_move

SPI\_scroll\_cursor\_move — 移动一个游标

### Synopsis

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,  
                             long count)
```

### 描述

SPI\_scroll\_cursor\_move在一个游标中跳过 一定数量的行。这等效于 SQL 命令MOVE。

### 参数

Portal portal

包含该游标的 portal

FetchDirection direction

FETCH\_FORWARD、 FETCH\_BACKWARD、 FETCH\_ABSOLUTE或者 FETCH\_RELATIVE之一

long count

FETCH\_FORWARD或者 FETCH\_BACKWARD方式中要移动的行数； FETCH\_ABSOLUTE方式中要移动到的绝对行号； FETCH\_RELATIVE方式中要移动到的相对行号

### 返回值

成功时，就像在SPI\_execute中会设置 SPI\_processed。 SPI\_tuptable被设置为NULL， 因为这个函数不需要返回行。

### 注解

参数direction和 count的详细解释请见 SQL FETCH命令。

如果该游标的计划不是用CURSOR\_OPT\_SCROLL 选项创建的，除FETCH\_FORWARD之外的方向值会 失败。

## SPI\_cursor\_close

SPI\_cursor\_close — 关闭一个游标

### Synopsis

```
void SPI_cursor_close(Portal portal)
```

### 描述

SPI\_cursor\_close关闭一个之前创建的游标 并且释放它的 portal 存储。

所有打开的游标会在事务结束时自动被关闭。 只有在希望尽快释放资源时，才需要调用 SPI\_cursor\_close。

### 参数

Portal portal

包含该游标的 portal

## SPI\_keepplan

SPI\_keepplan — 保存一个预备语句

### Synopsis

```
int SPI_keepplan(SPIPlanPtr plan)
```

### 描述

SPI\_keepplan保存一个被传入的语句（由SPI\_prepare准备好），这样它将不会被SPI\_finish或者事务管理器释放。这让你能够在当前会话的后续C函数调用中重用预备语句。

### 参数

SPIPlanPtr plan

要保存的预备语句

### 返回值

成功返回 0；如果plan为NULL 或者无效则返回SPI\_ERROR\_ARGUMENT

### 注解

这个函数通过指针调整的方法（不需要数据复制）将被传入的语句重定位到永久存储中。如果你后来需要删除它，可以对它使用 SPI\_freeplan。

## SPI\_saveplan

SPI\_saveplan — 保存一个预备语句

## Synopsis

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

## 描述

SPI\_saveplan把一个被传入的语句（由 SPI\_prepare准备好）复制到不会被 SPI\_finish或者事务管理器释放的内存中。这让你能够在当前会话的后续C函数调用中重用预备语句。

## 参数

SPIPlanPtr plan

要保存的预备语句

## 返回值

要被复制的语句的指针；如果没有成功则返回NULL。错误时，SPI\_result会被这样设置：

SPI\_ERROR\_ARGUMENT

如果plan为NULL或无效

SPI\_ERROR\_UNCONNECTED

如果从一个未连接的C函数调用

## 注解

原始的被传入的语句不会被释放，因此你可能希望在其上执行 SPI\_freepplan以避免在 SPI\_finish之前发生内存泄露。

在大部分情况下，SPI\_keepplan更适合于 执行这种功能，因为它极大程度上达到了同样的结果而无需物理地 复制该预备语句的数据结构。

## SPI\_register\_relation

SPI\_register\_relation — make an ephemeral named relation available by name in SPI queries

### Synopsis

```
int SPI_register_relation(EphemeralNamedRelation enr)
```

### 简介

SPI\_register\_relation用相关信息建立一个短暂的命名关系，它对通过当前SPI连接规划和执行的查询可用。

### 参数

EphemeralNamedRelation enr

短暂的命名关系的注册项

### 返回值

如果该命令的执行成功，则将返回下列（非负）值：

SPI\_OK\_REL\_REGISTER

如果该关系已经成功地用名称注册

在出错时，会返回下列负值之一：

SPI\_ERROR\_ARGUMENT

如果enr是NULL或者其name字段是NULL

SPI\_ERROR\_UNCONNECTED

如果从一个未连接的C函数中调用

SPI\_ERROR\_REL\_DUPLICATE

如果enr的name字段中指定的名称已经为这个连接注册

## SPI\_unregister\_relation

SPI\_unregister\_relation — remove an ephemeral named relation from the registry

### Synopsis

```
int SPI_unregister_relation(const char * name)
```

### 简介

SPI\_unregister\_relation从当前连接的注册项中移除一个短暂存在的关系。

### 参数

const char \* name

关系的注册项名称

### 返回值

如果该命令的执行成功，则会返回下列（非负）值：

SPI\_OK\_REL\_UNREGISTER

如果tuplestore已经被成功地从注册项中移除

出现错误时，会返回下列负值之一：

SPI\_ERROR\_ARGUMENT

如果name为NULL

SPI\_ERROR\_UNCONNECTED

如果从一个未连接的C函数中调用

SPI\_ERROR\_REL\_NOT\_FOUND

如果没有在当前连接的注册项中找到name



## SPI\_register\_trigger\_data

SPI\_register\_trigger\_data — make ephemeral trigger data available in SPI queries

### Synopsis

```
int SPI_register_trigger_data(TriggerData *tdata)
```

### 简介

SPI\_register\_trigger\_data会造出被触发器捕获的任何短暂存在的关系，它们对通过当前SPI连接规划和执行的查询可用。当前，这表示用REFERENCING OLD/NEW TABLE AS ... 子句定义的被AFTER触发器捕获的传递表。这个函数应该被一个PL触发器的处理器函数在连接之后调用。

### 参数

TriggerData \*tdata

以fcinfo->context传递给触发器处理器函数的TriggerData对象

### 返回值

如果命令的执行成功，则会返回下列（非负）值：

SPI\_OK\_TD\_REGISTER

如果被捕获的触发器数据（如果有）已经被成功地注册

出现错误时，会返回下列负值之一：

SPI\_ERROR\_ARGUMENT

如果tdata为NULL

SPI\_ERROR\_UNCONNECTED

如果从一个未连接的C函数中调用

SPI\_ERROR\_REL\_DUPLICATE

如果任何触发器数据瞬时关系的名字已经为这个连接注册过

## 43.2. 接口支持函数

这里描述的函数提供了一个接口从SPI\_execute 及其他 SPI 函数返回的结果集中抽取信息。

这一小节中描述的所有函数都可以被用在已连接和未连接的C函数中。

## SPI\_fname

SPI\_fname — 为指定的列号确定列名

## Synopsis

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

## 描述

SPI\_fname返回一个指定列的列名的拷贝（当你不再需要该列名拷贝后，可以使用pfree 释放它）。

## 参数

TupleDesc rowdesc

    输入行描述

int colnumber

    列号（从 1 开始计）

## 返回值

列名；如果colnumber超出范围则返回 NULL。出错时 SPI\_result会被设置成 SPI\_ERROR\_NOATTRIBUTE。

## SPI\_fnumber

SPI\_fnumber — 为一个指定的列名确定列号

## Synopsis

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

## 描述

SPI\_fnumber返回指定列名的列号。

如果colname引用的是一个系统列（例如，ctid），那么将返回对应的负值列号。调用者应该小心地测试返回值是不是正好为SPI\_ERROR\_NOATTRIBUTE来检测错误；除非系统列应该被拒绝，测试结果是否小于或者等于零这种方式是不正确的。

## 参数

TupleDesc rowdesc

输入行描述

const char \* colname

列名

## 返回值

列号（用户定义的列从1开始计），如果没有找到所提到的列名则返回SPI\_ERROR\_NOATTRIBUTE。

## SPI\_getvalue

SPI\_getvalue — 返回指定列的字符串值

## Synopsis

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

## 描述

SPI\_getvalue返回指定列的值的字符串表示。

结果在使用palloc分配的内存中返回（当你不再需要该结果时，你可以使用pfree释放该内存）。

## 参数

HeapTuple row

要检查的输入行

TupleDesc rowdesc

输入行描述

int colnumber

列号（从 1 开始计）

## 返回值

列值，如果列为空值、colnumber超出范围（SPI\_ERROR\_NOATTRIBUTE）或者没有输出函数（SPI\_ERROR\_NOOUTFUNC）则返回 NULL。

（SPI\_result被设置为可用（SPI\_result被设置为

## SPI\_getbinval

SPI\_getbinval — 返回指定列的二进制值

### Synopsis

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,  
                    bool * isnull)
```

### 描述

SPI\_getbinval以内部格式（以Datum类型）返回指定列的值。

这个函数不会为该 datum 分配新空间。在传引用数据类型的情况下，返回值将是一个被传递的指针。

### 参数

HeapTuple row

要检查的输入行

TupleDesc rowdesc

输入行描述

int colnumber

列号（从 1 开始计）

bool \* isnull

列中是否为空值的标志

### 返回值

该列的二进制值会被返回。如果该列为空值，由isnull 指向的变量将被设置为真，否则会被设置为假。

错误时SPI\_result会被设置成 SPI\_ERROR\_NOATTRIBUTE。

## SPI\_gettype

SPI\_gettype — 返回指定列的数据类型名称

## Synopsis

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

## 描述

SPI\_gettype返回该指定列的数据类型名称的拷贝（当你不再需要该拷贝后，可以使用pfree 释放它）。

## 参数

TupleDesc rowdesc

输入行描述

int colnumber

列号（从 1 开始计）

## 返回值

指定列的数据类型名称，或者在错误时返回NULL。  
SPI\_ERROR\_NOATTRIBUTE。

错误时SPI\_result会被设置成

## SPI\_gettypeid

SPI\_gettypeid — 返回指定列的数据类型的OID

### Synopsis

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

### 描述

SPI\_gettypeid返回该指定列的数据类型的 OID。

### 参数

TupleDesc rowdesc

    输入行描述

int colnumber

    列号（从 1 开始计）

### 返回值

指定列的数据类型的OID，或者出错时返回 InvalidOid。出错时，SPI\_result会被设置成 SPI\_ERROR\_NOATTRIBUTE。

## SPI\_getrelname

SPI\_getrelname — 返回指定关系的名称

### Synopsis

```
char * SPI_getrelname(Relation rel)
```

### 描述

SPI\_getrelname返回该指定关系的名称 的拷贝（当你不再需要该拷贝后，可以使用pfree 释放它）。

### 参数

Relation rel

输入关系

### 返回值

指定关系的名称。



## SPI\_getnspname

SPI\_getnspname — 返回指定关系的名字空间

## Synopsis

```
char * SPI_getnspname(Relation rel)
```

## 描述

SPI\_getnspname返回指定 关系所属的名字空间的名称拷贝。这等效 于该关系的模式。当你用完这个函数的返回值后，应该调用 pfree释放它。

## 参数

Relation rel

输入关系

## 返回值

指定关系的名字空间的名称。

## SPI\_result\_code\_string

SPI\_result\_code\_string — return error code as string

### Synopsis

```
const char * SPI_result_code_string(int code);
```

### 简介

SPI\_result\_code\_string返回之前各种SPI函数返回的或者存储在SPI\_result中的结果代码的字符串表示。

### 参数

int code

结果代码

### 返回值

结果代码的字符串表示。

## 43.3. 内存管理

LightDB 在内存上下文中分配内存，内存上下文为管理 在多个不同位置、具有不同生存时间需要的分配提供了一种便捷的方法。 销毁一个上下文会释放所有在其中分配的内存。因此不必跟踪单个对象来 避免内存泄露，而是只需要管理数量相对较少的上下文即可。 palloc和相关的函数可以从“当前”上下文中分配内存。

SPI\_connect创建一个新的内存上下文并且让它 成为当前上下文。SPI\_finish恢复之前的当前上下 文并且销毁由SPI\_connect创建的内存上下文。 这些动作确保在你的C函数中分配的内存存在C函数退出时被回收，从而避免内存 泄露。

不过，如果你的C函数需要返回一个在已分配内存中的对象（例如一个 传引用数据类型的值），你不能使用palloc 分配内存，或者说至少不能在连接到 SPI 时这样做。如果你试着这样 做，该对象会被SPI\_finish接触分配，那么 你的C函数将无法可靠地工作。要解决这个问题，应使用 SPI\_palloc来为要返回的对象分配内存。 SPI\_palloc会在 “上层执行器上下文”中分配内存，也就是当 SPI\_connect被调用时的当前内存上下文， 它才是从你的C函数中返回的值最适合的上下文。这一节中描述的几个其他实用函数也会返回在上层执行器上下文中创建的对象。

当SPI\_connect被调用时，这个C函数的私有 上下文（由SPI\_connect）会被作为当前上 下文。所有用palloc、 repalloc或者 SPI 功能函数（除了这个小节中描述的例外）分配的内存都在这个上下文中。 当一个C函数从SPI管理器断开连接时（通过 SPI\_finish），当前上下文被恢复到上层的 执行器上下文，并且在该过程的内存上下文中分配的内存都会被释放， 之后再不能被使用。

## SPI\_palloc

SPI\_palloc — 在上层执行器上下文中分配内存

### Synopsis

```
void * SPI_palloc(Size size)
```

### 描述

SPI\_palloc在上层的执行器上下文中分配内存。

这个函数只能在连接到SPI时使用。否则，它会报出错误。

### 参数

Size size

要分配的存储空间大小（以字节计）

### 返回值

指向具有指定大小的新存储空间的指针

## SPI\_realloc

SPI\_realloc — 在上层执行器上下文中重分配内存

### Synopsis

```
void * SPI_realloc(void * pointer, Size size)
```

### 描述

SPI\_realloc改变之前用SPI\_malloc 分配的内存段的大小。

这个函数不再和普通的realloc相区别。保留它只是为了对现有代码保持向后兼容。

### 参数

void \* pointer

指向要改变的现有存储空间的指针

Size size

要分配的存储空间大小（以字节计）

### 返回值

指向具有指定大小的新存储空间的指针，现有区域的内容会被复制到其中

## SPI\_pfree

SPI\_pfree — 在上层执行器上下文中释放内存

## Synopsis

```
void SPI_pfree(void * pointer)
```

## 描述

SPI\_pfree释放之前使用 SPI\_palloc或者 SPI\_realloc分配的内存。

这个函数不再和普通的pfree相区别。保留它只是为了对现有代码保持向后兼容。

## 参数

```
void * pointer
```

指向要释放的现有存储空间的指针

## SPI\_copytuple

SPI\_copytuple — 在上层执行器上下文中创建一行的拷贝

## Synopsis

```
HeapTuple SPI_copytuple(HeapTuple row)
```

## 描述

SPI\_copytuple在上层执行器上下文中为一行创建一份拷贝。这通常被用来从一个触发器中返回一个被修改的行。在一个被声明为返回组合类型的函数中，应使用 SPI\_returntuple。

这个函数只能在连接到SPI时使用。否则，它会返回NULL并且把SPI\_result设置为SPI\_ERROR\_UNCONNECTED。

## 参数

HeapTuple row

要拷贝的行

## 返回值

被拷贝的行，或者在出错时返回NULL（错误的内容请参考SPI\_result）

## SPI\_returntuple

SPI\_returntuple — 准备把一个元组返回为一个 Datum

## Synopsis

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

## 描述

SPI\_returntuple为一个行在上层执行器上下文中 `row` 创建一个拷贝，把它以一种行类型Datum的形式返回。被 `row` 返回的指针只需要在返回前通过PointerGetDatum 被转换成Datum。

这个函数只能在连接到SPI时使用。否则，它会返回NULL并且把SPI\_result设置为SPI\_ERROR\_UNCONNECTED。

注意这应该被用于声明为要返回组合类型的函数。它不能用于触发器，在 `row` 触发器中应使用SPI\_copytuple来返回一个被修改的行。

## 参数

HeapTuple row

要被拷贝的行

TupleDesc rowdesc

行的描述符（对大部分有效的缓存，每次都传递相同的描述符）

## 返回值

指向被拷贝行的HeapTupleHeader，或者在出错时返回NULL（错误的内容请参考SPI\_result）

## SPI\_modifytuple

SPI\_modifytuple — 通过替换一个给定行的选定域来创建一行

### Synopsis

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

### 描述

SPI\_modifytuple创建一个新行，其中选定的列 用新值替代，其他列则从输入行中拷贝。输入行本身不被修改。新行被返回在上层的执行器上下文中。

这个函数只能在连接到SPI时使用。否则，它会返回NULL并且把SPI\_result设置为SPI\_ERROR\_UNCONNECTED。

### 参数

Relation rel

只被用作该行的行描述符的来源（传递一个关系而不是 一个行描述符是一种令人头痛的设计）。

HeapTuple row

要被修改的行

int ncols

要被修改的列数

int \* colnum

一个长度为ncols的数组，包含了要被修改的列号（列号从 1 开始）

Datum \* values

一个长度为ncols的数组，包含了指定列的新值

const char \* nulls

一个长度为ncols的数组，描述哪些新值为空值

如果nulls为NULL，那么 SPI\_modifytuple假定没有新值为空值。否则， 如果对应的新值为非空，nulls数组的每一项都应该是' '；而如果对应的新值为空值则为'n'（在后一种情况中，对应的values项中的新值无关紧要）。注意nulls不是一个文本字符串，只是一个 数组：它不需要一个'\0'终止符。

### 返回值

修改过的新行，它被分配在上层的执行器上下文中，或者在出错时返回NULL（错误的内容请参考SPI\_result）

出错时，SPI\_result被设置如下：



SPI\_ERROR\_ARGUMENT

如果rel为NULL，或者 row为NULL，或者ncols 小于等于 0，或者colnum为NULL，或者values为NULL。

SPI\_ERROR\_NOATTRIBUTE

如果colnum包含一个无效的列号（小于等于 0 或者大于 row中的列数）。

SPI\_ERROR\_UNCONNECTED

如果SPI不是活跃状态

## SPI\_freetuple

SPI\_freetuple — 释放一个在上层执行器上下文中分配的行

## Synopsis

```
void SPI_freetuple(HeapTuple row)
```

## 描述

SPI\_freetuple释放之前在上层执行器上下文中 分配的一个行。

这个函数不再和普通的heap\_freetuple相区别。 保留它只是为了对现有代码保持向后兼容。

## 参数

HeapTuple row

要释放的行

## SPI\_freetuptable

SPI\_freetuptable — 释放一个由SPI\_execute 或者类似函数创建的行集合

### Synopsis

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

### 描述

SPI\_freetuptable释放一个由之前的 SPI 命令 执行函数（例如SPI\_execute）创建的行集合。因此， 调用这个函数时，常常使用SPI\_tuptable作为 参数。

如果一个使用SPI的C函数需要执行多个命令并且不想保留早期命令的结果，这个 函数就有用了。注意，SPI\_finish会释放任何还未释放的 行集合。还有，如果在一个使用SPI的C函数的执行中开始了一个子事务并且后来 被中止，SPI 会自动释放该子事务运行期间创建的任何行集合。

从LightDB 9.3 开始， SPI\_freetuptable包含了保护逻辑以避免对于同 一行集的重复删除请求。在以前的发布中，重复的删除将会导致崩溃。

### 参数

SPITupleTable \* tuptable

要释放的行集的指针，NULL 表示什么也不做

## SPI\_freeplan

SPI\_freeplan — 释放一个之前保存的预备语句

## Synopsis

```
int SPI_freeplan(SPIPlanPtr plan)
```

## 描述

SPI\_freeplan释放一个之前由 SPI\_prepare返回的或者由 SPI\_keepplan、SPI\_saveplan 保存的预备语句。

## 参数

SPIPlanPtr plan

要释放的语句的指针

## 返回值

成功返回 0；如果plan为 NULL或无效则返回 SPI\_ERROR\_ARGUMENT

## 43.4. 事务管理

不能通过SPI\_execute这样的SPI函数运行COMMIT和ROLLBACK之类的事务控制命令。不过，也有单独的接口函数允许通过SPI进行事务控制。

如果不考虑被调用的上下文，在任意的用户定义的可从SQL调用的函数中开始以及结束事务通常并不是安全和明智的。例如，一个事务位于一个函数内，而该函数是某个SQL命令中的一个复杂SQL表达式的一部分，这样的事务有可能会隐藏的内部错误或者崩溃。这里介绍的接口函数的主要目的是被过程语言的实现用于支持在CALL命令调用的SQL层过程中进行事务管理，同时把CALL调用的上下文也加以考虑。用C实现的使用SPI的过程可以实现同样的逻辑，但是其细节超出了这份文档的范围。

## SPI\_commit

SPI\_commit, SPI\_commit\_and\_chain — commit the current transaction

## Synopsis

```
void SPI_commit(void)
```

```
void SPI_commit_and_chain(void)
```

## 简介

SPI\_commit提交当前事务。它近似等效于运行SQL命令COMMIT。在一个事务被提交后，在进一步的数据库动作被执行前必须使用SPI\_start\_transaction开始一个新的事务。

SPI\_commit\_and\_chain是相同的，事务特征与刚刚完成的事务相同的新事务会立即启动，类似 SQL 命令 COMMIT AND CHAIN。

只有当SPI连接已经在对SPI\_connect\_ext的调用中被设置为非原子的情况下才能执行这些函数。

## SPI\_rollback

SPI\_rollback, SPI\_rollback\_and\_chain — abort the current transaction

## Synopsis

```
void SPI_rollback(void)
```

```
void SPI_rollback_and_chain(void)
```

## 简介

SPI\_rollback回滚当前事务。它近似等效于运行SQL命令ROLLBACK。在一个事务被回滚后，在进一步的数据库动作被执行前必须使用SPI\_start\_transaction开始一个新的事务。

SPI\_rollback\_and\_chain是相同的，事务特征与刚刚完成的事务相同的新事务会立即启动，类似 SQL 命令 ROLLBACK AND CHAIN。

只有当SPI连接已经在对SPI\_connect\_ext的调用中被设置为非原子的情况下才能执行这些函数。

## SPI\_start\_transaction

SPI\_start\_transaction — start a new transaction

### Synopsis

```
void SPI_start_transaction(void)
```

### 简介

SPI\_start\_transaction开始一个新事务。它只能在SPI\_commit或SPI\_rollback之后被调用，因为在那时没有事务在活动。通常，当一个使用SPI的过程被调用时，会有一个事务已处于活跃状态，因此在关闭当前事务之前尝试启动另一个事务将会导致错误。

## 43.5. 数据改变的可见性

下列规则主导了使用 SPI 的函数（或者任何其他 C 函数）中数据改变的可见性：

- 在一个 SQL 命令的执行期间，该命令所作的任何数据更改对该命令本身 是不可见的。例如，在

```
INSERT INTO a SELECT * FROM a;
```

中，被插入的行对SELECT部分不可见。

- 一个命令 C 所作的更改对所有在 C 之后开始的命令可见，不管它们是否在 C 之中（在 C 的执行期间）开始还是在 C 完成之后开始。
- 在一个 SQL 命令（或者一个普通函数或者触发器）调用的函数内通过 SPI 执行的命令遵循以上哪条规则取决于传递给 SPI 的读/写标志。以 只读模式执行的命令遵循第一条规则：它们不能看到调用它们的命令的 改变。在读写模式中执行的命令遵循第二条规则：它们能看见目前为止 所有的改变。
- 所有的标准过程语言会基于函数的易变性属性设置 SPI 读写模式。STABLE和IMMUTABLE函数的命令会以 只读模式完成，而VOLATILE函数的命令会以读写模式 完成。虽然 C 函数的作者可以违反这种习惯，但是最好不要那样做。

下一节包含一个关于这些规则应用的例子：

## 43.6. 例子

这一节包含了 SPI 用法的一个非常简单的例子。C函数 execq用一个 SQL 命令作为其第一个参数 并且用一个行计数作为第二个参数，使用 SPI\_exec执行该命令并且返回被该命令 处理过的行的数量。你可以在源代码树的 src/test/regress/regress.c和 spi模块中找到 SPI 的更复杂的例子。

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(execq);
```

```

Datum
execq(PG_FUNCTION_ARGS)
{
    char *command;
    int cnt;
    int ret;
    uint64 proc;

    /* 把给定的文本对象转换成一个 C 字符串 */
    command = text_to_cstring(PG_GETARG_TEXT_PP(0));
    cnt = PG_GETARG_INT32(1);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;

    /*
     * 如果取出了一些行，通过 elog(INFO) 打印它们。
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        SPITupleTable *tuptable = SPI_tuptable;
        TupleDesc tupdesc = tuptable->tupdesc;
        char buf[8192];
        uint64 j;

        for (j = 0; j < tuptable->numvals; j++)
        {
            HeapTuple tuple = tuptable->vals[j];
            int i;

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : "|");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    PG_RETURN_INT64(proc);
}

```

在把该函数编译到一个共享库中（详见Section 36.10.5 之后，这样声明该函数：

```

CREATE FUNCTION execq(text, integer) RETURNS int8
AS 'filename'
LANGUAGE C STRICT;

```

下面是一个会话实例：

```

=> SELECT execq('CREATE TABLE a (x integer)', 0);
execq

```



```

-----
      0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO: EXECQ: 0    -- inserted by execq
INFO: EXECQ: 1    -- returned by execq and inserted by upper INSERT

execq
-----
      2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
execq
-----
      1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO: EXECQ: 0
INFO: EXECQ: 1
INFO: EXECQ: 2    -- 0 + 2, 按照所指定的, 只有一行被插入

execq
-----
      3                -- 10 只是最大值, 3 是实际的行数
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
x
-----
      1                -- 没有行在 a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO: EXECQ: 1
INSERT 0 1
=> SELECT * FROM a;
x
-----
      1
      2                -- 有一行在 in a + 1
(2 rows)

-- 这证明了数据改变可见性规则:

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 1
INFO: EXECQ: 2

```

```
INFO: EXECQ: 2
INSERT 0 2
=> SELECT * FROM a;
 x
----
 1
 2
 2          -- 2 行 * 1 (第一行中的 x)
 6          -- 3 rows (2 + 1 被插入) * 2 (第二行中的 x)
(4 rows)    ~~~~~

          不同调用中 execq() 的行可见性
```

---

## Chapter 44. 后台工作者进程

LightDB可以被扩展来在独立进程中运行用户提供的代码。这种进程被lightdb启动、停止和监控，这使它们的生命期与服务器的状态紧密联系。这些进程具有选项可以挂接上LightDB的共享内存区域，并且可以从内部连接到数据库。它们也可以连续地运行多个事务，就像一个正常的被客户端连接的服务器进程。同样，通过链接到libpq，它们可以连接到服务器并像一个正常客户端应用工作。

### Warning

在使用后台工作者进程时具有相当大的鲁棒性和安全性风险，因为它们由C语言编写，对数据具有无限制的访问权。希望使用包括后台工作者进程在内的模块的管理人员必须要极度小心。只有仔细审计过的模块才会被允许运行后台工作者进程。

通过将模块名放在shared\_preload\_libraries中，可以在LightDB被启动时初始化后台工作者。一个希望运行后台工作者的模块需要通过在其\_PG\_init()函数中调用RegisterBackgroundWorker(BackgroundWorker\*worker)来注册它。也可以在系统启动后通过调用RegisterDynamicBackgroundWorker(BackgroundWorker\*worker, BackgroundWorkerHandle\*\*handle)来启动后台工作者。不像只能通过postmaster调用的RegisterBackgroundWorker, RegisterDynamicBackgroundWorker必须从一个常规后端或其他后台工作者调用。

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
{
    char        bgw_name[BGW_MAXLEN];
    char        bgw_type[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time; /* in seconds, or BGW_NEVER_RESTART */
    char        bgw_library_name[BGW_MAXLEN];
    char        bgw_function_name[BGW_MAXLEN];
    Datum      bgw_main_arg;
    char        bgw_extra[BGW_EXTRALEN];
    int         bgw_notify_pid;
} BackgroundWorker;
```

bgw\_name和bgw\_type是要被用在日志消息、进程列表和类似环境中的字符串。对于同种类型的所有后台工作者，bgw\_type应该相同，例如这样才能将进程列表中的这些工作组分组。另一方面，bgw\_name可以包含有关特定进程的额外信息（通常，bgw\_name中的字符串在某种程度上也会包含类型，但是并没有严格的要求）。

bgw\_flags是一个按位与的位掩码，它用于指示模块想要的功能。可能的值是：

**BGWORKER\_SHMEM\_ACCESS**

请求共享内存访问。没有共享内存使用权的工作者不能访问任何的LightDB共享数据结构，例如重量级或者轻量级锁、共享缓冲区以及该工作者本身想要创建和使用的任何自定义数据结构。

**BGWORKER\_BACKEND\_DATABASE\_CONNECTION**

请求建立数据库连接的能力，这样它后面可以通过建立起的连接运行事务和查询。一个使用BGWORKER\_BACKEND\_DATABASE\_CONNECTION来连接一个数据库的后台工作者也必须使用BGWORKER\_SHMEM\_ACCESS挂接到共享内存，否则工作者启动将会失败。

`bgw_start_time`是服务器状态，在该状态中lightdb会启动该进程，它可以是`BgWorkerStart_PostmasterStart`（在lightdb本身完成初始化之后立即启动，这种进程不能使用数据库连接）、`BgWorkerStart_ConsistentState`（当一个热后备中达到一个一致性状态之后立即启动，允许进程连接到数据库并运行只读查询）和`BgWorkerStart_RecoveryFinished`（在系统进入到正常读写状态后立即启动）之一。注意后两种值在服务器不是一个热后备的情况下是等同的。注意这种设置仅仅表示何时启动进程，当一个不同状态到达时它们不会停止。

`bgw_restart_time`是时间间隔，以秒计；是在进程崩溃情况下，lightdb重新启动进程之前等待的时间间隔。它可以是任何正值，或者`BGW_NEVER_RESTART`，表示在出现崩溃后不重启进程。

`bgw_library_name`是应该在其中定位后台工作者初始入口点的库名称。所指的库将被工作者进程动态载入并且`bgw_function_name`将被用来标识要调用的函数。如果从核心代码载入一个函数，这必须被设置为“lightdb”。

`bgw_function_name`是一个动态载入库中的一个函数名，该函数将被用作一个新后台工作者的初始入口点。

`bgw_main_arg`是后台工作者主函数的Datum参数。这个主函数应该有一个单一的Datum类型的参数，并且返回void。`bgw_main_arg`将被作为参数传递。此外，全局变量`MyBgworkerEntry`指向注册时传入的`BackgroundWorker`结构的一份拷贝，工作者会发现检查这个结构会很有用。

在Windows（以及任何定义了`EXEC_BACKEND`的地方）上或者动态后台工作者中，用引用的方式传递Datum是不安全的，只有传值才安全。如果要求一个参数，最安全的方式是传递一个`int32`或者其他的小型值，并且把它当做共享内存中分配的一个数组的索引来使用。如果被传递的是一个`cstring`或者`text`这样的值，那么在新的后台工作者进程中该指针将不会有效。

`bgw_extra`可以包含要传递给后台工作者的额外数据。与`bgw_main_arg`不同，这个数据不会被作为一个参数传递给工作者的主函数，而是按照上面所述通过`MyBgworkerEntry`来访问。

`bgw_notify_pid`是一个LightDB后端进程的PID，当后台工作者进程启动或者退出时，`postmaster`会向这个PID所指的进程发送SIGUSR1。对于在`postmaster`启动时注册的工作者，它应该为0；或者注册该工作者的后端不希望等待该工作者启动时，它也应该为0。否则，它应该被初始化为`MyProcPid`。

一旦运行起来，进程可以通过调用`BackgroundWorkerInitializeConnection(char *dbname, char *username)`或者`BackgroundWorkerInitializeConnectionByOid(Oid dboid, Oid useroid)`来连接到一个数据库。这使得该进程可以使用SPI接口运行事务和查询。如果`dbname`为NULL或者`dboid`为`InvalidOid`，该会话没有连接到任何特定数据库，但共享的目录可以被访问。如果`username`为NULL或者`useroid`为`InvalidOid`，该进程将在`initdb`阶段创建的超级用户身份运行。如果`BGWORKER_BYPASS_ALLOWCONN`被指定为`flags`，就可以绕过该限制连接不允许用户连接的数据库。在每一个后台进程中，只能调用两者之一，并且只能调用一次，所以不可能切换数据库。

当控制到达后台工作者的主函数时，信号初始会被阻塞，并且必须被它解除阻塞。这是为了允许进程自定义它的信号处理器。在新进程中可以通过调用`BackgroundWorkerUnblockSignals`来解除对信号的阻塞，还可以通过调用`BackgroundWorkerBlockSignals`来阻塞信号。

如果一个后台工作者的`bgw_restart_time`被配置为`BGW_NEVER_RESTART`，或者它退出时的退出码为0，又或者它是被`TerminateBackgroundWorker`所终止，它将会被`postmaster`在退出时自动解除注册。否则，它将在等待通过`bgw_restart_time`配置的时间段之后被重新启动，或者在`postmaster`因为一次后端失败重新初始化集簇时立刻被重启。需要临时禁止执行的后端应该使用可中断的休眠而不是退出，这可以通过调用`WaitLatch()`实现。调用该函数时要确保`WL_POSTMASTER_DEATH`标志被设置，并且验证在lightdb本身被终止的紧急情况下产生的快速退出返回码。

当一个后台工作者是通过RegisterDynamicBackgroundWorker函数注册时，后端可以执行该注册以获得有关该工作者的状态信息。希望这样做的后端应该把一个BackgroundWorkerHandle的地址作为第二个参数传递给RegisterDynamicBackgroundWorker。如果工作者被成功地注册，这个指针将被用一个非透明句柄初始化，它之后会被传递给GetBackgroundWorkerPid (BackgroundWorkerHandle \*, pid\_t \*)或者TerminateBackgroundWorker (BackgroundWorkerHandle \*)。GetBackgroundWorkerPid可以被用来测试工作者的状态：返回值为BGWH\_NOT\_YET\_STARTED表示该工作者还未被postmaster启动；BGWH\_STOPPED表示它已经被启动但是不再运行；而BGWH\_STARTED表示它正在运行。在最后一种情况下，PID也将被通过第二个参数返回。TerminateBackgroundWorker导致postmaster发送SIGTERM给工作者（如果它在运行），并且在它不再运行时尽快解除注册。

在某些情况下，一个注册后台工作者的进程可能希望等待该工作者启动起来。其实现方式是：把bgw\_notify\_pid初始化成MyProcPid并且接着把注册时得到的BackgroundWorkerHandle地址传递给WaitForBackgroundWorkerStartup (BackgroundWorkerHandle \*handle, pid\_t \*)函数。这个函数将阻塞直到postmaster已经尝试启动该后台工作者，或者直到postmaster死亡。如果后台工作者正在运行，返回值将是BGWH\_STARTED，并且其PID将被写入到所提供的地址。否则，返回值将是BGWH\_STOPPED或者BGWH\_POSTMASTER\_DIED。

进程也可以等待一个后台工作者关闭，方法是使用WaitForBackgroundWorkerShutdown (BackgroundWorkerHandle \*handle)函数并且传入注册时得到的BackgroundWorkerHandle \*。这个函数将阻塞直至后台工作者退出或者postmaster死掉。当后台工作者退出时，返回值是BGWH\_STOPPED，如果postmaster死掉则会返回BGWH\_POSTMASTER\_DIED。

后台工作者可以发送异步通知消息，可以通过SPI的NOTIFY命令，或者直接通过Async\_Notify()。这样的通知将在事务提交的时候发送。后台工作者不应该使用LISTEN命令接收异步通知，因为工作者没有可以使用这些通知的基础架构。

src/test/modules/worker\_spi模块包含了一个实例，它展示了一些有用的技巧。

注册的后台工作者的最大数量由max\_worker\_processes限制。

---

## Chapter 45. 逻辑解码

LightDB 提供了方法将所执行的修改通过 SQL 以流的方式传送给外部消费者。 这种功能可以被用于多种目的，包括复制方案以及审计。

在流中被送出的更改通过逻辑复制槽标识。

流式传输这些更改的格式由使用的输出插件决定。LightDB 发布中包括了一个例子插件。可以编写额外的插件来扩展可用的格式选择，而无需修改任何核心代码。每一个输出插件都能访问每一个由 INSERT 产生的新行以及每一个由 UPDATE 创建的新行版本。UPDATE 和 DELETE 的旧行版本的可用性取决于配置的复制标识（见 REPLICA IDENTITY）。

可以通过流复制协议（见 Section 49. 和 Section 45.3 或者通过 SQL 调用函数（Section 45.4 来接收流式传送的更改。也可以编写额外的接收复制槽输出的模块而无需修改核心代码（Section 45.7）。

### 45.1. 逻辑解码的例子

下面的例子演示了使用 SQL 接口控制逻辑解码。

在你使用逻辑解码之前，你必须设置 wal\_level 为 logical，并且 max\_replication\_slots 必须至少被设置为 1。然后，你应该作为一个超级用户连接到目标数据库（在下面 的例子中是 postgres）。

```
lightdb@postgres=# -- 使用输出插件 'test_decoding' 创建一个名为 'regression_slot' 的槽
```

```
lightdb@postgres=# SELECT * FROM
pg_create_logical_replication_slot('regression_slot', 'test_decoding');
 slot_name | lsn
-----+-----
 regression_slot | 0/16B1970
(1 row)
```

```
lightdb@postgres=# SELECT slot_name, plugin, slot_type, database, active,
restart_lsn, confirmed_flush_lsn FROM pg_replication_slots;
 slot_name | plugin | slot_type | database | active | restart_lsn |
confirmed_flush_lsn
-----+-----+-----+-----+-----+-----+-----
 regression_slot | test_decoding | logical | postgres | f | 0/16A4408 |
0/16A4440
(1 row)
```

```
lightdb@postgres=# -- 目前还看不到更改
lightdb@postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot',
NULL, NULL);
 lsn | xid | data
-----+-----+-----
(0 rows)
```

```
lightdb@postgres=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE
```

```
lightdb@postgres=# -- DDL 没有被复制，因此你将看到的只有事务
lightdb@postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot',
NULL, NULL);
 lsn | xid | data
```

```

-----+-----+-----
0/BA2DA58 | 10297 | BEGIN 10297
0/BA5A5A0 | 10297 | COMMIT 10297
(2 rows)

lightdb@postgres=# -- 一旦读到更改，它们会被消费掉并且不会在一个后续调用中被发出：
lightdb@postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot',
NULL, NULL);
 lsn | xid | data
-----+-----+-----
(0 rows)

lightdb@postgres=# BEGIN;
lightdb@postgres=# INSERT INTO data(data) VALUES('1');
lightdb@postgres=# INSERT INTO data(data) VALUES('2');
lightdb@postgres=# COMMIT;

lightdb@postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot',
NULL, NULL);
 lsn | xid | data
-----+-----+-----
0/BA5A688 | 10298 | BEGIN 10298
0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1 data[text]:'1'
0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2 data[text]:'2'
0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)

lightdb@postgres=# INSERT INTO data(data) VALUES('3');

lightdb@postgres=# -- 你也可以不消费更改而在更改流中先看一看
lightdb@postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot',
NULL, NULL);
 lsn | xid | data
-----+-----+-----
0/BA5A8E0 | 10299 | BEGIN 10299
0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

lightdb@postgres=# -- 接下来对 pg_logical_slot_peek_changes() 的调用再次返回相同的更改
lightdb@postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot',
NULL, NULL);
 lsn | xid | data
-----+-----+-----
0/BA5A8E0 | 10299 | BEGIN 10299
0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

lightdb@postgres=# -- 可以向输出插件传递选项来影响格式化
lightdb@postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot',
NULL, NULL, 'include-timestamp', 'on');
 lsn | xid | data
-----+-----+-----
0/BA5A8E0 | 10299 | BEGIN 10299
0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'

```

```
0/BA5A990 | 10299 | COMMIT 10299 (at 2017-05-10 12:07:21.272494-04)
(3 rows)
```

```
lightdb@postgres=# -- 当不再需要一个槽后记住销毁它以停止消耗服务器资源:
lightdb@postgres=# SELECT pg_drop_replication_slot('regression_slot');
pg_drop_replication_slot
```

```
-----
(1 row)
```

下面的例子展示了如何在流复制协议上使用 LightDB 发布所包括的程序 `lt_recvlogical` 来控制逻辑解码。这要求设置客户端认证以允许复制连接（见Section 24.2.5.1），并且把 `max_wal_senders` 设置成足够高以允许一个额外的连接。

```
$ lt_recvlogical -d postgres --slot=test --create-slot
$ lt_recvlogical -d postgres --slot=test --start -f -
Control+Z
$ ltsql -d postgres -c "INSERT INTO data(data) VALUES('4');"
$ fg
BEGIN 693
table public.data: INSERT: id[integer]:4 data[text]:'4'
COMMIT 693
Control+C
$ lt_recvlogical -d postgres --slot=test --drop-slot
```

## 45.2. 逻辑解码概念

### 45.2.1. 逻辑解码

逻辑解码是一种将对数据库表的所有持久更改抽取到一种清晰、易于理解的格式 的处理，这种技术允许在不了解数据库内部状态的详细知识的前提下解释该格式。

在LightDB中，逻辑解码通过解码 预写式日志的内容来实现，预写式日志描述了存储 层面上的更改，而逻辑解码则会把更改解码成一种应用相关的形式，例如一个元组 流或者 SQL 语句流。

### 45.2.2. 复制槽

在逻辑复制的环境下，一个槽表示一个更改流，这些更改可以在客户机上以它们在原服务器上产生的顺序被重播。每一个流从一个单一数据库中流式传送更改序列。

#### Note

LightDB也有流复制槽（见Section 24.2.5，但是它们的使用有所不同。

一个复制槽在一个LightDB集簇的所有数据库之间具有一个唯一的标识符。槽在使用它们的连接之间保持独立并且 对于崩溃是安全的。

在常规操作中，一个逻辑槽只会把每次更改发出一次。 只有在检查点时才会持久化每一个槽的当前位置，因此如果发生崩溃，槽可能会回到一个较早的 LSN，这会导致服务器重启时再次发送最近的更改。 逻辑解码客户端负责避免多次处理同一消息导致的副作用。 客户端可能会希望在解码时记录它们看到的最新的 LSN，并且跳过任何从该 LSN 解码得到的重复数据或者（使用复制协议时的）请求，而不是让服务器来决定开始点。 复制进度跟踪特性就是为此服务的，请参考复制源头。



对于同一个数据库可能会存在多个独立的槽。每一个槽有自己的状态，允许不同的消费者从该数据库的更改流中的不同点开始接收更改。对于大多数应用，每一个消费者都将要求一个单独的槽。

逻辑复制槽完全不知道接收者的状态。甚至可能会有多个不同的接收者在不同时间使用同一个槽，它们将只是从上一个接收者停止消费更改的地方开始得到更改。但在任一给定时刻，只有一个接收者可以从一个槽中消费更改。

### Caution

复制槽可以在崩溃时保持，并且不知道其消费者的状态。即便没有连接使用它们，它们也将阻止移除所需的资源。这会消耗存储，因为只要还有一个复制槽需要，WAL 和来自于系统目录的行就不能被VACUUM移除。在极端情况下这会导致数据库关闭以防止事务ID回卷（见Section 22.1.5）。因此如果不再需要一个槽，那就应该删除它。

## 45.2.3. 输出插件

输出插件将数据从预写式日志的内部表示转换成复制槽的消费者所需的格式。

## 45.2.4. 导出快照

当使用流复制接口创建一个新的复制槽时（见CREATE\_REPLICATION\_SLOT），一个快照将被导出（见Section 9.27.5，在它所显示的数据库状态之后所有的更改都将被包括在更改流中。通过使用SET TRANSACTION SNAPSHOT读取槽被创建时的数据库状态，这可以用来创建一个新的复制。然后这个事务可以被用来及时转储那一点的数据库状态，它后来可以槽的内容更新而不丢失任何更改。

并非总能够创建快照。特别是在连接到热备时，快照创建将会失败。不要求快照导出的应用可以用NOEXPORT\_SNAPSHOT选项来抑制它。option.

## 45.3. 流复制协议接口

命令

- CREATE\_REPLICATION\_SLOT slot\_name LOGICAL output\_plugin
- DROP\_REPLICATION\_SLOT slot\_name [ WAIT ]
- START\_REPLICATION SLOT slot\_name LOGICAL ...

被用来创建、删除以及流式传送一个复制槽。这些命令只能在一个复制连接上使用。它们不同通过SQL使用。这些命令的详情请见Section 49.4

命令lt\_recvlogical可以被用来控制一个流复制连接上的逻辑解码（它在内部使用上述命令）。

## 45.4. 逻辑解码的 SQL 接口

与逻辑解码互动的SQL层API详见Section 9.27.6

同步复制（见Section 24.2.8）只在使用流复制接口的复制槽上支持。函数接口以及额外的、非核心的接口不支持同步复制。

## 45.5. 与逻辑解码相关的系统目录

`pg_replication_slots` 视图和`pg_stat_replication` 视图分别提供了有关复制槽和流复制连接的当前状态的信息。这些视图适用于物理和逻辑复制。

## 45.6. 逻辑解码输出插件

可以在 LightDB 源码树的 `contrib/test_decoding` 子目录中找到一个输出插件的例子。

### 45.6.1. 初始化函数

一个输出插件是通过动态载入一个以输出插件名称作为基础名称的共享库来载入的。将使用普通的库搜索路径来定位该库。为了提供所要求的输出插件回调并且指示该库确实是一个输出插件，需要提供一个名为 `_PG_output_plugin_init` 的函数。这个函数会被传入一个结构，其中被填充了各个动作的回调函数指针。

```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeTruncateCB truncate_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeMessageCB message_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit) (struct OutputPluginCallbacks *cb);
```

回调函数`begin_cb`、`change_cb`以及`commit_cb`是必需的，而`startup_cb`、`filter_by_origin_cb`、`truncate_cb`和`shutdown_cb`是可选的。如果没有设置`truncate_cb`但是要对一个TRUNCATE进行编码，则该动作将被忽略。

### 45.6.2. 能力

要解码、格式化并且输出更改，输出插件可以使用大部分后端的标准功能，包括调用输出函数。只要访问的关系是`initdb`在 `pg_catalog`模式中创建的或者被使用

```
ALTER TABLE user_catalog_table SET (user_catalog_table = true);
CREATE TABLE another_catalog_table(data text) WITH (user_catalog_table = true);
```

标记为用户提供的系统表，就允许对关系的只读访问。任何导致事务 ID 分配的动作 都被禁止。其中包括写表、执行 DDL 更改以及调用`pg_current_xact_id()`。

### 45.6.3. 输出模式

输出插件回调可以以近乎任意格式向消费者传递数据。对于某些用例，例如通过 SQL 查看更改，以可能包含任何数据的数据类型（例如`bytea`）返回数据 可能会很麻烦。如果输出插件只输出服务器编码的文本数据，它可以在 启动回调中通过把`OutputPluginOptions.output_type` 置为`OUTPUT_PLUGIN_TEXTUAL_OUTPUT`替代`OUTPUT_PLUGIN_BINARY_OUTPUT`来声明这一点。在这种情况下， 所有的数据必须是属于服务器的编码，这样一个`text`数据就能包含它。在 启用了断言的编译中会检查这一点。

## 45.6.4. 输出插件回调

一个输出插件需要提供一些回调，它通过它们得到有关更改发生的通知。

并发事务以提交顺序被解码，并且只有属于特定事务的更改会在 `begin`和`commit`回调之间被解码。被显式 或隐式回滚的事务不会被解码。成功的检查点被折叠到包含它们的事务中，并且 保持它们在该事务中被执行的顺序。

### Note

只有已经被安全地刷入磁盘的事务将会被解码。当 `synchronous_commit`被设置为`off` 时，这会导致一个`COMMIT`在随后的 `pg_logical_slot_get_changes()`中不会立即被解码。

### 45.6.4.1. 启动回调

只要一个复制槽被创建或者被要求流式传送更改，可选的 `startup_cb`回调就会被调用，不管有多少更改准备输出。

```
typedef void (*LogicalDecodeStartupCB) (struct LogicalDecodingContext *ctx,
                                       OutputPluginOptions *options,
                                       bool is_init);
```

当复制槽被创建时，`is_init`参数将为真，否则为假。 `options`指向一个输出插件可以设置的选项 的结构：

```
typedef struct OutputPluginOptions
{
    OutputPluginOutputType output_type;
    bool receive_rewrites;
} OutputPluginOptions;
```

`output_type`必须被设置为 `OUTPUT_PLUGIN_TEXTUAL_OUTPUT` 或者 `OUTPUT_PLUGIN_BINARY_OUTPUT`。另见 Section 45.6.3 如果`receive_rewrites`为真，还将为在某些DDL操作期间的堆重写造成的更改调用输出插件。这些是处理DDL复制的插件感兴趣的事情，但是它们要求特殊的处理。

启动回调应该验证出现在 `ctx->output_plugin_options`中的选项。如果输出插件 需要有一个状态，它可以使用 `ctx->output_plugin_private`来存储之。

### 45.6.4.2. 关闭回调

只要一个之前活跃的复制槽不再使用，就会调用可选的 `shutdown_cb`回调，它可以被用来释放输出插件 私有的资源。该槽并不一定需要被删除，只要其中的流被停止即可。

```
typedef void (*LogicalDecodeShutdownCB) (struct LogicalDecodingContext *ctx);
```

### 45.6.4.3. 事务开始回调

只要一个已提交事务的开始动作被解码，就会调用必须提供的 `begin_cb`回调。被中止的事务及其内容不会被解码。

```
typedef void (*LogicalDecodeBeginCB) (struct LogicalDecodingContext *ctx,
```

```
ReorderBufferTXN *txn);
```

txn参数包含有关该事务的元信息，例如该事务被提交的时间戳以及该事务的XID。

#### 45.6.4.4. 事务结束回调

只要一个已提交事务的提交动作被解码，就会调用必须提供的commit\_cb回调。在此之前，如果有任何被修改的行，将为所有被修改的行调用change\_cb回调。

```
typedef void (*LogicalDecodeCommitCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       XLogRecPtr commit_lsn);
```

#### 45.6.4.5. 更改回调

对于一个事务中的每一个行修改，都将调用必须提供的change\_cb回调，这种修改可能是一个INSERT、UPDATE或者DELETE。即使原始命令一次修改了多行，该回调也会为其中的每一行调用一次。

```
typedef void (*LogicalDecodeChangeCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       Relation relation,
                                       ReorderBufferChange *change);
```

ctx和txn参数与begin\_cb和commit\_cb回调具有相同的内容，但是额外多出一个关系描述符relation指向该行所属的关系以及一个结构change描述被传入的行修改。

#### Note

只有没有被标记为“不做日志”（见UNLOGGED）并且非临时（见TEMPORARY or TEMP）的用户定义表中的更改才能用逻辑解码抽取。

#### 45.6.4.6. 截断回调

truncate\_cb回调会为一个TRUNCATE命令被调用。

```
typedef void (*LogicalDecodeTruncateCB) (struct LogicalDecodingContext *ctx,
                                          ReorderBufferTXN *txn,
                                          int nrelations,
                                          Relation relations[],
                                          ReorderBufferChange *change);
```

参数类似于change\_cb回调。不过，由于通过外键连接起来的表上的TRUNCATE动作需要一起被执行，这个回调会接收到一个关系的数组而不是单个关系。详情请见对TRUNCATE语句的介绍。

#### 45.6.4.7. 源过滤器回调

可选的filter\_by\_origin\_cb回调被用来决定从origin\_id重放的数据是否是输出插件感兴趣的数据。

```
typedef bool (*LogicalDecodeFilterByOriginCB) (struct LogicalDecodingContext
                                              *ctx,
                                              RepOriginId origin_id);
```

ctx参数具有和其他回调相同的内容。对这个回调只有复制源的信息可用。要标志传进来的节点上发生的更改是无关的，返回真，这会导致这些更改被过滤掉，否则返回假。对于被过滤掉的事务和更改将不会调用其他回调。

在实现级联或者多向复制方案时，这个回调可以派上用场。用源头过滤允许阻止在这样的设置下来回地复制同样的更改。虽然事务和更改也携带了有关源头的信息，通过这个回调过滤明显更有效些。

#### 45.6.4.8. 通用消息回调

只要一个逻辑解码消息被解码出来，可选的message\_cb回调就会被调用。

```
typedef void (*LogicalDecodeMessageCB) (struct LogicalDecodingContext *ctx,
                                         ReorderBufferTXN *txn,
                                         XLogRecPtr message_lsn,
                                         bool transactional,
                                         const char *prefix,
                                         Size message_size,
                                         const char *message);
```

txn参数包含关于该事务的元信息，如被提交的时间戳和XID。不过要注意，当消息是非事务性的并且记录该消息的事务中还没有被分配XID时，这个参数可以为NULL。lsn是该消息的WAL位置。transactional说明该消息是否为事务性的。prefix是一个任意的空终结的前缀，它当前插件被用来标识感兴趣的消息。最后的message参数保存着大小为message\_size的消息。

应该格外小心确保输出插件用于标识感兴趣消息的前缀是唯一的。建议使用扩展或者输出插件本身的名称。

#### 45.6.5. 用于产生输出的函数

在begin\_cb、commit\_cb或者change\_cb回调中，为了实际产生输出，输出插件可以把数据写入到ctx->out中的StringInfo输出缓冲区中。在写出到输出缓冲区之前，必须先调用OutputPluginPrepareWrite(ctx, last\_write)，在完成写入到缓冲区后，必须调用OutputPluginWrite(ctx, last\_write)来执行写出。last\_write指出一次特定的写出是否为该回调的最后一次写出。

下面的例子展示了如何把数据输出给一个输出插件的消费者：

```
OutputPluginPrepareWrite(ctx, true);
appendStringInfo(ctx->out, "BEGIN %u", txn->xid);
OutputPluginWrite(ctx, true);
```

### 45.7. 逻辑解码输出写入器

可以为逻辑解码增加更多输出方法。详情可见src/backend/replication/logical/logicalfuncs.c。本质上，需要提供三个函数：一个读取WAL，一个准备写输出，另一个写输出（见Section 45.6.5）。

## 45.8. 逻辑解码的同步复制支持

### 45.8.1. 总览

逻辑解码可以被用来构建同步复制方案，该方案具有和流复制的同步复制相同的用户接口。要这样做，流复制接口（见Section 45.3）必须被用来流式传出数据。正如流复制客

户端所作的一样，逻辑解码的客户端必须发出 后备机状态更新 (F)（见Section 49.4 消息）。

### Note

一个通过逻辑解码接收更改的同步复制机将工作在一个单一数据库的范围内。因为与之相反，`synchronous_standby_names` 当前是服务器范围的，这意味着如果有多于一个数据库被活跃地使用，这种技术将 无法正常工作。

## 45.8.2. 注意事项

在同步复制设置中，如果事务已经独占锁定了[user]目录表， 可能会发生死锁。有关用户目录表的信息，请参见Section 45.6.2 这是因为事务的逻辑解码会锁定目录表以访问它们。 为避免此问题，用户必须避免对[user]目录表进行独占锁定。以下是可能发生这种情况的方式：

- 在事务中对pg\_class进行显式的LOCK操作。
- 在事务中对pg\_class执行CLUSTER操作。
- 在事务中对[user]目录表执行TRUNCATE操作。

请注意，上述可能导致死锁的命令不仅适用于上面明确指定的系统目录表，也适用于任何其他[user]目录表。

---

## Chapter 46. 复制进度追踪

复制源是为了更容易地在逻辑解码 上实现逻辑复制解决方案而设计。它们提供了对两种常见问题的解决方案：

- 如何安全地跟踪复制进度？
- 如何基于一行的来源更改复制行为？例如，阻止双向复制 设置中的循环

复制源只有两个属性，名称和 OID。名称应该可以被用来在系统间引用该源，它是一种自由形式的文本。为了避免复制源之间的冲突，可以在复制源的名称前加上复制解决方案的名称。在空间效率很重要的情况下，OID 被用来避免不得不存储长版本。OID 不能在系统间被共享。

可以使用函数 `pg_replication_origin_create()` 创建复制源，使用函数 `pg_replication_origin_drop()` 删除复制源，并且在系统目录 `pg_replication_origin` 中查看复制源。

构建一套复制解决方案的一个重要部分是以一种安全的方式跟踪重放进度。当应用 过程或者整个集群死掉时，需要能够找出数据被成功地复制到了什么地方。对此的 简单解决方案（例如为每一个被重放的事务更新一个表行）有运行时负荷和数据库 膨胀的问题。

通过使用复制源，一个会话可以被标记为从一个远程节点重放（使用 `pg_replication_origin_session_setup()` 函数）。此外， 可以使用 `pg_replication_origin_xact_setup()` 以每一个事务为基础配置每一个源事务的 LSN和提交时间戳。如果完成这样的配置， 复制过程将保持在一种对崩溃安全的方式中。所有复制源的重放进度可以在 `pg_replication_origin_status` 视图中看到。一个源的进度（例如在继续复制时）可以使用 `pg_replication_origin_progress()`（用于任何源）或者 `pg_replication_origin_session_progress()`（用于在当前会话中配置的源）获得。

在比从一个系统复制到另一个系统更复杂的复制拓扑中，另一个问题是很难避免再次 复制已经被重放的行。这可能导致复制中的循环和低效。复制源提供了一种可选的机制 来识别和阻止这种问题。在使用前一段提到的函数配置时，每一个被传递 给输出插件回调（见Section 45.6 的由该会话 生成的改变和事务会被标记上该会话的复制源。这使得可以在输出插件中以不同的方式 对待它们，例如忽略除本地生成的行之外的所有行。此外，`filter_by_origin_cb`回调可以被用来基于来源过滤逻辑 解码改变流。虽然灵活性较低，通过这种回调进行过滤比用输出插件过滤效率更高。

---

## Part VI. 参考

这份参考中的条目意欲提供关于相应主题的权威、完整和正式的总结。关于使用LightDB的更多信息（以叙述、教程或例子的形式）可以在本书的其他部分找到。见每个参考页面上列出的交叉引用。

这些参考条目也在传统“man”页面中可用。

---



---

# Table of Contents

I. SQL 命令	1298
ABORT	1302
ALTER AGGREGATE	1303
ALTER COLLATION	1305
ALTER CONVERSION	1307
ALTER DATABASE	1308
ALTER DEFAULT PRIVILEGES	1310
ALTER DOMAIN	1313
ALTER EVENT TRIGGER	1316
ALTER EXTENSION	1317
ALTER FOREIGN DATA WRAPPER	1320
ALTER FOREIGN TABLE	1322
ALTER FUNCTION	1327
ALTER GROUP	1331
ALTER INDEX	1333
ALTER LANGUAGE	1336
ALTER LARGE OBJECT	1337
ALTER MATERIALIZED VIEW	1338
ALTER OPERATOR	1340
ALTER OPERATOR CLASS	1342
ALTER OPERATOR FAMILY	1343
ALTER POLICY	1347
ALTER PROCEDURE	1349
ALTER PUBLICATION	1352
ALTER ROLE	1354
ALTER ROUTINE	1358
ALTER RULE	1359
ALTER SCHEMA	1360
ALTER SEQUENCE	1361
ALTER SERVER	1364
ALTER SESSION	1366
ALTER STATISTICS	1367
ALTER SUBSCRIPTION	1368
ALTER SYSTEM	1370
ALTER TABLE	1372
ALTER TABLESPACE	1391
ALTER TEXT SEARCH CONFIGURATION	1393
ALTER TEXT SEARCH DICTIONARY	1395
ALTER TEXT SEARCH PARSER	1397
ALTER TEXT SEARCH TEMPLATE	1398
ALTER TRIGGER	1399
ALTER TYPE	1401
ALTER USER	1405
ALTER USER MAPPING	1406
ALTER VIEW	1407
ANALYZE	1409
BEGIN	1412
CALL	1414
CHECKPOINT	1415
CLOSE	1416
CLUSTER	1417
COMMENT	1419
COMMIT	1423
COMMIT PREPARED	1424
COPY	1425

CREATE ACCESS METHOD .....	1434
CREATE AGGREGATE .....	1435
CREATE CAST .....	1442
CREATE COLLATION .....	1446
CREATE CONVERSION .....	1448
CREATE DATABASE .....	1450
CREATE DOMAIN .....	1456
CREATE EVENT TRIGGER .....	1459
CREATE EXTENSION .....	1461
CREATE FOREIGN DATA WRAPPER .....	1463
CREATE FOREIGN TABLE .....	1465
CREATE FUNCTION .....	1469
CREATE GROUP .....	1476
CREATE INDEX .....	1477
CREATE LANGUAGE .....	1485
CREATE MATERIALIZED VIEW .....	1487
CREATE OPERATOR .....	1489
CREATE OPERATOR CLASS .....	1492
CREATE OPERATOR FAMILY .....	1495
CREATE POLICY .....	1496
CREATE PROCEDURE .....	1501
CREATE PUBLICATION .....	1504
CREATE ROLE .....	1506
CREATE RULE .....	1510
CREATE SCHEMA .....	1513
CREATE SEQUENCE .....	1515
CREATE SERVER .....	1519
CREATE STATISTICS .....	1521
CREATE SUBSCRIPTION .....	1523
CREATE TABLE .....	1526
CREATE TABLE AS .....	1555
CREATE TABLESPACE .....	1558
CREATE TEXT SEARCH CONFIGURATION .....	1560
CREATE TEXT SEARCH DICTIONARY .....	1561
CREATE TEXT SEARCH PARSER .....	1563
CREATE TEXT SEARCH TEMPLATE .....	1565
CREATE TRANSFORM .....	1566
CREATE TRIGGER .....	1568
CREATE TYPE .....	1574
CREATE USER .....	1585
CREATE USER MAPPING .....	1586
CREATE VIEW .....	1587
DEALLOCATE .....	1592
DECLARE .....	1593
DELETE .....	1596
DISCARD .....	1599
DO .....	1600
DROP ACCESS METHOD .....	1602
DROP AGGREGATE .....	1603
DROP CAST .....	1605
DROP COLLATION .....	1606
DROP CONVERSION .....	1607
DROP DATABASE .....	1608
DROP DOMAIN .....	1609
DROP EVENT TRIGGER .....	1610
DROP EXTENSION .....	1611
DROP FOREIGN DATA WRAPPER .....	1612
DROP FOREIGN TABLE .....	1613

DROP FUNCTION .....	1614
DROP GROUP .....	1616
DROP INDEX .....	1617
DROP LANGUAGE .....	1619
DROP MATERIALIZED VIEW .....	1620
DROP OPERATOR .....	1621
DROP OPERATOR CLASS .....	1623
DROP OPERATOR FAMILY .....	1625
DROP OWNED .....	1627
DROP POLICY .....	1628
DROP PROCEDURE .....	1629
DROP PUBLICATION .....	1631
DROP ROLE .....	1632
DROP ROUTINE .....	1633
DROP RULE .....	1634
DROP SCHEMA .....	1635
DROP SEQUENCE .....	1636
DROP SERVER .....	1637
DROP STATISTICS .....	1638
DROP SUBSCRIPTION .....	1639
DROP TABLE .....	1640
DROP TABLESPACE .....	1642
DROP TEXT SEARCH CONFIGURATION .....	1643
DROP TEXT SEARCH DICTIONARY .....	1644
DROP TEXT SEARCH PARSER .....	1645
DROP TEXT SEARCH TEMPLATE .....	1646
DROP TRANSFORM .....	1647
DROP TRIGGER .....	1648
DROP TYPE .....	1649
DROP USER .....	1650
DROP USER MAPPING .....	1651
DROP VIEW .....	1652
END .....	1653
EXECUTE .....	1654
EXPLAIN .....	1655
FETCH .....	1660
GRANT .....	1664
IMPORT FOREIGN SCHEMA .....	1669
INSERT .....	1671
LISTEN .....	1680
LOAD .....	1682
LOCK .....	1683
MOVE .....	1686
MERGE .....	1688
NOTIFY .....	1692
PREPARE .....	1694
PREPARE TRANSACTION .....	1697
REASSIGN OWNED .....	1699
REFRESH MATERIALIZED VIEW .....	1700
REINDEX .....	1702
RELEASE SAVEPOINT .....	1706
REPLACE INTO .....	1707
RESET .....	1710
REVOKE .....	1711
ROLLBACK .....	1715
ROLLBACK PREPARED .....	1716
ROLLBACK TO SAVEPOINT .....	1717
SAVEPOINT .....	1719

SECURITY LABEL .....	1721
SELECT .....	1724
SELECT INTO .....	1752
SET .....	1754
SET CONSTRAINTS .....	1757
SET ROLE .....	1758
SET SESSION AUTHORIZATION .....	1760
SET TRANSACTION .....	1762
SHOW .....	1765
START TRANSACTION .....	1769
TRUNCATE .....	1770
UNLISTEN .....	1772
UPDATE .....	1774
VACUUM .....	1779
VALUES .....	1783
II. LightDB 客户端应用 .....	1786
clusterdb .....	1787
createdb .....	1790
lt_createdb .....	1794
createuser .....	1797
dropdb .....	1801
dropuser .....	1804
ecpg .....	1807
executor .....	1809
lightdb_service.py .....	1810
lt_distributed_probackup.py .....	1815
lt_distributed_dump.py .....	1821
lt_distributed_restore.py .....	1832
ltdts_recvlogical .....	1839
ltdts_ora .....	1843
ltdts_mysql .....	1846
ltldr .....	1850
ltuldr .....	1853
lt_basebackup .....	1856
lt_config .....	1863
lt_dump .....	1866
lt_dumpall .....	1879
lt_isready .....	1886
lt_probackup .....	1888
lt_receivewal .....	1938
lt_recvlogical .....	1942
lt_restore .....	1946
lt_verifybackup .....	1955
ltbench .....	1958
ltcenter .....	1975
ltsql .....	1977
reindexdb .....	2017
vacuumdb .....	2021
III. LightDB 服务器应用 .....	2027
initdb .....	2028
lt_initdb .....	2033
lt_patch .....	2038
lt_check .....	2040
lt_archivecleanup .....	2042
lt_checksums .....	2044
lt_controldata .....	2046
lt_ctl .....	2047
lt_resetwal .....	2052

---

lt_rewind.....	2055
lt_test_fsync.....	2059
lt_test_timing.....	2060
lt_upgrade.....	2063
lt_upgrade_check.....	2071
lt_waldump.....	2073
lightdb.....	2075
postmaster.....	2082
IV. LightDB 工具应用程序.....	2083
check_list.....	2084
iftop.....	2085
iotop.....	2086
keepalived.....	2087
linux-ftools.....	2088
ltloader.....	2089
lvs.....	2090
ntp.....	2091
ora2pg.....	2092
PostgREST.....	2093
tshark.....	2094
vmtouch.....	2095

---

# SQL 命令

这部分包含LightDB支持的SQL命令的参考信息。每条命令的标准符合和兼容的信息可以在相关的参考页中找到。

## Table of Contents

ABORT .....	1302
ALTER AGGREGATE .....	1303
ALTER COLLATION .....	1305
ALTER CONVERSION .....	1307
ALTER DATABASE .....	1308
ALTER DEFAULT PRIVILEGES .....	1310
ALTER DOMAIN .....	1313
ALTER EVENT TRIGGER .....	1316
ALTER EXTENSION .....	1317
ALTER FOREIGN DATA WRAPPER .....	1320
ALTER FOREIGN TABLE .....	1322
ALTER FUNCTION .....	1327
ALTER GROUP .....	1331
ALTER INDEX .....	1333
ALTER LANGUAGE .....	1336
ALTER LARGE OBJECT .....	1337
ALTER MATERIALIZED VIEW .....	1338
ALTER OPERATOR .....	1340
ALTER OPERATOR CLASS .....	1342
ALTER OPERATOR FAMILY .....	1343
ALTER POLICY .....	1347
ALTER PROCEDURE .....	1349
ALTER PUBLICATION .....	1352
ALTER ROLE .....	1354
ALTER ROUTINE .....	1358
ALTER RULE .....	1359
ALTER SCHEMA .....	1360
ALTER SEQUENCE .....	1361
ALTER SERVER .....	1364
ALTER SESSION .....	1366
ALTER STATISTICS .....	1367
ALTER SUBSCRIPTION .....	1368
ALTER SYSTEM .....	1370
ALTER TABLE .....	1372
ALTER TABLESPACE .....	1391
ALTER TEXT SEARCH CONFIGURATION .....	1393
ALTER TEXT SEARCH DICTIONARY .....	1395
ALTER TEXT SEARCH PARSER .....	1397
ALTER TEXT SEARCH TEMPLATE .....	1398
ALTER TRIGGER .....	1399
ALTER TYPE .....	1401
ALTER USER .....	1405
ALTER USER MAPPING .....	1406
ALTER VIEW .....	1407
ANALYZE .....	1409
BEGIN .....	1412
CALL .....	1414
CHECKPOINT .....	1415
CLOSE .....	1416

---

CLUSTER .....	1417
COMMENT .....	1419
COMMIT .....	1423
COMMIT PREPARED .....	1424
COPY .....	1425
CREATE ACCESS METHOD .....	1434
CREATE AGGREGATE .....	1435
CREATE CAST .....	1442
CREATE COLLATION .....	1446
CREATE CONVERSION .....	1448
CREATE DATABASE .....	1450
CREATE DOMAIN .....	1456
CREATE EVENT TRIGGER .....	1459
CREATE EXTENSION .....	1461
CREATE FOREIGN DATA WRAPPER .....	1463
CREATE FOREIGN TABLE .....	1465
CREATE FUNCTION .....	1469
CREATE GROUP .....	1476
CREATE INDEX .....	1477
CREATE LANGUAGE .....	1485
CREATE MATERIALIZED VIEW .....	1487
CREATE OPERATOR .....	1489
CREATE OPERATOR CLASS .....	1492
CREATE OPERATOR FAMILY .....	1495
CREATE POLICY .....	1496
CREATE PROCEDURE .....	1501
CREATE PUBLICATION .....	1504
CREATE ROLE .....	1506
CREATE RULE .....	1510
CREATE SCHEMA .....	1513
CREATE SEQUENCE .....	1515
CREATE SERVER .....	1519
CREATE STATISTICS .....	1521
CREATE SUBSCRIPTION .....	1523
CREATE TABLE .....	1526
CREATE TABLE AS .....	1555
CREATE TABLESPACE .....	1558
CREATE TEXT SEARCH CONFIGURATION .....	1560
CREATE TEXT SEARCH DICTIONARY .....	1561
CREATE TEXT SEARCH PARSER .....	1563
CREATE TEXT SEARCH TEMPLATE .....	1565
CREATE TRANSFORM .....	1566
CREATE TRIGGER .....	1568
CREATE TYPE .....	1574
CREATE USER .....	1585
CREATE USER MAPPING .....	1586
CREATE VIEW .....	1587
DEALLOCATE .....	1592
DECLARE .....	1593
DELETE .....	1596
DISCARD .....	1599
DO .....	1600
DROP ACCESS METHOD .....	1602
DROP AGGREGATE .....	1603
DROP CAST .....	1605
DROP COLLATION .....	1606
DROP CONVERSION .....	1607
DROP DATABASE .....	1608

---

---

DROP DOMAIN	1609
DROP EVENT TRIGGER	1610
DROP EXTENSION	1611
DROP FOREIGN DATA WRAPPER	1612
DROP FOREIGN TABLE	1613
DROP FUNCTION	1614
DROP GROUP	1616
DROP INDEX	1617
DROP LANGUAGE	1619
DROP MATERIALIZED VIEW	1620
DROP OPERATOR	1621
DROP OPERATOR CLASS	1623
DROP OPERATOR FAMILY	1625
DROP OWNED	1627
DROP POLICY	1628
DROP PROCEDURE	1629
DROP PUBLICATION	1631
DROP ROLE	1632
DROP ROUTINE	1633
DROP RULE	1634
DROP SCHEMA	1635
DROP SEQUENCE	1636
DROP SERVER	1637
DROP STATISTICS	1638
DROP SUBSCRIPTION	1639
DROP TABLE	1640
DROP TABLESPACE	1642
DROP TEXT SEARCH CONFIGURATION	1643
DROP TEXT SEARCH DICTIONARY	1644
DROP TEXT SEARCH PARSER	1645
DROP TEXT SEARCH TEMPLATE	1646
DROP TRANSFORM	1647
DROP TRIGGER	1648
DROP TYPE	1649
DROP USER	1650
DROP USER MAPPING	1651
DROP VIEW	1652
END	1653
EXECUTE	1654
EXPLAIN	1655
FETCH	1660
GRANT	1664
IMPORT FOREIGN SCHEMA	1669
INSERT	1671
LISTEN	1680
LOAD	1682
LOCK	1683
MOVE	1686
MERGE	1688
NOTIFY	1692
PREPARE	1694
PREPARE TRANSACTION	1697
REASSIGN OWNED	1699
REFRESH MATERIALIZED VIEW	1700
REINDEX	1702
RELEASE SAVEPOINT	1706
REPLACE INTO	1707
RESET	1710

---



---

REVOKE .....	1711
ROLLBACK .....	1715
ROLLBACK PREPARED .....	1716
ROLLBACK TO SAVEPOINT .....	1717
SAVEPOINT .....	1719
SECURITY LABEL .....	1721
SELECT .....	1724
SELECT INTO .....	1752
SET .....	1754
SET CONSTRAINTS .....	1757
SET ROLE .....	1758
SET SESSION AUTHORIZATION .....	1760
SET TRANSACTION .....	1762
SHOW .....	1765
START TRANSACTION .....	1769
TRUNCATE .....	1770
UNLISTEN .....	1772
UPDATE .....	1774
VACUUM .....	1779
VALUES .....	1783

---

# ABORT

ABORT — 中止当前事务

## Synopsis

```
ABORT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

## 描述

ABORT回滚当前事务并且导致由该事务所作的所有更新被丢弃。这个命令的行为与标准SQL命令 ROLLBACK的行为一样，并且只是为了历史原因存在。

## 参数

WORK  
TRANSACTION

可选关键词。它们没有效果。

AND CHAIN

如果规定了AND CHAIN，新事务立即启动，具有与刚刚完成的事务相同的事务特征(参见 SET TRANSACTION)。否则，不会启动新事务。

## 注解

使用COMMIT 成功地终止一个事务。

在一个事务块之外发出ABORT会发出一个警告消息并且不会产生效果。

## 例子

中止所有更改:

```
ABORT;
```

## 兼容性

这个命令是一个因为历史原因而存在的LightDB扩展。ROLLBACK是等效的标准 SQL 命令。

## 参见

BEGIN, COMMIT, ROLLBACK

---

# ALTER AGGREGATE

ALTER AGGREGATE — 更改一个聚集函数的定义

## Synopsis

```
ALTER AGGREGATE name ( aggregate_signature ) RENAME TO new_name
ALTER AGGREGATE name ( aggregate_signature )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER AGGREGATE name ( aggregate_signature ) SET SCHEMA new_schema
```

其中 `aggregate_signature` 是：

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
] argtype [ , ... ]
```

## 描述

ALTER AGGREGATE更改一个聚集函数的定义。

要使用ALTER AGGREGATE，你必须拥有该聚集函数。要更改一个聚集函数的模式，你还必须具有新模式上的 CREATE特权。要修改拥有者，你还必须是新拥有角色的一个直接或者间接成员，并且那个角色必须在聚集函数的模式上拥有 CREATE特权（这些限制强制要求拥有者不能通过丢弃并重建该聚集函数来做任何你不能做的事情。不过，一个超级用户可以更改任何聚集函数的所有权）。

## 参数

`name`

一个现有聚集函数的名称（可以是模式限定的）。

`argmode`

一个参数的模式：IN或VARIADIC。如果省略，默认为IN。

`argname`

一个参数的名称。注意ALTER AGGREGATE 并不真正关心参数名称，因为决定聚集函数的身份时只需要参数的数据类型。

`argtype`

聚集函数要在其上操作的输入数据类型。要引用一个零参数聚集函数，在参数 说明列表的位置写上\*。要引用一个有序集聚集函数，在直接参数 说明和聚集参数说明之间写上ORDER BY。

`new_name`

聚集函数的新名称。

`new_owner`

聚集函数的新拥有者。

new\_schema

聚集函数的新模式。

## 注解

引用有序集聚集的推荐语法是在直接参数说明和聚集参数说明之间写上 `ORDER BY`，这和 `CREATE AGGREGATE` 中的风格相同。不过，省略 `ORDER BY` 并且只把直接和聚集参数说明放到一个单一列表中也是可以的。在这种简写形式中，如果在直接和聚集参数列表中都使用了 `VARIADIC "any"`，只用写一次 `VARIADIC "any"`。

## 示例

要把用于类型 `integer` 的聚集函数 `myavg` 重命名为 `my_average`：

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

要把用于类型 `integer` 的聚集函数 `myavg` 的拥有者改为 `joe`：

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

把带有 `float8` 类型直接参数和 `integer` 类型聚集参数的有序集聚集 `mypercentile` 移动到模式 `myschema` 中：

```
ALTER AGGREGATE mypercentile(float8 ORDER BY integer) SET SCHEMA myschema;
```

这也能行：

```
ALTER AGGREGATE mypercentile(float8, integer) SET SCHEMA myschema;
```

## 兼容性

在 SQL 标准中没有 `ALTER AGGREGATE` 语句。

## 另见

`CREATE AGGREGATE`, `DROP AGGREGATE`

---

# ALTER COLLATION

ALTER COLLATION — 更改一个排序规则的定义

## Synopsis

```
ALTER COLLATION name REFRESH VERSION
```

```
ALTER COLLATION name RENAME TO new_name
```

```
ALTER COLLATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

```
ALTER COLLATION name SET SCHEMA new_schema
```

## 描述

ALTER COLLATION更改一个排序规则的定义。

你必须拥有要对其使用ALTER COLLATION的排序规则。要更改拥有者，你必须是新的拥有角色的直接或者间接成员，并且该角色必须在排序规则的模式上具有CREATE特权（这些限制强制要求拥有者不能通过丢弃并重建该排序规则来做任何你不能做的事情。不过，一个超级用户可以更改任何排序规则的所有权）。

## 参数

name

一个现有排序规则的名称（可以是模式限定的）。

new\_name

排序规则的新名称。

new\_owner

排序规则的新拥有者。

new\_schema

排序规则的新模式。

REFRESH VERSION

更新排序规则的版本。参阅下面的Notes。

## 注意

使用ICU库提供的排序规则时，创建排序规则对象时，系统目录中会记录排序规则的特定ICU版本。使用排序规则时，将根据记录的版本检查当前版本，并在发生不匹配时发出警告，例如：

```
WARNING: collation "xx-x-icu" has version mismatch
```

```
DETAIL: The collation in the database was created using version 1.2.3.4, but  
the operating system provides version 2.3.4.5.
```

```
HINT: Rebuild all objects affected by this collation and run ALTER COLLATION  
pg_catalog."xx-x-icu" REFRESH VERSION, or build LightDB with the right library  
version.
```

排序规则定义的更改会导致索引损坏和其他问题，因为数据库系统依赖于具有特定排序顺序的存储对象。通常，应该避免这种情况，但它可以在合法的情况下发生，例如使用 `lt_upgrade` 升级到与更新版本的ICU链接的服务器二进制文件。发生这种情况时，应该重建所有依赖于该排序规则的对象，例如，使用 `REINDEX`。完成后，使用命令 `ALTER COLLATION ... REFRESH VERSION` 可以刷新排序规则版本。这将更新系统目录以记录当前的排序规则版本，并会使警告消失。请注意，这实际上并不检查是否所有受影响的对象都已正确重建。

当使用 `libc` 提供的排序规则时，版本信息记录在使用GNU C库（大多数Linux系统）、FreeBSD和Windows的系统上。

当前，没有针对数据库默认排序规则的版本跟踪。

以下查询可用于识别当前数据库中需要刷新的所有排序规则以及依赖它们的对象：

```
SELECT pg_describe_object(refclassid, refobjid, refobjsubid) AS "Collation",
       pg_describe_object(classid, objid, objsubid) AS "Object"
FROM pg_depend d JOIN pg_collation c
     ON refclassid = 'pg_collation'::regclass AND refobjid = c.oid
WHERE c.collversion <> pg_collation_actual_version(c.oid)
ORDER BY 1, 2;
```

## 例子

要把排序规则 `de_DE` 重命名为 `german`：

```
ALTER COLLATION "de_DE" RENAME TO german;
```

要把排序规则 `en_US` 的拥有者改成 `joe`：

```
ALTER COLLATION "en_US" OWNER TO joe;
```

## 兼容性

在 SQL 标准中没有 `ALTER COLLATION` 语句。

## 参见

`CREATE COLLATION`, `DROP COLLATION`

---

# ALTER CONVERSION

ALTER CONVERSION — 改变一个转换的定义

## Synopsis

```
ALTER CONVERSION name RENAME TO new_name
ALTER CONVERSION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER CONVERSION name SET SCHEMA new_schema
```

## 描述

ALTER CONVERSION改变一个转换的定义。

你必须拥有要对其使用ALTER CONVERSION的转换。要更改所有者，你必须是新的拥有角色的直接或者间接成员，并且该角色必须在转换的模式上具有CREATE特权（这些限制强制要求所有者不能通过丢弃并重建该转换来做任何你不能做的事情。不过，一个超级用户可以更改任何转换的所有权）。

## 参数

name

一个现有转换的名称（可以是模式限定的）。

new\_name

转换的新名称。

new\_owner

转换的新所有者。

new\_schema

转换的新模式。

## 例子

要把转换iso\_8859\_1\_to\_utf8重命名为latin1\_to\_unicode:

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

要把转换iso\_8859\_1\_to\_utf8的拥有者改成joe:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

## 兼容性

在 SQL 标准中没有ALTER CONVERSION语句。

## 参见

CREATE CONVERSION, DROP CONVERSION

---

# ALTER DATABASE

ALTER DATABASE — 更改一个数据库

## Synopsis

```
ALTER DATABASE name [ [ WITH ] option [ ... ] ]
```

这里 option 可以是：

```
ALLOW_CONNECTIONS allowconn  
CONNECTION LIMIT conlimit  
IS_TEMPLATE istemplate
```

```
ALTER DATABASE name RENAME TO new_name
```

```
ALTER DATABASE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

```
ALTER DATABASE name SET TABLESPACE new_tablespace
```

```
ALTER DATABASE name SET configuration_parameter { TO | = } { value | DEFAULT }  
ALTER DATABASE name SET configuration_parameter FROM CURRENT  
ALTER DATABASE name RESET configuration_parameter  
ALTER DATABASE name RESET ALL
```

## 描述

ALTER DATABASE更改一个数据库的属性。

第一种形式更改某些针对每个数据库的设置（详见下文）。只有数据库所有者或者超级用户可以更改这些设置。

第二种形式更改数据库的名称。只有数据库所有者或者超级用户可以重命名一个数据库，非超级用户所有者还必须拥有CREATEDB特权。当前数据库不能被重命名（如果你需要这样做请连接到一个不同的数据库）。

第三种形式更改数据库的拥有者。要修改拥有者，你必须拥有该数据库并且也是新拥有角色的一个直接或间接成员，并且你必须具有CREATEDB特权（注意超级用户自动拥有所有这些特权）。

第四种形式更改数据库的默认表空间。只有数据库所有者或超级用户能够这样做，你还必须对新表空间具有创建特权。这个命令会在物理上移动位于该数据库旧的默认表空间中的任何表或索引到新的表空间中。新的默认表空间对于这个数据库必须是空的，并且不能有人可以连接到该数据库。在非默认表空间中的表和索引不受影响。

剩下的形式更改用于一个LightDB数据库的运行时配置变量的会话默认值。接下来只要一个新的会话在该数据库中开始，指定的值就会成为该会话的默认值。数据库相关的默认值会覆盖出现在lightdb.conf中或者从lightdb命令行接收到的设置。只有数据库所有者或超级用户可以更改一个数据库的会话默认值。一些变量不能用这种方式设置或者只能由超级用户更改。

## 参数

name

要被修改属性的数据库名称。



allowconn

如果为假，则没有人能连接到这个数据库。

connlimit

与这个数据库可以建立多少个并发连接。-1 表示没有限制。

istemplate

如果为真，则任何具有CREATEDB特权的用户都可以从这个数据库进行克隆。如果为假，则只有超级用户或者这个数据库的拥有者可以克隆它。

new\_name

数据库的新名称。

new\_owner

数据库的新所有者。

new\_tablespace

数据库的新默认表空间。

这种形式的命令不能在事务块内执行。

configuration\_parameter

value

将这个数据库的指定配置参数的会话默认值设置为给定值。如果value是DEFAULT，或者等效地使用了RESET，数据库相关的设置会被移除，因此系统范围的默认设置将会在新会话中继承。使用RESET ALL可清除所有数据库相关的设置。SET FROM CURRENT会保存该会话的当前参数值作为数据库相关的值。

更多关于允许的参数名称和值的信息可参考SET和Chapter 17

## 注解

也可以把一个会话的默认值绑定到一个特定角色而不是一个数据库，见ALTER ROLE。如果有冲突，角色相关的设置会覆盖数据库相关的值。

## 例子

要在数据库test中默认禁用索引扫描：

```
ALTER DATABASE test SET enable_indexscan TO off;
```

## 兼容性

ALTER DATABASE语句是一个LightDB扩展。

## 参见

CREATE DATABASE, DROP DATABASE, SET, CREATE TABLESPACE

---

# ALTER DEFAULT PRIVILEGES

ALTER DEFAULT PRIVILEGES — 定义默认访问特权

## Synopsis

```
ALTER DEFAULT PRIVILEGES
  [ FOR { ROLE | USER } target_role [, ...] ]
  [ IN SCHEMA schema_name [, ...] ]
  abbreviated_grant_or_revoke
```

其中abbreviated\_grant\_or\_revoke是下列之一:

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
  [, ...] | ALL [ PRIVILEGES ] }
  ON TABLES
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { USAGE | SELECT | UPDATE }
  [, ...] | ALL [ PRIVILEGES ] }
  ON SEQUENCES
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON { FUNCTIONS | ROUTINES }
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON TYPES
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | CREATE | ALL [ PRIVILEGES ] }
  ON SCHEMAS
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
REVOKE [ GRANT OPTION FOR ]
  { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
  [, ...] | ALL [ PRIVILEGES ] }
  ON TABLES
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
  { { USAGE | SELECT | UPDATE }
  [, ...] | ALL [ PRIVILEGES ] }
  ON SEQUENCES
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
  { EXECUTE | ALL [ PRIVILEGES ] }
  ON { FUNCTIONS | ROUTINES }
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
  { USAGE | ALL [ PRIVILEGES ] }
ON TYPES
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
  { USAGE | CREATE | ALL [ PRIVILEGES ] }
ON SCHEMAS
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

## 描述

ALTER DEFAULT PRIVILEGES允许你设置将被应用于未来要创建的对象的特权（它不会影响分配给已经存在的对象的特权）。当前，只能修改用于模式、表（包括视图和外部表）、序列、函数和类型（包括域）的特权。其中，可设置权限的函数包括聚集函数和过程函数。当这个命令应用于函数时，单词FUNCTIONS和ROUTINES是等效的。（推荐使用ROUTINES，因为它是用来囊括函数和过程的一个标准术语。在较早的LightDB发行版中，只允许单词FUNCTIONS。无法为函数或过程单独设置默认特权。）

你只能改变你自己或者你属于其中的角色所创建的对象默认特权。这些特权可以对全局范围设置（即对当前数据库中创建的所有对象），或者只对指定模式中创建的对象设置。

如Section 5. 中所述，用于任何对象类型的默认特权通常会把所有可授予的权限授予给对象所有者，并且也可能授予一些特权给PUBLIC。不过，这种行为可以通过使用ALTER DEFAULT PRIVILEGES修改全局默认特权来改变。

每个模式规定的默认权限将添加到特定对象类型的全局默认特权的内容中。这意味着你不能撤销每个模式的特权，如果它们已经被全局授予（默认情况下，或者根据以前未规定模式的ALTER DEFAULT PRIVILEGES命令）。每个模式的REVOKE仅用于反转以前每个模式GRANT的影响。

## 参数

target\_role

一个现有角色的名称，当前角色是它的一个成员。如果FOR ROLE被忽略，将假定为当前角色。

schema\_name

一个现有模式的名称。如果被指定，以后在那个模式中创建的对象默认特权会被修改。如果IN SCHEMA被忽略，全局默认特权会被修改。当设置特权给模式时不能使用IN SCHEMA，因为模式不能嵌套。

role\_name

要为其授予或者收回特权的一个现有角色的名称。这个参数以及所有abbreviated\_grant\_or\_revoke中的其他参数会按照GRANT或者REVOKE中描述的方式运作，不过这里是为一整类的对象而不是特别指定的对象设置权限。

## 注解

使用ltsql的\ddp命令可以获得关于默认特权的现有分配信息。特权显示的含义和Section 5. 中的\dp描述的相同。

如果你希望删除一个默认特权被修改的角色，有必要撤销其默认特权上的改变或者使用DROP OWNED BY来为该角色去除默认特权项。

## 例子

为你后续在模式myschema中创建的所有表（和视图）授予 `SELECT` 特权，并且也允许角色webuser向它们之中 `INSERT` 数据：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;  
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

撤销上面的动作，因此后续创建的表不会有任何不寻常的权限：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;  
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM  
webuser;
```

为后续由角色admin创建的所有函数移除通常在函数上会授予的公共 `EXECUTE` 权限：

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

但是注意你不能使用限制为单个模式的命令来实现该效果。 此命令不起作用，除非它撤销匹配的`GRANT`：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public REVOKE EXECUTE ON FUNCTIONS FROM  
PUBLIC;
```

这是因为每个模式的默认特权只能向全局设置添加特权，而不能移除它授予的特权。

## 兼容性

在 SQL 标准中没有`ALTER DEFAULT PRIVILEGES`语句。

## 参见

`GRANT`, `REVOKE`

---

# ALTER DOMAIN

ALTER DOMAIN — 更改一个域的定义

## Synopsis

```
ALTER DOMAIN name
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
    { SET | DROP } NOT NULL
ALTER DOMAIN name
    ADD domain_constraint [ NOT VALID ]
ALTER DOMAIN name
    DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER DOMAIN name
    VALIDATE CONSTRAINT constraint_name
ALTER DOMAIN name
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER DOMAIN name
    RENAME TO new_name
ALTER DOMAIN name
    SET SCHEMA new_schema
```

## 描述

ALTER DOMAIN更改一个现有域的定义。有几种形式：

SET/DROP DEFAULT

这些形式设置或者移除一个域的默认值。注意默认值只会应用到后续的 INSERT命令，它们不影响使用该域的已经存在于表中的行。

SET/DROP NOT NULL

这些形式更改一个域是被标记为允许 NULL 值还是拒绝 NULL 值。只有当使用该域的列不包含空值时才能SET NOT NULL。

ADD domain\_constraint [ NOT VALID ]

这种形式使用和CREATE DOMAIN相同的语法为一个域增加一个新的约束。当一个新的约束被增加到一个域时，所有使用该域的列都会被根据新加的约束进行检查。可以通过增加使用 NOT VALID选项的新约束来抑制这类检查，而该约束则可以在以后使用 ALTER DOMAIN ... VALIDATE CONSTRAINT 变为可用。新插入和更新的行总是会被根据所有约束进行检查（包括被标记为 NOT VALID的约束）。只有CHECK约束接受 NOT VALID。

DROP CONSTRAINT [ IF EXISTS ]

这种形式删除一个域上的约束。如果指定了IF EXISTS并且约束不存在，不会抛出错误。在这种情况下会转而发出一个提示。

RENAME CONSTRAINT

这种形式更改一个域上的一个约束的名称。

## VALIDATE CONSTRAINT

这种形式验证一个之前作为NOT VALID增加的约束，也就是说 它验证该域类型的表列中所有值满足指定的约束。

## OWNER

这种形式更改域的拥有者为指定用户。

## RENAME

这种形式更改域的名称。

## SET SCHEMA

这种形式更改域的模式。任何与该域关联的约束也会被移动到新的模式中。

要使用ALTER DOMAIN，你必须拥有该域。要更改一个域的模式，你还必须具有新模式上的CREATE特权。要更改拥有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须具有该域的模式上的CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重建域做不到的事情。不过，一个超级用户怎么都能更改任何域的所有权。）。

## 参数

## name

要修改的一个现有域的名称（可能是模式限定的）。

## domain\_constraint

用于该域的新域约束。

## constraint\_name

要删除或重命名的一个现有约束的名称。

## NOT VALID

不为约束的合法性验证现有的存储数据。

## CASCADE

自动删除依赖于该约束的对象，并且接着删除依赖于那些对象的所有对象（见Section 5.14）。

## RESTRICT

如果有任何依赖对象则拒绝删除该约束。这是默认行为。

## new\_name

域的新名称。

## new\_constraint\_name

约束的新名称。

## new\_owner

域的新拥有者的用户名。

## new\_schema

域的新模式。

## 注解

尽管ALTER DOMAIN ADD CONSTRAINT尝试验证现有存储的数据是否满足新约束，但此检查不是万无一失的，因为命令无法“see”新插入或更新但尚未提交的表行。如果存在并发操作可能插入坏数据的危险，则处理方法是使用NOT VALID选项添加约束，提交该命令，等到所有事务在提交完成之前启动，然后发出ALTER DOMAIN VALIDATE CONSTRAINT以搜索违反约束的数据。此方法是可靠的，因为一旦提交约束，所有新事务都保证针对域类型的新值强制执行约束。

当前，如果域或者任何衍生域被数据库中的任意表的一个容器类型列（组合、数组、范围类型的列）使用，ALTER DOMAIN ADD CONSTRAINT、ALTER DOMAIN VALIDATE CONSTRAINT和ALTER DOMAIN SET NOT NULL将会失败。这些命令最终将会被改进成能够对这类嵌套值进行约束验证。

## 示例

要把一个NOT NULL约束加到一个域：

```
ALTER DOMAIN zipcode SET NOT NULL;
```

要从一个域中移除一个NOT NULL约束：

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

要把一个检查约束增加到一个域：

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

要从一个域移除一个检查约束：

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

要重命名一个域上的一个检查约束：

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

要把域移动到一个不同的模式：

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

## 兼容性

ALTER DOMAIN conforms to the SQL standard, except for the 除OWNER、RENAME、SET SCHEMA 以及VALIDATE CONSTRAINT变体之外（它们是 LightDB的扩展），ALTER DOMAIN符合SQL标准。ADD CONSTRAINT变体的NOT VALID子句也是一个 LightDB扩展。

## 另见

CREATE DOMAIN, DROP DOMAIN

---

# ALTER EVENT TRIGGER

ALTER EVENT TRIGGER — 更改一个事件触发器的定义

## Synopsis

```
ALTER EVENT TRIGGER name DISABLE
ALTER EVENT TRIGGER name ENABLE [ REPLICa | ALWAYS ]
ALTER EVENT TRIGGER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER EVENT TRIGGER name RENAME TO new_name
```

## 描述

ALTER EVENT TRIGGER更改一个现有事件触发器 的属性。

必须作为一个超级用户才能修改一个事件触发器。

## 参数

name

要修改的现有触发器的名称。

new\_owner

该事件触发器的新拥有者的用户名。

new\_name

该事件触发器的新名称。

DISABLE/ENABLE [ REPLICa | ALWAYS ] TRIGGER

这些形式配置事件触发器的触发。一个被禁用的触发器对系统来说仍然是可知的，但是当期触发事件发生时却不会执行它。另见 `session_replication_role`。

## 兼容性

在 SQL 标准中没有ALTER EVENT TRIGGER语句。

## 另见

CREATE EVENT TRIGGER, DROP EVENT TRIGGER



---

# ALTER EXTENSION

ALTER EXTENSION — 更改一个扩展的定义

## Synopsis

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object
```

其中 member\_object 是:

```
ACCESS METHOD object_name |
AGGREGATE aggregate_name ( aggregate_signature ) |
CAST (source_type AS target_type) |
COLLATION object_name |
CONVERSION object_name |
DOMAIN object_name |
EVENT TRIGGER object_name |
FOREIGN DATA WRAPPER object_name |
FOREIGN TABLE object_name |
FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
MATERIALIZED VIEW object_name |
OPERATOR operator_name (left_type, right_type) |
OPERATOR CLASS object_name USING index_method |
OPERATOR FAMILY object_name USING index_method |
[ PROCEDURAL ] LANGUAGE object_name |
PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
SCHEMA object_name |
SEQUENCE object_name |
SERVER object_name |
TABLE object_name |
TEXT SEARCH CONFIGURATION object_name |
TEXT SEARCH DICTIONARY object_name |
TEXT SEARCH PARSER object_name |
TEXT SEARCH TEMPLATE object_name |
TRANSFORM FOR type_name LANGUAGE lang_name |
TYPE object_name |
VIEW object_name
```

并且 aggregate\_signature 是:

```
* |
[ argmode ] [ argname ] argtype [, ... ] |
[ [ argmode ] [ argname ] argtype [, ... ] ] ORDER BY [ argmode ] [ argname ]
] argtype [, ... ]
```

## 描述

ALTER EXTENSION更改一个已安装扩展的定义。 有几种形式:

## UPDATE

这种形式把该扩展更新到一个新版本。该扩展必须提供一个适当的更新脚本（或者一系列脚本）来把当前已安装的版本修改成所要求的版本。

## SET SCHEMA

这种形式把该扩展的对象移动到另一个模式中。要使这个命令成功，该扩展必须是可重定位的。

## ADD member\_object

这种形式把一个现有的对象加入到该扩展中。这主要对扩展更新脚本有用。该对象后续将被当作该扩展的一个成员。尤其是该对象只有通过删除扩展才能删除。

## DROP member\_object

这种形式从该扩展移除一个成员对象。这主要对扩展更新脚本有用。只有撤销该对象与其扩展之间的关联后才能删除该对象。

关于这些操作详见Section 36.17

要使用ALTER EXTENSION，你必须拥有该扩展。ADD/DROP形式还要求被增加/删除对象的所有权。

## 参数

## name

一个已安装扩展的名称。

## new\_version

想要得到的该扩展的新版本。这可以写成一个标识符或者一个字符串。如果没有指定，ALTER EXTENSION UPDATE会尝试更新到该扩展的控制文件中的默认版本。

## new\_schema

该扩展的新模式。

## object\_name

## aggregate\_name

## function\_name

## operator\_name

## procedure\_name

## routine\_name

要从该扩展增加或者移除的对象的名称。表、聚集、域、外部表、函数、操作符、操作符类、操作符族、过程、例程、序列、文本搜索对象、类型和视图的名称可以被模式限定。

## source\_type

该转换的源数据类型的名称。

## target\_type

该转换的目标数据类型的名称。

## argmode

一个函数、过程或者聚集参数的模式：IN、OUT、INOUT或者VARIADIC。如果被忽略，默认值是IN。注意，ALTER EXTENSION并不真正关心OUT参数，因为决定该函数的身份时只需要输入参数。因此列出IN、INOUT和VARIADIC参数足矣。

argname

一个函数、过程或者聚集参数的名称。注意，ALTER EXTENSION并不真正关心参数名称，因为决定该函数的身份时只需要参数的数据类型。

argtype

一个函数、过程或者或聚集参数的数据类型。

left\_type

right\_type

该操作符参数的数据类型（可以用模式限定）。对一个前缀操作符的缺失的参数可以写NONE。

PROCEDURAL

这是一个噪声词。

type\_name

该转换的数据类型的名称。

lang\_name

该转换的语言的名称。

## 示例

把hstore扩展更新到版本 2.0:

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

把hstore扩展的模式更改到utils:

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

要向hstore扩展增加一个现有函数:

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anelement, hstore);
```

## 兼容性

ALTER EXTENSION是一个LightDB 扩展。

## 另见

CREATE EXTENSION, DROP EXTENSION

---

# ALTER FOREIGN DATA WRAPPER

ALTER FOREIGN DATA WRAPPER — 更改一个外部数据包装器的定义

## Synopsis

```
ALTER FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER FOREIGN DATA WRAPPER name OWNER TO { new_owner | CURRENT_USER |
SESSION_USER }
ALTER FOREIGN DATA WRAPPER name RENAME TO new_name
```

## 描述

ALTER FOREIGN DATA WRAPPER更改一个 外部数据包装器的定义。该命令的第一种形式用于更改外部数据包装器的 支持函数或者一般选项（至少要求一个子句）。第二种形式更改外部数据包装器的拥有者。

只有超级用户能修改外部数据包装器。此外，只有超级用户能够拥有外部数据包装器。

## 参数

name

一个已有的外部数据包装器的名称。

HANDLER handler\_function

为外部数据包装器指定一个新的处理器函数。

NO HANDLER

用于指定该外部数据包装器不再具有一个处理器函数。

注意使用没有处理器的外部数据包装器的外部表不能被访问。

VALIDATOR validator\_function

为外部数据包装器指定一个新的验证器函数。

注意，新的验证器可能会认为该外部数据包装器或者依赖于它的独立服务器的已有选项、用户映射、外部表无效。LightDB 不会做这种检查。在使用修改过的外部数据包装器之前确认这些选项正确是 用户的责任。不过，在这个ALTER FOREIGN DATA WRAPPER命令中指定的选项将会被使用新的验证器检查。

NO VALIDATOR

用于指定该外部数据包装器不再拥有一个验证器函数。

OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )

为该外部数据包装器更改选项。ADD、SET 以及DROP指定要被执行的动作。如果没有显式地指定操作， 将假定为ADD。选项名称必须唯一，选项名称和值（如果有） 也会使用该外部数据包装器的验证器函数来验证。

`new_owner`

该外部数据包装器的新拥有者的用户名。

`new_name`

该外部数据包装器的新名称。

## 示例

更改一个外部数据包装器 `dbi`，增加选项 `foo` 并删除 `bar`：

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

把外部数据包装器 `dbi` 的验证器改为 `bob.myvalidator`：

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

## 兼容性

`ALTER FOREIGN DATA WRAPPER` 符合 ISO/IEC 9075-9 (SQL/MED)。不过 `HANDLER`、`VALIDATOR`、`OWNER TO` 以及 `RENAME` 子句是扩展。

## 另见

`CREATE FOREIGN DATA WRAPPER`, `DROP FOREIGN DATA WRAPPER`

---

# ALTER FOREIGN TABLE

ALTER FOREIGN TABLE — 更改一个外部表的定义

## Synopsis

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
```

其中 action 是以下之一：

```
    ADD [ COLUMN ] column_name data_type [ COLLATE collation ]
    [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation
]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
MAIN }
    ALTER [ COLUMN ] column_name OPTIONS ( [ ADD | SET | DROP ] option ['value']
[, ... ] )
    ADD table_constraint [ NOT VALID ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    SET WITHOUT OIDS
    INHERIT parent_table
    NO INHERIT parent_table
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

## 描述

ALTER FOREIGN TABLE更改一个现有外部表的定义。 有几种子形式：

### ADD COLUMN

这种形式使用与CREATE FOREIGN TABLE相同的语法把 一个新的列增加到该外部表。和为常规表增加一列不同，这种形式并不影响底层 的存储：这个动作只是简单地声明通过该外部表可以访问某个新的列而已。

## DROP COLUMN [ IF EXISTS ]

这种形式从一个外部表删掉一列。如果在该表外部有任何东西依赖于该列，你将需要写上CASCADE，典型的例子就是视图。如果指定了 IF EXISTS并且该列不存在，将不会抛出错误。在这种情况下会转而发出一个提示。

## SET DATA TYPE

这种形式更改一个外部表的一列的类型。同样，这种形式并不影响底层的存储：这个动作只是简单地更改LightDB相信该列所具有的类型。

## SET/DROP DEFAULT

这些形式设置或者移除一列的默认值。默认值只会应用于后续的 INSERT或UPDATE命令，它们不会导致已经在表中的行被更改。

## SET/DROP NOT NULL

把一列标记为允许或者不允许空值。

## SET STATISTICS

这种形式为后续的ANALYZE操作设置针对每列的统计收集目标。详见ALTER TABLE的类似形式。

SET ( attribute\_option = value [, ... ] )

RESET ( attribute\_option [, ... ] )

这种形式设置或重置针对每个属性的选项。详见ALTER TABLE 的类似形式。

## SET STORAGE

这种形式设置一个列的存储模式。详见 ALTER TABLE中类似的模式。注意存储模式不会产生效果，除非该表的外部数据包装器选择处理它。

## ADD table\_constraint [ NOT VALID ]

这种形式为外部表增加一个新的约束，使用的语法和 CREATE FOREIGN TABLE中相同。当前只支持CHECK约束。

和向常规表增加约束的情况不同，为外部表增加约束时不会做任何事情来验证该约束是否正确。这个动作只是简单地声明了该外部表中所有的行都应该满足的某种新的条件（见 CREATE FOREIGN TABLE中的讨论）。如果该约束被标记为NOT VALID，那么它不被假设为有效，而只是被记录下来以备未来使用。

## VALIDATE CONSTRAINT

这种形式把一个之前被标记为NOT VALID的约束标记为有效。不会做任何动作来验证该约束，但是未来的查询将会假定该约束是保持的。

## DROP CONSTRAINT [ IF EXISTS ]

这种形式删掉在一个外部表上的指定约束。如果指定了 IF EXISTS但约束并不存在，则不会抛出错误。在这种情况下会发出一个提示。

## DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER

这些形式配置属于该外部表的触发器的触发情况。详见 ALTER TABLE的类似形式。

## SET WITHOUT OIDS

向后兼容性语法，表示移除oid系统列。由于不能再为外部表添加oid系统列，此语句不再产生效用。

INHERIT parent\_table

这种形式把目标外部表作为指定的父表的新后代。详见 ALTER TABLE 的类似的形式。

NO INHERIT parent\_table

这种形式把目标外部表从指定的父表的子女列表中移除。

OWNER

这种形式把该外部表的拥有者改成指定的用户。

OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )

更改该外部表或者其中一个列的选项。ADD、SET 以及 DROP 指定要执行的动作。如果没有显式地指定操作，将假定 为 ADD。不允许重复的名称（不过一个表选项和一个列选项可以重名）。选项名称和值也会用外部数据包装器库来验证。

RENAME

RENAME 形式更改外部表的名称或者外部表中一个列的名称。

SET SCHEMA

这种形式把外部表移动到另一个模式中。

所有除了 RENAME 和 SET SCHEMA 的动作都能被整合到一个多修改列表以便能被并行应用。例如，可以在一个 命令中增加几个列并且/或者修改几个列的类型。

如果该命令被写作 ALTER FOREIGN TABLE IF EXISTS ... 并且 该外部表不存在，则不会抛出错误。这种情况下会发出一个提示。

你必须拥有该表以使用 ALTER FOREIGN TABLE。要更改一个 外部表的模式，你必须还拥有新模式上的 CREATE 特权。要 更改拥有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须 具有在该表的模式上的 CREATE 特权（这些限制强制修改拥有者不能做一些通过删除和重建该表做不到的事情。不过，一个超级用户怎么都能 更改任何表的所有权）。要增加一列或者修改一个列的类型，你还必须具有该数据类型上的 USAGE 特权。

## 参数

name

一个要修改的现有外部表的名称（可以被模式限定）。如果在表名前指定了 ONLY，则只有该表被修改。如果没有指定 ONLY， 该表和它所有的后代表（如果有）都会被修改。可选地，在表名后面指定 \* 可以显式地表示将后代表包括在内。

column\_name

一个新的或者现有列的名称。

new\_column\_name

一个现有列的新名称。

new\_name

该表的新名称。

data\_type

新列的数据类型或者一个现有列的新数据类型。



table\_constraint

New table constraint for the foreign table.

constraint\_name

Name of an existing constraint to drop.

CASCADE

自动删除依赖于被删除列或约束的对象（例如，引用该列的视图），并且接着删除依赖于那些对象的所有对象（见Section 5.14）。

RESTRICT

如果有任何依赖对象就拒绝删除该列或约束。这是默认行为。

trigger\_name

要禁用或启用的一个触发器的名称。

ALL

禁用或者启用所有属于该外部表的触发器（如果任何触发器是内部生成的触发器，这都要求超级用户特权。核心系统不会向外部表增加这类触发器，但是附加代码会这样做。）。

USER

禁用或者启用属于该外部表的除了内部生成的触发器之外的所有触发器。

parent\_table

要与这个外部表关联或者解除关联的父表。

new\_owner

该表的新拥有者的用户名。

new\_schema

该表要被移动到其中的模式的名称。

## 注解

关键词COLUMN是噪声词并且可以被忽略。

当使用ADD COLUMN或 DROP COLUMN增加或移除一列、增加一个NOT NULL 或者CHECK 约束或者用SET DATA TYPE更改一个列类型时，不会检查与外部服务器的一致性。确保该表定义匹配远端是用户的责任。

关于有效参数的进一步描述可参考CREATE FOREIGN TABLE。

## 示例

要把一列标记为非空：

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

要更改一个外部表的选项：

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2
'value2', DROP opt3 'value3');
```

## 兼容性

形式ADD、DROP以及 SET DATA TYPE符合 SQL 标准。其他形式是 SQL 标准的 LightDB扩展。在一个 ALTER FOREIGN TABLE命令中指定多于一个操作也是一种扩展。

ALTER FOREIGN TABLE DROP COLUMN可以被用来删除 一个外部表的唯一一列，从而留下一个没有列的表。这是一种 SQL 的扩展，它 允许没有列的外部表。

## 另见

CREATE FOREIGN TABLE, DROP FOREIGN TABLE

---

# ALTER FUNCTION

ALTER FUNCTION — 更改一个函数的定义

## Synopsis

```
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]  
    action [ ... ] [ RESTRICT ]  
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]  
    RENAME TO new_name  
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]  
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }  
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]  
    SET SCHEMA new_schema  
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]  
    [ NO ] DEPENDS ON EXTENSION extension_name
```

其中 action 是以下之一：

```
CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT  
IMMUTABLE | STABLE | VOLATILE  
[ NOT ] LEAKPROOF  
[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER  
PARALLEL { UNSAFE | RESTRICTED | SAFE }  
COST execution_cost  
ROWS result_rows  
SUPPORT support_function  
SET configuration_parameter { TO | = } { value | DEFAULT }  
SET configuration_parameter FROM CURRENT  
RESET configuration_parameter  
RESET ALL
```

## 描述

ALTER FUNCTION更改一个函数的定义。

你必须拥有该函数以使用ALTER FUNCTION。要更改一个函数的模式，你还必须具有新模式上的CREATE特权。要更改所有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须具有在该函数的模式上的CREATE特权（这些限制强制修改所有者不能做一些通过删除和重建该函数做不到的事情。不过，一个超级用户怎么都能更改任何函数的所有权）。

## 参数

name

一个现有函数的名称（可以被模式限定）。如果没有指定参数列表，则该名称必须在它的模式中唯一。

argmode

一个参数的模式：IN、OUT、INOUT或者VARIADIC。如果被忽略，默认为IN。注意ALTER FUNCTION并不真正关心OUT参数，因为在决定函数的身份时只需要输入参数。因此列出IN、INOUT以及VARIADIC参数即可。

argname

一个参数的名称。注意ALTER FUNCTION 并不真正参数名称，因为在确定函数的身份时只需要参数的数据类型即可。

argtype

该函数的参数（如果有）的数据类型（可以被模式限定）。

new\_name

该函数的新名称。

new\_owner

该函数的新所有者。注意如果该函数被标记为 SECURITY DEFINER，它的后续执行将会使用新所有者。

new\_schema

该函数的新模式。

DEPENDS ON EXTENSION extension\_name

NO DEPENDS ON EXTENSION extension\_name

此格式将功能标记为依赖于扩展名，或者如果指定了NO，则不再依赖于该扩展名。删除扩展名后，会自动删除标记为依赖于扩展名的功能。

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

CALLED ON NULL INPUT将该函数改为在某些 或者全部参数为空值时可以被调用。RETURNS NULL ON NULL INPUT或者 STRICT将该函数改为只要任一参数为空值就不被调用而是自动假定一个空值结果。详见CREATE FUNCTION。

IMMUTABLE

STABLE

VOLATILE

把该函数的稳定性更改为指定的设置。详见 CREATE FUNCTION。

[ EXTERNAL ] SECURITY INVOKER

[ EXTERNAL ] SECURITY DEFINER

更改该函数是否为一个安全性定义者。关键词EXTERNAL 是为了符合 SQL，它会被忽略。关于这项能力的详情请见 CREATE FUNCTION。

PARALLEL

决定该函数对于并行是否安全。详见 CREATE FUNCTION。

LEAKPROOF

更改该函数是否被认为是防泄漏的。关于这项能力的详情请见 CREATE FUNCTION。

COST execution\_cost

更改该函数的估计执行代价。详见CREATE FUNCTION。

ROWS result\_rows

更改一个集合返回函数的估计行数。详见 CREATE FUNCTION。

SUPPORT support\_function

设置或更改计划器支持函数以使用这个函数。详情请参见Section 36.11 你必须是超级用户才可以使用此选项。

此选项不能用于完全删除支持功能，因为它必须命名新的支持函数。如果需要这样做，可以使用CREATE OR REPLACE FUNCTION。

configuration\_parameter  
value

当该函数被调用时，要对一个配置参数做出增加或者更改的赋值。如果 value是DEFAULT 或者使用等价的RESET，该函数本地的设置将会被 移除，这样该函数会使用其环境中存在的值执行。使用RESET ALL可以清除所有函数本地的设置。SET FROM CURRENT把ALTER FUNCTION 执行时该参数的当前值保存为进入 该函数时要应用的值。

有关允许的参数名称和值可详见SET以及 Chapter 17

RESTRICT

为了符合 SQL 标准存在，被忽略。

## 示例

要把用于类型integer的函数sqrt 重命名为square\_root:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

要把用于类型integer的函数sqrt 的拥有者改为joe:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

要把用于类型integer的函数sqrt 的模式改为maths:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

要把类型integer的函数sqrt 标记为依赖于扩展mathlib:

```
ALTER FUNCTION sqrt(integer) DEPENDS ON EXTENSION mathlib;
```

要调整一个函数的自动搜索路径:

```
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;
```

要禁止一个函数的search\_path的自动设置:

```
ALTER FUNCTION check_password(text) RESET search_path;
```

这个函数将用其调用者使用的搜索路径来执行。

## 兼容性

这个语句部分兼容 SQL 标准中的ALTER FUNCTION语句。该标准允许修改一个函数的更多属性，但是不提供 重命名一个函数、标记一个函数为安全性定义者、为一个函数附加配置

参 数值或者更改一个函数的拥有者、模式或者稳定性等功能。该标准还要求 RESTRICT关键字，它在LightDB 中是可选的。

## 另见

CREATE FUNCTION, DROP FUNCTION, ALTER PROCEDURE, ALTER ROUTINE

---

# ALTER GROUP

ALTER GROUP — 更改角色名称或者成员关系

## Synopsis

```
ALTER GROUP role_specification ADD USER user_name [, ... ]
ALTER GROUP role_specification DROP USER user_name [, ... ]
```

其中 `role_specification` 可以是：

```
    role_name
  | CURRENT_USER
  | SESSION_USER
```

```
ALTER GROUP group_name RENAME TO new_name
```

## 描述

ALTER GROUP更改一个用户组的属性。这是一个被废弃的命令，不过为了向后兼容还是会被接受，因为组（以及用户）已经被更一般的角色概念替代了。

前两个变体向一个组增加用户或者从一个组中移除用户（为了这个目的，任何角色都可以扮演“用户”或者“组”）。这些变体实际上等效于在被称为“组”的角色中授予或者回收成员关系，因此最好的方法是使用GRANT或者 REVOKE。

第三种变体会更改该组的名称。这恰好等效于用ALTER ROLE 重命名该角色。

## 参数

`group_name`

要修改的组（角色）的名称。

`user_name`

要被加入到该组或者从该组移除的用户（角色）。这些用户必须已经存在，ALTER GROUP不会创建或者删除用户。

`new_name`

该组的新名称。

## 示例

向一个组增加用户：

```
ALTER GROUP staff ADD USER karl, john;
```

从一个组移除一个用户：

```
ALTER GROUP workers DROP USER beth;
```

## 兼容性

在 SQL 标准中没有ALTER GROUP语句。

## 另见

GRANT, REVOKE, ALTER ROLE



---

# ALTER INDEX

ALTER INDEX — 更改一个索引的定义

## Synopsis

```
ALTER INDEX [ IF EXISTS ] name RENAME TO [schema_name.]new_name
ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name
ALTER INDEX name ATTACH PARTITION index_name
ALTER INDEX name [ NO ] DEPENDS ON EXTENSION extension_name
ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter [= value] [, ... ] )
ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter [, ... ] )
ALTER INDEX [ IF EXISTS ] name ALTER [ COLUMN ] column_number
    SET STATISTICS integer
ALTER INDEX ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
ALTER INDEX [ IF EXISTS ] index_name REBUILD
```

## 描述

ALTER INDEX更改一个现有索引的定义。下面描述了几种子窗体。注意每个子窗体所需的锁级别可能不同。除非显式说明，ACCESS EXCLUSIVE锁被持有。列出多个子命令时，锁的持有将是任何子命令所需的最严格的子命令。ALTER INDEX index\_name REBUILD 类似于REINDEX index name，此语法特性自LightDB 23.2起包含，以实现与Oracle的兼容性。

### RENAME

RENAME形式更改该索引的名称。如果索引与一个表约束（UNIQUE、PRIMARY KEY或者EXCLUDE）关联，该约束也会被重命名。这对已存储的数据没有影响。

重命名索引取得一个 SHARE UPDATE EXCLUSIVE锁。

### SET TABLESPACE

这种形式更改该索引的表空间为指定的表空间，并且把与该索引相关联的数据文件 移动到新的表空间中。要更改一个索引的表空间，你必须拥有该索引并且具有新表空间上的CREATE特权。可以使用 ALL IN TABLESPACE形式把当前数据库中在一个表空间内的所有索引全部移动到另一个表空间中，这将会锁定所有要被移动的索引然后挨个移动它们。这种形式也支持OWNED BY，即只移动属于指定角色的索引。如果指定了NOWAIT选项，那么当该命令无法立刻获得所有锁时将会失败。注意这个命令不会移动系统目录，如果想要移动系统目录，应使用ALTER DATABASE或者显式的 ALTER INDEX调用。另见CREATE TABLESPACE。

### ATTACH PARTITION

导致提到的索引变成附着于被修改的索引。提及的索引必须在包含被修改索引的表的一个分区上，并且具有一种等效的定义。一个附着索引不能被单独删除，它会在其父索引被删除时自动连带删除。

```
DEPENDS ON EXTENSION extension_name
NO DEPENDS ON EXTENSION extension_name
```

这种形式将索引标记为依赖于扩展，或者如果指定了 NO ，则不再依赖于该扩展名。标记为依赖于扩展名的索引会在删除扩展名时自动删除。

SET ( storage\_parameter [= value] [, ... ] )

这种形式为该索引更改一个或者多个索引方法相关的存储参数。可用的参数详见 CREATE INDEX。注意这个命令不会立刻修改索引内容，根据参数你可能需要用REINDEX重建索引来得到想要的效果。

RESET ( storage\_parameter [, ... ] )

这种形式把一个或者多个索引方法相关的存储参数重置为其默认值。正如 SET一样，可能需要一次REINDEX来完全更新该索引。

ALTER [ COLUMN ] column\_number SET STATISTICS integer

这种形式为后续的ANALYZE操作设置针对每个列的统计信息收集目标，不过只能用在被定义为表达式的索引列上。由于表达式缺少唯一的名称，我们通过该索引列的顺序号来引用它们。收集目标可以被设置为范围0到10000之间的值。另外，把它设置为-1会恢复到使用系统的默认统计信息目标 (default\_statistics\_target)。更多有关LightDB查询规划器使用统计信息的内容，请参考Section 14.2

REBUILD

REBUILD 使用索引表中存储的数据重建索引。

## 参数

IF EXISTS

如果该索引不存在不要抛出错误。这种情况下将发出一个提示。

column\_number

引用该索引列的顺序（从左往右）位置的顺序号。

name

要更改的一个现有索引的名称（可能被模式限定）。

new\_name

该索引的新名称。

schema\_name

为了兼容Oracle数据库语法，ALTER INDEX RENAME TO语句支持为新的索引名称指定schema\_name。这个功能仅是语法糖。

tablespace\_name

该索引将被移动到的表空间。

extension\_name

该索引所依赖的扩展的名称。

storage\_parameter

一个索引方法相关的存储参数的名称。

value

一个索引方法相关的存储参数的新值。根据该参数，这可能是一个数字或者一个词。

index\_name

重建索引的名字。

## 注解

也可以用ALTER TABLE来做这些操作。实际上，ALTER INDEX只是ALTER TABLE应用在索引上的形式的别名而已。

以前有一种ALTER INDEX OWNER变体，但现在已被忽略（会出现一个警告）。一个索引的拥有者不能与其基表的拥有者不同。更改基表的拥有者会自动地更改索引的拥有者。

不允许更改系统目录索引的任何部分。

## 示例

要重命名一个现有索引：

```
ALTER INDEX distributors RENAME TO suppliers;
```

把一个索引移动到一个不同的表空间：

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

更改一个索引的填充因子（假设该索引方法支持填充因子）：

```
ALTER INDEX distributors SET (fillfactor = 75);  
REINDEX INDEX distributors;
```

为一个表达式索引设置统计信息收集目标：

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));  
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

在重命名索引时使用schema\_name前缀。

```
create table t1(id int);  
create index hahaha.index_t1_id on t1(id);  
alter index index_t1_id rename to xxww.index_t1_id_1;
```

重建索引：

```
create table t1(id int);  
create index idx_t1_id on t1(id);  
ALTER INDEX idx_t1_id REBUILD;
```

## 兼容性

ALTER INDEX是一种 LightDB扩展。

## 另见

CREATE INDEX, REINDEX

---

# ALTER LANGUAGE

ALTER LANGUAGE — 更改一种过程语言的定义

## Synopsis

```
ALTER [ PROCEDURAL ] LANGUAGE name RENAME TO new_name
ALTER [ PROCEDURAL ] LANGUAGE name OWNER TO { new_owner | CURRENT_USER |
SESSION_USER }
```

## 描述

ALTER LANGUAGE更改一种过程语言的定义。唯一的功能是重命名该语言或者为它赋予一个新的所有者。要使用 ALTER LANGUAGE，你必须是一个超级用户或者该语言的拥有者。

## 参数

name

语言的名称

new\_name

该语言的新名称

new\_owner

该语言的新所有者

## 兼容性

在 SQL 标准中没有ALTER LANGUAGE语句。

## 另见

CREATE LANGUAGE, DROP LANGUAGE

---

# ALTER LARGE OBJECT

ALTER LARGE OBJECT — 更改一个大对象的定义

## Synopsis

```
ALTER LARGE OBJECT large_object_oid OWNER TO { new_owner | CURRENT_USER |  
SESSION_USER }
```

## 描述

ALTER LARGE OBJECT更改一个大对象的定义。

您必须拥有大对象才能使用ALTER LARGE OBJECT。要更改所有者，您还必须是新所有者的直接或间接成员。（不过，超级用户仍然可以更改任何大对象。）当前，唯一的功能是分配新所有者，因此两者的约束都始终适用。

## 参数

large\_object\_oid

要被修改的大对象的 OID

new\_owner

该大对象的新所有者

## 兼容性

在 SQL 标准中没有ALTER LARGE OBJECT 语句。

## 另见

Chapter 33

---

# ALTER MATERIALIZED VIEW

ALTER MATERIALIZED VIEW — 更改一个物化视图的定义

## Synopsis

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    action [, ... ]
ALTER MATERIALIZED VIEW name
    [ NO ] DEPENDS ON EXTENSION extension_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME TO new_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER MATERIALIZED VIEW ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

其中 action是下列之一：

```
ALTER [ COLUMN ] column_name SET STATISTICS integer
ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
MAIN }
ALTER [ COLUMN ] column_name SET COMPRESSION compression_method
CLUSTER ON index_name
SET WITHOUT CLUSTER
SET ( storage_parameter [= value] [, ... ] )
RESET ( storage_parameter [, ... ] )
OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

## 描述

ALTER MATERIALIZED VIEW更改一个现有物化视图的 多个辅助属性。

要使用ALTER MATERIALIZED VIEW，你必须拥有该物化视图。要 更改一个物化视图的模式，你还必须拥有新模式上的CREATE特权。要更 改所有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须拥有该 物化视图所在模式上的CREATE特权（这些限制强制修改所有者不 能做一些通过删除和重建该物化视图做不到的事情。不过，一个超级用户怎么都能更改 任何视图的所有权。）。

可用于ALTER MATERIALIZED VIEW的语句形式和动作是 ALTER TABLE的一个子集，并且在用于物化视图时具有相 同的含义。详见ALTER TABLE的描述。

## 参数

name

一个现有物化视图的名称（可以是模式限定的）。

column\_name

一个新的或者现有的列的名称。

`extension_name`

该物化视图所依赖的扩展的名称（如果指定了 `NO`，则不再依赖）。删除扩展时，会自动删除标记为依赖于扩展的实例化视图。

`new_column_name`

一个现有列的新名称。

`new_owner`

该物化视图的新拥有者的用户名。

`new_name`

该物化视图的新名称。

`new_schema`

该物化视图的新模式。

## 示例

把物化视图 `foo` 重命名为 `bar`：

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

## 兼容性

`ALTER MATERIALIZED VIEW` 是一种 `LightDB` 扩展。

## 另见

`CREATE MATERIALIZED VIEW`, `DROP MATERIALIZED VIEW`, `REFRESH MATERIALIZED VIEW`

---

# ALTER OPERATOR

ALTER OPERATOR — 更改一个操作符的定义

## Synopsis

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    SET SCHEMA new_schema
```

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    SET ( { RESTRICT = { res_proc | NONE }
          | JOIN = { join_proc | NONE }
          } [, ... ] )
```

## 描述

ALTER OPERATOR更改一个操作符的定义。

要使用ALTER OPERATOR，你必须拥有该操作符。要更改拥有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须具有该操作符所在模式上的CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重建操作符做不到的事情。不过，一个超级用户怎么都能更改任何操作符的所有权。）。

## 参数

name

一个现有操作符的名称（可以是模式限定的）。

left\_type

该操作符左操作数的数据类型，如果该操作符没有左操作数可以写成 NONE。

right\_type

该操作符右操作数的数据类型。

new\_owner

该操作符的新拥有者。

new\_schema

该操作符的新模式。

res\_proc

这个操作符的约束选择度估算器函数，写成 NONE 可以移除现有的选择度估算器。

join\_proc

这个操作符的连接选择度估算器函数，写成 NONE 可以移除现有的选择度估算器。



## 示例

更改类型text的一个自定义操作符a @@ b 的拥有者:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

更改类型int[]的自定义操作符a && b的 约束和连接选择度估算器函数:

```
ALTER OPERATOR && (_int4, _int4) SET (RESTRICT = _int_contsel, JOIN =  
_int_contjoinsel);
```

## 兼容性

在 SQL 标准中没有ALTER OPERATOR语句。

## 另见

CREATE OPERATOR, DROP OPERATOR

---

# ALTER OPERATOR CLASS

ALTER OPERATOR CLASS — 更改一个操作符类的定义

## Synopsis

```
ALTER OPERATOR CLASS name USING index_method  
    RENAME TO new_name
```

```
ALTER OPERATOR CLASS name USING index_method  
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

```
ALTER OPERATOR CLASS name USING index_method  
    SET SCHEMA new_schema
```

## 描述

ALTER OPERATOR CLASS更改一个操作符类的定义。

要使用ALTER OPERATOR CLASS，你必须拥有该操作符类。要修改所有者，你还必须是新拥有角色的一个直接或间接成员，并且该角色必须具有该操作符类所在模式上的CREATE特权（这些限制强制修改所有者不能做一些通过删除和重建操作符类做不到的事情。不过，一个超级用户怎么都能更改任何操作符类的所有权。）。

## 参数

name

一个现有操作符类的名称（可以是模式限定的）。

index\_method

这个操作符类所服务的索引方法的名称。

new\_name

该操作符类的新名称。

new\_owner

该操作符类的新所有者。

new\_schema

该操作符类的新模式。

## 兼容性

在 SQL 标准中没有ALTER OPERATOR CLASS语句。

## 另见

CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY

---

# ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY — 更改一个操作符族的定义

## Synopsis

```
ALTER OPERATOR FAMILY name USING index_method ADD
{ OPERATOR strategy_number operator_name ( op_type, op_type )
  [ FOR SEARCH | FOR ORDER BY sort_family_name ]
| FUNCTION support_number [ ( op_type [ , op_type ] ) ]
  function_name [ ( argument_type [, ...] ) ]
} [, ... ]
```

```
ALTER OPERATOR FAMILY name USING index_method DROP
{ OPERATOR strategy_number ( op_type [ , op_type ] )
| FUNCTION support_number ( op_type [ , op_type ] )
} [, ... ]
```

```
ALTER OPERATOR FAMILY name USING index_method
  RENAME TO new_name
```

```
ALTER OPERATOR FAMILY name USING index_method
  OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

```
ALTER OPERATOR FAMILY name USING index_method
  SET SCHEMA new_schema
```

## 描述

ALTER OPERATOR FAMILY更改一个操作符族 的定义。你能增加操作符以及支持函数到该家族、从该族中移除它们或者更改 该族的名称或者拥有者。

在用ALTER OPERATOR FAMILY增加操作符和 支持函数到一个族中时，它们不是族内任何特定操作符类的组成部分，而只是 “松散”地存在于该族中。这表示这些操作符和函数与该族的语义兼容，但是没有被任何特定索引的正确功能所要求（所要求的操作符和函数应该 被作为一个操作符类的一部分声明，见CREATE OPERATOR CLASS）。LightDB将允许一个族的松散成员在 任何时候被从该族中删除，但是在删除一个操作符类的成员之前，必须已经删除整个类以及依赖于该成员的索引。具有代表性的是，单一数据类型操作符和 函数是操作符类的一部分，因为在特定数据类型上的索引需要它们的支持。而 多数据类型操作符和函数则被作为该族的松散成员。

要使用ALTER OPERATOR FAMILY，你必须是超级用户（ 做这样的限制是因为一个错误的操作符族定义可能会迷惑服务器甚至让它崩溃）。

ALTER OPERATOR FAMILY目前不检测操作符族 定义是否包括该索引方法所要求的所有操作符和函数，也不检查操作符和函数是 否形成了一个有理的集合。定义一个合法的操作符族是用户的责任。

进一步的信息请参考Section 36.16

## 参数

name

一个现有操作符族的名称（可以是模式限定的）。

index\_method

这个操作符族所应用的索引方法的名称。

strategy\_number

与该操作符族相关的一个操作符的索引方法策略号。

operator\_name

与该操作符族相关的一个操作符的名称（可以是模式限定的）。

op\_type

在一个OPERATOR子句中指定该操作符的操作数数据类型，或者用NONE来表示一个前缀操作符。不同于 CREATE OPERATOR CLASS中类似的语法，操作数数据类型总是必须被指定。

在一个ADD FUNCTION子句中指定该函数意图支持的操作数数据类型（如果不同于该函数的输入数据类型）。对于 B-树比较函数和哈希函数，有必要指定op\_type，因为该函数的输入数据类型总是正确的。对于B树排序支持功能，B树相等的图像函数以及GiST，SP-GiST和GIN运算符类中的所有函数，必须指定要与该函数一起使用的操作数数据类型。

在一个DROP FUNCTION子句中，必须指定该函数要支持的操作数数据类型。

sort\_family\_name

一个现有btree操作符族的名称（可能是模式限定的），它描述与一个排序操作符相关的排序顺序。

如果既没有指定FOR SEARCH也没有指定FOR ORDER BY，默认值是FOR SEARCH。

support\_number

一个与该操作符族相关的函数的索引方法支持过程编号。

function\_name

作为该操作符族的一种索引方法支持函数的函数名称（可以是模式限定的）。如果没有指定参数列表，则该名称必须在其模式中唯一。

argument\_type

该函数的参数数据类型。

new\_name

该操作符族的新名称。

new\_owner

该操作符族的新所有者。

new\_schema

该操作符族的新模式。

OPERATOR和FUNCTION子句可以以任何顺序出现。

## 注解

注意DROP语法只通过策略或者支持号以及输入数据类型指定该操作符族中的“slot”。占用这个槽的操作符或函数的名称不会被提及。还有，对于DROP FUNCTION，要指定的类型是

该函数意图支持的输入数据类型。对于 GiST 以及 GIN 索引，可能无需对该函数的实际输入参数类型做任何事情。

因为索引机制在使用函数之前不会检查其上的访问权限，包括一个操作符族中的函数或操作符都等同于授予了其上的公共执行权限。这对于操作符族中很有用的这类函数来说，这通常不成问题。

操作符应该由 SQL 函数定义。一个 SQL 函数很可能被内联到调用查询中，这将阻止优化器识别出该查询匹配一个索引。

在 LightDB 8.4 之前，OPERATOR 子句可以包括一个 RECHECK 选项。这不再被支持，因为一个索引操作符是否为“lossy”现在会在运行时即时决定。这允许高效地处理一个操作符可能或者不可能为有损的情况。

## 示例

下列示例命令为一个操作符族增加跨数据类型的操作符和支持函数，该操作符族已经包含用于数据类型 int4 以及 int2 的 B-树操作符类。

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD
```

```
-- int4 vs int2
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
OPERATOR 4 >= (int4, int2) ,
OPERATOR 5 > (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,

-- int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ;
```

再次移除这些项：

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP
```

```
-- int4 vs int2
OPERATOR 1 (int4, int2) ,
OPERATOR 2 (int4, int2) ,
OPERATOR 3 (int4, int2) ,
OPERATOR 4 (int4, int2) ,
OPERATOR 5 (int4, int2) ,
FUNCTION 1 (int4, int2) ,

-- int2 vs int4
OPERATOR 1 (int2, int4) ,
OPERATOR 2 (int2, int4) ,
OPERATOR 3 (int2, int4) ,
OPERATOR 4 (int2, int4) ,
OPERATOR 5 (int2, int4) ,
FUNCTION 1 (int2, int4) ;
```

## 兼容性

在 SQL 标准中没有 ALTER OPERATOR FAMILY 语句。

## 另见

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

---

# ALTER POLICY

ALTER POLICY — 更改一条行级安全性策略的定义

## Synopsis

```
ALTER POLICY name ON table_name RENAME TO new_name
```

```
ALTER POLICY name ON table_name  
  [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]  
  [ USING ( using_expression ) ]  
  [ WITH CHECK ( check_expression ) ]
```

## 描述

ALTER POLICY更改一条现有行级安全性策略的定义。 请注意，ALTER POLICY只允许修改策略所应用的角色集合， 和要修改的USING和WITH CHECK表达式。 要更改策略的其他属性，例如其应用的命令，或者是允许还是限制， 则必须删除并重新创建策略。

要使用ALTER POLICY，你必须拥有该策略所适用的 表。

在ALTER POLICY的第二种形式中，如果指定了角色列表、 using\_expression以及 check\_expression， 它们会被独立地替换。当这些子句之一被省略时，策略的对应部分不会被更改。

## 参数

name

要更改的现有策略的名称。

table\_name

该策略所在的表的名称（可以被模式限定）。

new\_name

该策略的新名称。

role\_name

该策略适用的角色。可以一次指定多个角色。要把该策略 应用于所有角色，可使用PUBLIC。

using\_expression

该策略的USING表达式。详见 CREATE POLICY。

check\_expression

该策略的WITH CHECK表达式。详见 CREATE POLICY。

## 兼容性

ALTER POLICY是一种LightDB扩展。

另见

CREATE POLICY, DROP POLICY



---

# ALTER PROCEDURE

ALTER PROCEDURE — 更改一个过程的定义

## Synopsis

```
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    [ NO ] DEPENDS ON EXTENSION extension_name
```

其中action是下列之一：

```
[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
SET configuration_parameter { TO | = } { value | DEFAULT }
SET configuration_parameter FROM CURRENT
RESET configuration_parameter
RESET ALL
```

## 简介

ALTER PROCEDURE更改一个过程的定义。

要使用ALTER PROCEDURE，你必须拥有该过程。要更改一个过程的方案，你还必须有新方案上的CREATE特权。要更改拥有者，你还必须是新拥有角色的直接或间接成员，并且那个角色在该过程的方案上拥有CREATE特权（这些限制强制更新拥有者无法做到通过删除和重建该过程无法做到的事情。不过，超级用户总是能够更改任何过程的拥有关系）。

## 参数

name

一个现有的过程的名字（可以被方案限定）。如果没有指定参数列表，这个名字必须在其方案中唯一。

argmode

参数的模式：IN，OUT，INOUT，或VARIADIC。如果被省略，默认是IN。

argname

参数的名字。注意ALTER PROCEDURE实际上并不关心参数名，因为仅有参数的数据类型被用来确定过程的身份。

argtype

如果该过程有参数，这是参数的数据类型（可以被方案限定）。有关如何使用参数数据类型来查找过程的详细信息，参见DROP PROCEDURE。

`new_name`

该过程的新名字。

`new_owner`

该过程的新所有者。注意，如果这个过程被标记为SECURITY DEFINER，接下来它将被作为新所有者执行。

`new_schema`

该过程的新方案。

`extension_name`

该过程所依赖的扩展的名称。

[ EXTERNAL ] SECURITY INVOKER

[ EXTERNAL ] SECURITY DEFINER

更改该过程是否为一个安全性定义器。关键词EXTERNAL由于SQL符合性的原因被忽略。更多有关这个能力的信息请见CREATE PROCEDURE。

`configuration_parameter`

`value`

增加或者更改在调用该过程时，要为一个配置参数做的赋值。如果value是DEFAULT或者等效的值，则会使用RESET，过程本地的设置会被移除，这样该过程的执行就会使用其所处环境中的值。使用RESET ALL可以清除所有的过程本地设置。SET FROM CURRENT会把ALTER PROCEDURE执行时该参数的当前值保存为进入该过程时要被应用的值。

关于允许的参数名和参数值的更多信息请见SET和Chapter 17

RESTRICT

为了符合SQL标准会被忽略。

## 示例

要重命名具有两个integer类型参数的过程insert\_data为insert\_record:

```
ALTER PROCEDURE insert_data(integer, integer) RENAME TO insert_record;
```

要把具有两个integer类型参数的过程insert\_data的拥有者改为joe:

```
ALTER PROCEDURE insert_data(integer, integer) OWNER TO joe;
```

要重把具有两个integer类型参数的过程insert\_data的方案改为accounting:

```
ALTER PROCEDURE insert_data(integer, integer) SET SCHEMA accounting;
```

把过程insert\_data(integer, integer)标记为依赖于扩展myext:

```
ALTER PROCEDURE insert_data(integer, integer) DEPENDS ON EXTENSION myext;
```

要调整一个过程自动设置的搜索路径:

```
ALTER PROCEDURE check_password(text) SET search_path = admin, pg_temp;
```

要为一个过程禁用search\_path的自动设置:

```
ALTER PROCEDURE check_password(text) RESET search_path;
```

现在这个过程将用其调用者所使用的任何搜索路径执行。

## 兼容性

这个语句与SQL标准中的ALTER PROCEDURE语句部分兼容。标注你允许修改一个过程的更多性质，但是不提供重命名过程、让过程成为安全性定义器、为过程附加配置参数值或者更改过程的拥有者、方案或者可变性的能力。标准还要求RESTRICT关键字，而它在LightDB中是可选的。

## 另见

CREATE PROCEDURE, DROP PROCEDURE, ALTER FUNCTION, ALTER ROUTINE

---

# ALTER PUBLICATION

ALTER PUBLICATION — 修改发布的定义

## Synopsis

```
ALTER PUBLICATION name ADD TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name DROP TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET ( publication_parameter [= value] [, ... ] )
ALTER PUBLICATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PUBLICATION name RENAME TO new_name
```

## 描述

命令ALTER PUBLICATION可以更改发布的属性。

前三个语句更改哪些表是该发布的一部分。SET TABLE 子句用指定的表替换发布中的表的列表。ADD TABLE和 DROP TABLE子句将从发布中添加和删除一个或多个表。 请注意，将表添加到已订阅的发布中将需要在订阅端执行ALTER SUBSCRIPTION ... REFRESH PUBLICATION操作才能生效。

第四条语句可以改变在CREATE PUBLICATION中指定的所有发布属性。 该命令中未提及的属性保留其先前的设置。

其余语句更改所有者和发布的名称。

你必须拥有该发布才能使用ALTER PUBLICATION。向发布中添加表还需要拥有该表。 要改变所有者，你也必须是新所有者角色的直接或间接成员。新的所有者必须在数据库上拥有CREATE权限。此外，FOR ALL TABLES 发布的新所有者必须是超级用户。但是，超级用户可以更改发布的所有权，而不受这些限制。

## 参数

name

要修改定义的现有发布的名称。

table\_name

现有表的名称。如果在表名之前指定了ONLY，则只有该表受到影响。 如果没有指定ONLY，则该表及其所有后代（如果有的话）都会受到影响。 可选地，可以在表名之后指定\*以明确指示包含后代。

SET ( publication\_parameter [= value] [, ... ] )

该子句修改最初由CREATE PUBLICATION设置的发布参数。

new\_owner

发布的新所有者的用户名。

new\_name

发布的新名称。

## 示例

将发布修改为只发布删除和更新： Change the publication to publish only deletes and updates:

```
ALTER PUBLICATION noinsert SET (publish = 'update, delete');
```

给发布添加一些表:

```
ALTER PUBLICATION mypublication ADD TABLE users, departments;
```

## 兼容性

ALTER PUBLICATION是LightDB的一个扩展。

## 又见

CREATE PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

---

# ALTER ROLE

ALTER ROLE — 更改一个数据库角色

## Synopsis

```
ALTER ROLE role_specification [ WITH ] option [ ... ]
```

其中option可以是：

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
| VALID UNTIL 'timestamp'
```

```
ALTER ROLE name RENAME TO new_name
```

```
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
  SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
  SET configuration_parameter FROM CURRENT
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
  RESET configuration_parameter
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL
```

其中role\_specification可以是：

```
role_name
| CURRENT_USER
| SESSION_USER
```

## 描述

ALTER ROLE更改一个 LightDB角色的属性。

前面列出的这个命令的第一种变体能够更改CREATE ROLE中指定的很多角色属性（覆盖了所有可能的属性，不过没有增加和移除成员关系的选项，如果要增加和移除成员关系可使用GRANT和REVOKE）。该命令中没有提到的属性保持它们之前的设置。数据库超级用户能够更改任何角色的任何这些设置。具有CREATEROLE特权的角色能够更改除SUPERUSER, REPLICATION和BYPASSRLS外的任何这些设置，但是只能为非超级用户和非复制角色修改。普通角色只能更改它们自己的口令。

第二种变体更改该角色的名称。数据库超级用户能重命名任何角色。具有CREATEROLE特权的角色能够重命名任何非超级用户角色。当前的会话用户不能被重命名（如果需要这样做，请以一个不同的用户连接）。由于MD5加密的口令使用角色名作为salt，因此如果一个角色的口令是MD5加密的，重命名该角色会清空其口令。

其余的变体用于更改一个角色的配置变量的会话默认值，可以为所有数据库设置，或者只为IN DATABASE中指定的数据库设置。如果指定的是ALL而不是一个角色名，将会为所有角

色更改该设置。把 ALL和IN DATABASE一起使用实际上和使用命令ALTER DATABASE ... SET ... 相同。

只要改角色后续开始一个新会话，指定的值将会成为该会话的默认值，并且会覆盖 lightdb.conf中存在的值或者从 lightdb命令行收到的值。这只在登录时发生，执行 SET ROLE或者 SET SESSION AUTHORIZATION不会导致新的配置值被设置。对于所有数据库设置的值会被附加到一个角色的数据库相关的设置所覆盖。特定数据库或角色的设置会覆盖为所有角色所作的设置。

超级用户能够更改任何人的会话默认值。具有CREATEROLE特权的角色能够更改非超级用户的默认值。普通角色只能为它们自己设置默认值。某些配置变量不能以这种方式设置，或者只能由一个超级用户发出的命令设置。只有超级用户能够更改所有角色在所有数据库中的设置。

## 参数

name

要对其属性进行修改的角色的名称。

CURRENT\_USER

修改当前用户而不是一个显式标识的角色。

SESSION\_USER

修改当前会话用户而不是一个显式标识的角色。

SUPERUSER

NOSUPERUSER

CREATEDB

NOCREATEDB

CREATEROLE

NOCREATEROLE

INHERIT

NOINHERIT

LOGIN

NOLOGIN

REPLICATION

NOREPLICATION

BYPASSRLS

NOBYPASSRLS

CONNECTION LIMIT connlimit

[ ENCRYPTED ] PASSWORD 'password'

PASSWORD NULL

VALID UNTIL 'timestamp'

这些子句修改原来有CREATE ROLE 设置的属性。更多信息请见 CREATE ROLE参考页。

new\_name

该角色的新名称。

database\_name

要在其中设置该配置变量的数据库名称。

configuration\_parameter

value

把这个角色的指定配置参数的会话默认值设置为给定值。如果 value为DEFAULT 或者等效地使用了RESET，角色相关的变量 设置会被移除，这样该角色将会在新会话中继承系

统范围的默认 设置。使用RESET ALL可清除所有角色相关的 设置。SET FROM CURRENT可以把会话中该参数的 当前值保存为角色相关的值。如果指定了 IN DATABASE, 只会为给定的角色和数据库 设置或者移除该配置参数。

角色相关的变量设置只在登录时生效, SET ROLE以及 SET SESSION AUTHORIZATION不会处理角色 相关的变量设置。

关于允许的参数名称和值详见SET和 Chapter 17

## 注解

使用CREATE ROLE增加新角色, 使用 DROP ROLE移除一个角色。

ALTER ROLE无法更改一个角色成员关系。 可以使用GRANT和 REVOKE来实现。

在使用这个命令指定一个未加密口令时要多加小心。该口令将会以明文 传送到服务器, 并且它还可能被记录在客户端的命令历史或者服务器 日志中。I\_tsql包含了一个命令 \password, 它可以被用来更改一个角色 的口令而不暴露明文口令。

也可以把一个会话默认值绑定到一个指定的数据库而不是一个角色, 详见 ALTER DATABASE。如果出现冲突, 数据库角色相关 的设置会覆盖角色相关的设置, 角色相关的又会覆盖数据库相关的设置。

## 示例

更改一个角色的口令:

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

移除一个角色的口令:

```
ALTER ROLE davide WITH PASSWORD NULL;
```

更改一个口令的失效日期, 指定该口令应该在 2015 年 5 月 4 日中午 (在一个比UTC快 1 小时的时区) 过期:

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

让一个口令永远有效:

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

让一个角色能够创建其他角色和新的数据库:

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

为一个角色指定 maintenance\_work\_mem参数的非默认设置:

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

为一个角色指定 client\_min\_messages参数的数据库相关的非 默认设置:

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```



## 兼容性

ALTER ROLE语句是一个 LightDB扩展。

## 另见

CREATE ROLE, DROP ROLE, ALTER DATABASE, SET

---

# ALTER ROUTINE

ALTER ROUTINE — 更改一个例程的定义

## Synopsis

```
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    [ NO ] DEPENDS ON EXTENSION extension_name
```

其中action是下列之一：

```
IMMUTABLE | STABLE | VOLATILE
[ NOT ] LEAKPROOF
[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
PARALLEL { UNSAFE | RESTRICTED | SAFE }
COST execution_cost
ROWS result_rows
SET configuration_parameter { TO | = } { value | DEFAULT }
SET configuration_parameter FROM CURRENT
RESET configuration_parameter
RESET ALL
```

## 描述

ALTER ROUTINE更改一个例程的定义，它可以是聚集函数、普通函数或者过程。参数的描述、更多的例子以及进一步的细节请参考ALTER AGGREGATE、ALTER FUNCTION以及ALTER PROCEDURE。

## 示例

将类型integer的例程foo重命名为foobar：

```
ALTER ROUTINE foo(integer) RENAME TO foobar;
```

不管foo是聚集、函数还是过程，这个命令都能使用。

## 兼容性

这个语句与SQL标准中的ALTER ROUTINE语句部分兼容。更多细节请参考ALTER FUNCTION和ALTER PROCEDURE。允许例程名称引用聚集函数是一种LightDB的扩展。

## 另见

ALTER AGGREGATE, ALTER FUNCTION, ALTER PROCEDURE, DROP ROUTINE

注意没有CREATE ROUTINE命令。

---

# ALTER RULE

ALTER RULE — 更改一个规则定义

## Synopsis

```
ALTER RULE name ON table_name RENAME TO new_name
```

## 描述

ALTER RULE更改一条现有规则的定义。当前，唯一可用的动作是更改规则的名称。

要使用ALTER RULE，你必须拥有该规则适用的表或者视图。

## 参数

name

要修改的一条现有规则的名称。

table\_name

该规则适用的表或视图的名称（可以是模式限定的）。

new\_name

该规则的新名称。

## 示例

要重命名一条现有的规则：

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

## 兼容性

ALTER RULE是一种 LightDB的语言扩展，整个查询重写系统也是。

## 另见

CREATE RULE, DROP RULE

---

# ALTER SCHEMA

ALTER SCHEMA — 更改一个模式的定义

## Synopsis

```
ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

## 描述

ALTER SCHEMA更改一个模式的定义。

要使用ALTER SCHEMA，你必须拥有该模式。要重命名一个模式，你还必须拥有该数据库的CREATE特权。要更改拥有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须具有该数据库上的CREATE特权（注意超级用户自动拥有所有这些特权）。

## 参数

name

一个现有模式的名称。

new\_name

该模式的新名称。新名称不能以pg\_开始，因为这些名称被保留用于系统模式。

new\_owner

该模式的新拥有者。

## 兼容性

在SQL标准中没有ALTER SCHEMA语句。

## 另见

CREATE SCHEMA, DROP SCHEMA

---

# ALTER SEQUENCE

ALTER SEQUENCE — 更改一个序列发生器的定义

## Synopsis

```
ALTER SEQUENCE [ IF EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
    [ NOORDER ]
    [ ORDER ]
    [ NOCACHE ]
    [ NOCYCLE ]
    [ NOMAXVALUE ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER |
SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

## 描述

ALTER SEQUENCE更改一个现有序列发生器的参数。任何没有在ALTER SEQUENCE命令中明确设置的参数 保持它们之前的设置。

要使用ALTER SEQUENCE，你必须拥有该序列。要更改一个序列 的模式，你还必须拥有新模式上的CREATE特权。要更改所有者，你还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该域的模式上的 CREATE特权（这些限制强制修改所有者不能做一些通过删除和重建该序列做不到的事情。不过，一个超级用户怎么都能更改任何序列的所有权。）。

## 参数

name

要修改的序列的名称（可以是模式限定的）。

IF EXISTS

在序列不存在时不要抛出一个错误。这种情况下会发出一个提示。

data\_type

可选子句AS data\_type 改变序列的数据类型。有效类型是smallint、integer和bigint。

当且仅当先前的最小值和最大值是旧数据类型的最小值或最大值时（换句话说，如果序列是使用NO MINVALUE或NO MAXVALUE， 隐式或显式创建的），则更改数据类型会自动更改序列的最小值和最大值。否则，将保留最小值和最大值，除非将新值作为同一命令的一部分给出。如果最小值和最大值不符合新的数据类型，则会生成错误。

## increment

子句 INCREMENT BY increment 是可选的。一个正值将产生一个上升序列，一个负值会产生一个下降序列。如果没有指定，旧的增量值将被保持。

## minvalue

## NO MINVALUE

可选的子句 MINVALUE minvalue 决定一个序列能产生的最小值。如果指定了 NO MINVALUE，上升序列和下降序列的默认值分别是 1 和数据类型的最小值。如果这些选项都没有被指定，将保持当前的最小值。

## maxvalue

## NO MAXVALUE

可选的子句 MAXVALUE maxvalue 决定一个序列能产生的最大值。如果指定了 NO MAXVALUE，上升序列和下降序列的默认值分别是 数据类型的最大值和 -1。如果这些选项都没有被指定，将保持当前的最大值。

如果 maxvalue 的值大于 INT64\_MAX (9223372036854775807)，则设置 maxvalue 为 sequence 类型所对应的最大值。

## start

可选的子句 START WITH start 更改该序列被记录的开始值。这对于当前序列值没有影响，它会简单地设置未来 ALTER SEQUENCE RESTART 命令将会使用的值。

## restart

可选的子句 RESTART [ WITH restart ] 更改该序列的当前值。这类似于用 is\_called = false 调用 setval 函数：被指定的值将会被下一次 nextval 调用返回。写上没有 restart 值的 RESTART 等效于提供被 CREATE SEQUENCE 记录的或者上一次被 ALTER SEQUENCE START WITH 设置的开始值。

与 setval 调用相比，序列上的 RESTART 操作是事务性的并阻止并发事务从同一序列中获取数字。如果这不是所需的操作模式，则应使用 setval。

## cache

子句 CACHE cache 使得序列数字被预先分配并且保存在内存中以便更快的访问。最小值是 1（每次只产生一个值，即无缓存）。如果没有指定，旧的缓冲值将被保持。

## CYCLE

可选的 CYCLE 关键词可以被用来允许该序列在达到 maxvalue（上升序列）或 minvalue（下降序列）时回卷。如果到达该限制，下一个被产生的数字将分别是 minvalue 或者 maxvalue。

## NO CYCLE

如果指定了可选的 NO CYCLE 关键词，任何在该序列到达其最大值后的 nextval 调用将会返回一个错误。如果既没有指定 CYCLE 也没有指定 NO CYCLE，旧的循环行为将被保持。

## OWNED BY table\_name.column\_name

## OWNED BY NONE

OWNED BY 选项导致该序列与一个特定的表列相关联，这样如果该列（或者整个表）被删除，该序列也会被自动删除。如果指定，这种关联会替代之前为该序列指定的任何关联。被指定的表必须具有相同的拥有者并且与该序列在同一个模式中。指定 OWNED BY NONE 可以移除任何现有的关联，让该序列“自立”。

NOORDER  
ORDER  
NOCACHE  
NOCYCLE

为了兼容Oracle数据库语法，ALTER SEQUENCE语句支持 NOORDER/ORDER/NOCACHE/NOCYCLE 关键字。这些关键字仅是语法糖。

new\_owner

该序列的新拥有者的用户名。

new\_name

该序列的新名称。

new\_schema

该序列的新模式。

NOMAXVALUE

为了兼容Oracle数据库语法，ALTER SEQUENCE语句支持 NOMAXVALUE 关键字。内部逻辑和NO MAXVALUE相同。

## 注解

ALTER SEQUENCE将不会立即影响除当前后端外 其他后端中的nextval结果，因为它们有预分配（缓存）的序列 值。在注意到序列生成参数被更改之前它们将用尽所有缓存的值。当前后端将被 立刻影响。

ALTER SEQUENCE不会影响该序列的 currval状态（在 LightDB 8.3 之前有时会影响）。

ALTER SEQUENCE阻塞并发nextval、 currval、lastval和 setval调用。

由于历史原因，ALTER TABLE也可以被用于序列， 但是只有等效于上述形式的ALTER TABLE变体才被 允许用于序列。

## 示例

在 105 重启一个被称为serial的序列：

```
ALTER SEQUENCE serial RESTART WITH 105;
```

使用ORDER/NOORDER/NOCACHE/NOCYCLE选项来修改sequence。

```
alter sequence lt_seq_test1 nocache nocycle noorder;
alter sequence lt_seq_test1 order nocycle nocache;
```

## 兼容性

ALTER SEQUENCE符合SQL 标准，不过AS、START WITH、 OWNED BY、OWNER TO、RENAME TO 以及SET SCHEMA子句是 LightDB扩展。

## 另见

CREATE SEQUENCE, DROP SEQUENCE

---

# ALTER SERVER

ALTER SERVER — 更改一个外部服务器的定义

## Synopsis

```
ALTER SERVER name [ VERSION 'new_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER SERVER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SERVER name RENAME TO new_name
```

## 描述

ALTER SERVER更改一个外部服务器的定义。第一种形式更改该服务器的版本字符串或者该服务器的一般选项（至少要求一个子句）。第二种形式更改该服务器的所有者。

要修改该服务器，你必须是它的拥有者。此外为了修改所有者，你必须拥有该服务器并且是新拥有角色的一个直接或者间接成员，并且你必须具有该服务器的外部数据包装器上的USAGE特权（注意超级用户自动满足所有这些政策）。

## 参数

name

一个现有服务器的名称。

new\_version

新的服务器版本。

OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )

更改该服务器的选项。ADD、SET和 DROP指定要执行的动作。如果没有显式地指定操作，将会假定为ADD。选项名称必须唯一，名称和值也会使用该服务器的外部数据包装器库进行验证。

new\_owner

该外部服务器的新拥有者的用户名。

new\_name

该外部服务器的新名称。

## 示例

修改服务器foo，增加连接选项：

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

修改服务器foo，更改版本、更改host选项：

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```



## 兼容性

ALTER SERVER符合 ISO/IEC 9075-9 (SQL/MED)。 OWNER TO和RENAME形式是 LightDB 扩展。

## 另见

CREATE SERVER, DROP SERVER

---

# ALTER SESSION

ALTER SESSION — 更改一个会话

## Synopsis

ALTER SESSION [ DISABLE | ENABLE ] PARALLEL [ DML | DDL | QUERY ]

## 兼容性

为了兼容Oracle数据库的用法，新增了ALTER SESSION命令。这个命令只是一个语法糖，且只在Oracle兼容模式下运行。

---

# ALTER STATISTICS

ALTER STATISTICS — 更改扩展统计对象的定义

## Synopsis

```
ALTER STATISTICS name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER STATISTICS name RENAME TO new_name
ALTER STATISTICS name SET SCHEMA new_schema
ALTER STATISTICS name SET STATISTICS new_target
```

## 描述

ALTER STATISTICS更改现有扩展统计对象的参数。任何在ALTER STATISTICS命令中没有明确设定的参数保持它们之前的设置。

您必须拥有统计对象才能使用ALTER STATISTICS。要更改统计对象的模式，还必须在新模式上具有CREATE权限。要更改所有者，还必须是新所有者角色的直接或间接成员，且该角色在统计对象的模式上必须具有CREATE权限。（这些限制强制了通过删除和重新创建统计对象来改变所有者不会做任何你不能做的事情，但是超级用户可以改变任何统计对象的所有权。）

## 参数

name

要修改的统计对象的名称（可能有模式修饰）。

new\_owner

统计对象的新所有者的用户名。

new\_name

统计对象的新名称。

new\_schema

统计对象的新模式。

new\_target

此统计信息对象的统计信息收集目标，用于随后的ANALYZE操作。可以在0到10000范围内设置目标。或者，将其设置为-1以恢复为使用所引用列的统计目标的最大值（如果已设置）或系统默认统计目标（default\_statistics\_target）。有关LightDB查询计划使用统计信息的更多信息，请参考Section 14.2

## 兼容性

SQL标准中没有ALTER STATISTICS命令。

## 又见

CREATE STATISTICS, DROP STATISTICS

---

# ALTER SUBSCRIPTION

ALTER SUBSCRIPTION — 修改订阅的定义

## Synopsis

```
ALTER SUBSCRIPTION name CONNECTION 'conninfo'  
ALTER SUBSCRIPTION name SET PUBLICATION publication_name [, ...] [ WITH  
  ( set_publication_option [= value] [, ... ] ) ]  
ALTER SUBSCRIPTION name REFRESH PUBLICATION [ WITH ( refresh_option [= value]  
  [, ... ] ) ]  
ALTER SUBSCRIPTION name ENABLE  
ALTER SUBSCRIPTION name DISABLE  
ALTER SUBSCRIPTION name SET ( subscription_parameter [= value] [, ... ] )  
ALTER SUBSCRIPTION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }  
ALTER SUBSCRIPTION name RENAME TO new_name
```

## 描述

ALTER SUBSCRIPTION可以修改大部分可以在 CREATE SUBSCRIPTION中指定的订阅属性。

要使用ALTER SUBSCRIPTION，你必须拥有该订阅。要改变所有者，你也必须是新所有者的直接或间接成员。新所有者必须是超级用户。（目前，所有的订阅所有者必须是超级用户，所以所有者的检查将在实践中被绕过，但这可能在未来发生变化。）

## 参数

name

要修改属性的订阅的名称。

CONNECTION 'conninfo'

该子句修改最初由CREATE SUBSCRIPTION设置的连接属性。

SET PUBLICATION publication\_name

更改已订阅发布的列表。SET用新列表替换整个发布列表，ADD将其他发布添加到发布列表中，而DROP将发布从发布列表中删除。有关详细信息，请参见CREATE SUBSCRIPTION。默认情况下，此命令的作用类似于REFRESH PUBLICATION。

publication\_option指定了这个操作的附加选项。支持的选项是：

refresh (boolean)

如果为false，则该命令将不会尝试刷新表信息。然后应单独执行 REFRESH PUBLICATION。默认值是true。

此外，可以指定REFRESH PUBLICATION下描述的选项，以控制隐式刷新操作。

REFRESH PUBLICATION

从发布者获取缺少的表信息。这将开始复制自上次调用REFRESH PUBLICATION 或从CREATE SUBSCRIPTION以来添加到订阅发布中的表。

refresh\_option指定了刷新操作的附加选项。支持的选项有：

copy\_data (boolean)

指定在复制启动后是否应复制正在订阅的发布中的现有数据。默认值是true。（以前订阅的表不会被复制。）

ENABLE

启用先前禁用的订阅，在事务结束时启动逻辑复制工作。

DISABLE

禁用正在运行的订阅，在事务结束时停止逻辑复制工作。

SET ( subscription\_parameter [= value] [, ... ] )

该子句修改原先由CREATE SUBSCRIPTION设置的参数。能够被设置的参数是slot\_name, synchronous\_commit。

new\_owner

订阅的新所有者的用户名。

new\_name

订阅的新名称。

## 示例

将订阅的发布更改为insert\_only:

```
ALTER SUBSCRIPTION mysub SET PUBLICATION insert_only;
```

禁用（停止）订阅:

```
ALTER SUBSCRIPTION mysub DISABLE;
```

## 兼容性

ALTER SUBSCRIPTION是LightDB 的一个扩展。

## 又见

CREATE SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

---

# ALTER SYSTEM

ALTER SYSTEM — 更改一个服务器配置参数

## Synopsis

```
ALTER SYSTEM SET configuration_parameter { TO | = } { value | 'value' |
DEFAULT }
```

```
ALTER SYSTEM RESET configuration_parameter
ALTER SYSTEM RESET ALL
```

## 描述

ALTER SYSTEM被用来在整个数据库集簇范围内更改服务器配置参数。它比传统的手动编辑lightdb.conf文件的方法更方便。ALTER SYSTEM会把给出的参数设置写入到lightdb.auto.conf文件中，该文件会随着lightdb.conf一起被读入。把一个参数设置为DEFAULT或者使用RESET变体可以把该配置项从lightdb.auto.conf文件中移除。使用RESET ALL可以移除所有这类配置项。

用ALTER SYSTEM设置的值将在下一次重载服务器配置后生效，那些只能在服务器启动时更改的参数则会在下一次服务器重启后生效。重载服务器配置可以通过以下做法实现：调用SQL函数pg\_reload\_conf()，运行lt\_ctl reload或者向主服务器进程发送一个SIGHUP信号。

只有超级用户能够使用ALTER SYSTEM。还有，由于这个命令直接作用于文件系统并且不能被回滚，不允许在一个事务块或者函数中使用它。

## 参数

configuration\_parameter

一个可设置配置参数的名称。可用的参数可见Chapter 17

value

该参数的新值。值可以被指定为字符串常量、标识符、数字或者以上这些构成的逗号分隔的列表，值的具体形式取决于特定的参数。写上DEFAULT可以用来把该参数及其值从lightdb.auto.conf中移除。

## 注解

不能用这个命令来设置data\_directory以及lightdb.conf中不被允许的参数（例如preset options）。

其他设置参数的方法见Section 17.1

## 示例

设置wal\_level:

```
ALTER SYSTEM SET wal_level = replica;
```

撤销以上的设置，恢复lightdb.conf中有效的设置:

```
ALTER SYSTEM RESET wal_level;
```

## 兼容性

ALTER SYSTEM语句是一种 LightDB扩展。

## 另见

SET, SHOW

---

# ALTER TABLE

ALTER TABLE — 更改一个表的定义

## Synopsis

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE name WITH UPDATE CURRENT_TIMESTAMP
ALTER TABLE name WITH PRIMARY KEY [ smallserial | serial | bigserial ]
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] name
    ATTACH PARTITION partition_name { FOR VALUES partition_bound_spec |
    DEFAULT }
ALTER TABLE [ IF EXISTS ] name
    DETACH PARTITION partition_name
ALTER TABLE [ IF EXISTS ] name
    ADD PARTITION partition_name { VALUES partition_bound_spec }
ALTER TABLE [ IF EXISTS ] name
    DROP PARTITION partition_name [ UPDATE GLOBAL INDEXES ]
ALTER TABLE [ IF EXISTS ] name
    TRUNCATE PARTITION partition_name [ UPDATE INDEXES ]
ALTER TABLE [ IF EXISTS ] name
    MODIFY PARTITION BY [ RANGE | LIST ] (column) { partition_spec | ONLINE
    | index_spec }
ALTER TABLE [ IF EXISTS ] name
    MODIFY ( [ column modify_column_type modify_column_properites ] [, ... ] )
```

其中action 是以下之一:

```
ADD [ COLUMN ] [ IF NOT EXISTS ] column_definition
ADD ( column_definition [, ...] )
DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation
] [ USING expression ]
ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
ALTER [ COLUMN ] column_name DROP EXPRESSION [ IF EXISTS ]
ALTER [ COLUMN ] column_name ADD GENERATED { ALWAYS | BY DEFAULT } AS
IDENTITY [ ( sequence_options ) ]
ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } |
SET sequence_option | RESTART [ [ WITH ] restart ] } [...]
ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
ADD AUTO_INCREMENT(column_name) [ =N ]
ALTER [ COLUMN ] column_name DROP AUTO_INCREMENT
```



---

```

ALTER [ COLUMN ] column_name SET STATISTICS integer
ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
MAIN }
  ADD table_constraint [ NOT VALID ] [ DISABLE ]
  ADD table_constraint_using_index
  ALTER CONSTRAINT constraint_name [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY
DEFERRED | INITIALLY IMMEDIATE ]
  VALIDATE CONSTRAINT constraint_name
  DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ] [ DROP
INDEX ]
  DISABLE TRIGGER [ trigger_name | ALL | USER ]
  ENABLE TRIGGER [ trigger_name | ALL | USER ]
  ENABLE REPLICA TRIGGER trigger_name
  ENABLE ALWAYS TRIGGER trigger_name
  DISABLE RULE rewrite_rule_name
  ENABLE RULE rewrite_rule_name
  ENABLE REPLICA RULE rewrite_rule_name
  ENABLE ALWAYS RULE rewrite_rule_name
  DISABLE ROW LEVEL SECURITY
  ENABLE ROW LEVEL SECURITY
  FORCE ROW LEVEL SECURITY
  NO FORCE ROW LEVEL SECURITY
  CLUSTER ON index_name
  SET WITHOUT CLUSTER
  SET WITHOUT OIDS
  SET TABLESPACE new_tablespace
  SET { LOGGED | UNLOGGED }
  SET ( storage_parameter [= value] [, ... ] )
  RESET ( storage_parameter [, ... ] )
  INHERIT parent_table
  NO INHERIT parent_table
  OF type_name
  NOT OF
  OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
  REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }

MODIFY column_name data_type [ USING expression ]
MODIFY column_name DEFAULT expression
MODIFY column_name { NOT NULL | NULL } [ENABLE]
MODIFY column_name GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
[ ( sequence_options ) ]
MODIFY column_name DROP IDENTITY
MODIFY column_name CONSTRAINT constraint_name { NOT NULL | NULL } [ENABLE]
MODIFY column_name CONSTRAINT constraint_name UNIQUE [ DEFERRABLE | NOT
DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
MODIFY column_name CONSTRAINT constraint_name PRIMARY KEY [ DEFERRABLE | NOT
DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
MODIFY column_name CONSTRAINT constraint_name REFERENCES reftable
[ ( refcolumn ) ]
  [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE
referential_action ] [ ON UPDATE referential_action ] [ DEFERRABLE | NOT
DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
  MODIFY column_name CONSTRAINT constraint_name CHECK '( a_expr
')' [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
IMMEDIATE ]
  DROP PRIMARY KEY [ RESTRICT | CASCADE ] [ DROP INDEX ]

```

---

并且 column\_definition 是:

```
column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
```

并且 partition\_bound\_spec 是:

```
IN ( partition_bound_expr [, ...] ) |
FROM ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] )
  TO ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal ) |
LESS THAN ( { partition_bound_expr | MAXVALUE } ) |
( partition_bound_expr [, ...] )
```

并且 column\_constraint 是:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  ON UPDATE CURRENT_TIMESTAMP |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH
SIMPLE ]
  [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

而table\_constraint是:

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ...] ) index_parameters |
  PRIMARY KEY ( column_name [, ...] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator
[, ...] ) index_parameters [ WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ...] ) REFERENCES reftable [ ( refcolumn
[, ...] ) ]
  [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE referential_action
] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

并且 table\_constraint\_using\_index 是:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE | PRIMARY KEY } USING INDEX index_name
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

UNIQUE、PRIMARY KEY以及EXCLUDE约束中的index\_parameters是:

```
[ INCLUDE ( column_name [, ...] ) ]
[ WITH ( storage_parameter [= value] [, ...] ) ]
[ USING INDEX | USING INDEX TABLESPACE tablespace_name ]
```

EXCLUDE约束中的exclude\_element是:

```
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

partition\_spec 是:

```
partition_element [, ...]
```

并且 partition\_element 是:

```
[PARTITION name] [VALUES LESS THAN | VALUES] ( [{ partition_value | MAXVALUE } | { partition_value } ] )
```

index\_spec 是:

```
index_element [, ...]
```

并且 index\_element 是:

```
[index name] [ GLOBAL | LOCAL ]
```

modify\_column\_type 是:

```
Typename alter_using
```

并且 alter\_using 是:

```
USING a_expr
```

modify\_column\_properties 是:

```
( [ { NULL | NOT NULL [ ENABLE_P ] | DEFAULT a_expr } ] [ ... ] )
```

## 描述

ALTER TABLE更改一个现有表的定义。下文描述了几种形式。注意每一种形式所要求的锁级别可能不同。如果没有明确说明，将会获得一个ACCESS EXCLUSIVE锁。当给出多个子命令时，获得的锁将是子命令所要求的最严格的那一个。

```
ADD COLUMN [ IF NOT EXISTS ]
```

这种形式向该表增加一个新列，使用与 CREATE TABLE相同的语法。如果指定了 IF NOT EXISTS并且使用这个名字的列已经存在，则不会抛出错误。

```
ADD (column_definition [,...])
```

这种形式可以向该表一次增加多个新列。

```
DROP COLUMN [ IF EXISTS ]
```

这种形式从表删除一列。涉及到该列的索引和表约束也将被自动删除。如果该列的移除会导致引用它的多元统计信息仅包含单一列的数据，则该多元统计信息也将被移除。如果在该表之外有任何东西（例如外键引用或者视图）依赖于该列，你将需要用到CASCADE。如果指定了 IF EXISTS但该列不存在，则不会抛出错误。这种情况中会发出一个提示。

```
SET DATA TYPE
```

这种形式更改表中一列的类型。涉及到该列的索引和简单表约束将通过重新解析最初提供的表达式被自动转换为使用新的列类型。可选的 COLLATE子句为新列指定一种排序规则，如果被省略，排序规则会是新列类型的默认排序规则。可选的USING子句指定如何从旧的列值计算新列值，如果被省略，默认的和从旧类型到新类型的赋值造型一样。如果没有从旧类型到新类型的隐式或者赋值造型，则必须提供一个USING子句。

```
SET/DROP DEFAULT
```

这些表单设置或删除列的默认值（其中删除等同于将默认值设置为NULL）。新的默认值仅适用于后续的INSERT或 UPDATE命令；它不会导致表中已有的行发生更改。

## SET/DROP NOT NULL

这些形式更改一列是否被标记为允许空值或者拒绝空值。

SET NOT NULL 只能应用于列，前提是表中没有任何记录包含该列的NULL值。通常，这一点在ALTER TABLE全表扫描时来检查；但是，如果找到有效的CHECK约束证明不存在NULL，则跳过表扫描。

如果这个表是一个分区，对于在父表中被标记为NOT NULL的列，不能在其上执行DROP NOT NULL。要从所有的分区中删除NOT NULL约束，可以在父表上执行DROP NOT NULL。即使在父表上没有NOT NULL约束，这样的约束还是能被增加到分区上。也就是说，即便父表允许空值，子表也可以不允许空值，但反过来不行。

## DROP EXPRESSION [ IF EXISTS ]

此表单将存储的生成的列转换为普通的基本列。列中的现有数据将会保留，但未来的更改将不会应用于生成的表达式。

如果指定了DROP EXPRESSION IF EXISTS，并且该列不是存储的已生成列，不会抛出任何错误。在这种情况下，将发出通知。

```
ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
SET GENERATED { ALWAYS | BY DEFAULT }
DROP IDENTITY [ IF EXISTS ]
```

这些形式更改一列是否是一个标识列，或者是更改一个已有的标识列的产生属性。详情请参考CREATE TABLE。像SET DEFAULT一样，这些形式仅影响随后的INSERT和UPDATE命令的行为。它们不会导致表中已有的行发生更改。

如果DROP IDENTITY IF EXISTS被指定并且该列不是一个标识列，则不会有错误被抛出。在这种情况下会发出一个提示。

## ADD AUTO\_INCREMENT(column\_name) [ =N ]

此表单将column\_name列更改为自动递增列，默认从1开始，如果set=N，则从N开始。

## DROP AUTO\_INCREMENT

此表单删除列的自动递增属性。删除后，该列无法在上自动递增。

## ON UPDATE CURRENT\_TIMESTAMP

此表单允许在行更新时自动更新时间戳列。

```
SET sequence_option
RESTART
```

这些形式修改位于一个现有标识列之下的序列。sequence\_option是一个ALTER SEQUENCE所支持的选项，例如INCREMENT BY。

## SET STATISTICS

这种形式为后续的ANALYZE操作设置针对每列的统计收集目标。目标可以被设置在范围0到10000之间，还可以把它设置为-1来恢复到使用系统默认的统计目标(default\_statistics\_target)。更多有关LightDB查询规划器使用统计信息的内容可见Section 14.2

SET STATISTICS要求一个SHARE UPDATE EXCLUSIVE锁。

```
SET ( attribute_option = value [, ... ] )
RESET ( attribute_option [, ... ] )
```

这种形式设置或者重置每个属性的选项。当前，已定义的针对每个属性的选项只有n\_distinct和n\_distinct\_inherited，它们会覆盖后续ANALYZE操作所得到的可区分

值数量 估计。`n_distinct`影响该表本身的统计信息，而 `n_distinct_inherited`影响为该表外加其继承子女收集的统计信息。当被设置为一个正值时，ANALYZE将假定该列刚好包含指定 数量的可区分非空值。当被设置为一个负值（必须大于等于 -1）时，ANALYZE将假定可区分非空值的数量与表的尺寸成线性比例， 确切的计数由估计的表尺寸乘以给定数字的绝对值计算得到。例如，值 -1 表示 该列中所有的值都是可区分的，而值 -0.5 则表示每一个值平均出现两次。当表 的尺寸随时间变化时，这会有所帮助，因为这种计算只有在查询规划时才会被 执行。指定值为 0 将回到正常的估计可区分值数量的做法。更多有关 LightDB查询规划器使用统计 信息的内容可见Section 14.2

更改针对每个属性的选项要求一个 SHARE UPDATE EXCLUSIVE锁。

## SET STORAGE

这种形式为一列设置存储模式。这会控制这列是会被保持在线内还是放在一个 二级 TOAST表中，以及数据是否应被压缩。对于 integer之类的定长、线内、未压缩值必须使用 PLAIN。MAIN用于线内、可压缩的 数据。EXTERNAL用于外部的、未压缩数据。而 EXTENDED用于外部的、压缩数据。对于大部分支持 非-PLAIN存储的数据类型，EXTENDED是默认值。使用EXTERNAL将会让很大的 text和bytea之上的子串操作运行得更快，但是代价是存储空间会增加。注意SET STORAGE本身并不改变 表中的任何东西，它只是设置在未来的表更新时要追求的策略。详见 Section 64.2

## ADD table\_constraint [ NOT VALID ] [ DISABLE ]

这种形式使用和CREATE TABLE相同的约束语法外加 NOT VALID选项为一个表增加一个新的约束，该选项 当前只被允许用于外键和 CHECK 约束。

通常，此窗体将导致对表进行扫描，以验证表中的所有现有行是否满足新约束。 但是如果使用了 NOT VALID选项 ，则跳过此可能很漫长的扫描。 该约束仍将被强制到后续的插入和删除上（也就是说，在外键的情况下如果在被引用表中没有一个匹配的行，操作会失败；或者如果新行不匹配指定的检查条件，操作也会失败）。但是数据库不会假定约束对该表中的所有行都成立，直到通过使用VALIDATE CONSTRAINT选项对它进行验证。 参见下述 Notes 以了解关于使用NOT VALID选项的更多信息。

尽管大多数形式的ADDtable\_constraint 都需要ACCESS EXCLUSIVE锁，但ADD FOREIGN KEY仅需要SHARE ROW EXCLUSIVE 锁。 请注意，除了声明约束的表上的锁之外，ADD FOREIGN KEY还获得了对引用表的SHARE ROW EXCLUSIVE锁。

当唯一或者主键约束被添加到分区表时，会有额外的限制，请参考CREATE TABLE。 此外，当前分区表上的外键约束不能被声明为NOT VALID。

DISABLE关键字是为了兼容Oracle语法而新增的，仅是语法糖，没有实际作用。仅限Oracle兼容模式下使用。

## ADD table\_constraint\_using\_index

这种形式基于一个已有的唯一索引为一个表增加新的 PRIMARY KEY或UNIQUE约束。该索引中的所有列将被包括在约束中。

该索引不能有表达式列或者是一个部分索引。还有，它必须是一个带有 默认排序顺序的B-树索引。这些限制确保该索引等效于使用常规 ADD PRIMARY KEY或者ADD UNIQUE命令时创建的索引。

如果PRIMARY KEY被指定，并且该索引的列没有被标记 NOT NULL，那么这个命令将尝试对每一个这样的列做 ALTER COLUMN SET NOT NULL。这需要一次全表扫描 来验证这些列不包含空值。在所有其他情况中，这都是一种很快的操作。

如果提供了一个约束名，那么该索引将被重命名以匹配该约束名。否则 该约束将被命名成索引的名称。

这个命令被执行后，该索引被增加的约束“拥有”，这和用常规 `ADD PRIMARY KEY`或`ADD UNIQUE`命令 创建的索引一样。特别地，删掉该约束将会导致该索引也消失。

当前在分区表上不支持这种形式。

### Note

如果需要增加一个新的约束但是不希望长时间阻塞表更新，那么使用现有索引增加约束会有所帮助。要这样做，用 `CREATE INDEX CONCURRENTLY`创建该索引，并且 接着使用这种语法把它安装为一个正式的约束。例子见下文。

#### ALTER CONSTRAINT

这种形式修改之前创建的一个约束的属性。当前只能修改外键约束。

#### VALIDATE CONSTRAINT

这种形式验证之前创建为`NOT VALID`的外键或检查约束， 它会扫描表来确保对于该约束没有行不满足约束。如果约束已经被标记为合法，则什么也不会发生。（参见下述 Notes 以了解此命令用途的说明。）

此命令需要一个`SHARE UPDATE EXCLUSIVE`锁。

#### DROP CONSTRAINT [ IF EXISTS ]

这种形式在一个表上删除指定的约束，还有位于该约束之下的任何索引。如果`IF EXISTS`被指定并且该约束不存在，不会抛出错误。在这种情况下会发出一个提示。

#### DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER

这些形式配置属于该表的触发器的触发设置。系统仍然知道被禁用触发器 的存在，但是即使它的触发事件发生也不会执行它。对于一个延迟触发器， 会在事件发生时而不是触发器函数真正被执行时检查其启用状态。可以禁 用或者启用用名称指定的单个触发器、表上的所有触发器、用户拥有的触 发器（这个选项会排除内部生成的约束触发器，例如用来实现外键约束或 可延迟唯一和排除约束）。禁用或者启用内部生成的约束触发器要求超级 用户特权，这样做要小心因为如果这类触发器不被执行，约束的完整性当 然无法保证。

触发器引发机制也受到配置变量 `session_replication_role`的影响。当复制角色是 “origin”（默认）或者“local”时，被简单启用的 触发器将被触发。被配置为`ENABLE REPLICA`的触发 器只有在会话处于“replica”模式时才将被触发。被配置为`ENABLE ALWAYS`的触发器的触发不会考虑当前复制 角色。

这种机制的效果就是，在默认配置中，触发器不会在复制体上引发。这种效果很有用，因为如果一个触发器在源头上被用来在表之间传播数据，那么复制系统也将复制被传播的数据，并且触发器不应该在复制体上引发第二次，因为那会导致重复。不过，如果一个触发器被用于另一种目的（例如创建外部告警），那么将它设置为`ENABLE ALWAYS`可能更加合适，这样它在复制体上也会被引发。

这个命令要求一个`SHARE ROW EXCLUSIVE`锁。

#### DISABLE/ENABLE [ REPLICA | ALWAYS ] RULE

这些形式配置属于表的重写规则的触发设置。系统仍然知道一个被禁用规则的 存在，但在查询重写时不会应用它。其语义与禁用的/启用的触发器的一样。 对于`ON SELECT`规则会忽略这个配置，即使当前会话处于 一种非默认的复制角色，这类规则总是会被应用以保持视图工作正常。

规则引发机制也受到配置变量`session_replication_role`的影响，这和上述的触发器类似。

#### DISABLE/ENABLE ROW LEVEL SECURITY

这些形式控制属于该表的行安全性策略的应用。如果被启用并且该表上不存在策略，则将应用一个默认否定的策略。注意即使行级安全性被禁用，在表上还是可以存在策略。在这种情况下，这些策略将不会被应用并且会被忽略。另见CREATE POLICY。

#### NO FORCE/FORCE ROW LEVEL SECURITY

这些形式控制当用户是表所有者时表上的行安全性策略的应用。如果被启用，当用户是表所有者时，行级安全性策略将被应用。如果被禁用（默认），则当用户是表所有者时，行级安全性将不会被应用。另见CREATE POLICY。

#### CLUSTER ON

这种形式为未来的CLUSTER操作选择默认的索引。它不会真正地对表进行聚簇。

改变聚簇选项要求一个SHARE UPDATE EXCLUSIVE锁。

#### SET WITHOUT CLUSTER

这种形式从表中移除最近使用的CLUSTER索引说明。这会影​​响未来的不指定索引的聚簇操作。

改变聚簇选项要求一个SHARE UPDATE EXCLUSIVE锁。

#### SET WITHOUT OIDS

向后兼容的语法，用于删除oid系统列。由于oid系统列无法再添加，所以不会有实际效果。

#### SET TABLESPACE

这种形式把该表的表空间更改为指定的表空间并且把该表相关联的数据文件移动到新的表空间中。表上的索引（如果有）不会被移动，但是它们可以用额外的SET TABLESPACE命令单独移动。当应用于分区表时，不会移动任何内容，但之后用CREATE TABLE PARTITION OF创建的任何分区将使用该表空间，除非被TABLESPACE子句覆盖。

当前数据库在一个表空间中的所有表可以用ALL IN TABLESPACE形式移动，这将会首先锁住所有将被移动的表然后逐个移动。这种形式也支持OWNED BY，它将只移动指定角色所拥有的表。如果指定了NOWAIT选项，则命令将在无法立刻获得所有所需的锁时失败。注意这个命令不移动系统目录；如果想要移动系统目录，应该用ALTER DATABASE或者显式的ALTER TABLE调用。对于这种形式来说，information\_schema关系不被认为是系统目录的一部分，因此它们将会被移动。另见CREATE TABLESPACE。

#### SET { LOGGED | UNLOGGED }

This form changes the table from unlogged to logged or vice-versa (see UNLOGGED). It cannot be applied to a temporary table.

#### SET ( storage\_parameter [= value] [, ... ] )

这种形式为该表更改一个或者更多存储参数。可用的参数请见Storage Parameters的CREATE TABLE。注意这个命令将不会立刻修改表内容，这取决于重写表以得到想要的结果可能需要的参数。可以用VACUUM FULL、CLUSTER或者ALTER TABLE的一种形式来强制一次表重写。对于规划器相关的参数，更改将从该表下一次被锁定开始生效，因此当前执行的查询不会受到影响。

对fillfactor、toast以及autovacuum存储参数，将会拿取SHARE UPDATE EXCLUSIVE锁，就像计划器参数parallel\_workers。

---

RESET ( storage\_parameter [, ... ] )

这种形式把一个或者更多存储参数重置到它们的默认值。和 SET一样，可能需要一次表重写来更新整个表。

INHERIT parent\_table

这种形式把目标表增加为指定父表的一个新子女。随后，针对父亲的查询将包括目标表中的记录。要被增加为一个子女，目标表必须已经包含和父表完全相同的列（也可以有额外的列）。这些列必须具有匹配的数据类型，并且如果它们在父表中具有NOT NULL约束，它们在子表中也必须具有NOT NULL约束。

也必须把子表约束与所有父表的CHECK约束进行匹配，不过父表中那些被标记为非可继承（也就是用ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT 创建的）除外，它们会被忽略。所有匹配得上的子表约束不能被标记为不可继承。当前，UNIQUE、PRIMARY KEY 以及FOREIGN KEY约束没有被考虑，但是这种情况可能会在未来发生变化。

NO INHERIT parent\_table

这种形式把目标表从指定父表的子女列表中移除。针对父表的查询将不再包括来自目标表的记录。

OF type\_name

这种形式把该表链接到一种组合类型，就好像CREATE TABLE OF所做的那样。该表的列名和类型列表必须精确地匹配该组合类型。该表必须不从任何其他表继承。这些限制确保CREATE TABLE OF能允许一个等价的表定义。

NOT OF

这种形式解除一个有类型的表和其类型之间的关联。

OWNER TO

这种形式把表、序列、视图、物化视图或外部表的拥有者改为指定用户。

REPLICA IDENTITY

此表单更改写入预写日志的信息，以标识更新或删除的行。在大多数情况下，只有当每列的旧值与新值不同时，才会记录该值；但是，如果旧值存储在外部，则无论它是否发生了更改，它都会被记录下来。除非正在使用逻辑复制，否则此选项无效。

DEFAULT

（非系统表的默认值）记录主键列（如果有）的旧值。

USING INDEX index\_name

记录被所提到的索引所覆盖的列的旧值，该索引必须是唯一索引、不是部分索引、不是可延迟索引并且只包括被标记成NOT NULL的列。如果删除此索引，则行为与NOTHING相同。

FULL

记录行中所有列的旧值。

NOTHING

Records no information about the old row. This is the default for system tables. 不记录有关旧行的任何信息（这是系统表的默认值）。



## RENAME

RENAME形式更改一个表（或者一个索引、序列、视图、物化视图 或者外部表）的名称、表中一个列的名称或者表的一个约束的名称。在重命名一个具有底层索引的约束时，该索引也会被重命名。它对已存储的数据 没有影响。

## WITH UPDATE CURRENT\_TIMESTAMP

WITH UPDATE CURRENT\_TIMESTAMP表单添加表的隐藏列“ltaut”。“ltaut”列等于“ltaut TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP”。

## WITH PRIMARY KEY [ smallserial | serial | bigserial ]

WITH PRIMARY KEY表单添加表的隐藏列“ltapk”。“ltapk”列会自动递增，并指定为主键约束。如果表已经有主键，则不允许使用WITH PRIMARY KEY。你可以像往常一样删除它并修改它的类型，但只能使用integer、smallint和bigint。

## SET SCHEMA

这种形式将该表移动到另一个模式中。相关的该表列拥有的索引、约束和序列也会被 移动。

## ATTACH PARTITION partition\_name { FOR VALUES partition\_bound\_spec | DEFAULT }

这种形式把一个已有表（自身也可能被分区）作为一个分区挂接到目标表。该表可以为特定的值使用FOR VALUES挂接为分区，或者用DEFAULT挂接为一个默认分区。对于目标表中的每一个索引，在被挂接的表上都创建一个响应的索引，如果已经存在等效的索引，该索引将被挂接到目标表的索引，就像执行了ALTER INDEX ATTACH PARTITION一样。请注意，如果现有表是外表，如果目标表上有UNIQUE索引，则当前不允许将表作为目标表的分区附加。（参见 CREATE FOREIGN TABLE.）对于目标表中存在的每个用户定义的行级触发器，将在附加表中创建一个对应的触发器。

一个使用FOR VALUES的分区使用与CREATE TABLE中partition\_bound\_spec相同的语法。分区边界说明必须对应于目标表的分区策略以及分区键。要被挂接的表必须具有和目标表完全相同的所有列，并且不能有多出来的列，而且列的类型也必须匹配。此外，它必须有目标表上所有的NOT NULL以及CHECK约束。当前不考虑FOREIGN KEY约束。来自于父表的UNIQUE和PRIMARY KEY约束将被创建在分区上（如果它们还不存在）。如果被挂接的表上的任何CHECK约束被标记为NO INHERIT，则命令将失败，这类约束必须被重建且重建时不能有NO INHERIT子句。

如果新分区是一个常规表，会执行一次全表扫描来检查表中现有行没有违背分区约束。可以通过对表增加一个有效的CHECK约束来避免这种扫描，该约束可以在运行这个命令之前仅允许满足所需分区约束的行。CHECK约束可以解决让表无需被扫描就能验证分区约束。但是，如果任一分区键是一个表达式并且该分区不接受NULL值，这种方式就无效了。如果挂接一个不接受NULL值的列表分区，还应该为分区键列增加NOT NULL约束，除非它是一个表达式。

如果新分区是一个外部表，则不需要验证该外部表中的所有行遵守分区约束（有关外部表上的约束请参考CREATE FOREIGN TABLE中的讨论）。

当一个表有默认分区时，定义新分区会更改默认分区的分区约束。默认分区不能包含任何需要被移动到新分区中的行，并且将被扫描以验证不存在那样的行。如果一个合适的CHECK约束存在，这种扫描（和新分区的扫描一样）可以被避免。还是和新分区的扫描一样，当默认分区是外部表时这种扫描总是会被跳过。

在父表上附加一个分区获得一个SHARE UPDATE EXCLUSIVE锁，除了附加的表和默认分区上的ACCESS EXCLUSIVE 锁（如果有）。

如果所附加的表本身是分区表，则还必须所有子分区上保持进一步的锁。同样，如果默认分区本身是分区表。可以通过添加CHECK约束来避免子分区的锁定，如Section 5.11.2. 中所述。

DETACH PARTITION `partition_name`

这种形式会分离目标表的指定分区。被分离的分区继续作为独立的表存在，但是与它之前挂接的表不再有任何联系。任何被挂接到目标表索引的索引也会被分离。任何作为目标表中那些表的副本创建的表也会被删除。SHARE锁在外键约束中引用此分区表的任何表上都可用获得。

ADD PARTITION `partition_name` { VALUES `partition_bound_spec` }

此语法用于为与Oracle兼容的现有分区表创建新分区。支持添加范围分区和列表分区，不支持添加哈希分区。需要将配置更改为Oracle兼容模式。

自LightDB23.3起，Oracle兼容模式下，支持列表默认分区。已经存在默认分区的情况下，添加分区将会失败。

创建分区的时候，子分区不会自动创建。

DROP PARTITION `partition_name`

此语法用于从与Oracle兼容的现有分区表中删除分区。支持丢弃范围分区和列表分区，不支持丢弃哈希分区。需要将配置更改为Oracle兼容模式。参数UPDATE GLOBAL INDEXES仅语法兼容，但函数不兼容。

TRUNCATE PARTITION `partition_name` [ UPDATE INDEXES ]

此语法用于清除一个分区表中的数据，使用时，需要将配置更改为Oracle兼容模式，自23.2开始支持。

MODIFY PARTITION BY [ RANGE | LIST ] (`column`)

此语法用于将普通表转换为分区表。支持转换范围分区和列表分区（不支持自动分区）。不支持转换分布式表。有必要将配置更改为Oracle兼容模式。

除了RENAME、SET SCHEMA、ATTACH PARTITION和DETACH PARTITION之外，所有形式的ALTER TABLE都作用在单个表上，前面这些形式可以被组合成一个多修改的列表被一起应用。例如，可以在一个命令中增加多个列并且/或者修改多个列的类型。对于大型表来说这会特别有用，因为只需要对表做一趟操作。

要使用ALTER TABLE，你必须拥有该表。要更改一个表的模式或者表空间，你还必须拥有新模式或表空间上的CREATE特权。要把一个表作为一个父表的新子表加入，你必须也拥有该父表。此外，要把一个表挂接为另一个表的新分区，你必须拥有被挂接的表。要更改所有者，你还必须是新拥有角色的一个直接或者间接成员，并且该角色必须具有该表的模式上的CREATE特权（这些限制强制修改所有者不能做一些通过删除和重建表做不到的事情。不过，一个超级用户怎么都能更改任何表的所有权。）。要增加一个列、修改一列的类型或者使用OF子句，你还必须具有该数据类型上的USAGE特权。

## 参数

IF EXISTS

如果表不存在则不要抛出一个错误。这种情况下会发出一个提示。

`name`

要修改的一个现有表的名称（可以是模式限定的）。如果在表名前指定了ONLY，则只会修改该表。如果没有指定ONLY，该表及其所有后代（如果有）都会被修改。可选地，在表名后面可以指定\*用来显式地指示包括后代表。

`column_name`

一个新列或者现有列的名称。

`new_column_name`

一个现有列的新名称。

`new_name`

该表的新名称。

`data_type`

一个新列的数据类型或者一个现有列的新数据类型。

`table_constraint`

该表的新的表约束。

`constraint_name`

一个新约束或者现有约束的名称。

CASCADE

自动删除依赖于被删除列或约束的对象（例如引用该列的视图），并且接着删除依赖于那些对象的所有对象（见Section 5.14）。

RESTRICT

如果有任何依赖对象时拒绝删除列或者约束。这是默认行为。

`trigger_name`

一个要禁用或启用的触发器的名称。

ALL

禁用或者启用属于该表的所有触发器（如果有任何触发器是内部产生的约束触发器则需要超级用户特权，例如那些被用来实现外键约束或者可延迟一致性和排他约束的触发器）。

USER

禁用或者启用属于该表的所有触发器，内部产生的约束触发器（例如那些被用来实现外键约束或者可延迟一致性和排他约束的触发器）除外。

`index_name`

一个现有索引的名称。

`storage_parameter`

一个表存储参数的名称。

`value`

一个表存储参数的新值。根据该参数，该值可能是一个数字或者一个词。

`parent_table`

要与这个表关联或者解除关联的父表。

`new_owner`

该表的新拥有者的用户名。

`new_tablespace`

要把该表移入其中的表空间的名称。

`new_schema`

要把该表移入其中的模式的名称。

`partition_name`

要被作为新分区附着到这个表或者从这个表上分离的表的名称。ORACLE兼容模式下，通过ORACLE语法CREATE/ADD/DROP/TRUNCATE/MODIFY创建的分名，与传入的名称相同。分区名与分区表名长度限制为31字符，可通过DBA\_TAB\_PARTITIONS视图查询分区名。可能通过\|d+命令查询表所代表的分区。

`partition_bound_spec`

新分区的分区边界说明。更多细节请参考CREATE TABLE中相同的语法。

DROP INDEX

为了兼容Oracle数据库的DROP CONSTRAINT DROP INDEX和DROP PRIMARY KEY DROP INDEX语句，增加了DROP INDEX后缀的语法糖。

## Notes

关键词COLUMN是噪声，可以被省略。

在使用ADD COLUMN增加一列并且指定了一个非易失性DEFAULT时，默认值会在该语句执行时计算并且结果会被保存在表的元数据中。这个值将被用于所有现有行的该列。如果没有指定DEFAULT，则使用NULL。在两种情况下都不需要重写表。

增加一个带有易失性DEFAULT子句的列或者更改一个现有列的类型将要求重写整个表及其索引。在更改一个现有列的类型时有一种例外：如果 USING子句不更改列的内容并且旧类型在二进制上与新类型可强制转换或者是新类型上的一个未约束域，则不需要重写表。但是受影响列上的任何索引仍必须被重建。对于一个大型表，表和/或索引重建可能会消耗相当多的时间，并且会临时要求差不多两倍的磁盘空间。

增加一个CHECK或者NOT NULL约束要求扫描表以验证现有行符合该约束，但是不要求一次表重写。

类似地，在挂接一个新分区时，它需要被扫描以验证现有行满足该分区约束。

提供在一个ALTER TABLE中指定多个更改的选项的主要原因就是多次表扫描或者重写可以因此被整合成一次。

扫描大型表以验证新的外键或检查约束可能需要很长时间，并且对表的其他更新将锁定，直到ALTER TABLE ADD CONSTRAINT命令被提交。NOT VALID约束选项的主要目的是减少对并发更新添加约束的影响。使用 NOT VALID, ADD CONSTRAINT命令不扫描表，可以立即提交。在之后，VALIDATE CONSTRAINT命令会被发出以验证现有行是否满足约束。验证步骤不需要锁定并发更新，因为它知道其他事务将强制执行它们插入或更新的行的约束；只有预先存在的行需要检查。因此，验证在被更改的表上仅获得一个SHARE UPDATE EXCLUSIVE锁。（如果约束是外键，则 ROW SHARE锁也需要约束的表引用。）除了改进并发性外，在已知该表包含预先存在的违规行为的情况下使用 NOT VALID 和 VALIDATE CONSTRAINT也能有作用。一旦约束就位，就不能再插入新的违规，并且现有问题可以在空闲时纠正，直到VALIDATE CONSTRAINT最终完成。

DROP COLUMN形式不会在物理上移除列，而只是简单地让它对 SQL 操作不可见。后续该表中的插入和更新操作将为该列存储一个空值。因此，删除一个列很快，但是它不会立刻减少

表所占的磁盘空间， 因为被删除列所占用的空间还没有被回收。随着现有列被更新，空间将被逐渐回收。

要强制立即回收被已删除列占据的空间，你可以执行一种能导致全表重写的 ALTER TABLE形式。这种形式会导致重新构造每一个把被删除列替换为空值的行。

ALTER TABLE的重写形式对于 MVCC 是不安全的。在一次表重写之后，如果并发事务使用的是一个在重写发生前取得的快照，该表将对这些并发事务呈现出空表的形态。详见 Section 13.5

SET DATA TYPE的USING选项能实际指定涉及该列旧值的任何表达式。也就是说，它可以不但可以引用要被转换的列，还可以引用其他列。这允许使用SET DATA TYPE语法完成十分普遍的转换。由于这种灵活性，USING表达式不适合于列的默认值（如果有），结果可能不是一个默认值所需的常量表达式。这意味着在没有从旧类型到新类型的隐式或者赋值造型时，即便提供了一个 USING子句，SET DATA TYPE还是可能无法转换默认值。在这种情况下，用DROP DEFAULT删除该默认值，执行ALTER TYPE并且接着使用SET DEFAULT增加一个合适的新默认值。类似的考虑也适用于涉及该列的索引和约束。

如果一个表有任何后代表，在不对后代表做相同操作的情况下，不允许在父表中增加列、重命名列或者更改列的类型。这确保了后代总是具有和父表匹配的列。类似地，如果不对所有后代上的CHECK约束进行重命名，就不能在父表中重命名该CHECK约束，这样CHECK约束也能在父表及其后代之间保持匹配（不过，这个限制不适用于基于索引的约束）。此外，因为从父表中选择也会从其后代中选择，父表上的约束不能被标记为有效，除非它在那些后代上也被标记为有效。在所有这些情况下，ALTER TABLE ONLY都将被拒绝。

只有当一个后代表的列不是从任何其他父表继承而来并且没有该列的独立定义时，一次递归的DROP COLUMN操作才会移除该列。一次非递归的DROP COLUMN（即 ALTER TABLE ONLY ... DROP COLUMN）不会移除任何后代列，而是会把它们标记成独立定义的列。对于一个分区表，一个非递归的DROP COLUMN命令将会失败，因为一个表的所有分区都必须有和分区根节点相同的列。

标识列的动作（ADD GENERATED、SET等、DROP IDENTITY）以及动作TRIGGER、CLUSTER、OWNER和TABLESPACE不会递归到后代表上，也就是说它们执行时总是好像指定了ONLY一样。增加约束的动作仅对没有标记为NO INHERIT的CHECK约束递归。

不允许更改一个系统目录表的任何部分。

可用参数的进一步描述请见CREATE TABLE。Chapter 5中有关于继承的进一步信息。

## 示例

要向一个表增加一个类型为varchar的列：

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

这将导致表中所有现有行都用新列的空值填充。

要添加默认值为非空的列：

```
ALTER TABLE measurements
  ADD COLUMN mtime timestamp with time zone DEFAULT now();
```

现有行将以当前时间填充为新列的值，然后新行直将使用其插入时间。

要添加一列并用不同于默认值的值填充它：

```
ALTER TABLE transactions
```

```
ADD COLUMN status varchar(30) DEFAULT 'old',
ALTER COLUMN status SET default 'current';
```

现有行将用old填充，但是随后的命令的默认值将是current。其效果与在单独的ALTER TABLE命令中发出两个子命令的效果相同。

要从表中删除一列：

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

要在一个操作中更改两个现有列的类型：

```
ALTER TABLE distributors
  ALTER COLUMN address TYPE varchar(80),
  ALTER COLUMN name TYPE varchar(100);
```

通过一个USING子句更改一个包含 Unix 时间戳的整数列为 timestamp with time zone：

```
ALTER TABLE foo
  ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
  USING
    timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

同样的，当该列具有一个不能自动造型成新数据类型的默认值表达式时：

```
ALTER TABLE foo
  ALTER COLUMN foo_timestamp DROP DEFAULT,
  ALTER COLUMN foo_timestamp TYPE timestamp with time zone
  USING
    timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
  ALTER COLUMN foo_timestamp SET DEFAULT now();
```

添加隐藏列“ltaut”：

```
ALTER TABLE distributors WITH UPDATE CURRENT_TIMESTAMP;
```

添加隐藏列“ltapk”：

```
ALTER TABLE distributors WITH PRIMARY KEY serial;
```

重命名一个现有列：

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

重命名一个现有的表：

```
ALTER TABLE distributors RENAME TO suppliers;
```

重命名一个现有的约束：

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

为一列增加一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

从一列移除一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

向一个表及其所有子女增加一个检查约束：

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

只向一个表增加一个检查约束（不为其子女增加）：

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5)
NO INHERIT;
```

（该检查约束也不会被未来的子女继承）。

从一个表及其子女移除一个检查约束：

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

只从一个表移除一个检查约束：

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

（该检查约束仍为子女表保留在某个地方）。

为一个表增加一个外键约束：

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES
addresses (address);
```

为一个表增加一个外键约束，并且尽量不要影响其他工作：

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES
addresses (address) NOT VALID;
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;
```

为一个表增加一个（多列）唯一约束：

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id,
zipcode);
```

为一个表增加一个自动命名的主键约束，注意一个表只能拥有一个主键：

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

把一个表移动到一个不同的表空间：

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

把一个表移动到一个不同的模式：

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

重建一个主键约束，并且在重建索引期间不阻塞更新：

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
    ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

要把一个分区挂接到一个范围分区表上：

```
ALTER TABLE measurement
    ATTACH PARTITION measurement_y2016m07 FOR VALUES FROM ('2016-07-01') TO
    ('2016-08-01');
```

要把一个分区挂接到一个列表分区表上：

```
ALTER TABLE cities
    ATTACH PARTITION cities_ab FOR VALUES IN ('a', 'b');
```

要把一个分区挂接到一个哈希分区表上：

```
ALTER TABLE orders
    ATTACH PARTITION orders_p4 FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

要把一个默认分区挂接到一个分区表上：

```
ALTER TABLE cities
    ATTACH PARTITION cities_partdef DEFAULT;
```

从一个分区表分离一个分区：

```
ALTER TABLE measurement
    DETACH PARTITION measurement_y2015m12;
```

要将范围分区添加到分区表，请执行以下操作：

```
ALTER TABLE oracle_partition_range ADD PARTITION oracle_partition_range_p4
    VALUES LESS THAN (to_date('2022-04-01', 'yyyy-mm-dd'));
```

要将范围分区添加到分区表，请执行以下操作：

```
ALTER TABLE oracle_partition_list ADD PARTITION oracle_partition_list_p4 VALUES
    ('0012');
```

要将列表默认分区添加到分区表，请执行以下操作：（Oracle 兼容模式）

```
ALTER TABLE oracle_partition_list ADD PARTITION oracle_partition_list_pdef
    VALUES (DEFAULT);
```



要从分区表中删除分区，请执行以下操作：

```
ALTER TABLE oracle_partition_range DROP PARTITION oracle_partition_range_p4;
```

要从具有更新全局索引的分区表中删除分区，请执行以下操作：

```
ALTER TABLE oracle_partition_range DROP PARTITION oracle_partition_range_p4
UPDATE GLOBAL INDEXES;
```

清除一个分区表中分区的数据：

```
ALTER TABLE oracle_partition_range TRUNCATE PARTITION oracle_partition_range_p4;
```

要将普通表转换为分区表，请执行以下操作：

```
ALTER TABLE ordinary_table MODIFY PARTITION BY RANGE(a)
(
    PARTITION p1 VALUES LESS THAN(10),
    PARTITION p2 VALUES LESS THAN(30),
    PARTITION p3 VALUES LESS THAN(20)
)ONLINE UPDATE INDEXES
(
    a_i global,
    b_i local,
    c_i local,
    d_i global
);
```

-- list分区

```
ALTER TABLE test_list_tab MODIFY PARTITION BY LIST (id) (
    PARTITION even VALUES (2, 4, 6, 8),
    PARTITION odd VALUES (1, 3, 5, 7),
    PARTITION rest VALUES (DEFAULT));
```

DROP INDEX后缀语法糖的使用演示。

```
create table t1(id number, name varchar2(30));

alter table t1 add constraint pk_t1 primary key (id);

alter table t1 drop constraint pk_t1 drop index;

alter table t1 drop primary key drop index;
```

## 兼容性

形式ADD（没有USING INDEX）、DROP [COLUMN]、DROP IDENTITY、RESTART、SET DEFAULT、SET DATA TYPE（没有USING）、SET GENERATED以及SET sequence\_option服从SQL标准。其他形式都是LightDB对SQL标准的扩展。此外，在单个ALTER TABLE命令中指定多个操作的能力是一种扩展。

ALTER TABLE DROP COLUMN可以被用来删除一个表的唯一的列，从而留下一个零列的表。这是一种SQL的扩展，SQL中不允许零列的表。

ALTER TABLE MODIFY COLUMN 可用于修改表的列。这是用于Oracle兼容性。

## Oracle Compatibility

ALTER TABLE MODIFY COLUMN 可用于修改表的列。

DROP PRIMARY KEY [ RESTRICT | CASCADE ] 可用于删除主键。

## 另见

CREATE TABLE

---

# ALTER TABLESPACE

ALTER TABLESPACE — 更改一个表空间的定义

## Synopsis

```
ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] )
ALTER TABLESPACE name RESET ( tablespace_option [, ... ] )
```

## 描述

ALTER TABLESPACE可以被用于更改一个 表空间的定义。

要更改一个表空间的定义，你必须拥有它。要修改拥有者，你还必须是 新拥有角色的一个直接或间接成员（注意超级用户自动拥有这些特权）。

## 参数

name

一个现有表空间的名称。

new\_name

该表空间的新名称。新名称不能以pg\_开始，因为这类名称被 保留用于系统表空间。

new\_owner

该表空间的新拥有者。

tablespace\_option

要设置或者重置的一个表空间参数。当前，唯一可用的参数是 seq\_page\_cost、random\_page\_cost、effective\_io\_concurrency和maintenance\_io\_concurrency。 为一个特定表空间设定这两个参数值将覆盖规划器对从该表空间中的表读取 页面代价的估计值，以及执行者的预取行为，这些估计值由具有相同名称配置参数建立（见 seq\_page\_cost、random\_page\_cost、effective\_io\_concurrency、maintenance\_io\_concurrency）。 如果一个表空间位于一个比其余 I/O 子系统更快或者更慢的磁盘上时，这些参数就能派上用场。

## 示例

将表空间index\_space重命名为fast\_raid:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

更改表空间index\_space的拥有者:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

## 兼容性

在 SQL 标准中没有 ALTER TABLESPACE 语句。

## 另见

CREATE TABLESPACE, DROP TABLESPACE

---

# ALTER TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION — 更改一个文本搜索配置的定义

## Synopsis

```
ALTER TEXT SEARCH CONFIGURATION name
    ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary
    WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name
ALTER TEXT SEARCH CONFIGURATION name OWNER TO { new_owner | CURRENT_USER |
    SESSION_USER }
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema
```

## 描述

ALTER TEXT SEARCH CONFIGURATION 更改一个文本搜索配置的定义。你可以修改其从记号类型到词典的映射 或者更改该配置的名称或者拥有者。

要使用ALTER TEXT SEARCH CONFIGURATION， 你必须是该配置的拥有者。

## 参数

name

一个现有文本搜索配置的名称（可以是模式限定的）。

token\_type

由该配置的解析器发出的记号类型的名称。

dictionary\_name

在其中查阅指定记号类型的文本搜索字典的名称。如果列出了 多个字典，会按照指定的顺序查阅它们。

old\_dictionary

在映射中要替换的文本搜索字典的名称。

new\_dictionary

被用来替代old\_dictionary 的文本搜索字典的名称。

new\_name

该文本搜索配置的新名称。

new\_owner

该文本搜索配置的新所有者。

new\_schema

该文本搜索配置的新模式。

ADD MAPPING FOR形式会安装一些词典（用列表列出）用于在其中 查阅指定的记号类型。如果对任一记号类型已经有一个映射，则会发生错误。 ALTER MAPPING FOR形式做同样的事情，但是首先会移除这些记号 类型的任何现有映射。ALTER MAPPING REPLACE形式用 new\_dictionary来替换任何位 置上的old\_dictionary。当出 现FOR时，只会为指定的记号类型做这样的事情。如果不出现 FOR，则会为该配置中所有的映射都这样做。 DROP MAPPING形式会移除指定记号类型的所有字典，导致该文本 搜索配置忽略这些类型。除非出 现IF EXISTS，在那些记号类型没有 任何映射时会发生错误。

## 示例

下面的例子把my\_config中任何位置上的english字典 替换为swedish字典。

```
ALTER TEXT SEARCH CONFIGURATION my_config
ALTER MAPPING REPLACE english WITH swedish;
```

## 兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH CONFIGURATION 语句。

## 另见

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

---

# ALTER TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH DICTIONARY — 更改一个文本搜索字典的定义

## Synopsis

```
ALTER TEXT SEARCH DICTIONARY name (  
    option [ = value ] [, ... ]  
)  
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name  
ALTER TEXT SEARCH DICTIONARY name OWNER TO { new_owner | CURRENT_USER |  
    SESSION_USER }  
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema
```

## 描述

ALTER TEXT SEARCH DICTIONARY更改一个 文本搜索字典的定义。你可以更改该字典的与模板相关的选项，或者更改该 字典的名称或者拥有者。

要使用 ALTER TEXT SEARCH DICTIONARY，你必须是超级用户。

## 参数

name

一个现有的文本搜索字典的名称（可以是模式限定的）。

option

要为这个字典设置的与模板相关的选项的名称。

value

用于一个模板相关选项的新值。如果等号和值被忽略，则会从该字典 中移除该选项之前的设置而允许使用默认值。

new\_name

该文本搜索字典的新名称。

new\_owner

该文本搜索字典的新拥有者。

new\_schema

该文本搜索字典的新模式。

模板相关的选项可以以任何顺序出现。

## 示例

下面的命令更改一个基于 Snowball 的字典的停用词列表。其他参数保持不变。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

下面的命令更改语言选项为dutch，并且完全移除停用词选项。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

下面的命令“更新”该字典的定义，但是实际没有做任何更改。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

（之所以能这样做是因为选项移除代码在选项不存在时也不会抱怨）。这种技巧在为该字典更改配置文件时有用：ALTER 将强制现有的数据库会话重读配置文件，否则如果会话之前已经读取过 就不会再次读取。

## 兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH DICTIONARY语句。

## 另见

CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY



---

# ALTER TEXT SEARCH PARSER

ALTER TEXT SEARCH PARSER — 更改一个文本搜索解析器的定义

## Synopsis

```
ALTER TEXT SEARCH PARSER name RENAME TO new_name
ALTER TEXT SEARCH PARSER name SET SCHEMA new_schema
```

## 描述

ALTER TEXT SEARCH PARSER更改一个文本 搜索解析器的定义。当前，唯一支持的功能是更改该解析器的名称。

要使用ALTER TEXT SEARCH PARSER，你必须是超级用户。

## 参数

name

一个现有文本搜索解析器的名称（可以是模式限定的）。

new\_name

该文本搜索解析器的新名称。

new\_schema

该文本搜索解析器的新模式。

## 兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH PARSER语句。

## 另见

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

---

# ALTER TEXT SEARCH TEMPLATE

ALTER TEXT SEARCH TEMPLATE — 更改一个文本搜索模板的定义

## Synopsis

```
ALTER TEXT SEARCH TEMPLATE name RENAME TO new_name
ALTER TEXT SEARCH TEMPLATE name SET SCHEMA new_schema
```

## 描述

ALTER TEXT SEARCH TEMPLATE更改一个 文本搜索模板的定义。当前唯一支持的功能是更改该模板的名称。

要使用ALTER TEXT SEARCH TEMPLATE，你必须是超级用户。

## 参数

name

一个现有文本搜索模板的名称（可以是模式限定的）。

new\_name

该文本搜索模板的新名称。

new\_schema

该文本搜索模板的新模式。

## 兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH TEMPLATE语句。

## 另见

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

---

# ALTER TRIGGER

ALTER TRIGGER — 更改一个触发器的定义

## Synopsis

```
ALTER TRIGGER name ON table_name RENAME TO new_name  
ALTER TRIGGER name ON table_name [ NO ] DEPENDS ON EXTENSION extension_name
```

## 描述

ALTER TRIGGER更改一个现有触发器的属性。 RENAME子句更改给定触发器的名称而不更改其定义。 DEPENDS ON EXTENSION子句把该触发器标记为依赖于一个扩展，这样如果扩展被删除，该触发器也会被自动删除。

要更改一个触发器的属性，你必须拥有该触发器所作用的表。

## 参数

name

要修改的一个现有触发器的名称。

table\_name

这个触发器所作用的表的名称。

new\_name

该触发器的新名称。

extension\_name

触发器依赖于某名称的扩展（如果指定了NO，则不再依赖）。删除扩展时，会自动删除标记为依赖于扩展的触发器。

## 注解

临时启用或者禁用一个触发器的功能由ALTER TABLE而不是 ALTER TRIGGER提供，因为ALTER TRIGGER 无法表示一次性启用或者禁用一个表上所有触发器的选项。

## 示例

要重命名一个现有的触发器：

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

要把一个触发器标记为依赖于一个扩展：

```
ALTER TRIGGER emp_stamp ON emp DEPENDS ON EXTENSION empplib;
```

## 兼容性

ALTER TRIGGER是一种 LightDB的 SQL 标准扩展。

另见

ALTER TABLE

---

# ALTER TYPE

ALTER TYPE — 更改一个类型的定义

## Synopsis

```
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE
| RESTRICT ]
ALTER TYPE name action [, ... ]
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE |
AFTER } neighbor_enum_value ]
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value
ALTER TYPE name SET ( property = value [, ... ] )
```

这里action 是以下之一：

```
    ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE |
RESTRUCT ]
    DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type
[ COLLATE collation ] [ CASCADE | RESTRICT ]
```

## 描述

ALTER TYPE更改一种现有类型的定义。 它有几种形式：

### OWNER

此表单更改类型的所有者。

### RENAME

此表单更改类型的名称。

### SET SCHEMA

这种形式将类型移动到另一个模式。

### RENAME ATTRIBUTE

该形式仅可用于复合类型。它更改类型的单个属性的名称。

### ADD ATTRIBUTE

这种形式为一种组合类型增加一个新属性，使用的语法和 CREATE TYPE相同。

### DROP ATTRIBUTE [ IF EXISTS ]

这种形式从一种组合类型删除一个属性。如果指定了 IF EXISTS并且该属性不存在，则不会抛出错误。 这种情况下会发出一个提示。

### ALTER ATTRIBUTE ... SET DATA TYPE

这种形式更改一种组合类型的一个属性类型。

ADD VALUE [ IF NOT EXISTS ] [ BEFORE | AFTER ]

这种形式为一种枚举类型增加一个新值。可以用BEFORE或者 AFTER一个现有值来指定新值在枚举顺序中的位置。 否则，新项会被增加在值列表的最后。

如果指定了IF NOT EXISTS，该类型已经包含新值时不会发生 错误：会发出一个提示但是不采取其他行动。否则，如果新值已经存在会发生错误。

RENAME VALUE

该形式重命名枚举类型的值。该值在枚举排序中的位置不受影响。 如果指定的值不存在或新名称已存在，则会发生错误。

SET ( property = value [, ... ] )

该表格仅适用于基本类型。 它允许调整可以在CREATE TYPE中设置的基本类型属性的子集。 具体来说，可以更改以下属性：

- 可以将RECEIVE设置为二进制输入函数的名称， 或者设置为NONE删除类型的二进制输入函数。 使用此选项需要超级用户特权。
- 可以将SEND设置为二进制输出函数的名称， 或者设置为NONE删除类型的二进制输入函数。 使用此选项需要超级用户特权。
- 可以将TYPMOD\_IN设置为类型修饰符输入函数的名称， 或者设置为NONE删除类型的类型修饰符输入函数。 使用此选项需要超级用户特权。
- 可以将TYPMOD\_OUT设置为类型修饰符输出函数的名称， 或者设置为NONE删除类型的类型修饰符输出函数的名称。 使用此选项需要超级用户特权。
- 可以将ANALYZE设置为特定于类型的统计信息收集功能的名称， 或者设置为NONE删除该类型的统计信息收集功能。 使用此选项需要超级用户特权。
- SUBSCRIPT可以设置为特定于类型的下标处理程序函数的名称，也可以设置为NONE以删除类型的下注处理程序函数。使用此选项需要超级用户权限。
- 可以将STORAGE设置为plain, extended, external, 或 main (有关这些含义的更多信息参考 Section 64.2)。但是，从plain更改为另一个设置需要超级用户特权（因为它要求该类型的C函数全部是TOAST-ready）， 而从另一个设置更改为 plain 则不是完全允许（因为类型可能已经在数据库中包含TOASTed值）。 请注意，更改此选项本身并不会更改任何存储的数据，它只是设置默认的TOAST策略以用于将来创建的表列。 请参见ALTER TABLE更改现有表列的TOAST策略。

有关这些类型属性的更多详细信息，请参见CREATE TYPE。 请注意，在适当的情况下，基本类型的这些属性中的更改将自动传播到基于该类型的域。

ADD ATTRIBUTE、DROP ATTRIBUTE和ALTER ATTRIBUTE动作 可以被整合到一个多个修改组成的列表中，以便被平行应用。例如， 可以在一个命令中增加多个属性并且/或者修改多个属性的类型。

要使用ALTER TYPE，你必须拥有该类型。要更改 一个类型的模式，你还必须拥有新模式上的CREATE特权。要更改拥有者，你还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该类型的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建该类型做不到的事情。不过，一个超级用户怎么都能更改任何类型的所有权。）。要增加一个属性或者修改一个属性类型，你还必须具有该属性数据类型上的 USAGE特权。

## 参数

name

要修改的一个现有类型的名称（可能被模式限定）。

`new_name`

该类型的新名称。

`new_owner`

该类型新拥有者的用户名。

`new_schema`

该类型的新模式。

`attribute_name`

要增加、修改或者删除的属性名称。

`new_attribute_name`

要被重命名的属性的新名称。

`data_type`

要增加的属性的数据类型，或者是要修改的属性的新类型。

`new_enum_value`

要被增加到一个枚举类型的值列表的新值，或将赋予现有值的新名称。和所有枚举文本一样，它需要被引号引用。

`neighbor_enum_value`

一个现有枚举值，新值应该被增加在紧接着该枚举值之前或者之后的位置上。和所有枚举文本一样，它需要被引号引用。

`existing_enum_value`

现有的应该重命名的枚举值。和所有的枚举文本一样，它需要被引号引用。

`property`

要修改的基本类型属性的名称；参见上面的可能值。

`CASCADE`

自动将操作传播到被更改类型的类型表及其后代。

`RESTRICT`

如果被更改的类型是类型表的类型，则拒绝该操作。这是默认设置。

## 注解

如果ALTER TYPE ... ADD VALUE（增加一个新值到枚举类型的形式）不能在一个事务块中执行，新值不能被使用直到事务被提交之后。

涉及到一个新增加枚举值的比较有时会被只涉及原始枚举值的比较更慢。这通常 只会利用BEFORE或者AFTER来把新值的 排序位置设置为非列表结尾的地方时发生。不过，有时候即使把新值增加到最后时 也会发生这种情况（如果创建了该枚举类型之后出现过 OID 计数器“回卷”，就会发生这种情况）。这种减速通常不明显，但是如果它确实 带来了麻烦，通过删除并且重建该枚举类型或者转储并且重载整个数据库可以重新 得到最优性能。

## 示例

要重命名一个数据类型:

```
ALTER TYPE electronic_mail RENAME TO email;
```

把类型email的拥有者改为 joe:

```
ALTER TYPE email OWNER TO joe;
```

把类型email的模式改为 customers:

```
ALTER TYPE email SET SCHEMA customers;
```

增加一个新属性到一个复合类型:

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

在一个特定的排序位置上为一个枚举类型增加一个新值:

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red' ;
```

重命名一个枚举值:

```
ALTER TYPE colors RENAME VALUE 'purple' TO 'mauve' ;
```

要为现有基本类型创建二进制I/O函数:

```
CREATE FUNCTION mytypesend(mytype) RETURNS bytea ... ;
CREATE FUNCTION mytyperecv(internal, oid, integer) RETURNS mytype ... ;
ALTER TYPE mytype SET (
    SEND = mytypesend,
    RECEIVE = mytyperecv
);
```

## 兼容性

增加和删除属性的变体是 SQL 标准的一部分, 而其他变体是 LightDB 扩展。

## 另见

CREATE TYPE, DROP TYPE



---

# ALTER USER

ALTER USER — 更改一个数据库角色

## Synopsis

```
ALTER USER role_specification [ WITH ] option [ ... ]
```

其中 option 可以是：

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
| VALID UNTIL 'timestamp'
```

```
ALTER USER name RENAME TO new_name
```

```
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
  SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
  SET configuration_parameter FROM CURRENT
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
  RESET configuration_parameter
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL
```

其中 role\_specification 可以是：

```
role_name
| CURRENT_USER
| SESSION_USER
```

## 描述

ALTER USER现在是 ALTER ROLE的一种别名。

## 兼容性

ALTER USER语句是一种 LightDB扩展。SQL 标准把用户的定义留给 实现来处理。

## 另见

ALTER ROLE

---

# ALTER USER MAPPING

ALTER USER MAPPING — 更改一个用户映射的定义

## Synopsis

```
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | SESSION_USER |  
PUBLIC }  
SERVER server_name  
OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

## 描述

ALTER USER MAPPING更改一个用户映射的定义。

一个外部服务器的所有者可以为任何用户修改用于该服务器的用户映射。还有，如果一个用户被授予了外部服务器上的USAGE特权，它就能为它们自己的用户名修改一个用户映射。

## 参数

user\_name

该映射的用户名。CURRENT\_ROLE, CURRENT\_USER, 和USER匹配当前用户的名称。PUBLIC被用来匹配系统中所有当前以及未来的用户名。

server\_name

该用户映射的服务器名。

OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )

为该用户映射更改选项。新选项会覆盖任何之前指定的选项。ADD、SET和DROP指定要被执行的动作。如果没有显式地指定操作，将假定为ADD。选项名称必须为唯一，该服务器的外部数据包装器也会验证选项。

## 示例

为服务器 foo的用户映射bob更改口令：

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password 'public');
```

## 兼容性

ALTER USER MAPPING符合 ISO/IEC 9075-9 (SQL/MED)。有一点细微的语法问题：该标准会忽略FOR关键字。由于CREATE USER MAPPING以及 DROP USER MAPPING都在类似的位置上使用FOR，并且 IBM DB2（作为其他主要的 SQL/MED 实现）也在 ALTER USER MAPPING中要求该关键字，因此为了一致性和互操作性，LightDB 在这里的实现与标准不同。

## 另见

CREATE USER MAPPING, DROP USER MAPPING

---

# ALTER VIEW

ALTER VIEW — 更改一个视图的定义

## Synopsis

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET
    DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER |
    SESSION_USER }
ALTER VIEW [ IF EXISTS ] name RENAME [ COLUMN ] column_name TO new_column_name
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value]
    [, ... ] )
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

## 描述

ALTER VIEW更改一个视图的多种辅助属性（如果想要 修改视图的查询定义，应使用CREATE OR REPLACE VIEW）。

要使用ALTER VIEW，你必须拥有该视图。要更改一个视图的模式， 你还必须具有新模式上的CREATE特权。要更改所有者，你还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该视图的模式上的 CREATE特权（这些限制强制修改所有者不能做一些通过删除和重 建视图做不到的事情。不过，一个超级用户怎么都能更改任何视图的所有权。）。

## 参数

name

一个现有视图的名称（可以是模式限定的）。

column\_name

现有列的名称。

new\_column\_name

现有列的新名称。

IF EXISTS

该视图不存在时不要抛出一个错误。这种情况下会发出一个提示。

SET/DROP DEFAULT

这些形式为一个列设置或者移除默认值。对于任何在该视图上的 INSERT或者UPDATE命令，一个视图列的默认值 会在引用该视图的任何规则或触发器之前被替换进来。因此，该视图的默认值将会 优先于来自底层关系的任何默认值。

new\_owner

该视图的新拥有者的用户名。

new\_name

该视图的新名称。

new\_schema

该视图的新模式。

SET ( view\_option\_name [= view\_option\_value] [, ... ] )

RESET ( view\_option\_name [, ... ] )

设置或者重置一个视图选项。当前支持的选项有：

check\_option (enum)

更改该视图的检查选项。值必须是local 或者cascaded。

security\_barrier (boolean)

更改该视图的安全屏障属性。值必须是一个布尔值，如 true或者false。

## 注解

由于历史原因，ALTER TABLE也可以用于视图，但是 只允许等效于以上形式的ALTER TABLE变体用于视图。

## 示例

把视图foo重命名为 bar：

```
ALTER VIEW foo RENAME TO bar;
```

要为一个可更新视图附加一个默认列值：

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1); -- ts will receive a NULL
INSERT INTO a_view(id) VALUES(2); -- ts will receive the current time
```

## 兼容性

ALTER VIEW是一种LightDB 的 SQL 标准扩展。

## 另见

CREATE VIEW, DROP VIEW

---

# ANALYZE

ANALYZE — 收集有关一个数据库的统计信息

## Synopsis

```
ANALYZE [ ( option [, ...] ) ] [ table_and_columns [, ...] ]  
ANALYZE [ VERBOSE ] [ table_and_columns [, ...] ]
```

其中option可以是：

```
VERBOSE [ boolean ]  
SKIP_LOCKED [ boolean ]
```

table\_and\_columns是：

```
table_name [ ( column_name [, ...] ) ]
```

## 描述

ANALYZE收集一个数据库中的表的内容的统计信息，并且将结果存储在pg\_statistic系统目录中。接下来，查询规划器会使用这些统计信息来帮助确定查询最有效的执行计划。

如果没有table\_and\_columns列表，则ANALYZE处理当前用户有权分析的当前数据库中的每个表和物化视图。使用列表，ANALYZE仅处理那些表。还可以给出表的列名列表，在这种情况下，仅收集这些列的统计信息。

当选项列表用括号括起来时，选项可以按任何顺序来写。带括号的语法是在LightDB 11中添加的；不带括号的语法已弃用。

## 参数

VERBOSE

允许显示进度消息。

SKIP\_LOCKED

规定ANALYZE在开始处理一个关系时不应等待任何冲突的锁被释放：如果关系不能无需等待立即锁定，则跳过该关系。 请注意即使采用此选项，ANALYZE在打开关系的索引或从分区、表继承子级和某些类型的外表获取样本行时仍可能阻塞。 此外，当ANALYZE通常处理指定分区表的所有分区时，如果分区表上有一个冲突的锁，这个选项将导致ANALYZE跳过所有分区表。

boolean

规定所选的选项打开或关闭。 您可以写TRUE、ON或1以启用该选项，或者是FALSE、OFF或0来禁用它。 boolean 值可以被省略，在假定为TRUE的情况下。

table\_name

要分析的一个指定表的名称（可以是模式限定的）。如果省略，则分析当前数据库中的所有常规表、分区表和物化视图（但不包含外部表）。如果指定的表是分区表，则整个分区表的继承统计信息和各个分区的统计信息都将更新。

column\_name

要分析的一个指定列的名称。默认是所有列。

## 输出

当指定了VERBOSE时，ANALYZE会发出进度消息来指示当前正在处理哪个表。还会打印有关那些表的多种统计信息。

## 注解

要分析表，通常必须是该表的所有者或超级用户。不过，数据库所有者可以分析数据库中的所有表，共享目录除外。（对共享目录的限制意味着真正的数据库范围的ANALYZE只能由超级用户执行。）ANALYZE将跳过调用用户没有分析权限的任何表。

只有被显式选中时才会分析外部表。并非所有外部数据包装器都支持ANALYZE。如果表的包装器不支持ANALYZE，该命令会打印一个警告并且什么也不做。

在默认的LightDB配置中，自动清理守护进程（见Section 22.1.0）会在表第一次载入数据或者用常规操作改变时负责表的自动分析。当启用自动清理时，定期运行ANALYZE是个好主意，或者可以在表内容做了大的修改后运行ANALYZE。准确的统计信息将帮助规划器选择最合适的查询计划，从而提升查询处理的速度。主读数据库的一般策略是在一天中使用量最低时运行一次VACUUM和ANALYZE（如果有大量的更新动作则是不够的）。

ANALYZE只要求目标表上的一个读锁，因此它可以和表上的其他动作并行。

ANALYZE收集的统计信息通畅包括每列中最常见值的列表以及展示每列中近似数据分布的一个直方图。如果ANALYZE认为这些东西无趣（例如在一个唯一键列中，没有共同值）或者该列的数据类型不支持合适的操作符，以上工作都会被省略。在Chapter 2中有与统计信息相关的更多信息。

对于大型的表，ANALYZE会对表内容做随机采样而不是检查每一行。这允许在很少的时间内完成对大型表的分析。不过要注意，这些统计信息只是近似值，并且即使实际表内容没有改变，每次运行ANALYZE时统计信息都会有微小地改变。这可能会导致EXPLAIN显示的规划器估算代价有小的改变。在很少的情况下，这会非决定性地导致规划器的查询计划选择在ANALYZE运行后改变。为了避免这种情况，可以按照下文所述提高ANALYZE所收集的统计信息量。

通过调整default\_statistics\_target配置变量可以控制分析量，对每个列可以用ALTER TABLE ... ALTER COLUMN ... SET STATISTICS设置每列的统计信息目标。目标值会设置最常用值列表中的最大项数以及直方图中的最大容器数。默认目标是 100，可以把它调大或者调小在规划器估计值精度和ANALYZE花费的时间以及pg\_statistic所占空间之间做出平衡。特别地，将统计信息目标设置为零会禁用该列的统计信息收集。在查询的WHERE、GROUP BY或者ORDER BY子句中从不出现的列上这样做会有所帮助，因为规划器用不上这些列上的统计信息。

被分析的列中最大的统计信息目标决定了为准备统计信息要采样的表行数。增加该目标会导致做ANALYZE所需的时间和空间成比例增加。

ANALYZE所估算的值之一是出现在每个列中的可区分值。因为只会检查行的一个子集，即便使用最大的统计信息目标，这种估计有时也可能很不精确。如果这种不精确导致不好的查询计划，可以手工确定一个更精确的值并且用ALTER TABLE ... ALTER COLUMN ... SET (n\_distinct = ...)设置该值。

如果被分析的表有一个或者更多子女，ANALYZE将会收集两次统计信息：一次只对父表的行收集，第二次则在父表及其所有子女表的行上收集。在规划需要遍历整个继承树的查询时需要第二个统计信息集。不过，在决定是否触发表上的自动分析时，自动清理后台进程将只考虑父表本身上的插入和更新。如果该表很少被插入或者更新，只有手工运行ANALYZE时才会把继承统计信息更新到最新。

对于分区表，ANALYZE通过对所有分区中的行进行采样来收集统计信息；此外，它将递归到每个分区并更新其统计信息。即使使用多级分区，每个叶分区也只会分析一次。只收集父表的统计信息（没有来自其分区的数据），因为使用分区可以保证它是空的。

相比之下，如果要分析的表具有继承子表，则ANALYZE会收集两组统计信息：一组仅针对父表的行，另一组包括父表及其所有子表的行。当计划将继承树作为一个整体进行处理的查询时，需要第二组统计信息。在这种情况下，不会单独分析子表本身。

autovacuum守护进程不处理分区表，也不处理继承父级（如果只修改了子级）。通常需要定期运行手动ANALYZE，以保持表层次结构的统计信息是最新的。

如果任何子表是外部表并且其外部数据包装器不支持ANALYZE，在收集继承统计信息时会忽略那些子表。

如果被分析的表不完全为空，ANALYZE将不会为该表记录新统计信息。任何现有统计信息将会被保留。

## 兼容性

SQL 标准中没有ANALYZE语句。

## 另见

VACUUM, vacuumdb, Section 17.4.4 Section 22.1.6 Section 25.4.1

---

# BEGIN

BEGIN — 开始一个事务块

## Synopsis

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
```

其中 `transaction_mode` 是以下之一：

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
READ WRITE | READ ONLY
[ NOT ] DEFERRABLE
```

## 描述

BEGIN开始一个事务块，也就是说所有 BEGIN命令之后的所有语句将在一个事务中执行，直到给出一个显式的COMMIT 或者ROLLBACK。默认情况下（没有BEGIN），LightDB在“自动提交”模式中执行事务，也就是说每个语句都在自己的事务中执行并且在语句结束时隐式地执行一次提交（如果执行成功，否则会完成一次回滚）。

在一个事务块内的语句会执行得更快，因为事务的开始/提交也要求可观的CPU和磁盘活动。在进行多个相关更改时，在一个事务内执行多个语句也有助于保证一致性：在所有相关更新还没有完成之前，其他会话将不能看到中间状态。

如果指定了隔离级别、读/写模式或者延迟模式，新事务也会有那些特性，就像执行了SET TRANSACTION一样。

## 参数

WORK  
TRANSACTION

可选的关键词。它们没有效果。

这个语句其他参数的含义请参考 SET TRANSACTION。

## 注解

START TRANSACTION具有和BEGIN 相同的功能。

使用COMMIT或者 ROLLBACK来终止一个事务块。

在已经在一个事务块中时发出BEGIN将惹出一个警告消息。事务状态不会被影响。要在一个事务块中嵌套事务，可以使用保存点（见SAVEPOINT）。

由于向后兼容的原因，连续的 `transaction_modes` 之间的逗号可以被省略。

## 示例

开始一个事务块：



BEGIN;

## 兼容性

BEGIN是一种 LightDB语言扩展。它等效于 SQL 标准的命令START TRANSACTION，它的参考页包含额外的兼容性信息。

DEFERRABLE transaction\_mode 是一种LightDB语言扩展。

附带地，BEGIN关键词被用于嵌入式 SQL 中的一种 不同目的。在移植数据库应用时，我们建议小心对待事务语义。

## 另见

COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT

---

# CALL

CALL — 调用一个过程

## Synopsis

```
CALL name ( [ argument ] [, ...] )
```

## 简介

CALL执行一个过程。

如果过程有任何输出参数，则会返回一个结果行，返回这些参数的值。

## 参数

name

过程的名称（可以被方案限定）。

argument

参数可以包括参数名称，使用语法为 `name => value`。这与普通函数调用的工作方式相同。详细参考Section 4.3

## 注解

用户必须有过程上的EXECUTE特权才能调用它。

要调用一个函数（不是过程），应使用SELECT。

如果在事务块中执行CALL，那么被调用的过程不能执行事务控制语句。只有当CALL在其自身的事务中执行时，才允许过程执行事务控制语句。

PL/pgSQL 以不同的方式处理CALL中的输出参数。详见Section 41.6.3

## 示例

```
CALL do_db_maintenance();
```

## 兼容性

CALL符合SQL标准，除了输出参数的处理。该标准规定，用户应该编写变量来接收输出参数的值。

## 另见

CREATE PROCEDURE

---

# CHECKPOINT

CHECKPOINT — 强制一个WAL日志检查点

## Synopsis

CHECKPOINT

## 描述

一个检查点是WAL日志序列中的一个点，在该点上所有数据文件 已经被更新为反映日志中的信息。所有数据文件将被刷写到磁盘。检查点期间发生的细节可见Section 27.4

CHECKPOINT命令在发出时强制一个 立即的检查点，而不用等待由系统规划的常规检查点（由 Section 17.5.2中的设置控制）。CHECKPOINT不是用来在普通操作中 使用的命令。

如果在恢复期间执行，CHECKPOINT 命令将强制一个重启点（见Section 27.4 而不是写一个新检查点。

只有超级用户能够调用CHECKPOINT。

## 兼容性

CHECKPOINT命令是一种 LightDB语言扩展。

---

# CLOSE

CLOSE — 关闭一个游标

## Synopsis

```
CLOSE { name | ALL }
```

## 描述

CLOSE释放与一个已打开游标相关的资源。在游标被关闭后，不允许在其上做后续的操作。当不再需要使用一个游标时应该关闭它。

当一个事务被COMMIT或者 ROLLBACK终止时，每一个非可保持的已打开游标会被隐式地关闭。当创建一个可保持游标的事务通过 ROLLBACK中止时，该可保持游标会被隐式地关闭。如果该创建事务成功地提交，可保持游标会保持打开，直至执行一个显式的CLOSE或者客户端连接断开。

## 参数

name

要关闭的已打开游标的名称。

ALL

关闭所有已打开的游标。

## 注解

LightDB没有一个显式的 OPEN游标语句，一个游标在被声明时就被认为是打开的。使用DECLARE语句可以声明游标。

通过查询pg\_cursors 系统视图可以看到所有可用的游标。

如果一个游标在一个保存点之后关闭，并且后来回滚到了这个保存点，那么CLOSE不会被回滚，也就是说回滚后游标仍然保持关闭。

## 示例

关闭游标liahona:

```
CLOSE liahona;
```

## 兼容性

CLOSE完全服从 SQL 标准。 CLOSE ALL是一种LightDB 扩展。

## 另见

DECLARE, FETCH, MOVE

---

# CLUSTER

CLUSTER — 根据一个索引聚簇一个表

## Synopsis

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

## 描述

CLUSTER指示LightDB 基于index\_name 所指定的索引来聚簇 table\_name 所指定的表。该索引必须已经定义在 table\_name上。

当一个表被聚簇时，会基于索引信息对它进行物理上的排序。聚簇是一种 一次性的操作：当表后续被更新时，更改没有被聚簇。也就是说，不会尝试根据新行或者被更新行的索引顺序来存储它们（如果想这样做，可以周期性地通过发出该命令重新聚簇。还有，把表的fillfactor存储参数设置为小于 100% 有助于在更新期间保持聚簇顺序，因为如果空间足够会把被更新行保留在同一个页面 中）。

当一个表被更新时，LightDB 会记住它是按照哪个索引聚簇的。形式 CLUSTER table\_name 会使用前面所用的同一个索引对表重新聚簇。你也可以使用 CLUSTER或者ALTER TABLE 的SET WITHOUT CLUSTER形式把索引设置为可用于 未来的聚簇操作，或者清除任何之前的设置。

不带任何参数的CLUSTER会重新聚簇调用用户所拥有的当前数据库中已经被聚簇过的表（如果是超级用户调用，则是 所有已被聚簇过的表）。这种形式的 CLUSTER不能在一个事务块内执行。

当一个表被聚簇时，会在其上要求一个ACCESS EXCLUSIVE锁。这会阻止任何其他数据库操作（包括读和写） 在CLUSTER结束前在该表上操作。

## 参数

table\_name

一个表的名称（可能是模式限定的）。

index\_name

一个索引的名称。

VERBOSE

在每一个表被聚簇时打印一个进度报告。

## 注解

在随机访问一个表中的行时，表中数据的实际顺序是无关紧要的。 不过，如果你想要更多地访问其中一些数据，并且有一个索引把它们分组在一起，使用CLUSTER就会带来好处。如果你从一个表中要求一个范围的被索引值或者多行都匹配的一个单一值，CLUSTER就会有所帮助，因为一旦该索引标识出了第一个匹配行所在的表页，所有其他匹配行很可能就在同一个表页中，并且因此节省了磁盘访问并且 提高了查询速度。

CLUSTER可以使用指定索引上的一次索引扫描 或者遵循排序的一次顺序扫描（如果索引是 B 树）对表重新排序。 它将会基于规划器代价参数以及可用的统计信息来选择较快的方法。

在使用索引扫描时，会创建该表的一份临时拷贝，其中包含按索引顺序排列的表数据。该表上每一个索引的临时拷贝也会被创建。因此，在磁盘上需要至少等于表尺寸加上索引尺寸的综合的空闲空间。

在使用顺序扫描以及排序时，也会创建一个临时排序文件，因此临时空间需求的峰值也就是表尺寸的两倍外加索引尺寸。这种方法通常比索引扫描方法更快，但是如果磁盘空间需求是不能接受的，你可以通过临时地把enable\_sort设置为off来禁用这种选择。

建议在聚簇前把maintenance\_work\_mem设置为一个合理地比较大的值（但是不能超过你可以用于CLUSTER操作的RAM容量）。

因为规划器会记录有关表顺序的统计信息，建议在新近被聚簇的表上运行ANALYZE。否则，规划器可能会产生很差的查询计划。

因为CLUSTER会记住哪些索引被聚簇，我们可以第一次手动聚簇想要聚簇的表，然后设置一个定期运行的维护脚本，其中执行不带任何参数的CLUSTER，这样那些表就会被周期性地重新聚簇。

## 示例

基于索引employees\_ind聚簇表 employees:

```
CLUSTER employees USING employees_ind;
```

使用之前用过的同一个索引聚簇employees表:

```
CLUSTER employees;
```

对数据库中以前被聚簇过的所有表进行聚簇:

```
CLUSTER;
```

## 兼容性

在 SQL 标准中没有CLUSTER语句。

为了兼容 8.3 之前的LightDB版本，

```
CLUSTER index_name ON table_name
```

语法也被支持。

## 另见

clusterdb

---

# COMMENT

COMMENT — 定义或者更改一个对象的注释

## Synopsis

```
COMMENT ON
{
ACCESS METHOD object_name |
AGGREGATE aggregate_name ( aggregate_signature ) |
CAST (source_type AS target_type) |
COLLATION object_name |
COLUMN relation_name.column_name |
CONSTRAINT constraint_name ON table_name |
CONSTRAINT constraint_name ON DOMAIN domain_name |
CONVERSION object_name |
DATABASE object_name |
DOMAIN object_name |
EXTENSION object_name |
EVENT TRIGGER object_name |
FOREIGN DATA WRAPPER object_name |
FOREIGN TABLE object_name |
FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
INDEX object_name |
LARGE OBJECT large_object_oid |
MATERIALIZED VIEW object_name |
OPERATOR operator_name (left_type, right_type) |
OPERATOR CLASS object_name USING index_method |
OPERATOR FAMILY object_name USING index_method |
POLICY policy_name ON table_name |
[ PROCEDURAL ] LANGUAGE object_name |
PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
PUBLICATION object_name |
ROLE object_name |
ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
RULE rule_name ON table_name |
SCHEMA object_name |
SEQUENCE object_name |
SERVER object_name |
STATISTICS object_name |
SUBSCRIPTION object_name |
TABLE object_name |
TABLESPACE object_name |
TEXT SEARCH CONFIGURATION object_name |
TEXT SEARCH DICTIONARY object_name |
TEXT SEARCH PARSER object_name |
TEXT SEARCH TEMPLATE object_name |
TRANSFORM FOR type_name LANGUAGE lang_name |
TRIGGER trigger_name ON table_name |
TYPE object_name |
VIEW object_name
} IS 'text'
```

其中 aggregate\_signature 是:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
] argtype [ , ... ]
```

## 描述

COMMENT存储关于一个数据库对象的注释。

对每一个对象只保存一个注释字符串，因此为了修改一段注释，对同一个对象发出一个新的COMMENT命令。要移除一段注释，可在文本字符串的位置上写上NULL。当对象被删除时，其注释也会被自动删除。

A SHARE UPDATE EXCLUSIVE lock is acquired on the object to be commented.

对大部分类型的对象，只有对象的拥有者可以设置注释。角色没有拥有者，因此COMMENT ON ROLE的规则是你必须作为一个超级用户来对一个超级用户角色设置注释，或者具有CREATEROLE特权来对非超级用户角色设置注释。同样的，访问方法也没有拥有者，你必须作为一个超级用户来对一个访问方法设置注释。当然，一个超级用户可以对任何东西设置注释。

使用ltsql的\d命令家族可以查看注释。其他检索注释的用户接口可以构建在ltsql使用的内建函数之上，即obj\_description、col\_description以及shobj\_description（见Table 9.74）。

## 参数

```
object_name
relation_name.column_name
aggregate_name
constraint_name
function_name
operator_name
policy_name
procedure_name
routine_name
rule_name
trigger_name
```

要被注释的对象的名称。驻留在模式（表、函数等）中的对象的名称可以是模式限定的。在注释一列时，relation\_name必须引用一个表、视图、组合类型或者外部表。

```
table_name
domain_name
```

当在一个约束、触发器、规则或者策略上创建一段注释时，这些参数指定在其上定义该对象的表或域的名称。

```
source_type
```

造型的源数据类型的名称。

```
target_type
```

造型的目标数据类型的名称。

```
argmode
```

一个函数，存储过程或者聚集函数的参数的模式：IN、OUT、INOUT或者VARIADIC。如果被省略，默认值是IN。注意COMMENT并不真正关心OUT参数，因为决定函数的身份只需要输入参数。因此列出IN、INOUT和VARIADIC参数就足够了。



argname

一个函数，存储过程或者聚集函数参数的名称。注意 COMMENT并不真正关心参数名称，因为决定函数的身份只需要参数数据类型。

argtype

一个函数，存储过程或者聚集函数参数的数据类型。

large\_object\_oid

大对象的 OID。

left\_type

right\_type

操作符的参数的数据类型（可以是模式限定的）。对一个前缀操作符 的缺失参数可以写NONE。

PROCEDURAL

这是一个噪声词。

type\_name

该转换的数据类型的名称。

lang\_name

该转换的语言的名称。

text

写成一个字符串的新注释。如果要删除注释，写成NULL。

## 注解

当前对查看注释没有安全机制：任何连接到一个数据库的用户能够看到 该数据库中所有对象的注释。对于数据库、角色、表空间这类共享对象， 注释被全局存储，因此连接到集群中任何数据库的任何用户可以看到共享对象的所有注释。因此，不要在注释中放置有安全性风险的信息。

## 示例

为表mytable附加一段注释：

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

移除它：

```
COMMENT ON TABLE mytable IS NULL;
```

更多的一些例子：

```
COMMENT ON ACCESS METHOD gin IS 'GIN index access method';  
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample  
variance';  
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
```

```
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Constrains column col';
COMMENT ON CONSTRAINT dom_col_constr ON DOMAIN dom IS 'Constrains col of
domain';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON EVENT TRIGGER abort_ddl IS 'Aborts all DDL commands';
COMMENT ON EXTENSION hstore IS 'implements the hstore data type';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'my foreign data wrapper';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Employee Information in other
database';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON MATERIALIZED VIEW my_matview IS 'Summary of order history';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, integer) IS 'Unary minus';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for
btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators for
btrees';
COMMENT ON POLICY my_policy ON mytable IS 'Filter rows by users';
COMMENT ON PROCEDURE my_proc (integer, integer) IS 'Runs a report';
COMMENT ON PUBLICATION alltables IS 'Publishes all operations on all tables';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON ROUTINE my_routine (integer, integer) IS 'Runs a routine (which is a
function or procedure)';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON SERVER myserver IS 'my foreign server';
COMMENT ON STATISTICS my_statistics IS 'Improves planner row estimations';
COMMENT ON SUBSCRIPTION alltables IS 'Subscription for all operations on all
tables';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for Swedish
language';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Splits text into words';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRANSFORM FOR hstore LANGUAGE plpythonu IS 'Transform between hstore
and Python dict';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

## 兼容性

SQL 标准中没有COMMENT命令。

---

# COMMIT

COMMIT — 提交当前事务

## Synopsis

```
COMMIT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

## 描述

COMMIT提交当前事务。所有由该事务所作的更改会变得对他人可见并且被保证在崩溃发生时仍能持久。

## 参数

WORK  
TRANSACTION

可选的关键词。它们没有效果。

AND CHAIN

如果指定了AND CHAIN，则立即启动与刚刚完成的事务具有相同事务特征（参见SET TRANSACTION）的新事务。否则，没有新事务被启动。

## 注解

使用ROLLBACK中止一个事务。

当不在一个事务内时发出COMMIT不会产生危害，但是它会产生一个警告消息。当COMMIT AND CHAIN不在事务内时是一个错误。

## 示例

要提交当前事务并且让所有更改持久化：

```
COMMIT;
```

## 兼容性

命令COMMIT符合 SQL 标准。表单COMMIT TRANSACTION为LightDB扩展。

## 另见

BEGIN, ROLLBACK

---

# COMMIT PREPARED

COMMIT PREPARED — 提交一个早前为两阶段提交预备的事务

## Synopsis

```
COMMIT PREPARED transaction_id
```

## 描述

COMMIT PREPARED提交一个处于预备状态的事务。

## 参数

transaction\_id

要被提交的事务的事务标识符。

## 注解

要提交一个预备的事务，你必须是原先执行该事务的同一用户或者超级用户。 但是不需要处于执行该事务的同一会话中。

这个命令不能在一个事务块中执行。该预备事务将被立刻提交。

pg\_prepared\_xacts 系统视图中列出了所有当前可用的预备事务。

## 例子

提交由事务标识符foobar标识的事务：

```
COMMIT PREPARED 'foobar';
```

## 兼容性

COMMIT PREPARED是一种 LightDB扩展。其意图是用于 外部事务管理系统，其中有些已经被标准涵盖（例如 X/Open XA），但是那些系统的 SQL 方面未被标准化。

## 另见

PREPARE TRANSACTION, ROLLBACK PREPARED

---

# COPY

COPY — 在一个文件和一个表之间复制数据

## Synopsis

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]
    [ WHERE condition ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]
```

其中 option 可以是下列之一：

```
FORMAT format_name
FREEZE [ boolean ]
DELIMITER 'delimiter_character'
NULL 'null_string'
HEADER [ boolean ]
QUOTE 'quote_character'
ESCAPE 'escape_character'
FORCE_QUOTE { ( column_name [, ...] ) | * }
FORCE_NOT_NULL ( column_name [, ...] )
FORCE_NULL ( column_name [, ...] )
ENCODING 'encoding_name'
```

## 描述

COPY在 LightDB表和标准文件系统文件之间 移动数据。COPY TO把一个表的内容复制 到一个文件，而COPY FROM 则从一个文件复制数据到一个表（把数据追加到表中原有数 据）。COPY TO也能复制一个 SELECT查询的结果。

如果指定了一个列列表，COPY TO将只把指定列的数据复制到文件。对于COPY FROM，文件中的每个字段将按顺序插入到指定列中。COPY FROM命令的列列表中并没有指定的表列则会采纳其默认值。

带一个文件名的COPY指示 LightDB服务器直接从一个文件读取 或者写入到一个文件。该文件必须是 LightDB用户（运行服务器的用户 ID） 可访问的并且应该以服务器的视角来指定其名称。当指定了 PROGRAM时，服务器执行给定的命令并且从该程序的标准 输出读取或者写入到该程序的标准输入。该程序必须以服务器的视角指定，并且 必须是LightDB用户可执行的。在指定 STDIN或者STDOUT时，数据会通过客 户端和服务器之间的连接传输。

## 参数

table\_name

一个现有表的名称（可以是模式限定的）。

column\_name

可选的要被复制的列列表。如果没有指定列列表，则该表的所有列除了生成的列都会被复制。

---

## query

其结果要被复制的SELECT、VALUES、INSERT、UPDATE或者DELETE命令。注意查询周围的圆括号是必要的。

对于INSERT、UPDATE以及DELETE查询，必须提供一个RETURNING子句并且目标关系不能具有会扩展成多条语句的条件规则、ALSO规则或者INSTEAD规则。

## filename

输入或者输出文件的路径名。一个输入文件的名称可以是一个绝对或相对路径，但一个输出文件的名称必须是绝对路径。Windows用户可能需要使用一个'E'字符串并且双写路径名称中使用的任何反斜线。

## PROGRAM

一个要执行的命令。在COPY FROM中，输入将从该命令的标准输出读取，而在COPY TO中，输出会写入到该命令的标准输入。

注意该命令是由shell调用，因此如果你需要传递任何来自不可信来源的参数给shell命令，你必须小心地剥离那些可能对shell有特殊意义的特殊字符。出于安全原因，最好使用一个固定的命令字符串，或者至少避免传递任何用户输入到其中。

## STDIN

指定输入来自客户端应用。

## STDOUT

指定输出会去到客户端应用。

## boolean

指定选中的选项是应该被关闭还是打开。可以写TRUE、ON或1来启用选项，写FALSE、OFF或0禁用它。boolean值也可以被省略，那样会假定为TRUE。

## FORMAT

选择要读取或者写入的数据格式：text、csv（逗号分隔值）或者binary。默认是text。

## FREEZE

请求复制已经完成了行冻结的数据，就好像在运行VACUUM FREEZE命令之后复制。这是为了初始数据载入的性能而设计的。只有被载入表已经在当前子事务中被创建或截断、该事务中没有游标打开并且该事务没有持有更旧的快照时，行才会被冻结。目前无法在分区表上执行COPY FREEZE。

注意一旦成功地载入，所有其他会话将能立即看到该数据。这违背了普通的MVCC可见性规则，指定该选项的用户应该注意这可能会导致的潜在问题。

## DELIMITER

指定分隔文件每行中各列的字符。文本格式中默认是一个制表符，而CSV格式中默认是一个逗号。这必须是一个单一的单字节字符。使用binary格式时不允许这个选项。

## NULL

指定表示一个空值的字符串。文本格式中默认是\n（反斜线-N），CSV格式中默认是一个未加引用的空串。在你不想区分空值和空串的情况下，即使在文本格式中你也可能更喜欢空串。使用binary格式时不允许这个选项。

### Note

在使用COPY FROM时，任何匹配 这个串的数据项将被存储为空值，因此你应该确定你使用的是和 COPY TO时相同的串。

#### HEADER

指定文件包含标题行，其中有每一列的名称。在输出时，第一行包含 来自表的列名。在输入时，第一行会被忽略。只有使用 CSV格式时才允许这个选项。

#### QUOTE

指定一个数据值被引用时使用的引用字符。默认是双引号。 这必须是一个单字的单字节字符。只有使用 CSV格式时才允许这个选项。

#### ESCAPE

指定应该出现在一个匹配QUOTE值的数据字符之前 的字符。默认和QUOTE值一样（这样如果引用字符 出现在数据中，它会被双写）。这必须是一个单字的单字节字符。 只有使用CSV格式时才允许这个选项。

#### FORCE\_QUOTE

强制必须对每个指定列中的所有非NULL值使用引用。 NULL输出不会被引用。如果指定了\*， 所有列的非NULL值都将被引用。只有在 COPY TO中使用CSV格式时才允许 这个选项。

#### FORCE\_NOT\_NULL

不要把指定列的值与空值串匹配。在空值串就是空串的默认情况下， 这意味着空串将被读作长度为零的字符串而不是空值（即使它们没有 被引用）。只有在COPY FROM中使用 CSV格式时才允许这个选项。

#### FORCE\_NULL

将指定列的值与空值串匹配（即使它已经被加上引号），并且在找到 匹配时将该值设置为NULL。在空值串就是空串的默认 情况下，这会把一个被引用的空串转换为 NULL。 只有在COPY FROM中使用 CSV格式时才允许这个选项。

#### ENCODING

指定文件被以encoding\_name编码。如果省略 这个选项，将使用当前的客户端编码。详见下文的注解。

#### WHERE

WHERE子句是可选的，其一般形式是：

WHERE condition

其中condition是计算结果为boolean类型的任意表达式。任何不满足此条件的行都不会插入到表中。在用实际的行值替换任何变量引用时，如果该行返回true，则该行满足条件。

目前，在WHERE表达式中不允许使用子查询，并且值的计算不会看到COPY本身所做的任何更改（当表达式包含对VOLATILE函数的调用时，这一点很重要）。

## 输出

在成功完成时，一个COPY命令会返回一个形为

COPY count

的命令标签。count是被复制的行数。

### Note

如果命令不是COPY ... TO STDOUT或者等效的 ltsql元命令\copy ... to stdout, ltsql将只打印这个命令标签。这是为了防止弄混命令标签和刚刚打印的数据。

## 注解

COPY TO只能用于普通表，而不能用于视图，并且不能从子表或子分区复制行。例如，COPY table TO 复制与SELECT \* FROM ONLY table 相同的行。语法COPY (SELECT \* FROM table) TO ... 可用于转储一个继承层次结构、分区表或视图中的所有行。

COPY FROM可以被用于普通表、外部表、分区表或者具有INSTEAD OF INSERT触发器的视图。

你必须拥有被COPY TO读取的表上的选择特权，以及被COPY FROM插入的表上的插入特权。拥有在命令中列出的列上的特权就足够了。

如果对表启用了行级安全性，相关的SELECT策略将应用于COPY table TO语句。当前，有行级安全性的表不支持COPY FROM。不过可以使用等效的INSERT语句。

COPY命令中提到的文件会被服务器（而不是客户端应用）直接读取或写入。因此它们必须位于数据库服务器（不是客户端）的机器上或者是数据库服务器可以访问的。它们必须是 LightDB用户（运行服务器的用户 ID）可访问的并且是可读或者可写的。类似地，用PROGRAM 指定的命令也会由服务器（不是客户端应用）直接执行，它也必须是 LightDB用户可以执行的。只允许数据库超级用户或者授予了角色pg\_read\_server\_files、pg\_write\_server\_files及pg\_execute\_server\_program 之一的用户COPY一个文件或者命令，因为它允许读取或者写入服务器有特权访问的任何文件或者运行服务器有特权访问的程序。

不要把COPY和 ltsql指令 \copy 弄混。 \copy会调用 COPY FROM STDIN或者COPY TO STDOUT，然后读取/存储一个 ltsql客户端可访问的文件中的数据。因此，在使用\copy时，文件的可访问性和访问权利取决于客户端而不是服务器。

我们推荐在COPY中使用的文件名总是指定为一个绝对路径。在COPY TO的情况下服务器会强制这一点，但是对于 COPY FROM你可以选择从一个用相对路径指定的文件中读取。该路径将根据服务器进程（而不是客户端）的工作目录（通常是集簇的数据目录）解释。

用PROGRAM执行一个命令可能会受到操作系统的访问控制机制（如 SELinux）的限制。

COPY FROM将调用目标表上的任何触发器和检查约束。但是它不会调用规则。

对于标识列，COPY FROM命令将总是写上输入数据中提供的列值，这和INSERT的选项OVERRIDING SYSTEM VALUE的行为一样。

COPY输入和输出受到 DateStyle的影响。为了确保到其他可能使用非默认DateStyle设置的 LightDB安装的可移植性，在使用 COPY TO前应该把 DateStyle设置为ISO。避免转储把 IntervalStyle设置为 sql\_standard的数据也是一个好主意，因为负的区间值可能会被具有不同IntervalStyle设置的服务器解释错误。

即使数据会被服务器直接从一个文件读取或者写入一个文件而不通过客户端，输入数据也会被根据ENCODING选项或者当前客户端编码解释，并且输出数据会被根据ENCODING或者当前客户端编码进行编码。



COPY会在第一个错误处停止操作。这在 COPY TO的情况下不会导致问题，但是在COPY FROM中目标表将已经收到了一些行。这些行将不会变得可见或者可访问，但是它们仍然占据磁盘空间。如果在一次大型的复制操作中出现错误，这可能浪费相当可观的磁盘空间。你可能希望调用VACUUM来恢复被浪费的空间。

FORCE\_NULL和FORCE\_NOT\_NULL可以被同时用在同一列上。这会导致把已被引用的空值串转换为空值并且把未引用的空值串转换为空串。

## 文件格式

### 文本格式

在使用text格式时，读取或写入的是一个文本文件，其中每一行就是表中的一行。一行中的列被定界字符分隔。列值本身是由输出函数产生的或者是可被输入函数接受的属于每个属性数据类型的字符串。在为空值的列的位置使用指定的空值串。如果输入文件的任何行包含比预期更多或者更少的列，COPY FROM将会抛出一个错误。

数据的结束可以表示为一个只包含反斜线-点号（\。）的单一行。从一个文件读取时，数据结束标记并不是必要的，因为文件结束符就已经足够用了。只有使用 3.0 客户端协议之前的客户端应用复制数据时才需要它。

反斜线字符（\）可以被用在 COPY数据中来引用被用作行或者列定界符的字符。特别地，如果下列字符作为一个列值的一部分出现，它们必须被前置一个反斜线：反斜线本身、换行、回车以及当前的定界符字符。

COPY TO会不加任何反斜线返回指定的空值串。相反，COPY FROM会在移除反斜线之前把输入与空值串相匹配。因此，一个空值串（例如\n）不会与实际的数据值\n（它会被表示为\n）搞混。

COPY FROM识别下列特殊的反斜线序列：

序列	表示
\b	退格 (ASCII 8)
\f	换页 (ASCII 12)
\n	换行 (ASCII 10)
\r	回车 (ASCII 13)
\t	制表 (ASCII 9)
\v	纵向制表 (ASCII 11)
\digits	反斜线后跟一到三个八进制数字表示该数字代码对应的字节
\xdigits	反斜线加x后跟一到三个十六进制数字表示该数字代码对应的字节

当前，COPY TO不会发出一个八进制或十六进制位反斜线序列，但是它确实把上面列出的其他序列用于那些控制字符。

任何上述表格中没有提到的其他反斜线字符将被当作表示其本身。不过，要注意增加不必要的反斜线，因为那可能意外地产生一个匹配数据结束标记（\。）或者空值串（默认是\n）的字符串。这些字符串将在完成任何其他反斜线处理之前被识别。

强烈建议产生COPY数据的应用把数据换行和回车分别转换为\n和\r序列。当前可以把一个数据回车表示为一个反斜线和回车，把一个数据换行表示为一个反斜线和换行。不过，未来的发行可能不会接受这些表示。如果在不同的机器之间（例如从 Unix 到 Windows）传输COPY文件，它们也很容易受到破坏。

所有反斜杠序列都在编码转换后进行解释。用八进制和十六进制数字反斜杠序列指定的字节必须在数据库编码中形成有效字符。

COPY TO将用一个 Unix 风格的新行（“\n”）终止每一行。运行在 Microsoft Windows 上的服务器则会输出回车/新行（“\r\n”），不过只对 COPY到一个服务器文件这样做。为了做到跨平台一致，COPY TO STDOUT总是发送“\n”而不管服务器平台是什么。COPY FROM能够处理以 新行、回车或者回车/新行结尾的行。为了减少由作为数据的未加反斜线的新行 或者回车带来的风险，如果输出中的行结束并不完全相似，COPY FROM将会抱怨。

## CSV 格式

这种格式选项被用于导入和导出很多其他程序（例如电子表格）使用的逗号 分隔值（CSV）文件格式。不同于 LightDB标准文本格式使用的转义 规则，它产生并且识别一般的 CSV 转义机制。

每个记录中的值用DELIMITER字符分隔。如果值包含 定界符字符、QUOTE字符、NULL字符串、 一个回车或者换行字符，那么整个值会被加上QUOTE字符 作为前缀或者后缀，并且在该值内QUOTE字符或者 ESCAPE字符的任何一次出现之前放上转义字符。在输出 指定列中非NULL值时，还可以使用 FORCE\_QUOTE来强制加上引用。

CSV格式没有标准方式来区分NULL值和空字符串。 LightDB的COPY用引用来处理 这种区分工作。NULL被按照NULL参数字符串输出 并且不会被引用，而匹配NULL参数字符串的非NULL 值会被加上引用。例如，使用默认设置时，NULL被写作一个未 被引用的空字符串，而一个空字符串数据值会被写成带双引号（"”）。 值的读取遵循类似的规则。你可以用FORCE\_NOT\_NULL来防止 对指定列的NULL输入比较。你还可以使用 FORCE\_NULL把带引用的空值字符串数据值转换成NULL。

因为反斜线在CSV格式中不是一种特殊字符，数据结束标记 \. 也可以作为一个数据值出现。为了避免任何解释误会，在 一行上作为孤项出现的\. 数据值输出时会自动被引用，并且 输入时如果被引用，则不会被解释为数据结束标记。如果正在载入一个由 另一个应用创建的文件并且其中具有一个未被引用的列且可能具有 \. 值，你可能需要在输入文件中引用该值。

### Note

CSV格式中，所有字符都是有意义的。一个被空白或者其他 非 DELIMITER字符围绕的引用值将包括那些字符。在导入 来自用空白填充CSV行到固定长度的系统的数据时，这可能会 会导致错误。如果出现这种情况，在导入数据到 LightDB之前，你可能需要预处理该 CSV文件以移除拖尾的空白。

### Note

CSV 格式将识别并且产生带有包含嵌入的回车和换行的引用值的 CSV 文件。因此文件并不限于文本格式文件的每个表行一行的形式。

### Note

很多程序会产生奇怪的甚至偶尔是不合常理的 CSV 文件，因此该文件 格式更像是一种习惯而不是标准。因此你可能会碰到一些无法使用这种 机制导入的文件，并且COPY也可能产生其他程序无 法处理的文件。

## 二进制格式

binary格式选项导致所有数据被以二进制格式 而不是文本格式存储/读取。它比文本和CSV格式要 快一些，但是二进制格式文件在不同的机器架构和 LightDB版本之间的可移 植

性要差些。还有，二进制格式与数据格式非常相关。例如不能从一个 `smallint` 列中输出二进制数据并且把它读入到一个 `integer` 列中，虽然这样做在文本格式中是可行的。

binary 文件格式由一个文件头、零个或者更多个包含一行数据的元组以及一个文件尾构成。头部和数据都以网络字节序表示。

## 文件头

文件头由 15 字节的固定域构成，后面跟着一个变长的头部扩展区。固定域有：

### 签名

11-字节的序列 `PGCOPY\n\377\r\n\0` — 注意 零字节是签名的一个必要的部分（该签名是为了能容易地发现文件被无法正确处理 8 位字符编码的传输所破坏。这个签名将被行尾翻译过滤器、删除零字节、删除高位或者奇偶修改等改变）。

### 标志域

32-位整数位掩码，用以表示该文件格式的重要方面。位被编号为从 0（LSB）到 31（MSB）。注意这个域以网络字节序存放（最高有效位在前），所有该文件格式中使用的整数域都是这样。16-31 位被保留用来表示严重的文件格式问题，读取者如果在这个范围内发现预期之外的被设置位，它应该中止。0-15 位被保留用来表示向后兼容的格式问题，读取者应该简单地略过这个范围内任何预期之外的被设置位。当前只定义了一个标志位，其他位必须为零：

#### 位 16

如果为 1，表示数据中包含 OID；如果为 0，则不包含。LightDB 不再支持 oid 系统列，但是格式仍然包含该指示符。

### 头部扩展区长度

32-为整数，表示头部剩余部分的以字节计的长度，不包括其本身。当前，这个长度为零，并且其后就紧跟着第一个元组。未来对该格式的更改可能会允许在头部中表示额外的数据。如果读取者不知道要对头部扩展区数据做什么，可以安静地跳过它。

头部扩展区域被预期包含一个能自我解释的块的序列。该标志域并不想告诉读取者扩展数据是什么。详细的头部扩展内容的设计留给后来的发行去做。

这种设计允许向后兼容的头部增加（增加头部扩展块或者设置低位标志位）以及非向后兼容的更改（设置高位标志位来表示这类更改并且在需要时向扩展区域中增加支持数据）。

## 元组

每一个元组由一个表示元组中域数量的 16 位整数计数开始（当前，一个表中的所有元组都应该具有相同的计数，但是这可能不会总是为真）。然后是元组中的每一个域，它是一个 32 位的长度字，后面则跟随着这么多个字节的域数据（长度字不包括其本身，并且可以是零）。作为一种特殊情况，-1 表示一个 NULL 域值。在 NULL 情况下，后面不会跟随值字节。

在域之间没有对齐填充或者任何其他额外的数据。

当前，一个二进制格式文件中的所有数据值都被假设为二进制格式（格式代码一）。可以预见未来的扩展可能会增加一个允许独立指定各列的格式代码的头部域。

要为实际的元组数据决定合适的二进制格式，你应该参考 LightDB 源码，特别是用于各列数据类型的 `*send` 和 `*recv` 函数（通常可以在源码的 `src/backend/utils/adt/` 目录中找到这些函数）。

如果文件中包含 OID，OID 域会紧跟在域计数之后。它是一个普通域，不过它没有被包含在域计数中。注意 LightDB 当前版本不支持 oid 系统列。

## 文件尾

文件尾由一个包含 -1 的 16 位整数字组成。这很容易与一个元组的域计数数字区分开。

如果一个域计数数字不是 -1 也不是期望的列数，读取者应该报告错误。这提供了一种针对某种数据不同步的额外检查。

## 示例

下面的例子使用竖线 (|) 作为域定界符把一个表复制到客户端：

```
COPY country TO STDOUT (DELIMITER '|');
```

从一个文件中复制数据到country表中：

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

只把名称以 'A' 开头的国家复制到一个文件中：

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/a_list_countries.copy';
```

要复制到一个压缩文件中，你可以用管道把输出导到一个外部压缩程序：

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

这里是一个适合于从STDIN复制到表中的数据：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

注意每一行上的空白实际是一个制表符。

下面是用二进制格式输出的相同数据。该数据是用 Unix 工具 `od -c` 过滤后显示的。该表具有三列，第一列类型是 `char(2)`，第二列类型是 `text`，第三列类型是 `integer`。所有行在第三列都是空值。

```
0000000 P G C O P Y \n 377 \r \n \0 \0 \0 \0 \0 \0
0000020 \0 \0 \0 \0 003 \0 \0 \0 002 A F \0 \0 \0 013 A
0000040 F G H A N I S T A N 377 377 377 377 \0 003
0000060 \0 \0 \0 002 A L \0 \0 \0 007 A L B A N I
0000100 A 377 377 377 377 \0 003 \0 \0 \0 002 D Z \0 \0 \0
0000120 007 A L G E R I A 377 377 377 377 \0 003 \0 \0
0000140 \0 002 Z M \0 \0 \0 006 Z A M B I A 377 377
0000160 377 377 \0 003 \0 \0 \0 002 Z W \0 \0 \0 \b Z I
0000200 M B A B W E 377 377 377 377 377 377
```

## 兼容性

SQL 标准中没有 COPY 语句。

下列语法仍然被支持:

```

COPY table_name [ ( column_name [, ...] ) ]
  FROM { 'filename' | STDIN }
  [ [ WITH ]
    [ BINARY ]
    [ DELIMITER [ AS ] 'delimiter_character' ]
    [ NULL [ AS ] 'null_string' ]
    [ CSV [ HEADER ]
      [ QUOTE [ AS ] 'quote_character' ]
      [ ESCAPE [ AS ] 'escape_character' ]
      [ FORCE NOT NULL column_name [, ...] ] ] ] ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
  TO { 'filename' | STDOUT }
  [ [ WITH ]
    [ BINARY ]
    [ DELIMITER [ AS ] 'delimiter_character' ]
    [ NULL [ AS ] 'null_string' ]
    [ CSV [ HEADER ]
      [ QUOTE [ AS ] 'quote_character' ]
      [ ESCAPE [ AS ] 'escape_character' ]
      [ FORCE QUOTE { column_name [, ...] | * } ] ] ] ]

```

注意在这种语法中, BINARY和CSV被视作独立的关键词, 而不是FORMAT选项的参数。

下列语法仍然被支持:

```

COPY [ BINARY ] table_name
  FROM { 'filename' | STDIN }
  [ [USING] DELIMITERS 'delimiter_character' ]
  [ WITH NULL AS 'null_string' ]

COPY [ BINARY ] table_name
  TO { 'filename' | STDOUT }
  [ [USING] DELIMITERS 'delimiter_character' ]
  [ WITH NULL AS 'null_string' ]

```

---

# CREATE ACCESS METHOD

CREATE ACCESS METHOD — 定义一种新的访问方法

## Synopsis

```
CREATE ACCESS METHOD name
    TYPE access_method_type
    HANDLER handler_function
```

## 简介

CREATE ACCESS METHOD创建一种新的访问方法。

访问方法名称在数据库中必须唯一。

只有超级用户可以定义新的访问方法。

## 参数

name

要创建的访问方法的名称。

access\_method\_type

这个子句指定要定义的访问方法的类型。当前只支持TABLE和INDEX。

handler\_function

handler\_function是一个之前已注册的函数的名称（可能被模式限定），该函数表示要创建的访问方法。处理器函数必须被声明为接收一个单一的internal类型的参数，并且它的返回类型取决于访问方法的类型；对于TABLE访问方法，它必须是table\_am\_handler，而对于INDEX访问方法，它必须是index\_am\_handler。处理器函数必须实现的 C 级别 API 取决于访问方法的类型。Chapter 5中描述了表访问方法的 API，而Chapter 58中描述了索引访问方法的 API。

## 示例

用处理器函数heptree\_handler创建一种索引访问方法heptree:

```
CREATE ACCESS METHOD heptree TYPE INDEX HANDLER heptree_handler;
```

## 兼容性

CREATE ACCESS METHOD是一种LightDB扩展。

## 另见

DROP ACCESS METHOD, CREATE OPERATOR CLASS, CREATE OPERATOR FAMILY

---

# CREATE AGGREGATE

CREATE AGGREGATE — 定义一个新的聚集函数

## Synopsis

```
CREATE [ OR REPLACE ] AGGREGATE name ( [ argmode ] [ argname ] arg_data_type
[ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mfunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)
```

```
CREATE [ OR REPLACE ] AGGREGATE name ( [ [ argmode ] [ argname ] arg_data_type
[ , ... ] ]
                                ORDER BY [ argmode ] [ argname ] arg_data_type
[ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , INITCOND = initial_condition ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
    [ , HYPOTHETICAL ]
)
```

或者旧的语法

```
CREATE [ OR REPLACE ] AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
```

```

[ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
[ , COMBINEFUNC = combinefunc ]
[ , SERIALFUNC = serialfunc ]
[ , DESERIALFUNC = deserialfunc ]
[ , INITCOND = initial_condition ]
[ , MSFUNC = msfunc ]
[ , MINVFUNC = minvfunc ]
[ , MSTYPE = mstate_data_type ]
[ , MSSPACE = mstate_data_size ]
[ , MFINALFUNC = mfunc ]
[ , MFINALFUNC_EXTRA ]
[ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
[ , MINITCOND = minitial_condition ]
[ , SORTOP = sort_operator ]
)

```

## 描述

CREATE AGGREGATE定义一个新的聚集函数。CREATE OR REPLACE AGGREGATE将定义新的聚合函数或替换现有定义。在发布中已经包括了一些基本的和常用的聚集函数；它们的文档请见Section 9.21。如果要定义一个新类型或者需要一个还没有被提供的聚集函数，那么CREATE AGGREGATE就可以被用来提供想要的特性。

在替换现有定义时，参数类型、结果类型和直接参数的数量可能不会更改。此外，新定义的类型（普通聚合、有序集聚合或假设集聚合）必须与旧定义相同。

如果给定了一个模式名（例如CREATE AGGREGATE myschema.myagg ...），那么该聚集会被创建在指定的模式中。否则它会被创建在当前模式中。

一个聚集函数需要用它的名称和输入数据类型标识。同一个模式中的两个聚集可以具有相同的名称，只要它们在不同的输入类型上操作即可。一个聚集的名称和输入数据类型必须与同一模式中的每一个普通函数区分开。这种行为与普通函数名的重载完全一样（见CREATE FUNCTION）。

一个简单的聚集函数由一个或者多个普通函数组成：一个状态转移函数 sfunc和一个可选的最终计算函数 ffunc。它们被这样使用：

```

sfunc( internal-state, next-data-values ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value

```

LightDB创建一个数据类型 stype的临时变量来保持聚集的当前内部状态。对每一个输入行，聚集参数值会被计算并且状态转移函数会被调用，它用当前状态值和新参数值计算一个新的内部状态值。等所有行都被处理完后，最终函数会被调用一次来计算该聚集的返回值。如果没有最终函数，则最终的状态值会被返回。

一个聚集函数可以提供一个初始条件，即一个用于内部状态值的初始值。它被作为一个类型text的值指定并且存储在数据库中，但是它必须是状态值数据类型的一个常量的合法外部表示。如果没有提供，则状态值从空值开始。

如果状态转移函数被声明为“strict”，那么不能用空值输入来调用它。如果有这种转移函数，聚集将按照下面的行为执行。带有任何空值的行会被忽略（函数不被调用并且之前的状态值被保持）。如果初始状态值就是空值，那么碰到第一个没有空值的行时，状态值会被替换成第一个参数值，并且对于每一个后续的空值的行都会调用该转移函数。这对实现max这样的聚集很方便。注意只有当 state\_data\_type 和第一个 arg\_data\_type相同时，这种行为才可用。当这些类型不同时，你必须提供一个非空初始条件或者使用一个非严格转移函数。

如果状态转移函数不是严格的，那么在碰到每个输入行时都将会调用它，并且它必须自行处理空值输入和空状态值。这允许聚集的作者完全控制该聚集如何处理空值。



如果最终函数被声明为“strict”，那么当最终状态值为空时将不会调用它，而是自动地返回一个空结果（当然，这只是严格函数的普通行为）。在任何情况下最终函数都可以返回一个空值。例如，avg的最终函数会在看到零个输入行时返回空。

有时候把最终函数声明成不仅采用状态值还采用对应于聚集输入值的额外参数是有用的。这样做的主要原因是，如果最终函数是多态的，那么状态值的数据类型将不适合于用来确定结果类型。这些额外的参数总是以 NULL 形式传递（因此使用FINALFUNC\_EXTRA选项时，最终函数不能是严格的），但尽管如此它们都是合法参数。例如，最终函数可以利用get\_fn\_expr\_argtype来标识当前调用中的实际参数类型。

如Section 36.12.中所述，一个聚集可以选择支持moving-aggregate mode。这要求指定MSFUNC、MINVFUNC以及MSTYPE参数，并且参数MSSPACE、MFINALFUNC、MFINALFUNC\_EXTRA和MINITCOND是可选的。除了MINVFUNC，这些参数的工作都和对应的不带M的简单聚集参数相似，它们定义了包括一个逆向转移函数的聚集的一种独立实现。

在参数列表中含有ORDER BY的语法会创建一种被称为有序集聚集的特殊聚集类型。如果指定了HYPOTHETICAL，则会创建一个假想集聚集。这些聚集以依赖排序的方法在排好序的值上操作，因此指定一个输入排序顺序是调用过程的重要一环。还有，它们可以有直接参数，这类参数只对每次聚集计算一次，而不是对每一个输入行计算一次。假想集聚集是有序集聚集的一个子类，其中一些直接参数要求在数量和类型上都匹配被聚集的参数列。这允许这些直接参数的值被当做一个附加的“假想”行被加入到聚集输入行的集合中。

如Section 36.12.4中所示，一个聚集可以支持部分聚集。这要求指定COMBINEFUNC参数。如果state\_data\_type为internal，通常也可以提供SERIALFUNC和DESERIALFUNC参数，这样可以让并行聚集成为可能。注意，该聚集还必须被标记为PARALLEL SAFE以启用并行聚集。

行为与MIN或MAX相似的聚集有时可以通过直接查看一个索引而不是扫描每一个输入行来优化。如果这个聚集可以被这样优化，请通过指定一个排序操作符来指出。基本要求是，该聚集必须得出由该操作符产生的排序顺序中的第一个元素，换句话说：

```
SELECT agg(col) FROM tab;
```

必须等价于：

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

进一步的假定是该聚集忽略空输入，并且当且仅当没有非空输入时它才会返回一个空结果。通常，一种数据类型的<操作符是 MIN的合适的排序操作符，而>是 MAX的合适的排序操作符。注意，除非指定的操作符是一个 B-树索引操作符类的“小于”或者“大于”策略成员，优化将永远不会产生实际效果。

要能够创建一个聚集函数，你必须具有参数类型、状态类型和返回类型上的 USAGE特权，还有在支持函数上的 EXECUTE特权。

## 参数

name

要创建的聚集函数的名称（可以是模式限定的）。

argmode

一个参数的模式：IN或者VARIADIC（聚集函数不支持OUT参数）。如果忽略，默认值是IN。只有最后一个参数能被标记为VARIADIC。

argname

一个参数的名称。当前这只用于文档的目的。如果被忽略，该参数就没有名称。

`arg_data_type`

这个聚集函数操作的一个输入数据类型。要创建一个零参数的聚集函数，可以写一个\*来替代参数说明的列表（这类聚集的一个例子是 `count(*)`）。

`base_type`

在CREATE AGGREGATE的旧语法中，输入数据类型是由一个`basetype`参数指定而不是写在聚集名之后。注意这种语法只允许一个输入参数。要用这种语法定义一个零参数的聚集函数，把`basetype`指定为“ANY”（不是\*）。有序集聚集不能用旧语法定义。

`sfunc`

要为每一个输入行调用的状态转移函数名。对于一个正常的N-参数的聚集函数，`sfunc`必须接收N+1个参数，第一个参数的类型是`state_data_type`而其余的参数匹配该聚集被声明的输入数据类型。该函数必须返回一个类型为`state_data_type`的值。这个函数会采用当前的状态值以及当前的输入数据值，并且返回下一个状态值。

对于有序集（包括假想集）聚集，状态转移函数只接收当前的状态值和聚集参数，但无需直接参数。否则它就和其他转移函数一样了。

`state_data_type`

聚集的状态值的数据类型。

`state_data_size`

聚集的状态值的近似平均尺寸（以字节计）。如果这个参数被忽略或者为零，将使用一个基于`state_data_type`的默认估计值。规划器使用这个值来估计一个分组聚集查询所需的内存。

`ffunc`

最终函数的名称，该函数在所有输入行都被遍历之后被调用来计算聚集的结果。对于一个常规聚集，这个函数必须只接受一个类型为`state_data_type`的单一参数。该聚集的返回数据类型被定义为这个函数的返回类型。如果没有指定`ffunc`，则结束状态值被用作聚集的结果，并且返回类型为`state_data_type`。

对于有序集（包括假想集）聚集，最终函数不仅接收最终状态值，还会接收所有直接参数的值。

如果指定了`FINALFUNC_EXTRA`，则除了最终状态值和任何直接参数之外，最终函数还接收额外的对应于该聚集的常规（聚集）参数的NULL值。这主要用于在定义了一个多态聚集时允许正确地决定聚集的结果类型。

`FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }`

此选项指定最终函数是否为不会修改参数的纯函数。`READ_ONLY`表示它不会修改；其他两个值表示它可能会更改迁移状态值。请参见Notes以获取更多详细信息。除了有序集合的聚合使用默认值`READ_WRITE`，其他默认值均为`READ_ONLY`。

`combinefunc`

`combinefunc`函数可以被有选择地指定以允许聚集函数支持部分聚集。如果提供这个函数，`combinefunc`必须组合两个`state_data_type`值，每一个都包含在输入值某个子集上的聚集结果，它会产生一个新的`state_data_type`来表示在两个输入集上的聚集结果。这个函数可以被看做是一个`sfunc`，和后者在一个个体输入行上操作并且把它加到运行聚集状态上不同，这个函数是把另一个聚集状态加到运行状态上。

`combinefunc`必须被声明为有两个`state_data_type`参数并且返回一个`state_data_type`值。这个函数可以有选择性地被标记为“strict”。在被标记的情况下，当任何一个输入状态为空时，将不会调用该函数，而是把另一个状态当作正确的结果。

对于state\_data\_type为 internal的聚集函数，combinefunc不能为 strict。这种情况下，combinefunc必须确保 正确地处理空状态并且被返回的状态能被恰当地存储在聚集内存上下文中。

#### serialfunc

state\_data\_type为 internal的一个聚集函数可以参与到并行聚集中，当且仅当它具有一个 serialfunc函数，该函数 必须把聚集状态序列化成一个bytea值以传递给另一个进程。这个函数 必须有一个单一的internal类型参数并且返回类型bytea。 相应地也需要一个deserialfunc。

#### deserialfunc

把一个之前序列化后的聚集状态反序列化为 state\_data\_type。这个函数 必须有两个类型分别为bytea和internal的参数，并且产生 一个类型internal的结果（注意：第二个类型为internal的 参数是无用的，但是为了类型安全的原因还是要求有该参数）。

#### initial\_condition

状态值的初始设置。这必须是以数据类型state\_data\_type能够接受的形式出现 的一个字符串常量。如果没有指定，状态值会从空值开始。

#### msfunc

前向状态转移函数的名称，在移动聚集模式中会为每个输入行调用这个函数。它 非常像常规的转移函数，不过它的第一个参数和结果类型是 mstate\_data\_type，这可能与state\_data\_type不同。

#### minvfunc

在移动聚集模式中用到的逆向状态转移函数的名称。这个函数与 msfunc具有相同的参数和结果类型，但是它被用于从当前聚集 状态中移除一个值，而不是向其中增加一个值。逆向转移函数必须具有和前向状态 转移函数相同的严格性属性。

#### mstate\_data\_type

使用移动聚集模式时，用于聚集状态值的数据类型。

#### mstate\_data\_size

使用移动聚集模式时，聚集状态值的近似平均尺寸（以字节计）。它的作用和state\_data\_size相同。

#### mffunc

使用移动聚集模式时用到的最终函数名称，当所有输入行都被遍历后会调用它来 计算聚集的结果。它的工作和ffunc一样，但是它的第一个参 数类型是mstate\_data\_type并且额外的空参数要通过书写 MFINALFUNC\_EXTRA来指定。mffunc 或者mstate\_data\_type决定的聚集结果类型必须匹配由聚集 的常规实现所确定的类型。

MFINALFUNC\_MODIFY = { READ\_ONLY | SHAREABLE | READ\_WRITE }

此选项类似于FINALFUNC\_MODIFY，只是它描述了移动聚集最终函数的行为。

#### minitial\_condition

使用移动聚集模式时，状态值的初始设置。它的作用和 initial\_condition一样。

#### sort\_operator

一个MIN- 类或者MAX-类聚集的相关 排序操作符。这只是一个操作符名称（可能被模式限定）。这个操作符被 假定为具有和该聚集（必须是一个单一参数的常规聚集）相同的输入数据 类型。

PARALLEL = { SAFE | RESTRICTED | UNSAFE }

PARALLEL SAFE、PARALLEL RESTRICTED和PARALLEL UNSAFE的含义和 CREATE FUNCTION中的相同。如果一个聚集被标记为 PARALLEL UNSAFE（默认）或者 PARALLEL RESTRICTED，将不会考虑将它并行化。注意 规划器不会参考聚集的支持函数的并行安全性标记，它只会考虑聚集本身 的这类标记。

HYPOTHETICAL

只用于有序集聚集，这个标志指定聚集参数会根据假想集聚集的要求进行处理：即后面的直接参数必须匹配聚集（WITHIN GROUP）参数的数据 类型。HYPOTHETICAL标志在运行时没有任何效果，它只在 命令解析期间对确定数据类型和聚集参数的排序规则有用。

CREATE AGGREGATE的参数可以用任意顺序书写， 而无需遵照以上说明的顺序。

## 注解

在指定支持函数名的参数中，如果需要你可以写一个模式名，例如 SFUNC = public.sum。在这里不能写参数类型 — 支持函数 的参数类型是根据其他参数决定的。

通常，LightDB函数是不要修改输入值的真函数。然而，一个聚合迁移函数在聚合上下文中使用时被允许诈欺并修改已在迁移状态的参数。因为与每次创建一个迁移状态的新的拷贝相比，这样可以提供实质的性能提升。

同样，虽然人们一般不期望聚合最终函数修改它的输入值，但有时回避修改迁移态参数是不切实际的。这种行为必须使用FINALFUNC\_MODIFY参数声明。READ\_WRITE值表示最终函数以某种未指定的方式修改了迁移状态值。这个值防止将聚合用作窗口函数，并且还可以防止因共用相同的输入值和迁移函数的聚合调用而合并迁移状态。SHAREABLE值表示过渡函数不能在最终功能之后使用用，但多重最终函数调用可以对最终的迁移状态值执行调用。这个值阻止将聚合用作窗口函数，但允许合并过渡状态。（也就是说，此处所关注的优化不是重复地使用相同的最终函数，而是把不同的最终函数应用到相同的最终迁移状态值。只要所有最终功能都没有标记为READ\_WRITE就被允许。）

如果一个聚集支持移动聚集模式，当该聚集被用于一个具有移动帧起点（即帧起点 模式不是UNBOUNDED PRECEDING）的窗口函数时，它将提升计 算效率。在概念上，当从底部进入窗口帧时前向转移函数会把输入值加到聚集的状 态上，而逆向转移函数会在从顶部离开帧时再次移除输入值。因此，当值被移除时， 它们总是按照被加入的相同顺序被移除。无论何时调用逆向转移函数，它都将因此 接收最近增加但是还未被移除的参数值。逆向转移函数可以假定在它移除最旧的行 之后至少有一行保留在当前状态中（当情况不是这样时，窗口函数机制会简单地开 始一次新的聚集，而不是使用逆向转移函数）。

用于移动聚集模式的前向转移函数不允许返回 NULL 作为新的状态值。如果逆向 转移函数返回 NULL，这表明逆向函数无法为这个特定的输入逆转状态计算，并且 该聚集计算因此必须从当前帧的开始位置“从零开始”重新被计算。在一些少见的情 况中，逆转正在计算中的状态值是不切实际的，这种习惯可以允许在此类情形中使用 移动聚集模式。

如果没有提供移动聚集实现，聚集仍然可以被用于移动帧，但是 只要帧起点移动，LightDB都将会重新计算 整个聚集。注意不管聚集有没有支持移动聚集模式， LightDB都能处理一个移动帧结束而无需重 新计算，这可以通过增加新值到聚集状态完成。这就是为什么使用聚合作为窗口函数需要最终函数只读的原因。人们认为最终函数不能破坏聚集的状 态值，这样即使已经为一组帧边界得到了一个聚集结果值，该聚集也能继续下去。

有序集聚集的语法允许为最后一个直接参数以及最后一个聚集（ WITHIN GROUP）参数指定VARIADIC。但是，当前的 实现限制只能以两种方式使用VARIADIC。第一种，有续集聚集只能 使用VARIADIC "any"，不能使用其他可变数组类型。第二种，如果最 后一个直接参数是VARIADIC "any"，那么只能有一个聚集参数并且它 也必须是VARIADIC "any"（在系统目录中使用的表示中，这两个参数 会被合并为一个单一的VARIADIC "any"项，因为 pg\_proc无法表示具有超过一个VARIADIC参数的 函数）。如果该聚集是一个假想集聚集，匹配VARIADIC "any"参数的 直接参数就是假想参数。任何在前面的参数表示额外的直接参数，它们不被约束为需 要匹配聚集参数。

当前，有序集聚集无须支持移动聚集模式，因为它们不能被用作窗口函数。

部分（包括并行）聚集当前不被有续集聚集支持。还有，包括 DISTINCT或者ORDER BY子句的聚集调用将不会使用 部分聚集，因为在部分聚集时无法支持那些语义。

## 示例

见Section 36.12

## 兼容性

CREATE AGGREGATE是一种 LightDB的语言扩展。SQL 标准没有提供 用户定义的聚集函数。

## 另见

ALTER AGGREGATE, DROP AGGREGATE

---

# CREATE CAST

CREATE CAST — 定义一种新的造型

## Synopsis

```
CREATE CAST (source_type AS target_type)
  WITH FUNCTION function_name [ (argument_type [, ...]) ]
  [ AS ASSIGNMENT | AS IMPLICIT ]
```

```
CREATE CAST (source_type AS target_type)
  WITHOUT FUNCTION
  [ AS ASSIGNMENT | AS IMPLICIT ]
```

```
CREATE CAST (source_type AS target_type)
  WITH INOUT
  [ AS ASSIGNMENT | AS IMPLICIT ]
```

## 描述

CREATE CAST定义一种新的造型。一种造型指定如何在两种数据类型之间执行转换。例如，

```
SELECT CAST(42 AS float8);
```

通过调用一个之前指定的函数（这种情况中是 `float8(int4)`）把整型常量 `42` 转换成类型 `float8`（如果没有定义合适的造型，该转换会失败）。

两种类型可以是二进制可强制，这表示该转换可以被“免费”执行而不用调用任何函数。这要求相应的值使用同样的内部表示。例如，类型 `text` 和 `varchar` 在双向都是二进制可强制的。二进制可强制性并不必是一种对称关系。例如，在当前实现中从 `xml` 到 `text` 的造型可以被免费执行，但是反向则需要一个函数来执行至少一次语法检查（两种在双向都二进制值兼容的类型也被称作二进制兼容）。

通过使用 `WITH INOUT` 语法，你可以把一种造型定义成 I/O 转换造型。一种 I/O 转换造型执行时，会调用源数据类型的输出函数，并且把结果字符串传递给目标数据类型的输入函数。在很多常见情况中，这种特性避免了为转换单独定义一个造型函数。一种 I/O 转换造型表现得和一个常规的基于函数的造型相同，只是实现不同而已。

默认情况下，只有一次显式造型请求才会调用造型，形式是 `CAST(x AS typename) or x::typename`。

如果造型被标记为 `AS ASSIGNMENT`，那么在为一个目标数据类型类型的列赋值时会隐式地调用它。例如，假设 `foo.f1` 是一个类型 `text` 的列，那么如果从类型 `integer` 到类型 `text` 的造型被标记为 `AS ASSIGNMENT`，则：

```
INSERT INTO foo (f1) VALUES (42);
```

将被允许，否则不会允许（我们通常使用赋值造型来描述此类造型）。

如果造型被标记为 `AS IMPLICIT`，那么可以在任何上下文中隐式地调用它，无论是赋值还是在一个表达式内部（我们通常用术语 隐式造型来描述这类造型）。例如，考虑这个查询：

```
SELECT 2 + 4.0;
```

解析器初始会把常量分别标记为类型 `integer` 和 `numeric`。在系统目录中没有 `integer + numeric` 操作符，但是有一个 `numeric + numeric` 操作符。因此，如果有一种可用的从 `integer` 到 `numeric` 的造型且被标记为 `AS IMPLICIT` — 实际上确实有 — 该查询将会成功。解析器将应用该隐式造型 并且解决该查询，就好像它被写成：

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

现在，系统目录也提供一种从 `numeric` 到 `integer` 的造型。如果这种造型被标记为 `AS IMPLICIT` — 实际上并没有 — 那么解析器将面临选择：是用前面介绍的过程，还是把 `numeric` 常量造型成 `integer` 并且应用 `integer + integer` 操作符。由于缺少哪种选择更好的知识，解析器会放弃并且说明查询有歧义。我们能告诉解析器把一个混合了 `numeric` 和 `integer` 的表达式解析成 `numeric` 更好的方法就是只让这两种造型中的一个为隐式的，没有有关于此的内建知识。

对标记造型为隐式持保守态度是明智的。过多的隐式造型路径可能导致 `LightDB` 以令人吃惊的方式解释命令，或者由于有多种可能解释而根本无法解析命令。一种好的经验是让一种造型只对于同一种一般类型分类中的类型间的信息保持转换隐式可调用。例如，从 `int2` 到 `int4` 的造型可以被合理地标记为隐式，但是从 `float8` 到 `int4` 的造型可能应该只能在赋值时使用。跨类型分类的造型（如 `text` 到 `int4`）最好只被用于显式使用。

### Note

有时为了可用性或者标准兼容的原因，有必要提供在一个类型集合之间的多种隐式造型，这会导致上述不可避免的歧义。解析器还有一招基于类型分类和优先类型的后手，它能帮助提供这类情况下预期的行为。详见 `CREATE TYPE`。

要创建一种造型，你必须拥有源数据类型和目标数据类型并且具有在其他类型上的 `USAGE` 特权。要创建一种二进制可强制造型，你必须是一个超级用户（这种限制是因为错误的二进制可强制造型转换很容易让服务器崩溃）。

## 参数

`source_type`

该造型的源数据类型的名称。

`target_type`

该造型的目标数据类型的名称。

`function_name`[(`argument_type` [, ...])]

被用于执行该造型的函数。函数名称可以用模式限定。如果没有被限定，将在模式搜索路径中查找该函数。函数的结果数据类型必须是该造型的目标数据类型。它的参数讨论如下。如果没有指定参数列表，则该函数名称在其模式中必须是唯一的。

`WITHOUT FUNCTION`

指示源类型可以二进制强制到目标类型，因此执行该造型不需要函数。

`WITH INOUT`

指示该造型是一种 I/O 转换造型，执行需要调用源数据类型的输出函数，并且把结果字符串传递给目标数据类型的输入函数。

## AS ASSIGNMENT

指示该造型可以在赋值的情况下被隐式调用。

## AS IMPLICIT

指示该造型可以在任何上下文中被隐式调用。

造型实现函数可以具有 1 到 3 个参数。第一个参数类型必须等于源类型或者能从源类型二进制强制得到。第二个参数（如果存在）必须是类型 `integer`，它接收与目标类型相关联的类型修饰符，如果没有类型修饰符，它会收到 -1。第三个参数（如果存在）必须是类型 `boolean`，如果该造型是一种显式造型，它会收到 `true`，否则会收到 `false`（奇怪地是，SQL 标准在某些情况中对显式和隐式造型要求不同的行为。这个参数被提供给必须实现这类造型的函数。不推荐在设计自己的数据类型时用它）。

一个造型函数的返回类型必须等于目标类型或者能二进制强制到目标类型。

通常，强制转换必须具有不同的源和目标数据类型。但是，如果它有一个带有多个参数的强制转换实现函数，则可以声明具有相同源类型和目标类型的造型。它用于表示系统目录中特定类型的长度强制函数。命名函数用于将类型的值强制转为其第二个参数提供的类型修饰符的值。

当强制转换具有不同的源类型和目标类型，并且一个函数使用多个参数时，它支持从一种类型转换为另一种类型，并在单个步骤中应用长度强制。如果没有这样的条目，强制转换为使用类型修饰符的类型将涉及两个强制转换步骤，一个是在数据类型之间进行转换，另一个是应用修饰符。

向域类型强制转换或从域类型强制转换当前无效。向域或从域强制转换使用与其基础类型关联的造型。

## 注解

使用 `DROP CAST` 移除用户定义的造型。

记住如果你想要能够双向转换类型，你需要在两个方向上都显式声明造型。

通常没有必要创建用户定义类型和标准字符串类型（`text`、`varchar`和`char(n)`，以及被定义在字符串分类中的用户定义类型）之间的造型。LightDB会为它们提供自动的 I/O 转换造型。到字符串类型的自动造型被当做赋值造型，而字符串类型作为源的自动造型只能是显式的。通过声明你自己的造型来替换自动造型可以覆盖这种行为，但是这样做的唯一原因是你想让该转换比标准的设置更容易被调用。另一种可能的原因是你想让该转换的行为与该类型的 I/O 函数不同，但这种原因足够令人感到意外，你应该考虑再三它是不是个好主意（确实有少量内建类型对转换具有不同的行为，绝大部分是因为 SQL 标准的要求）。

虽然不必要，推荐你继续遵循这种在目标数据类型后面命名造型实现函数的习惯。很多用户习惯于能够使用一种函数风格的记法来造型数据类型，即 `typename(x)`。这种记法正好是对造型实现函数的调用，这里它没有被作为造型特殊对待。如果你的转换函数没有被指定支持这种习惯，那么你的用户会觉得意外。由于LightDB允许用不同的参数类型重载同一个函数名，因此存在多个从不同类型到同一目标类型的同名转换函数并不困难。

### Note

实际上前一段过于简化了：有两种情况中一个函数调用结构在没有被匹配到一个实际函数时将被当作一次造型请求。如果函数调用 `name(x)` 没有正好匹配任何现有函数，但 `name` 是一种数据类型的名称并且 `pg_cast` 提供了一种从 `x` 的类型到这种类型的二进制可强制造型，那么该调用将被翻译为一次二进制可强制造型。通过这种例外，二进制可强制造型能够以函数语法调用，即便没有该函数。同样的，如果没有 `pg_cast` 项，但是该造型是要造型到一种字符



串类型或者是要从一种字符串类型造型，调用将被翻译成一次 I/O 转换造型。这种例外允许以函数语法调用 I/O 转换造型。

### Note

还有一种例外中的例外：从组合类型到字符串类型的 I/O 转换造型不能使用函数语法调用，而必须被写成显式造型语法（CAST或者 ::记号）。增加这种例外是因为在引入了自动提供的 I/O 转换造型之后，在想要引用一个函数或者列时太容易意外地调用这种造型。

## 示例

要使用函数int4(bigint)创建一种从类型 bigint到类型int4的赋值造型：

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

（在系统中这种造型已经被预定义）。

## 兼容性

CREATE CAST命令符合 SQL标准，不过 SQL 没有为二进制可强制 类型或者实现函数的额外参数做好准备。 AS IMPLICIT也是一种 LightDB扩展。

## 另见

CREATE FUNCTION, CREATE TYPE, DROP CAST

---

# CREATE COLLATION

CREATE COLLATION — 定义一种新排序规则

## Synopsis

```
CREATE COLLATION [ IF NOT EXISTS ] name (  
    [ LOCALE = locale, ]  
    [ LC_COLLATE = lc_collate, ]  
    [ LC_CTYPE = lc_ctype, ]  
    [ PROVIDER = provider, ]  
    [ DETERMINISTIC = boolean, ]  
    [ VERSION = version ]  
)  
CREATE COLLATION [ IF NOT EXISTS ] name FROM existing_collation
```

## 描述

CREATE COLLATION使用指定的操作系统区域 设置或者复制一个现有的排序规则来定义新的排序规则。

要创建一种排序规则，你必须拥有目标模式上的 CREATE特权。

## 参数

IF NOT EXISTS

如果已经存在了同名的排序规则，则不要抛出错误。在这种情况下发出一个通知。 请注意，不保证已经存在的排序规则与要创建的这个类似。

name

排序规则的名字，可以被模式限定。如果没有用模式限定，该排序规则 会被定义在当前模式中。排序规则名称在其所处的模式中必须唯一（系统 目录可以为其他编码包含具有相同名称的排序规则，但数据库编码不匹配 时它们会被忽略）。

locale

这是一种一次设置LC\_COLLATE 和LC\_CTYPE的快捷方式。如果你指定它，你就 不能指定那两个参数。

lc\_collate

为LC\_COLLATE区域分类使用指定的操作系统 区域。

lc\_ctype

为LC\_CTYPE区域分类使用指定的操作系统 区域。

provider

指定用于与此排序规则相关的区域服务的提供程序。可能的值是： icu、 libc。 默认是libc。 可用的选择取决于操作系统和构建选项。

DETERMINISTIC

规定排序是否应使用确定性比较。默认值为true。确定性比较认为那些在字节上不相等的字符串是不相等的，即使它们在逻辑上被比较认为相等。 LightDB使用字节比较来断

开连接。不确定的比较可能会使排序规则大小写不敏感或音调不敏感。因此，您需要选择一个适当的LC\_COLLATE设置并且在此处将排序规则设置为不确定。

非确定性排序规则仅被ICU提供者支持。

version

指定使用该排序规则存储的版本字符串。通常忽略该选项，这会导致版本从操作系统提供的排序规则实际版本中计算出来。此选项旨在供lt\_upgrade用于复制现有安装中的版本。

又见ALTER COLLATION获取如何处理排序规则版本错误匹配。

existing\_collation

要复制的一种现有的排序规则的名称。新的排序规则将和现有的具有同样的属性，但是它是一个独立的对象。

## 注解

CREATE COLLATION 采用SHARE ROW EXCLUSIVE锁，在pg\_collation系统分类上，这是自我冲突的，所以一次只能运行一条CREATE COLLATION 命令。

使用DROP COLLATION可移除用户定义的排序规则。

关于如何创建排序规则的更多信息可见Section 21.2.2.3

使用libc排序规则提供程序时，语言环境必须适用于当前的数据库编码。有关精确的规则，请参见CREATE DATABASE。

## 示例

从操作系统区域fr\_FR.utf8创建一种排序规则（假定当前数据库编码是UTF8）：

```
CREATE COLLATION french (locale = 'fr_FR.utf8');
```

使用German phone book排序顺序使用ICU提供程序创建排序规则：

```
CREATE COLLATION german_phonebook (provider = icu, locale = 'de-u-co-phonebk');
```

从一个现有的排序规则创建一个新的排序规则：

```
CREATE COLLATION german FROM "de_DE";
```

能在应用中使用与操作系统无关的排序规则名称就很方便了。

## 兼容性

在 SQL 标准中有一个CREATE COLLATION 语句，但是它被限制为只能复制一个现有的排序规则。创建新排序规则的语法是一种LightDB扩展。

## 另见

ALTER COLLATION, DROP COLLATION

---

# CREATE CONVERSION

CREATE CONVERSION — 定义一种新的编码转换

## Synopsis

```
CREATE [ DEFAULT ] CONVERSION name
    FOR source_encoding TO dest_encoding FROM function_name
```

## 描述

CREATE CONVERSION定义一种两个字符集编码之间的新转换。

标记为DEFAULT的转换将被自动地用于客户端和服务器之间的编码转换。 为了支持这种用法，必须定义两个转换（从编码 A 到 B 以及从编码 B 到 A）。

要创建一个转换，你必须拥有该函数上的EXECUTE特权 以及目标模式上的CREATE特权。

## 参数

DEFAULT

DEFAULT子句表示这个转换是从源编码到目标编码的默认 转换。在一个模式中对于每一个编码对，只应该有一个默认转换。

name

转换的名称，可以被模式限定。如果没有被模式限定，该转换被定义在 当前模式中。在一个模式中，转换名称必须唯一。

source\_encoding

源编码名称。

dest\_encoding

目标编码名称。

function\_name

被用来执行转换的函数。函数名可以被模式限定。如果没有，将在路径 中查找该函数。

该函数必须具有以下的特征：

```
conv_proc(
    integer, -- 源编码 ID
    integer, -- 目标编码 ID
    cstring, -- 源字符串（空值终止的 C 字符串）
    internal, -- 目标（用一个空值终止的 C 字符串填充）
    integer -- 源字符串长度
) RETURNS void;
```

## 注解

源编码和目标编码都不可以是SQL\_ASCII， 因为在涉及SQL\_ASCII “encoding” 的情况下，服务器的行为是硬连接的。

使用DROP CONVERSION可以移除用户定义的转换。

创建转换所要求的特权可能在未来的发行中被更改。

## 示例

使用myfunc创建一个从编码UTF8到 LATIN1的转换：

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

## 兼容性

CREATE CONVERSION是一种 LightDB扩展。在 SQL 标准中 没有CREATE CONVERSION语句，但是有一个目的和语法都类似的 CREATE TRANSLATION语句。

## 另见

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

---

# CREATE DATABASE

CREATE DATABASE — 创建一个新数据库

## Synopsis

```
CREATE DATABASE name
  [ [ WITH ] [ OWNER [=] user_name ]
    [ TEMPLATE [=] template ]
    [ ENCODING [=] encoding ]
    [ LOCALE [=] locale ]
    [ LC_COLLATE [=] lc_collate ]
    [ LC_CTYPE [=] lc_ctype ]
    [ TABLESPACE [=] tablespace_name ]
    [ ALLOW_CONNECTIONS [=] allowconn ]
    [ CONNECTION LIMIT [=] conlimit ]
    [ IS_TEMPLATE [=] istemplate ] ]
  [ LIGHTDB_SYNTAX_COMPATIBLE_TYPE [=] compa_type ] ]
  [ LIGHTDB_MYSQL_LOWER_CASE_TABLE_NAMES [=] mysql_lower_case ] ]
  [ LIGHTDB_ASCII_ZERO_STORE_VALUE [=] ascii_zero_store_value ] ]
```

## 描述

CREATE DATABASE 创建一个新的LightDB数据库。

要创建一个数据库，你必须是一个超级用户或者具有特殊的CREATEDB特权。见CREATE ROLE。

在 LightDB 23.4 版本之前，默认情况下，新数据库将通过克隆标准系统数据库template1被创建。从 LightDB 23.4 开始，标准系统数据库不再是template1，而是template\_pg、template\_oracle 或者 template\_mysql 三者之一。

Table 269. 不同的语法兼容模式LIGHTDB\_SYNTAX\_COMPATIBLE\_TYPE 支持不同的模板：

Template	LIGHTDB_SYNTAX_COMPATIBLE_TYPE		
	off	oracle	mysql
template0	support	support	support
template1	not support	not support	not support
template_pg	support	not support	not support
template_oracle	not support	support	not support
template_mysql	not support	not support	support

在创建数据库时，数据库的语法兼容模式必然是 off、oracle 或者 mysql 三者之一。此时如果额外指定的模板参数和语法兼容模式不兼容，则会提示错误。如果 GUC 参数lightdb\_ignore\_template\_compatible\_type\_mismatch 的值被设置为 true，则LightDB 会忽略模板选项，off、oracle 和 mysql 语法兼容模式会各自使用template\_pg、template\_oracle 和 template\_mysql 作为默认模板。

可以通过写TEMPLATE name指定一个不同的模板。特别地，通过写TEMPLATE template0你可以创建一个原始的数据库（其中没有用户定义的对象存在并且系统对象没有被更改），它将只包含你的LightDB版本所预定义的标准对象。如果你希望避免拷贝任何可能被加入到标准系统数据库中的本地安装对象，这将有所帮助。

请注意：当在创建数据库的过程中提供了用户创建的模板时，兼容性模式检查会被跳过。

## 参数

name

要创建的数据库名。

user\_name

将拥有新数据库的用户的角色名，或者用DEFAULT来使用默认值（即，执行该命令的用户）。要创建一个被另一个角色拥有的数据库，你必须该角色的一个直接或间接成员，或者是一个超级用户。

template

要为其创建新数据库的模板名称，或者用DEFAULT来使用默认模板（在 LightDB 23.4 版本之前默认为 `template_1`，从 LightDB 23.4 开始为 `template_pg`，`template_oracle` 或 `template_mysql`）。

encoding

要在新数据库中使用的字符集编码。指定一个字符串常量（例如 `'SQL_ASCII'`），或者一个整数编码编号，或者DEFAULT来使用默认的编码（即，模板数据库的编码）。LightDB 服务器所支持的字符集在Section 21.3. 中描述。附加限制见下文。

locale

这是一个同时设置LC\_COLLATE和LC\_CTYPE的快捷方式。如果指定此参数，您不能指定这些参数中的任何一个。

### Tip

其他语言环境设置 `lc_messages`，`lc_monetary`，`lc_numeric`，and `lc_time`不是每个数据库固定的，也不是由该命令设置的。如果要将它们设置为特定数据库的默认值，则可以使用 `ALTER DATABASE... SET`。

lc\_collate

要在新数据库中使用的排序规则顺序（LC\_COLLATE）。这会影响到字符串的排序顺序，例如在带 `ORDER BY` 的查询中，以及文本列上索引所使用的顺序。默认是使用模板数据库的排序规则顺序。附加限制见下文。

lc\_ctype

要在新数据库中使用的字符分类（LC\_CTYPE）。会影响字符的类别，如小写、大写和数字。默认是使用模板数据库的字符分类。附加限制见下文。

tablespace\_name

将与新数据库相关联的表空间名称，或者DEFAULT来使用模板数据库的表空间。这个表空间将是在这个数据库中创建的对象表空间。详见CREATE TABLESPACE。

allowconn

如果为假，则没有人能连接到这个数据库。默认为真，表示允许连接（除了 被其他机制约束以外，例如GRANT/REVOKE CONNECT）。

conlimit

这个数据库允许多少并发连接。-1（默认值）表示没有限制。

istemplate

如果为真，则任何具有CREATEDB特权的用户都可以从 这个数据库克隆。如果为假（默认），则只有超级用户或者该数据库的拥有者 可以克隆它。

compa\_type

如果值是oracle，那么这个数据库可以使用oracle的一些功能。如果值是mysql，那么这个数据库可以使用mysql的一些功能。如果这些值处于禁用状态，则此数据库无法使用上述功能。默认值由lightdb\_syntax\_compatible\_type指定

更多关于oracle和mysql的特性，见 lightdb\_dblevel\_syntax\_compatible\_type

mysql\_lower\_case

在Mysql兼容模式下，是否开启表名全小写特性。mysql\_lower\_case的取值为（1, true, on, 0, false, off）中的一个，默认值是1。（1, true, on）：开启（0, false, off）：不开启

ascii\_zero\_store\_value

Lightdb 23.3引入了一个新的数据库级别GUC参数lightdb\_ascii\_zero\_store\_value。此参数取值范围为0到32，包含0，32。数据库创建时设定，之后不能修改。

更多关于GUC lightdb\_ascii\_zero\_store\_value 的信息，请参考 lightdb\_ascii\_zero\_store\_value

可选的参数可以被写成任何顺序，不用按照上面说明的顺序。

## 注解

CREATE DATABASE不能在一个事务块内被执行。

带有一行“不能初始化数据库目录”的错误大部分与在数据目录上权限不足、磁盘满或其他文件系统问题有关。

使用DROP DATABASE移除一个数据库。

程序createdb是这个命令的一个包装器程序，为了使用方便而提供。

数据库层面的配置参数(参见 ALTER DATABASE)和 数据库级权限(参见GRANT)不会从模板数据库中复制。

尽管可以通过指定一个数据库作为模板来从其中而不是标准系统数据库复制，这（还）不是“COPY DATABASE”功能的一般目的。主要的限制是在模板数据库被拷贝期间其他会话不能连接到它。如果CREATE DATABASE启动时还存在任何其他连接，它将失败。否则，到模板数据库的新连接将被挡在外面直到CREATE DATABASE完成。详见Section 20.3

为新数据库指定的字符集编码必须与选定的区域设置（LC\_COLLATE和LC\_CTYPE）相兼容。如果区域是C（或者等效的POSIX），那么所有编码都被允许，但是对于其他区域设置只有一种编码能正确工作（不过，在 Windows 上 UTF-8 编码能够与任何区域一起使用）。CREATE DATABASE将允许超级用户指定SQL\_ASCII编码而不管区域设置，但是这种选择已被废弃并且可能在数据与数据库中存储的区域不是编码兼容时让字符串函数行为失当。

编码和区域设置必须匹配模板数据的编码和区域，除非template0被用作模板。这是因为其他数据库可能包含不匹配指定编码的数据，或者可能包含排序顺序受LC\_COLLATE和LC\_CTYPE影响的索引。拷贝这种数据将导致一个由于该新设置损坏的数据库。不过，template0是不会含有任何可能被影响的数据或索引的。



CONNECTION LIMIT选项大概是唯一会被强制的，如果两个新会话在大约同一时间开始并且那时该数据库只剩有一个连接“槽”，可能两者都会失败。还有，该限制对超级用户或后台工作进程无效。

自23.1版本以来，LightDB支持在执行CREATE DATABASE命令时创建具有相同名称的用户。该特性仅在oracle兼容模式时生效，不支持分布式<sup>1</sup>。自动创建的同名用户对创建的数据库具有CREATE、CONNECT、TEMPORARY和LOGIN权限。

## 例子

要创建一个新数据库：

```
CREATE DATABASE lusiadas;
```

要在一个默认表空间salespace中创建一个被用户salesapp拥有的新数据库sales：

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salespace;
```

要用不同的语言环境创建数据库music：

```
CREATE DATABASE music
  LOCALE 'sv_SE.utf8'
  TEMPLATE template0;
```

在这个例子中，如果指定的语言环境与template1中的语言环境不同，则需要TEMPLATE template0子句。（如果不是，则明确指定区域设置是多余的。）

要用不同的语言环境和不同的字符集编码创建数据库music2：

```
CREATE DATABASE music2
  LOCALE 'sv_SE.iso885915'
  ENCODING LATIN9
  TEMPLATE template0;
```

指定的区域设置和编码设置必须匹配，否则会报告错误。

请注意，区域名称是特定于操作系统的，因此上述命令可能无法在任何地方以相同的方式工作。

在创建数据库testdb之前，查询数据库列表和用户列表：

```
lightdb@postgres=# \l
```

Name	Owner	Encoding	Collate	Ctype	Access privileges
lt_test	lightdb	UTF8	zh_CN.UTF-8	zh_CN.UTF-8	
postgres	lightdb	UTF8	zh_CN.UTF-8	zh_CN.UTF-8	
template0	lightdb	UTF8	zh_CN.UTF-8	zh_CN.UTF-8	=c/lightdb
lightdb					lightdb=CtC/

<sup>1</sup> <http://www.light-pg.com/docs/canopy/current/index.html>

CREATE DATABASE

```

template1 | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/lightdb
+
lightdb | | | | | lightdb=CTc/
lightdb
template_pg | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/lightdb
+
lightdb | | | | | lightdb=CTc/
lightdb
template_mysql | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/lightdb
+
lightdb | | | | | lightdb=CTc/
lightdb
template_oracle | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/lightdb
+
lightdb | | | | | lightdb=CTc/
(7 rows)

```

```

lightdb@postgres=#
lightdb@postgres=# \du

```

List of roles  
Attributes

```

Role name | | Member
of
-----+-----
lightdb | Superuser, Create role, Create DB, Replication, Bypass RLS | {}

```

```

lightdb@postgres=#

```

To create a database in oracle compatibility mode:

```

lightdb@postgres=# create database testdb with lightdb_syntax_compatible_type
oracle;
NOTICE: auto create user "testdb" success
CREATE DATABASE
lightdb@postgres=#

```

Query the database list and user list again:

```

lightdb@postgres=# \l

```

List of databases

```

Name | Owner | Encoding | Collate | Ctype | Access
privileges
-----+-----
lt_test | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 |
postgres | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 |
template0 | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/lightdb
+
lightdb | | | | | lightdb=CTc/
lightdb
template1 | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/lightdb
+
lightdb | | | | | lightdb=CTc/
lightdb
template_pg | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/lightdb
+

```

```

lightdb
template_mysql | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/lightdb
+
lightdb
template_oracle | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/lightdb
+
lightdb
testdb | lightdb | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =Tc/lightdb
+
lightdb+
lightdb
(8 rows)

```

```

lightdb@postgres=#
lightdb@postgres=# \du

```

```

                                List of roles
Role name | Attributes | Member
-----+-----+-----
lightdb | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
testdb | | {}

```

```

lightdb@postgres=#

```

## 兼容性

在 SQL 标准中没有CREATE DATABASE语句。数据库等效于目录，而目录的创建由实现定义。

## 参见

ALTER DATABASE, DROP DATABASE

---

# CREATE DOMAIN

CREATE DOMAIN — 定义一个新的域

## Synopsis

```
CREATE DOMAIN name [ AS ] data_type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
```

其中 constraint 是:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

## 描述

CREATE DOMAIN创建一个新的域。域本质上是一种带有可选约束（在允许的值集合上的限制）的数据类型。定义一个域的用户将成为它的拥有者。

如果给定一个模式名（例如CREATE DOMAIN myschema.mydomain ...），那么域将被创建在该指定的模式中。否则它会被创建在当前模式中。域的名称在其模式中的类型和域之间必须保持唯一。

域主要被用于把字段上的常用约束抽象到一个单一的位置以便维护。例如，几个表可能都包含电子邮件地址列，而且都要求相同的 CHECK 约束来验证地址的语法。可以为此定义一个域，而不是在每个表上都单独设置一个约束。

要创建一个域，你必须在其底层类型上拥有USAGE特权。

## 参数

name

要被创建的域的名称（可以被模式限定）。

data\_type

域的底层数据类型。可以包括数组指示符。

collation

用于该域的可选的排序规则。如果没有指定排序规则，将使用底层数据类型的默认排序规则。如果指定了COLLATE，底层类型必须是可排序的。

DEFAULT expression

DEFAULT子句为该域数据类型的列指定一个默认值。该值是所有没有变量的表达式（但不允许子查询）。默认值表达式的数据类型必须匹配域的数据类型。如果没有指定默认值，那么默认值就是空值。

默认值表达式将被用在任何没有指定列值的插入操作中。如果为一个特定列定义了默认值，它会覆盖与域相关的默认值。继而，域默认值会覆盖任何与底层数据类型相关的默认值。

CONSTRAINT constraint\_name

一个约束的名称（可选）。如果没有指定，系统会生成一个名称。

NOT NULL

这个域的值通常不能为空值（但是看看下面的注释）。

NULL

这个域的值允许为空值。这是默认值。

这个子句只是为了与非标准 SQL 数据库相兼容而设计。在新的 应用中不鼓励使用它。

CHECK (expression)

CHECK子句指定该域的值必须满足的完整性 约束或者测试。每一个约束必须是一个产生布尔结果的表达式。 它应该使用关键词VALUE来引用要被测试的值。计算为 TRUE 或者 UNKNOWN 的表达式成功。如果该表达式产生一个 FALSE 结果，会报告一个错误并且该值不允许被转换成该域类型。

当前，CHECK表达式不能包含子查询，也不能 引用除VALUE之外的其他变量。

当一个域有多个CHECK约束，将按照其名字的字 母顺序测试它们（版本 9.5 之前的 LightDB 不遵循任何用于CHECK约束的特定触发顺序）。

## 注解

在把一个值转换成域类型时会检查域约束（特别是NOT NULL）。即使有一个这样的约束，有可能一个名义上属于该域类型的列也会被读成空值。 例如，如果在一次外连接查询中，属于该域的列出现在外连接的空值端。下面 是一个更精细的例子：

```
INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false));
```

空的标量子-SELECT 将产生一个空值，它被认为是该域类型的值，因此不会 在其上应用任何进一步的约束检查，并且插入将会成功。

要避免这类问题很难，因为 SQL 的一般假设是空值也是每一种数据类型的合法值。因此，最好的方法是设计一个允许空值的域约束，然后根据需要在该域类型的列上 应用列的NOT NULL约束。

LightDB假定CHECK约束的条件是不可变的，也就是说，对于相同的输入值它们总会给出相同的结果。 仅在首次将值转换为域类型时，此假设检查CHECK约束的正确性，而不是在其他时候。（这与处理表CHECK约束基本上相同，如 Section 5.4. 中所述。）

打破此假设的常见方法的一个示例是在 CHECK表达式中引用一个用户定义的函数，然后更改该函数的行为。 LightDB不会禁止这样做，但是此间如果有域类型的存储值违反CHECK约束，它也不会注意到。这将导致后续数据库转储和重新加载失败。 处理此类更改的建议方法是删除约束（使用ALTER DOMAIN），调整函数定义，重新添加约束，然后根据存储的数据重新检查约束。

## 示例

这个例子创建us\_postal\_code数据类型并且把它用在 一个表定义中。一个正则表达式测试被用来验证值是否看起来像一个 合法的 US 邮政编码：

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK (
```

```
    VALUE ~ '^\\d{5}$'  
OR VALUE ~ '^\\d{5}-\\d{4}$'  
);  
  
CREATE TABLE us_snail_addy (  
    address_id SERIAL PRIMARY KEY,  
    street1 TEXT NOT NULL,  
    street2 TEXT,  
    street3 TEXT,  
    city TEXT NOT NULL,  
    postal us_postal_code NOT NULL  
);
```

## 兼容性

命令CREATE DOMAIN符合 SQL 标准。

## 另见

ALTER DOMAIN, DROP DOMAIN

---

# CREATE EVENT TRIGGER

CREATE EVENT TRIGGER — 定义一个新的事件触发器

## Synopsis

```
CREATE EVENT TRIGGER name
  ON event
  [ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ]
  EXECUTE { FUNCTION | PROCEDURE } function_name()
```

## 描述

CREATE EVENT TRIGGER创建一个新的事件触发器。只要指定的事件发生并且与该触发器相关的WHEN条件（如果有）被满足，该触发器的函数将被执行。关于事件触发器的一般性介绍可见 Chapter 38 创建事件触发器的用户会成为它的拥有者。

## 参数

name

给新触发器的名称。在该数据库中这个名称必须唯一。

event

会触发对给定函数调用的事件名称。更多事件名称的信息请见 Section 38.1

filter\_variable

用来过滤事件的变量名称。这可以用来限制触发器只为它支持的那一部分情况引发。当前唯一支持的 filter\_variable 是TAG。

filter\_value

与该触发器要为其引发的 filter\_variable相关联的一个值列表。对于TAG，这表示一个命令标签列表（例如 'DROP FUNCTION'）。

function\_name

一个用户提供的函数，它被声明为没有参数并且返回类型 event\_trigger。

在CREATE EVENT TRIGGER的语法中，关键字CREATE EVENT TRIGGER和PROCEDURE是等效的，但是被引用的函数在任何情况下都必须是函数，而不是过程。此处关键字PROCEDURE的使用是历史性的，已弃用。

## 注解

只有超级用户能创建事件触发器。

在单用户模式（见lightdb）中事件触发器被禁用。如果一个错误的事件触发器禁用了数据库让你甚至无法删除它，可以重启到单用户模式，这样你就能删除它。

## 示例

禁止执行任何DDL命令：

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
  LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'command % is disabled', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
  EXECUTE FUNCTION abort_any_command();
```

## 兼容性

在 SQL 标准中没有 CREATE EVENT TRIGGER 语句。

## 另见

ALTER EVENT TRIGGER, DROP EVENT TRIGGER, CREATE FUNCTION



---

# CREATE EXTENSION

CREATE EXTENSION — 安装一个扩展

## Synopsis

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
    [ WITH ] [ SCHEMA schema_name ]
    [ VERSION version ]
    [ CASCADE ]
```

## 描述

CREATE EXTENSION把一个新的扩展载入到 当前数据库中。不能有同名扩展已经被载入。

载入一个扩展本质上是运行该扩展的脚本文件。该脚本通常将创建新的 SQL对象，例如函数、数据类型、操作符以及索引支持 方法。CREATE EXTENSION会额外地记录 所有被创建对象的标识，这样发出 DROP EXTENSION时可以删除它们。

运行CREATE EXTENSION的用户成为扩展的所有者， 以进行以后的特权检查，并且通常也成为由扩展的脚本创建的任何对象的所有者。

通常，加载扩展需要与创建扩展组件对象所需的特权相同的特权。 对于许多扩展，这意味着需要超级用户特权。 但是，如果扩展名在其控制文件中标记为trusted，则该扩展名可以由对当前 数据库具有CREATE特权的任何用户安装。 在这种情况下，扩展对象本身将由主叫用户拥有，但是包含的对象将由引导超级用户拥有（除非扩展的脚本将其明确分配给主叫用户）。 此配置使呼叫用户有权删除该扩展名，但不能修改其中的单个对象。

## 参数

IF NOT EXISTS

已有同名扩展存在时不要抛出错误。这种情况下会发出一个提示。 注意，不保证现有的扩展与将要从当前可用的脚本文件创建的脚本 有任何相似。

extension\_name

要安装的扩展的名称。LightDB 将使用文件 SHAREDIR/extension/extension\_name.control 中的指令来创建该扩展。

schema\_name

假定该扩展允许其内容被重定位，这是要在其中安装该扩展的对象的 模式名称。被提到的模式必须已经存在。如果没有指定并且该扩展的 控制文件也没有指定一个模式，将使用当前的默认对象创建模式。

如果该扩展在其控制文件中指定了一个schema参数， 那么不能用SCHEMA子句覆盖该模式。通常，如果 给出了一个SCHEMA子句并且它与扩展的 schema参数冲突，则会发生错误。不过，如果也给出了CASCADE子句，则schema冲突时会忽略 schema\_name。 给定的schema\_name 将被用来安装任何需要的并且没有在其控制文件中指定 schema的扩展。

记住扩展本身被认为不在任何模式中：扩展具有无限定的名称，并且 要在整个数据库范围内唯一。但是属于扩展的对象可以在模式中。

version

要安装的扩展的版本。这可以写成一个标识符或者一个字符串。 默认版本在该扩展的控制文件中指定。

## CASCADE

自动安装这个扩展所依赖的任何还未安装的扩展。它们的依赖也会同样 被自动安装。如果给出SCHEMA子句，它会应用于这种方式 下安装的所有扩展。这个语句的其他选项不会被应用于自动安装的扩展。特别地，这些自动安装的扩展的默认版本将被选中。

## 注解

在使用CREATE EXTENSION载入扩展到数据库中之前，必须先安装好该扩展的支持文件。关于安装 LightDB提供的扩展的信息可以在 额外提供的模块中找到。

当前可以用于载入的扩展可以在系统视图 `pg_available_extensions` 或者 `pg_available_extension_versions` 中看到。

### Caution

以超级用户身份安装扩展程序需要相信扩展程序的作者以安全的方式编写了扩展程序安装脚本。对于恶意用户而言，创建特洛伊木马对象并不困难，而这些特洛伊木马对象会损害以后粗心编写 的扩展脚本的执行，从而使该用户获得超级用户特权。但是，仅当木马对象在脚本执行期间位于 `search_path`中时，它们才是危险的，这意味着它们位于扩展的安装 目标模式或它依赖的某些扩展的模式中。因此，在处理未经仔细审查其脚本的扩展时，一个好的 经验法则是仅将其安装到尚未具有CREATE特权且不会授予任何不可信用户的模式中。对它们依赖的任何扩展也是一样。

针对此类安装时攻击，LightDB提供的插件可以认为是 针对这种安装时攻击的安全，除了少数插件依赖于其它的扩展。如这些扩展的文档中所述，应将 它们安装到安全模式中，或与它们所依赖的扩展安装在相同的模式中，或拓展和其依赖都安装到 安全模式。

更多有关编写新扩展的内容请见Section 36.17

## 示例

安装hstore扩展到当前数据库中，将其对象放置在 addons模式中：

```
CREATE EXTENSION hstore SCHEMA addons;
```

另一种方法：

```
SET search_path = addons;
CREATE EXTENSION hstore;
```

## 兼容性

CREATE EXTENSION是一种 LightDB扩展。

## 另见

ALTER EXTENSION, DROP EXTENSION

---

# CREATE FOREIGN DATA WRAPPER

CREATE FOREIGN DATA WRAPPER — 定义一个新的外部数据包装器

## Synopsis

```
CREATE FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

## 描述

CREATE FOREIGN DATA WRAPPER创建一个 新的外部数据包装器。定义外部数据包装器的用户将成为它的拥有者。

在数据库内外部数据包装器名称必须唯一。

只有超级用户能够创建外部数据包装器。

## 参数

name

要创建的外部数据包装器的名称。

HANDLER handler\_function

handler\_function是一个以前注册 好的函数的名称，它将被调用来为外部表检索执行函数。处理器函数必须不能有参数，并且它的返回类型必须是fdw\_handler。

可以创建一个没有处理器函数的外部数据包装器，但是使用这个包装 器的外部表只能被声明，但不能被访问。

VALIDATOR validator\_function

validator\_function 是一个之前已注册的函数的名称，它将被调用来检查给予该外部数据包装器 的选项，还有用于外部服务器、用户映射以及使用 该外部数据包装器的外部表的选项。如果没有验证器函数或者指定了 NO VALIDATOR，那么在创建时不会检查选项（ 外部数据包装器可能会在运行时忽略或者拒绝无效的选项说明，这取决于 实现）。验证器函数必须接受两个参数：一个类型是text[]， 它将包含存储在系统目录中的选项数组，另一个是类型 oid，它将是包含该选项的系统目录的 OID。返回类型 会被忽略，该函数应该使用ereport(ERROR) 函数报告无效选项。

OPTIONS ( option 'value' [, ... ] )

这个子句为新的外部数据包装器指定选项。允许的选项名称和值与每一个 外部数据包装器有关，并且它们会被该外部数据包装器的验证器函数验证。 选项名称必须唯一。

## 注解

LightDB的外部数据功能仍在积极的开发中。 查询的优化还很原始（也是剩下工作最多的部分）。因此，未来还有很 可观的性能提升空间。

## 示例

创建一个无用的外部数据包装器dummy:

```
CREATE FOREIGN DATA WRAPPER dummy;
```

用处理器函数file\_fdw\_handler创建一个外部数据包装器 file:

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

用一些选项创建一个外部数据包装器mywrapper:

```
CREATE FOREIGN DATA WRAPPER mywrapper  
  OPTIONS (debug 'true');
```

## 兼容性

CREATE FOREIGN DATA WRAPPER 符合 ISO/IEC 9075-9 (SQL/MED), 不过HANDLER和VALIDATOR子句是扩展, 并且标准子句 LIBRARY和LANGUAGE还没有在LightDB中被实现。

不过要注意, 整体上来说 SQL/MED 功能还没有符合。

## 另见

ALTER FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE

---

# CREATE FOREIGN TABLE

CREATE FOREIGN TABLE — 定义一个新的外部表

## Synopsis

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
    { column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ]
  [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint }
  [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
PARTITION OF parent_table [ (
    { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
  [, ... ]
) ]
{ FOR VALUES partition_bound_spec | DEFAULT }
SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]
```

其中 column\_constraint 是:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED }
```

而 table\_constraint 是:

```
[ CONSTRAINT constraint_name ]
CHECK ( expression ) [ NO INHERIT ]
```

而 partition\_bound\_spec 是:

```
IN ( partition_bound_expr [, ...] ) |
FROM ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] )
  TO ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )
```

## 描述

CREATE FOREIGN TABLE在当前数据库中创建 一个新的外部表。该表将由发出这个命令的用户所拥有。

如果给定了一个模式名称（例如CREATE FOREIGN TABLE myschema.mytable ...），那么该表会被创建在指定的模式中。 否则它会被创建在当前模式中。该外部表的名称必须与同一个模式中的任何其他外部表、表、序列、索引、视图或者物化视图区分开来。

CREATE FOREIGN TABLE还将自动创建一个数据类型来表示该外部表行相应的组合类型。因此，外部表不能和同一个模式中任何现有的数据类型同名。

如果指定了PARTITION OF子句，则该表被创建为具有指定边界的parent\_table的分区。

要创建一个外部表，你必须具有该外部服务器上的USAGE 特权，以及该表中用到的所有列类型上的USAGE特权。

## 参数

IF NOT EXISTS

已经存在同名关系时不要抛出错误。这种情况下会发出一个提示。注意，并不保证已经存在的关系与将要创建的那一个相似。

table\_name

要创建的表的名称（可以被模式限定）。

column\_name

要在新表中创建的列名。

data\_type

该列的数据类型。可以包括数组指示符。更多 LightDB支持的数据类型可见Chapter 8

COLLATE collation

COLLATE子句为该列（必须是一个可排序的数据类型）赋予一个排序规则。如果没有指定，则会使用该列的数据类型的默认排序规则。

INHERITS ( parent\_table [, ... ] )

可选的INHERITS子句指定了一个表的列表，新的外部表会自动从中继承所有列。父表可以是普通表或者外部表。详见 CREATE TABLE的类似形式。

PARTITION OF parent\_table { FOR VALUES partition\_bound\_spec | DEFAULT }

此语句可以用来将外部表创建为父表的一个指定区间的分区表。详见CREATE TABLE的类似形式。注意如果父表存在UNIQUE类型的索引时，当前是不允许将外部表创建为该父表的分区表。（另见 ALTER TABLE ATTACH PARTITION。）

CONSTRAINT constraint\_name

一个可选的用于列或者表约束的名字。如果该约束被违背，这个约束名字会出现在错误消息中，这样col must be positive这种约束名就可以被用来与客户端应用交流有用的约束信息（指定包含空格的约束名需要使用双引号）。如果没有指定约束名，系统会自动生成一个。

NOT NULL

该列不允许包含空值。

NULL

该列可以包含空值，这是默认值。

提供这个子句只是为了兼容非标准的 SQL 数据库。在新的应用中不鼓励使用它。

CHECK ( expression ) [ NO INHERIT ]

CHECK子句指定一个产生布尔结果的表达式，该外部表中的每一行都应该满足该表达式。也就是说，对于该外部表中所有的行，这个表达式应该产生 TRUE 或者 UNKNOWN 而

不能产生 FALSE。被 作为列约束指定的检查约束应该只引用该列的值，而出现在表约束中的 表达式可以引用多列。

当前，CHECK表达式不能包含子查询，也不能 引用除当前行的列之外的其他变量。可以引用系统列 tableoid，但是不能引用其他系统列。

被标记为NO INHERIT的约束将不会传播到子表上。

DEFAULT default\_expr

DEFAULT子句为包含它的列定义赋予一个默认数据值。该 值是任意不包含变量的表达式（不允许子查询和对当前表中其他列的交叉 引用）。默认值表达式的数据类型必须匹配列的数据类型。

默认值表达式将被用在任何没有指定列值的插入操作中。如果一列没有 默认值，则默认值为空值。

GENERATED ALWAYS AS ( generation\_expr ) STORED

该子句创建一个生成列。 此列不能写入，只能在读取时返回所指定的表达式的值。

关键字STORED是必须的，用来表明该列将在写入时计算。（计算出的值将会传递给外部数据封装器负责保存，并在读取时返回。）

生成表达式可以引用表中的其他列，但不能引用其他的生成列。所使用的函数和操作符必须是不可变。不能引用其他表。

server\_name

要用于该外部表的一个现有外部服务器的名称。有关定义一个服务器 的细节可以参考CREATE SERVER。

OPTIONS ( option 'value' [, ...] )

要与新外部表或者它的一个列相关联的选项。被允许的选项名称和值是与 每一个外部数据包装器相关的，并且它们会被该外部数据包装器的验证器 函数验证。不允许重复的选项名称（不过一个表选项和一个列选项重名是 可以的）。

## 注解

LightDB核心系统不会强制外部表上的约束（例如CHECK或NOT NULL子句），大部分外部 数据包装器也不会尝试强制它们。也就是说，这类约束会被简单地认为保持 为真。这种强制其实没什么意义，因为它只适用于通过该外部表插入或者更 新的行，而对通过其他方式修改的行（例如直接在远程服务器上修改）没有 作用。附着在外部表上的约束应该表示由外部服务器强制的一个约束。

有些特殊目的的外部数据包装器可能是它们所访问的数据的唯一一种访问 机制，在那种情况下让外部数据包装器自己来执行约束强制可能是合适的。 但是不应该假设包装器会这样做，除非它的文档说它会。

尽管LightDB不会尝试强制外部表上的约束， 但它确实假定它们对于查询优化的目的是正确的。如果在外部表中有不 满足约束的行可见，在该表上的查询可能会产生不正确的回答。确保 约束定义符合实际是用户的责任。

### Caution

当外部表用作分区表的分区时，存在一个隐式约束，即其内容必须满足分区规则。同样，用户有责任确保这是真的，最好在远程服务器上安装匹配的约束。

在包含外部表分区的分区表中，如果外部数据包装器支持元组路由，则更改分区键值的UPDATE可以导致行从本地分区移动到外部表分区。然而，目前不可能将一行从外部表分区移动到另一个分区。如果远程服务器正确执行了分区约束，则需要执行的UPDATE将失败。

类似的要点也适用于生成列。生成列在本地LightDB 服务器发生插入或更新时进行计算，并将计算结果传递给外部数据封装器，由外部数据封装器负责把结果写到外部数据存储中，但是并不强制要求查询外部表时返回的生成列 的值一定是与生成表达式一致。总而言之，此行为可能导致查询到不正确的结果。

## 示例

创建外部表films，通过服务器film\_server访问它：

```
CREATE FOREIGN TABLE films (
    code      char(5) NOT NULL,
    title     varchar(40) NOT NULL,
    did       integer NOT NULL,
    date_prod date,
    kind      varchar(10),
    len       interval hour to minute
)
SERVER film_server;
```

创建外部表measurement\_y2016m07，通过服务器 server\_07访问它，作为范围分区表measurement 的分区：

```
CREATE FOREIGN TABLE measurement_y2016m07
    PARTITION OF measurement FOR VALUES FROM ('2016-07-01') TO ('2016-08-01')
    SERVER server_07;
```

## 兼容性

CREATE FOREIGN TABLE命令大部分符合 SQL标准。不过，与 CREATE TABLE 很相似，它允许NULL约束以及零列外部表。能够指定列默认值 也是一种LightDB扩展。LightDB定义的表继承形式是非标准的。

## 另见

ALTER FOREIGN TABLE, DROP FOREIGN TABLE, CREATE TABLE, CREATE SERVER, IMPORT FOREIGN SCHEMA



---

# CREATE FUNCTION

CREATE FUNCTION — 定义一个新函数

## Synopsis

```
CREATE [ OR REPLACE ] FUNCTION
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
  [, ...] ] )
  [ RETURNS rettype
  | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [, ... ]
  | WINDOW
  | { IMMUTABLE | STABLE | VOLATILE }
  | [ NOT ] LEAKPROOF
  | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
  | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
  | PARALLEL { UNSAFE | RESTRICTED | SAFE }
  | COST execution_cost
  | ROWS result_rows
  | SUPPORT support_function
  | SET configuration_parameter { TO value | = value | FROM CURRENT }
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
  } ...
```

## 描述

CREATE FUNCTION定义一个新函数。CREATE OR REPLACE FUNCTION将创建一个新函数或者替换一个现有的函数。要定义一个函数，用户必须具有该语言上的USAGE特权。

如果包括了一个模式名，那么该函数会被创建在指定的模式中。否则，它会被创建在当前模式中。新函数的名称不能匹配同一个模式中具有相同输入参数类型的任何现有函数或过程。不过，不同参数类型的函数和过程能够共享一个名字（这被称作重载）。

要替换一个现有函数的当前定义，可以使用CREATE OR REPLACE FUNCTION。但不能用这种方式更改函数的名称或者参数类型（如果尝试这样做，实际上就会创建一个新的不同的函数）。还有，CREATE OR REPLACE FUNCTION将不会让你更改一个现有函数的返回类型。要这样做，你必须先删除再重建该函数（在使用OUT参数时，这意味着除了删除函数之外无法更改任何OUT参数的类型）。

当CREATE OR REPLACE FUNCTION被用来替换一个现有的函数，该函数的拥有权和权限不会改变。所有其他的函数属性会按照该命令中所指定的或者隐含的来赋值。必须拥有（包括成为拥有角色的成员）该函数才能替换它。

如果你删除并且重建一个函数，新函数将和旧的不一样，你将必须删掉引用旧函数的现有规则、视图、触发器等。使用CREATE OR REPLACE FUNCTION更改一个函数定义不会破坏引用该函数的对象。还有，ALTER FUNCTION可以被用来更改一个现有函数的大部分辅助属性。

创建该函数的用户将成为该函数的拥有者。

要创建一个函数，你必须拥有参数类型和返回类型上的USAGE特权。

有关编写函数的详细信息，请参阅Section 36.3

## 参数

name

要创建的函数的名称（可以被模式限定）。

argmode

一个参数的模式：IN、OUT、INOUT或者VARIADIC。如果省略，默认为IN。只有OUT参数能跟在一个VARIADIC参数后面。还有，OUT和INOUT参数不能和RETURNS TABLE符号一起使用。

argname

一个参数的名称。一些语言（包括SQL和PL/pgSQL）让你在函数体中使用该名称。对于其他语言，一个输入参数的名字只是额外的文字（就该函数本身所关心的来说）。但是你可以在调用一个函数时使用输入参数名来提高可读性（见Section 4.3）。在任何情况下，输出参数的名称是有意义的，因为它定义了结果行类型中的列名（如果忽略一个输出参数的名称，系统将选择一个默认的列名）。

argtype

该函数参数（如果有）的数据类型（可以是模式限定的）。参数类型可以是基本类型、组合类型或者域类型，或者可以引用一个表列的类型。

根据实现语言，也可以允许指定cstring之类的“伪类型”。伪类型表示实际参数类型没有被完整指定或者不属于普通SQL数据类型集合。

可以写table\_name.column\_name%TYPE来引用一列的类型。使用这种特性有时可以帮助创建一个不受表定义更改影响的函数。

default\_expr

如果参数没有被指定值时要用作默认值的表达式。该表达式必须能被强制为该参数的参数类型。只有输入（包括INOUT）参数可以具有默认值。所有跟随在一个具有默认值的参数之后的输入参数也必须具有默认值。

rettype

返回数据类型（可能被模式限定）。返回类型可以是一种基本类型、组合类型或者域类型，也可以引用一个表列的类型。根据实现语言，也可以允许指定cstring之类的“伪类型”。如果该函数不会返回一个值，可以指定返回类型为void。

当有OUT或者INOUT参数时，可以省略RETURNS子句。如果存在，该子句必须和输出参数所表示的结果类型一致：如果有多个输出参数，则为RECORD，否则与单个输出参数的类型相同。

SETOF修饰符表示该函数将返回一个项的集合而不是一个单一项。

可以写table\_name.column\_name%TYPE来引用一列的类型。

column\_name

RETURNS TABLE语法中一个输出列的名称。这实际上是另一种声明OUT参数的方法，不过RETURNS TABLE也隐含了RETURNS SETOF。

column\_type

RETURNS TABLE语法中的输出列的数据类型。

lang\_name

用以实现该函数的语言的名称。可以是sql、c、internal或者一个用户定义的过程语言的名称，例如plpgsql。如果指定了sql\_body，则默认值为sql。不推荐使用单引号将名称括起来，并要求大小写匹配。

TRANSFORM { FOR TYPE type\_name } [, ... ] }

一个由转换构成的列表，对该函数的调用适用于它们。转换在SQL类型和语言相关的数据类型之间进行变换，详见CREATE TRANSFORM。过程语言实现通常把有关内建类型的知识硬编码在代码中，因此那些不需要列举在这里。如果一种过程语言实现不知道如何处理一种类型并且没有转换被提供，它将回退到一种默认的行为来转换数据类型，但是这取决于具体实现。

WINDOW

WINDOW表示该函数是一个窗口函数而不是一个普通函数。当前只用于用C编写的函数。在替换一个现有函数定义时，不能更改WINDOW属性。

IMMUTABLE

STABLE

VOLATILE

这些属性告知查询优化器该函数的行为。最多只能指定其中一个。如果这些都不出现，则会默认为VOLATILE。

IMMUTABLE表示该函数不能修改数据库并且对于给定的参数值总是会返回相同的值。也就是说，它不会做数据库查找或者使用没有在其参数列表中直接出现的信息。如果给定合格选项，任何用全常量参数对该函数的调用可以立刻用该函数值替换。

STABLE表示该函数不能修改数据库，并且对于相同的参数值，它在一次表扫描中将返回相同的结果。但是这种结果在不同的SQL语句执行期间可能会变化。对于那些结果依赖于数据库查找、参数变量（例如当前时区）等的函数来说，这是合适的（对希望查询被当前命令修改的行的AFTER触发器不适合）。还要注意current\_timestamp函数族适合被标记为稳定，因为它们的值在一个事务内不会改变。

VOLATILE表示该函数的值在一次表扫描中都有可能改变，因此不能做优化。在这种意义上，相对较少的数据库函数是不稳定的，一些例子是random()、currval()、timeofday()。但是注意任何有副作用的函数都必须被分类为不稳定的，即便其结果是可以预测的，这是为了调用被优化掉。一个例子是setval()。

更多细节可见Section 36.7

LEAKPROOF

LEAKPROOF表示该函数没有副作用。它不会泄露有关其参数的信息（除了通过返回值）。例如，一个只对某些参数值抛出错误消息而对另外一些却不抛出错误的函数不是防泄漏的，一个把参数值包括在任何错误消息中的函数也不是防泄漏的。这会影响到系统如何执行在使用security\_barrier选项创建的视图或者开启了行级安全性的表上执行查询。对于包含有非防泄漏函数的查询，系统将在任何来自查询本身的用户提供条件之前强制来自安全策略或者安全屏障的条件，防止无意中的数据暴露。被标记为防泄漏的函数和操作符被假定是可信的，并且可以在安全性策略和安全性屏障视图的条件之前被执行。此外，没有参数的函数或者不从安全屏障视图或表传递任何参数的函数不一定要被标记为防泄漏的。详见CREATE VIEW和Section 39.5 这个选项只能由超级用户设置。

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

CALLED ON NULL INPUT（默认）表示在某些参数为空值时应正常调用该函数。如果有必要，函数的作者应该负责检查空值并且做出适当的相应。

RETURNS NULL ON NULL INPUT或STRICT表示只要其任意参数为空值，该函数就会返回空值。如果指定了这个参数，当有空值参数时该函数不会被执行，而是自动返回一个空值结果。

[EXTERNAL] SECURITY INVOKER  
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER表示要用调用该函数的用户的特权来执行它。这是默认值。SECURITY DEFINER指定要用拥有该函数的用户的特权来执行该函数。

为了符合 SQL，允许使用关键词EXTERNAL。但是它是可选的，因为与 SQL 中不同，这个特性适用于所有函数而不仅是那些外部函数。

PARALLEL

PARALLEL UNSAFE表示该函数不能在并行模式中运行并且 SQL 语句中存在一个这样的函数会强制使用顺序执行计划。这是默认选项。PARALLEL RESTRICTED表示该函数能在并行模式中运行，但是其执行被限制在并行组的领导者中。PARALLEL SAFE表示该函数对于在并行模式中运行是安全的并且不受限制。

如果函数修改任何数据库状态、会使用子事务之类的方式改变事务、访问序列或者对设置（如setval）做出持久性的更改，它们就应该被标记为并行不安全。如果它们访问临时表、客户端连接状态、游标、预备语句或者系统无法在并行模式中同步的本地后端状态（例如setseed只能在组领导者中执行，因为另一个进程所作的更改不会在领导者中被反映出来），它们应该被标为并行受限。通常，如果一个函数是受限的或者不安全的却被标成了安全，或者它本来是不安全的却被标成了受限，在并行查询中执行时它可能会抛出错误或者产生错误的结果。如果被错误的标记，C 语言函数理论上可能展现出完全无法定义的行为，因为系统没有办法保护自己不受任意的 C 代码影响，但是在大部分情况下其结果也不会比其他函数差到哪里去。如果有疑问，函数应该被标为UNSAFE，这也是默认值。

COST execution\_cost

一个给出该函数的估计执行代价的正数，单位是cpu\_operator\_cost。如果该函数返回一个集合，这就是每个被返回行的代价。如果没有指定代价，对 C 语言和内部函数会指定为 1 个单位，对其他语言的函数则会指定为 100 单位。更大的值会导致规划器尝试避免对该函数的不必要的过多计算。

ROWS result\_rows

一个正数，它给出规划器期望该函数返回的行数估计。只有当该函数被声明为返回一个集合时才允许这个参数。默认假设为 1000 行。

SUPPORT support\_function

用于此函数的planner support\_function的名称（可选的模式限定）。详情请参见Section 36.11 你必须是超级用户才能使用此选项。

configuration\_parameter  
value

SET子句导致进入该函数时指定配置参数将被设置为指定值。并且在该函数退出时恢复到该参数之前的值。SET FROM CURRENT会把CREATE FUNCTION被执行时该参数的当前值保存为进入该函数时将被应用的值。

如果一个SET子句被附加到一个函数，那么在该函数内为同一个变量执行的SET LOCAL命令会被限制于该函数：在函数退出时该配置参数之前的值仍会被恢复。不过，一个普通的SET命令（没有LOCAL）会覆盖SET子句，更像一个之前的SET LOCAL命令所做的那样：这种命令的效果在函数退出后将会持续，除非当前事务被回滚。

更多有关允许的参数名和参数值的信息请见SET和Chapter 17

definition

一个定义该函数的字符串常量，其含义取决于语言。它可以是一个内部函数名、一个对象文件的路径、一个 SQL 命令或者用一种过程语言编写的文本。

美元引用Section 4.1.2. 通常对书写函数定义字符串有所帮助，而普通单引号语法则不会有。如果没有美元引用，函数定义中的任何单引号或者反斜线必须用双写来转义。

obj\_file, link\_symbol

当 C 语言源代码中该函数的名称与 SQL 函数的名称不同时，这种形式的AS子句被用于动态可载入 C 语言函数。字符串obj\_file是包含编译好的C函数的动态库文件的名称，它会由LOAD命令解析。字符串link\_symbol是该函数的链接符号，也就是该函数在 C 语言源代码中的名称。如果省略链接符号，它将被假定为要定义的 SQL 函数的名称。所有函数的C名称都必须不同，因此必须为重载的C函数给出不同的C名称（例如把参数类型作为C名称的一部分）。

在重复调用引用同一对象文件的CREATE FUNCTION时，对每个会话该文件只会被载入一次。要卸载并且重新装载该文件（可能是在开发期间），需要开始一个新会话。

## 重载

LightDB允许函数重载，也就是说同一个名称可以被用于多个不同的函数，只要它们具有可区分的输入参数类型。不管是否使用它，在有些用户不信任另一些用户的数据库中调用函数时，这种兼容性需要安全性的预防措施，请参考Section 10.3

如果两个函数具有相同的名称和输入参数类型，它们被认为相同（不考虑任何OUT参数）。因此这些声明会冲突：

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

具有不同参数类型列表的函数在创建时将不会被认为是冲突的，但是如果默认值被提供，在使用时它们有可能会冲突。例如，考虑

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

调用foo(10)将会失败，因为在要决定应该调用哪个函数时会有歧义。

## 注解

允许把完整的SQL类型语法用于声明一个函数的参数和返回值。不过，CREATE FUNCTION会抛弃带圆括号的类型修饰符（例如类型numeric的精度域）。例如CREATE FUNCTION foo (varchar(10)) ...和CREATE FUNCTION foo (varchar) ...完全一样。

在用CREATE OR REPLACE FUNCTION替换一个现有函数时，对于更改参数名是有限制的。不能更改已经分配给任何输入参数的名称（不过可以给之前没有名称的参数增加名称）。如果有多于一个输出参数，不能更改输出参数的名称，因为可能会改变描述函数结果的匿名组合类型的列名。这些限制是为了确保函数被替换时，已有的对该函数的调用不会停止工作。

如果一个被声明为STRICT的函数带有一个VARIADIC参数，会严格检查该可变数组作为一个整体是否为非空。如果该数组有空值元素，该函数仍将被调用。

## 示例

使用SQL函数对两个整数相加：

```
CREATE FUNCTION add(integer, integer) RETURNS integer
  AS 'select $1 + $2;'
  LANGUAGE SQL
  IMMUTABLE
  RETURNS NULL ON NULL INPUT;
```

在PL/pgSQL中，使用一个参数名称增加一个整数：

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
  BEGIN
    RETURN i + 1;
  END;
$$ LANGUAGE plpgsql;
```

返回一个包含多个输出参数的记录：

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
  AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
  LANGUAGE SQL;
```

```
SELECT * FROM dup(42);
```

你可以用更复杂的方式（用一个显式命名的组合类型）来做同样的事情：

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
  AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
  LANGUAGE SQL;
```

```
SELECT * FROM dup(42);
```

另一种返回多列的方法是使用一个TABLE函数：

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
  AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
  LANGUAGE SQL;
```

```
SELECT * FROM dup(42);
```

不过，TABLE函数与之前的例子不同，因为它实际返回了一个记录集合而不只是一个记录。

## 安全地编写 SECURITY DEFINER函数

因为一个SECURITY DEFINER函数会被以创建它的用户的特权来执行，需要小心地确保该函数不会被误用。为了安全，search\_path应该被设置为排除任何不可信用户可写的模式。这可以阻止恶意用户创建对象（例如表、函数以及操作符）来掩饰该函数所要用到的对象。在这方面特别重要的是临时表模式，默认情况下它会第一个被搜索并且通常对任何用户都是可写的。可以通过强制最后搜索临时模式来得到一种安全的布局。要这样做，把pg\_temp写成search\_path中的最后一项。这个函数展示了安全的用法：

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
  RETURNS BOOLEAN AS $$
```

```

DECLARE passed BOOLEAN;
BEGIN
    SELECT (pwd = $2) INTO passed
    FROM   pwds
    WHERE  username = $1;

    RETURN passed;
END;
$$ LANGUAGE plpgsql
SECURITY DEFINER
-- 设置一个安全的 search_path: 受信的模式, 然后是 'pg_temp'.
SET search_path = admin, pg_temp;

```

这个函数的目的是为了访问表admin.pwds。但是如果没有SET子句或者带有SET子句却只提到admin，该函数会变成创建一个名为pwds的临时表。

在LightDB 版本 8.3 之前，SET子句不可用，因而较老的函数可能包含相当复杂的逻辑来保存、设置以及恢复search\_path。对于这种目的，SET子句更容易。

另一点要记住的是默认情况下，会为新创建的函数给PUBLIC授予执行特权（详见Section 5.7）。你常常会希望把安全定义器函数的使用限制在某些用户中。要这样做，你必须收回默认的PUBLIC特权，然后选择性地授予执行特权。为了避免出现新函数能被所有人访问的时间窗口，应在一个事务中创建它并且设置特权。例如：

```

BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;

```

## 兼容性

SQL标准中定义了CREATE FUNCTION命令。LightDB实现可以以兼容的方式使用，但有许多扩展。相反，SQL标准指定了许多未在LightDB中实现的可选功能。

为了与其他一些数据库系统兼容，argmode 可以在argname之前或之后写入。但只有第一种方式符合标准。

对于参数默认值，SQL标准仅指定带有DEFAULT关键字的语法。=的语法是在T-SQL和Firebird中被使用的。

## 另见

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE

---

# CREATE GROUP

CREATE GROUP — 定义一个新的数据库角色

## Synopsis

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]
```

其中 option 可以是:

```
SUPERUSER | NOSUPERUSER  
| CREATEDB | NOCREATEDB  
| CREATEROLE | NOCREATEROLE  
| INHERIT | NOINHERIT  
| LOGIN | NOLOGIN  
| REPLICATION | NOREPLICATION  
| BYPASSRLS | NOBYPASSRLS  
| CONNECTION LIMIT connlimit  
| [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL  
| VALID UNTIL 'timestamp'  
| IN ROLE role_name [, ...]  
| IN GROUP role_name [, ...]  
| ROLE role_name [, ...]  
| ADMIN role_name [, ...]  
| USER role_name [, ...]  
| SYSID uid
```

## 描述

CREATE GROUP现在是 CREATE ROLE的一种别名。

## 兼容性

在 SQL 标准中没有CREATE GROUP语句。

## 另见

CREATE ROLE



---

# CREATE INDEX

CREATE INDEX — 定义一个新索引

## Synopsis

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] [schema_name.]name
] ON [ ONLY ] table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass
[ ( opclass_parameter = value [, ... ] ) ] ] [ ASC | DESC ] [ NULLS { FIRST |
LAST } ] [, ... ] )
    [ INCLUDE ( column_name [, ... ] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ LOCAL | GLOBAL ]
    [ LOGGING | NOLOGGING ]
    [ COMPRESS | NOCOMPRESS ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

## 描述

CREATE INDEX在指定关系的指定列上构建 一个索引，该关系可以是一个表或者一个物化视图。索引主要被用来提升 数据库性能（不过不当的使用会导致性能变差）。

索引的键域被指定为列名或者写在圆括号中的表达式。如果索引方法支持 多列索引，可以指定多个域。

一个索引域可以是一个从表行的一列或者更多列值进行计算的表达式。 这种特性可以被用来获得对基于基本数据某种变换的数据的快速访问。 例如，一个在upper(col)上计算的索引可以允许子句 WHERE upper(col) = 'JIM' 使用索引。

LightDB提供了索引方法 B-树、哈希、GiST、SP-GiST、GIN 以及 BRIN。用户也可以定义自己的索引 方法，但是相对较复杂。

当WHERE子句存在时，会创建一个 部分索引。部分索引只包含表中一部分行的项，通常索引这一部分会比表的其他部分更有用。例如，如果有一个表包含了 已付和未付订单，其中未付订单占了整个表的一小部分并且是经常被使用 的部分，可以通过只在这一部分上创建一个索引来改进性能。另一种可能 的应用是使用带有UNIQUE的 WHERE在表的一个子集上强制唯一性。更多的讨论 请见Section 11.8

WHERE子句中使用的表达式只能引用底层表的列，但 它可以引用所有列而不仅仅是被索引的列。当前， WHERE中也禁止使用子查询和聚集表达式。同样的 限制也适用于表达式索引中的表达式域。

所有在索引定义中使用的函数和操作符必须是“不可变的”， 就是说它们的结果必须仅依赖于它们的参数而不受外在因素（例如另 一个表的内容和当前的时间）的影响。这种限制确保了索引的行为是 良定的。要在一个索引表达式或者WHERE子句中 使用用户定义的函数，记住在创建函数时把它标记为不可变。

LightDB支持在创建索引后立即自动收集统计信息。 可以设置lightdb\_enable\_indexautoanalyze配置变量来决定是否自动收集统计信息。

## 参数

### UNIQUE

导致系统在索引被创建时（如果数据已经存在）或者加入数据时 检查重复值。会导致重复项的数据插入或者更新尝试将会产生一个错误。

当唯一索引被应用在分区边上时会有额外的限制，请参考CREATE TABLE。

### CONCURRENTLY

当使用了这个选项时，LightDB在构建索引时 不会取得任何会阻止该表上并发插入、更新或者删除的锁。而标准的索引 构建将会把表锁住以阻止对表的写（但不阻塞读），这种锁定会持续到索引创建完毕。在使用这个选项时有多个需要注意的地方 — 请见 Building Indexes Concurrently。

对于临时表，CREATE INDEX始终是非并发的，因为没有其他会话可以访问它们，并且创建非并发索引的成本更低。

### IF NOT EXISTS

如果一个同名关系已经存在则不要抛出错误。这种情况下会发出一个提示。 注意着并不保证现有的索引与将要创建的索引有任何相似。当 IF NOT EXISTS被指定时，需要指定索引名。

### INCLUDE

可选的INCLUDE子句指定一个列的列表，其中的列将被包括在索引中作为非键列。非键列不能作为索引扫描的条件，并且该索引所强制的任何唯一性或者排除约束都不会考虑它们。不过，只用索引的扫描可以返回非键列的内容而无需访问该索引的基表，因为在索引项中就能直接拿到它们。因此，非键列的增加允许查询使用只用索引的扫描，否则就无法使用。

保守地向索引中增加非键列是明智的，特别是很宽的列。如果一个索引元组超过索引类型允许的最大尺寸，数据插入将会失败。在任何情况下，非键列都会重复来自索引基表的数据并且让索引的尺寸膨胀，因此可能会拖慢搜索。此外，B树重复数据删除永远不会与具有非关键列的索引一起使用。

INCLUDE子句中列出的列不需要合适的操作符类，甚至数据类型没有为给定的访问方法定义操作符类的列都可以包括在这个子句中。

不支持把表达式作为被包括列，因为它们不能被用在只用索引的扫描中。

当前，有B-树、GiST和SP-GiST索引访问方法支持这一特性。 在这些索引中，INCLUDE子句中列出的列的值被包括在对应于堆元组的叶子元组中，但是不包括在用于树导航的上层索引项中。

### name

要创建的索引名称。这里不能包括模式名，因为索引总是被创建在其基表所在 的模式中。如果索引名称被省略，LightDB将基于基 表名称和被索引列名称选择一个合适的名称。

### ONLY

如果该表是分区表，指示不要在分区上递归创建索引。默认会递归创建索引。

### table\_name

要被索引的表的名称（可以被模式限定）。

### method

要使用的索引方法的名称。可以选择 btree、hash、gist以及 gin。 默认方法是btree。

column\_name

一个表列的名称。

expression

一个基于一个或者更多个表列的表达式。如语法中所示，表达式通常必须 被写在圆括号中。不过，如果该表达式是一个函数调用的形式，圆括号可 以被省略。

collation

要用于该索引的排序规则的名称。默认情况下，该索引使用被索引列 的排序规则或者被索引表达式的结果排序规则。当查询涉及到使用非 默认排序规则的表达式时，使用非默认排序规则的索引就能排上用场。

opclass

一个操作符类的名称。详见下文。

opclass\_parameter

运算符类参数的名称。详情请见下文。

ASC

指定上升排序（默认）。

DESC

指定下降排序。

NULLS FIRST

指定把空值排序在非空值前面。在指定DESC时， 这是默认行为。

NULLS LAST

指定把空值排序在非空值后面。在没有指定DESC时， 这是默认行为。

storage\_parameter

索引方法相关的存储参数的名称。详见 Index Storage Parameters。

tablespace\_name

在其中创建索引的表空间。如果没有指定，将会使用 default\_tablespace。或者对临时表上的索引使用 temp\_tablespaces。

predicate

部分索引的约束表达式。

local

Oracle兼容语法，不支持函数。有必要将配置更改为Oracle兼容模式。

global

Oracle兼容语法，不支持函数。有必要将配置更改为Oracle兼容模式。

COMPRESS LOGGING

为了兼容Oracle数据库语法，新增了四个非保留关键字COMPRESS, LOGGING, NOCOMPRESS, NOLOGGING。这四个关键字仅是语法糖，没有实现实际的功能。这四个关键字是可选的，如果必须要使用，请放在TABLESPACE关键字之前。

schema\_name

为了兼容Oracle数据库语法，CREATE INDEX语句支持为索引名称指定schema\_name。这个功能仅是语法糖。

## 索引存储参数

可选的WITH子句为索引指定存储参数。每一种索引方法都有自己的存储参数集合。B-树、哈希、GiST以及SP-GiST索引方法都接受这个参数：

fillfactor (integer)

索引的填充因子是一个百分数，它决定索引方法将尝试填充索引页面的充满程度。对于B-树，在初始的索引构建过程中，叶子页面会被填充至该百分数，当在索引右端扩展索引（增加新的最大键值）时也会这样处理。如果页面后来被完全填满，它们就会被分裂，导致在磁盘上的索引结构上产生碎片。B-树使用了默认的填充因子 90，但是也可以选择为 10 到 100 的任何整数值。

B-树索引还接受这个参数：

deduplicate\_items (boolean)

控制Section 60.4.4中描述的 B-树重复数据删除技术的使用。设置为 ON或OFF以启用或禁用优化（ON 和OFF的其他写法在Section 17.4中有介绍）。默认值为ON。

### Note

通过ALTER INDEX关闭deduplicate\_items 可以防止将来的插入触发重复数据删除，但本身不会使现有的发布列表元组使用标准的元组表示。

GiST还额外接受这个参数：

buffering (enum)

决定是否用Section 61.4.4中描述的缓冲构建技术来构建索引。当OFF时，缓冲被禁用，当ON时，它被启用，当AUTO时，它最初被禁用，但当索引大小达到effective\_cache\_size时，会立即启用。默认值为AUTO。请注意，如果排序构建是可能的，则除非指定了buffering=ON，否则将使用排序构建而不是缓冲构建。

GIN索引接受不同的参数：

fastupdate (boolean)

这个设置控制Section 62.4.4中描述的快速更新 技术的使用。它是一个布尔参数：ON启用快速更新，OFF禁用之。默认是 ON。

### Note

通过ALTER INDEX关闭fastupdate 会阻止未来的更新进入到待处理索引项列表中，但它不会自己处理之前的待处理项。可以使用VACUUM或者调用gin\_clean\_pending\_list确保处理完待处理列表的项。

gin\_pending\_list\_limit (integer)

自定义gin\_pending\_list\_limit参数。这个值 要以千字节来指定。

## 并发构建索引

创建索引可能会干扰数据库的常规操作。通常 LightDB 会锁住要被索引的表，让它不能被写入，并且用该表上的一次扫描来执行整个索引的构建。其他事务仍然可以读取表，但是如果它们尝试在该表上进行插入、更新或者删除，它们会被阻塞直到索引构建完成。如果系统是一个生产数据库，这可能会导致严重的后果。索引非常大的表可能会需要很多个小时，而且即使是较小的表，在构建索引过程中阻塞写入者一段时间在生产系统中也是不能接受的。

LightDB 支持构建索引时不阻塞写入。这种方法通过指定 CREATE INDEX 的 CONCURRENTLY 选项实现。当使用这个选项时，LightDB 必须执行该表的两次扫描，此外它必须等待所有有可能会修改或者使用该索引的事务终止。因此这种方法比起标准索引构建过程来说要做更多工作并且需要更多时间。不过，由于它允许在构建索引时继续普通操作，这种方式对于在生产环境中增加新索引很有用。当然，由索引创建带来的额外 CPU 和 I/O 开销可能会拖慢其他操作。

在并发索引构建中，索引实际上在一个事务中被录入到系统目录，然后在两个事务中发生两次表扫描。在每一次表扫描之前，索引构建必须等待已经修改了表的现有事务终止。在第二次扫描之后，索引构建必须等待任何持有早于第二次扫描的快照（见 Chapter 13 的事务终止，如果所涉及的索引是部分的或具有非简单列引用的列，则还包括在其他表上构建并发索引的任何阶段所使用的事务。然后该索引最终能被标记为准备好使用，并且 CREATE INDEX 命令终止。不过即便那样，该索引也不是立刻可以用于查询：在最坏的情况下，只要早于索引构建开始时存在的事务存在，该索引就无法使用。

如果在扫描表示出现问题，例如死锁或者唯一索引中的唯一性被违背，CREATE INDEX 将会失败，但留下一个“不可用”的索引。这个索引会被查询所忽略，因为它可能不完整。不过它仍将消耗更新开销。ltsql 的 \d 命令将把这类索引报告为 INVALID：

```
lightdb@postgres=# \d tab
      Table "public.tab"
  Column | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
   col   | integer |           |          |
Indexes:
  "idx" btree (col) INVALID
```

这种情况下推荐的恢复方法是删除该索引并且尝试再次执行 CREATE INDEX CONCURRENTLY。（另一种可能性是用 REINDEX INDEX CONCURRENTLY 重建该索引）。

并发构建一个唯一索引时需要注意的另一点是，当第二次表扫描开始时，唯一约束已经被强制在其他事务上。这意味着在该索引变得可用之前，其他查询中可能就会报告该约束被违背，或者甚至在索引构建最终失败的情况中也是这样。还有，如果在第二次扫描时发生失败，“无效的”索引也会继续强制它的唯一性约束。

表达式索引和部分索引的并发构建也被支持。在这些表达式计算过程中发生的错误可能导致和上述唯一约束违背类似的行为。

常规索引构建允许在同一个表上同时构建其他常规索引，但是在一个表上同时只能有一个并发索引构建发生。在两种情况下，在索引被构建时不允许表的模式修改。另一个不同是，一个常规 CREATE INDEX 命令可以在一个事务块中执行，但是 CREATE INDEX CONCURRENTLY 不行。

当前不支持在分区表上并发生成索引。然而，你可以在每个分区上单独的并发构建索引，然后最终以非并发的方式创建分区索引，以减少对分区表的写入被锁定的时间。在这种情况下，生成分区索引仅是元数据操作。

## 注解

关于索引何时能被使用、何时不被使用以及什么情况下它们有用的信息请见 Chapter 11

当前，只有 B-树、GiST、GIN 索引方法支持多键列索引。是否可以有多个键列与是否可以将INCLUDE列添加到索引无关。索引最多可以有32列，包括INCLUDE列。（可以在构建LightDB时修改这个限制）。当前只有 B-树支持唯一索引。

为索引的每一列可以指定一个带可选参数的操作符类。该操作符类标识要被该索引用于该列的操作符。例如，一个四字节整数上的 B-树索引 会使用int4\_ops类。这个操作符类包括了用于四字节整数的比较函数。实际上，通常列数据类型的默认操作符类就足够了。对某些数据类型指定操作符类的主要原因是，可能会有多于一种有意义的顺序。例如，我们可能想用绝对值或者实数部分对复数类型排序。我们可以通过为该数据类型定义两个操作符类来做到，并且在创建索引时选择其中合适的类。更多有关操作符类的信息请见Section 11.10以及Section 36.16

当在一个分区表上调用CREATE INDEX时，默认的行为是递归到所有的分区上以确保它们都具有匹配的索引。每一个分区首先会被检查是否有一个等效的索引存在，如果有则该索引将被挂接为被创建索引的一个分区索引，而被创建的索引将成为其父索引。如果不存在匹配的索引，则会创建一个新的索引并且自动进行挂接。如果命令中没有指定索引名称，每个分区中的新索引的名称将被自动决定。如果指定了ONLY选项，则不会进行递归，并且该索引会被标记为无效（一旦所有的分区都得到该索引，ALTER INDEX ... ATTACH PARTITION可以将该索引标记为有效）。不过，要注意不管是否指定这一选项，未来使用CREATE TABLE ... PARTITION OF创建的任何分区将自动有一个匹配的索引，不管有没有指定ONLY。

对于支持有序扫描的索引方法（当前只有 B-树），可以指定可选子句ASC、DESC、NULLS FIRST以及NULLS LAST 来修改索引的排序顺序。由于一个有序索引能前向或者反向扫描，通常创建一个单列DESC索引没什么用处——一个常规索引已经提供了排序顺序。这些选项的价值是可以创建多列索引，让它的排序顺序匹配有混合排序要求的查询，例如SELECT ... ORDER BY x ASC, y DESC。如果你想要在依靠索引避免排序步骤的查询中支持“空值排序低”这种行为，NULLS选项就能派上用场，默认的行为是“空值排序高”。

系统定期收集表中所有列的统计信息。新创建的非表达式索引可以立即使用这些统计数据来确定索引的有用性。对于新的表达式索引，需要运行 ANALYZE 或等待autovacuum守护程序分析表以生成这些索引的统计信息。

对于大多数索引方法，索引的创建速度取决于 maintenance\_work\_mem的设置。较大的值将会减少索引创建所需的时间，当然不要把它设置得超过实际可用的内存量（那会迫使机器进行交换）。

LightDB可以在构建索引时利用多个CPU以更快地处理表行。这种特性被称为并行索引构建。对于支持并行构建索引的索引方法（当前只有B-树），maintenance\_work\_mem指定每次索引构建操作整体可用的最大内存量，而不管启动了多少工作者进程。一般来说，一个代价模型（如果有）自动判断应该请求多少工作者进程。

增加maintenance\_work\_mem可以让并行索引构建受益，而等效的串行索引构建将无法受益或者得到很小的益处。注意maintenance\_work\_mem可能会影响请求的工作者进程的数量，因为并行工作者必须在总的maintenance\_work\_mem预算中占有至少32MB的份额。还必须有32MB的份额留给领袖进程。增加max\_parallel\_maintenance\_workers可以允许使用更多的工作者，这将降低索引创建所需的时间，只要索引构建不是I/O密集型的。当然，还需要有足够的CPU计算能力，否则工作者们会闲置。

通过ALTER TABLE为parallel\_workers设置一个值直接控制着CREATE INDEX会对表请求多少并行工作者进程。这会完全绕过代价模型，并且防止maintenance\_work\_mem对请求多少并行工作者产生影响。通过ALTER TABLE将parallel\_workers设置为0将禁用所有情况下的并行索引构建。

### Tip

在把parallel\_workers用于调优一次索引构建之后，你可能想要重置parallel\_workers。这可以避免对查询计划的无意更改，因为parallel\_workers影响所有的并行表扫描。

虽然带有CONCURRENTLY选项的CREATE INDEX支持并行构建并且没有特殊的限制，但只有第一次表扫描会实际以并行方式执行。

使用DROP INDEX可以移除一个索引。

与任何长时间运行的事务一样，表上的CREATE INDEX会影响其他表上的并发VACUUM可以删除哪些元组。除此之外，对于非部分且不索引任何表达式的索引，还可以使用CONCURRENTLY选项进行操作。

以前的LightDB发行也有一种 R-树 索引方法。这种方法已经被移除，因为它比起 GiST 方法来说没有什么明显的优势。如果指定了USING rtree，CREATE INDEX 将会把它解释为USING gist，以便把旧的数据库转换成 GiST。

## 示例

在表films中的列title上创建一个 B-树索引：

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

要在表films的列title上创建一个唯一的B-树索引并且包括列director和rating：

```
CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);
```

要创建禁用重复数据删除的 B 树索引：

```
CREATE INDEX title_idx ON films (title) WITH (deduplicate_items = off);
```

在表达式lower(title)上创建一个索引来允许高效的大小写无关搜索：

```
CREATE INDEX ON films ((lower(title)));
```

（在这个例子中我们选择省略索引名称，这样系统会选择这个名字，通常是films\_lower\_idx）。

创建一个具有非默认排序规则的索引：

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

创建一个具有非默认空值排序顺序的索引：

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

创建一个具有非默认填充因子的索引：

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

创建一个禁用快速更新的GIN索引：

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off);
```

在表films中的列code上创建一个索引并且把索引放在表空间indexspace中：

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexespace;
```

在一个点属性上创建一个 GiST 索引，这样我们可以在转换函数的结果 上有效地使用 box 操作符：

```
CREATE INDEX pointloc
  ON points USING gist (box(location, location));
SELECT * FROM points
  WHERE box(location, location) && '(0,0),(1,1)::box;
```

创建一个表而不排斥对表的写操作：

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

创建本地或全局索引

```
CREATE INDEX index_local ON sales_table (quantity) LOCAL;
CREATE INDEX index_global ON sales_table (quantity) GLOBAL;
```

使用COMPRESS/NOCOMPRESS/LOGGING/NOLOGGING选项来创建一个索引。

```
create table comp_log_test(id int);
create index idx_id on comp_log_test(id) nologging compress tablespace
  pg_default;
```

在创建索引时使用schema\_name前缀。

```
create table t1(id int);
create index hahaha.index_t1_id on t1(id);
```

## 兼容性

CREATE INDEX是一种 LightDB的语言扩展。在 SQL 标准中 没有对于索引的规定。

## 另见

ALTER INDEX, DROP INDEX, REINDEX



---

# CREATE LANGUAGE

CREATE LANGUAGE — 定义一种新的过程语言

## Synopsis

```
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
```

## 描述

CREATE LANGUAGE为一个 LightDB数据库注册一种新的 过程语言。接着，可以用这种新语言定义函数和存储过程。

CREATE LANGUAGE实际上把该语言名称与 负责执行用该语言编写的函数的处理器函数关联在一起。有关语言处理器的 更多信息可以参考Chapter 52

CREATE OR REPLACE LANGUAGE将创建 或者替换一种现有的定义。如果该语言已经存在，其参数会被根据命令更新。但 该语言的拥有关系和权限设置不会更改，并且任何已有的用该语言编写的 函数仍然被假定有效。

必须具有LightDB超级用户权限才能注册新语言或更改现有语言的参数。 然而，一旦语言被创建，将它的所有权分配给非超级用户是有效的，然后他可以删除它、更改其权限、重命名它或 将其分配给新的所有者。（但是，不要将底层 C 函数的所有权分配给非超级用户；这会为该用户创建权限提升路径。）

不提供任何处理函数的CREATE LANGUAGE形式已过时。 为了向后兼容旧的转储文件，它被解释为CREATE EXTENSION。 如果语言被打包到同名的扩展中，这将起作用，这是设置过程语言的传统方式。

## 参数

TRUSTED

TRUSTED指定该语言不会授予用户不该具有的数据访问。如果在注册语言时这个关键词被省略，只有具有 LightDB超级用户特权的用户才能 使用该语言创建新函数。

PROCEDURAL

这是一个噪声词。

name

新过程语言的名称。该名称必须在该数据库的语言中唯一。

为了向后兼容，名称可以用单引号括起来 quotes.

HANDLER call\_handler

call\_handler 是一个之前注册的函数的名称，它将被调用来执行该过程语言的函数。一种过程语言的调用处理器必须以一种编译型语言（如 C）编写并且 具有版本 1 的调用约定，它必须在 LightDB内注册为一个没有 参数并且返回language\_handler类型的函数。 language\_handler是一种占位符类型， 它被用来 标识该函数为一个调用处理器。

INLINE inline\_handler

inline\_handler 是一个之前注册的函数的名称，它将被调用来执行一个该语言的匿名代码块（DO命令）。如果没有指定 inline\_handler函数，则 该语言不支持匿名代码块。该处理器函数必须接受一个internal 类型的参数，该参数将是DO命令的内部表示，而且它通常 将返回void。该处理器的返回值会被忽略。

VALIDATOR valfunction

valfunction is the 是一个之前注册的函数的名称，当一个该语言的新函数被创建时会调用该函数来 验证新函数。如果没有指定验证器函数，那么一个新函数被创建时不会被检查。验证器函数必须接受一个oid类型的参数，它将是 要被创建的 函数的 OID，而且它通常将返回void。

一个验证器函数通常会检查函数体中的语法正确性，但是它也能查看函数的其他 属性，例如该语言能否处理特定的参数类型。为了发出一个错误，验证器函数应该 使用ereport()函数。验证器函数的返回值会被忽略。

## 注解

使用DROP LANGUAGE删除过程语言。

系统目录pg\_language（见Section 48.37）记录着有关当前已安装的语言的信息。还有，ltsql命令\dL列出已安装的语言。

要以一种过程语言创建函数，用户必须具有对于该语言的 USAGE特权。默认情况下，对于可信语言，USAGE被授予给PUBLIC（即所有人）。如果需要可以将它收回。

过程语言对于单个数据库来说是本地的。但是，一种语言可以被安装在 template1数据库中，这会导致它在所有后续创建的 数据库中自动变得可用。

## 示例

创建新的过程语言的最小顺序是：

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
AS '$libdir/plsample'
LANGUAGE C;
CREATE LANGUAGE plsample
HANDLER plsample_call_handler;
```

通常这会写在扩展的创建脚本中，用户会这样做来安装扩展：

```
CREATE EXTENSION plsample;
```

## 兼容性

CREATE LANGUAGE是一种 LightDB扩展。

## 另见

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE

---

# CREATE MATERIALIZED VIEW

CREATE MATERIALIZED VIEW — 定义一个新的物化视图

## Synopsis

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
  [ (column_name [, ...] ) ]
  [ USING method ]
  [ WITH ( storage_parameter [= value] [, ... ] ) ]
  [ TABLESPACE tablespace_name ]
AS query
[ WITH [ NO ] DATA ]
```

## 描述

CREATE MATERIALIZED VIEW定义一个查询的物化视图。 在该命令被发出时，查询会被执行并且被用来填充该视图（除非使用了 WITH NO DATA），并且后来可能会用 REFRESH MATERIALIZED VIEW进行刷新。

CREATE MATERIALIZED VIEW类似于 CREATE TABLE AS，不过它还会记住被用来初始化该视图的查询， 这样它可以在后来被命令刷新。一个物化视图有很多和表相同的属性，但是不支持临时物化视图。

## 参数

IF NOT EXISTS

如果已经存在一个同名的物化视图时不要抛出错误。这种情况下会发出一个 提示。注意这不保证现有的物化视图与即将创建的物化视图相似。

table\_name

要创建的物化视图的名称（可以被模式限定）。

column\_name

新物化视图中的一个列名。如果没有提供列名，会从查询的输出列名来得到。

USING method

此可选子句指定用于存储新具体化视图内容的表访问方法；该方法需要是TABLE类型的访问方法。 详细信息请参考 Chapter 57 如果未指定此选项，则为新具体化视图选择默认表访问方法。 详细信息请参考 default\_table\_access\_method 。

WITH ( storage\_parameter [= value] [, ... ] )

这个子句为新的物化视图指定可选的存储参数，详见CREATE TABLE的 Storage Parameters。所有CREATE TABLE支持的参数CREATE MATERIALIZED VIEW也支持。 详见CREATE TABLE。

TABLESPACE tablespace\_name

tablespace\_name是 要把新物化视图创建在其中的表空间的名称。如果没有指定， 将查阅default\_tablespace。

query

一个SELECT、TABLE 或者VALUES命令。这个查询将在一个安全受限的操作中运行。特别地，对本身会创建临时表的函数的调用将会失败。

WITH [ NO ] DATA

这个子句指定物化视图是否在创建时被填充。如果不是，该物化视图将被标记为不可扫描并且在REFRESH MATERIALIZED VIEW被使用前不能被查询。

## 兼容性

CREATE MATERIALIZED VIEW是一种 LightDB扩展。

## 另见

ALTER MATERIALIZED VIEW, CREATE TABLE AS, CREATE VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

---

# CREATE OPERATOR

CREATE OPERATOR — 定义一个新的操作符

## Synopsis

```
CREATE OPERATOR name (  
    {FUNCTION|PROCEDURE} = function_name  
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]  
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]  
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]  
    [, HASHES ] [, MERGES ]  
)
```

## 描述

CREATE OPERATOR定义一个新的操作符 `name`。定义操作符 的用户会成为该操作符的拥有者。如果给出一个模式名，该操作符将被创建 在指定的模式中。否则它会被创建在当前模式中。

操作符名称是最多NAMEDATALEN-1（默认为 63） 个字符的序列，这些字符可以是：

+ - \* / < > = ~ ! @ # % ^ & | ` ?

对名称的选择有一些限制：

- -- and /\*不能出现在 操作符名称中，因为它们会被当做一段注释的开始。
- 多字符操作符名称不能以+或者- 结束，除非该名称也包含至少一个下列字符：

~ ! @ # % ^ & | ` ?

例如，@-是一个被允许的操作符名称，而 \*-不是。这种限制允许 LightDB解析 SQL 兼容的命令 而无需记号之间的空格。

- 将=>用作一个操作符名称已经不被推荐。在未来的发行中 可能会被禁用。

在输入时!=会被映射为<>， 因此这两个名字总是等效的。

对于二元运算符，必须同时定义LEFTARG和RIGHTARG。 对于前缀运算符，仅应定义RIGHTARG。 `function_name`函数 必须在之前已经用CREATE FUNCTION定义好， 并且必须被定义为接受正确数量的指定类型的参数。

### Note

右一元运算符（也称为后缀）已弃用，并将在LightDB版本14中删除。

`function_name`函数必须事先使用CREATE FUNCTION定义，并且必须定义为接受所指示类型的正确数量的参数（一个或两个）。

在CREATE OPERATOR的语法中，关键词FUNCTION和PROCEDURE是等效的，但不管哪种情况下被引用的函数都必须是一个函数而不是过程。这里对关键词PROCEDURE的是用是有历史原因的，现在已经被废弃。

其他子句指定可选的操作符优化子句。它们的含义在 Section 36.15 中有详细描述。

要创建一个操作符，必须具有参数类型和返回类型上的 `USAGE` 特权，以及底层函数上的 `EXECUTE` 特权。如果指定了一个 交换子或者求反器操作符，必须拥有这些操作符。

## 参数

`name`

要定义的操作符的名称。允许使用的字符请见上文。名称可以被模式 限定，例如 `CREATE OPERATOR myschema.+ (...)`。如果没有被模式限定，该操作符将被创建在当前模式中。如果两个同一模式 中的操作符在不同的数据类型上操作，它们可以具有相同的名称。这被称为重载。

`function_name`

用来实现这个操作符的函数。

`left_type`

这个操作符的左操作数（如果有）的数据类型。忽略这个选项 可以表示一个前缀操作符。

`right_type`

这个操作符的右操作数（如果有）的数据类型。

`com_op`

这个操作符的交换子。

`neg_op`

这个操作符的求反器。

`res_proc`

用于这个操作符的限制选择度估计函数。

`join_proc`

用于这个操作符的连接选择度估算函数。

`HASHES`

表示这个操作符可以支持哈希连接。

`MERGES`

表示这个操作符可以支持归并连接。

要在 `com_op` 或者其他可选参数中给出一个模式限定的操作符名称， 请使用 `OPERATOR()` 语法，例如：

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

## 注解

进一步的信息可参考 Section 36.14

无法在CREATE OPERATOR中指定一个操作符的 词法优先级，因为解析器的优先级行为是硬写在代码中的。详见 Section 4.1.6

废弃的选项SORT1、SORT2、 LTCMP以及GTCMP以前被用来指定与支持 归并连接的操作符相关的排序操作符的名称。现在不再需要它们了，因为 相关操作符的信息可以在 B-树的操作符族中找到。如果给出了这些选项 之一，它会被忽略（除非是为了隐式设置MERGES为真）。

使用DROP OPERATOR从数据库中删除用户定义的操作符。 使用ALTER OPERATOR修改数据库中的操作符。

## 示例

下面的命令为数据类型box定义了一种新的操作符—面积相等：

```
CREATE OPERATOR === (  
    LEFTARG = box,  
    RIGHTARG = box,  
    FUNCTION = area_equal_function,  
    COMMUTATOR = ===,  
    NEGATOR = !==,  
    RESTRICT = area_restriction_function,  
    JOIN = area_join_function,  
    HASHES, MERGES  
);
```

## 兼容性

CREATE OPERATOR是一种 LightDB扩展。在 SQL 标准中没有用户定义操作符的规定。

## 另见

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR

---

# CREATE OPERATOR CLASS

CREATE OPERATOR CLASS — 定义一个新的操作符类

## Synopsis

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  { OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ FOR
  SEARCH | FOR ORDER BY sort_family_name ]
  | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name
  ( argument_type [, ...] )
  | STORAGE storage_type
  } [, ... ]
```

## 描述

CREATE OPERATOR CLASS创建新的操作符类。一个操作符类定义一种特殊的数据类型如何被用于一个索引。操作符类指定为该数据类型和索引方法扮演特殊角色或者“策略”的操作符。操作符类还指定当该操作符类被选择用于一个索引列时，索引方法要使用的支持函数。操作符类所使用的所有操作符和函数必须在操作符类被创建之前被定义好。

如果给出了一个模式名称，那么该操作符类会被创建在指定模式中。否则，它会被创建在当前模式中。同一模式中的两个操作符类只有在被用于不同的索引方法时才可以具有相同的名称。

定义操作符类的用户将成为其拥有者。当前，创建用户必须是超级用户（做出这种限制是因为错误的操作符类定义会让服务器混淆甚至崩溃）。

CREATE OPERATOR CLASS当前不会检查操作符类定义是否包括该索引方法所要求的所有操作符和函数，也不会检查这些操作符和函数是否构成一个一致的集合。定义一个合法的操作符类是用户的责任。

相关的操作符类可以被组成操作符族。要把一个新的操作符类加入到一个现有的族中，可以在CREATE OPERATOR CLASS中指定FAMILY选项。如果没有这个选项，新的类会被放到一个同名的族中（如果族不存在会创建之）。

进一步的信息可参考Section 36.16

## 参数

name

要创建的操作符类的名称。该名称可以被模式限定。

DEFAULT

如果存在，该操作符类将成为其数据类型的默认操作符类。对一种特定的数据类型和索引方法至多有一个默认操作符类。

data\_type

这个操作符类所用于的列数据类型。

index\_method

这个操作符类所用于的索引方法的名称。



family\_name

要把这个操作符类加入其中的已有操作符族的名称。如果没有指定，将使用一个同名操作符族（如果还不存在则创建之）。

strategy\_number

用于一个与该操作符类相关联的操作符的索引方法策略号。

operator\_name

一个与该操作符类相关联的操作符的名称（可以被模式限定）。

op\_type

在一个OPERATOR子句中，这表示该操作符的操作数数据类型，或者用NONE来表示一个前缀操作符。在操作数数据类型与该操作符的数据类型相同的一般情况下，操作数的数据类型可以被省略。

在一个FUNCTION子句中，这表示该函数要支持的操作数数据类型，如果它与该函数的输入数据类型（对于 B-树比较函数和哈希函数）或者操作符类的数据类型（对于 B-树排序支持函数和与B-树相同的的图像函数以及所有GiST、GIN操作符类中的函数）不同。这些默认值是正确的，并且 op\_type因此不必在FUNCTION子句中被指定，对于 B-树排序支持函数的情况来说，这表示跨数据类型比较。

sort\_family\_name

一个现有btree操作符族的名称（可以是模式限定的），它描述与一种排序操作符相关联的排序顺序。

如果FOR SEARCH和FOR ORDER BY都没有被指定，那么FOR SEARCH是默认值。

support\_number

用于一个与该操作符类相关联的函数的索引方法支持函数编号。

function\_name

一个用于该操作符类的索引方法支持函数的函数名称（可以是模式限定的）。

argument\_type

该函数的参数数据类型。

storage\_type

实际存储在索引中的数据类型。通常这和列数据类型相同，但是有些索引方法（当前有 GiST、GIN、SP-GiST和 BRIN）允许它们不同。除非索引方法允许使用不同的类型，STORAGE子句必须被省略。如果data\_type列被指定为anyarray，那么storage\_type可以被声明为anyelement以指示索引条目是属于为每个特定索引创建的实际数组类型的元素类型的成员。

OPERATOR、FUNCTION和STORAGE子句可以以任何顺序出现。

## 注解

因为索引机制在使用函数之前不检查它们的权限，将一个函数或者操作符包括在一个操作符类中相当于在其上授予公共执行权限。这对操作符类中很有用的函数来说通常不成问题。

操作符不应该用 SQL 函数定义。SQL 函数很有可能会被内联到调用查询中，这会妨碍优化器识别该查询匹配一个索引。

在LightDB 8.4 之前， OPERATOR子句可以包括一个RECHECK选项。现在 已经不再支持，因为一个索引操作符是否为“有损的”现在是在运行时实时决定的。这允许在一个操作符可能是或者可能不是有损的情况下有效地处理。

## 示例

下面的例子为数据类型\_int4 (int4数组) 定义了一个 GiST 索引操作符。完整的例子请见 intarray模块。

```
CREATE OPERATOR CLASS gist__int_ops
  DEFAULT FOR TYPE _int4 USING gist AS
  OPERATOR          3      &&,
  OPERATOR          6      = (anyarray, anyarray),
  OPERATOR          7      @>,
  OPERATOR          8      <@,
  OPERATOR          20     @@ (_int4, query_int),
  FUNCTION          1      g_int_consistent (internal, _int4, smallint,
oid, internal),
  FUNCTION          2      g_int_union (internal, internal),
  FUNCTION          3      g_int_compress (internal),
  FUNCTION          4      g_int_decompress (internal),
  FUNCTION          5      g_int_penalty (internal, internal, internal),
  FUNCTION          6      g_int_picksplit (internal, internal),
  FUNCTION          7      g_int_same (_int4, _int4, internal);
```

## 兼容性

CREATE OPERATOR CLASS是一种 LightDB扩展。在 SQL 标准中没有 CREATE OPERATOR CLASS语句。

## 另见

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY

---

# CREATE OPERATOR FAMILY

CREATE OPERATOR FAMILY — 定义一个新的操作符族

## Synopsis

```
CREATE OPERATOR FAMILY name USING index_method
```

## 描述

CREATE OPERATOR FAMILY创建一个新的操作符族。一个操作符族定义一个相关操作符类组成的集合，并且可能还包含一些额外的、与这些操作符类兼容但对于任何个体索引的功能不是至关重要的操作符和支持函数（对索引至关重要的操作符和函数应该被分组在相关的操作符类中，而不是“松散地”在操作符中。通常，单一数据类型操作符被限制在操作符类中，而跨数据类型操作符可以松散地存在于一个包含用于两种数据类型的操作符类的操作符族中）。

新的操作符族初始时空。应该通过发出后续的 CREATE OPERATOR CLASS命令来增加包含在其中的操作符类，还可以用可选的 ALTER OPERATOR FAMILY命令来增加“松散的”操作符和它们对应的支持函数。

如果给出一个模式名称，该操作符族会被创建在指定的模式中。否则，它会被创建在当前模式中。只有当同一个模式中的两个操作符族是用于不同的索引方法时，它们才能拥有相同的名字。

定义一个操作符族的用户将成为它的拥有者。当前，创建用户必须是超级用户（做出这样的限制是因为错误的操作符族会让服务器混淆甚至崩溃）。

进一步的信息可以参考Section 36.16

## 参数

name

要创建的操作符族的名称。该名称可以被模式限定。

index\_method

这个操作符族要用于的索引方法的名称。

## 兼容性

CREATE OPERATOR FAMILY是一种 LightDB扩展。在 SQL 标准中没有 CREATE OPERATOR FAMILY语句。

## 另见

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

---

# CREATE POLICY

CREATE POLICY — 为一个表定义一条新的行级安全性策略

## Synopsis

```
CREATE POLICY name ON table_name
  [ AS { PERMISSIVE | RESTRICTIVE } ]
  [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
  [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
  [ USING ( using_expression ) ]
  [ WITH CHECK ( check_expression ) ]
```

## 描述

CREATE POLICY为一个表定义一条行级 安全性策略。注意为了应用已被创建的策略，在表上必须启用行级安全性（使用ALTER TABLE ... ENABLE ROW LEVEL SECURITY）。

一条策略授予权限以选择、插入、更新或者删除匹配相关策略表达式的行。 现有的表行会按照USING中指定的表达式进行检查， 而将要通过INSERT或UPDATE创建 的新行会按照WITH CHECK中指定的表达式进行检查。 当USING表达式对于一个给定行返回真时，该行对用户 可见，而返回假或空时该行不可见。当一个WITH CHECK 表达式对一行返回真时，该行会被插入或更新，而如果返回假或空时会发生 一个错误。

对于INSERT和 UPDATE语句，在BEFORE 触发器被引发后并且在任何数据修改真正发生之前，WITH CHECK表达式会被强制。因此， 一个BEFORE ROW触发器可以修改要被插入的数据， 从而影响安全性策略检查的结果。WITH CHECK表达式 在任何其他约束之前被强制。

策略名称是针对每个表的。因此，一个策略名称可以被用于很多个不同的表 并且对于不同的表呈现适合于该表的定义。

策略可以被应用于特定的命令或者特定的角色。除非特别指定，新创建的策略 的默认行为是适用于所有命令和角色。多个策略可以应用于单个命令，更多细节请见下文。Table 27总结了不同类型的策略如何应用于特定的命令。

对同时具有USING和WITH CHECK 表达式（ALL和UPDATE）的策略， 如果没有定义WITH CHECK表达式，那么 USING表达式将被用于决定哪些行可见（普通 USING情况）以及允许哪些新行被增加（ WITH CHECK情况）。

如果为一个表启用了行级安全性但是没有适用的策略存在，将假定作为一种 “默认否定” 策略，这样任何行都不可见也不可更新。

## 参数

name

要创建的策略的名称。这必须和该表上已有的任何其他策略名称相区分。

table\_name

该策略适用的表的名称（可以被模式限定）。

PERMISSIVE

指定策略被创建为宽容性策略。适用于一个给定查询的所有宽容性策略将被使用布尔“OR”操作符组合在一起。通过创建宽容性策略，管理员可以在能被访问的记录集合中进行增加。策略默认是宽容性的。

## RESTRICTIVE

指定策略被创建为限制性策略。适用于一个给定查询的所有限制性策略将被使用布尔“AND”操作符组合在一起。通过创建限制性策略，管理员可以减少能被访问的记录集合，因为每一条记录都必须通过所有的限制性策略。

注意在限制性策略真正能发挥作用减少访问之前，需要至少一条宽容性策略来授予对记录的访问。如果只有限制性策略存在，则没有记录能被访问。当宽容性和限制性策略混合存在时，只有当一个记录能通过至少一条宽容性策略以及所有的限制性策略时，该记录才是可访问的。

## command

该策略适用的命令。合法的选项是 ALL、SELECT、INSERT、UPDATE 以及DELETE。ALL为默认。有关这些策略如何被应用的 细节见下文。

## role\_name

该策略适用的角色。默认是PUBLIC，它将把策略应用到所有的角色。

## using\_expression

任意的SQL条件表达式（返回 boolean）。该条件表达式不能包含任何聚集或者窗口 函数。如果行级安全性被启用，这个表达式将被增加到引用该表的查询。让这个表达式返回真的行将可见。让这个表达式返回假或者空的任何行 将对用户不可见（在SELECT中）并且将对修改不可用（在UPDATE或DELETE中）。这类行 会被悄悄地禁止而不会报告错误。

## check\_expression

任意的SQL条件表达式（返回 boolean）。该条件表达式不能包含任何聚集或者窗口 函数。如果行级安全性被启用，这个表达式将被用在表上的 INSERT以及 UPDATE查询中。只有让该表达式计算为真 的行才被允许。如果任何被插入的记录或者跟新后的记录导致该表达式计 算为假或者空，则会抛出一个错误。注意 check\_expression 是根据行的新内容而不是原始内容计算的。

## 针对每种命令的策略

## ALL

为一条策略使用ALL表示它将适用于所有命令， 不管命令的类型如何。如果存在一条ALL策略 以及更多特定的策略，则ALL策略和那些策略 会被应用。此外， ALL策略将同时适用于一个查询的选择端和修 改端，如果只定义了一个USING表达式则将 该USING表达式用于两种情况。

例如，如果发出一个UPDATE，那么 ALL策略将同时影响UPDATE 能更新哪些行（应用USING表达式）以及更新后 的行是否被允许加入到表中（如果定义了WITH CHECK 表达式，则应用之；否则使用USING表达式）。 如果一条INSERT 或者UPDATE命令尝试增加行到表中， 但行没有通过ALL策略的 WITH CHECK表达式，则整个语句将会中断。

## SELECT

对一条策略使用SELECT表示它将适用于 SELECT查询，并且无论何时都要求该约束所在的关系上 的SELECT权限。其结果是在一次 SELECT查询期间，只有该关系中那些通过了SELECT策略的记录才将被返回，并且查询要求 SELECT权限，例如 UPDATE也将只能看到那些 SELECT策略允许的行。一条 SELECT策略不能具有WITH CHECK表达式，因为它只适用于正在从关系中检索记录的情况。

## INSERT

为一条策略使用INSERT表示它适用于 INSERT命令。没有通过这种策略的正在被插入的行会导致策略违背错误，并且整个INSERT命令将会中止。 一条INSERT策略不能具有USING表达式，因为它只适用于正在向关系增加记录的情况。

注意在带有ON CONFLICT DO UPDATE的INSERT中，只有对通过 INSERT路径追加到关系的行才会检查 INSERT策略的WITH CHECK 表达式。

UPDATE

对策略使用UPDATE 意味着它将应用于UPDATE、SELECT FOR UPDATE和SELECT FOR SHARE 命令，还有INSERT 命令的辅助性的ON CONFLICT DO UPDATE 子句。由于UPDATE 需要提取现有的记录并且用新修改的记录代替，故UPDATE 策略接受USING 表达式和WITH CHECK 表达式。USING 表达式决定UPDATE 命令将能看到哪些要对其操作的记录，而WITH CHECK 表达式定义哪些被修改的行允许存回到关系中。

任何更新后的值无法通过WITH CHECK表达式的行 将会导致错误，并且整个命令将被中止。如果只指定了一个 USING子句，那么该子句将被用于 USING和WITH CHECK两种情况。

典型地，UPDATE命令也需要从待更新关系中的列读取数据（例如在WHERE子句、RETURNING子句或在SET子句右侧的表达式中）。这种情况下，正被更新的关系上也需要SELECT权限，并且除了UPDATE策略外，也要应用适当的SELECT或者ALL策略。这样，除由UPDATE或ALL策略授权更新行之外，通过SELECT或ALL策略用也必须能访问正被更新的行。

当INSERT命令附加了ON CONFLICT DO UPDATE子句时，如果采用UPDATE路径，先以任何UPDATE策略的USING表达式检查待更新的行，然后以WITH CHECK表达式检查新修改的行。但要注意的是，不同于单独的UPDATE命令，如果现有的行不能通过USING表达式检查，则抛出错误（UPDATE路径永不会静默地避免）。

DELETE

为一条策略使用DELETE表示它适用于 DELETE命令。只有通过这条策略的行才将能被DELETE命令所看到。如果有行不能通过该 DELETE策略的USING表达式，则 它们可以通过SELECT看到但不能被删除。

大多数情况下，DELETE命令也需要从其所删除的关系中的列读取数据（例如在WHERE子句或RETURNING子句中）。这种情况下，在该关系上也需要SELECT权限，并且除了DELETE策略，也要应用适当的SELECT或ALL策略。这样，除由DELETE或ALL策略授权删除行之外，通过SELECT或ALL策略，用户也必须能访问正被删除的行。

DELETE策略不能具有WITH CHECK表达式，因为它只适用于正在从关系中删除记录的情况， 所以没有新行需要检查。

Table 270. 按命令类型应用的策略

命令	SELECT/ALL策略	INSERT/ALL策略	UPDATE/ALL策略		DELETE/ALL策略
	USING表达式	WITH CHECK表达式	USING表达式	WITH CHECK表达式	USING表达式
SELECT	现有行	—	—	—	—
SELECT FOR UPDATE/SHARE	现有行	—	现有行	—	—
INSERT	—	新行	—	—	—
INSERT ... RETURNING	新行 <sup>a</sup>	新行	—	—	—
UPDATE	现有 & 新行 <sup>a</sup>	—	现有行	新行	—
DELETE	现有行 <sup>a</sup>	—	—	—	现有行
ON CONFLICT DO UPDATE	现有 & 新行	—	现有行	新行	—

<sup>a</sup> 对于现有行或新行，如果需要读访问的话（例如涉及到关系内列的WHERE或RETURNING子句）。

## 多重策略的应用

当多种不同命令类型的策略应用于相同命令（例如SELECT和UPDATE策略应用于UPDATE命令）时，用户就必须同时具有这两种类型的权限（例如从关系中选取行和更新的权限）。这样一种策略类型的表达式就与另一种策略类型的表达式通过使用AND操作符组合在一起。

当相同命令类型的多种策略应用于同一命令时，则必须至少有一个PERMISSIVE策略授权对该关系的访问，所有的RESTRICTIVE策略必须通过。这样，所有的PERMISSIVE策略表达式都用OR来组合，所有的RESTRICTIVE策略表达式都用AND来组合，而结果用AND来组合。如果没有PERMISSIVE策略，则拒绝访问。

要注意的是，出于组合多种策略的目的，将ALL策略视为与所应用的任何其他类型的策略具有相同的类型。

例如，在UPDATE命令中，SELECT和UPDATE两种权限都需要，如果每种类型都有多个适用的策略，则将其以下面的方式组合：

```
expression from RESTRICTIVE SELECT/ALL policy 1
AND
expression from RESTRICTIVE SELECT/ALL policy 2
AND
...
AND
(
  expression from PERMISSIVE SELECT/ALL policy 1
  OR
  expression from PERMISSIVE SELECT/ALL policy 2
  OR
  ...
)
AND
expression from RESTRICTIVE UPDATE/ALL policy 1
AND
expression from RESTRICTIVE UPDATE/ALL policy 2
AND
...
AND
(
  expression from PERMISSIVE UPDATE/ALL policy 1
  OR
  expression from PERMISSIVE UPDATE/ALL policy 2
  OR
  ...
)
```

## 注解

要为一个表创建或者修改策略，你必须是该表的拥有者。

虽然策略将被应用于针对数据库中表的显式查询上，但当系统正在执行内部引用完整性检查或者验证约束时不会应用它们。这意味着有间接的方法来决定一个给定的值是否存在。一个例子是向一个作为主键或者拥有唯一约束的列中尝试插入重复值。如果插入失败则用户可以推导出该值已经存在（这个例子假设用户被策略允许插入他们看不到的记录）。另一个例子是一个用户被允许向一个引用了其他表的表中插入，然而另一个表是隐藏表。通过用户向引用表中插入值可以判断存在性，成功表示该值存在于被引用表中。为了解决这些问题，应该仔细地制作策略以完全阻止用户插入、删除或者更新那些可能指示他们不能看到的值的记录，或者使用生成的值（例如代理键）来代替具有外部含义的键。

通常，系统将在应用用户查询中出现的条件之前先强制由安全性策略施加的过滤条件，这是为了防止无意中把受保护的数据暴露给可能不可信的用户定义函数。不过，被系统（或者系统管理员）标记为 LEAKPROOF的函数和操作符可以在策略表达式之前被计算，因为它们已经被假定为可信。

因为策略表达式会被直接加到用户查询上，它们将使用运行整个查询的用户的权限运行。因此，使用一条给定策略的用户必须能够访问表达式中引用的任何表或函数，否则在尝试查询启用了行级安全性的表时，他们将简单地收到一条没有权限的错误。不过，这不会改变视图的工作方式。就普通查询和视图来说，权限检查和视图所引用的表的策略将使用视图拥有者的权限以及任何适用于视图拥有者的策略。

在Section 5.8中可以找到额外的讨论和实际的例子。

## 兼容性

CREATE POLICY是一种LightDB扩展。

## 另见

ALTER POLICY, DROP POLICY, ALTER TABLE



---

# CREATE PROCEDURE

CREATE PROCEDURE — 定义一个新的过程

## Synopsis

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
    [, ...] ] )
    { LANGUAGE lang_name
      | TRANSFORM { FOR TYPE type_name } [, ... ]
      | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
      | SET configuration_parameter { TO value | = value | FROM CURRENT }
      | AS 'definition'
      | AS 'obj_file', 'link_symbol'
    } ...
```

## 简介

CREATE PROCEDURE定义一个新的过程。CREATE OR REPLACE PROCEDURE将会创建一个新过程或者替换一个已有的定义。为了能够定义过程，用户必须具有所使用的语言上的USAGE特权。

如果这个命令中包括了一个方案名称，则该过程将被创建在该方案中。否则过程将被创建在当前的方案中。新过程的名称不能匹配同一方案中具有相同输入参数类型的任何现有过程或函数。不过，具有不同参数类型的过程和函数可以共享同一个名称（这被称为重载）。

要替换一个已有过程的当前定义，请使用CREATE OR REPLACE PROCEDURE。不能用这种方式更改过程的名称或者参数类型（如果尝试这样做，实际上会创建一个新的、不同的过程）。

当CREATE OR REPLACE PROCEDURE被用来替换一个现有的过程时，该过程的拥有关系和权限保持不变。所有其他的过程属性会被赋予这个命令中指定的或者暗示的值。必须拥有（包括成为拥有角色的成员）该过程才能替换它。

创建过程的用户将成为该过程的拥有者。

为了能够创建一个过程，用户必须具有参数类型上的USAGE特权。

有关编写过程的详细信息，请参阅Section 36.4

## 参数

name

要创建的过程的名称（可以是被方案限定的）。

argmode

参数的模式可以是：IN、OUT、 INOUT或者VARIADIC。 如果省略，则默认为IN。

argname

参数的名称。

argtype

过程的参数（如果有）的数据类型（可以是被方案限定的）。参数类型可以是基础类型、组合类型或者域类型，或者可以引用一个表列的类型。

根据具体的实现语言，还可能指定“伪类型”，例如cstring。伪类型表示实际的参数类型没有完全确定，或者是位于普通SQL数据类型的集合之外。

写上table\_name.column\_name%TYPE可以引用某个列的类型。使用这种特性有时可以让过程不受表定义改变的影响。

default\_expr

没有指定参数时要被用作默认值的表达式。这个表达式必须符合该参数的参数类型。跟在有默认值的参数后面的输入参数也都必须有默认值。

lang\_name

用于实现该过程的语言名称。它可以是sql、c、internal或者一种用户定义的过程语言的名称，例如plpgsql。如果指定了sql\_body，则默认值为sql。将名称包裹在单引号内的方式已经被废弃，并且要求大小写匹配。

TRANSFORM { FOR TYPE type\_name } [, ... ] }

列出对过程的调用应该应用哪些Transform。Transform负责在SQL类型和语言相关的数据类型之间进行转换，请参考CREATE TRANSFORM。过程语言实现通常采用硬编码的方式保存内建类型的知识，因此它们无需在这里列出。但如果一种过程语言实现不知道如何处理一种类型并且没有提供Transform，它将回退到默认的行为来转换数据类型，但是这依赖于其实现。

[EXTERNAL] SECURITY INVOKER

[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER指示过程以调用它的用户的特权来执行。这是默认方式。SECURITY DEFINER指定过程以拥有它的用户的特权来执行。

为了符合SQL标注，允许使用EXTERNAL关键词，但它是可选的，因为和SQL中不同，这个特性适用于所有的过程而不仅仅是外部过程。

SECURITY DEFINER过程不能执行事务控制语句（例如COMMIT和ROLLBACK，具体取决于实现的语言）。

configuration\_parameter

value

SET子句导致在进入该过程时指定的配置参数被设置为指定的值，并且在过程退出时恢复到之前的值。SET FROM CURRENT把CREATE PROCEDURE执行时该参数的当前值保存为在进入该过程时要应用的值。

如果对过程附加一个SET子句，那么在该过程中为同一个变量执行的SET LOCAL命令的效果就被限制于该过程：在过程退出时还是会恢复到该配置参数的以前的值。不过，一个普通的SET命令（没有LOCAL）会重载这个SET子句，很像它对一个之前的SET LOCAL命令所做的事情：这样一个命令的效果将持续到过程退出之后，除非当前事务被回滚。

如果对过程附加一个SET子句，则该过程不能执行事务控制语句（例如COMMIT和ROLLBACK，具体取决于实现的语言）。

有关允许的参数名和值的更多信息请参考SET和Chapter 17

definition

一个定义该过程的字符串常量，其含义取决于语言。它可以是一个内部的过程名、一个对象文件的路径、一个SQL命令或者以一种过程语言编写的文本。

在编写过程的定义字符串时，使用美元引用（见Section 4.1.2.4 而不是普通的单引号语法常常会有帮助。如果没有美元引用，过程定义中的任何单引号或者反斜线必须以双写的方式进行转义。

obj\_file, link\_symbol

当C语言源码中的过程名与SQL过程的名称不同时，这种形式的AS子句被用于动态可装载的C语言过程。字符串obj\_file是包含已编译好的C过程的共享库文件名，并且被按照LOAD命令的方式解析。字符串link\_symbol是该过程的链接符号，也就是该过程在C语言源代码中的名称。如果链接符号被省略，则会被假定为与正在被定义的SQL过程的名称相同。

当重复的CREATE PROCEDURE调用引用同一个对象文件时，只会对每一个会话装载该文件一次。要卸载或者重新载入该文件（可能是在开发期间），应该开始一个新的会话。

## 注解

函数创建也适用于过程，更多细节请参考CREATE FUNCTION。

使用CALL来执行过程。

## 示例

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;

CALL insert_data(1, 2);
```

## 兼容性

SQL标准中定义有一个CREATE PROCEDURE命令。LightDB实现可以以兼容的方式使用，但有许多扩展。有关详细信息，请参见CREATE FUNCTION。

## 另见

ALTER PROCEDURE, DROP PROCEDURE, CALL, CREATE FUNCTION

---

# CREATE PUBLICATION

CREATE PUBLICATION — 定义一个新的发布

## Synopsis

```
CREATE PUBLICATION name
  [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
    | FOR ALL TABLES ]
  [ WITH ( publication_parameter [= value] [, ... ] ) ]
```

## 描述

CREATE PUBLICATION向当前数据库添加一个新的发布。发布的名称必须与当前数据库中任何现有发布的名称不同。

发布本质上是一组表，其数据更改旨在通过逻辑复制进行复制。有关发布如何适应逻辑复制设置的详细信息，请参见Section 28.1

## 参数

name

新发布的名称。

FOR TABLE

指定要添加到发布的表的列表。如果在表名之前指定了ONLY，那么只有该表被添加到发布中。如果没有指定ONLY，则添加表及其所有后代表（如果有的话）。可选地，可以在表名之后指定\*以明确指示包含后代表。但是，这不适用于分区表。分区表的分区始终被隐式视为发布的一部分，因此永远不会明确将其添加到发布中。

只有持久基表和分区表才能成为出版物的一部分。临时表，未记录表，外部表，物化视图和常规视图不能成为发布的一部分。

将分区表添加到发布时，其所有现有分区和将来分区都被隐式视为发布的一部分。因此，即使直接在分区上执行的操作也会通过其祖先所在的发布来发布。

FOR ALL TABLES

将发布标记为复制数据库中所有表的更改，包括在将来创建的表。

WITH ( publication\_parameter [= value] [, ... ] )

该子句指定发布的可选参数。支持下列参数：

publish (string)

这个参数决定了哪些DML操作将由新的发布发布给订阅者。该值是用逗号分隔的操作列表。允许的操作是insert, update, delete和truncate。默认是发布所有的动作，所以这个选项的默认值是'insert, update, delete, truncate'。

publish\_via\_partition\_root (boolean)

此参数确定是否会使用分区表的标识和模式而不是实际更改的单个分区的标识和模式来发布发布中包含的分区表（或其分区）中的更改。单个分区的标识和模式是默

认设置。 启用此功能可以将更改复制到非分区表或由一组不同的分区组成的分区表中。

如果启用此功能，则不会复制直接在分区上执行的TRUNCATE操作。

## 注意

如果既没有指定FOR TABLE，也没有指定FOR ALL TABLES，那么这个发布就是以一组空表开始的。这在稍后添加表格的情况下是有用的。

创建发布不会开始复制。它只为未来的订阅者定义一个分组和过滤逻辑。

要创建一个发布，调用者必须拥有当前数据库的CREATE权限。（当然，超级用户不需要这个检查。）

要将表添加到发布中，调用者必须拥有该表的所有权。FOR ALL TABLES 子句要求调用者是超级用户。

添加到发布UPDATE和/或DELETE 操作的发布的表必须已经定义了REPLICA IDENTITY。否则将在这些表上禁止这些操作。

对于INSERT ... ON CONFLICT命令，发布将公布从命令实际产生的操作。因此，根据结果，它可以作为 INSERT或UPDATE发布，也可以根本不发布。

ATTACH将表添加到分区树中，（该分区树的根是使用publish\_via\_partition\_root设置为true发布的）不会导致复制表的现有内容。

COPY ... FROM命令是作为INSERT操作发布的。

不发布DDL操作。

## 示例

创建一个发布，发布两个表中所有更改布：

```
CREATE PUBLICATION mypublication FOR TABLE users, departments;
```

创建一个发布，发布所有表中的所有更改：

```
CREATE PUBLICATION alltables FOR ALL TABLES;
```

创建一个发布，只发布一个表中的INSERT操作：

```
CREATE PUBLICATION insert_only FOR TABLE mydata
WITH (publish = 'insert');
```

## 兼容性

CREATE PUBLICATION是一个LightDB 扩展。

## 又见

ALTER PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

---

# CREATE ROLE

CREATE ROLE — 定义一个新的数据库角色

## Synopsis

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]
```

where option可以是:

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
| VALID UNTIL 'timestamp'
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN role_name [, ...]
| USER role_name [, ...]
| SYSID uid
```

## 描述

CREATE ROLE向LightDB数据库集簇增加一个新的角色。一个角色是一个实体，它可以拥有数据库对象并且拥有数据库特权。根据一个角色如何被使用，它可以被考虑成一个“用户”、一个“组”或者两者。有关管理用户和认证的信息可以参考Chapter 19和Chapter 18 要使用这个命令，你必须具有CREATEROLE特权或者成为一个数据库超级用户。

注意角色是定义在数据库集簇层面上的，并且因此在集簇中的所有数据库中都可用。

## 参数

name

新角色的名称。

SUPERUSER  
NOSUPERUSER

这些子句决定新角色是否是一个“超级用户”，它可以越过数据库内的所有访问限制。超级用户状态很危险并且只应该在确实需要时才用。要创建一个新超级用户，你必须自己是一个超级用户。如果没有指定，默认值是NOSUPERUSER。

CREATEDB  
NOCREATEDB

这些子句定义一个角色创建数据库的能力。如果指定了CREATEDB，被定义的角色将被允许创建新的数据库。指定NOCREATEDB将否定一个角色创建数据库的能力。如果没有指定，默认值是NOCREATEDB。

CREATEROLE  
NOCREATEROLE

这些子句决定一个角色是否被允许创建新的角色（也就是执行CREATE ROLE）。一个带有CREATEROLE特权的角色也能修改和删除其他角色。如果没有指定，默认值是NOCREATEROLE。

INHERIT  
NOINHERIT

如果新的角色是其他角色的成员，这些子句决定新角色是否从那些角色中“继承”特权，把新角色作为成员的角色称为新角色的父角色。一个带有INHERIT属性的角色能够自动使用已经被授予给其直接或间接父角色的任何数据库特权。如果没有INHERIT，在另一个角色中的成员关系只会把SET ROLE的能力授予给那个其他角色，只有在这样做后那个其他角色的特权才可用。如果没有指定，默认值是INHERIT。

LOGIN  
NOLOGIN

这些子句决定一个角色是否被允许登录，也就是在客户端连接期间该角色是否能被 给定为初始会话认证名称。一个具有LOGIN属性的角色可以被 考虑为一个用户。没有这个属性的角色对于管理数据库特权很有用，但是却不是 用户这个词的通常意义。如果没有指定，默认值是NOLOGIN， 不过当CREATE ROLE被通过 CREATE USER调用时默认值会是LOGIN。

REPLICATION  
NOREPLICATION

这些子句决定一个角色是否为复制角色。角色必须具有这个属性（或者成为一个超级用户）才能以复制模式（物理复制或者逻辑复制）连接到服务器以及创建或者删除复制槽。一个具有REPLICATION属性的角色是一个具有非常高特权的角色，并且只应被用于确实需要复制的角色上。如果没有指定，默认值是NOREPLICATION。 您必须是超级用户才能创建具有REPLICATION属性的新角色。

BYPASSRLS  
NOBYPASSRLS

这些子句决定是否一个角色可以绕过每一条行级安全性（RLS）策略。 默认是NOBYPASSRLS。 您必须是超级用户才能创建具有BYPASSRLS属性的新角色。

注意 `lt_dump` 将默认把`row_security`设置为OFF， 以确保一个表的所有内容被转储出来。如果运行 `lt_dump` 的用户不具有适当的权限，将会返回一个错误。 但是，超级用户和被转储表的拥有者总是可以绕过 RLS。

CONNECTION LIMIT `connlimit`

如果角色能登录，这指定该角色能建立多少并发连接。-1（默认值）表示无限制。注意这个限制仅针对于普通连接。预备事务和后台工作者连接都不受这一限制管辖。

[ ENCRYPTED ] PASSWORD 'password'  
PASSWORD NULL

设置角色的口令（口令只对具有LOGIN属性的角色有用，但是不管怎样你还是可以为没有该属性的角色定义一个口令）。如果你没有计划使用口令认证，你可以忽略这个选项。如果没有指定口令，口令将被设置为空并且该用户的口令认证总是会失败。也可以用PASSWORD NULL显式地写出一个空口令。

#### Note

指定一个空字符串也将把口令设置为空，但是在LightDB版本10之前是不能这样做的。在早期的版本中，是否可以使用空字符串取决于认证方法和

确切的版本，而libpq在任何情况下都将拒绝使用空字符串。为了避免混淆，应该避免指定空字符串。

口令总是以加密的方式存放在系统目录中。ENCRYPTED关键词没有实际效果，它只是为了向后兼容性而存在。加密的方法由配置参数password\_encryption决定。如果当前的口令字符串已经是MD5加密或者SCRAM加密的格式，那么不管password\_encryption的值是什么，口令字符串还是原样存储（因为系统无法解密以不同格式加密的口令字符串）。这种方式允许在转储/恢复时重载加密的口令。

VALID UNTIL 'timestamp'

VALID UNTIL机制设置一个日期和时间，在该时间点之后角色的口令将会失效。如果这个子句被忽略，那么口令将总是有效。

IN ROLE role\_name

IN ROLE子句列出一个或多个现有的角色，新角色将被立即作为新成员加入到这些角色中（注意没有选项可以把新角色作为一个管理员加入，需要用单独的GRANT命令来完成）。

IN GROUP role\_name

IN GROUP是IN ROLE的一种已废弃的拼写方式。

ROLE role\_name

ROLE子句列出一个或者多个现有角色，它们会被自动作为成员加入到新角色中（这实际上新角色变成了一个“组”）。

ADMIN role\_name

ADMIN子句与ROLE相似，但是被提及的角色被使用WITH ADMIN OPTION加入到新角色中，让它们能够把这个角色中的成员关系授予给其他人。

USER role\_name

USER子句是ROLE子句的一个已废弃的拼写方式。

SYSID uid

SYSID子句会被忽略，但是会为了向后兼容，还是会接受它。

## 注解

使用ALTER ROLE来更改一个角色的属性，以及使用DROP ROLE来移除一个角色。所有用CREATE ROLE指定的属性可以被后来的ALTER ROLE命令所修改。

增加和移除组角色成员的最佳方式是使用GRANT和REVOKE。

VALID UNTIL子句只为一个口令而不是为一个角色本身定义了一个过期时间。特别地，当使用一个非基于口令认证的方法登录时，过期时间是不会被强制的。

INHERIT属性管理可授予特权（也就是对数据库对象和角色成员关系的访问特权）的继承性。它并不适用于由CREATE ROLE和ALTER ROLE设置的特殊角色属性。例如，作为具有CREATEDB特权的角色的一个成员，并不会立刻授予创建数据库的角色，即便INHERIT被设置也是如此，在创建一个数据库之前必须通过SET ROLE成为该角色。

INHERIT属性是用于向后兼容原因的默认值：在早前的LightDB发布中，用户总是能够访问其所属组的所有特权。不过，NOINHERIT更加接近于SQL标准中指定的语义。



要小心CREATEROLE特权。对于一个CREATEROLE角色的特权没有继承的概念。那意味着即使一个角色没有特定的特权但被允许创建其他角色，它可以轻易地创建与自身特权不同的另一个角色（除了创建具有超级用户特权的角色）。例如，如果角色“user”具有CREATEROLE特权但是没有CREATEDB特权，但是它能够创建一个带有CREATEDB特权的新角色。因此，可以把具有CREATEROLE特权的角色看成是准超级用户角色。

LightDB包括一个程序createuser，它具有和CREATE ROLE相同的功能（事实上，它会调用这个命令），但是它可以命令 shell 中运行。

CONNECTION LIMIT只被近似地强制，如果两个新会话在几乎相同时间被开始，而此时该角色只剩下刚好一个连接“槽”，两者可能都将失败。还有，该限制从不对超级用户强制。

用这个命令指定一个非加密口令时必须加以注意。该命令将被以明文的形式传输到服务器，并且它也可能被记录在客户端命令历史或者服务器日志中。不过，命令createuser会传输加密的口令。还有，ltsql包含一个命令\password，它可以被用来安全地改变该口令。

## 例子

创建一个能登录但是没有口令的角色：

```
CREATE ROLE jonathan LOGIN;
```

创建一个有口令的角色：

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

（CREATE USER和CREATE ROLE完全相同，除了它带有LOGIN）。

创建一个角色，它的口令有效期截止到 2004 年底。在进入 2005 年第一秒时，该口令会失效。

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

创建一个能够创建数据库并且管理角色的角色：

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

## 兼容性

SQL 标准中有CREATE ROLE语句，但是标准只要求语法

```
CREATE ROLE name [ WITH ADMIN role_name ]
```

多个初始管理员以及CREATE ROLE的所有其他选项都是LightDB扩展。

SQL 标准定义了用户和角色的概念，但是它把它们看成是可区分的概念并且将定义用户的所有命令留给每一种数据库实现来指定。在LightDB中，我们已经选择把用户和角色统一成一种单一实体类型。因此角色比起标准中拥有更多可选的属性。

SQL 标准指定的行为可以通过给用户NOINHERIT属性来得到最大的金丝，而角色会被给予INHERIT属性。

## 参见

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser

---

# CREATE RULE

CREATE RULE — 定义一条新的重写规则

## Synopsis

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

其中 event 可以是以下之一：

```
SELECT | INSERT | UPDATE | DELETE
```

## 描述

CREATE RULE定义一条应用于指定表或视图的 新规则。CREATE OR REPLACE RULE将创建一条新规则或者替换同一个表上具有同一名称的现有规则。

LightDB规则系统允许我们定义 针对数据库表中插入、更新或者删除动作上的替代动作。大约来说，当在 一个给定表上执行给定命令时，一条规则会导致执行额外的命令。或者，INSTEAD规则可以用另一个命令替换给定的命令，或 者导致一个命令根本不被执行。规则也被用来实现 SQL 视图。规则实际上 是一种命令转换机制或者命令宏。这种转换会在命令的执行开始之前进行。 如果你实际上想要为每一个物理行独立地触发一个操作，你可能更需要一个 触发器而不是规则。更多有关规则系统的信息请见Chapter 39

当前，ON SELECT规则必须是无条件 INSTEAD规则并且其动作必须由一个单一 SELECT命令构成。因此，一条 ON SELECT规则实际上把表变成了一个视图，它的可见 内容是由该规则的SELECT命令返回，而不是直 接存在该表中的内容（如果有）。不过，使用 CREATE VIEW命令还是要比创建一个真实表并且 在其上定义一条ON SELECT规则更好。

可以通过定义ON INSERT、ON UPDATE 以及ON DELETE规则（或者这些规则的任意子集）来创建 可更新的视图，这些规则可以把视图上的更新动作替换为其他表上适当的更新 动作。如果想要支持INSERT RETURNING等等，那么一定要 在每一个这类规则中放上一个合适的RETURNING子句。

如果你尝试为复杂视图更新使用有条件的规则，有一点是很重要的：对于 你希望在该视图上允许的每一个动作，必须有一条 INSTEAD规则。如果该规则是有条件的，或者不是INSTEAD，那么系统仍将拒绝尝试执行该更新动作， 因为它会认为在某些情况下它应该停止尝试在该视图的傀儡表上执行动作。 如果你想处理有条件规则中的所有有用的情况，可以增加一条无条件的 DO INSTEAD NOTHING规则来确保系统理解它将 永远不会被调用来更新傀儡表。然后让有条件规则变成 非-INSTEAD。在它们适用的情况下，它们会加到 默认的INSTEAD NOTHING动作（不过，当前这种方法不 支持RETURNING查询）。

### Note

足够简单的视图自动就是可更新的（见CREATE VIEW），它们不需要依靠用户创建的规则来变成可 更新的。不过还是可以创建一条显式规则，自动更新转换通常比显式规则效 率高。

另一种值得考虑的办法是使用INSTEAD OF触发器（见 CREATE TRIGGER）代替规则。

## 参数

name

要创建的规则的名称。它必须与同一个表上任何其他规则的名称相区分。同一个表上同一种事件类型的多条规则会按照其名称的字符顺序被应用。

event

时间是SELECT、INSERT、UPDATE或者DELETE之一。注意包含ON CONFLICT子句的INSERT不能被用在具有INSERT或者UPDATE规则的表上。那种情况下请考虑使用可更新的视图。

table\_name

规则适用的表或者视图的名称（可以是模式限定的）。

condition

任意的SQL条件表达式（返回boolean）。该条件表达式不能引用除NEW以及OLD之外的任何表，并且不能包含聚集函数。

INSTEAD

INSTEAD指示该命令应该取代原始命令被执行。

ALSO

ALSO指示应该在原始命令之外执行这些命令。

如果ALSO和INSTEAD都没有被指定，默认是ALSO。

command

组成规则动作的命令。可用的命令有SELECT、INSERT、UPDATE、DELETE或者NOTIFY。

在condition和command中，名为NEW和OLD的表可以被用来引用被引用表中的值。在ON INSERT和ON UPDATE规则中，NEW被用来引用被插入或者更新的新行。在ON UPDATE和ON DELETE规则中，OLD被用来引用被更新或者删除的现有行。

## 注解

要在表上创建或者修改规则，必须是表的拥有者。

在一条用于视图上INSERT、UPDATE或者DELETE的规则中，可以增加一个RETURNING子句来发出视图的列。如果该规则被一个INSERT RETURNING、UPDATE RETURNING或者DELETE RETURNING命令触发，这个子句将被用来计算输出。当规则被一个没有RETURNING的命令触发时，该规则的RETURNING子句将被忽略。当前的实现只允许无条件INSTEAD规则包含RETURNING。此外，用于同一事件的所有规则中至多只能有一个RETURNING子句（这确保了只有一个候选RETURNING子句被用来计算结果）。如果在任何可用规则中都没有RETURNING子句，视图上的RETURNING查询将被拒绝。

避免循环规则非常重要。例如，尽管下面的两条规则定义都被LightDB所接受，SELECT命令将导致LightDB报告一个错误，因为会产生一条规则的递归扩展：

```
CREATE RULE "_RETURN" AS
  ON SELECT TO t1
  DO INSTEAD
    SELECT * FROM t2;
```

```
CREATE RULE "_RETURN" AS
  ON SELECT TO t2
  DO INSTEAD
    SELECT * FROM t1;

SELECT * FROM t1;
```

当前，如果一个规则动作包含一个NOTIFY命令，该NOTIFY命令将被无条件执行，也就是说，即使该规则不被应用到任何行上，也会发出NOTIFY。例如，在

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

中，UPDATE期间将发出一个NOTIFY事件，不管是否有行匹配条件id = 42。这是一种实现限制，它可能会在未来的发行中被修复。

## 兼容性

CREATE RULE是一种LightDB语言扩展，整个查询重写系统也是这样。

## 另见

ALTER RULE, DROP RULE

---

# CREATE SCHEMA

CREATE SCHEMA — 定义一个新模式

## Synopsis

```
CREATE SCHEMA schema_name [ AUTHORIZATION role_specification ] [ schema_element
[ ... ] ]
CREATE SCHEMA AUTHORIZATION role_specification [ schema_element [ ... ] ]
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION role_specification ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification
```

其中 `role_specification` 可以是：

```
user_name
| CURRENT_USER
| SESSION_USER
```

## 描述

CREATE SCHEMA输入一个新模式到当前数据库中。该模式名必须与当前数据库中任何现有模式的名称不同。

一个模式本质上是一个名字空间：它包含命令对象（表、数据类型、函数以及操作符），对象可以与在其他模式中存在的对象重名。可以通过用模式名作为一个前缀“限定”命名对象的名称来访问它们，或者通过把要求的模式包括在搜索路径中来访问命名对象。一个指定非限定对象名的 CREATE命令在当前模式（搜索路径中的第一个模式，由函数 `current_schema` 决定）中创建对象。

CREATE SCHEMA中可以选择包括子命令用以在新模式中创建对象。这些子命令实际被当做独立的在创建该模式后被发出的命令一样，除非使用AUTHORIZATION子句，所有被创建的对象都会由该用户拥有。

## 参数

`schema_name`

要创建的一个模式名。如果省略，`user_name`将被用作模式名。该名称不能以`pg_`开始，因为这样的名称是用作系统模式的。

`user_name`

将拥有新模式的用户的角色名。如果省略，默认为执行该命令的用户。要创建由另一个角色拥有的角色，你必须那个角色的一个直接或者间接成员，或者是一个超级用户。

`schema_element`

要在该模式中创建的对象定义 SQL 语句。当前，只有CREATE TABLE、CREATE VIEW、CREATE INDEX、CREATE SEQUENCE、CREATE TRIGGER以及GRANT被接受为 CREATE SCHEMA中的子句。其他类型的对象可以在模式被创建之后用单独的命令创建。

IF NOT EXISTS

如果一个具有同名的模式已经存在，则什么也不做（不过发出一个提示）。使用这个选项时不能包括 `schema_element`子命令。

## 注解

要创建一个模式，调用用户必须拥有当前数据库的CREATE 特权（当然，超级用户可以绕过这种检查）。

## 示例

创建一个模式：

```
CREATE SCHEMA myschema;
```

为用户joe创建一个模式，该模式也将被命名为 joe：

```
CREATE SCHEMA AUTHORIZATION joe;
```

创建一个被用户joe拥有的名为test的模式，除非已经有一个名为test的模式（不管joe 是否拥有该已经存在的模式）。

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

创建一个模式并且在其中创建一个表和视图：

```
CREATE SCHEMA hollywood
  CREATE TABLE films (title text, release date, awards text[])
  CREATE VIEW winners AS
    SELECT title, release FROM films WHERE awards IS NOT NULL;
```

注意子命令不以分号结束。

下面是达到相同结果的等效的方法：

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
  SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

## 兼容性

SQL 标准允许在CREATE SCHEMA中有一个 DEFAULT CHARACTER SET子句，以及当前 LightDB接受的更多子命令类型。

SQL 标准制定CREATE SCHEMA中的子命令 可以以任何顺序出现。当前的 LightDB实现不能处理子命令中 所有情况的向前引用。有时候可能有必要对子命令进行重排序以避免向前 引用。

根据 SQL 标准，模式的拥有者总是拥有其中的所有对象。 LightDB允许模式包含非模式 拥有者所拥有的对象。只有模式拥有者把其模式上的CREATE 特权授予给了其他人或者一个超级用户选择在该模式中创建对象时才会 发生这种事情。

IF NOT EXISTS选项是一种 LightDB扩展。

## 另见

ALTER SCHEMA, DROP SCHEMA

---

# CREATE SEQUENCE

CREATE SEQUENCE — 定义一个新的序列发生器

## Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name
  [ AS data_type ]
  [ INCREMENT [ BY ] increment ]
  [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
  [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
  [ OWNED BY { table_name.column_name | NONE } ]
  [ NOORDER ]
  [ ORDER ]
  [ NOCACHE ]
  [ NOCYCLE ]
  [ NOMAXVALUE ]
```

## 描述

CREATE SEQUENCE 创建一个新的序列数 发生器。这涉及到用名称 name 创建并且初始化 一个新的特殊的单行表。该发生器将由发出该命令的用户所拥有。

如果给出一个模式名称，则该序列将被创建在指定的模式中。否则它会被 创建在当前模式中。临时序列存在于一个特殊的模式中，因此在创建临时序列 时不能给出模式名。序列名称必须与同一模式中任何其他序列、表、索引、视图或者外部表的名称不同。

在序列被创建后，可以使用函数 nextval、currval 以及 setval 来操作该序列。这些函数在 Section 9.1 中有介绍。

尽管无法直接更新一个序列，可以使用这样的查询：

```
SELECT * FROM name;
```

来检查一个序列的参数以及当前状态。特别地，序列的 last\_value 域显示被任意会话最后一次取得的值（当然，在被打印时该值可能已经过时了，因为可能有其他会话正在执行 nextval 调用）。

## 参数

TEMPORARY or TEMP

如果被指定，只会为这个会话创建序列对象，并且在会话退出时自动 删除它。当临时序列存在时，已有的同名永久序列（在这个会话中） 会变得不可见，不过可以用模式限定的名称来引用同名永久序列。

IF NOT EXISTS

如果已经存在一个同名的关系时不要抛出错误。这种情况下会发出一个 提示。注意这不保证现有的关系与即将创建的序列相似 — 它甚至可能 都不是一个序列。

name

要创建的序列的名称（可以是模式限定的）。

## data\_type

可选的子句AS data\_type 制定序列的数据类型。有效类型是 smallint、integer、和bigint。默认是bigint。数据类型决定了序列的默认最小和最大值。

## increment

可选的子句INCREMENT BY increment指定为了 创建新值会把哪个值加到当前序列值上。一个正值将会创建一个上升 序列，负值会创建一个下降序列。默认值是 1。

## minvalue

## NO MINVALUE

可选的子句MINVALUE minvalue决定一个序列 能产生的最小值。如果没有提供这个子句或者指定了 NO MINVALUE，那么会使用默认值。升序序列的默认值为1。降序序列的默认值为数据类型的最小值。

## maxvalue

## NO MAXVALUE

可选的子句MAXVALUE maxvalue决定该序列 的最大值。如果没有提供这个子句或者指定了 NO MAXVALUE，那么将会使用默认值。升序序列的默认值是数据类型的最大值。降序序列的默认值是-1。

如果maxvalue的值大于INT64\_MAX（9223372036854775807），则设置maxvalue为sequence类型所对应的最大值。

## start

可选的子句START WITH start 允许序列从任何 地方开始。对于上升序列和下降序列来说，默认的开始值分别是 minvalue和 maxvalue。

## cache

可选的子句CACHE cache指定要预分配多少 个序列数并且把它们放在内存中以便快速访问。最小值为 1 （一次只生成 一个值，即没有缓存），默认值也是 1。

## CYCLE

## NO CYCLE

对于上升序列和下降序列，CYCLE选项允许序列 在分别达到maxvalue和minvalue时回卷。如果达到 该限制，下一个产生的数字将分别是minvalue和maxvalue。

如果指定了NO CYCLE，当序列到达其最大值 后任何nextval调用将返回一个错误。如果CYCLE和NO CYCLE都没有 被指定，则默认为NO CYCLE。

## OWNED BY table\_name.column\_name

## OWNED BY NONE

OWNED BY选项导致序列被与一个特定的表列关联 在一起，这样如果该列（或者整个表）被删除，该序列也将被自动删除。指定的表必须和序列具有相同的拥有者并且在同一个模式中。默认选项 OWNED BY NONE指定该序列不与某个列关联。

## NOORDER

## ORDER

## NOCACHE

## NOCYCLE

为了兼容Oracle数据库语法，CREATE SEQUENCE语句支持 NOORDER/ORDER/NOCACHE/NOCYCLE 关键字。这些关键字仅是语法糖。



## NOMAXVALUE

为了兼容Oracle数据库语法，CREATE SEQUENCE语句支持 NOMAXVALUE 关键字。内部逻辑和NO MAXVALUE相同。

## 注解

使用DROP SEQUENCE移除一个序列。

序列是基于bigint算法的，因此范围是不能超过一个八字节 整数的范围  
(-9223372036854775808 到 9223372036854775807)。

由于nextval和setval调用绝不会回滚， 如果需要序数的“无间隙”分配，则不能使用序列对象。可以通过在一个只包含一个计数器的表上使用排他锁来构建无间隙的分配， 但是这种方案比序列对象开销更大，特别是当有很多事务并发请求序数 时。

如果对一个将由多个会话并发使用的序列对象使用了大于 1 的cache设置，可能会得到意想不到的结果。 每个会话会在访问该序列对象时分配并且缓存后续的序列值，并且相应地增加 该序列对象的last\_value。然后，在该会话中下一次 nextval会做 cache-1，并且简单地 返回预分配的值而不修改序列对象。因此，任何已分配但没有在会话中使用的 数字将会在该会话结束时丢失，导致该序列中的“空洞”。

进一步，尽管多个会话能分配到不同的序列值，这些值可能会在所有会话都被 考虑时生成出来。例如， cache的设置为 10，会话 A 可能储存值 1..10 并且返回nextval=1，然后会话 B 可能储存值 11..20 并且在 A 生成nextval=2 之前返回 nextval=11。因此，如果cache设置为 1，可以 安全地假设nextval值被顺序地生成。如果cache设置大于 1，就只能假定 nextval值都是可区分的，但不能保证它们被完全地顺序生成。 还有，last\_value将反映服务于任意会话的最后一个值，不管它 是否已经被nextval返回过。

另一个考虑是，在这样一个序列上执行的setval将不会通知 其他会话，直到它们用尽了任何已缓存的预分配值。

## 示例

创建一个称作serial的上升序列，从 101 开始：

```
CREATE SEQUENCE serial START 101;
```

从这个序列中选取下一个数字：

```
SELECT nextval('serial');
```

```
nextval
-----
      101
```

再从这个序列中选取下一个数字：

```
SELECT nextval('serial');
```

```
nextval
-----
      102
```

在一个INSERT命令中使用这个序列：

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

在一次COPY FROM后更新新列值:

```
BEGIN;  
COPY distributors FROM 'input_file';  
SELECT setval('serial', max(id)) FROM distributors;  
END;
```

使用ORDER/NOORDER/NOCACHE/NOCYCLE选项来创建sequence。

```
create sequence lt_seq_test5 start 100 ORDER nocache NOCYCLE;  
create sequence lt_seq_test6 start 100 NOCACHE NOORDER nocycle;
```

## 兼容性

CREATE SEQUENCE符合SQL 标准，不过下列除外:

- 使用nextval() 而不是标准的NEXT VALUE FOR 表达式获取下一个值。
- OWNED BY子句是一种LightDB扩展。

## 另见

ALTER SEQUENCE, DROP SEQUENCE

---

# CREATE SERVER

CREATE SERVER — 定义一个新的外部服务器

## Synopsis

```
CREATE SERVER [ IF NOT EXISTS ] server_name [ TYPE 'server_type' ] [ VERSION
'server_version' ]
  FOREIGN DATA WRAPPER fdw_name
  [ OPTIONS ( option 'value' [, ... ] ) ]
```

## 描述

CREATE SERVER定义一个新的外部服务器。 定义该服务器的用户会成为拥有者。

外部服务器通常包装了外部数据包装器用来访问一个外部数据源所需的 连接信息。额外的用户相关的连接信息可以通过用户映射的方式来指定。

服务器名称在数据库中必须唯一。

创建服务器要求所使用的外部数据包装器上的USAGE特权。

## 参数

IF NOT EXISTS

如果已经存在同名的服务器，不要抛出错误。在这种情况下发出一个通知。 请注意，不能保证现有服务器与要创建的服务器类似。

server\_name

要创建的外部服务器的名称。

server\_type

可选的服务器类型，可能对外部数据包装器有用。

server\_version

可选的服务器版本，可能对外部数据包装器有用。

fdw\_name

管理该服务器的外部数据包装器的名称。

OPTIONS ( option 'value' [, ... ] )

这个子句为服务器指定选项。这些选项通常定义该服务器的连接细节， 但是实际的名称和价值取决于该服务器的外部数据包装器。

## 注解

在使用dblink模块时，一个外部服务器的名称可以被 用作dblink\_connect函数的一个参数来指示 连接参数。以这种方式使用外部服务器，需要在其上具有 USAGE特权。

## 示例

创建使用外部数据包装器`postgres_fdw` 的服务器`myserver`:

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'foo',
    dbname 'foodb', port '5432');
```

详见`postgres_fdw`。

## 兼容性

CREATE SERVER符合 ISO/IEC 9075-9 (SQL/MED)。

## 另见

ALTER SERVER, DROP SERVER, CREATE FOREIGN DATA WRAPPER, CREATE FOREIGN TABLE,  
CREATE USER MAPPING

---

# CREATE STATISTICS

CREATE STATISTICS — 定义扩展统计

## Synopsis

```
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
  [ ( statistics_kind [, ... ] ) ]
  ON column_name, column_name [, ...]
  FROM table_name
```

## 描述

CREATE STATISTICS将创建一个新的扩展统计对象，追踪指定表、外部表或物化视图的数据。该统计对象将在当前数据库中创建，被发出该命令的用户所有。

如果给定了模式名（比如，CREATE STATISTICS myschema.mystat ...），那么在给定的模式中创建统计对象。否则在当前模式中创建。统计对象的名称必须与相同模式中的任何其他统计对象不同。

## 参数

IF NOT EXISTS

如果具有相同名称的统计对象已经存在，不会抛出一个错误，只会发出一个提示。请注意，这里只考虑统计对象的名称，不考虑其定义细节。

statistics\_name

要创建的统计对象的名称（可以有模式限定）。

statistics\_kind

在此统计对象中计算的多变量统计种类。目前支持的种类是启用n-distinct统计的ndistinct，启用功能依赖性统计的dependencies，以及启用最常见的值列表的mcv。如果省略该子句，则统计对象中将包含所有支持的统计类型。如果统计信息定义包含任何复杂表达式而不仅仅是简单的列引用，单变量表达式统计会自动构建。有关更多信息，请参阅Section 14.2.4和Section 66.2

column\_name

被计算的统计信息包含的表格列的名称。这里只在建立多变量统计信息时才被允许。至少必须指定两个列名或表达式，它们的顺序是不重要的。

table\_name

包含计算统计信息的列的表的名称（可以是模式限定的）。

## 注意

你必须是表的所有者才能创建读取它的统计对象。不过，一旦创建，统计对象的所有权与基础表无关。

## 示例

用两个功能相关的列创建表t1，即第一列中的值的信息足以确定另一列中的值。然后，在这些列上构建函数依赖关系统计信息：

```

CREATE TABLE t1 (
  a   int,
  b   int
);

INSERT INTO t1 SELECT i/100, i/500
           FROM generate_series(1,1000000) s(i);

ANALYZE t1;

-- 匹配行的数量将被大大低估:
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;

ANALYZE t1;

-- 现在行计数估计会更准确:
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

```

如果没有函数依赖性统计，规划器会认为两个WHERE条件是独立的，并且会将它们的选择性乘以一起，以致得到太小的行数估计。通过这样的统计，规划器认识到WHERE条件是多余的，并且不会低估行数。

创建表t2与两个完全相关的列(包含相同的数据)，并且在这些列上创建一个MCV列表：

```

CREATE TABLE t2 (
  a   int,
  b   int
);

INSERT INTO t2 SELECT mod(i,100), mod(i,100)
           FROM generate_series(1,1000000) s(i);

CREATE STATISTICS s2 (mcv) ON a, b FROM t2;

ANALYZE t2;

-- valid combination (found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 1);

-- invalid combination (not found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 2);

```

MCV列表为计划器提供了关于表中普遍出现的特定值的更详细的信息，以及表中未显示的值组合的选择性上限，允许它在这两种情况下产生更好的估计值。

## 兼容性

SQL标准中没有CREATE STATISTICS命令。

## 又见

ALTER STATISTICS, DROP STATISTICS

---

# CREATE SUBSCRIPTION

CREATE SUBSCRIPTION — 定义一个新的订阅

## Synopsis

```
CREATE SUBSCRIPTION subscription_name
    CONNECTION 'conninfo'
    PUBLICATION publication_name [, ...]
    [ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

## 描述

CREATE SUBSCRIPTION为当前数据库添加一个新的订阅。 订阅名称必须与数据库中任何现有的订阅不同。

订阅表示到发布者的复制连接。因此，此命令不仅在本地目录中添加定义，还会在发布者上创建复制插槽。

在运行此命令的事务提交时，将启动逻辑复制工作器以复制新订阅的数据。

关于订阅和逻辑复制的更多信息，请参见 Section 28.4 和 Chapter 28

## 参数

subscription\_name

新订阅的名称。

CONNECTION 'conninfo'

连接发布者的字符串。详细信息请见Section 32.1.1

PUBLICATION publication\_name

要订阅的发布者上的发布名称。

WITH ( subscription\_parameter [= value] [, ... ] )

该子句指定订阅的可选参数。支持的参数有：

copy\_data (boolean)

指定在复制启动后是否应复制正在订阅的发布中的现有数据。默认值是true。

create\_slot (boolean)

指定该命令是否要在发布者上创建复制槽。默认值是true。

enabled (boolean)

指定订阅是否应该主动复制，或者是否应该只是设置，但尚未启动。默认值是true。

slot\_name (string)

要使用的复制插槽的名称。默认行为是使用订阅名称作为插槽的名称。

当`slot_name`设置为`NONE`时，将不会有复制槽与订阅关联。这在需要稍后手动设置复制槽的情况下会使用。这样的订阅必须同时`enabled`并且`create_slot`设置为`false`。

`synchronous_commit` (enum)

在这个订阅的应用worker进程时，该参数的值会覆盖`synchronous_commit`设置。默认值是`off`。

对于逻辑复制使用`off`是安全的：如果订阅者由于缺少同步而丢失事务，数据将从发布者再次发送。

进行同步逻辑复制时，不同的设置可能是合适的。逻辑复制工作者向发布者报告写入和刷新的位置，当使用同步复制时，发布者将等待实际刷新。这意味着，当订阅用于同步复制时，将订阅者的`synchronous_commit`设置为`off`可能会增加发布服务器上`COMMIT`的延迟。在这种情况下，将`synchronous_commit`设置为`local`或更高是有利的。

`binary` (boolean)

指定订阅是否要求发布者以二进制格式(相对的是文本格式)发送数据。默认值是`false`。即使此选项被启用，也只有具备二进制发送和接收函数的数据类型将以二进制传输。

在进行跨版本复制时，可能会出现发布者具有某些数据类型的二进制发送函数，但订阅器缺乏针对此类型的二进制接收函数的情况。在这种情况下，数据传输将失败，并且`binary`选项不能被使用。

`connect` (boolean)

指定`CREATE SUBSCRIPTION`是否应该连接到发布者。将其设置为`false`将会改变默认值`enabled`、`create_slot`和`copy_data`为`false`。

不允许将`connect`设置为`false`的同时将`enabled`、`create_slot`或`copy_data`设置为`true`。

因为该选项设置为`false`时不会建立连接，因此表没有被订阅，所以当启用订阅后，不会复制任何内容。需要运行`ALTER SUBSCRIPTION ... REFRESH PUBLICATION`才能订阅表。

`streaming` (boolean)

指定是否对此订阅启用进行中事务的流。默认情况下，所有事务都在发布者上完全解码，直到那时才作为整体发送给订阅器。

## 注意

有关如何在订阅和发布实例之间配置访问控制的详细信息，请参见 Section 28.7

创建复制槽时(默认行为)，`CREATE SUBSCRIPTION`不能在事务块内部执行。

如果复制插槽不是作为同一命令的一部分创建的，则创建连接到相同数据库集群的订阅(例如，在同一集群中的数据库之间进行复制或在同一个数据库中进行复制)只能成功。否则，`CREATE SUBSCRIPTION`调用将挂起。要做到这一点，单独创建复制插槽(使用函数`pg_create_logical_replication_slot`和插件名称`pgoutput`)，并使用参数`create_slot = false`创建订阅。这是一个实施限制，可能会在未来的版本中解除。

## 示例

创建一个到远程服务器的订阅，复制发布`mypublication`和`insert_only`中的表，并在提交时立即开始复制：



```
CREATE SUBSCRIPTION mysub
    CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
    PUBLICATION mypublication, insert_only;
```

创建一个到远程服务器的订阅，复制insert\_only发布中的表，并且不开始复制直到稍后启用复制。

```
CREATE SUBSCRIPTION mysub
    CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
    PUBLICATION insert_only
    WITH (enabled = false);
```

## 兼容性

CREATE SUBSCRIPTION是一个LightDB 扩展。

## 又见

ALTER SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

---

# CREATE TABLE

CREATE TABLE — 定义一个新表

## Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED | IMMUTABLE ] TABLE
  [ IF NOT EXISTS ] table_name ( [
    { column_name data_type [ COLLATE collation ] [ column_constraint [enable|
disable] [ ... ] ] [ COMMENT column_comment ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
  [, ... ]
] ) [ INHERITS ( parent_table [, ... ] ) ]
[ lt_partition_option ]
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ lt_table_options ]
[NO DELETE][NO UPDATE][NO DROP][NO TRUNCATE ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED | IMMUTABLE] TABLE
  [ IF NOT EXISTS ] table_name
    OF type_name [ (
      { column_name [ WITH OPTIONS ] [ column_constraint [enable|disable] [ ... ] ]
      | table_constraint }
    [, ... ]
  ) ]
[ lt_partition_option ]
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
[NO DELETE][NO UPDATE][NO DROP][NO TRUNCATE ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED | IMMUTABLE] TABLE
  [ IF NOT EXISTS ] table_name
    PARTITION OF parent_table [ (
      { column_name [ WITH OPTIONS ] [ column_constraint [enable|disable] [ ... ] ]
      | table_constraint }
    [, ... ]
  ) ] { FOR VALUES partition_bound_spec | DEFAULT }
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) }
  [ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ PARTITIONS partitions_value ]
[ INTERVAL interval_value ]
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
[NO DELETE][NO UPDATE][NO DROP][NO TRUNCATE ]
```

其中 `lt_partition_option` 是:

```

{
  PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) }
[ COLLATE collation ] [ opclass ] [, ... ] )
[ PARTITIONS partitions_value ]
[ INTERVAL interval_value ] |
PARTITION BY partition_type (column)
  [ SUBPARTITION BY partition_type (column) ]
  [ SUBPARTITIONS subpartitions_value ]
  [ SUBPARTITION TEMPLATE ( template_spec ) ]
  [...]
  ( partition_spec )
  | [ SUBPARTITION BY partition_type (column) ]
  [...]
  ( partition_spec
  [ ( subpartition_spec
    [(...)]
  ) ]
  )
}

```

并且 `lt_table_options` 是:

```

{ TABLESPACE tablespace_name |
  WITH UPDATE CURRENT_TIMESTAMP |
  WITH PRIMRAY KEY [ smallserial | serial | bigserial ] |
  { DISTRIBUTED BY [HASH | APPEND](column) [ distribute_option ... ] |
  DISTRIBUTED REPLICATED } |
  COMMENT = table_comment |
  { COMPRESS | NOCOMPRESS } |
  { LOGGING | NOLOGGING } |
  STORAGE(value) |
  ENGINE = { INNODB | MYISAM } |
  DEFAULT CHARSET = value |
  COLLATE = value |
  NESTED TABLE column_name STORE AS nested_table_name [...]
}

```

并且 `column_constraint` 是:

```

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  ON UPDATE CURRENT_TIMESTAMP |
  AUTO_INCREMENT |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH
SIMPLE ]
  [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

```

并且 `table_constraint` 是:

```

[ CONSTRAINT constraint_name ]

```

```
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator
  [, ... ] ) index_parameters [ WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn
  [, ... ] ) ]
  [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE referential_action
  ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

并且 like\_option 是:

```
{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | GENERATED |
  IDENTITY | INDEXES | STATISTICS | STORAGE | SEQUENCES | ALL }
```

and partition\_bound\_spec is:

```
IN ( partition_bound_expr [, ...] ) |
FROM ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] )
  TO ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )
```

UNIQUE、PRIMARY KEY以及EXCLUDE约束中的index\_parameters是:

```
[ INCLUDE ( column_name [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

一个EXCLUDE约束中的exclude\_element是:

```
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST |
  LAST } ]
```

where partition\_type is:

```
LIST
| RANGE
| HASH
```

where partition\_spec is:

partition\_element [, ...]

and partition\_element is:

```
DEFAULT PARTITION name
| [PARTITION name] VALUES ( list_value [,...] )
| [PARTITION name]
  START ([datatype] start_value ) [INCLUSIVE | EXCLUSIVE]
  [ END ([datatype] end_value ) [INCLUSIVE | EXCLUSIVE] ]
  [ EVERY ([datatype] [number | INTERVAL] interval_value ) ]
| [PARTITION name]
  END ([datatype] end_value ) [INCLUSIVE | EXCLUSIVE]
  [ EVERY ([datatype] [number | INTERVAL] interval_value ) ]
| [PARTITION name]
  FROM ([datatype] from_value ) [INCLUSIVE | EXCLUSIVE]
  [ TO ([datatype] to_value ) [INCLUSIVE | EXCLUSIVE] ]
  [ EVERY ([datatype] [number | INTERVAL] interval_value ) ]
| [PARTITION name]
  TO ([datatype] to_value ) [INCLUSIVE | EXCLUSIVE]
  [ EVERY ([datatype] [number | INTERVAL] interval_value ) ]
| [PARTITION name] VALUES WITH ( MODULUS numeric_literal,
  REMAINDER numeric_literal )
```

```

| [PARTITION name] VALUES LESS THAN ( { partition_value | MAXVALUE } )
| [PARTITION name]
[ TABLESPACE tablespace ]
[ NOCOMPRESS | COMPRESS ]
where subpartition_spec or template_spec is:
subpartition_element [, ...]
and subpartition_element is:
  DEFAULT SUBPARTITION name
  | [SUBPARTITION name] VALUES ( list_value [,...] )
  | [SUBPARTITION name]
  | START ([datatype] start_value ) [INCLUSIVE | EXCLUSIVE]
  | [ END ([datatype] end_value ) [INCLUSIVE | EXCLUSIVE] ]
  | [ EVERY ([datatype] [number | INTERVAL] interval_value ) ]
  | [SUBPARTITION name]
  | END ([datatype] end_value ) [INCLUSIVE | EXCLUSIVE]
  | [ EVERY ([datatype] [number | INTERVAL] interval_value ) ]
  | [SUBPARTITION name]
  | FROM ([datatype] from_value ) [INCLUSIVE | EXCLUSIVE]
  | [ TO ([datatype] to_value ) [INCLUSIVE | EXCLUSIVE] ]
  | [ EVERY ([datatype] [number | INTERVAL] interval_value) ]
  | [SUBPARTITION name]
  | TO ([datatype] to_value ) [INCLUSIVE | EXCLUSIVE]
  | [ EVERY ([datatype] [number | INTERVAL] interval_value ) ]
  | [SUBPARTITION name] VALUES WITH ( MODULUS numeric_literal,
  | REMAINDER numeric_literal )
[ NOCOMPRESS | COMPRESS ]
where distribute_option is:
  SHARD_COUNT(int_value)
  | COLOCATE_WITH('string_value')
```

## 描述

CREATE TABLE将在当前数据库中创建一个新的、初始为空的表。该表将由发出该命令的用户所拥有。

如果给定了一个模式名（例如CREATE TABLE myschema.mytable ...），那么该表被创建在指定的模式中。否则它被创建在当前模式中。临时表存在于一个特殊的模式中，因此在创建一个临时表时不能给定一个模式名。该表的名称必须与同一个模式中的任何其他表、序列、索引、视图或外部表的名称区分开。

CREATE TABLE也会自动地创建一个数据类型来表示对应于该表一行的组合类型。因此，表不能用同一个模式中任何已有数据类型的名称。

lt\_table\_options 中的任何选项都可以以任何顺序指定。

可选的约束子句指定一个插入或更新操作要成功，新的或更新过的行必须满足的约束（测试）。一个约束是一个 SQL 对象，它帮助以多种方式定义表中的合法值集合。

有两种方式来定义约束：表约束和列约束。一个列约束会作为列定义的一部分定义。一个表约束定义不与一个特定列绑定，并且它可以包含多于一个列。每一个列约束也可以被写作一个表约束，列约束只是一种当约束只影响一列时方便书写的记号习惯。

当指定column\_comment后便会在该列上添加注释。相似地，在表定义的最后指定table\_comment将添加该表的注释。

要能创建一个表，你必须分别具有所有列类型或OF子句中类型的USAGE特权。

## 参数

### TEMPORARY or TEMP

如果指定，该表被创建一个临时表。临时表会在会话结束时自动被删除，或者也可以选择在当前事务结束时删除（见下文的ON COMMIT）。默认搜索路径首先包括临时模式，因此当临时表存在时，不会为新计划选择同名的已有的永久表，除非它们使用模式限定的名称进行引用。在一个临时表上创建的任何索引也自动地变为临时的。

自动清理守护进程不能访问并且因此也不能清理或分析临时表。由于这个原因，应该通过会话的 SQL 命令执行合适的清理和分析操作。例如，如果一个临时表将要被用于复杂的查询，最好在把它填充完毕后在其上运行ANALYZE。

可以选择将GLOBAL或LOCAL写在TEMPORARY或TEMP的前面。这当前在LightDB中没有区别并且已被废弃，见Compatibility。

### UNLOGGED

如果指定，该表被创建一个不受日志记录的表。被写入到不做日志的表中的数据不会被写到预写式日志中（见Chapter 27，这让它们比普通表快非常多。不过，它们在崩溃时是不安全的：一个不做日志的表在一次崩溃或非干净关闭之后会被自动地截断。一个不做日志的表中的内容也不会被复制到后备服务器中。在一个不做日志的表上创建的任何索引也会自动地不被日志记录。

### IF NOT EXISTS

如果一个同名关系已经存在，不要抛出一个错误。在这种情况下会发出一个提示。注意这不保证现有的关系是和将要被创建的表相似的东西。

### table\_name

要被创建的表名（可以选择用模式限定）。请注意：ORACLE模式下创建分区表时，分区表的名字限制为31字符长。

### table\_comment

将被创建的表注释内容。

### OF type\_name

创建一个类型化的表，它的结构取自于指定的组合类型（名字可以选择用模式限定）。一个类型化的表和它的类型绑定在一起，例如如果类型被删除，该表也将被删除（用DROP TYPE ... CASCADE）。

当一个类型化的表被创建时，列的数据类型由底层的组合类型决定而没有在CREATE TABLE命令中直接指定。但是CREATE TABLE命令可以对表增加默认值和约束，并且可以指定存储参数。

### column\_name

列的名称会在新表中被建立。

值得注意的是，如果当前数据库在初始化时指定为mysql模式，同时在lightdb.conf中将'lightdb\_sql\_mode'设置包含'uppercase\_identifier'规则，那么列名不能够出现多次，即使它们大小写不同。该特性将与mysql保持一致，但不同于常规下的lightdb对待列名大小写敏感，也就是说常规下lightdb仅仅禁止列名完全相等。

考虑到oracle模式中的类似情况，与mysql模式略有不同。如果guc参数'lightdb\_oracle\_sql\_mode'在oracle会话内设置为'show\_identifier\_uppercase'，那么即使对列标签使用别名，select语句的执行结果亦为大写。因此仅有对该列名使用

双引号包裹，才可使得显示保持原样。 设置'lightdb\_oracle\_sql\_mode'为''，那么该功能会被禁用并恢复默认模式。

column\_comment

将被创建的表中列的注释内容。

data\_type

列的数据类型。 这可以包括数组 规格。 有关LightDB支持数据类型的详细信息， 请参考Chapter 8

COLLATE collation

COLLATE子句为该列（必须是一种可排序数据类型）赋予一个排序规则。 如果没有指定， 将使用该列数据类型的默认排序规则。

INHERITS ( parent\_table [, ... ] )

可选的INHERITS子句指定一个表的列表， 新表将从其中自动地继承所有列。 父表可以是普通表或者外部表。

INHERITS的使用在新的子表和它的父表之间创建一种持久的关系。 对于父表的模式修改通常也会传播到子表， 并且默认情况下子表的数据会被包括在对父表的扫描中。

如果在多个父表中存在同名的列， 除非父表中每一个这种列的数据类型都能匹配， 否则会报告一个错误。 如果没有冲突， 那么重复列会被融合来形成新表中的一个单一列。 如果新表中的列名列表包含一个也是继承而来的列名， 该数据类型也必须匹配继承的列， 并且列定义会被融合成一个。 如果新表显式地为列指定了任何默认值， 这个默认值将覆盖来自该列继承声明中的默认值。 否则， 任何父表都必须为该列指定相同的默认值， 或者会报告一个错误。

CHECK约束本质上也采用和列相同的方式被融合： 如果多个父表或者新表定义中包含相同的命名CHECK约束， 这些约束必须全部具有相同的检查表达式， 否则将报告一个错误。 具有相同名称和表达式的约束将被融合成一份拷贝。 一个父表中的被标记为NO INHERIT的约束将不会被考虑。 注意新表中一个未命名的CHECK约束将永远不会被融合， 因为那样总是会为它选择一个唯一的名字。

列的STORAGE设置也会从父表复制过来。

如果父表中的列是标识列， 那么该属性不会被继承。 如果需要， 可以将子表中的列声明为标识列。

PARTITION BY { RANGE | LIST | HASH } ( { column\_name | ( expression ) } [ opclass ] [, ... ] )

可选的PARTITION BY子句指定了对表进行分区的策略。 这样创建的表称为分区表。 带括号的列或表达式的列表构成表的分区键。 使用范围或哈希分区时， 分区键可以包含多个列或表达式（最多32个， 但在构建 LightDB时可以更改此限制）， 但对于列表分区， 分区键必须由单个列或表达式组成。

范围和列表分区需要 btree 运算符类， 而哈希分区需要哈希运算符类。 如果没有运算符类被显式指定， 将使用相应类型的默认运算符类； 如果不存在默认运算符类， 则将引发错误。 使用哈希分区时， 所使用的运算符类必须实现支持功能 2（详情请参阅Section 36.16.3）。

分区表被分成多个子表（称为分区）， 它们是使用单独的CREATE TABLE命令创建的。 分区表本身是空的。 插入到表中的数据行将根据分区键中的列或表达式的值路由到分区。 如果没有现有的分区与 newRow 中的值匹配， 则会报告错误。

分区表不支持EXCLUDE约束； 但是， 你可以在各个分区上定义这些约束。

有关表分区的更多讨论，请参阅Section 5.11 ORACLE模式下创建分区时，分区名与子分区名均限制为31字符长。

```
PARTITION OF parent_table { FOR VALUES partition_bound_spec | DEFAULT }
```

将表创建为指定父表的分区。该表建立时，可以使用FOR VALUES创建为特定值的分区，也可以使用DEFAULT创建默认分区。父表中存在的任何索引、约束和用户定义的行级触发器都将克隆到新分区上。

partition\_bound\_spec 必须对应于父表的分区方法和分区键，并且必须不能与该父表的任何现有分区重叠。具有IN的形式用于列表分区，具有FROM和TO的形式用于范围分区，具有WITH的形式用于哈希分区。

partition\_bound\_expr 是任何无变量表达式（不允许子查询、窗口函数、聚合函数和集返回函数）。它的数据类型必须与相应分区键列的数据类型相匹配。表达式在表创建时只计算一次，因此它甚至可以包含易失性表达式，如CURRENT\_TIMESTAMP。

在创建列表分区时，可以指定NULL来表示分区允许分区键列为空。但是，给定父表不能有多于一个这样的列表分区。无法为范围分区指定 NULL。

创建范围分区时，由FROM指定的下限是一个包含范围，而用TO指定的上限是排除范围。也就是说，在FROM列表中指定的值是该分区的相应分区键列的有效值，而TO列表中的值不是。请注意，必须根据按行比较的规则来理解此语句（Section 9.24.5）。例如，给定PARTITION BY RANGE (x, y)，分区范围 FROM (1, 2) TO (3, 4) 允许x=1与任何y>=2，x=2与任何非空y，和x=3与任何y<4。

在创建范围分区以指示列值没有下限或上限时，可以使用特殊值MINVALUE 和MAXVALUE。例如，使用FROM (MINVALUE) TO (10) 定义的分​​区允许任何小于10的值，并且使用FROM (10) TO (MAXVALUE) 定义的分​​区允许任何大于或等于10的值。

创建涉及多个列的范围分区时，将MAXVALUE作为下限的一部分并将 MINVALUE作为上限的一部分也是有意义的。例如，使用 FROM (0, MAXVALUE) TO (10, MAXVALUE) 定义的分​​区允许第一个分区键列大于0且小于或等于10的任何行。类似地，使用FROM ('a', MINVALUE) TO ('b', MINVALUE) 定义的分​​区 允许第一个分区键列以“a”开头的任何行。

请注意，如果MINVALUE或MAXVALUE用于分区边界的一列，则必须为所有后续列使用相同的值。例如，(10, MINVALUE, 0) 不是有效的边界；你应该写(10, MINVALUE, MINVALUE)。

还要注意，某些元素类型，如timestamp，具有“无穷”的概念，这只是另一个可以存储的值。这与MINVALUE和MAXVALUE不同，它们不是可以存储的实际值，而是它们表示值无界的方式。MAXVALUE 可以被认为比任何其他值（包括“无穷”）都大的值，MINVALUE 可以被认为是比任何其他值（包括“负无穷”）都小的值。因此，范围FROM ('infinity') TO (MAXVALUE)不是空的范围；它只允许存储一个值—“infinity”。

如果指定了DEFAULT，则表将创建为父表的默认分区。此选项不适用于哈希分区表。不适合给定父级表的任何其他分区的分区键值将路由到默认分区。

当一个表已有DEFAULT 分区并且要对它添加新分区时，必须扫描默认分区以验证它不包含可能属于新分区的任何行。如果默认分区包含大量行，则速度可能会很慢。如果默认分区是外表或者它具有可证明的不可能包含能放置在新分区中的行的约束，则将略过扫描

当创建哈希分区时，必须指定模数和余数。模数必须是正整数，余数必须是小于模数的非负整数。通常情况下，当初始设置哈希分区表时，应选择一个与分区数相等的模数，并为每个表分配相同的模数和不同的余数（请参阅下方示例）。不过，并不要求每个分区都具有相同的模数，只要求哈希分区表里面的分区中出现的每个模数都是下一个较大模数的因数。这允许以增量的方式增加分区数量而不需要一次移动所有数据。例如，假设你有一个包含 8 个分区的哈希分区表，每个分区有模数8，但发现有必要将分区数



增加到 16 个。您可以拆分其中一个模数-8分区，然后创建两个新的模数-16分区来覆盖键空间的相同部分（一个的余数等于被拆分的分区的余数，另一个的余数等于该值加 8），而后用数据重新填充他们。然后，你可以对每一个余数-8分区重复此操作过程，直到没有剩余。虽然这其中的每个步骤都可能会导致大量的数据移动操作，它仍然要好于建一个全新的表并一次移动全部数据。

分区必须与其所属的分区表的字段名和类型相同。对分区表字段名或类型的修改，将自动传播到所有分区。CHECK约束将自动被每一个分区继承，但是单独的分区可以指定额外的CHECK约束；与父表相同名称和条件的额外约束将被父表约束合并。可以为每个分区分别指定默认值。但是请注意，在通过分区表插入元组时不会应用分区的默认值。

插入分区表中的行将自动路由到正确的分区。如果不存在合适的分区，则会发生错误。

像TRUNCATE这样的操作通常会影响到一个表及其所有继承子级，这些操作将级联到所有分区，但也可在单个分区上执行。请注意，使用DROP TABLE 删除分区需要在父表上采用ACCESS EXCLUSIVE锁。

LIKE source\_table [ like\_option ... ]

LIKE指定新表将从哪一个表自动地复制所有的列名、数据类型以及它们的非空约束。

和INHERITS不同，新表和原始表在创建完成之后是完全分离的。对原始表的更改将不会被应用到新表，并且不可能在原始表的扫描中包括新表的数据。

同样与INHERITS不同，用LIKE拷贝的列和约束不会和相似的命名列及约束融合。如果显式指定了相同的名称或者在另一个LIKE子句中指定了相同的名称，将会发出一个错误。

可选的like\_option子句指定要复制的原始表的附加属性。指定 INCLUDING 复制该属性，指定 EXCLUDING 忽略该属性。EXCLUDING为默认值。如果对同一类型的对象指定了多个规范，则使用最后一个规范。可用的选项包括：

INCLUDING COMMENTS

复制的列、约束和索引的注释将被复制。默认行为是去除注释，从而导致新表中复制的列和约束没有注释。

INCLUDING CONSTRAINTS

CHECK约束将被复制。列约束和表约束之间没有区别。非空约束始终复制到新表。

INCLUDING DEFAULTS

复制列定义的默认表达式将被复制。否则，不会复制默认表达式，从而导致新表中复制的列具有空默认值。注意，复制调用数据库修改函数的默认值，例如nextval，可能在原始表和新表之间创建功能联系。

INCLUDING GENERATED

列定义的任何生成表达式将被复制。默认情况下，新列将是常规基本列。

INCLUDING IDENTITY

已复制列定义的任何标识规范都将被复制。为新表的每个标识列创建一个新序列，与与旧表关联的序列分开。

INCLUDING INDEXES

原始表上的索引、PRIMARY KEY、UNIQUE和 EXCLUDE约束将被建立在新表上。根据默认规则选择新索引和约束的名称，而不考虑原始的命名。（此行为可避免新索引可能出现重复名称失败。）

## INCLUDING STATISTICS

扩展统计信息将复制到新表。

## INCLUDING STORAGE

已复制列定义的STORAGE设置将被复制。默认行为是排除STORAGE设置，从而导致新表中已复制列具有类型规定的默认设置。关于STORAGE设置的更多信息，参见Section 64.2

## INCLUDING SEQUENCES

使用INCLUDING DEFAULTS选项进行转移。当旧表的列使用序列类型（serial, smallserial, bigserial）之一创建时，为新表的每个序列列创建一个新序列，与旧表相关联的序列是独立的。

## INCLUDING ALL

INCLUDING ALL 是选择所有可用的单独选项的缩写形式。（它被用于在INCLUDING ALL之后写单独的EXCLUDING子句，以选择部分指定选项之外的所有选项。）

LIKE子句也能被用来从视图、外部表或组合类型拷贝列定义。不适合的选项（例如来自视图的INCLUDING INDEXES）会被忽略。

## CONSTRAINT constraint\_name

一个列约束或表约束的可选名称。如果该约束被违背，约束名将会出现在错误消息中，这样类似列必须为正的约束名可以用来与客户端应用沟通有用的约束信息（指定包含空格的约束名时需要用到双引号）。如果没有指定约束名，系统将生成一个。

## NOT NULL

该列不允许包含空值。

## NULL

该列允许包含空值。这是默认情况。

这个子句只是提供与非标准 SQL 数据库的兼容。在新的应用中不推荐使用。

## CHECK ( expression ) [ NO INHERIT ]

CHECK指定一个产生布尔结果的表达式，一个插入或更新操作要想成功，其中新的或被更新的行必须满足该表达式。计算出 TRUE 或 UNKNOWN 的表达式就会成功。只要任何一个插入或更新操作的行产生了 FALSE 结果，将报告一个错误异常并且插入或更新不会修改数据库。一个被作为列约束指定的检查约束只应该引用该列的值，而一个出现在表约束中的表达式可以引用多列。

当前，CHECK表达式不能包含子查询，也不能引用当前行的列之外的变量（参见Section 5.4.1）。可以引用系统列tableoid，但不能引用其他系统列。

一个被标记为NO INHERIT的约束将不会传播到子表。

当一个表有多个CHECK约束时，检查完NOT NULL约束后，对于每一行会以它们名称的字母表顺序来进行检查（版本 9.5 之前的LightDB对于CHECK约束不遵从任何特定的引发顺序）。

## [NO DELETE] | [NO UPDATE] | [NO DROP] | [NO TRUNCATE ]

[NO DELETE] | [NO UPDATE] | [NO DROP] | [NO TRUNCATE ]只能在oracle模式下使用这些创建table的参数，并且该表是immutable表。

NO DELETE当IMMUTABLE性质的值指定NO DELETE时，该表的数据将禁止DELETE操作。

NO UPDATE当IMMUTABLE性质的值指定NO UPDATE时，该表的数据将禁止UPDATE操作。

NO DROP当IMMUTABLE性质的值指定NO DROP时，该表的数据将禁止DROP操作。

NO TRUNCATE当IMMUTABLE性质的值指定NO TRUNCATE时，该表的数据将禁止TRUNCATE操作。

DEFAULT default\_expr

DEFAULT子句为出现在其定义中的列赋予一个默认数据。该值是可以使用变量的表达式（特别是，不允许用对其他列的交叉引用）。子查询也是不允许的。默认值表达式的数据类型必须匹配列的数据类型。

默认值表达式将被用在任何没有为该列指定值的插入操作中。如果一列没有默认值，那么默认值为空值。

ON UPDATE CURRENT\_TIMESTAMP

TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP列可以自动初始化并更新为当前时间戳。对于表中的任何TIMESTAMP列，您可以将当前时间戳分配为默认值、ON UPDATE CURRENT\_TIMESTAMP，或两者都分配。但不建议同时使用默认值和ON UPDATE CURRENT\_TIMESTAMP，因为ON UPDATE CURRENT\_TIMESTAMP已经包含了默认的当前时间戳语义。

ON UPDATE CURRENT\_TIMESTAMP只能用于TIMESTAMP类型的列，不能用于其他类型的列。

AUTO\_INCREMENT

该子句创建一个自增列。该列无法被写入，它将隐式附加一个序列，并且新行中的该列将自动分配从序列中分配的值。这样的列隐式为 NOT NULL。

该子句等同于GENERATED ALWAYS AS IDENTITY

GENERATED ALWAYS AS ( generation\_expr ) STORED

此子句将列创建为generated column。列无法被写入，读取时将返回指定表达式的结果。

关键字STORED表示将在写入时计算列并将存储在磁盘上。

生成表达式可以引用表中的其他列，但不能引用其他生成的列。使用的任何函数和运算符都必须是不可改变的。不允许引用其他表。

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence\_options ) ]

该子句将列创建为标识列。它将拥有一个隐式序列附加到它，并且新行中的列将自动从分配给它的序列中获取值。这样的列为隐式NOT NULL。

子句ALWAYS和BY DEFAULT确定如何在 INSERT和UPDATE命令中明确处理用户指定的值。

在INSERT命令中，如果选择了ALWAYS，则仅当 INSERT 语句指定OVERRIDING SYSTEM VALUE时才接受用户指定的值。如果选择BY DEFAULT，则用户指定的值优先。有关详细信息，请参阅INSERT。（在COPY命令中，无论此设置如何，始终使用用户指定的值。）

在UPDATE命令中，如果选择了ALWAYS，则将列更新为除DEFAULT之外的任何值都将被拒绝。如果选择BY DEFAULT，则该列可以正常更新。（UPDATE命令没有OVERRIDING子句。）

可选的sequence\_options子句可用于覆盖序列的选项。有关详细信息，请参见CREATE SEQUENCE。

UNIQUE (列约束)

UNIQUE ( column\_name [, ... ] ) [ INCLUDE ( column\_name [, ...] ) ] (表约束)

UNIQUE约束指定一个表中的一列或多列组成的组包含唯一的值。唯一表约束的行为与唯一列约束的行为相同，只是表约束能够跨越多列。约束因此强制在这些列中的至少一列的任何两行必须不同。

对于一个唯一约束的目的来说，空值不被认为是相等的。

每一个唯一表约束将命名一个列的集合，并且它与该表上任何其他唯一或主键约束所命名的列集合都不相同。（否则，冗余的唯一约束将被丢弃。）

在为多级分区层次结构建立唯一约束时，目标分区表的分区键中的所有列，以及那些由它派生的所有分区表，必须被包含在约束定义中。

添加唯一约束将自动在用于约束的列或列组上创建唯一的 btree索引。

可选 INCLUDE子句向该索引添加一个或多个列是简单的“payload”：在它们上面唯一性是不强制的，并且该索引不能基于这些列搜索。然而它们可以通过一个仅对索引的扫描检索。请注意虽然约束在包含的列上是非强制的，但是它仍然依赖于它们。因此，这样的列上的某些操作（例如DROP COLUMN）可能会导致级联约束和索引删除。

PRIMARY KEY (列约束)

PRIMARY KEY ( column\_name [, ... ] ) [ INCLUDE ( column\_name [, ...] ) ] (表约束)

PRIMARY KEY约束指定表的一个或者多个列只能包含唯一（不重复）、非空的值。一个表上只能指定一个主键，可以作为列约束或表约束。

主键约束所涉及的列集合应该不同于同一个表上定义的任何唯一约束的列集合（否则，该唯一约束是多余的并且会被丢弃）。

PRIMARY KEY强制的数据约束可以看成是UNIQUE和NOT NULL的组合，然而，把一组列标识为主键也为模式设计提供了元数据，因为主键标识其他表可以依赖这一个列集合作为行的唯一标识符。

当放到分区表上时，PRIMARY KEY 约束共享前面描述的UNIQUE 约束的限制。

添加PRIMARY KEY约束将自动在用于约束的列或列组上创建唯一的 btree 索引。

可选 INCLUDE子句向该索引添加一个或多个列是简单的“payload”：唯一性对它们是非强制的，并且该索引不能基于这些列搜索。然而它们可以通过一个仅对索引的扫描检索。注意虽然包含的列的约束的唯一性是非强制的，但仍依赖于它们。因此，这样的列上的某些操作（例如DROP COLUMN）可能会导致级联约束和索引删除。

EXCLUDE [ USING index\_method ] ( exclude\_element WITH operator [, ... ] )  
index\_parameters [ WHERE ( predicate ) ]

EXCLUDE子句定一个排除约束，它保证如果任意两行在指定列或表达式上使用指定操作符进行比较，不是所有的比较都将会返回TRUE。如果所有指定的操作符都测试相等，这就等价于一个UNIQUE约束，尽管一个普通的唯一约束将更快。不过，排除约束能够指定比简单相等更通用的约束。例如，你可以使用&&操作符指定一个约束，要求表中没有两行包含相互覆盖的圆（见 Section 8.8）。

排除约束使用一个索引实现，这样每一个指定的操作符必须与用于索引访问方法index\_method的一个适当的操作符类（见Section 11.10 相关联。操作符被要求是交换的。每一个exclude\_element可以选择性地指定一个操作符类或者顺序选项，这些在CREATE INDEX中有完整描述。

访问方法必须支持amgettuple（见Chapter 58，目前这意味着GIN无法使用。尽管允许，但是在一个排除约束中使用 B-树或哈希索引没有意义，因为它无法做得比一个普通唯一索引更出色。因此在实践中访问方法将总是GiST。

predicate允许你在该表的一个子集上指定一个排除约束。在内部这会创建一个部分索引。注意在为此周围的圆括号是必须的。

```
REFERENCES reftable [ ( refcolumn ) ] [ MATCH matchtype ] [ ON DELETE
referential_action ] [ ON UPDATE referential_action ] (column constraint)
FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
[ MATCH matchtype ] [ ON DELETE referential_action ] [ ON UPDATE referential_action ]
] (表约束)
```

这些子句指定一个外键约束，它要求新表的一列或一个列的组必须只包含能匹配被引用表的某个行在被引用列上的值。如果refcolumn列表被忽略，将使用reftable的主键。被引用列必须是被引用表中一个非可延迟唯一约束或主键约束的列。用户必须在被引用的表（或整个表，或特定的引用列）上拥有REFERENCES权限。增加的外键约束需要SHARE ROW EXCLUSIVE 锁定引用的表。注意外键约束不能在临时表和永久表之间定义。

被插入到引用列的一个值会使用给定的匹配类型与被引用表的值进行匹配。有三种匹配类型：MATCH FULL、MATCH PARTIAL以及MATCH SIMPLE（这是默认值）。MATCH FULL将不允许一个多列外键中的一列为空，除非所有外键列都是空；如果它们都是空，则不要求该行在被引用表中有一个匹配。MATCH SIMPLE允许任意外键列为空，如果任一为空，则不要求该行在被引用表中有一个匹配。MATCH PARTIAL现在还没有被实现（当然，NOT NULL约束能被应用在引用列上来组织这些情况发生）。

另外，当被引用列中的数据被改变时，在这个表的列中的数据上可以执行特定的动作。ON DELETE指定当被引用表中一个被引用行被删除时要执行的动作。同样，ON UPDATE指定当被引用表中一个被引用列被更新为新值时要执行的动作。如果该行被更新，但是被引用列并没有被实际改变，不会做任何动作。除了NO ACTION检查之外的引用动作不能被延迟，即便该约束被声明为可延迟的。对每一个子句可能有以下动作：

NO ACTION

产生一个错误指示删除或更新将会导致一个外键约束违背。如果该约束被延迟，并且仍存在引用行，这个错误将在约束检查时被产生。这是默认动作。

RESTRICT

产生一个错误指示删除或更新将会导致一个外键约束违背。这个动作与NO ACTION相同，不过该检查不是可延迟的。

CASCADE

删除任何引用被删除行的行，或者把引用列的值更新为被引用列的新值。

SET NULL

将引用列设置为空。

SET DEFAULT

设置引用列为它们的默认值（如果该默认值非空，在被引用表中必须有一行匹配该默认值，否则该操作将会失败）。

如果被引用列被频繁地更改，最好在引用列上加上一个索引，这样与外键约束相关的引用动作能够更高效地被执行。

DEFERRABLE

NOT DEFERRABLE

这个子句控制该约束是否能被延迟。一个不可延迟的约束将在每一次命令后立刻被检查。可延迟约束的检查将被推迟到事务结束时进行（使用SET CONSTRAINTS命令）。NOT DEFERRABLE是默认值。当前，只有UNIQUE、PRIMARY KEY、EXCLUDE以及REFERENCES（外键）约束接受这个子句。NOT NULL以及CHECK约束是不可延迟的。注意在包括ON CONFLICT DO UPDATE子句的INSERT语句中，可延迟约束不能被用作冲突裁判者。

INITIALLY IMMEDIATE  
INITIALLY DEFERRED

如果一个约束是可延迟的，这个子句指定检查该约束的默认时间。如果该约束是INITIALLY IMMEDIATE，它会在每一个语句之后被检查。这是默认值。如果该约束是INITIALLY DEFERRED，它只会在事务结束时被检查。约束检查时间可以用SET CONSTRAINTS命令修改。

USING method

此可选子句指定用于存储新表内容的表访问方法；该方法需要的是类型TABLE的访问方法。详见 Chapter 57。如果未指定此选项，则为新表选择默认表访问方法。详见default\_table\_access\_method。

WITH UPDATE CURRENT\_TIMESTAMP

该子句为表指定了一个可选的隐含的“ltaut”列。ltaut列可以像TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP列一样自动初始化并更新为当前时间戳。它可以自动初始化并更新为当前时间戳。如果使用了WITH UPDATE CURRENT\_TIMESTAMP，则不允许使用ON UPDATE CURRENT\_TIMESTAMP，反之亦然。

请注意，在创建分布式表时不支持使用WITH UPDATE CURRENT\_TIMESTAMP。

WITH PRIMARY KEY [ smallserial | serial | bigserial ]

该子句为表指定了一个可选的隐含的“ltapk”列。“ltapk”列可以自动递增，并且可以指定为主键约束。如果表已经有了主键，则不允许使用WITH PRIMARY KEY。

在使用CREATE TABLE、CREATE TABLE AS或SELECT INTO创建表时，不允许将“ltapk”作为新表的显式列名。（除非在CREATE TABLE中使用like\_option时包括INDEXES和DEFAULTS）

请注意，在创建分布式表时不支持使用WITH PRIMARY KEY。

DISTRIBUTED REPLICATED

该子句创建引用表，请确保Canopy扩展已准备就绪。

请注意，只有哈希分布类型支持创建分布式表，不支持创建只读表和临时表。如果您要创建分布式表，则仅在分布列是主键的一部分时才强制执行主键约束。

DISTRIBUTED BY [HASH | APPEND](column) [ distribute\_option ... ]

该子句创建一个分布式表，请确保Canopy扩展已准备就绪。分布列为“column”，分布类型为“hash”或“append”，默认为“hash”。

distribute\_option可以指定shard\_count选项或colocate\_with选项。shard\_count是可选的，用于选择要分发的表的分片。colocate\_with是可选的，是当前分布式表将与之共存的表。可能的值包括“default”，“none”以启动新的共存组，或另一个表的名称，该表将与之共存。

请注意，只有哈希分布类型支持创建分布式表，不支持创建只读表和临时表。如果您要创建分布式表，则仅在分布列是主键的一部分时才强制执行主键约束。

WITH ( storage\_parameter [= value] [, ... ] )

这个子句为一个表或索引指定可选的存储参数，详见Storage Parameters。为了向后兼容性，表的WITH子句还可以包括OIDS=FALSE以便指定新表的行不应包含OIDS（对象标识符），OIDS=TRUE不再受支持。

WITHOUT OIDS

这是向后兼容的语法，用于声明表WITHOUT OIDS，不再支持创建表WITH OIDS。

## ON COMMIT

临时表在一个事务块结束时的行为由ON COMMIT控制。三种选项是：

## PRESERVE ROWS

在事务结束时不采取特殊的动作。这是默认行为。

## DELETE ROWS

在每一个事务块结束时将删除临时表中的所有行。实质上，在每一次提交时会完成一次自动的TRUNCATE。当应用于分区表上时，这不会级联到它的分区。

## DROP

在当前事务块结束时将删除临时表。当在分区表上使用时，这个操作会删除他的分区，而在具有继承子级的表上使用时，它将删除依赖的子级。

COMPRESS  
LOGGING

为了兼容Oracle数据库语法，新增了四个非保留关键字COMPRESS, LOGGING, NOCOMPRESS, NOLOGGING。这四个关键字仅是语法糖，没有实现实际的功能。这四个关键字是可选的，如果必须要使用，请放在TABLESPACE关键字之前。

## STORAGE

为了兼容Oracle数据库语法，CREATE TABLE语句支持STORAGE关键字。STORAGE关键字后面括号内的内容是被忽略的。STORAGE关键字是可选的，如果一定要使用，请在COMPRESS LOGGING关键字和TABLESPACE关键字之间使用。

## ENGINE

ENGINE 关键字只是为了在语法上兼容 MySQL 数据库

## INNODB

大小写不敏感

## MYISAM

大小写不敏感

## DEFAULT CHARSET = ...

关键字 DEFAULT CHARSET 只是为了在语法上兼容 MySQL 数据库。注意：DEFAULT CHARACTER SET 还没有被 lightdb 支持。

## COLLATE = ...

关键字 COLLATE 只是为了在语法上兼容 MySQL 数据库。

## TABLESPACE tablespace\_name

tablespace\_name是新表要创建于其中的表空间名称。如果没有指定，将参考default\_tablespace，或者如果表是临时的则参考temp\_tablespaces。对于分区表，由于表本身不需要存储，指定表空间将 default\_tablespace作为默认表空间覆盖，在显式指定其他表空间时用于任何新创建的分区。

## USING INDEX TABLESPACE tablespace\_name

这个子句允许选择与一个UNIQUE、PRIMARY KEY或者EXCLUDE约束相关的索引将被创建在哪个表空间中。如果没有指定，将参考default\_tablespace，或者如果表是临时的则参考temp\_tablespaces。

PARTITIONS partitions\_value

Oracle 兼容的语法用于创建哈希分区。需要将配置更改为 Oracle 兼容模式。

SUBPARTITIONS subpartitions\_value

Oracle 兼容的语法用于创建哈希子分区。目前，仅支持将范围作为主分区、列表作为子分区和哈希作为子分区。需要将配置更改为 Oracle 兼容模式。

INTERVAL interval\_value

Oracle 兼容的语法用于创建范围分区。支持两个时间转换函数：NUMTOYMINTERVAL() 和 NUMTODSINTERVAL()。目前，仅支持时间间隔分区。需要将配置更改为 Oracle 兼容模式。

NESTED TABLE column\_name STORE AS nested\_table\_name

Oracle 兼容语法用于创建带嵌套表类型的表。目前仅做语法兼容，在未来可能做功能支持。

enable

disable

为了兼容Oracle数据库的语法，执行create table语句时列约束支持enable/disable关键字。这个功能仅仅是语法糖。

## 存储参数

WITH子句能够为表或与一个UNIQUE、PRIMARY KEY或者EXCLUDE约束相关的索引指定存储参数。用于索引的存储参数已经在CREATE INDEX中介绍过。当前可用于表的存储参数在下文中列出。如下文所示，对于很多这类参数，都有一个名字带有toast.前缀的附加参数，它能被用来控制该表的二级TOAST表（如果存在）的行为（关于 TOAST 详见Section 64.2）。如果一个表的参数值被设置但是相应的toast.参数没有被设置，那么 TOAST 表将使用该表的参数值。不支持为分区表指定这些参数，但可以为单个叶子分区指定它们。

fillfactor (integer)

一个表的填充因子是一个 10 到 100 之间的百分数。100（完全填满）是默认值。当一个较小的填充因子被指定时，INSERT操作会把表页面只填满到指定的百分比，每个页面上剩余的空间被保留给该页上行的更新。这就让UPDATE有机会把一行的已更新版本放在与其原始版本相同的页面上，这比把它放在一个不同的页面上效率更高。对于一个项从来不会被更新的表来说，完全填满是最好的选择，但是在更新繁重的表上则较小的填充因子更合适。这个参数不能对 TOAST 表设置。

toast\_tuple\_target (integer)

在我们尝试压缩和/或将长列值移动到TOAST表中之前，toast\_tuple\_target指定需要的最小元组长度，也是在toasting开始时尝试减少长度的目标长度。这会影响标记为External（用于移动）、Main（用于压缩）或Extended（用于两者）的列，并且仅适用于新元组。对现有行没有影响。这仅影响标记为“外部”或“扩展”的列，并且仅适用于新元组 - 对现有行没有影响。默认情况下此参数设置为允许每个块至少 4 个元组，默认块大小为 2040 字节。有效值介于 128 字节和(块大小-标头)之间，默认大小为 8160 字节。更改此值对于非常短或非常长的行可能没有用处。请注意默认设置通常接近最佳状态，在某些情况下设置此参数可能会产生负面影响。不能对TOAST表设置此参数。

parallel\_workers (integer)

这个参数设置用于辅助并行扫描这个表的工作者数量。如果没有设置这个参数，系统将基于关系的尺寸来决定一个值。规划者或使用并行扫描的实用程序选择的工作者数量可能会比较少，例如max\_worker\_processes的设置较小就是一种可能的原因。



`autovacuum_enabled, toast.autovacuum_enabled` (boolean)

为一个特定的表启用或者禁用自动清理守护进程。如果为真，自动清理守护进程将遵照Section 22.1.6中讨论的规则在这个表上执行自动的VACUUM或者ANALYZE操作。如果为假，这个表不会被自动清理，不过为了阻止事务 ID 回卷时还是会对其进行自动的清理。有关回卷阻止请见Section 22.1.5 如果`autovacuum`参数为假，自动清理守护进程根本就不会运行（除非为了阻止事务 ID 回卷），设置独立的表存储参数也不会覆盖这个设置。因此显式地将这个存储参数设置为`true`很少有大的意义，只有设置为`false`才更有用。

`vacuum_index_cleanup, toast.vacuum_index_cleanup` (boolean)

当VACUUM在此表上运行时强制或禁用索引清理。默认值为AUTO。用OFF，索引清理被禁用，用ON则被启用，而采用AUTO，决定被动态做出，每次VACUUM运行时。动态行为允许VACUUM不需要的扫描索引以移除很少的死元组。强制禁用全部索引清理可以显著加快VACUUM，但如果表修改很频繁，也可能导致索引严重膨胀。VACUUM的INDEX\_CLEANUP参数，如果指定，将覆盖此选项的值。

`vacuum_truncate, toast.vacuum_truncate` (boolean)

启用或禁用vacuum以尝试截断此表末尾的任何空页。默认值为true。如果true，VACUUM和autovacuum将执行截断，截断页的磁盘空间将返回到操作系统。请注意，截断需要 ACCESS EXCLUSIVE在表上锁定。VACUUM的TRUNCATE参数，如果指定，覆盖此选项的值。

`autovacuum_vacuum_threshold, toast.autovacuum_vacuum_threshold` (integer)

`autovacuum_vacuum_threshold`参数对于每个表的值。

`autovacuum_vacuum_scale_factor, toast.autovacuum_vacuum_scale_factor` (floating point)

`autovacuum_vacuum_scale_factor`参数对于每个表的值。

`autovacuum_vacuum_insert_threshold, toast.autovacuum_vacuum_insert_threshold` (integer)

`autovacuum_vacuum_insert_threshold`参数对于每个表的值。特殊值-1可用于禁用表上的插入为空。

`autovacuum_vacuum_insert_scale_factor, toast.autovacuum_vacuum_insert_scale_factor` (floating point)

`autovacuum_vacuum_insert_scale_factor`参数对于每个表的值。

`autovacuum_analyze_threshold` (integer)

`autovacuum_analyze_threshold`参数对于每个表的值。

`autovacuum_analyze_scale_factor` (floating point)

`autovacuum_analyze_scale_factor`参数对于每个表的值。

`autovacuum_vacuum_cost_delay, toast.autovacuum_vacuum_cost_delay` (floating point)

`autovacuum_vacuum_cost_delay`参数对于每个表的值。

`autovacuum_vacuum_cost_limit, toast.autovacuum_vacuum_cost_limit` (integer)

`autovacuum_vacuum_cost_limit`参数对于每个表的值。

`autovacuum_freeze_min_age`, `toast.autovacuum_freeze_min_age` (integer)

`vacuum_freeze_min_age`参数对于每个表的值。注意自动清理将忽略超过系统范围`autovacuum_freeze_max_age`参数一半的针对每个表的`autovacuum_freeze_min_age`参数。

`autovacuum_freeze_max_age`, `toast.autovacuum_freeze_max_age` (integer)

`autovacuum_freeze_max_age`参数对于每个表的值。注意自动清理将忽略超过系统范围参数（只能被设置得较小）一半的针对每个表的`autovacuum_freeze_max_age`参数。

`autovacuum_freeze_table_age`, `toast.autovacuum_freeze_table_age` (integer)

`vacuum_freeze_table_age`参数对于每个表的值。

`autovacuum_multixact_freeze_min_age`, `toast.autovacuum_multixact_freeze_min_age` (integer)

`vacuum_multixact_freeze_min_age`参数对于每个表的值。注意自动清理将忽略超过系统范围`autovacuum_multixact_freeze_max_age`参数一半的针对每个表的`autovacuum_multixact_freeze_min_age`参数。

`autovacuum_multixact_freeze_max_age`, `toast.autovacuum_multixact_freeze_max_age` (integer)

`autovacuum_multixact_freeze_max_age`参数对于每个表的值。注意自动清理将忽略超过系统范围参数（只能被设置得较小）一半的针对每个表的`autovacuum_multixact_freeze_max_age`参数。

`autovacuum_multixact_freeze_table_age`,  
`toast.autovacuum_multixact_freeze_table_age` (integer)

`vacuum_multixact_freeze_table_age`参数对于每个表的值。

`log_autovacuum_min_duration`, `toast.log_autovacuum_min_duration` (integer)

`log_autovacuum_min_duration`参数对于每个表的值。

`user_catalog_table` (boolean)

声明该表是一个用于逻辑复制目的的额外的目录表。详见Section 45.6.2 不能对 TOAST 表设置这个参数。

## 注解

LightDB为每一个唯一约束和主键约束创建一个索引来强制唯一性。因此，没有必要显式地为主键列创建一个索引（详见CREATE INDEX）。

在当前的实现中，唯一约束和主键不会被继承。这使得继承和唯一约束的组合相当不正常。

一个表不能有超过 1600 列（实际上，由于元组长度限制，有效的限制通常更低）。

## 例子

创建表`films`和表`distributors`：

```
CREATE TABLE films (
    code      char(5) CONSTRAINT firstkey PRIMARY KEY,
    title     varchar(40) NOT NULL,
```

```

    did          integer NOT NULL,
    date_prod    date,
    kind         varchar(10),
    len          interval hour to minute
);

```

```

CREATE TABLE distributors (
    did          integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    name        varchar(40) NOT NULL CHECK (name <> '')
);

```

创建一个二维数组的表:

```

CREATE TABLE array_int (
    vector      int[][]
);

```

为表films定义一个唯一表约束。唯一表约束能够被定义在表的一列或多列上:

```

CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);

```

定义一个列检查约束:

```

CREATE TABLE distributors (
    did         integer CHECK (did > 100),
    name        varchar(40)
);

```

定义部分列的注释及表注释:

```

CREATE TABLE films (
    code        char(5) COMMENT 'code of films',
    title       varchar(40),
    did         integer CHECK (did > 100) COMMENT 'comment for did',
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute COMMENT 'length of the films',
    CONSTRAINT production UNIQUE(date_prod)
) COMMENT = 'films table';

```

定义一个表检查约束:

```

CREATE TABLE distributors (
    did         integer,
    name        varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);

```

为表films定义一个主键表约束:

```
CREATE TABLE films (
    code      char(5),
    title     varchar(40),
    did       integer,
    date_prod date,
    kind      varchar(10),
    len       interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code, title)
);
```

为表distributors定义一个主键约束。下面的两个例子是等价的，第一个使用表约束语法，第二个使用列约束语法:

```
CREATE TABLE distributors (
    did       integer,
    name      varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did       integer PRIMARY KEY,
    name      varchar(40)
);
```

为列name赋予一个文字常量默认值，安排列did的默认值是从一个序列对象中选择下一个值产生，并且让modtime的默认值是该行被插入的时间:

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

在表distributors上定义两个NOT NULL列约束，其中之一被显式给定了一个名称:

```
CREATE TABLE distributors (
    did       integer CONSTRAINT no_null NOT NULL,
    name      varchar(40) NOT NULL
);
```

在表 distributors 上定义一个 ON UPDATE CURRENT\_TIMESTAMP 列，其中一个列被显式地命名:

```
CREATE TABLE distributors (
    did       int,
    modtime   timestamp ON UPDATE CURRENT_TIMESTAMP
);
```

创建表 distributors，并定义 WITH UPDATE CURRENT\_TIMESTAMP。

```
CREATE TABLE distributors (
    did       int,
```

```
) WITH UPDATE CURRENT_TIMESTAMP;
```

在表 `distributors` 上定义 `AUTO_INCREMENT` 列约束，其中一个列被显式地命名：

```
CREATE TABLE distributors (  
    did    int AUTO_INCREMENT,  
    name   varchar(40) NOT NULL  
);
```

为`name`列定义一个唯一约束：

```
CREATE TABLE distributors (  
    did    integer,  
    name   varchar(40) UNIQUE  
);
```

同样的唯一约束用表约束指定：

```
CREATE TABLE distributors (  
    did    integer,  
    name   varchar(40),  
    UNIQUE(name)  
);
```

创建相同的表，指定表和它的唯一索引指定 70% 的填充因子：

```
CREATE TABLE distributors (  
    did    integer,  
    name   varchar(40),  
    UNIQUE(name) WITH (fillfactor=70)  
)  
WITH (fillfactor=70);
```

创建表`circles`，带有一个排除约束阻止任意两个圆重叠：

```
CREATE TABLE circles (  
    c circle,  
    EXCLUDE USING gist (c WITH &&)  
);
```

在表空间`diskvol1`中创建表`cinemas`：

```
CREATE TABLE cinemas (  
    id serial,  
    name text,  
    location text  
)  
TABLESPACE diskvol1;
```

创建一个组合类型以及一个类型化的表：

```
CREATE TYPE employee_type AS (name text, salary numeric);
```

```
CREATE TABLE employees OF employee_type (  
    PRIMARY KEY (name),
```

```

    salary WITH OPTIONS DEFAULT 1000
);

```

创建一个范围分区表:

```

CREATE TABLE measurement (
    logdate      date not null,
    peaktemp     int,
    unitsales    int
) PARTITION BY RANGE (logdate);

```

创建在分区键中具有多个列的范围分区表:

```

CREATE TABLE measurement_year_month (
    logdate      date not null,
    peaktemp     int,
    unitsales    int
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate), EXTRACT(MONTH FROM logdate));

```

创建列表分区表:

```

CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));

```

建立哈希分区表:

```

CREATE TABLE orders (
    order_id     bigint not null,
    cust_id      bigint not null,
    status       text
) PARTITION BY HASH (order_id);

```

创建范围分区表的分区:

```

CREATE TABLE measurement_y2016m07
    PARTITION OF measurement (
        unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');

```

使用分区键中的多个列创建范围分区表的几个分区:

```

CREATE TABLE measurement_ym_older
    PARTITION OF measurement_year_month
    FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);

```

```

CREATE TABLE measurement_ym_y2016m11
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 11) TO (2016, 12);

```

```

CREATE TABLE measurement_ym_y2016m12
    PARTITION OF measurement_year_month

```

```
FOR VALUES FROM (2016, 12) TO (2017, 01);
```

```
CREATE TABLE measurement_ym_y2017m01
PARTITION OF measurement_year_month
FOR VALUES FROM (2017, 01) TO (2017, 02);
```

创建列表分区表的分区：

```
CREATE TABLE cities_ab
PARTITION OF cities (
CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b');
```

创建本身是分区的列表分区表的分区，然后向其添加分区：

```
CREATE TABLE cities_ab
PARTITION OF cities (
CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b') PARTITION BY RANGE (population);
```

```
CREATE TABLE cities_ab_10000_to_100000
PARTITION OF cities_ab FOR VALUES FROM (10000) TO (100000);
```

建立哈希分区表的分区：

```
CREATE TABLE orders_p1 PARTITION OF orders
FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE orders_p2 PARTITION OF orders
FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE orders_p3 PARTITION OF orders
FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE orders_p4 PARTITION OF orders
FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

建立默认分区：

```
CREATE TABLE cities_partdef
PARTITION OF cities DEFAULT;
```

创建范围分区表：

```
CREATE TABLE sales (
id int,
date date
) PARTITION BY RANGE (date)
(
START (date '2020-01-01') INCLUSIVE
END (date '2021-01-01') EXCLUSIVE
EVERY (INTERVAL '1 day')
);
```

创建范围分区表 (Oracle 兼容语法)： 需要将配置更改为 Oracle 兼容模式。

```
CREATE TABLE oracle_partition_range
(
```

```

a int,
b float,
c date,
d timestamp
)PARTITION BY RANGE (c)
(
PARTITION p1 VALUES LESS THAN(to_date('2022-01-01', 'yyyy-mm-dd')),
PARTITION p2 VALUES LESS THAN(to_date('2022-02-01', 'yyyy-mm-dd')),
PARTITION p3 VALUES LESS THAN(to_date('2022-03-01', 'yyyy-mm-dd'))
);

```

创建列表分区表:

```

CREATE TABLE rank (
    id int,
    gender char(1)
) PARTITION BY LIST (gender)
(
PARTITION girls VALUES ('F'),
PARTITION boys VALUES ('M'),
DEFAULT PARTITION other
);

```

创建包含列表默认分区的基于列表分区的分区表 (Oracle 兼容语法): 需要将配置更改为 Oracle 兼容模式。

```

CREATE TABLE l(a int) partition by list(a) (
    partition l_1 values(1),
    partition l_def values(default)
);

```

创建哈希分区表:

```

CREATE TABLE thash (
    a int,
    b int,
    c text
) PARTITION BY HASH(a, b)
(
PARTITION p1 VALUES WITH (MODULUS 3, REMAINDER 0),
PARTITION p2 VALUES WITH (MODULUS 3, REMAINDER 1),
PARTITION p3 VALUES WITH (MODULUS 3, REMAINDER 2)
);

```

创建哈希分区表 (Oracle 兼容语法): 需要将配置更改为 Oracle 兼容模式。

```

CREATE TABLE oracle_hash1
(
a int,
b float,
c VARCHAR(20),
d DATE,
e timestamp
) PARTITION BY HASH (a)
(
PARTITION p1,

```



```

PARTITION p2,
PARTITION p3
);
CREATE TABLE oracle_hash2
(
  a int,
  b float,
  c VARCHAR(20),
  d DATE,
  e timestamp
) PARTITION BY HASH (a) PARTITIONS 3;

```

创建范围间隔分区表 (Oracle 兼容语法)： 需要将配置更改为 Oracle 兼容模式。

```

CREATE TABLE lt_oracle_intervall
(
  a int,
  b float,
  c date,
  d timestamp,
  e varchar(20)
)PARTITION BY RANGE (c) INTERVAL (numtoyminterval(3, 'month'))
(
  PARTITION p1 VALUES LESS THAN(to_date('2022-01-01', 'yyyy-mm-dd'))
);
CREATE TABLE lt_oracle_interval2
(
  a int,
  b float,
  c date,
  d timestamp,
  e varchar(20)
)PARTITION BY RANGE (c) INTERVAL (numtodsinterval(3, 'day'))
(
  PARTITION p1 VALUES LESS THAN(to_date('2022-01-01', 'yyyy-mm-dd'))
);

```

创建子分区:

```

CREATE TABLE sub_partition_list (a text) PARTITION BY LIST(a)
(PARTITION p1 VALUES ('1','2','3'));
CREATE TABLE sub_partition_range (a int) PARTITION BY RANGE(a)
(PARTITION p1 FROM (0) TO (10));
CREATE TABLE sub_partition_range2 (a int) PARTITION BY RANGE(a)
(PARTITION p1 START (0) END (10));
CREATE TABLE sub_partition_hash (a int) PARTITION BY HASH(a)
(
  PARTITION p1 VALUES WITH (MODULUS 3, REMAINDER 0),
  PARTITION p2 VALUES WITH (MODULUS 3, REMAINDER 1),
  PARTITION p3 VALUES WITH (MODULUS 3, REMAINDER 2)
);
CREATE TABLE sub_partition_sp (
  a int, b int, c text
) PARTITION BY RANGE(a)
SUBPARTITION BY LIST (c)
SUBPARTITION TEMPLATE (
  VALUES ('0000', '0001', '0002'),

```

```

VALUES ('0003')
)
(
PARTITION p1 FROM (0) TO (12),
PARTITION p2 FROM (12) TO (20)
);

```

创建子分区 (Oracle 兼容语法) :

```

CREATE TABLE lt_oracle_partition_rh
(
transaction_id number,
item_id number(8) not null,
item_description VARCHAR(300),
transaction_date DATE
)
PARTITION BY RANGE(transaction_date)
SUBPARTITION BY HASH(transaction_id)
SUBPARTITIONS 10
(
PARTITION P1 VALUES LESS THAN(to_date('2020-01-01','yyyy-mm-dd')),
PARTITION P2 VALUES LESS THAN(to_date('2021-01-01','yyyy-mm-dd')),
PARTITION P3 VALUES LESS THAN(to_date('2022-01-01','yyyy-mm-dd'))
);

```

创建基于范围-列表复合分区, 且包含列表默认分区的分区表 (Oracle 兼容语法) : 需要将配置更改为 Oracle 兼容模式。

```

CREATE TABLE RL(a int, b int) PARTITION BY RANGE(a)
SUBPARTITION BY LIST(b)
SUBPARTITION TEMPLATE
(
SUBPARTITION 10 VALUES(0, 1, 2, 3, 4, 5, 6, 7, 8, 9),
SUBPARTITION 11 VALUES(10, 11, 12, 13, 14, 15, 16, 17, 18, 19),
SUBPARTITION l_default VALUES(DEFAULT) tablespace pg_default
)
(
PARTITION r_lt10 VALUES less than(10)
);

```

创建基于列表-哈希复合分区, 不指定子分区数目的分区表 (Oracle 兼容语法) : 需要将配置更改为 Oracle 兼容模式。

```

create table LH(a int, b int)
partition by list(a)
subpartition by hash(b)
(
partition l_1 values(1),
partition l_2 values(2)
);

```

创建基于列表-哈希复合分区, 指定子分区数目的分区表 (Oracle 兼容语法) : 需要将配置更改为 Oracle 兼容模式。

```

create table LH(a int, b int)
partition by list(a)

```

```

subpartition by hash(b) subpartitions 4
(
  partition l_1 values(1),
  partition l_2 values(2)
);

```

创建基于列表-哈希复合分区，指定子分区模板的分区表（Oracle 兼容语法）：需要将配置更改为 Oracle 兼容模式。

```

create table LH(a int, b int)
partition by list(a)
subpartition by hash(b)
subpartition template
(
  subpartition h_0,
  subpartition h_1
)
(
  partition l_1 values(1),
  partition l_2 values(2)
);

```

创建引用表：

```
CREATE TABLE dist_reference_table(a int, b text) DISTRIBUTED REPLICATED;
```

创建分布式哈希表：

```
CREATE TABLE dist_table(a int, b text) DISTRIBUTED BY(a);
CREATE TABLE dist_hash_table(a int, b text) DISTRIBUTED BY HASH(a);
```

创建分布式追加表：

```
CREATE TABLE dist_append_table(a int, b text) DISTRIBUTED BY APPEND(a);
```

创建分布式表，并指定分片数量为 4：

```
CREATE TABLE dist_shard_table(a int, b text) DISTRIBUTED BY(a) shard_count(4);
```

创建分布式表，并指定与 dist\_table 共存：

```
CREATE TABLE dist_colocate_table(a int, b text) DISTRIBUTED BY (a)
colocate_with('dist_table');
```

创建一个会话级别的全局临时表。

```
CREATE GLOBAL TEMPORARY TABLE gtt_session(id number,ename varchar(15)) ON COMMIT
PRESERVE ROWS;
```

创建一个事务级别的全局临时表。

```
CREATE GLOBAL TEMPORARY TABLE gtt_transaction(id number,ename varchar(15)) ON
  COMMIT DELETE ROWS;
```

使用COMPRESS/NOCOMPRESS/LOGGING/NOLOGGING选项来创建表。

```
CREATE TABLE comp_log1(id int) COMPRESS;
CREATE TABLE comp_log2(id int) NOCOMPRESS;
CREATE TABLE comp_log3(id int) LOGGING;
CREATE TABLE comp_log4(id int) NOLOGGING;
CREATE TABLE comp_log5(id int) COMPRESS LOGGING;
CREATE TABLE comp_log6(id int) COMPRESS NOLOGGING;
CREATE TABLE comp_log7(id int) NOCOMPRESS LOGGING;
CREATE TABLE comp_log8(id int) NOCOMPRESS NOLOGGING;
CREATE TABLE comp_log9(id int) LOGGING COMPRESS;
CREATE TABLE comp_log10(id int) LOGGING NOCOMPRESS;
CREATE TABLE comp_log11(id int) NOLOGGING COMPRESS;
CREATE TABLE comp_log12(id int) NOLOGGING NOCOMPRESS;
```

使用STORAGE选项来创建表。

```
create table comp_log1(id int) storage(INITIAL 65536 NEXT 1048576 MINEXTENTS 1
  MAXEXTENTS 2147483645 PCTINCREASE 0 FREELISTS 1
  FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE
  DEFAULT);
create table comp_log2(id int) storage(464 131 fw f31 fe31);
create table comp_log3(id int) storage(dfa fdsa fdasf fdsa fdsa fads dfsa 1234
  454);
create table comp_log4(id int) storage(abcd);
create table comp_log5(id int) storage(654123);
```

创建一张带嵌套表类型的表。

```
create type nt_type is table of int;
create table nt_table (col nt_type);
```

## Compatibility

CREATE TABLE命令遵从SQL标准，除了以下例外。

### 临时表

尽管CREATE TEMPORARY TABLE的语法很像 SQL 标准的语法，但事实是并不相同。在标准中，临时表只需要被定义一次并且会自动地存在（从空内容开始）于需要它们的每一个会话中。LightDB则要求每一个会话为每一个要用的临时表发出它自己的CREATE TEMPORARY TABLE命令。这允许不同的会话为不同的目的使用相同的临时表名，而标准的方法约束一个给定临时表名的所有实例都必须具有相同的表结构。

标准中对于临时表行为的定义被广泛地忽略了。LightDB在这一点上的行为和多种其他 SQL 数据库是相似的。

SQL 标准也区分全局和局部临时表，其中一个局部临时表为每一个会话中的每一个 SQL 模块具有一个独立的内容集合，但是它的定义仍然是多个会话共享的。因为LightDB不支持 SQL 模块，这种区别与LightDB无关。

为了兼容性目的，LightDB将在临时表声明中接受GLOBAL和LOCAL关键词，但是它们当前没有效果。我们不鼓励使用这些关键词，因为未来版本的LightDB可能采用一种更兼容标准的（对它们含义的）解释。

LightDB 目前支持全局临时表 (global temporary table) 功能，支持会话级别临时表 (session-level temporary tables) 和事务级别临时表 (transaction-level temporary tables)。暂时不支持全局临时表的分区 (partitioning)。全局临时表支持分布式部署。

临时表的ON COMMIT子句也和 SQL 标准相似，但是有一些不同。如果忽略ON COMMIT子句，SQL 指定默认行为是ON COMMIT DELETE ROWS。但是，LightDB中的默认行为是ON COMMIT PRESERVE ROWS。SQL 中不存在ON COMMIT DROP选项。

## 本地表

CREATE LOCAL TABLE 的语法与 SQL 标准相似，其效果与 CREATE TABLE 相同。当 lightdb\_arch\_mode 设置为 dist 时，创建的是一个普通的本地表，而非分布式表。

lightdb\_arch\_mode 有两个取值。取值为 dist 表示当前处于分布式架构模式。取值为 classic 表示当前数据库系统不是分布式架构模式。这个配置正在开发中，不建议用户使用。当处于 dist 模式时，会改变创建表的语法。在 dist 模式下，如果创建表有主键，将会按照主键列创建分布式表，这个功能也正在开发中，不建议用户使用。

## 非延迟唯一性约束

但一个UNIQUE或PRIMARY KEY约束是非可延迟的，只要一个行被插入或修改，LightDB就会立即检查唯一性。SQL 标准指出只有在语句结束时才应该强制唯一性。当一个单一命令更新多个键值时，这两者是不同的。要得到兼容标准的行为，将该约束声明为DEFERRABLE但是不延迟（即INITIALLY IMMEDIATE）。注意这可能要显著地慢于立即唯一性检查。

## 列检查约束

SQL 标准指出CHECK列约束只能引用它们应用到的列，只有CHECK表约束能够引用多列。LightDB并没有强制这个限制，它同样处理列检查约束和表检查约束。

## EXCLUDE 约束

EXCLUDE约束类型是一种LightDB扩展。

## NULL “约束”

NULL “约束”（实际上是一个非约束）是一个LightDB对 SQL 标准的扩展，它也被包括（以及对称的NOT NULL约束）在一些其他的数据库系统中以实现兼容性。因为它是任意列的默认值，它的存在就像噪声一样。

## Constraint Naming

SQL标准规定在包含表或域的模式范围内表和域的约束必须具有唯一的名称。LightDB是比较宽松的：它只需要约束名称在附加到特定表或域的约束之间是唯一的。但是，对于基于索引的约束 (UNIQUE, PRIMARY KEY, and EXCLUDE constraints)，这个特别的自由度并不存在，因为关联的索引被命名为与约束相同的名称，并且索引名称在相同模式的所有关系中必须是唯一的。

当前，LightDB没有记录NOT NULL约束的名称，因此它们不受唯一性限制的影响。这在将来的版本中可能会改变。

## 继承

通过INHERITS子句的多继承是一种LightDB的语言扩展。SQL:1999 以及之后的标准使用一种不同的语法和不同的语义定义了单继承。SQL:1999-风格的继承还没有被LightDB。

## 零列表

LightDB允许创建一个没有列的表（例如CREATE TABLE foo();）。这是一个对于 SQL 标准的扩展，它不允许零列表。零列表本身并不是很有用，但是不允许它们会为ALTER TABLE DROP COLUMN带来奇怪的特殊情况，因此忽略这种规则限制看起来更加整洁。

## 多个标识列

LightDB允许一个表拥有多个标识列。该标准指定一个表最多只能有一个标识列。这主要是为了给模式更改或迁移提供更大的灵活性。请注意，INSERT命令仅支持一个适用于整个语句的覆盖子句，因此不支持具有不同行为的多个标识列。

## Generated Columns

STORED选项不是标准，但也用于其他 SQL 实现。SQL 标准不规定生成列的存储。

## LIKE 子句

虽然 SQL 标准中有一个LIKE子句，但是LightDB接受的很多LIKE子句选项却不在标准中，并且有些标准中的选项也没有被LightDB实现。

## WITH子句

WITH子句是一个LightDB扩展，存储参数不在标准中。

## 表空间

LightDB的表空间概念不是标准的一部分。因此，子句TABLESPACE和USING INDEX TABLESPACE是扩展。

## 类型化的表

类型化的表实现了 SQL 标准的一个子集。根据标准，一个类型化的表具有与底层组合类型相对应的列，以及其他的“自引用列”。LightDB不显式支持自引用列。

## PARTITION BY 子句

PARTITION BY子句是LightDB的一个扩展。

## PARTITION OF 子句

PARTITION OF子句LightDB的一个扩展。

## 参见

ALTER TABLE, DROP TABLE, CREATE TABLE AS, CREATE TABLESPACE, CREATE TYPE

---

# CREATE TABLE AS

CREATE TABLE AS — 从一个查询的结果创建一个新表

## Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ USING method ]
    [ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ COMPRESS | NOCOMPRESS | LOGGING | NOLOGGING | COMPRESS LOGGING | COMPRESS
NOLOGGING | NOCOMPRESS LOGGING | NOCOMPRESS NOLOGGING | LOGGING COMPRESS |
LOGGING NOCOMPRESS | NOLOGGING COMPRESS | NOLOGGING NOCOMPRESS ]
    [ STORAGE(...) ]
    [ TABLESPACE tablespace_name ]
AS query
[ WITH [ NO ] DATA ]
```

## 描述

CREATE TABLE AS 创建一个表，并且用 由一个SELECT命令计算出来的数据填充 该表。该表的列具有和SELECT的输出列 相关的名称和数据类型（不过可以通过给出一个显式的新列名列表来覆 盖这些列名）。

CREATE TABLE AS和创建一个视图有些 相似，但是实际上非常不同：它会创建一个新表并且只计算该查询一次 用来初始填充新表。这个新表将不会跟踪该查询源表的后续变化。相反， 一个视图只要被查询，它的定义SELECT 语句就会被重新计算。

## 参数

GLOBAL或者LOCAL

为兼容性而忽略。不推荐使用这些关键词，详见 CREATE TABLE。

TEMPORARY或者TEMP

如果被指定，该表会被创建一个临时表。详见 CREATE TABLE。

UNLOGGED

如果被指定，该表会被创建一个不做日志的表。详见 CREATE TABLE。

IF NOT EXISTS

如果已经存在一个同名的关系时不要抛出错误。这种情况下会发出一个 提示。详见CREATE TABLE。

table\_name

要创建的表的名称（可以被模式限定）。

column\_name

新表中一列的名称。如果没有提供列名，会从查询的输出列名中得到。

## USING method

这个可选的子句指定了用于存储新表内容的表访问方法；该方法需要是一个类型TABLE的访问方法。详见Chapter 57 如果没有指定这个选项，则选择新表的默认表访问方法。详见default\_table\_access\_method。

## WITH ( storage\_parameter [= value] [, ... ] )

这个子句为新表指定可选的存储参数，详见CREATE TABLE中的 Storage Parameters。为了向后兼容，表的WITH子句也能包含OIDS=FALSE来指定新表的行将不包含OID（对象标识符）。OIDS=TRUE不再支持。

## WITHOUT OIDS

这是向后兼容的语法，用于声明表WITHOUT OIDS，创建表WITH OIDS不再被支持。

## ON COMMIT

临时表在事务块结束时的行为可以用ON COMMIT 控制。三个选项是：

## PRESERVE ROWS

在事务结束时不采取特殊的动作。这是默认行为。

## DELETE ROWS

在每一个事务块结束时临时表中的所有行都将被删除。本质上，在每次提交时会完成一次自动的TRUNCATE。

## DROP

在当前事务块结束时将删掉临时表。

## COMPRESS LOGGING

为了兼容Oracle数据库语法，新增了四个非保留关键字COMPRESS, LOGGING, NOCOMPRESS, NOLOGGING。这四个关键字仅是语法糖，没有实现实际的功能。这四个关键字是可选的，如果必须要使用，请放在TABLESPACE关键字之前。

## STORAGE

为了兼容Oracle数据库语法，CREATE TABLE AS语句支持STORAGE关键字。STORAGE关键字后面括号内的内容是被忽略的。STORAGE关键字是可选的，如果一定要使用，请在COMPRESS LOGGING关键字和TABLESPACE关键字之间使用。

## TABLESPACE tablespace\_name

tablespace\_name 是要在其中创建新表的表空间名称。如果没有指定，将会查询default\_tablespace，临时表会查询 temp\_tablespaces。

## query

一个SELECT、TABLE、或 VALUES命令，或者是一个运行准备好的SELECT、TABLE、或者VALUES查询的EXECUTE命令。

## WITH [ NO ] DATA

这个子句指定查询产生的数据是否应该被复制到新表中。如果不是，只有表结构会被复制。默认是复制数据。

## 注解

这个命令在功能上类似于SELECT INTO，但是它更好，因为不太可能被SELECT INTO语法的其他使用混淆。更进一步，CREATE TABLE AS提供了 SELECT INTO的功能的一个超集。



## 示例

创建一个新表films\_recent，它只由表 films中最近的项组成：

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

要完全地复制一个表，也可以使用TABLE命令的 简短形式：

```
CREATE TABLE films2 AS
  TABLE films;
```

用一个预备语句创建一个新的临时表films\_recent， 它仅由表films中最近的项组成，使用准备好的声明。新表将在提交时被丢弃：

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

使用COMPRESS/NOCOMPRESS/LOGGING/NOLOGGING选项来创建表。

```
CREATE TABLE films_recent compress logging tablespace pg_default AS SELECT 10
  age, 'xiaohong' name, 123456 id;
```

使用STORAGE选项来创建表。

```
CREATE TABLE films_recent compress logging storage(INITIAL 65536 NEXT 1048576
  MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 FREELISTS 1
  FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE
  DEFAULT) tablespace pg_default AS SELECT 10 age, 'xiaohong' name, 123456 id;
```

## 兼容性

CREATE TABLE AS符合 SQL标准。下面的是非标准扩展：

- 标准要求在于查询子句周围有圆括号，在 LightDB中这些圆括号是可选的。
- 在标准中，WITH [ NO ] DATA子句是必要的，而 LightDB 中是可选的。
- LightDB处理临时表的方式和标准不同。 详见CREATE TABLE。
- WITH子句是一种 LightDB扩展， 标准中没有存储参数。
- LightDB的表空间概念是标准的一部分。因此，子句TABLESPACE是一种扩展。

## 另见

CREATE MATERIALIZED VIEW, CREATE TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

---

# CREATE TABLESPACE

CREATE TABLESPACE — 定义一个新的表空间

## Synopsis

```
CREATE TABLESPACE tablespace_name
  [ OWNER { new_owner | CURRENT_USER | SESSION_USER } ]
  LOCATION 'directory'
  [ WITH ( tablespace_option = value [, ... ] ) ]
```

## 描述

CREATE TABLESPACE注册一个新的集簇范围的表空间。表空间的名称必须与数据库集簇中现有的任何表空间不同。

表空间允许超级用户在文件系统上定义另一个位置，可以把包含数据库对象（例如表和索引）的数据文件放在那里。

一个具有适当特权的用户可以把 tablespace\_name传递给 CREATE DATABASE、CREATE TABLE、CREATE INDEX或者ADD CONSTRAINT 来让这些对象的数据文件存储在指定的表空间中。

### Warning

表空间不能独立于定义它的集簇使用，见 Section 20.6

## 参数

tablespace\_name

The name of a tablespace to be created. The name cannot begin with pg\_, as such names are reserved for system tablespaces.

user\_name

将拥有该表空间的用户名。如果省略，默认为执行该命令的用户。只有超级用户能创建表空间，但是它们能把表空间的拥有权赋予给非超级用户。

directory

要被用于表空间的目录。该目录必需存在（CREATE TABLESPACE 将不创建它），应该为空，并且必须由 LightDB系统用户拥有。该目录必须用一个绝对路径指定。

tablespace\_option

要设置或者重置的表空间参数。当前，唯一可用的参数是 seq\_page\_cost、random\_page\_cost、effective\_io\_concurrency以及maintenance\_io\_concurrency。为一个特定表空间设定这些值将覆盖规划器对该表空间中表页读取的常规代价估计，和执行程序的预取行为，由同名的配置参数确定（见 seq\_page\_cost、random\_page\_cost、effective\_io\_concurrency、maintenance\_io\_concurrency）。如果一个表空间位于一个比其他 I/O 子系统更慢或者更快的磁盘上，这些参数就能发挥作用。

## 注解

只有在支持符号链接的系统上才支持表空间。

CREATE TABLESPACE不能在一个事务块内被执行。

## 示例

要在文件系统位置/data/dbs创建表空间dbspace，请首先使用操作系统工具创建目录并设置正确的所有权：

```
mkdir /data/dbs  
chown lightdb:lightdb /data/dbs
```

然后在内部发出表空间创建命令 LightDB:

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

要创建由不同数据库用户拥有的表空间，可用类似这样的命令：

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

## 兼容性

CREATE TABLESPACE是一种 LightDB扩展。

## 另见

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

---

# CREATE TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH CONFIGURATION — 定义一个新的文本搜索配置

## Synopsis

```
CREATE TEXT SEARCH CONFIGURATION name (  
    PARSER = parser_name |  
    COPY = source_config  
)
```

## 描述

CREATE TEXT SEARCH CONFIGURATION 创建一个新的文本搜索配置。一个文本搜索配置指定一个文本搜索解析器（它能将字符串解析成记号），外加一些词典（可被用来决定哪些记号是搜索感兴趣的）。

如果只指定了解析器，那么新文本搜索配置最初没有从记号类型到词典的映射，并且因此将忽略所有词。后续的ALTER TEXT SEARCH CONFIGURATION命令必须被用来创建映射以让该配置变得可用。另一种方式是复制一个现有的文本搜索配置。

如果给定一个模式名称，则文本搜索配置会被创建在指定的模式中。否则它将会被创建在当前模式中。

定义一个文本搜索配置的用户会成为其拥有者。

进一步的信息请参考Chapter 12

## 参数

name

要创建的文本搜索配置的名称。该名称可以被模式限定。

parser\_name

这个配置要使用的文本搜索解析器的名称。

source\_config

要复制的已有文本搜索配置的名称。

## 注解

PARSER和COPY选项是互斥的，因为当一个已有的配置被复制时，它的解析器选择也会被复制。

## 兼容性

在 SQL 标准中没有 CREATE TEXT SEARCH CONFIGURATION语句。

## 另见

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

---

# CREATE TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH DICTIONARY — 定义一个新的文本搜索字典

## Synopsis

```
CREATE TEXT SEARCH DICTIONARY name (  
    TEMPLATE = template  
    [, option = value [, ... ]]  
)
```

## 描述

CREATE TEXT SEARCH DICTIONARY 创建一个 新的文本搜索字典。一个文本搜索字典指定一种识别搜索感兴趣或者不感兴趣 的单词的方法。 一个字典依赖于一个文本搜索模板，后者指定了实际执行该工 作的函数。通常该字典提供一些控制该模板函数细节行为的选项。

如果给出了一个模式名称，那么该文本搜索字典会被创建在指定的模式中。 否则它会被创建在当前模式中。

定义文本搜索字典的用户将成为其拥有者。

进一步的信息可参考Chapter 12

## 参数

name

要创建的文本搜索字典的名称。该名称可以是模式限定的。

template

将定义这个字典基本行为的文本搜索模板的名称。

option

要为此字典设置的模板相关选项的名称。

value

用于模板相关选项的值。如果该值不是一个简单标识符或者数字，它必须 被加引用（你可以按照你所希望的总是对它加上引用）。

选项可以以任意顺序出现。

## 示例

下面的例子命令创建了一个基于 Snowball 的字典，它使用了非标准的 停用词列表。

```
CREATE TEXT SEARCH DICTIONARY my_russian (  
    template = snowball,  
    language = russian,  
    stopwords = myrussian  
);
```

## 兼容性

在 SQL 标准中没有 CREATE TEXT SEARCH DICTIONARY 语句。

## 另见

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

---

# CREATE TEXT SEARCH PARSER

CREATE TEXT SEARCH PARSER — 定义一个新的文本搜索解析器

## Synopsis

```
CREATE TEXT SEARCH PARSER name (  
    START = start_function ,  
    GETTOKEN = gettoken_function ,  
    END = end_function ,  
    LEXTYPES = lextypes_function  
    [, HEADLINE = headline_function ]  
)
```

## 描述

CREATE TEXT SEARCH PARSER 创建一个 新的文本搜索解析器。一个文本搜索解析器定义把文本字符串分解成记号 并且为记号分配类型（分类）的方法。一个解析器本身并不特别有用，但 是必须与一些要用于搜索的文本搜索字典一起被绑定到一个文本搜索配置 中。

如果给出了一个模式名称，那么文本搜索解析器将被创建在指定的模式中。 否则它会被创建在当前模式中。

只有超级用户才能使用 CREATE TEXT SEARCH PARSER（这是因为 错误的文本搜索定义可能会让服务器混淆甚至崩溃）。

更多信息可以参考Chapter 12

## 参数

name

要创建的文本搜索解析器的名称。该名称可以是模式限定的。

start\_function

用于该解析器的开始函数的名称。

gettoken\_function

用于该解析器的取下一个记号的函数名称。

end\_function

用于该解析器的结束函数的名称。

lextypes\_function

用于该解析器的词法分析器函数（一个返回其产生的记号类型集合信息的函数）的名称。

headline\_function

用于该解析器的标题函数（一个对记号集合进行综述的函数）的名称。

如有必要，函数的名称可以被模式限定。参数类型没有给出， 因为函数的每个类型的参数列表无法被预先决定。除了标题函数之外， 所有函数都是必要的。

参数可以以任何顺序出现，而不是必须按照上面所展示的顺序。

## 兼容性

在 SQL 标准中没有 CREATE TEXT SEARCH PARSER 语句。

## 另见

ALTER TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER



---

# CREATE TEXT SEARCH TEMPLATE

CREATE TEXT SEARCH TEMPLATE — 定义一种新的文本搜索模板

## Synopsis

```
CREATE TEXT SEARCH TEMPLATE name (  
    [ INIT = init_function , ]  
    LEXIZE = lexize_function  
)
```

## 描述

CREATE TEXT SEARCH TEMPLATE 创建一个 新的文本搜索模板。文本搜索模板定义实现文本搜索字典的函数。一个模板 本身没什么用处，但是必须被实例化为一个字典来使用。字典通常指定要给 予模板函数的参数。

如果给出了一个模式名，文本搜索模板会被创建在指定模式中。否则它会被 创建在当前模式中。

必须成为超级用户以使用 CREATE TEXT SEARCH TEMPLATE。做出 这种限制是因为错误的模板定义会让服务器混淆甚至崩溃。将模板与字典 分隔开来的原因是模板中封装了定义字典的“不安全”方面。在 定义字典时可以被设置的参数对非特权用户是可以安全设置的，因此创建 字典不需要拥有特权来操作。

进一步的信息可以参考Chapter 12

## 参数

name

要创建的额文本搜索模板的名称。该名称可以被模式限定。

init\_function

用于模板的初始化函数的名称。

lexize\_function

用于模板的分词函数的名称。

如有必要，函数名称可以被模式限定。参数类型没有给出，因为每一类 函数的参数列表是预先定义好的。分词函数是必需的，但是初始化函数 是可选的。

参数可以以任何顺序出现，而不是只能按照上文的顺序。

## 兼容性

在 SQL 标准中没有 CREATE TEXT SEARCH TEMPLATE 语句。

## 另见

ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

---

# CREATE TRANSFORM

CREATE TRANSFORM — 定义一个新的转换

## Synopsis

```
CREATE [ OR REPLACE ] TRANSFORM FOR type_name LANGUAGE lang_name (  
    FROM SQL WITH FUNCTION from_sql_function_name [ (argument_type [, ...]) ],  
    TO SQL WITH FUNCTION to_sql_function_name [ (argument_type [, ...]) ]  
);
```

## 简介

CREATE TRANSFORM定义一种新的转换。CREATE OR REPLACE TRANSFORM将 创建一种新的转换或者替换现有的定义。

一种转换指定了两个函数：

- 一个“from SQL”函数负责将类型从 SQL 环境转换到语言。这个函数将在该语言编写的一个函数的参数上调用。
- 一个“to SQL”函数负责将类型从语言转换到 SQL 环境。这个函数将在该语言编写的一个函数的返回值上调用。

没有必要同时提供这些函数。如果有一种没有被指定，将在必要时使用与语言相关的默认行为（为了完全阻止在一个方向上发生转换，你也可以写一个总是报错的转换函数）。

要创建一种转换，你必须拥有该类型并且具有该类型上的 USAGE特权，拥有该语言上的 USAGE特权，并且拥有 from-SQL 和 to-SQL 函数（如果 指定了）及其上的EXECUTE特权。

## 参数

type\_name

该转换的数据类型的名称。

lang\_name

该转换的语言的名称。

from\_sql\_function\_name[(argument\_type [, ...])]

将该类型从 SQL 环境转换到该语言的函数名。它必须接受一个 internal类型的参数并且返回类型internal。实参将是该转换所适用的类型，并且该函数也应该被写成为它是那种类型（但是不允许声明一个返回internal但没有至少一个 internal类型参数的 SQL 层函数）。实际的返回值将与 语言的实现相关。如果没有指定参数列表，则函数名在该模式中必须唯一。

to\_sql\_function\_name[(argument\_type [, ...])]

将该类型从语言转换到 SQL 环境的函数名。它必须接受一个 internal类型的参数并且返回该转换所适用的类型。实参值 将与语言的实现相关。如果没有指定参数列表，则函数名在该模式中必须唯一。

## 注解

使用DROP TRANSFORM移除转换。

## 示例

要为类型 `hstore` 和语言 `plpythonu` 创建一种转换，先搞定该类型和语言：

```
CREATE TYPE hstore ...;
```

```
CREATE EXTENSION plpythonu;
```

然后创建需要的函数：

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

```
CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

最后创建转换把它们连接起来：

```
CREATE TRANSFORM FOR hstore LANGUAGE plpythonu (
    FROM SQL WITH FUNCTION hstore_to_plpython(internal),
    TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

实际上，这些命令将被包裹在扩展中。

`contrib` 小节包含了一些提供转换的扩展，它们可以作为实际的例子。

## 兼容性

这种形式的 `CREATE TRANSFORM` 是一种 `LightDB` 扩展。在 `SQL` 标准中有一个 `CREATE TRANSFORM` 命令，但是它是用于把数据类型适配到客户端语言。该用法不受 `LightDB` 支持。

## 另见

`CREATE FUNCTION`, `CREATE LANGUAGE`, `CREATE TYPE`, `DROP TRANSFORM`

---

# CREATE TRIGGER

CREATE TRIGGER — 定义一个新触发器

## Synopsis

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event
  [ OR ... ] }
  ON table_name
  [ FROM referenced_table_name ]
  [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY
DEFERRED ] ]
  [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name }
  [ ... ] ]
  [ FOR [ EACH ] { ROW | STATEMENT } ]
  [ WHEN ( condition ) ]
  EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
```

这里的event可以是下列之一：

```
INSERT
UPDATE [ OF column_name [, ... ] ]
DELETE
TRUNCATE
```

## 描述

CREATE TRIGGER创建一个新触发器。 CREATE OR REPLACE TRIGGER 将创建一个新触发器，或者替换一个已有的触发器。 该触发器将被关联到指定的表、视图或者外部表并且在表上发生特定操作时将执行指定的函数function\_name。

该触发器可以被指定为在一行上尝试该操作之前触发（在约束被检查并且INSERT、UPDATE或者DELETE被尝试之前）；也可以在该操作完成之后触发（在约束被检查并且INSERT、UPDATE或者DELETE完成之后）；或者取代该操作（在对一个视图插入、更新或删除的情况中）。如果该触发器在事件之前触发或者取代事件，该触发器可以跳过对当前行的操作或者改变正在被插入的行（只对INSERT以及UPDATE操作）。如果该触发器在事件之后触发，所有更改（包括其他触发器的效果）对该触发器“可见”。

一个被标记为FOR EACH ROW的触发器会对该操作修改的每一行都调用一次。例如，一个影响 10 行的DELETE将导致在目标关系上的任何ON DELETE触发器被独立调用 10 次，也就是为每一个被删除的行调用一次。与此相反，一个被标记为FOR EACH STATEMENT的触发器只会为任何给定的操作执行一次，不管该操作修改多少行（特别地，一个修改零行的操作将仍会导致任何可用的FOR EACH STATEMENT触发器被执行）。

被指定为要触发INSTEAD OF触发器事件的触发器必须被标记为FOR EACH ROW，并且只能被定义在视图上。一个视图上的BEFORE和AFTER触发器必须被标记为FOR EACH STATEMENT。

此外，触发器可以被定义成为TRUNCATE触发，但只能是FOR EACH STATEMENT。

下面的表格总结了哪些触发器类型可以被用在表、视图和外部表上：

何时	事件	行级	语句级
BEFORE	INSERT/ UPDATE/DELETE	表和外部表	表、视图和外部表

何时	事件	行级	语句级
	TRUNCATE	—	表
AFTER	INSERT/ UPDATE/DELETE	表和外部表	表、视图和外部表
	TRUNCATE	—	表
INSTEAD OF	INSERT/ UPDATE/DELETE	视图	—
	TRUNCATE	—	—

还有，一个触发器定义可以指定一个布尔的WHEN条件，它将被测试来看看该触发器是否应该被触发。在行级触发器中，WHEN条件可以检查该行的列的新旧值。语句级触发器也可以有WHEN条件，尽管该特性对于它们不是很有用（因为条件不能引用表中的任何值）。

如果有多个同种触发器被定义为相同事件触发，它们将按照名称的字母表顺序被触发。

当CONSTRAINT选项被指定，这个命令会创建一个约束触发器。 这和一个常规触发器相同，不过触发该触发器的时机可以使用SET CONSTRAINTS. 调整。 约束触发器必须是表上的AFTER ROW触发器。 它们可以在导致触发器事件的语句末尾被引发或者在包含该语句的事务末尾被引发。 在后一种情况中，它们被称作是被延迟。 一个待处理的延迟触发器的引发也可以使用SET CONSTRAINTS立即强制发生。 当约束触发器实现的约束被违背时，约束触发器应该抛出一个异常。

REFERENCING选项启用对传递关系的收集，传递关系是包括被当前SQL语句插入、删除或者修改的行的行集合。这个特性让触发器能看到该语句做的事情的全局视图，而不是一次只看到一行。仅对非约束触发器的AFTER触发器允许这个选项。此外，如果触发器是一个UPDATE触发器，则它不能指定column\_name列表。OLD TABLE仅可以被指定一次，并且只能为在UPDATE或DELETE事件上引发的触发器指定，它创建的传递关系包含有该语句更新或删除的所有行的前映像。类似地，NEW TABLE仅可以被指定一次，并且只能为在UPDATE或INSERT事件上引发的触发器指定，它创建的传递关系包含有该语句更新或插入的所有行的后映像。

SELECT不修改任何行，因此你无法创建SELECT触发器。规则和视图可以为需要SELECT触发器的问题提供可行的解决方案。

关于触发器的更多信息请见Chapter 37

## 参数

name

给新触发器的名称。这必须与同一个表上的任何其他触发器相区别。名称不能是模式限定的 — 该触发器会继承它所在表的模式。对于一个约束触发器，这也是使用SET CONSTRAINTS修改触发器行为时要用到的名字。

BEFORE

AFTER

INSTEAD OF

决定该函数是要在事件之前、之后被调用还是会取代该事件。一个约束触发器也能被指定为AFTER。

event

INSERT、UPDATE、DELETE或者TRUNCATE之一，这指定了将要引发该触发器的事件。多个事件可以用OR指定，要求传递关系的时候除外。

对于UPDATE事件，可以使用下面的语法指定一个列的列表：

UPDATE OF column\_name1 [, column\_name2 ... ]

只有当至少一个被列出的列出现在UPDATE命令的更新目标中时，或者如果列出的列之一是生成的列，而且依赖的列是UPDATE的目标，该触发器才会触发。

INSTEAD OF UPDATE事件不允许列的列表。在请求传递关系时，也不能指定列的列表。

table\_name

要使用该触发器的表、视图或外部表的名称（可能是模式限定的）。

referenced\_table\_name

约束引用的另一个表的名称（可能是模式限定的）。这个选项被用于外键约束并且不推荐用于一般的目的。这只能为约束触发器指定。

DEFERRABLE

NOT DEFERRABLE

INITIALLY IMMEDIATE

INITIALLY DEFERRED

该触发器的默认时机。这些约束选项的细节可参考CREATE TABLE文档。这只能为约束触发器指定。

REFERENCING

这个关键词紧接在一个或者两个关系名的声明之前，这些关系提供对触发语句的传递关系的访问。

OLD TABLE

NEW TABLE

这个子句指示接下来的关系名是用于前映像传递关系还是后映像传递关系。

transition\_relation\_name

在该触发器中这个传递关系要使用的（未限定）名称。

FOR EACH ROW

FOR EACH STATEMENT

这指定该触发器函数是应该为该触发器事件影响的每一行被引发一次，还是只为每个SQL语句被引发一次。如果都没有被指定，FOR EACH STATEMENT会是默认值。约束触发器只能被指定为FOR EACH ROW。

condition

一个决定该触发器函数是否将被实际执行的布尔表达式。如果指定了WHEN，只有condition返回true时才会调用该函数。在FOR EACH ROW触发器中，WHEN条件可以分别写OLD.column\_name或者NEW.column\_name来引用列的新旧行值。当然，INSERT触发器不能引用OLD并且DELETE触发器不能引用NEW。

INSTEAD OF触发器不支持WHEN条件。

当前，WHEN表达式不能包含子查询。

注意对于约束触发器，对于WHEN条件的计算不会被延迟，而是直接在行更新操作被执行之后立刻发生。如果该条件计算得不到真，那么该触发器就不会被放在延迟执行的队列中。

function\_name

一个用户提供的函数，它被声明为不用参数并且返回类型trigger，当触发器引发时会执行该函数。

在CREATE TRIGGER的语法中，关键词FUNCTION和PROCEDURE是等效的，但是任何情况下被引用的函数必须是一个函数而不是过程。这里，关键词PROCEDURE的使用是有历史原因的并且已经被废弃。

arguments

一个可选的逗号分隔的参数列表，它在该触发器被执行时会被提供给该函数。参数是字符串常量。简单的名称和数字常量也可以被写在这里，但是它们将全部被转换成字符串。请检查该触发器函数的实现语言的描述来找出在函数内部如何访问这些参数，这可能与普通函数参数不同。

## 注解

要在一个表上创建或替换一个触发器，用户必须具有该表上的TRIGGER特权。用户还必须具有在触发器函数上的EXECUTE特权。

使用DROP TRIGGER移除一个触发器。

当一个列相关的触发器（使用UPDATE OF column\_name语法定义的触发器）的列被列为UPDATE命令的SET列表目标时，它会被触发。即便该触发器没有被引发，一个列的值也可能改变，因为BEFORE UPDATE触发器对行内容所作的改变不会被考虑。相反，一个诸如UPDATE ... SET x = x ...的命令将引发一个位于列x上的触发器，即便该列的值没有改变。

有一些内置的触发器函数可以用来解决常见问题，而无需编写自己的触发器代码；请参见Section 9.28

在一个BEFORE触发器中，WHEN条件正好在函数被或者将被执行之前被计算，因此使用WHEN与在触发器函数的开始测试同一个条件没有实质上的区别。特别注意该条件看到的NEW行是当前值，虽然可能已被早前的触发器所修改。还有，一个BEFORE触发器的WHEN条件不允许检查NEW行的系统列（例如ctid），因为那些列还没有被设置。

在一个AFTER触发器中，WHEN条件正好在行更新发生之后被计算，并且它决定一个事件是否要被放入队列以便在语句的末尾引发该触发器。因此当一个AFTER触发器的WHEN条件不返回真时，没有必要把一个事件放入队列或者在语句末尾重新取得该行。如果触发器只需要为一些行被引发，就能够显著地加快修改很多行的语句的速度。

在一些情况下，单一的SQL命令可能会引发多种触发器。例如，一个带有ON CONFLICT DO UPDATE子句的INSERT可能同时导致插入和更新操作，因此它将根据需要引发这两种触发器。提供给触发器的传递关系与它们的事件类型有关，因此INSERT触发器将只看到被插入的行，而UPDATE触发器将只看到被更新的行。

由外键强制动作导致的行更新或删除（例如ON UPDATE CASCADE或ON DELETE SET NULL）被当做导致它们的SQL命令的一部分。受影响的表上的相关触发器将被引发，这样就提供了另一种方法让SQL命令引发不直接匹配其类型的触发器。在简单的情况中，请求传递关系的触发器将在一个传递关系中看到由原始SQL命令在其表中做出的所有改变。不过，有些情况中一个请求传递关系的AFTER ROW触发器的存在将导致由单个SQL命令触发的外键强制动作被分成多步，每一步都有其自己的传递关系。在这种情况下，没创建一个传递关系集合都会引发存在的所有语句级触发器，确保那些触发器能够在一个传递关系中看到每个受影响的行一次，并且只看到一次。

只有当视图上的动作被一个行级INSTEAD OF触发器处理时才会引发视图上的语句级触发器。如果动作被一个INSTEAD规则处理，那么该语句发出的任何语句都会代替提及该视图的原始语句执行，这样将被引发的触发器是替换语句中提及的表上的那些触发器。类似地，如果视图是自动可更新的，则该动作将被处理为把该语句自动重写成在视图基表上的一个动作，这样基表的语句级触发器就是要被引发的。

在分区表上创建行级触发器将导致在其所有现有分区中创建相同的触发器；任何后来创建或附加的分区也将包含相同的触发器。如果将分区从其父级中分离，则触发器将被删除。分区表上的触发器可能不是 INSTEAD OF 类型。

修改分区表或者带有继承子表的表会引发挂接到显式提及表的语句级触发器，但不会引发其分区或子表的语句级触发器。相反，行级触发器会在受影响的分区或子表上引发，即便它们在查询中没有被明确提及。如果一个语句级触发器用REFERENCING子句定义有传递关系，则来自所有受影响分区或子表中的行的前后映像都是可见的。在继承子表的情况下，行映像仅包括该触发器所附属的表中存在的列。

## 例子

只要表accounts的一行即将要被更新时会执行函数check\_account\_update:

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  EXECUTE FUNCTION check_account_update();
```

修改该触发器定义，使其仅在UPDATE命令中将balance列指定为目标时执行函数:

```
CREATE TRIGGER check_update
  BEFORE UPDATE OF balance ON accounts
  FOR EACH ROW
  EXECUTE FUNCTION check_account_update();
```

这种形式只有列balance具有真正被改变的值时才执行该函数:

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
  EXECUTE FUNCTION check_account_update();
```

调用一个函数来记录accounts的更新，但是只有在有东西被改变时才调用:

```
CREATE TRIGGER log_update
  AFTER UPDATE ON accounts
  FOR EACH ROW
  WHEN (OLD.* IS DISTINCT FROM NEW.*)
  EXECUTE FUNCTION log_account_update();
```

为每一个要插入到视图底层表中的行执行函数view\_insert\_row:

```
CREATE TRIGGER view_insert
  INSTEAD OF INSERT ON my_view
  FOR EACH ROW
  EXECUTE FUNCTION view_insert_row();
```

为每个语句执行函数check\_transfer\_balances\_to\_zero以确认transfer的行不会有净值增加:

```
CREATE TRIGGER transfer_insert
  AFTER INSERT ON transfer
  REFERENCING NEW TABLE AS inserted
  FOR EACH STATEMENT
  EXECUTE FUNCTION check_transfer_balances_to_zero();
```



为每一行执行函数check\_matching\_pairs以确认（同一个语句）同时对匹配对做了更改：

```
CREATE TRIGGER paired_items_update
  AFTER UPDATE ON paired_items
  REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
  FOR EACH ROW
  EXECUTE FUNCTION check_matching_pairs();
```

Section 37. 包含一个用 C 编写的触发器函数的完整例子。

## 兼容性

LightDB中的CREATE TRIGGER语句实现了SQL标准的一个子集。目前缺少下列功能：

- 虽然AFTER触发器的传递表名是以标准的方式用REFERENCING子句指定，但REFERENCING子句中不能指定FOR EACH ROW触发器中用到的行变量。它们以依赖于编写该触发器函数的语言的方式可用，但是对任意一种语言来说是固定的。一些语言实际上的行为就像有包含OLD ROW AS OLD NEW ROW AS NEW的REFERENCING子句存在一样。
- 标准允许把传递表与和列相关的UPDATE触发器一起使用，那么应该在传递表中可见的行集合取决于该触发器的列列表。当前LightDB没有实现这一点。
- LightDB只允许为被触发动作执行一个用户定义的函数。标准允许执行许多其他的 SQL 命令作为被触发的动作，例如CREATE TABLE。这种限制可以很容易地通过创建一个执行想要的命令的用户定义函数来绕过。

SQL 指定多个触发器应该以被创建时间的顺序触发。LightDB则使用名称顺序，这被认为更加方便。

SQL 指定级联删除上的BEFORE DELETE触发器在级联的DELETE完成之后引发。LightDB的行为则是BEFORE DELETE总是在删除动作之前引发，即使是一个级联删除。这被认为更加一致。如果BEFORE触发器修改行或者在引用动作引起的更新期间阻止更新，这也是非标准行为。这能导致约束违背或者被存储的数据不遵从引用约束。

使用OR为一个单一触发器指定多个动作的能力是 SQL 标准的一个LightDB扩展。

为TRUNCATE引发触发器的能力是 SQL 标准的一个LightDB扩展，在视图上定义语句级触发器的能力也是一样。

CREATE CONSTRAINT TRIGGER是SQL标准的一个LightDB扩展。所以是OR REPLACE 选项。

## 参见

ALTER TRIGGER, DROP TRIGGER, CREATE FUNCTION, SET CONSTRAINTS

---

# CREATE TYPE

CREATE TYPE — 定义一种新的数据类型

## Synopsis

```
CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name

CREATE [ OR REPLACE ] TYPE name AS TABLE OF data_type

CREATE [ OR REPLACE ] TYPE name as_is OBJECT ( [ object_type_element_list ] )

where as_is is:

AS | IS

and object_type_element_list is:
```

```

TableFuncElementList [ object_type_func_list ]

and TableFuncElementList is:

attribute_name data_type [ COLLATE collation ] [, ... ]

and object_type_func_list is:

{ type_function_spec | type_procedure_spec } [, ... ]

and type_function_spec is:

MEMBER FUNCTION func_name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT |
= } default_expr ] [, ...] ] ) RETURN rettype

and type_procedure_spec is:

MEMBER FUNCTION func_name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT |
= } default_expr ] [, ...] ] )

CREATE [ OR REPLACE ] TYPE BODY name as_is subprog_decl_in_type_list END

where subprog_decl_in_type_list is:
{ func_decl_in_type | proc_decl_in_type } [ ... ]

and func_decl_in_type is:
MEMBER FUNCTION func_name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT |
= } default_expr ] [, ...] ] ) RETURN rettype as_is pl_block ;

and proc_decl_in_type is:
MEMBER PROCEDURE func_name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT |
= } default_expr ] [, ...] ] ) as_is pl_block ;

and as_is is:
AS | IS

and pl_block is:
[ DECLARE decl_stmts ] BEGIN statement [ , ... ] END

```

## 描述

CREATE TYPE在当前数据库中注册一种新的数据类型。定义数据类型的用户将成为它的拥有者。

如果给定一个模式名，那么该类型将被创建在指定的模式中。否则它会被创建在当前模式中。类型名称必须与同一个模式中任何现有的类型或者域相区别（因为表具有相关的数据类型，类型名称也必须与同一个模式中任何现有表的名字不同）。

如上面的语法所示，有五种形式的CREATE TYPE。它们分别创建组合类型、枚举类型、范围类型、基础类型或者 shell 类型。下文将依次讨论前四种形式。shell 类型仅仅是一种用于后面要定义的类型占位符，通过发出一个不带除类型名之外其他参数的CREATE TYPE命令可以创建这种类型。在创建范围类型和基础类型时，需要 shell 类型作为一种向前引用。

## 组合类型

第一种形式的CREATE TYPE创建组合类型。组合类型由一个属性名和数据类型的列表指定。如果属性的数据类型是可排序的，也可以指定该属性的排序规则。组合类型本质上和表的行类型相同，但是如果只想定义一种类型，使用CREATE TYPE避免了创建一个实际的表。单独的组合类型也是很有用的，例如可以作为函数的参数或者返回类型。

为了能够创建组合类型，必须拥有在其所有属性类型上的USAGE特权。

## 枚举类型

如Section 8.17.7中所述，第二种形式的CREATE TYPE创建枚举类型。枚举类型需要一个带引号的标签构成的列表，每一个标签长度必须不超过NAMEDATALEN字节（在标准的LightDB编译中是64字节）。可以创建具有零个标签的枚举类型，但是在使用ALTER TYPE添加至少一个标签之前，不能使用这种类型来保存值。

## 范围类型

如Section 8.17.8中所述，第三种形式的CREATE TYPE创建范围类型。

范围类型的subtype可以是任何带有一个相关的B树操作符类（用来决定该范围类型值的顺序）的类型。通常，子类型的默认B树操作符类被用来决定顺序。要使用一种非默认操作符类，可以用subtype\_opclass指定它的名字。如果子类型是可排序的并且希望在该范围的顺序中使用一种非默认的排序规则，可以用collation选项来指定。

可选的canonical函数必须接受一个所定义的范围类型的参数，并且返回同样类型的一个值。在适用时，它被用来把范围值转换成一种规范的形式。更多信息请见Section 8.17.8。创建一个canonical函数有点棘手，因为必须在声明范围类型之前定义它。要这样做，必须首先创建一种shell类型，它是一种没有属性只有名称和拥有者的占位符类型。这可以通过发出不带额外参数的命令CREATE TYPE name来完成。然后可以使用该shell类型作为参数和结果来声明该函数，并且最终用同样的名称来声明范围类型。这会使用一种合法的范围类型替换shell类型项。

可选的subtype\_diff函数必须接受两个subtype类型的值作为参数，并且返回一个double precision值表示两个给定值之间的差别。虽然这是可选的，但是提供这个函数会让该范围类型列上GiST索引效率更高。详见Section 8.17.8。

## 基础类型

第四种形式的CREATE TYPE创建一种新的基本类型（标量类型）。为了创建一种新的基本类型，你必须是一个超级用户（做这种限制的原因是一种错误的类型定义可能让服务器混淆甚至崩溃）。

参数可以以任意顺序出现（而不仅是按照上面所示的顺序），并且大部分是可选的。在定义类型前，必须注册两个或者更多函数（使用CREATE FUNCTION）。支持函数input\_function以及output\_function是必需的，而函数receive\_function、send\_function、type\_modifier\_input\_function、type\_modifier\_output\_function和analyze\_function是可选的。通常来说这些函数必须是用C或者另外一种低层语言编写的。

input\_function将类型的外部文本表达转换为该类型定义的操作符和函数所使用的内部表达。output\_function执行反向的转换。输入函数可以被声明为有一个cstring类型的参数，或者有三个类型分别为cstring、oid、integer的参数。第一个参数是以C字符串存在的输入文本，第二个参数是该类型自身的OID（对于数组类型则是其元素类型的OID），第三个参数是目标列的typmod（如果知道，不知道则将传递-1）。输入函数必须返回一个该数据类型本身的值。通常，一个输入函数应该被声明为STRICT。如果不是这样，在读到一个NULL输入值时，调用它时第一个参数会是NULL。在这种情况下，该函数必须仍然返回

NULL，除非它发生了错误（这种情况主要是想支持域输入函数，它们可能需要拒绝 NULL 输入）。输出函数必须被声明为有一个新数据类型的参数。输出函数必须返回类型 `cstring`。对于 NULL 值不会调用输出函数。

T可选的`receive_function` 会把类型的外部二进制表达转换成内部表达。如果没有提供这个函数，该类型不能参与到二进制输入中。二进制表达转换成内部形式代价更低，然而却更容易移植（例如，标准的整数数据类型使用网络字节序作为外部二进制表达，而内部表达是机器本地的字节序）。接收函数应该执行足够的检查以确保该值是有效的。接收函数可以被声明为有一个 `internal` 类型的参数，或者有三个类型分别为 `internal`、`oid`、`integer` 的参数。第一个参数是一个指向 `StringInfo` 缓冲区的指针，其中保存着接收到的字节串。其余可选的参数和文本输入函数的相同。接收函数必须返回一个该数据类型本身的值。通常，一个接收函数应该被声明为 `STRICT`。如果不是这样，在读到一个 NULL 输入值时，调用它时第一个参数会是 NULL。在这种情况下，该函数必须仍然返回 NULL，除非它发生了错误（这种情况主要是想支持域接收函数，它们可能需要拒绝 NULL 输入）。类似地，可选的 `send_function` 将内部表达转换成外部二进制表达。如果没有提供这个函数，该类型将不能参与到二进制输出中。发送函数必须被声明为有一个新数据类型的参数。发送函数必须返回类型 `bytea`。对于 NULL 值不会调用发送函数。

到这里你应该在疑惑输入和输出函数是如何能被声明为具有新类型的结果或参数的？因为必须在创建新类型之前创建这两个函数。这个问题的答案是，新类型应该首先被定义为一种 `shell type`，它是一种占位符类型，除了名称和拥有者之外它没有其他属性。这可以通过不带额外参数的命令 `CREATE TYPE name` 做到。然后用 C 写的 I/O 函数可以被定义为引用这种 `shell type`。最后，用带有完整定义的 `CREATE TYPE` 把该 `shell type` 替换为一个完全的、合法的类型定义，之后新类型就可以正常使用了。

如果该类型支持修饰符（附加在类型声明上的可选约束，例如 `char(5)` 或者 `numeric(30,2)`），则需要可选的 `type_modifier_input_function` 以及 `type_modifier_output_function`。LightDB 允许用户定义的类型有一个或者多个简单常量或者标识符作为修饰符。不过，为了存储在系统目录中，该信息必须能被打包到一个非负整数值中。所声明的修饰符会被以 `cstring` 数组的形式传递给 `type_modifier_input_function`。它必须检查该值的合法性（如果值错误就抛出一个错误），如果值正确，要返回一个非负 `integer` 值，它将被存储在“`typmod`”列中。如果类型没有 `type_modifier_input_function` 则类型修饰符将被拒绝。`type_modifier_output_function` 把内部的整数 `typmod` 值转换回正确的形式用于用户显示。它必须返回一个 `cstring` 值，该值就是追加到类型名称后的字符串。例如 `numeric` 的函数可能会返回 `(30,2)`。如果默认的数据显示格式就是只把存储的 `typmod` 整数值放在圆括号内，则允许省略 `type_modifier_output_function`。

可选的 `analyze_function` 为该数据类型的列执行与类型相关的统计信息收集。默认情况下，如果该类型有一个默认的 B-树操作符类，`ANALYZE` 将尝试用类型的“`equals`”和“`less-than`”操作符来收集统计信息。这种行为对于非标量类型并不合适，因此可以通过指定一个自定义分析函数来覆盖这种行为。分析函数必须被声明为有一个类型为 `internal` 的参数，并且返回一个 `boolean` 结果。分析函数的详细 API 请见 `src/include/commands/vacuum.h`。

虽然只有 I/O 函数和其他为该类型创建的函数才知道新类型的内部表达的细节，但是内部表达的一些属性必须被向 LightDB 声明。其中最重要的是 `internallength`。基本数据类型可以是定长的（这种情况下 `internallength` 是一个正整数）或者是变长的（把 `internallength` 设置为 `VARIABLE`，在内部通过把 `typlen` 设置为 `-1` 表示）。所有变长类型的内部表达都必须以一个 4 字节整数开始，它给出了这个值的总长度（注意如 Section 64.4 中所述，长度域常常是被编码过的，直接接受它是不明智的）。

可选的标志 `PASSEDBYVALUE` 表示这种数据类型的值需要被传值而不是传引用。传值的类型必须是定长的，并且它们的内部表达不能超过 `Datum` 类型（某些机器上是 4 字节，其他机器上是 8 字节）的尺寸。

`alignment` 参数指定数据类型的存储对齐要求。允许的值等同于以 1、2、4 或 8 字节边界对齐。注意变长类型的 `alignment` 参数必须至少为 4，因为它们需要包含一个 `int4` 作为它们的第一个组成部分。

`storage`参数允许为变长数据类型选择存储策略（对定长类型只允许 `plain`）。`plain`指定该类型的数据将总是被存储在线内并且不会被压缩。`extended`指定系统将首先尝试压缩一个长的数据值，并且将在数据仍然太长的情况下把值移出主表行。`external`允许值被移出主表，但是系统将不会尝试对它进行压缩。`main`允许压缩，但是不鼓励把值移出主表（如果没有其他办法让行的大小变得合适，具有这种存储策略的数据项仍将被移出主表，但比起 `extended`以及 `external`项来，这种存储策略的数据项会被优先考虑保留在主表中）。

如Section 64.和Section 36.13.1所述，除 `plain`之外所有的 `storage`值都暗示该数据类型的函数能处理被 `TOAST` 过的值。指定的值仅仅是决定一种可 `TOAST` 数据类型的列的默认 `TOAST` 存储策略，用户可以使用 `ALTER TABLE SET STORAGE`为列选取其他策略。

`like_type`参数提供了另一种方法来指定一种数据类型的基本表达属性：从某种现有的类型中拷贝。`internallength`、`passedbyvalue`、`alignment`和 `storage`的值会从指定的类型中复制而来（也可以通过在 `LIKE`子句中指定这些属性的值来覆盖复制过来的值，不过通常并不这么做）。当新类型的低层实现是以一种现有的类型为“载体”时，用这种方式指定表达特别有用。

`category`和 `preferred`参数可以被用来帮助控制在混淆的情况下应用哪一种隐式造型。每一种数据类型都属于一个用单个 ASCII 字符命名的分类，并且每一种类型可以是其所属分类中的“首选”。当有助于解决重载函数或操作符时，解析器将优先造型到首选类型（但是只能从同类的其他类型造型）。更多细节请见 Chapter 10 对于没有隐式造型到任何其他类型或者从任意其他类型造型的类型，让这些设置保持默认即可。不过，对于一组具有隐式造型的相关类型，把它们都标记为属于同一个类别并且选择一种或两种“最常用”的类型作为该类别的首选通常是很有用的。在把一种用户定义的类型增加到一个现有的内建类别（例如数字或者字符串类型）中时，`category`参数特别有用。不过，也可以创建新的全部是用户定义类型的类别。对这样的类别，可选择除大写字母之外的任何 ASCII 字符。

如果用户希望该数据类型的列被默认为某种非空值，可以指定一个默认值。默认值可以用 `DEFAULT`关键词指定（这样一个默认值可以被附加到一个特定列的显式 `DEFAULT`子句覆盖）。

若要指示一个类型是一个数组，请使用 `ELEMENT` 关键字指定数组元素的类型。例如，要定义一个 4 字节整数 (`int4`) 数组，指定 `ELEMENT = int4`。有关数组类型的更多详细信息，请参见下面的内容。

要指定在这种类型数组的外部表达中分隔值的定界符，可以把 `delimiter`设置为一个特定字符。默认的定界符是逗号 (,)。注意定界符是与数组元素类型相关的，而不是数组类型本身相关。

如果可选的布尔参数 `collatable`为真，这种类型的列定义和表达式可能通过使用 `COLLATE`子句携带有排序规则信息。在该类型上操作的函数的实现负责真正利用这些信息，仅把类型标记为可排序的并不会让它们自动地去使用这类信息。

## 数组类型

只要一种用户定义的类型被创建，LightDB会自动地创建一种相关的数组类型，其名称由元素类型的名称前面加上一个下划线组成，并且如果长度超过 `NAMEDATALEN`字节会自动地被截断。（如果这样生成的名称与一种现有类型的名称冲突，该过程将会重复直到找到一个不冲突的名字）。这种隐式创建的数组类型是变长的并且使用内建的输入和输出函数（`array_in`以及 `array_out`）。此外，这种类型是系统用于构建诸如用户定义类型之上的 `ARRAY[]`。该数组类型会追随其元素类型的拥有者或所在模式的任何更改，并且在元素类型被删除时也被删除。

如果系统会自动地创建正确的数组类型，你可能会很合理地问为什么会有一个 `ELEMENT`选项。使用 `ELEMENT`主要有用的情况是：当你在创建一种定长类型，它正好在内部是一个多个相同东西的数组，并且除了计划给该类型提供的整体操作之外，你想要允许用下标来直接访问这些东西。例如，类型 `point`被表示为两个浮点数，可以使用 `point[0]`以及 `point[1]`来访问它们。注意，这种功能只适用于内部形式正好是一个相同定长域序列的定长类型。由于历史原因（即很明显是错的，但现在改已经太晚了），定长数组类型的下标是从零开始的，而不是像变长数组那样。

## 嵌套表类型

类似于数组，但没有长度限制，主要用于存储过程中。

## 对象类型

类似于组合类型，但是可以携带函数。

## 参数

name

要创建的类型的名称（可以被模式限定）。

attribute\_name

组合类型的一个属性（列）的名称。

data\_type

要成为组合类型的一个列的现有数据类型的名称。

collation

要关联到组合类型的一列或者范围类型的现有排序规则的名称。

label

一个字符串，它表达与枚举类型的一个值相关的文本标签。

subtype

范围类型的元素类型的名称，范围类型表示的范围属于该类型。

subtype\_operator\_class

用于 subtype 的 B 树操作符类的名称。

canonical\_function

范围类型的规范化函数的名称。

subtype\_diff\_function

用于 subtype 的差函数的名称。

input\_function

将数据从类型的外部文本形式转换为内部形式的函数名。

output\_function

将数据从类型的内部形式转换为外部文本形式的函数名。

receive\_function

将数据从类型的外部二进制形式转换成内部形式的函数名。

send\_function

将数据从类型的内部形式转换为外部二进制形式的函数名。

`type_modifier_input_function`

将类型的修饰符数组转换为内部形式的函数名。

`type_modifier_output_function`

将类型的修饰符的内部形式转换为外部文本形式的函数名。

`analyze_function`

为该数据类型执行统计分析的函数名。

`internallength`

一个数字常量，它指定新类型的内部表达的字节长度。默认的假设是它是变长的。

`alignment`

该数据类型的存储对齐需求。如果被指定，它必须是 `char`、`int2`、`int4`或者`double`。默认是 `int4`。

`storage`

该数据类型的存储策略。如果被指定，必须是 `plain`、`external`、`extended`或者`main`。默认是`plain`。

`like_type`

与新类型具有相同表达的现有数据类型的名称。会从这个类型中复制 `internallength`、`passedbyvalue`、`alignment`以及 `storage`的值（除非在这个`CREATE TYPE`命令的其他地方用显式说明覆盖）。

`category`

这种类型的分类码（一个 ASCII 字符）。默认是“用户定义类型”的‘U’。其他的标准分类码可见 Table 48.71 为了创建自定义分类，你也可以选择其他 ASCII 字符。

`preferred`

如果这种类型是其类型分类中的优先类型则为真，否则为假。默认 为假。在一个现有类型分类中创建一种新的优先类型要非常小心，因为这可能会导致行为上令人惊奇的改变。

`default`

数据类型的默认值。如果被省略，默认值是空。

`element`

被创建的类型是一个数组，这指定了数组元素的类型。

`delimiter`

在由这种类型组成的数组中值之间的定界符。

`collatable`

如果这个类型的操作可以使用排序规则信息，则为真。默认为假。

`func_name`

要创建的函数的名称。



**argmode**

一个参数的模式：IN、OUT、INOUT或者VARIADIC。如果省略，默认为IN。只有OUT参数能跟在一个VARIADIC参数后面。还有，OUT和INOUT参数不能和RETURNS TABLE符号一起使用。

**argname**

一个参数的名称。一些语言（包括SQL和PL/pgSQL）让你在函数体中使用该名称。对于其他语言，一个输入参数的名字只是额外的文字（就该函数本身所关心的来说）。但是你可以在调用一个函数时使用输入参数名来提高可读性（见Section 4.3）。在任何情况下，输出参数的名称是有意义的，因为它定义了结果行类型中的列名（如果忽略一个输出参数的名称，系统将选择一个默认的列名）。

**argtype**

该函数参数（如果有）的数据类型（可以是模式限定的）。参数类型可以是基本类型、组合类型或者域类型，或者可以引用一个表列的类型。

根据实现语言，也可以允许指定cstring之类的“伪类型”。伪类型表示实际参数类型没有被完整指定或者不属于普通SQL数据类型集合。

可以写table\_name.column\_name%TYPE来引用一列的类型。使用这种特性有时可以帮助创建一个不受表定义更改影响的函数。

**default\_expr**

如果参数没有被指定值时要用作默认值的表达式。该表达式必须能被强制为该参数的参数类型。只有输入（包括INOUT）参数可以具有默认值。所有跟随在一个具有默认值的参数之后的输入参数也必须有默认值。

**rettype**

返回数据类型（可能被模式限定）。返回类型可以是一种基本类型、组合类型或者域类型，也可以引用一个表列的类型。根据实现语言，也可以允许指定cstring之类的“伪类型”。如果该函数不会返回一个值，可以指定返回类型为void。

当有OUT或者INOUT参数时，可以省略RETURNS子句。如果存在，该子句必须和输出参数所表示的结果类型一致：如果有多个输出参数，则为RECORD，否则与单个输出参数的类型相同。

SETOF修饰符表示该函数将返回一个项的集合而不是一个单一项。

可以写table\_name.column\_name%TYPE来引用一列的类型。

**decl\_stmts**

PL/oraSQL 声明语句块。

**statement**

PL/oraSQL 执行语句块。

## 注解

由于一旦数据类型被创建，对该数据类型的使用就没有限制，创建一种基本类型 或者范围类型就等同于在类型定义中提到的函数上授予公共执行权限。对于在类型定义中有用的函数来说这通常不是问题。但是如果设计一种类型时要求在转换 到外部形式或者从外部形式转换时使用“秘密”信息，你就应该三思而后行。

在LightDB版本 8.3 之前，自动生成的 数组类型的名称总是正好为元素类型的名称外加一个前置的下划线字符（\_）。因此类型名称的长度限制比其他名称还要少一个字符。虽然现

在这仍然是通常情况，但如果名称达到最大长度或者与其他下划线开头的用户类型名称冲突，数组类型的名称也可以不同于这种规则。因此依靠这种习惯编写代码现在已经不适用了。现在，可以使用 `pg_type.typarray` 来定位与给定类型相关的数组类型。

建议避免使用以下划线开始的类型名和表名。虽然服务器会改变生成的数组类型名称以避免与用户给定的名称冲突，仍然有混淆的风险，特别是对旧的客户端软件来说，它们可能会假定以下划线开始的类型名总是表示数组。

在LightDB 版本 8.2 之前，`shell-type` 的创建语法 `CREATE TYPE name` 不存在。创建一种新基本类型的方法是先创建它的输入函数。在这种方法中，LightDB 将首先把新数据类型的名称看做是输入函数的返回类型。在这种情况下 `shell type` 会被隐式地创建，并且能在剩余的 I/O 函数的定义中引用。这种方法现在仍然有效，但是已经被弃用并且可能会在未来的某个发行中被禁止。还有，为了避免由于函数定义中的打字错误导致 `shell type` 弄乱系统目录，当输入函数用 C 编写时，将只能用这种方法创建一种 `shell type`。

## 示例

这个例子创建了一种组合类型并且将其用在了一个函数定义中：

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

这个例子创建了一个枚举类型并且将其用在一个表定义中：

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

这个例子创建了一个范围类型：

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

这个例子创建了基本数据类型`box`然后将它用在一个表定义中：

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
```

```
);
```

如果box的内部结构是四个 float4元素的一个数组，我们可能会使用：

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

这将允许用下标来访问一个 box 值的组件编号。否则该类型的行为和 前面的一样。

这个例子创建了一个大对象类型并且将它用在了一个表定义中：

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

这个例子创建了一个嵌套表类型，并在匿名块中使用它：

```
CREATE TYPE names2 IS TABLE OF VARCHAR(15);

DECLARE
    nt names2 ;
BEGIN
    nt(1) = 'D Caruso';
    nt(2) = 'J Hamil';
    nt(3) = 'R Singh';
    DBMS_OUTPUT.PUT_LINE(nt(2));
END;
/
```

创建对象类型：

```
CREATE TYPE tp AS OBJECT (
    x INT,
    y INT,
    member FUNCTION d(prompt VARCHAR) RETURN VARCHAR,
    member PROCEDURE p(prompt VARCHAR)
);
```

创建对象类型体：

```
CREATE TYPE BODY tp AS
MEMBER FUNCTION d(prompt VARCHAR) RETURN VARCHAR IS
DECLARE
    v VARCHAR;
BEGIN
    v := '(' || self.x || ', ' || self.y || ')';
```

```
    raise notice '%', v;
    return v;
END;

MEMBER PROCEDURE p(prompt VARCHAR) IS
BEGIN
    raise notice '%', '(' || self.x || ', ' || self.y || ')';
END;
END;
/
```

更多例子（包括配套的输入和输出函数）请见Section 36.13

## 兼容性

创建组合类型的第一种形式的CREATE TYPE命令符合SQL标准。其他的形式都是 LightDB扩展。SQL 标准中的CREATE TYPE语句也定义了其他 LightDB中没有实现的形式。

创建一种具有零个属性的组合类型的能力是一种 LightDB对标准的背离（类似于 CREATE TABLE中相同的情况）。

CREATE OR REPLACE TYPE name AS TABLE OF data\_type即可以创建一个新的Nested Table，也可以替换一个旧的Nested Table。

CREATE OR REPLACE TYPE name AS OBJECT即可以创建一个新的OBJECT，也可以替换一个旧的OBJECT。

CREATE OR REPLACE TYPE BODY name AS subprog\_decl\_in\_type\_list END即可以创建一个新的BODY，也可以替换一个旧的BODY。

## 另见

ALTER TYPE, CREATE DOMAIN, CREATE FUNCTION, DROP TYPE

---

# CREATE USER

CREATE USER — 定义一个新的数据库角色

## Synopsis

```
CREATE USER name [ [ WITH ] option [ ... ] | IDENTIFIED
BY password option_for_oracle ]
```

这里 option 可以是:

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
| VALID UNTIL 'timestamp'
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN role_name [, ...]
| USER role_name [, ...]
| SYSID uid
```

where option\_for\_oracle can be:

```
DEFAULT TABLESPACE tablespace
| TEMPORARY TABLESPACE tablespace
```

## 参数

IDENTIFIED BY

用于与 Oracle 兼容的 CREATE USER 语句。仅提供语法支持，不提供功能支持的 option\_for\_oracle。为了与 Oracle 授权功能兼容，在 Oracle 模式下不允许使用名称为“connect”和“resource”的用户进行 CREATE USER。必须在 Oracle 数据库中使用。

## 描述

CREATE USER现在是 CREATE ROLE的一个别名。唯一的区别是 CREATE USER中LOGIN 被作为默认值，而NOLOGIN是 CREATE ROLE的默认值。

## 兼容性

CREATE USER语句是一种 LightDB扩展。SQL 标准 把用户的定义留给实现来解释。

## 另见

CREATE ROLE

---

# CREATE USER MAPPING

CREATE USER MAPPING — 定义一个用户到一个外部服务器的新映射

## Synopsis

```
CREATE USER MAPPING [ IF NOT EXISTS ] FOR { user_name | USER | CURRENT_USER |
PUBLIC }
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

## 描述

CREATE USER MAPPING定义一个用户 到一个外部服务器的新映射。一个用户映射通常会包含连接信息，外部数据包装器 会使用连接信息和外部服务器中包含的信息一起来访问一个外部数据源。

一个外部服务器的拥有者可以为任何服务器任何用户创建用户映射。还有， 如果一个用户被授予了服务器上的USAGE特权，该用户可以 为他们自己的用户名创建用户映射。

## 参数

IF NOT EXISTS

如果给定用户到给定外部服务器的映射已经存在，则不要抛出错误。 在这种情况下发出通知。请注意，不能保证现有的用户映射与要创建的映射完全相同。

user\_name

要映射到外部服务器的一个现有用户的名称。 CURRENT\_USER和USER匹配当前用户的名称。 当PUBLIC被指定时，一个所谓的公共映射会被创建，当没有 特定用户的映射可用时将会使用它。

server\_name

将为其创建用户映射的现有服务器的名称。

OPTIONS ( option 'value' [ , ... ] )

这个子句指定用户映射的选项。这些选项通常定义该映射实际的用户名和 口令。选项名必须唯一。允许的选项名和值与该服务器的外部数据包装器 有关。

## 示例

为用户bob、服务器foo创建一个用户映射：

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

## 兼容性

CREATE USER MAPPING符合 ISO/IEC 9075-9 (SQL/MED)。

## 另见

ALTER USER MAPPING, DROP USER MAPPING, CREATE FOREIGN DATA WRAPPER, CREATE SERVER

---

# CREATE VIEW

CREATE VIEW — 定义一个新视图

## Synopsis

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] [ EDITIONABLE ]  
VIEW name [ ( column_name [, ...] ) ]  
    [ WITH ( view_option_name [= view_option_value] [, ...] ) ]  
AS query  
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

## 描述

CREATE VIEW定义一个查询的视图。该视图不会被物理上物质化。相反，在每一次有查询引用该视图时，视图的查询都会被运行。

CREATE OR REPLACE VIEW与之相似，但是如果已经存在一个同名视图，该视图会被替换。新查询必须产生和现有视图查询相同的列（也就是相同的列序、相同的列名、相同的数据类型），但是它可以在列表的末尾加上额外的列。产生输出列的计算可以完全不同。

如果给定了一个模式名（例如CREATE VIEW myschema.myview ...），那么该视图会被创建在指定的模式中。否则，它会被创建在当前模式中。临时视图存在于一个特殊模式中，因此创建临时视图时不能给定一个模式名。视图的名称不能与同一模式中任何其他视图、表、序列、索引或外部表同名。

## 参数

TEMPORARY或者TEMP

如果被指定，视图被创建为一个临时视图。在当前会话结束时会自动删掉临时视图。当临时视图存在时，具有相同名称的已有永久视图对当前会话不可见，除非用模式限定的名称引用它们。

如果视图引用的任何表是临时的，视图将被创建为临时视图（不管有没有指定TEMPORARY）。

RECURSIVE

创建一个递归视图。语法

```
CREATE RECURSIVE VIEW [ schema . ] view_name (column_names) AS SELECT ...;
```

等效于

```
CREATE VIEW [ schema . ] view_name AS WITH RECURSIVE view_name (column_names)  
AS (SELECT ...) SELECT column_names FROM view_name;
```

对于一个递归视图必须指定一个视图列名列表。

EDITIONABLE

创建一个编辑视图。语法

```
CREATE EDITIONABLE VIEW [ schema . ] view_name (column_names) AS SELECT ...;
```

该用法仅支持在oracle兼容模式使用。

name

要创建的视图的名字（可以是模式限定的）。

column\_name

要用于视图列的名称列表，可选。如果没有给出，列名会根据查询 推导。

```
WITH ( view_option_name [= view_option_value] [, ... ] )
```

这个子句为视图指定一些可选的参数，支持下列参数：

check\_option (enum)

这个参数可以是local或者cascaded，并且它 等效于指定 WITH [ CASCADED | LOCAL ] CHECK OPTION（见下文）。 可以使用ALTER VIEW在一个现有视图上修改这个选项。

security\_barrier (boolean)

如果希望视图提供行级安全性，应该使用这个参数。详见 Section 39.5

query

提供视图的行和列的一个SELECT 或者 VALUES命令。

```
WITH [ CASCADED | LOCAL ] CHECK OPTION
```

这个选项控制自动可更新视图的行为。这个选项被指定时，将检查该视图上的INSERT和UPDATE命令以确保新行满足 视图的定义条件（也就是，将检查新行来确保通过视图能看到它们）。如果新行 不满足条件，更新将被拒绝。如果没有指定CHECK OPTION， 会允许该视图上的INSERT和UPDATE命令 创建通过该视图不可见的行。支持下列检查选项：

LOCAL

只根据直接定义在该视图本身的条件检查新行。任何定义在底层基视图上的 条件都不会被检查（除非它们也指定了CHECK OPTION）。

CASCADED

会根据该视图和所有底层基视图上的条件检查新行。如果 CHECK OPTION被指定，并且没有指定 LOCAL和CASCADED，则会假定为 CASCADED。

CHECK OPTION不应该和RECURSIVE视图一起使用。

注意，只有在自动可更新的、没有INSTEAD OF触发器或者 INSTEAD规则的视图上才支持CHECK OPTION。 如果一个自动可更新的视图被定义在一个具有INSTEAD OF 触发器的基视图之上，那么LOCAL CHECK OPTION可以被 用来检查该自动可更新的视图之上的条件，但具有INSTEAD OF 触发器的基视图上的条件不会被检查（一个级联检查选项将不会级联到一个 触发器可更新的视图，并且任何直接定义在一个触发器可更新视图上的检查 选项将被忽略）。如果该视图或者任何基础关系具有导致 INSERT或UPDATE命令被重写的 INSTEAD规则，那么在被重写的查询中将忽略所有检查选项， 包括任何来自于定义在带有INSTEAD规则的关系之上的自动 可更新视图的检查。

## 注解

使用DROP VIEW语句删除视图。



要小心视图列的名称和类型将会按照你想要的方式指定。例如：

```
CREATE VIEW vista AS SELECT 'Hello World';
```

是不好的形式，因为列名默认为?column?，而且列的数据类型默认为text，这可能不是用户想要的。视图结果中一个字符串更好的风格类似于这样：

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

对视图中引用的表的访问由视图拥有者的权限决定。在某些情况下，这可以被用来提供安全但是受限的底层表访问。不过，并非所有视图都对篡改是安全的，详见Section 39.5 在视图中调用的函数会被同样对待，就好像是直接在使用该视图的查询中调用它们一样。因此，一个视图的用户必须具有调用视图所使用的全部函数的权限。

当CREATE OR REPLACE VIEW被用于一个现有视图上时，只有该视图的定义 SELECT 规则被改变。其他包括拥有关系、权限和非 SELECT 规则在内的视图属性不会被更改。要替换视图，你必须拥有它（包括作为拥有角色的一个成员）。

## 可更新视图

简单视图是自动可更新的：系统将允许在这类视图上以在常规表上相同的方式使用INSERT、UPDATE以及DELETE语句。如果一个视图满足以下条件，它就是自动可更新的：

- 在该视图的FROM列表中刚好只有一项，并且它必须是一个表或者另一个可更新视图。
- 视图定义的顶层不能包含WITH、DISTINCT、GROUP BY、HAVING、LIMIT或者OFFSET子句。
- 视图定义的顶层不能包含集合操作（UNION、INTERSECT或者EXCEPT）。
- 视图的选择列表不能包含任何聚集、窗口函数或者集合返回函数。

一个自动可更新的视图可以混合可更新列以及不可更新列。如果一个列是对底层基本关系中一个可更新列的简单引用，则它是可更新的。否则该列是只读的，并且在一个INSERT或者UPDATE语句尝试对它赋值时会报出一个错误。

如果视图是自动可更新的，系统将把视图上的任何INSERT、UPDATE或者DELETE语句转换成在底层基本关系上的对应语句。带有ON CONFLICT UPDATE子句的INSERT语句已经被完全支持。

如果一个自动可更新视图包含一个WHERE条件，该条件会限制基本关系的哪些行可以被该视图上的UPDATE以及DELETE语句修改。不过，一个允许被UPDATE修改的行可能让该行不再满足WHERE条件，并且因此也不再能从视图中可见。类似地，一个INSERT命令可能插入不满足WHERE条件的基本关系行，并且因此从视图中也看不到这些行（ON CONFLICT UPDATE可能会类似地影响无法通过该视图见到的现有行）。CHECK OPTION可以被用来阻止INSERT和UPDATE命令创建这类从视图中无法看到的行。

如果一个自动可更新视图被标记了security\_barrier属性，那么所有该属性的WHERE条件（以及任何使用标记为LEAKPROOF的操作符的条件）将在该视图使用者的任何条件之前计算。详见Section 39.5 注意正因为这样，不会被最终返回的行（因为它们不会通过用户的WHERE条件）可能仍会结束被锁定的状态。可以用EXPLAIN来查看哪些条件被应用在关系层面（并且因此不锁定行）以及哪些不会被应用在关系层面。

一个更加复杂的不满足所有这些条件的视图默认是只读的：系统将不允许在该视图上的插入、更新或者删除。可以通过在该视图上创建一个INSTEAD OF触发器来获得可更新视图的效果，该触发器必须把该视图上的尝试的插入等转换成其他表上合适的动作。更多信息请见CREATE TRIGGER。另一种可能性是创建规则（见CREATE RULE），不过实际中触发器更容易理解和正确使用。

注意在视图上执行插入、更新或删除的用户必须具有该视图上相应的插入、更新或删除特权。此外，视图的拥有者必须拥有底层基本关系上的相关特权，但是执行更新的用户并不需要底层基本关系上的任何权限（见 Section 39.5）。

## 示例

创建一个由所有喜剧电影组成的视图：

```
CREATE VIEW comedies AS
  SELECT *
  FROM films
  WHERE kind = 'Comedy';
```

创建的视图包含创建时film表中的列。尽管\* 被用来创建该视图，后来被加入到该表中的列不会成为该视图的组成部分。

创建带有LOCAL CHECK OPTION的视图：

```
CREATE VIEW universal_comedies AS
  SELECT *
  FROM comedies
  WHERE classification = 'U'
  WITH LOCAL CHECK OPTION;
```

这将创建一个基于comedies视图的视图，只显示 kind = 'Comedy' 和classification = 'U' 的电影。如果新行没有classification = 'U'，在该视图中的任何 INSERT或UPDATE尝试将被拒绝，但是电影的kind将不会被检查。

用CASCADED CHECK OPTION创建一个视图：

```
CREATE VIEW pg_comedies AS
  SELECT *
  FROM comedies
  WHERE classification = 'PG'
  WITH CASCADED CHECK OPTION;
```

这将创建一个检查新行的kind和classification 的视图。

创建一个由可更新列和不可更新列混合而成的视图：

```
CREATE VIEW comedies AS
  SELECT f.*,
         country_code_to_name(f.country_code) AS country,
         (SELECT avg(r.rating)
          FROM user_ratings r
          WHERE r.film_id = f.id) AS avg_rating
  FROM films f
  WHERE f.kind = 'Comedy';
```

这个视图将支持INSERT、UPDATE 以及DELETE。所有来自于films表的列都将是可更新的，而计算列country和avg\_rating 将是只读的。

创建一个由数字 1 到 100 组成的递归视图：

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
```

```
VALUES (1)
UNION ALL
SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

注意在这个CREATE中尽管递归的视图名称是方案限定的，但它内部的自引用不是方案限定的。这是因为隐式创建的CTE的名称不能是方案限定的。

## 兼容性

CREATE OR REPLACE VIEW是一种 LightDB的语言扩展。临时 视图的概念也是这样。WITH (... )子句也是一种扩展。

## 另见

ALTER VIEW, DROP VIEW, CREATE MATERIALIZED VIEW

---

# DEALLOCATE

DEALLOCATE — 释放一个预备语句

## Synopsis

```
DEALLOCATE [ PREPARE ] { name | ALL }
```

## 描述

DEALLOCATE被用来释放一个之前准备好的 SQL 语句。如果不显式地释放一个预备语句，会话结束时会自动释放它。

更多关于预备语句的信息请见PREPARE。

## 参数

PREPARE

这个关键词会被忽略。

name

要释放的预备语句的名称。

ALL

释放所有预备语句。

## 兼容性

SQL 标准包括一个DEALLOCATE语句，但是只用于嵌入式 SQL。

## 另见

EXECUTE, PREPARE

---

# DECLARE

DECLARE — 定义一个游标

## Synopsis

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]  
          CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

## 描述

DECLARE允许用户创建游标，游标可以被用来在大型查询暂停时检索少量的行。游标被创建后，可以用FETCH从中取得行。

### Note

这个页面描述在 SQL 命令层面上游标的用法。如果想要在 PL/pgSQL函数中使用游标，其规则是不同的 — 详见Section 41.7

## 参数

name

要创建的游标的名称。

BINARY

让游标返回二进制数据而不是返回文本格式数据。

INSENSITIVE

游标敏感性决定了是否更改为游标下面的数据，在同一事务中完成，当游标被声明后，在游标中可见。INSENSITIVE 表示它们不可见，ASENSITIVE表示行为是实现相关的。第三个行为，SENSITIVE，意味着此类更改在游标中可见，但在LightDB中不可用。在LightDB中，所有游标都不敏感；因此这些关键字没有任何影响，并且只在兼容SQL标准时才被接受。

SCROLL

NO SCROLL

SCROLL指定游标可以用非顺序（例如，反向）的方式从中检索行。根据查询的执行计划的复杂度，指定 SCROLL可能导致查询执行时间上的性能损失。NO SCROLL指定游标不能以非顺序的方式从中检索行。默认是允许在某些情况下滚动，但这和指定 SCROLL不完全相同。详情请见 Notes。

WITH HOLD

WITHOUT HOLD

WITH HOLD指定该游标在创建它的事务提交之后还能被继续使用。WITHOUT HOLD指定该游标不能在创建它的事务之外使用。如果两者都没有指定，则默认为 WITHOUT HOLD。

query

用于提供该游标返回的行的SELECT或者VALUES命令。

关键词ASENSITIVE、BINARY、INSENSITIVE和SCROLL可以以任意顺序出现。

## 注解

普通游标以文本格式返回数据，这和SELECT产生的数据一样。 BINARY选项指定游标应该以二进制格式返回数据。这减少了服务器和客户端的转换负担，但程序员需要付出更多工作来处理与平台相关的二进制数据格式。例如，如果一个查询从一个整数列中返回一个值1，用一个默认游标将得到一个字符串1，而使用一个二进制游标将得到该值的四字节内部表示（big-endian 大端字节顺序）。

使用二进制游标时应该小心。很多应用（包括 ltsql）还没有准备好处理二进制游标，它们仍然期待数据以文本格式到来。

### Note

当客户端应用使用“扩展查询”协议发出一个 FETCH命令，绑定协议消息会指定使用文本还是二进制格式检索数据。这种选择会覆盖定义游标时指定的方式。因此 在使用扩展查询协议时，这样一个二进制游标的概念实际是被废弃的 — 任何游标都可以被视作文本或者二进制。

除非指定了WITH HOLD，这个命令创建的游标只能在当前事务中使用。 因此，没有WITH HOLD的DECLARE在事务块外是没有用的：游标只会生存到该语句结束。 因此如果这种命令在事务块之外被使用，LightDB会报告一个错误。 定义事务块需要使用BEGIN和COMMIT（或者ROLLBACK）。

如果指定了WITH HOLD并且创建游标的事务 成功提交，在同一个会话中的后续事务中还能够继续访问该游标（但是如果创建事务被中止，游标会被移除）。一个用 WITH HOLD创建的游标可以用一个显式的 CLOSE命令关闭，或者会话结束时它 也会被关闭。在当前的实现中，由一个被保持游标表示的行会被复制到一个临时文件或者内存区域中，这样它们才会在后续事务中保持可用。

当查询包括FOR UPDATE或FOR SHARE时，不能指定WITH HOLD。

在定义一个将被反向取元组的游标时，应该指定SCROLL 选项。这是 SQL 标准所要求的。不过，为了和早期版本兼容，如果游标的查询计划足够简单到支持它不需要额外的开销，LightDB会允许在没有 SCROLL的情况下反向取元组。不过，建议应用开发者不要依赖于从没有用SCROLL创建的游标中反向取元组。如果指定了NO SCROLL，那么任何情况下都不允许反向取元组。

当查询包括FOR UPDATE或FOR SHARE时，也不允许反向取元组。因此在这种情况下不能指定SCROLL。

### Caution

如果可滚动游标调用了任何不稳定的函数（见Section 36.7，它们可能给出预期之外的结果。当重新取得一个之前取得过的行时，那些函数会被重新执行，这可能导致得到与第一次不同的结果。它最好指定NO SCROLL对频繁调用的易变函数。如果那样不可行，一个变通方法是，声明游标为SCROLL WITH HOLD并且在从其中读任何行之前提交事务。这将强制该游标的整个输出被物化在临时存储中，这样针对每一行只会执行一次不稳定函数。

如果游标的查询包括FOR UPDATE或者FOR SHARE，那么被返回的行会在它们第一次被取得时被锁定，这和带有这些选项的常规SELECT命令一样。此外，被返回的行将是最新的版本。

### Caution

如果游标要和UPDATE ... WHERE CURRENT OF或者 DELETE ... WHERE CURRENT OF一起使用，通常推荐 使用FOR UPDATE。使用FOR UPDATE可以 阻止其他会话在行被取得和被更新之间修改行。如果没有 FOR UPDATE，当行在游标创建后被更改后，一个后续的 WHERE CURRENT OF命令将不会产生效果。

另一个使用FOR UPDATE的原因是，如果没有它，当游标查询不符合 SQL 标准的“简单可更新”规则时，后续的 WHERE CURRENT OF可能会失败（特别地，该游标必须只引用一个 表并且没有使用分组或者ORDER BY）。不是简单可更新的游标可能 成功也可能不成功，这取决于计划选择的细节。因此在最坏的情况下，应用可能会 在测试时成功但是在生产中失败。如果指定了FOR UPDATE， 则保证游标是可更新的。

不把FOR UPDATE和WHERE CURRENT OF一起用的主要原因是，需要游标时可滚动的，或者与当前的更新隔离（也就是说，继续显示旧的数据）。 如果这是你的需求，应密切关注安上述警示。

SQL 标准只对嵌入式SQL中的游标做出了规定。 LightDB服务器没有为游标实现 OPEN语句。当游标被声明时就被认为已经 被打开。不过，ECPG（ LightDB的嵌入式 SQL 预处理器）支持标准 SQL 游标习惯，包括那些DECLARE 和OPEN语句。

你可以通过查询pg\_cursors 系统视图可以看到所有可用的游标。

## 示例

声明一个游标：

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

更多游标的例子请见FETCH。

## 兼容性

SQL 标准规定，游标是否默认对底层数据的并发更新敏感取决于实现。在 LightDB 中，默认情况下游标是不敏感的， 可以通过指定 FOR UPDATE 使其变得敏感。其他产品可能有不同的工作方式。

SQL 标准只允许在嵌入式 SQL 和模块中使用游标。 LightDB 允许交互式使用游标。

二进制游标是一种LightDB 扩展。

## 另见

CLOSE, FETCH, MOVE

---

# DELETE

DELETE — 删除一个表的行

## Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

## 描述

DELETE从指定表中删除满足 WHERE子句的行。如果WHERE 子句没有出现，效果将会是删除表中的所有行。结果是一个合法的空表。

### Tip

TRUNCATE提供移除表中所有行的快速机制。

有两种方式可以使用数据库中其他表中包含的信息来删除一个表的行：使用子选择或者在USING子句中指定额外的表。哪种技术更合适取决于特定的环境。

可选的RETURNING子句导致DELETE 基于实际被删除的每一行计算并且返回值。任何使用被删除表列或者 USING中提到的其他表的列的表达式都可以被计算。RETURNING列表的语法和SELECT的 输出列表语法相同。

要从表中删除行，你必须具有其上的DELETE特权，以及USING子句中任何表以及其值在condition中被读取的表上的 SELECT特权。

如果指定了 ORDER BY 子句，可以按指定的顺序删除行。它的语义类似于 SELECT 中的 ORDER BY 子句。请注意，不支持带有 ORDER BY 子句的继承表删除，也不支持 Canopy。

如果指定了 LIMIT 或 OFFSET 子句，则可以删除行的子集。它的语义类似于 SELECT 中的 LIMIT 子句。请注意，不支持带有 LIMIT 或 OFFSET 子句的继承表更新，也不支持 Canopy。

## 参数

with\_query

WITH子句允许你指定一个或者多个子查询，在 DELETE查询中可以用子查询的名字来引用它们。详见Section 7.8和SELECT。

table\_name

要从其中删除行的表名（可以是模式限定的）。如果在表名前指定 ONLY，只会从提到的表中删除匹配的行。如果没有指定 ONLY，还会删除该表的任何继承表中的匹配行。可选地，可以在表名后面指定\*来显式指定要包括继承表。



alias

目标表的一个别名。提供别名时，它会完全隐藏该表的真实名称。例如，对于DELETE FROM foo AS f，DELETE语句的剩余部分都会用 f而不是 foo来引用该表。

from\_item

一个表的表达式允许在WHERE条件中出现来自其他表的列。这使用与SELECT语法的FROM子句相同的语法；例如，可以指定表名的别名。除非您希望建立自联接（在这种情况下，它必须与 from\_item中的别名一起出现），否则不要将目标表重复为from\_item。

condition

一个返回boolean类型值的表达式。只有让这个表达式返回true的行才将被删除。

cursor\_name

要在WHERE CURRENT OF情况中使用的游标的名称。最近一次从这个游标中取出的行将被删除。该游标必须是DELETE的目标表上的非分组查询。注意不能在使用WHERE CURRENT OF的同时指定一个布尔条件。有关将游标用于WHERE CURRENT OF的更多信息请见DECLARE。

output\_expression

在每一行被删除后，会被DELETE计算并且返回的表达式。该表达式可以使用table\_name以及USING中的表的任何列。写成\*可以返回所有列。

output\_name

被返回列的名称。

## 输出

在成功完成时，一个DELETE命令会返回以下形式的命令标签：

```
DELETE count
```

count是被删除行的数目。注意如果有一个BEFORE DELETE触发器抑制删除，那么该数目可能小于匹配condition的行数。如果count为0，表示查询没有删除行（这并非一种错误）。

如果DELETE命令包含RETURNING子句，则结果会与包含有RETURNING列表中定义的列和值的SELECT语句结果相似，这些结果是在被该命令删除的行上计算得来。

## 注解

通过在USING子句中指定其他的表，LightDB允许在WHERE条件中引用其他表的列。例如，要删除有一个给定制片人制作的所有电影，可以这样做：

```
DELETE FROM films USING producers
WHERE producer_id = producers.id AND producers.name = 'foo';
```

这里实际发生的事情是在films和producers之间进行连接，然后删除所有成功连接的films行。这种语法不属于标准。更标准的方式是：

```
DELETE FROM films
```

```
WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

在一些情况下，连接形式比子查询形式更容易书写或者执行更快。

## 示例

删除所有电影，但音乐剧除外：

```
DELETE FROM films WHERE kind <> 'Musical';
```

清空表films：

```
DELETE FROM films;
```

删除已完成的任務，返回被刪除行的明細：

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

刪除tasks中游標c\_tasks 當前位於其上的行：

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

刪除 citys 表中按 id 排序的前 5 行。

```
DELETE FROM citys order by id LIMIT 5;
```

## 兼容性

這個命令符合SQL標準，不過 USING和RETURNING子句是 LightDB擴展，在 DELETE中使用WITH也是擴展。

FROM 自 LightDB 23.2開始可選，詳細使用示例如下：

清除表films：

```
DELETE films;
```

刪除表中除kind不等musicals之外的所有行：

```
DELETE films WHERE kind <> 'Musical';
```

## 又見

TRUNCATE

---

# DISCARD

DISCARD — 抛弃会话状态

## Synopsis

```
DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }
```

## 描述

DISCARD释放与一个数据库会话相关的内部资源。这个命令有助于部分或者完全重置该会话的状态。有几个子命令来释放不同类型的资源。DISCARD ALL变体把所有其他形式都包含在内，并且还会重置额外的状态。

## 参数

PLANS

释放所有已缓存的查询计划，强制在下次使用相关预备语句时重新做计划。

SEQUENCES

丢弃所有已缓存的序列相关的状态，包括 `currval()`/`lastval()`信息 以及任何还未被`nextval()`返回的预分配的序列值（预分配序列值的描述请见 `CREATE SEQUENCE`）；

TEMPORARY or TEMP

删除当前会话中创建的所有临时表。

ALL

释放与当前会话相关的所有临时资源并且把会话重置为初始状态。当前这和执行以下语句序列的效果相同：

```
CLOSE ALL;  
SET SESSION AUTHORIZATION DEFAULT;  
RESET ALL;  
DEALLOCATE ALL;  
UNLISTEN *;  
SELECT pg_advisory_unlock_all();  
DISCARD PLANS;  
DISCARD TEMP;  
DISCARD SEQUENCES;
```

## 注解

DISCARD ALL不能在事务块内执行。

## 兼容性

DISCARD是一种 LightDB扩展。

---

## D0

D0 — 执行一个匿名代码块

### Synopsis

```
D0 [ (type [, ...]) USING (argument [, ...]) ] [ LANGUAGE lang_name ] code
```

### 描述

D0执行一个匿名代码块，或者换句话说 执行一个以一种过程语言编写的瞬时匿名函数。

代码块就好像是一个没有参数并且返回void的函数的函数体。 它会被在一次时间内解析并且执行。

通过使用 USING 子句，代码块被当成是一个带输入输出参数的函数体那样执行，并返回 record。 与 prepareStatement 一起用时，它可以只被解析一次，然后可以多次执行，但 do stmt 的解析花费很小，所以意义不大，故一般只用于 ecpg 中。

可选的USING子句可以写在代码块之前或者之后。但 LANGUAGE 只能为 plorasql。

可选的LANGUAGE子句可以写在代码块之前或者之后。

### 参数

code

要被执行的过程语言代码。就像在 CREATE FUNCTION中一样，必须把它指定为一个 字符串。推荐使用一个美元引用的文本。

lang\_name

编写该代码的过程语言的名称。如果省略，默认为plpgsql。

type

参数类型，参数的模式总是输入输出型。

argument

实际用于执行匿名执行块的参数，类似 CALL。

### 注解

要使用的过程语言必须已经用CREATE EXTENSION安装在 当前数据库中。默认已经安装了plpgsql，但是其他语言没有被 安装。

用户必须拥有该过程语言的USAGE特权，如果该语言 是不可信的则必须是一个超级用户。这和创建一个该语言的函数对 特权的要求相同。

如果在事务块中执行D0，过程代码则无法执行事务控制语句。只有在自己的事务中执行D0时，才允许使用事务控制语句。

### 例子

把模式public中所有视图上的所有特权授予 给角色webuser:

```

DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
              WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' ||
quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;

```

在 LANGUAGE plorasql 下 通过参数把模式public中所有视图上的所有特权授予 给角色webuser:

```

DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
              WHERE table_type = $1 AND table_schema = $2
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' ||
quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$ (text, text) USING ('VIEW', 'public') LANGUAGE plorasql;

```

在 ecpg 中的使用案例, 请参见 Section 34.5.4

## 兼容性

SQL 标准中没有DO语句。

## 另见

CREATE LANGUAGE

---

# DROP ACCESS METHOD

DROP ACCESS METHOD — 移除一种访问方法

## Synopsis

```
DROP ACCESS METHOD [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 简介

DROP ACCESS METHOD移除一种现有的访问方法。只有超级用户能够删除访问方法。

## 参数

IF EXISTS

如果该访问方法不存在，则不会抛出错误。这种情况下会发出一个提示。

name

一种现有的访问方法的名称。

CASCADE

自动删除依赖于该访问方法的对象（例如操作符类、操作符族以及索引），并且接着删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该访问方法，则拒绝删除它。这是默认设置。

## 示例

删除访问方法heptree:

```
DROP ACCESS METHOD heptree;
```

## 兼容性

DROP ACCESS METHOD是一种LightDB扩展。

## 另见

CREATE ACCESS METHOD

---

# DROP AGGREGATE

DROP AGGREGATE — 移除一个聚集函数

## Synopsis

```
DROP AGGREGATE [ IF EXISTS ] name ( aggregate_signature ) [, ...] [ CASCADE |  
RESTRICT ]
```

这里aggregate\_signature是:

```
* |  
[ argmode ] [ argname ] argtype [ , ... ] |  
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname  
] argtype [ , ... ]
```

## 描述

DROP AGGREGATE移除一个现有的 聚集函数。要执行这个命令，当前用户必须是该聚集函数的所有者。

## 参数

IF EXISTS

如果该聚集不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有聚集函数的名称（可以是模式限定的）。

argmode

一个参数的模式：IN或VARIADIC。 如果被忽略，默认值是IN。

argname

一个参数的名称。注意DROP AGGREGATE 并不真正关心参数名称，因为决定聚集函数的身份时只需要参数数据类型。

argtype

该聚集函数所操作的一个输入数据类型。要引用一个零参数的聚集函数，写 \*来替代参数说明列表。要引用一个有序集聚集函数，在直接和 聚集参数说明之间写上ORDER BY。

CASCADE

自动删除依赖于该聚集函数的对象（例如使用它的视图），然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该聚集函数，则拒绝删除它。这是默认值。

## 注解

ALTER AGGREGATE下描述了另一种引用有序集聚集的语法。

## 示例

要为类型integer移除聚集函数myavg:

```
DROP AGGREGATE myavg(integer);
```

要移除假想聚集函数myrank，该函数接收一个排序列的任意列表和直接参数的一个匹配的列表:

```
DROP AGGREGATE myrank(VARIADIC "any" ORDER BY VARIADIC "any");
```

要在一个命令中删除多个聚合函数:

```
DROP AGGREGATE myavg(integer), myavg(bigint);
```

## 兼容性

在 SQL 标准中没有DROP AGGREGATE语句。

## 另见

ALTER AGGREGATE, CREATE AGGREGATE



---

# DROP CAST

DROP CAST — 移除一个造型

## Synopsis

```
DROP CAST [ IF EXISTS ] (source_type AS target_type) [ CASCADE | RESTRICT ]
```

## 描述

DROP CAST移除一个之前定义好的造型。

要能删除一个造型，你必须拥有源数据类型或目标数据类型。这也是 创建一个造型所要求的特权。

## 参数

IF EXISTS

如果该造型不存在则不要抛出一个错误，而是发出一个提示。

source\_type

该造型的源数据类型的名称。

target\_type

该造型的目标数据类型的名称。

CASCADE

RESTRICT

这些关键词没有任何效果，因为在造型上没有依赖性。

## 示例

要移除从类型text到类型int的造型：

```
DROP CAST (text AS int);
```

## 兼容性

DROP CAST命令符合 SQL 标准。

## 另见

CREATE CAST

---

# DROP COLLATION

DROP COLLATION — 移除一个排序规则

## Synopsis

```
DROP COLLATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 描述

DROP COLLATION移除一个之前定义好的排序规则。要能删除一个排序规则，你必须拥有它。

## 参数

IF EXISTS

如果该排序规则不存在则不要抛出一个错误，而是发出一个提示。

name

排序规则的名称。排序规则名称可以是模式限定的。

CASCADE

自动删除依赖于该排序规则的对象，然后删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该排序规则，则拒绝删除它。这是默认值。

## 示例

要删除名为german的排序规则：

```
DROP COLLATION german;
```

## 兼容性

除了IF EXISTS选项之外，DROP COLLATION命令符合SQL标准。该选项是一个LightDB扩展。

## 另见

ALTER COLLATION, CREATE COLLATION

---

# DROP CONVERSION

DROP CONVERSION — 移除一个转换

## Synopsis

```
DROP CONVERSION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 描述

DROP CONVERSION移除一个之前定义好的 转换。要能删除一个转换，你必须拥有该转换。

## 参数

IF EXISTS

如果该转换不存在则不要抛出一个错误，而是发出一个提示。

name

转换的名称。转换名称可以是模式限定的。

CASCADE

RESTRICT

这些关键词没有任何效果，因为在转换上没有依赖性。

## 示例

要删除名为myname的转换：

```
DROP CONVERSION myname;
```

## 兼容性

在 SQL 标准中没有DROP CONVERSION语句，但是有一个DROP TRANSLATION语句。还有 对应的CREATE TRANSLATION语句，它与 LightDB 中的CREATE CONVERSION 语句相似。

## 另见

ALTER CONVERSION, CREATE CONVERSION

---

# DROP DATABASE

DROP DATABASE — 移除一个数据库

## Synopsis

```
DROP DATABASE [ IF EXISTS ] name [ [ WITH ] ( option [, ...] ) ]
```

其中选项 可以是：

FORCE

## 描述

DROP DATABASE移除一个数据库。它会 移除该数据库的系统目录项并且删除包含数据的文件目录。它只能由数据库 所有者执行。当你已经连接到目标数据库时，它不能被执行（连接到postgres或者任何其他数据库来发出这个命令）。另外，如果其他任何人已经连接到目标数据库，这个命令将会失败，除非您 使用以下所述的FORCE选项。

DROP DATABASE不能被撤销。请小心使用！

## 参数

IF EXISTS

如果该数据库不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的数据库的名称。

FORCE

尝试终止与目标数据库的所有现有连接。如果目标数据库中存在准备好的事务、活跃的逻辑复制槽或订阅，则不会终止。

如果当前用户没有终止其他连接的权限，则此操作将失败。所需权限与 Section 9.27.2中描述的 `pg_terminate_backend`相同。如果我们不能 终止连接，则此操作也将会失败。

## 注解

DROP DATABASE不能在一个事务块内执行。

在连接到目标数据库时，这个命令不能被执行。因此，使用程序 `dropdb`会更方便，它是这个命令的一个包装器。

## 兼容性

SQL 标准中没有DROP DATABASE语句。

## 另见

CREATE DATABASE

---

# DROP DOMAIN

DROP DOMAIN — 移除一个域

## Synopsis

```
DROP DOMAIN [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP DOMAIN 移除一个域。只有域的拥有者 才能移除它。

## 参数

IF EXISTS

如果该域不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有域的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该域的对象（例如表列），然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该域，则拒绝删除它。这是默认值。

## 示例

要移除域box:

```
DROP DOMAIN box;
```

## 兼容性

除了IF EXISTS选项，这个命令符合 SQL 标准。该选项 是一个LightDB扩展。

## 另见

CREATE DOMAIN, ALTER DOMAIN

---

# DROP EVENT TRIGGER

DROP EVENT TRIGGER — 移除一个事件触发器

## Synopsis

```
DROP EVENT TRIGGER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 描述

DROP EVENT TRIGGER 移除一个已有的 事件触发器。要执行这个命令，当前用户必须是事件触发器的拥有者。

## 参数

IF EXISTS

如果该事件触发器不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的事件触发器的名称。

CASCADE

自动删除依赖于该触发器的对象，然后删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该触发器，则拒绝删除它。这是默认值。

## 示例

销毁触发器snitch:

```
DROP EVENT TRIGGER snitch;
```

## 兼容性

在 SQL 标准中没有DROP EVENT TRIGGER语句。

## 另见

CREATE EVENT TRIGGER, ALTER EVENT TRIGGER

---

# DROP EXTENSION

DROP EXTENSION — 移除一个扩展

## Synopsis

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP EXTENSION从数据库移除扩展。删除一个扩展会导致其组成对象也被删除。

你必须拥有该扩展才能使用DROP EXTENSION。

## 参数

IF EXISTS

如果该扩展不存在则不要抛出一个错误，而是发出一个提示。

name

一个已安装扩展的名称。

CASCADE

自动删除依赖于该扩展的对象，然后删除所有  
(见Section 5.14)。

依赖于那些对象的对象

RESTRICT

如果有任何对象依赖于该扩展（而不是它自己拥有的成员  
个DROP命令中的扩展），则拒绝删除它。这是默认值。

对象和其他被列在同一个

## 示例

要从当前数据库移除扩展hstore:

```
DROP EXTENSION hstore;
```

如果hstore的任何对象在该数据库库中 正在使用，例如有一个表的列是hstore类型，这个命令都将会失败。加上CASCADE选项可以强制 把这些依赖对象也移除。

## 兼容性

DROP EXTENSION是一个 LightDB扩展。

## 另见

CREATE EXTENSION, ALTER EXTENSION

---

# DROP FOREIGN DATA WRAPPER

DROP FOREIGN DATA WRAPPER — 移除一个外部数据包装器

## Synopsis

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP FOREIGN DATA WRAPPER 移除一个已有的外部数据包装器。要执行这个命令，当前用户必须是该外部数据包装器的拥有者。

## 参数

IF EXISTS

如果该外部数据包装器不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有外部数据包装器的名称。

CASCADE

自动删除依赖于该外部数据包装器的对象（例如服务器），然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该外部数据包装器，则拒绝删除它。这是默认值。

## 示例

删除外部数据包装器dbi:

```
DROP FOREIGN DATA WRAPPER dbi;
```

## 兼容性

DROP FOREIGN DATA WRAPPER 符合 ISO/IEC 9075-9 (SQL/MED)。IF EXISTS子句 是一个 LightDB扩展。

## 另见

CREATE FOREIGN DATA WRAPPER, ALTER FOREIGN DATA WRAPPER



---

# DROP FOREIGN TABLE

DROP FOREIGN TABLE — 移除一个外部表

## Synopsis

```
DROP FOREIGN TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP FOREIGN TABLE 移除一个 外部表。只有一个外部表的拥有者才能移除它。

## 参数

IF EXISTS

如果该外部表不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的外部表的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该外部表的对象（例如视图），然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该外部表，则拒绝删除它。这是默认值。

## 示例

要销毁两个外部表films 和distributors:

```
DROP FOREIGN TABLE films, distributors;
```

## 兼容性

这个命令符合 ISO/IEC 9075-9 (SQL/MED)，不过该标准只允许每个命令 中删除一个外部表并且没有IF EXISTS选项。该选项是一个 LightDB扩展。

## 另见

ALTER FOREIGN TABLE, CREATE FOREIGN TABLE

---

# DROP FUNCTION

DROP FUNCTION — 移除一个函数

## Synopsis

```
DROP FUNCTION [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype  
[ , ... ] ) ) ] [ , ... ]  
[ CASCADE | RESTRICT ]
```

## 描述

DROP FUNCTION 移除一个已有函数的定义。要执行这个命令用户必须是该函数的所有者。该函数的参数类型必须被指定，因为多个不同的函数可能会具有相同的函数名和不同的参数列表。

## 参数

IF EXISTS

如果该函数不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有函数的名称（可以是模式限定的）。如果未指定参数列表，则该名称在其模式中必须是唯一的。

argmode

一个参数的模式：IN、OUT、INOUT 或者 VARIADIC。如果被忽略，则默认为 IN。注意 DROP FUNCTION 并不真正关心 OUT 参数，因为决定函数的身份时只需要输入参数。因此列出 IN、INOUT 和 VARIADIC 参数足以。

argname

一个参数的名称。注意 DROP FUNCTION 并不真正关心参数名称，因为决定函数的身份时只需要参数的数据类型。

argtype

如果函数有参数，这是函数参数的数据类型（可以是模式限定的）。

CASCADE

自动删除依赖于该函数的对象（例如操作符和触发器），然后删除所有依赖于那些对象的对象（见 Section 5.14）。

RESTRICT

如果有任何对象依赖于该函数，则拒绝删除它。这是默认值。

## 示例

这个命令移除平方根函数：

```
DROP FUNCTION sqrt(integer);
```

在一个命令中删除多个函数:

```
DROP FUNCTION sqrt(integer), sqrt(bigint);
```

如果函数名称在其模式中是唯一的, 则可以在不带参数列表的情况下引用它:

```
DROP FUNCTION update_employee_salaries;
```

请注意, 这与

```
DROP FUNCTION update_employee_salaries();
```

不同, 后者引用一个零个参数的函数, 而第一个变体才可以引用具有任意数量参数的函数, 包括零, 只要该名称是唯一的。

## 兼容性

该命令符合SQL标准, 使用这些LightDB扩展:

- 该标准只允许每个命令删除一个函数。
- IF EXISTS选项
- 能够指定参数模式和名称

## 另见

CREATE FUNCTION, ALTER FUNCTION, DROP PROCEDURE, DROP ROUTINE

---

# DROP GROUP

DROP GROUP — 移除一个数据库角色

## Synopsis

```
DROP GROUP [ IF EXISTS ] name [, ...]
```

## 说明

DROP GROUP现在是 DROP ROLE的一个别名。

## 兼容性

在SQL标准中没有DROP GROUP语句。

## 另见

DROP ROLE

---

# DROP INDEX

DROP INDEX — 移除一个索引

## Synopsis

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...] [ ON tname ] [ CASCADE  
| RESTRICT ]
```

## 说明

DROP INDEX从数据库系统中 移除一个已有的索引。要执行这个命令你必须是该索引的拥有者。

## 参数

CONCURRENTLY

删除索引并且不阻塞在索引基表上的并发选择、插入、更新和删除操作。一个普通的DROP INDEX会要求该表上的排他锁，这会阻塞其他访问直至索引删除完成。通过这个选项，该命令会等待直至冲突事务完成。

在使用这个选项时有一些需要注意的事情。只能指定一个索引名称，并且不支持CASCADE选项（因此，一个支持UNIQUE或者 PRIMARY KEY约束的索引不能以这种方式删除）。还有，常规的DROP INDEX命令可以在一个事务块内执行，而 DROP INDEX CONCURRENTLY不能。最后，不能使用此选项删除分区表上的索引。

对于临时表，DROP INDEX始终是非并发的，因为没有其他会话可以访问它们，而且非并发的索引移除代价更小。

IF EXISTS

如果该索引不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的索引名称（可以是模式限定的）。

tname

要删除的索引的表名(可选的表名)。但这是一种语法糖，相当于不使用`ON tname`。

CASCADE

自动删除依赖于该索引的对象，然后删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该索引，则拒绝删除它。这是默认值。

## 示例

这个命令将移除索引title\_idx:

```
DROP INDEX title_idx;
```

## 兼容性

DROP INDEX是一个 LightDB语言扩展。在 SQL 标准中没有提供索引。

## 另见

CREATE INDEX

---

# DROP LANGUAGE

DROP LANGUAGE — 移除一种过程语言

## Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 说明

DROP LANGUAGE 移除一种之前注册的过程语言 的定义。你必须是一个超级用户或者该语言的拥有者才能使用 DROP LANGUAGE。

### Note

自LightDB 9.1 起，大部分过程语言 已经被做成了“扩展”，因此应该用 DROP EXTENSION 而不是 DROP LANGUAGE 删除。

## 参数

IF EXISTS

如果该语言不存在则不要抛出一个错误，而是发出一个提示。

name

一个已有过程语言的名称。为了向前兼容，这个名称可以用单引号包围。

CASCADE

自动删除依赖于该语言的对象（例如该语言中的函数），然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该语言，则拒绝删除它。这是默认值。

## 示例

这个命令移除过程语言plsample:

```
DROP LANGUAGE plsample;
```

## 兼容性

在SQL标准中没有DROP LANGUAGE语句。

## 另见

ALTER LANGUAGE, CREATE LANGUAGE

---

# DROP MATERIALIZED VIEW

DROP MATERIALIZED VIEW — 移除一个物化视图

## Synopsis

```
DROP MATERIALIZED VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP MATERIALIZED VIEW 删除一个 现有的物化视图。要执行这个命令你必须是该物化视图的拥有者。

## 参数

IF EXISTS

如果该物化视图不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的物化视图的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该物化视图的对象（例如其他物化视图或常规视图），然后删除所有 依赖于那些对象的对象（见Section 5.14 。

RESTRICT

如果有任何对象依赖于该物化视图，则拒绝删除它。这是默认值。

## 示例

这个命令将移除名为order\_summary的物化视图：

```
DROP MATERIALIZED VIEW order_summary;
```

## 兼容性

DROP MATERIALIZED VIEW 是一个 LightDB 扩展。

## 另见

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW



---

# DROP OPERATOR

DROP OPERATOR — 移除一个操作符

## Synopsis

```
DROP OPERATOR [ IF EXISTS ] name ( { left_type | NONE } , { right_type | NONE } ) [, ...] [ CASCADE | RESTRICT ]
```

## 说明

DROP OPERATOR从数据库系统中 删除一个现有的操作符。要执行这个命令，你必须该操作符的拥有者。

## 参数

IF EXISTS

如果该操作符不存在则不会抛出一个错误，而是发出一个提示。

name

一个现有的操作符的名称（可以是模式限定的）。

left\_type

该操作符左操作数的数据类型，如果没有左操作数就写NONE。

right\_type

该操作符右操作数的数据类型，如果没有右操作数就写NONE。

CASCADE

自动删除依赖于该操作符的对象（例如使用它的视图），然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该操作符，则拒绝删除它。这是默认值。

## 示例

为类型integer移除幂操作符 $a^b$ :

```
DROP OPERATOR ^ (integer, integer);
```

为类型bit移除左一元按位补操作符 $\sim b$ :

```
DROP OPERATOR ~ (none, bit);
```

为类型bigint移除右一元阶乘操作符 $x!$ :

DROP OPERATOR ! (bigint, none);

在一条命令中删除多个操作符:

DROP OPERATOR ~ (none, bit), ! (bigint, none);

## 兼容性

SQL标准中没有DROP OPERATOR语句。

## 另见

CREATE OPERATOR, ALTER OPERATOR

---

# DROP OPERATOR CLASS

DROP OPERATOR CLASS — 移除一个操作符类

## Synopsis

```
DROP OPERATOR CLASS [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

## 描述

DROP OPERATOR CLASS删除一个现有的 操作符类。要执行这个命令，你必须是该操作符类的拥有者。

DROP OPERATOR CLASS不会删除任何被 该类引用的操作符或者函数。如果有索引依赖于该操作符类，你将需要指定CASCADE来完成删除。

## 参数

IF EXISTS

如果该操作符类不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有的操作符类的名称（可以是模式限定的）。

index\_method

该操作符类适用的索引访问方法的名称。

CASCADE

自动删除依赖于该操作符类的对象（例如索引），然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该操作符类，则拒绝删除它。这是默认值。

## 注解

DROP OPERATOR CLASS将不会删除包含该类的 操作符族，即使该族中已经没有任何成员（特别是由 CREATE OPERATOR CLASS隐式创建的族）。一个 空操作符族是无害的，但是为了整洁你可能希望用 DROP OPERATOR FAMILY移除该操作符族，或者 一开始就使用DROP OPERATOR FAMILY会更好。

## 示例

移除 B-树操作符类widget\_ops:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

如果有任何使用该操作符类的索引存在，这个命令都不会成功。增加 CASCADE可以把这类索引与该操作符类一起删除。

## 兼容性

SQL 标准中没有DROP OPERATOR CLASS语句。

## 另见

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR FAMILY

---

# DROP OPERATOR FAMILY

DROP OPERATOR FAMILY — 移除一个操作符族

## Synopsis

```
DROP OPERATOR FAMILY [ IF EXISTS ] name USING index_method [ CASCADE |  
RESTRICT ]
```

## 描述

DROP OPERATOR FAMILY删除一个 现有的操作符族。要执行这个命令，你必须是该操作符族的拥有者。

DROP OPERATOR FAMILY包括删除 该族所包含的任何操作符类，但是它不会删除该族所引用的任何操作符或函数。如果有任何依赖于该族中操作符类的索引存在，你将需要 指定CASCADE来完成删除。

## 参数

IF EXISTS

如果该操作符族不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有操作符族的名称（可以是模式限定的）。

index\_method

该操作符族适用的索引访问方法的名称。

CASCADE

自动删除依赖于该操作符族的对象，然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该操作符族，则拒绝删除它。这是默认值。

## 示例

移除 B-树操作符族float\_ops:

```
DROP OPERATOR FAMILY float_ops USING btree;
```

如果有任何使用该族中操作符类的索引存在，这个命令都不会成功。增加 CASCADE可以把这类索引与该操作符族一起删除。

## 兼容性

SQL 标准中没有DROP OPERATOR FAMILY 语句。

另见

ALTER OPERATOR FAMILY, CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

---

# DROP OWNED

DROP OWNED — 移除一个数据库角色拥有的数据库对象

## Synopsis

```
DROP OWNED BY { name | CURRENT_USER | SESSION_USER } [, ...] [ CASCADE |  
RESTRICT ]
```

## 说明

DROP OWNED删除当前数据库中被指定角色之一拥有的所有对象。任何已被授予给定角色在当前数据库对象或者共享对象（数据库、表空间）上的特权也将会被收回。

## 参数

name

将被删除所拥有对象并且其特权将被收回的角色的名称。

CASCADE

自动删除依赖于受影响对象的对象，然后删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何其他数据库对象依赖于一个受影响的对象，则拒绝删除这个受影响角色所拥有的对象。这是默认值。

## 注解

DROP OWNED常常被用来为移除一个或者多个角色做准备。因为DROP OWNED只影响当前数据库中的对象，通常有必要在包含将被移除角色拥有对象的每一个数据库中都执行这个命令。

使用CASCADE选项可能导致这个命令递归去删除由其他用户所拥有的对象。

REASSIGN OWNED命令是另一种选择，它可以把一个或多个角色所拥有的所有数据库对象重新授予给其他角色。不过，REASSIGN OWNED不处理其他对象的特权。

角色所拥有的数据库和表空间将不会被移除。

更多讨论请见Section 19.4

## 兼容性

DROP OWNED命令是一个LightDB扩展。

## 另见

REASSIGN OWNED, DROP ROLE

---

# DROP POLICY

DROP POLICY — 从一个表移除一条行级安全性策略

## Synopsis

```
DROP POLICY [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

## 说明

DROP POLICY从该表移除指定的策略。注意如果从一个表移除了最后一条策略并且该表的行级安全性仍被ALTER TABLE启用，则默认的否定策略将被使用。不管该表的策略存在与否，ALTER TABLE ... DISABLE ROW LEVEL SECURITY都可以被用来禁用一个表的行级安全性。

## 参数

IF EXISTS

如果该策略不存在也不抛出一个错误。这种情况下会发出一个提示。

name

要删除的策略的名称。

table\_name

该策略所在的表的名称（可以被模式限定）。

CASCADE

RESTRICT

这些关键词不会产生效果，因为策略之上没有依赖关系。

## 例子

要在名为my\_table的表上删除策略p1:

```
DROP POLICY p1 ON my_table;
```

## 兼容性

DROP POLICY是一种LightDB扩展。

## 另见

CREATE POLICY, ALTER POLICY



---

# DROP PROCEDURE

DROP PROCEDURE — 移除一个过程

## Synopsis

```
DROP PROCEDURE [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype  
[ , ... ] ] ) ] [ , ... ]  
[ CASCADE | RESTRICT ]
```

## 说明

DROP PROCEDURE 移除一个现有过程的定义。为了执行这个命令，用户必须是该过程的拥有者。该过程的参数类型必须被指定，因为可能存在多个不同的过程具有相同名称和不同参数列表。

## 参数

IF EXISTS

如果该过程不存在也不抛出一个错误。这种情况下会发出一个提示。

name

现有过程的名称（可以被模式限定的）。如果没有指定参数列表，则该名称在其所属的模式中必须是唯一的。

argmode

参数的模式：IN或者VARIADIC。如果省略，默认为IN。

argname

参数的名称。注意，其实DROP PROCEDURE并不真正在意参数名称，因为只需要参数的数据类型来确定过程的身份。

argtype

该过程如果有参数，这里是参数的数据类型（可以是模式限定的）。

CASCADE

自动删除依赖于该过程的对象，然后接着删除依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该过程，则拒绝删除它。这是默认选项。

## 示例

```
DROP PROCEDURE do_db_maintenance();
```

## 兼容性

这个命令符合SQL标准，不过LightDB做了这些扩展：

- 标准仅允许每个命令删除一个过程。
- IF EXISTS选项
- 指定参数模式和名称的能力

## 另见

CREATE PROCEDURE, ALTER PROCEDURE, DROP FUNCTION, DROP ROUTINE

---

# DROP PUBLICATION

DROP PUBLICATION — 删除一个发布

## Synopsis

```
DROP PUBLICATION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP PUBLICATION从数据库中删除一个现有的发布。

发布只能被它自己的所有者或超级用户删除。

## 参数

IF EXISTS

如果发布不存在，不要抛出一个错误。在这种情况下发出一个提示。

name

现有发布的名称。

CASCADE

RESTRICT

这些关键词没有任何作用，因为发布没有依赖关系。

## 示例

删除一个发布：

```
DROP PUBLICATION mypublication;
```

## 兼容性

DROP PUBLICATION是一个LightDB 扩展。

## 又见

CREATE PUBLICATION, ALTER PUBLICATION

---

# DROP ROLE

DROP ROLE — 移除一个数据库角色

## Synopsis

```
DROP ROLE [ IF EXISTS ] name [, ...]
```

## 说明

DROP ROLE移除指定的角色。要删除一个超级用户角色，你必须自己就是一个超级用户。要删除一个非超级用户角色，你必须具有CREATEROLE特权。

如果一个角色仍然在集簇中任何数据库中被引用，它就不能被移除，如果尝试移除将会抛出一个错误。在删除该角色前，你必须删除（或者重新授予所有权）它所拥有的所有对象并且收回该已经授予给该角色的在其他对象上的特权。REASSIGN OWNED和DROP OWNED命令可以用于这个目的。更多讨论请见Section 19.4

不过，没有必要移除涉及该角色的角色成员关系。DROP ROLE会自动收回目标角色在其他角色中的成员关系以及其他角色在目标角色中的成员关系。其他角色不会被删除也不会被影响。

## 参数

IF EXISTS

如果该角色不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的角色的名称。

## 注解

LightDB包括一个具有和这个命令完全相同的功能（事实上它会调用这个命令）的程序dropuser，但是该程序可以从命令行运行。

## 示例

要删除一个角色：

```
DROP ROLE jonathan;
```

## 兼容性

SQL标准定义了DROP ROLE，但是它只允许一次删除一个角色并且它指定了和LightDB不同的特权需求。

## 另见

CREATE ROLE, ALTER ROLE, SET ROLE

---

# DROP ROUTINE

DROP ROUTINE — 删除一个例程

## Synopsis

```
DROP ROUTINE [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype  
[ , ... ] ] ) ] [ , ... ]  
[ CASCADE | RESTRICT ]
```

## 说明

DROP ROUTINE删除一个现有例程的定义，它可以是一个聚集函数、一个普通函数或者过程。有关参数的描述、更多的示例以及进一步的细节请参考DROP AGGREGATE、DROP FUNCTION以及DROP PROCEDURE。

## 示例

删除类型integer的例程foo:

```
DROP ROUTINE foo(integer);
```

不管foo是一个聚集、函数或是过程，这个命令都能起作用。

## 兼容性

这个命令符合SQL标准，不过LightDB做了下面这些扩展：

- 标准仅允许每个命令删除一个例程。
- IF EXISTS选项
- 指定参数模式和名称的能力
- 聚集函数是一种扩展。

## 另见

DROP AGGREGATE, DROP FUNCTION, DROP PROCEDURE, ALTER ROUTINE

注意CREATE ROUTINE命令不存在。

---

# DROP RULE

DROP RULE — 移除一个重写规则

## Synopsis

```
DROP RULE [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

## 描述

DROP RULE删除一个重写规则。

## 参数

IF EXISTS

如果该规则不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的规则的名称。

table\_name

该规则适用的表或视图的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该规则的对象，然后删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该规则，则拒绝删除它。这是默认值。

## 示例

要删除重写规则newrule:

```
DROP RULE newrule ON mytable;
```

## 兼容性

DROP RULE是一个 LightDB语言扩展，整个 查询重写系统也是这样。

## 另见

CREATE RULE, ALTER RULE

---

# DROP SCHEMA

DROP SCHEMA — 移除一个模式

## Synopsis

```
DROP SCHEMA [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP SCHEMA从数据库中移除模式。

一个模式只能由其所有者或一个超级用户删除。注意即使所有者不拥有该模式中的某些对象，也能删除该模式（以及所有含有的对象）。

## 参数

IF EXISTS

如果该模式不存在则不要抛出一个错误，而是发出一个提示。

name

一个模式的名称。

CASCADE

自动删除包含在该模式中的对象（表、函数等），然后删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果该模式含有任何对象，则拒绝删除它。这是默认值。

## 注解

使用CASCADE选项可能会使这条命令移除除指定模式之外其他模式中的对象。

## 示例

要从数据库中移除模式mystuff及其中所包含的对象：

```
DROP SCHEMA mystuff CASCADE;
```

## 兼容性

DROP SCHEMA完全符合 SQL 标准，不过该标准只允许在每个命令中删除一个模式并且没有 IF EXISTS选项。该选项是一个 LightDB扩展。

## 另见

ALTER SCHEMA, CREATE SCHEMA

---

# DROP SEQUENCE

DROP SEQUENCE — 移除一个序列

## Synopsis

```
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP SEQUENCE 移除序数生成器。 一个序列只能被其拥有者或超级用户删除。

## 参数

IF EXISTS

如果该序列不存在则不要抛出一个错误，而是发出一个提示。

name

一个序列的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该序列的对象，然后删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该序列，则拒绝删除它。这是默认值。

## 示例

要移除序列serial:

```
DROP SEQUENCE serial;
```

## 兼容性

DROP SEQUENCE符合 SQL标准，不过该标准只允许每个命令中删除一个序列并且没有IF EXISTS选项。该选项是一个 LightDB扩展。

## 另见

CREATE SEQUENCE, ALTER SEQUENCE



---

# DROP SERVER

DROP SERVER — 移除一个外部服务器描述符

## Synopsis

```
DROP SERVER [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP SERVER 移除一个现有的外部服务器 描述符。要执行这个命令，当前用户必须是该服务器的所有者。

## 参数

IF EXISTS

如果该服务器不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有服务器的名称。

CASCADE

自动删除依赖于该服务器的对象（例如用户映射），然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该服务器，则拒绝删除它。这是默认值。

## 示例

如果一个服务器foo存在则删除它：

```
DROP SERVER IF EXISTS foo;
```

## 兼容性

DROP SERVER符合 ISO/IEC 9075-9 (SQL/MED)。IF EXISTS子句是一个 LightDB扩展。

## 另见

CREATE SERVER, ALTER SERVER

---

# DROP STATISTICS

DROP STATISTICS — 删除扩展统计

## Synopsis

```
DROP STATISTICS [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 说明

DROP STATISTICS删除数据库中的统计对象。只有统计对象的所有者、模式的所有者或超级用户可以删除统计对象。

## 参数

IF EXISTS

如果要删除的统计对象不存在则不要抛出错误，这种情况下会发出一个提示。

name

要删除的统计对象的名称（可以是模式限定的）。

CASCADE

RESTRICT

由于没有统计依赖关系，这些关键字没有任何效果。

## 示例

删除不同模式中的两个统计对象，如果不存在时不会失败：

```
DROP STATISTICS IF EXISTS
  accounting.users_uid_creation,
  public.grants_user_role;
```

## 兼容性

SQL标准中没有DROP STATISTICS命令。

## 另见

ALTER STATISTICS, CREATE STATISTICS

---

# DROP SUBSCRIPTION

DROP SUBSCRIPTION — 删除一个订阅

## Synopsis

```
DROP SUBSCRIPTION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 说明

DROP SUBSCRIPTION删除数据库集群中的一个订阅。

只有超级用户才可以删除订阅。

如果订阅与复制槽相关联，就不能在事务块内部执行DROP SUBSCRIPTION。（可以使用ALTER SUBSCRIPTION取消关联复制槽。）

## 参数

name

要删除的订阅的名称。

CASCADE

RESTRICT

这些关键词没有任何作用，因为订阅之上没有依赖关系。

## 注解

在删除与远程主机（正常状态）上的复制槽相关联的订阅时，DROP SUBSCRIPTION将连接到远程主机，并尝试删除该复制槽作为其操作的一部分。这是必要的，以便释放远程主机上为订阅分配的资源。如果由于远程主机不可访问、远程复制槽不能被删除、复制槽不存在或没有存在过等原因删除复制槽失败，则DROP SUBSCRIPTION命令将失败。要在这种情况下继续，请执行ALTER SUBSCRIPTION ... SET (slot\_name = NONE)来解除复制槽与订阅的关联。之后，DROP SUBSCRIPTION将不会在远程主机上尝试任何操作。注意，如果远程复制槽仍然存在，则应手动删除它；否则它将继续保留WAL，最终可能导致磁盘空间被填满。另见Section 28.2.1

如果订阅与复制槽相关联，那么不能在事务块内部执行DROP SUBSCRIPTION。

## 示例

删除一个订阅：

```
DROP SUBSCRIPTION mysub;
```

## 兼容性

DROP SUBSCRIPTION是一个LightDB扩展。

## 另见

CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

---

# DROP TABLE

DROP TABLE — 移除一个表

## Synopsis

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE [CONSTRAINTS] | RESTRICT ]
```

## 说明

DROP TABLE从数据库移除表。只有表所有者、模式所有者和超级用户能删除一个表。要清空一个表中的行但是不销毁该表，可以使用DELETE或者TRUNCATE。

DROP TABLE总是移除目标表的任何索引、规则、触发器和约束。不过，要删除一个被视图或者另一个表的外键约束所引用的表，必须指定CASCADE（CASCADE将会把依赖的视图也完全移除，但是对于外键它将只移除外键约束，而不是完全移除其他表）。

## 参数

IF EXISTS

如果该表不存在则不要抛出一个错误，这种情况下会发出一个提示。

name

要删除的表的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该表的对象（例如视图），然后删除所有依赖于那些对象的对象（见Section 5.14）。

CASCADE CONSTRAINTS

自动删除依赖于该表的对象（例如视图），然后删除所有依赖于那些对象的对象。此为兼容Oracle语法，实际功能与CASCADE相同（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该表，则拒绝删除它。这是默认值。

## 示例

要销毁两个表films和distributors:

```
DROP TABLE films, distributors;
```

删除表films，并自动删除依赖于该表的对象:

```
DROP TABLE films CASCADE;
```

或

```
DROP TABLE films CASCADE CONSTRAINTS;
```

## 兼容性

这个命令符合SQL标准，不过该标准只允许每个命令删除一个表并且没有IF EXISTS选项。该选项是一个LightDB扩展。

## 另见

ALTER TABLE, CREATE TABLE

---

# DROP TABLESPACE

DROP TABLESPACE — 移除一个表空间

## Synopsis

```
DROP TABLESPACE [ IF EXISTS ] name
```

## 描述

DROP TABLESPACE从系统中移除一个表空间。

一个表空间只能被其所有者或超级用户删除。在一个表空间能被删除前，其中 必须没有任何数据库对象。即使当前数据库中没有对象正在使用该表空间，也 可能有其他数据库的对象存在于其中。还有，如果该表空间被列在任何活动会话的temp\_tablespaces设置中，DROP也可能会失败，因为可能有临时文件存在其中。

## 参数

IF EXISTS

如果该表空间不存在则不要抛出一个错误，而是发出一个提示。

name

一个表空间的名称。

## 注解

DROP TABLESPACE不能在一个事务块内执行。

## 示例

要从系统移除表空间mystuff:

```
DROP TABLESPACE mystuff;
```

## 兼容性

DROP TABLESPACE是一个 LightDB扩展。

## 另见

CREATE TABLESPACE, ALTER TABLESPACE

---

# DROP TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH CONFIGURATION — 移除一个文本搜索配置

## Synopsis

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 描述

DROP TEXT SEARCH CONFIGURATION 删除一个现有的文本搜索配置。要执行这个命令，你必须 是该配置的拥有者。

## 参数

IF EXISTS

如果该文本搜索配置不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索配置的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索配置的对象，然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该文本搜索配置，则拒绝删除它。这是默认值。

## 示例

移除文本搜索配置my\_english:

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

如果有任何在to\_tsvector调用中引用该配置的 索引存在，这个命令都不会成功。增加CASCADE 可以把这类索引与该文本搜索配置一起删除。

## 兼容性

SQL 标准中没有DROP TEXT SEARCH CONFIGURATION 语句。

## 另见

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION

---

# DROP TEXT SEARCH DICTIONARY

DROP TEXT SEARCH DICTIONARY — 移除一个文本搜索字典

## Synopsis

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 描述

DROP TEXT SEARCH DICTIONARY 删除一个 现有的文本搜索字典。要执行这个命令，你必须该字典的拥有者。

## 参数

IF EXISTS

如果该文本搜索字典不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索字典的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索字典的对象，然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该文本搜索字典，则拒绝删除它。这是默认值。

## 示例

移除文本搜索字典english:

```
DROP TEXT SEARCH DICTIONARY english;
```

如果有任何使用该字典的文本搜索配置存在，这个命令都不会成功。增加 CASCADE可以把这类配置与字典一起删除。

## 兼容性

SQL 标准中没有DROP TEXT SEARCH DICTIONARY 语句。

## 另见

ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY



---

# DROP TEXT SEARCH PARSER

DROP TEXT SEARCH PARSER — 移除一个文本搜索解析器

## Synopsis

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 描述

DROP TEXT SEARCH PARSER删除一个 现有的文本搜索解析器。你必须作为一个超级用户来使用这个命令。

## 参数

IF EXISTS

如果该文本搜索解析器不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索解析器的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索解析器的对象，然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该文本搜索解析器，则拒绝删除它。这是默认值。

## 示例

移除文本搜索解析器my\_parser:

```
DROP TEXT SEARCH PARSER my_parser;
```

如果有任何使用该解析器的文本搜索配置存在，这个命令都不会成功。增加 CASCADE可以把这类配置与解析器一起删除。

## 兼容性

SQL 标准中没有DROP TEXT SEARCH PARSER 语句。

## 另见

ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER

---

# DROP TEXT SEARCH TEMPLATE

DROP TEXT SEARCH TEMPLATE — 移除一个文本搜索模板

## Synopsis

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## 描述

DROP TEXT SEARCH TEMPLATE删除一个 现有的文本搜索模板。你必须作为一个超级用户来使用这个命令。

## 参数

IF EXISTS

如果该文本搜索模板不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索模板的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索模板的对象，然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该文本搜索模板，则拒绝删除它。这是默认值。

## 示例

移除文本搜索模板thesaurus:

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

如果有任何使用该模板的文本搜索字典存在，这个命令都不会成功。增加 CASCADE可以把这类字典与模板一起删除。

## 兼容性

SQL 标准中没有DROP TEXT SEARCH TEMPLATE 语句。

## 另见

ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE

---

# DROP TRANSFORM

DROP TRANSFORM — 移除转换

## Synopsis

```
DROP TRANSFORM [ IF EXISTS ] FOR type_name LANGUAGE lang_name [ CASCADE |  
RESTRICT ]
```

## 简介

DROP TRANSFORM 移除一个之前定义的转换。

为了删除一种转换，你必须拥有该类型和语言。这些同样也是创建转换所需要的 特权。

## 参数

IF EXISTS

如果该转换不存在也不要抛出一个错误。这种情况下会发出一个提示。

type\_name

该转换的数据类型的名称。

lang\_name

该转换的语言的名称。

CASCADE

自动删除依赖于该转换的对象，然后删除所有 依赖于那些对象的对象  
(见Section 5.14)。

RESTRICT

如果有任何对象依赖于该转换，则拒绝删除它。这是默认行为。

## 示例

要删除用于类型hstore和语言plpythonu 的转换：

```
DROP TRANSFORM FOR hstore LANGUAGE plpythonu;
```

## 兼容性

这种形式的DROP TRANSFORM是一种 LightDB扩展。详见 CREATE TRANSFORM。

## 另见

CREATE TRANSFORM

---

# DROP TRIGGER

DROP TRIGGER — 移除一个触发器

## Synopsis

```
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

## 描述

DROP TRIGGER移除一个现有的触发器定义。要执行这个命令，当前用户必须是触发器基表的拥有者。

## 参数

IF EXISTS

如果该触发器不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的触发器的名称。

table\_name

定义了该触发器的表的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该触发器的对象，然后删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该触发器，则拒绝删除它。这是默认值。

## 示例

销毁表films上的触发器 if\_dist\_exists:

```
DROP TRIGGER if_dist_exists ON films;
```

## 兼容性

LightDB中的 DROP TRIGGER语句与 SQL 标准不兼容。在 SQL 标准中，不同表上也不能有同名的触发器，因此其命令是简单的DROP TRIGGER name.

## 另见

CREATE TRIGGER

---

# DROP TYPE

DROP TYPE — 移除一个数据类型

## Synopsis

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

```
DROP TYPE BODY name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP TYPE 移除一种用户定义的数据类型。 只有一个类型的拥有者才能移除它。

DROP TYPE BODY 移除用户定义对象类型的函数定义。 只有一个类型的拥有者才能移除它。

## 参数

IF EXISTS

如果该类型不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的数据类型的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该类型的对象（例如表列、函数、操作符），然后删除所有依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该类型，则拒绝删除它。这是默认值。

## 示例

要移除数据类型box:

```
DROP TYPE box;
```

要移除数据类型体tp:

```
DROP TYPE BODY tp;
```

## 兼容性

这个命令类似于 SQL 标准中的对应命令，但IF EXISTS子句 是一个LightDB扩展。但要注意LightDB中 CREATE TYPE命令的很大部分以及数据类型扩展机制都与 SQL 标准不同。

## 另见

ALTER TYPE, CREATE TYPE

---

# DROP USER

DROP USER — 移除一个数据库角色

## Synopsis

```
DROP USER [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 说明

DROP USER现在是DROP ROLE另一种写法。

## 参数

CASCADE

用于与 Oracle 兼容的 DROP USER 语句。自动删除包含在用户中的对象（表、函数等），以及依赖于这些对象的所有对象（请参阅 Section 5.14）。必须在 Oracle 数据库中使用。

RESTRICT

用于与 Oracle 兼容的 DROP USER 语句。如果用户包含任何对象，则拒绝删除该用户。这是默认行为。必须在 Oracle 数据库中使用。

## 兼容性

DROP USER语句是一个LightDB扩展。SQL标准把用户的定义留给具体实现自行解释。

## 另见

DROP ROLE

---

# DROP USER MAPPING

DROP USER MAPPING — 移除一个用于外部服务器的用户映射

## Synopsis

```
DROP USER MAPPING [ IF EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC }  
SERVER server_name
```

## 说明

DROP USER MAPPING从外部服务器移除一个已有的用户映射。

一个外部服务器的所有者可以为该服务器的任何用户删除用户映射。如果该服务器上的USAGE特权被授予了一个用户，它也能删除用于它们自己的用户名的用户映射。

## 参数

IF EXISTS

如果该用户映射不存在则不要抛出一个错误，这种情况下会发出一个提示。

user\_name

该映射的用户名。CURRENT\_USER和USER匹配当前用户的名称。PUBLIC被用来匹配系统中所有现存和未来的用户名。

server\_name

用户映射的服务器名。

## 示例

删除一个用于服务器foo的用户映射bob（如果它存在）：

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

## 兼容性

DROP USER MAPPING符合 ISO/IEC 9075-9 (SQL/MED)。IF EXISTS子句是一个LightDB扩展。

## 另见

CREATE USER MAPPING, ALTER USER MAPPING

---

# DROP VIEW

DROP VIEW — 移除一个视图

## Synopsis

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## 描述

DROP VIEW删除一个现有的视图。要执行 这个命令你必须是该视图的拥有者。

## 参数

IF EXISTS

如果该视图不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的视图的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该视图的对象（例如其他视图），然后删除所有 依赖于那些对象的对象（见Section 5.14）。

RESTRICT

如果有任何对象依赖于该视图，则拒绝删除它。这是默认值。

## 示例

这个命令将移除名为kinds的视图：

```
DROP VIEW kinds;
```

## 兼容性

这个命令符合 SQL 标准，不过该标准只允许在每个命令中删除一个视图 并且没有IF EXISTS选项。该选项是一个 LightDB扩展。

## 另见

ALTER VIEW, CREATE VIEW



---

# END

END — 提交当前事务

## Synopsis

```
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

## 描述

END提交当前事务。所有该事务做的更改使得对他人可见并且被保证发生崩溃时仍然是持久的。这个命令是一种LightDB扩展，它等效于COMMIT。

## 参数

WORK  
TRANSACTION

可选关键词，它们没有效果。

AND CHAIN

如果规定了AND CHAIN，则立即启动与刚完成事务具有相同事务特征(参见 SET TRANSACTION)的新事务。否则，没有新事务被启动。

## 注解

使用ROLLBACK可以中止一个事务。

当不在一个事务中时发出END没有危害，但是会产生一个警告消息。

## 示例

要提交当前事务并且让所有更改持久化：

```
END;
```

## 兼容性

END是一种LightDB扩展，它提供和COMMIT等效的功能，后者在 SQL 标准中指定。

## 另见

BEGIN, COMMIT, ROLLBACK

---

# EXECUTE

EXECUTE — 执行一个预备语句

## Synopsis

```
EXECUTE name [ ( parameter [, ...] ) ]
```

## 描述

EXECUTE被用来执行一个之前准备好的语句。 由于预备语句只在会话期间存在，该预备语句必须在当前会话中由一个更早 执行的PREPARE语句所创建。

如果创建预备语句的PREPARE语句指定了一些参数， 必须向EXECUTE语句传递一组兼容的参数，否则会 发生错误。注意（与函数不同）预备语句无法基于其参数的类型或者数量重载。 在一个数据库会话中，预备语句的名称必须唯一。

更多创建和使用预备语句的信息请见PREPARE。

## 参数

name

要执行的预备语句的名称。

parameter

给预备语句的参数的实际值。这必须是一个能得到与该参数数据类型（在预备语句创建时决定）兼容的值的表达式。

## 输出

EXECUTE返回的命令标签是预备语句的命令标签而不是 EXECUTE。

## 例子

在PREPARE文档的Examples小节给出了例子。

## 兼容性

SQL 标准包括了一个EXECUTE语句， 但是只被用于嵌入式 SQL。这个版本的 EXECUTE语句也用了一种有点不同的语法。

## 另见

DEALLOCATE, PREPARE

---

# EXPLAIN

EXPLAIN — 显示一个语句的执行计划

## Synopsis

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

这里 option可以是：

```
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
SETTINGS [ boolean ]
BUFFERS [ boolean ]
WAL [ boolean ]
TIMING [ boolean ]
SUMMARY [ boolean ]
FORMAT { TEXT | XML | JSON | YAML }
```

## 描述

这个命令显示LightDB计划器为提供的语句所生成的执行计划。该执行计划会显示将怎样扫描语句中引用的表 — 普通的顺序扫描、索引扫描等等 — 以及在引用多个表时使用何种连接算法来把来自每个输入表的行连接在一起。

显示中最重要的部分是估计出的语句执行代价，它是计划器对于该语句要运行多久的猜测（以任意的代价单位度量，但是习惯上表示取磁盘页面的次数）。事实上会显示两个数字：在第一行能被返回前的启动代价，以及返回所有行的总代价。对于大部分查询来说总代价是最重要的，但是在一些情景中（如EXISTS中的子查询），计划器将选择更小的启动代价来代替最小的总代价（因为执行器将在得到一行后停止）。此外，如果你用一个LIMIT子句限制返回行的数量，计划器会在终端代价之间做出适当的插值来估计到底哪个计划是真正代价最低的。

ANALYZE选项导致该语句被实际执行，而不仅仅是被计划。那么实际的运行时间统计会被显示出来，包括在每个计划结点上花费的总时间（以毫秒计）以及它实际返回的行数。这对观察计划器的估计是否与实际相近很有用。

### Important

记住当使用了ANALYZE选项时语句会被实际执行。尽管EXPLAIN将丢弃SELECT所返回的任何输出，照例该语句的其他副作用还是会发生。如果你希望在INSERT、UPDATE、DELETE、CREATE TABLE AS或者EXECUTE上使用EXPLAIN ANALYZE而不希望它们影响你的数据，可以使用下面的方法：

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

只有ANALYZE和VERBOSE选项能被指定，并且必须按照上述的顺序，不要把选项列表放在圆括号内。在LightDB 9.0 之前，只支持没有圆括号的语法。我们期望所有新的选项将只在圆括号语法中支持。

## 参数

### ANALYZE

执行命令并且显示实际的运行时间和其他统计信息。这个参数默认被设置为FALSE。

### VERBOSE

显示关于计划的额外信息。特别是：计划树中每个结点的输出列表、模式限定的表和函数名、总是把表达式中的变量标上它们的范围表别名，以及总是打印统计信息被显示的每个触发器的名称。如果已经被计算，查询标识符也会被显示，详情请参见compute\_query\_id。这个参数默认被设置为FALSE。

### COSTS

包括每一个计划结点的估计启动和总代价，以及估计的行数和每行的宽度。这个参数默认被设置为TRUE。

### SETTINGS

包括有关配置参数的信息。具体来说，包括影响查询计划的选项，其值与内置默认值不同。此参数默认为FALSE。

### BUFFERS

包括缓冲区使用的信息。特别是：共享块命中、读取、标记为脏和写入的次数、本地块命中、读取、标记为脏和写入的次数、临时块读取和写入的次数以及track\_io\_timing启用时读取和写入数据文件块所用的时间（毫秒）。一次命中表示避免了一次读取，因为需要的块已经在缓存中找到了。共享块包含着来自于常规表和索引的数据；本地块包含着来自于临时表和索引的数据；而临时块包含着在排序、哈希、物化计划结点和类似情况中使用的短期工作数据。脏块的数量表示被这个查询改变的之前未被修改块的数量，而写入块的数量表示这个后台在查询处理期间从缓存中替换出去的脏块的数量。为一个较高层结点显示的块数包括它的所有子结点所用到的块数。在文本格式中，只会打印非零值。它默认为FALSE。

### WAL

包括有关WAL记录生成的信息。具体来说，包括记录数、整页图像数（fpi）和生成的WAL字节数量。在文本格式中，仅打印非零值。此参数只能在同时启用ANALYZE时使用。它默认为FALSE。

### TIMING

在输出中包括实际启动时间以及在每个结点中花掉的时间。反复读取系统时钟的负荷在某些系统上会显著地拖慢查询，因此在只需要实际的行计数而不是实际时间时，把这个参数设置为FALSE可能会有用。即使用这个选项关闭结点层的计时，整个语句的运行时间也总是会被度量。只有当ANALYZE也被启用时，这个参数才能使用。它的默认被设置为TRUE。

### SUMMARY

在查询计划之后包含摘要信息（例如，总计的时间信息）。当使用ANALYZE时默认包含摘要信息，但默认情况下不包含摘要信息，但可以使用此选项启用摘要信息。使用EXPLAIN EXECUTE中的计划时间包括从缓存中获取计划所需的时间以及重新计划所需的时间（如有必要）。

### FORMAT

指定输出格式，可以是TEXT、XML、JSON或者YAML。非文本输出包含和文本输出格式相同的信息，但是更容易被程序解析。这个参数默认被设置为TEXT。

boolean

指定被选中的选项是否应该被打开或关闭。可以写TRUE、ON或1来启用选项，写FALSE、OFF或0禁用它。boolean值也能被忽略，在这种情况下会假定值为TRUE。

statement

你想查看其执行计划的任何SELECT、INSERT、UPDATE、DELETE、VALUES、EXECUTE、DECLARE、CREATE TABLE AS或者CREATE MATERIALIZED VIEW AS语句。

## 输出

这个命令的结果是为statement选中的计划的文本描述，可能还标注了执行统计信息。Section 14. 描述了所提供的信息。

## 注解

为了允许LightDB查询计划器在优化查询时能做出合理的知情决策，查询中用到的所有表的pg\_statistic数据应该能保持为最新。通常这个工作会由autovacuum daemon负责自动完成。但是如果一个表最近在内容上有大量的改变，我们可能需要做一次手动的ANALYZE而不是等待 autovacuum 捕捉这些改变。

为了度量执行计划中每个节点的运行时成本，当前的EXPLAIN ANALYZE的当前实现为查询执行增加了性能分析开销。这样，在一个查询上运行EXPLAIN ANALYZE有时候比正常执行该查询要慢很多。开销的量取决于该查询的性质，以及使用的平台。最坏的情况会发生在那些自身执行时间很短的结点上，以及在那些具有相对较慢的有关时间的操作系统调用的机器上。

## 例子

有一个具有单个integer列和 10000 行的表，要显示在其上的一个简单查询的计划：

```
EXPLAIN SELECT * FROM foo;
```

QUERY PLAN

```
-----
Seq Scan on foo (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

这里有同样一个查询的 JSON 输出格式：

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
```

QUERY PLAN

```
-----
[
  {
    "Plan": {
      "Node Type": "Seq Scan",
      "Relation Name": "foo",
      "Alias": "foo",
      "Startup Cost": 0.00,
      "Total Cost": 155.00,
      "Plan Rows": 10000,
      "Plan Width": 4
    }
  }
]
```

```
]
(1 row)
```

如果有一个索引，并且我们使用了一个带有可索引WHERE条件的查询，EXPLAIN可能会显示一个不同的计划：

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
```

```
QUERY PLAN
```

```
-----
Index Scan using fi on foo (cost=0.00..5.98 rows=1 width=4)
  Index Cond: (i = 4)
(2 rows)
```

这里是同一查询的 YAML 格式：

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
QUERY PLAN
```

```
-----
- Plan: +
  Node Type: "Index Scan" +
  Scan Direction: "Forward"+
  Index Name: "fi" +
  Relation Name: "foo" +
  Alias: "foo" +
  Startup Cost: 0.00 +
  Total Cost: 5.98 +
  Plan Rows: 1 +
  Plan Width: 4 +
  Index Cond: "(i = 4)"
(1 row)
```

XML 格式我们留给读者做练习。

这里是去掉了代价估计的同样一个计划：

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;
```

```
QUERY PLAN
```

```
-----
Index Scan using fi on foo
  Index Cond: (i = 4)
(2 rows)
```

这里是一个使用聚集函数的查询的查询计划例子：

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;
```

```
QUERY PLAN
```

```
-----
Aggregate (cost=23.93..23.93 rows=1 width=4)
-> Index Scan using fi on foo (cost=0.00..23.92 rows=6 width=4)
  Index Cond: (i < 10)
(3 rows)
```

这里是一个使用EXPLAIN EXECUTE显示预备查询的执行计划的例子：

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
  WHERE id > $1 AND id < $2
  GROUP BY foo;
```

```
EXPLAIN ANALYZE EXECUTE query(100, 200);
```

QUERY PLAN

```
-----
HashAggregate (cost=9.54..9.54 rows=1 width=8) (actual time=0.156..0.161
rows=11 loops=1)
  Group Key: foo
    -> Index Scan using test_pkey on test (cost=0.29..9.29 rows=50 width=8)
(actual time=0.039..0.091 rows=99 loops=1)
      Index Cond: ((id > $1) AND (id < $2))
Planning time: 0.197 ms
Execution time: 0.225 ms
(6 rows)
```

当然，这里显示的有关数字取决于表涉及到的实际内容。还要注意这些数字甚至选中的查询策略，可能在LightDB的不同版本之间变化，因为计划器可能被改进。此外，ANALYZE命令使用随机采样来估计数据统计。因此，在一次新的ANALYZE运行之后，代价估计可能会改变，即便是表中数据的实际分布没有改变也是如此。

## 兼容性

在 SQL 标准中没有定义EXPLAIN语句。

## 参见

ANALYZE

---

# FETCH

FETCH — 使用游标从查询中检索行

## Synopsis

```
FETCH [ direction [ FROM | IN ] ] cursor_name
```

其中 direction 可以为空或者以下之一：

```
NEXT
PRIOR
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
BACKWARD
BACKWARD count
BACKWARD ALL
```

## 描述

FETCH从之前创建的一个游标中检索行。

游标具有一个相关联的位置，FETCH会用到该位置。游标位置可能会位于查询结果的第一行之前、结果中任意行之上或者结果的最后一行之后。在被创建时，游标被定位在第一行之前。在取出一些行后，该游标被定位在最近被取出的行上。如果FETCH运行超过了可用行的末尾，则该游标会被定位在最后一行之后（如果向后取，则是第一行之前）。FETCH ALL或者FETCH BACKWARD ALL将总是让游标被定位于最后一行之后或者第一行之前。

NEXT、PRIOR、FIRST、LAST、ABSOLUTE、RELATIVE 形式会在适当移动游标后取出一行。如果没有这样一行，将返回一个空结果，并且视情况将游标定位在第一行之前或者最后一行之后。

使用FORWARD和BACKWARD的形式会在向前移动或者向后移动的方向上检索指定数量的行，然后将游标定位在 最后返回的行上（如果count超过可用的行数，则定位 在所有行之后或者之前）。

RELATIVE 0、FORWARD 0以及 BACKWARD 0都会请求检索当前行但不移动游标，也就是重新取最近被取出的行。只要游标没有被定位在第一行之前或者最后一行 之后，这种操作都会成功，否则不会返回任何行。

### Note

这个页面描述在 SQL 命令层面对游标的使用。如果想要在 PL/pgSQL函数中使用游标，规则会有所不同 — 请见Section 41.7.3



## 参数

direction

direction 定义获取方向以及要取得的行数。它可以是下列之一：

NEXT

取出下一行。如果省略direction，这将是默认值。

PRIOR

取出当前位置之前的一行。

FIRST

取出该查询的第一行（和ABSOLUTE 1相同）。

LAST

取出该查询的最后一行（和ABSOLUTE -1相同）。

ABSOLUTE count

取出该查询的第count个行，如果count为负则是从尾部开始取出 第abs(count)个行。如果 count超出范围，将定位在第一行 之前或者最后一行之后。特别地，ABSOLUTE 0 会定位在第一行之前。

RELATIVE count

取出第count个后继行，如果 count为负 则是取出前面的第abs(count)个行。RELATIVE 0重新取出当前行（如果有）。

count

取出接下来的count行（和 FORWARD count相同）。

ALL

取出所有剩余的行（和FORWARD ALL相同）。

FORWARD

取出下一行（和NEXT相同）。

FORWARD count

取出接下来的count行。 FORWARD 0重新取出当前行。

FORWARD ALL

取出所有剩下的行。

BACKWARD

取出当前行前面的一行（和PRIOR相同）。

BACKWARD count

取出前面的count行（反向扫描）。 BACKWARD 0会重新取出当前行。

BACKWARD ALL

取出所有当前位置之前的行（反向扫描）。

count

count 是一个可能带有符号的整数常量，它决定要取得的位置或者行数。对于 FORWARD 和 BACKWARD 情况，指定一个负的 count 等效于改变 FORWARD 和 BACKWARD 的意义。

cursor\_name

一个已打开游标的名称。

## 输出

如果成功完成，FETCH 命令返回一个下面形式的命令标签：

```
FETCH count
```

count 是取得的行数（可能为零）。注意在 ltsql 中，命令标签将不会实际显示，因为 ltsql 会显示被取得的行。

## 注解

如果想要使用 FETCH 的任意变体而不使用带有正计数的 FETCH NEXT 或者 FETCH FORWARD，应该用 SCROLL 声明游标。对于简单查询，LightDB 将允许从不带 SCROLL 的游标中反向取得行，但最好不要依赖这种行为。如果游标被声明为带有 SCROLL，则不允许反向取得。

用 ABSOLUTE 取行并不比用相对移动快多少：不管则样，底层实现都必须遍历所有的中间行。用绝对值获取的情况更糟：必须读到查询尾部来找到最后一行，并且接着从那里反向开始遍历。不过，回卷到查询的开始（正如 FETCH ABSOLUTE 0）是很快的。

DECLARE 被用来定义游标。使用 MOVE 可改变游标位置而不检索数据。

## 示例

下面的例子用一个游标遍历一个表：

```
BEGIN WORK;
```

```
-- 建立一个游标：
```

```
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;
```

```
-- 在游标 liahona 中取出前 5 行：
```

```
FETCH FORWARD 5 FROM liahona;
```

code	title	did	date_prod	kind	len
BL101	The Third Man	101	1949-12-23	Drama	01:44
BL102	The African Queen	101	1951-08-11	Romantic	01:43
JL201	Une Femme est une Femme	102	1961-03-12	Romantic	01:25
P_301	Vertigo	103	1958-11-14	Action	02:08
P_302	Becket	103	1964-02-03	Drama	02:28

```
-- 取出前面一行：
```

```
FETCH PRIOR FROM liahona;
```

code	title	did	date_prod	kind	len
P_301	Vertigo	103	1958-11-14	Action	02:08

```
-- 关闭游标并且结束事务：  
CLOSE liahona;  
COMMIT WORK;
```

## 兼容性

SQL 标准只定义FETCH在嵌入式 SQL 中使用。这里描述的FETCH变体返回数据时就好像数据是一个SELECT结果，而不是被放在主变量中。除这一点之外，FETCH完全向上兼容于 SQL 标准。

涉及FORWARD和BACKWARD的 FETCH形式，以及形式FETCH count和FETCH ALL（其中FORWARD是隐式的）都是 LightDB扩展。

SQL 标准只允许FROM在游标名之前。使用 IN的选项或者完全省去它们是一种扩展。

添加GUC参数lightdb\_cursor\_after\_commit，当参数值为 on，事务提交后，可以继续操作该游标。当参数值为 off，事务提交后，不可以继续操作该游标。这个参数的默认值为off。

## 另见

CLOSE, DECLARE, MOVE

---

# GRANT

GRANT — 定义访问特权

## Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
        [, ...] | ALL [ PRIVILEGES ] }
      ON { [ TABLE ] table_name [, ...]
          | ALL TABLES IN SCHEMA schema_name [, ...] }
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
        [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
      ON [ TABLE ] table_name [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { USAGE | SELECT | UPDATE }
        [, ...] | ALL [ PRIVILEGES ] }
      ON { SEQUENCE sequence_name [, ...]
          | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
      ON DATABASE database_name [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
      ON DOMAIN domain_name [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
      ON FOREIGN DATA WRAPPER fdw_name [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
      ON FOREIGN SERVER server_name [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
      ON { { FUNCTION | PROCEDURE | ROUTINE } routine_name [ ( [ [ argmode ]
        [ arg_name ] arg_type [, ...] ] ) ] [, ...]
          | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name
        [, ...] }
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
      ON LANGUAGE lang_name [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
      ON LARGE OBJECT loid [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
      ON SCHEMA schema_name [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { CREATE | ALL [ PRIVILEGES ] }
      ON TABLESPACE tablespace_name [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
      ON TYPE type_name [, ...]
      TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT role_name [, ...] TO role_specification [, ...]
      [ WITH ADMIN OPTION ]
      [ GRANTED BY role_specification ]
```

```
GRANT option_for_oracle TO role_specification [, ...]
      [ WITH ADMIN OPTION ]
```

where role\_specification can be:

```
[ GROUP ] role_name
| PUBLIC
| CURRENT_USER
| SESSION_USER
```

其中 option\_for\_oracle 可以是:

```
CONNECT
| RESOURCE
| UNLIMITED TABLESPACE
```

## 描述

GRANT命令由两种基本的变体：一种授予在一个数据库对象（表、列、视图、外部表、序列、数据库、外部数据包装器、外部服务器、函数、过程、过程语言、模式或表空间）上的特权，另一个授予一个角色中的成员关系。这些变体在很多方面都相似，但是也有很多不同，所以还是得分别描述它们。

## 在数据库对象上 GRANT

这种GRANT命令的变体将一个数据库对象上的指定特权交给一个或多个角色。如果有一些已经被授予，这些特权会被加入到它们之中。

关键词PUBLIC指示特权要被授予给所有角色，包括那些可能稍后会被创建的角色。PUBLIC可以被认为是一个被隐式定义的总是包含所有角色的组。任何特定角色都将具有直接授予给它的特权、授予给它作为成员所在的任何角色的特权以及被授予给PUBLIC的特权。

如果指定了WITH GRANT OPTION，特权的接收者可以接着把它授予给其他人。如果没有授权选项，接收者就不能这样做。授权选项不能被授予给PUBLIC。

没有必要把权限授予给一个对象的拥有者（通常就是创建该对象的用户），因为拥有者默认具有所有的特权（不过拥有者可能为了安全选择撤回一些 它们自己的特权）。

删除一个对象或者以任何方式修改其定义的权力是不被当作一个可授予特权的，它被固化在拥有者中，并且不能被授予和撤回（不过，相似地效果可以通过授予或者撤回在拥有该对象的角色中的成员关系来实现，见下文）。拥有者也隐式地拥有该对象的所有授权选项。

The possible privileges are:

SELECT  
 INSERT  
 UPDATE  
 DELETE  
 TRUNCATE  
 REFERENCES  
 TRIGGER  
 CREATE  
 CONNECT  
 TEMPORARY  
 EXECUTE  
 USAGE

特定类型的权限，如 Section 5. 中所定义。

TEMP

TEMPORARY的替代拼写。

ALL PRIVILEGES

授予对象的类型可用的所有权限。PRIVILEGES关键词在LightDB中是可选的，尽管它是严格的 SQL 所需要的。

FUNCTION语法适用于简单函数、聚合函数和窗口函数，但不适用于过程；对过程使用PROCEDURE。 或者，使用ROUTINE来引用函数、聚合函数、窗口函数或过程而不管其精确类型。

还有一个选项，可以在一个或多个模式中对所有相同类型的对象授予特权。此功能当前仅支持表、序列、函数和过程。 ALL TABLES也会影响视图和外表，就像特定对象 GRANT命令。 ALL FUNCTIONS也会影响聚合和窗口函数，但不影响过程，就像特定对象GRANT命令一样。 使用 ALL ROUTINES 来包括过程。

## 角色上的 GRANT

GRANT命令的这种变体把一个角色中的成员关系授予一个或者多个其他角色。一个角色中的成员关系是有意义的，因为它会把授予给一个角色的特权带给该角色的每一个成员。

如果指定了WITH ADMIN OPTION，成员接着可以把该角色中的成员关系授予给其他用户，也可以撤回该角色中的成员关系。如果没有管理选项，普通用户就不能做这些工作。一个角色不被认为持有自身的WITH ADMIN OPTION，但是它可以从一个会话用户匹配该角色的数据库会话中授予或撤回自身中的成员关系。数据库超级用户能够授予或撤回任何角色中任何人的成员关系。具有CREATEROLE特权的角色能够授予或者撤回任何非超级用户角色中的成员关系。

如果指定了GRANTED BY，则该授予被记录为已由指定角色完成。 只有数据库超级用户可以使用此选项，除非它为执行命令指定了相同的角色。

和特权的情况不同，一个角色中的成员关系不能被授予PUBLIC。还要注意 这种形式的命令不允许role\_specification 中的噪声词GROUP。

## 注解

REVOKE命令被用来撤回访问特权。

从LightDB 8.1 开始，用户和组的概念已经被统一到一种单一类型的实体（被称为一个角色）。因此不再需要使用关键词GROUP来标识一个被授权者是一个用户或者一个组。在该命令中仍然允许GROUP，但是它只是一个噪音词而已。

如果一个用户持有特定列或者其所在的整个表的特权， 该用户可以在该列上执行SELECT、INSERT等命令。在表层面上授予特权 然后对一列撤回该特权将不会按照你希望的运作： 表级别的授权不会受到列级别操作的影响。

当一个对象的非拥有者尝试GRANT该对象上的特权，如果该用户在该对象上什么特权都不拥有，该命令将立刻失败。只要有一些特权可用，该命令将继续，但是它将只授予那些用户具有授权选项的特权。如果不持有授权选项，GRANT ALL PRIVILEGES形式将发出一个警告消息。而如果不持有命令中特别提到的任何特权的授权选项，其他形式将会发出一个警告（原则上这些语句也适用于对象拥有者，但是由于拥有者总是被视为持有所有授权选项，因此这种情况不会发生）。

需要注意的是，数据库超级用户可以访问所有对象而不管对象特权的设置。这可与 Unix 系统中的root权力相提并论。对于root来说，除非绝对必要，使用一个超级用户来操作是不明智的。

如果一个超级用户选择发出一个GRANT或者REVOKE命令，该命令将被执行，好像它是由受影响对象的拥有者发出的一样。特别地，通过这样一个命令授予的特权将好像是由对象拥有者授予的一样（对于角色成员关系，该成员关系好像是由该角色本身授予的一样）。

GRANT以及REVOKE也可以由一个不是受影响对象拥有者的角色完成，不过该角色是拥有该对象的角色中的一个成员，或者是在该对象上持有特权的WITH GRANT OPTION的角色中的一个成员。在这种情况下，特权将被记录为由实际拥有该对象的角色授予或者是由持有特权的WITH GRANT OPTION的角色授予。例如，如果表t1被角色g1拥有，u1是它的一个成员，那么u1可以把t1上的特权授予给u2，但是那些特权将好像是直接由g1授予的。角色g1的任何其他成员可以稍后撤回它们。

如果执行GRANT的角色间接地通过多于一条角色成员关系路径持有所需的特权，将不会指定哪一个包含它的角色将被记录为完成了该授权。在这样的情况中，最好使用SET ROLE来成为你想用其做GRANT的特定角色。

授予一个表上的权限不会自动地扩展权限给该表使用的任何序列，包括绑定在SERIAL列上的序列。序列上的权限必须被独立设置。

有关特定的特权类型以及如何检查对象特权的更多信息，请参见Section 5.7

关于与 Oracle 兼容的授权语句 option\_for\_oracle。目前，仅提供语法支持，不提供功能支持。必须在 Oracle 数据库中使用。

## 例子

把表films上的插入特权授予给所有用户：

```
GRANT INSERT ON films TO PUBLIC;
```

把视图kinds上的所有可用特权授予给用户manuel：

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

注意虽然上述语句被一个超级用户或者kinds的拥有者执行时确实会授予所有特权，但是当由其他人执行时将只会授予那些执行者拥有授权选项的权限。

把角色admins中的成员关系授予给用户joe：

```
GRANT admins TO joe;
```

Oracle 兼容的授权语句：

```
GRANT CONNECT TO HS_SES;  
GRANT RESOURCE TO HS_SES;  
GRANT UNLIMITED TABLESPACE TO HS_SES;
```

## 兼容性

根据 SQL 标准，ALL PRIVILEGES中的PRIVILEGES关键词是必须的。SQL 标准不支持在每个命令中设置超过一个对象上的特权。

LightDB允许一个对象所有者 撤回它们拥有的普通特权：例如，一个表所有者可以通过撤回其自身拥有 的INSERT、UPDATE、DELETE 和TRUNCATE特权让该表对它们自己只读。根据 SQL 标准 这是不可能发生的。原因在于LightDB 认为拥有者的特权是由拥有者授予给它们自己的，因此它们也能够撤回它们。 在 SQL 标准中，拥有者的特权是有一个假设的实体 “\_SYSTEM” 所授予。 由于不是 “\_SYSTEM”， 拥有者就不能撤回这些权力。

根据 SQL 标准，授权选项可以被授予给PUBLIC， LightDB 只支持把授权选项授予给角色。

SQL标准允许GRANTED BY选项仅指定CURRENT\_USER或CURRENT\_ROLE。 另外的变体是LightDB扩展。

SQL 标准提供了其他对象类型上的USAGE特权：字符集、排序规则、翻译。

在 SQL 标准中，序列只有一个USAGE特权，它控制NEXT VALUE FOR表达式的使用，该表达式等效于 LightDB 中的函数nextval。序列的特权SELECT和UPDATE是 LightDB 扩展。应用序列的USAGE特权到currval函数也是一个 LightDB 扩展（该函数本身也是）。

数据库、表空间、模式和语言上的特权都是LightDB扩展。

## 参见

REVOKE, ALTER DEFAULT PRIVILEGES



---

# IMPORT FOREIGN SCHEMA

IMPORT FOREIGN SCHEMA — 从一个外部服务器导入表定义

## Synopsis

```
IMPORT FOREIGN SCHEMA remote_schema
  [ { LIMIT TO | EXCEPT } ( table_name [, ...] ) ]
FROM SERVER server_name
INTO local_schema
  [ OPTIONS ( option 'value' [, ...] ) ]
```

## 简介

IMPORT FOREIGN SCHEMA创建表示存在于 外部服务器上的表的外部表。新外部表将由发出该命令的用户所拥有并且用 匹配远程表的正确的列定义和选项创建。

默认情况下，存在于外部服务器上一个特定模式中的所有表和视图都会被导入。 根据需要，表的列表可以被限制到一个指定的子集，或者可以排除特定的表。 新外部表都被创建在一个必须已经存在的目标模式中。

要使用IMPORT FOREIGN SCHEMA，用户必须具有外部服务器上的USAGE特权以及在目标模式上的 CREATE特权。

## 参数

remote\_schema

要从哪个远程模式导入。一个远程模式的特定含义依赖于所使用的外部数据包装器。

LIMIT TO ( table\_name [, ...] )

只导入匹配给定表名之一的外部表。外部模式中其他的表将被忽略。

EXCEPT ( table\_name [, ...] )

把指定的外部表排除在导入之外。除了列在这里的表之外，外部模式 中存在的所有表都将被导入。

server\_name

要从哪个外部服务器导入。

local\_schema

被导入的外部表将创建在其中的模式。

OPTIONS ( option 'value' [, ...] )

要在导入期间使用的选项。允许使用的选项名称和值与每一个外部数据包装器 有关。

## 示例

从服务器film\_server上的远程模式foreign\_films 中导入表定义，把外部表创建在本地模式films中：

```
IMPORT FOREIGN SCHEMA foreign_films  
  FROM SERVER film_server INTO films;
```

同上，但是只导入两个表actors和 directors（如果存在）：

```
IMPORT FOREIGN SCHEMA foreign_films LIMIT TO (actors, directors)  
  FROM SERVER film_server INTO films;
```

## 兼容性

IMPORT FOREIGN SCHEMA命令符合 SQL标准，不过OPTIONS 子句是一种LightDB扩展。

## 另见

CREATE FOREIGN TABLE, CREATE SERVER

---

# INSERT

INSERT — 在一个表中创建新行

## Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT [IGNORE] INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...]
  | query }
    [ ON CONFLICT [ conflict_target ] conflict_action | ON DUPLICATE KEY UPDATE
  { column_name = { expression | DEFAULT } } [, ...] ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

其中 `conflict_target` 可以是以下之一：

```
( { index_column_name | ( index_expression ) } [ COLLATE collation ]
  [ opclass ] [, ...] ) [ WHERE index_predicate ]
ON CONSTRAINT constraint_name
```

并且 `conflict_action` 是以下之一：

```
DO NOTHING
DO UPDATE SET { column_name = { expression | DEFAULT } |
               ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT }
  [, ...] ) |
               ( column_name [, ...] ) = ( sub-SELECT )
               } [, ...]
  [ WHERE condition ]
```

```
INSERT ALL insert_into_clause [values_clause] query
| INSERT ALL conditional_insert_clause query
```

where `insert_into_clause` can be list of:

```
INTO table_name [ ( column_name [, ...] ) ]
```

and `values_clause` is:

```
VALUES(, ...)
```

and `conditional_insert_clause` is one of:

```
WHEN condition THEN insert_into_clause [values_clause]
ELSE insert_into_clause [values_clause]
```

## 描述

INSERT将新行插入到一个表中。我们可以 插入一个或者更多由值表达式指定的行，或者插入来自一个查询的零行 或者更多行。

目标列的名称可以以任意顺序列出。如果没有给出列名列表，则有两种确定 目标列的可能性。第一种是以被声明的顺序列出该表的所有列。另一种可能 性是，如果VALUES 子句或者query只提供N个列，则以被声明的顺序列出该表的前 N列。VALUES 子句或者 query提供的值会被从左至右关联到这些显式或者隐式 给出的目标列。

每一个没有出现在显式或者隐式列列表中的列都将被默认填充，如果为该列 声明过默认值则用默认值填充，否则用空值填充。

如果任意列的表达式不是正确的数据类型，将会尝试自动类型转换。

如果表缺少唯一索引，则并发操作不会阻塞对表的 INSERT 操作。对于具有唯一索引的表，如果并发会话执行锁定或修改与插入的唯一索引值匹配的行的操作，则可能会被阻塞； 详细信息请参见 Section 58.5 可以使用 ON CONFLICT 指定替代操作以避免唯一约束或排除约束违规错误。（请参见下文的 ON CONFLICT Clause。）

可选的RETURNING子句让INSERT根据 实际被插入（如果使用了ON CONFLICT DO UPDATE子句， 可能是被更新）的每一行来计算和返回值。这主要用来获取由默认值提供的值，例如一个序列号。不过，允许在其中包括使用该表列的任何表达式。 RETURNING列表的语法与SELECT的输出 列表的相同。只有被成功地插入或者更新的行才将被返回。例如，如果一行被锁定但由于不满足ON CONFLICT DO UPDATE ... WHERE clause condition没有被更新，该行将 不被返回。

为了向表中插入，你必须具有其上的INSERT特权。 如果存在ON CONFLICT DO UPDATE子句，还要求该表上的UPDATE特权。

如果一个列列表被指定，你只需要其中的列上的INSERT 特权。类似地，在指定了ON CONFLICT DO UPDATE时，你只需要被列出要更新的列上的UPDATE特权。不过， ON CONFLICT DO UPDATE还要求其值被 ON CONFLICT DO UPDATE表达式或者 condition使用的列上的SELECT特权。

使用RETURNING子句需要RETURNING中提到的所有列的 SELECT权限。 如果使returning子句从查询中插入行， 则当然需要对查询中使用的任何表或列具有SELECT权限。

要对多表进行插入， 你可以使用 INSERT ALL 语句。 在一个多表插入中，将把根据子查询返回的行派生出的计算行插入到一个或多个表中。

## 参数

### 插入

这个小节介绍了在只插入新行时可以使用的参数。 专门用于ON CONFLICT子句的 参数会单独介绍。

with\_query

WITH子句允许指定一个或者更多子查询，在 INSERT查询中可以用名称引用这些子查询。 详见 Section 7.8以及SELECT。

query（SELECT语句）也可以包含一个 WITH子句。在这种情况下 query中可以引用 两组with\_query，但是第二个优先级更高（因为它被嵌套更近）。

ignore

关键字 ignore 为兼容 MySQL 语法添加。使用 ignore 时，在当前的实现中，违反唯一性约束的插入行将产生一个警告，而不是错误。若要使用关键字 ignore，必须设置为MySQL兼容模式。

table\_name

一个已有表的名称（可以被模式限定）。

alias

table\_name 的替补名称。当提供了一个别名时，它会完全隐藏掉表的实际名称。 当ON CONFLICT DO UPDATE的目标是一个被排除的表时这特别有用，因为那将被当作表示要被插入行的特殊表的名称。

column\_name

名为table\_name的表中的一个列的名称。如有必要，列名可以用一个子域名或者数组下标限定（指向一个组合列的某些列中插入会让其他域为空）。当用 ON CONFLICT DO UPDATE引用一列时，不要在一个目标列的说明中包括表名。例如，INSERT INTO table\_name ... ON CONFLICT DO UPDATE SET table\_name.col = 1是非法的（这遵循UPDATE的一般行为）。

OVERRIDING SYSTEM VALUE

如果指定了此子句，那么为标识列提供的任何值都将覆盖默认的序列生成的值。

对于定义为GENERATED ALWAYS的标识列，插入显式值（DEFAULT除外）而不指定OVERRIDING SYSTEM VALUE或OVERRIDING USER VALUE是错误的。（对于定义为GENERATED BY DEFAULT的标识列，OVERRIDING SYSTEM VALUE是正常行为，并指定其不执行任何操作，但是LightDB允许它作为扩展名。）

OVERRIDING USER VALUE

如果指定了此子句，则将忽略为标识列提供的任何值，并应用默认的序列生成的值。

例如，当在表之间拷贝值时，这个子句有能派上用场。INSERT INTO tb12 OVERRIDING USER VALUE SELECT \* FROM tb11将从tb11中拷贝所有在tb12中不是标识列的列，而tb12中标识列的值将由与tb12关联的序列产生。

DEFAULT VALUES

所有列都将被其默认值填充，就像为每个列显式指定了DEFAULT。（例如这种形式下不允许OVERRIDING子句）。

expression

要赋予给相应列的表达式或者值。

DEFAULT

相应的列将填充其默认值。标识列将由关联序列生成的新值填充。对于生成的列，允许指定该值，但仅指定根据其生成表达式计算该列的正常行为。

query

提供要被插入行的查询（SELECT语句）。其语法描述请参考SELECT语句。

output\_expression

在每一行被插入或更新后由INSERT命令计算并且返回的表达式。该表达式可以使用table\_name指定的表中的任何列。写成\*可返回被插入或更新行的所有列。

output\_name

要用于被返回列的名称。

conditional\_insert\_clause

通过指定 conditional\_insert\_clause 来进行带条件的多表插入。LightDB 通过相应的 WHEN 条件对每个 insert\_into\_clause 进行过滤，该条件决定是否执行对应的 insert\_into\_clause。

## ON CONFLICT 子句

可选的ON CONFLICT子句为出现唯一性违背或排除 约束违背错误时提供另一种可供选择的动作。对于每一个要插入的行， 不管是插入进行下去还是由conflict\_target 指定的一个仲裁者约束或者索引被违背，都会 采取可供选择的conflict\_action。 ON CONFLICT DO NOTHING简单地把避免插入行。 ON CONFLICT DO UPDATE则会 更新与要插入的行冲突的已有行。

conflict\_target可以执行 唯一索引推断。在执行推断时，它由一个或者多个index\_column\_name 列或者 index\_expression 表达式以及一个可选的 index\_predicate构成。所有刚好包含 conflict\_target指定的列/表达式的table\_name唯一索引（不管顺序）都会被推断为（选择为）仲裁者索引。如果指定了 index\_predicate，它 必须满足仲裁者索引（也是推断过程的一个进一步的要求）。注意这意味着如果 有一个满足其他条件的非部分唯一索引（没有谓词的唯一索引）可用，它将被 推断为仲裁者（并且会被ON CONFLICT使用）。如果推断 尝试不成功，则会发生一个错误。

ON CONFLICT DO UPDATE保证一个原子的 INSERT或者 UPDATE结果。在没有无关错误的前提下，这两种 结果之一可以得到保证，即使在很高的并发度也能保证。这也可以被称作UPSERT — “UPDATE 或 INSERT”。

conflict\_target

通过选择仲裁者索引来指定哪些行与 ON CONFLICT在其上采取可替代动作的行相冲突。要么执行唯一索引推断，要么显式命名一个 约束。对于ON CONFLICT DO NOTHING来说， 它对于指定一个conflict\_target是可选的。 在被省略时，与所有有效约束（以及唯一索引）的冲突都会被处理。对于 ON CONFLICT DO UPDATE，必须 提供一个conflict\_target。

conflict\_action

conflict\_action指定一个可替换的 ON CONFLICT动作。它可以是 DO NOTHING，也可以是一个指定在冲突情况下 要被执行的UPDATE动作细节的DO UPDATE子句。ON CONFLICT DO UPDATE中的SET和 WHERE子句能够使用该表的名称（或者别名） 访问现有的行，并且可以用特殊的被排除 表访问要插入的行。这个动作要求被排除 列所在目标表的任何列上的SELECT特权。

注意所有行级BEFORE INSERT触发器的效果都会 反映在被排除值中，因为那些效果可能会 让该行避免被插入。

index\_column\_name

一个table\_name列 的名称。它被用来推断仲裁者索引。它遵循CREATE INDEX格式。这要求 index\_column\_name 上的SELECT特权。

index\_expression

和index\_column\_name类似，但是 被用来推断出现在索引定义中的table\_name列（非简单列）上的 表达式。遵循CREATE INDEX格式。这要求 任何出现在index\_expression中的列上的 SELECT特权。

collation

指定时，强制相应的index\_column\_name或 index\_expression 使用一种特定的排序规则以便在推断期间能被匹配上。通常 会被省略，因为排序规则通常不会影响约束违背的发生。遵循 CREATE INDEX格式。

opclass

指定时，强制相应的index\_column\_name或 index\_expression 使用特定的操作符类以便在推断期间能被匹配上。通常会被省略， 因为相等语义在一种类型的操作符类 之间都是等价的，或者因为足以信任已定义的唯一索引具有适当的 相等定义。遵循CREATE INDEX格式。

index\_predicate

用于允许推断部分唯一索引。任何满足该谓词（不一定需要真的是 部分索引）的索引都能被推断。遵循CREATE INDEX格式。这要求任何出现在index\_predicate中的列上的SELECT特权。

constraint\_name

用名称显式地指定一个仲裁者约束， 而不是推断一个约束或者索引。

condition

一个能返回boolean值的表达式。只有让这个表达式返回 true的行才将被更新，不过在采用 ON CONFLICT DO UPDATE动作时所有的行都会被锁定。 注意condition会被最后计算，即一个冲突 被标识为要更新的候选对象之后。

注意不支持把排除约束作为ON CONFLICT DO UPDATE的 仲裁者。在所有的情况中，只支持NOT DEFERRABLE约束和 唯一索引作为仲裁者。

带有ON CONFLICT DO UPDATE子句的 INSERT是一种“确定性的” 语句。这表明不允许该命令影响任何单个现有行超过一次，如果发生则会 发生一个基数违背错误。要插入的行不应该在仲裁者索引或约束所限制的 属性上相重复。

注意，当前不支持用分区表上的INSERT的ON CONFLICT DO UPDATE子句更新冲突行的分区键，因为那样会让行移动到新的分区中。

### Tip

使用唯一索引推断通常比使用ON CONFLICT ON CONSTRAINT constraint\_name直接提名一个约束更好。当底层索引被以重叠方式替换成另一个或多或少等效的索引时，推断将继续正确地工作，例如在删除要被替换的索引之前使用CREATE UNIQUE INDEX ... CONCURRENTLY。

## ON DUPLICATE KEY UPDATE 子句

如果您指定了一个 ON DUPLICATE KEY UPDATE 子句，并且要插入的行会导致主键中出现重复值， 则会更新旧的行。例如，如果列 a 被声明为 PRIMARY KEY 并包含值 1，则以下两个语句具有类似的效果：

```
INSERT INTO t1 (a, b, c) VALUES (1, 2, 3) ON DUPLICATE KEY UPDATE c=t1.c+1;
```

```
UPDATE t1 SET c=c+1 WHERE a=1;
```

在 ON DUPLICATE KEY UPDATE 子句中的赋值表达式中， 您可以使用 VALUES(col\_name)函数来引用 INSERT ... ON DUPLICATE KEY UPDATE 语句的 INSERT 部分中的列值， 而不是使用 excluded 表。换句话说，在 ON DUPLICATE KEY UPDATE 子句中，VALUES(col\_name)指的是如果没有重复键冲突将要插入的 col\_name 的值。 此函数在多行插入中特别有用。VALUES() 函数仅受支持在 ON DUPLICATE KEY UPDATE 子句或 INSERT 语句中。

## 输出

成功完成时，INSERT命令会返回以下形式的命令标签：

```
INSERT oid count
```

count是被插入或更新的行数。 oid总是0（过去，如果count恰好为1，并且目标表被声明为WITH OIDS，则它是分配给插入行的OID，否则为0，但现在已不再支持创建WITH OIDS表）。

如果INSERT命令包含RETURNING子句，其结果会类似于包含RETURNING列表中定义的列和值的SELECT语句，这些结果是由该命令在被插入或更新行上计算得到。

## 注解

如果指定的表是一个分区表，每一行都会被路由到合适的分区并且插入其中。如果指定的表是一个分区，如果输入行之一违背该分区的约束则将发生错误。

## 示例

向films中插入一行：

```
INSERT INTO films VALUES
('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

在这个例子中，len列被省略并且因此会具有默认值：

```
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

这个例子为日期列使用DEFAULT子句而不是指定一个值：

```
INSERT INTO films VALUES
('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

插入一个完全由默认值构成的行：

```
INSERT INTO films DEFAULT VALUES;
```

用多行VALUES语法插入多个行：

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

这个例子从表tmp\_films中获得一些行插入到表 films中，两个表具有相同的列布局：

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

这个例子插入数组列：

```
-- 为 noughts-and-crosses 游戏创建一个空的 3x3 棋盘
INSERT INTO tictactoe (game, board[1:3][1:3])
VALUES (1, '{{" ", " ", " "}, {" ", " ", " "}, {" ", " ", " "}}');
```



-- 实际上可以不用上面例子中的下标

```
INSERT INTO tictactoe (game, board)
VALUES (2, '{X," ", " "},{ " ",0," "},{ " ",X," "}');
```

向表distributors中插入一行，返回由 DEFAULT子句生成的序号：

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
RETURNING did;
```

增加为 Acme Corporation 管理账户的销售人员的销量，并且把整个被更新的行以及当前时间记录到一个日志表中：

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
  RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

酌情插入或者更新新的 distributor。假设已经定义了一个唯一索引来约束 出现在did列中的值。注意，特殊的 excluded表被用来引用原来要插入的值：

```
INSERT INTO distributors (did, dname)
VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname;
```

插入一个 distributor，或者在一个被排除的行（具有一个匹配约束的列或者 会让行级前（或者后）插入触发器引发的列的行）存在时不处理要插入的行。 例子假设已经定义了一个唯一触发器来约束出现在did列 中的值：

```
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
ON CONFLICT (did) DO NOTHING;
```

酌情插入或者更新新的 distributor。例子假设已经定义了一个唯一触发器来 约束出现在did列中的值。WHERE子句被用 来限制实际被更新的行（不过，任何没有被更新的已有行仍将被锁定）：

```
-- 根据一个特定的 ZIP 编码更新 distributors
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
ON CONFLICT (did) DO UPDATE
SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
WHERE d.zipcode <> '21201';
```

-- 直接在语句中命名一个约束（使用相关的索引来判断是否做  
-- DO NOTHING 动作）

```
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;
```

如果可能就插入新的 distributor，否则DO NOTHING。 例子假设已经定义了一个唯一索引，它约束让is\_active 布尔列为true的行子集上did列中的值：

```
-- 这个语句可能推断出一个在 "did" 上带有谓词 "WHERE is_active"
-- 的部分唯一索引，但是它可能也只是使用了 "did" 上的一个常规唯一约束
INSERT INTO distributors (did, dname) VALUES (10, 'Conrad International')
```

---

```
ON CONFLICT (did) WHERE is_active DO NOTHING;
```

考虑以下 CREATE TABLE 语句创建的表，然后运行 ltsql 客户端中显示的语句，结果如下所示：

```
CREATE TABLE test (
  id INT NOT NULL,
  name CHAR(16),
  age INT DEFAULT 0,
  PRIMARY KEY(id)
);
```

```
lightdb@testdb=# INSERT INTO test VALUES (1, 'Old name', '18');
INSERT 0 1
```

```
lightdb@testdb=# INSERT INTO test VALUES (2, 'Old name', '19') ON DUPLICATE KEY
UPDATE name = 'New name';
INSERT 0 1
```

```
lightdb@testdb=# SELECT * FROM test;
 id |      name      | age
----+-----+----
  1 | Old name      | 18
  2 | Old name      | 19
(2 rows)
```

```
lightdb@testdb=# INSERT INTO test VALUES (2, 'Old name', '20') ON DUPLICATE KEY
UPDATE name = 'New name', age = test.age + 5;
INSERT 0 1
```

```
lightdb@testdb=# SELECT * FROM test;
 id |      name      | age
----+-----+----
  1 | Old name      | 18
  2 | New name      | 24
(2 rows)
```

```
lightdb@testdb=# INSERT INTO test VALUES (2, 'Old name', '20') ON DUPLICATE KEY
UPDATE name = 'New name', age = excluded.age;
INSERT 0 1
```

```
lightdb@testdb=# SELECT * FROM test;
 id |      name      | age
----+-----+----
  1 | Old name      | 18
  2 | New name      | 20
(2 rows)
```

```
lightdb@testdb=# INSERT INTO test VALUES (2, 'Old name', '20') ON DUPLICATE KEY
UPDATE name = VALUES(name), age = VALUES(age);
INSERT 0 1
```

```
lightdb@testdb=# SELECT * FROM test;
 id |      name      | age
----+-----+----
  1 | Old name      | 18
  2 | Old name      | 20
```

---

(2 rows)

```
lightdb@testdb=# INSERT INTO test VALUES (2, 'New name', '29') ON DUPLICATE KEY
UPDATE name = name, age = VALUES(age);
INSERT 0 1
```

```
lightdb@testdb=# SELECT * FROM test;
```

id	name	age
1	Old name	18
2	Old name	29

(2 rows)

可以通过使用 `insert all` 语句去对多个表进行插入，同时也支持通过条件来进行插入，案例如下：

```
CREATE TABLE T1(a int , b text);
CREATE TABLE T1_1(a int , b text);
CREATE TABLE T1_2(a int , b text);

-- 多表插入
INSERT ALL INTO T1_1 VALUES(1,1) INTO T1_2 VALUES(2,2) SELECT * FROM DUAL;
INSERT ALL INTO T1_1 VALUES(a,b) INTO T1_2 VALUES(a%100, b) SELECT * FROM T1;
-- 带条件的多表插入
INSERT ALL WHEN a > 2 THEN INTO T1_1 ELSE INTO T1_2 SELECT * FROM T1;
```

MySQL兼容模式下使用Insert Ignore:

```
create database mydb with lightdb_syntax_compatible_type 'mysql';
\c mydb;
create table t( a int primary key, b int);
INSERT IGNORE INTO t(a,b) values(1,1), (2,2), (1,1);
```

## 兼容性

INSERT符合 SQL 标准，不过 RETURNING子句是一种 LightDB扩展，在 INSERT中使用WITH也是，用ON CONFLICT指定一个替代动作也是扩展。还有，标准不允许省略列名列表但不通过VALUES子句或者query填充 所有列的情况。

SQL标准指定只有存在一个总是会生成值的标识列时才能指定OVERRIDING SYSTEM VALUE。而LightDB在任何情况下都允许这个子句，并且在不适用时会忽略它。

ON DUPLICATE KEY UPDATE 子句是 LightDB 对 SQL 标准的扩展。它仅在数据库的语法兼容类型为 mysql 时受支持。有关详细信息，请参见 `lightdb_syntax_compatible_type`。

INSERT ALL 语句 是 LightDB 对标准 SQL 的一个扩展。它只在 ORACLE 兼容模式下的数据库中使用。有关兼容模式的详细信息，请参见 `lightdb_syntax_compatible_type`。

query子句可能的限制在 SELECT有介绍。

---

# LISTEN

LISTEN — 监听一个通知

## Synopsis

LISTEN channel

## 描述

LISTEN在名为channel的通知频道上将当前会话注册为一个监听者。如果当前会话已经被注册为这个通知频道的一个监听者，则什么也不会发生。

只要命令NOTIFY channel被调用（不管是在这个会话还是在另一个连接到同一数据库的会话中），所有当前正在该通知频道上监听的会话都会被通知，并且每一个会话将会接着通知连接到它的客户端应用。

可以使用UNLISTEN命令在一个给定通知频道上反注册一个会话。当会话结束时，它的监听注册会被自动清除。

一个客户端应用检测通知事件的必用方法取决于它使用的LightDB应用编程接口。如果使用libpq库，应用会将LISTEN作为一个普通 SQL 命令发出，并且接着必须周期性地调用函数PQnotifies来查看是否接收到通知事件。其他诸如libpqctl的接口提供了更高层次上的处理通知事件的方法。事实上，通过使用libpqctl应用程序员甚至不必直接发出LISTEN或UNLISTEN。更多细节可参阅所使用的接口的文档。

## 参数

channel

一个通知频道的名称（任意标识符）。

## 注解

LISTEN在事务提交时生效。如果在一个后来被回滚的事务中执行了LISTEN或UNLISTEN，被监听的通知频道集合不会变化。

一个已经执行了LISTEN的事务不能为两阶段提交做准备。

第一次设置侦听会话时有一个竞争条件：如果并发提交的事务正在发送通知事件，那么新的侦听会话将接收哪些事件？答案是，会话将接收在事务提交步骤中的一瞬间之后所有提交的事件。但这比事务在查询中可能观察到的任何数据库状态都要晚一些。这将导致使用LISTEN的以下规则：首先执行（并提交！）该命令，然后在一个新事务中根据应用程序逻辑的需要检查数据库状态，然后依靠通知来了解数据库状态的后续更改。最初收到的几个通知可能涉及在初始数据库检查中已经观察到的更新，但这通常是无害的。

NOTIFY对LISTEN和NOTIFY的使用进行了更广泛的讨论。

## 例子

从ltsql中配置并执行一个监听/通知序列

```
LISTEN virtual;
```

```
NOTIFY virtual;  
Asynchronous notification "virtual" received from server process with PID 8448.
```

## 兼容性

在 SQL 标准中没有LISTEN语句。

## 参见

NOTIFY, UNLISTEN

---

# LOAD

LOAD — 载入一个共享库文件

## Synopsis

```
LOAD 'filename'
```

## 描述

这个命令把一个共享库文件载入到LightDB服务器的地址空间中。如果该文件已经被载入，这个命令什么也不会做。只要调用包含 C 函数的共享库文件中的一个函数，这些共享库文件就会被自动载入。因此，一次显式的LOAD通常只在载入一个通过“钩子”修改服务器行为而不是提供一组函数的库时需要。

库文件名通常只是一个裸文件名，在服务器的库搜索路径（由 `dynamic_library_path` 设置）中寻找。或者，它可以作为完整的路径名称给出。无论哪种情况，平台的标准共享库文件扩展名都可以省略。有关于此话题的更多信息可见Section 36.10.1

非超级用户只能把LOAD应用在位于 `$libdir/plugins/` 中的库文件 — 指定的 `filename` 必须正好以该字符串开始（确保在那里只安装了“安全的”库是数据库管理员的责任）。

## 兼容性

LOAD是一种 LightDB扩展。

## 另见

CREATE FUNCTION

---

# LOCK

LOCK — 锁定一个表

## Synopsis

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]
```

其中 lockmode 可以是以下之一：

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE  
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

## 描述

LOCK TABLE获得一个表级锁，必要时会等待任何冲突锁被释放。如果指定了NOWAIT，LOCK TABLE不会等待以获得想要的锁：如果它不能立刻得到，该命令会被中止并且发出一个错误。一旦获取到，该锁会被在当前事务中一直持有（没有UNLOCK TABLE命令，锁总是在事务结束时被释放）。

当一个视图被锁定时，出现在该视图定义查询中的所有关系也将被使用同样的锁模式递归地锁住。

在为引用表的命令自动获取锁时，LightDB总是尽可能使用最不严格的锁模式。提供LOCK TABLE是用于想要更严格的锁定的情况。例如，假设一个应用运行一个READ COMMITTED隔离级别的事务，并且需要确保一个表中的数据在该事务的期间保持稳定。要实现这个目的，必须在查询之前在表上获得SHARE锁模式。这将阻止并发的数据更改并且确保该表的后续读操作会看到已提交数据的一个稳定视图，因为SHARE锁模式与写入者所要求的ROW EXCLUSIVE锁有冲突，并且你的LOCK TABLE name IN SHARE MODE语句将等待，直到任何并发持有ROW EXCLUSIVE模式锁的持有者提交或者回滚。因此，一旦得到锁，就不会有未提交的写入还没有解决。更进一步，在释放该锁之前，任何人都不能开始。

要在运行在REPEATABLE READ或SERIALIZABLE隔离级别的事务中得到类似的效果，你必须在执行任何SELECT或者数据修改语句之前执行LOCK TABLE语句。一个REPEATABLE READ或者SERIALIZABLE事务的数据视图将在它的第一个SELECT或者数据修改语句开始时被冻结。在该事务中稍后的一个LOCK TABLE仍将阻止并发写 — 但它不会确保该事务读到的东西对应于最新的已提交值。

如果一个此类事务正要修改表中的数据，那么它应该使用SHARE ROW EXCLUSIVE锁模式来取代SHARE模式。这会保证一次只有一个此类事务运行。如果不用这种模式，死锁就可能出现：两个事务可能都要求SHARE模式，并且都不能获得ROW EXCLUSIVE模式来真正地执行它们的更新（注意一个事务所拥有的锁不会冲突，因此一个事务可以在它持有SHARE模式时获得ROW EXCLUSIVE模式 — 但是如果有其他人持有SHARE模式时则不能）。为了避免死锁，确保所有的事务在同样的对象上以相同的顺序获得锁，并且如果在一个对象上涉及多种锁模式，事务应该总是首先获得最严格的那种模式。

更多关于锁模式和锁策略的信息可见Section 13.3

## 参数

name

要锁定的一个现有表的名称（可以是模式限定的）。如果在表名前指定了ONLY，只有该表会被锁定。如果没有指定了ONLY，该表和它所有的后代表（如果有）都会被锁定。可选地，在表名后指定\*来显式地表示把后代表包括在内。

命令LOCK TABLE a, b;等效于 LOCK TABLE a; LOCK TABLE b;。这些表会被按照在 LOCK TABLE中指定的顺序一个一个 被锁定。

#### lockmode

锁模式指定这个锁和哪些锁冲突。锁模式在Section 13.8中描述。

如果没有指定锁模式，那儿将使用最严格的模式ACCESS EXCLUSIVE。

#### NOWAIT

指定LOCK TABLE不等待任何冲突锁被释放： 如果所指定的锁不能立即获得，那么事务就会中止。

## 注解

LOCK TABLE ... IN ACCESS SHARE MODE要求目标表上的SELECT特权。LOCK TABLE ... IN ROW EXCLUSIVE MODE要求目标表上的INSERT、UPDATE、DELETE或TRUNCATE特权。所有其他形式的LOCK要求表级UPDATE、DELETE或TRUNCATE特权。

在该视图上执行锁定的用户必须具有该视图上相应的特权。此外视图的拥有者必须拥有底层基关系上的相关特权，但是执行锁定的用户不需要底层基关系上的任何权限。

LOCK TABLE在一个事务块外部没有用处：锁将只保持到语句完成。因此如果在一个事务块外部使用了LOCK，LightDB会报告一个错误。使用BEGIN和COMMIT（或者ROLLBACK）定义一个事务块。

LOCK TABLE只处理表级锁，因此涉及到 ROW的模式名称在这里都是不当的。这些模式名称应该通常 被解读为用户在被锁定表中获取行级锁的意向。还有， ROW EXCLUSIVE模式是一个可共享的表锁。记住就 LOCK TABLE而言，所有的锁模式都具有相同的语义， 只有模式的冲突规则有所不同。关于如何获取一个真正的行级锁的信息， 请见SELECT文档中的Section 13.3.4和The Locking Clause。

## 示例

在将要向一个外键表中执行插入时在主键表上获得一个 SHARE锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
  WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- 如果记录没有被返回就做 ROLLBACK
INSERT INTO films_user_comments VALUES
  (_id, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

在将要执行一次删除操作前在主键表上取一个 SHARE ROW EXCLUSIVE锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
  (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

## 兼容性

在 SQL 标准中没有LOCK TABLE，SQL 标准中使用 SET TRANSACTION指定事务上的并发层次。LightDB也支持这样做，详见 SET TRANSACTION。



除ACCESS SHARE、ACCESS EXCLUSIVE和 SHARE UPDATE EXCLUSIVE锁模式之外， LightDB锁模式和 LOCK TABLE语法与 Oracle中的兼容。

---

# MOVE

MOVE — 定位一个游标

## Synopsis

```
MOVE [ direction [ FROM | IN ] ] cursor_name
```

其中direction可以为空或者以下之一：

```
NEXT  
PRIOR  
FIRST  
LAST  
ABSOLUTE count  
RELATIVE count  
count  
ALL  
FORWARD  
FORWARD count  
FORWARD ALL  
BACKWARD  
BACKWARD count  
BACKWARD ALL
```

## 描述

MOVE重新定位一个游标而不检索任何数据。MOVE的工作完全像FETCH命令，但是它只定位游标并且不返回行。

用于MOVE命令的参数和FETCH命令的一样，可参考FETCH。

## 输出

成功完成时，MOVE命令返回的命令标签形式是

```
MOVE count
```

count是一个具有同样参数的FETCH命令会返回的行数（可能为零）。

## 示例

```
BEGIN WORK;  
DECLARE liahona CURSOR FOR SELECT * FROM films;  
  
-- 跳过前 5 行：  
MOVE FORWARD 5 IN liahona;  
MOVE 5  
  
-- 从游标 liahona 中取第 6 行：  
FETCH 1 FROM liahona;  
code | title | did | date_prod | kind | len
```

```
-----+-----+-----+-----+-----+-----  
P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37  
(1 row)
```

```
-- 关闭游标 liahona 并且结束事务:  
CLOSE liahona;  
COMMIT WORK;
```

## 兼容性

在 SQL 标准中没有MOVE语句。

## 另见

CLOSE, DECLARE, FETCH

---

# MERGE

MERGE — 基于源数据插入、更新表中的行

## Synopsis

```
MERGE INTO target_table_name [ [ AS ] target_alias ]
USING data_source
ON join_condition
when_clause [...]
```

其中 data\_source 是

```
{ source_table_name |
( source_query )
}
[ [ AS ] source_alias ]
```

而 when\_clause 是

```
{ WHEN MATCHED THEN { merge_update } |
WHEN NOT MATCHED THEN { merge_insert }
}
```

而 merge\_insert 是

```
INSERT [( column_name [, ...] ) ]
{ VALUES ( { expression | DEFAULT } [, ...] ) | DEFAULT VALUES
}
[ WHERE condition ]
```

而 merge\_update 是

```
UPDATE SET { column_name = {
expression | DEFAULT } |
( column_name [, ...] ) = ( {
expression | DEFAULT } [, ...] )
} [, ...]
[ WHERE condition ]
```

## 描述

MERGE 命令通过使用 data\_source，对 target\_table\_name 中的行进行修改。MERGE 提供了一个单一的 SQL 语句，可以有条件地 INSERT 或 UPDATE 行，这是一个通常需要多个过程式语言语句才能完成的任务。

首先，MERGE 命令从 data\_source 到 target\_table\_name 执行联接，产生零个或多个候选更改行。对于每个候选更改行，MATCHED 或 NOT MATCHED 的状态仅设置一次，然后按指定顺序评估 WHEN 子句。如果其中一个被激活，则执行指定的操作。对于任何候选更改行，最多只能激活一个 WHEN 子句。

MERGE 命令的操作与同名的常规 UPDATE、INSERT 命令具有相同的效果。这些命令的语法不同，特别是没有指定表名。所有操作都参考 target\_table\_name。

没有 MERGE 权限。如果指定了更新操作，则必须在 SET 子句中引用 `target_table_name` 的列的 UPDATE 权限，如果指定了插入操作，则必须在 `target_table_name` 上拥有 INSERT 权限。权限在语句开始时进行测试，无论是否激活特定的 WHEN 子句，在随后的执行过程中都会进行检查。在引用 `condition` 中的 `data_source` 和 `target_table_name` 的任何列时，都需要具有 SELECT 权限。

如果 `target_table_name` 上定义了 RULES，则不支持 MERGE。

## 注意

必须确保目标表中的一条记录不能与源表中的记录多次匹配，因为这可能导致目标表中更新的记录没有特定值。

## 参数

`target_table_name`

要合并到的目标表的名称（可选带模式限定符）。

`target_alias`

目标表的替代名称。提供别名时，它完全隐藏了表的实际名称。例如，给定 `MERGE foo AS f`，MERGE 语句的其余部分必须将该表称为 `f` 而不是 `foo`。

`source_table_name`

源表、视图或转换表的名称（可选带模式限定符）。

`source_query`

一个查询（SELECT 语句或 VALUES 语句），提供要合并到 `target_table_name` 中的行。请参考 SELECT 语句或 VALUES 语句，了解其语法的说明。

`source_alias`

数据源的替代名称。提供别名时，它完全隐藏了指定了表还是查询。

`join_condition`

`join_condition` 是一个表达式，其结果为 boolean 类型的值（类似于 WHERE 子句），指定哪些 `data_source` 中的行与 `target_table_name` 中的行匹配。

### Warning

只有尝试匹配 `data_source` 行的 `target_table_name` 列应出现在 `join_condition` 中。仅引用 `target_table_name` 列的 `join_condition` 子表达式只能影响采取哪些操作，通常是以令人惊讶的方式。

`when_clause`

至少需要一个 WHEN 子句。

如果 WHEN 子句指定了 WHEN MATCHED，并且候选更改行与 `target_table_name` 中的一行匹配，如果 `condition` 不存在或存在且评估为 true，则会激活 WHEN 子句。如果 WHEN 子句指定了 WHEN NOT MATCHED，并且候选更改行与 `target_table_name` 中的一行不匹配，如果 `condition` 不存在或存在且评估为 true，则会激活 WHEN 子句。

### merge\_insert

指定一个 INSERT 操作，将一行插入目标表中。目标列名可以以任何顺序列出。如果根本没有给出列名列表，则默认为表中所有列按其声明的顺序。

未在显式或隐式列列表中出现的每个列将填充其声明的默认值或 null（如果没有默认值）。

如果任何列的表达式不是正确的数据类型，则会尝试进行自动类型转换。

如果 target\_table\_name 是分区表，则每行将路由到适当的分区并插入其中。如果 target\_table\_name 是分区，则如果其中一个输入行违反分区约束，则会出现错误。

列名不能指定多次。INSERT 操作不能包含子查询。

只能指定一个 VALUES 子句。VALUES 子句只能引用来自源关系的列，因为根据定义，没有匹配的目标行。

### merge\_update

指定一个 UPDATE 操作，更新 target\_table\_name 的当前行。列名不能指定多次。

不要包含表名，因为这与 UPDATE 命令通常不同。例如，UPDATE tab SET col = 1 是无效的。

### column\_name

target\_table\_name 中的列名。如果需要，可以使用子字段名或数组下标限定列名。（仅插入复合列的某些字段将使其他字段为空。）在引用列时，不要在目标列的规范中包含表名。

### expression

赋给该列的表达式。该表达式可以使用该表中此列和其他列的旧值。

### condition

返回类型为 boolean 的表达式。如果此表达式返回 true，则将激活 WHEN 子句，并对该行执行相应的操作。该表达式可能不包含可能对数据库执行写操作的函数。

对于 WHEN MATCHED 子句的条件可以引用源关系和目标关系中的列。对于 WHEN NOT MATCHED 子句的条件只能引用来自源关系的列，因为根据定义，没有匹配的目标行。

## 输出

成功完成后，MERGE 命令将返回一个命令标记，格式如下：

```
MERGE total-count
```

total-count 是更改的总行数（无论是插入、更新还是删除）。如果 total-count 为 0，则没有任何行发生任何更改。

## 示例

基于新事务对 CustomerAccounts 进行维护。

```
MERGE INTO CustomerAccount CA  
USING RecentTransactions T
```

```
ON T.CustomerId = CA.CustomerId
WHEN MATCHED THEN
UPDATE SET Balance = Balance + TransactionValue
WHEN NOT MATCHED THEN
INSERT (CustomerId, Balance)
VALUES (T.CustomerId, T.TransactionValue);
```

注意，由于执行过程中 MATCHED 结果不会改变，因此这与以下语句完全等效：

```
MERGE INTO CustomerAccount CA
USING (Select CustomerId, TransactionValue From
RecentTransactions) AS T
ON CA.CustomerId = T.CustomerId
WHEN NOT MATCHED THEN
INSERT (CustomerId, Balance)
VALUES (T.CustomerId, T.TransactionValue)
WHEN MATCHED THEN
UPDATE SET Balance = Balance + TransactionValue;
```

尝试插入一个新的库存项以及库存数量。如果该项已经存在，则更新现有项的库存计数。不允许具有零库存的条目。

```
MERGE INTO wines w
USING wine_stock_changes s
ON s.winename = w.winename
WHEN NOT MATCHED THEN
INSERT VALUES(s.winename, s.stock_delta) WHERE s.stock_delta > 0
WHEN MATCHED THEN
UPDATE SET stock = w.stock + s.stock_delta WHERE w.stock +
s.stock_delta > 0;
```

wine\_stock\_changes 表可以是最近加载到数据库中的临时表，例如。

---

# NOTIFY

NOTIFY — 生成一个通知

## Synopsis

```
NOTIFY channel [ , payload ]
```

## 描述

NOTIFY命令发送一个通知事件以及一个可选的“载荷”字符串给每个正在监听的客户端应用，这些应用之前都在当前数据库中为指定的频道名执行过LISTEN channel。通知对所有用户都可见。

NOTIFY为访问同一个LightDB数据库的进程集合提供了一种简单的进程间通讯机制。伴随着通知可以发送一个载荷字符串，通过使用数据库中的表从通知者向监听者传递额外的数据，也可以构建用于传输结构化数据的高层机制。

由一个通知事件传递给客户端的信息包括通知频道名称、发出通知的会话的服务器进程PID以及载荷字符串，如果载荷字符串没有被指定则它为空字符串。

将在一个给定数据库以及其他数据库中使用的频道名称由数据库设计者定义。通常，频道名称与数据库中某个表的名称相同，并且通知事件就意味着：“我改变了这个表，来看看改了什么吧”。但是NOTIFY和LISTEN命令并未强制这样的关联。例如，一个数据库设计者可以使用几个不同的频道名称来标志一个表上的不同种类的改变。另外，载荷字符串可以被用于多种不同的情况。

当NOTIFY被用来标志对一个特定表的改变时，一种有用的编程技巧是把NOTIFY 放在一个由表更新触发的语句触发器中。在这种方式中，每当表被改变时 都将自动发生通知，并且应用程序员不可能忘记发出通知。

NOTIFY以一些重要的方式与 SQL 事务互动。首先，如果一个NOTIFY在一个事务内执行，在事务被提交之前，该通知事件都不会被递送。这是合适的，因为如果该事务被中止，所有其中的命令都不会产生效果，包括NOTIFY在内。但如果期望通知事件被立即递送，那这种行为就会令人不安。其次，如果一个监听会话收到了一个通知信号而它正在一个事务中，在该事务完成（提交或者中止）之前，该通知事件将不会被递送给它连接的客户端。同样，原因在于如果一个通知在一个事务内被递送且该事务后来被中止，我们会希望该通知能以某种方式被撤销 — 但是服务器一旦把通知发送给客户端就无法“收回它”。因此通知事件只能在事务之间递送。其中的要点是把NOTIFY用作实时信号的应用应该让它们事务尽可能短小。

如果在同一事务中使用相同的有效负载字符串多次向同一通道名称发送信号，则只向侦听器传递通知事件的一个实例。另一方面，带有不同载荷字符串的通知将总是作为不同的通知被递送。类似地，来自不同事务的通知将不会被折叠成一个通知。除了丢弃后来生成的重复通知实例之外，NOTIFY保证来自同一个事务的通知按照它们被发送的顺序被递送。还可以保证的是，来自不同事务的消息会按照其事务被提交的顺序递送。

一个执行NOTIFY的客户端自己也同时在监听同一个通知频道是很常见的事。在这种情况下，和所有其他监听会话一样，它会取回一个通知事件。根据应用的逻辑，这可能导致无用的工作，例如从自己刚刚写入的一个表中读出相同的更新。可以通过关注发出通知的服务器进程PID（在通知事件消息中提供）与自己的会话PID（可以从libpq得到）是否相同来避免这种额外的工作。当两者相同时，该通知事件就是当前会话自己发出的，所以可以忽略。

## 参数

channel

要对其发信号的通知频道的名称（任意标识符）。



payload

要通过通知进行沟通的“载荷”字符串。这必须是一个简单的字符串。在默认配置下，该字符串不能超过 8000 字节（如果需要发送二进制数据或者更多信息，最好是把它放在一个数据库表中并且发送该记录的键）。

## 注解

有一个队列保持着已经发送但是还没有被所有监听会话处理的通知。如果该队列被占满，调用NOTIFY的事务将在提交时失败。该队列非常大（标准安装中是 8GB）并且应该足以应付几乎每一种用例。不过，如果一个会话执行了NOTIFY并且接着长时间进入一个事务，不会发生清理操作。一旦该队列使用过半，你将在日志文件中看到警告，它指出哪个会话阻止了清理。在这种情况下，应该确保这个会话结束它的当前事务，这样清理才能够进行下去。

函数`pg_notification_queue_usage`返回队列中当前被待处理通知所占据的比例。详见Section 9.26

一个已经执行了NOTIFY的事务不能为两阶段提交做准备。

## pg\_notify

要发送一个通知，你也能使用函数`pg_notify(text, text)`。该函数采用频道名称作为第一个参数，而载荷则作为第二个参数。如果你需要使用非常量的频道名称和载荷，这个函数比NOTIFY命令更容易使用。

## 例子

从ltsql配置和执行一个监听/通知序列：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received
from server process with PID 8448.
```

```
LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server
process with PID 14728.
```

## 兼容性

在 SQL 标准中没有NOTIFY语句。

## 参见

LISTEN, UNLISTEN

---

# PREPARE

PREPARE — 预备一个语句用于执行

## Synopsis

```
PREPARE name [ ( data_type [, ...] ) ] AS statement  
PREPARE name FROM variable
```

## 描述

PREPARE创建一个预备语句。预备语句是一种服务器端对象，它可以被用来优化性能。当PREPARE语句被执行时，指定的语句会被解析、分析并且重写。当后续发出一个EXECUTE命令时，该预备语句会被规划并且执行。这种工作的划分避免了重复性的解析分析工作，同时允许执行计划依赖所提供的特定参数值。

预备语句可以接受参数：在执行时会被替换到语句中的值。在创建预备语句时，可以用位置引用参数，如\$1、\$2等。也可以选择性地指定参数数据类型的一个列表。当一个参数的数据类型没有被指定或者被声明为unknown时，其类型会从该参数第一次被引用的环境中推知（如果可能）。在执行该语句时，在EXECUTE语句中为这些参数指定实际值。更多有于此的信息可参考EXECUTE。

预备语句只在当前数据库会话期间存在。当会话结束时，预备语句会消失，因此在重新使用之前必须重新建立它。这也意味着一个预备语句不能被多个数据库客户端同时使用。不过，每一个客户端可以创建它们自己要使用的预备语句。预备语句可以用DEALLOCATE命令手工清除。

当一个会话要执行大量类似语句时，预备语句可能会有最大性能优势。如果该语句很复杂（难于规划或重写，例如查询涉及很多表的连接或者要求应用多条规则），性能差异将会特别明显。如果语句相对比较容易规划和重写，但是执行起来开销相对较大，那么预备语句的性能优势就不那么显著了。

## 参数

name

给这个特定预备语句的任意名称。它在一个会话中必须唯一并且后续将被用来执行或者清除一个之前准备好的语句。

data\_type

预备语句一个参数的数据类型。如果一个特定参数的数据类型没有被指定或者被指定为unknown，其类型将从该参数第一次被引用的环境中推得。要在预备语句本身中引用参数，可以使用 \$1、\$2等。

statement

任何SELECT、INSERT、UPDATE、DELETE或者VALUES语句。

variable

在mysql模式（参见lightdb\_dblevel\_syntax\_compatible\_type）中，variable是包含SQL语句文本的用户变量。任何ALTER [ TABLE | INDEX | SEQUENCE | VIEW | MATERIALIZED VIEW ]、SELECT、INSERT、UPDATE、DELETE或者VALUES语句。文本必须代表单个语句，而不是多个语句。

## 注解

可以使用generic plan（通用计划）或custom plan（自定义计划）执行预备语句。通用计划在所有执行中都是相同的，而自定义计划是为特定一次执行使用调用中给出的参数值所生成的。使用通用计划可以避免规划开销，但在某些情况下，自定义计划的执行效率要高得多，因为规划器可以利用参数值的知识（当然，如果准备好的语句没有参数，则这是没有意义的，并且始终应使用通用计划）。

默认情况下（也就是当 plan\_cache\_mode 设定为 auto 时），对已经准备好的带有参数的语句，服务器将自动选择使用通用或自定义计划。当前的规则是，前五次执行都是使用自定义计划完成的，并且计算这些计划的平均估计代价。之后就会创建通用计划，并将其估计代价与自定义计划的平均代价相比较。如果相对于重复规划来说，通用计划的代价不比自定义计划的平均代价高太多，那么后续执行将倾向于使用通用计划。

这种启发式方法可以被推翻，通过将 plan\_cache\_mode 分别设置为 force\_generic\_plan 或 force\_custom\_plan 可以强迫服务器使用通用或自定义计划。这个设置主要用于当通用计划的代价估计由于某种原因不太好的情况，这允许在通用计划的实际代价远高于自定义计划的实际代价时也能选中通用计划。

要检查 LightDB 为一个预备语句使用的查询计划，可以使用 EXPLAIN，例如：

```
EXPLAIN EXECUTE name(parameter_values);
```

如果使用的是一个通用计划，它将包含参数符号 \$n，而一个定制计划则会把提供的参数值替换进去。

更多关于查询规划以及 LightDB 为此所收集的统计信息的内容，请见 ANALYZE 文档。

尽管预备语句的主要目的是为了避免重复对语句进行解析分析以及规划，但是只要上一次使用该预备语句后该语句中用到的数据库对象发生了定义性（DDL）改变，LightDB 将会对该语句强制进行重新分析和重新规划。还有，如果 search\_path 的值发生变化，也将使用新的 search\_path 重新解析该语句（后一种行为是从 LightDB 9.3 开始加入的）。这些规则让预备语句的使用在语义上几乎等效于反复提交相同的查询文本，但是能在性能上获利（如果没有对象定义被改变，特别是如果最优计划保持不变时）。该语义等价性不完美的一个例子是：如果语句用一个未限定的名称引用某个表，并且之后在 search\_path 中更靠前的模式中创建了一个新的同名表，则不会发生自动的重解析，因为该语句使用的对象没有被改变。不过，如果某些其他更改造成了重解析，后续使用中都会引用新表。

可以通过查询 pg\_prepared\_statements 系统视图来看到会话中所有可用的预备语句。

## 例子

为一个 INSERT 语句创建一个预备语句，然后执行它：

```
PREPARE fooplan (int, text, bool, numeric) AS
  INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

为一个 SELECT 语句创建一个预备语句，然后执行它：

```
PREPARE usrrptplan (int) AS
  SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
  AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

在这个示例中，第二个参数的数据类型没有被指定，因此会从使用 \$2 的环境中推知。

在mysql模式（参见lightdb\_dblevel\_syntax\_compatible\_type）中，为一个INSERT语句创建一个预备语句，然后执行它：

```
SET @sql = 'INSERT INTO my_table values(a)';  
PREPARE stmt FROM @sql;  
EXECUTE stmt;  
DEALLOCATE PREPARE stmt;
```

在这个示例中，用户变量内容中没有需要绑定的参数。

## 兼容性

SQL标准包括一个PREPARE语句，但是它只用于嵌入式SQL。这个版本的PREPARE语句同时也使用了一种略有不同的语法。

## 另见

DEALLOCATE, EXECUTE

---

# PREPARE TRANSACTION

PREPARE TRANSACTION — 为两阶段提交准备当前事务

## Synopsis

```
PREPARE TRANSACTION transaction_id
```

## 描述

PREPARE TRANSACTION为两阶段提交准备 当前事务。在这个命令之后，该事务不再与当前会话关联。相反，它的状态 被完全存储在磁盘上，并且有很高的可能性它会被提交成功（即便在请求提交前发生数据库崩溃）。

一旦被准备好，事务稍后就可以分别用COMMIT PREPARED或者ROLLBACK PREPARED提交或者回滚。 可以从任何会话而不仅仅是执行原始事务的会话中发出这些命令。

从发出命令的会话的角度来看，PREPARE TRANSACTION不像ROLLBACK命令： 在执行它之后，就没有活跃的当前事务，并且该预备事务的效果也不再可见（ 如果该事务被提交，效果将重新变得可见）。

如果由于任何原因PREPARE TRANSACTION 命令失败，它会变成一个ROLLBACK：当前事务会被取消。

## 参数

transaction\_id

一个任意的标识符， COMMIT PREPARED或者ROLLBACK PREPARED 以后将用这个标识符来标识这个事务。该标识符必须写成一个字符串， 并且长度必须小于 200 字节。它也不能与任何当前已经准备好的事务的标识符相同。

## 注解

PREPARE TRANSACTION并不是设计为在应用或者交互式 会话中使用。它的目的是允许一个外部事务管理器在多个数据库或者其他事务性 来源之间执行原子的全局事务。除非你在编写一个事务管理器，否则你可能不会 用到PREPARE TRANSACTION。

这个命令必须在一个事务块中使用。事务块用BEGIN开始。

当前在已经执行过任何涉及到临时表或者会话的临时命名空间、创建带 WITH HOLD的游标或者执行 LISTEN、UNLISTEN或NOTIFY的 事务中，不允许PREPARE该事务。这些特性与当前会话绑定得太紧密，所以对一个要被准备的事务来说没有什么用处。

如果用SET（不带LOCAL选项）修改过事务的 任何运行时参数，这些效果会持续到 PREPARE TRANSACTION之后，并且将不会被后续的任何 COMMIT PREPARED或 ROLLBACK PREPARED所影响。因此，在这一 方面PREPARE TRANSACTION的行为更像 COMMIT而不是ROLLBACK。

所有当前可用的准备好事务被列在pg\_prepared\_xacts系统视图中。

### Caution

让一个事务处于准备好状态太久是不明智的。这将会干扰 VACUUM回收存储的能力，并且在极端情况下可能导致 数据库关闭以阻止事务 ID 回卷

（见Section 22.1.5。还要记住，该事务会继续持有它已经持有的锁。该特性的设计用法是，只要一个外部事务管理器已经验证其他数据库也准备好了要提交，一个准备好的事务将被正常地提交或者回滚。

如果没有建立一个外部事务管理器来跟踪准备好的事务并且确保它们被迅速地结束，最好禁用准备好事务特性（设置 `max_prepared_transactions` 为零）。这将防止意外地创建准备好事务，不然该事务有可能被忘记并且最终导致问题。

## 例子

为两阶段提交准备当前事务，使用 `foobar` 作为事务标识符：

```
PREPARE TRANSACTION 'foobar';
```

## 兼容性

`PREPARE TRANSACTION` 是一种 `LightDB` 扩展。其意图是用于外部事务管理系统，其中有些已经被标准涵盖（例如 `X/Open XA`），但是那些系统的 `SQL` 方面未被标准化。

## 另见

`COMMIT PREPARED`, `ROLLBACK PREPARED`

---

# REASSIGN OWNED

REASSIGN OWNED — 更改一个数据库角色拥有的数据库对象的拥有关系

## Synopsis

```
REASSIGN OWNED BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
                    TO { new_role | CURRENT_USER | SESSION_USER }
```

## 描述

REASSIGN OWNED指示系统把 old\_role们拥有的任何数据库对象的拥有关系更改为 new\_role。

## 参数

old\_role

一个角色的名称。这个角色在当前数据库中所拥有的所有对象以及所有共享对象（数据库、表空间）的所有权都将被重新赋予给 new\_role。

new\_role

将作为受影响对象的新拥有者的角色名称。

## 注解

REASSIGN OWNED经常被用来为移除一个或者多个角色做准备。因为REASSIGN OWNED不影响其他数据库中的对象，通常需要在包含有被删除的角色所拥有的对象的每一个数据库中都执行这个命令。

REASSIGN OWNED同时要求源角色和目标角色上的成员资格。

DROP OWNED命令可以简单地删掉一个或者多个角色所拥有的所有数据库对象。

REASSIGN OWNED命令不会影响授予给old\_roles的在它们不拥有的对象上的任何特权。同样，它不会影响ALTER DEFAULT PRIVILEGES创建的默认特权。DROP OWNED可以回收那些特权。

更多讨论请见Section 19.4

## 兼容性

REASSIGN OWNED命令是一种 LightDB扩展。

## 另见

DROP OWNED, DROP ROLE, ALTER DATABASE

---

# REFRESH MATERIALIZED VIEW

REFRESH MATERIALIZED VIEW — 替换一个物化视图的内容

## Synopsis

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name  
[ WITH [ NO ] DATA ]
```

## 描述

REFRESH MATERIALIZED VIEW完全替换一个物化视图的内容。你必须是该物化视图的属主才能执行这个命令。旧的内容会被抛弃。如果指定了 WITH DATA（或者作为默认值），支持查询将被执行以提供新的数据，并且会让物化视图将处于可扫描的状态。如果指定了 WITH NO DATA，则不会生成新数据并且会让物化视图处于一种不可扫描的状态。

CONCURRENTLY和WITH NO DATA 不能被一起指定。

## 参数

CONCURRENTLY

对物化视图的刷新不阻塞在该物化视图上的并发选择。如果没有这个选项，一次影响很多行的刷新将使用更少的资源并且更快结束，但是可能会阻塞其他尝试从物化视图中读取的连接。这个选项在只有少量行被影响的情况下可能会更快。

只有当物化视图上有至少一个UNIQUE索引（只用列名并且包括所有行）时，才允许这个选项。也就是说，它不能是表达式索引或者包括WHERE子句。

当物化视图还未被填充时，这个选项不能被使用。

即使带有这个选项，对于任意一个物化视图一次也只能运行一个 REFRESH。

name

要刷新的物化视图的名称（可以被模式限定）。

## 注解

虽然用于未来的CLUSTER操作的默认索引会被保持，REFRESH MATERIALIZED VIEW不会基于这个属性排序产生的行。如果希望数据在产生时排序，必须在支持查询中使用 ORDER BY子句。

## 示例

这个命令将使用物化视图order\_summary定义中的查询来替换该物化视图的内容，并且让它处于一种可扫描的状态：

```
REFRESH MATERIALIZED VIEW order_summary;
```

这个命令将释放与物化视图annual\_statistics\_basis相关的存储并且让它变成一种不可扫描的状态：



```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

## 兼容性

REFRESH MATERIALIZED VIEW是一种 LightDB扩展。

## 另见

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, DROP MATERIALIZED VIEW

---

# REINDEX

REINDEX — 重建索引

## Synopsis

```
REINDEX [ ( option [, ...] ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM }  
[ CONCURRENTLY ] name
```

其中 选项 可以是以下之一：

VERBOSE

## 描述

REINDEX使用索引的表里存储的数据重建一个索引， 并且替换该索引的旧拷贝。有一些场景需要使用REINDEX：

- 一个索引已经损坏，并且不再包含合法数据。尽管理论上这不会发生， 实际上索引会因为软件缺陷或硬件失效损坏。 REINDEX提供了一种恢复方法。
- 一个索引变得“臃肿”，其中包含很多空的或者近乎为空的页面。LightDB中的 B-树索引在特定的非 常规访问模式下可能会发生这种情况。REINDEX 提供了一种方法来减少索引的空间消耗，即制造一个新版本的索引，其中没有 死亡页面。详见Section 22.2
- 修改了一个索引的存储参数（例如填充因子），并且希望确保这种修改完全 生效。
- 如果索引在用CONCURRENTLY选项创建失败，该索引保留为一个“invalid”。 这类索引是无用的，但是可以方便的用REINDEX来重建它们。注意，只有REINDEX INDEX可以在无效的索引上执行并发创建。

## 参数

INDEX

重新创建指定的索引。 当与分区索引一起使用时，REINDEX的这种形式不能在事务块内部执行。

TABLE

重新创建指定表的所有索引。 如果该表有一个二级“TOAST”表，它也会被重索引。 当与分区表一起使用时，REINDEX的这种形式不能再事务块内部执行。

SCHEMA

重建指定方案的所有索引。如果这个方案中的一个表有次级的“TOAST”表，它也会被重建索引。共享系统目录上的索引也会被处理。这种形式的REINDEX不能在事务块内执行。

DATABASE

重新创建当前数据库内的所有索引。共享的系统目录上的索引也会被 处理。这种形式的REINDEX不能在一个 事务块内执行。

SYSTEM

重新创建当前数据库中在系统目录上的所有索引。共享系统目录上的 索引也被包括在内。用户表上的索引则不会被处理。这种形式的 REINDEX不能在一个事务块内执行。

name

要被重索引的特定索引、表或者数据库的名字。索引和表名可以被 `REINDEX DATABASE` 模式限定。当前，`REINDEX DATABASE`和 `REINDEX SYSTEM`只能重索引当前数据库，因此 它们的参数必须匹配当前数据库的名称。

CONCURRENTLY

使用此选项时，LightDB 将重建索引，而不在表上采取任何阻止并发插入、更新或删除的锁；标准的索引重建将会锁定表上的写操作(而不是读操作)，直到它完成。使用此选项一时，有几个事项需要注意；请参阅下面的Rebuilding Indexes Concurrently。

对于临时表，REINDEX始终是非并发的，因为没有其他会话可以访问它们，并且非并发重新索引更便宜。

VERBOSE

在每个索引被重建时打印进度报告。

## 注解

如果怀疑一个用户表上的索引损坏，可以使用 `REINDEX INDEX`或者 `REINDEX TABLE`简单地重建该索引 或者表上的所有索引。

如果你需要从一个系统表上的索引损坏中恢复，就更困难一些。在 这种情况下，对系统来说重要的是没有使用过任何可疑的索引本身（ 实际上，这种场景中，你可能会发现服务器进程会在启动时立刻崩溃， 这是因为对于损坏的索引的依赖）。要安全地恢复，服务器必须用 `-P`选项启动，这将阻止它使用索引来进行系统 目录查找。

这样做的一种方法是关闭服务器，并且启动一个单用户的 LightDB服务器，在其命令行 中包括`-P`选项。然后，可以发出 `REINDEX DATABASE`、`REINDEX SYSTEM`、`REINDEX TABLE`或者`REINDEX INDEX`， 具体使用哪个命令取决于你想要重构多少东西。如果有疑问，可以使用`REINDEX SYSTEM`来选择重建数据库中的所有系统索引。然后退出单用户服务器会话并且重启常规的服务器。更多关于如何与 单用户服务器接口交互的内容请见lightdb参考页。

在另一种方法中，可以开始一个常规的服务器会话，在其命令行选项 中包括`-P`。这样做的方法与客户端有关，但是在 所有基于libpq的客户端中都可以在开始客户端 之前设置PGOPTIONS环境变量为`-P`。注意虽然这种方法不要求用锁排斥其他客户端，在修复完成之前避免 其他用户连接到受损的数据库才是更加明智的。

REINDEX类似于删除索引并且重建索引，在其中索引内容会被从头开始建立。 不过，锁定方面的考虑却相当不同。 REINDEX会用锁排斥写，但不会排斥在索引的父表上的读。 它也会在处理的索引上取得一个ACCESS EXCLUSIVE锁，该锁将会阻塞对该索引的使用尝试。 相反，DROP INDEX 会暂时在附表上取得一个ACCESS EXCLUSIVE锁，阻塞写和读。 后续的CREATE INDEX会排斥写但不排斥读，由于该索引不存在，所以不会有读取它的尝试，这意味着不会有阻塞但是读操作可能被强制成昂贵的顺序扫描。

重索引单独一个索引或者表要求用户是该索引或表的拥有者。对方案或数据库重建索引要求是该方案或者数据库的拥有者。请特别注意，因此非超级用户有时无法重建其他用户拥有的表上的索引。不过，作为一种特例，当一个非超级用户发出`REINDEX DATABASE`、`REINDEX SCHEMA`或者`REINDEX SYSTEM`时，共享目录上的索引将被跳过，除非该用户拥有该目录（通常不会是这样）。当然，超级用户总是可以重建所有的索引。

不支持重新索引分区表或分区索引。可以单独重新索引每个分区。

## Rebuilding Indexes Concurrently

重建索引可能会影响数据库的常规操作。通常LightDB会锁定重建的表以防止写操作，并通过单次扫描表来执行整个索引构建。 其他事务仍可以读取表，但如果它们尝试在表中插入、更新或删除行，它们将被阻止，直到索引重建完成。 如果系统是实时生产数据库，这

可能会产生严重影响。非常大的表可能需要几个小时才能编制索引，即使对于较小的表，索引重建也会锁定编写器，这些时间段对于生产系统来说是不可接受的。

LightDB支持以最少的写入锁定来重建索引。此方法通过指定REINDEX的CONCURRENTLY选项来调用。使用此选项时，LightDB必须对需要重新生成的每个索引执行两次表扫描，并等待可能使用索引的所有现有事务的终止。此方法需要比标准索引重建更大的工作量，并且需要相当长的时间才能完成，因为它需要等待可能修改索引的未完成的事务。但是，由于它允许在重建索引时继续正常操作，此方法可用于在生产环境中重建索引。当然，重建索引所需的额外 CPU、内存和 I/O 负载可能会减慢其他操作的速度。

以下步骤发生在并发重建索引中。每个步骤在单独的事务中运行。如果要重建多个索引，则每个步骤在进入下一步之前都要循环遍历所有索引。

1. 新的临时索引定义将添加到目录pg\_index中。此定义将用于替换旧索引。一个SHARE UPDATE EXCLUSIVE会话级别的锁将放在要重建的索引以及其关联的表上，以防止处理时的任何模式修改。
2. 为每个新索引完成生成索引的首个操作。生成索引后，其标志pg\_index.indisready切换到“true”使其准备好插入，使其在执行生成的事务完成后对其他会话可见。此步骤在每个索引的单独事务中完成。
3. 然后执行第二个操作以添加在第一个操作运行时添加的元组。此步骤也在每个索引的单独事务中完成。
4. 引用索引的所有约束都已更改以引用新的索引定义，并且索引名称也已经更改。此时，pg\_index.indisvalid会为新索引切换到“true”，以及为旧索引切换到“false”，并且缓存无效会导致引用旧索引的所有会话失效。
5. 旧索引有pg\_index.indisready切换到“false”以防止任何新的元组插入，在等待可能引用旧索引的查询之后完成。
6. 旧索引被丢弃。索引和表的SHARE UPDATE EXCLUSIVE会话锁被释放。

如果在重建索引时出现问题，例如唯一索引中的唯一性冲突，REINDEX命令将失败，但会留下一个“invalid”新索引，在已经存在的索引之外。出于查询目的此索引将被忽略，因为它可能不完整；但是它仍将消耗更新开销。lsql \d命令将此类索引报告为 INVALID：

```
lightdb@postgres=# \d tab
      Table "public.tab"
  Column | Type   | Modifiers
  -----+-----+-----
   col   | integer |
Indexes:
    "idx" btree (col)
    "idx_ccnew" btree (col) INVALID
```

如果标记为INVALID的索引后缀为ccnew，那么它对应的是并发操作时创建的临时索引，推荐的恢复方法是使用DROP INDEX删除，然后再次尝试 REINDEX CONCURRENTLY。如果无效索引改为后缀ccold，则对应于无法删除的原始索引；推荐的恢复方法是删除所述索引，因为正确的重建已经成功。

常规索引创建允许在同一表上的其他常规索引创建同时发生，但在一个表上一次只能发生一个并发索引创建。在这两种情况下，不允许同时对表上其他类型的模式进行修改。另一个区别是，常规REINDEX TABLE或REINDEX INDEX命令可以在事务块中执行，但REINDEX CONCURRENTLY不能执行。

就像任何长时间运行的事务一样，表上的REINDEX会影响可以通过任何其他表上的并发VACUUM删除哪些元组。对非部分使用CONCURRENTLY选项的操作除外，并且不索引任何表达式。

REINDEX SYSTEM 不支持 CONCURRENTLY 因为系统目录不能并发重新索引。

此外，排除约束的索引不能并发重新编制索引。如果此命令中直接命名了这样的索引，则会引发错误。如果并发重新编制具有排除约束索引的表或数据库，将跳过这些索引。（它可以使用CONCURRENTLY选项来重新编制这样的索引）。

## 示例

重建单个索引：

```
REINDEX INDEX my_index;
```

重建表my\_table上的所有索引：

```
REINDEX TABLE my_table;
```

重建一个特定数据库中的所有索引，且不假设系统索引已经可用：

```
$ export PGOPTIONS="-P"  
$ !tsql broken_db  
...  
broken_db=> REINDEX DATABASE broken_db;  
broken_db=> \q
```

重建表的索引，在重建索引过程中不阻止对相关关系进行读写操作：

```
REINDEX TABLE CONCURRENTLY my_broken_table;
```

## 兼容性

在 SQL 标准中没有REINDEX命令。

## See Also

CREATE INDEX, DROP INDEX, reindexdb

---

# RELEASE SAVEPOINT

RELEASE SAVEPOINT — 销毁一个之前定义的保存点

## Synopsis

```
RELEASE [ SAVEPOINT ] savepoint_name
```

## 描述

RELEASE SAVEPOINT销毁在当前事务 中之前定义的一个保存点。

销毁一个保存点会使得它不能再作为一个回滚点，但是它没有其他用户 可见的行为。它不会撤销在该保存点被建立之后执行的命令的效果（要 这样做，可见ROLLBACK TO SAVEPOINT）。当不再需要一个 保存点时销毁它允许系统在事务结束之前回收一些资源。

RELEASE SAVEPOINT也会销毁所有 在该保存点建立之后建立的保存点。

## 参数

savepoint\_name

要销毁的保存点的名称。

## 注解

指定一个不是之前定义的保存点名称是错误。

当事务处于中止状态时不能释放保存点。

如果多个保存点具有相同的名称，只有最近被定义的那个会被释放。

## 示例

建立并且销毁一个保存点：

```
BEGIN;  
  INSERT INTO table1 VALUES (3);  
  SAVEPOINT my_savepoint;  
  INSERT INTO table1 VALUES (4);  
  RELEASE SAVEPOINT my_savepoint;  
COMMIT;
```

上述事务将插入 3 和 4。

## 兼容性

这个命令符合SQL标准。该标准指定关键词 SAVEPOINT是强制需要的，但 LightDB允许省略。

## 另见

BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

---

# REPLACE INTO

REPLACE INTO — 在主键约束冲突时更新表中的行

## Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
REPLACE INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
    VALUES ( { expression | DEFAULT } [, ...] ) [, ...]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

## 描述

REPLACE 和 INSERT 的工作方式完全相同，唯一的区别是，如果表中已存在与 newRow 具有相同值的旧行（针对主键索引），则在插入新行之前将删除旧行。

### Note

仅当表具有主键索引时，REPLACE 才有意义（不支持唯一索引）。否则，它将等同于 INSERT，因为没有索引可用于确定新行是否重复另一行。

所有列的值都来自于 REPLACE 语句中指定的值。任何缺少的列都将设置为默认值，就像 INSERT 一样。

要替换表中的行，必须同时具有 INSERT 和 UPDATE 权限。

## 参数

### 替换

本节介绍仅在替换新行时可能使用的参数。

`with_query`

WITH 子句允许您指定一个或多个子查询，可以通过名称在 REPLACE 查询中引用。有关详细信息，请参见 Section 7.8 和 SELECT。

`table_name`

`table_name` 的名称（可选模式限定符），表示现有表。

`alias`

`table_name` 的替代名称。提供别名时，它完全隐藏表的实际名称。

`column_name`

`table_name` 中的列名。如果需要，列名可以使用子字段名或数组下标进行限定。（仅替换复合列的某些字段会使其他字段为空。）

OVERRIDING SYSTEM VALUE

如果指定了这个子句，则为标识列提供的任何值都将覆盖默认的序列生成值。

对于定义为GENERATED ALWAYS的标识列，在未指定 OVERRIDING SYSTEM VALUE或OVERRIDING USER VALUE的情况下插入显式值（除DEFAULT外）是错误的。（对于定义为GENERATED BY DEFAULT的标识列， OVERRIDING SYSTEM VALUE是正常行为，指定它不起作用，但是LightDB将其作为扩展允许。）

#### OVERRIDING USER VALUE

如果指定了这个子句，则将忽略为标识列提供的任何值，并应用默认的序列生成值。

例如，在表之间复制值时，可以使用这个子句。写入 INSERT INTO tb12 OVERRIDING USER VALUE SELECT \* FROM tb11 将从tb11复制所有不是tb12的标识列的列，而tb12的标识列的值将由与tb12相关联的序列生成。

#### expression

要分配给相应列的表达式或值。

#### DEFAULT

相应的列将填充其默认值。标识列将填充由相关序列生成的新值。对于生成的列，虽然允许指定此选项，但它只是指定从其生成表达式计算列的正常行为。

#### output\_expression

在每行插入或更新后由REPLACE命令计算并返回的表达式。表达式可以使用table\_name指定的表的任何列名。写入\*以返回所插入或更新的行的所有列。

#### output\_name

用于返回列的名称。

## 输出

成功完成REPLACE命令后，将返回以下形式的命令标记：

```
REPLACE oid count
```

count和oid的含义与INSERT命令相同。

## 注意事项

如果指定的表是分区表，则每行将路由到相应的分区并插入其中。如果指定的表是分区，则如果输入行中有一行违反了分区约束，则会出现错误。

## 示例

考虑以下CREATE TABLE语句创建的表：

```
CREATE TABLE test (
  id INT NOT NULL,
  name CHAR(16),
  age INT DEFAULT 0,
  PRIMARY KEY(id)
);
```

当我们创建这个表并在1tsql客户端中运行以下语句时，结果如下所示：



```
lightdb@testdb=# REPLACE INTO test VALUES (1, 'Old name', '18');
INSERT 0 1
```

```
lightdb@testdb=# REPLACE INTO test VALUES (1, 'New name');
INSERT 0 1
```

```
lightdb@testdb=# SELECT * FROM test;
 id |      name      | age
-----+-----+-----
  1 | New name      |   0
(1 row)
```

考虑以下CREATE TABLE语句创建的表:

```
CREATE TABLE test (
  id INT NOT NULL,
  name CHAR(16),
  seq SERIAL,
  PRIMARY KEY(id)
);
```

当我们创建这个表并在1tsql客户端中运行以下语句时, 由于主键约束冲突, 'seq' 的值变为了4:

```
lightdb@testdb=# REPLACE INTO test VALUES (1, 'Old name1'), (2, 'Old name2');
INSERT 0 2
```

```
lightdb@testdb=# SELECT * FROM test;
 id |      name      | seq
-----+-----+-----
  1 | Old name1      |   1
  2 | Old name2      |   2
(2 rows)
```

```
lightdb@testdb=# REPLACE INTO test VALUES (1, 'New name');
INSERT 0 1
```

```
lightdb@testdb=# SELECT * FROM test;
 id |      name      | seq
-----+-----+-----
  2 | Old name2      |   2
  1 | New name      |   4
(2 rows)
```

## 兼容性

REPLACE是LightDB对SQL标准的扩展。它可以插入、删除并插入。只有当数据库的语法兼容类型为mysql时, 才支持REPLACE。详情请参见lightdb\_syntax\_compatible\_type。

---

# RESET

RESET — 把一个运行时参数的值恢复到默认值

## Synopsis

```
RESET configuration_parameter  
RESET ALL
```

## 描述

RESET把运行时参数恢复到它们的默认值。RESET是

```
SET configuration_parameter TO DEFAULT
```

的另一种写法。 详见SET。

默认值被定义为如果在当前会话中没有发出过SET， `configuration_parameter` 参数必须具有的值。这个值的实际来源可能是一个编译在内部的默认值、 `configuration_file` 配置文件、命令行选项、或者针对每个数据库或者每个用户的默认设置。 `RESET` 和把它定义成“在会话开始时该参数得到的值”有细微的差别，因为如果该值来自于配置文件，它将被重置为现在配置文件所指定 `configuration_file` 的任何东西。详见Chapter 17

RESET的事务行为和SET相同： 它的效果会被事务回滚撤销。

## 参数

`configuration_parameter`

一个可设置的运行时参数名称。可用的参数记录在 Chapter 1以及 SET参考页中。

ALL

把所有可设置的运行时参数重置为默认值。

## 示例

把timezone配置变量设置为默认值：

```
RESET timezone;
```

## 兼容性

RESET是一种 LightDB扩展。

## 另见

SET, SHOW

---

# REVOKE

REVOKE — 移除访问特权

## Synopsis

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
      [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
        | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
      [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
      [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
        | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
```

```

    ON { { FUNCTION | PROCEDURE | ROUTINE } function_name [ ( [ [ argmode ]
[ arg_name ] arg_type [, ...] ] ) ] [, ...]
        | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name
[, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
    role_name [, ...] FROM role_specification [, ...]
    [ GRANTED BY role_specification ]
    [ CASCADE | RESTRICT ]

```

where role\_specification can be:

```

    [ GROUP ] role_name
    | PUBLIC
    | CURRENT_USER
    | SESSION_USER

```

## 描述

REVOKE命令收回之前从一个或者更多角色 授予的特权。关键词PUBLIC隐式定义的全部角色的组。

特权类型的含义见GRANT命令的描述。

注意任何特定角色拥有的特权包括直接授予给它的特权、从它作为其成员的 角色中得到的特权以及授予给PUBLIC的特权。因此， 从PUBLIC收回SELECT特权并不一定会意味 着所有角

色都会失去在该对象上的SELECT特权：那些直接被授予的或者通过另一个角色被授予的角色仍然会拥有它。类似地，从一个用户收回SELECT后，如果PUBLIC或者另一个成员关系角色仍有SELECT权利，该用户还是可以使用SELECT。

如果指定了GRANT OPTION FOR，只会回收该特权的授予选项，特权本身不被回收。否则，特权及其授予选项都会被回收。

如果一个用户持有一个带有授予选项的特权并且把它授予给了其他用户，那么被那些其他用户持有的该特权被称为依赖特权。如果第一个用户持有的该特权或者授予选项正在被收回且存在依赖特权，指定CASCADE可以连带回收那些依赖特权，不指定则会导致回收动作失败。这种递归回收只影响通过可追溯到该REVOKE命令的主体的用户链授予的特权。因此，如果该特权经由其他用户授予给受影响用户，受影响用户可能实际上还保留有该特权。

在回收一个表上的特权时，也会在该表的每一个列上自动回收对应的列特权（如果有）。在另一方面，如果一个角色已经被授予一个表上的特权，那么从个别的列上回收同一个特权将不会生效。

在回收一个角色中的成员关系时，GRANT OPTION被改称为ADMIN OPTION，但行为是类似的。这种形式的命令也允许使用GRANTED BY选项，但该选项当前被忽略（除了检查指定角色的存在）。也要注意这种形式的命令不允许在role\_specification中使用噪声词GROUP。

## 注解

用户只能回收由它直接授出的特权。例如，如果用户A已经把一个带有授予选项的特权授予给了用户B，并且用户B接着把它授予给了用户C，那么用户A无法直接从C收回该特权。反而，用户A可以从用户B收回该授予选项并且使用CASCADE选项，这样该特权会被依次从用户C回收。对于另一个例子，如果A和B都把同一个特权授予给了C，A能够收回它们自己的授权但不能收回B的授权，因此C实际上仍将拥有该特权。

当一个对象的非拥有者尝试REVOKE该对象上的特权时，如果该用户在该对象上什么特权都不拥有，该命令会立刻失败。只要有某个特权可用，该命令将继续，但是它只会收回那些它具有授予选项的特权。如果没有持有授予选项，REVOKE ALL PRIVILEGES形式将发出一个警告，而其他形式在没有持有该命令中特别提到的任何特权的授予选项时就会发出警告（原则上，这些语句也适用于对象拥有者，但是由于拥有者总是被认为持有所有授予选项，这些情况永远不会发生）。

如果一个超级用户选择发出一个GRANT或者REVOKE命令，该命令就好像被受影响对象的拥有者发出的一样被执行。因为所有特权最终来自于对象拥有者（可能是间接地通过授予选项链），可以由超级用户收回所有特权，但是这可能需要前述的CASCADE。

REVOKE也可以由一个并非受影响对象的拥有者的角色完成，但是该角色应该是一个拥有该对象的角色的成员或者是一个在该对象上拥有特权的WITH GRANT OPTION的角色的成员。在这种情况下，该命令就好像被实际拥有该对象或者特权的WITH GRANT OPTION的包含角色发出的一样被执行。例如，如果表t1被角色g1拥有，而u1是g1的一个成员，那么u1能收回t1上被记录为由g1授出的特权。这会包括由u1以及由角色g1的其他成员完成的授予。

如果执行REVOKE的角色持有通过多于一条角色成员关系路径间接得到的特权，其中哪一条包含将被用于执行该命令的角色是没有被指明的。在这种情况下，最好使用SET ROLE成为你想作为其身份执行REVOKE的特定角色。如果无法做到这一点可能会导致回收超过你预期的特权，或者根本回收不了任何东西。

关于特定权限类型的更多信息请参加Section 5.7，以及如何检查对象的权限。

## 示例

从public收回表films上的插入特权：

```
REVOKE INSERT ON films FROM PUBLIC;
```

从用户manuel收回视图 kinds上的所有特权:

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

注意着实际意味着“收回所有我授出的特权”。

从用户joe收回角色admins中的成员关系:

```
REVOKE admins FROM joe;
```

## 兼容性

GRANT命令的兼容性注解同样适用于 REVOKE。根据标准，关键词 RESTRICT或CASCADE 是必要的，但是LightDB默认假定为 RESTRICT。

## 另见

GRANT, ALTER DEFAULT PRIVILEGES

---

# ROLLBACK

ROLLBACK — 中止当前事务

## Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

## 描述

ROLLBACK回滚当前事务并且导致 该事务所作的所有更新都被抛弃。

## 参数

WORK  
TRANSACTION

可选关键词，没有效果。

AND CHAIN

如果指定了AND CHAIN，则立即启动与刚刚完成事务具有相同事务特征（参见 SET TRANSACTION）的新事务。 否则，不会启动任何新事务。

## 注解

使用COMMIT可成功地终止一个事务。

在一个事务块之外发出ROLLBACK会发出一个警告并且不会有效果。 事务块之外的ROLLBACK AND CHAIN 是一个错误。

## 示例

要中止所有更改：

```
ROLLBACK;
```

## 兼容性

命令ROLLBACK符合 SQL 标准。窗体ROLLBACK TRANSACTION是一个LightDB 扩展。

## 另见

BEGIN, COMMIT, ROLLBACK TO SAVEPOINT

---

# ROLLBACK PREPARED

ROLLBACK PREPARED — 取消一个之前为两阶段提交准备好的事务

## Synopsis

```
ROLLBACK PREPARED transaction_id
```

## 描述

ROLLBACK PREPARED回滚 一个处于准备好状态的事务。

## 参数

`transaction_id`

要被回滚的事务的事务标识符。

## 注解

要回滚一个准备好的事务，你必须是原先执行该事务的同一个用户或者 是一个超级用户。但是你必须处在执行该事务的同一个会话中。

这个命令不能在一个事务块内被执行。准备好的事务会被立刻回滚。

`pg_prepared_xacts` 系统视图中列出了当前可用的所有准备好的事务。

## 例子

用事务标识符`foobar`回滚对应的事务：

```
ROLLBACK PREPARED 'foobar';
```

## 兼容性

ROLLBACK PREPARED是一种 LightDB扩展。其意图是用于 外部事务管理系统，其中有些已经被标准涵盖（例如 X/Open XA），但是那些系统的 SQL 方面未被标准化。

## 另见

PREPARE TRANSACTION, COMMIT PREPARED



---

# ROLLBACK TO SAVEPOINT

ROLLBACK TO SAVEPOINT — 回滚到一个保存点

## Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

## 描述

回滚在该保存点被建立之后执行的所有命令。该保存点保持有效并且可以在以后再次回滚到它（如果需要）。

ROLLBACK TO SAVEPOINT隐式地销毁在所提及的保存点之后建立的所有保存点。

## 参数

savepoint\_name

要回滚到的保存点。

## 注解

使用RELEASE SAVEPOINT销毁一个保存点而不抛弃在它建立之后被执行的命令的效果。

指定一个没有被建立的保存点是一种错误。

相对于保存点，游标有一点非事务的行为。在保存点被回滚时，任何在该保存点内被打开的游标将会被关闭。如果一个先前打开的游标在一个保存点内被FETCH或MOVE命令所影响，而该保存点后来又被回滚，那么该游标将保持FETCH使它指向的位置（也就是说由FETCH导致的游标动作不会被回滚）。回滚也不能撤销关闭一个游标。不过，其他由游标查询导致的副作用（例如被该查询所调用的易变函数的副作用）可以被回滚，只要它们发生在一个后来被回滚的保存点期间。如果一个游标的执行导致事务中止，它会被置于一种不能被执行的状态，这样当事务被用ROLLBACK TO SAVEPOINT恢复后，该游标也不再能被使用。

## 示例

要撤销在my\_savepoint建立后执行的命令的效果：

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

游标位置不会受保存点回滚的影响：

```
BEGIN;
```

```
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
```

```
SAVEPOINT foo;
```

```
FETCH 1 FROM foo;  
?column?
```

```
-----  
1  
  
ROLLBACK TO SAVEPOINT foo;  
  
FETCH 1 FROM foo;  
?column?  
-----  
2  
  
COMMIT;
```

## 兼容性

SQL标准指定关键词 `SAVEPOINT`是强制的，但是LightDB 和Oracle允许省略它。SQL 只允许WORK而不是TRANSACTION作为ROLLBACK之后的噪声词。 还有，SQL 有一个可选的子句 `AND [ NO ] CHAIN`，当前 LightDB并不支持。在其他方面，这个命令符合 SQL 标准。

## 另见

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT

---

# SAVEPOINT

SAVEPOINT — 在当前事务中定义一个新的保存点

## Synopsis

```
SAVEPOINT savepoint_name
```

## 描述

SAVEPOINT在当前事务中建立一个新保存点。

保存点是事务内的一种特殊标记，它允许所有在它被建立之后执行的命令被回滚，把该事务的状态恢复到它处于保存点时的样子。

## 参数

savepoint\_name

给新保存点的名字。

## 注解

使用ROLLBACK TO回滚到一个保存点。使用RELEASE SAVEPOINT销毁一个保存点，但保持在它被建立之后执行的命令的效果。

保存点只能在一个事务块内建立。可以在一个事务内定义多个保存点。

## 示例

要建立一个保存点并且后来撤销在它建立之后执行的所有命令的效果：

```
BEGIN;
  INSERT INTO table1 VALUES (1);
  SAVEPOINT my_savepoint;
  INSERT INTO table1 VALUES (2);
  ROLLBACK TO SAVEPOINT my_savepoint;
  INSERT INTO table1 VALUES (3);
COMMIT;
```

上面的事务将插入值 1 和 3，但不会插入 2。

要建立并且稍后销毁一个保存点：

```
BEGIN;
  INSERT INTO table1 VALUES (3);
  SAVEPOINT my_savepoint;
  INSERT INTO table1 VALUES (4);
  RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上面的事务将插入 3 和 4。

要使用单个保存点名称:

```
BEGIN;
  INSERT INTO table1 VALUES (1);
  SAVEPOINT my_savepoint;
  INSERT INTO table1 VALUES (2);
  SAVEPOINT my_savepoint;
  INSERT INTO table1 VALUES (3);

  -- rollback to the second savepoint
  ROLLBACK TO SAVEPOINT my_savepoint;
  SELECT * FROM table1;           -- shows rows 1 and 2

  -- release the second savepoint
  RELEASE SAVEPOINT my_savepoint;

  -- rollback to the first savepoint
  ROLLBACK TO SAVEPOINT my_savepoint;
  SELECT * FROM table1;           -- shows only row 1
COMMIT;
```

上述事务显示先回滚第三行，然后回滚第二行。

## 兼容性

当建立另一个同名保存点时，SQL 要求之前的那个保存点自动被销毁。在 LightDB 中，旧的保存点会被保留，不过在进行回滚或释放时只能使用最近的那一个（用 `RELEASE SAVEPOINT` 释放较新的保存点将会导致较旧的保存点再次变得可以被 `ROLLBACK TO SAVEPOINT` 和 `RELEASE SAVEPOINT` 访问）。在其他方面，`SAVEPOINT` 完全符合 SQL。

## 另见

`BEGIN`, `COMMIT`, `RELEASE SAVEPOINT`, `ROLLBACK`, `ROLLBACK TO SAVEPOINT`

---

# SECURITY LABEL

SECURITY LABEL — 定义或更改应用到一个对象的安全标签

## Synopsis

```
SECURITY LABEL [ FOR provider ] ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  DATABASE object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN TABLE object_name
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [ , ... ] ) ) ] |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [ , ... ] ) ) ] |
  PUBLICATION object_name |
  ROLE object_name |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [ , ... ] ) ) ] |
  SCHEMA object_name |
  SEQUENCE object_name |
  SUBSCRIPTION object_name |
  TABLESPACE object_name |
  TYPE object_name |
  VIEW object_name
} IS 'label'
```

其中 `aggregate_signature` 是:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ]
argtype [ , ... ]
```

## 描述

SECURITY LABEL 对一个数据库对象应用一个安全标签。可以把任意数量的安全标签（每个标签提供者对应一个）关联到一个给定的数据库对象。标签提供者是使用函数 `register_label_provider` 注册自己的可装载模块。

### Note

`register_label_provider` 不是一个 SQL 函数，它只能在被载入到后端的 C 代码中调用。

标签提供者决定一个给定标签是否合法并且它是否可以被分配该标签给一个给定对象。一个给定标签的含义也同样由标签提供者判断。LightDB 没有对一个标签提供者是否必须或者如何解释安全标签做出限定，它仅仅只是提供了一种机制来存储它们。实际上，这个功能

是 为了允许与基于标签的强制访问控制（MAC）系统集成（例如 SELinux）。这类系统会基于对象标签而不是传统的自主 访问控制（DAC）概念（例如用户和组）做出所有访问控制决定。

## 参数

object\_name  
table\_name.column\_name  
aggregate\_name  
function\_name  
procedure\_name  
routine\_name

要被贴上标签的对象的名称。位于模式中的对象（表，函数，等。）名称可以是模式限定的。

provider

这个标签相关联的提供者的名称。所提到的提供者必须已被载入并且必须赞同所提出 的标签操作。如果正好只载入了一个提供者，可以出于简洁的需要忽略提供者的名称。

argmode

一个函数，存储过程或者聚集函数参数的模式：IN、OUT、 INOUT或者VARIADIC。如果被忽略，默认值会是 IN。注意SECURITY LABEL并不真正 关心OUT参数，因为判断函数的身份时只需要输入参数。因此列出 IN、INOUT和VARIADIC参数足矣。

argname

一个函数，存储过程或者聚集函数参数的名称。注意SECURITY LABEL 并不真正关心参数的名称，因为判断函数的身份时只需要参数的数据类型。

argtype

一个函数，存储过程或聚集函数参数的数据类型。

large\_object\_oid

大对象的 OID。

PROCEDURAL

这是一个噪声词。

label

写成一个字符串文本的新安全标签。如果写成NULL表示删除 原有的安全标签。

## 示例

下面的例子展示了如何更改一个表的安全标签。

```
SECURITY LABEL FOR selinux ON TABLE mytable IS
'system_u:object_r:sepgsql_table_t:s0';
```

## 兼容性

在 SQL 标准中没有SECURITY LABEL命令。

另见

sepgsql, [src/test/modules/dummy\\_seclabel](#)

---

# SELECT

SELECT, TABLE, WITH — 从一个表或视图检索行

## Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ { USE | IGNORE | FORCE } { INDEX | KEY } [ FOR { JOIN | ORDER BY | GROUP
BY } ] (index_list) ]
    [ WHERE condition ]
    [ START WITH start_expression ]
    [ CONNECT BY { PRIOR parent_expr = child_expr | child_expr =
PRIOR parent_expr } ]
    [ GROUP BY grouping_element [, ...] [WITH ROLLUP] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition )[, ...] ]
    [ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES } ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name
[, ...] ] [ NOWAIT | SKIP LOCKED ] [...] ]
```

其中 from\_item 可以是以下之一:

```
[ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE
( seed ) ] ]
    [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] )
        [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias
[, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias
( column_definition [, ...] )
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition
[, ...] )
    [ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS
( column_definition [, ...] ) ] [, ...] )
        [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias
[, ...] ) ] ]
    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias
[, ...] ) ] ] opt_conversion_clause
    ( select ) [ AS ] alias [ ( column_alias [, ...] ) ] opt_conversion_clause
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING
( join_column [, ...] ) ]
```

并且 grouping\_element 可以是以下之一:



```
( )
expression
( expression [, ...] )
ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
CUBE ( { expression | ( expression [, ...] ) } [, ...] )
GROUPING SETS ( grouping_element [, ...] )
```

并且 with\_query 是:

```
with_query_name [ ( column_name [, ...] ) ] AS [ [ NOT ] MATERIALIZED ]
( select | values | insert | update | delete )
```

TABLE [ ONLY ] table\_name [ \* ]

并且 opt\_conversion\_clause 是:

```
pivot_clause [ * ] [ AS ] alias
```

并且 pivot\_clause 是:

```
PIVOT ( function_name (argument) [ AS ] alias
FOR columnref IN ( column_definition [, ...] ) ) [ AS ] alias
```

## 描述

SELECT从零或更多表中检索行。SELECT的通常处理如下:

1. WITH列表中的所有查询都会被计算。这些查询实际充当了在FROM列表中可以引用的临时表。在FROM中被引用多次的WITH查询只会被计算一次,除非另有说明,否则NOT MATERIALIZED。(见下文的WITH Clause)。
2. FROM列表中的所有元素都会被计算(FROM中的每一个元素都是一个真实表或者虚拟表)。如果在FROM列表中指定了多于一个元素,它们会被交叉连接在一起(见下文的FROM Clause)。
3. 如果指定了WHERE子句,所有不满足该条件的行都会被从输出中消除(见下文的WHERE Clause)。
4. 如果指定了WHERE子句,则不满足条件的所有行都将从输出中删除。(请参见下面的WHERE Clause。)
5. 通过在条件中使用 '(+)', 你可以像 Oracle 一样指定外连接。(See oracle plus below.)
6. 如果指定了CONNECT BY子句,则可以按层次顺序选择行。(请参见下面的CONNECT BY Clause。)对于由分层查询返回的每一行,LEVEL伪列对于根行返回1,对于根的子项返回2,依此类推。SYS\_CONNECT\_BY\_PATH仅在分层查询中有效。对于由CONNECT BY条件返回的每一行,SYS\_CONNECT\_BY\_PATH(column, char)使用char分隔的column值。CONNECT\_BY\_ROOT是一种仅在分层查询中有效的一元运算符。当使用此运算符对列进行限时,它会返回使用根行数据的列值。CONNECT BY LEVEL < n 显示 n 级以下的所有递归结果。CONNECT\_BY\_ISLEAF 是一个伪列,通常与递归查询(使用CONNECT BY子句)一起使用。它的作用是标识在递归查询的上下文中,当前行是否为叶子节点(没有子节点)。CONNECT\_BY\_ISLEAF 列返回一个布尔值,如果当前行没有子节点,则返回 1(表示是叶子节点),否则返回 0(表示不是叶子节点)。
7. 如果指定了GROUP BY子句或者如果有聚集函数,输出会被组合成由在一个或者多个值上匹配的行构成的分组,并且在其上计算聚集函数的结果。如果出现了HAVING子句,它会消除不满足给定条件的分组(见下文的GROUP BY Clause以及HAVING Clause)。

8. 对于每一个被选中的行或者行组，会使用SELECT 输出表达式计算实际的输出行（见下文的 SELECT List）。
9. SELECT DISTINCT从结果中消除重复的行。SELECT DISTINCT ON消除在所有指定表达式上匹配的行。SELECT ALL（默认）将返回所有候选行，包括重复的行（见下文的DISTINCT Clause）。
10. 通过使用操作符UNION、INTERSECT和EXCEPT，多于一个SELECT语句的输出可以被整合形成一个结果集。UNION操作符返回位于一个或者两个结果集中的全部行。INTERSECT操作符返回同时位于两个结果集中的所有行。EXCEPT操作符返回位于第一个结果集但在第二个结果集中的行。在所有三种情况下，重复行都会被消除（除非指定ALL）。可以增加噪声词DISTINCT来显式地消除重复行。注意虽然 ALL是SELECT自身的默认行为，但这里DISTINCT是默认行为（见下文的 UNION Clause、INTERSECT Clause以及 EXCEPT Clause）。
11. 如果指定了ORDER BY子句，被返回的行会以指定的顺序排序。如果没有给定ORDER BY，系统会以能最快产生行的顺序返回它们（见下文的 ORDER BY Clause）。
12. 如果指定了LIMIT（或FETCH FIRST）或者OFFSET子句，SELECT 语句只返回结果行的一个子集（见下文的LIMIT Clause）。
13. 如果指定了FOR UPDATE、FOR NO KEY UPDATE、FOR SHARE 或者FOR KEY SHARE，SELECT语句会把被选中的行锁定而不让并发更新访问它们（见下文的The Locking Clause）。

你必须拥有在一个SELECT命令中使用的每一列上的 SELECT特权。FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE或者FOR KEY SHARE 还要求（对这样选中的每一个表至少一列的）UPDATE特权。

## 参数

### WITH 子句

WITH子句允许你指定一个或者多个在主查询中可以 其名称引用的子查询。在主查询期间子查询实际扮演了临时表或者视图 的角色。每一个子查询都可以是一个SELECT、TABLE、VALUES、INSERT、UPDATE或者 DELETE语句。在WITH中书写 一个数据修改语句（INSERT、UPDATE或者 DELETE）时，通常要包括一个 RETURNING子句。构成被主查询读取的临时表的是 RETURNING的输出，而不是该语句修改的 底层表。如果省略RETURNING，该语句仍会被执行，但是它 不会产生输出，因此它不能作为一个表从主查询引用。

对于每一个WITH查询，都必须指定一个名称（无需模式限定）。可选地，可以指定一个列名列表。如果省略该列表，会从该子查询中推导列名。

如果指定了RECURSIVE，则允许一个 SELECT子查询使用名称引用自身。这样一个子查询的形式必须是

```
non_recursive_term UNION [ ALL | DISTINCT ] recursive_term
```

其中递归自引用必须出现在UNION的右手边。每个 查询中只允许一个递归自引用。不支持递归数据修改语句，但是 可以在一个数据查询语句中使用一个递归 SELECT查询的结果。例子可见 Section 7.8

RECURSIVE的另一个效果是 WITH查询不需要被排序：一个查询可以引用另一个 在列表中比它靠后的查询（不过，循环引用或者互递归没有实现）。如果没有RECURSIVE，WITH 查询只能引用在WITH列表中位置更前面的兄弟 WITH查询。

当WITH子句中有多个查询时，RECURSIVE应只编写一次，紧跟在WITH之后。 它适用于WITH子句中的所有查询，尽管它对不使用递归或前向引用的查询没有影响。

主查询和 WITH 查询都(概念上)同时执行。这意味着 WITH 中的修改数据语句的影响不能从查询的其他部分看到, 除非读取其 RETURNING 输出。如果两个这样的修改数据语句尝试修改同一行, 则结果是未指定的。

WITH 查询的一个关键特性是, 它们通常只在主查询的每次执行中被评估一次, 即使主查询多次引用它们。特别是, 修改数据语句保证只执行一次, 无论主查询是否读取它们的所有或任何输出。

但是, WITH查询可以标记为NOT MATERIALIZED以移除此保证。在这种情况下, WITH查询可以折叠到主查询中, 就好像它是主查询的FROM子句中的简单的sub-SELECT。如果主查询多次引用WITH查询, 则会导致重复计算;但是, 如果每次此类使用只需要WITH查询的总输出中的几行, NOT MATERIALIZED可以通过允许查询联合优化来节省开销。NOT MATERIALIZED被忽略, 如果它被附加到一个递归的WITH查询, 或者不是边际效应无关的(也就是说, 不是包含非易失性函数的普通的SELECT)。

默认情况下, 如果查询在主查询的FROM子句中仅一次使用, 则边际效应无关的WITH查询将折叠到主查询中。这允许在语义不可见的情况下两个查询级别的联合优化。但是, 通过将WITH查询标记为MATERIALIZED, 可以防止此类折叠。这可能很有用, 例如, 如果WITH查询被用作优化围栏, 以防止规划者选择错误计划。LightDBv12版本之前没有做过这样的折叠, 所以为旧版本编写的查询可能依赖于WITH作为优化围栏。

更多信息请见Section 7.8

## FROM 子句

FROM子句为SELECT 指定一个或者更多源表。如果指定了多个源表, 结果将是所有源表的笛卡尔积(交叉连接)。但是通常会增加限定条件(通过 WHERE)来把返回的行限制为该笛卡尔积的一个小子集。

FROM子句可以包含下列元素:

`table_name`

一个现有表或视图的名称(可以是模式限定的)。如果在表名前指定了 ONLY, 则只会扫描该表。如果没有指定 ONLY, 该表及其所有后代表(如果有)都会被扫描。可选地, 可以在表名后指定\*来显式地指示包括后代表。

`alias`

一个包含别名的FROM项的替代名称。别名被用于 让书写简洁或者消除自连接中的混淆(其中同一个表会被扫描多次)。当提供一个别名时, 表或者函数的实际名称会被隐藏。例如, 给定FROM foo AS f, SELECT的剩余部分就必须以 f而不是foo来引用这个FROM项。如果写了一个别名, 还可以写一个列别名列表来为该表的一个或者多个列提供替代名称。

`TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ]`

`table_name`之后的 TABLESAMPLE子句表示应该用指定的 `sampling_method` 来检索表中的子集。这种采样优先于任何其他过滤器(例如 WHERE子句)。标准 LightDB发布包括两种采样方法: BERNOULLI和SYSTEM, 其他采样方法可以通过扩展安装在数据库中。

BERNOULLI以及SYSTEM采样方法都接受 一个参数, 它表示要采样的表的分数, 表示为一个 0 到 100 之间的百分数。这个参数可以是任意的 实数值表达式(其他的采样方法可能接受更多或者不同的 参数)。这两种方法都返回一个随机选取的该表采样, 其中包含了指定 百分数的表行。BERNOULLI方法扫描整个表并且 用指定的几率选择或者忽略行。SYSTEM方法会做 块层的采样, 每个块都有指定的机会能被选中, 被选中块中的所有行都会被返回。在指定较小的采样百分数时, SYSTEM 方法要比BERNOULLI方法快很多, 但是前者可能 由于聚簇效应返回随机性较差的表采样。

可选的REPEATABLE子句指定一个用于产生采样方法中随机数的种子数或表达式。种子值可以是任何非空浮点值。如果查询时表没有被更改，指定相同种子和argument值的两个查询将会选择该表相同的采样。但是不同的种子值通常将会产生不同的采样。如果没有给出REPEATABLE，则会基于一个系统产生的种子为每一个查询选择一个新的随机采样。注意有些扩展采样方法不接受REPEATABLE，并且将总是为每一次使用产生新的采样。

#### select

一个子-SELECT可以出现在FROM子句中。这就好像把它的输出创建为一个存在于该SELECT命令期间的临时表。注意子-SELECT必须用圆括号包围，并且必须为它提供一个别名。也可以在这里使用一个VALUES command命令。

#### with\_query\_name

可以通过写一个WITH查询的名称来引用它，就好像该查询的名称是一个表名（实际上，该WITH查询会为主查询隐藏任何具有相同名称的真实表。如果必要，你可以使用带模式限定的方式以相同的名称来引用真实表）。可以像表一样，以同样的方式提供一个别名。

#### function\_name

函数调用可以出现在FROM子句中（对于返回结果集合的函数特别有用，但是可以使用任何函数）。这就好像把该函数的输出创建为一个存在于该SELECT命令期间的临时表。如果函数的结果类型是合成的（包括函数有多个OUT参数的情况），在隐式表中每个属性成为单独的列。

当为该函数调用增加可选的WITH ORDINALITY子句时，会在该函数的结果列之后附加一个bigint类型的列。这个列的编号函数的结果集的行，从1开始。默认的，这个列被命名为ordinality。

可以用和表一样的方式提供一个别名。如果写了一个别名，还可以写一个列别名列表来为该函数的组合返回类型的一个或者多个属性提供替代名称，如果有的话包括ordinality列。

通过把多个函数调用包围在ROWS FROM( ... )中可以把它们整合在单个FROM-子句中。这样一个项的输出是把每一个函数的第一行串接起来，然后是每个函数的第二行，以此类推。如果有些函数产生的行比其他函数少，则在缺失数据的地方放上空值，这样被返回的总行数总是和产生最多行的函数一样。

如果函数被定义为返回record数据类型，那么必须出现一个别名或者关键词AS，后面跟上形为( column\_name data\_type [, ... ] )的列定义列表。列定义列表必须匹配该函数返回的列的实际数量和类型。

在使用ROWS FROM( ... )语法时，如果函数之一要求一个列定义列表，最好把该列定义列表放在ROWS FROM( ... )中该函数的调用之后。当且仅当正好只有一个函数并且没有WITH ORDINALITY子句时，才能把列定义列表放在ROWS FROM( ... )结构后面。

要把ORDINALITY和列定义列表一起使用，你必须使用ROWS FROM( ... )语法，并且把列定义列表放在ROWS FROM( ... )里面。

#### join\_type

One of

- [ INNER ] JOIN
- LEFT [ OUTER ] JOIN
- RIGHT [ OUTER ] JOIN

- FULL [ OUTER ] JOIN

- CROSS JOIN

对于INNER和OUTER连接类型，必须指定一个连接条件，即 NATURAL、ON join\_condition或者 USING (join\_column [, ...]) 之一（只能有一种）。其含义见下文。对于 CROSS JOIN，上述子句不能出现。

一个JOIN子句联合两个FROM项（为了方便我们称之为“表”，尽管实际上它们可以是任何类型的FROM项）。如有必要可以使用圆括号确定嵌套的顺序。在没有圆括号时，JOIN会从左至右嵌套。在任何情况下，JOIN的联合比分隔FROM-列表项的逗号更强。

CROSS JOIN和INNER JOIN 会产生简单的笛卡尔积，也就是与在FROM的顶层列出两个表得到的结果相同，但是要用连接条件（如果有）约束该结果。CROSS JOIN与INNER JOIN ON (TRUE)等效，也就是说条件不会移除任何行。这些连接类型只是一种记号上的方便，因为没有什么是你用纯粹的FROM和 WHERE能做而它们不能做的。

LEFT OUTER JOIN返回被限制过的笛卡尔积中的所有行（即所有通过了其连接条件的组合行），外加左手中没有相应的通过了连接条件的右手行的每一行的拷贝。通过在右手中插入空值，这种左手行会被扩展为连接表的完整行。注意在决定哪些行匹配时，只考虑JOIN子句自身的条件。之后才应用外条件。

相反，RIGHT OUTER JOIN返回所有连接行，外加每一个没有匹配上的右手行（在左端用空值扩展）。这只是为了记号上的方便，因为你可以通过交换左右表把它转换成一个LEFT OUTER JOIN。

FULL OUTER JOIN返回所有连接行，外加每一个没有匹配上的左手行（在右端用空值扩展），再外加每一个没有匹配上的右手行（在左端用空值扩展）。

#### ON join\_condition

join\_condition 是一个会得到boolean类型值的表达式（类似于一个 WHERE子句），它说明一次连接中哪些行被认为相匹配。

#### USING ( join\_column [, ...] )

形式USING ( a, b, ... )的子句是 ON left\_table.a = right\_table.a AND left\_table.b = right\_table.b ... 的简写。还有，USING表示每一对相等列中只有一个会被包括在连接输出中。

#### NATURAL

NATURAL是一个USING列表的速记，该列表中提到两个表中具有匹配名称的所有的列。如果没有公共列名，则NATURAL等效于ON TRUE。

#### LATERAL

LATERAL关键词可以放在一个子-SELECT FROM项前面。这允许该子-SELECT引用FROM列表中在它之前的FROM项的列（如果没有LATERAL，每一个子-SELECT会被独立计算并且因此不能交叉引用任何其他FROM项）。

LATERAL也可以放在一个函数调用 FROM项前面，但是在这种情况下它只是一个噪声词，因为在任何情况下函数表达式都可以引用在它之前的 FROM项。

LATERAL项可以出现在FROM列表顶层，或者一个JOIN中。在后一种情况中，它也可以引用其作为右手端的JOIN左手端上的任何项。

当一个FROM项包含LATERAL交叉引用时，计算会如此进行：对提供被交叉引用列的FROM项的每一行或者提供那些列的多个FROM项的每一个行集，使用该行或者行集的

那些列值计算LATERAL项。结果行会与 计算得到它们的行进行通常的连接。对来自哪些列的源表的每一行或者行 集都会重复这样的步骤。

列的源表必须以INNER或者LEFT的方式连接到 LATERAL项，否则就没有用于为 LATERAL项计算每一个行集的良好行集。尽管 X RIGHT JOIN LATERAL Y这样的结构在语法上是合法的，但实际上不允许用于在Y中引用 X。

opt\_conversion\_clause

是

- PIVOT  
PIVOT只支持单聚集函数，单列多组的情景。 它作为FROM子句的一部分，可以在JOIN、NATURAL、LATERAL 连接中使用，完成行转列的功能。

PIVOT不支持以下五种情景：

- 不支持XML、ANY关键字
- 不支持多个聚集函数

```
PIVOT (
  SUM(sale_amount),
  COUNT(sale_amount)
  FOR customer_id
  IN (1, 2, 3, 4)
);
```

- 不支持多列分组

```
PIVOT (
  SUM(sale_amount)
  FOR (customer_id, prod_category)
  IN (
    (1, 'furniture') AS furn1,
    (2, 'furniture') AS furn2,
    (1, 'electronics') AS elec1,
    (2, 'electronics') AS elec2
  )
);
```

- 不支持IN( )

```
select * from test123
PIVOT (
  SUM(score) as tc
  FOR course
  IN ());
```

- PIVOT子句中使用的聚集函数参数只支持字段类型，不支持数字类型和'\*'

## WHERE 子句

可选的WHERE子句的形式

WHERE condition

其中condition 是任一计算得到布尔类型结果的表达式。任何不满足 这个条件的行都会从输出中被消除。如果用一行的实际值替换其中的 变量引用后，该表达式返回真，则该行符合条件。

## CONNECT BY 子句

可选的CONNECT BY子句的形式

```
CONNECT BY { PRIOR parent_expr = child_expr |
            child_expr = PRIOR parent_expr}.
```

CONNECT BY 指定分层结构中父行和子行之间的关系。 在分层查询中，条件中的一个表达式必须使用 PRIOR 运算符限定，以引用父行。

## START 子句

可选的START WITH子句的形式

```
[ START WITH start_expression ].
```

START WITH 指定分层结构的根行。

## GROUP BY 子句

可选的GROUP BY子句的形式

```
GROUP BY grouping_element [, ...] [WITH ROLLUP]
```

GROUP BY将会把所有被选择的行中共享相同分组表达式 值的那些行压缩成一个行。一个被用在 grouping\_element中的 expression可以是输入列名、输出列（SELECT列表项）的名称或序号或者由输入列 值构成的任意表达式。在出现歧义时，GROUP BY名称 将被解释为输入列名而不是输出列名。

如果任何GROUPING SETS、ROLLUP或者CUBE作为分组元素存在，则GROUP BY子句整体上定义了数个独立的分组集。 其效果等效于在子查询间构建一个UNION ALL，子查询带有分组集作为它们的GROUP BY子句。 在处理前可选的DISTINCT子句移除副本集合；它不做UNION ALL到UNION DISTINCT的转换 处理分组集的进一步细节请见Section 7.2.4

聚集函数（如果使用）会在组成每一个分组的所有行上进行计算，从而为每一个分组产生一个单独的值（如果有聚集函数但是没有 GROUP BY子句，则查询会被当成是由所有选中行构成 的一个单一分组）。传递给每一个聚集函数的行集合可以通过在聚集函数调用附加一个FILTER子句来进一步过滤，详见 Section 4.2.7 当存在一个 FILTER子句时，只有那些匹配它的行才会被包括在该聚集函数的输入中。

当存在GROUP BY子句或者任何聚集函数时， SELECT列表表达式不能引用非分组列（除非它出现在聚集函数中或者它函数依赖于分组列），因为这样做会导致返回 非分组列的值时会有多种可能的值。如果分组列是包含非分组列的表的主键（ 或者主键的子集），则存在函数依赖。

记住所有的聚集函数都是在HAVING子句或者 SELECT列表中的任何“标量”表达式之前被计算。 这意味着一个CASE表达式不能被用来跳过一个聚集表达式的 计算，见Section 4.2.15

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE不能和 GROUP BY一起指定。

WITH ROLLUP在mysql模式下生效。 WITH ROLLUP使用的基本价值是，在聚合查询中生成额外行来提供汇总数据。

## HAVING 子句

可选的HAVING子句的形式

```
HAVING condition
```

其中condition与 WHERE子句中指定的条件相同。

HAVING消除不满足该条件的分组行。 HAVING与WHERE不同： WHERE会在应用GROUP BY之前过滤个体行，而HAVING过滤由 GROUP BY创建的分组行。 condition中引用的每一个列必须无歧义地引用一个分组列（除非该引用出现在一个聚集 函数中或者该非分组列函数依赖于分组列）。

即使没有GROUP BY子句，HAVING 的存在也会把一个查询转变成一个分组查询。这和查询中包含聚集函数但没有 GROUP BY子句时的情况相同。所有被选择的行都被认为是一个 单一分组，并且SELECT列表和 HAVING子句只能引用聚集函数中的表列。如果该 HAVING条件为真，这样一个查询将会发出一个单一行； 否则不返回行。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE不能与 HAVING一起指定。

## WINDOW 子句

可选的WINDOW子句的形式

```
WINDOW window_name AS ( window_definition ) [, ...]
```

其中window\_name 是一个可以从OVER子句或者后续窗口定义中引用的名称。  
window\_definition是

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ]
[, ...] ]
[ frame_clause ]
```

如果指定了一个existing\_window\_name， 它必须引用WINDOW列表中一个更早出现的项。新窗口将从 该项中复制它的划分子句以及排序子句（如果有）。在这种情况下，新窗口 不能指定它自己的PARTITION BY子句，并且它只能在被复制 窗口没有ORDER BY的情况下指定该子句。新窗口总是使用它 自己的帧子句，被复制的窗口不必指定一个帧子句。

PARTITION BY列表元素的解释以 GROUP BY子句元素的方式 进行，不过它们总是简单表达式并且绝不能是输出列的名称或编号。另一个区 别是这些表达式可以包含聚集函数调用，而这在常规GROUP BY 子句中是不被允许的。它们被允许的原因是窗口是出现在分组和聚集之后的。

类似地，ORDER BY列表元素的解释也以语句级 ORDER BY子句元素的方式进行， 不过该表达式总是被当做简单表达式并且绝不会是输出列的名称或编号。

可选的frame\_clause为依赖帧的窗口函数 定义窗口帧（并非所有窗口函数都依赖于帧）。窗口帧是查询中 每一样（称为当前行）的相关行的集合。 frame\_clause可以是

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
```



---

```
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

之一，其中frame\_start和frame\_end可以是

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

之一，并且frame\_exclusion可以是

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

之一。如果省略frame\_end，它会被默认为CURRENT ROW。限制是：frame\_start不能是UNBOUNDED FOLLOWING，frame\_end不能是UNBOUNDED PRECEDING，并且frame\_end的选择在上面of frame\_start以及frame\_end选项的列表中不能早于frame\_start的选择 — 例如RANGE BETWEEN CURRENT ROW AND offset PRECEDING是不被允许的。

默认的帧选项是RANGE UNBOUNDED PRECEDING，它和RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW相同。它把帧设置为从分区开始直到当前行的最后一个平级行（被该窗口的ORDER BY子句认为等价于当前行的行，如果没有ORDER BY则所有的行都是平级的）。通常，UNBOUNDED PRECEDING表示从分区第一行开始的帧，类似地UNBOUNDED FOLLOWING表示以分区最后一行结束的帧，不论是处于RANGE、ROWS或者GROUPS模式中。在ROWS模式中，CURRENT ROW表示以当前行开始或者结束的帧。而在RANGE或者GROUPS模式中它表示当前行在ORDER BY排序中的第一个或者最后一个平级行开始或者结束的帧。offset PRECEDING和offset FOLLOWING选项的含义会随着帧模式而变化。在ROWS模式中，offset是一个整数，表示帧开始或者结束于当前行之前或者之后的那么多行处。在GROUPS模式中，offset是一个整数，表示真开始或者结束于当前行的平级组之前或者之后那么多个平级组处，其中平级组是一组根据窗口的ORDER BY子句等效的行。在RANGE模式中，offset选项的使用要求在窗口定义中正好有一个ORDER BY列。那么该帧包含的行的排序列值不超过offset且小于（对于PRECEDING）或者大于（对于FOLLOWING）当前行的排序列值。在这些情况中，offset表达式的数据类型取决于排序列的数据类型。对于数字排序列，它通常与排序列是相同类型，但对于datetime类型的排序列它是interval。在所有这些情况中，offset的值必须是非空和非负。此外，虽然offset并非必须是简单常量，但它不能包含变量、聚集函数或者窗口函数。

frame\_exclusion选项允许从帧中排除当前行周围的行，即便根据帧的起始选项来说它们应该被包含在帧中。EXCLUDE CURRENT ROW把当前行从帧中排除。EXCLUDE GROUP把当前行和它在排序上的平级行从帧中排除。EXCLUDE TIES从帧中排除当前行的任何平级行，但是不排除当前行本身。EXCLUDE NO OTHERS只是明确地指定不排除当前行或其平级行的默认行为。

注意，如果ORDER BY排序无法把行唯一地排序，则ROWS模式可能产生不可预测的结果。RANGE以及GROUPS模式的目的是确保在ORDER BY顺序中平等的行被同样对待：一个给定平级组中的所有行将在一个帧中或者被从帧中排除。

WINDOW子句的目的在于指定出现在查询的SELECT list或ORDER BY子句中的窗口函数的行为。这些函数可以在它们的OVER子句中用名称引用WINDOW子句项。不过，WINDOW子句项不是必须被引用。如果在查询中没有用到它，它会被简单地忽略。可以使用根本没有任何WINDOW子句的窗口函数，因为窗口函数调用可以直接在其OVER子句中指定它的窗口定义。不过，当多个窗口函数都需要相同的窗口定义时，WINDOW子句能够减少输入。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE和FOR KEY SHARE不能和WINDOW一起被指定。

窗口函数的详细描述在 Section 3.5 Section 4.2.8以及 Section 7.2.5中。

## SELECT 列表

SELECT列表（位于关键词 SELECT和FROM之间）指定构成 SELECT语句输出行的表达式。这些表达式可以（并且通常确实会）引用FROM子句中计算得到的列。

正如在表中一样，SELECT的每一个输出列都有一个名称。在一个简单的SELECT中，这个名称只是被用来标记要显示的列，但是当SELECT是一个大型查询的一个子查询时，大型查询会把该名称看做子查询产生的虚表的列名。要指定用于输出列的名称，在该列的表达式后面写上 AS output\_name（你可以省略AS，但只能在期望的输出名称不匹配任何 LightDB关键词（见Appendix B时省略。为了避免和未来增加的关键词冲突，推荐总是写上AS或者用双引号引用输出名称）。如果你不指定列名，LightDB会自动选择一个名称。如果列的表达式是一个简单的列引用，那么被选择的名称就和该列的名称相同。在使用函数或者类型名称的更复杂的情况下，系统可能会生成诸如 ?column?之类的名称。

然而，显示的列名将被转换成小写，即便特地使用 AS 来指定其别名，除非使用双引号或反引号引用它们。但是如果在mysql模式下，即创建数据库时指定 lightdb\_syntax\_compatible\_type mysql 同时设置 lightdb\_sql\_mode 包含 'uppercase\_identifier' 参数，那么可保证大小写不变。特别地，'uppercase\_identifier' 选项建议在 lightdb.conf 中设置而不是在会话中设置，因为这个参数最好设置后就不做改动。缺少上述条件中的任意一项都会使这项特性失效。为实现这个特性，达成与mysql一致的效果，查询时列名与存储文件内列名进行匹配时将忽略大小写，比如查询时列名 'f0o' 将认为与磁盘中存储的 'FoO' 相等。

与mysql模式类似但略有不同的是，oracle模式下guc参数 'lightdb\_oracle\_sql\_mode' 可以在会话中配置，取值为 'show\_identifier\_uppercase' 或 ''。如果设置为 'show\_identifier\_uppercase'，则执行一个select语句返回的结果列名默认显示大写。用户只有通过使用双引号包含列名才能使得大小写不变。设置 'lightdb\_oracle\_sql\_mode' 为 ''，那么该功能会被禁用并恢复默认模式。

一个输出列的名称可以被用来在ORDER BY以及 GROUP BY子句中引用该列的值，但是不能用于WHERE和HAVING子句（在其中必须写出表达式）。

可以在输出列表中写\*来取代表达式，它是被选中行的所有列的一种简写方式。还可以写 table\_name.\*，它是只来自那个表的所有列的简写形式。在这些情况中无法用 AS指定新的名称，输出行的名称将与表列的名称相同。

根据 SQL 标准，输出列表中的表达式应该在应用DISTINCT、ORDER BY或者LIMIT之前计算。在使用DISTINCT时显然必须这样做，否则就无法搞清楚到底在区分什么值。不过，在很多情况下如果先计算ORDER BY和LIMIT再计算输出表达式会很方便，特别是如果输出列表中包含任何 volatile 函数或者代价昂贵的函数时尤其如此。通过这种行为，函数计算的顺序更加直观并且对于从未出现在输出中的行将不会进行计算。只要输出表达式没有被DISTINCT、ORDER BY或者GROUP BY引用，LightDB实际将在排序和限制行数之后计算输出表达式（作为一个反例，SELECT f(x) FROM tab ORDER BY 1显然必须在排序之前计算f(x)）。包含有集合返回函数的输出表达式实际是在排序之后和限制行数之前被计算，这样LIMIT才能切断来自集合返回函数的输出。

### Note

9.6 版本之前的LightDB不对执行输出表达式、排序、限制行数的时间顺序做任何保证，那将取决于被选中的查询计划的形式。

## DISTINCT 子句

如果指定了SELECT DISTINCT，所有重复的行会被从结果集中移除（为每一组重复的行保留一行）。SELECT ALL则指定相反的行为：所有行都会被保留，这也是默认情况。

SELECT DISTINCT ON ( expression [, ...] ) 只保留在给定表达式上计算相等的行集合中的第一行。DISTINCT ON表达式使用和 ORDER BY相同的规则（见上文）解释。注意，除非用 ORDER BY来确保所期望的行出现在第一位，每一个集合的“第一行”是不可预测的。例如：

```
SELECT DISTINCT ON (location) location, time, report
  FROM weather_reports
  ORDER BY location, time DESC;
```

为每个地点检索最近的天气报告。但是如果我们不使用 ORDER BY来强制对每个地点的时间值进行降序排序，我们为每个地点得到的报告的时间可能是无法预测的。

DISTINCT ON表达式必须匹配最左边的 ORDER BY表达式。ORDER BY子句通常 将包含额外的表达式，这些额外的表达式用于决定在每一个 DISTINCT ON分组内行的优先级。

DISTINCT支持与CONNECT BY一同使用。例如：

```
select distinct test_areal.id, test_areal.name, test_areal.pid
  from test_areal join test_area2
  on test_areal.id = test_area2.id
  connect by level < 3 order by test_areal.id;
```

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE和FOR KEY SHARE不能和 DISTINCT一起使用。

## UNION 子句

UNION子句具有下面的形式：

```
select_statement UNION [ ALL | DISTINCT ] select_statement
```

select\_statement 是任何没有ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE和FOR KEY SHARE子句的 SELECT语句（如果子表达式被包围在圆括号内，ORDER BY和LIMIT可以被附着到其上。如果没有 圆括号，这些子句将被应用到UNION的结果而不是右手边 的表达式上）。

UNION操作符计算所涉及的 SELECT语句所返回的行的并集。如果一行 至少出现在两个结果集中的一个内，它就会在并集中。作为 UNION两个操作数的 SELECT语句必须产生相同数量的列并且 对应位置上的列必须具有兼容的数据类型。

UNION的结果不会包含重复行，除非指定了 ALL选项。ALL会阻止消除重复（因此，UNION ALL通常显著地快于UNION，尽量使用ALL）。可以写DISTINCT来 显式地指定消除重复行的行为。

除非用圆括号指定计算顺序， 同一个SELECT语句中的多个 UNION操作符会从左至右计算。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE和 FOR KEY SHARE不能用于UNION结果或者 UNION的任何输入。

## INTERSECT 子句

INTERSECT子句具有下面的形式：

```
select_statement INTERSECT [ ALL | DISTINCT ] select_statement
```

select\_statement 是任何没有ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE以及FOR KEY SHARE子句的 SELECT语句。

INTERSECT操作符计算所涉及的 SELECT语句返回的行的交集。如果一行同时出现在两个结果集中，它就在交集中。

INTERSECT的结果不会包含重复行，除非指定了 ALL选项。如果有ALL，一个在左表中有  $m$ 次重复并且在右表中有  $n$ 次重复的行将会在结果集中出现  $\min(m, n)$ 次。DISTINCT可以写DISTINCT来显式地指定消除重复行的行为。

除非用圆括号指定计算顺序，同一个SELECT语句中的多个 INTERSECT操作符会从左至右计算。INTERSECT的优先级比 UNION更高。也就是说，A UNION B INTERSECT C将被读成A UNION (B INTERSECT C)。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE和 FOR KEY SHARE不能用于INTERSECT结果或者 INTERSECT的任何输入。

## EXCEPT 子句

EXCEPT子句具有下面的形式：

```
select_statement EXCEPT [ ALL | DISTINCT ] select_statement
```

select\_statement 是任何没有ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE以及FOR KEY SHARE子句的 SELECT语句。

EXCEPT操作符计算位于左 SELECT语句的结果中但不在右 SELECT语句结果中的行集合。

EXCEPT的结果不会包含重复行，除非指定了 ALL选项。如果有ALL，一个在左表中有  $m$ 次重复并且在右表中有  $n$ 次重复的行将会在结果集中出现  $\max(m-n, 0)$ 次。DISTINCT可以写DISTINCT来显式地指定消除重复行的行为。

除非用圆括号指定计算顺序，同一个SELECT语句中的多个 EXCEPT操作符会从左至右计算。EXCEPT的优先级与 UNION相同。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE和 FOR KEY SHARE不能用于EXCEPT结果或者 EXCEPT的任何输入。

## ORDER BY 子句

可选的ORDER BY子句的形式如下：

```
ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ]
[, ...]
```

ORDER BY子句导致结果行被按照指定的表达式排序。如果两行按照最左边的表达式是相等的，则会根据下一个表达式比较它们，依次类推。如果按照所有指定的表达式它们都是相等的，则它们被返回的顺序取决于实现。

每一个expression 可以是输出列（SELECT列表项）的名称或者序号，它也可以是由输入列值构成的任意表达式。

序号指的是输出列的顺序（从左至右）位置。这种特性可以为不具有唯一名称的列定义一个顺序。这不是绝对必要的，因为总是可以使用 AS子句为输出列赋予一个名称。

也可以在ORDER BY子句中使用任意表达式，包括没有出现在SELECT输出列表中的列。因此，下面的语句是合法的：

```
SELECT name FROM distributors ORDER BY code;
```

这种特性的一个限制是一个应用在UNION、 INTERSECT或EXCEPT子句结果上的 ORDER BY只能指定输出列名称或序号，但不能指定表达式。

如果一个ORDER BY表达式是一个既匹配输出列名称又匹配 输入列名称的简单名称，ORDER BY将它解读成输出列名称。这与在同样情况下GROUP BY会做出的选择相反。这种不一致是为了与 SQL 标准兼容。

可以为ORDER BY子句中的任何表达式之后增加关键词 ASC（上升）DESC（下降）。如果没有指定，ASC被假定为默认值。或者，可以在USING 子句中指定一个特定的排序操作符名称。一个排序操作符必须是某个 B-树操作符族的小于或者大于成员。ASC通常等价于 USING <而DESC通常等价于 USING >（但是一种用户定义数据类型的创建者可以 准确地定义默认排序顺序是什么，并且它可能会对应于其他名称的操作符）。

如果指定NULLS LAST，空值会排在非空值之后；如果指定 NULLS FIRST，空值会排在非空值之前。如果都没有指定，在指定或者隐含ASC时的默认行为是NULLS LAST，而指定或者隐含DESC时的默认行为是 NULLS FIRST（因此，默认行为是空值大于非空值）。当指定USING时，默认的空值顺序取决于该操作符是否为 小于或者大于操作符。

注意顺序选项只应用到它们所跟随的表达式上。例如 ORDER BY x, y DESC和 ORDER BY x DESC, y DESC是不同的。

字符串数据会被根据引用到被排序列上的排序规则排序。根据需要可以通过在 expression中包括一个 COLLATE子句来覆盖，例如 ORDER BY mycolumn COLLATE "en\_US"。更多信息请见 Section 4.2.1和 Section 21.2

## LIMIT 子句

LIMIT子句由两个独立的子句构成：

```
LIMIT { count | ALL }
OFFSET start
```

参数count指定要返回 的最大行数，而start 指定在返回行之前要跳过的行数。在两者都被指定时，在开始计算要返回的 count行之前会跳过 start行。

如果count表达式计算 为 NULL，它会被当成LIMIT ALL，即没有限制。如果 start计算为 NULL，它会被当作OFFSET 0。

SQL:2008 引入了一种不同的语法来达到相同的结果， LightDB也支持它：

```
OFFSET start { ROW | ROWS }
FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES }
```

在这种语法中，标准要求start或count是一个文本常量、一个参数或者一个变量名。而作为一种LightDB的扩展，还允许其他的表达式，但通常需要被封闭在圆括号中以避免歧义。如果在一个FETCH子句中省略count，它的默认值为 1。 WITH TIES选项用于根据ORDER BY子句返回与结果集中最后一个位置相关的任何附加行；ORDER BY在这种情况下是强制性的，并且SKIP LOCKED是不被允许的。 ROW和ROWS以及FIRST和NEXT是噪声，它们不影响这些子句的效果。 根据标准，如果都存在，OFFSET子句必须出现在FETCH子句之前。 但是LightDB更宽松，它允许两种顺序。

在使用LIMIT时，用一个ORDER BY子句把 结果行约束到一个唯一顺序是个好办法。否则你讲得到该查询结果行的 一个不可预测的子集 — 你可能要求从第 10 到第 20 行，但是在 什么顺序下的第 10 到第 20 呢？除非指定ORDER BY，你 是不知道顺序的。

查询规划器在生成一个查询计划时会考虑LIMIT，因此 根据你使用的LIMIT和OFFSET，你很可能 得到不同的计划（得到不同的行序）。所以，使用不同的 LIMIT/OFFSET值来选择一个查询结果的 不同子集将会给出不一致的结果，除非你 用ORDER BY强制一种可预测的结果顺

序。这不是一个缺陷，它是 SQL 不承诺以任何特定顺序（除非使用 ORDER BY来约束顺序）给出一个查询结果这一事实造成的必然后果。

如果没有一个ORDER BY来强制选择一个确定的子集，重复执行同样的LIMIT查询甚至可能会返回一个表中行的不同子集。同样，这也不是一种缺陷，再这样一种情况下也无法保证结果的确定性。

## 锁定子句

FOR UPDATE、FOR NO KEY UPDATE、FOR SHARE和FOR KEY SHARE 是锁定子句，它们影响SELECT把行从表中取得时如何对它们加锁。

锁定子句的一般形式：

```
FOR lock_strength [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ]
```

其中lock\_strength可以是

```
UPDATE
NO KEY UPDATE
SHARE
KEY SHARE
```

更多关于每一种行级锁模式的信息可见 Section 13.3.2

为了防止该操作等待其他事务提交，可使用NOWAIT或者SKIP LOCKED选项。使用NOWAIT时，如果选中的行不能被立即锁定，该语句会报告错误而不是等待。使用 SKIP LOCKED时，无法被立即锁定的任何选中行都会被跳过。跳过已锁定行会提供数据的一个不一致的视图，因此这不适合于一般目的的工作，但是可以被用来避免多个用户访问一个类似队列的表时出现锁竞争。注意NOWAIT和SKIP LOCKED只适合行级锁——所要求的ROW SHARE表级锁仍然会以常规的方式（见Chapter 13）取得。如果想要不等待的表级锁，你可以先使用带NOWAIT的LOCK。

如果在一个锁定子句中提到了特定的表，则只有来自于那些表的行会被锁定，任何SELECT中用到的其他表还是被简单地照常读取。一个没有表列表的锁定子句会影响该语句中用到的所有表。如果一个锁定子句被应用到一个视图或者子查询，它会影响在该视图或子查询中用到的所有表。不过，这些子句不适用于主查询引用的WITH查询。如果你希望在一个WITH查询中发生行锁定，应该在该WITH查询内指定一个锁定子句。

如果有必要对不同的表指定不同的锁定行为，可以写多个锁定子句。如果同一个表在多于一个锁定子句中被提到（或者被隐式的影响到），那么会按照所指定的最强的锁定行为来处理它。类似地，如果在任何影响一个表的子句中指定了NOWAIT，就会按照NOWAIT的行为来处理该表。否则如果SKIP LOCKED在任何影响该表的子句中被指定，该表就会被按照SKIP LOCKED来处理。

如果被返回的行无法清晰地与表中的行保持一致，则不能使用锁定子句。例如锁定子句不能与聚集一起使用。

当一个锁定子句出现在一个SELECT查询的顶层时，被锁定的行正好就是该查询返回的行。在连接查询的情况下，被锁定的行是那些对返回的连接行有贡献的行。此外，自该查询的快照起满足查询条件的行将被锁定，如果它们在该快照后被更新并且不再满足查询条件，它们将不会被返回。如果使用了LIMIT，只要已经返回的行数满足了限制，锁定就会停止（但注意被OFFSET跳过的行将被锁定）。类似地，如果在一个游标的查询中使用锁定子句，只有被该游标实际取出或者跳过的行才将被锁定。

当一个锁定子句出现在一个子-SELECT中时，被锁定的行是那些该子查询返回给外层查询的行。这些被锁定的行的数量可能比从子查询自身的角度看到的要少，因为来自外层查询的条件可能会被用来优化子查询的执行。例如：

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

将只锁定具有col1 = 5的行（虽然在子查询中并没有写上 该条件）。

### Caution

一个运行在READ COMMITTED事务隔离级别并且使用ORDER BY和锁定子句的SELECT命令有可能返回无序的行。这是因为ORDER BY会被首先应用。该命令对结果排序，但是可能接着在尝试获得一个或者多个行上的锁时阻塞。一旦SELECT解除阻塞，某些排序列值可能已经被修改，从而导致那些行变成无序的（尽管它们根据原始列值是有序的）。根据需要，可以通过在子查询中放置 FOR UPDATE/SHARE来解决之一问题，例如

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY
column1;
```

注意这将导致锁定mytable的所有行，而顶层的 FOR UPDATE只会锁定实际被返回的行。这可能会导致显著的性能差异，特别是把ORDER BY与LIMIT或者其他限制组合使用时。因此只有在并发更新排序列并且要求严格的排序结果时才推荐使用这种技术。

在REPEATABLE READ或者SERIALIZABLE 事务隔离级别上这可能导致一个序列化失败（SQLSTATE 是'40001'），因此在这些隔离级别下不可能收到无序行。

## TABLE 命令

命令

TABLE name

等价于

```
SELECT * FROM name
```

它可以被用作一个顶层命令，或者用在复杂查询中以节省空间。只有 WITH、UNION、INTERSECT、EXCEPT、ORDER BY、LIMIT、OFFSET、FETCH以及FOR锁定子句可以用于TABLE。不能使用WHERE子句和任何形式的聚集。

## 示例

mysql模式下显示列名不变：

```
CREATE DATABASE films_distributors WITH lightdb_syntax_compatible_type mysql;
\c films_distributors;
-- it not suggest to set 'uppercase_identifier' on session to enable that as
-- 'set lightdb_sql_mode to uppercase_identifier;'. users should specify this
-- option in lightdb.conf instead, which follows by the current value of
-- 'lightdb_sql_mode' split with ',''. Excuting 'lt_ctl --reload' to reload this
-- rule and don't modify it as far as possible.
```

```
SELECT Title as tiTLe, Did DiD, name, DAte_pROd, films.KIND
FROM films;
```

tiTLe	DiD	name	DAte_pROd	KIND
The Third Man	101	British Lion	1949-12-23	Drama
The African Queen	101	British Lion	1951-08-11	Romantic
...				

示例oracle模式下显示大写列名

-- 需确保处于oracle模式

```
show lightdb_dblevel_syntax_compatible_type;
lightdb_dblevel_syntax_compatible_type
```

Oracle

(1 row)

```
set lightdb_oracle_sql_mode to 'show_identifier_uppercase';
```

```
show lightdb_oracle_sql_mode;
```

```
lightdb_oracle_sql_mode
```

```
show_identifier_uppercase
```

(1 row)

```
create table a(foo varchar(10));
```

-- 下述内部查询不会转换成大写

```
select "foo" from (select foo as Foo from a) as temp;
```

-- 显示

```
foo
```

```
foo
```

(0 rows)

```
select foo from (select foo as Foo from a) as temp;
```

-- 显示

```
F00
```

```
F00
```

(0 rows)

```
select "fo0" foo from (select foo as Foo from a) as temp;
```

-- 失败,

"fo0"列不存在

```
ERROR: column "fo0" does not exist
```

```
LINE 1: select "fo0" foo from (select foo as Foo from a) as temp;
```

把表films与表 distributors连接:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
```

```
FROM distributors d, films f
```

```
WHERE f.did = d.did
```

title	did	name	date_prod	kind
The Third Man	101	British Lion	1949-12-23	Drama
The African Queen	101	British Lion	1951-08-11	Romantic
...				

要对所有电影的len列求和并且用 kind对结果分组:

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;
```



kind	total
Action	07:34
Comedy	02:58
Drama	14:28
Musical	06:42
Romantic	04:38

要对所有电影的len列求和、对结果按照 kind分组并且显示总长小于 5 小时的分组:

```
SELECT kind, sum(len) AS total
  FROM films
  GROUP BY kind
  HAVING sum(len) < interval '5 hours';
```

kind	total
Comedy	02:58
Romantic	04:38

下面两个例子都是根据第二列 (name) 的内容来排序结果:

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

did	name
109	20th Century Fox
110	Bavaria Atelier
101	British Lion
107	Columbia
102	Jean Luc Godard
113	Luso films
104	Mosfilm
103	Paramount
106	Toho
105	United Artists
111	Walt Disney
112	Warner Bros.
108	Westward

接下来的例子展示了如何得到表distributors和 actors的并集, 把结果限制为那些在每个表中以 字母 W 开始的行。只想要可区分的行, 因此省略了关键词 ALL。

distributors:		actors:	
did	name	id	name
108	Westward	1	Woody Allen
111	Walt Disney	2	Warren Beatty
112	Warner Bros.	3	Walter Matthau
...		...	

```
SELECT distributors.name
  FROM distributors
  WHERE distributors.name LIKE 'W%'
```

```
UNION
SELECT actors.name
   FROM actors
   WHERE actors.name LIKE 'W%';
```

```

      name
-----
Walt Disney
Walter Matthau
Warner Bros.
Warren Beatty
Westward
Woody Allen
```

这个例子展示了如何在FROM子句中使用函数， 分别使用和不使用列定义列表：

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
  SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distributors(111);
 did | name
-----+-----
 111 | Walt Disney
```

```
CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
  SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1 | f2
-----+-----
 111 | Walt Disney
```

这里是带有增加的序数列的函数的例子：

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f']) WITH ORDINALITY;
unnest | ordinality
-----+-----
 a      |          1
 b      |          2
 c      |          3
 d      |          4
 e      |          5
 f      |          6
(6 rows)
```

这个例子展示了如何使用简单的WITH子句：

```
WITH t AS (
  SELECT random() as x FROM generate_series(1, 3)
)
SELECT * FROM t
UNION ALL
SELECT * FROM t
```

x

```

-----
0.534150459803641
0.520092216785997
0.0735620250925422
0.534150459803641
0.520092216785997
0.0735620250925422

```

注意该WITH查询只被计算一次，这样我们得到的两个集合具有相同的三个随机值。

这个例子使用WITH RECURSIVE从一个只显示直接下属的表中寻找雇员 Mary 的所有下属（直接的或者间接的）以及他们的间接层数：

```

WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
    UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
)
SELECT distance, employee_name FROM employee_recursive;

```

注意这种递归查询的典型形式：一个初始条件，后面跟着 UNION，然后是查询的递归部分。要确保查询的递归部分最终将不返回任何行，否则该查询将无限循环（更多例子见Section 7.8）。

这个例子使用LATERAL为manufacturers表的每一行应用一个集合返回函数get\_product\_names()：

```

SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id) pname;

```

当前没有任何产品的制造商不会出现在结果中，因为这是一个内连接。如果我们希望把这类制造商的名称包括在结果中，我们可以：

```

SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true;

```

此示例使用 CONNECT BY 子句定义员工和管理者之间的关系，使用 LEVEL 伪列显示父行和子行，并使用 START WITH 子句为分层结构指定一个根行。

```

SELECT last_name, employee_id, manager_id, LEVEL
FROM employees
START WITH employee_id = 100
CONNECT BY PRIOR employee_id = manager_id;

```

LAST_NAME	EMPLOYEE_ID	MANAGER_ID	LEVEL
King	100		1
Cambrault	148	100	2
Bates	172	148	3
Bloom	169	148	3

SELECT

Fox	170	148	3
Kumar	173	148	3
Ozer	168	148	3
Smith	171	148	3
De Haan	102	100	2
Hunold	103	102	3
Austin	105	103	4
Ernst	104	103	4
Lorentz	107	103	4
Pataballa	106	103	4
Errazuriz	147	100	2
Ande	166	147	3
Banda	167	147	3

以下示例返回从员工 KING 到 KING 的员工（以及他们的员工）的员工姓名路径：

```
SELECT SYS_CONNECT_BY_PATH(last_name, '/') "Path"
FROM employees
START WITH last_name = 'KING'
CONNECT BY PRIOR employee_id = manager_id;
        Path
```

```
-----
/KING
/KING/JONES
/KING/BLAKE
/KING/CLARK
/KING/BLAKE/ALLEN
/KING/BLAKE/WARD
/KING/BLAKE/MARTIN
/KING/JONES/SCOTT
/KING/BLAKE/TURNER
/KING/BLAKE/JAMES
/KING/JONES/FORD
/KING/CLARK/MILLER
/KING/JONES/FORD/SMITH
/KING/JONES/SCOTT/ADAMS
```

(14 rows)

以下示例返回 110 部门中每个员工的姓氏、层次结构中该员工之上最高级别的经理、经理和员工之间的级别以及两者之间的路径：

```
SELECT last_name "Employee", CONNECT_BY_ROOT last_name "Manager",
LEVEL-1 "Pathlen", SYS_CONNECT_BY_PATH(last_name, '/') "Path"
FROM employees
WHERE LEVEL > 1 and department_id = 110
CONNECT BY PRIOR employee_id = manager_id
ORDER BY "Employee", "Manager", "Pathlen", "Path";
```

Employee	Manager	Pathlen	Path
Gietz	Higgins	1	/Higgins/Gietz
Gietz	King	3	/King/Kochhar/Higgins/Gietz
Gietz	Kochhar	2	/Kochhar/Higgins/Gietz
Higgins	King	2	/King/Kochhar/Higgins

SELECT

Higgins                      Kochhar                                      1 /Kochhar/Higgins

以下示例显示 n 层级以下的所有递归结果。

```
select * from staff_table connect by level < 2;
```

staff_id	name	manager_id
1001	Michael North	
1002	Megan Berry	1001
1003	Sarah Berry	1001
1004	Zoe Black	1001
1005	Tim James	1001
1006	Bella Tucker	1002
1007	Ryan Metcalfe	1002
1008	Max Mills	1002
1009	Benjamin Glover	1002
1010	Carolyn Henderson	1003
1011	Nicola Kelly	1003
1012	Alexandra Climo	1003
1013	Dominic King	1003
1014	Leonard Gray	1004
1015	Eric Rampling	1004
1016	Piers Paige	1007
1017	Ryan Henderson	1007
1018	Frank Tucker	1008
1019	Nathan Ferguson	1008
1020	Kevin Rampling	1008

以下示例显示员工是否是叶节点（没有下属）。

```
SELECT staff_id, name, manager_id,
SYS_CONNECT_BY_PATH (name, '/') , connect_by_isleaf
FROM staff_table
START WITH staff_id = '1001'
CONNECT BY PRIOR staff_id = manager_id;
```

staff_id	name	manager_id	sysconnectpath	connect_by_isleaf
1001	Michael North	1000	/Michael North	
		0		
1002	Megan Berry	1001	/Michael North/Megan Berry	
		0		
1003	Sarah Berry	1001	/Michael North/Sarah Berry	
		0		
1004	Zoe Black	1001	/Michael North/Zoe Black	
		0		
1005	Tim James	1001	/Michael North/Tim James	
		1		
1006	Bella Tucker	1002	/Michael North/Megan Berry/Bella	
Tucker		1		
1007	Ryan Metcalfe	1002	/Michael North/Megan Berry/Ryan	
Metcalfe		0		
1008	Max Mills	1002	/Michael North/Megan Berry/Max Mills	
		0		

1009   Benjamin Glover	1002   /Michael North/Megan Berry/Benjamin Glover
1010   Carolyn Henderson	1003   /Michael North/Sarah Berry/Carolyn Henderson
1011   Nicola Kelly	1003   /Michael North/Sarah Berry/Nicola Kelly
1012   Alexandra Climo	1003   /Michael North/Sarah Berry/Alexandra Climo
1013   Dominic King	1003   /Michael North/Sarah Berry/Dominic King
1014   Leonard Gray	1004   /Michael North/Zoe Black/Leonard Gray
1015   Eric Rampling	1004   /Michael North/Zoe Black/Eric Rampling
1016   Piers Paige Metcalf/Piers Paige	1007   /Michael North/Megan Berry/Ryan Metcalf/Piers Paige
1017   Ryan Henderson Metcalf/Ryan Henderson	1007   /Michael North/Megan Berry/Ryan Metcalf/Ryan Henderson
1018   Frank Tucker Mills/Frank Tucker	1008   /Michael North/Megan Berry/Max Mills/Frank Tucker
1019   Nathan Ferguson Mills/Nathan Ferguson	1008   /Michael North/Megan Berry/Max Mills/Nathan Ferguson
1020   Kevin Rampling Mills/Kevin Rampling	1008   /Michael North/Megan Berry/Max Mills/Kevin Rampling

这个例子说明了 MySQL 索引提示的用法。

```
-- multiple index
select * from lt_test_mysql_ddl use
index(pk_lt_test_mysql_ddl,uk_lt_test_mysql_ddl);
select * from lt_test_mysql_ddl force
index(pk_lt_test_mysql_ddl,uk_lt_test_mysql_ddl);
select * from lt_test_mysql_ddl ignore
index(pk_lt_test_mysql_ddl,uk_lt_test_mysql_ddl);

-- multiple table join
select * from lt_test_mysql_ddl a use index for order by(primary) join b
using(id);
select * from lt_test_mysql_ddl a force index for order
by(pk_lt_test_mysql_ddl) join b using(id);
select * from lt_test_mysql_ddl a ignore index for order
by(primary,pk_lt_test_mysql_ddl) join b using(id);
select * from lt_test_mysql_ddl a ignore index for order
by(pk_lt_test_mysql_ddl) join b using(id);
```

## 兼容性

当然，SELECT语句是兼容 SQL 标准的。但是也有一些扩展和缺失的特性。

### 省略的FROM子句

LightDB允许我们省略 FROM子句。一种简单的使用是计算简单表达式 的结果：

```
SELECT 2+2;

?column?
-----
         4
```

某些其他SQL数据库需要引入一个假的 单行表放在该SELECT的 FROM子句中才能做到这一点。

请注意，如果没有指定 FROM 子句，查询就不能引用任何数据库表。例如，以下查询是无效的：

```
SELECT distributors.* WHERE distributors.name = 'Westward' ;
```

## 空SELECT列表

SELECT之后的输出表达式列表可以为空， 这会产生一个零列的结果表。对 SQL 标准来说这不是合法的 语法。LightDB允许 它是为了与允许零列表保持一致。不过在使用 DISTINCT时不允许空列表。

## 省略AS关键词

在 SQL 标准中，只要新列名是一个合法的列名（就是说与任何保留关键词不同）， 就可以省略输出列名之前的可选关键词AS。 LightDB要稍微严格些：只要新列名匹配 任何关键词（保留或者非保留）就需要AS。推荐的习惯是使用 AS或者带双引号的输出列名来防止与未来增加的关键词可能的冲突。

在FROM项中，标准和 LightDB都允许省略非保留 关键词别名之前的AS。但是由于语法的歧义，这无法 用于输出列名。

## ONLY和继承

在书写ONLY时，SQL 标准要求要在表名周围加上圆括号，例如 SELECT \* FROM ONLY (tab1), ONLY (tab2) WHERE ...。LightDB 认为这些圆括号是可选的。

LightDB允许写一个拖尾的\*来 显式指定包括子表的非-ONLY行为。而标准则不允许 这样。

（这些点同等地适用于所有支持ONLY选项的 SQL 命令）。

## TABLESAMPLE子句限制

当前只在常规表和物化视图上接受TABLESAMPLE子句。 根据 SQL 标准，应该可以把它应用于任何FROM项。

## FROM中的函数调用

LightDB允许一个函数调用被直接写作 FROM列表的一个成员。在 SQL 标准中，有必要把这样一个函数 调用包裹在一个子-SELECT中。也就是说，语法 FROM func(...) alias 近似等价于 FROM LATERAL (SELECT func(...)) alias。 注意该LATERAL被认为是隐式的，这是因为标准对于 FROM中的一个UNNEST()项要求 LATERAL语义。LightDB会把 UNNEST()和其他集合返回函数同样对待。

## GROUP BY和ORDER BY可用的名字空间

在 SQL-92 标准中，一个ORDER BY子句只能使用输出 列名或者序号，而一个GROUP BY子句只能使用基于输入列名的表达式。LightDB扩展了 这两种子句以允许它们使用其他的选择（如果有歧义时还是使用标准的 解释）。LightDB也允许两种子句 指定任意表达式。注意出现在一个表达式中的名称将总是被当做输入列名而 不是输出列名。

SQL:1999 及其后的标准使用了一种略微不同的定义，它并不完全向后兼容 SQL-92。不过，在大部分的情况下，LightDB会以与 SQL:1999 相同的方式解释ORDER BY或GROUP BY表达式。

## 函数依赖

只有当一个表的主键被包括在GROUP BY列表中时，LightDB才识别函数依赖（允许从GROUP BY中省略列）。SQL 标准指定了应该要识别的额外情况。

## LIMIT和OFFSET

LIMIT和OFFSET子句是 LightDB-特有的语法，在 MySQL也被使用。SQL:2008 标准已经引入了具有相同功能的子句OFFSET ... FETCH {FIRST|NEXT} ...（如上文 LIMIT Clause中所示）。这种语法也被IBM DB2使用（Oracle编写的的应用常常使用自动生成的 rownum列来实现这些子句的效果，这在 LightDB 中是没有的）。

## FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHARE

尽管 SQL 标准中出现了FOR UPDATE，但标准只允许它作为 DECLARE CURSOR的一个选项。LightDB允许它出现在任何 SELECT查询以及子-SELECT中，但这是 一种扩展。FOR NO KEY UPDATE、FOR SHARE 以及FOR KEY SHARE变体以及NOWAIT 和SKIP LOCKED选项没有在标准中出现。

## WITH中的数据修改语句

LightDB允许把INSERT、UPDATE以及DELETE用作WITH 查询。这在 SQL 标准中是找不到的。

## 非标准子句

DISTINCT ON ( ... )是 SQL 标准的扩展。

ROWS FROM( ... )是 SQL 标准的扩展。

WITH的MATERIALIZED 和 NOT MATERIALIZED 选项是SQL标准的扩展。

## oracle (+)

可以在 where 条件中使用 '(+)' 来像 Oracle 一样指定外连接，案例如下：

```
select * from t1 a, t2 b where a.key1(+) = b.key1;
          QUERY PLAN
-----
Hash Right Join (cost=60.85..86.01 rows=2260 width=48)
  Hash Cond: (a.key1 = b.key1)
    -> Seq Scan on t1 a (cost=0.00..22.00 rows=1200 width=40)
    -> Hash (cost=32.60..32.60 rows=2260 width=8)
          -> Seq Scan on t2 b (cost=0.00..32.60 rows=2260 width=8)
(5 rows)
```

下面描述 Oracle (+) 的一些限制。

- (+) 操作符只能在 WHERE 子句中使用，并且只能作用域表或视图的字段。
- (+) 操作符必须完全匹配，中间不能有空格。oracle 中间可以有空格。

```
select * from t1 a, t2 b where a.key1( +) = b.key1;
```



```
ERROR: syntax error at or near ")"
LINE 1: explain select * from t1 a, t2 b where a.key1(+) = b.key1;
```

- 如果两个表间有多个关联条件，那么每个条件都要使用 (+)，这样才能把所有的关联条件作为 join on 的关联条件。不然，没有使用 (+) 的条件会保留在 where 中，然后会导致满足空值拒绝，最终被优化为 inner join(比如 left join 中的右表被 where 条件约束为不能为 null，则等价与 inner join)。

```
explain select * from t1 a, t2 b where a.key1(+)=b.key1 and a.key2=b.key2;
QUERY PLAN
```

```
-----
Merge Join (cost=317.01..352.19 rows=128 width=16)
  Merge Cond: ((a.key1 = b.key1) AND (a.key2 = b.key2))
    -> Sort (cost=158.51..164.16 rows=2260 width=8)
        Sort Key: a.key1, a.key2
        -> Seq Scan on t1 a (cost=0.00..32.60 rows=2260 width=8)
    -> Sort (cost=158.51..164.16 rows=2260 width=8)
        Sort Key: b.key1, b.key2
        -> Seq Scan on t2 b (cost=0.00..32.60 rows=2260 width=8)
(8 rows)
```

- (+) 操作符只能作用于字段，不能作用于表达式，但可以作用于表达式中的字段。

```
select * from t1 a, t2 b where mod(a.key1,10)(+)=b.key1;
ERROR: syntax error at or near "(+)"
LINE 1: select * from t1 a, t2 b where mod(a.key1,10)(+)=b.key1;
```

```
select * from t1 a, t2 b where mod(a.key1(+),10)=b.key1;
 key1 | key2 | key1 | key2
-----+-----+-----+-----
(0 rows)
```

- 使用了 (+) 操作符的约束不能与其他约束通过 or 组合。

```
select * from t1 a, t2 b where a.key1(+)=b.key1 or a.key2(+)=b.key2;
ERROR: Operator "(+)" is not allowed used with "OR" together
```

- 不能将任何用 (+) 操作符标记的列与子查询进行比较。

```
select * from t1 a, t2 b where a.key1(+)=(select key1 from t3);
ERROR: Operator "(+)" can not be used in outer join with SubQuery.
LINE 1: select * from t1 a, t2 b where a.key1(+)=(select key1 from t...
```

- 表不能彼此进行外关联。

```
select * from t1 a, t2 b where a.key1(+)=b.key1 and b.key2(+)=a.key2;
ERROR: Relation can't outer join with each other.
```

- (+) 作用的字段所属的表需要在当前查询块，不能位于上层的查询块。

```
select * from t1 a where exists (select * from t2 b where a.key1(+)=b.key1);
```

```
ERROR: Operator "(+)" can't specify on "a" which cannot be referenced from
this part of the query.
```

```
LINE 1: ...rom t1 a where exists (select * from t2 b where a.key1(+)=...
```

- (+) 操作符不能用于嵌套的and/or 表达式，这与 Oracle 不同，Oracle 支持，示例如下：

```
# Oracle 也不支持，因为在展开后，这实际是个 or 表达式。
```

```
select * from t1 a, t2 b where not (a.key1(+)= b.key1 and a.key2(+)=b.key2);
```

```
ERROR: Operator "(+)" can not be used in nested and/or expression.
```

```
LINE 1: select * from t1 a, t2 b where not (a.key1(+)= b.key1 and a...
```

```
# Oracle 支持 'not (a.key1(+)= b.key1 or a.key2(+)=b.key2)'
```

```
select * from t1 a, t2 b where not (a.key1(+)= b.key1 or a.key2(+)=b.key2);
```

```
ERROR: Operator "(+)" can not be used in nested and/or expression.
```

```
LINE 1: select * from t1 a, t2 b where not (a.key1(+)= b.key1 or a.k...
```

- (+) 操作符不能和 ansi join 一起用。

```
select * from t1 a join t2 b on a.key1=b.key1, t3 c, t4 d where
```

```
c.key1(+)=d.key1;
```

```
ERROR: Operator "(+)" and Join in FromClause can't be used together
```

- 一个表不能同时right join 多个表. oracle 支持这种用法.

```
select * from t1 a, t2 b, t3 c where a.key1(+)=b.key1+c.key1;
```

```
ERROR: "a" can't outer join with more than one relation
```

```
HINT: "b", "c" are outer join with "a".
```

```
select * from t1 a, t2 b, t3 c where a.key1(+)=b.key1 and a.key1(+)=c.key1;
```

```
ERROR: "a" can't outer join with more than one relation.
```

- (+) 操作符不支持指定full out join.

```
select * from t1 a, t2 b, t3 c where a.key1(+) =b.key1(+);
```

```
ERROR: Operator "(+)" can't be specified on more than one relation in one
join condition
```

```
HINT: "a", "b"...are specified Operator "(+)" in one condition.
```

## pivot

提供简单pivot使用的例子.

```
create table test123(name varchar(40),chinese int,math int, course varchar(40),
score int);
```

```
insert into test123 values('lisi',88,99,'math',99);
```

```
insert into test123 values('lisi',88,99,'chinese',88);
```

```
insert into test123 values('zhangsan',90,100,'chinese',90);
```

```
insert into test123 values('zhangsan',90,100,'math',100);
```

```
select * from test123 pivot (sum(score) for course in('chinese','math'));
```

```
name | chinese | math | 'chinese' | 'math'
```

```
-----+-----+-----+-----+-----
```

SELECT

---

lisi	88	99	88	99
lisi	89	100		99
lisi	100	70	100	
zhangsan	76	89	99	
zhangsan	90	100	90	100
zhangsan	95	85		100

(6 rows)

drop table test123;

---

# SELECT INTO

SELECT INTO — 从一个查询的结果定义一个新表

## Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

## 描述

SELECT INTO创建一个新表并且用一个查询 计算得到的数据填充它。这些数据不会像普通的SELECT那样被返回给客户端。新表的列具有 和SELECT的输出列相关的名称和数据类型。

## 参数

TEMPORARY or TEMP

如果被指定, 该表被创建为一个临时表。详见 CREATE TABLE。

UNLOGGED

如果被指定, 该表被创建为一个不做日志的表。详见 CREATE TABLE。

new\_table

要创建的表的名字 (可以是模式限定的)。

所有其他参数在SELECT中有详细描述。

## 注解

CREATE TABLE AS在功能上与 SELECT INTO相似。CREATE TABLE AS 是被推荐的语法, 因为这种形式的SELECT INTO在ECPG 或PL/pgSQL中不可用, 因为它们对 INTO子句的解释不同。此外, CREATE TABLE AS提供的功能是 SELECT INTO的超集。

与CREATE TABLE AS相比, SELECT INTO 不允许指定属性, 就像带有USING method的表访问方法或者带有TABLESPACE tablespace\_name的表的表空间。 如果必要, 使用CREATE TABLE AS。因此, 为新表选择默认表访问方法。 详细信息请参阅default\_table\_access\_method。

## 示例

创建一个只由来自films的最近项构成的 新表films\_recent:

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

## 兼容性

SQL 标准使用SELECT INTO表示把值选择到一个宿主程序的标量变量中，而不是创建一个新表。这实际上就是ECPG（见Chapter 34）和PL/pgSQL（见Chapter 41）中的用法。LightDB使用SELECT INTO的来表示表创建是有历史原因的。一些其它SQL实现也以此方式使用 SELECT INTO（但是大多数SQL实现支持CREATE TABLE AS）。除了这样的兼容性考虑之外，最好在新代码中使用CREATE TABLE AS。

## 另见

CREATE TABLE AS

---

# SET

SET — 更改一个运行时参数

## Synopsis

```
SET [ SESSION | LOCAL ] configuration_parameter { TO | = } { value | 'value' |
  DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT }
```

## 描述

SET命令更改运行时配置参数。很多 Chapter 1中列出的参数可以用 SET即时更改（但是有些需要超级用户 特权才能更改，并且还有一些在服务器或者会话启动之后不能被更改）。SET只影响当前会话所使用的值。

如果在一个事务内发出SET（或者等效的SET SESSION）而该事务后来 中止，在该事务被回滚时SET命令的效果会 消失。一旦所在的事务被提交，这些效果将会持续到会话结束（除非被另一个SET所覆盖）。

SET LOCAL的效果只持续到当前事务结束， 不管事务是否被提交。一种特殊情况是在一个事务内 SET后面跟着 SET LOCAL： SET LOCAL值将会在该事务结束前一直可见， 但是之后（如果该事务被提交）SET值将会生效。

SET或SET LOCAL 的效果也会因为回滚到早于它们的保存点而消失。

如果在一个函数内使用SET LOCAL并且该函数 还有对同一变量的SET选项（见 CREATE FUNCTION），在函数退出时 SET LOCAL命令的效果会消失。也就是说，该 函数被调用时的值会被恢复。这允许用 SET LOCAL在函数内动态地或者重复地更改 一个参数，同时仍然能便利地使用SET选项来保存以及恢复调用 者的值。不过，一个常规的SET命令会覆盖它所在的任何函 数的SET选项，除非回滚，它的效果将一直保持。

### Note

在LightDB 版本 8.0 到 8.2 中，一个SET LOCAL的效果会因为释放较早的 保存点或者成功地从一个PL/pgSQL异常块 中退出而被取消。这种行为已经被更改，因为它被认为不直观。

## 参数

SESSION

指定该命令对当前会话有效（这是默认值）。

LOCAL

指定该命令只对当前事务有效。在COMMIT或者 ROLLBACK之后，会话级别的设置会再次生效。 在事务块外部发出这个参数会发出一个警告并且不会有效果。

configuration\_parameter

一个可设置运行时参数的名称。可用的参数被记录在 Chapter 1和下文。

value

参数的新值。根据特定的参数，值可以被指定为字符串常量、标识符、数字或者以上构成的逗号分隔列表。写DEFAULT 可以指定把该参数重置成它的默认值（也就是说在当前会话中还没有 执行SET命令时它具有的值）。

除了在Chapter 1中记录的配置参数， 还有一些参数只能用SET命令设置 或者具有特殊的语法：

SCHEMA

SET SCHEMA 'value' 是 SET search\_path TO value的一个别名。 使用这种语法只能指定一个模式。

NAMES

SET NAMES value是 SET client\_encoding TO value的一个别名。

SEED

为随机数生成器（函数random）设置 一个内部种子。允许的值是 -1 和 1 之间的浮点数，它会被乘上  $2^{31}-1$ 。

也可以通过调用函数setseed来设置种子：

```
SELECT setseed(value);
```

TIME ZONE

SET TIME ZONE value是 SET timezone TO value的一个别名。语法SET TIME ZONE允许用于时区指定的特殊语法。这里是合法值的例子：

'PST8PDT'

加州伯克利的时区。

'Europe/Rome'

意大利的时区。

-7

UTC 以西 7 小时的时区（等效于 PDT）。正值则是 UTC 以东。

INTERVAL '-08:00' HOUR TO MINUTE

UTC 以西 8 小时的时区（等效于 PST）。

LOCAL

DEFAULT

把时区设置为你的本地时区（也就是说服务器的timezone默认值）。

以数字或区间给出的时区设置在内部被翻译成 POSIX 时区语法。 例如，在SET TIME ZONE -7之后， SHOW TIME ZONE将会报告 <-07>+07。

有关时区的更多信息可见Section 8.5.3

## 注解

函数set\_config提供了等效的功能，见 Section 9.27.1 此外，可以更新 pg\_settings 系统视图来执行与SET等效的工作。

## 示例

设置模式搜索路径:

```
SET search_path TO my_schema, public;
```

把日期风格设置为传统 POSTGRES的 “日在月之前” 的输入习惯:

```
SET datestyle TO postgres, dmy;
```

设置时区为加州伯克利:

```
SET TIME ZONE 'PST8PDT';
```

设置时区为意大利:

```
SET TIME ZONE 'Europe/Rome';
```

## 兼容性

SET TIME ZONE扩展了 SQL 标准定义的语法。标准 只允许数字的时区偏移量而 LightDB允许更灵活的时区说明。 所有其他SET特性都是 LightDB扩展。

## 另见

RESET, SHOW



---

# SET CONSTRAINTS

SET CONSTRAINTS — 为当前事务设置约束检查时机

## Synopsis

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

## 描述

SET CONSTRAINTS设置当前事务内约束检查的行为。IMMEDIATE约束在每个语句结束时被检查。DEFERRED约束直到事务提交时才被检查。每个约束都有自己的IMMEDIATE或DEFERRED模式。

在创建时，一个约束会被给定三种特性之一：DEFERRABLE INITIALLY DEFERRED、DEFERRABLE INITIALLY IMMEDIATE或者NOT DEFERRABLE。第三类总是IMMEDIATE并且不会受到SET CONSTRAINTS命令的影响。前两类在每个事务开始时都处于指定的模式，但是它们的行为可以在一个事务内用SET CONSTRAINTS更改。

带有一个约束名称列表的SET CONSTRAINTS只更改那些约束（都必须是可延迟的）的模式。每一个约束名称都可以是模式限定的。如果没有指定模式名称，则当前的模式搜索路径将被用来寻找第一个匹配的名称。SET CONSTRAINTS ALL更改所有可延迟约束的模式。

当SET CONSTRAINTS把一个约束的模式从DEFERRED改成IMMEDIATE时，新模式会有追溯效果：任何还没有解决的数据修改（本来会在事务结束时被检查）会转而在SET CONSTRAINTS命令的执行期间被检查。如果任何这种约束被违背，SET CONSTRAINTS将会失败（并且不会改变该约束模式）。这样，SET CONSTRAINTS可以被用来在一个事务中的特定点强制进行约束检查。

当前，只有UNIQUE、PRIMARY KEY、REFERENCES（外键）以及EXCLUDE约束受到这个设置的影响。NOT NULL以及CHECK约束总是在一行被插入或修改时立即检查（不是在语句结束时）。没有被声明为DEFERRABLE的唯一和排除约束也会被立刻检查。

被声明为“约束触发器”的触发器的引发也受到这个设置的控制 — 它们会在相关约束被检查的同时被引发。

## 注解

因为LightDB并不要求约束名称在模式内唯一（但是在表内要求唯一），可能有多于一个约束匹配指定的约束名称。在这种情况下SET CONSTRAINTS将会在所有的匹配上操作。对于一个非模式限定的名称，一旦在搜索路径中的某个模式中发现一个或者多个匹配，路径中后面的模式将不会被搜索。

这个命令只修改当前事务内约束的行为。在事务块外部发出这个命令会产生一个警告并且也不会有任何效果。

## 兼容性

这个命令符合SQL标准中定义的行为，但有一点限制：在LightDB中，它不会应用在NOT NULL和CHECK约束上。还有，LightDB会立刻检查非可延迟的唯一约束，而不是按照标准建议在语句结束时检查。

---

# SET ROLE

SET ROLE — 设置当前会话的当前用户标识符

## Synopsis

```
SET [ SESSION | LOCAL ] ROLE role_name
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

## 描述

这个命令把当前 SQL 会话的当前用户标识符设置为 `role_name`。角色名可以写成一个标识符或者一个字符串。在 SET ROLE 之后，对 SQL 命令的权限检查时就好像该角色就是原先登录的角色一样。

当前会话用户必须是指定的角色 `role_name` 的一个成员（如果会话用户是一个超级用户，则可以选择任何角色）。

SESSION 和 LOCAL 修饰符发挥的作用和常规的 SET 命令一样。

SET ROLE NONE 设置当前用户标识符为当前会话用户标识符，由 `session_user` 返回。RESET ROLE 设置当前用户标识符为由 `command-line options`、ALTER ROLE，或 ALTER DATABASE 指定的连接时间设定，如果存在这样的设置的话。否则，RESET ROLE 设置当前用户标识符到当前会话用户标识符。这些形式可以由任何用户执行。

## 注解

使用这个命令可以增加特权或者限制特权。如果会话用户角色具有 INHERIT 属性，则它会自动具有它能 SET ROLE 到的所有角色的全部特权。在这种情况下 SET ROLE 实际会删除所有直接分配给会话用户的特权以及分配给会话用户作为其成员的其他角色的特权，只留下所提及角色可用的特权。换句话说，如果会话用户没有 NOINHERIT 属性，SET ROLE 会删除直接分配给会话用户的特权而得到所提及角色可用的特权。

特别地，当一个超级用户选择 SET ROLE 到一个非超级用户角色时，它们会丢失其超级用户特权。

SET ROLE 的效果堪比 SET SESSION AUTHORIZATION，但是涉及的特权检查完全不同。还有，SET SESSION AUTHORIZATION 决定后来的 SET ROLE 命令可以使用哪些角色，不过用 SET ROLE 更改角色并不会改变后续 SET ROLE 能够使用的角色集。

SET ROLE 不会处理角色的 ALTER ROLE 设置指定的会话变量。这只在登录期间发生。

SET ROLE 不能在一个 SECURITY DEFINER 函数中使用。

## 示例

```
SELECT SESSION_USER, CURRENT_USER;
```

```
session_user | current_user
-----+-----
peter       | peter
```

```
SET ROLE 'paul';
```

```
SELECT SESSION_USER, CURRENT_USER;
```

```
session_user | current_user  
-----+-----  
peter       | paul
```

## 兼容性

LightDB允许标识符语法（"rolename"），而 SQL 标准要求角色名被写成字符串。SQL 不允许在事务中使用这个命令，而 LightDB并不做此限制，因为并没有原因需要这样做。和RESET语法一样，SESSION和 LOCAL修饰符是一种 LightDB扩展。

## 另见

SET SESSION AUTHORIZATION

---

# SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION — 设置当前会话的会话用户标识符和当前用户标识符

## Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION user_name
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

## 描述

这个命令把当前 SQL 会话的会话用户标识符和当前用户标识符设置为 `user_name`。用户名可以被写成一个标识符或者一个字符串。例如，可以使用这个命令临时成为一个无特权用户并且稍后切换回来成为一个超级用户。

会话用户标识符初始时被设置为客户端提供的（可能已认证的）用户名。当前用户标识符通常等于会话用户标识符，但是可能在 SECURITY DEFINER函数和类似机制的环境中临时更改。也可以用SET ROLE更改它。当前用户标识符与 权限检查相关。

会话用户标识符只能在初始会话用户 已认证用户具有超级用户特权时被更改。否则，只有该命令指定已认证用户名时才会被接受。

SESSION和LOCAL修饰符发挥的作用和常规 SET命令一样。

DEFAULT和RESET形式把会话用户标识符和 当前用户标识符重置为初始的已认证用户名。这些形式可以由任何用户执行。

## 注解

SET SESSION AUTHORIZATION不能在一个 SECURITY DEFINER函数中使用。

## 示例

```
SELECT SESSION_USER, CURRENT_USER;
```

```
session_user | current_user
-----+-----
peter        | peter
```

```
SET SESSION AUTHORIZATION 'paul';
```

```
SELECT SESSION_USER, CURRENT_USER;
```

```
session_user | current_user
-----+-----
paul         | paul
```

## 兼容性

SQL 标准允许一些其他表达式出现在文本 `user_name`的位置上，但是实际上这些选项并不重要。LightDB允许标识符语法（“username”），而 SQL 标准不允许。SQL 不允许在事务

中使用这个命令，而 LightDB并不做此限制，因为并没有原因需要这样做。和RESET语法一样，SESSION和 LOCAL修饰符是一种 LightDB扩展。

标准把执行这个命令所需的特权留给实现定义。

## 另见

SET ROLE

---

# SET TRANSACTION

SET TRANSACTION — 设置当前事务的特性

## Synopsis

```
SET TRANSACTION transaction_mode [, ...]  
SET TRANSACTION SNAPSHOT snapshot_id  
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

其中 `transaction_mode` 是下列之一：

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ  
UNCOMMITTED }  
READ WRITE | READ ONLY  
[ NOT ] DEFERRABLE
```

## 描述

SET TRANSACTION命令设置当前 会话的特性。SET SESSION CHARACTERISTICS设置一个会话后  
续事务的默认 事务特性。在个体事务中可以用 SET TRANSACTION覆盖这些默认值。

可用的事务特性是事务隔离级别、事务访问模式（读/写或只读）以及 可延迟模式。此外，  
可以选择一个快照，不过只能用于当前事务而不能 作为会话默认值。

一个事务的隔离级别决定当其他事务并行运行时该事务能看见什么数据：

READ COMMITTED

一个语句只能看到在它开始前提交的行。这是默认值。

REPEATABLE READ

当前事务的所有语句只能看到这个事务中执行的第一个查询或者 数据修改语句之前提交  
的行。

SERIALIZABLE

当前事务的所有语句只能看到这个事务中执行的第一个查询或者 数据修改语句  
之前提交的行。如果并发的可序列化事务间的读写 模式可能导致一种那些事务串  
行（一次一个）执行时不可能出现 的情况，其中之一将会被回滚并且得到一个  
serialization\_failure错误。

SQL 标准定义了一种额外的级别：READ UNCOMMITTED。在 LightDB中READ UNCOMMITTED被视  
作 READ COMMITTED。

一个事务执行了第一个查询或者数据修改语句（ SELECT、 INSERT、DELETE、  
UPDATE、FETCH或 COPY）之后就无法更改事务隔离级别。 更多有关事务隔离级别和并发控制  
的信息可见Chapter 13

事务的访问模式决定该事务是否为读/写或者只读。读/写是默认值。 当一个事务为只读  
时，如果 SQL 命令 INSERT、UPDATE、 DELETE和COPY FROM 要写的表不是一个临时表，则  
它们不被允许。不允许 CREATE、ALTER以及 DROP命令。不允许COMMENT、 GRANT、REVOKE、  
TRUNCATE。如果EXPLAIN ANALYZE 和EXECUTE要执行的命令是上述命令之一， 则它们也不被  
允许。这是一种高层的只读概念，它不能阻止所有对 磁盘的写入。

只有事务也是SERIALIZABLE以及 READ ONLY时，DEFERRABLE 事务属性才会有效。当一个事务的所有这三个属性都被选择时，该事务在第一次获取其快照时可能会阻塞，在那之后它运行时就不会有SERIALIZABLE事务的开销并且不会有任何牺牲或者被一次序列化失败取消的风险。这种模式很适合于长时间运行的报表或者备份。

SET TRANSACTION SNAPSHOT命令允许新的事务使用与一个现有事务相同的快照运行。已经存在的事务必须已经把它快照用pg\_export\_snapshot函数（见Section 9.27.5 导出。该函数会返回一个快照标识符，SET TRANSACTION SNAPSHOT需要被给定一个快照标识符来指定要导入的快照。在这个命令中该标识符必须被写成一个字符串，例如 '000003A1-1'。SET TRANSACTION SNAPSHOT只能在一个事务的开始执行，并且要在该事务的第一个查询或者数据修改语句（SELECT、INSERT、DELETE、UPDATE、FETCH或COPY）之前执行。此外，该事务必须已经被设置为SERIALIZABLE或者 REPEATABLE READ隔离级别（否则，该快照将被立刻抛弃，因为READ COMMITTED模式会为每一个命令取一个新快照）。如果导入事务使用了SERIALIZABLE隔离级别，那么导入快照的事务也必须使用该隔离级别。还有，一个非只读可序列化事务不能导入来自只读事务的快照。

## 注解

如果执行SET TRANSACTION之前没有 START TRANSACTION或者 BEGIN，它会发出一个警告并且不会有任何效果。

可以通过在BEGIN或者 START TRANSACTION中指定想要的transaction\_modes来省掉 SET TRANSACTION。但是在 SET TRANSACTION SNAPSHOT中该选项不可用。

会话默认的事务模式也可以通过配置参数 default\_transaction\_isolation、default\_transaction\_read\_only和 default\_transaction\_deferrable来设置或检查（实际上 SET SESSION CHARACTERISTICS只是用 SET设置这些变量的等效体）。这意味着可以通过配置文件、ALTER DATABASE等方式设置默认值。详见 Chapter 17

当前事务的模式可以类似的通过配置参数 transaction\_isolation、transaction\_read\_only、和transaction\_deferrable来设置或检查。设置这其中参数的作用与相应的SET TRANSACTION选项相同，在它何时可以完成方面，也有相同的限制。但是，这些参数不能在配置文件中设置，或者从活动SQL以外的任何来源来设置。

## 示例

要用一个已经存在的事务的同一快照开始一个新事务，首先要从该现有事务导出快照。这将会返回快照标识符，例如：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
pg_export_snapshot
-----
00000003-0000001B-1
(1 row)
```

然后在一个新开始的事务的开头把该快照标识符用在一个 SET TRANSACTION SNAPSHOT命令中：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

## 兼容性

SQL标准中定义了这些命令，不过 DEFERRABLE事务模式和 SET TRANSACTION SNAPSHOT形式除外，这两者是 LightDB扩展。

SERIALIZABLE是标准中默认的事务隔离级别。在 LightDB中默认值是普通的 READ COMMITTED，但是你可以按上述的方式更改。

在 SQL 标准中，可以用这些命令设置一个其他的事务特性：诊断区域 的尺寸。这个概念与嵌入式 SQL 有关，并且因此没有在 LightDB服务器中实现。

SQL 标准要求连续的transaction\_modes之间有逗号，但是出于历史原因 LightDB允许省略逗号。



---

# SHOW

SHOW — 显示一个运行时参数的值

## Synopsis

```
SHOW name
SHOW ALL
SHOW name%
SHOW %name
SHOW %name%
```

## 描述

SHOW将显示运行时参数的当前设置。这些变量可以使用SET语句、编辑 `lightdb.conf`配置参数、通过 `PGOPTIONS`环境变量（使用 `libpq`或者基于`libpq`的应用时）或者启动`lightdb`服务器时通过命令行标志设置。详见Chapter 17

## 参数

name

一个运行时参数的名称。可用的参数记录在 Chapter 17和SET参考页。此外，有一些可以显示但不能设置的参数：

SERVER\_VERSION

显示服务器的版本号。

LIGHTDB\_SERVER\_VERSION

显示 LightDB 服务器的版本号。

LIGHTDB\_SERVER\_VERSION\_NUM

显示 LightDB 服务器的版本号，其值为 `major_version * 10000 + minor_version * 100`。

SERVER\_ENCODING

显示服务器端的字符集编码。当前，这个参数可以被显示但不能被设置，因为该设置是在数据库创建时决定的。

LC\_COLLATE

显示数据库的排序规则（文本序）的区域设置。当前，这个参数可以被显示但不能被设置，因为该设置是在数据库创建时决定的。

LC\_CTYPE

显示数据库的字符分类的区域设置。当前，这个参数可以被显示但不能被设置，因为该设置是在数据库创建时决定的。

IS\_SUPERUSER

如果当前角色具有超级用户特权则为真。

ALL

显示所有配置参数的值，并带有描述。

name%

显示所有以 name 开头的配置参数的值和描述。

%name

显示所有以 name 结尾的配置参数的值和描述。

%name%

显示所有包含 name 的配置参数的值和描述。

## 注解

函数 `current_setting` 产生等效的输出，见 Section 9.27.1 还有，`pg_settings` 系统事务产生同样的信息。

## 示例

显示参数 `DateStyle` 的当前设置：

```
SHOW DateStyle;
DateStyle
-----
ISO, MDY
(1 row)
```

显示参数 `geqo` 的当前设置：

```
SHOW geqo;
geqo
-----
on
(1 row)
```

显示 LightDB 服务器版本：

```
SHOW lightdb_server_version;
lightdb_server_version
-----
13.8-23.1
(1 row)
```

显示 LightDB 服务器版本号：

```
SHOW lightdb_server_version_num;
lightdb_server_version_num
-----
220400
(1 row)
```

显示所有设置：

```
SHOW ALL;
```

name	setting	description
allow_system_table_mods	off	Allows modifications of the structure of ...
.	.	.
xmloption	content	Sets whether XML data in implicit parsing ...
zero_damaged_pages	off	Continues processing past damaged page headers.

(196 rows)

显示所有以参数 WAL% 开头的设置:

```
SHOW WAL%;
```

name	setting	description
wal_block_size	8192	Shows the block size in the write ahead log.
.	.	.
wal_writer_flush_after	1MB	Amount of WAL written out by WAL writer that triggers a flush.

(19 rows)

显示所有以参数 %LEVEL 结尾的设置:

```
SHOW %LEVEL;
```

name	setting	description
auto_explain.log_level	log	Log level for the plan.
.	.	.
wal_level	replica	Set the level of information written to the WAL.

(7 rows)

显示所有包含参数 %FREEZE 的设置:

```
SHOW %_FREEZE_%;
```

name	setting	description
autovacuum_freeze_max_age	200000000	Age at which to autovacuum a table to prevent transaction ID wraparound.

```
·  
·  
·  
vacuum_multixact_freeze_table_age | 150000000 | Multixact age at which VACUUM  
should scan whole table to freeze tuples.  
(6 rows)
```

## 兼容性

SHOW命令是一种 LightDB扩展。

## 另见

SET, RESET

---

# START TRANSACTION

START TRANSACTION — 开始一个事务块

## Synopsis

```
START TRANSACTION [ transaction_mode [, ...] ]
```

其中 `transaction_mode` 是下列之一：

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ  
UNCOMMITTED }  
READ WRITE | READ ONLY  
[ NOT ] DEFERRABLE
```

## 描述

这个命令开始一个新的事务块。如果指定了隔离级别、读写模式 或者可延迟模式，新的事务将会具有这些特性，就像执行了 `SET TRANSACTION`一样。这和 `BEGIN`命令一样。

## 参数

这些参数对于这个语句的含义可参考 `SET TRANSACTION`。

## 兼容性

在标准中，没有必要发出`START TRANSACTION` 来开始一个事务块：任何 SQL 命令会隐式地开始一个块。LightDB的行为可以被视作 在每个命令之后隐式地发出一个没有跟随在 `START TRANSACTION`（或者`BEGIN`）之后的 `COMMIT`并且因此通常被称作 “自动提交”。为了方便，其他关系型数据库系统也可能会 提供自动提交特性。

`DEFERRABLE transaction_mode` 是一种LightDB语言扩展。

SQL 标准要求连续的`transaction_modes`之间有逗号， 但是由于历史原因LightDB允许 省略逗号。

另见`SET TRANSACTION`的兼容性小节。

## 另见

`BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, `SET TRANSACTION`

---

# TRUNCATE

TRUNCATE — 清空一个表或者一组表

## Synopsis

```
TRUNCATE [ TABLE ] [ ONLY ] name [ * ] [, ... ]
        [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

## 描述

TRUNCATE可以从一组表中快速地移除所有行。它具有和在每个表上执行无条件DELETE相同的效果，不过它会更快，因为它没有实际扫描表。此外，它会立刻回收磁盘空间，而不是要求一个后续的VACUUM操作。在大表上它最有用。

## 参数

name

要截断的表的名字（可以是模式限定的）。如果在表名前指定了 ONLY，则只会截断该表。如果没有指定ONLY，该表及其所有后代表（如果有）都会被截断。可选地，可以在表名后指定 \*来显式地包括后代表。

RESTART IDENTITY

自动重新开始被截断表的列所拥有的序列。

CONTINUE IDENTITY

不更改序列值。这是默认值。

CASCADE

自动截断所有对任一所提及表有外键引用的表以及任何由于 CASCADE被加入到组中的表。

RESTRICT

如果任一表上具有来自命令中没有列出的表的外键引用，则拒绝截断。这是默认值。

## 注解

要截断一个表，你必须具有其上的TRUNCATE特权。

TRUNCATE在要操作的表上要求一个 ACCESS EXCLUSIVE锁，这会阻塞所有其他在该表上的并发操作。当指定RESTART IDENTITY时，任何需要被重新开始的序列也会被排他地锁住。如果要求表上的并发访问，那么应该使用DELETE命令。

TRUNCATE不能被用在被其他表外键引用的表上，除非那些表也在同一个命令中被阶段。这些情况中的可行性检查将会要求表扫描，并且重点不是为了做扫描。CASCADE选项可以被用来自动地包括所有依赖表——但使用它时要非常小心，否则你可能丢失数据！特别注意的是，当要被截断的表是一个分区时，兄弟节点分区不会受到影响，但是所有的引用表都发生级联，他们的分区也没有区别。

TRUNCATE将不会引发表上可能存在的任何 ON DELETE触发器。但是它将会引发 ON TRUNCATE触发器。如果在这些表的任意一个上定义了ON TRUNCATE触发器，那么所有的

BEFORE TRUNCATE触发器将在任何截断发生之前被引发，而所有AFTER TRUNCATE触发器将在最后一次截断完成并且所有序列被重置之后引发。触发器将以表被处理的顺序被引发（首先是那些被列在命令中的，然后是由于级联被加入的）。

TRUNCATE不是 MVCC 安全的。截断之后，如果并发事务使用的是一个在截断发生前取得的快照，表将对这些并发事务呈现为空。详见Section 13.5

从表中数据的角度来说，TRUNCATE是事务安全的：如果所在的事务没有提交，阶段将会被安全地回滚。

在指定了RESTART IDENTITY时，隐含的 ALTER SEQUENCE RESTART操作也会被事务性地完成。也就是说，如果所在事务没有提交，它们也将被回滚。注意如果事务回滚前在被重启序列上还做了额外的序列操作，这些操作在序列上的效果也将被回滚，但是它们在currval()上的效果不会被回滚。也就是说，在事务之后，currval()将继续反映在失败事务内得到的最后一个序列值，即使序列本身可能已经不再与此一致。这和失败事务之后 currval()的通常行为类似。

TRUNCATE当前不能用于被发布封装器支持的外部表。这意味着，如果指定的表具有任何外部的后代表，则该命令将失败。

## 示例

截断表bigtable和 fattable:

```
TRUNCATE bigtable, fattable;
```

做同样的事情，并且还重置任何相关联的序列发生器:

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

截断表othertable，并且级联地截断任何通过外键约束引用othertable的表:

```
TRUNCATE othertable CASCADE;
```

## 兼容性

SQL:2008 标准包括了一个TRUNCATE命令，语法是TRUNCATE TABLE tablename。子句CONTINUE IDENTITY/RESTART IDENTITY也出现在了该标准中，但是含义有些不同。这个命令的一些并发行为被标准留给实现来定义，因此如果必要应该考虑上述注解并且与其他实现进行比较。

## 参见

DELETE

---

# UNLISTEN

UNLISTEN — 停止监听一个通知

## Synopsis

```
UNLISTEN { channel | * }
```

## 描述

UNLISTEN被用来移除一个已经存在的对 NOTIFY事件的注册。 UNLISTEN取消任何已经存在的把当前 LightDB会话作为名为 channel的通知 频道的监听者的注册。特殊的通配符\*取消当前会话 的所有监听者注册。

NOTIFY包含有关LISTEN 和NOTIFY使用的更深入讨论。

## 参数

channel

一个通知频道的名称（任何标识符）。

\*

所有用于这个会话的当前监听注册都会被清除。

## 注解

你可以 unlisten 你没有监听的东西，不会出现警告或者错误。

在每一个会话末尾，会自动执行UNLISTEN \*。

一个已经执行了UNLISTEN的事务不能为 两阶段提交做准备。

## 示例

做一次注册：

```
LISTEN virtual;  
NOTIFY virtual;  
Asynchronous notification "virtual" received from server process with PID 8448.
```

一旦执行了UNLISTEN，进一步的NOTIFY 消息将被忽略：

```
UNLISTEN virtual;  
NOTIFY virtual;  
-- no NOTIFY event is received
```

## 兼容性

SQL 标准中没有UNLISTEN命令。



另见

LISTEN, NOTIFY

---

# UPDATE

UPDATE — 更新一个表的行

## Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
        ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] )
    |
        ( column_name [, ...] ) = ( sub-SELECT )
    } [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

## 描述

UPDATE更改满足条件的所有行中指定列 的值。只有要被修改的列需要在SET子句中提及， 没有被显式修改的列保持它们之前的值。

有两种方法使用包含在数据库其他表中的信息来修改一个表：使用子选择 或者在FROM子句中指定额外的表。这种技术只适合 特定的环境。

可选的RETURNING子句导致UPDATE 基于实际被更新的每一行计算并且返回值。任何使用该表的列以及 FROM中提到的其他表的列的表达式都能被计算。 计算时会使用该表的列的新（更新后）值。RETURNING 列表的语法和SELECT的输出列表相同。

你必须拥有该表上的UPDATE特权， 或者至少拥有 要被更新的列上的该特权。如果任何一列的值需要被 expressions或者 condition读取， 你还必须拥有该列上的SELECT特权。

如果指定了 ORDER BY 子句，则可以按指定的顺序更新行。 它的语义类似于 SELECT 的 ORDER BY 子句。请注意，不支持在具有 ORDER BY 子句的继承表中进行更新，也不支持 Canopy。

如果指定了 LIMIT 或 OFFSET 子句，则可以更新行的子集。 它的语义类似于 SELECT 的 LIMIT 子句。 请注意，不支持在具有 LIMIT 或 OFFSET 子句的继承表中进行更新，也不支持 Canopy。

## 参数

with\_query

WITH子句允许你指定一个或者更多个在 UPDATE中可用其名称引用的子查询。详见 Section 7.8和SELECT。

table\_name

要更新的表的名称（可以是模式限定的）。如果在表名前指定了 ONLY，只会更新所提及表中的匹配行。如果没有指定 ONLY，任何从所提及表继承得到的表中的匹配行也会 被更新。可选地，在表名之后指定\*可以显式地指示要 把后代表也包括在内。

**alias**

目标表的一个替代名称。在提供了一个别名时，它会完全隐藏表的真实名称。例如，给定UPDATE foo AS f， UPDATE语句的剩余部分必须用 f而不是foo来引用该表。

**column\_name**

指定表 table\_name 中的列名。如果需要，列名可以带有子字段名或数组下标。也可以使用别名来限定名称，例如 UPDATE table\_name AS t SET t.col = 1。请勿在目标列的指定中包含表的名称，例如 UPDATE table\_name SET table\_name.col = 1 是无效的。

**expression**

要被赋值给该列的一个表达式。该表达式可以使用该表中这一列或者其他列的旧值。

**DEFAULT**

将该列设置为它的默认值（如果没有为它指定默认值表达式，默认值将会为 NULL）。标识列将设置为关联序列生成的新值。对于生成的列，允许指定此项，但仅指定从其生成表达式计算列的正常行为。

**sub-SELECT**

一个SELECT子查询，它产生和在它之前的圆括号中列列表中一样多的输出列。被执行时，该子查询必须得到不超过一行。如果它得到一行，其列值会被赋予给目标列。如果它得不到行，NULL 值将被赋予给目标列。该子查询可以引用被更新表中当前行的旧值。

**from\_item**

表表达式允许来自其他表的列出现在WHERE条件和更新表达式中。这使用与 SELECT语句的FROM子句相同的语法；例如，可以指定表名的别名。不要将目标表作为from\_list 重复，除非你想做自连接（这种情况下它必须以别名出现在from\_list中）。

**condition**

一个返回boolean类型值的表达式。让这个表达式返回true的行将会被更新。

**cursor\_name**

要在WHERE CURRENT OF条件中使用的游标名。要被更新的是从这个游标中最近取出的行。该游标必须是一个在UPDATE目标表上的非分组查询。注意 WHERE CURRENT OF不能和一个布尔条件一起指定。有关对游标使用WHERE CURRENT OF的更多信息请见DECLARE。

**output\_expression**

在每一行被更新后，要被UPDATE命令计算并且返回的表达式。该表达式可以使用 table\_name指定的表或者FROM列出的表中的任何列名。写\* 可以返回所有列。

**output\_name**

用于一个被返回列的名称。

## 输出

成功完成时，一个UPDATE命令返回形如

```
UPDATE count
```

的命令标签。 count是被更新的行数，包括值没有更改的匹配行。注意，当更新被一个BEFORE UPDATE 触发器抑制时，这个数量可能比匹配 condition的行数少。如果 count为零，没有行被该查询更新（这不是一个错误）。

如果UPDATE命令包含一个RETURNING子句，其结果将类似于一个包含RETURNING列表中定义的列和值的SELECT语句（在被该命令更新的行上计算）的结果。

## 注解

当存在FROM子句时，实际发生的是：目标表被连接到 from\_item列表中的表，并且该连接的每一个输出行表示对目标表的一个更新操作。在使用FROM时，你应该确保该连接对每一个要修改的行产生至多一个输出行。换句话说，一个目标行不应该连接到来自其他表的多于一行上。如果发生这种情况，则只有一个连接行将被用于更新目标行，但是将使用哪一行是很难预测的。

由于这种不确定性，只在一个子选择中引用其他表更安全，不过这种语句通常很难写并且也比使用连接慢。

在分区表的情况下，更新一行有可能导致它不再满足其所在分区的分区约束。此时，如果这个行满足分区树中某个其他分区的分区约束，那么这个行会被移动到那个分区。如果没有这样的分区，则会发生错误。在后台，行的移动实际上是一次DELETE操作和一次INSERT操作。

在移动的行上的并发UPDATE或DELETE可能会收到序列化失败错误。假设会话1正在分区键上执行UPDATE，同时，对可访问该行的并发会话2在此行上执行UPDATE或DELETE操作。在这种情况下，会话2的UPDATE或DELETE将检测行移动并引发序列化失败错误（该错误始终返回SQLSTATE代码“40001”）。如果发生这种情况，应用程序可能希望重试事务。在通常情况下，表没有分区或没有行移动，会话2将标识新更新的行，并执行UPDATE/DELETE在此新行版本中。

请注意，虽然行可以从本地分区移动到外表分区（如果外数据包装器支持元组路由），但它们不能从外表分区移动到另一个分区。

## 示例

把表films的列kind中的单词Drama改成Dramatic:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

在表weather的一行中调整温度项并且把沉淀物重置为它的默认值:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2003-07-03';
```

执行相同的操作并且返回更新后的项:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2003-07-03'
RETURNING temp_lo, temp_hi, prcp;
```

使用另一种列列表语法来做同样的更新:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
WHERE city = 'San Francisco' AND date = '2003-07-03';
```

为管理Acme Corporation账户的销售人员增加销售量，使用FROM子句语法:

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
WHERE accounts.name = 'Acme Corporation'
AND employees.id = accounts.sales_person;
```

执行相同的操作，在 WHERE子句中使用子选择：

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
(SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

更新 accounts 表中的联系人姓名以匹配当前被分配的销售员：

```
UPDATE accounts SET (contact_first_name, contact_last_name) =
(SELECT first_name, last_name FROM salesmen
WHERE salesmen.id = accounts.sales_id);
```

可以用连接完成类似的结果：

```
UPDATE accounts SET contact_first_name = first_name,
                    contact_last_name = last_name
FROM salesmen WHERE salesmen.id = accounts.sales_id;
```

不过，如果salesmen.id不是一个唯一键，第二个查询可能会给出令人意外的结果，然而如果有多个id匹配，第一个查询保证会发生错误。还有，如果对于一个特定的accounts.sales\_id项没有匹配，第一个查询将把相应的姓名域设置为 NULL，而第二个查询完全不会更新该行。

更新一个统计表中的统计数据以匹配当前数据：

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
(SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
WHERE d.group_id = s.group_id);
```

尝试插入一个新库存项及其库存量。如果该项已经存在，则转而更新已有项的库存量。要这样做并且不让整个事务失败，可以使用保存点：

```
BEGIN;
-- 其他操作
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- 假定上述语句由于未被唯一键失败，
-- 那么现在我们发出这些命令：
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- 继续其他操作，并且最终
COMMIT;
```

更改表films中由游标c\_films定位的行的kind列：

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

更新表citys中按id排序的前5条记录的name列。

```
UPDATE citys SET name = 'null' order by id LIMIT 5;
```

## 兼容性

这个命令符合SQL标准，不过FROM和RETURNING子句是LightDB扩展，把WITH用于UPDATE也是扩展。

有些其他数据库系统提供了一个FROM选项，在其中在其中目标表可以在FROM中被再次列出。但 LightDB不是这样解释 FROM的。在移植使用这种扩展的应用时要小心。

根据标准，一个目标列名的圆括号子列表的来源值可以是任意得到正确列数的行值表达式。LightDB只允许来源值是一个行构造器或者一个子-SELECT。一个列的被更新值可以在行构造器的情况中被指定为DEFAULT，但在子-SELECT的情况中不能这样做。

Oracle模式下，如果分区键包含此列，列的更新可能导致数据从源分区表迁移至目标分区表。

---

# VACUUM

VACUUM — 垃圾收集并根据需要分析一个数据库

## Synopsis

```
VACUUM [ ( option [, ...] ) ] [ table_and_columns [, ...] ]  
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ] [ table_and_columns [, ...] ]
```

其中option可以是下列之一：

```
FULL [ boolean ]  
FREEZE [ boolean ]  
VERBOSE [ boolean ]  
ANALYZE [ boolean ]  
DISABLE_PAGE_SKIPPING [ boolean ]  
SKIP_LOCKED [ boolean ]  
INDEX_CLEANUP [ boolean ]  
TRUNCATE [ boolean ]  
PARALLEL integer
```

而table\_and\_columns是：

```
table_name [ ( column_name [, ...] ) ]
```

## 描述

VACUUM收回由死亡元组占用的存储空间。在通常的LightDB操作中，被删除或者被更新废弃的元组并没有在物理上从它们的表中移除，它们将一直存在直到一次VACUUM被执行。因此有必要周期性地做VACUUM，特别是在频繁被更新的表上。

在没有table\_and\_columns列表的情况下，VACUUM会处理当前用户具有清理权限的当前数据库中的每一个表和物化视图。如果给出一个列表，VACUUM可以只处理列表中的那些表。

VACUUM ANALYZE对每一个选定的表ANALYZE。这是两种命令的一种方便的组合形式，可以用于例行的维护脚本。其处理细节可参考ANALYZE。

简单的 VACUUM（不带FULL）简单地收回空间并使其可以被重用。这种形式的命令可以和表的普通读写操作并行，因为它不会获得一个排他锁。但是，这种形式中额外的空间并没有被还给操作系统（在大多数情况下），它仅仅被保留在同一个表中以备重用。它还允许我们利用多个 CPU 来处理索引。此功能称为parallel vacuum。要禁用此功能，可以使用PARALLEL选项并将并行工作程序指定为零。VACUUM FULL将表的整个内容重写到一个新的磁盘文件中，并且不包含额外的空间，这使得没有被使用的空间被还给操作系统。这种形式的命令更慢并且在其被处理时要求在每个表上保持一个ACCESS EXCLUSIVE 锁。

当选项列表被包围在圆括号中时，选项可以被写成任何顺序。如果没有圆括号，选项必须严格按照上面所展示的顺序指定。有圆括号的语法在LightDB 9.0时被加入，无圆括号的语法则被废弃。

## 参数

FULL

选择“完全”清理，它可以收回更多空间，并且需要更长时间和表上的排他锁。这种方法还需要额外的磁盘空间，因为它会创建该表的一个新拷贝，并且在操作完成之前都不会释放旧的拷贝。通常这种方法只用于需要从表中收回数量庞大的空间时。

## FREEZE

选择激进的元组“冻结”。指定FREEZE等价于参数vacuum\_freeze\_min\_age和vacuum\_freeze\_table\_age设置为0的VACUUM。当表被重写时总是会执行激进的冻结，因此指定FULL时这个选项是多余的。

## VERBOSE

为每个表打印一份详细的清理活动报告。

## ANALYZE

更新优化器用以决定最有效执行一个查询的方法的统计信息。

## DISABLE\_PAGE\_SKIPPING

通常，VACUUM将基于可见性映射跳过页面。已知所有元组都被冻结的页面总是会被跳过，而那些所有元组对所有事务都可见的页面则可能会被跳过（除非执行的是激进的清理）。此外，除非在执行激进的清理时，一些页面也可能被跳过，这样可避免等待其他页面完成对其使用。这个选项禁用所有的跳过页面的行为，其意图是只在可见性映射内容被怀疑时使用，这种情况只有在硬件或者软件问题导致数据库损坏时才会发生。

## SKIP\_LOCKED

规定VACUUM在开始处理关系时不等待任何冲突锁被释放：如果关系不能立即锁定而不等待，则跳过关系。请注意即使采用此选项，VACUUM在打开关系的索引时仍可能阻塞。此外，VACUUM ANALYZE在从分区、继承子表和某些类型的外表获取示例行时，仍然可能阻塞。还有，虽然VACUUM通常处理指定分区表的所有分区，但如果分区表上的锁冲突，此选项将导致VACUUM跳过所有分区。

## INDEX\_CLEANUP

指定VACUUM应尝试删除指向无效元组的索引条目。这通常是期望的行为，并且是默认设置，除非为要进行VACUUM的表设置了vacuum\_index\_cleanup选项为false。将此选项设置为false可能在需要尽快运行VACUUM的情况下很有用，例如避免即将发生的事务ID环绕（请参阅Section 22.1.5）。然而，如果不经常执行索引清理，性能可能会受到影响，因为随着表被修改，索引将累积无效元组，而表本身将累积无法删除的死行指针，直到索引清理完成。此选项对没有索引的表没有影响，如果使用了FULL选项，则忽略此选项。

## TRUNCATE

指定VACUUM尝试截断表末尾的任何空页，并允许将截断页的磁盘空间返回到操作系统。这通常是所需的行为，并且是默认行为，除非将vacuum\_truncate选项设置为false，对要被清空的表。将此选项设置为false可能有助于避免ACCESS EXCLUSIVE锁定需要截断的表。如果使用FULL选项，则忽略此选项。

## PARALLEL

使用integer后台处理器并行执行VACUUM的索引真空和索引清理阶段（每个真空阶段的详细信息请参考Table 25.37）。用于执行操作的处理器数量等于关系上支持并行清理的索引数量，该数量受PARALLEL选项指定的工人数量的限制，如果有的话，该数量还受到max\_parallel\_maintenance\_workers限制。当且仅当索引的大小大于min\_parallel\_index\_scan\_size时，索引才能参与并行清理。请注意，不保证在执行期间会使用integer中指定的并行工作线程数。清理运行时可能需要比指定的更少的处理器，甚至根本没有处理器。每个索引只能使用一名处理器。所以只有当表中至少有2索引时才会启动并行工作程序。在每个阶段开始之前启动清理工作进程，并在阶段结束时退出。这些行为可能会在未来的版本中发生变化。此选项不能与FULL选项一起使用。



boolean

指定打开还是关闭所选选项。你可以写入TRUE、ON或1以启用该选项，以及FALSE、OFF或0来禁用它。在TRUE被假定的情况下，boolean 值也可以被省略。

integer

指定传递给所选选项的非负整数值。

table\_name

要清理的表或物化视图的名称（可以有模式修饰）。如果指定的表示一个分区表，则它所有的叶子分区也会被清理。

column\_name

要分析的指定列的名称。缺省是所有列。如果指定了一个列的列表，则ANALYZE也必须被指定。

## 输出

如果声明了VERBOSE，VACUUM会发出进度消息来表明当前正在处理哪个表。各种有关这些表的统计信息也会打印出来。

## 注意

要清理一个表，操作者通常必须是表的拥有者或者超级用户。但是，数据库拥有者被允许清理他们的数据库中除了共享目录之外的所有表（对于共享目录的限制意味着一个真正的数据库范围的VACUUM只能被超级用户执行）。VACUUM将会跳过执行者不具备清理权限的表。

VACUUM不能在一个事务块内被执行。

对具有GIN索引的表，VACUUM（任何形式）也会通过将待处理索引项移动到主要GIN索引结构中的合适位置来完成任何待处理的索引插入。详见Section 62.4.1

我们建议经常清理活动的生产数据库（至少每晚一次），以保证移除失效的行。在增加或删除了大量行之后，对受影响的表执行VACUUM ANALYZE命令是一个很好的做法。这样做将把最近的更改更新到系统目录，并且允许LightDB查询规划器在规划用户查询时做出更好的选择。

日常使用时，不推荐FULL选项，但在特殊情况时它会有用。一个例子是当你删除或者更新了一个表中的绝大部分行时，如果你希望在物理上收缩表以减少磁盘空间占用并且允许更快的表扫描，则该选项是比较合适的。VACUUM FULL通常会比简单VACUUM更多地收缩表。

PARALLEL选项仅用于清理目的。如果此选项与ANALYZE选项一起指定，则不会影响ANALYZE。

VACUUM会导致I/O流量的大幅度增加，这可能导致其他活动会话性能变差。因此，有时建议使用基于代价的清理延迟特性。对于并行清理，每个处理器的睡眠与该处理器完成的工作成比例。详情请参阅Section 17.4.4

LightDB包括了一个“autovacuum”工具，它可以自动进行例行的清理维护。关于自动和手动清理的更多信息请见Section 22.1

## 例子

清理单一表onek，为优化器分析它并且打印出详细的清理活动报告：

```
VACUUM (VERBOSE, ANALYZE) onek;
```

## 兼容性

在SQL标准中没有VACUUM语句。

## 参见

`vacuumdb`, Section 17.4.4 Section 22.1.6

---

# VALUES

VALUES — 计算一个行集合

## Synopsis

```
VALUES ( expression [, ...] ) [, ...]  
  [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]  
  [ LIMIT { count | ALL } ]  
  [ OFFSET start [ ROW | ROWS ] ]  
  [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

## 描述

VALUES计算由值表达式指定的一个行值或者 一组行值。更常见的是把它用来生成一个大型命令内的“常量表”，但是它也可以被独自使用。

当多于一行被指定时，所有行都必须具有相同数量的元素。结果表的列数据类型 由出现在该列的表达式显式或者推导类型组合决定，决定的规则与 UNION相同（见Section 10.5。

在大型的命令中，在语法上允许VALUES出现在 SELECT出现的任何地方。因为语法把它当作一个 SELECT，可以为一个VALUES 命令使用ORDER BY、 LIMIT（或者等效的FETCH FIRST）以及OFFSET子句。

## 参数

expression

要在结果表（行集合）中指定位置计算并且插入的一个常量或者表达式。 在一个出现于INSERT顶层的 VALUES列表中， expression可以被DEFAULT替代以表示应该插入目标列的默认值。当VALUES出现在其他环境中时，不能使用 DEFAULT。

sort\_expression

一个指示如何排序结果行的表达式或者整型常量。这个表达式 可以用column1、column2等来 引用该VALUES结果的列。有关详细信息，请参阅 SELECT文档中的ORDER BY Clause。

operator

一个排序操作符。详见 SELECT文档中的ORDER BY Clause。

count

要返回的最大行数。详见 SELECT文档中的LIMIT Clause。

start

开始返回行之前要跳过的行数。详见 SELECT文档中的LIMIT Clause。

## 注解

应该避免具有大量行的VALUES列表，否则可能会 碰到内存不足失败或者很差的性能。出现在INSERT 中的VALUES是一种特殊情况（因为想要的列类型 可以从INSERT的目标表得知，并且不需要通过扫描 该VALUES列表来推导），因此它可以处理比其他 环境中更大的列表。

## 示例

一个纯粹的VALUES命令：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

这将返回一个具有两列、三行的表。它实际等效于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

更常用地，VALUES可以被用在一个大型 SQL 命令中。在INSERT中最常用：

```
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

在INSERT的环境中，一个VALUES列表的项可以是DEFAULT来指示应该使用该列的默认值而不是指定一个值：

```
INSERT INTO films VALUES
('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES也可以被用在可以写子-SELECT 的地方，例如在一个FROM子句中：

```
SELECT f.*
FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
WHERE f.studio = t.studio AND f.kind = t.kind;
```

```
UPDATE employees SET salary = salary * v.increase
FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

注意当VALUES被用在一个FROM子句中时， 需要提供一个AS子句，与SELECT相同。不需要为所有的列用AS子句指定名称，但是那样做是一种好习惯（在LightDB中，VALUES的默认列名是column1、column2等，但在其他数据库系统中可能会不同）。

当在INSERT中使用VALUES时，值都会被自动地强制为相应目标列的数据类型。当在其他环境中使用时，有必要指定正确的数据类型。如果项都是带引号的字符串常量，强制第一个就足以以为所有项假设数据类型：

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'),
('192.168.1.43'));
```

### Tip

对于简单的IN测试，最好使用IN的 list-of-scalars形式 而不是写一个上面那样的VALUES查询。标量列表方法的 书写更少并且常常更加高效。

## 兼容性

VALUES符合 SQL 标准。LIMIT和OFFSET是 LightDB扩展，另见 SELECT。

## 另见

INSERT, SELECT

---

# LightDB 客户端应用

这部份包含LightDB客户端应用和工具的参考信息。不是所有这些命令都是通用工具，某些需要特殊权限。这些应用的共同特征是它们可以被运行在任何主机上，而不管数据库服务器在哪里。

当在命令行上指定用户和数据库名时，它们的大小写会被保留 — 空格或特殊字符的出现可能需要使用引号。表名和其他标识符的大小写不会被保留并且可能需要使用引号。

## Table of Contents

clusterdb .....	1787
createdb .....	1790
lt_createdb .....	1794
createuser .....	1797
dropdb .....	1801
dropuser .....	1804
ecpg .....	1807
executor .....	1809
lightdb_service.py .....	1810
lt_distributed_probackup.py .....	1815
lt_distributed_dump.py .....	1821
lt_distributed_restore.py .....	1832
ltdts_recvlogical .....	1839
ltdts_ora .....	1843
ltdts_mysql .....	1846
ltldr .....	1850
ltuldr .....	1853
lt_basebackup .....	1856
lt_config .....	1863
lt_dump .....	1866
lt_dumpall .....	1879
lt_isready .....	1886
lt_probackup .....	1888
lt_receivewal .....	1938
lt_recvlogical .....	1942
lt_restore .....	1946
lt_verifybackup .....	1955
ltbench .....	1958
ltcenter .....	1975
ltsql .....	1977
reindexdb .....	2017
vacuumdb .....	2021

---

# clusterdb

clusterdb — 聚簇一个LightDB数据库

## Synopsis

```
clusterdb [connection-option...] [ --verbose | -v ] [ --table | -t table ] ...  
[dbname]
```

```
clusterdb [connection-option...] [ --verbose | -v ] --all | -a
```

## 描述

clusterdb是一个工具，它用来对一个LightDB数据库中的表进行重新聚簇。它会寻找之前已经被聚簇过的表，并且再次在最后使用过的同一个索引上对它们重新聚簇。没有被聚簇过的表将不会被影响。

clusterdb是 SQL 命令CLUSTER的一个包装器。在通过这个工具和其他方法访问服务器来聚簇数据库之间没有实质性的区别。

## 选项

clusterdb接受下列命令行参数：

-a  
--all

聚簇所有数据库。

[-d] dbname  
[--dbname=]dbname

当不使用-a/--all时，指定要被聚簇的数据库名称。如果数据库名称未指定，则从环境变量PGDATABASE中读取数据库名称。如果该环境变量也没有被设置，则使用为连接指定的用户名作数据库名。dbname可以是connection string。如果是这样，连接时的字符串参数将覆盖所有冲突的命令行选项。

-e  
--echo

回显clusterdb生成并发送给服务器的命令。

-q  
--quiet

不显示进度消息。

-t table  
--table=table

只聚簇table。可以通过写多个-t开关来聚簇多个表。

-v  
--verbose

在处理期间打印详细信息。

-V  
--version

打印clusterdb版本并退出。

-?  
--help

显示关于clusterdb命令行参数的帮助并退出。

clusterdb也接受下列命令行参数用于连接参数：

-h host  
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port  
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username  
--username=username

要作为哪个用户连接。

-w  
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 .pgpass 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W  
--password

强制clusterdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，clusterdb将自动提示要求一个口令。但是，clusterdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

--maintenance-db=dbname

当使用-a/--all时，指定要连接到的数据库名称来发现哪些其他数据库应该被聚簇。如果没有指定，将使用postgres数据库。而如果它也不存在，将使用template1。这可以是connection string。如果是这样，连接时的字符串参数将覆盖所有冲突的命令行选项。另外，连接到其他数据库时，除了数据库名字本身其他的连接时字符串参数将被重新使用。

## 环境

PGDATABASE  
LTHOST  
LTPORT  
LTUSER

默认连接参数



PG\_COLOR

规定在诊断消息中是否使用颜色。可选的值为always, auto, never

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 诊断

在有困难时，可以在CLUSTER和1tsql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

## 例子

要聚簇数据库test:

```
$ clusterdb test
```

要聚簇在数据库xyzyz中的一个表foo:

```
$ clusterdb --table=foo xyzyz
```

## 参见

CLUSTER

---

# createdb

createdb — 创建一个新的LightDB数据库

## Synopsis

```
createdb [connection-option...] [option...] [dbname [description]]
```

## 描述

createdb创建一个新的LightDB数据库。

通常，执行这个命令的数据库用户将成为新数据库的所有者。但是，如果执行用户具有合适的权限，可以通过-o选项指定一个不同的所有者。

createdb是SQL命令CREATE DATABASE的一个包装器。在通过这个工具和其他方法访问服务器来创建数据库之间没有实质性的区别。

## 选项

createdb接受下列命令行参数：

dbname

指定要被创建的数据库名。该名称必须在这个集簇中所有LightDB数据库中唯一。默认是创建一个与当前系统用户同名的数据库。

description

指定与新创建的数据库相关联的一段注释。

-D tablespace

--tablespace=tablespace

指定该数据库的默认表空间（这个名称被当做一个双引号引用的标识符处理）。

-e

--echo

回显createdb生成并发送到服务器的命令。

-E encoding

--encoding=encoding

指定要在这个数据库中使用的字符编码模式。LightDB服务器支持的字符集在Section 21.3. 中描述。

-l locale

--locale=locale

指定要在这个数据库中使用的区域。这等效于同时指定--lc-collate和--lc-ctype。

--lc-collate=locale

指定要在这个数据库中使用的 LC\_COLLATE 设置。

---

`--lc-ctype=locale`

指定要在这个数据库中使用的 LC\_CTYPE 设置。

`-O owner`

`--owner=owner`

指定拥有这个新数据库的数据库用户（这个名称被当做一个双引号引用的标识符处理）。

`-T template`

`--template=template`

指定用于创建这个数据库的模板数据库（这个名称被当做一个双引号引用的标识符处理）。

`-V`

`--version`

打印createdb版本并退出。

`-?`

`--help`

显示关于createdb命令行参数的帮助并退出。

选项-D、-l、-E、-O和-T对应于底层 SQL 命令CREATE DATABASE的选项，关于这些选项的信息可见该命令的内容。

createdb也接受下列命令行参数用于连接参数：

`-h host`

`--host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。这可以是connection string>。如果是这样，连接时字符串参数将覆盖所有冲突的命令行选项。

`-p port`

`--port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

`-U username`

`--username=username`

要作为哪个用户连接。

`-w`

`--no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

`-W`

`--password`

强制createdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，createdb将自动提示要求一个口令。但是，createdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-w来避免额外的连接尝试。

`--maintenance-db=dbname`

指定要连接到来发现哪些其他数据库应该被聚簇的数据库名。如果没有指定，将使用postgres数据库。而如果它也不存在（或者如果它就是创建新数据库的名称），将使用template1。

`--mysql-lower-case=value`

在Mysql兼容模式下，是否开启表名全小写特性。取值为（1, true, on, 0, false, off）中的一个，默认值是1。（1, true, on）：开启（0, false, off）：不开启

`--compatible-type=type`

数据库兼容类型GUC参数，此值用于指示当前数据库兼容类型是什么。

更多数据库兼容类型 GUC `lightdb_syntax_compatible_type`, 参考 `lightdb_syntax_compatible_type`

`--ascii-zero-store-value=value`

`lightdb_ascii_zero_store_value` 是一个在Lightdb 23.3引入的GUC参数，此值用于在Oracle以及Mysql模式下用来替换'\0'字符。

更多关于GUC `lightdb_ascii_zero_store_value` 的信息，请参考 `lightdb_ascii_zero_store_value`

## 环境

PGDATABASE

如果被设置，就是要创建的数据库名，除非在命令行中覆盖。

LTHOST

LTPORT

LTUSER

默认连接参数。如果没有在命令行或PGDATABASE指定要创建的数据库名，LTUSER也决定要创建的数据库名。

PG\_COLOR

规定在诊断消息中是否使用颜色。可能的值为 `always`, `auto`和`never`。

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 诊断

在有困难时，可以在CREATE DATABASE和ltsql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

## 例子

要使用默认数据库服务器创建数据库demo:

```
$ createdb demo
```

要在主机eden、端口 5000 上使用template0 模板数据库创建数据库demo，这里是命令行命令和底层SQL命令：

```
$ createdb -p 5000 -h eden -T template0 -e demo  
CREATE DATABASE demo TEMPLATE template0;
```

## 参见

dropdb, CREATE DATABASE

---

# lt\_createdb

lt\_createdb — 创建一个新的LightDB数据库

## Synopsis

```
lt_createdb [connection-option...] [option...] [dbname [description]]
```

## 描述

lt\_createdb是创建一个LightDB数据库工具。

lt\_createdb是 `createdb` 命令的一个包装器。即通过这个工具和使用 `createdb` 命令创建数据库是一样的。但在分布式环境(canopy)下会自动在所有节点创建数据库，并且这些数据库同样也是组成一个分布式集群。

注意: canopy和数据库相关，此工具仅判断postgres库中的canopy部署情况，拉取postgres库中的canopy节点列表，然后在每一个节点中创建同名数据库。

## 选项

lt\_createdb接受下列命令行参数:

dbname

指定要被创建的数据库名。该名称必须在这个集簇中所有LightDB数据库中唯一，默认是创建一个与当前系统用户同名的数据库。

-D tablespace

--tablespace=tablespace

指定该数据库的默认表空间（这个名称被当做一个双引号引用的标识符处理）。

-e

--echo

回显lt\_createdb生成并发送到服务器的命令。

-E encoding

--encoding=encoding

指定要在这个数据库中使用的字符编码模式。LightDB服务器支持的字符集在Section 21.3. 中描述。

-l locale

--locale=locale

指定要在这个数据库中使用的区域。这等效于同时指定--lc-collate和--lc-ctype。

--lc-collate=locale

指定要在这个数据库中使用的 LC\_COLLATE 设置。

--lc-ctype=locale

指定要在这个数据库中使用的 LC\_CTYPE 设置。

-O owner

--owner=owner

指定拥有这个新数据库的数据库用户（这个名称被当做一个双引号引用的标识符处理）。

-T template

--template=template

指定用于创建这个数据库的模板数据库（这个名称被当做一个双引号引用的标识符处理）。

--compatible-type=compatible-type

兼容类型。默认是off：与其他数据库功能不兼容，可以设置为off、mysql和oracle。

--mysql-lower-case=mysql-lower-case

是否在MySQL兼容模式下启用全小写特性。该值可为（1、true、on、0、false、off）之一，缺省值为1。

--ascii-zero-store-value=ascii-zero-store-value

是否存储ASCII零的另一个ASCII值。可用的值范围从0到32（包括0和32）。

-V

--version

打印lt\_createdb版本并退出。

-?

--help

显示关于lt\_createdb命令行参数的帮助并退出。

lt\_createdb也接受下列命令行参数用于连接参数：

-h host

--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。这可以是connection string>。如果是这样，连接时字符串参数将覆盖所有冲突的命令行选项。

-p port

--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username

--username=username

要作为哪个用户连接。

-w

--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

`-W`  
`--password`

强制lt\_createdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，lt\_createdb将自动提示要求一个口令。但是，lt\_createdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

`--maintenance-db=dbname`

指定要连接到来发现哪些其他数据库应该被聚簇的数据库名。如果没有指定，将使用postgres数据库。而如果它也不存在（或者如果它就是创建新数据库的名称），将使用templatel。

## 环境

LTDATABASE

如果被设置，就是要创建的数据库名，除非在命令行中覆盖。

LTHOST

LTPORT

LTUSER

默认连接参数。如果没有在命令行或PGDATABASE指定要创建的数据库名，LTUSER也决定要创建的数据库名。

LT\_COLOR

规定在诊断消息中是否使用颜色。可能的值为 always, auto和never。

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 例子

在cn节点上执行，创建分布式数据库demo：

```
$ lt_createdb demo
```

要在非cn节点主机eden、端口 5000 上使用templatel 模板数据库创建非分布式数据库demo，这里是命令行命令：

```
$ lt_createdb -p 5000 -h eden -T templatel -e demo
```



---

# createuser

createuser — 定义一个新的LightDB用户账户

## Synopsis

```
createuser [connection-option...] [option...] [username]
```

## 描述

createuser创建一个新的LightDB用户（或者更准确些，是一个角色）。只有超级用户和具有CREATEROLE特权的用户才能创建新用户，因此createuser必须被以上两种用户调用。

如果你希望创建一个新的超级用户，你必须作为一个超级用户连接，而不仅仅是具有CREATEROLE特权。作为一个超级用户意味着绕过数据库中所有访问权限检查的能力，因此超级用户访问权限不能轻易被授予。

createuser是SQL命令CREATE ROLE的一个包装器。在通过这个工具和其他方法访问服务器来创建用户之间没有实质性的区别。

## 选项

createuser接受下列命令行参数：

username

指定要被创建的LightDB用户的名称。这个名称必须与这个LightDB安装中所有现存角色不同。

-c number

--connection-limit=number

为该新用户设置一个最大连接数。默认值为不设任何限制。

-d

--createdb

新用户将被允许创建数据库。

-D

--no-createdb

新用户将不被允许创建数据库。这是默认值。

-e

--echo

回显createuser生成并发送给服务器的命令。

-E

--encrypted

此选项已过时，但为了实现向后兼容仍然接受。

-g role

--role=role

指定一个角色，这个角色将立即加入其中成为其成员。 如果要把这个角色加入到多个角色中作为成员， 可以写多个-g开关。

`-i`  
`--inherit`

新角色将自动继承把它作为成员的角色特权。这是默认值。

`-I`  
`--no-inherit`

新角色将不会自动继承把它作为成员的角色特权。

`--interactive`

如果在命令行没有指定用户名，提示要求用户名，并且在命令行没有指定选项 `-d/-D`、`-r/-R`、`-s/-S`时也提示（一直到 LightDB 9.1 这都是默认行为）。

`-l`  
`--login`

新用户将被允许登入（即，该用户名能被用作初始会话用户标识符）。这是默认值。

`-L`  
`--no-login`

新用户将不被允许登入（一个没有登录特权角色仍然可以作为管理数据库权限的方式而存在）。

`-P`  
`--pwprompt`

如果给定，`createuser`将发出一个提示要求新用户的口令。如果你没有计划使用口令认证，这就不是必须的。

`-r`  
`--createrole`

新用户将被允许创建新的角色（即，这个用户将具有CREATEROLE特权）。

`-R`  
`--no-createrole`

新用户将不被允许创建新角色。这是默认值。

`-s`  
`--superuser`

新用户将成为一个超级用户。

`-S`  
`--no-superuser`

新用户将不会成为一个超级用户。这是默认值。

`-V`  
`--version`

打印`createuser`版本并退出。

`--replication`

新用户将具有REPLICATION特权，这在CREATE ROLE的文档中有更完整的描述。

`--no-replication`

新用户将不具有REPLICATION特权，这在CREATE ROLE的文档中有更完整的描述。

-?  
--help

显示有关createuser命令行参数的帮助并退出。

createuser也接受下列命令行参数作为连接参数：

-h host  
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port  
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username  
--username=username

要作为哪个用户连接（不是要创建的用户名）。

-w  
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 .pgpass 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W  
--password

强制createuser在连接到一个数据库之前提示要求一个口令（用来连接到服务器，而不是新用户的口令）。

这个选项不是必不可少的，因为如果服务器要求口令认证，createuser将自动提示要求一个口令。但是，createuser将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

## 环境

LTHOST  
LTPORT  
LTUSER

默认连接参数

PG\_COLOR

规定在诊断消息中是否使用颜色。可能的值为 always, auto和never。

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 诊断

在有困难时，可以在CREATE ROLE和ltsql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

## 例子

要在默认数据库服务器上创建一个用户joe:

```
$ createuser joe
```

要在默认数据库服务器上创建一个用户joe并提示要求一些额外属性:

```
$ createuser --interactive joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

要使用在主机eden、端口 5000 上的服务器创建同一个用户joe，并带有显式指定的属性，看看下面的命令:

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

要创建用户joe为一个超级用户并且立刻分配一个口令:

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5bala423792b526f799ae4eb3d59e' SUPERUSER
CREATEDB CREATEROLE INHERIT LOGIN;
```

在上面的例子中，在录入新口令时新口令并没有真正地被回显，但是为了清晰，我们特意把它列了出来。如你所见，该口令在被发送给客户端之前会被加密。

## 参见

dropuser, CREATE ROLE

---

# dropdb

dropdb — 移除一个LightDB数据库

## Synopsis

```
dropdb [connection-option...] [option...] dbname
```

## 说明

dropdb毁掉一个现有的LightDB数据库。执行这个命令的用户必须是一个数据库超级用户或该数据库的拥有者。

dropdb是SQL命令DROP DATABASE的一个包装器。在通过这个工具和其他方法访问服务器来删除数据库之间没有实质性的区别。

## 选项

dropdb接受下列命令行参数:

dbname

指定要被移除的数据库的名字。

-e  
-#echo

回显dropdb生成并发送给服务器的命令。

-f  
-#force

在删除目标数据库之前, 尝试终止与该数据库的所有现有连接。有关此选项的详细信息, 请参见DROP DATABASE。

-i  
-#interactive

在做任何破坏性的工作之前发出一个验证提示。

-V  
-#version

打印dropdb版本并退出。

-#if-exists

如果数据库不存在也不抛出一个错误。在这种情况下会发出一个提示。

-?  
-#help

显示有关dropdb命令行参数的帮助并退出。

dropdb也接受下列命令行参数作为连接参数:

```
-h host
-#host=host
```

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作Unix域套接字的目录。

```
-p port
-#port=port
```

指定服务器正在监听连接的TCP端口或本地Unix域套接字文件扩展。

```
-U username
-#username=username
```

要作为哪个用户连接。

```
-w
-#no-password
```

不发出口令提示。如果服务器要求口令认证并且没有可用的口令（例如一个.pgpass文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

```
-W
-#password
```

强制dropdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，dropdb将自动提示要求一个口令。但是，dropdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

```
-#maintenance-db=dbname
```

指定一个数据库的名称，将连接到这个数据库以便删除目标数据库。如果没有指定，将使用postgres数据库。而如果它也不存在（或者它就是正在被删除的目标数据库），将使用template1。这个值可以是一个连接字符串。如果是这样，连接字符串参数将覆盖任何有冲突的命令行选项。

## 环境

```
LTHOST
LTPORT
LTUSER
```

默认连接参数

```
PG_COLOR
```

规定在诊断消息中是否使用颜色。可能的值为always、auto以及never。

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 诊断

在有困难时，可以在DROP DATABASE和1tsql中找找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

## 示例

要在默认数据库服务器上毁掉数据库demo:

```
$ dropdb demo
```

要使用在主机eden、端口5000上的服务器中毁掉数据库demo，并带有验证和回显，看看下面的命令:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

## 另见

createdb, DROP DATABASE

---

# dropuser

dropuser — 移除一个LightDB用户账户

## Synopsis

```
dropuser [connection-option...] [option...] [username]
```

## 说明

dropuser移除一个已有的LightDB用户。只有超级用户以及具有CREATEROLE特权的用户能够移除LightDB用户（要移除一个超级用户，你必须自己是一个超级用户）。

dropuser是SQL命令DROP ROLE的一个包装器。在通过这个工具和其他方法访问服务器来删除用户之间没有实质性的区别。

## 选项

dropuser接受下列命令行参数：

username

指定要移除的LightDB用户的名字。如果没有在命令行指定并且使用了-i/-#interactive选项，你将被提醒要求一个用户名。

-e  
-#echo

回显dropuser生成并发送给服务器的命令。

-i  
-#interactive

在实际移除该用户之前提示要求确认，并且在没有在命令行指定用户名提示要求一个用户名。

-V  
-#version

打印dropuser版本并退出。

-#if-exists

如果用户不存在时不要抛出一个错误。在这种情况下将发出一个提示。

-?  
-#help

显示有关dropuser命令行参数的帮助并退出。

dropuser也接受下列命令行参数作为连接参数：

-h host  
-#host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作Unix域套接字的目录。



```
-p port  
-#port=port
```

指定服务器正在监听连接的TCP端口或本地Unix域套接字文件扩展。

```
-U username  
-#username=username
```

要作为哪个用户连接。

```
-w  
-#no-password
```

不发出口令提示。如果服务器要求口令认证并且没有可用的口令（例如一个.pgpass文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

```
-W  
-#password
```

强制dropuser在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，dropuser将自动提示要求一个口令。但是，dropuser将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

## 环境

```
LTHOST  
LTPORT  
LTUSER
```

默认连接参数

```
PG_COLOR
```

规定在诊断消息中是否使用颜色。可能的值为always、auto以及never。

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 诊断

在有困难时，可以在DROP ROLE和ltsql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

## 示例

要从默认数据库服务器移除用户joe：

```
$ dropuser joe
```

要使用在主机eden、端口5000上的服务器移除用户joe，并带有验证和回显，可使用下面的命令：

```
$ dropuser -p 5000 -h eden -i -e joe
```

```
Role "joe" will be permanently removed.  
Are you sure? (y/n) y  
DROP ROLE joe;
```

另见

createuser, DROP ROLE

---

# ecpg

ecpg — 嵌入式 SQL C 预处理器

## Synopsis

ecpg [option...] file...

## 描述

ecpg是用于 C 程序的嵌入式 SQL 预处理器。它通过将 SQL 调用替换为特殊函数调用把带有嵌入式 SQL 语句的 C 程序转换为普通 C 代码。输出文件可以被任何 C 编译器工具链处理。

ecpg将把命令行中给出的每一个输入文件转换为相应的 C 输出文件。如果输入文件名没有任何扩展名, 则假定为.pgc。文件扩展名将由 .c替换以构造输出文件名。但是输出文件名可以使用-o选项覆盖。

如果输入文件名只是-, ecpg从标准输入 读取程序 (并写入标准输出, 除非用-o重写)。

这个参考页没有描述嵌入式 SQL 语言。关于该主题请参考Chapter 34

## 选项

ecpg接受下列命令行参数:

-c

自动从 SQL 代码生成确定的 C 代码。当前, 这对EXEC SQL TYPE起效。

-C mode

设置一个兼容性模式。mode可以是INFORMIX, INFORMIX\_SE或ORACLE。

-D symbol

定义一个 C 预处理器符号。

-h

处理头文件。指定此选项后, 输出文件扩展名变为.h而不是 .c, 默认输入文件扩展名为.pgh 而不是.pgc。此外, 将强制启用-c选项。

-i

分析系统也包括文件。

-I directory

指定一个额外的包括路径, 用来寻找通过EXEC SQL INCLUDE包括的文件。默认值是。(当前目录)、/usr/local/include、在编译时定义的LightDB包括目录(默认: /usr/local/lightdb-x/include)以及/usr/include。

-o filename

指定ecpg应该将它的所有输出写到给定的filename。写-o将所有输出发送到标准输出。

`-r option`

选择运行时行为。option可以是下列之一：

`no_indicator`

不使用指示器而使用特殊值来表示空值。历史上曾有数据库使用这种方法。

`prepare`

在使用所有语句之前准备它们。libecpg 将保持一个预备语句的缓冲并当语句再被执行时重用该语句。如果缓冲满了，libecpg 将释放最少使用的语句。

`questionmarks`

为兼容性原因允许使用问号作为占位符。在很久以前这被用作默认值。

`-t`

打开事务的自动提交。在这种模式下，每一个 SQL 命令会被自动提交，除非它位于一个显式事务块中。在默认模式中，命令只有当EXEC SQL COMMIT被发出时才被提交。

`-v`

打印额外信息，包括版本和“包括”路径。

`--version`

打印ecpg版本并退出。

`-?`

`--help`

显示关于ecpg命令行参数的帮助并退出。

## 注解

在编译预处理好的 C 代码文件时，编译器需要能够找到LightDB包括目录中的ECPG头文件。因此，在调用编译器时，你可能必须使用-I选项（例如，`-I/usr/local/lightdb-x/include`）。

使用带有嵌入式 SQL 的 C 代码的程序必须被链接到libecpg库，例如使用链接器选项`-L/usr/local/lightdb-x/lib -lecpg`。

适合于安装的这些目录的值可以使用`lt_config`找到。

## 例子

如果你有一个名为`progl.pgc`的嵌入式 SQL C 源文件，你可以使用下列命令序列创建一个可执行程序：

```
ecpg progl.pgc
cc -I/usr/local/lightdb-x/include -c progl.c
cc -o progl progl.o -L/usr/local/lightdb-x/lib -lecpg
```

---

## executor

executor — 用于并行执行分组的SQL文件

### Synopsis

```
java -jar lt_lightdb_executor.jar [url] [username] [password] [filePath]
```

### 描述

一批SQL语句被分成在文本文件中定义的组。 同一组的SQL在同一个线程中顺序执行，不同组并发执行。

文件格式为每行一个\${group},\${sql}，使用','分隔组和SQL，SQL中不能包含字符' '。 这里是一个简单的示例文件：

```
group1, create table test(v int)
group1, insert into test(v) values(1)
group2, create table test2(v int)
group2, insert into test2(v) values(1)
```

### 选项

exector接受以下命令行参数：

url

目标数据库的JDBC连接字符串。

username

登录的用户名。

password

登录的密码。

filePath

要执行的SQL文件。

### 诊断

并行计数与运行lt\_lightdb\_executor.jar的计算机上的CPU核心数相同。

### 示例

```
$ java -jar lt_lightdb_executor.jar \  
"jdbc:postgresql://localhost:5432/postgres" "lightdb" "123" t.sql
```

---

# lightdb\_service.py

lightdb\_service.py — 启动/停止/重启LightDB服务

## Synopsis

```
lightdb_service.py --command | -c start,stop,restart,status [ --filename | -F
install_info ] [ --cn_only | -C ] [ --dn_only | -D ] [ --primary_only | -P ] [
--standby_only | -S ] [ --node | -n node_info ] --dry-run [ --force | -f ] [ --
verbose | -v ] [ --quiet | -q ] [ --parallel | -B parallel_processes ] [ --log_dir
| -l directory ] [ --ignore-keepalived | -i ] [ --help | -h ]
```

## 描述

lightdb\_service.py是一个实用程序，用于启动/停止LightDB服务。它可以完全启动或停止安装程序安装的LightDB服务中的所有组件。它还具有简单的状态检测功能，并且可以使用一些特殊模式来启动或停止服务。

必须通过安装程序安装LightDB服务才能使用LightDB服务。

lightdb\_service.py必须在python3下执行。

## 选项

lightdb\_service.py 接受以下命令行参数：

-h  
--help

显示帮助信息并退出。

[-c] start,stop,restart,status  
[--command ]start,stop,restart,status

启动/停止/重启/检查LightDB服务。

[-F] install\_info  
[--filename ]install\_info

指定包含安装信息的json文件的路径（默认使用uninstall/uninstallFile.json）。

-C  
--cn\_only

仅启动/停止/重启协调节点。

-D  
--dn\_only

仅启动/停止/重启数据节点。

-P  
--primary\_only

仅为主要重启启动/停止/重启主节点。

-S  
--standby\_only

仅为备份重启启动/停止/重启备份节点。

-D  
--quiet  
不显示进度消息。

-n node\_info  
--node node\_info  
仅启动/停止/重启指定的节点，节点信息的格式为ip:port。

--dry-run  
显示将执行的操作，但不执行它。

-f  
--force  
强制停止，即使有一些节点处于错误状态，也跳过不正确的节点。

-B parallel\_processes  
--parallel parallel\_processes  
并行运行的段主机数。默认为1。

-v  
--verbose  
调试打印。

-q  
--quiet  
抑制标准输出日志的状态消息。

-l directory  
--log\_dir directory  
日志文件目录，默认为/tmp/ltAdminLogs。

--ignore-keepalived  
忽略keepalived，通常用于没有keepalived的LightDB服务。

## 诊断

如果出现问题，请查看日志文件获取详细错误信息。

以下是错误代码和正常原因：

Table 271. 错误代码表

错误代码	原因
100	启动 LightDB 失败（请参考 LightDB 日志获取详细信息），不是主节点
101	停止 LightDB 失败（请参考 LightDB 日志获取详细信息），不是主节点
102	启动主 LightDB 失败（请参考 LightDB 日志获取详细信息）
103	停止主 LightDB 失败（请参考 LightDB 日志获取详细信息）

错误代码	原因
104	要操作的节点不在当前 LightDB 服务中。使用 '-n' 参数
105	获取 LightDB 角色失败。
106	无法获取 LightDB 进程状态，执行检查 LightDB 端口的命令失败。通常是 SSH 加密未配置。
107	LightDB 进程正在运行，但无法连接以执行 SQL。
108	LightDB 已停止，但应该正在运行。
109	LightDB 正在运行，但应该已停止。
110	无法获取 ltcluster 进程状态，因为执行获取命令失败。通常是 SSH 加密未配置或 LightDB 已停止。
111	ltclusterd 已停止，但应该正在运行。
112	ltclusterd 正在运行，但应该已停止。
113	集群中的所有 ltclusterd 均已暂停，但应该已暂停。
114	集群中的所有 ltclusterd 均已暂停，但应该已恢复。
115	集群中的所有 ltclusterd 暂停状态均未知。
116	集群中的 ltclusterd 暂停状态不一致。
117	ltclusterd 的配置文件未找到。
118	ltclusterd 的配置文件错误。
119	ltclusterd 没有配置文件。
120	ltclusterd 启动失败。
121	ltclusterd 停止失败。
122	ltclusterd 重启失败。
123	keepalived 正在运行，但应该已停止。
124	keepalived 启动失败。
125	keepalived 停止失败。
126	通用场景下执行命令失败，现在仅用于获取 standby.signal 文件数量错误。
127	判断 LightDB 角色时 standby.signal 文件的数量不正确。
128	无法获取高可用状态，因为执行命令失败。
129	高可用状态不正确（正确意味着所有服务都在正确运行）。
130	LightDB 服务的安装文件格式有误。
131	LightDB 服务的安装文件内容有误。

## 示例

启动所有测试：



```
$ lightdb_service.py -c start
```

停止所有服务:

```
$ lightdb_service.py -c stop
```

重启所有服务:

```
$ lightdb_service.py -c restart
```

检查状态:

```
$ lightdb_service.py -c status
```

仅启动/停止/重启主节点（活动节点和备用节点状态不变）:

```
$ lightdb_service.py -c start/stop/restart --primary_only
```

仅启动/停止/重启备用节点（活动节点和备用节点状态不变）:

```
$ lightdb_service.py -c start/stop/restart --standby_only
```

仅启动/停止/重启计算节点:

```
$ lightdb_service.py -c start/stop/restart --cn_only
```

仅启动/停止/重启数据节点:

```
$ lightdb_service.py -c start/stop/restart --dn_only
```

仅启动/停止/重启数据节点主节点:

```
$ lightdb_service.py -c start/stop/restart --dn_only --primary_only
```

仅启动/停止/重启指定节点（活动节点和备用节点状态不变）:

```
$ lightdb_service.py -c start/stop/restart -n 10.20.148.122:54333
```

强制停止:

```
$ lightdb_service.py -c stop -f
```

模拟运行启动命令:

```
$ lightdb_service.py -c start --dry-run
```

---

# lt\_distributed\_probackup.py

lt\_distributed\_probackup.py — 备份和恢复 LightDB 分布式集群

## lt\_distributed\_probackup

lt\_distributed\_probackup.py 是一个用于管理 LightDB 分布式数据库集群备份和恢复的工具。它旨在定期备份 LightDB 实例，以便在发生故障时恢复服务器。要求使用 Python 3 版本。

lt\_distributed\_probackup.py 提供以下类似于 lt\_probackup 的好处，可以帮助您实现不同的备份策略和处理大量数据：

- 增量备份：基于页面的增量备份可以节省磁盘空间，加快备份和恢复速度。使用三种不同的增量模式，您可以根据数据流计划备份策略。
- 增量恢复：基于页面的增量恢复可以通过重用目标目录中有效未更改的页面，极大地加快恢复速度。
- 合并：使用此功能可以实现“增量更新备份”策略，消除了定期全量备份的需求。
- 验证：自动数据一致性检查和按需备份验证，无需实际数据恢复。
- 验证：使用 checkdb 命令对 LightDB 分布式实例进行按需验证。
- 保留：根据保留策略管理 WAL 存档和备份。您可以根据恢复时间或要保留的备份数量配置保留策略，还可以为特定备份指定生存时间（TTL）。过期的备份可以合并或删除。
- 并行处理：在多个并行线程上运行备份、恢复、合并、删除、验证和验证过程。
- 压缩：将备份数据存储在压缩状态下，以节省磁盘空间。
- 去重：通过不复制未更改的非数据文件，例如 \_vm 或 \_fsm，来节省磁盘空间。
- 远程操作：备份位于远程系统上的 LightDB 分布式实例或远程恢复备份。
- 从备用服务器备份：通过从备用服务器获取备份，避免对主服务器造成额外负载。
- 外部目录备份：备份位于 LightDB data directory (LTDATA) 之外的文件和目录，例如脚本、配置文件、日志或 SQL 转储文件。
- 备份目录：以纯文本或 JSON 格式获取备份列表和相应的元信息。
- 存档目录：以纯文本或 JSON 格式获取所有 WAL 时间轴和相应的元信息列表。
- 部分恢复：仅恢复指定的数据库或从恢复中排除指定的数据库。

为了管理备份数据，lt\_distributed\_probackup.py 创建了一个备份目录。该目录存储所有备份文件和额外的元信息，以及用于时点恢复所需的 WAL 存档。您可以在单个备份目录的不同子目录中存储不同实例的备份。

使用 lt\_distributed\_probackup.py，您可以进行完全备份或增量备份：

- 完全备份 包含从头开始恢复数据库集群所需的所有数据文件。
- 增量备份 仅存储自上次备份以来发生更改的数据。它可以减小备份大小并加速备份操作。lt\_distributed\_probackup.py 支持以下几种增量备份模式：

- PAGE 备份。在此模式下，lt\_distributed\_probackup.py 从上一次完全备份或增量备份开始扫描存档中的所有 WAL 文件。新创建的备份仅包含在 WAL 记录中提到的页面。这需要上一次备份以来的所有 WAL 文件存在于 WAL 存档中。如果这些文件的大小与数据库集群文件的总大小相当，则速度提高较小，但备份仍然占用较少的空间。
- DELTA 备份。在此模式下，lt\_distributed\_probackup.py 读取 LTDATA 目录中的所有数据文件，并仅复制自上次备份以来发生更改的页面。它不需要连续归档才能运行。此模式也可能会导致与完全备份相同的只读 I/O 压力。
- PTRACK 备份。在此模式下，LightDB 实时跟踪页面更改。不需要连续归档才能运行此模式。每次更新关系页面时，该页面都会在此关系的特殊 PTRACK 位图中标记。由于每个页面只需要一个位于 PTRACK 分支中的位，因此此类位图非常小。跟踪意味着在数据库服务器操作上会有一些小的开销，但可以显著加速增量备份。

无论选择何种备份类型，使用 lt\_distributed\_probackup.py 所进行的所有备份都支持以下 WAL 传递策略：

- 自主备份 通过复制协议流式传输所有 WAL 文件，以便在备份时将集群恢复到一致状态。即使没有设置连续归档，所需的 WAL 段也会被包含在备份中。
- 归档备份依赖于连续归档。

lt\_distributed\_probackup.py 将使用 lt\_probackup 来管理 LightDB 分布式数据库集群的备份和恢复。它几乎与 lt\_probackup 具有相同的使用方式，并完全支持 lt\_probackup 支持的所有功能。具体的区别如下：

## 与 lt\_probackup 不同之处

lt\_distributed\_probackup.py 在添加实例和备份时使用连接选项从协调器获取数据节点信息。

lt\_distributed\_probackup.py 在恢复、设置配置、显示配置、验证、检查数据库、显示、删除和删除实例时，通过协调器的实例名称（由 '--instance' 指定）在备份路径中查找匹配的数据节点实例名称。

为了为协调器和数据节点设置不同的选项，一些选项可以设置为由分号分隔的列表。如果未设置为列表，则所有节点将设置相同的选项。

'-i --backup-id' 可以是分布式备份 ID，您可以通过 show 命令查看分布式备份 ID。通过指定它，您可以同时操作数据节点的备份。从 23.1 版本开始支持此功能。没有 '--detail' 选项的 show 命令中，旧备份的状态都将为 'ERROR'。

## 日志选项

- 日志选项用于控制 lt\_distributed\_probackup.py 的日志，但不包括 lt\_probackup 的日志。
- 添加选项 '--lt-probackup-log-level' 以更改 lt\_probackup 的控制台日志级别。
- 默认情况下启用文件日志记录。日志级别为 `verbose`，日志名称为 `'lt_distributed_probackup-%Y-%m-%d.log'`。默认日志目录为 `'BACKUP_PATH/log'`。但是，当使用带有 '--with-init' 选项的备份命令时，`'BACKUP_PATH'` 可能不存在，日志目录将是 `'/tmp/ltAdminLogs'`。
- 不支持 '--error-log-filename'、 '--log-rotation-size' 和 '--log-rotation-age' 选项。 '--log-rotation-size' 和 '--log-rotation-age' 用于单个日志文件，但默认情况下它不是单个日志文件。
- '--log-level-console' 和 '--log-rotation-age' 用于单个日志文件，但默认情况下它不是单个日志文件。

## 添加实例

- 添加连接选项以连接到协调器节点。如果它是主节点，则将为所有主节点添加实例，否则将为所有备用节点添加实例。
- `--remote-host` 将被忽略，因为它与 `-h` 相同。
- `--remote-path` 可以是一个列表。
- 添加选项 `--no_distribution` 以类似于 `lt_probackup` 执行。
- 执行 `add-instance` 命令后，它将输出生成的数据节点实例名称。在数据节点的 `archive_comand` 中，`--instance` 必须与其相同，或者您可以通过 `'cn_instance_name_dn_id'`（例如：`'cn_dn_1'`）获得它，`'cn_instance_name'` 是由 `--instance` 指定的协调器实例名称，`id` 是 `pg_dist_node` 中的 `node_id`。

## 备份

- 使用 `connection` 选项连接到协调节点。如果它是主节点，它将备份所有主节点；否则，它将备份所有备用节点。
- `--remote-host` 选项将被忽略，因为它等同于 `-h` 选项。
- `-D pgdata-path` 选项将被忽略，因为它已经在 `add-instance` 命令中添加到配置中了。
- 添加选项 `--with-init`，设置后，如果备份不存在，则会初始化并添加实例。
- 添加选项 `--parallel-num` 以并行执行 `lt_probackup`。默认值为 1。

## 恢复

- 通过指定 `cn` 实例名称，它将获取所有 `datanode` 实例名称并将它们还原。
- 目前不支持选项 `--db-include` 和 `--db-exclude`。
- `-D pgdata-path`、`-i backup_id`、`--recovery-taget-xid`、`--recovery-taget-lsn`、`--recovery-taget-xid`、`--recovery-taget-timeline`、`--recovery-taget-name`、`--restore_command`、`--primary_conninfo`、`--primary-slot-name`、`--tablespace-mapping`、`--remote-host` 和 `--remote-path` 可以是一个列表。
- 恢复完成后，您可能需要调用 `canopy_update_node` 命令来更改协调器和数据节点的元数据，如主机和端口。
- 添加选项 `--parallel-num` 以并行执行 `lt_probackup`。默认值为 1。
- 添加选项 `--no_distribution`，以像 `lt_probackup` 一样执行。

## 展示

- 通过指定协调器实例名称，它将获取所有数据节点实例名称，并对它们执行命令。
- 展示命令支持选项 `--no_distribution`，以像 `lt_probackup` 一样执行。
- 展示命令将显示分布式备份状态。如果节点的备份状态不正确，则状态将不会是 `ok`。
- 使用 `--detail` 选项，展示命令将显示所有节点的备份信息。这是 LightDB 版本 22.4 的默认行为。在 LightDB 版本 23.1 中，您必须指定 `--detail` 才能显示所有节点的备份信息。

## 设置配置/展示配置/检查数据库/验证/删除/删除实例

- 通过指定协调器实例名称，它将获取所有数据节点实例名称，并对它们执行命令。

- set-config、delete 和 del-instance 命令支持选项 '--no\_distribution'，以像 lt\_probackup 一样执行。
- 对于 checkdb 命令，如果存在，则必须将 connection 选项设置为协调器。建议使用指定 -B 和 --instance 的方式来使用它。
- delete 命令的选项 '-i' 可以是一个列表。
- validate 命令的选项 '-i'、'--recovery-target-xid'、'--recovery-target-lsn'、'--recovery-target-timeline' 和 '--recovery-target-name' 可以是一个列表。

## 设置备份/合并

- 使用方法与 lt\_probackup 相同，但有更多的日志输出，您可以直接使用 lt\_probackup。

## 示例

### 预处理

- ssh 加密免费

### 使用方法

备份服务器: 192.168.247.126; 备份目录: /home/lightdb/backup 协调器信息: 192.168.247.127:54332 协调器数据目录: /home/lightdb/data 数据节点1信息: 192.168.247.128:54332 数据节点2信息: 192.168.247.129:54332 新集群(cn;dn1;dn2): 192.168.247.130;192.168.247.131;192.168.247.132

1. 初始化备份目录，目录路径: /home/lightdb/backup

```
lt_distributed_probackup.py init -B /home/lightdb/backup
```

2. 添加实例，协调器实例名称: cn; 协调器实例 IP、端口: 192.168.247.128:54332; 协调器数据目录: /home/lightdb/data。

```
lt_distributed_probackup.py add-instance -B /home/lightdb/backup -D /home/lightdb/data --instance cn -h192.168.247.128 -p5432
```

3. 在添加实例后设置连续的 WAL 存档。执行 add-instance 命令时，将生成数据节点实例。

```
# 协调器
archive_mode=on
archive_command="/home/lightdb/lightdb-x/13.8-23.1/bin/lt_probackup archive-push -B /home/lightdb/backup --instance=cn --wal-file-name=%f --remote-host=192.168.247.126"
# 数据节点1
archive_mode=on
archive_command="/home/lightdb/lightdb-x/13.8-23.1/bin/lt_probackup archive-push -B /home/lightdb/backup --instance=cn_dn_1 --wal-file-name=%f --remote-host=192.168.247.126"
```

4. 对分布式集群进行完整备份。

```
lt_distributed_probackup.py backup -B /home/lightdb/backup -D /home/lightdb/  
data --instance cn -h192.168.247.128 -p5432 -b full --parallel-num=1
```

5. 恢复 LightDB 分布式集群。

恢复到原始 LightDB 分布式集群。

```
lt_distributed_probackup.py restore -B /home/lightdb/backup --instance cn --  
parallel-num=1
```

恢复到新的 LightDB 分布式集群。

```
lt_distributed_probackup.py restore -B /home/lightdb/  
backup --instance cn --parallel-num=1 --remote-  
host='192.168.247.130;192.168.247.131;192.168.247.132'
```

6. checkdb, 备份必须在本地服务器上。

```
lt_distributed_probackup.py checkdb -B /home/lightdb/backup --instance cn  
lt_distributed_probackup.py checkdb -h10.19.70.50 -p54332
```

7. 为实例设置配置。

```
lt_distributed_probackup.py set-config -B /home/lightdb/backup --instance cn  
--archive-timeout=6min
```

8. 显示实例的配置。

```
lt_distributed_probackup.py show-config -B /home/lightdb/backup --instance  
cn
```

9. 验证备份。

```
lt_distributed_probackup.py validate -B /home/lightdb/backup --instance cn
```

10. 显示备份。

```
lt_distributed_probackup.py show -B /home/lightdb/backup --instance cn
```

11. 删除备份。

```
lt_distributed_probackup.py delete -B /home/lightdb/backup --instance cn --  
delete-expired --retention-redundancy=1
```

12. 删除实例。

```
lt_distributed_probackup.py del-instance -B /home/lightdb/backup --instance
cn
```

13. 合并, set-backup 与 lt\_probackup 相同。

```
lt_distributed_probackup.py merge -B /home/lightdb/backup --instance cn -i
RMICH6 -j 2 --progress --no-validate --no-sync
lt_distributed_probackup.py set-backup -B /home/lightdb/backup --instance
cn_dn_2 -i RMI6YR --note='cn_dn_2'
```

## 限制

lt\_distributed\_probackup.py 目前存在以下限制:

- 进行备份的服务器和恢复的服务器必须满足以下条件: block\_size (报告磁盘块的大小, 由构建服务器时的BLCKSZ值确定。默认值为8192字节) 和wal\_block\_size (报告WAL磁盘块的大小, 由构建服务器时的XLOG\_BLCKSZ值确定。默认值为8192字节) 参数相容, 并且具有相同的主版本号。
- 如果您没有创建目录的权限, 则实际上不会创建该目录, 尽管init命令已成功执行。
- 通过ssh运行远程操作时, 远程和本地的lt\_probackup版本必须相同。
- 即使您只需要备份本地的LightDB分布式数据库集群, 也需要为127.0.0.1和本地IP配置本地非加密。
- 最大并行度受'-j' 和 '--parallel-num' 的限制, 可能会受到sshd配置'MaxStartups' 的限制。 当'MaxStartups' 较小时, 使用大并行数执行lt\_distributed\_probackup.py可能会报错: 'ERROR: Agent error: kex\_exchange\_identification: read: Connection reset by peer'。 如果您想要更多并发性, 应该在/etc/ssh/sshd\_config中将'MaxStartups' 调高, 并重新启动sshd服务。



---

# lt\_distributed\_dump.py

lt\_distributed\_dump.py — 提取一个分布式LightDB数据库到一个目录中，包括脚本文件或其他存档文件。

## Synopsis

```
lt_distributed_dump.py [connection-option...] [option...]
```

## 描述

lt\_distributed\_dump 是用于备份分布式 LightDB 数据库的实用程序。只有在数据库没有并发使用时，它才能进行一致的备份。使用 lt\_distributed\_dump 不会阻止其他用户访问数据库（读者或写入者）。

lt\_distributed\_dump 使用 lt\_dump 来转储数据库。

lt\_distributed\_dump 仅转储单个分布式数据库。

转储可以以脚本或存档文件格式输出。不推荐使用脚本格式，因为对于分布式数据库，在协调器中创建分布式表之后，纯文本文件中的分布式子表名称将与工作节点中的子表名称不同。

脚本转储是包含重建数据库所需的 SQL 命令的纯文本文件，以使其恢复到保存时的状态。要从此脚本中恢复，请将其提供给 ltsql。脚本文件可用于在其他计算机和其他体系结构上重建数据库；通过一些修改，甚至可用于其他 SQL 数据库产品。

另一种备份文件格式必须与 lt\_distributed\_restore.py 结合使用以重建数据库。它们允许 lt\_distributed\_restore.py 选择性地恢复内容，甚至在恢复之前重新排序项目。存档文件格式旨在在不同体系结构之间移植。

当与其中一种存档文件格式结合使用，并与 lt\_distributed\_restore.py 结合使用时，lt\_distributed\_dump.py 提供了一种灵活的归档和传输机制。可以使用 lt\_distributed\_dump.py 备份整个数据库，然后使用 lt\_distributed\_restore.py 来检查存档和/或选择要恢复的数据库部分。最灵活的输出文件格式是“custom”格式（-Fc）和“directory”格式（-Fd）。它们允许选择和重新排序所有存档项目，支持并行恢复，并默认进行压缩。只有“directory”格式支持并行转储。

lt\_distributed\_dump.py 支持转储 plorasql 包和过程，例如转储表等。

运行 lt\_distributed\_dump.py 时，应检查输出是否存在任何警告（打印在标准错误上），特别是考虑到下面列出的限制。

## 选项

下面的命令行选项控制输出的内容和格式。使用“--folder”指定一个目录来存储转储文件。该目录必须不存在。不支持“--create”，因为对于分布式数据库，恢复必须有数据库存在。

-a  
--data-only

仅转储工作节点的数据和模式

此选项类似于指定 --section=data。

`-b`  
`--blobs`

将大型对象包含在转储中。这是默认行为，除非指定了 `--schema`、`--table` 或 `--schema-only`。因此，`-b` 开关只有在请求特定模式或表的转储中添加大型对象时才有用。请注意，BLOB 被视为数据，因此在使用 `--data-only` 时将包含它们，但在使用 `--schema-only` 时不会包含它们。

`-B`  
`--no-blobs`

在转储中排除大型对象。

当同时使用 `-b` 和 `-B` 时，行为是在转储数据时输出大型对象，请参阅 `-b` 文档。

`-K`  
`--recreate-schema`

在输出创建命令之前直接输出删除模式的命令（使用级联模式），比普通模式更快。（除非也指定了 `--if-exists`，否则如果在目标数据库中不存在任何对象，则恢复可能会生成一些无害的错误消息。）

在发出存档（非文本）输出文件时，将忽略此选项。对于存档格式，可以在调用 `lt_distributed_restore.py` 时指定选项。

`-E encoding`  
`--encoding=encoding`

使用指定的字符集编码创建转储。默认情况下，转储是使用数据库编码创建的。（获得相同结果的另一种方法是将 `LTCLIENTENCODING` 环境变量设置为所需的转储编码。）

`-f dirname`  
`--folder=dirname`

将输出发送到指定的目录。该目录由 `lt_distributed_dump.py` 创建，必须在之前不存在。

`-F format`  
`--format=format`

选择输出的格式。 `format` 可以是以下之一：

`p`  
`plain`

输出一个纯文本的 SQL 脚本文件。

`c`  
`custom`

输出适用于输入到 `lt_restore` 的自定义格式存档。与目录输出格式一起，这是最灵活的输出格式，因为它允许在恢复过程中手动选择和重新排序存档项。此格式默认也是压缩的。

`d`  
`directory`

输出适用于输入到 `lt_distributed_restore.py`（默认值）的目录格式存档。这将创建一个目录，其中每个被转储的表和 BLOB 都有一个文件，以及一个所谓的目录文件，以机器可读的格式描述被转储的对象，`lt_distributed_restore.py` 可以读取该目录文件。目录格式存档可以使用标准 Unix 工具进行操作；例如，未压缩的存档中的文件可以使用 `gzip` 工具进行压缩。此格式默认也是压缩的，还支持并行转储。

t  
tar

输出适用于输入到 `lt_distributed_restore.py` 的 tar 格式存档。tar 格式与目录格式兼容：提取 tar 格式存档会产生一个有效的目录格式存档。但是，tar 格式不支持压缩。此外，在使用 tar 格式时，无法在恢复过程中更改表数据项的相对顺序。

-j njobs  
--jobs=njobs

使用并行方式运行转储，同时转储 njobs 个表。此选项可能会减少执行转储所需的时间，但也会增加数据库服务器的负载。只能在目录输出格式下使用此选项，因为这是唯一一个多个进程可以同时写入其数据的输出格式。

lt\_dump 将打开 njobs + 1 个连接到数据库，因此请确保您的 max\_connections 设置足够高，以容纳所有连接。

在运行并行转储时，请求数据库对象的排他锁可能会导致转储失败。原因是 lt\_dump 主进程请求要在工作进程稍后将转储的对象上共享锁，以确保在转储运行时没有人删除它们并使其消失。如果另一个客户端然后请求一个表的排他锁，该锁将不会被授予，但将在等待主进程的共享锁被释放时排队。因此，对表的任何其他访问也不会被授予，并将在排他锁请求之后排队。这包括试图转储表的工作进程。没有任何预防措施，这将是一个经典的死锁情况。为了检测到此冲突，lt\_dump 工作进程使用 NOWAIT 选项请求另一个共享锁。如果工作进程没有被授予此共享锁，那么在此期间其他人必须已经请求了排他锁，因此无法继续转储，因此 lt\_dump 没有选择而只能中止转储。

为了进行一致的备份，数据库服务器需要支持同步快照，这是在 LightDB 9.2 的主服务器和 10 的备机中引入的功能。使用此功能，数据库客户端可以确保他们看到相同的数据集，即使他们使用不同的连接。lt\_distributed\_dump.py -j 使用多个数据库连接；它用主进程连接到数据库，然后对每个工作任务再连接一次。如果没有同步快照功能，不同的工作任务不能保证在每个连接中看到相同的数据，这可能导致备份不一致。

-n name  
--schema=name

仅转储与 name 匹配的模式；这将选择模式本身以及其包含的所有对象。当未指定此选项时，将转储目标数据库中的所有非系统模式。可以通过编写多个 -n 开关来选择多个模式。请参见下面的 Examples。

#### Note

当指定 -n 时，lt\_distributed\_dump.py 不会尝试转储所选模式可能依赖的任何其他数据库对象。因此，不能保证特定模式的转储结果可以成功地单独还原到干净的数据库中。

#### Note

当指定 -n 时，不会转储诸如 blob 等非模式对象。您可以使用 --blobs 开关将 blob 添加回转储中。

-N name  
--exclude-schema=name

不要转储与 name 匹配的任何模式。该模式的解释方式与 -n 相同。可以多次给出 -N 来排除匹配多个模式的模式。

当同时给出 `-n` 和 `-N` 时，行为是仅转储与至少一个 `-n` 开关匹配但不匹配任何 `-N` 开关的模式。如果 `-N` 在没有 `-n` 的情况下出现，则与否则正常转储的模式匹配的模式都将被排除。

`-O`

`--no-owner`

不输出命令以使对象的所有权与原始数据库匹配。默认情况下，`lt_distributed_dump.py` 发出 `ALTER OWNER` 或 `SET SESSION AUTHORIZATION` 语句来设置创建的数据库对象的所有权。除非由超级用户（或拥有脚本中所有对象的用户）启动脚本，否则这些语句将在运行脚本时失败。要创建可以由任何用户还原的脚本，但将赋予该用户所有对象的所有权，请指定 `-O`。

在发出存档（非文本）输出文件时，此选项将被忽略。对于存档格式，可以在调用 `lt_distributed_restore.py` 时指定该选项。

`-R`

`--no-reconnect`

此选项已过时，但仍然被接受以保持向后兼容性。

`-s`

`--schema-only`

仅在协调器节点中转储对象定义（模式），不包括数据。不会转储工作节点。

该选项是 `--data-only` 的反向选项。它类似于指定 `--section=pre-data` `--section=post-data`。

（不要将其与 `--schema` 选项混淆，该选项使用单词“schema”具有不同的含义。）

`-S username`

`--superuser=username`

指定在禁用触发器时使用的超级用户用户名。这仅在使用 `--disable-triggers` 时相关。（通常最好将其省略，而是以超级用户身份启动生成的脚本。）

`-t name`

`--table=name`

仅转储名称与 `name` 匹配的表。可以通过写多个 `-t` 开关来选择多个表，详情见下面的 Examples。

此选项除了表外，还可用于转储匹配的视图、物化视图、外部表和序列的定义。它不会转储视图或物化视图的内容。

当使用 `-t` 时，`-n` 和 `-N` 开关不起作用，因为由 `-t` 选择的表将被转储，无论这些开关如何，而非表对象不会被转储。

### Note

当指定 `-t` 时，`lt_distributed_dump.py` 不会尝试转储选定表所依赖的任何其他数据库对象。因此，无法保证特定表的转储结果可以自行成功地还原到干净的数据库中。

### Note

写入 `-t tab` 将转储默认搜索路径中可见的任何一个表。您必须编写类似 `-t sch.tab` 的内容才能选择特定模式中的表。

-T name  
--exclude-table=name

不要转储与 name 匹配的任何表。可以多次使用 -T 来排除匹配多个模式的表。

当同时使用 -t 和 -T 时，行为是仅转储至少匹配一个 -t 开关但不匹配任何 -T 开关的表。如果 -T 在没有 -t 的情况下出现，则将排除与否则正常转储的表匹配的表。

-v  
--verbose

指定详细模式。这将导致 lt\_dump 将详细的对象注释和开始/结束时间输出到转储文件中，并将进度消息输出到标准错误中。这也将导致 lt\_distributed\_dump.py 输出调试日志级别。

-V  
--version

打印版本信息并退出。

-x  
--no-privileges  
--no-acl

防止转储访问权限（授予/撤销命令）。

-Z 0..9  
--compress=0..9

指定要使用的压缩级别。零表示不压缩。对于定制和目录存档格式，这指定了单个表数据段的压缩，而默认情况下是以适度的水平压缩。对于纯文本输出，设置非零压缩级别会导致整个输出文件被压缩，就像它已经通过 gzip 处理一样；但默认情况下不压缩。tar 存档格式目前根本不支持压缩。

--binary-upgrade

目前不支持

--column-inserts  
--attribute-inserts

将数据转储为带有显式列名的 INSERT 命令（INSERT INTO table (column, ...) VALUES ...）。这将使还原非常缓慢；它主要用于制作可以加载到非 LightDB 数据库中的转储。还原过程中发生任何错误将仅导致丢失问题 INSERT 的行，而不是整个表内容。

--disable-dollar-quoting

此选项禁用函数正文的美元引用，并强制使用 SQL 标准字符串语法进行引用。

--disable-triggers

仅在创建仅数据转储时，此选项才相关。它指示 lt\_distributed\_dump.py 在恢复数据时包括命令以临时禁用目标表上的触发器。如果表上有引用完整性检查或其他触发器，您不想在数据还原期间调用它们，请使用此选项。

目前，对于 --disable-triggers 发出的命令必须由超级用户执行。因此，您还应该使用 -S 指定超级用户名称，或者最好小心地以超级用户身份启动生成的脚本。

在发出存档（非文本）输出文件时，此选项将被忽略。对于存档格式，您可以在调用 lt\_restore 时指定此选项。

`--enable-row-security`

仅在转储具有行安全性的表的内容时才相关。默认情况下，`lt_distributed_dump.py` 调用的 `lt_dump` 将 `row_security` 设置为关闭，以确保从表中转储所有数据。如果用户没有足够的特权来绕过行安全性，则会抛出错误。此参数指示 `lt_distributed_dump.py` 调用的 `lt_dump` 将 `row_security` 设置为打开，允许用户转储他们可以访问的表内容的部分。

请注意，如果您当前使用此选项，则可能还希望将转储格式设置为 `INSERT`，因为还原期间的 `COPY FROM` 不支持行安全性。

`--exclude-table-data=pattern`

目前不支持。

`--extra-float-digits=ndigits`

在转储浮点数据时，使用指定值的 `extra_float_digits`，而不是最大可用精度。为备份目的制作的常规转储不应使用此选项。

`--if-exists`

在清理数据库对象时使用条件命令（即添加一个 `IF EXISTS` 子句）。除非还指定了 `--clean`，否则此选项无效。

`--include-foreign-data=foreignserver`

目前不支持。

`--lt-exclude-lightdb-objects`

排除 LightDB 在转储中创建的所有内置对象。

<p><b>Note</b></p> <p>对于 LightDB 创建的内置表或模式，当同时给出 <code>-t</code> 或 <code>-n</code> 时，在转储数据时，行为是忽略表或模式，请参见 <code>-t</code> 和 <code>-n</code> 文档。</p>
---

`--lt-dump-lightdb-tables`

目前不支持。

`--lt-disable-auto-analyze`

转储索引时不自动收集统计信息。（有关详细信息，请参见 `CREATE INDEX`）。

`--inserts`

将数据转储为 `INSERT` 命令（而不是 `COPY`）。这将使还原非常缓慢；它主要用于制作可以加载到非 LightDB 数据库中的转储。还原过程中发生任何错误将仅导致丢失问题 `INSERT` 的行，而不是整个表内容。请注意，如果您重新排列列顺序的话，还原可能会完全失败。`--column-inserts` 选项对列顺序更改是安全的，但速度更慢。

`--load-via-partition-root`

当为表分区转储数据时，使 `COPY` 或 `INSERT` 语句的目标是包含它的分区层次结构的根，而不是分区本身。这将导致在加载数据时为每行重新确定适当的分区。当在行不总是落入与原始服务器上相同的分区的服务器上还原数据时，这可能很有用。例如，如果分区

列的类型为 `text`，并且两个系统对用于对分区列进行排序的排序规则的定义不同，则可能会发生这种情况。

最好不要在从使用此选项创建的归档还原时使用并行性，因为 `lt_restore` 将不知道给定归档数据项将加载数据到哪个分区（或哪些分区）。这可能会导致由于并行作业之间的锁冲突而导致效率低下，或者由于在加载所有相关数据之前设置了外键约束而导致还原失败。

`--lock-wait-timeout=timeout`

在转储开始时不要永远等待以获取共享表锁。相反，如果在指定的 `timeout` 内无法锁定表，则失败。超时可以以任何格式指定，这取决于您从中进行转储的服务器版本，但所有版本都接受以整数毫秒数为单位的超时。

`--no-comments`

不转储注释。

`--no-publications`

不转储发布。

`--no-security-labels`

不转储安全标签。

`--no-subscriptions`

不转储订阅。

`--no-sync`

默认情况下，`lt_distributed_dump.py` 调用的 `lt_dump` 将等待所有文件安全写入磁盘。此选项会导致 `lt_distributed_dump.py` 调用的 `lt_dump` 在不等待的情况下返回，这样速度更快，但意味着后续的操作系统崩溃可能会导致转储数据损坏。通常，此选项适用于测试，但在从生产安装转储数据时不应使用。

`--no-tablespaces`

不输出选择表空间的命令。使用此选项，所有对象都将在还原期间默认的表空间中创建。

在发出存档（非文本）输出文件时，此选项将被忽略。对于存档格式，您可以在调用 `lt_distributed_restore.py` 时指定此选项。

`--no-unlogged-table-data`

不转储无记录表的内容。此选项对于是否转储表定义（模式）没有影响；它只会禁止转储表数据。从备用服务器转储时，无记录表中的数据始终被排除。

`--on-conflict-do-nothing`

将 `ON CONFLICT DO NOTHING` 添加到 `INSERT` 命令中。除非也指定了 `--inserts`、`--column-inserts` 或 `--rows-per-insert`，否则此选项无效。

`--quote-all-identifiers`

强制引用所有标识符。当从其主要版本与 `lt_distributed_dump.py` 调用的 `lt_dump` 不同的服务器转储数据库，或者输出旨在加载到不同主要版本的服务器时，建议使用此选项。默认情况下，`lt_dump` 仅引用其自己主要版本中的保留字标识符。这有时会导致与其他版本的服务器处理时可能具有略有不同的保留字集的兼容性问题。使用 `--quote-all-identifiers` 可以避免这些问题，但会导致生成的转储脚本难以阅读。

--rows-per-insert=nrows

将数据转储为 INSERT 命令（而不是 COPY）。控制每个 INSERT 命令的最大行数。指定的值必须是大于零的数字。还原期间的任何错误都将导致仅丢失有问题的 INSERT 的一部分行，而不是整个表的内容。

--section=sectionname

仅转储指定的部分。部分名称可以是 pre-data、data 或 post-data。可以多次指定此选项以选择多个部分。默认情况下，会转储所有部分。

数据部分包含实际的表数据、大对象内容、序列值和工作节点中的数据定义项。Post-data 项包括索引、触发器、规则和除验证的检查约束之外的约束定义，它们位于协调器节点中。Pre-data 项包括协调器节点中的所有其他数据定义项。

--serializable-deferrable

尚不支持

--snapshot=snapshotname

尚不支持

--strict-names

尚不支持

--use-set-session-authorization

输出 SQL 标准的 SET SESSION AUTHORIZATION 命令，而不是 ALTER OWNER 命令来确定对象的所有权。这使得转储更符合标准，但取决于转储中对象的历史记录，可能无法正确还原。此外，使用 SET SESSION AUTHORIZATION 的转储肯定需要超级用户权限才能正确还原，而 ALTER OWNER 则需要较低的权限。

--parallel-num=number-of-lt\_dump

并行执行 lt\_dump。

--help

显示关于 lt\_distributed\_dump.py 命令行参数的帮助信息，并退出。

下面的命令行选项控制数据库连接参数。

-d dbname

--dbname=dbname

指定要连接和转储的数据库的名称。dbname 不能是连接字符串。

如果未指定，则使用环境变量 LTDATABASE。如果未设置该变量，则使用为连接指定的用户名。

-h host

--host=host

指定服务器运行的计算机的主机名。如果该值以斜线开头，则它将用作 Unix 域套接字的目录。默认值取自环境变量 LTHOST（如果设置），否则为 'localhost'。

-p port

--port=port

指定服务器用于侦听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认值取自环境变量 LTPORT（如果设置），否则为 '5432'。



-U username  
--username=username

连接时使用的用户名。默认值为环境变量 LTUSER（如果设置）或当前用户。

-w  
--no-password

不提示输入密码。如果服务器需要密码身份验证，但没有其他方式（如 .pgpass 文件）提供密码，则连接尝试将失败。此选项可用于批处理作业和脚本，其中没有用户可输入密码。

-W  
--password

强制 lt\_distributed\_dump.py 在连接到数据库之前提示输入密码。

--role=rolename

指定要用于创建转储的角色名称。此选项会导致 lt\_distributed\_dump.py 调用的 lt\_dump 在连接到数据库后发出 SET ROLE rolename 命令。当经过身份验证的用户（由 -U 指定）缺少 lt\_dump 所需的权限时，但可以切换到具有所需权限的角色时，此选项很有用。一些安装程序禁止直接登录超级用户，使用此选项允许在不违反策略的情况下进行转储。

以下命令行选项控制日志记录参数。

-l log-directory  
--log-directory=log-directory

指定日志目录路径。默认为 '/tmp/ltAdminLogs'。

--log-level-console=log-level-console

指定控制台日志级别。

--log-level-file=log-level-file

指定文件日志级别。

--log-filename=log-filename

指定日志文件名。默认为 'lt\_distributed\_dump-%Y-%m-%d.log'。

## 环境变量

LTDATABASE  
LTHOST  
LTOPTIONS  
LTPORT  
LTUSER

默认连接参数。

## 诊断

lt\_distributed\_dump.py 内部调用的 lt\_dump 执行 SELECT 语句。如果您运行 lt\_dump 时遇到问题，请确保您能够使用例如 ltsql 从数据库中选择信息。此外，libpq 前端库使用的任何默认连接设置和环境变量也将适用。

lt\_dump 的数据库活动通常由统计收集器收集。如果不想这样做，您可以通过 LTOPTIONS 或 ALTER USER 命令将参数 track\_counts 设置为 false。

## 注意事项

如果您的数据库群集对 template1 数据库有任何本地添加，则要小心使用 '--lt-exclude-lightdb-objects' 进行转储。

当选择数据转储并使用选项 --disable-triggers 时，lt\_dump 会发出命令，在插入数据之前禁用用户表上的触发器，然后在插入数据后重新启用它们的命令。如果还原在中途停止，系统目录可能会处于错误的状态。

由 lt\_distributed\_dump.py 生成的转储文件不包含优化器用于制定查询计划决策的统计信息。因此，最好在从转储文件还原后运行 ANALYZE，以确保最佳性能；有关更多信息，请参见 Section 22.1.3 和 Section 22.1.6

由于 lt\_distributed\_dump.py 用于传输数据到较新版本的 LightDB，因此可以预期其输出将加载到比 lt\_dump 的版本较新的 LightDB 服务器版本中。lt\_distributed\_dump.py 还可以从比其自身版本旧的 LightDB 服务器转储。（目前支持的服务器版本为 8.0 及更早版本。）但是，lt\_distributed\_dump.py 不能从比其自身主要版本新的 LightDB 服务器转储；它将拒绝尝试，而不是冒险制作无效的转储。此外，不能保证 lt\_distributed\_dump.py 的输出可以加载到较旧主要版本的服务器中 - 即使转储是从该版本的服务器中获取的。将转储文件加载到较旧的服务器中可能需要手动编辑转储文件，以删除较旧服务器不理解的语法。在跨版本情况下推荐使用选项 --quote-all-identifiers，因为它可以防止在不同的 LightDB 版本中出现不同的保留字列表引起的问题。

在转储逻辑复制订阅时，lt\_distributed\_dump.py 将生成使用 connect = false 选项的 CREATE SUBSCRIPTION 命令，以便还原订阅不会为创建复制槽或初始表复制而进行远程连接。这样，可以在不需要访问远程服务器的情况下还原转储。然后，由用户自行选择适当的方式重新激活订阅。如果涉及的主机已更改，则可能必须更改连接信息。在启动新的完整表复制之前，截断目标表可能也是适当的。

## 示例

将数据库转储为自定义格式存档文件：

```
$ lt_distributed_dump.py -Fc --folder dumpdir -d mydb --lt-exclude-lightdb-objects
```

将数据库转储为目录格式存档：

```
$ lt_distributed_dump.py -Fd --folder dumpdir -d mydb --lt-exclude-lightdb-objects
```

并行将数据库转储为目录格式存档，使用 5 个工作线程：

```
$ lt_distributed_dump.py -Fd --folder dumpdir -d mydb -j 5 --lt-exclude-lightdb-objects
```

将存档文件重新加载到名为 newdb 的（新创建的）数据库中：

```
$ lt_distributed_restore.py -d newdb --folder dumpdir
```

将存档文件重新加载到与其来自同一个数据库中，丢弃该数据库的当前内容：

```
$ lt_distributed_restore.py -d postgres --clean --create --folder dumpdir
```

转储名为 mytab 的单个表:

```
$ lt_distributed_dump.py -t mytab -d mydb --folder dumpdir --lt-exclude-lightdb-objects
```

## 另请参阅

lt\_dump, lt\_restore, lt\_distributed\_restore.py, ltsql

---

# lt\_distributed\_restore.py

lt\_distributed\_restore.py — 从由 lt\_distributed\_dump.py 创建的归档文件中还原一个 LightDB 数据库。

## Synopsis

```
lt_distributed_restore.py [connection-option...] [option...]
```

## 描述

lt\_distributed\_restore.py 是一个实用工具，用于从一个非普通文本格式的归档文件中还原一个 LightDB 数据库。该归档文件是由 lt\_distributed\_dump.py 创建的。它将发出必要的命令以将数据库重构为保存时的状态。归档文件还允许 lt\_distributed\_restore.py 有选择地恢复内容，甚至可以在恢复之前重新排序项目。归档文件旨在在不同体系结构之间进行移植。

显然，lt\_distributed\_restore.py 无法恢复归档文件中不存在的信息。例如，如果使用“将数据转储为INSERT命令”的选项创建了归档，则 lt\_distributed\_restore.py 将无法使用COPY语句加载数据。

## 选项

lt\_distributed\_restore.py 接受以下命令行参数。

-f dirname

--folder=dirname

指定恢复的转储文件所在的目录。

-a

--data-only

仅还原数据，而不还原模式（数据定义）。如果在归档中存在，则还原表数据、大对象和序列值。

此选项类似于指定 --section=data。

-c

--clean

在重新创建数据库对象之前清除（删除）这些对象。（除非使用 --if-exists，否则如果在目标数据库中存在某些对象，则可能会生成一些无害的错误消息。）

-K

--recreate-schema

直接输出删除模式的命令（使用级联模式），然后输出创建模式的命令，比普通模式更快。（除非也指定了 --if-exists，否则如果在目标数据库中不存在某些对象，则还原过程可能会生成一些无害的错误消息。）

-d dbname

--dbname=dbname

连接到数据库 dbname 并直接还原到该数据库中。 dbname 不能是 连接字符串。

-e  
--exit-on-error

对于由 `lt_distributed_restore.py` 调用的 `lt_restore`，如果在向数据库发送 SQL 命令时遇到错误，则退出。默认情况下，会继续执行，并在还原结束时显示错误计数。

-F format  
--format=format

指定归档的格式。不必指定格式，因为 `lt_restore` 会自动确定格式。如果指定了格式，则可以是以下之一：

c  
custom

归档以 `lt_dump` 的自定义格式为基础。

d  
directory

归档是目录归档。

t  
tar

归档是 tar 归档。

-I index  
--index=index

目前不支持。

-j number-of-jobs  
--jobs=number-of-jobs

并行运行 `lt_restore` 的最耗时步骤，包括加载数据、创建索引或创建约束等，最多使用 `number-of-jobs` 个并发会话。此选项可以大大缩短将大型数据库还原到运行在多台服务器上的服务器所需的时间。在发出脚本而不是直接连接到数据库服务器时，此选项将被忽略。

每个作业是一个进程或一个线程，取决于操作系统，并使用单独的连接到服务器。

此选项的最佳值取决于服务器、客户端和网络的硬件设置。因素包括 CPU 核心数和磁盘设置。一个好的起点是服务器上的 CPU 核心数，但在许多情况下，比这个值更大的值也可以导致更快的还原时间。当然，值过高会因为抖动而导致性能下降。

此选项仅支持自定义和目录归档格式。输入必须是常规文件或目录（例如，不能是管道或标准输入）。

-l  
--list

目前不支持。

-L list-file  
--use-list=list-file

目前不支持。

-n schema  
--schema=schema

仅还原指定模式中的对象。可以使用多个 `-n` 开关指定多个模式。可以与 `-t` 选项结合使用以仅还原特定表。

-N schema  
--exclude-schema=schema

不还原指定模式中的对象。可以使用多个 -N 开关指定要排除的多个模式。

当给定了相同的模式名称时，-n 和 -N，-N 开关胜出，并且该模式将被排除。

-O  
--no-owner

不输出命令以设置对象的所有权以匹配原始数据库。默认情况下，lt\_restore 发出 ALTER OWNER 或 SET SESSION AUTHORIZATION 语句来设置创建的模式元素的所有权。除非使用超级用户（或拥有脚本中所有对象的同一用户）进行与数据库的初始连接，否则这些语句将失败。使用 -O，可以使用任何用户名进行初始连接，该用户将拥有所有创建的对象。

-P function-name(argtype [, ...])  
--function=function-name(argtype [, ...])

目前不支持。

-s  
--schema-only

仅还原模式（数据定义），而不是数据，只要归档中存在模式条目。

此选项是 --data-only 的反面。它类似于指定 --section=pre-data --section=post-data。

（不要将此选项与 --schema 选项混淆，后者在不同的含义上使用单词 “schema”。）

-S username  
--superuser=username

指定禁用触发器时要使用的超级用户用户名。只有在使用 --disable-triggers 时才相关。

-t table  
--table=table

仅还原指定表的定义和/或数据。对于此目的，“table” 包括视图、物化视图、序列和外部表。可以通过编写多个 -t 开关来选择多个表。可以与 -n 选项结合使用，以指定特定模式中的表。

### Note

当指定 -t 时，lt\_distributed\_restore.py 不会尝试还原所选表可能依赖的任何其他数据库对象。因此，不能保证将特定表还原到干净的数据库中将成功。

### Note

虽然 lt\_distributed\_dump.py 的 -t 标志也会转储所选表的附属对象（例如索引），但 lt\_distributed\_restore.py 的 -t 标志不包括这些附属对象。

-T trigger  
--trigger=trigger

目前不支持。

-v  
--verbose

指定详细模式。

-V  
--version

打印 lt\_distributed\_restore.py 的版本并退出。

-x  
--no-privileges  
--no-acl

防止恢复访问权限（授权/撤销命令）。

-l  
--single-transaction

目前不支持。

--disable-triggers

此选项仅在执行仅数据还原时才相关。它指示 lt\_restore 在恢复数据时临时禁用目标表上的触发器。如果表上存在引用完整性检查或其他触发器，您不希望在数据还原期间调用它们，则使用此选项。

目前，为 --disable-triggers 发出的命令必须作为超级用户执行。因此，您还应使用 -S 指定超级用户名称或更好地作为 LightDB 超级用户运行 lt\_distributed\_restore.py。

--enable-row-security

此选项仅在还原具有行安全的表的内容时才相关。默认情况下，lt\_restore 将设置 row\_security 为 off，以确保将所有数据还原到表中。如果用户没有足够的权限绕过行安全，则会抛出错误。此参数指示 lt\_restore 将 row\_security 设置为 on，从而允许用户尝试启用行安全来还原表的内容。如果用户没有权利将转储中的行插入表中，则此操作仍可能失败。

请注意，此选项当前还要求转储以 INSERT 格式存在，因为 COPY FROM 不支持行安全。

--if-exists

使用条件命令（即添加 IF EXISTS 子句）删除数据库对象。除非还指定了 --clean，否则此选项无效。

--no-comments

不要输出还原注释的命令，即使归档文件包含它们。

--no-data-for-failed-tables

尚未支持。

--no-publications

不要输出用于还原发布的命令，即使归档文件包含它们。

`--no-security-labels`

不要输出用于还原安全标签的命令，即使归档文件包含它们。

`--no-subscriptions`

不要输出用于还原订阅的命令，即使归档文件包含它们。

`--no-tablespaces`

不要输出用于选择表空间的命令。使用此选项，所有对象都将在还原期间的默认表空间中创建。

`--section=sectionname`

仅还原指定的部分。部分名称可以是 `pre-data`、`data` 或 `post-data`。此选项可以多次指定以选择多个部分。默认情况下，将恢复所有部分。

数据部分包含实际的表数据以及大对象定义。后数据项包括索引、触发器、规则和除已验证的检查约束之外的所有约束定义。预数据项包括所有其他数据定义项。

`--strict-names`

尚未支持。

`--use-set-session-authorization`

输出 SQL 标准的 `SET SESSION AUTHORIZATION` 命令，而不是 `ALTER OWNER` 命令来确定对象的所有权。这使得转储更符合标准，但是取决于转储中对象的历史，可能无法正确还原。

`--table_exists_action`

告诉 `lt_distributed_restore.py` 如果它要创建的表已经存在该怎么办。`table_exists_action` 有四个选项：`skip`、`append`、`truncate` 或 `replace`，可能的取值具有以下效果：

`skip`

`skip` 保留表不变并继续处理下一个对象。

`append`

`append` 从源加载行并保留现有行不变。

`truncate`

`truncate` 删除现有行，然后从源加载行。

`replace`

`replace` 删除现有行，然后从源加载行。

当使用 `truncate` 或 `replace` 时，请确保受影响的表中的行不是任何参照约束的目标。当使用 `append` 或 `truncate` 时，请确保在执行任何操作之前，来自源的行与现有表兼容。

`--parallel-num=number-of-lt_restore`

并行执行 `lt_restore`。



--help

显示有关 lt\_distributed\_restore.py 命令行参数的帮助信息，并退出。

lt\_distributed\_restore.py 还接受以下用于连接参数的命令行参数：

-h host

--host=host

指定运行服务器的机器的主机名。如果值以斜杠开头，则将其用作Unix域套接字的目录。如果设置了 LTHOST 环境变量，则使用其默认值，否则使用 'localhost'。

-p port

--port=port

指定服务器正在侦听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。如果设置了 LTPORT 环境变量，则使用其默认值，否则使用 '5432'。

-U username

--username=username

要连接的用户名。如果设置了 LTUSER 环境变量，则使用其默认值，否则使用当前用户。

-w

--no-password

永远不要提示输入密码。如果服务器需要密码身份验证，而又没有其他方式（例如 .pgpass 文件）提供密码，则连接尝试将失败。此选项在批处理作业和脚本中非常有用，在这些作业和脚本中没有用户可以输入密码。

-W

--password

强制 lt\_distributed\_restore.py 在连接到数据库之前提示输入密码。

--role=rolename

指定要用于执行还原的角色名。此选项使 lt\_restore 在连接到数据库之后发出 SET ROLE rolename 命令。当经过身份验证的用户（由 -U 指定）缺少 lt\_restore 需要的权限时，它非常有用，但可以切换到具有所需权限的角色。某些安装不允许直接以超级用户身份登录，使用此选项可以允许在不违反策略的情况下执行还原。

以下命令行选项控制日志参数。

-l log-directory

--log-directory=log-directory

指定日志目录路径。默认为 '/tmp/ltAdminLogs'。

--log-level-console=log-level-console

指定控制台日志级别。

--log-level-file=log-level-file

指定文件日志级别。

--log-filename=log-filename

指定日志文件名。默认为 'lt\_distributed\_restore-%Y-%m-%d.log'。

## 环境变量

LTHOST  
LTOPTIONS  
LTPORT  
LTUSER

默认连接参数

## 诊断

如果您在运行 `lt_distributed_restore.py` 时遇到问题，请确保您能够使用例如 `ltsql` 选择来自数据库的信息。此外，`libpq` 前端库使用的任何默认连接设置和环境变量都将适用。

## 注意事项

如果您的安装中有任何对 `template1` 数据库的本地添加，请小心使用 `'--lt-exclude-lightdb-objects'` 选项进行转储。

`lt_distributed_restore.py` 的限制如下。

- 当将数据恢复到现有表中，并使用选项 `--disable-triggers` 时，`lt_distributed_restore.py` 调用 `lt_restore` 发出命令，在插入数据之前禁用用户表上的触发器，然后发出命令在插入数据后重新启用它们。如果恢复在中途停止，则系统目录可能处于错误的状态。
- `lt_distributed_restore.py` 无法有选择地恢复大对象；例如，仅针对特定表格。如果存档包含大对象，则将恢复所有大对象，如果它们通过 `-t` 或其他选项排除，则不恢复任何大对象。

有关 `lt_distributed_restore.py` 限制的详细信息，请参见 `lt_distributed_dump.py` 文档。

恢复完成后，最好对每个恢复的表运行 `ANALYZE`，以便优化器具有有用的统计信息；有关更多信息，请参见 Section 22.1.3 和 Section 22.1.6

## 示例

假设我们已将名为 `mydb` 的数据库转储到自定义格式的转储文件中：

```
$ lt_dump -Fc -d mydb -f dumpdir --lt-exclude-lightdb-objects
```

注意，我们需要使用 `'--lt-exclude-lightdb-objects'` 进行转储以排除扩展。

将转储还原到名为 `newdb` 的新分布式数据库中：

```
$ lt_restore -d newdb -f dumpdir
```

## 另请参阅

`lt_dump`, `lt_restore`, `lt_distributed_dump.py`, `ltsql`

---

# ltdts\_recvlogical

`ltdts_recvlogical` 是从 `lt_recvlogical` 分支出来的，因此它具有 `lt_recvlogical` 中的大部分行为，例如控制 LightDB 逻辑解码流。之后，它使用 `wal2sql` 插件将来自 LightDB 的更改转换为 Oracle 数据库，并将支持 Redis、Kafka、LightDB-A 等。

## Synopsis

```
ltdts_recvlogical [option...]
```

## 描述

`ltdts_recvlogical` 控制逻辑解码复制插槽并从这些复制插槽中流数据。

### Note

在 HA 环境中，请确保您启动了 `$LTHOME/bin/ltdts_logicalrepl_copier.sh`。

它创建一个复制模式的连接，因此受到与 `lt_receivewal` 相同的约束，再加上逻辑复制的约束（请参见 Chapter 45）。

`ltdts_recvlogical` 没有逻辑解码 SQL 接口的 `peek` 和 `get` 模式的等价物。它在接收到数据时惰性地发送回放确认，并在干净退出时发送。要检查插槽中未消耗的待处理数据，请使用 `pg_logical_slot_peek_changes`。

## 选项

必须指定以下至少一个选项来选择一个操作：

`--create-slot`

使用由 `--plugin` 指定的输出插件，为由 `--dbname` 指定的数据库，创建一个名为 `--slot` 的新逻辑复制插槽。

`--drop-slot`

删除名为 `--slot` 的复制插槽，然后退出。

`--start`

从由 `--slot` 指定的逻辑复制插槽开始流式传输更改，直到被信号终止。如果服务器端更改流以服务器关闭或断开连接结束，则在一个循环中重试，除非指定了 `--no-loop`。

流格式由创建插槽时指定的输出插件确定。

连接必须是与用于创建插槽的相同数据库连接。

`--create-slot` 和 `--start` 可以一起指定。`--drop-slot` 不能与其他操作结合使用。

以下命令行选项控制输出的位置和格式以及其他复制行为：

`-E lsn`

`--endpos=lsn`

在 `--start` 模式下，当接收到指定的 LSN 时自动停止复制并以正常退出状态 0 退出。如果不在 `--start` 模式下指定，则会引发错误。

如果存在 LSN 等于 lsn 的记录，则该记录将被输出。

`--endpos` 选项不知道事务边界，可能会在事务的中途截断输出。任何部分输出的事务都不会被消耗，并将在下次从插槽读取时再次重放。单个消息永远不会被截断。

`-f filename`  
`--file=filename`

将接收到和解码的事务数据写入此文件中。使用 `-` 表示 `stdout`。

`-F interval_seconds`  
`--fsync-interval=interval_seconds`

指定 `ltdts_recvlogical` 应该多久发出一次 `fsync()` 调用，以确保输出文件被安全地刷新到磁盘。

服务器偶尔会要求客户端执行刷新操作，并将刷新位置报告给服务器。此设置除了执行更频繁的刷新操作之外，还要进行刷新操作。

指定间隔为 0 将完全禁用 `fsync()` 调用，但仍会向服务器报告进度。在这种情况下，在崩溃事件中可能会丢失数据。

`-I lsn`  
`--startpos=lsn`

在 `--start` 模式下，从给定的 LSN 开始进行复制。有关此的详细信息，请参见 Chapter 45 和 Section 49.4 中的文档。在其他模式下忽略。

`--if-not-exists`

在指定 `--create-slot` 并且使用指定名称的插槽已经存在时，不会出错。

`-n`  
`--no-loop`

连接到服务器时，如果失去连接，则不要在循环中重试，只需退出。

`-o name[=value]`  
`--option=name[=value]`

将选项 `name` 和（如果指定）选项值 `value` 传递给输出插件。存在哪些选项以及它们的效果取决于所使用的输出插件。

`-P plugin`  
`--plugin=plugin`

在创建插槽时，使用指定的逻辑解码输出插件。请参见 Chapter 45 如果插槽已经存在，则此选项不起作用。

`-s interval_seconds`  
`--status-interval=interval_seconds`

此选项与 `lt_receivewal` 中同名选项具有相同的效果。请参见那里的描述。

`-S slot_name`  
`--slot=slot_name`

在 `--start` 模式下，使用名为 `slot_name` 的现有逻辑复制插槽。在 `--create-slot` 模式下，创建具有此名称的插槽。在 `--drop-slot` 模式下，删除具有此名称的插槽。

`-v`  
`--no-verbose`

禁用详细模式。

以下命令行选项控制数据库连接参数。

`-d dbname`  
`--dbname=dbname`

要连接的数据库。有关详细信息，请参见操作的描述。 `dbname` 可以是连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。默认为用户名。

`-h hostname-or-ip`  
`--host=hostname-or-ip`

指定运行服务器的计算机的主机名。如果值以斜杠开头，则将其用作 Unix 域套接字的目录。如果设置了 `LTHOST` 环境变量，则使用默认值，否则尝试进行 Unix 域套接字连接。

`-p port`  
`--port=port`

指定服务器用于侦听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。如果设置了 `LTPORT` 环境变量，则默认为该环境变量的值，否则使用编译时的默认值。

`-U user`  
`--username=user`

要连接的用户名称。默认为当前操作系统用户名称。

`-w`  
`--no-password`

永远不提示输入密码。如果服务器需要密码身份验证，并且没有其他方式（例如 `.pgpass` 文件）提供密码，则连接尝试将失败。此选项可用于批处理作业和脚本，其中没有用户输入密码。

`-W`  
`--password`

强制 `ltdts_recvlogical` 在连接到数据库之前提示输入密码。

此选项从未是必要的，因为如果服务器要求密码身份验证，`ltdts_recvlogical` 将自动提示输入密码。然而，`ltdts_recvlogical` 将浪费一次连接尝试来发现服务器需要密码。在某些情况下，值得输入 `-W` 以避免额外的连接尝试。

`-T target_type`  
`--target=target_type`

同步到何处。`only-debug|oracle|lightdb|redis|kafka|es|mysql`。LightDB 22.2 仅支持 `only-debug`（不插入 Oracle，只进行 JSON 解析）| Oracle 目标。

以下命令行选项控制 Oracle 数据库连接参数。

`-t tnsname`  
`--tnsname=TNSNAME`

要连接的 Oracle `tnsname`。

`-u Oracle username`  
`--oracle-username=NAME`

指定连接到 Oracle 的用户名。

`-k Oracle password`  
`--oracle-password=PASSWORD`

指定连接到 Oracle 的密码。

--ignore-duplicate

在插入到 Oracle 时忽略重复错误。

--ignore-not-found

在更新或删除 Oracle 时忽略 0 行受影响。

-z date format

--date\_format=DATE\_FORMAT

nls\_date\_format, 默认为 YYYY-MM-DD HH24:MI:SS。

-Z date format

--timestamp\_format=TIMESTAMP\_FORMAT

nls\_timestamp\_format, 默认为 YYYY-MM-DD HH24:MI:SS.FF6。

以下是可用的其他选项:

-V

--version

打印 ltdts\_recvlogical 的版本并退出。

-?

--help

显示关于 ltdts\_recvlogical 命令行参数的帮助信息, 并退出。

## 环境变量

与大多数其他 LightDB 实用程序一样, 此实用程序使用由 libpq 支持的环境变量 (参见 Section 32.14)。

环境变量 LT\_COLOR 指定是否在诊断消息中使用颜色。可能的值为 always、auto 和 never。

## 注意事项

如果源集群启用了组权限, 则 ltdts\_recvlogical 将保留接收到的 WAL 文件的组权限。

## 另请参阅

lt\_receivewal

---

# ltdts\_ora

ltdts\_ora — Ltdts\_ora用于将数据从Oracle同步到LightDB。

## Synopsis

```
java -jar ltdts_ora.jar [option...] [ 示例: java -jar ltdts_ora.jar --
server.port=7777 --oh=192.168.1.1:1521/test --ou=XSTRM --op=dbz --oa=DBZXOUT --
lh=192.168.1.2:7000/postgres --lu=lightdb --lp=lightdb123 ]
```

## Description

ltdts\_ora用于将数据从Oracle同步到LightDB。 在运行ltdts\_ora.jar之前，您要先完成以下准备工作。

```
# 打开Oracle归档日志并启用XStream。
sqlplus /nolog
CONNECT sys/password@host:port AS SYSDBA;

alter system set db_recovery_file_dest_size = 100G;
alter system set db_recovery_file_dest = '/opt/oracle/oradata/recovery_area'
scope=spfile;
alter system set enable_goldengate_replication=true;
shutdown immediate;
startup mount;
alter database archivelog;
alter database open;
-- select status from v$instance;
-- 应显示 "Database log mode: Archive Mode"
archive log list;

exit;

# 创建XStream管理员用户
sqlplus sys/password@host:port/SID as sysdba

CREATE TABLESPACE xstream_adm_tbs DATAFILE '/opt/oracle/oradata/orcl/
xstream_adm_tbs.dbf' SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
CREATE USER xstrmadmin IDENTIFIED BY dbz DEFAULT TABLESPACE xstream_adm_tbs
QUOTA UNLIMITED ON xstream_adm_tbs;
GRANT CREATE SESSION TO xstrmadmin;
BEGIN
  DBMS_XSTREAM_AUTH.GRANT_ADMIN_PRIVILEGE(
    grantee          => 'xstrmadmin',
    privilege_type    => 'CAPTURE',
    grant_select_privileges => TRUE,
    container         => 'ALL'
  );
END;

# 创建连接器的XStream用户

CREATE TABLESPACE xstream_tbs DATAFILE '/opt/oracle/oradata/orcl/
xstream_tbs.dbf' SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
```

```

CREATE USER xstrm IDENTIFIED BY dbz DEFAULT TABLESPACE xstream_tbs QUOTA
  UNLIMITED ON xstream_tbs;
GRANT CREATE SESSION TO xstrm;
GRANT SELECT ON V_$DATABASE to xstrm;
GRANT FLASHBACK ANY TABLE TO xstrm;
GRANT SELECT ANY TABLE to xstrm;
GRANT LOCK ANY TABLE TO xstrm;
grant select_catalog_role to xstrm;
GRANT EXECUTE_CATALOG_ROLE TO xstrm;

# alter database add supplemental log data (all) columns;
alter database add supplemental log data (primary key, unique index) columns;

exit;

# 创建XStream出站服务器
sqlplus xstrmadmin/password@host:port/SID

DECLARE
tables DBMS_UTILITY.UNCL_ARRAY;
schemas DBMS_UTILITY.UNCL_ARRAY;
BEGIN
  tables(1) := NULL;
  schemas(1) := 'XSTRM' ;
  DBMS_XSTREAM_ADM.CREATE_OUTBOUND(
    server_name => 'dbzxout',
    table_names => tables,
    schema_names => schemas
  );
END;
/

exit;

# 配置XStream用户帐户以连接到XStream出站服务器
sqlplus sys/password@host:port/SID as sysdba

BEGIN
  DBMS_XSTREAM_ADM.ALTER_OUTBOUND(
    server_name => 'dbzxout',
    connect_user => 'xstrm'
  );
END;
/

exit;

```

## Options

必须至少指定以下选项之一才能选择操作：

--oh

指定Oracle的主机、Oracle的端口和Oracle的数据库名称。  
192.168.1.1:1521/ORCL。

格式类似于



--ou

要连接的Oracle用户名。

--op

Oracle用户密码。

--oa

Oracle XStream out应用程序名称。

--m

数据同步格式模式，选项为{SQL, JSON}，默认值为SQL。

--sd

您要在哪里同步数据，选项为{LT, FILE}，默认值为LT。 [提示：LT是LightDB的缩写]

--lh

指定LightDB的主机、LightDB的端口和LightDB的数据库名称。 格式类似于  
192.168.1.1:5432/postgres。

--lu

要连接的LightDB用户名。

--lp

LightDB用户密码。

--bl

不要同步的黑名单表，如果有多个，请用逗号分隔。

## See Also

`ltdts_recvlogical`

---

# ltdts\_mysql

ltdts\_mysql (Experiment) — ltdts\_mysql用于将MySQL / MariaDB中的数据同步到LightDB。

## Synopsis

```
java -jar ltdts_mysql.jar [option...]
```

例如:

```
java -jar ltdts_mysql.jar --server.port=8888 --zk=127.0.0.1:2181 --  
kafka=127.0.0.1:9092 --db=database --lh=127.0.0.1:5432/test --lu=lightdb --  
lp=lightdb123 --bl=table1, table2
```

## 描述

ltdts\_mysql 用于将MySQL / MariaDB中的数据同步到LightDB。 在运行ltdts\_mysql.jar之前，您应该首先完成以下准备工作。

## Zookeeper

您可以从以下网址下载: <https://d1cdn.apache.org/zookeeper/zookeeper-3.8.0/apache-zookeeper-3.8.0-bin.tar.gz>

```
# 解压缩包  
tar -zxvf apache-zookeeper-3.8.0-bin.tar.gz  
# 创建一个新目录以存储zookeeper数据  
mkdir zk-data  
cd conf/  
# 复制文件  
cp zoo_sample.cfg zoo.cfg  
# 编辑文件  
vim zoo.cfg  
dataDir=/data/lightdb/zm/apache-zookeeper-3.8.0-bin/zk-data  
  
cd apache-zookeeper-3.8.0-bin/bin  
# 启动zookeeper服务器  
./zkServer.sh start
```

## Kafka

您可以从以下网址下载Kafka: [https://downloads.apache.org/kafka/3.3.1/kafka\\_2.12-3.3.1.tgz](https://downloads.apache.org/kafka/3.3.1/kafka_2.12-3.3.1.tgz)

您可以从以下网址下载Debezium for mysql: <https://repo1.maven.org/maven2/io/debezium/debezium-connector-mysql/1.9.7.Final/debezium-connector-mysql-1.9.7.Final-plugin.tar.gz>

```
# 解压Kafka包
```

```
tar -zxvf kafka_2.12-3.3.1.tgz
# 解压Debezium包
unzip debezium-debezium-connector-mysql-1.9.7.zip

# 创建一个新目录以存储kafka数据
mkdir kafka-data

# 创建一个新目录以存储kafka插件
mkdir kafka-plugin

# 将Debezium目录移动到kafka-plugin下
mv debezium-debezium-connector-mysql-1.9.7 kafka-plugin/

# 然后kafka目录如下所示
--kafka_2.12-3.3.1
  --config
  --bin
  --kafka-data
  --kafka-plugin
  --debezium-debezium-connector-mysql-1.9.7

# 编辑kafka的server.properties文件
vim server.properties
# 将以下值更改为
log.dirs=/data/lightdb/zm/kafka_2.12-3.3.1/kafka-data
# 将以下值更改为, 使用您的地址
listeners=PLAINTEXT://Your IP:9092

# 编辑kafka的connect-distributed.properties文件
vim connect-distributed.properties
# 将以下值更改为, 使用您的地址
bootstrap.servers=Your IP:9092
listeners=HTTP://Your IP:8083
# 将以下值更改为
plugin.path=/data/lightdb/zm/kafka_2.12-3.3.1/kafka-plugin

# 启动kafka
cd kafka_2.12-3.3.1/bin
./kafka-server-start.sh -daemon ../config/server.properties

# 启动kafka connect
./connect-distributed.sh -daemon ../config/connect-distributed.properties
```

## Debezium

```
# 首先检查Kafka Connect是否可用, 使用您的IP地址
curl -H "Accept:application/json" IP:8083/

# 列出所有Kafka连接器
curl -H "Accept:application/json" IP:8083/connectors

# 部署MySQL连接器
```

```
curl -i -X POST -H "Accept:application/json" -H "Content-Type:application/json"
IP:8083/connectors/ -d '{
  "name": "test-connector",
  "config": {
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "tasks.max": "1",
    "database.hostname": "mysql host, example:127.0.0.1",
    "database.port": "3306",
    "database.user": "mysql user",
    "database.password": "mysql password",
    "database.server.id": "184077",
    "database.server.name": "mysql database name, example:acm",
    "database.include.list": "mysql database name, example:acm",
    "database.history.kafka.bootstrap.servers": "your kafka IP and
port, example:127.0.0.1:9092",
    "database.history.kafka.topic": "schema-changes.acm",
    "include.schema.changes": "true",
    "decimal.handling.mode": "string",
    "include.schema.comments": "false"
  }
}'

# 删除MySQL连接器
curl -v -X DELETE IP:8083/connectors/test-connector

# 检查连接器的工作状态
curl -i IP:8083/connectors/test-connector/status

# 查看连接器的配置
curl -i IP:8083/connectors/test-connector/config
```

## Options

必须指定以下至少一个选项来选择操作：

`--zk`

指定Zookeeper的主机和端口。格式类似于127.0.0.1:2181。

`--kafka`

指定Kafka的主机和端口。格式类似于127.0.0.1:9092。

`--db`

您要同步的源数据库。它也可以被理解为Kafka主题，因为使用了Kafka。

`--lh`

指定LightDB的主机、端口和数据库名称的LightDB。格式类似于192.168.1.1:5432/postgres。

`--lu`

用于连接的LightDB用户名。

`--lp`

LightDB用户密码。

--bl

不要同步的黑名单表，如果有多个，用逗号分隔。

## See Also

[ltdts\\_recvlogical](#)

---

# ltldr

ltldr — 兼容 ORACLE，将外部文件中的数据加载到 LightDB 数据库表中。

## Synopsis

```
ltldr [connection-option...] [option...]
```

## 描述

ltldr 是一个小巧而灵活的工具，将外部文件中的数据加载到 LightDB 数据库表中。

ltldr 可以指定导入数据的格式，包括字段和字段、行和行之间的分隔符。

ltldr 目前支持的文件格式包括 txt 和 csv，也支持行定长的文件格式。

ltldr 支持将日志写入指定的文件。

## 选项

以下命令行选项控制导入的内容和格式。

user=

指定导入表的用戶。

dbname=

指定导入表的数据库名。

host=

指定导入表的主机名。

port=

指定导入表的端口号。

password=

指定导入表的密码。

control=

指定描述如何加载数据的控制文件的路径。

log=

指定用于存储有关加载过程的日志信息的日志文件录路径。

## 示例

连接到数据库，根据控制文件 LTLDR000003.ct1 将外部数据导入到数据库 testdb 中的表：

```
$ ltblr control=/home/lightdb/ltldrtest/LTLDR000003.ct1 host=127.0.0.1  
port=5432 userid=lightdb password=111111 dbname=testdb
```

## 控制文件语法

```
[ OPTIONS ( option [ , ... ] ) ]
LOAD DATA
[ CHARACTERSET charset ]
INFILE 'filename' [ "STR 'char'" ] [ "FIX count" ]
[ APPEND | INSERT | REPLACE ]
INTO TABLE tablename
[ WHEN condition ]
[ FIELDS [CSV] TERMINATED BY 'char' ]
[ [ OPTIONALLY ] ENCLOSED BY 'char' ]
[ TRAILING NULLCOLS ]
(
  column1 constant '321',
  column2 position(5:10),
  column3 position(12:14),
  ignore1 FILLER
  column4 FILLER
  [ ... ]
)
```

### OPTIONS

支持的option有skip和load，skip表示跳过多少行，load表示最多导入多少行。

### CHARACTERSET

数据文件字符集，需要保证控制文件(从环境获取字符集)和数据文件的字符集一致。

### INFILE

指定导入的数据文件路径

### STR

指定行分隔符，默认是'\n'，支持多字节和十六进制，可选项

### FIX

指定某一固定长度的数据来当做一行处理，可选项；

### APPEND | INSERT | REPLACE

APPEND 追加；INSERT 插入，只能导入到空表中；REPLACE 替换，导入之前先清空表

### INTO TABLE

指定导入的表名

### WHEN

判断某一行是否可以导入，支持等于和不等操作，可选。支持列名或者字符范围判断；

### CSV

表示使用CSV格式解析数据文件，可选；

**TERMINATED BY**

指定列分割符，默认是'\t'，支持多字节和十六进制，可选项

**ENCLOSED BY**

当指定了CSV时，ENCLOSED BY表示字段包围符，只支持一个字节，可选；

**TRAILING NULLCOLS**

当数据文件列数目不够时，表中剩余的列为null，可选；

**constant**

表示字段使用固定值，用单引号引用常量，可选；

**position**

表示通过字符范围填充该字段，可选；

**FILLER**

表示该列值跳过，不填充数据，可选；



---

# ltuldr

ltuldr — 兼容 ORACLE, 从 LightDB 数据库获取表的所有信息。

## Synopsis

```
ltuldr [connection-option...] [option...]
```

## 描述

ltuldr 是一个小巧而灵活的工具, 可导出 LightDB 数据。它可以直接导出指定表的所有数据, 也可以使用 SQL 语句导出特定数据。

ltuldr 可以指定导出数据的格式, 包括字段和字段、行和行之间的分隔符。

ltuldr 目前支持的文件格式包括 txt 和 csv, 也可以直接将数据输出到标准设备。

ltuldr 支持将日志写入指定的文件。

## 选项

以下命令行选项控制输出的内容和格式。

user=

指定导出表的用戶。

dbname=

指定导出表的数据库名。

host=

指定导出表的主机名。

port=

指定导出表的端口号。

password=

指定导出表的密码。

query=

指定导出表的信息。您可以直接使用表名或使用 SQL 语句。

对于 ltuldr, 查询和连接信息是必需的。您还可以使用以下 SQL 参数来指定 SQL 语句。

sql=

指定 SQL 语句的文件地址, 应与连接信息一起使用。

如果 SQL 语句非常复杂或很长, 很难在命令行上指定它。此时, 您可以编写一个包含复杂 SQL 的 SQL 文件 (注意 SQL 语句末尾不应该有分号或反斜杠), 然后告诉用户从指定的 SQL 文件中读取复杂的 SQL 语句以进行数据导出。

sql 参数和 query 参数不能同时使用。

field=

指定导出文件中字段之间的分隔符。默认值为“，”。

字段分隔符通常为一个字符，但实际上可以是多个字符；此外，可见的可打印字符、字符串和不可见字符都可以用作分隔符。

record=

指定导出文件中行之间的分隔符。默认值为“\n”。

记录和文件分隔符的命名方式相同。

file=

指定导出文件的名称。如果文件名的第一个字符是加号 (+)，则表示添加到现有文件之后。否则，将生成一个新文件。如果已经存在同名文件，则现有文件将被覆盖。如果未指定，则默认为小写的“uldrdata.txt”。存储路径是运行 ltuldr 的当前目录。

在文件选项中有几个特殊的格式代码，可以用于指定动态文件名（在生成的文件名中包含日期信息）。合理使用这些格式代码可以满足一些特定需求，减轻脚本编写的工作量。  
 %Y、%y：表示当前时间的四位年份； %M、%m：表示当前时间的两位月份；  
 %D、%d：表示当前时间的两位日期； %W、%w：表示当前时间所在的星期中的天数；  
 %B、%b：500000 是一个批次，它还与选项 batch 相关。如果 batch 是 2，则每导出 100 万次，计数器就会增加一次。

log=

指定导出日志的文件。其他用法与文件参数相同，除了日志文件不允许使用 %B 和 %b。

batch=

指定文件中最大保存的批次，每 500000 个数据为一批。在分批写入文件时，文件名必须是动态的，最好添加 %B 和 %b。否则，原始文件中的数据将被覆盖。

## 示例

连接到数据库，查询 pg\_class 表并将结果写入默认文件 uldrdata.txt:

```
$ ltuldr user=lightdb dbname=postgres host=127.0.0.1 port=5432 password=1
query=pg_class
```

将结果写入指定文件 lightdb.txt:

```
$ ltuldr user=lightdb dbname=postgres host=127.0.0.1 port=5432 password=1
query=pg_class file=lightdb.txt
```

将日志文件写入动态文件 %Y\_%M\_%D.log:

```
$ ltuldr user=lightdb dbname=postgres host=127.0.0.1 port=5432 password=1
query=pg_class file=lightdb.txt log=%Y_%M_%D.log
```

将表中的所有字段写入文件 lightdb.txt:

```
$ ltuldr user=lightdb dbname=postgres host=127.0.0.1 port=5432 password=1  
query=pg_class file=lightdb.txt log=%Y_%M_%D.log head=yes
```

每次获取两个批次的的数据，并将其写入文件 lightdb.txt:

```
$ ltuldr user=lightdb dbname=postgres host=127.0.0.1 port=5432 password=1  
query=pg_class file=lightdb.txt log=%Y_%M_%D.log head=yes batch=2
```

---

# lt\_basebackup

lt\_basebackup — 获得一个LightDB集簇的一个基础备份

## Synopsis

```
lt_basebackup [option...]
```

## 描述

lt\_basebackup被用于获得一个正在运行的LightDB数据库集簇的基础备份。获得这些备份不会影响数据库的其他客户端，并且可以被用于时间点恢复（见Section 23.3 以及用作一个日志传送或流复制后备服务器的开始点（见Section 24.2）。

lt\_basebackup对数据库群集的文件进行精确复制，同时确保服务器自动进入和退出备份模式。备份总是从整个数据库集簇获得，不可能备份单个数据库或数据库对象。关于选择性备份，必须使用一个像lt\_dump的工具。

备份通过一个使用复制协议常规LightDB连接制作。该连接必须由一个具有REPLICATION权限（详见Section 19.2 或者具有超级用户权限的用户ID建立，并且lt\_hba.conf必须允许该复制连接。该服务器还必须被配置，使max\_wal\_senders设置得足够高以提供至少一个walsender用于备份以及一个用于WAL流（如果使用流）。

在同一时间可以有多个lt\_basebackup运行，但是从性能的角度来说，只进行一次备份并且复制结果通常更好。

lt\_basebackup不仅能从主控机也能从后备机创建一个基础备份。要从后备机获得一个备份，设置后备机让它能接受复制连接（也就是，设置max\_wal\_senders和hot\_standby，并且适当配置其lt\_hba.conf）。你将也需要在主控机上启用full\_page\_writes。

注意在来自后备机的备份中有一些限制：

- 不会在被备份的数据库集簇中创建备份历史文件。
- lt\_basebackup不能强制备用服务器在备份结束时切换到新的WAL文件。当正在使用-X none时，如果服务器上的写活动比较低，lt\_basebackup可能需要等待很长时间，以便切换和归档备份所需要的最后的WAL文件。在这种情况下，在主服务器上运行pg\_switch\_wal以触发立即的WAL文件切换可能是有用的。
- 如果在备份期间后备机被提升为主控机，备份会失败。
- 备份所需的所有WAL记录必须包含足够的全页写，这要求你在主控机上启用full\_page\_writes并且不使用一个类似pg\_compresslog的工具以archive\_command从WAL文件中移除全页写。

每当lt\_basebackup进行基本备份时，服务器的pg\_stat\_progress\_basebackup视图将报告备份的进度。有关详细信息，请参见Section 25.4.5

## 选项

下列命令行选项控制输出的位置和格式：

```
-D directory  
--pgdata=directory
```

设置目标目录以将输出写入。如果该目录不存在，lt\_basebackup将创建该目录（以及所有丢失的父目录）。如果已经存在，则必须为空。

当备份处于 `tar` 模式中，目标目录可以被指定为`-`（破折号），从而将 `tar` 文件写入 `stdout`。

这个选项是必需的。

`-F format`  
`--format=format`

为输出选择格式。format可以是下列之一：

`p`  
`plain`

把输出写成平面文件，使用和源服务器数据目录和表空间相同的布局。当集簇没有额外表空间时，整个数据库将被放在目标目录中。如果集簇包含额外的表空间，主数据目录将被放置在目标目录中，但是所有其他表空间将被放在它们位于源服务器上的相同的绝对路径中。（参见 `--tablespace-mapping` 来更改。）

这是默认格式。

`t`  
`tar`

将输出作为tar文件写入目标目录中。主数据目录的内容将被写入名为`base.tar`的文件，而其他表空间将被写入以该表空间的OID命名的单独的tar文件。

如果将目标目录指定为`-`（破折号），则tar内容将被写入标准输出，该标准输出适用于管道传输至`gzip`（例如）。只有当集簇没有额外表空间并且没有使用WAL流时才允许这样做。

`-R`  
`--write-recovery-conf`

创建一个`standby.signal`文件，并将连接设置附加到目标目录（或使用tar格式的基本存档文件中）的`lightdb.auto.conf`文件中。这样可以简化使用备份结果设置备用服务器的过程。

`lightdb.auto.conf`文件将记录连接设置（如果有）以及`lt_basebackup`所使用的复制槽，这样流复制后面就会使用相同的设置。

`-T olddir=newdir`  
`--tablespace-mapping=olddir=newdir`

在备份期间将目录`olddir`中的表空间重定位到`newdir`中。为使之有效，`olddir`必须与源服务器上定义的表空间的路径规范完全匹配。（但如果备份中没有包含`olddir`中的表空间也不是错误）。同时，`newdir`是接收主机文件系统中的目录。与主目标目录一样，`newdir`不必已经存在，但是如果确实存在，则必须为空。`olddir`和`newdir`必须是绝对路径。如果一个路径凑巧包含了一个`=`符号，可用反斜线对它转义。对于多个表空间可以多次使用这个选项。

如果以这种方法重定位一个表空间，主数据目录中的符号链接会被更新成指向新位置。因此新数据目录已经可以被一个所有表空间位于更新后位置的新服务器实例使用。

目前，这个选项仅以普通输出格式工作，如果选择了tar格式，请忽略它。

`--waldir=waldir`

指定用于预写式日志目录的位置。`waldir`必须是绝对路径。只有当备份是平面文件模式时才能指定事务日志目录。设置要写入WAL（预写日志）文件的目录。默认情况下，WAL文件将放置在目标目录的`lt_wal`子目录中，但是此选项可用于将它们放置在其他位置。`waldir`必须是绝对路径。与主目标目录一样，`waldir`不必已经存在，但是如果确实存在，则必须为空。只有当备份是平面文件模式时才可以指定此选项。

`-X method`  
`--wal-method=method`

在备份中包括所需的WAL（预写式日志）文件。这包括所有在备份期间产生的预写式日志。除非指定了方法none，可以在目标目录中启动postmaster而无需参考日志归档，从而使输出成为完全独立的备份。

支持下列收集预写式日志的方法：

n  
none

不要在备份中包括预写式日志。

f  
fetch

在备份末尾收集预写式日志文件。因此，有必要把源服务器的wal\_keep\_size参数设置得足够高，这样在备份末尾之前需要的日志数据不会被移除。如果所需的日志数据在传输之前已被回收，则备份将失败并且无法使用。

如果使用tar格式，预写式日志文件将被包含在base.tar文件。

s  
stream

在进行备份时，流式传输预写式日志数据。将开启一个到服务器的第二连接并且在运行备份时并行开始流传输预写式日志。因此，它将需要两个复制连接，而不仅仅是一个。只要客户端可以跟上预写式日志数据，使用此方法就不需要在源服务器上保存额外的预写式日志。

如果使用tar格式，预写式日志文件被写入到一个单独的名为lt\_wal.tar的文件（如果服务器的版本超过10，该文件将被命名为pg\_xlog.tar）。

这个值是默认值。

-z  
--gzip

启用对tar文件输出的gzip压缩，使用默认的压缩级别。只有使用tar格式时压缩才可用，并且会在所有tar文件名后面自动加上后缀.gz。

-Z level  
--compress=level

启用对tar文件输出的gzip压缩，并且制定压缩级别（0到9，0是不压缩，9是最佳压缩）。只有使用tar格式时压缩才可用，并且会在所有tar文件名后面自动加上后缀.gz。

下列命令行选项控制备份的生成和程序的调用：

-c fast|spread  
--checkpoint=fast|spread

将检查点模式设置为fast（立刻）或spread（默认）（见Section 23.3.3）。

-C  
--create-slot

指定在启动备份之前应创建由--slot选项命名的复制插槽。如果插槽已存在，则会引发错误。

-l label  
--label=label

为备份设置标签。如果没有指定，将使用一个默认值“lt\_basebackup base backup”。

-n  
--no-clean

默认情况下，当lt\_basebackup因为一个错误而中止时，它会把它意识到无法完成该工作之前已经创建的目录（例如目标目录和预写式日志目录）都移除。这个选项可以禁止这种清洗，因此可以用于调试。

注意不管哪一种方式都不会清除表空间目录。

-N  
--no-sync

默认情况下，lt\_basebackup将等待所有文件被安全地写到磁盘上。这个选项导致lt\_basebackup不做这种等待就返回，这样会更快一些，但是也意味着后续发生的操作系统崩溃可能会使得这个基础备份损坏。通常这个选项对测试比较有用，在创建生产安装时不应该使用。

-P  
--progress

启用进度报告。启用这个选项将在备份期间发表一个大致的进度报告。由于数据库可能在备份期间改变，这仅仅是一种近似并且可能不会刚好在100%结束。特别地，当WAL日志被包括在备份中时，总数据量无法预先估计，并且在这种情况中估计的目标尺寸会在它经过不带WAL的总估计后增加。

-r rate  
--max-rate=rate

设置从源服务器收集数据的最大传输速率。这有助于限制lt\_basebackup对服务器的影响。值以每秒千字节为单位。使用后缀M表示每秒兆字节数。后缀k也可以接受，没有任何影响。有效值介于每秒32千字节和每秒1024兆字节之间。

此选项始终影响数据目录的传输。只有收集方法为fetch时，才会影响WAL文件的传输。

-S slotname  
--slot=slotname

这个选项仅能与-X stream一起使用。它导致WAL流使用指定的复制槽。如果该基础备份的目的是被用作一台使用复制槽的流复制后备，则它应该使用与primary\_slot\_name中相同的复制槽名称。这可以确保主服务器不会移除位于该基础备份结束与新备用数据库上流复制开始之间产生的任何所需的WAL数据。

指定的复制槽必须已经存在，除非同时使用了选项-C。

如果这个选项没有被指定并且服务器支持临时复制槽（版本10以后），则会自动使用一个临时复制槽来进行WAL流。

-v  
--verbose

启用冗长模式。将在启动和关闭期间输出一些额外步骤，并且如果进度报告也被启用，还会显示当前正在被处理的确切文件名。

--manifest-checksums=algorithm

指定应用于备份清单中包含的每个文件的校验和算法。目前，可用的算法有NONE、CRC32C、SHA224、SHA256、SHA384和SHA512。默认值为CRC32C。

如果选择了NONE，备份清单将不包含任何校验和。否则，它将包含备份中使用指定算法的每个文件的校验和。此外，清单将始终包含自身内容的SHA256校验和。SHA算法比CRC32C算法占用大量CPU，因此选择其中一种算法可能会增加完成备份所需的时间。

使用SHA散列函数可为希望验证备份是否遭到篡改的用户提供每个文件的加密安全摘要，而CRC32C算法提供的校验和计算速度更快。它擅长捕获由于意外更改引起的错误，但不能抵抗恶意修改。请注意，为了对有权访问备份的对手有用，备份清单必须安全地存储在其他位置，否则必须经过验证以确保自从进行备份以来未进行任何修改。

lt\_verifybackup可用于根据备份清单检查备份的完整性。

`--manifest-force-encode`

强制备份清单中的所有文件名采用十六进制编码。如果未指定此选项，则仅对非UTF8文件名进行十六进制编码。此选项主要用于测试读取备份清单文件的工具是否正确处理此情况。

`--no-estimate-size`

阻止服务器估计将要流式传输的备份数据总量，从而导致pg\_stat\_progress\_basebackup视图中的backup\_total列始终为NULL。

如果没有此选项，则备份将从枚举整个数据库的大小开始，然后返回并发送实际内容。这可能会使备份花费的时间稍长一些，特别是在发送第一个数据之前，备份将花费更长的时间。如果估计时间过长，此选项将有助于避免此类估计时间。

使用`--progress`时不允许使用此选项。

`--no-manifest`

禁用备份清单的生成。如果未指定此选项，则服务器将生成并发送备份清单，可以使用lt\_verifybackup进行验证。清单是备份中存在的每个文件的列表，可能包含的所有WAL文件除外。它还存储大小、上次修改时间和每个文件的可选校验和。

`--no-slot`

防止为备份创建临时复制插槽。

默认情况下，如果选择了日志流，但没有用选项-S指定槽名称，则会创建一个临时复制插槽（如果源服务器支持）。

这个选项的主要目的是允许在服务器没有空闲复制槽可用时制作基础备份。使用复制槽几乎总是最好的方式，因为它能防止备份期间所需的WAL被删除。

`--no-verify-checksums`

如果在取基础备份的服务器上启用了校验码验证，则禁用校验码验证。

默认情况下，校验码会被验证并且校验码失败将会导致一种非零的退出状态。不过，基础备份在这种情况下将不会被移除，就好像使用了`--no-clean`选项一样。校验和验证失败也将报告在pg\_stat\_database视图中。

以下命令行选项控制到源服务器的连接：

`-d connstr`

`--dbname=connstr`

指定用于连接到服务器的参数，比如连接字符串；这些将覆盖所有冲突的命令行选项。

为了和其他客户端应用一致，该选项被称为`--dbname`。但是因为lt\_basebackup并不连接到集簇中的任何特定数据库，连接字符串中的任何数据库名将被忽略。



-h host  
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。默认值取自LTHOST环境变量（如果设置），否则会尝试一个 Unix 域套接字连接。

-p port  
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。默认用LTPORT环境变量中的值（如果设置），或者一个编译在程序中的默认值。

-s interval  
--status-interval=interval

指定发送回源服务器的状态包之间的秒数。较小的值可以更准确地监视服务器的备份进度。一个零值完全禁用这种周期性的状态更新，不过当服务器需要时还是会有一个更新会被发送来避免超时导致的断开连接。默认值是 10 秒。

-U username  
--username=username

指定连接的用户名。

-w  
--no-password

防止发出口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W  
--password

强制lt\_basebackup在连接到源服务器之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，lt\_basebackup将自动提示要求一个口令。但是，lt\_basebackup将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

其他选项也可用：

-V  
--version

打印lt\_basebackup版本并退出。

-?  
--help

显示有关lt\_basebackup命令行参数的帮助并退出。

## 环境

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

环境变量LT\_COLOR规定在诊断消息中是否使用颜色。可能的值为always、auto、never。

## 注解

在备份的开始时，需要在源服务器上执行检查点。这可能需要一点时间（尤其在未使用选项`--checkpoint=fast`时），在此期间`lt_basebackup`看起来处于闲置状态。

备份将包括数据目录和表空间中的所有文件，包括配置文件以及由第三方放在该目录中的任何额外文件，不过由LightDB管理的特定临时文件除外。但只有常规文件和目录会被拷贝，但用于表空间的符号链接会被保留。指向LightDB已知的特定目录的符号链接被拷贝为空白目录。其他符号链接和特殊设备文件会被跳过。准确的细节请参考Section 49.4

在普通格式中，表空间将备份到源服务器上的相同路径，除非使用了`--tablespace-mapping`选项。如果没有这个选项并且表空间正在使用，在同一台服务器上进行普通格式的基础备份将无法工作，因为备份必须要写入到与原始表空间相同的目录位置。

在使用 `tar` 格式时，用户应负责在启动使用数据的 LightDB 服务器前解压每一个 `tar` 文件。如果有额外的表空间，用于它们的 `tar` 文件需要被解压到正确的位置。在这种情况下，服务器将根据包含在`base.tar`文件中的`tablespace_map`文件的内容为那些表空间创建符号链接。

`lt_basebackup`可以和具有相同或较低主版本的服务器一起工作，最低是 9.1。但是，WAL 流模式（`-X` 流）只能和版本为 9.3 及以上版本的服务器一起工作。`tar` 格式（`--format=tar`）只能用于版本 9.5 及以上的服务器。

如果在源集簇上启用了组权限，`lt_basebackup`将保留数据文件的组权限。

## 例子

要创建服务器`mydbserver`的一个基础备份并将它存储在本地目录`/usr/local/lightdb-x/data`中：

```
$ lt_basebackup -h mydbserver -D /usr/local/lightdb-x/data
```

要创建本地服务器的一个备份，为其中每一个表空间产生一个压缩过的 `tar` 文件，并且将它存储在目录`backup`中，在运行期间显示一个进度报告：

```
$ lt_basebackup -D backup -Ft -z -P
```

要创建一个单一表空间本地数据库的备份并且使用`bzip2`压缩它：

```
$ lt_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

（如果在该数据库中有多个表空间，这个命令将失败）。

要创建一个本地数据库的备份，其中`/opt/ts`中的表空间被重定位到`./backup/ts`：

```
$ lt_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

## 参见

`lt_dump`

---

# lt\_config

lt\_config — 获取已安装的LightDB的信息

## Synopsis

```
lt_config [option...]
```

## 描述

lt\_config工具打印当前安装版本的LightDB的配置参数。它的设计目的之一是便于想与LightDB交互的软件包能够找到所需的头文件和库。

## 选项

要使用lt\_config，提供一个或多个下列选项：

--bindir

打印用户可执行文件的位置。例如使用这个选项来寻找ltsql程序。这通常也是lt\_config程序所在的位置。

--docdir

打印文档文件的位置。

--htmldir

打印 HTML 文档文件的位置。

--includedir

打印客户端接口的 C 头文件的位置。

--pkgincludedir

打印其它 C 头文件的位置。

--includedir-server

打印用于服务器编程的 C 头文件的位置。

--libdir

打印对象代码库的位置。

--pkglibdir

打印动态可载入模块的位置，或者服务器可能搜索它们的位置（其它架构独立数据文件可能也被安装在这个目录）。

--localedir

打印区域支持文件的位置（如果在LightDB被编译时没有配置区域支持，这将是一个空字符串）。

--mandir

打印手册页的位置。

`--sharedir`

打印架构独立支持文件的位置。

`--sysconfdir`

打印系统范围配置文件的位置。

`--pgxs`

打印扩展 `makefile` 的位置。

`--configure`

打印当LightDB被配置编译时给予`configure`脚本的选项。这可以被用来重新得到相同的配置，或者找出是哪个选项编译了一个二进制包（不过注意二进制包通常包含厂商相关的自定补丁）。参见下面的例子。

`--cc`

打印用来编译LightDB的CC变量值。这显示被使用的 C 编译器。

`--cppflags`

打印用来编译LightDB的CPPFLAGS变量值。这显示在预处理时需要的 C 编译器开关（典型的是-I开关）。

`--cflags`

打印用来编译LightDB的CFLAGS变量值。这显示被使用的 C 编译器开关。

`--cflags_sl`

打印用来编译LightDB的CFLAGS\_SL变量值。这显示被用来编译共享库的额外 C 编译器开关。

`--ldflags`

打印用来编译LightDB的LDFLAGS变量值。这显示链接器开关。

`--ldflags_ex`

打印用来编译LightDB的LDFLAGS\_EX变量值。这只显示被用来编译可执行程序的链接器开关。

`--ldflags_sl`

打印用来编译LightDB的LDFLAGS\_SL变量值。这只显示被用来编译共享库的链接器开关。

`--libs`

打印用来编译LightDB的LIBS变量值。这通常包含用于链接到LightDB中的外部库的-l开关。

`--version`

打印LightDB的版本。

`-?`

`--help`

显示有关`lt_config`命令行参数的帮助并退出。

如果给定多于一个选项，将按照相同的顺序打印信息，每行一项。如果没有给定选项，将打印所有可用信息，并带有标签。

## 注解

选项`--docdir`、`--pkgincludedir`、`--localedir`、`--mandir`、`--sharedir`、`--sysconfdir`、`--cc`、`--cppflags`、`--cflags`、`--cflags_sl`、`--ldflags`、`--ldflags_sl`和`--libs`在LightDB 8.1 被加入。选项`--htmldir`在LightDB 8.4 被加入。选项`--ldflags_ex`在LightDB 9.0 被加入。

## 例子

要重建当前 LightDB 安装的编译配置，可运行下列命令：

```
eval `./configure `lt_config --configure`
```

`lt_config --configure`的输出包含 shell 引号，这样带空格的参数可以被正确地表示。因此，为了得到正确的结果需要使用`eval`。

---

# lt\_dump

lt\_dump — 把LightDB数据库抽取为一个脚本文件或其他归档文件

## Synopsis

```
lt_dump [connection-option...] [option...] [dbname]
```

## 描述

lt\_dump是用于备份一种LightDB数据库的工具。即使数据库正在被并发使用，它也能创建一致的备份。lt\_dump不阻塞其他用户访问数据库（读取或写入）。

lt\_dump只转储单个数据库。要备份一个集簇或者集簇中 对于所有数据库公共的全局对象（例如角色和表空间），应使用 lt\_dumpall。

转储可以被输出到脚本或归档文件格式。脚本转储是包含 SQL 命令的纯文本文件，它们可以用来重构数据库到它被转储时的状态。要从这样一个脚本恢复，将它喂给ltsql。脚本文件甚至可以被用来在其他机器和其他架构上重构数据库。在经过一些修改后，甚至可以在其他SQL 数据库产品上重构数据库。

另一种可选的归档文件格式必须与lt\_restore配合使用来重建数据库。它们允许lt\_restore能选择恢复什么，或者甚至在恢复之前对条目重排序。归档文件格式被设计为在架构之间可移植。

当使用归档文件格式之一并与lt\_restore组合时，lt\_dump提供了一种灵活的归档和传输机制。lt\_dump可以被用来备份整个数据库，然后lt\_restore可以被用来检查归档并/或选择数据库的哪些部分要被恢复。最灵活的输出文件格式是“自定义”格式（-Fc）和“目录”格式（-Fd）。它们允许选择和重排序所有已归档项、支持并行恢复并且默认是压缩的。“目录”格式是唯一一种支持并行转储的格式。

lt\_dump 支持转储 plorasql 包和过程，就像转储表等一样。

当运行lt\_dump时，我们应该检查输出中有没有任何警告（打印在标准错误上），特别是考虑到下面列出的限制。

## 选项

下列命令选项控制输出的内容和格式。

dbname

指定要被转储的数据库名。如果没有指定，将使用环境变量LTDATABASE。如果环境变量也没有设置，则使用指定给该连接的用户名。

-a  
--data-only

只转储数据，而不转储模式（数据定义）。表数据、大对象和序列值都会被转储。

这个选项类似于指定--section=data，但是由于历史原因又不完全相同。

-b  
--blobs

在转储中包括大对象。这是当--schema、--table或--schema-only被指定时的默认行为。因此，只有在请求转储一个特定方案或者表的情况下，-b开关才对向转储中加入大

对象有用。注意blobs是被考虑的数据，因此在使用--data-only时将被包括在内，但在使用--schema-only时则不会包括。

-B  
--no-blobs

在转储中排除大对象。

当同时给定-b和-B时，行为是在数据被转储时输出大对象，请参考-b文档。

-c  
--clean

在输出创建数据库对象的命令之前输出清除（删除）它们的命令（除非也指定了--if-exists，如果任何对象不存在于目的数据库中，恢复可能会产生一些伤害性的错误消息）。

在生成档案（非文本）输出文件时，此选项将被忽略。对于档案格式，您可以在调用lt\_restore时指定该选项。

-K  
--recreate-schema

在输出创建模式命令之前，输出直接删除模式的命令（使用级联模式），比正常模式更快。（除非还指定了--if-exists，否则如果目标数据库中不存在任何对象，则还原可能会生成一些无害的错误消息。）

当发出归档(非文本)输出文件时，这个选项被忽略。对于归档格式，你可以在调用lt\_restore时指定该选项。

-C  
--create

使得在输出的开始是一个创建数据库本身并且重新连接到被创建的数据库的命令（通过这种形式的一个脚本，在运行脚本之前你连接的是目标安装中的哪个数据库都没有关系）。如果也指定了--clean，脚本会在重新连接到目标数据库之前先删除它然后再重建。

通过--create，输出还会包括数据库的注释（如果有）以及与这个数据库相关的任何配置变量设置，也就是任何提到了这个数据库的ALTER DATABASE ... SET ... 命令和ALTER ROLE ... IN DATABASE ... SET ... 命令。该数据库本身的访问特权也会被转储，除非指定有--no-acl。

当发出归档(非文本)输出文件时，这个选项被忽略。对于归档格式，你可以在调用lt\_restore时指定该选项。

Lightdb 23.3引入了一个新的数据库级别GUC参数lightdb\_ascii\_zero\_store\_value，此值存储于表 lt\_database\_ext中，如果我们运行lt\_dump -C（或运行 lt\_dumpall），此选项将会出现在数据库创建语句中。

更多关于GUC `lightdb_ascii_zero_store_value` 的信息，请参考 `lightdb_ascii_zero_store_value`

-E encoding  
--encoding=encoding

以指定的字符集编码创建转储。在默认情况下，该转储会以该数据库的编码创建（另一种得到相同结果的方式是将LTCLIENTENCODING环境变量设置成想要的转储编码）。支持的编码在 Section 21.3. 中介绍。

-f file  
--file=file

将输出发送到指定文件。对于基于输出格式的文件这个参数可以被忽略，在那种情况下将使用标准输出。不过对于目录输出格式必须给定这个参数，在目录输出格式中指定的是一个目录而不是一个文件。在这种情况下，该目录会由lt\_dump创建并且不需要以前就存在。

-F format  
--format=format

选择输出的格式。format可以是下列之一：

p  
plain

输出一个纯文本形式的SQL脚本文件（默认值）。

c  
custom

输出一个适合于作为lt\_restore输入的自定义格式归档。和目录输出格式一起，这是最灵活的输出格式，它允许在恢复时手动选择和排序已归档的项。这种格式在默认情况还会被压缩。

d  
directory

输出一个适合作为lt\_restore输入的目录格式归档。这将创建一个目录，其中每个被转储的表和大对象都有一个文件，外加一个所谓的目录文件，该文件以一种lt\_restore能读取的机器可读格式描述被转储的对象。一个目录格式归档能用标准Unix 工具操纵，例如一个未压缩归档中的文件可以使用gzip工具压缩。这种格式默认情况下是被压缩的并且也支持并行转储。

t  
tar

输出一个适合于输入到lt\_restore中的tar-格式归档。tar 格式可以兼容目录格式，抽取一个 tar 格式的归档会产生一个合法的目录格式归档。不过，tar 格式不支持压缩。还有，在使用 tar 格式时，表数据项的相对顺序不能在恢复过程中被更改。

-j njobs  
--jobs=njobs

通过同时归档njobs个表来运行并行转储。这个选项可能会减少执行转储所需的时间，但也会增加数据库服务器上的负载。你只能和目录输出格式一起使用这个选项，因为这是唯一一种让多个进程能在同一时间写其数据的输出格式。

lt\_dump将打开njobs + 1 个到该数据库的连接，因此确保你的max\_connections设置足够高以容纳所有的连接。

在运行一次并行转储时请求数据库对象上的排他锁可能导致转储失败。其原因是，lt\_dump领导者进程会在工作者进程将要稍后转储的对象上请求共享锁，以便确保在转储运行时不会有人删除它们并让它们出错。如果另一个客户端接着请求一个表上的排他锁，那个锁将不会被授予但是会被排入队列等待领导者进程的共享锁被释放。因此，任何其他对该表的访问将不会被授予或者将排在排他锁请求之后。这包括尝试转储该表的工作者进程。如果没有任何防范措施，这可能会是一种经典的死锁情况。要检测这种冲突，lt\_dump工作者进程使用NOWAIT选项请求另一个共享锁。如果该工作者进程没有被授予这个共享锁，其他某人必定已经在同时请求了一个排他锁并且没有办法继续转储，因此lt\_dump除了中止转储之外别无选择。



对于一个一致的备份，数据库服务器需要支持同步的快照，在LightDB 9.2的主服务器和10的后备服务器中引入了一种特性。有了这种特性，即便数据库客户端使用不同的连接，也可以保证他们看到相同的数据集。lt\_dump -j使用多个数据库连接，它用领导者进程连接到数据一次，并且为每一个工作者任务再一次连接数据库。如果没有同步快照特征，在每一个连接中不同的工作者任务将不能被保证看到相同的数据，这可能导致一个不一致的备份。

如果你希望运行一个 9.2 之前服务器的并行转储，你需要确保数据库内容从领导者连接到数据库一直到最后一个工作者任务连接到数据库之间不会改变。做这些最简单的方法是在开始备份之前停止任何访问数据库的数据修改进程（DDL 以及 DML）。当对一个 9.2 之前的LightDB服务器运行lt\_dump -j时，你还需要指定--no-synchronized-snapshots参数。

-n pattern  
--schema=pattern

只转储匹配pattern的模式，这会选择模式本身以及它所包含的所有对象。当没有指定这个选项时，目标数据库中所有非系统模式都将被转储。多个模式可以通过书写多个-n开关来选择。另外，pattern参数可以被解释为一种根据ltsql's \d命令所用的相同规则（请参见下面的Patterns）编写的模式，这样多个模式也可以通过在该模式中书写通配字符来选择。在使用通配符时，如果需要阻止 shell 展开通配符需要小心引用该模式，请参见下面的Examples。

#### Note

当-n被指定时，lt\_dump不会尝试转储所选模式可能依赖的任何其他数据库对象。因此，无法保证一次指定模式转储的结果能够仅凭其本身被成功地恢复到一个干净的数据库中。

#### Note

当-n被指定时，非模式对象（如二进制大对象）不会被转储。你可以使用--blobs开关将二进制大对象加回到该转储中。

-N pattern  
--exclude-schema=pattern

不转储匹配pattern模式的任何模式。该模式被根据-n所用的相同规则被解释。-N可以被给定多次来排除匹配几个模式中任意一个的模式。

当-n和-N都被给定时，该行为是只转储匹配至少一个-n开关但是不匹配-N开关的模式。如果只有-N而没有-n，那么匹配-N的模式会被从一个正常转储中排除。

-0  
--no-owner

不输出设置对象拥有关系来匹配原始数据库的命令。默认情况下，lt\_dump会发出ALTER OWNER或SET SESSION AUTHORIZATION语句来设置被创建的数据库对象的拥有关系。除非该脚本被一个超级用户（或是拥有脚本中所有对象的同一个用户）启动，这些语句都会失败。要使一个脚本能够被任意用户恢复，但把所有对象的拥有关系都给这个用户，可指定-0。

当发出归档(非文本)输出文件时，这个选项被忽略。对于归档格式，你可以在调用lt\_restore时指定该选项。

-R  
--no-reconnect

这个选项已经废弃，但是为了向后兼容仍然能被接受。

-s  
--schema-only

只转储对象定义（模式），而非数据。

这个选项是--data-only的逆选项。它和指定--section=pre-data --section=post-data相似，但是由于历史原因又不完全相同。

（不要把这个选项和--schema选项混淆，后者在“schema”的使用上有不同的含义）。

要为数据库中表的一个子集排除表数据，见--exclude-table-data。

-S username  
--superuser=username

指定要在禁用触发器时使用的超级用户的用户名。只有使用--disable-triggers时，这个选项才相关（通常，最好省去这个选项，而作为超级用户来启动结果脚本来取而代之）。

-t pattern  
--table=pattern

只转储名字匹配pattern的表。通过写多个-t开关可以选择多个表。另外，pattern参数可以被解释为一种根据ltsql's \d命令所用的相同规则（请参见下面的Patterns）编写的模式，这样多个表也可以通过在该模式中书写通配字符来选择。在使用通配符时，如果需要阻止 shell 展开通配符需要小心引用该模式，请参见下面的Examples。

除了表之外，此选项还可用于转储匹配视图、物化视图、外部表和序列的定义。它不会转储视图或物化视图的内容，并且只有在使用 --include-foreign-data 指定相应的外部服务器时，外部表的内容才会被转储。

当-t被使用时，-n和-N开关不会有效果，因为被-t选择的表将被转储而无视那些开关，并且非表对象将不会被转储。

### Note

当-t被指定时，lt\_dump不会尝试转储所选表可能依赖的任何其他数据库对象。因此，无法保证一次指定表转储的结果能够仅凭其本身被成功地恢复到干净的一个数据库中。

### Note

选项 -t 的行为与 8.2 版本之前的 LightDB 版本不完全向上兼容。以前，写 -t tab 将转储名为 tab 的所有表，但现在它只转储在默认搜索路径中可见的那个表。要获得旧的行为，可以写 -t '\*.\*tab'。此外，您必须写类似于 -t sch.tab 的内容以选择特定模式中的表，而不是旧的使用法 -n sch -t tab。

-T pattern  
--exclude-table=pattern

不转储匹配pattern模式的任何表。该模式被根据-t所用的相同规则被解释。-T可以被给定多次来排除匹配几个模式中任意一个的模式。

当-t和-T都被给定时，该行为是只转储匹配至少一个-t开关但是不匹配-T开关的表。如果只有-T而没有-t，那么匹配-T的表会被从一个正常转储中排除。

-v  
--verbose

指定冗长模式。这将导致lt\_dump向标准错误输出详细的对象注释以及转储文件的开始/停止时间，还有进度消息。重复该选项会导致在标准错误上出现附加的调试级消息。

-V  
--version

lt\_dump版本并退出。

-x  
--no-privileges  
--no-acl

防止转储访问特权（授予/收回命令）。

-Z 0..9  
--compress=0..9

指定要使用的压缩级别。零意味着不压缩。对于自定义和目录归档格式，这会指定个体表数据段的压缩，并且默认是进行中等级别的压缩。对于纯文本输出，设置一个非零压缩级别会导致整个输出文件被压缩，就好像它被gzip处理过一样，但是默认是不压缩。tar 归档格式当前完全不支持压缩。

--binary-upgrade

这个选项用于就地升级功能。我们不推荐也不支持把它用于其他目的。这个选项在未来的发行中可能被改变而不做通知。

--column-inserts  
--attribute-inserts

将数据转储为带有显式列名的INSERT命令（INSERT INTO table (column, ...) VALUES ...）。这将使得恢复过程非常慢，这主要用于使转储能够被载入到非LightDB数据库中。重新加载期间的任何错误都将导致有问题的INSERT相关的行将丢失，而不是整个表内容。

--disable-dollar-quoting

这个选项禁止在函数体中使用美元符号引用，并且强制它们使用 SQL 标准字符串语法被引用。

--disable-triggers

只有在创建一个只转储数据的转储时，这个选项才相关。它指示lt\_dump包括在数据被重新载入时能够临时禁用目标表上的触发器的命令。如果你在表上有引用完整性检查或其他触发器，并且你在数据重新载入期间不想调用它们，请使用这个选项。

当前，为--disable-triggers发出的命令必须作为超级用户来执行。因此，你还应当使用-S指定一个超级用户名，或者宁可作为一个超级用户启动结果脚本。

当发出归档(非文本)输出文件时，这个选项被忽略。对于归档格式，你可以在调用lt\_restore时指定该选项。

--enable-row-security

只有在转储具有行安全性的表的内容时，这个选项才相关。默认情况下，lt\_dump将把row\_security设置为 off 来确保从该表中转储 出所有的数据。如果用户不具有足够能

绕过行安全性的特权，那么会抛出一个错误这个参数指示lt\_dump将 row\_security设置为 on，允许用户只转储该表中 它们能够访问到的部分内容。

注意如果当前你使用了这个选项，你可能还想得到INSERT格式的转储，因为恢复期间的COPY FROM不支持行安全性。

`--exclude-table-data=pattern`

不转储匹配pattern模式的任何表中的数据。该模式根据-t的相同规则被解释。`--exclude-table-data`可以被给定多次来排除匹配多个模式的表。当你需要一个特定表的定义但不想要其中的数据时，这个选项就有用了。

要排除数据库中所有表的数据，见`--schema-only`。

`--extra-float-digits=ndigits`

在转储浮点数据时使用规定的extra\_float\_digits值，而不是最大可用精度。以备份目的生成的常规转储不使用此选项。

`--if-exists`

时间条件性命令（即增加一个IF EXISTS子句）来清除数据库和其他对象。只有同时指定了`--clean`时，这个选项才可用。

`--include-foreign-data=foreignserver`

使用与foreignserver模式匹配的外部服务器转储任何外部表的数据。可以通过编写多个`--include-foreign-data`开关来选择多个外部服务器。同样，根据ltsql's \d命令使用的相同规则，将foreignserver参数解释为模式。（请参见下面的Patterns），因此也可以通过在模式中写入通配符来选择多个外部服务器。使用通配符时，如果需要，请小心引用该模式，以防止Shell扩展通配符。请参见下面的Examples。唯一的例外是不允许使用空模式。

### Note

指定`--include-foreign-data`时，lt\_dump不会检查外部表是否可写。因此，不能保证可以成功还原外部表转储的结果。

`--lt-target-version`

指定导出时目标 lightdb 的版本。

`--lt-exclude-lightdb-objects`

在转储中排除 LightDB 创建的所有内置对象。

### Note

对于 LightDB 创建的内置表或模式，当同时给出 -t 或 -n 时，在转储数据时忽略表或模式，请参见 -t 和 -n 的文档。

`--lt-dump-lightdb-tables`

仅转储在“pg\_extension\_config\_dump”中指定的内置表的数据（在扩展的 SQL 脚本中）。

这些表在还原数据时首先将被截断。

### Note

当同时给出 `-s` 或 `--section` 时，在转储数据时忽略表数据。

对于 LightDB 创建的内置表，当给出 `--exclude-table-data` 时，在转储数据时忽略表数据，请参见 `--exclude-table-data` 的文档。

对于 LightDB 创建的内置表，当给出 `-t` 时，仅转储名称与 `pattern` 匹配的表的数据。

对于 LightDB 创建的内置表，当给出 `-T` 时，排除名称与 `pattern` 匹配的表。

对于 LightDB 创建的内置表，当给出 `-n` 时，仅转储名称与 `pattern` 匹配的模式下的表的数据。

对于 LightDB 创建的内置表，当给出 `-N` 时，排除名称与 `pattern` 匹配的模式下的表。

选项 `--lt-exclude-lightdb-objects` 和 `--lt-dump-lightdb-tables` 不能同时使用。

#### `--lt-escape-identifiers`

强制转义表名中的单引号。如果表名中存在单引号字符，则该选项是必须的。通过该选项，单个 `'` 字符会被转义为 `''`。

#### `--lt-disable-auto-analyze`

在转储索引时不自动收集统计信息（有关详细信息，请参见 `CREATE INDEX`）。

#### `--inserts`

将数据转储为 `INSERT` 命令（而不是 `COPY`）。这将使得恢复非常慢，这主要用于使转储能够被载入到非 LightDB 数据库中。重新加载期间的任何错误都将导致有问题的 `INSERT` 相关的行将丢失，而不是整个表内容。注意如果你已经重新安排了列序，该恢复可能会一起失败。`--column-inserts` 选项对于列序改变是安全的，但是会更慢。

#### `--load-via-partition-root`

在为一个分区表转储数据时，让 `COPY` 语句或者 `INSERT` 语句把包含它的分区层次的根而不是分区自身作为目标。这导致在数据被装载时，会为每一个行重新确定合适的分区。如果在一台服务器上重新装载数据时会出现行并不是总是落入到和原始服务器上相同的分区中的情况，这个选项就很有用。例如，如果分区列是文本类型并且两个系统中用于排序分区列的排序规则有着不同的定义，就会发生这种情况。

在从用这个选项制作的归档恢复时，最好不要使用并行，因为 `lt_restore` 将不能准确地知道一个给定的归档数据项将把数据装载到哪个分区中。这会导致效率不高，因为在并行任务间会有锁冲突，或者甚至可能由于在相关数据被装载前建立了外键约束而导致重新装载失败。

#### `--lock-wait-timeout=timeout`

在转储的开始从不等待共享表锁的获得。而是在指定的 `timeout` 内不能锁定一个表时失败。超时时长可以用 `SET statement_timeout` 接受的任何格式指定（允许的值根据你从其转出的服务器版本变化，但是从 7.3 以来的所有版本都接受一个整数表示的毫秒数。如果从 7.3 以前的服务器转出，这个选项会被忽略。）。

#### `--no-comments`

不转储注释。

---

`--no-publications`

不转储publication。

`--no-security-labels`

不转储安全标签。

`--no-subscriptions`

不转储订阅。

`--no-sync`

默认情况下，`lt_dump`将等待所有文件被安全地写入磁盘。这个选项会让`lt_dump`不等待直接返回，这样会更快，但是也意味着后续的一次操作系统崩溃会让该转储损坏。通常这个选项对测试有用，但是不应该在从生产安装中转储数据时使用。

`--no-synchronized-snapshots`

这个选项允许对 9.2 以前的服务器运行`lt_dump -j`，详见`-j`参数的文档。

`--no-tablespaces`

不要输出选择表空间的命令。通过这个选项，在恢复期间所有的对象都会被创建在任何作为默认的表空间中。

当发出归档(非文本)输出文件时，这个选项被忽略。对于归档格式，你可以在调用`lt_restore`时指定该选项。

`--no-unlogged-table-data`

不转储非日志记录表的内容。这个选项对于表定义(模式)是否被转储没有影响，它只会限制转储表数据。当从一个后备服务器转储时，在非日志记录表中的数据总是会被排除。

`--on-conflict-do-nothing`

增加 `ON CONFLICT DO NOTHING` 到`INSERT` commands。除非规定了 `--inserts`，`--column-inserts` 或 `--rows-per-insert`，否则此选项是无效的，。

`--quote-all-identifiers`

强制引用所有标识符。当从LightDB主版本与`lt_dump`不同的服务器上转储一个数据库时或者当输出准备载入到一个具有不同主版本的服务器时，推荐使用这个选项。默认情况下，`lt_dump`只对在其主版本中是被保留词的标识符加上引号。在转储其他版本服务器时，这种默认行为有时会导致兼容性问题，因为那些版本可能具有些许不同的被保留词集合。使用`--quote-all-identifiers`能阻止这种问题，但代价是转储脚本更难阅读。

`--rows-per-insert=nrows`

数据转储为`INSERT`命令(而不是`COPY`)。控制每个`INSERT`命令的最大行数。指定的值必须大于零。重新加载期间的任何错误都将导致有问题的`INSERT`相关的行将丢失，而不是整个表内容。

`--section=sectionname`

只转储命名节。节的名称可以是`pre-data`、`data`或`post-data`。这个选项可以被指定多次来选择多个节。默认是转储所有节。

数据节包含真正的表数据、大对象内容和序列值。数据后项包括索引、触发器、规则和除了已验证检查约束之外的约束的定义。数据前项包括所有其他数据定义项。

`--serializable-deferrable`

为转储使用一个可序列化事务，以保证所使用的快照与后来的数据库状态是一致的。但是这样做是在事务流中等待一个点，在该点上不能存在异常，这样就不会有转储失败或者导致其他事务带着`serialization_failure`回滚的风险。关于事务隔离和并发控制详见Chapter 13

对于一个只为灾难恢复存在的转储，这个选项没什么益处。如果一个转储被用来在原始数据库持续被更新期间载入一份用于报表或其他只读负载的数据库拷贝时，这个选项就有所帮助。如果没有这个选项，转储可能会反映一个与最终提交事务的任何执行序列都不一致的状态。例如，如果使用了批处理技术，一个批处理在转储中可以显示为关闭，而其中的所有项都不出现。

如果 `lt_dump` 被启动时没有读写事务在活动，则这个选项没有什么不同。如果有读写事务在活动，该转储的启动可能会被延迟一段不确定的时间。一旦开始运行，有没有这个开关的表现是相同的。

`--snapshot=snapshotname`

在做一个数据库的转储时指定一个同步的快照（详见 Table 9.89）。

在需要把转储和一个逻辑复制槽（见Chapter 45）或者一个并发会话同步时可以用上这个选项。

在并行转储的情况下，将使用这个选项指定的快照名而不是取一个新快照。

`--strict-names`

要求每一个扩展（`-e/--extension`），模式（`-n/--schema`）和表（`-t/--table`）限定符匹配要转储的数据库中至少一个扩展/模式/表。注意，如果没有找到有这样的扩展/模式/表限定符匹配，即便没有`--strict-names`，`lt_dump`也将生成一个错误。

这个选项对`-N/--exclude-schema`、`-T/--exclude-table`或者`--exclude-table-data`没有效果。无法匹配任何对象的排除模式不会被当作错误。

`--use-set-session-authorization`

输出 SQL-标准的`SET SESSION AUTHORIZATION`命令取代`ALTER OWNER`命令来确定对象的所有关系。这让该转储更加兼容标准，但是取决于该转储中对象的历史，该转储可能无法正常恢复。而且，一个使用`SET SESSION AUTHORIZATION`的转储将一定会要求超级用户特权来正确地恢复，而`ALTER OWNER`要求更少的特权。

`-?``--help`

显示有关`lt_dump`命令行参数的帮助并退出。

下列命令行选项控制数据库连接参数。

`-d dbname``--dbname=dbname`

指定要连接的数据库的名称。这等效于在命令行上将`dbname`指定为第一个非选项参数。`dbname`可以是连接字符串。如果是这样，连接字符串参数将覆盖所有冲突的命令行选项。

`-h host``--host=host`

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从`LTHOST`环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

`-p port`  
`--port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在LTPORT环境变量中（如果被设置），否则使用编译在程序中的默认值。

`-U username`  
`--username=username`

要作为哪个用户连接。

`-w`  
`--no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

`-W`  
`--password`

强制lt\_dump在连接到一个数据库之前提示要求一个口令。

这个选项从来不是必须的，因为如果服务器要求口令认证，lt\_dump将自动提示要求一个口令。但是，lt\_dump将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入-W来避免额外的连接尝试。

`--role=rolename`

指定一个用来创建该转储的角色名。这个选项导致lt\_dump在连接到数据库后发出一个SET ROLE rolename命令。当已认证用户（由-U指定）缺少lt\_dump所需的特权但是能够切换到一个具有所需权利的角色时，这个选项很有用。一些安装有针对直接作为超级用户登录的策略，使用这个选项可以让转储在不违反该策略的前提下完成。

## 环境

LTDATABASE  
 LTHOST  
 PGOPTIONS  
 LTPORT  
 LTUSER

默认连接参数

LT\_COLOR

规定在诊断消息中是否使用颜色。可能的值为always、 auto、 never。 .

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 诊断

lt\_dump在内部执行SELECT语句。如果你运行lt\_dump时出现问题，确定你能够从正在使用的数据库中选择信息，例如ltsql。此外，libpq前端-后端库所使用的任何默认连接设置和环境变量都将适用。

lt\_dump的数据库活动会被统计收集器正常地收集。如果不想这样，你可以通过PGOPTIONS或ALTER USER命令设置参数track\_counts为假。



## 注解

如果你的数据库集簇对于template1数据库有任何本地添加，要注意将lt\_dump的输出恢复到一个真正的空数据库。否则你很可能由于以增加对象的重复定义而得到错误。要创建一个不带任何本地添加的空数据库，从template0而不是template1复制它，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

当一个只含数据的转储被选中并且使用了选项--disable-triggers时，lt\_dump在开始插入数据之前会发出命令禁用用户表上的触发器，并且接着在数据被插入之后发出命令重新启用它们。如果恢复中途被停止，系统目录可能会停留在一种错误状态。

lt\_dump产生的转储文件不包含优化器用来做出查询计划决定的统计信息。因此，在从一个转储文件恢复后运行ANALYZE来确保最优性能是明智的，详见Section 22.1.4和Section 22.1.6

因为lt\_dump被用来传输数据到更新版本的LightDB，lt\_dump的输出被认为可以载入到比lt\_dump版本更新的LightDB服务器中。lt\_dump也能够从比其版本更旧的LightDB服务器中转储（当前支持回退到版本 7.0）。不过，lt\_dump无法从比起主版本号更新的LightDB服务器中转储，它甚至将拒绝冒着创建一个非法转储的风险尝试。还有，不保证lt\_dump的输出能被载入到一个更旧主版本的服务器 — 即使该转储是从该版本的服务器中被取得也不行。将一个转储文件载入到一个更旧的服务器可能需要手工编辑该转储文件来移除旧服务器无法理解的语法。在跨版本的情况下，推荐使用--quote-all-identifiers选项，因为它可以避免因为不同LightDB版本间的保留词列表变化而发生问题。

在转储逻辑复制订阅时，lt\_dump将生成使用connect = false选项的CREATE SUBSCRIPTION命令，这样恢复订阅时不会建立远程连接来创建复制槽或者进行初始的表拷贝。通过这种方式，可以无需到远程服务器的网络访问就能恢复该转储。然后就需要用户以一种合适的方式重新激活订阅。如果涉及到的主机已经改变，连接信息可能也必须被改变。在开启一次新的全表拷贝之前，截断目标表也可能是合适的。

## 实例

要把一个数据库mydb转储到一个 SQL 脚本文件：

```
$ lt_dump mydb > db.sql
```

要把这样一个脚本重新载入到一个（新创建的）名为newdb的数据库中：

```
$ ltsql -d newdb -f db.sql
```

要转储一个数据库到一个自定义格式归档文件：

```
$ lt_dump -Fc mydb > db.dump
```

要转储一个数据库到一个目录格式的归档：

```
$ lt_dump -Fd mydb -f dumpdir
```

要用 5 个并行的工作者任务转储一个数据库到一个目录格式的归档：

```
$ lt_dump -Fd mydb -j 5 -f dumpdir
```

要把一个归档文件重新载入到一个（新创建的）名为newdb的数据库：

```
$ lt_restore -d newdb db.dump
```

把一个归档文件重新装载到同一个数据库（该归档正是从这个数据库中转储得来）中，丢掉那个数据库中的当前内容：

```
$ lt_restore -d postgres --clean --create db.dump
```

要转储一个名为mytab的表：

```
$ lt_dump -t mytab mydb > db.sql
```

要转储detroit模式中名称以emp开始的所有表，排除名为employee\_log的表：

```
$ lt_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

要转储名称以east或者west开始并且以gsm结束的所有模式，排除名称包含词test的任何模式：

```
$ lt_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

同样，用正则表达式记号法来合并开关：

```
$ lt_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

要转储除了名称以ts开头的表之外的所有数据库对象：

```
$ lt_dump -T 'ts_*' mydb > db.sql
```

要在-t和相关开关中指定一个大写形式或混合大小写形式的名称，你需要双引用该名称，否则它会被折叠到小写形式（请参见下面的Patterns）。但是双引号对于 shell 是特殊的，所以反过来它们必须被引用。因此，要转储一个有混合大小写名称的表，你需要类似这样的东西：

```
$ lt_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

## 参见

lt\_dumpall, lt\_restore, ltsql

---

# lt\_dumpall

lt\_dumpall — 将一个LightDB数据库集簇抽取到一个脚本文件中

## Synopsis

```
lt_dumpall [connection-option...] [option...]
```

## 描述

lt\_dumpall工具可以一个集簇中所有的LightDB数据库写出到（“转储”）一个脚本文件。该脚本文件包含可以用作ltsql的输入SQL命令来恢复数据库。它会对集簇中的每个数据库调用lt\_dump来完成该工作。lt\_dumpall还转储对所有数据库公用的全局对象（lt\_dump不保存这些对象），也就是说数据库角色和表空间都会被转储。目前这包括适数据库用户和组、表空间以及适合所有数据库的访问权限等属性。

因为lt\_dumpall从所有数据库中读取表，所以你很可能需要以一个数据库超级用户的身份连接以便生成完整的转储。同样，你也需要超级用户特权执行保存下来的脚本，这样才能增加角色和组以及创建数据库。

SQL 脚本将被写出到标准输出。使用 -f/--file 选项或者 shell 操作符可以把它重定向到一个文件。

lt\_dumpall需要多次连接到LightDB服务器（每个数据库一次）。如果你使用口令认证，可能每次都会要求口令。这种情况下使用一个~/.pgpass会比较方便。详见Section 32.15

## 选项

下列命令行选项用于控制输出的内容和格式。

-a  
--data-only

只转储数据，不转储模式（数据定义）。

-c  
--clean

包括在重建数据库之前清除（移除）它们的 SQL 命令。角色和表空间的DROP命令也会被加入进来。

-E encoding  
--encoding=encoding

用指定的字符集编码创建转储。默认情况下，转储使用数据库的编码创建（另一种得到相同结果的方法是设置PGCLIENTENCODING环境变量为想要的转储编码）。

-f filename  
--file=filename

将输出发送到指定的文件中。如果省略，将使用标准输出。

-g  
--globals-only

只转储全局对象（角色和表空间），而不转储数据库。

---

-0

--no-owner

不输出用于设置对象所有权以符合原始数据库的命令。默认情况下，lt\_dumpall发出ALTER OWNER或SET SESSION AUTHORIZATION语句来设置被创建的模式元素的所有权。除非脚本是由一个超级用户（或者是拥有脚本中所有对象的同一个用户）所运行，这些语句在脚本运行时可能会失败。要使得一个脚本能被任意用户恢复，但又不想给予该用户所有对象的所有权，可以指定-0。

-r

--roles-only

只转储角色，不转储数据库和表空间。

-s

--schema-only

只转储对象定义（模式），不转储数据。

-S username

--superuser=username

指定要在禁用触发器时使用的超级用户的用户名。只有使用--disable-triggers时，这个选项才相关（通常，最好省去这个选项，而作为超级用户来启动结果脚本来取而代之）。

-t

--tablespaces-only

只转储表空间，不转储数据库和角色。

-v

--verbose

指定细节模式。这将导致lt\_dumpall向标准错误输出详细的对象注释以及转储文件的开始/停止时间，还有进度消息。重复该选项会导致在标准错误上出现附加的调试级信息。这个选项还会被传递到lt\_dump。

-V

--version

打印lt\_dumpall版本并退出。

-x

--no-privileges

--no-acl

防止转储访问特权（授予/收回命令）。

--binary-upgrade

这个选项用于就地升级功能。我们不推荐也不支持把它用于其他目的。这个选项在未来的发行中可能被改变而不做通知。

--column-inserts

--attribute-inserts

将数据转储为带有显式列名的INSERT命令（INSERT INTO table(column, ...) VALUES ...）。这将使得恢复过程非常慢，这主要用于使转储能够被载入到非LightDB数据库中。

---

**--disable-dollar-quoting**

这个选项禁止在函数体中使用美元符号引用，并且强制它们使用 SQL 标准字符串语法被引用。

**--disable-triggers**

只有在创建一个只转储数据的转储时，这个选项才相关。它指示lt\_dumpall包括在数据被重新载入时能够临时禁用目标表上的触发器的命令。如果你在表上有引用完整性检查或其他触发器，并且你在数据重新载入期间不想调用它们，请使用这个选项。

当前，为--disable-triggers发出的命令必须作为超级用户来执行。因此，你还应当使用-S指定一个超级用户名，或者宁可作为一个超级用户启动结果脚本。

**--exclude-database=pattern**

不要转储名字与 pattern 匹配的数据库。可以通过编写多个--exclude-database开关来排除多个模式。 pattern参数被解释为模式，根据ltsql的\d命令使用的相同规则（请参见下面的 Patterns），因此，通过在模式中编写通配符也可以排除多个数据库。使用通配符时，请谨慎的引用模式，如果需要防止shell通配符扩展。

**--extra-float-digits=ndigits**

在转储浮点数据时使用extra\_float\_digits规定的值，而不是最大可用精度。备份目的进行的常规转储不使用此选项。

**--if-exists**

时间条件性命令（即增加一个IF EXISTS子句）来清除数据库和其他对象。只有同时指定了--clean时，这个选项才可用。

**--inserts**

将数据转储为INSERT命令（而不是COPY）。这将使得恢复非常慢，这主要用于使转储能够被载入到非LightDB数据库中。注意如果你已经重新安排了列序，该恢复可能会一起失败。--column-inserts选项对于列序改变是安全的，但是会更慢。

**--load-via-partition-root**

在为一个分区表转储数据时，让COPY语句或者INSERT语句把包含它的分区层次的根而不是分区自身作为目标。这导致在数据被装载时，会为每一个行重新确定合适的分区。如果在一台服务器上重新装载数据时会出现行并不是总是落入到和原始服务器上相同的分区中的情况，这个选项就很有用。例如，如果分区列是文本类型并且两个系统中用于排序分区列的排序规则有着不同的定义，就会发生这种情况。

**--lock-wait-timeout=timeout**

在转储的开始从不等待共享表锁的获得。而是在指定的timeout内不能锁定一个表时失败。超时时长可以用SET statement\_timeout接受的任何格式指定。

**--no-comments**

不转储注释。

**--no-publications**

不转储publication。

**--no-role-passwords**

不为角色转储口令。在恢复完后，角色的口令将是空口令，并且在设置口令之前口令认证都不会成功。由于指定这个选项时并不需要口令值，角色信息将从目录视

图pg\_roles而不是pg\_authid中读出。因此，如果对pg\_authid的访问被某条安全性策略所限制，那么这个选项也会有所帮助。

`--no-security-labels`

不转储安全标签。

`--no-subscriptions`

不转储subscription。

`--no-sync`

默认情况下，lt\_dumpall将等待所有文件被安全地写入到磁盘。这个选项会让lt\_dumpall不做这种等待而返回，这样会更快，但是意味着后续的操作系统崩溃可能留下被损坏的转储。通常来说，这个选项对测试有用，但不应该在从生产安装中转储数据时使用。

`--no-tablespaces`

不要输出选择表空间的命令。通过这个选项，在恢复期间所有的对象都会被创建在任何作为默认的表空间中。

`--no-toast-compression`

不要输出命令以设置 TOAST 压缩方法。采用这个选项，所有列将会以默认压缩设置来恢复。

`--no-unlogged-table-data`

不转储非日志记录表的内容。这个选项对于表定义（模式）是否被转储没有影响，它只会限制转储表数据。

`--on-conflict-do-nothing`

添加ON CONFLICT DO NOTHING到INSERT命令。除非--inserts或--column-inserts也被规定，否则此选项不生效。

`--quote-all-identifiers`

强制引用所有标识符。在从一个与lt\_dumpall主版本不同的LightDB服务器转储数据库时或者要将输出载入到一个不同主版本的服务器时，推荐使用这个选项。默认情况下，lt\_dumpall只会对为其主版本中保留词的标识符加上引号。在与其他版本的具有不同保留词集合的服务器交互时，这有时会导致兼容性问题。使用--quote-all-identifiers可以阻止这类问题，但是代价是转储脚本会更加难读。

`--rows-per-insert=nrows`

将数据转储为INSERT命令（而不是COPY）。控制每个INSERT命令的最大行数。指定的值必须是大于零的数。重新加载期间的任何错误都将导致仅丢失有问题的INSERT的行，而不是整个表内容。

`--use-set-session-authorization`

输出 SQL-标准的SET SESSION AUTHORIZATION命令取代ALTER OWNER命令来确定对象的所有关系。这让该转储更加兼容标准，但是取决于该转储中对象的历史，该转储可能无法正常恢复。

`--lt-target-version`

lt\_dumpall 直接将该选项传递给 lt\_dump。

---

`--lt-exclude-lightdb-objects`

lt\_dumpall 直接将该选项传递给 lt\_dump。

`--lt-dump-lightdb-tables`

lt\_dumpall 直接将该选项传递给 lt\_dump。

`--lt-escape-identifiers`

lt\_dumpall 直接将该选项传递给 lt\_dump。

`-?`

`--help`

显示有关lt\_dumpall命令行参数的帮助并退出。

下列命令行选项控制数据库连接参数。

`-d connstr`

`--dbname=connstr`

指定用于连接到服务器的参数，比如连接字符串；这些将覆盖所有冲突的命令行选项。

这个选项被称为--dbname是为了和其他客户端应用一致，但是因为lt\_dumpall需要连接多个数据库，连接字符串中的数据库名将被忽略。使用-l选项指定一个数据库，该数据库被用于初始连接，这将转储全局对象并且发现需要转储哪些其他数据库。

`-h host`

`--host=host`

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从LTHOST环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

`-l dbname`

`--database=dbname`

指定要连接到哪个数据库转储全局对象以及发现要转储哪些其他数据库。如果没有指定，将会使用postgres数据库，如果postgres不存在，就使用 template1。

`-p port`

`--port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在LTPORT环境变量中（如果被设置），否则使用编译在程序中的默认值。

`-U username`

`--username=username`

要作为哪个用户连接。

`-w`

`--no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

`-W`

`--password`

强制lt\_dumpall在连接到一个数据库之前提示要求一个口令。

这个选项从来不是必须的，因为如果服务器要求口令认证，`lt_dumpall`将自动提示要求一个口令。但是，`lt_dumpall`将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入`-W`来避免额外的连接尝试。

注意对每个要被转储的数据库，口令提示都会再次出现。通常，最好设置一个`~/pgpass`文件来减少手工口令输入。

`--role=rolename`

指定一个用来创建该转储的角色名。这个选项导致`lt_dump`在连接到数据库后发出一个`SET ROLE rolename`命令。当已认证用户（由`-U`指定）缺少`lt_dump`所需的特权但是能够切换到一个具有所需权利的角色时，这个选项很有用。一些安装有针对直接作为超级用户登录的策略，使用这个选项可以让转储在不违反该策略的前提下完成。

## 环境

LTHOST  
PGOPTIONS  
LTPORT  
LTUSER

默认连接参数

`PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为`always`、`auto`、`never`。

和大部分其他LightDB工具相似，这个工具也使用`libpq`（见Section 32.14 支持的环境变量）。

## 注解

因为`lt_dumpall`在内部调用`lt_dump`，所以，一些诊断消息可以参考`lt_dump`。

即使当用户的目的是把转储脚本恢复到一个空的集簇中，`--clean`选项也有用武之地。`--clean`的使用让该脚本删除并且重建内建的`postgres`和`template1`数据库，确保这两个数据库保持与源集簇中相同的属性（例如`locale`和编码）。如果不用这个选项，这两个数据库将保持它们现有的数据库级属性以及任何已有的内容。

一旦恢复，建议在每个数据库上运行`ANALYZE`，这样优化器就可以得到有用的统计信息。你也可以运行`vacuumdb -a -z`来分析所有数据库。

不应该预期转储脚本运行到结束都不出错。特别是由于脚本将为源集簇中已有的每一个角色发出`CREATE ROLE`语句，对于`bootstrap`超级用户当然会得到一个“`role already exists`”错误，除非目标集簇用一个不同的`bootstrap`超级用户名完成的初始化。这种错误是无害的并且应该被忽略。`--clean`选项的使用很可能会产生额外的有关于不存在对象的无害错误消息，不过可以通过加上`--if-exists`减少这类错误消息。

`lt_dumpall`要求所有需要的表空间目录在进行恢复之前就必须存在；否则，数据库创建就会由于在非默认位置创建数据库而失败。

## 例子

要转储所有数据库：

```
$ lt_dumpall > db.out
```

要从这个文件重新载入数据库，你可以使用：



```
$ ltsql -f db.out postgres
```

这里你连接哪一个数据库并不重要，因为由lt\_dumpall创建的脚本将包含合适的命令来创建和连接到被保存的数据库。一个例外是，如果指定了--clean，则开始时必须连接到postgres数据库，该脚本将立即尝试删除其他数据库，并且这种动作对于已连接上的这个数据库将会失败。

## 参见

可能的错误情况请查看lt\_dump。

---

# lt\_isready

lt\_isready — 检查一个LightDB服务器的连接状态

## Synopsis

```
lt_isready [connection-option...] [option...]
```

## 描述

lt\_isready是一个用来检查一个LightDB数据库服务器的连接状态的工具。其退出状态指定了连接检查的结果。

## 选项

-d dbname

--dbname=dbname

指定要连接的数据库名。dbname可以是连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

-h hostname

--host=hostname

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port

--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。默认值取自LTPORT环境变量。如果环境变量没有设置，则默认值使用编译时指定的端口（通常是5432）。

-q

--quiet

不显示状态消息。当脚本编程时有用。

-t seconds

--timeout=seconds

尝试连接时，在返回服务器不响应之前等待的最大秒数。设置为 0 则禁用。默认值是 3 秒。

-U username

--username=username

作为用户username连接数据库，而不是用默认用户。

-V

--version

打印lt\_isready版本并退出。

-?

--help

显示有关lt\_isready命令行参数的帮助并退出。

## 退出状态

如果服务器正常接受连接，`lt_isready`返回0给 shell；如果服务器拒绝连接（例如处于启动阶段）则返回1；如果连接尝试没有被相应则返回2；如果没有尝试（例如由于非法参数）则返回3。

## 环境

和大部分其他LightDB工具相似，`lt_isready`也使用`libpq`（见Section 32.14 支持的环境变量）。

环境变量`PG_COLOR`规定在诊断消息中是否使用颜色。可能的值为`always`、`auto`、`never`。

## 注解

要获得服务器状态，不一定需要提供正确的用户名、口令或数据库名。不过，如果提供了不正确的值，服务器将会记录一次失败的连接尝试。

## 例子

标准用法：

```
$ lt_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

使用连接参数运行连接到处于启动中的LightDB集簇：

```
$ lt_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

使用连接参数运行连接到无响应的LightDB集簇：

```
$ lt_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

---

# lt\_probackup

lt\_probackup — 管理LightDB数据库集群的备份和恢复

## Synopsis

```
lt_probackup version
lt_probackup help [command]
lt_probackup init -B backup_dir
lt_probackup add-instance -B backup_dir -D data_dir --instance instance_name
lt_probackup del-instance -B backup_dir --instance instance_name
lt_probackup set-config -B backup_dir --instance instance_name [option...]
lt_probackup set-backup -B backup_dir --instance instance_name -i backup_id
[option...]
lt_probackup show-config -B backup_dir --instance instance_name [--format=format]
lt_probackup show -B backup_dir [option...]
lt_probackup backup -B backup_dir --instance instance_name -b backup_mode
[option...]
lt_probackup restore -B backup_dir --instance instance_name [option...]
lt_probackup checkdb -B backup_dir --instance instance_name -D data_dir [option...]
lt_probackup validate -B backup_dir [option...]
lt_probackup merge -B backup_dir --instance instance_name -i backup_id [option...]
lt_probackup delete -B backup_dir --instance instance_name { -i backup_id | --
delete-wal | --delete-expired | --merge-expired } [option...]
lt_probackup archive-push -B backup_dir --instance instance_name --wal-file-name
wal_file_name [option...]
lt_probackup archive-get -B backup_dir --instance instance_name --wal-file-path
wal_file_path --wal-file-name wal_file_name [option...]
```

## 描述

lt\_probackup 是一个管理 LightDB 数据库集群备份和恢复的实用程序。它旨在定期备份 LightDB 实例，以便在发生故障时恢复服务器。

- 概述
- 安装和设置
- 命令行参考
- 用法

## 概述

与其他备份解决方案相比，lt\_probackup 提供以下优势，有助于您实现不同的备份策略和处理大量数据：

- 增量备份：有三种不同的增量模式，可以根据数据流计划备份策略。增量备份可以节省磁盘空间，并加快备份速度，相比于进行完全备份。使用增量备份恢复集群的速度也比重放 WAL 文件更快。
- 增量恢复：通过重用 LTDATA 中可用的有效未更改的页面，加速从备份中的恢复。
- 验证：自动进行数据一致性检查，并可在不进行实际数据恢复的情况下进行按需备份验证。
- 验证：使用 checkdb 命令，按需验证 LightDB 实例。
- 保留：根据保留策略管理 WAL 存档和备份。您可以根据恢复时间或要保留的备份数量配置保留策略，还可以为特定备份指定生存时间 (TTL)。过期备份可以合并或删除。
- 并行处理：在多个并行线程上运行 backup、restore、merge、delete、validate 和 checkdb 进程。
- 压缩：将备份数据以压缩状态存储，以节省磁盘空间。
- 重复数据删除：如果非数据文件（例如 \_vm 或 \_fsm）自从它们被复制到这个增量备份链中的一个之后就没有更改，那么在增量备份中排除这些文件，以节省磁盘空间。
- 远程操作：备份位于远程系统上的 LightDB 实例或远程还原备份。
- 从备用服务器备份：通过从备用服务器中进行备份，避免在主服务器上额外增加负载。
- 外部目录：备份位于 LightDB 数据目录 (LTDATA) 之外的文件和目录，例如脚本、配置文件、日志或 SQL 转储文件。
- 备份目录：获取备份列表和相应的元数据信息，以纯文本或 JSON 格式呈现。
- 档案目录：获取所有 WAL 时间轴和相应的元数据信息列表，以纯文本或 JSON 格式呈现。
- 部分恢复：仅恢复指定的数据库。

为了管理备份数据，lt\_probackup 创建一个 备份目录。这是一个目录，存储所有备份文件及其附加的元数据信息，以及进行时段恢复所需的 WAL 存档。可以在单个备份目录的不同子目录中存储不同实例的备份。

使用 lt\_probackup 工具，您可以进行全量或增量备份：

- FULL 备份包含恢复数据库集群所需的所有数据文件。
- 增量备份在页面级别上操作，仅存储与上一次备份不同的数据。它允许您节省磁盘空间，加快备份过程，与全量备份相比。通过应用增量备份来恢复集群也比通过重放WAL文件来恢复快得多。lt\_probackup 支持以下增量备份模式：
  - DELTA备份。在此模式下，lt\_probackup 会读取数据目录中所有数据文件，并仅复制与上一次备份不同的页面。此模式可能会对只读I/O产生与全量备份等价的压力。
  - PAGE备份。在此模式下，lt\_probackup 从上一次全量或增量备份开始扫描归档中的所有WAL文件。新创建的备份仅包含WAL记录中提到的页面。这要求自上一次备份以来的所有WAL文件都存在于WAL归档中。如果这些文件的大小与数据库集群文件的总大小相当，则速度提升较小，但备份仍占用更少的空间。您需要按 设置持续的WAL归档 中所述进行WAL归档配置，以进行PAGE备份。
  - PTRACK备份。在此模式下，LightDB 实时跟踪页面更改。不需要持续归档即可运行。每次更新关系页面时，此页面都会在特殊的PTRACK位图中标记。跟踪会对数据库服务器操作产生一些轻微的开销，但可以显著加快增量备份速度。

lt\_probackup 只能进行物理在线备份，而在线备份需要WAL以进行一致性恢复。因此，无论选择哪种备份模式 (FULL、PAGE或DELTA)，使用 lt\_probackup 进行的任何备份都必须使用以下 WAL传递模式之一：

- ARCHIVE。这种备份依赖于 持续归档 以确保一致的恢复。这是默认的WAL传递模式。
- STREAM。此类备份包括将集群恢复到备份时的一致状态所需的所有文件。无论是否已设置持续归档，在备份过程中，用于一致恢复所需的WAL段都会通过复制协议进行流式传输，并包含在备份文件中。这就是这种备份被称为 独立 或 独立式 的原因。

## 限制

lt\_probackup 目前存在以下限制：

- Windows系统不支持远程模式。

- 在Unix系统上，只有启动了 LightDB 服务器的同一OS用户才能进行备份。例如，如果 LightDB 服务器由用户 lightdb 启动，则备份命令也必须由用户 lightdb 运行。在使用 SSH进行远程备份时，为了满足此要求，必须将 `--remote-user` 选项设置为 lightdb。
- 进行备份的 LightDB 服务器和恢复服务器必须通过 `block_size`<sup>1</sup> 和 `wal_block_size`<sup>2</sup> 参数兼容，并且具有相同的主版本号。根据群集配置，LightDB 本身可能会施加其他限制，例如CPU架构或 libc/libicu 版本。

## 安装和设置

安装了 lt\_probackup 后，请完成以下设置：

- 初始化备份目录。
- 在备份目录中添加新的备份实例。
- 配置数据库集群以启用 lt\_probackup 备份。
- 可选地，配置SSH以在远程模式下运行 lt\_probackup 操作。

### 初始化备份目录

lt\_probackup 将所有WAL和备份文件存储在备份目录的相应子目录中。

要初始化备份目录，请运行以下命令：

```
lt_probackup init -B backup_dir
```

其中 backup\_dir 是备份目录的路径。如果 backup\_dir 已经存在，则必须为空。否则，lt\_probackup 将返回错误。

启动 lt\_probackup 的用户必须对 backup\_dir 目录具有完全访问权限。

lt\_probackup 创建备份目录 backup\_dir，包含以下子目录：

- wal/ — WAL文件目录。
- backups/ — 备份文件存放的目录。

初始化备份目录后，您可以添加新的备份实例。

### 添加新的备份实例

lt\_probackup 可以在单个备份目录中存储多个数据库集群的备份。为了设置所需的子目录，您必须为每个要备份的数据库集群在备份目录中添加备份实例。

要添加新的备份实例，请运行以下命令：

```
lt_probackup add-instance -B backup_dir -D data_dir --instance instance_name
[remote_options]
```

其中：

- data\_dir 是您要备份的集群的数据目录。为了设置和使用 lt\_probackup，需要对此目录进行写访问。
- instance\_name 是将存储此集群的 WAL 和备份文件的子目录的名称。
- remote\_options 是可选参数，只有当 data\_dir 位于远程系统上时才需要指定。

lt\_probackup 在备份目录的 backups/ 和 wal/ 目录下创建 instance\_name 子目录。backups/instance\_name 目录包含 pg\_probackup.conf 配置文件，该文件控制此备份实例的

<sup>1</sup> runtime-config-preset.html#GUC-BLOCK-SIZE

<sup>2</sup> runtime-config-preset.html#GUC-WAL-BLOCK-SIZE

lt\_probackup 设置。如果您使用 remote\_options 参数运行此命令，则指定的参数将添加到 pg\_probackup.conf 文件中。

有关如何微调 lt\_probackup 配置的详细信息，请参见 the section called “配置 lt\_probackup”。

启动 lt\_probackup 的用户必须对 backup\_dir 目录具有完全访问权限，至少对 data\_dir 目录具有只读访问权限。如果您在 BACKUP\_PATH 环境变量中指定了备份目录的路径，则运行 lt\_probackup 命令时可以省略相应的选项。

### Note

建议使用 allow-group-access<sup>3</sup> 特性，以便可以由与集群所有者相同组的任何操作系统用户进行备份。在这种情况下，用户应该对集群目录具有读取权限。

## 配置数据库集群

虽然超级用户可以使用 lt\_probackup，但建议创建一个具有所选备份策略所需的最小权限的单独角色。在这些配置说明中，backup 角色被用作示例。

执行 backup，需要为角色 backup 分配以下权限，仅需在与 LightDB 服务器建立连接的数据库中分配：

```
BEGIN;
CREATE ROLE backup WITH LOGIN;
GRANT USAGE ON SCHEMA pg_catalog TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.current_setting(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_is_in_recovery() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_start_backup(text, boolean, boolean) TO
  backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_stop_backup(boolean, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_create_restore_point(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_switch_wal() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_last_wal_replay_lsn() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current_snapshot() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_snapshot_xmax(txid_snapshot) TO
  backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_control_checkpoint() TO backup;
COMMIT;
```

在 lt\_hba.conf<sup>4</sup> 文件中，允许代表 backup 角色连接数据库群集。

因为 lt\_probackup 需要直接读取群集文件，lt\_probackup 必须由拥有对将要备份的数据目录 (LTDATA) 中所有文件和目录的读取权限的操作系统用户启动（或连接到，如果在远程模式下使用）。

根据您的计划进行 单机 或 归档 备份的情况，LightDB 群集配置将有所不同，具体在下面的章节中指定。要从备用服务器备份数据库群集，请在远程模式下运行 lt\_probackup，或创建 PTRACK 备份，需要进行额外设置。

有关详细信息，请参见以下章节：设置 STREAM 备份、设置持续 WAL 存档、设置备用备份、配置远程模式、设置部分还原和设置 PTRACK 备份。

<sup>3</sup> app-initdb.html#APP-INITDB-ALLOW-GROUP-ACCESS

<sup>4</sup> auth-pg-hba-conf.html

## 设置 STREAM 备份

要为 STREAM 备份设置群集，请完成以下步骤：

- 授权 backup 角色 REPLICATION 权限：

```
ALTER ROLE backup WITH REPLICATION;
```

- 在 `lt_hba.conf`<sup>5</sup> 文件中，允许代表 backup 角色进行复制。
- 确保参数 `max_wal_senders`<sup>6</sup> 设置足够高，以便至少留出一个会话可用于备份进程。
- 将参数 `wal_level`<sup>7</sup> 设置为高于 `minimal`。

如果您计划在 STREAM 模式下进行 PAGE 备份或使用 STREAM 备份执行 PITR，则仍需要配置 WAL 存档，如设置持续 WAL 存档章节中所述。

一旦完成了这些步骤，您就可以在 STREAM WAL 模式下开始进行 FULL、PAGE、DELTA 和 PTRACK 备份。

### Note

如果您计划在流模式下运行备份，并依赖于 `.pgpass`<sup>8</sup> 进行身份验证，则 `.pgpass` 必须包含用于通过复制协议建立连接的 `replication` 数据库的凭据。例如：`pghost:5432:replication:backup_user:my_strong_password`

## 设置持续 WAL 存档

使用 PAGE 备份模式进行备份、执行 PITR、使用 ARCHIVE WAL 传递模式进行备份以及在时间线切换后运行增量备份都需要启用持续 WAL 存档<sup>9</sup>。要在群集中设置持续存档，请执行以下步骤：

- 确保参数 `wal_level`<sup>10</sup> 高于 `minimal`。
- 如果您在主服务器上配置归档，则必须将 `archive_mode`<sup>11</sup> 参数设置为 `on` 或 `always`。要在备用服务器上执行存档，请将此参数设置为 `always`。
- 设置 `archive_command`<sup>12</sup> 参数，如下所示：

```
archive_command = '"/install_dir/lt_probackup" archive-push -B "backup_dir" --instance instance_name --wal-file-name=%f [remote_options]'
```

这里 `install_dir` 是你将要使用的 `lt_probackup` 版本的安装目录，`backup_dir` 和 `instance_name` 是此数据库集群已初始化的备份目录实例的引用，`remote_options` 只需要在远程主机上归档 WAL 才需要指定。有关所有可能的 `archive-push` 参数的详细信息，请参阅 `archive-push` 章节。

完成这些步骤后，您就可以开始使用 ARCHIVE WAL 模式备份、PAGE 备份模式备份，以及执行 PITR。

<sup>5</sup> `auth-pg-hba-conf.html`

<sup>6</sup> `runtime-config-replication.html#GUC-MAX-WAL-SENDERS`

<sup>7</sup> `runtime-config-wal.html#GUC-WAL-LEVEL`

<sup>8</sup> `libpq-pgpass.html`

<sup>9</sup> `continuous-archiving.html`

<sup>10</sup> `runtime-config-wal.html#GUC-WAL-LEVEL`

<sup>11</sup> `runtime-config-wal.html#GUC-ARCHIVE-MODE`

<sup>12</sup> `runtime-config-wal.html#GUC-ARCHIVE-COMMAND`



您可以使用 `show` 命令查看WAL归档的当前状态。有关详细信息，请参阅 the section called “查看WAL归档信息” 章节。

如果您计划从生成少量WAL流量的备用服务器进行PAGE备份和/或使用 `ARCHIVE WAL` 模式备份，而不需要长时间等待WAL段填充，请考虑在主服务器上设置 `LightDB` 参数 `archive_timeout`<sup>13</sup>。此参数的值应略低于 `--archive-timeout` 设置（默认值为5分钟），以便在备份因 `--archive-timeout` 而中止之前有足够的时间将旋转的段流式传输到备用服务器并发送到WAL归档。

#### Note

您可以使用任何其他工具来设置连续归档，只要它将WAL段传递到 `backup_dir/wal/instance_name` 目录中。如果使用压缩，则应使用 `gzip`，并且文件名中的后缀 `.gz` 是必需的。而不是使用 `lt_probackup` 提供的 `archive-push` 命令。

#### Note

您可以选择使用 `pg_receivewal`<sup>14</sup> 实用程序来配置连续归档，而不是通过设置 `archive_mode` 和 `archive_command` 参数。在这种情况下，`pg_receivewal` 的 `-D directory` 选项应指向 `backup_dir/wal/instance_name` 目录。`lt_probackup` 支持由 `pg_receivewal` 完成的WAL压缩。仅使用 `pg_receivewal` 才能实现“零数据丢失”归档策略。

## 设置备用数据库备份

对于LightDB，`lt_probackup`可以从备用服务器上备份。这需要以下额外的设置：

- 在备用服务器上，将`hot_standby`<sup>15</sup>参数设置为`on`。
- 在主服务器上，将`full_page_writes`<sup>16</sup>参数设置为`on`。
- 要在备用服务器上执行独立备份，请完成“设置流备份”一节中的所有步骤。
- 要在备用服务器上执行归档备份，请完成“设置连续WAL归档”一节中的所有步骤。

完成这些步骤后，您可以从备用服务器开始采用适当的WAL交付模式（归档或流）进行FULL，PAGE，DELTA或PTRACK备份。

从备用服务器备份具有以下限制：

- 如果在备份期间将备用服务器升级为主服务器，则备份将失败。
- 备份所需的所有WAL记录必须包含足够的全页写入。这要求您在主服务器上启用`full_page_writes`，并且不使用像`pg_compresslog`这样的工具作为`archive_command`<sup>17</sup>，以从WAL文件中删除全页写入。

## 设置集群验证

数据库集群的逻辑验证需要以下额外的设置。此处以角色`backup`为例：

- 在集群的每个数据库中安装`amcheck`<sup>18</sup>：

<sup>13</sup> `runtime-config-wal.html#GUC-ARCHIVE-TIMEOUT`  
<sup>14</sup> `app-pgreceivewal.html`  
<sup>15</sup> `runtime-config-replication.html#GUC-HOT-STANDBY`  
<sup>16</sup> `runtime-config-wal.html#GUC-FULL-PAGE-WRITES`  
<sup>17</sup> `runtime-config-wal.html#GUC-ARCHIVE-COMMAND`  
<sup>18</sup> `amcheck.html`

```
CREATE EXTENSION amcheck;
```

- 在集群的每个数据库中，将以下权限授予backup角色：

```
GRANT SELECT ON TABLE pg_catalog.pg_am TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_class TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_database TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_namespace TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_extension TO backup;
GRANT EXECUTE ON FUNCTION bt_index_check(regclass) TO backup;
GRANT EXECUTE ON FUNCTION bt_index_check(regclass, bool) TO backup;
```

## 设置部分恢复

如果您计划使用部分恢复，请完成以下额外步骤：

- 仅在用于连接到LightDB服务器的数据库中，将只读访问权限授予backup角色以访问pg\_catalog.pg\_database：

```
GRANT SELECT ON TABLE pg_catalog.pg_database TO backup;
```

## 配置远程模式

lt\_probackup支持远程模式，允许远程执行备份，恢复和WAL归档操作。在此模式下，备份目录存储在本地系统上，而要备份和/或恢复的LightDB实例位于远程系统上。目前，唯一支持的远程协议是SSH。

## 设置SSH

如果您要通过SSH远程模式使用lt\_probackup，请按照以下步骤完成：

- 在两个系统上都安装lt\_probackup： backup\_host和db\_host。
- 为了让主机之间进行通信，请在backup\_host上的backup用户和db\_host上的lightdb用户之间设置无密码SSH连接：

```
[backup@backup_host] ssh-copy-id lightdb@db_host
```

- 如果您要依赖于 连续的WAL归档， 请在db\_host上的lightdb用户和backup\_host上的backup用户之间设置无密码SSH连接：

```
[lightdb@db_host] ssh-copy-id backup@backup_host
```

这里：

- backup\_host是具有备份目录的系统。
- db\_host是具有LightDB集群的系统。
- backup是在backup\_host上运行lt\_probackup的操作系统用户。
- lightdb是在db\_host上启动LightDB集群的操作系统用户。 对于LightDB，可以使用更安全的方法，因为可以利用initdb的允许组访问<sup>19</sup>功能。

通过SSH的远程模式下，lt\_probackup的工作方式如下：

- 只有以下命令可以在远程模式下启动： add-instance、 backup、 restore、 archive-push、 archive-get。

<sup>19</sup> app-initdb.html#APP-INITDB-ALLOW-GROUP-ACCESS

- 在远程模式下操作需要在本地和远程系统上都安装lt\_probackup二进制文件。本地和远程二进制文件的版本必须相同。
- 当以远程模式启动时，本地系统上的主要lt\_probackup进程通过SSH连接到远程系统，并在远程系统上启动一个或多个代理进程，称为远程代理。远程代理的数量等于-j/--threads设置。
- 主要lt\_probackup进程使用远程代理访问远程文件并在本地和远程系统之间传输数据。
- 远程代理尽量减少主机之间的网络流量和往返次数。
- 主进程通常在备份主机上启动，并连接到数据库主机，但在使用archive-push和archive-get命令时，主进程会在数据库主机上启动，并连接到备份主机。
- 数据传输完成后，远程代理被终止，SSH连接被关闭。
- 如果远程代理遇到错误条件，则所有代理都会被终止，并且错误详细信息会由主要的lt\_probackup进程报告，该进程会以错误退出。
- 压缩始终在db\_host上进行，而解压始终在backup\_host上进行。

### Note

您可以强制在SSH设置上实施额外限制<sup>20</sup>，以保护系统在账户遭到妥协的情况下。

## 设置PTRACK备份

LightDB可以使用PTRACK备份模式。可以在此处找到指向PTRACK补丁的链接 [here](#)<sup>21</sup>。

### Note

PTRACK将在未来直接使用。

如果您将使用PTRACK备份，请完成以下附加步骤。执行PTRACK备份的角色（在下面的示例中为backup角色）必须能够访问集群中的所有数据库。

对于LightDB:

1. 创建PTRACK扩展:

```
CREATE EXTENSION ptrack;
```

2. 要启用跟踪页面更新，请将ptrack.map\_size 参数设置为正整数并重新启动服务器。

为了获得最佳性能，建议将ptrack.map\_size设置为  $N / 1024$ ，其中  $N$ 是LightDB集群的大小(以MB为单位)。如果将此参数设置为较低的值，则PTRACK更有可能将多个块映射在一起，这会导致跟踪更改的块时出现误报，并增加增量备份大小，因为未更改的块也会被复制到增量备份中。将ptrack.map\_size设置为较高的值不会影响PTRACK操作，但不建议将此参数设置为高于1024的值。

### Note

如果更改了ptrack.map\_size参数值，则先前创建的PTRACK映射文件将被清除，并重新开始跟踪新更改的块。因此，在更改ptrack.map\_size后，在进行增量PTRACK备份之前，必须重新进行完整备份。

<sup>20</sup> [https://man.openbsd.org/OpenBSD-current/man8/sshd.8#AUTHORIZED\\_KEYS\\_FILE\\_FORMAT](https://man.openbsd.org/OpenBSD-current/man8/sshd.8#AUTHORIZED_KEYS_FILE_FORMAT)

<sup>21</sup> [https://github.com/postgrespro/pg\\_probackup#ptrack-support](https://github.com/postgrespro/pg_probackup#ptrack-support)

## 用法

### 创建备份

要创建备份，请执行以下命令：

```
lt_probackup backup -B backup_dir --instance instance_name -b backup_mode
```

其中backup\_mode可以取以下值之一：

- FULL - 创建包含要恢复的集群的所有数据文件的完整备份。
- DELTA - 读取数据目录中的所有数据文件，并为自上次备份以来已更改的页面创建增量备份。
- PAGE - 基于自上次完整备份或增量备份以来生成的WAL文件创建增量备份。仅从数据文件中读取已更改的块。
- PTRACK - 实时跟踪页面更改，创建增量备份。

当从增量备份恢复集群时，lt\_probackup依赖于父完整备份和它们之间的所有增量备份，这称为备份链。在进行增量备份之前，必须至少创建一个完整备份。

### 归档模式

ARCHIVE是默认的WAL传递模式。

例如，要在ARCHIVE模式下进行完整备份，请运行以下命令：

```
lt_probackup backup -B backup_dir --instance instance_name -b FULL
```

归档备份依赖于持续归档，以获取在备份时恢复集群所需的WAL段，使其处于一致的状态。

当进行备份时，lt\_probackup确保包含Start LSN和Stop LSN之间WAL记录的WAL文件实际上存在于 backup\_dir/wal/instance\_name 目录中。lt\_probackup还确保可以解析Start LSN和Stop LSN之间的WAL记录。这项预防措施消除了隐式WAL损坏的风险。

### 流模式

STREAM是可选的WAL传递模式。

例如，要在STREAM模式下进行完整备份，请在之前示例的命令中添加--stream标志：

```
lt_probackup backup -B backup_dir --instance instance_name -b FULL --stream --temp-slot
```

可选的--temp-slot标志确保在备份完成之前WAL被旋转时，所需的段仍然可用。

与ARCHIVE模式下的备份不同，STREAM备份包括在备份时恢复集群所需的所有WAL段，以使其处于一致的状态。

backup期间，lt\_probackup将包含 Start LSN和Stop LSN之间WAL记录的WAL文件流式传输到 backup\_dir/backups/instance\_name/backup\_id/database/lt\_wal目录中。为了消除隐式WAL损坏的风险，lt\_probackup还检查Start LSN和Stop LSN之间的WAL记录是否可以解析。

即使您使用了持续归档，在以下情况下STREAM备份仍然很有用：

- STREAM备份可以在无法访问WAL归档的服务器上恢复。

- STREAM备份使您可以在WAL归档中的WAL文件不再可用的时间点还原集群状态。
- 在流模式下，可以从生成少量WAL流量的服务器的备用服务器中进行备份，而不需要长时间等待WAL段填满。

## 页面验证

如果数据库集群中启用了数据校验和<sup>22</sup>，`lt_probackup`将使用此信息在备份过程中检查数据文件的正确性。在读取每个页面时，`lt_probackup`会检查计算出的校验和是否与存储在页面头部的校验和相符。这保证了LightDB实例和备份本身没有损坏的页面。请注意，`lt_probackup`直接从文件系统中读取数据库文件，因此在备份过程中，如果存在大量写入负载，它可能会显示错误的校验和不匹配，因为存在部分写入。如果页面校验和不匹配，则重新读取页面并重复校验和比较。

如果校验和比较失败超过100次，则认为页面已损坏。在这种情况下，备份将被中止。

即使没有启用数据校验和，`lt_probackup`也始终对页面头执行健全性检查。

## 外部目录

要备份位于数据目录之外的目录，请使用可选的`--external-dirs`参数，该参数指定此目录的路径。如果您想要添加多个外部目录，则可以在Linux系统上使用冒号或Windows系统上使用分号将多个路径分隔开。

例如，要将`/etc/dir1`和`/etc/dir2`目录包含在将存储在Linux的`backup_dir`目录下的完全备份中，请运行以下命令：

```
lt_probackup backup -B backup_dir --instance instance_name -b FULL --external-dirs=/etc/dir1:/etc/dir2
```

同样，要将`C:\dir1`和`C:\dir2`目录包含在Windows上的完全备份中，请运行以下命令：

```
lt_probackup backup -B backup_dir --instance instance_name -b FULL --external-dirs=C:\dir1;C:\dir2
```

`lt_probackup`递归地将每个外部目录的内容复制到备份目录中的单独子目录中。由于包含在不同备份中的外部目录不必相同，因此在从增量备份恢复集群时，只会恢复属于此特定备份的目录。存储在先前备份中的任何外部目录都将被忽略。

要在您的实例的每个备份中包含相同的目录，可以使用`set-config`命令和`--external-dirs`选项在`pg_probackup.conf`配置文件中指定它们。

## 执行集群验证

要验证LightDB数据库集群未损坏，请运行以下命令：

```
lt_probackup checkdb [-B backup_dir [--instance instance_name]] [-D data_dir] [connection_options]
```

该命令通过运行页面头健全性检查以及块级校验和验证（如果启用了校验和），对指定数据目录中的所有数据文件执行物理验证。如果检测到损坏的页面，则`checkdb`将继续集群验证，直到验证集群中的所有页面。

默认情况下，`lt_probackup`在进行备份时会自动执行类似的页面验证。`checkdb`命令使您能够按需执行此类页面验证，而无需创建任何备份副本，即使不使用`lt_probackup`备份集群。

<sup>22</sup> runtime-config-preset.html#GUC-DATA-CHECKSUMS

要执行集群验证，lt\_probackup需要连接到要验证的集群。通常，只需指定此集群的备份实例，lt\_probackup就可以确定所需的连接选项。但是，如果省略了-B和--instance选项，则必须通过环境变量或命令行选项提供data\_dir及其所需的连接选项。

物理验证无法检测到逻辑不一致、缺失或已作废的块和整个文件，或类似的异常情况。扩展amcheck<sup>23</sup>提供了这些问题的部分解决方案。

如果除了进行物理验证之外，您还想使用这些扩展验证所有数据库中的所有索引，可以在运行checkdb命令时指定--amcheck标志：

```
lt_probackup checkdb -D data_dir --amcheck [connection_options]
```

您可以通过指定--skip-block-validation标志来跳过物理验证。在这种情况下，您可以省略backup\_dir和data\_dir选项，只需要提供连接选项即可：

```
lt_probackup checkdb --amcheck --skip-block-validation [connection_options]
```

可以通过使用--heapallindexed标志更彻底地进行逻辑验证，检查所有应该索引的堆元组是否实际上被索引，但这会以更高的CPU、内存和I/O消耗为代价。

## 验证备份

lt\_probackup在备份过程中为每个文件计算校验和。检查备份数据文件的校验和的过程称为备份验证。默认情况下，验证在备份完成后立即运行，并在恢复之前运行，以检测可能的备份损坏。

如果您想跳过备份验证，可以在运行backup和restore命令时指定--no-validate标志。

为确保所有必需的备份文件都存在并可以用于恢复数据库集群，可以使用与恢复时相同的恢复目标选项运行validate命令。

例如，要检查您可以将数据库集群从备份副本还原到事务ID 4242，运行以下命令：

```
lt_probackup validate -B backup_dir --instance instance_name --recovery-target-xid=4242
```

如果验证成功完成，lt\_probackup会显示相应的消息。如果验证失败，您将收到一条错误消息，其中包含恢复的确切时间、事务ID和LSN的信息。

如果通过-i/--backup-id选项指定backup\_id，则仅验证具有指定备份ID的备份副本。如果使用恢复目标选项指定backup\_id，validate命令将检查是否可以将指定备份还原到指定的恢复目标。

例如，要检查您可以将数据库集群从具有备份ID PT8XFX的备份副本还原到指定的时间戳，请运行以下命令：

```
lt_probackup validate -B backup_dir --instance instance_name -i PT8XFX --recovery-target-time="2017-05-18 14:18:11+03"
```

如果您指定了增量备份的 backup\_id，则将验证从完全备份开始的所有父备份。

如果您省略所有参数，则会验证所有备份。

## 恢复集群

要从备份中恢复数据库集群，请至少使用以下选项运行restore命令：

<sup>23</sup> amcheck.html

```
lt_probackup restore -B backup_dir --instance instance_name -i backup_id
```

其中：

- 备份目录是存储所有备份文件和元信息的备份目录。
- 实例名称是要恢复的集群的备份实例。
- backup\_id指定要从中恢复集群的备份。如果您省略此选项，lt\_probackup会使用指定实例中可用的最新有效备份。如果您指定要还原的增量备份，lt\_probackup会自动还原底层的全备份，然后依次应用所有必要的增量备份。

一旦完成restore命令，启动数据库服务。

如果您恢复了归档备份，执行了PITR，或者在restore命令中指定了--restore-as-replica标志以设置备用服务器，lt\_probackup会在所有数据文件都复制到目标目录后创建恢复配置文件。此文件包括恢复所需的最小设置，除了primary\_conninfo<sup>24</sup>参数中的密码；您必须手动添加密码或使用--primary-conninfo选项（如果需要）。对于LightDB，lt\_probackup将这些设置写入probackup\_recovery.conf文件，然后将其包含在lightdb.auto.conf中。

如果您正在恢复STREAM备份，则一次完成恢复，集群将在备份拍摄时返回到自治状态。对于归档备份，LightDB会重放所有可用的已归档WAL段，因此集群将恢复到当前时间轴上可能的最新状态。您可以使用restore命令的恢复目标选项更改此行为，如the section called “执行时间点恢复（PITR）”中所述。

如果要恢复的集群包含表空间，则lt\_probackup默认将其恢复到原始位置。要将表空间恢复到其他位置，请使用--tablespace-mapping/-T选项。否则，在同一主机上恢复集群将失败，因为备份必须写入相同的目录，而表空间正在使用中。

使用--tablespace-mapping/-T选项时，必须提供旧表空间目录和新表空间目录的绝对路径。如果路径包含等号（=），请使用反斜杠进行转义。该选项可多次指定，用于多个表空间。例如：

```
lt_probackup restore -B backup_dir --instance instance_name -
D data_dir -j 4 -i backup_id -T tablespace1_dir=tablespace1_newdir -T
tablespace2_dir=tablespace2_newdir
```

要在远程主机上恢复集群，请按照the section called “使用远程模式下的lt\_probackup”中的说明操作。

### Note

默认情况下，restore命令会在恢复集群之前验证指定的备份。如果您定期运行备份验证并希望在恢复集群时节省时间，可以指定--no-validate标志来跳过验证并加快恢复速度。

## 增量恢复

使用带有增量恢复选项的restore命令，可以通过替换已经存在的LightDB数据目录中仅无效和更改的页面，显著提高从备份恢复的速度。

要以增量模式从备份中恢复数据库集群，请使用以下选项运行restore命令：

<sup>24</sup> runtime-config-replication.html#GUC-PRIMARY-CONNINFO

```
lt_probackup restore -B backup_dir --instance instance_name -D data_dir -
I incremental_mode
```

其中 `incremental_mode` 可以采用以下值之一：

- CHECKSUM — 读取数据目录中的所有数据文件，验证每个页面的头部和校验和，并仅替换无效页面和具有校验和和LSN与备份中相应页面不匹配的页面。这是最简单，最傻瓜式的增量模式。建议默认使用。
- LSN — 读取数据目录中的 `lt_control`，以获取重做LSN和重做TLI，从而允许确定历史上的一个点（`shiftpoint`），在此数据目录状态从目标备份链历史中转移。如果`shiftpoint`不在备份链历史的范围内，则恢复操作将被中止。如果`shiftpoint`在备份链历史的范围内，则读取数据目录中的所有数据文件，验证每个页面的头部和校验和，并仅替换无效页面和具有LSN大于`shiftpoint`的页面。与CHECKSUM相比，此模式提供了更快的速度，但要满足两个条件。首先，在数据目录中必须启用 `数据校验和`<sup>25</sup> 参数（以避免由提示位引起的损坏）。增量恢复开始时将检查此条件，如果禁用了校验和，则操作将被中止。其次，`lt_control` 文件必须与数据目录的状态同步。在恢复开始时无法检查此条件，因此用户有责任确保 `lt_control` 包含有效信息。因此，在无法信任 `lt_control` 或经过篡改的情况下，不建议在任何情况下使用LSN模式：在执行 `pg_resetxlog` 后，在没有运行恢复的备份还原之后等。
- NONE — 常规恢复，没有任何增量优化。

无论选择哪种增量模式，`lt_probackup` 都会检查给定目标目录中的 `postmaster` 是否未运行，并且 `system-identifier` 与备份中的相同。

假设您想使用LSN模式的增量恢复将旧主服务器作为副本返回，在切换后执行以下操作：

```
=====
```

Instance Time	Version Data	ID WAL	Recovery Zratio	Time Start LSN	Stop LSN	Mode Status	WAL Mode	TLI
node	12	QBRNBP	2020-06-11 17:40:58+03	DELTA	ARCHIVE	16/15		
40s	194MB	16MB	8.26	15/2C000028	15/2D000128	OK		
node	12	QBRIDX	2020-06-11 15:51:42+03	PAGE	ARCHIVE	15/15		
11s	18MB	16MB	5.10	14/DC000028	14/DD0000B8	OK		
node	12	QBRIAJ	2020-06-11 15:51:08+03	PAGE	ARCHIVE	15/15		
20s	141MB	96MB	6.22	14/D4BABFE0	14/DA9871D0	OK		
node	12	QBRHT8	2020-06-11 15:45:56+03	FULL	ARCHIVE	15/0		
2m:11s	1371MB	416MB	10.93	14/9D000028	14/B782E9A0	OK		

```
=====
```

```
lt_probackup restore -B /backup --instance node -R -I lsn
INFO: Running incremental restore into nonempty directory: "/var/lib/lightdb-
x/12/data"
INFO: Destination directory redo point 15/2E000028 on tli 16 is within reach of
backup QBRIDX with Stop LSN 14/DD0000B8 on tli 15
INFO: shift LSN: 14/DD0000B8
INFO: Restoring the database from backup at 2020-06-11 17:40:58+03
INFO: Extracting the content of destination directory for incremental restore
INFO: Destination directory content extracted, time elapsed: 1s
INFO: Removing redundant files in destination directory
INFO: Redundant files are removed, time elapsed: 1s
INFO: Start restoring backup files. LTDATA size: 15GB
INFO: Backup files are restored. Transferred bytes: 1693MB, time elapsed: 43s
INFO: Restore incremental ratio (less is better): 11% (1693MB/15GB)
INFO: Restore of backup QBRNBP completed.
```

<sup>25</sup> runtime-config-preset.html#GUC-DATA-CHECKSUMS



### Note

仅对具有 `program_version` 大于或等于 2.4.0 的备份才可以进行增量恢复。

## 部分恢复

如果您在备份之前启用了 部分恢复，则可以使用带有 部分恢复选项 的 `restore` 命令仅恢复部分数据库。

要仅恢复指定的数据库，请使用以下选项运行 `restore` 命令：

```
lt_probackup restore -B backup_dir --instance instance_name --db-include=database_name
```

可以多次指定 `--db-include` 选项。例如，要仅恢复数据库 `db1` 和 `db2`，请运行以下命令：

```
lt_probackup restore -B backup_dir --instance instance_name --db-include=db1 --db-include=db2
```

要从恢复中排除一个或多个数据库，请使用 `--db-exclude` 选项：

```
lt_probackup restore -B backup_dir --instance instance_name --db-exclude=database_name
```

可以多次指定 `--db-exclude` 选项。例如，要从恢复中排除数据库 `db1` 和 `db2`，请运行以下命令：

```
lt_probackup restore -B backup_dir --instance instance_name --db-exclude=db1 --db-exclude=db2
```

部分恢复依赖于 LightDB 恢复进程对截断文件的宽松行为。为了使恢复正常工作，已排除的数据库的文件将作为大小为零的文件进行恢复。成功启动 LightDB 集群后，必须使用 `DROP DATABASE` 命令删除已排除的数据库。

为了将包含多个数据库的单个集群分解为具有最小停机时间的单独集群，您可以使用 `--restore-as-replica` 选项对特定数据库进行集群的部分恢复，使其作为备用数据库。

### Note

总是会恢复 `template0` 和 `template1` 数据库。

## 执行时间点恢复（PITR）

如果在进行备份之前启用了 持续WAL归档，则可以使用 `recovery` 目标选项 以任意时间点（恢复目标）恢复集群状态，使用 `restore` 命令进行恢复。

只要WAL归档从备份开始的时间点可用，您可以同时使用STREAM和ARCHIVE备份进行恢复。如果省略 `-i/--backup-id` 选项，`lt_probackup` 将自动选择最接近指定恢复目标的备份并开始恢复过程，否则 `lt_probackup` 将尝试将指定备份还原到指定的恢复目标。

- 要恢复集群在指定时间点的状态，请使用 `--recovery-target-time` 选项，以时间戳格式指定时间。例如：

```
lt_probackup restore -B backup_dir --instance instance_name --recovery-target-time="2017-05-18 14:18:11+03"
```

- 要将集群状态还原到特定事务ID，请使用 `--recovery-target-xid` 选项：

```
lt_probackup restore -B backup_dir --instance instance_name --recovery-target-xid=687
```

- 要将集群状态还原到特定LSN，请使用 `--recovery-target-lsn` 选项：

```
lt_probackup restore -B backup_dir --instance instance_name --recovery-target-lsn=16/B374D848
```

- 要将集群状态还原到特定的命名还原点，请使用 `--recovery-target-name` 选项：

```
lt_probackup restore -B backup_dir --instance instance_name --recovery-target-name="before_app_upgrade"
```

- 要将备份还原到WAL归档中可用的最新状态，请使用 `--recovery-target` 选项，并将值设置为 `latest`：

```
lt_probackup restore -B backup_dir --instance instance_name --recovery-target="latest"
```

- 要将集群还原到最早的一致性点，请使用 `--recovery-target` 选项，并将值设置为 `immediate`：

```
lt_probackup restore -B backup_dir --instance instance_name --recovery-target='immediate'
```

## 使用远程模式下的 lt\_probackup

lt\_probackup 支持远程模式，可以通过SSH远程执行备份和恢复操作。在此模式下，备份目录存储在本地系统上，而要备份的 LightDB 实例位于远程系统上。您必须在两个系统上都安装 lt\_probackup。

### Note

lt\_probackup 依赖于无密码SSH连接来在主机之间进行通信。

典型的工作流程如下：

- 在备份主机上，按照安装和设置部分中的说明配置 lt\_probackup。对于 `add-instance` 和 `set-config` 命令，请确保指定 `remote` 选项，指向带有 LightDB 实例的数据库主机。
- 如果您想在 `PAGE` 模式下进行远程备份，或者依赖于 `ARCHIVE WAL` 传递模式，或者使用 `PITR`，请按照设置持续的WAL归档部分中的说明，从数据库主机到备份主机配置持续的WAL归档。对于 `archive-push` 和 `archive-get` 命令，您必须指定指向备份目录所在的备份主机的 `remote` 选项。
- 在备份主机上使用 `backup` 或 `restore` 命令，并使用 `remote` 选项。lt\_probackup 通过 SSH 连接到远程系统，并在本地创建备份或将之前拍摄的备份恢复到远程系统上。

例如，要代表用户lightdb通过SSH连接通过端口2302与远程主机地址 192.168.0.2 上的LightDB 集群创建一个归档全备份，运行：

```
lt_probackup backup -B backup_dir --instance instance_name -b FULL --remote-user=lightdb --remote-host=192.168.0.2 --remote-port=2302
```

要代表用户lightdb通过SSH连接通过端口2302在远程主机地址 192.168.0.2 上恢复最新可用的备份，运行：

```
lt_probackup restore -B backup_dir --instance instance_name --remote-user=lightdb--remote-host=192.168.0.2 --remote-port=2302
```

在远程模式下恢复ARCHIVE备份或执行PITR需要额外的信息：目标地址、端口和用户名，用于从具有数据库的主机到备份目录的主机建立SSH连接。 这些信息将由restore\_command用于将WAL段从归档复制到LightDB lt\_wal目录。

要解决这个问题，您可以使用远程WAL归档选项。

例如，要通过 SSH 连接到地址为 192.168.0.2 的主机上的用户 lightdb，并通过端口号 2302，以及连接到地址为 192.168.0.3 的备份目录主机上的用户 backup，使用远程模式还原最新的备份，可以运行以下命令：

```
lt_probackup restore -B backup_dir --instance instance_name --remote-user=lightdb--remote-host=192.168.0.2 --remote-port=2302 --archive-host=192.168.0.3 --archive-port=2303 --archive-user=backup
```

提供的参数将用于构造restore\_command：

```
restore_command = ' "install_dir/lt_probackup" archive-get -B "backup_dir" --instance instance_name --wal-file-path=%p --wal-file-name=%f --remote-host=192.168.0.3 --remote-port=2303 --remote-user=backup'
```

或者，您可以使用选项--restore-command来提供整个restore\_command：

```
lt_probackup restore -B backup_dir --instance instance_name --remote-user=lightdb --remote-host=192.168.0.2 --remote-port=2302 --restore-command=' "install_dir/lt_probackup" archive-get -B "backup_dir" --instance instance_name --wal-file-path=%p --wal-file-name=%f --remote-host=192.168.0.3 --remote-port=2303 --remote-user=backup'
```

### Note

目前，远程模式对Windows系统不可用。

## 在并行线程上运行lt\_probackup

backup、restore、merge、delete、checkdb和validate进程可以在几个并行线程上执行。如果有足够的资源（CPU核心、磁盘和网络带宽），这可以显著加快lt\_probackup操作。

并行执行由-j/--threads命令行选项控制。例如，要使用四个并行线程创建备份，请运行：

```
lt_probackup backup -B backup_dir --instance instance_name -b FULL -j 4
```

### Note

并行恢复仅适用于从备份目录复制数据到集群的数据目录。启动LightDB服务器时，需要回放WAL记录，这不能并行执行。

## 配置lt\_probackup

一旦初始化备份目录并添加了新的备份实例，您可以使用位于backup\_dir/backups/instance\_name目录中的pg\_probackup.conf配置文件来微调lt\_probackup的配置。

例如，backup和checkdb命令使用常规的LightDB连接。为了避免每次在命令行上指定连接选项，您可以使用set-config命令在pg\_probackup.conf配置文件中设置它们。

### Note

pg\_probackup.conf手动编辑不建议。

最初，pg\_probackup.conf包含以下设置：

- LTDATA — 要备份的集群数据目录的路径。
- system-identifier — LightDB 实例的唯一标识符。

此外，您可以使用set-config命令定义远程、保留、日志记录和压缩设置：

```
lt_probackup set-config -B backup_dir --instance instance_name
[--external-dirs=external_directory_path] [remote_options] [connection_options]
[retention_options] [logging_options]
```

要查看当前设置，请运行以下命令：

```
lt_probackup show-config -B backup_dir --instance instance_name
```

在运行lt\_probackup 命令时，您可以通过相应的环境变量和/或命令行选项覆盖pg\_probackup.conf中定义的设置。

## 指定连接设置

如果在pg\_probackup.conf配置文件中定义了连接设置，您可以在所有后续的lt\_probackup命令中省略连接选项。然而，如果相应环境变量已设置，则它们具有更高的优先级。命令行提供的选项会覆盖环境变量和配置文件设置。

如果没有给出任何值，则采用默认值。默认情况下，lt\_probackup尝试通过Unix域套接字（在Windows上为localhost）使用本地连接，并尝试从LTUSER环境变量或当前操作系统用户名获取数据库名称和用户名。

## 管理备份目录

使用lt\_probackup，您可以从命令行管理备份：

- 查看备份信息

- 查看WAL归档信息
- 验证备份
- 合并备份
- 删除备份

## 查看备份信息

要查看每个实例的现有备份列表，请运行以下命令：

```
lt_probackup show -B backup_dir
```

lt\_probackup显示所有可用备份的列表。 例如：

```
BACKUP INSTANCE 'node'
```

Instance	Version	ID	Recovery time		Mode	WAL Mode	TLI	Time
Data	WAL	Zratio	Start LSN	Stop LSN	Status			
node	10	PYSUE8	2019-10-03	15:51:48+03	FULL	ARCHIVE	1/0	16s
9047kB	16MB	4.31	0/12000028	0/12000160	OK			
node	10	P7XDQV	2018-04-29	05:32:59+03	DELTA	STREAM	1/1	11s
19MB	16MB	1.00	0/15000060	0/15000198	OK			
node	10	P7XDJA	2018-04-29	05:28:36+03	PTRACK	STREAM	1/1	21s
32MB	32MB	1.00	0/13000028	0/13000198	OK			
node	10	P7XDHU	2018-04-29	05:27:59+03	PAGE	STREAM	1/1	15s
33MB	16MB	1.00	0/11000028	0/110001D0	OK			
node	10	P7XDHB	2018-04-29	05:27:15+03	FULL	STREAM	1/0	11s
39MB	16MB	1.00	0/F000028	0/F000198	OK			

对于每个备份，提供以下信息：

- Instance — 实例名称。
- Version — LightDB的主要版本。
- ID — 备份标识符。
- 恢复时间 — 您可以恢复数据库集群状态的最早时间。
- Mode — 用于进行此备份的方法。可能的值：FULL，PAGE，DELTA，PTRACK。
- WAL模式 — WAL传递模式。可能的值：STREAM和ARCHIVE。
- TLI — 当前备份及其父备份的时间线标识符。
- Time — 执行备份所需的时间。
- Data — 此备份中数据文件的大小。此值不包括WAL文件的大小。对于STREAM备份，备份的总大小可以计算为Data + WAL。
- WAL — 在恢复期间需要应用的未压缩的WAL文件大小，以使备份达到一致的状态。
- Zratio — 压缩比，计算公式为“未压缩字节” / “数据字节”。
- Start LSN — WAL日志序列号对应于备份过程的开始。LightDB恢复过程的REDO点。
- Stop LSN — WAL日志序列号对应于备份过程的结束。LightDB恢复过程的一致性点。

- Status — 备份状态。可能的值：
  - OK — 备份已完成且有效。
  - DONE — 备份已完成，但未经过验证。
  - RUNNING — 备份正在进行中。
  - MERGING — 正在合并备份。
  - MERGED — 备份数据文件已成功合并，但其元数据正在更新过程中。只有完整备份才能具有此状态。
  - DELETING — 正在删除备份文件。
  - CORRUPT — 部分备份文件已损坏。
  - ERROR — 因为意外错误备份被中止。
  - ORPHAN — 备份无效，因为其中一个父备份文件已损坏或丢失。

只有当备份状态为 OK 或 DONE 时才能从备份中恢复集群。

要获取有关备份的更详细信息，请使用备份ID运行 show 命令：

```
lt_probackup show -B backup_dir --instance instance_name -i backup_id
```

示例输出如下所示：

```
#Configuration
backup-mode = FULL
stream = false
compress-alg = zlib
compress-level = 1
from-replica = false

#Compatibility
block-size = 8192
wal-block-size = 8192
checksum-version = 1
program-version = 2.1.3
server-version = 10

#Result backup info
timelineid = 1
start-lsn = 0/04000028
stop-lsn = 0/040000f8
start-time = '2017-05-16 12:57:29'
end-time = '2017-05-16 12:57:31'
recovery-xid = 597
recovery-time = '2017-05-16 12:57:31'
expire-time = '2020-05-16 12:57:31'
data-bytes = 22288792
wal-bytes = 16777216
uncompressed-bytes = 39961833
pgdata-bytes = 39859393
status = OK
parent-backup-id = 'PT8XFX'
primary_conninfo = 'user=backup passfile=/var/lib/lightdb-x/.pgpass port=5432
sslmode=disable sslcompression=1 target_session_attrs=any'
```

详细输出包含以下附加属性：

- compress-alg — 备份期间使用的压缩算法。可能的值为：zlib, pglz, none。
- compress-level — 备份期间使用的压缩级别。

- `from-replica` — 此备份是否在备用节点上进行？可能的值为：1, 0。
- `block-size` — 备份开始时LightDB集群的设置`block_size`<sup>26</sup>。
- `checksum-version` — `data block checksums`<sup>27</sup>在LightDB备份集群中是否启用？可能的值为：1, 0。
- `program-version` — 用于创建备份的`lt_probackup`二进制文件的完整版本。
- `start-time` — 备份开始时间。
- `end-time` — 备份结束时间。
- `expire-time` — 根据保留策略，已固定备份可以删除的时间点。此属性仅适用于固定备份。
- `uncompressed-bytes` — 添加页面头并应用压缩之前的数据文件大小。如果使用了压缩，则可以通过比较`uncompressed-bytes`和`data-bytes`来评估压缩的有效性。
- `pgdata-bytes` — 备份时LightDB集群数据文件的大小。可以通过比较`pgdata-bytes`和`uncompressed-bytes`来评估增量备份的有效性。
- `recovery-xid` — 备份结束时的事务ID。
- `parent-backup-id` — 父备份的ID。仅适用于增量备份。
- `primary_conninfo` — 用于连接到LightDB集群以进行此备份的`libpq`连接参数。不包括密码。
- `note` — 附加到备份的文本说明。
- `content-crc` — `backup_content.control`文件的CRC32校验和。用于检测备份元信息的损坏。

您也可以以 JSON 格式获取备份的详细信息：

```
lt_probackup show -B backup_dir --instance instance_name --format=json -i
backup_id
```

示例输出如下：

```
[
  {
    "instance": "node",
    "backups": [
      {
        "id": "PT91HZ",
        "parent-backup-id": "PT8XFX",
        "backup-mode": "DELTA",
        "wal": "ARCHIVE",
        "compress-alg": "zlib",
        "compress-level": 1,
        "from-replica": false,
        "block-size": 8192,
        "xlog-block-size": 8192,
        "checksum-version": 1,
        "program-version": "2.1.3",
        "server-version": "10",
        "current-tli": 16,
        "parent-tli": 2,
        "start-lsn": "0/8000028",
        "stop-lsn": "0/8000160",
        "start-time": "2019-06-17 18:25:11+03",
        "end-time": "2019-06-17 18:25:16+03",
        "recovery-xid": 0,

```

<sup>26</sup> [runtime-config-preset.html#GUC-BLOCK-SIZE](#)

<sup>27</sup> [runtime-config-preset.html#GUC-DATA-CHECKSUMS](#)

```

        "recovery-time": "2019-06-17 18:25:15+03",
        "data-bytes": 106733,
        "wal-bytes": 16777216,
        "primary_conninfo": "user=backup passfile=/var/lib/lightdb-
x/.pgpass port=5432 sslmode=disable sslcompression=1 target_session_attrs=any",
        "status": "OK"
    }
]
}
]

```

## 查看WAL归档信息

要查看每个实例的WAL归档信息，请运行以下命令：

```
lt_probackup show -B backup_dir [--instance instance_name] --archive
```

lt\_probackup 显示按时间线分组的所有可用 WAL 文件列表。例如：

```
ARCHIVE INSTANCE 'node'
```

TLI	Parent TLI	Switchpoint	Min Segno	Max Segno
N segments	Size	Zratio	N backups	Status
5	1	0/B000000	000000050000000000000000B	
00000005000000000000000000C	2		685kB	48.00 0 OK
4	3	0/1800000	0000000400000000000000018	
0000000400000000000000001A	3		648kB	77.00 0 OK
3	2	0/1500000	0000000300000000000000015	
00000003000000000000000017	3		648kB	77.00 0 OK
2	1	0/B000108	000000020000000000000000B	
00000002000000000000000015	5		892kB	94.00 1 DEGRADED
1	0	0/0	0000000100000000000000001	
000000010000000000000000A	10		8774kB	19.00 1 OK

对于每个时间线，提供以下信息：

- TLI — 时间线标识符。
- Parent TLI — 该时间线从哪个时间线分支出的标识符。
- Switchpoint — 时间线从其父时间线分支出的时刻的 LSN。
- Min Segno — 属于时间线的第一个 WAL 日志段。
- Max Segno — 属于时间线的最后一个 WAL 日志段。
- N segments — 属于时间线的 WAL 日志段的数量。
- Size — 文件在磁盘上的大小。
- Zratio — 压缩比，计算公式为  $N \text{ segments} * \text{wal\_segment\_size} * \text{wal\_block\_size} / \text{Size}$ 。
- N backups — 属于时间线的备份数量。要获取有关备份的详细信息，请使用 JSON 格式。
- Status — 该时间线的 WAL 存档状态。可能的值有：
  - OK — Min Segno 和 Max Segno 之间的所有 WAL 日志段都存在。
  - DEGRADED — Min Segno 和 Max Segno 之间的某些 WAL 日志段丢失。要查找丢失的文件，请使用 JSON 格式查看此报告。

要以 JSON 格式获取有关 WAL 存档的更详细信息，请运行以下命令：



```
lt_probackup show -B backup_dir [--instance instance_name] --archive --
format=json
```

示例输出如下:

```
[
  {
    "instance": "replica",
    "timelines": [
      {
        "tli": 5,
        "parent-tli": 1,
        "switchpoint": "0/B000000",
        "min-segno": "000000050000000000000000B",
        "max-segno": "000000050000000000000000C",
        "n-segments": 2,
        "size": 685320,
        "zratio": 48.00,
        "closest-backup-id": "PXS920",
        "status": "OK",
        "lost-segments": [],
        "backups": []
      },
      {
        "tli": 4,
        "parent-tli": 3,
        "switchpoint": "0/1800000",
        "min-segno": "0000000400000000000000018",
        "max-segno": "000000040000000000000001A",
        "n-segments": 3,
        "size": 648625,
        "zratio": 77.00,
        "closest-backup-id": "PXS9CE",
        "status": "OK",
        "lost-segments": [],
        "backups": []
      },
      {
        "tli": 3,
        "parent-tli": 2,
        "switchpoint": "0/1500000",
        "min-segno": "0000000300000000000000015",
        "max-segno": "0000000300000000000000017",
        "n-segments": 3,
        "size": 648911,
        "zratio": 77.00,
        "closest-backup-id": "PXS9CE",
        "status": "OK",
        "lost-segments": [],
        "backups": []
      },
      {
        "tli": 2,
        "parent-tli": 1,
        "switchpoint": "0/B000108",
        "min-segno": "000000020000000000000000B",
        "max-segno": "0000000200000000000000015",
```

```

    "n-segments": 5,
    "size": 892173,
    "zratio": 94.00,
    "closest-backup-id": "PXS920",
    "status": "DEGRADED",
    "lost-segments": [
      {
        "begin-segno": "000000020000000000000000D",
        "end-segno": "000000020000000000000000E"
      },
      {
        "begin-segno": "0000000200000000000000010",
        "end-segno": "0000000200000000000000012"
      }
    ],
    "backups": [
      {
        "id": "PXS9CE",
        "backup-mode": "FULL",
        "wal": "ARCHIVE",
        "compress-alg": "none",
        "compress-level": 1,
        "from-replica": "false",
        "block-size": 8192,
        "xlog-block-size": 8192,
        "checksum-version": 1,
        "program-version": "2.1.5",
        "server-version": "10",
        "current-tli": 2,
        "parent-tli": 0,
        "start-lsn": "0/C000028",
        "stop-lsn": "0/C000160",
        "start-time": "2019-09-13 21:43:26+03",
        "end-time": "2019-09-13 21:43:30+03",
        "recovery-xid": 0,
        "recovery-time": "2019-09-13 21:43:29+03",
        "data-bytes": 104674852,
        "wal-bytes": 16777216,
        "primary_conninfo": "user=backup passfile=/
var/lib/lightdb-x/.pgpass port=5432 sslmode=disable sslcompression=1
target_session_attrs=any",
        "status": "OK"
      }
    ]
  },
  {
    "tli": 1,
    "parent-tli": 0,
    "switchpoint": "0/0",
    "min-segno": "0000000100000000000000001",
    "max-segno": "000000010000000000000000A",
    "n-segments": 10,
    "size": 8774805,
    "zratio": 19.00,
    "closest-backup-id": "",
    "status": "OK",
    "lost-segments": [],
    "backups": [

```

```

    {
      "id": "PXS920",
      "backup-mode": "FULL",
      "wal": "ARCHIVE",
      "compress-alg": "none",
      "compress-level": 1,
      "from-replica": "true",
      "block-size": 8192,
      "xlog-block-size": 8192,
      "checksum-version": 1,
      "program-version": "2.1.5",
      "server-version": "10",
      "current-tli": 1,
      "parent-tli": 0,
      "start-lsn": "0/4000028",
      "stop-lsn": "0/6000028",
      "start-time": "2019-09-13 21:37:36+03",
      "end-time": "2019-09-13 21:38:45+03",
      "recovery-xid": 0,
      "recovery-time": "2019-09-13 21:37:30+03",
      "data-bytes": 25987319,
      "wal-bytes": 50331648,
      "primary_conninfo": "user=backup passfile=/
var/lib/lightdb-x/.pgpass port=5432 sslmode=disable sslcompression=1
target_session_attrs=any",
      "status": "OK"
    }
  ]
}
],
{
  "instance": "master",
  "timelines": [
    {
      "tli": 1,
      "parent-tli": 0,
      "switchpoint": "0/0",
      "min-segno": "0000000100000000000000000001",
      "max-segno": "000000010000000000000000000B",
      "n-segments": 11,
      "size": 8860892,
      "zratio": 20.00,
      "status": "OK",
      "lost-segments": [],
      "backups": [
        {
          "id": "PXS92H",
          "parent-backup-id": "PXS92C",
          "backup-mode": "PAGE",
          "wal": "ARCHIVE",
          "compress-alg": "none",
          "compress-level": 1,
          "from-replica": "false",
          "block-size": 8192,
          "xlog-block-size": 8192,
          "checksum-version": 1,
          "program-version": "2.1.5",

```

```

        "server-version": "10",
        "current-tli": 1,
        "parent-tli": 1,
        "start-lsn": "0/4000028",
        "stop-lsn": "0/50000B8",
        "start-time": "2019-09-13 21:37:29+03",
        "end-time": "2019-09-13 21:37:31+03",
        "recovery-xid": 0,
        "recovery-time": "2019-09-13 21:37:30+03",
        "data-bytes": 1328461,
        "wal-bytes": 33554432,
        "primary_conninfo": "user=backup passfile=
var/lib/lightdb-x/.pgpass port=5432 sslmode=disable sslcompression=1
target_session_attrs=any",
        "status": "OK"
    },
    {
        "id": "PXS92C",
        "backup-mode": "FULL",
        "wal": "ARCHIVE",
        "compress-alg": "none",
        "compress-level": 1,
        "from-replica": "false",
        "block-size": 8192,
        "xlog-block-size": 8192,
        "checksum-version": 1,
        "program-version": "2.1.5",
        "server-version": "10",
        "current-tli": 1,
        "parent-tli": 0,
        "start-lsn": "0/2000028",
        "stop-lsn": "0/2000160",
        "start-time": "2019-09-13 21:37:24+03",
        "end-time": "2019-09-13 21:37:29+03",
        "recovery-xid": 0,
        "recovery-time": "2019-09-13 21:37:28+03",
        "data-bytes": 24871902,
        "wal-bytes": 16777216,
        "primary_conninfo": "user=backup passfile=
var/lib/lightdb-x/.pgpass port=5432 sslmode=disable sslcompression=1
target_session_attrs=any",
        "status": "OK"
    }
]
}
]
]

```

大多数字段与纯格式一致，但有一些例外：

- 大小以字节为单位。
- `closest-backup-id` 属性包含属于先前时间线之一的最近的有效备份的 ID。您可以使用此备份执行该时间线的时间点恢复。如果不存在这样的备份，则此字符串为空。
- `lost-segments` 数组提供关于 DEGRADED 时间线中缺失段的间隔信息。在 OK 时间线中，`lost-segments` 数组为空。
- `backups` 数组列出属于时间线的所有备份。如果时间线没有备份，则此数组为空。

## 配置保留策略

使用 `lt_probackup`，您可以配置保留策略以删除冗余备份、清理不需要的 WAL 文件，并钉住指定备份以确保它们在指定时间内保留，如下文所述。所有这些操作都可以以任何方式组合。

### 删除冗余备份

默认情况下，使用 `lt_probackup` 创建的所有备份副本都存储在指定的备份目录中。为了节省磁盘空间，您可以配置保留策略以删除冗余备份副本。

要配置保留策略，请通过 `set-config` 在 `pg_probackup.conf` 文件中设置以下一个或多个变量：

```
--retention-redundancy=redundancy
```

指定在备份目录中保留的完整备份副本的数量。

```
--retention-window=window
```

定义了 `lt_probackup` 可以完成恢复的最早时间点。该选项是以从当前时刻起的天数来设定的。例如，如果 `retention-window=7`，则 `lt_probackup` 必须保留至少一份早于七天的备份副本，以及所有对应的 WAL 文件和随之而来的备份。

如果同时设置了 `--retention-redundancy` 和 `--retention-window` 选项，则在清理备份目录时必须考虑这两个条件。例如，如果设置 `--retention-redundancy=2` 和 `--retention-window=7`，则 `lt_probackup` 必须保留两个完整备份副本，以及确保最近七天内可恢复性所需的所有备份副本。

```
lt_probackup set-config -B backup_dir --instance instance_name --retention-redundancy=2 --retention-window=7
```

要根据保留策略清理备份目录，必须使用保留标志运行 `delete` 命令，如下所示；或者在创建新备份时使用这些标志来处理过时的备份副本，可以使用 `backup` 命令。

例如，要删除不再满足定义的保留策略的所有备份副本，请使用 `--delete-expired` 标志运行以下命令：

```
lt_probackup delete -B backup_dir --instance instance_name --delete-expired
```

如果您还想删除不再需要用于任何备份的 WAL 文件，则应该指定 `--delete-wal` 标志：

```
lt_probackup delete -B backup_dir --instance instance_name --delete-expired --delete-wal
```

您还可以通过在运行 `delete` 或 `backup` 命令时直接指定 `--retention-redundancy` 和 `--retention-window` 选项来设置或覆盖当前的保留策略：

```
lt_probackup delete -B backup_dir --instance instance_name --delete-expired --retention-window=7 --retention-redundancy=2
```

由于增量备份要求其父全备份和所有之前的增量备份都可用，如果其中任何一个备份过期，则仍然不能删除，只要该链中的至少一个增量备份满足保留策略。为避免保留过期备份以恢复活动增量备份，可以在运行 `backup` 或 `delete` 命令时使用 `--merge-expired` 标志将它们与此备份合并。

假设您已经使用 `--retention-window` 选项将 `node` 实例备份到 `backup_dir` 目录中，并且在 2019 年 4 月 10 日有以下备份可用：

```
BACKUP INSTANCE 'node'
```

Instance Data	Version WAL	ID Zratio	Recovery time Start LSN	Stop LSN	Mode Status	WAL	TLI	Time	
node	10	P7XDHR	2019-04-10 05:27:15+03	0/18000059	0/18000197	FULL	STREAM	1/0	11s
200MB	16MB	1.0			OK				
node	10	P7XDQV	2019-04-08 05:32:59+03	0/15000060	0/15000198	PAGE	STREAM	1/0	11s
19MB	16MB	1.0			OK				
node	10	P7XDJA	2019-04-03 05:28:36+03	0/13000028	0/13000198	DELTA	STREAM	1/0	21s
32MB	16MB	1.0			OK				
-----retention									
window-----									
node	10	P7XDHU	2019-04-02 05:27:59+03	0/11000028	0/11000100	PAGE	STREAM	1/0	31s
33MB	16MB	1.0			OK				
node	10	P7XDHB	2019-04-01 05:27:15+03	0/F000028	0/F000198	FULL	STREAM	1/0	11s
200MB	16MB	1.0			OK				
node	10	P7XDFT	2019-03-29 05:26:25+03	0/D000028	0/D000198	FULL	STREAM	1/0	11s
200MB	16MB	1.0			OK				

即使 P7XDHB 和 P7XDHU 备份超出了保留窗口，它们也不能被删除，因为这会使后续的增量备份 P7XDJA 和 P7XDQV 失效，而这些备份仍然是必需的。因此，如果运行带有 `--delete-expired` 标志的 `delete` 命令，则只有 P7XDFT 全量备份将被删除。

使用 `--merge-expired` 选项，P7XDJA 备份与底层的 P7XDHU 和 P7XDHB 备份合并成一个完整备份，因此不再需要保留这些过期备份：

```
lt_probackup delete -B backup_dir --instance node --delete-expired --merge-expired
lt_probackup show -B backup_dir
```

```
BACKUP INSTANCE 'node'
```

Instance Data	Version WAL	ID Zratio	Recovery time Start LSN	Stop LSN	Mode Status	WAL	TLI	Time	
node	10	P7XDHR	2019-04-10 05:27:15+03	0/18000059	0/18000197	FULL	STREAM	1/0	11s
200MB	16MB	1.0			OK				
node	10	P7XDQV	2019-04-08 05:32:59+03	0/15000060	0/15000198	PAGE	STREAM	1/0	11s
19MB	16MB	1.0			OK				
node	10	P7XDJA	2019-04-03 05:28:36+03	0/13000028	0/13000198	FULL	STREAM	1/0	21s
32MB	16MB	1.0			OK				

合并后备份的 `Time` 字段显示了合并所需的时间。

## 备份固定

如果您需要保留某些备份的时间比规定的保留政策更长，您可以将它们固定一段任意的时间。例如：

```
lt_probackup set-backup -B backup_dir --instance instance_name -i backup_id --ttl=30d
```

此命令将指定备份的过期时间设置为从其 `recovery-time` 属性中指定的时间开始的30天。

您也可以使用 `--expire-time` 选项显式设置备份的过期时间。例如：

```
lt_probackup set-backup -B backup_dir --instance instance_name -i backup_id --
expire-time="2020-01-01 00:00:00+03"
```

或者，您可以使用 `backup` 命令的 `--ttl` 和 `--expire-time` 选项来固定新创建的备份：

```
lt_probackup backup -B backup_dir --instance instance_name -b FULL --ttl=30d
lt_probackup backup -B backup_dir --instance instance_name -b FULL --expire-
time="2020-01-01 00:00:00+03"
```

若要检查备份是否被固定，运行 `show` 命令：

```
lt_probackup show -B backup_dir --instance instance_name -i backup_id
```

如果备份已被固定，它将具有 `expire-time` 属性，显示其过期时间：

```
...
recovery-time = '2017-05-16 12:57:31'
expire-time = '2020-01-01 00:00:00+03'
data-bytes = 22288792
...
```

您可以通过将 `--ttl` 选项设置为零来取消固定备份：

```
lt_probackup set-backup -B backup_dir --instance instance_name -i backup_id --
ttl=0
```

### Note

固定的增量备份隐式固定了其所有父备份。如果您稍后取消固定此类备份，则其隐式固定的父代备份也将自动取消固定。

## 配置WAL归档保留政策

当启用 `连续WAL归档` 时，归档的WAL段可能会占用大量磁盘空间。即使您不时删除旧的备份副本，`--delete-wal` 标志也只能清除那些不适用于备份目录中任何剩余备份的WAL段。但是，如果对于最近的备份而言关键的是时间点恢复，则可以配置WAL归档保留政策，以保留有限深度的WAL归档，并重新获得更多的磁盘空间。

要配置WAL归档保留政策，您必须使用 `--wal-depth` 选项运行 `set-config` 命令，指定可用于PITR的备份数。此设置适用于所有时间轴，因此如果可用，则应能够对每个时间轴执行相同数量的备份进行PITR。不包括 `固定的备份` 在此计数中：如果最新的备份之一被固定，`lt_probackup` 将确保可以为一份额外的备份进行PITR。

要删除不符合定义的WAL归档保留政策的WAL段，您只需使用 `--delete-wal` 标志运行 `delete` 或 `backup` 命令。对于归档备份，始终保留 `Start LSN` 和 `Stop LSN` 之间的WAL段，因此这种备份保持有效，无论 `--wal-depth` 设置如何，如果需要，仍然可以进行恢复。

您还可以使用 `--wal-depth` 选项，使用 `delete` 和 `backup` 命令覆盖先前定义的WAL归档保留政策，并在运行时清除旧的WAL段。

假设您已经在 backup\_dir 目录中备份了 node 实例，并配置了 连续WAL归档：

```
lt_probackup show -B backup_dir --instance node
```

```
BACKUP INSTANCE 'node'
```

Instance Data	Version WAL	ID Zratio	Recovery Time Start LSN	Stop LSN	Mode Status	WAL Mode	TLI	Time
node	11	PZ9442	2019-10-12 10:43:21+03	0/46000028	0/46000160 OK	DELTA	STREAM	1/0 10s
node	11	PZ943L	2019-10-12 10:43:04+03	0/44000028	0/44000160 OK	FULL	STREAM	1/0 10s
node	11	PZ7YR5	2019-10-11 19:49:56+03	0/41000028	0/41000160 OK	DELTA	STREAM	1/1 10s
node	11	PZ7YMP	2019-10-11 19:47:16+03	0/3E000028	0/3F0000B8 OK	DELTA	STREAM	1/1 10s
node	11	PZ7YK2	2019-10-11 19:45:45+03	0/3C000028	0/3C000198 OK	FULL	STREAM	1/0 11s
node	11	PZ7YFO	2019-10-11 19:43:04+03	0/2000028	0/200ADD8 OK	FULL	STREAM	1/0 10s

您可以通过运行带有--archive标志的 show命令来检查WAL存档的状态：

```
lt_probackup show -B backup_dir --instance node --archive
```

```
ARCHIVE INSTANCE 'node'
```

TLI	Parent TLI	Switchpoint	Min Segno	Max Segno
N segments	Size	Zratio	N backups	Status
1	0	0/0	000000010000000000000001	
0000000100000000000000047	71	36MB	31.00	6 OK

如果没有使用--wal-depth选项进行WAL清除，则不能取得很大效果，只能删除一个段：

```
lt_probackup delete -B backup_dir --instance node --delete-wal
```

```
ARCHIVE INSTANCE 'node'
```

TLI	Parent TLI	Switchpoint	Min Segno	Max Segno
N segments	Size	Zratio	N backups	Status
1	0	0/0	000000010000000000000002	
0000000100000000000000047	70	34MB	32.00	6 OK

如果没有使用--wal-depth选项进行WAL清除，则不能取得很大效果，只能删除一个段：

```
lt_probackup delete -B backup_dir --instance node --delete-wal --wal-depth=1
```

```
ARCHIVE INSTANCE 'node'
```



TLI	Parent TLI	Switchpoint	Min Segno	Max Segno
N segments	Size	Zratio	N backups	Status
1	0	0/0	000000010000000000000046	
000000010000000000000047	2		143kB	228.00 6 OK

或者，您也可以使用`--wal-depth`选项来运行`backup`命令：

```
lt_probackup backup -B backup_dir --instance node -b DELTA --wal-depth=1 --delete-wal
```

```
ARCHIVE INSTANCE 'node'
```

TLI	Parent TLI	Switchpoint	Min Segno	Max Segno
N segments	Size	Zratio	N backups	Status
1	0	0/0	000000010000000000000048	
000000010000000000000049	1		72kB	228.00 7 OK

## 合并备份

随着您进行越来越多的增量备份，备份目录的总大小可能会显著增长。为了节省磁盘空间，您可以运行`merge`命令，将增量备份合并到它们的父全量备份中，指定您想要合并的最新增量备份的备份ID：

```
lt_probackup merge -B backup_dir --instance instance_name -i backup_id
```

此命令合并属于公共增量备份链的备份。如果指定了一个全量备份，则它将其第一个增量备份合并。如果指定了一个增量备份，则它与其父全量备份一起合并，包括它们之间的所有增量备份。合并完成后，全量备份将包含所有合并的数据，并且增量备份将作为冗余数据被删除。因此，合并操作实际上等同于重新拍摄一个全量备份并删除所有过期的备份，但它节省大量时间，特别是对于大数据量的情况，而且如果您使用`lt_probackup`在远程模式下，还可以节省I/O和网络流量。

在合并之前，`lt_probackup`会验证所有受影响的备份以确保它们是有效的。您可以通过运行带有备份ID的`show`命令来检查当前备份的状态：

```
lt_probackup show -B backup_dir --instance instance_name -i backup_id
```

如果合并仍在进行中，则备份状态显示为`MERGING`。对于全量备份，还可以在合并的最后阶段更新元数据时显示为`MERGED`。合并是幂等的，因此如果合并被中断，您可以重新启动合并。

## 删除备份

要删除不再需要的备份，请运行以下命令：

```
lt_probackup delete -B backup_dir --instance instance_name -i backup_id
```

此命令将删除指定的`backup_id`备份，以及任何从`backup_id`备份派生的增量备份。这样，您可以删除一些最近的增量备份，同时保留底层的全量备份和一些随后的增量备份。

要删除不必要的WAL文件以恢复任何剩余备份，使用`--delete-wal`标志：

```
lt_probackup delete -B backup_dir --instance instance_name --delete-wal
```

要删除根据当前保留策略已过期的备份，请使用--delete-expired标志：

```
lt_probackup delete -B backup_dir --instance instance_name --delete-expired
```

只要有至少一个基于过期备份的增量备份满足保留策略，就无法删除过期备份。如果您希望最小化仍需要保留的备份数量以保持增量备份的有效性，请在运行此命令时指定--merge-expired标志：

```
lt_probackup delete -B backup_dir --instance instance_name --delete-expired --merge-expired
```

在这种情况下，lt\_probackup会查找最早的增量备份，以满足保留策略，并将此备份与底层的全量备份和已过期的增量备份合并，从而将其变为全量备份。合并完成后，剩余的过期备份将被删除。

在合并或删除备份之前，您可以运行带有--dry-run标志的delete命令，它会根据当前的保留策略显示所有可用备份的状态，而不执行任何不可逆转的操作。

要删除具有特定状态的所有备份，请使用--status：

```
lt_probackup delete -B backup_dir --instance instance_name --status=ERROR
```

按状态删除备份会忽略已经建立的保留策略。

## 命令行参考

### 命令

本节介绍 lt\_probackup 命令。可选参数用方括号括起来。有关详细参数说明，请参见 选项 部分。

### 版本

```
lt_probackup version
```

输出 lt\_probackup 版本。

### 帮助

```
lt_probackup help [command]
```

显示 lt\_probackup 命令的概要。如果指定了其中一个 lt\_probackup 命令，则显示可用于该命令的选项的详细信息。

### init

```
lt_probackup init -B backup_dir [--help]
```

初始化在backup\_dir中存储备份副本、WAL归档和备份数据库集群的元信息的备份目录。如果指定的backup\_dir已经存在，则必须为空。否则，lt\_probackup会显示相应的错误消息。

详见初始化备份目录。

## add-instance

```
lt_probackup add-instance -B backup_dir -D data_dir --instance instance_name [--help]
```

在备份目录backup\_dir中初始化新的备份实例，并生成控制具有指定data\_dir数据目录的集群的pg\_probackup.conf配置文件，控制lt\_probackup的设置。

详见添加新的备份实例。

## 删除实例

```
lt_probackup del-instance -B backup_dir --instance instance_name [--help]
```

删除与指定实例关联的所有备份和WAL文件。

## set-config

```
lt_probackup set-config -B backup_dir --instance instance_name
[--help] [--pgdata=pgdata-path]
[--retention-redundancy=redundancy] [--retention-window=window] [--wal-
depth=wal_depth]
[--compress-algorithm=compression_algorithm] [--compress-
level=compression_level]
[-d dbname] [-h host] [-p port] [-U username]
[--archive-timeout=timeout] [--external-dirs=external_directory_path]
[--restore-command=cmdline]
[remote_options] [remote_wal_archive_options] [logging_options]
```

将指定的连接、压缩、保留、日志和外部目录设置添加到pg\_probackup.conf配置文件中，或修改先前定义的值。

查看所有可用设置，详见选项部分。

不建议手动编辑pg\_probackup.conf。

## set-backup

```
lt_probackup set-backup -B backup_dir --instance instance_name -i backup_id
{--ttl=ttl | --expire-time=time}
[--note=backup_note] [--help]
```

将提供的备份特定设置设置到backup.control配置文件中，或修改先前定义的值。

```
--note=backup_note
```

设置备份副本的文本备注。如果backup\_note包含换行符，则只保存第一个换行符之前的子字符串。文本备注的最大大小为1 KB。'none'值将删除当前备注。

查看所有可用的固定设置，详见固定选项。

## show-config

```
lt_probackup show-config -B backup_dir --instance instance_name [--format=plain|
json]
```

显示位于 `backup_dir/backups/instance_name` 目录中的 `pg_probackup.conf` 配置文件的内容。您可以指定 `--format=json` 选项以获得JSON格式的结果。默认情况下，将以纯文本形式显示配置设置。

要编辑 `pg_probackup.conf`，请使用 `set-config` 命令。

## 显示

```
lt_probackup show -B backup_dir
[--help] [--instance instance_name [-i backup_id | --archive]] [--format=plain|json]
```

显示备份目录的内容。如果指定了 `instance_name` 和 `backup_id`，则显示有关此备份的详细信息。如果指定了 `--archive` 选项，则显示备份目录的WAL归档内容。

默认情况下，备份目录的内容显示为纯文本。您可以指定 `--format=json` 选项以获取JSON格式的结果。

有关用法的详细信息，请参阅以下部分：[管理备份目录](#) 和 [查看WAL归档信息](#)。

## 备份

```
lt_probackup backup -B backup_dir -b backup_mode --instance instance_name
[--help] [-j num_threads] [--progress]
[-C] [--stream [-S slot_name] [--temp-slot]] [--backup-pg-log]
[--no-validate] [--skip-block-validation]
[-w --no-password] [-W --password]
[--archive-timeout=timeout] [--external-dirs=external_directory_path]
[--no-sync] [--note=backup_note]
[connection_options] [compression_options] [remote_options]
[retention_options] [pinning_options] [logging_options]
```

创建 LightDB 实例的备份副本。

```
-b mode
--backup-mode=mode
```

指定要使用的备份模式。可能的值为：

- FULL — 创建包含要恢复的所有群集数据文件的完整备份。
- DELTA — 读取数据目录中的所有数据文件，并为自上次备份以来更改的页创建增量备份。
- PAGE — 基于自上次完整或增量备份以来更改的 WAL 文件创建增量 PAGE 备份。
- PTRACK — 实时跟踪页面更改，创建增量 PTRACK 备份。

```
-C
--smooth-checkpoint
```

将检查点分散在一段时间内。默认情况下，`lt_probackup` 尝试尽快完成检查点。

```
--stream
```

通过复制数据库服务器通过复制协议将所有必需的 WAL 文件流式传输来创建 STREAM 备份。

```
--temp-slot
```

为从备份的 LightDB 实例流式传输 WAL 创建临时物理复制槽。如果在备份过程中旋转 WAL，则确保所有所需的 WAL 段仍然可用。此标志只能与 `--stream` 标志一起使用。默认槽名称为 `lt_probackup_slot`，可以使用 `--slot/-S` 选项更改。

`-S slot_name`  
`--slot=slot_name`

指定 WAL 流式传输的复制槽。此选项只能与 `--stream` 标志一起使用。

`--backup-pg-log`

将日志目录包含到备份中。此目录通常包含日志消息。默认情况下，日志目录被排除在外。

`-E external_directory_path`  
`--external-dirs=external_directory_path`

通过递归地将其内容复制到备份目录中的单独子目录中，将指定的目录包含到备份中。此选项对于备份位于数据目录之外的脚本、SQL 转储文件和配置文件非常有用。如果要备份多个外部目录，请在 Unix 上使用冒号，在 Windows 上使用分号分隔它们的路径。

`--archive-timeout=wait_time`

设置 WAL 段归档和流式传输的超时时间（以秒为单位）。默认情况下，`lt_probackup` 等待 300 秒。

`--skip-block-validation`

禁用块级校验和验证以加快备份过程。

`--no-validate`

在备份完成后跳过自动验证。如果定期验证备份并希望在运行备份操作时节省时间，则可以使用此标志。

`--no-sync`

不将备份文件同步到磁盘。您可以使用此标志来加快备份过程。使用此标志可能会导致数据损坏，在操作系统或硬件崩溃的情况下会发生数据损坏。如果使用此选项，则建议在备份完成后运行 `validate` 命令以检测可能的问题。

`--note=backup_note`

设置备份副本的文本注释。如果 `backup_note` 包含换行符，则只会保存第一个换行符之前的子字符串。文本注释的最大大小为 1 KB。将值设置为 `'none'` 可以删除当前注释。

此外，可以使用以下选项：连接选项、保留选项、固定选项、远程模式选项、压缩选项、日志选项 和 常用选项。

有关使用详细信息，请参阅 [创建备份](#) 部分。

## 还原

```
lt_probackup restore -B backup_dir --instance instance_name
[--help] [-D data_dir] [-i backup_id]
[-j num_threads] [--progress]
[-T OLDDIR=NEWDIR] [--external-mapping=OLDDIR=NEWDIR] [--skip-external-dirs]
[-R | --restore-as-replica] [--no-validate] [--skip-block-validation]
[--force] [--no-sync]
[--restore-command=cmdline]
[--primary-conninfo=primary_conninfo]
[-S | --primary-slot-name=slot_name]
[recovery_target_options] [logging_options] [remote_options]
```

[partial\_restore\_options] [remote\_wal\_archive\_options]

从位于 `backup_dir` 备份目录中的备份副本还原 LightDB 实例。如果指定了 恢复目标选项, `lt_probackup` 将查找最接近的备份并将其还原到指定的恢复目标。如果未提供备份ID或恢复目标选项, 则 `lt_probackup` 使用最近的备份来执行恢复。

-R

`--restore-as-replica`

创建一个最小的恢复配置文件, 以便设置一个备用服务器。如果复制连接需要密码, 则必须在 `primary_conninfo`<sup>28</sup> 参数中手动指定密码, 因为它不会被包含在内。`lt_probackup` 将这些设置写入数据目录中的 `probackup_recovery.conf` 文件中, 然后在启动集群时将它们包含到 `lightdb.auto.conf` 中。

`--primary-conninfo=primary_conninfo`

将 `primary_conninfo`<sup>29</sup> 参数设置为指定值。除非指定了 -R 标志, 否则将忽略此选项。

例如: `--primary-conninfo="host=192.168.1.50 port=5432 user=foo password=foopass"`

-S

`--primary-slot-name=slot_name`

将 `primary_slot_name`<sup>30</sup> 参数设置为指定值。除非指定 -R 标志, 否则将忽略此选项。

-T OLDDIR=NEWDIR

`--tablespace-mapping=OLDDIR=NEWDIR`

将表空间从 OLDDIR 移动到 NEWDIR 目录进行恢复。OLDDIR 和 NEWDIR 必须是绝对路径。如果路径包含等号 (=), 请使用反斜杠进行转义。可以使用此选项多次指定多个表空间。

`--external-mapping=OLDDIR=NEWDIR`

将备份中包含的外部目录从 OLDDIR 移动到 NEWDIR 目录进行恢复。OLDDIR 和 NEWDIR 必须是绝对路径。如果路径包含等号 (=), 请使用反斜杠进行转义。可以使用此选项多次指定多个目录。

`--skip-external-dirs`

跳过使用 `--external-dirs` 选项包含的外部目录。这些目录的内容将不会被恢复。

`--skip-block-validation`

禁用块级校验和验证, 以加快验证速度。在恢复之前自动验证期间, 只会验证文件级别的校验和。

`--no-validate`

跳过备份验证。如果您定期验证备份并且希望在运行恢复操作时节省时间, 可以使用此标志。

`--restore-command=cmdline`

将 `restore_command`<sup>31</sup> 参数设置为指定的命令。例如: `--restore-command='cp /mnt/server/archivedir/%f "%p"'`

<sup>28</sup> runtime-config-replication.html#GUC-PRIMARY-CONNINFO

<sup>29</sup> runtime-config-replication.html#GUC-PRIMARY-CONNINFO

<sup>30</sup> runtime-config-replication.html#GUC-PRIMARY-SLOT-NAME

<sup>31</sup> archive-recovery-settings.html#RESTORE-COMMAND

`--force`

允许忽略备份的无效状态。如果需要从损坏或无效备份恢复 LightDB 集群，可以使用此标志。请谨慎使用。如果 LTDATA 包含具有与要恢复的备份不同的系统 ID 的非空目录，则使用此标志的增量恢复将覆盖目录内容（未使用该标志会出现错误）。如果通过 `--tablespace-mapping` 选项将表空间重新映射到非空目录中，则会删除此类目录的内容。

`--no-sync`

不将恢复的文件同步到磁盘。您可以使用此标志加快恢复进程。使用此标志可能会导致数据损坏，如果发生操作系统或硬件崩溃，您必须再次运行 `restore` 命令。

另外，可以使用恢复目标选项、远程模式选项、远程 WAL 存档选项、日志选项、部分恢复选项和通用选项。

有关用法的详细信息，请参见恢复群集部分。

## checkdb

```
lt_probackup checkdb
[-B backup_dir] [--instance instance_name] [-D data_dir]
[--help] [-j num_threads] [--progress]
[--skip-block-validation] [--amcheck] [--heapallindexed]
[connection_options] [logging_options]
```

通过检测物理和逻辑损坏，验证 LightDB 数据库群集的正确性。

`--amcheck`

如果在检查数据文件时没有发现损坏，将为指定的 LightDB 实例执行索引的逻辑验证。您必须在数据库中安装 `amcheck` 扩展或 `amcheck_next` 扩展才能检查其索引。对于没有 `amcheck` 的数据库，将跳过索引验证。

`--skip-block-validation`

跳过数据文件的验证。您只能与 `--amcheck` 标志一起使用此标志，以便仅执行索引的逻辑验证。

`--heapallindexed`

检查所有应该被索引的堆元组是否实际上被索引。您只能与 `--amcheck` 标志一起使用此标志。

仅当您使用版本为 2.0 或更高版本的 `amcheck` 扩展，或任何版本的 `amcheck_next` 扩展时，才能进行此检查。

另外，可以使用连接选项和日志选项。

有关用法的详细信息，请参见验证群集部分。

## validate

```
lt_probackup validate -B backup_dir
[--help] [--instance instance_name] [-i backup_id]
[-j num_threads] [--progress]
[--skip-block-validation]
[recovery_target_options] [logging_options]
```

验证恢复群集所需的所有文件是否存在且未损坏。如果未指定 `instance_name`, `lt_probackup` 将验证备份目录中所有可用的备份。如果您仅指定了 `instance_name`, 没有任何其他选项, `lt_probackup` 将验证此备份实例中所有可用的备份。如果您指定了 `instance_name` 和 恢复目标选项 和/或 `backup_id`, `lt_probackup` 将检查使用这些选项是否可以还原群集。

有关详细信息, 请参见 验证备份 部分。

## merge

```
lt_probackup merge -B backup_dir --instance instance_name -i backup_id
[--help] [-j num_threads] [--progress] [--no-validate] [--no-sync]
[logging_options]
```

合并属于公共增量备份链的备份。如果指定完全备份, 则将其与其第一个增量备份合并。如果指定增量备份, 则将其与其父完全备份一起合并, 以及它们之间的所有增量备份。合并完成后, 完全备份收取所有合并的数据, 并将增量备份作为冗余项删除。

`--no-validate`

跳过合并前后的自动验证。

`--no-sync`

不将合并的文件同步到磁盘。您可以使用此标志加速合并过程。使用此标志可能会导致操作系统或硬件崩溃的数据损坏。

有关详细信息, 请参见 合并备份 部分。

## delete

```
lt_probackup delete -B backup_dir --instance instance_name
[--help] [-j num_threads] [--progress]
[--retention-redundancy=redundancy][--retention-window=window][--wal-
depth=wal_depth] [--delete-wal]
{-i backup_id | --delete-expired [--merge-expired] | --merge-expired | --
status=backup_status}
[--dry-run] [--no-validate] [--no-sync] [logging_options]
```

删除指定 `backup_id` 的备份, 或启动保留保留政策不满足的备份和归档 WAL 的清除。

`--no-validate`

跳过保留合并前后的自动验证。

`--no-sync`

不将合并的文件同步到磁盘。您可以使用此标志加速保留合并过程。使用此标志可能会导致操作系统或硬件崩溃的数据损坏。

有关详细信息, 请参见 删除备份、保留选项 和 配置保留策略 部分。

## archive-push

```
lt_probackup archive-push -B backup_dir --instance instance_name
--wal-file-name=wal_file_name [--wal-file-path=wal_file_path]
[--help] [--no-sync] [--compress] [--no-ready-rename] [--overwrite]
[-j num_threads] [--batch-size=batch_size]
[--archive-timeout=timeout]
```



```
[--compress-algorithm=compression_algorithm]
[--compress-level=compression_level]
[remote_options] [logging_options]
```

将 WAL 文件复制到备份目录的相应子目录中，并通过 `instance_name` 和 `system-identifier` 验证备份实例。如果备份实例和群集的参数不匹配，则此命令将失败，并显示以下错误消息：拒绝将 WAL 段 `segment_name` 推送到归档。实例参数不匹配。

如果要复制的文件已存在于备份目录中，`lt_probackup` 将计算并比较它们的校验和。如果校验和匹配，则 `archive-push` 跳过相应的文件并返回成功的执行代码。否则，`archive-push` 将失败并显示错误。如果您想在校验和不匹配的情况下替换 WAL 文件，请使用 `--overwrite` 标志运行 `archive-push` 命令。

每个文件都会复制到具有 `.part` 后缀的临时文件中。如果临时文件已经存在，则 `lt_probackup` 将等待 `archive_timeout` 秒后将其丢弃。复制完成后，执行原子重命名。此算法确保失败的 `archive-push` 不会阻塞连续归档，并且从多个来源并发归档到单个 WAL 归档没有存档损坏的风险。

为加速归档，您可以指定 `--batch-size` 选项，以指定批量复制 WAL 段的大小。如果使用了 `--batch-size` 选项，则还可以指定 `-j` 选项，以在多个线程上复制 WAL 段批次。

复制到归档的 WAL 段将同步到磁盘，除非使用 `--no-sync` 标志。

您可以在 `archive_command`<sup>32</sup> LightDB 参数中使用 `archive-push` 来设置连续 WAL 归档。

有关详细信息，请参见 归档选项 和 压缩选项 部分。

## archive-get

```
lt_probackup archive-get -B backup_dir --instance instance_name --wal-file-
path=wal_file_path --wal-file-name=wal_file_name
[-j num_threads] [--batch-size=batch_size]
[--prefetch-dir=prefetch_dir_path] [--no-validate-wal]
[--help] [remote_options] [logging_options]
```

将 WAL 文件从备份目录的相应子目录复制到群集的预写日志位置。在使用 WAL 归档恢复备份时，此命令会作为 `restore_command` 的一部分自动由 `lt_probackup` 设置。您无需手动设置它。

为加速恢复，您可以指定 `--batch-size` 选项，以指定批量复制指定大小的 WAL 段。如果使用了 `--batch-size` 选项，则还可以指定 `-j` 选项，以在多个线程上复制 WAL 段批次。

有关详细信息，请参见 归档选项 部分。

## 选项

This section describes command-line options for `lt_probackup` commands. If the option value can be derived from an environment variable, this variable is specified below the command-line option, in the uppercase. Some values can be taken from the `pg_probackup.conf` configuration file located in the backup catalog.

详见 the section called “配置lt\_probackup”。

如果使用多种方法指定了一个选项，则命令行输入具有最高优先级，而 `pg_probackup.conf` 设置具有最低优先级。

## 常用选项

一般选项列表。

<sup>32</sup> runtime-config-wal.html#GUC-ARCHIVE-COMMAND

-B directory  
 --backup-path=directory  
 BACKUP\_PATH

指定备份目录的绝对路径。备份目录是存储所有备份文件和元信息的目录。由于大多数 `lt_probackup` 命令都需要此选项，建议您在 `BACKUP_PATH` 环境变量中指定它。在这种情况下，您无需每次在命令行中使用此选项。

-D directory  
 --pgdata=directory  
 LTDATA

指定数据库群集数据目录的绝对路径。此选项指仅对 `add-instance` 命令强制执行。其他命令可以从 `LTDATA` 环境变量或 `pg_probackup.conf` 配置文件中获取其值。

-i backup\_id  
 --backup-id=backup\_id

指定备份的唯一标识符。

-j num\_threads  
 --threads=num\_threads

为 `backup`、`restore`、`merge`、`validate`、`checkdb` 和 `archive-push` 进程设置并行线程数。

--progress

显示操作的进度。

--help

显示有关可与此命令一起使用的选项的详细信息。

## 恢复目标选项

如果已经配置了连续的 WAL 归档，您可以使用以下选项和 `restore` 或 `validate` 命令共同指定必须恢复或验证数据库集群的时间点。

--recovery-target=immediate|latest

定义恢复的停止时间：

- 当设置为 `immediate` 时，在恢复到指定备份的一致状态之后，或者当省略了 `-i/--backup-id` 选项时，恢复到最新可用的备份后停止。当使用 `STREAM` 备份时，这是默认行为。
- 当设置为 `latest` 时，会一直进行恢复，直到应用存档中的所有 WAL 分段。当使用 `ARCHIVE` 备份时，这是默认行为。

--recovery-target-timeline=timeline

指定用于恢复的特定时间线。默认情况下，使用指定备份的时间线。

--recovery-target-lsn=lsn

指定写入前日志位置的 LSN，恢复将进行到该位置。仅可在恢复主版本号为 10 或更高版本的数据库集群时使用。

--recovery-target-name=recovery\_target\_name

指定要恢复集群的命名保存点。

`--recovery-target-time=time`

指定恢复到的时间戳。如果未指定时区偏移量，则使用本地时区。

例如：`--recovery-target-time="2020-01-01 00:00:00+03"`

`--recovery-target-xid=xid`

指定事务 ID，恢复将进行到该事务。

`--recovery-target-inclusive=boolean`

指定是否在恢复目标点之后立即停止（true），还是在目标点之前停止（false）。该选项只能与 `--recovery-target-name`、`--recovery-target-time`、`--recovery-target-lsn` 或 `--recovery-target-xid` 选项一起使用。默认行为取决于 `recovery_target_inclusive`<sup>33</sup> 参数。

`--recovery-target-action=pause|promote|shutdown`

指定恢复到达恢复目标点时服务器应采取的操作<sup>34</sup>。

默认设置为 `pause`。

## 保留选项

你可以与 `backup` 和 `delete` 命令一起使用这些选项。

有关配置保留策略的详细信息，请参阅 `配置保留策略` 部分。

`--retention-redundancy=redundancy`

指定数据目录中保留完整备份副本的数量。必须是非负整数。零值将禁用此设置。

默认值：0

`--retention-window=window`

恢复的天数。必须是非负整数。零值将禁用此设置。

默认值：0

`--wal-depth=wal_depth`

每个时间线上必须保留最近的有效备份的数量，以便执行 PITR。必须是非负整数。零值将禁用此设置。

默认值：0

`--delete-wal`

删除不再需要的 WAL 文件，以便从任何现有备份还原集群。

`--delete-expired`

删除不符合 `pg_probackup.conf` 配置文件中定义的保留策略的备份。

`--merge-expired`

将最早满足保留策略要求的增量备份与其已过期的父备份合并。

<sup>33</sup> `runtime-config-wal#GUC-RECOVERY-TARGET-INCLUSIVE`

<sup>34</sup> `recovery-target-settings.html#RECOVERY-TARGET-ACTION`

`--dry-run`

显示所有可用备份的当前状态，而不删除或合并任何过期的备份。

## 固定选项

你可以与 `backup` 和 `set-backup` 命令一起使用这些选项。

有关备份固定的详细信息，请参阅 [备份固定](#) 部分。

`--ttl=ttl`

指定备份应保持固定的时间。必须是非负整数。零值将取消已固定的备份。支持的单位：`ms`、`s`、`min`、`h`、`d`（默认为 `s`）。

示例：`--ttl=30d`

`--expire-time=time`

指定备份将保持固定的时间戳。必须是符合 ISO-8601 标准的时间戳。如果未指定时区偏移量，则使用本地时区。

示例：`--expire-time="2020-01-01 00:00:00+03"`

## 日志选项

你可以与任何命令一起使用这些选项。

`--log-level-console=log_level`

控制哪些信息级别将被发送到控制台日志。有效值为 `verbose`、`log`、`info`、`warning`、`error` 和 `off`。每个级别包括其后面的所有级别。级别越靠后，发送的消息越少。`off` 级别禁用控制台日志记录。

默认值：`info`

### Note

所有控制台日志消息都会发送到 `stderr`，因此 `show` 和 `show-config` 命令的输出不会与日志消息混在一起。

`--log-level-file=log_level`

控制哪些信息级别将被发送到日志文件。有效值为 `verbose`、`log`、`info`、`warning`、`error` 和 `off`。每个级别包括其后面的所有级别。级别越靠后，发送的消息越少。`off` 级别禁用文件日志记录。

默认值：`off`

`--log-filename=log_filename`

定义所创建的日志文件的文件名。文件名被视为 `strftime` 模式，因此您可以使用 `%-` 转义符指定变化的文件名。

默认值：`pg_probackup.log`

例如，如果您指定 `lt_probackup-%u.log` 模式，`lt_probackup` 将为每个星期的每一天生成一个单独的日志文件，其中 `%u` 被相应的十进制数替换：周一的 `lt_probackup-1.log`，周二的 `lt_probackup-2.log`，以此类推。

如果启用了 `--log-level-file` 选项，则此选项会生效。

`--error-log-filename=error_log_filename`

仅为错误消息定义日志文件的文件名。文件名被视为`strftime`模式，因此您可以使用`%-`转义符指定变化的文件名。

默认值：无

例如，如果您指定`error-lt_probackup-%u.log`模式，`lt_probackup`将为每个星期的每一天生成一个单独的日志文件，其中`%u`被相应的十进制数替换：周一的`error-lt_probackup-1.log`，周二的`error-lt_probackup-2.log`，以此类推。

此选项对故障排除和监控非常有用。

`--log-directory=log_directory`

定义将创建日志文件的目录。必须指定绝对路径。此目录是惰性创建的，在第一条日志消息被写入时创建。

默认值：`$BACKUP_PATH/log/`

`--log-rotation-size=log_rotation_size`

单个日志文件的最大大小。如果达到此值，`lt_probackup`命令启动时会将日志文件轮换一次，除了`help`和`version`命令外。零值禁用基于大小的轮换。支持的单位：`kB`、`MB`、`GB`、`TB`（默认为`kB`）。

默认值：0

`--log-rotation-age=log_rotation_age`

单个日志文件的最长存活时间。如果达到此值，`lt_probackup`命令启动时会将日志文件轮换一次，除了`help`和`version`命令外。上一次日志文件创建的时间存储在`$BACKUP_PATH/log/log_rotation`中。零值禁用基于时间的轮换。支持的单位：`ms`、`s`、`min`、`h`、`d`（默认为`min`）。

默认值：0

## 连接选项

您可以将这些选项与`backup`和`checkdb`命令一起使用。

所有 `libpq` 环境变量<sup>35</sup>都被支持。

`-d dbname`

`--pgdatabase=dbname`

`LTDATABASE`

指定要连接的数据库名称。连接仅用于管理备份进程，因此您可以连接到任何现有的数据库。如果没有在命令行中提供此选项，则 `LTDATABASE` 环境变量或 `pg_probackup.conf` 配置文件将尝试从 `LTUSER` 环境变量中获取此值，如果未设置 `LTUSER` 变量，则从当前用户名称中获取。

`-h host`

`--pghost=host`

`LTHOST`

指定服务器运行的系统的主机名。如果值以斜杠开头，则将其用作 `Unix` 域套接字的目录。

默认值：`localhost`

<sup>35</sup> `libpq-envvars.html`

-p port  
 --pgport=port  
 LTPORT

指定服务器用于侦听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。

默认值: 5432

-U username  
 --pguser=username  
 LTUSER

要连接的用户名。

-w  
 --no-password

禁用密码提示。如果服务器需要密码身份验证，但没有其他可用的密码（例如 `.pgpass`<sup>36</sup> 文件或 `PGPASSWORD` 环境变量），连接尝试将失败。此标志可用于批处理作业和脚本，其中没有用户可以输入密码。

-W  
 --password

强制密码提示。（已过时）

## 压缩选项

您可以与 `backup` 和 `archive-push` 命令一起使用这些选项。

--compress-algorithm=compression\_algorithm

定义用于压缩数据文件的算法。可能的值包括 `zlib`、`pglz` 和 `none`。如果设置为 `zlib` 或 `pglz`，则启用压缩。默认情况下，压缩已禁用。对于 `archive-push` 命令，不支持 `pglz` 压缩算法。

默认值: none

--compress-level=compression\_level

定义压缩级别（0 到 9，0 表示无压缩，9 表示最佳压缩）。此选项可以与 `--compress-algorithm` 选项一起使用。

默认值: 1

--compress

`--compress-algorithm=zlib` 和 `--compress-level=1` 的别名。

## 归档选项

这些选项可以与 `archive-push` 命令中的 `archive_command`<sup>37</sup> 设置以及 `archive-get` 命令中的 `restore_command`<sup>38</sup> 设置一起使用。

此外，还可以使用 远程模式选项 和 日志记录选项。

--wal-file-path=wal\_file\_path

提供 `archive_command` 和 `restore_command` 中 WAL 文件的路径。请使用 `%p` 变量作为此选项的值以进行正确处理。

<sup>36</sup> libpq-pgpass.html

<sup>37</sup> runtime-config-wal.html#GUC-ARCHIVE-COMMAND

<sup>38</sup> archive-recovery-settings.html#RESTORE-COMMAND

`--wal-file-name=wal_file_name`

提供 `archive_command` 和 `restore_command` 中 WAL 文件的名称。请使用 `%f` 变量作为此选项的值以正确处理。

`--overwrite`

覆盖已归档的 WAL 文件。如果备份目录的指定子目录已包含此 WAL 文件并需要用其较新的副本替换它，则与 `archive-push` 命令一起使用此标志。否则，`archive-push` 报告 WAL 段已存在，并终止操作。如果要替换的文件未更改，则无论 `--overwrite` 标志如何，`archive-push` 都会跳过此文件。

`--batch-size=batch_size`

设置单个 `archive-push` 进程复制到归档的文件的最大数量，或设置单个 `archive-get` 进程从归档中复制的文件的最大数量。

`--archive-timeout=wait_time`

设置将现有的 `.part` 文件视为过时的超时时间。默认情况下，`lt_probackup` 等待 300 秒。此选

`--no-ready-rename`

不要重命名 `archive_status` 目录中的状态文件。仅当 `archive_command` 包含多个命令时才应使用此选项。此选项只能与 `archive-push` 命令一起使用。

`--no-sync`

不要将复制的 WAL 文件同步到磁盘。您可以使用此标志以加速归档过程。使用此标志可能导致 WAL 归档损坏，以防操作系统或硬件崩溃。此选项只能与 `archive-push` 命令一起使用。

`--prefetch-dir=path`

如果使用 `--batch-size` 选项，则用于存储预取的 WAL 段的目录。目录必须位于相同的文件系统和相同的挂载点上，`LTDATA/lt_wal` 也位于该位置。默认情况下，文件存储在 `LTDATA/lt_wal/pbk_prefetch` 目录中。此选项只能与 `archive-get` 命令一起使用。

`--no-validate-wal`

不要在使用之前验证预取的 WAL 文件。如果要提高恢复速度，请使用此选项。此选项只能与 `archive-get` 命令一起使用。

## 远程模式选项

本节介绍与通过 SSH 远程运行 `lt_probackup` 操作相关的选项。这些选项可以与 `add-instance`、`set-config`、`backup`、`restore`、`archive-push` 和 `archive-get` 命令一起使用。

有关配置和使用远程模式的详细信息，请参阅 the section called “配置远程模式” 和 the section called “使用远程模式下的 `lt_probackup`”。

`--remote-proto=proto`

指定远程操作使用的协议。目前只支持SSH协议。可能的值为：

- `ssh` 启用SSH连接的远程模式。这是默认值。
- `none` 显式禁用远程模式。

如果指定了 `--remote-host` 选项，则可以省略此选项。

`--remote-host=destination`

指定要连接的远程主机IP地址或主机名。

`--remote-port=port`

指定要连接的远程主机端口。

默认值：22

`--remote-user=username`

指定SSH连接的远程主机用户。如果省略此选项，则使用启动SSH连接的当前用户。

`--remote-path=path`

指定远程系统上lt\_probackup的安装目录。

`--ssh-options=ssh_options`

提供一组SSH命令行选项。例如，可以使用以下选项为lt\_probackup打开的SSH连接设置keep-alive：`--ssh-options="-o ServerAliveCountMax=5 ServerAliveInterval=60"`。有关可能选项的完整列表，请参见 `ssh_config` 手册页<sup>39</sup>。

## 远程WAL归档选项

本节介绍用于提供archive-get中使用的远程模式选项（远程模式选项）的参数，这些选项在恢复归档备份或执行PITR时在restore\_command<sup>40</sup>命令中使用。

`--archive-host=destination`

提供--remote-host选项在archive-get命令中的参数。

`--archive-port=port`

提供--remote-port选项在archive-get命令中的参数。

默认值：22

`--archive-user=username`

提供--remote-user选项在archive-get命令中的参数。如果省略此选项，则使用启动LightDB集群的用户。

默认值：LightDB用户

## 增量恢复选项

本节介绍用于增量集群恢复的选项。这些选项可以与restore命令一起使用。

`-I incremental_mode`

`--incremental-mode=incremental_mode`

指定要使用的增量模式。可能的值为：

- CHECKSUM — 仅替换校验和和LSN不匹配的页。
- LSN — 仅替换LSN大于分歧点的页。
- NONE — 常规恢复。

## 部分恢复选项

本节介绍用于部分集群恢复的选项。这些选项可以与restore命令一起使用。

<sup>39</sup> [https://man.openbsd.org/ssh\\_config.5](https://man.openbsd.org/ssh_config.5)

<sup>40</sup> [archive-recovery-settings.html#RESTORE-COMMAND](#)



`--db-exclude=dbname`

指定要从恢复中排除的数据库的名称。集群中的所有其他数据库都将像往常一样进行恢复，包括template0和template1。此选项可以为多个数据库指定多次。

`--db-include=dbname`

指定要从备份中恢复的数据库的名称。集群中的所有其他数据库都不会被恢复，除了template0和template1之外。此选项可以为多个数据库指定多次。

## 副本选项

本节介绍与从备用节点备份相关的选项。

### Note

从lt\_probackup 2.0.24开始，可以在备用节点上进行备份而无需连接到主服务器，因此这些选项不再需要。在较早的版本中，lt\_probackup需要连接到主服务器来确定恢复时间，即可以恢复数据库集群的一致状态的最早时间点。

`--master-db=dbname`

已弃用。指定连接的主服务器上的数据库名称。连接仅用于管理备份过程，因此可以连接到任何现有的数据库。可以使用set-config命令在pg\_probackup.conf中设置。

默认值：postgres，默认的LightDB数据库。

`--master-host=host`

已弃用。指定主服务器运行的系统的主机名。

`--master-port=port`

已弃用。指定主服务器侦听连接的TCP端口或本地Unix域套接字文件扩展名。

默认值：5432，LightDB默认端口。

`--master-user=username`

已弃用。要连接的用户名。

默认值：lightdb，LightDB默认用户名。

`--replica-timeout=timeout`

已弃用。通过复制等待WAL段流的时间，以秒为单位。默认情况下，lt\_probackup等待300秒。您也可以使用set-config命令在pg\_probackup.conf配置文件中定义此参数。

默认值：300 sec。

## How-To

以下所有示例均假定通过SSH进行远程操作。如果您计划在本地运行备份和还原操作，请跳过“设置无密码SSH连接”步骤并省略所有选项`--remote*`。

示例基于Centos 7.9、LightDB 22.3和lt\_probackup 2.4.15。

- backup — 用于连接到LightDB集群的LightDB角色。
- backupdb — 用于连接到LightDB集群的数据库。
- backup\_host — 包含备份目录的主机。

- backupman — 运行所有lt\_probackup操作的backup\_host用户。
- /mnt/backups — 存储备份目录的backup\_host目录。
- lightdb\_host — 包含LightDB集群的主机。
- lightdb — 在lightdb\_host上启动LightDB集群的用户。
- /var/lib/postgresql/11/main — LightDB 数据目录位于 lightdb\_host 上。

## 最小化安装

该场景演示了如何设置独立的 FULL 和 DELTA 备份。

1. 从 backup\_host 到 lightdb\_host 设置无密码 SSH 连接:

```
[backupman@backup_host] ssh-copy-id lightdb@lightdb_host
```

2. 配置你的 LightDB 集群。

出于安全考虑, 建议使用单独的数据库进行备份操作。

```
lightdb@postgres=#  
CREATE DATABASE backupdb;
```

连接到 backupdb 数据库, 创建 probackup 角色, 并赋予该角色以下权限:

```
backupdb=#  
BEGIN;  
CREATE ROLE backup WITH LOGIN REPLICATION;  
GRANT USAGE ON SCHEMA pg_catalog TO backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.current_setting(text) TO backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.pg_is_in_recovery() TO backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.pg_start_backup(text, boolean,  
boolean) TO backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.pg_stop_backup(boolean, boolean) TO  
backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.pg_create_restore_point(text) TO  
backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.pg_switch_wal() TO backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.pg_last_wal_replay_lsn() TO backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current() TO backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current_snapshot() TO backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.txid_snapshot_xmax(txid_snapshot) TO  
backup;  
GRANT EXECUTE ON FUNCTION pg_catalog.pg_control_checkpoint() TO backup;  
COMMIT;
```

3. 初始化备份目录:

```
[backupman@backup_host]$ lt_probackup init -B /mnt/backups  
INFO: Backup catalog '/mnt/backups' successfully initied
```

4. 将实例 node 添加到备份目录中:

```
[backupman@backup_host]$ lt_probackup add-instance -B /mnt/backups --
instance node --remote-host=lightdb_host --remote-user=lightdb -D /var/lib/
postgres/11/main
INFO: Instance 'node' successfully initied
```

#### 5. 进行完全备份:

```
[backupman@backup_host] lt_probackup backup -B /mnt/backups --instance
node -b FULL --stream --remote-host=lightdb_host --remote-user=lightdb -U
backup -d backupdb
INFO: Backup start, lt_probackup version: 2.4.15, instance: node, backup
ID: PZ7YK2, backup mode: FULL, wal mode: STREAM, remote: true, compress-
algorithm: none, compress-level: 1
INFO: Start transferring data files
INFO: Data files are transferred
INFO: wait for pg_stop_backup()
INFO: pg_stop_backup() successfully executed
INFO: Validating backup PZ7YK2
INFO: Backup PZ7YK2 data files are valid
INFO: Backup PZ7YK2 resident size: 196MB
INFO: Backup PZ7YK2 completed
```

#### 6. 让我们来看看备份目录:

```
[backupman@backup_host] lt_probackup show -B /mnt/backups --instance node

BACKUP INSTANCE 'node'
```

```
=====
Instance Version ID      Recovery Time           Mode  WAL Mode  TLI
Time  Data  WAL  Zratio  Start LSN  Stop LSN  Status
=====
node   13      PZ7YK2  2019-10-11 19:45:45+03  FULL  STREAM    1/0
11s 180MB 16MB   1.00  0/3C000028  0/3C000198  OK
=====
```

#### 7. 以 DELTA 模式进行增量备份:

```
[backupman@backup_host] lt_probackup backup -B /mnt/backups --instance
node -b delta --stream --remote-host=lightdb_host --remote-user=lightdb -U
backup -d backupdb
INFO: Backup start, lt_probackup version: 2.4.15, instance: node, backup
ID: PZ7YMP, backup mode: DELTA, wal mode: STREAM, remote: true, compress-
algorithm: none, compress-level: 1
INFO: Parent backup: PZ7YK2
INFO: Start transferring data files
INFO: Data files are transferred
INFO: wait for pg_stop_backup()
INFO: pg_stop_backup() successfully executed
INFO: Validating backup PZ7YMP
INFO: Backup PZ7YMP data files are valid
INFO: Backup PZ7YMP resident size: 32MB
INFO: Backup PZ7YMP completed
```

8. 让我们向 `lt_probackup` 配置文件中添加一些参数，这样你就可以在命令行中省略它们：

```
[backupman@backup_host] lt_probackup set-config -B /mnt/backups --instance
node --remote-host=lightdb_host --remote-user=lightdb -U backup -d backupdb
```

9. 以 DELTA 模式再次进行增量备份，省略一些先前的参数：

```
[backupman@backup_host] lt_probackup backup -B /mnt/backups --instance
node -b delta --stream
INFO: Backup start, lt_probackup version: 2.4.15, instance: node, backup
ID: PZ7YR5, backup mode: DELTA, wal mode: STREAM, remote: true, compress-
algorithm: none, compress-level: 1
INFO: Parent backup: PZ7YMP
INFO: Start transferring data files
INFO: Data files are transferred
INFO: wait for pg_stop_backup()
INFO: pg_stop_backup() successfully executed
INFO: Validating backup PZ7YR5
INFO: Backup PZ7YR5 data files are valid
INFO: Backup PZ7YR5 resident size: 32MB
INFO: Backup PZ7YR5 completed
```

10. 让我们来看看实例配置：

```
[backupman@backup_host] lt_probackup show-config -B /mnt/backups --
instance node
```

```
# Backup instance information
pgdata = /var/lib/postgresql/11/main
system-identifier = 6746586934060931492
xlog-seg-size = 16777216
# Connection parameters
pgdatabase = backupdb
pghost = lightdb_host
pguser = backup
# Replica parameters
replica-timeout = 5min
# Archive parameters
archive-timeout = 5min
# Logging parameters
log-level-console = INFO
log-level-file = OFF
log-filename = lt_probackup.log
log-rotation-size = 0
log-rotation-age = 0
# Retention parameters
retention-redundancy = 0
retention-window = 0
wal-depth = 0
# Compression parameters
compress-algorithm = none
```

```
compress-level = 1
# Remote access parameters
remote-proto = ssh
remote-host = lightdb_host
```

注意，我们正在获取其他选项的默认值，这些选项没有被 `set-config` 命令覆盖。

11. 让我们来看看备份目录：

```
[backupman@backup_host] lt_probackup show -B /mnt/backups --instance node
```

```
=====
 Instance Version ID      Recovery Time           Mode  WAL Mode  TLI
Time   Data   WAL  Zratio  Start LSN   Stop LSN   Status
=====
```

Instance	Version	ID	Recovery Time	Mode	WAL Mode	TLI
node	11	PZ7YR5	2019-10-11 19:49:56+03	DELTA	STREAM	1/1
10s	112kB	32MB	1.00 0/41000028 0/41000160	OK		
node	11	PZ7YMP	2019-10-11 19:47:16+03	DELTA	STREAM	1/1
10s	376kB	32MB	1.00 0/3E000028 0/3F0000B8	OK		
node	11	PZ7YK2	2019-10-11 19:45:45+03	FULL	STREAM	1/0
11s	180MB	16MB	1.00 0/3C000028 0/3C000198	OK		

```
=====
```

---

# lt\_receivewal

lt\_receivewal — 以流的方式从一个LightDB服务器得到预写式日志

## Synopsis

```
lt_receivewal [option...]
```

## 描述

lt\_receivewal被用来从一个运行着的LightDB集簇以流的方式得到预写式日志。预写式日志会被使用流复制协议以流的方式传送，并且被写入到文件的一个本地目录。这个目录可以被用作归档位置来做一次使用时间点恢复的恢复（见Section 23.3）。

当预写式日志在服务器上被产生时，lt\_receivewal实时以流的方式传输预写式日志，并且不像archive\_command那样等待段完成。由于这个原因，在使用lt\_receivewal时不必设置archive\_timeout。

与 LightDB 后备服务器上的 WAL 接收进程不同，lt\_receivewal默认只在一个 WAL 文件被关闭时才刷入 WAL 数据。要实时刷入 WAL 数据，必须指定选项--synchronous。由于lt\_receivewal不应用于WAL，当synchronous\_commit 等于 remote\_apply时，你将不允许它成为同步备用。如果发生这样的情况，它将成为一个永远不能拉起的备用数据库，并且会导致事务提交阻塞。为了避免这种情况，你应该为synchronous\_standby\_names配置一个适当的值，或规定为lt\_receivewal 的application\_name 与它不匹配，或将synchronous\_commit的值更改为remote\_apply以外的内容。

预写式日志在一个常规LightDB连接上被以流式传送，并且使用复制协议。连接必须由一个具有REPLICATION权限（见Section 19.2 的用户或者一个超级用户建立，并且lt\_hba.conf必须允许复制连接。服务器也必须被配置一个足够高的max\_wal\_senders来至少留出一个可用会话给流。

The starting point of the write-ahead log streaming is calculated when lt\_receivewal starts:

1. 首先，扫描WAL段文件所写入的目录，并发现最新完成的段文件，作为下一个段文件的开始的起始点。这是独立计算的，根据用于压缩每个段的压缩方法。
2. 如果用前面的方法无法计算出起点，最新的WAL刷写位置用作由服务器通过IDENTIFY\_SYSTEM命令的报告。

如果该连接丢失，或者它一开始就由于一个非致命错误而没有被建立，lt\_receivewal将无限期地重试连接并且尽可能重新建立流。为了避免这种行为，使用-n参数。

如果不出现致命错误，lt\_receivewal将一直运行直至被SIGINT信号（Control+C）终止。

## 选项

```
-D directory  
--directory=directory
```

要把输出写到哪个目录。

这个参数是必需的。

```
-E lsn  
--endpos=lsn
```

当接收到达指定的LSN时，自动停止复制并且以正常退出状态0退出。

如果有一个记录的LSN正好等于lsn，则该记录将会被处理。

`--if-not-exists`

当指定`--create-slot`并且具有指定名称的槽已经存在时不要抛出错误。

`-n`

`--no-loop`

不要在连接错误上循环。相反，碰到一个错误时立刻退出。

`--no-sync`

这个选项导致`lt_receivewal`不强制WAL数据被刷回磁盘。这样会更快，但是也意味着接下来的操作系统崩溃会让WAL段损坏。通常，这个选项对于测试有用，但不应该在对生产部署进行WAL归档时使用。

这个选项与`--synchronous`不兼容。

`-s interval`

`--status-interval=interval`

指定发送回服务器的状态包之间的秒数。这允许我们更容易地监控服务器的进度。一个零值完全禁用这种周期性的状态更新，不过当服务器需要时还是会有一个更新会被发送来避免超时导致的断开连接。默认值是 10 秒。

`-S slotname`

`--slot=slotname`

要求`lt_receivewal`使用一个已有的复制槽（见 Section 24.2.6）。在使用这个选项时，`lt_receivewal`将会报告给服务器一个刷写位置，指示每一个段是何时被同步到磁盘的，这样服务器可以在不需要该段时移除它。

当`lt_receivewal`的复制客户端在服务器上被配置为一个同步后备时，那么使用一个复制槽将会向服务器报告刷写位置，但只在一个WAL文件被关闭时报告。因此，该配置将导致主服务器上的事务等待很长的时间并且无法令人满意地工作。要让这种配置工作正确，还必须制定选项`--synchronous`（见下文）。

`--synchronous`

在WAL数据被收到后立即刷入到磁盘。还要在刷写后立即向服务器回送一个状态包（不考虑`--status-interval`）。

如果`lt_receivewal`的复制客户端在服务器上被配置为一个同步后备，应该指定这个选项来确保向服务器发送及时的反馈。

`-v`

`--verbose`

启用冗长模式。

`-Z level`

`--compress=level`

启用预写式日志上的gzip压缩，并且指定压缩级别（0到9，0是不压缩而9是最大压缩）。所有的文件名后都将被追加后缀.gz。

下列命令行选项控制数据库连接参数。

`-d connstr`

`--dbname=connstr`

指定用于连接到服务器的参数，作为连接字符串；这些将覆盖所有冲突的命令行选项。

为了和其他客户端应用一致，该选项被称为`--dbname`。但是因为`lt_receivewal`并不连接到集簇中的任何特定数据库，连接字符串中的数据库名将被忽略。

`-h host`  
`--host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。默认值取自`LTHOST`环境变量（如果设置），否则会尝试一个 Unix 域套接字连接。

`-p port`  
`--port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。默认用`LTPORT`环境变量中的值（如果设置），或者一个编译在程序中的默认值。

`-U username`  
`--username=username`

要作为哪个用户连接。

`-w`  
`--no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个`.pgpass`文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

`-W`  
`--password`

强制`lt_receivewal`在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，`lt_receivewal`将自动提示要求一个口令。但是，`lt_receivewal`将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用`-W`来避免额外的连接尝试。

为了控制物理复制槽，`lt_receivewal` 可以执行下列两种动作之一：

`--create-slot`

用`--slot`中指定的名称创建一个新的物理复制槽，然后退出。

`--drop-slot`

删除`--slot`中指定的复制槽，然后退出。

其他选项也可用：

`-V`  
`--version`

打印`lt_receivewal`版本并退出。

`-?`  
`--help`

显示有关`lt_receivewal`命令行参数的帮助并退出。

## 退出状态

在被`SIGINT`信号终止（没有正常的方式结束它。因此这不是一种错误）时，`lt_receivewal`将以状态0退出。对于致命错误或者其他信号，退出状态将不是零。



## 环境

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

环境变量PG\_COLOR规定在诊断消息中是否使用颜色。可能的值为always、auto、never。

## 注解

在使用lt\_receivewal替代 archive\_command作为主要的 WAL 备份方法时，强烈建议使用复制槽。否则，服务器可能会在预写式日志文件被备份好之前重用 或者移除它们，因为没有任何信息（不管是来自 archive\_command或是复制槽）能够指示 WAL 流已经被归档到什么程度。不过要注意，如果接收者没有持续地取走 WAL 数据，一个复制槽将会填满服务器的磁盘空间。

如果在源集簇上启用了组权限，lt\_receivewal将保留接收到的WAL文件上的组权限。

## 例子

要从位于mydbserver的服务器流式传送预写式日志并且将它存储在本地目录/usr/local/lightdb-x/archive:

```
$ lt_receivewal -h mydbserver -D /usr/local/lightdb-x/archive
```

## 参见

lt\_basebackup

---

# lt\_recvlogical

lt\_recvlogical — 控制 LightDB 逻辑解码流

## Synopsis

```
lt_recvlogical [option...]
```

## 描述

lt\_recvlogical控制逻辑解码复制槽以及来自这种复制槽的流数据。

它会创建一个复制模式的连接，因此它受到和lt\_receivewal 相同的约束，还有逻辑复制 (Chapter 45 的约束。

lt\_recvlogical与逻辑解码SQL接口的peek和get模式没有等效性。它咋接收到数据以及干净地退出时，它会惰性地发送数据的确认。为了检查一个槽上还未消费的待处理数据，可以使用pg\_logical\_slot\_peek\_changes。

## 选项

必须至少要指定下列选项之一来选择动作：

`--create-slot`

为--dbname指定的数据库用--slot 指定的名称创建一个新的逻辑复制槽，使用 --plugin指定的输出插件。

`--drop-slot`

删除名称由--slot指定的复制槽，然后退出。

`--start`

从--slot指定的逻辑复制槽开始进行流式传送更改，一直继续 到一个信号终止。如果服务器端关机或者断开连接导致更改流结束，会进入一个 循环一直重试，通过指定--no-loop可以防止这种情况下进入 循环重试。

流格式由槽创建时指定的输出插件决定。

连接必须是连接到用于创建该槽的同一个数据库上。

`--create-slot`和`--start`可以被一起指定。 `--drop-slot`不能和另一个动作组合在一起。

下面的命令行选项控制输出的位置和格式以及其他复制行为：

`-E lsn`

`--endpos=lsn`

在--start模式中，当接收过程到达指定的LSN时会自动地停止复制并且以正常的退出状态0退出。如果不处于--start模式时指定这个选项，则会发生错误。

如果有一个记录的LSN正好等于lsn，则该记录将被输出。

--endpos不会察觉到事务边界并且可能会在一个事务中间截断输出。任何部分输出的事务都不会被消费，并且在下一次从该槽中读取时将会重放该事务。单个的消息不会被截断。

-f filename  
--file=filename

把接收到并且解码好的事务数据写入到一个文件。使用-可以写到stdout。

-F interval\_seconds  
--fsync-interval=interval\_seconds

指定lt\_recvlogical发出 fsync() 调用确保输出文件被安全地刷到磁盘的频率。

服务器将会偶尔要求客户端执行一次刷写并且把刷写位置报告给服务器。这个设置可以在此之外更加频繁地执行刷写。

指定间隔为0会完全禁止发出fsync() 调用，但是仍会报告进度给服务器。在这种情况下，发生崩溃会导致数据丢失。

-I lsn  
--startpos=lsn

在--start模式中，从给定的 LSN 开始复制。这个参数的效果请见Chapter 4和Section 49. 中的文档。在其他模式中会忽略这个参数。

--if-not-exists

当指定--create-slot并且具有指定名称的槽已经存在时不要抛出错误。

-n  
--no-loop

当服务器连接丢失时，不要在循环中重试，直接退出。

-o name[=value]  
--option=name[=value]

如果指定了输出插件，把选项值value 传递给选项name。存在哪些选项以及它们的效果取决于使用的输出插件。

-P plugin  
--plugin=plugin

在创建一个槽时使用指定的逻辑解码输出插件。见Chapter 45 如果该槽已经存在，这个选项没有效果。

-s interval\_seconds  
--status-interval=interval\_seconds

这个选项和lt\_receivewal中的同名选项具有 相同的效果。请参考那里的描述。

-S slot\_name  
--slot=slot\_name

在--start模式中，使用名为slot\_name 的已有逻辑复制槽。在--create-slot模式中，使用这个名称 创建该槽。在--drop-slot模式中，删除这个名称指定的槽。

-v  
--verbose

开启详细输出模式。

下列命令行选项控制数据库连接参数。

-d database  
--dbname=database

要连接的数据库。这个选项的详细含义请见动作的描述。dbname 可以是连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。默认为用户名。

-h hostname-or-ip  
--host=hostname-or-ip

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从 LTHOST环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

-p port  
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在LTPORT环境变量中（如果被设置），否则使用编译在程序中的默认值。

-U user  
--username=user

要作为哪个用户连接。默认是用当前操作系统用户名。

-w  
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W  
--password

强制lt\_dump在连接到一个数据库之前提示要求一个口令。

这个选项不是必须的，因为如果服务器要求口令认证，lt\_dump将自动提示要求一个口令。但是，lt\_dump将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入 -W来避免额外的连接尝试。

还有下列附加选项可用：

-V  
--version

打印lt\_recvlogical的版本并且退出。

-?  
--help

显示关于lt\_recvlogical命令行参数的帮助，并且退出。

## 环境

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

环境变量PG\_COLOR规定在诊断消息中是否使用颜色。可能的值为always、auto、never。

## 注解

如果在源服务器上启用了组权限，lt\_recvlogical将会在接收到的WAL文件上保留组权限。

## 例子

一个例子请见Section 45.1

## 另见

lt\_receivewal

---

# lt\_restore

lt\_restore — 从一个由lt\_dump创建的归档文件恢复一个LightDB数据库

## Synopsis

```
lt_restore [connection-option...] [option...] [filename]
```

## 描述

lt\_restore是一个用来从lt\_dump创建的非文本格式归档恢复LightDB数据库的工具。它将发出必要的命令把该数据库重建成它被保存时的状态。这些归档文件还允许lt\_restore选择恢复哪些内容或者在恢复前对恢复项重排序。这些归档文件被设计为可以在不同的架构之间迁移。

lt\_restore可以在两种模式下操作。如果指定了一个数据库名称，lt\_restore会连接那个数据库并且把归档内容直接恢复到该数据库中。否则，会创建一个脚本，其中包含着重建该数据库所必要的 SQL 命令，它会被写入到一个文件或者标准输出。这个脚本输出等效于lt\_dump的纯文本输出格式。因此，一些控制输出的选项与lt\_dump的选项类似。

显然，lt\_restore无法恢复不在归档文件中的信息。例如，如果归档使用“以INSERT命令存储数据”选项创建，lt\_restore将无法使用COPY语句装载数据。

## 选项

lt\_restore接受下列命令行参数。

filename

指定要被恢复的归档文件（对于一个目录格式的归档则是目录）的位置。如果没有指定，则使用标准输入。

-a  
--data-only

只恢复数据，不恢复模式（数据定义）。如果在归档中存在，表数据、大对象和序列值会被恢复。

这个选项类似于指定--section=data，但是由于历史原因两者不完全相同。

-c  
--clean

在重新创建数据库对象之前清除（丢弃）它们（除非使用了--if-exists，如果有对象在目标数据库中不存在，这可能会生成一些无害的错误消息）。

-K  
--recreate-schema

输出命令在输出命令以创建它们之前直接删除模式（使用级联模式），比正常模式更快。（除非使用了--if-exists，如果有对象在目标数据库中不存在，这可能会生成一些无害的错误消息）。

输出存档文件时（非文本）会忽略此选项。对于存档格式，您可以在调用 lt\_restore 时指定选项。

---

-C

--create

在恢复一个数据库之前先创建它。如果还指定了--clean，在连接到目标数据库之前丢弃并且重建它。

如果使用--create，lt\_restore还会恢复数据库的注释（如果有）以及与其相关的配置变量设置，也就是任何提到过这个数据库的ALTER DATABASE ... SET ...和ALTER ROLE ... IN DATABASE ... SET ...命令。不管是否指定--no-acl，数据库本身的访问特权都会被恢复。

在使用这个选项时，-d提到的数据库只被用于发出初始的DROP DATABASE和CREATE DATABASE命令。所有要恢复到该数据库名中的数据都出现在归档中。

-d dbname

--dbname=dbname

连接到数据库dbname并且直接恢复到该数据库中。dbname 可以是 连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

-e

--exit-on-error

在发送 SQL 命令到该数据库期间如果碰到一个错误就退出。默认行为是继续并且在恢复结束时显示一个错误计数。

-f filename

--file=filename

为生成的脚本指定输出文件，或在与-l选项一起使用时为列表指定输出文件。为 stdout 用 -。

-F format

--format=format

指定归档的格式。并不一定要指定该格式，因为lt\_restore将会自动决定格式。如果指定，可以是下列之一：

c

custom

归档是lt\_dump的自定义格式。

d

directory

归档是一个目录归档。

t

tar

归档是一个tar归档。

-I index

--index=index

只恢复提及的索引的定义。可以通过写多个-I开关指定多个索引。

-j number-of-jobs

--jobs=number-of-jobs

使用并发任务运行lt\_restore中最耗时的步骤 — 载入数据、创建索引或者创建约束；同时，最多使用number-of-jobs个并发会话。对于一个运行在多处理器机器上的服

务器，这个选项能够大幅度减少恢复一个大型数据库的时间。当发出脚本而不是直接连接到数据库服务器时，将忽略此选项。

每一个任务是一个进程或者一个线程，这取决于操作系统，它们都使用一个独立的服务器连接。

这个选项的最佳值取决于服务器、客户端以及网络的硬件设置。因素包括 CPU 的核心数和磁盘设置。一个好的建议是服务器上 CPU 的核心数，但是更大的值在很多情况下也能导致更快的恢复时间。当然，过高的值会由于超负荷反而导致性能降低。

这个选项只支持自定义和目录归档格式。输入必须是一个常规文件或目录（例如，不能是一个管道或者标准输出）。还有，多任务不能和选项`--single-transaction`一起用。

`-l`  
`--list`

列出归档的内容的表格。这个操作的输出能被用作`-L`选项的输入。注意如果把`-n`或`-t`这样的过滤开关与`-l`一起使用，它们将会限制列出的项。

`-L list-file`  
`--use-list=list-file`

只恢复在`list-file`中列出的归档元素，并且按照它们出现在该文件中的顺序进行恢复。注意如果把`-n`或`-t`这样的过滤开关与`-L`一起使用，它们将会进一步限制要恢复的项。

`list-file`通常是编辑一个`-l`操作的输出来创建。行可以被移动或者移除，并且也可以通过在行首放一个（`;`）将其注释掉。例子见下文。

`-n schema`  
`--schema=schema`

只恢复在被提及的模式中的对象。可以用多个`-n`开关来指定多个模式。这可以与`-t`选项组合在一起只恢复一个指定的表。

`-N schema`  
`--exclude-schema=schema`

不恢复所提及方案中的对象。可以用多个`-N`开关指定多个要被排除的方案。

如果对同一个方案名称同时给出了`-n`和`-N`，则`-N`会胜出并且该方案会被排除。

`-O`  
`--no-owner`

不要输出将对象的所有权设置为与原始数据库匹配的命令。默认情况下，`lt_restore`会发出`ALTER OWNER`或者`SET SESSION AUTHORIZATION`语句来设置已创建的模式对象的所有权。除非到该数据库的初始连接是一个超级用户（或者拥有脚本中所有对象的同一个用户）建立的，这些语句将会失败。通过`-O`，任何用户名都可以被用于初始连接，并且这个用户将会拥有所有被创建的对象。

`-P function-name(argtype [, ...])`  
`--function=function-name(argtype [, ...])`

只恢复被提及的函数。要小心地拼写函数的名称和参数使它们正好就是出现在转储文件的内容表中的名称和参数。可以使用多个`-P`开关指定多个函数。

`-R`  
`--no-reconnect`

这个选项已被废弃，但是出于向后兼容性的目的，系统仍然还接受它。



`-s`  
`--schema-only`

只恢复归档中的模式（数据定义）不恢复数据。

这个选项是`--data-only`的逆选项。它与指定`--section=pre-data` `--section=post-data`相似，但是由于历史原因并不完全相同。

（不要把这个选项和`--schema`选项弄混，后者把词“schema”用于一种不同的含义）。

`-S username`  
`--superuser=username`

指定在禁用触发器时要用的超级用户名。只有使用`--disable-triggers`时这个选项才相关。

`-t table`  
`--table=table`

只恢复所提及的表的定义和数据。出于这个目的，“table”包括视图、物化视图、序列和外部表。可以写上多个`-t`开关可以选择多个表。这个选项可以和`-n`选项结合在一起指定一个特定模式中的表。

#### Note

在指定`-t`时，`lt_restore`不会尝试恢复所选表可能依赖的任何其他数据库对象。因此，无法确保能成功地把一个特定表恢复到一个干净的数据库中。

#### Note

这个标志的行为和`lt_dump`的`-t`标志不一样。在`lt_restore`中当前没有任何通配符匹配的规定，也不能在其`-t`选项中包括模式的名称。而且，虽然`lt_dump`的`-t`标志也会转储选中表的附属对象（例如索引），但是`lt_restore`的`-t`标志不包括这些附属对象。

#### Note

在 9.6 版本之前的LightDB 9.6 中，这个标志只匹配表，而并不匹配其他类型的关系。

`-T trigger`  
`--trigger=trigger`

只恢复所提及的触发器。可以用多个`-T`开关指定多个触发器。

`-v`  
`--verbose`

指定冗长模式。 这将导致`lt_restore`输出详细的对象注释和启动/停止时间到输出文件，并将推送消息输出到标准错误。 重复该选项会造成附加的调试级消息出现在标准报错上。

-V  
--version

打印该lt\_restore的版本并退出。

-x  
--no-privileges  
--no-acl

阻止恢复访问特权（授予/收回命令）。

-l  
--single-transaction

将恢复作为单一事务执行（即把发出的命令包裹在BEGIN/COMMIT中）。这可以确保要么所有命令完全成功，要么任何改变都不被应用。这个选项隐含了--exit-on-error。

--disable-triggers

只有在执行一个只恢复数据的恢复时，这个选项才相关。它指示lt\_restore在装载数据时执行命令临时禁用目标表上的触发器。如果你在表上有参照完整性检查或者其他触发器并且你不希望在数据载入期间调用它们时，请使用这个选项。

目前，为--disable-triggers发出的命令必须以超级用户身份完成。因此你还应该用-S指定一个超级用户名，或者更好的方法是以一个LightDB超级用户运行lt\_restore。

--enable-row-security

只有在恢复具有行安全性的表的内容时，这个选项才相关。默认情况下，lt\_restore将把row\_security设置为 off 来确保所有数据都被恢复到表中。如果用户不拥有足够绕过行安全性的特权，那么会抛出一个错误。这个参数指示lt\_restore把row\_security设置为on允许用户尝试恢复启用了行安全性的表的内容。如果用户没有从转储向表中插入行的权限，这仍将失败。

注意当前这个选项还要求转储处于INSERT格式，因为COPY FROM不支持行安全性。

--if-exists

使用条件命令（即增加一个IF EXISTS子句）删除数据库对象。只有指定了--clean时，这个选项才有效。

--no-comments

即便归档中包含注释也不输出恢复注释的命令。

--no-data-for-failed-tables

默认情况下，即便表的创建命令失败（例如因为表已经存在），表数据也会被恢复。通过这个选项，对这类表的数据会被跳过。如果目标数据库已经包含了想要的表内容，这种行为又很有用。例如，LightDB扩展（如PostGIS）的辅助表可能已经被载入到目标数据库中，指定这个选项就能阻止把重复的或者废弃的数据载入到这些表中。

只有当直接恢复到一个数据库中时这个选项才有效，在产生 SQL脚本输出时这个选项不会产生效果。

--no-publications

即便归档中包含publication也不输出恢复publication的命令。

--no-security-labels

不要输出恢复安全标签的命令，即使归档中包含安全标签。

`--no-subscriptions`

即便归档中包含subscription也不输出恢复subscription的命令。

`--no-tablespaces`

不输出命令选择表空间。通过这个选项，所有的对象都会被创建在恢复时的默认表空间中。

`--ignore-restore-error`

lt\_restore 如果成功恢复则返回 0，如果在发送 SQL 命令到数据库期间碰到一个错误，则返回 1。通过这个选项，可以让 lt\_restore 始终返回 0。

`--section=sectionname`

只恢复提及的小节。小节的名称可以是pre-data、data或者post-data。可以把这个选项指定多次来选择多个小节。默认值是恢复所有小节。

数据小节包含实际的表数据以及大对象定义。Post-data 项由索引定义、触发器、规则和除已验证的检查约束之外的约束构成。Pre-data 项由所有其他数据定义项构成。

`--strict-names`

要求每一个模式 (`-n/--schema`) 以及表 (`-t/--table`) 限定词匹配备份文件中至少一个模式/表。

`--use-set-session-authorization`

输出 SQL 标准的SET SESSION AUTHORIZATION命令取代ALTER OWNER命令来决定对象拥有权。这会让转储更加兼容标准，但是依赖于转储中对象的历史，可能无法正确恢复。

`-?`

`--help`

显示有关lt\_restore命令行参数的帮助，并且退出。

lt\_restore也接受下列用于连接参数的命令行参数：

`-h host`

`--host=host`

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从LTHOST环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

`-p port`

`--port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在LTPORT环境变量中（如果被设置），否则使用编译在程序中的默认值。

`-U username`

`--username=username`

要作为哪个用户连接。

`-w`

`--no-password`

不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

`-W`  
`--password`

强制lt\_restore在连接到一个数据库之前提示要求一个口令。

这个选项不是必须的，因为如果服务器要求口令认证，lt\_restore将自动提示要求一个口令。但是，lt\_restore将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入-W来避免额外的连接尝试。

`--role=rolename`

指定一个用来创建该转储的角色名。这个选项导致lt\_restore在连接到数据库后发出一个SET ROLE rolename命令。当已认证用户（由-U指定）缺少lt\_restore所需的特权但是能够切换到一个具有所需权利的角色时，这个选项很有用。一些安装有针对直接作为超级用户登录的策略，使用这个选项可以让转储在不违反该策略的前提下完成。

## 环境

LTHOST  
 PGOPTIONS  
 LTPORT  
 LTUSER

默认连接参数

PG\_COLOR

规定在诊断消息中是否使用颜色。可能的值为always、auto、never。 never.

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 诊断

当使用-d选项指定一个直接数据库连接时，lt\_restore在内部执行SELECT语句。如果你运行lt\_restore时出现问题，确定你能够从正在使用的数据库中选择信息，例如ltsql。此外，libpq前端-后端库所使用的任何默认连接设置和环境变量都将适用。

## 注解

如果你的数据库集簇对于template1数据库有任何本地添加，要注意将lt\_restore的输出载入到一个真正的空数据库。否则你很可能由于以增加对象的重复定义而得到错误。要创建一个不带任何本地添加的空数据库，从template0而不是template1复制它，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

下面将详细介绍lt\_restore的局限性。

- 在恢复数据到一个已经存在的表中并且使用了选项--disable-triggers时，lt\_restore会在插入数据之前发出命令禁用用户表上的触发器，然后在完成数据插入后重新启用它们。如果恢复在中途停止，可能会让系统目录处于错误的状态。
- lt\_restore不能有选择地恢复大对象，例如只恢复特定表的大对象。如果一个归档包含大对象，那么所有的大对象都会被恢复，如果通过-L、-t或者其他选项进行了排除，它们一个也不会被恢复。

lt\_dump的局限性详见lt\_dump文档。

一旦完成恢复，应该在每一个被恢复的表上运行ANALYZE，这样优化器能得到有用的统计信息。更多信息请见Section 22.1.4和Section 22.1.6

## 示例

假设我们已经以自定义格式转储了一个叫做mydb的数据库：

```
$ lt_dump -Fc mydb > db.dump
```

要删除该数据库并且从转储中重新创建它：

```
$ dropdb mydb
$ lt_restore -C -d postgres db.dump
```

-d开关中提到的数据库可以是任何已经存在于集簇中的数据库，lt\_restore只会用它来为mydb发出CREATE DATABASE命令。通过-C，数据总是会被恢复到出现在归档文件的数据库名中。

要把转储重新载入到一个名为newdb的新数据库中：

```
$ createdb -T template0 newdb
$ lt_restore -d newdb db.dump
```

注意我们不使用-C，而是直接连接到要恢复到其中的数据库。还要注意我们是从template0而不是templatel创建了该数据库，以保证它最初是空的。

要对数据库项重排序，首先需要转储归档的表内容：

```
$ lt_restore -l db.dump > db.list
```

列表文件由一个头部和一些行组成，这些行每一个都用于一个项，例如：

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;   dbname: DBDEMOS
;   TOC Entries: 81
;   Compression: 9
;   Dump Version: 1.10-0
;   Format: CUSTOM
;   Integer: 4 bytes
;   Offset: 8 bytes
;   Dumped from database version: 8.3.5
;   Dumped by lt_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

分号表示开始一段注释，行首的数字表明了分配给每个项的内部归档 ID。

文件中的行可以被注释掉、删除以及重排序。例如：

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

把这样一个文件作为lt\_restore的输入将会只恢复项 10 和 6，并且先恢复 10 再恢复 6。

```
$ lt_restore -L db.list db.dump
```

## 另见

lt\_dump, lt\_dumpall, ltsql

---

# lt\_verifybackup

lt\_verifybackup — 验证LightDB集群的基础备份的完整性

## Synopsis

lt\_verifybackup [option...]

## 简介

lt\_verifybackup用于根据备份时服务器生成的backup\_manifest检查使用lt\_basebackup进行的数据库群集备份的完整性。备份必须以“普通”格式存储；“tar”格式的备份可以在解压缩后进行检查。

需要注意的是，由lt\_verifybackup执行的验证不包括也不可能包括运行中的服务器在尝试使用备份时执行的所有检查。即使使用此工具，也应执行测试还原，并验证生成的数据库是否按预期工作，以及它们是否包含正确的数据。但是，lt\_verifybackup可以检测到由于存储问题或用户错误而经常出现的许多问题。

备份验证分四个阶段进行。首先，lt\_verifybackup读取backup\_manifest文件。如果该文件不存在、无法读取、格式不正确或无法根据其内部校验和进行验证，lt\_verifybackup 将以致命错误终止。

其次，lt\_verifybackup 将尝试验证当前存储在磁盘上的数据文件是否与服务器打算发送的数据文件完全相同，下面将介绍一些例外情况。除了少数例外，额外和丢失的文件将被检测到。此步骤将忽略 lightdb.auto.conf、standby.signal 和 recovery.signal 的存在与否或对其的任何修改，因为预计这些文件可能是在备份过程中创建或修改的。它也不会抱怨目标目录中的 backup\_manifest 文件或 lt\_wal 中的任何内容，即使这些文件不会列在备份清单中。只检查文件；不验证目录的存在与否，除非间接验证：如果目录丢失，则它应该包含的任何文件也必然会丢失。

接下来，lt\_verifybackup将对所有文件进行校验和计算，将校验和与清单中的值进行比较，并对计算出的校验和与清单中存储的校验和不匹配的任何文件发出错误。对于在上一步中产生错误的任何文件，不执行此步骤，因为已知这些文件存在问题。在上一步中被忽略的文件在此步骤中也被忽略。

最后，lt\_verifybackup 将使用清单来验证恢复备份所需的预写式日志记录是否存在，并且它们可以被读取和解析。 backup\_manifest 包含有关需要哪些预写式日志记录的信息，并且 lt\_verifybackup 将使用该信息来调用 lt\_waldump 来解析这些预写式日志记录。 --quiet 标志将被使用，因此 lt\_waldump 只会报告错误，而不会产生任何其他输出。虽然这种级别的验证足以检测明显的问题，例如丢失的文件或内部校验和不匹配的问题，但它们还不足以检测尝试恢复时可能出现的所有问题。例如，此方法无法检测到产生具有正确校验和但指定无意义操作的预写式日志记录的服务器错误。

请注意，如果存在不需要恢复备份的额外 WAL 文件，则此工具不会检查它们，尽管可以为此使用单独的 lt\_waldump 调用。另请注意，WAL 验证是特定于版本的：您必须使用 lt\_verifybackup 的版本，因此是 lt\_waldump 的版本，它与正在检查的备份有关。相比之下，数据文件完整性检查应适用于生成 backup\_manifest 文件的任何版本的服务器。

## 选项

lt\_verifybackup 接受以下命令行参数：

`-e`  
`--exit-on-error`

检测到备份问题后立即退出。如果没有指定这个选项，`lt_verifybackup` 将在检测到问题后继续检查备份，并将检测到的所有问题报告为错误。

`-i path`  
`--ignore=path`

在将备份中实际存在的数据文件列表与 `backup_manifest` 文件中列出的数据文件列表进行比较时，忽略指定的文件或目录，该文件或目录应表示为相对路径名。如果指定了目录，则此选项会影响以该位置为根的整个子树。如果相对路径名与指定的路径名匹配，有关额外文件、丢失文件、文件大小差异或校验和不匹配的投诉将被抑制。可以多次指定此选项。

`-m path`  
`--manifest-path=path`

使用指定路径的清单文件，而不是位于备份目录根目录中的清单文件。

`-n`  
`--no-parse-wal`

不要试图解析从该备份恢复所需的预写式日志数据。

`-q`  
`--quiet`

成功验证备份后不要打印任何内容。

`-s`  
`--skip-checksums`

不要验证数据文件校验和。但仍检查是否存在文件以及这些文件的大小。这样将会快得多，因为文件本身不需要读取。

`-w path`  
`--wal-directory=path`

尝试解析存储在指定目录中的 WAL 文件，而不是 `lt_wal`。如果备份存储在与 WAL 存档不同的位置，则这可能很有用。

其他选项也可用：

`-V`  
`--version`

打印 `lt_verifybackup` 版本并退出。

`-?`  
`--help`

显示有关 `lt_verifybackup` 命令行参数的帮助，然后退出。

## 示例

要在 `mydbserver` 上创建服务器的基本备份并验证备份的完整性：

```
$ lt_basebackup -h mydbserver -D /usr/local/lightdb-x/data
$ lt_verifybackup /usr/local/lightdb-x/data
```



要在 `mydbserver` 上创建服务器的基本备份，请将清单移动到备份目录之外的某个位置，并验证备份：

```
$ lt_basebackup -h mydbserver -D /usr/local/lightdb-x/backup1234
$ mv /usr/local/lightdb-x/backup1234/backup_manifest /my/secure/location/
  backup_manifest.1234
$ lt_verifybackup -m /my/secure/location/backup_manifest.1234 /usr/local/
  lightdb-x/backup1234
```

要在忽略手动添加到备份目录的文件的同时验证备份，并跳过校验和验证：

```
$ lt_basebackup -h mydbserver -D /usr/local/lightdb-x/data
$ edit /usr/local/lightdb-x/data/note.to.self
$ lt_verifybackup --ignore=note.to.self --skip-checksums /usr/local/lightdb-x/
  data
```

## 另见

`lt_basebackup`

---

# ltpbench

ltpbench — 在LightDB上运行一个基准测试

## Synopsis

```
ltpbench -i [option...] [dbname]
```

```
ltpbench [option...] [dbname]
```

## 描述

ltpbench是一种在LightDB上运行基准测试的简单程序。它可能在并发的数据库会话中一遍一遍地运行相同序列的 SQL 命令，并且计算平均事务率（每秒的事务数）。默认情况下，ltpbench会测试一种基于 TPC-B 但是要更宽松的场景，其中在每个事务中涉及五个SELECT、UPDATE以及INSERT命令。但是，通过编写自己的事务脚本文件很容易用来测试其他情况。

ltpbench的典型输出像这样：

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
latency average = 11.013 ms
latency stddev = 7.351 ms
initial connection time = 45.758 ms
tps = 896.967014 (without initial connection time)
```

前六行报告一些最重要的参数设置。接下来的行报告完成的事务数以及预期的事务数（后者就是客户端数量与每个客户端事务数的乘积），除非运行在完成之前失败，这些值应该是相等的（在-T模式中，只有实际的事务数会被打印出来）。最后一行报告每秒的事务数。

默认的类型 TPC-B 事务测试要求预先设置好特定的表。可以使用-i（初始化）选项调用ltpbench来创建并且填充这些表（当你在测试一个自定义脚本时，你不需要这一步，但是需要按你自己的测试需要做一些设置工作）。初始化类似这样：

```
ltpbench -i [ other-options ] dbname
```

其中dbname是要在其中进行测试的预先创建好的数据库的名称（你可能还需要-h、-p或-U选项来指定如何连接到数据库服务器）。

### Caution

ltpbench -i会创建四个表ltpbench\_accounts、ltpbench\_branches、ltpbench\_history以及ltpbench\_tellers，如果同名表已经存在会被先删除。如果你已经有同名表，一定要注意要使用另一个数据库！

在默认的情况下“比例因子”为 1，这些表初始包含的行数为：

table	# of rows
ltbench_branches	1
ltbench_tellers	10
ltbench_accounts	100000
ltbench_history	0

你可以使用-s（比例因子）选项增加行的数量。-F（填充因子）选项也可以在这里使用。

一旦你完成了必要的设置，你就可以用不包括-i的命令运行基准，也就是：

```
ltbench [ options ] dbname
```

在近乎所有的情况中，你将需要一些选项来做一次有用的测试。最重要的选项是-c（客户端数量）、-t（事务数量）、-T（时间限制）以及-f（指定一个自定义脚本文件）。完整的列表见下文。

## 选项

下面分成三个部分。数据库初始化期间使用的选项和运行基准时会使用不同的选项，但也有些选项在两种情况下都使用。

### 初始化选项

ltbench接受下列命令行初始化参数：

dbname

指定要测试的数据库的名称。如果这个没有被指定，将使用环境变量LTDATABASE。如果那个也没有设定，将使用指定要连接的用户名称。

-i

--initialize

要求调用初始化模式。

-I init\_steps

--init-steps=init\_steps

只执行选出的一组普通初始化步骤。init\_steps指定要被执行的初始化步骤，每一个步骤使用一个字符代表。每一个步骤都以指定的顺序被调用。默认是dtgvp。可用的步骤是：

d（删除）

删除任何已有的ltbench表。

t（创建表）

创建标准ltbench场景使用的表，即ltbench\_accounts、ltbench\_branches、ltbench\_history以及ltbench\_tellers。

g或G（生成数据、客户端或服务端）

生成数据并且装入到标准的表中，替换掉已经存在的任何数据。

使用 g（客户端数据生成），数据在 ltbench 客户端生成，然后发送到服务器。这通过 COPY 广泛使用客户端/服务器带宽。使用 g 会导致日志记录每 100,000 行打印一条消息，同时为 ltbench\_accounts 表生成数据。

使用G（服务器端数据生成），仅从ltbench客户端发送少量查询，然后在服务器中实际生成数据。此变体不需要大量带宽，但服务器将完成更多工作。使用G会导致日志记录在生成数据时不打印任何进度消息。

默认的初始化行为使用客户端数据生成（相当于g）。

v（清理）

在标准的表上调用VACUUM。

p（创建主键）

在标准的表上创建主键索引。

f（创建外键）

在标准的表之间创建外键约束（注意这一步默认不会被执行）。

-F fillfactor  
--fillfactor=fillfactor

用给定的填充因子创建表ltbench\_accounts、ltbench\_tellers以及ltbench\_branches。默认是100。

-n  
--no-vacuum

在初始化期间不执行清理（这个选项会抑制v初始化步骤，即便在-I中指定了该步骤）。

-q  
--quiet

把记录切换到安静模式，只是每 5 秒产生一个进度消息。默认的记录会每 100,000 行打印一个消息，这经常会在每秒钟输出很多行（特别是在好的硬件上）。

如果在 -I 中指定了 G，则此设置无效。

-s scale\_factor  
--scale=scale\_factor

将生成的行数乘以比例因子。例如，-s 100将在ltbench\_accounts表中创建 10,000,000 行。默认为 1。当比例为 20,000 或更高时，用来保存账号标识符的列（aid列）将切换到使用更大的整数（bigint），这样才能足以保存账号标识符。

--foreign-keys

在标准的表之间创建外键约束（如果f在初始化步骤序列中不存在，这个选项会把它加入）。

--index-tablespace=index\_tablespace

在指定的表空间而不是默认表空间中创建索引。

--partition-method=NAME

使用 NAME 方法创建一个分区的 ltbench\_accounts 表。预期值为 range 或 hash。此选项要求 --partitions 设置为非零。如果未指定，默认值为 range。

--partitions=NUM

创建一个分区 ltbench\_accounts 表，其中 NUM 分区的大小与按比例缩放的帐户数几乎相等。默认为 0，表示没有分区。

---

`--tablespace=tablespace`

在指定的表空间而不是默认表空间中创建表。

`--unlogged-tables`

把所有的表创建为非日志记录表而不是永久表。

## 基准选项

ltbench接受下列命令行基准参数：

`-b scriptname[@weight]`

`--builtin=scriptname[@weight]`

把指定的内建脚本加入到要被执行的脚本列表中。 可用的内建脚本有：tpcb-like、simple-update和select-only。 这里也接受内建名称无歧义的前缀缩写。 如果用上特殊的名字list，将会显示内建脚本的列表并且立刻退出。

可选的，在@后面写一个整数权重以调整选择这个脚本而不是其它的概率。 默认的权重为1。 详情如下。

`-c clients`

`--client=clients`

模拟的客户端数量，也就是并发数据库会话数量。默认为 1。

`-C`

`--connect`

为每一个事务建立一个新连接，而不是只为每个客户端会话建立一个连接。这对于度量连接开销有用。

`-d`

`--debug`

打印调试输出。

`-D varname=value`

`--define=varname=value`

定义一个由自定义脚本（见下文）使用的变量。允许多个-D选项。

`-f filename[@weight]`

`--file=filename[@weight]`

把一个从filename读到的事务脚本加入到被执行的脚本列表中。

可选的，在@后写一个整数权重以调整选择这个脚本而非其他的概率。 默认权重为1。（要使用包含@字符的脚本文件名，附加权重在后面以避免歧义，例如filen@me@1。） 详情如下。

`-j threads`

`--jobs=threads`

ltbench中的工作者线程数量。在多 CPU 机器上使用多于一个线程会有用。客户端会尽可能均匀地分布到可用的线程上。默认为 1。

`-l`

`--log`

把与每一个事务相关的信息写到一个日志文件中。详见下文。

-L limit  
--latency-limit=limit

对持续超过limit毫秒的事务进行独立的计数和报告，这些事务被认为是迟到 (late) 了的事务。

在使用限流措施时 (--rate=...)，滞后于计划超过 limit毫秒并且因此没有希望满足延迟限制的事务根本不会被发送给服务器。这些事务被认为是被跳过 (skipped) 的事务，它们会被单独计数并且报告。

-M querymode  
--protocol=querymode

要用来提交查询到服务器的协议：

- simple: 使用简单查询协议。
- extended使用扩展查询协议。
- prepared: 使用带预备语句的扩展查询语句。

在prepared模式中，ltbench重用从第二次查询迭代开始的语法分析结果，因此ltbench运行速度比其他模式快。

默认是简单查询协议（详见Chapter 49）。

-n  
--no-vacuum

在运行测试前不进行清理。如果你在运行一个不包括标准的表ltbench\_accounts、ltbench\_branches、ltbench\_history和 ltbench\_tellers的自定义测试场景时，这个选项是必需的。

-N  
--skip-some-updates

运行内建的简单更新脚本。这是-b simple-update的简写。

-P sec  
--progress=sec

每sec秒显示进度报告。该报告包括运行了多长时间、从上次报告以来的 tps 以及从上次报告以来事务延迟的平均值和标准偏差。如果低于限流值 (-R)，延迟会相对于事务预定的开始时间（而不是实际的事务开始时间）计算，因此其中也包括了平均调度延迟时间。

-r  
--report-latencies

在基准结束后，报告平均的每个命令的每语句等待时间（从客户端的角度来说是执行时间）。详见下文。

-R rate  
--rate=rate

按照指定的速率执行事务而不是尽可能快地执行（默认行为）。该速率以 tps（每秒事务数）形式给定。如果目标速率高于最大可能速率，则 该速率限制不会影响结果。

该速率的目标是按照一条泊松分布的调度时间线开始事务。期望的开始 时间表会基于客户端第一次开始的时间（而不是上一个事务结束的时间）前移。这种方法意味着当事务超过它们的原定结束时间时，更迟的 那些有机会再次追赶上来。

当限流措施被激活时，运行结束时报告的事务延迟是从预订的开始时间计算而来的，因此它包括每一个事务不得不等待前一个事务结束所花的时间。该等待时间被称作调度延迟时间，并且它的平均值和最大值也会被单独报告。关于实际事务开始时间的事务延迟（即在数据库中执行事务所花的时间）可以用报告的延迟减去调度延迟时间计算得到。

如果把`--latency-limit`和`--rate`一起使用，当一个事务在前一个事务结束时已经超过了延迟限制时，它可能会滞后非常多，因为延迟是从计划的开始时间计算得来。这类事务不会被发送给服务器，而是一起被跳过并且被单独计数。

一个高的调度延迟时间表示系统无法用选定的客户端和线程数按照指定的速率处理事务。当平均的事务执行时间超过每个事务之间的调度间隔时，每一个后续事务将会落后更多，并且随着测试运行时间越长，调度延迟时间将持续增加。发生这种情况时，你将不得不降低指定的事务速率。

`-s scale_factor`  
`--scale=scale_factor`

在ltbench的输出中报告指定的比例因子。对于内建测试，这并非必需；正确的比例因子将通过对ltbench\_branches表中的行计数来检测。不过，当只测试自定义基准（`-f`选项）时，比例因子将被报告为1（除非使用了这个选项）。

`-S`  
`--select-only`

执行内建的只有选择的脚本。是`-b select-only`简写形式。

`-t transactions`  
`--transactions=transactions`

每个客户端运行的事务数量。默认为10。

`-T seconds`  
`--time=seconds`

运行测试这么多秒，而不是为每个客户端运行固定数量的事务。`-t`和`-T`是互斥的。

`-v`  
`--vacuum-all`

在运行测试前清理所有四个标准的表。在没有用`-n`以及`-v`时，ltbench将清理ltbench\_tellers和ltbench\_branches表，并且截断ltbench\_history。

`--aggregate-interval=seconds`

聚集区间的长度（单位是秒）。仅可以与`-l`选项一起使用。通过这个选项，日志会包含针对每个区间的概要数据，如下文所述。

`--log-prefix=prefix`

设置`--log`创建的日志文件的文件名前缀。默认是ltbench\_log。

`--progress-timestamp`

当显示进度（选项`-P`）时，使用一个时间戳（Unix时间）取代从运行开始的秒数。单位是秒，在小数点后是毫秒精度。这可以有助于比较多种工具生成的日志。

`--random-seed=seed`

设置随机数生成器种子。为系统的随机数生成器提供种子，然后随机数生成器会产生一个初始生成器状态序列，每一个线程一个状态。seed的值可以是：`time`（默认值，种子基于当前时间）、`rand`（使用一种强随机源，如果没有可用的源则失败）或者一个无符

号十进制整数值。一个ltbench脚本中会显式（random...函数）地或者隐式地（如--rate使用随机数生成器调度事务）调用随机数生成器。在被明确设置时，用作种子的值会显示在终端上。还可以通过环境变量PGBENCH\_RANDOM\_SEED提供用于seed的值。为了确保所提供的种子影响所有可能的使用，把这个选项放在第一位或者使用环境变量。

明确地设置种子允许准确地再生一个ltbench运行，对随机数而言。因为随机状态是针对每个线程管理，这意味着如果每一个线程有一个客户端并且没有外部或者数据依赖，则对于一个相同的调用就会有完全相同的ltbench运行。从一种统计的角度来看，再生运行不是什么好主意，因为它能隐藏性能可变性或者不正当地改进性能，即通过命中前一次运行的相同页面来改进性能。不过，它也可以对调试起到很大帮助作用，例如重新运行一种导致错误的棘手用例。请善用。

--sampling-rate=rate

采样率，在写入数据到日志时被用来减少日志产生的数量。如果给出这个选项，只有指定比例的事务被记录。1.0 表示所有事务都将被记录，0.05 表示只有 5% 的事务会被记录。

在处理日志文件时，记得要考虑这个采样率。例如，当计算TPS值时，你需要相应地乘以这个数字（例如，采样率是 0.01，你将只能得到实际TPS的 1/100）。

--show-script=scriptname

在 stderr 上显示内置脚本 scriptname 的实际代码，并立即退出。

## 普通选项

ltbench 还接受以下用于连接参数的常见命令行参数：

-h hostname

--host=hostname

数据库服务器的主机名

-p port

--port=port

数据库服务器的端口号

-U login

--username=login

要作为哪个用户连接

-V

--version

打印ltbench版本并退出。

-?

--help

显示有关ltbench命令行参数的信息，并且退出。

## Exit Status

成功运行将以状态0退出。退出状态为1表示静态问题，如无效的命令行选项。运行过程中的错误，例如数据库错误或脚本中的问题将导致退出状态为2。在后一种情况下，ltbench将打印部分结果。



## 环境

LTDATABASE  
LTHOST  
LTPORT  
LTUSER

默认连接参数。

此实用程序与大多数其他 LightDB 实用程序一样，使用 libpq 支持的环境变量（请参阅 Section 32.14）。

环境变量 LT\_COLOR 指定是否在诊断消息中使用颜色。可能的值是always、auto 和 never。

## 注解

### 在ltbench中实际执行的“事务”是什么？

ltbench执行从指定列表中随机选中的测试脚本。这个脚本包括带有-b指定的内建脚本和带有-f指定的用户提供的自定义脚本。每一个脚本可以在其后用@指定一个相对权重，这样可以更改该脚本的选择概率。默认权重是1。权重为0的脚本会被忽略。

默认的内建事务脚本（也会被-b tpcb-like调用）会在每个事务上发出七个从aid、tid、bid和delta中随机选择的命令。该场景来自于 TPC-B 基准，但并不是真正的 TPC-B，只是名字像而已。

1. BEGIN;
2. UPDATE ltbench\_accounts SET abalance = abalance + :delta WHERE aid = :aid;
3. SELECT abalance FROM ltbench\_accounts WHERE aid = :aid;
4. UPDATE ltbench\_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
5. UPDATE ltbench\_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
6. INSERT INTO ltbench\_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT\_TIMESTAMP);
7. END;

如果选择simple-update内建脚本（还有-N），第 4 和 5 步不会被包括在事务中。这将避免更新那些表中的内容，但是它会让该测试用例更不像 TPC-B。

如果选择select-only内建脚本（还有-S），只会发出SELECT。

## 自定义脚本

ltbench支持通过从一个文件中（-f选项）读取事务脚本替换默认的事务脚本（如上文所述）来运行自定义的基准场景。在这种情况下，一个“事务”就是一个脚本文件的一次执行。

脚本文件包含一个或者多个被分号终结的 SQL 命令。空行以及以--开始的行会被忽略。脚本文件也可以包含“元命令”，它会由ltbench自身解释，详见下文。

### Note

在LightDB 9.6 之前，脚本文件中的 SQL 命令被新行终结，因此命令不能跨行。现在需要分号来分隔连续的 SQL 命令（如果

SQL 命令后面跟着一个元命令则不需要一个分号)。如果需要创建一个能在新旧版本ltpbench下工作的脚本文件, 要确保把每个 SQL 命令写在一个由分号终结的行中。

对脚本文件有一种简单的变量替换功能。变量名必须由字母(包括非拉丁字母)、数字以及下划线构成, 并且首个字符不能是数字。如上所述, 变量可以用命令行的 -D选项设置, 或者按下文所说的使用元命令设置。除了用-D命令行选项预先设置的任何变量之外, 还有一些被自动预先设置的变量, 它们被列在Table 272中。一个用-D为这些变量值指定的值会优先于自动的预设值。一旦被设置, 可以在 SQL 命令中写:variablename来插入一个变量的值。当运行多于一个客户端会话时, 每一个会话拥有它自己的变量集合。ltpbench在一个语句中支持使用多达255个变量。

Table 272. ltpbench 自动变量

变量	简介
client_id	标识客户端会话的唯一编号(从零开始)
default_seed	默认在哈希和伪随机置换函数中使用的种子
random_seed	随机数生成器种子(除非用-D重载)
scale	当前的缩放因子

脚本文件元命令以反斜线(\)开始并且通常延伸到行的末尾, 不过它们也能够通过写一个反斜线回车继续到额外行。一个元命令和它的参数用空白分隔。支持的元命令是:

```
\gset [prefix] \aset [prefix]
```

这些命令可以用于结束 SQL 查询, 代替终止分号 (;)。

当使用\gset命令时, 前面的 SQL 查询预期返回一行, 存储变量名的列在列名后面, 如果已经提供的话, 则以prefix作为前缀。

使用\aset命令时, 所有组合的SQL查询(由\;分隔)将其列存储到以列名命名的变量中, 如果提供, 则以前缀作为前缀。如果查询不返回任何行, 则不进行赋值, 并且可以测试变量是否存在以检测这一点。如果查询返回多行, 则保留最后一个值。

下面的示例将第一个查询中的最终帐户余额放入变量abalance, 并且用第三个查询中的整数填充变量p\_two和p\_three。第二个查询的结果将被丢弃。最后两个组合查询的结果存储在变量four 和 five 中。

```
UPDATE ltpbench_accounts
  SET abalance = abalance + :delta
  WHERE aid = :aid
  RETURNING abalance \gset
-- compound of two queries
SELECT 1 \;
SELECT 2 AS two, 3 AS three \gset p_
SELECT 4 AS four \; SELECT 5 AS five \aset
```

```
\if expression
\elif expression
\else
\endif
```

这一组命令实现了可嵌套的条件块, 类似于ltsql的\if expression。条件表达式与\set的相同, 非零值会被解释为真。

```
\set varname expression
```

设置变量varname为一个从expression计算出的值。该表达式可以包含NULL常量、布尔常量TRUE和FALSE、5432这样的整数常量、3.14159这样的double常量、对变量的引用:variablename、操作符（保持它们通常的SQL优先级和结合性）、函数调用、SQL CASE一般条件表达式以及括号。

函数和大部分操作符在NULL输入上会返回NULL。

对于条件目的，非零数字值是TRUE，数字零值以及NULL是FALSE。

太大或太小的整数和双常量，以及整数算术运算符（+，-，\* 和 /）会引发溢出错误。

在没有为CASE提供最终的ELSE子句时，默认值是NULL。

示例：

```
\set ntellers 10 * :scale
\set aid (1021 * random(1, 100000 * :scale)) % \
        (100000 * :scale) + 1
\set divx CASE WHEN :x <> 0 THEN :y/:x ELSE NULL END
```

```
\sleep number [ us | ms | s ]
```

导致脚本执行休眠指定的时间，时间的单位可以是微妙（us）、毫秒（ms）或者秒（s）。如果单位被忽略，则秒是默认值。number要么是一个整数常量，要么是一个引用了具有整数值的变量的:variablename。

例子：

```
\sleep 10 ms
```

```
\setshell varname command [ argument ... ]
```

用给定的argument设置变量varname为 shell 命令command的结果。该命令必须通过它的标准输出返回一个整数值。

command和每个argument要么是一个文本常量，要么是一个引用了一个变量的:variablename。如果你想要使用以冒号开始的argument，在argument的开头写一个额外的冒号。

例子：

```
\setshell variable_to_be_assigned command
        literal_argument :variable ::literal_starting_with_colon
```

```
\shell command [ argument ... ]
```

与\setshell相同，但是结果被抛弃。

例子：

```
\shell command literal_argument :variable ::literal_starting_with_colon
```

## 内建操作符

Table 27中列举的算数、按位、比较以及逻辑操作符都被编译到了ltbench中并且可以被用于\set中出现的表达式中。运算符以升序排列。除非另有说明，如果任一输入为双精度，则采用两个数字输入的运算符将产生双精度值，否则产生整数结果。

Table 273. ltbench操作符

操作符 简介 例子
boolean OR boolean → boolean 逻辑或 5 or 0 → TRUE
boolean AND boolean → boolean 逻辑与 3 and 0 → FALSE
NOT boolean → boolean 逻辑非 not false → TRUE
boolean IS [NOT] (NULL TRUE FALSE) → boolean 布尔值测试 1 is null → FALSE
value ISNULL NOTNULL → boolean 空测试 1 notnull → TRUE
number = number → boolean 等于 5 = 4 → FALSE
number <> number → boolean 不等于 5 <> 4 → TRUE
number != number → boolean 不等于 5 != 5 → FALSE
number < number → boolean 小于 5 < 4 → FALSE
number <= number → boolean 小于等于 5 <= 4 → FALSE
number > number → boolean 大于 5 > 4 → TRUE
number >= number → boolean 大于等于 5 >= 4 → TRUE
integer   integer → integer 按位或 1   2 → 3
integer # integer → integer 按位异或 1 # 3 → 2
integer & integer → integer 按位与 1 & 3 → 1
~ integer → integer 按位非

操作符 简介 例子
$\sim 1 \rightarrow -2$
integer << integer $\rightarrow$ integer 按位左移 $1 \ll 2 \rightarrow 4$
integer >> integer $\rightarrow$ integer 按位右移 $8 \gg 2 \rightarrow 2$
number + number $\rightarrow$ number 加 $5 + 4 \rightarrow 9$
number - number $\rightarrow$ number 减 $3 - 2.0 \rightarrow 1.0$
number * number $\rightarrow$ number 乘 $5 * 4 \rightarrow 20$
number / number $\rightarrow$ number 除法（如果两个输入都是整数，则将结果截断为零） $5 / 3 \rightarrow 1$
integer % integer $\rightarrow$ integer 模（余数） $3 \% 2 \rightarrow 1$
- number $\rightarrow$ number 取反 $- 2.0 \rightarrow -2.0$

## 内建函数

Table 27中列出的函数被编译在ltbench中，并且可能被用在出现于\set的表达式中。

Table 274. ltbench 函数

函数 简介 例子
abs ( number ) $\rightarrow$ 与输入类型相同 绝对值 $\text{abs}(-17) \rightarrow 17$
debug ( number ) $\rightarrow$ 与输入类型相同 将参数打印到stderr，并返回参数。 $\text{debug}(5432.1) \rightarrow 5432.1$
double ( number ) $\rightarrow$ double 造型成double。 $\text{double}(5432) \rightarrow 5432.0$
exp ( number ) $\rightarrow$ double 指数（e 给定幂） $\text{exp}(1.0) \rightarrow 2.718281828459045$
greatest ( number [ , ... ] ) $\rightarrow$ double 如果任何参数是双精度的，否则 integer 选择参数中的最大值。 $\text{greatest}(5, 4, 3, 2) \rightarrow 5$

函数 简介 例子
hash ( value [, seed ] ) → integer 这是 hash_murmur2 的别名。 hash(10, 5432) → -5817877081768721676
hash_fnv1a ( value [, seed ] ) → integer 计算 FNV-1a hash <sup>1</sup> 。 hash_fnv1a(10, 5432) → -7793829335365542153
hash_murmur2 ( value [, seed ] ) → integer 计算 MurmurHash2 hash <sup>2</sup> 。 hash_murmur2(10, 5432) → -5817877081768721676
int ( number ) → integer 造型成int。 int(5.4 + 3.8) → 9
least ( number [, ... ] ) → double 如果任何参数是双精度的, 否则 integer 选择参数中的最小值。 least(5, 4, 3, 2.1) → 2.1
ln ( number ) → double 自然对数 ln(2.718281828459045) → 1.0
mod ( integer, integer ) → integer 模 (余数) mod(54, 32) → 22
pi () → double π 的近似值 pi() → 3.14159265358979323846
pow ( x, y ) → double power ( x, y ) → double x 的 y 次幂 pow(2.0, 10) → 1024.0
random ( lb, ub ) → integer 计算[lb, ub]中的均匀分布随机整数。 random(1, 10) → 介于1和10之间的整数
random_exponential ( lb, ub, parameter ) → integer 计算[lb, ub]中的指数分布随机整数, 如下所示。 random_exponential(1, 10, 3.0) → 介于1和10之间的整数
random_gaussian ( lb, ub, parameter ) → integer 计算[lb, ub]中的高斯分布随机整数, 如下所示。 random_gaussian(1, 10, 2.5) → 介于1和10之间的整数
random_zipfian ( lb, ub, parameter ) → integer 计算[lb, ub]中的Zipfian分布随机整数, 如下所示。 random_zipfian(1, 10, 1.5) → 介于1和10之间的整数
sqrt ( number ) → double 平方根 sqrt(2.0) → 1.414213562

random函数使用均匀分布生成值, 即所有的值都以相等的概率从指定的范围中抽出。random\_exponential、random\_gaussian以及random\_zipfian函数要求一个额外的double 参数, 它决定分布的精确形状。

<sup>1</sup> [https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo\\_hash\\_function](https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function)

<sup>2</sup> <https://en.wikipedia.org/wiki/MurmurHash>

- 对于指数分布，parameter通过在parameter处截断一个快速下降的指数分布来控制分布，然后投影到边界之间的整数上。确切地说，

$$f(x) = \exp(-\text{parameter} * (x - \text{min}) / (\text{max} - \text{min} + 1)) / (1 - \exp(-\text{parameter}))$$

然后min和max之间（包括两者）的值i会被以概率 $f(i) - f(i + 1)$ 抽出。

直观上，parameter越大，接近min的值会被越频繁地访问，并且接近max的值会被越少访问。parameter越接近 0，访问分布会越平坦（更均匀）。该分布的粗近似值是范围中当时被抽取 parameter% 次接近min的最频繁的 1% 值。parameter值必须严格为正。

- 对于高斯分布，区间被映射到一个在左边-parameter和右边+parameter截断的标准正态分布（经典钟型高斯曲线）。区间中间的值更可能被抽到。准确地说，如果 $\text{PHI}(x)$ 是标准正态分布的累计分布函数，均值mu定义为 $(\text{max} + \text{min}) / 2.0$ ，有

$$f(x) = \text{PHI}(2.0 * \text{parameter} * (x - \text{mu}) / (\text{max} - \text{min} + 1)) / (2.0 * \text{PHI}(\text{parameter}) - 1)$$

则min和max（包括两者）之间的值i被抽出的概率是： $f(i + 0.5) - f(i - 0.5)$ 。直观上，parameter越大，靠近区间终端的值会被越频繁地抽出，并且靠近上下界两端的值会被更少抽出。大约 67% 的值会被从中间 $1.0 / \text{parameter}$ 的地方抽出，即均值周围 $0.5 / \text{parameter}$ 的地方。并且 95% 的值会被从中间 $2.0 / \text{parameter}$ 的地方抽出，即均值周围 $1.0 / \text{parameter}$ 的地方。例如，如果parameter是 4.0，67% 的值会被从该区间的中间四分之一（ $1.0 / 4.0$ ）抽出（即从 $3.0 / 8.0$ 到 $5.0 / 8.0$ ）。并且 95% 的值会从该区间的中间一半（ $2.0 / 4.0$ ）抽出（第二和第三四分位）。最小允许的parameter值为 2.0。

- random\_zipfian生成一个有界的Zipfian分布。parameter定义该分布有多么倾斜。parameter越大，绘制越接近间隔开头的值越频繁。分布是这样的，假设范围从1开始，绘制k与绘制k+1的概率之比为  $((k+1)/k)**\text{parameter}$ 。例如，random\_zipfian(1, ..., 2.5)生成值1大约 $(2/1)**2.5 = 5.66$ 次，比 2 更频繁，它本身被产生 $(3/2)**2.5 = 2.76$ 次，比3更频繁，依此类推。

ltbench的实现是基于“Non-Uniform Random Variate Generation”，Luc Devroye, p. 550-551, Springer 1986。由于该算法的限制，parameter值限制范围为[1.001, 1000]。

哈希函数 hash、hash\_murmur2 和 hash\_fnv1a 接受一个输入值和一个可选的种子参数。如果未提供种子，则使用 :default\_seed 的值，除非通过命令行选项 -D 设置。哈希函数可用于散列随机函数（例如 random\_zipfian 或 random\_exponential）的分布。例如，以下的 ltbench 脚本模拟了社交媒体和博客平台的可能真实工作负载，其中少数账户会产生过多的负载：

```
\set r random_zipfian(0, 100000000, 1.07)
\set k abs(hash(:r)) % 1000000
```

在某些情况下，需要几个不相互关联的不同分布，这时隐式的种子参数非常有用：

```
\set k1 abs(hash(:r, :default_seed + 123)) % 1000000
\set k2 abs(hash(:r, :default_seed + 321)) % 1000000
```

作为一个例子，内建的类 TPC-B 事务的全部定义是：

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
```

```

\set delta random(-5000, 5000)
BEGIN;
UPDATE ltpbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM ltpbench_accounts WHERE aid = :aid;
UPDATE ltpbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE ltpbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO ltpbench_history (tid, bid, aid, delta, mtime) VALUES
(:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;

```

这个脚本允许该事务的每一次迭代能够引用不同的、被随机选择的行（这个例子也展示了为什么让每一个客户端会话有其自己的变量很重要 — 否则它们不会独立地接触不同的行）。

## 对每个事务做日志

通过-l选项（但是没有--aggregate-interval选项），ltpbench把关于每个事务的信息写入到一个日志文件。该日志文件将被命名为prefix.nnn，其中prefix默认为ltpbench\_log，而nnn是ltpbench进程的PID。前缀可以用--log-prefix选项更改。如果-j选项是2或者更高（有多个工作者线程），那么每一个工作者线程将会有它自己的日志文件。第一个工作者的日志文件的命名将和标准的单工作者情况相同。其他工作者的额外日志文件将被命名为prefix.nnn.mmm，其中mmm是每一个工作者的一个序列号，这种序列号从1开始编。

日志的格式是：

```
client_id transaction_no time script_no time_epoch time_us [ schedule_lag ]
```

其中client\_id表示哪个客户端会话运行该事务，transaction\_no是那个会话已经运行了多少个事务的计数，time是以微秒计的总共用掉的事务时间，script\_no标识了要使用哪个脚本文件（当用-f或者-b指定多个脚本时有用），而time\_epoch/time\_us是一个 Unix 纪元格式的时间戳以及一个显示事务完成时间的以微秒计的偏移量（适合于创建一个带有分数秒的ISO 8601 时间戳）。域schedule\_lag是事务的预定开始时间和实际开始时间之间的差别，以微秒计。只有使用--rate选项时它才存在。当--rate和--latency-limit同时被使用时，一个被跳过的事务的time会被报告为skipped。

这里是在单个客户端运行中生成的一个日志文件的片段：

```

0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663

```

另一个例子使用的是--rate=100以及--latency-limit=5（注意额外的 schedule\_lag列）：

```

0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
0 83 skipped 0 1412881037 914578 5217
0 83 skipped 0 1412881037 914578 5099
0 83 4722 0 1412881037 916203 3108
0 84 4142 0 1412881037 918023 2333
0 85 2465 0 1412881037 919759 740

```

在这个例子中，事务 82 迟到了，因为它的延迟（6.173 ms）超过了 5 ms 限制。接下来的两个事务被跳过，因为它们在开始之前就已经迟到了。

在能够处理大量事务的硬件上运行一次长时间的测试时，日志文件可能变得非常大。--sampling-rate选项能被用来只记录事务的一个随机采样。



## 聚合的日志记录

通过`--aggregate-interval`选项，日志文件会使用一种不同的格式：

```
interval_start num_transactions
sum_latency sum_latency_2 min_latency max_latency
[ sum_lag sum_lag_2 min_lag max_lag [ skipped ] ]
```

其中`interval_start`是区间的开始（作为一个Unix纪元的时间戳）、`num_transactions`是该区间中的事务数、`sum_latency`是该区间中事务时延的总量、`sum_latency_2`是该区间中事务时延的平方和、`min_latency`是该区间中的最小时延、`max_latency`是该区间中的最大时延。接下来的字段`sum_lag`、`sum_lag_2`、`min_lag`以及`max_lag`只有在使用`--rate`选项时才存在。它们提供每个事务要等待前一个事务完成所花的时间的统计信息，即每个事务的计划启动时间与实际启动时间之间的差值。最后一个字段`skipped`只有在使用`--latency-limit`选项时才存在。它对因为启动过完被跳过的事务进行计数。每一个事务被计入在其提交时的区间中。

这里是一些输出示例：

```
1345828501 5601 1542744 483552416 61 2573
1345828503 7884 1979812 565806736 60 1479
1345828505 7208 1979422 567277552 59 1391
1345828507 7685 1980268 569784714 60 1398
1345828509 7073 1979779 573489941 236 1411
```

注意虽然纯（非聚合）日志文件显示为每个事务使用了哪个脚本，但聚合日志却不包含索引。因此如果你需要针对每个脚本的数据，你需要自行聚合数据。

## 每语句延迟

通过`-r`选项，`ltbench`收集每一个客户端执行的每一个语句花费的事务时间。然后在基准完成后，它会报告这些值的平均值，作为每个语句的延迟。

对于默认脚本，输出看起来会像这样：

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
latency average = 15.844 ms
latency stddev = 2.715 ms
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
statement latencies in milliseconds:
    0.002  \set aid random(1, 100000 * :scale)
    0.005  \set bid random(1, 1 * :scale)
    0.002  \set tid random(1, 10 * :scale)
    0.001  \set delta random(-5000, 5000)
    0.326  BEGIN;
    0.603  UPDATE ltbench_accounts SET abalance = abalance + :delta WHERE
aid = :aid;
    0.454  SELECT abalance FROM ltbench_accounts WHERE aid = :aid;
```

```
5.528 UPDATE ltbench_tellers SET tbalance = tbalance + :delta WHERE tid
= :tid;
7.335 UPDATE ltbench_branches SET bbalance = bbalance + :delta WHERE
bid = :bid;
0.371 INSERT INTO ltbench_history (tid, bid, aid, delta, mtime) VALUES
(:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
1.212 END;
```

如果指定了多个脚本文件，会为每一个脚本文件单独报告平均值。

注意为每个语句的延迟计算收集额外的时间信息会增加一些负荷。这将拖慢平均执行速度并且降低计算出的 TPS。降低的总量会很显著地依赖于平台和硬件。对比使用和不适用延迟报告时的平均 TPS 值是评估时间开销是否明显的好方法。

## 良好的做法

很容易使用ltbench产生完全没有意义的数字。这里有一些指导可以帮你得到有用的结果。

排在第一位的是，永远不要相信任何只运行了几秒的测试。使用-t或-T选项让运行持续至少几分钟，这样可以用平均值去掉噪声。在一些情况中，你可能需要数小时来得到能重现的数字。多运行几次测试是一个好主意，这样可以看看你的数字是不是可以重现。

对于默认类 TPC-B 测试场景，初始化的比例因子 (-s) 应该至少和你想要测试的最大客户端数量一样大 (-c)，否则你将主要在度量更新争夺。在ltbench\_branches表中只有-s行，并且每个事务都想更新其中之一，因此-c值超过-s将毫无疑问地导致大量事务被阻塞来等待其他事务。

默认的测试场景也对表被初始化了多久非常敏感：表中死亡行和死亡空间的累积会改变结果。要理解结果，你必须跟踪更新的总数以及何时发生清理。如果开启了自动清理，它可能会在度量的性能上产生不可预估的改变。

ltbench的一个限制是在尝试测试大量客户端会话时，它自身可能成为瓶颈。这可以通过在数据库服务器之外的一台机器上运行ltbench来缓解，不过必须是具有低网络延迟的机器。甚至可以在多个客户端机器上针对同一个数据库服务器并发地运行多个ltbench实例。

## 安全性

如果不可信用户能够访问没有采用安全方案使用模式的数据库，不要在那个数据库中运行ltbench。ltbench使用非限定名称并且不会操纵搜索路径。

---

# ltcenter

ltcenter — ltcenter 是一个命令行管理工具，用于观察和排查 LightDB 问题。

## 主要目标

LightDB 提供有关其运行时的各种活动统计信息，例如连接、语句、数据库操作、复制、资源使用情况等。统计信息的一般目的是帮助 DBA 监视和排查 LightDB。但是，这些以文本形式提供的统计信息是从 SQL 函数和视图中检索的，LightDB 不提供用于处理统计信息视图的原生工具。

ltcenter 的主要目标是帮助 LightDB DBA 处理统计信息，并提供一种方便的方式观察 LightDB 的运行时状态。

## 主要功能

- 类似 top 的界面，允许您在监视统计信息更改时进行监视。
- 配置管理功能允许查看和编辑当前配置文件，并在需要时重新加载服务。
- 日志文件功能允许您快速检查 LightDB 日志而不停止统计信息监视。
- “贫民监控”允许您将 LightDB 统计信息收集到文件中，并稍后生成报告。
- 等待事件分析器允许查看查询执行过程中发生的等待事件。

## 支持的统计信息

### LightDB 统计信息

- 汇总活动 - 从不同的源中编译/选择的指标 - LightDB 运行时间、版本、恢复状态、按状态分组的客户端数量、（自动）清理数、每秒语句执行次数、最长事务的年龄和最长清理时间。
- `pg_stat_activity`<sup>1</sup> - 已连接客户端和后台进程的活动情况。
- `pg_stat_database`<sup>2</sup> - 数据库范围和会话统计信息，例如提交/回滚次数、处理的元组、死锁、临时文件等。
- `pg_stat_replication`<sup>3</sup> - 复制统计信息，例如已连接的备库、它们的活动和复制滞后。
- `pg_stat_user_tables`<sup>4</sup>, `pg_statio_user_tables`<sup>5</sup> - 访问（包括 I/O）表的统计信息。
- `pg_stat_user_indexes`<sup>6</sup>, `pg_statio_user_indexes`<sup>7</sup> - 访问（包括 I/O）索引的统计信息。
- `pg_stat_user_functions`<sup>8</sup> - 函数执行的统计信息。
- `pg_stat_wal`<sup>9</sup> - WAL 使用统计信息。

---

<sup>1</sup> [www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-ACTIVITY-VIEW](http://www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-ACTIVITY-VIEW)

<sup>2</sup> [www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-DATABASE-VIEW](http://www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-DATABASE-VIEW)

<sup>3</sup> [www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-REPLICATION-VIEW](http://www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-REPLICATION-VIEW)

<sup>4</sup> [www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-ALL-TABLES-VIEW](http://www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-ALL-TABLES-VIEW)

<sup>5</sup> [www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STATIO-ALL-TABLES-VIEW](http://www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STATIO-ALL-TABLES-VIEW)

<sup>6</sup> [www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-ALL-INDEXES-VIEW](http://www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-ALL-INDEXES-VIEW)

<sup>7</sup> [www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STATIO-ALL-INDEXES-VIEW](http://www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STATIO-ALL-INDEXES-VIEW)

<sup>8</sup> [www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-USER-FUNCTIONS-VIEW](http://www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-USER-FUNCTIONS-VIEW)

<sup>9</sup> [www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-WAL-VIEW](http://www.hs.net/lightdb/docs/current/monitoring-stats.html#PG-STAT-WAL-VIEW)

- `lt_stat_statements`<sup>10</sup> - 执行的 SQL 语句的统计信息，包括时间和资源使用情况。
- 基于 `pg_relation_size()` 和 `pg_total_relation_size()` 函数的表大小统计信息；
- `pg_stat_progress_vacuum`<sup>11</sup> - （自动）清理操作的进度。
- `pg_stat_progress_cluster`<sup>12</sup> - CLUSTER 和 VACUUM FULL 操作的进度。
- `pg_stat_progress_create_index`<sup>13</sup> - CREATE INDEX 和 REINDEX 操作的进度。
- `pg_stat_progress_analyze`<sup>14</sup> - ANALYZE 操作的进度。
- `pg_stat_progress_basebackup`<sup>15</sup> - basebackup 操作的进度。
- `pg_stat_progress_copy`<sup>16</sup> - COPY 操作的进度。

## 系统统计信息

`ltcenter top` 还提供基于 `procfs` 文件系统的系统使用情况信息：

- 负载平均值和 CPU 使用时间（用户、系统、nice、空闲、iowait、软件和硬件中断、steal）；
- 内存和交换空间使用情况，缓存和脏内存的数量，写回活动；
- 存储设备统计信息：IOPS、吞吐量、延迟、平均队列和请求大小、设备利用率；
- 网络接口统计信息：字节和数据包吞吐量、不同种类的错误、饱和和利用率。
- 挂载的文件系统使用情况统计信息：总大小、空闲/已用/保留空间和索引节点数量。

在连接到远程 LightDB 的情况下，有可能使用额外的 SQL 函数用于从远程主机检索 `/proc` 统计信息。

---

<sup>10</sup> [www.hs.net/lightdb/docs/current/pgstatstatements.html](http://www.hs.net/lightdb/docs/current/pgstatstatements.html)

<sup>11</sup> [www.hs.net/lightdb/docs/current/progress-reporting.html#VACUUM-PROGRESS-REPORTING](http://www.hs.net/lightdb/docs/current/progress-reporting.html#VACUUM-PROGRESS-REPORTING)

<sup>12</sup> [www.hs.net/lightdb/docs/current/progress-reporting.html#CLUSTER-PROGRESS-REPORTING](http://www.hs.net/lightdb/docs/current/progress-reporting.html#CLUSTER-PROGRESS-REPORTING)

<sup>13</sup> [www.hs.net/lightdb/docs/current/progress-reporting.html#CREATE-INDEX-PROGRESS-REPORTING](http://www.hs.net/lightdb/docs/current/progress-reporting.html#CREATE-INDEX-PROGRESS-REPORTING)

<sup>14</sup> [www.hs.net/lightdb/docs/current/progress-reporting.html#ANALYZE-PROGRESS-REPORTING](http://www.hs.net/lightdb/docs/current/progress-reporting.html#ANALYZE-PROGRESS-REPORTING)

<sup>15</sup> [www.hs.net/lightdb/docs/current/progress-reporting.html#BASEBACKUP-PROGRESS-REPORTING](http://www.hs.net/lightdb/docs/current/progress-reporting.html#BASEBACKUP-PROGRESS-REPORTING)

<sup>16</sup> [www.hs.net/lightdb/docs/current/progress-reporting.html#COPY-PROGRESS-REPORTING](http://www.hs.net/lightdb/docs/current/progress-reporting.html#COPY-PROGRESS-REPORTING)

---

# ltsql

ltsql — LightDB的交互式终端

## Synopsis

```
ltsql [option...] [dbname [username]]
```

## 描述

ltsql是一个LightDB的基于终端的前端。它让你能交互式地键入查询，把它们发送给LightDB，并且查看查询结果。或者，输入可以来自于一个文件或者命令行参数。此外，ltsql还提供一些元命令和多种类似 shell 的特性来为编写脚本和自动化多种任务提供便利。

## 选项

-a  
--echo-all

把所有非空输入行按照它们被读入的形式打印到标准输出（不适用于交互式行读取）。这等效于把变量ECHO设置为 all。

-A  
--no-align

切换到非对齐输出模式（默认输出模式是对齐的）。这等效于\pset format unaligned。

-b  
--echo-errors

把失败的 SQL 命令打印到标准错误输出。这等效于把变量ECHO设置为errors。

-c command  
--command=command

指定ltsql执行一个给定的命令字符串command。这个选项可以重复多次并且以任何顺序与-f选项组合在一起。当-c或者-f被指定时，ltsql不会从标准输入读取命令，直到它处理完序列中所有的-c和-f选项之后终止。

command必须是一个服务器完全可解析的命令字符串（即不包含ltsql相关的特性）或者单个反斜线命令。因此不能在一个-c选项中混合SQL和ltsql元命令。要那样做，可以使用多个-c选项或者把字符串用管道输送到ltsql中，例如：

```
ltsql -c '\x' -c 'SELECT * FROM foo;'
```

或者

```
echo '\x \\ SELECT * FROM foo;' | ltsql
```

（\\是分隔符元命令）。

每一个被传递给-c的SQL命令字符串会被当做一个单独的请求发送给服务器。因此，即便该字符串包括多个SQL命令，服务器也会把它当做一个事务来执行，除非在该字符串中有显式的BEGIN/COMMIT命令把它划分成多个事务（服务器如何处理多查询字符串的更多细节请参考Section 49.2.2.1）。此外，ltsql只会打印出该字符串中最后一个SQL命令的

结果。这和从文件中读取同一字符串或者把同一字符串传给ltsql的标准输出时的行为不同，因为那两种情况下ltsql会独立地发送每一个SQL命令。

由于这种行为，把多于一个SQL命令放在-c字符串中通常会得到意料之外的结果。最好使用多个-c命令或者把多个命令输送给ltsql的标准输入，按照上文所说的使用echo或者通过一个 shell，例如：

```
ltsql <<EOF
\x
SELECT * FROM foo;
EOF
```

--csv

切换到CSV（逗号分隔值）输出模式。这相当于\pset format csv。

-d dbname

--dbname=dbname

指定要连接的数据库的名称。这等效于指定dbname为命令行上的第一个非选项参数。dbname 可以是 连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

-e

--echo-queries

也把发送到服务器的所有 SQL 命令复制到标准输出。这等效于把变量ECHO设置为queries。

-E

--echo-hidden

回显\d以及其他反斜线命令生成的实际查询。可以用它来学习ltsql的内部操作。这等效于把变量ECHO\_HIDDEN设置为on。

-f filename

--file=filename

从文件filename而不是标准输入中读取命令。这个选项可以被重复多次，也可以以任意顺序与-c选项组合。当-c或者-f被指定时，ltsql不会从标准输入读取命令，直到它处理完序列中所有的-c和-f选项之后终止。除此以外，这个选项很大程度上等价于元命令\i。

如果filename是-（连字符），那么会读取标准输入直到遇见一个 EOF 指示或者\q元命令。这种方式可以用把自多个文件的输入组合成一种交互式输入。不过注意在这种情况下不会使用 Readline（很像指定了-n的情况）。

使用这个选项与ltsql < filename有细微的不同。通常，两种形式都可以做到我们所期望的，但是使用-f启用了一些好的特性，例如带有行号的错误消息。使用这个选项还有一丝机会可以降低启动开销。在另一方面，使用 shell输入重定向的变体（理论上）保证会得到与手工输入时相同的输出。

-F separator

--field-separator=separator

使用separator作为非对齐输出的域分隔符。这等效于\pset fieldsep或者\f。

-h hostname

--host=hostname

指定运行服务器的机器的主机名。如果这个值由一个斜线开始，它会被用作 Unix 域套接字的目录。

-H  
--html

切换到HTML输出模式。这等效于`\pset format html`或者`\H`命令。

-l  
--list

列出所有可用的数据库，然后退出。其他非连接选项会被忽略。这与元命令`\list`类似。

在使用这个选项时，`ltsql`将连接到数据库`postgres`，除非在命令行上提及一个不同的数据（选项`-d`或非选项参数，可能是通过一个服务项，但不能通过一个环境变量）。

-L filename  
--log-file=filename

除了把所有查询输出写到普通输出目标之外，还写到文件`filename`中。

-n  
--no-readline

不使用`Readline`做行编辑并且不使用命令历史。在剪切和粘贴时，关掉 `Tab` 展开会有所帮助。

-o filename  
--output=filename

把所有查询输出放到文件`filename`中。这等效于命令`\o`。

-p port  
--port=port

指定服务器用于监听连接的 `TCP` 端口或者本地 `Unix` 域套接字文件扩展。默认是`LTPORT`环境变量的值，如果没有设置，则默认为编译时指定的端口号（通常是5432）。

-P assignment  
--pset=assignment

以`\pset`的形式指定打印选项。注意，这里你必须用一个等号而不是空格来分隔名称和值。例如，要设置输出格式为`LaTeX`，应该写成`-P format=latex`。

-q  
--quiet

指定`ltsql`应该安静地工作。默认情况下，它会打印出欢迎消息以及多种输出。如果使用了这个选项，以上那些就都不会输出。在使用`-c`选项时，配合这个选项很有用。这等效于设置变量`QUIET`为`on`。

-R separator  
--record-separator=separator

把`separator`用作非对齐输出的记录分隔符。这等效于`\pset recordsep`命令。

-s  
--single-step

运行在单步模式中。这意味着在每个命令被发送给服务器之前都会提示用户一个可以取消执行的选项。使用这个选项可以调试脚本。

-S  
--single-line

运行在单行模式中，其中新行会终止一个 `SQL` 命令，就像分号的作用一样。

### Note

这种模式被提供给那些坚持使用它的用户，但是并不一定要使用它。特别地，如果在一行中混合了SQL和元命令，那对于没有经的用户来说，它们的执行顺序可能不总是那么清晰。

-t  
--tuples-only

关闭打印列名和结果行计数页脚等。这等效于\t或者\pset tuples\_only命令。

-T table\_options  
--table-attr=table\_options

指定要替换HTML table标签的选项。详见\pset tableattr。

-U username  
--username=username

作为用户username而不是默认用户连接到数据库（当然，你必须具有这样做的权限）。

-v assignment  
--set=assignment  
--variable=assignment

执行一次变量赋值，和\set元命令相似。注意你必须在命令行上用等号分隔名字和值（如果有）。要重置一个变量，去掉等号就行。要把一个变量置为空值，使用等号但是去掉值。这些赋值在命令行处理期间被完成，因此反映连接状态的变量将在稍后被覆盖。

-V  
--version

打印ltsql版本并且退出。

-w  
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且口令不能从其他来源（例如一个.pgpass文件）获得，那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

注意这个选项将对整个会话保持设置，并且因此它会影响元命令\connect的使用，就像初始的连接尝试那样。

-W  
--password

强制ltsql在连接到一个数据库之前提示要求一个口令，即使口令不会被使用。

如果服务器需要口令认证并且口令不能从其他来源获得，例如.pgpass文件，ltsql在任何情况下都会提示输入口令。然而，ltsql将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

注意这个选项将对整个会话保持设置，并且因此它会影响元命令\connect的使用，就像初始的连接尝试那样。

-x  
--expanded

打开扩展表格式模式。这等效于\x或者\pset expanded命令。



-X,  
--no-ltsqlrc

不读取启动文件（要么是系统范围的ltsqlrc文件，要么是用户的~/.ltsqlrc文件）。

-z  
--field-separator-zero

设置非对齐输出的域分隔符为零字节。这等效于\pset fieldsep\_zero。

-0  
--record-separator-zero

设置非对齐输出的记录分隔符为零字节。例如，这对与xargs -0配合有关。这等效于\pset recordsep\_zero。

-1  
--single-transaction

这个选项只能被用于与一个或者多个-c以及/或者-f选项组合。它会让ltsql在第一个上述选项之前发出一条BEGIN命令并且在最后一个上述选项之后发出一条COMMIT命令，这样就把所有的命令都包裹在一个事务中。这个选项可以保证要么所有的命令都成功地完成，要么不应用任何更改。

如果命令本身包含BEGIN、COMMIT或者ROLLBACK，这个选项将不会得到想要的效果。还有，如果单个命令不能在一个事务块中执行，指定这个选项将导致整个事务失败。

-?  
--help[=topic]

显示有关ltsql的帮助并且退出。可选的topic参数（默认为options）选择要解释哪一部分的ltsql：commands描述ltsql的反斜线命令；options描述可以被传递给ltsql的命令选项；而variables则显示有关ltsql配置变量的帮助。

## 退出状态

如果ltsql正常完成，它会向 shell 返回 0。如果它自身发生一个致命错误（例如内存用完、找不到文件），它会返回 1。如果到服务器的连接出问题并且事务不是交互式的，它会返回 2。如果在脚本中发生错误，它会返回 3 并且变量ON\_ERROR\_STOP会被设置。

## 用法

### 连接到数据库

ltsql是一个常规LightDB客户端应用。为了连接到数据库，你需要知道你的目标数据库的名称、主机名和该服务器的端口号，还有要作为哪个用户名连接。可以通过命令行选项告知ltsql这些参数，分别是-d、-h、-p以及-U。如果发现一个参数不属于任何选项，它将被解释为数据库名称（如果已经给出数据库名称，就解释为用户名）。并非所有这些选项都是必需的，它们都有可用的默认值。如果省略主机名，ltsql将通过一个 Unix 域套接字连接到本地主机上的服务器，或者通过 TCP/IP 连接到没有 Unix 域套接字的主机上的localhost。默认端口号则在编译时决定。由于数据库服务器使用相同的默认值，大多数情况下你将不必指定端口。默认的用户名是你的操作系统用户名，它也会是默认的数据库名。注意你不一定能连接到任意用户名下的任何数据库。你的数据库管理员应该已经告知过你有关你的访问权限。

当默认值不是很符合实际时，可以把环境变量LTDATABASE、LTHOST、LTPORT以及LTUSER设置为适当的值，这样也能节省一些敲打键盘的工作（额外的环境变量可见Section 32.14）。用一个~/.pgpass文件来避免定期输入密码也很方便。详见Section 32.15

另一种指定连接参数的方法是用一个conninfo字符串或者一个URI，它可以被用来替代数据库名。这种机制可以让我们对连接具有很广的控制权。例如：

```
$ ltsql "service=myservice sslmode=require"
$ ltsql postgresql://dbmaster:5433/mydb?sslmode=require
```

用这种方式，你也可以把LDAP用于Section 32.1中描述的连接参数查找。可用连接选项的更多信息请见Section 32.1.2

如果由于任何原因（例如权限不足、服务器没有在目标主机上运行等）导致连接无法建立，ltsql将返回一个错误并且终止。

如果标准输入和标准输出都是一个终端，那么ltsql会把客户端编码设置成“auto”，这会使ltsql从区域设置（Unix 系统上的LC\_CTYPE环境变量）中检测合适的客户端编码。如果这样不起作用，可以使用环境变量LTCLIENTENCODING覆盖客户端编码。

## 输入 SQL 命令

在正常操作时，ltsql会提供一个提示符，该提示符是ltsql当前连接到的数据库名称后面跟上字符串=>。例如：

```
$ ltsql test
ltsql (13.8-23.4)
compatible type: postgresql
Type "help" for help.
```

```
test=>
```

在提示符下，用户可以键入SQL命令。正常情况下，当碰到一个表示命令终结的分号时，输入的行会被发送给服务器。一行的结束并不表示命令的完结。因此，为了清晰，可以把命令散布在多个行上。如果命令被发送并且执行而不产生错误，该命令的结果将会显示在屏幕上。

如果不可信用户对还没有采用安全方案使用模式的一个而数据库拥有访问，通过从search\_path移除公共可写的方案来开始你的会话。人们可以在连接字符串中加入options=-csearch\_path=或者在其他SQL命令之前发出SELECT pg\_catalog.set\_config('search\_path', '', false)。这种考虑并非专门针对ltsql，它适用于每一种执行任意SQL命令的接口。

只要执行命令，ltsql还会测试LISTEN和NOTIFY产生的异步通知。

虽然 C 风格的注释块会被传给服务器处理并且移除，ltsql会自己移除掉 SQL 标准的注释。

## prompt 命令

命令prompt将提示文本打印到标准输出，这对于在脚本输出中散布信息很有用，可以通过ltsql -f 文件名的方式使用，也可以在ltsql的交互模式中使用。命令prompt不区分大小写，允许命令前有空格、制表符和空白行。例如：

```
lightdb@postgres=# prompt create table start
create table start
lightdb@postgres=# create table test(id int);
CREATE TABLE
lightdb@postgres=# Prompt select start
select start
lightdb@postgres=# select count(*) from test;
```

```

count
-----
      0
(1 row)

```

## spool 命令

spool指令来源于oracle的sqlplus工具。在交互模式下该指令可以将sqlplus的输出重定向至指定文件。此命令是大小写无关的且需要跟随某个文件路径或用关键词'off'将其重定向至标准输出。虽然该指令能忽略 多余的空格，但是值得注意的是，其遇到分号或是换行符时将会停止。 示例：

```

lightdb@postgres=#spool test_spool.log;
lightdb@postgres=#create table spool1(spool int);
lightdb@postgres=#insert into spool1 values(5);
lightdb@postgres=#select * from spool1;
lightdb@postgres=#spool off;
-- 检查test_spool.log是否包含记录
lightdb@postgres=# \! cat test_spool.log
spool
-----
      5
(1 row)

lightdb@postgres=#select * from spool1;
spool
-----
      5
(1 row)

lightdb@postgres=#spool;
lightdb@postgres=#select * from spool1;
spool
-----
      5
(1 row)

```

## rename 命令

rename指令来源于oracle的sqlplus工具。在交互模式下该指令可以对一个表或视图通过给定的参数"old\_table"及"new\_table"进行重命名。此命令是大小写无关的且忽略自身内部多余的空格但当遇到一个分号或换行符时将停止。因此用户无法跨行使用此命令。 示例：

```

lightdb@test_mysql_select=# rename table_oldname to table_newname;
ALTER TABLE

```

## 元命令

你输入到ltsql中的任何以未加引用的反斜线开始的东西都是一个ltsql元命令，它们由ltsql自行处理。这些命令让ltsql对管理和编写脚本更有用。元命令常常被称作斜线或者反斜线命令。

ltsql命令的格式是用反斜线后面直接跟上一个命令动词，然后是一些参数。参数与命令动词和其他参数之间用任意多个空白字符分隔开。

要在一个参数中包括空白，可以将它加上单引号。要在一个参数中包括一个单引号，则需要  
在文本中写上两个单引号。任何包含在单引号中的东西都服从与 C 语言中 \n（新  
行）、\t（制表符）、\b（退格）、\r（回车）、\f（换页）、\digits（10 进制）以及  
\xdigits（16 进制）类似的替换规则。单引号内文本中的其他任何字符（不管它是什  
么）前面的反斜线都没有实际意义（会被忽略）。

如果在一个参数中出现一个未加引号的冒号（:）后面跟着一个ltsql变量名，它会被该变量  
的值替换，如下面的SQL Interpolation所述。在其中描述的形式：`'variable_name'`和`"variable_name"`也有同样的效果。`:{?variable_name}`语法允许测  
试一个变量是否被定义。它会被TRUE或FALSE替换。用一个反斜线转义该冒号可以防止它被  
替换。

在一个参数中，封闭在反引号（`）中的文本会被当做一个传递给shell的命令行。该命令的  
输出（移除任何拖尾的新行）会替换反引号文本。在封闭在反引号的文本中，不会有特别的  
引号或者其他处理发生，`:variable_name`的出现除外，其中`variable_name`是一个会被其值  
替换的ltsql变量名。此外，Also, appearances of `'variable_name'`的出现会被替换为该  
变量的值，而值会被适当地加以引用以变成一个单一shell命令参数（后一种形式几乎总是  
优先，除非你非常确定变量中有什么）。因为回车和换行字符在所有的平台上都不能被安全  
地引用，`: 'variable_name'`形式会打印一个错误消息并且在这类字符出现在值中时不替换该  
变量值。

有些命令把SQL标识符（例如一个表名）当作参数。这些参数遵循SQL的语法规则：无引号  
的字母被强制变为小写，而双引号（"）可以保护字母避免大小写转换并且允许在标识符中包  
含空白。在双引号内，成对的双引号会被缩减为结果名称中的单个双引号。例  
如，`FOO"BAR"BAZ`会被解释成`fooBARbaz`，而`"A weird" name"`会变成`A weird' name`。

对参数的解析会在行尾或者碰到另一个未加引号的反斜线时停止。一个未加引号的反斜线被  
当做新元命令的开始。特殊的序列\\（两个反斜线）表示参数结束并且应继续解析SQL命令  
（如果还有）。使用这种方法，SQL命令和ltsql命令可以被自由地混合在一行中。但是无论  
在何种情况中，元命令的参数都无法跨越一行。

很多元命令作用在当前查询缓冲区上。这就是一个缓冲区而已，它保存任何已经被键入但是  
还没有发送到服务器执行的SQL命令文本。这将包括之前输入的行以及在该元命令同一行上  
出现在前面的任何文本。

可以使用下列元命令：

`\a`

如果当前的表输出格式是非对齐的，则切换成对齐格式。如果不是非对齐格式，则设置  
成非对齐格式。保留这个命令是为了向后兼容性。更一般的方案请见`\pset`。

`\c` or `\connect` [ `-reuse-previous=on|off` ] [ `dbname` [ `username` ] [ `host` ] [ `port`  
] | `conninfo` ]

与一台LightDB服务器建立一个新连接。可以使用位置语法（数据库名称、用户、主机和  
端口中的一个或多个）指定要使用的连接参数，或者使用Section 32.1.中详细介绍  
的`conninfo`连接串。如果没有给出参数，则使用与以前相同的参数建立新连接。

把`dbname`、`username`、`host`或者`port`中的任何一个指定为`-`等价于省略该参数。

新连接可以重用之前连接的连接参数；不仅是数据库名称、用户、主机和端口，还有其  
他设置，例如 `sslmode`。默认情况下，参数会在位置语法中重复使用，但在给出  
`conninfo` 字符串时不会。传递 `-reuse-previous=on` 或 `-reuse-previous=off`的  
第一个参数会覆盖该默认值。如果重复使用参数，则任何未明确指定为位置参数或在  
`conninfo` 字符串中的参数都将从现有连接的参数中获取。一个例外是，如果 `host` 设置  
使用位置语法从其先前的值更改，则现有连接参数中存在的任何 `hostaddr` 设置都将被  
删除。并且，用作已存在的连接的任何密码，只有在用户、主机、和端口设置没有变化  
的时候，将被重新使用。当命令既不指定也不重用特定参数时，将使用 `libpq` 默认值。

如果新连接成功地被建立，之前的连接会被关闭。如果连接尝试失败（错误的用户名、访问被拒绝等），在ltsql处于交互模式的情况下会保留之前的连接。但是当执行一个非交互式脚本时出现连接尝试失败，老的连接将关闭，并且报出一个错误。这可能会也可能不会终止脚本；如果没有，所有的数据库访问命令将失败，直到另一个\connect命令被成功执行。这种区别一方面可以帮助用户发现打字错误，另一方面也可以作为一种安全机制防止脚本在错误的数据库上执行动作。注意每当\connect命令尝试重用参数，重用的是最后一次成功连接的值，而不是后续任何失败尝试的。可是，在非交互的\connect失败的情况下，后续没有参数被允许重新使用，因为脚本会可能期望从失败的\connect中的值被重新使用。

例子：

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10
  sslmode=disable"
=> \c -reuse-previous=on sslmode=require -- changes only sslmode
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

### \C [ title ]

设置查询结果的任何表的标题，或者重置这类标题。这个命令等效于\pset title title（这个命令的名称来自于“caption”，因为它之前只被用来在HTML表格中设置标题）。

### \cd [ directory ]

把当前工作目录改为directory。如果不带参数，则切换到当前用户的主目录。

#### Tip

要打印当前的工作目录，可以使用\! pwd。

### \conninfo

输出有关当前数据库连接的信息。

```
\copy { table [ ( column_list ) ] } from { 'filename' | program 'command' | stdin
| pstdin } [ [ with ] ( option [, ... ] ) ] [ where condition ]
\copy { table [ ( column_list ) ] | ( query ) } to { 'filename' | program 'command'
| stdout | pstdout } [ [ with ] ( option [, ... ] ) ]
```

执行一次前端拷贝。这个操作会运行一个SQL COPY命令，不过不是服务器读取或者写入指定的文件，而是由ltsql读写文件并且把数据从本地文件系统导向服务器。这意味着文件的可访问性和权限是本地用户的而非服务器上的，并且不需要 SQL 超级用户特权。

当program被指定时，command被ltsql执行并且传给command的数据或者从command传出的数据会在服务器和客户端之间流动。同样地，执行特权是本地用户的而非服务器上的，并且不需要 SQL 超级用户特权。

对于\copy ... from stdin，数据行从发出该命令的同一来源读取，一直到读到\或者数据流到达EOF。这个选项可以用来填充内嵌在一个 SQL 脚本文件中的表。对于\copy ... to stdout，输出被发送到与ltsql命令输出相同的位置，并且COPY count命令的状态不会被打印（因为它会被一个数据行搞乱）。要读/写ltsql的标准输入或者输出而不管当前命令的来源或者\o选项，可以写from pstdin或者to pstdout。

这个命令的语法和SQL COPY命令类似。所有除开数据来源/目的地的选项都和COPY指定的一样。因此，\copy元命令由特殊的解析规则。与大部分其他元命令不同，该行的所有剩余部分总是会被当做\copy的参数，并且在参数中不会执行变量篡改以及反引号展开。

### Tip

获得与\copy ... to相同结果的另一种方法是使用SQL COPY ... TO STDOUT命令并使用\g filename或\g | program 终止它。 \copy不与同，此方法允许命令跨越多行；此外，可以使用变量插值和反引号扩展。

### Tip

这些操作不如带有文件或程序数据源或目标的SQL COPY命令有效，因为所有数据都必须通过客户端/服务器连接。对于大量数据，SQL命令可能更可取。

\copyright

显示LightDB的版权以及发布条款。

\crosstabview [ colV [ colH [ colD [ sortcolH ] ] ] ]

执行当前的查询缓冲区（像\g那样）并且在一个交叉表格子中显示结果。该查询必须返回至少三列。由colV标识的输出列会成为垂直页眉并且colH所标识的输出列会成为水平页眉。colD标识显示在格子中的输出列。sortcolH标识用于水平页眉的可选的排序列。

每一个列说明可以是一个列编号（从 1 开始）或者一个列名。常用的 SQL 大小写折叠和引用规则适用于列名。如果省略，colV被当做列 1 并且colH被当做列 2。colH必须和colV不同。如果没有指定colD，那么在查询结果中必须正好有三列，并且colV和colH之外的那一列会被当做colD。

垂直页眉显示为最左边的列，它包含列colV中找到的值，值的顺序和查询结果中的顺序相同，但是重复值会被移除。

水平页眉显示为第一行，它包含列colH中找到的值，其中的重复值被移除。默认情况下，这些值会以查询结果中相同的顺序出现。但是如果给出了可选的sortcolH参数，它标识一个值必须为整数编号的列，并且来自colH的值将会根据相应的sortcolH值排序后出现在水平页眉中。

在交叉表格子中，对于colH的每一个可区分的值x以及colV的每一个可区分的值y，位于交叉点(x, y)的单元包含colH值为x且colV值为y的查询结果行中colD列的值。如果没有这样的行，则该单元为空。如果有多个这样的行，则会报告一个错误。

\d[S+] [ pattern ]

对于每一个匹配pattern的关系（表、视图、物化视图、索引、序列或者外部表）或者组合类型，显示所有的列、它们的类型、表空间（如果非默认表空间）以及任何诸如NOT NULL或者默认值的特殊属性。相关的索引、约束、规则以及触发器也会被显示。对于外部表，还会显示相关的外部服务器（下文的Patterns中定义了“匹配模式”）。

对于某些类型的关系，\d会为每一列显示额外的信息：对于序列会显示列值，对于索引显示被索引的表达式，对于外部表显示外部数据包装器选项。

命令形式\d+相同，只是显示更多的信息：显示与表列相关的任何注释，显示表中OID的存在，如果关系是视图，则显示视图定义，以及非默认的副本标识设置。

默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

### Note

如果使用\d但不带有pattern参数，它等价于\dtvmsE，后者将显示所有可见的表、视图、物化视图、序列和外部表的列表。这纯粹是一种便利措施。

`\da[S] [ pattern ]`

列出聚集函数，以及它们的返回类型和它们所操作的数据类型。如果指定了pattern，只显示名称匹配该模式的聚集。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

`\dA[+] [ pattern ]`

列出访问方法。如果指定了pattern，只显示名称匹配该模式的访问方法。如果在命令名称后面追加+，则与访问方法相关的处理器函数和描述也会和访问方法本身一起被列出。

`\dAc[+] [access-method-pattern [input-type-pattern]]`

列出运算符类（请参阅 Section 36.16.1）。如果指定了 access-method-pattern，则仅列出与名称与该模式匹配的访问方法关联的运算符类。如果指定了 input-type-pattern，则仅列出与名称与该模式匹配的输入类型关联的运算符类。如果将 + 附加到命令名称，则会列出每个运算符类别及其关联的运算符系列和所有者。

`\dAf[+] [access-method-pattern [input-type-pattern]]`

列出运算符系列（请参阅 Section 36.16.5）。如果指定了 access-method-pattern，则仅列出与名称与该模式匹配的访问方法关联的运算符系列。如果指定了 input-type-pattern，则仅列出与名称与该模式匹配的输入类型关联的运算符系列。如果将 + 附加到命令名称，则会列出每个运算符系列及其所有者。

`\dAo[+] [access-method-pattern [operator-family-pattern]]`

列出与运算符系列关联的运算符（请参阅 Section 36.16.2）。如果指定了 access-method-pattern，则仅列出与名称与该模式匹配的访问方法关联的运算符系列的成员。如果指定了 operator-family-pattern，则仅列出名称与该模式匹配的运算符系列的成员。如果将 + 附加到命令名称，则会列出每个运算符及其排序运算符系列（如果它是排序运算符）。

`\dAp[+] [access-method-pattern [operator-family-pattern]]`

列出与运算符系列相关的支持函数（请参阅 Section 36.16.3）。如果指定了 access-method-pattern，则仅列出与名称与该模式匹配的访问方法关联的运算符系列的函数。如果指定了 operator-family-pattern，则仅列出名称与该模式匹配的运算符系列的函数。如果将 + 附加到命令名称，则会详细显示函数及其实际参数列表。

`\db[+] [ pattern ]`

列出表空间。如果指定了pattern，只显示名称匹配该模式的表空间。如果在命令名称后面追加+，则与表空间相关的选项、磁盘上的尺寸、权限以及描述也会和表空间本身一起被列出。

`\dc[S+] [ pattern ]`

列出字符集编码之间的转换。如果指定了pattern，只列出名称匹配该模式的转换。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。如果在命令名称后面追加+，则每一个对象相关的描述也会被列出。

`\dC[+] [ pattern ]`

列出类型转换。如果指定了pattern，只列出源类型和目标类型匹配该模式的转换。如果在命令名称后面追加+，则每一个对象相关的描述也会被列出。

`\dd[S] [ pattern ]`

显示约束、操作符类、操作符族、规则以及触发器类型对象的描述。所有其他注释可以通过那些对象类型相应的反斜线命令查看。

\dd显示匹配pattern的对象的描述，如果没有给出参数则显示合适类型的可见对象的描述。但是在任一种情况下都只列出具有描述的对象。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

对象的描述可以用SQL命令COMMENT创建。

`\dD[S+] [ pattern ]`

列出域。如果指定了pattern，只有名称匹配该模式的域会被显示。默认情况下，只有用户创建的对象会被显示，可以提供一个模式或者S修饰符以包括系统对象。如果+被追加到命令名称上，每一个被列出的对象会带有其相关的权限和描述。

`\ddp [ pattern ]`

列出默认的访问特权设置。对那些默认特权设置已经被改变得与内建默认值不同的角色（以及模式，如果适用），为每一个角色（以及模式）显示一项。如果指定了pattern，只列出角色名称或者模式名称匹配该模式的项。

ALTER DEFAULT PRIVILEGES命令被用来设置默认访问特权。在Section 5. 中解释了显示的权限的含义。

`\dE[S+] [ pattern ]`

`\di[S+] [ pattern ]`

`\dm[S+] [ pattern ]`

`\ds[S+] [ pattern ]`

`\dt[S+] [ pattern ]`

`\dv[S+] [ pattern ]`

在这一组命令中，字母E、i、m、s、t和v分别对应着外部表、索引、物化视图、序列、表和视图。你可以以任何顺序指定这些字母中的任意一个或者多个，这样可以得到这些类型的对象的列表。例如，\dti会列出表和索引。如果在命令名称后面追加+，则会列出每个对象及其持久性状态（永久、临时或未记录）、磁盘上的物理大小以及相关的描述（如果有）。如果指定了pattern，只列出名称匹配该模式的对象。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

`\des[+] [ pattern ]`

列出外部服务器（助记：“外部服务器”）。如果指定了pattern，只列出名称匹配该模式的那些服务器。如果使用了\des+形式，将显示每个服务器的完整描述，包括该服务器的访问特权、类型、版本、选项和描述。

`\det[+] [ pattern ]`

列出外部表（助记：“外部表”）。如果指定了pattern，只列出表名称或者模式名称匹配该模式的项。如果使用了\det+选项，一般选项和外部表描述也会被显示。

`\deu[+] [ pattern ]`

列出用户映射（助记：“外部用户”）。如果指定了pattern，只列出用户名匹配该模式的那些映射。如果使用了\deu+形式，有关每个映射的额外信息也会被显示。



### Caution

\deu+可能也会显示远程用户的用户名和口令，所以要小心不要把它们泄露出去。

\dew[+] [ pattern ]

列出外部数据包装器（助记：“外部包装器”）。如果指定了pattern，只列出名称匹配该模式的那些外部数据包装器。如果使用了\dew+形式，外部数据包装器的访问特权、选项和描述也会被显示。

\df[anptwS+] [ pattern ]

列出函数，以及它们的结果数据类型、参数数据类型和函数类型，函数类型被分为“agg”（聚集）、“normal”、“procedure”、“trigger”以及“window”。如果要只显示指定类型的函数，可以在该命令上增加相应的字母a、n、p、t或者w。如果指定了pattern，只显示名称匹配该模式的函数。任何附加参数都是类型名称模式，它们与函数的第一个、第二个、以及等等参数的类型名相匹配。（匹配函数可以有比指定的更多的参数。为了防止这种情况，写入破折号-作为最后一个arg\_pattern。）默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。如果使用了\df+形式，则有关每个函数的额外信息也会被显示，包括易失性、并行安全性、拥有者、安全性分类、访问特权、语言、源代码和描述。

### Tip

要查找接受特定数据类型参数或返回特定数据类型值的函数，请使用分页器的搜索功能在 \df 输出中滚动查找。

\dF[+] [ pattern ]

列出文本搜索配置。如果指定了pattern，只显示名称匹配该模式的配置。如果使用了\dF+形式，每种配置的完整描述也会被显示，包括底层的文本搜索解析器和用于每一种解析器记号类型的字典列表。

\dFd[+] [ pattern ]

列出文本搜索字典。如果指定了pattern，只显示名称匹配该模式的字典。如果使用了\dFd+形式，有关每一种选中的字典的额外信息也会被显示，包括底层的文本搜索模板和选项值。

\dFp[+] [ pattern ]

列出文本搜索解析器。如果指定了pattern，只显示名称匹配该模式的解析器。如果使用了\dFp+形式，每一种解析器的完整描述也会被显示，包括底层的函数和可识别的记号类型列表。

\dFt[+] [ pattern ]

列出文本搜索模板。如果指定了pattern，只显示名称匹配该模式的模板。如果使用了\dFt+形式，每一种模板有关的额外信息也会被显示，包括底层的函数名称。

\dg[S+] [ pattern ]

列出数据库角色（因为“用户”和“组”的概念已经被统一成“角色”，这个命令现在等价于\du）。默认情况下只会显示用户创建的角色，提供一个模式或者S修饰符可以把系统角色包括在内。如果指定了pattern，只列出名称匹配该模式的那些角色。如果使用了\dg+形式，有关每种角色的额外信息也将被显示，当前这种形式会为角色增加显示注释。

`\dl`

这是`\lo_list`的一个别名，它显示大对象的列表。

`\dL[S+] [ pattern ]`

列出过程语言。如果指定了`pattern`，只列出名称匹配该模式的语言。默认情况下只会显示用户创建的语言，提供一个模式或者`S`修饰符可以把系统对象包括在内。如果向命令名称追加`+`，则每一种语言会和它的调用处理器、验证器、访问特权以及它是否为系统对象一起列出。

`\dn[S+] [ pattern ]`

列出模式（名字空间）。如果指定了`pattern`，只列出名称匹配该模式的模式。默认情况下只会显示用户创建的对象，提供一个模式或者`S`修饰符可以把系统对象包括在内。如果向命令名称追加`+`，每个对象会与它相关的权限及描述（如果有）一起被列出。

`\do[S+] [ pattern ]`

列出操作符及其操作数和结果类型。如果指定了`pattern`，只列出名称匹配该模式的操作符。如果指定了一个`arg_pattern`，则只列出右侧参数的类型名称与所列出的模式相匹配的前缀操作符。如果指定了两个`arg_pattern`，则只列出参数类型名称与所列出的模式相匹配的二进制操作符。（或者，为一元运算符的未使用参数写入-。）默认情况下只会显示用户创建的对象，提供一个模式或者`S`修饰符可以把系统对象包括在内。如果向命令名称追加`+`，有关每个操作符的额外信息也将被显示，当前只包括底层函数的名称。

`\dO[S+] [ pattern ]`

列出排序规则。如果指定了`pattern`，只列出名称匹配该模式的排序规则。默认情况下只会显示用户创建的对象，提供一个模式或者`S`修饰符可以把系统对象包括在内。如果向命令名称追加`+`，每个排序规则将和它相关的描述（如果有）一起被列出。注意只有可用于当前数据库编码的排序规则会被显示，因此在同一个安装下的不同数据库中执行此命令可能会得到不同的结果。

`\dp [ pattern ]`

列表、视图和序列，包括与它们相关的访问特权。如果指定了`pattern`，只列出名称匹配该模式的表、视图以及序列。

`GRANT`和`REVOKE`命令被用来设置访问特权。所显示的特权的含义在Section 5. 中有介绍。

`\dP[itn+] [ pattern ]`

列出分区关系。如果指定了`pattern`，则仅列出名称与模式匹配的条目。修改符`t`（表）和`i`（索引）可以追加到命令中，筛选要列出的关系类型。默认会列出分区表和索引。

如果用了修饰符`n`（“nested”）或指定了模式，则包括非根分区关系，并显示每个分区关系的父级的列。

如果`+`被附加到命令中，那么还会显示每个关系分区的大小总和，以及关系的描述。如果`n`与`_`相结合，将显示两种大小：一种包含直接连接的叶分区的总大小，另一种显示所有分区的总大小，包括间接附加的子分区。

`\drds [ role-pattern [ database-pattern ] ]`

列出已定义的配置设置。这些设置可以是针对角色的、针对数据库的或者同时针对两者的。`role-pattern`和`database-pattern`分别被用来选择要列出的角色和数据库。如果省略它们或者指定了`*`，则会列出所有设置，分别会包括针对角色和针对数据库的设置。

`ALTER ROLE`以及`ALTER DATABASE`命令可以用来定义一个角色以及一个数据库的配置设置。

`\dRp[+] [ pattern ]`

列出复制的publication。如果指定有pattern，只有那些名称匹配该模式的publication会被列出。如果+被追加到命令的名称上，与每个publication相关的表也会被显示。

`\dRs[+] [ pattern ]`

列出复制的订阅。如果指定有pattern，只有那些名字匹配该模式的订阅才会被列出。如果+被追加到命令的名称上，订阅的额外属性会被显示。

`\dT[S+] [ pattern ]`

列出数据类型。如果指定了pattern，只列出名称匹配该模式的类型。如果向命令名称追加+，每一种类型、其内部名称和尺寸、允许的值（如果是一种enum类型）以及相关权限会被一同列出。默认情况下只会显示用户创建的对象，提供一个模式或者S修饰符可以把系统对象包括在内。

`\du[S+] [ pattern ]`

列出数据库角色（因为“用户”和“组”的概念已经被统一成“角色”，这个命令现在等价于\dg）。默认情况下只会显示用户创建的角色，提供一个模式或者S修饰符可以把系统角色包括在内。如果指定了pattern，只列出名称匹配该模式的那些角色。如果使用了\du+形式，有关每一种角色的额外信息也会被显示，当前只会多显示角色的注释。

`\dx[+] [ pattern ]`

列出已安装的扩展。如果指定了pattern，只列出名称匹配该模式的那些扩展。如果使用了\dx+形式，所有属于每个匹配扩展的对象会被列出。

`\dy[+] [ pattern ]`

列出事件触发器。如果指定了pattern，只列出名称匹配该模式的事件触发器。如果在命令名称后面加上+，还会为每个列出的对象显示其相关的描述。

`\e或\edit [ filename ] [ line_number ]`

如果指定了filename，则它将被编辑的文件，在编辑器退出后，该文件的内容会被拷贝到当前查询缓冲区中。如果没有给定filename，当前查询缓冲区会被拷贝到一个临时文件中，并且接着以相同的方式编辑。或者，如果当前查询缓冲区为空，则最近被执行的查询会被拷贝到一个临时文件并且以同样的方式编辑。

如果你编辑了一个文件或者以前的查询，并且你退出编辑时没有变更文件，查询缓冲将被清除。否则，会根据ltsql的一般规则重新解析查询缓冲区的新内容，把整个缓冲区当作一个单行来处理。任何完整的查询都会立即执行；也就是说，如果查询缓冲区包含分号或以分号结尾，则执行该点之前的所有内容并将其从查询缓冲区中删除。查询缓冲区中剩余的内容将重新显示。输入分号或者\g会把它发送出去，输入\r会通过清除查询缓冲区来取消它。

把缓冲区当作单行主要会影响元命令：缓冲区中在一个元命令之后的任何东西都将被当作该元命令的参数，即便元命令之后的内容跨越多行也是如此。（因此不能以这种方式来制作使用元命令的脚本。应该使用\i。）

如果指定了一个行号，ltsql将会把游标（注意不是服务器端的游标）定位到文件或者查询缓冲区的指定行上。注意如果给出了一个全是数字的参数，ltsql就会假定它是行号而不是文件名。

### Tip

有关如何配置和自定义编辑器的信息，请参见下面的Environment。

`\echo text [ ... ]`

将经过计算的参数打印到标准输出，以空格分隔，后跟换行符。这可以用来在脚本的输出中间混入信息，例如：

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

如果第一个参数是一个没有加引号的-n，则不会加上最后的新行（第一个参数也不会）。

### Tip

如果使用\o命令来重定向查询的输出，你可能希望使用\qecho来取代这个命令。另请参见\warn。

`\ef [ function_description [ line_number ] ]`

这个命令会以一个CREATE OR REPLACE FUNCTION或CREATE OR REPLACE PROCEDURE命令的形式取出并且编辑指定函数或过程的定义。编辑的方式与\edit完全相同。如果你退出编辑器而没有保存，语句会被丢弃。如果你保存并退出编辑器，如果你给它增加了分号，更新的命令会被立即执行 否则重新显示；可以键入分号或者\g把它发出，也可以用\r取消之。

目标函数可以单独用名称或者用名称和参数（例如foo(integer, text)）来指定。如果有多于一个函数具有同样的名称，则必须给出参数的类型。

如果没有指定函数，将会给出一个空白的CREATE FUNCTION模板来编辑。

如果指定了一个行号，ltsql将把光标定位在该函数体的指定行上（注意函数体通常不是开始于文件的第一行）。

和大部分其他元命令不同，该行的整个剩余部分总是会被当作\ef的参数，并且在参数中不会执行变量篡改以及反引号展开。

### Tip

有关如何配置和自定义编辑器可见下面的Environment。

`\encoding [ encoding ]`

设置客户端字符集编码。如果没有参数，这个命令会显示当前的编码。

`\errverbose`

以最详细的程度重复最近的服务器错误消息，就好像VERBOSITY被设置为verbose且SHOW\_CONTEXT被设置为always。

`\ev [ view_name [ line_number ] ]`

这个命令会以一个CREATE OR REPLACE VIEW的形式取出并且编辑指定函数的定义。编辑的方式与\edit完全相同。如果你退出编辑器而没有保存，语句会被丢弃。如果你保存并退出编辑器，如果你给它增加了分号，更新的命令会被立即执行 否则重新显示；可以键入分号或者\g把它发出，也可以用\r取消之。

如果没有指定函数，将会给出一个空白的CREATE VIEW模板来编辑。

如果指定了一个行号，ltsql将把游标定位在该视图定义的指定行上。

和大部分其他元命令不同，该行的整个剩余部分总是会被当作\ev的参数，并且在参数中不会执行变量篡改以及反引号展开。

\f [ string ]

设置用于非对齐查询输出的域分隔符。默认值是竖线（|）。它等效于\pset fieldsep。

\g [ (option=value [...]) ] [ filename ]

\g [ (option=value [...]) ] [ |command ]

把当前查询缓冲区发送给服务器执行。

如果括号出现在 \g 之后，它们将包围以空格分隔的 option=value 格式选项子句列表，其解释方式与 \psetoptionvalue 命令相同，但仅在此查询期间生效。在此列表中，= 符号周围不允许有空格，但选项子句之间需要空格。如果省略 =value，命名的 option 的更改方式与 \pset option 相同，没有明确的 value。

如果给定了 filename 或 |command 参数，则查询的输出将写入命名文件或通过管道传输到给定的 shell 命令，而不是像往常一样显示它。仅当查询成功返回零个或多个元组时才会写入文件或命令，而不是在查询失败或者是不返回数据的 SQL 命令时写入。

如果当前查询缓冲区为空，则重新执行最近一次被发送的查询。除了这种行为之外，没有参数的\g实际上等效于一个分号。对于参数，\g提供了一个替代\o命令的“一次性”，另外还允许对通常由\pset设置的输出格式选项进行一次性调整。

当最后一个参数以|开始时，则该行的所有剩余部分总是会被当做要执行的command，并且在参数中不会执行变量篡改以及反引号展开。该行的剩余部分会被简单地按字面传给 shell。

\gdesc

显示当前查询缓冲区的结果的描述（即列名和数据类型）。查询不会被实际执行，不过，如果它含有某种类型的语法错误，该错误将被以通常的方式报出。

如果当前查询缓冲区为空，则会描述最近被发送的查询。

\gexec

把当前查询缓冲区发送到服务器，然后该查询输出（如果有）中的每一行的每一列都当作一个要被执行的 SQL 语句。例如，要在my\_table的每一列上都创建一个索引：

```
=> SELECT format('create index on my_table(%I)', attname)
-> FROM pg_attribute
-> WHERE attrelid = 'my_table'::regclass AND attnum > 0
-> ORDER BY attnum
-> \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

产生的查询会按照其所在行被返回的顺序执行，如果有多个列，则同一行中按照从左至右的顺序执行。NULL 域会被忽略。产生的查询会被原样发送给服务器处理，因此它们即不能是ltsql元命令，也不能包含ltsql变量引用。如果其中任何一个查询失败，剩余查询的执行将会继续，除非设置了ON\_ERROR\_STOP。每个查询的执行都遵照ECHO的处理（在

使用\gexec时，通常建议设置ECHO为all或者queries)。查询日志、单步模式、计时以及其他查询执行特性也适用于每一个生成的查询。

如果当前查询缓冲区为空，则会重新执行最近被发送的查询。

\gset [ prefix ]

把当前查询输入缓冲区发送给服务器并且将查询的输出存储在ltsql变量中（参见下面的Variables）。被执行的查询必须只返回一行。该行的每一列会被存储到一个单独的变量中，变量和该列的名字一样。例如：

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset
=> \echo :var1 :var2
hello 10
```

如果指定了一个prefix，那么该字符串会被追加在该查询的输出列名称之前用来创建要使用的变量名：

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
hello 10
```

如果一个列的结果为 NULL，那么对应的变量会被重置而不是被设置。

如果查询失败或者没有返回一行，则不会有任何变量被更改。

如果当前查询缓冲区为空，则重新执行最近被发送的查询。

\gx [ (option=value [...]) ] [ filename ]  
 \gx [ (option=value [...]) ] [ |command ]

\gx等效于\g，\gx 等价于 \g，除了它强制此查询的扩展输出模式，就好像 expanded=on被包含在列表中 \pset 选项。请参考\x。

\h or \help [ command ]

给出指定SQL命令的语法帮助。如果没有指定command，则ltsql会列出可以显示语法帮助的所有命令。如果command是一个星号(\*)，则会显示所有SQL命令的语法帮助。

与大部分其他元命令不同，该行的所有剩余部分总是会被当做\help的参数，并且在参数中不会执行变量篡改以及反引号展开。

### Note

为了简化输入，由几个词构成的命令不需要被加上引号。因此，键入\help alter table是可以的。

\H or \html

开启HTML查询输出格式。如果HTML格式已经开启，这会把它切换回默认的对齐文本格式。这个命令是为了兼容性和方便，有关设置其他输出选项请见\pset。

\i or \include filename

从文件filename读取输入并且把它当作从键盘输入的命令来执行。

如果filename是-（连字符），那么会一直读取标准输入直到碰到一个 EOF 指示符或者\q元命令。这可以用来把交互式输入与文件输入混杂。注意只有在最外层激活了readline 行为的情况下才将会使用 readline 行为。

### Note

如果想在屏幕上看到被读入的行，必须把变量ECHO设置成all。

```
\if expression
\elif expression
\else
\endif
```

这一组命令实现可嵌套的条件块。条件块必须以一个\if开始并且以一个\endif结束。两者之间可能有任意数量的\elif子句，后面也可能有选择地跟着一个单一的\else子句。一般查询以及其他类型的反斜线命令可以出现在这些命令之间构成条件块。

\if和\elif命令读取它们的参数并且将它们作为布尔表达式进行计算。如果表达式得到真则处理正常继续下去，否则会跳过下面的行直到到达一个匹配的\elif、\else或者\endif。一旦一个\if或者\elif测试成功，同一个块中后面的\elif命令的参数将不会被计算但会被当作假。跟在一个\else后面的行只有在先前的匹配的\if或\elif成功时才被处理。

就像任何其他反斜线命令参数一样，\if或者\elif命令的expression参数服从变量篡改以及反引号展开。然后会像一个on/off选项变量的值一样来计算它。因此，对下列项无歧义、大小写无关的匹配都是有效的值：true、false、1、0、on、off、yes、no。例如，t、T以及tR都将被认为是真。

无法被正确计算为真或假的表达式将产生一个警告并且被当做假。

正在被跳过的行还是会被正常地解析以标识查询和反斜线命令，但是查询不会被发送到服务器，并且非条件（\if、\elif、\else、\endif）反斜线命令会被忽略。条件命令会被检查以判断嵌套是否有效。被跳过的行中的变量引用不会被展开，并且也不会执行反引号展开。

一个给定条件块中的所有反斜线命令必须出现在相同的源文件中。如果在所有的本地\if块被关闭之前，主输入文件或者一个\include进来的文件上就达到了EOF，则ltsql将产生一个错误。

这里是一个例子：

```
-- 检查数据库中两个单独记录的存在性并且把结果存在单独的psql变量中
SELECT
    EXISTS(SELECT 1 FROM customer WHERE customer_id = 123) as is_customer,
    EXISTS(SELECT 1 FROM employee WHERE employee_id = 456) as is_employee
\gset
\if :is_customer
    SELECT * FROM customer WHERE customer_id = 123;
\elif :is_employee
    \echo 'is not a customer but is an employee'
    SELECT * FROM employee WHERE employee_id = 456;
\else
    \if yes
        \echo 'not a customer or employee'
    \else
        \echo 'this will never print'
    \endif
```

`\endif`

`\ir` or `\include_relative filename`

`\ir`命令类似于`\i`，但是以不同的方式处理相对路径文件名。在交互模式中执行时，这两个命令的行为相同。不过，当被从脚本中调用时，`\ir`相对于脚本所在的目录而不是根据当前工作目录来解释文件名。

`\l[+]` or `\list[+] [ pattern ]`

列出服务器中的数据库并且显示它们的名称、拥有者、字符集编码以及访问特权。如果指定了`pattern`，则只列出名称匹配该模式的数据库。如果向命令名称追加`+`，则还会显示数据库的尺寸、默认表空间以及描述（尺寸信息只对当前用户能连接的数据库可用）。

`\lo_export loid filename`

从数据库中读取具有OID `loid`的大对象并且将它写入到`filename`。注意这和服务器函数`lo_export`有微妙的不同，后者会以运行数据库服务器的用户权限来执行并且运行在服务器的文件系统上。

### Tip

使用`\lo_list`可以找出大对象的OID。

`\lo_import filename [ comment ]`

把该文件存储到LightDB大对象。可选地，它可以把给定的注释关联到该对象。例如：

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

该响应表示该大对象得到的对象 ID 是 152801，未来可以用这个 ID 来访问这个新创建的大对象。为了便于阅读，推荐总是给每一个对象都关联人类可读的注释。OID 和注释都可以用`\lo_list`命令查看。

注意这个命令和服务器端的`lo_import`有微妙的不同，因为它以本地文件系统上的本地用户的身份运行，而不是服务器用户和文件系统。

`\lo_list`

显示当前存储在数据库中的所有LightDB大对象，同时显示它们的任何注释。

`\lo_unlink loid`

从数据库中删除OID为`loid`的大对象。

### Tip

使用`\lo_list`可以找出该大对象的OID。

`\o` or `\out [ filename ]`

`\o` or `\out [ |command ]`

安排把未来的查询结果保存到文件`filename`中或者用管道导向到 `shell` 命令`command`。如果没有指定参数，查询输出会被重置到标准输出。



如果该参数以|开始，则该行的所有剩余部分总是会被当做要执行的command，并且在参数中不会执行变量篡改以及反引号展开。该行的剩余部分会被简单地按字面传给shell。

“查询结果”包括从数据库服务器得到的所有表、命令响应和提示，还有查询数据库的各种反斜线命令（如\d）的输出，但不包括错误消息。

### Tip

要在查询结果之间混入文本输出，可以使用\qecho。

\p or \print

把当前查询缓冲区打印到标准输出。如果当前查询缓冲区为空，会打印最近被执行的查询。

\password [ username ]

更改指定用户（默认情况下是当前用户）的口令。这个命令会提示要求输入新口令、对口令加密然后把加密后的口令作为一个ALTER ROLE命令发送到服务器。这确保新口令不会以明文的形式出现在命令历史、服务器日志或者其他地方。

\prompt [ text ] name

提示用户提供一个文本用于分配给变量name。可以指定一个可选的提示字符串text（对于多个词组成的提示，把文本包裹在单引号中）。

默认情况下，\prompt使用终端进行输入和输出。不过，如果使用了-f命令行开关，\prompt会使用标准输入和标准输出。

\pset [ option [ value ] ]

这个命令设置影响查询结果表输出的选项。option表示要设置哪个选项。value的语义取决于选中的选项。对于某些选项，如果省略value会导致该选项值被切换或者被重置，具体是哪些选项可见特定选项的描述。如果没有上面提到的那种行为，那么省略value只会导致当前设置被显示。

不带任何参数的\pset显示所有打印选项的当前状态。

可调整的打印选项有：

border

value必须是一个数字。通常，数字越大，表格就会有更多的边框和线条，但具体要看是哪一种格式。在HTML格式中，这会直接被转换成border=...属性。在大部分其他格式中，只有值 0（没有边框）、1（内部分隔线）和 2（表格边框）有意义，并且 2 以上的值会被视为与border = 2相同。latex和latex-longtable格式会额外地允许一个值 3 表示在数据行之间增加分隔线。

columns

为wrapped格式设置目标宽度，还有扩展自动模式中决定输出是否足够多到需要分页器或者切换到垂直显示的宽度限制。零（默认）导致目标宽度由环境变量COLUMNS所控制，如果没有设置COLUMNS则使用检测到的屏幕宽度。此外，如果columns为零则wrapped格式只影响屏幕输出。如果columns为非零则文件和管道输出也会被包裹成该宽度。

csv\_fieldsep

规定要用于CSV输出格式的字段分隔符。如果分隔符出现在字段的值中，则该字段遵循标准CSV规则在双引号中输出。默认值为逗号。

expanded (or x)

如果value被指定，它必须是on或者off，它们分别会启用或者禁用扩展模式，也可以是auto。如果value被省略，则该命令会在开启和关闭设置之间切换。当扩展模式被启用时，查询结果被显示在两列中，第一列是列名而第二列是列值。如果在通常的“水平”模式中数据不适合屏幕，则可以用这种模式。在自动设置中，只要查询输出有多于一列并且比屏幕宽，就会使用扩展模式。否则，将使用常规模式。只有在对齐格式和 wrapped 格式中自动设置才有效。在其他格式中，它的行为总是像扩展模式被关闭一样。

fieldsep

指定在非对齐输出格式中使用的域分隔符。用那种方式，用户可以创建 tab 分隔的输出，这种形式其他程序可能更喜欢。要设置 tab 为域分隔符，可以键入\pset fieldsep 't'。默认的域分隔符是'|'（一个竖线）。

fieldsep\_zero

把用在非对齐输出格式中的域分隔符设置为一个零字节。

footer

如果value被指定，它必须是on或者off，它们分别会启用或者禁用表格页脚（(n rows)计数）的显示。如果value被省略，则该命令会切换页脚显示为打开或者关闭。

format

设置输出格式为下列的一种：aligned, asciidoc, csv, html, latex, latex-longtable, troff-ms, unaligned, 或 wrapped。允许唯一的缩写。

aligned 格式是标准,人类可阅读的、良好格式化的文本输出；这是默认值。

unaligned格式把一个数据行的所有列都写在一行上，之间用当前活动的域分隔符分隔。这可用于生成意图由其他程序读取的输出，例如，tab 分隔或者逗号分隔格式。但是，如果字段分隔符出现在列的值中，则不专门处理该字符；因此CSV格式可能更适合此类目的。

csv 格式 写入以逗号分隔的列值，应用引号规则描述在RFC 4180<sup>1</sup>。此输出与服务器的COPY命令的 CSV 格式兼容。除非tuples\_only参数为on，否则将生成具有列名称的标头行。不打印标题和页脚。每行都由系统相关的行尾字符结束，该字符通常是类似 Unix 的系统的单一换行符（\n）或应用于微软Windows的回车和换行顺序（\r\n）。除了逗号之外的\pset csv\_fieldsep字段分隔符，还可以使用\pset csv\_fieldsep选择。

wrapped格式和aligned相似，但是前者会把过宽的数据值分成多个行以便输出能够适合目标行的宽度。目标行的宽度由columns选项决定。注意ltsql将不会尝试对列头部标题进行换行，因此如果列头部需要的总宽度超过目标宽度，wrapped格式的行为就变得和aligned一样了。

The asciidoc, html, latex, latex-longtable, 和troff-ms 格式分别用相应的标记语言把要输出的表格放在文档中。不过它们的输出并不是完整的文档。这在HTML中也许不重要，但是在LaTeX中必须有完整的文档包装器。latex 格式使用 LaTeX的 tabular环境。latex-longtable 格式需要 LaTeX、longtable 和 booktabs 包。

linestyle

设置边框线的绘制样式为ascii、old-ascii或者unicode之一。允许不产生歧义的缩写（这意味着一个字母就足够了）。默认的设置是ascii。这个选项只影响aligned以及wrapped输出格式。

<sup>1</sup> <https://tools.ietf.org/html/rfc4180>

ascii样式使用纯ASCII字符。数据中的新行使用一个+符号在右手边的空白处显示。当在wrapped格式中包裹两行中间没有新行字符的数据时，会在第一行右手边空白处显示一个点号（.），并且在下一行的左手边空白处也显示一个点号（.）。

old-ascii样式使用纯ASCII字符，使用LightDB 8.4 及更早版本中用过的格式化样式。数据中的新行使用:符号来代替左手边的列分隔符显示。在包裹两行中间没有新行字符的数据时，会用一个;符号取代左手边的列分隔符。

unicode样式使用 Unicode 的方框绘制字符。数据中的新行会使用一个回车符号显示在右手边的空白处。在包裹两行中间没有新行字符的数据时，会在第一行的右手边空白处显示一个省略号，并且在下一行的左手边空白处也显示一个省略号。

当border设置大于零时，linestyle选项也决定边框线用什么字符绘制。纯ASCII字符到处都可以使用，但是在识别 Unicode 字符的显示上使用 Unicode 字符会更好看。

#### null

设置要用来替代空值被打印的字符串。默认是什么也不打印，对于一个空字符串这很容易弄错。例如，有人可能更想用\pset null '(null)'。

#### numericlocale

如果value被指定，它必须是on或者off，它们将分别启用或者禁用一个与区域相关的字符来分隔数字和左边的十进制标记。如果value被省略，该命令会在常规输出和区域相关的数字输出之间切换。

#### pager

控制对查询和ltsql的帮助输出使用分页器程序。如果环境变量PSQL\_PAGER或PAGER被设置，输出会被用管道输送到指定的程序。否则将使用与平台相关的默认分页器程序（例如more）。

如果pager选项被设为off，则不会使用分页器程序。如果pager选项被设为on，则会在适当的时候使用分页器，即当输出到终端并且无法适合屏幕时就会使用分页器。pager选项也可以被设置为always，这会导致对所有的终端输出都是用分页器而不管输出是否适合屏幕。不带value的\pset pager会切换分页器开、关状态。

#### pager\_min\_lines

如果pager\_min\_lines被设置为一个大于页面高度的数字，在至少这么多输出行被显示之前都不会调用分页器程序。默认设置为 0。

#### recordsep

指定用在非对齐输出格式中的记录（行）分隔符。

#### recordsep\_zero

把用在非对齐输出格式中的记录分隔符设置为一个零字节。

#### tableattr (or T)

在HTML格式中，这会指定要放在table标记内的属性。例如，这可能是cellpadding或者bgcolor。注意你可能不想在这里指定border，因为那由\pset border负责。如果没有给出value，则表属性会被重置。

在latex-longtable格式中，这个选项控制每个包含左对齐数据类型的列的宽度比例。这个选项的值是一个由空格分隔的值列表，例如'0.2 0.2 0.6'。没有指定的输出列会使用最后一个指定的值。

title (or C)

设置用于任何后续被打印表的表标题。这可以用来给输出加上描述性的标签。如果没有给出value，这个标题会被复原。

tuples\_only (or t)

如果value被指定，它必须是on或者off，这个选项将启用或者禁用只显示元组的模式。如果value被省略，则该命令会在常规输出和只显示元组输出之间切换。常规输出包括列头、标题以及多种页脚之类的额外信息。在只显示元组的模式中，只会显示实际的表数据。

unicode\_border\_linestyle

设置unicode线型的边框绘制风格为single或者double之一。

unicode\_column\_linestyle

设置unicode线型的列绘制风格为single或者double之一。

unicode\_header\_linestyle

设置unicode线型的页眉绘制风格为single或者double之一。

这些不同格式的外观可以在Examples小节的图示中看到。

### Tip

\pset有多种快捷命令。请参见\a、\C、\f、\H、\t、\T以及\x。

\q or \quit

退出ltsql程序。在一个脚本文件中，只有该脚本的执行会被终止。

\qecho text [ ... ]

这个命令和\echo一样，不过输出将被写到\o所设置的查询输出通道。

\r or \reset

重置（清除）查询缓冲区。

\s [ filename ]

打印ltsql的命令行历史到filename。如果省略filename，该历史会被写入到标准输出（如果适用则使用分页器）。如果编译ltsql时没有加上Readline支持，则这个命令不可用。

\set [ name [ value [ ... ] ] ]

设置ltsql变量name为value，如果给出了多于一个值，则把该变量的值设置为所有给出的值的串接。如果只给了一个参数，该变量会被设置为空字符串值。要重置一个变量，可以使用\unset 命令。

不带任何参数的\set显示所有当前设置的ltsql变量的名称和值。

合法的变量名可以包含字母、数字和下划线。详见下文的Variables。变量名是大小写敏感的。

某些变量是特殊的，它们控制ltsql的行为或者会被自动设置以反映连接状态。这些变量在下文的Variables中记录。

## Note

这个命令和SQL命令SET无关。

`\setenv name [ value ]`

把环境变量name设置为value，如果没有提供value，则会重置该环境变量。例如：

```
test=> \setenv PAGER less
test=> \setenv LESS -imx4F
```

`\sf[+] function_description`

这个命令以一个CREATE OR REPLACE FUNCTION命令或者CREATE OR REPLACE PROCEDURE命令取出并且显示指定函数或者过程的定义。定义会被打印到当前的查询输出渠道，就像\o所作的那样。

目标函数可以单独用名称指定，也可以用名称和参数指定，例如foo(integer, text)。如果有多于一个函数具有相同的名字，则必须给出参数的类型。

如果向命令名称追加+，那么输出行会被编号，函数体的第一行会被编为 1。

与大部分其他元命令不同，该行的所有剩余部分总是会被当做\sf的参数，并且在参数中不会执行变量篡改以及反引号展开。

`\sv[+] view_name`

这个命令以一个CREATE OR REPLACE VIEW命令取出并且显示指定视图的定义。定义会被打印到当前的查询输出渠道，就像\o所作的那样。

如果在命令名称上追加+，那么输出行会从 1 开始编号。

与大部分其他元命令不同，该行的所有剩余部分总是会被当做\sv的参数，并且在参数中不会执行变量篡改以及反引号展开。

`\t`

切换输出列名标题和行计数页脚的显示。这个命令等效于\pset tuples\_only，提供它只是为了使用方便而已。

`\T table_options`

指定在HTML输出格式中，要放在table标签内的属性。这个命令等效于\pset tableattr table\_options。

`\timing [ on | off ]`

如果给出一个参数，这个参数用来打开或者关闭对每个SQL语句执行时长的显示。如果没有参数，则在打开和关闭之间切换。显示的数据以毫秒为单位，超过1秒的区间还会被显示为“分钟:秒”的格式，如果必要还会加上小时和日的字段。

`\unset name`

重置（删除）ltsql变量name。

大部分控制ltsql行为的变量不能被重置，相反，\unset命令会被解释为把它们设置为其默认值。请参考下文的Variables。

```
\w or \write filename
\w or \write |command
```

把当前查询缓冲区写到文件filename或者用管道导出到 shell 命令command。如果当前查询缓冲区为空，则写最近被执行的查询。

如果参数以|开始，则该行的整个剩余部分会被当做要执行的command，并且在参数中不会执行变量篡改以及反引号展开。该行的剩余部分会被简单地按字面传递给shell。

```
\warn text [ ... ]
```

此命令与 \echo 相同，只是输出将写入 ltsql 的标准错误通道，而不是标准输出。

```
\watch [ seconds ]
```

反复执行当前的查询缓冲区（就像\g那样）直到被中止或者查询失败。两次执行之间等待指定的秒数（默认是 2 秒）。显示每个查询结果时带上一个由\pset title字符串（如果有）、从查询开始起的时间以及延时间隔组成的页眉。

如果当前查询缓冲区为空，则会重新执行最近被发送的查询。

```
\x [ on | off | auto ]
```

设置或者切换扩展表格格式化模式。究其本身而言，这个命令等效于\pset expanded。

```
\z [ pattern ]
```

列表、视图和序列，以及它们相关的访问特权。如果指定了pattern，则只会列出名称匹配该模式的表、视图和序列。

这是\dp（“display privileges”）的一个别名。

```
\! [ command ]
```

如果没有参数，就跳出到一个子shell，当子shell退出时ltsql会继续。如果有一个参数，则执行shell命令command。

与大部分其他元命令不同，该行的所有剩余部分总是会被当做\!的参数，并且在参数中不会执行变量篡改以及反引号展开。该行的剩余部分会被简单地按字面传递给shell。

```
\? [ topic ]
```

显示帮助信息。可选的topic参数（默认是commands）选择解释ltsql的哪一部分：commands表示ltsql的反斜线命令；options表示可以传递给ltsql的命令选项；而variables显示有关ltsql配置变量的帮助。

```
\;
```

反斜线分号并非和前述命令相同的元命令，它只是会把一个分号加入到查询缓冲区且不会进一步执行。

通常，只要ltsql达到了命令结束的分号，它就将分发一个SQL命令给服务器，即使在当前行上还留有更多输入。因此，例如输入

```
select 1; select 2; select 3;
```

将导致三个SQL命令被逐个发送给服务器，在继续到下一个命令前会显示每一个命令的结果。不过，被输入为\;的分号将不会触发命令处理，这样在它之前的命令以及其后的命令实际上会被组合在一个请求中发送给服务器。例如

```
select 1\; select 2\; select 3;
```

会导致在到达非反斜线分号时用一个单一的请求把三个SQL命令发送给服务器。服务器会把这样一个请求当作单一的事务执行，除非该字符串中有显式的BEGIN/COMMIT命令把它划分成多个事务（服务器如何处理多查询字符串的更多细节请参考Section 49.2.2.1）。ltsql对每个请求仅打印出它接收到的最后一个查询结果。在这个例子中，尽管所有三个SELECT确实都被执行了，但ltsql只会打印出3。

## 模式

很多\dt命令都可以用一个pattern参数来指定要被显示的对象名称。在最简单的情况下，模式正好就是该对象的准确名称。在模式中的字符通常会被变成小写形式（就像在SQL名称中那样），例如\dt FOO将会显示名为foo的表。就像在SQL名称中那样，把模式放在双引号中可以阻止它被转换成小写形式。如果需要在模式中包含一个真正的双引号字符，则需要把它写成两个相邻的双引号，这同样是符合SQL引用标识符的规则。例如，\dt "FOO" "BAR"将显示名为FOO"BAR（不是foo"bar）的表。和普通的SQL名称规则不同，你不能只在模式的一部分周围放上双引号，例如\dt FOO"FOO"BAR将会显示名为fooFOObar的表。

只要pattern参数被完全省略，\dt命令会显示在当前 schema 搜索路径中可见的全部对象——这等价于用\*作为模式（如果一个对象所在的 schema 位于搜索路径中并且没有同类且同名的对象出现在搜索路径中该 schema 之前的 schema 中，则说该对象是可见的。这表示可以直接用名称引用该对象，而不需要用 schema 来进行限定）。要查看数据库中所有的对象而不管它们的可见性，可以把\*. \*用作模式。

如果放在一个模式中，\*将匹配任意字符序列（包括空序列），而?会匹配任意的单个字符（这种记号方法就像Unix shell的文件名模式一样）。例如，\dt int\*会显示名称以int开始的表。但是如果被放在双引号内，\*和?就会失去这些特殊含义而变成普通的字符。

包含一个点号(.)的模式被解释为一个 schema 名称模式后面跟上一个对象名称模式。例如，\dt foo\*. \*bar\*会显示名称以foo开始的 schema 中所有名称包括bar的表。如果没有出现点号，那么模式将只匹配当前 schema 搜索路径中可见的对象。同样，双引号内的点号会失去其特殊含义并且变成普通的字符。

高级用户可以使用字符类等正则表达式记法，如[0-9]可以匹配任意数字。所有的正则表达式特殊字符都按照Section 9.7.1所说的工作，以下字符除外：.会按照上面所说的作为一种分隔符，\*会被翻译成正则表达式记号\*，?会被翻译成.，而\$则按字面意思匹配。根据需要，可以通过书写?(R+|)、(R|)和R?来分别模拟模式字符.、R\*和R?。\$不需要作为一个正则表达式字符，因为模式必须匹配整个名称，而不是像正则表达式的常规用法那样解释（换句话说，\$会被自动地追加到模式上）。如果不希望该模式的匹配位置被固定，可以在开头或者结尾写上\*。注意在双引号内，所有的正则表达式特殊字符会失去其特殊含义并且按照其字面意思进行匹配。还有，在操作符名称模式中（即作为\do的参数），正则表达式特殊字符也按照字面意思进行匹配。

## 高级特性

### 变量

ltsql提供了和普通Unix命令shell相似的变量替换特性。变量简单来说就是一对名称/值，其中值可以是任意长度的任意字符串。名称必须由字母（包括非拉丁字母）、数字和下划线构成。

要设置一个变量，可以使用ltsql的元命令\set。例如，

```
test=> \set foo bar
```

会设置foo为值bar。要检索该变量的内容，可以在名称前放一个分号，例如：

```
test=> \echo :foo
bar
```

这在常规 SQL 命令和元命令中均有效，下文的SQL Interpolation中有更多细节。

如果调用\set时没有第二个参数，该变量会被设置为一个空字符串值。要重置（即删除）一个变量，可以使用命令\unset。要显示所有变量的值，在调用\set时不带任何参数即可。

### Note

\set的参数服从与其他命令相同的替换规则。因此可以构造有趣的引用，例如\set :foo 'something' 以及分别得到Perl或者PHP的“软链接”或者“可变量”。不幸的是（或者幸运的是？），这些构造出来的东西并没有什么用处。在另一方面，\set bar :foo是一种很好的拷贝变量的方法。

有一些变量会被ltsql特殊对待。它们表示特定的选项设置，运行时这类选项设置可以通过修改该变量的值来改变，或者在某些情况下它们表示ltsql的可更改的状态。按照惯例，所有被特殊对待的变量的名称由全部大写形式的 ASCII 字母（还有可能是数字和下划线）组成。为了确保未来最大的兼容性，最好避免把这类变量名用于自己的目的。

控制ltsql行为的变量通常不能被重置或者设置为无效值。允许\unset命令，但它会被解释为将变量设置为它的默认值。没有第二参数的\set命令会被解释为将变量设置为on（对于接受该值的控制变量），对不接受该值的变量则会拒绝这个命令。此外，接受值on和off的控制变量也能接受其他常见的布尔值拼写方式，例如true和false。

被特殊对待的变量是：

#### AUTOCOMMIT

在被设置为on（默认）时，每一个 SQL 命令在成功完成时会被自动提交。在这种模式中要推迟提交，必须输入一个BEGIN或者START TRANSACTION SQL 命令。当被设置为off或者被重置时，在显式发出COMMIT或者END之前，SQL 命令不会被提交。自动提交打开模式会为你发出一个隐式的BEGIN，这会发生在任何不在一个事务块中且本身即不是BEGIN及其他事务控制命令且不是无法在事务块中执行的命令（例如VACUUM）之前。

### Note

在自动提交关闭模式中，必须通过ABORT或者ROLLBACK显式地放弃任何失败的事务。还要记住，如果退出会话时没有提交，则所有的工作都会丢失。

### Note

自动提交打开模式是LightDB的传统行为，但是自动提交关闭模式更接近于 SQL 的规范。如果更喜欢自动提交关闭模式，可以在系统级的ltsqlrc文件或者个人的~/.ltsqlrc文件中设置它。

#### COMP\_KEYWORD\_CASE

确定在补全一个 SQL 关键词时要使用的大小写形式。如果被设置为lower或者upper，补全后的词将分别是小写或者大写形式。如果被设置为preserve-lower或者preserve-upper（默认），补全后的词将会保持该词已输入部分的大小写形式，但是如果被补全的词还没有被输入，则它会被分别补全成小写或者大写形式。

#### DBNAME

当前已连接的数据库名称。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者重置。



## ECHO

如果被设置为all，所有非空输入行会被按照读入它们的样子打印到标准输出（不适用于交互式读取的行）。要在程序开始时选择这种行为，可以使用开关-a。如果被设置为queries，ltsql会在发送每个查询给服务器时将它们打印到标准输出。选择这种行为的开关是-e。如果被设置为errors，那么只有失败的查询会被显示在标准错误输出上。这种行为的开关是-b。如果被重置或者设置为none（默认值）则不会显示任何查询。

## ECHO\_HIDDEN

当这个变量被设置为on且一个反斜线命令查询数据库时，相应的查询会被先显示。这种特性可以帮助我们学习LightDB的内部并且在自己的程序中提供类似的功能（要在程序开始时选择这种行为，可以使用开关-E）。如果把把这个变量设置为值noexec，则对应的查询只会被显示而并不真正被发送给服务器执行。默认值是off。

## ENCODING

当前的客户端字符集编码。每一次你连接到一个数据库（包括程序启动）时以及当你用\encoding更改编码时，这个变量都会被设置，但它可以被更改或者重置。

## ERROR

如果上一个SQL查询失败则为true，如果成功则是false。另见SQLSTATE。

## FETCH\_COUNT

如果这个变量被设置为一个大于零的整数值，SELECT查询的结果会以一组一组的方式取出并且显示（而不是像默认的那样把整个结果集拿到以后再显示），每一组就会包括这么多个行。因此，这种方式只会使用有限的内存量，而不管整个结果集的大小。在启用这个特性时，通常会使用 100 到 1000 的设置。记住在使用这种特性时，一个查询可能会在已经显示了一些行之后失败。

## Tip

尽管可以把这种特性用于任何的输出格式，但是默认的aligned格式看起来会比较糟糕，因为每一组的FETCH\_COUNT个行将被单独格式化，这就会导致不同的行组的列宽不同。其他的输出格式会更好。

## HIDE\_TABLEAM

如果此变量设置为true，则不显示表的访问方法的详细信息。这主要用于回归测试。

## HISTCONTROL

如果这个变量被设置为ignorespace，则以一个空格开始的行不会被放入到历史列表中。如果被设置为值ignoredups，则匹配之前的历史行的行不会被放入。值ignoreboth组合了上述两种值。如果被重置或者被设置为none（默认值），所有在交互模式中被读入的行都会保存在历史列表中。

## Note

这个特性是可耻地从Bash抄袭过来的。

## HISTFILE

该文件名将被用于存储历史列表。如果被重设，文件名将从PSQL\_HISTORY环境变量中取得。如果该环境变量也没有被设置，则默认值是~/.psql\_history，在Windows上是%APPDATA%\postgresql\psql\_history。例如，

```
\set HISTFILE ~/.psql_history- :DBNAME
```

放在`~/.ltsqlrc`中将会导致ltsql为每一个数据库维护一个单独的历史。

#### Note

这个特性是可耻地从Bash抄袭过来的。

#### HISTSIZE

存储在命令历史中的最大命令数（默认值是500）。如果被设置为一个负值，则不会应用限制。

#### Note

这个特性是可耻地从Bash抄袭过来的。

#### HOST

当前连接到的数据库服务器端口。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者重置。

#### IGNOREEOF

如果被设置为1或者更小，向一个ltsql的交互式会话发送一个EOF字符（通常是Control+D）将会终止应用。如果设置为一个较大的数字值，则必须键入多个连续的EOF字符才能让交互式会话终止。如果该变量被设置为一个非数字值，则它会被解释为10。默认值为0。

#### Note

这个特性是可耻地从Bash抄袭过来的。

#### LASTOID

最后被影响的OID的值，这可能会由INSERT或者`\lo_import`命令返回。这个变量只保证在下一个SQL命令被显示完之前有效。LightDB服务器从12版开始不再支持OID系统列，因此，在面向此类服务器时，跟随在INSERT后面的LASTOID将始终为0。

#### LAST\_ERROR\_MESSAGE

#### LAST\_ERROR\_SQLSTATE

当前ltsql会话中最近一个失败查询的主错误消息和相关的SQLSTATE代码，如果在当前会话中没有发生错误，则是一个空字符串和00000。

#### ON\_ERROR\_ROLLBACK

当被设置为on时，如果事务块中的一个语句产生一个错误，该错误会被忽略并且该事务会继续。当被设置为interactive时，只在交互式会话中忽略这类错误，而读取脚本文件时则不会忽略错误。当被重置或者设置为off（默认值）时，事务块中产生错误的一个语句会中止整个事务。错误回滚模式的工作原理是在事务块的每个命令之前都为你发出一个隐式的SAVEPOINT，然后在命令失败时回滚到该保存点。

---

## ON\_ERROR\_STOP

默认情况下，出现一个错误后命令处理会继续下去。当这个变量被设置为on后，出现错误后命令处理会立即停止。在交互模式下，ltsql将会返回到命令提示符；否则，ltsql将会退出并且返回错误代码 3 来把这种情况与致命错误区分开来，致命错误会被报告为错误代码 1。在两种情况下，任何当前正在运行的脚本（顶层脚本以及任何它已经调用的其他脚本）将被立即中止。如果顶层命名字符串包含多个 SQL 命令，将在当前命令处停止处理。

## PORT

当前连接到的数据库服务器端口。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者重置。

## PROMPT1

## PROMPT2

## PROMPT3

这些变量指定ltsql发出的提示符的模样。见下文的Prompting。

## QUIET

把这个变量设置为on等效于命令行选项-q。在交互模式下可能用处不大。

## ROW\_COUNT

上一个SQL查询返回的行数或者受影响的行数，如果该查询失败或者没有报告行计数则为0。

## SERVER\_VERSION\_NAME

## SERVER\_VERSION\_NUM

字符串形式的服务器版本号，例如9.6.2、10.1或者11beta1，以及数字形式的服务器版本号，例如90602或者100001。每次你连接到一个数据库（包括程序启动）时，这些都会被设置，但可以被改变或者重设。

## SHOW\_CONTEXT

这个变量可以被设置为值never、errors或者always来控制是否在来自服务器的消息中显示CONTEXT域。默认是errors（表示在错误消息中显示上下文，但在通知和警告消息中不显示）。当VERBOSITY被设置为terse或sqlstate时，这个设置无效（另见\errverbose，它可以用来得到刚遇到的错误的详细信息）。

## SINGLELINE

设置这个变量为on等效于命令行选项-S。

## SINGLESTEP

设置这个变量为on等效于命令选项-s。

## SQLSTATE

与上一个SQL查询的失败相关的错误代码（见Appendix 0，如果上一个查询成功则为00000）。

## USER

当前连接的数据库用户。每次连接到一个数据库时都会设置该变量（包括程序启动时），但是可以被更改或者重置。

## VERBOSITY

这个变量可以被设置为值default、verbose、terse或者sqlstate来控制错误报告的详细程度（另见\errverbose，在想得到之前的错误的详细版本时使用）。

## VERSION

VERSION\_NAME

VERSION\_NUM

这些变量在程序启动时被设置以反映ltsql的版本，分别是一个详细的字符串、一个短字符串（例如9.6.2、10.1或者11beta1）以及一个数字（例如90602或者100001）。它们可以被更改或重设。

## SQL 中插入变量

ltsql变量的一个关键特性是可以把它们替换（“插入”）到常规SQL语句中，也可以把它们作为元命令的参数。此外，ltsql还提供了功能来确保被用作 SQL 文字和标识符的变量值会被正确地引用。插入一个值而不需要加引用的语法是在变量名前面加上一个冒号（:）。例如，

```
test=> \set foo 'my_table'
test=> SELECT * FROM :foo;
```

将查询表my\_table。注意这可能会不安全：该变量的值会被按字面拷贝，因此它可能包含不平衡的引号甚至反斜线命令。必须确保把它放在那里是有意义的。

当一个值被用作 SQL 文本或者标识符时，最安全的是把它加上引用。要引用一个变量的值作为 SQL 文本，可以把变量名称放在单引号中并且在引号前面写一个冒号。要引用作为SQL标识符，则可以把变量名称放在双引号中并且在引号前面写一个冒号。这种结构可以正确地处理变量值中嵌入的引号和其他特殊字符。之前的例子用这种方法写会更安全：

```
test=> \set foo 'my_table'
test=> SELECT * FROM :"foo";
```

在被引用的SQL文本和标识符中将不会执行变量插入。因此，一个诸如':foo'的结构不会从一个变量的值产生一个被引用的文本（即便能够也会不安全，因为无法正确地处理嵌入在值中的引号）。

使用这种机制的一个例子是把一个文件的内容拷贝到一个表列中。首先把该文件载入到一个变量，然后把该变量的值作为一个被引用的字符串插入：

```
test=> \set content `cat my_file.txt`
test=> INSERT INTO my_table VALUES (: 'content');
```

（注意如果my\_file.txt包含 NUL 字节，这样也不行。ltsql不支持在变量值中嵌入 NUL 字节）。

因为冒号可以合法地出现在 SQL 命令中，一次明显的插入尝试（即:name、:'name'或者:"name"）不会被替换，除非所提及的变量就是当前被设置的。在任何情况下，可以用一个反斜线对冒号进行转义以避免它被替换。

:{?name}特殊语法根据该变量存在与否返回TRUE或者FALSE，并且因此总是会被替换，除非分号被反斜线转义。

变量的冒号语法对嵌入式查询语言（例如ECPG）来说是标准的SQL。用于数组切片和类型造型的冒号语法是LightDB扩展，它有时可能会与标准用法冲突。把一个变量值转义成 SQL 文本或者标识符的冒号引用语法是一种ltsql扩展。

## 提示符

ltsql发出的提示符可以根据用户的喜好自定义。PROMPT1、PROMPT2和PROMPT3这三个变量包含了描述提示符外观的字符串和特殊转义序列。Prompt 1 是当ltsql等待新命令时发出的常规提示符。Prompt 2 是在命令输入时需要更多输入时发出的提示符，例如因为当命令没有被分号终止或者引用没有被关闭时就会发出这个提示符。在运行一个SQL COPY FROM STDIN命令并且需要在终端上输入一个行值时，会发出 Prompt 3。

被选中的提示符变量会被原样打印，除非碰到一个百分号（%）。百分号的下一个字符会被特定的其他文本替换。预定义好的替换有：

%M

数据库服务器的完整主机名（带有域名），或者当该连接是建立在一个 Unix 域套接字上时则是[local]，或者当 Unix 域套接字不在编译在系统内的默认位置上时则是[local:/dir/name]。

%m

数据库服务器的主机名称（在第一个点处截断），或者当连接建立在一个 Unix 域套接字上时是[local]。

%>

数据库服务器正在监听的端口号。

%n

数据库会话的用户名（在数据库会话期间，这个值可能会因为命令SET SESSION AUTHORIZATION的结果而改变）。

%/

当前数据库的名称。

%~

和%/类似，但是如果数据库是默认数据库时输出是~（波浪线）。

%#

如果会话用户是一个数据库超级用户，则是#，否则是一个>（在数据库会话期间，这个值可能会因为命令SET SESSION AUTHORIZATION的结果而改变）。

%p

当前连接到的后端的进程 ID。

%R

在提示符1下通常是=，但如果会话位于一个条件块的一个非活动分支中则是@，如果会话处于单行模式中则是^，如果会话从数据库断开连接（\connect失败时会发生这种情况）则是!。在提示符 2 中，根据为什么ltsql期待更多的输入，%R会被一个相应的字符替换：如果命令还没有被终止是-，如果有一个未完的/\* ... \*/注释则是\*，如果有一个未完的被引用字符串则是一个单引号，如果有一个未完的被引用标识符则是一个双引号，如果有一个未完的美元引用字符串则是一个美元符号，如果有一个还没有被配对的左圆括号则是(。在提示符 3 中%R不会产生任何东西。

%x

事务状态：当不在事务块中时是一个空字符串，在一个事务块中时是\*，在一个失败的事务块中时是!，当事务状态是未判定时（例如因为没有连接）为?。

%l

当前语句中的行号，从1开始。

%digits

带有指定的八进制码的字符会被替换。

:%name:

ltsql变量name的值。有关详细信息，请参阅上面的 Variables。

%`command`

command的输出，类似于平常的“反引号”替换。

%[ ... %]

提示符可以包含终端控制字符，例如改变提示符文本的颜色、背景或者风格以及更改终端窗口标题的控制字符。为了让Readline的行编辑特性正确工作，这些不可打印的控制字符必须被包裹在%[和%]之间以指定它们是不可见的。在提示符中可以出现多个这样的标识对。例如：

```
test=> \set PROMPT1 '%[%033[1;33;40m%]n@%/%R%[%033[0m%]## '
```

会导致一个在兼容 VT100 的彩色终端上的粗体（1;）的、黑底黄字（33;40）的提示符。

%w

与 PROMPT1 的最近输出相同宽度的空白。这可以用作 PROMPT2 设置，以便多行语句与第一行对齐，但没有可见的辅助提示。

要在你的提示符中插入一个百分号，可以写成%%。提示符 1 和 2 的默认提示是'/%R%x%# '，提示符 3 的提示是'>>'。

### Note

这个特性是可耻地从tcsh抄袭过来的。

## 命令行编辑

为了方便的行编辑和检索，ltsql支持Readline库。ltsql退出时命令历史会被自动保存，而当ltsql启动时命令历史会被重新载入。ltsql也支持 tab 补全，不过补全逻辑绝不是一个SQL解析器。tab 补全产生的查询也可能会受其他 SQL 命令干扰，例如SET TRANSACTION ISOLATION LEVEL。如果出于某种原因不想用 tab 键补全，可以把下面的代码放在主目录下的名为.inputrc文件中关闭该特性：

```
$if ltsql
set disable-completion on
$endif
```

（这不是ltsql特性而是Readline的特性。进一步的细节请阅读它的文档。）

## set define on/off 命令

set define on/off命令来源于oracle的sqlplus工具。set define on/off命令目前只支持oracle模式的SELECT语句。您可以在SELECT语句的任意位置使用替换变量（SELECT除

外)。 您可以使用&标记变量，&&标记变量不支持。遇到连续三个以上&标记变量取最后一个&。 变量名可以由数字，英文字母，下划线混合组成。 允许&与变量名之间有空格。 当ltsql执行SQL语句遇到替换变量时，ltsql会提示您输入该值。 您可以在提示符下输入任何字符串，即使是包含空格和标点符号的字符串。 输入字符串不超过240个字符。

这个例子展示了如何使用替换变量功能：

```
lightdb@oracle=# select &a
lightdb@oracle-# from &a;
Enter value for a : 123
Enter value for a : dual
old 1: select &a
new 1: select 123
old 2: from &a;
new 2: from dual;
?column?
-----
          123
(1 row)
```

这个例子不支持替换变量功能：

```
lightdb@oracle=# prepare stmt as select '&name';
PREPARE
```

## 环境

### COLUMNS

如果\pset columns为零，这个环境变量控制用于wrapped格式的宽度以及用来确定是否输出需要用到分页器或者切换到扩展自动模式中的垂直格式的宽度。

### LTDATABASE

### LTHOST

### LTPORT

### LTUSER

默认连接参数（见Section 32.14）。

### LT\_COLOR

规定在诊断消息中是否使用颜色。可能的值为always、 auto、 never。

### PSQL\_EDITOR

### EDITOR

### VISUAL

\e、\ef以及\ev命令所使用的编辑器。会按照列出的顺序检查这些变量，第一个被设置的将被使用。如果都没有被设置，默认是使用Unix系统上的vi或者Windows系统上的notepad.exe。

### PSQL\_EDITOR\_LINENUMBER\_ARG

当\e、\ef或者\ev带有一个行号参数时，这个变量指定用于传递起始行号给用户编辑器的命令行参数。对于Emacs或者vi之类的编辑器，这个变量是一个加号。如果需要在选项名称和行号之间有空格，可以在该变量的值中包括一个结尾的空格。例如：

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line'
```

在 Unix 系统上默认是+（对应于默认编辑器vi，且对很多其他常见编辑器可用）。在 Windows 系统上没有默认值。

#### PSQL\_HISTORY

命令历史文件的替代位置。波浪线（~）扩展会被执行。

#### PSQL\_PAGER PAGER

如果一个查询的结果在屏幕上放不下，它们会通过这个命令分页显示。典型的值是more或less。通过把PSQL\_PAGER或PAGER设置为空字符串可以禁用分页器的使用，调整\pset命令与分页器相关的选项也能达到同样的效果。会按照列出的顺序检查这些变量，第一个被设置的将被使用。如果都没有被设置，则大部分平台上默认使用more，但在Cygwin上使用less。

#### PSQLRC

用户的.ltsqlrc文件的替代位置。波浪线（~）扩展会被执行。

#### SHELL

被\!命令执行的命令。

#### TMPDIR

存储临时文件的目录。默认是/tmp。

和大部分其他LightDB工具一样，这个工具也使用libpq所支持的环境变量（见Section 32.14）。

## 文件

### ltsqlrc and ~/.ltsqlrc

如果没有-X选项，在连接到数据库后但在接收正常的命令之前，ltsql会尝试依次从系统级的启动文件（ltsqlrc）和用户的个人启动文件（~/.ltsqlrc）中读取并且执行命令。这些文件可以被用来设置客户端或者服务器，通常是一些\set和SET命令。

系统级的启动文件是ltsqlrc，它应该在安装好的LightDB的“系统配置”目录中，最可靠的定位方法是运行lt\_config --sysconfdir。默认情况下，这个目录将是../etc/（相对于包含LightDB可执行文件的目录）。可以通过PGSYSCONFDIR环境变量显式地设置这个目录的名称。

用户个人的启动文件是.ltsqlrc，它应该在调用用户的主目录中。在 Windows 上，由于没有用户主目录的概念，个人的启动文件是%APPDATA%\postgresql\ltsqlrc.conf。用户启动文件的位置可以通过PSQLRC环境变量设置。

系统级和用户个人的启动文件都可以弄成是针对特定ltsql版本的，方法是在文件名后面加上一个横线以及LightDB的主、次版本号，例如~/.ltsqlrc-9.2或者~/.ltsqlrc-9.2.5。版本最为匹配的文件会优先于不那么匹配的文件读入。

### .psql\_history

命令行历史被存储在文件~/.psql\_history中，或者是 Windows 的文件%APPDATA%\postgresql\psql\_history中。



历史文件的位置可以通过 HISTFILE ltsql 变量或者 PSQL\_HISTORY 环境变量明确的设置。

## 注解

- ltsql 和具有相同主版本或者更老的主版本服务器最为匹配。如果服务器的版本比 ltsql 本身要高，则反斜线命令尤其容易失败。不过，\d 家族的反斜线命令应该可以和版本 7.4 之后的服务器一起使用，但服务器的版本不必比 ltsql 本身新。运行 SQL 命令并且显示查询结果的一般功能应该也能和具有更新主版本的服务器一起使用，但是并非在所有的情况下都能保证如此。

如果你想用 ltsql 连接到多个具有不同主版本的服务器，推荐使用最新版本的 ltsql。或者，你可以为每一个主版本保留一份 ltsql 拷贝，并且针对相应的服务器使用匹配的版本。但实际上，这种额外的麻烦是不必要的。

- 在 LightDB 9.6 之前，-c 选项表示 -X (--no-ltsqlrc)，但现在不是这样了。
- 在 LightDB 8.4 之前，ltsql 允许一个单字母反斜线命令的第一个参数直接写在该命令后面，中间不需要空格。现在则要求一些空格。

## 给 Windows 用户的注解

ltsql 是一个“控制台应用”。由于 Windows 的控制台窗口使用的是一种和系统中其他应用不同的编码，在 ltsql 中使用 8 位字符时要特别注意。如果 ltsql 检测到一个有问题的控制台代码页，它将会在启动时警告你。要更改控制台代码页，有两件事是必要的：

- 输入 `cmd.exe /c chcp 1252` 可以设置代码页（1252 是适用于德语的一个代码页，请在这里替换成你的值）。如果正在使用 Cygwin，可以把这个命令放在 `/etc/profile` 中。
- 把控制台字体设置为 Lucida Console，因为栅格字体无法与 ANSI 代码页一起使用。

## LightDB 的注解

- 为了与 Oracle 数据库兼容，ltsql 允许用户在单独一行上输入斜杠并忽略它。此功能仅在 Oracle 兼容模式下可用。

```
lightdb@oracle_test=# show lightdb_dblevel_syntax_compatible_type ;
lightdb_dblevel_syntax_compatible_type
-----
Oracle
(1 row)

lightdb@oracle_test=#
lightdb@oracle_test=#
lightdb@oracle_test=# /
lightdb@oracle_test=# /
lightdb@oracle_test=# /
lightdb@oracle_test=# ;
```

## 示例

第一个例子展示了如何如何跨越多行输入一个命令。注意提示符的改变：

```
test=> CREATE TABLE my_table (
test(> first integer not null default 0,
test(> second text)
test-> ;
```

CREATE TABLE

现在再看看表定义:

```
test=> \d my_table
          Table "public.my_table"
  Column | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
  first  | integer |           | not null | 0
  second | text    |           |          |
```

现在我们把提示符改一改:

```
test=> \set PROMPT1 '%n@%m %~%R%#'
peter@localhost test=>
```

假定已经用数据填充了这个表并且想看看其中的数据:

```
peter@localhost test=> SELECT * FROM my_table;
 first | second
-----+-----
      1 | one
      2 | two
      3 | three
      4 | four
(4 rows)
```

你可以用\pset命令以不同的方式显示表:

```
peter@localhost test=> \pset border 2
Border style is 2.
peter@localhost test=> SELECT * FROM my_table;
+-----+-----+
| first | second |
+-----+-----+
|      1 | one    |
|      2 | two    |
|      3 | three  |
|      4 | four   |
+-----+-----+
(4 rows)
```

```
peter@localhost test=> \pset border 0
Border style is 0.
peter@localhost test=> SELECT * FROM my_table;
first second
-----
1 one
2 two
3 three
4 four
(4 rows)
```

```
peter@localhost test=> \pset border 1
Border style is 1.
```

```

peter@localhost test=> \pset format csv
Output format is csv.
peter@localhost test=> \pset tuples_only
Tuples only is on.
peter@localhost test=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
peter@localhost test=> \pset format unaligned
Output format is unaligned.
peter@localhost test=> \pset fieldsep '\t'
Field separator is "\t".
peter@localhost test=> SELECT second, first FROM my_table;
one      1
two      2
three    3
four     4

```

或者使用短命令:

```

peter@localhost test=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost test=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first | 1
second | one
-[ RECORD 2 ]-
first | 2
second | two
-[ RECORD 3 ]-
first | 3
second | three
-[ RECORD 4 ]-
first | 4
second | four

```

这是一个示例，用\df命令以发现名称匹配int\*pl并且第二参数是bigint类型的函数:

```

peter@localhost test=> SELECT * FROM my_table
peter@localhost test-> \g (format=aligned tuples_only=off expanded=on)
-[ RECORD 1 ]-
first | 1
second | one
-[ RECORD 2 ]-
first | 2
second | two
-[ RECORD 3 ]-
first | 3
second | three
-[ RECORD 4 ]-
first | 4
second | four

```

如果需要，可以用\crosstabview命令以交叉表的形式显示查询结果:

```
test=> SELECT first, second, first > 2 AS gt2 FROM my_table;
```

first	second	gt2
1	one	f
2	two	f
3	three	t
4	four	t

(4 rows)

```
test=> \crosstabview first second
```

first	one	two	three	four
1	f			
2		f		
3			t	
4				t

(4 rows)

这第二个例子展示了表的“乘法”（连接），行按照序号降序排序且列按照独立的、升序的方式排序。

```
test=> SELECT t1.first as "A", t2.first+100 AS "B", t1.first*(t2.first+100) as "AxB",
```

```
test(> row_number() over(order by t2.first) AS ord
```

```
test(> FROM my_table t1 CROSS JOIN my_table t2 ORDER BY 1 DESC
```

```
test(> \crosstabview "A" "B" "AxB" ord
```

A	101	102	103	104
4	404	408	412	416
3	303	306	309	312
2	202	204	206	208
1	101	102	103	104

(4 rows)

---

# reindexdb

reindexdb — 重索引一个LightDB数据库

## Synopsis

```
reindexdb [connection-option...] [option...] [ -S | --schema schema ] ... [ -t |  
--table table ] ... [ -i | --index index ] ... [dbname]
```

```
reindexdb [connection-option...] [option...] -a | --all
```

```
reindexdb [connection-option...] [option...] -s | --system [dbname]
```

## 描述

reindexdb是用于重建一个LightDB数据库中索引的工具。

reindexdb是 SQL 命令REINDEX的一个包装器。 在通过这个工具和其他方法访问服务器来重索引数据库之间没有实质性的区别。

## 选项

reindexdb接受下列命令行参数:

-a  
--all

重索引所有数据库。

--concurrently

使用 CONCURRENTLY 选项。 请参阅 REINDEX, 其中详细解释了此选项的所有注意事项。

[-d] dbname  
[--dbname=]dbname

当 -a/--all 未使用时, 指定要重新索引的数据库的名称。 如果未指定, 则从环境变量 PGDATABASE 中读取数据库名称。 如果未设置, 则使用为连接指定的用户名。 dbname 可以是 连接字符串。 如果是这样, 连接字符串参数将覆盖任何冲突的命令行选项。

-e  
--echo

回显reindexdb生成并发送到服务器的命令。

-i index  
--index=index

只是重建index。可以通过写多个-i开关来重建多个索引。

-j njobs  
--jobs=njobs

通过同时运行 njobs 命令并行执行 reindex 命令。 此选项可能会减少处理时间, 但也会增加数据库服务器上的负载。

reindexdb将打开到数据库的njobs连接，因此请确保max\_connections设置足够高，可以容纳所有连接。

请注意，此选项与 --index 和 --system 选项不兼容。

-q  
--quiet

不显示进度消息。

-s  
--system

仅索引数据库的系统目录。

-S schema  
--schema=schema

只对schema重建索引。通过写多个-S开关可以指定多个要重建索引的模式。

-t table  
--table=table

只索引table。可以通过写多个-t开关来重索引多个表。

--tablespace=tablespace

指定要重建索引的表空间。（这个名字以双引号标识符来处理。）

-v  
--verbose

在处理时打印详细信息。

-V  
--version

打印reindexdb版本并退出。

-?  
--help

显示有关reindexdb命令行参数的帮助并退出。

reindexdb也接受下列命令行参数用于连接参数：

-h host  
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port  
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username  
--username=username

要作为哪个用户连接。

`-w`  
`--no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

`-W`  
`--password`

强制 `reindexdb` 在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，`reindexdb` 将自动提示要求一个口令。但是，`reindexdb` 将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-w` 来避免额外的连接尝试。

`--maintenance-db=dbname`

当使用 `-a/--all` 时，指定要连接到的数据库名称以发现应重新索引哪些数据库。如果未指定，将使用 `postgres` 数据库，如果不存在，将使用 `template1`。这可以是连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。此外，在连接到其他数据库时，将重新使用除数据库名称本身之外的连接字符串参数。

## 环境

`PGDATABASE`  
`LTHOST`  
`LTPORT`  
`LTUSER`

默认连接参数

`PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、`auto`、`never`。

和大部分其他 LightDB 工具相似，这个工具也使用 `libpq`（见 Section 32.14 支持的环境变量）。

## 诊断

在有困难时，可以在 `REINDEX` 和 `ltsql` 中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何 `libpq` 前端库使用的默认连接设置和环境变量都将适用于此。

## 注解

`reindexdb` 可能需要多次连接到 LightDB 服务器，每一次都会询问一个口令。在这种情况下使用一个 `~/.pgpass` 文件会更方便。详见 Section 32.15

## 例子

要重索引数据库 `test`：

```
$ reindexdb test
```

要重索引名为 `abcd` 的数据库中的表 `foo` 和索引 `bar`：

```
$ reindexdb --table foo --index bar abcd
```

参见

REINDEX



---

# vacuumdb

vacuumdb — 对一个LightDB数据库进行垃圾收集和分析

## Synopsis

```
vacuumdb [connection-option...] [option...] [ -t | --table table [( column  
[,...] )] ] ... [dbname]
```

```
vacuumdb [connection-option...] [option...] -a | --all
```

## 描述

vacuumdb是用于清理一个LightDB数据库的工具。vacuumdb也将产生由LightDB查询优化器所使用的内部统计信息。

vacuumdb是 SQL 命令VACUUM的一个包装器。在通过这个工具和其他方法访问服务器来清理和分析数据库之间没有实质性的区别。

## 选项

vacuumdb接受下列命令行参数：

-a  
--all

清理所有数据库。

[-d] dbname  
[--dbname=]dbname

指定要清理或分析的数据库的名称，当不使用-a/--all时。如果未指定，则从环境变量PGDATABASE中读取数据库名称。如果环境变量也没有设置，指定给该连接的用户名将用作数据库名。dbname可以是connection string。如果是这样，连接时的字符串参数将覆盖任何冲突的命令行选项。

--disable-page-skipping

根据可见性地图的内容禁用跳过页面。

### Note

此选项仅对运行LightDB 9.6或更高版本的服务器有效。

-e  
--echo

回显vacuumdb生成并发送给服务器的命令。

-f  
--full

执行“完全”清理。

-F  
--freeze

强有力地“冻结”元组。

--force-index-cleanup

总是移除指向死元组的索引条目。

### Note

这个选项仅在服务器运行LightDB12或更晚版本时有效。

-j njobs  
--jobs=njobs

通过同时运行njobs 个命令来并行执行清理或者分析命令。这个选项可能会减少处理的时间，但是它也会增加数据库服务器的负载。

vacuumdb将开启 njobs个到数据库的连接，因此请确认你的max\_connections 设置足够高以容纳所有的连接。

注意如果某些系统目录被并行处理，使用这种模式加上 -f (FULL) 选项可能会导致 死锁失败。

--min-mxid-age mxid\_age

仅在multixact ID 年龄至少为mxid\_age的表上执行清空或分析命令。此设置对于确定要处理的表的优先级比较有用，以防止multixact ID 回绕。（参见Section 22.1.5.1）。

对于此选项的用途，关系的multixact ID年龄是主关系及其关联的TOAST表的年龄中最大的，如果存在的话。由于vacuumdb发出的命令在需要时还将处理TOAST表的关系，它无需单独考虑。

### Note

此选项仅对运行LightDB 9.6或更高版本的服务器有效。

--min-xid-age xid\_age

仅在事务ID 年龄至少为xid\_age的表上执行清空或分析命令。此设置对于确定要处理的表的优先级比较有用，以防止事务ID 回绕（参见 Section 22.1.5.2）。

对于此选项的用途，关系的事务ID年龄是主关系及其关联的TOAST表的年龄中最大的，如果存在的话。由于vacuumdb发出的命令在需要时还将处理TOAST表的关系，它无需单独考虑。

### Note

此选项仅对运行LightDB 9.6或更高版本的服务器有效。

--no-index-cleanup

不移除指向死元组的索引条目。

**Note**

这个选项仅在服务器运行LightDB 12 或更新版本时有效。

`--no-process-toast`

略过与要清理的表有关联的任何 TOAST 表。

**Note**

这个选项仅在服务器运行LightDB 14 或更新版本时有效。

`--no-truncate`

不要在表的结尾截断空页面。

**Note**

这个选项仅在服务器运行LightDB 12 或更新版本时有效。

`-P parallel_workers`

`--parallel=parallel_workers`

指定`parallel vacuum`的并发数量。 这允许清理利用多个 CPU 来处理索引。 请参见VACUUM。

**Note**

此选项仅适用于运行LightDB13 及更高版本的服务器。

`-q`

`--quiet`

不显示进度消息。

`--skip-locked`

跳过无法立即锁定以进行处理的关系。

**Note**

此选项仅对运行LightDB 12或更高版本的服务器有效。

`-t table [ (column [,...]) ]`

`--table=table [ (column [,...]) ]`

只清理或分析`table`。列名只能和`--analyze`或`--analyze-only`选项一起被指定。通过写多个`-t`开关可以清理多个表。

### Tip

如果你指定列，你可能必须转义来自 shell 的括号（见下面的例子）。

-v  
--verbose

在处理期间打印详细信息。

-V  
--version

打印vacuumdb版本并退出。

-z  
--analyze

也计算优化器使用的统计信息。

-Z  
--analyze-only

只计算优化器使用的统计信息（不清理）。

--analyze-in-stages

与--analyze-only相似，只计算优化器使用的统计信息（不做清理）。使用不同的配置设置运行分析的几个（目前是 3个）阶段以更快地产生可用的统计信息。

这个选项对分析一个刚从转储恢复或者通过lt\_upgrade得到的数据库有用。这个选项将尝试尽可能快地创建一些统计信息来让该数据库可用，然后在后续的阶段中产生完整的统计信息。

-?  
--help

显示有关vacuumdb命令行参数的帮助并退出。

vacuumdb也接受下列命令行参数用于连接参数：

-h host  
--host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

-p port  
--port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

-U username  
--username=username

要作为哪个用户连接。

-w  
--no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个.pgpass文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

-W  
--password

强制vacuumdb在连接到一个数据库之前提示要求一个口令。

这个选项不是必不可少的，因为如果服务器要求口令认证，vacuumdb将自动提示要求一个口令。但是，vacuumdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用-W来避免额外的连接尝试。

--maintenance-db=dbname

当使用-a/--all时，指定要连接到的数据库名称以发现应该清理的数据库。如果未指定，将使用postgres数据库，如果不存在，将使用template1。这可以是connection string。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。此外，在连接到其他数据库时，将重新使用除数据库名称本身之外的连接字符串参数。

## 环境

PGDATABASE  
LTHOST  
LTPORT  
LTUSER

默认连接参数

PG\_COLOR

规定在诊断消息中是否使用颜色。可能的值为always、auto和never。

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 诊断

在有困难时，可以在VACUUM和ltsql中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

## 注解

vacuumdb可能需要多次连接到LightDB服务器，每次都询问一个口令。在这种情况下有一个~/.pgpass文件会很方便。详见Section 32.15

## 例子

要清理数据库test:

```
$ vacuumdb test
```

要清理和为优化器分析一个名为bigdb的数据库:

```
$ vacuumdb --analyze bigdb
```

要清理在名为xyzyz的数据库中的一个表foo，并且为优化器分析该表的bar列:

```
$ vacuumdb --analyze --verbose --table='foo(bar)' xyzyz
```

参见

VACUUM

---

# LightDB 服务器应用

这一部分包含LightDB服务器应用和支持工具的参考信息。这些命令只在数据库服务器所在的主机上运行才有用。其他工具程序在LightDB 客户端应用中列出。

## Table of Contents

initdb .....	2028
lt_initdb .....	2033
lt_patch .....	2038
lt_check .....	2040
lt_archivecleanup .....	2042
lt_checksums .....	2044
lt_controldata .....	2046
lt_ctl .....	2047
lt_resetwal .....	2052
lt_rewind .....	2055
lt_test_fsync .....	2059
lt_test_timing .....	2060
lt_upgrade .....	2063
lt_upgrade_check .....	2071
lt_waldump .....	2073
lightdb .....	2075
postmaster .....	2082

---

# initdb

initdb — 创建一个新的LightDB数据库集簇

## Synopsis

```
initdb [option...] [ --pgdata | -D ] directory
```

## 描述

initdb创建一个新的LightDB数据库集簇。一个数据库集簇是由一个单一服务器实例管理的数据库的集合。

一个数据库集簇的创建包括创建存放数据库数据的目录、生成共享目录表（属于整个集簇而不是任何特定数据库的表）并且创建template1和postgres数据库。当你后来创建一个新的数据库时，任何在template1数据库中的东西都会被复制（因此，任何已安装在template1中的东西都会被自动地复制到后来创建的每一个数据库中）。postgres数据库是便于用户、工具和第三方应用使用的默认数据库。

尽管initdb将尝试创建指定的数据目录，它可能没有权限（如果想要的数据目录的父目录被根用户拥有）。要在这样一种设置中初始化，作为 root 创建一个空数据目录，然后使用chown将该目录赋予给数据库用户账户，再然后su成为该数据库用户并运行initdb。

initdb必须以将拥有该服务器进程的用户运行，因为该服务器需要访问initdb创建的文件和目录。因为该服务器不能作为 root 运行，你不能以 root 运行initdb（事实上它会拒绝这样做）。

由于安全原因，由initdb创建的新集簇默认将只能由集簇所有者访问。--allow-group-access选项允许与集簇所有者同组的任何用户读取集簇中的文件。这对非特权用户执行备份很有用。

initdb初始化该数据库集簇的默认区域和字符集编码。当一个数据库被创建时，其字符集编码、排序顺序（LC\_COLLATE）和字符集类（LC\_CTYPE，例如大写、小写、数字）可以被独立设置。initdb为template1数据库确定那些设置，它们将作为所有其他数据库的默认值。

要修改默认排序顺序或字符集类，使用--lc-collate和--lc-ctype选项。除C或POSIX之外的排序顺序也有性能罚值。由于这些原因，在运行initdb时选择正确的区域很重要。

余下的区域分类可以在服务器启动之后改变。你也可以使用--locale来为所有区域分类设置默认值，包括排序顺序和字符集类。所有服务器区域值（lc\_\*）可以通过SHOW ALL显示。详见Section 21.1

要修改默认编码，使用--encoding。详见Section 21.3

在使用 initdb 初始化数据库集群之后，initdb 会将存储数据库集群的目录记录在 \$LTHOME/instance\_list.conf 文件中。

## 选项

```
-A authmethod  
--auth=authmethod
```

这个选项为本地用户指定在lt\_hba.conf中使用的默认认证方法（host和local行）。initdb将使用指定的认证方法为非复制连接以及复制连接填充lt\_hba.conf项。



除非你信任你系统上的所有本地用户，不要使用`trust`。为了安装方便，`trust`是默认值。

`--auth-host=authmethod`

这个选项为通过 TCP/IP 连接的本地用户指定在`lt_hba.conf`中使用的认证方法（`host`行）。

`--auth-local=authmethod`

这个选项为通过 Unix 域套接字连接的本地用户指定在`lt_hba.conf`中使用的认证方法（`local`行）。

`-p port`

此选项指定数据库集群的端口。

`-D directory`

`--pgdata=directory`

这个选项指定数据库集簇应该存放的目录。这是`initdb`要求的唯一信息，但是您可以通过设定`LTDATA`环境变量来避免书写它，这很方便因为之后数据库服务器（`lightdb`）可以使用同一个变量来找到数据库目录。

`-E encoding`

`--encoding=encoding`

选择模板数据库的编码。这也将是后来创建的任何数据库的默认编码，除非你覆盖它。默认值来自于区域，或者如果该值不起作用则为`SQL_ASCII`。`LightDB`服务器所支持的字符集在Section 21.3. 中描述。

`-g`

`--allow-group-access`

允许与集簇所有者同组的用户读取`initdb`创建的所有集簇文件。Windows会忽略此选项，因为它不支持POSIX样式的组权限。

`-k`

`--data-checksums`

在数据页面上使用校验码来帮助检测 I/O 系统造成的损坏。启用校验码将会引起显著的性能惩罚。如果设置，则为所有对象计算校验和，在整个数据库中。所有校验和失败都将报告在 `pg_stat_database` 视图。

`-K`

`--encryption-key-command`

用于透明数据加密。在创建数据库实例之前，您必须编写一些代码，以确保数据库可以在启动和实例创建期间读取密钥。密钥是一个 32 字节的十六进制字符串。

这是最简单的例子：

```
cat /somewhere/provide_key.sh
#!/bin/sh
echo 882fb7c12e80280fd664c69d2d636913
```

您只需要编写一个程序将密钥打印到标准输出。请确保 `LightDB` 能够执行此程序：

```
chmod +x /somewhere/provide_key.sh
```

注意：您不必编写一个 shell 脚本，可以使用任何可执行文件，例如 C、Go 或 Python 等。

创建数据库实例：

```
initdb -p 5432 -D ./data -K /somewhere/provide_key.sh
```

`--locale=locale`

为数据库集簇设置默认区域。如果这个选项没有被指定，该区域将从initdb所运行的环境中继承。区域支持在Section 21. 中描述。

`--lc-collate=locale`

`--lc-ctype=locale`

`--lc-messages=locale`

`--lc-monetary=locale`

`--lc-numeric=locale`

`--lc-time=locale`

和`--locale`相似，但是只在指定的分类中设置区域。

`--no-locale`

等效于`--locale=C`。

`-N`

`--no-sync`

默认情况下，initdb将等待所有文件被安全地写到磁盘。这个选项会导致initdb不等待就返回，这当然更快，但是也意味着一次后续的操作系统崩溃可能让数据目录损坏。通常，这个选项对测试有用，但是不应该在创建生产安装时使用。

`--pwfile=filename`

让initdb从一个文件读取数据库超级用户的口令。该文件的第一行被当作口令。

`-S`

`--sync-only`

安全地把所有数据库文件写入到磁盘并退出。这不会执行任何正常的initdb操作。

`-T config`

`--text-search-config=config`

设置默认的文本搜索配置。详见default\_text\_search\_config。

`-U username`

`--username=username`

选择数据库超级用户的用户名。这个的默认值是实际运行initdb的用户的名称。超级用户的名字是什么真的不重要，但是你可以选择保留常用的名字postgres，即使操作系统的用户名不同。

`-W`

`--pwprompt`

让initdb提示要求为数据库超级用户给予一个口令。如果你没有计划使用口令认证，这就不重要。否则在你设置一个口令之前你就无法使用口令认证。

-X directory  
--waldir=directory

这个选项指定预写式日志会被存储在哪个目录中。

--wal-segsize=size

设置WAL段尺寸，以兆字节为单位。这是WAL日志中每个文件的尺寸。默认的尺寸为16兆字节。该值必须位于2的1次幂和1024次幂（兆字节）之间。这个选项只能在初始化期间设置，并且之后不能更改。

调整这个值来控制WAL日志传送或者归档可能会有用。此外，在有大量WAL的数据库中，每个目录中数量巨大的WAL文件可能会成为性能和管理问题。增加WAL文件尺寸将会降低WAL文件的数量。

其他较少使用的选项：

-d  
--debug

打印来自引导后端的调试输出以及普通大众不那么感兴趣的一些消息。引导后端被程序initdb用来创建目录表。这个选项会生成大量极端无聊的输出。

-L directory

指定initdb应从哪里寻找它的输入文件来初始化数据库集簇。这通常没有必要。如果你需要显式指定它们的位置，你应该被告知。

-n  
--no-clean

默认情况下，当initdb确定有一个错误阻止它完整地创建数据库集簇，它会移除在它发现无法完成任务之前创建的任何文件。这个选项会抑制这种整理并且对调试有用。

其他选项：

-V  
--version

打印initdb版本并退出。

-?  
--help

显示有关initdb命令行参数的帮助并退出。

## 环境

LTDATA

指定数据库集簇应该被存放的目录，可以使用-D选项覆盖。

LT\_COLOR

规定在诊断消息中是否使用颜色。可能的值为always、 auto、 never。

TZ

指定创建的数据集簇的默认时区。值应该是一个完整的时区名称（见Section 8.5.3）。

和大部分其他LightDB工具相似，这个工具也使用libpq（见Section 32.14 支持的环境变量）。

## 注解

initdb可以通过`lt_ctl initdb`被调用。

## 参见

`lt_ctl`, `lightdb`

---

# lt\_initdb

lt\_initdb — initdb 工具包，用于创建一个新的 LightDB 数据库集群

## Synopsis

```
lt_initdb [option...] [ --pgdata | -D ] directory
```

## 描述

lt\_initdb 工具包，用于创建一个新的 LightDB 数据库集群。数据库集群是由单个服务器实例管理的一组数据库。

## 选项

-A authmethod

--auth=authmethod

此选项指定了本地用户在 lt\_hba.conf (host 和 local 行) 中使用的默认身份验证方法。initdb 将使用指定的身份验证方法预填充 lt\_hba.conf 条目，用于非复制以及复制连接。

除非您信任系统上的所有本地用户，请不要使用 trust。为了方便安装，trust 是默认值。

--auth-host=authmethod

此选项指定了通过 TCP/IP 连接在 lt\_hba.conf (host 行) 中使用的本地用户的身份验证方法。

--auth-local=authmethod

此选项指定了通过 Unix 套接字连接在 lt\_hba.conf (local 行) 中使用的本地用户的身份验证方法。

-p port

此选项指定数据库集群的端口。默认端口为 5432。

-D directory

--pgdata=directory

此选项指定应存储数据库集群的目录。这是 initdb 所需的唯一信息，但您可以通过设置 LTDATA 环境变量来避免编写它，这可能很方便，因为数据库服务器 (lightdb) 可以在后来通过相同的变量找到数据库目录。

-E encoding

--encoding=encoding

选择模板数据库的编码。这也将是您稍后创建的任何数据库的默认编码，除非您在那里覆盖它。默认值来自区域设置，如果不起作用，则为 SQL\_ASCII。LightDB 服务器支持的字符集在 Section 21.3.1 中描述。

-g

--allow-group-access

允许与集群所有者在同一组中的用户读取由 initdb 创建的所有集群文件。在 Windows 上，此选项将被忽略，因为它不支持 POSIX 风格的组权限。

`-u`  
`--upgrade-mode`

创建一个没有扩展的数据库集群，相当于直接使用 `initdb`。

`-k`  
`--data-checksums`

在数据页上使用校验和以帮助检测 I/O 系统可能会默默发生的损坏。启用校验和可能会产生明显的性能损失。如果设置了校验和，则所有对象的所有数据库都会计算校验和。所有校验和故障都将在 `pg_stat_database` 中报告。

`-K`  
`--encryption-key-command`

用于透明数据加密。在创建数据库实例之前，您必须编写一些代码，以确保数据库在启动和实例创建期间可以读取密钥。密钥是一个 32 字节的十六进制字符串。

以下是可能的最简单示例：

```
cat /somewhere/provide_key.sh
#!/bin/sh
echo 882fb7c12e80280fd664c69d2d636913
```

您只需要编写一个将密钥打印到标准输出的程序即可。确保 LightDB 能够执行此程序：

```
chmod +x /somewhere/provide_key.sh
```

注意：您不必编写 shell 脚本 - 您可以使用任何可执行文件，例如 C、Go 或 Python。

创建数据库实例：

```
lt_initdb -p 5432 -D ./data -K /somewhere/provide_key.sh
```

`--locale=locale`

设置数据库集群的默认区域设置。如果未指定此选项，则从 `initdb` 运行的环境中继承区域设置。区域设置支持在 Section 21.1 中描述。

`--lc-collate=locale`  
`--lc-ctype=locale`  
`--lc-messages=locale`  
`--lc-monetary=locale`  
`--lc-numeric=locale`  
`--lc-time=locale`

类似于 `--locale`，但仅在指定的类别中设置区域设置。

`--no-locale`

等同于 `--locale=C`。

`-N`  
`--no-sync`

默认情况下，`initdb` 会等待所有文件安全写入磁盘。此选项使 `initdb` 返回而不等待，这样速度更快，但意味着随后的操作系统崩溃可能会导致数据目录损坏。通常，此选项对于测试很有用，但在创建生产安装时不应使用。

`--pwfile=filename`

从文件中读取数据库超级用户的密码。文件的第一行被视为密码。

`-S`

`--sync-only`

安全地将所有数据库文件写入磁盘并退出。这不执行任何正常的 `initdb` 操作。

`-T config`

`--text-search-config=config`

设置默认的文本搜索配置。有关更多信息，请参见 `default_text_search_config`。

`-U username`

`--username=username`

选择数据库超级用户的用户名。默认情况下，它是运行 `initdb` 的有效用户的名称。超级用户的名称并不重要，但您可以选择使用惯常的名称 `lightdb`，即使操作系统用户的名称不同。

`-W`

`--pwprompt`

强制 `initdb` 提示输入数据库超级用户的密码。如果您不打算使用密码验证，那么这个选项不重要。否则，在设置密码之前，您将无法使用密码验证。

`-X directory`

`--waldir=directory`

指定写入前日志应存储的目录。

`--wal-segsize=size`

设置 WAL 日志段的大小，以兆字节为单位。这是 WAL 日志中每个单独文件的大小。默认大小为 512 兆字节。该值必须是 1 到 1024（兆字节）之间的 2 的幂。此选项只能在初始化期间设置，无法在以后更改。

调整此大小可用于控制 WAL 日志传送或归档的粒度。此外，在 WAL 日志量很大的数据库中，每个目录中的 WAL 文件数量可能会成为性能和管理问题。增加 WAL 文件大小将减少 WAL 文件数量。

`--install-mode=mode`

数据库集群安装模式，共有三种安装模式。

- `single`: 独立版本。
- `ha`: 高可用版本。
- `distributed`: 分布式版本。

默认安装模式为 `single`。

`--compatible-type=type`

数据库集群兼容性类型，共有三种兼容性类型：

- `off`: 与其他数据库功能不兼容。
- `mysql`: 与 MySQL 数据库功能兼容。
- `oracle`: 与 Oracle 数据库功能兼容。

默认兼容性类型为 `off`。

还有其他不太常用的选项可用：

`-d`  
`--debug`

从引导后端和其他一些不太重要的消息中打印调试输出。引导后端是程序 `initdb` 用于创建目录表的程序。此选项会生成大量极其无聊的输出。

`-L directory`

指定 `initdb` 初始化数据库集群时应查找其输入文件的位置。通常情况下，这不是必需的。如果需要明确指定其位置，程序会提示您。

`-n`  
`--no-clean`

默认情况下，当 `initdb` 确定错误阻止其完全创建数据库集群时，它会删除在发现无法完成作业之前可能创建的任何文件。此选项禁止清理，因此对于调试非常有用。

其他选项：

`-V`  
`--version`

打印 `initdb` 的版本并退出。

`-?`  
`--help`

显示有关 `initdb` 命令行参数的帮助信息，并退出。

## 环境

`LTDATA`

指定数据库集簇存储的目录；可以使用 `-D` 选项来覆盖。

`LT_COLOR`

指定是否在诊断信息中使用颜色。可能的值为 `always`、`auto` 和 `never`。

`TZ`

指定创建的数据库集簇的默认时区。该值应为完整的时区名称（参见 Section 8.5.3）。

像大多数其他 LightDB 实用程序一样，此实用程序也使用 `libpq` 支持的环境变量（请参见 Section 32.14）。

## 注意事项

建议使用 `lt_initdb`，否则某些功能将不可用。

如果您直接运行 `initdb`，可以像以下这样使用：

```
$LTHOME/scripts/5_lightdb_modify_para.sh $LTHOME $LTDATA OLTP/
OLAP shared_buffers(MB) effective_cache_size(MB) $LTPORT $INSTALL_MODE
$COMPATIBLE_TYPE
```



```
$LTHOME/scripts/6_lightdb_start.sh $LTHOME $LTDATA (如果未启动)  
$LTHOME/scripts/8_lightdb_create_extension.sh $LTHOME $LTPORT  
$INSTALL_MODE
```

---

# lt\_patch

lt\_patch — the LightDB 补丁工具

## Synopsis

```
lt_patch init [-i patchid]
lt_patch make
lt_patch install [-D ltdata] [-l lthome]
lt_patch remove [-D ltdata] [-l lthome] [-f] -i patchid
lt_patch list [-l lthome] [-i patchid]
```

## 描述

lt\_patch 可以帮助确保您的 LightDB 软件保持最新和安全，它可以在选定的LightDB主目录（LTHOME）上启用补丁的应用和回滚到数据库上。

lt\_patch 解析补丁配置并执行补丁操作而无需用户干预。它可以：

- 执行各种检查。
- 应用补丁。
- 回滚补丁。

## 选项

-l  
--lthome

指定 LightDB 软件的文件系统位置。 如果省略此选项，则使用环境变量 LTHOME。

-D  
--ltdata

指定 LightDB 实例的文件系统位置。 如果省略此选项，则使用环境变量 LTDATA。

-i  
--patch-id

指定一个补丁的唯一 ID。

-f  
--force

在执行卸载 shell 脚本时忽略错误。

-V  
--version

打印 lt\_patch 的版本信息并退出。

-?  
--help

显示关于 lt\_patch 命令行参数的帮助信息，并退出。

## 补丁流程概述

获取所需的补丁：第一步是确定您需要哪些补丁。关注补丁的以下几个项目：

- 名称：patch-1.0-8085
- 适用版本：13.8-22.4
- 适用 CPU：x86\_64
- 适用操作系统：kylinV10sp1

将补丁应用到所需的目标（LTHOME）。

### Note

在执行任何补丁任务之前，请始终阅读补丁 README 文件以获取任何特殊的补丁说明。

## 使用方法

应用一个补丁：解压您获取的补丁包并进入补丁目录，然后运行以下命令：

```
$ lt_patch install -l /home/lightdb/stage/lightdb-x
```

回滚一个补丁：

```
$ lt_patch remove -l /home/lightdb/stage/lightdb-x -i patch-1.0-8085
```

查询补丁信息（列出所有补丁信息）：

```
$ lt_patch list -l /home/lightdb/stage/lightdb-x
```

查询补丁信息（列出一个补丁信息）：

```
$ lt_patch list -l /home/lightdb/stage/lightdb-x patch-1.0-8085
```

### Note

“lt\_patch init” 和 “lt\_patch make” 仅供内部开发人员使用。

---

# lt\_check

lt\_check — LightDB 的 sha256sum 校验工具

## Synopsis

```
lt_check --check [=filename] [--lthome=$LTHOME]
```

```
lt_check --generate [--lthome="$LTHOME"] --comment="comment info"
```

## 描述

lt\_check 可以帮助您检查或生成 LightDB 中 \$LTHOME/lib、\$LTHOME/bin、\$LTHOME/share/extension 目录下的文件的 sha256sum。

sha256sum 文件保存在 \$LTHOME/etc 目录下，并以 lt\_checksum.X (X 是一个数字) 开头。

## 选项

--lthome=\$LTHOME

指定 LightDB 软件的文件系统位置。如果省略此选项，则使用环境变量 LTHOME。

--generate

在 \$LTHOME/etc 目录下生成一个 sha256sum 文件。

--check=filename

使用 sha256sum 检查指定的文件名。如果省略文件名，则会在 \$LTHOME/etc 目录下获取一个最大的序号。

--comment="test info"

将注释信息保存到 sha256sum 文件的第二行。此选项只能与 --generate 一起使用。

-?

--help

显示关于 lt\_check 命令行参数的帮助信息，并退出。

## 使用方法

在 \$LTHOME/etc 目录下生成一个 sha256sum 文件，并将文件头添加注释 "test info"。

```
$ lt_check --generate --comment="test info"
```

检查一个 sha256sum 文件。默认使用 \$LTHOME/etc/lt\_checksum.X 文件（如果有多个文件，则 X 是最大的编号）。

```
$ lt_check --check
```

指定 lthome 环境变量并检查给定的 sha256sum 文件。

```
$ lt_check --lthome=/usr/local/lightdb/lightdb-x --check=/usr/local/lightdb/lightdb-x/etc/lt_checksum.1
```

指定 lthome 环境变量并在 \$LTHOME/etc 目录下生成一个 sha256sum 文件。

```
$ lt_check --lthome=/usr/local/lightdb/lightdb-x --generate --comment="test info"
```

---

# lt\_archivecleanup

lt\_archivecleanup — 清理LightDB WAL 归档文件

## Synopsis

```
lt_archivecleanup [option...] archivelocation oldestkeptwalfile
```

## 简介

lt\_archivecleanup被设计用作 archive\_cleanup\_command在作为后备服务器运行（Section 24.2 时来清理 WAL 文件归档。lt\_archivecleanup也可以被用作一个单独的程序来清理 WAL 文件归档。

要配置一个后备服务器以使用lt\_archivecleanup，把下面的内容放在lightdb.conf配置文件中：

```
archive_cleanup_command = 'lt_archivecleanup archivelocation %r'
```

其中archivelocation是要从中移除 WAL 段文件的目录。

当被用在archive\_cleanup\_command中时，所有逻辑上在 %r参数的值之前的 WAL 文件都将从 archivelocation移除。这能最小化需要被保留的文件数量，同时能保留崩溃后重启的能力。如果对于这台特定的后备服务器， archivelocation是一个短暂需要的区域，使用这个参数就是合适的，但是当archivelocation要用作一个长期的 WAL 归档 区域或者当多个后备服务器正在从这个归档位置恢复时，使用这个参数就不合适。

当被用作一个单独的程序时，所有逻辑上在oldestkeptwalfile 之前的 WAL 文件将从archivelocation中移除。在这种模式中，如果指定了.partial或者.backup文件名，则只有该文件前缀将被用作oldestkeptwalfile。这种对 .backup文件名的处理允许你移除所有在一个特定基础备份之前归档的 WAL 文件而不出错。例如，下面的例子将移除所有比WAL 文件名 000000010000003700000010老的文件：

```
lt_archivecleanup -d archive 000000010000003700000010.00000020.backup
```

```
lt_archivecleanup: keep WAL file "archive/000000010000003700000010" and later
lt_archivecleanup: removing file "archive/00000001000000370000000F"
lt_archivecleanup: removing file "archive/00000001000000370000000E"
```

lt\_archivecleanup假定 archivelocation是一个可读的目录并且对于服务器拥有者是可写的。

## 选项

lt\_archivecleanup接受下列命令行参数：

-d

在stderr上打印很多调试日志输出。

-n

在stdout上打印将被移除的文件的名字（执行一次演习）。

-V  
--version

打印lt\_archivecleanup版本并退出。

-x extension

提供一个扩展名，在决定所有的文件 是否应该被删除之前，将从文件名中剥离这个扩展名。这通常有助于清理已经 存储期间被压缩过并且被压缩程序增加了一个扩展名的归档。例如： -x .gz。

-?  
--help

显示lt\_archivecleanup命令行参数的帮助并退出。

## 环境

环境变量 `LT_COLOR` 指定是否在诊断消息中使用颜色。可能的值是 `always`、`auto` 和 `never`。

## 注解

在作为一个单独的工具时，`lt_archivecleanup` 被设计为与LightDB 8.0 及其后的版本一起工作。如果 作为一个归档清理命令使用，则需要和LightDB 9.0 及其后的版本一起工作。

`lt_archivecleanup`以 C 写成并且具有很容易修改的 源代码，其中有特别指定的区域用于修改以符合你的需要

## 示例

在 Linux 或者 Unix 系统上，你可能会用：

```
archive_cleanup_command = 'lt_archivecleanup -d /mnt/standby/archive %r  
2>>cleanup.log'
```

其中归档目录位于后备服务器上，这样`archive_command`通过 NFS 来访问它，但是文件对于后备服务器来说是本地的。这将会

- 在`cleanup.log`中产生调试输出
- 从归档目录中移除不再需要的文件

## 参见

`lt_standby`

---

# lt\_checksums

lt\_checksums — 在LightDB数据库集簇中启用、禁用或检查数据校验和

## Synopsis

```
lt_checksums [option...] [[ -D | --pgdata ]datadir]
```

## 简介

lt\_checksums在LightDB集簇中检查、启用或禁用数据校验和。运行lt\_checksums之前，必须彻底关闭服务器。验证校验和时，如果没有校验和错误，则退出状态为零，如果检测到至少一个校验和失败，则退出状态为非零。启用或禁用校验和时，如果操作失败，则退出状态为非零。

验证校验和时，集簇中的每个文件都要被扫描。启用校验和时，集簇中的每个文件都会被原地重写。禁用校验和时，仅更新lt\_control文件。

## 选项

下列命令选项可用：

-D directory

--pgdata=directory

指定存储数据库集簇的目录。

-c

--check

检查校验和。如果未指定其它任何内容，这是缺省模式。

-d

--disable

禁用校验和。

-e

--enable

启用校验和。

-f filenode

--filenode=filenode

仅验证文件节点为filenode的关系中的校验和。

-N

--no-sync

缺省地，lt\_checksums会等待所有文件安全地写到磁盘上。该选项使得lt\_checksums不等待就返回，这样更快，但意味着后续如果操作系统崩溃会让更新的数据目录损坏。一般地，该选项对测试有用，但不应用在生产安装上。当使用--check时，该选项无效。

-P

--progress

启用进度报告。在检查或启用校验和时，打开该选项，会提供进度报告。



-v  
--verbose

启用详细输出。列出所有检查的文件。

-V  
--version

打印lt\_checksums版本并退出。

-?  
--help

显示关于lt\_checksums命令行参数的帮助并退出。

## 环境

LTDATA

指定数据库集簇存储的目录；可以用-D选项覆盖。

LT\_COLOR

指定是否在诊断消息中使用颜色。可能的值为always, auto, never。

## 注意

在大型集簇中启用校验和的时间可能很长。在此操作期间，写到数据目录的集簇或其它程序必须是未启动的，否则可能出现数据丢失。

当复制设置与执行关系文件块的直接拷贝的工具（例如lt\_rewind）一起使用时，启用和禁用校验和会导致以不正确校验和形式出现的页面损坏，如果未在所有节点上执行一致的操作的话。故在复制设置中启用或禁用校验和时，推荐一致地切换所有集簇之前停止所有集簇。此外销毁所有备用数据库，在主数据库上执行操作，最后从头开始重建备用服务器，也是安全的。

如果在启用或禁用校验和时异常终止或杀掉lt\_checksums，那么集簇的数据校验和配置保持不变，lt\_checksums可以重新运行以执行相同操作。

---

# lt\_controldata

lt\_controldata — 显示一个LightDB数据库集簇的控制信息

## Synopsis

```
lt_controldata [option] [[ -D | --pgdata ]datadir]
```

## 描述

lt\_controldata打印在initdb期间初始化的信息，例如目录版本。它也显示关于预写式日志和检查点处理的信息。这种信息是集簇范围的，并且不针对任何一个数据库。

这个工具只能由初始化集簇的用户运行，因为它要求对数据目录的读访问。你可以在命令行中指定数据目录，或者使用环境变量LTDATA。这个工具支持选项-V和--version，它们打印lt\_controldata版本并退出。它也支持选项-?和--help，它们输出支持的参数。

## 环境

LTDATA

默认的数据目录位置。

PG\_COLOR

规定在诊断消息中是否使用颜色。可能的值为always、 auto、 never。

---

# lt\_ctl

lt\_ctl — 初始化、启动、停止或控制一个LightDB服务器

## Synopsis

```
lt_ctl init[db] [-D datadir] [-s] [-o initdb-options]
```

```
lt_ctl start [-D datadir] [-l filename] [-W] [-t seconds] [-s] [-o options] [-p path] [-c]
```

```
lt_ctl stop [-D datadir] [-m s[mart] | f[ast] | i[mmediate] ] [-W] [-t seconds] [-s]
```

```
lt_ctl restart [-D datadir] [-m s[mart] | f[ast] | i[mmediate] ] [-W] [-t seconds] [-s] [-o options] [-c]
```

```
lt_ctl reload [-D datadir] [-s]
```

```
lt_ctl status [-D datadir]
```

```
lt_ctl promote [-D datadir] [-W] [-t seconds] [-s]
```

```
lt_ctl logrotate [-D datadir] [-s]
```

```
lt_ctl kill signal_name process_id
```

在Microsoft Windows上，还有：

```
lt_ctl register [-D datadir] [-N servicename] [-U username] [-P password] [-S a[uto] | d[emand] ] [-e source] [-W] [-t seconds] [-s] [-o options]
```

```
lt_ctl unregister [-N servicename]
```

## 描述

### Note

在高可用模式下直接使用 `lt_ctl` 可能会导致意外的切换。建议改用 `lightdb_service.py`。

`lt_ctl` 是一个用于初始化LightDB数据库集簇，启动、停止或重启LightDB数据库服务器（`lightdb`），或者显示一个正在运行服务器的状态的工具。尽管服务器可以被手工启动，`lt_ctl` 包装了重定向日志输出以及正确地从终端和进程组脱离等任务。它也提供了方便的选项用来控制关闭。

`init`或`initdb`模式会创建一个新的LightDB数据库集簇，也就是将由一个单一服务器实例管理的数据库集合。这个模式调用`initdb`命令。详见`initdb`。

`start`模式启动一个新的服务器。该服务器被启动在后台，并且它的标准输出被附加到`/dev/null`（或Windows上的`nul`）。在Unix类系统上，默认情况下服务器的标准输出和标准错误被发送到`lt_ctl`的标准输出（不是标准错误）。`lt_ctl`的标准输出应该接着被重定向到一个文件或用管道导向另一个进程（例如日志轮转程序`rotatelogs`）。否则`lightdb`将把它的输出写到控制终端（从后台）并且将不会离开`shell`的进程组。在Windows上，默认情况下服务器的标准输出和标准错误被发送到终端。这些默认行为可以使用`-l`追加服务器的输出到一个日志文件来改变。我们推荐使用`-l`或输出重定向。

stop模式关闭运行在指定数据目录中的服务器。对-m选项可以选择三种不同的关闭方法。“Smart”模式不允许新连接，然后等待所有现有的客户端断开连接以及任何在线备份结束。如果该服务器是热备，一旦所有的客户端已经断开连接，恢复和流复制将被终止。“Fast”模式（默认）不会等待客户端断开连接并且将终止进行中的在线备份。所有活动事务都被回滚并且客户端被强制断开连接，然后服务器被关闭。“Immediate”模式将立刻中止所有服务器进程，而不是做一次干净的关闭。这中选择将导致下一次重启时进行一次崩溃恢复。

restart模式实际上会先执行一个停止操作然后紧接着执行一个启动操作。这使得我们能够更改lightdb的命令行选项，或者更改不通过重启服务器无法更改的配置文件选项。如果在服务器启动期间在命令行上使用了相对路径，则restart可能会失败，除非lt\_ctl被运行在与上次启动服务器相同的目录中。

reload模式简单地向lightdb服务器进程发送一个SIGHUP信号，导致它重新读取它的配置文件（lightdb.conf、lt\_hba.conf等）。这允许改变配置文件选项而无需一次完整的服务器重启来让改变生效。

status模式检查一个服务器是否运行在指定的数据目录中。如果有一个服务器正在运行，其PID和用来调用它的命令行选项将被显示。如果服务器没有在运行，lt\_ctl将返回退出状态3。如果没有指定一个可以访问的数据目录，lt\_ctl将返回退出状态4。

promote模式命令运行在指定数据目录中的后备服务器结束后备模式并且开始读写操作。

logrotate模式轮换服务器日志文件。有关如何将此模式与外部日志轮换工具一起使用的详细信息，参见Section 22.3

kill模式向一个指定进程发送一个消息。这主要用于没有kill命令的Microsoft Windows。使用--help来查看受支持的信号名称列表。

register模式把LightDB服务器注册为Microsoft Windows上的一个系统服务。-S选项允许选择服务启动类型，可以是“auto”（随系统自动启动）或“demand”（按需启动）。

unregister模式在Microsoft Windows上移除一个系统服务的注册。这会撤销register命令的效果。

## 选项

-c  
--core-files

在可行的平台上尝试允许服务器崩溃产生核心文件，方法是提升在核心文件上的任何软性资源限制。这通过允许从一个失败的服务器进程中获得一个栈跟踪而有助于调试或诊断问题。

-D datadir  
--pgdata=datadir

指定数据库配置文件的文件系统位置。如果这个选项被忽略，将使用环境变量LTDATA。

-l filename  
--log=filename

追加服务器日志输出到filename。如果该文件不存在，它会被创建。umask被设置成077，这样默认情况下不允许其他用户访问该日志文件。

-m mode  
--mode=mode

指定关闭模式。mode可以是smart、fast或immediate，或者这三者之一的第一个字母。如果这个选项被忽略，则fast是默认值。

-o options  
--options=options

指定被直接传递给lightdb命令的选项。-o可以被指定多次，所有给定的选项都会被传过去。

这些选项应该通常被单引号或双引号包围来确保它们被作为一个组传递。

-o initdb-options  
--options=initdb-options

指定要直接传递给initdb命令的选项。-o可以被指定多次，所有给定的选项都会被传过去。

这些选项应该通常被单引号或双引号包围来确保它们被作为一个组传递。

-p path

指定postgres可执行程序的位置。默认情况下，postgres可执行程序可以从lt\_ctl相同的目录得到，或者如果没有在那里找到，则在硬写的安装目录中获得。除非你正在做一些不同寻常的事并且得到错误说没有找到postgres可执行程序，这个选项不是必需的。

在init模式中，这个选项类似于指定了initdb可执行程序的位置。

-s  
--silent

只打印错误，不打印信息性的消息。

-t seconds  
--timeout=seconds

指定等待一个操作完成时要等待的最大秒数（见选项-w）。默认为LTCTLTIMEOUT环境变量的值，如果该环境变量没有设置则默认为60秒。

-V  
--version

打印lt\_ctl版本并退出。

-w  
--wait

等待操作完成。模式start、stop、restart、promote以及register支持这个选项，并且对那些模式是默认的。

在等待时，lt\_ctl会一遍又一遍地检查服务器的PID文件，在两次检查之间会休眠一小段时间。当PID文件指示该服务器已经做好准备接受连接时，启动操作被认为完成。当服务器移除PID文件时，关闭操作被认为完成。lt\_ctl会基于启动或关闭的成功与否返回一个退出代码。

如果操作在超时时间（见选项-t）内未能完成，则lt\_ctl会以一个非零退出状态退出。但是注意该操作可能会在后台继续进行并且最终取得成功。

-W  
--no-wait

不等待操作完成。这是选项-w的对立面。

如果禁用等待，所请求的动作会被触发，但是不会有关于其成功与否的反馈。在这种情况下，可能必须用服务器日志文件或外部监控系统来检查该操作的进度以及成功与否。

在以前版本的LightDB中，这是除stop模式之外的模式的默认选项。

-?  
--help

显示有关lt\_ctl命令行参数的帮助并退出。

如果一个指定的选项有效，但与选中的操作模式无关，则lt\_ctl会忽略它。

## 用于 Windows 的选项

-e source

作为一个 Windows 服务运行时，lt\_ctl用来在事件日志中记录日志的事件源的名称。默认是LightDB。注意这只控制由lt\_ctl本身发送的消息，一旦开始，服务器将使用event\_source参数中指定的事件源。如果服务器在启动时很早（在该参数被设置前）就失败，它可能也会使用默认的事件源名称 LightDB来记录。

-N servicename

要注册的系统服务的名称。这个名称将被用于服务名和显示名。默认是LightDB。

-P password

用于运行该服务的用户的口令。

-S start-type

要注册的系统服务的启动类型。启动类型可以是auto、demand或者两者之一的第一个字母。如果这个选项被忽略，则auto是默认值。

-U username

用于运行该服务的用户的用户名。对于域用户，使用格式DOMAIN\username。

## 环境

LTCTLTIMEOUT

等待启动或者关闭完成时要等待的默认秒数限制。如果没有设置，默认值是 60 秒。

LTDATA

默认的数据目录位置。

大部分的lt\_ctl模式都要求知道数据目录的位置，因此-D选项是必需的，除非LTDATA被设置。

和大部分其他LightDB工具相似，lt\_ctl也使用libpq（见Section 32.14 支持的环境变量）。

更多影响服务器的变量请见lightdb。

## 文件

postmaster.pid

lt\_ctl在数据目录中检查这个文件来判断服务器当前是否正在运行。

lightdb.opts

如果这个文件存在于数据目录中，lt\_ctl（处于restart模式中）将把该文件的内容作为选项传递给postgres，除非通过-o选项进行了覆盖。这个文件的内容也会被显示在status模式中。

---

## 例子

### 启动服务器

要启动服务器并且等到服务器接受连接：

```
$ lt_ctl start
```

要使用端口 5433 启动服务器并且运行时不使用fsync：

```
$ lt_ctl -o "-F -p 5433" start
```

### 停止服务器

要停止服务器，使用：

```
$ lt_ctl stop
```

-m选项允许控制服务器如何关闭：

```
$ lt_ctl stop -m smart
```

### 重启服务器

重启服务器几乎等价于停止服务器并且再次启动它，不过lt\_ctl默认会保存并重用被传递给之前的运行实例的命令行选项。要以和之前相同的选项重启服务器，使用：

```
$ lt_ctl restart
```

但是如果指定了-o，则会替换任何之前的选项。要使用端口 5433 重启并在重启时禁用fsync：

```
$ lt_ctl -o "-F -p 5433" restart
```

### 显示服务器状态

这里是lt\_ctl状态输出的例子：

```
$ lt_ctl status
```

```
lt_ctl: server is running (PID: 13718)
/usr/local/lightdb-x/bin/lightdb "-D" "/usr/local/lightdb-x/data" "-p" "5433" "-B" "128"
```

第二行是在重启模式可能被调用的命令行。

## 参见

initdb, lightdb

---

# lt\_resetwal

lt\_resetwal — 重置一个LightDB数据库集簇的预写式日志以及其他控制信息

## Synopsis

```
lt_resetwal [ -f | --force ] [ -n | --dry-run ] [option...] [ -D | --pgdata ]datadir
```

## 描述

lt\_resetwal会清除预写式日志（WAL）并且有选择地重置存储在lt\_control文件中的一些其他控制信息。如果这些文件已经被损坏，某些时候就需要这个功能。当服务器由于这样的损坏而无法启动时，这应该被用作最后的手段。

在运行这个命令之后，就可能可以启动服务器，但是记住数据库可能包含由于部分提交事务产生的不一致数据。你应当立刻转储你的数据、运行initdb并且重新载入。重新载入后，检查不一致并且根据需要修复之。

这个工具只能被安装服务器的用户运行，因为它要求对数据目录的读写访问。出于安全原因，你必须在命令行中指定数据目录。lt\_resetwal不使用环境变量LTDATA。

如果lt\_resetwal抱怨它无法为lt\_control决定合法数据，你可以通过指定-f（强制）选项强制它继续。在这种情况下，丢失的数据将被替换为看似合理的值。可以期望大部分域是匹配的，但是下一个OID、下一个事务ID和纪元、下一个多事务ID和偏移以及WAL开始位置域可能还是需要人工协助。这些域可以使用下面讨论的选项设置。如果你不能为所有这些域决定正确的值，-f还是可以被使用，但是恢复的数据库还是值得怀疑：一次立即的转储和重新载入是势在必行的。在你转储之前不要在该数据库中执行任何数据修改操作，因为任何这样的动作都可能使破坏更严重。

## 选项

-f  
--force

即使lt\_resetwal无法从lt\_control中确定有效的数据（如前面所解释的），也强迫lt\_resetwal继续运行。

-n  
--dry-run

-n/--dry-run选项指示lt\_resetwal打印从lt\_control重构出来的值以及要被改变的值，然后不修改任何东西退出。这主要是一个调试工具，但是可以用来在允许lt\_resetwal真正执行下去之前进行完整性检查。

-V  
--version

显示版本信息然后退出。

-?  
--help

显示帮助然后退出。

只有当lt\_resetwal无法通过读取lt\_control确定合适的值时，才需要下列选项。安全值可以按下文所述来确定。对于接收数字参数的值，可以使用前缀0x指定16进制值。



```
-c xid, xid
--commit-timestamp-ids=xid, xid
```

手工设置提交时间可以检索到的最老的和最新的事务 ID。

能检索到提交时间的最老事务 ID 的安全值（第一部分）可以通过在数据目录下 `lt_commit_ts` 目录中数字上最小的文件名来决定。反过来，能检索到提交时间的最新事务 ID 的安全值（第二部分）可以通过同一个目录中数字上最大的文件名来决定。文件名都是十六进制的。

```
-e xid_epoch
--epoch=xid_epoch
```

手工设置下一个事务 ID 的 epoch。

事务 ID 的 epoch 实际上并没有存储在数据库中的任何地方，除了被 `lt_resetwal` 设置在这个域中，所以只要关心的是数据库本身，任何值都可以用。你可能需要调整这个值来确保诸如 Slony-I 和 Skytools 之类的复制系统正确地工作 — 如果确实需要调整，应该可以从下游的复制数据库的状态中获得一个合适的值。

```
-l walfile
--next-wal-file=walfile
```

通过指定下一个 WAL 段文件名称来手工设置 WAL 开始位置。

下一个 WAL 段文件的名称应该比当前存在于数据目录下 `lt_wal` 目录中的任意 WAL 段文件名更大。这些名称也是十六进制的并且有三个部分。第一部分是“时间线 ID”并且通常应该被保持相同。例如，如果 `00000001000000320000004A` 是 `lt_wal` 中最大的项，则使用 `1 00000001000000320000004B` 或更高的值。

注意在使用非默认 WAL 段尺寸时，WAL 文件名中的数字与系统函数和系统视图报告的 LSN 不同。这个选项要的是 WAL 文件名而不是 LSN。

### Note

`lt_resetwal` 本身查看 `lt_wal` 中的文件并选择一个超出最新现存文件名的默认 `-l` 设置。因此，只有当你知道 WAL 段文件当前不在 `lt_wal` 中时，或者当 `lt_wal` 的内容完全丢失时，才需要对 `-l` 的手工调整，例如一个离线归档中的项。

```
-m mxid, mxid
--multixact-ids=mxid, mxid
```

手工设置下一个和最老的多事务 ID。

确定下一个多事务 ID（第一部分）的安全值的方法：在数据目录下的 `lt_multixact/offsets` 目录中查找最大的数字文件名，然后在它的基础上加一并且乘以 `65536` (`0x10000`)。反过来，确定最老的多事务 ID（`-m` 的第二部分）的方法：在同一个目录中查找最小的数字文件名并且乘以 `65536`。文件名是十六进制的数字，因此实现上述方法最简单的方式是以十六进制指定选项值并且追加四个零。

```
-o oid
--next-oid=oid
```

手工设置下一个 OID。

没有相对容易的方法来决定超过数据库中最大 OID 的下一个 OID。但幸运的是正确地得到下一个 OID 设置并不是决定性的。

`-O mxoff`  
`--multixact-offset=mxoff`

手工设置下一个多事务偏移量。

确定安全值的方法：查找数据目录下`lt_multixact/members`目录中最大的数字文件名，然后在它的基础上加一并且乘以 52352 (0xCC80)。文件名是十六进制数字。没有像其他选项那样追加零的简单方法。

`--wal-segsize=wal_segment_size`

设置新的WAL段尺寸，以兆字节为单位。这个值必须被设为2的1次幂和1024次幂（兆字节）之间。更多信息请参考`initdb`的相同选项。

### Note

虽然`lt_resetwal`将把WAL起始地址设置成超过最新的现有WAL段文件，但一些段尺寸的改变可能导致之前的WAL文件名被重用。如果WAL文件名重叠会导致归档策略出现问题，推荐把`-l`和这个选项一起使用来手动设置WAL起始地址。

`-u xid`  
`--oldest-transaction-id=xid`

手工设置最老的未冻结事务ID。

一个安全值，可以通过在数据目录下的`lt_xact`目录中查找数字最小的文件名然后乘以 1048576 (0x100000)的方式来确定。注意该文件名是十六进制的。以十六进制来指定选项值通常也是最简单的。例如，如果0007是`lt_xact`中最小的记录，`-u 0x700000`将有效(五个后补零提供适当的乘数)。

`-x xid`  
`--next-transaction-id=xid`

手工设置下一个事务 ID。

确定安全值的方法：在数据目录下的`lt_xact`目录中查找最大的数字文件名，然后在它的基础上加一并且乘以 1048576 (0x100000)。注意文件名是十六进制的数字。通常以十六进制的形式指定该选项值也是最容易的。例如，如果0011是`lt_xact`中的最大项，`-x 0x1200000`就可以（五个尾部的零就表示了前面说的乘数）。

## Environment

`PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为`always`、`auto`、`never`。

## 注解

这个命令不能在服务器正在运行时被使用。如果在数据目录中发现一个服务器锁文件，`lt_resetwal`将拒绝启动。如果服务器崩溃那么一个锁文件可能会被留下，在那种情况下你能移除该锁文件来让`lt_resetwal`运行。但是在你那样做之前，再次确认没有服务器进程仍然存活。

`lt_resetwal`仅能在具有相同主版本的服务器上使用。

## 另见

`lt_controldata`

---

# lt\_rewind

lt\_rewind — 把一个LightDB数据目录与另一个从该目录中复制出来的数据目录同步

## Synopsis

```
lt_rewind [option...] { -D | --target-pgdata } directory { --source-pgdata=directory | --source-server=connstr }
```

## 简介

lt\_rewind是用于在集簇的时间线分叉以后，同步一个 LightDB 集簇和同一集簇的另一份拷贝的工具。一种典型的场景是在失效后让一个旧的主服务器重新上线，同时有一个后备机跟着新的主机。

成功回放后，目标数据目录的状态类似于源数据目录的基本备份。与进行新的基本备份或使用rsync等工具不同，lt\_rewind不需要比较或复制集群中未更改的关系块。仅复制现有关系文件中更改的块；所有其他文件（包括新的关系文件、配置文件和WAL段）都将被完整复制。因此，当数据库很大并且集群之间只有一小部分块不同时，倒带操作比其他方法要快得多。

lt\_rewind检查源集簇和目标集簇的时间线历史来判断它们在哪一点分叉，并且期望在目标集簇的lt\_wal目录中找到 WAL 来返回到分叉点。分叉点可能会在目标时间线、源时间线或者它们的共同祖先上找到。在典型的失效场景中，目标集簇在分叉后很快就被关闭，这不是问题，但是如果目标集簇在分叉后已经运行了很长时间，旧的 WAL 文件可能已经不存在了。在这样的情况下，您可以手动将它们从WAL存档复制到lt\_wal目录，或使用-c选项运行lt\_rewind以自动从WAL存档检索它们。lt\_rewind的使用并不限于失效的场景，例如一个后备服务器可能被提升、运行一些写事务，然后被倒回再次成为一个后备。

在运行lt\_rewind之后，需要完成WAL重放以使数据目录处于一致状态。当目标服务器再次启动时，它将进入归档恢复，并从分歧点之前的最后一个检查点重放源服务器中生成的所有WAL。当lt\_rewind被运行时某些 WAL 在源服务器上不可用，并且因此无法被lt\_rewind会话所复制，则在目标服务器被启动时必须让这些 WAL 可用。这可以通过在目标数据目录中创建一个recovery.signal文件并且在lightdb.conf中配置适合的restore\_command来实现。

lt\_rewind要求目标服务器在lightdb.conf中启用了wal\_log\_hints选项，或者在用initdb初始化集簇时启用了数据校验。目前默认情况下这两者都没有被打开。full\_page\_writes也必须被设置为on，这是默认的。

### Warning

如果在处理时lt\_rewind失败，则目标的数据目录很可能不在可恢复的状态。在这种情况下，推荐创建一个新的备份。

由于 lt\_rewind 完全从源复制配置文件，因此可能需要在重新启动目标服务器之前更正用于恢复的配置，特别是当目标服务器作为源的备用服务器重新引入时。如果在倒带操作完成后重新启动服务器但未配置恢复，则目标可能会再次与主服务器分离。

如果lt\_rewind发现它无法直接写入的文件，它将立刻失败。例如当源服务器和目标服务器为只读的SSL密钥及证书使用相同的文件映射，就会发生这种情况。如果在目标服务器上存在这样的文件，推荐在运行lt\_rewind之前移除它们。在做了rewind之后，一些那样的文件可能已经被从源服务器拷贝，这样就有必要移除已经拷贝的数据并且恢复到rewind之前使用的链接集合。

## 选项

lt\_rewind接受下列命令行参数:

-D directory  
--target-pgdata=directory

这个选项指定要与源数据目录同步的目标数据目录。在运行lt\_rewind之前目标服务器必须被干净地关闭。

--source-pgdata=directory

指定要和目标服务器同步的源服务器的数据目录的文件系统路径。这个选项要求源服务器必须被干净地关闭。

--source-server=connstr

指定一个 libpq 连接串用于连接要与目标服务器同步的源LightDB服务器。连接必须是常规（非复制）连接，角色具有足够权限执行源服务器上lt\_rewind使用的函数（请参阅备注部分）或超级用户角色。这个选项要求源服务器正在运行并且接受连接。

-R  
--write-recovery-conf

创建standby.signal并将连接设置附加到输出目录中的lightdb.auto.conf中。--source-server对于此选项是必需的。

-n  
--dry-run

做除了实际修改目标目录之外的其他所有事情。

-N  
--no-sync

默认情况下，lt\_rewind将等待所有文件安全地写入磁盘。此选项会导致lt\_rewind不等待即可返回，这更快，但意味着后续操作系统崩溃会使同步数据目录损坏。通常情况，此选项可用于测试，但不应使用于生产安装。

-P  
--progress

启用进度报告。在从源集簇拷贝数据时，打开这个选项将会发送一个近似的进度报告。

-c  
--restore-target-wal

如果在lt\_wal目录中不再可用这些文件，请用在目标群集配置中定义的restore\_command从WAL存档中检索WAL文件。

--debug

打印冗长的调试输出，这主要对于调试lt\_rewind的开发者有用。

--no-ensure-shutdown

lt\_rewind 要求目标服务器在倒带之前彻底关闭。默认情况下，如果目标服务器没有完全关闭，lt\_rewind 会以单用户模式启动目标服务器，先完成崩溃恢复，然后将其停止。通过传递这个选项，如果服务器没有完全关闭，lt\_rewind 会跳过这个并立即出错。在这种情况下，用户应该自己处理这种情况。

- V  
--version  
显示版本信息然后退出。
- ?  
--help  
显示帮助然后退出。

## 环境

在使用--source-server选项时，lt\_rewind也使用libpq支持的环境变量（见Section 32.14）。

环境变量PG\_COLOR规定在诊断消息中是否使用颜色。可能的值为always、auto、never。

## 注解

当使用在线群集作为源执行lt\_rewind时，具有充足权限来执行lt\_rewind在源群集上使用的函数的角色可以用来代替超级用户。这里介绍如何创建这样的角色，在这里命名rewind\_user：

```
CREATE USER rewind_user LOGIN;
GRANT EXECUTE ON function pg_catalog.pg_ls_dir(text, boolean, boolean) TO
  rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_stat_file(text, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text, bigint, bigint,
  boolean) TO rewind_user;
```

当使用近期升级的在线群集作为源执行lt\_rewind时，必须在升级后执行CHECKPOINT以便其控制文件反映最新的时间线信息，lt\_rewind使用这些信息检查目标群集是否可以使用指定的源群集倒回。

## 如何工作

其基本思想是从源集簇拷贝所有文件系统级别的改变到目标集簇：

1. 以源集簇的时间线历史从目标集簇分叉出来的点之前的最后一个检查点为起点，扫描目标集簇的WAL日志。对于每一个WAL记录，读取每一个被动过的数据块。这会得到在目标集簇中从源集簇被分支出去以后所有被更改过的数据块列表。如果某些WAL文件不再可用，请尝试使用-c选项重新运行lt\_rewind以在WAL存档中搜索丢失的文件。
2. 使用直接的文件系统访问（--source-pgdata）或者SQL（--source-server），把所有那些更改过的块从源集簇拷贝到目标集簇。关系文件现在的状态相当于源和目标的WAL时间线偏离点之前最后一个完成的检查点的时刻加上偏离点之后目标上更改的任何块的源上的当前状态。
3. 将所有其他文件，包括新的关系文件、WAL段、lt\_xact和配置文件，从源集群复制到目标集群。与基本备份类似，从源集群复制的数据中省略了目录lt\_dynshmem/、lt\_notify/、lt\_replslot/、lt\_serial/、lt\_snapshots/、lt\_stat\_tmp/以及lt\_subtrans/的内容。文件backup\_label、tablespace\_map、lt\_internal.init、postmaster.opts以及postmaster.pid，以及任何以pgsql\_tmp开始的文件或目录都会被忽略。
4. 创建一个backup\_label文件，在故障转移时创建的检查点处开始WAL重放，并将lt\_control文件配置为最小一致性LSN，该LSN定义为从活动源回放时的pg\_current\_wal\_insert\_lsn()结果，或从停止的源回放时的最后一个检查点LSN。

5. 启动目标时，LightDB将重放所有必需的WAL，从而使数据目录处于一致状态。

---

# lt\_test\_fsync

lt\_test\_fsync — 为LightDB判断最快的 wal\_sync\_method

## Synopsis

```
lt_test_fsync [option...]
```

## 简介

lt\_test\_fsync是想告诉你在特定的系统上，哪一种 wal\_sync\_method最快，还可以在发生认定的 I/O 问题时提供诊断信息。不过，lt\_test\_fsync 显示的区别可能不会在真实的数据库吞吐量上产生显著的区别，特别是由于 很多数据库服务器被它们的预写日志限制了速度。lt\_test\_fsync为 wal\_sync\_method报告以微秒计的平均文件同步操作时间， 也能被用来提示用于优化commit\_delay值的方法。

## 选项

lt\_test\_fsync接受下列命令行选项：

-f  
--filename

指定要写入测试数据到其中的文件名。这个文件必须位于和 lt\_wal目录所在或者将被放置的同一个文件系统中（ lt\_wal包含WAL文件）。默认是当前 目录中的lt\_test\_fsync.out。

-s  
--secs-per-test

指定每次测试的秒数。每个测试的时间越长，测试的精度就越高，但是它需要更多时间来运行。默认是 5 秒，这允许程序在 2 分钟以内完成。

-V  
--version

打印lt\_test\_fsync版本并且退出。

-?  
--help

显示有关lt\_test\_fsync命令行参数的帮助并且退出。

## 环境

环境变量PG\_COLOR指定是否在诊断消息中使用颜色。可能的值为 always、auto 和 never.

## 另见

lightdb

---

# lt\_test\_timing

lt\_test\_timing — 度量计时开销

## Synopsis

lt\_test\_timing [option...]

## 描述

lt\_test\_timing是一种度量在你的系统上计时开销以及确认系统时间绝不会回退的工具。收集计时数据很慢的系统会给出不太准确的EXPLAIN ANALYZE结果。

## 选项

lt\_test\_timing接受下列命令行选项：

-d duration  
--duration=duration

指定测试的持续时间，以秒计。更长的持续时间会给出更好一些的精确度，并且更可能发现系统时钟回退的问题。默认测试持续时间是 3 秒。

-V  
--version

打印lt\_test\_timing版本并退出。

-?  
--help

显示有关lt\_test\_timing的命令行参数，然后退出。

## 用法

### 结果解读

好的结果将显示大部分 (>90%) 的单个计时调用用时都小于 1 微秒。每次循环的平均开销将会更低，低于 100 纳秒。下面的例子来自于使用了一份 TSC 时钟源码的 Intel i7-860 系统，它展示了非常好的性能：

```
Testing timing overhead for 3 seconds.  
Per loop time including overhead: 35.96 ns  
Histogram of timing durations:  
  < us   % of total   count  
    1    96.40465   80435604  
    2     3.59518   2999652  
    4     0.00015     126  
    8     0.00002     13  
   16     0.00000     2
```

注意每次循环时间和柱状图用的单位是不同的。循环的解析度可以在几个纳秒 (ns)，而单个计时调用只能解析到一个微秒 (us)。



## 度量执行器计时开销

当查询执行器使用EXPLAIN ANALYZE运行一个语句时，单个操作会被计时，总结也会被显示。你的系统的负荷可以通过使用ltsql程序计数行来检查：

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

i7-860 系统测到运行该计数查询用了 9.8 ms 而EXPLAIN ANALYZE版本则需要 16.6 ms，每次处理都在 100,000 行上进行。6.8 ms 的差别意味着在每行上的计时负荷是 68 ns，大概是 lt\_test\_timing 估计的两倍。即使这样相对少量的负荷也造成了带有计时的计数语句耗时多出了 70%。在更大量的查询上，计时开销带来的问题不会有这么明显。

## 改变时间来源Changing time sources

在一些较新的 Linux 系统上，可以在任何时候更改用来收集计时数据的时钟来源。第二个例子显示了在上述快速结果的相同系统上切换到较慢的 acpi\_pm 时间源可能带来的降速：

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/
current_clocksource
# lt_test_timing
Per loop time including overhead: 722.92 ns
Histogram of timing durations:
  < us    % of total    count
    1     27.84870    1155682
    2     72.05956    2990371
    4      0.07810      3241
    8      0.01357       563
   16      0.00007         3
```

在这种配置中，上面的例子EXPLAIN ANALYZE用了 115.9 ms。其中有 1061 ns 的计时开销，还是用这个工具直接度量结果的一个小倍数。这么多的计时开销意味着实际的查询本身只占了时间的一个很小的分数，大部分的时间都耗在了计时所需的管理开销上。在这种配置中，任何涉及到很多计时操作的EXPLAIN ANALYZE都会受到计时开销的显著影响。

FreeBSD 也允许即时更改时间源，并且它会记录在启动期间有关计时器选择的信息：

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

其他系统可能只允许在启动时设定时间源。在旧的 Linux 系统上，“clock”内核设置是做这类更改的唯一方法。并且即使在一些更近的系统上，对于一个时钟源你将只能看到唯一的选项“jiffies”。Jiffies 是老的 Linux 软件时钟实现，当有足够快的计时硬件支持时，它能够具有很好的解析度，就像在这个例子中：

```

$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ lt_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  < us    % of total    count
    1     90.23734   27694571
    2      9.75277    2993204
    4      0.00981     3010
    8      0.00007      22
   16      0.00000      1
   32      0.00000      1

```

## 时钟硬件和计时准确性

收集准确的计时信息在计算机上通常是使用具有不同精度的时钟硬件完成的。使用一些硬件，操作系统能几乎直接把系统时钟时间传递给程序。一个系统时钟也可以得自于一块简单地提供计时中断、在某个已知时间区间内的周期性滴答的芯片。在两种情况中，操作系统内核提供一个隐藏这些细节的时钟源。但是时钟源的精确度以及能多快返回结果会根据底层硬件而变化。

不精确的计时能够导致系统不稳定性。对任何时钟源的更改都要仔细地测试。操作系统默认是有时会更倾向于可靠性而不是最好的精确性。并且如果你在使用一个虚拟机器，应查看与之兼容的推荐时间源。在模拟计时器时虚拟硬件面临着额外的困难，并且提供商常常会建议每个操作系统的设置。

时间戳计数器（TSC）时钟源是当前一代 CPU 上最精确的一种。当操作系统支持 TSC 并且 TSC 可靠时，它是跟踪系统时间更好的方式。有多种方式会使 TSC 无法提供准确的计时源，这会让它不可靠。旧的系统能有一种基于 CPU 温度变化的 TSC 时钟，这让它不能用于计时。尝试在一些就的多核 CPU 上使用 TSC 可能在多个核心之间给出不一致的时间报告。这可能导致时间倒退，这个程序会检查这种问题。并且即使最新的系统，在非常激进的节能配置下也可能无法提供准确的 TSC 计时。

更新的操作系统可能检查已知的 TSC 问题并且当它们被发现时切换到一种更慢、更稳定的时钟源。如果你的系统支持 TSC 时间但是并不默认使用它，很可能是由于某种充分的理由才禁用它。某些操作系统可能无法正确地检测所有可能的问题，或者即便在知道 TSC 不精确的情况下也允许使用 TSC。

如果系统上有高精度事件计时器（HPET）并且 TSC 不准确，该系统将会更喜欢 HPET 计时器。计时器芯片本身是可编程的，最高允许 100 纳米的解析度，但是在你的系统时钟中可能见不到那么高的准确度。

高级配置和电源接口（ACPI）提供了一种电源管理（PM）计时器，Linux 把它称之为 `acpi_pm`。得自于 `acpi_pm` 的时钟最好时将能提供 300 纳秒的解析度。

在旧的 PC 硬件上使用的计时器包括 8254 可编程区间计时器（PIT）、实时时钟（RTC）、高级可编程中断控制器（APIC）计时器以及 Cyclone 计时器。这些计时器是以毫秒解析度为目标的。

## 参见

EXPLAIN

---

# lt\_upgrade

lt\_upgrade — 升级LightDB服务器实例

## Synopsis

```
lt_upgrade -b oldbindir [-B newbindir] -d oldconfigdir -D newconfigdir [option...]
```

## 描述

lt\_upgrade (之前被称为pg\_migrator) 允许存储在LightDB数据文件中的数据被升级到一个较晚的LightDB主版本而无需进行主版本升级 (例如从9.5.8到9.6.4或者从10.7到11.2) 通常所需的数据转储/重载。对于次版本升级 (例如从9.6.2到9.6.3或者从10.1到10.2) 则不需要这个程序。

LightDB主发行版本通常会加入新的特性, 这些新特性常常会更改系统表的布局, 但是内部数据存储格式很少会改变。lt\_upgrade 使用这一事实来通过创建新系统表并且重用旧的用户数据文件来执行快速升级。如果未来的主发行版本更改了数据存储格式, 导致旧数据格式不可读, 那么 lt\_upgrade将无法用于此类升级。(社区将努力避免这种情况)。

lt\_upgrade会尽力 (例如通过检查兼容的编译时设置) 确保新旧集簇在二进制上也是兼容的, 包括 32/64 位二进制。保持外部模块也是二进制兼容的也很重要, 不过 lt\_upgrade无法检查这一点。

lt\_upgrade 支持从 8.4.X 及其后版本升级到当前的 LightDB主发布, 包括快照和 beta 发布。

## 选项

lt\_upgrade接受下列命令行参数:

-b bindir

--old-bindir=bindir

旧的 LightDB 可执行文件目录; 环境变量LTBINOLD

-B bindir

--new-bindir=bindir

新的 LightDB 可执行文件目录; 默认为lt\_upgrade所在的目录; 环境变量LTBINNEW

-c

--check

只检查集簇, 不更改任何数据

-d configdir

--old-datadir=configdir

旧的集簇数据目录; 环境变量 LTDATAOLD

-D configdir

--new-datadir=configdir

新的集簇数据目录; 环境变量 LTDATANEW

`-j`  
`--jobs=njobs`  
要同时使用的进程或线程数

`-k`  
`--link`  
使用硬链接来代替将文件拷贝到新集簇

`-o options`  
`--old-options options`  
直接传送给旧 `lightdb`命令的选项，多个选项可以追加在后面

`-O options`  
`--new-options options`  
直接传送给新 `lightdb`命令的选项，多个选项可以追加在后面

`-p port`  
`--old-port=port`  
旧的集簇端口号；环境变量 `LTPORTOLD`

`-P port`  
`--new-port=port`  
新的集簇端口号；环境变量 `LTPORTNEW`

`-r`  
`--retain`  
即使在成功完成后也保留 `SQL` 和日志文件

`-s dir`  
`--socketdir=dir`  
用于升级期间 `postmaster`套接字的目录；默认是当前目录；环境变量 `LTSOCKETDIR`

`-U username`  
`--username=username`  
集簇的安装用户名；环境变量 `LTUSER`

`-v`  
`--verbose`  
启用详细的内部日志

`-V`  
`--version`  
显示版本信息，然后退出

`--clone`  
使用有效的文件克隆（在一些系统上也被称为“`reflinks`”），而不是将文件拷贝到新群集。这可以导致数据文件接近瞬时的复制，从而获得 `-k/--link`的速度优势，同时保留旧群集不受影响。

文件克隆仅在某些操作系统和文件系统上得到支持。如果选中但不被支持，则 `lt_upgrade`运行将会出错。目前，它支持在Linux（内核4.5或更高版本）上的 `Btrfs`和 `XFS`（在文件系统创建 `reflink`支持），以及 `macOS`上的 `APFS`。

-?  
--help

显示帮助，然后退出

## 使用

下面是用lt\_upgrade执行一次升级的步骤：

### 1. 移动旧集簇（可选）

需要移动当前的 LightDB 安装目录，以便不会干扰新的 LightDB 安装。一旦当前的 LightDB 服务器关闭，就可以安全地重命名 LightDB 安装目录。假设旧目录为 /usr/local/lightdbx，则可以执行以下操作：

```
mv /usr/local/lightdbx /usr/local/lightdbx.old
```

来重命名该目录。

### 2. 对于源码安装，编译新版本

用兼容旧集簇的configure标记编译新的 LightDB 源码。在开始升级之前，lt\_upgrade将检查lt\_controldata来确保所有设置都是兼容的。

### 3. 安装新的 LightDB 二进制文件

安装新服务器的二进制文件和支持文件。lt\_upgrade 会被包含在默认的安装中。

### 4. 初始化新的 LightDB 集簇

如果您的 LightDB 是独立的，请使用 lt\_initdb 初始化新的集群。同样，使用与旧集群匹配的兼容 lt\_initdb 标志，然后添加另一个选项 --upgrade-mode。无需启动新集群。

```
lt_initdb -p 5432 -D /home/lightdb/lightdbx/data --upgrade-mode
```

如果您的 LightDB 部署在高可用性或分布式环境中，请使用 lightdb-installer 初始化新的集群。您必须修改 shell 脚本 script/8\_lightdb\_create\_extension.sh，在启动 install.sh 之前删除其中所有的 exeuteSQL 命令。

### 5. 安装扩展共享对象文件

许多扩展和自定义模块，无论是来自contrib或其他源，使用共享对象文件(或DLLs)，例如，pgcrypto.so。如果旧集群使用过这些，匹配新服务器二进制的共享对象文件，必须安装在新集群中。通常是通过操作系统命令。不要加载模式定义，例如CREATE EXTENSION pgcrypto，因为这些将从旧集群复制。如果扩展更新是可用的，lt\_upgrade将报告这一点，并创建一个脚本，可以稍后运行来更新它们。

您不需要担心内置扩展，LightDB 会自动处理它们。

### 6. 调整认证

lt\_upgrade将会多次连接到旧服务器和新服务器，因此 你可能想要在lt\_hba.conf中把认证设置成 peer或者使用一个~/.pgpass文件（见 Section 32.15）。

### 7. 停止两个服务器

确认两个数据库服务器都被停止使用，例如在 Unix 上可以：

```
lt_ctl -D /opt/LightDB/13 stop
lt_ctl -D /opt/LightDB/13 stop
```

直到后面的步骤之前，流复制和日志传送后备服务器可以保持运行。

## 8. 为后备服务器升级做准备

如果正在使用小节Step 10中给出的方法升级后备服务器，请对旧的主集簇和后备集簇运行lt\_controldata以验证旧的后备服务器已经完全追上。验证“Latest checkpoint location”值在所有集簇中都匹配（如果旧后备服务器在旧的主服务器之前被关闭或者如果旧的后备服务器仍在运行，则将会出现失配）。此外，请确保在新的主集簇上的lightdb.conf文件中，wal\_level未设置为minimal。

## 9. 运行 lt\_upgrade

总是应该运行新服务器而不是旧服务器的lt\_upgrade二进制文件。lt\_upgrade要求制定新旧集簇的数据和可执行文件(bin)目录。你也可以指定用户和端口值，以及你是否想要用链接或克隆来取代默认的复制行为对数据文件进行处理。

如果你使用链接模式，升级将会快很多（不需要文件拷贝）并且将使用更少的磁盘空间，但是在升级后一旦启动新集簇，旧集簇就无法被访问。链接模式也要求新旧集簇数据目录位于同一个文件系统中（表空间和lt\_wal可以在不同的文件系统中）。克隆模式提供了相同的速度以及磁盘空间优势，但不会导致新群集启动后旧群集不可用。克隆模式还需要新旧数据目录位于同一文件系统中。此模式仅在某些操作系统和文件系统上可用。

--jobs选项允许多个CPU核心被用来复制/链接文件以及并行地转储和重载数据库模式。这个选项一个比较好的值是CPU核心数和表空间数的最大值。这个选项可以显著地减少升级运行在一台多处理器机器上的多数据库服务器的时间。

对于Windows用户，您必须以管理员账户登录，然后以postgres用户身份启动shell并设置正确的路径：启动后，lt\_upgrade将验证两个集群是否兼容，然后进行升级。即使旧服务器仍在运行，您也可以使用lt\_upgrade --check仅执行检查。lt\_upgrade --check还将概述升级后需要进行的任何手动调整。如果您将使用链接或克隆模式，则应使用选项--link或--clone以及--check以启用特定于模式的检查。lt\_upgrade要求在当前目录中具有写权限。

显然，没有人可以在升级期间访问这些集簇。lt\_upgrade默认会在端口50432上运行服务器来避免意外的客户端连接。在做升级时，可以对两个集簇使用相同的端口号，因为新旧集簇不会在同时被运行。不过，在检查一个旧的运行中服务器时，新旧端口号必须不同。

如果在恢复数据库模式时发生错误，lt\_upgrade将会退出并且你必须按照下文Step 16中所说的恢复旧集簇。要再次尝试lt\_upgrade，你将需要修改旧集簇，这样lt\_upgrade模式会成功恢复。如果问题是一个contrib模块，你可能需要从旧集簇中卸载该模块并且在升级后重新把它安装在新集簇中，不过这样做的前提是该模块没有被用来存储用户数据。

## 10. 升级流复制和日志传送后备服务器

如果使用链接模式并且有流复制（见Section 24.2.5或者日志传送（见Section 24.2 后备服务器，你可以遵照下面的步骤对它们进行快速的升级。你将不用在这些后备服务器上运行lt\_upgrade，而是在主服务器上运行rsync。到这里还不要启动任何服务器。

如果你没有使用链接模式、没有或不想使用rsync或者想用一种更容易的解决方案，请跳过这一节中的过程并且在lt\_upgrade完成并且新的主集簇开始运行后重建后备服务器。

- a. 在后备服务器上安装新的 LightDB 二进制文件  
确保新的二进制和支持文件被安装在所有后备服务器上。
- b. 确保不存在新的后备机数据目录  
确保新的后备机数据目录不存在或者为空。如果 运行过initdb，请删除后备服务器的新数据目录。
- c. 安装扩展共享对象文件  
在新的后备机上安装和新的主集簇中相同的扩展共享对象文件。
- d. 停止后备服务器  
如果后备服务器仍在运行，现在使用上述的指令停止它们。
- e. 保存配置文件  
从旧后备机的配置目录保存任何需要保留的配置文件，例如 lightdb.conf（以及它包含的任何文件）、lightdb.auto.conf、lt\_hba.conf，因为这些文件在下一步中会被重写或者移除。
- f. 运行rsync

在使用链接模式时，后备服务器可以使用rsync快速升级。为了实现这一点，在主服务器上一个高于新旧数据库集簇目录的目录中为每个后备服务器运行这个命令：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive  
old_cluster new_cluster remote_dir
```

其中old\_cluster和new\_cluster是相对于主服务器上的当前目录的，而remote\_dir是后备服务器上高于新旧集簇目录的一个目录。在主服务器和后备服务器上指定目录之下的目录结构必须匹配。指定远程目录的详细情况请参考rsync的手册，例如：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /  
opt/LightDB/22.1 \  
/opt/LightDB/22.2 standby.example.com:/opt/LightDB
```

可以使用rsync的--dry-run选项验证该命令将做的事情。虽然在主服务器上必须为至少一台后备运行rsync，可以在一台已经升级过的后备服务器上运行rsync来升级其他的后备服务器，只要已升级的后备服务器还没有被启动。

这个命令所做的事情是记录由lt\_upgrade的链接模式创建的链接，它们连接主服务器上新旧集簇中的文件。该命令接下来在后备服务器的旧集簇中寻找匹配的文件并且为它们在该后备的新集簇中创建链接。主服务器上没有被链接的文件会被从主服务器拷贝到后备服务器（通常都很小）。这提供了快速的后备服务器升级。不幸的是，rsync会不必要地拷贝与临时表和不做日志表相关的文件，因为通常在后备服务器上不存在这些文件。

如果有表空间，你将需要为每个表空间目录运行一个类似的rsync命令，例如：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /  
home/lightdbx/LT_22.1_201510051 \  

```

```
/home/lightdbx/LT_22.2_201608131 standby.example.com:/home/
lightdbx
```

如果你已经把lt\_wal放在数据目录外面，也必须在那些目录上运行rsync。

g. 配置流复制和日志传送后备服务器

为日志传送配置服务器（不需要运行pg\_start\_backup() 以及pg\_stop\_backup() 或者做文件系统备份，因为从属机 仍在与主机同步）。

11. 恢复 lt\_hba.conf

如果你修改了lt\_hba.conf，则要将其恢复到原始的设置。 也可能需要调整新集簇中的其他配置文件来匹配旧集簇，例如 lightdb.conf（以及它包含的任何文件）和 lightdb.auto.conf。

12. 启动新服务器

现在可以安全地启动新的服务器，并且可以接着启动任何 rsync过的后备服务器。

13. 升级后处理

如果需要做任何升级后处理，lt\_upgrade 将在完成后发出警告。它也将 生成必须由管理员运行的脚本文件。这些脚本文件将连接到每一个需要做 升级后处理的数据库。每一个脚本应该这样运行：

```
ltsql --username=lightdb --file=script.sql postgres
```

这些脚本可以以任何顺序运行并且在运行之后立即删除。

### Caution

通常在重建脚本运行完成之前访问重建脚本中引用的表是不安全的，这样做 可能会得到不正确的结果或者很差的性能。没有在重建脚本中引用的表可以 随时被访问。

14. 统计信息

由于lt\_upgrade并未传输优化器统计信息，在升级的尾声 你将被指示运行一个命令来生成这些信息。你可能需要设置连接参数来匹配你 的新集簇。

15. 删除旧集簇

一旦你对升级表示满意，你就可以通过运行 lt\_upgrade完成时提到的脚本来删除旧集簇的 数据目录（如果在旧数据目录中有用户定义的表空间就不可能实现自动删除）。你也可以删除旧安装目录（例如bin、share）。

16. 恢复到旧集簇

在运行lt\_upgrade之后，如果你希望恢复到 旧集簇，有几个选项：

- 如果使用了 --check 选项，则旧集群没有被修改；它可以被重新启动。
- 如果 --link 选项 没有被使用，旧集群没有被修改；它可以被重新启动。
- 如果使用了--link 选项，数据文件可能在新旧群集之间共享：
  - 如果lt\_upgrade在链接启动之前中止，旧群集没有被修改，它可以重新启动。



- 如果你没有启动新集群，旧集群没有被修改，当链接启动时，一个.old后缀会附加到\$LTDATA/global/lt\_control。如果要重用旧集群，从\$LTDATA/global/lt\_control移除.old后缀；你就可以重启旧集群。
- 如果你已经启动新群集，它已经写入了共享文件，并且使用旧群集会不安全。这种情况下，需要从备份中还原旧群集。

## 注解

lt\_upgrade创建不同的工作文件，如模式转储，在当前工作目录中。为了安全，请确保该目录不可被任何其他用户读取或者写入。

lt\_upgrade在新旧数据目录中启动短期的postmasters。临时 Unix 套接字文件用于与这些postmasters通信，默认情况下，在当前工作目录中进行。在某些情况下，当前目录的路径名称可能太长，无法成为有效的套接字名称。这种情况下你可以使用-s选项将套接字文件放在某些具有较短路径名称的目录中。为了安全原因，请确保该目录不可被任何其他用户读取或者写入。（这在 Windows 上不受支持。）

如果失败、重建和重索引会影响你的安装，lt\_upgrade 将会报告这些情况。用来重建表和索引的升级后脚本将会自动被建立。如果你正在尝试自动升级很多集簇，你应该发现具有相同数据库模式的集簇 对所有集簇升级都要求同样的升级后步骤，这是因为升级后步骤是基于数据库模式而不是用户数据。

LightDB 内置了扩展和计划任务。这些扩展和任务在 lt\_extension.dat 和 lt\_cron.dat 中有描述。您无法修改这两个文件，因为它们由 LightDB 维护。lt\_upgrade 使用这两个文件来升级和更新内置扩展和计划任务。从 LightDB 22.3 开始，lt\_extension.dat 和 lt\_cron.dat 合并为一个文件 lt\_builtin.dat。但是，lt\_extension.dat 和 lt\_cron.dat 仍然被保留。

从 LightDB 22.3 开始，lt\_upgrade 将检查 lightdb\_syntax\_compatible\_type 的一致性。旧集群的值必须与新集群的值相同。

从 LightDB 22.3 开始，lt\_upgrade 将在升级后为新集群创建名为 lt\_test 的数据库。

从 LightDB 22.3 开始，当 lightdb\_syntax\_compatible\_type 的值为 'oracle' 时，空字符串将作为 null 值存储。因此，如果您正在从 22.2 升级到 22.3，且兼容性都为 'oracle'，不建议使用 lt\_upgrade。

对于部署测试，创建一个只有模式的旧集簇副本，在其中插入假数据并且升级。

lt\_upgrade不支持包含使用这些reg\* OID-引用 系统数据类型的表列的数据库的升级：

```
regcollation
regconfig
regdictionary
regnamespace
regoper
regoperator
regproc
regprocedure
```

(regclass, regrole, and regtype can be upgraded.)

如果你想要使用链接模式并且你不想让你的旧集簇在新集簇启动时被修改，考虑使用克隆模式。如果(克隆模式)不可用，可以复制一份旧集簇并且在副本上以链接模式进行升级。要创建旧集簇的一份合法拷贝，可以在服务器运行时使用rsync创建旧集簇的一份脏拷贝，然后关闭旧服务器并且再次运行rsync --checksum 把更改更新到该拷贝以让其一致（--checksum是必要的，因为 rsync在判断文件修改时间的更改时的精度只能到秒级）。如 Section 23.3.8中所述，你可能想要排除一些文件，例如postmaster.pid。如果你的文件系

统支持文 件系统快照或者 copy-on-write 文件副本，你可以使用它们来创建旧集簇和 表空间的一个备份，不过快照和副本必须被同时创建或者在数据库服务器关闭 期间被创建。

## 另见

initdb, lt\_ctl, lt\_dump, lightdb

---

# lt\_upgrade\_check

lt\_upgrade\_check — lt\_upgrade升级前检查工具

## Synopsis

```
lt_upgrade_check -b oldbindir [-B newbindir] -d oldconfigdir -D newconfigdir  
[option...]
```

## 描述

本程序在lt\_upgrade升级之前执行，用于提前检查升级不兼容及异常项，尽量避免在升级过程中出现失败。

目前支持的检查内容如下：

1, 关键字检查。

2, 存储空间检查，仅当空闲存储空间超过原目录大小乘以 $(100 + \text{reserved-space-percent})\%$ 的时候才允许升级，尽量避免升级过程中储存空间不足，其中reserved-space-percent是选项参数，下文有具体介绍。

## 选项

lt\_upgrade\_check接受下列命令行参数：

-b bindir  
--old-bindir=bindir

旧的 LightDB 可执行文件目录； 环境变量LTBINOLD。

-B bindir  
--new-bindir=bindir

新的 LightDB 可执行文件目录； 默认为lt\_upgrade\_check所在的目录； 环境变量LTBINNEW。

-d configdir  
--old-datadir=configdir

旧的集簇数据目录； 环境变量 LTDATAOLD。

-D configdir  
--new-datadir=configdir

新的集簇数据目录； 环境变量 LTDATANEW。

-p port  
--old-port=port

旧的集簇端口号； 环境变量 LTPORTOLD。

-P port  
--new-port=port

新的集簇端口号； 环境变量 LTPORTNEW。

- `-r`  
`--reserved-space-percent`
- 剩余磁盘空间百分比，计算新目录所需空间时规则是原目录大小乘以  $(1 + \text{reserved-space-percent})\%$ ，默认 20，即剩余磁盘空间要大于原目录大小\*120%。
- `-s`  
`--strict`
- 采用严格模式检查，默认不开启。若开启此选项，则会检查表名、列名、函数名等对象名称，并检查函数、存储过程、视图、等编译式要素的代码中是否用了任何关键字，若不开启，则只检查增量关键字（增量关键字是指新旧数据库实例之间新增的关键字），如果包含了关键字，则视为检查失败。
- `-v`  
`--verbose`
- 启用详细的内部日志。
- `-V`  
`--version`
- 显示版本信息，然后退出。
- `-?`  
`--help`
- 显示帮助，然后退出。

## 使用

下面是在新库的数据库实例调用 `lt_upgrade_check` 执行一次检查的命令：

```
new-bindir/lt_upgrade_check -d old-datadir -D new-datadir -b old-bindir -B
new-bindir -v -r 50
```

这个命令表示在指定的旧库数据库实例升级到新库数据库实例时，关键字检查是非严格检查，需要剩余磁盘空间大于原目录大小乘以150%，并打印详细检查日志。

- 运行 `lt_upgrade_check`

运行 `lt_upgrade_check` 前，要求新旧数据库实例都是在同一台服务器上并都是停止状态，没有人可以在检查期间访问这些实例。默认会在端口 50432 上运行服务来避免意外的客户端连接。在做检查时，可以对两个实例使用相同的端口号，因为新旧实例不会同时被运行。

检查流程是：获取新库、旧数据库实例关键字信息后，首先做关键字检查，然后进行存储空间检查，最后在新库调用 `lt_upgrade --check` 命令，进行进一步检查。

---

# lt\_waldump

lt\_waldump — 以人类可读的形式显示一个LightDB 数据库集簇的预写式日志

## Synopsis

```
lt_waldump [option...] [startseg [endseg]]
```

## 简介

lt\_waldump显示预写式日志（WAL），它主要 用于调试或者教育目的。

这个工具只能由安装该服务器的用户运行，因为它要求对数据目录的只读访问。

## 选项

下列命令行选项控制输出的位置和格式：

startseg

从指定的日志段文件开始读取。这也隐含地决定了要搜索文件的路径以及 要使用的时间线。

endseg

在读取指定的日志段文件后停止。

-b

--bkp-details

输出有关备份块的细节。

-e end

--end=end

在指定的WAL位置停止读取，而不是一直读取到日志流的末尾。

-f

--follow

在到达可用 WAL 的末尾之后，保持每秒轮询一次是否有新的 WAL 出现。

-n limit

--limit=limit

显示指定数量的记录，然后停止。

-p path

--path=path

指定搜索日志段文件的目录或包含这些文件的包含lt\_wal子目录的目录。缺省值是在当前目录中搜索，当前目录的lt\_wal子目录和 LTDATA的lt\_wal子目录。

-q

--quiet

除错误外，不要打印任何输出。当您想知道一系列WAL记录是否可以成功解析但不关心记录内容时，此选项非常有用。

```
-r rmgr
--rmgr=rmgr
```

只显示由指定资源管理器生成的记录。如果把list作为资源管理器名称 传递给这个选项，则打印出可用资源管理器名称的列表然后退出。

```
-s start
--start=start
```

要从哪个WAL位置开始读取。默认是从找到的最早的文件的第一个可用日志记录开始。

```
-t timeline
--timeline=timeline
```

要从哪个时间线读取日志记录。默认是使用startseg（如果指定） 中的值，否则默认为1。

```
-V
--version
```

打印lt\_waldump版本并且退出。

```
-x xid
--xid=xid
```

只显示用给定事务 ID 标记的记录。

```
-z
--stats[=record]
```

显示概括统计信息（记录的数量和尺寸以及全页镜像）而不是显示 每个记录。可以选择针对每个记录生成统计信息，而不是针对每个 资源管理器生成。

```
-?
--help
```

显示有关lt\_waldump命令行参数的帮助并且退出。

## Environment

LTDATA

数据目录；另请参阅 -p 选项。

PG\_COLOR

规定在诊断消息中是否使用颜色。可能的值为always、 auto、 never。

## 注解

当服务器正在运行时可能会给出错误的结果。

只有指定的时间线 会被显示（如果没有指定，则显示默认时间线）。 其他时间线上的记录会被忽略。

lt\_waldump不能读取具有后缀.partial 的 WAL 文件。如果需要读取那些文件，需要从文件名中移除 .partial后缀。

## 另见

Section 27.5

---

# lightdb

lightdb — LightDB数据库服务器

## Synopsis

lightdb [选项...]

## 描述

lightdb是LightDB数据库服务器。一个客户端应用为了能访问一个数据库，它会（通过网络或者本地）连接到一个运行着的lightdb实例。该lightdb实例接着会开始一个独立的服务器进程来处理该连接。

一个lightdb实例总是管理正好一个数据库集簇的数据。一个数据库集簇是一个数据库的集合，它们被存储在一个共同的文件系统位置（“数据区”）上。一个系统上可以同时运行多个lightdb实例，只要它们使用不同的数据区和不同的通信端口（见下文）。lightdb启动时需要知道数据区的位置，该位置必须通过-D选项或LTDATA环境变量指定，对此是没有默认值的。通常，-D或LTDATA会直接指向由initdb创建的数据区目录。其他可能的文件布局在Section 17.2中讨论。

默认情况下，lightdb会在前台启动并将日志消息打印到标准错误流。但在实际应用中，lightdb应当作为一个后台进程启动，而且多数是在系统启动时自动启动。

lightdb还能在单用户模式中被调用。这种模式的主要用途是在启动过程中由initdb使用。有时候它也被用于调试或者灾难性恢复。注意，运行一个单用户模式服务器并不真地适合调试服务器，因为不会发生实际的进程间通信和锁定。当从 shell 中调用单用户模式时，用户可以输入查询并且结果会被以一种更适合开发者阅读（不适合普通用户）的形式打印在屏幕上。在单用户模式中，会话用户将被设置为 ID 为 1 的用户，并且这个用户会被隐式地赋予超级用户权限。该用户不必实际存在，因此单用户模式运行可以被用来对某些意外损坏的系统目录进行手工恢复。

## 选项

lightdb接受下列命令行参数。关于这些选项的详细讨论请参考Chapter 17 你也可以通过设置一个配置文件来减少键入大部分这些选项。有些（安全）选项还可以从连接的客户端以一种与应用相关只应用于会话的方法设置。例如，如果设置了PGOPTIONS环境变量，那么基于libpq的客户端将都把那个字符串传递给服务器，它将被服务器解释成lightdb命令行选项。

## 通用选项

-B nbuffers

设置被服务器进程使用的共享内存缓冲区数量。这个参数的默认值是initdb自动选择的。指定这个选项等效于设置shared\_buffers配置参数。

-c name=value

设置一个命名的运行时参数。LightDB支持的配置参数在Chapter 17中描述。大多数其它命令行选项实际上都是这种参数赋值的短形式。-c可以出现多次用于设置多个参数。

-C name

打印命名运行时参数的值，并且退出（详见上面的-c选项）。这可以被用于在一个运行服务器上，并且从lightdb.conf中返回值，这些值可能被在这次调用中的任何参数修改过。它并不反映集簇启动时提供的参数。

这个选项用于与一个服务器实例交互的其他程序来查询配置参数值，例如lt\_ctl。面向用户的应用应该使用SHOW or the pg\_settings视图。

-d debug-level

设置调试级别。数值设置得越高，写到服务器日志的调试输出就越多。取值范围是从 1 到 5。还可以针对某个特定会话使用-d 0来阻止父lightdb进程的服务器日志级别被传播到这个会话。

-D datadir

指定数据库配置文件的文件系统位置。详见Section 17.2

-e

把默认日期风格设置为“European”，也就是输入日期域的顺序是DMY。这也导致在一些日期输出格式中把日打印在月之前。详见Section 8.5

-F

禁用fsync调用以提高性能，但是要冒系统崩溃时数据损坏的风险。指定这个选项等效于禁用fsync配置参数。在使用之前阅读详细文档！

-h hostname

指定lightdb监听来自客户端应用 TCP/IP 连接的 IP 主机名或地址。该值也可以是一个用逗号分隔的地址列表，或者\*表示监听所有可用的地址。一个空值表示不监听任何 IP 地址，在这种情况下可以使用 Unix 域套接字连接到服务器。缺省只监听localhost。声明这个选项等效于设置listen\_addresses配置参数。默认只监听localhost。指定这个选项等效于设置listen\_addresses配置参数。

-i

允许远程客户端使用 TCP/IP（互联网域）连接。没有这个选项，将只接受本地连接。这个选项等效于在lightdb.conf中或者通过-h选项将listen\_addresses设为\*。

这个选项已经被废弃，因为它不允许访问listen\_addresses的完整功能。所以最好直接设置listen\_addresses。

-k directory

指定lightdb用来监听来自客户端应用连接的 Unix 域套接字的目录。这个值也可以是一个逗号分隔的目录列表。一个空值指定不监听任何 Unix 域套接字，在这种情况下只能用 TCP/IP 套接字来连接到服务器。默认值通常是/tmp，但是可以在编译的时候修改。指定这个选项等效于设置unix\_socket\_directories配置参数。

-l

启用使用SSL的安全连接。要使这个选项可用，编译LightDB时必须打开SSL支持。有关使用SSL的更多信息，请参考Section 16.9

-N max-connections

设置该服务器将接受的最大客户端连接数。该参数的默认值由initdb自动选择。指定这个选项等效于设置max\_connections配置参数。

-o extra-options

extra-options 中指定的命令行参数将传递给此 lightdb 进程启动的所有服务器进程。

extra-options 中的空格被视为分隔符，除非使用反斜杠 ( ) 进行转义；写入 \ 以表示字面意义的反斜杠。也可以通过多次使用 -o 来指定多个参数。



此选项的使用已过时；所有服务器进程的命令行选项都可以直接在 `lightdb` 命令行上指定。

`-p port`

指定lightdb用于监听客户端应用连接的 TCP/IP 端口或本地 Unix 域套接字文件扩展。默认为LTPORT环境变量的值。如果LTPORT没有设置，那么默认值是编译期间设立的值（通常是 5432）。如果你指定了一个非默认端口，那么所有客户端应用都必须用命令行选项或者LTPORT指定同一个端口。

`-s`

在每条命令结束时打印时间信息和其它统计信息。这个选项对测试基准和调节缓冲区数量有用处。

`-S work-mem`

指定在使用临时磁盘文件之前排序和散列表使用的基本内存量。请参阅 Section 17.4.1 中 `work_mem` 配置参数的说明。

`-V`

`--version`

打印lightdb版本并退出。

`--name=value`

设置一个命名的运行时参数；其缩写形式是 `-c`。

`--describe-config`

这个选项会用制表符分隔的COPY格式导出服务器的内部配置变量、描述以及默认值。设计它的目的是用于管理工具。

`-?`

`--help`

显示有关lightdb的命令行参数，并且退出。

## 半内部选项

这里描述的选项主要被用于调试目的，并且在某些情况下可以协助恢复严重受损的数据库。在生产数据库环境中应该不会去使用它们。在这里列举它们只是为了让LightDB系统开发者使用。此外，这些选项可能在将来的版本中更改或删除而不另行通知。

`-f { s | i | o | b | t | n | m | h }`

禁止某种扫描和连接方法的使用：`s`和`i`分别禁用顺序和索引扫描，`o`、`b`和`t`分别禁用只用索引扫描、位图索引扫描以及 TID 扫描，而`n`、`m`和`h`则分别禁用嵌套循环、归并和哈希连接。

顺序扫描和嵌套循环连接都不可能完全被禁用。`-fs`和`-fn`选项仅仅是在有其他选择时不鼓励优化器使用这些计划类型。

`-n`

该选项主要用于调试导致服务器进程异常崩溃的问题。对付这种情况的一般策略是通知所有其它服务器进程，让它们终止并且接着重新初始化共享内存和信号量。这是因为一个错误的服务器进程可能在终止之前就已经对共享状态造成了破坏。该选项指定lightdb将不会重新初始化共享数据结构。一个有经验的系统程序员这时就可以使用调试器检查共享内存和信号量状态。

- O  
允许修改系统表的结构。这个选项用于initdb。
- P  
读取系统表时忽略系统索引（但在更改系统表时仍然更新索引）。这在从损坏的系统索引中恢复时有用。
- t pa[rser] | pl[anner] | e[xecutor]  
打印与每个主要系统模块相关的查询的时间统计。这个选项不能和-s选项一起使用。
- T  
该选项主要用于调试导致服务器进程异常崩溃的问题。对付这种情况的一般策略是通知所有其它服务器进程，让它们终止并且接着重新初始化共享内存和信号量。这是因为一个错误的服务器进程可能在终止之前就已经对共享状态造成了破坏。该选项指定lightdb将通过发送SIGSTOP信号停止其他所有服务器进程，但是并不让它们终止。这样就允许系统程序员手动从所有服务器进程收集内核转储。
- v protocol  
声明这次会话使用的前/后服务器协议的版本数。该选项仅在内部使用。
- W seconds  
在一个新服务器进程被启动时，它实施认证过程之后会延迟这个选项所设置的秒数。这就留出了机会来用一个调试器附着在服务器进程上。

## 用于单用户模式的选项

下面的选项仅适用于单用户模式（见Single-User Mode）。

- single  
选择单用户模式。这必须是命令行中的第一个选项。
- database  
指定要访问的数据库的名称。这必须是命令行中的最后一个参数。如果省略它，则默认为用户名。
- E  
在执行命令之前回显所有命令到标准输出。
- j  
使用跟着两个新行的分号而不是仅用新行作为命令终止符。
- r filename  
将所有服务器日志输出发送到filename中。只有在作为一个命令行选项提供时，这个选项才会兑现。

## 环境

### LTCLIENTENCODING

客户端使用的默认字符编码（客户端可以独立地覆盖它）。这个值也可以在配置文件中设置。

## LTDATA

默认的数据目录位置。

## LTDATESTYLE

DateStyle运行时参数的默认值（这个环境变量的使用已被废弃）。

## LTPORT

默认端口号（在配置文件中设置更好）

## 诊断

一个提到了semget或shmget的错误消息可能意味着你需要配置内核来提供足够的共享内存和信号量。更多讨论请见Section 16.4 你也可以通过降低shared\_buffers值减少LightDB的共享内存消耗，或者降低max\_connections值减少信号量消耗，这样可以推迟对内核的重新配置。

如果一个消息说另外一个服务器已经在运行，应该仔细地检查，例如根据你的系统可以用命令

```
$ ps ax | grep lightdb
```

或

```
$ ps -ef | grep lightdb
```

如果你确信没有冲突的服务器正在运行，那么你可以删除消息中提到的锁文件然后再次尝试。

如果一个失败消息指示它无法绑定到一个端口，可能意味着该端口已经被某些非LightDB进程所使用。如果你终止lightdb并且立即使用相同的端口重启它，你也可能会得到这种错误。在这种情况下，你必须等待几秒直到操作系统关闭该端口，然后再重试。最后，如果你指定了一个操作系统认为需要保留的端口号，你可能也会得到这个错误。例如，很多版本的Unix 认为低于 1024 的端口号是“可信的”并且只允许 Unix 超级用户访问它们。

## 注解

实用命令lt\_ctl可以用来安全方便地启动和关闭lightdb服务器。

只要有可能，就不要使用SIGKILL杀死主lightdb服务器。这样会阻止lightdb在终止前释放它持有的系统资源（例如共享内存和信号量）。这样可能会导致启动新的lightdb进程时出现问题。

要正常地终止lightdb服务器，可以使用SIGTERM、SIGINT或者SIGQUIT信号。第一个在退出前将等待所有客户端终止，第二个将强行断开所有客户端的连接，第三个会不做正确的关闭立即退出并且会导致重启时的恢复。

SIGHUP信号会重新加载服务器配置文件。也可以向一个单独的服务器进程发送SIGHUP信号，但是这样做通常没什么意义。

要取消一个正在运行的查询，可以向运行该查询的进程发送SIGINT信号。要干净地终止一个后端进程，可向它发送SIGTERM。在 SQL 中可调用的与这两种动作等效的命令可参考Section 9.27. 中的pg\_cancel\_backend和pg\_terminate\_backend。

lightdb服务器使用SIGQUIT来告诉子服务器进程终止但不做正常的清理。该信号不应该被用户使用。向一个服务器进程发送SIGKILL也是不明智的 — 主lightdb进程将把这解释为一次崩溃，并且作为其标准崩溃恢复过程的一部分，它将强制所有的后代进程退出。

## 缺陷

--选项在FreeBSD或OpenBSD上无法运行，应该使用-c。这在受影响的系统里是个缺陷；如果这个缺陷没有被修复，将来的LightDB版本将提供一种解决方案。

## 单用户模式

要启动一个单用户模式的服务器，使用这样的命令

```
lightdb --single -D /usr/local/lightdb-x/data other-options my_database
```

用-D给服务器提供正确的数据库目录的路径，或者确保环境变量LTDATA被设置。同时还要指定你想在其中工作的特定数据库的名字。

通常，单用户模式的服务器会把换行符当做命令输入的终止符。它不明白分号的作用，因为那属于ltsql。要想把一个命令分成多行，必须在最后一个换行符以外的每个换行符前面敲一个反斜线。这个反斜线和旁边的新行都会被从输入命令中去掉。注意即使在字符串或者注释中也会这样做。

但是如果使用了-j命令行选项，那么单个新行将不会终止命令输入。相反，分号-新行-新行的序列才会终止命令输入。也就是说，输入一个紧跟着空行的分号。在这种模式下，反斜线-新行不会被特殊对待。此外，在字符串或者注释内的这类序列也不会被特殊对待。

不管在哪一种输入模式中，如果输入的一个分号不是正好在命令终止符之前或者不是命令终止符的一部分，它会被认为是一个命令分隔符。当真正输入一个命令终止符时，已经输入的多个语句将被作为一个单个事务执行。

要退出会话，输入EOF（通常是Control+D）。如果从上一个命令终止符以来已经输入了任何文本，那么EOF将被当作命令终止符，并且如果要退出则需要另一个EOF。

请注意单用户模式的服务器不会提供复杂的行编辑特性（例如没有命令历史）。但用户模式也不会做任何后台处理，例如自动检查点或者复制。

## 例子

要用默认值在后台启动lightdb:

```
$ nohup lightdb >logfile 2>&1 </dev/null &
```

要用指定端口启动lightdb，例如 1234:

```
$ lightdb -p 1234
```

要使用ltsql连接到这个服务器，用 -p 选项指定这个端口:

```
$ ltsql -p 1234
```

或者设置环境变量LTPORT:

```
$ export LTPORT=1234
$ ltsql
```

命名运行时参数可以用这些形式之一设置:

```
$ lightdb -c work_mem=1234  
$ lightdb --work-mem=1234
```

两种形式都覆盖lightdb.conf中可能存在的work\_mem设置。请注意在参数名中的下划线在命令行可以写成下划线或连字符。除了用于短期的实验外，更好的习惯是编辑lightdb.conf中的设置，而不是倚赖命令行开关来设置参数。

## 参见

initdb, lt\_ctl

---

# postmaster

postmaster — LightDB数据库服务器

## Synopsis

```
postmaster [option...]
```

## 描述

postmaster是postgres的一个废弃的别名。

## 参见

lightdb

---

# LightDB 工具应用程序

安装 LightDB 后，将提供一些第三方工具，这些工具可以在 `$LTHOME/tools` 目录中使用。

## Table of Contents

<code>check_list</code> .....	2084
<code>iftop</code> .....	2085
<code>iotop</code> .....	2086
<code>keepalived</code> .....	2087
<code>linux-ftools</code> .....	2088
<code>ltloader</code> .....	2089
<code>lvs</code> .....	2090
<code>ntp</code> .....	2091
<code>ora2pg</code> .....	2092
<code>PostgREST</code> .....	2093
<code>tshark</code> .....	2094
<code>vmtouch</code> .....	2095

---

# check\_list

check\_list — 检查LightDB运行环境状态的工具。

## Synopsis

```
python3 check_list/lightdb_maa_checklist.py [-d FILENAME] [-p PORT] [-t TAG] [-m LTHOME] [-n DBNAME]
```

## 描述

check\_list用于检查操作系统、配置、网络等不稳定性的可能原因，并检查LightDB的可用性、正确性、稳定性、易于故障排除性以及数据一致性。

## 选项

-d/--output\_dir

输出文件存放的目录名，默认为当前目录。

-p/--port

LightDB数据库服务器端口号（默认值为“5432”）。

-t/--tag

用于创建输出HTML文件名的字符串。（默认值为 None）

--m/--lthome

指定LightDB软件的文件系统位置。如果省略此选项，则使用环境变量\$ LTHOME。

--n/--dbname

要连接的LightDB数据库名称（默认值为“postgres”）。

-?

--help

显示有关check\_list命令行参数的帮助信息，并退出。

## 用法

使用所有默认参数。如果执行成功，则打印文件名。

```
$ python3 check_list/lightdb_maa_checklist.py
```

指定参数执行。如果执行成功，则打印文件名。

```
$ python3 check_list/lightdb_maa_checklist.py --port 5432 --tag "master_node" --dbname postgres --lthome /usr/local/lightdb
```



---

# iftop

iftop — 显示接口的带宽使用情况

## 说明

有关详细信息，请参见 iftop<sup>1</sup>。

---

<sup>1</sup> <http://www.ex-parrot.com/pdw/iftop/>

---

# iotop

iotop — 有多少 I/O 活动正在进行。

## 说明

有关详细信息，请参见 [iotop](http://guichaz.free.fr/iotop/)<sup>1</sup>。

---

<sup>1</sup> <http://guichaz.free.fr/iotop/>

---

# keepalived

keepalived — 提供简单而强大的负载均衡和高可用性功能。

## 说明

有关详细信息，请参见 [keepalived](https://www.keepalived.org/)<sup>1</sup>。

---

<sup>1</sup> <https://www.keepalived.org/>

---

# linux-ftools

linux-ftools — 用于 `fallocate`、`fincore`、`fadvise` 等的 Linux 命令行工具。

## 说明

有关详细信息，请参见 `linux-ftools`<sup>1</sup>。

---

<sup>1</sup> <https://code.google.com/archive/p/linux-ftools/>

---

# ltloader

ltloader — ltloader 将数据加载到 LightDB 中。

## 说明

有关详细信息，请参见 [pgloader](https://pgloader.io/)<sup>1</sup>。

---

<sup>1</sup> <https://pgloader.io/>

---

# lvs

lvs — 一个高度可扩展和高可用的服务器，构建在一组真实服务器上，使用 Linux 操作系统运行负载均衡器。

## 说明

有关详细信息，请参见 [lvs<sup>1</sup>](http://www.linuxvirtualserver.org/)。

---

<sup>1</sup> <http://www.linuxvirtualserver.org/>

---

## ntp

ntp — NTP（网络时间协议）广泛用于将计算机与互联网时间服务器或其他来源（如无线电或卫星接收器或电话调制解调器服务）同步。

## 说明

有关详细信息，请参见 `ntp`<sup>1</sup>。

---

<sup>1</sup> <http://www.ntp.org/index.html>

---

# ora2pg

ora2pg — 将 Oracle 和 MySQL 数据库迁移到 LightDB。

## 说明

有关详细信息，请参见 [ora2pg](https://ora2pg.darold.net/)<sup>1</sup>。

---

<sup>1</sup> <https://ora2pg.darold.net/>



---

# PostgREST

PostgREST — 从现有的 LightDB 数据库提供完全符合 RESTful API 的服务。

## 说明

有关详细信息，请参见 PostgREST<sup>1</sup>。

---

<sup>1</sup> <https://postgrest.org/en/stable/>

---

# tshark

tshark — 转储和分析网络流量。

## 说明

有关详细信息，请参见 `tshark`<sup>1</sup>。

---

<sup>1</sup> <https://www.wireshark.org/docs/man-pages/tshark.html>

---

# vmtouch

vmtouch — 一个用于了解和控制文件系统缓存的工具。

## 说明

有关详细信息，请参见 [vmtouch](https://hoytech.com/vmtouch/)<sup>1</sup>。

---

<sup>1</sup> <https://hoytech.com/vmtouch/>

---

# Part VII. 内部

这一部分包含LightDB开发者可能用到的各类信息。

---

## Table of Contents

47.	LightDB内部概述	2102
47.1.	一个查询的路径	2102
47.2.	连接如何建立	2102
47.3.	分析器阶段	2103
47.3.1.	分析器	2103
47.3.2.	转换处理	2103
47.4.	LightDB规则系统	2103
47.5.	规划器/优化器	2104
47.5.1.	生成可能的计划	2104
47.6.	执行器	2105
48.	系统目录	2106
48.1.	概述	2106
48.2.	lt_attri_ext	2108
48.3.	lt_database_ext	2108
48.4.	lt_global_temp_table_ext	2108
48.5.	lt_package	2109
48.6.	lt_partition_ext	2109
48.7.	lt_proc_ext	2110
48.8.	lt_type_ext	2110
48.9.	lt_variable	2111
48.10.	pg_aggregate	2111
48.11.	pg_am	2113
48.12.	pg_amop	2113
48.13.	pg_amproc	2114
48.14.	pg_attrdef	2115
48.15.	pg_attribute	2115
48.16.	pg_authid	2117
48.17.	pg_auth_members	2118
48.18.	pg_cast	2118
48.19.	pg_class	2119
48.20.	pg_collation	2121
48.21.	pg_constraint	2122
48.22.	pg_conversion	2124
48.23.	pg_database	2124
48.24.	pg_db_role_setting	2125
48.25.	pg_default_acl	2126
48.26.	pg_depend	2126
48.27.	pg_description	2128
48.28.	pg_enum	2128
48.29.	pg_event_trigger	2129
48.30.	pg_extension	2129
48.31.	pg_foreign_data_wrapper	2130
48.32.	pg_foreign_server	2130
48.33.	pg_foreign_table	2131
48.34.	pg_index	2131
48.35.	pg_inherits	2132
48.36.	pg_init_privs	2133
48.37.	pg_language	2133
48.38.	pg_largeobject	2134
48.39.	pg_largeobject_metadata	2134
48.40.	pg_namespace	2135
48.41.	pg_opclass	2135
48.42.	pg_operator	2136
48.43.	pg_opfamily	2137
48.44.	pg_partitioned_table	2137

---

48.45.	pg_policy	2138
48.46.	pg_proc	2138
48.47.	pg_publication	2140
48.48.	pg_publication_rel	2141
48.49.	pg_range	2141
48.50.	pg_replication_origin	2142
48.51.	pg_rewrite	2142
48.52.	pg_seclabel	2143
48.53.	pg_sequence	2143
48.54.	pg_shdepend	2144
48.55.	pg_shdescription	2145
48.56.	pg_shseclabel	2145
48.57.	pg_statistic	2146
48.58.	pg_statistic_ext	2147
48.59.	pg_statistic_ext_data	2147
48.60.	pg_subscription	2148
48.61.	pg_subscription_rel	2149
48.62.	pg_tablespace	2149
48.63.	pg_transform	2149
48.64.	pg_trigger	2150
48.65.	pg_ts_config	2151
48.66.	pg_ts_config_map	2152
48.67.	pg_ts_dict	2152
48.68.	pg_ts_parser	2153
48.69.	pg_ts_template	2153
48.70.	pg_type	2154
48.71.	pg_user_mapping	2157
48.72.	系统视图	2157
48.73.	pg_available_extensions	2158
48.74.	pg_available_extension_versions	2158
48.75.	pg_config	2159
48.76.	pg_cursors	2159
48.77.	pg_file_settings	2160
48.78.	pg_group	2161
48.79.	pg_hba_file_rules	2161
48.80.	pg_indexes	2162
48.81.	pg_locks	2162
48.82.	pg_matviews	2164
48.83.	pg_policies	2165
48.84.	pg_prepared_statements	2165
48.85.	pg_prepared_xacts	2166
48.86.	pg_publication_tables	2166
48.87.	pg_replication_origin_status	2167
48.88.	pg_replication_slots	2167
48.89.	pg_roles	2168
48.90.	pg_rules	2169
48.91.	pg_seclabels	2169
48.92.	pg_sequences	2170
48.93.	pg_settings	2171
48.94.	pg_shadow	2173
48.95.	pg_shmem_allocations	2173
48.96.	pg_stats	2174
48.97.	pg_stats_ext	2175
48.98.	pg_tables	2176
48.99.	pg_timezone_abbrevs	2176
48.100.	pg_timezone_names	2177
48.101.	pg_user	2177
48.102.	pg_user_mappings	2178

---

---

48.103.	pg_views	2178
49.	前端/后端协议	2180
49.1.	概述	2180
49.1.1.	消息概貌	2180
49.1.2.	扩展查询概述	2181
49.1.3.	格式和格式代码	2181
49.2.	消息流	2181
49.2.1.	启动	2181
49.2.2.	简单查询	2184
49.2.3.	扩展查询	2186
49.2.4.	流水线处理	2188
49.2.5.	函数调用	2189
49.2.6.	COPY操作	2189
49.2.7.	异步操作	2190
49.2.8.	取消正在处理的请求	2191
49.2.9.	终止	2191
49.2.10.	SSL会话加密	2192
49.2.11.	GSSAPI 会话加密	2192
49.3.	SASL认证	2193
49.3.1.	SCRAM-SHA-256认证	2193
49.4.	流复制协议	2194
49.5.	逻辑流复制协议	2201
49.5.1.	逻辑流复制参数	2201
49.5.2.	逻辑复制协议消息	2201
49.5.3.	逻辑复制协议的消息流	2201
49.6.	消息数据类型	2202
49.7.	消息格式	2202
49.8.	错误和通知消息域	2218
49.9.	逻辑复制消息格式	2220
49.10.	自协议2.0以来的变化总结	2224
50.	LightDB编码习惯	2226
50.1.	格式化	2226
50.2.	在服务器中报告错误	2226
50.3.	错误消息风格指导	2229
50.4.	其他编码习惯	2233
51.	本国语言支持	2235
51.1.	给翻译者	2235
51.1.1.	需求	2235
51.1.2.	概念	2235
51.1.3.	创建并维护消息目录	2236
51.1.4.	编辑 PO 文件	2236
51.2.	给编程者	2237
51.2.1.	技术	2237
51.2.2.	消息书写指南	2238
52.	编写一个过程语言处理器	2240
53.	编写一个外部数据包装器	2243
53.1.	外部数据包装器函数	2243
53.2.	外部数据包装器回调例程	2243
53.2.1.	用于扫描外部表的FDW例程	2243
53.2.2.	用于扫描外部连接的 FDW 例程	2245
53.2.3.	用于规划扫描/连接后处理的 FDW 例程	2246
53.2.4.	更新外部表的FDW例程	2246
53.2.5.	用于行锁定的 FDW 例程	2251
53.2.6.	EXPLAIN的FDW例程	2252
53.2.7.	ANALYZE的FDW例程	2253
53.2.8.	IMPORT FOREIGN SCHEMA的 FDW 例程	2253
53.2.9.	并行执行的 FDW 例程	2254
53.2.10.	用于路径重新参数化的FDW例程	2255

---

---

53.3.	外部数据包装器助手函数 .....	2255
53.4.	外部数据包装器查询规划 .....	2257
53.5.	外部数据包装器中的行锁定 .....	2258
54.	编写一种表采样方法 .....	2260
54.1.	采样方法支持函数 .....	2260
55.	编写一个自定义扫描提供者 .....	2263
55.1.	创建自定义扫描路径 .....	2263
55.1.1.	自定义扫描路径回调 .....	2264
55.2.	创建自定义扫描计划 .....	2264
55.2.1.	自定义扫描计划回调 .....	2265
55.3.	执行自定义扫描 .....	2265
55.3.1.	自定义扫描执行回调 .....	2265
56.	遗传查询优化器 .....	2268
56.1.	将查询处理看成是一个复杂的优化问题 .....	2268
56.2.	遗传算法 .....	2268
56.3.	LightDB 中的遗传查询优化 (GEQO) .....	2269
56.3.1.	用GEQO产生可能的计划 .....	2270
56.3.2.	LightDB GEQO的未来实现任务 .....	2270
56.4.	进一步阅读 .....	2270
57.	表访问方法接口定义 .....	2271
58.	索引访问方法接口定义 .....	2272
58.1.	索引的基本 API 结构 .....	2272
58.2.	索引访问方法函数 .....	2274
58.3.	索引扫描 .....	2279
58.4.	索引锁定考虑 .....	2280
58.5.	索引唯一性检查 .....	2281
58.6.	索引开销估计函数 .....	2282
59.	通用WAL 记录 .....	2285
60.	B-树索引 .....	2287
60.1.	简介 .....	2287
60.2.	B-树操作符类的行为 .....	2287
60.3.	B-树支持函数 .....	2288
60.4.	实现 .....	2290
60.4.1.	B-Tree 结构 .....	2290
60.4.2.	Deduplication .....	2290
61.	GiST 索引 .....	2293
61.1.	简介 .....	2293
61.2.	内建操作符类 .....	2293
61.3.	可扩展性 .....	2294
61.4.	实现 .....	2304
61.4.1.	GiST 缓冲构建 .....	2304
61.5.	示例 .....	2305
62.	GIN 索引 .....	2306
62.1.	简介 .....	2306
62.2.	内建操作符类 .....	2306
62.3.	可扩展性 .....	2306
62.4.	实现 .....	2309
62.4.1.	GIN 快速更新技术 .....	2309
62.4.2.	部分匹配算法 .....	2310
62.5.	GIN 提示和技巧 .....	2310
62.6.	限制 .....	2311
62.7.	例子 .....	2311
63.	哈希索引 .....	2312
63.1.	概述 .....	2312
63.2.	实现 .....	2312
64.	数据库物理存储 .....	2314
64.1.	数据库文件布局 .....	2314
64.2.	TOAST .....	2316

---



---

64.2.1.	线外磁盘上 TOAST 存储 .....	2316
64.2.2.	线外内存中 TOAST 存储 .....	2317
64.3.	空闲空间映射 .....	2318
64.4.	可见性映射 .....	2318
64.5.	初始化分支 .....	2318
64.6.	数据库页面布局 .....	2319
64.6.1.	表行布局 .....	2320
65.	系统目录声明和初始内容 .....	2322
65.1.	系统目录声明规则 .....	2322
65.2.	系统目录初始数据 .....	2323
65.2.1.	数据文件格式 .....	2323
65.2.2.	OID分配 .....	2324
65.2.3.	OID引用查找 .....	2325
65.2.4.	数组类型的自动创建 .....	2325
65.2.5.	编辑数据文件的方法 .....	2326
65.3.	BKI文件格式 .....	2327
65.4.	BKI命令 .....	2327
65.5.	自举BKI文件的结构 .....	2328
65.6.	BKI例子 .....	2329
66.	规划器如何使用统计信息 .....	2330
66.1.	行估计例子 .....	2330
66.2.	多变量统计例子 .....	2335
66.2.1.	函数依赖 .....	2335
66.2.2.	N 个不同变量的计数 .....	2336
66.2.3.	MCV 列表 .....	2337
66.3.	规划器统计和安全 .....	2338
67.	备份清单格式 .....	2339
67.1.	备份清单顶级(Top-level)对象 .....	2339
67.2.	Backup Manifest File Object .....	2339
67.3.	Backup Manifest WAL Range Object .....	2340

---

# Chapter 47. LightDB内部概述

## 作者

这一章的内容来自于[sim98]的一部分，它是Stefan Simkovics在维也纳技术大学的硕士学位论文，指导人是 Georg Gottlob教授和Mag. Katrin Seyr。

本章给出了LightDB后台内部结构的一个总览。在阅读了下面的小节后，你将对一个查询是如何被执行的有一定了解。本章希望帮助读者理解当后台收到一个查询后，在结果被返回给客户之前通常会发生怎样的操作。

## 47.1. 一个查询的路径

这里我们将对查询到获取结果之间的阶段给与简要概述。

1. 一个由应用程序到LightDB服务器的连接必须被建立。应用程序传递一个查询给服务器并等待接收由服务器传回的结果。
2. 分析阶段对由应用程序传递的查询进行语法检查，并创建一个查询树。
3. 重写系统得到分析阶段创建的查询树，并查找可以应用到该查询树的任何规则（存储在系统目录中）。对找到的规则，它会执行规则体中给定的转换。

重写系统的一个应用是实现视图。不论何时发生一个视图（即一个虚拟表）上的查询，重写系统将用户查询重写为一个访问视图定义中给定的基本表的查询来替代。

4. 规划器/优化器接手（重写过的）查询树并创建一个将被作为执行器输入的查询计划。

它会先创建所有可能导向相同结果的路径。例如，如果在一个被扫描的关系上有一个索引，则有两条可供扫描的路径。其中之一是一个简单的顺序扫描，而另一个则是使用索引。接下来执行每条路径所需的代价被估算出来并且代价最低的路径将被选中。代价最低的路径将被扩展成一个完整的计划可供执行器使用。

5. 执行器递归地逐步通过计划树并按照计划表述的方式获取行。执行器在扫描关系时会使用存储系统、执行排序和连接、估算条件并最后归还得到的行。

在下面的小节中我们将覆盖上述列出的每一项的详细内容，以便更好地理解LightDB的内部控制和数据结构。

## 47.2. 连接如何建立

LightDB实现了“process per user”客户端/服务器模型。在这个模型中，每个客户端进程只连接到一个后端进程。由于我们无法预先知道会有多少连接被建立，我们必须使用一个“守护者进程(supervisor process)”在每次连接请求时生产一个新的服务器进程。该守护者进程被称为postmaster，它在一个特定的TCP/IP端口监听进入的连接。当一个连接请求被监测到时，它会产生一个新的后端进程。后端进程相互之间，以及与instance中的其他进程之间，通过semaphores 和 shared memory进行通信，以保证并发数据访问时的数据完整性。

客户端进程可以是任何符合LightDB协议（见Chapter 49 的程序。很多客户端基于C语言库libpq，但也有一些该协议的独立实现存在，例如Java的JDBC驱动。

一旦一个连接被建立，客户端进程可以给它连接的后端进程发送一个查询。查询被以纯文本传送，在客户端不做分析。后端进程会分析查询，创建一个执行计划，然后执行这个计划，并通过已建立的连接向客户端返回检索到的行。

## 47.3. 分析器阶段

分析器阶段由两部分组成：

- 分析器定义在`gram.y`和`scan.l`中，它使用Unix工具**bison**和**flex**构建。
- 转换处理将对分析器返回的数据结构进行修改和增加。

### 47.3.1. 分析器

分析器必须检查查询字符串（以纯文本形式到达）是否为合法语法。如果语法正确将建立一个分析树并返回之，否则将返回一个错误。语法分析器和词法分析器使用著名的Unix工具**bison**和**flex**实现。

词法分析器定义在文件`scan.l`中，并负责识别标识符、SQL关键词等。对于找到的每一个关键词或标识符将生成一个记号并返回给语法分析器。

语法分析器定义在`gram.y`文件中，它由一组语法规则和动作，动作将在规则被触发时被执行。动作的代码（实际上是C代码）将被用于构建分析树。

程序**flex**把文件`scan.l`转换成C源文件`scan.c`，程序**bison**把文件`gram.y`转换为`gram.c`。在这些转换结束后，一个正规的C编译器就可以用于创建分析器。绝不要对生成的C文件做任何修改，因为每次**flex**或**bison**被调用都会重写它们。

#### Note

前面提到的转换和编译通常是由随LightDB源代码发布的makefiles自动完成。

对于**bison**的详细介绍或者`gram.y`中的语法规则超出了本指南的范围。有很多书籍和文档介绍**flex**和**bison**。在学习`gram.y`中的语法之前你应该先熟悉**bison**，否则你将无法理解发生了什么。

### 47.3.2. 转换处理

分析阶段根据SQL的语法结构的固定规则创建一个分析树。它不会在系统目录做任何查找，这样它不可能了解所请求的操作的详细语义。在分析器完成之后，转换处理接手分析器返回的树，并进行语义解释来理解该查询引用了哪些表、函数和操作符。用于表示该信息的数据结构被称为查询树。

将原始分析从语义分析中分离出来的原因是系统目录的查找只能在一个事务中完成，而不希望在收到一个查询字符串时立即开始一个事务。原始分析阶段足以识别事务控制命令（`BEGIN`、`ROLLBACK`等），并且这些可以在没有任何进一步分析之前正确地被执行。一旦我们知道我们正在处理一个确切的查询（例如`SELECT`或`UPDATE`），就可以开始一个事务（如果我们还不其中）。只有到这时转换处理才能被调用。

由转换处理创建的查询树在结构上和原始分析树有很多地方相似，但是在细节上有很多不同之处。例如，分析树中的一个`FuncCall`节点表示某些在语法上看起来像一个函数调用的东西。它可能被转换成一个`FuncExpr`或`Aggref`节点，取决于被引用的名字是一个普通函数或是一个聚集函数。此外，关于列和表达式结果的实际数据类型的信息被加入到了查询树中。

## 47.4. LightDB规则系统

LightDB支持一个强大的规则系统用于支持视图说明和模糊不清的视图更新。本来LightDB规则系统由两种实现组成：

- 第一种利用行级处理工作，并且被实现于执行器深层。当单独一行被访问时规则系统被调用。这种实现在1995年被移除，那时最后一个Berkeley Postgres项目的官方发布被转换为Postgres95。
- 第二种规则系统的实现是一种称为查询重写的技术。重写系统是一个存在于分析器阶段和规划器/优化器之间的模块。这种技术也被实现了。

更多关于查询重写器的讨论可见Chapter 39 因此没有必要在这里涉及这些内容。我们将仅指出重写器的输入和输出都是查询树，即在树的表现形式或语义细节层次上都没有改变。重写可以被看成是某种形式的宏扩展。

## 47.5. 规划器/优化器

规划器/优化器的任务是创建一个最佳的执行计划。一个给定的SQL查询（今后将是一个查询树）实际上可以以很多种不同的方式被执行，其中的每一种都会产生相同的结果集。如果在计算上可行，查询优化器将检查这些可能的执行计划中的每一个，最后选择其中被期望“跑得最快的”那一个。

### Note

在某些情况下，检查一个查询的每一种可能的执行方式会耗费非常多的时间和内存。特别是当查询涉及到大量连接操作时。为了能在合理的时间内决定一个合理的（不一定是最佳的）查询计划，当连接数量超过一个阈值（见`geqo_threshold`）时LightDB使用了一种遗传查询优化器（见Chapter 56）。

规划器搜索过程实际上依靠称为路径的数据结构工作，它是一种缩短版的计划，其中只包含规划器做决定所需要的信息。当最低代价的路径被确定后，一个全功能的计划树将被建立并传递给执行器。这表示所期望的执行计划已经拥有足够的细节以供执行器执行它。在本节剩下的部分，我们将忽略路径和计划之间的区别。

### 47.5.1. 生成可能的计划

规划器/优化器从扫描查询中用到的每一个单独的关系（表）开始生成计划。可能的计划根据每一个关系上可用的索引决定。在一个关系上总是有执行一个顺序扫描的可能，因此一个顺序扫描计划总是会被创建。假设在一个关系上定义有一个索引（例如一个B-tree索引）并且查询包含限制`relation.attribute OPR constant`。如果`relation.attribute`正好匹配该B-tree索引的键并且OPR是该索引的操作符类之一，另一个使用B-tree索引扫描该索引的计划将被创建。如果还有索引存在且查询中的限制正好匹配一个索引的键，其他计划也会考虑。如果有索引的顺序能匹配ORDER BY子句（如果有）或者对于归并连接有用（见下文），也会为该索引创建索引扫描计划。

如果查询需要连接两个或更多关系，在所有扫描单个关系的可能计划都被找到后，连接计划将会被考虑。三种可用的连接策略是：

- 嵌套循环连接：对左关系找到的每一行都要扫描右关系一次。这种策略最容易实现但是可能非常耗时（但是，如果右关系可以通过索引扫描，这将是一个不错的策略。因为可以用左关系当前行的值来作为右关系上索引扫描的键）。
- 归并连接：在连接开始之前，每一个关系都按照连接属性排好序。然后两个关系会被并行扫描，匹配的行被整合成连接行。由于这种连接中每个关系只被扫描一次，因此它很具有吸引力。它所要求的排序可以通过一个显式的排序步骤得到，或使用一个连接键上的索引按适当顺序扫描关系得到。
- 哈希连接：右关系先被扫描并且被载入到一个哈希表，使用连接属性作为哈希键。接下来左关系被扫描，扫描中找到的每一行的连接属性值被用作哈希键在哈希表中查找匹配的行。

当查询涉及两个以上的关系时，最终结果必须由一个连接步骤树构成，每个连接步骤有两个输入。规划器会检查不同可能的连接序列来找到代价最小的那一个。

如果查询是用的关系数少于`geqo_threshold`，将使用一次接近穷举的搜索来查找最好的连接顺序。如果任何两个关系在WHERE条件中存在一个相应的连接子句（即存在类似于where `rel1.attr1=rel2.attr2`的限制），规划器会有限考虑它们之间的连接。没有任何连接子句的连接对只有在别无选择时才会被考虑，即一个关系没有任何可用的对于其他关系的连接子句。对规划器所考虑的每一个连接对会生成所有可能的计划，其中代价（被估计为）最低的一个将被选择。

当连接关系数超过`geqo_threshold`时，连接序列将考虑通过启发式方法来确定，详见Chapter 56 否则处理将和前面相同。

成品计划树包含基本关系的顺序或索引扫描，外加所需的嵌套循环、归并或哈希连接节点，以及任何所需的辅助步骤，例如排序节点或聚集函数计算节点。这些节点中的大部分具有执行选择（丢弃不符合指定布尔条件的行）和投影（根据指定列值计算派生列，即标量表达式的计算）的能力。规划器的职责之一就是要在计划树最合适的节点上附加来自于子句的选择条件和需要的输出表达式。

## 47.6. 执行器

执行器接手规划器/优化器创建的计划，并递归地处理之以抽取所需的行集。这本质上是一种需求拉动的管道机制。每次一个计划节点被调用时，它必须交付一个或多个行，或者报告已经完成了行的交付。

为了提供一个具体例子，假设顶层节点是一个MergeJoin节点。在归并完成之前，两个行必须先被获取（每一个来自于一个子计划）。因此执行器递归地调用它自己去处理子计划（从附加在`lefttree`的子计划开始）。新的顶层节点（左子计划的顶层节点），我们说是一个Sort节点，并且又要递归来获取一个输入行。Sort的子节点可以是一个SeqScan节点，表示真正地读取一个表。该节点的执行将会使执行器从表中获取一行并将它返回给调用节点。Sort节点将反复调用它的子节点来获得所有需要排序的行。当输入耗尽后（子节点将返回一个NULL来标识），Sort节点执行排序，并且最后能够返回它的第一个输出行，及排序后的第一个。它会把剩下的行保存下来，这样它可以根据后续的要求按照排好的顺序返回这些行。

MergeJoin节点也会相似地从其右子计划要求第一个行。然后它会比较两个子节点提供的行看它们是否能被连接，如果可以它会返回一个连接行给调用者。在下次调用时，或者它无法连接当前的输入行，它会前进到一个表或另一个表的下一行（取决于比较的结果），并再次检查匹配。最后，某个子计划耗尽，MergeJoin节点返回NULL表示它没有更多连接行可以提供。

复杂的查询可能涉及多层计划节点，但是一般的方法是相同的：每个节点在被调用时计算并返回它的下一个输出行。每个节点同时也负责应用由规划器分配给它的选择或投影表达式。

执行器机制被用于评估四种基本SQL查询类型：SELECT, INSERT, UPDATE, 和 DELETE。对于SELECT，顶层执行器代码只需要发送查询计划树返回的每个行给客户端。INSERT ... SELECT, UPDATE, 和 DELETE 是在一个被称为ModifyTable的特殊顶层计划节点下面的有效的SELECT。

INSERT ... SELECT 把行输送到ModifyTable来执行插入。对于UPDATE，规划器会安排每一个计算行包含所有被更新的列值，加上原始目标行的TID（元组ID或行ID）；这些数据也会被输送到一个ModifyTable节点，该节点将利用这些信息创建一个新的被更新行并标记旧行为被删除。对于DELETE，被计划实际返回的唯一的列是TID，ModifyTable节点简单地使用TID访问每一个目标行并将其标记为被删除。

一个简单的INSERT ... VALUES命令会创建一个由一个Result节点组成的简单计划树，只计算一个结果行，然后输送到ModifyTable来执行插入。

# Chapter 48. 系统目录

系统目录是关系型数据库存放模式元数据的地方，比如表和列的信息，以及内部统计信息等。LightDB的系统目录就是普通表。你可以删除并重建这些表、增加列、插入和更新数值，然后彻底把你的系统搞垮。通常情况下，我们不应该手工修改系统目录，通常有SQL命令可以做这些事情。（例如，CREATE DATABASE向 pg\_database表插入一行 — 并且实际上在磁盘上创建该数据库。）。有几种特别深奥的操作例外，但是随着时间的流逝其中的很多也可以用 SQL 命令来完成，因此对系统目录直接修改的需求也越来越小。

## 48.1. 概述

Table 48. 列出了系统目录。每个目录更详细的文档见后文。

大多数系统目录都是在数据库创建的过程中从模版数据库中拷贝过来的，因此都是数据库相关的。少数目录在物理上是在一个集簇的所有数据库间中共享的，这些将在每一个目录单独的描述中介绍。

Table 48.1. 系统目录

目录名	用途
lt_attr_ext	表列扩展属性
lt_database_ext	数据库额外属性
lt_global_temp_table_ext	全局临时表
lt_package	plorasql包定义
lt_partition_ext	分区表额外属性
lt_proc_ext	包函数
lt_type_ext	类型优先级
lt_variable	包变量
pg_aggregate	聚集函数
pg_am	关系访问方法
pg_amop	访问方法操作符
pg_amproc	访问方法支持函数
pg_attrdef	列默认值
pg_attribute	表列（“属性”）
pg_authid	认证标识符（角色）
pg_auth_members	认证标识符成员关系
pg_cast	转换（数据类型转换）
pg_class	表、索引、序列、视图（“关系”）
pg_collation	排序规则（locale信息）
pg_constraint	检查约束、唯一约束、主键约束、外键约束
pg_conversion	编码转换信息
pg_database	本数据库集簇中的数据库
pg_db_role_setting	每角色和每数据库的设置
pg_default_acl	对象类型的默认权限
pg_depend	数据库对象间的依赖
pg_description	数据库对象上的描述或注释
pg_enum	枚举标签和值定义

目录名	用途
pg_event_trigger	事件触发器
pg_extension	已安装扩展
pg_foreign_data_wrapper	外部数据包装器定义
pg_foreign_server	外部服务器定义
pg_foreign_table	外部表信息
pg_index	索引信息
pg_inherits	表继承层次
pg_init_privs	对象初始特权
pg_language	编写函数的语言
pg_largeobject	大对象的数据页
pg_largeobject_metadata	大对象的元数据
pg_namespace	模式
pg_opclass	访问方法操作符类
pg_operator	操作符
pg_opfamily	访问方法操作符族
pg_partitioned_table	表的分区键的信息
pg_policy	行安全策略
pg_proc	函数和过程
pg_publication	用于逻辑复制的发布
pg_publication_rel	发布映射的关系
pg_range	范围类型的信息
pg_replication_origin	已注册的复制源
pg_rewrite	查询重写规则
pg_seclabel	数据库对象上的安全标签
pg_sequence	有关序列的信息
pg_shdepend	共享对象上的依赖
pg_shdescription	共享对象上的注释
pg_shseclabel	共享数据库对象上的安全标签
pg_statistic	规划器统计
pg_statistic_ext	扩展的规划器统计信息（定义）
pg_statistic_ext_data	扩展的规划器统计信息（已构建的统计信息）
pg_subscription	逻辑复制订阅
pg_subscription_rel	订阅的关系状态
pg_tablespace	本数据库集簇内的表空间
pg_transform	转换（将数据类型转换为过程语言需要的形式）
pg_trigger	触发器
pg_ts_config	文本搜索配置
pg_ts_config_map	文本搜索配置的记号映射
pg_ts_dict	文本搜索字典

目录名	用途
pg_ts_parser	文本搜索分析器
pg_ts_template	文本搜索模板
pg_type	数据类型
pg_user_mapping	将用户映射到外部服务器

## 48.2. lt\_attri\_ext

目录 `lt_attri_ext` 存储有关表列的时间戳信息。

Table 48.2. `lt_attri_ext` Columns

列类型	描述
relid oid	此列所属的表
colname name	列名称
updatetimestamp int16	自动更新时间戳标志。

## 48.3. lt\_database\_ext

目录 `lt_data_ext` 存储有关数据的额外信息。像 `PG_DATA` 目录一样，`LT_DATA_EXT` 在群集的所有数据中共享：每个群集只有一个 `lt_data_ext` 的副本，每个数据只有一个。

Table 48.3. `lt_database_ext` Columns

列类型	描述
datoid oid (references pg_database.oid)	数据库OID
compa_type int4	兼容模式类型
ascii_zero_store_value int4	Lightdb 23.3 引入了一个新的数据库级别的GUC参数 <code>lightdb_ascii_zero_store_value</code> ，我们将此参数存储于此，此参数取值范围为0到32，包含0，32。此值只在Oracle或Mysql模式下有意义。更多关于GUC <code>lightdb_ascii_zero_store_value</code> 的信息，请参考 <code>lightdb_ascii_zero_store_value</code>

## 48.4. lt\_global\_temp\_table\_ext

目录 `lt_global_temp_table_ext` 存储有关全局临时表的信息。

Table 48.4. `lt_global_temp_table_ext` Columns

列类型	描述
relid oid	行标识符
nspname name	



列类型	描述
	包含此关系命名空间的OID
relname name	表名称
preserved boolean	如果是会话级全局临时表，则为true，如果交易级全局临时表，则为false
code text	其他信息，暂时作为保留字段。

## 48.5. lt\_package

目录 `lt_package` 存储有关plorasql过程语言的包装信息。

Table 48.5.lt\_package Columns

列类型	描述
oid oid	包OID
pkgname name	包名
pkgnamespace oid	包命名空间OID
pkgowner oid	包所有者OID
pkgsecdef bool	包安全条款
pkgspec text	包规范文本字符串
pkgbody text	包正文的文本字符串
pkgaccessor anyarray	包登录列表
pkgacl aclitem[]	包访问权限

## 48.6. lt\_partition\_ext

目录 `lt_partition_ext` 存储有关分区表的信息额外属性。

Table 48.6.lt\_partition\_ext Columns

列类型	描述
oid oid	行标识符
auto_partition_type int32	自动分区类型。目前仅支持类型3。类型3是时间范围的自动分区。随后的版本将支持其他类型的自动分区。
interval_type int32	

列类型	描述
	时间类型在时间范围内自动分区。 总共6种类型， 分别对应于年/月/天/小时/分钟/秒。
interval_value int32	间隔
interval_start_time text	时间范围的自动分区开始时间

## 48.7. lt\_proc\_ext

目录lt\_proc\_ext 存储包函数信息。

Table 48.7.lt\_proc\_ext Columns

列类型	描述
oid oid	行标识符
procoid oid (references pg_proc.oid)	函数OID
proname name	函数名称
pronamespace oid	函数所属包OID
proaccess char	函数访问属性
procspec text	包构造函数匿名块

## 48.8. lt\_type\_ext

目录 lt\_type\_ext 存储有关类型优先级的信息。 有关详细信息， 请参见Section 10. 部分。

Table 48.8.lt\_type\_ext Columns

列类型	描述
oid oid	行标识符
typeid oid (references pg_type.oid)	类型OID
priority int8	类型的优先级
typnamespace oid	类型的命名空间
typaccess char	类型访问属性
typtype integer	类型属性值

列类型	描述
typename name	类型名称

## 48.9. `lt_variable`

目录`lt_variable`存储包变量信息。

Table 48.9. `lt_variable` Columns

列类型	描述
oid oid	行标识符
varname name	变量名称
varnamespace oid	变量所属包OID
vartype oid	变量类型
vartypmod integer	域使用vartypmod来记录被应用于它们基类型的typmod（如果基类型不使用typmod，则为-1）。如果此类型不是一个域则为-1。
varowner oid	变量的拥有者
varcollation oid (references pg_collation.oid)	varcollation指定此类型的排序规则。如果类型不支持排序规则，此列为0。一个支持排序规则的基类型此处会有一个非零值，典型值为DEFAULT_COLLATION_OID。可排序类型上的域可以有一个不同于其基类型的排序规则OID，如果为该域指定了一个排序规则OID的话。
varisnotnull boolean	varisnotnull表示类型上的一个非空约束。只用于域。
varisimmutable boolean	是否为表类型变量
vareoxaction char	变量的EOX动作。
vardefexpr pg_node_tree	如果vardefexpr为非空，那么它是该类型默认表达式的nodeToString()表现形式。这个列只用于域。
varacl aclitem[]	访问权限，另请参阅Section 5.7for details.
varaccess char	变量访问属性
cursorquery text	光标定义语句

## 48.10. `pg_aggregate`

目录`pg_aggregate`存储关于聚集函数的信息。聚集函数是对一个数值集合（典型的是每个匹配查询条件的行中的同一个列的值）进行操作的函数，它返回从这些值中计算出的一个

数值。典型的聚集函数是 sum、count和max。 pg\_aggregate里的每个项都是一个pg\_proc项的扩展。 pg\_proc项记载该聚集的名字、输入和输出数据类型，以及其他一些和普通函数类似的信息。

Table 48.10. pg\_aggregate Columns

列类型	描述
aggfnoid regproc (references pg_proc.oid)	聚集函数的pg_proc OID
aggkind char	Aggregate kind: n 表示 “普通” 聚集, o 表示 “有序集” 聚集, 或 h 表示 “假想集” 聚集
aggnumdirectargs int2	一个有序集或者假想集聚集的直接（非聚集）参数的数量，一个可变数组算作一个参数。如果等于pronargs，该聚集必定是可变的并且该可变数组描述聚集参数以及最终直接参数。对于普通聚集总是为零。
aggtransfn regproc (references pg_proc.oid)	转移函数
aggfinalfn regproc (references pg_proc.oid)	最终函数（如果没有就为零）
aggcombinefn regproc (references pg_proc.oid)	结合函数（如果没有就为零）
aggserialfn regproc (references pg_proc.oid)	序列化函数（如果没有就为零）
aggdeserialfn regproc (references pg_proc.oid)	反序列化函数（如果没有就为零）
aggmtransfn regproc (references pg_proc.oid)	用于移动聚集模式的向前转移函数（如果没有就为零）
aggminvtransfn regproc (references pg_proc.oid)	用于移动聚集模式的反向转移函数（如果没有就为零）
aggmfinalfn regproc (references pg_proc.oid)	用于移动聚集模式的最终函数（如果没有就为零）
aggfinalextra bool	True 将传递额外的虚拟参数到 aggfinalfn
aggmfinalextra bool	True 将传递额外的虚拟参数到 aggmfinalfn
aggfinalmodify char	aggfinalfn 是否修改传递状态值：如果是只读则为r，如果不能在aggfinalfn之后应用aggtransfn则为s，或者如果它修改该值则为w
aggmfinalmodify char	和aggfinalmodify类似，但是用于aggmfinalfn
aggsortop oid (references pg_operator.oid)	相关联的排序操作符（如果没有则为0）
aggtranstype oid (references pg_type.oid)	聚集函数的内部转移（状态）数据的数据类型
aggtransspace int4	转移状态数据的近似平均尺寸（字节），或者为零表示使用一个默认估算值
aggmtranstype oid (references pg_type.oid)	聚集函数用于移动聚集欧氏的内部转移（状态）数据的数据类型（如果没有则为零）

列类型	描述
aggmtransspace int4	移动聚合模式的转移状态数据的近似平均尺寸（字节），或者为零表示使用一个默认估算值
agginitval text	转移状态的初始值。这是一个文本域，它包含初始值的外部字符串表现形式。如果这个域为空，则转移状态值从空值开始。
aggminitval text	用于移动聚集模式的转移状态初值。这是一个文本域，它包含了以其文本字符串形式表达的初值。如果这个域为空，则转移状态值从空值开始。

新的聚集函数可通过CREATE AGGREGATE命令注册。更多关于编写聚集函数以及转移函数的含义等信息请参见Section 36.12

## 48.11. pg\_am

目录pg\_am存储关于关系访问方法的信息。系统支持的每种访问方法在这个目录中都有一行。目前只有表和索引拥有访问方法。表和索引访问方法的需求分别在Chapter 5和Chapter 58中详细讨论。

Table 48.11. pg\_am Columns

类类型	描述
oid oid	行标识符
amname name	访问方法的名字
amhandler regproc (references pg_proc.oid)	负责提供有关该访问方法信息的处理器函数的 OID
amtype char	t = 表(包括物化视图), i = 索引.

### Note

在LightDB 9.6 之前，pg\_am包含很多额外的列以表示索引访问方法的性质。那些数据现在只有在 C 代码级别才是直接可见的。不过，系统中增加了pg\_index\_column\_has\_property()和一些相关函数来允许 SQL 查询检查索引访问方法的性质，请见Table 9.69

## 48.12. pg\_amop

目录pg\_amop存储关于与访问方法操作符族相关的操作符信息。对于一个操作符族中的每一个成员即操作符都在这个目录中有一行。一个成员可以是一个搜索操作符或者一个排序操作符。一个操作符可以出现在多个族中，但在同一个组中既不能出现在多个搜索位置也不能出现在多个排序位置（虽然不太可能出现，但是允许一个操作符同时用于搜索和排序目的）。

Table 48.12. pg\_amop Columns

列类型	描述
oid oid	

列类型	描述
	行标识符
amopfamily oid (references pg_opfamily.oid)	这个项所在的操作符系列
amoplefttype oid (references pg_type.oid)	操作符的左手输入数据类型
amoprightrighttype oid (references pg_type.oid)	操作符的右手输入数据类型
amopstrategy int2	操作符策略号
amoppurpose char	操作符目的, s表示搜索, o表示排序
amopopr oid (references pg_operator.oid)	操作符的OID
amopmethod oid (references pg_am.oid)	使用此操作符系列的索引访问方法
amopsortfamily oid (references pg_opfamily.oid)	如果是一个排序操作符, 该项会根据这个 B树操作符族排序, 如果是一个搜索操作符则为0

一个“搜索”操作符项意味着该操作符族的一个索引可以被搜索来查找所有满足如下条件的行: WHERE indexed\_column operator constant。显然, 这样的操作符必须返回boolean, 且它的左手输入类型必须匹配索引列的数据类型。

一个“排序”操作符项意味着该操作符族的一个索引可以被扫描来返回以如下顺序排列的行: ORDER BY indexed\_column operator constant。这样一个操作符能够返回任何可排序数据类型, 尽管它的左手输入类型必须匹配索引列的数据类型。ORDER BY的准确语义由amopsortfamily列指定, 它必须引用一个适合于操作符结果类型的B树操作符族。

**Note**

目前, 一个排序操作符的排序顺序被假设为其引用的操作符族的默认值, 即ASC NULLS LAST。未来可能会通过增加额外的列来显式地指定排序选项。

一个项的amopmethod必须和它所包含的操作符族的opmethod相匹配 (这里包括amopmethod是一个为了性能原因而故意对目录结构做的反规范化)。此外, amoplefttype和amoprightrighttype也必须匹配被引用的pg\_operator项的oprleft和oprright域。 此

## 48.13. pg\_amproc

目录pg\_amproc存储关于访问方法操作符族相关的支持函数。属于一个操作符族的每一个支持函数在这个目录中都有一行。

Table 48.13. pg\_amproc Columns

列类型	描述
oid oid	行标识符
amprocfamily oid (references pg_opfamily.oid)	使用这个项的操作符系列

列类型	描述
amproclefttype oid (references pg_type.oid)	相关操作符的左手输入数据类型
amprocrighttype oid (references pg_type.oid)	相关操作符的右手输入数据类型
amprocnum int2	支持的函数编号
amproc regproc (references pg_proc.oid)	函数的OID

amproclefttype和amprocrighttype列的通常解释是它们标识了一个特定支持过程所支持的操作符的左右输入类型。对于某些访问方法它们和支持过程本身的输入数据类型相匹配，而对其他的则不会匹配。对于一个索引有一个“默认”支持过程的概念，这些支持过程的amproclefttype和amprocrighttype都等于索引操作符类的opcintype。

## 48.14. pg\_attrdef

pg\_attrdef存储列的默认值。列的主要信息存储在pg\_attribute。只有那些显式指定了一个默认值的列才会在这个目录中有一个项。

Table 48.14. pg\_attrdef Columns

列类型	描述
oid oid	行标识符
adrelid oid (references pg_class.oid)	该列所属的表
adnum int2 (references pg_attribute.attnum)	列的编号
adbin pg_node_tree	列的默认值，以nodeToString()表示。用pg_get_expr(adbin, adrelid)将其转换为SQL表达式。

## 48.15. pg\_attribute

目录pg\_attribute存储有关表列的信息。数据库中的每一个表的每一个列都恰好有pg\_attribute中有一行。（这其中也会有索引的属性项，并且事实上所有对象都具有pg\_class项） entries。）

术语属性等同于列，这里使用它只是出于历史原因。

Table 48.15. pg\_attribute Columns

列类型	描述
attrelid oid (references pg_class.oid)	此列所属的表
attname name	列名称
atttypid oid (references pg_type.oid)	此列的数据类型（删除的列为零）

列类型	描述
attstattarget int4	attstattarget控制由ANALYZE. 对此列收集的统计信息的细节层次。 0值表示不会收集任何统计信息。一个负值则说明直接使用系统默认的目标。正值的确切含义取决于数据类型。对于标量数据类型，attstattarget既是要收集的“最常见值”的目标号，也是要创建的柱状图容器的目标号。
attlen int2	本列类型的pg_type.typlen一个拷贝 type
attnum int2	列的编号。一般列从1开始向上编号。系统列，如ctid，则拥有（任意）负值编号。
attnums int4	如果该列是一个数组类型，这里就是其维度数；否则为0。（在目前一个数组的维度数并不被强制，因此任何非零值都能有效地表明“这是一个数组”。）
attcacheoff int4	在存储中总是为-1，但是当被载入到一个内存中的行描述符后，这里可能会被更新为属性在行内的偏移
atttypmod int4	atttypmod记录了在表创建时提供的类型相关数据（例如一个varchar列的最大长度）。它会被传递给类型相关的输入函数和长度强制函数。对于那些不需要atttypmod的类型，这个值通常总是为-1。
attbyval bool	该列类型的pg_type.typbyval的一个拷贝
attstorage char	通常是该列类型的pg_type.typstorage的一个拷贝。对于可TOAST的数据类型，这可以在列创建后被修改以控制存储策略。
attalign char	该列类型的 pg_type.typalign 的一个拷贝
attnotnull bool	这表示一个非空约束。
atthasdef bool	该列有一个默认表达式或生成的表达式，在此情况下在pg_attrdef目录中会有一个对应项来真正定义该表达式。（检查attgenerated以确定是默认还是生成的表达式。）
atthasmissing bool	该列在行中完全缺失时会用到这个列的值，如果在行创建之后增加一个有非易失DEFAULT值的列，就会发生这种情况。实际使用的值被存放在attmissingval列中。
attidentity char	如果是一个零字节（''），则不是一个标识列。否则，a = 总是生成，d = 默认生成。
attgenerated char	如果为零字节（''），则不是生成的列。否则，s = stored。（将来可能会添加其他值。）
attisdropped bool	该列被删除且不再有效。一个删除的列仍然物理存在于表中，但是会被分析器忽略并因此无法通过SQL访问。
attislocal bool	该列是由关系本地定义的。注意一个列可以同时是本地定义和继承的。
attinhcount int4	



列类型	描述
	该列的直接祖先的编号。一个具有非零编号祖先的列不能被删除或者重命名。
attcollation oid (references pg_collation.oid)	该列被定义的排序规则，如果该列不是一个可排序数据类型则为0。
attacl aclitem[]	列级访问权限，如果此列上已有特别授予的权限
attoptions text[]	属性级选项，以“keyword=value”形式的字符串
attfdwoptions text[]	属性级的外部数据包装器选项，以“keyword=value”形式的字符串
attmissingval anyarray	这个列中是一个含有一个元素的数组，其中的值被用于该列在行中完全缺失时，如果在行创建之后增加一个有非易失DEFAULT值的列，就会发生这种情况。只有当atthasmissing为真时才使用这个值。如果没有值则该列为空。

在一个被删除的列的pg\_attribute的项中，atttypid被重置为0，但attlen以及其他从pg\_type拷贝的域仍然有效。这种安排用于处理一种情况，即被删除列的数据类型后来被删除，并且因此不再有相应的pg\_type行。attlen和其他域可以被用来解释表的一行的内容。

## 48.16. pg\_authid

目录pg\_authid包含关于数据库授权标识符（角色）的信息。角色把“用户”和“组”的概念包含在内。一个用户实际上就是一个rolcanlogin标志被设置的角色。任何角色（不管rolcanlogin设置与否）都能够把其他角色作为成员，参见pg\_auth\_members。

由于这个目录包含口令，它不能是公共可读的。pg\_roles是在pg\_authid上的一个公共可读视图，它隐去了口令域。

Chapter 19包含关于用户和权限管理的详细信息。

由于用户标识符是集簇范围的，pg\_authid在一个集簇的所有数据库之间共享：在一个集簇中只有一份pg\_authid拷贝，而不是每个数据库一份。

Table 48.16. pg\_authid Columns

列类型	描述
oid oid	行标识符
rolname name	角色名
rolsuper bool	角色有超级用户权限
rolinherit bool	如果本角色是另一个角色的成员，本角色是否自动另一个角色的权限
rolcreatorole bool	角色能创建更多角色
rolcreatedb bool	角色能创建数据库
rolcanlogin bool	角色是否能登录。即该角色是否能够作为初始会话授权标识符
rolreplication bool	

列类型	描述
	角色是一个复制角色。复制角色可以启动复制连接并且创建和删除复制槽。
rolbypassrls bool	角色是否可以绕过所有的行级安全性策略，详见Section 5.8
rolconnlimit int4	对于可以登录的角色，本列设置该角色可以同时发起最大连接数。-1表示无限制。
rolpassword text	密码（可能被加密过），如果没有口令则为空。格式取决于使用的加密方法的形式。
rolvaliduntil timestampz	口令过期时间（只用于口令鉴定），如果永不过期则为空

对于一个MD5加密的口令，rolpassword列将由字符串md5后面跟上一个32字符的十六进制MD5哈希值构成。MD5哈希值将是该用户的口令串接上它们的用户名。例如，如果用户joe的口令是xyzyzy，则LightDB将存储xyzyzyjoe的md5哈希。

如果口令采用SCRAM-SHA-256加密，它的格式是：

```
SCRAM-SHA-256$<iteration count>:<salt>$<StoredKey>:<ServerKey>
```

其中salt、StoredKey和ServerKey是Base64编码格式。这种格式与RFC 5803<sup>1</sup>说明的格式相同。

不遵守上述格式的口令被假定为未加密。

## 48.17. pg\_auth\_members

目录pg\_auth\_members展示了角色之间的成员关系。允许任何无环的关系集合。

由于用户标识符是集簇范围的，pg\_auth\_members在一个集簇的所有数据库之间共享：在一个集簇中只有一份pg\_auth\_members拷贝，而不是每个数据库一份。

Table 48.17. pg\_auth\_members Columns

列类型	描述
roleid oid (references pg_authid.oid)	拥有成员的角色的ID
member oid (references pg_authid.oid)	roleid的成员角色的ID
grantor oid (references pg_authid.oid)	授权此成员关系的角色的ID
admin_option bool	如果member能把roleid的成员关系授予他人，则为真(True)

## 48.18. pg\_cast

目录pg\_cast存储数据类型转换路径，包括内建的和用户定义的类型。

需要注意的是，pg\_cast并不表示系统知道如何执行的所有类型转换，它只包括哪些不能从某些普通规则推导出的转换。例如，一个域及其基类型之间的转换并未显式地在pg\_cast中

<sup>1</sup> <https://tools.ietf.org/html/rfc5803>

展示。另一个重要的例外是“自动 I/O转换造型”，它们通过数据类型自己的I/O函数来转换成（或者转换自）text或其他字符串类型，这些转换也没有显式地在pg\_cast中表示。

Table 48.18. pg\_cast Columns

列类型	描述
oid oid	行标识符
castsource oid (references pg_type.oid)	源数据类型的OID
casttarget oid (references pg_type.oid)	目标数据类型的OID
castfunc oid (references pg_proc.oid)	执行该转换的函数的OID。如果该转换方法不需要一个函数则存储0。
castcontext char	指示该转换能被调用的环境。 e表示仅能作为一个显式转换（使用CAST或::语法）。 a表示在赋值给目标列时隐式调用，和显式调用一样。 i表示在表达式中隐式调用，和其他转换一样。
castmethod char	指示转换如何被执行。 f表明使用castfunc中指定的函数。 i表明使用输入/输出函数。 b表明该类型是二进制可转换的，因此不需要转换。

在pg\_cast里列出的类型转换函数必须总是以转换的源类型作为它的第一个参数类型，并且返回转换的目标类型作为它的结果类型。一个类型转换函数最多有三个参数。如果出现了第二个参数，必须是integer类型；它接受与目标类型关联的修饰词，如果没有，就是-1。如果出现了第三个参数，那么必须是boolean类型；如果该类型转换是一种明确的转换，那么它接受true，否则接受false。

在pg\_cast里创建一条源类型和目标类型相同的记录是合理的，只要相关联的函数接受多过一个参数。这样的记录代表“长度转换函数”，它们把该类型的值转换为对特定的类型合法的值。

如果一个pg\_cast的项有着不同的原类型和目标类型，并且有一个接收多于一个参数的函数，那么它会在一个步骤中完成从一种类型到另外一种类型的转换并应用一个长度转换。如果没有这样的项，使用一个类型修改器的转换涉及两个步骤，一个是在数据类型之间转换，另外一个应用是应用修改器。

## 48.19. pg\_class

目录pg\_class记录表和几乎所有具有列或者像表的东西。这包括索引（但还要参见pg\_index）、序列（但还要参见pg\_sequence）、视图、物化视图、组合类型和TOAST表，参见relkind。下面，当我们提及所有这些类型的对象时我们使用“关系”。并非所有列对于所有关系类型都有意义。

Table 48.19. pg\_class Columns

列类型	描述
oid oid	行标识符
relname name	表、索引、视图等的名字
relnamespace oid (references pg_namespace.oid)	包含该关系的名字空间的OID

列类型	描述
reltype oid (references pg_type.oid)	可能存在的表行类型所对应数据类型的OID; 对索引为0, 序列, 和toast表, 索引没有pg_type项
reloftype oid (references pg_type.oid)	对于有类型的表, 为底层组合类型的OID, 对于其他所有关系为0
relowner oid (references pg_authid.oid)	关系的拥有者
relam oid (references pg_am.oid)	如果这是一个表或者索引, 表示索引使用的访问方法(堆、B树、哈希等); 否则为0 (0出现在序列, 以及没有存储的关系, 例如视图)
relfilenode oid	该关系的磁盘文件的名称, 0表示这是一个“映射”关系, 其磁盘文件名取决于底层状态
reltablespace oid (references pg_tablespace.oid)	该关系所存储的表空间。如果为0, 使用数据库的默认表空间。(如果关系无磁盘文件时无意义)
relpages int4	该表磁盘表示的尺寸, 以页面计(页面尺寸为BLCKSZ)。这只是一个由规划器使用的估计值。它被VACUUM、ANALYZE以及一些DDL命令(如CREATE INDEX)所更新。
reltuples float4	表中的存活行数。这只是一个由规划器使用的估计值。它被VACUUM、ANALYZE以及一些DDL命令(如CREATE INDEX)所更新。如果该表从未被清理或分析, 包含-1的reltuples表示行数是未知的。
relallvisible int4	在表的可见性映射表中被标记为全可见的页数。这只是一个由规划器使用的估计值。它被VACUUM、ANALYZE以及一些DDL命令, 如CREATE INDEX所更新。
reltoastrelid oid (references pg_class.oid)	与该表相关联的TOAST表的OID, 如果没有则为零。TOAST表将大属性“线外”存储在一个二级表中。
relhasindex bool	如果这是一个表并且其上建有(或最近建有)索引则为真
relisshared bool	如果该表在集簇中的所有数据库间共享则为真。只有某些系统目录(如pg_database)是共享的。
relpersistence char	p = 永久表, u = 无日志表, t = 临时表
relkind char	r = 普通表, i = 索引, S = 序列, t = TOAST表, v = 视图, m = 物化视图, c = 组合类型, f = 外部表, p = 分区表, I = 分区索引
relnatts int2	关系中用户列的数目(系统列不计算在内)。在pg_attribute中必须有这么多对应的项。另请参阅pg_attribute.attnum.。
relchecks int2	表上CHECK约束的数目, 参见pg_constraint目录
relhasrules bool	如果表有(或曾有)规则则为真, 参见pg_rewrite目录
relhastriggers bool	如果表有(或曾有)触发器则为真, 参见 pg_trigger目录

列类型	描述
relhassubclass bool	如果表或索引具有（或曾经具有）任何继承子项，则为真(True)
relrowsecurity bool	如果表上启用了行级安全性则为真，参见 pg_policy目录
relforcerowsecurity bool	如果行级安全性（启用时）也适用于表所有者则为真，参见 pg_policy目录
relispopulated bool	如果表已被填充则为真（对于所有关系该列都为真，但对于某些物化视图却不是）
relreplident char	用来为行形成“replica identity”的列： d = 默认（主键，如果存在）， n = 无， f = 所有列 i = 索引的indisreplident被设置或者为默认
relispartition bool	如果表或索引是一个分区，则为真
relrewrite oid (references pg_class.oid)	对于在要求表重写的DDL操作期间被写入的新关系，这个域包含原始关系的OID，否则为零。那种状态仅在内部可见，对于一个用户可见的关系这个域应该从不包含不是零的值。
relfrozenxid xid	在此之前的所有事务ID在表中已经被替换为一个永久的（“冻结的”）事务ID。这用于跟踪表是否需要被清理，以便阻止事务ID回卷或者允许lt_xact被收缩。如果该关系不是一个表则为0 (InvalidTransactionId)。
relminmxid xid	在此之前的多事务ID在表中已经被替换为一个事务ID。这被用于跟踪表是否需要被清理，以阻止 多事务ID回卷或者允许lt_multixact被收缩。如果关系不是一个表则 为0 (InvalidMultiXactId)。
relacl aclitem[]	访问权限，更多信息参见Section 5.7
reloptions text[]	访问方法相关的选项，以“keyword=value”字符串形式
relpartbound pg_node_tree	如果表示一个分区（见relispartition），分区边界的内部表达

pg\_class中的一些逻辑标志被以一种懒惰的方式维护：在正确状态时它们被保证为真，但是当条件不再为真时它们并不会被立刻重置为假。例如，relhasindex由CREATE INDEX设置，但它从不会被DROP INDEX清除。作为替代，VACUUM会在找到无索引表后清除其relhasindex。这种安排避免了竞争条件并且提高了并发性。

## 48.20. pg\_collation

目录pg\_collation描述了可用的排序规则，其本质是从一个SQL名字到操作系统locale分类的映射。更多信息参见Section 21.2

Table 48.20. pg\_collation Columns

列类型	描述
oid oid	行标识符
collname name	

列类型	描述
	排序规则名字（在每一个名字空间和编码中唯一）
collnamespace oid (references pg_namespace.oid)	包含该排序规则的名字空间的OID
collowner oid (references pg_authid.oid)	排序规则的拥有者
collprovider char	排序规则的提供者：d = 数据库默认，c = libc，i = icu
collisdeterministic bool	排序规则是确定性的吗？
collencoding int4	该排序规则可应用的编码，或以-1表示它可用于任何编码
collcollate name	该排序规则对象的LC_COLLATE
collctype name	该排序规则对象的LC_CTYPE
collversion text	排序规则的提供者相关的版本。这是在排序规则创建时记录下来的，并且在使用排序规则时会被检查以检测可能导致数据损坏的排序规则定义的改变。

注意在这个目录中的唯一键是 (collname、 collencoding、 collnamespace)， 不仅仅是 (collname, collnamespace)。 所有collencoding不等于当前数据库编码或-1的编码规则通常都会被LightDB忽略，且禁止创建和collencoding = -1的项重名的项。因此使用一个受限的SQL名字 (schema.name) 来标识一个排序规则是足够的，即使这根据目录定义是不唯一的。以这种方式定义这个目录的原因是initdb会在集簇初始化时使用系统上所有可用的locale填充这个目录，所以它必须能够为所有可能在集簇中使用的编码保持项。

在template0数据库中，创建与数据库编码不匹配的编码是有用的，因为它们可以匹配后面从template0克隆的数据库的编码。这在目前必须手动完成。

## 48.21. pg\_constraint

目录pg\_constraint存储表上的检查、主键、唯一、外键和排他约束。（列约束也不会被特殊对待。每一个列约束都等同于某种表约束。） 非空约束不在这里，而是在pg\_attribute目录中表示。

用户定义的约束触发器（使用CREATE CONSTRAINT TRIGGER创建）也会在这个表中产生一项。域上的检查约束也存储在这里。

Table 48.21. pg\_constraint Columns

列类型	描述
oid oid	行标识符
conname name	约束名字（不需要唯一！）
connamespace oid (references pg_namespace.oid)	包含此约束的名字空间的OID
contype char	c = 检查约束， f = 外键约束， p = 主键约束， u = 唯一约束， t = 约束触发器， x = 排他约束

列类型	描述
condeferrable bool	该约束是否能被延迟?
condeferred bool	该约束是否默认被延迟?
convalidated bool	此约束是否被验证过? 当前对于外键和检查约束只能是假
conrelid oid (references pg_class.oid)	该约束所在的表, 如果不是表约束则为零
contypid oid (references pg_type.oid)	该约束所在的域, 如果不是域约束则为0
conindid oid (references pg_class.oid)	如果该约束是唯一、主键、外键或排他约束, 此列表示支持此约束的索引, 否则为零
conparentid oid (references pg_constraint.oid)	如果这是一个分区中的约束, 则是父分区表中对应的约束; 否则为零
confrelid oid (references pg_class.oid)	如果此约束是一个外键约束, 此列为被引用的表, 否则为零
confupdtype char	外键更新动作代码: a = 无动作, r = 限制, c = 级联, n = 置空, d = 置为默认值
confdeltype char	外键删除动作代码: a = 无动作, r = 限制, c = 级联, n = 置空, d = 置为默认值
confmatchtype char	外键匹配类型: f = 完全, p = 部分, s = 简单
conislocal bool	此约束是定义在关系本地。注意一个约束可以同时是本地定义和继承。
coninhcount int4	此约束的直接继承祖先数目。一个此列非零的约束不能被删除或重命名。
connoinherit bool	为真表示此约束被定义在关系本地。它是一个不可继承约束。
conkey int2[] (references pg_attribute.attnum)	如果是一个表约束 (包括外键但不包括约束触发器), 此列是被约束列的列表
confkey int2[] (references pg_attribute.attnum)	如果是一个外键, 此列是被引用列的列表
conpfeqop oid[] (references pg_operator.oid)	如果是一个外键, 此列是用于PK = FK比较的等值操作符的列表
conppeqop oid[] (references pg_operator.oid)	如果是一个外键, 此列是用于PK = PK比较的等值操作符的列表
conffeqop oid[] (references pg_operator.oid)	如果是一个外键, 此列是用于FK = FK比较的等值操作符的列表
conexcllop oid[] (references pg_operator.oid)	如果是一个排他约束, 此列是每列排他操作符的列表
conbin pg_node_tree	如果是一个检查约束, 此列是表达式的一个内部表示。建议使用 pg_get_constraintdef() 提取检查约束的定义。

在一个排他约束的情况下，conkey只对约束元素是单一列引用时有用。对于其他情况，conkey为0且必须查阅相关索引来发现被约束的表达式。（对于索引，conkey因此和pg\_index.indkey具有相同的内容）。

Note

pg\_class.relchecks需要和每个关系在此目录中的检查约束数量保持一致。

## 48.22. pg\_conversion

目录pg\_conversion描述编码转换函数。更多信息参见CREATE CONVERSION。

Table 48.22. pg\_conversion Columns

列类型	描述
oid oid	行标识符
conname name	转换的名字（在一个名字空间内唯一）
connamespace oid (references pg_namespace.oid)	包含此转换的名字空间的OID
conowner oid (references pg_authid.oid)	转换的拥有者
conforencoding int4	源编码ID
contoencoding int4	目标编码ID
conproc regproc (references pg_proc.oid)	转换函数
condefault bool	如果这是默认转换则为真

## 48.23. pg\_database

目录pg\_database存储有关可用数据库的信息。数据库通过CREATE DATABASE命令创建。更多关于其参数的信息请查阅Chapter 20

和大部分系统目录不同，pg\_database是在集簇的所有数据库之间共享的：在一个集簇中只有一份pg\_database拷贝，而不是每个数据库一份。

Table 48.23. pg\_database Columns

列类型	描述
oid oid	行标识符
datname name	数据库名称
datdba oid (references pg_authid.oid)	数据库的拥有者，通常是创建它的用户



列类型	描述
encoding int4	此数据库的字符编码的编号 (pg_encoding_to_char() 可将此编号转换成编码的名字)
datcollate name	此数据库的LC_COLLATE
datctype name	此数据库的LC_CTYPE
datistemplate bool	如果为真, 则此数据库可被任何具有CREATEDB特权的用户克隆; 如果为假, 则只有超级用户或者该数据库的属主能够克隆它。
datallowconn bool	如果为假则没有人能连接到这个数据库。这可以用来保护template0数据库不被修改。
datconlimit int4	设置能够连接到这个数据库的最大并发连接数。-1表示没有限制。
datlastsysoid oid	数据库中最后一个系统OID, 对lt_dump特别有用
datfrozenxid xid	在此之前的所有事务ID在数据库中已经被替换为一个永久的 (“冻结的”) 事务ID。 这用于跟踪数据库是否需要被清理, 以便组织事务ID回环或者允许lt_xact被收缩。 它是此数据库中所有表的pg_class.relfrozenxid 值的最小值。
datminmxid xid	在此之前的所有多事务ID在数据库中已经被替换为一个事务ID。这用于跟踪数据库是否需要被清理, 以便组织事务ID回环或者允许lt_multixact被收缩。它是此数据库中 所有表的pg_class.relminmxid值的最小值。
dattablespace oid (references pg_tablespace.oid)	此数据库的默认表空间。 在此数据库中, 所有pg_class.reltablespace为0的表都将被存储在这个表空间中; 尤其是非共享系统目录都会在其中。
datacl aclitem[]	访问权限, 更多信息参见Section 5.7

## 48.24. pg\_db\_role\_setting

目录 pg\_db\_role\_setting为每一个角色和数据库组合记录被设置到运行时配置变量的默认值。

和大部分系统目录不同, pg\_db\_role\_setting是在集簇的所有数据库之间共享的: 在一个集簇中只有一份pg\_db\_role\_setting拷贝, 而不是每个数据库一份。

Table 48.24. pg\_db\_role\_setting Columns

列类型	描述
setdatabase oid (references pg_database.oid)	此设置可用的数据库OID, 如果不与具体数据库相关则为0
setrole oid (references pg_authid.oid)	此设置可用的角色OID, 如果不与具体角色相关则为0
setconfig text[]	运行时配置变量的默认值

## 48.25. pg\_default\_acl

目录pg\_default\_acl存储要被分配给新创建对象的初始权限。

Table 48.25. pg\_default\_acl Columns

列类型	描述
oid oid	行标识符
defaclrole oid (references pg_authid.oid)	与此项相关的角色的OID
defaclnamespace oid (references pg_namespace.oid)	与此项相关的名字空间的OID, 如果没有则为零
defaclobjtype char	此项适合的对象类型: r = 关系 (表、视图), S = 序列, f = 函数, T = 类型, n = 模式
defaclacl aclitem[]	此类对象在创建时应用有的访问权限

一个pg\_default\_acl项展示了要分配给属于一个指定用户的对象的初始权限。当前有两类项: defaclnamespace = 零的“全局”项和引用一个特殊模式的“per-schema”项。如果一个全局项存在, 则它重载该对象类型的普通hard-wired默认权限。一个每模式项如果存在, 表示权限将被加入到全局或hard-wired默认权限中。

注意当在另一个表中的一个ACL项为空时, 它用来表示其对象的hard-wired默认权限, 而不是当时可能在pg\_default\_acl中的任何权限。只有在对象创建期间才会查阅pg\_default\_acl。

## 48.26. pg\_depend

目录pg\_depend记录数据库对象之间的依赖关系。这些信息允许DROP命令查找必须被DROP CASCADE删除的其他对象, 或者在DROP RESTRICT情况下阻止删除。

另请参阅pg\_shdepend, 它对在一个数据库集簇中共享的对象之间的依赖提供了相似的功能。

Table 48.26. pg\_depend Columns

列类型	描述
classid oid (references pg_class.oid)	依赖对象所在的系统目录的OID, 或对于DEPENDENCY_PIN条目为零。
objid oid (references any OID column)	指定依赖对象的OID, 或对于DEPENDENCY_PIN条目为零。
objsubid int4	对于一个表列, 这里是列号 (objid和classid指表本身)。对于所有其他对象类型, 此列为0。
refclassid oid (references pg_class.oid)	被引用对象所在的系统目录的OID
refobjid oid (references any OID column)	指定被引用对象的OID
refobjsubid int4	

列类型	描述
	对于一个表列，这里是列号（refobjid和refclassid指表本身）。对于所有其他对象类型，此列为0。
deptype char	定义此依赖关系语义的一个代码，见文本

在所有情况下，一个pg\_depend项表明被引用对象不能在没有删除其依赖对象的情况下被删除。但是，其中也有多种依赖类型，由deptype标识：

DEPENDENCY\_NORMAL (n)

在独立创建的对象之间的一个普通关系。依赖对象可以在不影响被依赖对象的情况下被删除。被引用对象只能通过指定CASCADE被删除，在这种情况下依赖对象也会被删除。例子：一个表列对于其数据类型有一个普通依赖。

DEPENDENCY\_AUTO (a)

依赖对象可以被独立于被依赖对象删除，且应该在被引用对象被删除时自动被删除（不管在RESTRICT或CASCADE模式）。例子：一个表上的一个命名约束应该被设置为自动依赖于表，这样在表被删除后它也会消失。

DEPENDENCY\_INTERNAL (i)

依赖对象作为被引用对象创建过程的一部分被创建，并且只是其内部实现的一部分。不允许直接DROP所依赖的对象（而是告诉用户对引用对象发出DROP操作）。无论是否指定了CASCADE，DROP被引用的对象都将导致自动删除从属对象。如果由于删除了对某些其他对象的依赖关系而不得不删除依赖对象，则其删除将转换为对所引用对象的删除，因此依赖对象的NORMAL和AUTO依赖关系的行为就像它们是所引用对象的依赖关系。示例：视图的ON SELECT规则使其在内部依赖于视图，以防止在视图保留时将其删除。规则的依赖关系（例如它引用的表）就好像他们是视图的依赖关系。

DEPENDENCY\_PARTITION\_PRI (P)

DEPENDENCY\_PARTITION\_SEC (S)

依赖对象被作为被引用对象创建过程的一部分创建，并且确实是其内部实现的一部分。但是，不像INTERNAL，有多个这样的引用对象。除非删除了这些引用对象中的至少一个对象，否则不得删除依赖对象；如果其中任何一个被删除，则不管是否指定了CASCADE，都应删除依赖对象。也不像INTERNAL，依赖对象所依赖的某些其他对象的删除不会导致任何分区引用的对象的自动删除。因此，如果删除没有通过其他路径级联到这些对象中的至少一个，它会被拒绝。（大多数情况下，依赖对象与至少一个分区引用对象共享所有非分区的依赖关系，因此此限制不会导致阻止任何级联的删除。）主分区和辅助分区的依赖关系表现相同，除了主分区依赖关系倾向于用于错误消息；因此，分区相关的对象应该有一个主分区依赖关系和一个或多个辅助分区依赖关系。注意到分区依赖关系是任何对象所正常拥有的依赖关系的补充，而不是替代。这简化了ATTACH/DETACH PARTITION操作：只要添加或删除分区的依赖关系。例如：子分区索引与其所基于的分区表和父分区索引是分区相关的，因此只要其中一个删除，则子分区索引就消失，否则，就不消失。父索引上的依赖关系是主要的，故如果用户试图删除子分区索引，错误消息反而会建议删除父索引（不是表）。

DEPENDENCY\_EXTENSION (e)

依赖对象是作为扩展的被引用对象的一个成员（参见pg\_extension）。依赖对象可以通过被引用对象上的DROP EXTENSION来删除。在功能上，这种依赖类型和一个INTERNAL依赖的作用相同，其存在只是为了清晰和简化lt\_dump。

DEPENDENCY\_AUTO\_EXTENSION (x)

依赖对象不是作为被引用对象的扩展的成员（因此不应该被lt\_dump忽略），但是没有该扩展它又无法工作，因此如果删除了扩展，则该依赖对象应自动删除。该依赖对象也可

以独立删除。功能上，该依赖关系类型与AUTO依赖相同，但是为了清晰起见和简化lt\_dump，将其分开。

DEPENDENCY\_PIN (p)

没有依赖对象，这种类型的项是一个信号，用于说明系统本身依赖于被引用对象，并且该对象永远不能被删除。这种类型的项只能被initdb创建。而此种项的依赖对象的列都为0。

在未来可能会需要其他依赖类型。

要注意的是，两个对象很有可能由不止一个pg\_depend条目来链接。例如子分区索引有一个依赖于其关联的分区表的分区类型的依赖关系和依赖于该表索引的每一列的自动依赖关系。此类情形表示多重依赖关系语义的并集，依赖对象的删除可以没有CASCADE，如果其任一依赖关系满足自动删除的条件。相反地，关于哪些对象必须一起删除的所有依赖关系的限制必须满足。

## 48.27. pg\_description

目录pg\_description存储对每一个数据库对象可选的描述（注释）。描述可以通过COMMENT操作，并可使用ltsql的\d命令查看。在pg\_description的初始内容中提供了很多内建系统对象的描述。

参见pg\_shdescription，它对在一个数据库集簇中共享的对象的描述提供了相似的功能。

Table 48.27. pg\_description Columns

列类型	描述
objoid oid (references any OID column)	描述所属对象的OID
classoid oid (references pg_class.oid)	对象所述的系统目录的OID
objsubid int4	对于一个表列上的一个注释，这里是列号（objoid和classoid指表本身）。对所有其他对象类型，此列为0。
description text	作为该对象描述的任意文本

## 48.28. pg\_enum

pg\_enum目录包含每一个枚举类型的项，其中包括了值和标签。一个给定枚举值的内部表示实际上是它在pg\_enum中的相关行的OID。

Table 48.28. pg\_enum Columns

列类型	描述
oid oid	行标识符
enumtypid oid (references pg_type.oid)	包含此枚举值的pg_type项的OID
enumsortorder float4	此枚举值在其枚举类型中的排序位置
enumlabel name	此枚举值的文本标签

pg\_enum行的OID值遵循一种特殊的规则：即OID的数值被保证按照其枚举类型的排序顺序进行排序。即如果两个偶数OID属于同一枚举类型，较小的OID必然具有较小的enumsortorder值。奇数OID值不需要遵循排序顺序。这种规则使得枚举比较例程在很多常见情况下可以避免系统目录查找。创建和修改枚举类型的例程将尝试尽可能地为枚举值分配偶数OID。

当一个枚举类型被创建后，其成员会被分配排序位置1..n。但后面增加的成员可能会被分配负值或者分数值的enumsortorder。对于这些值的唯一要求是它们必须被正确地排序且和保持唯一。

## 48.29. pg\_event\_trigger

目录pg\_event\_trigger存储事件触发器。更多信息参见Chapter 38

Table 48.29. pg\_event\_trigger Columns

列类型	描述
oid oid	行标识符
evtname name	触发器名（必须唯一）
evtevent name	此触发器触发的事件的标识符
evtowner oid (references pg_authid.oid)	事件触发器的拥有者
evtfoid oid (references pg_proc.oid)	将被调用的函数
evtenabled char	控制事件触发器触发的session_replication_role模式。 0 = 触发器在“origin”和“local”模式触发， D = 触发器被禁用， R = 触发器在“replica”模式触发， A = 触发器总是触发。
evttags text[]	此触发器将触发的命令标签。如果为空，此触发器的触发不受命令标签的限制。

## 48.30. pg\_extension

目录pg\_extension存储有关已安装扩展的信息。有关扩展的细节请参见Section 36.17

Table 48.30. pg\_extension Columns

列类型	描述
oid oid	行标识符
extname name	扩展的名字
extowner oid (references pg_authid.oid)	扩展的拥有者
extnamespace oid (references pg_namespace.oid)	包含此扩展的导出对象的模式
extrelocatable bool	如果扩展可被重定位到另一个模式则为真

列类型	描述
extversion text	扩展的版本名字
extconfig oid[] (references pg_class.oid)	扩展的配置表的regclass项的OID数组，如果没有配置表则为NULL
extcondition text[]	扩展的配置表的WHERE子句过滤条件的数组，如果没有则为NULL

注意和大部分具有一个“namespace”列的模式不同，extnamespace不是用来表示扩展属于该模式。扩展的名字从不用模式进行限定。extnamespace表明该模式包含了该扩展的大部分或全部对象。如果extrelocatable为真，则该模式事实上必须包含属于此扩展的全部模式限定的对象。

## 48.31. pg\_foreign\_data\_wrapper

目录pg\_foreign\_data\_wrapper存储外部数据包装器定义。外部数据包装器是一种访问位于外部服务器上数据的机制。

Table 48.31. pg\_foreign\_data\_wrapper Columns

列类型	描述
oid oid	行标识符
fdwname name	外部数据包装器的名字
fdwowner oid (references pg_authid.oid)	外部数据包装器的拥有者
fdwhandler oid (references pg_proc.oid)	指一个负责为外部数据包装器提供执行例程的处理函数。如果没有提供处理函数则为0
fdwvalidator oid (references pg_proc.oid)	指一个负责检查传给外部数据包装器的选项的有效性的验证函数，包括外部服务器选项以及使用外部数据包装器的用户映射。如果没有提供验证函数则为0
fdwacl aclitem[]	访问权限，详见Section 5.7
fdwoptions text[]	外部数据包装器特定选项，以“keyword=value”字符串形式

## 48.32. pg\_foreign\_server

目录pg\_foreign\_server存储外部服务器定义。外部服务器定义了外部数据的来源，例如一个远程服务器。外部服务器通过外部数据包装器来访问。

Table 48.32. pg\_foreign\_server Columns

列类型	描述
oid oid	行标识符
srvname name	

列类型	描述
	外部服务器的名字
srvowner oid (references pg_authid.oid)	外部服务器的拥有者
srvfdw oid (references pg_foreign_data_wrapper.oid)	此外部服务器的外部数据包装器的OID
srvtype text	服务器的类型 (可选)
srvversion text	服务器的版本 (可选)
srvacl aclitem[]	访问权限, 详见Section 5.7
srvoptions text[]	外部服务器特定选项, 以“keyword=value”字符串形式

## 48.33. pg\_foreign\_table

目录pg\_foreign\_table包含关于外部表的辅助信息。一个外部表和普通表一样, 主要由一个pg\_class项表示。它的pg\_foreign\_table项包含外部表所特有的信息。

Table 48.33. pg\_foreign\_table Columns

列类型	描述
ftrelid oid (references pg_class.oid)	外部表的pg_class项的OID
ftserver oid (references pg_foreign_server.oid)	外部表所在的外部服务器的OID
ftoptions text[]	外部表选项, 以“keyword=value”字符串形式

## 48.34. pg\_index

目录pg\_index包含关于索引的部分信息。其他信息大部分在pg\_class中。

Table 48.34. pg\_index Columns

列类型	描述
indexrelid oid (references pg_class.oid)	此索引的pg_class项的OID
indrelid oid (references pg_class.oid)	此索引所对应的基表的pg_class项的OID
indnatts int2	索引中的总列数 (与pg_class.relnatts重复), 这个数目包括键和被包括的属性
indnkeyatts int2	索引中键列的编号, 不计入任何的内含列, 它们只是被存储但不参与索引的语义
indisunique bool	如为真, 这是唯一索引
indisprimary bool	

列类型	描述
	如为真，表示索引为表的主键（如果此列为真，indisunique也总是为真）
indisexclusion bool	如为真，此索引支持一个排他约束
indimmediate bool	如为真，唯一性检查在插入时立即被执行（如果indisunique为假，此列无关）
indisclustered bool	如果为真，表示表最后以此索引进行了聚簇
indisvalid bool	如果为真，此索引当前可以用于查询。为假表示此索引可能不完整：它肯定还在被INSERT/UPDATE操作所修改，但它不能安全地被用于查询。如果索引是唯一索引，唯一性属性也不能被保证。
indcheckxmin bool	如果为真，直到此pg_index行的xmin低于查询的TransactionXmin视界之前，查询都不能使用此索引，因为表可能包含具有它们可见的不相容行的损坏HOT链
indisready bool	如果为真，表示此索引当前可以用于插入。为假表示索引必须被INSERT/UPDATE操作忽略。
indislive bool	如果为假，索引正处于被删除过程中，并且必须被所有处理忽略（包括HOT安全的决策）
indisreplident bool	如果为真，这个索引被选择为使用ALTER TABLE ... REPLICA IDENTITY USING INDEX ...的“replica identity”
indkey int2vector (references pg_attribute.attnum)	这是一个indnatts值的数组，它表示了此索引索引的表列。例如一个1 3值可能表示表的第一和第三列组成了索引项。键列出现在非键（内含）列前面。数组中的一个0表示对应的索引属性是一个在表列上的表达式，而不是一个简单的列引用。
indcollation oidvector (references pg_collation.oid)	对于索引键（indnkeyatts值）中的每一列，这包含要用于该索引的排序规则的OID，如果该列不是一种可排序数据类型则为零。
indclass oidvector (references pg_opclass.oid)	对于索引键中的每一列（indnkeyatts值），这里包含了要使用的操作符类的OID。详见pg_opclass。
indoption int2vector	这是一个indnkeyatts值的数组，用于存储每列的标志位。位的意义由索引的访问方法定义。
indexprs pg_node_tree	非简单列引用索引属性的表达式树（以nodeToString()形式）。对于indkey中每一个为0的项，这个列表中都含有一个元素。如果所有的索引属性都是简单引用，此列为空。
indpred pg_node_tree	部分索引谓词的表达式树（以nodeToString()形式）。如果不是部分索引，此列为空。

## 48.35. pg\_inherits

目录pg\_inherits记录有关表和索引的继承层次的信息。数据库中每一个直接父子表和索引关系在这里都有一项（非直接继承可以通过顺着项构成的链来决定）。



Table 48.35. pg\_inherits Columns

列类型	描述
inhrelid oid (references pg_class.oid)	子表或索引的OID
inhparent oid (references pg_class.oid)	父表或索引的OID
inhseqno int4	如果一个孩子表有多于一个直接父表（多继承），这个数字说明了继承列被排列的顺序。计数从1开始。 索引不能具有多个继承，因为它们只能在使用声明性分区时继承。

## 48.36. pg\_init\_privs

目录pg\_init\_privs记录系统中对象的初始特权。数据库中每一个具有非默认（非-NULL）初始特权集合的对象都有一个条目在其中。

对象可以在系统初始化（initdb）时获得其初始特权，也可以在CREATE EXTENSION期间创建该对象并且在扩展脚本中用GRANT来设置对象的初始特权。注意系统将自动处理扩展脚本执行期间对特权的记录，扩展的作者们只需要在他们的脚本中使用GRANT以及REVOKE语句以便特权被记录下来。privtype列表示初始特权是被initdb设置还是在一次CREATE EXTENSION命令期间被设置。

具有被initdb设置的初始特权的对象的条目中privtype是'i'，而具有被CREATE EXTENSION设置的初始特权的对象的条目中privtype为'e'。

Table 48.36. pg\_init\_privs Columns

列类型	描述
objoid oid (参考任何 OID 列)	指定对象的 OID
classoid oid (references pg_class.oid)	对象所在的系统目录的 OID
objsubid int4	对于一个表列，这里是列编号（objoid和classoid指向表本身）。对于所有其他对象类型，这列为零。
privtype char	定义这个对象初始特权类型的代码，见文字说明
initprivs aclitem[]	初始的访问权限，详见Section 5.7

## 48.37. pg\_language

目录pg\_language注册了可用于编写函数或存储过程的语言。更多关于语言处理器的信息请参阅CREATE LANGUAGE和Chapter 40

Table 48.37. pg\_language Columns

列类型	描述
oid oid	行标识符

列类型	描述
lanname name	语言的名字
lanowner oid (references pg_authid.oid)	语言的拥有者
lanispl bool	内部语言为假（如SQL），用户定义语言为真。当前，lt_dump仍然使用这个列来决定要转储哪些语言，但在未来这可能会被一种不同的机制所取代。
lanpltrusted bool	为真表示这是一种可信的语言，即它被相信不会向普通SQL执行环境之外的任何东西授予权限。只有超级用户可以在非可信语言中创建函数。
lanplcallfoid oid (references pg_proc.oid)	对于非内部语言，此列引用语言处理器，它是一个特殊函数负责执行所有用这种语言编写的函数。对于内部语言用零。
laninline oid (references pg_proc.oid)	此列引用一个负责执行“内联”匿名代码块的函数（DO 块）。如果不支持内联块则为0。
lanvalidator oid (references pg_proc.oid)	此列引用一个负责在函数创建时对其进行语法和可用性检查的语言验证函数。如果没有提供验证器则为0。
lanacl aclitem[]	访问权限，详情参见Section 5.7

## 48.38. pg\_largeobject

目录pg\_largeobject保存构成“大对象”的数据。一个大对象在被创建时会被分配一个OID。每个大对象被分解成段或“页”，以便小到可以被方便地作为行存储在pg\_largeobject中。每页中的数据量被定义为LOBLKSIZE（目前是BLCKSZ/4或是2 kB）。

在LightDB 9.0之前，大对象没有相关的权限结构。作为结果，pg\_largeobject是公共可读的并且可以用来获得系统中所有大对象的OID（和内容）。但现在不是这样了，可使用pg\_largeobject\_metadata来获得大对象OID的列表。

Table 48.38. pg\_largeobject Columns

列类型	描述
loid oid (references pg_largeobject_metadata.oid)	包含此页的大对象的标识符
pageno int4	此页在它所属大对象中的页号（从0开始计）
data bytea	实际存储在大对象中的数据。它从不会超过LOBLKSIZE字节，也可能更少。

pg\_largeobject的每一行保存一个大对象的一个页的数据，从对象内部的字节偏移量（pageno \* LOBLKSIZE）开始。现在的实现允许稀疏存储：页面可能丢失，并且可能比LOBLKSIZE字节短（即便不是最后一页）。一个大对象中丢失的区域会被读出为0。

## 48.39. pg\_largeobject\_metadata

目录pg\_largeobject\_metadata保持着与大对象有关的元数据。真正的大对象数据被存储在pg\_largeobject中。

Table 48.39. pg\_largeobject\_metadata Columns

列类型	描述
oid oid	行标识符
lomowner oid (references pg_authid.oid)	大对象的拥有者
lomacl aclitem[]	访问权限, 详见Section 5.7

## 48.40. pg\_namespace

目录pg\_namespace存储名字空间。名字空间是SQL模式之下的结构：每个名字空间拥有一个独立的表、类型等的集合，且其中没有名字冲突。

Table 48.40. pg\_namespace Columns

列类型	描述
oid oid	行标识符
nspname name	名字空间的名字
nspowner oid (references pg_authid.oid)	名字空间的拥有者
nspacl aclitem[]	访问权限, 详见Section 5.7

## 48.41. pg\_opclass

目录pg\_opclass定义索引访问方法的操作符类。每一个操作符类定义了一种特定数据类型和一种特定索引访问方法的索引列的语义。一个操作符类实际上指定了一个特定的操作符族可以用于一个特定索引列数据类型。该族中可用于索引列的操作符能够接受该列的数据类型作为它们的左输入。

操作符类详见Section 36.16

Table 48.41. pg\_opclass Columns

列类型	描述
oid oid	行标识符
opcmethod oid (references pg_am.oid)	操作符类所属的索引访问方法
opcname name	操作符类的名称
opcnamespace oid (references pg_namespace.oid)	操作符类所属的名字空间
opcowner oid (references pg_authid.oid)	操作符类的拥有者
opcfamily oid (references pg_opfamily.oid)	

列类型	描述
	包含此操作符类的操作符系列
opcintype oid (references pg_type.oid)	操作符类索引的数据类型
opcdefault bool	如果此操作符类为opcintype的默认值则为真
opckeytype oid (references pg_type.oid)	存储在索引中的数据的类型, 如果值为0表示与opcintype相同

一个操作符类的opcmethod必须匹配包含它的操作符族的opfmethod。而且, 对于任何给定的opcmethod和opcintype组合, 只有不超过一个pg\_opclass行的opcdefault值为真。

## 48.42. pg\_operator

目录pg\_operator存储关于操作符的信息。详见CREATE OPERATOR和Section 36.14

Table 48.42. pg\_operator Columns

列类型	描述
oid oid	行标识符
oprname name	操作符的名称
oprnamespace oid (references pg_namespace.oid)	操作符所属的名字空间的OID
oprowner oid (references pg_authid.oid)	操作符的拥有者
oprkind char	b = 中缀操作符 (“并”), 或l = 前缀操作符 (“左”)
oprcanmerge bool	该操作符支持归并连接
oprcanhash bool	该操作符支持哈希连接
oprleft oid (references pg_type.oid)	左操作数类型 (对于前缀操作符用零)
oprright oid (references pg_type.oid)	右操作数类型
oprresult oid (references pg_type.oid)	结果类型 (对于尚未定义的“shell”操作符为零)
oprcom oid (references pg_operator.oid)	该操作符的交换子 (如没有则为零)
oprnegate oid (references pg_operator.oid)	该操作符的否定 (如没有则为零)
oprcode regproc (references pg_proc.oid)	实现该操作符的函数 (没有定义的“shell”操作符则为零)
oprrest regproc (references pg_proc.oid)	该操作符的限制选择度估算函数 (如果没有则为零)
oprjoin regproc (references pg_proc.oid)	

列类型	描述
	该操作符的连接选择度估算函数（如果没有则为零）

未使用的列包含零。例如，对于前缀运算符，oprleft为零。

## 48.43. pg\_opfamily

目录pg\_opfamily定义了操作符族。每一个操作符族是操作符和相关支持例程的集合，支持例程用于实现一个特定索引访问方法的语义。此外，按照访问方法指定的某种方式，一个族内的操作符都是“兼容的”。操作符族概念允许在索引中使用跨数据类型操作符，并可以使用访问方法语义的知识推导出。

操作符族最终在Section 36.16中描述。

Table 48.43. pg\_opfamily Columns

列类型	描述
oid oid	行标识符
opfmethod oid (references pg_am.oid)	操作符族适用的索引访问方法
opfname name	操作符系列的名字
opfnamespace oid (references pg_namespace.oid)	操作符系列所属的名字空间
opfowner oid (references pg_authid.oid)	操作符系列的拥有者

定义操作符族的主要信息不在它的pg\_opfamily行，而是在相关的pg\_amop、pg\_amproc和pg\_opclass行中。

## 48.44. pg\_partitioned\_table

目录pg\_partitioned\_table存放有关表如何被分区的信息。

Table 48.44. pg\_partitioned\_table Columns

列类型	描述
partrelid oid (references pg_class.oid)	这个分区表的pg_class项的OID
partstrat char	分区策略；h = 哈希分区表，l = 列表分区表，r = 范围分区表
partnatts int2	分区键中的列数
partdefid oid (references pg_class.oid)	这个分区表的默认分区的pg_class项的OID，如果这个分区表没有默认分区则为零。
partattrs int2vector (references pg_attribute.attnum)	这是一个长度为partnatts值的数组，它指示哪些表列是分区键的组成部分。例如，值1 3表示第一个和第三个表列组成了分区键。这个数组中的零表示对应的分区键列是一个表达式而不是简单的列引用。

列类型	描述
partclass oidvector (references pg_opclass.oid)	对于分区键中的每一个列，这个域包含要使用的操作符类的OID。 详见pg_opclass。
partcollation oidvector (references pg_collation.oid)	对于分区键中的每一个列，这个域包含要用于分区的排序规则的OID，如果该列不是一种可排序数据类型则为零。
partexprs pg_node_tree	非简单列引用的分区键列的表达式树（以nodeToString()的表达方式）。 这是一个列表，partattrs中每一个零项都有一个元素。如果所有分区键列都是简单列引用，则这个域为空。

## 48.45. pg\_policy

目录pg\_policy存储着表的行级安全性策略。 一个策略包括它适用于的命令种类（可能适用于所有命令）、它适用于的角色、 被作为安全屏障条件增加到包括该表的查询的表达式以及被作为 WITH CHECK选项增加到尝试向表增加新纪录的查询的表达式。

Table 48.45.pg\_policy Columns

列类型	描述
oid oid	行标识符
polname name	策略的名称
polrelid oid (references pg_class.oid)	策略适用的表
polcmd char	策略适用的命令类型： r 表示 SELECT, a 表示 INSERT, w 表示 UPDATE, d 表示 DELETE, *表示所有命令类型
polpermissive bool	策略是宽容性的还是限制性的？
polroles oid[] (references pg_authid.oid)	策略适用的角色；零意味着PUBLIC(通常在数组中单独出现)
polqual pg_node_tree	被作为安全屏障条件增加到使用该表的查询的表达式树
polwithcheck pg_node_tree	被作为WITH CHECK 条件增加到尝试向表增加行的查询的表达式树

### Note

存储在pg\_policy中的策略只有在它们所适用的表的pg\_class.relrowsecurity被设置时才起作用。

## 48.46. pg\_proc

目录pg\_proc存放有关函数、过程、聚集函数以及窗口函数（共称为例程）的信息。更多信息请参考CREATE FUNCTION、CREATE PROCEDURE和Section 36.3

如果prokind显示该条目用于一个聚集函数，在pg\_aggregate中应该有一个相匹配的行。

Table 48.46. pg\_proc Columns

列类型	描述
oid oid	行标识符
proname name	函数的名字
pronamespace oid (references pg_namespace.oid)	函数所属的名字空间的OID
proowner oid (references pg_authid.oid)	函数的拥有者
prolang oid (references pg_language.oid)	实现语言或该函数的调用接口
procost float4	估计的执行代价（以cpu_operator_cost为单位），如果proretset为真，这是每行返回的代价
prorows float4	估计的结果行数量（如果proretset为假，该值为0）
provariadic oid (references pg_type.oid)	可变数组参数的元素的数据类型，如果函数没有可变参数则为0
prosupport regproc (references pg_proc.oid)	对于该函数的计划器支持函数（参见Section 36.11），如果没有则为零
prokind char	f表示普通函数，p表示过程，a表示聚集函数，w表示窗口函数
prosecdef bool	函数是一个安全性定义者（即，一个“setuid”函数）
proleakproof bool	该函数没有副作用。除了通过返回值，没有关于参数的信息被传播。任何会抛出基于其参数值的错误信息的函数都不是泄露验证的。
proisstrict bool	当任意调用函数为空时，函数是否会返回空值。在那种情况下函数实际上根本不会被调用。非“strict”函数必须准备好处理空值输入。
proretset bool	函数是否返回一个集合（即，指定数据类型的多个值）
provolatile char	provolatile说明函数是仅仅只依赖于它的输入参数，还是会被外部因素影响。值i表示“不变的”函数，它对于相同的输入总是输出相同的结果。值s表示“稳定的”函数，它的结果（对于固定输入）在一次扫描内不会变化。值v表示“不稳定的”函数，它的结果在任何时候都可能变化（使用v页表示函数具有副作用，所以对它们的调用无法得到优化）
proparallel char	proparallel说明该函数在并行模式下是否能安全地运行。对于能在并行模式下不受限制安全运行的函数，这列是s。对于可以在并行模式下运行但是只限于由并行分组的领导者执行的函数，这列是r。对于在并行模式中不安全的函数，这列是u，这种函数的存在会强制一个顺序执行计划。
pronargs int2	输入参数的个数
pronargdefaults int2	

列类型	描述
	具有默认值的参数个数
proreftype oid (references pg_type.oid)	返回值的数据类型
proargtypes oidvector (references pg_type.oid)	一个函数参数的数据类型数组。这只包括输入参数（含INOUT和VARIADIC参数），因此也表现了函数的调用特征。
proallargtypes oid[] (references pg_type.oid)	一个函数参数的数据类型数组。这包括所有参数（含OUT和INOUT参数）。但是，如果所有参数都是IN参数，这个域将为空。注意下标是从1开始，然而由于历史原因proargtypes的下标是从0开始。
proargmodes char[]	一个函数参数的模式的数组。这里包括：i表示IN参数，o表示OUT参数，b表示INOUT参数，v表示VARIADIC参数，t表示TABLE参数。如果所有的参数都是IN参数，这个域为空。注意这里的下标对应着proallargtypes而不是proargtypes中的位置。
proargnames text[]	一个函数参数的名字的数组。没有名字的参数在数组中设置为空字符串。如果没有一个参数有名字，这个域为空。注意这里的下标对应着proallargtypes而不是proargtypes中的位置。
proargdefaults pg_node_tree	默认值的表达式树（按照nodeToString()的表现方式）。这是一个proargdefaults元素的列表，对应于最后N个input参数（即最后N个proargtypes位置）。如果没有一个参数具有默认值，这个域为空。
protrftypes oid[] (references pg_type.oid)	要应用转换的（来自函数的TRANSFORM子句）参数/结果数据类型的数组。如果没有则为空。
prosrc text	这个域告诉函数处理器如何调用该函数。它可能是针对解释型语言的真实源码、一个符号链接、一个文件名或任何其他东西，这取决于实现语言/调用规范。
probin text	关于如何调用函数的附加信息。其解释是与语言相关的。
proconfig text[]	函数对于运行时配置变量的本地设置值
proacl aclitem[]	访问权限，详见Section 5.7

对于编译好的函数，包括内建的和动态载入的，prosrc包含了函数的C语言名字（链接符号）。对于SQL-语言函数，prosrc包含着函数的源文本，如果它被指定为字符串；但是如果函数体被指定为函数标准样式，prosrc是不使用的（通常是空字符串）并且由prosqlbody包含预解析的定义。所有其他已知的语言类型，prosrc包含函数的源码文本。除了对于动态载入的C函数之外，probin为空，对于动态载入的C函数，它给定了包含该函数的共享库文件的名称。

## 48.47. pg\_publication

目录pg\_publication包含数据库中创建的所有publication。更多关于publication的内容请见Section 28.1



Table 48.47. pg\_publication Columns

列类型	描述
oid oid	行标识符
pubname name	publication的名称
pubowner oid (references pg_authid.oid)	publication的拥有者
puballtables bool	如果为真, 这个publication自动包括数据库中的所有表, 包括未来将会创建的任何表。
pubinsert bool	如果为真, 为publication中的表复制INSERT操作。
pubupdate bool	如果为真, 为publication中的表复制UPDATE操作。
pubdelete bool	如果为真, 为publication中的表复制DELETE操作。
pubtruncate bool	如果为真, 为publication中的表复制TRUNCATE操作。
pubviaroot bool	如果为真, 则使用publication中提及的顶端分区祖先的标识和模式复制叶分区上的操作, 而不是复制其自己的(标识和模式)。

## 48.48. pg\_publication\_rel

目录pg\_publication\_rel包含数据库中关系和publication之间的映射。这是一种多对多映射。这些信息对用户更加友好的视图请参考Section 48.86

Table 48.48. pg\_publication\_rel Columns

列类型	描述
oid oid	行标识符
prpubid oid (references pg_publication.oid)	对publication的引用
prrelid oid (references pg_class.oid)	对关系的引用

## 48.49. pg\_range

目录pg\_range存储关于范围类型的信息。它是类型在pg\_type中项的补充。

Table 48.49. pg\_range Columns

列类型	描述
rngtypid oid (references pg_type.oid)	范围类型的OID

列类型	描述
rngsubtype oid (references pg_type.oid)	该范围类型的元素类型（子类型）的OID
rngcollation oid (references pg_collation.oid)	用于范围比较的排序规则的OID，如果没有则为零
rngsubopc oid (references pg_opclass.oid)	用于范围比较的子类型的操作符类的OID
rngcanonical regproc (references pg_proc.oid)	将一个范围值转换为规范形式的函数的OID，如果没有则为零
rngsubdiff regproc (references pg_proc.oid)	以双精度返回两个元素值不同的函数的OID，如果没有则为零

rngsubopc（加上rngcollation，如果元素类型是可排序的）决定了被该范围类型所使用的排序顺序。rngcanonical用于离散类型的元素类型。rngsubdiff是可选的，但是提供它可以提高范围类型上的GiST索引性能。

## 48.50. pg\_replication\_origin

pg\_replication\_origin目录包含所有已创建的复制源。更多复制源的信息请见Chapter 46

和大部分系统目录不同，pg\_replication\_origin在一个集簇的所有数据库之间共享：每个集簇只有一份pg\_replication\_origin拷贝，而不是每个数据库一份。

Table 48.50. pg\_replication\_origin Columns

列类型	描述
roident oid	一个集簇范围内唯一的复制源标识符。应该绝不会脱离系统。
roname text	外部的由用户定义的复制源名称。

## 48.51. pg\_rewrite

目录pg\_rewrite存储对于表和视图的重写规则。

Table 48.51. pg\_rewrite Columns

列类型	描述
oid oid	行标识符
rulename name	规则名称
ev_class oid (references pg_class.oid)	使用该规则的表
ev_type char	使用该规则的事件类型： 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE
ev_enabled char	

列类型	描述
	控制在哪种session_replication_role模式中触发该规则。 O = 规则在“origin”和“local”模式触发， D = 规则被禁用， R = 规则在“replica”模式触发， A = 规则总是被触发。
is_instead bool	为真表示是一个INSTEAD规则
ev_qual pg_node_tree	规则条件的表达式树（按照nodeToString()的表现形式）
ev_action pg_node_tree	规则动作的查询树（按照nodeToString()的表现形式）

**Note**

如果一个表在这个目录中有任何规则， pg\_class.relhasrules必须为真。

## 48.52. pg\_seclabel

目录pg\_seclabel存储数据库对象上的安全标签。安全标签可以通过SECURITY LABEL命令操纵。简单的查看安全标签方法请见Section 48.91

同时请见pg\_shseclabel，它对集簇共享的数据库对象的安全标签执行相似的功能。

Table 48.52. pg\_seclabel Columns

列类型	描述
objoid oid (参考任意OID列)	该安全标签依附的对象的OID
classoid oid (references pg_class.oid)	该对象所出现的系统目录的OID
objsubid int4	对于一个在表列上的安全标签，这将是列号（objoid和classoid指表本身）。对于所有其他对象类型，本列为0。
provider text	与该标签相关的标签提供者。
label text	应用于该对象的安全标签。

## 48.53. pg\_sequence

目录pg\_sequence包含有关序列的信息。  
在pg\_class中。

一些序列的信息（例如名称和方案）放

Table 48.53. pg\_sequence Columns

列类型	描述
seqrelid oid (references pg_class.oid)	这个序列的pg_class项的OID
seqtypid oid (references pg_type.oid)	

列类型	描述
	序列的数据类型
seqstart int8	序列的起始值
seqincrement int8	序列的增量值
seqmax int8	序列的最大值
seqmin int8	序列的最小值
seqcache int8	序列的缓冲尺寸
seqcycle bool	序列是否循环

## 48.54. pg\_shdepend

目录pg\_shdepend记录数据库对象和共享对象之间的依赖关系，例如角色。这些信息使得LightDB可以确保对象在被删除时没有被其他对象引用。

另请参阅pg\_depend，它对单个数据库中对象之间的依赖提供了相似的功能。

与大部分其他系统目录不同，pg\_shdepend在整个集簇的所有数据库之间共享：在每一个集簇中只有一个pg\_shdepend的拷贝，而不是每个数据库一份。

Table 48.54. pg\_shdepend Columns

列类型	描述
dbid oid (references pg_database.oid)	依赖者对象所在的数据库OID，如果是一个共享对象则值为0，或者是一个SHARED_DEPENDENCY_PIN条目
classid oid (references pg_class.oid)	依赖者对象所在的系统目录的OID，或在SHARED_DEPENDENCY_PIN条目时为零
objid oid (参考任意OID列)	依赖者对象的OID，或在SHARED_DEPENDENCY_PIN条目时为零
objsubid int4	对于一个表列，这将是列号（objid和classid指向表本身）。对于所有其他对象类型，该列值为0。
refclassid oid (references pg_class.oid)	被引用对象所在的系统目录的OID（必须是一个共享的目录）
refobjid oid (参考任意OID列)	被引用对象的OID
deptype char	定义该依赖关系的特定语义的代码，参见文本

在所有情况下，一个pg\_shdepend项表明被引用对象不能在没有删除其依赖对象的情况下被删除。但是，其中也有多种依赖类型，由deptype标识：

SHARED\_DEPENDENCY\_OWNER (o)

被引用对象（必须是一个角色）是依赖对象的拥有者。

SHARED\_DEPENDENCY\_ACL (a)

被引用对象（必须是一个角色）在依赖对象的ACL（访问控制列表，即权限列表）中被提到。（不会为对象的拥有者创建一个SHARED\_DEPENDENCY\_ACL项，因为拥有者将会会有一个SHARED\_DEPENDENCY\_OWNER项。）

SHARED\_DEPENDENCY\_POLICY (r)

作为一个依赖策略对象的目标被引用的对象（必须是一个角色）。

SHARED\_DEPENDENCY\_PIN (p)

没有依赖对象，这种类型的项是一个信号，用来指示系统本身依赖于被引用对象，并且因此该对象必须永远不能被删除。这种类型的项只能被initdb创建。这种项中关于依赖对象的列值都为0。

SHARED\_DEPENDENCY\_TABLESPACE (t)

引用对象（必须是表空间），做为不拥有存储空间的关系的表空间被提到。

未来可能会需要其他的依赖类型。特别要注意的是在当前定义中只支持角色和表空间作为被引用对象。

## 48.55. pg\_shdescription

目录pg\_shdescription存储共享数据库对象的可选描述（注释）。描述可以通过COMMENT命令操作，并且可以使用\tsql的\d命令来查看。

另请参阅pg\_description，它对单个数据库中对象之间的依赖提供了相似的功能。

与大部分其他系统目录不同，pg\_shdescription在整个集簇的所有数据库之间共享：在每一个集簇中只有一个pg\_shdescription的拷贝，而不是每个数据库一份。

Table 48.55. pg\_shdescription Columns

列类型	描述
objoid oid (参考任意OID列)	该描述所属的对象的OID
classoid oid (references pg_class.oid)	该对象所在系统目录的OID
description text	作为该对象描述的任意文本

## 48.56. pg\_shseclabel

目录pg\_shseclabel存储共享数据库对象上的安全标签。安全标签可以通过SECURITY LABEL命令操纵。更简单的查看安全标签的方式请见Section 48.91

另请参阅pg\_seclabel，它对单个数据库中对象的安全标签提供了相似的功能。

与大部分其他系统目录不同，pg\_shseclabel在整个集簇的所有数据库之间共享：在每一个集簇中只有一个pg\_shseclabel的拷贝，而不是每个数据库一份。

Table 48.56. pg\_shseclabel Columns

列类型	描述
objoid oid (参考任意OID列)	该安全标签所属对象的OID

列类型	描述
classoid oid (references pg_class.oid)	对象所属系统目录的OID
provider text	与此标签关联的标签提供者
label text	应用到该对象的安全标签。

## 48.57. pg\_statistic

目录pg\_statistic存储有关数据库内容的统计数据。其中的项由ANALYZE创建，查询规划器会使用这些数据来进行查询规划。注意所有的统计数据天然就是近似的，即使它刚刚被更新。

通常对于数据表中一个已经被 ANALYZE 过的列，在本目录中会存在一个stainherit = false的项。如果该列所在的表具有后代（即有其他表继承该表），对于该列还会创建第二个stainherit = true的项。stainherit = true的项表示列在整个继承树上的统计数据，即通过SELECT column FROM table\*看到的数据的统计，而stainherit = false的项表示对SELECT column FROM ONLY table的结果的统计。

pg\_statistic也存储关于索引表达式值的统计数据，就好像它们是真正的数据列，但在这种情况中starelid指索引。对一个普通非表达式索引列不会创建项，因为它将是底层表列的项的冗余。当前，索引表达式的项都具有stainherit = false。

因为不同类型的统计信息适用于不同类型的数据，pg\_statistic 被设计成不太在意自己存储的是什么类型的统计。只有极为常用的统计信息（比如NULL的含量）才在pg\_statistic里给予专用的字段。其它所有东西都存储在“槽位”中，而槽位是一组相关的列，它们的内容用槽位中的一个列里的代码表示。更详细的信息请参阅 src/include/catalog/pg\_statistic.h。

pg\_statistic不应该是公共可读的，因为即使是一个表内容的统计性信息也可能被认为是敏感的（例子：一个薪水列的最大和最小值可能是非常有趣的）。pg\_stats是pg\_statistic上的一个公共可读的视图，它只会显示出当前用户可读的表的信息。

Table 48.57. pg\_statistic Columns

列类型	描述
starelid oid (references pg_class.oid)	被描述列所属的表或索引
staattnum int2 (references pg_attribute.attnum)	被描述列的编号
stainherit bool	如果为真，统计包含了继承后代的列而不仅仅是指定关系的列
stanullfrac float4	列的项为空的比例
stawidth int4	非空项的平均存储宽度，以字节计
stadistinct float4	列中非空唯一值的数目。一个大于零的值是唯一值的真正数目。一个小于零的值是表中行数的乘数的负值；例如，对于一个 80% 的值为非空且每个非空值平均出现两次的列，可以表示为stadistinct = -0.4。一个0值表示唯一值的数目未知。
stakindN int2	一个代码，它表示存储在该pg_statistic行中第N个“槽位”的统计类型。

列类型	描述
staopN oid (references pg_operator.oid)	一个用于生成这些存储在第N个“槽位”的统计信息的操作符。比如，一个柱面图槽位会用<操作符，该操作符定义了该数据的排序顺序。如果统计类型不需要操作符则为零。
stacollN oid (references pg_collation.oid)	排序规则用于导出存储在第N个“槽”中的统计信息。例如，可排序列的直方图槽将显示定义数据排序顺序的排序规则。对于不可整理数据，为零。
stanumbersN float4[]	第N个“槽位”的类型的数值类型统计，如果该槽位不涉及数值类型则为NULL
stavaluesN anyarray	第N个“槽位”的类型的列值，如果该槽位类型不存储任何数据值则为 NULL。每个数组的元素值实际上都是指定列的数据类型或者是一个相关类型（如数组元素类型），因此，除了把这些列的类型定义成anyarray之外别无他法。

## 48.58. pg\_statistic\_ext

目录pg\_statistic\_ext包含了扩展的规划器统计信息的定义。这个目录中的每一行对应于一个用CREATE STATISTICS创建的统计信息对象。

Table 48.58. pg\_statistic\_ext Columns

列类型	描述
oid oid	行标识符
stxrelid oid (references pg_class.oid)	包含这个对象所描述的列的表
stxname name	统计信息对象的名称
stxnamespace oid (references pg_namespace.oid)	包含这个统计信息对象的名字空间的OID
stxowner oid (references pg_authid.oid)	统计信息对象的拥有者
stxstattarget int4	stxstattarget通过ANALYZE 控制这个统计对象积累的统计信息的详细级别。零值表示不应收集任何统计信息。负值表示使用引用列（如果设置）的统计信息目标的最大值或系统默认统计目标。stxstattarget的正值决定要收集的“most common values”的目标数。
stxkeys int2vector (references pg_attribute.attnum)	一个属性编号的数组，表示哪些表列被这个统计信息对象覆盖。例如值1 3表示第一个和第三个表列被覆盖
stxkind char[]	包含被启用统计类型代码的数组，可用的值有：d表示n-distinct统计信息，f表示函数依赖统计信息和m表示最常见值（MCV）列表的统计信息 e 表示表达式的统计信息

pg\_statistic\_ext条目在CREATE STATISTICS期间完全填充，但是随后不计算实际的统计值。后来ANALYZE命令计算所需的值，并在pg\_statistic\_ext\_data目录中填充条目。

## 48.59. pg\_statistic\_ext\_data

目录pg\_statistic\_ext\_data保存在pg\_statistic\_ext中定义的扩展规划器统计信息的数据。该目录的每一行对应用CREATE STATISTICS创建的一个统计信息对象。

就像pg\_statistic, pg\_statistic\_ext\_data不应该被公众读取到, 因为其内容可能被视为敏感内容。(例如: 列中值的最常见组合可能非常令人关注。)

pg\_stats\_ext是pg\_statistic\_ext\_data上的公共可读视图(与pg\_statistic\_ext连接后), 它仅暴露有关当前用户可读的表和列的信息。

Table 48.59. pg\_statistic\_ext\_data Columns

列类型	描述
stxoid oid (references pg_statistic_ext.oid)	包含此数据定义的扩展统计信息对象
stxnddistinct pg_ndistinct	N-distinct计数, 序列化为pg_ndistinct类型
stxddependencies pg_dependencies	函数依赖统计信息, 序列化为pg_dependencies类型
stxdmcv pg_mcv_list	MCV (最频值) 列表统计信息, 序列化为pg_mcv_list类型

## 48.60. pg\_subscription

目录pg\_subscription包含所有现有的逻辑复制订阅。更多有关逻辑复制的信息请见Chapter 28

和大部分系统目录不同, pg\_subscription在集簇的所有数据库之间共享: 每个集簇只有一份pg\_subscription拷贝, 而不是每个数据库一份。

对列subconninfo的访问被从普通用户那里收回, 因为该列可能含有明文口令。

Table 48.60. pg\_subscription Columns

列类型	描述
oid oid	行标识符
subdbid oid (references pg_database.oid)	订阅所在的数据库的OID
subname name	订阅的名称
subowner oid (references pg_authid.oid)	订阅的拥有者
subenabled bool	如果为真, 订阅被启用并且应该被复制。
subbinary bool	如果为真, 订阅将请求发布者以二进制格式发送数据。
subconninfo text	到上游数据库的连接字符串
subslotname name	上游数据库中复制槽的名称(也用于本地复制源名称);空表示 NONE
subsynchronous_commit text	用于订阅工作者的synchronous_commit设置。
subpublications text[]	



列类型	描述
	被订阅的publication名称的数组。这些引用上游数据库中定义的publication。更多有关publication的内容请见Section 28.1

## 48.61. pg\_subscription\_rel

目录pg\_subscription\_rel包含每个订阅中每个被复制关系的状态。这是一种多对多映射。

这个目录仅包含运行CREATE SUBSCRIPTION 或 ALTER SUBSCRIPTION ... REFRESH PUBLICATION. 以后对订阅已知的表。

Table 48.61.pg\_subscription\_rel Columns

列类型	描述
srsubid oid (references pg_subscription.oid)	对订阅的引用
srrelid oid (references pg_class.oid)	对关系的引用
srsubstate char	状态代码: i = 初始化, d = 数据正在被拷贝, f = 结束表拷贝, s = 已同步, r = 准备好 (普通复制)
srsublsn pg_lsn	在s或r状态中, 用于同步协调的状态更改的远程 LSN, 否则为空

## 48.62. pg\_tablespace

目录pg\_tablespace存储关于可用表空间的信息。表可以被放置在特定表空间中以实现磁盘布局的管理。

与大部分其他系统目录不同, pg\_tablespace在整个集簇的所有数据库之间共享: 在每一个集簇中只有一个pg\_tablespace的拷贝, 而不是每个数据库一份。

Table 48.62.pg\_tablespace Columns

列类型	描述
oid oid	行标识符
spcname name	表空间名
spcowner oid (references pg_authid.oid)	表空间的拥有者, 通常是创建它的用户
spcacl aclitem[]	访问权限, 详见Section 5.7
spcoptions text[]	表空间级别的选项, 形如“keyword=value”的字符串

## 48.63. pg\_transform

目录pg\_transform存储有关转换的信息, 转换是一种让数据类型适应过程语言的机制。详见CREATE TRANSFORM。

Table 48.63. pg\_transform Columns

列类型	描述
oid oid	行标识符
trftype oid (参考 pg_type.oid)	这个转换所针对的数据类型的 OID
trflang oid (references pg_language.oid)	这个转换所针对的语言的 OID
trffromsql regproc (references pg_proc.oid)	一个函数的 OID, 该函数用来将数据类型转换为过程语言的输入 (例如函数参数)。如果要使用缺省行为, 这里存储零。
trftosql regproc (references pg_proc.oid)	一个函数的 OID, 该函数被用来转换过程语言的输出 (例如返回值) 为该数据类型。如果要使用缺省行为, 这里存储零。

## 48.64. pg\_trigger

目录pg\_trigger存储表和视图上的触发器。详见CREATE TRIGGER。

Table 48.64. pg\_trigger Columns

列类型	描述
oid oid	行标识符
tgrelid oid (references pg_class.oid)	触发器所在的表
tgparentid oid (参考 pg_trigger.oid)	此被克隆的触发器的父触发器 (这会在创建分区或附加到分区表时发生), 如果不是克隆则为0;。
tgname name	触发器名 (在同一个表的触发器中必须唯一)
tgfoid oid (references pg_proc.oid)	要被触发器调用的函数
tgtype int2	标识触发器触发条件的位掩码
tgenabled char	控制触发器在session_replication_role模式中的触发。 0 = 触发器在“origin”和“local”模式触发, D = 触发器被禁用, R = 触发器在“replica”模式触发, A = 触发器总是触发。
tgisinternal bool	为真表示触发器是内部生成的 (通常是为了强制由tgconstraint指定的约束)
tgconstrrelid oid (references pg_class.oid)	被一个引用完整性约束引用的表 (如果触发器不是针对一个引用完整性约束则为零)
tgconstrindid oid (references pg_class.oid)	支持一个唯一、主键、引用完整性约束或者排除约束的索引 (如果触发器不是针对这些约束类型中的一个则为零)
tgconstraint oid (参考 pg_constraint.oid)	与触发器相关的pg_constraint项 (如果触发器不用于约束则为零)

列类型	描述
tgdeferrable bool	如果约束触发器可推迟则为真
tginitdeferred bool	如果约束触发器初始可推迟则为真
tgnargs int2	传递给触发器函数的参数字符串个数
tgattr int2vector (references pg_attribute.attnum)	如果触发器是列限定的，这里存放列号；否则这是一个空数组
tgargs bytea	传递给触发器的参数字符串，每一个都以NULL结尾
tgqual pg_node_tree	触发器WHEN条件的表达式树（以nodeToString()的表现形式），如果没有则为空
tgoldtable name	OLD TABLE的REFERENCING子句名称，如果没有则为空
tgnewtable name	NEW TABLE的REFERENCING子句名称，如果没有则为空

当前，列限定触发器只被UPDATE事件支持，因此tgattr只用于这种事件类型。tgtype页可以包含用于其他事件类型的位，但其他事件类型是表范围的触发器且会忽略tgattr。

Note

当tgconstraint非零时，tgconstrrelid、tgconstrindid、tgdeferrable和tginitdeferred与被引用的pg\_constraint项有很大的冗余。但是，存在将一个不可延迟触发器关联到一个可延迟约束的可能性：外键约束可以有一些可延迟和一些不可延迟触发器。

Note

如果一个关系在本目录中拥有任何触发器，其pg\_class.relhastriggers必须为真。

## 48.65. pg\_ts\_config

pg\_ts\_config目录包含表示文本搜索配置的选项。一个配置指定了一个特定的文本搜索分析器和一个用于分析器输出记号的字典列表。分析器由pg\_ts\_config项展现，而记号到字典的映射则由pg\_ts\_config\_map中的辅助项定义。

LightDB的文本搜索特性在Chapter 12中有更详尽的描述。

Table 48.65. pg\_ts\_config Columns

列类型	描述
oid oid	行标识符
cfgname name	文本搜索配置名

列类型	描述
cfgnamespace oid (references pg_namespace.oid)	包含该配置的名字空间的OID
cfgowner oid (references pg_authid.oid)	配置的拥有者
cfgparser oid (references pg_ts_parser.oid)	该配置的文本搜索分析器的OID

## 48.66. pg\_ts\_config\_map

pg\_ts\_config\_map目录包含的项展示了对于每一个文本搜索配置的每一种输出记号类型，有哪些文本搜索字典可供查询以及以何种顺序。

LightDB的文本搜索特性在Chapter 1中有更详尽的描述。

Table 48.66. pg\_ts\_config\_map Columns

列类型	描述
mapcfg oid (references pg_ts_config.oid)	拥有该映射项的pg_ts_config项的OID
maptokentype int4	一种由配置的分析器送出的记号类型
mapseqno int4	查询该项的顺序 (mapseqno值小的优先)
mapdict oid (参考 pg_ts_dict.oid)	要查询的文本搜索字典的OID

## 48.67. pg\_ts\_dict

pg\_ts\_dict目录包含定义文本搜索字典的项。一个字典依赖于一个文本搜索模板，它指定了所有需要的函数实现，字典本身则为模板支持的用户可设置参数提供值。这种分工允许无权限的用户创建字典。参数由一个文本串dictinitoption定义，其格式和意义随着模板而变化。

LightDB的文本搜索特性在Chapter 1中有更详尽的描述。

Table 48.67. pg\_ts\_dict Columns

列类型	描述
oid oid	行标识符
dictname name	文本搜索字典名
dictnamespace oid (references pg_namespace.oid)	包含该字典的名字空间OID
dictowner oid (references pg_authid.oid)	字典的拥有者
dicttemplate oid (references pg_ts_template.oid)	该字典的文本搜索模板的OID
dictinitoption text	

列类型	描述
	模板的初始化选项字符串

## 48.68. pg\_ts\_parser

pg\_ts\_parser目录包含定义文本搜索分析器的项。一个分析器负责将输入文本分割成词位并为每一个词位分配一个记号类型。由于一个分析器必须用C语言级别的函数实现，创建新分析器的工作只限于数据库的超级用户。

LightDB的文本搜索特性在Chapter 12中有更详尽的描述。

Table 48.68. pg\_ts\_parser Columns

列类型	描述
oid oid	行标识符
prsname name	文本搜索分析器的名字
prsnamespace oid (references pg_namespace.oid)	包含此分析器的名字空间的OID
prsstart regproc (references pg_proc.oid)	分析器启动函数的OID
prstoken regproc (references pg_proc.oid)	分析器的下一记号函数的OID
prsend regproc (references pg_proc.oid)	分析器的关闭函数的OID
prsheadline regproc (references pg_proc.oid)	分析器的大标题函数的OID（没有则为零）
prslextype regproc (references pg_proc.oid)	分析器的词汇类型函数的OID

## 48.69. pg\_ts\_template

pg\_ts\_template目录包含定义文本搜索模板的项。一个模板是一类文本搜索字典的实现骨架。由于一个模板必须用C语言级别的函数实现，新模板的创建只限于数据库超级用户。

LightDB的文本搜索特性在Chapter 12中有更详尽的描述。

Table 48.69. pg\_ts\_template Columns

列类型	描述
oid oid	行标识符
tplname name	文本搜索模板的名字
tplnamespace oid (references pg_namespace.oid)	包含此模板的名字空间的OID
tplinit regproc (references pg_proc.oid)	模板的初始化函数的OID（没有则为零）
tpllexize regproc (references pg_proc.oid)	

列类型	描述
	模板的词汇化函数的OID

## 48.70. pg\_type

目录pg\_type存储有关数据类型的信息。基类和枚举类型（标度类型）使用CREATE TYPE创建，而域使用CREATE DOMAIN创建。数据库中的每一个表都会有一个自动创建的组类型，用于表示表的行结构。也可以使用CREATE TYPE AS创建组类型。

Table 48.70. pg\_type Columns

列类型	描述
oid oid	行标识符
typename name	数据类型的名子
typnamespace oid (references pg_namespace.oid)	包含此类型的名字空间的OID
typowner oid (references pg_authid.oid)	类型的拥有者
typlen int2	对于一个固定尺寸的类型，typlen是该类型内部表示的字节数。对于一个变长类型，typlen为负值。-1表示一个“varlena”类型（具有长度字），-2表示一个以空值结尾的C字符串。
typbyval bool	typbyval判断内部例程传递这个类型的数值时是通过传值还是传引用。如果typlen不是1、2或4（或者在Datum为8字节的机器上为8），因此typbyval最好是假。变长类型总是传引用。注意即使长度允许传值，typbyval也可以为假。
typtype char	typtype可以是：b表示一个基类，c表示一个组类型（例如一个表的行类型），d表示一个域，e表示一个枚举类型，p表示一个伪类型，r表示一个范围类型，或m表示一个多范围类型。另请参阅typrelid和typbasetype。
typcategory char	typcategory是一种任意的数据类型分类，它被分析器用来决定哪种隐式转换“更好”。参见Table 48.71
typispreferred bool	如果此类型在它的typcategory中是一个更好的转换目标，此列为真
typisdefined bool	如果此类型已被定义则为真，如果此类型只是一个表示还未定义类型的占位符则为假。当typisdefined为假，除了类型名字、名字空间和OID之外什么都不能被依赖。
typdelim char	在分析数组输入时，分隔两个此类型值的字符。注意该分隔符是与数组元素数据类型相关联的，而不是和数组的数据类型关联。
typrelid oid (references pg_class.oid)	如果这是一个复合类型（见typtype），那么这个列指向pg_class中定义对应表的项（对于自由存在的复合类型，pg_class项并不表示一个表，但不管怎样该类型的pg_attribute项需要链接到它）。对非复合类型此列为零。
typelem oid (参考 pg_type.oid)	

列类型	描述
	如果typelem不为零，则它标识pg_type里面的另外一行。如果typsubscript为零则该项为零。但是，当typsubscript不为零时它也可以为零，如果处理机不需要typelem来决定下标结果类型。注意typelem依赖被认为是表示该类型中的元素类型的物理包含；所以这个元素类型上的DDL变更可能会受到该类型存在的限制。
typarray oid (参考 pg_type.oid)	如果typarray不是零，则它标识pg_type中的另一行，这一行是一个将此类型作为元素的“真的”数组类型
typinput regproc (references pg_proc.oid)	输入转换函数（文本格式）
typoutput regproc (references pg_proc.oid)	输出转换函数（文本格式）
typreceive regproc (references pg_proc.oid)	输入转换函数（二进制格式），如果没有则为零
typsend regproc (参考 pg_proc.oid)	输出转换函数（二进制格式），如果没有则为零
typmodin regproc (参考 pg_proc.oid)	类型修改器输入函数，如果类型没有提供修改器则为零
typmodout regproc (参考 pg_proc.oid)	类型修改器输出函数，或为零以使用标准格式
typanalyze regproc (参考 pg_proc.oid)	自定义ANALYZE函数，零表示使用标准函数
typalign char	<p>typalign是当存储此类型值时要求的对齐性质。它应用于磁盘存储以及该值在LightDB内部的大多数表现形式。如果数值是连续存放的，比如在磁盘上的一个完整行，在这种类型的数据前会插入填充，这样它就可以按照指定边界存储。对齐引用是该序列中第一个数据的开头。对齐引用是序列中第一个数据的开始。可能的值为：</p> <ul style="list-style-type: none"> <li>• c = char对齐，即不需要对齐。</li> <li>• s = short对齐（在大部分机器上为2字节）。</li> <li>• i = int对齐（在大部分机器上为4字节）。</li> <li>• d = double对齐（在很多机器上为8字节，但绝不是全部）。</li> </ul>
typstorage char	<p>如果一个变长类型（typlen = -1）可被TOAST，typstorage说明这种类型的列应采取的默认策略。可能的值是：</p> <ul style="list-style-type: none"> <li>• p（普通的）：值必须始终以普通方式存储（非varlena类型总是使用这个值）。</li> <li>• e（外部的）：值可以存储在一个次要“TOAST”关系中（如果有一个关系，参见pg_class.reltoastrelid）。</li> <li>• m（主要的）：值可以被压缩并内联存储。</li> <li>• x（扩展的）：值可以被压缩和/或移动到次要关系。</li> </ul> <p>x toast-able 类型的常用选项。注意m值也可以被移动到二级存储，但只能是作为最后一种方案（e和x值会先被移动）。</p>
typnotnull bool	typnotnull表示类型上的一个非空约束。只用于域。

列类型	描述
typbasetype oid (references pg_type.oid)	如果这是一个域（见typtype），则typbasetype标识这个域基于的类。如果此类不是一个域则为0。
typtypmod int4	域使用typtypmod来记录被应用于它们基类型的typmod（如果基类型不使用typmod，则为-1）。如果此类型不是一个域则为-1。
typndims int4	对于一个数组上的域，typndims是数组维度数（即，typbasetype是一个数组类型）。除数组类型上的域之外的类型的此列为0。
typcollation oid (references pg_collation.oid)	typcollation指定此类型的排序规则。如果类型不支持排序规则，此列为0。一个支持排序规则的基类型此处会有一个非零值，典型值为DEFAULT_COLLATION_OID。可排序类型上的域可以有一个不同于其基类型的排序规则OID，如果为该域指定了一个排序规则OID的话。
typdefaultbin pg_node_tree	如果typdefaultbin为非空，那么它是 该类型默认表达式的nodeToString()表现形式。这个列只用于域。
typdefault text	如果某类型没有相关默认值，那么typdefault为空。如果typdefaultbin不为空，那么typdefault必须包含一个typdefaultbin所指的默认表达式的人类可读的版本。如果typdefaultbin为空但typdefault不为空，则typdefault是该类型默认值的外部表现形式，它可以被交给该类型的输入转换器来产生一个常量。
typacl aclitem[]	访问权限，另请参阅Section 5.7

Note

对于系统表中使用的固定宽度类型，关键的是在pg\_type定义的大小和对齐与编译器在表示表行的结构中布局列的方式一致。

Table 48.7列出了typcategory的系统定义值。任何未来对此列表的增加都将是大写ASCII字母。所有其他ASCII字符都保留给用户定义类别。

Table 48.71. typcategory Codes

编码	类别
A	数组类型
B	布尔类型
C	组合类型
D	日期/时间类型
E	枚举类型
G	几何类型
I	网络地址类型
N	数字类型
P	伪类型
R	范围类型
S	字符串类型



编码	类别
T	时间间隔类型
U	用户定义类型
V	位字符串类型
X	未知 type

## 48.71. pg\_user\_mapping

目录pg\_user\_mapping存储从本地用户到远程的映射。对这个目录的访问对普通用户有限制，可使用视图pg\_user\_mappings替代。

Table 48.72. pg\_user\_mapping Columns

列类型	描述
oid oid	行标识符
umuser oid (参考 pg_authid.oid)	将要被映射的本地角色的OID，如果用户映射是公共的则为零
umserver oid (references pg_foreign_server.oid)	包含此映射的外部服务器的OID
umoptions text[]	用户映射特定的选项，以“keyword=value”字符串形式

## 48.72. 系统视图

除系统目录外，LightDB提供了一些内建视图。一些系统视图为系统目录上一些常用查询提供了便利的访问。其他视图提供了对内部服务器状态的访问。

信息模式 (Chapter 35 提供了一组可供选择的视图，它和系统视图在功能上有所重叠。由于信息模式是SQL标准，而这里描述的视图是LightDB特有的，如果信息模式能提供你所需要的信息，通常最好使用它。

Table 48.7列出了这里描述的系统视图。每一个视图的详细文档都在后文中。还有一些附加视图用于提供对于统计收集器结果的访问，它们在Table 25.2中描述。

除了特别注明的，所有这里描述的视图都是只读的。

Table 48.73. 系统视图

视图名字	用途
pg_available_extensions	可用的扩展
pg_available_extension_versions	所有版本的扩展
pg_config	编译时配置参数
pg_cursors	打开的游标
pg_file_settings	配置文件内容摘要
pg_group	数据库用户组
pg_hba_file_rules	客户端认证配置文件内容的摘要
pg_indexes	索引
pg_locks	当前保持或者等待的锁
pg_matviews	物化视图

视图名字	用途
pg_policies	策略
pg_prepared_statements	预备好的语句
pg_prepared_xacts	预备好的事务
pg_publication_tables	publication和它们相关的表
pg_replication_origin_status	有关复制源的信息，包括复制进度
pg_replication_slots	复制槽信息
pg_roles	数据库角色
pg_rules	规则
pg_seclabels	安全标签
pg_sequences	序列
pg_settings	参数设置
pg_shadow	数据库用户
pg_shmem_allocations	共享内存分配
pg_stats	规划器统计信息
pg_stats_ext	扩展的计划器统计信息
pg_tables	表
pg_timezone_abbrevs	时区简写
pg_timezone_names	时区名字
pg_user	数据库用户
pg_user_mappings	用户映射
pg_views	视图

## 48.73. pg\_available\_extensions

pg\_available\_extensions视图列出了可用于安装的扩展。参见pg\_extension目录，它显示当前已安装的扩展。

Table 48.74. pg\_available\_extensions Columns

列类型	描述
name name	扩展名
default_version text	默认版本的名字，如果没有指定则为NULL
installed_version text	当前已安装的扩展版本，如果没有安装则为NULL
comment text	来自于扩展的控制文件的注释字符串

pg\_available\_extensions视图是只读的。

## 48.74. pg\_available\_extension\_versions

pg\_available\_extension\_versions视图列出了可用于安装的指定扩展版本。参见pg\_extension目录，它显示当前已安装的扩展。

Table 48.75. pg\_available\_extension\_versions Columns

列类型	描述
name name	扩展名
version text	版本名
installed bool	如果此版本的扩展当前已安装则为真
superuser bool	如果只有超级用户被允许安装此扩展则为真(更多请参见 trusted)
trusted bool	如果扩展可以由具有适当权限的非超级用户安装, 则为真
relocatable bool	如果扩展能被重定位到另一个模式则为真
schema name	此扩展必须被安装到的模式名, 如果此扩展是部分或者全部可以重定位的, 此列为NULL
requires name[]	先决条件扩展的名字, 如果没有则为NULL
comment text	来自于扩展的控制文件的注释字符串

pg\_available\_extension\_versions视图是只读的。

## 48.75. pg\_config

视图pg\_config描述了当前安装的LightDB版本的编译时配置参数。例如, 它旨在供希望与LightDB接口的软件包使用, 以便于查找所需的头文件和库。它提供与lt\_configLightDB客户端应用程序相同的基本信息。

默认情况下, pg\_config视图只能由超级用户读取。

Table 48.76. pg\_config Columns

列类型	描述
name text	内存上下文的名称
setting text	参数值

## 48.76. pg\_cursors

pg\_cursors视图列出了当前可用的游标。游标可以以几种方式定义:

- 通过SQL中的DECLARE语句
- 通过前端/后端协议中的绑定消息, 如Section 49.2.8中所描述的
- 通过服务器编程接口 (SPI), 如Section 43.中所描述的

pg\_cursors视图显示由任何这些方式创建的游标。视图只存在于定义它们的事务期间，除非声明了WITH HOLD。因此非保持游标只在它们的创建事务结束前存在于这个视图中。

Note

视图用于在内部实现LightDB的某些部件，例如过程语言。因此，pg\_cursors视图可能包括那些不是由用户显式创建的游标。

Table 48.77. pg\_cursors Columns

列类型	描述
name text	游标名
statement text	提交用于定义此游标的查询语句
is_holdable bool	如果游标是可保持的（即，它可以在其定义事务提交后被访问）则为true，否则为false
is_binary bool	如果游标被声明为BINARY则为true，否则为false otherwise
is_scrollable bool	如果游标是可滚动的（即，允许以一种非顺序的方式检索行）则为true，否则为false
creation_time timestampz	游标被声明的时间

pg\_cursors视图是只读的。

## 48.77. pg\_file\_settings

视图pg\_file\_settings提供了服务器配置文件 内容的概要。这个视图中的每一行表示配置文件中出现的一个 “name = value” 项，还带有注解指示该值是否被成功地应用。在 配置文件有问题时，有可能出现额外的行，它们没有相关的 “name = value” 项，一个例子是配置文件中语法错误。

这个视图有助于检查在配置文件中打算做的修改是否能工作，或者用来诊断 之前的失败。注意这个视图报告的是配置文件的当前内容， 而不是服务器最后应用的值（这些值通常查看 pg\_settings 视图就够了）。

默认情况下，pg\_file\_settings视图只有超级用户可读。

Table 48.78. pg\_file\_settings Columns

列类型	描述
sourcefile text	配置文件的完整路径名
sourceline int4	该项在配置文件中出现的行号
seqno int4	项被处理的顺序 (1..n)

列类型	描述
name text	配置参数名
setting text	被赋予给参数的值
applied bool	为真表示值已被成功应用
error text	如果非空，表示一个错误消息，它说明为什么这个项不能被应用

如果配置文件包含语法错误或者非法参数名，服务器将不会尝试从其中应用任何设置，并且因此所有的applied域都为假。在这种情况下，将会有有一个或者多个行的error域为非空，它们说明了为什么出问题。否则，将尽可能应用每个设置。如果一个设置不能被应用（例如非法值或者该设置不能在服务器开始后改变），会有一个合适的消息存储在它的error域中。一个项的applied域为假的另一种情况是它被后面一个具有相同参数名的项所覆盖，这种情况不会被认为是一种错误，因此在error域中不会有错误消息。

关于更改运行时参数的各种方法请见Section 17.1

## 48.78. pg\_group

视图pg\_group为向后兼容而存在：它模拟了存在于LightDB 8.1之前版本中的一个目录。它显式所有角色的名称和未被标记rolcanlogin的成员，它是被用做组的角色集合的近似。

Table 48.79. pg\_group Columns

列类型	描述
groname name (references pg_authid.rolname)	组的名称
grosysid oid (references pg_authid.oid)	组ID
grolist oid[] (references pg_authid.oid)	包含此组中角色ID的一个数组

## 48.79. pg\_hba\_file\_rules

视图pg\_hba\_file\_rules提供客户端认证配置文件lt\_hba.conf内容的摘要。该文件中每个非空、非注释行都会在这个视图中出现一行，行中还有标记表示该规则是否被成功地应用。

这个视图可用来检查认证配置文件中按计划的更改是否起作用，或者诊断之前的失败。注意这个视图报告的是该文件的当前内容，而不是服务器最后一次载入的内容。

默认情况下，只有超级用户可以读取pg\_hba\_file\_rules视图。

Table 48.80. pg\_hba\_file\_rules Columns

列类型	描述
line_number int4	这条规则在lt_hba.conf中的行号
type text	

列类型	描述
	连接类型
database text[]	这条规则应用的数据库名列表
user_name text[]	这条规则应用的用户及组名列表
address text	主机名或IP地址，或者all、samehost、samenet之一，对于本地连接为空
netmask text	IP地址掩码，如果不适用则为空
auth_method text	认证方法
options text[]	为认证方法指定的选项（如果有）
error text	如果非空，则是一个错误消息，它表示为什么这一行无法被处理

通常，反映一个不正确项的行只有line\_number和error域中有值。

更多有关客户端认证配置的信息请参考Chapter 18

## 48.80. pg\_indexes

视图pg\_indexes提供对于数据库中每一个索引信息的访问。

Table 48.81. pg\_indexes Columns

列类型	描述
schemaname name (参考 pg_namespace.nspname)	包含表和索引的模式名
tablename name (参考 pg_class.relname)	此索引的基表的名字
indexname name (参考 pg_class.relname)	索引名
tablespace name (参考 pg_tablespace.spcname)	包含索引的表空间名（如果是数据库的默认值则为空）
indexdef text	索引定义（CREATE INDEX命令的重构） command

## 48.81. pg\_locks

视图pg\_locks提供了数据库服务器上活动进程中保持的锁的信息。更多锁的讨论参见Chapter 13

pg\_locks中对每一个活动可锁对象、请求锁模式和相关进程的组合都有一行。因此，如果多个进程持有或者正在等待一个可锁对象上的锁，同一个可锁对象可能出现很多次。但是，一个当前没有被锁的对象根本不会出现。

有多种不同类型的可锁对象：整个关系（如表）、关系的单个页、关系的单个元组、事务ID（包括虚拟和永久ID）和普通数据库对象（由类OID和对象OID标识，和

pg\_description或pg\_depend中的相同方式)。扩展一个关系的权力也被表示为一个独立的可锁对象，就像更新pg\_database.datfrozenxid的权力。“advisory”锁可以具有用户定义的意义。

Table 48.82. pg\_locks Columns

列类型	描述
locktype text	可锁对象的类型: relation, extend, frozenid, page, tuple, transactionid, virtualxid, spectoken, object, userlock, or advisory. (参见 Table 25.11)
database oid (参考 pg_database.oid)	锁目标存在的数据库的OID, 如果目标是一个共享对象则为0, 如果目标是一个事务ID则为空
relation oid (参考 pg_class.oid)	作为锁目标的关系的OID, 如果目标不是一个关系或者只是关系的一部分则此列为空
page int4	作为锁目标的页在关系中的页号, 如果目标不是一个关系页或元组则此列为空
tuple int2	作为锁目标的元组在页中的元组号, 如果目标不是一个元组则此列为空
virtualxid text	作为锁目标的事务虚拟ID, 如果目标不是一个虚拟事务ID则此列为空
transactionid xid	作为锁目标的事务ID, 如果目标不是一个事务ID则此列为空ID
classid oid (参考 pg_class.oid)	包含锁目标的系统目录的OID, 如果目标不是一个普通数据库对象则此列为空
objid oid (参考 any OID column)	锁目标在它的系统目录中的OID, 如果目标不是一个普通数据库对象则为空
objsubid int2	锁的目标列号 (classid和objid指表本身), 如果目标是某种其他普通数据库对象则此列为0, 如果目标不是一个普通数据库对象则此列为空
virtualtransaction text	保持这个锁或者正在等待这个锁的事物的虚拟ID
pid int4	保持这个锁或者正在等待这个锁的服务器进程的PID, 如果此锁被一个预备事务所持有则此列为空
mode text	此进程已持有或者希望持有的锁模式的名称 (参见Section 13.3.和Section 13.2.3)
granted bool	如果锁已授予则为真, 如果锁被等待则为假
fastpath bool	如果锁通过快速路径获得则为真, 通过主锁表获得则为假

一个行的granted为真表示一个被指定进程持有的锁。为假表示该进程当前正在等待获取这个锁，这意味着至少一个其他进程正持有或等待同一个可锁对象上的一个冲突锁。该等待进程将一直休眠直到其他锁被释放（或者一个死锁状态被检测到）。单个进程在同一时间只能等待最多一个锁。

贯穿一个事务的运行，一个服务器进程在其生存周期内都持有一个在其虚拟事务ID上的排他锁。如果一个永久ID被分配给事务（通常发生在事务改变数据库状态时），它也会持有一个

在其永久事务ID上的排他锁直到它结束。当一个事务发现它需要等待另一个事务，它也会尝试获取其他事务ID上的共享锁（不管是虚拟还是永久ID，视情况而定）。这只有当其他进程终止并释放其锁后才会成功。

尽管元组是一种可锁对象，关于行级锁的信息被存储在磁盘而不是内存中，因此行级锁通常不在这个视图出现。如果一个进程正在等待一个行级锁，它通常在这个视图出现，并且表示形式为正在等待已持有该行级锁的永久事务ID上的锁。

咨询锁可以在由一个单一bigint值或两个整形值构成的键上获取。一个bigint键被显示为其高位部分在classid列中，低位部分在objid列中，并且objsubid等于1。原来的bigint值可以使用表达式(classid::bigint << 32) | objid::bigint重组。整形键被显示为第一个键在classid列中，第二个键在objid列中，并且objsubid等于2。键的实际意义由用户决定。咨询锁是每一个数据库的本地锁，所以database列对于一个咨询锁没有意义。

pg\_locks提供了一个对于整个数据集簇中所有锁的全局视图，而不仅仅是与当前数据库相关的锁。尽管它的relation列可以被连接到pg\_class.oid来标识被锁关系，但这种方法只有在关系属于当前数据库（database列是当前数据库OID或者0的锁对应的关系）的情况下才会得到正确的结果。

pid列可以被连接到 pg\_stat\_activity视图的pid列来得到持有或等待持有每一个锁的会话的信息。例如

```
SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
ON pl.pid = psa.pid;
```

另外，如果正在使用预备事务，virtualtransaction列可以被连接到pg\_prepared\_xacts视图的transaction列来得到持有该锁的预备事务的信息（一个预备事务不可能正在等待一个锁，但它在运行中会一直持有已获得的锁）。例如：

```
SELECT * FROM pg_locks pl LEFT JOIN pg_prepared_xacts ppx
ON pl.virtualtransaction = '-1/' || ppx.transaction;
```

虽然通过自连接pg\_locks可以获得哪些进程阻塞了其他哪些进程的信息，但是很难得到其中的细节。这样一个查询隐藏了关于哪些锁模式与其他哪些锁模式冲突的知识。更糟糕的是，pg\_locks视图无法给出所等待队列中进程的等待顺序，也无法显示哪些进程是代表其他客户端会话运行的并行工作者。更好的方法是使用pg\_blocking\_pids()函数（见Table 9.64 来标识一个等待进程是被哪些进程阻塞的。

pg\_locks视图显示来自于普通锁管理器和谓词锁管理器的数据，它们是独立的系统。此外，普通锁管理器把它的锁分为普通锁和快速路径锁。这些数据并不被保证是完全一致的。当视图被查询时，快速路径锁上的数据（fastpath = true）会被一次性从每一个后端收集起来，且并不冻结整个锁管理器的状态。因此有可能某些锁在上述信息被收集的过程中被获得或者释放。注意，不管怎样这些锁是已知不会和任何当前正在发生的锁冲突。在所有后端已经查询了快速路径锁后，普通锁管理器的剩余部分被作为一个单元锁住，并且所有剩余锁的一个一致快照被作为一个原子动作收集。在解锁普通锁管理器后，谓词锁管理器也被类似地锁住并且所有谓词锁被作为一个原子动作收集。因此，在快速路径锁这种特殊情况下，每一个锁管理器会传递一个一致的结果组。但由于我们并不会同时锁上两个锁管理器，在我们询问完普通锁管理器后或者询问谓词锁管理器之前，锁可以被获得或者释放。

如果对此视图频繁地访问，对普通或者谓词锁管理器加锁可能会对数据库性能产生一定影响。虽然这些锁只会在最少的时间被保持（足以从锁管理器获得数据），但这无法完全消除可能产生的性能影响。

## 48.82. pg\_matviews

视图pg\_matviews提供了关于数据库中每一个物化视图的信息。



Table 48.83. pg\_matviews Columns

列类型	描述
schemaname name (参考 pg_namespace.nspname)	包含物化视图的模式的名字
matviewname name (references pg_class.relname)	物化视图的名字
matviewowner name (references pg_authid.rolname)	物化视图拥有者的名字
tablespace name (参考 pg_tablespace.spcname)	包含物化视图的表空间名 (如使用数据库默认表空间则为空)
hasindexes bool	如果物化视图有 (或者最近有过) 任何索引, 则此列为真
ispopulated bool	如果物化视图当前已被填充, 则此列为真
definition text	物化视图的定义 (一个重构的SELECT查询)

## 48.83. pg\_policies

视图pg\_policies提供了有关数据库中行级 安全性策略的信息。

Table 48.84. pg\_policies Columns

列类型	描述
schemaname name (参考 pg_namespace.nspname)	包含策略所在表的模式的名称
tablename name (参考 pg_class.relname)	策略所在表的名称
policyname name (参考 pg_policy.polname)	策略名称
permissive text	策略是宽容性的还是限制性的?
roles name[]	这个策略适用的角色
cmd text	这个策略适用的命令类型
qual text	作为这个策略适用的查询的安全屏障条件增加的表达式
with_check text	作为尝试向该表增加行的查询的 WITH CHECK 条件增加的表达式

## 48.84. pg\_prepared\_statements

pg\_prepared\_statements视图显示在当前会话中可用的所有预备语句。关于预备语句详见PREPARE。

pg\_prepared\_statements为每一个预备语句包含一行。 当一个新的预备语句被创建时在此视图中会增加一行, 反之当一个预备语句被释放时在此视图中会删除一行 (例如, 通过DEALLOCATE命令)。

Table 48.85. pg\_prepared\_statements Columns

列类型	描述
name text	预备语句的标识符
statement text	客户端提交用于创建此预备语句的查询语句。对于通过SQL创建的预备语句，这里是由客户端提交的PREPARE语句。对于通过前端/后端协议创建的预备语句，这里是预备语句本身的文本。
prepare_time timestamptz	预备语句被创建的时间
parameter_types regtype[]	预备语句期望的参数类型，以一个regtype数组的形式。这个数组中一个元素所对应的OID可通过将regtype值转换为oid获得。
from_sql bool	如果预备语句通过SQL命令PREPARE创建，则为true；如果预备语句通过前端/后端协议创建，则为false

pg\_prepared\_statements视图为只读。

## 48.85. pg\_prepared\_xacts

视图pg\_prepared\_xacts显示关于两阶段提交（详见PREPARE TRANSACTION）的当前准备好事务的信息。

pg\_prepared\_xacts为每一个预备事务包含一行。当事务被提交或回滚时，相应的项将被移除。

Table 48.86. pg\_prepared\_xacts Columns

列类型	描述
transaction xid	预备事务的数字事务标识符
gid text	分配给事务的全局标识符
prepared timestamptz	此事务为提交准备好的时间
owner name (references pg_authid.rolname)	执行此事务的用户名
database name (参考 pg_database.datname)	执行此事务所在数据库的名字

当pg\_prepared\_xacts视图被访问时，内部事务管理器数据结构被暂时地锁住，并为视图的显示产生一个副本。这确保了视图中是一组一致的结果，并且不会阻塞普通操作。不管怎样，当此视图被频繁访问时，会对数据库性能有所影响。

## 48.86. pg\_publication\_tables

视图pg\_publication\_tables提供publication与其所包含的表之间的映射信息。和底层的目录pg\_publication\_rel不同，这个视图展开了定义为FOR ALL TABLES的publication，这样对这类publication来说，每一个合格的表都有一行。

Table 48.87. pg\_publication\_tables Columns

列类型	描述
pubname name (参考 pg_publication.pubname)	发行的名称
schemaname name (参考 pg_namespace.nspname)	包含表的模式名称
tablename name (references pg_class.relname)	表的名称

## 48.87. pg\_replication\_origin\_status

pg\_replication\_origin\_status视图包含有关一个特定源已经重放了多少的信息。更多有关复制源的内容请见Chapter 46

Table 48.88. pg\_replication\_origin\_status Columns

列类型	描述
local_id oid (参考 pg_replication_origin.roident)	内部的节点标识符
external_id text (参考 pg_replication_origin.roname)	外部的节点标识符
remote_lsn pg_lsn	源节点的 LSN，到这个位置的数据都已经被复制。
local_lsn pg_lsn	这个节点的 LSN，remote_lsn已经被复制到这里。使用异步提交时，在将数据持久化到磁盘前用它来刷入提交记录。

## 48.88. pg\_replication\_slots

pg\_replication\_slots视图提供了当前存在于数据库集簇上的所有复制槽的列表，其中也包括复制槽的当前状态。

更多关于复制槽的信息，请见Section 24.2.6和Chapter 45

Table 48.89. pg\_replication\_slots Columns

列类型	描述
slot_name name	一个唯一的、集簇范围内的复制槽标识符
plugin name	包含这个逻辑槽正在使用的输出插件的共享对象基础名称，对于物理槽为空值。
slot_type text	槽类型 - physical或者logical
datoid oid (参考 pg_database.oid)	与这个槽相关的数据库的OID，或者为空值。只有逻辑槽具有相关的数据库。
database name (参考 pg_database.datname)	与这个槽相关的数据库的名称，或者为空值。只有逻辑槽具有相关的数据库。
temporary bool	

列类型	描述
	如果这是一个临时复制槽则为真。临时槽不会被保存在磁盘上并且会在出错或会话结束时自动被删除掉。
active bool	如果这个槽当前正在被使用则为真
active_pid int4	如果槽当前正在被使用，则记录使用这个槽的会话的进程 ID。如果不活动则为NULL。
xmin xid	这个槽要需要数据库保留的最旧事务。VACUUM不能移除被其后续事务删除的元组。
catalog_xmin xid	这个槽要需要数据库保留的影响系统目录的最旧事务。VACUUM不能移除被其后续事务删除的目录元组。
restart_lsn pg_lsn	可能仍被这个槽的消费者要求的最旧WAL地址（LSN），并且因此不会在检查点期间自动被移除。如果这个槽的LSN从未被保留过，则为NULL。
confirmed_flush_lsn pg_lsn	代表逻辑槽的消费者已经确认接收数据到什么位置的地址（LSN）。比这个地址更旧的数据已经不再可用。对于物理槽这里是NULL。
wal_status text	此插槽声称的 WAL 文件的可用性。可能的值为： <ul style="list-style-type: none"> <li>reserved 意味着声称的文件包含 max_wal_size。</li> <li>extended意味着max_wal_size已超出，但文件仍保留，通过复制插槽或wal_keep_size。</li> <li>unreserved意味着该插槽不再保留所需的 WAL 文件，并且将在下一个检查点删除其中一些文件。此状态可以返回到reserved或extended。</li> <li>lost意味着某些需要的 WAL 文件已被删除，并且此插槽不再可用。</li> </ul> <p>最后两种状态仅在max_slot_wal_keep_size为非负值时才看到。如果restart_lsn为 NULL，则此字段为空。</p>
safe_wal_size int8	可写入 WAL 的字节数，以便此插槽不会处于“丢失”状态的危险中。对丢失插槽它是NULL，以及如果max_slot_wal_keep_size是-1。
two_phase bool	如果该插槽为解码准备事务所启用则为真。物理插槽总是为假。

## 48.89. pg\_roles

视图pg\_roles提供了关于数据库角色的信息。这是pg\_authid的一个公共可读视图，它隐去了口令域。

Table 48.90. pg\_roles Columns

列类型	描述
rolname name	角色名
rolsuper bool	

列类型	描述
	角色具有超级用户权限
rolinherit bool	如果此角色是另一个角色的成员，角色能自动继承另一个角色的权限
rolcreatorole bool	角色能创建更多角色
rolcreatedb bool	角色能创建数据库
rolcanlogin bool	角色能登录。即此角色能否被作为初始会话授权标识符
rolreplication bool	角色是一个复制角色。复制角色可以开启复制连接并且创建和删除复制槽。
rolconnlimit int4	对于一个可登录的角色，这里设置角色可以发起的最大并发连接数。-1表示无限制。
rolpassword text	不是口令（总是读起来像*****）
rolvaliduntil timestampz	口令失效时间（只用于口令认证），如果永不失效则为空
rolbypassrls bool	绕过每一条行级安全性策略的角色，详见Section 5.8
rolconfig text[]	运行时配置变量的角色特定默认值
oid oid (references pg_authid.oid)	角色的ID

## 48.90. pg\_rules

视图pg\_rules提供对查询重写规则的信息访问。

Table 48.91. pg\_rules Columns

列类型	描述
schemaname name (references pg_namespace.nspname)	包含表的模式名
tablename name (references pg_class.relname)	规则适用的表名
rulename name (references pg_rewrite.rulename)	规则名
definition text	规则定义（创建命令的重构）

pg\_rules视图排除了视图和物化视图的ON SELECT规则，它们可以在pg\_views和pg\_matviews中找到。

## 48.91. pg\_seclabels

视图pg\_seclabels提供对安全标签的信息访问。它是pg\_seclabel目录的一个便于查询的版本。

Table 48.92. pg\_seclabels Columns

列类型	描述
objoid oid (参考任意 OID 列)	安全标签所属对象的OID
classoid oid (参考 pg_class.oid)	对象出现的系统目录的OID
objsubid int4	对于一个表列上的安全标签，这里是列号 (objoid和classoid指表本身)。对于所有其他对象类型，此列为0。
objtype text	此标签应用的对象类型，以文本方式。
objnamespace oid (references pg_namespace.oid)	如果适用，为此对象的名字空间的OID；否则为空。
objname text	此标签应用的对象名，以文本形式。
provider text (参考 pg_seclabel.provider)	与此标签相关的标签提供者。
label text (references pg_seclabel.label)	应用于此对象的安全标签。

## 48.92. pg\_sequences

视图pg\_sequences提供对数据库中每个序列的信息的访问。

Table 48.93. pg\_sequences Columns

列类型	描述
schemaname name (参考 pg_namespace.nspname)	包含序列的模式名
sequencename name (references pg_class.relname)	序列的名称
sequenceowner name (references pg_authid.rolname)	序列的拥有者的名称
data_type regtype (references pg_type.oid)	序列的数据类型
start_value int8	序列的起始值
min_value int8	序列的最小值
max_value int8	序列的最大值
increment_by int8	序列的增量值
cycle bool	序列是否循环
cache_size int8	序列的缓冲尺寸

列类型	描述
last_value int8	最后一个被写入到磁盘的序列值。如果使用了缓冲，这个值可能比从序列中取出的最后一个值大。如果还没有从该序列读取过，则为空。此外，如果当前用户没有该序列上的USAGE或SELECT特权，则这个值为空。

## 48.93. pg\_settings

视图pg\_settings提供了对服务器上运行时参数的访问。它本质上是SHOW和SET命令的可替换接口。它还提供了SHOW不能提供的关于每一个参数的一些现实，例如最大值和最小值。

Table 48.94. pg\_settings Columns

列类型	描述
name text	运行时配置参数名
setting text	参数的当前值
unit text	参数的隐式单元
category text	参数的逻辑组
short_desc text	参数的简短描述
extra_desc text	附加的参数的详细描述
context text	要求设置此参数值的上下文（参见下方）
vartype text	参数类型 (bool, enum, integer, real, or string)
source text	当前参数值的来源
min_val text	参数的最小允许值（对非数字值为空）
max_val text	参数的最大允许值（对非数字值为空）
enumvals text[]	一个枚举参数的允许值（对非数字值为空）
boot_val text	如果参数没有被别的其他设置，此列为在服务器启动时设定的参数值
reset_val text	在当前会话中，RESET将会设置的参数值
sourcefile text	当前值被设置的配置文件（空值表示从非配置文件的其他来源设置，由不是超级用户也不是pg_read_all_settings成员的用户检查时也为空值），在配置文件中使用时include指令时有用
sourceline int4	当前值被设置的配置文件中的行号（空值表示从非配置文件的其他来源设置，由不是超级用户也不是pg_read_all_settings成员的用户检查时也为空值）。

列类型	描述
pending_restart bool	如果配置文件中修改了该值但需要重启，则为true，否则为false

对于context有多种可能的取值。为了降低改变设置的难度，它们是：

#### internal

这些设置不能被直接修改，它们反映了内部决定的值。某些可能在使用不同配置选项重建系统时或者改变initdb的选项时可以调整。

#### postmaster

这些设置只能在服务器启动时应用，因此任何修改都需要重启服务器。这些设置的值通常都存储在lightdb.conf文件中，或者在启动服务器时通过命令行传递。当然，具有更低context类型的设置也可以在服务器启动时间被设置。

#### sighup

对于这些设置的修改可以在lightdb.conf中完成并且不需要重启服务器。发送一个SIGHUP信号给postmaster会导致它重新读取lightdb.conf并应用修改。Postmaster将会把SIGHUP信号传递给它的孩子进程，这样它们也会获得新的值。

#### superuser-backend

对于这些设置的更改可以在lightdb.conf中进行而无需重启服务器。也可以在连接请求包（例如通过libpq的PGOPTIONS环境变量）中为一个特定的会话设定它们，但是只有在连接用户是超级用户时才能这样做。如果，在会话启动后这些设置就不会改变。如果你在lightdb.conf改变了它们，向postmaster发送一个SIGHUP信号让postmaster重新读取lightdb.conf。新的值将只会影响后续启动的会话。

#### backend

对于这些设置的修改可以在lightdb.conf中完成并且不需要重启服务器。它们也可以在一个连接请求包（例如，通过libpq的PGOPTIONS环境变量）中为一个特定会话设置，任何用户都可以为这个会话做这种修改。然而，这些设置在会话启动后永不变化。如果你在lightdb.conf中修改它们，可以向postmaster发送一个SIGHUP信号让它重读lightdb.conf。新值只会影响后续启动的会话。

#### superuser

这些设置可以从lightdb.conf设置，或者在会话中用SET命令设置。仅当没有通过SET设置会话本地值时，lightdb.conf中的改变才会影响现有的会话。

#### user

这些设置可以从lightdb.conf设置，或者在会话中用SET命令设置。任何用户都被允许修改它们的会话本地值。仅当没有通过SET设置会话本地值时，lightdb.conf中的改变才会影响现有的会话。

更多关于修改这些参数的方法的信息请见Section 17.1

这个视图不显示customized options，直到扩展模块定义它们已经被加载。

这个视图视图不能被插入或者从中删除，但是它可以被更新。在pg\_settings的一行上的一个UPDATE等价于在该参数上执行一个SET命令。修改将只会影响当前会话使用的值。如果一个UPDATE在一个后来中断的事务中被发出，UPDATE命令的效果也会随着事务的回滚而消失。一旦所在的事务被提交，效果将一直保持到会话结束，除非有其他UPDATE或SET重新修改它。



## 48.94. pg\_shadow

视图pg\_shadow的存在是为了向后兼容：它模拟了在LightDB版本8.1之前的一个系统目录。它显示pg\_authid中所有被标记为rolcanlogin的角色的属性。

由于这个表包含口令，所以不能是公众可读的，这也是采用pg\_shadow这个名字的原因。而pg\_user是pg\_shadow上的一个公共可读视图，它屏蔽了口令域。

Table 48.95. pg\_shadow Columns

列类型	描述
username name (参考 pg_authid.rolname)	用户名
usesysid oid (references pg_authid.oid)	用户的ID
usecreatedb bool	用户可创建数据库
usesuper bool	用户为一个超级用户
userepl bool	用户可开启流复制并将系统设置或者取消备份模式。
usebypassrls bool	用户能否绕过所有的行级安全性策略，详见Section 5.8
passwd text	口令（可能被加密），如果没有则为空。关于加密口令如何存储请参见pg_authid。
valuntil timestamptz	口令过期时间（仅用于口令认证）
useconfig text[]	运行时配置变量的会话默认值

## 48.95. pg\_shmem\_allocations

pg\_shmem\_allocations视图显示从服务器的主共享内存段的分配模式。这包括由postgres本身分配的内存和使用Section 36.10.10中详细说明了的通过扩展分配内存的机制。

请注意，此视图不包括使用动态共享内存基础架构分配的内存。

Table 48.96. pg\_shmem\_allocations Columns

列类型	描述
name text	共享内存分配的名字。未使用的内存为NULL，并且匿名分配的为 <anonymous>。
off int8	分配开始的偏移量。匿名分配和未使用的内存为 NULL，因为它们相关的详细信息为未知。
size int8	分配的大小

列类型	描述
allocated_size int8	包括填充的分配的大小。对于匿名分配，没有有关填充的信息可用，因此此size和allocated_size列将始终相等。填充对于可用内存没有意义，因此在这种情况下，列也将相等。

匿名分配是直接使用 `ShmemAlloc()` 分配，而不是通过 `ShmemInitStruct()` 或 `ShmemInitHash()`。

默认的，`pg_shmem_allocations` 视图只能被超级用户读取。

## 48.96. pg\_stats

视图 `pg_stats` 提供对存储在 `pg_statistic` 目录中信息的访问。此视图能访问 `pg_statistic` 行是有限制的，可访问行所对应的表必须是用户有读权限的。因此让所有用户都可以读此视图是安全的。

`pg_stats` 也被设计为能以更适合阅读的格式显示底层目录的信息——但代价是只要为 `pg_statistic` 定义了新的槽类型，就必须扩展此视图的模式。

Table 48.97. `pg_stats` Columns

列类型	描述
schemaname name (references pg_namespace.nspname)	包含表的模式名
tablename name (参考 pg_class.relname)	表的名称
attname name (参考 pg_attribute.attname)	被此行描述的列名
inherited bool	如果为真，表示此行包括继承子列，不仅仅是指定表中的值
null_frac float4	列项中为空的比例
avg_width int4	列的条目的平均字节宽度
n_distinct float4	如果大于零，表示列中可区分值的估计个数。如果小于零，是可区分值个数除以行数的负值（当ANALYZE认为可区分值的数量会随着表增长而增加时采用负值的形式，而如果认为列具有固定数量的可选值时采用正值的形式）。例如，-1表示一个唯一列，即其中可区分值的个数等于行数。
most_common_vals anyarray	列中最常用值的一个列表（如果没有任何一个值看起来比其他值更常用，此列为空）
most_common_freqs float4[]	最常用值的频率列表，即每一个常用值的出现次数除以总行数（如果most_common_vals为空，则此列为空）
histogram_bounds anyarray	将列值划分成大小接近的组的值列表。如果存在most_common_vals，其中的值会被直方图计算所忽略（如果列类型没有一个<操作符或者most_common_vals等于整个值集合，则此列为空）
correlation float4	

列类型	描述
	物理行顺序和列值逻辑顺序之间的统计关联。其范围从-1到+1。当值接近-1或+1时，在列上的一个索引扫描被认为比值接近0时的代价更低，因为这种情况减少了对磁盘的随机访问（如果列数据类型不具有一个<操作符，则此列为空）
most_common_elems anyarray	在列值中，最经常出现的非空元素列表（对标度类型为空）
most_common_elem_freqs float4[]	最常用元素值的频度列表，即含有至少一个给定值实例的行的分数。在每个元素的频度之后有二至三个附加值，它们是每个元素频度的最小和最大值，以及可选的空元素的频度（如果most_common_elems为空，则此列为空）
elem_count_histogram float4[]	在列值中可区分非空元素值计数的一个直方图，后面跟随可区分非空元素的平均数（对于标度类型为空）

在数组域中项的最大数目可以使用ALTER TABLE SET STATISTICS命令控制，或者设置default\_statistics\_target运行时参数从全局上进行控制

## 48.97. pg\_stats\_ext

视图pg\_stats\_ext提供了访问关于数据库中每个扩展统计对象的信息，结合存储在pg\_statistic\_ext和pg\_statistic\_ext\_data目录中的信息。该视图仅允许访问pg\_statistic\_ext和pg\_statistic\_ext\_data的行，这些行对应于用户有权读取的表，因此允许公众读取该视图是安全的。

pg\_stats\_ext也设计成以比底层目录更可读的格式来展示信息——代价是每当有扩展的统计信息的新类型加到pg\_statistic\_ext中时其模式必须扩展。

Table 48.98. pg\_stats\_ext Columns

列类型	描述
schemaname name (references pg_namespace.nspname)	包含表的模式名
tablename name (参考 pg_class.relname)	表的名称
statistics_schemaname name (references pg_namespace.nspname)	包含扩展的统计信息对象的模式名
statistics_name name (参考 pg_statistic_ext.stxname)	扩展的统计信息对象的名称
statistics_owner name (参考 pg_authid.rolname)	扩展的统计信息的拥有者
attnames name[] (references pg_attribute.attname)	包含在扩展统计信息对象中的列的名称
kinds char[]	为此记录启用的扩展统计信息对象的类型
n_distinct pg_ndistinct	列值组合的N-不同计数。如果大于零，则为组合中不同值的估计数量。如果小于零，则为不同值数量的负数除以行数。（当ANALYZE认为随着表的增长不同值的数量可能会增加时，使用负数形式；当该列似乎具有固定数量的可能值时，则使用正数形式。）例如，-1表示列的唯一组合，其中不同组合的数量与行数相同。
dependencies pg_dependencies	功能的依赖关系统计信息

列类型	描述
most_common_vals text[]	列中值的最常见组合的列表（如果没有组合看上去比其它的更常见，则为空。）
most_common_val_nulls bool[]	值最常见组合的NULL标志的列表（当most_common_vals是空值时，为空。）
most_common_freqs float8[]	最常见组合的频率的列表，即每个出现的数量除以行的总数（当most_common_vals是空值时，为空）
most_common_base_freqs float8[]	最常见组合的基本频率的列表，即每个值频率的乘积。（当most_common_vals是空值时，为空。）

数组字段条目的最大数量可以基于一列列地使用ALTER TABLE SET STATISTICS命令来控制，或者通过设置全局的default\_statistics\_target运行时参数来控制。

## 48.98. pg\_tables

视图pg\_tables提供了对数据库中每个表的有用信息的访问。

Table 48.99. pg\_tables Columns

列类型	描述
schemaname name (references pg_namespace.nspname)	包含表的模式的名称
tablename name (references pg_class.relname)	表的名称
tableowner name (references pg_authid.rolname)	表拥有者的名字
tablespace name (参考 pg_tablespace.spcname)	包含表的表空间的名字（如果使用数据库的默认表空间，此列为空）
hasindexes bool (参考 pg_class.relhasindex)	如果表有（或最近有过）任何索引，此列为真
hasrules bool (references pg_class.relhasrules)	如果表有（或曾经有过）规则，此列为真
hastriggers bool (references pg_class.relhastriggers)	如果表有（或者曾经有过）触发器，此列为真
rowsecurity bool (references pg_class.relrowsecurity)	如果表上启用了行安全性则为真

## 48.99. pg\_timezone\_abbrevs

视图pg\_timezone\_abbrevs提供了对当前被时间输入例程识别的时区缩写的列表。当timezone\_abbreviations运行时参数被修改，此视图的内容会发生变化。

Table 48.100. pg\_timezone\_abbrevs Columns

列类型	描述
abbrev text	

列类型	描述
	时区缩写
utc_offset interval	相对于UTC的偏移（正值表示格林威治东部）
is_dst bool	如果这是一个夏令时缩写，则为真

虽然大部分时区缩写表示从 UTC 开始的固定偏移，但是有一些在历史上有值的变化（详见Section D.4）。在这种情况下，这个视图表示它们现在的含义。

## 48.100. pg\_timezone\_names

视图pg\_timezone\_names提供了一个被SET TIMEZONE识别的时区名字的列表，以及它们的相关缩写、UTC偏移和夏令时状态（从技术上来说，LightDB不使用UTC是因为闰秒没有被处理）。和pg\_timezone\_abbrevs中展示的缩写不同，这里很多名字隐含了一组夏令时转换日期规则。因此，相关信息在本地DST边界间变化。所显示的信息基于CURRENT\_TIMESTAMP的当前值计算得来。

Table 48.101. pg\_timezone\_names Columns

列类型	描述
name text	时区名
abbrev text	时区缩写
utc_offset interval	相对于UTC的偏移（正值表示格林威治东部）
is_dst bool	如果当前保持为夏令时则为真

## 48.101. pg\_user

视图pg\_user提供关于数据库用户的信息。这是pg\_shadow的一个公共可读的视图，它消除了口令域。

Table 48.102. pg\_user Columns

列类型	描述
username name	用户名
usesysid oid	用户的ID
usecreatedb bool	用户能创建数据库
usesuper bool	用户为超级用户
userepl bool	用户可开启流复制以及将系统转入/转出备份模式。
usebypassrls bool	

列类型	描述
	用户能否绕过所有的行级安全性策略，详见 Section 5.8
passwd text	不是口令（总是显示为*****）
valuntil timestamptz	口令过期时间（只用于口令认证）
useconfig text[]	运行时配置变量的会话默认值

## 48.102. pg\_user\_mappings

视图pg\_user\_mappings提供有关用户映射的信息。这是pg\_user\_mapping的一个公共可读视图，它对无权使用的用户省去了选项域。

Table 48.103. pg\_user\_mappings Columns

列类型	描述
umid oid (references pg_user_mapping.oid)	用户映射的OID
srvid oid (参考 pg_foreign_server.oid)	包含该映射的外部服务器的OID
srvname name (references pg_foreign_server.srvname)	外部服务器名
umuser oid (references pg_authid.oid)	将被映射的本地角色的OID，如果用户映射是公共的则为零
username name	将被映射的本地用户名
umoptions text[]	用户映射指定选项，以“keyword=value”字符串的形式

为了保护存储为用户映射选项的口令信息，umoptions列将被读作空，除非满足下列情况之一：

- 当前用户就是被映射的用户，并且拥有该服务器或者持有其上的USAGE特权
- 当前用户是服务器的拥有者并且映射是用于PUBLIC
- 当前用户是一个超级用户

## 48.103. pg\_views

视图pg\_views提供了数据库中每个视图的信息。

Table 48.104. pg\_views Columns

列类型	描述
schemaname name (references pg_namespace.nspname)	包含视图的模式名
viewname name (参考 pg_class.relname)	

列类型	描述
	视图名称
viewowner name (references pg_authid.rolname)	视图拥有者的名字
definition text	视图定义 (一个重构的SELECT查询)

---

## Chapter 49. 前端/后端协议

LightDB使用一种基于消息的协议用于前端和后端（服务器和客户机）之间通讯。该协议是在TCP/IP和Unix 域套接字上实现的。端口号 5432 已经在IANA 注册为支持这种协议的服务器的常用端口，但实际上任何非特权端口号都可以使用。

这份文档描述了版本3.0的协议，它在LightDB版本 7.4 和以后的版本中实现。对于以前版本协议的描述，请参考以前版本的LightDB文档。一台服务器能够支持多种协议版本。初始的启动请求消息告诉服务器客户端尝试使用哪个协议版本。如果客户端请求的主版本不被服务器支持，连接将被拒绝（例如，如果客户端请求的协议版本是4.0就会发生这种情况，因为在写作这份文档时，4.0根本还不存在）。如果客户端请求的次版本不被服务器支持（例如客户端请求版本3.1，但服务器仅支持3.0），服务器可能会拒绝该连接或者用一个包含它支持的最高次协议版本的NegotiateProtocolVersion消息进行响应。然后客户端可以选择使用指定的协议版本继续连接或者中止连接。

为了可以有效地为多个客户端提供服务，服务器为每个客户端派生一个新的“后端”进程。在目前的实现里，在检测到新来的连接请求后，马上创建一个新的子进程。不过，这些是对协议透明的。对于协议而言，术语“后端”和“服务器”是可以互换的；类似的还有“前端”和“客户端”也是可以互换的。

### 49.1. 概述

协议在启动和正常操作过程中有不同的阶段。在启动阶段里，前端打开一个到服务器的连接并且认证自身以满足服务器（这可能涉及到一条或多条消息，取决于使用的认证方法）。如果一切正常，服务器就发送状态信息给前端，并最后进入正常操作。除了最初的启动请求消息之外，协议的这个部分是服务器驱动的。

在正常操作中，前端发送查询和其它命令到后端，然后后端返回查询结果和其它响应。在少数几种情况（比如NOTIFY）中，后端会发送未被请求的消息，但这个会话中的绝大多数部分都是由前端请求驱动的。

会话的终止通常是由前端来选择的，但是也可以在某些情况下由后端强制执行。不管在那种情况下，如果后端关闭连接，那么它将在退出之前回滚所有打开的（未完成的）事务。

在正常操作中，SQL命令可以通过两个子协议中的任何一个执行。在“简单查询”协议中，前端只是发送一个文本查询串，然后后端马上分析并执行它。在“扩展查询”协议中，查询的处理被分割为多个步骤：分析、参数值绑定和执行。这样就可以提供灵活性和性能的改进，但代价是额外的复杂性。

正常操作还有用于类似COPY这样的额外的子协议。

#### 49.1.1. 消息概貌

所有通讯都是通过一个消息流进行的。消息的第一个字节标识消息类型，然后后面跟着的四个字节声明消息剩下部分的长度（这个长度包括长度域自身，但是不包括消息类型字节）。剩下的消息内容由消息类型决定。由于历史原因，客户端发送的最初的消息（启动消息）不包含消息类型字节。

为了避免失去与消息流的同步，服务器和客户端通常都是把整个消息读取到一个缓冲区里（使用字节计数），然后才试图处理其内容。这样在处理内容的过程时如果发现错误，就比较容易恢复。在非常极端的情况下（比如说没有足够的内存缓冲消息），接收端可以使用字节计数来判断它在继续读取消息之前需要跳过多少输入。

反之，服务器和客户端都需要注意决不能发送一条不完整的消息。保证这一点的方法通常是在发送整条信息之前先在一个缓冲区里整理整条消息。如果在发送或者接受一条消息的中间发生了通讯错误，那么唯一合理的响应是放弃连接，因为恢复消息边界同步的希望很小。



## 49.1.2. 扩展查询概述

在扩展查询协议中，SQL命令的执行是分割成多个步骤的。步骤与步骤之间保存的状态是由两类的对象代表的：预备语句（prepared statements）和入口（portals）。一个预备语句代表一个文本查询字符串的经过分析、语意解析以及规划之后的结果。一个预备语句不代表它已经可以被执行，因为它可能还缺乏参数的值。一个入口代表一个已经可以执行的或者已经被部分执行过的语句，所有缺失的参数值都已经填充到位了（对于SELECT语句，入口等效于一个打开的游标，但我们使用不同的术语是因为游标不能处理非SELECT语句）。

完整的执行周期包括一个分析步骤，它从一个文本的查询字符串里创建一个预备语句；一个绑定步骤，它用一个预备语句和任何所需的参数值创建一个入口；以及一个执行步骤，它运行一个入口中的查询。如果查询会返回数据行（SELECT、SHOW等），执行步骤会被告知只抓取有限的一些行，这样就可能需要多个执行步骤来完成操作。

后端可以跟踪多个预备语句和入口（但是要注意，这些只存在于一个会话内部，不能在会话之间共享）。现有的预备语句和入口都是用创建它们的时候赋予的名字引用的。另外，还存在一个“未命名”的预备语句和入口。尽管它们的行为和命名对象大部分相同，但是它们是针对只执行一次然后就抛弃的查询而优化的，而在命名对象上的操作是针对多次使用而优化的。

## 49.1.3. 格式和格式代码

特定数据类型的数据可以用几种不同的格式中的任意一种来传递。从LightDB 7.4开始，只支持“文本”和“二进制”两种格式，但是协议为未来的扩展提供了的手段。任意值要求的格式用一个格式代码声明。客户端可以为每个传输的参数值和查询结果的每个列指定一个格式代码。文本的格式代码是零，二进制的格式代码是一，所有其它的格式代码都保留给将来定义。

文本形式的数值是特定数据类型的输入/输出转换函数生成或接受的任何字符串。在传输形式上，字符串没有末尾空字符；如果前端要想把收到的值当作C字符串处理，那么必须自己加上一个（顺便说一下，文本格式不允许嵌入空字符）。

整数的二进制形式采用网络字节序（高位在前）。对于其它数据类型，请参考文档或者源代码获取其二进制形式的信息。请注意，复杂数据类型的二进制形式可能在不同服务器版本之间变化；文本格式通常是最具有移植性的选择。

## 49.2. 消息流

本节描述消息流以及每种消息类型的语意（每种信息的准确形式在Section 49. 𐄀）。根据连接的状态不同，存在几种不同的子协议：启动、查询、函数调用、COPY和终止。还有特殊的规定用于一步操作（包括通知响应和命令取消），这些可能在启动阶段过后的任何时间产生。

### 49.2.1. 启动

要开始一个会话，前端打开一个与服务器的连接并且发送一个启动消息。这个消息包括用户名以及用户希望连接的数据库名；它还标识要使用的特定的协议版本（启动信息可以有选择地包括运行时参数的额外设置）。服务器然后就使用这些信息及服务器配置文件的内容（比如 `lt_hba.conf`）来判断这个连接是否可以接受以及需要什么样的额外的认证（如果需要）。

然后服务器就发送合适的认证请求信息，前端必须用合适的认证响应信息来响应（比如一个口令）。对于除了GSSAPI、SSPI和SASL之外的所有认证方式都至少有一个请求和一个响应。在有些方法中前端不需要发出任何响应，并且因此就不会由任何认证请求发生。对于GSSAPI、SSPI和SASL，可能需要多个包的交换才能完成认证。

认证周期要么以服务器的拒绝连接（ErrorResponse）结束，要么以AuthenticationOK 结束。

这个阶段来自服务器可能消息是：

#### ErrorResponse

连接请求被拒绝。然后服务器马上关闭连接。

#### AuthenticationOk

认证交换成功完成。

#### AuthenticationKerberosV5

现在前端必须与服务器进行一次Kerberos V5认证对话（在这里没有描述，Kerberos规范的一部分）。如果对话成功，服务器响应一个AuthenticationOk，否则它响应一个ErrorResponse。这已经不再被支持。

#### AuthenticationCleartextPassword

现在前端必须以明文形式发送一个包含口令的PasswordMessage。如果这是正确的口令，服务器用一个 AuthenticationOk，否则它响应一个ErrorResponse。

#### AuthenticationMD5Password

现在前端必须发送一个PasswordMessage，其中包含口令，且口令先用用户名做MD5加密，然后使用在AuthenticationMD5Password消息里指定的4字节盐粒加密。如果这是正确口令，服务器用一个AuthenticationOk 响应，否则它用一个ErrorResponse 响应。实际的PasswordMessage可以用SQL来计算：`concat('md5', md5(concat(md5(concat(password, username)), random-salt)))`（记住md5()函数返回的结果是一个十六进制串）。

#### AuthenticationSCMCredential

这个响应只用于在支持SCM信任消息的平台上的本地Unix域连接。前端必须发出一条SCM信任消息然后发送一个数据字节（数据字节的内容并没有意义；它只被用于确保服务器等待足够长的时间来接受信任信息）。如果信任是可以接受的，那么服务器用AuthenticationOk响应，否则用ErrorResponse响应（该消息只在9.1之前的服务器中发出。它可能最终会从协议规范中被删除）。

#### AuthenticationGSS

前端必须现在开始一次GSSAPI谈判。前端将发送一个带有GSSAPI数据流第一部分的GSSResponse消息来响应。如果需要进一步的消息，服务器将会响应AuthenticationGSSContinue。

#### AuthenticationSSPI

前端必须现在开始一次SSPI谈判。前端将发送一个带有SSPI数据流第一部分的GSSResponse来响应。如果需要进一步的消息，服务器将会响应AuthenticationGSSContinue。

#### AuthenticationGSSContinue

这个消息包含对于前一步的GSSAPI或SSPI谈判（AuthenticationGSS、AuthenticationSSPI或者前一个AuthenticationGSSContinue）的响应数据。如果这个消息中的GSSAPI或SSPI数据指示需要更多数据来完成认证，前端必须将所需的数据作为另一个GSSResponse发送。如果这个消息就能完成GSSAPI或SSPI认证，服务器将接着发送AuthenticationOk来指示成功认证，或者发送ErrorResponse来指示失败。

#### AuthenticationSASL

前端现在必须发起一次SASL协商，使用该消息中列出的一种SASL机制。前端将用选中机制的名字发送一个SASLInitialResponse，并且SASL数据流的第一部分对应于此。如果需要进一步的消息，服务器将用AuthenticationSASLContinue回复。详见Section 49.3

#### AuthenticationSASLContinue

这个消息包含从SASL协商（AuthenticationSASL或者一个之前的AuthenticationSASLContinue）的前一步中得到的挑战数据。前端必须用一个SASLResponse消息响应。

#### AuthenticationSASLFinal

已经用额外的与机制相关的数据为该客户端完成SASL认证。服务器接下来将发送AuthenticationOk来表示认证成功，或者发送ErrorResponse表示失败。只有在SASL机制指定要在完成时从服务器发送额外数据到客户端的情况下才会发送这个消息。

#### NegotiateProtocolVersion

服务器不支持客户端请求的次协议版本，但是支持该协议更早的一个版本，这个消息会指出受支持的最高次版本。如果客户端在启动包中请求了不受支持的协议选项（例如以\_pq\_开始），则这个消息也会被发出。这个消息后面将会跟着一个ErrorResponse或者一个指示认证成功或失败的消息。

如果前端不支持服务器要求的认证方式，那么它应该马上关闭连接。

在收到AuthenticationOk包之后，前端必须等待来自服务器的进一步消息。在这个阶段会启动一个后端进程，而前端只是一个感兴趣的旁观者。启动尝试仍有可能失败（ErrorResponse），服务器也有可能拒绝支持所请求的次协议版本（NegotiateProtocolVersion），但是通常情况下，后端将发送一些ParameterStatus消息、BackendKeyData以及最后的ReadyForQuery。

在这个阶段，后端将尝试应用任何在启动消息里给出的额外的运行时参数设置。如果成功，这些值将成为会话的缺省值。错误将导致ErrorResponse并退出。

这个阶段来自后端的可能消息是：

#### BackendKeyData

这个消息提供了密钥数据，前端如果想要在稍后发出取消请求，则必须保存这个数据。前端不应该响应这个信息，但是应该继续侦听等待ReadyForQuery消息。

#### ParameterStatus

这个消息告诉前端有关后端参数的当前（初始）设置，比如client\_encoding或者DateStyle等。前端可以忽略这些信息，或者记录其设置用于将来使用；参阅Section 49.2. 获取更多细节。前端不应该响应这些信息，而是应该继续侦听ReadyForQuery消息。

#### ReadyForQuery

启动成功，前端现在可以发出命令。

#### ErrorResponse

启动失败，在发送完这个消息之后连接被关闭。

#### NoticeResponse

发出了一个警告信息。前端应该显示这个信息，但是要继续监听ReadyForQuery或ErrorResponse。

后端在每个命令周期后都会发出一个相同的ReadyForQuery消息。出于前端的编码需要，前端可以合理地认为ReadyForQuery是一个命令周期的开始，或者认为ReadyForQuery是启动阶段和每个随后命令周期的结束，具体是那种情况取决于前端的编码需要。

## 49.2.2. 简单查询

一个简单查询周期是由前端发送一条Query消息给后端进行初始化的。这条消息包含一个用文本字符串表达的 SQL 命令（或者一些命令）。后端根据查询命令串的内容发送一条或者更多条响应消息给前端，并且最后是一条ReadyForQuery响应消息。ReadyForQuery通知前端它可以安全地发送新命令了（实际上前端不必在发送其它命令之前等待ReadyForQuery，但是这样一来，前端必须能发现较早发出的命令失败而稍后发出的命令成功的情况）。

来自后端的可能的消息是：

CommandComplete

一个SQL命令正常结束。

CopyInResponse

后端已经准备好从前端拷贝数据到一个表里面去，参见Section 49.2.6

CopyOutResponse

后端已经准备好从一个表里拷贝数据到前端，参见Section 49.2.6

RowDescription

表示为了响应一个SELECT、FETCH等查询，将要返回行。这条消息的内容描述了行的布局。这条消息后面将跟着DataRow消息，每个DataRow消息包含一个要被返回的行。

DataRow

SELECT、FETCH等查询返回的结果集中的一行。

EmptyQueryResponse

识别了一个空的查询字符串。

ErrorResponse

出错了。

ReadyForQuery

查询字符串的处理完成。发送一个独立的的消息来标识这种情况是因为查询字符串可能包含多个SQL命令（CommandComplete只是标记一条SQL命令处理完毕，而不是整个查询字符串处理完毕）。不管是处理成功结束还是产生错误，ReadyForQuery总会被发送。

NoticeResponse

发送了一个与查询有关的警告信息。提示信息是附加在其他响应上的，也就是说，后端将继续处理该命令。

对SELECT查询（或其它返回行集的查询，比如EXPLAIN或SHOW）的响应通常包含RowDescription、零个或者多个 DataRow 消息以及最后的 CommandComplete。COPY到前端或者从前端COPY会调用Section 49.2.6里描述的特殊协议。所有其他查询类型通常只产生一个CommandComplete消息。

因为查询字符串可能包含若干个查询（用分号分隔），所以在后端完成查询字符串的处理之前可能有好几个这样的响应序列。如果整个字符串已经处理完，后端已经准备好接受新查询字符串的时候则发出 ReadyForQuery消息。

如果收到一个完全空（除了空白之外没有内容）的查询字符串，那么响应是一条EmptyQueryResponse后面跟着ReadyForQuery。

在出现错误的时候，发出一个ErrorResponse消息，后面跟着ReadyForQuery。查询字符串的所有进一步的处理都被ErrorResponse中止（即使里面还有查询）。请注意这些事情可能在处理一个查询产生的消息序列的中途发生。

在简单查询模式中，检索出来的值的格式总是文本，除非给出的命令是在一个使用BINARY选项声明的游标上FETCH。在这种情况下，检索出来的值是二进制格式的。在 RowDescription消息里给出的格式代码将告诉我们用了那种格式。

前端在等待其他类型的消息时必须准备接收ErrorResponse和NoticeResponse消息。 参阅 Section 49.2. 来了解后端因为外部事件可能生成的消息。

我们建议的方法是把前端代码写成状态机的风格，它可以在任何时刻接受任何有意义的消息类型，而不是假设消息的序列总是准确。

### 49.2.2.1. 一个简单查询中的多条语句

当一个简单查询消息中包含多于一条SQL语句（被分号分隔）时，那些语句会被当做一个事务中执行，除非其中包括显式事务控制命令来强制不同的行为。例如，如果消息包括

```
INSERT INTO mytable VALUES(1);
SELECT 1/0;
INSERT INTO mytable VALUES(2);
```

则SELECT中的除零失败将强制回滚第一个INSERT。进而，因为该消息的执行在第一个错误时就被放弃，第二个INSERT根本都不会被尝试。

如果该消息包含的是

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELECT 1/0;
```

那么第一个INSERT会被这个显式的COMMIT命令提交。第二个INSERT以及SELECT仍会被当作一个单一事务，这样除零失败将回滚第二个INSERT，但不会回滚第一个。

这种行为通过在一个隐式事务块中的一个多语句Query消息中运行那些语句来实现，除非它们运行在某个显式事务块中。隐式事务块与常规事务块之间的区别在于隐式块会在Query消息结束时自动被关闭，或者是在没有错误的情况下由一个隐式提交关闭，或者是在有错误时由一个隐式的回滚关闭。这类似于一个语句自己执行（当不在事务块中时）时发生的隐式提交或回滚。

如果会话已经在事务块中，作为前面某个消息中BEGIN的结果，那么Query消息会简单地继续那个事务块，不管该消息包含一个语句还是多个语句。不过，如果该Query消息包含一个关闭现有事务块的COMMIT或者ROLLBACK，那么任何接下来的语句都会在一个隐式事务块中被执行。反过来，如果在多语句Query消息中出现一个BEGIN，那么它会开始一个常规事务块，这个常规事务块将只能被一个显式的COMMIT或者ROLLBACK终止，不管这两种命令是出现在这个Query消息还是后面的一个Query消息中。如果BEGIN跟在一些作为隐式事务块执行的语句后面，那些语句不会被立刻提交。实际上，它们会被包括到新的常规事务块中。

出现在一个隐式事务块中的COMMIT或者ROLLBACK会被正常执行并且关闭该隐式块。不过，由于没有先前的BEGIN配对的COMMIT或者ROLLBACK表示一种错误，所以将会发出一个警告。如果后面还有更多语句，将会为它们开始一个新的隐式事务块。

在隐式事务块中不允许保存点，因为它们会与发生错误时自动关闭块的行为发生冲突。

记住，不管任何事务控制命令存不存在，Query消息的执行会在第一个错误时停止。因此，对于下面的在一个Query消息中的例子

```
BEGIN;
SELECT 1/0;
ROLLBACK;
```

会话中将留下一个失败的常规事务块，因为在出现除零错误后不会到达ROLLBACK。将需要另一个ROLLBACK把会话恢复到一种可用的状态。

另一种要注意的行为是，最初的词法和语法分析是在整个查询字符串被执行之前进行的。因此后面的语句中的简单错误（例如拼写错误的关键词）可能会阻止任何语句的执行。这通常对用户是不可见的，因为在当作一个隐式事务块执行时，这些语句不管怎样都会全部被回滚。不过，在尝试于一个多语句Query中执行多个事务时，这种现象可能是可见的。例如，如果一个拼写错误把我们之前的例子变成

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELCT 1/0;
```

那么这些语句都不会被运行，导致可见的差别，即第一个INSERT没有被提交。在语义分析及其后阶段检测到的错误（例如拼错的表名或者列名）不会有这种效果。

### 49.2.3. 扩展查询

扩展查询协议把上面描述的简单协议分裂成若干个步骤。准备步骤的结果可以被多次复用以提高效率。另外，还可以获得额外的特性，比如可以把数据值作为独立的参数提供而不是必须把它们直接插入一个查询字符串。

在扩展协议里，前端首先发送一个Parse消息，它包含一个文本查询字符串，另外还有一些可选的有关参数占位符的数据类型的信息，以及一个最终的预备语句对象的名字（一个空字符串选择未命名的预备语句）。响应是一个ParseComplete或者ErrorResponse。参数的数据类型可以用OID来指定；如果没有给出，那么分析器将试图用应付无类型文字字符串常量的方法来推导其数据类型。

#### Note

一个参数的数据类型可以通过设置为零，或者让参数类型OID的数目比查询字符串里的参数符号（\$n）的数目少的方式不予指定。另外一个特例是参数的类型可以声明为void（也就是说，伪类型void的OID）。这是为了允许用于某些函数参数的参数符号实际上是OUT参数。通常情况下，没有什么环境会用到void参数，但是在函数的参数列表里出现了这么一个参数符号，那么它实际上会被忽略。比如，一个像这样的函数调用：foo(\$1,\$2,\$3,\$4)，如果\$3和\$4被指定为具有类型是void，那么这个函数调用会匹配一个带有两个IN和两个OUT参数的函数。

#### Note

在一个Parse消息里包含的查询字符串不能包含超过一个SQL语句；否则就会报告一个语法错误。这个限制在简单查询协议中并不存在，它只存在于扩展协议中，因为允许预备语句或者入口包含多个命令将导致协议过度地复杂。

如果成功创建了一个命名的预备语句对象，那么它将持续到当前会话结束，除非被明确地删除。一个未命名的预备语句只持续到下一个声明未命名语句的Parse语句发出为止（请注

意一个简单的查询消息也会销毁未命名语句)。命名预备语句必须被明确地关闭，然后才能用一个Parse消息重新定义，但是未命名语句并不要求这个动作。命名预备语句也可以在SQL命令级别创建和访问，方法是使用PREPARE和EXECUTE。

一旦一个预备语句存在，就可以使用Bind消息使之进入执行状态。Bind消息给出源预备语句的名字（空字符串表示未命名预备语句）、目标入口的名字（空字符串表示未命名的入口）及用于那些在预备语句中出现的所有参数占位符的值。提供的参数集必须匹配那些预备语句所需要的参数（如果你在Parse消息里声明任何void参数，那么在Bind消息里给它们传递NULL值）。Bind还指定被查询返回的数据的格式；格式可以在总体上声明，也可以对每个列进行声明。响应是BindComplete或ErrorResponse。

### Note

输出的格式是文本还是二进制是由Bind里给出的格式代码决定的，而不管涉及的是什么SQL命令。在使用扩展查询协议的时候，游标声明里的BINARY属性是无关的。

当Bind消息被处理时通常会进行查询规划。如果预备语句没有参数或者是被反复执行，服务器可能会保存创建好的计划并在后续对同一个预备语句的Bind消息中重用之。不过，当它发现可以创建一个效率比依赖指定参数值的计划不低很多的一般性计划时，它仍然会进行查询规划。但是这对于协议所关注的来说是透明的。

如果成功创建了一个命名入口对象，那么它将持续到当前事务的结尾，除非被明确地删除。一个未命名入口在事务的结尾删除，或者是在发出的下一个Bind语句声明了一个未命名入口为止（请注意一个简单的查询消息也会删除这个未命名入口）。命名入口在可以用一个Bind消息重新定义之前必须明确地关闭，但是未命名入口不要求这个动作。命名入口也可以在SQL命令的级别创建和访问，方法是使用DECLARE CURSOR和FETCH。

一旦一个入口存在，那么就可以用一个Execute消息执行它。Execute消息指定入口的名字（空字符串表示未命名入口）和一个最大的结果行数（零表示“取出所有行”）。结果行数只对包含返回行集的入口有意义；在其它情况下，该命令总是被执行到结束，而行计数会被忽略。Execute消息的可能响应和那些通过简单查询协议发出的查询一样，只不过执行不会导致后端发出ReadyForQuery或者RowDescription。

如果Execute在入口的执行完成之前终止（因为达到了一个非零的结果行数），它将发送一个PortalSuspended消息；这个消息的出现告诉前端应该在同一个入口上发出另外一个Execute消息以完成操作。在入口的执行完成之前，不会发出表示源SQL命令结束的CommandComplete消息。因此执行阶段总是由下列消息之一出现标志着结束：CommandComplete、EmptyQueryResponse（如果入口是从一个空字符串创建出来的）、ErrorResponse或者PortalSuspended。

每个扩展查询消息序列完成后，前端都应该发出一条Sync消息。这个无参数的消息导致后端关闭当前事务——如果当前事务不是在一个BEGIN/COMMIT事务块中（“关闭”的意思就是在没有错误的情况下提交，或者是有错误的情况下回滚）。然后响应一条ReadyForQuery消息。Sync的目的是提供一个错误恢复的重新同步的点。如果在处理任何扩展查询消息的时候侦测到任何错误，那么后端会发出ErrorResponse，然后读取并抛弃消息直到一个Sync到来，然后发出ReadyForQuery并且返回到正常的消息处理中（但是要注意如果正在处理Sync的时候发生了错误，那么不会忽略任何东西——这样就保证了为每个Sync发出一个并且只是一个ReadyForQuery）。

### Note

Sync并不导致一个用BEGIN打开的事务块关闭。我们可以侦测到这种情况，因为ReadyForQuery消息包含事务状态信息。

除了这些基本的、必须的操作之外，在扩展查询协议里还有几种可选的操作可以使用。

Describe消息（入口变体）指定一个现有的入口的名字（或者一个空字符串表示未命名入口）。响应是一个RowDescription消息，它描述了执行该入口将要返回的行；或者是一个NoData消息——如果入口并不包含会返回行的查询；或者是一个ErrorResponse——如果入口不存在。

Describe消息（语句变体）指定一个现有的预备语句的名字（或者一个空字符串表示未命名预备语句）。响应是一个描述该语句需要的参数的ParameterDescription消息，后面跟着一个描述该语句最终执行后返回的行的RowDescription消息（或者是 NoData 消息，如果该语句不返回行）。如果没有这样的预备语句，则返回ErrorResponse。请注意因为还没有发出Bind，所以后端还不知道用于返回列的格式；在这种情况下，RowDescription消息里面的格式代码域将是零。

### Tip

在大多数情况下，前端在发出Execute之前应该发出某种Describe的变体，以保证它知道如何解析它将得到的结果。

Close消息关闭一个现有的预备语句或者入口，并且释放资源。对一个不存在的语句或者入口发出Close不是一个错误。响应通常是CloseComplete，但如果在释放资源的时候遇到了一些困难也可以是ErrorResponse。请注意关闭一个预备语句会隐含地关闭任何从该语句构造出来的打开的入口。

Flush消息不产生任何特定的输出，但会强制后端发送任何还在它的输出缓冲区中待处理的数据。Flush必须在除Sync外的任何扩展查询命令后面发出——如果前端希望在发出更多的命令之前检查该命令的结果的话。如果没有Flush，后端返回的消息将组合成尽可能少的数据包，以减少网络负荷。

### Note

简单查询消息大概等于一系列使用未命名预备语句和无参数入口对象的Parse、Bind、入口Describe、Execute、Close、Sync。一个区别是它会在查询字符串中接受多个SQL语句，并连续地为每个语句自动执行绑定/描述/执行序列。另外一个区别是它不会返回ParseComplete、Bindcomplete、CloseComplete或者NoData消息。

## 49.2.4. 流水线处理

使用扩展查询协议允许 流水线传输，这意味着可以发送一系列查询，而无需等待之前的查询完成。这减少了完成一系列操作所需的网络往返次数。但是，如果其中一个步骤失败，用户必须仔细考虑所需的行为，因为后续查询已经在向服务器发送。

应对此问题的一种方法是将整个查询系列作为单个事务处理，即将其包装在BEGIN...COMMIT 中。但是，如果希望某些命令独立于其他命令提交，则此方法无济于事。

扩展查询协议提供了另一种管理此问题的方法，即在依赖的步骤之间省略发送 Sync 消息。由于在出现错误后，后端将跳过命令消息直到找到 Sync，因此当较早的命令失败时，这允许流水线中后续的命令自动跳过，而无需客户端使用 BEGIN 和 COMMIT 显式地管理。可独立提交的流水线段可以通过 Sync 消息分隔开来。

如果客户端没有发出显式的 BEGIN，那么每个 Sync 通常会在前面的步骤成功时隐式地执行 COMMIT，或在前面的步骤失败时隐式地执行 ROLLBACK。然而，有一些 DDL 命令（例如 CREATE DATABASE）不能在事务块中执行。如果在流水线中执行其中的一个，它将在成功时强制立即提交以保持数据库的一致性。在这些命令之后紧接着的 Sync 没有任何效果，除了用 ReadyForQuery 响应。



使用这种方法时，必须通过计算 `ReadyForQuery` 消息并等待其达到发送的 `Sync` 数量来确定管道的完成。计算命令完成响应是不可靠的，因为其中一些命令可能不会被执行，因此不会产生完成消息。

## 49.2.5. 函数调用

函数调用子协议允许客户端请求一个对存在于数据库 `pg_proc` 系统表中的任意函数的直接调用。客户端必须在该函数上有执行的权限。

### Note

函数调用子协议是一个遗留的特性，在新代码里可能最好避免用它。类似的结果可以通过设置一个执行 `SELECT function($1, ...)` 的预备语句得到。这样函数调用周期就可以用 `Bind/Execute` 代替。

一个函数调用周期是由前端向后端发送一条 `FunctionCall` 消息初始化的。然后后端根据函数调用的结果发送一条或者更多响应消息，并且最后是一条 `ReadyForQuery` 响应消息。 `ReadyForQuery` 通知前端它可以安全地发送一个新的查询或者函数调用了。

来自后端的可能的响应消息是：

`ErrorResponse`

发生了一个错误。

`FunctionCallResponse`

函数调用完成并且在消息中返回一个结果（请注意函数调用协议只能处理单个标量结果，不能处理行类型或者集合类型的结果）。

`ReadyForQuery`

函数调用处理完成。 `ReadyForQuery` 将总是被发送，不管是成功完成处理还是发生一个错误。

`NoticeResponse`

发出了一条有关该函数调用的警告信息。通知是附加在其他响应上的，也就是说，后端将继续处理该命令。

## 49.2.6. COPY操作

`COPY` 命令允许在服务器和客户端之间进行高速大批量数据传输。拷贝入和拷贝出操作每个都把连接切换到一个独立的子协议中，并且持续到操作结束。

拷贝入模式（数据传输到服务器）是在后端执行一个 `COPY FROM STDIN` 语句的时候初始化的。后端发送一个 `CopyInResponse` 消息给前端。前端应该发送零条或者更多 `CopyData` 消息，形成一个输出数据的流（消息的边界和行的边界没有任何相关性要求，尽管通常那是合理的选择）。前端可以通过发送一个 `CopyDone` 消息来终止拷贝入操作（允许成功终止），也可以发出一个 `CopyFail` 消息（它将导致 `COPY` 语句带着错误失败）。然后后端就恢复回它在 `COPY` 开始之前的命令处理模式，可能是简单查询协议，也可能是扩展查询协议。然后它会发送 `CommandComplete`（如果成功）或者 `ErrorResponse`（如果失败）。

如果在拷贝入模式下后端检测到了错误（包括接受接收到 `CopyFail` 消息，或者是任何除了 `CopyData` 或者 `CopyDone` 之外的前端消息），那么后端将发出一个 `ErrorResponse` 消息。如果 `COPY` 命令是通过一个扩展查询消息发出的，那么后端从现在开始将抛弃前端消息，直到

一个Sync消息到达，然后它将发出ReadyForQuery并且返回到正常的处理中。如果COPY命令是在一个简单查询消息里发出的，那么该消息剩余部分被丢弃然后发出ReadyForQuery消息。不管是哪种情况，任何前端发出的CopyData、CopyDone或者CopyFail消息都将被简单地抛弃。

在拷贝入模式下，后端将忽略所收到的Flush和Sync消息。收到任何其他非拷贝消息类型都会造成一个错误，它将导致上面所描述的拷贝入状态中断（Flush和Sync的例外是为了方便客户端库，它们总是在一个Execute消息之后发送Flush和Sync，而不检查被执行的命令是否为一个COPY FROM STDIN）。

拷贝出模式（数据从服务器发出）是在后端执行一个COPY TO STDOUT语句的时候初始化的。后端发出一个CopyOutResponse消息给前端，后面跟着零或者多个CopyData消息（总是每行一个），然后跟着CopyDone。然后后端回退到它在COPY开始之前的命令处理模式，然后发送CommandComplete。前端不能退出传输（除非是关闭连接或者发出一个Cancel请求），但是它可以抛弃不需要的CopyData和CopyDone消息。

在拷贝出模式中，如果后端检测到错误，那么它将发出一个ErrorResponse消息并且回到正常的处理。前端应该把收到ErrorResponse当作终止拷贝出模式的标志。

在CopyData消息中间可能会散布有NoticeResponse和ParameterStatus消息。前端必须处理这些情况，并且应该也为异步消息类型（参见Section 49.2.7 准备好。否则任何除CopyData或CopyDone之外的消息类型都会被认为是要中止拷贝出模式。

还有另外一种被称为双向拷贝的与拷贝相关的模式，它允许“向”和“从”服务器高速传输大批量数据。当后端处于walsender模式中执行一个START\_REPLICATION语句时，它会启动双向拷贝模式。后端会发送一个CopyBothResponse消息给前端。然后前端和后端都会发送CopyData消息，然后直到最后发送一个CopyDone消息。在客户端发送一个CopyDone消息后，连接将从双向拷贝模式转换到拷贝出模式，并且客户端将不能发送更多CopyData消息。类似的，当服务器发送了一个CopyDone消息，连接进入到拷贝入模式，并且服务器将不能发送更多CopyData消息。在双方发送完一个CopyDone消息后，拷贝模式被中断，而后端将回到之前的命令处理模式。如果在双向拷贝模式中出现一个后端检测到的错误，后端将发出一个ErrorResponse消息，然后将发出ReadyForQuery并返回到普通处理。前端将把收到ErrorResponse作为在双向上中断拷贝的信号，在这种情况下不会有CopyDone被发出。关于在双向拷贝模式下传输的子协议请参见Section 49.4

CopyInResponse、CopyOutResponse和CopyBothResponse消息包括域和格式代码，域告诉前端每行的列数，而格式代码则用于具体每个列（就目前的实现而言，一个给定COPY操作中的所有列都将使用同样的格式，但是消息设计并不做这个假设）。

## 49.2.7. 异步操作

有几种情况下后端会发送一些并非由特定前端命令流传达的消息。在任何时候前端都必须准备处理这些消息，即使它是并未参与一个查询。在最低限度下，我们应该在开始读取查询响应之前检查这些情况。

NoticeResponse消息有可能是因为外部的活动而产生的；比如，如果数据库管理员进行一次“快速”数据库关闭，那么后端将在关闭连接之前发送一个NoticeResponse来表明这些。相应地，前端应该总是准备接受和显示NoticeResponse消息，即使连接事实上是空闲的。

如果任何时候有任何参数值的活跃值改变且后端认为前端应该知道这些，那么都会产生ParameterStatus消息。这种情况最常见发生的情形是对前端执行的一个SET命令进行响应，并且这种情况实际上是同步的 — 但是也有可能是数据库管理员改变了配置文件然后项服务器发出SIGHUP信号导致了参数状态的变化。同样，如果一个SET命令回滚，那么也会生成一个合适的ParameterStatus 消息以报告当前有效值。

目前，系统内有一套会生成ParameterStatus消息的写成硬代码的参数，它们是：（server\_encoding, TimeZone 和 integer\_datetimes 在 8.0 版本之前没有报告。standard\_conforming\_strings 在版本 8.1 之前没有报告。） 请注意 server\_version, server\_encoding 和 integer\_datetimes 是伪参数，启动后不能

修改。 这些可能在将来改变，或者甚至是变成可以配置的。 因此，前端应该简单地忽略那些它不懂或者不关心的 `ParameterStatus`。 `server_version`、`server_encoding`、`client_encoding`、`application_name`、`is_superuser`、`session_authorization`、`DateStyle`、`IntervalStyle`、`TimeZone`、`integer_datetimes`以及`standard_conforming_strings` (`server_encoding`、`TimeZone`以及`integer_datetimes`在版本8.0之前不会被报告；`standard_conforming_strings`在版本8.1之前不会被报告；`IntervalStyle`在版本8.4之前不会被报告；`application_name`在版本9.0之前不会被报告)。 注意`server_version`、`server_encoding`以及`integer_datetimes`是伪参数，它们不能在启动之后被改变。这种设置可能在未来改变，甚至变成可配置的。相应地，一个前端应该简单地忽略那些与它不懂或者不关心的参数相关的`ParameterStatus`。

如果前端发出一个LISTEN命令，那么无论何时在为同一个通道名NOTIFY时，后端将发送一个`NotificationResponse`消息（不要和`NoticeResponse`搞混！）。

### Note

目前，`NotificationResponse`只能在一个事务外面发送，因此它将不会在一个命令响应序列中间出现，但是它可能正好在`ReadyForQuery`之前出现。不过，在前端逻辑中做上述假设是不明智的。好的做法是在协议的任何点上都可以接受`NotificationResponse`。

## 49.2.8. 取消正在处理的请求

在一条查询正在处理的时候，前端可以请求取消该查询。这种取消请求不是直接通过打开的连接发送给后端的，这么做是因为实现的效率：我们不希望后端在处理查询的过程中不停地检查前端的输入。取消请求应该相对而言比较少见，所以我们把取消做得稍微笨拙一些，以便不影响正常状况的性能。

要发出一条取消请求，前端打开一个与服务器的新连接并且发送一条`CancelRequest`消息，而不是通常在新连接中经常发送的`StartupMessage`消息。服务器将处理这个请求然后关闭连接。出于安全原因，对取消请求消息不做直接的响应。

除非`CancelRequest`消息包含在连接启动过程中传递给前端的相同的关键数据（PID和密钥），否则它将被忽略。如果该请求匹配当前运行着的后端的PID和密钥，则退出当前查询的处理（目前的实现里采用的方法是向正在处理该查询的后端进程发送一个特殊的信号）。

取消信号可能产生或者不产生效果——例如，如果它在后端完成查询的处理后到达，那么它就没有做用。如果取消起作用了，会导致当前命令伴随着一个错误消息提前退出。

这么做是对安全性和有效性通盘考虑的结果，前端没有直接的方法获知一个取消请求是否成功。它必须继续等待后端对查询响应。发出一个取消仅仅是增加了当前查询快些结束的可能性，同时也增加了当前查询会伴随着一条错误消息失败而不是成功执行的可能性。

因为取消请求是通过新的联接发送给服务器而不是通过平常的前端/后端通讯链接，所以取消请求可能被任意进程发出的，而不仅仅是要取消查询的前端。这样可能对创建多进程应用提供了更多的灵活性。同时这样也带来了安全风险，因为任何一个非授权用户都可能试图取消查询。这个安全风险通过要求在取消请求中提供一个动态生成的密钥来解决。

## 49.2.9. 终止

通常优雅的终止过程是前端发送一条`Terminate`消息并且立刻关闭连接。一旦收到消息，后端马上关闭连接并且终止。

在少数情况下（比如一个管理员命令数据库关闭），后端可能在没有任何前端请求的情况下断开连接。在这种情况下，后端将在它断开连接之前尝试发送一个错误或者通知消息给出断开的原因。

其它终止的情况发生在各种失败的场合，比如某一方的内核转储、失去通讯链路、丢失了消息边界同步等。不管是前端还是后端看到了一个意外的连接关闭，那么它应该清理现场并且终止。如果前端不想终止自己，那么它有一个选项是重连服务器的方法启动一个新的后端。如果收到了一个无法识别的消息类型，那么我们也建议关闭连接，因为出现这种情况可能意味着是丢失了消息边界的同步。

不管是正常还是不正常的终止，任何打开的事务都会回滚而不是提交。不过，我们应该注意的是如果一个前端在一个非SELECT查询正在处理的时候断开，那么后端很可能在发现断开之前先完成查询的处理。如果查询处于任何事务块之外（BEGIN ... COMMIT序列），那么其结果很可能在得知断开之前被提交。

## 49.2.10. SSL会话加密

如果编译LightDB的时候打开了SSL支持，那么前后端通讯就可以用SSL加密。这样就提供了一种在攻击者可能捕获会话通讯数据包的环境下保证通讯安全的方法。有关使用SSL加密LightDB会话的更多信息，请参阅Section 16.9

要开始一次SSL加密连接，前端先是发送一个SSLRequest消息，而不是StartupMessage。然后服务器以一个包含S或N的字节响应，分别表示它愿意还是不愿意进行SSL。如果此时前端对响应不满意，那么它可以关闭连接。要在S之后继续，那么先进行与服务器的SSL启动握手（没有在这里描述，是SSL规范的一部分）。如果这些成功了，那么继续发送普通的StartupMessage。这种情况下，StartupMessage和所有随后的数据都将由SSL加密。要在N之后继续，则发送普通的StartupMessage并不适用加密继续处理。

前端应该也准备处理一个来自服务器的给SSLRequest的ErrorMessage响应。这种情况只在服务器早于LightDB的SSL支持的情况下才会出现（这种服务器现在非常古老，并且可能不再存在了）。在这种情况下，连接必需关闭，但是前端可以选择打开一个新的连接然后不使用SSL进行连接。

当可以执行SSL加密时，服务器预计只发送单个S字节，然后等待前端启动SSL握手。如果此时有其他字节可读，则很可能意味着中间人正在尝试执行缓冲区填充攻击（CVE-2021-23222<sup>1</sup>）。前端应该编写代码，要么在将套接字交给SSL库之前从套接字中精确读取一个字节，要么在发现已读取额外字节时将其视为协议违规行为。

一个初始化的SSLRequest也可以用于打开来用于发送一条CancelRequest消息的联接中。

如果协议本身并未提供某种方法强制SSL加密，那么管理员可以把服务器配置为拒绝未加密的会话，这是认证检查的一个副产品。

## 49.2.11. GSSAPI 会话加密

如果LightDB是使用GSSAPI支持构建的，则可以使用GSSAPI加密前端/后端之间的通信。这为攻击者可能捕获会话流量的环境中提供了通信安全性。有关使用GSSAPI加密LightDB会话的详细信息，请参阅Section 16.10

要启动GSSAPI加密连接，前端最初会发送GSSENCRequest消息，而不是StartupMessage。然后，服务器使用包含G或N的单个字节进行响应，分别表示愿意或不愿意执行GSSAPI加密。如果前端对响应不满意，则此时可能会关闭连接。要在G之后继续，使用RFC2744或等效部分讨论的GSSAPI C绑定，通过在循环中调用gss\_init\_sec\_context()并将结果发送到服务器执行GSSAPI初始化，从空输入开始然后从服务器返回每个结果，直到返回无输出。在向服务器发送gss\_init\_sec\_context()的结果时，按照网络字节顺序，将消息的长度预置为四个字节的整数。如果成功，则使用gss\_wrap()加密通常的StartupMessage和所有后续数据，将gss\_wrap()的结果长度作为网络字节顺序中的四个字节整数预置到实际加密的有效负载。请注意，服务器仅接受来自小于16kB的客户端的加密数据包；gss\_wrap\_size\_limit()应由客户端用于确定将适合此限制的未加密消息的大小，并且较大的消息将分解为多个gss\_wrap()调用。典型段为8kB未加密数据，导致加密数据包略大

<sup>1</sup> [www.hs.net/lightdb/support/security/CVE-2021-23222/](http://www.hs.net/lightdb/support/security/CVE-2021-23222/)

于 8kB，但远低于 16kB 最大值。服务器可以预期不会向客户端发送大于 16kB 的加密数据包。要在N之后继续，请发送常用的StartupMessage，然后继续而不进行加密。

前端还应准备好处理来自服务器的 GSSAPI的ErrorMessage响应。仅在服务器将GSSAPI加密支持添加到LightDB之前，才会发生这种情况。在这种情况下连接必须关闭，但前端可能会选择打开新的连接，并在不请求 GSSAPI加密的情况下继续。考虑到前面指定的长度限制，ErrorMessage不能与来自服务器的具有适当长度的正常响应相混淆。

当可以执行 GSSAPI 加密时，服务器预计只发送单个 G 字节，然后等待前端启动GSSAPI 握手。如果此时有其他字节可读，则很可能意味着中间人正在尝试执行缓冲区填充攻击（CVE-2021-23222<sup>2</sup>）。前端应该编写代码，要么在将套接字交给 GSSAPI 库之前从套接字中精确读取一个字节，要么在发现已读取额外字节时将其视为协议违规行为。

初始 GSSAPIRequest也可用于正在打开的连接中发送CancelRequest消息。

一旦成功建立 GSSAPI 加密，使用 gss\_wrap() 对通常的 StartupMessage 和所有后续数据进行加密，将 gss\_wrap() 函数的结果长度作为四字节整数（网络字节顺序）添加到实际的加密有效载荷前缀。请注意，服务器只接受来自客户端的小于 16kB 的加密数据包；客户端应该使用 gss\_wrap\_size\_limit() 函数确定适合在此限制内的未加密消息的大小，较大的消息应该分成多个 gss\_wrap() 调用。典型的段是 8kB 的未加密数据，导致略大于 8kB 但完全在 16kB 最大限制内的加密数据包。可以期望服务器不会向客户端发送大于 16kB 的加密数据包。

虽然协议本身不为服务器提供强制GSSAPI加密的方法，但管理员可以将服务器配置为拒绝未加密的会话，作为身份验证检查的副产品。

## 49.3. SASL认证

SASL是一种针对面向连接协议的认证框架。当前，LightDB实现了两种SASL认证机制，即SCRAM-SHA-256和SCRAM-SHA-256-PLUS。未来可能会增加更多。下面的步骤展示了通常SASL认证是如何执行的，而接下来的小节会给出更多有关SCRAM-SHA-256和SCRAM-SHA-256-PLUS的细节。

### SASL认证消息流

1. 为了开始一次SASL认证交换，服务器发送一个AuthenticationSASL消息。它包括一个服务器能够接受的SASL认证机制的列表，这个列表按照服务器喜欢的顺序组织。
2. 客户端从该列表中选取一种受支持的机制，并且发送一个SASLInitialResponse消息给服务器。该消息包括选中的机制名称以及一个可选的初始客户端响应（如果选中的机制要用到）。
3. 接下来将是一个或者多个服务器挑战和客户端响应消息。每一次的服务器挑战都通过一个AuthenticationSASLContinue消息发送，后面会跟着客户端通过SASLResponse消息发送的响应。这些消息的细节与机制有关。
4. 最后，当认证交换成功完成，服务器发送一个AuthenticationSASLFinal消息，后面马上跟上一个AuthenticationOk消息。AuthenticationSASLFinal包含服务器发给客户端的额外数据，其内容与选中的认证机制有关。如果认证机制不使用在完成时发送的额外数据，则不发送AuthenticationSASLFinal消息。

在出错时，服务器可以在任何阶段中止认证，并且发送一个ErrorMessage。

### 49.3.1. SCRAM-SHA-256认证

目前实现的SASL机制是SCRAM-SHA-256及其带有通道绑定的变体SCRAM-SHA-256-PLUS。它们在RFC 7677和RFC 5802中有详细描述。

<sup>2</sup> [www.hs.net/lightdb/support/security/CVE-2021-23222/](http://www.hs.net/lightdb/support/security/CVE-2021-23222/)

当LightDB使用SCRAM-SHA-256时，服务器将忽略客户端在client-first-message中发送的用户名，而是使用在启动消息中已经发过来的用户名。LightDB支持多字符编码，然而SCRAM要求用户名使用UTF-8，因此可能无法表示LightDB中的UTF-8用户名。

SCRAM规范要求口令也是UTF-8，并且要用SASLprep算法对口令进行处理。但是，LightDB不要求对口令使用UTF-8。在设置一个用户的口令时，会把口令当作是UTF-8一样用SASLprep处理，而不管实际使用的编码是什么。不过，如果口令不是一种合法的UTF-8字节序列或者含有SASLprep算法禁止的UTF-8字节序列，将直接使用未被SASLprep处理过的原始口令，而不是抛出错误。这使得不是UTF-8的口令能够被规范化，但是也允许使用非UTF-8的口令，并且不要求系统了解口令到底是什么编码。

在编译了SSL支持的LightDB中支持通道绑定。带有通道绑定的SCRAM的SASL机制名是SCRAM-SHA-256-PLUS。被LightDB使用的通道绑定类型是tls-server-end-point。

在没有通道绑定的SCRAM中，服务器选择一个随机数，它被传送给客户端，客户端会把这个随机数与用户提供的口令混合在被传输的口令哈希中。虽然这能够防止在后面的会话中成功地重新传送这个口令哈希，但是却无法防止位于真实服务器和客户端之间的伪服务器经手服务器的随机值并且认证成功。

带有通道绑定的SCRAM通过将服务器证书的签名混入被传输的口令哈希来防止这类中间人攻击。虽然伪服务器能够重新传输真实服务器的证书，但是它无法访问匹配该证书的私钥，因此无法证明它是其拥有者，进而导致SSL连接失败。

## 示例

1. 服务器发送一个AuthenticationSASL消息。它包括一个服务器能够接受的SASL认证机制列表。如果服务器编译有SSL支持，这个列表将会是SCRAM-SHA-256-PLUS和SCRAM-SHA-256，否则列表中只会有后者。
2. 客户端通过发送一个SASLInitialResponse消息进行响应，该消息指出选中的机制，即SCRAM-SHA-256或者SCRAM-SHA-256-PLUS（客户端可以自由地选择机制，但是为了更安全在可以支持通道绑定变体时应该选择它）。在Initial Client响应字段中，该消息包含SCRAM的client-first-message。client-first-message还包含客户端选中的通道绑定类型。
3. 服务器发送一个AuthenticationSASLContinue消息，其内容是一个SCRAM的server-first message。
4. 客户端发送一个SASLResponse消息，其内容是SCRAM的client-final-message。
5. 服务器发送一个AuthenticationSASLFinal消息，其中是SCRAM的server-final-message，后面紧接着一个AuthenticationOk消息。

## 49.4. 流复制协议

要启动流复制，前端在启动消息中发送replication参数。布尔值true（或者on、yes、1）告诉后端进入到物理复制的walsender模式，在其中可以发出一个小型的复制命令集合（见下文）而不是普通的SQL命令。

将database作为replication参数的值传递会指示后端进入到逻辑复制的walsender模式，连接到dbname参数指定的数据库。在逻辑复制的walsender模式中，可以发出下文所示的复制命令以及普通的SQL命令。

在物理复制或者逻辑复制的walsender模式中，只能使用简单查询协议。

出于测试复制命令的目的，你可以通过ltsql或者任何使用libpq的工具使用包含replication选项的连接字符串建立一个复制连接，例如：

```
ltsql "dbname=postgres replication=database" -c "IDENTIFY_SYSTEM;"
```

不过更常用的是：`lt_receivewal`（对于物理复制）或者`lt_recvlogical`（对于逻辑复制）。

当`log_replication_commands`被启用时，复制命令会被记录在服务器日志中。

在复制模式中可以接受的命令有：

`IDENTIFY_SYSTEM`

请求服务器标识它自己。服务器以一个行构成的结果集作为答复，其中包含四个域：

`systemid` (text)

标识该集簇的唯一的系统标识符。这可以被用来检查用于初始化后备机的基础备份是否来自于同一个集簇。

`timeline` (int4)

当前的时间线 ID。也对于检查后备机是否与主控机一致有用。

`xlogpos` (text)

当前的WAL刷写位置。用于得到一个在预写式日志中的已知位置作为流的开始位置。

`dbname` (text)

要连接到的数据库或者空。

`SHOW name`

请求服务器发送一个运行时参数的当前设置。这类似于SQL命令`SHOW`。

`name`

运行时参数的名称。可用的参数在Chapter 1中。

`TIMELINE_HISTORY tli`

请求服务器将时间线`tli`的历史文件发送过来。服务器将以一行组成的结果集作为答复，其中包含两个域：

`filename` (text)

时间线历史文件的文件名，例如`00000002.history`。

`content` (bytea)

时间线历史文件的内容。

```
CREATE_REPLICATION_SLOT slot_name [ TEMPORARY ] { PHYSICAL [ RESERVE_WAL ] | LOGICAL
output_plugin [ EXPORT_SNAPSHOT | NOEXPORT_SNAPSHOT | USE_SNAPSHOT ] }
```

创建一个物理的或者逻辑的复制槽。更多关于复制槽的内容请见 Section 24.2.6

`slot_name`

要创建的槽的名称。必须是一个合法的复制槽名称（见 Section 24.2.6.1）。

`output_plugin`

被用于逻辑解码的输出插件的名称（见 Section 45.6）。

## TEMPORARY

指定这个复制槽是一个临时复制槽。临时复制槽不会被保存在磁盘上并且在发生错误或者会话结束时会被自动删除。

## RESERVE\_WAL

指定这个物理复制槽立即保留WAL。否则，只有来自流复制客户端的连接上才会保留WAL。

EXPORT\_SNAPSHOT  
NOEXPORT\_SNAPSHOT  
USE\_SNAPSHOT

决定如何处理在逻辑槽初始化期间取到的快照。默认值EXPORT\_SNAPSHOT将把该快照导出以备在其他会话中使用。这个选项不能在事务外面使用。USE\_SNAPSHOT将把该快照用于正在执行命令的当前事务。这个选项必须被用于一个事务中，并且CREATE\_REPLICATION\_SLOT必须是该事务中运行的第一个命令。最后，NOEXPORT\_SNAPSHOT将按照正常把该快照用于逻辑解码，但是不会对它做其他事情。

作为对这个命令的响应，服务器将发送一个一行的结果集，其中包含下列字段：

slot\_name (text)

新创建的复制槽的名称。

consistent\_point (text)

该槽达到一致的WAL位置。这是在该复制槽上可以开始流的最早位置。

snapshot\_name (text)

被这个命令导出的快照的标识符。该快照一直是有效的，直到在这个连接中执行了新的命令或者该复制连接被关闭。如果被创建的槽是物理槽，则为空。

output\_plugin (text)

新创建的复制槽使用的输出插件的名称。如果被创建的槽是物理槽，则为空。

START\_REPLICATION [ SLOT slot\_name ] [ PHYSICAL ] XXX/XXX [ TIMELINE tli ]

指示服务器开始启动流WAL，从 WAL 位置XXX/XXX开始。如果TIMELINE选项被指定，流传送会在时间线tli上开始，否则会选择服务器的当前时间线。服务器可以回复一个错误，例如如果被请求的WAL节已经被回收了。如果成功，服务器将会响应一个CopyBothResponse消息，并且然后开始以流的方式把WAL传送给前端。

如果通过slot\_name提供了一个槽的名称，它将被更新复制进度，这样该服务器知道哪些WAL 段以及哪些事务（如果hot\_standby\_feedback为打开）仍然被后备机所需要。

如果客户端请求一个并非最新的时间线，但是属于服务器历史的一部分，服务器将会把该时间线上从请求点开始的所有WAL以流式传送，一直到服务器切换到另外一个时间线的点。如果客户端请求在一个老的时间线末尾进行流传送，服务器将在不进入COPY模式的情况下立即响应CommandComplete。

在流传送完一个非最新时间线上所有的WAL之后，服务器将会通过退出COPY模式来结束流。当客户端认识到这一点并也退出COPY模式时，服务器会发送一个包含一行两列的结果集，以指示在该服务器历史中的下一个时间线。第一列是下一个时间线的 ID（类型int8），而第二列是发生切换的 WAL 位置（类型text）。通常，切换位置是被流传送的WAL的末尾，但是在很少的情况下服务器会从旧的时间线中发送一些WAL，而该时间线是服务器本身在提示之前还没有重放的。最后，服务器发送两个 CommandComplete 消息（一个结束 CopyData，另一个结束START\_REPLICATION本身），并准备接受新命令。



WAL数据以一系列CopyData消息的形式被发送（这允许其他信息穿插其中，特别是服务器可以在开始流传送后遇到失败时发送一个ErrorResponse消息）。每个从服务器到客户端的CopyData消息承载了一个下列格式之一的消息：

XLogData (B)

Byte1('w')

标识该消息是WAL数据。

Int64

在消息中WAL数据的起始点。

Int64

服务器上WAL的当前终点。

Int64

在传送时服务器的系统时钟，以从2000-01-01午夜开始的微妙计。

Byten

WAL数据流的一节。

一个WAL记录绝不会被分割到两个XLogData消息。如果一个WAL记录跨越了一个WAL页面的边界，并且因此已经被使用连续的记录分割，它可以在页面边界被分割。换句话说，第一个主要WAL记录和它的后续记录可以在不同的XLogData消息中被发送。

主要存活消息 (B)

Byte1('k')

标识该消息是一个发送者存活消息。

Int64

服务器上WAL的当前终点。

Int64

在传送时服务器的系统时钟，以从2000-01-01午夜开始的微妙计。

Byte1

1表示客户端应该尽快回复该消息，以避免连接超时。否则为0。

接收进程可以在任何时候给发送者发送回复，回复可以使用下列消息格式之一（也在CopyData消息中使用）：

后备机状态更新 (F)

Byte1('r')

标识该消息是一个接收者状态更新。

Int64

接收到并且写入到后备机磁盘的最后一个WAL比特的位置+1。

Int64

被刷入到后备机磁盘的最后一个WAL比特的位置+1。

Int64

被应用在后备机上的最后一个WAL比特的位置+1。

Int64

在传送时客户端的系统时钟，以从2000-01-01午夜开始的微妙计。

Byte1

如果为1，客户端要求服务器马上回复这个消息。这可以被用来ping服务器以测试连接是否仍然完好。

热备机反馈消息 (F)

Byte1('h')

标识该消息是一个热备机反馈消息。

Int64

在传送时客户端的系统时钟，以从2000-01-01午夜开始的微妙计。

Int32

后备机的当前全局xmin，排除来自任何复制槽的catalog\_xmin。如果这个值和接下来的catalog\_xmin都为0，这会被当作一个通知表示在这个连接上将不会再发送热备反馈。后来的非0消息将重新启动反馈机制。

Int32

后备机上全局xmin事务ID的世代。

Int32

后备机上任意复制槽中的最小catalog\_xmin。如果在后备机上不存在catalog\_xmin或者热备反馈被禁用，这个值被设置为0。

Int32

后备机上全局catalog\_xmin事务ID的世代。

```
START_REPLICATION SLOT slot_name LOGICAL XXX/XXX [ ( option_name [ option_value ] [, ...] ) ]
```

指示服务器为逻辑复制开始流式传送 WAL，从 WAL 位置XXX/XXX开始。服务器可以回复一个错误，例如如果请求的 WAL 小节已经回环。如果成功，服务器会响应一个CopyBothResponse 消息，并且接着开始流式传送 WAL 给前端。

消息内部的消息与START\_REPLICATION ... PHYSICAL中记录的格式相同，包括两个CommandComplete 消息。

与选中槽关联的输出插件被用来处理流的输出。

```
SLOT slot_name
```

要从哪个槽流式传送改变。这个参数是必须的，并且必须对应于一个现有的用LOGICAL模式的CREATE\_REPLICATION\_SLOT创建的逻辑复制槽。

```
XXX/XXX
```

要开始流式传送的 WAL 位置。

option\_name

一个传递给该槽的逻辑解码插件的选项的名称。

option\_value

字符串常量形式的选项值，与前面指定的选项关联。

DROP\_REPLICATION\_SLOT slot\_name [ WAIT ]

删除一个复制槽，释放任何保留的服务器端资源。如果该槽是一个在不同于walsender所连接的数据库中创建的逻辑槽，这个命令会失败。

slot\_name

要删除的槽的名称。

WAIT

如果槽正处于活跃状态，这个选项会导致命令等待，直到槽变得不活跃，而不是像默认行为那样直接报错。

BASE\_BACKUP [ LABEL 'label' ] [ PROGRESS ] [ FAST ] [ WAL ] [ NOWAIT ] [ MAX\_RATE rate ] [ TABLESPACE\_MAP ] [ NOVERIFY\_CHECKSUMS ] [ MANIFEST manifest\_option ] [ MANIFEST\_CHECKSUMS checksum\_algorithm ]

指示服务器开始流传送一个基础备份。在备份开始之前系统将自动被置于备份模式，而在备份结束时会自动被退出备份模式。可以接受下列选项：

LABEL 'label'

设置备份的标签。如果没有指定，将会使用base backup作为标签。标签的引号规则和standard\_conforming\_strings开启时标准SQL字符串的一样。

PROGRESS

请求用以生成一个进度报告的信息。这将送回位于每个表空间头部的一个近似大小，它可以被用于计算流还有多久才能被完成。它通过在传输开始之前枚举所有文件大小来计算，并且可能会对性能产生一种负面影响 -- 特别情况下它可能会在流传送第一个数据之前就耗费很长时间。因为数据库文件可能在备份期间改变，这个大小只是近似的并且可能在近似计算和发送真正的文件之间增长或者收缩。

FAST

请求一个快速检查点。

WAL

在备份中包含必需的WAL段。这将把开始和停止备份之间的所有文件包括在base目录tar文件中的1t\_wal目录中。

NOWAIT

默认情况下，备份会等待直到最后一个要求的 WAL 段被归档，或者当日至归档被禁用时发出一个警告。指定NOWAIT会禁用等待和警告，而让客户端负责确保所要求的日志是可用的。

MAX\_RATE rate

单位时间内从服务器传输到客户端的最大数据量限制。期望的单位是千字节每秒。如果指定了这个选项，值必须等于零或者位于 32 kB到 1 GB（包括）范围之间。如果 0 被传入或者没有指定该选项，对于传输将没有限制。

## TABLESPACE\_MAP

在名为tablespace\_map的文件中包括有关lt\_tblspc目录中存在的符号链接的信息。这个表空间映射文件包括了在目录lt\_tblspc中存在的每一个符号链接的名字以及它的完整路径。

## NOVERIFY\_CHECKSUMS

默认情况下，如果启用了校验码，在基础备份期间会验证校验码。指定NOVERIFY\_CHECKSUMS可以禁用这种验证。

## MANIFEST manifest\_option

当此选项指定为yes或force-encode时，将创建备份清单并与备份一起发送。清单是备份中存在的每个文件的列表，但可能包含的任何 WAL 文件除外。它还存储每个文件的大小、上次修改时间和可选的校验和。force-encode的值强制所有文件名进行十六进制编码；否则，仅对名称为非 UTF8 八位字节序列的文件执行这种类型的编码。force-encode主要用于测试目的，以确保读取备份清单的客户端可以处理这种情况。为了与以前的版本兼容，默认值为MANIFEST 'no'。

## MANIFEST\_CHECKSUMS checksum\_algorithm

指定应用于备份清单中包含的每个文件的校验和算法。目前可用的算法有NONE、CRC32C、SHA224、SHA256、SHA384和SHA512。默认值为CRC32C。

当备份被启动，服务器将首先发送两个普通结果集，后面会跟着一个或多个CopyResponse结果。

第一个普通结果集在一行两列中包含了备份的起始位置。第一列包含使用XLogRecPtr格式给出的开始位置，第二列包含相应的时间线ID。

第二个普通结果集中为每一个表空间都有一行。行中的域有：

spcoid (oid)

表空间的 OID，如果是base目录则为空。

spclocation (text)

表空间目录的完整路径，如果是base目录则为空。

size (int8)

如果已请求进度报告，则表空间的大致大小，以千字节（1024 字节）为单位；否则为空。

在第二个普通结果集之后，一个或多个CopyResponse结果将被发送，一个用于主数据目录而对每一个除pg\_default和pg\_global之外的额外表空间也会有一个。CopyResponse结果中的数据将会使一个tar格式（遵循POSIX 1003.1-2008标准中指定的“ustar交换格式”）的表空间内容转储，不过标准中定义的两个拖尾全0块将被忽略。tar 数据完成后，如果请求了备份清单，则会发送另一个 CopyResponse 结果，其中包含当前基本备份的清单数据。在任何情况下，都会发送一个最终的普通结果集，其中包含备份的 WAL 结束位置，格式与起始位置相同。

用于数据目录和每个表空间的tar归档将包含目录中的所有文件，不管它们是否为LightDB文件或者是被加入的其他文件。唯一被排除的文件是：

- postmaster.pid
- lightdb.opts
- lt\_internal.init（在多个目录中都有）
- 在LightDB服务器操作过程中创建的各种临时文件和目录，例如何以pgsql\_tmp开头的文件或目录以及临时关系。
- 不做日志的关系，init分支除外，恢复时重建（空的）不做日志关系时需要它。

- `lt_wal`及其子目录。如果备份运行时要求包括WAL文件，一个`lt_wal`的合成版本将被包括进来，但是只会包含那些备份工作必需的文件，而不是包含剩下的内容。
- `lt_dynshmem`、`lt_notify`、`lt_replslot`、`lt_serial`、`lt_snapshots`、`lt_stat_tmp`以及 `lt_subtrans`会被拷贝为空目录（即使它们是符号链接）。
- 跳过除常规文件和目录之外的其他文件，例如符号链接（不同于上面所列出的目录）和特殊设备文件。（`lt_tblspc`中的符号链接会被保留）。

如果服务器上的底层文件系统支持，所有者、组合文件模式都会被设置。

## 49.5. 逻辑流复制协议

这一节介绍逻辑复制协议，它是一种以复制命令`START_REPLICATION` `SLOT` `slot_name` `LOGICAL`开始的消息流。

逻辑流复制协议建立在物理流复制协议的原始积累之上。

### 49.5.1. 逻辑流复制参数

逻辑复制命令`START_REPLICATION`接受下列参数：

`proto_version`

协议版本。当前支持版本1。

`publication_names`

要订阅（接收更改）的`publication`名称列表，用逗号分隔。每一个`publication`名称个体都被当作一个标准的对象名称，并且可以根据需要加上引号。

### 49.5.2. 逻辑复制协议消息

协议消息的个体在接下来的小节中讨论。个体的消息在Section 49.9中介绍。

所有的顶层协议消息都以一个消息类型字节开头。虽然被表示为字符代码，但这是一个没有相关编码的有符号字节。

由于流复制协议提供了一个消息长度，因此不需要顶层协议消息在其头部嵌入长度。

### 49.5.3. 逻辑复制协议的消息流

除`START_REPLICATION`命令和重放进度消息之外，所有信息流的方向都是从后端到前端。

逻辑复制协议会逐个发送事务个体。这意味着在一对`Begin`消息和`Commit`消息之间的所有消息都属于同一个事务。

每一个被发送的事务都包含零个或者多个DML消息（插入、更新、删除）。在级联设置的情况下，它还包括`Origin`消息。`Origin`消息表示该事务是在不同的复制节点上产生的。由于逻辑复制协议范围内的复制节点可以是任何东西，所以唯一的标识符是源头的名称。其下游的责任是根据需要处理这一信息（如果需要处理）。`Origin`消息总是在事务中任何DML消息之前被发送。

每个DML消息中都包含一个任意的关系ID，它可以被映射到`Relation`消息中的一个ID。`Relation`消息描述给定关系的模式。为一个给定的关系发送`Relation`消息的时机是：在当前会话中第一次为该关系发送DML消息，或者从上一次该关系的`Relation`消息以后该关系的定义发生改变。协议假定客户端有能力缓存够用的关系元数据。

关系消息通过它们的 `OID` 标识列类型。对于内置类型，假定客户端可以在本地查找该类型`OID`，因此不需要附加数据。对于非内置类型 `OID`，将在 `Relation` 消息之前发送 `Type` 消

息，以提供与该 OID 相关联的类型名称。因此，需要特别标识关系列类型的客户端应缓存 Type 消息的内容，并首先查询该缓存以查看类型 OID 是否在其中定义。如果没有，请在本地查找类型 OID。

## 49.6. 消息数据类型

本节描述消息里用到的基本数据类型。

Intn(i)

一个网络字节序（高位在前）的n位整数。如果指定了i，它就是将出现的准确值，否则该值就是一个变量。如 Int16、Int32(42)。

Intn[k]

一个k个n位整数的数组，每个都是以网络字节序表示的。数组长度k总是由消息前面的域来判断的。比如，Int16[M]。

String(s)

一个（C风格的）空值结束的字符串。对字符串没有特别的长度限制。如果指定了s，那么它是将出现的的确切的值，否则这个值就是一个变量。比如，String, String("user")。

### Note

后端能返回的字符串的长度没有预定义的限制。所以对前端比较好的编码策略是使用某种可扩展的缓冲区，以便能接受任何能放进内存里的东西。如果那样做不可行，则读取完整的字符串然后抛弃不能放进固定大小缓冲区的尾部字符。

Byten(c)

精确的n字节。如果域宽度n不是一个常量，那么我们总是可以从消息中更早的域中判断它。如果指定了c，那么它就是确切数值。例如，Byte2, Byte1('n')。

## 49.7. 消息格式

本节描述各种消息的详细格式。每种消息都标记来指示它是由前端（F）、后端（B）或者两者（F & B）发送的。请注意，尽管每条消息在开头都包含一个字节计数，但是消息格式也被定义为无需参考字节计数就可以找到消息的结尾。这样是为了有效性检查（CopyData消息是一个例外，因为它形成一个数据流的一部分；任意独立的CopyData消息可能是无法自解释的）。

AuthenticationOk (B)

Byte1('R')

标识该消息是一条认证请求。

Int32(8)

以字节计的消息内容长度，包括这个长度本身。

Int32(0)

指定该认证是成功的。

AuthenticationKerberosV5 (B)

Byte1('R')

标识该消息是一条认证请求。

Int32(8)

以字节计的消息内容长度，包括长度自身。

Int32(2)

指定要求Kerberos V5认证。

AuthenticationCleartextPassword (B)

Byte1('R')

标识该消息是一条认证请求。

Int32(8)

以字节计的消息内容长度，包括长度自身。

Int32(3)

指定要求一个明文的口令。

AuthenticationMD5Password (B)

Byte1('R')

标识这条消息是一个认证请求。

Int32(12)

以字节计的消息内容的长度，包括长度本身。

Int32(5)

指定要求一个MD5加密的口令。

Byte4

加密口令的时候使用的盐粒。

AuthenticationSCMCredential (B)

Byte1('R')

标识这条消息是一个认证请求。

Int32(8)

以字节计的消息内容长度，包括长度本身。

Int32(6)

指定请求一个SCM信任消息。

AuthenticationGSS (B)

Byte1('R')

标识该消息是一个认证请求。

Int32(8)

消息内容的长度以字节为单位，包括自己。

Int32(7)

指定被请求的是GSSAPI认证。

AuthenticationSSPI (B)

Byte1('R')

标识该消息是一个认证请求。

Int32(8)

消息内容的长度以字节为单位，包括自己。

Int32(9)

指定被请求的是SSPI认证。

AuthenticationGSSContinue (B)

Byte1('R')

标识该消息是一个认证请求。

Int32

消息内容的长度以字节为单位，包括自己。

Int32(8)

指定该消息包含GSSAPI或SSPI数据。

Byten

GSSAPI或SSPI认证数据。

AuthenticationSASL (B)

Byte1('R')

标识该消息是一个认证请求。

Int32

消息内容的长度，以字节为单位，包括其自身。

Int32(10)

指定SASL认证被要求。

消息体是一个SASL认证机制的列表，该列表按照服务器偏爱的顺序组织。在最后一个认证机制名称的后面需要一个零字节作为终结符。对于每一种机制，有下列信息：

String

一种SASL认证机制的名称。

AuthenticationSASLContinue (B)

Byte1('R')

标识该消息是一个认证请求。



Int32

消息内容的长度，以字节为单位，包括其自身。

Int32(11)

指定这个消息包含一个SASL挑战。

Byten

SASL数据，与使用的SASL机制有关。

AuthenticationSASLFinal (B)

Byte1('R')

标识该消息是一个认证请求。

Int32

消息内容的长度，以字节为单位，包括其自身。

Int32(12)

指定SASL认证已经完成。

Byten

SASL产出的“额外数据”，与使用的SASL机制有关。

BackendKeyData (B)

Byte1('K')

标识该消息是一个取消键数据。如果前端希望能够在稍后发出CancelRequest消息，那么它必须保存这个值。

Int32(12)

以字节计的消息内容的长度，包括长度本身。

Int32

后端的进程号（PID）。

Int32

此后端的密钥。

Bind (F)

Byte1('B')

标识该信息是一个绑定命令。

Int32

以字节计的消息内容的长度，包括长度本身。

String

目标入口的名字（空字符串则选取未命名的入口）。

String

源预备语句的名字（空字符串则选取未命名的预备语句）。

Int16

后面跟着的参数格式代码的数目（由下文的C说明）。这个数值可以是零，表示没有参数，或者是参数都使用缺省格式（文本）；也可以是一，这种情况下指定的格式代码被应用于所有参数；或者它可以等于实际参数的数目。

Int16[C]

参数格式代码。目前每个都必须是零（文本）或者一（二进制）。

Int16

后面跟着的参数值的数目（可能为零）。这些必须和查询需要的参数个数匹配。

然后，每个参数都会出现下面的域对：

Int32

参数值的长度，以字节计（这个长度并不包含长度本身）。可以为零。一个特殊的情况是，-1 表示一个NULL参数值。在NULL 的情况下，后面不会跟着字节值。

Byten

参数值，使用关联的格式代码表示的格式。n是上文的长度。

在最后一个参数之后，出现下面的域：

Int16

后面跟着的结果列格式代码数目（下文的R描述）。这个数目可以是零表示没有结果列或者结果列都使用缺省格式（文本）；也可以是一，这种情况下指定的格式代码被应用于所有结果列（如果有的话）；或者它可以等于查询的结果列的实际数目。

Int16[R]

结果列格式代码。目前每个必须是零（文本）或者一（二进制）。

BindComplete (B)

Byte1('2')

标识该消息为一个绑定结束标识符。

Int32(4)

以字节计的消息长度，包括长度本身。

CancelRequest (F)

Int32(16)

以字节计的消息长度。包括长度本身。

Int32(80877102)

取消请求代码。该值被选中在高16位包含1234，并且在低16位包含 5678（为避免混淆，这个代码不能和任何协议版本号相同）。

Int32

目标后端的进程号 (PID)。

Int32

目标后端的密钥。

Close (F)

Byte1('C')

标识这条消息是一个Close命令。

Int32

以字节计的消息内容长度，包括长度本身。

Byte1

'S' 关闭一个准备的语句；或者'P' 关闭一个入口。

String

一个要关闭的预备语句或者入口的名字（一个空字符串选择未命名的预备语句或者入口）。

CloseComplete (B)

Byte1('3')

标识消息是一个Close完成指示器。

Int32(4)

以字节计的消息内容的长度，包括长度本身。

CommandComplete (B)

Byte1('C')

标识此消息是一个命令结束响应。

Int32

以字节计的消息内容的长度，包括长度本身。

String

命令标记。它通常是一个单字，标识被完成的SQL命令。

对于INSERT命令，该标记是INSERT oid rows，其中rows是已被插入的行数。如果rows为1并且目标表已有OIDs，oid为已插入行的对象ID；但是OIDs系统列已经不再被支持；因此oid始终为0。

对于DELETE命令，该标记是DELETE rows，其中rows是已被删除的行数。

对于UPDATE命令，该标记是UPDATE rows，其中rows是已被更新的行数。

对于SELECT或CREATE TABLE AS命令，该标记是SELECT rows，其中rows是被检索的行数。

对于MOVE命令，该标记是MOVE rows，其中rows是游标位置被移动的行数。

对于FETCH命令，该标记是FETCH rows，其中rows是已从游标中检索出来的行数。

对于COPY命令，该标记是COPY rows，其中rows是已拷贝的行数（注意，行计数只在LightDB 8.2及其后的版本中出现）。

#### CopyData (F & B)

Byte1('d')

标识这条消息是一个COPY数据。

Int32

以字节计的消息内容的长度，包括长度本身。

Byten

构成COPY数据流的一部分的数据。从后端发出的消息总是对应单一的数据行，但是前端发出的消息可能会任意分割数据流。

#### CopyDone (F & B)

Byte1('c')

标识这条信息是一个COPY结束指示器。

Int32(4)

以字节计的消息内容长度，包括长度本身。

#### CopyFail (F)

Byte1('f')

标识这条消息是一个COPY失败指示器。

Int32

以字节计的消息内容的长度，包括长度本身。

String

一个报告失败原因的错误消息。

#### CopyInResponse (B)

Byte1('G')

标识这条消息是一条Start Copy In（开始拷贝入）响应消息。前端现在必须发送拷贝入数据（如果还没准备好做这些事情，那么发送一条CopyFail消息）。

Int32

以字节计的消息内容的长度，包括长度本身。

Int8

0表示全体拷贝格式都是文本（数据行由新符分隔，列由分隔字符分隔等等）。1表示全体拷贝格式都是二进制的（类似于DataRow格式）。参阅COPY获取更多信息。

Int16

要拷贝的数据中的列数（由下文的N解释）。

Int16[N]

每个列要使用的格式代码。目前每个都必须是零（文本）或者一（二进制）。如果全体拷贝格式都是文本，那么所有的都必须是零。

CopyOutResponse (B)

Byte1('H')

标识这条消息是一条Start Copy Out（开始拷贝出）响应消息。这条消息后面将跟着拷贝出数据。

Int32

以字节计的消息内容的长度，包括它自己。

Int8

0表示全体拷贝格式都是文本（数据行由新符分隔，列由分隔字符分隔等等）。1表示全体拷贝格式都是二进制的（类似于DataRow格式）。参阅COPY获取更多信息。

Int16

要拷贝的数据的列数（在下文的N说明）。

Int16[N]

每个列要使用的格式代码。目前每个都必须是零（文本）或者一（二进制）。如果全体拷贝格式都是文本，那么所有的都必须是零。

CopyBothResponse (B)

Byte1('W')

标识这个消息是一个Start Copy Both（启动双向复制）响应。这个消息只用于流复制。

Int32

以字节计的消息内容的长度，包括长度自身。

Int8

0表示全体COPY格式都是文本（数据行由新符分隔，列由分隔字符分隔等等）。1表示全体拷贝格式都是二进制的（类似于DataRow格式）。参阅COPY获取更多信息。

Int16

要拷贝的数据中的列数目（在下文的N说明）。

Int16[N]

每个列要使用的格式代码。目前每个都必须是零（文本）或者一（二进制）。如果全体拷贝格式都是文本，那么所有的代码都必须是零。

DataRow (B)

Byte1('D')

标识这个消息是一个数据行。

Int32

以字节计的消息内容的长度，包括长度自身。

Int16

后面跟着的列值的个数（可能是零）。

然后，为每个列都会出现下面的域对：

Int32

列值的长度，以字节计（这个长度不包括它自己）。可以为零。一个特殊的情况是，-1表示一个NULL的域值。如果是NULL的情况则后面不会跟着值字节。

Byten

一个列的数值，以相关的格式代码指示的格式展现。n是上文的长度。

Describe (F)

Byte1('D')

标识该消息是一个Describe（描述）命令。

Int32

以字节计的消息内容的长度，包括字节本身。

Byte1

'S' 描述一个预备语句；或者 'P' 描述一个入口。

String

要描述的预备语句或者入口的名字（或者一个空字符串，就会选取未命名的预备语句或者入口）。

EmptyQueryResponse (B)

Byte1('I')

标识这条消息是对一个空查询字符串的响应（这个消息替换了CommandComplete）。

Int32(4)

以字节计的消息内容长度，包括它自己。

ErrorResponse (B)

Byte1('E')

标识该消息是一条错误。

Int32

以字节计的消息内容的长度，包括长度本身。

消息体由一个或多个标识域组成，后面跟着一个零字节作为终止符。域可以以任何顺序出现。对于每个域都有下面的东西：

Byte1

一个标识域类型的代码；如果为零，这就是消息终止符并且不会跟着有字符串。目前定义的域类型在Section 49. 列出。由于将来可能增加更多的域类型，所以前端应该默默地忽略未识别类型的域。

String

域值。

Execute (F)

Byte1('E')

标识该消息是一个Execute命令。

Int32

以字节计的消息内容的长度，包括长度自身。

String

要执行的入口的名字（空字符串选择未命名的入口）。

Int32

要返回的最大行数，如果入口包含返回行的查询（否则忽略）。零表示“无限制”。

Flush (F)

Byte1('H')

标识该消息是一条Flush命令。

Int32(4)

以字节计的消息内容的长度，包括长度本身。

FunctionCall (F)

Byte1('F')

标识该消息是一个函数调用。

Int32

以字节计的消息内容的长度，包括长度本身。

Int32

指定要调用的函数的对象ID (OID)。

Int16

后面跟着的参数格式代码的数目（用下文的C表示）。它可以是零，表示没有参数，或者是所有参数都使用缺省格式（文本）；也可以是一，这种情况下声明的格式代码被应用于所有参数；或者它可以等于参数的实际个数。

Int16[C]

参数格式代码。目前每个必须是零（文本）或者一（二进制）

Int16

指定提供给函数的参数个数。

然后，对每个参数都出现下面域对：

Int32

以字节计的参数值的长度（不包括长度本身）。可以为零。一个特殊的例子是，-1表示一个NULL参数值。如果是NULL，则没有参数字节跟在后面。

Byten

参数的值，格式由相关的格式代码指示。n是上文的长度。

在最后一个参数之后，出现下面的域：

Int16

函数结果的格式代码。目前必须是零（文本）或者一（二进制）。

FunctionCallResponse (B)

Byte1('V')

标识这条消息是一个函数调用结果。

Int32

以字节计的消息内容长度，包括长度本身。

Int32

以字节计的函数结果值的长度（不包括长度本身）。可以为零。一个特殊的情况是，-1表示NULL函数结果。如果是NULL则后面没有值字节跟随。

Byten

函数结果的值，格式由相关联的格式代码指示。n是上文的长度。

GSSResponse (F)

Byte1('p')

标识该消息是一个GSSAPI或SSPI响应。注意这也被用于SASL和口令响应消息。准确的消息类型可以从上下文中得出。

Int32

消息内容的长度，以字节为单位，包括其自身。

Byten

GSSAPI/SSPI相关的消息数据。

NegotiateProtocolVersion (B)

Byte1('v')

标识该消息是一个协议版本协商消息。

Int32

消息内容的长度，以字节为单位，包括其自身。

Int32

对于客户端请求的主协议版本，服务器能支持的最新的次协议版本。



Int32

服务器无法识别的协议选项数目。

然后，对于服务器无法识别的协议选项，还有下列信息：

String

选项名称。

NoData (B)

Byte1('n')

标识这条消息是一个无数据指示器。

Int32(4)

以字节计的消息内容长度，包括长度本身。

NoticeResponse (B)

Byte1('N')

标识这条消息是一个通知。

Int32

以字节计的消息内容长度，包括长度本身。

消息体由一个或多个标识域组成，后面跟着零字节作为中止符。域可以以任何顺序出现。对于每个域，都有下面的东西：

Byte1

一个标识域类型的代码；如果为零，那么它就是消息终止符，并且后面不会跟着字符串。目前定义的域类型在Section 49.8列出。由于将来可能会增加更多域类型，所以前端应该将不能识别的域安静地忽略掉。

String

域值。

NotificationResponse (B)

Byte1('A')

标识这条消息是一个通知响应。

Int32

以字节计地消息内容地长度，包括长度本身。

Int32

通知后端进程的进程ID。

String

通知被抛出的通道的名字。

String

从通知进程传递过来的“载荷”字符串。

ParameterDescription (B)

Byte1('t')

标识该消息是一个参数描述。

Int32

以字节计的消息内容长度，包括长度本身。

Int16

语句所使用的参数的个数（可以为零）。

然后，对每个参数，有下面的东西：

Int32

指定参数数据类型的对象ID。

ParameterStatus (B)

Byte1('S')

标识这条消息是一个运行时参数的状态报告。

Int32

以字节计的消息内容的长度，包括长度自身。

String

被报告的运行时参数的名字。

String

参数的当前值。

Parse (F)

Byte1('P')

标识该消息是一条Parse命令。

Int32

以字节计的消息内容的长度，包括长度自身。

String

目的预备语句的名字（空字符串选取未命名的预备语句）。

String

要分析的查询字符串。

Int16

指定的参数数据类型的数目（可以为零）。请注意这个参数并不表示可能在查询字符串里出现的参数个数，只是前端希望预先为其指定类型的参数数目。

然后，对每个参数，有下面的东西：

Int32

指定参数数据类型的对象ID。这里为零等效于不指定该类型。

ParseComplete (B)

Byte1('1')

标识该消息是一个Parse完成指示器。

Int32(4)

以字节计的消息内容长度，包括长度自身。

PasswordMessage (F)

Byte1('p')

标识该消息是一个口令响应。注意这也被用于GSSAPI、SSPI以及SASL响应消息。确切的消息类型可以从上下文推出。

Int32

以字节计的消息内容的长度，包括长度本身。

String

口令（如果要求了，就是加密后的）。

PortalSuspended (B)

Byte1('s')

标识这条消息是一个入口暂停指示器。请注意这个消息只出现在达到一条Execute消息的行计数限制的时候。

Int32(4)

以字节计的消息内容的长度，包括长度自身。

Query (F)

Byte1('Q')

标识该消息是一个简单查询。

Int32

以字节计的消息内容的长度，包括长度自身。

String

查询字符串自身。

ReadyForQuery (B)

Byte1('Z')

标识消息的类型。在后端为新的查询周期准备好的时候，总会发送ReadyForQuery。

Int32(5)

以字节计的消息内容的长度，包括长度本身。

Byte1

当前后端事务状态指示器。可能的值是空闲状况下的'I'（不在事务块里）；在事务块里是'T'；或者在一个失败的事务块里是'E'（在事务块结束之前，任何查询都将被拒绝）。

RowDescription (B)

Byte1('T')

标识该消息是一个行描述。

Int32

以字节计的消息内容的长度，包括长度自身。

Int16

指定在一个行里面的域的数目（可以为零）。

然后对于每个字段，有下面的东西：

String

字段名字。

Int32

如果域可以被标识为一个指定表的列，这里就是表的对象ID；否则就是零。

Int16

如果该域可以被标识为一个指定表的列，这里就是该列的属性号；否则就是零。

Int32

域数据类型的对象ID。

Int16

数据类型尺寸（参阅pg\_type.typelen）。请注意负值表示变宽类型。

Int32

类型修饰词（参阅pg\_attribute.atttypmod）。修饰词的含义是类型相关的。

Int16

用于该域的格式码。目前会是零（文本）或者一（二进制）。在Describe语句的变体返回的RowDescription里，格式码还是未知的，因此总是零。

SASLInitialResponse (F)

Byte1('p')

标识该消息是一个初始SASL响应。注意这也被用于GSSAPI、SSPI以及口令响应消息。确切的消息类型可以从上下文推知。

Int32

消息内容的长度，以字节为单位，包括其自身。

String

客户端选择的SASL认证机制的名称。

Int32

后面跟着的SASL机制相关的“Initial Client Response”的长度，如果没有Initial Response则为-1。

Byten

SASL机制相关的“Initial Response”。

SASLResponse (F)

Byte1('p')

标识该消息是一个初始SASL响应。注意这也被用于GSSAPI、SSPI以及口令响应消息。确切的消息类型可以从上下文推知。

Int32

消息内容的长度，以字节为单位，包括其自身。

Byten

SASL机制相关的消息数据。

SSLRequest (F)

Int32(8)

以字节计的消息内容的长度，包括长度本身。

Int32(80877103)

SSL请求码。选取的值在高16位里包含1234，在低16位里包含5679（为了避免混淆，这个编码必须和任何协议版本号不同）。

GSSAPIRequest (F)

Int32(8)

消息内容的长度以字节为单位，包括自己。

Int32(80877104)

GSSAPI加密请求代码。选择该值以在高16位中包含1234，并且在低16位中包含5680。（为了避免混淆，此代码不得与任何协议版本号相同。）

StartupMessage (F)

Int32

以字节计的消息内容长度，包括长度自身。

Int32(196608)

协议版本号。高16位是主版本号（对这里描述的协议而言是 3）。低16位是次版本号（对于这里描述的协议而言是 0）。

协议版本号后面跟着一个或多个参数名和值字符串的对。要求在最后一个名字/数值对后面有个零字节作为终止符。参数可以以任意顺序出现。user是必须的，其它都是可选的。每个参数是这样指定的：

## String

参数名。目前可以识别的名字是：

user

用于连接的数据库用户名。必须；无缺省。

database

要连接的数据库。缺省是用户名。

options

给后端的命令行参数（这个特性已经废弃，更好的方法是设置单独的运行时参数）。这个字符串中的空格会被当做参数的分隔符，除非用一个反斜线（\）对它转义。写\\可表示一个而字面意义上的反斜线。

replication

用于连入流复制模式，其中可以发出复制命令的一个小型集合而不是SQL语句。值可以是true、false或者database，默认值是false。详情请参考Section 49.4

除了上述参数之外，还可以列出其他参数。以\_pq\_开头的参数名被保留给协议扩展之用，而其他的参数名被当做在后端开始时要设置的运行时参数。这类设置将在后端启动期间被应用（如果有命令行参数，则在解析完命令行参数之后）并且将作为会话的默认值。

## String

参数值。

## Sync (F)

Byte1('S')

表示该消息为一条 Sync 命令。

Int32(4)

以字节计的消息内容的长度，包括长度自身。

## Terminate (F)

Byte1('X')

标识本消息是一个终止。

Int32(4)

以字节计的消息内容的长度，包括长度自身。

## 49.8. 错误和通知消息域

本节描述那些可能出现在ErrorResponse和NoticeResponse消息里的域。每个域类型有一个单字节标识记号。请注意，任意给定的域类型在每条消息里应该最多出现一次。

## S

严重性：该域的内容是ERROR、FATAL或者PANIC（在一个错误消息里），或者WARNING、NOTICE、DEBUG、INFO或者LOG（在一条通知消息里），或者是这些形式的某种本地化翻译。总是会出现。

V

严重性：该域的内容是ERROR、FATAL或者PANIC（在一个错误消息里），或者WARNING、NOTICE、DEBUG、INFO或者LOG（在一条通知消息里）。这和S域相同，不过其内容没有被本地化。这只存在于LightDB版本 9.6 及其后版本产生的消息中。

C

代码：错误的SQLSTATE代码（参阅Appendix 0。非本地化。总是出现。

M

消息：人类可读的错误消息的主体。这些信息应该准确并且简洁（通常是一行）。总是出现。

D

细节：一个可选的二级错误消息，携带了有关问题的更多错误消息。可以是多行。

H

提示：一个可选的有关如何处理问题的建议。它和细节不同的地方是它提出了建议（可能并不合适）而不仅仅是事实。可以是多行。

P

位置：这个域值是一个十进制ASCII整数，表示一个错误游标的位置，它是一个指向原始查询字符串的索引。第一个字符的索引是 1，位置是以字符计算而非字节计算的。

p

内部位置：这个域和P域定义相同，但是它被用于当游标位置指向一个内部生成的命令的情况，而不是用于客户端提交的命令。这个域出现的时候，总是会出现q域。

q

内部查询：失败的内部生成的命令的文本。比如，它可能是一个PL/pgSQL函数发出的SQL 查询。

W

哪里：一个指示错误发生的环境的指示器。目前，它包含一个活跃的过程语言函数的调用堆栈的路径和内部生成的查询。这个路径每个项记录一行，最新的在最前面。

s

模式名：如果错误与一个指定数据库对象相关，这里是包含该对象的模式名（如果有）。

t

表名：如果错误与一个指定表相关，这里是表的名字（引用该表模式的名字的模式名域）。

c

列名：如果错误与一个指定表列相关，这里是该列的名字（引用该模式和表的名字来标识该表）。

d

数据类型名：如果错误与一个指定数据类型相关，这里是该数据类型的名字（引用该数据类型模式的名字的模式名域）。

n

约束名：如果错误是和一个指定约束相关，这里是该约束的名字。引用至上面列出的相关表或域的域（为了这个目的，索引被视作约束，即使它们并不是按照约束语法创建的）。

F

文件：报告的错误在源代码中的文件名。

L

行：报告的错误所在的源代码的位置的行号。

R

例程：报告错误的例程在源代码中的名字。

### Note

用于模式名、表名、列名、数据类型名和约束名只提供给有限的几种错误类型；请参考Appendix C 前端不应当假设任何一个这些域的出现会保证其他域的出现。核心错误资源会看到上面提示的相互关系，但是用户定义的函数可能会以其他方式使用这些域。同样的道理，客户端不应该假设这些域表示当前数据库中同一时期的对象。

客户端负责格式化要显示的信息以符合需要；特别是它应该根据需要断开长的行。在错误消息域里出现的新行字符应该被当作一个分段的符号，而不是换行。

## 49.9. 逻辑复制消息格式

这一节介绍每一种逻辑复制消息的详细格式。这些消息会通过复制槽的SQL接口返回或者由walsender发送。在由walsender发送的情况下，它们被封装在Section 49.4中所述的复制协议WAL消息中，并且通常服从和物理复制相同的消息流。

Begin

Byte1('B')

标识该消息是一个开始消息。

Int64

该事务的最终LSN。

Int64

该事务的提交时间戳。该值是自LightDB纪元（2000-01-01）以来的微秒数。

Int32

事务的XID。

Commit

Byte1('C')

标识该消息是一个提交消息。



Int8

标志，当前未使用（必须为0）。

Int64

提交的LSN。

Int64

该事务的结束LSN。

Int64

该事务的提交时间戳。该值是自LightDB纪元（2000-01-01）以来的微秒数。

Origin

Byte1('0')

标识该消息是一个源头消息。

Int64

源头服务器上的提交LSN。

String

源头的名称。

注意在一个单一事务中可能有多个Origin消息。

Relation

Byte1('R')

标识该消息是一个关系消息。

Int32

关系的ID。

String

名字空间（pg\_catalog是空字符串）。

String

关系名。

Int8

该关系的副本标识设置（和pg\_class中的relreplident相同）。

Int16

列数。

接下来，会为每一列出现下面的消息部分（生成的列除外）：

Int8

该列的标志。当前可以是0（无标志）或者1（标记该列为键的一部分）。

String

列的名称。

Int32

列的数据类型的ID。

Int32

列的类型修饰符 (atttypmod)。

Type

Byte1('Y')

标识该消息是一个类型消息。

Int32

数据类型的ID。

String

名字空间 (pg\_catalog的是空字符串)。

String

数据类型的名称。

Insert

Byte1('I')

标识该消息是一个插入消息。

Int32

对应于关系消息中的ID的关系的ID。

Byte1('N')

标识接下来的TupleData消息是一个新元组。

TupleData

TupleData消息部分表示新元组的内容。

Update

Byte1('U')

标识该消息是一个更新消息。

Int32

对应于关系消息中的ID的关系的ID。

Byte1('K')

标识接下来的TupleData子消息为一个键。这个字段是可选的并且仅当任意列中被更新更改的数据是REPLICA IDENTITY索引的一部分时才存在。

Byte1('0')

标识接下来的TupleData子消息为一个旧元组。这个字段是可选的并且仅当发生更新的表的REPLICA IDENTITY被设置为FULL时才存在。

TupleData

TupleData消息部分表示旧元组或主键的内容。仅当之前的'0'或'K'部分存在时才存在。

Byte1('N')

标识接下来的TupleData消息是一个新元组。

TupleData

TupleData消息部分表示一个新元组的内容。

Update消息可能包含一个'K'消息部分或者一个'0'消息部分，或者两者都不包含，但是绝不会同时包含两者。

Delete

Byte1('D')

标识该消息是一个删除消息。

Int32

对应于关系消息中的ID的关系的ID。

Byte1('K')

标识接下来的TupleData子消息是一个键。如果发生删除的表正好用一个索引作为REPLICA IDENTITY，那么就会存在这个字段。

Byte1('0')

标识接下来的TupleData子消息为一个旧元组。这个字段是可选的并且仅当发生删除的表的REPLICA IDENTITY被设置为FULL时才存在。

TupleData

TupleData消息部分表示旧元组或主键的内容，取决于前面的字段。

Delete消息可能包含一个'K'消息部分或者一个'0'消息部分，或者两者都不包含，但是绝不会同时包含两者。

Truncate

Byte1('T')

标识该消息是一个截断消息。

Int32

关系的数目。

Int8

TRUNCATE的选项位：1表示CASCADE，2表示RESTART IDENTITY

Int32

对应于关系消息中的ID的关系的ID。这个字段会为每个关系重复出现。

下面的消息部分由上面的消息共享。

TupleData

Int16

列数。

接下来，为每一列会有下列子消息之一出现（生成的列除外）：

Byte1('n')

标识该数据为NULL值。

或者

Byte1('u')

标识未更改的被TOAST过的值（实际值没有被发送）。

或者

Byte1('t')

标识数据为文本格式化的值。

Int32

列值的长度。

Byten

该列的值的文本格式（未来的发行可能会支持额外的格式）。n是上面的长度。

## 49. 10. 自协议2.0以来的变化总结

本节提供一个快速的改变检查列表，以便于那些试图将现有的客户端库更新到3.0协议的开发人员。

初始化的启动包用了一个灵活的字符串列表格式取代了固定的格式。请注意，运行时参数的会话缺省值现在可以直接在启动包中指定（实际上，你可以在使用options域之前干这件事情，但是因为options的宽度限制以及缺乏引用值中空白的办法，这并不是很安全的技巧）。

现在所有的消息在消息类型字节后面都有一个长度计数（除了启动包之外，它没有类型字节）。同时还要注意现在PasswordMessage有一个类型字节。

ErrorResponse和NoticeResponse('E'和'N')消息现在包含多个域，从这些域里客户端代码可以组合出自己所希望的详细程度的错误消息。请注意个体的域通常不是用新行终止的，虽然在老的协议里发送的单个字符串总是会用新行终止。

ReadyForQuery('Z')消息包括一个事务状态指示符。

BinaryRow和DataRow消息类型之间的区别不再存在了；单个DataRow消息类型用于返回所有格式的数据。请注意DataRow的布局已经被变得更容易分析。同样，二进制值的表现形式已经改变了：它不再直接和服务器的内部表现形式绑定。

有了一种新的“扩展查询”的子协议，它增加了前端消息类型 Parse、Bind、Execute、Describe、Close、Flush和Sync，以及后端消息类型 ParseComplete、BindComplete、PortalSuspended、ParameterDescription、NoData和 CloseComplete。现有的客户端不用关心这个子协议，但是利用这个子协议将令我们可能提升性能或者功能。

COPY数据现在封装到了CopyData和CopyDone消息里。现在有种很好的方法从正在进行的COPY动作中的错误恢复。最后一行的特殊的“\.”不再需要了，并且在COPY OUT的过程中不再发送（在COPY IN的时候它仍然被认为是一个终止符，但是它的使用已经废弃了并且最终将被删除）。现在支持二进制COPY。CopyInResponse和CopyOutResponse消息包括指示列数目和每个列格式的域。

FunctionCall和FunctionCallResponse消息的布局变化了。FunctionCall现在支持给函数传递NULL参数。它同样可以处理以文本或者二进制格式传递参数和检索结果。我们不用再认为FunctionCall有潜在的安全性漏洞，因为它并不提供对内部服务器数据表现形式的直接访问。

在连接启动期间，后端会为它认为客户端库感兴趣的所有参数发送ParameterStatus（'S'）消息。随后，如果这些参数的任何活跃值发生变化，那么将发送一条ParameterStatus消息。

RowDescription（'T'）消息为所描述的行的每个列运载新表的OID和列编号域。它同样还表示每个列的格式代码。

后端不再生成 CursorResponse（'P'）消息。

NotificationResponse（'A'）消息有一个附加的字符串域，它能携带来自NOTIFY事件发送者的一个“载荷”字符串。

EmptyQueryResponse（'I'）以前包含一个空字符串参数；现在已经被删除。

---

# Chapter 50. LightDB编码习惯

## 50.1. 格式化

源代码格式化使用 4 列制表间隔，并且保留制表符（即制表符不会被扩展成空格）。每一个逻辑缩进层就是一个额外的制表位。

布局规则（括号定位等）遵循 BSD 习惯。特别地，if、while、switch等受控块的花括号要独自占据一行。

限制行长度，这样在 80 列窗口中代码也是可读的（这并不意味着你不能超过 80 列。例如，为了保持代码在 80 列以内而在任意位置打断一段长的错误消息字符串可能不会给可读性带来什么好处）。

要保持一致的编码风格，请勿使用C++风格注释（// comments）。pgindent 会用 /\* ... \*/ 替换它们。

多行注释块更好的风格是

```
/*
 * 注释文本从这里开始
 * 延续到这里
 */
```

注意从列 1 开始的注释块将被pgindent原样保留，但是它将重新对缩进的注释块断行，好像它们是纯文本一样。如果你想要保留一个缩进块中的换行，可以像这样增加破折号：

```
/*-----
 * 注释文本从这里开始
 * 延续到这里
 *-----
 */
```

虽然在提交补丁时并不是一定要遵守这些格式化规则，但是最好能遵守。在下次发行前你提交的代码将会通过pgindent，因此使用某种其他格式化习惯无法使代码被编排得很好。对于补丁，一种比较好的经验规则是“让新代码看起来像它周围的原有代码”。

src/tools目录包含有可以用于emacs、xemacs或者vim编辑器的设置文件，它们可以帮助确保这些编辑器会根据上述习惯格式化代码。

可以这样调用文本浏览工具more和less：

```
more -x4
less -x4
```

来让它们以合适的方式显示制表符。

## 50.2. 在服务器中报告错误

服务器代码内产生的错误、警告和日志消息应该使用ereport或者更老的elog生成。这个函数的使用有些复杂，因此有必要做一些解释。

对于每一个消息都有两个必要的元素：一个严重性级别（从DEBUG到PANIC）和一个主消息文本。此外还有一些可选元素，其中最常见的是一个遵守 SQL 规范中 SQLSTATE 习惯的错误标识符代码。ereport本身只是一个 shell 宏，它的存在主要是为了在语法习惯上让消息的产

生更像 C 源代码中的单个函数调用而已。ereport直接接受的唯一参数是严重性级别。主消息文本和任何其他可选消息元素通过在ereport调用中使用辅助函数产生，例如errmsg。

对于ereport的一次典型调用可能看起来像：

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

这会指定错误严重性级别为ERROR（一个普通错误）。errcode调用使用src/include/utils/errcodes.h中定义的一个宏指定 SQLSTATE 错误代码。errmsg调用提供主消息文本。

您还会经常看到这种较旧样式，在辅助函数调用周围有一组额外的圆括号：

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

在 LightDB 版本 12 之前需要额外的括号，但现在是可选的。

这里有一个更复杂的例子：

```
ereport(ERROR,
        errcode(ERRCODE_AMBIGUOUS_FUNCTION),
        errmsg("function %s is not unique",
              func_signature_string(funcname, nargs,
                                   NIL, actual_arg_types)),
        errhint("Unable to choose a best candidate function. "
               "You might need to add explicit typecasts."));
```

这展示了使用格式代码把运行时值嵌入到一个消息文本中的方法，其中还提供了一个可选的“hint”消息。辅助函数调用可以按任何顺序写入，但通常先出现errcode和errmsg。

如果严重级别是ERROR或更高，ereport 会中止当前查询的执行并且不会返回到调用者。如果严重级别低于 ERROR，ereport会正常返回。

ereport可用的辅助例程是：

- `errcode(sqlerrcode)` 指定对于该情况的 SQLSTATE 错误标识符代码。如果没有调用这个例程，错误严重性级别是ERROR或更高时错误标识符会默认成ERRCODE\_INTERNAL\_ERROR，错误级别是WARNING时标识符为ERRCODE\_WARNING，否则（对于NOTICE及以下的级别）标识符会被设置为ERRCODE\_SUCCESSFUL\_COMPLETION。虽然这些默认值常常很方便，在忽略errcode()调用之前请总是思考一下它们是否合适。
- `errmsg(const char *msg, ...)` 指定主错误消息文本，以及需要插入其中的运行时值。这种插入以sprintf风格的格式代码指定。除了sprintf接受的标准格式代码，格式代码%m可以用来插入由strerror为errno的当前值返回的错误消息。<sup>1</sup> %m不要求errmsg参数列表中的任何对应项。注意在格式代码被处理之前，消息字符串将通过gettext来进行可能的本地化。
- `errmsg_internal(const char *msg, ...)` 与errmsg相同，不过消息串将不会被翻译，也不会被包括在国际化的消息字典中。这不应该被用于“不能发生”的情况中，因为那些情况下不值得花费精力去做翻译。
- `errmsg_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` 很像errmsg，但是支持消息的多种复数形式。fmt\_singular是英语单数格

<sup>1</sup> 也就是说，该值时到达ereport调用时的当前值，在辅助报告例程内errno的改变不会影响它。但如果你在errmsg的参数列表中显式地写了strerror(errno)则不是这样，因此不要那样做。

式, `fmt_plural`是英语复数格式, `n`是决定需要何种复数形式的整数值, 剩下的参数会根据选中的格式字符串进行格式化。详见Section 51.2.2

- `errdetail(const char *msg, ...)` 提供了一个可选的“详情”消息, 如果有额外的信息但不适合放在主消息中时就可以使用这种方式。消息字符串的处理与`errmsg`相同。
- `errdetail_internal(const char *msg, ...)`与`errdetail`相同, 不过消息串将不会被翻译, 也不会被包括在国际化的消息字典中。这应该被用于不值得花费精力翻译的详情消息上, 例如它们对大部分用户太过技术化而没什么用处。
- `errdetail_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)`与`errdetail`相似, 但是支持消息的多种复数形式。详见Section 51.2.2
- `errdetail_log(const char *msg, ...)`与`errdetail`相同, 除了这个字符串只会进入服务器的日志而不会发往客户端。如果同时使用了`errdetail` (或者上述的一种等效函数) 以及`errdetail_log`, 那么一个字符串会被发往客户端而另一个会被发往日志。如果错误细节的安全性过于敏感或者体积过于庞大而不适合于发往客户端, 这个函数就非常有用。
- `errdetail_log_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)`与`errdetail_log`相似, 但是支持多种复数形式的消息。详见Section 51.2.2
- `errhint(const char *msg, ...)`提供一个可选的“hint”消息, 它被用来提供关于如何修复该问题的建议。该消息字符串以和`errmsg`相同的方式处理。
- `errhint_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)`与`errhint`相似, 但是支持多种复数形式的消息。详见Section 51.2.2
- `errcontext(const char *msg, ...)`通常不会被直接从一个`ereport`消息站点调用, 它被用在`error_context_stack`回调函数中来提供错误发生的上下文, 例如一个 PL 函数中的当前位置。该消息字符串以和`errmsg`相同的方式处理。不同于其他辅助函数, 在每次`ereport`调用中可以多次调用这个函数, 这样提供的连续的字符串将被用单独的新行串接在一起。
- `errposition(int cursorpos)`指定一个查询字符串中错误的文本位置。当前, 它只对查询处理的词法和语法分析阶段中检测到的错误有用。
- `errtable(Relation rel)`指定一个关系, 它的名称和模式名称应该被包括在错误报告中作为辅助域。
- `errtablecol(Relation rel, int attnum)`指定一个列, 它的名称、表名和模式名称应该被包括在错误报告中作为辅助域。
- `errtableconstraint(Relation rel, const char *conname)`指定一个约束, 它的名称、表名和模式名称应该被包括在错误报告中作为辅助域。索引应当为考虑成用于这种目的的约束, 不管它们有没有一个相关联的`pg_constraint`项。要小心地以`rel`传递底层堆关系而不是索引本身。
- `errdatatype(Oid datatypeOid)`指定一个数据类型, 它的名称和模式名称应该被包括在错误报告中作为辅助域。
- `errdomainconstraint(Oid datatypeOid, const char *conname)`指定一个域约束, 它的名称、域名和模式名称应该被包括在错误报告中作为辅助域。
- `errcode_for_file_access()`是一个便捷函数, 它可以在一个文件访问相关的系统调用中为一个失败选择一个合适的 SQLSTATE 错误标识符。它使用保存下来的`errno`来决定要差生哪种错误代码。通常, 应该把它和主错误消息文本中的`%m`联合使用。
- `errcode_for_socket_access()`是一个便捷函数, 它可以在一个套接字相关的系统调用中为一个失败选择一个合适的 SQLSTATE 错误标识符。



- 可以调用`errhidestmt(bool hide_stmt)`来指定抑制 `postmaster` 日志中消息里的STATEMENT:部分。如果消息文本已经包含当前语句，通常这是合适的。
- 可以调用`errhidecontext(bool hide_ctx)`来指示抑制 `postmaster` 日志中消息里的CONTEXT:部分。这应该只被用于 `verbose` 模式的调试消息，这类消息中被重复包含的上下文信息会让日志膨胀得非常厉害。

### Note

在一个`ereport`调用中，最多可以使用一个`errtable`、`errtablecol`、`errtableconstraint`、`errdatatype`或者`errdomainconstraint`函数。这些函数存在是为了允许应用抽取与错误情况相关的数据库对象名，而不需要检查可能已被本地化的错误消息文本。这些函数应该被用在应用需要对其进行自动错误处理的错误报告中。从LightDB 9.3开始，完整的覆盖只为 `SQLSTATE` 类别 23 中的错误存在（完整性约束违背），但是在未来这些很可能会被扩展。

有一个还在大量使用的旧的函数`eelog`。一个`eelog`调用：

```
eelog(level, "format string", ...);
```

完全等效于：

```
ereport(level, errmsg_internal("format string", ...));
```

注意 `SQLSTATE` 错误代码总是被给予默认值，并且消息字符串不受翻译影响。因此，`eelog`应该只被用于内部错误和低层次的调试日志。任何普通用户感兴趣的消息应该通过`ereport`。不管怎样，在仍广泛使用`eelog`的系统中，有足够多的内部“不可能发生”的错误检查，出于记号简洁的目的这更适用于那些消息。

有关编写好的错误消息的建议可见Section 50.3

## 50.3. 错误消息风格指导

提供这个风格指导是希望所有由LightDB产生的消息间维护一种一致的、用户友好的风格。

### 哪儿该放什么

主要的消息应该简短、实事求是并且避免引用诸如特定函数名这样的实现细节。“简短”意味着“在正常情况下应该能放在一行内”。如果需要保持主要消息简短，或者如果你觉得有必要提到诸如特定系统调用失败这样的实现细节，可以使用一个详细消息。主要消息和详细消息都应该实事求是。对于有关如何修复问题的建议可以使用一个提示消息，特别是该建议可能不总是合适时。

例如，与其用：

```
IpcMemoryCreate: shmget(key=%d, size=%u, 0%) failed: %m
(plus a long addendum that is basically a hint)
```

不如写：

```
Primary:    could not create shared memory segment: %m
Detail:    Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint:      the addendum
```

原理：保持主要消息简短有助于扣住主题并且有助于客户端基于错误消息放在一行足以的假设来安排屏幕空间。详细消息和提示消息可以被归入一种详情模式，或者可能是一种弹出式错误细节窗口。还有，详细消息和提示消息通常会被排除在服务器日志之外以节省空间。最好避免对实现细节的引用，因为用户根本就不想知道细节。

## 格式化

不要对消息文本的格式化做任何特定假设。可以期望客户端和服务器日志换行来适应其自身的需要。在长消息中，`\n` 可以被用来指示建议的分段。不要用一个换行结束一个消息。不要使用制表符或其他格式化字符（在错误上下文显示中，换行会自动被增加来分割上下文的层次，例如函数调用）。

原理：消息不一定非得显示在终端上。在 GUI 显示或浏览器中，这些格式化指令最多会被忽略。

## 引号

在引用时，英语文本应该使用双引号。其他语言中的文本应该一致地使用一种符合出版习惯和其他程序计算机输出的引号。

原理：选择双引号而不是单引号其实有点武断，但是更倾向于首选用法。有人建议过按照 SQL 习惯根据对象类型来选择引号的种类（即，字符串单引号，标识符双引号）。但是这是一个很多用户根本不熟悉的语言内部的技术问题，它不会扩散到其他种类的被引用术语，也不会翻译成其他语言，并且也相当没有意义。

## 引号的使用

总是用引号界定文件名、用户提供的标识符以及其他可能包含词的变量。不要用它们来标记不会包含词的变量（例如，操作符名称）。

在后端有函数会根据需要对其自身的输出加上双引号（例如 `format_type_be()`）。不要在这种函数的输出周围再加上额外的引号。

原理：在被嵌入到一个消息时，对象可能具有产生歧义的名称。对于标记一个插入名称的开始和结束要始终如一。但不要在消息中混入不必要的或者重复的引号。

## 语法和标点

主要错误消息的规则与详细/提示消息的规则不同：

主要错误消息：不要大写第一个字母。不要用一个句号结束一个消息。甚至不要考虑用一个感叹号结束一个消息。

详细和提示消息：使用完整的句子，并且每一个都用句号结束。对句子的第一个词进行首字母大写。如果后面跟着另一个句子，在句号后面放两个空格（对于英语文本有效，在其他语言中可能不合适）。

错误上下文字符串：不要大写第一个字母并且不要用一个句号结束字符串。上下文字符串通常不应该是完整的句子。

原理：避免标点让客户端应用能更容易地把消息嵌入到各种各样的语法上下文中。通常主要消息不是语法上完整的句子（并且如果它们比一个句子还长，它们应该被分成主要和详细部分）。不过，详细和提示消息会更长并且需要包括多个句子。为了一致，即便它们只是一个句子，它们也应该遵循完整句子的风格。

## 大写 vs. 小写

对消息使用小写形式，包括一个主要错误消息的第一个字母。如果 SQL 命令或者关键词出现在消息中，为它们使用大写形式。

原理：用这种方式能更容易使所有的东西看起来更加一致，因为一些消息是完整的句子而另一些不是。

## 避免被动态

使用主动语态。在有主语时使用完整句子（“A could not do B”）。如果主语是程序本身，使用没有主语的电报风格，但不要为程序使用“I”。

原理：程序不是人类。所以不要假装。

## 现在式 vs. 过去式

如果一次尝试做某事失败但是可能在下一次成功（也许在修复某个问题之后），则使用过去式。如果失败必定是持久的，使用现在式。

在以下形式的句子之间存在着非平凡的语义区别：

```
could not open file "%s": %m
```

和

```
cannot open file "%s"
```

第一个表示打开文件的尝试失败。该消息应该给出一个原因，例如“磁盘满”或者“文件不存在”。过去式更合适，因为下一次磁盘可能不再满或者请求的文件可能就存在了。

第二种形式指示打开所提及文件的功能在程序中根本就不存在，或者概念上是不可能的。现在式更合适，因为该情况将无限期保持。

原理：诚然，普通用户没有能力从消息的时态上得出重要的结论，但是既然语言为我们提供了语法，我们就应该正确地使用它。

## 对象类型

在引用一个对象的名称时，说明该对象的种类。

原理：否则没有人会知道“foo.bar.baz”是什么。

## 括号

方括号只被用于：（1）在命令对照表中标记可选参数，或者（2）用于标记一个数组下标。

原理：任何其他用法都无法符合众所周知的使用习惯，并且会使人们感到困惑。

## 组装错误消息

当一个消息包括在别处产生的文本时，这样将它嵌入：

```
could not open file %s: %m
```

原理：将这种文本粘贴到一个单一的语句中很难解释所有可能的错误代码，因此需要某种标点。也有人建议把被嵌入的文本放在圆括号中，但是如果被嵌入的文本可能是消息中最重要的一部分（这是常有的事）时就显得不自然。

## 错误原因

消息应该总是说明为什么错误会发生的原因。例如：

```
BAD:    could not open file %s
BETTER: could not open file %s (I/O failure)
```

如果不知道原因，你最好修复代码。

## 函数名

不要在错误文本中包括报告例程的名称。需要时，我们有其他机制能够知道这些，并且对于大部分用户来说这种信息没有用处。如果没有函数名错误文本就没有意义，那么请重写它。

```
BAD:    pg_strtoint32: error in "z": cannot parse "z"
BETTER: invalid input syntax for type integer: "z"
```

也要避免提及被调用的函数名，而是说出代码试图做什么：

```
BAD:    open() failed: %m
BETTER: could not open file %s: %m
```

如果它真地看起来必要，在详细消息中提及系统调用（在某些情况中，可以为详细消息提供传递给系统调用的实际值）。

原理：用户不知道所有那些函数做了什么。

## 应该避免的捣蛋词

Unable. “Unable”接近于被动态。酌情使用“cannot”或者“could not”更好。

Bad. “bad result”之类的错误消息实在很难被解释。最好写出为什么结果“不好”，例如“无效格式”。

Illegal. “Illegal”表示违背了法律，类似的还有“invalid”。更好的是，说明为什么无效。

Unknown. 尝试避免“unknown”。考虑“error: unknown response”。如果你不知道响应是什么，你怎么知道它是错误的？“Unrecognized”常常是一个更好的选择。还有，一定要包括被抱怨的值。

```
BAD:    unknown node type
BETTER: unrecognized node type: 42
```

找到 vs. 存在. 如果程序使用了一个非平凡的算法来定位一个资源（例如一个路径搜索）并且该算法失败了，说该程序无法“找到”该资源比较公平。换句话说，如果该资源应该在的位置是已知的，但是程序无法在那里访问它，那么才说该资源不“存在”。在这种情况下使用“找到”听起来很弱并且会使问题混淆。

May vs. Can vs. Might. “May”表示权限（例如，“You may borrow my rake.”），并且在文档或错误消息中用处有限。“Can”表示能力（例如，“I can lift that log.”），

而“might”表示可能性（例如，“It might rain today.”）。请使用合适的词来使含义清晰并且便于翻译。

缩略形式。 避免缩略，如“can't”，请使用“cannot”。

Non-negative. 避免“non-negative”，因为它不能清楚表达是否接受0。最好使用“greater than zero”或“greater than or equal to zero”。

## 适当的拼写

完整地拼出单词。例如，避免：

- spec
- stats
- parens
- auth
- xact

基本原理：这将增强一致性。

## 本地化

记住错误消息文本需要被翻译成其他语言。请遵循Section 51.2.2中的方针以避免让翻译者为难。

## 50.4. 其他编码习惯

### C 标准

LightDB中的代码应该只依赖于 C99 标准中的语言特性。这意味着遵循 C99 的编译器一定能编译 postgres，至少除开少数平台依赖问题之外。

目前，C99 标准中包含的一些特性，不允许用于核心功能 LightDB代码。这当前包括可变长度数组、混合声明和代码、// 注释、通用字符名称。原因包括可移植性和历史实践。

如果提供了回退，则可以使用 C 标准或编译器后续版本中的特性。

例如当前使用的 `_Static_assert()` 和 `__builtin_constant_p`，即使它们分别来自 C 标准的更新修订和 GCC 扩展。如果没有，我们分别回退到 C99 兼容替换来执行相同的检查，但会发出相当神秘的消息，并且不使用 `__builtin_constant_p`。

## 类函数的宏以及内联函数

带有参数的宏以及 `static inline` 函数都可能被使用。当类似如下情况写作宏时会有多次计算风险或者宏可能非常长时，使用后者会更好。

```
#define Max(x, y) ((x) > (y) ? (x) : (y))
```

在其他情况下只能使用宏，或者说使用宏至少更容易。例如，由于多种类型的表达式需要被传递给宏。

当一个内联函数的定义引用只在后端中可用的符号（即变量、函数）时，从前端代码引用该函数时该函数可能不可见。

```

#ifndef FRONTEND
static inline MemoryContext
MemoryContextSwitchTo(MemoryContext context)
{
    MemoryContext old = CurrentMemoryContext;

    CurrentMemoryContext = context;
    return old;
}
#endif /* FRONTEND */

```

在这个例子中，CurrentMemoryContext只在后端中可用，但该函数引用了它并且该函数因此被#ifndef FRONTEND隐藏。之所以存在这条规则，是因为即使内联函数中包含的符号没有被使用，有些编译器也会发出对它们的引用。

## 编写信号处理器

为了能适合在信号处理器中运行，代码必须被非常仔细地编写。根本问题是，除非被阻塞，信号处理器能在任何时候打断代码。如果信号处理器内部的代码使用和外面代码相同的状态，很可能会出现混乱。例如，可以想想如果一个信号处理器试图取得已经在被打断代码中持有的锁时会发生什么。

除特殊安排的代码之外，在信号处理器中只应该调用对异步信号安全的函数（如 POSIX 中定义的那样）并且只访问volatile sig\_atomic\_t类型的变量。一些lightdb中的函数也被视作是信号安全的，其中很重要的一个是SetLatch()。

在大部分情况下，信号处理器应该只提示一个信号已经到达，并且使用一个 latch 唤醒运行在处理器之外的代码。这样一个处理器的例子如下：

```

static void
handle_sighup(SIGNAL_ARGS)
{
    int          save_errno = errno;

    got_SIGHUP = true;
    SetLatch(MyLatch);

    errno = save_errno;
}

```

errno会被保存并且恢复，因为SetLatch()可能会更改它。如果不这样做，当前正在观测errno的被中断代码可能会看到错误的值。

## 调用函数指针

为了清晰，如果函数指针是一个简单的变量，在调用指向的函数时显式地对其解除引用会更好。例如：

```
(*emit_log_hook) (edata);
```

（虽然emit\_log\_hook(edata)还会有效）。当函数指针是一个结构的组成部分时，则额外的标点符号能够被省略并且通常也应该被省略，例如：

```
paramInfo->paramFetch(paramInfo, paramId);
```

---

# Chapter 51. 本国语言支持

## 51.1. 给翻译者

LightDB程序（服务器和客户端）可以用你最喜欢的语言发出它们的消息 — 如果消息已经被翻译过。常见和维护已翻译的消息集需要以该语言为母语并且愿意为LightDB做贡献的人们的帮助。要做这些你根本不需要成为一个程序员。本节将解释如何来帮助我们。

### 51.1.1. 需求

我们将不会判断你的语言技巧 — 这一节是关于软件工具。理论上，你只需要一个文本编辑器。但是这仅仅是最少见的情况：你不想试验你的已翻译好的消息。当你配置你的源代码树时，一定要使用`--enable-nls`选项。这将也会检查`libintl`库和`msgfmt`程序，这些是所有终端用户将会需要的。要试验你的工作，请按安装指导的相应部分进行。

如果你想开始一个新的翻译工作或想做一次消息目录合并（在后面描述），你将分别需要程序`xgettext`和`msgmerge`的 GNU 兼容的实现。稍后，我们将试着安排它这样如果你使用一个打包好的源代码发布，你将不需要`xgettext`（如果使用来自 Git 的发布，你仍将需要它）。我们目前推荐GNU Gettext 0.10.36或更高版本。

你的本地 `gettext` 实现应该有其自身的文档。其中的某些可能与接下来的内容重复，但是你应该看看来了解更多细节。

### 51.1.2. 概念

原始（英语）消息和它们（可能的）已翻译等价物组成的对被保持在消息目录中，每个程序（尽管相关程序可以共享一个消息目录）以及每个目标语言一个目录。消息目录有两种文件格式：第一种是“PO”文件（用于可移植对象），它是一个翻译者编辑的具有特殊语法的普通文本文件。第二种是“MO”文件（用于机器对象），它是从相应的 PO 文件生成的一个二进制文件并且在国际化程序运行时使用。翻译者不需要处理 MO 文件；事实上几乎没有人这样做。

消息目录文件的扩展名不出意料地是 `.po` 或 `.mo`。基础名要么是与之合作的程序名，要么是该文件的语言名称，取决于具体情况。这有一点让人困惑。例子是 `ltsql.po`（用于 `ltsql` 的 PO 文件）或 `fr.mo`（法语的 MO 文件）。

PO 文件的文件格式示例如下：

```
# comment

msgid "original string"
msgstr "translated string"

msgid "more original"
msgstr "another translated"
"string can be broken up like this"
```

...

`msgid`行被从程序源代码中抽取（不是必须这样做，但是这是最常用的方法）。`msgstr` 行最初是空的并且被翻译者用以填充有用的字符串。字符串可以包含 C 风格的转义字符并且可以跨越行（下一行必须开始于行首）。

`#` 字符引入一个注释。如果空白紧跟着 `#` 字符，那么这是一个由翻译者维护的注释。也可能有自动注释，它在 `#` 后紧跟一个非空白字符。这些由多种操作 PO 文件并且意图辅助翻译者的多种工具维护。

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

风格的注释是从使用消息的源文件中抽取而来。有可能程序员已经为翻译者插入了信息，例如关于期望的对齐。#: 注释指示该消息在源代码中被使用的确切位置。翻译者不需要查看程序源代码，但是如果有关于正确翻译的疑问，他也能够去查看。#, 注释包含以某种方法描述该消息的标志。当前有两种标志：如果该消息可能由于程序源码中的改变而过时，则fuzzy被设置。翻译者则可以验证这个标识并且可能移除 fuzzy 标志。注意 fuzzy 消息不会对终端用户可用。另一种标志是c-格式，它指示该消息是一个printf-风格的格式模板。这意味着翻译也应当是一个格式字符串，其中有相同个数和相同类型的占位符。有工具能验证这点，它会切断该 c-格式标志。

### 51.1.3. 创建并维护消息目录

好，那么我们如何创建一个“空的”消息目录呢？首先，进入到包含你希望翻译其消息的程序的目录中。如果那里有一个文件nls.mk，那么这个程序就已经准备好被翻译了。

如果已经有某些.po文件，那么有些人已经完成了一些翻译工作。这些文件被命名为语言.po，其中语言是 ISO 639-1 双字符语言代码（小写形式）<sup>1</sup>，例如法语是fr.po。如果真的对于一种语言有需要多个翻译任务那么这些文件也可以被命名为语言\_区域.po，其中region是 ISO 3166-1 双字符国家代码（大写形式）<sup>2</sup>，例如巴西的葡萄牙语是pt\_BR.po。如果你找到了你想要的语言，你就可以开始在那个文件上工作了。

如果你需要开始一项新的翻译工作，那么首先运行命令：

```
make init-po
```

这将创建一个文件 程序名.pot（.pot用来把它与“在生产中的” PO 文件区分开。T表示“template”）。将这个文件拷贝到语言.po并编辑它。要让让人知道新语言可用，还要编辑文件nls.mk并且增加该语言（或语言与国家）代码到如下的行中：

```
AVAIL_LANGUAGES := de fr
```

（当然可以出现其他的语言）。

如果底层程序或库改变，消息可能会被程序员改变或增加。在这种情况下，你不需要从头开始。相反，运行命令：

```
make update-po
```

它将创建一个新的空白消息目录文件（你从其开始的 pot 文件）并且将把它合并到现有的 PO 文件中。如果合并算法不确定一个特定的消息，它会把该消息标记为如上所释的“fuzzy”。新的 PO 文件会被保存为一个.po.new扩展名。

### 51.1.4. 编辑 PO 文件

PO 文件可以用一个常规文本编辑器编辑。翻译者应该只改变 msgstr 指令之后引号之间的区域、增加注释并且修改 fuzzy 标志。Emacs 有一个 PO 模式（不出意料），我们觉得它非常有用。

PO 文件不需要被完全填充。如果无翻译（或一个空翻译）可用，该软件将自动回退到原始的字符串。提交一个未完成的翻译包含到源码树中不存在问题，那样可以让其他人有机会继

<sup>1</sup> [https://www.loc.gov/standards/iso639-2/php/English\\_list.php](https://www.loc.gov/standards/iso639-2/php/English_list.php)

<sup>2</sup> <https://www.iso.org/iso-3166-country-codes.html>



续你的工作。不过，我们鼓励你在做完合并之后优先移除模糊项。记住模糊项将不会被安装，它们只作为什么可能是正确翻译的参考。

这里有一些在编辑翻译时要记住的事情：

- 确定如果原始消息以一个新行结束，那么翻译后的消息也要保持一样。制表符等也类似。
- 如果原始消息是一个printf格式字符串，翻译消息也需要是那样。翻译也需要保留相同顺序的格式指示符。有时语言的天然规则让这样做不可能或者至少很尴尬。在那样的情况下，你可以这样修改格式指示符：

```
msgstr "Die Datei %2$s hat %1$u Zeichen."
```

那么第一个占位符将实际上使用列表中的第二个参数。digits\$需要在任何其他格式操纵符之前紧跟 %（这个特性真实地存在于printf函数族中。你可能之前没有听说过它，因为除消息国际化之外很少会用到它）。

- 如果原始字符串包含一个语法错误，报告它（或者在程序源码中自己修复它）并且正常进行翻译。更正过的字符串可以在程序源码已被更新时合并。如果原始字符串包含一个事实错误，报告它（或自己修复它）并且不要翻译它。相反，你可以在 PO 文件中的该字符串标注一个注释。
- 保持原始字符串的风格和语气。特别地，不是句子的消息（cannot open file %s）可能不以一个大写字母开始（如果你的语言区分字母大小写）或者一个句号结尾（如果你的语言使用标点符号）。阅读Section 50. 可能有所帮助。
- 如果你不知道一个消息的含义，或者如果它是模棱两可的，在开发者邮件列表中询问其正确含义。如果说英语的终端用户可能也无法理解它或者发现它是模棱两可的，那么最好改进该消息。

## 51.2. 给编程者

### 51.2.1. 技术

这一节描述如何在LightDB发布中的一个程序或库中实现本地语言支持。当前，这些知识只适用于 C 程序。

为一个程序增加 NLS 支持

1. 将这里的代码插入到该程序的启动序列中：

```
#ifdef ENABLE-NLS
#include <locale.h>
#endif

...

#ifdef ENABLE-NLS
setlocale(LC_ALL, "");
bindtextdomain("programe", LOCALEDIR);
textdomain("programe");
#endif
```

（程序名实际上可以自由选择）。

2. 不管在哪里找到一个可被翻译的消息，需要插入一个gettext()调用，例如：

```
fprintf(stderr, "panic level %d\n", lvl);
```

将被改成：

```
fprintf(stderr, gettext("panic level %d\n"), lvl);
```

（如果 NLS 支持没有被配置，`gettext` 被定义为一个空操作）。

这容易增加很多混乱。一种常用的捷径是：

```
#define _(x) gettext(x)
```

如果该程序通过一个或几个函数（例如后端中的 `ereport()`）完成他的大部分通信，则有另一种可行的解决方案。那么你可以在所有输入字符串上都内部调用这个函数 `gettext`。

3. 在程序源码的目录中增加一个文件 `nlsmk`。这个文件将被读作一个 `makefile`。其中要创建下列变量赋值：

```
CATALOG_NAME
```

程序名，如 `textdomain()` 调用中所提供的。

```
AVAIL_LANGUAGES
```

提供的翻译列表 — 初始为空。

```
GETTEXT_FILES
```

包含可翻译字符串的文件列表，即那些被用 `gettext` 或另一种替代方案标记的文件。最终，这将包括该程序近乎所有的源文件。如果这个列表太长你可以让第一个“file”变成一个+并且第二个词变成一个包含那些文件名的文件，第二个词指向的文件中每个文件名一行。

```
GETTEXT_TRIGGERS
```

为翻译者产生在其上工作的消息目录的工具需要知道，哪些函数调用包含可翻译的字符串。默认情况下，只有 `gettext()` 调用是已知的。如果你使用 `_` 或其他标识符，你需要在这里列出它们。如果可翻译的字符串不是第一个参数，该条目需要是形式 `func:2`（用于第二个参数）。如果你有一个支持复数消息的函数，该条目应该看起来像 `func:1,2`（标识单数和复数消息参数）。

构建系统将自动处理消息目录的编译和安装。

## 51.2.2. 消息书写指南

下面是一些书写已于翻译的消息的指南。

- 不要在运行时构建句子，如：

```
printf("Files were %s.\n", flag ? "copied" : "removed");
```

该句子中的词序可能在其他语言中完全不同。同样，即使你记得在每一个片段上调用 `gettext()`，片段也可能不会被独立翻译得很好。更好的方式是复制一点代码，这样每个消息将被以一个整体被翻译。只有数字、文件名和这样的运行时变量才应该被在运行时插入到一个消息文本中。

- 由于类似的原因，下面的例子也不会工作：

```
printf("copied %d file%s", n, n!=1 ? "s" : "");
```

因为它假定了复数形式。如果你发现了这个问题，你可以这样解决它：

```
if (n==1)
    printf("copied 1 file");
else
    printf("copied %d files", n);
```

然后会失望。某些语言有多于两种形式，使用某些古怪的规则。通常最好设计消息以完全避免该问题，例如像这样：

```
printf("number of copied files: %d", n);
```

如果你真的想要构建一个正确的复数消息，有对此的支持，但是有点笨拙。当在 `errreport()` 中产生一个主要或详细错误消息时，你可以这样写一些东西：

```
errmsg_plural("copied %d file",
              "copied %d files",
              n,
              n)
```

第一个参数是适用于英语单数形式的格式字符串，第二个参数是适用于英语复数形式的格式字符串，并且第三个参数是控制使用哪种复数形式的整数。后续参数针对每个格式字符串按照常规被格式化（通常，复数控制值也将是要被格式化的值之一，因此它必须被写两次）。在英语中只有 `n` 是否为 1 才重要，但是在其他语言中可以有多种不同的复数形式。翻译者将这两种英语形式视为一组，并有机会提供多个替代字符串，并根据 `n` 的运行时值选择适当的替代字符串。

如果你需要复数化一个不直接进入 `errmsg` 或 `errdetail` 报告的消息，你必须使用底层函数 `ngettext`。见 `gettext` 文档。

- 如果您想与翻译人员交流某些内容，例如关于消息如何与其他输出对齐，在该字符串出现的地方之前放上一个以 `translator` 开始的注释，例如：

```
/* translator: This message is not what it seems to be. */
```

这些注释被复制到消息目录文件中，这样翻译者可以看到它们。

---

## Chapter 52. 编写一个过程语言处理器

所有对用不是当前“版本 1”接口（用于编译型语言）语言编写的函数（这包括用户定义的过程语言中的函数、SQL 编写的函数）的调用都会流经一个用于指定语言的调用处理器。调用处理器负责以一种有意义的方式执行该函数，例如通过解释所提供的源文本。本章勾勒了如何编写一个新的过程语言调用处理器的轮廓。

一个过程语言的调用处理器是一个“正常的”函数，它必须以一种编译型语言（如 C）编写、使用版本-1接口并且在LightDB中注册为无参数且返回类型language\_handler。这种特殊的伪类型标识该函数是一个调用处理器并且阻止它在 SQL 命令中被直接调用。关于 C 语言调用惯例和动态载入的更多细节，请见Section 36.10

调用处理器的调用方式和其他任何函数相同：它接收一个包含参数值和有关被调用函数信息的FunctionCallInfoBaseData 结构，并且它被期望返回一个Datum结果（并且如果它希望返回一个 SQL 空值结果，它可能设置FunctionCallInfoBaseData结构的isnull域）。一个调用处理器和一个普通被调用函数之间的区别是FunctionCallInfoBaseData结构的flinfo->fn\_oid域将包含要被调用的实际函数的 OID，而不是调用处理器本身。调用处理器必须使用这个域来决定要执行哪个函数。同样，被传递的参数列表已经被根据目标函数而不是调用处理器的声明被设置好。

调用处理器负责从pg\_proc系统目录中取得该函数的项并且分析被调用函数的参数和返回类型。该函数的CREATE FUNCTION命令的AS子句可以在pg\_proc行的prosrc列中被找到。通常这是过程语言中的源文本，但是在理论上它可以是其他某种东西，例如一个文件的路径名或其他任何详细告诉调用处理器做什么的东西。

同一个函数在每个 SQL 命令中常常被调用多次。一个调用处理器可以通过使用flinfo->fn\_extra域来避免重复查找关于被调用函数的信息。这个域最初将为NULL，但是可以被调用处理器设置为指向关于被调用函数的信息。在后续调用中，如果flinfo->fn\_extra已经为非-NUL，则它可以被使用并且信息查找步骤将被跳过。调用处理器必须确保flinfo->fn\_extra被指向直到当前查询的末尾都存活的内存，因为一个FmgrInfo数据接口可以被保持那么久。一种方式是在flinfo->fn\_mcxt指定的内存上下文中分配额外的数据；这样的数据通常必须与FmgrInfo本身具有相同的生命期。但是处理器也可以选择使用一个生存时间更长的内存上下文，这样它能够在查询之间缓存函数定义信息。

当一个过程语言函数被作为一个触发器调用时，不会有参数通过常用方式被传递，但是FunctionCallInfoBaseData的context域指向一个TriggerData结构而不是为NULL（就像它在一个普通函数调用中那样）。一个语言处理器应该为过程语言函数提供机制来得到触发器信息。

这是一个用 C 编写的过程语言处理器的模板：

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;
```

```

if (CALLED_AS_TRIGGER(fcinfo))
{
    /*
     * Called as a trigger function
     */
    TriggerData    *trigdata = (TriggerData *) fcinfo->context;

    retval = ...
}
else
{
    /*
     * Called as a function
     */

    retval = ...
}

return retval;
}

```

要完成该调用处理器，只需要加入几千行代码来替代点号即可。

在将处理器函数编译成一个可载入模块（Section 36.10.5 后，接着用下列命令注册例子过程语言：

```

CREATE FUNCTION plsample_call_handler() RETURNS language_handler
AS 'filename'
LANGUAGE C;
CREATE LANGUAGE plsample
HANDLER plsample_call_handler;

```

尽管提供一个调用处理器对于创建一个最小过程语言已经足够，还可以提供其他两个可选的函数来让该函数更易用。它们是验证器和内联处理器。一个验证器可以被提供来允许在CREATE FUNCTION期间完成语言相关的检查。一个内联处理器可以被提供来允许语言支持通过DO命令执行匿名代码块。

如果一个验证器被一个过程语言提供，它必须被声明为一个采用一个单一oid类型参数的函数。该验证器的结果被忽略，因为它通常被声明为返回void。验证器将在一个已经创建了或更新了一个以该过程语言编写的函数的CREATE FUNCTION命令之后被调用。被传入的OID是函数的pg\_proc行的OID。验证器必须用通常方式取得这个行，并且做任何合适的检查。首先，调用CheckFunctionValidatorAccess()来诊断对用户通过CREATE FUNCTION无法达到的验证器的显式调用。典型的检查包括验证函数的参数和结果类型是否被该语言支持，以及该函数体在该语言中语法是否正确。如果验证器发现该函数是好的，它应该只是返回。如果它发现一个错误，它应该通过通常的ereport()错误报告机制报告该错误。抛出一个错误将强制一次事务回滚并且因此阻止不正确的函数定义被提交。

验证器函数通常应该尊重check\_function\_bodies参数：如果它被关闭那么任何代价大的或上下文敏感的检查应该被跳过。如果该语言提供了编译时代码执行，验证器必须抑制可能引起这种执行的检查。特别地，这个参数会被lt\_dump关闭，这样它能载入过程语言函数而不用担心副作用或那些函数体对其他数据库对象的依赖（因为这种要求，调用处理器应该避免假设验证器已经完整地检查过该函数。拥有一个验证器的要点不是让调用处理器忽略检查，而是如果在一个CREATE FUNCTION命令中发现明显错误时立即提示用户）。然而究竟检查什么的选择大部分都留给了验证器函数，注意当check\_function\_bodies为打开时，核心CREATE FUNCTION代码只执行附加到一个函数的SET子句。因此，为了避免在重新载入一个转储时的伪失败，当check\_function\_bodies为关闭时，结果可能会被GUC参数影响的检查绝对应当被跳过。

如果一个过程语言提供了一个内联处理器，它必须被声明为一个采用一个单一internal类型参数的函数。内联处理器的结果会被忽略，因此它通常被声明为返回void。当一个DO语句被调用执行指定过程语言时，内联处理器将被调用。实际被传递的参数是一个指向一个InlineCodeBlock结构的指针，它包含有关DO语句参数的而信息，特别是将被执行的匿名代码块的文本。内联处理器应该执行该代码并返回。

我们推荐你包装所有这些函数声明，以及CREATE LANGUAGE命令本身到一个extension中，这样一个简单的CREATE EXTENSION命令就足以安装该语言。关于编写扩展的信息请见Section 36.17

在尝试编写你自己的语言处理器时，包括在标准发布中的过程语言是很好的参考。看看源码树中的src/pl子目录。CREATE LANGUAGE参考页也有一些有用的细节。

---

## Chapter 53. 编写一个外部数据包装器

所有在一个外部表上的操作都通过它的外部数据包装器来处理，外部数据包装器由一组被核心服务器调用的函数组成。外部数据包装器负责从远程数据源取得数据并把它返回给 LightDB 执行器。如果要支持更新外部表，包装器也需要处理更新。本章将介绍如何编写一个新的外部数据包装器。

在你试图编写你自己的外部数据包装器时，包含在标准发布中的外部数据包装器会是很好的参考。请看看源代码树的 contrib 子目录。CREATE FOREIGN DATA WRAPPER 参考页也会有很多有用的细节。

### Note

SQL 标准声明了一个接口用来编写外部数据包装器。但是，LightDB 没有实现该 API，因为将其纳入到 LightDB 中的工作量将会很大，并且标准的 API 并没有得到广泛地采用。

### 53. 1. 外部数据包装器函数

FDW 的作者需要实现一个处理器函数，并且可以有选择地实现一个验证器函数。两个函数都必须被用一种编译语言（如 C）来编写，并使用版本-1 接口。关于 C 语言调用规范和动态载入的细节，请见 Section 36.10

处理器函数简单地返回一个包含了多个回调函数指针的结构体，回调函数将被规划器、执行器和多种维护命令调用。编写一个 FDW 的大部分工作量都在实现这些回调函数上。处理器函数必须被注册在 LightDB 中，并且注册为不需要参数并且返回特殊的伪类型 fdw\_handler。回调函数则是普通的 C 函数并且对于 SQL 层是不可见的或者不可调用的。回调函数在 Section 53. 中描述。

验证器函数负责验证 CREATE 和 ALTER 命令中对它的外部数据包装器给出的选项，以及使用该包装器的外部服务器、用户映射和外部表。验证器函数必须被注册为要求两个参数：一个包含需要被验证的选项的文本数组，以及一个表示与这些选项相关联的对象类型的 OID（以该对象可能被存储的系统目录的 OID 的形式，可以是 ForeignDataWrapperRelationId、ForeignServerRelationId、UserMappingRelationId 或 ForeignTableRelationId）。如果没有提供验证器函数，在对象创建或修改时选项不会被检查。

### 53. 2. 外部数据包装器回调例程

FDW 处理器函数返回一个 palloc 过的 FdwRoutine 结构，它包含下文描述的回调函数的指针。扫描相关的函数是必需的，剩下的是可选的。

FdwRoutine 结构类型被声明在 src/include/foreign/fdwapi.h 中，可以查看它来获得额外的信息。

#### 53. 2. 1. 用于扫描外部表的 FDW 例程

```
void
GetForeignRelSize(PlannerInfo *root,
                  RelOptInfo *baserel,
                  Oid foreigntableid);
```

获取一个外部表的关系尺寸估计。在对一个扫描外部表的查询进行规划的开头将调用该函数。root 是规划器的关于该查询的全局信息；baserel 是规划器的关于该表的信

息；foreigntableid是外部表在pg\_class中的OID（foreigntableid可以从规划器的数据结构中获得，但是为了减少工作量，这里直接显式地将它传递给函数）。

这个函数应该更新baserel->rows为表扫描根据限制条件完成了过滤后将返回的预期行数。baserel->rows的初始值只是一个常数的默认估计值，应该尽可能把它替换掉。如果该函数能够计算出一个平均结果行宽度的更好的估计值，该函数也可能选择更新baserel->width。（初始值基于列数据类型和最后一个ANALYZE测量的列平均宽度值。）此外，如果该函数能够更好地计算出外部表的总行数，则该函数可以更新baserel->tuples。（初始值来自pg\_class.reltuples表示最后一次ANALYZE看到的总行数。）

更多信息请见Section 53.4

```
void
GetForeignPaths(PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid);
```

为一个外部表上的扫描创建可能的访问路径。这个函数在查询规划过程中被调用。参数和GetForeignRelSize相同，后者已经被调用过了。

这个函数必须为外部表上的扫描生成至少一个访问路径（ForeignPath节点），并且必须调用add\_path把每一个这样的路径加入到baserel->pathlist中。我们推荐使用create\_foreignscan\_path来建立ForeignPath节点。该函数可以生成多个访问路径，例如一个具有合法pathkeys的路径表示一个预排序好的结果。每一个访问路径必须包含代价估计，并且能包含任何FDW的私有信息，这种信息被用来标识想要使用的指定扫描方法。

更多信息请见Section 53.4

```
ForeignScan *
GetForeignPlan(PlannerInfo *root,
               RelOptInfo *baserel,
               Oid foreigntableid,
               ForeignPath *best_path,
               List *tlist,
               List *scan_clauses,
               Plan *outer_plan);
```

从选择的外部访问路径创建一个ForeignScan计划节点。这个函数在查询规划的末尾被调用。参数和GetForeignRelSize的一样，外加选中的ForeignPath（在前面由GetForeignPaths、GetForeignJoinPaths或者GetForeignUpperPaths产生）、被计划节点发出的目标列表以及计划节点强制的限制子句以及被RecheckForeignScan执行的复查所使用的ForeignScan的外子计划（如果该路径是用于一个连接而非基本关系，则foreigntableid是InvalidOid）。

这个函数必须创建并返回一个ForeignScan计划节点，我们对它使用make\_foreignscan来建立ForeignScan节点。

更多信息见Section 53.4

```
void
BeginForeignScan(ForeignScanState *node,
                 int eflags);
```

开始执行一个外部扫描。这个函数在执行器启动阶段被调用。它应该执行任何在扫描能够开始之前需要完成的初始化工作，但是并不开始执行真正的扫描（会在第一次调用IterateForeignScan时完成）。ForeignScanState节点已经被创建好了，但是它的fdw\_state域仍然为NULL。关于要被扫描的表的信息可以通过ForeignScanState节点访问



（特殊地，从底层的ForeignScan计划节点，它包含任何由GetForeignPlan提供的FDW私有信息）。eflags包含描述执行器对该计划节点操作模式的标志位。

注意当(eflags & EXEC\_FLAG\_EXPLAIN\_ONLY)为真时，这个函数不应该执行任何外部可见的动作；它应当只做最少的事情来创建对ExplainForeignScan和EndForeignScan有效的节点状态。

```
TupleTableSlot *
IterateForeignScan(ForeignScanState *node);
```

从外部源获得一行，将它放在一个元组表槽中返回（节点的ScanTupleSlot应当被用于此目的）。如果没有更多的行可用则返回 NULL。元组表槽设施允许一个物理的或者虚拟的元组被返回；在大部分情况下出于性能的考虑会倾向于选择后者。注意这是在一个短期存在的内存上下文中被调用的，该内存上下文会在调用之间被重置。如果你需要长期存在的存储，请在BeginForeignScan中创建内存上下文，或者使用节点的EState中的es\_query\_cxt。

如果提供了fdw\_scan\_tlist目标列表，被返回的行必须匹配它，如果没有提供则它们必须匹配被扫描的外部表的行类型。如果选择优化掉不需要的列，你应该在那些列的位置上插入控制或者生成一个忽略了那些列的fdw\_scan\_tlist列表。

注意LightDB的执行器并不在乎被返回的行是否违背了定义在该外部表上的任何约束 — 但是规划器会在乎这一点，并且如果在外部表中有可见行不满足一个约束，规划器可能会错误地优化查询。如果当用户已经声明一个约束应该为真时它却被违背，最合适的处理可能是产生一个错误（就像在数据类型失配的情况下所作的那样）。

```
void
ReScanForeignScan(ForeignScanState *node);
```

从头开始重启一个扫描。注意扫描所依赖的任何参数可能已经改变了值，因此新扫描不一定会返回完全相同的行。

```
void
EndForeignScan(ForeignScanState *node);
```

结束扫描并释放资源。通常释放palloc过的内存并不重要，但是打开的文件和到远程服务器的连接等应该被清理。

## 53.2.2. 用于扫描外部连接的 FDW 例程

如果一个 FDW 支持远程执行外部连接（而不是先把两个表的数据取到本地然后做本地连接），它应该提供这个回调函数：

```
void
GetForeignJoinPaths(PlannerInfo *root,
                   RelOptInfo *joinrel,
                   RelOptInfo *outerrel,
                   RelOptInfo *innerrel,
                   JoinType jointype,
                   JoinPathExtraData *extra);
```

它为两个（或更多）同属于一台外部服务器的外部表的连接创建可能的访问路径。这个可选的函数会在查询规划过程中被调用。和GetForeignPaths一样，这个函数应该为提供的joinrel生成ForeignPath路径(用 create\_foreign\_join\_path 构建它们)，并且调用add\_path把这些路径加入到该连接应该考虑的路径集合中。但是和GetForeignPaths不一样的是，不需要这个函数产生最少一个路径，因为涉及本地连接的路径总是可用的。

注意为相同的连接关系将会重复地调用这个函数用来生成内外关系的不同组合。FDW 需要负责最小化其中重复的工作。

如果一个ForeignPath路径被选中用于该连接，它将在整个连接处理中存在，为其中的成分表和子连接产生的路径将不会被使用。后续对该连接路径的处理大部分和扫描单个外部表的路径一样。一点不同是ForeignScan计划节点的scanrelid应该被设置为零，因为它表示的不是单个关系，而是用ForeignScan节点的fs\_relids域来表示被连接的关系集合（后一个域会被核心规划器代码自动设置，不需要由 FDW 填充）。另一点不同是，由于一个远程连接的列列表无法在系统目录中找到，FDW 必须用一个合适的TargetEntry节点列表来填充fdw\_scan\_tlist，表示运行时它返回的元组中提供的列的集合。

更多信息请见Section 53.4

### 53.2.3. 用于规划扫描/连接后处理的 FDW 例程

如果一个 FDW 支持执行远程的扫描/连接后处理，例如远程聚集，那么它应该提供这个回调函数：

```
void
GetForeignUpperPaths(PlannerInfo *root,
                    UpperRelationKind stage,
                    RelOptInfo *input_rel,
                    RelOptInfo *output_rel,
                    void *extra);
```

为上层关系处理创建可能的访问路径，这是规划器针对所有扫描/连接后查询处理的术语，例如聚集、窗口函数、排序和表更新。在查询规划期间会调用这个可选的函数。当前，只有当该查询中涉及的所有基本关系都属于同一个 FDW 时才会调用这个函数。这个函数应该为 FDW 知道如何远程执行的任何扫描/连接后处理生成ForeignPath路径(用create\_foreign\_upper\_path 构建它们)，并且调用add\_path把这些路径加入到上层关系中。就GetForeignJoinPaths来说，并不要求这个函数在创建任何路径时都能成功，因为路径总是有可能涉及到本地处理。

stage参数表示当前正在考虑的是哪一个扫描/连接后处理步骤。output\_rel是接收表示这一个步骤的路径的上层关系，而input\_rel是表示这个步骤输入的关系。extra参数提供额外的细节，当前只会为UPPERREL\_PARTIAL\_GROUP\_AGG或者UPPERREL\_GROUP\_AGG设置它，这种情况下它会指向一个GroupPathExtraData结构；或者对于 UPPERREL\_FINAL，在某种情况下它指向一个FinalPathExtraData 结构。（注意被加入到output\_rel中的ForeignPath路径通常对input\_rel的路径没有直接的依赖，因为它们的处理被认为是在外部处理的。不过，检查为前一个处理步骤生成的路径有助于避免冗余的规划工作）。

更多信息请见Section 53.4

### 53.2.4. 更新外部表的FDW例程

如果一个FDW支持可写的外部表，根据FDW的需要和功能它应该提供某些或者全部下列回调函数：

```
void
AddForeignUpdateTargets(Query *parsetree,
                      RangeTblEntry *target_rte,
                      Relation target_relation);
```

UPDATE和DELETE操作是在之前由表扫描函数取出的行上被执行的。FDW可能需要额外的信息（例如一个行ID或主键列的值）来保证它能够找到要更新或删除的准确行。要支持这些要求，这个函数可以项列列表中增加额外的隐藏或“junk”的目标列，它们在一个UPDATE或DELETE期间会被从外部表中获取。

要做到这一点，向`parsetree->targetList`中增加`TargetEntry`项，它们包含要被获取的额外值的表达式。每一个这样的项必须被标记为`resjunk = true`，并且必须有一个可区分的`resname`用于在执行期间标识它。请避免使用匹配`ctidN`、`wholerow`或`wholerowN`的名字，因为核心系统可能会生成使用这些名字的`junk`列。如果额外的表达式比简单的`Var`更加复杂，在把它们加入到目标列表之前必须把它们用`eval_const_expressions`进行处理。

尽管这个函数在规划过程中被调用，但所提供的信息与其他规划例程可用的信息有点区别。`parsetree`是`UPDATE`或`DELETE`命令的分析树，而`target_rte`和`target_relation`描述目标外部表。

如果`AddForeignUpdateTargets`指针被设置为`NULL`，则不会有额外的目标表达式被加入（这将使得我们不可能实现`DELETE`操作，而`UPDATE`则还有可能是可行的，前提是`FDW`依赖一个未改变的主键来标识行）。

```
List *
PlanForeignModify(PlannerInfo *root,
                  ModifyTable *plan,
                  Index resultRelation,
                  int subplan_index);
```

执行外部表上插入、更新或删除所需的任何附加规划动作。这个函数生成`FDW`私有信息，该信息将被附加到执行该更新动作的`ModifyTable`计划节点。这个私有信息的形式必须是一个`List`，并将会在执行阶段被传递给`BeginForeignModify`。

`root`是规划器关于该查询的全局信息。`plan`是`ModifyTable`计划节点，它除了`fdwPrivLists`域之外是完整的。`resultRelation`通过目标外部表的范围表索引来标识它。`subplan_index`标识这是`ModifyTable`计划节点的哪个目标，从零开始计数；如果你希望索引到`plan->plans`或其他`plan`节点的子结构中，请使用它。

更多信息见Section 53.4

如果`PlanForeignModify`指针被设置为`NULL`，则不会有额外的计划时动作被执行，并且传递给`BeginForeignModify`的`fdw_private`列表也将为`NIL`。

```
void
BeginForeignModify(ModifyTableState *mtstate,
                  ResultRelInfo *rinfo,
                  List *fdw_private,
                  int subplan_index,
                  int eflags);
```

开始执行一个外部表修改操作。这个例程在执行器启动期间被调用。它应该执行任何先于实际表修改的初始化工作。随后，`ExecForeignInsert`、`ExecForeignUpdate`或`ExecForeignDelete`将被为每一个将被插入、更新或删除的元组调用。

`mtstate`是要被执行的`ModifyTable`计划节点的状态信息；通过这个结构可以得到关于规划和执行阶段的全局数据。`rinfo`是描述目标外部表的`ResultRelInfo`结构（`ResultRelInfo`的`ri_FdwState`域用于`FDW`来存储它在此操作中需要的任何私有状态）。`fdw_private`包含`PlanForeignModify`生成的私有数据。`subplan_index`标识这是`ModifyTable`计划节点的哪个目标。`eflags`包含描述执行器对该计划节点操作模式的标志位。

注意当`(eflags & EXEC_FLAG_EXPLAIN_ONLY)`为真，这个函数不应执行任何外部可见的动作；它只应该做最少的工作来创建`ExplainForeignModify`和`EndForeignModify`可用的节点状态。

如果`BeginForeignModify`指针被设置为`NULL`，在执行器启动期间将不会采取任何动作。

```

TupleTableSlot *
ExecForeignInsert(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);

```

插入一个元组到外部表。estate是查询的全局执行状态。rinfo是描述目标外部表的ResultRelInfo结构。slot包含要被插入的元组；它将匹配外部表的行类型定义。planSlot包含由ModifyTable计划节点的子计划生成的元组；它与slot不同，它可能包含额外的“junk”列（INSERT情况通常不关心planSlot，但是为了完整性还是在这里提供它）。

返回值可以是一个包含实际被插入的数据的槽（这可能会和所提供的数据不同，例如一个触发器动作的结果），或者为 NULL 表示实际没有插入行（还是触发器的结果）。被传入的slot可以被重用于这个目的。

在返回槽中的数据只有在INSERT语句具有一个RETURNING子句或包括一个视图WITH CHECK OPTION；或者如果外部表具有一个AFTER ROW触发器时才被使用。触发器要求所有的列，但是FDW应该选择优化成根据RETURNING子句的内容或WITH CHECK OPTION约束返回某些或全部列。不管怎样，某些槽必须被返回来指示成功，或者查询报告的行计数将会是错误的。

如果ExecForeignInsert指针被设置为NULL，尝试向外部表插入将会失败并报告一个错误消息。

请注意，在外表上将路由元组插入外表分区或执行 COPY FROM 时也会调用此函数，在这种情况下，它的调用方式与INSERT中的情况有所不同。请参阅下面所述的允许FDW支持的回调函数。

```

TupleTableSlot *
ExecForeignUpdate(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);

```

更新外部表中的一个元组。estate是查询的全局执行状态。rinfo是描述目标外部表的ResultRelInfo结构。slot包含元组的新数据；它将匹配外部表的行类型定义。planSlot包含由ModifyTable计划节点的子计划生成的元组；它与slot不同，它可能包含额外的“junk”列（INSERT情况通常不关心planSlot，但是为了完整性还是在这里提供它）。特殊地，任何AddForeignUpdateTargets所要求的junk列在这个槽中都是有效的。

返回值可以是一个包含实际被更新的数据的槽（这可能会和所提供的数据不同，例如一个触发器动作的结果），或者为 NULL 表示实际没有更新行（还是触发器的结果）。被传入的slot可以被重用于这个目的。

在返回槽中的数据只有在UPDATE语句具有一个RETURNING子句或者包括一个视图WITH CHECK OPTION；或者如果外部表具有一个AFTER ROW触发器时才被使用。触发器要求所有的列，但是FDW应该选择优化成根据RETURNING子句的内容或WITH CHECK OPTION约束返回某些或全部列。不管怎样，某些槽必须被返回来指示成功，或者查询报告的行计数将会是错误的。

如果ExecForeignUpdate指针被设置为NULL，尝试更新外部表将会失败并报告一个错误消息。

```

TupleTableSlot *
ExecForeignDelete(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);

```

从外部表删除一个元组。estate是查询的全局执行状态。rinfo是描述目标外部表的ResultRelInfo结构。slot在调用时不包含任何有用的东西，但是可以被用于保持被返回的元组。planSlot包含由ModifyTable计划节点的子计划生成的元组；特殊地，它将携带AddForeignUpdateTargets所要求的任意垃圾列。垃圾列被用来标识要被删除的元组。

返回值可以是一个包含实际被删除的数据的槽（这可能会和所提供的数据不同，例如一个触发器动作的结果），或者为 NULL 表示实际没有删除行（还是触发器的结果）。被传入的slot可以被重用于这个目的。

在返回槽中的数据只有在DELETE查询具有一个RETURNING子句或者外部表具有一个AFTER ROW触发器时才被使用。触发器要求所有的列，但是 FDW 应该选择优化成根据RETURNING子句的内容返回某些或全部列。不管怎样，某些槽必须被返回来指示成功，或者查询报告的行数将会是错误的。

如果ExecForeignDelete指针被设置为NULL，尝试从外部表中删除将会失败并报告一个错误消息。

```
void
EndForeignModify(EState *estate,
                 ResultRelInfo *rinfo);
```

结束表更新并释放资源。通常释放palloc的内存并不重要，但是打开的文件和到远程服务器的连接等应当被清除。

如果EndForeignModify指针被设置为NULL，在执行器关闭期间不会采取任何动作。

被INSERT或者COPY FROM插入到分区表中的元组会被路由到分区。如果一个FDW支持可路由的外部表分区，它还应该提供下面的回调函数。当在外部表上执行COPY FROM时，也会调用这些函数。

```
void
BeginForeignInsert(ModifyTableState *mtstate,
                   ResultRelInfo *rinfo);
```

开始在外部表上执行插入操作。当外部表被选中作为元组路由的分区以及COPY FROM命令中指定的目标时，在第一个元组被插入到该外部表之前会调用这个例程。它应该执行实际插入之前所需的任何初始化工作。随后，为每一个被插入到该外部表的元组都将调用ExecForeignInsert。

mtstate是正在被执行的ModifyTable计划节点的总体状态，通过这个结构可以得到有关计划和执行的全局数据。rinfo是描述目标外部表的ResultRelInfo结构（对于FDW，ResultRelInfo的ri\_FdwState字段用来存放这个操作所需要的私有状态）。

当这个例程被一个COPY FROM命令调用时，不会提供mtstate中与计划相关的全局数据，并且后续为每个插入元组调用的ExecForeignInsert的planSlot参数为NULL，不管该外部表是为元组路由选中的分区还是命令中指定的目标。

如果BeginForeignInsert指针被设置为NULL，则不会采取初始化动作。

请注意如果 FDW 不支持在外部表上的可路由外表分区和/或执行 COPY FROM ，这个函数或ExecForeignInsert 后续的调用必须根据需要发出错误。

```
void
EndForeignInsert(EState *estate,
                 ResultRelInfo *rinfo);
```

结束插入操作并且释放资源。通常释放palloc的内存并不重要，但是打开的文件和与远程服务器的连接应该被清除。

如果EndForeignInsert指针被设置为NULL，则不会采取终止动作。

```
int
IsForeignRelUpdatable(Relation rel);
```

报告指定的外部表支持哪些更新操作。返回值应该是一个规则事件编号的位掩码，它指示了哪些操作被外部表支持，它使用CmdType枚举，即： $(1 \ll \text{CMD\_UPDATE}) = 4$ 表示UPDATE、 $(1 \ll \text{CMD\_INSERT}) = 8$ 表示INSERT以及 $(1 \ll \text{CMD\_DELETE}) = 16$ 表示DELETE。

如果IsForeignRelUpdatable指针被设置为NULL，而FDW提供了ExecForeignInsert、ExecForeignUpdate或ExecForeignDelete，则外部表分别被假定为可插入、可更新或可删除。只有在FDW支持某些表是可更新的而某些不是可更新的时候，才需要这个函数（即便如此，也允许在执行例程中抛出一个错误而不是在这个函数中检查。但是，这个函数被用来决定显示在information\_schema视图中的可更新性）。

一些对于外部表的插入、更新和删除可以通过实现另一组接口来优化。普通的插入、更新和删除接口会从远程服务器取得行，然后一次修改其中一行。在某些情况下，这种逐行的方式是必要的，但是可能效率不高。如果有可能让外部服务器判断哪些行可以直接修改而无需先检索它们，并且没有本地结构会影响该操作（行级本地触发器，存储生成的列，或来自父视图的WITH CHECK OPTION约束。那么可以让整个操作在远程服务器上执行。下面介绍的接口能让这种做法变成可能。

```
bool
PlanDirectModify(PlannerInfo *root,
                 ModifyTable *plan,
                 Index resultRelation,
                 int subplan_index);
```

决定在远程服务器上执行直接修改是否安全。如果安全，执行所需的规划动作然后返回true。否则返回false。这个可选的函数在查询规划期间被调用。如果这个函数成功，在执行阶段将会调用BeginDirectModify、IterateDirectModify和EndDirectModify。否则，对表的修改将采用上文描述的表更新函数来执行。参数和PlanForeignModify的相同。

要在远程服务器上执行直接修改，这个函数必须用一个ForeignScan计划节点（它在远程服务器上执行直接修改）重写目标子计划。ForeignScan的operation域必须被合适地设置为CmdType枚举值，即CMD\_UPDATE表示UPDATE、CMD\_INSERT表示INSERT而CMD\_DELETE表示DELETE。

更多信息请见Section 53.4

如果PlanDirectModify指针被设置为NULL，不会尝试在远程服务器上执行直接修改。

```
void
BeginDirectModify(ForeignScanState *node,
                 int eflags);
```

准备在远程服务器上执行一次直接修改。这个函数会在执行器启动时被调用。它应该执行直接修改所需的任何初始化工作（应该在第一次IterateDirectModify调用之前完成）。ForeignScanState节点已经被创建，但是它的fdw\_state域仍然为NULL。有关要被修改的表的信息可以通过ForeignScanState节点（具体地，从底层的ForeignScan计划节点，它包含了PlanDirectModify提供的FDW-私有信息）访问。eflags包含描述执行器对于这个计划节点操作模式的标志位。

注意当(eflags & EXEC\_FLAG\_EXPLAIN\_ONLY)为真时，这个函数不应该执行任何外部可见的动作。它应当只做最少的工作让该节点状态对ExplainDirectModify和EndDirectModify有效。

如果BeginDirectModify指针被设置为NULL，不会尝试在远程服务器上执行直接修改。

```

TupleTableSlot *
IterateDirectModify(ForeignScanState *node);

```

当INSERT、UPDATE或者DELETE查询没有RETURNING子句时，完成远程服务器上的直接修改后返回 NULL。当查询有该子句时，取出一个包含RETURNING计算所需数据的结果，用一个元组表槽返回它（节点的ScanTupleSlot应被用于这一目的）。实际被插入、更新或者删除的数据必须被存储在该节点的EState的es\_result\_relation\_info->ri\_projectReturning->pi\_exprContext->ecxt\_scantuple中。如果没有更多行可用，则返回 NULL。注意这个函数会在一个短期生存的内存上下文中被调用，该上下文会在两次调用之间被重置。如果需要长期存在的存储，可以在BeginDirectModify中创建一个内存上下文，或者使用该节点的EState中的es\_query\_cxt。

如果提供了fdw\_scan\_tlist目标列表，则被返回的行必须匹配它。否则，被返回的行必须匹配被更新的外部表的行类型。如果选择优化掉RETURNING计算不需要的列，应该在这些列的位置上插入空值，或者生成一个忽略这些列的fdw\_scan\_tlist列表。

不管该查询是否具有RETURNING子句，查询所报告的行计数必须由 FDW 本身增加。当查询没有该子句时，FDW 还必须为EXPLAIN ANALYZE情况下的ForeignScanState节点增加行计数。

如果IterateDirectModify指针被设置为NULL，不会尝试在远程服务器上执行直接修改。

```

void
EndDirectModify(ForeignScanState *node);

```

在远程服务器上的直接修改后进行清理。通常释放在 malloc 分配的内存并不重要，但是诸如打开的文件和到远程服务器的连接应该被清除。

如果EndDirectModify指针被设置为NULL，不会尝试在远程服务器上执行直接修改。

## 53.2.5. 用于行锁定的 FDW 例程

如果一个 FDW 希望支持后期行锁定（如Section 53.5中所述），它必须提供下列回调函数：

```

RowMarkType
GetForeignRowMarkType(RangeTblEntry *rte,
                      LockClauseStrength strength);

```

报告要对一个外部表使用哪个行标记选项。rte是该表的RangeTblEntry节点，而strength描述FOR UPDATE/SHARE子句（如果有）所要求的锁长度。结果必须是RowMarkType枚举类型的一个成员。

这个函数在查询规划期间会为每一个出现在UPDATE、DELETE或者SELECT FOR UPDATE/SHARE查询中的外部表调用，并且该外部表不是UPDATE和DELETE的目标。

如果GetForeignRowMarkType指针被设置为NULL，将总是使用ROW\_MARK\_COPY选项（这意味着将不会调用RefetchForeignRow，因此也不必提供它）。

更多信息请见Section 53.5

```

void
RefetchForeignRow(EState *estate,
                  ExecRowMark *erm,
                  Datum rowid,
                  TupleTableSlot *slot,
                  bool *updated);

```

从外部表中重新取得一个元组槽，如有必要先锁定它。estate是该查询的全局执行状态。erm是描述目标外部表以及要获取的行锁类型（如果有）的ExecRowMark结构。rowid标识要取得的元组。slot在调用时不包含有用内容，但可用于保存返回的元组。updated是一个输出参数。

此函数应将元组存储到提供的槽中，或者在无法获得行锁时清除该槽。要获得的行锁由erm->markType定义，它是之前由GetForeignRowMarkType返回的值（ROW\_MARK\_REFERENCE标识只重新取得元组但不获得任何锁，这个例程将不会看到ROW\_MARK\_COPY）。

此外，如果取得的是一个更新过的版本而不是之前获得的同一版本，\*updated应被设置为true（如果FDW无法确定这一点，推荐总是返回true）。

注意在默认情况下，获取行锁失败应该导致产生错误。如果erm->waitPolicy指定了SKIP LOCKED，只有返回空槽才是合适的。

rowid是要被重新取得的行之之前读到的ctid值。尽管rowid值被作为Datum传递，但是目前它只能被读作tid。选择该函数API是希望未来能允许其他的行ID数据类型。

如果RefetchForeignRow指针被设置为NULL，重新取得行的尝试将会失败并伴随有一个错误消息。

更多信息请见Section 53.5

```
bool
RecheckForeignScan(ForeignScanState *node, TupleTableSlot *slot);
```

重新检查之前返回的元组是否仍然匹配相关的扫描和连接条件，并且可能提供该元组的一个修改版本。对于不执行连接下推的外部数据包装器，通常把这设置为NULL并且恰当地设置fdw\_recheck\_qual会更方便。不过当外部连接被下推时，把与所有基表相关的检查重新应用在结果元组上是不够的，即便所有需要的属性都存在也是如此，因为匹配某个条件失败可能会导致某些属性变成NULL，而不是没有元组被返回。RecheckForeignScan能够重新检查条件，并且在它们仍然满足时返回真，否则返回假，但是它也能够提供的槽中存储一个替换元组。

要实现连接下推，外部数据包装器通常将构造一个可替代的本地连接计划，它只被用来做重新检查。这将变成ForeignScan的外子计划。在需要一次重新检查时，这个子计划可以被执行并且结果元组可以被存储在槽中。这个计划不需要效率很高，因为不会有基表返回超过一行。例如，它可以把所有的连接实现为嵌套循环。函数GetExistingLocalJoinPath可以被用来在已有的路径中搜索合适的本地连接路径，它可以被用作替换的本地连接计划。GetExistingLocalJoinPath会在指定连接关系的路径列表中搜索一个非参数化路径（如果没有找到这样的路径，它会返回NULL，这种情况下外部数据包装器可以自行构造本地路径或者可以选择不为这个连接创建访问路径）。

## 53.2.6. EXPLAIN的FDW例程

```
void
ExplainForeignScan(ForeignScanState *node,
                  ExplainState *es);
```

为一个外部表扫描打印额外的EXPLAIN输出。这个函数可以调用ExplainPropertyText和相关函数来向EXPLAIN输出中增加域。es中的标志域可以被用来决定什么将被打印，并且ForeignScanState节点的状态可以被检查来为EXPLAIN ANALYZE提供运行时统计数据。

如果ExplainForeignScan指针被设置为NULL，在EXPLAIN期间不会打印任何额外的信息。



```
void
ExplainForeignModify(ModifyTableState *mtstate,
                    ResultRelInfo *rinfo,
                    List *fdw_private,
                    int subplan_index,
                    struct ExplainState *es);
```

为一个外部表更新打印额外的EXPLAIN输出。这个函数可以调用ExplainPropertyText和相关函数来向EXPLAIN输出中增加域。es中的标志域可以被用来决定什么将被打印，并且ModifyTableState节点的状态可以被检查来为EXPLAIN ANALYZE提供运行时统计数据。前四个参数和BeginForeignModify相同。

如果ExplainForeignModify指针被设置为NULL，在EXPLAIN期间不会打印任何额外的信息。

```
void
ExplainDirectModify(ForeignScanState *node,
                   ExplainState *es);
```

为远程服务器上的直接修改打印额外的EXPLAIN输出。这个函数可以调用ExplainPropertyText和相关函数来为EXPLAIN输出增加域。es中的标志域可以被用来判断要打印什么，并且在EXPLAIN ANALYZE情况中可以观察ForeignScanState节点的状态来提供运行时统计信息。

如果ExplainDirectModify指针被设置为NULL，EXPLAIN期间不会打印出额外的信息。

## 53.2.7. ANALYZE的FDW例程

```
bool
AnalyzeForeignTable(Relation relation,
                   AcquireSampleRowsFunc *func,
                   BlockNumber *totalpages);
```

当ANALYZE被执行在一个外部表上时会调用这个函数。如果FDW可以为这个外部表收集统计信息，它会返回true并提供一个函数指针，该函数将从func中的表上收集采样行，外加totalpages中页面中的表尺寸估计值。否则，返回false。

如果FDW不支持为任何表收集统计信息，AnalyzeForeignTable指针可以被设置为NULL。

如果提供，采样收集函数必须具有签名

```
int
AcquireSampleRowsFunc(Relation relation,
                     int elevel,
                     HeapTuple *rows,
                     int targrows,
                     double *totalrows,
                     double *totaldeadrows);
```

应该从该表上收集最多targrows行的一个随机采样并将它存放到调用者提供的rows数组中。实际被收集的行的数量必须被返回。另外，将表中有效行和死亡行的总数存储到输出参数totalrows和totaldeadrows中（如果FDW没有死亡行的概念，将totaldeadrows设置为0）。

## 53.2.8. IMPORT FOREIGN SCHEMA的 FDW 例程

```
List *
ImportForeignSchema(ImportForeignSchemaStmt *stmt, Oid serverOid);
```

取得一个外部表创建命令的列表。在执行IMPORT FOREIGN SCHEMA时会调用这个函数，并且会把该语句的解析树以及要使用的外部服务器的OID传递给它。它应该返回一个C字符串的列表，每一个必须包含一个CREATE FOREIGN TABLE命令。这些命令将被核心服务器所解析和执行。

在ImportForeignSchemaStmt结构中，remote\_schema是要从其中导入这些表的远程模式的名称。list\_type标识如何过滤表名：FDW\_IMPORT\_SCHEMA\_ALL表示该远程模式中的所有表都应该被导入（这种情况下table\_list为空），FDW\_IMPORT\_SCHEMA\_LIMIT\_TO表示只包括table\_list中列出的表，而FDW\_IMPORT\_SCHEMA\_EXCEPT则表示排除table\_list中列出的表。options是一个用于该导入处理的选项列表。选项的含义由FDW决定。例如，一个FDW可以用一个选项来定义是否应该导入列的NOT NULL属性。这些选项不需要与那些FDW支持的数据库对象选项有什么关系。

FDW可能会忽略ImportForeignSchemaStmt的local\_schema域，因为核心服务器会自动地向解析好的CREATE FOREIGN TABLE命令中插入本地模式的名称。

FDW也不必担心实现list\_type以及table\_list所指定的过滤，因为核心服务器将自动根据那些选项跳过为被排除的表所返回的命令。不过，起初就避免为被排除的表创建命令当然更好。函数IsImportableForeignTable()可以用来测试一个给定的外部表名是否能通过该过滤器。

如果FDW不支持导入表定义，ImportForeignSchema指针可以被设置为NULL。

## 53.2.9. 并行执行的FDW例程

ForeignScan节点可以选择支持并行执行。一个并行的ForeignScan将在多个进程中被执行并且在相互合作的进程中每一个元组必须只被返回一次。要做到这样，进程可以通过动态共享内存的固定尺寸块来协作。并不保证在每一个进程中这部份共享内存都被映射到相同的地址，因此不能包含指针。下面的函数通常都是可选的，但是如果支持并行执行就必须提供其中的大部分。

```
bool
IsForeignScanParallelSafe(PlannerInfo *root, RelOptInfo *rel,
                          RangeTblEntry *rte);
```

测试一个扫描是否可以在一个并行工作者中被执行。只有当规划器相信可以使用并行计划时才会调用这个函数，如果该扫描在并行工作者中可以安全运行这个函数应该返回真。如果远程数据源具有事务语义，情况通常都不是这样，除非工作者到数据的连接能够以某种方式共享与领导者相同的事务环境。

如果没有定义这个函数，则假定该扫描必须被放置在并行领导者中。注意返回真并不意味着该扫描本身可以被并行完成，只是说明该扫描可以在一个并行工作者中执行。因此，即便当不支持并行执行时，定义这个方法也是有用的。

```
Size
EstimatedDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt);
```

估算并行操作所需的动态共享内存的数量。这可能比实际要用的数量更大，但是绝不能更小。返回值的单位是字节。这个函数是可选的，并且在不需要时可以省略。但是如果它被省略，接下来的三个函数也必须被省略，因为不会为FDW分配共享内存。

```
void
InitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
```

```
void *coordinate);
```

初始化并行操作所需的动态共享内存。coordinate指向一块共享内存区域，其尺寸等于EstimateDSMForeignScan的返回值。这个函数是可选的，并且在不需要时可以省略。

```
void
ReInitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
void *coordinate);
```

当外部扫描计划将要被重新扫描时，重新初始化并行操作所要求的动态共享内存。这个函数是可选的，并且在不需要时可以省略。推荐的措施是这个函数只重置共享状态，而ReScanForeignScan函数仅重置本地状态。当前，这个函数将在ReScanForeignScan之前被调用，但是最好不要依赖于这种顺序。

```
void
InitializeWorkerForeignScan(ForeignScanState *node, shm_toc *toc,
void *coordinate);
```

基于领导者在InitializeDSMForeignScan期间建立的共享状态初始化并行工作者的本地状态。这个函数是可选的，并且在不需要时可以省略。

```
void
ShutdownForeignScan(ForeignScanState *node);
```

在预见到节点将不会被执行完时释放资源。这个函数不会在所有的情况中执行，有时会在没有先调用这个函数之前调用EndForeignScan。由于在这个回调被调用之后并行查询使用的DSM段将被销毁，希望在DMS段消失前采取某种行动的外部数据包装器应该实现这个方法。

## 53. 2. 10. 用于路径重新参数化的FDW例程

```
List *
ReparameterizeForeignPathByChild(PlannerInfo *root, List *fdw_private,
RelOptInfo *child_rel);
```

在把一个由给定子关系child\_rel的最顶层父关系参数化的路径转换成由该子关系参数化的路径时会调用这个函数。该函数被用于重新参数化任意路径或者转化一个ForeignPath的给定fdw\_private成员中保存的任意表达式节点。该回调可能会根据需要使用reparameterize\_path\_by\_child、adjust\_appendrel\_attrs或者adjust\_appendrel\_attrs\_multilevel。

## 53. 3. 外部数据包装器助手函数

多个助手函数被从核心服务器输出，这样外部数据包装器的作者们可以很容易访问到FDW相关对象的属性，例如FDW选项。要使用任何其中一个函数，你需要在你的源文件中包括头文件foreign/foreign.h。这个头也定义了被这些函数返回的结构类型。

```
ForeignDataWrapper *
GetForeignDataWrapperExtended(Oid fdwid, bits16 flags);
```

此功能返回具有给定 `OID` 的外数据包装器的ForeignDataWrapper对象。ForeignDataWrapper对象包含 `FDW` 的属性（详细信息请参阅foreign/foreign.h）。flags是一个 `bitwise-or` d位掩码，指示一组外加的选项集。它可以取值FDW\_MISSING\_OK，在这种情况下NULL 结果将返回给调用方，而不是返回错误给未定义的对象。

```
ForeignDataWrapper *  
GetForeignDataWrapper(Oid fdwid);
```

这个函数为具有给定 OID 的外部数据包装器返回一个ForeignDataWrapper对象。一个ForeignDataWrapper对象包含该FDW的特性（详见foreign/foreign.h）。

```
ForeignServer *  
GetForeignServerExtended(Oid serverid, bits16 flags);
```

此功能返回具有给定 OID 的外数据包装器的ForeignServer对象。ForeignServer对象包含FDW 的属性（详细信息请参阅foreign/foreign.h）。flags是一个 bitwise-or'd位掩码，指示一组外加的选项集。它可以取值FDW\_MISSING\_OK，在这种情况下NULL 结果将返回给调用方，而不是返回错误给未定义的对象。

```
ForeignServer *  
GetForeignServer(Oid serverid);
```

这个函数为一个具有给定 OID 的外部服务器返回ForeignServer对象。一个ForeignServer对象包含该服务器的特性（详见foreign/foreign.h）。

```
UserMapping *  
GetUserMapping(Oid userid, Oid serverid);
```

这个函数为在给定服务器上的给定角色的用户映射返回UserMapping对象（如果指定用户没有映射，它将返回PUBLIC的映射，如果也没有则抛出错误）。一个UserMapping对象包含该用户映射的特性（详见foreign/foreign.h）。

```
ForeignTable *  
GetForeignTable(Oid relid);
```

该函数为一个具有给定 OID 的外部表返回ForeignTable对象。一个ForeignTable对象包含该外部表的特性（详见foreign/foreign.h）。

```
List *  
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

这个函数为一个具有给定外部表 OID 和属性号的列返回针对每一列的FDW选项，形式为一个DefElem列表。如果该列没有选项则返回 NIL。

某些对象类型除了基于OID的查找函数之外，还具有基于名称的查找函数：

```
ForeignDataWrapper *  
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

这个函数为一个具有给定名称的外部数据包装器返回ForeignDataWrapper对象。如果包装器没有找到，在missing\_ok为真时返回 NULL，否则抛出一个错误。

```
ForeignServer *  
GetForeignServerByName(const char *name, bool missing_ok);
```

这个函数为一个具有给定名称的外部服务器返回ForeignServer对象。如果该服务器没有被找到，在missing\_ok为真时返回 NULL，否则抛出一个错误。

## 53.4. 外部数据包装器查询规划

FDW回调函

数GetForeignRelSize、GetForeignPaths、GetForeignPlan、PlanForeignModify、GetForeignJoinPaths、G及PlanDirectModify必须适合LightDB规划器的工作。这里有一些关于它们必须做什么的注记。

root和basere1中的信息可以被用来减少必须从外部表获得的信息量（并且因此降低代价）。basere1->baserestrictinfo是特别有趣的，因为它包含限制条件（WHERE）子句，它应该被用来过滤要被获取的行（FDW本身并不要求强制这些条件，因为核心执行器可以检查它们）。basere1->reltarget->exprs可以被用来决定哪些类需要被获取；但是注意它仅列出了ForeignScan计划节点所发出的列，不包含在条件计算中使用但并不被查询输出的列。

有多个私有域可以给FDW规划函数来保存信息。通常，不管你存储什么在FDW私有域中，它们都应该被palloc，这样它会在规划结束时被回收。

basere1->fdw\_private是一个void指针，它可以被FDW规划函数用来存储与特定外部表相关的信息。核心规划器不会碰它除非当RelOptInfo节点被创建时把它初始化为NULL。它对从GetForeignRelSize传递信息给GetForeignPaths和/或从GetForeignPaths传递信息给GetForeignPlan非常有用，这样避免了重新计算。

GetForeignPaths可以通过在ForeignPath节点的fdw\_private域中存储私有信息来标识不同的访问路径。fdw\_private被声明为一个List指针，但是可能实际上包含任何东西，因为规划器不会触碰它。但是，最好是使用一种nodeToString可导出的形式，这样在后端可以用于调试支持。

GetForeignPlan可以检查选中的ForeignPath节点的fdw\_private域，并且可以生成被放置于ForeignPath计划节点中的fdw\_exprs和fdw\_private列表。这两个列表必须被表示为一种copyObject可复制的形式。fdw\_private列表没有任何其他限制并且不会被核心后端以任何形式解释。非NIL的fdw\_exprs应该包含表达式树，该树会在运行时被执行。这些树将由规划器在后期处理，以便让它们变成完全可执行的。

在GetForeignPlan中，通常被传入的目标列表可以被照样复制到计划节点中。被传入的scan\_clauses列表包含和basere1->baserestrictinfo相同的子句，但是可能为了更好的执行效率会被重新排序。在简单情况下，FDW可以只把RestrictInfo节点从scan\_clauses列表剥离（使用extract\_actual\_clauses）并且把所有子句放到计划节点的条件列表中，这意味着所有子句将在运行时由执行器检查。更复杂的FDW可能可以在内部检查某些子句，着这种情况下哪些子句可以从计划节点的条件列表中删除，这样执行器就不用浪费时间去检查它们。

作为一个例子，FDW可以标识某些foreign\_variable = sub\_expression形式的限制子句，它决定哪些可以使用由sub\_expression给出的本地计算值在远程服务器上被执行。这样一个子句的实际标识应该在GetForeignPaths期间发生，因为它可能会影响路径的代价估计。路径的fdw\_private域可能包括一个已标识的子句的RestrictInfo节点。然后GetForeignPlan将从scan\_clauses中移除该子句，但是将sub\_expression加到fdw\_exprs来保证它被揉成可执行的形式。它可能还将把控制信息放入到计划节点的fdw\_private域来告诉执行函数在运行时要做什么。传递给远程服务器的查询将涉及类似WHERE foreign\_variable = \$1的东西，使用在运行时从fdw\_exprs表达式树获得的参数值。

任何从该计划节点的条件列表移除的子句必须被加入到fdw\_recheck\_qual或者由RecheckForeignScan重新检查以便确保在READ COMMITTED隔离级别的正确行为。当查询中涉及的某个其他表上发生并发更新时，执行器可能需要验证原来的所有条件仍然对该元组满足（可能用一组不同的参数值）。使用fdw\_recheck\_qual通常比在RecheckForeignScan中实现检查要更容易，但是这种方法不足以应付外连接被下推的情况，因为那种情况下的连接元组可能会有一些域具有NULL但是不会导致整个元组被拒绝。

另一个可以由FDW填充的ForeignScan域是fdw\_scan\_tlist，它描述FDW为这个计划节点返回的元组。对于简单的外部表扫描这可以设置为NIL，表示返回的元组具有为外部表声明的行类型。非NIL值必须是一个包含表示返回列的Var或表达式的目标列表（TargetEntry的

列表)。例如，这可以被用来显示 FDW 省略了某些查询不需要的列。还有，如果 FDW 计算表达式比在本地计算代价更低，可以把那些表达式加入到 `fdw_scan_tlist`。注意连接计划（从 `GetForeignJoinPaths` 创建的路径得到）必须总是提供 `fdw_scan_tlist` 来描述它们将返回的列集合。

FDW 应该总是只依靠表的限制子句构建至少一个路径。在连接查询中，它可能还会选择依靠连接子句构建路径，例如 `foreign_variable = local_variable`。这样的子句将不会在 `baserel->baserestrictinfo` 中找到，但是必须出现在关系的连接列表中。使用这样一个子句的路径被称为一个“参数化路径”。它必须用一个合适的 `param_info` 值来标识其他被使用在选中的连接子句中的关系；使用 `get_baserel_parampathinfo` 来计算该值。在 `GetForeignPlan` 中，连接子句的 `local_variable` 部分将被加到 `fdw_exprs` 中，并且接着在运行时和一个普通限制子句一样工作。

如果一个 FDW 支持远程连接，`GetForeignJoinPaths` 应该和 `GetForeignPaths` 对基本表所作的那样为潜在的远程连接产生 `ForeignPath`。有关想要进行的连接的信息可以以上述相同的方式传递给 `GetForeignPlan`。不过，`baserestrictinfo` 与连接关系无关，一个特定连接的相关连接子句将被作为一个独立的参数 (`extra->restrictlist`) 被传递给 `GetForeignJoinPaths`。

FDW 可能会额外地支持直接执行某些在扫描和连接层次之上的计划动作，例如分组或者聚集。为了提供这类选项，FDW 应该生成路径并且把它们插入到合适的上层关系中。例如，一条表示远程聚集的路径应该被使用 `add_path` 插入到 `UPPERREL_GROUP_AGG` 关系中。这条路径的代价将会与通过读取外部关系的简单扫描路径的本地聚集（注意这样一条路径也必须被提供，否则规划时会有错误）进行比较。如果远程聚集路径胜出（通常是这样的），它会被以通常的方式（调用 `GetForeignPlan`）转化成计划。如果该查询的所有基本关系都来自于同一个 FDW，推荐在 `GetForeignUpperPaths` 回调函数中生成这种路径，该函数会为每一个上层关系被调用（即每一次扫描/连接后处理步骤）。

Section 53.2.4 中描述的 `PlanForeignModify` 以及其他回调的设计是建立在这样一个假设之上：外部表将以通常的方式被扫描并且行更新将被一个本地 `ModifyTable` 计划节点所驱动。这种方法对于更新需要读取本地表以及外部表的一般情况下是必要的。不过，如果操作可以完全由外部服务器执行，FDW 可以产生一个表示这种操作的计划并且把它插入到 `UPPERREL_FINAL` 上层关系中，在其中它会与 `ModifyTable` 方法竞争。这种方法还可以被用来实现远程 `SELECT FOR UPDATE`，而不使用 Section 53.2.5 中描述的行锁定回调。记住插入到 `UPPERREL_FINAL` 中的路径负责实现查询的所有行为。

在规划一个 `UPDATE` 或 `DELETE` 时，`PlanForeignModify` 和 `PlanDirectModify` 能为外部表查找 `RelOptInfo` 结构，并利用之前由扫描规划函数创建的 `baserel->fdw_private` 数据。但是，在 `INSERT` 中目标表不会被扫描，因此不会有它的 `RelOptInfo`。由 `PlanForeignModify` 返回的 `List` 具有和 `ForeignScan` 计划节点的 `fdw_private` 列表相同的限制，即它必须只包含 `copyObject` 知道怎么拷贝的结构。

带有一个 `ON CONFLICT` 子句的 `INSERT` 不支持指定冲突目标，因为本地不知道远程表上的唯一约束和排除约束的情况。然后这也意味着 `ON CONFLICT DO UPDATE` 不被支持，因为该说明是强制性的。

## 53.5. 外部数据包装器中的行锁定

如果一个 FDW 的底层存储机制具有锁定行的概念来阻止对行的并发更新，通常值得 FDW 去执行行级锁定以尽可能接近在普通 `LightDB` 表中所实际使用的语义。涉及这个问题有多种考虑。

要做出一个关键决定是执行早期锁定还是晚期锁定。在早期锁定中，当一行被第一次从底层存储中检索到时，它会被锁定；而在晚期锁定中，只有当行需要被锁定时才锁定它（由于某些行可能被本地检查的限制或者连接条件抛弃，所以会出现不同）。早期锁定更加简单并且能避免额外地与远程存储交互，但是可能会导致一些不需要锁定的行也被锁定，最终造成并发性下降甚至意外的死锁。还有，只有在要被锁定的行可以在后期唯一地重新标识时才可以用晚期锁定。较好的行标识符应该能标识行的特定版本，就像 `LightDB TID` 那样。

默认情况下，LightDB在与 FDW 交互时会忽略锁定考虑，但是 FDW 可以在没有核心代码显式支持的情况下执行早期锁定。Section 53.2.5 中描述的 API 函数（在LightDB 9.5 中加入）允许 FDW 按照意愿使用晚期锁定。

一个额外的考虑是在READ COMMITTED隔离模式中，LightDB可能需要对某个目标元组的更新版本进行限制以及连接条件的重新检查。重新检查连接条件要求重新获得之前连接成目标元组的非目标行拷贝。在标准LightDB表的情况下，这可以通过在连接投影出的列列表中包括非目标表的 TID 并且在需要时重新取得非目标行来做到。这种方法可以让连接数据集保持紧凑，但是它要求代价较低的重新取得元组的功能，还有 TID 要能够唯一地标识要被重新取得的行版本。因此，默认情况下用于外部表的方法是将整个外部表元组的拷贝包括在从连接投影出的列列表中。这不会对 FDW 有特殊的要求，但是会导致归并和哈希连接性能下降。要满足重新取得元组需求的 FDW 可以选择第一种方式。

对于在外部表上的UPDATE或者DELETE，推荐目标表上的ForeignScan操作在它取得的行上执行早期锁定（可能通过SELECT FOR UPDATE的等效体）。通过在规划时比较一个表的 relid 和root->parse->resultRelation或在执行时使用ExecRelationIsTargetRelation()，一个 FDW 可以检测该表是否为UPDATE/DELETE的目标。另一种可能性是在ExecForeignUpdate或者ExecForeignDelete回调中执行晚期锁定，但是对此没有特别的支持。

对于通过SELECT FOR UPDATE/SHARE命令指定要被锁定的外部表，ForeignScan操作同样可以通过用SELECT FOR UPDATE/SHARE的等效体取元组来执行早期锁定。要执行晚期锁定，请提供Section 53.2.5 中定义的回调函数。在GetForeignRowMarkType中，根据请求的锁长度来选择行标记选项ROW\_MARK\_EXCLUSIVE、ROW\_MARK\_NOKEYEXCLUSIVE、ROW\_MARK\_SHARE或者ROW\_MARK\_KEYSHARE（不管选择哪一种选项，核心代码都会做同样的事情）。在别的地方，可以在规划时用get\_plan\_rowmark或者在执行时用ExecFindRowMark来检测一个外部表是否被指定由这种类型的命令锁定。你必须不仅仅检测是否返回了一个非空的行标记结构，还要检测它的strength域不是LCS\_NONE。

最后，对于在UPDATE、DELETE或者SELECT FOR UPDATE/SHARE命令中使用但是没有被指定要行锁定的外部表，你可以在看到锁长度LCS\_NONE时通过使用GetForeignRowMarkType选择选项ROW\_MARK\_REFERENCE来把默认选择覆盖为拷贝整个行。这将导致用那个值作为markType来调用RefetchForeignRow。它应该接着重新取得该行而不获取任何新锁（如果你有一个GetForeignRowMarkType函数，但是不想重新取未锁定的行，可为LCS\_NONE选择选项ROW\_MARK\_COPY）。

更多信息可见src/include/nodes/lockoptions.h，以及src/include/nodes/plannodes.h中RowMarkType和PlanRowMark的注释，还有src/include/nodes/execnodes.h中ExecRowMark的注释。

---

## Chapter 54. 编写一种表采样方法

LightDB的TABLESAMPLE子句实现支持在SQL标准要求的BERNOULLI和SYSTEM方法之外自定义表采样方法。采样方法决定了使用TABLESAMPLE子句时表的哪些行会被选择。

在SQL层上，一种表采样方法被表达为一个SQL函数（通常用C实现），其签名是：

```
method_name(internal) RETURNS tsm_handler
```

函数的名称是出现在TABLESAMPLE子句中的同一个方法名称。internal参数是不起作用的（总是值为零），它仅仅是为了阻止直接从SQL命令中调用该函数。函数的结果必须是一个palloc好的TsmRoutine结构，它包含了该采样方法的支持函数的指针。这些支持函数是纯C函数并且对于SQL层面不可见也不可调用。支持函数见Section 54.1

除了函数指针之外，TsmRoutine结构必须提供这些额外的域：

```
List *parameterTypes
```

这是一个OID列表，它包含了使用这种采样方法时TABLESAMPLE子句接受的参数的数据类型OID。例如，对于内建方法，这个列表只包含一个值为FLOAT4OID的项，它表示采样的百分数。自定义采样方法可以有更多或者不同的参数。

```
bool repeatable_across_queries
```

如果为true，当每次查询时给出相同的参数和REPEATABLE种子值且表内容没有改变时，采样方法可以在连续的查询中给出相同的采样。当这个域为false时，不能把REPEATABLE子句用于这种采样方法。

```
bool repeatable_across_scans
```

如果为true，这种采样方法在同一个查询的连续扫描中给出相同的采样（假定参数、种子值和快照都不变）。当这个域为false时，规划器将不会选择要求扫描被采样表多于一次的计划，因为那会导致不一致的查询输出。

TsmRoutine结构类型被声明在src/include/access/tsmapi.h中，需要更多细节可以参考该文件。

在尝试编写自己的采样方法时，包括在标准发布中的表采样方法是很好的参考。内建采样方法的源代码树可见src/backend/access/tablesample子目录，在contrib子目录中可以找到额外的方法。

### 54.1. 采样方法支持函数

TSM处理器函数返回一个palloc好的TsmRoutine结构，其中包含了下文所述的支持函数的指针。大部分函数是必须的，但是有些是可选的（它们的指针可以为NULL）。

```
void  
SampleScanGetSampleSize (PlannerInfo *root,  
                          RelOptInfo *baserel,  
                          List *paramexprs,  
                          BlockNumber *pages,  
                          double *tuples);
```

这个函数在规划期间被掉欧勇。它必须估计在一次采样扫描中会被读到的关系页面数，以及将被该扫描所选择的元组数（例如，可以先估计采样分数，乘上baserel->pages和baserel->tuples数，并且把结果圆整）。paramexprs列表保存作为



TABLESAMPLE子句的参数的表达式。如果出于优化的目的需要 这些表达式的值，我们推荐使用estimate\_expression\_value() 来尝试将这些表达式变成常量。但是即便这些表达不能简化，该函数必须提供 估计的尺寸，并且即使出现不合法的值它也不应该失败（记住它们只是运行时值 的估计）。pages和tuples参数是输出。

```
void
InitSampleScan (SampleScanState *node,
                int eflags);
```

为 SampleScan 计划节点的执行进行初始化。这会在执行器启动时被调用。 它应该执行执行处理启动所需的任何初始化工作。 SampleScanState节点已经被创建，但是它的tsm\_state域为 NULL。 InitSampleScan函数可以 palloc 任何采样方法需要的内部状态数据，并且把它的一个指针存储在node->tsm\_state 中。有关要扫描的表的信息可以通过SampleScanState 节点的其他域访问（但是要注意 node->ss.ss\_currentScanDesc扫描描述符还没有被设置）。 eflags包含描述这个计划节点的执行器操作模式的标志位。

当(eflags & EXEC\_FLAG\_EXPLAIN\_ONLY)为真时，该 扫描将不会被真正执行，因此这个函数应该只做最少的事情，让该节点状态对 EXPLAIN和EndSampleScan可用。

这个函数可以被省略（把指针设置为 NULL），在那种情况下 BeginSampleScan必须执行采样方法所需的所有初始化工作。

```
void
BeginSampleScan (SampleScanState *node,
                 Datum *params,
                 int nparams,
                 uint32 seed);
```

开始执行一次采样扫描。就在第一次尝试取得一个元组时就会调用这个函数， 如果该扫描需要被重启可能还要再次调用它。有关要扫描的表的信息可以通过 SampleScanState节点的其他域访问（但是要注意 node->ss.ss\_currentScanDesc扫描描述符还没有被设置）。 长度为nparams的params数组包含在 TABLESAMPLE子句中提供的参数的值。这些参数的编号和类型 在采样方法的parameterTypes列表中指定，并且已经被 检查过不为空。seed包含用于在采样方法中生成任何随机数的 种子。如果给定了REPEATABLE值，种子将是该值的哈希。如果 没有指定REPEATABLE值，种子将是random()的 结果。

这个函数可能会调整域node->use\_bulkread 以及node->use\_pagemode。 如果node->use\_bulkread为true（默认）， 该扫描将使用一种鼓励重用缓冲区的缓冲区访问策略。如果该扫描只访问 该表的页面的一小部分，将这个域设置为false比较合理。 如果node->use\_pagemode为true（默认）， 那么对于每一个被访问的页面上的所有元组，该扫描将会在一趟中执行它们的 可见性检查。如果该扫描只选择每个被访问页面上的一小部分，将这个域 设置为false比较合理。这将导致执行较少次的元组可见性检查， 但是每一次都会代价更大，因为需要更多锁定。

如果采样方法被标记为repeatable\_across\_scans，在重 新扫描时，它必须能够选择和第一次扫描相同的元组集合，也就是说对 BeginSampleScan的一次新调用必须选择和之前相同的元组（如果TABLESAMPLE参数和种子没有变化）。

```
BlockNumber
NextSampleBlock (SampleScanState *node, BlockNumber nblocks);
```

返回下一个要扫描的页面的块号，如果没有剩余的页面需要扫描则返回 InvalidBlockNumber。

这个函数可以被省略（设置指针为 NULL），在那种情况下核心代码将 执行整个关系的一次顺序扫描。这样一个扫描可以使用同步扫描，这样 采样方法不能假定每一次扫描都用同样的顺序访问关系页面。

```
OffsetNumber  
NextSampleTuple (SampleScanState *node,  
                 BlockNumber blockno,  
                 OffsetNumber maxoffset);
```

返回指定页面上下一个要被采样的元组的偏移量，如果没有剩余元组需要被采样，则返回InvalidOffsetNumber。maxoffset是页面上使用的最大偏移量。

### Note

NextSampleTuple没有被显式地告知范围 1 .. maxoffset中的哪些偏移量真正包含了合法的元组。这通常不成问题，因为核心代码会忽略采样丢失或者不可见元组的请求。这不会导致采样中的任何偏差。不过，如果必要，该函数可以用node->donetuples来检查它返回的元组有多少是合法并且可见的。

### Note

NextSampleTuple不能假定 blockno是最近一次NextSampleBlock调用返回的同一个页面号。它由之前某次NextSampleBlock调用所返回，但是核心代码被允许在真正扫描页面之前调用NextSampleBlock，以便支持预取。假定一旦一个给定页面的采样开始，连续的 NextSampleTuple调用都将引用同一个页面（直到返回 InvalidOffsetNumber）。

```
void  
EndSampleScan (SampleScanState *node);
```

结束扫描并且释放资源。释放 palloc 过的内存通常并不重要，但是任何外部可见的资源应该被清除。在没有这类资源存在的通常情况下，这个函数可以被省略（设置指针为NULL）。

---

## Chapter 55. 编写一个自定义扫描提供者

LightDB支持一组实验性的功能，它们的目的是允许扩展模块向系统中增加新的扫描类型。与外部数据包装器不同（只负责给出如何扫描其自身的外部表的知识），自定义扫描提供者可以提供另一种扫描系统中任一关系的方法。通常，编写一个自定义扫描提供者的动机是允许使用某种核心系统不支持的优化，例如缓冲或者某种形式的硬件加速。这一章简要介绍了如何编写一个新的自定义扫描提供者。

实现一个新类别的自定义扫描分成三步。首先，在规划期间需要生成表达使用所提出策略的扫描的访问路径。然后，如果规划器选择这些访问路径之一作为最优策略来扫描一个特定关系，该访问路径必须被转换成计划。最后，必须能执行该计划并且产生和其他以同一关系为目标的访问路径相同的结果。

### 55.1. 创建自定义扫描路径

一个自定义扫描提供者将通过设置下面的钩子函数来为基本关系增加路径，在核心代码已经为该关系产生了所有访问路径集后（除了在此调用之后生成的Gather路径，以便它们可以使用被钩子添加的部分路径），这个钩子函数将被调用。

```
typedef void (*set_rel_pathlist_hook_type) (PlannerInfo *root,
                                           RelOptInfo *rel,
                                           Index rti,
                                           RangeTblEntry *rte);

extern PGDLLIMPORT set_rel_pathlist_hook_type set_rel_pathlist_hook;
```

尽管这个钩子函数可被用来检查、修改或者移除核心系统产生的路径，自定义扫描提供程序通常还是局限于产生CustomPath对象并且使用add\_path把它们加入到rel中。自定义扫描提供者负责初始化CustomPath对象，它被声明为这样：

```
typedef struct CustomPath
{
    Path      path;
    uint32    flags;
    List      *custom_paths;
    List      *custom_private;
    const CustomPathMethods *methods;
} CustomPath;
```

path必须像任何其他路径一样被初始化，包括行计数估计、启动和总代价以及这条路径提供的排序顺序。flags是一个位掩码，如果该自定义路径能够支持反向扫描则它应该包括CUSTOMPATH\_SUPPORT\_BACKWARD\_SCAN，如果支持标记和恢复则应该包括CUSTOMPATH\_SUPPORT\_MARK\_RESTORE。这两种能力都是可选的。可选的custom\_paths域是由这个自定义路径节点使用的Path节点的列表，这些将被规划器转换成Plan节点。custom\_private可被用来存储该自定义路径的私有数据。私有数据应该被存储为能被nodeToString处理的形式，这样尝试打印该自定义路径的调试例程才能正常工作。methods必须指向一个实现了所需自定义路径方法的对象（通常是静态分配的），当前只有一种方法。

一个自定义扫描提供者还能提供连接路径。就和基本关系一样，这样一条路径也应该产生和它将要替换的连接所产生的相同的输出。要做到这一点，连接提供程序应该设置下面的钩子函数，并且在该钩子函数里为连接关系创建CustomPath路径。

```
typedef void (*set_join_pathlist_hook_type) (PlannerInfo *root,
                                             RelOptInfo *joinrel,
```

```

RelOptInfo *outerrel,
RelOptInfo *innerrel,
JoinType jointype,
JoinPathExtraData *extra);

extern PGDLLIMPORT set_join_pathlist_hook_type set_join_pathlist_hook;

```

对于同一个连接关系，这个钩子将被反复调用，因为要对不同的内外关系组合生成路径，所以如何最小化可能的重复工作是钩子函数的责任。

### 55.1.1. 自定义扫描路径回调

```

Plan *(*PlanCustomPath) (PlannerInfo *root,
                          RelOptInfo *rel,
                          CustomPath *best_path,
                          List *tlist,
                          List *clauses,
                          List *custom_plans);

```

将一条自定义路径转换为一个完成的计划。返回值通常将是一个CustomScan对象，回调函数必须负责分配并且初始化这个对象。详见Section 55.2

## 55.2. 创建自定义扫描计划

以一棵已完成的计划树表示的自定义扫描使用下面的结构：

```

typedef struct CustomScan
{
    Scan        scan;
    uint32      flags;
    List        *custom_plans;
    List        *custom_exprs;
    List        *custom_private;
    List        *custom_scan_tlist;
    Bitmapset   *custom_relids;
    const CustomScanMethods *methods;
} CustomScan;

```

scan必须和任何其他扫描一样被初始化，包括估计代价、目标列表、条件等等。flags是一个位掩码，它的含义和CustomPath中的一样。custom\_plans可以用来存储子Plan节点。custom\_exprs应该被用来存储需要由setrefs.c和subselect.c修整的表达式树，而custom\_private应该被用来存储其他只由自定义扫描提供者本身使用的私有数据。在扫描一个基本关系时，custom\_scan\_tlist可以为 NIL，表示该自定义扫描返回符合该基本关系行类型的扫描元组。否则，它是一个描述实际扫描元组的目标列表。对于连接必须提供custom\_scan\_tlist。如果自定义扫描提供者能够计算某些非-Var 表达式，也应该提供这个域的值。custom\_relids会被核心代码设置成这个扫描节点要处理的关系的集合（范围表索引）。当这个扫描被放在一个链接上时是一种例外，那时其中只有一个成员。methods必须指向一个实现了所需自定义扫描方法的对象（通常是静态分配的），将进一步在下文详细介绍。

当一个CustomScan扫描单个关系时，scan.scanrelid必须是被扫描的表的范围表索引。当它替代的是一个连接时，scan.scanrelid应该为零。

计划树必须能够被使用copyObject复制，因此所有存储在“custom”域中的数据必须由该函数能处理的节点构成。更进一步，自定义扫描提供者不能把CustomScan结构本身替换成包含CustomScan的更大的结构（就好像CustomPath或者CustomScanState）。

## 55.2.1. 自定义扫描计划回调

```
Node *(*CreateCustomScanState) (CustomScan *cscan);
```

为这个CustomScan分配一个CustomScanState。实际的分配常常会比一个普通CustomScanState所要求的空间要大，因为很多提供者希望把它嵌入在一个更大的结构中作为第一个域。返回的值必须有节点标签并且设置好了合适的methods，不过在这个阶段其他域应该被设置为零。在ExecInitCustomScan执行基本的初始化之后，将调用BeginCustomScan回调函数来让自定义扫描提供者有机会做其他需要做的事情。

## 55.3. 执行自定义扫描

在执行一个CustomScan时，它的执行状态由一个CustomScanState表示，其定义如下：

```
typedef struct CustomScanState
{
    ScanState ss;
    uint32    flags;
    const CustomExecMethods *methods;
} CustomScanState;
```

ss和任何其他扫描状态一样被初始化，不过如果该扫描是用于连接而不是基本关系，则ss.ss\_currentRelation会被留成NULL。flags是一个位掩码，它的含义与CustomPath和CustomScan中的一样。methods必须指向一个实现了所需自定义扫描状态方法的对象（通常是静态分配的），将进一步在下文详细介绍。通常一个CustomScanState（不需要支持copyObject）实际将是一个较大的结构，上面的结构将嵌入在其中作为第一个成员。

### 55.3.1. 自定义扫描执行回调

```
void (*BeginCustomScan) (CustomScanState *node,
                        EState *estate,
                        int eflags);
```

完成所提供的CustomScanState的初始化。标准的域已经被ExecInitCustomScan初始化，但是任何私有的域应该在这里被初始化。

```
TupleTableSlot *(*ExecCustomScan) (CustomScanState *node);
```

取下一个扫描元组。如果还有任何元组剩余，它应该用当前扫描方向的下一个元组填充ps\_ResultTupleSlot，并且接着返回该元组槽。如果没有，则用NULL填充或者返回一个空槽。

```
void (*EndCustomScan) (CustomScanState *node);
```

清除任何与CustomScanState相关的私有数据。这个方法是必需的，但是如果没有任何相关的数据或者相关数据将被自动清除，则它不需要做任何事情。

```
void (*ReScanCustomScan) (CustomScanState *node);
```

把当前扫描倒回到开始处，并且准备重新扫描该关系。

```
void (*MarkPosCustomScan) (CustomScanState *node);
```

保存当前的扫描位置，这样可以在以后由RestrPosCustomScan回调函数恢复。这个回调函数是可选的，只有在CUSTOMPATH\_SUPPORT\_MARK\_RESTORE标志被设置时才需要提供。

```
void (*RestrPosCustomScan) (CustomScanState *node);
```

恢复由MarkPosCustomScan回调函数保存的扫描位置。这个回调函数是可选的，只有在CUSTOMPATH\_SUPPORT\_MARK\_RESTORE标志被设置时才需要提供。

```
Size (*EstimatedDSMCustomScan) (CustomScanState *node,
                                  ParallelContext *pcxt);
```

估计并行操作所需要的动态共享内存的数量。这可能会比实际使用的量更大，但是绝不能更低。返回值的单位是字节。这个回调是可选的，只有在这个自定义扫描提供者支持并行执行时才必须提供这个回调。

```
void (*InitializeDSMCustomScan) (CustomScanState *node,
                                  ParallelContext *pcxt,
                                  void *coordinate);
```

初始化并行操作所需的动态共享内存。coordinate 指向一块大小等于EstimateDSMCustomScan 返回值的共享内存区域。这个回调是可选的，只有在这个自定义扫描提供者支持并行执行时才必须提供这个回调。

```
void (*ReInitializeDSMCustomScan) (CustomScanState *node,
                                    ParallelContext *pcxt,
                                    void *coordinate);
```

当自定义扫描计划节点即将被重新扫描时，重新初始化并行操作所需的动态共享内存。这个回调是可选的，只有在这个自定义扫描提供者支持并行执行时才必须提供这个回调。推荐的做法是，此回调仅重置共享状态，而ReScanCustomScan 回调仅重置本地状态。目前，该回调将在ReScanCustomScan 之前调用，但最好不要依赖该顺序。

```
void (*InitializeWorkerCustomScan) (CustomScanState *node,
                                      shm_toc *toc,
                                      void *coordinate);
```

基于InitializeDSMCustomScan期间通过领导者 设置的共享状态初始化并行工作者的本地状态。这个回调是可选的，只有在这个自定义扫描提供者支持并行执行时才必须提供这个回调。

```
void (*ShutdownCustomScan) (CustomScanState *node);
```

预计节点将不会执行完成时释放资源。并不是在所有情况下都调用；有时，EndCustomScan可能会在调用此函数之前调用。由于并行查询使用的DSM段在调用此回调后即被销毁，因此希望在DSM段消失之前采取某些操作的自定义扫描提供程序应实现此方法。

```
void (*ExplainCustomScan) (CustomScanState *node,
                            List *ancestors,
                            ExplainState *es);
```

为一个自定义扫描计划节点的EXPLAIN输出额外的信息。这个回调函数是可选的。即使没有这个回调函数，被存储在ScanState中的公共的数据（例如目标列表和扫描关系）也将被显示，但是该回调函数允许显示额外的信息（例如私有状态）。

---

# Chapter 56. 遗传查询优化器

作者

Written by Martin Utesch (<utesch@aut.tu-freiberg.de>) for the Institute of Automatic Control at the University of Mining and Technology in Freiberg, Germany.

## 56.1. 将查询处理看成是一个复杂的优化问题

在所有关系操作符中，最难于处理和优化的是连接。可能的查询计划数目以查询中连接数量的指数增长。对各种各样处理独立连接的方法（如LightDB中的嵌套循环、哈希连接、归并连接）和多种关系访问方法的indexes（如LightDB中的B树、哈希、GiST和GIN）的支持也进一步加重了优化的负担。

通常的LightDB查询优化器会执行一次在可选策略空间上的近似穷举搜索。这个算法最早由IBM的系统R数据库引入，它能产生接近最优的连接顺序，但是当查询中的连接数增长到很大时，该算法需要大量的时间和内存空间。这使得普通的LightDB查询优化器不适合需要连接大量表的查询。

德国弗莱堡的矿业大学自动控制学院在使用LightDB作为电力网络维护决策支持知识系统的后端时遇到了一些问题。DBMS需要为知识系统中的推理机器处理大量连接查询。这些查询中的连接数不可能用普通的查询优化器来处理。

接下来我们将介绍一种遗传算法的实现，它被用来以一种更有效率的方式为涉及大量连接的查询解决连接顺序问题。

## 56.2. 遗传算法

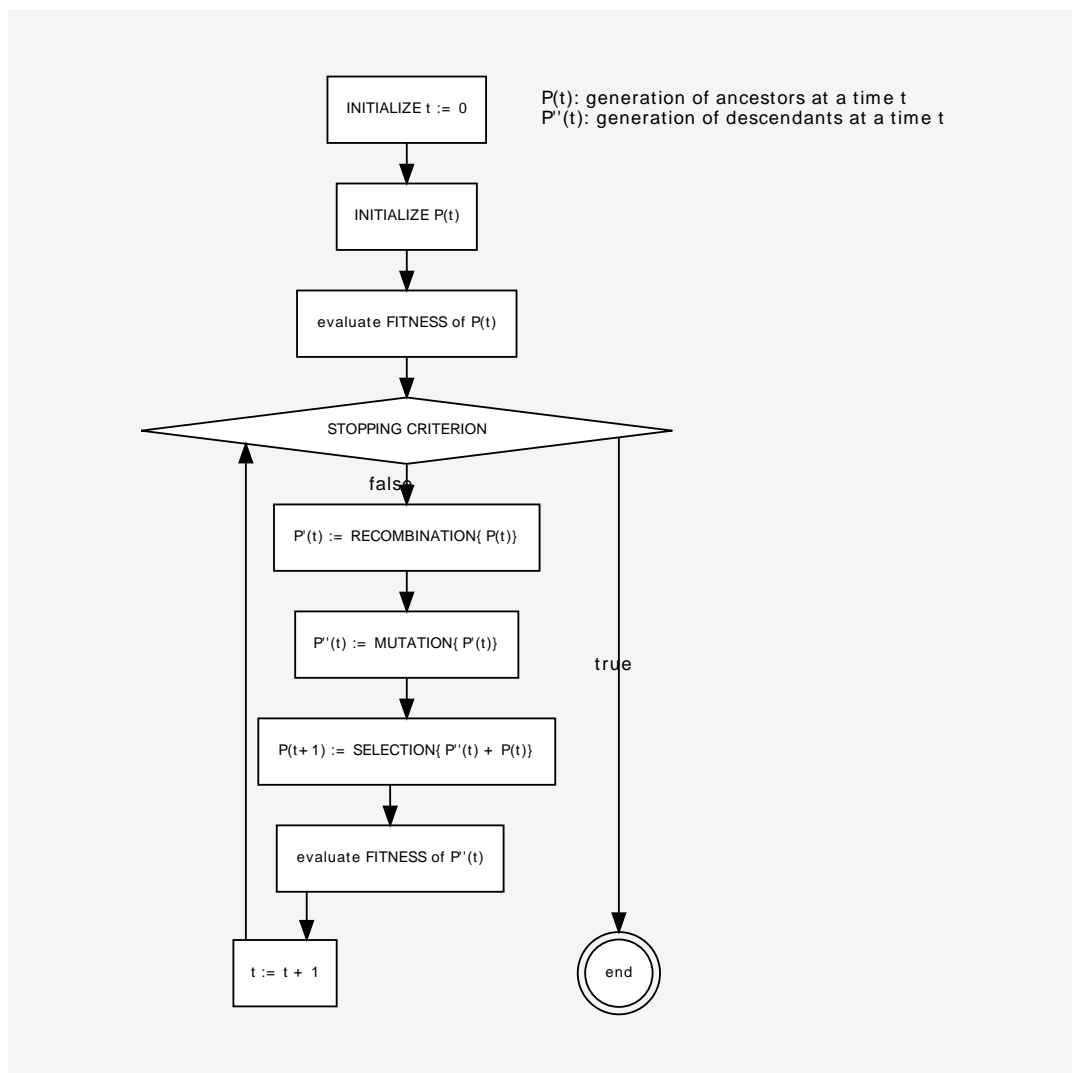
遗传算法（GA）是一种通过随机搜索操作的启发式优化方法。优化问题的可能的解决方案集合被看成是个体的种群。一个个体对于它的环境的适应程度由其适应度指定。

搜索空间中一个个体的坐标被表示为染色体，实质上是一个字符串集合。一个基因是一个染色体的一个片段，它编码了一个要被优化参数的值。一个基因的典型编码包括二进制或整数。

通过对重组、变异和选择的模拟，比父辈平均适应度更好的新一代搜索点将被找到。Figure 56.1说明了这些步骤。



Figure 56.1. 遗传算法的结构



根据comp.ai.genetic FAQ, GA并非是一个纯粹的随机搜索。GA会使用随机处理,但是结果的确是非随机的(比随机好)。

### 56.3. LightDB 中的遗传查询优化 (GEQO)

GEQO模块把查询优化问题当做著名的货郎担问题 (TSP) 来处理。可能的查询计划被编码为整数的串。每一个串表示从查询中一个关系到下一个关系的连接顺序。例如, 连接树

```

  /\
 /\ 2
 /\ 3
4 1

```

被编码为整数串 '4-1-3-2', 它表示首先连接关系 '4' 和 '1', 然后连接 '3', 最后连接 '2'。这里 1、2、3、4 是LightDB优化器中的关系 ID。

LightDB中GEQO实现的特点有:

- 一种稳态 GA (在种群中替换适应度最差的个体, 而不是整代替换) 的使用允许对改进的查询计划快速收敛。这对在合理时间内处理查询是最重要的;
- 边重组杂交的使用特别适合于通过GA为TSP的解决方案保持低丢边率;

- 遗传操作符变异被废弃，这样不需要修补机制来产生合法的TSP旅行。

GEQO模块的一部分是从 D. Whitley 的遗传算法中改编而来。

GEQO模块允许LightDB查询优化器支持通过非穷举搜索高效地处理大量连接的查询。

### 56.3.1. 用GEQO产生可能的计划

GEQO规划处理使用标准的规划器代码来产生用于扫描个体关系的计划。然后使用遗传方法发展连接计划。如上所示，每一个候选连接计划被表示为一个连接基本关系的序列。在初始阶段，GEQO代码简单地随机产生某些可能的连接序列。对于被考虑的每一个连接序列，标准规划器代码被调用来估算使用该序列执行查询的代价（对于连接序列的每一步，所有三种连接策略都被考虑；并且所有初始决定的关系扫描计划都可用。估计的代价是这些可能性中最低的那个。）。具有较低估计代价的连接序列被认为比具有较高代价的“更适合”。遗传算法会丢弃最不适宜的候选。然后通过组合更适合的候选的基因来产生新的候选——即使从已知代价低的连接序列随机选择片段来创建用于考虑的新序列。这个处理将被重复，直到已经考虑的连接序列的数量达到一个预设值。然后在搜索中任何时候找到的最好的一个将被用来产生最终的计划。

由于在初始种群选择和后续最佳候选的“变异”过程中都采用了随机选择，所以这种处理天生就是非确定性的。要避免被选中计划发生出乎意料的变化，每次 GEQO 算法的运行都会使用当前geqo\_seed参数设置来重启它的随机数生成器。只要geqo\_seed以及其他 GEQO 参数保持固定（以及其他规划器输入，如统计信息），对一个给定的查询将产生相同的计划。要试验不同的搜索路径，可以尝试改变geqo\_seed。

### 56.3.2. LightDB GEQO的未来实现任务

仍需对改进遗传算法的参数设置做一些工作。在文件src/backend/optimizer/geqo/geqo\_main.c、例程gimme\_pool\_size和gimme\_number\_generations中，我们必须为参数设置找到一种折中来满足两个互相竞争的需求：

- 查询计划的最优性
- 计算时间

在当前的实现中，每一个候选连接序列的适应度通过运行标准规划器的连接选择和代价估计代码从零估算而来。即使不同的候选中使用了相似的连接子序列，还是需要重复大量的工作。通过保留子连接的代价估计可以在这种情况下节省很多时间。但问题在于要避免为了保留那样的状态而不合理地占用过多内存。

在更基础的层面上，用一个为 TSP 设计的 GA 算法来解决查询优化问题是否合适也还需要探讨。在 TSP 情况中，与任何子串（部分旅程）相关的代价独立于剩余的旅程，但是对于查询优化肯定不是这样。因此边重组杂交是否比变异过程更有效还存有疑问。

## 56.4. 进一步阅读

下列资源包含关于遗传算法更多的信息：

- The Hitch-Hiker’s Guide to Evolutionary Computation<sup>1</sup>, (FAQ for news://comp.ai.genetic)
- Evolutionary Computation and its application to art and design<sup>2</sup>, by Craig Reynolds
- [elma04]
- [fong]

<sup>1</sup> <http://www.faqs.org/faqs/ai-faq/genetic/part1/>

<sup>2</sup> <https://www.red3d.com/cwr/evolve.html>

---

## Chapter 57. 表访问方法接口定义

本章阐述LightDB核心系统与表访问方法间的接口，其管理表的存储。除这里指出的内容，核心系统对于这些访问方法知之甚少，因此可以通过编写附加代码来开发全新的访问方法类型。

每个表访问方法都有pg\_am系统目录中的一行来描述。pg\_am条目为表访问方法指定一个名字和句柄函数。这些条目可以用SQL命令CREATE ACCESS METHOD和DROP ACCESS METHOD来创建和删除。

表访问方法句柄函数必须声明成接受单个internal类型的参数，并返回伪类型table\_am\_handler。该参数是伪值，仅用于防止从SQL命令直接调用句柄函数。函数的结果必须是指向类型TableAmRoutine的结构指针，其包含了核心代码所需要知道的所有内容，以利用表访问方法。返回值需具有服务器生命周期，通常是通过将其定义为一个全局范围的static const变量来实现。TableAmRoutine结构，也称为访问方法的API结构，定义使用回调的访问方法的行为。这些回调是普通C函数指针，在SQL级别并不可见或者可调用。所有回调及其行为在TableAmRoutine结构中定义（结构内部有定义回调需求的注释）。大多数回调有包装函数，从用户（而不是实现者）视点记录表访问方法。详细信息请参考src/include/access/tableam.h<sup>1</sup>文件。

为了实现访问方法，实现者通常要实现特定AM类型的元组表槽（见src/include/executor/tuptable.h<sup>2</sup>），允许访问方法之外的代码保存对AM元组的引用，并且能访问元组的列。

目前AM存储数据的实际方式完全没有限制。例如，可以使用postgres的共享缓冲区的缓存，但这并非必需的。如果使用它，使用Section 64.6中所述的LightDB的标准页面布局可能很有意义。

当前表访问方法API的一个相当大的限制是，如果AM要支持修改和/或索引，每个元组都需要有由块号和项目号组成的元组标识符（TID）（见Section 64.6。TIDs的子部分不一定具有与堆相同的含义，但如果需要位图扫描支持（可选），则块号需要提供局部性。

为安全起见，AM可使用postgres的WAL或自定义实现。如果选用了WAL，则可使用通用WAL记录或实现新型的WAL记录。通用WAL记录较简单，但意味着WAL量更高。目前实现新型WAL记录需要修改核心代码（特别是src/include/access/rmgrlist.h）。

要以允许在单个事务中访问不同表访问方法的方式实现事务支持，可能需要与src/backend/access/transam/xlog.c中的机制紧密集成。

新的表访问方法的开发人员可参考src/backend/access/heap/heapam\_handler.c中已有的堆实现代码，以了解其实现的细节。

---

<sup>1</sup> <https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/include/access/tableam.h;hb=HEAD>

<sup>2</sup> <https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/include/executor/tuptable.h;hb=HEAD>

---

## Chapter 58. 索引访问方法接口定义

本章定义核心LightDB系统和管理个别索引类型的索引访问方法之间的接口。除了在这里指定的内容之外，核心系统对索引一无所知，因此可以通过编写附加代码来开发全新的索引类型。

LightDB中所有的索引在技术上都叫做二级索引。也就是说，索引在物理上与它描述的表文件分离。每个索引被存储为它自己的物理关系并且被pg\_class目录中的一个项所描述。一个索引的内容完全受到其索引访问方法控制。实际上，所有索引访问方法都把索引划分成标准大小的页面，这样它们就可以使用常规的存储管理器和缓冲区管理器来访问索引内容（所有现有的索引访问方法还使用Section 64.6中描述的标准页面布局，并且大部分都使用相同的索引元组头部格式；但是这些决定都不是强制在访问方法上的）。

索引实际上是一些数据键值与索引父表中行版本（元组）的元组标识符或TIDs之间的映射。一个TID由一个块号和一个块内的项编号组成（见Section 64.6）。这对于从表中取一个特定行就足够了。索引并不直接知道在MVCC下，同一个逻辑行可能有多个现存版本；对于索引而言，每个行都是一个独立的对象，都需要自己的索引项。因此，对一行的更新总是为该行创建全新的索引项，即使键值没有改变（HOT元组对这段陈述来说是个异常，但是索引也不会处理这些）。死亡元组的索引项将在随着死亡元组的回收而被回收（通过清理）。

### 58.1. 索引的基本 API 结构

每一个索引访问方法都由pg\_am系统目录中的一行所描述。pg\_am项为该索引访问方法指定了名称和一个处理器函数。这些项可以用CREATE ACCESS METHOD和DROP ACCESS METHOD SQL命令创建和删除。

一个索引访问方法的处理器函数必须被声明为接受单一的类型为internal类型的参数并且返回伪类型index\_am\_handler。该参数是一个无用值，它只是被用来防止从SQL命令直接调用处理器函数。该函数的结果必须是一个已经palloc过的IndexAmRoutine类型结构，它包含核心代码使用该索引访问方法所需的所有信息。IndexAmRoutine结构（也被称为访问方法的API结构）中的域指定了该访问方法的各种固定性质，例如它是否支持多列索引。更重要的是，它包含用于该访问方法的支持函数的指针，这些函数会完成真正访问索引的工作。这些支持函数是纯C函数，并且在SQL层面不可见也不可调用。支持函数在Section 58.2中介绍。

IndexAmRoutine结构定义如下：

```
typedef struct IndexAmRoutine
{
    NodeTag      type;

    /*
     * Total number of strategies (operators) by which we can traverse/search
     * this AM. Zero if AM does not have a fixed set of strategy assignments.
     */
    uint16      amstrategies;
    /* total number of support functions that this AM uses */
    uint16      amsupport;
    /* opclass options support function number or 0 */
    uint16      amoptsprocnum;
    /* does AM support ORDER BY indexed column's value? */
    bool        amcanorder;
    /* does AM support ORDER BY result of an operator on indexed column? */
    bool        amcanorderbyop;
    /* does AM support backward scanning? */
```

```

bool        amcanbackward;
/* does AM support UNIQUE indexes? */
bool        amcanunique;
/* does AM support multi-column indexes? */
bool        amcanmulticol;
/* does AM require scans to have a constraint on the first index column? */
bool        amoptionalkey;
/* does AM handle ScalarArrayOpExpr quals? */
bool        amsearcharray;
/* does AM handle IS NULL/IS NOT NULL quals? */
bool        amsearchnulls;
/* can index storage data type differ from column data type? */
bool        amstorage;
/* can an index of this type be clustered on? */
bool        amclusterable;
/* does AM handle predicate locks? */
bool        ampredlocks;
/* does AM support parallel scan? */
bool        amcanparallel;
/* does AM support columns included with clause INCLUDE? */
bool        amcaninclude;
/* does AM use maintenance_work_mem? */
bool        amusemaintenanceworkmem;
/* OR of parallel vacuum flags */
uint8       amparallelvacuumoptions;
/* type of data stored in index, or InvalidOid if variable */
Oid         amkeytype;

/* interface functions */
ambuild_function ambuild;
ambuildempty_function ambuildempty;
aminsert_function aminsert;
ambulkdelete_function ambulkdelete;
amvacuumcleanup_function amvacuumcleanup;
amcanreturn_function amcanreturn; /* can be NULL */
amcostestimate_function amcostestimate;
amoptions_function amoptions;
amproperty_function amproperty; /* can be NULL */
ambuildphasename_function ambuildphasename; /* can be NULL */
amvalidate_function amvalidate;
ambeginscan_function ambeginscan;
amrescan_function amrescan;
amgettuple_function amgettuple; /* can be NULL */
amgetbitmap_function amgetbitmap; /* can be NULL */
amendscan_function amendscan;
ammarkpos_function ammarkpos; /* can be NULL */
amrestrpos_function amrestrpos; /* can be NULL */

/* interface functions to support parallel index scans */
amestimateparallelscan_function amestimateparallelscan; /* can be NULL */
aminitparallelscan_function aminitparallelscan; /* can be NULL */
amparallelrescan_function amparallelrescan; /* can be NULL */
} IndexAmRoutine;

```

要想真正有用，一个索引访问方法还必须有一个或多个定义在 `pg_opfamily`、`pg_opclass`、`pg_amop` 和 `pg_amproc` 中的操作符族和操作符类。这些项允许规划器判断哪种查询条件适用于这个索引访问方法的索引。操作符族和类在 Section 36.16 中描述，它是阅读本章所需的前导材料。

一个独立的索引是由一个`pg_class`项定义的，该项描述索引为一个物理关系。还要加上一个`pg_index`项来显示索引的逻辑内容——也就是说，它所拥有的索引列集以及这些列的语义是被相关操作符类刻画的。索引列（键值）可以是底层表的简单列，也可以是该表行上的表达式。索引访问方法通常不关心索引的键值来自那里（它总是操作预计算过的键值），但是它会对`pg_index`中的操作符类信息很感兴趣。所有这些目录项都可以被当作关系数据结构的一部分访问，这个数据结构会被传递给索引上的所有操作。

`IndexAmRoutine`中的有些标志域的含义并不那么直观。`amcanunique`的要求在Section 58.5中讨论。`amcanmulticol`标志断言该索引访问方法支持多键列索引，`amoptionalkey`断言它允许对那种在第一个索引列上没有给出可索引限制子句的扫描。如果`amcanmulticol`为假，那么`amoptionalkey`实际上说的是该访问方法是否允许不带限制子句的全索引扫描。那些支持多索引列的访问方法必须支持那些在省略了除第一个列之外的任何或所有其它列上约束的扫描；不过，它们被允许去要求在第一个列上出现一些限制，并且这一点是以把`amoptionalkey`设置为假作为标志的。一个索引AM可能将`amoptionalkey`设置为假的一种原因是，如果它不索引空值。因为大多数可索引的操作符都是严格的并且因此不能对空输入返回真，所以不为空值存储索引项乍看上去很吸引人：因为它们不可能被一个索引扫描返回。不过，当一个索引扫描对于一个给定索引列上没有约束子句时，这种讨论就不成立了。实际上，这意味着设置了`amoptionalkey`为真的索引必须索引空值，因为规划器可能会决定在根本没有扫描键的时候使用这样的索引。一个相关的限制是一个支持多索引列的索引访问方法必须支持索引第一列之后的列中的空值，因为规划器会认为这个索引可以用于在那些列上没有限制的查询。例如，考虑一个在(a, b)上的索引和一个有WHERE a = 4的查询。系统会认为该索引可以用于扫描 a = 4的行，如果索引忽略了 b 为空的行，那么就是错误的。不过，忽略那些在第一个索引列上值为空的行是OK的。一个索引空的索引访问方法可能也会设置`amsearchnulls`，表明它支持将IS NULL和IS NOT NULL子句作为搜索条件。

`amcaninclude` 一个标志指示此访问方法是否支持“included”列，这些列可以包含除键列之外的其他列（不处理它们）。上一段中的要求仅适用于关键列。除其他外，`amcanmulticol=false` 和 `amcaninclude=true`组合是实用的。这表明可以有包含列，而只有一个键列。此外，`amoptionalkey`独立地，包含列必须可以为空。

## 58.2. 索引访问方法函数

索引访问方法必须在`IndexAmRoutine`中提供的索引构造和维护函数有：

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

创建一个新索引。索引关系已经被物理创建，但是是空的。必须用索引访问方法要求的固定数据填充它，外加所有已经在表里的行的项。通常，`ambuild`函数会调用`table_index_build_scan()`来扫描表以获取现有元组并计算需要被插入到索引的键。该函数必须返回一个已分配内存的结构，其中包含关于新索引的统计信息。

```
void
ambuildempty (Relation indexRelation);
```

构建一个空索引，并且把它写入到给定关系的初始化分叉中（`INIT_FORKNUM`）。只会为不做日志的索引调用这个方法，被写入到初始化分叉的空索引在每次服务器启动时将被复制到主关系分叉中。

```
bool
aminsert (Relation indexRelation,
          Datum *values,
```

```

bool *isnull,
ItemPointer heap_tid,
Relation heapRelation,
IndexUniqueCheck checkUnique,
IndexInfo *indexInfo);

```

向现有索引插入一个新元组。values和isnull数组给出需要被索引的键值，而heap\_tid是要被索引的 TID。如果该访问方法支持唯一索引（它的amcanunique标志为真），那么checkUnique指示要执行的唯一性检查类型。这根据唯一约束是否为可推迟的而变化，详见Section 58.5 通常在执行唯一性检查时访问方法仅需要heapRelation参数（因为那时它将不得不到堆中验证元组的存活性）。

该函数的布尔结果值仅仅在checkUnique为UNIQUE\_CHECK\_PARTIAL时才有意义。这种情况下一个“真”结果意味着这个新项是已知唯一的，反之“假”结果意味着它可能不是唯一的（并且一个延迟的唯一性校验必须是预定的）。对于其他情况，建议使用一个常量“假”结果。

有些索引可能不会索引所有元组。如果元组不被索引，aminert应该仅返回而什么都不做。

如果索引AM希望在SQL语句中连续的索引插入之间缓冲数据，它可以在indexInfo->ii\_Context中分配空间并且在indexInfo->ii\_AmCache（初始为NULL）中存放一个指向该数据的指针。

```

IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);

```

从索引中删除元组。这是一个“批量删除”操作，它的意图是通过扫描整个索引并检查每个项看它是否需要被删除。被传递进来的callback函数必须被调用（调用风格是：callback(TID, callback\_state) returns bool）来判断任何其引用的 TID 标识的索引项是否需要删除。必须返回 NULL 或者是一个 palloc 过的、包含删除操作效果的统计信息的结构。如果不需要向amvacuumcleanup传递信息，返回 NULL 也是 OK 的。

由于maintenance\_work\_mem被限制，在删除多行的时候ambulkdelete可能需要被调用多次。stats参数是对这个索引上一次调用的结果（在一个VACUUM操作中第一次调用时是NULL）。这将允许 AM 在整个操作过程中积累统计信息。典型的，如果被传递的stats非空，ambulkdelete将会修改并返回相同的结构。

```

IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);

```

在一个VACUUM操作（零个或更多次ambulkdelete调用）后清空。虽然不必做任何返回索引统计信息之外的事情，但是它可能执行批量清理，例如回收空索引页面。stats是最后一次ambulkdelete 调用返回的东西或者 NULL（如果没有元组需要删除而未调用ambulkdelete）。如果结果不是 NULL，那么它必须是一个已经被 palloc 的结构。它包含的统计信息将用于更新pg\_class并且由VACUUM报告（如果给出了VERBOSE）。如果索引在VACUUM操作期间根本没有改变，那么返回 NULL 也是可以的，否则必须返回正确的统计信息。

amvacuumcleanup将也会在一个ANALYZE操作结束时被调用。这种情况中stats总是 NULL 并且任何返回值都将会被忽略。这种情况可以通过检测info->analyze\_only来区分。我们建议，在这样的调用中访问方法除了做插入后的清理之外什么也不做，并且那是仅仅是在一个自动清理工作者进程中。

```
bool
amcanreturn (Relation indexRelation, int attno);
```

通过返回型为一个IndexTuple的索引项的被索引列值，检查索引是否能在给定列上支持只用索引的扫描。属性编号从 1 开始编号，即第一列的 attno 是 1。如果支持返回 TRUE，否则返回 FALSE。对于包含列（如果支持），此函数将始终返回 true，因为包含列不能被检索是没有意义的。如果访问方法 完全不支持只用索引的扫描，其IndexAmRoutine结构中的amcanreturn域可以被设置为 NULL。

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

估计一次索引扫描的开销。这个函数在下面的Section 58.6中有完整的讨论。

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

分析和验证一个索引的 reloptions 数组。仅当一个索引存在非空 reloptions 数组时才会被调用。reloptions是一个text数组，包含name=value形式的项。该函数应当构建一个bytea值，该值将被拷贝进索引的 relcache 项的rd\_options域。bytea值的数据内容是开放由访问方法定义的，大部分的标准访问方法都使用StdRdOptions结构。当validate为真时，如果任何一个选项都不可识别或者含有非法值，该函数都应当报告一个适当的错误消息；当validate为假时，非法项应该被安静地忽略（当正在载入的选项已经在pg\_catalog中时，validate为假；仅在访问方法已经改变了选项的规则时才可能找到非法项，并且在此情况下忽略废弃的项是合适的）。如果想要默认行为，那么返回 NULL 也OK。

```
bool
amproperty (Oid index_oid, int attno,
            IndexAMProperty prop, const char *propname,
            bool *res, bool *isnull);
```

amproperty方法允许索引方法覆盖pg\_index\_column\_has\_property和相关函数的默认行为。如果访问方法对于索引性质查询没有指定特殊的行为，其IndexAmRoutine结构的amproperty域可以被设置为 NULL。否则，对于pg\_indexam\_has\_property调用会使用均为 0 的index\_oid和attno参数来调用amproperty方法；对于pg\_index\_has\_property调用会使用有效的index\_oid和为 0 的attno参数来调用amproperty方法；对于pg\_index\_column\_has\_property调用会使用有效的index\_oid以及大于零的attno参数来调用amproperty方法。prop是用于标识被测试性质的枚举值，而propname是原始的性质名称字符串。如果核心代码不能识别该性质名称，则prop为AMPROP\_UNKNOWN。访问方法可以通过检查propname是否匹配（为与核心代码一致，使用pg\_strcasecmp来匹配）来定义自定义性质名称；对于核心代码已知的名称，最好检查prop。如果amproperty方法返回true则表示它已经确定了性质测试的结果：它必定会设置\*res为要返回的布尔值，如果要返回 NULL 则设置\*isnull为true（两个被引用的变量在调用之前要被初始化为false）。如果amproperty方法返回false则核心代码将会用其通常的逻辑来确定性质测试的结果。

支持排序操作符的访问方法应该实现AMPROP\_DISTANCE\_ORDERABLE性质测试，因为核心代码不知道如何做该测试并且会返回 NULL。如果有比打开索引并调用amcanreturn（这是核心代



码的默认行为) 更廉价的方法来做AMPROP\_RETURNABLE测试, 最好也实现它。默认行为应该对所有其他标准性质是符合要求的。

```
char *
ambuildphasename (int64 phasenum);
```

返回给定构建相数号码的文本名称。相数号码是在通过pgstat\_progress\_update\_param 接口构建索引期间报告的。然后相数号码在pg\_stat\_progress\_create\_index视图中公开。

```
bool
amvalidate (Oid opclassoid);
```

只要访问方法能够, 为指定的操作符类验证系统目录项。例如, 这可能包括所有所需支持函数所提供的测试。如果该 opclass 不合法, amvalidate函数必须返回假。所存在的问题应由ereport消息报告, 通常情况下在 INFO 级。

当然, 索引的目的是支持扫描那些匹配一个可索引WHERE情况的元组, 常常也被称为限定词或扫描键。索引扫描的语义在下面的Section 58.8中有完整的描述。一个索引访问方法可以支持“普通”索引扫描、“位图”索引扫描或者两者。一个索引访问方法必须或可能提供的与扫描相关的函数是:

```
IndexScanDesc
ambeginscan (Relation indexRelation,
            int nkeys,
            int norderbys);
```

为一个索引扫描做准备。nkeys和norderbys参数说明要被用在扫描中的条件和排序操作符的数目, 它们可以用于空间分配目的。注意扫描键的实际值还没有被提供。结果必须是一个 palloc 过的结构。由于实现的原因, 索引访问方法必须通过调用RelationGetIndexScan() 来创建这个结构。在大多数情况中, ambeginscan除了做这个调用和获取锁之外不会做很多工作, 索引扫描启动中有趣的部分在amrescan中。

```
void
amrescan (IndexScanDesc scan,
         ScanKey keys,
         int nkeys,
         ScanKey orderbys,
         int norderbys);
```

开始或者重新开始一个索引扫描, 可能使用的是一个新的扫描键(要想使用之前传递的键重新开始, 给keys 和/或orderbys传递 NULL)。请注意, 使用的键或排序操作符的个数不能大于传递给ambeginscan的个数。实际上这个重新开始特性的使用场景是: 在一个嵌套循环连接选取了一个新的 outer 元组时, 因此需要一个新的键比较值, 但扫描键结构仍然保持相同。

```
bool
amgettuple (IndexScanDesc scan,
           ScanDirection direction);
```

在给定扫描中取下一个元组, 向给定方向移动(在索引中向前或者向后)。如果取到了元组, 则返回 TRUE, 如果取到匹配的元组, 返回 FALSE。在 TRUE 的情况中, 该元组的TID 被存储在scan结构中。请注意“成功”只意味着索引包含一个匹配扫描键的项, 并不意味着该元组仍然在堆中存在, 或者是能够通过调用者的快照测试。在成功时, amgettuple也必须把scan->xs\_recheck设置成 TRUE 或者 FALSE。FALSE 意味着它确定索引项匹配扫描键。TRUE 意味着它并不确定, 而且必须在取得堆元组之后对它重新检查扫描键表示的条

件。 这条规定支持“有损的”索引操作符。注意重新检查仅仅对扫描条件扩展；一个部分索引谓词（如果有）从不被amgettupple调用者重新检查。

如果索引支持只用索引扫描（即amcanreturn对其任何一列返回 TRUE），则在成功时AM 也必须检查scan->xs\_want\_itup，并且如果检查为真它必须返回索引项的原始被索引数据。amcanreturn返回false的列可以作为nulls返回。该数据的返回形式可以是一个存储在scan->xs\_itup中的IndexTuple指针外加元组描述符scan->xs\_itupdesc，或者是一个存储在scan->xs\_hitup中的HeapTuple指针外加元组描述符scan->xs\_hitupdesc（在重构可能无法放在一个IndexTuple中的数据时，应该使用后一种格式）。不管是哪种形式，访问方法应该负责管理好指针引用的数据。至少在为扫描下一次调用amgettupple、amrescan或amendscan之前，该数据必须是完好的。

如果访问方法支持“普通”索引扫描，只需要提供amgettupple函数。如果不支持，它的IndexAmRoutine结构的amgettupple域必须被设置为 NULL。

```
int64
amgetbitmap (IndexScanDesc scan,
            TIDBitmap *tbm);
```

在给定扫描中取所有元组并且把它们添加到调用者提供的TIDBitmap中（即，把元组 ID 的集合 OR 到已经存在于位图中的东西里面）。返回被取得的元组的数量（这可能仅仅是一个近似计数，例如一些 AM 不会去重）。在把元组 ID 插入到位图时，amgetbitmap可以指明对指定元组 ID 要求重新检查扫描条件。这与amgettupple的 xs\_recheck输出参数类似。注意：在当前的实现中，这个特性的支持是和对位图本身有损存储的支持合并在一起的，并且调用者会对可重新检查的元组检查扫描条件和部分索引谓词（如果有）。但是，那不会总是真的。amgetbitmap和amgettupple不能被用于同一个索引扫描；正如Section 58.8所解释的，在使用amgetbitmap时也有其他的限制条件。

如果访问方法支持“bitmap”索引扫描，则仅需要提供amgetbitmap函数。如果不支持，它的IndexAmRoutine结构中的amgetbitmap域必须被设置为 NULL。

```
void
amendscan (IndexScanDesc scan);
```

结束扫描并释放资源。不应该释放scan结构本身，但访问方法内部使用的任何锁或者 pin 都应该被释放，以及ambeginscan和其他扫描相关函数分配的任何其他内存。

```
void
ammarkpos (IndexScanDesc scan);
```

标记当前扫描位置。访问方法只需要支持每个扫描里面有一个被标记的扫描位置。

ammarkpos函数只有在访问方法支持有序扫描时才需要提供。如果不支持，则访问方法的IndexAmRoutine结构的ammarkpos域可以设置为 NULL。

```
void
amrestrpos (IndexScanDesc scan);
```

把扫描恢复到最近标记的位置。

amrestrpos函数只有在访问方法支持有序扫描时才需要提供。如果不支持，则访问方法的IndexAmRoutine结构的amrestrpos域可以设置为 NULL。

除了支持普通的索引扫描之外，某些类型的索引可能希望支持并行索引扫描，这种方式允许多个后端合作来执行一次索引扫描。索引访问方法应该安排好各种事情，这样每个参与合作

的进程才能返回原本会由普通非并行索引扫描执行得到的元组的一个子集，但是得到的那些子集的并集应该等于普通非并行索引扫描得到的元组集合。此外，虽然不需要并行扫描返回的元组有任何全局顺序，但每个参与合作的后端中返回的元组子集的顺序必须匹配所要求的顺序。必须实现下列函数才能支持并行索引扫描：

```
Size
amestimateparallelsan (void);
```

估算并且返回访问方法执行一次并行扫描所需要的动态共享内存的字节数（这个数字是对ParallelIndexScanDescData中访问方法无关的数据所需空间量的补充而不是替代）。

对于不支持并行扫描或者额外存储需求的的字节数为零的访问方法，无需实现这个函数。

```
void
aminitparallelsan (void *target);
```

在一次并行扫描的开头将调用这个函数来初始化动态共享内存。target将指向一段动态共享内存空间，其大小至少为之前amestimateparallelsan返回的字节数，并且这个函数可以使用这部分空间来存放它希望存放的任何数据。

对于不支持并行扫描或者不要求初始化共享内存空间的情况，无需实现这个函数。

```
void
amparallelrescan (IndexScanDesc scan);
```

如果实现了这个函数，当并行索引扫描必须被重启时，将会调用这个函数。它应该重置由aminitparallelsan建立的任何共享状态，这样扫描将会被重头重新开始。

## 58.3. 索引扫描

在一个索引扫描中，索引访问方法负责提供它拿到的匹配扫描键的所有元组的TID。访问方法不会涉及从索引的父表中实际取得那些元组，也不会涉及判断它们是否通过了扫描的可见性测试或者是其它条件。

一个扫描键是一个WHERE子句的内部表示，WHERE子句的形式是index\_key operator constant，其中索引键字索引中的一个列，而操作符是和该索引列相关联的操作符族的一个成员。一个索引扫描拥有零个或者多个扫描键，它们是隐式 AND 关系 — 返回的元组被认为满足所有列出的条件。

对于一个特定查询，访问方法可能报告索引是有损的或者要求重新检查。这就暗示着该索引扫描会返回所有通过扫描键的项，外加上一些可能没通过扫描键的项。核心系统的索引扫描机制然后就会再次在堆元组上应用索引条件来验证它是否真地应该被选择。如果没有指定重新检查选项，索引扫描必须返回准确的匹配项集合。

请注意，确保找到所有（只有）通过所有给定扫描键的条目的工作完全由访问方法负责。还有，核心系统将只是简单地 放过所有匹配扫描键和操作符族的WHERE子句，而不会做任何语义分析来判断它们是否冗余或者矛盾。例如，给定WHERE x > 4 AND x > 14（其中x是一个 B-树 索引列，它被留给 B-树 amrescan函数来发现第一个扫描键是冗余并且可以被丢弃。amrescan期间需要的预处理的范围将取决于索引访问方法需要什么来把扫描键缩减为一种“正规化的”形式。

一些访问方法按照一个良定义的顺序来返回索引项，其他的则不会。实际上一个访问方法可以有两种不同的方式支持排序输出：

- 总是按数据的自然序返回项的访问方法应该设置amcanorder为真。当前，这样的访问方法必须对它们的等值和排序操作符使用b-tree兼容的策略号。

- 支持排序操作符的访问方法应该设置`amcanorderbyop`为真。这表示索引有能力按照满足`ORDER BY index_key operator constant`的一种顺序返回项。如前所述，这种形式的扫描修饰符可以被传递给`amrescan`。

`amgettuple`函数有一个`direction`参数，它可以是 `ForwardScanDirection`（正常情况）或者`BackwardScanDirection`。如果`amrescan`之后的第一次调用指定了`BackwardScanDirection`，那么匹配条件的索引项集合是从后向前扫描的，而不是通常的从前向后扫描，因此`amgettuple`必须返回索引中最后一个匹配元组，而不是通常情况下的第一个（这只对设置了`amcanorder`为真访问方法发生）。在第一次调用后，`amgettuple`必须被准备好从最近被返回项的位置按照任何一种方向推进扫描（但是如果`amcanbackward`为假，所有后续调用将使用第一次相同的方向）。

支持排序扫描的访问方法必须支持在扫描里“标记”一个位置并且随后返回到这个标记过的位置。同一个位置可能会被重复多次还原。但是，每个扫描中只有一个位置需要被记住；一个新的`ammarkpos`调用将重写之前标记的位置。一个不支持排序扫描的访问方法无需在`IndexAmRoutine`中提供`ammarkpos`和`amrestrpos`函数，把这些指针设置为 `NULL` 即可。

扫描位置和标记位置（如果存在）都必须在面对索引中的并发插入和删除时保持一致性。如果一个新插入的项并未被一个扫描返回（如果该扫描开始的时候该项已经存在，该扫描已经找到该项），或者说扫描通过重新扫描或者反向扫描返回这样一个项（即使它第一次没有返回这样一个项），这些情况都是可以接受的。类似的还有，一个并发的删除可能或不可能被反映在一个扫描的结果中。重要的是，插入或者删除不会导致扫描错过或者多次返回本身不是被插入或者删除的项。

如果索引存储原始被索引的数据值（并且不是它们的某种有损表示），它可用来支持只用索引的扫描，着这种扫描中索引返回的就是实际的数据而不只是堆元组的 `TID`。这只有在可见性映射显示该 `TID` 位于一个全部可见的页面时才能避免 `I/O`；否则必须访问堆元组来检查 `MVCC` 可见性。但是这就不用访问方法操心了。

除了使用`amgettuple`，一个索引扫描可以通过`amgetbitmap`在一次调用中取得所有元组来完成。这样做可能会比`amgettuple`有显著的效率提升，因为它可以避免在访问方法内的加锁/解锁循环。原则上`amgetbitmap`应该和重复调用`amgettuple`的效果相同，不过我们强加了一些限制来简化这件事。首先，`amgetbitmap`一次返回 所有元组并且标记并且不支持标记或恢复扫描位置。第二，在一个位图中返回的元组没有任何指定的顺序，这也是为什么`amgetbitmap`没有一个`direction`参数的原因（排序操作符也将永远不会提供给这种扫描）。还有，对于使用`amgetbitmap`的只用索引扫描没有规定，因为没有办法返回索引元组的内容。最后，如Section 58.4所说的，`amgetbitmap`不保证被返回元组上的任何锁。

注意如果访问方法的内部实现不适合一个 `API` 或其他 `API`，允许一个访问方法只实现`amgetbitmap`而不实现`amgettuple`，或者反过来。

## 58.4. 索引锁定考虑

索引访问方法必须支持多个进程对索引的并发更新。在索引扫描期间，核心LightDB系统在索引上获取 `AccessShareLock`，并且在更新索引时（包括普通`VACUUM`）获取`RowExclusiveLock`。因为这些锁类型不会冲突，所以访问方法负责处理它可能需要的任何细粒度锁。把索引作为一个整体的`ACCESS EXCLUSIVE` 锁只会在索引创建、删除或`REINDEX`时被使用（`SHARE UPDATE EXCLUSIVE` 取而代之的是用 `CONCURRENTLY`）。

创建一个支持并发更新的索引类型通常要求对所需的行为进行广泛并且细致的分析。对于**tree** 和哈希索引类型，你可以阅读`src/backend/access/nbtree/README`和`src/backend/access/hash/README`中的设计决策。

除了索引自己内部的一致性要求之外，并发更新带来了一些父表（堆）和索引之间的一致性问题。因为 `LightDB`是把堆的访问和更新与索引的访问和更新分开的，所以存在一些窗口期，在其间索引可能会与堆不一致。我们用下面的规则处理这样的问题：

- 一个新堆项在其索引项之前被制作（因此并发的索引扫描很可能看不到堆项。这么做应该是可以的，因为索引的读取者对未提交的行不感兴趣。见Section 58.5）。

- 当一个堆项要被删除（通过VACUUM）时，它的所有索引项都必须首先删除。
- 一次索引扫描必须在保存有amgettupple最后返回项的索引页面上维护一个 pin，并且ambulkdelete不能从页面中删除被其他后端加 pin 的项。下面会解释需要这条规则的原因。

没有第三条规则，那么一个索引读取者是在一条索引项被VACUUM删除之前看到它的，并且然后在VACUUM删除它之后找到其对应的堆项。如果读取者到达该项时，该项编号仍然没有被使用，那么这种情况不会导致严重的问题，因为空的项槽位会被heap\_fetch()忽略。但是如果第三个后端已经为其它什么东西重用了这个项槽位又会怎样？在使用MVCC兼容的快照时，那么就不会有问题，因为槽位的新占据者太新了以至于无法通过快照测试。但是，对于非MVCC兼容的快照（例如SnapshotAny），那么就有可能接受并返回一个实际上并不匹配扫描键的行。可以通过要求扫描键在所有情况下都在堆行上重新检查来避免这种情况，但是这种方法开销太大了。取而代之的是，通过在索引页面上使用一个pin作为一个代理来表示，读取者可能还处于从索引项到匹配的堆项的“过程中”。用ambulkdelete来操作这样一个pin上的块确保VACUUM无法在读取者完成之前删除堆项。这种解决方案在运行时只有一点开销，而只是在真有一个冲突的非常罕见情况下才导致阻塞开销。

这个解决方法要求索引扫描是“同步的”：我们不得不在扫描完对应的索引项之后马上去取每个堆元组。这样的方案开销比较大，原因有多个。而一个“异步的”扫描可以先从索引里收集很多TID，并且在稍后的某个时间只访问堆元组，这样要求更少的索引锁定负荷并且能够允许一种更高效的堆访问模式。但是按照上面的分析，在非MVCC兼容的快照上我们必须使用同步方法，而异步扫描则适合于使用MVCC快照的查询。

在一个amgetbitmap索引扫描中，访问方法不会在任何被返回的元组上保持一个索引pin。因此只有把这种扫描与MVCC兼容的快照一起使用才是安全的。

当ampredlocks标志没有被设置时，在一个可序列化事务中使用该索引访问方法的任何扫描将在整个索引上获取一个非阻塞的谓词锁。这将和一个并发可序列化事务中项索引中插入任何元组发生读-写冲突。如果在一组并发可序列化事务之间检测到特定模式的读-写冲突，其中一个事务可能会被取消来保护数据完整性。当该标志被设置，它表示该索引访问方法实现了细粒度的谓词锁，这将有望缩减这种事务取消的频率。

## 58.5. 索引唯一性检查

LightDB使用唯一索引来强制SQL唯一性约束，唯一索引实际上是不允许多个项有相同键的索引。一个支持这个特性的访问方法要设置amcanunique为真（目前，只有b-tree支持它）。在强制唯一性时不会考虑INCLUDE子句中列出的列。

因为MVCC，必须允许重复的项在物理上存在于索引之中：这些项可能指向某个单一逻辑行的后继版本。实际想强制的行为是，任何MVCC快照都不能包含两个具有相同索引键的行。在向一个唯一索引中插入一个新行时需要被检查的情况可分解成：

- 如果一个有冲突的合法行已被当前事务删除，这是可以的（特别是因为一个UPDATE总是在插入新版本之前删除旧版本，这样就允许一个行上的UPDATE不改变键）。
- 如果一个有冲突的行已经被还未提交的事务插入，那么准备插入的事务必须等待看看前面那个事务是否提交。如果它回滚就不会有冲突。如果它提交并且没有删除存在冲突的行，则有一个唯一性违背（实际上我们只是等待那个其他事务结束，然后在全部事务里重做可见性检查）。
- 类似的，如果一个有冲突的有效行被一个准备提交的事务删除，那么另外一个准备插入的事务必须等待该事务提交或者退出，然后重做测试。

此外，在根据上述规则报告唯一性违背之前，访问方法必须重新检查刚被插入的行的存活性。如果已经因为事务的提交而死亡，那么不应当报告任何违背（这种情况不可能出现在插入在同一事务中创建的行的普通场景中。但是在CREATE UNIQUE INDEX CONCURRENTLY的过程中是可能发生的）。

要求索引访问方法自己应用这些测试，这就意味着它必须到达堆来查看那些根据索引内容有重复键的任意行的提交状态。这无疑是丑陋并且非模块化的，但是这样可以节约重复的工作：如果我们进行一次独立的探测，那么查找一个冲突行的索引查找本质上将在查找插入新索引项位置时被重复。此外，没有很明显的方法来避免竞争情况，除非冲突检查是插入新索引项动作的一部分。

如果唯一约束是可延迟的，就存在额外的复杂性：我们需要能够为一个新行插入一个索引项，但是推迟任何唯一性违背错误直到语句结束或者更晚。为了避免对索引不必要的重复搜索，索引访问方法应该在初始插入过程中做一次初步的唯一性检查。如果显示绝对不会有冲突的活元组，就可以完成。否则，我们计划一次重新检查，它将在强制约束的时候发生。在重新检查时，如果具有相同键的被插入元组和某个其他元组都活着，则必须报告错误（注意为了这个目的，“活着”实际意味着“在索引项的 HOT 链上的任何元组都活着”）。要实现这一点，需要给 `aminsert` 传递一个 `checkUnique` 参数，其中包含下列值之一：

- `UNIQUE_CHECK_NO`表明不需要做唯一性检测（这不是一个唯一索引）。
- 如上所述，`UNIQUE_CHECK_YES`表明有一个不可延迟的唯一索引，并且必须立即做唯一性检测。
- `UNIQUE_CHECK_PARTIAL`表明唯一性约束是可延迟的。LightDB将会用这个模式来插入每一行的索引项。访问方法必须允许重复的项进入索引，并且通过从 `aminsert` 返回 `FALSE` 来报告任何可能的重复。对于返回 `FALSE` 的每一行，将计划一个延迟的重新检查。

访问方法必须能够标识任何可能违反唯一约束的行，但是对它来说假阳性报告不是错误。这样就允许检查不用等到其他事务都结束；这里报告的冲突不会被当做错误来看待，并且随后将会被重新检查，而到那时它们可能不再是冲突了。

- `UNIQUE_CHECK_EXISTING`表明这是一个行的延迟重新检查，该行被报告为一个潜在的唯一性违背。尽管这会通过调用 `aminsert` 来实现，在这种情况下这个访问方法不能插入一个新索引项。该索引项已经存在。当然，访问方法必须检查是否有另一个活着的索引项。如果有，并且如果目标行也仍然存活，那么报告错误。

我们推荐，在一个 `UNIQUE_CHECK_EXISTING` 调用中，访问方法进一步验证目标行真的在索引中有一个现有的项，并且如果不是这样就报错。这是个好主意，因为被传到 `aminsert` 的索引元组值将已经被重新计算过。如果索引定义涉及不是真正不变的函数，我们可能正在检查索引的错误区域。对重新检查中找到的目标行的检查会验证我们正在扫描之前被用于原始插入的同一元组值。

## 58.6. 索引开销估计函数

`amcostestimate` 函数被给定描述一个可能的索引扫描的信息，包括决定在索引中使用的 `WHERE` 和 `ORDER BY` 子句的列表。它必须返回访问该索引的开销估计以及 `WHERE` 子句的选择度（也就是说，在索引扫描期间将检索的行在父表中所占据的比例）。对于简单情况，几乎开销估计器的所有工作都可以通过调用优化器中的标准过程完成；有 `amcostestimate` 函数的目的是允许索引访问方法提供和索引类型相关的知识，这种情况下可以改进标准的估计。

每个 `amcostestimate` 函数的签名必须是：

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

前三个参数是输入参数：

root

规划器的有关正在被处理的查询的信息。

path

被考虑的索引访问路径。其中除了开销和选择度值之外的域都有效。

loop\_count

应该被开销估计所考虑的索引扫描重复次数。当考虑用在一个嵌套循环连接中的参数化扫描时，这个参数通常会大于 1。注意代价估计应该仍然是对于一次扫描的，一个更大的loop\_count意味着可能在多次扫描间允许一些缓冲效果比较合适。

后四个参数是传引用的输出参数：

\*indexStartupCost

设置为索引启动处理的开销。

\*indexTotalCost

设置为索引处理的总开销。

\*indexSelectivity

设置为索引的选择度。

\*indexCorrelation

设置为索引扫描顺序和下层的表的顺序之间的相关性。

\*indexPages

设置为索引叶子页的数量

请注意开销估计函数必须用 C 编写，而不能用 SQL 或者任何可用的过程语言，因为它们必须访问规划器/优化器的内部数据结构。

索引访问开销应该采用被src/backend/optimizer/path/costsize.c使用的参数进行计算：一次顺序磁盘块获取的开销是seq\_page\_cost、一次非顺序获取的开销是random\_page\_cost并且处理一个索引行的开销通常是cpu\_index\_tuple\_cost。另外，在索引处理期间（尤其是索引条件本身的计算）调用的任何比较操作符都会耗费cpu\_operator\_cost倍数的开销。

访问开销应该包括所有与扫描索引本身相关的磁盘和 CPU 开销，但是不包括检索或者处理被索引标识出来的父表行的开销。

“启动开销”是整个扫描开销中的一部分：在能够开始取第一行之前必须花掉的开销。对于大多数索引这个开销是零，但是那些启动开销很大的索引类型不会把它设置为零。

indexSelectivity应该设置成在索引扫描期间，父表行被检索的估计比例。在一个有损查询的情况下，这个值通常高于实际通过给定查询条件的行的比例。

indexCorrelation应该被设置成索引顺序和表顺序之间的相关性（范围从 -1.0 到 1.0）。这个数值被用于调整从父表中取出行的开销估计。

indexPages应该被设置为叶子页面的数量。它会被用来估算并行索引扫描用到的工作者的数目。

当loop\_count大于一时，返回的数应该是该索引任何一次扫描的平均期望值。

## 开销估计

一个典型的开销估计器会像下面这样进行处理：

1. 基于给出的条件情况，估计并返回父表行将被访问的比例。如果缺乏索引类型相关的知识，那么使用标准的优化器函数 `clauselist_selectivity()`：

```
*indexSelectivity = clauselist_selectivity(root, path->indexquals,
                                           path->indexinfo->rel->relid,
                                           JOIN_INNER, NULL);
```

2. 估计在扫描过程中将被访问的索引行数。对于许多索引类型，这个等于 `indexSelectivity` 乘以索引中的行数，但是可能更多（请注意，页面和行中的索引尺寸从 `path->indexinfo` 结构中获得）。
3. 估计在扫描中将检索的索引页面数量。这个可能就是 `indexSelectivity` 乘以索引的总页面数。
4. 计算索引访问开销。一个通用的估计器可能会：

```
/*
 * Our generic assumption is that the index pages will be read
 * sequentially, so they cost seq_page_cost each, not random_page_cost.
 * Also, we charge for evaluation of the indexquals at each index row.
 * All the costs are assumed to be paid incrementally during the scan.
 */
cost_qual_eval(&index_qual_cost, path->indexquals, root);
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = seq_page_cost * numIndexPages +
                 (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

不过，上面没有考虑重复索引扫描间的索引读分期补偿（amortization）。

5. 估计索引的相关性。对于一个简单的在单列有序索引，这个值可以从 `pg_statistic` 中检索。如果相关性是未知，那么保守的估计是零（没有相关性）。

开销估计器函数的例子可以在 `src/backend/utils/adt/selfuncs.c` 中找到。



---

## Chapter 59. 通用WAL 记录

虽然所有内建的被 WAL 记录的模块都有它们自己的 WAL 记录类型，系统中也还是有一种通用 WAL 记录类型，它以一种通用的方式描述了对页面的改变。这对于提供自定义访问方法的扩展有用，因为这类扩展无法注册自己的 WAL 重做例程。

构建通用 WAL 记录的 API 定义在access/generic\_xlog.h中，实现在access/transam/generic\_xlog.c中。

要使用通用 WAL 记录工具执行一次被 WAL 记录的数据更新，要遵循这些步骤：

1. `state = GenericXLogStart(relation)` — 为给定的关系构建一个通用 WAL 记录。
2. `page = GenericXLogRegisterBuffer(state, buffer, flags)` — 注册一个要在当前的通用 WAL 记录中修改的缓冲区。这个函数会返回一个指针指向该缓冲区页面的一份临时拷贝，修改将会在该拷贝上进行（不要直接修改该缓冲区的内容）。第三个参数是适用于该操作的标志的位掩码。当前这类标志只有`GENERIC_XLOG_FULL_IMAGE`，它表示在 WAL 记录中应该包括一个完整页面镜像而不是增量更新。如果是新页面或者页面已经被完全重写，通常会设置这个标志。如果被 WAL 记录的动作需要修改多个页面，可以反复调用`GenericXLogRegisterBuffer`。
3. 对包含在上一步中的页面镜像应用修改。
4. `GenericXLogFinish(state)` — 将更改应用到缓冲区并且发出通用 WAL 记录。

在上述步骤之间都可以调用`GenericXLogAbort(state)`取消 WAL 记录构造。这会丢弃所有对于页面镜像拷贝的更改。

在使用通用 WAL 记录功能时请注意以下几点：

- 不允许直接修改缓冲区！所有的修改必须在`GenericXLogRegisterBuffer()`取得的拷贝上完成。换句话说，制造通用 WAL 记录的代码不能为自己调用`BufferGetPage()`。不过，在合适的时间对缓冲区进行 `pin/unpin` 以及加锁/解锁仍然是调用者的责任。从`GenericXLogRegisterBuffer()`之前直到`GenericXLogFinish()`之后，每个目标上必须保持排他锁。
- 可以自由地混合注册缓冲区（步骤 2）和页面镜像修改（步骤 3），即两个步骤可以以任何顺序重复。记住注册缓冲区的顺序应该和重放时对它们加锁的顺序相同。
- 一个通用 WAL 记录能注册的缓冲区最大数量是`MAX_GENERIC_XLOG_PAGES`。如果超出这个限制将会抛出一个错误。
- 通用 WAL 假定要被修改的页面具有标准布局，特别是在`pd_lower`和`pd_upper`之间没有有用的数据。
- 由于正在修改缓冲区页面的拷贝，`GenericXLogStart()`不会开始临界区。因此可以在`GenericXLogStart()`之间`GenericXLogFinish()`安全地进行内存分配、抛出错误等。唯一真正的临界区存在于`GenericXLogFinish()`内。还有，不需要担心在错误退出期间对`GenericXLogAbort()`的调用。
- `GenericXLogFinish()`会负责标记缓冲区为脏并且设置它们的 LSN。你不需要显式地做这些工作。
- 对于不做日志的关系，所有的事情都一样，不过不会发出实际的 WAL 记录。因此，对于不做日志的关系你通常不需要做任何显式的检查。
- 通用 WAL 重做函数将按照注册缓冲区的顺序对它们获得排他锁。在重做所有更改后，这些锁将按照同样的顺序被释放。

- 如果对一个已注册的缓冲区没有指定GENERIC\_XLOG\_FULL\_IMAGE，通用 WAL 记录包含了新旧页面镜像之间的不同。这个不同是以逐字节比较的方式形成的。对于在页面内移动数据的情况来说这种方式不是很紧凑，未来可能会有改进。

---

# Chapter 60. B-树索引

## 60.1. 简介

LightDB包括了对标准btree（多路平衡树）索引数据结构的一个实现。任何能够被排序为良定义线性顺序的数据结构都可以用一个btree来索引。唯一的限制是一个索引项不能超过大约三分之一页面（如果适用，可以是TOAST压缩后的大小）。

因为每一种btree操作符类都会在其数据类型上施加一种排序顺序，btree的操作符类（或者实际上是操作符族）已经被用作LightDB对排序语义的一般表达和理解。因此，它们需要一些支持btree索引之外的特性，并且这个系统的一些部分与利用它们的btree访问方法有较大的不同。

## 60.2. B-树操作符类的行为

如Table 36.8所示，一个btree操作符类必须提供五种比较操作符： $<$ 、 $<=$ 、 $=$ 、 $>=$ 以及 $>$ 。有人可能会想 $<>$ 应该也是操作符类的一部分，但不是这样，因为几乎从不会在索引搜索中使用有 $<>$ 的WHERE子句（出于某种原因，规划器会认为 $<>$ 与一个btree操作符类相关，但它是通过 $=$ 操作符的逆操作符链接来找到这个操作符，而不是从`pg_amop`中查找）。

当一些数据类型共享近乎相同的排序语义时，它们的操作符类可以被组合成一个操作符族。这样做是有好处的，因为这样就允许规划器对跨类型比较进行推演。在操作符族中的每一种操作符类对其输入数据类型应该包含单一类型的操作符（及其相关的支持函数），而跨类型比较操作符及其支持函数则“松散”地放在操作符族中。推荐在操作符族中包括一套完整的跨类型操作符，这样能确保规划器可以表达它通过传递性推演出的任何比较条件。

这里有一些btree操作符类必须满足的基本假设：

- 一个 $=$ 操作符必须是一种等值关系。也就是说，对于该数据类型的所有非空值A、B、C：
  - $A = A$ 为真（自反律）
  - 如果 $A = B$ ，则有 $B = A$ （对称律）
  - 如果 $A = B$ 并且 $B = C$ ，则有 $A = C$ （传递律）
- 一个 $<$ 操作符必须是一种强排序关系。也就是说，对于所有的非空值A、B、C：
  - $A < A$ 为假（非自反律）
  - 如果 $A < B$ 以及 $B < C$ ，则有 $A < C$ （传递律）
- 此外，该排序是完全的。也就是说，对于所有非空值A、B：
  - $A < B$ 、 $A = B$ 和 $B < A$ 之中恰好有一个为真（三分律）  
（三分律无疑证明了比较支持函数定义的正确性。）

其他三种操作符可以以显而易见的方式用 $=$ 和 $<$ 来定义，并且必须和它们的行为保持一致。

对于一个支持多种数据类型的操作符族来说，当A、B、C取自该族中任意数据类型时，上述定律都必须保持。传递律是最难以保证的，因为在跨类型的情况中，传递律说明两种或者三种不同的操作符的行为是一致的。举个例子，把float8和numeric放在同一个操作符族中是行不通的，至少在当前的语义（为了和一个float8比较，numeric值会被转换成float8）下不行。因为float8有限的精度，这意味着不同的numeric值将被认为等于同一个float8值，因此传递律将被破坏。

对于多数据类型操作符族的另一个要求是，其中包括的定义在数据类型之间的任何隐式或者二进制强制造型不能改变相关的排序顺序。

为何一个btree索引要求这些定律在单一数据类型中必须保持的原因应该相对比较清楚：没有这些定律就不存在用于安排键的顺序。此外，使用不同数据类型键的索引搜索也要求比较操作在两种数据类型之间表现得稳定。btree索引机制本身并不严格要求在一个操作符族中扩展到三种或者更多种数据类型，但是规划器依赖于这种扩展来实现其优化的目的。

## 60.3. B-树支持函数

如Table 36. 中所示，btree定义了一种必需的和四种可选的支持函数。这五种用户定义的方法为：

order

对于btree操作符族为其提供了比较操作符的每一种数据类型组合，操作符族必须提供一个比较支持函数，在pg\_amproc中注册：支持函数编号为1，amproclefttype/amprocrighttype等于比较的左右数据类型（即匹配的操作符注册在pg\_amop中的数据类型）。比较函数必须接收两个非空值A和B并且返回一个int32值，返回值在 $A < B$ 、 $A = B$ 以及 $A > B$ 时分别为 $< 0$ 、 $0$ 和 $> 0$ 。不允许空值结果：该数据类型的所有值必须是可比较的。例子请见src/backend/access/nbtree/nbtcompare.c。

如果被比较的值是一种可排序的数据类型，合适的排序规则OID将使用标准的PG\_GET\_COLLATION()机制被传递给比较支持函数。

sortsupport

可选地，btree操作符族可以提供排序支持函数，它们以支持函数编号2注册。这些函数允许以一种比单纯调用比较支持函数更加高效的方式实现排序比较。涉及的API在src/include/utils/sortsupport.h中定义。

in\_range

可选地，btree操作符族可以提供in\_range支持函数，它们以支持函数编号3注册。在btree索引操作期间不会用到这些函数，它们扩展了操作符族的语义，这样就能支持包含RANGE offset PRECEDING以及RANGE offset FOLLOWING窗口帧界类型（见Section 4.2.8 的窗口子句。归根到底，这些函数所提供的额外信息是如何以一种与该操作符族的数据排序相兼容的方式加上或者减去一个offset值。

An in\_range 函数必须有签名

```
in_range(val type1, base type1, offset type2, sub bool, less bool)
returns bool
```

val和base必须是同一种类型，该类型也是操作符族所支持的类型之一（即它提供排序的一种类型）。不过，offset可以是一种不同的类型，该类型有可能不被该操作符族所支持。例如内建的时间\_ops族提供了一个in\_range函数，其offset是类型interval。一个操作符族可以为其所支持的任意类型提供in\_range函数以及一个或者更多种offset类型。每一个in\_range函数在进入pg\_amproc时，需要有amproclefttype等于type1以及amprocrighttype等于type2。

in\_range函数的本质语义取决于两个布尔标志参数。它应该将base和offset相加或者相减，然后用val与其结果比较：

- 如果!sub并且!less，则返回 $val \geq (base + offset)$
- 如果!sub并且less，则返回 $val \leq (base + offset)$
- 如果sub并且!less，则返回 $val \geq (base - offset)$
- 如果sub并且less，则返回 $val \leq (base - offset)$

在这样做之前，该函数应该检查offset的符号：如果它小于零，则抛出错误ERRCODE\_INVALID\_PRECEDING\_OR\_FOLLOWING\_SIZE (22013)外加“invalid preceding or following size in window function”这样的错误文本（这是SQL标准所要求的，不过非标准操作符族可能会选择忽视这一限制，因为似乎其语义必要性很小）。这种要求被委托给了in\_range函数，这样核心代码不需要理解对一种特定数据类型“less than zero”表示什么。

一个额外的期望是，如果可行，in\_range函数应当在base + offset或者base - offset溢出时避免抛出错误。即便值超过了该数据类型的范围，也可以确定正确的比较结果。注意，如果数据类型包括诸如“infinity”或者“NaN”之类的概念，就需要额外的注意确保in\_range的结果符合该操作符族的正常排序顺序。

in\_range函数的结果必须与操作符族施加的排序顺序保持一致。准确的说，给定任意固定的offset值以及sub值，那么：

- 如果带有less = true的in\_range对某个val1和base为真，则它必须对每一个有相同base的val2 <= val1为真。
  - 如果带有less = true的in\_range对某个val1和base为假，则它必须对每一个有相同base的val2 >= val1为假。
  - 如果带有less = true的in\_range对于某些val和base1为真，那么它对于每一个有相同val的base2 >= base1也必须为真。
  - 如果带有less = true的in\_range对于某些val和base1为假，那么它对于每一个有相同val的base2 <= base1也必须为假。
- 当less = false时，类似的具有相逆条件的语句成立。

如果被排序的类型（type1）是可排序的，合适的排序规则OID将使用标准的PG\_GET\_COLLATION()机制被传递给in\_range函数。

in\_range函数不需要处理NULL输入，并且通常将被标记为strict。

## equalimage

可选地，btree操作符系列可以提供equalimage（“equality implies image equality”）支持功能，注册在支持函数 4 下面。这些函数允许核心代码确定何时可以安全地应用 btree 重复数据消除进行优化。目前，equalimage函数仅在生成或重建索引时调用。

一个 equalimage 函数必须包含签名

```
equalimage(opcintype oid) returns bool
```

返回值是有关运算符类和排序规则的静态信息。返回true表示运算符类的 order 函数保证仅返回0（“arguments are equal”）时，A和B参数也可以互换，而不会丢失任何语义信息。不注册equalimage函数或返回false表示不能假定此条件成立。

opcintype参数是操作符类索引的数据类型的pg\_type.oid。这提供一种便利，允许在操作符类之间重用相同的底层equalimage函数。如果opcintype是可排序数据类型，则相应的排序规则OID将传递给equalimage函数，使用标准的PG\_GET\_COLLATION()机制。

就操作符类而言，返回true意味着重复数据删除是安全的（或安全排序规则，其OID传递给它的equalimage函数）。但是，核心代码只有在every索引列使用注册了equalimage函数的操作符类时，并且每个函数在调用时实际返回true时，才会认为重复数据删除是安全的。

图像相等几乎与简单位元(simple bitwise)相等的条件相同。有一个细微的区别：在索引 varlena 数据类型时，由于输入上TOAST压缩应用不一致，两个图像相等基准的磁盘

表示可能不是位相等的。从形式上看，当操作符类的 `equalimage` 函数返回 `true` 时，可以安全地假定 `datum_image_eq()` C 函数将始终与操作符类的 `order` 函数一致（前提是将同一个排序规则 OID 传递给 `equalimage` 和 `order` 函数）。

核心代码从根本上无法推断出任何关于基于同一系列中其他运算符类的详细信息的具有多数据类型系列中的操作符类的“equality implies image equality”状态。此外，操作符系列注册跨类型 `equalimage` 函数是不明智的，尝试这样做将导致错误。这是因为“equality implies image equality”状态不仅仅取决于排序/相等语义，这些语义或多或少在操作符系列级别定义。通常，特定数据类型实现的语义必须单独考虑。

核心 LightDB 分发中包含的操作符类所遵循的约定是注册股票 (`register a stock`)，通用 `equalimage` 函数。大多数操作符类注册 `btequalimage()`，这表明重复数据删除是安全的。可排序数据类型的操作符类（如 `text` 注册 `btvarstrequalimage()`，这表明在确定性排序规则下应用重复数据删除是安全的。第三方扩展的最佳实践是注册自己的自定义函数以保持控制。

options

可选地，B 树操作符系列可以提供 options（“操作符类指定选项”）支持功能，注册在支持函数 5 下面。这些函数定义一组用户可见的参数，用于控制操作符类行为。

options 支持函数必须具有签名

```
options(relopts local_relopts *) returns void
```

函数传递一个指向 `local_relopts` 结构的指针，需要用一组操作符类特定的选项来填充。这些选项可以通过 `PG_HAS_OPCLASS_OPTIONS()` 和 `PG_GET_OPCLASS_OPTIONS()` 宏，从其他支持的函数进行访问。

目前，没有 B-Tree 操作符类具有 options 支持功能。B-tree 不允许灵活表示键，例如 GiST，GIN。因此，options 在当前的 B-tree 索引访问方法中可能没有太多的应用。不过，此支持函数已添加到 B-tree 中以保持一致性，并且可能会在 LightDB 中 B-tree 的进一步演进中找到更多应用。

## 60.4. 实现

本节介绍 B-Tree 索引实现详细信息，这些对高级用户可能有用。更多信息请参见在源分发中的 `src/backend/access/nbtree/README` 文件，内部聚焦的 B-Tree 实现描述。

### 60.4.1. B-Tree 结构

LightDB B-Tree 索引是多级树结构，其中树的每个级别都可以用作双链接的页列表。单个元页存储在索引的第一个段文件开始时的固定位置。所有其他页都是叶页或内部页。叶页是在树的最低层上的页。所有其他层级由内部页组成。每个叶页都包含指向表行的元组。每个内部页都包含指向树中下一级别的元组。通常，超过 99% 的页面是叶页。内部页和叶页都使用 Section 64.6 中描述的标准页格式。

当已有叶页不能适应传入元组时，新叶页将添加到 B-Tree 索引中。 `page split` 操作通过将部分项目移动到新页面为最初属于溢出页上的项目提供空间。页拆分还必须插入新的 `downlink` 到父页中的新页，这可能会导致父页依次拆分。页拆分“cascade upwards”以递归的方式。当根页最终无法适合新的下行链接时，发生 `root page split` 操作。通过创建比原始根页高一个级别的新根页，为树结构添加新级别。

### 60.4.2. Deduplication

重复项是叶页元组（指向表行的元组），其中所有索引键列的值与同一索引中至少一个其他叶页元组的相应列值匹配。重复元组在实践中很常见。当启用可选技术：重复时，B-Tree 索引可以对重复项使用特殊的、节省空间的表达方式。

重复数据删除的工作为通过定期将重复元组合并在一起，为每个组构建一个posting list元组。列键值在此表示形式中仅显示一次。后面跟着指向表中行的TID的排序数组。这显著减少了索引的存储大小，在其中每个值（或列值的每个不同组合）平均出现多次时。查询的延迟可以显著降低。总体查询吞吐量可能会显著增加。常规索引清空的开销也可以显著降低。

### Note

B-Tree重复数据删除对于包含 NULL 值的“duplicates”同样有效，即使根据任何 B-Tree 操作符类的 = 成员，NULL 值永远不会彼此相等。对于理解磁盘上 B-Tree 结构的实现的任何部分，NULL 只是索引值域中的另一个值。

当插入的新项无法适应现有叶页时，重复数据删除过程进行缓慢，虽然只有当索引元组删除不能为新项释放足够的空间时（通常删除会被简单考虑然后跳过）不像 GIN 倒排列表元组，B-Tree倒排列表元组不需要在每次插入新重复项时展开；它们仅仅是叶页原始逻辑内容的替代物理表示方式。此设计优先考虑混合读写工作负载的一致性能。大多数客户端应用程序可以从使用重复数据删除中获得适度的性能收益。默认情况下，将启用重复数据删除。

CREATE INDEX和 REINDEX 应用重复数据删除来创建倒排列表元组，尽管它们使用的策略有所不同。每组重复的普通元组遇到从表中取出的排序输入将合并到倒排列表元组，在添加到当前挂起的叶页之前。单独倒排列表元组尽可能以TIDs封装。叶页以通常的方式写出，没有任何分开的重复数据删除步骤。此策略非常适合CREATE INDEX和REINDEX，因为它们是一次性的批处理操作。

由于索引中重复值很少或没有重复值，因此无法从重复数据删除中获益的写频繁工作负载将产生少量的、固定性能损耗（除非显式禁用重复数据删除）。deduplicate\_items存储参数可用于禁用单个索引中的重复数据消除。只读工作负载永远不会有性能损失，因为读取倒排列表元组至少与读取标准元组表示一样高效。禁用重复数据删除通常没有帮助。

有的时候惟一索引（以及惟一约束）可能被使用重复数据删除。这允许叶页临时“absorb”额外的版本混杂副本。惟一索引中的重复数据删除增强了bottom-up索引删除，特别是在长时间运行的事务持有块垃圾收集的快照的情况下。目的是为了bottom-up索引删除策略变得再次生效争取时间。延迟分页分割，直到单个长时间运行的事务自然消失，可以允许bottom-up删除传递成功，在较早的删除传递失败的地方。

### Tip

应用一种特殊的启发式方法来确定是否应在惟一索引中进行重复数据删除操作。它通常可以直接跳到拆分叶页，避免在无益的重复数据删除传递上浪费周期会降低性能。如果你担心重复数据删除的开销，可以考虑deduplicate\_items = off。在惟一索引中启用重复数据删除没有什么坏处。

由于实现级别的限制，不能在所有情况下使用重复数据消除。在CREATE INDEX 或 REINDEX时决定重复数据删除安全性。

请注意，重复数据删除被认为是不安全的，不能用于下列涉及相同数据之间语义显著差异的情况：

- text, varchar, 和 char在使用nondeterministic排序规则时不能使用重复数据删除。在等值基准之间必须保留大小写和重音差异。
- numeric重复数据删除。数字显示比例必须在相等的基准之间保留。
- jsonb不能使用重复数据消除，因为jsonbB-Tree操作符类在内部使用numeric。

- float4和float8不能使用重复数据删除。这些类型对 -0 和 0具有不同的表示形式，但被视为相等。必须保留此差异。

在未来版本的LightDB中，还有一个实现级限制可能取消：

- 容器类型（例如复合类型、数组或范围类型）不能使用重复数据删除。

无论使用操作符类或排序规则如何，还有一个实现级别限制适用：

- INCLUDE 索引不能使用重复数据删除。



# Chapter 61. GiST 索引

## 61.1. 简介

GiST表示通用搜索树。它是一种平衡的树结构的访问方法，它作为一种模板可用来实现任意索引模式。B 树、R 树和很多其他索引模式都可以在GiST中实现。

GiST的一个优势是它允许自定义数据类型的领域专家使用合适的访问方法开发自定义数据类型，而不是让数据库专家来开发。

这里的一些信息是来自加州大学伯克利分校的 GiST 索引项目网站<sup>1</sup>和 Marcel Kornacker 的学位论文Access Methods for Next-Generation Database Systems<sup>2</sup>。LightDB中的GiST实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护，在他们的网站<sup>3</sup>上有更多信息。

## 61.2. 内建操作符类

LightDB核心发布中包括如Table 61.1中所示的GiST操作符类（Appendix A中描述的一些可选模块提供了额外的GiST操作符类）。

Table 61.1. 内建GiST操作符类

名称	索引数据类型	可索引操作符	排序操作符
box_ops	box	&& &> &< &<  >> << <<  <@ @> @  &>  >> ~ ~ =	<->
circle_ops	circle	&& &> &< &<  >> << <<  <@ @> @  &>  >> ~ ~ =	<->
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =	
point_ops	point	>> >^ << <@ <@ <@ <^ ~ =	<->
poly_ops	polygon	&& &> &< &<  >> << <<  <@ @> @  &>  >> ~ ~ =	<->
range_ops	任何范围类型	&& &> &< >> << <@ - - = @> @>	
tsquery_ops	tsquery	<@ @>	
tsvector_ops	tsvector	@@	

由于历史原因，inet\_ops操作符类不是类型inet和cidr的默认操作符类。要使用它，需要在CREATE INDEX中指明操作符类的名称，例如

```
CREATE INDEX ON my_table USING GIST (my_inet_column inet_ops);
```

<sup>1</sup> <http://gist.cs.berkeley.edu/>

<sup>2</sup> <http://www.sai.msu.su/~megeera/postgres/gist/papers/concurrency/access-methods-for-next-generation.pdf.gz>

<sup>3</sup> <http://www.sai.msu.su/~megeera/postgres/gist/>

## 61.3. 可扩展性

在传统上，实现一种新的索引访问方法意味着很多困难的工作。开发者必须要理解数据库的内部工作，例如锁管理器和预写式日志。GiST接口有一个高层的抽象，要求访问方法实现者只实现要被访问的数据类型的语义。GiST层本身会处理并发、日志和对树结构的搜索。

这种可扩展性不应该与其他标准搜索树对于它们所处理的数据上的可扩展性混淆。例如，LightDB支持可扩展的 B 树和哈希索引。也就是说你可以用LightDB在任何你想要的数据类型上构建一个 B 树或哈希。但是 B 树只支持范围谓词 (<<、=、>>)，而哈希索引支持等值查询。

这样如果你用一个LightDB的 B 树索引一个图像集合，你只能发出例如“imagex 等于 imagey 吗”、“imagex 小于 imagey 吗”以及“imagex 大于 imagey 吗”的查询。取决于你如何在这种上下文中定义“等于”、“小于”和“大于”，这可能会有用。但是，通过使用一个基于GiST的索引，你可以创建提问领域相关问题的方法，可能是“找所有马的图片”或者“找所有曝光过度的图片”。

建立一个GiST访问方法并让其运行的所有工作是实现几个用户定义的方法，它们定义了树中键的行为。当然这些方法必须相当特别来支持特别的查询，但是对于所有标准查询（B 树、R 树等）它们相对直接。简而言之，GiST在可扩展性之上结合了通用型、代码重用和一个干净的接口。

一个用于GiST的索引操作符类必须提供五种方法，并且还有五种可选的方法。索引的正确性由正确实现的same、consistent和union方法保证，而索引的效率（尺寸和速度）将依赖于penalty和picksplit方法。两种可选的方法是compress和decompress，它们允许一个索引能对内部数据使用一种不同于被其索引的数据的类型。叶子是被索引的数据类型，而其他树结点可以是任何 C 结构（但是你仍必须遵循LightDB的数据类型规则，见用于可变尺寸数据的varlena）。如果树的内部数据类型在 SQL 层上存在，可以使用CREATE OPERATOR CLASS命令的STORAGE选项。可选的第八个方法是distance，如果操作符类希望支持有序扫描（最近邻搜索）就需要它。如果该操作符希望支持只用索引的扫描，则需要可选的第九个方法fetch，但compress方法被省略时不需要。如果操作符类提供了用户指定的参数，则需要第10个options方法选项。

consistent

给定一个索引项p和一个查询值q，这个函数决定该索引项是否与该查询“一致”，就是说：是否该索引项表示的行使得谓词“indexed\_columnindexable\_operator q”为真？对于一个叶子索引项，这等效于测试索引条件；而对于一个内部树结点，这会决定是否扫描由该树结点表示的索引子树。当结果为true时，还必须返回一个recheck标志。这指示该谓词一定为真或者只是可能为真。如果recheck = false那么该索引已经完全测试过该谓词条件，而如果recheck = true则该行只是一个候选匹配。在那种情况下，系统将根据实际的行值自动评估indexable\_operator来看它是否真的是一个匹配。这允许GiST同时支持有损和无损的索引结构。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid,
internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
```

```

{
    GISTENTRY *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type *key = DatumGetDataTypes(entry->key);
    bool retval;

    /*
     * 根据策略、键和查询确定返回值。
     *
     * 使用 GIST_LEAF(entry) 可以了解当前函数是在索引树的哪里被调用，
     * 这在支持例如 = 操作符时很方便（可以在非叶子节点中检查非空 union()
     * 以及在叶子节点中检查等值）。
     */

    *recheck = true; /* 如果检查是准确的则返回 false */

    PG_RETURN_BOOL(retval);
}

```

这里，key是该索引中的一个元素而query是在该索引中查找的值。StrategyNumber参数指示在你的操作符类中哪个操作符被应用 — 它匹配CREATE OPERATOR CLASS命令中的操作符编号之一。

取决于在操作符类中包含着哪些操作符，query的数据类型可能随着操作符而变化，因为它可能是该操作符右手边的任何类型，而这种类型可能和出现在其左手边的被索引数据类型不同（上面的代码框架假定只有一种类型；如果不是这样，取query参数值的方式可能必须取决于操作符）。我们推荐让consistent函数的 SQL 声明对query参数使用操作符类的被索引数据类型，即便实际类型可能是其他依赖于操作符的类型也是如此。

#### union

这个方法联合树中的信息。给定一组项，这个函数产生一个新的索引项，它表示所有给定的项。

该函数的SQL声明必须看起来像这样：

```

CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS storage_type
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

在 C 模块中匹配的代码则应该遵循这样的框架：

```

PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY *ent = entryvec->vector;
    data_type *out,
               *tmp,
               *old;
    int numranges,
        i = 0;

```

```

numranges = entryvec->n;
tmp = DatumGetDataType(ent[0].key);
out = tmp;

if (numranges == 1)
{
    out = data_type_deep_copy(tmp);

    PG_RETURN_DATA_TYPE_P(out);
}

for (i = 1; i < numranges; i++)
{
    old = out;
    tmp = DatumGetDataType(ent[i].key);
    out = my_union_implementation(out, tmp);
}

PG_RETURN_DATA_TYPE_P(out);
}

```

如你所见，在这个框架中我们处理一种数据类型 $\text{union}(X, Y, Z) = \text{union}(\text{union}(X, Y), Z)$ 。通过在这个GiST支持方法中实现正确的联合算法，支持不是这种情况的数据类型足够简单。

`union`函数的结果必须是该索引的存储类型的一个值，它可能与被索引列的类型不同，也可能相同。`union`函数应该返回一个指针指向新`palloc()`的内存。不能照原样返回输入值，即使没有类型改变也不能。

如上所示，`union`函数的第一个`internal`参数实际上是一个`GistEntryVector`指针。第二个参数是一个指向整数变量的指针，它可以被忽略（过去要求`union`函数将其结果值的尺寸存储在这个变量中，但现在这已不再必要）。

#### `compress`

把数据项转换成适合于一个索引页面中物理存储的格式。如果`compress`方法被省略，数据项会被不加修改地存储在索引中。

该函数的SQL声明必须看起来像这样：

```

CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

在 C 模块中匹配的代码则应该遵循这样的框架：

```

PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY *retval;

    if (entry->leafkey)
    {
        /* 用一个压缩版本替换 entry->key */

```

```

        compressed_data_type *compressed_data =
        palloc(sizeof(compressed_data_type));

        /* 从 entry->key ... 填充 *compressed_data */

        retval = palloc(sizeof(GISTENTRY));
        gistentryinit(*retval, PointerGetDatum(compressed_data),
                      entry->rel, entry->page, entry->offset, FALSE);
    }
    else
    {
        /* 通常我们不需要对非叶子项做任何事情 */
        retval = entry;
    }

    PG_RETURN_POINTER(retval);
}

```

当然，为了压缩你的叶结点，你必须把`compressed_data_type`改编成你正在转换到的指定类型。

#### decompress

将一个数据项的存储表达转换成该操作符类中其他GiST方法能够操纵的格式。如果`decompress`方法被省略，则假设其他GiST方法能够直接在存储的数据格式上工作（`decompress`不一定是`compress`方法的逆操作，特别是如果`compress`是有损的，那么`decompress`是不可能准确地重构原始数据的。`decompress`也不一定与`fetch`等效，因为其他GiST方法可能不需要数据的完整重构）。

该函数的SQL声明必须看起来像这样：

```

CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

在 C 模块中匹配的代码则应该遵循这样的框架：

```

PG_FUNCTION_INFO_V1(my_decompress);

Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}

```

上述框架适合于不需要解压的情况（但是，将该方法一并省去当然更加容易，并且在这种情况下推荐这样做）。

#### penalty

返回一个值，它指示在树的一个特定分支插入新项的“代价”。项将被插入到树中具有最小`penalty`的路径中。`penalty`返回的值应该为非负。如果一个赋值被返回，它将被当作零来处理。

该函数的SQL声明必须看起来像这样：

```

CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal

```

```
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT; -- in some cases penalty functions need not be strict
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float *penalty = (float *) PG_GETARG_POINTER(2);
    data_type *orig = DatumGetDataType(origentry->key);
    data_type *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

由于历史原因，penalty函数不只是返回一个float结果，而是必须把该值存储在由第三个参数指定的位置。虽然传回该参数的地址符合惯例，但返回值本身可以被忽略。

penalty函数对于索引的好性能是至关重要的。在插入时，当要选择树中的哪个位置加入新项时，这个函数有助于决定应该顺着哪个分支进行。在查询时，索引越平衡，查找越快。

#### picksplit

当需要一次索引页面分裂时，这个函数决定在该页面上哪些项会留在旧页面上，以及哪些项会移动到新页面上。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY *ent = entryvec->vector;
    int i,
        nbytes;
    OffsetNumber *left,
        *right;
    data_type *tmp_union;
    data_type *unionL;
    data_type *unionR;
    GISTENTRY **raw_entryvec;
```

```

maxoff = entryvec->n - 1;
nbytes = (maxoff + 1) * sizeof(OffsetNumber);

v->spl_left = (OffsetNumber *) palloc(nbytes);
left = v->spl_left;
v->spl_nleft = 0;

v->spl_right = (OffsetNumber *) palloc(nbytes);
right = v->spl_right;
v->spl_nright = 0;

unionL = NULL;
unionR = NULL;

/* 初始化裸的项向量。 */
raw_entryvec = (GISTENTRY **) malloc(entryvec->n * sizeof(void *));
for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
    raw_entryvec[i] = &(entryvec->vector[i]);

for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
{
    int          real_index = raw_entryvec[i] - entryvec->vector;

    tmp_union = DatumGetDataType(entryvec->vector[real_index].key);
    Assert(tmp_union != NULL);

    /*
     * 选择在哪里放置索引项并且相应地更新 unionL 和 unionR。
     * 把项追加到 v->spl_left 或者 v->spl_right, 并且设置好计数器。
     */

    if (my_choice_is_left(unionL, curl, unionR, curr))
    {
        if (unionL == NULL)
            unionL = tmp_union;
        else
            unionL = my_union_implementation(unionL, tmp_union);

        *left = real_index;
        ++left;
        ++(v->spl_nleft);
    }
    else
    {
        /*
         * 和在右边的过程相同
         */
    }
}

v->spl_ldatum = DataTypeGetDatum(unionL);
v->spl_rdatum = DataTypeGetDatum(unionR);
PG_RETURN_POINTER(v);
}

```

注意 `picksplit` 函数的结果通过修改传入的 `v` 结构来传递。尽管传回 `v` 的地址符合惯例，但返回值本身可以被忽略。

和penalty一样，picksplit函数对于索引的好性能至关重要。设计合适的penalty和picksplit是实现一个好的GiST索引中最大的挑战。

same

如果两个索引项相同则返回真，否则返回假（一个“索引项”是该索引的存储类型的一个值，而不一定是原始被索引列类型的值）。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_same(storage_type, storage_type, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_same);
```

Datum

```
my_same(PG_FUNCTION_ARGS)
```

```
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool          *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

由于历史原因，same函数不只返回一个布尔结果。相反它必须把该标志存储在第三个参数指示的位置。尽管传回该参数的地址符合惯例，但返回值本身可以被忽略。

distance

给定一个索引项p和一个查询值q，这个函数决定两者之间的“距离”。如果操作符类包含任何排序操作符，就必须提供这个函数。一个使用排序操作符的查询将首先返回具有最小“距离”值的索引项，因此结果必须与操作符的语义一致。对于一个页索引项，结果只表示到索引项的距离；对于一个内部树结点，结果必须是到任何子项的最小距离。

该函数的SQL声明必须看起来像这样：

```
CREATE OR REPLACE FUNCTION my_distance(internal, data_type, smallint, oid,
internal)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

在 C 模块中匹配的代码则应该遵循这样的框架：

```
PG_FUNCTION_INFO_V1(my_distance);
```

Datum

```
my_distance(PG_FUNCTION_ARGS)
```

```
{
    GISTENTRY *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
```



```

/* Oid subtype = PG_GETARG_OID(3); */
/* bool *recheck = (bool *) PG_GETARG_POINTER(4); */
data_type *key = DatumGetDataTypes(entry->key);
double     retval;

/*
 * determine return value as a function of strategy, key and query.
 */

PG_RETURN_FLOAT8(retval);
}

```

distance函数的参数和consistent函数的相同。

在决定距离时允许有某种近似，只要结果不要超过该项的实际距离即可。因此，例如在几何应用中到一个外包盒的距离就足够了。对于一个内部树节点，返回的距离不能超过到其任意一个子节点的距离。如果返回的距离不准确，该函数必须设置\*recheck为真（这对于内部树节点是不必要的，对于它们，计算总是被假定为不准确）。在这种情况下，执行器将在从堆中取出元组后计算精确的距离，并且在必要时记录这些元组。

如果距离函数对任意叶子节点都返回\*recheck = true，初始的排序操作符的返回类型必须是float8或者float4，并且距离函数的结果值必须能和初始排序操作符的结果进行比较，因为执行器将使用距离函数结果和重新计算的排序操作符结果进行排序。否则，该距离函数的结果值可以是任意有限的float8值，只要这些结果值的相对顺序匹配该排序操作符返回的顺序（在内部会使用无穷以及负无穷来处理空值等情况，因此我们不推荐distance函数返回这些值）。

## fetch

为只用索引的扫描将一个数据项压缩过的索引表达转换成原始的数据类型。被返回的数据必须是原始被索引值的一份准确的、非有损的拷贝。

该函数的SQL声明必须看起来像这样：

```

CREATE OR REPLACE FUNCTION my_fetch(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

参数是一个指向GISTENTRY结构的指针。在项上，它的key域包含一个压缩形式的非-NULL叶子数据。返回值是另一个GISTENTRY结构，其key域包含同一数据的原始的未压缩形式。如果操作符类的压缩函数不对叶子项做任何事情，fetch方法可以原样返回参数。或者，如果该opclass没有一个压缩函数，则fetch方法也可以被省略，因为它必须是一个空操作。

C 模块中相应的代码可能会遵循下面的框架：

```

PG_FUNCTION_INFO_V1(my_fetch);

Datum
my_fetch(PG_FUNCTION_ARGS)
{
    GISTENTRY *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    input_data_type *in = DatumGetPointer(entry->key);
    fetched_data_type *fetched_data;
    GISTENTRY *retval;

    retval = palloc(sizeof(GISTENTRY));
    fetched_data = palloc(sizeof(fetched_data_type));

```

```

/*
 * 将 'fetched_data' 转换成原始数据类型的一个 Datum。
 */

/* 从 fetched_data. 填充 *retval. */
gistentryinit(*retval, PointerGetDatum(converted_datum),
              entry->rel, entry->page, entry->offset, FALSE);

PG_RETURN_POINTER(retval);
}

```

如果该压缩方法对于叶子项是有损的，操作符类就不能支持只用索引的扫描，并且不能定义fetch函数。

### 选项

允许定义控制操作符类行为的用户可见参数。

该函数的SQL声明必须看起来如下所示：

```

CREATE OR REPLACE FUNCTION my_options(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

向函数传递一个指向local\_relopts结构体的指针，该结构体需要用一组特定于操作符类的选项来填充。可以使用PG\_HAS\_OPCLASS\_OPTIONS() 和 PG\_GET\_OPCLASS\_OPTIONS() 宏从其他支持函数访问这些选项。

下面给出了my\_options()和其他支持函数参数的示例实现：

```

typedef enum MyEnumType
{
    MY_ENUM_ON,
    MY_ENUM_OFF,
    MY_ENUM_AUTO
} MyEnumType;

typedef struct
{
    int32   vl_len_; /* varlena header (do not touch directly!) */
    int     int_param; /* integer parameter */
    double  real_param; /* real parameter */
    MyEnumType enum_param; /* enum parameter */
    int     str_param; /* string parameter */
} MyOptionsStruct;

/* String representation of enum values */
static relopt_enum_elt_def myEnumValues[] =
{
    {"on", MY_ENUM_ON},
    {"off", MY_ENUM_OFF},
    {"auto", MY_ENUM_AUTO},
    {(const char *) NULL} /* list terminator */
};

static char *str_param_default = "default";

```

```

/*
 * Sample validator: checks that string is not longer than 8 bytes.
 */
static void
validate_my_string_relopt(const char *value)
{
    if (strlen(value) > 8)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_PARAMETER_VALUE),
                 errmsg("str_param must be at most 8 bytes")));
}

/*
 * Sample filler: switches characters to lower case.
 */
static Size
fill_my_string_relopt(const char *value, void *ptr)
{
    char    *tmp = str_tolower(value, strlen(value), DEFAULT_COLLATION_OID);
    int     len = strlen(tmp);

    if (ptr)
        strcpy((char *) ptr, tmp);

    pfree(tmp);
    return len + 1;
}

PG_FUNCTION_INFO_V1(my_options);

Datum
my_options(PG_FUNCTION_ARGS)
{
    local_relopts *relopts = (local_relopts *) PG_GETARG_POINTER(0);

    init_local_reloptions(relopts, sizeof(MyOptionsStruct));
    add_local_int_reloption(relopts, "int_param", "integer parameter",
                           100, 0, 1000000,
                           offsetof(MyOptionsStruct, int_param));
    add_local_real_reloption(relopts, "real_param", "real parameter",
                              1.0, 0.0, 1000000.0,
                              offsetof(MyOptionsStruct, real_param));
    add_local_enum_reloption(relopts, "enum_param", "enum parameter",
                              myEnumValues, MY_ENUM_ON,
                              "Valid values are: \"on\", \"off\" and \"auto
\".",
                              offsetof(MyOptionsStruct, enum_param));
    add_local_string_reloption(relopts, "str_param", "string parameter",
                               str_param_default,
                               &validate_my_string_relopt,
                               &fill_my_string_relopt,
                               offsetof(MyOptionsStruct, str_param));

    PG_RETURN_VOID();
}

PG_FUNCTION_INFO_V1(my_compress);

```

```

Datum
my_compress(PG_FUNCTION_ARGS)
{
    int    int_param = 100;
    double real_param = 1.0;
    MyEnumType enum_param = MY_ENUM_ON;
    char   *str_param = str_param_default;

    /*
     * Normally, when opclass contains 'options' method, then options are
     always
     * passed to support functions. However, if you add 'options' method to
     * existing opclass, previously defined indexes have no options, so the
     * check is required.
     */
    if (PG_HAS_OPCLASS_OPTIONS())
    {
        MyOptionsStruct *options = (MyOptionsStruct *)
PG_GET_OPCLASS_OPTIONS();

        int_param = options->int_param;
        real_param = options->real_param;
        enum_param = options->enum_param;
        str_param = GET_STRING_RELOPTION(options, str_param);
    }

    /* the rest implementation of support function */
}

```

由于GiST中的键表示是弹性的，它可能依赖于用户指定的参数。例如，可以指定密钥签名的长度。示例可参见gtsvector\_options()。

所有的 GiST 支持方法通常都在一个短暂存在的内存上下文中被调用；就是说，每个元组被处理之后CurrentMemoryContext将被重置。因此没有必要操心释放你 palloc 的所有东西。但是，在某些情况下，一个支持方法在重复调用之间缓存数据是有用的。要这样做，将这些长期生存的数据分配在fcinfo->flinfo->fn\_mcxt中，并且在fcinfo->flinfo->fn\_extra中保持一个到它的指针。这种数据将在索引操作期间都存在（例如一次 GiST 索引扫描、索引构建或索引元组插入）。注意当替换一个fn\_extra值时要释放之前的值，否则在操作期间该泄露会累积。

## 61.4. 实现

### 61.4.1. GiST 缓冲构建

通过简单地插入所有元组来构建大型 GiST 索引很容易变得很慢，因为如果索引元组分散在索引中并且索引大到不足以放入在缓存中，插入操作需要执行很多随机 I/O。从版本 9.2 开始，LightDB 支持一种更有效率的方法来基于缓冲构建 GiST 索引，这能显著地减少用于非排序数据集所需的随机 I/O 数量。对于排序好的数据集这种收益很小甚至不存在，因为在那时只有少数页面会接收新元组，并且那些页面能放在缓存中（即便整个索引不能放在缓存中）。

但是，缓冲索引构建需要更频繁地调用penalty函数，这会消耗更多额外的 CPU 资源。还有，在缓冲构建中使用的缓冲区需要临时磁盘空间，最多为结果索引的尺寸。缓冲也可能影响结果索引的质量，不管是正向还是负向。这种影响取决于多种因素，如输入数据的分布和操作符类的实现。

默认情况下，当索引尺寸达到`effective_cache_size`时，一个 GiST 索引构建会切换到缓冲方法。可以通过 `CREATE INDEX` 命令的`buffering`参数手工打开或关闭这个特性。默认行为对大部分情况是好的，但是如果输入数据是排序好的，关闭缓冲特性可能会加速构建过程。

## 61.5. 示例

LightDB源码包括了多个用GiST实现的索引方法的例子。核心系统当前提供文本搜索支持（用于`tsvector`和`tsquery`的索引）以及用于一些内建几何数据类型（`src/backend/access/gist/gistproc.c`）的 R 树等效功能。下列`contrib`模块也包含GiST操作符类：

`btree_gist`

多种数据类型的 B 树等效功能

`cube`

多维立方体的索引

`hstore`

存储键值对的模块

`intarray`

一维 `int4` 值数组的 RD 树

`ltree`

树状结构的索引

`lt_trgm`

使用 `trigram` 匹配的文本相似性

`seg`

“float ranges” 的索引

---

# Chapter 62. GIN 索引

## 62.1. 简介

GIN意思是通用倒排索引。GIN被设计为处理被索引项为组合值的情况，并且这种索引所处理的查询需要搜索出现在组合项中的元素值。例如，项可以是文档，并且查询可以是搜索包含指定词的文档。

我们使用词项来表示要被索引的一个组合值，并且用词键来表示一个元素值。GIN总是存储和搜索键，而不是项值本身。

一个GIN存储一个（键，位置列表）对的集合，这里一个posting list是键在其中出现的一个行 ID 的集合。相同的行 ID 可以出现在多个位置列表中，因为一个项可以包含多于一个键。每个键值只被存储一次，因此对于同一个键出现多次的情况，一个GIN索引是非常紧凑的。

GIN访问方法代码不需要知道它所加速的是什么操作，从这个意义上来说，GIN是通用的。相反，它使用为特定数据类型定义的自定义策略。策略定义如何从被索引项和查询条件中抽取键，并且如何决定一个包含查询中某些键值的行是否真正满足查询。

GIN的一个优点是它允许由数据类型的领域专家开发有合适访问方法的自定义数据类型，而不是让一个数据库专家来做这件事。在这一点上很像GiST。

LightDB中的GIN实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护。在他们的网站<sup>1</sup>上有更多关于GIN的信息。

## 62.2. 内建操作符类

LightDB的核心发布包括Table 62.1中所示的GIN操作符类（Appendix 中描述的一些可选模块提供了额外的GIN操作符类）。

Table 62.1. 内建GIN操作符类

Name	Indexed Data Type	Indexable Operators
array_ops	anyarray	&& <@ = @>
jsonb_ops	jsonb	? ?& ?  @> @? @@
jsonb_path_ops	jsonb	@> @? @@
tsvector_ops	tsvector	@@ @@@

在两种用于类型jsonb的操作符类中，jsonb\_ops是默认项。jsonb\_path\_ops支持较少的操作符但是为那些操作符提供了更好的性能。详见Section 8.14.4

## 62.3. 可扩展性

GIN接口有一个高层次的抽象，要求访问方法实现者只需要实现数据类型被访问的语义。GIN层本身会操心并发、日志和搜索树结构的事情。

要让一个GIN访问方法工作起来所要做的全部事情就是实现一些用户定义的方法，它们定义了树中键的行为以及键、被索引项以及可索引查询之间的关系。简而言之，GIN的可扩展性结合了通用性、代码重用和一个干净的接口。

一个用于GIN的操作符类必须提供的两种方法是：

---

<sup>1</sup> <http://www.sai.msu.su/~megeera/wiki/Gin>

```
Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)
```

给定一个要被索引的项，返回一个 `palloc` 过的键的数组。被返回的键的数量必须被存储在 `*nkeys` 中。如果键中的任意一个可能为空，还要 `palloc` 一个 `*nkeys` 个 `bool` 域的数组，将它的地址存储在 `*nullFlags` 中，并且根据需要设置这些空值标志。如果所有的键都非空，`*nullFlags` 可以被留成 `NULL`（其初始值）。如果该项不包含键，返回值可以为 `NULL`。

```
Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch,
Pointer **extra_data, bool **nullFlags, int32 *searchMode)
```

给定一个要被查询的值，返回一个 `palloc` 过的键的数组。即 `query` 是一个可索引操作符（左手边是被索引列）的右手边的值。`n` 是操作符类中操作符的策略号（见 Section 36.16.2）。通常，`extractQuery` 将需要参考 `n` 来判断 `query` 的数据类型以及它应该用什么方法来抽取键值。被返回的键的数量必须被存储在 `*nkeys` 中。如果键中的任意一个可能为空，还要 `palloc` 一个 `*nkeys` 个 `bool` 域的数组，将它的地址存储在 `*nullFlags` 中，并且根据需要设置这些空值标志。如果所有的键都非空，`*nullFlags` 可以被留成 `NULL`（其初始值）。如果该项不包含键，返回值可以为 `NULL`。

`searchMode` 是一个输出参数，它允许 `extractQuery` 指定有关搜索如何被完成的细节。如果 `*searchMode` 被设置为 `GIN_SEARCH_MODE_DEFAULT`（这是在被调用之前它被初始化的值），只有那些匹配至少一个被返回键的项才会被考虑作为候选匹配。如果 `*searchMode` 被设置为 `GIN_SEARCH_MODE_INCLUDE_EMPTY`，那么除了至少包含一个匹配键的项之外，根本不包含键的项也被考虑作为候选匹配（例如，这种模式对于实现“是...的子集”操作符有用）。如果 `*searchMode` 被设置为 `GIN_SEARCH_MODE_ALL`，那么索引中所有非空项都被考虑作为候选匹配，不管它们是否匹配被返回的键（这种模式比其他两种选择要慢很多，但是它对于正确实现极端情况可能是必要的。需要这种模式的操作符在大部分情况下可能并不是一个 GIN 操作符类的好选择）。用于设置这个模式的符号被定义在 `access/gin.h` 中。

`pmatch` 是一个输出参数，它用于在部分匹配匹配被支持时使用。要用它，`extractQuery` 必须分配一个 `*nkeys` 个布尔值的数组，并且把它的地址存储在 `*pmatch` 中。如果一个键要求部分匹配，该数组的对应元素应该被设置为 `TRUE`，否则设置为 `FALSE`。如果 `*pmatch` 被设置为 `NULL`，则 GIN 假定不需要部分匹配。在调用前，该变量被初始化为 `NULL`，这样这个参数可以简单地被不支持部分匹配的操作符类忽略。

`extra_data` 是一个输出参数，它允许 `extractQuery` 传递额外数据给 `consistent` 和 `comparePartial` 方法。要用它，`extractQuery` 必须分配一个 `*nkeys` 个指针的数组，并且把它的地址存储在 `*extra_data` 中，然后把任何它想存储的东西存到单个指针中。在调用前该变量被初始化为 `NULL`，这样这个参数可以简单地被不需要额外数据的操作符类忽略。如果 `*extra_data` 被设置，整个数组被传递给 `consistent` 方法，并且适当的元素会被传递给 `comparePartial` 方法。

一个操作符类必须提供一个函数检查一个被索引的项是否匹配查询。有两种形式，一个布尔函数 `consistent`，以及一个三元函数 `triConsistent`。`triConsistent` 覆盖了两者的功能，因此提供 `triConsistent` 一个足矣。但是，如果布尔变体的计算代价要更低，两者都提供就会有好处。如果只提供布尔变体，一些基于在取得所有键之前拒绝索引项的优化将会被禁用。

```
bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer
extra_data[], bool *recheck, Datum queryKeys[], bool nullFlags[])
```

如果一个被索引项满足（如果重新检查指示被返回，则表示可能满足）有策略号 `n` 的查询操作符，则返回 `TRUE`。这个函数并没有直接访问被索引项的值，因为 GIN 没有显式存储项。可用的是关于哪些从查询抽取出的键值出现在一个给定被索引项中的知识。`check` 数组的长度是 `nkeys`，它和前面由 `extractQuery` 为这个查询数据返回的键的数目相同。如果被索引项包含一个查询键，那么 `check` 数组的对应元素为 `TRUE`，即如果 (`check[i] == TRUE`)，则 `extractQuery` 结果数组的第 `i` 个键存在于被索引项中。在 `consistent` 方法需

要参考原始query数据的情况中，它会被传递进来，前面由extractQuery返回的queryKeys[]和nullFlags[]数组也一样。extra\_data是由extractQuery返回的额外数据数组，如果没有额外数据则为NULL。

当extractQuery在queryKeys[]中返回一个空值键时，如果被索引项包含一个空值键则对应的check[]元素为 TRUE。即，check[]的语义类似IS NOT DISTINCT FROM。如果consistent函数需要说出一个常规值匹配和一个空值匹配之间的区别，它可以检查对应的nullFlags[]元素。

在成功时，如果堆元组需要根据查询操作符被重新检查，则\*recheck应该被设置为 TRUE，或者如果索引测试是准确的则设置为 FALSE。即，一个 FALSE 返回值保证堆元组不匹配查询；一个 TRUE 返回值以及设置为 FALSE 的\*recheck保证堆元组匹配查询；并且一个 TRUE 返回值和设置为 TRUE 的\*recheck表示堆元组可能匹配查询，因此它需要被取出并且通过在原始的被索引项上计算查询操作符来重新检查。

```
GinTernaryValue triConsistent(GinTernaryValue check[], StrategyNumber n, Datum
query, int32 nkeys, Pointer extra_data[], Datum queryKeys[], bool nullFlags[])
```

triConsistent类似于consistent，但和check[]中的布尔值不同，对每个键有三种可能值：GIN\_TRUE、GIN\_FALSE和GIN\_MAYBE。GIN\_FALSE和GIN\_TRUE具有和常规布尔值相同的含义，而GIN\_MAYBE意味着键的存在未知。当GIN\_MAYBE值出现时，如果项必定匹配（不管该索引项是否包含对应的查询键），该函数应该只返回GIN\_TRUE。同样地，如果项必定不匹配（不管它是否包含GIN\_MAYBE），该函数必须只返回GIN\_FALSE。如果结果依赖于GIN\_MAYBE项，即无法根据已知查询键确认或拒绝匹配，该函数必须返回GIN\_MAYBE。

当在check向量中没有GIN\_MAYBE值时，GIN\_MAYBE返回值等效于在布尔函数consistent中设置 recheck标志等效。

此外，GIN必须有方法能排序存储在索引中的键值。操作符类可以通过指定一种比较方法来定义排序顺序：

```
int compare(Datum a, Datum b)
```

比较两个键（不是被索引项！）并且返回一个整数，该整数为小于零、等于零或者大于零分别表示第一个键小于、等于或者大于第二个键。空值键绝不会被传递给这个函数。

或者，如果操作符类不提供compare方法，GIN将查看索引键数据类型的默认btree操作符类，并且使用它的比较函数。推荐为只用于一种数据类型的GIN操作符类指定这个比较函数，因为查找btree操作符类需要消耗一些周期。不过，多态的GIN操作符类（例如array\_ops）通常无法指定单一的比较函数。

一个用于GIN的操作符类可以选择性的提供下列方法：

```
int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer
extra_data)
```

比较一个部分匹配键和一个索引键。返回一个整数，其符号指示结果：小于零表示索引键不匹配查询，但是索引扫描应该继续；零表示索引键匹配查询；大于零表示索引扫描应该停止，因为没有更多可能的匹配。产生该部分匹配查询的操作符的策略号n将被提供，可以通过其语义决定什么时候结束扫描。还有，extra\_data是由extractQuery产生的额外数据数组中的对应元素，如果没有则为NULL。空值不会被传递给这个函数。

```
void options(local_relopts *relopts)
```

定义一组用户可见的参数以控制操作符类的行为。

options 函数传递一个指针到一个local\_relopts结构，该结构需要用一组特定的操作符类的选项来填充。该选项可以使



用PG\_HAS\_OPCLASS\_OPTIONS()和PG\_GET\_OPCLASS\_OPTIONS()宏，从其他支持的函数进行访问。

由于索引值的键提取和键在GIN中的表示都是灵活的，它们可能取决于用户指定的参数。

要支持“部分匹配”查询，一个操作符类必须提供comparePartial方法，并且它的extractQuery方法必须在遇到一个部分匹配查询时设置pmatch参数。详见Section 62.4.2

上面提到的多个Datum值的实际数据类型随着操作符类而变化。被传递给extractValue的项值总是操作符类的输入类型，并且所有的键值必须是类的STORAGE类型。被传递给extractQuery、consistent和triConsistent的query参数是由该策略号标识的类成员操作符的右边输入类型。这不需要和被索引类型相同，只要正确类型的键值能从其中被抽取出来。不过，推荐这三个支持函数的SQL声明对query参数使用操作符类的被索引数据类型，即便实际类型可能是某种其他依赖于操作符的东西时也应如此。

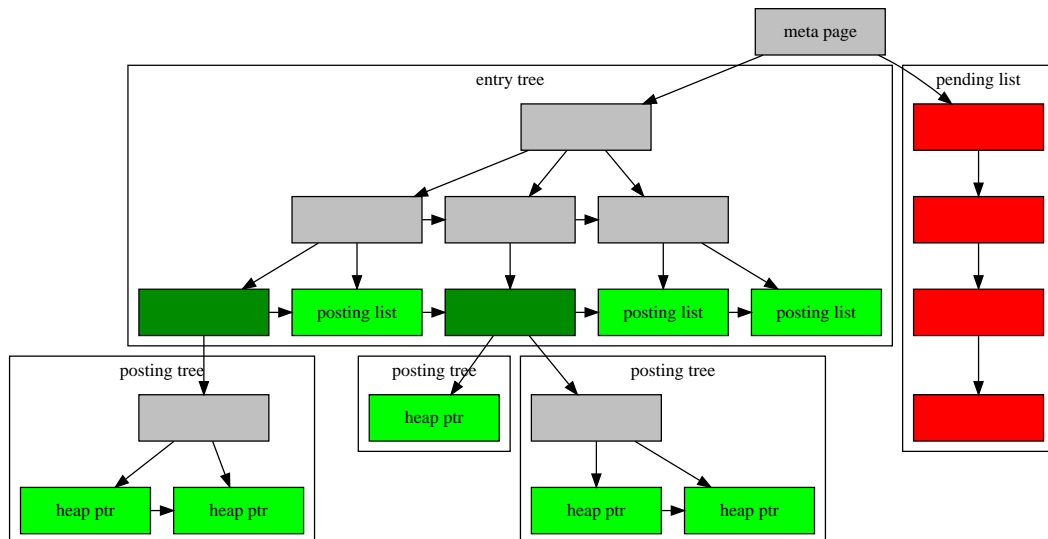
## 62.4. 实现

在内部，一个GIN索引包含一个在键上构建的B树索引，其中每一个键是一个或者多个被索引项的一个元素（例如，数组的一个成员），并且叶子页中的每一个元组包含一个指向堆指针B树的指针（一个“位置树”）或者一个堆指针的简单列表（“位置列表”），只有位置列表小到能够和键值一起放入索引时才使用后一种形式。Figure 62. 举例说明了这些GIN索引的组件。

自LightDB 9.1 起，空键值可以被包括在索引中。同样，用于为空或者根据extractValue不包含键的被索引项的占位符空值也被包括在索引中。这允许实现应该找到空项的搜索。

多列GIN索引可以通过在组合值（列号，键值）上建立一个单一B树实现。不同列的键值可以是不同类型。

Figure 62.1. GIN内部



### 62.4.1. GIN 快速更新技术

更新GIN索引往往会很慢，由于反向索引的固有特性：插入或更新一个heap row会导致许多项目插入到索引中（每个索引键从索引项目中提取一个）。GIN能够通过将新的tuple插入临时的未排序的待处理条目列表来延迟大部分工作。当表被清理或自动分析时，或者gin\_clean\_pending\_list函数被调用时，又或者待处理列表变得大于gin\_pending\_list\_limit时，使用在初始索引建立期间使用的相同批次插入技术将项目

移动到主要的GIN数据结构中。即使考虑到额外的清理开销，这也显著加快了GIN索引更新速度。此外，后台进程可以执行这种开销工作，而不是前端查询处理来完成。

这种方式的主要缺点是搜索必须在搜索普通索引之外扫描待处理条目的列表，并且因此一个大型的待处理条目列表会显著地拖慢搜索。另一个缺点是，虽然大部分更新变快了，一次导致待处理列表变得“太大”的更新将导致一次立即清理循环并且因此会比其他更新慢很多。正确使用自动清理可以把这些问题的影响变得最小。

如果一致的响应时间比更新速度更重要，可以通过为一个GIN关闭fastupdate存储参数来禁用对待处理条目的使用。详见CREATE INDEX。

## 62.4.2. 部分匹配算法

GIN 可以支持“部分匹配”查询，在其中查询不能判断一个或多个键的精确匹配，但是可以确定落在键值（在compare支持方法决定的键排序顺序中）的一个合理的狭窄范围内的可能匹配。extractQuery方法，不会返回一个要被精确匹配的键值，而是返回一个作为要被搜索范围下界的键值，并且将pmatch标志设置为真。然后键范围将被使用comparePartial方法扫描。comparePartial必须对于一个匹配的索引键返回零，对于一个不匹配但仍在要被搜索的范围内的返回小于零，对于超过被搜索范围的索引键返回大于零。

## 62.5. GIN 提示和技巧

### 创建 vs. 插入

插入到一个GIN索引可能会很慢，因为为一个项可能需要插入很多歌键。因此，对于一个表的批量插入，我们建议删除 GIN 索引，然后在完成批量插入后重建它。

当 GIN 启用 fastupdate 时（参见 Section 62.4.1），惩罚比其他情况要小。但是对于非常大的更新，删除并重新创建索引可能仍然是最好的。

### maintenance\_work\_mem

一个GIN索引的建立时间对maintenance\_work\_mem设置很敏感；它不考虑在索引创建期间在工作内存上的节省。

### gin\_pending\_list\_limit

在向一个已有的开启了fastupdate的GIN 中进行插入时，系统将在待处理项列表增长到超过gin\_pending\_list\_limit 时清理该列表。为了避免在观测到的响应时间上出现波动，让待处理列表清理操作 在后台进行（即通过 autovacuum）比较好。可以通过增加gin\_pending\_list\_limit 或者让 autovacuum 更激进来避免前台的清理操作。不过，增大清理操作的 阈值意味着如果一次前台清理发生，它将需要更长的时间。

可以通过改变存储参数为每个 GIN 索引覆盖它所用的gin\_pending\_list\_limit，这样允许每个 GIN 索引拥有自己的清理阈值。例如，可以只为被大量更新的 GIN 索引增加该阈值，而对其他的索引减小该阈值。

### gin\_fuzzy\_search\_limit

开发GIN索引的主要目的是在LightDB中创建对高可扩展全文搜索的支持，并且一次全文搜索返回一个非常大的结果集也是很常见的情况。此外，当查询包含非常频繁的单词时情况也是如此，因此大结果集不是非常有用。因为从磁盘读取很多元组并且对它们排序可能会花费很多时间，这对于产品来说是不可接受的（注意索引搜索本身是很快的）。

为了能够控制这类查询的执行，GIN对于返回的行数有一个可配置的软上限：gin\_fuzzy\_search\_limit配置参数。默认它被设置为 0（意为无限制）。如果设置了一个非零限制，那么被返回的集合是整个结果集的一个子集，并且是随机选择的。

“软”意味着被返回结果的实际数量可以与指定的限制不同，这取决于查询和系统随机数生成器的质量。

根据经验，在数千级别的值（如 5000 — 20000）比较好。

## 62.6. 限制

GIN假定可索引操作符是严格的。这意味着对于一个空项值，`extractValue`将根本不会被调用（相反，一个占位符索引项将被自动创建），并且在一个空查询值上也不会调用`extractQuery`（相反，该查询被假定为不可满足的）。不过注意，在一个非空组合项或查询值中的空键值是被支持的。

## 62.7. 例子

LightDB的核心发布中包括之前Table 62. 展示的GIN操作符类。`tsvector`中的前缀搜索就是使用GIN的部分匹配特性实现的。下列`contrib`模块也包含GIN操作符类：

`btree_gin`

多种数据类型的 B 树等效功能

`hstore`

存储键值对的模块

`intarray`

`int[]`的增强支持

`lt_trgm`

使用 `trigram` 匹配的文本相似性

---

# Chapter 63. 哈希索引

## 63.1. 概述

LightDB 包含永久性磁盘上的哈希索引的实现，这些索引可完全崩溃恢复。任何数据类型都可以通过哈希索引进行索引，包括没有明确定义线性排序的数据类型。哈希索引只存储被索引数据的哈希值，因此对被索引数据列的大小没有限制。

哈希索引仅支持单列索引并且不允许唯一性检查。

哈希索引仅支持=运算符，因此使用了范围操作的WHERE子句将无法利用哈希索引。

每个哈希索引元组只存储4字节的哈希值，而不是实际的列值。因此，当索引较长的数据项（如UUID、URL等）时，哈希索引可能比B树小得多。缺少列值也会使所有哈希索引扫描有损。哈希索引可以参与位图索引扫描和反向扫描。

哈希索引对于在较大表上使用相等扫描的SELECT和UPDATE的繁重工作进行了最佳优化。在B树索引中，搜索必须在树中向下搜索，直到找到叶子页。在具有数百万行的表中，这种向下搜索会增加数据访问时间。哈希索引中叶子页的等价物称为桶页。相反，哈希索引允许直接访问桶页，从而可能减少较大表中的索引访问时间。在大于shared\_buffers/RAM的索引/数据上，这种“逻辑I/O”的减少变得更加明显。

哈希索引被设计用于处理哈希值的不均匀分布。如果哈希值均匀分布，则直接访问桶页效果良好。当插入使得桶页变满时，额外的溢出页链接到该特定的桶页，本地扩展存储来容纳与该哈希值匹配的索引元组。在查询期间扫描哈希桶时，我们需要扫描所有溢出页。因此，就某些数据所需的块访问数而言，不平衡散列索引实际上可能比B树更差。

通过溢出情况得到的结论是，我们可以说哈希索引最适合于唯一、几乎唯一的数据或每个哈希桶的行数较少的数据。避免问题的一种可能方法是使用部分索引条件从索引中排除高度非唯一的值，但这在许多情况下可能不适用。

与B树一样，哈希索引执行简单的索引元组删除。这是一个延迟维护操作，用于删除已知可以安全删除的索引元组（其项标识符的LP\_DEAD位已设置的那些）。如果insert发现页面上没有可用空间，我们会尝试通过删除死索引元组来避免创建新的溢出页面。如果此时页面已被锁定，则无法删除。VACUUM期间也会删除死索引指针。

如果可以，VACUUM还将尝试将索引元组压缩到尽可能少的溢出页上，以最小化溢出链。如果溢出页变为空，溢出页可以被回收以在其他桶中重用，尽管我们从未将它们返回到操作系统。目前，除了使用REINDEX重建哈希索引外，没有任何收缩哈希索引的方法。也没有减少桶数量的方法。

哈希索引可能会随着索引行数的增加而扩展桶页数。选择哈希键到桶号的映射，以便可以增量扩展索引。当一个新的桶要添加到索引中时，只需要“拆分”一个现有的桶，其中的一些元组将根据更新后的key到桶号的映射转移到新桶。

扩展发生在前端，这可能会增加用户插入的执行时间。因此，哈希索引可能不适用于行数快速增加的表。

## 63.2. 实现

哈希索引中有四种页面：元页面（零页），其中包含静态分配的控制信息；主桶页；溢出页；和位图页，它们跟踪已释放并可重用的溢出页。出于寻址目的，位图页被视为溢出页的子集。

扫描索引和插入元组都需要定位给定元组应该位于的桶。为此，我们需要元页面中的桶计数、highmask和lowmask；然而，出于性能原因，必须为每个此类操作加锁和锁定元页是不可取的。相反，我们在每个后端的relcache条目中保留了元页面的缓存副本。只要目标桶自上次缓存刷新后未被拆分，这将生成正确的桶映射。

主桶页和溢出页是独立分配的，因为任何给定的索引相对于其桶的数量可能需要更多或更少的溢出页。哈希代码使用一组有趣的寻址规则来支持可变数量的溢出页，而不必在创建主桶页后四处移动。

索引表中的每一行由哈希索引中的单个索引元组表示。哈希索引元组存储在桶页中，如果存在，则存储在溢出页中。我们通过将任意一个索引页中的索引项按哈希代码排序来加快搜索速度，从而允许在索引页中使用二进制搜索。但是，请注意，对于一个桶的不同索引页上的哈希代码的相对排序，没有上述排序。

用于扩展哈希索引的桶分割算法过于复杂，不值得在此提及，下面有更详细的描述[src/backend/access/hash/README](http://src/backend/access/hash/README)。拆分算法是崩溃安全的，如果未成功完成，可以重新启动。

# Chapter 64. 数据库物理存储

本章对LightDB数据库使用的物理存储格式进行一个概述。

## 64. 1. 数据库文件布局

本节在文件和目录的层次上描述存储格式。

在传统上，数据库集簇所使用的配置和数据文件都被一起存储在集簇的数据目录里，通常用LTDATA来引用（用的是可以定义它的环境变量的名字）。LTDATA的一个常见位置是/var/lib/lightdb-x/data。由不同数据库实例所管理的多个集簇可以在同一台机器上共存。

LTDATA目录包含几个子目录以及一些控制文件，如Table 64. 所示。除了这些必要的东西之外，集簇的配置文件lightdb.conf、lt\_hba.conf和pg\_ident.conf通常都存储在LTDATA中，不过可以把它放在别的地方。

Table 64. 1. LTTDATA的内容

项	描述
PG_VERSION	一个包含LightDB主版本号的文件
base	包含每个数据库对应的子目录的子目录
current_logfiles	记录当前被日志收集器写入的日志文件的文件
global	包含集簇范围的表的子目录，比如pg_database
lt_commit_ts	包含事务提交时间戳数据的子目录
lt_dynshmem	包含被动态共享内存子系统所使用的文件的子目录
lt_logical	包含用于逻辑复制的状态数据的子目录
lt_multixact	包含多事务（multi-transaction）状态数据的子目录（用于共享的行锁）
lt_notify	包含LISTEN/NOTIFY状态数据的子目录
lt_replslot	包含复制槽数据的子目录
lt_serial	包含已提交的可序列化事务信息的子目录
lt_snapshots	包含导出的快照的子目录
lt_stat	包含用于统计子系统的永久文件的子目录
lt_stat_tmp	包含用于统计信息子系统的临时文件的子目录
lt_subtrans	包含子事务状态数据的子目录
lt_tblspc	包含指向表空间的符号链接的子目录
lt_twophase	包含用于预备事务状态文件的子目录
lt_wal	包含 WAL（预写日志）文件的子目录
lt_xact	包含事务提交状态数据的子目录
lightdb.auto.conf	一个用于存储由ALTER SYSTEM 设置的配置参数的文件
postmaster.opts	一个记录服务器最后一次启动时使用的命令行参数的文件
postmaster.pid	一个锁文件，记录着当前的 postmaster 进程ID（PID）、集簇数据目录路

项	描述
	径、postmaster启动时间戳、端口号、Unix域套接字目录路径（Windows上为空）、第一个可用的listen_address（IP地址或者*，或者为空表示不在TCP上监听）以及共享内存段ID（服务器关闭后该文件不存在）

对于集簇里的每个数据库，在LTDATA/base里都有一个子目录对应，子目录的名字为该数据库在 pg\_database里的 OID。这个子目录是该数据库文件的缺省位置；特别值得一提的是，该数据库的系统目录存储在此。

注意后续章节描述了内置的heap 表访问方法的行为，和内置 索引访问方法。由于LightDB的可扩展特性，其他访问方法的工作方式可能不同。

每个表和索引都存储在独立的文件里。对于普通关系，这些文件以表或索引的filenode号命名，它可以在pg\_class.relfilenode中找到。但是对于临时关系，文件名的形式为tBBB\_FFF，其中BBB是创建该文件的后台的后台ID，FFF是文件节点号。在每种情况下，在主文件（a/k/a 主分支）之外，每个表和索引有一个空闲空间映射（见Section 64.3，它存储关系中可用空闲空间的信息。空闲空间映射存储在一个文件中，该文件以节点号加上后缀\_fsm命名。表还有一个可见性映射，存储在一个后缀为\_vm的分支中，它用于跟踪哪些页面已知含有非死亡元组。可见性映射将在Section 64.4进一步描述。不被日志记录的表和索引还有第三个分支，即初始化分支，它存储在后缀为\_init的分支中（见Section 64.5）。

### Caution

请注意，虽然一个表的文件节点通常和它的 OID 相匹配，但实际上并不必须如此；有些操作，比如 TRUNCATE、REINDEX、CLUSTER以及某些形式的ALTER TABLE，都可以改变文件节点而同时保留 OID。我们不应该假设文件节点和表 OID 相同。此外，对于包含pg\_class本身在内的特定系统目录，其pg\_class.relfilenode包含0。这些目录的实际文件节点号被存储在一个低层数据结构中，并且可以使用pg\_relation\_filenode()函数获取。

在表或者索引超过 1GB 之后，它就被划分成1G大小的段。第一个段的文件名和文件节点相同；随后的段被命名为 filenode.1、filenode.2等等。这样的安排避免了在某些有文件大小限制的平台上的问题（实际上，1GB只是默认的段尺寸。段尺寸可以在编译LightDB时使用配置选项--with-segsize进行调整）。原则上，空闲空间映射和可见性映射分支也可以要求多个段，但实际上这很少发生。

如果一个表的列中可能存储相当大的项，那么该表就会有与之相关联的TOAST表，它用于存储无法保留在在表行中的域值的线外存储。如果表有TOAST表，该表的pg\_class.reltoastrelid链接到它的TOAST表。参阅Section 64.6获取更多信息。

表和索引的内容在Section 64.6中进一步讨论。

表空间的情况更复杂些。每个用户定义的表空间都在LTDATA/lt\_tblspc目录里面有一个符号连接，它指向物理的表空间目录（就是在CREATE TABLESPACE命令里指定的那个目录）。这个符号连接是用表空间的 OID 命名的。在物理表空间目录中有一个名称取决于LightDB服务器版本的子目录，例如PG\_9.0\_201008051（使用该子目录的原因是后续版本的数据库可以使用CREATE TABLESPACE指定相同的目录位置而不会造成冲突）。在这个版本相关的子目录中，为每个在这个表空间里有元素的数据库都有一个子目录，以数据库的OID命名。该目录里的表和索引遵循文件节点命名模式。pg\_default不需要通过lt\_tblspc来访问，而是对应于LTDATA/base。类似地，pg\_global表空间也不通过lt\_tblspc访问，而是对应于LTDATA/global。

pg\_relation\_filepath()函数显示任何关系的完整路径（相对于LTDATA）。它可以作为记住上面这么多规则的替代方法。但是记住该函数只给出关系的主分支的第一个段的名称——你也许需要追加一个段号和/\_fsm、\_vm或者\_init来找到与该关系相关的所有文件。

临时文件（用于如排序不能放在内存中的数据等操作）被创建在LTDATA/base/pgsql\_tmp中，如果临时文件被指定在一个非pg\_default表空间中则它们会被创建在该表空间的pgsql\_tmp子目录中。临时文件的名称的形式为pgsql\_tmpPPP.NNN，其中PPP是其所属后端的PID，而NNN用于区别该后端的不同临时文件。

## 64. 2. TOAST

本节提供TOAST的概述（超尺寸属性存储技术—The Oversized-Attribute Storage Technique）。

LightDB使用固定的页面尺寸（通常是8kB），并且不允许元组跨越多个页面。因此不可能直接存储非常大的域值。为了克服这个限制，大的域值会被压缩并/或分解成多个物理行。这些处理对用户都是透明的，只是在大部分的后端代码上有一些小的影响。这个技术的昵称是TOAST（或者“切片面包之后的最好的东西”）。TOAST 机制也被用来提升内存中大型数据值的处理。

只有特定的数据类型支持TOAST — 我们没必要在那些不可能生成大域值的数据类型上强加这种负担。要支持TOAST，数据类型必须有变长（varlena）的表现形式，通常在存储的值中，头四个字节表示值的总长度（包括长度本身，以字节计）。TOAST并不约束该数据类型的表达的剩余部分。这种特殊的表达被统称为已TOAST值，对它们的操作都必须通过修改或者重新解释这个初始长度字来进行。因此，支持一种可TOAST数据类型的 C 函数必须要小心它们可能会处理被TOAST过的输入值：一个输入值可能并不真正由一个四字节长度和内容构成，直到它被反TOAST（通常是在对一个输入值做任何事情之前，先用PG\_DETOAST\_DATUM；但是在某些情况下也存在更高效的方法，详见Section 36.13.1）。

TOAST占用使用变长类型的长度字的最高两个二进制位（大端法机器上的高位，小端法机器上的低位），这样就任何可TOAST值的逻辑长度限制在1GB（ $2^{30} - 1$ 字节）。如果两个位都是零，那么数值是该数据类型一个普通的未TOAST的值，并且长度字的剩余位给出整个数据以字节计的大小（包括长度字）。当最高位或者最低位被设置时，该值只是有一个单字节头部而不是通常的四字节头部，并且该字节的剩余位数给出了以字节计的总数据尺寸（包括长度字节）。这种节省空间的方案支持对低于127字节的值的存储，不过需要时仍然允许数据类型增长到1GB。带有单字节头部的值不会按照任何特别的边界对齐，反之带有四字节头部的值会按照至少一个四字节边界对齐。这种对齐填充的省略额外地节省了空间，这种节省比起短值来说更加显著。作为一种特殊情况，如果一个单字节头部的剩余位全是零（对于一个自包含的长度来说是不可能的），该值就是一个线外数据的指针，这就可能有下文所述的几种可能的情况。这样一个TOAST指针的类型和尺寸由该数据的第二个字节中存储的一个代码决定。最后，如果最高位或最低位被清除而另一位被设置，则表示该数据的内容被压缩过并且在使用前必须先解压。在这种情况下四字节长度字的剩余位指出了压缩过的数据的大小，而不是原始数据的大小。注意对于线外数据也可能存在压缩，但是变长数据的头部不会告诉我们压缩是否发生 — TOAST指针的内容将说明这个问题。

如前所述，有多种类型的TOAST指针数据。最古老且最常见的类型是指向存储在一个TOAST表中的线外数据的指针，TOAST表与包含该TOAST指针数据本身的表是相关的，但两者又是被分离存储的。当一个要被存储在磁盘上的元组过大时，这些磁盘上的指针数据由TOAST管理代码（在access/common/toast\_internals.c中）所创建。Section 64.2.1中给出了更多的细节。或者，一个TOAST指针数据能够包含一个出现在内存中某处的线外数据的指针。这种数据必定是短命的并且将不会出现在磁盘上，但是它们对于避免大型数据值的复制和冗余处理非常有用。详见Section 64.2.2

无论是内联压缩数据还是离线压缩数据，所使用的压缩技术都属于LZ压缩技术家族中简单且非常快速的一种。详见src/common/pg\_lzcompress.c。

### 64. 2. 1. 线外磁盘上 TOAST 存储

如果一个表中有任何一个列是可以TOAST的，那么该表将有一个与之关联的TOAST表，其OID存储在表的pg\_class.reltoastrelid项中。磁盘上的被TOAST过的值保存在TOAST表里，下文有更详细的描述。



线外值被分裂成（如果压缩过，在压缩之后分裂）最大为TOAST\_MAX\_CHUNK\_SIZE（默认情况下该值应选为使得四个块（chunk）行能放在一个页面中，这个数值大约为2000 字节）字节的块。每个块都作为独立的行存储在从属于所属表的TOAST表中。每个TOAST表都有列chunk\_id（一个表示特定的被TOAST过的数据的OID）、chunk\_seq（一个序列号，存储该块在值中的位置）和一个chunk\_data（该块的实际数据）。在chunk\_id和chunk\_seq上有一个唯一索引，提供对值的快速检索。因此，一个表示线外磁盘上TOAST过的值的指针数据应存储要查看的TOAST表的OID以及指定值的OID（它的chunk\_id）。为了方便，指针数据还存储逻辑数据的尺寸（原始的未压缩的数据长度），物理存储的尺寸（如果应用了压缩，则两者不同）以及采用的压缩方法（如果有的话）。加上变长数据头部的字节，一个磁盘上TOAST指针数据的总尺寸是18字节，不管它代表的值的实际长度是多大。

TOAST管理代码只有在准备向一个表中存储超过TOAST\_TUPLE\_THRESHOLD字节（通常是2kB）的行值的时候才会触发。TOAST代码将压缩和/或线外存储域值，直到行值比TOAST\_TUPLE\_TARGET字节（通常也是2kB）短，或者无法得到更好的结果的时候才停止。在一个 UPDATE 操作过程中，未改变的域的值通常原样保存；所以，如果 UPDATE 一个带有线外值的行时，假如线外值没有变化，那么将不会产生TOAST开销。

TOAST代码识别四种不同的在磁盘上存储可TOAST列的策略：

- PLAIN避免压缩或者线外存储；而且它禁用变长类型的单字节头部。这是不可TOAST数据类型列的唯一可能的策略。只是对那些不能TOAST的数据类型才有可能。
- EXTENDED允许压缩和线外存储。这是大多数可TOAST数据类型的默认策略。首先将尝试进行压缩，如果行仍然太大，那么则进行线外存储。
- EXTERNAL允许线外存储，但是不许压缩。使用EXTERNAL将令那些在宽text和 bytea列上的子串操作更快（代价是增加了存储空间），因此这些操作被优化为只抓取未压缩线外数据中需要的部分。
- MAIN允许压缩，但不允许线外存储（实际上，在这样的列上仍然会进行线外存储，但只是作为没有办法把行变得足以放入一页的情况下的最后手段）。

每个可TOAST的数据类型都为该数据类型的列指定了一个缺省策略，但是一个给定表的列的存储策略可以用ALTER TABLE ... SET STORAGE修改。

可以使用ALTER TABLE ... SET (toast\_tuple\_target = N)为每个表调整TOAST\_TUPLE\_TARGET

这个方法比那些更直接的方法（比如允许行值跨越多个页面）有更多优点。假设查询通常是用相对比较短的键值进行匹配的，那么执行器的大多数工作都将使用主行项完成。TOAST过的属性的大值只是在把结果集发送给客户端的时候才被抽出来（如果它被选中）。因此，主表要小得多，并且它的能放入到共享缓冲区中的行要比没有任何线外存储的方案更多。排序集也缩小了，并且排序将更多地是在内存里完成。一个小测试表明，一个典型的保存 HTML 页面以及它们的 URL 的表占用的存储（包括TOAST表在内）大约只有裸数据的一半，而主表只包含全部数据的 10%（URL和一些小的 HTML 页面）。与在一个非TOAST的对照表里面存储（把全部 HTML 页面裁剪成 7Kb 以匹配页面大小）同样的数据相比，运行时没有任何区别。

## 64.2.2. 线外内存中 TOAST 存储

TOAST指针可以指向不在磁盘上但在当前服务器进程内存中的数据。这样的指针显然不是长期存在的，但是它们是有用的。当前有两种子情况：指向间接数据的指针以及指向扩展数据的指针。

间接TOAST指针指向存储在内存中某个地方的非间接 varlena 值。这种情况仅作为一种概念验证而创建，但是当前它被用来在逻辑解码期间避免创建超过 1GB 的物理元组（把所有线外域值都拉入元组就会这样）。这种情况用处有限，因为该指针数据的创建者需要负责确保只要指针存在，被引用数据就应该存在，并且没有其他设施来帮助它。

扩展的TOAST指针对于复杂数据类型有用，这些数据类型的磁盘上表示形式不是特别适合计算性的目的。例如，一个LightDB 数组的标准 varlena 表达包括了维度信息、一个空值

位图（如果有任何空值元素），然后按顺序是所有元素的值。当元素类型本身是变长时，找到第N个元素的唯一方式是扫描所有在它前面的元素。这种表达适合于磁盘上的存储，因为它很紧凑。但是为了对该数组进行计算，则“扩展”或者“结构”表达会更好，这些表达中所有元素的开始位置都会被标记出来。为了支持这种需要，TOAST指针机制通过允许一个传引用的数据指向一个标准 `varlena` 值（磁盘上的表达）或者一个TOAST指针指向内存中某处的一个扩展表达。这种扩展表达的细节取决于数据类型，不过它必须具有一个标准的头部并且符合 `src/include/utils/expandeddatum.h`中给定的其他API要求。该数据类型的C-级别函数可以选择处理任一表达。不了解扩展表达但简单地在其输入上应用 `PG_DETOAST_DATUM`的函数将自动地接收到传统的 `varlena` 表达。因此对于一种扩展表达的支持可以被增量式地引入，一次一个函数。

扩展值的TOAST指针会被进一步分解成 `read-write`和`read-only`指针。两种方式下被指向的表达是相同的，但是收到一个读写指针的函数被允许就地修改被引用值，而接收到只读指针的函数则不能。如果后者想要做一个该值的被修改的版本，它必须先创建一个副本。这种区分和一些相关的惯例使得可以在查询执行期间避免不必要的扩展值副本。

对于所有类型的内存中TOAST指针，TOAST管理代码会确保这类指针数据不会意外地被存储在磁盘上。在存储之前内存中TOAST指针会被自动地扩展成通常的线内 `varlena` 值，然后可能会被转换成磁盘上的TOAST指针（如果包含的元组不是太大）。

## 64.3. 空闲空间映射

每一个堆和索引关系（除了哈希索引）都有一个空闲空间映射（FSM）来保持对关系中可用空间的跟踪。它伴随着主关系数据被存储在一个独立的关系分支中，以关系的文件节点号加上一个 `_fsm`后缀命名。例如，如果一个关系的文件节点是12345，那么FSM被存储在一个名为 `12345_fsm`的文件中，该文件与主关系文件在同一个目录中。

空闲空间映射被组织成一棵FSM页面的树。底层FSM页面存储了在每一个堆（或索引）页面中可用的空闲空间，对于每一个这样的页面使用一个字节来表示。上层FSM页面则聚集来自于下层页面的信息。

在每一个FSM页面中是一个二叉树，存储在一个数组中，每一个节点一个字节。每个叶节点表示一个堆页面或者一个下层FSM页面。在每一个非叶节点中存储了它孩子节点中的最大值。因此叶节点中的最大值被存储在根中。

关于如何构建、更新和搜索FSM的更多信息请参考 `src/backend/storage/freespace/README`。`lt_freespacemap`模块可以用来检查存储在空闲空间映射中的信息。

## 64.4. 可见性映射

每一个堆关系都有一个可见性映射（VM）用来跟踪哪些页面只包含已知对所有活动事务可见的元组，它也跟踪哪些页面只包含未被冻结的元组。它伴随着主关系数据被存储在一个独立的关系分支中，以该关系的文件节点号加上一个 `_vm`后缀来命名。例如，如果一个关系的文件节点为12345，其VM被存储在名为 `12345_vm`的文件中，该文件域主关系文件在同一个目录中。注意索引没有VM。

可见性映射仅为每个堆页面存储两个位。第一位如果被设置，表示该页面上的元组都是可见的，或者换句话说该页面不含有任何需要被清理的元组。这些信息也可以被 `index-only scans`用来只依靠索引元组回答查询。第二位如果被设置，表示该页面上的元组都已经被冻结。这也意味着防回卷清理操作也不需要重新访问该页面。

该映射是保守的，我们可以确定不论何时一个位被设置，那就说明条件为真，但是如果一个位没有被设置，它可能为真也可能不为真。可见性映射的位只会被清理操作设置，但是可以被任何在页面上进行的数据修改操作清除。

`lt_visibility`模块可以被用来检查存储在可见性映射中的信息。

## 64.5. 初始化分支

每一个不被日志记录的表以及在这类表上的每一个索引，都有一个初始化分支。初始化分支是一个适当类型的空表或空索引。当一个不被日志记录的表由于崩溃必须被重置为空时，初始化分支被随着主分支复制，而任何其他分支则被擦除（它们会在需要时自动被重建）。

## 64.6. 数据库页面布局

本章提供一个LightDB的表和索引所使用的页面格式的概述。<sup>1</sup> 序列和TOAST表的格式就像一个普通表一样。

在下面解释中，一个字节被假定包含 8 个位。另外，术语item指的是存储在一个页面里的独立数据值。在一个表里，一个项是一个行；在一个索引里，一个项是一条索引记录。

每个表和索引都以一个固定尺寸（通常是 8kB，不过在编译服务器时可以选择其他不同的尺寸）的页面数组存储。在表中，所有页面在逻辑上都相同，所以一个特定的项（行）可以被存储在任何页面里。在索引里，第一个页面通常保留为元页来保存控制信息，并且依索引访问方法的不同，在索引里可能有不同类型的页面。

Table 64. 显示一个页面的总体布局。每个页面有五个部分。

Table 64.2. 总体页面布局

项	描述
PageHeaderData	24字节长。包含关于页面的一般信息，包括空闲空间指针。
ItemIdData	指向实际项的项标识符数组。每一个条目是一对(偏移量、长度)。每个项 4 字节。
Free space	未分配的空间（空闲空间）。新项标识符从这个区域的开头开始分配，新项从其结尾开始分配。
Items	实际的项本身。
Special space	索引访问模式相关的数据。不同的索引访问方式存放不同的数据。在普通表中为空。

每个页面的头24个字节组成页头（PageHeaderData）。它的格式在Table 64. 详细介绍。第一个域跟踪与此页面相关的最近的 WAL 项。第二个域包含该页面的校验码（如果data checksums被启用）。接下来一个2字节的域包含标志位。此后跟着三个 2 字节的整数域（pd\_lower、pd\_upper和pd\_special）。这些域包含从页面开始位置到未分配空间开头的字节偏移、到未分配空间结尾的字节偏移以及到特殊空间开头的字节偏移。页面头中再接下来的 2 字节（pd\_pagesize\_version）存储页面尺寸和一个版本指示器。从LightDB 8.3开始的版本号为4；LightDB 8.1和8.2使用版本号3；LightDB 8.0 使用版本号 2；LightDB 7.3 和 7.4 使用版本号 1；更早的版本使用版本号 0（基本页面布局和头格式在大部分这些版本里都没有改变，但是堆的行头部布局有所变化）。页面大小主要用于交叉检查；目前在一次安装里，还不能支持多于一种页面大小。最后的域是一个提示，它显示删除该页是否可能获益：它跟踪在页面上最老的未删除的XMAX。

Table 64.3. PageHeaderData布局

域	类型	长度	描述
pd_lsn	PageXLogRecPtr	8 bytes	LSN: 最后修改这个页面的WAL记录最后一个字节后面的第一个字节

<sup>1</sup> 实际上，表和索引访问模式并不需要使用这种页面格式。heap表访问方法总是采用这种格式。目前所有的索引方法的确也使用这个基本格式，但索引元页里的数据通常并不遵循项布局规则。

域	类型	长度	描述
pd_checksum	uint16	2 bytes	页面校验码
pd_flags	uint16	2 bytes	标志位
pd_lower	LocationIndex	2 bytes	到空闲空间开头的偏移量
pd_upper	LocationIndex	2 bytes	到空闲空间结尾的偏移量
pd_special	LocationIndex	2 bytes	到特殊空间开头的偏移量
pd_pagesize_version	uint16	2 bytes	页面大小和布局版本号信息
pd_prune_xid	TransactionId	4 bytes	页面上最老未删除 XMAX, 如果没有则为0

所有细节都可以在src/include/storage/bufpage.h中找到。

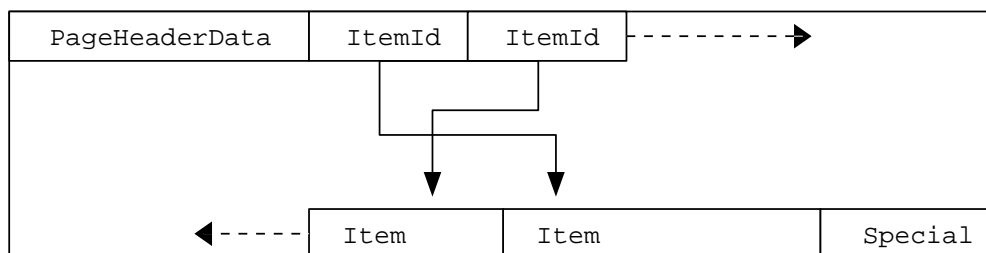
在页头后面是项标识符 (ItemIdData)，每个占用四个字节。一个项标识符包含一个到项开头的字节偏移量（它的长度以字节计），以及一些属性位，这些属性位影响对它的解释。新的项标识符根据需求从未分配空间的开头分配。项标识符的数目可以通过查看pd\_lower来判断，在分配新标识符的时候pd\_lower会增长。因为一个项标识符在被释放前绝对不会移动，所以它的索引可以用于长期地引用一个项，即使该项本身因为压缩空闲空间在页面内部进行了移动。实际上，LightDB创建的每个指向项的指针 (ItemPointer，也叫做CTID) 都由一个页号和一个项标识符的索引组成。

项本身存储在从未分配空间末尾开始从后向前分配的空间里。它们的实际结构取决于表包含的内容。表和序列都使用一种叫做 HeapTupleHeaderData的结构，如下文所述。

最后一部分是“特殊部分”，它可以包含访问方法想存放的任何东西。比如，b-tree 索引用它存储指向页面的左右兄妹的链接，以及其他一些和索引结构相关的数据。普通表并不使用这个部分（通过设置pd\_special等于页面大小来表示）。

Figure 64.1举例说明这些组件如何在一个页面中布局。

Figure 64.1. 页面布局



### 64.6.1. 表行布局

所有表行都用同样方法构造。它们有一个定长的头部（在大多数机器上占据 23 个字节），后面跟着一个可选的空值位图、一个可选的对象 ID 域以及用户数据。该头部在Table 64.1详细描述。实际的用户数据（行的列）从t\_hoff指示的偏移位置开始，它必须总是该平台的 MAXALIGN 距离的倍数。空值位图只有在t\_infomask中的HEAP\_HASNULL位被设置时存在。如果存在，那么它紧跟在定长的头部后面，并占据足够的字节来容纳每个数据列对应的一个位（也就是说，位数等于t\_infomask2中的属性计数。）。在这个位的列表

中，为 1 的位表示非空，而为 0 的位表示空。如果这个位图不存在，那么所有列都被假设为非空的。对象 ID 只有在设置了 `t_infomask` 里面的 `HEAP_HASOID_OLD` 位时才存在。如果存在，它正好出现在 `t_hoff` 边界之前。如果需要对齐 `t_hoff` 使之成为 `MAXALIGN` 的倍数，那么填充将出现在空值位图和对象 ID 之间（这样也保证了对象 ID 得到恰当的对齐）。

Table 64.4. HeapTupleHeaderData 布局

域	类型	长度	描述
<code>t_xmin</code>	TransactionId	4 bytes	插入XID标志
<code>t_xmax</code>	TransactionId	4 bytes	删除XID标志
<code>t_cid</code>	CommandId	4 bytes	插入和/或删除CID标志（覆盖 <code>t_xvac</code> ）
<code>t_xvac</code>	TransactionId	4 bytes	VACUUM操作移动一个行版本的XID
<code>t_ctid</code>	ItemPointerData	6 bytes	当前版本的TID或者指向更新的行版本
<code>t_infomask2</code>	uint16	2 bytes	一些属性，加上多个标志位
<code>t_infomask</code>	uint16	2 bytes	多个标志位
<code>t_hoff</code>	uint8	1 byte	到用户数据的偏移量

所有细节都可以在 `src/include/access/htup_details.h` 中找到。

只有从其他表里获取了信息之后才能对确切数据进行，这些信息大多数  
在 `pg_attribute` 中。标识域位置的关键值是 `attlen` 和 `attalign`。我们没有办法直接获取某个特定属性，除非它们是定宽并且没有空值。所有这些复杂的操作都封装在函数 `heap_getattr`、`fastgetattr` 和 `heap_getsysattr` 中。

要读取数据的话，你需要轮流检查每个属性。首先根据空值位图检查该域是否为 `NULL`。如果是，那么跳到下一个。然后保证你的对齐是正确的。如果域是一个定宽域，那么所有字节都简单地放在其中。如果它是一个变长域（`attlen = -1`），那么它就会有点复杂。所有变长数据类型都使用一个通用的头部结构 `struct varlena`，它包含所存储的值的总长度以及一些标志位。根据标志的不同，数据可能在线内或者是在一个 `TOAST` 中，还可能是压缩的（参阅 [Section 64.2](#)）。

---

## Chapter 65. 系统目录声明和初始内容

LightDB使用很多不同的系统目录来跟踪数据库对象（例如表和函数）的存在以及属性。系统目录和普通用户表之间在物理上没有什么不同，但是后端的C代码知道每一个目录的结构和属性，并且能够在较低的层次上直接操纵它们。因此，不建议尝试在运行中修改目录的结构，那样做会破坏内建在C代码中对目录行如何放置的设想。但是目录的结构可能会在主版本之间发生变化。

目录的结构声明在特殊格式的C头文件中，它们位于源码树的src/include/catalog/目录中。对每一个目录都有一个以其名称命名的头文件（例如，pg\_class的头文件是pg\_class.h），头文件定义了目录具有的列集合，以及一些其他的基本属性（例如OID）。

很多目录都有初始数据，这些数据必须在initdb的“bootstrap”阶段装载到对应的目录中，这样才能让系统达到能够执行SQL命令的状态点（例如，pg\_class.h必须包含表示其自身的一个项，还要为每个系统目录和索引都分别包含一项）。这些初始数据以可编辑的形式保存在src/include/catalog/目录下的数据文件中。例如，pg\_proc.dat描述了所有必须被插入到pg\_proc目录的初始行。

为了创建目录文件并且将这些初始数据载入其中，一个以bootstrap模式运行的后端会读取包含着命令和初始数据的BKI（后端接口）文件。这种模式中用到的lightdb.bki文件正是在编译LightDB时从前述的头文件和数据文件准备而来，这一过程由名为genbki.pl的Perl脚本负责。尽管lightdb.bki与特定LightDB发行版相关，但它是平台无关的并且被安装在安装树的share子目录中。

genbki.pl还会为每个目录产生一个头文件，例如为pg\_class生成的头文件是pg\_class\_d.h。这个文件含有自动生成的宏定义，并且可能包含其他的宏、枚举声明，因此对于读取特定目录的客户端C代码很有用。

大部分LightDB的开发者不需要直接与BKI文件打交道，但是几乎在后端中增加任何非平凡的特性都需要修改目录头文件或者初始数据文件。本章的剩余部分会给出一些相关的信息，并且将会完整地描述BKI文件格式。

### 65.1. 系统目录声明规则

一个目录头文件的关键部分是一个C的结构定义，它描述该目录中每一行的布局。这个结构开始于一个CATALOG宏，它对于C编译器而言只不过是typedef struct FormData\_catalogname的一个简写。该结构中的每一个域会导致出现一个目录列。域可以用genbki.h中描述的BKI属性宏进行标注，例如可以为域定义默认值或者把域标记为可以为空或者不能为空。CATALOG行也可以用genbki.h中描述的一些其他BKI属性宏标注，用于定义该目录整体的其他属性，例如是否为共享的关系。

系统目录缓冲代码（以及大部分目录功能代码）假定所有的系统目录元组的定长部分是实际的存在形式，因为它会把这个C结构声明映射到定长部分之上。因此，所有变长域和可以为空的域必须被放置在最后，并且不能够以结构的域的方式访问。例如，如果尝试设置pg\_type.typrelid为NULL，当某段代码尝试引用typetup->typrelid（或者更糟糕的是引用typetup->typelem，因为它跟随在typrelid之后）时将会出现失败。这会导致随机错误乃至段错误。

作为对这类错误的一种部分保护，变长或可以为空的域不应该对C编译器可见。通过将它们包裹在#ifdef CATALOG\_VARLEN ... #endif（其中CATALOG\_VARLEN是一个永不被定义的符号）中可以实现这一点。这能防止C代码不小心尝试访问可能不在那里或者可能在其他某个偏移位置的域。作为一种防止创建不正确行的措施，我们要求所有应该为非空的列在pg\_attribute中也被标记为非空。如果目录列是定长的并且前面没有任何可以为空或可变宽度的列，bootstrap代码将自动把它标记为NOT NULL。在这一规则不适用的地方，可以根据需要使用BKI\_FORCE\_NOT\_NULL和BKI\_FORCE\_NULL标注强制正确的标记。

前端代码不应该包括任何pg\_XXX.h目录头文件，因为这些文件可能包含在后端之外无法编译的C代码（通常，这是因为这些文件还包含src/backend/catalog/文件中函数的声明）。不过，前端代码可以包括相应的pg\_XXX.d.h头文件，它将包含OID #define以及任何其他可能要在客户端使用的数据。如果希望前端代码能看到目录头文件中的宏或者其他代码，可以在相应部分的周围写上#ifdef EXPOSE\_TO\_CLIENT\_CODE ... #endif，这样会指示genbki.pl把相应的部分拷贝到pg\_XXX.d.h头文件中。

少数目录是非常基础的，以至于它们无法用大部分目录采用的BKI create命令来创建，因为那个命令需要在这些目录中写入信息来描述新的目录。这些目录被称为bootstrap目录，定义一个这样的目录需要一些额外的工作：开发者必须为它在pg\_class和pg\_type的预装载内容中手工准备合适的项，并且后续对该目录结构的更改将会更新那些项（bootstrap目录还需要pg\_attribute中的预装载项，但是幸运地是现今的genbki.pl会处理这些杂务）。如果可能，一定避免将新目录创建为bootstrap目录。

## 65.2. 系统目录初始数据

每个有手工创建的初始数据（有些没有）的目录都有一个相应的.dat文件，其中以可编辑的格式包含着该目录的初始数据。

### 65.2.1. 数据文件格式

每个.dat文件含有Perl数据结构文本，它可以简单地通过eval产生由一个哈希引用数组构成的内存数据结构，每个目录行一个。从pg\_database.dat摘出的经过略微修改的一小部分可以展示关键特性：

```
[
# A comment could appear here.
{ oid => '1', oid_symbol => 'TemplateDbOid',
  descr => 'database\'s default template',
  datname => 'template1', encoding => 'ENCODING', datcollate => 'LC_COLLATE',
  datctype => 'LC_CTYPE', datistemplate => 't', dataallowconn => 't',
  datconnlimit => '-1', datlastsysoid => '0', datfrozenxid => '0',
  datminmxid => '1', dattablespace => 'pg_default', dataacl => '_null_' },
]
```

需要注意的点：

- 总体的文件布局是：开方括号，一个或者多个花括号集合（每一个表示一个目录行），闭方括号。在每一个闭花括号之后写一个逗号。
- 在每个目录行内，写成逗号分隔的key => value对。允许的key是该目录的列名，外加上元数据键oid、oid\_symbol、array\_type\_oid以及descr（oid和oid\_symbol的使用在下文的Section 65.2.2中描述，而array\_type\_oid在Section 65.2.4中描述。descr为该对象提供一个描述字符串，它将被插入到pg\_description或pg\_shdescription中）。虽然元数据键是可选的，但是目录中定义的列必须全部提供，除非目录的.h文件为该列指定了默认值。以上的例子里，忽略了datdba字段是因为pg\_database.h为其提供了适当的默认值。
- 所有的值都必须被放在单引号中。用反斜线可以转义值中用到的单引号。作为数据的反斜线可以（但是不必）被双写，这遵循的是Perl对简单引用文本的规则。注意，作为数据出现的反斜线将被bootstrap扫描器根据转义字符串常量的相同规则（见Section 4.1.2.2当作转义处理。例如\t转换为一个制表符。如果在最终值中确实想要一个反斜线，则需要写成四个：Perl会剥离掉两个，留下\\给bootstrap扫描器。
- 空值被表示为\_null\_（注意没有办法创建就是该字符串的值）。
- 注释以#开头，并且必须位于它们自己的行上。

- 那些是其他目录条目的OID的字段值应该用符号名而不是实际的数字OID来表示（在上述例子中，`dattablespace`包含此类的引用。）这会在下文的Section 65.2.2中描述。
- 因为哈希是无序的数据结构，域顺序和行布局并不重要。不过，为了维持一种一致的外貌，我们设定了一些规则，它们由格式化脚本`reformat_dat_file.pl`实施：
  - 在每一对花括号内，元数据域`oid`、`oid_symbol`、`array_type_oid`和`descr`（如果存在）按照这个顺序放在最前面，然后以定义时的顺序放上该目录自己的域。
  - 如果可能，根据需要在域之间插入新行以限制行的长度低于80字符。在元数据域和普通域之间也插入一个新行。
  - 如果目录的`.h`文件为一个列指定了默认值并且一个数据项具有相同的值，`reformat_dat_file.pl`将从数据文件中省去它。这能使得数据表达紧凑。
  - `reformat_dat_file.pl`原样保留空行和注释行。
- 推荐在提交目录数据补丁前运行`reformat_dat_file.pl`。为了方便起见，可以简单地更改`src/include/catalog/`并且运行`make reformat-dat-files`。
- 如果想要增加一种新方法让数据表达更小，必须在`reformat_dat_file.pl`中实现该方法并且还要教会`Catalog::ParseData()`如何将数据展开回完整的表达。

## 65.2.2. OID分配

通过写一个`oid => nnnn`元数据域，出现在初始数据中的目录行可以被给予一个手工分配的OID。此外，如果分配一个OID，可以通过书写一个`oid_symbol => name`元数据域为该OID创建一个C宏。

如果预装载的目录行被其他预装载行用OID引用，则必须给它们预先分配OID。如果行的OID必须被C代码引用，也需要预分配的OID。如果两种情况都不符合，则`oid`元数据域可以被省略，在这种情况下`bootstrap`代码会自动分配OID。实际上对于一个给定的目录，即便其中某些行实际并没有被交叉引用，我们也通常会为其中预装载的行全部预分配OID或者全部不分配OID。

在C代码中写出任何OID的实际数字值是一种非常糟糕的形式，通常应该使用宏。对`pg_proc` OID的直接引用太常见了，因此有一种特别的机制自动创建必需的宏，见`src/backend/utils/Gen_fmgrtab.pl`。类似地 — 但是由于历史原因，实现的方式不同 — 也有一种自动的为`pg_type` OID创建宏的方法。因此在这两个目录中，`oid_symbol`项不是必需的。同样，系统目录和索引的`pg_class` OID的宏是自动设置的。对于所有其他系统目录，开发者必需通过`oid_symbol`项手动指定所需的宏。

要为一个新的预装载行找到一个可用的OID，可以运行脚本`src/include/catalog/unused_oids`。它能打印出未被使用的OID的闭区间范围（例如，输出行45-900表示OID 45到900都还没有被分配出去）。当前，OID 1-9999被保留给手工分配，`unused_oids`脚本会简单地查看目录头部以及`.dat`文件来看看哪些OID没有出现。也可以使用`duplicate_oids`脚本来检查错误（`genbki.pl`将为没有手工分配给他们的任何行分配OID，还会在编译时检测重复的OID）。

在为一个不会立即提交的补丁选择OID时，最佳实践是使用一组接近连续的OID，以8000—9999范围中随机选择的值作为开始。这样就将其与其他同时开发的补丁的冲突的风险降至最低。为了保持8000—9999范围对于开发来讲可自由使用，在补丁提交到git库之后，应使用低于该范围的可用空间重新编号。通常这项工作会在接近每次开发周期的尾声时做，与此同时移动由该周期提交的补丁所消费掉的所有OID。`renumber_oids.pl`脚本可用来完成该项工作。如果发现一个未提交的补丁与一些近来提交的补丁有OID冲突，`renumber_oids.pl`也可用于从此状况下的恢复。

因为此约定可能会对补丁程序分配的OID重新编号，在补丁纳入到官方发布版本之前，补丁所分配的OID不应该认为是稳定的。一旦发布，我们并不修改手工分配的对象OID，但是那样会产生各种各样的兼容性问题。



如果需要为没有手工分配的OID的目录条目分配OID，genbki.pl将使用10000—11999范围内的值。在自举运行的开始，服务器的OID计数器设置为12000。这样自举的后期由SQL命令创建的对象，例如运行information\_schema.sql脚本时创建的对象，接收的OID为12000或更高。

在正常数据库操作期间分配的OID限制为16384或更高。这样确保对于genbki.pl自动分配或自举过程中的OID可以使用范围10000—16383内的值。这些自动分配的OID不稳定，从一种安装到另一安装可能会改变。

### 65.2.3. OID引用查找

原则上，从一个初始目录行到另一个初始目录行的交叉引用只需要在引用字段中写上被引用行的预分配OID就可以实现。但是这违反了项目策略，因为它容易出错，难理解，并且如果新分配的OID重新编号的话容易损坏。因此genbki.pl提供了替代的方式来编写符号引用。规则如下：

- 通过对特定的目录列定义附加BKI\_LOOKUP(lookuprule)来开启对符号化引用的使用，其中lookuprule是被引用目录的名称，例如pg\_proc。BKI\_LOOKUP可以被附加到类型为oid、regproc、oidvector或者Oid[]的列上，在后两种情况中它意味着在数组的每个元素上执行查找。
- 也允许将BKI\_LOOKUP(encoding)附加到整数列以引用字符集编码，字符集编码目前还没有表示为目录OID，但有对于genbki.pl已知的一组值。
- 在一些目录列，允许条目为零而不是有效的引用。如果这是允许的，写BKI\_LOOKUP\_OPT而不是BKI\_LOOKUP。然后你能对一个条目写0。（如果列为regproc声明，你可以有选择的写-而不是0。）除了这种特殊情况外，BKI\_LOOKUP列中的所有条目都必须是符号引用。genbki.pl将会警告无法识别的名称。
- 大多数种类的目录对象仅通过其名称引用。注意类型的名称必须完全匹配被引用的pg\_type项的typename，不能使用任何别名，例如用integer来替代int4。
- 函数可以用其prname来表示，前提是它在pg\_proc.dat项中是唯一的（这和regproc输入类似）。否则，要将函数写成prname(argtypename, argtypename, ...)，就像regprocedure那样。参数的类型名称必须被拼写准确，和它们在pg\_proc.dat项的proargtypes域中的值一致。不要插入任何空白。
- 操作符的名称由oprname(lefttype, righttype)表示，类型的名称要写得准确，与它们出现在pg\_operator.dat项的oprleft和oprright域中的值一样（对于一元操作符省略的操作数，可以写成0）。
- 操作符类和操作符族的名称仅在一个访问方法中唯一，因此它们用access\_method\_name/object\_name表示。
- 在这些情况中都不能有模式限定，所有在bootstrap期间创建的对象都应该出现在pg\_catalog模式中。
- 除了通用查找机制外，还有一个特殊的惯例，即PGNSP被pg\_catalog模式的OID取代，而PGUID被Bootstrap Superuser角色的OID取代。这些用法有些历史性，但是到目前为止，不需要将它们概括。

genbki.pl在运行时解决所有符号化引用并且把简单的数字OID放到输出的BKI文件中。因此不需要bootstrap后端处理符号化引用。

### 65.2.4. 数组类型的自动创建

大多数标量数据类型应该有一相应的数组类型（即元素类型为标量类型的标准变长数组类型，该类型由标量类型的pg\_type条目的typarray字段引用）。genbki.pl在大多数情况下能够自动为数组类型生成pg\_type条目。

要使用此工具，只需在标量类型的pg\_type条目中写入array\_type\_oid => nnnn元数据字段，指定要用于数组类型的OID。然后可以省略typarray字段，因为它会使用该OID自动填充。

生成的数组类型名是标量类型名称，前面加下划线。数组条目的其他字段由pg\_type.h中的BKI\_ARRAY\_DEFAULT(value)注释填充，如果没有则从标量类型复制。（对于typalign也有一种特殊情况）然后，将两个条目的typelem和typarray字段设置为相互交叉引用。

## 65.2.5. 编辑数据文件的方法

在更新目录数据文件时，对于执行常用任务的简便方法，这里有一些建议。

向一个目录增加一个带有默认值的新列：`. . .` 用BKI\_DEFAULT(value)标注将列增加到头文件中。数据文件的调整仅需要在要求非默认值的现有行中增加该域即可。

为没有默认值的现有列增加默认值：`. . .` 在头文件中增加一个BKI\_DEFAULT标注，然后运行make reformat-dat-files以移除现在变得冗余的域项。

移除一列（不管有默认值还是没有）：`. . .` 从头文件中移除该列，然后运行make reformat-dat-files以移除现在无用的域项。

更改或者移除现有的默认值：`. . .` 不能简单地更改头文件，因为这将会导致当前的数据被不正确地解读。首先运行make expand-dat-files用显式插入所有默认值的形式重写数据文件，然后更改或者移除BKI\_DEFAULT标注，然后运行make reformat-dat-files移除多余的域。

临时批量编辑：`. . .` 可以修改reformat\_dat\_file.pl执行很多种批量更改。寻找其中展示可以插入一次性代码的注释块。在下面的例子中，我们将把pg\_proc中的两个Boolean域联合成一个char域：

1. 在pg\_proc.h中增加一个带有默认值的新列：

```
+ /* see PROKIND_ categories below */
+ char          prokind BKI_DEFAULT(f);
```

2. 基于reformat\_dat\_file.pl创建一个新脚本以插入合适的值：

```
-          # At this point we have the full row in memory as a hash
-          # and can do any operations we want. As written, it only
-          # removes default values, but this script can be adapted to
-          # do one-off bulk-editing.
+          # One-off change to migrate to prokind
+          # Default has already been filled in by now, so change to other
+          # values as appropriate
+          if ($values{proisagg} eq 't')
+          {
+              $values{prokind} = 'a';
+          }
+          elsif ($values{proiswindow} eq 't')
+          {
+              $values{prokind} = 'w';
+          }
```

3. 运行新的脚本：

```
$ cd src/include/catalog
$ perl rewrite_dat_with_prokind.pl pg_proc.dat
```

到这里pg\_proc.dat拥有所有三个列prokind、proisagg以及proiswindow，不过它们将只出现在它们有非默认值的行中。

4. 从pg\_proc.h移除旧的列:

```
- /* is it an aggregate? */
- bool          proisagg BKI_DEFAULT(f);
-
- /* is it a window function? */
- bool          proiswindow BKI_DEFAULT(f);
```

5. 最后，运行make reformat-dat-files从pg\_proc.dat中移除无用的旧项。

用于批量编辑的脚本的更多例子，请参考这个消息[www.hs.net/lightdb/message-id/CAJVSX8gXnPm+Xa=DxR7kFYprcQ1tNcCT5D003ShfnM6jehA@mail.gmail.com](http://www.hs.net/lightdb/message-id/CAJVSX8gXnPm+Xa=DxR7kFYprcQ1tNcCT5D003ShfnM6jehA@mail.gmail.com)的附件convert\_oid2name.pl和remove\_pg\_type\_oid\_symbols.pl。

## 65.3. BKI文件格式

本节描述LightDB后端如何解释BKI文件。结合一份实际的lightdb.bki文件，本节的内容将会更容易理解。

BKI输入由一个命令序列组成。根据命令的语法，命令由一系列记号构成。记号之间通常由空白分隔，但是在没有歧义时也可不用。没有什么特殊的命令分隔符；语法上无法属于前一命令的记号将开始新的一条命令（通常你会把一个新命令放在一个新行上以保持清晰）。记号可以是某些关键字、特殊字符（圆括弧，逗号等）、标识符、数字或者单引号字串。所有东西都是大小写敏感的。

以#开头的行会被忽略。

## 65.4. BKI命令

```
create tablename tableoid [bootstrap] [shared_relation] [rowtype_oid oid] (name1
= type1 [FORCE NOT NULL | FORCE NULL ] [, name2 = type2 [FORCE NOT NULL | FORCE
NULL ], ...])
```

创建一个叫做tablename，OID为tableoid的表，它的列在圆括弧中给出。

bootstrap.c直接支持下列列类型：bool、bytea、char (1 字节)、name、int2、int4、regproc、regclass、regtype、text、oid、tid、xid、cid、int2vector、oid数组)、\_text (数组)、\_oid (数组)、\_char (数组)、\_aclitem (数组)。尽管我们可以创建包含其它类型列的表，但是我们只有在创建完pg\_type并且填充了合适的记录之后才行（这实际上就意味着在自举目录中，只能使用这些列类型，而非自举目录可以使用任意内置类型）。

如果声明了bootstrap，那么该表将只在磁盘上创建；不会向pg\_class、pg\_attribute等表里面输入任何与该表相关的东西。因此这样的表将无法被普通的SQL操作访问，直到那些记录被用硬办法（用insert命令）建立。这个选项用于创建pg\_class等表本身。

如果声明了shared\_relation，那么表就作为共享表创建。除非声明了without\_oids，否则表将会有OID。表的行类型OID（pg\_type的OID）可以有选择性地通过rowtype\_oid子句指定。如果没有指定，会为之自产生一个OID（如果bootstrap被指定，则rowtype\_oid是无效的，但不管怎样它还是被写在了文档中）。

```
open tablename
```

打开名为tablename的表进行数据插入。任何当前打开的表将被关闭。

```
close [tablename]
```

关闭打开着的表。给出的表名用于交叉检查，但并不是必须的。

```
insert ( [oid_value] value1 value2 ... )
```

用value1、value2 等作为列值向打开的表插入一条新记录。

NULL 可以用特殊的关键字 `_null_` 指定。看起来不像标识符或者数字字符串的值必须被加上单引号。（要在一个值中包含单引号，将其写两次。转义字符串风格的反斜杠转义在字符串中也是允许的。）

```
declare [unique] index indexname indexoid on tablename using amname ( opclass1
name1 [, ...] )
```

在名为tablename的表上用amname访问方法创建一个OID为indexoid的名为indexname的索引。索引的域被称为name1、name2等，而使用的操作符类分别是opclass1、opclass2等。该命令将会创建索引文件和适当的系统目录项，但是索引内容不会被此命令初始化。

```
declare toast toasttableoid toastindexoid on tablename
```

为名为tablename的表创建一个TOAST表。该TOAST表将被赋予由toasttableoid表示的OID，且它的索引将被赋予由toastindexoid表示的OID。和declare index一样，索引的填充将被推迟。

```
build indices
```

填充之前声明的索引。

## 65.5. 自举BKI文件的结构

在open命令打开某个表时，它需要系统中已经存在一些表并且其中要具有与被打开表相关的项，在这些先决条件满足之前，open命令不能被使用（这些至少应该存在的表是pg\_class、pg\_attribute、pg\_proc和pg\_type）。为了允许这些表本身被填充，带着bootstrap选项的create将会隐式打开所创建的表用于插入数据。

同样，declare index和declare toast命令也必须在相关系统目录被创建和填充之后才能被使用。

因此，lightdb.bki文件的结构必须是：

1. create bootstrap其中一个关键表
2. insert数据，这些数据至少要能描述这些关键表
3. close
4. 重复创建其他关键表。
5. create（不带bootstrap）一个非关键表
6. open
7. insert需要的数据
8. close
9. 重复创建其他非关键表。
10. 定义索引和TOAST表。

11.build indices

无疑还有其它未被文档记录的顺序依赖关系。

## 65.6. BKI例子

下面的命令序列将创建test\_table表，表的OID为420，它有三列oid、 cola和colb，类型分别为oid、int4和text，然后向该表插入两行：

```
create test_table 420 (oid = oid, cola = int4, colb = text)
open test_table
insert ( 421 1 'value 1' )
insert ( 422 2 _null_ )
close test_table
```

---

## Chapter 66. 规划器如何使用统计信息

本章建立在Section 14.和Section 14.中讨论的材料之上，展示了关于规划器如何使用系统统计信息来估计一个查询的各个部分可能返回的行数。这是规划处理中的一个重要的部分，它提供了代价计算中的很多原始材料。

本章的目的不是详细地解释代码，而是给出一个规划器如何使用统计信息的概述。这样可以降低那些想以后阅读这部份代码的人的学习曲线。

### 66. 1. 行估计例子

下面展示的例子使用LightDB回归测试数据库中的表。输出结果是从版本 8.3 获得。之前（或之后）版本的动作可能会有所变化。同时还要注意的，由于ANALYZE使用随机采样来产生统计信息，在任何新的ANALYZE之后结果将有轻微改变。

让我们从一个很简单的查询开始：

```
EXPLAIN SELECT * FROM tenk1;
```

QUERY PLAN

```
-----  
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)
```

规划器如何判断tenk1的势在Section 14.中介绍，但为了完整还会在这里重复介绍。行数或页数是从pg\_class中查出来的：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

```
relpages | reltuples  
-----+-----  
358 | 10000
```

这些数字是在表上的最后一次VACUUM或ANALYZE以来的当前值。之后，规划器取出该表中实际的当前页数（这个操作的开销很小，不需要扫描全表）。如果与relpages不同，则对reltuples进行相应的缩放以得到一个当前的行数估计。在上面的例子中，relpages的值是最新的，因此行估计与reltuples相同。

换一个在WHERE子句中带有范围条件的例子：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
```

QUERY PLAN

```
-----  
Bitmap Heap Scan on tenk1 (cost=24.06..394.64 rows=1007 width=244)  
Recheck Cond: (unique1 < 1000)  
-> Bitmap Index Scan on tenk1_unique1 (cost=0.00..23.80 rows=1007 width=0)  
Index Cond: (unique1 < 1000)
```

规划器检查WHERE子句条件，并在pg\_operator中查找<操作符的选择度函数。这被保持在oprrest列中，并且在这个例子中的项是scalarltsel。scalarltsel函数从pg\_statistic为unique1检索直方图。对于手工查询来说，查看更简单的pg\_stats视图会更方便：

```
SELECT histogram_bounds FROM pg_stats
```

```
WHERE tablename='tenk1' AND attname='unique1';
```

```
          histogram_bounds
```

```
-----
{0, 993, 1997, 3050, 4040, 5036, 5957, 7057, 8029, 9016, 9995}
```

然后，把直方图里面被“< 1000”占据的部分找出来。这就是选择度。直方图把范围分隔成等频的桶，所以我们要做的只是把我们的值所在的桶找出来，然后计数其中的部分以及所有该值之前的部分。值 1000 很明显在第二个桶（993 - 1943）中。假设每个桶中的值是线性分布，那么就可以计算出选择度：

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/
num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

也就是一整个桶加上第二个桶的线性部分，除以桶数。那么估计的行数现在可以用选择度乘以tenk1的势来计算：

```
rows = rel_cardinality * selectivity
      = 10000 * 0.100697
      = 1007 (rounding off)
```

然后让我们考虑一个在WHERE子句有等于条件的例子：

```
EXPLAIN SELECT * FROM tenk1 WHERE string1 = 'CRAAAA';
```

```
          QUERY PLAN
```

```
-----
Seq Scan on tenk1 (cost=0.00..483.00 rows=30 width=244)
  Filter: (string1 = 'CRAAAA'::name)
```

规划器还是检查WHERE子句条件，并为=查找选择度函数（这次是eqsel）。对于等值估计而言，直方图是没用的；相反，最常见值（MCV）列表可以用来决定选择度。让我们来看一下MCV，以及一些额外的后面用得上的列：

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='string1';
```

```
 null_frac      | 0
n_distinct     | 676
most_common_vals | {EJAAAA, BBAAAA, CRAAAA, FCAAAA, FEAAAA, GSAAAA,
JOAAAA, MCAAAA, NAAAAA, WGAAAA}
most_common_freqs | {0.00333333, 0.003, 0.003, 0.003, 0.003, 0.003,
0.003, 0.003, 0.003, 0.003}
```

因为CRAAAA出现在 MCV 列表中，那么选择度只是最常见频度（MCF）列表中的一个对应项：

```
selectivity = mcf[3]
            = 0.003
```

像之前一样，行数的估计只是和前面一样用tenk1的势乘以选择度：

```
rows = 10000 * 0.003
```

= 30

现在看看同样的查询，但是常量不在MCV列表中：

```
EXPLAIN SELECT * FROM tenk1 WHERE stringul = 'xxx';
```

QUERY PLAN

```
Seq Scan on tenk1 (cost=0.00..483.00 rows=15 width=244)
  Filter: (stringul = 'xxx'::name)
```

这是完全不同的一个问题：当值不在MCV列表中时，如何估计选择度。解决方法是利用该值不在列表中的事实，结合所有MCV出现的频率的知识：

$$\begin{aligned} \text{selectivity} &= (1 - \text{sum}(\text{mvf})) / (\text{num\_distinct} - \text{num\_mcv}) \\ &= (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 + \\ &\quad 0.003 + 0.003 + 0.003 + 0.003)) / (676 - 10) \\ &= 0.0014559 \end{aligned}$$

也就是，把所有MCV的频度加起来并从 1 减去，然后除以其他可区分值的个数。这相当于假设不是 MCV 中的列的部分均匀分布在所有其他可区分值上。需要注意的是，这里没有空值，因此我们不需要担心这些（否则需要从分子中减去空值的部分）。估计的行数然后按照常规计算：

$$\begin{aligned} \text{rows} &= 10000 * 0.0014559 \\ &= 15 \text{ (rounding off)} \end{aligned}$$

之前带有unique1 < 1000的例子是scalarltset实际工作的过度简化。现在我们已经看过了使用 MCV 的例子，可以增加一些具体细节了。这个例子到目前为止是正确的，因为unique1是一个唯一列，它没有 MCV（显然，没有一个值能比其他值更通用）。对一个非唯一列而言，通常会有直方图和 MCV 列表，并且直方图不包括由 MCV 表示的那部分列。之所以这样做是因为可以得到更精确的估计。在这种情况下，scalarltset直接应用条件（如“< 1000”）到 MCV 列表中的每个值，并且把那些条件判断为真的 MCV 的频度加起来。这对表中是 MCV 的那一部分给出了准确的选择度估计。然后以上述同样的方式使用直方图估计表中不是 MCV 的那部分的选择度，并且组合这两个数字来估计总的选择度。例如，考虑：

```
EXPLAIN SELECT * FROM tenk1 WHERE stringul < 'IAAAAA';
```

QUERY PLAN

```
Seq Scan on tenk1 (cost=0.00..483.00 rows=3077 width=244)
  Filter: (stringul < 'IAAAAA'::name)
```

我们已看到stringul的 MCV 信息，这里是它的直方图：

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringul';
```

histogram\_bounds

```
{AAAAAA, CQAAAA, FRAAAA, IBAAAA, KRAAAA, NFAAAA, PSAAAA, SGAAAA, VAAAAA, XLAAAA, ZZAAAA}
```

检查 MCV 列表，我们发现前 6 项满足条件stringul < 'IAAAAA'，而最后 4 项不满足，所以 MCV 部分的选择度是：



```
selectivity = sum(relevant mvfs)
             = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
             = 0.01833333
```

累加所有的 MCF 也告诉我们由 MCV 表示的群体中的比例是 0.03033333，并且因此由直方图表示的比例是 0.96966667（同样，没有空值，否则我们在这里必须排除它们）。我们可以看到值IAAAAA差不多落在第三个直方图桶的结尾。通过使用一些关于不同字符频率的相当漂亮的假设，规划器对小于IAAAAA的直方图群体部分得到估计值 0.298387。我们然后组合 MCV 和非 MCV 群体的估计：

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
             = 0.01833333 + 0.298387 * 0.96966667
             = 0.307669
```

```
rows       = 10000 * 0.307669
             = 3077 (rounding off)
```

在这个特别的例子中，来自 MCV 列表的纠正相当小，因为列分布实际上很平坦（统计显示这些特殊值比其它值更常见的原因大部分是由于抽样误差）。在更典型的情况下某些值显著地比其它的更常见，这种复杂的处理过程有助于提高准确度，因为那些最常见值的选择度可以被准确地找到。

现在考虑一个WHERE子句中带有多个条件的情况：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringul = 'xxx';
```

QUERY PLAN

```
-----
Bitmap Heap Scan on tenk1 (cost=23.80..396.91 rows=1 width=244)
  Recheck Cond: (unique1 < 1000)
  Filter: (stringul = 'xxx'::name)
  -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..23.80 rows=1007 width=0)
      Index Cond: (unique1 < 1000)
```

规划器假定这两个条件是独立的，因此子句各自的选择度可以被乘在一起：

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringul = 'xxx')
             = 0.100697 * 0.0014559
             = 0.0001466
```

```
rows       = 10000 * 0.0001466
             = 1 (rounding off)
```

需要注意的是，从位图索引扫描中返回的估计行数只反映和索引一起使用的条件；这一点很重要，因为它会影响后续取堆元组的代价估计。

最后我们将检查一个涉及连接的查询：

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```

Nested Loop (cost=4.64..456.23 rows=50 width=488)
  -> Bitmap Heap Scan on tenk1 t1 (cost=4.64..142.17 rows=50 width=244)
        Recheck Cond: (unique1 < 50)
        -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.63 rows=50
width=0)
                Index Cond: (unique1 < 50)
        -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.00..6.27 rows=1
width=244)
                Index Cond: (unique2 = t1.unique2)
    
```

在tenk1上的限制unique1 < 50在嵌套循环连接之前被计算。它的处理类似之前的那个范围查询例子。但是这次值 50 落在unique1直方图的第一个桶内：

```

selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/
num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows       = 10000 * 0.005035
            = 50 (rounding off)
    
```

连接的限制是t2.unique2 = t1.unique2。操作符是我们熟悉的=，然而选择度函数是从pg\_operator的oprjoin列获得的，并且是eqjoinsel。eqjoinsel为tenk2和tenk1查找统计信息：

```

SELECT tablename, null_frac, n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';
    
```

tablename	null_frac	n_distinct	most_common_vals
tenk1	0	-1	
tenk2	0	-1	

在这种情况下，没有unique2的MCV信息，因为所有值看上去都是唯一的，因此我们可以为关系和它们的空值部分使用一个只依赖可区分值数目的算法：

```

selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/
num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
    
```

也就是说，从 1 中减去每个表的空值部分，并且除以可区分值的最大数目。连接可能发出的行数的计算是：嵌套循环里的两个输入值的笛卡尔积的势乘以选择度：

```

rows = (outer_cardinality * inner_cardinality) * selectivity
      = (50 * 10000) * 0.0001
      = 50
    
```

这里有两列的 MCV 列表，eqjoinsel将使用 MCV 列表的直接比较来决定在由 MCV 表示的列群体部分中的连接选择度。群体剩下部分的估计遵循这里展示的不同方法。

需要注意的是，我们把inner\_cardinality显示为 10000，也就是未修改的tenk2尺寸。它可能出现在EXPLAIN输出检查，连接行的估计来自 50 \* 1，即由 outer 行数乘以由tenk2上每个 inner 索引扫描的估计行数。但是这并非那种情况：连接关系尺寸的估计在任何特定的连接计划被考虑之前进行。如果一切顺利，那么两种方式估计的连接尺寸将产生 大概同样的答案，但是由于舍入误差和其它因素它们有时差别显著。

如果对更进一步的细节感兴趣，一个表的尺寸（在任何WHERE子句之前）的估计在src/backend/optimizer/util/plancat.c中完成。子句选择度的一般逻辑在src/backend/optimizer/path/clausesel.c中。操作符相关的选择度函数大部分可以在src/backend/Utils/adt/selfuncs.c中找到。

## 66. 2. 多变量统计例子

### 66. 2. 1. 函数依赖

多元相关性可以用一个非常简单的数据集来演示 — 一个有两列的表，它们都包含相同的值：

```
CREATE TABLE t (a INT, b INT);
INSERT INTO t SELECT i % 100, i % 100 FROM generate_series(1, 10000) s(i);
ANALYZE t;
```

如Section 14.所述，规划人员可以使用从 pg\_class获取的页面和行数来确定 t的基数：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 't';
```

relpages	reltuples
45	10000

他的数据分布非常简单；每列中只有100个不同的值，均匀分布。

以下示例显示了在a列上估计WHERE条件的结果：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1;
QUERY PLAN
```

```
Seq Scan on t (cost=0.00..170.00 rows=100 width=8) (actual rows=100 loops=1)
  Filter: (a = 1)
  Rows Removed by Filter: 9900
```

规划器检查条件并确定该子句的选择性为1%。通过比较这个估计值和实际的行数，我们可以看到估计值非常准确（事实上，是因为表格非常小）。更改WHERE条件以使用b列，将生成一个完全相同的计划。但是观察一下，如果我们在两列上应用相同的条件，将它们用AND结合起来会发生什么：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
QUERY PLAN
```

```
Seq Scan on t (cost=0.00..195.00 rows=1 width=8) (actual rows=100 loops=1)
  Filter: ((a = 1) AND (b = 1))
  Rows Removed by Filter: 9900
```

规划器分别估算每个条件的选择性，达到与上述相同的1%估计值。然后它假定条件是独立的，因此它乘以它们的选择性，产生最终选择性估计值仅为0.01%。这是一个明显的低估，因为符合条件（100）的实际行数要高于两个数量级。

通过创建一个指示ANALYZE 计算两列上的函数依赖性多变量统计信息的统计对象，可以解决此问题。

```
CREATE STATISTICS stts (dependencies) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
          QUERY PLAN
```

```
-----
Seq Scan on t (cost=0.00..195.00 rows=100 width=8) (actual rows=100 loops=1)
  Filter: ((a = 1) AND (b = 1))
  Rows Removed by Filter: 9900
```

## 66.2.2. N 个不同变量的计数

估计多列集合的基数时会出现类似的问题，例如由GROUP BY 子句生成的组的数量。当GROUP BY列出单个列时，n个不同估计值（作为HashAggregate节点返回的估计行数可见）非常准确：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a;
          QUERY PLAN
```

```
-----
HashAggregate (cost=195.00..196.00 rows=100 width=12) (actual rows=100
loops=1)
  Group Key: a
  -> Seq Scan on t (cost=0.00..145.00 rows=10000 width=4) (actual rows=10000
loops=1)
```

但是，如果没有多变量统计信息，那么在GROUP BY 中包含两列的查询中的组数量估计值将在下面的示例中偏离一个数量级：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
          QUERY PLAN
```

```
-----
HashAggregate (cost=220.00..230.00 rows=1000 width=16) (actual rows=100
loops=1)
  Group Key: a, b
  -> Seq Scan on t (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000
loops=1)
```

通过重新定义统计对象以包括两列的n个不同值的计数，估计得到了很大改进：

```
DROP STATISTICS stts;
CREATE STATISTICS stts (dependencies, ndistinct) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
          QUERY PLAN
```

```
-----
HashAggregate (cost=220.00..221.00 rows=100 width=16) (actual rows=100
loops=1)
  Group Key: a, b
  -> Seq Scan on t (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000
loops=1)
```

### 66.2.3. MCV 列表

如 Section 66.2. 中所述，函数依赖是非常廉价和高效的统计类型，但它们的主要限制是其全局特性（仅跟踪列级别的依赖项，而不是在单个列值之间）。

本节介绍 MCV（最常见值）列表的多变量变体，Section 66.1 中描述的每列统计数据的简单扩展。这些统计数据通过存储单独的值来解决这个限制，但是就构建 ANALYZE 中的统计数据、存储和规划时间而言，它的成本自然更高。

让我们再来看看来自 Section 66.2. 的查询，但这次在相同列集上创建了 MCV 列表（请确保删除函数依赖，以确保规划器使用新创建的统计数据）。

```
DROP STATISTICS stts;
CREATE STATISTICS stts2 (mcv) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
          QUERY PLAN
-----
Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual rows=100 loops=1)
  Filter: ((a = 1) AND (b = 1))
  Rows Removed by Filter: 9900
```

The estimate is as accurate as with the functional dependencies, mostly thanks to the table being fairly small and having a simple distribution with a low number of distinct values. Before looking at the second query, which was not handled by functional dependencies particularly well, let's inspect the MCV list a bit. 估计值与函数依赖一样准确，这主要是由于表相当小而且具有少量不同值的简单分布。在查看第二个查询之前，这个函数依赖处理得不是很好，让我们先检查一下 MCV 列表。

可以使用 `pg_mcv_list_items` 集返回函数检查 MCV 列表。

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
          pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts2';
 index | values | nulls | frequency | base_frequency
-----+-----+-----+-----+-----
      0 | {0, 0} | {f, f} |      0.01 |      0.0001
      1 | {1, 1} | {f, f} |      0.01 |      0.0001
      ...
     49 | {49, 49} | {f, f} |      0.01 |      0.0001
     50 | {50, 50} | {f, f} |      0.01 |      0.0001
      ...
     97 | {97, 97} | {f, f} |      0.01 |      0.0001
     98 | {98, 98} | {f, f} |      0.01 |      0.0001
     99 | {99, 99} | {f, f} |      0.01 |      0.0001
(100 rows)
```

这确认了在这两列中有 100 种不同的组合，并且它们的概率都差不多（每个的频率为 1%）。基本频率是从每列统计数据中计算的频率，好像没有多列统计数据。如果任一列中有任何空值，将在 `nulls` 列中标识。

在估计选择性时，规划器对 MCV 列表中的项目应用所有条件，然后对匹配项的频率求和。详情请参阅 `src/backend/statistics/mcv.c` 中的 `mcv_clauselist_selectivity`。

与函数依赖相比，MCV 列表有两大主要优点。首先，列表存储实际值，从而可以决定哪些组合是兼容的。

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 10;
QUERY PLAN
```

```
Seq Scan on t (cost=0.00..195.00 rows=1 width=8) (actual rows=0 loops=1)
  Filter: ((a = 1) AND (b = 10))
  Rows Removed by Filter: 10000
```

第二，MCV 列表处理更广泛的子句类型，而不仅仅是类似函数依赖的相等子句。例如，请考虑对同一表的以下范围查询：

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a <= 49 AND b > 49;
QUERY PLAN
```

```
Seq Scan on t (cost=0.00..195.00 rows=1 width=8) (actual rows=0 loops=1)
  Filter: ((a <= 49) AND (b > 49))
  Rows Removed by Filter: 10000
```

## 66.3. 规划器统计和安全

对表`pg_statistic`的访问仅限于超级用户，以便普通用户无法从中了解其他用户的表的内容。一些选择性估算函数将使用用户提供的操作符（出现在查询中的操作符或相关操作符）来分析所存储的统计。例如，为了确定存储的最常用值是否适用，选择性估计器将必须运行适当的`=`运算符来将查询中的常量与存储的值进行比较。因此，`pg_statistic`中的数据可能会传递给用户定义的运算符。适当制作的操作符可以故意泄漏传递的操作数（例如，通过记录它们或将它们写入不同的表），或者通过在错误消息中显示它们的值来意外泄漏它们，无论哪种情况都可能暴露`pg_statistic`中的数据给一个不应该能够看到它的用户。

为了防止这种情况，以下内容适用于所有内置的选择性估计函数。在规划查询时，为了能够使用存储的统计信息，当前用户必须在表或相关列上具有`SELECT`权限，或者使用的操作符必须是`LEAKPROOF`（更准确地说，是操作符所基于的函数）。如果不是，那么选择性估计器将表现得好像没有统计数据可用，并且规划器将继续进行默认或回退假设。

如果用户对表或列没有所需的权限，那么在很多情况下，查询最终会收到权限被拒绝的错误，在这种情况下，这种机制在实践中是不可见的。但是，如果用户正在从安全屏障视图读取数据，则规划器可能希望检查用户无法访问的基础表的统计数据。在这种情况下，操作符应该是防漏的，否则统计数据将不会被使用。没有直接的反馈意见，除非计划可能是次优的。如果有人怀疑是这种情况，可以尝试将查询作为更有特权的用户来运行，以查看是否产生了不同的计划。

此限制仅适用于规划器需要对`pg_statistic`中的一个或多个值执行用户定义的运算符的情况。因此，无论访问权限如何，规划器都可以使用通用的统计信息，例如列中空值的比例或列中不同值的数量。

可能对用户定义的操作符进行统计操作的第三方扩展中包含的选择性估计函数，应遵循相同的安全规则。请参考LightDB源代码获取指导。

---

## Chapter 67. 备份清单格式

由`lt_basebackup`生成的备份清单主要用于允许使用`lt_verifybackup`验证备份。但是，其他工具也可以读取备份清单文件并使用其中包含的信息用于其自己的用途。为此，本章介绍备份清单文件的格式。

备份清单是编码为 UTF-8 的 JSON 文档。（虽然通常情况下 JSON 文档需要 Unicode，LightDB 允许 `json` 和 `jsonb` 数据类型可使用任何受支持的服务器编码。备份清单没有类似的例外。）JSON 文档始终是一个对象；此对象中存在的键将放在下一节中进行介绍。

### 67.1. 备份清单顶级 (Top-level) 对象

备份清单 JSON 文档包含以下键。

`LightDB-Backup-Manifest-Version`

关联的值始终为整数 1。

`Files`

关联的值始终是对象列表，每个描述备份中存在的一个文件。此列表中不存在使用备份所需要的 WAL 文件的条目或备份清单本身的条目。列表中每个对象的结构在 Section 67.2 中描述。

`WAL-Ranges`

关联的值始终是对象的列表，每个对象描述一个 WAL 记录范，并且必须从特定时间线读取，才能使用备份。这些对象的结构在 Section 67.2 中进一步描述。

`Manifest-Checksum`

此键始终位于于备份清单文件的最后一行。关联的值是前面所有行的 SHA256 校验和。我们在此处使用固定校验和的方法使客户端能够对清单进行增量分析。虽然 SHA256 校验和比 CRC32C 校验和代价大得多，但清单通常应该足够小，因此额外的计算的影响并不很大。

### 67.2. Backup Manifest File Object

描述单个文件的对象包含 `Path` 键或一个 `Encoded-Path` 键。通常，`Path` 键将会出现。关联的字符串值是文件相对于备份目录根目录的路径。位于用户定义的表空间中的文件将具有其前两个组件为 `lt_tblspc` 和表空间的 OID 的路径。如果路径不是 UTF-8 中合法的字符串，或者用户需要编码路径用于所有文件，则将替换为 `Encoded-Path` 键。这样会存储相同的数据，但它被编码为十六进制数字的字符串。字符串中的每对十六进制数字表示一个八位数。

下面两个键始终存在：

`Size`

此文件的预期大小，为整数。

`Last-Modified`

服务器在备份时报告的文件的最后修改时间。与备份中存储的其他字段不同，此字段不用于 `lt_verifybackup`。它在此只是为了提供参考的作用。

如果备份是在启用文件校验和的情况下进行，将存在以下键：

#### Checksum-Algorithm

用于计算此文件校验和的校验和算法。目前，备份清单中每个文件的算法都是相同的，但在将来的版本中可能会改变。目前，支持的校验和算法包括CRC32C、SHA224、SHA256、SHA384、和SHA512。

#### Checksum

为此文件计算的校验和，存储为一系列十六进制字符，每两个字符对应校验和的一个字节。

## 67.3. Backup Manifest WAL Range Object

描述 WAL 范围的对象始终有三个键：

#### Timeline

此 WAL 记录范围的时间线，为整数。

#### Start-LSN

此LSN位于必须在指示的时间线上开始的回放，以便使用此备份。此LSN 通常使用LightDB的格式存储；也就是说，它是一个字符串，由两个十六进制字符字符串组成，每个字符串的长度介于 1 和 8 之间，用斜杠分隔。

#### End-LSN

当使用此备份时，在指示的时间线上回放的最早 LSN 可能会结束。其存储的格式与Start-LSN相同。

一般情况下，只有一个 WAL 范围。但是，如果备份来自备用的(standby)备份，在备份期间由于上游升级而切换时间线，则可能存在多个范围，每个具有不同的时间线。对于同一时间线，永远不会有多个 WAL 范围出现。



---

## Part VIII. Oracle/MySQL兼容

本章节包含了LightDB数据库对于已兼容支持的Oracle和MySQL的功能特性清单列表。

---

## Table of Contents

A. Oracle兼容特性 .....	2343
B. MySQL兼容特性 .....	2351
B. 1. MySQL兼容特性列表 .....	2351
B. 1. 1. 数据类型 .....	2351
B. 1. 2. 函数类型 .....	2352
B. 1. 3. SQL语法 .....	2368

# Appendix A. Oracle兼容特性

本章节主要梳理了Oracle数据库的主要关系型特性，验证LightDB Oracle模式数据库的SQL特性兼容实现情况，包括语法和语义差异，LightDB Oracle模式与Oracle数据库基础功能兼容情况。

Table A.1. 对象类型

对象类型	概述
表(Table)	支持普通表、临时表、分区表
索引(Index)	可支持普通索引、唯一索引，不支持位图索引
约束	支持主键、唯一、外键、检查、非空约束
分区	支持 Range、List、Hash、Composite、Interval 分区
序列(Sequence)	完全支持Oracle序列
类型(Type)	支持复杂类型, 如object type, varray, nested table等
包(Package)	不支持。
存储过程	支持。
函数(Function)	不支持函数体内包含事务语句。
视图(View)	完全支持Oracle view。
同义词	不支持(计划24年支持)。
触发器(Trigger)	不支持Oracle的触发器。
数据库链接(dblink)	支持Oracle的@字样dblink查询(计划24年支持dblink的删除等完善功能)。

详情请参考数据定义.

Table A.2. 数据类型

数据类型	概述
定长字符串	CHAR(n) n以字符数为单位
可变长度字符串	VARCHAR2(n) n以字符数为单位
国家字符集定长字符串	NCHAR(n) n以字符数为单位
国家字符集可变长字符串	NVARCHAR2(n) 上限约为1GB
小数	NUMBER[ (p [, s]) ] p范围区间为[1, 38], 默认值为38
浮点数	FLOAT(p) 最大二进制精度 53
整数	INTEGER 二进制精度38位
单精度浮点数	Oracle为BINARY_FLOAT存储32 位单精度 浮点数数据类型，空间存储占用4个字节； LightDB不支持，替换为FLOAT
双精度浮点数	Oracle为BINARY_DOUBLE存储64 位单精度 浮点数数据类型，空间存储占用4个字节； LightDB不支持，替换为DOUBLE
日期和时间	DATE 时间间隔为秒

数据类型	概述
不带时区的时间戳，包含纳秒级精度	TIMESTAMP WITHOUT TIME ZONE
带时区的时间戳	TIMESTAMP[(fractional_seconds_precision)] WITH TIME ZONE
带本地时区的日期和时间	TIMESTAMP WITH TIME ZONE
年到月的间隔	INTERVAL 'integer [- integer]' {YEAR   MONTH} [(precision)][TO {YEAR   MONTH}]
日、时、分、秒的间隔	INTERVAL DAY[(days_precision)] TO SECOND[(seconds_precision)]
二进制大对象	LightDB支持BLOB，实现为BYTEA的别名
字符大对象	LightDB支持CLOB，实现为TEXT的别名
Unicode字符大对象	Oracle使用NCLOB最大支持4GB，LightDB使用text替代
存储原始二进制数据	RAW(L) 类似于CHAR，声明方式RAW(L)，L为长度，以字节为单位，当RAW类型用作数据库列时，最大长度是2000字节；历史遗留，不推荐使用
存储较大的可变长度字符串（最大2GB）	LONG可变长字符列，最大长度限制是2GB，用于不需要作字符串搜索的长串数据，如果需要进行字符串搜索就要用varchar2类型。推荐CLOB或VARCHAR2；LightDB不支持
存储较大的二进制数据（最大2GB）	LONG RAW可变长二进制数据，最大长度是2GB。历史遗留；LightDB不支持
行标识符ROWID	ROWID可以分为物理 rowid 和逻辑 rowid 两种。普通的堆表中的 rowid 是物理 rowid
空值和null	NULL必须用IS NULL/IS NOT NULL判断，LightDB与Oracle一致

Table A.3. 函数

函数	概述
求数值 n 的绝对值。	ABS(n)
位与，求两个数值型数值按位进行 AND 运算的结果	BITAND (expr1, expr2)
向上取整, 参数可以是数值、日期时间、时间间隔	CEIL(n)
返回e的n次方	EXP(n)
向下取整, 参数可以是数值、日期时间、时间间隔	FLOOR(n)
定长字符串	CHAR(n) n以字符数为单位
返回X的自然对数。	LN(n)
单个参数版本将返回n1的自然对数, 两个参数, 它返回n1的对数的任意基数n2。	LOG(n1[, n2])
求余数。	MOD (X, Y)
求X的Y次方, 该函数是 POW() 的别名。	POWER(X, Y)
对所传参数进行四舍五入, 将一个数值四舍五入到指定的小数位数。	ROUND (X[, D])

函数	概述
返回参数的符号, 如果 n 为正数, 返回 1, 如果 n 为负数, 返回-1, 如果 n 为 0, 返回 0。	SIGN(n)
求二次方根, 返回非负数字X的平方根。	SQRT(X)
用于返回X的截断到小数位D号的值。 如果D为0, 则小数点被除去。	TRUNC (X[, D])
将一个ASCII码转换为一个字符。	CHR(n) 此函数将NUMBER值或任何可以隐式转换为NUMBER的值作为参数, 并返回一个字符。
合并字符串函数, 返回结果为连接参数产生的字符串, 参数可以是一个或多个。	CONCAT (char1, char2[, ...]) 。
返回首字母大写, 可将字符串中由空格或非字母数字字符分隔的每个单词的首字母大写。	INITCAP(char)
将字符串中的字母转换为小写。	LOWER(char)
在输入字符串的左边填充上 char2 指定的字符, 将其拉伸至 n 个字节长度。	LPAD(char1, n[, char2])
删除字符串左侧的指定字符或空格。	LTRIM(char1 [, char2]) 第二个参数不传表示删除左侧的空格。
通过正则匹配字符来替换给定的字符串。	REGEXP_REPLACE(str, pattern [, replace_str [, position [, occurrence [, match_param]]]])
通过正则匹配返回子字符串。	REGEXP_SUBSTR(str, pattern [, position [, occurrence [, match_param[, subexpr]]]])
字符串替换函数, 返回替换后的新字符串。	REPLACE(str, search_string [, replace_string])
在输入字符串的右边填充上 char2 指定的字符, 将其拉伸至 n 个字节长度。	RPAD(char1, n[, char2])
删除字符串右侧的指定字符或空格。	RTRIM(char1 [, char2]) 第二个参数不传表示删除左侧的空格。
截取字符串。	SUBSTR(char, position, substring_length) 。
删除字符串左右两侧的指定字符或空格。	TRIM(str)
将字符串中的字母转换为大写。	UPPER(char)
将字符串中的字母转换为大写。	UPPER(char)
返回某字符的 ascii 数值	ASCII(char)
计算字符串长度	LENGTH(char)
正则匹配子串, 返回首次出现的位置	REGEXP_INSTR(str, pattern[, position[, occurrence [, return_opt [, match_param [, subexpr]]]])
将给定日期date增加指定的月份数n, 如果 date 是该月的最后一天, 或者结果月份的天数少于 date 的天数, 则结果为结果月份的最后一天。否则, 结果与 date 具有相同的日期	ADD_MONTHS(date, n)
获取当前 session 时区的DATE 类型日期, 以会话时区为准, 返回值为 DATE 类型, 使用公历历	CURRENT_DATE()

函数	概述
获取当前 session 时区的 TIMESTAMP WITH TIME ZONE 类型日期	CURRENT_TIMESTAMP[(p)], p默认为 6
提取一个日期类型中的某个域, 如年、月、日等。如果指定提取 YEAR 或 MONTH, 则 datetime 数据类型为日期时间类型	EXTRACT ({ YEAR MONTH DAY HOUR MINUTE SECOND TIMEZONE_HOUR TIMEZONE_MINUTE TIMEZONE_REGION TIMEZONE_ABBR} FROM datetime)
返回某日期类型对应月的最后一天的 DATE 类型日期	LAST_DAY(date)
计算两日期之间的月数	MONTHS_BETWEEN(date1, date2)
将日期舍入到最近的一天, 返回 DATE 类型	ROUND (date, [fmt]) 如果不设置fmt, 则返回最近的一天
获取系统日期时间, 返回数据库所在服务器的操作系统时间, 返回格式受 NLS_DATE_FORMAT的影响	SYSDATE
将字符串转换为 TIMESTAMP 类型	TO_TIMESTAMP(char[, fmt])
将日期截断到最近的一天, 返回 DATE 类型	TRUNC (date[, fmt]); date: DATE 数据类型; fmt: 指定了函数返回值与 date 的匹配单位
返回若干表达式中的最大值	GREATEST(expr [, expr ]...)
返回若干表达式中的最小值	LEAST(expr [, expr ]...)

Table A.4. SQL语法

SQL语法	概述
创建表	建表时可声明唯一约束、主键约束、外键约束、创建范围约束
复制表结构	CREATE TABLE tableName AS select * from old_table_name
修改表名	ALTER TABLE table_name RENAME TO new_table_name 标准的重命名表名语法
修改表属性	ALTER TABLE table_name MODIFY col_name NUMBER(18, 0); 修改表列的属性
清空表, 不删除表本身	TRUNCATE TABLE table_name;
对数据库表和列增加注释	comment on [table column] object_name is string1
新增列	ALTER TABLE ADD column_name column_definition 对表新增列
新增表约束	可支持主键、唯一、范围等约束
删除表约束	ALTER TABLE table_name DROP CONSTRAINT IDX 删除表约束
新增索引	用可创建普通, 唯一 索引等, 暂不支持位图索引
删除索引	DROP INDEX index_name
创建临时表	支持全局临时表和会话私有临时表
创建分区表	支持范围分区、列表、哈希分区、自动分区、间隔分区以及组合分区
清空分区	ALTER TABLE table_name TRUNCATE PARTITION partition_names 清空分区数据

SQL语法	概述
合并分区	常用可支持merge、coalesce两种合并分区的操作。MERGE操作可以用于范围分区、列表分区和哈希分区，需要手动选择要合并的分区。COALESCE操作仅适用于哈希分区，自动选择要合并的分区，主要关注空闲扩展的合并。在实际应用中，根据需求和分区类型选择合适的操作
删除分区	ALTER TABLE table_name DROP PARTITION partition_names 删除分区

Table A.5. 过程性语言

过程性语言	概述
匿名块	支持Oracle匿名块语法，begin end;
数据类型	支持完备的数据类型long、raw等
流程控制	支持条件选择、循环控制、顺序控制
集合与记录	支持条件选择、循环控制、顺序控制
静态SQL	支持
动态SQL	支持
自治事务	支持自治事务
触发器	支持行级、语句级和系统级别触发器, 但是不兼容Oracle的触发器
异常处理	支持完备异常处理、支持抛出异常到数据库日志等特性
程序包	支持
自定义数据类型	支持record、type自定义类型
PLSQL系统包	支持 DBMS_ALERT, DBMS_LOB, DBMS_OUTPUT, DBMS_PIPE, DBMS_RANDOM, 等总计16个系统包
PLSQL 标签安全包	不支持

存储过程相关使用请参考plorasql存储过程.

Table A.6. 系统视图

系统视图	概述
dba/user/all数据字段视图	兼容138个字典视图
v\$数据字段视图	兼容1个 (v\$nls_parameters)

性能视图请参考性能视图.

Table A.7. 字符集和字符序

字符集和字符序	概述
UTF-8	支持UTF-8字符集
GBK	服务端、客户端支持GBK, GB18030-2022, 已通过人行国标认证

字符集和字符序的使用请参考字符集.

```
lightdb@test=# select datname, datcollate, datctype from pg_database;
 datname | datcollate | datctype
-----+-----+-----
 postgres | en_US.utf8 | en_US.utf8
 lt_test  | en_US.utf8 | en_US.utf8
 template1 | en_US.utf8 | en_US.utf8
 template0 | en_US.utf8 | en_US.utf8
 test     | en_US.utf8 | en_US.utf8
(6 rows)
```

Table A.8. 数据库导入导出

数据库导入导出	概述
csv文本数据导出与加载	COPY、ltuldr、ltldr、lt_bulkload
逻辑二进制数据导出与加载	Oracle用法:expdp、impdp、exp、imp;LightDB用法:lt_dump、lt_restore

lt\_bulkload的细的使用请参考 lt\_bulkload. 下面主要演示lt\_dump和lt\_restore的逻辑备份恢复功能, 详细的使用请参考lt\_dump和lt\_restore.

```
PGPASSWORD=p1 lt_dump -Fd -U lightdb -h 10.19.36.28 -p 5432 -d fund60 -j8 -f fund60 -c --if-exists -v > exp.log
```

```
PGPASSWORD=p1 lt_restore fund60 -U lightdb -h 10.19.36.28 -p 5432 -n fund1 -n fund2 -j8 -d fund60 --section=pre-data --section=data --if-exists -c -v > imp.log
```

```
PGPASSWORD=p1 lt_restore fund60 -U lightdb -h 10.19.36.28 -p 5432 -n fund1 -n fund2 -d fund60 --section=post-data --if-exists -c -v >> imp.log
```

#### 说明

- PGPASSWORD=密码 表示指定数据库密码, 同样也可以export PGPASSWORD=pwd1
- lt\_dump 逻辑备份命令
- -U 指定导出用户名
- -h 指定导出实例IP
- -p 指定导出实例端口
- -d 指定导出数据库名
- -n 导出数据库的schema名字, 可以指定多个schema, 如: -n subacc1 -n subacc2
- -j 10 表示使用10个并行进行导出
- -F d 其中-F表示导出文件格式, d表示为目录格式
- --if-exists -c 如果存在就删除重新创建
- -f 导出文件名
- -v 表示输出日志详情

习惯于MySQL的mysqldump的同学可能会有疑问, 一般的密码提示都是放在备份命令中, 而PGPASSWORD为什么要放前面, 其实这个PG/LightDB的规范, 他其实也是shell中的变量



Table A.9. 事务一致性与隔离级别

事务一致性与隔离级别	概述
隔离级别	支持SQL标准的四种隔离级别，包括读未提交、读已提交、可重复读和串行化。默认RC读已提交隔离级别。
并发控制机制	LightDB使用行记录信息来记录当前事务号(xmin为insert或回滚时的事务号、xmax为update或删除的事务号，注意xmin还会记录回滚时的事务号)，以此实现多版本并发控制。
Undo	LightDB中通过行记录中xmin/xmax标记事务信息来进行回滚

Table A.10. 优化器

优化器	概述
优化器	支持基于成本的优化器CBO，生成执行计划，不支持布隆过滤器，对于复杂SQL支持良好
执行器	LightDB 执行器也支持各种算子、并行、链接算法、平铺、分区裁剪

Table A.11. 优化器提示

优化器提示	概述
优化器提示	LightDB支持完备的优化器提示、包含并行执行、表关联方式控制、星型链接等多种优化器提示，暂不支持跨库访问

优化器提示相关请参考lt\_hint\_plan.

Table A.12. 高可用

高可用	概述
高可用	使用RAC来实现高可用性。RAC是一种共享磁盘数据库集群解决方案，允许多个节点访问同一数据库，提供了故障转移、负载均衡等功能。此外，Oracle还提供了数据复制技术Data Guard。使用流复制Streaming Replication来实现高可用性。主库将变更操作的Write-Ahead Logs发送到从库，从库根据这些日志来更新其数据

Table A.13. 备份恢复

备份恢复	概述
备份	逻辑备份：lt_dump/lt_dumpall来备份数据库或表的逻辑结果和数据；物理备份：使用lt_basebackup或拷贝备份LTDATA目录。
恢复	Oracle数据库使用RMAN可以恢复整个数据库、单个表空间、单个数据文件或者某个时间点的数据库。它还可以在恢复过程中自动处理故障。LightDB使用lt_dump进行逻辑备份，使用lt_restore进行逻辑恢复。对于

备份恢复	概述
	lt_basebackup的备份，可以直接将备份文件拷贝到数据目录进行物理恢复

Table A.14. 安全特性

安全特性	概述
安全相关	Oracle支持TDE透明加密、提供数据库级别审计日志，静态数据加密，IP白名单访问控制 LightDB支持TDE透明加密，OPE保序加密，SM国密，提供多种数据库级别审计日志，IP访问控制

Table A.15. 分区支持

分区支持	概述
安全相关	LightDB支持List、hash、range分区和组合二级分区方式，同时支持global索引、分区索引、分区裁剪、分区交换、丢弃等特性

使用请参考数据定义.

Table A.16. 并行执行

并行执行	概述
并行执行	LightDB支持支持广播、hash hash、random多种并行的数据分发处理方式，并行不支持bloom过滤

---

# Appendix B. MySQL兼容特性

LightDB 在语法、用法上和MySQL存在差异，本附录从数据类型、函数类型、SQL语法几个方面，介绍MySQL的兼容性。

## B. 1. MySQL兼容特性列表

### B. 1. 1. 数据类型

数值类型:

- unsigned int、signed int 在mysql兼容模式下，支持cast as signed/unsigned功能

示例:

```
lightdb@mysqltest=# create table test143 as select cast(123 as signed) c1,
cast(-123 as unsigned) c2;
```

```
SELECT 1
```

```
lightdb@mysqltest=# \d test143
```

```
Table "public.test143"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----
c1     | signed |           |          |
c2     | unsigned |           |          |
```

```
lightdb@mysqltest=# drop table test143;
```

```
DROP TABLE
```

```
lightdb@mysqltest=# create table test143 as select cast('123' as signed) c1,
cast('-123' as unsigned) c2;
```

```
SELECT 1
```

```
lightdb@mysqltest=# \d test143
```

```
Table "public.test143"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----
c1     | signed |           |          |
c2     | unsigned |           |          |
```

大对象类型:

- LONGTEXT LONGTEXT是TEXT的别名。

示例:

```
lightdb@mysqltest=# create table test01(t longtext);
```

```
CREATE TABLE
```

```
lightdb@mysqltest=# \d test01
```

```
Table "public.test01"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----
t     | longtext |           |          |
```

时间日期类型:

- DATETIME DATETIME 是 timestamp without zone 的别名。

示例:

```
lightdb@mysqltest=# create table vatsa(Dt DATETIME);
CREATE TABLE
lightdb@mysqltest=# INSERT INTO vatsa VALUES('2023-11-05 23:13:59');
INSERT 0 1
lightdb@mysqltest=# \d vatsa
                Table "public.vatsa"
Column |          Type          | Collation | Nullable | Default
-----+-----+-----+-----+-----
dt      | timestamp without time zone |           |          |
```

## B. 1. 2. 函数类型

系统信息函数:

- DATABASE() 同current\_schema()的同义词。在MySQL中，模式的概念类似于数据库。您可以通过设置search\_path来指定当前模式。语法如下：DATABASE() returns TEXT 1) 返回搜索路径中第一个模式的名称（如果搜索路径为空，则返回空值）。 2) 与MySQL类似，这是创建没有指定目标模式的任何表或其他命名对象时将使用的模式。

示例:

```
lightdb@mysqltest=# select database();
database
-----
public
(1 row)
```

数值函数:

- TRUNCATE(X, D) 在mysql兼容模式下，TRUNCATE(X, D) 函数，其中，X是数值，D是保留小数的位数，其作用就是按照小数位数，进行数值截取（此处的截取是按保留位数直接进行截取，没有四舍五入），数值保留规则如下： 1) D大于0，将数值X的小数部分保留D位，其余的删除 2) D等于0，将数值X的小数部分去除，只保留整数部分 3) D小于0，将数值X的小数部分去除，并将整数部分按照D指定位数，用0替换

示例:

```
lightdb@mysqltest=# SELECT TRUNCATE(-123.4567, 3);
truncate
-----
-123.456
(1 row)

lightdb@mysqltest=# SELECT TRUNCATE(123.4567, 3);
truncate
-----
123.456
(1 row)
```

- CONV conv( p\_num text, p\_from\_base int4, p\_to\_base int4) RETURNS text。 1) 在不同的进制之间转换数字。 2) 将数字N从进制from\_base转换为进制to\_base并返回字符串表示形式。如果任何参数为NULL, 则返回NULL。参数N被解释为整数, 但可以指定为整数或字符串。 3) 最小进制为2, 最大进制为36。如果from\_base是负数, 则N被视为带符号数。否则, N被视为无符号数。CONV()使用64位精度。

示例:

```
lightdb@mysqltest=# select conv('123456', 10, 16);
conv
-----
1E240
(1 row)
```

- LOG10 返回参数的以10为底的对数。语法如下: log10(p1 numeric) RETURNS numeric  
log10(p1 double precision) RETURNS double precision

示例:

```
lightdb@mysqltest=# select log10(2);
log10
-----
0.3010299956639812
(1 row)
```

- LOG2 返回参数的以2为底的对数。语法如下: log2(p\_num numeric) RETURNS numeric

示例:

```
lightdb@mysqltest=# select log2(64);
log2
-----
6.0000000000000000
(1 row)
```

- RAND 返回范围在0和1.0之间的随机浮点值v。语法如下: rand(p\_seed int default null) RETURNS double precision

示例:

```
lightdb@mysqltest=# select rand(123);
rand
-----
0.05425240159967615
(1 row)
```

字符函数:

- ELT 返回索引号处的字符串。语法如下: elt(str\_pos int, VARIADIC strlist text[]) RETURNS text。 1) ELT()返回字符串列表中的第str\_pos个元素: 如果str\_pos = 1, 则为str1, 如果str\_pos = 2, 则为str2, 以此类推。 2) 如果N小于1或大于参数个数, 则返回NULL。 3) ELT()是FIELD()的补充函数。

示例:

```
lightdb@mysqltest=# select elt( 2, 'Aa', 'Bb', 'Cc', 'Dd', 'Ff');
elt
-----
Bb
(1 row)
```

- FIELD(str, str1, str2, str3, ...) 返回str参数在后续参数中的位置, 如果没有找到则返回0。

示例:

```
lightdb@mysqltest=# SELECT FIELD('Bb', 'Aa', 'Bb', 'Cc', 'Dd', 'Ff');
field
-----
2
(1 row)
```

- FIND\_IN\_SET(str, strlist) FIND\_IN\_SET(str, strlist)函数, 用来查询strlist中包含str的结果, 其中str是要查询的字符串, strlist是一个字符串列表, 列表中各字符串以“, ”分隔, 如‘a, b, c, d’ 该函数的返回值规则如下: (1) 如果字符串str存在于由N个字符串组成的字符串列表strlist中, 则返回值的范围在1到N之间, 具体取决于在列表中匹配到的位置; (2) 如果str不在strlist 或strlist 为空字符串, 则返回值为0; (3) 如果任意一个参数为NULL, 则返回值为NULL; (4) 如果第一个参数含有逗号 ‘,’, 该函数将无法正常运行, 会始终返回0。

示例:

```
lightdb@mysqltest=# select find_in_set('b', 'a ,b,c,d');
find_in_set
-----
2
(1 row)
```

- INSTR 获取子字符串的第一次出现的位置。语法如下: instr(str text, patt text) RETURNS int 1) 从1开始, 在字符串str中获取子字符串patt的第一次出现的位置。2) 如果未找到, 则返回0。

示例:

```
lightdb@mysqltest=# select instr('abc124deff4de', '4de');
instr
-----
6
(1 row)
```

- FROM\_BASE64 解码 base64 编码的字符串并返回结果。语法如下: from\_base64(p\_str text) RETURNS text 1) 接受一个使用 TO\_BASE64() 使用的 base-64 编码规则编码的字符串, 并将解码后的结果作为二进制字符串返回。2) 如果参数为 NULL 或不是有

效的 base-64 字符串，则结果为 NULL。 3) 有关编码和解码规则的详细信息，请参见 TO\_BASE64() 的描述。

示例：

```
lightdb@mysqltest=# select from_base64('YWJj');
from_base64
-----
abc
(1 row)
```

- INSTR 获取子字符串的第一次出现的位置。语法如下：instr(str text, patt text) RETURNS int 1) 从1开始，在字符串str中获取子字符串patt的第一次出现的位置。 2) 如果未找到，则返回0。

示例：

```
lightdb@mysqltest=# select instr('abc124deff4de', '4de');
instr
-----
6
(1 row)
```

- LCASE LOWER() 的同义词。语法如下：lcase(p\_source text) RETURNS text 1) LCASE() 是LOWER() 的同义词。 2) 根据当前字符集映射，将字符串str中的所有字符更改为小写。默认为utf8mb4。

示例：

```
lightdb@mysqltest=# select lcase('QuadRatic');
lcase
-----
quadratic
(1 row)
```

- group\_concat group\_concat行转列功能函数，可以将select xxx group by分组查询的各行非NULL值拼接成一个字符串返回，其中的各行value默认以逗号',' 隔开。

示例：

```
lightdb@mysqltest=# select student_name,group_concat(test_score) as score
lightdb@mysqltest-# from student
lightdb@mysqltest-# where student_name='张三'
lightdb@mysqltest-# group by student_name;
student_name | score
-----+-----
张三          | 90,100,76
(1 row)
```

- if(expr1,expr2,expr3) 实现if(expr1,expr2,expr3)函数，返回值固定为字符串类型，以部分实现与mysql兼容，可以满足绝大多数情况下mysql的实际用法。

示例:

```
lightdb@mysqltest=# select if(2 > 1, 2, 1);
if
-----
2
(1 row)
```

- LOCATE(substr, str)、LOCATE(substr, str, pos) locate函数功能, 用户实现mysql locate函数功能, 支持数值类型和字符类型。

示例:

```
lightdb@mysqltest=# SELECT LOCATE(' bar', ' foobarbar');
locate
-----
4
(1 row)

lightdb@mysqltest=# SELECT LOCATE(' bar', ' foobarbar', 5);
locate
-----
7
(1 row)
```

- MID 从指定位置开始返回一个子字符串。语法如下: mid(p\_source text, p\_pos bigint, p\_len bigint) RETURNS text 1) 不带 len 参数的形式从字符串 str 的位置 pos 开始返回一个子字符串。带有 len 参数的形式从字符串 str 的位置 pos 开始返回一个长度为 len 的子字符串。2) pos 也可以使用负值。在这种情况下, 子字符串的开头是距离字符串末尾 pos 个字符, 而不是开头。负值可以用于此函数的任何形式中的 pos。对于 pos 的值为 0, 返回一个空字符串。

示例:

```
lightdb@mysqltest=# select mid('Hongye', 2, 4);
mid
-----
ongy
(1 row)
```

- SPACE 返回指定数量的空格字符串。语法如下: space(p\_num int) RETURNS text 返回由 N 个空格字符组成的字符串。

示例:

```
lightdb@mysqltest=# select concat('|', space(11), '|');
concat
-----
|           |
(1 row)
```



- STRCMP 比较两个字符串。语法如下： `strcmp(p1 text, p2 text) RETURNS int` STRCMP() 返回 0 表示两个字符串相同，返回 -1 表示第一个参数小于第二个参数按照当前的排序顺序，返回 1 表示其他情况。

示例：

```
lightdb@mysqltest=# select strcmp('text', 'text2');
      strcmp
-----
          -1
(1 row)
```

- TO\_BASE64 将参数转换为 base-64 字符串并返回。语法如下： `to_base64(p_str text) RETURNS text` 1) 将字符串参数转换为 base-64 编码形式，并返回一个使用连接字符集和排序规则的字符串作为结果。 2) 如果参数不是字符串，则在转换之前将其转换为字符串。如果参数为 NULL，则结果为 NULL。 3) 使用 FROM\_BASE64() 函数可以解码 base-64 编码字符串。

示例：

```
lightdb@mysqltest=# select to_base64('abc');
      to_base64
-----
          YWJj
(1 row)
```

- UCASE UPPER() 的同义词。语法如下： `ucase(p_source text) RETURNS text` 将字符串 str 中的所有字符根据当前字符集映射更改为大写。默认字符集为 utf8mb4。

示例：

```
lightdb@mysqltest=# select ucase('QuadRatic');
      ucase
-----
    QUADRATIC
(1 row)
```

- SUBSTRING(s, start)、SUBSTRING(s, start, length) 兼容mysql substring函数，-N偏移的含义为从尾部往前(最后一个字符数1)数N。 SUBSTRING(s, start)，则第一个参数为待截取的字符串，第二个参数为截取的起始位置。如果第二个参数为负整数，则为倒数的起始位置； SUBSTRING(s, start, length)，则第一个参数为待截取的字符串，第二个参数为截取的起始位置，第三个参数为截取的长度。如果第二个参数为负整数，则为倒数的起始位置。

示例：

```
lightdb@mysqltest=# select substring('abc', -2, 2);
      substring
-----
          bc
(1 row)
```

- SUBSTR 返回按指定要求截取的字字符串。语法如下： substr(str text, pos int) RETURNS text substr(str text, pos int, len int) RETURNS text 1) 不带 len 参数的形式返回从字符串 str 中位置 pos 开始的子字符串。带有 len 参数的形式返回从字符串 str 中位置 pos 开始长度为 len 的子字符串。 2) pos 参数也可以使用负数。在这种情况下，子字符串的开始位置是距离字符串结尾 pos 个字符，而不是开头。 pos 参数可以在此函数的任何形式中使用负值。当 pos 参数为 0 时，返回空字符串。 3) 此函数支持多字节字符集。如果任何参数为 NULL，则返回 NULL。 4) 如果 len 小于 1，则结果为空字符串。 5) 如果 pos 和 len 都接受数字类型，则也是可以的。如果输入的小数部分小于 0.5，则将其四舍五入为整数部分。否则，将正数的四舍五入为输入值加上 0.5 的下限，将负数的四舍五入为输入值减去 0.5 的下限。

示例：

```
lightdb@mysqltest=# select substr('abc',-2,2);
substr
-----
bc
(1 row)
```

- cast as char (str as char) 兼容mysql cast as char函数，返回正常字符串，无截断。

示例：

```
lightdb@mysqltest=# select cast(823666644 as char);
varchar
-----
823666644
(1 row)
```

时间日期函数：

- DATEDIFF(date1,date2) datediff支持文本，timestamp,date。date1 和 date2 参数是合法的日期或日期/时间表达式。

示例：

```
lightdb@mysqltest=# select datediff(20140102,20140301);
datediff
-----
-58
(1 row)
```

- DATE\_ADD 将指定的时间间隔添加到日期中。语法如下： DATE\_ADD(expr1 timestamptz, expr2 interval) RETURN date DATE\_ADD(expr1 text, expr2 interval) RETURN date 1) 返回添加间隔（年/月/日）到 expr1 日期参数后的日期类型。 2) 第二个参数 expr2 仅支持间隔（年/月/日）。

示例：

```
lightdb@mysqltest=# select date_add('20200101', interval 1 day);
date_add
-----
2020-01-02
```

(1 row)

- DATE\_FORMAT 日期格式显示。语法如下：DATE\_FORMAT(expr1 timestamp, expr2 text) RETURN text DATE\_FORMAT(expr1 timestamptz, expr2 text) RETURN text DATE\_FORMAT(expr1 text, expr2 text) RETURN text 返回expr1时间参数，在expr2格式中显示和输出。

示例：

```
lightdb@mysqltest=# select date_format('20211201 11:59:59', '%Y/%m/%d %h,%i,%s');
      date_format
-----
2021/12/01 11, 59, 59
(1 row)
```

- DATE\_FORMAT 日期格式显示。语法如下：DATE\_FORMAT(expr1 timestamp, expr2 text) RETURN text DATE\_FORMAT(expr1 timestamptz, expr2 text) RETURN text DATE\_FORMAT(expr1 text, expr2 text) RETURN text 返回expr1时间参数，在expr2格式中显示和输出。

示例：

```
lightdb@mysqltest=# select date_format('20211201 11:59:59', '%Y/%m/%d %h,%i,%s');
      date_format
-----
2021/12/01 11, 59, 59
(1 row)
```

- DATE\_SUB 日期减法。语法如下：DATE\_SUB(expr1 timestamptz, expr2 interval) RETURN date DATE\_SUB(expr1 text, expr2 interval) RETURN date 返回 expr1 日期参数减去间隔（年/月/日）后的日期类型 第二个参数 expr2 只支持间隔（年/月/日）

示例：

```
lightdb@mysqltest=# select date_sub('20200101', interval 1 day);
      date_sub
-----
2019-12-31
(1 row)
```

- DAY 返回月份日期。语法如下：DAY(expr1 text) RETURN int2 DAY(expr1 timestamptz) RETURN int4 返回给定 expr1 日期的月份部分 仅支持月和日参数，例如“0101”、“0704”。

示例：

```
lightdb@mysqltest=# select day('2011/12/01');
      day
-----
1
```

(1 row)

- **DAYOFWEEK** 返回星期几。语法如下：`DAY(expr1 text) RETURN int2 DAY(expr1 timestamp tz) RETURN int4` 根据 `expr1` 日期参数返回星期几。返回值为 1 ~ 7，星期日为 1。至少包括完整的时间格式，例如 20200202、200202。不支持少于 6 位的参数。

示例：

```
lightdb@mysqltest=# select dayofweek(20200101);
      dayofweek
-----
              4
(1 row)
```

- **FROM\_UNIXTIME** 根据 UNIX 时间戳返回时间格式。语法如下：`FROM_UNIXTIME(expr1 text) RETURN timestamp FROM_UNIXTIME(expr1 int8) RETURN timestamp FROM_UNIXTIME(expr1 text, expr2 text) RETURN text FROM_UNIXTIME(expr1 int8, expr2 text) RETURN text` 根据 `expr1` 参数的 UNIX 时间戳返回时间格式。日期格式可以根据 `expr2` 参数进行指定。

示例：

```
lightdb@mysqltest=# select from_unixtime(20220910);
      from_unixtime
-----
1970-08-23 08:55:10
(1 row)
```

```
lightdb@mysqltest=# select from_unixtime(20220910, '%Y/%m/%d %h,%i,%s');
      from_unixtime
-----
1970/08/23 08,55,10
(1 row)
```

- **SEC\_TO\_TIME** 将秒转换为“hh:mm:ss”格式。语法如下：`sec_to_time(p_secs int) RETURNS interval` 返回将秒结果的范围受到 `TIME` 数据类型的限制。如果参数对应于超出该范围的值，则会发出警告。参数转换为小时、分钟和秒的时间值。

示例：

```
lightdb@mysqltest=# select mysql.sec_to_time(80580);
      sec_to_time
-----
22:23:00
(1 row)
```

- **str\_to\_date** 在mysql模式下，`str_to_date`函数与mysql功能一致，在输入仅有日期的时候，返回date类型。在输入有日期时间的时候，返回timestamp类型(without timezone)。在仅有time字段的时候，返回time类型。

示例：

```
lightdb@mysqltest=# select str_to_date('04/30/2004', '%m/%d/%Y');
```

```
str_to_date
```

```
-----
2004-04-30
(1 row)
```

```
lightdb@mysqltest=# select str_to_date('04/30/2004 01:02:03', '%m/%d/%Y
%h:%i:%s');
```

```
str_to_date
-----
2004-04-30 01:02:03
(1 row)
```

```
lightdb@mysqltest=# select str_to_date('09:30:17', '%h:%i:%s');
```

```
str_to_date
-----F
09:30:17
(1 row)
```

- `sysdate` 在mysql模式下，可以使用`sysdate()`函数来获取当前的时间。

示例：

```
lightdb@mysqltest=# select sysdate();
sysdate
-----
2023-11-07 00:05:40
(1 row)
```

- `TIME_TO_SEC` 将参数转换为秒并返回。语法如下：`time_to_sec(p_time time) RETURNS int` 返回转换为秒的时间参数。

示例：

```
lightdb@mysqltest=# select time_to_sec('22:23:00');
time_to_sec
-----
80580
(1 row)
```

- `TIMEDIFF` 计算时间差。语法如下：`timediff(p_date1 timestamptz, p_date2 timestamptz) RETURNS interval` `TIMEDIFF()` 将两个时间表达式相减并返回一个时间值。`expr1` 和 `expr2` 都必须是时间或日期时间表达式，并且类型必须相同。`TIMEDIFF()` 返回的结果范围限制在 `TIME` 值允许的范围内。或者，您可以使用 `TIMESTAMPDIFF()` 和 `UNIX_TIMESTAMP()` 这两个函数，两者都返回整数值。

示例：

```
lightdb@mysqltest=# select mysql.timediff('2000-01-01 00:00:00', '2000-01-01
00:00:00.000001');
timediff
-----
-00:00:00.000001
(1 row)
```

- `TIMESTAMPDIFF(unit, start_date, end_date)` 使用 `TIMESTAMPDIFF` 函数计算了订单的处理时长（以`unit`为单位，`unit`可以为`YEAR`、`MONTH`、`DAY`、`HOUR`、`MINUTE`、`SECOND` 等）。函数将 `created_at` 作为起始时间，`completed_at` 作为结束时间，并计算它们之间的分钟差异。

示例：

```
lightdb@mysqltest=# select timestampdiff ('SECOND', '1911-11-11
11:23:45'::timestamp, '2021-12-12 12:12:12'::timestamp);
timestampdiff
-----
3474060507
(1 row)
```

```
lightdb@mysqltest=# select timestampdiff ('MINUTE', '1911-11-11
11:23:45'::timestamp, '2021-12-12 12:12:12'::timestamp);
timestampdiff
-----
57901008
(1 row)
```

- `TO_DAYS` 返回将日期参数转换为天数的结果。语法如下：`to_days(p_date timestamp) RETURNS bigint` `to_days(p_date timestamptz) RETURNS bigint` `to_days(p_date text) RETURNS bigint` `to_days(p_date text, p_format text) RETURNS bigint` `to_days(p_date bigint) RETURNS bigint` 给定一个日期 `date`，返回一个日期编号（自公元 0 年以来的天数）。`TO_DAYS()` 不适用于格里高利历实行之前的日期（1582年），因为它没有考虑到更改日历时丢失的天数。对于 1582 年之前（以及其他地区可能的更晚年份）的日期，此函数的结果不可靠。

示例：

```
lightdb@mysqltest=# select to_days('1111-11-11');
to_days
-----
406098
(1 row)
```

- `TO_SECONDS` 返回将日期或日期时间参数转换为自公元 0 年以来的秒数。语法如下：`to_seconds(p_date timestamp) RETURNS bigint` `to_seconds(p_date timestamptz) RETURNS bigint` 给定日期或日期时间表达式 `expr`，返回自公元 0 年以来的秒数。如果 `expr` 不是有效的日期或日期时间值，则返回 `NULL`。与 `TO_DAYS()` 一样，`TO_SECONDS()` 不适用于格里高利历实行之前的日期（1582年），因为它没有考虑到更改日历时丢失的天数。对于 1582 年之前（以及其他地区可能的更晚年份）的日期，此函数的结果不可靠。

示例：

```
lightdb@mysqltest=# select to_seconds('2009-11-29'::timestamp);
to_seconds
-----
63426672000
(1 row)
```

- UNIX\_TIMESTAMP 返回根据时间格式的 UNIX 时间戳。语法如下： UNIX\_TIMESTAMP() RETURN int8 UNIX\_TIMESTAMP(expr1 timestampz) RETURN int8 UNIX\_TIMESTAMP(expr1 text) RETURN int8 根据 expr1 参数的时间格式返回 UNIX 时间戳。如果没有参数，则返回当前 UNIX 时间戳。 expr1 参数至少包含完整的日志格式，例如 20200101。

示例：

```
lightdb@mysqltest=# select unix_timestamp(20200101);
unix_timestamp
-----
1577808000
(1 row)
```

```
lightdb@mysqltest=# select unix_timestamp();
unix_timestamp
-----
1699754361
(1 row)
```

- WEEKOFYEAR 根据时间返回当年的周数。语法如下： WEEKOFYEAR(expr1 timestampz) RETURN int4 WEEKOFYEAR(expr1 text) RETURN int4 根据时间表达式参数expr1，返回当年的周数。至少包括完整的时间格式，例如20200202、200202。不支持少于6位的参数。

示例：

```
lightdb@mysqltest=# select weekofyear('20220910');
weekofyear
-----
36
(1 row)
```

- YEAR 返回日期的年份。语法如下： YEAR(expr1 timestampz) RETURN int4 YEAR(expr1 text) RETURN int4 返回根据expr1参数日期计算出的年份。支持完整的年、月、日时间参数，例如“20200101”和“200101”。不支持仅包括月和日参数，例如“0101”和“0704”。

示例：

```
lightdb@mysqltest=# select year(20200101);
year
-----
2020
(1 row)
```

```
lightdb@mysqltest=# select year('200101');
year
-----
2020
(1 row)
```

聚合函数：

- ANY\_VALUE 为了与MySQL模式兼容，仅在BY模式下的分组行为语法。语法如下： ANY\_VALUE(anyelement) 1) 此函数对于启用ONLY\_FULL\_GROUP\_BY SQL模式的MySQL兼容

GROUP BY查询非常有用，对于MySQL拒绝您知道有效的查询的情况，因为MySQL无法确定，任何值函数将返回导致歧义的第一行。函数返回值和类型与其参数的返回值和类型相同。2) 请注意，此行为仅是与MySQL兼容的非标准语法。它用于确保在LightDB中的执行不会报告错误，但无法保证结果与MySQL完全一致。不建议使用此功能。请使用标准语法格式。

示例：

```
lightdb@mysqltest=# select any_value(empno) as col, any_value(ename) as co2,
max(sal) from emp group by empno;
```

col	co2	max
7844	TURNER	1500.00
7900	JAMES	950.00
7369	SMITH	800.00
7566	JONES	2975.00
7876	ADAMS	1100.00
7782	CLARK	2450.00
7902	FORD	3000.00
7788	SCOTT	3000.00
7654	MARTIN	1250.00
7934	MILLER	1300.00
7521	WARD	1250.00
7839	KING	5000.00
7499	ALLEN	1600.00
7698	BLAKE	2850.00

- GROUP\_CONCAT 返回字符串值的连接，以分隔符为分隔符。语法如下：  
GROUP\_CONCAT([DISTINCT] column\_name1 [ORDER BY {unsigned\_integer | column\_name2 | expr} [ASC | DESC] [SEPARATOR delimiter TEXT]) returns TEXT  
1) 此函数从组中返回一个具有连接的非NULL值的字符串结果。2) 如果没有非NULL值，则返回NULL。3) 要对结果中的值进行排序，请使用ORDER BY子句。要按相反的顺序排序，请在ORDER BY子句中按排序列的名称添加DESC（降序）关键字。默认为升序。可以使用ASC关键字明确指定。4) 要消除重复值，请使用DISTINCT子句。5) 默认情况下，组中值之间的分隔符是逗号（,）。要明确指定分隔符，请使用SEPARATOR后跟应在组值之间插入的字符串文字值。要完全消除分隔符，请指定SEPARATOR''。

示例：

```
lightdb@mysqltest=# select GROUP_CONCAT(col1 separator ':') from
(values('BBBB',1), ('AAAA',1), ('CCCC',2)) as t(col1 ,col2) group by col2;
```

```
group_concat
-----
BBBB:AAAA
CCCC
(2 rows)
```

JSON函数：

- JSON\_ARRAY 生成一个包含指定元素的json数组。

示例：

```
lightdb@mysqltest=# SELECT JSON_ARRAY(1, 2, 3, 4, 5) AS jsonArray;
```



```
jsonarray
```

```
-----
[1, 2, 3, 4, 5]
(1 row)
```

- `JSON_OBJECT` 生成一个包含指定K-V对的json object。如果有key为NULL或参数个数为奇数，则抛错。

示例：

```
lightdb@mysqltest=# SELECT JSON_OBJECT('name', 'John', 'age', 30, 'city',
'New York') AS jsonObject;
-----
{"age": 30, "city": "New York", "name": "John"}
(1 row)
```

- `JSON_EXTRACT` 从json文档里抽取数据。如果有参数有NULL或path不存在，则返回NULL。如果抽取多个path，则返回的数据封闭在一个json array里。

示例：

```
lightdb@mysqltest=# select json_extract('{"name": "John", "age": 30,
"city": "New York"}', '$.city');
-----
"New York"
(1 row)
```

- `JSON_CONTAINS` 查询json文档是否在指定path包含指定的数据，包含则返回true，否则返回false。如果有参数为NULL或path不存在，则返回NULL。

示例：

```
lightdb@mysqltest=# SELECT JSON_CONTAINS('{"name": "John", "age": 30,
"city": "New York"}', '"John"', '$.name') AS containsJohn,
lightdb@mysqltest=# JSON_CONTAINS('{"name": "John", "age": 30,
"city": "New York"}', '30', '$.age') AS containsAge30;
-----
containsjohn | containsage30
-----+-----
t            | t
(1 row)
```

- `JSON_CONTAINS_PATH` 判断 JSON 文档是否包含指定的一个或多个 JSON 路径，返回 true 或 false。语法如下：`JSON_CONTAINS_PATH(json_doc jsonb, one_or_all text, variadic jsonpath[]) RETURNS bool` 1) 如果 JSON 路径均不存在，则返回 false。否则根据 `one_or_all` 参数确定返回值：'one'：如果任意一个 JSON 路径存在，则返回 true。否则 false。'all'：如果所有的 JSON 路径都存在，则返回 true。否则 false。2) 如果 JSON 文档或 JSON 路径是非法的值或 `one_or_all` 参数不是 'one' 或 'all'，则提示错误。3) 如果为了检测某个路径是否存在于 JSON 文档中，请使用 `JSON_CONTAINS()`。

示例：

```
lightdb@mysqltest=# SELECT JSON_CONTAINS_PATH('{"a": 111, "b": 222, "c": {"d":
4444}}', 'one', '$.a', '$.z');
json_contains_path
```

```
t
(1 row)
```

```
lightdb@mysqltest=# SELECT JSON_CONTAINS_PATH('{"a": 111, "b": 222, "c": {"d":
4444}}', 'one', '$.d');
json_contains_path
```

```
f
(1 row)
```

```
lightdb@mysqltest=# SELECT JSON_CONTAINS_PATH('{"a": 111, "b": 222, "c": {"d":
4444}}', 'all', '$.a', '$.b', '$.c');
json_contains_path
```

```
t
(1 row)
```

```
lightdb@mysqltest=# SELECT JSON_CONTAINS_PATH('{"a": 111, "b": 222, "c": {"d":
4444}}', 'all', '$.a', '$.b', '$.d');
json_contains_path
```

```
f
(1 row)
```

- JSON\_PRETTY 格式化 JSON 文档。语法如下：JSON\_PRETTY(json\_doc jsonb) RETURNS text 1) 返回值类型为 text。2) 无意义的空格和换行符不会影响输出结果。3) 对于参数为 NULL，则函数返回 NULL。如果参数是非法值，则提示错误。

示例：

```
lightdb@mysqltest=# SELECT JSON_PRETTY('{"a":{"b":{"z":1},"c":[5,
{"z":2}], "z":3}, "z":4}');
json_pretty
```

```
-----
{
  "a": {
    "b": {
      "z": 1
    },
    "c": [
      5,
      {
        "z": 2
      }
    ],
    "z": 3
  },
  "z": 4
}
(1 row)
```

- **JSON\_REMOVE**            **JSON**            文档中删除数据，返回修改后的结果。语法如下：  
JSON\_REMOVE(jsonb, jsonpath) RETURNS jsonb 1) 从 JSON 文档中删除数据，返回修改后的结果。如果任意参数为 NULL，则返回 NULL。2) 如果 JSON 文档是非法的值，或 JSON 路径是非法的值，或 JSON 路径为 \$，或 JSON 路径包含 \* 或 \*\* 通配符，则提示错误。3) 如果指定的 JSON 路径不存在，函数不会提示错误，而是直接返回原 JSON 文档；

示例：

```
lightdb@mysqltest=# SELECT JSON_REMOVE('{"a": 1, "b": [1,2,3], "c": {"d":
4}}', '$.c.d');
          json_remove
-----
{"a": 1, "b": [1, 2, 3], "c": {}}
(1 row)
```

- **JSON\_INSERT**        向        **JSON**        文档中插入数据，返回修改后的结果。语法如下：  
JSON\_INSERT(jsonb, jsonpath, anyelement) RETURNS jsonb JSON\_INSERT(jsonb,  
jsonpath, text) RETURNS jsonb 1) 向 JSON 文档中插入数据，返回修改后的结果。如果任意参数为 NULL，则返回 NULL。2) 如果 JSON 文档是非法的值，或 JSON 路径是非法的值，或 JSON 路径包含 \* 或 \*\* 通配符，则提示错误。3) 如果 JSON 路径存在，则该函数什么也不做，直接返回原值。否则先插入数据并返回修改后的结果。

示例：

```
lightdb@mysqltest=# SELECT JSON_INSERT('{ "a": 1, "b": [2, 3]}', '$.a', 10);
          json_insert
-----
{"a": 1, "b": [2, 3]}
(1 row)

lightdb@mysqltest=# SELECT JSON_INSERT('{ "a": 1, "b": [2, 3]}', '$.c', 200);
          json_insert
-----
{"a": 1, "b": [2, 3], "c": 200}
(1 row)
```

- **JSON\_REPLACE**        替换        **JSON**        文档的数据，返回修改后的结果。语法如下：  
JSON\_REPLACE(jsonb, jsonpath, anyelement) RETURNS jsonb JSON\_REPLACE(jsonb,  
jsonpath, text) RETURNS jsonb 1) 替换 JSON 文档的数据，返回修改后的结果。如果任意参数为 NULL，则返回 NULL。2) 如果 JSON 文档是非法的值，或 JSON 路径是非法的值，或 JSON 路径包含 \* 或 \*\* 通配符，则提示错误。3) 如果 JSON 路径不存在，则该函数什么也不做，直接返回原值。否则先替换数据并返回修改后的结果。

示例：

```
lightdb@mysqltest=# SELECT JSON_REPLACE('{ "a": 1, "b": [2, 3]}', '$.a', 10);
          json_replace
-----
{"a": 10, "b": [2, 3]}
(1 row)

lightdb@mysqltest=# SELECT JSON_REPLACE('{ "a": 1, "b": [2, 3]}', '$.c',
'null');
```

```

      json_replace
-----
{"a": 1, "b": [2, 3]}
(1 row)

lightdb@mysqltest=# SELECT JSON_REPLACE('100', '$', '{ "a": 1, "b": [2,
      3]}')::jsonb);
      json_replace
-----
{"a": 1, "b": [2, 3]}
(1 row)

```

- **JSON\_SET** 替换 JSON 向 JSON 文档中插入数据或替换 JSON 文档的数据，返回修改后的结果。语法如下：  
**JSON\_SET(jsonb, jsonpath, anyelement)** RETURNS jsonb  
**JSON\_SET(jsonb, jsonpath, text)** RETURNS jsonb  
 1) 向 JSON 文档中插入数据或替换 JSON 文档的数据，返回修改后的结果。如果任意参数为 NULL，则返回 NULL。  
 2) 如果 JSON 文档是非法的值，或 JSON 路径是非法的值，或 JSON 路径包含 \* 或 \*\* 通配符，则提示错误。  
 3) 如果 JSON 路径存在，先替换数据并返回修改后的结果。如果 JSON 路径不存在，则先插入数据并返回修改后的结果。

示例：

```

lightdb@mysqltest=# SELECT JSON_SET('{ "a": 1, "b": [2, 3]}', '$.a', 10);
      json_set
-----
{"a": 10, "b": [2, 3]}
(1 row)

lightdb@mysqltest=# SELECT JSON_SET('{ "a": 1, "b": [2, 3]}', '$.c', '[true,
      false]');
      json_set
-----
{"a": 1, "b": [2, 3], "c": "[true, false]"}
(1 row)

lightdb@mysqltest=# SELECT JSON_SET('{ "a": 1, "b": [2, 3]}', '$.b.z', 4);
      json_set
-----
{"a": 1, "b": [2, 3]}
(1 row)

```

### B. 1. 3. SQL语法

参数配置：

- **lightdb\_only\_full\_group\_by** 仅在MySQL兼容模式下，此参数用于控制mysql的SELECT DISTINCT... ORDER BY语法是否与ONLY\_FULL\_GROUP\_BY以外的mysql语法兼容。此参数默认值为on。

示例：

默认情况下，lightdb将报告SELECT DISTINCT... ORDER BY语法中不一致列的错误：

```

lightdb@mysqltest=# set lightdb_only_full_group_by = on;
SET

```

```
lightdb@mysqltest=# select distinct job from emp order by empno;
ERROR: for SELECT DISTINCT, ORDER BY expressions must appear in select
list
LINE 1: select distinct job from emp order by empno;
```

只有在MySQL兼容模式和lightdb\_only\_full\_group\_by设置off时，上述SQL才能正常执行：

```
lightdb@mysqltest=# select distinct job from emp order by empno;
job
-----
CLERK
SALESMAN
MANAGER
ANALYST
PRESIDENT
(5 rows)
```

转换类型：

- 数值类型与空字符比较 在mysql兼容模式下，数值跟空串比较，会将空串转换为0，即实际为数值与0进行比较。

示例：

```
lightdb@mysqltest=# create table my_table(id1 int, id2 bigint, id3 smallint,
id4 number(5,3));
CREATE TABLE
lightdb@mysqltest=# insert into my_table
values(213453, 92233720368547, 32765, 12.233);
INSERT 0 1

lightdb@mysqltest=# Select * from my_table where id1 != '';
id1 | id2 | id3 | id4
-----+-----+-----+-----
213453 | 92233720368547 | 32765 | 12.233
(1 row)
```

- 整数相除兼容 在mysql兼容模式下，整数相除得到浮点数。

示例：

```
lightdb@mysqltest=# select 1/3;
?column?
-----
0.33333333333333333333333333333333
(1 row)

lightdb@mysqltest=# select 4/2;
?column?
-----
2.00000000000000000000000000000000
(1 row)
```

- 数值类型与空字符串比较 在mysql兼容模式下，数值跟空串比较，会将空串转换为0，即实际为数值与0进行比较。

示例：

```
lightdb@mysqltest=# create table my_table(id1 int, id2 bigint, id3 smallint,
id4 number(5, 3));
CREATE TABLE
lightdb@mysqltest=# insert into my_table
values(213453, 92233720368547, 32765, 12.233);
INSERT 0 1

lightdb@mysqltest=# Select * from my_table where id1 != '';
id1 | id2 | id3 | id4
-----+-----+-----+-----
213453 | 92233720368547 | 32765 | 12.233
(1 row)
```

- 时间类型和字符串比较 兼容mysql支持时间类型和字符串进行比较（注：当字符串为常量时支持两者的比较）。

示例：

```
lightdb@mysqltest=# create table foo(a text, b varchar(19), c char(19));
CREATE TABLE
lightdb@mysqltest=# insert into foo values ('2022-01-01 00:01:02', '2022-01-01
00:01:02', '2022-01-01 00:01:02');
INSERT 0 1
lightdb@mysqltest=# insert into foo values ('2024-01-01 00:01:02', '2024-01-01
00:01:02', '2024-01-01 00:01:02');
INSERT 0 1
lightdb@mysqltest=# select a, b, c from foo where now() > a;
a | b | c
-----+-----+-----
2022-01-01 00:01:02 | 2022-01-01 00:01:02 | 2022-01-01 00:01:02
(1 row)
```

- bool->varchar 兼容mysql，bool类型true转换至字符串'1'，false至字符串'0'。

示例：

```
lightdb@mysqltest=# create table test(col varchar(2), co2 varchar(8));
CREATE TABLE
lightdb@mysqltest=# insert into test(col) values(true);
INSERT 0 1
lightdb@mysqltest=# insert into test(co2) values(true);
INSERT 0 1
lightdb@mysqltest=# select * from test;
col | co2
-----+-----
1 | 1
(2 rows)
```

SQL语句:

- MySQL兼容DDL语法 兼容mysql ddl 包括key索引, comment注释, collate关键字, 字符集声明, engine。支持create\insert/update/delete操作。

示例:

```
lightdb@mysqltest=# CREATE TABLE `lt_test_mysql_ddl` (
lightdb@mysqltest(#   `name` char(64) COLLATE utf8_bin NOT NULL DEFAULT '',
lightdb@mysqltest(#   `ret` tinyint(1) NOT NULL DEFAULT '0',
lightdb@mysqltest(#   `dl` char(128) COLLATE utf8_bin NOT NULL DEFAULT '',
lightdb@mysqltest(#   -- `type` enum('function','aggregate') CHARACTER SET
    utf8 NOT NULL,
lightdb@mysqltest(#   PRIMARY KEY (`name`))
lightdb@mysqltest(# ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin
    COMMENT='User defined functions';
CREATE TABLE
```

- CREATE支持AUTO\_INCREMENT 属性 创建带AUTO\_INCREMENT属性字段的表, 增加带AUTO\_INCREMENT属性字段的表, 字段支持smallint、integer (int)、bigint类型。

示例:

```
lightdb@mysqltest=# create table t1(id int auto_increment, name text);
CREATE TABLE
lightdb@mysqltest=# insert into t1(name) values('mm');
INSERT 0 1
lightdb@mysqltest=# insert into t1(name) values('nn');
INSERT 0 1
lightdb@mysqltest=# select *from t1;
id | name
----+-----
 1 | mm
 2 | nn
(2 rows)
```

- CREATE支持COMMENT 属性 注释的添加, 是通过在定义表或列的时候在末尾加上 COMMENT关键字来实现的。

示例:

```
lightdb@mysqltest=# CREATE TABLE employees (
lightdb@mysqltest(#   id INT AUTO_INCREMENT PRIMARY KEY,
lightdb@mysqltest(#   first_name VARCHAR(50) NOT NULL COMMENT '员工名',
lightdb@mysqltest(#   last_name VARCHAR(50) NOT NULL COMMENT '员工姓',
lightdb@mysqltest(#   email VARCHAR(100) NOT NULL UNIQUE COMMENT '员工
电子邮件地址',
lightdb@mysqltest(#   birth_date DATE NOT NULL COMMENT '员工出生日期',
lightdb@mysqltest(#   hire_date DATE NOT NULL COMMENT '员工入职日期',
lightdb@mysqltest(#   department_id INT COMMENT '员工所属部门ID'
lightdb@mysqltest(# ) COMMENT='员工信息表';
CREATE TABLE
```

- ALTER支持AUTO\_INCREMENT 属性 增加带AUTO\_INCREMENT属性字段的表，字段支持smallint、integer (int)、bigint类型。

示例:

```
lightdb@mysqltest=# create table t2(id int NOT NULL, name text);
CREATE TABLE
lightdb@mysqltest=# alter table t2 add auto_increment(id);
ALTER TABLE
lightdb@mysqltest=# \d t2
                Table "public.t2"
Column | Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
id     | integer  |           | not null | generated always as identity
name   | text     |           |          |
```

- DROP INDEX 支持指定表名 在MySQL数据库中，drop index语句需要指定表名，这是因为MySQL支持在不同表上创建同名索引，所以仅凭索引名称不能唯一确定要删除的索引。

示例:

```
lightdb@mysqltest=# drop table t1;
DROP TABLE
lightdb@mysqltest=# create table t1(id int);
CREATE TABLE
lightdb@mysqltest=# create unique index idx_t1_id on t1(id);
CREATE INDEX
lightdb@mysqltest=# drop index idx_t1_id on t1;
DROP INDEX
```

- INSERT 支持自增序列NULL插入 在MySQL中可以通过AUTO\_INCREMENT关键字，实现列数据自增。一般用于单表的唯一标识列（常见的有主键）。每张表最多只能有一个自增列。Mysql的自增列支持null，当自增列插入NULL时，将设置为下一个序列值，一般就是当前最大值+1（效果等同于default）。

示例:

```
lightdb@mysqltest=# CREATE TABLE worker_node(id int NOT NULL
AUTO_INCREMENT, host_name varchar(64) NOT NULL);
CREATE TABLE
lightdb@mysqltest=# insert into worker_node (host_name) values('hello');
INSERT 0 1
lightdb@mysqltest=# insert into worker_node values(null, 'world');
INSERT 0 1

lightdb@mysqltest=# select * from worker_node;
 id | host_name
----+-----
  1 | hello
  2 | world
(2 rows)
```



- INSERT on duplicate 在MySQL数据库中，on duplicate key update的作用也就是说存在重复数据则更新，不存在则插入。

示例：

```
lightdb@mysqltest=# create table test(
  a int not null primary key,
  b int not null unique,
  c int not null
);
CREATE TABLE

lightdb@mysqltest=# insert into test values(1,1,1), (2,2,2);
INSERT 0 2
lightdb@mysqltest=# insert into test values(1,2,3) on duplicate key
update c = c + 1;
INSERT 0 1

lightdb@mysqltest=# select *from test;
a | b | c
---+---+---
2 | 2 | 2
1 | 1 | 2
(2 rows)
```

- insert ignore 在MySQL兼容模式下，使用IGNORE关键字可以导致服务器将某些类型的错误降级，并生成警告而不是错误。对于多行语句，将错误降级为警告可能会使某些行得以处理。否则，使用IGNORE关键字将导致语句跳过当前行而不是中止执行。对于不可忽略的错误，无论是否使用IGNORE关键字，都会导致错误发生。

示例：

```
lightdb@mysqltest=# CREATE TABLE test (i INT NOT NULL PRIMARY KEY, name
text);
CREATE TABLE
lightdb@mysqltest=# INSERT INTO test values(1,'xiao wang'),(2,'xiao
gang'),(3,'xiao ming');
INSERT 0 3

lightdb@mysqltest=# INSERT IGNORE INTO test values(3,'xiao hong');
WARNING: duplicate key value violates unique constraint "test_pkey"
DETAIL: Key (i)=(3) already exists.
INSERT 0 0
```

- INSERT 指定位置插入字符串 在指定位置插入子字符串，直到指定的字符数。语法如下：insert(p\_source text, p\_pos bigint, p\_len bigint, p\_replacement text) RETURNS text 1) 返回字符串str，将从位置pos开始并长度为len的子字符串替换为新str。 2) 如果pos不在字符串的长度范围内，则返回原始字符串。如果len不在其余字符串的长度范围内，则从位置pos替换其余字符串。 3) 如果任何参数为NULL，则返回NULL。

示例：

```
lightdb@mysqltest=# select insert('Quadratic', 3, 4, 'What');
```

```
insert
```

```
-----  
QuWhattic  
(1 row)
```

- `replace into` 在MySQL数据库中, `replace`与`insert`类似, 但是`REPLACE`在插入数据之前, 如果新行的主键值或唯一索引列值与旧行相同, 则会先删除旧行, 再插入新行。否则就等价于`insert into`。它是MySQL的专属语法, 是对SQL标准的扩展。

示例:

```
lightdb@mysqltest=# create table test ( col1 int not null, col2 char(16)  
unique, col3 int, col4 int, primary key(col1) );  
CREATE TABLE
```

```
lightdb@mysqltest=# insert into test values(1,'STR 1'), (2, 'STR 2'), (3,  
'STR 3'), (4, 'STR 4'), (6, 'STR 6');  
INSERT 0 5
```

```
lightdb@mysqltest=# replace into test values (1, 'STR M');  
INSERT 0 1
```

```
lightdb@mysqltest=# select * from test;
```

```
col1 |          col2          | col3 | col4  
-----+-----+-----+-----  
2 | STR 2                |      |      |  
3 | STR 3                |      |      |  
4 | STR 4                |      |      |  
6 | STR 6                |      |      |  
1 | STR M                |      |      |  
(5 rows)
```

---

Part IX. 附录

---

---

## Table of Contents

C. LightDB错误代码	2384
D. 日期/时间支持	2393
D. 1. 日期/时间输入解释	2393
D. 2. 处理无效或不明确的时间戳	2394
D. 3. 日期/时间关键词	2395
D. 4. 日期/时间配置文件	2395
D. 5. POSIX 时区规范	2397
D. 6. 单位的历史	2398
D. 7. 儒略日期	2399
E. SQL关键词	2401
F. SQL 符合性	2436
F. 1. 已支持特性	2437
F. 2. 未支持特性	2448
F. 3. XML对于SQL的限制和符合性/XML	2457
F. 3. 1. 查询只限于XPath 1.0	2457
F. 3. 2. 实现中的附带限制	2459
G. 版本说明	2460
G. 1. 版本发布 13.8-23.3	2460
G. 1. 1. Oracle 兼容	2460
G. 1. 2. plorasql 增强	2467
G. 1. 3. MySQL 兼容	2467
G. 1. 4. lightdb 新特性	2468
G. 1. 5. ltjdbc 增强	2469
G. 2. 版本发布 13.8-23.2	2469
G. 2. 1. Oracle 兼容	2469
G. 2. 2. plorasql 增强	2473
G. 2. 3. MySQL 兼容	2475
G. 2. 4. lightdb 新特性	2477
G. 2. 5. ltjdbc 增强	2478
G. 3. 版本发布 13.8-23.1	2478
G. 3. 1. 新增功能	2478
G. 3. 2. 优化	2478
G. 3. 3. 兼容性	2479
G. 4. 版本发布 13.8-22.4	2484
G. 5. 版本发布 13.8-22.3	2489
G. 6. 版本发布 13.3-22.2	2492
G. 7. 版本发布 13.3-22.1	2494
G. 8. 13.3-21.3版本发布	2497
G. 9. 版本发布 13.3-21.2	2499
G. 10. 版本 13.3-21.1	2500
H. 额外提供的模块	2502
H. 1. adminpack	2502
H. 2. amcheck	2503
H. 2. 1. 函数	2504
H. 2. 2. 可选的heapallindexed验证	2505
H. 2. 3. 有效地使用amcheck	2505
H. 2. 4. 修复损坏	2506
H. 3. auth_delay	2506
H. 3. 1. 配置参数	2506
H. 3. 2. 作者	2507
H. 4. auto_explain	2507
H. 4. 1. 配置参数	2507
H. 4. 2. 例子	2509
H. 4. 3. 作者	2509
H. 5. bloom	2509

---

H. 5.1.	参数	2509
H. 5.2.	例子	2510
H. 5.3.	操作符类接口	2512
H. 5.4.	限制	2512
H. 5.5.	作者	2512
H. 6.	btree_gin	2512
H. 6.1.	用法示例	2513
H. 6.2.	作者	2513
H. 7.	btree_gist	2513
H. 7.1.	用法示例	2513
H. 7.2.	作者	2514
H. 8.	canopy	2514
H. 9.	citext	2514
H. 9.1.	基本原理	2515
H. 9.2.	如何使用它	2515
H. 9.3.	串比较行为	2515
H. 9.4.	限制	2516
H. 9.5.	作者	2516
H. 10.	cube	2516
H. 10.1.	语法	2517
H. 10.2.	精度	2517
H. 10.3.	用法	2517
H. 10.4.	默认值	2519
H. 10.5.	注解	2520
H. 10.6.	工作人员	2520
H. 11.	dblink	2520
H. 12.	dict_int	2549
H. 12.1.	配置	2549
H. 12.2.	用法	2549
H. 13.	dict_xsyn	2550
H. 13.1.	配置	2550
H. 13.2.	用法	2550
H. 14.	file_fdw	2551
H. 15.	fuzzystrmatch	2553
H. 15.1.	Soundex	2553
H. 15.2.	Levenshtein	2554
H. 15.3.	Metaphone	2555
H. 15.4.	双 Metaphone	2555
H. 16.	hstore	2556
H. 16.1.	hstore 外部表示	2556
H. 16.2.	hstore 操作符和函数	2557
H. 16.3.	索引	2560
H. 16.4.	例子	2560
H. 16.5.	统计	2561
H. 16.6.	兼容性	2561
H. 16.7.	转换	2562
H. 16.8.	作者	2562
H. 17.	intagg	2562
H. 17.1.	函数	2562
H. 17.2.	使用示例	2562
H. 18.	intarray	2563
H. 18.1.	intarray 函数和操作符	2563
H. 18.2.	索引支持	2565
H. 18.3.	例子	2565
H. 18.4.	测试基准	2566
H. 18.5.	作者	2566
H. 19.	isn	2566
H. 19.1.	数据类型	2566

---

---

H. 19.2.	造型 .....	2567
H. 19.3.	函数和操作符 .....	2568
H. 19.4.	例子 .....	2568
H. 19.5.	参考文献 .....	2569
H. 19.6.	作者 .....	2569
H. 20.	lo .....	2569
H. 20.1.	原理 .....	2570
H. 20.2.	如何使用它 .....	2570
H. 20.3.	限制 .....	2570
H. 20.4.	作者 .....	2570
H. 21.	lt_buffercache .....	2570
H. 21.1.	pg_buffercache 视图 .....	2571
H. 21.2.	示例输出 .....	2571
H. 21.3.	作者 .....	2572
H. 22.	lt_bulkload (批量加载) .....	2572
H. 22.1.	概述 .....	2572
H. 22.2.	描述 .....	2572
H. 22.3.	用法 .....	2572
H. 22.4.	选项 .....	2573
H. 22.5.	控制文件 .....	2574
H. 22.6.	限制 .....	2579
H. 22.7.	详细信息 .....	2581
H. 23.	lt_cheat_funcs .....	2582
H. 23.1.	函数 .....	2582
H. 23.2.	编码转换 .....	2588
H. 23.3.	配置参数 .....	2589
H. 24.	ltcluster .....	2590
H. 25.	lt_cron .....	2590
H. 25.1.	lt_cron 是什么? .....	2590
H. 26.	lt_freemap .....	2594
H. 26.1.	函数 .....	2594
H. 26.2.	示例输出 .....	2595
H. 26.3.	作者 .....	2595
H. 27.	lt_hint_plan .....	2595
H. 27.1.	Name .....	2595
H. 27.2.	Synopsis .....	2595
H. 27.3.	Description .....	2596
H. 27.4.	The hint table .....	2596
H. 27.5.	提示中的详细信息 .....	2635
H. 27.6.	子查询 .....	2639
H. 27.7.	与 canopy 兼容 .....	2641
H. 27.8.	错误 .....	2644
H. 27.9.	功能限制 .....	2645
H. 27.10.	提示列表 .....	2645
H. 28.	lt_log_long_xact .....	2649
H. 28.1.	使用方法 .....	2649
H. 29.	lt_ope .....	2651
H. 29.1.	What is lt_ope? .....	2651
H. 29.2.	function usage .....	2651
H. 30.	lt_pathman .....	2655
H. 30.1.	概述 .....	2655
H. 30.2.	功能亮点 .....	2656
H. 30.3.	可用函数 .....	2656
H. 30.4.	视图和表 .....	2660
H. 30.5.	声明式分区 .....	2661
H. 30.6.	自定义计划节点 .....	2662
H. 30.7.	示例 .....	2665
H. 31.	lt_patch .....	2668

---

---

H. 31.1.	lt_patch 表	2668
H. 31.2.	函数	2669
H. 31.3.	示例输出	2669
H. 32.	lt_prewarm	2669
H. 32.1.	函数	2669
H. 32.2.	配置参数	2670
H. 32.3.	作者	2671
H. 33.	lt_proctab	2671
H. 33.1.	functions	2671
H. 34.	lt_profile (轻量级开发工具配置文件)	2673
H. 34.1.	概念	2673
H. 34.2.	扩展架构	2673
H. 34.3.	先决条件	2695
H. 34.4.	创建扩展	2695
H. 34.5.	权限	2696
H. 34.6.	使用lt_profile	2696
H. 34.7.	PWR报告中的节	2704
H. 34.8.	PSH报告的部分	2722
H. 35.	lt_qualstats	2723
H. 35.1.	Configuration	2723
H. 35.2.	更新扩展	2724
H. 35.3.	使用方法	2724
H. 36.	lt_show_plans	2727
H. 36.1.	配置参数	2727
H. 36.2.	用法	2727
H. 36.3.	Functions	2729
H. 37.	lt_sm	2729
H. 37.1.	什么是lt_sm?	2729
H. 37.2.	函数用法	2729
H. 37.3.	密钥管理	2732
H. 38.	lt_sql_inspect	2732
H. 38.1.	SQL 最佳实践	2732
H. 38.2.	SQL 拦截	2734
H. 39.	lt_standby_forward	2739
H. 39.1.	配置参数	2739
H. 39.2.	使用方法	2739
H. 39.3.	认证	2739
H. 39.4.	功能/限制	2739
H. 40.	lt_stat_activity	2740
H. 40.1.	概念	2740
H. 40.2.	先决条件	2740
H. 40.3.	创建扩展	2740
H. 41.	lt_stat_statements	2740
H. 41.1.	pg_stat_statements 视图	2741
H. 41.2.	函数	2743
H. 41.3.	配置参数	2743
H. 41.4.	样例输出	2744
H. 41.5.	作者	2746
H. 42.	lt_tempfile	2746
H. 43.	lt_trgm	2747
H. 43.1.	三元组 (或三图) 概念	2747
H. 43.2.	函数和运算符	2747
H. 43.3.	GUC参数	2749
H. 43.4.	索引支持	2749
H. 43.5.	文本搜索集成	2751
H. 43.6.	参考文献	2752
H. 43.7.	作者	2752
H. 44.	lt_visibility	2752

---

---

H. 44. 1.	函数	2752
H. 44. 2.	作者	2753
H. 45.	ltaudit	2753
H. 45. 1.	介绍	2753
H. 45. 2.	为什么使用ltaudit?	2753
H. 45. 3.	使用注意事项	2754
H. 45. 4.	设置	2754
H. 45. 5.	会话审计日志记录	2756
H. 45. 6.	对象审计日志记录	2757
H. 45. 7.	格式	2758
H. 45. 8.	注意事项	2759
H. 46.	ltpcrypto	2759
H. 46. 1.	通用哈希函数	2759
H. 46. 2.	密码哈希函数	2760
H. 46. 3.	PGP Encryption Functions	2762
H. 46. 4.	原始加密函数	2767
H. 46. 5.	随机数据函数	2768
H. 46. 6.	注释	2768
H. 46. 7.	作者	2770
H. 47.	ltfce	2771
H. 47. 1.	所有模式兼容的功能	2771
H. 47. 2.	函数	2771
H. 47. 3.	运算符类型兼容性	2772
H. 48.	ltfincore	2773
H. 48. 1.	DESCRIPTION	2773
H. 48. 2.	EXAMPLES	2773
H. 48. 3.	概要	2774
H. 48. 4.	文档	2775
H. 48. 5.	DEBUG	2777
H. 48. 6.	REQUIREMENTS	2778
H. 48. 7.	LIMITATIONS	2778
H. 49.	ltree	2778
H. 49. 1.	定义	2778
H. 49. 2.	操作符和函数	2779
H. 49. 3.	索引	2781
H. 49. 4.	例子	2782
H. 49. 5.	作者	2784
H. 50.	ltrowlocks	2784
H. 50. 1.	概述	2784
H. 50. 2.	示例输出	2785
H. 50. 3.	作者	2785
H. 51.	ltstattuple	2785
H. 51. 1.	函数	2785
H. 51. 2.	作者	2789
H. 52.	lt_walminer	2789
H. 52. 1.	什么是lt_walminer?	2789
H. 53.	myfce	2789
H. 53. 1.	与Mysql数据库兼容的功能	2790
H. 53. 2.	SQL函数参考	2793
H. 53. 3.	查询	2844
H. 54.	mysql_fdw	2844
H. 54. 1.	Installation	2844
H. 54. 2.	Enhancements	2845
H. 54. 3.	Usage	2845
H. 54. 4.	Examples	2846
H. 55.	oracle_fdw	2847
H. 55. 1.	Cookbook	2847
H. 55. 2.	Objects created by the extension	2848

---



H. 55.3.	选项	2849
H. 55.4.	用法	2851
H. 55.5.	内部结构	2856
H. 55.6.	存在的问题	2856
H. 56.	orafce	2857
H. 56.1.	与 Oracle 数据库兼容的特性	2857
H. 56.2.	使用orafce的注意事项	2867
H. 56.3.	数据类型	2868
H. 56.4.	查询	2875
H. 56.5.	SQL 函数参考	2875
H. 56.6.	包参考	2947
H. 56.7.	事务行为	3004
H. 56.8.	系统视图参考	3005
H. 57.	pageinspect	3026
H. 57.1.	通用函数	3026
H. 57.2.	Heap Functions	3027
H. 57.3.	B树函数	3028
H. 57.4.	GIN函数	3031
H. 57.5.	Hash函数	3032
H. 58.	passwordcheck	3033
H. 59.	postgres_fdw	3034
H. 59.1.	postgres_fdw 的 FDW 选项	3034
H. 59.2.	连接管理	3037
H. 59.3.	事务管理	3037
H. 59.4.	远程查询优化	3037
H. 59.5.	远程查询执行环境	3038
H. 59.6.	跨版本兼容性	3038
H. 59.7.	例子	3038
H. 59.8.	作者	3039
H. 60.	pldebugger	3039
H. 60.1.	安装	3039
H. 60.2.	用法	3039
H. 60.3.	架构	3039
H. 61.	rum	3040
H. 61.1.	介绍	3040
H. 61.2.	常用操作符和函数	3040
H. 61.3.	操作符类	3040
H. 62.	seg	3044
H. 62.1.	原理	3044
H. 62.2.	语法	3045
H. 62.3.	精度	3045
H. 62.4.	用法	3045
H. 62.5.	注解	3046
H. 62.6.	开发人员	3046
H. 63.	sepgsql	3046
H. 63.1.	概述	3047
H. 63.2.	安装	3047
H. 63.3.	回归测试	3048
H. 63.4.	GUC 参数	3049
H. 63.5.	特性	3049
H. 63.6.	Sepgsql 函数	3052
H. 63.7.	限制	3053
H. 63.8.	外部资源	3053
H. 63.9.	作者	3053
H. 64.	spi	3054
H. 64.1.	refint — 用于实现参照完整性的函数	3054
H. 64.2.	autoinc — 用于自增域的函数	3054
H. 64.3.	insert_username — 用于跟踪谁修改了一个表的函数	3054

---

H. 64. 4. moddatetime — 用于跟踪上一次修改时间的函数 .....	3054
H. 65. sslinfo .....	3055
H. 65. 1. 提供的函数 .....	3055
H. 65. 2. 作者 .....	3056
H. 66. system_stats .....	3056
H. 66. 1. 安全性 .....	3056
H. 66. 2. 函数 .....	3057
H. 66. 3. 每个函数的详细输出 .....	3057
H. 67. tablefunc .....	3059
H. 67. 1. 所提供的函数 .....	3060
H. 67. 2. 作者 .....	3068
H. 68. tcn .....	3068
H. 69. test_decoding .....	3069
H. 70. tsm_system_rows .....	3070
H. 70. 1. 示例 .....	3070
H. 71. tsm_system_time .....	3070
H. 71. 1. 示例 .....	3071
H. 72. tsvector2 .....	3071
H. 72. 1. tsvector2特定函数 .....	3071
H. 72. 2. 可安全用于两种类型的常见函数 .....	3071
H. 73. unaccent .....	3072
H. 73. 1. 配置 .....	3072
H. 73. 2. 用法 .....	3072
H. 73. 3. 函数 .....	3073
H. 74. uuid-ossip .....	3073
H. 74. 1. uuid-ossip 函数 .....	3074
H. 74. 2. 作者 .....	3075
H. 75. wal2sql .....	3075
H. 75. 1. 配置 .....	3075
H. 75. 2. 配置参数 .....	3075
H. 75. 3. 参数示例 .....	3077
H. 75. 4. 注意 .....	3078
H. 76. xml2 .....	3078
H. 76. 1. 废弃公告 .....	3079
H. 76. 2. 函数的描述 .....	3079
H. 76. 3. xpath_table .....	3079
H. 76. 4. XSLT 函数 .....	3082
H. 76. 5. 作者 .....	3082
H. 77. zhparser .....	3082
H. 77. 1. 配置 .....	3082
H. 77. 2. 示例 .....	3083
H. 77. 3. 自定义字典 .....	3084
H. 77. 4. 自定义字典 2.1 .....	3084
H. 78. lt_t3sdk .....	3085
H. 78. 1. 安装 .....	3085
H. 78. 2. 概述 .....	3085
H. 78. 3. 函数 .....	3086
H. 78. 4. 例子 .....	3086
H. 79. lt_http .....	3087
H. 79. 1. 安装 .....	3087
H. 79. 2. 概述 .....	3087
H. 79. 3. 函数 .....	3087
H. 79. 4. CURL选项 .....	3088
H. 79. 5. 保持连接和超时 .....	3089
H. 79. 6. 例子 .....	3090
I. 额外提供的程序 .....	3093
I. 1. 客户端应用 .....	3093
I. 2. 服务器应用 .....	3100

---

J.	外部项目 .....	3104
J.1.	客户端接口 .....	3104
J.2.	管理工具 .....	3104
J.3.	过程语言 .....	3104
J.4.	扩展 .....	3105
K.	LightDB限制 .....	3106
L.	首字母缩写词 .....	3107
M.	术语表 .....	3113
N.	颜色支持 .....	3124
N.1.	当使用颜色时 .....	3124
N.2.	配置颜色 .....	3124
O.	LightDB JDBC .....	3125
P.	扩展的钩子列表 .....	3126

# Appendix C. LightDB错误代码

LightDB服务器发出的所有消息都被赋予了五个字符错误代码，这遵循 SQL 标准对“SQLSTATE”代码的习惯。需要知道发生了什么错误条件的应用通常应该测试错误代码，而不是查看文本形式的错误消息。这些错误代码轻易不会在LightDB的版本之间改变，并且一般也不会随着错误消息的本地化而改变。请注意有些（但不是全部）LightDB生成的错误代码是由 SQL 标准定义的；有些标准没有定义的额外错误代码是LightDB发明的或者是从其它数据库借用的。

根据标准，错误代码的前两个字符表示错误类别，而后三个字符表示在该类别内的一种特定情况。因此，那些不能识别特定错误代码的应用仍然可以从错误类别中推断要做什么。

Table C. 中列出了LightDB 13.8-23.4定义的所有错误代码（有些实际上目前并没有使用，但是 SQL 标准中有定义）。错误类别也在其中列出。对于每个错误类别都有一个“标准”错误代码，它的最后三个字符是000。这个代码只用于那些属于该类别但是没有被赋予更特定代码的错误情况。

“情况名称”列中显示的符号是在PL/pgSQL中使用的情况名称。情况名称可以被写成大写或小写形式（注意PL/pgSQL不识别警告（与错误不同）情况名称，它们是类别 00、01 和 02）。

对于某些类型的错误，服务器会报告与错误相关的数据库对象（一个表、表列、数据类型或约束）的名称。例如，导致一个unique\_violation错误的唯一约束的名称。这些名字被包含在错误报告消息的独立域中，这样应用就不必从可能是本地化的人类可读的消息文本中抽取它们。到LightDB 9.3 位置，对于这个特性的完全覆盖只在 SQLSTATE 类别 23（完整性约束未被）的错误中存在，但是很可能会在未来进行扩展。

Table C.1. LightDB错误代码

错误代码	情况名称
Class 00 — Successful Completion	
00000	successful_completion
Class 01 — Warning	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
Class 02 — No Data (this is also a warning class per the SQL standard)	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
Class 03 — SQL Statement Not Yet Complete	
03000	sql_statement_not_yet_complete
Class 08 — Connection Exception	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure

错误代码	情况名称
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
Class 09 — Triggered Action Exception	
09000	triggered_action_exception
Class 0A — Feature Not Supported	
0A000	feature_not_supported
Class 0B — Invalid Transaction Initiation	
0B000	invalid_transaction_initiation
Class 0F — Locator Exception	
0F000	locator_exception
0F001	invalid_locator_specification
Class 0L — Invalid Grantor	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Class 0P — Invalid Role Specification	
0P000	invalid_role_specification
Class 0Z — Diagnostics Exception	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Class 20 — Case Not Found	
20000	case_not_found
Class 21 — Cardinality Violation	
21000	cardinality_violation
Class 22 — Data Exception	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast

错误代码	情况名称
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
22013	invalid_preceding_or_following_size
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
22030	duplicate_json_object_key_value
22031	invalid_argument_for_sql_json_datetime_function
22032	invalid_json_text
22033	invalid_sql_json_subscript
22034	more_than_one_sql_json_item

错误代码	情况名称
22035	no_sql_json_item
22036	non_numeric_sql_json_item
22037	non_unique_keys_in_a_json_object
22038	singleton_sql_json_item_required
22039	sql_json_array_not_found
2203A	sql_json_member_not_found
2203B	sql_json_number_not_found
2203C	sql_json_object_not_found
2203D	too_many_json_array_elements
2203E	too_many_json_object_members
2203F	sql_json_scalar_required
Class 23 — Integrity Constraint Violation	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24 — Invalid Cursor State	
24000	invalid_cursor_state
Class 25 — Invalid Transaction State	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
Class 29 — Invalid or Unexpected Status	
29000	invalid_status
29003	assert_failed
29P01	cache_lookup_failed
29P02	fetch_data_failed
29P03	flush_data_size_mismatch
29P04	relation_open_error

错误代码	情况名称
29P05	relation_close_error
29P06	invalid_cache_plan
Class 26 — Invalid SQL Statement Name	
26000	invalid_sql_statement_name
Class 27 — Triggered Data Change Violation	
27000	triggered_data_change_violation
Class 28 — Invalid Authorization Specification	
28000	invalid_authorization_specification
28P01	invalid_password
Class 2B — Dependent Privilege Descriptors Still Exist	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Class 2D — Invalid Transaction Termination	
2D000	invalid_transaction_termination
Class 2F — SQL Routine Exception	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34 — Invalid Cursor Name	
34000	invalid_cursor_name
Class 38 — External Routine Exception	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39 — External Routine Invocation Exception	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Class 3B — Savepoint Exception	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Class 3D — Invalid Catalog Name	
3D000	invalid_catalog_name
Class 3F — Invalid Schema Name	



错误代码	情况名称
3F000	invalid_schema_name
Class 40 — Transaction Rollback	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class 42 — Syntax Error or Access Rule Violation	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function

错误代码	情况名称
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Class 44 — WITH CHECK OPTION Violation	
44000	with_check_option_violation
Class 53 — Insufficient Resources	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Class 54 — Program Limit Exceeded	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Class 55 — Object Not In Prerequisite State	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
55P04	unsafe_new_enum_value_usage
Class 57 — Operator Intervention	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
Class 58 — System Error (errors external to PostgreSQL itself)	
58000	system_error
58030	io_error
58P01	undefined_file

错误代码	情况名称
58P02	duplicate_file
Class 72 — Snapshot Failure	
72000	snapshot_too_old
Class F0 — Configuration File Error	
F0000	config_file_error
F0001	lock_file_exists
Class HV — Foreign Data Wrapper Error (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
P0005	forall_need_dml

错误代码	情况名称
Class XX — Internal Error	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted
Class LT — DDL Standard Check Failure	
LT000	keyword_not_allowed
LT001	bad_datatype
LT002	must_have_pk
LT003	must_have_gmt
LT004	no_more_100cols
LT005	invalid_procedure_definition
LT006	no_paging
LT007	high_risk_ddl
LT008	hint_need_index
LT009	index_invalid_in_hint
LT010	multi_index_in_hint
LT011	hint_disallowed
22012	zero_divide
24000	invalid_cursor
28P01	login_denied
23505	dup_val_on_index

---

## Appendix D. 日期/时间支持

LightDB使用一个内部的启发式解析器来进行所有日期/时间输入的支持。日期和时间被作为字符串输入，并且被分解为不同的域，这些域都已经预定义好了要存放哪一类信息。每个域会被解释并且被分配一个数字值或者被忽略、拒绝。该解析器为所有的文本形式的域都包含了内部查找表，包括月份、一周中的日和时区。

这个附录包括这些查找表内容的信息并且描述了解析器解码日期和时间所用的步骤。

### D. 1. 日期/时间输入解释

日期/时间类型输入使用下面的过程解码。

1. 将输入字符串打断成记号并且把每一个记号分类为一个字符串、时间、时区或数字。
  - a. 如果数字记号包含一个分号 (:), 那么这是一个时间字符串。包括所有后续数位和分号。
  - b. 如果数字记号包含一个连字符 (-)、斜线 (/) 或两个以上的句点 (.), 那么这是一个日期字符串, 它可能含有一个文本形式的月份。如果一个日期记号已经被看到, 它会转而被解释为一个时区名 (例如America/New\_York)。
  - c. 如果记号只是数字, 那么它要么是一个单一域, 要么是一个 ISO 8601 串连日期 (例如 1999年1月13日是19990113) 或时间 (例如 14:15:16 是141516)。
  - d. 如果记号以一个加号 (+) 或减号 (-) 开始, 那么它要么是一个数字的时区, 要么是一个特殊域。
2. 如果记号是一个字母字符串, 使之匹配可能的串:
  - a. 查看记号是否与任何已知的时区缩写匹配。这些缩写由在Section D. 中描述的配置文件提供。
  - b. 如果没有找到, 搜索一个内部表来查找将该记号是否匹配为一个特殊串 (例如today)、日 (例如Thursday)、月 (例如January) 或噪音词 (例如at、on)。
  - c. 如果仍然没有找到, 则抛出一个错误。
3. 当记号是一个数字或数字域时:
  - a. 如果有 8 位或 6 位, 并且之前没有读到其他日期域, 那么解释为一个“串连日期” (例如19990118或990118)。翻译是YYYYMMDD或YYMMDD。
  - b. 如果记号是 3 位并且已经读到了一个年域, 那么解释为一年中的第几日。
  - c. 如果是 4 位或 6 位并且已经读到了一个年域, 那么解释为一个时间域 (HHMM或HHMMSS)。
  - d. 如果是 3 位或更多位并且还没有读到日期域, 解释为一个年域 (这会强制剩余日期域的 yy-mm-dd 顺序)。
  - e. 否则日期域顺序被假定为遵循DateStyle设置: mm-dd-yy、dd-mm-yy 或 yy-mm-dd。如果一个月或日域被发现超过范围, 则抛出一个错误。
4. 如果已经指定了 BC, 对年求反并且加一用于内部存储 (在格里高利历中没有 0 年, 因此数字的 1 BC 就是 0 年)。
5. 如果没有指定 BC, 并且如果该年域长度为两位, 那么把该年域调整为四位。如果该域小于 70, 则增加 2000, 否则增加 1900。

## Tip

公历年 AD 1 - 99 可以使用带有前导零的 4 位形式录入（例如0099是 AD 99）。

## D. 2. 处理无效或不明确的时间戳

通常，如果日期/时间字符串在语法上有效但包含超出范围的字段值，将引发错误。例如，输入2月31日将被拒绝。

在夏令时转换期间，看似有效的的时间戳字符串可能表示不存在或不明确的时间戳。这样的输入不会被拒绝；不确定性可以通过要应用哪个UTC偏移来解决。例如，假设TimeZone参数设置为America/New\_York，请考虑

```
=> SELECT '2018-03-11 02:30'::timestampz;
       timestampz
```

```
-----
2018-03-11 03:30:00-04
(1 row)
```

因为那天是那个时区的春天过渡日期，所以没有民用时间凌晨2:30；时钟从2AM EST 跳转到3AM EDT。LightDB将给定时间解释为标准时间（UTC-5），然后呈现为3:30AM EDT（UTC-4）。

相反，请考虑后向过渡期间的行为：

```
=> SELECT '2018-11-04 02:30'::timestampz;
       timestampz
```

```
-----
2018-11-04 02:30:00-05
(1 row)
```

在那一天，上午2:30有两种可能的解释；2:30AM EDT，以及一小时以后，如果转换到标准时间，即2:30AM EST。同样，LightDB将给定时间解释为标准时间（UTC-5）。我们可以通过指定夏令时来强行控制：

```
=> SELECT '2018-11-04 02:30 EDT'::timestampz;
       timestampz
```

```
-----
2018-11-04 01:30:00-05
(1 row)
```

此时间戳可以有效地呈现为2:30 UTC-4或1:30 UTC-5；时间戳输出代码选择后者。

在这些情况下应用的精确规则是，出现在前向跳转的夏令时转换中的无效时间戳被分配了转换之前的时区对应的UTC偏移；可能落在后向跳转的两边的不确定的时间戳被分配了转换之后的时区对应的UTC偏移。在大多数时区，这相当于说“在有疑问时，标准时间解释是首选”。

在所有情况下，可以显式使用数字UTC偏移或对应于固定UTC偏移的时区缩写明确指定与时间戳关联的UTC偏移。刚刚给出的规则仅在需要推断偏移量变化的时区的UTC偏移时才适用。

## D. 3. 日期/时间关键词

Table D. 展示了被识别为月份名称的记号。

Table D.1. 月份名称

月份	简写
一月	Jan
二月	Feb
三月	Mar
四月	Apr
五月	
六月	Jun
七月	Jul
八月	Aug
九月	Sep, Sept
十月	Oct
十一月	Nov
十二月	Dec

Table D. 展示了被识别为一周内每一天的名称的记号。

Table D.2. 一周内每一天的名称

天	简写
周日	Sun
周一	Mon
周二	Tue, Tues
周三	Wed, Weds
周四	Thu, Thur, Thurs
周五	Fri
周六	Sat

Table D. 展示了服务于多种修饰目的的记号。

Table D.3. 日期/时间域修饰语

标识符	描述
AM	12:00 之前的时间
AT	被忽略
JULIAN, JD, J	下一个域是儒略日期
ON	被忽略
PM	12:00 之后的时间
T	下一个域是时间

## D. 4. 日期/时间配置文件

因为时区缩写并未被很好地标准化，LightDB提供了一种方法来自定义服务器所接受的缩写集合。timezone\_abbreviations运行时参数决定活动的缩写集合。虽然这个参数可以被任何数据库用户修改，但它的可能值是受到数据库管理员的控制的——它们实际上是存储在安装目录的.../share/timezonesets/子目录中的一些配置文件。通过在那个目录中增加或修改文件，管理员可以为时区缩写设定本地策略。

timezone\_abbreviations可以被设置为任何在.../share/timezonesets/中可以找到的文件名，前提该文件的名字完全是字母的（timezone\_abbreviations中禁止非字母字符防止从预期目录的外面读取文件以及读取编辑器的备份文件和其他外部文件）。

一个时区缩写文件可以包含空行和以#开始的注释。非注释行必须具有下列格式之一：

```
zone_abbreviation offset
zone_abbreviation offset D
zone_abbreviation time_zone_name
@INCLUDE file_name
@OVERRIDE
```

一个zone\_abbreviation就是被定义的缩写。offset是一个整数，它给出以秒计的到 UTC 的等效偏移量，为正表示东起格林威治，为负表示西起格林威治。例如，-18000 表示格林威治西边的五个小时，或者北美东海岸标准时间。D指示该区域名表示本地夏令时而非标准时间。另外，还可以给出一个time\_zone\_name，在这种情况下会查阅该时区定义，并且会使用该时区中的缩写含义。这种替代方案只用于那些含义在历史上有变化的缩写，因为比起使用一个固定的整数值，查找该含义开销要大得多。

另外，还可以给出一个time\_zone\_name，它引用 IANA 时区数据库中定义的时区名。这时会参考该时区的定义来判断在时区中是否有或者使用了该缩写。如果是，会使用适当的含义——也就是正在判断其值的时间戳中当前使用的含义，或者之前刚刚使用的含义（如果当时不是当前），或者最老的含义（如果只在那时之后用过）。这种行为对于处理其含义在历史上变化过的缩写是至关重要的。也允许按照缩写没有出现在其中的时区名来定义缩写，这样使用该缩写就等效于直接写出该时区名。

### Tip

在定义其 UTC 偏移没有改变过的缩写时，使用简单的整数offset更好，这样的缩写在处理时代价比那些需要查阅时区定义的缩写更低。

@INCLUDE语法允许包括.../share/timezonesets/目录中的其它文件。允许进行嵌套包括，但是嵌套深度有限制。

@OVERRIDE语法表示文件中后续项可以覆盖前面的项（典型的：从被包括的文件中得到的项）。如果没有它，同一个时区缩写的相互冲突的定义会被认为是一种错误。

在一个未被修改的安装中，文件Default包含用于世界大部分地区的非冲突时区缩写。附加文件Australia和India被提供给那些地区：这些文件首先会包括Default文件，并且接着根据需要增加或修改缩写。

为了便于参考，标准安装也包含了Africa.txt、America.txt 等文件，它们包含了所有根据IANA 时区数据库中已知正在使用的时区缩写信息。如果需要，这些文件中的时区名定义可以复制并粘贴到自定义的配置文件中。注意这些文件名不能直接被timezone\_abbreviations设置引用，因为它们的名称中嵌有句点。

### Note

如果在读取时区缩写集时发生错误，将不会应用任何新值并且保留旧的集合。如果该错误是在数据库启动时发生，那么启动将失败。



## Caution

配置文件中定义的时区缩写将会覆盖LightDB中内建的非时区含义。例如Australia配置文件定义了SAT（南澳洲标准时间）。当该文件为活动时，SAT将不会被识别为周六的缩写。

## Caution

如果你修改../share/timezonesets/中的文件，那么你必须自己创建备份 — 因为通常的数据库转储不会包括这个目录。

## D. 5. POSIX 时区规范

LightDB可以接受根据POSIX标准的针对TZ环境变量的规则编写的时区规范。 POSIX时区规范不足以处理真实世界时区历史的复杂性，但有时使用它们是有原因的。

POSIX时区规范具有形式

```
STD offset [ DST [ dstoffset ] [ , rule ] ]
```

(为了可读性，我们在字段之间显示空格，但在实践中不应使用空格。) 这些区域包括：

- STD 是用于标准时间的区域缩写。
- offset 是区域相对UTC的标准时间偏移量。
- DST是用于夏令时的区域缩写。如果省略此字段和以下字段，则该区域使用没有夏令时规则的固定 UTC 偏移量。
- dstoffset是相对UTC 的夏令时偏移。 此字段通常会被省略，因为它默认会比标准时间offset少一个小时，这通常是正确的。
- rule定义夏令时生效的规则，如下所述。

在此语法中，区域缩写可以是字母字符串，如 EST，或者是被尖括号包围的任意字符串，如 <UTC-05>。 请注意，此处给出的区域缩写仅用于输出，即使这样也仅用于某些时间戳输出格式。 时间戳输入中识别的区域缩写按照Section D. 解释的那样来确定。

偏移字段指定为小时（可选为分钟和秒），与 UTC 的差异。 它们包括格式hh[:mm[:ss]] 可选带有前导符号 (+ 或 -)。 正符号用于格林威治west区域。（请注意，这与在LightDB中其他地方使用的ISO-8601符号约定相反。） hh可以有1或2位数字； mm 和 ss (如果使用)必须包含两位数字。

夏令时转换rule的格式为

```
dstdate [ / dsttime ] , stddate [ / stdtime ]
```

(像前面一样，在实际使用中不应该包含空格。) 在夏令时开始时定义dstdate和 dsttime 字段， 在标准时间开始时定义stddate和 stdtime 定义。（在某些情况下，特别是在赤道以南地区，前者可能比后者在一年内更晚。） 日期字段具有如下格式之一：

n

普通整数表示一年中的哪一天，从0到364，或者在闰年到365。

Jn

在这种格式下，n计数从1到365，即使2月29日存在，也不被计算。（因此，不能以这种方式指定发生在2月29日的转换。然而，无论是否是闰年，2月之后的天数都是相同的，所以对于固定日期的转换，这种形式通常比纯整数形式更有用。）

Mm. n. d

此格式指定始终在同一月和一一周的同一天发生的转换。m表示月，从1到12。n指定第n个按d标识的星期日期(weekday)。n为1和4之间的数字，或5表示该月中此星期日期(weekday)的最后一次出现（可能是第四或第五个）。d为0和6之间的数字，0表示星期日。例如，M3.2.0意味着“3月的第二个星期日”。

### Note

M格式足以描述许多常见的夏令时转换定律。但请注意，这些变体都无法处理夏令时法律的更改，因此在实践中，为命名时区（IANA 时区数据库中）存储的历史数据对于正确解释过去的时间戳是必要的。

转换规则中的时间字段的格式与前面描述的偏移字段相同，除了它们不能包含符号。它们定义对另一个时间发生更改的当前本地时间。如果省略，则默认为 02:00:00。

如果给定了夏令时缩写，但省略了转换rule字段，回退的行为是使用规则 M3.2.0, M11.1.0，这对应于美国截至 2020 年的做法。（也就是说，春天在3月的第二个星期日向前，秋天在11月的第一个星期日向后，两个转换都发生在凌晨2点）。请注意，此规则没有给出 2007 年之前的正确的美国转换日期。

例如，CET-1CEST, M3.5.0, M10.5.0/3描述了巴黎的当前（截至2020年）的计时做法。该规范说明标准时间缩写为 CET，并且比 UTC 的早(东)一小时；夏令时缩写为CEST，并且隐式比 UTC 早两个小时；夏令时从 CET 3 月的最后一个星期日开始，到CEST 10 月的最后一个星期日凌晨 3 点结束。

四个时区名称 EST5EDT, CST6CDT, MST7MDT和PST8PDT看起来像是POSIX区域规范。但是，它们实际上被用来命名时区，因为（出于历史原因）IANA 时区数据库中存在这些名称的文件。这样做的实际含义是，这些区域名称将产生有效的历史上的美国夏令时转换，即使普通POSIX规范不会产生。

需要警惕的是很容易拼错 POSIX 样式的时区规范，因为没有检查区域缩写的合理性。例如，SET TIMEZONE TO FOOBAR0将工作，让系统有效地使用UTC的一个相当奇特的缩写。

## D.6. 单位的历史

SQL 标准说到“在一个‘日期时间文字’的定义中，‘日期时间值’根据格里高利历被日期和时间的自然规则所约束”。LightDB遵循 SQL 标准，导致只在格里高利历内计算日期，即使对于该历法开始使用之前的日期也是如此。这个规则被称作外推格里高利历。

儒略日期是由 Julius Caesar 在公元前 45 年引入的。直到 1582 年开始转为使用公历之前，西方世界一直使用儒略日期。在儒略日期中，一回归年近似等于  $365 + 1/4$  天 = 365.25 天。它大约在128年中会出现 1 天的误差。

不断积累的历法错误促使教皇格里高利十三世按照特伦托会议的指示改革了历法。在格里高利历中，一回归年近似为  $365 + 97/400$  天 = 365.2425 天。因此对于格里高利历，大约要 3300 年一回归年才会积累一天的误差。

近似值  $365+97/400$  是通过利用下面的规则，并规定每 400 年有 97 个闰年实现的：

每个可被 4 整除的年是一个闰年。

不过，可被 100 整除的年份不是闰年。  
但是，可以被 400 整除的年份还是闰年。

因此，1700、1800、1900、2100 和 2200 都不是闰年。而 1600、2000、2400 是闰年。相比之下，旧式的儒略历法里面只有能被 4 整除的年份是闰年。相反，在旧的儒略历法中所有能被 4 整除的年份都是闰年。

罗马教皇在 1582 年 2 月宣布从 1582 年的 10 月中减除 10 天，这样 10 月 15 日就紧跟在 10 月 4 日的后面。意大利、波兰、葡萄牙和西班牙遵守了这个要求。其他天主教的国家也紧跟它们的步伐。但新教国家拒绝改变，而希腊东正教国家却一直拖延到 20 世纪开始时才逐渐遵守这个规定。大英帝国及其殖民地（包含今天的美国）在 1752 年开始遵守这项改革。因此 1752 年 9 月 2 日之后紧跟着 14 日。这就是为什么 Unix 系统上的 `cal` 程序会产生如下输出的原因：

```
$ cal 9 1752
September 1752
S M Tu W Th F S
      1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

不过，这种历法只对大英帝国及其殖民地有效，对其他地方无效。因为尝试在多个地方多个时刻使用的实际历法很难并且也很让人困惑，LightDB并没有做这种尝试，而是对所有日期遵循格里高利历规则，即使该方法在历史上是不精确的。

在世界的多个地方，发展了不同的历法，很多比格里高利系统还早。例如，中国历法的开端可以追溯到公元前 14 世纪。传说黄帝在公元前 2637 年就发明了这个历法。中华人民共和国把格里高利历作为民用。中国历法则被用于决定节日/节气。

## D. 7. 儒略日期

儒略日期系统是一种计算天数的方法。它虽然在命名上很相似，但是它和儒略历法无关。儒略日期系统是法国学者 Joseph Justus Scaliger (1540 - 1609)（可能是取自其父亲的名字，即意大利学者 Julius Caesar Scaliger (1484 - 1558) 发明的。

在儒略日期系统中，每天都有一个序数，从 JD 0 开始（有时被叫做那个儒略日期）。JD 0 在儒略历法中对应公元前 4713 年 1 月 1 日，或者在格里高利历中对应公元前 4714 年 12 月 24 日。儒略日期计数经常被天文学家用来标注夜间观测，并且因此一个日期就是从正午 UTC 到下一个正午 UTC，而不是从午夜到另一个午夜：JD 0 设计的 24 小时是从公元前 4714 年 12 月 24 日的正午 UTC 到公元前 4714 年 12 月 25 日的正午 UTC。

尽管 LightDB 在输入输出日期时支持儒略日期符号（并且也用在一些内部的日期时间计算上使用儒略日期），它不遵守从正午到正午。LightDB 把儒略日期当作是从当地午夜到当地午夜，如同一个普通日期。

然而，这个定义确实提供了一种在需要时获得天文定义的方法：在时区 UTC+12 里运算。例如，

```
=> SELECT extract(julian from '2021-06-23 7:00:00-04'::timestamp_tz at time zone
'UTC+12');
      date_part
-----
2459388.9583333335
(1 row)
=> SELECT extract(julian from '2021-06-23 8:00:00-04'::timestamp_tz at time zone
'UTC+12');
      date_part
```

```
-----  
      2459389  
(1 row)  
=> SELECT extract(julian from date '2021-06-23');  
      date_part  
-----  
      2459389  
(1 row)
```

## Appendix E. SQL关键词

Table E. 列出了在SQL标准以及LightDB 13.8-23.4中作为关键词的所有记号。背景资料可以在Section 4.1. 中找到（由于篇幅的缘故，只包括了SQL标准的最近两个版本以及用于与历史比较的SQL-92。这些版本以及其他中间标准的版本之间的差别很小）。

SQL区分保留关键词和非保留关键词。根据标准，保留关键词才是真正的关键词，它们绝不会被允许作为标识符。非关键词仅仅是在特定上下文中具有特殊的含义并且可以在其他上下文中被用作标识符。大部分非保留关键词实际上是SQL指定的内建表和内建函数的名字。非保留关键词的概念存在的意义上实际上是声明某些上下文中的一个词被附加了某种预定义的含义。

在LightDB的解析器中，情况更加复杂。其中有多种不同的记号分类，从那些决不能被用作标识符的记号到那些在解析器中绝对没有特殊状态，但被认为是普通的标识符。（后者通常是SQL中指定的函数）。在LightDB中甚至保留关键词也不是完全被保留的，而是可以被用作列标签（例如可以写SELECT 55 AS CHECK，虽然CHECK是一个保留关键词）。

在Table E. 的LightDB列中，我们把解析器明确知道但允许作为列名或者表名的那些关键词分类为“非保留”。有一些关键词是非保留的，但是不能被用作函数或数据类型名称，因此它们会被标记（大部分这些词表示有特殊语法的内建函数或数据类型。这种函数或类型仍然可用，但是不能被用户重新定义）。不允许作为列名或表名的记号被打上“保留”的标签。某些保留关键词被允许作为函数或数据类型的名字，这也显示在该表中。如果没有被那样标记，保留关键词仅被允许作为列标签。这一列的空白项意味着这个词被LightDB当作一个普通的标识符。

此外，虽然大多数关键词可以用作“bare”列标签，而不用在它们前面写AS（如Section 7.3. 中描述的那样），但有一些需要在前面加上AS以避免歧义。这些在表中被标记为“requires AS”。

一般来说，如果你在使用任何所列出关键词作为标识符的命令，遇到虚假的解析器错误，应该尝试将该标识符加上引号来看看是否能解决问题。

在学习Table E. 之前有一件重要的事情是理解一个在LightDB中不被保留的关键词并不意味着与该词相关的特性没有被实现。反过来，一个关键词的存在也不表示相应特性的存在。

Table E.1. SQL Key Words

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
A			non-reserved	non-reserved	
ABORT	non-reserved				
ABS			reserved	reserved	
ABSENT			non-reserved	non-reserved	
ABSOLUTE	non-reserved		non-reserved	non-reserved	reserved
ACCESS	non-reserved				
ACCORDING			non-reserved	non-reserved	
ACOS			reserved		
ACTION	non-reserved		non-reserved	non-reserved	reserved
ACTIVE		non-reserved			

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
ADA			non-reserved	non-reserved	non-reserved
ADD	non-reserved		non-reserved	non-reserved	reserved
ADMIN	non-reserved		non-reserved	non-reserved	
AFTER	non-reserved		non-reserved	non-reserved	
AGGREGATE	non-reserved				
ALL	reserved, requires AS		reserved	reserved	reserved
ALLOCATE			reserved	reserved	reserved
ALSO	non-reserved				
ALTER	non-reserved		reserved	reserved	reserved
ALWAYS	non-reserved		non-reserved	non-reserved	
ANALYSE	reserved				
ANALYZE	reserved				
AND	reserved		reserved	reserved	reserved
ANY	reserved, requires AS		reserved	reserved	reserved
APPEND		non-reserved			
ARE			reserved	reserved	reserved
ARRAY	reserved, requires AS		reserved	reserved	
ARRAY_AGG			reserved	reserved	
ARRAY_MAX_CARDINALITY			reserved	reserved	
AS	reserved, requires AS		reserved	reserved	reserved
ASC	reserved		non-reserved	non-reserved	reserved
ASENSIIVE					
ASENSITIVE	non-reserved		reserved	reserved	
ASIN			reserved		
ASSERTION	non-reserved		non-reserved	non-reserved	reserved
ASSIGNMENT	non-reserved		non-reserved	non-reserved	
ASYMMETRIC	reserved		reserved	reserved	
AT	non-reserved		reserved	reserved	reserved
ATAN			reserved		
ATOMIC			reserved	reserved	
ATTACH	non-reserved				

## SQL关键词

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
ATTRIBUTE	non-reserved		non-reserved	non-reserved	
ATTRIBUTES			non-reserved	non-reserved	
AUTHID		non-reserved			
AUTHORIZATION	reserved (can be function or type)		reserved	reserved	reserved
AUTO_INCREMENT		non-reserved			
AVG			reserved	reserved	reserved
BACKWARD	non-reserved				
BASE64			non-reserved	non-reserved	
BEFORE	non-reserved		non-reserved	non-reserved	
BEGIN	non-reserved		reserved	reserved	reserved
BEGIN_FRAME			reserved	reserved	
BEGIN_PARTITION			reserved	reserved	
BERNOULLI			non-reserved	non-reserved	
BETWEEN	non-reserved (cannot be function or type)		reserved	reserved	reserved
BIGINT	non-reserved (cannot be function or type)		reserved	reserved	
BIGSERIAL		non-reserved			
BINARY	reserved (can be function or type)		reserved	reserved	
BIT	non-reserved (cannot be function or type)				reserved
BIT_LENGTH					reserved
BLOB			reserved	reserved	
BLOCKED			non-reserved	non-reserved	
BODY		non-reserved			
BOM			non-reserved	non-reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
BOOLEAN	non-reserved (cannot be function or type)		reserved	reserved	
BOTH	reserved		reserved	reserved	reserved
BREADTH			non- reserved	non- reserved	
BY	non-reserved		reserved	reserved	reserved
BYTE		non- reserved			
C			non- reserved	non- reserved	non- reserved
CACHE	non-reserved				
CALL	non-reserved		reserved	reserved	
CALLED	non-reserved		reserved	reserved	
CARDINALITY			reserved	reserved	
CASCADE	non-reserved		non- reserved	non- reserved	reserved
CASCADED	non-reserved		reserved	reserved	reserved
CASE	reserved		reserved	reserved	reserved
CAST	reserved		reserved	reserved	reserved
CATALOG	non-reserved		non- reserved	non- reserved	reserved
CATALOG_NAME			non- reserved	non- reserved	non- reserved
CEIL			reserved	reserved	
CEILING			reserved	reserved	
CHAIN	non-reserved		non- reserved	non- reserved	
CHAINING			non- reserved		
CHAR	non-reserved (cannot be function or type), requires AS		reserved	reserved	reserved
CHARACTER	non-reserved (cannot be function or type), requires AS		reserved	reserved	reserved
CHARACTERISTICS	non-reserved		non- reserved	non- reserved	
CHARACTERS			non- reserved	non- reserved	
CHARACTER_LENGTH			reserved	reserved	reserved



Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
CHARACTER_SET_CATALOG			non-reserved	non-reserved	non-reserved
CHARACTER_SET_NAME			non-reserved	non-reserved	non-reserved
CHARACTER_SET_SCHEMA			non-reserved	non-reserved	non-reserved
CHARSET		non-reserved			
CHAR_LENGTH			reserved	reserved	reserved
CHECK	reserved		reserved	reserved	reserved
CHECKPOINT	non-reserved				
CLASS	non-reserved				
CLASSIFIER			reserved		
CLASS_ORIGIN			non-reserved	non-reserved	non-reserved
CLOB			reserved	reserved	
CLOSE	non-reserved		reserved	reserved	reserved
CLUSTER	non-reserved				
COALESCE	non-reserved (cannot be function or type)		reserved	reserved	reserved
COBOL			non-reserved	non-reserved	non-reserved
COLLATE	reserved		reserved	reserved	reserved
COLLATION	reserved (can be function or type)		non-reserved	non-reserved	reserved
COLLATION_CATALOG			non-reserved	non-reserved	non-reserved
COLLATION_NAME			non-reserved	non-reserved	non-reserved
COLLATION_SCHEMA			non-reserved	non-reserved	non-reserved
COLLECT			reserved	reserved	
COLOCATE_WITH		non-reserved			
COLUMN	reserved		reserved	reserved	reserved
COLUMNS	non-reserved		non-reserved	non-reserved	
COLUMN_NAME			non-reserved	non-reserved	non-reserved
COMMAND_FUNCTION			non-reserved	non-reserved	non-reserved
COMMAND_FUNCTION_CODE			non-reserved	non-reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
COMMENT	non-reserved				
COMMENTS	non-reserved				
COMMIT	non-reserved		reserved	reserved	reserved
COMMITTED	non-reserved		non-reserved	non-reserved	non-reserved
COMPRESS		non-reserved			
CONCURRENCY		non-reserved			
CONCURRENTLY	reserved (can be function or type)				
CONDITION			reserved	reserved	
CONDITIONAL			non-reserved		
CONDITION_NUMBER			non-reserved	non-reserved	non-reserved
CONFIGURATION	non-reserved				
CONFLICT	non-reserved				
CONNECT		non-reserved	reserved	reserved	reserved
CONNECTION	non-reserved		non-reserved	non-reserved	reserved
CONNECTION_NAME			non-reserved	non-reserved	non-reserved
CONNECT_BY_ISLEAF		non-reserved			
CONNECT_BY_ROOT		non-reserved			
CONSTRAINT	reserved		reserved	reserved	reserved
CONSTRAINTS	non-reserved		non-reserved	non-reserved	reserved
CONSTRAINT_CATALOG			non-reserved	non-reserved	non-reserved
CONSTRAINT_NAME			non-reserved	non-reserved	non-reserved
CONSTRAINT_SCHEMA			non-reserved	non-reserved	non-reserved
CONSTRUCTOR			non-reserved	non-reserved	
CONTAINS		non-reserved	reserved	reserved	
CONTENT	non-reserved		non-reserved	non-reserved	
CONTINUE	non-reserved		non-reserved	non-reserved	reserved

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
CONTROL			non-reserved	non-reserved	
CONVERSION	non-reserved				
CONVERT			reserved	reserved	reserved
COPY	non-reserved		reserved		
CORR			reserved	reserved	
CORRESPONDING			reserved	reserved	reserved
COS			reserved		
COSH			reserved		
COST	non-reserved				
COUNT			reserved	reserved	reserved
COVAR_POP			reserved	reserved	
COVAR_SAMP			reserved	reserved	
CREATE	reserved, requires AS		reserved	reserved	reserved
CROSS	reserved (can be function or type)		reserved	reserved	reserved
CSV	non-reserved				
CUBE	non-reserved		reserved	reserved	
CUME_DIST			reserved	reserved	
CURRENT	non-reserved		reserved	reserved	reserved
CURRENT_CATALOG	reserved		reserved	reserved	
CURRENT_DATE	reserved		reserved	reserved	reserved
CURRENT_DEFAULT_TRANSFORM_GROUP			reserved	reserved	
CURRENT_PATH			reserved	reserved	
CURRENT_ROLE	reserved		reserved	reserved	
CURRENT_ROW			reserved	reserved	
CURRENT_SCHEMA	reserved (can be function or type)		reserved	reserved	
CURRENT_TIME	reserved		reserved	reserved	reserved
CURRENT_TIMESTAMP	reserved		reserved	reserved	reserved
CURRENT_TRANSFORM_GROUP_FOR_TYPE			reserved	reserved	
CURRENT_USER	reserved		reserved	reserved	reserved
CURSOR	non-reserved		reserved	reserved	reserved
CURSOR_NAME			non-reserved	non-reserved	non-reserved
CYCLE	non-reserved		reserved	reserved	
DATA	non-reserved		non-reserved	non-reserved	non-reserved
DATABASE	non-reserved				

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
DATALINK			reserved	reserved	
DATE			reserved	reserved	reserved
DATETIME		non-reserved (cannot be function or type)			
DATETIME_INTERVAL_CODE			non-reserved	non-reserved	non-reserved
DATETIME_INTERVAL_PRECISION			non-reserved	non-reserved	non-reserved
DAY	non-reserved, requires AS		reserved	reserved	reserved
DB			non-reserved	non-reserved	
DEALLOCATE	non-reserved		reserved	reserved	reserved
DEC	non-reserved (cannot be function or type)		reserved	reserved	reserved
DECFLOAT			reserved		
DECIMAL	non-reserved (cannot be function or type)		reserved	reserved	reserved
DECLARE	non-reserved, requires ""		reserved	reserved	reserved
DECODE	non-reserved				
DEFAULT	reserved		reserved	reserved	reserved
DEFAULTS	non-reserved		non-reserved	non-reserved	
DEFERRABLE	reserved		non-reserved	non-reserved	reserved
DEFERRED	non-reserved		non-reserved	non-reserved	reserved
DEFINE			reserved		
DEFINED			non-reserved	non-reserved	
DEFINER	non-reserved		non-reserved	non-reserved	
DEGREE			non-reserved	non-reserved	
DELETE	non-reserved		reserved	reserved	reserved
DELIMITER	non-reserved				
DELIMITERS	non-reserved				
DENSE_RANK			reserved	reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
DEPENDS	non-reserved				
DEPTH			non-reserved	non-reserved	
DEREF			reserved	reserved	
DERIVED			non-reserved	non-reserved	
DESC	reserved		non-reserved	non-reserved	reserved
DESCRIBE			reserved	reserved	reserved
DESCRIPTOR			non-reserved	non-reserved	reserved
DETACH	non-reserved				
DETERMINISTIC			reserved	reserved	
DIAGNOSTICS			non-reserved	non-reserved	reserved
DICTIONARY	non-reserved				
DISABLE	non-reserved				
DISCARD	non-reserved				
DISCONNECT			reserved	reserved	reserved
DISPATCH			non-reserved	non-reserved	
DISTINCT	reserved		reserved	reserved	reserved
DISTRIBUTED		non-reserved			
DLNEWCOPY			reserved	reserved	
DLPREVIOUSCOPY			reserved	reserved	
DLURLCOMPLETE			reserved	reserved	
DLURLCOMPLETEONLY			reserved	reserved	
DLURLCOMPLETEWRITE			reserved	reserved	
DLURLPATH			reserved	reserved	
DLURLPATHONLY			reserved	reserved	
DLURLPATHWRITE			reserved	reserved	
DLURLSCHEME			reserved	reserved	
DLURLSERVER			reserved	reserved	
DLVALUE			reserved	reserved	
DO	reserved				
DOCUMENT	non-reserved		non-reserved	non-reserved	
DOMAIN	non-reserved		non-reserved	non-reserved	reserved
DOUBLE	non-reserved		reserved	reserved	reserved
DROP	non-reserved		reserved	reserved	reserved
DUPLICATE		non-reserved			

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
DYNAMIC			reserved	reserved	
DYNAMIC_FUNCTION			non-reserved	non-reserved	non-reserved
DYNAMIC_FUNCTION_CODE			non-reserved	non-reserved	
EACH	non-reserved		reserved	reserved	
EDITIONABLE		non-reserved			
EDITIONING		non-reserved			
ELEMENT			reserved	reserved	
ELSE	reserved		reserved	reserved	reserved
ELSIF		non-reserved			
EMPTY			reserved	non-reserved	
ENABLE	non-reserved				
ENCODING	non-reserved		non-reserved	non-reserved	
ENCRYPTED	non-reserved				
END	reserved		reserved	reserved	reserved
END-EXEC			reserved	reserved	reserved
END_FRAME			reserved	reserved	
END_PARTITION			reserved	reserved	
ENFORCED			non-reserved	non-reserved	
ENGINE		non-reserved			
ENUM	non-reserved				
EQUALS			reserved	reserved	
ERROR		non-reserved	non-reserved		
ESCAPE	non-reserved		reserved	reserved	reserved
EVENT	non-reserved				
EVERY		non-reserved	reserved	reserved	
EXCEPT	reserved, requires AS		reserved	reserved	reserved
EXCEPTION		non-reserved			reserved
EXCHANGE		non-reserved			
EXCLUDE	non-reserved		non-reserved	non-reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
EXCLUDING	non-reserved		non-reserved	non-reserved	
EXCLUSIVE	non-reserved				
EXEC			reserved	reserved	reserved
EXECUTE	non-reserved		reserved	reserved	reserved
EXISTS	non-reserved (cannot be function or type)		reserved	reserved	reserved
EXP			reserved	reserved	
EXPAND		non-reserved			
EXPLAIN	non-reserved				
EXPRESSION	non-reserved		non-reserved	non-reserved	
EXTENSION	non-reserved				
EXTERNAL	non-reserved		reserved	reserved	reserved
EXTRACT	non-reserved (cannot be function or type)		reserved	reserved	reserved
FALSE	reserved		reserved	reserved	reserved
FAMILY	non-reserved				
FETCH	reserved, requires AS		reserved	reserved	reserved
FILE			non-reserved	non-reserved	
FILTER	non-reserved, requires AS		reserved	reserved	
FINAL			non-reserved	non-reserved	
FINISH			non-reserved		
FIRST	non-reserved		non-reserved	non-reserved	reserved
FIRST_VALUE			reserved	reserved	
FLAG			non-reserved	non-reserved	
FLOAT	non-reserved (cannot be function or type)		reserved	reserved	reserved
FLOOR			reserved	reserved	
FOLLOWING	non-reserved		non-reserved	non-reserved	
FOR	reserved, requires AS		reserved	reserved	reserved

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
FORCE	non-reserved				
FOREIGN	reserved		reserved	reserved	reserved
FORMAT		non-reserved	non-reserved		
FORTRAN			non-reserved	non-reserved	non-reserved
FORWARD	non-reserved				
FOUND			non-reserved	non-reserved	reserved
FRAME_ROW			reserved	reserved	
FREE			reserved	reserved	
FREEZE	reserved (can be function or type)				
FROM	reserved, requires AS		reserved	reserved	reserved
FS			non-reserved	non-reserved	
FULFILL			non-reserved		
FULL	reserved (can be function or type)		reserved	reserved	reserved
FULLSCAN		non-reserved			
FUNCTION	non-reserved		reserved	reserved	
FUNCTIONS	non-reserved				
FUSION			reserved	reserved	
G			non-reserved	non-reserved	
GENERAL			non-reserved	non-reserved	
GENERATED	non-reserved		non-reserved	non-reserved	
GET			reserved	reserved	reserved
GLOBAL	non-reserved		reserved	reserved	reserved
GO			non-reserved	non-reserved	reserved
GOTO			non-reserved	non-reserved	reserved
GRANT	reserved, requires AS		reserved	reserved	reserved
GRANTED	non-reserved		non-reserved	non-reserved	
GREATEST	non-reserved (cannot be				



Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
	function or type)				
GROUP	reserved, requires AS		reserved	reserved	reserved
GROUPING	non-reserved (cannot be function or type)		reserved	reserved	
GROUPS	non-reserved		reserved	reserved	
GROUP_CONCAT		non-reserved			
HANDLER	non-reserved				
HASH		non-reserved			
HAVING	reserved, requires AS		reserved	reserved	reserved
HEADER	non-reserved				
HEX			non-reserved	non-reserved	
HIERARCHY			non-reserved	non-reserved	
HOLD	non-reserved		reserved	reserved	
HOURL	non-reserved, requires AS		reserved	reserved	reserved
ID			non-reserved	non-reserved	
IDENTIFIED		non-reserved			
IDENTITY	non-reserved		reserved	reserved	reserved
IF	non-reserved				
IGNORE		non-reserved	non-reserved	non-reserved	
ILIKE	reserved (can be function or type)				
IMMEDIATE	non-reserved		non-reserved	non-reserved	reserved
IMMEDIATELY			non-reserved	non-reserved	
IMMUTABLE	non-reserved				
IMPLEMENTATION			non-reserved	non-reserved	
IMPLICIT	non-reserved				
IMPORT	non-reserved		reserved	reserved	
IN	reserved		reserved	reserved	reserved
INCLUDE	non-reserved				

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
INCLUDING	non-reserved		non-reserved	non-reserved	
INCLUSIVE		non-reserved			
INCREMENT	non-reserved		non-reserved	non-reserved	
INDENT			non-reserved	non-reserved	
INDEX	non-reserved				
INDEXES	non-reserved				
INDICATOR			reserved	reserved	reserved
INHERIT	non-reserved				
INHERITS	non-reserved				
INITIAL			reserved		
INITIALLY	reserved		non-reserved	non-reserved	reserved
INITRANS		non-reserved			
INLINE	non-reserved				
INNER	reserved (can be function or type)		reserved	reserved	reserved
INNODB		non-reserved			
INOUT	non-reserved (cannot be function or type)		reserved	reserved	
INPUT	non-reserved		non-reserved	non-reserved	reserved
INSENSITIVE	non-reserved		reserved	reserved	reserved
INSERT	non-reserved		reserved	reserved	reserved
INSTANCE			non-reserved	non-reserved	
INSTANTIABLE			non-reserved	non-reserved	
INSTEAD	non-reserved		non-reserved	non-reserved	
INT	non-reserved (cannot be function or type)		reserved	reserved	reserved
INTEGER	non-reserved (cannot be function or type)		reserved	reserved	reserved

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
INTEGRITY			non-reserved	non-reserved	
INTERSECT	reserved, requires AS		reserved	reserved	reserved
INTERSECTION			reserved	reserved	
INTERVAL	non-reserved (cannot be function or type)		reserved	reserved	reserved
INTO	reserved, requires AS		reserved	reserved	reserved
INVOKER	non-reserved		non-reserved	non-reserved	
IS	reserved (can be function or type)		reserved	reserved	reserved
ISNULL	reserved (can be function or type), requires AS				
ISOLATION	non-reserved		non-reserved	non-reserved	reserved
JOIN	reserved (can be function or type)		reserved	reserved	reserved
JSON			non-reserved		
JSON_ARRAY			reserved		
JSON_ARRAYAGG			reserved		
JSON_EXISTS			reserved		
JSON_OBJECT			reserved		
JSON_OBJECTAGG			reserved		
JSON_QUERY			reserved		
JSON_TABLE			reserved		
JSON_TABLE_PRIMITIVE			reserved		
JSON_VALUE			reserved		
K			non-reserved	non-reserved	
KEEP			non-reserved		
KEY	non-reserved		non-reserved	non-reserved	reserved
KEYS			non-reserved		
KEY_MEMBER			non-reserved	non-reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
KEY_TYPE			non-reserved	non-reserved	
LABEL	non-reserved				
LAG			reserved	reserved	
LANGUAGE	non-reserved		reserved	reserved	reserved
LARGE	non-reserved		reserved	reserved	
LAST	non-reserved		non-reserved	non-reserved	reserved
LAST_VALUE			reserved	reserved	
LATERAL	reserved		reserved	reserved	
LEAD			reserved	reserved	
LEADING	reserved		reserved	reserved	reserved
LEAKPROOF	non-reserved				
LEAST	non-reserved (cannot be function or type)				
LEFT	reserved (can be function or type)		reserved	reserved	reserved
LENGTH			non-reserved	non-reserved	non-reserved
LESS		non-reserved			
LEVEL	non-reserved		non-reserved	non-reserved	reserved
LIBRARY			non-reserved	non-reserved	
LIKE	reserved (can be function or type)		reserved	reserved	reserved
LIKE_REGEX			reserved	reserved	
LIMIT	reserved, requires AS		non-reserved	non-reserved	
LINK			non-reserved	non-reserved	
LIST		non-reserved			
LISTAGG			reserved		
LISTEN	non-reserved				
LN			reserved	reserved	
LOAD	non-reserved				
LOCAL	non-reserved		reserved	reserved	reserved
LOCALTIME	reserved		reserved	reserved	
LOCALTIMESTAMP	reserved		reserved	reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
LOCATION	non-reserved		non-reserved	non-reserved	
LOCATOR			non-reserved	non-reserved	
LOCK	non-reserved				
LOCKED	non-reserved				
LOG			reserved		
LOG10			reserved		
LOGGED	non-reserved				
LOGGING		non-reserved			
LOOP		non-reserved			
LOWER			reserved	reserved	reserved
LT_SYNTAX_COMPATIBLE_TYPE	reserved (can be function or type)				
M			non-reserved	non-reserved	
MAP			non-reserved	non-reserved	
MAPPING	non-reserved		non-reserved	non-reserved	
MASTER		non-reserved			
MATCH	non-reserved		reserved	reserved	reserved
MATCHED		non-reserved	non-reserved	non-reserved	
MATCHES			reserved		
MATCH_NUMBER			reserved		
MATCH_RECOGNIZE			reserved		
MATERIALIZED	non-reserved				
MAX			reserved	reserved	reserved
MAXVALUE	non-reserved		non-reserved	non-reserved	
MEASURES			reserved		
MEMBER		non-reserved	reserved	reserved	
MERGE		non-reserved	reserved	reserved	
MESSAGE_LENGTH			non-reserved	non-reserved	non-reserved
MESSAGE_OCTET_LENGTH			non-reserved	non-reserved	non-reserved

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
MESSAGE_TEXT			non-reserved	non-reserved	non-reserved
METHOD	non-reserved		reserved	reserved	
MIN			reserved	reserved	reserved
MINUS	reserved, requires AS				
MINUTE	non-reserved, requires AS		reserved	reserved	reserved
MINVALUE	non-reserved		non-reserved	non-reserved	
MISSING		non-reserved			
MOD			reserved	reserved	
MODE	non-reserved				
MODIFIES			reserved	reserved	
MODIFY		non-reserved			
MODULE			reserved	reserved	reserved
MONITORING		non-reserved			
MONTH	non-reserved		reserved	reserved	reserved
MORE			non-reserved	non-reserved	non-reserved
MOVE	non-reserved				
MULTISET			reserved	reserved	
MUMPS			non-reserved	non-reserved	non-reserved
MYISAM		non-reserved			
NAME	non-reserved		non-reserved	non-reserved	non-reserved
NAMES	non-reserved		non-reserved	non-reserved	reserved
NAMESPACE			non-reserved	non-reserved	
NATIONAL	non-reserved (cannot be function or type)		reserved	reserved	reserved
NATURAL	reserved (can be function or type)		reserved	reserved	reserved
NCHAR	non-reserved (cannot be function or type)		reserved	reserved	reserved
NCLOB			reserved	reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
NESTED		non-reserved	non-reserved		
NESTING			non-reserved	non-reserved	
NEW	non-reserved		reserved	reserved	
NEXT	non-reserved		non-reserved	non-reserved	reserved
NFC	non-reserved		non-reserved	non-reserved	
NFD	non-reserved		non-reserved	non-reserved	
NFKC	non-reserved		non-reserved	non-reserved	
NFKD	non-reserved		non-reserved	non-reserved	
NIL			non-reserved	non-reserved	
NO	non-reserved		reserved	reserved	reserved
NOCACHE		non-reserved			
NOCOMPRESS		non-reserved			
NOCYCLE		non-reserved			
NOLOGGING		non-reserved			
NOMAXVALUE		non-reserved			
NOMINVALUE		non-reserved			
NONE	non-reserved (cannot be function or type)		reserved	reserved	
NONEDITIONABLE		non-reserved			
NOORDER		non-reserved			
NOPARALLEL		non-reserved			
NORMALIZE	non-reserved (cannot be function or type)		reserved	reserved	
NORMALIZED	non-reserved		non-reserved	non-reserved	
NOT	reserved		reserved	reserved	reserved

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
NOTHING	non-reserved				
NOTIFY	non-reserved				
NOTNULL	reserved (can be function or type), requires AS				
NOWAIT	non-reserved				
NTH_VALUE			reserved	reserved	
NTILE			reserved	reserved	
NULL	reserved		reserved	reserved	reserved
NULLABLE			non-reserved	non-reserved	non-reserved
NULLIF	non-reserved (cannot be function or type)		reserved	reserved	reserved
NULLS	non-reserved		non-reserved	non-reserved	
NUMBER		non-reserved (cannot be function or type)	non-reserved	non-reserved	non-reserved
NUMERIC	non-reserved (cannot be function or type)		reserved	reserved	reserved
OBJECT	non-reserved		non-reserved	non-reserved	
OCCURRENCES_REGEX			reserved	reserved	
OCTETS			non-reserved	non-reserved	
OCTET_LENGTH			reserved	reserved	reserved
OF	non-reserved		reserved	reserved	reserved
OFF	non-reserved		non-reserved	non-reserved	
OFFSET	reserved, requires AS		reserved	reserved	
OIDS	non-reserved				
OLD	non-reserved		reserved	reserved	
OMIT			reserved		
ON	reserved, requires AS		reserved	reserved	reserved
ONE			reserved		
ONLINE		non-reserved			



Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
ONLY	reserved		reserved	reserved	reserved
OPEN			reserved	reserved	reserved
OPERATOR	non-reserved				
OPTION	non-reserved		non-reserved	non-reserved	reserved
OPTIONS	non-reserved		non-reserved	non-reserved	
OR	reserved		reserved	reserved	reserved
ORDER	reserved, requires AS		reserved	reserved	reserved
ORDERED		non-reserved			
ORDERING			non-reserved	non-reserved	
ORDINALITY	non-reserved		non-reserved	non-reserved	
OTHERS	non-reserved		non-reserved	non-reserved	
OUT	non-reserved (cannot be function or type)		reserved	reserved	
OUTER	reserved (can be function or type)		reserved	reserved	reserved
OUTPUT			non-reserved	non-reserved	reserved
OVER	non-reserved, requires AS		reserved	reserved	
OVERFLOW		non-reserved	non-reserved		
OVERLAPS	reserved (can be function or type), requires AS		reserved	reserved	reserved
OVERLAY	non-reserved (cannot be function or type)		reserved	reserved	
OVERRIDING	non-reserved		non-reserved	non-reserved	
OWNED	non-reserved				
OWNER	non-reserved				
P			non-reserved	non-reserved	
PACKAGE		non-reserved			

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
PAD			non-reserved	non-reserved	reserved
PARALLEL	non-reserved				
PARALLEL_ENABLE		non-reserved			
PARAMETER			reserved	reserved	
PARAMETER_MODE			non-reserved	non-reserved	
PARAMETER_NAME			non-reserved	non-reserved	
PARAMETER_ORDINAL_POSITION			non-reserved	non-reserved	
PARAMETER_SPECIFIC_CATALOG			non-reserved	non-reserved	
PARAMETER_SPECIFIC_NAME			non-reserved	non-reserved	
PARAMETER_SPECIFIC_SCHEMA			non-reserved	non-reserved	
PARSER	non-reserved				
PARTIAL	non-reserved		non-reserved	non-reserved	reserved
PARTITION	reserved, requires AS		reserved	reserved	
PARTITIONS		non-reserved			
PASCAL			non-reserved	non-reserved	non-reserved
PASS			non-reserved		
PASSING	non-reserved		non-reserved	non-reserved	
PASSTHROUGH			non-reserved	non-reserved	
PASSWORD	non-reserved				
PAST			non-reserved		
PATH			non-reserved	non-reserved	
PATTERN			reserved		
PCTFREE		non-reserved			
PER			reserved		
PERCENT			reserved	reserved	
PERCENTILE_CONT			reserved	reserved	
PERCENTILE_DISC			reserved	reserved	
PERCENT_RANK			reserved	reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
PERIOD			reserved	reserved	
PERMISSION			non-reserved	non-reserved	
PERMUTE			reserved		
PIVOT	reserved				
PLACING	reserved		non-reserved	non-reserved	
PLAN			non-reserved		
PLANS	non-reserved				
PLI			non-reserved	non-reserved	non-reserved
POLICY	non-reserved				
PORTION			reserved	reserved	
POSITION	non-reserved (cannot be function or type)		reserved	reserved	reserved
POSITION_REGEX			reserved	reserved	
POWER			reserved	reserved	
PRECEDES			reserved	reserved	
PRECEDING	non-reserved		non-reserved	non-reserved	
PRECISION	non-reserved (cannot be function or type), requires AS		reserved	reserved	reserved
PREPARE	non-reserved		reserved	reserved	reserved
PREPARED	non-reserved				
PRESERVE	non-reserved		non-reserved	non-reserved	reserved
PRIMARY	reserved		reserved	reserved	reserved
PRIOR	reserved		non-reserved	non-reserved	reserved
PRIVATE			non-reserved		
PRIVILEGES	non-reserved		non-reserved	non-reserved	reserved
PROCEDURAL	non-reserved				
PROCEDURE	non-reserved		reserved	reserved	reserved
PROCEDURES	non-reserved				
PROGRAM	non-reserved				
PRUNE			non-reserved		

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
PTF			reserved		
PUBLIC			non-reserved	non-reserved	reserved
PUBLICATION	non-reserved				
QUOTE	non-reserved				
QUOTES			non-reserved		
RANDOMLY		non-reserved			
RANGE	non-reserved		reserved	reserved	
RANK			reserved	reserved	
READ	non-reserved		non-reserved	non-reserved	reserved
READABLE		non-reserved			
READS			reserved	reserved	
REAL	non-reserved (cannot be function or type)		reserved	reserved	reserved
REASSIGN	non-reserved				
REBUILD		non-reserved			
RECHECK	non-reserved				
RECORD		non-reserved			
RECOVERY			non-reserved	non-reserved	
RECURSIVE	non-reserved		reserved	reserved	
REF	non-reserved		reserved	reserved	
REFERENCES	reserved		reserved	reserved	reserved
REFERENCING	non-reserved		reserved	reserved	
REFRESH	non-reserved				
REGR_AVGX			reserved	reserved	
REGR_AVGY			reserved	reserved	
REGR_COUNT			reserved	reserved	
REGR_INTERCEPT			reserved	reserved	
REGR_R2			reserved	reserved	
REGR_SLOPE			reserved	reserved	
REGR_SXX			reserved	reserved	
REGR_SXY			reserved	reserved	
REGR_SYY			reserved	reserved	
REINDEX	non-reserved				

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
RELATIVE	non-reserved		non-reserved	non-reserved	reserved
RELEASE	non-reserved		reserved	reserved	
RENAME	non-reserved				
REPEATABLE	non-reserved		non-reserved	non-reserved	non-reserved
REPLACE	non-reserved				
REPLICA	non-reserved				
REPLICATED		non-reserved			
REQUIRING			non-reserved	non-reserved	
RESET	non-reserved				
RESPECT			non-reserved	non-reserved	
RESTART	non-reserved		non-reserved	non-reserved	
RESTORE			non-reserved	non-reserved	
RESTRICT	non-reserved		non-reserved	non-reserved	reserved
RESULT			reserved	reserved	
RETURN	reserved, requires AS		reserved	reserved	
RETURNED_CARDINALITY			non-reserved	non-reserved	
RETURNED_LENGTH			non-reserved	non-reserved	non-reserved
RETURNED_OCTET_LENGTH			non-reserved	non-reserved	non-reserved
RETURNED_SQLSTATE			non-reserved	non-reserved	non-reserved
RETURNING	reserved, requires AS		non-reserved	non-reserved	
RETURNS	non-reserved		reserved	reserved	
REVOKE	non-reserved		reserved	reserved	reserved
RIGHT	reserved (can be function or type)		reserved	reserved	reserved
ROLE	non-reserved		non-reserved	non-reserved	
ROLLBACK	non-reserved		reserved	reserved	reserved
ROLLUP	non-reserved		reserved	reserved	
ROOTPARTITION		non-reserved			

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
ROUTINE	non-reserved		non-reserved	non-reserved	
ROUTINES	non-reserved				
ROUTINE_CATALOG			non-reserved	non-reserved	
ROUTINE_NAME			non-reserved	non-reserved	
ROUTINE_SCHEMA			non-reserved	non-reserved	
ROW	non-reserved (cannot be function or type)		reserved	reserved	
ROWNUM	reserved				
ROWS	non-reserved		reserved	reserved	reserved
ROWTYPE		non-reserved			
ROW_COUNT			non-reserved	non-reserved	non-reserved
ROW_NUMBER			reserved	reserved	
RULE	non-reserved				
RUNNING			reserved		
SAVEPOINT	non-reserved		reserved	reserved	
SCALAR			non-reserved		
SCALE			non-reserved	non-reserved	non-reserved
SCHEMA	non-reserved		non-reserved	non-reserved	reserved
SCHEMAS	non-reserved				
SCHEMA_NAME			non-reserved	non-reserved	non-reserved
SCOPE			reserved	reserved	
SCOPE_CATALOG			non-reserved	non-reserved	
SCOPE_NAME			non-reserved	non-reserved	
SCOPE_SCHEMA			non-reserved	non-reserved	
SCROLL	non-reserved		reserved	reserved	reserved
SEARCH	non-reserved		reserved	reserved	
SECOND	non-reserved, requires AS		reserved	reserved	reserved
SECTION			non-reserved	non-reserved	reserved

## SQL关键词

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
SECURITY	non-reserved		non-reserved	non-reserved	
SEEK			reserved		
SELECT	reserved		reserved	reserved	reserved
SELECTIVE			non-reserved	non-reserved	
SELF			non-reserved	non-reserved	
SENSITIVE			reserved	reserved	
SEPARATOR		non-reserved			
SEQUENCE	non-reserved		non-reserved	non-reserved	
SEQUENCES	non-reserved				
SERIAL		non-reserved			
SERIALIZABLE	non-reserved		non-reserved	non-reserved	non-reserved
SERVER	non-reserved		non-reserved	non-reserved	
SERVER_NAME			non-reserved	non-reserved	non-reserved
SESSION	non-reserved		non-reserved	non-reserved	reserved
SESSION_USER	reserved		reserved	reserved	reserved
SET	non-reserved		reserved	reserved	reserved
SETOF	non-reserved (cannot be function or type)				
SETS	non-reserved		non-reserved	non-reserved	
SHARD_COUNT		non-reserved			
SHARE	non-reserved				
SHOW	non-reserved		reserved		
SIMILAR	reserved (can be function or type)		reserved	reserved	
SIMPLE	non-reserved		non-reserved	non-reserved	
SIN			reserved		
SINH			reserved		
SIZE			non-reserved	non-reserved	reserved
SKIP	non-reserved		reserved		

## SQL关键词

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
SMALLINT	non-reserved (cannot be function or type)		reserved	reserved	reserved
SMALLSERIAL		non- reserved			
SNAPSHOT	non-reserved				
SOME	reserved, requires AS		reserved	reserved	reserved
SOURCE			non- reserved	non- reserved	
SPACE			non- reserved	non- reserved	reserved
SPECIFIC			reserved	reserved	
SPECIFICTYPE			reserved	reserved	
SPECIFIC_NAME			non- reserved	non- reserved	
SPLIT		non- reserved			
SQL	non-reserved		reserved	reserved	reserved
SQLCODE					reserved
SQLERROR					reserved
SQLEXCEPTION			reserved	reserved	
SQLSTATE			reserved	reserved	reserved
SQLWARNING			reserved	reserved	
SQRT			reserved	reserved	
STABLE	non-reserved				
STANDALONE	non-reserved		non- reserved	non- reserved	
START	non-reserved		reserved	reserved	
STATE			non- reserved	non- reserved	
STATEMENT	non-reserved		non- reserved	non- reserved	
STATIC			reserved	reserved	
STATISTICS	non-reserved				
STDDEV_POP			reserved	reserved	
STDDEV_SAMP			reserved	reserved	
STDIN	non-reserved				
STDOUT	non-reserved				
STORAGE	non-reserved				
STORE		non- reserved			
STORED	non-reserved				



Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
STRICT	non-reserved				
STRING			non-reserved		
STRIP	non-reserved		non-reserved	non-reserved	
STRUCTURE			non-reserved	non-reserved	
STYLE			non-reserved	non-reserved	
SUBCLASS_ORIGIN			non-reserved	non-reserved	non-reserved
SUBMULTISET			reserved	reserved	
SUBPARTITION		non-reserved			
SUBPARTITIONS		non-reserved			
SUBSCRIPTION	non-reserved				
SUBSET			reserved		
SUBSTRING	non-reserved (cannot be function or type)		reserved	reserved	reserved
SUBSTRING_REGEX			reserved	reserved	
SUBTYPE		non-reserved			
SUCCEEDS			reserved	reserved	
SUM			reserved	reserved	reserved
SUPPORT	non-reserved				
SYMMETRIC	reserved		reserved	reserved	
SYSDATE	reserved (can be function or type)				
SYSID	non-reserved				
SYSTEM	non-reserved		reserved	reserved	
SYSTEM_TIME			reserved	reserved	
SYSTEM_USER			reserved	reserved	reserved
SYSTIMESTAMP	reserved (can be function or type)				
SYS_CONNECT_BY_PATH		non-reserved			
T			non-reserved	non-reserved	
TABLE	reserved		reserved	reserved	reserved
TABLES	non-reserved				

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
TABLESAMPLE	reserved (can be function or type)		reserved	reserved	
TABLESPACE	non-reserved				
TABLE_NAME			non-reserved	non-reserved	non-reserved
TAN			reserved		
TANH			reserved		
TEMP	non-reserved				
TEMPLATE	non-reserved				
TEMPORARY	non-reserved		non-reserved	non-reserved	reserved
TEXT	non-reserved				
THAN		non-reserved			
THEN	reserved		reserved	reserved	reserved
THROUGH			non-reserved		
TIES	non-reserved		non-reserved	non-reserved	
TIME	non-reserved (cannot be function or type)		reserved	reserved	reserved
TIMESTAMP	non-reserved (cannot be function or type)		reserved	reserved	reserved
TIMEZONE_HOUR			reserved	reserved	reserved
TIMEZONE_MINUTE			reserved	reserved	reserved
TINYINT		non-reserved (cannot be function or type)			
TO	reserved, requires AS		reserved	reserved	reserved
TOKEN			non-reserved	non-reserved	
TOP_LEVEL_COUNT			non-reserved	non-reserved	
TRAILING	reserved		reserved	reserved	reserved
TRANSACTION	non-reserved		non-reserved	non-reserved	reserved
TRANSACTIONS_COMMITTED			non-reserved	non-reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
TRANSACTIONS_ROLLED_BACK			non-reserved	non-reserved	
TRANSACTION_ACTIVE			non-reserved	non-reserved	
TRANSFORM	non-reserved		non-reserved	non-reserved	
TRANSFORMS			non-reserved	non-reserved	
TRANSLATE			reserved	reserved	reserved
TRANSLATE_REGEX			reserved	reserved	
TRANSLATION			reserved	reserved	reserved
TREAT	non-reserved (cannot be function or type)		reserved	reserved	
TRIGGER	non-reserved		reserved	reserved	
TRIGGER_CATALOG			non-reserved	non-reserved	
TRIGGER_NAME			non-reserved	non-reserved	
TRIGGER_SCHEMA			non-reserved	non-reserved	
TRIM	non-reserved (cannot be function or type)		reserved	reserved	reserved
TRIM_ARRAY			reserved	reserved	
TRUE	reserved		reserved	reserved	reserved
TRUNCATE	non-reserved		reserved	reserved	
TRUSTED	non-reserved				
TYPE	non-reserved		non-reserved	non-reserved	non-reserved
TYPES	non-reserved				
UESCAPE	non-reserved		reserved	reserved	
UNBOUNDED	non-reserved		non-reserved	non-reserved	
UNCOMMITTED	non-reserved		non-reserved	non-reserved	non-reserved
UNCONDITIONAL			non-reserved		
UNDER			non-reserved	non-reserved	
UNENCRYPTED	non-reserved				
UNION	reserved, requires AS		reserved	reserved	reserved
UNIQUE	reserved		reserved	reserved	reserved

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
UNKNOWN	non-reserved		reserved	reserved	reserved
UNLIMITED		non-reserved			
UNLINK			non-reserved	non-reserved	
UNLISTEN	non-reserved				
UNLOGGED	non-reserved				
UNMATCHED			reserved		
UNNAMED			non-reserved	non-reserved	non-reserved
UNNEST			reserved	reserved	
UNSIGNED	non-reserved				
UNTIL	non-reserved				
UNTYPED			non-reserved	non-reserved	
UPDATE	non-reserved		reserved	reserved	reserved
UPPER			reserved	reserved	reserved
URI			non-reserved	non-reserved	
USAGE			non-reserved	non-reserved	reserved
USE		non-reserved			
USER	reserved		reserved	reserved	reserved
USER_DEFINED_TYPE_CATALOG			non-reserved	non-reserved	
USER_DEFINED_TYPE_CODE			non-reserved	non-reserved	
USER_DEFINED_TYPE_NAME			non-reserved	non-reserved	
USER_DEFINED_TYPE_SCHEMA			non-reserved	non-reserved	
USING	reserved		reserved	reserved	reserved
UTF16			non-reserved		
UTF32			non-reserved		
UTF8			non-reserved		
VACUUM	non-reserved				
VALID	non-reserved		non-reserved	non-reserved	
VALIDATE	non-reserved				
VALIDATOR	non-reserved				
VALUE	non-reserved		reserved	reserved	reserved

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
VALUES	non-reserved (cannot be function or type)		reserved	reserved	reserved
VALUE_OF			reserved	reserved	
VARBINARY			reserved	reserved	
VARCHAR	non-reserved (cannot be function or type)		reserved	reserved	reserved
VARIADIC	reserved				
VARYING	non-reserved, requires AS		reserved	reserved	reserved
VAR_POP			reserved	reserved	
VAR_SAMP			reserved	reserved	
VERBOSE	reserved (can be function or type)				
VERSION	non-reserved		non- reserved	non- reserved	
VERSIONING			reserved	reserved	
VIEW	non-reserved		non- reserved	non- reserved	reserved
VIEWS	non-reserved				
VOLATILE	non-reserved				
WELLFORMED		non- reserved			
WHEN	reserved		reserved	reserved	reserved
WHENEVER			reserved	reserved	reserved
WHERE	reserved, requires AS		reserved	reserved	reserved
WHILE		non- reserved			
WHITESPACE	non-reserved		non- reserved	non- reserved	
WIDTH_BUCKET			reserved	reserved	
WINDOW	reserved, requires AS		reserved	reserved	
WITH	reserved, requires AS		reserved	reserved	reserved
WITHIN	non-reserved, requires AS		reserved	reserved	
WITHOUT	non-reserved, requires AS		reserved	reserved	
WORK	non-reserved		non- reserved	non- reserved	reserved

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
WRAPPER	non-reserved		non-reserved	non-reserved	
WRITABLE		non-reserved			
WRITE	non-reserved		non-reserved	non-reserved	reserved
XML	non-reserved		reserved	reserved	
XMLAGG			reserved	reserved	
XMLATTRIBUTES	non-reserved (cannot be function or type)		reserved	reserved	
XMLBINARY			reserved	reserved	
XMLCAST			reserved	reserved	
XMLCOMMENT			reserved	reserved	
XMLCONCAT	non-reserved (cannot be function or type)		reserved	reserved	
XMLDECLARATION			non-reserved	non-reserved	
XMLDOCUMENT			reserved	reserved	
XMLELEMENT	non-reserved (cannot be function or type)		reserved	reserved	
XML EXISTS	non-reserved (cannot be function or type)		reserved	reserved	
XMLFOREST	non-reserved (cannot be function or type)		reserved	reserved	
XMLITERATE			reserved	reserved	
XMLNAMESPACES	non-reserved (cannot be function or type)		reserved	reserved	
XMLPARSE	non-reserved (cannot be function or type)		reserved	reserved	
XMLPI	non-reserved (cannot be function or type)		reserved	reserved	
XMLQUERY			reserved	reserved	

Key Word	PostgreSQL	LightDB	SQL:2016	SQL:2011	SQL-92
XMLROOT	non-reserved (cannot be function or type)				
XMLSCHEMA			non- reserved	non- reserved	
XMLSERIALIZE	non-reserved (cannot be function or type)		reserved	reserved	
XMLTABLE	non-reserved (cannot be function or type)		reserved	reserved	
XMLTEXT			reserved	reserved	
XMLVALIDATE			reserved	reserved	
YEAR	non-reserved, requires AS		reserved	reserved	reserved
YES	non-reserved		non- reserved	non- reserved	
ZONE	non-reserved		non- reserved	non- reserved	reserved

---

## Appendix F. SQL 符合性

这一节尝试勾勒出LightDB与当前 SQL 标准相符合的范围。下面的信息并不是符合性的完整说明，但是它尽可能详细地表达了对用户合理且有用的主要主题。

SQL 标准的正式名称是 ISO/IEC 9075 “数据库语言 SQL”。该标准会不时地发布一个修改的版本，最近一次更新出现在 2016 年。2016年的版本被称为 ISO/IEC 9075:2016，或者简单地称为 SQL:2016。之前的版本是 SQL:2011, SQL:2008, SQL:2006, SQL:2003, SQL:1999 和 SQL-92。每一个版本会替换之前的一个版本，因此声称与早前的版本相符合没有意义。LightDB的开发希望与该标准的最新官方版本相符，并且这种符合不会与额外特性或尝试相冲突。很多 SQL 标准所要求的特性都被支持，不过有时在语法或函数上有所不同。随着时间的推移，符合性会得到进一步的提高。

SQL-92为符合性定义了三个特性集：入口、中间和完整。大部分号称SQL标准符合的数据库管理系统只是在入口级别上的符合，因为中间和完整级别的整个特性集合要么太多要么与遗留行为冲突。

从SQL:1999开始，SQL 标准定义了一个大型的个体特性集合，而没有无用地拓宽SQL-92中的三个级别。这些特性中的一个大型子集代表“核心”特性，每一个符合 SQL 的实现都必须提供。剩下的特性纯粹是可选的。

从SQL:2003开始的标准版本也被划分成数个部分。每一个部分有一个速记名。注意这些部分不是连续编号。

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)
- ISO/IEC 9075-15 Multi-dimensional arrays (SQL/MDA)

LightDB核心覆盖了部分 1、2、9、11 和 14。部分 3 被 ODBC 驱动所覆盖，并且部分 13 被 PL/Java 插件所覆盖，但是目前准确的符合性还没有在这些组件上被验证。目前对于LightDB没有4、10和15部分的实现。

LightDB 支持 SQL:2016 的大部分主要特性。在 177 个完整核心符合所要求的强制特性中，LightDB 至少符合 170 个。另外，还有一个受支持的可选特性的长长的列表。值得注意的是，在编写此文档时，还没有任何数据库管理系统的当前版本声称完全符合核心的 SQL:2016。



在下面的两节中，我们提供了一个LightDB所支持特性的列表，以及一个在SQL:2016中定义却还未被LightDB支持的特性的列表。这两个列表都是大概的：对于被列为支持的一个特性可能会有少量的细节不符合，而且大部分未被支持的特性可能事实上已经被实现。本文档的主体部分包含了哪些能用哪些不能用的准确信息。

### Note

包含一个连字符的特性编码是子特性。因此，如果一个特定的子特性没有被支持，其主特性被列为未支持，即使其他的子特性都已被支持。

## F. 1. 已支持特性

标识符	核心?	描述	注释
B012		Embedded C	
B021		Direct SQL	
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	
E011-02	Core	REAL, DOUBLE PRECISION, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	
E021-04	Core	CHARACTER_LENGTH function	trims trailing spaces from CHARACTER values before counting
E021-05	Core	OCTET_LENGTH function	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenation	
E021-08	Core	UPPER and LOWER functions	
E021-09	Core	TRIM function	
E021-10	Core	Implicit casting among the character string types	
E021-11	Core	POSITION function	
E021-12	Core	Character comparison	
E031	Core	Identifiers	
E031-01	Core	Delimited identifiers	
E031-02	Core	Lower case identifiers	
E031-03	Core	Trailing underscore	

标识符	核心?	描述	注释
E051	Core	Basic query specification	
E051-01	Core	SELECT DISTINCT	
E051-02	Core	GROUP BY clause	
E051-04	Core	GROUP BY can contain columns not in <select list>	
E051-05	Core	Select list items can be renamed	
E051-06	Core	HAVING clause	
E051-07	Core	Qualified * in select list	
E051-08	Core	Correlation names in the FROM clause	
E051-09	Core	Rename columns in the FROM clause	
E061	Core	Basic predicates and search conditions	
E061-01	Core	Comparison predicate	
E061-02	Core	BETWEEN predicate	
E061-03	Core	IN predicate with list of values	
E061-04	Core	LIKE predicate	
E061-05	Core	LIKE predicate ESCAPE clause	
E061-06	Core	NULL predicate	
E061-07	Core	Quantified comparison predicate	
E061-08	Core	EXISTS predicate	
E061-09	Core	Subqueries in comparison predicate	
E061-11	Core	Subqueries in IN predicate	
E061-12	Core	Subqueries in quantified comparison predicate	
E061-13	Core	Correlated subqueries	
E061-14	Core	Search condition	
E071	Core	Basic query expressions	
E071-01	Core	UNION DISTINCT table operator	
E071-02	Core	UNION ALL table operator	
E071-03	Core	EXCEPT DISTINCT table operator	
E071-05	Core	Columns combined via table operators need not have exactly the same data type	
E071-06	Core	Table operators in subqueries	
E081	Core	Basic Privileges	
E081-01	Core	SELECT privilege	
E081-02	Core	DELETE privilege	
E081-03	Core	INSERT privilege at the table level	
E081-04	Core	UPDATE privilege at the table level	
E081-05	Core	UPDATE privilege at the column level	
E081-06	Core	REFERENCES privilege at the table level	
E081-07	Core	REFERENCES privilege at the column level	
E081-08	Core	WITH GRANT OPTION	

标识符	核心?	描述	注释
E081-09	Core	USAGE privilege	
E081-10	Core	EXECUTE privilege	
E091	Core	Set functions	
E091-01	Core	AVG	
E091-02	Core	COUNT	
E091-03	Core	MAX	
E091-04	Core	MIN	
E091-05	Core	SUM	
E091-06	Core	ALL quantifier	
E091-07	Core	DISTINCT quantifier	
E101	Core	Basic data manipulation	
E101-01	Core	INSERT statement	
E101-03	Core	Searched UPDATE statement	
E101-04	Core	Searched DELETE statement	
E111	Core	Single row SELECT statement	
E121	Core	Basic cursor support	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER BY columns need not be in select list	
E121-03	Core	Value expressions in ORDER BY clause	
E121-04	Core	OPEN statement	
E121-06	Core	Positioned UPDATE statement	
E121-07	Core	Positioned DELETE statement	
E121-08	Core	CLOSE statement	
E121-10	Core	FETCH statement implicit NEXT	
E121-17	Core	WITH HOLD cursors	
E131	Core	Null value support (nulls in lieu of values)	
E141	Core	Basic integrity constraints	
E141-01	Core	NOT NULL constraints	
E141-02	Core	UNIQUE constraints of NOT NULL columns	
E141-03	Core	PRIMARY KEY constraints	
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	
E141-06	Core	CHECK constraints	
E141-07	Core	Column defaults	
E141-08	Core	NOT NULL inferred on PRIMARY KEY	
E141-10	Core	Names in a foreign key can be specified in any order	
E151	Core	Transaction support	
E151-01	Core	COMMIT statement	
E151-02	Core	ROLLBACK statement	

标识符	核心?	描述	注释
E152	Core	Basic SET TRANSACTION statement	
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	
E153	Core	Updatable queries with subqueries	
E161	Core	SQL comments using leading double minus	
E171	Core	SQLSTATE support	
E182	Core	Host language binding	
F021	Core	Basic information schema	
F021-01	Core	COLUMNS view	
F021-02	Core	TABLES view	
F021-03	Core	VIEWS view	
F021-04	Core	TABLE_CONSTRAINTS view	
F021-05	Core	REFERENTIAL_CONSTRAINTS view	
F021-06	Core	CHECK_CONSTRAINTS view	
F031	Core	Basic schema manipulation	
F031-01	Core	CREATE TABLE statement to create persistent base tables	
F031-02	Core	CREATE VIEW statement	
F031-03	Core	GRANT statement	
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause	
F031-13	Core	DROP TABLE statement: RESTRICT clause	
F031-16	Core	DROP VIEW statement: RESTRICT clause	
F031-19	Core	REVOKE statement: RESTRICT clause	
F032		CASCADE drop behavior	
F033		ALTER TABLE statement: DROP COLUMN clause	
F034		Extended REVOKE statement	
F034-01		REVOKE statement performed by other than the owner of a schema object	
F034-02		REVOKE statement: GRANT OPTION FOR clause	
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F041	Core	Basic joined table	
F041-01	Core	Inner join (but not necessarily the INNER keyword)	
F041-02	Core	INNER keyword	
F041-03	Core	LEFT OUTER JOIN	
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Outer joins can be nested	
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join	

标识符	核心?	描述	注释
F041-08	Core	All comparison operators are supported (rather than just =)	
F051	Core	Basic date and time	
F051-01	Core	DATE data type (including support of DATE literal)	
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types	
F051-05	Core	Explicit CAST between datetime types and character string types	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	
F051-08	Core	LOCALTIMESTAMP	
F052		Intervals and datetime arithmetic	
F053		OVERLAPS predicate	
F081	Core	UNION and EXCEPT in views	
F111		Isolation levels other than SERIALIZABLE	
F111-01		READ UNCOMMITTED isolation level	
F111-02		READ COMMITTED isolation level	
F111-03		REPEATABLE READ isolation level	
F131	Core	Grouped operations	
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	
F131-02	Core	Multiple tables supported in queries with grouped views	
F131-03	Core	Set functions supported in queries with grouped views	
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views	
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F171		Multiple schemas per user	
F181	Core	Multiple module support	
F191		Referential delete actions	
F200		TRUNCATE TABLE statement	
F201	Core	CAST function	
F202		TRUNCATE TABLE: identity column restart option	
F221	Core	Explicit defaults	

标识符	核心?	描述	注释
F222		INSERT statement: DEFAULT VALUES clause	
F231		Privilege tables	
F231-01		TABLE_PRIVILEGES view	
F231-02		COLUMN_PRIVILEGES view	
F231-03		USAGE_PRIVILEGES view	
F251		Domain support	
F261	Core	CASE expression	
F261-01	Core	Simple CASE	
F261-02	Core	Searched CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	
F262		Extended CASE expression	
F271		Compound character literals	
F281		LIKE enhancements	
F302		INTERSECT table operator	
F302-01		INTERSECT DISTINCT table operator	
F302-02		INTERSECT ALL table operator	
F304		EXCEPT ALL table operator	
F311	Core	Schema definition statement	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE for persistent base tables	
F311-03	Core	CREATE VIEW	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F311-05	Core	GRANT statement	
F321		User authorization	
F361		Subprogram support	
F381		Extended schema manipulation	
F381-01		ALTER TABLE statement: ALTER COLUMN clause	
F381-02		ALTER TABLE statement: ADD CONSTRAINT clause	
F381-03		ALTER TABLE statement: DROP CONSTRAINT clause	
F382		Alter column data type	
F383		Set column not null clause	
F384		Drop identity property clause	
F385		Drop column generation expression clause	
F386		Set identity column generation clause	
F391		Long identifiers	
F392		Unicode escapes in identifiers	
F393		Unicode escapes in literals	
F394		Optional normal form specification	

标识符	核心?	描述	注释
F401		Extended joined table	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	
F402		Named column joins for LOBs, arrays, and multisets	
F411		Time zone specification	differences regarding literal interpretation
F421		National character	
F431		Read-only scrollable cursors	
F431-01		FETCH with explicit NEXT	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		Extended set function support	
F442		Mixed column references in set functions	
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predicate	
F491		Constraint management	
F501	Core	Features and conformance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	
F502		Enhanced documentation tables	
F531		Temporary tables	
F555		Enhanced seconds precision	
F561		Full value expressions	
F571		Truth value tests	
F591		Derived tables	
F611		Indicator data types	
F641		Row and table constructors	
F651		Catalog name qualifiers	
F661		Simple tables	
F672		Retrospective check constraints	
F690		Collation support	but no character set support
F692		Extended collation support	
F701		Referential update actions	
F711		ALTER domain	

标识符	核心?	描述	注释
F731		INSERT column privileges	
F751		View CHECK enhancements	
F761		Session management	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		Connection management	
F781		Self-referencing operations	
F791		Insensitive cursors	
F801		Full set function	
F850		Top-level <order by clause> in <query expression>	
F851		<order by clause> in subqueries	
F852		Top-level <order by clause> in views	
F855		Nested <order by clause> in <query expression>	
F856		Nested <fetch first clause> in <query expression>	
F857		Top-level <fetch first clause> in <query expression>	
F858		<fetch first clause> in subqueries	
F859		Top-level <fetch first clause> in views	
F860		<fetch first row count> in <fetch first clause>	
F861		Top-level <result offset clause> in <query expression>	
F862		<result offset clause> in subqueries	
F863		Nested <result offset clause> in <query expression>	
F864		Top-level <result offset clause> in views	
F865		<offset row count> in <result offset clause>	
F867		FETCH FIRST clause: WITH TIES option	
S071		SQL paths in function and type name resolution	
S092		Arrays of user-defined types	
S095		Array constructors by query	
S096		Optional array bounds	
S098		ARRAY_AGG	
S111		ONLY in query expressions	
S201		SQL-invoked routines on arrays	
S201-01		Array parameters	
S201-02		Array as result type of functions	
S211		User-defined cast functions	



标识符	核心?	描述	注释
S301		Enhanced UNNEST	
T031		BOOLEAN data type	
T071		BIGINT data type	
T121		WITH (excluding RECURSIVE) in query expression	
T122		WITH (excluding RECURSIVE) in subquery	
T131		Recursive query	
T132		Recursive query in subquery	
T141		SIMILAR predicate	
T151		DISTINCT predicate	
T152		DISTINCT predicate with negation	
T171		LIKE clause in table definition	
T172		AS subquery clause in table definition	
T173		Extended LIKE clause in table definition	
T174		Identity columns	
T177		Sequence generator support: simple restart option	
T178		Identity columns: simple restart option	
T191		Referential action RESTRICT	
T201		Comparable data types for referential constraints	
T211-01		Triggers activated on UPDATE, INSERT, or DELETE of one base table	
T211-02		BEFORE triggers	
T211-03		AFTER triggers	
T211-04		FOR EACH ROW triggers	
T211-05		Ability to specify a search condition that must be true before the trigger is invoked	
T211-07		TRIGGER privilege	
T212		Enhanced trigger capability	
T213		INSTEAD OF triggers	
T231		Sensitive cursors	
T241		START TRANSACTION statement	
T261		Chained transactions	
T271		Savepoints	
T281		SELECT privilege with column granularity	
T285		Enhanced derived column names	
T312		OVERLAY function	
T321-01	Core	User-defined functions with no overloading	
T321-02	Core	User-defined stored procedures with no overloading	
T321-03	Core	Function invocation	

标识符	核心?	描述	注释
T321-04	Core	CALL statement	
T321-06	Core	ROUTINES view	
T321-07	Core	PARAMETERS view	
T323		Explicit security for external routines	
T325		Qualified SQL parameter references	
T331		Basic roles	
T341		Overloading of SQL-invoked functions and procedures	
T351		Bracketed SQL comments (/*...*/ comments)	
T431		Extended grouping capabilities	
T432		Nested and concatenated GROUPING SETS	
T433		Multiargument GROUPING function	
T441		ABS and MOD functions	
T461		Symmetric BETWEEN predicate	
T491		LATERAL derived table	
T501		Enhanced EXISTS predicate	
T521		Named arguments in CALL statement	
T523		Default values for INOUT parameters of SQL-invoked procedures	
T524		Named arguments in routine invocations other than a CALL statement	
T525		Default values for parameters of SQL-invoked functions	
T551		Optional key words for default syntax	
T581		Regular expression substring function	
T591		UNIQUE constraints of possibly null columns	
T611		Elementary OLAP operations	
T612		Advanced OLAP operations	
T613		Sampling	
T614		NTILE function	
T615		LEAD and LAG functions	
T617		FIRST_VALUE and LAST_VALUE function	
T620		WINDOW clause: GROUPS option	
T621		Enhanced numeric functions	
T622		Trigonometric functions	
T623		General logarithm functions	
T624		Common logarithm functions	
T631	Core	IN predicate with one list element	
T651		SQL-schema statements in SQL routines	
T653		SQL-schema statements in external routines	
T655		Cyclically dependent routines	

标识符	核心?	描述	注释
T831		SQL/JSON path language: strict mode	
T832		SQL/JSON path language: item method	
T833		SQL/JSON path language: multiple subscripts	
T834		SQL/JSON path language: wildcard member accessor	
T835		SQL/JSON path language: filter expressions	
T836		SQL/JSON path language: starts with predicate	
T837		SQL/JSON path language: regex_like predicate	
X010		XML type	
X011		Arrays of XML type	
X014		Attributes of XML type	
X016		Persistent XML values	
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	
X034		XMLAgg	
X035		XMLAgg: ORDER BY option	
X036		XMLComment	
X037		XMLPI	
X040		Basic table mapping	
X041		Basic table mapping: nulls absent	
X042		Basic table mapping: null as nil	
X043		Basic table mapping: table as forest	
X044		Basic table mapping: table as element	
X045		Basic table mapping: with target namespace	
X046		Basic table mapping: data mapping	
X047		Basic table mapping: metadata mapping	
X048		Basic table mapping: base64 encoding of binary strings	
X049		Basic table mapping: hex encoding of binary strings	
X050		Advanced table mapping	
X051		Advanced table mapping: nulls absent	
X052		Advanced table mapping: null as nil	
X053		Advanced table mapping: table as forest	
X054		Advanced table mapping: table as element	
X055		Advanced table mapping: with target namespace	
X056		Advanced table mapping: data mapping	
X057		Advanced table mapping: metadata mapping	

标识符	核心?	描述	注释
X058		Advanced table mapping: base64 encoding of binary strings	
X059		Advanced table mapping: hex encoding of binary strings	
X060		XMLParse: character string input and CONTENT option	
X061		XMLParse: character string input and DOCUMENT option	
X070		XMLSerialize: character string serialization and CONTENT option	
X071		XMLSerialize: character string serialization and DOCUMENT option	
X072		XMLSerialize: character string serialization	
X090		XML document predicate	
X120		XML parameters in SQL routines	
X121		XML parameters in external routines	
X221		XML passing mechanism BY VALUE	
X301		XMLTable: derived column list option	
X302		XMLTable: ordinality column option	
X303		XMLTable: column default option	
X304		XMLTable: passing a context item	must be XML DOCUMENT
X400		Name and identifier mapping	
X410		Alter column data type: XML type	

## F. 2. 未支持特性

下列定义在SQL:2016中的特性还没有在这个LightDB发行中被实现。在一些情况中，有等效的功能可用。

标识符	核心?	描述	注释
B011		Embedded Ada	
B013		Embedded COBOL	
B014		Embedded Fortran	
B015		Embedded MUMPS	
B016		Embedded Pascal	
B017		Embedded PL/I	
B031		Basic dynamic SQL	
B032		Extended dynamic SQL	
B032-01		<describe input statement>	
B033		Untyped SQL-invoked function arguments	
B034		Dynamic specification of cursor attributes	
B035		Non-extended descriptor names	

标识符	核心?	描述	注释
B041		Extensions to embedded SQL exception declarations	
B051		Enhanced execution rights	
B111		Module language Ada	
B112		Module language C	
B113		Module language COBOL	
B114		Module language Fortran	
B115		Module language MUMPS	
B116		Module language Pascal	
B117		Module language PL/I	
B121		Routine language Ada	
B122		Routine language C	
B123		Routine language COBOL	
B124		Routine language Fortran	
B125		Routine language MUMPS	
B126		Routine language Pascal	
B127		Routine language PL/I	
B128		Routine language SQL	
B200		Polymorphic table functions	
B201		More than one PTF generic table parameter	
B202		PTF Copartitioning	
B203		More than one copartition specification	
B204		PRUNE WHEN EMPTY	
B205		Pass-through columns	
B206		PTF descriptor parameters	
B207		Cross products of partitionings	
B208		PTF component procedure interface	
B209		PTF extended names	
B211		Module language Ada: VARCHAR and NUMERIC support	
B221		Routine language Ada: VARCHAR and NUMERIC support	
F054		TIMESTAMP in DATE type precedence list	
F121		Basic diagnostics management	
F121-01		GET DIAGNOSTICS statement	
F121-02		SET TRANSACTION statement: DIAGNOSTICS SIZE clause	
F122		Enhanced diagnostics management	
F123		All diagnostics	
F263		Comma-separated predicates in simple CASE expression	

标识符	核心?	描述	注释
F291		UNIQUE predicate	
F301		CORRESPONDING in query expressions	
F312		MERGE statement	consider INSERT ... ON CONFLICT DO UPDATE
F313		Enhanced MERGE statement	
F314		MERGE statement with DELETE branch	
F341		Usage tables	no ROUTINE_*_USAGE tables
F403		Partitioned joined tables	
F404		Range variable for common column names	
F451		Character set definition	
F461		Named character sets	
F492		Optional table constraint enforcement	
F521		Assertions	
F671		Subqueries in CHECK	intentionally omitted
F673		Reads SQL-data routine invocations in CHECK constraints	
F693		SQL-session and client module collations	
F695		Translation support	
F696		Additional translation documentation	
F721		Deferrable constraints	foreign and unique keys only
F741		Referential MATCH types	no partial match yet
F812	Core	Basic flagging	
F813		Extended flagging	
F821		Local table references	
F831		Full cursor update	
F831-01		Updatable scrollable cursors	
F831-02		Updatable ordered cursors	
F841		LIKE_REGEX predicate	
F842		OCCURRENCES_REGEX function	
F843		POSITION_REGEX function	
F844		SUBSTRING_REGEX function	
F845		TRANSLATE_REGEX function	
F846		Octet support in regular expression operators	
F847		Nonconstant regular expressions	
F866		FETCH FIRST clause: PERCENT option	
R010		Row pattern recognition: FROM clause	

标识符	核心?	描述	注释
R020		Row pattern recognition: WINDOW clause	
R030		Row pattern recognition: full aggregate support	
S011	Core	Distinct data types	
S011-01	Core	USER_DEFINED_TYPES view	
S023		Basic structured types	
S024		Enhanced structured types	
S025		Final structured types	
S026		Self-referencing structured types	
S027		Create method by specific method name	
S028		Permutable UDT options list	
S041		Basic reference types	
S043		Enhanced reference types	
S051		Create table of type	partially supported
S081		Subtables	
S091		Basic array support	partially supported
S091-01		Arrays of built-in data types	
S091-02		Arrays of distinct types	
S091-03		Array expressions	
S094		Arrays of reference types	
S097		Array element assignment	
S151		Type predicate	
S161		Subtype treatment	
S162		Subtype treatment for references	
S202		SQL-invoked routines on multisets	
S231		Structured type locators	
S232		Array locators	
S233		Multiset locators	
S241		Transform functions	
S242		Alter transform statement	
S251		User-defined orderings	
S261		Specific type method	
S271		Basic multiset support	
S272		Multisets of user-defined types	
S274		Multisets of reference types	
S275		Advanced multiset support	
S281		Nested collection types	
S291		Unique constraint on entire row	
S401		Distinct types based on array types	

标识符	核心?	描述	注释
S402		Distinct types based on distinct types	
S403		ARRAY_MAX_CARDINALITY	
S404		TRIM_ARRAY	
T011		Timestamp in Information Schema	
T021		BINARY and VARBINARY data types	
T022		Advanced support for BINARY and VARBINARY data types	
T023		Compound binary literal	
T024		Spaces in binary literals	
T041		Basic LOB data type support	
T041-01		BLOB data type	
T041-02		CLOB data type	
T041-03		POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	
T041-04		Concatenation of LOB data types	
T041-05		LOB locator: non-holdable	
T042		Extended LOB data type support	
T043		Multiplier T	
T044		Multiplier P	
T051		Row types	
T053		Explicit aliases for all-fields reference	
T061		UCS support	
T076		DECFLOAT data type	
T101		Enhanced nullability determination	
T111		Updatable joins, unions, and columns	
T175		Generated columns	mostly supported
T176		Sequence generator support	supported except for NEXT VALUE FOR
T180		System-versioned tables	
T181		Application-time period tables	
T211		Basic trigger capability	
T211-06		Support for run-time rules for the interaction of triggers and constraints	
T211-08		Multiple triggers for the same event are executed in the order in which they were created in the catalog	intentionally omitted
T251		SET TRANSACTION statement: LOCAL option	
T272		Enhanced savepoint management	
T301		Functional dependencies	partially supported
T321	Core	Basic SQL-invoked routines	
T321-05	Core	RETURN statement	



标识符	核心?	描述	注释
T322		Declared data type attributes	
T324		Explicit security for SQL routines	
T326		Table functions	
T332		Extended roles	mostly supported
T434		GROUP BY DISTINCT	
T471		Result sets return value	
T472		DESCRIBE CURSOR	
T495		Combined data change and retrieval	different syntax
T502		Period predicates	
T511		Transaction counts	
T522		Default values for IN parameters of SQL-invoked procedures	supported except DEFAULT key word in invocation
T561		Holdable locators	
T571		Array-returning external SQL-invoked functions	
T572		Multiset-returning external SQL-invoked functions	
T601		Local cursor references	
T616		Null treatment option for LEAD and LAG functions	
T618		NTH_VALUE function	function exists, but some options missing
T619		Nested window functions	
T625		LISTAGG	
T641		Multiple column assignment	only some syntax variants supported
T652		SQL-dynamic statements in SQL routines	
T654		SQL-dynamic statements in external routines	
T811		Basic SQL/JSON constructor functions	
T812		SQL/JSON: JSON_OBJECTAGG	
T813		SQL/JSON: JSON_ARRAYAGG with ORDER BY	
T814		Colon in JSON_OBJECT or JSON_OBJECTAGG	
T821		Basic SQL/JSON query operators	
T822		SQL/JSON: IS JSON WITH UNIQUE KEYS predicate	
T823		SQL/JSON: PASSING clause	
T824		JSON_TABLE: specific PLAN clause	
T825		SQL/JSON: ON EMPTY and ON ERROR clauses	
T826		General value expression in ON ERROR or ON EMPTY clauses	
T827		JSON_TABLE: sibling NESTED COLUMNS clauses	
T828		JSON_QUERY	

标识符	核心?	描述	注释
T829		JSON_QUERY: array wrapper options	
T830		Enforcing unique keys in SQL/JSON constructor functions	
T838		JSON_TABLE: PLAN DEFAULT clause	
T839		Formatted cast of datetimes to/from character strings	
M001		Datalinks	
M002		Datalinks via SQL/CLI	
M003		Datalinks via Embedded SQL	
M004		Foreign data support	partially supported
M005		Foreign schema support	
M006		GetSQLString routine	
M007		TransmitRequest	
M009		GetOpts and GetStatistics routines	
M010		Foreign data wrapper support	different API
M011		Datalinks via Ada	
M012		Datalinks via C	
M013		Datalinks via COBOL	
M014		Datalinks via Fortran	
M015		Datalinks via M	
M016		Datalinks via Pascal	
M017		Datalinks via PL/I	
M018		Foreign data wrapper interface routines in Ada	
M019		Foreign data wrapper interface routines in C	different API
M020		Foreign data wrapper interface routines in COBOL	
M021		Foreign data wrapper interface routines in Fortran	
M022		Foreign data wrapper interface routines in MUMPS	
M023		Foreign data wrapper interface routines in Pascal	
M024		Foreign data wrapper interface routines in PL/I	
M030		SQL-server foreign data support	
M031		Foreign data wrapper general routines	
X012		Multisets of XML type	
X013		Distinct types of XML type	
X015		Fields of XML type	
X025		XMLCast	
X030		XMLDocument	

标识符	核心?	描述	注释
X038		XMLText	
X065		XMLParse: BLOB input and CONTENT option	
X066		XMLParse: BLOB input and DOCUMENT option	
X068		XMLSerialize: BOM	
X069		XMLSerialize: INDENT	
X073		XMLSerialize: BLOB serialization and CONTENT option	
X074		XMLSerialize: BLOB serialization and DOCUMENT option	
X075		XMLSerialize: BLOB serialization	
X076		XMLSerialize: VERSION	
X077		XMLSerialize: explicit ENCODING option	
X078		XMLSerialize: explicit XML declaration	
X080		Namespaces in XML publishing	
X081		Query-level XML namespace declarations	
X082		XML namespace declarations in DML	
X083		XML namespace declarations in DDL	
X084		XML namespace declarations in compound statements	
X085		Predefined namespace prefixes	
X086		XML namespace declarations in XMLTable	
X091		XML content predicate	
X096		XMLExists	XPath 1.0 only
X100		Host language support for XML: CONTENT option	
X101		Host language support for XML: DOCUMENT option	
X110		Host language support for XML: VARCHAR mapping	
X111		Host language support for XML: CLOB mapping	
X112		Host language support for XML: BLOB mapping	
X113		Host language support for XML: STRIP WHITESPACE option	
X114		Host language support for XML: PRESERVE WHITESPACE option	
X131		Query-level XMLBINARY clause	
X132		XMLBINARY clause in DML	
X133		XMLBINARY clause in DDL	
X134		XMLBINARY clause in compound statements	
X135		XMLBINARY clause in subqueries	
X141		IS VALID predicate: data-driven case	
X142		IS VALID predicate: ACCORDING TO clause	

标识符	核心?	描述	注释
X143		IS VALID predicate: ELEMENT clause	
X144		IS VALID predicate: schema location	
X145		IS VALID predicate outside check constraints	
X151		IS VALID predicate with DOCUMENT option	
X152		IS VALID predicate with CONTENT option	
X153		IS VALID predicate with SEQUENCE option	
X155		IS VALID predicate: NAMESPACE without ELEMENT clause	
X157		IS VALID predicate: NO NAMESPACE with ELEMENT clause	
X160		Basic Information Schema for registered XML Schemas	
X161		Advanced Information Schema for registered XML Schemas	
X170		XML null handling options	
X171		NIL ON NO CONTENT option	
X181		XML (DOCUMENT (UNTYPED)) type	
X182		XML (DOCUMENT (ANY)) type	
X190		XML (SEQUENCE) type	
X191		XML (DOCUMENT (XMLSCHEMA)) type	
X192		XML (CONTENT (XMLSCHEMA)) type	
X200		XMLQuery	
X201		XMLQuery: RETURNING CONTENT	
X202		XMLQuery: RETURNING SEQUENCE	
X203		XMLQuery: passing a context item	
X204		XMLQuery: initializing an XQuery variable	
X205		XMLQuery: EMPTY ON EMPTY option	
X206		XMLQuery: NULL ON EMPTY option	
X211		XML 1.1 support	
X222		XML passing mechanism BY REF	parser accepts BY REF but ignores it; passing is always BY VALUE
X231		XML (CONTENT (UNTYPED)) type	
X232		XML (CONTENT (ANY)) type	
X241		RETURNING CONTENT in XML publishing	
X242		RETURNING SEQUENCE in XML publishing	
X251		Persistent XML values of XML (DOCUMENT (UNTYPED)) type	
X252		Persistent XML values of XML (DOCUMENT (ANY)) type	
X253		Persistent XML values of XML (CONTENT (UNTYPED)) type	

标识符	核心?	描述	注释
X254		Persistent XML values of XML(CONTENT(ANY)) type	
X255		Persistent XML values of XML(SEQUENCE) type	
X256		Persistent XML values of XML(DOCUMENT(XMLSCHEMA)) type	
X257		Persistent XML values of XML(CONTENT(XMLSCHEMA)) type	
X260		XML type: ELEMENT clause	
X261		XML type: NAMESPACE without ELEMENT clause	
X263		XML type: NO NAMESPACE with ELEMENT clause	
X264		XML type: schema location	
X271		XMLValidate: data-driven case	
X272		XMLValidate: ACCORDING TO clause	
X273		XMLValidate: ELEMENT clause	
X274		XMLValidate: schema location	
X281		XMLValidate with DOCUMENT option	
X282		XMLValidate with CONTENT option	
X283		XMLValidate with SEQUENCE option	
X284		XMLValidate: NAMESPACE without ELEMENT clause	
X286		XMLValidate: NO NAMESPACE with ELEMENT clause	
X300		XMLTable	XPath 1.0 only
X305		XMLTable: initializing an XQuery variable	

### F. 3. XML对于SQL的限制和符合性/XML

ISO/IEC 9075-14中与XML相关的规范进行了重大修订 (SQL/XML)与SQL:2006一起被引入。LightDB's 的XML数据类型和相关函数的实现基本上沿用了2003年的早期版本。有一些借鉴了后来的版本。特别是:

- 现行标准提供了一系列可扩展标记语言数据类型, 其中将“文档”或“内容”保存在非类型或XML Schema类型的变体, 以及类型XML(SEQUENCE)用于保存任意的XML内容片段。LightDB提供了一个单一的xml类型, 可以容纳“文档”或“内容”。没有相应的标准的“序列”类型。
- LightDB提供了两个函数在SQL:2006中引入的, 但是在变体中, 使用的是XPath 1.0语言, 而不是在标准中为其指定的XML查询。

本节介绍了一些你可能会遇到的差异。

#### F. 3. 1. 查询只限于XPath 1.0

LightDB-指定的函数 `xpath()` 和 `xpath_exists()` 使用XPath语言查询XML文档。LightDB也提供了标准函数中的XPath-only变体 `XML EXISTS` 和 `XMLTABLE`, 其中正式采用XQuery语言。所有这些函数 LightDB都依赖于 `libxml2` 库, 而这个库仅在XPath 1.0中提供。

在XQuery语言和XPath 2.0及以后的版本之间有很强的联系: 任何语法上有效的表达式, 在这两个版本中都能成功地执行, 都会产生相同的结果 (对于包含数字字符引用或预定义的实

体引用的表达式会略有不同，XQuery会用相应的字符替换，而XPath则不会。)但这些语言和XPath 1.0之间没有这种联系，它是一种较早的语言，在很多方面都有区别。

有两类限制需要记住：一是对SQL标准中指定的函数从XQuery到XPath的限制，二是对标准和LightDB特定函数XPath的限制是1.0版本。

### F.3.1.1. XQuery到XPath的限制

除了XPath的特性之外，XQuery的特性还包括：

- 除了所有可能的XPath值之外，XQuery表达式还可以构造和返回新的XML节点。XPath 可以创建和返回原子类型（数字、字符串等）的值，但只能返回作为表达式输入的文档中已经存在的XML节点。
- XQuery有用于迭代、排序和分组的控制结构。
- XQuery允许声明和使用局部函数。

最近的 XPath 版本开始提供与这些功能重叠的功能（例如函数式的 for-each和sort，匿名函数，以及 parse-xml从字符串中创建节点的parse-xml），但这些功能在 XPath 3.0 之前是不具备的。

### F.3.1.2. 对XPath的限制为1.0

对于熟悉XQuery和XPath 2.0或更高版本的开发人员来说，XPath 1.0带来了许多不同的地方，需要解决的是：

- 一个XQuery/XPath表达式的基本类型，即 sequence，它可以包含XML节点、原子值或两者，在XPath 1.0中不存在。一个1.0表达式只能产生一个节点集（包含0个或更多的XML节点），或者一个原子值。
- 与XQuery/XPath序列不同的是，XPath 1.0节点集没有保证顺序，和任何集一样，不允许同一个项目多次出现。

#### Note

libxml2库似乎总是将节点集返回到LightDB的节点集，其成员在输入文档中的相对顺序是一样的。它的文档并没有承诺这种行为，而且XPath 1.0表达式不能控制它。

- 虽然XQuery/XPath提供了XML Schema中定义的所有类型和许多操作符和函数，但XPath 1.0只有节点集和三种原子类型boolean、double和string。
- XPath 1.0没有条件运算符。一个XQuery/XPath表达式，如if ( hat ) then hat/@size else "no hat"没有XPath 1.0的等价物。
- XPath 1.0没有对字符串进行排序比较运算符。“cat” < “dog”和“cat” > “dog”都是假的，因为每一个都是两个NaN的数值比较。相比之下，=和!=确实将字符串作为字符串进行比较。
- XPath 1.0模糊了XQuery/XPath定义的值比较和一般比较之间的区别。sale/@hatsize = 7和sale/@customer = "alice"都是存在的量化比较，如果有任何sale属性的给定值，则为真。但sale/@taxable = false()是与整个节点集的有效布尔值的值比较。只有当没有sale有一个taxable属性时，它才是真值。
- 在XQuery/XPath数据模型中，一个文档节点既可以有文档形式（即正好有一个顶层元素，只有注释和处理指令以外的注释和处理指令），也可以有内容形式（放宽了这些限制）。

在XPath 1.0中，它的等价物是根节点，只能是文档形式。这也是xml的值被作为上下文项到任何 LightDB的上下文项 基于XPath的函数必须是以文档形式出现。

这里强调的区别并不是全部。在XQuery和XPath的2.0及以后的版本中，有一个XPath 1.0的兼容性模式，W3C列出的 函数库变化<sup>1</sup>和语言变化<sup>2</sup>在该模式下应用的列表提供了一个更完整（但仍然不是详尽的）的差异说明。 兼容性模式不能使后来的语言与XPath 1.0完全等同。

### F.3.1.3. SQL和XML数据类型和值之间的映射

在SQL:2006及以后的版本中，标准SQL数据类型和XML Schema类型之间的转换方向都被精确地指定了。但是，这些规则都是用XQuery/XPath的类型和语义来表示的，对于XPath 1.0的不同数据模型没有直接应用。

当LightDB将SQL数据值映射到XML（如xmlelement），或XML映射到SQL（如输出 xmltable的列），除了少数情况下的特殊处理，LightDB只需假定 XML 数据类型的 XPath 1.0 字符串形式将被视为有效的文本输入形式的SQL数据类型，反之。 该规则的优点是简单，同时对许多数据类型来说，产生的结果与标准中规定的映射类似。

如果与其他系统的互操作性是一个问题，对于某些数据类型，可能需要使用数据类型格式化函数（如Section 9.8中的函数）来生成标准映射。

## F.3.2. 实现中的附带限制

本节涉及的限制并不是libxml2库所固有的，而是适用于LightDB中的当前实现。

### F.3.2.1. 仅支持y VALUE传递机制

SQL标准定义了两种传递机制，适用于从SQL向XML函数传递XML参数或接收结果时。BY REF，其中一个特定的XML 值保留其节点身份，以及BY VALUE，其中传递XML的内容，但不保留节点身份。可以在参数列表之前指定，作为所有参数的默认机制，也可以在任何参数之后指定，以覆盖默认机制。

为了说明区别，如果x是一个XML值，那么在SQL:2006环境下的这两个查询将分别产生true和false。

```
SELECT XMLQUERY('$a is $b' PASSING BY REF x AS a, x AS b NULL ON EMPTY);
SELECT XMLQUERY('$a is $b' PASSING BY VALUE x AS a, x AS b NULL ON EMPTY);
```

LightDB将在XMLEXISTS或XMLTABLE构造中接受BY VALUE或BY REF，但它会忽略它们。 xml数据类型持有字符串的序列化表示，因此没有节点标识需要保留，并且传递总是有效的BY VALUE。

### F.3.2.2. 不能将命名的参数传递到查询中

基于 XPath 的函数支持传递一个参数作为 XPath 表达式的上下文项，但不支持传递额外的值作为命名参数提供给表达式。

### F.3.2.3. 无XML (SEQUENCE) 类型

LightDB xml数据类型只能保存一个值，在DOCUMENT或 CONTENT形式。 一个XQuery/XPath表达式上下文项必须是一个单一的XML节点或原子值，但XPath 1.0进一步限制它只能是一个XML节点，并且没有允许CONTENT的节点类型。 最终的结果是，一个完善的 DOCUMENT 是LightDB可以作为XPath上下文项提供的唯一形式的XML值。

<sup>1</sup> <https://www.w3.org/TR/2010/REC-xpath-functions-20101214/#xpath1-compatibility>

<sup>2</sup> <https://www.w3.org/TR/xpath20/#id-backwards-compatibility>

---

# Appendix G. 版本说明

版本说明包含每个LightDB版本中重大变化，在上面列出的主要特性和迁移问题。版本说明不包含只影响少数用户或者改变是内部的不是用户可见的那些变化。比如，在几乎每一个版本中都会改进优化器。不过这种改进往往要通过查询速度的加快才能被用户所觉察。

## G. 1. 版本发布 13.8–23.3

版本发布日期：. 2023年9月30日

### G. 1. 1. Oracle 兼容

- CREATE SEQUENCE 命令创建序列时支持声明 NOMAXVALUE，兼容Oracle创建序列时 NOMAXVALUE 定义。请参见 CREATE SEQUENCE。

示例：

```
create sequence s1 nomaxvalue;
\d+ s1
```

Sequence "public.s1"						
Type	Start	Minimum	Maximum	Increment	Cycles?	Cache
bigint	1	1	9223372036854775807	1	no	1

- CREATE SEQUENCE 命令创建序列时支持声明 MAXVALUE，兼容Oracle创建序列时 MAXVALUE 定义。请参见 CREATE SEQUENCE。

示例：

```
create sequence s2 maxvalue 100000;
\d+ s2
```

Sequence "public.s2"						
Type	Start	Minimum	Maximum	Increment	Cycles?	Cache
bigint	1	1	100000	1	no	1

- CREATE TABLE 命令创建分区时支持 nocompress/compress 关键字。请参见 CREATE TABLE。

示例：

```
CREATE TABLE lt_oracle_partition_list1
(
  a int,
  b float,
  c date,
  d timestamp,
  e varchar2(20)
) PARTITION BY LIST(e)
(
  PARTITION p1 VALUES ('0001', '0002', '0003', '0004', '0005') compress,
  PARTITION p2 VALUES ('0006', '0007', '0008', '0009') nocompress,
```



```

PARTITION p3 VALUES ('0010', '0011')
);
\d+ lt_oracle_partition_list1

Partitioned table "public.lt_oracle_partition_list1"
Column |          Type          | Collation | Nullable | Default |
Storage | Stats target | Description
-----+-----+-----+-----+-----+
a      | integer      |           |          |         | plain
b      | double precision |           |          |         | plain
c      | oracle.date   |           |          |         | plain
d      | timestamp without time zone |           |          |         | plain
e      | varchar2(20)  |           |          |         |
extended |           |           |          |         |
Partition key: LIST (e)
Partitions: "lt_oracle_partition_list1$p$p1" FOR VALUES IN ('0001', '0002',
'0003', '0004', '0005'),
"lt_oracle_partition_list1$p$p2" FOR VALUES IN ('0006', '0007', '0008',
'0009'),
"lt_oracle_partition_list1$p$p3" FOR VALUES IN ('0010', '0011')

```

- CREATE TABLE 命令创建分区时支持 list+hash 复合分区。请参见 CREATE TABLE。

示例:

```

create table lt_lh_partition(a int, b int)
partition by list(a)
subpartition by hash(b)
(
    partition l_1 values(1),
    partition l_2 values(2)
);
\d+ lt_lh_partition

Partitioned table "public.lt_lh_partition"
Column | Type | Collation | Nullable | Default | Storage | Stats target |
Description
-----+-----+-----+-----+-----+
a      | integer |           |          |         | plain |          |
b      | integer |           |          |         | plain |          |
Partition key: LIST (a)
Partitions: "lt_lh_partition$p$p1" FOR VALUES IN (1), PARTITIONED,
"lt_lh_partition$p$p2" FOR VALUES IN (2), PARTITIONED

\d+ lt_lh_partition$p$p1

Partitioned table "public.lt_lh_partition$p$p1"
Column | Type | Collation | Nullable | Default | Storage | Stats target |
Description

```

```

-----+-----+-----+-----+-----+-----+-----
+-----+
a      | integer |           |           |           | plain |           |
b      | integer |           |           |           | plain |           |
Partition of: lt_lh_partition FOR VALUES IN (1)
Partition constraint: ((a IS NOT NULL) AND (a = 1))
Partition key: HASH (b)
Partitions: "lt_lh_partition$p$l_1_p0" FOR VALUES WITH (modulus 1, remainder
0)

```

- CREATE TABLE 命令支持了一些选项，如compress/nocompress, logging/nologging。这些选项在历史版本使用上有顺序限制，该版本允许这些选项可以任意顺序出现。请参见 CREATE TABLE。
- 禁止 CREATE TABLE 创建的表名和 CREATE PACKAGE 创建的包名重复。
- ALTER TABLE 命令支持一次添加多列时可以把列定义放在要括号里。请参见 ALTER TABLE。

示例：

```

CREATE TABLE foo(a int);
ALTER TABLE foo ADD COLUMN a1 int;
ALTER TABLE foo ADD COLUMN a2 int default 3;
ALTER TABLE foo ADD COLUMN b1 int, ADD COLUMN b2 int;
ALTER TABLE foo ADD (c1 int);
ALTER TABLE foo ADD (c2 int default 3);
ALTER TABLE foo ADD (e1 int), ADD e2 int, ADD COLUMN e3 int, ADD (e4 int, e5
int);
begin
    execute immediate 'alter table foo add(d int)';
end;
/

```

- ALTER TABLE 命令支持 TRUNCATE PARTITION partition\_name 时指定 update indexes。请参见 ALTER TABLE。

示例：

```

create table part_update_indexes_range(id int primary key)
PARTITION by range(id) (
    partition p1 VALUES LESS THAN (202000) nocompress,
    partition p2 VALUES LESS THAN (202001) compress,
    partition p3 VALUES LESS THAN (202002)
);

ALTER TABLE part_update_indexes_range TRUNCATE PARTITION p1 update
indexes;

```

- DROP TABLE 命令支持通过 cascade constraints 级联删除该表所有的约束。请参见 DROP TABLE。

示例：

```
CREATE TABLE student(id int, name varchar2(100));

ALTER TABLE student ADD CONSTRAINT uk_student unique(id);

\d student
Table "public.student"
Column |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
id     | integer        |           |          |
name   | varchar2(100)  |           |          |
Indexes:
"uk_student" UNIQUE CONSTRAINT, btree (id)

DROP TABLE student cascade constraints;
```

- 新增 CREATE TYPE name { AS | IS } OBJECT 语法，支持创建对象类型。请参见 CREATE TYPE。

示例：

```
select dbms_output.serveroutput(true);
CREATE TYPE stock AS OBJECT (
    stock_no int,
    price    numeric(16,2),
    member procedure dis(prompt varchar)
);

CREATE TYPE BODY stock AS
member procedure dis(prompt varchar) is
begin
    dbms_output.put_line(prompt || ':' || self.stock_no || '-' ||
self.price);
end;
END;
/

DECLARE
    s stock := stock(600570, 180.00);
BEGIN
    s.dis('current');
END;
/

current:600570-180.00
DO
```

- 新增 TO\_CLOB 函数，通过该函数我们将能够将数据从文本字符串类型转换为 CLOB 类型。请参见 orafce。

示例：

```
CREATE TABLE testorafce_to_clob (
    col_char CHAR(10),
    col_varchar2 VARCHAR2(20),
    col_varchar VARCHAR(20),
```

```

col_nchar NCHAR(10),
col_nvarchar2 NVARCHAR2(20),
col_smallint smallint,
col_integer integer,
col_bigint bigint,
col_decimal decimal,
col_numeric numeric,
col_real real,
col_double double precision,
col_clob CLOB,
col_raw raw(10)
);

```

```

INSERT INTO testorafce_to_clob (col_char, col_varchar2, col_varchar,
col_nchar, col_nvarchar2, col_smallint, col_integer, col_bigint, col_decimal,
col_numeric, col_real, col_double, col_clob, col_raw)
VALUES ('ABC1', 'Hello', 'world', 'hello', 'world', 1, 2, 3, 4, 5, 6, 7, 'This
is a CLOB', 'AB');

```

```

INSERT INTO testorafce_to_clob (col_char, col_varchar2, col_varchar,
col_nchar, col_nvarchar2, col_smallint, col_integer, col_bigint, col_decimal,
col_numeric, col_real, col_double, col_clob, col_raw)
VALUES ('ABC2', 'Hello', 'world', 'hello', 'world', 1, 2, 3, 4, 5, 6, 7, 'This
is a CLOB', '1');

```

```

INSERT INTO testorafce_to_clob (col_char, col_varchar2, col_varchar,
col_nchar, col_nvarchar2, col_smallint, col_integer, col_bigint, col_decimal,
col_numeric, col_real, col_double, col_clob, col_raw)
VALUES ('ABC3', 'Hello', 'world', 'hello', 'world', 1, 2, 3, 4, 5, 6, 7,
to_clob('This is a CLOB'), '1AB456789');

```

```

SELECT to_clob(col_char) AS clob_char,
to_clob(col_varchar2) AS clob_varchar2,
to_clob(col_varchar) AS col_varchar,
to_clob(col_nchar) AS clob_nchar,
to_clob(col_nvarchar2) AS clob_nvarchar2,
to_clob(col_clob) AS clob_clob,
to_clob(col_smallint) AS col_smallint,
to_clob(col_integer) AS col_integer,
to_clob(col_bigint) AS col_bigint,
to_clob(col_decimal) AS col_decimal,
to_clob(col_numeric) AS col_numeric,
to_clob(col_real) AS col_real,
to_clob(col_double) AS col_double,
to_clob(col_raw) AS clob_nclob

```

```

FROM testorafce_to_clob order by col_char asc;

```

```

clob_char | clob_varchar2 | col_varchar | clob_nchar | clob_nvarchar2 |
clob_clob | col_smallint | col_integer | col_bigint | col_decimal |
col_numeric | col_real | col_double | clob_nclob
-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
ABC1      | Hello        | world      | hello     | world      | This
is a CLOB | 1           | 2         | 3         | 4         | 5
| 6         | 7         | AB

```

```

ABC2      | Hello      | world      | hello      | world      | This
is a CLOB | 1          | 2          | 3          | 4          | 5
| 6        | 7          | 01         |            |            |
ABC3      | Hello      | world      | hello      | world      | This
is a CLOB | 1          | 2          | 3          | 4          | 5
| 6        | 7          | 01AB456789|            |            |
(3 rows)

```

- 新增 XMLType 对象类型，支持 XMLType.getClobVal 方法。请参见 orafce。

示例：

```

CREATE TABLE xml_table (id NUMBER, xml_column XMLType );

INSERT INTO xml_table (id, xml_column) VALUES (1, XMLType('<root><name>John</name></root>'));

DECLARE
    v_clob CLOB;
BEGIN
    SELECT XMLType.getClobVal(xml_column) INTO v_clob FROM xml_table WHERE id
    = 1;
    DBMS_OUTPUT.PUT_LINE('CLOB value: ' || v_clob);
END;
/
CLOB value: <root><name>John</name></root>
DO

DECLARE
    xml_data XMLType;
    clob_data CLOB;
BEGIN
    xml_data := XMLType('<root><name>John Doe</name></root>');
    clob_data := xml_data.getClobVal();
    DBMS_OUTPUT.PUT_LINE(clob_data);
END;
/
<root><name>John Doe</name></root>
DO

SELECT XMLType.getClobVal(xml_column) from xml_table;
getclobval
-----
<root><name>John</name></root>
(1 row)

```

- 新增 lightdb\_oracle\_sql\_mode GUC参数，用于支持Oracle模式下列名显示默认转为大写。请参见 lightdb\_oracle\_sql\_mode。

示例：

```
create table std(no int, name varchar2(32));
```

```
insert into std values(1, 'zhangsan');
```

```
select * from std;
no | name
----+-----
1 | zhangsan
(1 row)
```

```
set lightdb_oracle_sql_mode to 'true';
```

```
select * from std;
NO | NAME
----+-----
1 | zhangsan
(1 row)
```

- ltsql 命令新增支持Oracle SQL\*Plus的 RENAME old\_name TO new\_name 命令，可以对表名进行重命名。请参见 RENAME 命令。
- ltsql 命令新增支持Oracle SQL\*Plus SPOOL命令。请参见 SPOOL 命令。
- Oracle模式下支持 binary\_float 和 binary\_double，都遵循IEEE754标准。请参见 orafce。
- Oracle模式下支持 '\0' 可以正常插入 char(n) 类型字符串中。

示例：

```
create database oradb with lightdb_ascii_zero_store_value 20
lightdb_syntax_compatible_type 'oracle';
\c oradb
```

```
create table t00(a char(1), b char(2));
```

```
insert into t00(a,b) values(E'\0',E'\0');
```

```
insert into t00(a,b) values(chr(0),chr(0));
```

```
select * from t00;
```

```
a | b
----+-----
\x14 | \x14
\x14 | \x14
(2 rows)
```

'\0' 转为了lightdb\_ascii\_zero\_store\_value值进行存储

- Oracle模式下日期作差结果兼容Oracle，返回numeric类型。
- col视图兼容Oracle，width字段与Oracle保持兼容。
- greatest/least兼容Oracle。请参见 GREATEST和LEAST。

主要实现了以下兼容特性：

只要有一个参数为 NULL 结果就返回 NULL。

把第一个参数作为被比较的类型，其余参数类型强制转换为第一个参数的类型。

当遇到一个参数为 NULL 时，不再解析其余参数。

- orafce 中的所有Oracle函数改成默认safe的，默认支持启用并行处理。

## G. 1. 2. plorasql 增强

- PL/oraSQL 函数支持事务提交回滚。请参见 针对Oracle的增强功能。
- PL/oraSQL 函数支持 inout 参数。请参见 声明函数参数。
- PL/oraSQL 函数支持DML语句。请参见 在PL/oraSQL中执行DML。
- PL/oraSQL 存储过程中支持使用CREATE DOMAIN创建游标类型的变量。

示例：

```
SELECT dbms_output.serveroutput(true);
CREATE TABLE employees(employee_id int, first_name varchar2(100), last_name
    varchar2(100));
INSERT INTO employees values(1500, 'first15', 'last15');
CREATE DOMAIN mycursor AS refcursor;
CREATE OR REPLACE PROCEDURE get_employee_name AS
    employee_cursor mycursor;
    v_employee_name VARCHAR2(50);
BEGIN
    OPEN employee_cursor FOR SELECT first_name FROM employees WHERE
    employee_id = 1500;
    LOOP
        FETCH employee_cursor INTO v_employee_name;
        EXIT WHEN employee_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(' Employee Name: ' || v_employee_name);
    END LOOP;
    CLOSE employee_cursor;
END;
/
call get_employee_name ();
Employee Name: first15
CALL
```

## G. 1. 3. MySQL 兼容

- 新增 INSERT IGNORE 功能。请参见 INSERT 。

示例：

```
create database test_mysql with lightdb_syntax_compatible_type mysql;
\c test_mysql

create table t(a int primary key, b int);
insert ignore into t(a,b) values(1,1), (1,1), (2,2);
select * from t;
a | b
---+---
1 | 1
```

2 | 2  
(2 rows)

- 新增 INSERT 支持对自增列插入NULL语法，每次插入都会根据序列进行自增。 请参见 INSERT NULL说明。
- 新增 JSON 函数集合。 请参见 myfce JSON函数。 函数清单如下：

JSON\_ARRAY

JSON\_OBJECT

JSON\_EXTRACT

JSON\_CONTAINS

JSON\_CONTAINS\_PATH

JSON\_PRETTY

JSON\_REMOVE

JSON\_INSERT

JSON\_REPLACE

JSON\_SET

- 新增 lightdb\_mysql\_lower\_case\_table\_names 参数。 使用 create database 语句创建数据库时，可以指定参数来设置是否开启大小写敏感功能。 请参见 CREATE DATABASE。
- varchar 类型兼容 boolean 类型数据。 请参见 myfce。

## G.1.4. lightdb 新特性

- 新增 lt\_upgrade\_check 升级前检查工具，用于提前检查升级不兼容及异常项，尽量避免在升级过程中出现失败。请参见 lt\_upgrade\_check。
- 新增 ECPG 代码中支持使用 ‘-’ 和 ‘//’ 注释。 请参见 ECPG。
- 新增服务端、客户端支持GBK, GB18030-2022, 已通过人行国标认证。 请参见 本地化。
- 新增 lt\_createdb 行为在CN分布式下默认传播到DN节点。 请参见 lt\_createdb。
- 新增 lightdb\_cursor\_after\_commit GUC参数来控制fetch行为。 lightdb\_cursor\_after\_commit 为 off 时不能在一个事务提交之后再去做 fetch操作这个结果集； 为 on 时一个事务提交之后，这个游标还可以继续使用。 请参见 lightdb\_cursor\_after\_commit。
- 新增 distinct connect by level 语法支持。
- 新增 connect\_by\_isleaf 语法支持。 请参见 SELECT。
- ltsql 增强支持提示当前服务数据库的兼容模式。
- lt\_upgrade 增强支持跨版本升级。
- ltloader 增强性能，达到Oracle sqlldr的80%以上。 请参见 ltldr。
- lt\_dump 增强。 请参见 lt\_dump。 清单如下：

--lt-exclude-lightdb-objects 支持过滤内置用户 lightdb 和 lt\_probackup。

--lt-exclude-lightdb-objects 支持导出 ACCESS METHOD。

lt\_dump 和 lt\_dumpall 增加 --lt-target-version 选项，用于指定导出的内容所适用的目标版本。

导出默认的创建模版支持为 templatel。

create database 语句支持 lightdb\_syntax\_compatible\_type 选项逻辑。



- mysql\_fdw 增强支持查询超过64K的字段。

## G.1.5. ltjdbc 增强

- ltjdbc 支持 numeric 使用 Short 类型。
- ltjdbc 支持setNull传入 Types.CLOB 和 Types.BLOB 。
- ltjdbc 支持整型数值使用 string 类型。
- ltjdbc 支持函数 xmlparse。

## G.2. 版本发布 13.8-23.2

版本发布日期： . 2023年6月30日

### G.2.1. Oracle 兼容

- CHAR 类型支持指定字节或字符长度。当使用CHAR 类型定义表中的列时，可以指定列值的长度。请参见 字符类型。

示例:

```
create table t (id char (1 char));
create table t (id char (1 byte));
```

- 层级查询增强。connect by 用于处理层次结构数据，已经支持了通过 LEVEL 关键字显示分层查询中的层级数，SYS\_CONNECT\_BY\_PATH 显示分层查询中的层级结构。当前版本对层级查询增加了新的功能，CONNECT\_BY\_ROOT 显示当前行的根节点，connect by level < n, 显示小于 n 层级的所有递归结果。此外，层级查询中支持了子查询，关联查询，表达式计算等。请参见 CONNECT BY 子句。

示例:

```
SELECT employee, CONNECT_BY_ROOT employee_id as "Manager",
       SYS_CONNECT_BY_PATH(employee, '/') "Path"
FROM example
START WITH manager_id is not null
CONNECT BY PRIOR employee_id = manager_id
ORDER BY employee_id, manager_id;
```

employee	Manager	Path
Josh	2	/Josh
Ali	3	/Ali
Ali	2	/Josh/Ali
Kyle	5	/Kyle

(4 rows)

- ALTER TABLE 增强。支持 ALTER TABLE 删除同一列的索引和约束。请参见ALTER TABLE。

示例:

```
ALTER TABLE my_table
DROP CONSTRAINT my_constraint,
```

```
DROP INDEX my_index;
```

- 支持外连接 + 号表示 LEFT JOIN 和 RIGHT JOIN 。请参见oracle (+)。

示例:

```
SELECT *
FROM table1, table2
WHERE table1.id = table2.id (+);
```

- 使用 USING INDEX 子句来指定使用索引来支持约束，在创建约束的时候同时为约束指定一个索引。请参见 ALTER TABLE。

示例:

```
ALTER TABLE employees
ADD CONSTRAINT emp_id_pk
PRIMARY KEY (id)
USING INDEX;
```

- 支持 DELETE 子句中，FROM 可选。请参见 DELETE。

示例:

```
DELETE table_name WHERE col = 1;
```

- 支持 SYS.COL 系统视图，它显示了所有表的列信息，包括列名、数据类型、长度、精度、是否为空、默认值等信息。请参见 orafce。

示例:

```
create table test_col_view(key0 varchar(100), key1 number not null, key2
number(38,0));
```

```
select * from col where tname= 'TEST_COL_VIEW' order by colno;
```

tname	colno	cname	coltype	width	scale	precision	nulls
defaultval	character_set_name						
TEST_COL_VIEW	1	KEY0	VARCHAR	100			NULL
TEST_COL_VIEW	2	KEY1	NUMBER				NOT NULL
TEST_COL_VIEW	3	KEY2	NUMBER	38,0	0	38	NULL

(3 rows)

- 支持 INSERT ALL 语法，用于一次性插入多条记录到一个或多个表中。请参见INSERT。

示例:

```
INSERT ALL
  INTO table_name (column1, column2, column3) VALUES (value1, value2, value3)
  INTO table_name (column1, column2, column3) VALUES (value1, value2, value3)
SELECT 1 FROM DUAL;
```

- INSERT INTO 语句支持使用别名来指定要插入数据的表。请参见 INSERT。

示例:

```
INSERT INTO employees e (employee_id, first_name, last_name)
VALUES (1001, 'John', 'Doe');
```

- merge into 支持分布式。请参见 MERGE INTO。

示例:

```
MERGE INTO target t
USING source s
ON t.tid = s.sid
WHEN MATCHED THEN
    update set balance = t.balance + s.delta
WHEN NOT MATCHED THEN
    INSERT VALUES(s.sid, s.delta);
```

- 支持 NUMBER 和 BOOLEAN 类型隐式转换。请参见 orafce。

示例:

```
select 0::NUMBER = TRUE;
?column?
```

```
-----
f
(1 row)
```

```
select 1::NUMBER = TRUE;
?column?
```

```
-----
t
(1 row)
```

- 增强分区表功能。分区命名兼容 Oracle，并且支持对子分区进行操作。请参见 ALTER TABLE。

示例:

```
alter table lt_truncate_t_r truncate partition p1;
alter table lt_truncate_t_r add partition p2 values less than(30);
alter table lt_truncate_t_r truncate partition p2;
alter table lt_truncate_t_r drop partition p2;
```

- 创建表时支持 STORAGE, compress / nocompress 和 logging / nologging 关键字，仅是语法糖功能，兼容 Oracle 语法，没有实现对应的功能。请参见 CREATE TABLE。

示例:

```
create table comp_log1(id int) compress;
create table comp_log3(id int) logging;
create table comp_log5(id int) compress logging;
create table comp_log13(id int) storage(abcd);
```

- 支持 prompt 命令，用于向用户显示文本消息。请参见 prompt 命令。

示例:

```
PROMPT Creating table...
```

```
CREATE TABLE test_table (  
    id NUMBER PRIMARY KEY,  
    name VARCHAR2(50)  
);
```

```
PROMPT Inserting data...
```

```
BEGIN;  
INSERT INTO test_table (id, name) VALUES (1, 'John Doe');  
INSERT INTO test_table (id, name) VALUES (2, 'Jane Smith');  
COMMIT;
```

```
PROMPT Querying data...
```

```
SELECT * FROM test_table;  
DROP table test_table;  
PROMPT Done.
```

- 支持 ALTER INDEX idx\_name REBUILD 语法，用于重建指定索引。该命令可以帮助我们优化索引性能，修复索引损坏或减少索引碎片。请参见 ALTER INDEX。

示例:

```
CREATE TABLE tt(a int);  
CREATE INDEX idx_tt_a on tt(a);  
ALTER INDEX idx_tt_a REBUILD;
```

- sequence 语句支持 order / noorder / nocycle / nocache 关键字。注意，这个功能仅是语法糖，没有实现实际功能。请参见 CREATE SEQUENCE。

示例:

```
create sequence lt_seq_test1 order start 100;  
create sequence lt_seq_test3 start 100 nocycle;  
create sequence lt_seq_test5 start 100 ORDER nocache NOCYCLE;  
alter sequence lt_seq_test1 nocache nocycle noorder;
```

- 支持 create force view。注意，这个功能仅仅是语法糖，没有实现实际功能。请参见 CREATE VIEW。

示例:

```
CREATE FORCE VIEW force_view as SELECT 1 as ltapk, 2 as ltaut;
```

- 支持在 ecpg 中使用绑定参数的匿名执行块。请参见 执行一个带输入输出参数的 Do Statement。

示例:

```
EXEC SQL EXECUTE
```

```

begin
  begin
    update/*3631412*/ t1 set c = 'C' where id = :v_error_no;
    exception
      when others then
        :iReturnCode = 456;
        rollback;
      when DUP_VAL_ON_INDEX then
        null;
    if SQL%NOTFOUND then
      null;
    end if;

  end;
end;
END-EXEC;

```

## G.2.2. plorasql 增强

- plorasql 支持 sqlcode。用户可以像在 Oracle 中那样创建返回“sqlcode”值的函数。请参见 支持 SQLCODE。

例如，当出现零除法错误时，以下代码将输出 SQL 错误代码：

```

create or replace function zero_divide_f return int as
  a int := 10;
  b int := 0;
  code int;
begin
  a := a / b;
  exception
    when zero_divide then
      code := sqlcode;
      dbms_output.put_line(' this is a normal error ' || code || ' ' || sqlcode
|| ' ' || sqlcode_to_sqlstate(code) || ' ' || sqlcode_to_sqlstate(sqlcode));
      return 1;
end;
/

```

- plorasql 中支持使用 with 子句。用户可以使用 with 子句在 SQL 语句中定义临时结果集。

示例：

```

declare
  v_count varchar(100);
begin
  with tmp as (select * from company where id=2) select * into v_count from
  tmp;
  DBMS_OUTPUT.PUT_LINE(v_count);
end;
/

```

- plorasql 支持 PRAGMA AUTONOMOUS\_TRANSACTION 语法解析。注意，这里仅是语法糖功能，兼容 Oracle 语法，没有实现对应的功能。请参见 AUTONOMOUS TRANSACTION Pragma。

示例：

```
CREATE OR REPLACE PROCEDURE test_pragma_procedure()
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  NULL;
END;
/
```

- plorasql 支持游标变量的大小写不敏感，这意味着在打开游标时指定的游标变量的大小写与声明时的大小写不匹配也不会出现语法错误。请参见 Cursors 。

示例：

```
DECLARE
  CURSOR D_tmp(zxc NUMBER, bnm NUMBER, ff smallint default 1234) IS
    SELECT ff || a as feiwu FROM tt;
  ddd D_tMp%ROWTYPE;
BEGIN
  OPEN D_tmP(5, 6);
  FETCH d_tmp INTO ddd;
  DBMS_OUTPUT.PUT_LINE(ddd.feiwu);
  CLOSE D_tmP;
end;
/
```

- plorasql 支持 %TYPE 引用包中变量的类型。请参见 声明 。

示例：

```
create table t_binjiang(a int, b int);
-- define package
create or replace package pkgtype is
  v_a int := 0;
  v_b int := 0;
end;
/

-- anonymous block references variables in the package
declare
  a pkgtype.v_a%type := 1;
  b pkgtype.v_b%type := 1;
begin
  insert into t_binjiang values(a, b);
end;
/
```

- plorasql 支持 goto 语句。用户可以使用 goto 语句跳转到块中的特定标签。请参见 GOTO 语句 。

示例：

```
DO $$
BEGIN
  RAISE NOTICE 'out block 1';
  goto testlabel;

```

```

RAISE NOTICE 'out block 2';
<<testlabel>>
BEGIN
    RAISE NOTICE 'in block 1';
    goto testlabel2;
    RAISE NOTICE 'in block 2';
END;

RAISE NOTICE 'out block 3';

<<testlabel2>>
RAISE NOTICE 'out block 4';

END$$;
NOTICE: out block 1
NOTICE: in block 1
NOTICE: out block 4

```

- plorasql 支持分布式场景下调用包函数。请参见 Packages 。

示例:

```

create or replace package typkg
as
    type nt is table of tab1%rowtype;
    res nt;
    function myfunc(v int) return nt;
end;
/

create or replace package body typkg
as
    function myfunc(v int) return nt
    is
    begin
        res(1) := ROW(1,1);
        res(2) := ROW(2,2);
        res(3) := ROW(3,3);
        return res;
    end;
end;
/

select * from table(typkg.myfunc(1)) t0
where t0.a in (select a from tab1) order by t0.a;

```

### G. 2. 3. MySQL 兼容

- 支持 DROP INDEX 语法 DROP INDEX [IF EXISTS] index\_name ON table\_name; 请参见 DROP INDEX 。

示例:

```

create table t1(id int);
create unique index idx_t1_id on t1(id);
drop index idx_t1_id on t1;

```

- 查询列时大小写不敏感、显示列大小写敏感。在 `mysql` 模式下，`guc` 参数 `lightdb_sql_mode` 设置 `uppercase_identifier`。请参见 `SELECT`。

示例：

```
create table test_case_1(identifier1 varchar(10));
```

```
select * from test_case_1;
identifier1
-----
```

(0 rows)

```
select Identifier1 from test_case_1;
Identifier1
-----
```

(0 rows)

```
select identifier1 as Identifier1 from test_case_1;
Identifier1
-----
```

(0 rows)

- 支持 `INSERT IGNORE INTO` 语法。`INSERT IGNORE INTO` 语句用于向表中插入数据，如果插入的数据违反了唯一性约束（如主键或唯一索引），则忽略这些数据，不会抛出错误或警告。请参见 `INSERT`。

示例：

```
INSERT IGNORE INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

- 支持通过 `COMMENT` 关键字可以用于向数据库对象（如表、列等）添加注释信息。请参见 `CREATE TABLE`。

示例：

```
create table t1 (
    id int primary comment 'this is id of t1' not null,
    name varchar(10) not null comment 'this is name of t1'
) comment = 'this is t1';
```

- 支持 `json` 相关函数。请参见 `myfcee`。

`JSON_ARRAY`：创建一个 `JSON` 数组。

`JSON_OBJECT`：创建一个 `JSON` 对象。

`JSON_EXTRACT`：从 `JSON` 数据中提取值。

`JSON_CONTAINS`：检查 `JSON` 数据是否包含指定的值。

`JSON_CONTAINS_PATH`：检查 `JSON` 字符串中是否存在指定路径。

`JSON_PRETTY`：用于格式化 `JSON` 字符串的函数。它将一个未格式化的 `JSON` 字符串作为输入，并将其格式化为易于阅读的形式。

- 支持 `sysdate()` 函数，用于取当前的执行时间。请参见 `当前日期/时间`。



示例:

```
select sysdate();
      sysdate
-----
2023-05-09 13:21:13
(1 row)
```

- 修复问题 `mysql` 模式下 `dual` 虚拟表, 普通用户没权限的问题。请参见 `myfce` 。

示例:

```
create user mytestuser with password 'mytestuser';
\c - mytestuser
select 2 from dual;
?column?
-----
      2
(1 row)
```

## G. 2. 4. lightdb 新特性

- PWR 准确测量实时查询各类 IO 事件, IPC 事件, 竞态等时间开销, 并优化了 PWR 性能报告的统计。请参见 `lt_profile` 。
- `lt_hint_plan` 对不支持 `hint` 的场景进行优化, 忽视未实现的 `hint`。请参见 `lt_hint_plan` 。
- 支持增量检查点, 减少触发检查点时对数据库系统的性能影响。请参见 `CHECKPOINT` 。
- 支持在 `pgadmin/navicat` 中调试 PL/pgsql 存储过程。请参见 `pldebugger` 。
- PL/pgsql 支持 `execute immediate` 语法。注意, 这里仅是语法糖功能, 没有实现对应的功能。请参见 `执行动态命令` 。

示例:

```
CREATE OR REPLACE FUNCTION get_employee_name_func(p_employee_id INTEGER)
RETURNS VARCHAR AS $$
DECLARE
    sql_query TEXT;
    emp_name VARCHAR;
BEGIN
    sql_query := 'SELECT first_name FROM test_employees WHERE employee_id = $1';
    EXECUTE IMMEDIATE sql_query INTO emp_name USING p_employee_id;

    RAISE NOTICE 'Employee name is: %', emp_name;
    RETURN emp_name;
END;
$$ LANGUAGE plpgsql;
```

- 发布客户端包, 解压配置环境变量后, 用户可以直接使用相关的工具。包括如下工具。

`lt_distributed_probackup.py`

`lt_distributed_dump.py`

`lt_distributed_restore.py`

`ltuldr`

`lt_basebackup`

```
lt_dump
lt_dumpall
lt_isready
lt_probackup
lt_restore
lt_verifybackup
ltsql
```

## G. 2.5. ltjdbc 增强

- `java.sql.ResultSet#getBlob(java.lang.String)` 支持读取字节数组格式的 blob 值。请参见 LightDB JDBC 。
- `ltjdbc` 支持直接执行 `create procedure`、`create function` 等语法创建存储过程和函数。请参见 LightDB JDBC 。
- `java.sql.ResultSet#getObject(int, java.lang.Class<T>)` 支持将 `date` 列类型转换为 `LocalDateTime`。请参见 LightDB JDBC 。

## G. 3. 版本发布 13.8-23.1

版本发布日期: . 2023年3月30日

### G. 3.1. 新增功能

- 当内存被分配时出现内存不足问题，添加详细信息和提示。
- LightDB 支持 Kubernetes(K8S) 的高可用部署。核心模块包括 `LightDB-patroni` 镜像和 `LightDB-operator` 镜像。`LightDB-patroni` 镜像包含 `LightDB` 和 `Patroni`，以支持容器中的高可用部署。`LightDB-operator` 镜像用于在 Kubernetes 中管理 `LightDB` 集群。
- 新增 `guc` 参数 `lightdb_server_version_num` 用于标识 `LightDB` 版本号。此值无法在会话级别下进行修改，需在 `lightdb.conf` 配置文件中进行设置。请参见 `SHOW`。

显示 `LightDB` 服务器版本号：

```
SHOW lightdb_server_version_num;
 lightdb_server_version_num
-----
220400
(1 row)
```

- 新增扩展 `lt_sql_inspect`，用于定义事务时段、拦截和审计交易期间执行的危险 SQL 和命令。请参见 `lt_sql_inspect`。

### G. 3.2. 优化

`lt_probackup` 支持分布式优化。请参见 `lt_distributed_probackup`。

- 在脚本中添加了分布式备份 ID。

- 您可以通过分布式备份 ID 查看备份的总体状态。

## G. 3. 3. 兼容性

### G. 3. 3. 1. MySQL 兼容

- 支持函数 `substr` 和 `substring`。请参见 `function`。

例如：

```
lightdb@postgres=# create database test_createdb_mysql with
  lightdb_syntax_compatible_type mysql;
CREATE DATABASE
lightdb@postgres=# \c test_createdb_mysql
You are now connected to database "test_createdb_mysql" as user "lightdb".
lightdb@test_createdb_mysql=# select substr('sadaytyui',-3,3) FROM DUAL;
 substr
-----
      yui
(1 row)
```

- 支持 MySQL 的除法，包括整数除以整数，除以 0 等操作。请参见 `myfce`。

例如：

```
lightdb@test_createdb_mysql=# select 4/0;
WARNING:  division by zero
 ?column?
-----
(1 row)
lightdb@test_createdb_mysql=# select 4/1;
 ?column?
-----
4.000000000000000000
(1 row)
```

- 支持 `str_to_date` 函数。请参见 `myfce`。
  - 不支持使用变量绑定的 Prepare Statements。
  - 由于使用 Julian 日来表示时间，无法表示非法日期 1999-00-00。
  - 当 `lightdb_sql_mode` 的值为 `no_zero_date` 时，返回的值为空字符串，而非 NULL。
  - 报告非法日期错误，使用 LightDB 的错误报告逻辑。

例如：

```
select str_to_date('2022/09/10 11,13,13', '%Y/%m/%d %h,%i,%s');
 str_to_date
-----
2022-09-10 11:13:13
(1 row)
```

- 支持 @ 变量。请参见 User Variable。

- 如果变量未初始化，则变量的类型为 `text`。
- 如果变量类型为 `float` 或 `float8`，则存储类型为 `float8`。
- 如果变量的值是 `integer` 类型 (`smallint`、`bigint`、`int`)，则存储类型为 `bigint`。
- 如果变量的类型是 `numeric`，则存储类型为 `numeric`。
- 如果变量的类型是其他类型，则变量类型为 `text`。

用户变量写作 `@var_name`，其中变量名 `var_name` 必须是标识符或关键字。如果您将其引用为标识符，`var_name` 将被视为列名（例如，`@"my-var"`）。`@var_name` 和 `@ var_name` 是不同的。`@var_name` 是用户变量表达式，`@ var_name` 是列 `var_name` 的绝对值。

```
lightdb@test=# select @c1 := 5;
```

```
@c1
-----
   5
(1 row)
```

```
lightdb@test=# SELECT c1, @c1, @ c1 FROM t;
```

```
 c1 | @c1 | ?column?
-----+-----+-----
  0 |   5 |         0
  1 |   5 |         1
(2 rows)
```

- 支持在查询时指定索引，仅支持语法。 `force index` 强制使用指定的索引； `ignore index` 忽略指定的索引。请参见 `SELECT`。

```
-- multiple index
select * from lt_test_mysql_ddl use
index(pk_lt_test_mysql_ddl,uk_lt_test_mysql_ddl);
select * from lt_test_mysql_ddl force
index(pk_lt_test_mysql_ddl,uk_lt_test_mysql_ddl);
select * from lt_test_mysql_ddl ignore
index(pk_lt_test_mysql_ddl,uk_lt_test_mysql_ddl);

-- multiple table join
select * from lt_test_mysql_ddl a use index for order by(primary) join b
using(id);
select * from lt_test_mysql_ddl a force index for order
by(pk_lt_test_mysql_ddl) join b using(id);
select * from lt_test_mysql_ddl a ignore index for order
by(primary,pk_lt_test_mysql_ddl) join b using(id);
select * from lt_test_mysql_ddl a ignore index for order
by(pk_lt_test_mysql_ddl) join b using(id);
```

- 支持在创建表时指定 `engine`、`collate`、`character set`、`comment` 等属性，其中 `engine` 和 `comment` 仅支持语法。兼容反引号 ( `'` ) 和 `tinyint` 类型兼容。请参见 `Integer Types`。
- 支持 `duplicate key update` 语法，当 `PRIMARY KEY` 或 `UNIQU` 约束被违反时，进行更新操作，反之亦然。请参见 `ON DUPLICATE KEY UPDATE`。

```
INSERT INTO t1 (a, b, c) VALUES (1, 2, 3) ON DUPLICATE KEY UPDATE c=t1.c+1;

UPDATE t1 SET c=c+1 WHERE a=1;
```

- 支持 replace into 语法，当 PRIMARY KEY 或 UNIQU 约束被违反时，先删除旧元组，然后插入新元组。请参见 REPLACE INTO。

```
lightdb@testdb=# REPLACE INTO test VALUES (1, 'Old name', '18');
INSERT 0 1
```

```
lightdb@testdb=# REPLACE INTO test VALUES (1, 'New name');
INSERT 0 1
```

```
lightdb@testdb=# SELECT * FROM test;
 id |      name      | age
----+-----+----
  1 | New name      |   0
(1 row)
```

### G.3.3.2. Oracle 兼容

- 添加了 no\_unnest 和 pq\_distribute 优化提示。请参阅 lt\_hint\_plan。

no\_unnest 优化提示。

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where
exists(select * from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Hash Join
  Hash Cond: (test1.key1 = test2.key1)
    -> Seq Scan on test1
    -> Hash
          -> HashAggregate
                Group Key: test2.key1
                -> Seq Scan on test2
```

(7 rows)

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where
exists(select /*no_unnest*/ * from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Seq Scan on test1 @lt#1"
  Filter: (alternatives: SubPlan 1 or hashed SubPlan 2)
  SubPlan 1
    -> Seq Scan on test2 @lt#0"
          Filter: (test1.key1 = key1)
  SubPlan 2
    -> Seq Scan on test2 test2_1 @lt#0"
```

(7 rows)

pq\_distribute 优化提示。

```
lightdb@lt_test=# /*set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
```

```
lightdb@lt_test-# EXPLAIN (COSTS false) SELECT /*+ leading(t1 t2) hashjoin(t1
t2)*/ * FROM t1 join t2 on t1.id=t2.id;
```

QUERY PLAN

```
-----
Gather
  Workers Planned: 3
  -> Parallel Hash Join
      Hash Cond: (t1.id = t2.id)
      -> Parallel Seq Scan on t1 @"lt#0"
      -> Parallel Hash
          -> Parallel Seq Scan on t2 @"lt#0"
```

(7 rows)

```
lightdb@lt_test=# /*+set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
```

```
lightdb@lt_test-# EXPLAIN (COSTS false) select/*+leading(t1 t2) hashjoin(t1
t2) pq_distribute(t2 none broadcast)*/ * from t1 join t2 on t1.id=t2.id;
```

QUERY PLAN

```
-----
Gather
  Workers Planned: 3
  -> Hash Join
      Hash Cond: (t1.id = t2.id)
      -> Parallel Seq Scan on t1 @"lt#0"
      -> Hash
          -> Seq Scan on t2 @"lt#0"
```

(7 rows)

- 支持 Oracle 的 create user ( CREATE USER) 语法； 支持级联删除用户 ( DROP USER) 和 grant 语句 (详见 示例)。

例如：

```
CREATE USER user_name IDENTIFIED BY passwd DEFAULT TABLESPACE dt_space
TEMPORARY TABLESPACE tt_space;
DROP USER user_name CASCADE;
GRANT CONNECT TO HS_SES;
GRANT RESOURCE TO HS_SES;
GRANT UNLIMITED TABLESPACE TO HS_SES;
```

- 支持全局临时表。 详情请参见 兼容性。
  - 支持Oracle全局临时会话表。
  - 支持Oracle全局临时事务表。

创建一个会话级别的全局临时表。

```
CREATE GLOBAL TEMPORARY TABLE gtt_session(id number,ename varchar(15)) ON
COMMIT PRESERVE ROWS;
```

创建一个事务级别的全局临时表。

```
CREATE GLOBAL TEMPORARY TABLE gtt_transaction(id number,ename varchar(15))
ON COMMIT DELETE ROWS;
```

- 自动创建一个与数据库同名的用户，自动创建的同名用户在所创建的数据库中具有 create、connect、temporary、login 权限，该功能不能在分布式系统中使用。详情请参见 CREATE DATABASE。

```
lightdb@postgres=# create database testdb with lightdb_syntax_compatible_type
oracle;
NOTICE: auto create user "testdb" success
CREATE DATABASE
```

- 当可支持变量类型声明的长度时，它可以支持字符或字节，当单位目前为字符时，仅支持语法。详情请参见 orafce。

```
create table hs_es(id int, name varchar2(10 byte));
create table hs_lt(id int, name varchar2(10 char));
```

- 增加导入工具 ltlldr，与 sqlldr 工具兼容。通过控制文件将数据从指定表导入数据库，可以同时导入多个表。对于使用 sqlldr 工具导入数据的业务，具有天然的优势。
- 兼容 DBMS\_UTILITY 内置包 format\_error\_backtrace 函数，仅记录 PloraSQL 执行错误的最新记录，使用此函数可以查看错误位置。详情请参见支持 DBMS\_UTILITY 内置包 format\_error\_backtrace。

使用 DBMS\_UTILITY.format\_error\_backtrace 函数，例如：

```
create or replace procedure pr_error_code_test1 is
    v_backtrace    varchar(1000);
    v_error        varchar(1000);
begin
    INSERT INTO test_error_stack values (2);
    INSERT INTO test_error_stack values (1);
    commit;
    exception when others then
        insert into t_errlog(operate_error_msg) select
to_char(DBMS_UTILITY.format_error_backtrace);
        commit;
        rollback;
    end pr_error_code_test1;
/

select select to_char(DBMS_UTILITY.format_error_backtrace)
```

- PloraSQL 支持存储过程使用未装饰的返回参数和返回 null。返回 null 函数用于终止存储过程。详情请参见从存储过程返回。

```
create or replace procedure p_fmt_post_update_cash
(
    p_src_system  varchar2 DEFAULT '0',
    p_init_date   number  DEFAULT 0,
    p_company_no  varchar2 DEFAULT '0',
    o_return_msg  out  varchar2,
    o_return_code out  integer
) as
    v_init_date   int := 0;
```

```

begin
  if v_init_date = 0 then
    o_return_code := 1;
    o_return_msg  := '请输入正确的校验日期';
  return;
  end if;
end;
/

```

## G. 4. 版本发布 13.8-22.4

发布日期: . 2022年12月27日

- 添加 `guc` 参数 `lightdb_arch_mode`, 有两个值。 `classic` 表示当前数据系统不是分布式架构。 `dist` 表示当前数据系统是分布式架构, 在此架构下, 使用 `create table` 创建的表默认为分布式表。 详情请参见 兼容性。
- `lt_check` 是一个 LightDB 检查 `sha256sum` 的工具。 确保 LightDB 安装目录中的文件完整。 详情请参见 `lt_check`。
- 新增检测无条件和无分页查询的功能。 详情请参见 LightDB DML 规范检测。

在以下场景中, 会打印提示消息, 指示用户应该使用分页/where:

- `select` 语句未指定 `limit`、`offset` 或 `where` 。
- `update` 语句未指定 `where`。
- `delete` 语句未指定 `where`。

例如:

```

lightdb@lt_test=# set lightdb_best_practice_level to 'warn';
SET
lightdb@lt_test=# set lightdb_best_practice_mode to 'no_paging';
SET
lightdb@lt_test=# create table dc_np_t1(c_a int primary key, c_b varchar,
gmt_create timestamp null, gmt_modified timestamp null);
CREATE TABLE
lightdb@lt_test=# SELECT * from dc_np_t1;
WARNING: LightDB DML check warning! select sql not where/limit/offset clause!
  c_a | c_b | gmt_create | gmt_modified
-----+-----+-----+-----
(0 rows)

```

- 检查清单中的每个项目的正确性。 它可用于检测操作系统参数、LightDB 安装文件 `sha256sum`、LightDB 版本号等是否存在风险。 如果存在风险, 则提供风险报告。 详情请参见 `check_list`。
- 支持在 Oracle 和 Postgres 模式下均可使用 `database()` 函数。 `database()` 函数位于扩展 `ltfce` 中。 详情请参见 `ltfce`。

例如:

```

lightdb@postgres=# select database();
database
-----

```



```

public
(1 row)
lightdb@postgres=# \c test_createdb_oracle
You are now connected to database "test_createdb_oracle" as user "lightdb".
lightdb@test_createdb_oracle=# show lightdb_dblevel_syntax_compatible_type;
 lightdb_dblevel_syntax_compatible_type
-----
Oracle
(1 row)
lightdb@test_createdb_oracle=# select database();
 database
-----
public
(1 row)

```

- 优化归档日志清理。详情请参见 `lightdb_archive_dir`。
- 优化归档日志清理。详情请参见 `lightdb_archive_dir`。
- 在启动LightDB时，当`archive_mode`打开时，请检查`archive_command`、`lightdb_archive_retion_size`和`lightdb_archive_dir`。
- 调整参数：
  - `lightdb_archive_retion_size`只能在启动时设置。
  - `lightdb_archive_dir`只能在启动时设置。
- 确保在归档中删除的WAL日志比`lt_wal`目录中保留的日志数少。
- 添加`walminer`扩展来挖掘WAL日志的所有有用信息，以提供LightDB数据恢复支持。数据库日志级别配置需要大于`minimal`。请参阅 `lt_walminer`。
- 支持单实例多模式。请参阅 `CREATE DATABASE`。

例如：

```

lightdb@postgres=# create database test_createdb_oracle with
 lightdb_syntax_compatible_type oracle;
CREATE DATABASE
lightdb@postgres=# \c test_createdb_oracle
You are now connected to database "test_createdb_oracle" as user "lightdb".
lightdb@test_createdb_oracle=# show lightdb_dblevel_syntax_compatible_type;
 lightdb_dblevel_syntax_compatible_type
-----
Oracle
(1 row)

```

- 添加`lightdb_aborted_rollback`参数，以支持在事务中报告当前语句的错误并回滚整个事务。请参阅 `lightdb_aborted_rollback`。

在添加此功能之前，例如：

```

lightdb@lt_test=# begin;
BEGIN
lightdb@lt_test=## create table aborted_rollback(name varchar(100));
CREATE TABLE
lightdb@lt_test=## insert into aborted_rollback1 values(1);

```

```
ERROR: relation "aborted_rollback1" does not exist
LINE 1: insert into aborted_rollback1 values(1);
          ^

lightdb@lt_test=# insert into aborted_rollback1 values(1);
ERROR: current transaction is aborted, commands ignored until end of
transaction block
lightdb@lt_test=# commit;
ROLLBACK
```

添加此功能后，例如：

```
lightdb@lt_test=# set lightdb_aborted_rollback to on;
SET
lightdb@lt_test=# begin;
BEGIN
lightdb@lt_test=# create table aborted_rollback(name varchar(100));
CREATE TABLE
lightdb@lt_test=# insert into aborted_rollback1 values(1);
ERROR: relation "aborted_rollback1" does not exist
LINE 1: insert into aborted_rollback1 values(1);
          ^

ERROR: current transaction is aborted, "insert into aborted_rollback1
values(1);" causes transaction rollback.
```

- 解决unknown类型参与运算时的类型不匹配问题。请参阅 `optional`。

例如：

```
lightdb@lt_test=# select '1'+1;
?column?
-----
          2
(1 row)
```

Note: Operator overloading may affect the return value of the expression. For example, for an expression such as `select '1' * 1`, it previously returned an `int` type, but now it returns a numeric type.

- 优化`lt_cron`进程的问题，减少CPU使用率。请参阅 `lt_cron`。
- 添加了一个选项`table_exists_action`，用于告诉`lt_restore`如果正在尝试创建的表已经存在该怎么办。`table_exists_action`有四个选项：`skip`、`append`、`truncate`或`replace`。请参阅 `lt_restore`。
- `lt_upgrade` 支持程序包升级。请查看 `lt_upgrade`。
- `lt_upgrade_check` 支持程序包跨版本升级检查。请查看 `lt_upgrade_check`。
- 语句级回滚优化，此特性在Oracle和MySQL两种模式下都支持。启用了语句级回滚，性能损失约为5%。请查看 `lt_statement_rollback_enable`。
- `ltdts_ora` 是一种用于将Oracle数据同步到LightDB的数据同步工具。请查看 `ltdts_ora`。
- `ltdts_mysql` 是一种用于将MySQL|MariaDB数据同步到LightDB的数据同步工具。请查看 `ltdts_mysql`。
- `lt_patch` 已进行增强，并添加了下面展示的许多新参数。请查看 `lt_patch`。

- 支持自动执行SQL脚本。
- 添加了 `lt_patch` 表以保存补丁信息。 补丁描述信息中添加了 `apply_type` 和 `apply_version` 属性。
- 添加了补丁依赖目录。
- 解决了补丁成功进程中的半状态问题。
- 更为严格的验证和其他细节优化。
- 使用 `lt_dump` 和 `lt_restore` 工具实现分布式集群的备份和恢复：
  - 使用 `lt_dump` 工具实现分布式集群的备份。 请查看 `lt_distributed_dump`。
  - 使用 `lt_restore` 工具实现分布式集群的恢复。 请查看 `lt_distributed_restore`。
- 使用 `lt_probackup` 工具实现分布式集群的备份和恢复。 请查看 `lt_distributed_probackup`。
- Oracle 兼容性增强，并添加了下面展示的许多新功能。
  - 从普通表在线更改为分区表。 请查看 `ALTER TABLE`。

例如：

```
lightdb@test_createdb_oracle=# CREATE TABLE lt_oracle_partition_range5
lightdb@test_createdb_oracle=# (
lightdb@test_createdb_oracle(# a int,
lightdb@test_createdb_oracle(# b float,
lightdb@test_createdb_oracle(# c date,
lightdb@test_createdb_oracle(# d timestamp
lightdb@test_createdb_oracle(# )PARTITION BY RANGE (c)
lightdb@test_createdb_oracle=# (
lightdb@test_createdb_oracle(# PARTITION p1 VALUES LESS
lightdb@test_createdb_oracle(#   THAN(to_date(' 2022-01-01', 'yyyy-mm-dd')),
lightdb@test_createdb_oracle(#   PARTITION p2 VALUES LESS
lightdb@test_createdb_oracle(#   THAN(to_date(' 2022-02-01', 'yyyy-mm-dd')),
lightdb@test_createdb_oracle(#   PARTITION p3 VALUES LESS
lightdb@test_createdb_oracle(#   THAN(to_date(' 2022-03-01', 'yyyy-mm-dd'))
lightdb@test_createdb_oracle(# );
CREATE TABLE
lightdb@test_createdb_oracle=#
lightdb@test_createdb_oracle=# \d+ lt_oracle_partition_range5
          Partitioned table
"public.lt_oracle_partition_range5"
Column |          Type          | Collation | Nullable | Default |
Storage | Stats target | Description
-----+-----+-----+-----+-----+
a      | integer          |           |          |         |
plain  |                 |           |          |         |
b      | double precision  |           |          |         |
plain  |                 |           |          |         |
c      | date              |           |          |         |
plain  |                 |           |          |         |
d      | timestamp without time zone |           |          |         |
plain  |                 |           |          |         |
```

```
Partition key: RANGE (c)
Partitions: lt_oracle_partition_range5_1_prt_p1 FOR VALUES FROM (MINVALUE)
            TO ('2022-01-01'),
            lt_oracle_partition_range5_1_prt_p2 FOR VALUES FROM
            ('2022-01-01') TO ('2022-02-01'),
            lt_oracle_partition_range5_1_prt_p3 FOR VALUES FROM
            ('2022-02-01') TO ('2022-03-01')
```

- 支持 Oracle 的 cardinality 和 ordered\_predicates 提示。请参见 lt\_hint\_plan。

例如：

```
lightdb@postgres=# explain select /*+ cardinality(b 1000)use_nl(a b)
leading(a b)*/ from t1 a , t2 b where a.key1=b.key1;
                QUERY PLAN
```

```
-----
Nested Loop  (cost=0.15..433.43 rows=1000 width=16)
  -> Seq Scan on t1 a @"lt#0"  (cost=0.00..32.60 rows=2260 width=8)
  -> Index Scan using t2_pkey on t2 b @"lt#0"  (cost=0.15..0.18 rows=1
width=8)
      Index Cond: (key1 = a.key1)
(4 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select /*+ordered_predicates*/
from test1 where mod(key1,10)=2 and key2=2;
                QUERY PLAN
```

```
-----
Seq Scan on test1 @"lt#0"
  Filter: ((mod(key1, 10) = 2) AND (key2 = 2))
(2 rows)
```

- 修改表的时候，可以交换默认值和非空属性的位置。请参见 ALTER TABLE。

例如：

```
lightdb@lt_test=# create table aborted_rollback(name varchar(100));
CREATE TABLE
lightdb@lt_test=# alter table aborted_rollback modify name text not null
default 1;
ALTER TABLE
```

- 新增导出工具 ltuldr（与 Oracle 的 sqluldr2 工具兼容）。请参见 ltuldr。
- 支持 connect by 层次查询。请参见 CONNECT BY Clause。

例如：

```
lightdb@test_createdb_oracle=# SELECT id, manager_id from t_tab CONNECT BY
prior id = manager_id order by id;
id | manager_id
```

```
-----+-----
1 |          0
2 |          1
2 |          1
3 |          2
```

```

3 |          2
3 |          2
(6 rows)

```

- orafce 的改进，详情请参见 orafce。
- varchar2 计算字符长度。
- 支持 raw 类型。

例如：

```

lightdb@test_createdb_oracle=# create table test_type(name varchar2(100), id
raw);
CREATE TABLE
lightdb@test_createdb_oracle=# \d test_type
          Table "public.test_type"
Column |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
name   | varchar2(100)  |           |          |
id     | raw             |           |          |

```

## G. 5. 版本发布 13.8-22.3

LightDB 13.8-22.3包含了从 LightDB 13.3-22.2 中修复的各种问题和新功能。

发布日期： . 2022-09-30

- 支持使用 q 转义。例如：

```

select q'!name LIKE '%DBMS_%%!';
       ?column?
-----
name LIKE '%DBMS_%!'
(1 row)

```

详情请参见 Q-Quoted String Constants。

- Rownum 性能优化。与 Oracle 类似，支持 count stop key 以提高 rownum 的性能。

```

lightdb@lt_test=# explain select * from public.test where rownum < 3;
          QUERY PLAN
-----
Count StopKey  (cost=0.00..0.03 rows=2 width=4)
-> Seq Scan on test  (cost=0.00..159.75 rows=11475 width=4)
(2 rows)

```

- libpq 支持获取 LightDB 版本号。
- listagg 支持分析函数。在 orafce 章节中有关于 listagg 的描述。
- NVL 函数参数支持更多的参数组合，如字符串和数字。
- 将默认数据库修改为 lt\_test，替换原来的默认数据库 postgres。

- `lt_upgrade` 支持从版本 22.2 升级到版本 22.3 的 LightDB（在升级时，`lt_upgrade` 将检查 `lightdb_syntax_compatible_type` 的一致性，请注意，如果这两个值都是 'oracle'，则不建议使用 `lt_upgrade`）。请参见 `lt_upgrade`。
- 新增 GUC 参数 `nls_date_format` 和 `nls_timestamp_format`，以实现与 Oracle 时间函数的兼容性，详情请参见 `nls_date_format` 和 `nls_timestamp_format`。
- `search_path contains lt_catalog by default.` see Search Path.

处理 `oracle` 和 `mysql` 时需要特殊处理，只能在它们与 `lightdb_syntax_compatible_type` 一致时进行设置。

- 支持将非保留关键字作为列别名（当没有关键字 `as` 和引号时）。请参见 `关键字`。

```
select 3 type;
 type
-----
      3
(1 row)
```

- 当启用 `lightdb_enable_default_tablespace` 时，添加默认表空间 `lt_users`。`lt_users` 是我们自己创建的其他数据库的默认表空间。（除非在 `CREATE DATABASE` 中使用 `TABLESPACE` 子句进行覆盖）。其存储路径与实例路径处于同一级别的目录中。这可以确保实例路径下的数据量不会太大。请参见 `表空间`。
- `lightdb_service.py` 支持任意单节点启动和停止。
- 已修复 `lt_probackup` 恢复失败的问题，当数据量超过 2G 时。
- 支持通过 `modify column` 子句修改列，与 Oracle 类似。详见 `ALTER TABLE`。
- 支持通过 `drop primary key` 子句在 `alter table` 时删除主键。详见 `ALTER TABLE`。
- 新增函数和表，用于修改具有视图依赖性的表。详见 `orafce`。
- 移除只读表相关功能。
- `lt_dump` 进行了增强，添加了许多新参数，如下所示。详见 `lt_dump`。
  - 添加参数 `--lt-exclude-lightdb-objects` 和参数 `--lt-dump-lightdb-tables`，解决了在导出和导入过程中 LightDB 内置插件创建的数据库对象冲突问题，并解决了在导入过程中覆盖由 LightDB 内置插件创建的表的问题。
  - 添加参数 `--lt-disable-auto-analyze`，以解决 `lt_restore` 多线程导入死锁的问题。
- 增强了 PL/oraSQL，并添加了下面许多新功能。
  - Package 支持使用 `lt_dump` 进行备份。请参阅 `lt_dump`。
  - Package 支持关联数组。请参阅 第 42.8 节。
  - 支持 `table function`。请参阅 `表函数`。
  - 支持嵌套表。请参阅 `嵌套表`。
  - 支持 `error handling`。请参阅 `错误处理`。
  - 支持在打开游标后的存储过程（包括 `package` 中的存储过程）和匿名块。在事务块中，您可以在 `commit` 或 `rollback` 后继续进行取出操作。请参阅 `游标和事务`。

- 修复了打开游标时参数与定义的游标参数不一致的问题。 请参阅 游标默认值。
- orafce 增强:
  - 添加新的 Oracle 内置包: dbms\_metadata、dbms\_job、dbms\_lock、dbms\_obfuscation\_toolkit、utl\_url、utl\_encode、utl\_raw 和 dbms\_snapshot。
  - 改进了原有的内置包: dbms\_application\_info、dbms\_utility 和 dbms\_lob。
  - 添加了新的 Oracle 系统视图, 如 [DBA/ALL/USER]\_TABLES、 [DBA/ALL/USER]\_TAB\_COLUMNS 和 [DBA/ALL/USER]\_INDEXES。
  - 添加了新的 Oracle 函数: BIN\_TO\_NUM、BITOR/BITXOR、REMAINDER、ROUND\_TIES\_TO\_EVEN、INSTRB、REGEXP\_INSTR/REGEXP\_REPLACE、REPLACE、SOUNDEX、MONTHS\_BETWEEN、TZ\_OFFSET、CONVERT、NUMTOYMINTERVAL、TO\_BLOB、NLS\_CHARSET\_ID/NLS\_CHARSET\_NAME、SYS\_CONTEXT/USERENV、EMPTY\_CLOB、ORA\_HASH 和 VSIZE。
  - 添加了新的 Oracle 聚合函数: ANY\_VALUE、BIT\_AND\_AGG/BIT\_OR\_AGG/BIT\_XOR\_AGG、KURTOSIS\_POP/KURTOSIS\_SAMP 和 SKEWNESS\_POP/SKEWNESS\_SAMP。

请参阅 orafce 了解详情。

- Oracle 兼容性增强, 以及下面列出的许多新功能。
  - 支持 Oracle 分区语法。 请参阅 CREATE TABLE 和 ALTER TABLE。
  - 支持语句回滚。 请参阅 lt\_statement\_rollback\_enable。
  - 支持 raise\_application\_error() 函数。 请参阅 第42.8节。
  - listagg 支持分析函数。 请参阅 orafce 章节。
  - 增强空字符串和 NULL 值之间的兼容性。 请参阅 第17.11节。
- MySQL 兼容性增强, 以及下面列出的许多新功能。
  - 支持 year() 函数。
  - 支持 day() 函数。
  - 支持 date\_format() 函数。
  - 支持 str\_to\_date() 函数。
  - 支持 from\_unixtime() 函数。
  - 支持 unix\_timestamp() 函数。
  - 支持 ifnull() 函数。
  - 增加 interval 常量的兼容性。
  - 支持 date\_add() 函数。
  - 支持 date\_sub() 函数。
  - 支持 dayofweek() 函数。
  - 支持 weekofyear() 函数。
  - 增加 cast 类型转换。

- 支持 `compress()` 函数。
- 支持 `uncompress()` 函数。
- 支持 `timestamp` 和 `string` 的比较。
- 修复在使用 `'is true'` 或 `'= true'` 时可能出现的 `smallint` 错误。
- 修复 `select and group by field inconsistent report error`。
- 修复 `distinct and order by field inconsistent report error`。

有关详细信息，请参见 `myfce` 章节。

- JDBC驱动库，我们修复了一些问题并增强了数据类型兼容性：
  - 支持将 `numeric` 数据类型转换为 `Java Integer`。
  - 添加连接参数 `extendSchema`，当前模式为当前模式设置时，默认值为 `'lt_catalog,oracle'`，`extendSchema`将自动附加到当前模式的值。
  - 移除过期时间限制。
  - 仅在顺序执行时支持MySQL的 `SQL_CALC_FOUND_ROWS` 和 `SELECT FOUND_ROWS()` 特性。

## G. 6. 版本发布 13.3-22.2

发布日期：. 2022年6月30日

- 支持快速升级的补丁，请参见 `lt_patch`。
- 添加可加载的PL/oraSQL过程语言，请参见 `plorasql`。
- `lt_hint_plan`增强。
  - 添加三个新提示：`use_hash`、`use_nl`、`full`，语义与Oracle一致。
  - 提示中的表名不区分大小写。
  - 提示 `parallel` 的默认值是`hard`，并且不支持表规范。
  - 当子查询只有一个表时，提示可以应用于子查询的表。

```
EXPLAIN (COSTS false) select /*+use_hash(t1 y)*/ from t1, (select * from t2)
y where t1.id=y.id;
```

### QUERY PLAN

```
-----
Hash Join
Hash Cond: (t1.id = t2.id)
-> Seq Scan on t1 @"lt#0"
-> Hash
      -> Seq Scan on t2 @"lt#1"
(5 rows)
```

- 支持用户定义类型优先级以获得更好的函数和运算符匹配，请参见 `类型兼容`。
- 支持使用 `CREATE TABLE DISTRIBUTED BY` 创建分布式表（依赖于扩展`canopy`），请参见 `CREATE TABLE`。



```
create table dist1(a int, b int) distributed by hash(a) shard_count(4);
```

- Auto-explain执行计划日志以JSON格式打印在单独的日志文件(\*.slow)中。
- 添加扩展 wal2sql, 支持与Oracle的实时同步。
- lt\_dump 支持 --recreate-schema, 以生成删除模式 (使用 CASCADE 模式) 命令以获得更好的性能, 请参见 lt\_dump。
- 客户端应用程序和 libpq支持 LightDB环境变量。
- sysdate和systemtimestamp 表示语句开始时间。

```
lightdb@postgres=# begin;
BEGIN
lightdb@postgres=# select sysdate from dual;
      sysdate
-----
2022-06-28 19:44:15
(1 row)

lightdb@postgres=# select sysdate from dual;
      sysdate
-----
2022-06-28 19:44:16
(1 row)

lightdb@postgres=# end;
COMMIT
```

有关详细信息, 请参见 datetime。

- 安装时启用 高可用性, 默认启用 复制槽。
- 支持保序加密, 请参阅 lt\_ope。
- 新增以下平台支持:

架构	操作系统
龙芯架构 <sup>1</sup>	麒麟V10 (SP1) <sup>2</sup>
x86_64	Rocky Linux <sup>3</sup>
aarch64	Rocky Linux <sup>4</sup>

- 添加集群控制工具lightdb\_service.py, 以方便在高可用性和分布式环境中启动和停止服务。

启动集群:

```
lightdb_service.py -c start
```

<sup>1</sup> <https://www.loongson.cn/>

<sup>2</sup> <https://www.kylinos.cn/>

<sup>3</sup> <https://rockylinux.org/>

<sup>4</sup> <https://rockylinux.org/>

停止集群:

```
lightdb_service.py -c stop
```

- PWR中的值0显示为空, 请参阅 `lt_profile` 了解详细信息。
- 在非分布式安装中, 未安装 `canopy` 扩展。
- 重新启动 `keepalived` 和 `ltcluster` 时, 将生成新的日志文件, 以避免与旧日志混淆。
- `lt_initdb` 不限制选项的顺序。
- 修复在 `lt_show_plans` 中进行并行查询时, 偶尔会出现 `cannot assign XIDs during a parallel operation` 错误的问题。
- 发布基于 PostgreSQL 的 JDBC 驱动程序库。
  - Maven 仓库

```
<dependency>  
  <groupId>io.github.hslightdb</groupId>  
  <artifactId>lightdb-jdbc</artifactId>  
  <version>42.2.30</version>  
</dependency>
```

我们修复了一些问题并增强了数据类型兼容性:

- 修复: 在自动提交模式下, `fetchSize` 不会生效, 导致整个数据被加载到内存中, 引发内存不足问题。
- 支持 Oracle 匿名块语法 (请参阅 `plorasql`) 。
- 支持与 Oracle 兼容的 `to_number()` 函数。
- 支持将 `numeric` 数据类型转换为 Java `Long`。
- 支持将 `char(1)` 数据类型转换为 Java `Character`。
- 修复 Java `setDouble` 在 `numeric` 字段上操作时精度丢失的问题。
- `lt_cron` 支持 Linux 命令。

```
SELECT cron.schedule('daily-touch', '59 23 * * *', 'rm -rf $LTDATA/log/*',  
  'next', '8', 'linux');
```

- 内置工具应用程序 `lvs`, 用于在分布式多CN环境中进行负载均衡。
- 内置工具应用程序 `postgresrest`。
- 在安装目录和实例目录中更改了文件命名规则。
- `public` 模式中的扩展对象都被移动到 `lt_catalog` 模式中。
- RPM 依赖项已经打包进了安装包中。
- 在分布式环境中支持 `merge into`。

## G. 7. 版本发布 13.3-22.1

发布日期: . 2022年4月6日

此版本相对于LightDB 13.3-21.3包含了多种更改。

- 增强了Oracle兼容性，并添加了下面许多新功能。
- 如果在SELECT FROM子句中只使用了一个子查询，则不需要使用AS子句来显式指定子查询的别名。有关更多详细信息，请参见 SELECT。
- 您可以在UPDATE和DELETE语句的WHERE子句中使用ROWNUM作为过滤条件来指定要更新或删除的行，例如：

```
UPDATE mytable SET column = value WHERE ROWNUM = rownum;
DELETE FROM mytable WHERE ROWNUM = rownum;
```

您还可以使用ROWNUM为表的每一行分配唯一值，例如：

```
UPDATE mytable SET column = ROWNUM;
```

要使用此功能，您需要将 lightdb\_syntax\_compatible\_type 设置为 oracle，例如：

```
SET lightdb_syntax_compatible_type = 'oracle';
```

有关更多详细信息，请参见 Section 7.9

- 支持MERGE语法，MERGE使用data\_source 对target\_table\_name中的行进行修改操作。MERGE提供了一个单一的SQL语句，可以有条件地INSERT或UPDATE 行，这是一项 否则需要多个过程语言语句才能完成的任务。有关更多详细信息，请参见 MERGE。
- 支持SYSTIMESTAMP函数，SYSTIMESTAMP返回所在数据库系统的日期，包括小数秒和时区。返回类型为TIMESTAMP WITH TIME ZONE。例如：

```
SELECT SYSTIMESTAMP FROM DUAL;
SELECT TO_CHAR(SYSTIMESTAMP, 'SSSS.FF') FROM DUAL;
```

有关更多详细信息，请参见 Section 9.9.4

- 如果您已将 lightdb\_syntax\_compatible\_type 设置为 oracle，则表达式“value || NULL”的结果将是value，否则结果将为NULL。
- 如果您已将 lightdb\_syntax\_compatible\_type 设置为 oracle，则表达式“’’:text is NULL”的结果将为true，否则结果将为false。
- orafce扩展中的LISTAGG函数现在支持DISTINCT和WITHIN GROUP语法，您可以像这个例子一样使用它：

```
SELECT LISTAGG(DISTINCT column, ',') WITHIN GROUP (ORDER BY
column) FROM mytable;
```

有关更多详细信息，请参见 orafce。

- 为SELECT语句添加了新关键字MINUS，并作为EXCEPT的别名，它们是完全等价的。有关更多详细信息，请参见 SELECT。 2495
-

- 已删除GUC参数`lightdb_enable_rowid`和`lightdb_enable_rownum`，并将`lightdb_syntax_compatible_type`设置为`oracle`代替。
- `pg_hint_plan`增强了，添加了几个新的提示，例如`USE_HASH_AGGREGATION`、`Semijoin`、`Antijoin`和`Swap_join_inputs`。此外，它还实现了对分布式数据库的`canopy`支持。有关更多详细信息，请参见`pghint_plan`。
- 支持透明数据加密(TDE)。只能通过在使用`initdb`或`lt_initdb`生成数据目录时指定`-K`选项来启用TDE，该选项接受一个32字节的十六进制密钥字符串，用于加密和解密。有关更多详细信息，请参见`encryption key command`。
- 添加新扩展模块`lt_sm`，以支持`sm2`、`sm3`和`sm4`加密。`lt_sm`是LightDB的一个扩展模块，提供了一系列函数，用于`sm2`、`sm3`和`sm4`加密算法。可以使用上述函数加密和解密数据。有关更多详细信息，请参见`lt_sm`。
- 添加了新的GUC参数`lightdb_tsearch_non_stopwords`和`lightdb_tsearch_word_superpose`，用于全文搜索。

LightDB全文搜索有许多内置的停用词。由停用词连接的词将被`to_tsvector`函数拆分为多个独立的单词，例如，“2022-04-10”将被拆分为“2022”、“04”和“10”。

`lightdb_tsearch_non_stopwords` 用于自定义非停用词，它与停用词相反，即非停用词被视为单词的一部分，因此不会被`to_tsvector`函数拆分为多个独立的单词。仍然以上面的例子为例，如果将“-”设置为非停用词，则只会得到一个“2022-04-10”的结果。

`lightdb_tsearch_word_superpose` 用于覆盖使用停用词和非停用词的效果。例如，如果将`lightdb_tsearch_word_superpose`设置为`on`，并将`lightdb_tsearch_non_stopwords`设置为“-”，在使用`to_tsvector("2022-04-10")`时将同时得到包括“2022”、“04”、“10”和“2022-04-10”的结果。

有关更多详细信息，请参见 Section 12.1.3

- 添加新的GUC参数 `lt_ring_buffers` ，用于设置在每个数据库会话中用于环形缓冲区的最大内存量。当运行需要访问大量页面(超过`shared_buffers`的总页面数量的四分之一)的顺序扫描查询时，会使用一种特殊的访问策略称为`bulk-read`。只有被这种扫描所触及的页面不太可能再次被需要，因此，不会清除整个缓冲区，而是分配一个由`lt_ring_buffers`指定的小环形缓冲区，并将这些缓冲区重复使用于整个扫描过程中。这也意味着这种语句所引起的大部分写入流量将由后端本身完成，而不是推到其他进程上。
- `pg_profile`增强，添加了名为`DBCpu`的等待事件类型(`wait_event_type`)，它表明用户活动在数据库上花费了多少CPU时间。有关更多详细信息，请参见`pgprofile`。
- `pg_cron`增强，支持三种任务调度模式：`asap`、下一个时间间隔和固定时间间隔。
  - `asap`: 每个任务仅创建一个会话并对应一个任务队列，多个任务可以存储在队列中。如果后续调度周期在当前任务执行期间到来，则它们将被放入队列中。同时，当前执行将逐个完成队列中的所有任务。
  - `next interval`: 每个任务仅创建一个会话并对应一个任务队列，只有一个任务存储在队列中。如果后续调度周期在当前任务执行期间到来，则它们不会被放入队列中，而是在当前任务执行后的下一个调度周期。
  - `fixed interval`: 此模式类似于`next interval`，但区别在于，如果后续调度周期在当前任务执行期间到来，则将为后续调度创建单独的会话以执行，即多个调度任务将并行执行，但最大并行执行数为`cron.max_connections_per_task`。

另外，还添加了任务超时功能。如果任务执行时间超过指定的最大时间，则任务将被强制中断，并且`cron.job_run_details`将记录相关的错误消息。默认超时时间为15秒，由`cron.task_running_timeout`确定。此参数只在数据库启动时生效。

详见 `lt_cron`。

- 添加新的GUC参数`pg_prewarm.lt_autoprewarm_dbs`、`pg_prewarm.lt_autoprewarm_tables`和`pg_prewarm.lt_autoprewarm_indexes`用于`pg_prewarm`扩展，在LightDB启动时自动选择性预热指定的数据库、表和索引。详见 `lt_prewarm`。
- Canopy扩展现在默认启用。
- 新的`pg_show_plans`扩展已集成，默认情况下处于禁用状态。详见 `lt_show_plans`。
- 执行`CREATE INDEX`时自动执行`ANALYZE`以收集统计信息。
- `UPDATE`和`DELETE`语句允许使用`LIMIT`子句来限制将被更新和删除的行数，例如：

```
UPDATE mytable SET column = value LIMIT n;
DELETE FROM mytable WHERE column > value LIMIT m, n;
```

详见 `UPDATE`和 `DELETE`。

- 如果您使用函数（包括内置函数和用户定义函数）来指定`CREATE TABLE`语句中的列的默认值，则函数参数可以是其他列名、变量名和包含其他列名和变量名的表达式，例如：

```
CREATE FUNCTION myfunc(v integer)
RETURNS integer AS $$
BEGIN
    RETURN v + 10;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE mytable (
    id   int   default 10,
    key  int   default myfunc(id),
    v1   int   default key,
    v2   int   default key + myfunc(key)
);
```

详见 `CREATE TABLE`。

- 当使用“`CREATE TABLE AS SELECT colname AS alias`”创建表时，不允许使用`ltapk`作为别名。详见 `CREATE TABLE`。
- `lt_standby_forward` (EXPERIMENTAL) 是一个模块，它将SQL语句从热备机转发到主机，提供了一种减轻主机负载的方法，并允许客户端从热备机查询。例如，客户端可以将`INSERT`、`CREATE`发送到热备机。详见 `lt_standby_forward`。
- `ltsql`的`sqlprompt`格式已更改为`username@database`。

## G. 8. 13. 3-21. 3版本发布

发布日期：. 2021-12-31

本次发布与LightDB 13. 3-21. 2相比包含了各种变更。

- WAL（预写式日志）可以放置在Intel Optane PMEM上，从而实现恢复时间和性能的显著改进。要启用此功能，请将GUC选项`wal_sync_method`设置为`pmem_drain`。详见 `wal_sync_method`。

- MySQL兼容性增强
  - 函数支持：  
find\_in\_set()、group\_concat()、if()、truncate()、datediff()、locate() 和 database()。详见 myfce。
  - longtext数据类型支持
  - LIMIT(m, n)语法支持。详见 Section 7.6
- Oracle兼容性增强，包括数据类型隐式转换，如数字类型（例如 int2、int4、int8、float4、float8和numeric）和字符串类型（例如 char、varchar、text和unknow）。详见 Chapter 10
- 当将表设置为只读时，自动执行自动清理。无需手动执行。
- 在使用SHOW语句查看GUC参数时，可以使用通配符'%'。例如，您可以使用“SHOW max %”、“SHOW %timeout”和“SHOW %vacuum%”。详见 SHOW。
- 当使用UPDATE语句更新单个表时，可以使用别名。例如，您可以使用“UPDATE mytable AS t SET t.colname = value WHERE t.id = 1”。
- 支持根据指定规则自动清理归档的WAL和日志文件。详见 Section 17.5.3
  - 归档WAL：在成功归档WAL时，清理存档目录( lightdb\_archive\_dir)中早于(检查点文件- lightdb\_archive\_retention\_size)的归档WAL文件。例如：lightdb\_archive\_retention\_size = 2，检查点文件为00000001000000000000000008，将删除早于000000010000000000000006的文件。
  - 日志文件：通过将 lightdb\_log\_retention\_age设置为N分钟（通常设置为xxxd以删除xxx天前的文件）来删除早于N分钟的文件。在切换到新文件时执行。时间取决于文件的修改时间。
- COPY和pg\_bulkload支持文本的多字符分隔符（最多三个）。详见 COPY。

```
COPY country TO STDOUT (DELIMITER 'ABC');
```

- 修改默认search\_path以包含Oracle模式。这意味着orafce扩展中的函数和数据类型可以直接在任何地方使用。
- 集成了Citus扩展，但默认情况下未启用。
- 集成了pg\_pathman扩展，并默认启用。
- 优化了pg\_profile，增加了LightDB配置文件(PWR)中的数据库时间统计。详见 pgprofile。
- 优化了pg\_hint\_plan，包括多个提示支持、提示范围限定、qb\_name语法支持和no\_merge提示语法支持。现在您只能为子查询使用提示。详见 pghint\_plan。
- 优化了pg\_cron，包括秒级定时任务支持和一次性调度任务支持。此外，任务的默认时区调整为UTC/GMT+08:00。详见 lt\_cron。
- 优化了system\_stats，添加了一个新函数lt\_get\_sys\_kernel\_settings()，用于获取Linux内核参数。
- lt\_initdb命令行参数支持指定非默认端口号(-p)。例如：lt\_initdb -p xxx -D xxx。详见 lt\_initdb。
- 支持ROWNUM。ROWNUM是一个类似于Oracle的伪列。在查询返回的每一行中，ROWNUM伪列返回一个数字，表示LightDB从表或一组连接的行中选择行的顺序。

现在ROWNUM支持在order by、group by、子查询、连接、union和分区表中使用。但是，它不支持在更新或删除中使用，也不支持在列表表达式中使用（例如从xxx中选择rownum + 1）。详见 Section 7.9

- 以下扩展已从LightDB中删除。
  - pg\_wait\_sampling
  - pg\_wait\_sampling
  - timescaledb

## G.9. 版本发布 13.3-21.2

发布日期： 2021-10-24

此版本包含了从 LightDB 13.3-21.1 中的各种变更。

- 该版本包含了数据库监视器和管理平台，即嵌入式版本的 LightDB 企业管理器 (LightDB-EM)，为 LightDB 提供监视和管理解决方案。
- 新的安装向导，具有 GUI 和 CLI 模式。能够让您简单轻松地安装独立或高可用部署的 LightDB，包括 LightDB-EM。
- 增强了 PWR 和 ASH，特别是等待事件。帮助您找出在 LightDB 数据库中消耗最多资源的活动和顶级 SQL。
- 增强了与 Oracle 的兼容性，如下所示：函数. rowid、sysdate、nvl 函数支持数字、实数、整数、文本、varchar、bpchar、bit 转换。guid 函数。数据类型. clob、blob、varchar2、number、rowid。
- 增强了与 MySQL 的兼容性，如下所示：函数. uuid 函数。数据类型. datetime。语法和特性. 支持 AUTO\_INCREMENT、ON UPDATE CURRENT\_TIMESTAMP。
- 添加了一个新的 GUC 参数 lightdb\_keep\_temp\_file，用于调试目的。允许保留临时文件，例如排序和哈希临时文件。
- 添加了 Ordered 优化器提示，它等价于 Leading 优化器提示。
- 为了改善数据库设计和最佳实践，LightDB通过DDL引入了设计检查。 现在支持五种检查：keyword\_not\_allowed, bad\_datatype, must\_have\_pk, must\_have\_gmt, no\_more\_100cols。这些规则可以全部启用或根据需要启用，就像MySQL的sql-mode一样。 GUC 参数 lightdb\_best\_practice\_mode和lightdb\_best\_practice\_level用于控制规则。 现有对象不受影响。
- 为了提高遗留系统拉模式增量同步速度，LightDB支持隐藏的自增主键列和隐藏的更新时间戳列。 添加WITH PRIMARY KEY以创建一个隐式列ltapk，当插入新行时将自动递增。 添加WITH UPDATE CURRENT\_TIMESTAMP以创建隐式列ltaut，在插入或修改行a时将自动更新。
- 下列内建扩展在每个数据库中默认自动启用，用户不需要手动执行 CREATE EXTENSION:
  - pg\_log\_long\_xact, pl/pgsql 函数用于记录长时间运行的事务。
  - pg\_bulkload, pg\_bulkload是一个高速数据加载工具，用于 LightDB，是 COPY 命令的替代品。
  - system\_stats, LightDB 的系统统计函数。
  - timescaledb, 启用时间序列数据的可扩展插入和复杂查询。

- zhparser, 用于中文全文搜索的解析器。默认配置名称为 zhcfg。
- rum, RUM索引访问方法, 更快的全文搜索实现。
- tsvector2, tsvector的扩展格式, 更好的空间使用数据类型, 用于全文搜索。
- 以下扩展也是内置的, 但不默认启用, 请在使用它们之前执行 CREATE EXTENSION。
  - mysql\_fdw, 用于MySQL的外部数据包装器 (FDW)
  - oracle\_fdw, 用于Oracle的外部数据包装器 (FDW)

## G. 10. 版本 13.3–21.1

发布日期: . 2021年8月17日

这是LightDB的第一个正式版本。

- 一些GUC参数的默认值已经被修改, 以适应现代硬件的更好性能。
- CREATE TABLE语句的子句中可以声明本机分区, 支持列表、范围和哈希语法。
- 添加lightdb\_explain\_no\_runtime\_pruned\_partition参数, 用于控制运行时剪枝的分区是否显示在EXPLAIN ANALYZE输出中。
- 内置了pg\_hint\_plan, 可以使用特殊形式的注释中的提示语句来控制执行计划。
- 内置pg\_wait\_sampling, 提供基于采样的等待事件统计信息。
- 内置pg\_profile, 可以帮助您找出在LightDB数据库中最耗费资源的活动。
- 内置pg\_cron, 一个简单的基于cron的作业调度程序, 运行在数据库内部。它使用与普通cron相同的语法, 但它允许您直接从数据库中调度LightDB命令。
- 内置pg\_cheat\_funcs, 它提供了一组在LightDB上使用的欺骗性 (但有用的) 函数。
- 内置pgfincore, 它提供了一组函数来处理关系的低级管理, 使用mincore来探索缓存内存。
- 内置了orafce, 一个Oracle兼容的函数和包。
- 内置pgaudit, 它通过标准的LightDB日志记录功能提供了详细的会话和/或对象审计日志记录。
- 内置pg\_probackup, 它管理LightDB数据库集群的备份和恢复。
- 以下内置扩展在每个数据库中默认自动启用 (除了dblink、pg\_profile和pg\_cron, 这些仅在postgres数据库中可用), 您不需要手动执行CREATE EXTENSION。
  - pg\_stat\_statements
  - auto\_explain
  - tablefunc
  - dblink
  - uuid-oss
  - pg\_hint\_plan



- `pg_wait_sampling`
- `pg_profile`
- `pg_cron`
- `pgfincore`
- `orafce`
- `pgaudit`

特别的，一个名为“`take_sample_every_10min`”的`pg_cron`作业将默认自动在postgres数据库中创建。此作业将每隔10分钟调用`take_sample()`（`take_sample()`由`pg_profile`扩展提供）。

---

# Appendix H. 额外提供的模块

这个附录和下一个附录介绍在LightDB发布的contrib目录中能找到的模块。包括移植工具、分析工具和插件特性，它们不是 LightDB 核心系统的一部分，主要因为只有很少的用户会用到或者是还处于实验阶段。但这不会影响它们的使用。

这个附录覆盖了在contrib中能找到的扩展和其他服务器插件模块。Appendix 覆盖了工具程序。

如果是用的是预打包版的LightDB，这些模块通常可以作为一个单独的子包来获得，如postgresql-contrib。

许多模块提供新的用户自定义函数、操作符或数据类型。在已经安装了代码之后，为了使用这些模块，需要在数据库系统中注册新的 SQL 对象。可以通过执行一个CREATE EXTENSION命令来完成。在一个新的数据库中，你可以简单地

```
CREATE EXTENSION module_name;
```

这个命令把新的 SQL 对象注册在当前数据库中，因此你需要在每一个你希望使用该模块功能的数据库中执行他。另外，可以在template1数据库中运行这个命令以便该扩展能被默认地复制到后续创建的数据库中。

对于所有这些模块，CREATE EXTENSION必须由数据库超级用户运行，除非该模块被视为“trusted”，在这种情况下，它可以由在当前数据库上具有CREATE特权的任何用户运行。受信任的模块将在下面的章节中标识为这样的模块。通常，受信任的模块不能提供对数据库外部函数的访问。

很多模块允许你将它们的对象安装在你选择的一个模式中。要这样做，需要将SCHEMA schema\_name加入到CREATE EXTENSION命令中。默认情况下，这些对象将被放置在你的当前创建目标模式中，通常是public。

不过注意，有一些这样的模块不是这种意义上的“扩展”，而是以某种其他方式被载入到服务器，例如使用shared\_preload\_libraries方式。每个模块详见其文档。

## H. 1. adminpack

adminpack 提供了一些支持函数，pgAdmin 和其他管理工具会用这些函数来提供额外的功能，例如服务器日志文件的远程管理。所有这些函数默认只允许由超级用户使用，但是可以通过使用 GRANT 命令允许其他用户使用。

Table H. 中展示的函数提供了向运行着服务器的机器上的文件进行写入的途径（另见 Table 9.9中的函数，它们提供了只读的访问途径）。对于这些函数，只有位于数据库集簇目录中的文件才能被访问（除非用户是一个超级用户或者被指定了 pg\_read\_server\_files 或者 pg\_write\_server\_files 角色之一），不过允许使用相对路径和绝对路径。

Table H.1. adminpack Functions

函数	描述
pg_catalog.pg_file_write ( filename text, data text, append boolean ) → bigint	写入或追加到一个文本文件.
pg_catalog.pg_file_sync ( filename text ) → void	刷新一个文件或目录到硬盘。

函数	描述
<code>pg_catalog.pg_file_rename ( oldname text, newname text [, archivename text ] )</code> → boolean	重命名一个文件。
<code>pg_catalog.pg_file_unlink ( filename text )</code> → boolean	移除一个文件。
<code>pg_catalog.pg_logdir_ls ()</code> → setof record	列出log_directory 目录中的日志文件。

`pg_file_write` 把指定的 data 写入到由 filename 命名的文件中。如果 append 为假，文件不能已经存在。如果 append 为真，该文件 可能已经存在，并且如果存在就会被追加。这个函数返回写入的字节数。

`pg_file_sync` fsyncs由filename命名的指定的文件或目录。 在失败时会出现错误（例如：指定的文件不存在）。 注意 `data_sync_retry` 对此函数没有影响,因此在刷新数据库文件失败的时候不会引发PANIC-级别的错误。

`pg_file_rename`重命名一个文件。如果 archivename被省略或者为 NULL，它简单地 把 oldname重命名为newname（不能 已经存在）。如果提供了archivename，该函数首先把 newname重命名为archivename（不 能已经存在），然后把oldname重命名为 newname。当第二次重命名失败时，这个函数会在报告 错误之前尝试把archivename重命名成 newname。成功时函数返回真，如果源文件不存在或者 不可写则返回假，其他情况下会抛出错误。

`pg_file_unlink`移除指定的文件。成功时函数返回真， 如果指定的文件不存在或者 `unlink()`调用失败则返回 假，其他情况下会抛出错误。

`pg_logdir_ls`返回log\_directory 目录中所有日志文件的开始时间戳以及路径名。要使用这个函数， log\_filename参数必须具有其默认设置（ `lightdb-%Y-%m-%d_%H%M%S.log`）。

## H. 2. amcheck

amcheck 模块提供的函数让用户能验证关系结构的逻辑一致性。

这些B-Tree检查函数验证特定关系的结构表达中的各种不变条件。 索引扫描以及其他重要操作背后的访问方法的正确性都要依仗这些不变条件的成立。 例如，在这些函数中，有一些负责验证所有B树页面中的项都按照“逻辑”顺序（比如，对于text上的B树索引，索引元组应该按照词典顺序排列）摆放。 如果特定的不变条件由于某种原因无法成立，则我们可以预料受影响页面上的二分搜索将无法正确地引导索引扫描，最终导致SQL查询得到错误的 答案。 如果结构看起来有效，则不会引发错误。

验证过程采用索引扫描自身使用的同种过程来执行，这些过程可能是用户定义的操作符类代码。例如，B树索引验证依赖于由一个或者多个B树支持函数1例程构成的比较。操作符类支持函数的详情请见Section 36.16.3

不像通过提示错误来报告损坏的B-Tree检查函数，堆检查函数`verify_heapam`检查表并尝试返回一组行，每次检测到损坏就返回一行。 尽管如此，如果`verify_heapam`依赖的工具本身已损坏，则函数可能无法继续，并且可能会引发错误。

执行amcheck函数的权限可以授予非超级用户，但在授予权限之前，应该仔细考虑数据的安全和隐私问题。 虽然这些函数所产生的破坏报告不关注的破坏数据的内容,而且这些数据的结构和发现破坏的本质， 获得的权限的攻击者可以执行这些函数,特别是如果攻击者也能导致破坏,也许可以从这些信息中推断出数据本身的某些信息。

## H. 2. 1. 函数

`bt_index_check(index regclass, heapallindexed boolean)` returns void

`bt_index_check`测试一个B树索引，检查各种不变条件。用法实例：

```
test=# SELECT bt_index_check(index => c.oid, heapallindexed =>
        i.indisunique),
        c.relname,
        c.relpages
FROM pg_index i
JOIN pg_opclass op ON i.indclass[0] = op.oid
JOIN pg_am am ON op.opcmethod = am.oid
JOIN pg_class c ON i.indexrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE am.amname = 'btree' AND n.nspname = 'pg_catalog'
-- Don't check temp tables, which may be from another session:
AND c.relpersistence != 't'
-- Function may throw an error when this is omitted:
AND c.relkind = 'i' AND i.indisready AND i.indisvalid
ORDER BY c.relpages DESC LIMIT 10;
```

bt_index_check	relname	relpages
	pg_depend_reference_index	43
	pg_depend_depender_index	40
	pg_proc_prname_args_nsp_index	31
	pg_description_o_c_o_index	21
	pg_attribute_relid_attnam_index	14
	pg_proc_oid_index	10
	pg_attribute_relid_attnum_index	9
	pg_amproc_fam_proc_index	5
	pg_amop_opr_fam_index	5
	pg_amop_fam_strat_index	5

(10 rows)

这个例子中的会话执行对数据库“test”中10个最大目录索引的验证。对于唯一索引会要求验证堆元组是否有对应的索引元组存在。由于没有错误报出，所有的被测索引都处于逻辑一致的状态。自然地，很容易将这个查询改为对支持验证的数据库中的每一个索引调用`bt_index_check`。

`bt_index_check`要求目标索引及其所属的堆关系上的`AccessShareLock`。这种锁模式与简单`SELECT`语句在关系上所要求的锁模式相同。`bt_index_check`不验证跨越父子关系的不变条件，但是在`heapallindexed`为`true`时将验证所有堆元组是否作为索引中的索引元组存在。当在生产环境中要求一个使用`bt_index_check`的例程进行轻量化损坏测试时，它常常需要在验证彻底性和减小对应用性能及可用性的影响之间做出权衡。

`bt_index_parent_check(index regclass, heapallindexed boolean, rootdescend boolean)` returns void

`bt_index_parent_check`测试一个B树索引，检查多种不变条件。可选地，当`heapallindexed`参数为`true`时，该函数验证所有应该在索引中找到的堆元组的存在。当可选参数`rootdescend`值为`true`时，对于每个元组，验证程序通过从根页面执行新的搜索来重新查找叶子层级的元组。`bt_index_parent_check`能够执行的检查是`bt_index_check`能执行的检查的超集。`bt_index_parent_check`可以被想成是`bt_index_check`的一种更全面的变体：和`bt_index_check`不同，`bt_index_parent_check`还检查跨越父/子关系的不变条件，包括检查索引结构中是否没有缺失的下链。如果找到逻辑不一致或者其他问题，`bt_index_parent_check`遵循通常的报错习惯。

`bt_index_parent_check`要求目标索引上的一个ShareLock（还要求对关系上的一个ShareLock）。这些锁阻止来自INSERT、UPDATE以及DELETE命令的并发数据修改。这些锁同时防止底层关系被并发的VACUUM以及其他工具命令处理。注意该函数只在其运行期间而不是整个事务期间持有锁。

`bt_index_parent_check`的额外验证更有可能检测到多种病态的情况。这些情况可能涉及到被查索引使用的一种不正确的实现的B-树操作符类，或者说不定是底层B-树索引访问方法代码中未被发现的缺陷。注意与`bt_index_check`不同，当热备模式被启用时（即在只读的物理复制机上）不能使用`bt_index_parent_check`。

### Tip

`bt_index_check` 和 `bt_index_parent_check` 都输出关于验证过程的日志信息，在DEBUG1 和 DEBUG2 严重性级别。 这些消息提供关于验证过程的详细信息，或许对LightDB的开发人员有作用。 高级用户也许会发现这些信息很有帮助，因为它提供了额外的上下文将验证实际检测的不一致。运行：

```
SET client_min_messages = DEBUG1;
```

在运行验证查询之前的交互式`ltsql`会话中，将显示有关验证进度的消息，并具有可管理级别的详细信息。

## H. 2. 2. 可选的`heapallindexed`验证

当B-Tree验证函数的`heapallindexed`参数为`true`时，会针对与目标索引关系关联的表执行一个额外的验证过程。这种验证由一个“假的”CREATE INDEX操作组成，它针对一个临时的、内存中的汇总表（根据需要在基础的第一阶段验证过程中建立）检查所有假想的新索引元组的存在。这个汇总表对目标索引中的每一个元组“采集指纹”。`heapallindexed`验证背后的高层原则是：等效于现有目标索引的新索引必须仅拥有能在现有结构中找到项。

额外的`heapallindexed`阶段会增加明显的开销：验证的时间通常将会延长几倍。不过，在执行`heapallindexed`验证时，所要求的关系级锁没有变化。

这一汇总表的结构尺寸以`maintenance_work_mem`为界。为了确保对于每个堆元组应该存在于索引中这一检测有不超过2%的失效概率能检测到不一致，每个元组需要大约2个字节的内存。因为每个元组可用的内存变少，错失一处不一致的概率就会慢慢增加。这种方法显著地限制了验证的开销，但仅仅略微降低了检测到问题的概率，对于将验证当作例行维护任务的安装来说更是如此。对于每一次新的验证尝试，任何单一的缺失或者畸形元组都有新的机会被检测到。

## H. 2. 3. 有效地使用`amcheck`

`amcheck`对于检测多种数据校验和无法捕捉到的失效模式非常有效。包括：

- 由不正确的操作符类实现导致的结构性不一致。

这包括操作系统排序规则的比较规则变化导致的问题。 `text`之类的可排序类型数据的比较必须是不变的（正如用于B-树索引扫描的所有比较必须不变一样），这意味着操作系统排序规则必须保持不变。 但是在很少的情况下，操作系统排序规则的更新会导致这些问题。 更常见的，主服务器和后备服务器之间排序顺序的不一致会相互牵连，这可能是由于使用的主操作系统版本不一致。 这类不一致通常仅出现在后备服务器上，因此通常也仅能在后备服务器上检测到。

如果这类问题出现，则它可能不会影响使用受影响排序规则排序的每一个索引，其原因是被索引值可能正好具有与行为不一致无关的相同的绝对顺序。关于LightDB如何使用操作系统`locale`和排序规则的进一步细节请参考Section 21. 和Section 21.2

- 索引和被索引的对关系之间的结构不一致（在执行heapallindexed验证时）。

在普通操作时没有将索引针对其对关系进行交叉检查。堆损坏的症状可能是很微妙的。

- 由于底层LightDB访问方法代码、排序代码或者事务管理代码中（假想的）未发现的缺陷导致的损坏。

在测试可能引入逻辑不一致的LightDB新特性或者被提议的特性时，索引的结构完整性自动验证扮演了重要角色。表结构、相关的可见性和事务状态信息的验证扮演了类似的角色。一种显而易见的测试策略是在运行标准回归测试时持续地调用amcheck函数。运行这些测试的详情请参考Section 30.1

- 正巧没有开启校验和的文件系统或者存储子系统故障。

注意，如果在访问块时仅有一次共享缓存命中，验证时amcheck会在检查表示在某个共享内存缓冲区中的页面。因此，amcheck没有必要在验证时检查从文件系统读出的数据。注意当校验和被启用时，如果一个损坏的块被读取到缓冲区中，amcheck可能会由于校验和失效而产生错误。

- 有缺陷的RAM或者内存子系统导致的损坏。

LightDB无法提供针对可更正内存错误的保护并且它假定用户使用的是具有工业标准纠错码（ECC）或更好保护技术的RAM。不过，ECC内存通常只能免疫单个位错误，并且不应该假定它能提供对导致内存损坏失效的绝对保护。

在执行heapallindexed验证时，通常有大幅增加的机会可以检测单个位错误，因为会测试严格的二元等值并且会在堆中测试被索引属性。

通常，amcheck仅能证明损坏的存在，但它无法证明损坏不存在。

## H. 2. 4. 修复损坏

amcheck没有产生与损坏相关的错误绝不应该被当做假阳性。amcheck会在（定义上）应该绝不会发生的情况中抛出错误，因此常常需要对amcheck错误进行仔细地分析。

对于amcheck检测到的问题没有一般性的修复方法。应该寻找产生不变条件违背的根本原因。在诊断amcheck检测到的损坏时，pageinspect可能会扮演一个非常有用的角色。REINDEX在修复损坏过程中可能无法起到效果。

## H. 3. auth\_delay

auth\_delay使得服务器在报告鉴定失败之前短暂地停顿一会儿，这使得对数据库口令的蛮力攻击更加困难。注意它无助于组织拒绝服务攻击，甚至可能会加剧它们，因为在报告鉴定失败之前等待的进程仍然要消耗连接。

要使之工作，该模块必须通过lightdb.conf中的shared\_preload\_libraries载入。

### H. 3. 1. 配置参数

auth\_delay.milliseconds (int)

在报告鉴定失败之前等待的毫秒数。默认值为0。

这些参数必须在lightdb.conf中设置。典型的使用是：

```
# lightdb.conf
shared_preload_libraries = 'auth_delay'
```

```
auth_delay.milliseconds = '500'
```

### H. 3.2. 作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

## H. 4. auto\_explain

auto\_explain模块提供了一种方式来自动记录慢速语句的执行计划，而不需要手工运行EXPLAIN。这在大型应用中追踪未被优化的查询时有用。

该模块没有提供 SQL 可访问的函数。要使用它，简单地将它载入服务器。你可以把它载入到一个单独的会话：

```
LOAD 'auto_explain';
```

（你必须作为超级用户来这样做）。更典型的用法是通过在lightdb.conf的session\_preload\_libraries或shared\_preload\_libraries参数中包括auto\_explain将它预先载入到某些或者所有会话中。然后你就可以追踪那些出乎意料地慢的查询，而不管它们何时发生。当然为此会付出一些额外的负荷作为代价。

auto\_explain 模块还将生成一个单独的日志文件，该文件称为慢日志。慢日志文件与默认日志文件名称相同，使用“.slow”作为后缀。慢日志的格式始终是JSON。

### H. 4.1. 配置参数

有几个配置参数用来控制auto\_explain的行为。注意默认行为是什么也不做，因此如果你想要任何结果就必须至少设置auto\_explain.log\_min\_duration。

auto\_explain.log\_min\_duration (integer)

auto\_explain.log\_min\_duration是最小语句执行时间（以毫秒计），这将导致语句的计划被记录。这个参数设置为0时将记录所有计划，设置为-1（默认值）时禁用记录计划。例如，如果你将它设置为250ms，则所有运行时间等于或超过 250ms 的语句将被记录。只有超级用户能够改变这个设置。

auto\_explain.log\_analyze (boolean)

当一个执行计划被记录时，auto\_explain.log\_analyze导致EXPLAIN ANALYZE输出（而不仅仅是EXPLAIN输出）被打印。默认情况下这个参数是关闭的。只有超级用户能够改变这个设置。

#### Note

当这个参数为打开时，对所有被执行的语句将引起对每个计划节点的计时，不管它们是否运行得足够长以至于被记录。这可能会对性能有极度负面的影响。

auto\_explain.log\_buffers (boolean)

当一个执行计划被记录时，auto\_explain.log\_buffers控制是否打印缓冲区使用统计信息；它等效于EXPLAIN的BUFFERS选项。除非 auto\_explain.log\_analyze参数被设置，否则这个参数没有效果。这个参数默认情况下是关闭的。只有超级用户能够改变这个设置。

`auto_explain.log_wal` (boolean)

`auto_explain.log_wal` 控制当执行计划被记录时是否打印WAL使用情况统计信息；它相当于EXPLAIN的WAL选项。除非启用了`auto_explain.log_analyze`，否则这个参数不生效。这个参数默认是关闭状态。只有超级用户能够改变这个设置。

`auto_explain.log_timing` (boolean)

当一个执行计划被记录时，`auto_explain.log_timing`控制是否打印每个结点上的计时信息；它等效于EXPLAIN的TIMING选项。重复读取系统锁的开销在某些系统上可能会显著地拖慢查询，因此当只需要实际行计数而非确切时间时，关闭这个参数将会很有帮助。只有当`auto_explain.log_analyze`也被启用时这个参数才有效。这个参数默认情况下是打开的。只有超级用户能够改变这个设置。

`auto_explain.log_triggers` (boolean)

当一个执行计划被记录时，`auto_explain.log_triggers`会导致触发器执行统计信息被包括在内。只有当`auto_explain.log_analyze`也被启用时这个参数才有效。这个参数默认情况下是关闭的。只有超级用户能够改变这个设置。

`auto_explain.log_verbose` (boolean)

当一个执行计划被记录时，`auto_explain.log_verbose`控制是否打印很长的详细信息；它等效于EXPLAIN的VERBOSE选项。这个参数默认情况下是关闭的。只有超级用户能够改变这个设置。

`auto_explain.log_settings` (boolean)

`auto_explain.log_settings`控制执行计划被日志记录时是否打印关于已修改的配置选项的信息。输出中仅包含影响查询计划的选项，其值与内置缺省值不同。该参数默认关闭。仅超级用户可修改该设置。

`auto_explain.log_format` (enum)

`auto_explain.log_format`选择要使用的EXPLAIN输出格式。允许的值是text、xml、json和yaml。默认是文本形式。只有超级用户能够改变这个设置。

`auto_explain.log_level` (enum)

`auto_explain.log_level`选择日志的级别，在此级别上，`auto_explain`会记录查询计划。有效值为DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING和LOG。缺省为LOG。仅超级用户可修改该设置。

`auto_explain.log_nested_statements` (boolean)

`auto_explain.log_nested_statements`导致嵌套语句（在一个函数内执行的语句）会被考虑在记录范围之内。当它被关闭时，只有顶层查询计划被记录。这个参数默认情况下是关闭的。只有超级用户能够改变这个设置。

`auto_explain.sample_rate` (real)

`auto_explain.sample_rate`会让 `auto_explain` 只解释每个会话中的一部分语句。默认值为1，表示解释所有的查询。在嵌套语句的情况下，要么所有语句都被解释，要么一个也不被解释。只有超级用户能够更改这个设置。

在普通用法中，这些参数都在`lightdb.conf`中设置，不过超级用户可以在他们自己的会话中随时修改这些参数。典型的用法可能是：

```
# lightdb.conf
session_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '3s'
```



## H. 4. 2. 例子

```
lightdb@postgres=# LOAD 'auto_explain';
lightdb@postgres=# SET auto_explain.log_min_duration = 0;
lightdb@postgres=# SET auto_explain.log_analyze = true;
lightdb@postgres=# SELECT count(*)
                    FROM pg_class, pg_index
                    WHERE oid = indrelid AND indisunique;
```

这可能会产生这样的日志输出：

```
LOG:  duration: 3.651 ms  plan:
       Query Text: SELECT count(*)
                    FROM pg_class, pg_index
                    WHERE oid = indrelid AND indisunique;
       Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1
loops=1)
       -> Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594
rows=92 loops=1)
           Hash Cond: (pg_class.oid = pg_index.indrelid)
           -> Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual
time=0.016..0.140 rows=255 loops=1)
           -> Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238
rows=92 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 4kB
               -> Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4)
(actual time=0.008..3.187 rows=92 loops=1)
                   Filter: indisunique
```

## H. 4. 3. 作者

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>

## H. 5. bloom

bloom提供了一种基于布鲁姆过滤器<sup>1</sup>的索引访问方法。

布鲁姆过滤器是一种空间高效的数据结构，它被用来测试一个元素是否为一个集合的成员。在索引访问方法的情况下，它可以通过尺寸在索引创建时决定的签名来快速地排除不匹配的元组。

签名是被索引属性的一种有损表达，并且因此容易报告伪肯定，也就是说对于一个不在集合中的元素有可能报告该元素在集合中。因此索引搜索结果必须使用来自堆项的实际属性值进行再次检查。较大的签名可以降低伪肯定的几率并且减少无用的堆访问的次数，但是这显然会让索引更大且扫描起来更慢。

当表具有很多属性并且查询可能会测试其中任意组合时，这种类型的索引最有用。传统的 btree 索引比布鲁姆索引更快，但是需要很多 btree 索引来支持所有可能的查询，而对于布鲁姆索引来说只需要一个即可。不过要注意 bloom 索引只支持等值查询，而 btree 索引还能执行不等和范围搜索。

### H. 5. 1. 参数

bloom索引在其WITH子句中接受下列参数：

<sup>1</sup> [https://en.wikipedia.org/wiki/Bloom\\_filter](https://en.wikipedia.org/wiki/Bloom_filter)

length

每个签名（索引项）的长度位数，它会被圆整成为最近的16的倍数。默认是80位，最长是4096位。

col1 — col32

从每一个索引列产生的位数。每个参数的名字表示它所控制的索引列的编号。默认是2位，最大是4095位。没有实际使用的索引列的参数会被忽略。

## H. 5. 2. 例子

这是一个创建布鲁姆索引的例子：

```
CREATE INDEX bloomidx ON tbloom USING bloom (i1,i2,i3)
    WITH (length=80, col1=2, col2=2, col3=4);
```

该索引是用长度为 80 位的签名所创建，其中属性 i1 和 i2 被映射为 2 位，属性 i3 被映射为 4 位。我们可以省略length、col1和col2说明，因为它们都有默认值。

这里是布鲁姆索引定义和使用的更完整的例子，其中还与等效的 btree 做了对比。布鲁姆索引比 btree 索引更小，并且效率更高。

```
=# CREATE TABLE tbloom AS
  SELECT
    (random() * 1000000)::int as i1,
    (random() * 1000000)::int as i2,
    (random() * 1000000)::int as i3,
    (random() * 1000000)::int as i4,
    (random() * 1000000)::int as i5,
    (random() * 1000000)::int as i6
  FROM
    generate_series(1,10000000);
SELECT 10000000
```

在这个大表上的顺序扫描需要很长时间：

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
      QUERY PLAN
```

```
-----
Seq Scan on tbloom (cost=0.00..2137.14 rows=3 width=24) (actual
time=15.480..15.480 rows=0 loops=1)
  Filter: ((i2 = 898732) AND (i5 = 123451))
  Rows Removed by Filter: 100000
Planning Time: 0.340 ms
Execution Time: 15.501 ms
(5 rows)
```

即使定义了btree索引，结果将仍然是顺序扫描：

```
=# CREATE INDEX btreeidx ON tbloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('btreeidx'));
```

```

pg_size_pretty
-----
3976 kB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                QUERY PLAN
-----

```

```

Seq Scan on tbloom (cost=0.00..2137.00 rows=2 width=24) (actual
time=12.604..12.604 rows=0 loops=1)
  Filter: ((i2 = 898732) AND (i5 = 123451))
  Rows Removed by Filter: 100000
Planning Time: 0.155 ms
Execution Time: 12.617 ms
(5 rows)

```

在处理这类搜索时, bloom 比 btree 更好:

```

=# CREATE INDEX bloomidx ON tbloom USING bloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size(' bloomidx'));
pg_size_pretty
-----
1584 kB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                QUERY PLAN
-----

```

```

Bitmap Heap Scan on tbloom (cost=1792.00..1799.69 rows=2 width=24) (actual
time=0.384..0.384 rows=0 loops=1)
  Recheck Cond: ((i2 = 898732) AND (i5 = 123451))
  Rows Removed by Index Recheck: 26
  Heap Blocks: exact=26
-> Bitmap Index Scan on bloomidx (cost=0.00..1792.00 rows=2 width=0)
(actual time=0.350..0.350 rows=26 loops=1)
   Index Cond: ((i2 = 898732) AND (i5 = 123451))
Planning Time: 0.122 ms
Execution Time: 0.407 ms
(8 rows)

```

现在, btree 搜索的主要问题是, 当搜索条件不约束前几个索引列时, btree 的效率不好。对于 btree 更好的策略是在每一列上创建一个独立的索引。那么规划器将选择这样的计划:

```

=# CREATE INDEX btreeidx1 ON tbloom (i1);
CREATE INDEX
=# CREATE INDEX btreeidx2 ON tbloom (i2);
CREATE INDEX
=# CREATE INDEX btreeidx3 ON tbloom (i3);
CREATE INDEX
=# CREATE INDEX btreeidx4 ON tbloom (i4);
CREATE INDEX
=# CREATE INDEX btreeidx5 ON tbloom (i5);
CREATE INDEX
=# CREATE INDEX btreeidx6 ON tbloom (i6);

```

```

CREATE INDEX
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                QUERY PLAN
-----
Bitmap Heap Scan on tbloom (cost=24.34..32.03 rows=2 width=24) (actual
time=0.032..0.033 rows=0 loops=1)
  Recheck Cond: ((i5 = 123451) AND (i2 = 898732))
    -> BitmapAnd (cost=24.34..24.34 rows=2 width=0) (actual time=0.029..0.030
rows=0 loops=1)
      -> Bitmap Index Scan on btreeidx5 (cost=0.00..12.04 rows=500 width=0)
(actual time=0.029..0.029 rows=0 loops=1)
        Index Cond: (i5 = 123451)
      -> Bitmap Index Scan on btreeidx2 (cost=0.00..12.04 rows=500 width=0)
(never executed)
        Index Cond: (i2 = 898732)
    Planning Time: 0.537 ms
    Execution Time: 0.064 ms
(9 rows)

```

尽管这个查询运行起来比在其中任一单个索引上都快，但是我们在索引尺寸上付出了代价。每一个单列 btree 索引占用 2 MB，因此总的空间会超过 12 MB，这是 bloom 索引所使用的空间的 8 倍。

### H. 5. 3. 操作符类接口

用于布鲁姆索引的操作符类只要一个用于被索引数据类型的哈希函数以及一个用于搜索的等值操作符。这个例子展示了用于 text 数据类型的操作符类定义：

```

CREATE OPERATOR CLASS text_ops
DEFAULT FOR TYPE text USING bloom AS
    OPERATOR    1    =(text, text),
    FUNCTION    1    hashtext(text);

```

### H. 5. 4. 限制

- 在模块中只包括了用于 int4 以及 text 的操作符类。
- 搜索只支持 = 操作符。但是未来可以为带合并和交集操作的数组增加支持。
- bloom 访问方法不支持 UNIQUE 索引。
- bloom 访问方法不支持对 NULL 值的搜索。

### H. 5. 5. 作者

Teodor Sigaev <teodor@postgrespro.ru>, Hundsun Technologies Inc, Hangzhou, China

Alexander Korotkov <a.korotkov@postgrespro.ru>, Hundsun Technologies Inc, Hangzhou, China

Oleg Bartunov <obartunov@postgrespro.ru>, Hundsun Technologies Inc, Hangzhou, China

## H. 6. btree\_gin

`btree_gin`提供了一个为数据类型 `int2`、`int4`、`int8`、`float4`、`float8`、`timestamp with time zone`、`timestamp without time zone`、`time with time zone`、`time without time zone`、`date`、`interval`、`oid`、`money`、`"char"`、`varchar`、`text`、`bytea`、`bit`、`varbit`、`macaddr`、`macaddr8`、`inet`、`cidr`、`uuid`、`name`、`bool`、`bpchar`和所有enum类型实现B树等价行为的GIN操作符类例子。

通常，这些操作符类不会比等效的标准B树索引方法更好，并且它们缺少标准B树代码的一个主要特性：强制一致性的能力。但是，它们有助于GIN测试并且有助于作为开发其他GIN操作符类的基础。另外，对于测试一个GIN可索引的列和一个B树可索引的列的查询，创建一个使用这些操作符类之一的多列GIN索引要比创建必须通过位图AND组合在一起的两个独立索引要更有效。

这个模块被认为是“trusted”，也就是说，它可以被当前数据库中拥有CREATE特权的非超级用户安装。

## H. 6. 1. 用法示例

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING GIN (a);
-- query
SELECT * FROM test WHERE a < 10;
```

## H. 6. 2. 作者

Teodor Sigaev (<teodor@stack.net>) 和 Oleg Bartunov (<oleg@sai.msu.su>)。更多信息请见<http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin>。

## H. 7. btree\_gist

`btree_gist`为数据类型 `int2`、`int4`、`int8`、`float4`、`float8`、`numeric`、`timestamp with time zone`、`timestamp without time zone`、`time with time zone`、`time without time zone`、`date`、`interval`、`oid`、`money`、`char`、`varchar`、`text`、`bytea`、`bit`、`varbit`、`macaddr`、`macaddr8`、`inet`、`cidr`、`uuid`和所有enum类型提供了实现B树等效行为的GiST索引操作符类。

通常，这些操作符类不会比等效的标准B树索引方法更好，并且它们缺少标准B树代码的一个主要特性：强制一致性的能力。但是，如下文所述，它们提供了在一个B树索引中没有的一些其他特性。另外，当需要一个多列GiST索引，并且其某些列的数据类型只在GiST中是可索引的而其他列是简单数据类型时，这些操作符类就有用了。最后，这些操作符可以用于GiST测试以及作为开发其他GiST操作符类的基础。

除了典型的B树搜索操作符之外，`btree_gist`也为`<>`（“不等于”）提供了索引支持。这可能与下文描述的排他约束组合在一起产生作用。

另外，对于那些具有自然距离度量的数据类型，`btree_gist`定义了一个距离操作符`<->`，并且为使用这个操作符的最近邻搜索提供了GiST索引支持。距离操作符还提供给了：`int2`、`int4`、`int8`、`float4`、`float8`、`timestamp with time zone`、`timestamp without time zone`、`time without time zone`、`date`、`interval`、`oid`和`money`。

这个模块被认为是“trusted”，也就是说，它可以被当前数据库中拥有CREATE特权的非超级用户安装。

## H. 7. 1. 用法示例

使用`btree_gist`代替`btree`的简单例子：

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING GIST (a);
-- query
SELECT * FROM test WHERE a < 10;
-- nearest-neighbor search: find the ten entries closest to "42"
SELECT *, a <-> 42 AS dist FROM test ORDER BY a <-> 42 LIMIT 10;
```

使用一个排他约束来强制规则：一个动物园里的一个笼子只能装一种动物：

```
=> CREATE TABLE zoo (
    cage    INTEGER,
    animal  TEXT,
    EXCLUDE USING GIST (cage WITH =, animal WITH <>)
);

=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint
        "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key (cage,
        animal)=(123, zebra).
=> INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

## H. 7. 2. 作者

Teodor Sigaev (<teodor@stack.net>)、Oleg Bartunov (<oleg@sai.msu.su>)、Janko Richter (<jankorichter@yahoo.de>)和 Paul Jungwirth (<pj@illuminatedcomputing.com>)。参阅 <http://www.sai.msu.su/~megera/postgres/gist/>。

## H. 8. canopy

更多详细信息请参见 [canopy](#)<sup>2</sup>。

## H. 9. citext

`citext`模块提供了一种大小写不敏感的字符串类型：`citext`。特别地，它在比较值时内部调用的是`lower`。除此之外，它的行为几乎与`text`完全相同。

### Tip

可考虑使用`nondeterministic collations` (参见 Section 21.2.2.)来代替这个模块。它们可被用于大小写不敏感的比较，不区分重音的比较，以及其他组合，并且他们可以正确的处理更多Unicode特殊情况。

此模块被视为“可信(trusted)”，也就是说，它可以由当前数据库上具有 `CREATE` 权限的非超级用户来安装。

<sup>2</sup> <http://www.light-pg.com/docs/canopy/13.8-23.4/index.html>

## H. 9. 1. 基本原理

在LightDB中做大小写不敏感匹配的标准方法曾经是在比较值时使用lower函数，例如：

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

这工作得比较好，但是有一些缺点：

- 它让你的 SQL 语句冗长，并且你必须总是要记住在列和查询值上使用lower。
- 它不会使用一个索引，除非你使用lower创建一个函数索引。
- 如果你声明一个列为UNIQUE或PRIMARY KEY，隐式生成的索引是大小写敏感的。因此，它对于大小写不敏感搜索是没有用处的，并且它不会强制大小写不敏感的唯一性。

citext数据类型允许你在 SQL 查询中消除对lower的调用，并且允许一个主键是大小写无关的。就和text一样，citext是区域相关的，这意味着大写和小写字母的匹配依赖于数据库LC\_CTYPE设置的规则。此外，这种行为和在查询中使用lower是一样的。但是因为它是通过数据类型以透明的方式完成的，你不需要记住在你的查询中做任何特殊的事情。

## H. 9. 2. 如何使用它

这里是一个简单的用法示例：

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT NOT NULL
);

INSERT INTO users VALUES ( 'larry', sha256(random)::text::bytea );
INSERT INTO users VALUES ( 'Tom', sha256(random)::text::bytea );
INSERT INTO users VALUES ( 'Damian', sha256(random)::text::bytea );
INSERT INTO users VALUES ( 'NEAL', sha256(random)::text::bytea );
INSERT INTO users VALUES ( 'Bjørn', sha256(random)::text::bytea );

SELECT * FROM users WHERE nick = 'Larry';
```

即使nick列被设置为larry而查询是Larry，SELECT语句也将只返回一个元组。

## H. 9. 3. 串比较行为

citext执行比较时先将每一个串转换成小写形式（调用lower）然后正常地比较结果。因此，如果两个串通过lower产生相同的结果，它们就被认为是相等。

为了尽可能接近地模拟一种大小写不敏感的排序规则，一些串处理操作符和函数有citext相关的版本。例如，当应用到citext时，正则表达式操作符~和~\*会展现出相同的行为：它们都以大小写不敏感的方式匹配。!~和!~\*也是一样，以及LIKE操作符~~和~~\*，以及!~~和!~~\*。如果你想以大小写敏感的方式匹配，你可以把该操作符的参数造型成text。

相似地，如果下列函数的参数是citext，它们会以大小写不敏感的方式执行匹配：

- regexp\_match()
- regexp\_matches()
- regexp\_replace()

- `regexp_split_to_array()`
- `regexp_split_to_table()`
- `replace()`
- `split_part()`
- `strpos()`
- `translate()`

对于 `regexp` 函数，如果你想要以大小写敏感的方式匹配，你可以指定“c”标志来强制大小写敏感的匹配。否则，如果你想要大小写敏感的行为，你必须在使用这些函数之一之前造型到`text`。

## H. 9. 4. 限制

- `citext`的大小写折叠行为取决于你的数据库的`LC_CTYPE`设置。因此它如何比较值是在数据库被创建时决定的。在 Unicode 标准定义的术语中没有真正的大小写不敏感。实际上，它的含义是，只要你对你的排序规则满意，你就应该对`citext`的比较满意。但是如果在你的数据库中存储有不同语言的数据，当排序规则是用于一种语言时，另一种语言的用户可能会发现他们的查询结果并不是所期待的。
- 自LightDB 9.1 其，你可以为`citext`列或数据值附加一个`COLLATE`说明。当前，在比较大写折叠过的字符串时，`citext`操作符将尊重一种非默认的`COLLATE`说明，但是最初到小写形式的折叠是根据数据库的 `LC_CTYPE`设置完成的（就是说，尽管给出了`COLLATE "default"`）。这可能在未来的发行中被改变，这样两步都能遵循输入的`COLLATE`说明。
- `citext`的效率不如`text`，因为操作符函数和 B 树比较函数必须创建数据的拷贝并且将它转换为小写形式来进行比较。此外，只有`text`可以支持B树重复数据删除。不过，在进行大小写不敏感的匹配时，`citext`比使用`lower`的效率要略高。
- 如果你在某些环境下需要以大小写敏感的方式比较数据并且在另一些环境下需要以大小写不敏感的方式比较数据，`citext`就帮不上什么忙。标准的答案是使用`text`类型并且在你需要以大小写不敏感的方式比较时手工使用`lower`函数。如果大小写不敏感的比较需求不频繁，这会工作得不错。如果你大部分时间需要大小写不敏感的行为，考虑将数据存储为`citext`并且在进行大小写敏感比较时显式地将列造型为`text`。不管在那种情况下，你都需要两个索引来让两种类型的搜索更快。
- 包含`citext`操作符的模式必须在当前的`search_path`（通常是`public`）中。如果它不在搜索路径中，普通的大小写敏感的`text`操作符将会取而代之。
- 比较小写字符串的方法不能正确处理一些Unicode特殊情况，例如当一个大写字母与两个小写字母相等时。因此Unicode区分了`case mapping` 和 `case folding`。用非确定顺序代替`citext`以便正确的处理。

## H. 9. 5. 作者

David E. Wheeler <david@kineticode.com>

受 Donald Fraser 的`citext`模块启发。

## H. 10. cube

这个模块实现了一种数据类型`cube`来表示多维立方体。

此模块被视为“trusted”，也就是说，它可以由在当前数据库上具有 `CREATE` 权限的非超级用户安装。



## H. 10. 1. 语法

Table H. 展示了cube类型有效的外部表示。x、y等表示浮点数。

Table H. 2. 立方体外部表示

外部语法	含义
x	一个一维点（或者长度为零的一维区间）
(x)	同上
x1, x2, ..., xn	n-维空间中的一个点，内部表示为一个零容积立方体
(x1, x2, ..., xn)	同上
(x), (y)	开始于x并且结束于y的一个一维区间，反之亦然。顺序并不重要
[(x), (y)]	同上
(x1, ..., xn), (y1, ..., yn)	一个 n-维立方体，用它的对角顶点对表示
[(x1, ..., xn), (y1, ..., yn)]	同上

一个立方体的对角录入的顺序无关紧要。如果需要创建一种统一的“左下 — 右上”的内部表示，cube函数会自动地交换值。当角重合时，cube只存储一个角和一个“is point”标志，这样避免浪费空间。

输入中的空白空间会被忽略，因此[(x), (y)]与[( x ), ( y )]相同。

## H. 10. 2. 精度

值在内部被存储为 64 位浮点数。这意味着超过 16 位有效位的数字将被截断。

## H. 10. 3. 用法

Table H. 展示了为类型cube提供的专用的操作符。

Table H. 3. 立方体操作符

操作符	描述
cube && cube	→ boolean 立方体是否有重叠?
cube @> cube	→ boolean 第一个立方体是否包含第二个立方体?
cube <@ cube	→ boolean 第一个立方体是否包含在第二个立方体里面?
cube -> integer	→ float8 提取立方体的第 n 个坐标，(从1开始计数)。
cube ~> integer	→ float8 提取立方体的第 n 个坐标，以下列方式计数：n = 2 * k - 1 表示第k个维度的下限，n = 2 * k 表示第k个维度的上限。负的n表示相应正坐标的倒数值。此操作符专为 KNN-GiST 支持而设计。
cube <-> cube	→ float8 计算两个立方体之间的欧几里德距离。
cube <#> cube	→ float8

操作符	描述
	计算两个立方体之间的直线 (L-1 度量) 距离。
<code>cube &lt;=&gt; cube → float8</code>	计算两个立方体之间的切比雪夫 (L-inf 度量) 距离。

除上述操作符外, Table 9.1中显示的常用比较操作符也可用于cube类型。这些操作符首先比较第一个坐标, 如果它们相等再比较第二个坐标等等。它们主要为支持cube的 b树索引操作符类而存在, 这类操作符对支持cube列上的 UNIQUE 约束等很有用。否则, 这种排序没有太大的实际作用。

cube模块也为cube值提供了一个 GiST 索引操作符类。cube GiST 索引可以被用于在WHERE子句中通过=、&&、@以及<@操作符来搜索值。

此外, cube GiST 索引可以被用在ORDER BY子句中通过度量操作符<->、<#>和<=>来查找最近邻。例如, 3-D 点(0.5, 0.5, 0.5)的最近邻可以用下面的查询很快地找到:

```
SELECT c FROM test ORDER BY c <-> cube(array[0.5, 0.5, 0.5]) LIMIT 1;
```

也可以用这种方式使用~>操作符来高效地检索通过选定坐标排序后的前几个值。例如, 可以用下面的查询得到通过第一个坐标 (左下角) 升序排列后的前几个立方体:

```
SELECT c FROM test ORDER BY c ~> 1 LIMIT 5;
```

以及得到通过右上角第一个坐标降序排列后的 2-D 立方体:

```
SELECT c FROM test ORDER BY c ~> 3 DESC LIMIT 5;
```

Table H.展示了可用的函数。

Table H.4. 立方体函数

函数	描述 示例
<code>cube ( float8 ) → cube</code>	制作一个一维立方体, 其两个坐标都是相同的。 <code>cube(1) → (1)</code>
<code>cube ( float8, float8 ) → cube</code>	制作一个一维立方体。 <code>cube(1, 2) → (1), (2)</code>
<code>cube ( float8[] ) → cube</code>	使用数组定义的坐标制作一个零值的立方体 <code>cube(ARRAY[1, 2, 3]) → (1, 2, 3)</code>
<code>cube ( float8[], float8[] ) → cube</code>	用由两个数组定义的右上和左下坐标制造一个立方体, 两个数组必须等长。 <code>cube(ARRAY[1, 2], ARRAY[3, 4]) → (1, 2), (3, 4)</code>
<code>cube ( cube, float8 ) → cube</code>	在一个现有的立方体上增加一维来制造一个新立方体, 对新坐标的各个端点都采用相同的值。这可以用于从计算得到的值逐渐地构建立方体。 <code>cube(' (1, 2), (3, 4)' ::cube, 5) → (1, 2, 5), (3, 4, 5)</code>
<code>cube ( cube, float8, float8 ) → cube</code>	在一个现有的立方体上增加一维来制造一个新立方体。这可以用于从计算得到的值逐渐地构建立方体。

函数	描述 示例
	<code>cube(' (1,2), (3,4)' ::cube, 5, 6) → (1, 2, 5), (3, 4, 6)</code>
<code>cube_dim ( cube ) → integer</code>	返回该立方体的维数。 <code>cube_dim(' (1,2), (3,4)') → 2</code>
<code>cube_ll_coord ( cube, integer ) → float8</code>	返回一个立方体的左下角的第 n个坐标值 <code>cube_ll_coord(' (1,2), (3,4)', 2) → 2</code>
<code>cube_ur_coord ( cube, integer ) → float8</code>	返回一个立方体的右上角的第n个坐标值 <code>cube_ur_coord(' (1,2), (3,4)', 2) → 4</code>
<code>cube_is_point ( cube ) → boolean</code>	如果一个立方体是一个点则返回真，也就是两个定义点相同。 <code>cube_is_point(cube(1,1)) → t</code>
<code>cube_distance ( cube, cube ) → float8</code>	返回两个立方体之间的距离。如果两个都是点，这就是普通距离函数。 <code>cube_distance(' (1,2)', ' (3,4)') → 2.8284271247461903</code>
<code>cube_subset ( cube, integer[] ) → cube</code>	从一个现有的立方体制造一个新立方体，使用来自于一个数组的维索引列表。它可以被用来抽取一个单一维度的端点，或者它可以被用来去除维度，或者按照需要对它们重新排序。 <code>cube_subset(cube(' (1,3,5), (6,7,8)'), ARRAY[2]) → (3), (7)</code> <code>cube_subset(cube(' (1,3,5), (6,7,8)'), ARRAY[3,2,1,1]) → (5, 3, 1, 1), (8, 7, 6, 6)</code>
<code>cube_union ( cube, cube ) → cube</code>	产生两个立方体的并集。 <code>cube_union(' (1,2)', ' (3,4)') → (1, 2), (3, 4)</code>
<code>cube_inter ( cube, cube ) → cube</code>	产生两个立方体的交集。 <code>cube_inter(' (1,2)', ' (3,4)') → (3, 4), (1, 2)</code>
<code>cube_enlarge ( c cube, r double, n integer ) → cube</code>	用一个指定的半径r在至少n个维度上增加立方体的尺寸。如果该半径是负值，则该立方体会收缩。所有已定义的维度都会按照半径r被改变。左下坐标按照 r被减小并且右上坐标按照 r被增加。如果一个左下坐标被增加得超过对应的右上坐标（这只会发生在r < 0 时），则两个坐标会被设置为它们的均值。如果n大于已定义的维度数并且该立方体被增加（r > 0），则额外的维度会被加入以让维度数达到n，对于额外的坐标将使用 0 作为初始值。这个函数可用来创建围绕一个点的边界框以搜索临近点。 <code>cube_enlarge(' (1,2), (3,4)', 0.5, 3) → (0.5, 1.5, -0.5), (3.5, 4.5, 0.5)</code>

## H. 10. 4. 默认值

我相信这个并：

```
select cube_union(' (0,5,2), (2,3,1)', '0');
cube_union
-----
(0, 0, 0), (2, 5, 2)
(1 row)
```

不会与常识矛盾，下面的交也不会

```
select cube_inter(' (0, -1), (1, 1)', ' (-2), (2)');
cube_inter
-----
(0, 0), (1, 0)
(1 row)
```

在所有不同维度立方体的二元操作中，我假定低纬度的那一个要做笛卡尔投影，即为字符串表示中被省略的坐标取零。上面的例子等同于：

```
cube_union(' (0, 5, 2), (2, 3, 1)', ' (0, 0, 0), (0, 0, 0)');
cube_inter(' (0, -1), (1, 1)', ' (-2, 0), (2, 0)');
```

下列包含谓词使用点语法，不过实际上第二个参数在内部被表示为一个盒子。这种语法让我们不必定义一种单独的点类型以及用于（盒子，点）谓词的函数。

```
select cube_contains(' (0, 0), (1, 1)', ' 0.5, 0.5');
cube_contains
-----
t
(1 row)
```

## H. 10.5. 注解

用法的例子可见回归测试sql/cube.sql。

为了不容易出问题，对于立方体的维度数有 100 的限制。如果你想要更大的立方体，可以在cubedata.h中修改。

## H. 10.6. 工作人员

原作者：Gene Selkov, Jr. <selkovjr@mcs.anl.gov>，数学与计算机科学部，阿尔贡国家实验室。

我的感谢主要要献给 Joe Hellerstein 教授(<https://dsf.berkeley.edu/jmh/>)，他阐明了 GiST (<http://gist.cs.berkeley.edu/>)，还要送给他以前的学生 Andy Dong，他为 Illustra 编写了例子。我也对所有的 Postgres 开发者（现在的和以前的）心存感激，他们让我能够创造自己的世界并且宁静地生活在其中。我也要感谢阿尔贡实验室和美国能源局对我多年数据库研究的支持。

这个包的小更新由 Bruno Wolff III <bruno@wolff.to>于 2002 年 8/9 月完成。这些修改包括将精度从单精度改为双精度以及增加了一些新的函数。

额外的更新由 Joshua Reich <josh@root.net> 在 2006 年 7 月做出。其中包括cube(float8[], float8[])并且将代码从废弃的 V0 协议改到 V1 调用协议。

## H. 11. dblink

dblink是一个支持在一个数据库会话中连接到其他LightDB数据库的模块。

还可以看看postgres\_fdw，它以一种更现代和更加兼容标准的架构提供了相同的功能。

## dblink\_connect

dblink\_connect — 打开一个到远程数据库的持久连接

## Synopsis

```
dblink_connect(text connstr) 返回 text
dblink_connect(text connname, text connstr) 返回 text
```

## 描述

dblink\_connect() 建立一个到远程LightDB数据库的连接。要联系的服务器和数据库通过一个标准的libpq连接串来标识。可以选择将一个名字赋予给该连接。多个命名的连接可以被一次打开，但是一次只允许一个未命名连接。连接将会持续直到被关闭或者数据库会话结束。

连接串也可以是一个现存外部服务器的名字。在使用外部服务器时，我们推荐使用外部数据包装器dblink\_fdw。见下面的例子，以及CREATE SERVER和CREATE USER MAPPING。

## 参数

connname

要用于这个连接的名字。如果被忽略，将打开一个未命名连接并且替换掉任何现有的未命名连接。

connstr

libpq-风格的连接信息串，例如 hostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswd。详见Section 32.1.1 此外，还可以是一个外部服务器的名字。

## 返回值

返回状态，它总是OK（因为任何错误会导致该函数抛出一个错误而不是返回）。

## 注解

如果不可信用户能够访问一个没有采用安全方案使用模式的数据库，应该在开始每个会话时从search\_path中移除公共可写的方案。例如，可以把options=-csearch\_path=增加到connstr。这种考虑不是特别针对dblink，它适用于每一种执行任意SQL命令的接口。

只有超级用户能够使用dblink\_connect来创建无口令认证连接。如果非超级用户需要这种能力，使用dblink\_connect\_u。

选择包含等号的连接名是不明智的，因为这会产生与在其他dblink函数中的连接信息串混淆的风险。

## 例子

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
-----
OK
```

```
(1 row)

SELECT dblink_connect('myconn', 'dbname=postgres options=-csearch_path=');
 dblink_connect
-----
OK
(1 row)

-- FOREIGN DATA WRAPPER functionality
-- Note: local connection must require password authentication for this to work
-- properly
--     Otherwise, you will receive the following error from dblink_connect():
--     ERROR: password is required
--     DETAIL: Non-superuser cannot connect if the server does not request a
--     password.
--     HINT: Target server's authentication method must be changed.

CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS (hostaddr
 '127.0.0.1', dbname 'contrib_regression');

CREATE USER regress_dblink_user WITH PASSWORD 'secret';
CREATE USER MAPPING FOR regress_dblink_user SERVER fdtest OPTIONS (user
 'regress_dblink_user', password 'secret');
GRANT USAGE ON FOREIGN SERVER fdtest TO regress_dblink_user;
GRANT SELECT ON TABLE foo TO regress_dblink_user;

\set ORIGINAL_USER :USER
\c - regress_dblink_user
SELECT dblink_connect('myconn', 'fdtest');
 dblink_connect
-----
OK
(1 row)

SELECT * FROM dblink('myconn', 'SELECT * FROM foo') AS t(a int, b text, c
 text[]);
 a | b |          c
-----+-----+-----
 0 | a | {a0, b0, c0}
 1 | b | {a1, b1, c1}
 2 | c | {a2, b2, c2}
 3 | d | {a3, b3, c3}
 4 | e | {a4, b4, c4}
 5 | f | {a5, b5, c5}
 6 | g | {a6, b6, c6}
 7 | h | {a7, b7, c7}
 8 | i | {a8, b8, c8}
 9 | j | {a9, b9, c9}
10 | k | {a10, b10, c10}
(11 rows)

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM regress_dblink_user;
REVOKE SELECT ON TABLE foo FROM regress_dblink_user;
DROP USER MAPPING FOR regress_dblink_user SERVER fdtest;
DROP USER regress_dblink_user;
DROP SERVER fdtest;
```

## dblink\_connect\_u

dblink\_connect\_u — 不安全地打开一个到远程数据库的持久连接

## Synopsis

dblink\_connect\_u(text connstr) 返回 text

dblink\_connect\_u(text connname, text connstr) 返回 text

## 描述

dblink\_connect\_u() 和 dblink\_connect() 一样，不过它将允许非超级用户使用任意认证方式来连接。

如果远程服务器选择了一种不涉及口令的认证方式，那么可能发生模仿以及后续的扩大权限，因为该会话看起来像由运行 LightDB 的用户发起的。此外，即使远程服务器不要求一个口令，也可能从服务器环境提供该口令，例如一个属于服务器用户的 `~/.pgpass` 文件。这带来的不只是模仿的风险，而且还有将口令暴露给不可信的远程服务器的风险。因此，dblink\_connect\_u() 最初是用所有从 PUBLIC 撤销的特权安装的，这让它只能被超级用户调用。在某些情况中，为 dblink\_connect\_u() 授予 EXECUTE 权限给可信的指定用户是合适的，但是必须小心。我们也推荐任何属于服务器用户的 `~/.pgpass` 文件不能包含任何指定了一个通配符主机名的记录。

详见 dblink\_connect()。

## dblink\_disconnect

dblink\_disconnect — 关闭一个到远程数据库的持久连接

### Synopsis

```
dblink_disconnect() 返回 text  
dblink_disconnect(text connname) 返回 text
```

### 描述

dblink\_disconnect() 关闭一个之前被 dblink\_connect() 打开的连接。不带参数的形式关闭一个未命名连接。

### 参数

connname

要被关闭的命名连接的名字。

### 返回值

它总是OK（因为任何错误会导致该函数抛出一个错误而不是返回）。

### 例子

```
SELECT dblink_disconnect();  
dblink_disconnect  
-----  
OK  
(1 row)  
  
SELECT dblink_disconnect('myconn');  
dblink_disconnect  
-----  
OK  
(1 row)
```



## dblink

dblink — 在一个远程数据库中执行一个查询

## Synopsis

```
dblink(text connname, text sql [, bool fail_on_error]) 返回记录集
dblink(text connstr, text sql [, bool fail_on_error]) 返回记录集
dblink(text sql [, bool fail_on_error]) 返回记录集
```

## 描述

dblink在一个远程数据库中执行一个查询（通常是一个SELECT，但是也可以是任意返回行的SQL 语句）。

当给定两个text参数时，第一个被首先作为一个持久连接的名称进行查找；如果找到，该命令会在该连接上被执行。如果没有找到，第一个参数被视作一个用于dblink\_connect的连接信息字符串，并且被指出的连接只是在这个命令的持续期间被建立。

## 参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

connstr

如之前为dblink\_connect所描述的一个连接信息字符串。

sql

你希望在远程数据库中执行的 SQL 查询，例如select \* from foo。

fail\_on\_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数不返回行。

## 返回值

该函数返回查询产生的行。因为dblink能与任何查询一起使用，它被声明为返回record，而不是指定任意特定的列集合。这意味着你必须指定在调用的查询中所期待的列集合 — 否则LightDB将不知道会得到什么。这里是一个例子：

```
SELECT *
FROM dblink('dbname=mydb options=-csearch_path=',
            'select proname, prosrc from pg_proc')
AS t1(proname name, prosrc text)
WHERE proname LIKE 'bytea%';
```

FROM子句的“alias”部分必须指定函数将返回的列名及类型（在一个别名中指定列名实际上是标准 SQL 语法，但是指定列类型是一种LightDB扩展）。这允许系统在尝试执行该函数之前就理解\*将展开成什么，以及WHERE子句中的proname指的什么。在运行时，如果来自远程数据库的实际查询结果和FROM子句中显示的列数不同，将会抛出一个错误。不过，列名不需要匹配，并且dblink并不坚持精确地匹配类型。只要被返回的数据字符串是FROM子句中声明的列类型的合法输入，它就将会成功。

## 注解

一种将预定义查询用于dblink的方便方法是创建一个视图。这允许列类型信息被埋藏在该视图中，而不是在每一个查询中都拼写出来。例如：

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres options=-csearch_path='
                'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

## 例子

```
SELECT * FROM dblink('dbname=postgres options=-csearch_path='
                    'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname | prosrc
-----+-----
 byteacat | byteacat
 byteaeq  | byteaeq
 bytealt  | bytealt
 byteale  | byteale
 byteagt  | byteagt
 byteage  | byteage
 byteane  | byteane
 byteacmp | byteacmp
 bytealike | bytealike
 byteanlike | byteanlike
 byteain  | byteain
 byteaout | byteaout
(12 rows)
```

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
-----
 OK
(1 row)
```

```
SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname | prosrc
-----+-----
 byteacat | byteacat
 byteaeq  | byteaeq
 bytealt  | bytealt
 byteale  | byteale
 byteagt  | byteagt
 byteage  | byteage
 byteane  | byteane
 byteacmp | byteacmp
 bytealike | bytealike
 byteanlike | byteanlike
 byteain  | byteain
 byteaout | byteaout
```

(12 rows)

```
SELECT dblink_connect('myconn', 'dbname=regression options=-csearch_path=');
dblink_connect
```

-----  
OK

(1 row)

```
SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
```

proname	prosrc
bytearecv	bytearecv
byteasend	byteasend
byteale	byteale
byteagt	byteagt
byteage	byteage
byteane	byteane
byteacmp	byteacmp
bytealike	bytealike
byteanlike	byteanlike
byteacat	byteacat
byteaeq	byteaeq
bytealt	bytealt
byteain	byteain
byteaout	byteaout

(14 rows)

## dblink\_exec

dblink\_exec — 在一个远程数据库中执行一个命令

## Synopsis

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

## 描述

dblink\_exec在一个远程数据库中执行一个命令（也就是，任何不返回行的 SQL 语句）。

当给定两个text参数时，第一个被首先作为一个持久连接的名称进行查找；如果找到，该命令会在该连接上被执行。如果没有找到，第一个参数被视作一个用于dblink\_connect的连接信息字符串，并且被指出的连接只是在这个命令的持续期间被建立。

## 参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

connstr

如之前为dblink\_connect所描述的一个连接信息字符串。

sql

你希望在远程数据库中执行的 SQL 命令，例如insert into foo values(0, 'a', '{ "a0", "b0", "c0" }')。

fail\_on\_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数的返回值被设置为ERROR。

## 返回值

返回状态，可能是命令的状态字符串或ERROR。

## 例子

```
SELECT dblink_connect(' dbname=dblink_test_standby');
 dblink_connect
-----
OK
(1 row)

SELECT dblink_exec(' insert into foo values(21, ''z'', '' { "a0", "b0", "c0" } '');');
 dblink_exec
-----
INSERT 943366 1
```

(1 row)

```
SELECT dblink_connect('myconn', 'dbname=regression');
dblink_connect
```

-----  
OK

(1 row)

```
SELECT dblink_exec('myconn', 'insert into foo values(21, ''z'',
''{"a0","b0","c0"}'');');
dblink_exec
```

-----  
INSERT 6432584 1

(1 row)

```
SELECT dblink_exec('myconn', 'insert into pg_class values (''foo''),false);
NOTICE: sql error
DETAIL: ERROR: null value in column "renamespace" violates not-null
constraint
```

dblink\_exec

-----  
ERROR

(1 row)

## dblink\_open

dblink\_open — 在一个远程数据库中打开一个游标

## Synopsis

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) 返回 text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) 返回 text
```

## 描述

dblink\_open() 在一个远程数据库中打开一个游标。该游标能够随后使用 dblink\_fetch() 和 dblink\_close() 进行操纵。

## 参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

cursorname

要赋予给这个游标的名称。

sql

你希望在远程数据库中执行的SELECT语句，例如 `select * from pg_class`。

fail\_on\_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数的返回值被设置为ERROR。

## 返回值

返回状态，OK或者ERROR。

## 注解

因为一个游标只能在一个事务中持续，如果远端还没有在一个事务中，dblink\_open会在远端开始一个显式事务块（BEGIN）。当匹配的dblink\_close被执行时，这个事务将再次被关闭。注意如果你使用dblink\_exec在dblink\_open和dblink\_close之间改变数据，并且接着发生了一个错误或者你在dblink\_close之前使用了dblink\_disconnect，你的更改将被丢失，因为事务将被中止。

## 例子

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
-----
      OK
(1 row)
```

```
SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
dblink_open
-----
OK
(1 row)
```

## dblink\_fetch

dblink\_fetch — 从一个远程数据库中的打开的游标返回行

## Synopsis

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) 返回 record 集合  
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error])  
返回 record 集合
```

## 描述

dblink\_fetch从一个之前由dblink\_open建立的游标中取得行。

## 参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

cursorname

要从中取数据的游标名。

howmany

要检索的最大行数。从当前游标位置向前的接下来howmany个行会被取出。一旦该游标已经到达了它的末端，将不会产生更多行。

fail\_on\_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数不返回行。

## 返回值

该函数返回从游标中取出的行。要使用这个函数，你将需要指定想要的列集合，如前面dblink中所讨论的。

## 注解

当FROM子句中指定的返回列的数量和远程游标返回的实际列数不匹配时，将抛出一个错误。在这个事件中，远程游标仍会被前进错误没发生时应该前进的行数。对于远程FETCH完成之后在本地查询中发生的任何其他错误，情况也是一样。

## 例子

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');  
dblink_connect  
-----  
OK  
(1 row)
```



```
SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname
like ''bytea%'');
dblink_open
```

```
-----
```

```
OK
(1 row)
```

```
SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
```

```
funcname | source
-----+-----
```

```
byteacat | byteacat
byteacmp | byteacmp
byteaeq  | byteaeq
byteage  | byteage
byteagt  | byteagt
```

```
(5 rows)
```

```
SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
```

```
funcname | source
-----+-----
```

```
byteain  | byteain
byteale  | byteale
bytealike| bytealike
bytealt  | bytealt
byteane  | byteane
```

```
(5 rows)
```

```
SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
```

```
funcname | source
-----+-----
```

```
byteanlike | byteanlike
byteaout   | byteaout
```

```
(2 rows)
```

```
SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
```

```
funcname | source
-----+-----
```

```
(0 rows)
```

## dblink\_close

dblink\_close — 关闭一个远程数据库中的游标

## Synopsis

```
dblink_close(text cursorname [, bool fail_on_error]) 返回 text
dblink_close(text connname, text cursorname [, bool fail_on_error]) 返回 text
```

## 描述

dblink\_close关闭一个之前由dblink\_open打开的游标。

## 参数

connname

要使用的连接名。忽略这个参数将使用未命名连接。

cursorname

要关闭的游标名。

fail\_on\_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数的返回值被设置为ERROR。

## 返回值

返回状态，OK或者ERROR。

## 注解

如果dblink\_open开始了一个显式事务块，并且这是这个连接中最后一个保持打开的游标，dblink\_close将发出匹配的COMMIT。

## 例子

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
-----
OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-----
OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
```

-----  
OK  
(1 row)

## dblink\_get\_connections

`dblink_get_connections` — 返回所有打开的命名 `dblink` 连接的名称

### Synopsis

`dblink_get_connections()` 返回 `text[]`

### 描述

`dblink_get_connections` 返回一个数组，其中是所有打开的命名 `dblink` 连接的名称。

### 返回值

返回一个连接名称的文本数组，如果没有则为 `NULL`。

### 例子

```
SELECT dblink_get_connections();
```

## dblink\_error\_message

dblink\_error\_message — 得到在命名连接上的最后一个错误消息

### Synopsis

dblink\_error\_message(text connname) 返回 text

### 描述

dblink\_error\_message为一个给定连接取得最近的远程错误消息。

### 参数

connname

要使用的连接名。

### 返回值

返回最后一个错误消息，如果在这个连接上没有错误则返回一个OK。

### 注释

当异步查询由dblink\_send\_query启动时，与连接相关的错误消息可能不会得到更新，直到服务器的响应消息被用掉。这通常意味着dblink\_is\_busy或dblink\_get\_result应在dblink\_error\_message之前被调用，以使得异步查询产生的任何错误都是可见的。

### 例子

```
SELECT dblink_error_message('dtest1');
```

## dblink\_send\_query

dblink\_send\_query — 发送一个异步查询到远程数据库

### Synopsis

dblink\_send\_query(text connname, text sql) 返回 int

### 描述

dblink\_send\_query发送一个要被异步执行的查询，也就是不需要立即等待结果。在该连接上不能有还在处理中的异步查询。

在成功地派送一个异步查询后，可以用dblink\_is\_busy检查完成状态，并且结果最终由dblink\_get\_result收集。也可以使用dblink\_cancel\_query尝试取消一个活动中的异步查询。

### 参数

connname

要使用的连接名。

sql

你希望在远程数据库中执行的 SQL 语句，例如select \* from pg\_class。

### 返回值

如果查询被成功地派送返回 1，否则返回 0。

### 例子

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

## dblink\_is\_busy

dblink\_is\_busy — 检查连接是否正在忙于一个异步查询

## Synopsis

dblink\_is\_busy(text connname) 返回 int

## 描述

dblink\_is\_busy测试是否一个异步查询正在进行中。

## 参数

connname

要检查的连接名。

## 返回值

如果连接正忙则返回 1，如果不忙则返回 0。如果这个函数返回 0，dblink\_get\_result将被保证不会阻塞。

## 例子

```
SELECT dblink_is_busy('dtest1');
```

## dblink\_get\_notify

dblink\_get\_notify — 在一个连接上检索异步通知

### Synopsis

dblink\_get\_notify() 返回 (notify\_name text, be\_pid int, extra text) 集合  
 dblink\_get\_notify(text connname) 返回 (notify\_name text, be\_pid int, extra text) 集合

### 描述

dblink\_get\_notify在一个未命名连接或者一个指定的命名连接上检索通知。要通过 dblink 接收通知，首先必须使用dblink\_exec发出LISTEN。详见LISTEN和NOTIFY。

### 参数

connname

要在其上得到通知的命名连接的名称。

### 返回值

返回 (notify\_name text, be\_pid int, extra text) 集合，或者一个空集。

### 例子

```
SELECT dblink_exec('LISTEN virtual');
dblink_exec
-----
LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-----+-----+-----
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-----+-----+-----
 virtual    | 1229  |
(1 row)
```



## dblink\_get\_result

dblink\_get\_result — 得到一个异步查询结果

## Synopsis

dblink\_get\_result(text connname [, bool fail\_on\_error]) 返回 record 集合

## 描述

dblink\_get\_result收集之前dblink\_send\_query发送的一个异步查询的结果。如果该查询还没有完成，dblink\_get\_result将等待直到它完成。

## 参数

connname

要使用的连接名。

fail\_on\_error

如果为真（忽略时的默认值），那么在连接的远端抛出的一个错误也会导致本地抛出一个错误。如果为假，远程错误只在本地被报告为一个 NOTICE，并且该函数不返回行。

## 返回值

对于一个异步查询（也就是一个返回行的 SQL 语句），该函数返回查询产生的行。要使用这个函数，你将需要指定所期待的列集合，如前面为dblink所讨论的那样。

对于一个异步命令（也就是一个不返回行的 SQL 语句），该函数返回一个只有单个文本列的单行，其中包含了该命令的状态字符串。仍必须在调用的FROM子句中指定结果将具有一个单一文本行。

## 注解

如果dblink\_send\_query返回 1，这个函数就必须被调用。对每一个已发送的查询都必须调用一次这个函数，并且在连接再次可用之前还要多调用一次来得到一个空结果集。

当使用dblink\_send\_query和dblink\_get\_result时，在将结果集中的任何一行返回给本地查询处理器之前，dblink将取得整个远程查询结果。如果该查询返回大量的行，这可能会导致本地会话中短暂的内存膨胀。最好将这样的查询用dblink\_open打开成一个游标并且接着每次取得数量可管理的行。也可以使用简单的dblink()，它会避免缓冲大型结果集到磁盘上导致的内存膨胀。

## 例子

```
contrib_regression=# SELECT dblink_connect('dtest1',
'dbname=contrib_regression');
dblink_connect
-----
OK
(1 row)

contrib_regression=# SELECT * FROM
```

```
contrib_regression=# dblink_send_query('dtest1', 'select * from foo where f1 <
3') AS t1;
t1
-----
 1
(1 row)
```

```
contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
text, f3 text[]);
f1 | f2 |      f3
-----+-----+-----
 0 | a  | {a0, b0, c0}
 1 | b  | {a1, b1, c1}
 2 | c  | {a2, b2, c2}
(3 rows)
```

```
contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
text, f3 text[]);
f1 | f2 | f3
-----+-----+-----
(0 rows)
```

```
contrib_regression=# SELECT * FROM
contrib_regression=# dblink_send_query('dtest1', 'select * from foo where f1 <
3; select * from foo where f1 > 6') AS t1;
t1
-----
 1
(1 row)
```

```
contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
text, f3 text[]);
f1 | f2 |      f3
-----+-----+-----
 0 | a  | {a0, b0, c0}
 1 | b  | {a1, b1, c1}
 2 | c  | {a2, b2, c2}
(3 rows)
```

```
contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
text, f3 text[]);
f1 | f2 |      f3
-----+-----+-----
 7 | h  | {a7, b7, c7}
 8 | i  | {a8, b8, c8}
 9 | j  | {a9, b9, c9}
10 | k  | {a10, b10, c10}
(4 rows)
```

```
contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2
text, f3 text[]);
f1 | f2 | f3
-----+-----+-----
(0 rows)
```

## dblink\_cancel\_query

dblink\_cancel\_query — 在命名连接上取消任何活动查询

### Synopsis

dblink\_cancel\_query(text connname) 返回 text

### 描述

dblink\_cancel\_query尝试命名连接上正在进行的任何查询。注意这不一定会成功（例如，远程查询可能已经结束）。一个取消请求仅仅提高了该查询将很快失败的几率。你仍必须完成通常的查询协议，例如通过调用dblink\_get\_result。

### 参数

connname

要使用的连接名。

### 返回值

如果取消请求已经被发送，则返回OK；如果失败，则返回一个错误消息的文本。

### 例子

```
SELECT dblink_cancel_query('dtest1');
```

## dblink\_get\_pkey

dblink\_get\_pkey — 返回一个关系的主键域的位置和域名称

## Synopsis

dblink\_get\_pkey(text relname) 返回 dblink\_pkey\_results 集合

## 描述

dblink\_get\_pkey提供有关于本地数据库中一个关系的主键的信息。这有时候有助于生成要被发送到远程数据库的查询。

## 参数

relname

一个本地关系的名称，例如foo或者myschema.mytab。如果该名称是大小写混合的或包含特殊字符，要包括双引号，例如"FooBar"；如果没有引号，字符串将被折叠到小写形式。

## 返回值

为每一个主键域返回一行，如果该关系没有主键则不返回行。结果行类型被定义为：

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

position列值可以从 1 到 N，它是该域在主键中的编号，而不是在表列中的编号。

## 例子

```
CREATE TABLE foobar (  
    f1 int,  
    f2 int,  
    f3 int,  
    PRIMARY KEY (f1, f2, f3)  
);  
CREATE TABLE  
  
SELECT * FROM dblink_get_pkey(' foobar');  
 position | colname  
-----+-----  
        1 | f1  
        2 | f2  
        3 | f3  
(3 rows)
```

## dblink\_build\_sql\_insert

`dblink_build_sql_insert` — 使用一个本地元组构建一个 `INSERT` 语句，将主键域值替换为提供的值

### Synopsis

```
dblink_build_sql_insert(text relname,  
                        int2vector primary_key_attnums,  
                        integer num_primary_key_atts,  
                        text[] src_pk_att_vals_array,  
                        text[] tgt_pk_att_vals_array) 返回 text
```

### 描述

`dblink_build_sql_insert`在选择性地将一个本地表复制到一个远程数据库时很有用。它基于主键从本地表选择一行，并且接着构建一个复制该行的`INSERT`命令，但是其中主键值被替换为最后一个参数中的值（要创建该行的一个准确拷贝，只要为最后两个参数指定相同的值）。

### 参数

`relname`

一个本地关系的名称，例如`foo`或者`myschema.mytab`。如果该名称是大小写混合的或包含特殊字符，要包括双引号，例如`"FooBar"`；如果没有引号，字符串将被折叠到小写形式。

`primary_key_attnums`

主键域的属性号（从 1 开始），例如`1 2`。

`num_primary_key_atts`

主键域的数量。

`src_pk_att_vals_array`

要被用来查找本地元组的主键域值。每一个域都被表示为文本形式。如果没有行具有这些主键值，则抛出一个错误。

`tgt_pk_att_vals_array`

要被替换到结果`INSERT`命令中的主键域值。每一个域被表示为文本形式。

### 返回值

将要求的 `SQL` 语句返回为文本。

### 注解

自`LightDB 9.0` 开始，`primary_key_attnums`中的属性号被解释为逻辑列号，对应于列在`SELECT * FROM relname`中的位置。之前的版本将属性号解释为物理列位置。如果指示出的列的左边有任意列在该表的生存期内被删除，这两种解释就有区别。

## 例子

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
       dblink_build_sql_insert
-----
INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

## dblink\_build\_sql\_delete

dblink\_build\_sql\_delete — 使用所提供的主键域值构建一个 DELETE 语句

### Synopsis

```
dblink_build_sql_delete(text relname,  
                        int2vector primary_key_attnums,  
                        integer num_primary_key_atts,  
                        text[] tgt_pk_att_vals_array) 返回 text
```

### 描述

dblink\_build\_sql\_delete在选择性地将一个本地表复制到一个远程数据库时很有用。它构建一个 SQL DELETE命令用来删除具有给定主键值的行。

### 参数

relname

一个本地关系的名称，例如foo或者myschema.mytab。如果该名称是大小写混合的或包含特殊字符，要包括双引号，例如"FooBar"；如果没有引号，字符串将被折叠到小写形式。

primary\_key\_attnums

主键域的属性号（从 1 开始），例如1 2。

num\_primary\_key\_atts

主键域的数量。

tgt\_pk\_att\_vals\_array

要用在结果DELETE命令中的主键域值。每一个域都被表示为文本形式。

### 返回值

将要求的 SQL 语句返回为文本。

### 注解

自LightDB 9.0 开始，primary\_key\_attnums中的属性号被解释为逻辑列号，对应于列在SELECT \* FROM relname中的位置。之前的版本将属性号解释为物理列位置。如果指示出的列的左边有任意列在该表的生存期内被删除，这两种解释就有区别。

### 例子

```
SELECT dblink_build_sql_delete('MyFoo', '1 2', 2, '{"1", "b"}');  
-----  
DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'  
(1 row)
```

## dblink\_build\_sql\_update

`dblink_build_sql_update` — 使用一个本地元组构建一个 UPDATE 语句，将主键域值替换为提供的值

### Synopsis

```
dblink_build_sql_update(text relname,  
                        int2vector primary_key_attnums,  
                        integer num_primary_key_atts,  
                        text[] src_pk_att_vals_array,  
                        text[] tgt_pk_att_vals_array) 返回 text
```

### 描述

`dblink_build_sql_update`在选择性地将一个本地表复制到一个远程数据库时很有用。它从本地表基于主键选择一行，并且接着构建一个 SQL UPDATE命令来复制该行，但是其中的主键值被替换为最后一个参数中的值（要创建该行的一个准确拷贝，只要为最后两个参数指定相同的值）。UPDATE命令总是为该行的所有域赋值 — 这个函数与`dblink_build_sql_insert`之间的主要区别是它假定目标行已经存在于远程表中。

### 参数

`relname`

一个本地关系的名称，例如`foo`或者`myschema.mytab`。如果该名称是大小写混合的或包含特殊字符，要包括双引号，例如`"FooBar"`；如果没有引号，字符串将被折叠到小写形式。

`primary_key_attnums`

主键域的属性号（从 1 开始），例如1 2。

`num_primary_key_atts`

主键域的数量。

`src_pk_att_vals_array`

要被用来查找本地元组的主键域值。每一个域都被表示为文本形式。如果没有行具有这些主键值，则抛出一个错误。

`tgt_pk_att_vals_array`

要用在结果UPDATE命令中的主键域值。每一个域都被表示为文本形式。

### 返回值

将要求的 SQL 语句返回为文本。

### 注解

自LightDB 9.0 开始，`primary_key_attnums`中的属性号被解释为逻辑列号，对应于列在SELECT \* FROM `relname`中的位置。之前的版本将属性号解释为物理列位置。如果指示出的列的左边有任意列在该表的生存期内被删除，这两种解释就有区别。



## 例子

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{ "1", "a" }, { "1", "b" }');
        dblink_build_sql_update
```

```
-----
UPDATE foo SET f1='1', f2='b', f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

## H. 12. dict\_int

dict\_int是一个附加全文搜索词典模板的例子。这个例子词典的动机是控制整数（有符号和无符号）的索引，允许在阻止唯一词数量的过度增长（会严重影响搜索性能）时也能索引这些数字。

这个模块被视为“trusted”，也就是说，它可以由在当前数据库上具有CREATE特权的非超级用户安装。

## H. 12. 1. 配置

该词典接受三个选项：

- maxlen参数指定在一个整数词中允许的最大位数。默认值为 6。
- rejectlong参数指定一个超长整数是否应该被截断或忽略。如果rejectlong为false（默认），该词典返回该整数的第一个数字。如果rejectlong为true，该词典将一个超长整数作为一个停用词对待，因此它将被索引。注意这也意味着这样一个整数不能被搜索。
- absval 参数规定“+”或“-”符号是否将被从整数前删除。缺省为 false。当为true时，在maxlen被应用之前，符号被删除。

## H. 12. 2. 用法

安装dict\_int扩展会使用默认参数创建一个文本搜索模板intdict\_template和一个基于它的词典intdict。你可以修改参数，例如

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

或者创建基于该模板的新词典。

要测试该词典，可以尝试

```
mydb# select ts_lexize('intdict', '12345678');
        ts_lexize
-----
        {123456}
```

但是现实世界的用法将涉及将它包括在一个Chapter 12中描述的文本搜索配置中。看起来像这样：

```
ALTER TEXT SEARCH CONFIGURATION english
        ALTER MAPPING FOR int, uint WITH intdict;
```

## H. 13. dict\_xsyn

dict\_xsyn（扩展同义词字典）是一个附加全文搜索字典模板的例子。这种字典类型将词替换为它们的同义词分组，并且让使用其任一同义词进行搜索变得可能。

### H. 13. 1. 配置

一个dict\_xsyn词典接受以下选项：

- matchorig控制该词典是否接受原生词。默认为true。
- matchsynonyms控制该词典是否接受同义词。默认为false。
- keeporig控制原生词是否被包括在词典的输出中。默认为true。
- keepsynonyms控制同义词是否被包括在词典的输出中。默认为true。
- rules是包含同义词列表的文件的基本名。这个文件必须被存储在\$SHAREDIR/tsearch\_data/（其中\$SHAREDIR表示LightDB安装的共享数据目录）中。它的名称必须以.rules结束（这不包括在rules参数中）。

规则文件具有下面的格式：

- 每一行表示一个单一词的同义词分组，它在该行中首先被给出。同义词被空白分隔，这样：

```
word syn1 syn2 syn3
```

- 井号（#）是注释定界符。它可以出现在一行中的任何位置。该行的剩余部分将被跳过。

例如，可以看看安装在\$SHAREDIR/tsearch\_data/中的xsyn\_sample.rules。

### H. 13. 2. 用法

安装dict\_xsyn扩展会用默认参数创建一个文本搜索模板xsyn\_template以及一个基于它的词典xsyn。你可以修改参数，例如

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

或者基于该模板创建新的词典。

要测试该词典，你可以尝试

```
mydb=# SELECT ts_lexize('xsyn', 'word');
         ts_lexize
```

```
-----
{syn1, syn2, syn3}
```

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY
```

```
mydb=# SELECT ts_lexize('xsyn', 'word');
         ts_lexize
```

```

{word, syn1, syn2, syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false,
MATCHSYNONYMS=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
          ts_lexize
-----
{syn1, syn2, syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true,
MATCHORIG=false, KEEPSYNONYMS=false);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
          ts_lexize
-----
{word}

```

现实世界的用法将涉及将它包括在一个Chapter 12中描述的文本搜索配置中。看起来像这样：

```

ALTER TEXT SEARCH CONFIGURATION english
ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;

```

## H. 14. file\_fdw

file\_fdw模块提供外部数据包装器file\_fdw，它被用来访问服务器的文件系统中的数据文件，或者在服务器上执行程序并读取它们的输出。数据文件或程序输出必须是能够被COPY FROM读取的格式，详见COPY。当前只能读取数据文件。

用这个包装器创建的一个外部表可以有如下选项：

filename

指定要被读取的文件。相对路径是相对于数据目录。必须指定filename或program，但不能同时指定两个。

program

指定要执行的命令。该命令的标准输出将被读取，就像使用COPY FROM PROGRAM一样。必须指定program或filename，但不能同时指定两个。

format

指定数据的格式，和COPY的FORMAT选项相同。

header

指定数据是否具有一个头部行，和COPY的HEADER选项相同。

delimiter

指定数据的定界符字符，和COPY的DELIMITER选项相同。

quote

指定数据的引用字符，和COPY的QUOTE选项相同。

escape

指定数据的转义字符，和COPY的ESCAPE选项相同。

null

指定数据的空字符串，和COPY的NULL选项相同。

encoding

指定数据的编码，和COPY的ENCODING选项相同。

注意虽然COPY允许诸如HEADER的选项不用一个相应的值指定，但是外部表选项语法要求在所有情况下都出现一个值。要激活通常写入没有值的 COPY选项，你可以传递值 TRUE，因为所有这些选项都是布尔值。

使用这个包装器创建的表的一列可以具有下列选项：

force\_not\_null

这是一个布尔选项。如果为真，它指定该列的值不应该与空字符串匹配（也就是表级别的null选项）。这和把该列放在COPY的FORCE\_NOT\_NULL选项中具有相同的效果。

force\_null

这是一个布尔选项。如果为真，它指定匹配空值字符串的列值会被返回为NULL，即使该值被引号引用。如果没有这个选项，只有匹配空值字符串的未被引用的值会被返回为NULL。这在COPY的FORCE\_NULL 选项中列出该列有同样的效果。

COPY的FORCE\_QUOTE选项当前不被file\_fdw支持。

这些选项只能为一个外部表及其列指定，而不能在file\_fdw外部数据包装器的选项中指定，也不能在使用该包装器的服务器或者用户映射的选项中指定。

出于安全原因，改变表级别的选项要求超级用户特权或具有角色pg\_read\_server\_files（使用文件名）或角色pg\_execute\_server\_program（使用程序）的权限：只有特定用户能够控制读取哪个文件或者运行哪个程序。原则上普通用户可以被允许改变其它选项，但是当前还不支持这样做。

当指定program选项时，请记住，选项字符串是通过shell执行的。如果想传递任何参数到来自不受信任的源的命令，必须小心去掉或转义任何对shell来说可能有特殊含义的字符。安全起见，最好使用固定的命令字符串，或者至少避免传递任何用户输入。

对于一个使用file\_fdw的外部表，EXPLAIN显示要读取的文件名或要运行的程序。对于文件来说，除非指定COSTS OFF，否则文件尺寸（以字节计）也会被显示。

### Example H.1. 为 LightDB CSV 日志创建一个外部表

一种file\_fdw的用法是把可用的 LightDB 活动日志变成一个表用于查询。要这样做，首先你必须写日志到一个CSV文件，这里我们将它称为pglog.csv。首先，将file\_fdw安装为一个扩展：

```
CREATE EXTENSION file_fdw;
```

然后创建一个外部服务器：

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
```

现在你已经准备好创建外部数据表。使用CREATE FOREIGN TABLE命令，你将需要为该表定义列、CSV 文件名以及格式：

```
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  backend_type text
) SERVER pglog
OPTIONS ( filename 'log/pglog.csv', format 'csv' );
```

就是这样了 — 现在你能够直接查询你的日志了。当然，在生产中你会需要定义一些方法来处理日志轮转。

## H. 15. fuzzystmatch

fuzzystmatch模块提供多个函数来判断字符串之间的相似性和距离。

### Caution

当前，soundex、metaphone、dmetaphone和dmetaphone\_alt函数使用多字节编码（例如 UTF-8）下工作得不好。

这个模块被视为“trusted”，就是说，它可由在当前数据库上拥有CREATE特权的非超级用户安装。

### H. 15. 1. Soundex

语音表示法系统是一种将相似发音的名字转换成相同的代码来匹配它们的方法。这最初由美国国家统计局在 1880 年、1900 年和 1910 年使用。注意语音表示法对于非英语名称不是很有用。

fuzzystmatch模块提供了两个函数用于语音表示法代码：

soundex(text) 返回 text  
 difference(text, text) 返回 int

soundex函数将一个字符串转换成它的语音表示法代码。difference函数将两个字符串转换成它们的语音表示法代码并且接着报告能匹配代码位置的数量。由于语音表示法代码具有四个字符，结果可以从零到四，零表示没有匹配而四表示完全匹配（因此这个函数的命名并不适当 — similarity才是更合适的名称）。

这里有一些例子：

```
SELECT soundex('hello world!');

SELECT soundex(' Anne'), soundex(' Ann'), difference(' Anne', ' Ann');
SELECT soundex(' Anne'), soundex(' Andrew'), difference(' Anne', ' Andrew');
SELECT soundex(' Anne'), soundex(' Margaret'), difference(' Anne', ' Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES (' john');
INSERT INTO s VALUES (' joan');
INSERT INTO s VALUES (' wobbly');
INSERT INTO s VALUES (' jack');

SELECT * FROM s WHERE soundex(nm) = soundex(' john');

SELECT * FROM s WHERE difference(s.nm, ' john') > 2;
```

## H. 15. 2. Levenshtein

这个函数计算两个字符串之间的编辑距离。

levenshtein(text source, text target, int ins\_cost, int del\_cost, int sub\_cost)  
 返回 int  
 levenshtein(text source, text target) 返回 int  
 levenshtein\_less\_equal(text source, text target, int ins\_cost, int del\_cost, int  
 sub\_cost, int max\_d) 返回 int  
 levenshtein\_less\_equal(text source, text target, int max\_d) 返回 int

source以及target都可以是任何非空字符串， 最长为 255 个字符。代价参数分别指定一个字符插入、删除或替换的开销。 你可以像这个函数的第二种版本那样忽略代价参数，那样它们都会默认为 1。

levenshtein\_less\_equal是 Levenshtein 函数的速度更快的版本，它被用于只对小距离感兴趣的情况。如果实际距离小于等于max\_d， 那么levenshtein\_less\_equal返回正确的距离。否则它返回某个 大于max\_d的值。如果max\_d是负值，那么其行为等同于 levenshtein。

例子：

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-----
                2
(1 row)
```

```

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2,1,1);
 levenshtein
-----
              3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',2);
 levenshtein_less_equal
-----
              3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',4);
 levenshtein_less_equal
-----
              4
(1 row)

```

### H. 15. 3. Metaphone

和 Soundex 相似，Metaphone 的思想是构建一个输入字符串的一种代码。如果两个字符串具有相同的代码则认为它们相似。

这个函数计算一个输入字符串的变音位代码：

metaphone(text source, int max\_output\_length) 返回 text

source 必须是一个非空字符串，最大长度为 255 个字符。max\_output\_length 设置输出的变音位代码的最大长度，如果超长，输出会被截断到这个长度。

例子：

```

test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----
      KM
(1 row)

```

### H. 15. 4. 双 Metaphone

双变音位系统为一个给定输入字符串计算两个“听起来像的”字符串 — 一个“主要”代码和一个“次要”代码。在大部分情况下它们是相同的，但是对于非英语名称它们可能有一点不同，这取决于发音。这些函数计算主要和次要代码：

dmetaphone(text source) 返回 text  
dmetaphone\_alt(text source) 返回 text

对输入字符串没有长度限制。

例子：

```
test=# SELECT dmetaphone('gumbo');
 dmetaphone
-----
 KMP
(1 row)
```

## H. 16. hstore

这个模块实现了hstore数据类型用来在一个单一LightDB值中存储键值对。这在很多情景下都有用，例如带有很多很少被检查的属性的行或者半结构化数据。键和值都是简单的文本字符串。

这个模块被视为“trusted”，也就是说，它可以由对当前数据库具有CREATE权限的非超级用户安装。

### H. 16. 1. hstore 外部表示

一个hstore的文本表示用于输入和输出，包括零个或者多个由逗号分隔的key => value对。一些例子：

```
k => v
foo => bar, baz => whatever
"1-a" => "anything at all"
```

键值对的顺序没有意义（并且在输出时也不会重现）。键值对之间或者=>号周围的空白会被忽略。双引号内的键和值可以包括空白、逗号、=或>。要在一个键或值中包括一个双引号或一个反斜线，用一个反斜线对它转义。

一个hstore中的每一个键是唯一的。如果你声明了一个有重复键的hstore，只有一个会被存储在hstore中并且无法保证哪一个将被保留：

```
SELECT 'a=>1, a=>2'::hstore;
 hstore
-----
 "a"=>"1"
```

一个值（但不是一个键）能够是一个 SQL NULL。例如：

```
key => NULL
```

NULL关键词是大小写不敏感的。将NULL放在双引号中可以将它当作一个普通的字符串“NULL”。

#### Note

记住当hstore文本格式当被用于输入时，它应用在任何必须的引用或转义之前。如果你通过一个参数传递一个hstore文字，那么不需要额外的处理。但是如果你将它作为一个引用的文字常数，那么任何单引号字符以及（取决于standard\_conforming\_strings配置参数的设置）反斜线字符需要被正确地转义。更多关于处理字符串常量的处理可见Section 4.1.2.1

在输出时，双引号总是围绕着键和值，即使这样做不是绝对必要。



## H. 16.2. hstore 操作符和函数

hstore模块所提供的操作符显示在Table H. 5中，函数在Table H. 6中。

Table H. 5. hstore 操作符

操作符	描述 例子
hstore → text → text	返回与给定键相关联的值，如果不存在则返回NULL。 'a=>x, b=>y'::hstore → 'a' → x
hstore → text[] → text[]	返回与给定键(多个)相关联的值，如果不存在则返回NULL。 'a=>x, b=>y, c=>z'::hstore → ARRAY['c', 'a'] → {"z", "x"}
hstore    hstore → hstore	连接两个 hstore。 'a=>b, c=>d'::hstore    'c=>x, d=>q'::hstore → "a"=>"b", "c"=>"x", "d"=>"q"
hstore ? text → boolean	hstore 是否包含键? 'a=>l'::hstore ? 'a' → t
hstore ?& text[] → boolean	hstore 是否包含所有指定的键? 'a=>l, b=>2'::hstore ?& ARRAY['a', 'b'] → t
hstore ?  text[] → boolean	hstore 是否包含任何指定的键? 'a=>l, b=>2'::hstore ?  ARRAY['b', 'c'] → t
hstore @> hstore → boolean	左操作符包含右操作符吗? 'a=>b, b=>l, c=>NULL'::hstore @> 'b=>l' → t
hstore <@ hstore → boolean	左操作符包含在右操作符里面吗? 'a=>c'::hstore <@ 'a=>b, b=>l, c=>NULL' → f
hstore - text → hstore	从左操作符删除键。 'a=>l, b=>2, c=>3'::hstore - 'b'::text → "a"=>"l", "c"=>"3"
hstore - text[] → hstore	从左操作符删除键(多个)。 'a=>l, b=>2, c=>3'::hstore - ARRAY['a', 'b'] → "c"=>"3"
hstore - hstore → hstore	从左操作符中删除与右操作符中的对(pairs)相匹配的对。 'a=>l, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore → "a"=>"l", "c"=>"3"
anyelement #= hstore → anyelement	用hstore中相匹配值替换左操作符(必须是复合类型)中的字段。 ROW(1, 3) #= 'f1=>11'::hstore → (11, 3)
%% hstore → text[]	将hstore转换为交替键和值的数组。 %'a=>foo, b=>bar'::hstore → {a, foo, b, bar}
%# hstore → text[]	将hstore转换为二维的键/值数组。 %'a=>foo, b=>bar'::hstore → {{a, foo}, {b, bar}}

Note

在 LightDB 8.2 之前，包含运算符@>和<@分别被称为@和~。 这些名称仍然可用，但已被弃用，并最终将不再使用。请注意，旧名称与核心几何数据类型以前遵循的约定相反！

Table H.6. hstore 函数

函数	描述 例子
hstore ( record ) → hstore	从一个记录或行构造一个hstore。 hstore(ROW(1,2)) → "f1"=>"1", "f2"=>"2"
hstore ( text[] ) → hstore	从一个数组构造hstore，可以是键/值数组，也可以是二维数组。 hstore(ARRAY['a', '1', 'b', '2']) → "a"=>"1", "b"=>"2" hstore(ARRAY[['c', '3'], ['d', '4']]) → "c"=>"3", "d"=>"4"
hstore ( text[], text[] ) → hstore	从单独的键和值数组构造一个hstore。 hstore(ARRAY['a', 'b'], ARRAY['1', '2']) → "a"=>"1", "b"=>"2"
hstore ( text, text ) → hstore	创建一个单项目 hstore。 hstore('a', 'b') → "a"=>"b"
akeys ( hstore ) → text[]	提取一个hstore的键作为数组。 akeys('a=>1,b=>2') → {a,b}
skeys ( hstore ) → setof text	提取一个hstore的键作为一个集合。 skeys('a=>1,b=>2') →  a b
avals ( hstore ) → text[]	提取一个hstore的值为一个数组。 avals('a=>1,b=>2') → {1,2}
svals ( hstore ) → setof text	提取一个hstore的值为一个集合。 svals('a=>1,b=>2') →  1 2
hstore_to_array ( hstore ) → text[]	提取一个hstore的键和值为键和值交替数组。 hstore_to_array('a=>1,b=>2') → {a, 1, b, 2}
hstore_to_matrix ( hstore ) → text[]	提取hstore的键和值为二维数组。 hstore_to_matrix('a=>1,b=>2') → {{a, 1}, {b, 2}}
hstore_to_json ( hstore ) → json	将hstore转换为json值，将所有非空值转换为JSON字符串。

函数	描述 例子						
	<p>当hstore值被转换为json时，隐式使用此函数。</p> <pre>hstore_to_json('a key'=&gt;1, b=&gt;t, c=&gt;null, d=&gt;12345, e=&gt;012345, f=&gt;1.234, g=&gt;2.345e+4') -&gt; {"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}</pre>						
	<p>hstore_to_jsonb ( hstore ) -&gt; jsonb 将hstore转换为jsonb值，将所有非空值转换为JSON字符串。 当hstore值转换为jsonb时，隐式使用此函数。</p> <pre>hstore_to_jsonb('a key'=&gt;1, b=&gt;t, c=&gt;null, d=&gt;12345, e=&gt;012345, f=&gt;1.234, g=&gt;2.345e+4') -&gt; {"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}</pre>						
	<p>hstore_to_json_loose ( hstore ) -&gt; json 将hstore转换为json值，但尝试区分数值和布尔值，所有它们在JSON中未引用。</p> <pre>hstore_to_json_loose('a key'=&gt;1, b=&gt;t, c=&gt;null, d=&gt;12345, e=&gt;012345, f=&gt;1.234, g=&gt;2.345e+4') -&gt; {"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}</pre>						
	<p>hstore_to_jsonb_loose ( hstore ) -&gt; jsonb 将hstore转换为jsonb值，但试图区分数值和布尔值，使它们在JSON中不带引号。</p> <pre>hstore_to_jsonb_loose('a key'=&gt;1, b=&gt;t, c=&gt;null, d=&gt;12345, e=&gt;012345, f=&gt;1.234, g=&gt;2.345e+4') -&gt; {"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}</pre>						
	<p>slice ( hstore, text[] ) -&gt; hstore 提取仅包含指定键的hstore的子集。</p> <pre>slice('a=&gt;1,b=&gt;2,c=&gt;3'::hstore, ARRAY['b','c','x']) -&gt; "b"=&gt;"2", "c"=&gt;"3"</pre>						
	<p>each ( hstore ) -&gt; setof record ( key text, value text ) 提取hstore的键和值作为一组记录。</p> <pre>select * from each('a=&gt;1,b=&gt;2')</pre> <table border="1" style="margin-left: 40px;"> <thead> <tr> <th>key</th> <th>value</th> </tr> </thead> <tbody> <tr> <td>a</td> <td>1</td> </tr> <tr> <td>b</td> <td>2</td> </tr> </tbody> </table>	key	value	a	1	b	2
key	value						
a	1						
b	2						
	<p>exist ( hstore, text ) -&gt; boolean hstore 是否包含键？</p> <pre>exist('a=&gt;1', 'a') -&gt; t</pre>						
	<p>defined ( hstore, text ) -&gt; boolean hstore 是否包含针对键的非NULL 值？</p> <pre>defined('a=&gt;NULL', 'a') -&gt; f</pre>						
	<p>delete ( hstore, text ) -&gt; hstore 删除带有匹配键的对(pair)。</p> <pre>delete('a=&gt;1,b=&gt;2', 'b') -&gt; "a"=&gt;"1"</pre>						
	<p>delete ( hstore, text[] ) -&gt; hstore 删除带有匹配键的对(pairs)。</p> <pre>delete('a=&gt;1,b=&gt;2,c=&gt;3', ARRAY['a','b']) -&gt; "c"=&gt;"3"</pre>						
	<p>delete ( hstore, hstore ) -&gt; hstore 删除在第二个参数中匹配的对。</p> <pre>delete('a=&gt;1,b=&gt;2', 'a=&gt;4,b=&gt;2'::hstore) -&gt; "a"=&gt;"1"</pre>						
	<p>populate_record ( anyelement, hstore ) -&gt; anyelement 用hstore中的匹配值替换左操作符(必须是复合类型)中的字段。</p>						

函数	描述
	例子
populate_record(ROW(1, 2), 'f1=>42'::hstore) → (42, 2)	

### H. 16. 3. 索引

hstore有对@>、?、?&和?|操作符的 GiST 和 GIN 索引支持。例如：

```
CREATE INDEX hidx ON testhstore USING GIST (h);
```

```
CREATE INDEX hidx ON testhstore USING GIN (h);
```

gist\_hstore\_ops GiST 操作符类(opclass)将一组键/值对近似计算为位图签名。它的可选整数参数siglen决定了签名的字节长度。默认长度为16字节。签名长度的有效值在1到2024字节之间。更长的签名将导致更精确的搜索(扫描更小的索引部分和更少的堆页)，以更大的索引为代价。

建立这样的一个带有32字节签名长度的示例：

```
CREATE INDEX hidx ON testhstore USING GIST (h gist_hstore_ops(siglen=32));
```

hstore也为=操作符支持btree或hash索引。这允许hstore列被声明为UNIQUE或者被使用在GROUP BY、ORDER BY或DISTINCT表达式中。hstore值的排序顺序不是特别有用，但是这些索引可能对等值查找有用。为=比较创建以下索引：

```
CREATE INDEX hidx ON testhstore USING BTREE (h);
```

```
CREATE INDEX hidx ON testhstore USING HASH (h);
```

### H. 16. 4. 例子

增加一个键，或者用一个新值更新一个现有的键：

```
UPDATE tab SET h = h || hstore('c', '3');
```

删除一个键：

```
UPDATE tab SET h = delete(h, 'k1');
```

将一个record转换成一个hstore：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');
```

```
SELECT hstore(t) FROM test AS t;
           hstore
```

```
-----
"col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

将一个hstore转换成一个预定义的record类型：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
-----+-----+-----
  456 | zzz  |
(1 row)
```

用来自于一个hstore的值修改一个现有的记录:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
-----+-----+-----
  123 | foo  | baz
(1 row)
```

## H. 16. 5. 统计

由于hstore类型本质的宽大的性，它能够包含一些不同的键。检查合法键是应用的任务。下列例子验证了用于检查键以及获得统计的一些技术。

简单例子:

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

使用一个表:

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

在线统计:

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
```

key	count
line	883
query	207
pos	203
node	202
space	197
status	195
public	194
title	190
org	189
.....	

## H. 16. 6. 兼容性

从 LightDB 9.0 开始，hstore使用了与之前版本不同的内部表示。这不会为转储/恢复升级造成障碍，因为文本表示（用于转储）没有改变。

在一次二进制升级中，通过让新代码识别旧格式数据来维持向上兼容。当处理还没有被新代码修改过的数据时，这会带来一定的性能惩罚。可以通过执行一个下面的UPDATE语句来强制升级表中的所有值：

```
UPDATE tablename SET hstorecol = hstorecol || '';
```

另一种方法：

```
ALTER TABLE tablename ALTER hstorecol TYPE hstore USING hstorecol || '';
```

ALTER TABLE方法要求表上的一个ACCESS EXCLUSIVE 锁，但是不会导致表因为旧行版本而膨胀。

## H. 16. 7. 转换

### Caution

强烈建议将转换扩展安装在与hstore相同的模式中。否则，如果转换扩展的模式包含敌对用户定义的对象，就会存在安装时的安全隐患。

## H. 16. 8. 作者

Oleg Bartunov <oleg@sai.msu.su>, 俄罗斯莫斯科大学

Teodor Sigaev <teodor@sigaev.ru>, 俄罗斯德尔塔软件有限公司

额外的提升由英国的 Andrew Gierth <andrew@tao11.riddles.org.uk> 提供

## H. 17. intagg

intagg模块提供了一个整数聚集器以及一个枚举器。intagg现在已被弃用，因为有内建的函数能提供其功能的超集。不过，该模块仍然作为内建函数的一个兼容性包装器被提供。

### H. 17. 1. 函数

聚集器是一个聚集函数int\_array\_aggregate(integer)，它产生一个完全包含输入整数的整数数组。这是一个array\_agg的包装器，它对任何数组类型做同样的事情。

枚举器是一个函数int\_array\_enum(integer[])，它返回setof integer。它本质上是聚集器的一个逆操作：给定一个整数数组，将它展开成行的集合。这是unnest的一个包装器，它对任何数组类型做相同的事情。

### H. 17. 2. 使用示例

很多数据库系统都有一对多表的概念这样一个表通常位于两个被索引表之间，例如：

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

通常这样用它：

```
SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right)
WHERE one_to_many.left = item;
```

这将为左手表中的一个项返回右手表中的所有项。这在 SQL 中是很常见的结构。

现在，这种方法对于具有非常多项的one\_to\_many表会有麻烦。通常，这样的一次连接将为一个特定的左手项导致一次索引扫描以及为每一个右手项导致一次取出。如果你有一个非常动态的系统，你没什么可做的。但是，如果你的数据相对比较静态，你可以使用聚集器创建一个汇总表。

```
CREATE TABLE summary AS
SELECT left, int_array_aggregate(right) AS right
FROM one_to_many
GROUP BY left;
```

这将创建一个表，其中对每一个左项都有一行，并且有一个右项的数组。现在如果没有某种方法来使用该数组，这样做是没有任何用处的。这时数组枚举器就派上用场了，你可以

```
SELECT left, int_array_enum(right) FROM summary WHERE left = item;
```

上述使用int\_array\_enum的查询产生与以下语句相同的结果

```
SELECT left, right FROM one_to_many WHERE left = item;
```

区别是针对汇总表的查询必须只从表中得到一行，而针对one\_to\_many的直接查询必须索引扫描并且为每项都获取一行。

在一个系统上，一个EXPLAIN显示一个查询的代价从 8488 降低到了 329。原始查询是一个涉及到one\_to\_many表的连接，它被替换为：

```
SELECT right, count(right) FROM
( SELECT left, int_array_enum(right) AS right
  FROM summary JOIN (SELECT left FROM left_table WHERE left = item) AS lefts
    ON (summary.left = lefts.left)
) AS list
GROUP BY right
ORDER BY count DESC;
```

## H. 18. intarray

intarray模块提供了一些有用的函数和操作符来操纵不含空值的整数数组。也提供了对使用某些操作符的索引搜索的支持。

如果一个提供的数组中包含任何 NULL 元素，所有这些操作都将抛出一个错误。

很多这些操作只对一维数组有意义。尽管它们将接受更多维数的数组输入，数据将被当作一个按照存储顺序排列的线性数组对待。

该模块被认为是“trusted”，也就是说，它可以由对当前数据库具有CREATE权限的非超级用户安装。

### H. 18.1. intarray 函数和操作符

intarray模块提供的函数被列在Table H. 中，操作符被列在Table H. 中。

Table H.7. intarray 函数

函数	描述 例子
icount ( integer[] ) → integer	返回数组中元素的数量 icount( ' {1, 2, 3}' ::integer[] ) → 3
sort ( integer[], dir text ) → integer[]	按升序或降序对数组进行排序。 dir必须是asc或desc。 sort( ' {1, 3, 2}' ::integer[], 'desc' ) → {3, 2, 1}
sort ( integer[] ) → integer[] sort_asc ( integer[] ) → integer[]	以升序排序 sort(array[11, 77, 44]) → {11, 44, 77}
sort_desc ( integer[] ) → integer[]	以降序排序 sort_desc(array[11, 77, 44]) → {77, 44, 11}
uniq ( integer[] ) → integer[]	移除相邻的重复项。 通常与sort一起使用以移除所有重复项。 uniq( ' {1, 2, 2, 3, 1, 1}' ::integer[] ) → {1, 2, 3, 1} uniq(sort( ' {1, 2, 3, 2, 1}' ::integer[] )) → {1, 2, 3}
idx ( integer[], item integer ) → integer	返回匹配item的第一个元素的索引 (如果没有为 0) idx(array[11, 22, 33, 22, 11], 22) → 2
subarray ( integer[], start integer, len integer ) → integer[]	提取从位置start开始的由len个元素组成的元组部分 subarray( ' {1, 2, 3, 2, 1}' ::integer[], 2, 3) → {2, 3, 2}
subarray ( integer[], start integer ) → integer[]	提取从位置start开始的元组部分 subarray( ' {1, 2, 3, 2, 1}' ::integer[], 2) → {2, 3, 2, 1}
intset ( integer ) → integer[]	创建单一元素数组 intset(42) → {42}

Table H.8. intarray 操作符

操作符	描述
integer[] && integer[] → boolean	数组是否重叠 (至少有一个共同元素) ?
integer[] @> integer[] → boolean	左数组是否包含右数组?
integer[] <@ integer[] → boolean	左数组是否包含在右数组中?
# integer[] → integer	返回数组中元素的数目
integer[] # integer → integer	返回与正确参数匹配的的第一个数组元素的索引, 如果不匹配则返回 0。 (与idx函数相同。)
integer[] + integer → integer[]	将元素添加到数组末尾。



操作符	描述
<code>integer[] + integer[]</code>	<code>→ integer[]</code> 连接数组。
<code>integer[] - integer</code>	<code>→ integer[]</code> 从数组中移除匹配右参数的项
<code>integer[] - integer[]</code>	<code>→ integer[]</code> 从左数组中移除右数组的元素
<code>integer[]   integer</code>	<code>→ integer[]</code> 计算参数的并集
<code>integer[]   integer[]</code>	<code>→ integer[]</code> 计算参数的并集
<code>integer[] &amp; integer[]</code>	<code>→ integer[]</code> 计算参数的交集。
<code>integer[] @@ query_int</code>	<code>→ boolean</code> 数组是否满足查询？（见下文）
<code>query_int ^^ integer[]</code>	<code>→ boolean</code> 数组是否满足查询？（@@的交换子）

（在 LightDB 8.2 之前，包含运算符@>和<@分别被称为@和~。 这些名称仍然可用，但已被弃用，并最终将不再使用。请注意，旧名称与核心几何数据类型以前遵循的约定相反！）

操作符&&、@>和<@等效于LightDB的内建同名操作符，不过它们只能在不含空值的整数数组上工作，而内建的操作符可以对任何数组类型工作。这种限制使它们在很多情况下比内建操作符更快。

@@和^^操作符测试一个数组是否满足一个query，它被表示成一种特殊数据类型query\_int的一个值。一个由整数值组成的查询会被针对数组的元素检查，可能会组合使用操作符&（AND）、|（OR）以及!（NOT）。根据需要可以使用圆括号。例如，查询1&(2|3)匹配包含 1 并且还包含 2 或 3 的数组。

## H. 18. 2. 索引支持

intarray提供对于&&、@>和@@操作符以及常规数组相等的索引支持。

提供了两种参数化的 GiST 索引操作符类：`gist__int_ops`（被默认使用）适合于中小尺寸的数据集，而`gist__intbig_ops`使用一种更大的签名并且更适合于索引大型数据集（即，包含大量可区分数组值的列）。该实现使用了一种带有内建有损压缩的 RD 树结构。

`gist__int_ops`将整数集近似为整数范围数组。 它的可选整数参数`numranges`决定了一个索引键中的最大范围数。 `numranges`的默认值是 100。有效值在 1 到 253 之间。 使用更大的数组作为 GiST 索引键会导致更精确的搜索（扫描更小的索引部分和更少的堆页面），以更大的索引为代价。

`gist__intbig_ops`将整数集近似为位图签名。 它的可选整数参数`siglen`确定签名长度（以字节为单位）。 默认签名长度为 16 字节。签名长度的有效值介于 1 到 2024 字节之间。 更长的签名导致更精确的搜索（扫描索引的一小部分和更少的堆页面），但代价是更大的索引。

也有一种非默认的 GIN 操作符类`gin__int_ops`支持这些操作符还有 <@。

在 GiST 和 GIN 索引之间的选择取决于 GiST 和 GIN 的相对性能特点， 这将在其他地方讨论。

## H. 18. 3. 例子

```

-- 一个消息可以在一个或多个“小节”中
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- 创建签名长度为 32 字节的专用索引
CREATE INDEX message_rdtree_idx ON message USING GIST (sections
gist__intbig_ops(siglen = 32));

-- 选择小节 1 或 2 中的消息 - OVERLAP 操作符
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- 选择小节 1 和 2 中的消息 - CONTAINS 操作符
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- 相同, 使用 QUERY 操作符
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;

```

## H. 18. 4. 测试基准

源代码目录contrib/intarray/bench包含有一个基准测试套件，可以针对一个安装好的LightDB服务器运行这个套件（还要求安装DBD::Pg）。要运行测试基准：

```

cd .../contrib/intarray/bench
createdb TEST
ltsql -c "CREATE EXTENSION intarray" TEST
./create_test.pl | ltsql TEST
./bench.pl

```

bench.pl脚本有多个选项，当它不使用任何参数运行时会显示这些选项。

## H. 18. 5. 作者

所有工作由 Teodor Sigaev (<teodor@sigae.ru>) 和 Oleg Bartunov (<oleg@sai.msu.su>) 完成。更多信息请见<http://www.sai.msu.su/~megera/postgres/gist/>。Andrey Oktyabrski 完成了一项伟大的工作，他增加了新的函数和操作。

## H. 19. isn

isn模块为下列国际产品编号标准提供数据类型：EAN13、UPC、ISBN（图书）、ISMN（音乐）以及 ISSN（期刊）。在输入时会按照一个硬编码的前缀列表对输入进行验证，这个前缀的列表也被用来在输出时连接号码。因为新的前缀总是不时地出现，这个前缀列表可能会过时。这个模块的一个未来版本有望得到一个来自于一个或多个表的前缀列表，这样用户可以根据需要来方便地更新前缀列表。不过，在当前该列表只能通过修改源代码并且重新编译来更新。另外一种方案是，在这个模块的未来版本中可能会直接移除掉前缀验证和连接支持。

该模块被认为是“trusted”，也就是说，它可以由对当前数据库具有CREATE权限的非超级用户安装。

### H. 19. 1. 数据类型

Table H. 展示了isn模块提供的数据类型。

Table H. 9. isn 数据类型

数据类型	描述
EAN13	欧洲文章号，总是以 EAN13 格式显示

数据类型	描述
ISBN13	国际标准图书号，以新的 EAN13 格式显示
ISMN13	国际标准音乐号，以新的 EAN13 格式显示
ISSN13	国际标准期刊号，以新的 EAN13 格式显示
ISBN	国际标准图书号，以旧的短格式显示
ISMN	国际标准音乐号，以旧的短格式显示
ISSN	国际标准期刊号，以旧的短格式显示
UPC	通用产品代码

一些注记：

1. ISBN13、ISMN13、ISSN13 号码都是 EAN13 号码。
2. EAN13 号码不总是 ISBN13、ISMN13 或 ISSN13（有些是）。
3. 一些 ISBN13 号码能够作为 ISBN 显示。
4. 一些 ISMN13 号码能够作为 ISMN 显示。
5. 一些 ISSN13 号码能够作为 ISSN 显示。
6. UPC 号码是 EAN13 号码的一个子集（它们基本上是去掉了第一个0位的 EAN13）。
7. 所有 UPC、ISBN、ISMN 以及 ISSN 号码可以被表示为 EAN13 号码。

在内部，所有这些类型使用同一种表达（一个 64 位整数）并且所有内部表达是可以互换的。多种类型被提供来控制显示格式化并且对假定为表示一种特定类型号码的输入进行更严格的合法性检查。

在可能时，ISBN、ISMN和ISSN类型将显示号码的短版本（ISxN 10），并且在无法适应短版本时显示号码的 ISxN 13 格式。EAN13、ISBN13、ISMN13和ISSN13类型总是显示长版本的 ISxN（EAN13）。

## H. 19. 2. 造型

isn模块提供了下列类型之间的造型：

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

当从EAN13造型为另一种类型时，会有对该值是否在另一种类型的域中的运行时检查，如果不在则抛出一个错误。其他的造型则是简单地重新贴个标签，因而总是会成功。

## H. 19.3. 函数和操作符

isbn模块提供了标准的比较操作符，外加对所有这些数据类型的 B 树和哈希索引支持。此外还有一些特殊的函数，它们展示在Table H.10中。在这个表中，isbn意味着该模块的数据类型中的任何一种。

Table H.10. isbn 函数

函数	描述
isbn_weak ( boolean ) → boolean	设置弱输入模式 (返回新设置)
isbn_weak () → boolean	返回弱模式的当前状态
make_valid ( isbn ) → isbn	验证一个非法号码 (清除非法标志)
is_valid ( isbn ) → boolean	检查非法标志的存在

弱模式被用来允许插入非法数据到一个表中。非法意味着校验位错误，而不是有丢失号码。

为什么你会想要使用弱模式？你可能有一个巨大的 ISBN 号码集合并且出于某种奇怪的原因其中具有错误的校验位（可能这些号码是从印刷稿中扫描并且 OCR 而来，也可能是手工输入的..... 谁知道呢）。不管怎样，重点是你可能希望清理这些混乱，但是你仍然想要能够把这些号码放在你的数据库中并且可能会使用一个外部工具在数据库中定位非法号码，这样你能够更容易地验证信息。因此你可能会想要在表中选择所有非法的号码。

当你使用弱模式在一个表中插入非法号码时，被插入的号码将会被加上修正过的校验位，但是它的最后将会有有一个感叹号 (!)，例如0-11-000322-5!。这种非法标志符可以用is\_valid函数检查并且可以用make\_valid函数清除。

即使不在弱模式中，你也能通过在号码某位追加!字符来强制非法号码的插入。

另一个特殊特性是在输入过程中，你可以写一个?代替校验位，然后正确的校验位将被自动插入。

## H. 19.4. 例子

--直接使用类型:

```
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');
```

--转换类型:

```
-- 注意只有在号码处于另一种类型的合法值之中时，才能从 EAN13 转换成另一种类型
-- 因此下面的用法将不会工作: select isbn(ean13('0220356483481'));
-- 但是下面的可以:
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));
```

--创建一个表，它有一个单一列来保存 ISBN 号码:

```
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');
```

--自动计算校验位 (观察 '?' ):

```
INSERT INTO test VALUES('220500896?');
```

```

INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');

--使用弱模式:
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;

SELECT isbn13(id) FROM test;

```

## H. 19.5. 参考文献

实现这个模块的信息可以从下列网站收集到:

- <https://www.isbn-international.org/>
- <https://www.issn.org/>
- <https://www.ismn-international.org/>
- <https://www.wikipedia.org/>

用于连接的前缀:

- <https://www.gsl.org/standards/id-keys>
- [https://en.wikipedia.org/wiki/List\\_of\\_ISBN\\_identifier\\_groups](https://en.wikipedia.org/wiki/List_of_ISBN_identifier_groups)
- <https://www.isbn-international.org/content/isbn-users-manual>
- [https://en.wikipedia.org/wiki/International\\_Standard\\_Music\\_Number](https://en.wikipedia.org/wiki/International_Standard_Music_Number)
- <https://www.ismn-international.org/ranges.html>

创建算法时已经注意严谨地使用 ISBN、ISMN、ISSN 官方用户手册中的推荐算法进行了验证。

## H. 19.6. 作者

Germán Méndez Bravo (Kronuz), 2004 - 2006

这个模块受到了 Garrett A. Wollman 的 isbn\_issn 代码的启发。

## H. 20. lo

lo 模块提供管理大对象（也被称为 L0 或 BLOB）的支持。这包括一种数据类型 lo 以及一个触发器 lo\_manage。

这个模块被视为“trusted”，也就是说，它能够由当前数据库中具有 CREATE 特权的非超级用户来安装。

## H. 20. 1. 原理

JDBC 驱动的问题之一（并且这也影响 ODBC 驱动）是其说明书假定对 BLOB（二进制大对象）的引用被存储在一个表中，并且如果该项被改变相关的 BLOB 会被从数据库删除。

但对于LightDB来说这并不会发生。大对象被当做自主的对象，一个表项可以通过 OID 引用一个大对象，但是可以有多个表项引用同一个大对象 OID，因此系统不会因为你改变或者删除这种项而删除大对象。

现在这对LightDB-相关的应用挺好的，但是使用 JDBC 或 ODBC 的标准代码不会删除那些对象，从而导致孤立对象 — 不被任何东西引用的对象，而且会占据磁盘空间。

lo允许通过附加一个触发器到包含 LO 引用列的表来修复这种问题。该触发器本质上只是在你删除或修改一个引用大对象的值时做lo\_unlink。当你使用这个触发器时，你必须假定在一个触发器控制的列中只有一个对任意大对象的数据库引用！

这个模块也提供了一种数据类型lo，它实际上只是oid类型的一个域。这有助于区分保存大对象引用的数据库列和保存其他东西 OID 的列。你并不是必须使用lo类型来使用该触发器，但是用它来追踪数据库中哪些列表示你要用触发器管理的大对象非常方便。也有传言说如果你不为 BLOB 列使用lo，ODBC 驱动会感到困惑。

## H. 20. 2. 如何使用它

这里是一个简单的用法示例：

```
CREATE TABLE image (title text, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
    FOR EACH ROW EXECUTE FUNCTION lo_manage(raster);
```

对每一个将包含到大对象唯一引用的列，创建一个BEFORE UPDATE OR DELETE触发器，并且将该列名作为唯一的触发器参数。你也可以用BEFORE UPDATE OF column\_name来限制该触发器只对该列上的更新事件执行。如果你需要在同一个表中有多个lo列，为每一个创建一个独立的触发器，记住为同一个表上的每个触发器指定一个不同的名称。

## H. 20. 3. 限制

- 删除一个表仍将让它包含的任何对象变成孤立的，因为触发器在这种情况下不会被执行。你可以在DROP TABLE之前放上DELETE FROM table来避免这种问题。

TRUNCATE有同样的危害。

如果你已经有或者怀疑有孤立的大对象，参考vacuumlo模块可以帮助你清理它们。偶尔运行vacuumlo作为lo\_manage触发器的后备是个好主意。

- 有些前端可能会创建它们自己的表，并且将不会创建相关的触发器。另外，用户可能不会记得（或知道）要创建触发器。

## H. 20. 4. 作者

Peter Mount <peter@retep.org.uk>

## H. 21. 1t\_buffercache

1t\_buffercache 模块提供了实时查看共享缓存区中发生的情况的方法。

该模块提供了一个 C 函数 `pg_buffercache_pages`，它返回一组记录，以及一个视图 `pg_buffercache`，该视图将函数包装为方便使用。

默认情况下，仅超级用户和 `pg_monitor` 角色的成员可以使用。可以使用 `GRANT` 授予其他用户访问权限。

## H. 21. 1. `pg_buffercache` 视图

该视图公开的列的定义在 Table H. 11中显示。

Table H. 11. `pg_buffercache` 列

列类型	描述
<code>bufferid integer</code>	ID, 范围为 <code>1..shared_buffers</code>
<code>relfilenode oid</code> (引用自 <code>pg_class .relfilenode</code> )	关系的文件节点号
<code>reltablespace oid</code> (引用自 <code>pg_tablespace .oid</code> )	关系的表空间 OID
<code>reldatabase oid</code> (引用自 <code>pg_database .oid</code> )	关系的数据库 OID
<code>relforknumber smallint</code>	关系内部的 Fork 编号; 请参见 <code>include/common/relpath.h</code>
<code>relblocknumber bigint</code>	关系中的页面编号
<code>isdirty boolean</code>	页面是否是脏页?
<code>usagecount smallint</code>	时钟算法扫描访问计数
<code>pinning_backends integer</code>	固定此缓存区的后端数量

共享缓存中每个缓存区都有一行记录。未使用的缓存区所有字段均为空，仅 `bufferid` 不为空。共享系统目录显示为属于数据库零。

由于缓存是由所有数据库共享的，因此通常会有来自不属于当前数据库的表的页面。这意味着有些行可能在 `pg_class` 中没有匹配的连接行，或者甚至可能存在错误的连接。如果您尝试与 `pg_class` 进行连接，则最好将连接限制为具有 `reldatabase` 等于当前数据库 OID 或零的行。

由于缓存管理器锁定不会被用于复制视图将显示的缓存状态数据，因此访问 `pg_buffercache` 视图对正常的缓存活动影响较小，但它不能在所有缓存区域提供一致的结果集。但是，我们确保每个缓存区域的信息是自洽的。

## H. 21. 2. 示例输出

```
regression=# SELECT n.nspname, c.relname, count(*) AS buffers
FROM pg_buffercache b JOIN pg_class c
ON b.relfilenode = pg_relation_filenode(c.oid) AND
b.reldatabase IN (0, (SELECT oid FROM pg_database
WHERE datname = current_database()))
JOIN pg_namespace n ON n.oid = c.relnamespace
GROUP BY n.nspname, c.relname
```

```
ORDER BY 3 DESC
LIMIT 10;

nspname | relname | buffers
-----+-----+-----
public | delete_test_table | 593
public | delete_test_table_pkey | 494
pg_catalog | pg_attribute | 472
public | quad_poly_tbl | 353
public | tenk2 | 349
public | tenk1 | 349
public | gin_test_idx | 306
pg_catalog | pg_largeobject | 206
public | gin_test_tbl | 188
public | spgist_text_tbl | 182
(10 rows)
```

### H. 21. 3. 作者

时间戳: {int(time.time())}

设计建议: Neil Conway <neilc@samurai.com>

调试建议: Tom Lane <tgl@sss.pgh.pa.us>

## H. 22. lt\_bulkload (批量加载)

lt\_bulkload (批量加载) 是 LightDB 的高速数据加载工具。

lt\_bulkload (批量加载) 旨在将大量的数据加载到数据库中。您可以通过绕过 LightDB 的共享缓冲区将数据加载到表中。

lt\_bulkload (批量加载) 还具有一些 ETL (Extract-Transform-Load) 功能; 输入数据验证和数据转换。

### H. 22. 1. 概述

lt\_bulkload (批量加载) [ OPTIONS ] [ controlfile (控制文件) ]

### H. 22. 2. 描述

lt\_bulkload (批量加载) 旨在将大量的数据加载到数据库中。您可以选择是否检查数据库约束以及在加载过程中忽略多少个错误。例如, 当您从另一个数据库复制数据到 LightDB 时, 可以跳过完整性检查以提高性能。另一方面, 当加载不干净的数据时, 可以启用约束检查。

lt\_bulkload (批量加载) 可以将加载数据转换为二进制文件, 该文件可以用作 lt\_bulkload 的输入文件。如果在将加载数据转换为二进制文件时检查其是否有效, 则可以在从二进制文件加载数据时跳过检查。

### H. 22. 3. 用法

您可以通过以下三个步骤使用 lt\_bulkload (批量加载):

1. 编辑控制文件 "sample\_csv.ctl" 或 "sample\_bin.ctl", 其中包含数据加载的设置。您可以指定表名、输入文件的绝对路径、输入文件的描述等。
2. 假设存在一个名为 \$LTDATA/lt\_bulkload 的目录, 在其中创建加载状态文件。



3. 使用控制文件作为参数执行命令。该参数可使用相对路径。

```
$ lt_bulkload sample_csv.ctl
NOTICE: BULK LOAD START
NOTICE: BULK LOAD END
0 Rows skipped.
8 Rows successfully loaded.
0 Rows not loaded due to parse errors.
0 Rows not loaded due to duplicate errors.
0 Rows replaced with new rows.
```

## H. 22. 4. 选项

lt\_bulkload (批量加载) 具有以下命令行选项:

### H. 22. 4. 1. 加载选项

```
-i INPUT
--input=INPUT
--infile=INPUT
数据源路径。与控制文件中的“INPUT”相同。
-O OUTPUT
--output=OUTPUT
数据存储路径。与控制文件中的“OUTPUT”相同。
-l LOGFILE
--logfile=LOGFILE
写入结果日志的路径。与控制文件中的“LOGFILE”相同。
-P PARSE_BADFILE
--parse-badfile=PARSE_BADFILE
写入无法正确解析的异常记录的路径。与控制文件中的“PARSE_BADFILE”
相同。
-u DUPLICATE_BADFILE
--duplicate-badfile=DUPLICATE_BADFILE
写入因在索引重建过程中与唯一约束冲突而被视为异常的记录的路径。与
控制文件中的“DUPLICATE_BADFILE”相同。
-o "key=val"
--option="key=val"
控制文件中可用的任何选项。可以传递多个选项。
```

### H. 22. 4. 2. 连接选项

连接服务器的选项。

```
-d dbname
--dbname dbname
指定要连接的数据库的名称。如果未指定，则从环境变量 LTDATABASE 中
读取数据库名称。如果未设置，则使用连接指定的用户名。
-h host
--host host
指定运行服务器的计算机的主机名。如果值以斜杠开头，则将其用作
Unix 域套接字的目录。
-p port
--port port
```

名。指定服务器用于侦听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。

`-U username`  
`--username username`  
 要连接的用户名。

`-W`  
`--password`  
 强制 `lt_bulkload` 提示输入密码后再连接到数据库。  
 该选项从不是必需的，因为如果服务器要求密码身份验证，`lt_bulkload` 会自动提示输入密码。但是，`vacuumdb` 将浪费一个连接尝试以找出服务器需要密码。在某些情况下，为了避免额外的连接尝试，输入 `-W` 是值得的。

### H. 22. 4. 3. 常规选项

`-e`  
`--echo`  
 回显发送到服务器的命令。

`-E`  
`--elevel`  
 选择从 DEBUG、INFO、NOTICE、WARNING、ERROR、LOG、FATAL 和 PANIC 中输出的消息级别。默认为 INFO。

`--help`  
 显示程序的用法。

`--version`  
 显示程序的版本号。

## H. 22. 5. 控制文件

您可以指定以下加载选项。控制文件可以使用绝对路径或相对路径指定。如果您使用相对路径指定它，则相对于执行 `lt_bulkload` 命令的当前工作目录。如果您没有指定控制文件，则应通过 `lt_bulkload` 的命令行参数传递所需的选项。

控制文件中可用以下参数。每行 “#” 后面的字符将被视为注释并被忽略。

### H. 22. 5. 1. 通用

`TYPE = CSV | BINARY | FIXED | FUNCTION`

输入数据的类型。默认为 CSV。

- CSV: 从 CSV 格式的文本文件加载。
- BINARY | FIXED: 从固定的二进制文件加载。
- FUNCTION: 从函数返回的结果集加载。

如果使用，则 “INPUT” 必须是调用函数的表达式。

`INPUT | INFILE = path | stdin | [ schemaname. ] function_name (argvalue, ...)`

要加载数据的源。始终需要。根据 TYPE 选项，该值的处理方式如下：

- 服务器上的文件：这是服务器中的文件路径。如果它是相对路径，则在控制文件中指定时，它将相对于控制文件，或者在命令行参数中指定时，它将相对于当前工作目录。LightDB 服务器的用户必须具有对该文件的读取权限。仅在 “TYPE=CSV” 或 “TYPE=BINARY” 时可用。

- `lt_bulkload` 命令的标准输入: “`INPUT=stdin`” 表示 `lt_bulkload` 将通过网络从 `lt_bulkload` 客户端程序的标准输入读取数据。当输入文件和数据库位于不同的服务器上时, 应使用此形式。仅在 “`TYPE=CSV`” 或 “`TYPE=BINARY`” 时可用。
- SQL 函数: 指定一个带有参数并返回输入数据集的 SQL 函数。仅在 “`TYPE=FUNCTION`” 时可用。您不仅可以使用内置函数, 还可以使用任何用户定义的函数。定义的函数应返回 `SETOF sometype` 类型或 `RETURNS TABLE(columns)` 类型。如果使用 `sometype`, 则应指定属性类型。因为有一种情况, 即 `lt_bulkload` 内部使用的赋值强制转换为 `cstring` 类型, 不被允许。请注意, 您可能需要使用 C 语言开发这些函数, 而不是使用 PL/pgSQL, 因为函数必须使用 `SFRM_ValuePerCall` 模式进行流式加载。

下面是用户定义函数的示例。

```

                                $ CREATE TYPE sample_type AS (sum integer, name
char(10));
                                $ CREATE FUNCTION sample_function() RETURNS SETOF
sample_type
                                AS $$ SELECT id1 + id2, upper(name) FROM
INPUT_TABLE $$
                                LANGUAGE SQL;

```

下面是控制文件的示例。

```

TABLE = sample_table
TYPE = FUNCTION
WRITER = DIRECT
INPUT = sample_function() # 如果使用自定义函数
#INPUT = generate_series(1, 1000) # 如果使用内置函数,
它将生成从 1 到 1000 的序号

```

`WRITER | LOADER = DIRECT | BUFFERED | BINARY | PARALLEL`

加载数据的方法。默认为 `DIRECT`。

- `DIRECT`: 直接将数据加载到表中。绕过共享缓冲区并跳过 WAL 记录, 但需要自己的恢复过程。这是默认值, 也是旧版本的模式。
- `BUFFERED`: 通过共享缓冲区将数据加载到表中。使用共享缓冲区, 写入 WAL, 并使用原始的 LightDB WAL 恢复。
- `BINARY`: 将数据转换为二进制文件, 该文件可用作输入文件以加载。创建加载输出二进制文件所需的控制文件示例。此示例文件将在与二进制文件相同的目录中创建。
- `PARALLEL`: 与 “`WRITER=DIRECT`” 和 “`MULTI_PROCESS=YES`” 相同。如果指定了 `PARALLEL`, 则忽略 `MULTI_PROCESS`。如果配置了密码身份验证以加载数据库, 则必须设置密码文件。有关详细信息, 请参见限制。

`OUTPUT | TABLE = { [ schema_name. ] table_name | outfile }`

要加载数据的目标。始终需要。根据 `WRITER` (或 `LOADER`) 选项, 该值的处理方式如下:

- 要加载到的表: 指定要加载到的表。如果省略了 `schema_name`, 则使用搜索路径中的第一个匹配表。仅当 `WRITER` 为 `DIRECT`、`BUFFERED` 或 `PARALLEL` 时, 才能将数据加载到表中。
- 服务器上的文件: 指定服务器上输出文件的路径。如果是相对路径, 则将以与 `INPUT` 选项相同的方式解释它。运行 LightDB 的 OS 用户必须对指定文件的父目录具有写权限。仅当 `WRITER` 为 `BINARY` 时, 才能将数据加载 (转换) 到文件中。

SKIP | OFFSET = n

要跳过的输入行数。默认为 0。您不能同时指定 "TYPE=FUNCTION" 和 SKIP。

LIMIT | LOAD = n

要加载的行数。默认为 INFINITE，即所有数据都将加载。即使使用 TYPE=FUNCTION，也可以使用此选项。

ENCODING = encoding

指定输入数据的编码。检查指定的编码是否有效，并根据需要将输入数据转换为数据库编码。默认情况下，不验证输入数据的编码，也不进行转换。如果您可以确保输入数据以数据库编码进行编码，则可以通过不指定此选项并跳过编码验证和转换来减少加载时间。请注意，仅当 INPUT 为 stdin 时，默认情况下才使用 client\_encoding 作为输入数据的编码。您不能同时指定 "TYPE=FUNCTION" 和 ENCODING。

下面是选项值和实际行为：

		DB encoding
	non-SQL_ASCII	SQL_ASCII
	not specified	neither checked nor
converted	neither checked nor converted	
	SQL_ASCII	neither checked nor
converted	only checked	
	non-SQL_ASCII,	same as DB only
checked	only checked	
	non-SQL_ASCII, different from DB	only checked
	checked and converted	

FILTER = [ schema\_name. ] function\_name [ ( argtype, ... ) ]

指定过滤函数以转换输入文件中的每一行。只要函数名在数据库中是唯一的，就可以省略 argtype 的定义。如果未指定，则输入数据将直接解析为加载目标表。另请参阅如何编写过滤器函数以创建过滤器函数。

您不能同时指定 "TYPE=FUNCTION" 和 FILTER。另外，在 CSV 选项中的 FORCE\_NOT\_NULL 不能与 FILTER 选项一起使用。

CHECK\_CONSTRAINTS = YES | NO

指定在加载过程中是否检查 CHECK 约束。默认为 NO。您不能同时指定 "WRITER=BINARY" 和 CHECK\_CONSTRAINTS。

PARSE\_ERRORS = n

在解析、编码检查、编码转换、FILTER 函数、CHECK 约束检查、NOT NULL 检查或数据类型转换期间引发错误的被忽略的元组数。无效的输入元组不会被加载，并记录在 PARSE BADFILE 中。默认值为 0。如果有相等或更多的解析错误，则已加载的数据将被提交，而其余的元组将不会被加载。0 表示不允许出现错误，-1 和 INFINITE 表示忽略所有错误。

DUPLICATE\_ERRORS = n

违反唯一约束的被忽略的元组数。冲突的元组将从表中删除，并记录在 DUPLICATE BADFILE 中。默认值为 0。如果有相等或更多的唯一违规行为，则整个加载将被回滚。0 表示不允许任何违规行为，-1 和 INFINITE 表示忽略所有违规行为。您不能同时指定 "WRITER=BINARY" 和 DUPLICATE\_ERRORS。

ON\_DUPLICATE\_KEEP = NEW | OLD

指定如何处理违反唯一约束的元组。删除的元组将记录在 BAD 文件中。默认为 NEW。如果启用此选项，还需要将 DUPLICATE\_ERRORS 设置为大于 0。您不能同时指定 "WRITER=BINARY" 和 ON\_DUPLICATE\_KEEP。

- NEW: 保留输入数据中的元组，并删除相应的现有元组。当数据中都有违反的元组时，保留后者。
- OLD: 保留现有元组并删除输入数据中的元组。

LOGFILE = path

写入结果日志的路径。如果指定了相对路径，则将其视为与 INPUT 相同。默认为 \$LTDATA/lt\_bulkload/timestamp\_dbname\_schema\_table.log。

PARSE\_BADFILE = 路径

BAD文件的路径，用于记录在解析、编码检查、编码转换、FILTER函数、CHECK约束检查、NOT NULL检查或数据类型转换过程中出现错误的无效记录。文件格式与输入源文件相同。如果指定相对路径，则被视为与输入文件相同。默认路径为 \$LTDATA/lt\_bulkload/timestamp\_dbname\_schema\_table.bad.extension-of-infile。

DUPLICATE\_BADFILE = 路径

用于在索引重建期间记录与唯一约束冲突的错误记录的文件路径。文件格式始终为CSV。如果指定相对路径，则被视为与输入文件相同。默认路径为 \$LTDATA/lt\_bulkload/timestamp\_dbname\_schema\_table.dup.csv。不应同时指定"WRITER=BINARY"和 DUPLICATE\_BADFILE。

TRUNCATE = YES | NO

如果为YES，则使用TRUNCATE命令从目标表中删除所有行。如果为NO，则不执行任何操作。默认值为NO。不应同时指定"WRITER=BINARY"和TRUNCATE。

VERBOSE = YES | NO

如果为YES，则在服务器日志中记录错误的元组。如果为NO，则不在服务器日志中记录。默认为NO。

MULTI\_PROCESS = YES | NO

如果为YES，则使用多个线程并行进行数据读取、解析和写入。如果为NO，则仅使用单个线程处理它们，而不进行并行处理。默认为NO。如果WRITER为PARALLEL，则忽略MULTI\_PROCESS。如果为加载数据库配置了密码身份验证，则必须设置密码文件。有关详细信息，请参见限制。启用MULTI\_PROCESS时，请确保没有其他LightDB后端进程尝试修改表的模式，因为这可能导致读取和写入进程所看到的表模式不同并导致问题。

## H. 22. 5. 2. CSV输入格式

DELIMITER = 分隔符字符

文件中每行内（行）分隔列的单个ASCII字符。默认为逗号。当您加载一个以制表符分隔的格式文件（TSV）时，可以将DELIMITER设置为制表符字符。然后，您需要将制表符用双引号括起来：

```
DELIMITER=" " # 一个双引号括起来的制表符
```

您还可以使用\$'\t'语法将DELIMITER指定为命令行-o选项。

```
$ lt_bulkload tsv.ct1 -o $'DELIMITER=\t'
```

QUOTE = 引号字符

指定ASCII引号字符。默认为双引号。

ESCAPE = 转义字符

指定应出现在引号数据字符值之前的ASCII字符。默认为双引号。

NULL = 空字符串

表示空值的字符串。默认为没有引号的空值。

FORCE\_NOT\_NULL = 列名

将每个指定的列视为非NULL值进行处理。可以根据需要使用多个列。不能与FILTER选项一起使用。

### H. 22. 5. 3. 二进制输入格式

COL = 类型 [ (大小) ] [ NULLIF { '空字符串' | 空值十六进制 } ]

从左到右的输入文件的列定义。定义包括类型名称、偏移量和字节长度。CHAR和VARCHAR表示输入数据是文本。否则，它是二进制数据。如果是二进制的，则必须在服务器和数据文件之间匹配字节序。

- CHAR | CHARACTER: 一个去除尾部空格的字符串。长度始终是必需的。
- VARCHAR | CHARACTER VARYING: 一个保留尾部空格的字符串。长度始终是必需的。
- SMALLINT | SHOPT: 2字节的带符号整数。
- INTEGER | INT: 2、4或8字节的带符号整数。默认值为4。
- BIGINT | LONG: 8字节的带符号整数。
- UNSIGNED SMALLINT | SHORT: 2字节的无符号整数。
- UNSIGNED INTEGER | INT: 2或4字节的无符号整数。默认值为4。
- FLOAT | REAL: 4或8字节的浮点数。默认值为4。
- DOUBLE: 8字节的浮点数。

可以如下指定类型的长度和偏移量:

- TYPE: 带有默认长度的TYPE。
- TYPE(L): 长度为L字节的TYPE。
- TYPE(S+L): 长度为L字节, 偏移量为S字节, 从行的开头计算。
- TYPE(S:E): 从S字节开始, 到E字节结束。

可以如下指定类型的长度和偏移量:

- NULLIF '空字符串': 当类型为CHAR或VARCHAR时, 指定表示NULL的字符串。字符串的长度必须与类型的长度相同。
- NULLIF 空值十六进制: 当类型不是CHAR和VARCHAR时, 指定表示NULL的十六进制值。十六进制值的长度必须与类型的长度相同。

另外, "COL N"也可用, 与COL CHAR(N)相同, 用于向后兼容性。

PRESERVE\_BLANKS = YES | NO

YES将“COL N”视为“COL CHAR(N)”，而NO将其视为“COL VARCHAR(N)”。默认为NO。

STRIDE = n

一行的长度。如果要截断行的末尾，请使用它。默认为整行，即所有COL的总和。

## H. 22. 5. 4. 二进制输出格式

OUT\_COL = 类型 [ (大小) ] [ NULLIF { '空字符串' | 空值十六进制 } ]

从左到右的输出文件的列定义。定义包括类型名称、偏移量和字节长度。CHAR和VARCHAR表示输入数据是文本。否则，它是二进制数据。如果是二进制的，则必须在服务器和数据文件之间匹配字节序。

- CHAR | CHARACTER: 固定长度的字符串。必须指定长度。如果要存储的字符串比声明的长度短，则值将填充空格。“COL=CHAR(size)”将在控制文件的示例中输出。
- VARCHAR | CHARACTER VARYING: 固定长度的字符串。必须指定长度。如果要存储的字符串比声明的长度短，则值将填充空格。“COL=VARCHAR(size)”将在控制文件的示例中输出。
- SMALLINT | SHOFT: 2字节的带符号整数。
- INTEGER | INT: 2、4或8字节的带符号整数。默认值为4。
- BIGINT | LONG: 8字节的带符号整数。
- UNSIGNED SMALLINT | SHORT: 2字节的无符号整数。
- UNSIGNED INTEGER | INT: 2或4字节的无符号整数。默认值为4。
- FLOAT | REAL: 4或8字节的浮点数。默认值为4。
- DOUBLE: 8字节的浮点数。

可以如下指定表示NULL的字符串。如果省略但输入为NULL，则在PARSE\_BADFILE中将NULL记录为无效数据。

- NULLIF '空字符串': 当类型为CHAR或VARCHAR时，指定表示NULL的字符串。字符串的长度必须与类型的长度相同。
- NULLIF 空值十六进制: 当类型不是CHAR和VARCHAR时，指定表示NULL的十六进制值。十六进制值的长度必须与类型的长度相同。

## H. 22. 6. 限制

### H. 22. 6. 1. lt\_bulkload的退出代码

当成功加载时，lt\_bulkload返回0。即使加载本身已完成，但存在一些解析错误或重复错误时，它还会返回带有“警告”消息的3。 请注意，跳过的行和替换的行（使用ON\_DUPLICATE\_KEEP = NEW）不被视为错误；退出代码将为0。

当存在不可继续的错误时，加载程序会引发一个“错误”消息。返回代码通常为1，因为在加载数据期间，数据库服务器会发生许多错误。 以下表格显示了lt\_bulkload可以返回的代码。

Return code	Description
0	Success
1	Error occurred during running SQL in LightDB
2	Failed to connect to LightDB

## H. 22. 6. 2. 关于直接加载

如果使用直接加载模式（WRITER=DIRECT或PARALLEL），则必须注意以下事项：

### H. 22. 6. 2. 1. LightDB启动顺序

当lt\_bulkload崩溃并且在\$LTDATA/lt\_bulkload中保留了一些.loadstatus文件时，必须使用“lt\_bulkload -r”命令进行lt\_bulkload自身的恢复，然后再调用lt\_ctl start。您必须使用LightDB脚本启动和停止LightDB，该脚本会正确地调用“lt\_bulkload -r”和“lt\_ctl start”。我们建议不要直接使用lt\_ctl。

如果您在Windows操作系统中使用lt\_bulkload，则lt\_bulkload包中不包含LightDB脚本。因此，您必须手动调用“lt\_bulkload -r”。

### H. 22. 6. 2. 2. PITR/复制

由于绕过了WAL，因此PITR的存档恢复不可用。这并不意味着可以在没有加载表数据的情况下进行PITR。如果要使用PITR，请在通过lt\_bulkload加载后对数据库进行完全备份。如果使用流式复制，则需要基于在lt\_bulkload之后进行备份的备份集重新创建备用站。

### H. 22. 6. 2. 3. \$LTDATA/lt\_bulkload中的加载状态文件

您不能删除\$LTDATA/lt\_bulkload目录中找到的加载状态文件 (\*.loadstatus)。这个文件在lt\_bulkload崩溃恢复中是必需的。

### H. 22. 6. 2. 4. 不要使用kill -9

尽可能不要使用“kill -9”终止lt\_bulkload命令。如果您这样做了，您必须调用LightDB脚本执行lt\_bulkload恢复并重新启动LightDB以继续。

### H. 22. 6. 2. 5. 当MULTI\_PROCESS=YES时，身份验证可能会失败

当MULTI\_PROCESS=YES且需要从localhost连接到数据库以加载时，即使在提示符中正确输入密码，身份验证也会失败。要避免这种情况，请配置以下任一项。

- 使用“信任”方法对来自localhost的连接进行身份验证

在UNIX环境中，来自localhost的连接使用UNIX域套接字，在Windows中，它使用TCP/IP环回地址。在UNIX中，将以下行添加到lt\_hba.conf中。

```
# TYPE DATABASE USER CIDR-
ADDRESS METHOD [for UNIX]
local all foo
trust
```

在Windows中，将以下行添加到lt\_hba.conf中。

```
# TYPE DATABASE USER CIDR-
ADDRESS METHOD [for Windows]
host all foo
127.0.0.1/32 trust
```

- 在.pgpass文件中指定密码



如果出于安全原因不想使用“trust”方法，请使用“md5”或“password”作为身份验证方法，并在.pgpass文件中指定密码。请注意，.pgpass文件必须位于运行LightDB服务器的操作系统用户（通常为“LightDB”用户）的主目录中。例如，如果lt\_bulkload连接到以DB用户“foo”为身份，其密码为“foopass”的端口5432上运行的服务器，管理员可以将以下行添加到.pgpass文件中：

```
localhost:5432:*:foo:foopass
```

- 在.pgpass文件中指定密码

不要使用“WRITER=PARALLEL”

使用除“WRITER=PARALLEL”之外的加载方法。

### H. 22. 6. 3. 数据库约束

在默认情况下，仅强制执行唯一约束和非空约束以进行数据加载。您可以设置“CHECK\_CONSTRAINTS=YES”以检查CHECK约束。无法检查外键约束。用户有责任提供有效的数据集。

## H. 22. 7. 详细信息

### H. 22. 7. 1. 如何编写FILTER函数

当您编写FILTER函数时，有一些注意事项和警告：

- 输入文件中的记录逐个传递给FILTER函数。
- 当FILTER函数发生错误时，传递的记录不会加载并写入PARSE BADFILE。
- FILTER函数必须返回记录类型或一些复合类型。此外，实际记录类型必须与目标表定义匹配。
- 如果FILTER函数返回NULL，则加载具有所有列中NULL的记录。
- 支持具有默认参数的函数。如果输入数据的列数少于函数的参数数，则将使用默认值。
- 不支持VARIADIC函数。
- SETOF函数不受支持。
- 不支持具有通用类型（任何，任何元素等）的函数。
- FILTER函数可以使用任何语言实现。SQL，C，PL等都可以，但是应尽可能快地编写函数，因为它们被调用的次数很多。
- 您只能指定FILTER或FORCE\_NOT\_NULL选项之一。如果需要此功能，请重新实现与FORCE\_NOT\_NULL兼容的FILTER函数。

这是一个FILTER函数的示例。

```
CREATE FUNCTION sample_filter(integer, text, text, real DEFAULT
0.05) RETURNS record
AS $$ SELECT $1 * $4, upper($3) $$
LANGUAGE SQL;
```

## H. 23. lt\_cheat\_funcs

该扩展提供了在LightDB上使用的作弊（但有用的）函数。

### H. 23. 1. 函数

请注意，在想要执行该扩展提供的函数的所有数据库中，都需要执行CREATE EXTENSION lt\_cheat\_funcs。

```
=# CREATE EXTENSION lt_cheat_funcs;
```

#### H. 23. 1. 1. SETOF record pg\_stat\_get\_memory\_context()

返回有关所有内存上下文的统计信息。此函数返回一个记录，如下表所示。

列名	数据类型	描述
name	text	上下文名称
parent	text	父级上下文的名称
level	integer	上下文树中距离TopMemoryContext的距离
total_bytes	bigint	从malloc请求的总字节数
total_nblocks	bigint	malloc块的总数
free_bytes	bigint	空闲空间的字节数
free_chunks	bigint	空闲块的数量
used_bytes	bigint	已使用空间的字节数

默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

#### H. 23. 1. 2. void pg\_stat\_print\_memory\_context()

让有关所有内存上下文的统计信息记录在日志中。每个内存上下文的日志消息格式如下：

```
[name]: [total_bytes] 总共 [total_nblocks] 个块; [free_bytes] 空闲 ([free_chunks] 个块); [used_bytes] 已使用
```

有关上述字段的描述，请参见 pg\_stat\_get\_memory\_context()。

默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

#### H. 23. 1. 3. record pg\_cached\_plan\_source(stmt text)

返回指定准备语句的缓存计划源的信息。此函数返回一个记录，如下表所示。

列名	数据类型	描述
generic_cost	double precision	通用计划的成本，如果不知道，则为-1
total_custom_cost	double precision	自定义计划的总成本
num_custom_plans	integer	总计划包含的计划数
force_generic	boolean	是否强制使用通用计划?
force_custom	boolean	是否强制使用自定义计划?

#### H. 23. 1. 4. void pg\_signal\_process(pid int, signame text)

向LightDB服务器进程发送信号。此函数仅能向postmaster、backend、walsender和walreceiver进程发送信号。有效的信号名称包括HUP、INT、QUIT、ABRT、KILL、TERM、USR1、USR2、CONT和STOP。默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

例如，终止walreceiver进程：

```
=# SELECT pg_signal_process(pid, 'TERM') FROM pg_stat_wal_receiver;
```

#### H. 23. 1. 5. integer pg\_get\_priority(pid int)

返回指定LightDB服务器进程的调度优先级（“nice”）。此函数仅能获取postmaster、backend、walsender和walreceiver进程的优先级。有关调度优先级的详细信息，请参见getpriority(2)手册页。默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

#### H. 23. 1. 6. void pg\_set\_priority(pid int, priority int)

将指定LightDB服务器进程的调度优先级（“nice”）设置为指定的值。此函数仅能更改postmaster、backend、walsender和walreceiver进程的优先级。有关调度优先级的详细信息，请参见getpriority(2)手册页。默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

#### H. 23. 1. 7. void pg\_segmentation\_fault(treat\_fatal\_as\_error boolean)

引起分段错误。如果启用了lt\_cheat\_funcs.exit\_on\_segfault并且treat\_fatal\_as\_error为true，则此函数引起的分段错误将导致ERROR而不是FATAL错误。这主要用于测试。默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

#### H. 23. 1. 8. void pg\_process\_config\_file()

读取并处理配置文件。请注意，如果发生错误，则会使用DEBUG2级别记录错误。默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

#### H. 23. 1. 9. text pg\_xlogfile\_name(location pg\_lsn, recovery boolean)

将事务日志位置字符串转换为文件名。此函数与LightDB核心提供的pg\_xlogfile\_name()函数几乎相同。它们之间的区别在于是否有第二个类型为boolean的参数。如果为false，则此函数在恢复过程中始终失败并显示错误信息。这与pg\_xlogfile\_name()的核心版本相同。如果为true，则即使在恢复过程中也可以执行此函数。但请注意，返回的WAL文件名的前8个数字（表示时间线ID）可能完全不正确。也就是说，此函数可能返回虚假的WAL文件名。有关此转换的详细信息，请参见 Section 9.27.3

#### H. 23. 1. 10. SETOF record pg\_stat\_get\_syncprep\_waiters()

返回所有等待同步复制的服务器进程的统计信息。此函数为每个等待同步复制的服务器进程返回一个记录，如下表所示。

列名	数据类型	描述
pid	integer	服务器进程的进程ID
wait_lsn	pg_lsn	等待的事务日志位置
wait_mode	text	服务器进程的等待模式

wait\_mode的可能值为write、flush和apply。

默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

### H. 23. 1. 11. void pg\_wait\_syncrep(location pg\_lsn)

等待同步复制。此函数一直等待，直到根据synchronous\_commit的设置，给定的事务日志位置被同步备用实例确认。

默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

### H. 23. 1. 12. xid pg\_set\_next\_xid(transactionid xid)

设置并返回下一个事务ID。请注意，此函数不会检查将给定的事务ID分配给下一个事务ID是否安全。调用者必须谨慎选择安全的事务ID，例如，不会引起事务ID环绕问题。默认情况下，此函数仅限超级用户使用，但可以授予其他用户EXECUTE权限以运行此函数。

### H. 23. 1. 13. SETOF record pg\_xid\_assignment()

返回有关事务ID分配状态的信息。此函数返回一个记录，如下表所示。

列名	数据类型	描述
next_xid	xid	要分配的下一个事务ID
oldest_xid	xid	集群中最小的datfrozenxid
xid_vac_limit	xid	开始强制执行自动清理的位置
xid_warn_limit	xid	开始发出警告的位置
xid_stop_limit	xid	拒绝将下一个事务ID超越此处
xid_wrap_limit	xid	世界末日的位置
oldest_xid_db	oid	具有最小datfrozenxid的数据库

默认情况下，此函数仅限超级用户使用，但可以授予其他用户执行此函数的EXECUTE权限。

### H. 23. 1. 14. xid pg\_set\_next\_oid(objectid oid)

设置并返回下一个对象ID (OID)。请注意，当给定的OID小于16384 (FirstNormalObjectId) 时，下一个OID将被设置为16384。默认情况下，此函数仅限超级用户使用，但可以授予其他用户执行此函数的EXECUTE权限。

### H. 23. 1. 15. SETOF record pg\_oid\_assignment()

返回有关对象ID (OID) 分配状态的信息。此函数返回一个记录，如下表所示。

列名	数据类型	描述
next_oid	oid	要分配的下一个对象ID
oid_count	integer	必须执行WAL工作之前可用的OID数量

默认情况下，此函数仅限超级用户使用，但可以授予其他用户执行此函数的EXECUTE权限。

### H. 23. 1. 16. integer pg\_advance\_vacuum\_cleanup\_age(integer)

指定VACUUM和HOT更新清理死行版本所需推进的事务数。调用此函数的会话中的vacuum\_defer\_cleanup\_age 被设置为指定数字的负值。如果省略了参数或指定了NULL，

则会将vacuum\_defer\_cleanup\_age重置为配置文件中指定的原始设置值。 该函数返回清理VACUUM的年龄 (vacuum cleanup age) 。

通过推进清理年龄，VACUUM和HOT更新可以清理甚至是自最旧事务开始以来产生的死元组。因此，这个函数对于防止长时间运行的事务期间由于无法删除的死元组而导致数据库膨胀非常有帮助。

请注意，这是一项极其危险的功能，很容易损坏数据库。任何重要数据都可能消失，数据一致性可能完全丢失。这个函数除了调试目的外，绝不能用于其他目的。

默认情况下，此函数仅限超级用户使用，但其他用户也可以被授予EXECUTE权限来运行该函数。

### H. 23. 1. 17. void pg\_checkpoint(fast bool, wait bool, force bool)

执行检查点。如果快速 (fast) 为true (默认值)，检查点将尽快完成。 否则，为了最小化查询处理的影响，检查点所需的I/O将分布在一段时间内。 如果等待 (wait) 为true (默认值)，则此函数在返回之前等待检查点完成。 否则，它只是向检查点发出信号并返回。 如果强制 (force) 为true (默认值)，则此函数即使自上次检查点以来没有WAL活动也会强制执行检查点。 默认情况下，此函数仅限超级用户使用，但其他用户也可以被授予EXECUTE权限来运行该函数。

### H. 23. 1. 18. SETOF 记录 pg\_recovery\_settings()

返回recovery.conf中所有参数设置的信息。此函数返回一个记录，如下表所示。

列名	数据类型	描述
名称	文本	配置参数名称
设置	文本	参数值

默认情况下，此函数仅限超级用户使用，但其他用户也可以被授予EXECUTE权限来运行该函数。

### H. 23. 1. 19. 文本型 pg\_show\_primary\_conninfo()

返回primary\_conninfo恢复参数的当前值。如果尚未设置，则返回NULL。 默认情况下，此函数仅限超级用户使用，但其他用户也可以被授予EXECUTE权限来运行该函数。 有关primary\_conninfo参数的详细信息，请参见primary\_conninfo。

### H. 23. 1. 20. 整数型 pg\_postmaster\_pid()

返回postmaster进程的进程ID。

### H. 23. 1. 21. 带时区的时间戳型 pg\_backend\_start\_time()

返回当前会话所附加的服务器进程启动时间。

### H. 23. 1. 22. SETOF 文本型 pg\_list\_relation\_filepath(relation regclass)

列出指定关系的文件路径名称。

### H. 23. 1. 23. 文本型 pg\_tablespace\_version\_directory()

返回特定于主版本的表空间子目录的名称。

### H. 23. 1. 24. 大整数型 `pg_file_write_binary(filepath text, data bytea)`

将bytea数据写入文件。如果文件不存在，则此函数创建文件，并将其截断为零长度。然后，此函数从文件开头写入bytea数据，并返回写入的字节数。默认情况下，此函数仅限超级用户使用，但其他用户也可以被授予EXECUTE权限来运行该函数。

### H. 23. 1. 25. 无返回值 `pg_file_fsync(filepath text)`

尝试对文件或目录进行fsync。默认情况下，此函数仅限超级用户使用，但其他用户也可以被授予EXECUTE权限来运行该函数。

### H. 23. 1. 26. 文本型 `to_octal(integer or bigint)`

将数字转换为其等效的八进制表示。

### H. 23. 1. 27. 文本型 `pg_text_to_hex(str text)`

将文本转换为其等效的十六进制表示。

这是一个将文本转换为十六进制的示例：

```
=# SELECT pg_text_to_hex('LightDB');
   pg_text_to_hex
-----
4c696768744442
(1 row)
```

### H. 23. 1. 28. 文本型 `pg_hex_to_text(hex text)`

将十六进制表示转换为其等效的文本。

这是一个将十六进制转换为文本的示例：

```
=# SELECT pg_hex_to_text('4c696768744442');
   pg_hex_to_text
-----
LightDB
(1 row)
```

### H. 23. 1. 29. 文本型 `pg_chr(code integer)`

返回具有给定代码的字符。此函数与LightDB核心提供的chr()函数几乎相同。它们的区别在于，当请求的字符过大或无效时，此函数返回NULL，而不是抛出错误。请注意，有效的Unicode代码点停止于U+10FFFF（1114111），即使4字节的UTF8序列可以容纳高达U+1FFFFF的值。因此，每当给定的代码大于1114111时，此函数都会返回NULL。

### H. 23. 1. 30. 文本型 `pg_utf8(code integer)`

pg\_chr(code integer)函数的别名。

### H. 23. 1. 31. SETOF 记录型 `pg_all_utf8()`

返回所有有效的UTF-8字符。此函数返回一个记录，如下表所示。

列名	数据类型	描述
代码	文本	字符的代码

列名	数据类型	描述
utf8	文本	UTF-8字符

此函数仅可在UTF-8数据库编码下执行。

### H. 23. 1. 32. 文本型 pg\_eucjp(code1 bit(8), code2 bit(8), code3 bit(8))

返回具有给定代码的EUC\_JP字符。以下表格显示代码的有效组合。

code1	code2	code3
x00 - x7f	-	-
x8e	xa1 - xdf	-
xa1 - xfe	xa1 - xfe	-
x8f	xa1 - xfe	xa1 - xfe

例如，返回具有代码“a1fa”（黑色星号）的EUC\_JP字符：

```
=# SELECT pg_eucjp('xa1', 'xfa');
```

如果请求的字符对于EUC\_JP无效，则此函数返回NULL。此函数仅可在EUC\_JP数据库编码下执行。

### H. 23. 1. 33. SETOF记录 pg\_all\_eucjp()

返回所有有效的 EUC\_JP 字符。该函数返回一个记录，如下表所示。

列名	数据类型	描述
code1	text	字符的第一个字节
code2	text	字符的第二个字节（x00 表示该字节不存在）
code3	text	字符的第三个字节（x00 表示该字节不存在）
eucjp	text	EUC_JP 字符

该函数只能在 EUC\_JP 数据库编码下执行。

### H. 23. 1. 34. bytea pglz\_compress(data text)

创建并返回文本数据的压缩版本。该函数使用 PGLZ（一种用于 LightDB 的 LZ 压缩实现）进行压缩。如果压缩失败（例如，压缩结果实际上比原始数据更大），则该函数返回原始数据。请注意，即使压缩失败，返回数据可能比原始数据多 4 个字节，因为它始终存储 4 个字节的额外信息，如原始数据的长度。该函数返回的 bytea 数据需要使用 pg\_lz\_decompress() 进行解压缩，以获取原始文本数据。

### H. 23. 1. 35. bytea pglz\_compress\_bytea(data bytea)

创建并返回 bytea 数据的压缩版本。该函数使用 PGLZ（一种用于 LightDB 的 LZ 压缩实现）进行压缩。如果压缩失败（例如，压缩结果实际上比原始数据更大），则该函数返回原始数据。请注意，即使压缩失败，返回数据可能比原始数据多 4 个字节，因为它始终存储 4 个字节的额外信息，如原始数据的长度。该函数返回的 bytea 数据需要使用 pg\_lz\_decompress\_bytea() 进行解压缩，以获取原始 bytea 数据。

### H. 23. 1. 36. text pglz\_decompress(data bytea)

将 bytea 数据的压缩版本解压缩为文本。请注意，该函数的输入必须是 pg\_lz\_compress() 或 pg\_lz\_compress\_bytea() 返回的 bytea 数据。否则该函数可能返回损坏的数据。

### H. 23. 1. 37. `bytea pg_lz_decompress_bytea(data bytea)`

将 `bytea` 数据的压缩版本解压缩为 `bytea`。请注意，该函数的输入必须是 `pg_lz_compress()` 或 `pg_lz_compress_bytea()` 返回的 `bytea` 数据。否则该函数可能返回损坏的数据。

### H. 23. 1. 38. `text pg_saslprep(input text)`

使用 SASLprep 对输入字符串进行规范化处理。SASLprep 规范化主要用于将用户提供的密码处理为 SCRAM 认证的规范形式。请注意，如果输入不是有效的 UTF-8 字符串或规范化后的字符串包含 SASLprep 档案禁止使用的字符，则会引发错误。

### H. 23. 1. 39. `pg_advisory_xact_unlock(bigint)`

释放先前获取的排他事务级别的咨询锁。如果成功释放锁，则返回 `true`。如果未持有锁，则返回 `false`，并且服务器将报告一个 SQL 警告。

### H. 23. 1. 40. `pg_advisory_xact_unlock_shared(bigint)`

释放先前获取的共享事务级别的咨询锁。如果成功释放锁，则返回 `true`。如果未持有锁，则返回 `false`，并且服务器将报告一个 SQL 警告。

### H. 23. 1. 41. `pg_advisory_xact_unlock(integer, integer)`

与 `pg_advisory_xact_unlock(bigint)` 相同。

### H. 23. 1. 42. `pg_advisory_xact_unlock_shared(integer, integer)`

与 `pg_advisory_xact_unlock_shared(bigint)` 相同。

## H. 23. 2. 编码转换

### H. 23. 2. 1. `pg_euc_jp_to_utf8`

这是从 EUC\_JP 到 UTF-8 的编码转换。它使用两个转换映射：普通映射和额外映射。它们分别定义在 `conv/euc_jp_to_utf8.map` 和 `conv/euc_jp_to_utf8.extra` 中。对于每个字符，首先查询普通映射。如果没有匹配，则查询额外映射。如果还没有匹配，则会引发错误。

普通映射的内容与 LightDB 提供的默认从 EUC\_JP 到 UTF-8 的转换映射 `euc_jp_to_utf8` 相同。额外映射包含一些普通映射没有的映射（例如，以下罗马数字和全角符号的映射）。

EUC_JP	UTF-8	描述
fcf1	e285b0 (U+2170)	小型罗马数字一
fcf2	e285b1 (U+2171)	小型罗马数字二
fcf3	e285b2 (U+2172)	小型罗马数字三
fcf4	e285b3 (U+2173)	小型罗马数字四
fcf5	e285b4 (U+2174)	小型罗马数字五
fcf6	e285b5 (U+2175)	小型罗马数字六
fcf7	e285b6 (U+2176)	小型罗马数字七
fcf8	e285b7 (U+2177)	小型罗马数字八
fcf9	e285b8 (U+2178)	小型罗马数字九
fcfa	e285b9 (U+2179)	小型罗马数字十
fcfb	efbfa2 (U+FFE2)	全角非符号



EUC_JP	UTF-8	描述
fcfc	efbfa4 (U+FFE4)	全角破折号
fcfd	efbc87 (U+FF07)	全角撇号
fcfe	efbc82 (U+FF02)	全角双引号

要将pg\_euc\_jp\_to\_utf8作为EUC\_JP到UTF-8的默认转换方式，需要启用其 pg\_conversion.condefault。 同时需要禁用euc\_jp\_to\_utf8（从EUC\_JP到UTF-8的内置转换）的condefault。 这里是这些目录更新的示例：

```

=# BEGIN;
=# UPDATE pg_conversion SET condefault = 'f' WHERE conname = 'euc_jp_to_utf8';
=# UPDATE pg_conversion SET condefault = 't' WHERE conname =
    'pg_euc_jp_to_utf8';
=# COMMIT;

```

可以通过直接修改映射文件并重建lt\_cheat\_funcs模块来使用自定义的转换映射表。 注意，映射文件中的条目必须按升序排序。

## H. 23. 3. 配置参数

请注意，如果要使用该扩展提供的配置参数，必须将 shared\_preload\_libraries 或 session\_preload\_libraries 设置为“lt\_cheat\_funcs”，并写入到lightdb.conf文件中。

### H. 23. 3. 1. lt\_cheat\_funcs.log\_memory\_context (布尔类型)

在查询执行结束时记录有关内存上下文的统计信息。有关日志格式的详细信息，请参见 pg\_stat\_print\_memory\_context()。 默认情况下，此参数处于关闭状态。只有超级用户可以更改此设置。

### H. 23. 3. 2. lt\_cheat\_funcs.hide\_appname (布尔类型)

如果为True，则忽略客户端的application\_name，并将其设置值存储在 lt\_cheat\_funcs.hidden\_appname参数中。 默认情况下，此参数设置为False，因此将使用客户指定的字符串作为应用程序名称。 此参数仅可在lightdb.conf文件中或服务器命令行中设置。

### H. 23. 3. 3. lt\_cheat\_funcs.hidden\_appname (字符串类型)

报告客户端隐藏的应用程序名称。默认为空字符串。任何用户都可以更改此设置。

### H. 23. 3. 4. lt\_cheat\_funcs.log\_session\_start\_options (布尔类型)

记录在连接开始时发送到服务器的选项。默认情况下，此参数处于关闭状态。 只有超级用户可以在会话开始时更改此参数，并且无法在会话内部更改此参数。

### H. 23. 3. 5. lt\_cheat\_funcs.scheduling\_priority (整数类型)

指定LightDB服务器进程的调度优先级（“nice”）。有效值介于-20和19之间。 较低的值会导致更有利的调度。默认值为0。任何用户都可以更改此设置。 有关调度优先级的详细信息，请参见getpriority(2)手册页。

### H. 23. 3. 6. lt\_cheat\_funcs.exit\_on\_segv (布尔类型)

如果为off，则缺页中断将导致服务器崩溃。如果为on，则仅终止导致缺页中断的当前会话。 默认值为off。任何用户都可以更改此设置。

## H. 24. ltcluster

有关更多详细信息，请参见 `ltcluster`<sup>1</sup>。

## H. 25. lt\_cron

### H. 25. 1. lt\_cron 是什么？

`lt_cron` 是 LightDB 的基于 `cron` 的简单作业调度程序，作为扩展运行在数据库内部。它允许您直接从数据库中调度 LightDB 命令：

```

-- 每周六的凌晨3点30分（美国东部时区）进行 VACUUM 操作
SELECT cron.schedule('30 3 * * 6', 'VACUUM');
schedule
-----
42

-- 每天上午10点（美国东部时区）进行 VACUUM 操作
SELECT cron.schedule('nightly-vacuum', '0 10 * * *', 'VACUUM');
schedule
-----
43

-- 将每天的 VACUUM 操作时间改为美国东部时区的凌晨3点
SELECT cron.schedule('nightly-vacuum', '0 3 * * *', 'VACUUM');
schedule
-----
43

-- 停止调度作业
SELECT cron.unschedule('nightly-vacuum' );
unschedule
-----
t
(1 row)

SELECT cron.unschedule(42);
unschedule
-----
t

```

`lt_cron` 可以支持秒级精度：

```

-- 每天上午10点零30秒（美国东部时区）进行 VACUUM 操作
SELECT cron.schedule('30 0 10 * * *', 'VACUUM');
schedule
-----
45

-- 每秒钟进行 VACUUM 操作
SELECT cron.schedule('daily-vacuum', '* * * * *', 'VACUUM');

```

<sup>1</sup> <http://www.light-pg.com/docs/ltcluster/13.8-23.4/index.html>

```
schedule
```

```
-----  
46
```

```
-- 将 VACUUM 操作时间改为每 10 秒钟一次
```

```
SELECT cron.schedule('daily-vacuum', '* /10 * * * *', 'VACUUM');
```

```
schedule
```

```
-----  
46
```

lt\_cron 可以支持四种任务模式，包括一次性任务、尽快执行任务、下一个时间间隔任务和固定时间间隔任务。您可以在第四个参数中传递任务模式，有四个参数可供选择（如果您想配置任务模式，则必须传递第一个参数任务名称）：

- 'single' 表示一次性任务，这意味着当任务第一次执行时，任务将不会再次执行。

```
-- 立即执行一次 VACUUM 操作
```

```
SELECT cron.schedule('daily-vacuum', '* * * * *', 'VACUUM',
```

```
'single');
```

```
schedule
```

```
-----  
46
```

```
-- 每 30 秒钟执行一次 VACUUM 操作
```

```
SELECT cron.schedule('daily-vacuum', '* /30 * * * *', 'VACUUM',
```

```
'next');
```

```
schedule
```

```
-----  
46
```

```
-- 仅在下一个上午10点零30秒（美国东部时区）执行一次 VACUUM 操作
```

```
SELECT cron.schedule('daily-vacuum', '30 0 10 * * *', 'VACUUM',
```

```
'single');
```

```
schedule
```

```
-----  
46
```

```
-- 每天上午10点零30秒（美国东部时区）执行 VACUUM 操作
```

```
SELECT cron.schedule('daily-vacuum', '30 0 10 * * *', 'VACUUM',
```

```
'next');
```

```
schedule
```

```
-----  
46
```

- 'asap' 表示尽快执行的任务，在同一任务中，每次最多运行一个作业实例。如果第二次运行在第一次完成之前应该开始，则第二次运行将排队并在第一次运行完成后立即启动。

```
-- 每 30 秒钟执行一次 VACUUM 操作
```

```
SELECT cron.schedule('daily-vacuum', '* /30 * * * *', 'VACUUM',
```

```
'asap');
```

```
schedule
```

```
-----  
46
```

```
-- 每天上午10点零30秒（美国东部时区）执行 VACUUM 操作
```

```
SELECT cron.schedule('daily-vacuum', '30 0 10 * * *', 'VACUUM',
'asap');
schedule
-----
46
```

- 'next' 表示下一个时间间隔调度任务，在同一任务中，每次最多运行一个作业实例。如果第二次运行在第一次完成之前应该开始，则第二次运行将排队并在下一个计时周期的时间点启动。

与以前版本兼容，模式参数输入 'timing' 与 'next' 相同。

```
-- 每 30 秒钟执行一次 VACUUM 操作
SELECT cron.schedule('daily-vacuum', '*/30 * * * *', 'VACUUM',
'next');
schedule
-----
46
```

```
-- 每天上午10点零30秒（美国东部时区）执行 VACUUM 操作
SELECT cron.schedule('daily-vacuum', '30 0 10 * * *', 'VACUUM',
'next');
schedule
-----
46
```

- 'fixed' 表示固定时间间隔调度任务，在同一任务中，默认情况下最多运行四个作业实例。如果第二次运行在第一次完成之前应该开始，则第二次运行不会等待，而会在定时周期的时间点启动，然后与第一个未完成的任务并行执行。

当同一任务过期时，您可以通过在 `lightdb.conf` 中配置 'cron.max\_connections\_per\_task' GUC 参数来修改最大并发执行数，并重新启动数据库以生效。最大上限为 16。

```
-- 每 30 秒钟执行一次 VACUUM 操作
SELECT cron.schedule('daily-vacuum', '*/30 * * * *', 'VACUUM',
'fixed');
schedule
-----
46
```

```
-- 每天上午10点零30秒（美国东部时区）执行 VACUUM 操作
SELECT cron.schedule('daily-vacuum', '30 0 10 * * *', 'VACUUM',
'fixed');
schedule
-----
46
```

lt\_cron 可以支持时区配置。您可以在第五个参数中传递时区值。如果您想配置时区，必须传递第一个参数任务名称和第四个参数任务模式。如果未配置时区，则默认为美国东部时区：

```
-- 每天上午10点（格林尼治标准时间）执行 VACUUM 操作
```

```

SELECT cron.schedule('daily-vacuum', '0 10 * * *', 'VACUUM', 'next',
'0');
schedule
-----
46

-- 每天上午10点（西十时区）执行 VACUUM 操作
SELECT cron.schedule('daily-vacuum', '0 10 * * *', 'VACUUM', 'next',
'-10');
schedule
-----
46

-- 仅在下一个上午10点（美国东部六时区）执行一次 VACUUM 操作
SELECT cron.schedule('daily-vacuum', '0 10 * * *', 'VACUUM',
'single', '6');
schedule
-----
46

```

lt\_cron 可以支持执行 Linux 操作系统命令。您可以在第六个参数中传递命令类型值。传递参数 'sql' 表示第二个参数为 SQL 命令，传递参数 'linux' 表示第二个参数为 Linux 命令。如果要配置命令类型，必须传递第一个参数任务名称、第四个参数任务模式和第五个参数时区。如果未配置命令类型，则默认为 SQL 命令。请注意，执行 Linux 命令必须是超级用户。

```

-- 每天上午10点（美国东部时区）执行 VACUUM 操作
SELECT cron.schedule('daily-vacuum', '0 10 * * *', 'VACUUM', 'next',
'8', 'sql');
schedule
-----
46

-- 每天晚上23点59分删除日志文件（美国东部时区）
SELECT cron.schedule('daily-touch', '59 23 * * *', 'rm -rf $LTDATA/
log/*', 'next', '8', 'linux');
schedule
-----
46

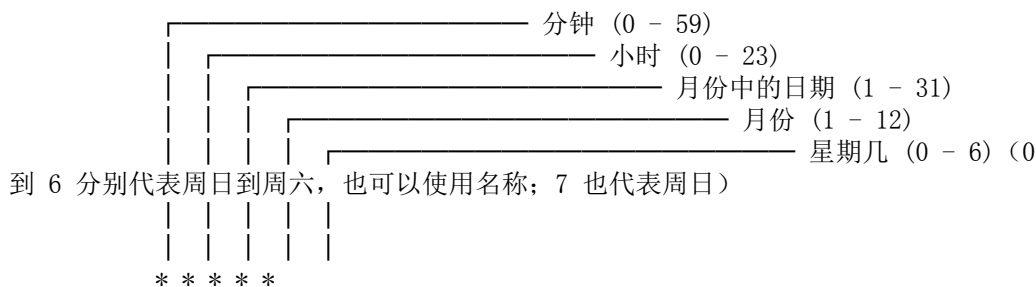
```

lt\_cron 可以并行运行多个作业，默认使用下一个时间间隔模式，即每次最多运行一个作业实例。如果第二次运行在第一次完成之前应该开始，则第二次运行将排队并在下一个计时的时间点启动。

lt\_cron 支持 SQL 命令的计划任务默认超时时间为 3 分钟，适用于所有类型的任务。如果任务执行超时，它将在 cron.job\_run\_details 中记录错误并返回，等待下一次执行。您可以通过在 lightdb.conf 中设置 cron.task\_running\_timeout GUC 参数来修改定时任务的超时时间，并重新启动数据库以生效。最大值为 60 分钟；如果设置为 0，则表示没有超时限制。

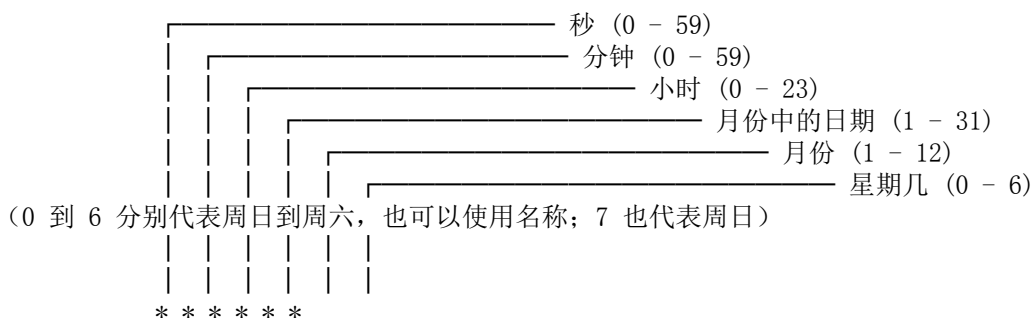
请注意，Linux 命令定时任务没有超时机制。

该计划使用标准的 cron 语法，其中 \* 表示“每个时间段运行”，而具体的数字表示“但仅在此时间运行”：



创建 cron 计划的简单方法是: `crontab.guru`<sup>2</sup>。

它在标准 cron 语法的基础上进行了增强, 支持秒级任务:



出于安全考虑, 作业以与当前用户相同的权限在调用 `cron.schedule` 函数的数据库中执行。此外, 用户仅能在 `cron.job` 表和 `cron.lt_job` 视图中看到自己的作业。

## H. 26. `lt_freemap`

`lt_freemap` 模块提供了一种检查自由空间图(FSM)的方法。它提供了一个名为 `pg_freemap` 的函数 (或两个重载的函数), 用于显示给定页面或关系中所有页面的自由空间图中记录的值。

默认情况下, 只允许超级用户和 `pg_stat_scan_tables` 角色的成员使用。可以使用 `GRANT` 授权给其他用户。

### H. 26. 1. 函数

`pg_freemap(rel regclass IN, blkno bigint IN) returns int2`

返回指定 `blkno` 的关系页面上, 根据FSM记录的自由空间的大小。

`pg_freemap(rel regclass IN, blkno OUT bigint, avail OUT int2)`

显示关系中每个页面上根据FSM记录的自由空间大小。返回一组 (`blkno bigint, avail int2`) 元组, 每个页面对应一个元组。

自由空间图中存储的值不是精确的。它们被舍入到 `BLCKSZ` 的  $1/256$  精度 (默认情况下为32字节), 并且在插入和更新元组时不会完全更新。

对于索引, 跟踪的是完全未使用的页面, 而不是页面内的自由空间。因此, 这些值没有实际意义, 只是用于判断页面是否为空或已满。

<sup>2</sup> <https://crontab.guru/>

Note

该接口在8.4版本中进行了更改，以反映在同一版本中引入的新FSM实现。

## H. 26. 2. 示例输出

```
lightdb@postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-----+-----
      0 |      0
      1 |      0
      2 |      0
      3 |     32
      4 |    704
      5 |    704
      6 |    704
      7 |   1216
      8 |    704
      9 |    704
     10 |    704
     11 |    704
     12 |    704
     13 |    704
     14 |    704
     15 |    704
     16 |    704
     17 |    704
     18 |    704
     19 |   3648
(20 rows)
```

```
lightdb@postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
-----
          1216
(1 row)
```

## H. 26. 3. 作者

初始版本由Mark Kirkwood <markir@paradise.net.nz>编写。在8.4版本中，由Heikki Linnakangas <heikki@enterprisedb.com>重新编写以适应新的FSM实现。

## H. 27. lt\_hint\_plan

### H. 27. 1. Name

lt\_hint\_plan -- 使用特殊格式的注释提示短语来控制执行计划。

### H. 27. 2. Synopsis

LightDB使用基于成本的优化器，它利用数据统计而不是静态规则。规划器（优化器）估计每个可能的SQL语句执行计划的成本，然后最终执行成本最低的执行计划。规划器尽力选择最佳的执行计划，但并不完美，因为它没有考虑数据之间的相关性等一些数据属性。

lt\_hint\_plan可以使用所谓的“提示”来微调执行计划，这些提示是特殊格式的SQL注释中的简单描述。

## H. 27. 3. Description

### H. 27. 3. 1. Basic Usage

lt\_hint\_plan会读取给定目标SQL语句中的特殊格式注释中的提示短语。这种特殊格式以字符序列 “/\*+” 开始，以 “\*/” 结尾。提示短语由提示名称和后面用括号括起来并由空格或逗号分隔的参数组成。每个提示短语都可以通过换行符进行分隔以提高可读性。在SQL语句开头的提示称为全局级别提示，而在语句中的提示称为语句级别提示（仅影响当前查询块而不使用qb\_name提示）。

在下面的示例中，选择哈希连接作为连接方法，并使用顺序扫描方法扫描ltbench\_accounts表。

```
lightdb@postgres=# /*+
lightdb@postgres*#   HashJoin(a b)
lightdb@postgres*#   SeqScan(a)
lightdb@postgres*#  */
lightdb@postgres-# EXPLAIN SELECT *
lightdb@postgres-#   FROM ltbench_branches b
lightdb@postgres-#   JOIN ltbench_accounts a ON b.bid = a.bid
lightdb@postgres-#   ORDER BY a.aid;
                                QUERY PLAN
-----
Sort (cost=31465.84..31715.84 rows=100000 width=197)
Sort Key: a.aid
-> Hash Join (cost=1.02..4016.02 rows=100000 width=197)
    Hash Cond: (a.bid = b.bid)
-> Seq Scan on ltbench_accounts a (cost=0.00..2640.00 rows=100000
width=97)
-> Hash (cost=1.01..1.01 rows=1 width=100)
    -> Seq Scan on ltbench_branches b (cost=0.00..1.01 rows=1
width=100)
(7 rows)

lightdb@postgres=#
```

## H. 27. 4. The hint table

在上面的部分中，提示在特殊格式的注释中进行描述。在无法编辑查询的情况下，这种方式并不方便。在这种情况下，提示可以放置在名为“hint\_plan.hints”的特殊表中。该表由以下列组成。

Table H.12. hint table

列名	描述
id	用于标识提示行的唯一编号，该列由序列自动填充。
norm_query_string	与待提示的查询匹配的模式。查询中的常量必须替换为“?”，如以下示例所示。模式中的空格是有意义的。
application_name	应用提示的会话的application_name值。以下示例中的提示适用于从psql连接的会话。空字符串表示任何application_name的会话。
hints	提示短语。这必须是一系列提示，不包括周围的注释标记。



以下示例展示了如何操作提示表。

```
lightdb@postgres=# INSERT INTO hint_plan.hints(norm_query_string,
application_name, hints)
lightdb@postgres=# VALUES (
lightdb@postgres=# 'EXPLAIN (COSTS false) SELECT * FROM t1 WHERE t1.id
= ?;',
lightdb@postgres=# '',
lightdb@postgres=# 'SeqScan(t1)'
lightdb@postgres=# );
INSERT 0 1
lightdb@postgres=# UPDATE hint_plan.hints
lightdb@postgres=# SET hints = 'IndexScan(t1)'
lightdb@postgres=# WHERE id = 1;
UPDATE 1
lightdb@postgres=# DELETE FROM hint_plan.hints
lightdb@postgres=# WHERE id = 1;
DELETE 1
lightdb@postgres=#
```

提示表由创建者用户拥有，并在创建时拥有默认权限。在创建扩展期间，表提示优先于注释提示。

## H. 27. 4. 1. The types of hints

根据提示短语可以对其进行分类，按照其可以影响计划的对象类型和方式进行分类，分为八种类型。扫描方法、连接方法、连接顺序、行数修正、并行查询、GUC设置、查询块名称和其他提示。您可以在提示列表中查看每种类型的提示短语列表。

### H. 27. 4. 1. 1. Hints for scan methods

扫描方法提示会强制在目标表上执行特定的扫描方法。如果有任何别名名称，`lt_hint_plan`将通过别名来识别目标表。在此类提示中，它们是“SeqScan”、“IndexScan”等。

扫描提示对普通表、继承表、UNLOGGED表、临时表和系统目录有效。外部（外键）表、表函数、VALUES子句、CTE、视图和子查询不受影响。

```
lightdb@postgres=# /*+
lightdb@postgres*# SeqScan(t1)
lightdb@postgres*# IndexScan(t2 t2_pkey)
lightdb@postgres*# */
lightdb@postgres=# SELECT * FROM table1 t1 JOIN table table2 t2 ON (t1.key =
t2.key);
```

为了与Oracle兼容，还支持“full”和“index”关键字。它们只是“seqscan”、“indexscan”的别名。

```
lightdb@postgres=# EXPLAIN SELECT* FROM t2, t1 WHERE t1.id = t2.id;
QUERY PLAN
```

```
-----
Hash Join (cost=60.85..99.39 rows=2260 width=16)
Hash Cond: (t2.id = t1.id)
-> Seq Scan on t2 (cost=0.00..32.60 rows=2260 width=8)
-> Hash (cost=32.60..32.60 rows=2260 width=8)
    -> Seq Scan on t1 (cost=0.00..32.60 rows=2260 width=8)
(5 rows)
```

```
lightdb@postgres=# EXPLAIN SELECT /*+index(t1)*/ FROM t2, t1 WHERE t1.id =
t2.id;
LOG:  lt_hint_plan:
used hint:
INDEX(t1@lt#0)
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Hash Join (cost=61.01..111.85 rows=2260 width=16)
Hash Cond: (t1.id = t2.id)
-> Index Scan using t1_pkey on t1 @lt#0 (cost=0.15..45.06 rows=2260 width=8)
-> Hash (cost=32.60..32.60 rows=2260 width=8)
    -> Seq Scan on t2 @lt#0 (cost=0.00..32.60 rows=2260 width=8)
(5 rows)
```

```
lightdb@postgres=# EXPLAIN SELECT /*+full(t1)*/ FROM t2, t1 WHERE t1.id =
t2.id;
LOG:  lt_hint_plan:
used hint:
FULL(t1@lt#0)
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Hash Join (cost=60.85..99.39 rows=2260 width=16)
Hash Cond: (t1.id = t2.id)
-> Seq Scan on t1 @lt#0 (cost=0.00..32.60 rows=2260 width=8)
-> Hash (cost=32.60..32.60 rows=2260 width=8)
    -> Seq Scan on t2 @lt#0 (cost=0.00..32.60 rows=2260 width=8)
(5 rows)
```

```
lightdb@postgres=#
```

#### H. 27.4.1.2. Hints for join methods

这只会影响普通表、继承表、UNLOGGED表、临时表、外部（外键）表、系统目录、表函数、VALUES命令结果和CTE的连接。但是对视图和子查询的连接不受影响。

为了与Oracle兼容，还支持“use\_hash”、“use\_nl”、“use\_merge”、“no\_use\_hash”、“no\_use\_nl”和“no\_use\_merge”关键字。

当lightdb\_dblevel\_syntax\_compatible\_type为“off”时，连接方法提示可以指定两个或多个表，并将强制执行涉及指定表的连接方法。

```
lightdb@postgres=# EXPLAIN (COSTS false) SELECT /*+leading(t1 t2 t3)*/ FROM t1,
t2 ,t3 WHERE t1.id = t2.id and t2.id=t3.id;
LOG:  lt_hint_plan:
used hint:
Leading(((t1@lt#0 t2@lt#0) t3@lt#0))
not used hint:
```

duplication hint:  
error hint:

QUERY PLAN

```
-----
Hash Join
Hash Cond: (t1.id = t3.id)
-> Hash Join
      Hash Cond: (t1.id = t2.id)
      -> Seq Scan on t1 @"lt#0"
      -> Hash
          -> Seq Scan on t2 @"lt#0"
-> Hash
      -> Seq Scan on t3 @"lt#0"
(9 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) SELECT /*+leading(t1 t2 t3) use_nl(t1
t2) use_merge(t1 t2 t3)*/ FROM t1, t2 ,t3 WHERE t1.id = t2.id and t2.id=t3.id;
LOG:  lt_hint_plan:
used hint:
USE_NL(t1@lt#0 t2@lt#0)
USE_MERGE(t1@lt#0 t2@lt#0 t3@lt#0)
Leading(((t1@lt#0 t2@lt#0) t3@lt#0))
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Merge Join
Merge Cond: (t1.id = t3.id)
-> Nested Loop
      -> Index Scan using t1_pkey on t1 @"lt#0"
      -> Index Scan using t2_pkey on t2 @"lt#0"
          Index Cond: (id = t1.id)
-> Index Scan using t3_pkey on t3 @"lt#0"
(7 rows)
```

lightdb@postgres=#

当lightdb\_dblevel\_syntax\_compatible\_type不为“off”时，连接方法提示可以指定一个或多个表，并指导优化器使用指定的连接方法将每个指定表与另一个数据源连接。当参考表被强制成连接的内部表时，优化器使用这些提示。如果参考表是外部表，则忽略这些提示。建议与连接顺序提示（'leading'、'ordered'）一起使用。

```
lightdb@postgres=# EXPLAIN (COSTS false) SELECT /*+leading(t1 t2 t3)*/ FROM t1,
t2 ,t3 WHERE t1.id = t2.id and t2.id=t3.id;
LOG:  lt_hint_plan:
used hint:
Leading(((t1@lt#0 t2@lt#0) t3@lt#0))
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Hash Join
```

```

Hash Cond: (t1.id = t3.id)
-> Hash Join
      Hash Cond: (t1.id = t2.id)
      -> Seq Scan on t1 @"lt#0"
      -> Hash
          -> Seq Scan on t2 @"lt#0"
-> Hash
      -> Seq Scan on t3 @"lt#0"
(9 rows)

```

```

lightdb@postgres=# EXPLAIN (COSTS false) SELECT /*+leading(t1 t2 t3) use_nl(t2)
use_merge(t3)*/ FROM t1, t2 ,t3 WHERE t1.id = t2.id and t2.id=t3.id;

```

LOG: lt\_hint\_plan:

used hint:

USE\_NL(t2@lt#0)

USE\_MERGE(t3@lt#0)

Leading(((t1@lt#0 t2@lt#0) t3@lt#0))

not used hint:

duplication hint:

error hint:

#### QUERY PLAN

```

-----
Merge Join
Merge Cond: (t1.id = t3.id)
-> Nested Loop
      -> Index Scan using t1_pkey on t1 @"lt#0"
      -> Index Scan using t2_pkey on t2 @"lt#0"
          Index Cond: (id = t1.id)
-> Index Scan using t3_pkey on t3 @"lt#0"
(7 rows)

```

```

lightdb@postgres=# EXPLAIN (COSTS false) SELECT /*+leading(t1 t2 t3) use_nl(t1)
use_merge(t2)*/ FROM t1, t2 ,t3 WHERE t1.id = t2.id and t2.id=t3.id;

```

LOG: lt\_hint\_plan:

used hint:

USE\_MERGE(t2@lt#0)

Leading(((t1@lt#0 t2@lt#0) t3@lt#0))

not used hint:

USE\_NL(t1@lt#0)

duplication hint:

error hint:

#### QUERY PLAN

```

-----
Hash Join
Hash Cond: (t1.id = t3.id)
-> Merge Join
      Merge Cond: (t1.id = t2.id)
      -> Index Scan using t1_pkey on t1 @"lt#0"
      -> Index Scan using t2_pkey on t2 @"lt#0"
-> Hash
      -> Seq Scan on t3 @"lt#0"
(8 rows)

```

lightdb@postgres=#

当lightdb\_dblevel\_syntax\_compatible\_type不为“off”时，指定多个表的连接方法提示可以被视为指定一个表的连接方法提示组，因此当存在冲突时，它只会影响涉及的表。例如：“use\_hash (a b)”和“use\_nl (a)”发生冲突，但对于表b，也使用了“use\_hash”。

```
lightdb@postgres=# EXPLAIN (COSTS false) SELECT /*+leading(t1 t2 t3) use_nl(t2
t3) use_merge(t2)*/ FROM t1, t2 ,t3 WHERE t1.id = t2.id and t2.id=t3.id;
INFO:  lt_hint_plan: hint syntax error at or near "use_nl(t2 t3) use_merge(t2)"
DETAIL:  Conflict join method hint.
LOG:  lt_hint_plan:
used hint:
USE_NL(t3@1t#0) --> USE_NL(t2@1t#0 t3@1t#0)
Leading(((t1@1t#0 t2@1t#0) t3@1t#0))
not used hint:
duplication hint:
USE_NL(t2@1t#0) --> USE_NL(t2@1t#0 t3@1t#0)
USE_MERGE(t2@1t#0)
error hint:
```

QUERY PLAN

```
-----
Nested Loop
-> Hash Join
    Hash Cond: (t1.id = t2.id)
    -> Seq Scan on t1 @"1t#0"
    -> Hash
        -> Seq Scan on t2 @"1t#0"
-> Index Scan using t3_pkey on t3 @"1t#0"
    Index Cond: (id = t1.id)
(8 rows)
```

lightdb@postgres=#

当lightdb\_dblevel\_syntax\_compatible\_type不为“off”时，连接方法提示可能与“swap\_join\_inputs”提示发生冲突，而不使用“leading”提示。

```
lightdb@postgres=# EXPLAIN (COSTS false) SELECT/*+leading(t1 t2) use_hash(t2)
swap_join_inputs(t2)*/ * FROM t1, t2 WHERE t1.id = t2.id;
LOG:  lt_hint_plan:
used hint:
USE_HASH(t2@1t#0)
Leading((t1@1t#0 t2@1t#0))
swap_join_inputs(t2@1t#0)
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Hash Join
Hash Cond: (t2.id = t1.id)
-> Seq Scan on t2 @"1t#0"
-> Hash
    -> Seq Scan on t1 @"1t#0"
(5 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) SELECT /*+use_hash(t2)
swap_join_inputs(t2)*/ * FROM t1, t2 WHERE t1.id = t2.id;
INFO:  lt_hint_plan: join hint conflict with swap_join_inputs hint, ignore
swap_join_inputs
DETAIL:  use USE_HASH
LOG:  lt_hint_plan:
used hint:
USE_HASH(t2@1t#0)
not used hint:
duplication hint:
swap_join_inputs(t2@1t#0)
error hint:
```

QUERY PLAN

```
-----
Hash Join
Hash Cond: (t1.id = t2.id)
-> Seq Scan on t1 @"1t#0"
-> Hash
    -> Seq Scan on t2 @"1t#0"
(5 rows)
```

```
lightdb@postgres=#
```

#### H. 27. 4. 1. 3. Hint for joining order

通常，“Leading”提示强制执行两个或多个表的连接顺序。有两种强制执行的方式。一种是强制执行连接的特定顺序，但不限制每个连接级别的方向。另一种方式是在强制执行连接顺序的同时，还强制执行连接方向。详细信息请参见提示列表表格。

```
lightdb@postgres=# /*+
lightdb@postgres*#      NestLoop(t1 t2)
lightdb@postgres*#      MergeJoin(t1 t2 t3)
lightdb@postgres*#      Leading(t1 t2 t3)
lightdb@postgres*#      */
lightdb@postgres=# SELECT * FROM table1 t1
lightdb@postgres=#      JOIN table table2 t2 ON (t1.key = t2.key)
lightdb@postgres=#      JOIN table table3 t3 ON (t2.key = t3.key);
```

为了与Oracle兼容，“ordered”提示也被支持。ORDERED提示指示LightDB按照FROM子句中出现的顺序连接表。LightDB的“ordered”提示只是“leading”提示的别名，因此它不会像Oracle一样覆盖“leading”提示。

```
lightdb@postgres=# EXPLAIN SELECT * FROM t3, t2, t1 WHERE t1.id = t2.id;
QUERY PLAN
```

```
-----
Nested Loop (cost=60.85..63982.64 rows=5107600 width=24)
-> Hash Join (cost=60.85..99.39 rows=2260 width=16)
    Hash Cond: (t2.id = t1.id)
    -> Seq Scan on t2 (cost=0.00..32.60 rows=2260 width=8)
    -> Hash (cost=32.60..32.60 rows=2260 width=8)
        -> Seq Scan on t1 (cost=0.00..32.60 rows=2260 width=8)
-> Materialize (cost=0.00..43.90 rows=2260 width=8)
    -> Seq Scan on t3 (cost=0.00..32.60 rows=2260 width=8)
(8 rows)
```

```
lightdb@postgres=# EXPLAIN SELECT /*+ordered*/ * FROM t3, t2, t1 WHERE t1.id =
t2.id;
LOG:  lt_hint_plan:
used hint:
Leading(@lt#0 ((t3@lt#0 t2@lt#0) t1@lt#0))
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Hash Join (cost=60.85..77409.29 rows=5107600 width=24)
Hash Cond: (t2.id = t1.id)
-> Nested Loop (cost=0.00..63915.85 rows=5107600 width=16)
    -> Seq Scan on t3 @"lt#0" (cost=0.00..32.60 rows=2260 width=8)
    -> Materialize (cost=0.00..43.90 rows=2260 width=8)
        -> Seq Scan on t2 @"lt#0" (cost=0.00..32.60 rows=2260 width=8)
-> Hash (cost=32.60..32.60 rows=2260 width=8)
    -> Seq Scan on t1 @"lt#0" (cost=0.00..32.60 rows=2260 width=8)
(8 rows)
```

```
lightdb@postgres=#
```

为了与Oracle兼容，“leading”提示也被支持，用于指定一个表作为连接的起点。

```
lightdb@postgres=# EXPLAIN SELECT * FROM t3, t2, t1 WHERE t1.id = t2.id;
QUERY PLAN
```

```
-----
Nested Loop (cost=60.85..63982.64 rows=5107600 width=24)
-> Hash Join (cost=60.85..99.39 rows=2260 width=16)
    Hash Cond: (t2.id = t1.id)
    -> Seq Scan on t2 (cost=0.00..32.60 rows=2260 width=8)
    -> Hash (cost=32.60..32.60 rows=2260 width=8)
        -> Seq Scan on t1 (cost=0.00..32.60 rows=2260 width=8)
-> Materialize (cost=0.00..43.90 rows=2260 width=8)
    -> Seq Scan on t3 (cost=0.00..32.60 rows=2260 width=8)
(8 rows)
```

```
lightdb@postgres=# EXPLAIN SELECT /*+leading(t1)*/ * FROM t3, t2, t1 WHERE t1.id
= t2.id;
LOG:  lt_hint_plan:
used hint:
Leading(t1@lt#0)
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Nested Loop (cost=10000000060.85..10000063982.64 rows=5107600 width=24)
-> Hash Join (cost=60.85..99.39 rows=2260 width=16)
    Hash Cond: (t1.id = t2.id)
    -> Seq Scan on t1 @"lt#0" (cost=0.00..32.60 rows=2260 width=8)
    -> Hash (cost=32.60..32.60 rows=2260 width=8)
        -> Seq Scan on t2 @"lt#0" (cost=0.00..32.60 rows=2260 width=8)
-> Materialize (cost=0.00..43.90 rows=2260 width=8)
    -> Seq Scan on t3 @"lt#0" (cost=0.00..32.60 rows=2260 width=8)
```

(8 rows)

```
lightdb@postgres=# EXPLAIN SELECT/*+leading(t3)*/ * FROM t3, t2, t1 WHERE t1.id
= t2.id;
```

```
LOG:  lt_hint_plan:
```

```
used hint:
```

```
Leading(t3@lt#0)
```

```
not used hint:
```

```
duplication hint:
```

```
error hint:
```

#### QUERY PLAN

```
-----
-
Nested Loop (cost=10000000000.16..10000064316.68 rows=5107600 width=24)
-> Seq Scan on t3 @"lt#0" (cost=0.00..32.60 rows=2260 width=8)
-> Materialize (cost=10000000000.16..10000000444.73 rows=2260 width=16)
    -> Nested Loop (cost=10000000000.16..10000000433.43 rows=2260
        width=16)
        -> Seq Scan on t2 @"lt#0" (cost=0.00..32.60 rows=2260 width=8)
        -> Index Scan using t1_pkey on t1 @"lt#0" (cost=0.15..0.18 rows=1
            width=8)
            Index Cond: (id = t2.id)
```

(7 rows)

```
lightdb@postgres=#
```

#### H.27.4.1.4. Hint for row number correction

“Rows” 提示可以纠正规划器限制导致的连接行数估计错误。

“Rows” 提示还可以纠正表或子查询的行数估计错误。

```
lightdb@postgres=# /*+ Rows(a #10) */ SELECT... ; Sets rows of table result to
10
lightdb@postgres=# /*+ Rows(a 10) */ SELECT... ; Sets rows of table result to 10
lightdb@postgres=# /*+ Rows(a +10) */ SELECT... ; Increments row number of
tableby 10
lightdb@postgres=# /*+ Rows(a -10) */ SELECT... ; Subtracts 10 from the row
number of table.
lightdb@postgres=# /*+ Rows(a *10) */ SELECT... ; Makes the row number of table
10 times larger.
lightdb@postgres=# /*+ Rows(a b #10) */ SELECT... ; Sets rows of join result to
10
lightdb@postgres=# /*+ Rows(a b 10) */ SELECT... ; Sets rows of join result to
10
lightdb@postgres=# /*+ Rows(a b +10) */ SELECT... ; Increments row number of
join by 10
lightdb@postgres=# /*+ Rows(a b -10) */ SELECT... ; Subtracts 10 from the row
number of join.
lightdb@postgres=# /*+ Rows(a b *10) */ SELECT... ; Makes the row number of join
10 times larger.

lightdb@postgres=# /*+ cardinality(a #10) */ SELECT... ; Sets rows of table
result to 10
```



```
lightdb@postgres=# /*+ cardinality(a 10) */ SELECT... ; Sets rows of table
result to 10
lightdb@postgres=# /*+ cardinality(a +10) */ SELECT... ; Increments row number
of table by 10
lightdb@postgres=# /*+ cardinality(a -10) */ SELECT... ; Subtracts 10 from the
row number of table.
lightdb@postgres=# /*+ cardinality(a *10) */ SELECT... ; Makes the row number of
table 10 times larger.
lightdb@postgres=# /*+ cardinality(a b #10) */ SELECT... ; Sets rows of join
result to 10
lightdb@postgres=# /*+ cardinality(a b 10) */ SELECT... ; Sets rows of join
result to 10
lightdb@postgres=# /*+ cardinality(a b +10) */ SELECT... ; Increments row number
of join by 10
lightdb@postgres=# /*+ cardinality(a b -10) */ SELECT... ; Subtracts 10 from the
row number of join.
lightdb@postgres=# /*+ cardinality(a b *10) */ SELECT... ; Makes the row number
of join 10 times larger.
```

对于参数化路径，该提示无效，但它会影响基于参数化路径的连接成本。这与Oracle不同，Oracle不会影响连接成本。

```
lightdb@postgres=# explain select /*+ use_nl(a b) leading(a b)*/ from t1 a , t2
b where a.key1=b.key1;
```

QUERY PLAN

```
-----
Nested Loop (cost=0.15..433.43 rows=2260 width=16)
  -> Seq Scan on t1 a @"t1#0" (cost=0.00..32.60 rows=2260 width=8)
  -> Index Scan using t2_pkey on t2 b @"t2#0" (cost=0.15..0.18 rows=1
width=8)
      Index Cond: (key1 = a.key1)
(4 rows)
```

```
lightdb@postgres=# explain select /*+ cardinality(b 1000)use_nl(a b) leading(a
b)*/ from t1 a , t2 b where a.key1=b.key1;
```

QUERY PLAN

```
-----
Nested Loop (cost=0.15..433.43 rows=1000 width=16)
  -> Seq Scan on t1 a @"t1#0" (cost=0.00..32.60 rows=2260 width=8)
  -> Index Scan using t2_pkey on t2 b @"t2#0" (cost=0.15..0.18 rows=1
width=8)
      Index Cond: (key1 = a.key1)
(4 rows)
```

```
chuhx@postgres=#
```

对于无法拉取的子查询，它将作用于子查询；对于已拉取的子查询，如果子查询中只有一个表，则将应用于子查询中的表。

```
chuhx@postgres=# explain with cte as (select distinct * from t1) select /*
+hashjoin(t2, cte) cardinality(cte 1000) */ from t2,cte where cte.key1=t2.key1;
```

QUERY PLAN

```
-----
Hash Join (cost=104.75..152.57 rows=1000 width=16)
  Hash Cond: (t1.key1 = t2.key1)
```

```

-> HashAggregate (cost=43.90..66.50 rows=2260 width=8)
    Group Key: t1.key1, t1.key2
    -> Seq Scan on t1 @"t1#1" (cost=0.00..32.60 rows=2260 width=8)
-> Hash (cost=32.60..32.60 rows=2260 width=8)
    -> Seq Scan on t2 @"t2#0" (cost=0.00..32.60 rows=2260 width=8)
(7 rows)

```

```

chuhx@postgres=# explain with cte as (select * from t1) select /*+hashjoin(t2,
cte) cardinality(cte 1000) */* from t2,cte where cte.key1=t2.key1;
          QUERY PLAN

```

```

-----
Hash Join (cost=45.10..83.64 rows=1000 width=16)
  Hash Cond: (t2.key1 = t1.key1)
  -> Seq Scan on t2 @"t2#0" (cost=0.00..32.60 rows=2260 width=8)
  -> Hash (cost=32.60..32.60 rows=1000 width=8)
      -> Seq Scan on t1 @"t1#1" (cost=0.00..32.60 rows=1000 width=8)
(5 rows)

```

```

chuhx@postgres=#

```

#### H. 27.4.1.5. Hint for parallel plan

“Parallel”提示强制执行扫描的并行执行配置。第三个参数指定了强制执行的强度。“soft”意味着`lt_hint_plan`仅更改`max_parallel_worker_per_gather`，而将其他参数保留给规划器。“hard”更改其他规划器参数，以强制应用数字。这可能会影响普通表、继承父项、非日志表和系统目录。外部表、表函数、值子句、CTE、视图和子查询不受影响。视图的内部表可以通过其实际名称/别名作为目标对象来指定。以下示例显示了如何在每个表上不同地强制执行查询。

```

lightdb@postgres=# explain /*+ Parallel(c1 3 hard) Parallel(c2 5 hard) */
SELECT c2.a FROM c1 JOIN c2 ON (c1.a = c2.a);
          QUERY PLAN

```

```

-----
Hash Join (cost=2.86..11406.38 rows=101 width=4)
  Hash Cond: (c1.a = c2.a)
  -> Gather (cost=0.00..7652.13 rows=1000101 width=4)
      Workers Planned: 3
      -> Parallel Seq Scan on c1 (cost=0.00..7652.13 rows=322613
width=4)
  -> Hash (cost=1.59..1.59 rows=101 width=4)
      -> Gather (cost=0.00..1.59 rows=101 width=4)
          Workers Planned: 5
          -> Parallel Seq Scan on c2 (cost=0.00..1.59 rows=59 width=4)

```

```

lightdb@postgres=# EXPLAIN /*+ Parallel(t1 5 hard) */ SELECT sum(a) FROM t1;
          QUERY PLAN

```

```

-----
Finalize Aggregate (cost=693.02..693.03 rows=1 width=8)
-> Gather (cost=693.00..693.01 rows=5 width=8)
    Workers Planned: 5
    -> Partial Aggregate (cost=693.00..693.01 rows=1 width=8)
        -> Parallel Seq Scan on t1 (cost=0.00..643.00 rows=20000
width=4)

```

这个“Parallel”提示也可以只指定一个整数来使用，表示它适用于当前查询块中的所有表。

```
lightdb@postgres=# explain select /*+parallel(3)*/ from t1, t2 where
t1.id=t2.id;
```

QUERY PLAN

```
-----
Hash Join (cost=28.25..34.19 rows=2260 width=16)
Hash Cond: (t1.id = t2.id)
-> Gather (cost=0.00..0.00 rows=2260 width=8)
    Workers Planned: 3
    -> Parallel Seq Scan on t1 @"lt#0" (cost=0.00..0.00 rows=729 width=8)
-> Hash (cost=0.00..0.00 rows=2260 width=8)
    -> Gather (cost=0.00..0.00 rows=2260 width=8)
        Workers Planned: 3
        -> Parallel Seq Scan on t2 @"lt#0" (cost=0.00..0.00 rows=729 width
=8)
(9 rows)
```

```
lightdb@postgres=#
```

#### H. 27. 4. 1. 6. 临时设置GUC参数

“Set”提示在规划过程中只更改GUC参数。在查询规划中显示的GUC参数可以对规划产生预期的影响，除非任何其他提示与规划器方法配置参数发生冲突。lt\_hint\_plan的GUC参数也可以通过此提示进行设置，但它可能不会像您期望的那样工作。

```
lightdb@postgres=# /*+ Set(random_page_cost 2.0) */
lightdb@postgres=# SELECT * FROM table1 t1 WHERE key = 'value';
...
```

‘Opt\_param’提示和‘Set’提示一样，用于在生成执行计划的过程中修改GUC参数。但是也有些与‘Set’提示不同的地方（不过与oracle的‘opt\_param’相同）：

- 参数名必须用单引号括起来。
- 字符型的参数值要用单引号括起来，数值型的不用。

```
EXPLAIN (COSTS false) select * from test_opt_param1 a join test_opt_param2 b on
a.key1=b.key1;
```

QUERY PLAN

```
-----
Hash Join
Hash Cond: (a.key1 = b.key1)
-> Seq Scan on test_opt_param1 a
-> Hash
    -> Seq Scan on test_opt_param2 b
(5 rows)
```

```
EXPLAIN (COSTS false) select/*+opt_param('enable_hashjoin', 'off')*/ * from
test_opt_param1 a join test_opt_param2 b on a.key1=b.key1;
```

```
LOG:  lt_hint_plan:
used hint:
opt_param('enable_hashjoin' 'off')
not used hint:
duplication hint:
```

error hint:

QUERY PLAN

```

Merge Join
  Merge Cond: (a.key1 = b.key1)
    -> Index Scan using test_opt_param1_pkey on test_opt_param1 a @"lt#0"
    -> Index Scan using test_opt_param2_pkey on test_opt_param2 b @"lt#0"
(4 rows)
EXPLAIN (COSTS false) select/*+opt_param(enable_hashjoin, 'off')*/ * from
  test_opt_param1 a join test_opt_param2 b on a.key1=b.key1;
INFO:  lt_hint_plan: hint syntax error at or near "opt_param(enable_hashjoin,
'off')"
DETAIL:  opt_param hint requires parameter name to be enclosed in single
  quotation marks.
LOG:  lt_hint_plan:
used hint:
not used hint:
duplication hint:
error hint:
opt_param(enable_hashjoin 'off')

```

QUERY PLAN

```

Hash Join
  Hash Cond: (a.key1 = b.key1)
    -> Seq Scan on test_opt_param1 a @"lt#0"
    -> Hash
        -> Seq Scan on test_opt_param2 b @"lt#0"
(5 rows)

```

```

lightdb@postgres=# explain select * from test_opt_param1 where key1=1;
                QUERY PLAN

```

```

Index Scan using test_opt_param1_pkey on test_opt_param1 (cost=0.15..2.17 r
ows=1 width=8)
  Index Cond: (key1 = 1)
(2 rows)

```

```

lightdb@postgres=# explain select/*+ Set(random_page_cost 1000) */ * from
  test_opt_param1 where key1=1;
                QUERY PLAN

```

```

Seq Scan on test_opt_param1 @"lt#0" (cost=0.00..38.25 rows=1 width=8)
  Filter: (key1 = 1)
(2 rows)

```

#### H. 27. 4. 1. 7. 查询块名称

使用qb\_name提示为查询块定义名称。然后可以在另一个查询块中使用该名称来提示出现在命名查询块中的表。如果两个或多个查询块具有相同的名称，或者同一个查询块用不同的名称提示两次，则所有名称和引用它们的提示都将被忽略。未使用此提示命名的查询块具有唯一的系统生成名称。

```
lightdb@postgres=# explain select /*+indexscan(t2@qb)*/ * from t1 ,(select /*
+qb_name(qb)*/ * from t2 where id >10) as tt;
```

```
LOG:  lt_hint_plan:
used hint:
IndexScan(t2@qb)
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Nested Loop (cost=0.13..3.21 rows=3 width=532)
-> Index Scan using idx_t2_id on t2 (cost=0.13..2.15 rows=1 width=8)
    Index Cond: (id > 10)
-> Seq Scan on t1 (cost=0.00..1.03 rows=3 width=524)
    (4 rows)
```

```
lightdb@postgres=#
lightdb@postgres=# explain select /*+indexscan(t2@qb) indexscan(t3@qb1)*/ * from
t1 ,(select /*+qb_name(qb)*/ * from t2 where id >10) as tt where exists (select/
+qb_name(qb1)*/ * from t3 where id>1);
```

QUERY PLAN

```
-----
Result (cost=2.28..5.35 rows=3 width=532)
One-Time Filter: $0
InitPlan 1 (returns $0)
-> Index Scan using idx_t3_id on t3 @qb1 (cost=0.13..2.15 rows=1 width=0)
    Index Cond: (id > 1)
-> Nested Loop (cost=2.28..5.35 rows=3 width=532)
    -> Index Scan using idx_t2_id on t2 @qb (cost=0.13..2.15 rows=1
width=8)
        Index Cond: (id > 10)
    -> Seq Scan on t1 @"1t#0" (cost=0.00..1.03 rows=3 width=524)
    (9 rows)
```

```
lightdb@postgres=#
```

#### H. 27. 4. 1. 8. 其他提示

不能归类为上述提示的提示。

##### H. 27. 4. 1. 8. 1. 不合并提示

“NO\_MERGE”提示指示优化器不要将外部查询和任何内联视图查询合并为单个查询。

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from t1, (select /*+no_merge*/
t2.id from t2,t3) tt where t1.id=tt.id;
```

```
LOG:  lt_hint_plan:
used hint:
no_merge
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```

-----
Hash Join
Hash Cond: (t2.id = t1.id)
-> Nested Loop
      -> Seq Scan on t2 @"1t#0"
      -> Materialize
          -> Seq Scan on t3 @"1t#0"
-> Hash
      -> Seq Scan on t1 @"1t#1"
(8 rows)

```

```

lightdb@postgres=# EXPLAIN (COSTS false) select/*+ no_merge(tt)*/ * from t1,
(select t2.id from t2,t3) tt where t1.id=tt.id;

```

```

LOG:  lt_hint_plan:
used hint:
no_merge(tt@1t#0)
not used hint:
duplication hint:
error hint:

```

QUERY PLAN

```

-----
Hash Join
Hash Cond: (t2.id = t1.id)
-> Nested Loop
      -> Seq Scan on t2
      -> Materialize
          -> Seq Scan on t3
-> Hash
      -> Seq Scan on t1 @"1t#0"
(8 rows)

```

```

lightdb@postgres=#

```

```

lightdb@postgres=# EXPLAIN (COSTS false) select/*+ no_merge(@qb)*/ * from t1,
(select/*+qb_name(qb)*/ t2.id from t2,t3) tt where t1.id=tt.id;

```

```

LOG:  lt_hint_plan:
used hint:
no_merge(@qb)
not used hint:
duplication hint:
error hint:

```

QUERY PLAN

```

-----
Hash Join
Hash Cond: (t2.id = t1.id)
-> Nested Loop
      -> Seq Scan on t2 @qb
      -> Materialize
          -> Seq Scan on t3 @qb
-> Hash
      -> Seq Scan on t1 @"1t#0"
(8 rows)

```

```

lightdb@postgres=#

```

H. 27. 4. 1. 8. 2. 分组方式提示（哈希/排序）

“use\_hash\_aggregation”提示指示优化器在分组时使用哈希聚合。  
 “no\_use\_hash\_aggregation”提示指示优化器在分组时不使用哈希聚合，而使用排序。

使用这些提示有两种方法。第一种没有参数，指定当前语句是否使用哈希算法，第二种与查询块名称组合使用。

以下示例显示了use\_hash\_aggregation提示的效果。

```
lightdb@postgres=# EXPLAIN (COSTS false) select max(id) from t1 where id>1
group by id order by id;
          QUERY PLAN
```

```
-----
GroupAggregate
Group Key: id
-> Index Only Scan using t1_pkey on t1
      Index Cond: (id > 1)
(4 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select/*+ use_hash_aggregation*/
max(id) from t1 where id>1 group by id order by id;
          QUERY PLAN
```

```
-----
Sort
Sort Key: id
-> HashAggregate
      Group Key: id
        -> Seq Scan on t1 @"1t#0"
          Filter: (id > 1)
(6 rows)
```

```
lightdb@postgres=#
```

以下示例显示了no\_use\_hash\_aggregation提示的效果。

```
lightdb@postgres=# EXPLAIN (COSTS false) select max(id) from t1 where id>1
group by id;
          QUERY PLAN
```

```
-----
HashAggregate
Group Key: id
-> Seq Scan on t1
      Filter: (id > 1)
(4 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select/*+ no_use_hash_aggregation*/
max(id) from t1 where id>1 group by id;
          QUERY PLAN
```

```
-----
GroupAggregate
Group Key: id
-> Index Only Scan using t1_pkey on t1 @"1t#0"
      Index Cond: (id > 1)
(4 rows)
```

```
lightdb@postgres=#
```

H. 27. 4. 1. 8. 3. 特殊连接提示（半连接/反连接）

半连接的提示

有“semijoin”，“hash\_sj”，“nl\_sj”，“merge\_sj”和“no\_semijoin”，用于控制是否使用半连接或指定半连接算法。反连接的提示

有“hash\_aj”，“nl\_aj”，“merge\_aj”，用于指定反连接算法。

使用这些提示有两种方法。第一种没有参数，必须放置在子链接中，第二种与查询块名称组合使用。

以下示例显示了使用半连接提示的效果。

```
lightdb@postgres=# create table test1 (key1 int primary key, key2 int);
CREATE TABLE
lightdb@postgres=# create table test2 (key1 int primary key, key2 int);
CREATE TABLE
lightdb@postgres=#
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where exists
(select * from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Hash Join
Hash Cond: (test1.key1 = test2.key1)
-> Seq Scan on test1
-> Hash
    -> Seq Scan on test2
(5 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where exists
(select /*+semijoin*/ from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Merge Semi Join
Merge Cond: (test1.key1 = test2.key1)
-> Index Scan using test1_pkey on test1 @"1t#1"
-> Index Only Scan using test2_pkey on test2 @"1t#0"
(4 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where exists
(select /*+hash_sj*/ from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Hash Semi Join
Hash Cond: (test1.key1 = test2.key1)
-> Seq Scan on test1 @"1t#1"
-> Hash
    -> Seq Scan on test2 @"1t#0"
(5 rows)
```

```
lightdb@postgres=#
```

以下示例显示了不使用半连接提示的效果。

```
lightdb@postgres=# create table tt1 (id int, t int, name varchar(255));
CREATE TABLE
lightdb@postgres=# create table tt2 (id int , salary int);
CREATE TABLE
```



```
lightdb@postgres=# create index idx_t1_id on tt1(id);
CREATE INDEX
lightdb@postgres=# create index idx_t2_id on tt2(id);
CREATE INDEX
lightdb@postgres=#
lightdb@postgres=# EXPLAIN (COSTS false) select * from tt1 where exists (select
* from tt2 where tt1.id=tt2.id);
QUERY PLAN
```

```
-----
Nested Loop Semi Join
-> Seq Scan on tt1
-> Index Only Scan using idx_t2_id on tt2
    Index Cond: (id = tt1.id)
(4 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from tt1 where exists
(select /*no_semijoin*/ from tt2 where tt1.id=tt2.id);
QUERY PLAN
```

```
-----
Hash Join
Hash Cond: (tt1.id = tt2.id)
-> Seq Scan on tt1 @"1t#1"
-> Hash
    -> HashAggregate
        Group Key: tt2.id
    -> Seq Scan on tt2 @"1t#0"
(7 rows)
```

```
lightdb@postgres=#
```

以下示例显示了使用反连接提示的效果。

```
lightdb@postgres=# create table tt1 (id int, t int, name varchar(255));
CREATE TABLE
lightdb@postgres=# create table tt2 (id int , salary int);
CREATE TABLE
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where not exists
(select 1 from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Hash Anti Join
Hash Cond: (test1.key1 = test2.key1)
-> Seq Scan on test1
-> Hash
    -> Seq Scan on test2
(5 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where not exists
(select/*+hash_aj*/ 1 from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Hash Anti Join
Hash Cond: (test1.key1 = test2.key1)
-> Seq Scan on test1 @"1t#1"
-> Hash
    -> Seq Scan on test2 @"1t#0"
(5 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where not exists
(select/*+nl_aj*/ 1 from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Nested Loop Anti Join
-> Seq Scan on test1 @"lt#1"
-> Index Only Scan using test2_pkey on test2 @"lt#0"
    Index Cond: (key1 = test1.key1)
(4 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where not exists
(select/*+merge_aj*/ 1 from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Merge Anti Join
Merge Cond: (test1.key1 = test2.key1)
-> Index Scan using test1_pkey on test1 @"lt#1"
-> Index Only Scan using test2_pkey on test2 @"lt#0"
(4 rows)
```

```
lightdb@postgres=#
```

“no\_semijoin”并不会阻止拉升子链接，而是在可以选择其他路径时不选择半连接路径。

no\_semijoin invalid:

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where exists
(select /*+no_semijoin*/ from test2 where test1.key1>test2.key2);
LOG:  lt_hint_plan:
used hint:
no_semijoin
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Nested Loop Semi Join
Join Filter: (test1.key1 > test2.key2)
-> Seq Scan on test1 @"lt#1"
-> Materialize
    -> Seq Scan on test2 @"lt#0"
(5 rows)
```

```
lightdb@postgres=#
```

当与连接方法提示一起使用时，连接方法提示将被忽略。以下示例展示了它。

```
lightdb@postgres=# explain select/*+nestloop(test1 test2@qb)*/ * from
test1 where exists (select /*+seminjoin qb_name(qb)*/ from test2 where
test1.key1=test2.key1);
INFO:  lt_hint_plan: join type hint is conflicted with special join hint, ignore
join type hint
DETAIL:  join type hint: NestLoop, special join hint: semijoin(@qb)
LOG:  lt_hint_plan:
used hint:
seminjoin
```

```
not used hint:
duplication hint:
NestLoop(test1@lt#0 test2@qb)
error hint:
```

QUERY PLAN

```
-----
Merge Semi Join (cost=0.31..124.01 rows=2260 width=8)
Merge Cond: (test1.key1 = test2.key1)
-> Index Scan using test1_pkey on test1 @"lt#0" (cost=0.15..45.06 rows=2260 width=8)
-> Index Only Scan using test2_pkey on test2 @qb (cost=0.15..45.06 rows=2260 width=4)
(4 rows)
```

lightdb@postgres=#

当与前导提示一起使用时，与这些提示冲突的前导提示部分将被忽略，但这仍可能使这些提示无效。当存在冲突时，结果可能不如预期，所以最好的方法是解决冲突。以下示例展示了冲突。

```
lightdb@postgres=# explain select /*+leading(test2@qb test1)*/ * from
test1 where exists (select /*+semijoin qb_name(qb)*/ * from test2 where
test1.key1=test2.key1);
INFO: lt_hint_plan: special join hint conflict with sub of leading hint, ignore
sub of leading hint
DETAIL: use special join hint: semijoin(@qb), the sub of leading has 2 relation
LOG: lt_hint_plan:
used hint:
Leading((test2@qb test1@lt#0))
semijoin
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Merge Semi Join (cost=0.31..124.01 rows=2260 width=8)
Merge Cond: (test1.key1 = test2.key1)
-> Index Scan using test1_pkey on test1 @"lt#0" (cost=0.15..45.06 rows=2260 width=8)
-> Index Only Scan using test2_pkey on test2 @qb (cost=0.15..45.06 rows=2260 width=4)
(4 rows)
```

lightdb@postgres=#

#### H. 27. 4. 1. 8. 4. 指定哈希连接的外部表的提示

这个提示“swap\_join\_inputs”指定哈希连接的外部表。

以下示例展示了“swap\_join\_inputs”的效果。

```

lightdb@postgres=# create table test1 (key1 int primary key, key2 int);
CREATE TABLE
lightdb@postgres=# create table test2 (key1 int primary key, key2 int);
CREATE TABLE
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1, test2 where
  test1.key1 = test2.key1;
      QUERY PLAN
-----
Hash Join
Hash Cond: (test1.key1 = test2.key1)
-> Seq Scan on test1
-> Hash
    -> Seq Scan on test2
(5 rows)

lightdb@postgres=# EXPLAIN (COSTS false) select /*+swap_join_inputs(test2)*/
  from test1, test2 where test1.key1 = test2.key1;
LOG:  lt_hint_plan:
used hint:
swap_join_inputs(test2@lt#0)
not used hint:
duplication hint:
error hint:

      QUERY PLAN
-----
Hash Join
Hash Cond: (test2.key1 = test1.key1)
-> Seq Scan on test2 @"lt#0"
-> Hash
    -> Seq Scan on test1 @"lt#0"
(5 rows)

lightdb@postgres=#

```

使用这个提示将强制SQL尽可能使用哈希连接。以下示例展示了它。

```

lightdb@postgres=# create table t_1(key1 int not null);
CREATE TABLE
lightdb@postgres=# create table t_2(key1 int not null);
CREATE TABLE
lightdb@postgres=# explain select /*+swap_join_inputs(t_2)*/ from t_1, t_2 where
  t_1.key1=t_2.key1;
      QUERY PLAN
-----
Merge Join  (cost=359.57..860.00 rows=32512 width=8)
Merge Cond: (t_1.key1 = t_2.key1)
-> Sort  (cost=179.78..186.16 rows=2550 width=4)
    Sort Key: t_1.key1
    -> Seq Scan on t_1  (cost=0.00..35.50 rows=2550 width=4)
-> Sort  (cost=179.78..186.16 rows=2550 width=4)
    Sort Key: t_2.key1
    -> Seq Scan on t_2  (cost=0.00..35.50 rows=2550 width=4)
(8 rows)

```

```
lightdb@postgres=# explain select /*+swap_join_inputs(t_2)*/ from t_1,t_2 where
  t_1.key1=t_2.key1;
LOG:  lt_hint_plan:
used hint:
swap_join_inputs(t_2@lt#0)
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Hash Join (cost=67.38..1247.18 rows=32512 width=8)
Hash Cond: (t_2.key1 = t_1.key1)
-> Seq Scan on t_2 @lt#0 (cost=0.00..35.50 rows=2550 width=4)
-> Hash (cost=35.50..35.50 rows=2550 width=4)
    -> Seq Scan on t_1 @lt#0 (cost=0.00..35.50 rows=2550 width=4)
(5 rows)
```

```
lightdb@postgres=#
```

当与前导提示一起使用时，与这些提示冲突的前导提示部分将被忽略。 以下示例展示了它。

```
lightdb@postgres=# EXPLAIN (COSTS false) select /*+ leading(test1 test2)
  swap_join_inputs(test2)*/ from test1, test2 where test1.key1 = test2.key1;
INFO:  lt_hint_plan: leading hint conflict with swap_join_inputs hint, ignore
  leading hint
DETAIL: use swap_join_hint(test2@lt#0)
LOG:  lt_hint_plan:
used hint:
Leading((test1@lt#0 test2@lt#0))
swap_join_inputs(test2@lt#0)
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Hash Join
Hash Cond: (test2.key1 = test1.key1)
-> Seq Scan on test2 @lt#0
-> Hash
    -> Seq Scan on test1 @lt#0
(5 rows)
```

```
lightdb@postgres=#
```

当与连接方法提示一起使用时，如果连接方法提示指定了非哈希连接方法，那么这个提示就不能使用。 以下示例展示了它。

```
lightdb@postgres=# EXPLAIN (COSTS false) select /*+ nestloop(test1 test2)
  swap_join_inputs(test2)*/ from test1, test2 where test1.key1 = test2.key1;
LOG:  lt_hint_plan:
used hint:
NestLoop(test1@lt#0 test2@lt#0)
not used hint:
swap_join_inputs(test2@lt#0)
```

duplication hint:  
error hint:

QUERY PLAN

```
-----
Nested Loop
-> Seq Scan on test1 @"lt#0"
-> Index Scan using test2_pkey on test2 @"lt#0"
      Index Cond: (key1 = test1.key1)
(4 rows)
```

lightdb@postgres=#

这个提示不应该用于半连接和反连接，因为半连接和反连接不能改变外部表。 当与半连接和反连接提示一起使用时，这个提示将被忽略。以下示例展示了它。

```
lightdb@postgres=# EXPLAIN (COSTS false) select /*+swap_join_inputs(test2@qb)*/
 * from test1 where exists ( select /*+semijoin qb_name(qb)*/ * from test2 where
 test1.key1 = test2.key1);
INFO:  lt_hint_plan: special join hint conflict with swap_join_inputs hint,
 ignore swap_join_inputs hint
DETAIL: ignore swap_join_hint(test2@qb), use semijoin(@qb)
LOG:  lt_hint_plan:
used hint:
semijoin
not used hint:
duplication hint:
swap_join_inputs(test2@qb)
error hint:
```

QUERY PLAN

```
-----
Merge Semi Join
Merge Cond: (test1.key1 = test2.key1)
-> Index Scan using test1_pkey on test1 @"lt#0"
-> Index Only Scan using test2_pkey on test2 @qb
(4 rows)
```

lightdb@postgres=#

#### H. 27. 4. 1. 8. 5. 保留谓词评估顺序的提示

这个提示“ordered\_predicates”强制优化器保留谓词评估的顺序，除了用作索引键的谓词。对于SELECT语句的WHERE子句使用这个提示。

您可以使用ordered\_predicates提示覆盖这些默认的评估规则，然后您的WHERE子句项将按照它们在查询中出现的顺序进行评估。 在查询的WHERE子句中使用函数时，通常会使用ordered\_predicates提示。 在您知道最具限制性的谓词并希望Lightdb首先评估它们的情况下，它也非常有用。

以下示例展示了“ordered\_predicates”的效果。

```
lightdb@postgres=# create table test1 (key1 int, key2 int);
CREATE TABLE
lightdb@postgres=# create table test2 (key1 int, key2 int);
CREATE TABLE
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where
 mod(key1,10)=2 and key2=2;
```

QUERY PLAN

```
Seq Scan on test1
  Filter: ((key2 = 2) AND (mod(key1, 10) = 2))
(2 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select /*+ordered_predicates*/ from
test1 where mod(key1,10)=2 and key2=2;
```

QUERY PLAN

```
Seq Scan on test1 @"lt#0"
  Filter: ((mod(key1, 10) = 2) AND (key2 = 2))
(2 rows)
```

```
lightdb@postgres=#
```

这个提示不会影响优化器生成的传递生成的谓词。

在以下示例中，“b.key2 = 1”被优化器重新生成，因此它的顺序不能被保留。

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 a left join test2 b
on a.key1=b.key1 and a.key2=b.key2 where mod(b.key2, 10) = 2 and b.key2 = 1;
```

QUERY PLAN

```
Nested Loop
  Join Filter: (a.key1 = b.key1)
  -> Seq Scan on test2 b
      Filter: ((key2 = 1) AND (mod(key2, 10) = 2))
  -> Seq Scan on test1 a
      Filter: (key2 = 1)
(6 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false) select/*+ordered_predicates*/ * from
test1 a left join test2 b on a.key1=b.key1 and a.key2=b.key2 where mod(b.key2,
10) = 2 and b.key2 = 1;
```

QUERY PLAN

```
Nested Loop
  Join Filter: (a.key1 = b.key1)
  -> Seq Scan on test2 b @"lt#0"
      Filter: ((key2 = 1) AND (mod(key2, 10) = 2))
  -> Seq Scan on test1 a @"lt#0"
      Filter: (key2 = 1)
(6 rows)
```

```
lightdb@postgres=#
```

#### H. 27. 4. 1. 8. 6. 控制提取子查询优化的提示

提取子查询的优化是RBO，它会尽可能地提取子查询。所以没有必要实现“unnest”提示，我们在这里只支持语法。

“no\_unnest”提示可用于强制不提取子查询。

以下示例展示了“no\_unnest”的效果。

```
lightdb@postgres=# create table test1 (key1 int, key2 int);
```

```
CREATE TABLE
lightdb@postgres=# create table test2 (key1 int, key2 int);
CREATE TABLE
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where exists(select
* from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Hash Join
  Hash Cond: (test1.key1 = test2.key1)
  -> Seq Scan on test1
  -> Hash
      -> HashAggregate
          Group Key: test2.key1
          -> Seq Scan on test2
```

(7 rows)

```
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1 where
exists(select/*+no_unnest*/ * from test2 where test1.key1=test2.key1);
QUERY PLAN
```

```
-----
Seq Scan on test1 @"lt#1"
  Filter: (alternatives: SubPlan 1 or hashed SubPlan 2)
  SubPlan 1
    -> Seq Scan on test2 @"lt#0"
        Filter: (test1.key1 = key1)
  SubPlan 2
    -> Seq Scan on test2 test2_1 @"lt#0"
```

(7 rows)

```
lightdb@postgres=#
```

#### H. 27. 4. 1. 8. 7. 控制并行连接模式的提示

现在'pq\_distribute'提示仅用于并行哈希连接，并且只有这种用法  
( 'pq\_distribute(inner\_table none broadcast)' )有效。

支持使用( 'pq\_distribute(inner\_table hash hash)' )，但不起作用。

当与'leading'提示一起使用时，只有在使用'leading'提示后指定的表是内部表时才会使用它。

以下示例展示了"pq\_distribute"的效果。

```
lightdb@postgres=# create table t1 (id int, val text);
CREATE TABLE
lightdb@postgres=# create table t2 (id int, val text);
CREATE TABLE
lightdb@lt_test=# /*+set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
lightdb@lt_test=# EXPLAIN (COSTS false) SELECT /*+ leading(t1 t2) hashjoin(t1
t2)*/ * FROM t1 join t2 on t1.id=t2.id;
QUERY PLAN
```

```
-----
Gather
  Workers Planned: 3
  -> Parallel Hash Join
      Hash Cond: (t1.id = t2.id)
```



```

-> Parallel Seq Scan on t1 @"lt#0"
-> Parallel Hash
    -> Parallel Seq Scan on t2 @"lt#0"
(7 rows)

lightdb@lt_test=# /*+set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
lightdb@lt_test=# EXPLAIN (COSTS false) select/*+leading(t1 t2) hashjoin(t1 t2)
pq_distribute(t2 none broadcast)*/ * from t1 join t2 on t1.id=t2.id;
QUERY PLAN

```

```

-----
Gather
Workers Planned: 3
-> Hash Join
    Hash Cond: (t1.id = t2.id)
    -> Parallel Seq Scan on t1 @"lt#0"
    -> Hash
        -> Seq Scan on t2 @"lt#0"
(7 rows)

```

```
lightdb@lt_test=#
```

当与' swap\_join\_inputs' 提示一起使用时，如果使用了' swap\_join\_inputs' 提示，它将始终被使用。它与Oracle不同，在Oracle中，如果指定的表在使用' swap\_join\_inputs' 后成为内部表，则' pq\_distribute' 将不起作用。

```

lightdb@lt_test=# /*+set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
EXPLAIN (COSTS false) select/*+leading(t1 t2) hashjoin(t1 t2) pq_distribute(t1
none broadcast)*/ * from t1 join t2 on t1.id=t2.id;
QUERY PLAN

```

```

-----
Gather
Workers Planned: 3
-> Parallel Hash Join
    Hash Cond: (t1.id = t2.id)
    -> Parallel Seq Scan on t1 @"lt#0"
    -> Parallel Hash
        -> Parallel Seq Scan on t2 @"lt#0"
(7 rows)

```

```

lightdb@lt_test=# /*+set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
EXPLAIN (COSTS false) select/*+leading(t1 t2) hashjoin(t1 t2) pq_distribute(t1
none broadcast) swap_join_inputs(t2)*/ * from t1 join t2 on t1.id=t2.id;
QUERY PLAN

```

```

-----
Gather
Workers Planned: 3
-> Hash Join
    Hash Cond: (t2.id = t1.id)
    -> Parallel Seq Scan on t2 @"lt#0"
    -> Hash
        -> Seq Scan on t1 @"lt#0"

```

(7 rows)

```
lightdb@lt_test=# /*+set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
EXPLAIN (COSTS false) select/*+leading(t1 t2) hashjoin(t1 t2) pq_distribute(t2
none broadcast) swap_join_inputs(t2)*/ * from t1 join t2 on t1.id=t2.id;
QUERY PLAN
```

```
-----
Gather
Workers Planned: 3
-> Hash Join
    Hash Cond: (t2.id = t1.id)
-> Parallel Seq Scan on t2 @"lt#0"
-> Hash
    -> Seq Scan on t1 @"lt#0"
```

(7 rows)

```
lightdb@lt_test=#
```

在没有指定连接顺序的情况下，'pq\_distribute' 提示可能会被使用，但不起作用，因为它仅在指定的表是内部表且连接是并行哈希连接时才能工作。 如果具有指定表作为外部表的另一路径的成本更低，优化器将选择另一路径。

```
lightdb@lt_test=# /*+set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
EXPLAIN (COSTS false) select/*+hashjoin(t1 t2)pq_distribute(t2 none broadcast)*/
* from t1 join t2 on t1.id=t2.id;
QUERY PLAN
```

```
-----
Gather
Workers Planned: 3
-> Parallel Hash Join
    Hash Cond: (t2.id = t1.id)
-> Parallel Seq Scan on t2 @"lt#0"
-> Parallel Hash
    -> Parallel Seq Scan on t1 @"lt#0"
```

(7 rows)

```
lightdb@lt_test=# /*+set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
EXPLAIN (COSTS false) select/*+hashjoin(t1 t2)pq_distribute(t1 none broadcast)*/
* from t1 join t2 on t1.id=t2.id;
QUERY PLAN
```

```
-----
Gather
Workers Planned: 3
-> Parallel Hash Join
    Hash Cond: (t1.id = t2.id)
-> Parallel Seq Scan on t1 @"lt#0"
-> Parallel Hash
    -> Parallel Seq Scan on t2 @"lt#0"
```

(7 rows)

```
lightdb@lt_test=# /*+set(parallel_tuple_cost 0) set(parallel_setup_cost 0)
set(min_parallel_table_scan_size 0) set(min_parallel_index_scan_size 0)
set(max_parallel_workers_per_gather 8)*/
EXPLAIN (COSTS false) select/*+hashjoin(t1 t2)pq_distribute(t1 none broadcast)
pq_distribute(t2 none broadcast)*/ * from t1 join t2 on t1.id=t2.id;
QUERY PLAN
```

```
-----
Gather
  Workers Planned: 3
  -> Hash Join
        Hash Cond: (t1.id = t2.id)
        -> Parallel Seq Scan on t1 @"lt#0"
        -> Hash
              -> Seq Scan on t2 @"lt#0"
```

(7 rows)

```
lightdb@lt_test=#
```

#### H. 27. 4. 1. 8. 8. 用于控制是否下推带子链接的过滤条件

在 LightDB 中，下推带子链接的过滤条件是一个 RBO 优化，过滤条件能下推就会下推。

“no\_push\_subq” 提示能够用来强制优化器不下推带子链接的过滤条件

“push\_subq” 提示能够用来强制优化器下推带子链接的过滤条件。但是因为 LightDB 默认就会尽可能下推带子链接的过滤条件，因此使用这个hint与不使用效果一致。如果你使用这个hint，在这个hint可以被使用时，它会被标记为已使用。

下面的例子展示了 “no\_push\_subq” 的作用。“push\_subq” 的用法与 “no\_push\_subq” 类似。

```
create table test_no_push_subq1 as select * from pg_class order by oid limit
100;
create table test_no_push_subq2 as select * from pg_class order by oid limit
100;
create table test_no_push_subq3 as select * from pg_class order by oid limit
100;
create table test_no_push_subq4 as select * from pg_class order by oid limit
100;
lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select a.oid from test_no_push_subq1 a join
test_no_push_subq3 b on a.reltypes=b.reltypes and a.oid = b.oid
lightdb@postgres=# where a.oid = (select max(oid) from test_no_push_subq2);
QUERY PLAN
```

```
-----
Nested Loop
  Join Filter: (a.reltypes = b.reltypes)
  InitPlan 1 (returns $0)
    -> Aggregate
          -> Seq Scan on test_no_push_subq2
    -> Seq Scan on test_no_push_subq1 a
          Filter: (oid = $0)
    -> Materialize
          -> Seq Scan on test_no_push_subq3 b
                Filter: (oid = $0)
```

(10 rows)

```
lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select a.oid from test_no_push_subq1 a join
test_no_push_subq3 b on a.reltypes=b.reltypes and a.oid = b.oid
```

```
lightdb@postgres=# where a.oid = (select/*+no_push_subq*/ max(oid) from
test_no_push_subq2);
```

QUERY PLAN

```
-----
Hash Join
  Hash Cond: ((a.reltype = b.reltype) AND (a.oid = b.oid))
  Join Filter: (a.oid = $0)
  InitPlan 1 (returns $0)
    -> Aggregate
        -> Seq Scan on test_no_push_subq2 @"1t#0"
  -> Seq Scan on test_no_push_subq1 a @"1t#1"
  -> Hash
        -> Seq Scan on test_no_push_subq3 b @"1t#1"
```

(9 rows)

```
lightdb@postgres=#
lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select a.oid from test_no_push_subq1 a join
test_no_push_subq3 b on a.reltype =b.reltype and a.oid = b.oid
lightdb@postgres=# where a.oid = (select max(oid) from test_no_push_subq2 c
where a.oid = c.oid);
```

QUERY PLAN

```
-----
Nested Loop
  Join Filter: ((a.reltype = b.reltype) AND (a.oid = b.oid))
  -> Seq Scan on test_no_push_subq3 b
  -> Materialize
        -> Seq Scan on test_no_push_subq1 a
            Filter: (oid = (SubPlan 2))
            SubPlan 2
              -> Result
                    InitPlan 1 (returns $1)
                      -> Limit
                            -> Seq Scan on test_no_push_subq2 c
                                Filter: ((oid IS NOT NULL) AND (a.oid =
oid))
(12 rows)
```

```
lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select a.oid from test_no_push_subq1 a join
test_no_push_subq3 b on a.reltype =b.reltype and a.oid = b.oid
lightdb@postgres=# where a.oid = (select/*+no_push_subq*/ max(oid) from
test_no_push_subq2 c where a.oid = c.oid);
```

QUERY PLAN

```
-----
Hash Join
  Hash Cond: ((a.reltype = b.reltype) AND (a.oid = b.oid))
  Join Filter: (a.oid = (SubPlan 2))
  -> Seq Scan on test_no_push_subq1 a @"1t#1"
  -> Hash
        -> Seq Scan on test_no_push_subq3 b @"1t#1"
  SubPlan 2
    -> Result
          InitPlan 1 (returns $1)
            -> Limit
                  -> Seq Scan on test_no_push_subq2 c @"1t#0"
                      Filter: ((oid IS NOT NULL) AND (a.oid = oid))
```

(12 rows)

```
lightdb@postgres=#
lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select * from test_no_push_subq1 a join test_no_push_subq3 b
on a.reltype =b.reltype
lightdb@postgres=# where (select oid from test_no_push_subq2 c where
c.oid=a.oid) = (select oid from test_no_push_subq3 d where d.oid=a.oid);
QUERY PLAN
```

-----

Nested Loop

```
Join Filter: (a.reltype = b.reltype)
-> Seq Scan on test_no_push_subq3 b
-> Materialize
-> Seq Scan on test_no_push_subq1 a
Filter: ((SubPlan 1) = (SubPlan 2))
SubPlan 1
-> Seq Scan on test_no_push_subq2 c
Filter: (oid = a.oid)
SubPlan 2
-> Seq Scan on test_no_push_subq3 d
Filter: (oid = a.oid)
```

(12 rows)

```
lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select * from test_no_push_subq1 a join test_no_push_subq3 b
on a.reltype =b.reltype
lightdb@postgres=# where (select/*+no_push_subq*/ oid from test_no_push_subq2 c
where c.oid=a.oid) = (select oid from test_no_push_subq3 d where d.oid=a.oid);
QUERY PLAN
```

-----

Hash Join

```
Hash Cond: (a.reltype = b.reltype)
Join Filter: ((SubPlan 1) = (SubPlan 2))
-> Seq Scan on test_no_push_subq1 a @"lt#2"
-> Hash
-> Seq Scan on test_no_push_subq3 b @"lt#2"
SubPlan 1
-> Seq Scan on test_no_push_subq2 c @"lt#0"
Filter: (oid = a.oid)
SubPlan 2
-> Seq Scan on test_no_push_subq3 d @"lt#1"
Filter: (oid = a.oid)
```

(12 rows)

```
lightdb@postgres=#
lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select a.oid from test_no_push_subq1 a join
test_no_push_subq3 b on a.reltype =b.reltype and a.oid = b.oid
lightdb@postgres=# where a.oid > all (select oid from test_no_push_subq2 c where
c.oid =a.oid);
QUERY PLAN
```

-----

Hash Join

```
Hash Cond: ((a.reltype = b.reltype) AND (a.oid = b.oid))
-> Seq Scan on test_no_push_subq1 a
Filter: (SubPlan 1)
SubPlan 1
```

```

-> Seq Scan on test_no_push_subq2 c
      Filter: (oid = a.oid)
-> Hash
      -> Seq Scan on test_no_push_subq3 b
(9 rows)

lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select a.oid from test_no_push_subq1 a join
test_no_push_subq3 b on a.reltypes=b.reltypes and a.oid = b.oid
lightdb@postgres=# where a.oid > all (select/*+ no_push_subq*/ oid from
test_no_push_subq2 c where c.oid =a.oid);
      QUERY PLAN

```

```

-----
Hash Join
  Hash Cond: ((a.reltypes = b.reltypes) AND (a.oid = b.oid))
  Join Filter: (SubPlan 1)
  -> Seq Scan on test_no_push_subq1 a @'1t#1'
  -> Hash
      -> Seq Scan on test_no_push_subq3 b @'1t#1'
  SubPlan 1
    -> Seq Scan on test_no_push_subq2 c @'1t#0'
      Filter: (oid = a.oid)
(9 rows)

```

```

lightdb@postgres=#
lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select a.oid from test_no_push_subq1 a join
test_no_push_subq3 b on a.reltypes=b.reltypes and a.oid = b.oid
lightdb@postgres=# where exists(select/*+no_unnest*/ * from test_no_push_subq2 c
where a.oid = c.oid and c.oid=2691);
      QUERY PLAN

```

```

-----
Hash Join
  Hash Cond: ((a.reltypes = b.reltypes) AND (a.oid = b.oid))
  -> Seq Scan on test_no_push_subq1 a @'1t#1'
      Filter: (alternatives: SubPlan 1 or hashed SubPlan 2)
  SubPlan 1
    -> Result
      One-Time Filter: (a.oid = '2691'::oid)
      -> Seq Scan on test_no_push_subq2 c @'1t#0'
        Filter: (oid = '2691'::oid)
  SubPlan 2
    -> Seq Scan on test_no_push_subq2 c_1 @'1t#0'
      Filter: (oid = '2691'::oid)
  -> Hash
      -> Seq Scan on test_no_push_subq3 b @'1t#1'
(14 rows)

```

```

lightdb@postgres=# EXPLAIN (COSTS false)
lightdb@postgres=# select a.oid from test_no_push_subq1 a join
test_no_push_subq3 b on a.reltypes=b.reltypes and a.oid = b.oid
lightdb@postgres=# where exists(select/*+no_unnest no_push_subq*/ * from
test_no_push_subq2 c where a.oid = c.oid and c.oid=2691);
      QUERY PLAN

```

```

-----
Hash Join
  Hash Cond: ((a.reltypes = b.reltypes) AND (a.oid = b.oid))
  Join Filter: (alternatives: SubPlan 1 or hashed SubPlan 2)

```

```

-> Seq Scan on test_no_push_subq1 a @"lt#1"
-> Hash
    -> Seq Scan on test_no_push_subq3 b @"lt#1"
SubPlan 1
  -> Result
    One-Time Filter: (a.oid = '2691'::oid)
    -> Seq Scan on test_no_push_subq2 c @"lt#0"
        Filter: (oid = '2691'::oid)
SubPlan 2
  -> Seq Scan on test_no_push_subq2 c_1 @"lt#0"
      Filter: (oid = '2691'::oid)
(14 rows)

```

lightdb@postgres=#

需要注意的是，在“push\_subq”与“no\_push\_subq”不作用于同一个子查询，但作用于同一个过滤条件时“push\_subq”起效，“no\_push\_subq”不被使用。与oracle一致，例子如下（SubPlan2 中的“no\_push\_subq”不被使用）；

```

EXPLAIN (COSTS false)
select * from test_push_subq1 a join test_push_subq3 b on a.reltype =b.reltype
where ((select/*+no_push_subq*/ relname from test_push_subq2 c where
c.oid=a.oid), (select/*+push_subq*/ reltype from test_push_subq2 c
where c.oid=a.oid)) = (select/*+no_push_subq*/ relname, reltype from
test_push_subq2);
LOG:  lt_hint_plan:
used hint:
no_push_subq
push_subq
not used hint:
no_push_subq
duplication hint:
error hint:

```

#### QUERY PLAN

```

-----
Hash Join
Hash Cond: (a.reltype = b.reltype)
Join Filter: ((SubPlan 1) = $2)
InitPlan 3 (returns $2,$3)
-> Seq Scan on test_push_subq2 @"lt#0"
-> Seq Scan on test_push_subq1 a @"lt#3"
    Filter: ((SubPlan 2) = $3)
    SubPlan 2
      -> Seq Scan on test_push_subq2 c_1 @"lt#2"
          Filter: (oid = a.oid)
-> Hash
    -> Seq Scan on test_push_subq3 b @"lt#3"
SubPlan 1
  -> Seq Scan on test_push_subq2 c @"lt#1"
      Filter: (oid = a.oid)
(15 rows)

```

#### H. 27. 4. 1. 8. 9. 用于忽略违反唯一键错误的提示

“ignore\_row\_on\_dupkey\_index”提示可用于忽略特定列集或指定索引的唯一键违规。目前使用这个提示等同于使用`on conflict do nothing`子句。如果使用了这个hint，且在插入数据时启用DML错误日志记录，则不会记录违反唯一键的日志，也不会导致语句终止。

如果违反了特定的规则，这个提示的语义效果将导致语句报错：

- 您必须指定一个正确的索引。如果没有指定索引，那么该语句将导致LT008错误(oracle是ORA-38912)。如果指定了多个索引，那么该语句会导致LT010错误(oracle是ORA-38915)。
- 如果指定了index，那么索引必须存在且唯一。否则，该语句导致LT009错误(oracle是ORA-38913)。
- IGNORE\_ROW\_ON\_DUPKEY\_INDEX提示仅适用于单表插入操作。它不支持更新、删除、合并(合并还不支持hint，将忽略)或多表插入操作，并将导致LT011错误(oracle是ORA-38917)。
- 如果指定的列对应多个索引，那么这条语句会导致LT010错误(oracle没有这个特性，因为同一个字段不能有多个索引)。
- 不能与on conflict子句一起使用，一起用会报LT011错误(oracle没有on conflict子句)。

下面的例子展示了“ignore\_row\_on\_dupkey\_index”提示的效果：

```

create table test_ignore_dupkey(key1 int, key2 int, key3 int, key4 int);
create unique index test_ignore_dupkey_ui1 on test_ignore_dupkey(key1, key2);
create unique index test_ignore_dupkey_ui2 on test_ignore_dupkey(key3, key4);
create index test_ignore_dupkey_i on test_ignore_dupkey(key3);

lightdb@postgres=# insert into test_ignore_dupkey values(1, 1, 1, 1);
INSERT 0 1
lightdb@postgres=# insert into test_ignore_dupkey values(1, 1, 1, 1);
ERROR: duplicate key value violates unique constraint "test_ignore_dupkey_ui1"
DETAIL: Key (key1, key2)=(1, 1) already exists.
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey
test_ignore_dupkey_ui1) */ into test_ignore_dupkey values(1, 1, 1, 2);
INSERT 0 0
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey(key1,
key2)) */ into test_ignore_dupkey values(1, 1, 1, 2);
INSERT 0 0
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey) */
into test_ignore_dupkey values(1, 1, 1, 2);
ERROR: An index must be specified in the index hint
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey
test_ignore_dupkey_ui3) */ into test_ignore_dupkey values(1, 1, 1, 2);
ERROR: An index must be specified in the index hint
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey
test_ignore_dupkey_i) */ into test_ignore_dupkey values(1, 1, 1, 2);
ERROR: Index specified in the index hint is invalid
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey
test_ignore_dupkey_ui1 test_ignore_dupkey_ui2) */ into test_ignore_dupkey
values(1, 1, 1, 2);
ERROR: Multiple indexes in ignore duplicate key hint
lightdb@postgres=# update/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey
test_ignore_dupkey_ui1) */ test_ignore_dupkey set key2 = 2 where key1 = 1;
ERROR: IGNORE_ROW_ON_DUPKEY_INDEX hint disallowed for this operation
lightdb@postgres=# create unique index test_ignore_dupkey_ui3 on
test_ignore_dupkey(key3, key4);
CREATE INDEX
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey(key3,
key4)) */ into test_ignore_dupkey values(1, 2, 1, 1);
ERROR: Multiple indexes in ignore duplicate key hint

```



```
DETAIL: find multiple indexes by fields
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey
test_ignore_dupkey_ui1) */ into test_ignore_dupkey values(1, 1, 1, 2)
lightdb@postgres=# on conflict (key1, key2) do nothing;
ERROR: IGNORE_ROW_ON_DUPKEY_INDEX hint disallowed for this operation
DETAIL: insert operation with on conflict clause is not supported
lightdb@postgres=#
```

ignore\_row\_on\_dupkey\_index”提示可以在canopy中工作，但只适用于insert select语句，不支持insert values语句。

```
create table test_ignore_dupkey(key1 int, key2 int, key3 int, key4 int);
SELECT create_distributed_table('test_ignore_dupkey', 'key1');
create unique index test_ignore_dupkey_ui1 on test_ignore_dupkey(key1, key2);
create unique index test_ignore_dupkey_ui2 on test_ignore_dupkey(key1, key3);
create index test_ignore_dupkey_i on test_ignore_dupkey(key3);
```

```
lightdb@postgres=# insert into test_ignore_dupkey values(1, 1, 1, 1);
INSERT 0 1
lightdb@postgres=# insert into test_ignore_dupkey values(1, 1, 1, 1);
ERROR: duplicate key value violates unique constraint
"test_ignore_dupkey_ui1_102045"
DETAIL: Key (key1, key2)=(1, 1) already exists.
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey
test_ignore_dupkey_ui1) */ into test_ignore_dupkey values(1, 1, 2, 1);
ERROR: duplicate key value violates unique constraint
"test_ignore_dupkey_ui1_102045"
DETAIL: Key (key1, key2)=(1, 1) already exists.
```

```
create table test_ignore_dupkey1(key1 int, key2 int, key3 int, key4 int);
insert into test_ignore_dupkey1 values(1, 1, 1, 1);
insert into test_ignore_dupkey1 values(1, 1, 1, 1);
insert into test_ignore_dupkey1 values(1, 2, 1, 1);
insert into test_ignore_dupkey1 values(1, 2, 1, 1);
insert into test_ignore_dupkey1 values(1, 1, 1, 2);
insert into test_ignore_dupkey1 values(1, 1, 1, 2);
insert into test_ignore_dupkey1 values(1, 3, 1, 3);
```

```
lightdb@postgres=# insert into test_ignore_dupkey select * from
test_ignore_dupkey1;
ERROR: duplicate key value violates unique constraint
"test_ignore_dupkey_ui1_102045"
DETAIL: Key (key1, key2)=(1, 1) already exists.
CONTEXT: COPY test_ignore_dupkey_102045, line 2
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey(key1,
key3)) */ into test_ignore_dupkey select * from test_ignore_dupkey1;
ERROR: duplicate key value violates unique constraint
"test_ignore_dupkey_ui1_102045"
DETAIL: Key (key1, key2)=(1, 1) already exists.
CONTEXT: COPY test_ignore_dupkey_102045, line 2
lightdb@postgres=# SELECT create_distributed_table('test_ignore_dupkey1',
'key1');
NOTICE: Copying data from local table...
NOTICE: copying the data has completed
DETAIL: The local data in the table is no longer visible, but is still on disk.
HINT: To remove the local data, run: SELECT
truncate_local_data_after_distributing_table($$public.test_ignore_dupkey1$$)
```

```
create_distributed_table
```

```
(1 row)
```

```
lightdb@postgres=# insert/*+IGNORE_ROW_ON_DUPKEY_INDEX(test_ignore_dupkey(key1,
key3)) */ into test_ignore_dupkey select * from test_ignore_dupkey1;
```

```
INSERT 0 1
```

```
lightdb@postgres=# select * from test_ignore_dupkey;
```

```
key1 | key2 | key3 | key4
```

```
-----+-----+-----+-----
 1 | 1 | 1 | 1
```

```
(1 row)
```

#### H. 27. 4. 1. 8. 10. 关于将谓词下推到子查询中的提示

如果你的查询中有一个不可合并的视图或子查询(可能因为no\_merge提示), 该如何操作其他表的联结呢? 是应该创建一个大的视图结果并连接一次(no\_push\_pred); 还是将连接谓词推到视图中, 然后从另一个表中为每个驱动行重新创建视图结果集(push\_pred 并且必须是nestloop)。

“no\_push\_pred” 提示指示优化器不要将连接谓词推送到子查询中。

LightDB 目前还不支持将连接谓词推入视图。目前只是为了兼容oracle。 因此, 如果你使用 “no\_push\_pred” 提示, 它将始终有效。

下面的例子展示了 “no\_push\_pred” 提示的用法:

```
create table test_no_push_pred1 (key1 int primary key, key2 int);
create table test_no_push_pred2 (key1 int primary key, key2 int);
create table test_no_push_pred3 (key1 int primary key, key2 int);
```

```
set enable_hashjoin to off;
set enable_mergejoin to off;
```

```
lightdb@test_o=# EXPLAIN (COSTS false)
```

```
lightdb@test_o=# select /*+leading(a)*/ from test_no_push_pred1 a join (select
* from test_no_push_pred2 order by key1 limit 1) b on a.key1 =b.key2;
QUERY PLAN
```

```
-----
Nested Loop
  Join Filter: (a.key1 = test_no_push_pred2.key2)
  -> Seq Scan on test_no_push_pred1 a @"1t#1"
  -> Materialize
      -> Limit
          -> Index Scan using test_no_push_pred2_pkey on
test_no_push_pred2
(6 rows)
```

```
lightdb@test_o=# EXPLAIN (COSTS false)
```

```
lightdb@test_o=# select /*+leading(a) no_push_pred(b)*/ from test_no_push_pred1
a join (select * from test_no_push_pred2 order by key1 limit 1) b on a.key1
=b.key2;
```

```
QUERY PLAN
```

```

-----
Nested Loop
  Join Filter: (a.key1 = test_no_push_pred2.key2)
    -> Seq Scan on test_no_push_pred1 a @"1t#1"
    -> Materialize
        -> Limit
            -> Index Scan using test_no_push_pred2_pkey on
test_no_push_pred2
(6 rows)

lightdb@test_o=# EXPLAIN (COSTS false)
lightdb@test_o=# select /*+leading(a) no_push_pred(@qb)*/ * from
test_no_push_pred1 a join (select/*+ qb_name(qb)*/ * from test_no_push_pred2
order by key1 limit 1) b on a.key1 =b.key2;
                                QUERY PLAN
-----

Nested Loop
  Join Filter: (a.key1 = test_no_push_pred2.key2)
    -> Seq Scan on test_no_push_pred1 a @"1t#0"
    -> Materialize
        -> Limit
            -> Index Scan using test_no_push_pred2_pkey on
test_no_push_pred2 @qb
(6 rows)

lightdb@test_o=# EXPLAIN (COSTS false)
lightdb@test_o=# select /*+leading(a) */ * from test_no_push_pred1 a join
(select/*+ no_push_pred*/ * from test_no_push_pred2 order by key1 limit 1) b on
a.key1 =b.key2;
                                QUERY PLAN
-----

Nested Loop
  Join Filter: (a.key1 = test_no_push_pred2.key2)
    -> Seq Scan on test_no_push_pred1 a @"1t#1"
    -> Materialize
        -> Limit
            -> Index Scan using test_no_push_pred2_pkey on
test_no_push_pred2 @"1t#0"
(6 rows)

```

#### H. 27. 4. 1. 8. 11. 星型转换相关提示

星型转换是一种优化器转换，它避免了对星型模式中事实表的全表扫描（星型模式将数据分为事实表和维度表）。在事实表和维度表的连接中，星型转换可以避免对事实表的全表扫描。星型转换通过位图索引来获取事实上需要的行（由维度表过滤），从而提高性能。在某些情况下，查询在维度表的非关联列上有过滤条件。通过组合这些过滤条件可以显著减少数据库需要处理的来自事实表的数据集。

“no\_star\_transformation” 提示指示优化器不执行星型查询转换。

LightDB 目前还不支持星形变换。目前只是为了兼容oracle。因此如果你正确使用了“no\_star\_transformation”提示，它总是会起效。

下面的例子展示了 “no\_star\_transformation” 提示的用法：

```

CREATE TABLE times (
    time_id NUMBER PRIMARY KEY,
    calendar_quarter_desc VARCHAR2(20)
);

CREATE TABLE customers (
    cust_id NUMBER PRIMARY KEY,
    cust_city VARCHAR2(20),
    cust_state_province VARCHAR2(20)
);

CREATE TABLE channels (
    channel_id NUMBER PRIMARY KEY,
    channel_class VARCHAR2(20),
    channel_desc VARCHAR2(20)
);

CREATE TABLE sales (
    time_id NUMBER REFERENCES times(time_id),
    cust_id NUMBER REFERENCES customers(cust_id),
    channel_id NUMBER REFERENCES channels(channel_id),
    amount_sold NUMBER
);

lightdb@postgres=# EXPLAIN (COSTS false) SELECT/*+no_star_transformation*/
ch.channel_class, c.cust_city, t.calendar_quarter_desc,
    SUM(s.amount_sold) sales_amount
FROM sales s, times t, customers c, channels ch
WHERE s.time_id = t.time_id
AND s.cust_id = c.cust_id
AND s.channel_id = ch.channel_id
AND c.cust_state_province = 'CA'
AND ch.channel_desc in ('Internet','Catalog')
AND t.calendar_quarter_desc IN ('1999-Q1','1999-Q2')
GROUP BY ch.channel_class, c.cust_city, t.calendar_quarter_desc;
LOG:  lt_hint_plan:
used hint:
no_star_transformation

```

not used hint:  
duplication hint:  
error hint:

QUERY PLAN

```

-----
GroupAggregate
  Group Key: ch.channel_class, c.cust_city, t.calendar_quarter_desc
  -> Sort
    Sort Key: ch.channel_class, c.cust_city, t.calendar_quarter_desc
    -> Nested Loop
      -> Nested Loop
        -> Hash Join
          Hash Cond: (s.cust_id = c.cust_id)
          -> Seq Scan on sales s @"lt#0"
          -> Hash
            -> Seq Scan on customers c @"lt#0"
              Filter: ((cust_state_province)::text
= 'CA'::text)
            -> Index Scan using times_pkey on times t @"lt#0"
              Index Cond: (time_id = s.time_id)
              Filter: ((calendar_quarter_desc)::text = ANY ('{1
999-Q1,1999-Q2}'::text[]))
            -> Index Scan using channels_pkey on channels ch @"lt#0"
              Index Cond: (channel_id = s.channel_id)
              Filter: ((channel_desc)::text = ANY ('{Internet,Catalog
}'::text[]))
(18 rows)

```

```

lightdb@postgres=# EXPLAIN (COSTS false)
select/*+no_star_transformation(@qb)*/ * from (SELECT/*+qb_name(qb)*/
ch.channel_class, c.cust_city, t.calendar_quarter_desc,
SUM(s.amount_sold) sales_amount
FROM sales s, times t, customers c, channels ch
WHERE s.time_id = t.time_id
AND s.cust_id = c.cust_id
AND s.channel_id = ch.channel_id
AND c.cust_state_province = 'CA'
AND ch.channel_desc in ('Internet','Catalog')
AND t.calendar_quarter_desc IN ('1999-Q1','1999-Q2')
GROUP BY ch.channel_class, c.cust_city, t.calendar_quarter_desc);

```

LOG: lt\_hint\_plan:  
used hint:  
no\_star\_transformation(@qb)  
not used hint:  
duplication hint:  
error hint:

QUERY PLAN

```

-----
GroupAggregate
  Group Key: ch.channel_class, c.cust_city, t.calendar_quarter_desc
  -> Sort
    Sort Key: ch.channel_class, c.cust_city, t.calendar_quarter_desc

```

```

-> Nested Loop
    -> Nested Loop
        -> Hash Join
            Hash Cond: (s.cust_id = c.cust_id)
            -> Seq Scan on sales s @qb
            -> Hash
                -> Seq Scan on customers c @qb
                    Filter: ((cust_state_province)::text
= 'CA'::text)
        -> Index Scan using times_pkey on times t @qb
            Index Cond: (time_id = s.time_id)
            Filter: ((calendar_quarter_desc)::text = ANY ('{1
999-Q1,1999-Q2}'::text[]))
    -> Index Scan using channels_pkey on channels ch @qb
        Index Cond: (channel_id = s.channel_id)
        Filter: ((channel_desc)::text = ANY ('{Internet,Catalog
}'::text[]))
(18 rows)

```

#### H. 27. 4. 1. 8. 12. 与direct-path插入相关的优化器提示

在使用 direct-path 插入时，表中的空闲空间不会被重用，数据库将插入的数据添加到表中已有的数据之后，数据被直接写入数据文件，不经过缓冲区缓存，同时完整性约束也会被忽略。直接路径插入的性能明显优于传统插入。

“append” 优化器提示指示优化器对insert select模式的语句使用 direct-path 插入。

LightDB 目前还不支持 direct-path 插入。支持这个hint目前只是为了兼容oracle。所以如果你使用“append”提示，它将不会生效。

下面的例子展示了“append”提示的用法：

```

create table test_append(key1 int, key2 int, key3 int, key4 int);
create table test_append1(key1 int, key2 int, key3 int, key4 int);

insert into test_append1 values(1,2,3,4);
insert into test_append1 values(11,21,31,41);
insert into test_append1 values(111,211,311,411);

lightdb@test_oracle_pg_hint_plan=# EXPLAIN (COSTS FALSE) insert/*+append */ into
test_append select 1, 1, 1, 2 from dual;
LOG:  lt_hint_plan:
used hint:
not used hint:
append
duplication hint:
error hint:

QUERY PLAN
-----
Insert on test_append @"lt#1"
-> Result
(2 rows)

lightdb@test_oracle_pg_hint_plan=# EXPLAIN (COSTS FALSE) insert/*+append */ into
test_append select * from test_append1;
LOG:  lt_hint_plan:
used hint:

```

```
not used hint:
append
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Insert on test_append @"lt#1"
-> Seq Scan on test_append1
(2 rows)
```

### H. 27. 4. 2. lt\_hint\_plan的GUC参数

下面的GUC参数会影响lt\_hint\_plan的行为。

Table H. 13. lt\_hint\_plan的GUC参数

参数名称	描述	默认值
lt_hint_plan.enable_hint_plan	启用lt_hint_plan。	on
lt_hint_plan.enable_table_hint	启用按表提示，true或false。	off
lt_hint_plan.plan_error_log_level	指定提示解析错误的日志级别。有效值为error、warning、notice、info、log、debug。	INFO
lt_hint_plan.debug_print_detail	控制调试打印和详细程度。有效值为off、on、detailed和verbose。	off
lt_hint_plan.debug_print_level	指定调试打印的消息级别。有效值为error、warning、notice、info、log、debug。	INFO
lt_hint_plan.show_debug_name	在调试打印中显示qb名称信息。	on

## H. 27. 5. 提示中的详细信息

### H. 27. 5. 1. 语法和放置

对于全局级别的提示，lt\_hint\_plan仅读取sql头部的第一个块注释（即第一个字符之前）。

```
lightdb@postgres=# /*+ indexscan(b)*/ explain select * from ltbench_accounts b
where b.aid>10;
```

QUERY PLAN

```
-----
Index Scan using ltbench_accounts_pkey on ltbench_accounts b
(cost=0.29..3666.12
rows=99990 width=97)
Index Cond: (aid > 10)
(2 rows)
```

```
lightdb@postgres=#
```

对于语句级别的提示, `lt_hint_plan`仅从`select`、`update`、`delete`、`insert`关键字后的第一个块注释中读取提示。提示仅在当前查询块中起作用, 而不使用`qb_name`。在以下示例中, `HashJoin(a b)`和`SeqScan(a)`被解析为提示, 但`IndexScan(a)`和`IndexScan(b)`不被解析为提示。

```
lightdb@postgres=# /*+
lightdb@postgres*#   HashJoin(a b)
lightdb@postgres*# */
lightdb@postgres=# /*+ IndexScan(a) */
lightdb@postgres=# EXPLAIN SELECT /*+ SeqScan(a) */ *
lightdb@postgres=#   FROM /*+ IndexScan(b) */ltbench_branches b
lightdb@postgres=#   JOIN ltbench_accounts a ON b.bid = a.bid
lightdb@postgres=#   ORDER BY a.aid;
                                QUERY PLAN
-----
Sort  (cost=24236.84..24486.84 rows=100000 width=461)
Sort Key: a.aid
-> Hash Join  (cost=1.02..4016.02 rows=100000 width=461)
    Hash Cond: (a.bid = b.bid)
    -> Seq Scan on ltbench_accounts a  (cost=0.00..2640.00 rows=100000
    width=
97)
        -> Hash  (cost=1.01..1.01 rows=1 width=364)
            -> Seq Scan on ltbench_branches b  (cost=0.00..1.01 rows=1 width=36
4)
(7 rows)

lightdb@postgres=#
```

## H. 27. 5. 2. 与PL/pgSQL一起使用

`lt_hint_plan`在PL/pgSQL脚本中对查询执行有一些限制。

- 提示仅对以下类型的查询起作用。
  - 返回一行结果的查询。(SELECT、INSERT、UPDATE和DELETE)
  - 返回多行结果的查询。(RETURN QUERY)
  - 动态 SQL 语句。(EXECUTE)
  - 打开光标。(OPEN)
  - 遍历查询结果 (FOR)
- 提示注释必须放在查询的第一个单词之后, 如下所示, 因为前面的注释不会作为查询的一部分发送。

```
lightdb@postgres=# CREATE FUNCTION hints_func(integer) RETURNS integer AS $$
lightdb@postgres$# DECLARE
lightdb@postgres$#   id integer;
lightdb@postgres$#   cnt integer;
lightdb@postgres$# BEGIN
lightdb@postgres$#   SELECT /*+ NoIndexScan(a) */ aid
lightdb@postgres$#     INTO id FROM ltbench_accounts a WHERE aid = $1;
lightdb@postgres$#   SELECT /*+ SeqScan(a) */ count(*)
lightdb@postgres$#     INTO cnt FROM ltbench_accounts a;
```



```
lightdb@postgres$# RETURN id + cnt;
lightdb@postgres$# END;
lightdb@postgres$# $$ LANGUAGE plpgsql;
```

### H. 27. 5. 3. 对象名称中的大小写

与 LightDB 处理对象名称的方式相似，`lt_hint_plan` 将把对象名称转换为小写。因此，在提示中，对象名称 `TBL` 对应的表可以是数据库中的 `TBL`、`tbl` 或 `Tbl`。使用引号时，`lt_hint_plan` 将不会将对象名称转换为小写。因此，在提示中，使用对象名称 `"TBL"` 只能匹配数据库中的 `"TBL"`，而不能匹配任何未加引号的表名，如 `TBL`、`tbl` 或 `Tbl`。

### H. 27. 5. 4. 在对象名称中转义特殊字符

如果对象名称包含括号、双引号和空格，则应该用双引号括起来作为提示参数。转义规则与 LightDB 相同。

### H. 27. 5. 5. 区分表的多个实例

如果存在别名，`lt_hint_plan` 将通过别名识别目标对象。这个行为可以用来指定在同一个表的多个实例中指向一个特定实例。

```
lightdb@postgres=# /*+ HashJoin(t1 t1) */
lightdb@postgres=# EXPLAIN SELECT * FROM s1.t1
lightdb@postgres=# JOIN public.t1 ON (s1.t1.id=public.t1.id);
INFO: hint syntax error at or near "HashJoin(t1 t1)"
DETAIL: Relation name "t1" is ambiguous.
...
lightdb@postgres=# /*+ HashJoin(pt st) */
lightdb@postgres=# EXPLAIN SELECT * FROM s1.t1 st
lightdb@postgres=# JOIN public.t1 pt ON (st.id=pt.id);
QUERY PLAN
-----
Hash Join (cost=64.00..1112.00 rows=28800 width=8)
Hash Cond: (st.id = pt.id)
-> Seq Scan on t1 st (cost=0.00..34.00 rows=2400 width=4)
-> Hash (cost=34.00..34.00 rows=2400 width=4)
    -> Seq Scan on t1 pt (cost=0.00..34.00 rows=2400 width=4)
```

### H. 27. 5. 6. 视图或规则的基础表

提示适用于视图本身，并且如果对象名称与在视图上扩展查询中的对象名称匹配，则可以影响其中的查询，当作为全局级别提示时。但是对于 `stmt` 级别的提示，需要使用 `qb_name` 提示来操作视图中的表。

```
create view test_v as select/*+qb_name(qb)*/ * from t1 where id >10;
EXPLAIN (COSTS false) select/*+indexscan(t1)*/ * from test_v;
LOG: lt_hint_plan:
used hint:
not used hint:
IndexScan(t1@lt#0)
duplication hint:
error hint:

QUERY PLAN
-----
Seq Scan on t1 @qb
Filter: (id > 10)
```

(2 rows)

```
EXPLAIN (COSTS false) select /*+indexscan(t1@qb)*/ * from test_v;
LOG:  lt_hint_plan:
used hint:
IndexScan(t1@qb)
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Index Scan using t1_pkey on t1 @qb
Index Cond: (id > 10)
(2 rows)
```

```
/*+indexscan(t1)*/
EXPLAIN (COSTS false) select * from test_v;
LOG:  lt_hint_plan:
used hint:
IndexScan(t1@lt#-1)
not used hint:
duplication hint:
error hint:
```

QUERY PLAN

```
-----
Index Scan using t1_pkey on t1 @qb
Index Cond: (id > 10)
(2 rows)
```

### H. 27. 5. 7. 继承表

提示只能指向继承表的父表，提示会影响整个继承关系。同时指向子表的提示并没有起作用。

### H. 27. 5. 8. 多语句提示

对于全局级别的提示，一个多语句只能有一个提示注释，提示会影响多语句中的所有单个语句。请注意，psql 交互界面上看似的多语句实际上是一系列单个语句，因此提示仅影响紧随其后的语句。对于 stmt 级别的提示，与单个语句相同。

### H. 27. 5. 9. VALUES 表达式

FROM 子句中的 VALUES 表达式在内部命名为 \*VALUES\*，因此如果它是查询中唯一的 VALUES，则可以进行提示。在查询中有两个或更多 VALUES 表达式似乎可以通过其 explain 结果进行区分。但实际上这只是一种表面上的区别，它们并不能区分。

```
lightdb@postgres=# /*+ MergeJoin(*VALUES*_1 *VALUES*) */
EXPLAIN SELECT * FROM (VALUES (1, 1), (2, 2)) v (a, b)
JOIN (VALUES (1, 5), (2, 8), (3, 4)) w (a, c) ON v.a = w.a;
INFO:  lt_hint_plan: hint syntax error at or near "MergeJoin(*VALUES*_1
*VALUES*)"
DETAIL:  Relation name "*VALUES*" is ambiguous.
```

QUERY PLAN

```
-----
Hash Join (cost=0.05..0.12 rows=2 width=16)
Hash Cond: ("*VALUES*_1".column1 = "*VALUES*".column1)
```

```

-> Values Scan on "*VALUES*_1" (cost=0.00..0.04 rows=3 width=8)
-> Hash (cost=0.03..0.03 rows=2 width=8)
    -> Values Scan on "*VALUES*" (cost=0.00..0.03 rows=2 width=8)

```

## H. 27.5.10. explain 中的成本值可能是错误的

lt\_hint\_plan 通过增加不需要的路径的成本 (cost += disable\_cost, disable\_cost=1.0e10) 来使执行计划选择所需的路径。

有时即使使用了提示, SQL 也不会按照提示执行, 因为所需的路径不可用, 比如在没有索引的表上使用 indexscan 提示。在这种情况下, explain 中的成本值不准确。

实际成本可以通过减去 disable\_cost 获得。

```

lightdb@postgres=# create table test(id int);
CREATE TABLE
lightdb@postgres=# explain select * from test where id=1;
          QUERY PLAN
-----
Seq Scan on test (cost=0.00..41.88 rows=13 width=4)
Filter: (id = 1)
(2 rows)

lightdb@postgres=# explain select /*+indexscan(test)*/ * from test where id=1;
          QUERY PLAN
-----
Seq Scan on test @"lt#0" (cost=10000000000.00..10000000041.88 rows=13 width=4)
Filter: (id = 1)
(2 rows)

lightdb@postgres=#

```

## H. 27.6. 子查询

在以下情况下, 子查询有时可以使用名称 "ANY\_subquery" 进行提示。

```

IN (SELECT ... {LIMIT | OFFSET ...} ...)
= ANY (SELECT ... {LIMIT | OFFSET ...} ...)
= SOME (SELECT ... {LIMIT | OFFSET ...} ...)

```

对于这些语法, 当规划器在计划包含子查询的表的联接时, 会为子查询分配名称, 因此可以使用隐式名称在此类联接上应用联接提示, 如下所示。

```

lightdb@postgres=# /*+HashJoin(a1 ANY_subquery)*/
lightdb@postgres=# EXPLAIN SELECT *
lightdb@postgres=#   FROM ltbench_accounts a1
lightdb@postgres=#   WHERE aid IN (SELECT bid FROM ltbench_accounts a2 LIMIT
10);
          QUERY PLAN
-----
Hash Semi Join (cost=0.49..2903.00 rows=1 width=97)
Hash Cond: (a1.aid = a2.bid)
-> Seq Scan on ltbench_accounts a1 (cost=0.00..2640.00 rows=100000
width=97)
-> Hash (cost=0.36..0.36 rows=10 width=4)
    -> Limit (cost=0.00..0.26 rows=10 width=4)

```

```

-> Seq Scan on ltbench_accounts a2 (cost=0.00..2640.00
rows=100000 width=4)

```

### H. 27. 6. 1. 使用 IndexOnlyScan 提示

当 IndexOnlyScan 提示指定的索引无法执行索引唯一扫描时，可能会在另一个索引上意外执行索引扫描。

### H. 27. 6. 2. NoIndexScan 的行为

NoIndexScan 提示涉及 NoIndexOnlyScan。

### H. 27. 6. 3. Parallel 提示和 UNION

仅当所有基础子查询都是并行安全的时，UNION 才能并行运行。相反，对任何一个子查询强制并行执行，都可以使可并行执行的 UNION 并行运行。同时，使用零个工作进程的并行提示会阻止扫描并行执行。

### H. 27. 6. 4. 使用 Set 提示设置 lt\_hint\_plan 参数

lt\_hint\_plan 参数会改变其自身的行为，因此某些参数可能不会按预期工作。

- 即使在调试日志中报告为“使用的提示”，也会忽略更改 enable\_hint、enable\_hint\_tables 的提示。
- 设置 debug\_print 和 message\_level 可以从目标查询处理的中间开始工作。

### H. 27. 6. 5. 对子查询使用扫描和联接提示

除非子查询只包含一个表，否则对引用的子查询（可合并）使用的扫描提示、联接提示（包括联接方法和联接顺序提示）和行提示将被忽略。 如果子查询只包含一个表，则对其应用的扫描路径提示或联接提示适用于子查询中的表。

```

lightdb@postgres=# create table test1(id int primary key, value int);
CREATE TABLE
lightdb@postgres=# create table test2(id int primary key, value int);
CREATE TABLE
lightdb@postgres=# EXPLAIN (COSTS false) select * from test1,(select * from
(select * from test2) x) y where test1.id=y.id;
          QUERY PLAN
-----
Hash Join
Hash Cond: (test1.id = test2.id)
-> Seq Scan on test1
-> Hash
      -> Seq Scan on test2
(5 rows)

lightdb@postgres=# EXPLAIN (COSTS false) select /*+indexscan(y)*/ from test1,
(select * from (select * from test2) x) y where test1.id=y.id;
          QUERY PLAN
-----
Hash Join
Hash Cond: (test2.id = test1.id)
-> Index Scan using test2_pkey on test2 @"lt#2"
-> Hash
      -> Seq Scan on test1 @"lt#0"
(5 rows)

```

```
lightdb@postgres=#
```

## H. 27. 7. 与 canopy 兼容

现在 `lt_hint_plan` 与 `canopy` 兼容，可以用于控制涉及分布式表和引用表的执行计划。

### H. 27. 7. 1. 提示状态

要查看数据节点上提示的使用状态，首先需要在数据节点上打开 `debug_print`，然后设置 `canopy.log_remote_commands` 为 `on`，最后将 `canopy.worker_min_messages` 设置为 `'log'`，`client_min_messages` 设置为 `'log'` 或将 `lt_hint_plan.message_level` 设置为 `'notice'`。以下是示例：

```
lightdb@postgres=# CREATE TABLE t1 (id int PRIMARY KEY, val int);
CREATE TABLE
lightdb@postgres=# SELECT create_distributed_table('t1', 'id');
create_distributed_table
-----

(1 row)

lightdb@postgres=# set lt_hint_plan.message_level = 'notice';
SET
lightdb@postgres=# set canopy.log_remote_commands=on;
SET
lightdb@postgres=# explain select /*+indexscan(t1)*/ from t1 where id=10;
NOTICE: lt_hint_plan:
used hint:
not used hint:
IndexScan(t1@lt#0)
duplication hint:
error hint:

NOTICE: issuing BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;SELECT
 assign_distributed_transaction_id(0, 24, '2022-03-14 15:49:45.634614+08');
DETAIL: on server lightdb@192.168.247.128:6432 connectionId: 2
NOTICE: issuing SAVEPOINT canopy_explain_savepoint
DETAIL: on server lightdb@192.168.247.128:6432 connectionId: 2
NOTICE: issuing /*+ IndexScan(t1) */EXPLAIN (ANALYZE FALSE, VERBOSE FALSE,
 COSTS TRUE, BUFFERS FALSE, WAL FALSE, TIMING FALSE, SUMMARY FALSE, FORMAT TEXT)
 SELECT id, val FROM public.t1_102044 t1 WHERE (id OPERATOR(pg_catalog.=) 10)
DETAIL: on server lightdb@192.168.247.128:6432 connectionId: 2
NOTICE: lt_hint_plan:
used hint:
IndexScan(t1@lt#-1)
not used hint:
duplication hint:
error hint:
DETAIL: from 192.168.247.128:6432
NOTICE: issuing ROLLBACK TO SAVEPOINT canopy_explain_savepoint
DETAIL: on server lightdb@192.168.247.128:6432 connectionId: 2
NOTICE: issuing COMMIT
DETAIL: on server lightdb@192.168.247.128:6432 connectionId: 2
QUERY PLAN
-----
```

```

-----
Custom Scan (Canopy Adaptive) (cost=0.00..0.00 rows=0 width=0)
Task Count: 1
Tasks Shown: All
-> Task
      Node: host=192.168.247.128 port=6432 dbname=postgres
      -> Index Scan using t1_pkey_102044 on t1_102044 t1 (cost=0.15..2.17
rows
=1 width=8)
      Index Cond: (id = 10)
(7 rows)

lightdb@postgres=#

```

## H. 27. 7. 2. 限制

不能下推且不包含对外部查询的引用的公共表达式和子查询是通过将子查询作为解析树递归调用规划器函数来进行规划的。给出的计划是独立执行的，SQL 语句已经被拆分。因此，在外部查询和子查询上同时起作用的提示，以及作用于子查询的 `no_merge`、`semijoin`/`antijoin` 提示将没有效果。以下是示例：

```

lightdb@postgres=# CREATE TABLE t1 (id int PRIMARY KEY, val int);
CREATE TABLE
lightdb@postgres=# CREATE TABLE t2 (id int PRIMARY KEY, val int);
CREATE TABLE
lightdb@postgres=# SELECT create_distributed_table('t1', 'id');
create_distributed_table
-----

(1 row)

lightdb@postgres=# SELECT create_distributed_table('t2', 'id');
create_distributed_table
-----

(1 row)

lightdb@postgres=#
lightdb@postgres=# explain select /*+indexscan(t2@qb) hashjoin(t1 t2@qb)*/ from
t1, (select/*+qb_name(qb)*/ * from t2 where t2.id >11) x where t1.id=10;
INFO:  lt_hint_plan: disable hint after splitting sql , hint: "HashJoin"
DETAIL:  it is invalid because of split SQL, defined at "hashjoin(t1 t2@qb)"
NOTICE:  lt_hint_plan:
used hint:
IndexScan(t2@qb)
not used hint:
duplication hint:
error hint:

NOTICE:  lt_hint_plan:
used hint:
IndexScan(t2@qb)
HashJoin(t1@1t#0 t2@qb)
not used hint:
duplication hint:
error hint:

```

```

NOTICE:  issuing BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;SELECT
  assign_distributed_transaction_id(0, 36, '2022-03-14 16:06:20.313852+08');
DETAIL:  on server lightdb@192.168.247.128:6432 connectionId: 3
NOTICE:  issuing SAVEPOINT canopy_explain_savepoint
DETAIL:  on server lightdb@192.168.247.128:6432 connectionId: 3
NOTICE:  issuing /*+ IndexScan(t2) */EXPLAIN (ANALYZE FALSE, VERBOSE FALSE,
  COSTS TRUE, BUFFERS FALSE, WAL FALSE, TIMING FALSE, SUMMARY FALSE, FORMAT TEXT)
  SELECT id, val FROM public.t2_102136 t2 WHERE (id OPERATOR(pg_catalog.>) 11)
DETAIL:  on server lightdb@192.168.247.128:6432 connectionId: 3
NOTICE:  lt_hint_plan:
used hint:
IndexScan(t2@lt#-1)
not used hint:
duplication hint:
error hint:
DETAIL:  from 192.168.247.128:6432
NOTICE:  issuing ROLLBACK TO SAVEPOINT canopy_explain_savepoint
DETAIL:  on server lightdb@192.168.247.128:6432 connectionId: 3
NOTICE:  issuing SAVEPOINT canopy_explain_savepoint
DETAIL:  on server lightdb@192.168.247.128:6432 connectionId: 3
NOTICE:  issuing EXPLAIN (ANALYZE FALSE, VERBOSE FALSE, COSTS TRUE, BUFFERS
  FALSE, WAL FALSE, TIMING FALSE, SUMMARY FALSE, FORMAT TEXT) SELECT t1.id,
  t1.val, x.id, x.val FROM public.t1_102108 t1, (SELECT intermediate_result.id,
  intermediate_result.val FROM read_intermediate_result('11_1'::text,
  'binary'::canopy_copy_format) intermediate_result(id integer, val integer)) x
  WHERE (t1.id OPERATOR(pg_catalog.=) 10)
DETAIL:  on server lightdb@192.168.247.128:6432 connectionId: 3
NOTICE:  issuing ROLLBACK TO SAVEPOINT canopy_explain_savepoint
DETAIL:  on server lightdb@192.168.247.128:6432 connectionId: 3
NOTICE:  issuing COMMIT
DETAIL:  on server lightdb@192.168.247.128:6432 connectionId: 3
      QUERY PLAN

```

```

-----
Custom Scan (Canopy Adaptive) (cost=0.00..0.00 rows=0 width=0)
-> Distributed Subplan 11_1
    -> Custom Scan (Canopy Adaptive) (cost=0.00..0.00 rows=100000 width=8)
        Task Count: 32
        Tasks Shown: One of 32
        -> Task
            Node: host=192.168.247.128 port=6432 dbname=postgres
            -> Index Scan using t2_pkey_102136 on t2_102136 t2
                (cost=0.1
                5..24.33 rows=753 width=8)
                    Index Cond: (id > 11)
        Task Count: 1
        Tasks Shown: All
        -> Task
            Node: host=192.168.247.128 port=6432 dbname=postgres
            -> Nested Loop (cost=0.16..22.17 rows=1000 width=16)
                -> Index Scan using t1_pkey_102108 on t1_102108 t1 (cost=0.15..2.1
                7 rows=1 width=8)
                    Index Cond: (id = 10)
                -> Function Scan on read_intermediate_result intermediate_result (
                cost=0.00..10.00 rows=1000 width=8)
                    (17 rows)

```

lightdb@postgres=#

## H. 27. 8. 错误

在任何错误上, `lt_hint_plan` 都会停止解析, 并在大多数情况下使用已经解析的提示。只有有在解释 SQL 时才会报告错误。以下是典型的错误。

### H. 27. 8. 1. 语法错误

任何语法错误或错误的提示名称都会被报告为语法错误。这些错误将在服务器日志中报告, 消息级别由 `lt_hint_plan.message_level` 指定, 如果 `lt_hint_plan.debug_print` 处于开启状态及以上。以数字开头的关键字不会成为提示名称, 它不会被报告为语法错误, 但会被忽略。

### H. 27. 8. 2. 对象错误规范

对象错误规范会导致提示被默默地忽略。这种类型的错误将会在服务器日志中与语法错误相同的条件下报告为“未使用的提示”。

### H. 27. 8. 3. 冗余或冲突的提示

对于同一查询块, 当出现冗余提示时, 其中一个提示将被激活, 当出现互相冲突的提示时, 将没有提示被激活。对于不同的查询块, 当出现冗余的提示或者互相冲突的提示时, 只有子查询的提示将被激活。无论何时出现冗余的提示或者互相冲突的提示, 这种类型的错误将以与语法错误相同的条件在服务器日志中报告为“重复的提示”。如果 `qb_name` 提示冲突, 则两个提示都将被忽略。

### H. 27. 8. 4. 嵌套注释

对于全局级别的提示, 提示注释中不能包含另一个块注释。如果 `lt_hint_plan` 发现了这种情况, 与其他错误不同, 它将停止解析并放弃已经解析的所有提示。这种类型的错误将以与其他错误相同的方式报告。对于语句级别的提示, 提示注释中不能包含另一个块注释。它将忽略提示注释中的 `'/*'`。如果报告错误, 它只会影响当前查询块的提示。

```
lightdb@postgres=# explain select * from t1 ,(select * from t2 where id >10) as
tt where exists (select /*+ /*+seqscan(t1)*/ from t3 where id>1);;
INFO:  lt_hint_plan: hint syntax error at or near "+seqscan(t1)"
DETAIL:  Unrecognized hint keyword "+seqscan".
          QUERY PLAN
```

```
-----
Result  (cost=1.04..3.13 rows=3 width=532)
One-Time Filter: $0
InitPlan 1 (returns $0)
  -> Seq Scan on t3 @"lt#0"  (cost=0.00..1.04 rows=1 width=0)
      Filter: (id > 1)
-> Nested Loop  (cost=1.04..3.13 rows=3 width=532)
    -> Seq Scan on t2  (cost=0.00..1.04 rows=1 width=8)
        Filter: (id > 10)
    -> Seq Scan on t1 @"lt#1"  (cost=0.00..1.03 rows=3 width=524)
(9 rows)
```

```
lightdb@postgres=# explain select * from t1 ,(select * from t2 where id >10) as
tt where exists (select /*+ /*seqscan(t1)*/ from t3 where id>1);;
LOG:  lt_hint_plan:
used hint:
not used hint:
SeqScan(t1@lt#0)
```



duplication hint:  
error hint:

QUERY PLAN

```
-----
Result (cost=1.04..3.13 rows=3 width=532)
One-Time Filter: $0
InitPlan 1 (returns $0)
    -> Seq Scan on t3 @"lt#0" (cost=0.00..1.04 rows=1 width=0)
        Filter: (id > 1)
-> Nested Loop (cost=1.04..3.13 rows=3 width=532)
    -> Seq Scan on t2 (cost=0.00..1.04 rows=1 width=8)
        Filter: (id > 10)
    -> Seq Scan on t1 @"lt#1" (cost=0.00..1.03 rows=3 width=524)
(9 rows)
```

lightdb@postgres=#

## H. 27. 9. 功能限制

### H. 27. 9. 1. 规划器GUC参数的影响

规划器不会尝试考虑超过 `from_collapse_limit` 的 FROM 子句条目的连接顺序。因此，`lt_hint_plan` 不能像预期的那样影响连接顺序。

### H. 27. 9. 2. 提示尝试强制执行无法执行的计划

当无法执行强制执行计划时，规划器会选择任何可执行的计划。

- 使用嵌套循环进行 FULL OUTER JOIN。
- 使用没有在 `quals` 中使用的列的索引。
- 对于没有 `ctid` 条件的查询，使用 TID 扫描。

### H. 27. 9. 3. ECPG中的查询

ECPG 会删除作为嵌入式 SQLs 编写的查询中的注释，因此无法通过这些查询传递提示信息。唯一的例外是 EXECUTE 命令传递给定字符串未经修改。请在这种情况下考虑提示表。

### H. 27. 9. 4. 使用 `lt_stat_statements`

`lt_stat_statements` 生成查询 ID 时会忽略注释。因此，具有不同提示的相同查询将被归纳为同一查询。

## H. 27. 10. 提示列表

下面列出了可用的提示。

Table H. 14. 提示列表

组	格式	描述
扫描方法	<code>SeqScan(table)</code> <code>full(table)</code>	强制对表进行顺序扫描
	<code>TidScan(table)</code>	强制对表进行 TID 扫描。
	<code>IndexScan(table[ index...])</code> <code>index(table[ index...])</code>	强制对表进行索引扫描。如果指定了，则限制为指定的索引。

组	格式	描述
	IndexOnlyScan(table[ index...])	强制对表进行仅索引扫描。如果指定了，则限制为指定的索引。如果不支持仅索引扫描，则可能使用索引扫描。
	BitmapScan(table[ index...])	强制对表进行位图扫描。如果指定了，则限制为指定的索引。
	IndexScanRegexp(table[ POSIX Regexp...]) IndexOnlyScanRegexp(table[ POSIX Regexp...]) BitmapScanRegexp(table[ POSIX Regexp...])	强制对表进行索引扫描或仅索引扫描或位图扫描。限制为与指定的 POSIX 正则表达式模式匹配的索引。
	NoSeqScan(table)	强制不对表进行顺序扫描。
	NoTidScan(table)	强制不对表进行 TID 扫描。
	NoIndexScan(table)	强制不对表进行索引扫描和仅索引扫描。
	NoIndexOnlyScan(table)	强制不对表进行仅索引扫描。
	NoBitmapScan(table)	强制不对表进行位图扫描。
连接方法	NestLoop(table table[ table...]) use_nl(table table[ table...])	强制为由指定表组成的连接使用嵌套循环。
	HashJoin(table table[ table...]) use_hash(table table[ table...])	强制为由指定表组成的连接使用哈希连接。
	MergeJoin(table table[ table...]) use_merge(table table[ table...])	强制为由指定表组成的连接使用归并连接。
	NoNestLoop(table table[ table...]) no_use_nl(table table[ table...])	强制不对由指定表组成的连接使用嵌套循环。
	NoHashJoin(table table[ table...]) no_use_hash(table table[ table...])	强制不对由指定表组成的连接使用哈希连接。
	NoMergeJoin(table table[ table...]) no_use_merge(table table[ table...])	强制不对由指定表组成的连接使用归并连接。
连接方法 (Oracle 模式)	NestLoop(table [ table...]) use_nl(table[ table...])	强制为内部表指定的连接使用嵌套循环。
	HashJoin(table[ table...]) use_hash(table[ table...])	强制为内部表指定的连接使用哈希连接。

组	格式	描述
	MergeJoin(table[ table...]) use_merge(table[ table...])	强制为内部表指定的连接使用归并连接。
	NoNestLoop(table[ table...]) no_use_nl(table[ table...])	强制不对内部表指定的连接使用嵌套循环。
	NoHashJoin(table[ table...]) no_use_hash(table[ table...])	强制不对内部表指定的连接使用哈希连接。
	NoMergeJoin(table[ table...]) no_use_merge(table[ table...])	强制不对内部表指定的连接使用归并连接。
连接顺序	Leading(table table[ table...]) Ordered(table table[ table...])	强制按指定顺序和方向连接。
	Leading([join pair]) Ordered([join pair])	强制按指定顺序和方向连接。连接对是由括号括起来的表和/或其他连接对组成的一对， 可以形成嵌套结构。
	Ordered Leading	强制按 FROM 子句中出现的顺序连接表。
	Leading(table) Ordered(table)	强制使用指定的表作为连接执行计划的第一个表进行连接。
行号修正	Rows(table [ table...] correction) cardinality(table [ table... ] correction)	修正由指定表组成的连接或子查询结果的行号。可用的修正方法有绝对值 (#[n] 或 [n])、加法 (+[n])、减法 (-[n]) 和乘法 (*[n])。[n] 应该是 strtod() 可以读取的字符串。
并行查询配置	Parallel(table [# of workers] [soft hard])	强制或禁止指定表的并行执行。[# of workers] 是所需的并行工作进程数，其中零表示禁止并行执行。如果第三个参数是 soft，它只会更改 max_parallel_workers_per_gather，其他的交给规划器。hard (默认) 表示强制指定数量的工作进程。
	Parallel([# of workers] [soft hard])	强制或禁止当前查询块中的表的并行执行。[# of workers] 是所需的并行工作进程数，其中零表示禁止并行执行。如果第二个参数是 soft，它只会更改 max_parallel_workers_per_gather，其他的交给规划器。hard (默认) 表示强制指定数量的工作进程。
GUC	Set (GUC-param value)	在规划器运行时将 GUC 参数设置为指定的值。

组	格式	描述
	Opt_param(' GUC-param' value)	在规划器运行时将 GUC 参数 设置为指定的值。
查询块名称	qb_name(name)	设置查询块名称，然后可以 在外部查询块中使用此查询 块中的表提示。不能用于 GUC。
不合并	no_merge[(子查询名称)] no_merge(@查询块名称)	强制不提升子查询。优先级 高于连接方法和连接顺序。
分组方法	use_hash_aggregation	强制使用哈希聚合方法进行 分组。
	no_use_hash_aggregation	强制不使用哈希聚合方法进 行分组，而是使用排序方 法。
半连接（特殊连接）	semi join	强制将相关的EXISTS子查询 转换为半连接以访问指定 的表。
	hash_sj	强制将相关的EXISTS子查询 转换为哈希半连接以访问 指定的表。
	nl_sj	强制将相关的EXISTS子查询 转换为嵌套循环半连接以 访问指定的表。
	merge_sj	强制将相关的EXISTS子查询 转换为合并半连接以访问 指定的表。
	no_semi join	如果可能的话，强制不使用 半连接，而不是不拉升子 链接。
反连接（特殊连接）	hash_aj	强制将相关的NOT EXISTS子 查询转换为哈希反连接以 访问指定的表。
	nl_aj	强制将相关的NOT EXISTS子 查询转换为嵌套循环反连 接以访问指定的表。
	merge_aj	强制将相关的NOT EXISTS子 查询转换为合并反连接以 访问指定的表。
Swap_join_inputs	swap_join_input(table)	在哈希连接时强制使用表作 为外部表
有序谓词	ordered_predicates	强制优化器保留谓词评估 的顺序
展开	unnest	拉升子链接是一种RBO，因 此只支持语法
不展开	no_unnest	强制优化器不拉升子链接
Pq_distribute	pq_distribute(inner_table outer_distribution inner_distribution)	pq_distribute提示指示 优化器如何执行并行连 接。inner_table是连接的 内部表，outer_distribution 是外部表的分 布，inner_distribution

组	格式	描述
		是内部表的分布。目前仅支持“none broadcast”和“hash hash”，而且只有“none broadcast”可以使用。
No_push_subq	no_push_subq	强制优化器不下推带子链接的过滤条件，而在最后才使用此条件过滤数据
Push_subq	push_subq	强制优化器下推带子链接的过滤条件，尽可能早的使用此条件过滤数据 (LightDB 的默认行为)。
Ignore_row_on_dupkey_index	ignore_row_on_dupkey_index (indexname) ignore_row_on_dupkey_index (table_name (colname[, colname...]))	强制忽略违反指定列集或指定索引的唯一键冲突错误。
No_push_pred	no_push_pred[(sub query name)] no_push_pred(@queryblock_name)	指示优化器不要将连接谓词推入不能合并的子查询中
No_star_transformation	no_star_transformation	指示优化器不执行星型查询转换。
Append	append	指示优化器对insert select模式的语句使用 direct-path 插入

## H. 28. lt\_log\_long\_xact

这是一个 Pl/pgsql 函数，用于在 LightDB 中记录运行时间较长的事务，可以选择输出任何阻塞语句的详细信息，并取消/终止有问题的语句/事务。

如有需要，可以使用 cron 或类似的工具进行调度。

谨慎使用取消/终止功能，并参阅有关 LightDB 中某些用例的更好替代方案的说明。

### H. 28. 1. 使用方法

要简单记录长时间运行的事务，只需调用该函数，并传递开始记录的最短持续时间：

```
TEST=# SELECT pg_log_long_xact(' 1s');
NOTICE: long_xact pid: 4465 duration: 431.325747 ms user: glyn application:
  ltsql client: [local] statement: <IDLE> in transaction
NOTICE: long_xact pid: 16532 duration: 327.438302 ms user: glyn application:
  ltsql client: [local] statement: UPDATE balls SET description = 'TEST' WHERE id
  = 5;
pg_log_long_xact
-----
(0 rows)
```

要记录有关长时间运行的事务可能等待的锁的额外信息，请将第二个参数设置为 true（尽管您可以通过启用 log\_lock\_waits 来记录类似的信息）：

```
TEST=# SELECT pg_log_long_xact(' 1s', true);
```

```

NOTICE: long_xact pid: 4465 duration: 471.885373 ms user: glyn application:
  ltsql client: [local] statement: <IDLE> in transaction
NOTICE: long_xact pid: 16532 duration: 367.997928 ms user: glyn application:
  ltsql client: [local] statement: UPDATE balls SET description = 'TEST' WHERE id
  = 5;
NOTICE: long_xact waiter pid: 16532 blocker detail is; pid: 4465 duration:
  471.885373 ms relation: any (public.balls (RowExclusiveLock)) lock type:
  transaction id 311824482 user: glyn application: ltsql client: [local]
  statement: <IDLE> in transaction
pg_log_long_xact
-----
(0 rows)

```

要设置所使用的 `raise` 级别，请传递第三个参数，它可以是以下任何一种：“debug”、“log”、“info”、“notice”、“warning”或“text”，以作为文本结果集输出。（回过头来看，默认值可能应该是“log”）

```

TEST=# \t off
仅显示元组。
TEST=# SELECT pg_log_long_xact('1s', true, 'text');
long_xact pid: 4465 duration: 574.477076 ms user: glyn application: ltsql
client: [local] statement: <IDLE> in transaction
long_xact pid: 16532 duration: 470.589631 ms user: glyn application: ltsql
client: [local] statement: UPDATE balls SET description = 'TEST' WHERE id = 5;
long_xact waiter pid: 16532 blocker detail is; pid: 4465 duration: 574.477076
ms relation: any (public.balls (RowExclusiveLock)) lock type: transaction id
311824482 user: glyn application: ltsql client: [local] statement: <IDLE> in
transaction

```

要使其开始取消超过特定持续时间的语句，我们将持续时间作为第四个参数传递；每次运行时，将取消第一个超出此限制的事务，并优先处理阻塞语句。

（有一个 `lock_timeout` 参数，将中止等待时间超过指定毫秒数的任何语句，这要好得多。请注意，这里的区别是该函数将尝试中止阻塞事务而不是等待语句，或者如果没有锁定，则中止最长事务。）

```

TEST=# SELECT pg_log_long_xact('1s', true, 'text', '10 minutes');
long_xact pid: 4465 duration: 895.57988 ms user: glyn application: ltsql
client: [local] statement: <IDLE> in transaction
long_xact unable to cancel backend with pid: 4465
long_xact pid: 16532 duration: 791.692435 ms user: glyn application: ltsql
client: [local] statement: UPDATE balls SET description = 'TEST' WHERE id = 5;
long_xact waiter pid: 16532 blocker detail is; pid: 4465 duration: 895.57988 ms
relation: any (public.balls (RowExclusiveLock)) lock type: transaction id 31182
4482 user: glyn application: ltsql client: [local] statement: <IDLE> in
transaction

```

我们还可以将持续时间作为第五个参数提供，以终止具有超过此时间长度的事务的后端进程，取消优先级与上述相同：

```

TEST=# SELECT pg_log_long_xact('1s', true, 'text', '10 minutes', '15 minutes');
long_xact pid: 4465 duration: 1026.90736 ms user: glyn application: ltsql
client: [local] statement: <IDLE> in transaction
long_xact terminated backend with pid: 4465
long_xact pid: 16532 duration: 923.019915 ms user: glyn application: ltsql
client: [local] statement: UPDATE balls SET description = 'TEST' WHERE id = 5;

```

默认情况下，该函数仅取消/终止最长运行的阻塞者或最长运行的事务，我们可以通过传递一个额外的参数来告诉它取消所有长时间运行的事务中的语句/终止后端进程，从而变得更加强制性：

```
TEST=#\t off
TEST=# SELECT pg_log_long_xact('1s', true, 'text', '2 minutes', '3 minutes',
true);
long_xact pid: 19065 duration: 187.279089 ms user: glyn application: ltsql
client: [local] statement: <IDLE> in transaction
long_xact terminated backend with pid: 19065
long_xact pid: 16532 duration: 184.251569 ms user: glyn application: ltsql
client: [local] statement: UPDATE balls SET description = 'TEST' WHERE id = 5;
long_xact cancelled backend with pid: 16532
(4 rows)
```

## H. 29. lt\_ope

### H. 29.1. What is lt\_ope?

Order-Preserving Encryption是一种加密方法，用于保留数据的顺序，可以直接对加密数据进行比较操作，而无需解密操作数。

lt\_ope支持三种数据类型的保序加密：bigint、double precision和text。text类型支持的最大明文长度为128字节，目前支持的字符类型包括ASCII中的0至127共128个字符。三种明文加密后得到的密文类型为ope\_type。

密文类型ope\_type支持以下操作：>、>=、<、<=、=和!=。同时，由文本类型明文加密的ope\_type类型密文支持like 'xxx%'操作。

密文类型ope\_type还支持order by、between...and、IN、NOT IN和BTREE index。

### H. 29.2. function usage

#### H. 29.2.1. bigint类型保序加密

bigint类型加密接口：ope\_bigint\_encrypt(plaintext bigint) return ope\_type。

```
SELECT ope_bigint_encrypt(12345678);
           ope_bigint_encrypt
-----
663667376838DD03DDFF2F57DE01DE028841DF02DF43DF63DF65DF68
```

#### H. 29.2.2. double precision类型保序加密

double precision类型加密接口：ope\_double\_encrypt(plaintext double precision) return ope\_type。

```
SELECT ope_double_encrypt(1234.5678);
           ope_double_encrypt
-----
663667376838DC02DDFF5227DE01DE026961DF02DDF4DF43DF63DF65DF68DDFDDDFDDDFDDDFDDA8CDDFDDDFD
```

### H. 29. 2. 3. text类型保序加密

text类型加密接口: `ope_text_encrypt(plaintext text) return ope_type`。

```
SELECT ope_text_encrypt('123456789@qq.com');
           ope_text_encrypt
```

```
-----
6535643463336232DDFF52CBDE01DE02FD63DF02DF43DF63DF65DF68DF6CDFE2FF82FF82DDF4FDDFDA8CFF01FDFC
```

### H. 29. 2. 4. OPE解密

OPE解密接口: `ope_decrypt(ciphertext ope_type) return text`。

### H. 29. 2. 5. OPE示例

以下是文本类型保序加密的示例:

```
-- The info field in the test_ope table is plaintext of text type, and the
   opedata field is order-preserving encrypted ciphertext of type ope_type:
create table test_ope(id int, info text, opedata ope_type, crt_time timestamp
not null);
CREATE TABLE
```

```
create index opedata_index on test_ope USING BTREE (opedata);
CREATE INDEX
```

```
select * from test_ope ;
```

id	info	opedata	crt_time
1	27025cc9acd114a4f5192930ebc6ffffb	6535643463336232DE01067CDF65DDFD8BE9DE01DF43FDDFFDFFDF6CFDDAFDDFFDE2DDFFDFFDFDF028584FDDADF02FD	2022-06-16 11:10:37.170803
2	cc97ddaf161894af7d5fbbe4ae31d0a1	6535643463336232FDDFE67CFDDDFDF6C8BE9DF65FDE2FDE2FDDAFDEBDDFFDF63DDFFDF68DF6CDF028584FDDAFDEBDF	2022-06-16 12:10:37.170955
3	433dc866e329799e7d134992b13960fa	6535643463336232DF021A7DDE02DE028BE9FDE2FDDDFDF68DF63DF63FDE3DE02DE01DF6CDF65DF6C8584DF6CFDE3DF	2022-06-16 13:10:37.170999
4	dff20cb13ce5d0365a6eb7625167e158	6535643463336232FDE2557DFDEBFDEB8BE9DE01DDFDFDDFFDDBDDFFDE02FDDFFDE3DF43FDE2DDFD8584DE02DF63DF	2022-06-16 14:10:37.171109
5	c07b52a7943c3ee93aea8b05c04edbf3	6535643463336232FDDFC17DDDFDFF658BE9FDDBDFF43DE01FDDADF65DF6CDF02DE02FDDFFDE02FDE38584FDE3DF6CDE	2022-06-16 15:10:37.171169
6	cf87a3591009b652ba01dd3eb6a50d42	6535643463336232FDDF047EFDEBDF688BE9DF65FDDADE02DF43DF6CDDFFDFFDFFDFFDF6CFDDBDF638584DF43DE01FD	2022-06-16 16:10:37.171208
7	c2a1654f7e4887339e552d9d70666c1c	6535643463336232FDDF157EDE01FDDA8BE9DDFFDF63DF43DF02FDEBDF65FDE3DF02DF68DF68DF658584DE02DE02FD	2022-06-16 17:10:37.171225



```

8 | 127306338277eef17554ce089d28eb9d |
6535643463336232DDFF257EDE01DF658BE9DE02DDFFDF63DE02DE02DF68DE01DF65DF65FDE3FDE38584FDEBDDFFDF
| 2022-06-16 18:10:37.171241
9 | 4b7c57af3114003805bc0dd861ac7996 |
6535643463336232DF02387EFDDDBDF658BE9FDDDF43DF65FDDAFDEBDE02DDFFDDFFDF02DDFFDDDFD8584DE02DF68DD
| 2022-06-16 19:10:37.171261
10 | 8ed90bd708f01094507636b80d9ef465 |
6535643463336232DF686F7EFDE3FDE28BE9DF6CDDFDFDDBFDE2DF65DDFFDF68FDEBDDFFDDFFDDF8584DF6CDF02DF
| 2022-06-16 20:10:37.171317

```

```
select * from test_ope order by opedata;
```

```

      id |          info          |
      opedata
      crt_time
-----+-----
+-----+-----
8 | 127306338277eef17554ce089d28eb9d |
6535643463336232DDFF257EDE01DF658BE9DE02DDFFDF63DE02DE02DF68DE01DF65DF65FDE3FDE38584FDEBDDFFDF
| 2022-06-16 18:10:37.171241
1 | 27025cc9acd114a4f5192930ebc6ffffb |
6535643463336232DE01067CDF65DDFD8BE9DE01DF43FDDFFDDFFDF6CFDDAFDDFFDE2DDFFDDFFDF028584FDDADF02FD
| 2022-06-16 11:10:37.170803
3 | 433dc866e329799e7d134992b13960fa |
6535643463336232DF021A7DDE02DE028BE9FDE2FDDFDF68DF63DF63FDE3DE02DE01DF6CDF65DF6C8584DF6CFDE3DF
| 2022-06-16 13:10:37.170999
9 | 4b7c57af3114003805bc0dd861ac7996 |
6535643463336232DF02387EFDDDBDF658BE9FDDDF43DF65FDDAFDEBDE02DDFFDDFFDF02DDFFDDDFD8584DE02DF68DD
| 2022-06-16 19:10:37.171261
10 | 8ed90bd708f01094507636b80d9ef465 |
6535643463336232DF686F7EFDE3FDE28BE9DF6CDDFDFDDBFDE2DF65DDFFDF68FDEBDDFFDDFFDDF8584DF6CDF02DF
| 2022-06-16 20:10:37.171317
5 | c07b52a7943c3ee93aea8b05c04edbf3 |
6535643463336232FDDFC17DDDFDDF658BE9FDDDBDF43DE01FDDADF65DF6CDF02DE02FDDFDE02FDE38584FDE3DF6CDE
| 2022-06-16 15:10:37.171169
7 | c2a1654f7e4887339e552d9d70666c1c |
6535643463336232FDDF157EDE01FDDA8BE9DDFFDF63DF43DF02FDEBDF65FDE3DF02DF68DF68DF658584DE02DE02DF
| 2022-06-16 17:10:37.171225
2 | cc97ddaf161894af7d5fbbbe4ae31d0a1 |
6535643463336232FDDFE67CFDDFDF6C8BE9DF65FDE2FDE2FDDAFDEBDDFFDF63DDFFDF68DF6CDF028584FDDAFDEBDF
| 2022-06-16 12:10:37.170955
6 | cf87a3591009b652ba01dd3eb6a50d42 |
6535643463336232FDDF047EFDEBDF688BE9DF65FDDADE02DF43DF6CDDFFDDFFDDF6CFDDDBDF638584DF43DE01FD
| 2022-06-16 16:10:37.171208
4 | dff20cb13ce5d0365a6eb7625167e158 |
6535643463336232FDE2557DFDEBFDEB8BE9DE01DDFDFDDFFDDDBDDFFDE02FDDFFDFE3DF43FDE2DDFD8584DE02DF63DF
| 2022-06-16 14:10:37.171109

```

```
select * from test_ope where opedata > ope_text_encrypt('c') and opedata <=
ope_text_encrypt('d');
```

```

      id |          info          |
      opedata
      crt_time
-----+-----
+-----+-----

```

```

2 | cc97ddaf161894af7d5fbbe4ae31d0a1 |
6535643463336232FDDFE67CFDDDF6C8BE9DF65FDE2FDE2FDDAFDEBDDFFDF63DDFFDF68DF6CDF028584FDDAFDEBDF
| 2022-06-16 12:10:37.170955
5 | c07b52a7943c3ee93aea8b05c04edbf3 |
6535643463336232FDDFC17DDDFDDF658BE9FDDDBDF43DE01FDDADF65DF6CDF02DE02FDDDFDE02FDE38584FDE3DF6CDE
| 2022-06-16 15:10:37.171169
6 | cf87a3591009b652ba01dd3eb6a50d42 |
6535643463336232FDDF047EFDEBDF688BE9DF65FDDADE02DF43DF6CDDFFDDFDDDFDDF6CFDDBDF638584DF43DE01FD
| 2022-06-16 16:10:37.171208
7 | c2a1654f7e4887339e552d9d70666c1c |
6535643463336232FDDF157EDE01FDDA8BE9DDFFDF63DF43DF02FDEBDF65FDE3DF02DF68DF68DF658584DE02DE02DF
| 2022-06-16 17:10:37.171225

```

```
select * from test_ope where opedata like 'dff20%';
```

```

id | info | opedata | crt_time
-----+-----
4 | dff20cb13ce5d0365a6eb7625167e158 |
6535643463336232FDE2557DFDEBFDEB8BE9DE01DDFDFDDFFDDDBDDFFDE02FDDFFDE3DF43FDE2DDFD8584DE02DF63DF
| 2022-06-16 14:10:37.171109

```

```
select * from test_ope where opedata between ope_text_encrypt('c') and
ope_text_encrypt('d');
```

```

id | info | opedata | crt_time
-----+-----
2 | cc97ddaf161894af7d5fbbe4ae31d0a1 |
6535643463336232FDDFE67CFDDDF6C8BE9DF65FDE2FDE2FDDAFDEBDDFFDF63DDFFDF68DF6CDF028584FDDAFDEBDF
| 2022-06-16 12:10:37.170955
5 | c07b52a7943c3ee93aea8b05c04edbf3 |
6535643463336232FDDFC17DDDFDDF658BE9FDDDBDF43DE01FDDADF65DF6CDF02DE02FDDDFDE02FDE38584FDE3DF6CDE
| 2022-06-16 15:10:37.171169
6 | cf87a3591009b652ba01dd3eb6a50d42 |
6535643463336232FDDF047EFDEBDF688BE9DF65FDDADE02DF43DF6CDDFFDDFDDDFDDF6CFDDBDF638584DF43DE01FD
| 2022-06-16 16:10:37.171208
7 | c2a1654f7e4887339e552d9d70666c1c |
6535643463336232FDDF157EDE01FDDA8BE9DDFFDF63DF43DF02FDEBDF65FDE3DF02DF68DF68DF658584DE02DE02DF
| 2022-06-16 17:10:37.171225

```

```
select * from test_ope where opedata =
ope_text_encrypt('127306338277eef17554ce089d28eb9d');
```

```

id | info | opedata | crt_time
-----+-----
8 | 127306338277eef17554ce089d28eb9d |
6535643463336232DDFF257EDE01DF658BE9DE02DDFFDF63DE02DE02DF68DE01DF65DF65FDE3FDE38584FDEBDDFFDF
| 2022-06-16 18:10:37.171241

```

```

select ope_text_encrypt('cc97ddaf161894af7d5fbb4ae31d0a1') IN (select opedata
  from test_ope);
  ?column?
-----
  t

select ope_text_encrypt('cc97ddaf161894af7d5fbb4ae31d0a1') NOT IN (select
  opedata from test_ope);
  ?column?
-----
  f

select ope_decrypt(test_ope.opedata) from test_ope where id = 2;
  ope_decrypt
-----
cc97ddaf161894af7d5fbb4ae31d0a1

```

## H. 30. lt\_pathman

lt\_pathman 模块提供了优化的分区机制和管理分区的函数。

### H. 30. 1. 概述

分区 是将一个大表拆分成较小的部分。根据分区键，表中的每一行都会被移动到单独的分区中。

LightDB 提供本地分区功能：

```

CREATE TABLE test(id int4, value text) PARTITION BY RANGE(id);
CREATE TABLE test_1 PARTITION OF test FOR VALUES FROM (1) TO (10);
CREATE TABLE test_2 PARTITION OF test FOR VALUES FROM (10) TO (20);

```

这与传统方法并没有太大的不同；隐含的检查约束仍然存在，并且大部分限制仍然适用。

尽管这种方法具有灵活性，但它会强制查询规划器进行全面搜索，并且需要检查每个分区的约束条件，以确定是否应该将其包含在计划中。大量的分区可能会导致显著的规划开销。

lt\_pathman 模块具有管理分区的函数和优化的规划机制，利用了对分区结构的了解。它将分区配置存储在 pathman\_config 表中；每行包含一个分区表的条目（关系名称、分区列及其类型）。在初始化阶段，lt\_pathman 模块将一些子分区的信息缓存到共享内存中，以供以后的计划构建使用。在执行 SELECT 查询之前，lt\_pathman 遍历条件树，搜索类似以下表达式的内容：

```
VARIABLE OP CONST
```

这里，VARIABLE 是分区键，OP 是比较运算符（支持的运算符包括 =、<、<=、>、>=），CONST 是一个标量值。例如：

```
WHERE id = 150
```

基于分区类型和条件运算符，lt\_pathman 搜索相应的分区并构建计划。目前 lt\_pathman 支持两种分区方案：

- RANGE - 将行映射到使用分配给每个分区的分区键范围的分区。通过使用二分搜索算法实现优化；
- HASH - 使用通用哈希函数将行映射到分区。

更多有趣的功能即将推出。敬请期待！

## H. 30. 2. 功能亮点

- HASH 和 RANGE 分区方案；
- 按表达式和复合键进行分区；
- 自动和手动 分区管理；
- 支持整数、浮点数、日期和其他类型，包括域；
- 针对分区表的有效查询规划（JOIN、子查询等）；
- RuntimeAppend 和 RuntimeMergeAppend 自定义计划节点，以在运行时选择分区；
- PartitionFilter: INSERT 触发器的高效替代品；
- PartitionRouter 和 PartitionOverseer 用于跨分区 UPDATE 查询（而不是触发器）；
- 新插入数据的自动分区创建（仅适用于 RANGE 分区）；
- 改进的 COPY FROM 语句，可以直接将行插入到分区中；
- 用于分区创建事件处理的 自定义回调函数；
- 非阻塞的 并发表分区；
- FDW 支持（外部分区）；
- 各种 GUC 开关和可配置设置。
- 部分支持 声明式分区。

## H. 30. 3. 可用函数

### H. 30. 3. 1. 模块版本

```
pathman_version()
```

此函数以 MAJOR.MINOR.PATCH 格式返回已加载的 `lt_pathman` 模块的完整版本号。

### H. 30. 3. 2. 分区创建

```
create_hash_partitions(parent_relid REGCLASS,
                       expression TEXT,
                       partitions_count INTEGER,
                       partition_data BOOLEAN DEFAULT TRUE,
                       partition_names TEXT[] DEFAULT NULL,
                       tablespaces TEXT[] DEFAULT NULL)
```

通过分区表达式 `expr` 对 `relation` 进行 HASH 分区。参数 `partitions_count` 指定要创建的分区数；之后无法更改。如果 `partition_data` 为 `true`，则所有数据将自动从父表复制到分区。请注意，数据迁移可能需要一段时间才能完成，而且在事务提交之前表将被锁定。有关无锁迁移数据的方法，请参见 `partition_table_concurrently()`。如果预先设置了分区创建回调，则为每个分区调用该回调（请参见 `set_init_callback()`）。

```
create_range_partitions(parent_relid REGCLASS,
                       expression TEXT,
                       start_value ANYELEMENT,
                       p_interval ANYELEMENT,
                       p_count INTEGER DEFAULT NULL,
                       partition_data BOOLEAN DEFAULT TRUE)
```

```
create_range_partitions(parent_relid REGCLASS,
                       expression TEXT,
                       start_value ANYELEMENT,
                       p_interval INTERVAL,
```

```

        p_count          INTEGER DEFAULT NULL,
        partition_data   BOOLEAN DEFAULT TRUE)

create_range_partitions(parent_relid  REGCLASS,
                        expression    TEXT,
                        bounds        ANYARRAY,
                        partition_names TEXT[] DEFAULT NULL,
                        tablespaces   TEXT[] DEFAULT NULL,
                        partition_data BOOLEAN DEFAULT TRUE)

```

通过分区表达式 `expr` 对 `relation` 进行 RANGE 分区, `start_value` 参数指定初始值, `p_interval` 设置自动创建分区或使用 `append_range_partition()` 或 `prepend_range_partition()` 创建分区的默认范围 (如果为 NULL, 则自动分区创建功能将无法使用), `p_count` 是预制分区的数量 (如果未设置, 则 `lt_pathman` 会尝试根据表达式的值确定它)。 `bounds` 数组可以使用 `generate_range_bounds()` 构建。如果预先设置了分区创建回调, 则为每个分区调用该回调。

```

generate_range_bounds(p_start  ANYELEMENT,
                     p_interval INTERVAL,
                     p_count   INTEGER)

generate_range_bounds(p_start  ANYELEMENT,
                     p_interval ANYELEMENT,
                     p_count   INTEGER)

```

为 `create_range_partitions()` 构建 `bounds` 数组。

### H. 30. 3. 3. 数据迁移

```

partition_table_concurrently(relation  REGCLASS,
                             batch_size INTEGER DEFAULT 1000,
                             sleep_time FLOAT8 DEFAULT 1.0)

```

启动一个后台进程, 将数据从父表移动到分区。该进程利用短事务复制小批量数据 (每个事务最多可达 10K 行), 因此不会显著干扰用户的活动。如果进程无法锁定批次中的行, 则在下一次尝试之前睡眠 `sleep_time` 秒, 并最多尝试 60 次, 如果仍然无法锁定批次, 则退出。

```
stop_concurrent_part_task(relation REGCLASS)
```

停止执行并发分区任务的后台进程。注意: 进程将在完成重新定位当前批次后退出。

### H. 30. 3. 4. 触发器

不再需要触发器, 无论是对 INSERT 还是跨分区 UPDATE。但是, 支持用户提供的触发器:

- 每个插入的行 导致执行 相应分区 的 BEFORE/AFTER INSERT 触发器函数。
- 每个更新的行 导致执行 相应分区 的 BEFORE/AFTER UPDATE 触发器函数。
- 每个移动的行 (跨分区更新) 导致执行 相应分区 的 BEFORE UPDATE + BEFORE/AFTER DELETE + BEFORE/AFTER INSERT 触发器函数。

### H. 30. 3. 5. 分区创建后管理

```

replace_hash_partition(old_partition REGCLASS,
                       new_partition REGCLASS,
                       lock_parent   BOOLEAN DEFAULT TRUE)

```

用另一个表替换 HASH 分区表的指定分区。lock\_parent 参数将阻止任何 INSERT/UPDATE/ALTER TABLE 查询到父表。

```
split_range_partition(partition_relid REGCLASS,
                     split_value ANYELEMENT,
                     partition_name TEXT DEFAULT NULL,
                     tablespace TEXT DEFAULT NULL)
```

将 RANGE partition 按 split\_value 分成两个。如果有可用的新分区，则为新分区调用分区创建回调。

```
merge_range_partitions(variadic partitions REGCLASS[])
```

合并多个相邻的 RANGE 分区。分区自动按增加的边界排序；所有数据将累积在第一个分区中。

```
append_range_partition(parent_relid REGCLASS,
                       partition_name TEXT DEFAULT NULL,
                       tablespace TEXT DEFAULT NULL)
```

使用 pathman\_config.range\_interval 作为区间，追加新的 RANGE 分区。

```
prepend_range_partition(parent_relid REGCLASS,
                       partition_name TEXT DEFAULT NULL,
                       tablespace TEXT DEFAULT NULL)
```

使用 pathman\_config.range\_interval 作为区间，前置新的 RANGE 分区。

```
add_range_partition(parent_relid REGCLASS,
                   start_value ANYELEMENT,
                   end_value ANYELEMENT,
                   partition_name TEXT DEFAULT NULL,
                   tablespace TEXT DEFAULT NULL)
```

为具有指定范围边界的 relation 创建新的 RANGE 分区。如果 start\_value 或 end\_value 为 NULL，则相应的范围边界将为无限。

```
drop_range_partition(partition TEXT, delete_data BOOLEAN DEFAULT TRUE)
```

如果 delete\_data 为 true，则删除 RANGE 分区及其所有数据。

```
attach_range_partition(parent_relid REGCLASS,
                      partition_relid REGCLASS,
                      start_value ANYELEMENT,
                      end_value ANYELEMENT)
```

将分区附加到现有的 RANGE 分区关系中。所附加的表必须与父表具有完全相同的结构，包括被删除的列。如果设置了分区创建回调（参见 pathman\_config\_params），则将调用该回调。

```
detach_range_partition(partition_relid REGCLASS)
```

从现有的 RANGE 分区关系中分离分区。

```
disable_pathman_for(parent_relid REGCLASS)
```

永久禁用指定父表的 `lt_pathman` 分区机制，并在存在触发器的情况下删除插入触发器。所有分区和数据保持不变。

```
drop_partitions(parent_relid REGCLASS,
                delete_data BOOLEAN DEFAULT FALSE)
```

删除 `parent` 表的分区（包括外键和本地关系）。如果 `delete_data` 是 `false`，则首先将数据复制到父表中。默认值为 `false`。

要完全删除分区表及其所有分区，请使用传统的 `DROP TABLE relation CASCADE` 命令。但是，在你正在运行逻辑复制且复制应用程序工作进程执行了 `DROP`（例如通过复制表上的触发器）时，需要小心。`lt_pathman` 使用 `pathman_ddl_trigger` 事件触发器从 `pathman_config` 中删除有关已删除表的记录。默认情况下，此触发器不会在副本上触发，导致 `lt_pathman` 认为该表仍然存在，但实际上并不存在。如果是这种情况，请将该触发器配置为在副本上也触发：

```
ALTER EVENT TRIGGER pathman_ddl_trigger ENABLE ALWAYS;
```

物理复制没有这个问题，因为 DDL 和 `pathman_config` 表也会被复制；主数据库和从数据库的 LightDB 实例基本相同，将此触发器保留在 `ALWAYS` 模式下只会产生负面影响。

### H. 30. 3. 6. 附加参数

```
set_interval(relation REGCLASS, value ANYELEMENT)
设置时间间隔 (relation REGCLASS, value ANYELEMENT)
```

更新 `RANGE` 分区表的时间间隔。注意，时间间隔不能为负数，也不能为微不足道的时间间隔。对于数值类型，其值应大于零，对于 `TIMESTAMP` 类型，其时间间隔至少为 1 微秒，对于 `DATE` 类型，其时间间隔至少为 1 天。

```
set_enable_parent(relation REGCLASS, value BOOLEAN)
设置是否启用父级关系 (relation REGCLASS, value BOOLEAN)
```

包含/排除父级表的查询计划。在原始的 LightDB 计划器中，即使父级表为空，它也始终包括在查询计划中，这可能会导致额外的开销。如果您永远不会将父级表用作存储，则可以使用 `disable_parent()`。默认值取决于在 `create_range_partitions()` 函数中指定的 `partition_data` 参数。如果 `partition_data` 参数为 `true`，则所有数据已经迁移到分区并禁用了父级表。否则，它是启用的。

```
set_auto(relation REGCLASS, value BOOLEAN)
设置自动更新 (relation REGCLASS, value BOOLEAN)
```

启用/禁用自动分区传播（仅适用于 `RANGE` 分区）。默认情况下它是启用的。

```
set_init_callback(relation REGCLASS, callback REGPROC DEFAULT 0)
设置初始化回调函数 (relation REGCLASS, callback REGPROC DEFAULT 0)
```

设置分区创建回调函数，以便为每个已附加或创建的分区（`HASH` 和 `RANGE`）调用。如果回调函数标记为 `SECURITY INVOKER`，则它将以产生导致新分区创建的语句的用户的权限执行（例如，`INSERT INTO partitioned_table VALUES (-5)`）。回调函数必须具有以下签名：`part_init_callback(args JSONB) RETURNS VOID`。参数 `arg` 由几个字段组成，其存在取决于分区类型：

```
/* RANGE 分区表 abc (子表 abc_4) */
```

```
{
  "parent":      "abc",
  "parent_schema": "public",
  "parttype":    "2",
  "partition":   "abc_4",
  "partition_schema": "public",
  "range_max":   "401",
  "range_min":   "301"
}

/* HASH 分区表 abc (子表 abc_0) */
{
  "parent":      "abc",
  "parent_schema": "public",
  "parttype":    "1",
  "partition":   "abc_0",
  "partition_schema": "public"
}
```

```
set_set_spawn_using_bgw(relation REGCLASS, value BOOLEAN)
设置是否使用后台工作进程生成 SET (relation REGCLASS, value BOOLEAN)
```

当插入超出分区范围的新数据时，请使用 `SpawnPartitionsWorker` 在单独的事务中创建新分区。

## H. 30. 4. 视图和表

### H. 30. 4. 1. `pathman_config` — 主配置存储

```
CREATE TABLE IF NOT EXISTS pathman_config (
  partrel      REGCLASS NOT NULL PRIMARY KEY,
  expr         TEXT NOT NULL,
  parttype     INTEGER NOT NULL,
  range_interval TEXT,
  cooked_expr  TEXT);
```

此表存储分区表的列表。

### H. 30. 4. 2. `pathman_config_params` — 可选参数

```
CREATE TABLE IF NOT EXISTS pathman_config_params (
  partrel      REGCLASS NOT NULL PRIMARY KEY,
  enable_parent BOOLEAN NOT NULL DEFAULT TRUE,
  auto         BOOLEAN NOT NULL DEFAULT TRUE,
  init_callback TEXT DEFAULT NULL,
  spawn_using_bgw BOOLEAN NOT NULL DEFAULT FALSE);
```

此表存储覆盖标准行为的可选参数。

### H. 30. 4. 3. `pathman_concurrent_part_tasks` — 当前正在运行的分区工作进程

```
-- 帮助 SRF 函数
CREATE OR REPLACE FUNCTION show_concurrent_part_tasks()
RETURNS TABLE (
```



```

    userid    REGROLE,
    pid       INT,
    dbid      OID,
    relid     REGCLASS,
    processed INT,
    status    TEXT)
AS 'lt_pathman', 'show_concurrent_part_tasks_internal'
LANGUAGE C STRICT;

```

```

CREATE OR REPLACE VIEW pathman_concurrent_part_tasks
AS SELECT * FROM show_concurrent_part_tasks();

```

此视图列出所有当前正在运行的并发分区任务。

#### H. 30. 4. 4. pathman\_partition\_list — 所有现有分区的列表

```

-- 帮助 SRF 函数
CREATE OR REPLACE FUNCTION show_partition_list()
RETURNS TABLE (
    parent    REGCLASS,
    partition REGCLASS,
    parttype  INT4,
    expr      TEXT,
    range_min TEXT,
    range_max TEXT)
AS 'lt_pathman', 'show_partition_list_internal'
LANGUAGE C STRICT;

```

```

CREATE OR REPLACE VIEW pathman_partition_list
AS SELECT * FROM show_partition_list();

```

此视图列出所有现有的分区，以及它们的父级和范围边界（HASH 分区为 NULL）。

#### H. 30. 4. 5. pathman\_cache\_stats — 每个后端内存消耗

```

-- 帮助 SRF 函数
CREATE OR REPLACE FUNCTION @extschema@.show_cache_stats()
RETURNS TABLE (
    context  TEXT,
    size     INT8,
    used     INT8,
    entries  INT8)
AS 'lt_pathman', 'show_cache_stats_internal'
LANGUAGE C STRICT;

```

```

CREATE OR REPLACE VIEW @extschema@.pathman_cache_stats
AS SELECT * FROM @extschema@.show_cache_stats();

```

显示各种缓存的内存消耗。

### H. 30. 5. 声明式分区

lt\_pathman 分区表可以使用 ATTACH PARTITION、DETACH PARTITION 和 CREATE TABLE .. PARTITION OF 命令进行操作：

```

CREATE TABLE child1 (LIKE partitioned_table);

```

```

--- 附加新分区
ALTER TABLE partitioned_table ATTACH PARTITION child1
    FOR VALUES FROM ('2015-05-01') TO ('2015-06-01');

--- 分离分区
ALTER TABLE partitioned_table DETACH PARTITION child1;

-- 创建分区
CREATE TABLE child2 PARTITION OF partitioned_table
    FOR VALUES IN ('2015-05-01', '2015-06-01');

```

## H. 30.6. 自定义计划节点

lt\_pathman 提供了一些旨在减少执行时间的 自定义计划节点<sup>3</sup>，包括：

- RuntimeAppend （覆盖 Append 计划节点）
- RuntimeMergeAppend （覆盖 MergeAppend 计划节点）
- PartitionFilter （INSERT 触发器的替代方案）
- PartitionOverseer （实现跨分区 UPDATE）
- PartitionRouter （实现跨分区 UPDATE）

PartitionFilter 充当 INSERT 的子扫描的 代理节点，这意味着它可以将输出元组重定向到相应的分区：

```

EXPLAIN (COSTS OFF)
INSERT INTO partitioned_table
SELECT generate_series(1, 10), random();
          QUERY PLAN

```

```

-----
Insert on partitioned table
-> Custom Scan (PartitionFilter)
    -> Subquery Scan on "*SELECT*"
        -> Result
(4 rows)

```

PartitionOverseer 和 PartitionRouter 是与 PartitionFilter 结合使用的另一种 代理节点，用于启用跨分区 UPDATE（即当分区键的更新要求我们将行移动到另一个分区时）。由于此节点具有大量副作用（普通的 UPDATE 变慢；跨分区 UPDATE 被转换为 DELETE + INSERT），因此默认情况下禁用。要启用它，请参阅下面的 GUCs 列表。

```

EXPLAIN (COSTS OFF)
UPDATE partitioned_table
SET value = value + 1 WHERE value = 2;
          QUERY PLAN

```

```

-----
Custom Scan (PartitionOverseer)
-> Update on partitioned_table_2
    -> Custom Scan (PartitionFilter)
        -> Custom Scan (PartitionRouter)
            -> Seq Scan on partitioned_table_2
                Filter: (value = 2)
(6 rows)

```

RuntimeAppend 和 RuntimeMergeAppend 有很多共同点：它们在 WHERE 条件采用以下形式时非常有用：

<sup>3</sup> <https://wiki.postgresql.org/wiki/CustomScanAPI>

VARIABLE OP PARAM

由于参数的值直到执行阶段才知道，因此这种表达式不能在计划时间进行优化。可以通过将 WHERE 条件分析例程 嵌入到原始 Append 代码中来解决，从而使其只选择所需的扫描，而不是从计划的分区扫描中选择全部。这实际上归结为创建一个能够执行此类检查的自定义节点。

至少有几种情况可以证明这些节点的有用性：

```
/* 创建要分区的表 */
CREATE TABLE partitioned_table(id INT NOT NULL, payload REAL);

/* 插入一些数据 */
INSERT INTO partitioned_table
SELECT generate_series(1, 1000), random();

/* 执行分区 */
SELECT create_hash_partitions('partitioned_table', 'id', 100);

/* 创建普通表 */
CREATE TABLE some_table AS SELECT generate_series(1, 100) AS VAL;
```

- id = (select ... limit 1)

```
EXPLAIN (COSTS OFF, ANALYZE) SELECT * FROM partitioned_table
WHERE id = (SELECT * FROM some_table LIMIT 1);
```

QUERY PLAN

```
-----
Custom Scan (RuntimeAppend) (actual time=0.030..0.033 rows=1 loops=1)
  InitPlan 1 (returns $0)
    -> Limit (actual time=0.011..0.011 rows=1 loops=1)
          -> Seq Scan on some_table (actual time=0.010..0.010 rows=1 loops=1)
    -> Seq Scan on partitioned_table_70 partitioned_table (actual
time=0.004..0.006 rows=1 loops=1)
      Filter: (id = $0)
      Rows Removed by Filter: 9
Planning time: 1.131 ms
Execution time: 0.075 ms
(9 rows)
```

```
/* 禁用 RuntimeAppend 节点 */
SET lt_pathman.enable_runtimeappend = f;
```

```
EXPLAIN (COSTS OFF, ANALYZE) SELECT * FROM partitioned_table
WHERE id = (SELECT * FROM some_table LIMIT 1);
```

QUERY PLAN

```
-----
Append (actual time=0.196..0.274 rows=1 loops=1)
  InitPlan 1 (returns $0)
    -> Limit (actual time=0.005..0.005 rows=1 loops=1)
          -> Seq Scan on some_table (actual time=0.003..0.003 rows=1 loops=1)
    -> Seq Scan on partitioned_table_0 (actual time=0.014..0.014 rows=0 loops=1)
      Filter: (id = $0)
      Rows Removed by Filter: 6
    -> Seq Scan on partitioned_table_1 (actual time=0.003..0.003 rows=0 loops=1)
      Filter: (id = $0)
```

```

    Rows Removed by Filter: 5
    ... /* more plans follow */
    Planning time: 1.140 ms
    Execution time: 0.855 ms
    (306 rows)

```

- id = ANY (select ...)

```

EXPLAIN (COSTS OFF, ANALYZE) SELECT * FROM partitioned_table
WHERE id = any (SELECT * FROM some_table limit 4);
                                QUERY PLAN

```

```

-----
Nested Loop (actual time=0.025..0.060 rows=4 loops=1)
  -> Limit (actual time=0.009..0.011 rows=4 loops=1)
      -> Seq Scan on some_table (actual time=0.008..0.010 rows=4 loops=1)
  -> Custom Scan (RuntimeAppend) (actual time=0.002..0.004 rows=1 loops=4)
      -> Seq Scan on partitioned_table_70 partitioned_table (actual
time=0.001..0.001 rows=10 loops=1)
      -> Seq Scan on partitioned_table_26 partitioned_table (actual
time=0.002..0.003 rows=9 loops=1)
      -> Seq Scan on partitioned_table_27 partitioned_table (actual
time=0.001..0.002 rows=20 loops=1)
      -> Seq Scan on partitioned_table_63 partitioned_table (actual
time=0.001..0.002 rows=9 loops=1)
    Planning time: 0.771 ms
    Execution time: 0.101 ms
    (10 rows)

```

```

/* disable RuntimeAppend node */
SET lt_pathman.enable_runtimeappend = f;

```

```

EXPLAIN (COSTS OFF, ANALYZE) SELECT * FROM partitioned_table
WHERE id = any (SELECT * FROM some_table limit 4);
                                QUERY PLAN

```

```

-----
Nested Loop Semi Join (actual time=0.531..1.526 rows=4 loops=1)
  Join Filter: (partitioned_table.id = some_table.val)
  Rows Removed by Join Filter: 3990
  -> Append (actual time=0.190..0.470 rows=1000 loops=1)
      -> Seq Scan on partitioned_table (actual time=0.187..0.187 rows=0
loops=1)
      -> Seq Scan on partitioned_table_0 (actual time=0.002..0.004 rows=6
loops=1)
      -> Seq Scan on partitioned_table_1 (actual time=0.001..0.001 rows=5
loops=1)
      -> Seq Scan on partitioned_table_2 (actual time=0.002..0.004 rows=14
loops=1)
  ... /* 96 scans follow */
  -> Materialize (actual time=0.000..0.000 rows=4 loops=1000)
      -> Limit (actual time=0.005..0.006 rows=4 loops=1)
          -> Seq Scan on some_table (actual time=0.003..0.004 rows=4
loops=1)
    Planning time: 2.169 ms
    Execution time: 2.059 ms
    (110 rows)

```

- NestLoop 涉及到一个分区表，但由于已经在文档中偶尔显示过，因此被省略了。

## H. 30. 7. 示例

### H. 30. 7. 1. 常见提示

- 您可以使用名为tableoid的系统属性轻松添加包含底层分区名称的分区列：

```
SELECT tableoid::regclass AS partition, * FROM partitioned_table;
```

- 尽管在父表上的索引不是特别有用（因为它应该是空的），但它们作为分区索引的原型。对于父表上的每个索引，lt\_pathman都会在每个分区上创建一个类似的索引。
- 可以使用pathman\_concurrent\_part\_tasks视图列出所有正在运行的并发分区任务：

```
SELECT * FROM pathman_concurrent_part_tasks;
userid | pid  | dbid | relid | processed | status
-----+-----+-----+-----+-----+-----
dmitry | 7367 | 16384 | test  | 472000    | working
(1 row)
```

- 与drop\_range\_partition()结合使用的pathman\_partition\_list可以比使用传统的DROP TABLE更灵活地删除RANGE分区：

```
SELECT drop_range_partition(partition, false) /* move data to parent */
FROM pathman_partition_list
WHERE parent = 'part_test'::regclass AND range_min::int < 500;
NOTICE:  1 rows copied from part_test_11
NOTICE: 100 rows copied from part_test_1
NOTICE: 100 rows copied from part_test_2
drop_range_partition
-----
dummy_test_11
dummy_test_1
dummy_test_2
(3 rows)
```

- 您可以使用attach\_range\_partition()函数将外部表转换为分区。原本应该插入父表的行将被重定向到外部分区（通常会涉及PartitionFilter），但默认情况下，禁止向非由postgres\_fdw提供的分区插入行。只有超级用户被允许设置lt\_pathman.insert\_into\_fdw的GUC变量。

### H. 30. 7. 2. HASH分区

考虑HASH分区的示例。首先创建一个带有某些整数列的表：

```
CREATE TABLE items (
    id      SERIAL PRIMARY KEY,
    name    TEXT,
    code    BIGINT);

INSERT INTO items (id, name, code)
SELECT g, md5(g::text), random() * 100000
FROM generate_series(1, 100000) as g;
```

现在使用适当的参数运行create\_hash\_partitions()函数：

```
SELECT create_hash_partitions('items', 'id', 100);
```

这将创建新的分区并将数据从父表移动到分区。

这里是一个通过分区键执行过滤的查询示例：

```
SELECT * FROM items WHERE id = 1234;
 id | name | code
-----+-----+-----
 1234 | 81dc9bdb52d04dc20036dbd8313ed055 | 1855
(1 row)

EXPLAIN SELECT * FROM items WHERE id = 1234;
          QUERY PLAN
-----
Append (cost=0.28..8.29 rows=0 width=0)
-> Index Scan using items_34_pkey on items_34 (cost=0.28..8.29 rows=0
width=0)
    Index Cond: (id = 1234)
```

注意，Append节点仅包含一个与WHERE子句对应的子扫描。

重要提示：请注意，lt\_pathman从查询计划中排除了父表。

要访问父表，请使用ONLY修饰符：

```
EXPLAIN SELECT * FROM ONLY items;
          QUERY PLAN
-----
Seq Scan on items (cost=0.00..0.00 rows=1 width=45)
```

### H. 30. 7. 3. RANGE分区

考虑RANGE分区的示例。让我们创建一个包含一些虚拟日志的表：

```
CREATE TABLE journal (
  id SERIAL,
  dt TIMESTAMP NOT NULL,
  level INTEGER,
  msg TEXT);

-- 类似的索引也将为每个分区创建
CREATE INDEX ON journal(dt);

-- 生成一些数据
INSERT INTO journal (dt, level, msg)
SELECT g, random() * 6, md5(g::text)
FROM generate_series('2015-01-01'::date, '2015-12-31'::date, '1 minute') as g;
```

运行create\_range\_partitions()函数以创建分区，使得每个分区包含一天的数据：

```
SELECT create_range_partitions('journal', 'dt', '2015-01-01'::date, '1
day'::interval);
```

这将创建365个分区并将数据从父表移动到分区。

新分区会自动通过插入触发器进行添加，但也可以使用以下函数手动执行：

-- 添加指定范围的新分区

```
SELECT add_range_partition('journal', '2016-01-01'::date, '2016-01-07'::date);
```

-- 使用默认范围附加新分区

```
SELECT append_range_partition('journal');
```

第一个函数将创建具有指定范围的分区。第二个函数将创建具有默认间隔的分区并将其附加到分区列表中。还可以将现有表作为分区进行附加。例如，我们可能希望为一些过时的数据附加一个归档表（甚至是来自另一个服务器的外部表）：

```
CREATE FOREIGN TABLE journal_archive (
    id      INTEGER NOT NULL,
    dt      TIMESTAMP NOT NULL,
    level   INTEGER,
    msg     TEXT)
SERVER archive_server;
```

```
SELECT attach_range_partition('journal', 'journal_archive', '2014-01-01'::date,
'2015-01-01'::date);
```

重要提示：附加表的定义必须与现有分区化表的定义相匹配，包括已删除的列。

要合并相邻的分区，请使用merge\_range\_partitions()函数：

```
SELECT merge_range_partitions('journal_archive', 'journal_1');
```

要按值拆分分区，请使用split\_range\_partition()函数：

```
SELECT split_range_partition('journal_366', '2016-01-03'::date);
```

要分离分区，请使用detach\_range\_partition()函数：

```
SELECT detach_range_partition('journal_archive');
```

这里是一个通过分区键执行过滤的查询示例：

```
SELECT * FROM journal WHERE dt >= '2015-06-01' AND dt < '2015-06-03';
```

id	dt	level	msg
217441	2015-06-01 00:00:00	2	15053892d993ce19f580a128f87e3dbf
217442	2015-06-01 00:01:00	1	3a7c46f18a952d62ce5418ac2056010c
217443	2015-06-01 00:02:00	0	92c8de8f82faf0b139a3d99f2792311d
...			

(2880 rows)

```
EXPLAIN SELECT * FROM journal WHERE dt >= '2015-06-01' AND dt < '2015-06-03';
QUERY PLAN
```

```
Append (cost=0.00..58.80 rows=0 width=0)
-> Seq Scan on journal_152 (cost=0.00..29.40 rows=0 width=0)
-> Seq Scan on journal_153 (cost=0.00..29.40 rows=0 width=0)
(3 rows)
```

#### H. 30.7.4. 禁用lt\_pathman

有几个用户可访问的 GUC变量设计用于切换整个模块或特定自定义节点的开启和关闭：

- `lt_pathman.enable` — 完全禁用（或启用）`lt_pathman`
- `lt_pathman.enable_runtimeappend` — 切换`RuntimeAppend`自定义节点
- `lt_pathman.enable_runtimemergeappend` — 切换`RuntimeMergeAppend`自定义节点
- `lt_pathman.enable_partitionfilter` — 切换`PartitionFilter`自定义节点（用于INSERT操作）
- `lt_pathman.enable_partitionrouter` — 切换`PartitionRouter`自定义节点（用于跨分区UPDATE操作）
- `lt_pathman.enable_auto_partition` — 切换自动分区创建开启（每个会话）
- `lt_pathman.enable_bounds_cache` — 切换边界缓存开启（加快分区方案的更新）
- `lt_pathman.insert_into_fdw` — 允许向各种FDW进行INSERT操作（`(disabled | postgres | any_fdw)`）
- `lt_pathman.override_copy` — 切换COPY语句注入

要永久性地禁用先前分区的某些表的`lt_pathman`，请使用`disable_pathman_for()`函数：

```
SELECT disable_pathman_for('range_rel');
```

所有的部分和数据都将保持不变，并将由标准的LightDB继承机制处理。

## H. 31. `lt_patch`

`lt_patch` 模块提供了一种跟踪服务器执行的所有补丁信息的方式。

在创建 `lt_patch` 之前，您必须先创建 `file_fdw` 扩展并设置 `LTHOME` 环境变量。

当加载 `lt_patch` 模块时，它会跟踪服务器中所有数据库的补丁。为了访问和操作这些补丁信息，该模块提供了一个名为 `lt_patch` 的表和实用函数 `lt_patch`。这些不是全局可用的，但可以通过 `CREATE EXTENSION lt_patch` 为特定的数据库启用。

### H. 31.1. `lt_patch` 表

该模块收集的信息可以通过名为 `lt_patch` 的表进行访问。该表包含每个不同补丁的一行，表的列在 Table H. 15中列出。

Table H. 15. `lt_patch` 列

列类型	描述
<code>name varchar</code>	补丁名称
<code>dependency varchar</code>	依赖补丁
<code>component_type varchar</code>	组件类型：PATCH BUGFIX
<code>apply_type varchar</code>	应用类型：INSTANCE APP BOTH
<code>apply_version varchar</code>	应用版本
<code>action varchar</code>	操作：REMOVED INSTALLED
<code>status varchar</code>	状态：“APP FAILED” “APP SUCCESS” “INSTANCE FAILED” “SUCCESS”
<code>create_time timestamp</code>	



列类型
描述
补丁创建时间
update_time timestamp 补丁更新时间

## H. 31. 2. 函数

lt\_patch(showtext boolean) returns setof record

lt\_patch 表是基于同名函数 lt\_patch 定义的。

## H. 31. 3. 示例输出

```
lightdb@postgres=# select * from lt_patch;
```

name	dependency	component_type	apply_type	apply_version
action	status	create_time	update_time	
patch-1.0-8717		patch	BOTH	13.8-22.4
removed		2022-12-29 18:25:48	2022-12-29 19:24:33	
1.0-8719		patch	BOTH	13.8-22.4
removed		2022-12-29 19:58:39	2022-12-29 21:18:52	

(2 rows)

```
lightdb@postgres=# select lt_patch();
```

lt_patch
(patch-1.0-8717, , patch, BOTH, 13.8-22.4, removed, , "2022-12-29 18:25:48", "2022-12-29 19:24:33")
(1.0-8719, , patch, BOTH, 13.8-22.4, removed, , "2022-12-29 19:58:39", "2022-12-29 21:18:52")

(2 rows)

## H. 32. lt\_prewarm

lt\_prewarm 模块提供了一种方便的方式，可以将关系数据加载到操作系统缓存或 LightDB 缓存中。可以使用 pg\_prewarm 函数手动进行预热，也可以在 shared\_preload\_libraries 中包含 lt\_prewarm 来自动执行预热。在后一种情况下，系统会运行一个后台进程，定期将共享缓存的内容记录到名为 autoprewarm.blocks 的文件中，并使用 2 个后台进程在重启后重新加载这些块。

LightDB 在启动（重启）过程中增加了自动预热的功能，包括自动预热数据库、表和索引。预热数据库时，它会预热数据库下的所有非系统表（包括表下的分区表和索引）。预热表时，它也会预热表的分区表和索引。当然，在同时配置预热数据库和数据库下的表时，我们不会重复预热。

### H. 32. 1. 函数

```
pg_prewarm(regclass, mode text default 'buffer', fork text default 'main',
first_block int8 default null,
last_block int8 default null) RETURNS int8
```

第一个参数是要预热的关系。第二个参数是要使用的预热方法，如下面所讨论的；第三个参数是要预热的关系分支，通常为 `main`。第四个参数是要预热的第一个块号（`NULL` 被接受为零的同义词）。第五个参数是要预热的最后一个块号（`NULL` 表示预热到关系中的最后一个块）。返回值是预热的块数。

有三种可用的预热方法。 `prefetch` 向操作系统发出异步预取请求（如果支持的话），否则会抛出错误。 `read` 读取请求范围内的块；与 `prefetch` 不同，它是同步的，并且在所有平台和构建中都支持，但可能会更慢。 `buffer` 将请求范围内的块读入数据库缓存中。

注意，使用任何一种方法，尝试预热的块数超过了可以缓存的数量 — 当使用 `prefetch` 或 `read` 时，由操作系统缓存；当使用 `buffer` 时，由 `LightDB` 缓存 — 可能会导致较低编号的块被逐出，因为高编号的块被读入。预热的数据也没有受到缓存逐出的特殊保护，因此可能会发生其他系统活动在读取后不久将新预热的块逐出；相反，预热也可能从缓存中逐出其他数据。因此，通常在启动时，当缓存大部分为空时，预热最有用。

```
autoprewarm_start_worker() RETURNS void
```

启动主要的 `autoprewarm` 工作进程。这通常会自动发生，但如果在服务器启动时未配置自动预热，并且您希望稍后启动工作进程，则此函数很有用。

```
autoprewarm_dump_now() RETURNS int8
```

立即更新 `autoprewarm.blocks`。如果 `autoprewarm` 工作进程未运行但您预计在下次重启后运行它，则此操作可能很有用。返回值是写入 `autoprewarm.blocks` 的记录数。

## H. 32. 2. 配置参数

```
lt_prewarm.autoprewarm (boolean)
```

控制服务器是否应该运行 `autoprewarm` 工作进程。默认情况下是开启的。此参数只能在服务器启动时设置。

```
lt_prewarm.autoprewarm_interval (int)
```

这是更新 `autoprewarm.blocks` 的时间间隔。默认值为 300 秒。如果设置为 0，则文件将不会在固定间隔内转储，而只有在服务器关闭时才会转储。

这些参数必须在 `lightdb.conf` 中设置。典型用法可能是：

```
# lightdb.conf
shared_preload_libraries = 'pg_prewarm'

pg_prewarm.autoprewarm = true
pg_prewarm.autoprewarm_interval = 300s
```

```
lt_prewarm.lt_autoprewarm_dbs (text)
```

此参数配置要自动预热的数据库列表。例如， `lt_prewarm.lt_autoprewarm_dbs = 'db1, db2, db3'`。

```
lt_prewarm.lt_autoprewarm_tables (text)
```

此参数配置要自动预热的表的列表。例如， `lt_prewarm.lt_autoprewarm_tables = 'db1.schema1.t1, db1.schema1.t2, db1.schema2.t1, db2.schema1.t1'`。

lt\_prewarm.lt\_autoprewarm\_indexes (text)

此参数配置要自动预热的索引列表。 例如， lt\_prewarm.lt\_autoprewarm\_indexes = 'db1.schema1.i1, db1.schema1.i2, db1.schema2.i1, db2.schema1.i1'。

### H. 32. 3. 作者

Robert Haas <rhaas@postgresql.org>

## H. 33. lt\_proctab

lt\_proctab 是一组存储函数，可以访问操作系统的进程表，从而通过数据库查询系统统计信息。

### H. 33. 1. functions

#### H. 33. 1. 1. pg\_proctab

获取关于所有 LightDB 进程的信息。所有进程列表从 pg\_stat\_activity 中获取， 进程信息从系统文件 /proc 中获取。

```
lightdb@postgres=# select * from pg_proctab();
-[ RECORD 1 ]-----+-----
pid              | 64994
comm             | lightdb
fullcomm        | lightdb: autovacuum launcher
state           | S
ppid            | 64986
pgrp            | 64994
session        | 64994
tty_nr          | 0
tpgid           | -1
flags           | 4202560
minflt          | 529
cminflt        | 0
majflt         | 0
cmajflt        | 0
utime           | 1
stime          | 0
cutime         | 0
cstime         | 0
priority       | 20
nice           | 0
num_threads    | 1
itrealvalue    | 0
starttime     | 3258371
vsize          | 1902161920
rss            | 3588
exit_signal    | 17
processor      | 3
rt_priority    | 0
policy         | 0
delayacct_blkio_ticks | 0
uid            | 1001
username       | lightdb
rchar          | 66914
```

```
wchar          | 26
syscr          | 72
syscw         | 26
reads         | 0
writes        | 0
cwrites       | 0
```

### H. 33. 1. 2. pg\_cputime

从 /proc/stat 文件获取 CPU 信息。

```
lightdb@postgres=# select * from pg_cputime();
 user | nice | system | idle | iowait
-----+-----+-----+-----+-----
 289580 | 61 | 29378 | 15540161 | 7253
(1 row)
```

### H. 33. 1. 3. pg\_loadavg

从 /proc/loadavg 文件获取负载信息。

```
lightdb@postgres=# select * from pg_loadavg();
 load1 | load5 | load15 | last_pid
-----+-----+-----+-----
 0.08 | 0.05 | 0.05 | 74363
(1 row)
```

### H. 33. 1. 4. pg\_memusage

从 /proc/meminfo 文件获取内存使用信息。

```
lightdb@postgres=# select * from pg_memusage();
 memused | memfree | memshared | membuffers | memcached | swapused | swapfree |
 swapcached
-----+-----+-----+-----+-----+-----+-----+
 4441516 | 3552556 | 0 | 2084 | 3770828 | 0 | 4064252 |
 0
(1 row)
```

### H. 33. 1. 5. pg\_diskusage

从 /proc/diskstats 文件获取磁盘使用信息。

```
lightdb@postgres=# select * from pg_diskusage();
-[ RECORD 1 ]-----
 major          | 8
 minor          | 0
 devname        | sda
 reads_completed | 36385
 reads_merged   | 67
 sectors_read   | 2060883
 readtime       | 25260
 writes_completed | 384347
 writes_merged  | 3697
 sectors_written | 16852473
```

```
writetime      | 351385
current_io     | 0
iotime        | 326759
totaliotime    | 372183
```

## H. 34. lt\_profile (轻量级开发工具配置文件)

这个LightDB的扩展可以帮助你找出在你的LightDB数据库中耗费最多资源的活动。

### H. 34. 1. 概念

这个扩展基于LightDB的统计视图和扩展lt\_stat\_statements、lt\_stat\_activity和system\_stats。它是用纯pl/pgsql编写的，不需要任何外部库或软件，只需要LightDB数据库本身。

此扩展将在您的数据库中创建历史存储库。这个存储库将保存您的LightDB群集的统计“样本”。通过调用take\_sample()函数来获取样本。LightDB还提供了另一个lt\_cron扩展工具来执行定期获取样本任务。

周期性的样本可以帮助您查找过去最耗费资源的活动。假设，您几个小时前报告了性能下降的问题。解决这样的问题，您可以在两个样本之间构建报告，将性能问题期间的负载配置文件绑定在一起。

您可以在运行任何批处理之前和之后明确地采取样本。

无论何时您采取样本，都将调用pg\_stat\_statements\_reset()，确保由于达到lt\_stat\_statements.max而不会丢失语句。此外，报告将包含一节，告知您是否在任何样本中捕获的语句计数达到了lt\_stat\_statements.max的90%。

lt\_profile在一个群集中安装后，可以从其他被称为服务器的群集中收集统计信息。您只需要定义一些服务器，提供名称和连接字符串，并确保可以连接到所有服务器的所有数据库。现在，您可以跟踪例如主机的备用统计信息，或来自任何其他服务器的统计信息。扩展安装后，将自动创建一个本地服务器-这是lt\_profile所在的群集的服务器。

### H. 34. 2. 扩展架构

扩展由四个部分组成：

- 历史存储库是采样数据的存储。存储库是一个扩展表的集合。
- 样本管理引擎是一组用于获取样本和通过从中删除过时的样本数据来支持存储库的函数。
- 报告引擎是一组用于生成基于历史存储库数据的报告的函数。
- 管理函数允许您创建并管理服务器和基线。

#### H. 34. 2. 1. 插件用到的表

##### H. 34. 2. 1. 1. servers

监测的服务器列表 (LightDB集群)。

Table H. 16. servers Columns

列类型	描述
server_id integer	服务器ID
server_name name	服务器名称

列类型	描述
server_description text	服务器描述
server_created timestamp with time zone DEFAULT	服务器创建时间，默认为创建时函数now() 的值
db_exclude name[]	排除的数据库
enabled boolean	是否开启，默认开启
connstr text	用于dblink的Connstr
max_sample_age integer	全局设置max_sample_age 将被此值重写，如果此值被设置
last_sample_id integer	上次采样ID，默认为0
size_smp_wnd_start time with time zone	开始采样时间
size_smp_wnd_dur interval hour to second	采样持续时间
size_smp_interval interval day to minute	采样时间间隔

#### H. 34. 2. 1. 2. samples

采样列表。

Table H.17. samples Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
sample_time timestamp (0) with time zone	采样时间

#### H. 34. 2. 1. 3. sample\_server\_hardware

服务器硬件配置信息（通过system\_stats收集）。信息包含CPU数量，线程数，核心数，插槽数，内存信息。

Table H.18. sample\_server\_hardware Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
architecture text	系统的 CPU 架构

列类型	描述
byte_order text	CPU架构的字节顺序
cpus integer	CPU数量
threads_per_core integer	单核CPU线程数
cores_per_socket integer	一个插槽CPU核心数
sockets integer	系统插槽数量
vendor_id text	厂家ID
cpu_family integer	CPU系列
model integer	型号
model_name text	型号名称
cpu_mhz text	CPU频率 (单位 mhz)
l1dcache text	一级数据缓存
l1icache text	一级指令缓存
l2cache text	二级缓存
l3cache t	三级缓存
memory_total integer	问题内存大小
swap_total integer	内存交换区大小

#### H. 34. 2. 1. 4. sample\_cluster\_instance

集群配置信息。

Table H. 19. sample\_cluster\_instance Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
hostname text	主机名 (参考 servers.server_name)
cluster_name text	

列类型	描述
	集群名 (参考 <code>current_setting('cluster_name')</code> )
role text	角色, 主 如果 <code>pg_is_in_recovery() = false</code> , 否则为备
release text	LightDB版本号, 示例: LightDB 13.8-23.1
startup_time timestamp	启动时间

#### H. 34. 2. 1. 5. sample\_kernel\_settings

采样内核设置。

Table H. 20. sample\_kernel\_settings Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
name text	名称
value text	值
sourcefile text	源文件

#### H. 34. 2. 1. 6. sample\_cpu\_usage

采样CPU使用情况, 通过分析 `/proc/stat` 产生统计数据。

Table H. 21. sample\_cpu\_usage Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
user_cpu float4	用户CPU
sys_cpu float4	系统CPU
idle_cpu float4	闲置CPU
io_wait float4	IO等待

#### H. 34. 2. 1. 7. sample\_db\_memory\_usage

采样内存使用情况。



Table H. 22. sample\_db\_memory\_usage Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
shared_memory bigint	共享内存大小 (单位: 字节)
local_memory bigint	本地内存大小 (单位: 字节)

#### H. 34. 2. 1. 8. sample\_settings

采样设置。

Table H. 23. sample\_settings Columns

列类型	描述
server_id integer	服务器ID
first_seen timestamp (0) with time zone	第一次登入时间
setting_scope smallint	设置范围. pg_settings 取值为1, 其它 adm 函数取值为2
name text	名称
setting text	设置
reset_val text	重置值
boot_val text	启动值
unit text	单元
sourcefile text	源文件
sourceline integer	源行
pending_restart boolean	重启标识挂起状态

#### H. 34. 2. 1. 9. sample\_timings

各函数采样时间。

Table H. 24. sample\_timings Columns

列类型	描述
server_id integer	

列类型	描述
	服务器ID
sample_id integer	采样ID
event text	采样事件
time_spent interval MINUTE TO SECOND (2)	此事件的时间开销

#### H. 34. 2. 1. 10. baselines

Baselines 设置。

Table H. 25. baselines Columns

列类型	描述
server_id integer	服务器ID
bl_id integer	Baseline 采样ID
bl_name varchar (25)	Baseline 采样名字
keep_until timestamp (0) with time zone	此baseline采样需要保存的截止时间

#### H. 34. 2. 1. 11. bl\_samples

Baseline 采样列表。

Table H. 26. bl\_samples Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
bl_id integer	Baseline ID

#### H. 34. 2. 1. 12. stmt\_list

SQL语句列表。

Table H. 27. stmt\_list Columns

列类型	描述
server_id integer	服务器ID
queryid_md5 char(32)	查询ID的MD5

列类型	描述
query text	查询SQL文本

#### H. 34. 2. 1. 13. sample\_stat\_database

数据库采样统计。

Table H. 28. sample\_stat\_database Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
datid oid	数据库ID
datname name	数据库
xact_commit bigint	提交事务数
xact_rollback bigint	回滚事务数
blks_read bigint	块读数
blks_hit bigint	块命中数
tup_returned bigint	元组返回数
tup_fetched bigint	元组获取数
tup_inserted bigint	元组插入数
tup_updated bigint	元组更新数
tup_deleted bigint	元组删除数
conflicts bigint	总冲突数，包含表空间，锁，快照，缓存，启动死锁
temp_files bigint	临时文件
temp_bytes bigint	临时文件大小（单位：字节）
deadlocks bigint	死锁
blk_read_time double precision	块读取时间
blk_write_time double precision	

列类型	描述
	块写入时间
stats_reset timestamp with time zone	上次重置统计的时间
datsize bigint	数据库大小
datsize_delta bigint	数据库大小变化
datistemplate boolean	真, 则为模板数据库

#### H. 34. 2. 1. 14. sample\_statements

SQL语句的采样统计。

Table H. 29. sample\_statements Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
userid oid	执行此条SQL的用户OID
datid oid	执行此条语句的数据库OID
queryid bigint	内部HASH码, 通过SQL解析树计算所得
queryid_md5 char(32)	查询MD5, 通过计算SQL文本计算所得
plans bigint	SQL计划次数 ( 开启 lt_stat_statements.track_planning 有效, 否则为零)
total_plan_time double precision	SQL计划总时间 (单位: 毫秒, 开启 lt_stat_statements.track_planning 有效, 否则为零)
min_plan_time double precision	SQL计划最短时间 (单位: 毫秒, 开启 lt_stat_statements.track_planning 有效, 否则为零)
max_plan_time double precision	SQL计划最长时间 (单位: 毫秒, 开启 lt_stat_statements.track_planning 有效, 否则为零)
mean_plan_time double precision	SQL计划平均时间 (单位: 毫秒, 开启 lt_stat_statements.track_planning 有效, 否则为零)
stddev_plan_time double precision	SQL计划标准差 (单位: 毫秒, 开启 lt_stat_statements.track_planning 有效, 否则为零)
calls bigint	SQL执行总次数

列类型	描述
total_exec_time double precision	SQL执行总时间（单位：毫秒）
min_exec_time double precision	SQL执行最短时间（单位：毫秒）
max_exec_time double precision	SQL执行最长时间（单位：毫秒）
mean_exec_time double precision	SQL执行平均时间（单位：毫秒）
stddev_exec_time double precision	SQL执行时间标准差（单位：毫秒）
rows bigint	SQL执行时获取或影响到的总行数
shared_blks_hit bigint	此SQL总共享块缓存命中次数
shared_blks_read bigint	此SQL总共享块缓存读取次数
shared_blks_dirtied bigint	此SQL总共享块缓存dirtied次数
shared_blks_written bigint	此SQL总共享块写入次数
local_blks_hit bigint	此SQL总本地块命中次数
local_blks_read bigint	此SQL总本地块读取次数
local_blks_dirtied bigint	此SQL总本地块dirtied次数
local_blks_written bigint	此SQL总本地块写入次数
temp_blks_read bigint	此SQL总临时块读取次数
temp_blks_written bigint	此SQL总临时块写入次数
blk_read_time double precision	读取块总时间（单位：in milliseconds, 需 track_io_timing 开启, 否则为0）
blk_write_time double precision	写入块总时间（单位：in milliseconds, 需 track_io_timing 开启, 否则为0）
wal_records bigint	此SQL产生的总WAL records数
wal_fpi bigint	此SQL产生的总WAL满页数
wal_bytes numeric	此SQL产生的WAL总大小（单位：字节）

#### H. 34. 2. 1. 15. sample\_statements\_total

以数据库为分组的总的采样统计信息。

Table H. 30. sample\_statements\_total Columns

列类型	描述
server_id integer	服务器ID
datid oid	执行此条语句的数据库OID
plans bigint	SQL计划次数 ( 开启 lt_stat_statements.track_planning 有效, 否则为零)
total_plan_time double precision	SQL计划总时间 (单位: 毫秒, 开启 lt_stat_statements.track_planning 有效, 否则为零)
calls bigint	SQL执行总次数
total_exec_time double precision	SQL执行总时间 (单位: 毫秒)
rows bigint	SQL执行时获取或影响到的总行数
shared_blks_hit bigint	此SQL总共享块缓存命中次数
shared_blks_read bigint	此SQL总共享块缓存读取次数
shared_blks_dirtied bigint	此SQL总共享块缓存dirtied次数
shared_blks_written bigint	此SQL总共享块写入次数
local_blks_hit bigint	此SQL总本地块命中次数
local_blks_read bigint	此SQL总本地块读取次数
local_blks_dirtied bigint	此SQL总本地块dirtied次数
local_blks_written bigint	此SQL总本地块写入次数
temp_blks_read bigint	此SQL总临时块读取次数
temp_blks_written bigint	此SQL总临时块写入次数
blk_read_time double precision	读取块总时间 (单位: in milliseconds, 需 track_io_timing 开启, 否则为0)
blk_write_time double precision	写入块总时间 (单位: in milliseconds, 需 track_io_timing 开启, 否则为0)
wal_records bigint	此SQL产生的总WAL records数
wal_fpi bigint	此SQL产生的总WAL满页数
wal_bytes numeric	此SQL产生的WAL总大小 (单位: 字节)

列类型	描述
statements bigint	总SQL数

H. 34. 2. 1. 16. sample\_wait\_event\_total

总的等待事件采样统计。

Table H. 31. sample\_wait\_event\_total Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
pid integer	当前事件的后台进程ID
queryid bigint	当前事件的查询ID
wait_event_type text	等待事件类型
wait_event text	等待事件子类型
wait_time double precision	自上次采样后，此类型事件总计时
state text	后台进程状态

H. 34. 2. 1. 17. tablespaces\_list

采样时采集到的表空间。

Table H. 32. tablespaces\_list Columns

列类型	描述
server_id integer	服务器ID
tablespaceid oid	表空间OID
tablespacename name	表空间名字
tablespacepath oid	表空间路径

H. 34. 2. 1. 18. sample\_stat\_tablespaces

表空间采样统计，数据来源于函数pg\_tablespace\_size(oid)。

Table H. 33. sample\_stat tablespaces Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
tablespaceid oid	表空间OID
size integer	表空间总磁盘占用（单位：字节）
size_delta integer	自上次采样后，表空间变化大小（单位：字节）

H. 34. 2. 1. 19. tables\_list

此服务器数据库中的表列表。

Table H. 34. tables\_list Columns

列类型	描述
server_id integer	服务器ID
datid oid	数据库OID
relid oid	表OID
relkind char(1)	表类型
reltoastrelid oid	此表的toast表OID
schemaname name	表的模式名称
relname name	表的名字（参考：pg_class.relname）

H. 34. 2. 1. 20. sample\_stat\_tables

表采样统计。

Table H. 35. sample\_stat\_tables Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
datid oid	数据库OID
relid oid	



列类型	描述
	表OID
tablespaceid oid	表空间OID
seq_scan bigint	自上次采样后, 顺序扫描次数差
seq_tup_read bigint	自上次采样后, 顺序元组读取差
idx_scan bigint	自上次采样后, 索引扫描差
idx_tup_fetch bigint	自上次扫描后, 元组读取差
n_tup_ins bigint	自上次扫描后, 插入元组数量差
n_tup_upd bigint	自上次扫描后, 更新元组数量差
n_tup_del bigint	自上次扫描后, 更新元组数量差
n_tup_hot_upd bigint	自上次扫描后, 热更元组数量差
n_live_tup bigint	在线元组数
n_dead_tup bigint	无效元组数
n_mod_since_analyze bigint	自analyze后, 修改的数量
n_ins_since_vacuum bigint	自vacuum后, 插入的数量
sample_id timestamp with time zone	上次vacuum时间
last_autovacuum timestamp with time zone	上次autovacuum时间
last_analyze timestamp with time zone	上次analyze时间
last_autoanalyze timestamp with time zone	上次autoanalyze时间
vacuum_count bigint	自上次采样后, vacumm数量差
autovacuum_count bigint	自上次采样后, autovacumm数量差
analyze_count bigint	自上次采样后, analyze数量差
autoanalyze_count bigint	自上次采样后, auto analyze 数量差
heap_blks_read bigint	自上次采样后, 堆块读取数量差
heap_blks_hit bigint	

列类型	描述
	自上次采样后, 堆块命中数量差
idx_blks_read bigint	自上次采样后, 索引块读取数量差
toast_blks_read integer	自上次采样后, toast块读取数量差
toast_blks_hit integer	自上次采样后, toast块命中数量差
tidx_blks_read integer	自上次采样后, toast块命中数量差
tidx_blks_hit integer	自上次采样后, toast索引块命中数量差
resize integer	Toast表大小
resize_diff integer	自上次采样后, toast表大小差

#### H. 34. 2. 1. 21. sample\_stat\_tables\_total

表采样统计, 以服务器, 采样ID, 数据库ID, 表类型, 表空间ID为组。

Table H. 36. sample\_stat\_tables\_total Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
datid oid	数据库OID
sample_id oid	表空间OID
relkind char(1)	表类型
seq_scan bigint	自上次采样, 总顺序扫描数
seq_tup_read bigint	自上次采样, 总元组读取数
idx_scan bigint	自上次采样, 总索引扫描数
idx_tup_fetch bigint	自上次采样, 总索引元组获取数
n_tup_ins bigint	自上次采样, 总元组插入数
n_tup_upd bigint	自上次采样, 总元组更新数
n_tup_del bigint	自上次采样, 总元组删除数

列类型	描述
n_tup_hot_upd bigint	自上次采样, 总元组热更数
sample_id bigint	采样ID
vacuum_count bigint	自上次采样, 总vacuum数
autovacuum_count bigint	自上次采样, 总autovacuum数
analyze_count bigint	自上次采样, 总analyze数
autoanalyze_count bigint	自上次采样, 总 auto analyze 数
heap_blks_read bigint	自上次采样, 总堆块读取数
heap_blks_hit bigint	自上次采样, 总堆块命中数
idx_blks_read bigint	自上次采样, 总的索引块读取数
idx_blks_hit bigint	自上次采样, 总的索引块命中数
toast_blks_read bigint	自上次采样, 总的toast块读取数
toast_blks_hit bigint	自上次采样, 总的toast块命中数
tidx_blks_read bigint	自上次采样, 总的toast索引块读取数
tidx_blks_hit bigint	自上次采样, 总的toast索引块命中数
resize_diff bigint	自上次采样, 磁盘空间占用差

#### H. 34. 2. 1. 22. indexes\_list

采样索引列表, 包含索引名字, 模式。

Table H. 37. indexes\_list Columns

列类型	描述
server_id integer	服务器ID
datid oid	数据库ID
indexrelid oid	索引OID
relid oid	此索引所在的表的OID
schemaname name	

列类型
描述
表模式
indexrelname name 表名字

H. 34. 2. 1. 23. sample\_stat\_indexes

索引采样统计。

Table H. 38. samples Columns

列类型
描述
server_id integer 服务器ID
datid oid 数据库OID
tablespaceid oid 表空间OID
idx_scan bigint 自上次采样, 索引扫描差
idx_tup_read bigint 自上次采样, 索引元组读取差
idx_tup_fetch bigint 自上次采样, 索引元组获取差
idx_blks_read bigint 自上次采样, 索引块读取差
idx_blks_hit bigint 自上次采样, 索引块命中差
resize bigint 索引当前占用的磁盘空间
resize_diff bigint 自上次采样, 索引占用空间差
indisunique bigint 真, 如果是唯一索引

H. 34. 2. 1. 24. sample\_stat\_indexes\_total

索引采样统计。以服务器ID, 采样ID, 数据库ID, 表空间ID分组。

Table H. 39. sample\_stat\_indexes\_total Columns

列类型
描述
server_id integer 服务器ID
sample_id integer 采样ID
datid oid 数据库OID

列类型	描述
tablespaceid oid	表空间OID
idx_scan bigint	自上次采样, 总索引扫描数
idx_tup_read bigint	自上次采样, 总索引元组读取数
idx_tup_fetch bigint	自上次采样, 总索引元组获取数
idx_blks_read bigint	自上次采样, 总索引块读取数
idx_blks_hit bigint	自上次采样, 总索引块命中数
resize_diff bigint	自上次采样, 索引占用空间差

#### H. 34. 2. 1. 25. funcs\_list

采样函数列表。

Table H. 40. funcs\_list Columns

列类型	描述
server_id integer	服务器ID
datid integer	数据库OID
funcid oid	函数OID (参考: pg_proc.oid)
schemaname name	函数模式 (参考 pg_namespace.schemaname)
funcname name	函数名 (参考 pg_proc.proname)
funcargs text	函数参数

#### H. 34. 2. 1. 26. sample\_stat\_user\_functions

用户函数采样统计。

Table H. 41. sample\_stat\_user\_functions Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
datid oid	数据库OID

列类型	描述
funcid oid	函数OID (参考 pg_proc.oid)
calls bigint	自上次采样, 函数调用差
total_time double precision	自上次采样, 函数总调用差
self_time double precision	自上次采样, 函数自身调用差
trg_fn boolean	真, 如果函数为触发器

#### H. 34. 2. 1. 27. sample\_stat\_user\_func\_total

用户函数采样统计, 以服务器ID, 采样ID, 数据库尖, 触发函数分组。

Table H. 42. samples Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
datid oid	数据库OID
calls bigint	自上次采样, 调用次数
total_time double precision	自上次采样, 总调用次数
trg_fn boolean	真, 如果函数为触发器

#### H. 34. 2. 1. 28. sample\_stat\_cluster

集群采样统计。

Table H. 43. sample\_stat\_cluster Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
checkpoints_timed bigint	自上次采样, checkpoint数量差
checkpoints_req bigint	自上次采样, 请求的checkpoint数量差
checkpoint_write_time double precision	自上次采样, 累积checkpoint写时间差 (单位: 毫秒)

列类型	描述
checkpoint_sync_time double precision	自上次采样, 累积checkpoint同步时间差 (单位: 毫秒)
buffers_checkpoint bigint	自上次采样, checkpoint写缓存时间差 (单位: 毫秒)
buffers_clean bigint	自上次采样, 脏缓存写入数量差
maxwritten_clean bigint	自上次采样, 最大写入差 (当 written >= bgwriter_lru_maxpages 时计1)
buffers_backend bigint	自上次采样, 后台请求写入缓存数量差
buffers_backend_fsync bigint	自上次采样, 后台fsync次数差
buffers_alloc bigint	自上次采样, 缓存分配差
stats_reset timestamp with time zone	pg_stat上次重置时间
wal_size bigint	自上次采样, WAL大小差 (wal_size 通过 pg_wal_lsn_diff计算)

#### H. 34. 2. 1. 29. sample\_stat\_archiver

归档采样统计, 包含成功或失败的次数及时间。

Table H. 44. sample\_stat\_archiver Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
archived_count integer	上次归档成功数
last_archived_wal integer	上次归档 wal 文件
last_archived_time integer	Last archive timestamp
failed_count integer	上次归档失败数
last_failed_wal integer	上次归档 wal 文件
last_failed_time integer	上次归档失败时间
stats_reset integer	上次归档的重置时间

#### H. 34. 2. 1. 30. sample\_stat\_tables\_failures

表磁盘空间计算采样统计。

Table H. 45. sample\_stat\_tables\_failures Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
datid oid	数据库OID
relid oid	表OID (参考 pg_class.oid)
size_failed boolean	真, 计算表磁盘空间成功
toastsize_failed boolean	真, 计算toast表磁盘空间失败

H. 34. 2. 1. 31. sample\_stat\_indexes\_failures

索引失败采样统计。

Table H. 46. sample\_stat\_indexes\_failures Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
datid oid	数据库OID
indexrelid oid	索引OID (参考 pg_index.indexrelid)
size_failed boolean	真, 计算索引磁盘空间失败

H. 34. 2. 1. 32. lt\_stat\_backend\_type\_id\_text

此表用于将后台进程类型的ID和显示文本进行关联, 后台类型可以为 autovacuum launcher, autovacuum worker, client backend, background worker, background writer etc. 一旦有新的后台类型添加, 我们也需要将其添加到此表中, 此表于 LightDB 23.2 , lt\_profile 0.3.7添加。

Table H. 47. lt\_stat\_backend\_type\_id\_text Columns

列类型	描述
backend_type integer	后台类型ID
backend_type_text text	后台类型显示文本



H. 34. 2. 1. 33. lt\_stat\_cost\_type\_id\_text

此表用于将开销类型的ID及显示文本进行关联，开销类型有 DB Time, DB Wait, DB CPU. 自 LightDB 23.2 , lt\_profile 0.3.7 可用。

Table H. 48. lt\_stat\_cost\_type\_id\_text Columns

列类型	描述
cost_type integer	开销ID
cost_type_text text	开销显示文本

H. 34. 2. 1. 34. lt\_stat\_wait\_evt\_id\_text

此表用于关联事件ID和事件显示文本。自 Available since LightDB 23.2 , lt\_profile 0.3.7 可用。

Table H. 49. lt\_stat\_wait\_evt\_id\_text Columns

列类型	描述
evt_id integer	事件类型ID
evt_id_text text	事件类型显示文本

H. 34. 2. 1. 35. lt\_stat\_wait\_evt\_sub\_id\_text

此表用于关联子子件事ID以及子事件显示文本。自 LightDB 23.2 , lt\_profile 0.3.7 可用。

Table H. 50. lt\_stat\_wait\_evt\_sub\_id\_text Columns

列类型	描述
evt_id integer	子事件ID (参考 lt_stat_wait_evt_id_text.evt_id)
evt_sub_id integer	事件ID
evt_sub_id_text text	子事件显示文本

H. 34. 2. 1. 36. lt\_stat\_cost\_ls

各事件累积时间开销统计，包含 DB Time, DB Wait。默认情况下新的采样数据每秒采样一次，然后插入此表。自 LightDB 23.2 , lt\_profile 0.3.7 。

Table H. 51. lt\_stat\_cost\_ls Columns

列类型	描述
bg_type INT	后台进程类型ID

列类型	描述
cost_type INT	采样开销类型ID
evt_id INT	采样事件ID
evt_sub_id INT	采样子事件ID
cost BIGINT	总开销（单位：微秒）

#### H. 34. 2. 1. 37. lt\_stat\_cost\_60s

数据库等待事件采样统计，默认以1分钟为间隔进行归档。触发函数“collect\_activity\_profile”，自 LightDB 23.2，lt\_profile 0.3.7。

Table H. 52. lt\_stat\_cost\_60s Columns

列类型	描述
bg_type INT	后台进程类型ID
cost_type INT	采样开销类型ID
evt_id INT	采样事件ID
evt_sub_id INT	采样子事件ID
cost BIGINT	总开销时间（单位：微秒）

#### H. 34. 2. 1. 38. lt\_stat\_cost\_600s

数据库等待事件采样统计，默认以10分钟为间隔进行归档。触发函数“collect\_activity\_profile”，自 LightDB 23.2，lt\_profile 0.3.7。

Table H. 53. lt\_stat\_cost\_600s Columns

列类型	描述
server_id integer	服务器ID
sample_id integer	采样ID
bg_type INT	后台进程类型ID
cost_type INT	采样开销类型ID
evt_id INT	采样事件ID
evt_sub_id INT	采样子事件ID

列类型	描述
cost BIGINT	总开销时间（单位：微秒）

### H. 34. 3. 先决条件

尽管通常将lt\_profile安装在目标集群中，但它也可以从其他集群中收集性能数据。因此，我们有lt\_profile数据库和服务器的先决条件。

#### H. 34. 3. 1. lt\_profile数据库的先决条件

lt\_profile扩展依赖于plpgsql, dblink 和 ltfce 扩展。

#### H. 34. 3. 2. 服务器的先决条件

考虑设置以下统计收集器参数：

```
track_activities = on
track_counts = on
track_io_timing = on
track_functions = all/pl
```

要在报告中获取语句统计信息，连接字符串中提到的数据库必须配置了lt\_stat\_statements扩展程序。设置lt\_stat\_statements参数以满足您的需求（请参阅LightDB文档）：

- lt\_stat\_statements.max - 如果此参数设置过低，则可能会导致在采样之前某些语句统计信息被清除。如果您的lt\_stat\_statements.max似乎过小，报告将警告您。
- lt\_stat\_statements.track = 'top' -使用all值将影响报告中与语句相关的部分的%Total字段的准确性。

### H. 34. 4. 创建扩展

最简单的方法是将所有内容都安装在数据库的公共模式中：

```
lightdb@postgres=# CREATE EXTENSION dblink;
lightdb@postgres=# CREATE EXTENSION lt_stat_statements;
lightdb@postgres=# CREATE EXTENSION lt_stat_activity;
lightdb@postgres=# CREATE EXTENSION system_stats;
lightdb@postgres=# CREATE EXTENSION system_stats;
lightdb@postgres=# CREATE SCHEMA IF NOT EXISTS lt_catalog;
lightdb@postgres=# CREATE EXTENSION ltfce SCHEMA lt_catalog;
lightdb@postgres=# CREATE EXTENSION lt_profile;
```

如果您想将lt\_profile安装在其他模式中，只需创建该模式，并在该模式中安装扩展程序：

```
lightdb@postgres=# CREATE EXTENSION dblink;
lightdb@postgres=# CREATE EXTENSION lt_stat_statements;
lightdb@postgres=# CREATE EXTENSION lt_stat_activity;
lightdb@postgres=# CREATE EXTENSION system_stats;
lightdb@postgres=# CREATE SCHEMA IF NOT EXISTS lt_catalog;
lightdb@postgres=# CREATE EXTENSION ltfce SCHEMA lt_catalog;
lightdb@postgres=# CREATE SCHEMA profile;
```

```
lightdb@postgres=# CREATE EXTENSION lt_profile SCHEMA profile;
```

所有对象将在SCHEMA子句定义的模式中创建。在专用模式中安装是推荐的方式 - 扩展将创建自己的表、视图、序列和函数。将它们分开是个好主意。如果您在使用模块时不想指定模式限定符，请考虑更改search\_path设置。

## H. 34. 5. 权限

使用超级用户权限使用lt\_profile没有任何问题，但如果您想避免使用超级用户权限，请参考以下指南：

### H. 34. 5. 1. 关于lt\_profile数据库

创建一个非特权用户：

```
create role profile_usr login password 'pwd' ;
```

创建lt\_profile安装的模式：

```
create schema profile authorization profile_usr;
```

授予对dblink扩展所在模式的使用权限：

```
grant usage on schema public to profile_usr;
```

使用profile\_usr帐户创建扩展：

```
lightdb@postgres=> create extension lt_profile schema profile;
```

### H. 34. 5. 2. 关于服务器数据库

创建一个用于连接lt\_profile的用户：

```
create role profile_mon login password 'pwd_mon' ;
```

确保该用户具有连接到群集中任何数据库的权限（默认情况下为是），并且lt\_hba.conf将允许来自lt\_profile数据库主机的此类连接。此外，我们需要pg\_read\_all\_stats权限，并在pg\_stat\_statements\_reset上执行权限：

```
grant pg_read_all_stats to profile_mon;
grant execute on function pg_stat_statements_reset TO profile_mon;
```

### H. 34. 5. 3. 在lt\_profile数据库上设置服务器

非超级用户无法在没有密码的情况下建立连接。提供密码的最佳方法是使用密码文件。

注意：lt\_profile将连接到服务器上的所有数据库，因此密码文件必须使用通配符作为数据库名称。

作为一种不安全的方式，您可以在连接字符串中提供密码：

```
select server_connstr('local', 'dbname=postgres port=5432 user=profile_mon
password=pwd_mon');
```

## H. 34. 6. 使用lt\_profile

### H. 34. 6. 1. 设置扩展参数

您可以在lightdb.conf中定义扩展参数。默认值如下：

- `lt_profile.topn = 10` - 每个排序报告表中要报告的顶部对象（语句、关系等）数量。此参数还影响样本的大小 - 您希望在报告中出现的对象越多，我们需要在样本中保留的对象就越多。
- `lt_profile.max_sample_age = 7` - 样本的保留时间（以天为单位）。年龄大于`lt_profile.max_sample_age`天的样本将在下一次`take_sample()`调用时自动删除。
- `lt_profile.track_sample_timings = on` - 当此参数打开时，`lt_profile`将跟踪详细的样本采取时间。

## H. 34. 6. 2. 管理服务器

安装后，扩展将创建一个启用的`local`服务器 - 这是安装扩展的集群。

服务器管理函数：

- `create_server(server_name, server_connstr text, server_enabled boolean = TRUE, max_sample_age integer = NULL, description text = NULL)` - 创建一个新的服务器描述函数参数：
  - `server` - 服务器名称（必须唯一）
  - `server_connstr` - 服务器连接字符串
  - `enabled` - 服务器启用标志。当设置时，服务器将包括在通用的`take_sample()`调用中
  - `max_sample_age` - 服务器样本保留参数覆盖了这个服务器的全局`lt_profile.max_sample_age`设置
  - `description` - 服务器描述文本。将包括在报告中
- `drop_server(server_name)` - 删除一个服务器及其所有样本。
- `enable_server(server_name)` - 将服务器包括在通用的`take_sample()`调用中。
- `disable_server(server_name)` - 将服务器从通用的`take_sample()`调用中排除。
- `rename_server(server_name, new_name)` - 重命名服务器。
- `set_server_max_sample_age(server_name, max_sample_age integer)` - 为服务器设置新的保留期（以天为单位）。`max_sample_age`是整数值。要重置服务器的`max_sample_age`设置，请将其设置为`NULL`。
- `set_server_db_exclude(server_name, exclude_db name[])` - 为服务器设置排除数据库列表。在集群中无法连接到某些数据库的情况下使用（例如在Amazon RDS实例中）。
- `set_server_connstr(server_name, new_connstr text)` - 为服务器设置新的连接字符串。
- `set_server_description(server_name, description text)` - 设置新的服务器描述。
- `show_servers()` - 显示现有的服务器。

服务器创建示例：

```
SELECT server_new('omega', 'host=name_or_ip dbname=postgres port=5432');
```

## H. 34. 6. 3. 稀有关系尺寸收集

`LightDB`关系大小函数可能需要相当长的时间来收集数据库中所有关系的大小。此外，这些函数需要`AccessExclusiveLock`锁定关系。但是，每天收集关系大小可能已经足够。在服务

器大小收集策略指导下，`lt_profile`可以跳过关系大小收集。策略定义为允许收集关系大小的每日窗口，以及收集关系大小的两个样本之间的最小间隔。因此，当定义大小收集策略时，采样函数仅在窗口中采取样本并且具有大小的前一个样本比间隔旧时才会收集关系大小。函数`set_server_size_sampling`定义了此策略：

- `set_server_size_sampling(server_name, window_start time with time zone = NULL, window_duration interval hour to second = NULL, sample_interval interval day to minute = NULL)`
  - `server` - 服务器名称
  - `window_start` - 大小收集窗口的开始时间
  - `window_duration` - 大小收集窗口的持续时间
  - `sample_interval` - 两个具有收集关系大小的样本之间的最小时间间隔。只有设置了所有三个参数，大小收集策略才能被定义。

示例：

```
SELECT set_server_size_sampling('local', '23:00+03', interval '2 hour', interval '8 hour');
```

函数`show_servers_size_sampling`显示了所有服务器定义的大小收集策略：

```
lightdb@postgres=# SELECT * FROM show_servers_size_sampling();
server_name | window_start | window_end | window_duration | sample_interval
-----+-----+-----+-----+-----
local      | 23:00:00+03 | 01:00:00+03 | 02:00:00        | 08:00:00
```

当你在两个样本之间生成报告时，如果其中一个样本缺少关系大小数据，那么关系增长部分将从报告中排除。但是，报告生成函数的`with_growth`参数将扩展报告边界到最近收集到关系大小数据的样本。计算表的顺序扫描容量和索引的显式清理负载需要关系大小。当使用稀有关系大小收集时，相应的报告部分数据基于线性插值。

## H. 34. 6. 4. 样本

每个样本都包含有关自上一个样本以来数据库工作负载的统计信息。

### H. 34. 6. 4. 1. 样本函数

- `take_sample()` 函数将为所有`enabled`服务器收集样本。服务器样本将一个接一个地串行获取。函数返回一个表：

```
server      name,
result      text,
elapsed     interval
```

其中：

- `server` 是服务器名称
- `result` 是采样结果。如果成功采样，它可以是“OK”，在异常情况下将包含错误文本
- `elapsed` 是采样`server`所花费的时间。这种返回值使得使用SQL查询轻松控制样本的创建。
- `take_sample_subset([sets_cnt integer], [current_set integer])` 由于串行采样处理，`take_sample()` 函数可能需要相当长的时间。函数`take_sample_subset()` 将为一组启用的服务器采集样本。这对于并行采集样本非常方便。 `sets_cnt`是服务器子集的数量。`current_set`是要处理的子集，取值介于0和`sets_cnt-1`之间。函数返回一个表：

```
server      name,
result      text,
elapsed     interval
```

其中：

- server 是服务器名称
- result 是取样的结果。如果取样成功，则可以返回“OK”，如果出现异常，则包含错误文本。
- elapsed 是为 server 取样所经过的时间。
- take\_sample(server name [, skip\_sizes boolean]) 将为指定的服务器收集一个样本。例如，当您想要使用不同的采样频率，或者想在服务器上进行显式采样时，可以使用它。
- server - 要取样的服务器名称
- skip\_sizes - 覆盖服务器关系大小收集策略。当省略或设置为 null 的 skip\_size 参数时，该策略才会应用。如果 skip\_sizes 参数的值为 false，则采样时将包括关系大小，而如果为 true，则在采样过程中将跳过关系大小的收集。
- show\_samples([server name,] [days integer]) 返回一个表，包含一个 server 的现有样本（如果省略 server，则默认为本地服务器），以及最近 days 天的样本（如果省略，则返回所有现有样本）：

```
sample      integer,
sample_time timestamp (0) with time zone,
dbstats_reset timestamp (0) with time zone,
clustats_reset timestamp (0) with time zone,
archstats_reset timestamp (0) with time zone
```

其中：

- sample 是样本标识符
- sample\_time 是取样时间
- dbstats\_reset、clustats\_reset 和 archstats\_reset 通常为 null，但如果自上一个样本以来发生了重置，则会包含 pg\_stat\_database、pg\_stat\_bgwriter 和 pg\_stat\_archiver 的统计信息重置时间戳。采样收集函数还支持服务器存储库 - 它将根据保留策略删除过时的样本和基线。

#### H. 34. 6. 4. 2. 取样

您必须创建至少 2 个样本，才能在第 1 个和第 2 个样本之间构建您的第一个报告。通过调用 take\_sample() 函数来为所有启用的服务器采样。无需频繁地创建样本 - 通常每小时一个或两个样本就足够了。您可以使用类似 cron 的工具来安排采样的创建。以下是一个间隔为 30 分钟的示例：

```
*/30 * * * * ltsql -c 'SELECT take_sample()' > /dev/null 2>&1
```

然而，这样的调用没有对 take\_sample() 函数结果进行错误检查。考虑更加智能地使用 take\_sample() 函数，将结果提供给您的监控系统。

对于所有成功采样的服务器，函数将返回“OK”，并显示失败服务器的错误文本：

```
select * from take_sample();
server | result
      | elapsed
```

```

-----
+-----+
+-----+
ok_node   | OK
          | 00:00:00.48
fail_node | could not establish connection
          +| 00:00:00
          | SQL statement "SELECT
dblink_connect('server_connection',server_connstr)" +|
          | PL/pgSQL function take_sample(integer) line 69 at PERFORM
          +|
          | PL/pgSQL function take_sample_subset(integer,integer) line 27 at
assignment+|
          | SQL function "take_sample" statement 1
          +|
          | FATAL:  database "nodb" does not exist
          |
(2 rows)

```

### H. 34. 6. 4. 3. 采样数据保留

我们无法永远存储样本数据，因此我们有一个保留策略。您可以在三个层面上定义保留：

- 在 `lightdb.conf` 文件中设置参数 `lt_profile.max_sample_age`。这是一个通用的保留策略，如果没有其他保留策略被定义，则该策略生效。
- 在创建服务器时，可以定义服务器的 `max_sample_age` 设置，或使用 `set_server_max_sample_age()` 函数为现有服务器设置。此设置将覆盖特定服务器的全局 `lt_profile.max_sample_age` 设置。
- 创建一个基线（见下文）。基线将以最高优先级覆盖包含样本的保留期。

### H. 34. 6. 4. 4. 列出样本

使用 `show_samples()` 函数获取存储库中现有样本的列表。此函数将显示检测到的统计信息重置时间。

### H. 34. 6. 4. 5. 取样时间

当参数 `lt_profile.track_sample_timings` 开启时，`lt_profile` 将收集详细的取样时间统计信息。结果可以从 `v_sample_timings` 视图中获取。`v_sample_timings` 字段说明：

- `server_name` - 取样的服务器名称
- `sample_id` - 样本标识符
- `sample_time` - 取样时间
- `event` - 取样阶段
- `time_spent` - 事件中花费的时间

事件描述：

- `total` - 取样（所有阶段）。
- `connect` - 建立到服务器的 `dblink` 连接。
- `get server environment` - 获取服务器 GUC 参数、可用的扩展等。
- `collect database stats` - 查询 `pg_stat_database` 视图以获取关于数据库的统计信息。



- `calculate database stats` - 计算自上次取样以来数据库的差异统计信息。
- `collect tablespace stats` - 查询 `pg_tablespace` 视图以获取关于表空间的统计信息。
- `collect statement stats` - 使用 `lt_stat_statements` 扩展收集语句的统计信息。
- `query pg_stat_bgwriter` - 使用 `pg_stat_bgwriter` 视图收集集群统计信息。
- `query pg_stat_archiver` - 使用 `pg_stat_archiver` 视图收集集群统计信息。
- `collect object stats` - 收集数据库对象的统计信息。包括以下事件：
  - `db:dbname collect tables stats` - 收集 `dbname` 数据库中表的统计信息。
  - `db:dbname collect indexes stats` - 收集 `dbname` 数据库中索引的统计信息。
  - `db:dbname collect functions stats` - 收集 `dbname` 数据库中函数的统计信息。
- `maintain repository` - 执行支持例程。
- `calculate tablespace stats` - 计算表空间的差异统计信息。
- `calculate object stats` - 计算数据库对象的差异统计信息。包括以下事件：
  - `calculate tables stats` - 计算所有数据库中表的差异统计信息。
  - `calculate indexes stats` - 计算所有数据库中索引的差异统计信息。
  - `calculate functions stats` - 计算所有数据库中函数的差异统计信息。
- `calculate cluster stats` - 计算集群差异统计信息。
- `calculate archiver stats` - 计算归档器的差异统计信息。
- `delete obsolete samples` - 删除过时的基线和样本。

## H. 34. 6. 5. 基线

基线是一个命名的样本序列，具有自己的保留设置。基线可以用作报告构建函数的样本间隔。未定义的基线保留意味着无限保留。您可以使用基线保存关于某个时间段内数据库工作负载的信息。例如，您可能希望保存在负载测试期间收集的样本，或者在您的系统上进行定期负载以供以后参考。基线管理函数：

- `create_baseline([server name,] baseline_name varchar(25), start_id integer, end_id integer [, days integer])` - 创建一个基线
  - `server` - 服务器名称。如果省略，则默认为本地服务器
  - `name` - 基线名称。每个基线必须在服务器内具有唯一的名称。
  - `start_id, end_id` - 基线中包含的第一个和最后一个样本。
  - `days` - 基线保留时间。以整数天数自 `now()` 开始定义。此参数可以省略（或设置为 `null`），表示无限保留。
- `create_baseline([server name,] baseline_name varchar(25), time_range tstzrange [, days integer])` - 创建一个基线
  - `server` - 服务器名称。如果省略，则默认为本地服务器
  - `name` - 基线名称。每个基线必须在服务器内具有唯一的名称。
  - `time_range` - 基线时间间隔。基线将包括所有可用的样本，重叠此间隔。

- `days` - 基线保留时间。以整数天数自`now()`起定义。 此参数可以省略（或设置为`null`），表示无限保留。
- `drop_baseline([server name,] name varchar(25))` - 删除基线
  - `server` - 服务器名称。 如果省略，则假定为本地服务器。
  - `name` - 要删除的基线名称。 删除基线并不意味着立即删除其所有样本，它们只是从基线中排除， 因此不再受基线保留的覆盖。
- `keep_baseline([server name,] name varchar(25) [, days integer])` - 更改基线的保留时间
  - `server` - 服务器名称。 如果省略，则假定为本地服务器。
  - `name` - 基线名称。 此参数可以省略（或设置为`null`）以更改所有现有基线的保留时间。
  - `days` - 基线保留时间，以自`now()`起的天数计算。 此参数可以省略（或设置为`null`），以设置无限保留。
- `show_baselines([server name])` - 显示现有的基线。 调用此函数以获取有关现有基线（名称、采样间隔和保留时间）的信息。
  - `server` - 服务器名称。 如果省略，则假定为本地服务器。

```
lightdb@postgres=# SELECT * FROM baseline_show('local');
```

## H. 34. 6. 6. 数据导出和导入

可以从`lt_profile`扩展的实例中导出收集的样本，然后加载到另一个实例中。 当您想要将服务器从一个实例移动到另一个实例时，或者当您需要将收集的数据发送给支持团队时，此功能非常有用。

### H. 34. 6. 6. 1. 数据导出

使用函数`export_data()`将数据导出为常规表格。 您可以使用任何方法从数据库中导出此表格。例如，您可以使用`ltsql`的`copy`命令获取单个`.csv`文件：

```
lightdb@postgres=# \copy (select * from export_data()) to 'export.csv'
```

默认情况下，`export_data()`函数将导出所有配置服务器的所有样本。 但是，您可以将导出限制为仅限一个服务器，并进一步限制样本范围：

- `export_data([server name, [min_sample_id integer,] [max_sample_id integer]] [, obfuscate_queries boolean])` - 导出收集的数据
  - `server`是服务器名称。 如果省略，则假定为所有服务器。
  - `min_sample_id`和`max_sample_id` - 导出边界样本标识符（包括）。 `min_sample_id`边界的空值导出所有样本，直到`max_sample_id`， `max_sample_id`边界的空值导出自`min_sample_id`以来的所有样本。
  - `obfuscate_queries` - 仅在您想要隐藏查询文本时使用此参数 - 它们将导出为MD5哈希值。

### H. 34. 6. 6. 2. 数据导入

数据只能从本地表格导入，因此需要先加载先前导出的数据。在我们的情况下，使用`copy`命令：

```
lightdb@postgres=# CREATE TABLE import (section_id bigint, row_data json);
CREATE TABLE
lightdb@postgres=# \copy import from 'export.csv'
COPY 6437
```

现在，我们可以执行数据导入，并将此表格提供给import\_data()函数：

```
lightdb@postgres=# SELECT * FROM import_data('import');
```

尽管也导入了服务器描述，但是具有匹配名称的本地lt\_profile服务器将禁止导入操作。考虑暂时重命名这些服务器。如果您需要为先前导入的服务器导入新数据，则它们将通过系统标识符进行匹配，因此可以自由地将导入的服务器重命名为您想要的名称。所有服务器都以禁用状态导入。import\_data()函数只接受导入的表格：

- import\_data(data regclass)
  - data - 包含导出数据的表格。此函数返回实际加载到扩展表格中的行数。成功导入操作后，可以删除import表格。

## H. 34. 6. 7. 报告

报告是由报告函数生成的HTML标记，其中包含有关报告间隔期间实例工作量的统计信息。

### H. 34. 6. 7. 1. PWR报告函数

- get\_pwr([服务器名称,] 起始样本ID整数, 结束样本ID整数 [, 描述文本 [, 是否包括增长值 布尔型]]) - 根据样本标识生成PWR报告
- get\_pwr([服务器名称,] 时间范围 tstzrange [, 描述文本 [, 是否包括增长值 布尔型]]) - 生成最短样本间隔的PWR报告，覆盖提供的时间范围。
- get\_pwr([服务器名称], 基线 varchar(25) [, 描述文本 [, 是否包括增长值 布尔型]]) - 使用基线作为样本间隔生成PWR报告
- get\_pwr\_latest([服务器名称]) - 生成最近两个样本的PWR报告。函数参数：
  - server - 服务器名称。如果省略，则默认为local服务器。
  - start\_id - 间隔开始的样本标识符
  - end\_id - 间隔结束的样本标识符
  - time\_range - 时间范围 (tstzrange类型)
  - baseline - 基线名称
  - with\_growth - 一个标志，请求将间隔扩展到最近的边界并使用关联的增长数据。默认值为false
  - description - 文本备忘录，将包含在报告中作为报告描述。

### H. 34. 6. 7. 2. PSH报告函数

- get\_psh([服务器名称,] 起始样本ID整数, 结束样本ID整数 [, 描述文本 [, 是否包括增长值 布尔型]]) - 根据样本标识符生成PSH报告
- get\_psh([服务器名称,] 时间范围 tstzrange [, 描述文本 [, 是否包括增长值 布尔型]]) - 生成最短样本间隔的PSH报告，覆盖提供的时间范围。
- get\_psh([服务器名称], 基线 varchar(25) [, 描述文本 [, 是否包括增长值 布尔型]]) - 使用基线作为样本间隔生成PSH报告

- `get_psh_latest([服务器名称])` - 生成最近两个样本的PSH报告。函数参数：
  - `server` - 服务器名称。如果省略，则默认为local服务器。
  - `start_id` - 间隔开始的样本标识符
  - `end_id` - 间隔结束的样本标识符
  - `time_range` - 时间范围 (tstzrange类型)
  - `baseline` - 基线名称
  - `with_growth` - 一个标志，请求将间隔扩展到最近的边界并使用关联的增长数据。默认值为false
  - `description` - 文本备忘录，将包含在报告中作为报告描述。

报告生成示例：

```
$ ltsql -Aqtc "SELECT get_pwr(480,482)" -o report_480_482.html
```

对于任何其他服务器，请使用它的名称：

```
$ ltsql -Aqtc "SELECT get_pwr('omega',12,14)" -o report_omega_12_14.html
```

使用时间范围生成报告：

```
ltsql -Aqtc "select get_psh(tstzrange('2020-05-13 11:51:35+03','2020-05-13 11:52:18+03'))" -o report_range.html
```

此外，时间范围对于生成周期性报告非常有用。我们来生成最近24小时的报告：

```
ltsql -Aqtc "select get_psh(tstzrange(now() - interval '1 day',now()))" -o last24h_report.html
```

现在，您可以使用任何Web浏览器查看报告文件。

## H. 34. 7. PWR报告中的节

本节描述了报告表及其列。

### H. 34. 7. 1. 服务器统计信息

#### H. 34. 7. 1. 1. CPU使用率统计信息

包含CPU使用情况统计数据，例如报告开头和结尾的用户CPU、系统CPU、空闲CPU和IOWait。

- %User - 用户CPU百分比
- %System - 系统CPU百分比
- %Idle - 空闲CPU百分比
- %IO Wait - IOWait CPU百分比

#### H. 34. 7. 1. 2. 数据库内存使用情况统计数据

包含LightDB在报告开头和结尾的内存使用情况统计数据。

- 主机内存 (MB) - 主机的总物理内存

- 共享内存使用 (MB) - LightDB使用的共享内存, 由LightDB服务器的所有进程使用
- 本地内存使用 (MB) - LightDB使用的本地内存, 由每个后端进程为自己的使用分配
- 用于共享内存+本地内存的主机内存使用率 (%) - 共享内存和本地内存占主机总内存的百分比

#### H. 34. 7. 1. 3. 数据库统计数据

包含报告间隔期间每个数据库的统计数据, 基于pg\_stat\_database视图。

- 数据库 - 数据库名称
- 事务 - 数据库事务统计数据
  - 提交 - 已提交事务的数量 (xact\_commit)
  - 回滚 - 已回滚事务的数量 (xact\_rollback)
  - 死锁 - 检测到的死锁数量 (deadlocks)
- 块统计数据 - 数据库块读取和命中统计数据
  - 命中率 (%) - 缓冲区高速缓存命中比率
  - 读取 - 在此数据库中读取的磁盘块数 (blks\_read)
  - 命中 - 在缓冲区高速缓存中已经发现磁盘块的次数 (blks\_hit)
- 元组 - 元组统计数据部分
  - 返回的 - 返回的元组数量 (tup\_returned)
  - 获取的 - 获取的元组数量 (tup\_fetched)
  - 插入的 - 插入的元组数量 (tup\_inserted)
  - 更新的 - 更新的元组数量 (tup\_updated)
  - 删除的 - 删除的元组数量 (tup\_deleted)
- 临时文件 - 临时文件统计数据
  - 大小 - 由此数据库中的查询写入临时文件的数据总量 (temp\_bytes)
  - 文件 - 此数据库中的查询创建的临时文件数量 (temp\_files)
- 大小 - 报告间隔结束时的数据库大小 (pg\_database\_size())
- 增长 - 报告间隔期间的数据库增长量 (pg\_database\_size() 差异)

#### H. 34. 7. 1. 4. 按数据库统计的语句统计数据

包含pg\_stat\_statements数据的每个数据库的聚合总统计数据 (如果在报告间隔期间可用lt\_stat\_statements扩展)

- 数据库 - 数据库名称
- 调用 - 所有语句执行的总次数 (calls的总和)
- 时间 (s) - 花费的时间 (秒)

- 计划 - 花费的规划时间 (total\_plan\_time的总和) - 自lt\_stat\_statements 1.8版本后可用
- 执行 - 花费的执行时间 (total\_time或 total\_exec\_time的总和)
- 读取 - 花费的读取时间 (blk\_read\_time的总和)
- 写入 - 花费的写入时间 (blk\_write\_time的总和)
- 触发器 - 花费的触发器函数执行时间
- 获取块数 - 从磁盘和缓存中获取的总块数
  - 共享 - 获取的共享块数 (shared\_blks\_read + shared\_blks\_hit的总和)
  - 本地 - 获取的本地块数 (local\_blks\_read + local\_blks\_hit的总和)
- 脏块数 - 数据库中被修改的总块数
  - 共享 - 数据库中被修改的共享块数 (shared\_blks\_dirtied的总和)
  - 本地 - 数据库中被修改的本地块数 (local\_blks\_dirtied的总和)
- 临时块数 - 用于操作 (例如连接和排序) 的块数
  - 读取 - 读取的块数 (temp\_blks\_read的总和)
  - 写入 - 写入的块数 (temp\_blks\_written的总和)
- 本地块数 - 用于临时表的块数
  - 读取 - 读取的块数 (local\_blks\_read的总和)
  - 写入 - 写入的块数 (local\_blks\_written的总和)
- 语句数 - 捕获的语句的总数
- WAL 大小 - 语句生成的 WAL 总量 (wal\_bytes的总和)

#### H. 34. 7. 1. 5. 集群统计信息

此表包含来自 pg\_stat\_bgwriter 视图的数据

- 计划检查点 - 完成的检查点总数, 由于 checkpoint\_timeout 参数而按计划完成 (checkpoints\_timed 字段)
- 请求检查点 - 其他检查点的总数: 由于 max\_wal\_size、archive\_timeout 和 CHECKPOINT 命令的值而触发 (checkpoints\_req 字段)
- 检查点写入时间 (s) - 写入检查点花费的总时间 (checkpoint\_write\_time 字段)
- 检查点同步时间 (s) - 同步检查点花费的总时间 (checkpoint\_sync\_time 字段)
- 检查点缓冲区写入数 - 被检查点进程写入的缓冲区总数 (buffers\_checkpoint 字段)
- 后台缓冲区写入数 - 被后台写进程写入的缓冲区总数 (buffers\_clean 字段)
- 后端缓冲区写入数 - 被后端进程写入的缓冲区总数 (buffers\_backend 字段)
- 后端 fsync 调用数 - 后端进程 fsync 调用的总数 (buffers\_backend\_fsync 字段)

- Bgwriter 中断（缓冲区过多） - 由于达到 `bgwriter_lru_maxpages` 参数的值而导致的背景写进程中中断总数。
- 分配的缓冲区数量 - 已分配的缓冲区总数（`buffers_alloc`字段）
- WAL 生成量 - WAL 生成的总量（基于`pg_current_wal_lsn()`）
- 已归档的 WAL 段数 - 已归档的 WAL 段数（基于`pg_stat_archiver`视图中的`archived_count`字段）
- 归档失败的 WAL 段数 - WAL 段归档失败的次数（基于`pg_stat_archiver`视图中的`failed_count`字段）

#### H. 34. 7. 1. 6. 表空间统计信息

此表包含有关表空间大小和增长的信息：

- 表空间 - 表空间名称
- 路径 - 表空间路径
- 大小 - 报告间隔内最后一次采样时的表空间大小
- 增长 - 报告间隔内的表空间增长量

#### H. 34. 7. 2. 等待事件统计信息

此报告部分包含按等待时间排序的报告间隔内顶部等待事件统计信息表。

##### H. 34. 7. 2. 1. 主要等待事件类别

- IO - SQL查询时产生的各种文件读写活动。
- LW Lock - 访问某些需要LW lock的变量或获取LW lock时计数。

##### H. 34. 7. 2. 2. Main wait event categories

- WALSync - XLOG文件Fsync操作。
- WALWrite - 写页至XLOG文件。
- LockManager - 用于预测锁定共享哈希表以减少竞态。
- WALInsert - LW lock, 插入WAL。
- DataFileExtend - 为表增加一个块。
- DataFileRead - 读取一个表产生块读。
- DataFileTruncate - 将关系截断为指定块数。
- BufFileRead - 加载数据至缓冲区。
- ProcArray - LW lock, 多进程模型下对内存访问时, 涉及到的有`ShmemVariableCache`, `MyPgXact`, `procArray`。
- RelationMapRead - 从共享或本地MAP文件加载数据。
- SLRUSync - 缓冲区写入磁盘。

- SLRUWrite - 写缓冲区后写入磁盘。
- RegisterSyncRequest - 本地注册的同步请求。
- XidGen - LW lock, 设置并返回下一个 Transaction ID。
- XactSLRU - LW lock, 涉及xlog, clog相关内存。

#### H. 34. 7. 2. 3. 按等待时间排序的顶部等待事件类型

此表包含按等待时间排序的顶部lt\_profile.topn等待事件类型。

- 等待事件类型 - 等待事件类型, 每种类型可能包含多个等待事件。
- 等待时间 (秒) - 此类型所有等待事件的总等待时间。
- %总等待时间 - 此类型的总等待时间占群集中所有事件总等待时间的百分比。

#### H. 34. 7. 2. 4. 按等待时间排序的顶部等待事件

此表包含按等待时间排序的顶部lt\_profile.topn等待事件。

- 等待事件 - 进程正在等待的具体事件。
- 等待事件类型 - 此事件的类型。
- 等待时间 (秒) - 所有此事件的总等待时间。
- %总等待时间 - 所有此事件的总等待时间占群集中所有事件总等待时间的百分比。

### H. 34. 7. 3. SQL查询统计信息

此报告部分包含按多个重要统计信息排序的报告间隔内顶部语句表。如果在采样时可用, 数据将从pg\_stat\_statements视图中获取。

#### H. 34. 7. 3. 1. 按经过时间排序的顶部 SQL

此表包含按经过时间total\_plan\_time + total\_exec\_time排序的顶部lt\_profile.topn语句, 数据从pg\_stat\_statements视图中获取。

- 查询 ID - 数据库、用户和查询文本的哈希值作为查询标识符。与ltcenter工具兼容。十六进制表示的本地pg\_stat\_statements字段queryid以方括号形式呈现。
- SQL 文本 - 完整查询的前 100 个字符。
- 数据库 - 语句所在的数据库名称 (从dbid字段派生)
- %总时间 - 此语句的总时间占群集中所有语句总时间的百分比
- 时间 (秒) - 此语句的执行时间 (秒)
  - 经过时间 - 执行此语句所花费的总时间 (total\_plan\_time + total\_exec\_time)
  - 计划 - 执行此语句的计划时间 (total\_plan\_time字段)
  - 执行 - 执行此查询所花费的时间 (total\_exec\_time字段)
- I/O 时间 (秒) :
  - Read - 读取块所花费的时间 (blk\_read\_time 字段)



- Write - 写入块所花费的时间 (blk\_write\_time 字段)
- Plans - 语句被执行计划的次数 (plans 字段)
- Executions - 语句被执行的次数 (calls 字段)

#### H. 34. 7. 3. 2. 计划时间最长的 SQL

按照 pg\_stat\_statements 视图中的 total\_plan\_time 字段对前 lt\_profile.topn 个语句进行排序

- Query ID - 数据库、用户和查询文本的哈希值表示的查询标识符。与 ltcenter 工具兼容。原生 pg\_stat\_statements 字段 qieryid 以十六进制表示，显示在方括号中。
- SQL Text - 完整查询语句的前100个字符。
- Database - 语句所在的数据库名称 (从 dbid 字段派生而来)
- Plan elapsed (s) - 计划该语句所花费的时间 (total\_plan\_time 字段)
- %Elapsed - 该语句的计划时间占语句总时间的百分比
- Plan times (ms) - 该语句的详细计划时间统计信息 (以毫秒为单位)
  - Mean - 计划该语句的平均时间 (mean\_plan\_time 字段)
  - Min - 计划该语句的最小时间 (min\_plan\_time 字段)
  - Max - 计划该语句的最大时间 (max\_plan\_time 字段)
  - StdErr - 计划该语句的时间的总体标准差 (stddev\_plan\_time 字段)
- Plans - 该语句被执行计划的次数 (plans 字段)
- Executions - 该语句被执行的次数 (calls 字段)

#### H. 34. 7. 3. 3. 执行时间最长的 SQL

按照 pg\_stat\_statements 视图中的 total\_time (或 total\_exec\_time) 字段对前 lt\_profile.topn 个语句进行排序

- Query ID - 数据库、用户和查询文本的哈希值表示的查询标识符。与 ltcenter 工具兼容。原生 pg\_stat\_statements 字段 qieryid 以十六进制表示，显示在方括号中。
- SQL Text - 完整查询语句的前100个字符。
- Database - 语句所在的数据库名称 (从 dbid 字段派生而来)
- Exec (s) - 执行该语句所花费的时间 (total\_exec\_time 字段)
- %Elapsed - 该语句的执行时间占语句总时间的百分比
- %Total - 该语句的执行时间占集群中所有语句总执行时间的百分比
- I/O time (s):
  - Read - 读取块所花费的时间 (blk\_read\_time 字段)
  - Write - 写入块所花费的时间 (blk\_write\_time 字段)

- Rows - 语句检索或影响的行数 (rows 字段)
- Execution times (ms) - 该语句的详细执行时间统计信息 (以毫秒为单位)
  - Mean - 执行该语句的平均时间 (mean\_exec\_time 字段)
  - Min - 最短时间执行此语句 (min\_exec\_time 字段)
  - Max - 最长时间执行此语句 (max\_exec\_time 字段)
  - StdErr - 执行此语句所花费时间的总体标准差 (stddev\_exec\_time 字段)
- Executions - 此语句执行的次数 (calls 字段)

#### H. 34. 7. 3. 4. 按执行次数排名的 SQL

按 pg\_stat\_statements 视图的 calls 字段排名, 取前 lt\_profile.topn 条 SQL 语句

- Query ID - 数据库、用户和查询文本的哈希值, 作为查询的标识符。与 ltcenter 工具兼容。方括号中显示十六进制表示的 pg\_stat\_statements 的原生字段 queryid。
- SQL Text - 完整查询的前100个字符。
- Database - 语句所在的数据库名称 (从 dbid 字段派生)
- Executions - 语句执行次数 (calls 字段)
- %Total - 此语句的执行次数占集群中所有语句执行次数的百分比
- Rows - 语句检索或影响的行数 (rows 字段)
- Mean(ms) - 语句执行的平均时间, 以毫秒为单位 (mean\_time 或 mean\_exec\_time 字段)
- Min(ms) - 语句执行的最短时间, 以毫秒为单位 (min\_time 或 min\_exec\_time 字段)
- Max(ms) - 语句执行的最长时间, 以毫秒为单位 (max\_time 或 max\_exec\_time 字段)
- StdErr(ms) - 语句执行所花费时间的总体标准差, 以毫秒为单位 (stddev\_time 或 stddev\_exec\_time 字段)
- Elapsed(s) - 执行此查询所花费的时间, 以秒为单位 (total\_time 或 total\_exec\_time 字段)

#### H. 34. 7. 3. 5. 按I/O等待时间排名的 SQL

按读写时间 (blk\_read\_time + blk\_write\_time) 排名, 取前 lt\_profile.topn 条 SQL 语句

- Query ID - 数据库、用户和查询文本的哈希值, 作为查询的标识符。与 ltcenter 工具兼容。方括号中显示十六进制表示的 pg\_stat\_statements 的原生字段 queryid。
- SQL Text - 完整查询的前100个字符。
- Database - 语句所在的数据库名称 (从 dbid 字段派生)
- IO(s) - 语句读写 (I/O) 所花费的时间, 以秒为单位 (blk\_read\_time + blk\_write\_time 字段)
- Read(s) - 语句读取所花费的时间, 以秒为单位 (blk\_read\_time 字段)
- Write(s) - 语句写入所花费的时间, 以秒为单位 (blk\_write\_time 字段)

- %Total - 此语句的 I/O 时间占集群中所有语句 I/O 时间的百分比
- Reads - 由此语句读取的块数，分为三个子列：
  - Shared - 共享读取次数 (shared\_blks\_read 字段)
  - Local - 本地读取次数 (local\_blks\_read 字段)
  - Temp - 读取临时表 (temp\_blks\_read 字段)
- Writes - 语句写入的块数，分为三个子列：
  - Shared - 共享写入 (shared\_blks\_written 字段)
  - Local - 本地写入 (local\_blks\_written 字段)
  - Temp - 临时表写入 (temp\_blks\_written 字段)
- Elapsed(s) - 执行此查询所花费的时间，以秒为单位 (total\_time 或 total\_exec\_time 字段)
- Executions - 该语句的执行次数 (calls 字段)

#### H. 34. 7. 3. 6. 按共享获取的块数排序的顶级 SQL

按读取和命中块排序的前 `lt_profile.topn` 条语句，有助于检测最多数据处理的语句。

- Query ID - 由数据库、用户和查询文本的哈希值构成的查询标识符。与 `ltcenter` 实用程序兼容。用十六进制表示的 `pg_stat_statements` 原生字段 `qieryid` 以方括号的形式显示。
- SQL Text - 完整查询的前 100 个字符。
- Database - 语句所在数据库的名称 (从 `dbid` 字段派生)
- Fetched - 获取的块数 (表达式: `shared_blks_hit + shared_blks_read`)
- %Total - 该语句获取的块数占整个集群所有语句获取的块数的百分比
- Hits(%) - 从缓存中获取的块数在所有获取的块数中所占的百分比
- Elapsed(s) - 执行此语句所花费的时间，以秒为单位 (total\_time 或 total\_exec\_time +total\_plan\_time 字段)
- Rows - 语句检索或影响的行数 (rows 字段)
- Executions - 该语句的执行次数 (calls 字段)

#### H. 34. 7. 3. 7. 按共享读取的块数排序的顶级 SQL

按共享读取次数排序的前 `lt_profile.topn` 条语句，有助于检测最耗费读取的语句。

- Query ID - 由数据库、用户和查询文本的哈希值构成的查询标识符。与 `ltcenter` 实用程序兼容。用十六进制表示的 `pg_stat_statements` 原生字段 `qieryid` 以方括号的形式显示。
- SQL Text - 完整查询的前 100 个字符。
- Database - 语句所在数据库的名称 (从 `dbid` 字段派生)
- Reads - 该语句的共享读取块数 (shared\_blks\_read 字段)

- %Total - 该语句的共享读取块数占整个集群所有语句共享读取块数的百分比
- Hits(%) - 从缓存中获取的块数在所有获取的块数中所占的百分比
- Elapsed(s) - 执行此语句所花费的时间，以秒为单位 (total\_time 或 total\_exec\_time +total\_plan\_time 字段)
- Rows - 语句检索或影响的行数 (rows 字段)
- Executions - 该语句的执行次数 (calls 字段)

#### H. 34. 7. 3. 8. 按共享脏块数排序的顶级 SQL

按共享脏缓冲区计数排序的前 lt\_profile.topn 条语句，可帮助检测最常更改数据的语句。

- Query ID - 数据库、用户和查询文本的哈希标识符。兼容 ltcenter 工具。以十六进制表示的 pg\_stat\_statements 原生字段 queryid 在方括号中显示。
- SQL文本 - 完整查询的前100个字符。
- 数据库 - 语句所在的数据库名称 (从 dbid 字段派生)
- 脏数据块数 - 该语句修改的共享块数 (shared\_blks\_dirtied 字段)
- %总共 - 该语句修改的共享块数占整个集群所有语句修改的共享块数的百分比
- 命中率(%) - 从所有获取的块中，从缓存中获取的块的百分比
- 耗时(s) - 该语句执行的时间，以秒为单位 (total\_time 或 total\_exec\_time +total\_plan\_time 字段)
- 行数 - 语句检索或影响的行数 (rows 字段)
- 执行次数 - 该语句执行的次数 (calls 字段)

#### H. 34. 7. 3. 9. 按共享块数排序的前 SQL

按写入块数排序的必须执行写入操作的前 lt\_profile.topn 条语句。

- Query ID - 数据库、用户和查询文本的哈希标识符。兼容 ltcenter 工具。以十六进制表示的 pg\_stat\_statements 原生字段 queryid 在方括号中显示。
- SQL文本 - 完整查询的前100个字符。
- 数据库 - 语句所在的数据库名称 (从 dbid 字段派生)
- 写入块数 - 该语句写入的块数 (shared\_blks\_written 字段)
- %总共 - 该语句写入的块数占整个集群所有语句写入的块数的百分比
- %后端写入 - 该语句写入的块数占有后端写入的块数的百分比 (pg\_stat\_bgwriter 视图的 buffers\_backend 字段)
- 命中率(%) - 从所有获取的块中，从缓存中获取的块的百分比
- 耗时(s) - 该语句执行的时间，以秒为单位 (total\_time 或 total\_exec\_time +total\_plan\_time 字段)
- 行数 - 语句检索或影响的行数 (rows 字段)
- 执行次数 - 该语句执行的次数 (calls 字段)

### H. 34. 7. 3. 10. 按WAL大小排序的前SQL

按生成的WAL大小排序的前`lt_profile.topn` 条语句（自 `lt_stat_statements v1.8` 版本起可用）

- Query ID - 数据库、用户和查询文本的哈希标识符。兼容 `ltcenter` 工具。以十六进制表示的 `pg_stat_statements` 原生字段 `queryid` 在方括号中显示。
- SQL文本 - 完整查询的前100个字符。
- 数据库 - 语句所在的数据库名称（从 `dbid` 字段派生）
- WAL - 语句生成的 WAL 数量（`wal_bytes` 字段）
- %Total - 语句生成的 WAL 数量相对于集群总 WAL 数量的百分比（`pg_current_wal_lsn()` 增量）
- Dirtied - 语句脏数据块的数量（`shared_blks_dirtied` 字段）
- WAL full page images - 语句生成的 WAL 全页镜像的总数量（`wal_fpi` 字段）
- WAL records - 语句生成的 WAL 字节数量总和（`wal_bytes` 字段）

### H. 34. 7. 3. 11. 按临时空间使用量排名的 SQL

按照临时 I/O 排序的前 `lt_profile.topn` 条语句，计算方式为 `temp_blks_read`、`temp_blks_written`、`local_blks_read` 和 `local_blks_written` 字段的总和

- Query ID - 查询标识符，由数据库、用户和查询文本的哈希值生成。与 `ltcenter` 实用程序兼容。用十六进制表示的原生 `pg_stat_statements` 字段 `queryid` 在方括号中显示。
- SQL Text - 完整查询的前 100 个字符。
- Database - 语句所在数据库名称（从 `dbid` 字段派生）
- Local fetched - 检索的本地数据块数量（表达式为：`local_blks_hit + local_blks_read`）
- Hits(%) - 从临时缓冲区中获取的本地数据块在所有本地数据块中的百分比
- Local blocks - 临时表中使用的块的 I/O 统计信息
  - Write - 写入的本地数据块数量（`local_blks_written`）
  - %Total - 该语句的 `local_blks_written` 占集群中所有语句的总 `local_blks_written` 的百分比
  - Read - 读取的本地数据块数量（`local_blks_read`）
  - %Total - 该语句的 `local_blks_read` 占集群中所有语句的总 `local_blks_read` 的百分比
- Temp blocks - 操作中使用的块的 I/O 统计信息（例如排序和连接）
  - Write - 写入的临时数据块数量（`temp_blks_written`）
  - %Total - 该语句的 `temp_blks_written` 占集群中所有语句的总 `temp_blks_written` 的百分比
  - Read - 读取的临时数据块数量（`temp_blks_read`）

- %Total - 该语句的 temp\_blks\_read 占集群中所有语句的总 temp\_blks\_read 的百分比
- Elapsed(s) - 语句执行时间，以秒为单位 (total\_time 或 total\_exec\_time +total\_plan\_time 字段)
- Rows - 语句检索或影响的行数 (rows 字段)
- Executions - 该语句的执行次数 (calls 字段)

#### H. 34. 7. 3. 12. 完整的 SQL 文本列表

报告中提到的所有语句的查询文本。您可以在任何统计表中使用 Query ID 链接进入并查看查询文本。

#### H. 34. 7. 4. 模式对象统计信息

报告的此部分包含使用统计收集器视图的顶级数据库对象。

##### H. 34. 7. 4. 1. 按估计顺序扫描的数据量排序的顶级表

按估计顺序扫描读取的数据库顶级表排序。 基于pg\_stat\_all\_tables视图。您可以在此搜索表，可能在其上缺少一些索引。

- Database - 表所在的数据库名称
- Tablespace - 表所在的表空间名称
- Schema - 表所在的模式名称
- Table - 表名称
- ~SeqBytes - 通过顺序扫描读取的估计数据量。 计算方式为报告样本中所有relation size乘以seq\_scan的总和。
- SeqScan - 表上执行的顺序扫描次数 (seq\_scan字段)
- IndexScan - 在此表上启动的索引扫描次数 (idx\_scan字段)
- IndexFetch - 由索引扫描获取的活跃行数 (idx\_tup\_fetch字段)
- Inserted - 插入的行数 (n\_tup\_ins字段)
- Updated - 更新的行数 (包括HOT) (n\_tup\_upd字段)
- Deleted - 删除的行数 (n\_tup\_del字段)
- HOT Updated - HOT更新的行数 (n\_tup\_hot\_upd字段)

##### H. 34. 7. 4. 2. 按获取的块排序的顶级表

获取的块是从磁盘（读取）或共享缓冲区（命中）中处理的块。 此列表中的表按表关系、其索引、表的TOAST（如果存在）以及TOAST索引（如果存在）的获取块总和排序。 此部分可以让您关注具有过多块处理的表。基于pg\_statio\_all\_tables视图中的数据。

- Database - 表所在的数据库名称
- Tablespace - 表所在的表空间名称
- Schema - 表所在的模式名称

- Table - 表名称
- Heap - 关系块获取的统计信息 (heap\_blks\_read + heap\_blks\_hit)
- Index - 获取所有关系索引块的统计信息 (idx\_blks\_read + idx\_blks\_hit)
- TOAST - 获取TOAST表块的统计信息 (toast\_blks\_read + toast\_blks\_hit)
- TOAST-Index - 获取TOAST索引块的统计信息 (tidx\_blks\_read + tidx\_blks\_hit)

此表中的每个统计字段都分为两列:

- Blocks - 获取关系堆、索引、TOAST或TOAST索引的块数
- %Total - 获取关系堆、索引、TOAST或TOAST索引的块数, 作为整个集群中所有块获取的百分比

#### H. 34. 7. 4. 3. 按读取的块排序的顶级表

按块读取排序的顶级表。此列表中的表按表、其索引、表的TOAST (如果存在) 以及TOAST索引 (如果存在) 的块读取总和排序。此部分可以让您关注具有过多块读取的表。基于pg\_statio\_all\_tables视图中的数据。

- Database - 表所在的数据库名称
- Tablespace - 表所在的表空间名称
- Schema - 表所在的模式名称
- Table - 表名称
- Heap - 关于关系的块读取统计信息 (heap\_blks\_read)
- Index - 关于所有关系索引块读取的统计信息 (idx\_blks\_read)
- TOAST - 关于TOAST表块读取的统计信息 (toast\_blks\_read)
- TOAST-Index - 关于TOAST索引块读取的统计信息 (tidx\_blks\_read)

此表中的每个读取统计信息均分为两列:

- Blocks - 关系heap、index、TOAST或TOAST index的块读取数量
- %Total - 关系heap、index、TOAST或TOAST index的块读取, 作为整个集群中所有块读取的百分比

#### H. 34. 7. 4. 4. Top DML表

按DML影响行数排序的前几个表, 即n\_tup\_ins、n\_tup\_upd和n\_tup\_del的总和 (包括TOAST表)。

- Database - 表所在的数据库名称
- Tablespace - 表所在的表空间名称
- Schema - 表所在的schema名称
- Table - 表名
- Inserted - 插入的行数 (n\_tup\_ins字段)

- Updated - 更新的行数 (包括HOT) (n\_tup\_upd字段)
- Deleted - 删除的行数 (n\_tup\_del字段)
- HOT Updated - HOT更新的行数 (n\_tup\_hot\_upd字段)
- SeqScan - 在表上执行的顺序扫描次数 (seq\_scan字段)
- SeqFetch - 通过顺序扫描获取的活跃行数 (seq\_tup\_read字段)
- IndexScan - 在此表上启动的索引扫描次数 (idx\_scan字段)
- IndexFetch - 通过索引扫描获取的活跃行数 (idx\_tup\_fetch字段)

#### H. 34. 7. 4. 5. 按更新/删除元组排序的前几个表

按引起自动清理负载操作数量排序的前几个表, 即n\_tup\_upd和n\_tup\_del的总和 (包括TOAST表)。根据提供的清理和分析运行统计信息, 考虑微调与清理相关的参数。

- Database - 表所在的数据库名称
- Tablespace - 表所在的表空间名称
- Schema - 表所在的schema名称
- Table - 表名
- 更新 - 更新的行数 (包括HOT) (n\_tup\_upd字段)
- HOT更新 - HOT更新的行数 (n\_tup\_hot\_upd字段)
- 删除 - 删除的行数 (n\_tup\_del字段)
- 清理 - 手动清理此表的次数 (不包括VACUUM FULL) (vacuum\_count字段)
- 自动清理 - 自动清理守护程序清理此表的次数 (autovacuum\_count字段)
- 分析 - 手动分析此表的次数 (analyze\_count字段)
- 自动分析 - 自动清理守护程序分析此表的次数 (autoanalyze\_count字段)

#### H. 34. 7. 4. 6. 增长最快的表

按增长排序的顶部表

- 数据库 - 表所在的数据库名称
- 表空间 - 表所在的表空间名称
- 架构 - 表所在的模式名称
- 表 - 表名称
- 大小 - 表大小, 即报告间隔内最后一次采样时的大小
- 增长 - 表增长
- 插入 - 插入的行数 (n\_tup\_ins字段)
- 更新 - 更新的行数 (包括HOT) (n\_tup\_upd字段)
- 删除 - 删除的行数 (n\_tup\_del字段)



- HQT更新 - HQT更新的行数 (n\_tup\_hot\_upd字段)

#### H. 34. 7. 4. 7. 按读取的块排序的索引排名

读取的块是正在从磁盘（读取）或共享缓存区（命中）中处理的块。基于pg\_statio\_all\_indexes视图的数据。

- 数据库 - 索引所在的数据库名称
- 表空间 - 索引所在的表空间名称
- 架构 - 索引所在的模式名称
- 表 - 表名称
- 索引 - 索引名称
- 扫描 - 在索引上执行的扫描次数 (idx\_scan字段)
- 块 - 从此索引中读取的块 (idx\_blks\_read + idx\_blks\_hit)
- %总计 - 该索引读取的块数，占整个集群读取的所有块数的百分比

#### H. 34. 7. 4. 8. 按读取的块数排序的索引排名

按块读取排序的顶部索引。基于pg\_statio\_all\_indexes视图的数据。

- 数据库 - 索引所在的数据库名称
- 表空间 - 索引所在的表空间名称
- 架构 - 索引所在的模式名称
- 表 - 表名称
- 索引 - 索引名称
- 扫描 - 在索引上执行的扫描次数 (idx\_scan 字段)
- 块读取 - 从该索引读取的磁盘块数 (idx\_blks\_read)
- %总数 - 该索引的块读取占整个集群所有块读取的百分比
- 命中率(%) - 该索引获取的缓存块在所有获取该索引的块中所占的百分比

#### H. 34. 7. 4. 9. 增长最快的索引

按增长排序的前索引

- 数据库 - 索引所在的数据库名称
- 表空间 - 索引所在的表空间名称
- 模式 - 索引所在的模式名称
- 表 - 表名称
- 索引 - 索引名称
- 索引 - 索引统计信息

- 大小 - 索引大小，报告间隔中最后一次采样时的大小
- 增长 - 报告间隔期间的索引增长
- 表 - 底层表的统计信息
  - 插入 - 插入到底层表中的行数 (n\_tup\_ins 字段)
  - 更新 - 在底层表中更新的行数 (不包括HOT更新) (n\_tup\_upd - n\_tup\_hot\_upd)
  - 删除 - 从底层表中删除的行数 (n\_tup\_del 字段)

#### H. 34. 7. 4. 10. 未使用的索引

报告间隔期间未被扫描的索引，按照底层表上的DML操作排序，可能导致索引支持。排除约束索引。

- 数据库 - 索引所在的数据库名称
- 表空间 - 索引所在的表空间名称
- 模式 - 索引所在的模式名称
- 表 - 表名称
- 索引 - 索引名称
- 索引 - 索引统计信息
  - 大小 - 索引大小，报告间隔中最后一次采样时的大小
  - 增长 - 报告间隔期间的索引增长
- Table - 底层表的统计信息
  - Inserted - 底层表中插入的行数 (n\_tup\_ins 字段)
  - Updated - 底层表中更新的行数 (不包括HOT更新) (n\_tup\_upd - n\_tup\_hot\_upd)
  - Deleted - 底层表中删除的行数 (n\_tup\_del 字段)

#### H. 34. 7. 5. 用户函数统计信息

本报告部分根据pg\_stat\_user\_functions视图，列出了集群中最常用的函数。

##### H. 34. 7. 5. 1. 按总时间排序的常用函数

按经过的时间排序的常用函数。

- Database - 函数所在的数据库名称
- Schema - 函数所在的 schema 名称
- Function - 函数名称
- Executions - 调用此函数的次数 (calls 字段)
- Time (s) - 函数时间统计，单位为秒
- Total - 此函数和所有由它调用的其他函数所花费的总时间 (total\_time 字段)

- Self - 此函数本身所花费的总时间，不包括由它调用的其他函数 (self\_time 字段)
- Mean - 单次函数执行的平均时间
- Mean self - 单次函数执行的平均 self 时间

#### H. 34. 7. 5. 2. 按执行次数排序的常用函数

按执行次数排序的常用函数。

- Database - 函数所在的数据库名称
- Schema - 函数所在的 schema 名称
- Function - 函数名称
- Executions - 调用此函数的次数 (calls 字段)
- Time (s) - 函数时间统计，单位为秒
  - Total - 此函数和所有由它调用的其他函数所花费的总时间 (total\_time 字段)
  - Self - 此函数本身所花费的总时间，不包括由它调用的其他函数 (self\_time 字段)
  - Mean - 单次函数执行的平均时间
  - Mean self - 单次函数执行的平均 self 时间

#### H. 34. 7. 5. 3. 按总时间排序的常用触发器函数

按经过的时间排序的常用触发器函数。

- Database - 函数所在的数据库名称
- Schema - 函数所在的 schema 名称
- Function - 函数名称
- Executions - 调用此函数的次数 (calls 字段)
- Time (s) - 函数时间统计，单位为秒
  - Total - 此函数和所有由它调用的其他函数所花费的总时间 (total\_time 字段)
  - Self - 此函数本身所花费的总时间，不包括由它调用的其他函数 (self\_time 字段)
  - Mean - 单次函数执行的平均时间
  - Mean self - 单次函数执行的平均 self 时间

#### H. 34. 7. 6. Vacuum 相关统计信息

##### H. 34. 7. 6. 1. 按 VACUUM 操作排序的顶级表

按处理的 VACUUM (手动和自动) 排序的表格排名

- Database - 表格所在的数据库名称
- Tablespace - 表格所在的表空间名称
- Schema - 表格所在的 schema 名称

- Table - 表格名称
- Vacuum count - 手动 VACUUM (不包括 VACUUM FULL) 的次数, 对此表进行了多少次清理 (vacuum\_count 字段)
- Autovacuum count - 自动 VACUUM 守护程序对此表进行过多少次清理 (autovacuum\_count 字段)
- Inserted - 插入的行数 (n\_tup\_ins 字段)
- Updated - 更新的行数 (包括 HOT 更新) (n\_tup\_upd 字段)
- Deleted - 删除的行数 (n\_tup\_del 字段)
- HOT Updated - HOT 更新的行数 (n\_tup\_hot\_upd 字段)

#### H. 34. 7. 6. 2. 按分析操作排序的表格排名

按分析运行 (手动和自动) 次数排序的表格排名

- Database - 表格所在的数据库名称
- Tablespace - 表格所在的表空间名称
- Schema - 表的模式名称
- Table - 表格名称
- Analyze count - 手动分析此表的次数 (analyze\_count 字段)
- Autoanalyze count - 自动 VACUUM 守护程序分析此表的次数 (autoanalyze\_count 字段)
- Inserted - 插入的行数 (n\_tup\_ins 字段)
- Updated - 更新的行数 (包括 HOT 更新) (n\_tup\_upd 字段)
- Deleted - 删除的行数 (n\_tup\_del 字段)
- HOT Updated - HOT 更新的行数 (n\_tup\_hot\_upd 字段)

#### H. 34. 7. 6. 3. 按估计 VACUUM I/O 负载排序的索引排名

此表提供了由表格索引引起的隐式 VACUUM 负载的估计值。以下是按底层表格上执行的 VACUUM 次数乘以索引大小排序的索引排名。

- Database - 索引所在的数据库名称
- Tablespace - 索引所在的表空间名称
- Schema - 索引所在的 schema 名称
- Table - 表格名称
- Index - 索引名称
- ~Vacuum bytes - 估计的 VACUUM 负载, 计算公式为 (vacuum\_count + autovacuum\_count) \* index\_size
- Vacuum count - 手动 VACUUM (不包括 VACUUM FULL) 的次数, 对此表进行了多少次清理 (vacuum\_count 字段)

- Autovacuum count - 自动 VACUUM 守护程序对此表进行过多少次清理 (autovacuum\_count 字段)
- Index size - 报告期间的平均索引大小
- Relsize - 报告期间的平均关系大小

#### H. 34. 7. 6. 4. 按死元组比率排序的表格排名

此部分包含最后一次 VACUUM 运行的修改后的表格。统计数据有效于报告期间的最后一个样本。基于 pg\_stat\_all\_tables 视图。

按死元组比率排序，大小为 5 MB 及以上的表格排名。

- Database - 表格所在的数据库名称
- Schema - 表格所在的 schema 名称
- Table - 表格名称
- Live - 估计的活跃行数 (n\_live\_tup)
- Dead - 估计的死亡行数 (n\_dead\_tup)
- %Dead - 表格中死亡行数占表格所有行数的百分比
- Last AutoVacuum - 自动 VACUUM 守护程序最后一次清理此表的时间 (last\_autovacuum)
- Size - 表格大小，即最后一次报告样本时的大小。

#### H. 34. 7. 6. 5. 按修改元组比率排序的表格排名

此部分包含最后一次 VACUUM 运行的修改后的表格。统计数据有效于报告期间的最后一个样本。基于 pg\_stat\_all\_tables 视图。

按修改元组比率排序的大小为5MB及以上的顶级表。

- Database - 表所在的数据库名称。
- Schema - 表所在的模式名称。
- Table - 表名。
- Live - 估计的活跃行数 (n\_live\_tup)。
- Dead - 估计的死亡行数 (n\_dead\_tup)。
- Mod - 自上次分析此表以来修改的行数的估计值 (n\_mod\_since\_analyze)。
- %Mod - 表中修改的行数占表中所有行的百分比。
- Last AutoAnalyze - 此表上次由自动清理进程分析的时间。
- Size - 表格大小，即最后一次报告样本时的大小。

#### H. 34. 7. 7. 报告间隔期间的群集设置

报告的此部分包含LightDB GUC参数和函数  
 数version()、pg\_postmaster\_start\_time()、pg\_conf\_load\_time()以及pg\_control\_system()函数的system\_identifier字段在报告间隔期间的值。

- Setting - 参数名称。

- `reset_val` - `pg_settings`视图的`reset_val`字段。加粗字体用于显示在报告间隔期间更改的设置。
- `Unit` - 设置单位。
- `Source` - 定义此设置的配置文件，分号后的行号。
- `Notes` - 此字段将包含在报告间隔期间首次观察到此值的样本的时间戳。

### H. 34. 7. 8. 报告间隔期间的内核设置

报告的此部分包含几个重要的Linux内核参数。

- `Setting` - 参数名称。
- `Value` - 参数的值。
- `Source` - 定义此设置的配置文件。

### H. 34. 7. 9. 需要记住的内容...

1. LightDB在执行完成之后收集执行统计信息。如果语句的单个执行持续多个样本，它将仅影响最后一个样本（在完成时）。您无法获取仍在运行的语句的统计信息。此外，像`vacuum`和`checkpointer`这样的维护流程仅在完成时更新统计信息。
2. 重置任何LightDB统计信息可能会影响下一个样本的准确性。
3. 对关系的独占锁与计算关系大小冲突。如果任何会话持有`AccessExclusiveLock`，样本将不会收集这些关系的大小信息。但是，会话可以在样本处理期间对关系进行`AccessExclusiveLock`。为了解决这个问题，将`lock_timeout`设置为3秒，因此如果`take_sample()`函数在3秒内无法获取锁，它将失败，不会生成样本。

## H. 34. 8. PSH报告的部分

本部分描述了报告表及其列。

### H. 34. 8. 1. 最常见的等待事件

#### H. 34. 8. 1. 1. 最常见的用户事件

此表包含按等待时间排序的前`lt_profile.topn`个用户等待事件。

- `Wait Event` - 进程正在等待的特定事件。
- `Wait Event Type` - 此事件的类型。
- `Wait Time(s)` - 所有此事件的总等待时间。
- `%Total Wait Time` - 所有此事件的总等待时间与群集中所有事件的总等待时间的百分比。

#### H. 34. 8. 1. 2. 最常见的后台事件

此表包含按等待时间排序的前`lt_profile.topn`个后台等待事件。

- `Wait Event` - 进程正在等待的特定事件。
- `Wait Event Type` - 此事件的类型。
- `Wait Time(s)` - 所有此事件的总等待时间。

- %Total Wait Time - 所有此事件的总等待时间与群集中所有事件的总等待时间的百分比。

## H. 34. 8. 2. 最常见的SQL语句

### H. 34. 8. 2. 1. 带顶部事件的顶部SQL

按等待时间排序的顶部等待事件，用于按pg\_stat\_statements视图的 total\_exec\_time字段排序的顶部语句。

- 查询ID - 数据库、用户和查询文本的哈希值作为查询标识符。与 ltcenter实用程序兼容。在方括号中显示十六进制表示的pg\_stat\_statements 字段queryid。
- SQL文本 - 完整查询的前100个字符。
- Exec (s) - 执行此语句所花费的时间 (total\_exec\_time 字段)。
- %总计 - 此语句的执行时间占群集中所有语句的总经过时间的百分比
- 等待事件 - 进程等待的具体事件。
- 等待时间 (s) - 此查询中事件的总等待时间。

### H. 34. 8. 2. 2. SQL文本的完整列表

报告中提到的所有语句的查询文本。您可以在任何统计表格中使用查询ID链接进入该页面，查看查询文本。

## H. 35. lt\_qualstats

lt\_qualstats是LightDB的一个扩展，用于统计WHERE语句和JOIN子句中出现的谓词的统计信息。

如果您想要分析数据库中最常执行的谓词，那么使用此功能将会很有帮助。

此外，它还可以帮助您识别相关列，通过识别哪些列最常一起查询。

该扩展通过查找查询中已知的模式来进行工作。目前，已知的模式包括：

- 二元OpExpr，其中至少一侧是来自表格的列。在可能的情况下，谓词将被交换，以将CONST OP VAR表达式转换为VAR COMMUTED\_OP CONST。AND和OR表达式成员被视为单独的条目。例如：WHERE column1 = 2, WHERE column1 = column2, WHERE 3 = column3。
- ScalarArrayOpExpr，其中左侧是一个VAR，右侧是一个数组常量。这将针对数组中的每个元素计数一次。例如：WHERE column1 IN (2, 3) 将被计为(column1, '=')操作符对的2个出现次数。
- BooleanTest，其中表达式是一个简单的布尔列引用。例如：WHERE column1 IS TRUE。请注意，像WHERE column1、WHERE NOT column1这样的子句尚未由lt\_qualstats处理。

该扩展还会保存每个执行的不同queryid的第一个查询文本，原封不动地保存，最多保存lt\_qualstats.max 条目。

请注意，当 LightDB 服务器重新启动时，收集的数据不会被保存。

### H. 35. 1. Configuration

可以在lightdb.conf中配置以下GUC参数：

- lt\_qualstats.enabled (布尔类型，默认值为true)：是否启用lt\_qualstats。

- `lt_qualstats.track_constants` (布尔类型, 默认值为true): 是否应将每个常量值单独跟踪。禁用此GUC将显著减少跟踪谓词所需的条目数。
- `lt_qualstats.max`: 跟踪的谓词和查询文本的最大数量(默认为1000)。
- `lt_qualstats.resolve_oids` (布尔类型, 默认值为false): 是否应在查询时解析oid, 或者只存储oid。启用此参数可以使数据分析更加容易, 因为不需要连接到执行查询的数据库, 但它会占用更多的空间(每个条目624字节而不是176字节)。此外, 这将需要一些不是免费的目录查找。
- `lt_qualstats.track_pg_catalog` (布尔类型, 默认值为false): 是否应在pg\_catalog架构中的对象上计算谓词。
- `lt_qualstats.sample_rate` (双精度浮点数, 默认值为-1): 应采样的查询的比例。例如, 0.1表示只对十个查询中的一个进行采样。默认值(-1)表示自动, 结果为1 / `max_connections`, 因此从统计上来说, 并发问题很少发生。

## H. 35. 2. 更新扩展

请注意, 由于所有扩展都配置在 `shared_preload_libraries` 中, 因此大多数更改仅在使用新共享库重新启动LightDB时应用。扩展对象本身仅提供SQL包装器以访问内部数据结构。

还请注意, `lt_qualstats`不提供扩展升级脚本, 因为在创建的任何对象中都没有保存数据。因此, 您需要先删除扩展, 然后再创建它以获得新版本。

## H. 35. 3. 使用方法

- 在任何数据库中创建扩展:

```
CREATE EXTENSION lt_qualstats;
```

### H. 35. 3. 1. 函数

该扩展定义了以下函数:

- `pg_qualstats`: 返回每个限定符的计数, 由表达式哈希标识。此哈希标识每个表达式。
  - `userid`: 执行查询的用户的oid。
  - `dbid`: 执行查询的数据库的oid。
  - `lrelid, lattnum`: 如果有, 则为左侧VAR的关系和属性编号的oid。
  - `opno`: 表达式中使用的运算符的oid。
  - `rrelid, rattnum`: 如果有, 则为右侧VAR的关系和属性编号的oid。
  - `qualid`: 如果有, 则为父“AND”表达式的标准化标识符。排除常量计算此标识符。这对于识别一起使用的谓词非常有用。
  - `uniquequalid`: 如果有, 则为父“AND”表达式的唯一标识符。包括常量在内计算此标识符。
  - `qualnodeid`: 此简单谓词的标准化标识符。排除常量计算此标识符。
  - `uniquequalnodeid`: 此简单谓词的唯一标识符。此标识符包括常量在内进行计算。
  - `occurrences`: 谓词被调用的次数, 即相关查询执行的次数。



- `execution_count`: 谓词执行的次数，即它处理的行数。
- `nbfiltered`: 谓词丢弃的元组数量。
- `constant_position`: 常量在原始查询字符串中的位置，由解析器报告。
- `queryid`: 如果安装了 `pg_stats_statements`，则为标识此查询的 `queryid`，否则为 `NULL`。
- `constvalue`: 右侧常量的字符串表示，如果有的话，截断为80个字节。需要超级用户或 `pg_read_all_stats` 成员，否则将显示“”。
- `eval_type`: 评估类型。'f' 表示在扫描后评估的谓词，'i' 表示索引谓词。

示例:

```

ro=# select * from pg_qualstats;
      userid | dbid | lrelid | lattnum | opno |
 rrelid | rattnum | qualid | uniquequalid | qualnodeid | uniquequalnodeid |
 | occurrences | execution_count | nbfiltered | constant_position |
 queryid | constvalue | eval_type
-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
          10 | 16384 | 16385 | 2 | 98 |
 |      |      |      |      |      |
 |      1 |      |      |      |      |
 'line 1'::text | f
          10 | 16384 | 16391 | 2 | 98 |
 16385 | 2 |      |      |      |
 |      1 |      |      |      |      |
 |      |      | 0 | 0 |      |
 |      |      |      |      |      |
 |      |      |      |      |      |
 
```

- `pg_qualstats_index_advisor(min_filter, min_selectivity, forbidden_am)`: 执行全局索引建议。默认情况下，仅考虑过滤至少1000行和平均30%行的谓词，但可以作为参数传递。如果您想要避免某些索引访问方法，可以提供索引访问方法数组。

示例:

```

SELECT v
FROM json_array_elements(
  pg_qualstats_index_advisor(min_filter => 50)-
>'indexes') v
ORDER BY v::text COLLATE "C";
v
-----
"CREATE INDEX ON public.adv USING btree (id1)"
"CREATE INDEX ON public.adv USING btree (val, id1,
id2, id3)"
"CREATE INDEX ON public.pgqs USING btree (id)"
(3 rows)

SELECT v
FROM json_array_elements(

```

```

                                pg_qualstats_index_advisor(min_filter => 50)-
>'unoptimised') v
                                ORDER BY v::text COLLATE "C";
                                v
                                -----
                                "adv. val ~* ?"
                                (1 row)

```

- `pg_qualstats_deparse_qual`: 以 `tablename.columnname operatorname ?` 的形式格式化存储的谓词。这主要是为了全局索引建议。
- `pg_qualstats_get_idx_col`: 针对给定的谓词，检索底层列名和所有可能的操作符类。这主要是为了全局索引建议。
- `pg_qualstats_get_qualnode_rel`: 针对给定的谓词，返回底层表的完全限定名称。这主要是为了全局索引建议。
- `pg_qualstats_example_queries`: 返回所有存储的查询文本。
- `pg_qualstats_example_query`: 如果有，则返回给定 `queryid` 的存储查询文本，否则为 `NULL`。
- `pg_qualstats_names`: 返回所有存储的谓词名称。
- `pg_qualstats_reset`: 重置内部计数器并忘记每个遇到的谓词。

### H. 35. 3. 2. 视图

此外，该扩展还在 `pg_qualstats` 函数之上定义了一些视图：

- `pg_qualstats`: 通过当前数据库过滤对 `pg_qualstats()` 的调用。
- `pg_qualstats_pretty`: 执行适当的连接，以显示 `pg_qualstats` 视图中每个属性的可读聚合形式。

示例：

```

                                ro=# select * from pg_qualstats_pretty;
                                left_schema | left_table | left_column |
operator | right_schema | right_table | right_column | occurrences |
execution_count | nbfiltered
                                -----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
pg_catalog.= | public | ltbench_accounts | aid | 5 |
5000000 | 4999995
pg_catalog.= | public | ltbench_tellers | tid | 10 |
10000000 | 9999990
pg_catalog.= | public | ltbench_branches | bid | 10 |
2000000 | 1999990
pg_catalog.= | public | t1 | id | 1 |
10000 | public | t2 | id_t1 | 9999

```

- `pg_qualstats_all`: 对每个属性/操作符对的计数求和, 无论其作为操作数 (左侧或右侧) 的位置如何, 都将用于AND子句中的属性分组在一起。

示例:

```

ro=# select * from pg_qualstats_all;
   dbid | relid | userid | queryid | attnums | opno |
-----+-----+-----+-----+-----+-----+
qualid | occurrences | execution_count | nbfiltered | qualnodeid |
-----+-----+-----+-----+-----+
      1 |          1 |         100000 |          99999 | 115075651 |
      2 |          2 |         16384 |          16391 |          0 | 497379130 |

```

- `pg_qualstats_by_query`: 仅按queryid汇总的形式VAR OPERATOR CONSTANT返回谓词。

## H. 36. `lt_show_plans`

`lt_show_plans` 是一个模块, 用于显示所有当前正在运行的SQL语句的查询计划。您可以选择计划的输出格式: TEXT或JSON。

当服务器启动时, `lt_show_plans`会在共享内存中创建一个哈希表, 用于临时存储查询计划。哈希表的大小不能更改, 因此如果哈希表已满, 则不会存储计划。

### H. 36. 1. 配置参数

`lt_show_plans.plan_format`

它控制查询计划的输出格式。它可以选择为text或json。默认值为text。

`lt_show_plans.max_plan_length`

它设置查询计划的最大长度。默认值为8192 [字节]。请注意, 该参数必须设置为整数。

`lt_show_plans.show_level`

它控制查询计划的级别。您可以选择其中之一: all、top和none。“all”显示查询计划的所有级别。例如, 当您执行由PL/pgSQL定义的函数时, 将显示调用SQL语句 (级别0) 和函数内部的内部SQL语句 (级别1)。“top”显示查询计划的顶级别。“none”不存储查询计划, 因此`lt_show_plans`视图不显示任何内容。默认值为none。

### H. 36. 2. 用法

在你的 `lightdb.conf` 中添加如下行 (参见上文 `lt_show_plans.show_level`):

```

shared_preload_libraries = 'lt_show_plans'
lt_show_plans.show_level = 'top'

```

在启动服务器后, 必须执行以下显示的 `CREATE EXTENSION` 语句:

```
testdb=# CREATE EXTENSION lt_show_plans;
```

执行后, 您必须调用下面显示的启用函数:

```
testdb=# select pg_show_plans_enable();
```

通过执行以下查询，可以显示当前运行的 SQL 语句的查询计划及相关信息：

```
testdb=# SELECT * FROM pg_show_plans;
 pid | level | userid | dbid | plan
-----+-----+-----+-----+-----
11473 |    0 |    10 | 16384 | Function Scan on pg_show_plans
(cost=0.00..10.00 rows=1000 width=56)
11504 |    0 |    10 | 16384 | Function Scan on print_item (cost=0.25..10.25
rows=1000 width=524)
11504 |    1 |    10 | 16384 | Result (cost=0.00..0.01 rows=1 width=4)
(3 rows)
```

pg\_show\_plans 视图：

- pid: 正在运行查询的进程的 pid。
- level: 运行查询的查询级别。顶级是 0。
- userid: 运行查询的用户的用户 ID。
- dbid: 正在运行查询的数据库的数据库 ID。
- plan: 正在运行查询的查询计划。

如果您需要正在运行的 SQL 语句的查询计划以及相应的查询字符串，则可以执行以下查询，该查询结合了 pg\_show\_plans 和 pg\_stat\_activity：

```
testdb=# \x
Expanded display is on.
testdb=# SELECT p.pid, p.level, p.plan, a.query FROM pg_show_plans p
LEFT JOIN pg_stat_activity a
ON p.pid = a.pid AND p.level = 0 ORDER BY p.pid, p.level;
-[ RECORD
1 ]-----
pid | 11473
level | 0
plan | Sort (cost=72.08..74.58 rows=1000 width=80)
      +
      | Sort Key: pg_show_plans.pid, pg_show_plans.level
      +
      | -> Hash Left Join (cost=2.25..22.25 rows=1000 width=80)
      +
      | Hash Cond: (pg_show_plans.pid = s.pid)
      +
      | Join Filter: (pg_show_plans.level = 0)
      +
      | -> Function Scan on pg_show_plans (cost=0.00..10.00 rows=1000
width=48)
      +
      | -> Hash (cost=1.00..1.00 rows=100 width=44)
      +
      | -> Function Scan on pg_stat_get_activity s
(cost=0.00..1.00 rows=100 width=44)
query | SELECT p.pid, p.level, p.plan, a.query FROM pg_show_plans p
      +
      | LEFT JOIN pg_stat_activity a
      +
```

```

        | ON p.pid = a.pid AND p.level = 0 ORDER BY p.pid, p.level;
-[ RECORD
 2 ]-----
pid | 11517
level | 0
plan | Function Scan on print_item (cost=0.25..10.25 rows=1000 width=524)
query | SELECT * FROM print_item(1,20);
-[ RECORD
 3 ]-----
pid | 11517
level | 1
plan | Result (cost=0.00..0.01 rows=1 width=4)
query |

```

## H. 36. 3. Functions

- `pg_show_plans_disable()` 禁用此功能。只有超级用户才能执行它。
- `pg_show_plans_enable()` 启用此功能。只有超级用户才能执行它。

## H. 37. 1t\_sm

### H. 37. 1. 什么是1t\_sm?

`1t_sm`扩展为LightDB提供SM密码学函数，包括sm2、sm3、sm4算法。可以使用上述算法对数据进行加密和解密。

请注意，当前的`1t_sm`扩展是试验版本。

### H. 37. 2. 函数用法

#### H. 37. 2. 1. SM2

SM2是非对称加密，有一对密钥，私钥用于加密，公钥用于解密。

SM2共有四个函数：

- `gm_sm2_genprikey(local_storage bool) return text` 生成SM2私钥，用于加密数据。输入参数 `true` 表示将密钥存储在数据库中，此时返回值为`true`。输入参数 `false` 表示在数据库重启时从外部导入密钥，返回值为文本类型的私钥字符串。

```

-- Generate SM2 private key using local storage
SELECT gm_sm2_genprikey(true);
gm_sm2_genprikey
-----
true

-- Generate SM2 private key using external import
SELECT gm_sm2_genprikey(false);
gm_sm2_genprikey
-----
-----BEGIN EC PRIVATE KEY-----+
MHCQAQEEIMattGsrutKOT7YFtJ/nRL3HzUTSoBuW+178TYuthSEXoAoGCCqBHM9V+
AYItouQDQgAEv12xwVT9SD4/qOqB8EYZkbMy12APj3sOBHq0krHUS1Q0UbV0cJ1s+
v1JkxGQSJ+EGXbmuhRErwiAu9gTd8+umlw==+
-----END EC PRIVATE KEY-----+

```

- `gm_sm2_genpubkey((local_storage bool) return text` 生成SM2公钥，用于解密数据。输入参数 `true` 表示将密钥存储在数据库中，此时返回值为`true`。输入参数 `false` 表示在数据库重启时从外部导入密钥，返回值为文本类型的私钥字符串。注意，在使用此函数生成公钥之前，必须使用 `gm_sm2_genprikey()` 函数生成私钥。

```
-- Generate SM2 public key using local storage
SELECT gm_sm2_genpubkey(true);
      gm_sm2_genprikey
-----
      true

-- Generate SM2 public key using external import
SELECT gm_sm2_genpubkey(false);
      gm_sm2_genpubkey
-----
-----BEGIN PUBLIC KEY-----+
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEv12xwVT9SD4/q0qB8EYZkbMy12AP+
j3s0BHq0krHUS1Q0UbV0cJ1sv1JkxGQSJ+EGXbmuhRErwiAu9gTd8+umlw==+
-----END PUBLIC KEY-----+
```

- `gm_sm2_encrypt(plaintext bytea) return bytea` 将明文加密为密文，输入参数为明文，返回值为密文。

```
-- Encrypt plaintext to generate ciphertext
select gm_sm2_encrypt('恒生电子LightDB_001');

      gm_sm2_encrypt
```

```
-----
\x043f115fde4ad84ca331e6d6edbeb806d09ffa6094f8d52a57a39c7c6c587bcd06eb72146f3d503923af813fa6
```

- `gm_sm2_decrypt(ciphertext bytea) return bytea` 将密文解密为明文，输入参数为密文，返回值为明文。

```
-- Decrypt ciphertext to generate plaintext
select
gm_sm2_decrypt('\x043f115fde4ad84ca331e6d6edbeb806d09ffa6094f8d52a57a39c7c6c587bcd06eb72146f3d503923af813fa6');
      gm_sm2_decrypt
-----
\xe68192e7949fe794b5e5ad904c6967687444425f303031

-- Convert the decrypted binary plaintext to display
select convert_from('\xe68192e7949fe794b5e5ad904c6967687444425f303031',
'SQL_ASCII');
      convert_from
-----
恒生电子LightDB_001
```

## H. 37. 2. 2. SM3

SM3是一种密码哈希算法，主要用于数字签名和验证、消息认证码生成和验证、随机数生成等。

SM3共有三个函数：

- `gm_sm3_encrypt(plaintext bytea) return bytea` 将明文信息加密为摘要信息，输入参数为明文信息，返回值为摘要信息。

```
-- plaintext message to generate digest message
select gm_sm3_encrypt('LightDB_002');
       gm_sm3_encrypt
-----
\x9de35b686bec17533e9ded7abfb039320bb08abbcec74eb0f44cf52b09e255c1
```

### H. 37. 2. 3. SM4

SM4是对称加密，使用CTR-128位模式，有一个密钥用于加密和解密。

SM4共有三个函数：

- `gm_sm4_genkey((local_storage bool) return text` 生成SM4密钥，用于加密和解密数据。输入参数 `true` 表示将密钥存储在数据库中，此时返回值为`true`。输入参数 `false` 表示在数据库重启时从外部导入密钥，返回值为文本类型的密钥字符串。

```
-- Generate SM4 key using local storage
select gm_sm4_genkey(true);
       gm_sm4_genkey
-----
true

-- Generate SM4 key using external import
select gm_sm4_genkey(false);
       gm_sm4_genkey
-----
A98B1904E5A74CDA5B2135519B941703
```

- `gm_sm4_encrypt(plaintext bytea) return bytea` 将明文加密为密文，输入参数为明文，返回值为密文。

```
-- Encrypt plaintext to generate ciphertext
select gm_sm4_encrypt('恒生电子LightDB_003');
       gm_sm4_encrypt
-----
\x33e7117d0ce6f38ec2b78bfab08b991628771bedeeecf8
```

- `gm_sm4_decrypt(ciphertext bytea) return bytea` 将密文解密为明文，输入参数为密文，返回值为明文。

```
-- Decrypt ciphertext to generate plaintext
select gm_sm4_decrypt('\x33e7117d0ce6f38ec2b78bfab08b991628771bedeeecf8');
       gm_sm4_decrypt
-----
\xe68192e7949fe794b5e5ad904c6967687444425f303033

-- Convert the decrypted binary plaintext to display
```

```
select convert_from('\xe68192e7949fe794b5e5ad904c6967687444425f303033',
'SQL_ASCII');
convert_from
-----
恒生电子LightDB_003
```

### H. 37. 3. 密钥管理

lt\_sm支持两种密钥管理方法:

- 密钥存储在本地，使用密钥生成函数时输入参数为true。 生成密钥后，可以使用加密和解密函数。
- 密钥从外部导入，使用生成密钥函数的输入参数为false。 如果数据库重启，支持外部导入密钥文件。

```
-- When using lt_ctl, specify the -Y parameter to pass in the key file
lt_ctl -D ./data -Y ./keyfile start
```

其中，密钥文件需要包含 username、database、sm2私钥、sm2公钥、sm4密钥 关键词，使用 : 分隔关键词和值:

```
-- key file format, the corresponding pre-keyword and ':' must contain
username:lightdb
database:postgres
sm2 private key:-----BEGIN EC PRIVATE KEY-----
MHcCAQEIEIC4cBmYiht3Kf4e/xQMNR3C2pLkafZ8Lm91Uxae937SjoAoGCCqBHM9V
AYItouQDQgAEdqAwhgrpt7NNBVKWuyXy81tI1c11YQhyum1GE3G4QK31fVBCgSYF
05+fNfThT1ppVC+Q11wJ/QJmciYVEDzY8A==
-----END EC PRIVATE KEY-----
sm2 public key:-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoEcz1UBgi0DQgAEdqAwhgrpt7NNBVKWuyXy81tI1c11
YQhyum1GE3G4QK31fVBCgSYF05+fNfThT1ppVC+Q11wJ/QJmciYVEDzY8A==
-----END PUBLIC KEY-----
sm4 key:2DB5E313E60A123DBB1A517B5355A7AC
```

## H. 38. lt\_sql\_inspect

lt\_sql\_inspect 模块提供两个功能: SQL 最佳实践和 SQL 拦截。

### H. 38. 1. SQL 最佳实践

#### H. 38. 1. 1. 概述

该功能将提供用于规范 SQL 编写的规则。

具体而言，它分为两部分: LightDB DDL 规范检查和 LightDB DML 规范检查。

#### H. 38. 1. 2. 配置参数

lightdb\_best\_practice\_mode (String)

lightdb\_best\_practice\_mode 用于指定用于检查 SQL 的规则。



目前支持以下规则：

keyword\_not\_allowed, bad\_datatype, must\_have\_pk, must\_have\_gmt, no\_more\_100cols, no\_paging。

可以在单个会话中更改此设置。

lightdb\_best\_practice\_level (String)

lightdb\_best\_practice\_level 用于指定检查失败时的消息级别。

它用于 SQL 最佳实践和 SQL 拦截。

可以在单个会话中更改此设置。

### H. 38.1.3. LightDB DDL 规范检查

在使用 DDL 语句时，LightDB 会首先检查您的语句是否符合规范。

目前，有五个检查：关键字检查（包括表名、字段名、库名和索引名）、字段类型检查（float、double、text、real、uuid、bytea、boolean、bit、character）、表是否必须包含主键、表是否包含创建时间和最后修改时间字段，以及操作的列数不超过 100。DDL 语句规范支持：

```
CREATE DATABASE...
CREATE TABLE...
CREATE INDEX...
ALTER TABLE...
```

关键字列表可以参考 “[www.hs.net/lightdb/docs/current/sql-keywords-appendix.html](http://www.hs.net/lightdb/docs/current/sql-keywords-appendix.html)”

当您使用非标准语句时，将直接报告警告或错误。 例如：

```
create database text;
WARNING: LightDB DDL check warn! 不允许使用关键字 [db name]:text

create table text(id int);
WARNING: LightDB DDL check warn! 不允许使用关键字 [table name]:text
WARNING: LightDB DDL check warn! 不允许使用关键字 [column name]:id
WARNING: LightDB DDL check warn! 必须包含主键!
WARNING: LightDB DDL check warn! 必须包含 gmt_create 或 gmt_modified 字段!
```

在这里，'text' 是一个非标准的数据库名称，因此报告了一个警告。 在创建表时，表名不是标准的，列名不是标准的，不包含主键，不包含创建时间和最后修改时间的字段，所以会发出警告。当然，还有 100 个操作列的上限，这里未列出。

默认情况下，DDL 语法检查未启用，有两种方法可以启用它。 全局修改方式是通过修改配置文件：

```
lightdb_best_practice_level = 'warn'
lightdb_best_practice_mode = 'all'
```

lightdb\_best\_practice\_level 的值为 'warn' 表示错误级别为警告。'error' 表示错误。当检测到非标准语句时，直接报告错误：

```
create database text;
ERROR: LightDB DDL check error! 不允许使用关键字 [db name]:text
```

如果 lightdb\_best\_practice\_mode 配置为 'all'，则表示同时检查 5 种类型。当然，您可以随意组合要配置的检查，例如：

```
lightdb_best_practice_mode = 'keyword_not_allowed,must_have_pk'
```

上述配置表示仅检查关键字和主键。

另一种配置方法是会话级别修改：

```
set lightdb_best_practice_level to 'warn';
set lightdb_best_practice_mode to 'all';
```

这里有五种完全开放的方式，还有另一种方式：

```
set lightdb_best_practice_mode to
'keyword_not_allowed,bad_datatype,must_have_pk,must_have_gmt,no_more_100cols';
```

### H. 38. 1. 4. LightDB DML 规范检查

在使用 DML 语句时，LightDB 会首先检查您的语句是否符合规范。

目前，只有一个检查，即无分页检查：当用户 SQL 没有指定查询条件或分页时，提醒用户。下面详细描述了这些检查的用法。

- no\_paging check。在以下场景中需要检查：
  - 查询语句没有 where 条件、没有 limit 或没有 offset。
  - 更新语句没有 where 条件。
  - 删除语句没有 where 条件。

如何使用它？需要在配置文件中配置 `lightdb_best_practice_level` 和 `lightdb_best_practice_mode`。

默认情况下，DML 语法检查未启用。全局修改方式是通过修改配置文件：

```
lightdb_best_practice_level = 'error'
lightdb_best_practice_mode = 'all'
```

`lightdb_best_practice_level` 的值为 'warn' 表示错误级别为警告。'error' 表示错误。当检测到非标准语句时，直接报告错误：

```
select * from table1;
ERROR: LightDB DML check error! select 语句中缺少 where/limit/offset 子句!
```

如果 `lightdb_best_practice_mode` 配置为 'all'，则表示同时检查所有类型。当然，您可以随意组合要配置的检查，例如：

```
lightdb_best_practice_mode = 'no_paging'
```

## H. 38. 2. SQL 拦截

### H. 38. 2. 1. 概述

此功能将拦截与 `lt_sql_inspect.secure_mode` 中指定规则匹配的风险 SQL。

仅操作指定数据库的 SQL 将被阻止，使用 `lt_sql_inspect.secure_enable_db` 指定数据库。

您可以通过 `lt_sql_inspect.secure_enable_time` 定义时间段，在此时间段内才会拦截 SQL。时间段仅限于一天内。

`lightdb_best_practice_level` 控制拦截 SQL 时的日志级别，默认为 `'warn'`。  
`lightdb_best_practice_level` 的值为 `'warn'` 表示错误级别为警告。`'error'` 表示错误。

详细规则请参阅 `lt_sql_inspect.secure_mode`。

## H. 38. 2. 2. 配置参数

`lt_sql_inspect.secure_mode` (String)

`lt_sql_inspect.secure_mode` 指定了一些 SQL 执行规则，例如拦截符合条件的 SQL。您可以指定多个规则，用逗号分隔。支持以下规则：  
`select_without_where`、`update_without_where`、`delete_without_where`、`no_paging`、`high_risk_ddl`。

可以在单个会话中更改此设置。

- `select_without_where`: 阻止仅查询一个表且没有 `where` 条件、没有 `limit` 和没有 `offset` 的 `select` 查询。
- `update_without_where`: 阻止仅查询一个表且没有 `where` 条件、没有 `limit` 和没有 `offset` 的 `update` 查询。
- `delete_without_where`: 阻止仅查询一个表且没有 `where` 条件、没有 `limit` 和没有 `offset` 的 `delete` 查询。
- `no_paging`: 相当于同时使用 `select_without_where`、`update_without_where` 和 `delete_without_where`。
- `high_risk_ddl`: 阻止以下 `ddl`: `drop table`、`truncate table`、`alter table add/drop column`、`alter table add/drop Constraint`、`alter table modify column(null/not null, default/no default, type)`。

`lt_sql_inspect.secure_enable_time` (String)

`lt_sql_inspect.secure_enable_time` 指定时间段，在此时间段内才会拦截 SQL。

格式为 `'start_time - end_time'`（例如：`'09:00 - 18:00'`）。

此参数只能在 `lightdb.conf` 文件或服务器命令行中设置，并可重新加载。

`lt_sql_inspect.secure_db` (String)

`lt_sql_inspect.secure_enable_db` 指定数据库，仅在这些数据库上才会拦截 SQL。

您可以指定多个数据库，用逗号分隔。

此参数只能在 `lightdb.conf` 文件或服务器命令行中设置，并可重新加载。

## H. 38. 2. 3. 规则

### H. 38. 2. 3. 1. `select_without_where`

如果一个DQL仅涉及一个表，且没有`where`、`limit`和`offset`语句，则会被阻塞。

查询`pg_catalog`、`lt_catalog`或`ltcluster`方案下的表时，SQL将不会被阻塞。需要注意的是，当`lightdb_best_practice_level`设置为`error`时，查询其他模式下符合要求的表（包括扩展创建的表）将会报错。

```

lightdb@lt_test=# set lt_sql_inspect.secure_mode=select_without_where;
SET
lightdb@lt_test=# select * from t1;
ERROR:  lt_sql_inspect: SQL cannot execute, because a full table scan or
modification may be performed on the table "t1"
lightdb@lt_test=# select * from (select * from t1);
ERROR:  lt_sql_inspect: SQL cannot execute, because a full table scan or
modification may be performed on the table "t1"
lightdb@lt_test=# select * from t1 limit 1;
key1 | key2
-----+-----
(0 rows)

lightdb@lt_test=# select * from t1 offset 1;
key1 | key2
-----+-----
(0 rows)

```

涉及多个表的DQL不会被阻塞。

```

lightdb@lt_test=# select * from t1, t2;
key1 | key2 | key1
-----+-----+-----
(0 rows)

```

这同样适用于视图，就像适用于子查询一样。

```

lightdb@lt_test=# create view t1_v as select * from t1;
CREATE VIEW
lightdb@lt_test=# select * from t1_v;
ERROR:  lt_sql_inspect: SQL cannot execute, because a full table scan or
modification may be performed on the table "t1"
lightdb@lt_test=# select * from t1_v where key1=1;
key1 | key2
-----+-----
(0 rows)

```

### H. 38. 2. 3. 2. update\_without\_where

与select\_without\_where类似，但针对update语句。

```

lightdb@lt_test=# set lt_sql_inspect.secure_mode = 'update_without_where';
SET
lightdb@lt_test=# update t1 set key1=1;
ERROR:  lt_sql_inspect: SQL cannot execute, because a full table scan or
modification may be performed on the table "t1"
lightdb@lt_test=# update t1 set key1=1 limit 1;
UPDATE 0
lightdb@lt_test=# update t1 set key1=1 offset 1;

```

```
UPDATE 0
lightdb@lt_test=# update t1 set key1=1 from t2 ;
UPDATE 0
lightdb@lt_test=# update t1 set key1=1 from t2 where t1.key1=t2.key1;
UPDATE 0
```

#### H. 38.2.3.3. delete\_without\_where

与select\_without\_where类似，但针对delete语句。

```
lightdb@lt_test=# set lt_sql_inspect.secure_mode = 'delete_without_where';
SET
lightdb@lt_test=# delete from t1 ;
ERROR:  lt_sql_inspect: SQL cannot execute, because a full table scan or
modification may be performed on the table "t1"
lightdb@lt_test=# delete from t1 where key1=1;
DELETE 0
lightdb@lt_test=# delete from t1 limit 1;
DELETE 0
lightdb@lt_test=# delete from t1 offset 1;
DELETE 0
lightdb@lt_test=# delete from t1 using t2;
DELETE 0
lightdb@lt_test=# delete from t1 using t2 where t1.key1=t2.key1;
DELETE 0
```

#### H. 38.2.3.4. no\_paging

等价于同时使用select\_without\_where、update\_without\_where和delete\_without\_where。

```
lightdb@lt_test=# set lt_sql_inspect.secure_mode=no_paging;
SET
lightdb@lt_test=#
ERROR:  lt_sql_inspect: SQL cannot execute, because a full table scan or
modification may be performed on the table "t1"
lightdb@lt_test=# update t1 set key1=1;
ERROR:  lt_sql_inspect: SQL cannot execute, because a full table scan or
modification may be performed on the table "t1"
lightdb@lt_test=# delete from t1 ;
ERROR:  lt_sql_inspect: SQL cannot execute, because a full table scan or
modification may be performed on the table "t1"
```

#### H. 38.2.3.5. high\_risk\_ddl

目前以下DDL将被视为高风险DDL，并将被拦截。

- DROP TABLE

```
lightdb@lt_test=# set lt_sql_inspect.secure_mode = high_risk_ddl;
设置
lightdb@lt_test=# drop table t1;
错误: lt_sql_inspect: SQL无法执行，因为它是高风险DDL
```

- TRUNCATE TABLE

```
lightdb@lt_test=# set lt_sql_inspect.secure_mode = high_risk_ddl;
设置
lightdb@lt_test=# truncate table t1;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
```

- ALTER TABLE ADD COLUMN

```
lightdb@lt_test=# set lt_sql_inspect.secure_mode = high_risk_ddl;
设置
lightdb@lt_test=# alter table t1 add column key3 int;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
```

- ALTER TABLE DROP COLUMN

```
lightdb@lt_test=# set lt_sql_inspect.secure_mode = high_risk_ddl;
设置
lightdb@lt_test=# alter table t1 drop column key1;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
```

- ALTER TABLE ADD CONSTRAINT

```
lightdb@lt_test=# set lt_sql_inspect.secure_mode = high_risk_ddl;
设置
lightdb@lt_test=# alter table t1 add constraint t_uq unique(key1);
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
```

- ALTER TABLE DROP CONSTRAINT

```
lightdb@lt_test=# set lt_sql_inspect.secure_mode = high_risk_ddl;
设置
lightdb@lt_test=# alter table t1 drop constraint t_uq;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
```

- ALTER TABLE MODIFY/ALTER COLUMN (仅以下示例中的类型将被阻止, 包括null/not-null, default/non-default, 以及type)。

```
lightdb@lt_test=# set lt_sql_inspect.secure_mode = high_risk_ddl;
设置
lightdb@lt_test=# alter table t1 modify key2 bigint;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
lightdb@lt_test=#
lightdb@lt_test=# alter table t1 alter key2 type text;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
lightdb@lt_test=#
lightdb@lt_test=# alter table t1 modify key2 null;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
lightdb@lt_test=#
lightdb@lt_test=# alter table t1 alter key2 drop not null;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
lightdb@lt_test=#
lightdb@lt_test=# alter table t1 modify key2 not null;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
```

```

lightdb@lt_test=#
lightdb@lt_test=# alter table t1 alter key2 set not null;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
lightdb@lt_test=#
lightdb@lt_test=# alter table t1 modify key2 default 10;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
lightdb@lt_test=#
lightdb@lt_test=# alter table t1 alter key2 set default 10;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
lightdb@lt_test=#
lightdb@lt_test=# alter table t1 alter key2 drop default;
错误: lt_sql_inspect: SQL无法执行, 因为它是高风险DDL
lightdb@lt_test=#

```

## H. 39. lt\_standby\_forward

lt\_standby\_forward是一个模块, 它将SQL语句从热备节点转发到主节点, 提供一个减轻主节点负载和允许客户端从备用节点查询的功能。例如, 客户端可以将INSERT、CREATE语句发送到热备节点。

### H. 39. 1. 配置参数

lt\_standby\_forward.enable\_forward

它控制用户是否启用此功能。默认为off。在备用节点上设置此选项, 或者为单个会话设置此选项。

### H. 39. 2. 使用方法

您必须在lightdb.conf中添加如下所示的行。

```
shared_preload_libraries = 'lt_standby_forward'
```

### H. 39. 3. 认证

lt\_standby\_forward使用primary\_conninfo连接到主节点, 将当前的dbname和username (即用于连接到备用节点的数据库和用户) 附加到primary\_conninfo中。连接信息类似于:

```
primary_conninfo dbname=a_db user=a_user application_name=lt_standby_forward
options='-c opt1=value1 -c opt2=value2'
```

不支持设置密码。在主节点上配置lt\_hba.conf或在备用节点上将密码放入passfile中以避免输入密码。

### H. 39. 4. 功能/限制

- 转发DELETE、UPDATE、INSERT、SELECT with modifying CTE语句到主节点。
- 转发包含volatile函数的SELECT/CALL语句到主节点, 否则在备用节点上执行。
- 在备用节点上执行只读事务, 否则转发到主节点。
- 事务中不允许DDL语句。

- 在转发到主节点之前，视图（View）将在备用节点上重写（expanded）。如果视图包含当前用户无法访问的对象，则查询失败。
- 在主备库上都执行 SET 命令，但在备库上执行 show 命令。不要在连接建立后执行 SET xxx\_timeout 命令，请使用 ltsql/jdbc 参数 options 设置这些参数。对于 ltsql/jdbc 的 options，只有 statement\_timeout、lock\_timeout 和 idle\_in\_transaction\_session\_timeout 会传递到主库。
- 不支持 COPY FROM 命令。
- 不支持 CREATE TEMP/UNLOGGED TABLE 命令。
- 未在上文提到的实用命令将会转发到主库。

## H. 40. lt\_stat\_activity

这个 LightDB 扩展添加了 lt\_stat\_activity 视图。

### H. 40. 1. 概念

该扩展提供了 lt\_stat\_activity 视图，其中包含所有 pg\_stat\_activity 字段，并添加了 query\_id 字段，该字段与 pg\_stat\_statements 中的 query\_id 一致。在等待事件发生时，可以通过 query\_id 使用 lt\_stat\_activity 和 pg\_stat\_statements 进行分析。

### H. 40. 2. 先决条件

lt\_stat\_activity 扩展依赖于 lt\_stat\_statements 扩展。

### H. 40. 3. 创建扩展

你必须将 lt\_stat\_activity 添加到 shared\_preload\_libraries 中，修改 shared\_preload\_libraries 后需要重新启动服务器。

```
# lightdb.conf
shared_preload_libraries = 'lt_stat_statements,lt_stat_activity'

lightdb@postgres=# CREATE EXTENSION lt_stat_statements;
lightdb@postgres=# CREATE EXTENSION lt_stat_activity;
```

你可以在 lightdb.conf 中定义扩展参数。

lt\_stat\_activity.track = top - 默认值，跟踪顶层查询。

lt\_stat\_activity.track = none - lt\_stat\_activity 中的 query\_id 字段始终为空。

## H. 41. lt\_stat\_statements

lt\_stat\_statements 模块提供了一种跟踪服务器执行的所有 SQL 语句的计划和执行统计信息的方法。

该模块必须通过在 lightdb.conf 中将 lt\_stat\_statements 添加到 shared\_preload\_libraries 来加载，因为它需要额外的共享内存。这意味着需要重新启动服务器才能添加或删除该模块。

当加载 lt\_stat\_statements 后，它会跨服务器上的所有数据库跟踪统计信息。要访问和操作这些统计信息，该模块提供了一个视图 pg\_stat\_statements 和实用函数



`pg_stat_statements_reset` 和 `pg_stat_statements` 。这些不是全局可用的，但可以通过 `CREATE EXTENSION lt_stat_statements`，为特定数据库启用。

## H.41.1. `pg_stat_statements` 视图

该模块收集的统计信息通过名为 `pg_stat_statements` 的视图可用。该视图对于每个不同的数据库 ID、用户 ID 和查询 ID（最多追踪模块可跟踪的不同语句数量）包含一行。该视图的列显示在 Table H.54中。

Table H.54. `pg_stat_statements` 列

列类型	描述
<code>userid oid</code> (参考 <code>pg_authid.oid</code> )	执行该语句的用户的 OID
<code>dbid oid</code> (参考 <code>pg_database.oid</code> )	执行该语句的数据库的 OID
<code>queryid bigint</code>	内部哈希码，从语句的解析树计算得出
<code>query text</code>	代表性语句的文本
<code>plans bigint</code>	该语句被计划的次数（如果启用了 <code>lt_stat_statements.track_planning</code> ，否则为零）
<code>total_plan_time double precision</code>	计划该语句的总时间，以毫秒为单位（如果启用了 <code>lt_stat_statements.track_planning</code> ，否则为零）
<code>min_plan_time double precision</code>	计划该语句最小的时间，以毫秒为单位（如果启用了 <code>lt_stat_statements.track_planning</code> ，否则为零）
<code>max_plan_time double precision</code>	计划该语句最大的时间，以毫秒为单位（如果启用了 <code>lt_stat_statements.track_planning</code> ，否则为零）
<code>mean_plan_time double precision</code>	计划该语句平均时间，以毫秒为单位（如果启用了 <code>lt_stat_statements.track_planning</code> ，否则为零）
<code>stddev_plan_time double precision</code>	计划该语句时间的总体标准偏差，以毫秒为单位（如果启用了 <code>lt_stat_statements.track_planning</code> ，否则为零）
<code>calls bigint</code>	该语句被执行的次数
<code>total_exec_time double precision</code>	执行该语句的总时间，以毫秒为单位
<code>min_exec_time double precision</code>	执行该语句最小的时间，以毫秒为单位
<code>max_exec_time double precision</code>	执行该语句最大的时间，以毫秒为单位
<code>mean_exec_time double precision</code>	执行该语句平均时间，以毫秒为单位
<code>stddev_exec_time double precision</code>	执行该语句时间的总体标准偏差，以毫秒为单位

列类型	描述
rows bigint	该语句检索或影响的总行数
shared_blks_hit bigint	该语句在共享块缓存中的总命中次数
shared_blks_read bigint	该语句读取的共享块的总数
shared_blks_dirtied bigint	该语句脏写的共享块的总数
shared_blks_written bigint	该语句写入的共享块的总数
local_blks_hit bigint	该语句在本地块缓存中的总命中次数
local_blks_read bigint	该语句读取的本地块总数
local_blks_dirtied bigint	该语句脏写的本地块总数
local_blks_written bigint	该语句写入的本地块总数
temp_blks_read bigint	该语句读取的临时块总数
temp_blks_written bigint	该语句写入的临时块总数
blk_read_time double precision	该语句读取块的总时间，以毫秒为单位（如果启用了 track_io_timing，否则为零）
blk_write_time double precision	该语句写入块的总时间，以毫秒为单位（如果启用了 track_io_timing，否则为零）
wal_records bigint	该语句生成的WAL记录总数
wal_fpi bigint	该语句生成的WAL完整页映像总数
wal_bytes numeric	该语句生成的WAL总字节数

基于安全考虑，只有超级用户和pg\_read\_all\_stats角色的成员才能查看其他用户执行的SQL文本和 queryid。但是，如果在用户数据库中安装了该视图，则其他用户可以查看统计信息。

可计划的查询（即SELECT、INSERT、UPDATE和DELETE）根据内部哈希计算，当它们具有相同的查询结构时，会被合并成一个pg\_stat\_statements条目。通常，如果两个查询在语义上是等价的，除了查询中出现的文字常量的值之外，它们将被视为相同。然而，对于实用程序命令（即所有其他命令），它们严格按照它们的文本查询字符串进行比较。

当一个常量的值已被忽略，以便将查询与其他查询匹配时，该常量将在pg\_stat\_statements中被参数符号（如\$1）替换。查询文本的其余部分是具有特定queryid哈希值的第一个查询的文本，与pg\_stat\_statements条目相关联。

在某些情况下，具有不同文本的查询可能会合并为单个pg\_stat\_statements条目。通常，只有语义等价的查询才会发生这种情况，但哈希冲突有可能导致不相关的查询合并为一个条目。（但是，对于属于不同用户或数据库的查询，这种情况不会发生。）

由于queryid哈希值是在查询的后分析表示上计算的，因此相反的情况也可能发生：如果由于不同的search\_path 设置等因素而具有不同含义的相同文本的查询可能会显示为单独的条目。

pg\_stat\_statements的使用者可能希望使用queryid（或许结合dbid 和userid）作为每个条目的更稳定和可靠的标识符，而不是使用其查询文本。然而，重要的是要理解在queryid 哈希值的稳定性方面只有有限的保证。由于该标识符是从后分析树导出的，因此其值是该表示中出现的内部对象标识符等因素的函数。这有一些反直觉的含义。例如，如果两个明显相同的查询引用了在两个查询执行之间被删除并重新创建的表，那么lt\_stat\_statements将认为这两个查询是不同的。哈希处理还对机器架构和平台的其他方面的差异敏感。此外，不能保证queryid在LightDB的主要版本之间稳定。

一般而言，只有在底层服务器版本和目录元数据细节完全相同的情况下，才能假定queryid值是稳定和可比的。基于物理WAL重放的复制中参与的两个服务器可以预期具有相同的 queryid值，用于相同的查询。但是，逻辑复制方案不能保证在所有相关细节上使副本完全相同，因此queryid 将不是跨一组逻辑副本累积成本的有用标识符。如果有疑问，建议进行直接测试。

用于替换代表性查询文本中的常量的参数符号从原始查询文本中最高的\$ n参数之后的下一个编号开始，如果没有，则从\$1开始。值得注意的是，在某些情况下，可能会有影响此编号的隐藏参数符号。例如， PL/pgSQL使用隐藏的参数符号将函数局部变量的值插入查询中，因此像SELECT i + 1 INTO j这样的 PL/pgSQL语句会具有像SELECT i + \$2这样的代表性文本。

代表性查询文本被保存在外部磁盘文件中，不占用共享内存。因此，即使是非常长的查询文本也可以成功存储。但是，如果积累了许多长查询文本，则外部文件可能会变得过于庞大而难以管理。如果发生这种情况， lt\_stat\_statements可能会选择丢弃查询文本，此时pg\_stat\_statements视图中的所有现有条目都将显示空的 query字段，尽管与每个queryid相关联的统计信息将被保留。如果发生这种情况，请考虑减少lt\_stat\_statements.max以防止再次发生。

plans和calls 不总是期望匹配，因为规划和执行统计信息是在各自的结束阶段更新的，并且仅更新成功的操作。例如，如果语句成功规划但在执行阶段失败，则仅更新其规划统计信息。如果由于使用缓存的计划而跳过规划，则仅更新其执行统计信息。

## H. 41. 2. 函数

pg\_stat\_statements\_reset(userid Oid, dbid Oid, queryid bigint) returns void

pg\_stat\_statements\_reset将与指定的userid、dbid 和queryid相对应的lt\_stat\_statements 收集到的统计数据丢弃。如果没有指定参数，则对于每个参数使用默认值0（无效），并将与其他参数匹配的统计数据重置。如果未指定任何参数或所有指定的参数均为0（无效），则将丢弃所有统计信息。默认情况下，只有超级用户才能执行此函数。可以使用GRANT授予其他用户访问权限。

pg\_stat\_statements(showtext boolean) returns setof record

pg\_stat\_statements视图是以同名函数pg\_stat\_statements定义的。客户端可以直接调用 pg\_stat\_statements函数，并通过指定showtext := false来省略查询文本（也就是说，对应于视图的 query列的OUT 参数将返回null）。此功能旨在支持可能希望避免重复检索长度不确定的查询文本的外部工具。这样的工具可以自行缓存每个条目观察到的第一个查询文本，因为这就是lt\_stat\_statements 所做的全部工作，然后根据需要检索查询文本。由于服务器将查询文本存储在文件中，因此这种方法可能会减少重复检索pg\_stat\_statements 数据的物理I/O。

## H. 41. 3. 配置参数

lt\_stat\_statements.max (integer)

lt\_stat\_statements.max是模块跟踪的语句的最大数量（即pg\_stat\_statements 视图中最大的行数）。如果观察到的不同语句超过该数量，则舍弃执行最少的语句的信息。默认值为5000。此参数只能在服务器启动时设置。

lt\_stat\_statements.track (enum)

lt\_stat\_statements.track控制模块计数的语句类型。指定top以跟踪顶层语句（直接由客户端发出的语句），指定all以跟踪嵌套语句（例如在函数内部调用的语句），或指定none以禁用语句统计信息收集。默认值为top。只有超级用户可以更改此设置。

lt\_stat\_statements.track\_utility (boolean)

lt\_stat\_statements.track\_utility控制模块是否跟踪实用程序命令。实用程序命令是除SELECT、INSERT、UPDATE和DELETE之外的所有命令。默认值为on。只有超级用户可以更改此设置。

lt\_stat\_statements.track\_planning (boolean)

lt\_stat\_statements.track\_planning参数控制模块是否跟踪规划操作及其持续时间。启用此参数可能会带来明显的性能损失，特别是当具有相同查询结构的语句被许多并发连接执行时，这些连接会竞争更新一个较少的pg\_stat\_statements条目。默认值为off。只有超级用户可以更改此设置。

lt\_stat\_statements.save (布尔型)

lt\_stat\_statements.save指定是否在服务器关闭时保存语句统计信息。如果设置为off，则在关闭服务器时不保存统计信息，也不会服务器启动时重新加载。默认值为on。该参数只能在lightdb.conf文件或服务器命令行中设置。

该模块需要额外的共享内存，数量与lt\_stat\_statements.max成比例。请注意，该内存存在模块加载时被消耗，即使lt\_stat\_statements.track设置为none。

这些参数必须在lightdb.conf中设置。典型的用法可能是：

```
# lightdb.conf
shared_preload_libraries = 'lt_stat_statements'

lt_stat_statements.max = 10000
lt_stat_statements.track = all
```

## H. 41. 4. 样例输出

```
bench=# SELECT pg_stat_statements_reset();

$ ltbench -i bench
$ ltbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
          nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
-[ RECORD 1 ]-----+-----
query          | UPDATE ltbench_branches SET bbalance = bbalance + $1 WHERE bid
= $2
calls          | 3000
total_exec_time | 25565.855387
rows          | 3000
hit_percent    | 100.0000000000000000
-[ RECORD 2 ]-----+-----
query          | UPDATE ltbench_tellers SET tbalance = tbalance + $1 WHERE tid
= $2
```

```

calls          | 3000
total_exec_time | 20756.669379
rows           | 3000
hit_percent    | 100.0000000000000000
-[ RECORD 3 ]---+-----
-----
query          | copy ltbench_accounts from stdin
calls          | 1
total_exec_time | 291.865911
rows           | 100000
hit_percent    | 100.0000000000000000
-[ RECORD 4 ]---+-----
-----
query          | UPDATE ltbench_accounts SET abalance = abalance + $1 WHERE aid
= $2
calls          | 3000
total_exec_time | 271.232977
rows           | 3000
hit_percent    | 98.8454011741682975
-[ RECORD 5 ]---+-----
-----
query          | alter table ltbench_accounts add primary key (aid)
calls          | 1
total_exec_time | 160.588563
rows           | 0
hit_percent    | 100.0000000000000000

bench=# SELECT pg_stat_statements_reset(0,0,s.queryid) FROM pg_stat_statements
AS s
        WHERE s.query = 'UPDATE ltbench_branches SET bbalance = bbalance +
$1 WHERE bid = $2';

bench=# SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
        nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
        FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
-[ RECORD 1 ]---+-----
-----
query          | UPDATE ltbench_tellers SET tbalance = tbalance + $1 WHERE tid
= $2
calls          | 3000
total_exec_time | 20756.669379
rows           | 3000
hit_percent    | 100.0000000000000000
-[ RECORD 2 ]---+-----
-----
query          | copy ltbench_accounts from stdin
calls          | 1
total_exec_time | 291.865911
rows           | 100000
hit_percent    | 100.0000000000000000
-[ RECORD 3 ]---+-----
-----
query          | UPDATE ltbench_accounts SET abalance = abalance + $1 WHERE aid
= $2
calls          | 3000
total_exec_time | 271.232977
rows           | 3000

```

```

hit_percent | 98.8454011741682975
-[ RECORD 4 ]-----+-----
-----
query       | alter table ltbench_accounts add primary key (aid)
calls       | 1
total_exec_time | 160.588563
rows        | 0
hit_percent | 100.0000000000000000
-[ RECORD 5 ]-----+-----
-----
query       | vacuum analyze ltbench_accounts
calls       | 1
total_exec_time | 136.448116
rows        | 0
hit_percent | 99.9201915403032721

bench=# SELECT pg_stat_statements_reset(0,0,0);

bench=# SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
        nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
        FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
-[ RECORD 1 ]-----+-----
-----
query       | SELECT pg_stat_statements_reset(0,0,0)
calls       | 1
total_exec_time | 0.189497
rows        | 1
hit_percent |
-[ RECORD 2 ]-----+-----
-----
query       | SELECT query, calls, total_exec_time, rows, $1 *
        shared_blks_hit /
        +
        nullif(shared_blks_hit + shared_blks_read, $2)
        AS hit_percent+
        FROM pg_stat_statements ORDER BY total_exec_time
        DESC LIMIT $3
calls       | 0
total_exec_time | 0
rows        | 0
hit_percent |

```

## H. 41.5. 作者

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>。 查询规范化由Peter Geoghegan <peter@2ndquadrant.com>添加。

## H. 42. lt\_tempfile

lt\_tempfile 提供了许多支持函数，用于管理在工作内存不足时生成的临时文件。更多细节，请参见 work\_mem 和 lightdb\_keep\_temp\_file 。

Table H.55. lt\_tempfile 函数

函数	描述
lt_catalog.lt_remove_temp_file	( mode text ) → setof record

函数	描述
	删除临时文件。's' 用于当前会话，'*' 用于所有会话。例如：select lt_remove_temp_file('*')。
lt_catalog.lt_list_temp_file ( mode text ) → boolean	列出临时文件。's' 用于当前会话，'*' 用于所有会话。例如：select lt_remove_temp_file('*')。

lt\_remove\_temp\_file 会删除 LTDATA /base/ltsql\_tmp/keep 目录中的临时文件。如果指定的文件不存在或 lt\_remove\_temp\_file() 调用失败，则返回 false；其他情况将抛出错误。

lt\_list\_temp\_file 会返回 LTDATA /base/ltsql\_tmp 目录中所有文件的开始时间戳、路径名和临时文件数。

## H. 43. lt\_trgm

lt\_trgm 模块提供了基于三元组匹配的字母数字文本相似度计算函数和运算符，以及支持快速搜索相似字符串的索引操作类。

该模块被视为“受信任的”，也就是说，可以由在当前数据库上具有 CREATE 权限的非超级用户安装。

### H. 43. 1. 三元组（或三图）概念

三元组是从字符串中取出的三个连续字符组成的组合。我们可以通过计算两个字符串共享的三元组数量来衡量它们之间的相似度。这个简单的想法在许多自然语言中用于衡量单词之间的相似度，非常有效。

**Note**

lt\_trgm 在从字符串中提取三元组时会忽略非单词字符（非字母数字字符）。每个单词在确定包含在字符串中的三元组集时，都被认为有两个前导空格和一个后缀空格。例如，在字符串“ cat ”中的三元组集合是“ c ”、“ ca ”、“ cat ”和“ at ”。在字符串“ foo|bar ”中的三元组集合是“ f ”、“ fo ”、“ foo ”、“ oo ”、“ b ”、“ ba ”、“ bar ”和“ ar ”。

### H. 43. 2. 函数和运算符

lt\_trgm 模块提供的函数在 Table H. 56 中列出，操作符在 Table H. 57 中列出。

Table H. 56. lt\_trgm 函数

函数	描述
similarity ( text, text ) → real	返回一个数字，表示两个参数有多相似。结果的范围是从零（表示两个字符串完全不相似）到一（表示两个字符串完全相同）。
show_trgm ( text ) → text []	返回给定字符串中所有三元组的数组。（实际上，这很少有用，只用于调试。）
word_similarity ( text, text ) → real	

函数	描述
	返回一个数字，表示第一个字符串中的三元组集合和第二个字符串中任意连续的三元组有序集合之间的最大相似度。有关详细信息，请参见下面的说明。
<code>strict_word_similarity ( text, text ) → real</code>	与 <code>word_similarity</code> 相同，但强制范围边界与单词边界匹配。由于我们没有跨单词的三元组，因此该函数实际上返回第一个字符串和第二个字符串中任意单词连续集合之间的最大相似度。
<code>show_limit () → real</code>	返回当前由 <code>%</code> 运算符使用的相似度阈值。这设置了两个单词被认为足够相似以被视为错拼字的最小相似度，例如。（已弃用；改用 <code>SHOW lt_trgm.similarity_threshold.</code> ）
<code>set_limit ( real ) → real</code>	设置由 <code>%</code> 运算符使用的当前相似度阈值。阈值必须在0和1之间（默认值为0.3）。返回传入的相同值。（已弃用；改用 <code>SET lt_trgm.similarity_threshold.</code> ）

考虑以下示例：

```
# SELECT word_similarity('word', 'two words');
word_similarity
-----
                0.8
(1 row)
```

在第一个字符串中，三元组集合为 { " w", " wo", " wor", " ord", " rd " }。在第二个字符串中，有序的三元组集合为 { " t", " tw", " two", " wo ", " w", " wo", " wor", " ord", " rds", " ds " }。第二个字符串中有序三元组集合的最相似的范围是 { " w", " wo", " wor", " ord" }，相似度为 0.8。

此函数返回一个值，可以近似理解为第一个字符串与第二个字符串的任何子字符串之间的最大相似度。但是，此函数不会在范围的边界添加填充。因此，第二个字符串中存在的额外字符数不会被考虑，除了不匹配的单词边界之外。

同时，`strict_word_similarity`在第二个字符串中选择一个单词的范围。在上面的示例中，`strict_word_similarity`将选择单词'words'的范围，其三元组集合为 { " w", " wo", " wor", " ord", " rds", " ds " }。

```
# SELECT strict_word_similarity('word', 'two words'), similarity('word',
'words');
strict_word_similarity | similarity
-----+-----
                0.571429 | 0.571429
(1 row)
```

因此，`strict_word_similarity`函数用于查找与整个单词的相似度，而`word_similarity`更适用于查找单词部分的相似度。

Table H.57. `lt_trgm` 运算符

运算符	描述
<code>text % text → boolean</code>	如果其参数的相似度大于由 <code>lt_trgm.similarity_threshold</code> 设置的当前相似度阈值，则返回true。
<code>text &lt;% text → boolean</code>	



运算符	描述
	如果第一个参数中的三元组集合与第二个参数中有序三元组集合的连续范围之间的相似度 大于由 <code>lt_trgm.word_similarity_threshold</code> 参数设置的当前单词相似度阈值， 则返回 <code>true</code> 。
<code>text %&gt; text</code>	<code>boolean</code> <%运算符的可交换运算符。
<code>text &lt;&lt;% text</code>	<code>boolean</code> 如果其第二个参数具有与单词边界匹配的有序三元组集合的连续范围， 并且与第一个参数中的三元组集合的相似度大于由 <code>lt_trgm.strict_word_similarity_threshold</code> 参数设置的当前严格单词相似度阈值， 则返回 <code>true</code> 。
<code>text %&gt;&gt; text</code>	<code>boolean</code> <<%运算符的可交换运算符。
<code>text &lt;-&gt; text</code>	<code>real</code> 返回参数之间的“距离”，即1减去 <code>similarity()</code> 值。
<code>text &lt;&lt;-&gt; text</code>	<code>real</code> 返回参数之间的“距离”，即1减去 <code>word_similarity()</code> 值。
<code>text &lt;-&gt;&gt; text</code>	<code>real</code> <<->运算符的可交换运算符。
<code>text &lt;&lt;&lt;-&gt; text</code>	<code>real</code> 返回参数之间的“距离”，即1减去 <code>strict_word_similarity()</code> 值。
<code>text &lt;-&gt;&gt;&gt; text</code>	<code>real</code> <<<->运算符的可交换运算符。

### H. 43. 3. GUC参数

`lt_trgm.similarity_threshold` (real)

设置当前由%运算符使用的相似度阈值。阈值必须在0到1之间（默认值为0.3）。

`lt_trgm.word_similarity_threshold` (real)

设置当前由<%和%>运算符使用的单词相似度阈值。阈值必须在0到1之间（默认值为0.6）。

`lt_trgm.strict_word_similarity_threshold` (real)

设置当前由<<%和%>>运算符使用的严格单词相似度阈值。阈值必须在0到1之间（默认值为0.5）。

### H. 43. 4. 索引支持

`lt_trgm` 模块提供了GiST和GIN索引操作符类，允许您在文本列上创建索引，以便进行非常快速的相似性搜索。这些索引类型支持上述描述的相似性运算符，另外还支持基于trigram的索引搜索，用于LIKE，ILIKE，~和~\*查询。（这些索引不支持相等性和简单比较运算符，因此您可能还需要一个常规的B-tree索引。）

示例：

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops);
```

or

```
CREATE INDEX trgm_idx ON test_trgm USING GIN (t gin_trgm_ops);
```

`gin_trgm_ops` GiST `opclass`将一组trigram近似为位图签名。它的可选整数参数`siglen` 决定了签名长度，以字节为单位。默认长度为12个字节。签名长度的有效值在1到2024字节之间。更长的签名会导致更精确的搜索（扫描较小比例的索引和较少的堆页面），但代价是更大的索引。

创建一个签名长度为32字节的索引示例如下：

```
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops(siglen=32));
```

此时，您将在`t`列上拥有一个用于相似性搜索的索引。一个典型的查询是：

```
SELECT t, similarity(t, 'word') AS sml
FROM test_trgm
WHERE t % 'word'
ORDER BY sml DESC, t;
```

这将返回与`word`足够相似的文本列中的所有值，按从最佳匹配到最差匹配的顺序排序。即使在非常大的数据集上，索引也将用于使此操作快速执行。

上述查询的一个变体是

```
SELECT t, t <-> 'word' AS dist
FROM test_trgm
ORDER BY dist LIMIT 10;
```

这可以通过GiST索引非常有效地实现，但无法通过GIN索引实现。当只需要少数最接近的匹配项时，它通常会超过第一种表述。

同样，您可以使用在`t`列上的索引进行单词相似性或严格单词相似性搜索。典型的查询如下：

```
SELECT t, word_similarity('word', t) AS sml
FROM test_trgm
WHERE 'word' <% t
ORDER BY sml DESC, t;
```

和

```
SELECT t, strict_word_similarity('word', t) AS sml
FROM test_trgm
WHERE 'word' <<% t
ORDER BY sml DESC, t;
```

这将返回与对应的有序trigram集中具有足够相似性的`word` 的trigram集的所有文本列值，按从最佳匹配到最差匹配的顺序排序。即使在非常大的数据集上，索引也将用于使此操作快速执行。

上述查询的可能变体包括：

```
SELECT t, 'word' <<-> t AS dist
FROM test_trgm
ORDER BY dist LIMIT 10;
```

和

```
SELECT t, 'word' <<<-> t AS dist
FROM test_trgm
ORDER BY dist LIMIT 10;
```

这可以通过GiST索引非常有效地实现，但无法通过GIN索引实现。

从LightDB 9.1开始，这些索引类型还支持对LIKE和ILIKE的索引搜索，例如：

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

索引搜索通过从搜索字符串中提取trigram，然后在索引中查找这些trigram来实现。搜索字符串中的trigram越多，索引搜索就越有效。与基于B-tree的搜索不同，搜索字符串不需要左对齐。

从LightDB 9.3开始，这些索引类型还支持正则表达式匹配的索引搜索（~和~\*运算符），例如：

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

索引搜索通过从正则表达式中提取trigram，然后在索引中查找这些trigram来实现。可以从正则表达式中提取的trigram越多，索引搜索就越有效。与基于B-tree的搜索不同，搜索字符串不需要左对齐。

对于LIKE和正则表达式搜索，需要注意的是，没有可提取的trigram的模式将退化为全索引扫描。

选择GiST和GIN索引取决于GiST和GIN的相对性能特征，这些特征在其他地方进行了讨论。

## H. 43. 5. 文本搜索集成

当与全文索引结合使用时，trigram匹配是一种非常有用的工具。特别是它可以帮助识别拼写错误的输入单词，这些单词不会直接被全文搜索机制匹配到。

第一步是生成一个包含文档中所有唯一单词的辅助表：

```
CREATE TABLE words AS SELECT word FROM
    ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

其中documents是具有文本字段bodytext的表，我们希望对其进行搜索。使用simple配置与to\_tsvector函数，而不是使用特定于语言的配置，是因为我们想要一个原始（未经词干处理的）单词列表。

接下来，在word列上创建一个trigram索引：

```
CREATE INDEX words_idx ON words USING GIN (word gin_trgm_ops);
```

现在，可以使用类似于先前示例的SELECT查询来提供用户搜索词中拼写错误的单词的建议拼写。一个有用的额外测试是要求所选的单词与拼错的单词长度相似。

### Note

由于words表被生成成为一个单独的静态表，因此需要定期重新生成它，以使其保持与文档集合相对应的状态。保持它完全更新通常是不必要的。

## H. 43. 6. 参考文献

GiST开发网站 <http://www.sai.msu.su/~megera/postgres/gist/>

Tsearch2开发网站 <http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/>

## H. 43. 7. 作者

Oleg Bartunov <oleg@sai.msu.su>, 俄罗斯莫斯科大学莫斯科

Teodor Sigaev <teodor@sigaev.ru>, 俄罗斯莫斯科Delta-Soft有限公司

Alexander Korotkov <a.korotkov@postgrespro.ru>, 俄罗斯莫斯科Postgres Professional公司

文档: Christopher Kings-Lynne

本模块由俄罗斯莫斯科Delta-Soft有限公司赞助。

## H. 44. `lt_visibility`

`lt_visibility`模块提供了一种查看表的可见性映射（VM）和页面级可见性信息的方法。它还提供了检查可见性映射完整性并强制重新构建它的函数。

三个不同的位用于存储页面级可见性信息。可见性映射中的全可见位表示关系对应页面中的每个元组对于每个当前和未来的事务都是可见的。可见性映射中的全冻结位表示页面中的每个元组都已被冻结；也就是说，直到插入、更新、删除或者在该页面上锁定元组时，将不需要对该页面进行任何未来的vacuum修改。页面头的PD\_ALL\_VISIBLE位与可见性映射中的全可见位具有相同的含义，但它是存储在数据页面本身而不是单独的数据结构中。这两个位通常是一致的，但在崩溃恢复后，页面的全可见位有时可能被设置而可见性映射位是未设置的。由于在`lt_visibility`检查可见性映射和检查数据页面之间发生的更改，报告的值也可能不一致。任何导致数据损坏的事件也可能导致这些位不一致。

显示与PD\_ALL\_VISIBLE位有关信息的函数比仅查询可见性映射的函数要昂贵得多，因为它们必须读取关系的数据块而不仅仅是（要小得多的）可见性映射。检查关系的数据块的函数同样昂贵。

### H. 44. 1. 函数

```
pg_visibility_map(relation regclass, blkno bigint, all_visible OUT boolean,
all_frozen OUT boolean) returns record
```

返回给定关系的给定块的可见性映射中的全可见位和全冻结位。

```
pg_visibility(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen
OUT boolean, pd_all_visible OUT boolean) returns record
```

返回给定关系的给定块的可见性映射中的全可见位和全冻结位，以及该块的PD\_ALL\_VISIBLE位。

```
pg_visibility_map(relation regclass, blkno OUT bigint, all_visible OUT boolean,
all_frozen OUT boolean) returns setof record
```

返回给定关系的每个块的可见性映射中的全可见位和全冻结位。

```
pg_visibility(relation regclass, blkno OUT bigint, all_visible OUT boolean,
all_frozen OUT boolean, pd_all_visible OUT boolean) returns setof record
```

返回给定关系的每个块的可见性映射中的全可见位和全冻结位，以及每个块的PD\_ALL\_VISIBLE位。

```
pg_visibility_map_summary(relation regclass, all_visible OUT bigint, all_frozen
OUT bigint) returns record
```

返回关系的可见性映射中所有可见页面的数量和所有冻结页面的数量。

```
pg_check_frozen(relation regclass, t_ctid OUT tid) returns setof tid
```

返回储存在标记为全冻结的可见性地图页面中的未冻结元组的TID。如果此函数返回非空的TID集合，则可见性地图已损坏。

```
pg_check_visible(relation regclass, t_ctid OUT tid) returns setof tid
```

返回储存在标记为全可见的可见性地图页面中的未全可见元组的TID。如果此函数返回非空的TID集合，则可见性地图已损坏。

```
pg_truncate_visibility_map(relation regclass) returns void
```

截断给定关系的可见性地图。如果您认为该关系的可见性地图已损坏并希望强制重建它，则此函数很有用。在执行此函数后，给定关系上执行的第一个VACUUM将扫描关系中的每个页面并重建可见性地图。（在执行该操作之前，查询将将可见性地图视为包含所有零。）

默认情况下，这些函数只能由超级用户和pg\_stat\_scan\_tables角色的成员执行，但pg\_truncate\_visibility\_map(relation regclass)除外，它只能由超级用户执行。

## H. 44. 2. 作者

Robert Haas <rhaas@postgresql.org>

## H. 45. ltaudit

### H. 45. 1. 介绍

LightDB审计扩展（ltaudit）通过标准的LightDB日志记录功能提供详细的会话和/或对象审计日志记录。

ltaudit的目标是为LightDB用户提供能力，以产生经常需要遵守政府、金融或ISO认证的审计日志。

审计是对个人或组织账户的官方检查，通常由独立机构进行。ltaudit收集的信息称为审计跟踪或审计日志。本文档中使用术语审计日志。

### H. 45. 2. 为什么使用ltaudit?

标准日志记录设施可以提供基本的语句日志记录，使用 `log_statement = all`。这对于监控和其他用途是可以接受的，但不能提供通常需要进行审核所需的详细级别。仅有一个对数据库执行的所有操作列表是不够的，还必须能够找到审计员感兴趣的特定语句。标准日志记录设施显示用户请求的内容，而ltaudit专注于数据库在满足请求时发生的详细信息。

例如，审计员可能希望验证特定表是否在已记录的维护窗口内创建。这似乎是grep的简单工作，但如果你被呈现出像这个（有意混淆）的例子，怎么办呢：

```
DO $$
BEGIN
    EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';
END $$;
```

标准日志记录将提供以下内容：

```
LOG: 语句: DO $$
BEGIN
    EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';
END $$;
```

在表是动态创建的情况下，似乎需要对代码有一些了解才能找到感兴趣的表。这并不理想，因为最好只搜索表名。这就是ltaudit的作用。对于相同的输入，它将在日志中产生以下输出：

```
AUDIT: SESSION, 33, 1, FUNCTION, DO, ,, "DO $$
BEGIN
    EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';
END $$;"
AUDIT: SESSION, 33, 2, DDL, CREATE TABLE, TABLE, public.important_table, CREATE TABLE
important_table (id INT)
```

不仅记录了DO块，而且子语句2包含了完整的CREATE TABLE文本，带有语句类型、对象类型和完全合格的名称，以便进行简单的搜索。

在记录SELECT和DML语句时，ltaudit可以被配置为为语句中引用的每个关系记录一个单独的条目。不需要解析来查找所有涉及特定表的语句。实际上，目标是提供语句文本主要用于深入的取证，而不是对于审计必须。

### H. 45. 3. 使用注意事项

根据设置，ltaudit可能会生成大量的日志记录。请仔细确定在您的环境中需要审计记录的内容，以避免记录过多。

例如，在OLAP环境中工作时，可能不明智地审计记录插入到大型事实表中。由于日志文件是以文本形式表示的，因此日志文件的大小可能是插入的实际数据大小的几倍。由于日志通常与操作系统一起存储，这可能会导致磁盘空间很快被耗尽。在无法将审计日志限制为某些表的情况下，请确保在测试时评估性能影响并在日志卷上分配足够的空间。这对于OLTP环境也可能是正确的。即使插入数量不高，审计日志记录的性能影响仍可能明显影响延迟。

为了限制记录SELECT和DML语句的关系数量，请考虑使用对象审计日志（请参见 [对象审计](#)）。对象审计日志允许选择要记录的关系，从而减少总日志量。但是，当添加新关系时，必须将它们显式添加到对象审计日志中。在这种情况下，从日志记录中排除指定的表并包含所有其他表的程序化解决方案可能是一个好选择。

### H. 45. 4. 设置

只有超级用户可以修改设置。允许普通用户更改其设置将使审计日志失去意义。

可以在全局范围（在lightdb.conf或使用ALTER SYSTEM ... SET）、在数据库级别（使用ALTER DATABASE ... SET）或在角色级别（使用ALTER ROLE ... SET）指定设置。请注意，设置不通过正常角色继承进行继承，SET ROLE不会更改用户的ltaudit设置。这是角色系统的限制，而不是ltaudit固有的。

ltaudit扩展必须在shared\_preload\_libraries中加载。否则，加载时会引发错误，不会发生审计日志记录。此外，在设置ltaudit.log之前，必须调用CREATE EXTENSION ltaudit。如果删除了ltaudit扩展并需要重新创建，则必须首先取消设置ltaudit.log，否则会引发错误。

#### H. 45. 4. 1. ltaudit.log

指定会话审计日志记录哪些语句类别。可能的值为：

- 读取：当源是关系或查询时，SELECT和COPY。
- 写入：当目标是关系时，INSERT、UPDATE、DELETE、TRUNCATE 和COPY。
- 函数：函数调用和DO块。
- 角色：与角色和权限相关的语句：GRANT、REVOKE、CREATE/ALTER/DROP ROLE。
- DDL：不包含在角色类中的所有DDL。
- 其他：杂项命令，例如DISCARD、FETCH、CHECKPOINT、VACUUM、SET。
- 其他\_SET：杂项SET命令，例如SET ROLE。
- 全部：包括上述所有。

可以使用逗号分隔的列表提供多个类别，并且可以通过在类别前面加上-符号来减去类别（请参见 会话审计日志记录）。

默认值为none。

#### H. 45. 4. 2. `ltaudit.log_catalog`

指定在语句中的所有关系都在`pg_catalog`中时应启用会话日志记录。禁用此设置将减少像psql和PgAdmin这样的工具从目录中查询产生的日志噪音。

默认值为on。

#### H. 45. 4. 3. `ltaudit.log_client`

指定日志消息是否对客户端进程（如psql）可见。通常应禁用此设置，但可能对调试或其他目的有用。

请注意，只有在`ltaudit.log_client`为on时才启用`ltaudit.log_level`。

默认值为off。

#### H. 45. 4. 4. `ltaudit.log_level`

指定用于日志条目的日志级别（请参见 消息级别以获取有效的级别），但请注意，不允许使用ERROR、FATAL和PANIC。此设置用于回归测试，对于测试或其他用途，也可能对最终用户有用。

请注意，只有在`ltaudit.log_client`为on时才启用`ltaudit.log_level`；否则将使用默认值。

默认值为log。

#### H. 45. 4. 5. `ltaudit.log_parameter`

指定审计日志记录应包括与语句一起传递的参数。当参数存在时，它们将在语句文本之后以CSV格式包含在内。

默认值为off。

#### H. 45. 4. 6. `ltaudit.log_relation`

指定会话审计日志记录是否应为在SELECT或DML语句中引用的每个关系（TABLE、VIEW 等）创建单独的日志条目。这是一种有用的快捷方式，可在不使用对象审计日志记录的情况下进行详尽的日志记录。

默认值为off。

### H. 45. 4. 7. ltaudit.log\_statement\_once

指定日志记录是否将语句文本和参数与语句/子语句组合的第一个日志条目或每个条目一起记录。禁用此设置将导致日志记录不太详细，但可能会使确定生成日志条目的语句更加困难，尽管语句/子语句对以及进程ID应足以标识先前记录的语句文本。

默认值为off。

### H. 45. 4. 8. ltaudit.role

指定用于对象审计日志记录的主角色。可以通过将它们授予主角色来定义多个审计角色。这允许多个组负责审核日志记录的不同方面。

没有默认值。

## H. 45. 5. 会话审计日志记录

会话审计日志记录提供有关用户在后端执行的所有语句的详细日志记录。

### H. 45. 5. 1. 配置

通过设置 ltaudit.log 启用会话日志记录。

启用所有DML和DDL的会话日志记录，并记录DML语句中的所有关系：

```
set ltaudit.log = 'write, ddl';
set ltaudit.log_relation = on;
```

启用除MISC之外的所有命令的会话日志记录，并将审计日志消息提升为NOTICE：

```
set ltaudit.log = 'all, -misc';
set ltaudit.log_level = notice;
```

### H. 45. 5. 2. 示例

在此示例中，会话审计日志记录用于记录DDL和SELECT语句。请注意，插入语句未被记录，因为未启用WRITE 类。

SQL:

```
set ltaudit.log = 'read, ddl';

create table account
(
  id int,
  name text,
  password text,
  description text
);

insert into account (id, name, password, description)
  values (1, 'user1', 'HASH1', 'blah, blah');

select *
  from account;
```



日志输出:

```
AUDIT: SESSION, 1, 1, DDL, CREATE TABLE, TABLE, public.account, create table account
(
  id int,
  name text,
  password text,
  description text
); <not logged>
AUDIT: SESSION, 2, 1, READ, SELECT, ,, select *
from account, <not logged>
```

## H. 45. 6. 对象审计日志记录

对象审计日志记录记录影响特定关系的语句。仅支持SELECT、INSERT、UPDATE和 DELETE命令，TRUNCATE不包括在对象审计日志记录中。

对象审计日志记录旨在作为ltaudit.log = 'read, write' 的细粒度替代。因此，可能没有必要将它们与其他设置同时使用，但一个可能的情况是使用会话日志记录来捕获每个语句，然后使用对象日志记录来获取有关特定关系的更多详细信息。

### H. 45. 6. 1. 配置

对象级审计日志记录通过角色系统实现。设置 ltaudit.role定义用于审计日志记录的角色。当审计角色具有执行命令的权限或从其他角色继承权限时，关系(TABLE 、VIEW等)将被记录审计日志。这使您可以在任何上下文中实际上拥有多个审计角色，即使只有一个主角色。

将 ltaudit.role设置为auditor，并授予account表的 SELECT和DELETE权限。现在，对account表的任何SELECT 或DELETE语句都将被记录日志:

```
set ltaudit.role = 'auditor';

grant select, delete
on public.account
to auditor;
```

### H. 45. 6. 2. 示例

本示例使用对象审计日志记录来说明如何对SELECT和DML语句进行细粒度的日志记录。请注意，对account 表的日志记录受列级权限控制，而对account\_role\_map表的日志记录则受表级权限控制。

SQL:

```
set ltaudit.role = 'auditor';

create table account
(
  id int,
  name text,
  password text,
  description text
);

grant select (password)
```

```

    on public.account
    to auditor;

select id, name
  from account;

select password
  from account;

grant update (name, password)
  on public.account
  to auditor;

update account
  set description = 'yada, yada';

update account
  set password = 'HASH2';

create table account_role_map
(
  account_id int,
  role_id int
);

grant select
  on public.account_role_map
  to auditor;

select account.password,
       account_role_map.role_id
  from account
       inner join account_role_map
         on account.id = account_role_map.account_id

```

日志输出:

```

AUDIT: OBJECT, 1, 1, READ, SELECT, TABLE, public.account, select password
      from account, <not logged>
AUDIT: OBJECT, 2, 1, WRITE, UPDATE, TABLE, public.account, update account
      set password = 'HASH2', <not logged>
AUDIT: OBJECT, 3, 1, READ, SELECT, TABLE, public.account, select account.password,
      account_role_map.role_id
      from account
      inner join account_role_map
        on account.id = account_role_map.account_id, <not logged>
AUDIT: OBJECT, 3, 1, READ, SELECT, TABLE, public.account_role_map, select
      account.password,
      account_role_map.role_id
      from account
      inner join account_role_map
        on account.id = account_role_map.account_id, <not logged>

```

## H. 45. 7. 格式

审计条目将被写入标准日志记录设施，并以逗号分隔格式包含以下列。仅当删除每个日志条目的日志行前缀部分时，输出才符合CSV格式。

- AUDIT\_TYPE - SESSION或OBJECT。
- STATEMENT\_ID - 此会话的唯一语句ID。每个语句ID表示一个后端调用。即使某些语句未记录，语句ID也是连续的。当记录了多个关系时，可能会有多个与语句ID相关的条目。
- SUBSTATEMENT\_ID - 主语句内每个子语句的顺序ID。例如，在查询中调用函数。即使某些子语句未记录，子语句ID也是连续的。当记录了多个关系时，可能会有多个与子语句ID相关的条目。
- CLASS - 例如：READ, ROLE（请参见 ltaudit.log）。
- COMMAND - 例如：ALTER TABLE, SELECT。
- OBJECT\_TYPE - TABLE, INDEX, VIEW等。可用于SELECT, DML和大多数DDL语句。
- OBJECT\_NAME - 完全限定的对象名称（例如public.account）。可用于SELECT, DML和大多数DDL语句。
- STATEMENT - 在后端执行的语句。
- PARAMETER - 如果设置了ltaudit.log\_parameter，则此字段将包含用引号括起来的CSV格式的语句参数，如果没有参数，则为<none>。否则，该字段为<未记录>。

使用 log\_line\_prefix添加满足您的审计日志要求所需的任何其他字段。典型的日志行前缀可能是 '%m %u %d [%p]: '，它将为每个审计日志提供日期/时间、用户名、数据库名称和进程ID。

## H. 45.8. 注意事项

对象重命名将在它们被重命名的名称下记录。例如，重命名表将产生以下结果：

```
ALTER TABLE test RENAME TO test2;
```

```
AUDIT: SESSION, 36, 1, DDL, ALTER TABLE, TABLE, public.test2, ALTER TABLE test RENAME TO test2, <not logged>
```

可能会将一个命令记录多次。例如，在创建表时指定主键时，主键的索引将独立记录，并且在创建条目下将为索引进行其他审计日志记录。但是，多个条目将包含在一个语句ID中。

自动清理（Autovacuum）和自动分析（Autoanalyze）不会记录日志。

在事务进入中止状态之后执行的语句将不会被审计日志记录。但是，导致错误的语句以及在中止事务中执行的任何后续语句将由标准日志记录设施记录为错误。

## H. 46. ltcrypto

ltcrypto 模块为 LightDB 提供了加密功能。

该模块被认为是“trusted”（可信任的），也就是说，它可以被非超级用户安装，只要他们在当前数据库上具有“CREATE”特权。

### H. 46.1. 通用哈希函数

#### H. 46.1.1. digest()

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

计算给定的data的二进制哈希值。 type是要使用的算法。 标准算法有md5、sha1、sha224、sha256、sha384和sha512。 如果使用 OpenSSL 编译了 lcrypto, 则可以使用更多算法, 详见 Table H. 61

如果您想将哈希值作为十六进制字符串返回, 请对结果使用encode()函数。例如:

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

### H. 46. 1. 2. hmac()

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

计算带有密钥key的data的哈希MAC。 type与 digest()中的一致。

这类似于digest(), 但只有知道密钥才能重新计算哈希值。 这可以防止有人篡改数据并更改哈希值以匹配。

如果密钥大于哈希块大小, 它将首先被哈希, 然后使用哈希结果作为密钥。

## H. 46. 2. 密码哈希函数

函数crypt()和gen\_salt()专门用于哈希密码。 crypt()用于哈希, gen\_salt()用于准备算法参数。

crypt()中的算法与通常的MD5或SHA1哈希算法有以下区别:

1. 它们不够快。由于数据量很小, 这是使暴力破解密码变得困难的唯一方法。
2. 它们使用一个随机值, 称为salt, 以便具有相同密码的用户将具有不同的加密密码。 这也是对抗反向算法的额外防御措施。
3. 它们在结果中包含算法类型, 因此可以共存使用使用不同算法哈希的密码。
4. 其中一些是适应性的, 这意味着当计算机变得更快时, 可以调整算法使其变慢, 而不会与现有密码不兼容。

Table H. 58列出了crypt()函数支持的算法。

Table H. 58. crypt()支持的算法

算法	最大密码长度	适应性?	盐位数	输出长度	描述
bf	72	yes	128	60	基于Blowfish的, 2a变体
md5	无限制	否	48	34	基于MD5的crypt
xdes	8	yes	24	20	扩展DES
des	8	否	12	13	原始UNIX crypt

### H. 46. 2. 1. crypt()

```
crypt(password text, salt text) 返回 text
```

计算password的crypt(3)格式哈希值。当存储新密码时，需要使用gen\_salt()生成新的salt值。要检查密码，请将存储的哈希值作为salt传递，并测试结果是否与存储的值匹配。

设置新密码的示例：

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

验证的示例：

```
SELECT (pswhash = crypt('entered password', pswhash)) AS pswmatch FROM ... ;
```

如果输入的密码正确，则返回true。

## H. 46. 2. 2. gen\_salt()

gen\_salt(type text [, iter\_count integer ]) 返回 text

为crypt()生成一个新的随机盐字符串。盐字符串还告诉crypt()使用哪个算法。

type参数指定哈希算法。接受的类型包括：des, xdes, md5和 bf。

iter\_count参数允许用户指定迭代次数（对于具有迭代次数的算法）。迭代次数越高，哈希密码所需的时间就越长，因此破解密码所需的时间也越长。但是，如果迭代次数过高，计算哈希值所需的时间可能会长达数年，这在实际中是不可行的。如果省略iter\_count参数，则使用默认的迭代次数。iter\_count参数的允许值取决于算法，并在 Table H. 59中显示。

Table H. 59. crypt()的迭代次数

算法	默认值	最小值	最大值
xdes	725	1	16777215
bf	6	4	31

对于xdes，还有一个额外的限制，即迭代次数必须是奇数。

为了选择适当的迭代次数，请考虑原始的DES crypt是为每秒4个哈希值的硬件设计的。每秒慢于4个哈希值可能会降低可用性。每秒快于100个哈希值可能太快了。

Table H. 60提供了不同哈希算法相对较慢的概述。该表格显示了尝试在8个字符密码中所有字符组合所需的时间，假设密码仅包含小写字母，或大写和小写字母和数字。在crypt-bf条目中，斜杠后面的数字是gen\_salt的 iter\_count参数。

Table H. 60. 哈希算法速度

算法	哈希值/秒	对于[a-z]	对于[A-Za-z0-9]	相对于md5 hash的持续时间
crypt-bf/8	1792	4年	3927年	100k
crypt-bf/7	3648	2年	1929年	50k
crypt-bf/6	7168	1年	982年	25k
crypt-bf/5	13504	188 天	521 年	12.5k
crypt-md5	171584	15 天	41 年	1k

算法	哈希值/秒	对于[a-z]	对于[A-Za-z0-9]	相对于md5 hash的持续时间
crypt-des	23221568	157.5 分钟	108 天	7
sha1	37774272	90 分钟	68 天	4
md5 (哈希)	150085504	22.5 分钟	17 天	1

注意事项:

- 使用的计算机是一台英特尔移动版Core i3。
- crypt-des 和 crypt-md5 算法的编号来源于John the Ripper v1.6.38的“-test”输出。
- md5 哈希编号来自于mdcrack 1.2。
- sha1 的编号来自于lcrack-20031130-beta。
- crypt-bf 的编号是使用一个简单的程序，循环遍历1000个8个字符的密码获得的。这样可以显示不同迭代次数的速度。供参考: john -test 显示 crypt-bf/5 的循环速度为每秒13506次。(结果的微小差异符合 ltcrypto 中的 crypt-bf 实现与 John the Ripper 中使用的实现相同的事实。)

请注意，“尝试所有组合”并不是一项现实的任务。通常密码破解是通过使用包含常规单词和各种变形的字典来完成的。因此，即使是类似单词的密码也可能比上述数字快得多被破解，而一个6个字符的不像单词的密码可能会逃脱破解。也可能不会。

## H. 46. 3. PGP Encryption Functions

这里的函数实现了OpenPGP(RFC 4880)标准的加密部分。支持对称密钥和公钥加密。

加密的PGP消息由两个部分或数据包(packets)组成:

- 包含会话密钥的数据包 — 可以是对称密钥或公钥加密。
- 包含使用会话密钥加密的数据的数据包。

使用对称密钥(即密码)进行加密时:

1. 给定的密码使用String2Key (S2K)算法进行哈希。这与crypt() 算法非常相似，目的是缓慢处理并带有随机盐，但它会生成一个完整长度的二进制密钥。
2. 如果请求一个单独的会话密钥，将生成一个新的随机密钥。否则，S2K密钥将直接用作会话密钥。
3. 如果要直接使用S2K密钥，则会仅将S2K设置放入会话密钥数据包中。否则，会话密钥将使用S2K密钥进行加密，并放入会话密钥数据包中。

使用公钥加密时:

1. 生成一个新的随机会话密钥。
2. 使用公钥对其进行加密，并将其放入会话密钥数据包中。

在任何一种情况下，要加密的数据都按以下方式处理:

1. 可选的数据处理: 压缩、转换为UTF-8和/或换行符的转换。
2. 数据前缀有一段随机字节块。这相当于使用随机IV。
3. 附加随机前缀和数据的SHA1哈希值。
4. 所有这些都使用会话密钥加密，并放入数据包中。

### H. 46. 3. 1. `pgp_sym_encrypt()`

`pgp_sym_encrypt(data text, psw text [, options text ])` 返回  
bytea  
`pgp_sym_encrypt_bytea(data bytea, psw text [, options text ])` 返  
回 bytea

使用对称PGP密钥psw加密data。参数options 可以包含选项设置，如下所述。

### H. 46. 3. 2. `pgp_sym_decrypt()`

`pgp_sym_decrypt(msg bytea, psw text [, options text ])` 返回 text  
`pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ])` 返  
回 bytea

解密对称密钥加密的 PGP 消息。

使用 `pgp_sym_decrypt` 解密 bytea 数据是被禁止的。这是为了避免输出无效的字符数据。  
使用 `pgp_sym_decrypt_bytea` 解密原始文本数据是可以的。

options 参数可以包含以下描述的选项设置。

### H. 46. 3. 3. `pgp_pub_encrypt()`

`pgp_pub_encrypt(data text, key bytea [, options text ])` 返回  
bytea  
`pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ])`  
返回 bytea

使用公共 PGP 密钥 key 加密 data。给予此函数一个私钥将会产生错误。

options 参数可以包含以下描述的选项设置。

### H. 46. 3. 4. `pgp_pub_decrypt()`

`pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options  
text ]])` 返回 text  
`pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [,  
options text ]])` 返回 bytea

解密一个公钥加密的消息。key 必须是用于加密的公钥对应的私钥。如果私钥受到密码保护，则必须在 psw 中提供密码。如果没有密码，但您想指定选项，则需要提供空密码。

使用 `pgp_pub_decrypt` 解密 bytea 数据是被禁止的。这是为了避免输出无效的字符数据。  
使用 `pgp_pub_decrypt_bytea` 解密原始文本数据是可以的。

options 参数可以包含以下描述的选项设置。

### H. 46. 3. 5. `pgp_key_id()`

`pgp_key_id(bytea)` 返回 `text`

`pgp_key_id` 提取 PGP 公钥或私钥的密钥 ID。如果给定加密消息，则提供用于加密数据的密钥 ID。

它可以返回两个特殊的密钥 ID:

- SYMKEY

该消息使用对称密钥加密。

- ANYKEY

消息是使用公钥加密的，但密钥 ID 已被删除。这意味着您需要尝试所有的私钥才能看到哪一个可以解密它。`ltpcrypto` 本身不会产生这样的消息。

请注意，不同的密钥可能具有相同的 ID。这是罕见但正常的事件。客户端应用程序应尝试使用每个密钥进行解密，以查看哪个适合 — 就像处理 ANYKEY 一样。

### H. 46. 3. 6. `armor()`, `dearmor()`

`armor(data bytea [ , keys text[], values text[] ])` 返回 `text`  
`dearmor(data text)` 返回 `bytea`

这些函数将二进制数据封装/解封为 PGP ASCII-armor 格式，这基本上是带有 CRC 和附加格式的 Base64。

如果指定了 `keys` 和 `values` 数组，则为每个键/值对向装甲格式添加一个 armor header。两个数组必须是单维数组，并且它们的长度必须相同。键和值不能包含任何非 ASCII 字符。

### H. 46. 3. 7. `pgp_armor_headers`

`pgp_armor_headers(data text, key out text, value out text)` 返回  
`setof record`

`pgp_armor_headers()` 从 `data` 中提取装甲头。返回值是具有两个列（键和值）的行集。如果键或值包含任何非 ASCII 字符，则将其视为 UTF-8。

### H. 46. 3. 8. PGP 函数的选项

选项的命名方式类似于 GnuPG。选项的值应在等号后给出；用逗号将选项分开。例如：

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

除了 `convert-crlf` 之外的所有选项都仅适用于加密函数。解密函数从 PGP 数据获取参数。

最有趣的选项可能是 `compress-algo` 和 `unicode-mode`。其余选项应具有合理的默认值。

#### H. 46. 3. 8. 1. `cipher-algo`

使用哪种密码算法。

Values: bf, aes128, aes192, aes256 (仅 OpenSSL: 3des, cast5)



Default: aes128  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

#### H. 46. 3. 8. 2. compress-algo

使用哪种压缩算法。仅在使用 zlib 构建 LightDB 时可用。

Values:  
0 - 不压缩  
1 - ZIP 压缩  
2 - ZLIB 压缩 (= ZIP 加上元数据和块 CRC)  
Default: 0  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

#### H. 46. 3. 8. 3. compress-level

压缩程度。较高的级别可以压缩更小的数据，但速度较慢。0 禁用压缩。

Values: 0, 1-9  
Default: 6  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

#### H. 46. 3. 8. 4. convert-crlf

在加密时是否将 \n 转换为 \r\n，在解密时将 \r\n 转换为 \n。RFC 4880 指定文本数据应使用 \r\n 换行符进行存储。使用此选项可以获得完全符合 RFC 的行为。

Values: 0, 1  
Default: 0  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt, pgp\_sym\_decrypt, pgp\_pub\_decrypt

#### H. 46. 3. 8. 5. disable-mdc

不使用 SHA-1 保护数据。使用此选项的唯一好处是实现与古老的 PGP 产品的兼容性，这些产品早于在 RFC 4880 中添加 SHA-1 保护数据包。最近的 gnupg.org 和 pgp.com 软件都可以很好地支持它。

Values: 0, 1  
Default: 0  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

#### H. 46. 3. 8. 6. sess-key

使用单独的会话密钥。公钥加密总是使用单独的会话密钥；此选项用于对称密钥加密，默认情况下直接使用 S2K 密钥。

Values: 0, 1  
Default: 0  
Applies to: pgp\_sym\_encrypt

#### H. 46. 3. 8. 7. s2k-mode

使用哪种 S2K 算法。

Values:  
0 - 不使用盐。危险!  
1 - 使用盐但使用固定的迭代次数。  
3 - 可变的迭代次数。  
Default: 3  
Applies to: pgp\_sym\_encrypt

#### H. 46. 3. 8. 8. s2k-count

使用 S2K 算法的迭代次数。它必须是介于 1024 和 65011712 之间的值，包括这两个值。

Default: 介于 65536 和 253952 之间的随机值  
Applies to: 仅在 s2k-mode=3 时适用于 pgp\_sym\_encrypt

#### H. 46. 3. 8. 9. s2k-digest-algo

在 S2K 计算中使用哪种摘要算法。

Values: md5, sha1  
Default: sha1  
Applies to: pgp\_sym\_encrypt

#### H. 46. 3. 8. 10. s2k-cipher-algo

用于加密单独会话密钥的密码算法。

Values: bf, aes, aes128, aes192, aes256  
Default: 使用 cipher-algo  
Applies to: pgp\_sym\_encrypt

#### H. 46. 3. 8. 11. unicode-mode

是否将数据库内部编码的文本数据转换为 UTF-8 编码并转换回来。如果您的数据库已经是 UTF-8，则不会进行转换，但是消息将被标记为 UTF-8。如果没有此选项，则不会进行标记。

Values: 0, 1  
Default: 0  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

### H. 46. 3. 9. 使用 GnuPG 生成 PGP 密钥

生成新密钥:

gpg --gen-key

首选密钥类型为 “DSA 和 Elgamal”。

对于 RSA 加密，您必须创建 DSA 或 RSA 仅签名密钥作为主密钥，然后使用 `gpg --edit-key` 添加 RSA 加密子密钥。

列出密钥：

```
gpg --list-secret-keys
```

以 ASCII-armor 格式导出公钥：

```
gpg -a --export KEYID > public.key
```

以 ASCII-armor 格式导出秘密密钥：

```
gpg -a --export-secret-keys KEYID > secret.key
```

在将这些密钥提供给 PGP 函数之前，您需要使用 `dearmor()` 对其进行转换。或者，如果您可以处理二进制数据，则可以从命令中删除 `-a`。

有关更多详细信息，请参见 `man gpg`，GNU 隐私手册<sup>4</sup> 和其他文档 <https://www.gnupg.org/>。

### H. 46. 3. 10. PGP 代码的限制

- 没有签名支持。这也意味着不会检查加密子密钥是否属于主密钥。
- 不支持以加密密钥作为主密钥。由于通常不鼓励这种做法，因此这不应该是一个问题。
- 不支持多个子密钥。这可能看起来像是一个问题，因为这是常见的做法。另一方面，您不应该使用您的常规 GPG/PGP 密钥与 `ltpcrypto`，而是要创建新的密钥，因为使用场景是非常不同的。

## H. 46. 4. 原始加密函数

这些函数仅在数据上运行密码算法；它们没有 PGP 加密的任何高级功能。因此，它们存在一些重大问题：

1. 它们直接使用用户密钥作为密码密钥。
2. 它们不提供任何完整性检查，无法查看加密数据是否已被修改。
3. 它们期望用户自己管理所有加密参数，甚至是初始向量（IV）。
4. 它们不处理文本。

因此，随着 PGP 加密的引入，不鼓励使用原始加密函数。

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea
```

```
encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

使用 `type` 指定的密码算法加密/解密数据。`type` 字符串的语法为：

<sup>4</sup> <https://www.gnupg.org/gph/en/manual.html>

```

algorithm
[
  -
  mode
]
[
  /pad:
  padding
]

```

其中 `algorithm` 是以下之一：

- `bf` — Blowfish
- `aes` — AES (Rijndael-128、-192 或 -256)

而 `mode` 是以下之一：

- `cbc` — 下一个块依赖于前一个块（默认）
- `ecb` — 每个块都单独加密（仅用于测试）

而 `padding` 是以下之一：

- `pkcs` — 数据可以是任意长度（默认）
- `none` — 数据必须是密码块大小的倍数

因此，例如，这些是等价的：

```

encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')

```

在 `encrypt_iv` 和 `decrypt_iv` 中，`iv` 参数是 CBC 模式的初始值；对于 ECB，它将被忽略。如果不完全是块大小，则它将被剪切或填充为零。在没有此参数的函数中，默认值为全零。

## H. 46. 5. 随机数据函数

`gen_random_bytes(count integer)` 返回 `bytea`

返回 `count` 个具有密码学强度的随机字节。一次最多可以提取 1024 个字节。这是为了避免耗尽随机数生成器池。

`gen_random_uuid()` 返回 `uuid`

返回版本 4（随机）UUID。（已过时，此函数现在也包含在核心 `LightDB` 中。）

## H. 46. 6. 注释

### H. 46. 6. 1. 配置

`ltdcrypto` 根据主 `LightDB configure` 脚本的结果来进行配置。影响它的选项有 `--with-zlib` 和 `--with-openssl`。

当使用zlib编译时，PGP加密函数可以在加密之前压缩数据。

当使用OpenSSL编译时，将会有更多的算法可用。同时，由于OpenSSL拥有更优化的BIGNUM函数，公钥加密函数将会更快。

Table H. 61. 使用和不使用OpenSSL的功能概述

功能	内置	使用OpenSSL
MD5	是	是
SHA1	是	是
SHA224/256/384/512	是	是
其他摘要算法	否	是（注1）
Blowfish	是	是
AES	是	是
DES/3DES/CAST5	否	是
原始加密	是	是
PGP对称加密	是	是
PGP公钥加密	是	是

当使用 OpenSSL 3.0.0及更高版本进行编译时，必须在 openssl.cnf 配置文件中激活旧的提供程序，以便使用旧的密码算法，如DES或Blowfish。

注意：

1. OpenSSL支持的任何摘要算法都会自动被使用。这对于密码算法来说并不适用，它需要显式地被支持。

### H. 46. 6. 2. NULL值处理

正如在SQL中的标准操作，如果任何一个参数为NULL，所有函数都会返回NULL。这可能会在不小心使用时造成安全风险。

### H. 46. 6. 3. 安全限制

所有 ltcrypto 函数在数据库服务器内运行。这意味着所有数据和密码在 ltcrypto 和客户端应用程序之间以明文形式传输。因此，您必须：

1. 本地连接或使用SSL连接。
2. 信任系统和数据库管理员。

如果您不能，那么最好在客户端应用程序内部进行加密。

该实现不抵御侧信道攻击。例如，ltcrypto 解密函数完成所需的时间在给定大小的密文之间会变化。

### H. 46. 6. 4. 有用的阅读材料

- <https://www.gnupg.org/gph/en/manual.html>  
GNU隐私手册。
- <https://www.openwall.com/crypt/>  
描述了crypt-blowfish算法。

- <https://www.iusmentis.com/security/passphrasefaq/>  
如何选择一个好的密码。
- <http://world.std.com/~reinhold/diceware.html>  
选择密码的有趣想法。
- <sup>5</sup>  
描述了好的和坏的加密。

### H. 46. 6. 5. 技术参考

- <https://tools.ietf.org/html/rfc4880>  
OpenPGP消息格式。
- <https://tools.ietf.org/html/rfc1321>  
MD5消息摘要算法。
- HMAC: Keyed-Hashing for Message Authentication<sup>6</sup>  
HMAC: 用于消息认证的密钥哈希。
- A Future-Adaptable Password Scheme<sup>7</sup>  
比较crypt-des、crypt-md5和bcrypt算法。
- Fortuna (PRNG)<sup>8</sup>  
Fortuna CSPRNG的描述。
- Jesse Cooke's page on random numbers and cryptography<sup>9</sup>  
Jean-Luc Cooke基于Fortuna的Linux /dev/random驱动程序。

### H. 46. 7. 作者

Marko Kreen <markokr@gmail.com>

ltdcrypto 使用以下来源的代码:

算法	作者	源起
DES密码加密	David Burren和其他人	FreeBSD libcrypt
MD5密码加密	Poul-Henning Kamp	FreeBSD libcrypt
Blowfish密码加密	Solar Designer	www.openwall.com
Blowfish密码算法	Simon Tatham	PuTTY
Rijndael密码算法	Brian Gladman	OpenBSD sys/crypto
MD5 hash and SHA1	WIDE Project	KAME kame/sys/crypto
SHA256/384/512	Aaron D. Gifford	OpenBSD sys/crypto

<sup>5</sup> <http://www.interhack.net/people/cmcurtin/snake-oil-faq.html>

<sup>6</sup> <https://tools.ietf.org/html/rfc2104>

<sup>7</sup> <https://www.usenix.org/legacy/events/usenix99/provos.html>

<sup>8</sup> [https://en.wikipedia.org/wiki/Fortuna\\_\(PRNG\)](https://en.wikipedia.org/wiki/Fortuna_(PRNG))

<sup>9</sup> <https://jlcooke.ca/random/>

算法	作者	源起
BIGNUM math	Michael J. Fromberger	dartmouth.edu/~sting/sw/imath

## H. 47. ltfce

ltfce - 所有模式的兼容函数和运算符。

### H. 47. 1. 所有模式兼容的功能

提供了与所有模式兼容的功能。函数和运算符是在 `lt_catalog` 下创建的。下表列出了可以在所有模式中使用的功能。

Table H. 62. 函数

项	概述
DATABASE	返回当前模式作为当前数据库
LOCATE	返回字符串中子字符串的位置

Table H. 63. 运算符类型兼容性

项
LEFTARG:int2 RIGHTARG:text
LEFTARG:int4 RIGHTARG:text
LEFTARG:int8 RIGHTARG:text
LEFTARG:text RIGHTARG:int2
LEFTARG:text RIGHTARG:int4
LEFTARG:text RIGHTARG:int8
LEFTARG:text RIGHTARG:numeric
LEFTARG:numeric RIGHTARG:text

### H. 47. 2. 函数

- DATABASE
- LOCATE

#### H. 47. 2. 1. 数据库

描述

是 `current_schema()` 的同义词。在 MySQL 中，模式的概念类似于数据库。您可以通过设置搜索路径来指定当前模式。

语法

```
database() RETURNS text
```

通用规则

- 返回搜索路径中第一个模式的名称（如果搜索路径为空，则返回 `null` 值）。

- 与 MySQL 类似，如果创建表或其他命名对象时未指定目标模式，则将使用此模式。

示例

```
lightdb@lt_test=# select database();
database
-----
public
(1 row)
```

## H. 47. 2. 2. LOCATE

描述

返回第一个子字符串的位置。

语法

```
LOCATE(substr TEXT, str TEXT) RETURNS INTEGER
LOCATE(substr TEXT, str TEXT, POS INTEGER) RETURNS INTEGER
```

通用规则

- 第一种语法返回子字符串 `substr` 在字符串 `str` 中第一次出现的位置。
- 第二种语法返回从位置 `pos` 开始，子字符串 `substr` 在字符串 `str` 中第一次出现的位置。
- 如果 `substr` 不在 `str` 中，则返回 0。
- 如果任何参数为 NULL，则返回 NULL。

示例

在以下示例中，返回从位置 5 开始，在字符串 'foobarbar' 中子字符串 'bar' 的位置。

```
SELECT LOCATE('bar', 'foobarbar', 5);
locate
-----
7
(1 row)
```

## H. 47. 3. 运算符类型兼容性

为文本类型和数值类型 (int2、int4、int8、numeric) 之间的运算符 (例如 '+', '-', '\*', '/', '<', '<=', '>', '>=', '=', '<>' 和 '%') 添加类型重载。

示例

使用它，您可以直接将 '1.1'::text 与 1 相加。

```
lightdb@test=# select '1.1'::text+1;
?column?
-----
```



```
2.1
(1 row)
```

如果在 Oracle 或 MySQL 兼容类型中（请参见 `lightdb_syntax_compatible_type`），根据“未知”可以在某些情况下转换为“text”的规则（有关详细信息，请参见 Chapter 10，您可以直接将 '1.1' 与 1 相加。

```
lightdb@test=# select '1.1'+1;
?column?
```

```
2.1
(1 row)
```

## H. 48. ltfincore

一组用于从 LightDB 中管理内存页面的函数

一组使用 `mincore` 探索缓存内存来处理关系低级管理的函数。

### H. 48. 1. DESCRIPTION

在 LightDB 中，每个表或索引被分割为（通常为）1GB 的段，每个段在内存中被分割为页面，然后在文件系统中被分割为块。

这些函数允许您了解操作系统页面缓存中关系的哪些磁盘块以及数量。它可以将结果提供为 `VarBit`，并可以存储在表中。然后使用此表，可以为关系的每个块恢复页面缓存状态，甚至在另一个服务器上，感谢流复制。

其他函数用于在整个关系（每个段）上设置 `POSIX_FADVISE` 标志。最有用的可能是 `WILLNEED` 和 `DONTNEED`，分别将关系每个段的块推入和弹出页面缓存。

每个函数至少使用表名或索引名（或 `oid`）作为参数调用，并遍历关系的每个段。

### H. 48. 2. EXAMPLES

这里是一些用法示例。如果您想要更多细节，请参阅文档。

#### H. 48. 2. 1. 获取关系的当前状态

可能有用：

```
cedric=# select * from pgfincore('ltbench_accounts');
 relpath      | segment | os_page_size | rel_os_pages | pages_mem |
 group_mem    | os_pages_free | databit | pages_dirty | group_dirty
-----+-----+-----+-----+-----+-----
base/11874/16447 | 0 | 4096 | 262144 | 262144 |
 1 | 81016 | 0 | 0 | 0 |
base/11874/16447.1 | 1 | 4096 | 65726 | 65726 |
 1 | 81016 | 0 | 0 | 0 |
(2 rows)
```

Time: 31.563 ms

## H. 48. 2. 2. 将表或索引加载到 OS 页面缓冲区中

您可能希望尝试将表或索引保留在 OS 页面缓存中，或在执行您熟知的大型查询之前预加载表（减少查询时间）。

要做到这一点，只需执行以下查询：

```
cedric=# select * from pgfadvise_willneed('ltbench_accounts');
      relpath      | os_page_size | rel_os_pages | os_pages_free
-----+-----+-----+-----
 base/11874/16447 |         4096 |         262144 |         169138
 base/11874/16447.1 |         4096 |          65726 |         103352
(2 rows)
```

Time: 4462,936 ms

- 列 `os_page_size` 报告页面大小为 4KB。
- 列 `rel_os_pages` 是指定文件的页面数。
- 列 `os_pages_free` 是内存中（用于缓存）的空闲页面数。

## H. 48. 2. 3. 快照和恢复表或索引（或更多）的 OS 页面缓冲区状态

您可能希望将表或索引还原到 OS 页面缓存中，就像在快照时一样。例如，如果您必须重新启动服务器，则当 LightDB 启动时，由于 LightDB 或操作系统关于涉及这些第一个查询的关系的缓存中都没有页面，因此第一个查询可能会变慢。

执行快照和恢复非常简单：

```
-- 快照
cedric=# create table pgfincore_snapshot as
cedric=#   select 'ltbench_accounts'::text as relname,*,now() as date_snapshot
cedric=#   from pgfincore('ltbench_accounts',true);

-- 恢复
cedric=# select * from pgfadvise_loader('ltbench_accounts', 0, true, true,
      (select databit from pgfincore_snapshot
      where relname='ltbench_accounts' and segment = 0));
      relpath      | os_page_size | os_pages_free | pages_loaded |
      pages_unloaded
-----+-----+-----+-----+-----
+-----+
 base/11874/16447 |         4096 |         80867 |         262144 |
(1 row)
```

Time: 35.349 ms

- 列 `pages_loaded` 报告已读取到内存中的页面数（它们可能已经在内存中）。
- 列 `pages_unloaded` 报告已从内存中移除的页面数（它们可能不在内存中）；

## H. 48. 3. 概要

`pgsysconf(OUT os_page_size bigint, OUT os_pages_free bigint, OUT os_total_pages bigint) RETURNS record`

`pgsysconf_pretty(OUT os_page_size text, OUT os_pages_free text, OUT os_total_pages text) RETURNS record`

pgfadvise(IN relname regclass, IN fork text, IN action int, OUT relpath text, OUT os\_page\_size bigint, OUT rel\_os\_pages bigint, OUT os\_pages\_free bigint) RETURNS setof record

pgfadvise\_willneed(IN relname regclass, OUT relpath text, OUT os\_page\_size bigint, OUT rel\_os\_pages bigint, OUT os\_pages\_free bigint) RETURNS setof record

pgfadvise\_dontneed(IN relname regclass, OUT relpath text, OUT os\_page\_size bigint, OUT rel\_os\_pages bigint, OUT os\_pages\_free bigint) RETURNS setof record

pgfadvise\_normal(IN relname regclass, OUT relpath text, OUT os\_page\_size bigint, OUT rel\_os\_pages bigint, OUT os\_pages\_free bigint) RETURNS setof record

pgfadvise\_sequential(IN relname regclass, OUT relpath text, OUT os\_page\_size bigint, OUT rel\_os\_pages bigint, OUT os\_pages\_free bigint) RETURNS setof record

pgfadvise\_random(IN relname regclass, OUT relpath text, OUT os\_page\_size bigint, OUT rel\_os\_pages bigint, OUT os\_pages\_free bigint) RETURNS setof record

pgfadvise\_loader(IN relname regclass, IN fork text, IN segment int, IN load bool, IN unload bool, IN databit varbit, OUT relpath text, OUT os\_page\_size bigint, OUT os\_pages\_free bigint, OUT pages\_loaded bigint, OUT pages\_unloaded bigint) RETURNS setof record

pgfadvise\_loader(IN relname regclass, IN segment int, IN load bool, IN unload bool, IN databit varbit, OUT relpath text, OUT os\_page\_size bigint, OUT os\_pages\_free bigint, OUT pages\_loaded bigint, OUT pages\_unloaded bigint) RETURNS setof record

pgfincore(IN relname regclass, IN fork text, IN getdatabit bool, OUT relpath text, OUT segment int, OUT os\_page\_size bigint, OUT rel\_os\_pages bigint, OUT pages\_mem bigint, OUT group\_mem bigint, OUT os\_pages\_free bigint, OUT databit varbit, OUT pages\_dirty bigint, OUT group\_dirty bigint) RETURNS setof record

pgfincore(IN relname regclass, IN getdatabit bool, OUT relpath text, OUT segment int, OUT os\_page\_size bigint, OUT rel\_os\_pages bigint, OUT pages\_mem bigint, OUT group\_mem bigint, OUT os\_pages\_free bigint, OUT databit varbit, OUT pages\_dirty bigint, OUT group\_dirty bigint) RETURNS setof record

pgfincore(IN relname regclass, OUT relpath text, OUT segment int, OUT os\_page\_size bigint, OUT rel\_os\_pages bigint, OUT pages\_mem bigint, OUT group\_mem bigint, OUT os\_pages\_free bigint, OUT databit varbit, OUT pages\_dirty bigint, OUT group\_dirty bigint) RETURNS setof record

## H. 48. 4. 文档

### H. 48. 4. 1. pgsysconf

这个函数输出操作系统块的大小以及操作系统页面缓冲区中的空闲页面数。

```
cedric=# select * from pgsysconf();
 os_page_size | os_pages_free | os_total_pages
-----+-----+-----
          4096 |          80431 |          4094174
```

### H. 48. 4. 2. pgsysconf\_pretty

与上面相同，但有更好的输出形式。

```
cedric=# select * from pgsysconf_pretty();
```

os_page_size	os_pages_free	os_total_pages
4096 bytes	314 MB	16 GB

### H. 48. 4. 3. pgfadvise\_WILLNEED

此函数在当前关系上设置WILLNEED标志。这意味着操作系统将尝试尽可能多地加载关系的页面。主要思想是在服务器启动时预加载文件，可能使用缓存命中/未命中比率或最常使用的关系/索引。

```
cedric=# select * from pgfadvise_willneed('ltbench_accounts');
```

relpath	os_page_size	rel_os_pages	os_pages_free
base/11874/16447	4096	262144	80650
base/11874/16447.1	4096	65726	80650

### H. 48. 4. 4. pgfadvise\_DONTNEED

此函数在当前关系上设置DONTNEED标志。这意味着如果需要释放一些内存，操作系统将首先卸载文件的页面。主要思想是在文件不再有用时卸载它们（而不是卸载可能更有趣的页面）。

```
cedric=# select * from pgfadvise_dontneed('ltbench_accounts');
```

relpath	os_page_size	rel_os_pages	os_pages_free
base/11874/16447	4096	262144	342071
base/11874/16447.1	4096	65726	408103

### H. 48. 4. 5. pgfadvise\_NORMAL

此函数在当前关系上设置NORMAL标志。

### H. 48. 4. 6. pgfadvise\_SEQUENTIAL

此函数在当前关系上设置SEQUENTIAL标志。

### H. 48. 4. 7. pgfadvise\_RANDOM

此函数在当前关系上设置RANDOM标志。

### H. 48. 4. 8. pgfadvise\_loader

此函数允许直接与页面缓存交互。它可以根据表示加载/卸载页面映射的varbit从内存中加载和/或卸载页面。

使用关系ltbench\_accounts，段0和任意的varbit映射：

```
-- Loading and Unloading
cedric=# select * from pgfadvise_loader('ltbench_accounts', 0, true, true,
B'111000');
```

relpath	os_page_size	os_pages_free	pages_loaded	pages_unloaded
base/11874/16447	4096	408376	3	3

```
-- Loading
cedric=# select * from pgfadvise_loader('ltbench_accounts', 0, true, false,
    B'111000');
    relpath      | os_page_size | os_pages_free | pages_loaded |
    pages_unloaded
-----+-----+-----+-----+-----
base/11874/16447 |          4096 |         408370 |           3 |
                                                         0

-- Unloading
cedric=# select * from pgfadvise_loader('ltbench_accounts', 0, false, true,
    B'111000');
    relpath      | os_page_size | os_pages_free | pages_loaded |
    pages_unloaded
-----+-----+-----+-----+-----
base/11874/16447 |          4096 |         408370 |           0 |
                                                         3
```

### H. 48. 4. 9. pgfincore

此函数提供有关文件系统缓存的信息（页面缓存）。

```
cedric=# select * from pgfincore('ltbench_accounts');
    relpath      | segment | os_page_size | rel_os_pages | pages_mem |
    group_mem | os_pages_free | databit | pages_dirty | group_dirty
-----+-----+-----+-----+-----+-----
base/11874/16447 |         0 |          4096 |          262144 |           3 |
1 |         408444 |           0 |           0 |
base/11874/16447.1 |         1 |          4096 |          65726 |           0 |
0 |         408444 |           0 |           0 |
```

对于指定的关系，它返回：

- relpath: 关系路径
- segment: 分析的段号
- os\_page\_size: 一页的大小
- rel\_os\_pages: 关系的总页面数
- pages\_mem: 在页面缓存中关系的总页面数。（不是来自LightDB的共享缓冲区，而是操作系统缓存中的页面）
- group\_mem: 相邻的pages\_mem页面组数
- os\_page\_free: 操作系统页面缓存中的空闲页面数
- databit: 文件的varbit映射，因为其大小而不输出。使用pgfincore（“ltbench\_accounts”，true）以激活它。
- pages\_dirty: 如果定义了HAVE\_FINCORE常量，并且平台提供相关信息，则与pages\_mem类似，但用于脏页面。
- group\_dirty: 如果定义了HAVE\_FINCORE常量，并且平台提供相关信息，则与group\_mem类似，但用于脏页面。

### H. 48. 5. DEBUG

您可以使用以下错误级别调试ltfincore：DEBUG1和DEBUG5。

例如：

```
set client_min_messages TO debug1; -- debug5 is only usefull to trace
each block
```

## H. 48. 6. REQUIREMENTS

- `lftincore`需要`mincore()`或`fincore()`和`POSIX_FADVISE`。

## H. 48. 7. LIMITATIONS

- 当平台未提供`POSIX_FADVISE`时，`lftincore`有一个受限模式。

## H. 49. ltree

这个模块实现了一种数据类型`ltree`用于表示存储在一个层次树状结构中的数据的标签。还提供了在标签树中搜索的扩展功能。

这个模块被视为“trusted”，也就是说，它可以是由拥有`CREATE`特权的非超级用户安装在当前数据库上。

### H. 49. 1. 定义

一个标签是一个字母数字字符和下划线的序列（例如，在 C 区域中允许字符`A-Za-z0-9_`）。标签长度必须少于 256 个字符。

例子：42, Personal\_Services

一个标签路径是由点号分隔的零个或者更多个标签的序列，例如`L1.L2.L3`，它表示一个从层次树的根到一个特定节点的路径。一个标签路径的长度不能超过 65535 个标签。

例子：Top.Countries.Europe.Russia

`ltree`模块提供多种数据类型：

- `ltree`存储一个标签路径。
- `lquery`表示一个用于匹配`ltree`值的类正则表达式的模式。一个简单词匹配一个路径中的那个标签。一个星号（\*）匹配零个或更多个标签。它们可以用点连接起来，以形成一个必须匹配整个标签路径的模式。例如：

```
foo           正好匹配标签路径foo
*.foo.*      匹配任何包含标签foo的标签路径
*.foo        匹配任何最后一个标签是foo的标签路径
```

星号和简单词都可以被限定来限制它能匹配多少标签：

```
*{n}         匹配正好n个标签
*{n,}        匹配至少n个标签
*{n,m}       匹配至少n个但是最多m个标签
*{,m}        匹配最多m个标签 — 与*{0,m}相同
foo{n,m}     匹配至少 n 但是不超过 m 并发的 foo
foo{,}       匹配任何数量的并发的 foo，包括零
```

在缺乏任何显式量词的情况下，星号的默认值是匹配任意数量的标签（也就是`{,}`），而非星号项的默认值是只匹配一次（也就是`{1}`）。

有几个修饰符可以放在一个非星号的`lquery`项的末尾，使它能匹配除了精确匹配之外更多的匹配：

```
@           不区分大小写匹配，例如a@匹配A
```

\* 匹配带此前缀的任何标签，例如foo\*匹配foobar  
 % 匹配开头以下划线分隔的词

%的行为有点复杂。它尝试匹配词而不是整个标签。例如，foo\_bar%匹配foo\_bar\_baz但是不匹配foo\_barbaz。如果和\*组合，前缀匹配可以单独应用于每一个词，例如foo\_bar%\*匹配foo1\_bar2\_baz但不匹配foo1\_br2\_baz。

此外，你可以写多个带有|（OR）的可能改过的非星号项目来匹配那些项目中的任何一个（或几个），并且你可以在非星号组最前面放上！（NOT）来匹配任何不匹配那些分支的标签。量词，若有的话，位于组的末尾；它意味着作为一个整体的组的一些匹配（也就是说，一些匹配或不匹配任何替代的标签）。

这里是一个lquery的例子：

```
Top.*{0,2}.sport*@.!football|tennis{1,}.Russ*|Spain
a. b. c. d. e.
```

这个查询将匹配任何这样的标签路径：

- a. 开始于标签Top
- b. 并且接着具有 0 到 2 个标签
- c. 之后是一个开始于大小写无关的前缀sport的标签
- d. 然后有一个或多个标签，没有匹配football和tennis。
- e. 并且结尾是一个开始于Russ的标签，或者完全匹配Spain的标签。

- ltxtquery表示一种用于匹配ltree值的类全文搜索的模式。一个ltxtquery值包含词，也可能在末尾带有修饰符@、\*、%，修饰符具有和lquery中相同的含义。词可以用&（AND）、|（OR）、！（NOT）以及圆括号组合。lquery和ltxtquery的关键区别是前者匹配词时不考虑它们在标签路径中的位置。

这是一个ltxtquery的例子：

```
Europe & Russia*@ & !Transportation
```

这将匹配包含标签Europe以及任何以Russia开始（大小写不敏感）的标签的路径，但是不匹配包含标签Transportation的路径。这些词在路径中的位置并不重要。还有，当使用%时，该次可以与一个标签中任何下划线分隔的词匹配，而不管它们的位置如何。

注意：ltxtquery允许符号之间的空白，但是ltree和lquery不允许。

## H. 49. 2. 操作符和函数

类型ltree有普通比较操作符 =、<>、<、>、<=、>=。比较会按照树遍历的顺序排序，一个节点的子女按照标签文本排序。另外，还有Table H. 64中显示的特殊操作符。

Table H. 64. ltree 操作符

操作符	描述
ltree @> ltree → boolean	左参数是右参数(或相等)的祖先么？
ltree <@ ltree → boolean	左参数是右参数(或相等)的后代么？

操作符	描述
<code>ltree ~ lquery</code>	<code>→ boolean</code>
<code>lquery ~ ltree</code>	<code>→ boolean</code>
	ltree 匹配 lquery 么?
<code>ltree ? lquery[]</code>	<code>→ boolean</code>
<code>lquery[] ? ltree</code>	<code>→ boolean</code>
	ltree 在数组中匹配任何 lquery 么?
<code>ltree @ ltxtquery</code>	<code>→ boolean</code>
<code>ltxtquery @ ltree</code>	<code>→ boolean</code>
	ltree 匹配 ltxtquery 么?
<code>ltree    ltree</code>	<code>→ ltree</code>
	连接 ltree 路径。
<code>ltree    text</code>	<code>→ ltree</code>
<code>text    ltree</code>	<code>→ ltree</code>
	把文本转换为 ltree 并连接。
<code>ltree[] @&gt; ltree</code>	<code>→ boolean</code>
<code>ltree &lt;@ ltree[]</code>	<code>→ boolean</code>
	数组中包含一个 ltree 的祖先么?
<code>ltree[] &lt;@ ltree</code>	<code>→ boolean</code>
<code>ltree @&gt; ltree[]</code>	<code>→ boolean</code>
	数组中包含一个 ltree 的后代么?
<code>ltree[] ~ lquery</code>	<code>→ boolean</code>
<code>lquery ~ ltree[]</code>	<code>→ boolean</code>
	数组中包含匹配 lquery 的任何路径么?
<code>ltree[] ? lquery[]</code>	<code>→ boolean</code>
<code>lquery[] ? ltree[]</code>	<code>→ boolean</code>
	Does ltree 数组中包含匹配任何 lquery 的任何路径么?
<code>ltree[] @ ltxtquery</code>	<code>→ boolean</code>
<code>ltxtquery @ ltree[]</code>	<code>→ boolean</code>
	数组中包含匹配 ltxtquery 的任何路径么?
<code>ltree[] ?@&gt; ltree</code>	<code>→ ltree</code>
	返回作为 ltree 祖先的第一个数组条目, 如果没有则返回 NULL。
<code>ltree[] ?&lt;@ ltree</code>	<code>→ ltree</code>
	返回作为 ltree 后代的第一个数组条目, 如果没有则返回 NULL。
<code>ltree[] ?~ lquery</code>	<code>→ ltree</code>
	返回匹配 lquery 的第一个数组条目, 如果没有, 则返回 NULL。
<code>ltree[] ?@ ltxtquery</code>	<code>→ ltree</code>
	返回匹配 ltxtquery 的第一个数组条目, 如果没有, 则返回 NULL。

操作符 `<@`、`@>`、`@` 以及 `~` 有类似的、`^<@`、`^@>`、`^@`、`^~`, 只是它们不适用索引。它们只对测试目的有用。

可用的函数在 Table H.65 中。

Table H.65. ltree 函数

函数	描述 例子
<code>subltree ( ltree, start integer, end integer )</code>	<code>→ ltree</code> 返回从位置 start 到位置 end-1 的 ltree 的子路径 (从 0 开始计数)。



函数	描述 例子
	<code>subltree('Top.Child1.Child2', 1, 2) → Child1</code>
	<code>subpath ( ltree, offset integer, len integer ) → ltree</code> 返回从位置offset开始的ltree的子路径，长度为len。如果offset为负，则子路径从距离路径终点的远端开始。如果len为负，将许多标签留在路径的末尾。 <code>subpath('Top.Child1.Child2', 0, 2) → Top.Child1</code>
	<code>subpath ( ltree, offset integer ) → ltree</code> 返回从位置offset开始的ltree的子路径，扩展到路径的结束。如果offset为负，则子路径从距离路径终点的远端开始。 <code>subpath('Top.Child1.Child2', 1) → Child1.Child2</code>
	<code>nlevel ( ltree ) → integer</code> 返回路径中标签的数量。 <code>nlevel('Top.Child1.Child2') → 3</code>
	<code>index ( a ltree, b ltree ) → integer</code> 返回b在a中第一次出现的位置，如果没有发现则返回-1。 <code>index('0.1.2.3.5.4.5.6.8.5.6.8', '5.6') → 6</code>
	<code>index ( a ltree, b ltree, offset integer ) → integer</code> 返回b在a中第一次出现的位置，如果没有发现则返回-1。搜索从位置offset开始；负的offset是指从路径的末端开始的-offset标签。 <code>index('0.1.2.3.5.4.5.6.8.5.6.8', '5.6', -4) → 9</code>
	<code>text2ltree ( text ) → ltree</code> 转换 text 到 ltree。
	<code>ltree2text ( ltree ) → text</code> 转换 ltree 到 text。
	<code>lca ( ltree [, ltree [, ... ]] ) → ltree</code> 计算路径的最长公共祖先(最多可支持8个参数)。 <code>lca('1.2.3', '1.2.3.4.5.6') → 1.2</code>
	<code>lca ( ltree[] ) → ltree</code> 计算数组中的路径的最长公共祖先。 <code>lca(array['1.2.3'::ltree, '1.2.3.4']) → 1.2</code>

## H. 49. 3. 索引

ltree支持一些能加速上述操作符的索引类型：

- ltree上的 B-树索引： <、<=、=、 >=、 >
- ltree 之上的GiST索引(gist\_ltree\_ops opclass)： <、<=、=、 >=、 >、 @>、<@、@、~、?

gist\_ltree\_ops GiST opclass将一组路径标签近似计算为位图签名。 它的可选整数参数siglen决定了签名的字节长度。默认签名长度为8字节。 签名长度的有效值在1到2024字节之间。更长的签名将导致更精确的搜索(扫描更小的索引部分和更少的堆页)，但代价是更大的索引。

创建默认签名长度为8字节的索引的例子：

```
CREATE INDEX path_gist_idx ON test USING GIST (path);
```

创建签名长度为100字节的索引的例子：

```
CREATE INDEX path_gist_idx ON test USING GIST (path
gist_ltree_ops(siglen=100));
```

- ltree[] 之上的GiST索引(gist\_\_ltree\_ops opclass): ltree[] <@ ltree、ltree @> ltree[], @、~、?

gist\_\_ltree\_ops GiST opclass 的工作类似于gist\_ltree\_ops 并且也使用签名长度作为参数。 gist\_\_ltree\_ops中的siglen的默认值为28字节。

创建这样一个默认签名长度为28字节的索引的例子:

```
CREATE INDEX path_gist_idx ON test USING GIST (array_path);
```

创建这样一个签名长度为100字节的索引的例子:

```
CREATE INDEX path_gist_idx ON test USING GIST (array_path
gist__ltree_ops(siglen=100));
```

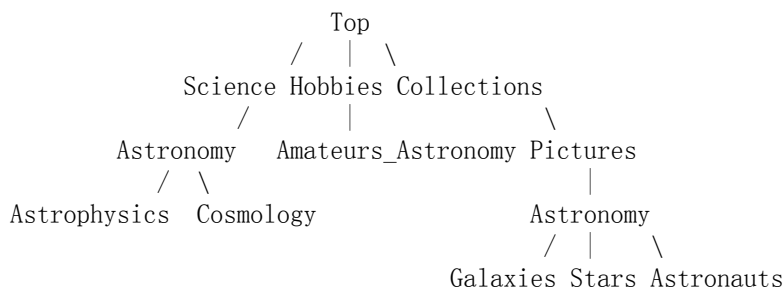
注意: 这种索引类型是有损的。

## H. 49. 4. 例子

这个例子使用下列数据 (在源代码发布的contrib/ltree/ltreetest.sql文件中也有):

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING GIST (path);
CREATE INDEX path_idx ON test USING BTREE (path);
```

现在, 我们有一个表test, 它被填充了描述下列层次的数据:



我们可以做继承:

```
lftreetest=> SELECT path FROM test WHERE path <@ 'Top. Science' ;
                path
```

```
-----
Top. Science
Top. Science. Astronomy
Top. Science. Astronomy. Astrophysics
Top. Science. Astronomy. Cosmology
(4 rows)
```

这里是一些路径匹配的例子:

```
lftreetest=> SELECT path FROM test WHERE path ~ '*. Astronomy.*' ;
                path
```

```
-----
Top. Science. Astronomy
Top. Science. Astronomy. Astrophysics
Top. Science. Astronomy. Cosmology
Top. Collections. Pictures. Astronomy
Top. Collections. Pictures. Astronomy. Stars
Top. Collections. Pictures. Astronomy. Galaxies
Top. Collections. Pictures. Astronomy. Astronauts
(7 rows)
```

```
lftreetest=> SELECT path FROM test WHERE path ~ '*. !pictures@. Astronomy.*' ;
                path
```

```
-----
Top. Science. Astronomy
Top. Science. Astronomy. Astrophysics
Top. Science. Astronomy. Cosmology
(3 rows)
```

这里是一些全文搜索的例子:

```
lftreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !pictures@' ;
                path
```

```
-----
Top. Science. Astronomy
Top. Science. Astronomy. Astrophysics
Top. Science. Astronomy. Cosmology
Top. Hobbies. Amateurs_Astronomy
(4 rows)
```

```
lftreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@' ;
                path
```

```
-----
Top. Science. Astronomy
Top. Science. Astronomy. Astrophysics
Top. Science. Astronomy. Cosmology
(3 rows)
```

使用函数的路径构建:

```
lftreetest=> SELECT subpath(path, 0, 2) || 'Space' || subpath(path, 2) FROM test WHERE
                path <@ 'Top. Science. Astronomy' ;
                ?column?
```

```
Top. Science. Space. Astronomy
Top. Science. Space. Astronomy. Astrophysics
Top. Science. Space. Astronomy. Cosmology
(3 rows)
```

我们可以通过常见一个在路径中指定位置插入标签的 SQL 函数来简化：

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
AS 'select subpath($1, 0, $2) || $3 || subpath($1, $2);'
LANGUAGE SQL IMMUTABLE;
```

```
ltreetest=> SELECT ins_label(path, 2, 'Space') FROM test WHERE path <@
'Top. Science. Astronomy';
           ins_label
```

```
-----
Top. Science. Space. Astronomy
Top. Science. Space. Astronomy. Astrophysics
Top. Science. Space. Astronomy. Cosmology
(3 rows)
```

## H. 49. 5. 作者

所有工作都是 Teodor Sigaev (<teodor@stack.net>) 和 Oleg Bartunov (<oleg@sai.msu.su>) 完成的。额外信息可见 <http://www.sai.msu.su/~megera/postgres/gist/>。作者还要感谢 Eugeny Rodichev 参与讨论。欢迎评论和缺陷报告。

## H. 50. ltrrowlocks

ltrrowlocks 模块提供了一个函数，用于显示指定表的行锁信息。

默认情况下，使用仅限于超级用户、pg\_stat\_scan\_tables角色的成员以及具有表上SELECT权限的用户。

### H. 50. 1. 概述

pgrowlocks(text) 返回记录集

参数是表的名称。结果是记录集，每个表中锁定的行对应一行记录。输出列在 Table H. 66中显示。

Table H. 66. pgrowlocks 输出列

名称	类型	描述
locked_row	tid	被锁定行的元组 ID (TID)
locker	xid	锁定者的事务 ID，如果是多事务，则为多事务 ID
multi	boolean	如果锁定者是多事务，则为 True
xids	xid[]	锁定者的事务 ID (如果是多事务，则可能有多个)
modes	text[]	锁定者的锁定模式 (如果是多事务，则可能有多个)，

名称	类型	描述
		一个由 Key Share, Share, For No Key Update, No Key Update, For Update, Update 组成的数组。
pids	integer[]	锁定后端的进程 ID (如果是多事务, 则可能有多个)

`pgrowlocks` 获取目标表的 `AccessShareLock`, 逐行读取以收集行锁定信息。这对于大表来说不是非常快速的。注意:

1. 如果在表上获取了 `ACCESS EXCLUSIVE` 锁定, `pgrowlocks` 将被阻塞。
2. `pgrowlocks` 不能保证生成一个自治的快照。在其执行期间可能会有新的行锁定被获取或旧的锁定被释放。

`pgrowlocks` 不显示被锁定行的内容。如果您想同时查看行内容, 可以尝试像这样的操作:

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
WHERE p.locked_row = a.ctid;
```

但是请注意, 这样的查询非常低效。

## H. 50. 2. 示例输出

```
=# SELECT * FROM pgrowlocks('t1');
locked_row | locker | multi | xids | modes | pids
-----+-----+-----+-----+-----+-----
(0, 1) | 609 | f | {609} | {"For Share"} | {3161}
(0, 2) | 609 | f | {609} | {"For Share"} | {3161}
(0, 3) | 607 | f | {607} | {"For Update"} | {3107}
(0, 4) | 607 | f | {607} | {"For Update"} | {3107}
(4 rows)
```

## H. 50. 3. 作者

Tatsuo Ishii

## H. 51. ltstattuple

`ltstattuple` 模块提供了各种函数来获取元组级别的统计信息。

由于这些函数返回详细的页面级别信息, 因此默认情况下访问受到限制。默认情况下, 只有角色 `pg_stat_scan_tables` 具有 `EXECUTE` 权限。当然, 超级用户可以绕过这个限制。安装扩展后, 用户可以发出 `GRANT` 命令来更改函数的权限, 以允许其他用户执行它们。然而, 将这些用户添加到 `pg_stat_scan_tables` 角色可能更可取。

### H. 51. 1. 函数

`pgstattuple(regclass)` 返回 `record`

`pgstattuple` 返回一个关系的物理长度、“dead”元组的百分比和其他信息。这可能有助于用户确定是否需要进行 `vacuum` 操作。参数是目标关系的名称 (可选地包含模式限定符) 或 `OID`。例如:

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]-----+-----
table_len      | 458752
tuple_count    | 1470
tuple_len      | 438896
tuple_percent  | 95.67
dead_tuple_count | 11
dead_tuple_len | 3157
dead_tuple_percent | 0.69
free_space     | 8932
free_percent   | 1.95
```

输出的列在 Table H. 67中描述。

Table H. 67. pgstattuple 输出列

列	类型	描述
table_len	bigint	关系的物理长度（以字节为单位）
tuple_count	bigint	存活元组的数量
tuple_len	bigint	存活元组的总长度（以字节为单位）
tuple_percent	float8	存活元组的百分比
dead_tuple_count	bigint	死亡元组的数量
dead_tuple_len	bigint	死亡元组的总长度（以字节为单位）
dead_tuple_percent	float8	死亡元组的百分比
free_space	bigint	总空闲空间（以字节为单位）
free_percent	float8	空闲空间的百分比

Note

table\_len 始终大于 tuple\_len、dead\_tuple\_len 和 free\_space 之和。  
差额由固定的页开销、指向元组的每页指针表和填充所占据。

pgstattuple 只对关系获取读取锁。因此，结果不反映即时快照；并发更新将影响它们。

pgstattuple 判断一个元组是否为“dead”，取决于 HeapTupleSatisfiesDirty 返回的结果是否为 false。

pgstattuple(text) 返回 record

这与 pgstattuple(regclass) 相同，只是目标关系以 TEXT 的形式指定。此函数因向后兼容而保留，但将在某个未来版本中弃用。

pgstatindex(regclass) 返回 record

pgstatindex 返回一个记录，其中包含有关 B-tree 索引的信息。例如：

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
```

```

-[ RECORD 1 ]-----+-----
version          | 2
tree_level      | 0
index_size      | 16384
root_block_no   | 1
internal_pages  | 0
leaf_pages      | 1
empty_pages     | 0
deleted_pages   | 0
avg_leaf_density | 54.27
leaf_fragmentation | 0
    
```

输出列为:

列	类型	描述
version	integer	B-tree 版本号
tree_level	integer	根页的树级别
index_size	bigint	索引的总大小（以字节为单位）
root_block_no	bigint	根页的位置（如果没有，则 为零）
internal_pages	bigint	“内部”（上层）页的数量
leaf_pages	bigint	叶子页的数量
empty_pages	bigint	空页的数量
deleted_pages	bigint	已删除页的数量
avg_leaf_density	float8	叶子页的平均密度
leaf_fragmentation	float8	叶子页的碎片化程度

报告的 `index_size` 通常比 `internal_pages + leaf_pages + empty_pages + deleted_pages` 所占的页数多一个，因为它还包括索引的元数据页。

与 `pgstattuple` 一样，结果是逐页累加的，不应该期望它们代表整个索引的即时快照。

`pgstatindex(text)` 返回 record

这与 `pgstatindex(regclass)` 相同，只是目标索引以 `TEXT` 的形式指定。此函数因向后兼容而保留，但将在某个未来版本中弃用。

`pgstatginindex(regclass)` 返回 record

`pgstatginindex` 返回一个记录，其中包含有关 GIN 索引的信息。例如：

```

test=> SELECT * FROM pgstatginindex('test_gin_index');
-[ RECORD 1 ]--+-+
version          | 1
pending_pages    | 0
pending_tuples   | 0
    
```

输出列包括:

列名	类型	描述
version	integer	GIN 索引的版本号
pending_pages	integer	待处理列表中的页数

列名	类型	描述
pending_tuples	bigint	待处理列表中的元组数

pgstathashindex(regclass) 返回 record

pgstathashindex 返回一个记录，其中包含有关 HASH 索引的信息。例如：

```
test=> select * from pgstathashindex('con_hash_index');
-[ RECORD 1 ]-----+-----
version          | 4
bucket_pages     | 33081
overflow_pages   | 0
bitmap_pages     | 1
unused_pages     | 32455
live_items       | 10204006
dead_items       | 0
free_percent     | 61.8005949100872
```

输出列包括：

列名	类型	描述
version	integer	HASH 索引的版本号
bucket_pages	bigint	桶页数
overflow_pages	bigint	溢出页数
bitmap_pages	bigint	位图页数
unused_pages	bigint	未使用页数
live_items	bigint	存活元组数
dead_tuples	bigint	死亡元组数
free_percent	float	可用空间的百分比

pg\_relpages(regclass) 返回 bigint

pg\_relpages 返回关系中的页数。

pg\_relpages(text) 返回 bigint

这与 pg\_relpages(regclass) 相同，只是将目标关系指定为 TEXT。此功能保留了向后兼容性，但将在将来的某个版本中被废弃。

pgstattuple\_approx(regclass) 返回 record

pgstattuple\_approx 是 pgstattuple 的一种更快的替代方法，返回近似结果。参数是目标关系的名称或 OID。例如：

```
test=> SELECT * FROM pgstattuple_approx('pg_catalog.pg_proc'::regclass);
-[ RECORD 1 ]-----+-----
table_len          | 573440
scanned_percent    | 2
approx_tuple_count | 2740
approx_tuple_len   | 561210
approx_tuple_percent | 97.87
dead_tuple_count    | 0
dead_tuple_len     | 0
dead_tuple_percent  | 0
```



```
approx_free_space | 11996
approx_free_percent | 2.09
```

输出列的描述在 Table H. 68中。

`pgstattuple` 总是执行全表扫描并返回活动和死亡元组（及其大小）以及可用空间的精确计数，而 `pgstattuple_approx` 尝试避免全表扫描并返回精确的死亡元组统计信息以及近似的存活元组数和大小和可用空间。

它通过跳过仅包含可见元组的页面（根据可见性映射）来实现此目的（如果页面设置了相应的 VM 位，则假定不包含死亡元组）。对于这些页面，它从空闲空间映射中获取空闲空间值，并假设页面上的其余空间被存活元组占据。

对于无法跳过的页面，它会扫描每个元组，记录其存在和大小于相应的计数器中，并累加页面上的可用空间。最后，它基于扫描的页面和元组数量估计存活元组的总数（与 VACUUM 估计 `pg_class.reltuples` 的方式相同）。

Table H. 68. `pgstattuple_approx` 输出列

列名	类型	描述
<code>table_len</code>	<code>bigint</code>	物理关系的长度（以字节为单位）（精确值）
<code>scanned_percent</code>	<code>float8</code>	表扫描的百分比
<code>approx_tuple_count</code>	<code>bigint</code>	存活元组数量（估算值）
<code>approx_tuple_len</code>	<code>bigint</code>	存活元组的总字节长度（估算值）
<code>approx_tuple_percent</code>	<code>float8</code>	存活元组的百分比
<code>dead_tuple_count</code>	<code>bigint</code>	死亡元组的数量（确切值）
<code>dead_tuple_len</code>	<code>bigint</code>	死亡元组的总字节长度（确切值）
<code>dead_tuple_percent</code>	<code>float8</code>	死亡元组的百分比
<code>approx_free_space</code>	<code>bigint</code>	总空闲空间的字节大小（估算值）
<code>approx_free_percent</code>	<code>float8</code>	空闲空间的百分比

在上面的输出中，空闲空间的数据可能无法完全匹配`pgstattuple`的输出结果，因为空闲空间映射提供了精确的数字，但不能保证以字节为单位的精度完全准确。

## H. 51. 2. 作者

Tatsuo Ishii, Satoshi Nagayasu and Abhijit Menon-Sen

## H. 52. `lt_walminer`

### H. 52. 1. 什么是`lt_walminer`?

请参考`lt_walminer`参考手册<sup>10</sup>。

## H. 53. `myfce`

`myfce` - Mysql的兼容性函数。

<sup>10</sup> [https://gitee.com/zhjh256/lt\\_walminer/blob/master/README.md](https://gitee.com/zhjh256/lt_walminer/blob/master/README.md)

## H. 53. 1. 与Mysql数据库兼容的功能

提供了与Mysql数据库兼容的功能。这些功能使您能够轻松迁移到LightDB，并减少重新配置应用程序的成本。下表列出了与Mysql数据库兼容的功能。

Table H. 69. 数据类型

项目	概览
DATETIME	DATETIME 是 timestamp without zone 的别名。
LONGTEXT	LONGTEXT是TEXT的别名。
SIGNED	SIGNED是一个带符号的整数，相当于 bigint。

Table H. 70. 数据转换

项目	概览
bool->varchar	兼容mysql，bool类型true转换至字符串'1'，false至字符串'0'
cast(expr as unsigned)	转换expr为unsigned bigint值，但是当前返回类型为numeric。

Table H. 71. 数学函数

项目	概览
CONV	在不同的数字基数之间转换数字
LOG10	返回参数的以10为底的对数
LOG2	返回参数的以2为底的对数
RAND	返回介于0和1.0之间的随机浮点值v
TRUNCATE	截断到指定的小数位数

Table H. 72. 字符串函数

项目	概览
ELT	返回索引号处的字符串
FIELD	第一个参数在后续参数中的索引（位置）
FIND_IN_SET	第一个参数在第二个参数中的索引（位置）
FROM_BASE64	解码base64编码的字符串并返回结果
INSERT	在指定位置插入子字符串，插入的长度为指定的字符数
INSTR	获取子字符串的第一次出现位置
LCASE	LOWER()的同义词
LOCATE	返回子字符串的第一次出现位置
MID	从指定位置开始返回一个子字符串
SPACE	返回指定数量空格的字符串
STRCMP	比较两个字符串
TO_BASE64	将参数转换为base-64字符串
UCASE	UPPER()的同义词

项目	概览
SUBSTR	返回指定的子字符串
SUBSTRING	返回指定的子字符串
SUBSTRING_INDEX	从指定字符串中返回指定分隔符出现n次数前的子字符串

Table H. 73. 日期/时间函数

项目	概览
DATEDIFF	计算两个日期之间的天数差（用第一个日期减去第二个日期）
DATE_ADD	将指定的时间间隔添加到日期中
DATE_FORMAT	以不同格式返回日期/时间数据
DATE_SUB	从日期中减去指定的时间间隔
DAY	返回日期月份的日期部分
DAYOFWEEK	返回当前星期的星期几索引（从1开始）
FROM_UNIXTIME	返回UNIX时间戳的日期/时间表示
SEC_TO_TIME	将秒转换为' hh:mm:ss' 格式
STR_TO_DATE	将字符串转换为日期/时间值
SYSDATE	返回此函数执行时的时间
TIME_TO_SEC	将参数转换为秒
TIMEDIFF	计算两个时间之间的时间差（用第一个时间减去第二个时间）
TIMESTAMPDIFF	从日期时间表达式中减去一个时间间隔
TO_DAYS	将日期参数转换为天数
TO_SECONDS	将日期或日期时间参数转换为公元元年以来的秒数
UNIX_TIMESTAMP	返回精确时间点的UNIX时间戳
WEEKOFYEAR	返回日期所在的周数
YEAR	返回日期的年份
QUARTER	返回日期的季度
LAST_DAY	返回参数所在月份的最后一天的日期值
CURDATE	返回当前日期

Table H. 74. JSON 函数

项目	概览
JSON_ARRAY	计算列表中的每个值（列表可能为空），返回包含这些值的 JSON 数组。
JSON_OBJECT	计算列表中的每个键值对（列表可能为空），返回包含这些键值对的 JSON 数组。
JSON_EXTRACT	通过 JSON 路径从 JSON 文档中选择匹配的数据，并返回。
JSON_CONTAINS	判断 JSON 文档是否包含指定的 JSON 路径，返回 true 或 false。
JSON_CONTAINS_PATH	判断 JSON 文档是否包含指定的一个或多个 JSON 路径，返回 true 或 false。

项目	概览
JSON_PRETTY	格式化 JSON 文档。
JSON_REMOVE	从 JSON 文档中删除数据，返回修改后的结果。
JSON_INSERT	向 JSON 文档中插入数据，返回修改后的结果。
JSON_REPLACE	替换 JSON 文档的数据，返回修改后的结果。
JSON_SET	向 JSON 文档中插入数据或替换 JSON 文档的数据，返回修改后的结果。

Table H. 75. 条件表达式

Item	概述
IF	if/else结构

Table H. 76. 聚合函数

Item	概述
ANY_VALUE	兼容MySQL模式，非ONLY_FULL_GROUP_BY语法下在BY模式中的group by行为
GROUP_CONCAT	返回连接起来的、用分隔符分隔的字符串值列表。

Table H. 77. 系统信息函数和运算符

Item	概述
DATABASE	current_schema()的同义词

Table H. 78. 杂项函数

项目	概述
BIN_TO_UUID	将二进制UUID转换为字符串
COMPRESS	压缩字符串并返回二进制数据
UNCOMPRESS	解压缩由compress()压缩的二进制数据，并还原原始字符串
UUID	生成一个36字节的uuid字符串
UUID_TO_BIN	将字符串UUID转换为二进制

Table H. 79. 运算符

项目	概述
IFNULL	根据参数是否为空返回相应的值
ISNULL	测试参数是否为NULL

Table H. 80. 算术运算符

项目	概述
/	除法运算符

Table H.81. 比较操作符

项目	概述
=	等号操作符，支持varchar与bool、text类型比较
!=	不等号操作符，支持varchar与bool、text类型比较

## H. 53. 2. SQL函数参考

### H. 53. 2. 1. 数学函数

支持以下数学函数：

- CONV
- LOG10
- LOG2
- RAND
- TRUNCATE

#### H. 53. 2. 1. 1. CONV

描述

在不同的进制之间转换数字。

语法

```
conv( p_num text, p_from_base int4, p_to_base int4) RETURNS text
```

常规规则

- 在不同的进制之间转换数字。
- 将数字N从进制from\_base转换为进制to\_base并返回字符串表示形式。如果任何参数为NULL，则返回NULL。参数N被解释为整数，但可以指定为整数或字符串。
- 最小进制为2，最大进制为36。如果from\_base是负数，则N被视为带符号数。否则，N被视为无符号数。CONV()使用64位精度。

示例

在以下示例中，'123456'从10进制转换为16进制是1E240。

```
select conv('123456', 10, 16);
conv
-----
1E240
(1 row)
```

#### H. 53. 2. 1. 2. LOG10

描述

返回参数的以10为底的对数。

语法

```
log10(p1 numeric) RETURNS numeric
log10(p1 double precision) RETURNS double precision
```

常规规则

- 返回p1的以10为底的对数。如果X小于或等于0.0E0，则该函数返回NULL并报告警告“对数的参数无效”。

示例

在以下示例中，返回参数2的以10为底的对数。

```
select log10(2);
       log10
-----
0.3010299956639812
(1 row)
```

### H. 53. 2. 1. 3. LOG2

描述

返回参数的以2为底的对数。

语法

```
log2(p_num numeric) RETURNS numeric
```

常规规则

- 返回p\_num的以2为底的对数。如果p\_num小于或等于0.0E0，则该函数返回NULL并报告警告“对数的参数无效”。

示例

在以下示例中，返回参数64的以2为底的对数。

```
select log2(64);
       log2
-----
6.0000000000000000
(1 row)
```

### H. 53. 2. 1. 4. RAND

描述

返回范围在0和1.0之间的随机浮点值v。

语法

rand(p\_seed int default null) RETURNS double precision

常规规则

- 使用常量初始化参数时，种子在语句准备好之后，在执行之前仅初始化一次。
- 使用非常量初始化参数（例如列名）时，种子将在每次调用RAND()时使用值进行初始化。
- 这种行为的一个影响是，对于相同的参数值，RAND(N)每次返回相同的值，从而产生一系列可重复的列值。
- 在WHERE子句中使用RAND()会对每一行（从单个表中选择时）或每一行组合（从多张表连接时）进行求值。

示例

在以下示例中，RAND(123)产生的值序列在出现的两个位置都相同。

```
select rand(123);
       rand
-----
0.05425240159967615
(1 row)
```

#### H. 53. 2. 1. 5. TRUNCATE

描述

截取指定小数位数。

语法

```
TRUNCATE (SMALLINT X, INTEGER D) returns INTEGER
TRUNCATE (INTEGER X, INTEGER D) returns INTEGER
TRUNCATE (BIGINT X, INTEGER D) returns BIGINT
TRUNCATE (DOUBLE PRECISION X, INTEGER D) returns DOUBLE PRECISION
TRUNCATE (NUMERIC, INTEGER D) returns NUMERIC
```

通用规则

- 返回截取到小数点后D位的数字X。
- 如果D为0，则结果没有小数点或小数部分。
- D可以是负数，使值X的小数点左侧的D位数字变为零。

示例

在以下示例中，返回将“1.223”截取到小数点后1位的结果。

```
SELECT TRUNCATE(1.223, 1);
       truncate
-----
          1.2
(1 row)
```

## H. 53. 2. 2. 字符串函数

以下字符串函数得到支持:

- ELT
- FIELD
- FIND\_IN\_SET
- INSTR
- FROM\_BASE64
- INSERT
- LCASE
- LOCATE
- MID
- SPACE
- STRCMP
- TO\_BASE64
- UCASE
- SUBSTR
- SUBSTRING
- SUBSTRING\_INDEX

### H. 53. 2. 2. 1. ELT

描述

返回索引号处的字符串。

语法

```
elt(str_pos int, VARIADIC strlist text[]) RETURNS text
```

通用规则

- ELT() 返回字符串列表中的第str\_pos个元素: 如果str\_pos = 1, 则为str1, 如果str\_pos = 2, 则为str2, 以此类推。
- 如果N小于1或大于参数个数, 则返回NULL。
- ELT() 是 FIELD() 的补充函数。

示例

在以下例子中, 返回第 2 个元素 'Bb'。



```
select elt( 2, 'Aa', 'Bb', 'Cc', 'Dd', 'Ff');
elt
-----
Bb
(1 row)
```

#### H. 53. 2. 2. 2. FIELD

##### 描述

第一个参数在后续参数中的位置（索引）。

##### 语法

```
field(str text, VARIADIC strlist text[]) RETURNS bigint
field(str numeric, VARIADIC strlist numeric[]) RETURNS bigint
```

##### 一般规则

- 返回字符串 `str` 在字符串列表 `str1, str2, str3, ...` 中的索引（位置）。如果 `str` 未找到，则返回 0。
- 如果 `FIELD()` 的所有参数都是字符串，则将它们作为字符串进行比较。如果所有参数都是数字，则将它们作为数字进行比较。否则，将参数作为双精度数进行比较。
- 如果 `str` 是 `NULL`，则返回值为 0，因为 `NULL` 与任何值都不相等。`FIELD()` 是 `ELT()` 的补充函数。

##### 示例

在以下例子中，返回字符串 `'Bb'` 在字符串列表 `'Aa', 'Bb', 'Cc', 'Dd', 'Ff'` 中的索引（位置） 2。

```
select field('Bb', 'Aa', 'Bb', 'Cc', 'Dd', 'Ff');
field
-----
      2
(1 row)
```

#### H. 53. 2. 2. 3. FIND\_IN\_SET

##### 描述

第一个参数在第二个参数中的位置（索引）。

##### 语法

```
FIND_IN_SET(str "any", strlist TEXT) returns INTEGER
```

##### 一般规则

- 如果字符串 `str` 在由 `N` 个子字符串组成的字符串列表 `strlist` 中，则返回 1 到 `N` 的值。 `strlist` 是由逗号字符分隔的子字符串组成的字符串。
- 如果 `str` 不在 `strlist` 中或 `strlist` 是空字符串，则返回 0。

- 如果任一参数为 NULL，则返回 NULL。如果第一个参数包含逗号 (,) 字符，则此函数不正常工作。
- 如果第一个参数不是字符串类型，则第一个参数将被转换为字符串类型。换句话说，`find_in_set(n, strlist)` 等效于 `find_in_set(n::text, strlist)`
- 返回值的数据类型为整数。

示例

在以下例子中，返回字符串 'a, b, c, d' 中字符 'b' 的位置（索引）。

```
SELECT FIND_IN_SET('b', 'a, b, c, d');
       find_in_set
-----
                2
(1 row)
```

#### H. 53. 2. 2. 4. FROM\_BASE64

描述

解码 base64 编码的字符串并返回结果。

语法

`from_base64(p_str text) RETURNS text`

一般规则

- 接受一个使用 `TO_BASE64()` 使用的 base-64 编码规则编码的字符串，并将解码后的结果作为二进制字符串返回。
- 如果参数为 NULL 或不是有效的 base-64 字符串，则结果为 NULL。
- 有关编码和解码规则的详细信息，请参见 `TO_BASE64()` 的描述。

示例

在以下例子中，解码 base64 编码的字符串 'YWJj' 并返回结果 'abc'。

```
select from_base64('YWJj');
       from_base64
-----
       abc
(1 row)
```

#### H. 53. 2. 2. 5. 插入

描述

在指定位置插入子字符串，直到指定的字符数。

语法

```
insert(p_source text, p_pos bigint, p_len bigint, p_replacement text)
RETURNS text
```

#### 通用规则

- 返回字符串str，将从位置pos开始并长度为len的子字符串替换为newstr。
- 如果pos不在字符串的长度范围内，则返回原始字符串。如果len不在其余字符串的长度范围内，则从位置pos替换其余字符串。
- 如果任何参数为NULL，则返回NULL。

#### 示例

在下面的示例中，在指定位置3插入子字符串'What'，直到指定的字符数4。

```
select insert('Quadratic', 3, 4, 'What');
insert
-----
QuWhattic
(1 row)
```

### H. 53. 2. 2. 6. INSTR

#### 描述

获取子字符串的第一次出现的位置。

#### 语法

```
instr(str text, patt text) RETURNS int
```

#### 通用规则

- 从1开始，在字符串str中获取子字符串patt的第一次出现的位置。
- 如果未找到，则返回0。

#### 示例

在下面的示例中，在字符串'abc124deff4de'中，子字符串'4de'的第一次出现的位置是6。

```
select instr('abc124deff4de', '4de');
instr
-----
6
(1 row)
```

### H. 53. 2. 2. 7. LCASE

#### 描述

LOWER()的同义词。

#### 语法

lcase(p\_source text) RETURNS text

通用规则

- LCASE()是LOWER()的同义词。
- 根据当前字符集映射，将字符串str中的所有字符更改为小写。默认为utf8mb4。

示例

在下面的示例中，返回字符串'quadratic'，其中所有字符都已更改为小写。

```
select lcase('QuadRatic');
   lcase
-----
quadratic
(1 row)
```

#### H. 53. 2. 2. 8. LOCATE

描述

返回子字符串第一次出现的位置。

语法

```
LOCATE(substr TEXT, str TEXT) returns INTEGER
LOCATE(substr TEXT, str TEXT, POS INTEGER) returns INTEGER
```

通用规则

- 第一种语法返回字符串str中子字符串substr的第一次出现的位置。
- 第二种语法返回字符串str中子字符串substr在位置pos开始的第一次出现的位置。
- 如果在str中没有找到substr，则返回0。
- 如果任何参数为NULL，则返回NULL。

示例

在下面的示例中，返回'foobarbar'中'bar'在位置5开始的出现位置。

```
SELECT LOCATE('bar', 'foobarbar', 5);
   locate
-----
       7
(1 row)
```

#### H. 53. 2. 2. 9. MID

描述

从指定位置开始返回一个子字符串。

语法

```
mid(p_source text, p_pos bigint, p_len bigint) RETURNS text
```

通用规则

- 不带 len 参数的形式从字符串 str 的位置 pos 开始返回一个子字符串。带有 len 参数的形式从字符串 str 的位置 pos 开始返回一个长度为 len 的子字符串。
- pos 也可以使用负值。在这种情况下，子字符串的开头是距离字符串末尾 pos 个字符，而不是开头。负值可以用于此函数的任何形式中的 pos。对于 pos 的值为 0，返回一个空字符串。

示例

在下面的示例中，从指定位置 2 开始返回子字符串 'ongy'。

```
select mid('Hongye', 2, 4);
mid
-----
ongy
(1 row)
```

#### H. 53. 2. 2. 10. SPACE

描述

返回指定数量的空格字符串。

语法

```
space(p_num int) RETURNS text
```

通用规则

- 返回由 N 个空格字符组成的字符串。

示例

在下面的示例中，返回指定数量的空格字符串。

```
select concat(' |', space(11), ' |');
concat
-----
|      |
(1 row)
```

#### H. 53. 2. 2. 11. STRCMP

描述

比较两个字符串。

语法

strcmp(p1 text, p2 text) RETURNS int

通用规则

- STRCMP() 返回 0 表示两个字符串相同，返回 -1 表示第一个参数小于第二个参数按照当前的排序顺序，返回 1 表示其他情况。

示例

在下面的示例中，如果第一个参数按照当前的排序顺序小于第二个参数，则返回 -1。

```
select strcmp('text', 'text2');
strcmp
-----
      -1
(1 row)
```

#### H. 53. 2. 2. 12. TO\_BASE64

描述

将参数转换为 base-64 字符串并返回。

语法

to\_base64(p\_str text) RETURNS text

通用规则

- 将字符串参数转换为 base-64 编码形式，并返回一个使用连接字符集和排序规则的字符串作为结果。
- 如果参数不是字符串，则在进行转换之前将其转换为字符串。如果参数为 NULL，则结果为 NULL。
- 使用 FROM\_BASE64() 函数可以解码 base-64 编码字符串。

示例

在下面的示例中，将参数 'abc' 转换为 base-64 字符串 'YWJj' 并返回。

```
select to_base64('abc');
to_base64
-----
YWJj
(1 row)
```

#### H. 53. 2. 2. 13. UCASE

描述

UPPER() 的同义词。

语法

`ucase(p_source text) RETURNS text`

通用规则

- 将字符串 `str` 中的所有字符根据当前字符集映射更改为大写。默认字符集为 `utf8mb4`。

示例

在下面的示例中，将字符串 `'QUADRATIC'` 中的所有字符根据当前字符集映射更改为大写并返回。

```
select ucase('QuadRatic');
       ucase
-----
QUADRATIC
(1 row)
```

#### H. 53. 2. 2. 14. SUBSTR

描述

返回按指定要求截取的子字符串。

语法

`substr(str text, pos int) RETURNS text`  
`substr(str text, pos int, len int) RETURNS text`

通用规则

- 不带 `len` 参数的形式返回从字符串 `str` 中位置 `pos` 开始的子字符串。带有 `len` 参数的形式返回从字符串 `str` 中位置 `pos` 开始长度为 `len` 的子字符串。
- `pos` 参数也可以使用负数。在这种情况下，子字符串的开始位置是距离字符串结尾 `pos` 个字符，而不是开头。`pos` 参数可以在此函数的任何形式中使用负值。当 `pos` 参数为 0 时，返回空字符串。
- 此函数支持多字节字符集。如果任何参数为 `NULL`，则返回 `NULL`。
- 如果 `len` 小于 1，则结果为空字符串。
- 如果 `pos` 和 `len` 都接受数字类型，则也是可以的。如果输入的小数部分小于 0.5，则将其四舍五入为整数部分。否则，将正数的四舍五入为输入值加上 0.5 的下限，将负数的四舍五入为输入值减去 0.5 的下限。

示例

在下面的示例中，结果为 `bc`。

```
select substr('abc', -2, 2);
       substr
-----
bc
(1 row)
```

### H. 53. 2. 2. 15. SUBSTRING

#### 描述

根据指定要求返回子字符串。

#### 语法

```
substring(str text, pos int) RETURNS text
substring(str text, pos int, len int) RETURNS text
```

#### 一般规则

- `substring` 是 `substr` 的同义词。

#### 示例

在以下的示例中，结果是 `bc`。

```
select substring(' abc', -2, 2);
  substring
-----
      bc
(1 row)
```

### H. 53. 2. 2. 16. SUBSTRING\_INDEX

#### 描述

从指定字符串中返回指定分隔符出现n次数前的子字符串

#### 语法

```
substring_index(str varchar, delim varchar, count integer) RETURNS varchar
```

#### 一般规则

- 从指定字符串中返回到达分隔符`delim`出现次数之前的子字符串。
- 如果`count`为正，则返回最后一个分隔符左侧的所有元素(从左侧开始计数)。
- 如果`count`为负数，则返回最后一个分隔符右侧的所有内容(从右侧开始计数)。
- `SUBSTRING_INDEX()` 在搜索`delim`时区分大小写。
- 这个函数是多字节安全的。
- 如果`SUBSTRING_INDEX()`的任何参数为`NULL`，则返回`NULL`。
- 当分隔符为 `'` 时，函数返回 `'` (和mysql一致)。

#### 示例

```
lightdb@test_m=# SELECT SUBSTRING_INDEX(' www.mysql.com', '.', 2);
  substring_index
-----
```



```
www.mysql  
(1 row)
```

```
lightdb@test_m=# SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);  
substring_index
```

```
-----  
mysql.com  
(1 row)
```

### H. 53. 2. 3. 日期/时间函数

支持以下日期/时间函数：

- DATEDIFF
- DATE\_ADD
- DATE\_FORMAT
- DATE\_SUB
- DAY
- DAYOFWEEK
- FROM\_UNIXTIME
- STR\_TO\_DATE
- SYSDATE
- SEC\_TO\_TIME
- TIME\_TO\_SEC
- TIMEDIFF
- TIMESTAMPDIFF
- TO\_DAYS
- TO\_SECONDS
- UNIX\_TIMESTAMP
- WEEKOFYEAR
- YEAR
- QUARTER
- LAST\_DAY
- CURDATE

#### H. 53. 2. 3. 1. DATEDIFF

描述

减去两个日期。

语法

```
DATEDIFF(expr1 timestamp, expr2 timestamp) RETURN integer
DATEDIFF(expr1 text, expr2 text) RETURN integer
DATEDIFF(expr1 timestampz, expr2 timestampz) RETURN integer
DATEDIFF(expr1 timestampz, expr2 timestamp) RETURN integer
DATEDIFF(expr1 timestamp, expr2 timestampz) RETURN integer
DATEDIFF(expr1 date, expr2 date) RETURN integer
DATEDIFF(expr1 date, expr2 numeric) RETURN integer
DATEDIFF(expr1 numeric, expr2 date) RETURN integer
DATEDIFF(expr1 numeric, expr2 numeric) RETURN integer
```

#### 一般规则

- `expr1 - expr2` 表示从一天到另一天的天数。`expr1` 和 `expr2` 是日期或日期时间表达式。计算中仅使用这些值的日期部分。
- 如果任一参数为 `NULL`，则返回 `NULL`。
- 不支持 `time` 类型相减

#### 示例

在以下示例中，返回 1，因为 '2007-12-31' 减去 '2007-12-30' 等于 1

```
SELECT DATEDIFF('2007-12-31 23:59:59', '2007-12-30');
datediff
-----
1
(1 row)
SELECT DATEDIFF(cast('2007-12-31 23:59:59' as datetime), cast('2007-12-30' as
  datetime));
datediff
-----
1
(1 row)
SELECT DATEDIFF(cast('2007-12-31 23:59:59' as date), cast('2007-12-30' as date));
datediff
-----
1
(1 row)
```

### H. 53. 2. 3. 2. DATE\_ADD

#### 描述

日期加法。

#### 语法

```
DATE_ADD(expr1 timestampz, expr2 interval) RETURN date
DATE_ADD(expr1 text, expr2 interval) RETURN date
```

#### 一般规则

- 返回添加间隔（年/月/日）到 `expr1` 日期参数后的日期类型。

- 第二个参数 `expr2` 仅支持间隔（年/月/日）。

示例

在下面的例子中，返回'2020-01-02'，因为'20200101'加上1天。

```
select date_add('20200101', interval 1 day);
       date_add
-----
2020-01-02
(1 row)
```

### H. 53. 2. 3. 3. DATE\_FORMAT

描述

日期格式显示。

语法

```
DATE_FORMAT(expr1 timestamp, expr2 text) RETURN text
DATE_FORMAT(expr1 timestamptz, expr2 text) RETURN text
DATE_FORMAT(expr1 text, expr2 text) RETURN text
```

通用规则

- 返回`expr1`时间参数，在`expr2`格式中显示和输出。

例子

在下面的例子中，返回'2021/12/01 11, 59, 59'，因为`expr2`是'%Y/%m/%d %h,%i,%s'格式。

```
select date_format('20211201 11:59:59', '%Y/%m/%d %h,%i,%s');
       date_format
-----
2021/12/01 11, 59, 59
(1 row)
```

Table H. 82显示了可用于格式化`date_format`值的模板模式。

Table H. 82. `date_format`格式化的模板模式

Pattern	Description
%a	缩写的星期名称
%b	缩写的月份名称
%c	月份，值
%D	带有英文前缀的月份中的日
%d	月份中的日，值（00-31）
%e	月份中的日，值（0-31）
%f	微秒
%H	小时（00-23）

Pattern	Description
%h	小时 (01-12)
%I	小时 (01-12)
%i	分钟, 值 (00-59)
%j	年份中的日 (001-366)
%k	小时 (0-23)
%l	小时 (1-12)
%M	月份名称
%m	月份, 值 (00-12)
%p	上午或下午
%r	时间, 12小时制 (hh:mm:ss AM或PM)
%S	秒 (00-59)
%s	秒 (00-59)
%T	时间, 24小时制 (hh:mm:ss)
%W	星期名称
%Y	年份, 4位数
%y	年份, 2位数字

#### H. 53. 2. 3. 4. DATE\_SUB

描述

日期减法。

语法

```
DATE_SUB(expr1 timestampz, expr2 interval) RETURN date
DATE_SUB(expr1 text, expr2 interval) RETURN date
```

一般规则

- 返回 expr1 日期参数减去间隔 (年/月/日) 后的日期类型
- 第二个参数 expr2 只支持间隔 (年/月/日)

示例

在以下示例中, “20200101” 减去一天后返回 “2019-12-31”

```
select date_sub('20200101', interval 1 day);
       date_sub
-----
2019-12-31
(1 row)
```

#### H. 53. 2. 3. 5. DAY

描述

返回月份日期。

语法

```
DAY(expr1 text) RETURN int2
DAY(expr1 timestampz) RETURN int4
```

一般规则

- 返回给定 expr1 日期的月份部分
- 仅支持月和日参数，例如“0101”、“0704”。

示例

在以下示例中，“2011/12/01” 的日期是 “1”，因此返回 “1”。

```
select day('2011/12/01');
day
-----
  1
(1 row)
```

#### H. 53. 2. 3. 6. DAYOFWEEK

描述

返回星期几。

语法

```
DAYOFWEEK(expr1 text) RETURN int4
DAYOFWEEK(expr1 timestampz) RETURN int4
```

一般规则

- 根据 expr1 日期参数返回星期几。返回值为 1 ~ 7，星期日为 1。
- 至少包括完整的时间格式，例如 20200202、200202。不支持少于 6 位的参数。

示例

在以下示例中，“20200101” 是星期三，因此返回 “4”。

```
select dayofweek(20200101);
dayofweek
-----
          4
(1 row)
```

#### H. 53. 2. 3. 7. FROM\_UNIXTIME

描述

根据 UNIX 时间戳返回时间格式。

语法

```
FROM_UNIXTIME(expr1 text) RETURN timestamp
FROM_UNIXTIME(expr1 int8) RETURN timestamp
FROM_UNIXTIME(expr1 text, expr2 text) RETURN text
FROM_UNIXTIME(expr1 int8, expr2 text) RETURN text
```

一般规则

- 根据 expr1 参数的 UNIX 时间戳返回时间格式。日期格式可以根据 expr2 参数进行指定。

示例

在以下示例中，根据 UNIX 时间戳 20220910 返回时间戳。

```
select from_unixtime(20220910);
      from_unixtime
-----
1970-08-23 08:55:10
(1 row)
select from_unixtime(20220910, '%Y/%m/%d %h,%i,%s');
      from_unixtime
-----
1970/08/23 08,55,10
(1 row)
```

Table H.83显示了可用于格式化 from\_unixtime 值的模板模式。

Table H.83. from\_unixtime 格式化的模板模式

模式	描述
%a	缩写星期名称
%b	缩写月份名称
%c	月份，数字表示
%D	带英文前缀的月份天数
%d	月份天数，数字表示（00-31）
%e	月份天数，数字表示（0-31）
%f	微秒
%H	小时数（00-23）
%h	小时数（01-12）
%I	小时数（01-12）
%i	分钟数，数字表示（00-59）
%j	年份中的天数（001-366）
%k	小时数（0-23）
%l	小时数（1-12）
%M	月份名称
%m	月份，数字表示（00-12）
%p	上午或下午
%r	时间，12小时制（hh:mm:ss 上午或下午）

模式	描述
%S	秒数 (00-59)
%s	秒数 (00-59)
%T	时间, 24小时制 (hh:mm:ss)
%W	星期名称
%Y	年份, 4位数字
%y	年份, 2位数字

#### H. 53. 2. 3. 8. SEC\_TO\_TIME

##### 描述

将秒转换为“hh:mm:ss”格式。

##### 语法

```
sec_to_time(p_secs int) RETURNS interval
```

##### 通用规则

- 返回将秒参数转换为小时、分钟和秒的时间值。
- 结果的范围受到 TIME 数据类型的限制。如果参数对应于超出该范围的值，则会发出警告。

##### 示例

在下面的例子中，将秒数 80580 转换成 'hh:mm:ss' 格式的时间，即 22:23:00。

```
select mysql.sec_to_time(80580);
sec_to_time
-----
22:23:00
(1 row)
```

#### H. 53. 2. 3. 9. STR\_TO\_DATE

##### 描述

将字符串转换为时间值。这是 DATE\_FORMAT() 函数的反函数。它接受一个字符串 str 和一个格式字符串 format。如果格式字符串包含日期和时间部分，STR\_TO\_DATE() 返回一个 DATETIME 值。如果字符串仅包含日期或时间部分，则返回 DATE 或 TIME 值。如果从 str 中提取的日期、时间或日期时间值是非法的，STR\_TO\_DATE() 将返回错误。

##### 语法

```
STR_TO_DATE(datetime_str text, datetime_format text) RETURN datetime
STR_TO_DATE(date_str text, date_format text) RETURN date
STR_TO_DATE(time_str text, time_format text) RETURN time
```

##### 通用规则

- 将字符串参数 `str` 和格式参数 `format` 转换为时间值。
- 如果启用了 `NO_ZERO_DATE` SQL 模式，则不允许使用零日期。在这种情况下，`STR_TO_DATE()` 返回 `NULL`。

示例

在下面的例子中，根据时间戳 `'2022/09/10 11,13,13'` 和格式字符串 `'%Y/%m/%d %h,%i,%s'` 返回 `'2022-09-10 11:13:13'`。如果格式字符串包含日期和时间部分，则返回一个 `DATETIME` 值。

```
select str_to_date('2022/09/10 11,13,13', '%Y/%m/%d %h,%i,%s');
      str_to_date
-----
2022-09-10 11:13:13
(1 row)
```

在下面的例子中，因为格式字符串仅包含日期部分，所以返回一个 `DATE` 值。

```
select str_to_date('2022/09/10', '%Y/%m/%d');
      str_to_date
-----
2022-09-10
(1 row)
```

在下面的例子中，因为格式字符串仅包含时间部分，所以返回一个 `TIME` 值。

```
select str_to_date('09:00:59', '%h:%i:%s');
      str_to_date
-----
09:00:59
(1 row)
```

如果启用了 `NO_ZERO_DATE` SQL 模式，则不允许使用零日期。在这种情况下，`STR_TO_DATE()` 返回 `NULL`。

```
set lightdb_sql_mode = 'no_zero_date';
select str_to_date('2022/09/0', '%Y/%m/%d');
      str_to_date
-----

(1 row)
```

Table H.84显示了可用于格式化 `STR_TO_DATE` 值的模板模式。

Table H.84. `str_to_date` 格式化的模板模式

Pattern	Description
<code>%a</code>	缩写的星期名称
<code>%b</code>	缩写的月份名称
<code>%c</code>	月份，值



Pattern	Description
%D	带有英文前缀的日期
%d	月份中的日, 值为 (00-31)
%e	月份中的日, 值为 (0-31)
%f	毫秒
%H	小时数, 值为 (00-23)
%h	小时数, 值为 (01-12)
%I	小时数, 值为 (01-12)
%i	分钟数, 值为 (00-59)
%j	年中的天数, 值为 (001-366)
%k	小时数, 值为 (0-23)
%l	小时数, 值为 (1-12)
%M	月份名称
%m	月份, 数字表示 (00-12)
%p	上午或下午
%r	时间, 12小时制 (hh:mm:ss 上午或下午)
%S	秒数 (00-59)
%s	从 1970 年 1 月 1 日开始到现在的秒数
%T	时间, 24小时制 (hh:mm:ss)
%W	星期名称
%Y	年份, 4位数字表示
%y	年份, 2位数字表示

### H. 53. 2. 3. 10. SYSDATE

#### 描述

返回函数执行时的时间。

#### 语法

`sysdate() RETURNS timestamp(0)`

#### 一般规则

- 返回的时间是函数执行时的时间, 是不确定的。
- 返回的时间受当前时区设置影响。
- 自版本23. 2开始支持此函数。

示例如下:

```
select sysdate;
       sysdate
-----
2023-05-06 16:06:59
(1 row)
```

### H. 53. 2. 3. 11. TIME\_TO\_SEC

描述

将参数转换为秒并返回。

语法

```
time_to_sec(p_time time) RETURNS int
```

通用规则

- 返回转换为秒的时间参数。

示例

在以下示例中，将时间参数 '22:23:00' 转换为秒数 80580 并返回。

```
select time_to_sec('22:23:00');
time_to_sec
-----
      80580
(1 row)
```

### H. 53. 2. 3. 12. TIMEDIFF

描述

计算时间差。

语法

```
timediff(p_date1 timestampz, p_date2 timestampz) RETURNS interval
```

通用规则

- TIMEDIFF() 将两个时间表达式相减并返回一个时间值。expr1 和 expr2 都必须是时间或日期时间表达式，并且类型必须相同。
- TIMEDIFF() 返回的结果范围限制在 TIME 值允许的范围内。或者，您可以使用 TIMESTAMPDIFF() 和 UNIX\_TIMESTAMP() 这两个函数，两者都返回整数值。

示例

在以下示例中，将 '2000-01-01 00:00:00' 和 '2000-01-01 00:00:00.000001' 相减并将结果表示为时间值。

```
select mysql.timediff('2000-01-01 00:00:00', '2000-01-01
00:00:00.000001');
      timediff
-----
-00:00:00.000001
(1 row)
```

### H. 53. 2. 3. 13. TIMESTAMPDIFF

#### 描述

从日期时间表达式中减去一个时间间隔。

#### 语法

```
timestampdiff( p_unit text, p_ts1 text, p_ts2 text) RETURNS bigint
timestampdiff( p_unit text, p_ts1 text, p_ts2 timestamptz) RETURNS bigint
timestampdiff( p_unit text, p_ts1 timestamptz, p_ts2 text) RETURNS bigint
timestampdiff( p_unit text, p_ts1 timestamptz, p_ts2 timestamptz) RETURNS
bigint
```

#### 通用规则

- 返回 `datetime_expr2 - datetime_expr1` 的值，其中 `datetime_expr1` 和 `datetime_expr2` 是日期或日期时间表达式。
- 其中一个表达式可以是日期，另一个可以是日期时间。必要时，日期值将被视为时间部分为 '00:00:00' 的日期时间。
- 结果的单位（整数）由单位参数给出。单位参数支持 MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER和YEAR, 可以带引号或者不带引号

#### 示例

在以下示例中，返回 '1911-11-11 11:23:45.123456' - '2021-12-12 12:12:12.654321'::timestamp, 结果（整数）由 MICROSECOND 给出。

```
select timestampdiff ('MICROSECOND', '1911-11-11 11:23:45.123456'::timestamp,
'2021-12-12 12:12:12.654321'::timestamp);
timestampdiff
```

```
-----
3474060507530865
(1 row)
```

```
select timestampdiff (MICROSECOND, '1911-11-11 11:23:45.123456'::timestamp,
'2021-12-12 12:12:12.654321'::timestamp);
timestampdiff
```

```
-----
3474060507530865
(1 row)
```

### H. 53. 2. 3. 14. TO\_DAYS

#### 描述

返回将日期参数转换为天数的结果。

#### 语法

```
to_days(p_date timestamp) RETURNS bigint
to_days(p_date timestamptz) RETURNS bigint
to_days( p_date text) RETURNS bigint
to_days( p_date text, p_format text) RETURNS bigint
```

`to_days(p_date bigint) RETURNS bigint`

一般规则

- 给定一个日期 `date`，返回一个日期编号（自公元 0 年以来的天数）。
- `TO_DAYS()` 不适用于格里高利历实行之前的日期（1582年），因为它没有考虑到更改日历时丢失的天数。对于 1582 年之前（以及其他地区可能的更晚年份）的日期，此函数的结果不可靠。

示例

在以下示例中，返回将日期参数 '1111-11-11' 转换为 406098 天的结果。

```
select to_days('1111-11-11');
to_days
-----
 406098
(1 row)
```

#### H. 53. 2. 3. 15. TO\_SECONDS

描述

返回将日期或日期时间参数转换为自公元 0 年以来的秒数。

语法

`to_seconds(p_date timestamp) RETURNS bigint`  
`to_seconds(p_date timestamptz) RETURNS bigint`

一般规则

- 给定日期或日期时间表达式 `expr`，返回自公元 0 年以来的秒数。如果 `expr` 不是有效的日期或日期时间值，则返回 `NULL`。
- 与 `TO_DAYS()` 一样，`TO_SECONDS()` 不适用于格里高利历实行之前的日期（1582年），因为它没有考虑到更改日历时丢失的天数。对于 1582 年之前（以及其他地区可能的更晚年份）的日期，此函数的结果不可靠。

示例

在以下示例中，返回将日期或日期时间参数 '2009-11-29' 转换为自公元 0 年以来的秒数 63426672000 的结果。

```
select to_seconds('2009-11-29'::timestamp);
to_seconds
-----
63426672000
(1 row)
```

#### H. 53. 2. 3. 16. UNIX\_TIMESTAMP

描述

返回根据时间格式的 UNIX 时间戳。

语法

```
UNIX_TIMESTAMP() RETURN int8
UNIX_TIMESTAMP(expr1 timestampz) RETURN int8
UNIX_TIMESTAMP(expr1 text) RETURN int8
```

一般规则

- 根据 `expr1` 参数的时间格式返回 UNIX 时间戳。如果没有参数，则返回当前 UNIX 时间戳。
- `expr1` 参数至少包含完整的日志格式，例如 20200101。

示例

在下面的示例中，返回对应参数20200101的Unix时间戳“1577808000”。

```
select unix_timestamp(20200101);
   unix_timestamp
-----
          1577808000
(1 row)
select unix_timestamp();
   unix_timestamp
-----
          1663051334
(1 row)
```

### H. 53. 2. 3. 17. WEEKOFYEAR

描述

根据时间返回当年的周数。

语法

```
WEEKOFYEAR(expr1 timestampz) RETURN int4
WEEKOFYEAR(expr1 text) RETURN int4
```

一般规则

- 根据时间表达式参数`expr1`，返回当年的周数。
- 至少包括完整的时间格式，例如20200202、200202。不支持少于6位的参数。

示例

在下面的示例中，返回“36”，因为“20220910”是一年中的第36周。

```
select weekofyear('20220910');
   weekofyear
-----
             36
```

(1 row)

### H. 53. 2. 3. 18. YEAR

描述

返回日期的年份。

语法

```
YEAR(expr1 timestamptz) RETURN int4
YEAR(expr1 text) RETURN int4
```

一般规则

- 返回根据expr1参数日期计算出的年份。
- 支持完整的年、月、日时间参数，例如“20200101”和“200101”。不支持仅包括月和日参数，例如“0101”和“0704”。

示例

在下面的示例中，返回“2020”，因为“20200101”的年份是2020。

```
select year(20200101);
   year
-----
   2020
(1 row)
select year('200101');
   year
-----
   2020
(1 row)
```

### H. 53. 2. 3. 19. QUARTER

描述

返回日期代表的季度，数字范围为1 ~ 4。

语法

```
QUARTER(timestamp) RETURN int4
```

一般规则

- 如果日期为 NULL 则返回 NULL
- 现在只支持timestamp（也即datetime）类型和任何能隐式转换为timestamp的入参

示例

在下面的例子中，因为日期是'20201001'，即10月，所以返回第4季度。

```

select quarter(cast('20201001' as date));
quarter
-----
         4
(1 row)

select quarter(cast('20201001' as datetime));
quarter
-----
         4
(1 row)

select quarter('20201001');
quarter
-----
         4
(1 row)

```

### H. 53. 2. 3. 20. LAST\_DAY

#### 描述

返回参数所在月份的最后一天的日期值

#### 语法

```

LAST_DAY(date) RETURN date
LAST_DAY(text) RETURN date

```

#### 一般规则

- 如果参数为NULL，则返回null
- 如果参数不可用，则报错
- 目前只支持date, text 类型(包括任何可以隐式转换到date, text的类型)

#### 示例

在下面的例子中，如果参数可用，参数所在月份的最后一天的日期值被返回：

```

lightdb@test_m=# SELECT LAST_DAY('2003-02-05');
last_day
-----
2003-02-28
(1 row)

lightdb@test_m=# SELECT LAST_DAY('2004-02-05');
last_day
-----
2004-02-29
(1 row)

lightdb@test_m=# SELECT LAST_DAY('2004-01-01 01:01:01');
last_day

```

```

-----
2004-01-31
(1 row)

lightdb@test_m=# SELECT LAST_DAY('2003-03-32');
ERROR: date/time field value out of range: "2003-03-32"
HINT: Perhaps you need a different "datestyle" setting.
CONTEXT: SQL function "last_day" statement 1
lightdb@test_m=#

```

### H. 53. 2. 3. 21. CURDATE

#### 描述

以 'YYYY-MM-DD' 格式返回当前日期。

#### 语法

CURDATE() RETURN date

#### 一般规则

- 目前不能像mysql 一样返回 YYYYMMDD 格式，因此不支持对返回结果进行乘法或除法，且对返回值的加法减法结果与mysql不同。Mysql 会根据函数使用的上下文是字符串还是数字上下文，来返回 'YYYY-MM-DD' 或 YYYYMMDD 格式的日期。

#### 示例

```

lightdb@test_m=# select curdate();
curdate
-----
2023-12-06
(1 row)

```

```

lightdb@test_m=# select curdate()+1;
?column?
-----
2023-12-07
(1 row)

```

```

lightdb@test_m=# select curdate()/2;
ERROR: operator does not exist: date / integer
LINE 1: select curdate()/2;
                ^

```

HINT: No operator matches the given name and argument types. You might need to add explicit type casts.

```

lightdb@test_m=# select curdate()*2;
ERROR: operator does not exist: date * integer
LINE 1: select curdate()*2;
                ^

```

HINT: No operator matches the given name and argument types. You might need to add explicit type casts.

```

lightdb@test_m=# select curdate()-2;
?column?
-----
2023-12-04

```



(1 row)

## H. 53. 2. 4. JSON Functions

提供了如下 JSON 函数：

- JSON\_ARRAY
- JSON\_OBJECT
- JSON\_EXTRACT
- JSON\_CONTAINS
- JSON\_CONTAINS\_PATH
- JSON\_PRETTY
- JSON\_REMOVE
- JSON\_INSERT
- JSON\_REPLACE
- JSON\_SET

### H. 53. 2. 4. 1. JSON\_ARRAY

描述

计算列表中的每个值（列表可能为空），返回包含这些值的 JSON 数组。

语法

```
JSON_ARRAY( VARIADIC "any" ) RETURNS jsonb
```

一般规则

- 返回 JSON 数组，数据类型为 jsonb。

示例

```
SELECT JSON_ARRAY(1, 'abc', NULL, TRUE, NOW());
           json_array
-----
[1, "abc", null, true, "2023-05-31T19:44:44.881055+08:00"]
(1 row)
```

### H. 53. 2. 4. 2. JSON\_OBJECT

描述

计算列表中的每个键值对（列表可能为空），返回包含这些键值对的 JSON 数组。

语法

JSON\_OBJECT( VARIADIC "any" ) RETURNS jsonb

一般规则

- 返回 JSON 对象，数据类型为 jsonb。
- 如果键的值为 NULL 或参数个数为奇数，则提示错误。

示例

```
SELECT JSON_OBJECT('id', 87, 'name', 'carrot');
      json_object
-----
{"id": 87, "name": "carrot"}
(1 row)

SELECT JSON_OBJECT('id', 87, 'name', 'carrot', 'id', 99);
      json_object
-----
{"id": 99, "name": "carrot"}
(1 row)
```

#### H. 53. 2. 4. 3. JSON\_EXTRACT

描述

通过 JSON 路径从 JSON 文档中选择匹配的数据，并返回。

语法

JSON\_EXTRACT(json\_doc jsonb, path jsonpath) RETURNS jsonb

一般规则

- 返回所有通过 JSON 路径选择出的值，数据类型为 jsonb。如果返回的值有多个，则这些值会按选择的顺序构成一个数组。否则返回单个值。
- 如果任意参数为 NULL，或者 JSON 未选中，则返回 NULL。
- 如果 JSON 文档或 JSON 路径是非法的值，则提示错误。

示例

```
SELECT JSON_EXTRACT('{"k1":11,"k2":22,"k2 2":33}', '$.k1');
      json_extract
-----
11
(1 row)

SELECT JSON_EXTRACT('{"k1":11,"k2":22,"k2 2":33}', '$."k2 2"');
      json_extract
-----
33
(1 row)
```

```
SELECT JSON_EXTRACT(' {"a":{"c":20, "d":{"b":{"b":30, "c":40}, "e":{"b":50}}},
"c":{"b":60, "c":70}, "b":80}', '$.a.c');
```

```
json_extract
```

```
-----
20
(1 row)
```

```
SELECT JSON_EXTRACT(' [10,"str", null, true]', '$[1]');
```

```
json_extract
```

```
-----
"str"
(1 row)
```

```
SELECT JSON_EXTRACT(' [10,"str", null, true]', '$[last]');
```

```
json_extract
```

```
-----
true
(1 row)
```

```
SELECT JSON_EXTRACT(' [10,"str", null, true]', '$[last-2]');
```

```
json_extract
```

```
-----
"str"
(1 row)
```

以下的示例中，有多个值匹配，所以返回的是数组。

```
SELECT JSON_EXTRACT(' [10,"str", null, true]', '$[1 to 3]');
```

```
json_extract
```

```
-----
["str", null, true]
(1 row)
```

```
SELECT JSON_EXTRACT(' [10,"str", null, true]', '$[*]');
```

```
json_extract
```

```
-----
[10, "str", null, true]
(1 row)
```

```
SELECT JSON_EXTRACT(' [10]', '$[*]');
```

```
json_extract
```

```
-----
[10]
(1 row)
```

```
SELECT JSON_EXTRACT(' {"k1":1, "k2":"22", "k2 2":[3, 33]}', '$.*');
```

```
json_extract
```

```
-----
[1, "22", [3, 33]]
(1 row)
```

#### H. 53. 2. 4. 4. JSON\_CONTAINS

描述

判断 JSON 文档是否包含指定的 JSON 路径，返回 true 或 false。

语法

JSON\_CONTAINS(json\_doc jsonb, json\_val jsonb, path jsonpath) RETURNS bool

一般规则

- 如果任意参数为 NULL，或者 JSON 未选中，则返回 NULL。
- 如果 JSON 文档或 JSON 路径是非法的值，则提示错误。
- 如果为了检测某个值是否存在于 JSON 文档中，请使用 JSON\_CONTAINS\_PATH()。

示例

```
SELECT JSON_CONTAINS('{"a": 111, "b": 222, "c": {"d": 4444}}', '111', '$.a');
      json_contains
```

```
-----
t
(1 row)
```

```
SELECT JSON_CONTAINS('{"a": "111", "b": 222, "c": {"d": 4444}}', '111', '$.a');
      json_contains
```

```
-----
f
(1 row)
```

```
SELECT JSON_CONTAINS('{"a": "111", "b": 222, "c": {"d": 4444}}', '" 111"',
      '$.a');
      json_contains
```

```
-----
f
(1 row)
```

```
SELECT JSON_CONTAINS('{"a": 111, "b": 222, "c": {"d": 4444}}', '{"d" :
      4444 }', '$.c');
      json_contains
```

```
-----
t
(1 row)
```

#### H. 53. 2. 4. 5. JSON\_CONTAINS\_PATH

描述

判断 JSON 文档是否包含指定的一个或多个 JSON 路径，返回 true 或 false。

语法

JSON\_CONTAINS\_PATH(json\_doc jsonb, one\_or\_all text, variadic jsonpath[]) RETURNS bool

一般规则

- 如果 JSON 路径均不存在，则返回 false。否则根据 one\_or\_all 参数确定返回值：

- 'one' : 如果任意一个 JSON 路径存在, 则返回 true。否则 false。
- 'all' : 如果所有的 JSON 路径都存在, 则返回 true。否则 false。
- 如果 JSON 文档或 JSON 路径是非法的值或 one\_or\_all 参数不是 'one' 或 'all', 则提示错误。
- 如果为了检测某个路径是否存在于 JSON 文档中, 请使用 JSON\_CONTAINS()。

示例

```
SELECT JSON_CONTAINS_PATH('{"a": 111, "b": 222, "c": {"d": 4444}}', 'one',
'$.a', '$.z');
json_contains_path
```

```
-----
t
(1 row)
```

```
SELECT JSON_CONTAINS_PATH('{"a": 111, "b": 222, "c": {"d": 4444}}', 'one',
'$.d');
json_contains_path
```

```
-----
f
(1 row)
```

```
SELECT JSON_CONTAINS_PATH('{"a": 111, "b": 222, "c": {"d": 4444}}', 'all', '$a',
'$.b', '$.c');
json_contains_path
```

```
-----
t
(1 row)
```

```
SELECT JSON_CONTAINS_PATH('{"a": 111, "b": 222, "c": {"d": 4444}}', 'all', '$a',
'$.b', '$.d');
json_contains_path
```

```
-----
f
(1 row)
```

#### H. 53. 2. 4. 6. JSON\_PRETTY

描述

格式化 JSON 文档。

语法

```
JSON_PRETTY(json_doc jsonb) RETURNS text
```

一般规则

- 返回值类型为 text。
- 无意义的空格和换行符不会影响输出结果。

- 对于参数为 NULL，则函数返回 NULL。如果参数是非法值，则提示错误。

示例

```
SELECT JSON_PRETTY('{"a":{"b":{"z":1},"c":[5,{"z":2}], "z":3}, "z":4}');
      json_pretty
-----
{
  "a": {
    "b": {
      "z": 1
    },
    "c": [
      5,
      {
        "z": 2
      }
    ],
    "z": 3
  },
  "z": 4
}
(1 row)
```

#### H. 53. 2. 4. 7. JSON\_REMOVE

描述

从 JSON 文档中删除数据，返回修改后的结果。

语法

JSON\_REMOVE(jsonb, jsonpath) RETURNS jsonb

一般规则

- 从 JSON 文档中删除数据，返回修改后的结果。 如果任意参数为 NULL，则返回 NULL。
- 如果 JSON 文档是非法的值，或 JSON 路径是非法的值，或 JSON 路径为 \$，或 JSON 路径包含 \* 或 \*\* 通配符，则提示错误。
- 如果指定的 JSON 路径不存在，函数不会提示错误，而是直接返回原 JSON 文档；

示例

```
SELECT JSON_REMOVE('{"a": 1, "b": [1, 2, 3], "c": {"d": 4}}', '$.c.d');
      json_remove
-----
{"a": 1, "b": [1, 2, 3], "c": {}}
(1 row)
```

```
SELECT JSON_REMOVE('{"a": 1, "b": [1, 2, 3], "c": {"d": 4}}', '$.b[last]');
      json_remove
-----
{"a": 1, "b": [1, 2], "c": {"d": 4}}
```

```
(1 row)

SELECT JSON_REMOVE('{"a": 1, "b": [1, 2, 3], "c": {"d": 4}}', '$.c.z');
           json_remove
-----
{"a": 1, "b": [1, 2, 3], "c": {"d": 4}}
(1 row)
```

和 MySQL 不同的时，在通过句号 `.` 访问 JSON 对象时，如果未使用双引号将对象名称引起来，并且该名称是合法的 ECMAScript 标识符名，但包含了部分特殊字符（比如 `$`），LightDB 会提示错误。

```
SELECT JSON_REMOVE('{"a": 1, "b": [1, 2, 3], "c": {"d": 4}}', '$.$z');
```

#### H. 53. 2. 4. 8. JSON\_INSERT

描述

向 JSON 文档中插入数据，返回修改后的结果。

语法

```
JSON_INSERT(jsonb, jsonpath, anyelement) RETURNS jsonb
JSON_INSERT(jsonb, jsonpath, text) RETURNS jsonb
```

一般规则

- 向 JSON 文档中插入数据，返回修改后的结果。如果任意参数为 NULL，则返回 NULL。
- 如果 JSON 文档是非法的值，或 JSON 路径是非法的值，或 JSON 路径包含 `*` 或 `**` 通配符，则提示错误。
- 如果 JSON 路径存在，则该函数什么也不做，直接返回原值。否则先插入数据并返回修改后的结果。

示例

```
SELECT JSON_INSERT('{ "a": 1, "b": [2, 3]}', '$.a', 10);
           json_insert
-----
{"a": 1, "b": [2, 3]}
(1 row)

SELECT JSON_INSERT('{ "a": 1, "b": [2, 3]}', '$.c', 200);
           json_insert
-----
{"a": 1, "b": [2, 3], "c": 200}
(1 row)
```

#### H. 53. 2. 4. 9. JSON\_REPLACE

描述

替换 JSON 文档的数据，返回修改后的结果。

语法

JSON\_REPLACE(jsonb, jsonpath, anyelement) RETURNS jsonb  
 JSON\_REPLACE(jsonb, jsonpath, text) RETURNS jsonb

一般规则

- 替换 JSON 文档的数据，返回修改后的结果。如果任意参数为 NULL，则返回 NULL。
- 如果 JSON 文档是非法的值，或 JSON 路径是非法的值，或 JSON 路径包含 \* 或 \*\* 通配符，则提示错误。
- 如果 JSON 路径不存在，则该函数什么也不做，直接返回原值。否则先替换数据并返回修改后的结果。

示例

```
SELECT JSON_REPLACE(' { "a": 1, "b": [2, 3]}', '$.a', 10);
      json_replace
-----
{"a": 10, "b": [2, 3]}
(1 row)

SELECT JSON_REPLACE(' { "a": 1, "b": [2, 3]}', '$.c', 'null');
      json_replace
-----
{"a": 1, "b": [2, 3]}
(1 row)

SELECT JSON_REPLACE('100', '$', '{ "a": 1, "b": [2, 3]}>::jsonb);
      json_replace
-----
{"a": 1, "b": [2, 3]}
(1 row)
```

和 MySQL 不同的时，在通过句号 . 访问 JSON 对象时，如果未使用双引号将对象名称引起来，并且该名称是非法的 ECMAScript 标识符名，LightDB 不会提示错误而是返回原值。

```
SELECT JSON_REPLACE(' { "a": 1, "b": [2, 3]}', '$.b.2z', 4);
      json_replace
-----
{"a": 1, "b": [2, 3]}
(1 row)
```

#### H. 53. 2. 4. 10. JSON\_SET

描述

向 JSON 文档中插入数据或替换 JSON 文档的数据，返回修改后的结果。

语法

JSON\_SET(jsonb, jsonpath, anyelement) RETURNS jsonb



JSON\_SET(jsonb, jsonpath, text) RETURNS jsonb

一般规则

- 向 JSON 文档中插入数据或替换 JSON 文档的数据，返回修改后的结果。如果任意参数为 NULL，则返回 NULL。
- 如果 JSON 文档是非法的值，或 JSON 路径是非法的值，或 JSON 路径包含 \* 或 \*\* 通配符，则提示错误。
- 如果 JSON 路径存在，先替换数据并返回修改后的结果。如果 JSON 路径不存在，则先插入数据并返回修改后的结果。

示例

```
SELECT JSON_SET('{"a": 1, "b": [2, 3]}', '$.a', 10);
      json_set
```

```
-----
{"a": 10, "b": [2, 3]}
(1 row)
```

```
SELECT JSON_SET('{"a": 1, "b": [2, 3]}', '$.c', '[true, false]');
      json_set
```

```
-----
{"a": 1, "b": [2, 3], "c": "[true, false]"}
(1 row)
```

```
SELECT JSON_SET('{"a": 1, "b": [2, 3]}', '$.b.z', 4);
      json_set
```

```
-----
{"a": 1, "b": [2, 3]}
(1 row)
```

## H. 53. 2. 5. 条件表达式

支持以下用于进行比较的函数：

- IF

### H. 53. 2. 5. 1. IF

描述

如果expr1为TRUE，则IF ( ) 返回expr2。否则，返回expr3。

语法

IF(expr1, expr2, expr3)

一般规则

- 如果expr2或expr3仅有一个明确为NULL，则IF ( ) 函数的结果类型为“text”。
- 如果expr2和expr3中有一个是字符串，则IF ( ) 函数的结果类型为“text”，否则为“numeric”。

- expr2和expr3可以使用以下数据类型：
- CHAR
- VARCHAR
- TEXT
- INTEGER
- BIGINT
- SMALLINT
- NUMERIC
- REAL
- DOUBLE PRECISION
- BOOLEAN
- TIMESTAMPTZ

示例

在以下示例中，返回3，因为 $1 > 2$ 的值为false。

```
SELECT IF(1>2, 2, 3);
      if
-----
      3
(1 row)
```

## H. 53. 2. 6. 集合函数

以下聚合函数得到支持：

- ANY\_VALUE
- GROUP\_CONCAT

### H. 53. 2. 6. 1. ANY\_VALUE

描述

为了与MySQL模式兼容，仅在BY模式下的分组行为语法。

语法

```
ANY_VALUE(anelement)
```

通用规则

- 此函数对于启用ONLY\_FULL\_GROUP\_BY SQL模式的MySQL兼容GROUP BY查询非常有用，对于MySQL拒绝您知道有效的查询的情况，因为MySQL无法确定，任何值函数将返回导致歧义的第一行。函数返回值和类型与其参数的返回值和类型相同。

请注意，此行为仅是与MySQL兼容的非标准语法。它用于确保在LightDB中的执行不会报告错误，但无法保证结果与MySQL完全一致。不建议使用此功能。请使用标准语法格式。

示例

在以下示例中，如果id是一个非索引列，则以下查询将失败：

```
select col, co2, max(co3) from t1 group by id;
ERROR: column "t1.col" must appear in the GROUP BY clause or be used in an
aggregate function
LINE 1: select col, co2, max(co3) from t1 group by id;
```

使用ANY\_VALUE()来引用col1和co2:

```
select any_value(col) as col, any_value(co2) as co2, max(co3) from t1 group by
id;
col | co2 | max
-----+-----+-----
 6 | 9 | 30
 4 | 5 | 5
 3 | 3 | 4
 2 | 3 | 4
(4 rows)
```

#### H. 53. 2. 6. 2. GROUP\_CONCAT

描述

返回字符串值的连接，以分隔符为分隔符。

语法

```
GROUP_CONCAT( [DISTINCT] column_name1 [ORDER BY {unsigned_integer | column_name2
| expr} [ASC | DESC] [SEPARATOR delimiter TEXT]) returns TEXT
```

通用规则

- 此函数从组中返回一个具有连接的非NULL值的字符串结果。
- 如果没有非NULL值，则返回NULL。
- 要对结果中的值进行排序，请使用ORDER BY子句。要按相反的顺序排序，请在ORDER BY子句中按排序列的名称添加DESC（降序）关键字。默认为升序。可以使用ASC关键字明确指定。
- 要消除重复值，请使用DISTINCT子句。
- 默认情况下，组中值之间的分隔符是逗号（,）。要明确指定分隔符，请使用SEPARATOR后跟应在组值之间插入的字符串文字值。要完全消除分隔符，请指定SEPARATOR''。

示例

在以下示例中，返回以':'为分隔符的表t中列col1的值的结果。

```
select GROUP_CONCAT(col1 separator ':') from (values('BBBB',1), ('AAAA',1),
('CCCC',2)) as t(col1 ,col2) group by col2;
```

```
group_concat
-----
BBBB:AAAA
CCCC
(2 rows)
```

## H. 53. 2. 7. 系统信息函数和运算符

支持以下系统信息函数：

- 数据库

### H. 53. 2. 7. 1. 数据库

描述

同`current_schema()`的同义词。在MySQL中，模式的概念类似于数据库。您可以通过设置`search_path`来指定当前模式。

语法

`DATABASE()` returns TEXT

一般规则

- 返回搜索路径中第一个模式的名称（如果搜索路径为空，则返回空值）。
- 与MySQL类似，这是创建没有指定目标模式的任何表或其他命名对象时将使用的模式。

示例

在下面的示例中，当前模式为`lightdb`。

```
select database();
database
```

```
-----
lightdb
(1 row)
```

## H. 53. 2. 8. 杂项函数

支持以下杂项函数：

- `BIN_TO_UUID`
- `COMPRESS`
- `UNCOMPRESS`
- `UUID`
- `UUID_TO_BIN`

### H. 53. 2. 8. 1. BIN\_TO\_UUID

#### 描述

将二进制UUID转换为字符串。

#### 语法

```
bin_to_uuid(p_uuid bytea, swap_flag int4 default 0) RETURNS uuid
```

#### 一般规则

- BIN\_TO\_UUID()是UUID\_TO\_BIN()的反函数。它将二进制UUID转换为字符串UUID并返回结果。二进制值应为VARBINARY(16)值的UUID。
- 返回值是由连字符分隔的五个十六进制数字组成的utf8字符串。（有关此格式的详细信息，请参见UUID()函数描述。）如果UUID参数为NULL，则返回值为NULL。如果任何参数无效，则会发生错误。

#### 示例

在下面的示例中，将二进制UUID “\x0768a47dd355372b4926ee668b6cb443” 转换为字符串 “0768a47d-d355-372b-4926-ee668b6cb443”。

```
select bin_to_uuid('\x0768a47dd355372b4926ee668b6cb443');
       bin_to_uuid
-----
0768a47d-d355-372b-4926-ee668b6cb443
(1 row)
```

### H. 53. 2. 8. 2. COMPRESS

#### 描述

压缩字符串并返回二进制数据。

#### 语法

```
COMPRESS(expr1 text) RETURN bytea
```

#### 一般规则

- 压缩字符串参数expr1并返回二进制数据。相应的解压函数是UNCOMPRESS。

#### 示例

在以下示例中，将' abc123' 压缩为二进制数据'\x06000000616263313233'。

```
select compress(' abc123');
       compress
-----
\x06000000616263313233
```

(1 row)

### H. 53. 2. 8. 3. UNCOMPRESS

#### 描述

解压由compress()函数压缩的二进制数据，并恢复原始字符串。

#### 语法

```
UNCOMPRESS(expr1 bytea) RETURN text
```

#### 通用规则

- 解压由compress()函数压缩的二进制数据expr1参数，并恢复原始字符串。

#### 示例

在以下示例中，将二进制数据'\x06000000616263313233'解压为bCompress 'abc123'。

```
select uncompress(compress('abc123'));
 uncompress
-----
 abc123
(1 row)
```

### H. 53. 2. 8. 4. UUID

#### Description

生成一个36字节的uuid字符串

#### Syntax

```
uuid() RETURNS text
```

#### General rules

- UUID()返回一个符合RFC 4122 版本4中描述的UUID的值。结果是一个128位整数，以五个十六进制数字组成的字符串格式呈现。

#### Example

以下示例生成一个uuid字符串

```
select uuid();
      uuid
-----
f31f29e5-481f-4666-aa74-8cee3aa44545
(1 row)
```

### H. 53. 2. 8. 5. UUID\_TO\_BIN

#### 描述

将字符串UUID转换为二进制。

#### 语法

```
uuid_to_bin(p_uuid uuid, swap_flag int4 default 0) RETURNS bytea
```

#### 通用规则

- 将字符串UUID转换为二进制UUID并返回结果。(IS\_UUID() 函数说明列出了允许的字符串UUID格式。)
- 返回的二进制UUID是VARBINARY(16)值。如果UUID参数为NULL，则返回值为NULL。如果任何参数无效，则会出现错误。

#### 示例

在以下示例中，将字符串UUID '0768a47d-d355-372b-4926-ee668b6cb443' 转换为二进制 '\x0768a47dd355372b4926ee668b6cb443'。

```
select uuid_to_bin('0768a47d-d355-372b-4926-ee668b6cb443'::uuid)::text;
       uuid_to_bin
-----
\x0768a47dd355372b4926ee668b6cb443
(1 row)
```

### H. 53. 2. 9. 运算符

支持以下运算符：

- IFNULL
- ISNULL

#### H. 53. 2. 9. 1. IFNULL

#### 描述

根据参数是否为空，返回相应的值。

#### 语法

```
IFNULL(expr1 anycompatible, expr2 anycompatible) RETURNS anycompatible
IFNULL(expr1 text, expr2 text) RETURNS text
```

#### 通用规则

- 判断第一个参数是否为null。如果为null，则返回第二个参数的值。如果不为null，则返回第一个参数的值。
- 两个输入参数必须属于相同的类别，并且可以确定 pg\_type 的 typcategory 字段。

### 示例

在以下示例中，当第一个参数为null时，返回第二个参数15。否则，返回第一个参数1。

```
select ifnull(1, 15);
   ifnull
-----
         1
(1 row)
select ifnull(null, 15);
   ifnull
-----
        15
(1 row)
```

### H. 53. 2. 9. 2. ISNULL

#### 描述

测试参数是否为NULL。

#### 语法

```
isnull(text) RETURNS bool
isnull(numeric) RETURNS bool
isnull(timestampz) RETURNS bool
```

#### 通用规则

- 如果expr为NULL，ISNULL()返回true，否则返回false。

#### 例子

在下面的例子中，expr不为NULL，ISNULL()返回false。

```
select isnull(0)::text;
   isnull
-----
   false
(1 row)
```

### H. 53. 2. 10. 算术运算符

支持以下算术运算符：

- /

#### H. 53. 2. 10. 1. /

#### 描述

除法运算符。被零除会产生一个NULL结果，报告一个警告日志。

#### 例子



除法运算符示例。被零除会产生一个NULL结果，报告一个警告日志。

```
select 2 / 4;
?column?
-----
0.50000000000000000000000000000000
(1 row)

select 1.0 / 2;
?column?
-----
0.50000000000000000000000000000000
(1 row)

select 2 / 0;
WARNING: division by zero
?column?
-----

(1 row)
```

## H. 53. 2. 11. 比较操作符

支持如下比较操作符：

- =
- !=

### H. 53. 2. 11. 1. =

描述

等于操作符，支持varchar与bool或text。相等返回true，不相等返回false。

示例

```
select '1'::varchar = true;
?column?
-----
t
(1 row)

select '1'::varchar = '1'::text;
?column?
-----
t
(1 row)

create table a(b varchar(5), c int);
insert into a values(true, 1);
insert into a values('true', 2);

select * from a;
 b | c
-----+---
```

```

1 | 1
true | 2
(2 rows)

```

```

select * from a where b = true;
 b | c
----+---
 1 | 1
(1 row)

```

```

select * from a where b = 'true';
  b | c
-----+---
 true | 2
(1 row)

```

#### H. 53. 2. 11. 2. !=

##### 描述

不等于操作符，支持varchar与bool或text。不相等返回true，相等返回false。

##### 示例

```

select '1' != true;
?column?
-----
 f
(1 row)

```

```

select '1'::varchar != '1'::text;
?column?
-----
 f
(1 row)

```

```

create table a(b varchar(5), c int);
insert into a values(true, 1);
insert into a values('true', 2);

```

```

select * from a;
  b | c
-----+---
 1 | 1
true | 2
(2 rows)

```

```

select * from a where b != true;
  b | c
-----+---
 true | 2
(1 row)

```

```

select * from a where b != 'true';
  b | c
-----+---
 1 | 1

```

(1 row)

## H. 53. 2. 12. 类型转换

### H. 53. 2. 12. 1. 转换bool类型至varchar

#### 描述

此前，bool类型转换为varchar形式上保持不变，如true转换成' true' 或false转换成' false'。Myfce改变上述规则，与mysql兼容，转换bool类型true至字符串' 1' 及false至字符串' 0'。

#### 示例

```
create table tb (a varchar(5), b int);
insert into tb values(' true', 2);      -- 插入字符串' true'，与期望一致
insert into tb values(true, 1);        -- 插入布尔类型true，而期望插入字符串' 1'
insert into tb values(1, 3);           -- 插入数值1，而期望插入字符串' 1'
insert into tb values(' 1', 4);        -- 插入字符串' 1'，与期望一致
insert into tb values(false, 5);       -- 插入布尔类型false，而期望插入字符串' 0'
insert into tb values(' false', 6);    -- 插入字符串' false'，与期望一致
insert into tb values(0, 7);           -- 插入数值0，而期望插入字符串' 0'
insert into tb values(' 0', 8);        -- 插入字符串' 0'，与期望一致
select * from tb;
```

a	b
true	2
1	1
1	3
1	4
0	5
false	6
0	7
0	8

(8 rows)

### H. 53. 2. 12. 2. cast(expr as unsigned)

#### 描述

转换expr表达式为unsigned bigint 值。

#### 通用规则

- 返回类型与mysql不同，LightDB 为 numeric，mysql 为 unsigned bigint.
- 表达式不能为日期/时间类型（mysql 支持）。具体见示例。
- 字符串格式的浮点型不会被截断，这与mysql不同，mysql会截断，如' 1.9' 会被截断为 1。具体见示例。
- 负浮点型常量不会被截断，这与mysql5.7不同，mysql5.7会截断（mysql8.0不会），如-1.9 会被截断为0（但mysql中浮点型字段中存负数，转换时不会被截断）。具体见示例。
- 负数在溢出的时候会被截断为 9223372036854775808，这与mysql8.0相同（mysql5.7会截断为0）。具体见示例。

- 支持转换科学计数法格式的字符串到unsigned, 如: '2e2' 转为200(mysql 不支持, 转换后为2)。具体见示例。

示例

```
lightdb@test_m=# select cast(123 as unsigned) as unsigned;
unsigned
-----
      123
(1 row)
```

```
lightdb@test_m=# select cast(-123 as unsigned) as unsigned;
unsigned
-----
18446744073709551493
(1 row)
```

```
lightdb@test_m=# select cast(18446744073709551615 as unsigned) as unsigned;
unsigned
-----
18446744073709551615
(1 row)
```

```
lightdb@test_m=# select cast(18446744073709551616 as unsigned) as unsigned;
WARNING: unsigned bigint out of range, Truncated
unsigned
-----
18446744073709551615
(1 row)
```

```
lightdb@test_m=# select cast(-9223372036854775808 as unsigned) as unsigned;
unsigned
-----
9223372036854775808
(1 row)
```

```
-- mysql5.7为 0 , mysql8.0: 9223372036854775808
lightdb@test_m=# select cast(-9223372036854775809 as unsigned) as unsigned;
WARNING: unsigned bigint out of range, Truncated
unsigned
-----
9223372036854775808
(1 row)
```

```
lightdb@test_m=# select cast(1.4 as unsigned) as unsigned;
unsigned
-----
      1
(1 row)
```

```
lightdb@test_m=# select cast(1.5 as unsigned) as unsigned;
unsigned
-----
      2
(1 row)
```

```
lightdb@test_m=# select cast(1.9 as unsigned) as unsigned;
```

```

unsigned
-----
      2
(1 row)

-- mysql5.7为 0 , mysql8.0: 18446744073709551614
lightdb@test_m=# select cast(-1.9 as unsigned) as unsigned;
      unsigned
-----
18446744073709551614
(1 row)

-- mysql5.7为 0 , mysql8.0: 18446744073709551614
lightdb@test_m=# select cast(-1.5 as unsigned) as unsigned;
      unsigned
-----
18446744073709551614
(1 row)

-- mysql5.7为 0 , mysql8.0: 18446744073709551615
lightdb@test_m=# select cast(-1.4 as unsigned) as unsigned;
      unsigned
-----
18446744073709551615
(1 row)

-- mysql: 1
lightdb@test_m=# select cast('1.4' as unsigned) as unsigned;
      unsigned
-----
      1
(1 row)

-- mysql: 1
lightdb@test_m=# select cast('1.5' as unsigned) as unsigned;
      unsigned
-----
      2
(1 row)

-- mysql: 1
lightdb@test_m=# select cast('1.9' as unsigned) as unsigned;
      unsigned
-----
      2
(1 row)

-- mysql: 18446744073709551615
lightdb@test_m=# select cast('-1.4' as unsigned) as unsigned;
      unsigned
-----
18446744073709551615
(1 row)

-- mysql: 18446744073709551615
lightdb@test_m=# select cast('-1.5' as unsigned) as unsigned;
      unsigned
-----

```

```

18446744073709551614
(1 row)

-- mysql: 18446744073709551615
lightdb@test_m=# select cast('-1.9' as unsigned) as unsigned;
          unsigned
-----
18446744073709551614
(1 row)

lightdb@test_m=# select cast(2e2 as unsigned) as unsigned;
          unsigned
-----
          200
(1 row)

lightdb@test_m=# select cast('2e2' as unsigned) as unsigned;
          unsigned
-----
          200
(1 row)

lightdb@test_m=# create table test_c(key1 float4);
CREATE TABLE
lightdb@test_m=# insert into test_c values('-1.6');
INSERT 0 1
lightdb@test_m=# insert into test_c values('-1.4');
INSERT 0 1
lightdb@test_m=# insert into test_c values('-1.5');
INSERT 0 1
lightdb@test_m=# select cast(key1 as unsigned) , key1 from test_c;
 numeric2unsigned | key1
-----+-----
18446744073709551614 | -1.6
18446744073709551615 | -1.4
18446744073709551614 | -1.5
(3 rows)

lightdb@test_m=# select cast(cast('20121212' as date) as unsigned);
ERROR:  cannot cast type date to numeric
LINE 1: select cast(cast('20121212' as date) as unsigned);
                ^

lightdb@test_m=#

```

### H. 53. 2. 13. AES加解密

支持下列AES加解密和十六进制转换操作。

- hex
- unhex
- aes\_encrypt
- aes\_decrypt

#### H. 53. 2. 13. 1. hex

描述

hex函数将bytea类型数据转成text类型。

语法

```
hex(data bytea) returns text
```

示例

```
select hex('\x0123456789abcdef');
      hex
-----
0123456789abcdef
(1 row)
```

#### H. 53. 2. 13. 2. unhex

描述

unhex函数将varchar类型数据转成bytea类型。

语法

```
unhex(data varchar) returns bytea
```

示例

```
select unhex('0123456789abcdef');
      unhex
-----
\x0123456789abcdef
(1 row)
```

#### H. 53. 2. 13. 3. aes\_encrypt

描述

AES加密函数。

语法

```
aes_encrypt(data bytea, key bytea) returns bytea
aes_encrypt(data varchar, key varchar) returns bytea
```

示例

```
select aes_encrypt('world', '123456');
      aes_encrypt
-----
\x85400d3bbeeb51259f497fcb4bee03e
(1 row)
```

#### H. 53. 2. 13. 4. aes\_decrypt

描述

AES解密函数。

语法

```

aes_decrypt(data bytea, key bytea) returns bytea
aes_decrypt(data varchar, key varchar) returns bytea
aes_decrypt(data bytea, key varchar) returns bytea
    
```

示例

```

select aes_decrypt(aes_encrypt(' world', ' 123456'), ' 123456');
aes_decrypt
-----
\x776f726c64
(1 row)
    
```

### H. 53. 3. 查询

支持如下查询：

- 对偶表

对偶表是系统提供了一种虚表。当执行SQL访问非必须的基表，如执行测试获取函数表达式或操作符的结果。

示例

下述示例，返回当前系统日期

```

SELECT CURRENT_DATE "date" FROM DUAL;
date
-----
2023-05-13
(1 row)
    
```

## H. 54. mysql\_fdw

本LightDB扩展实现了一个外部数据包装器（FDW），用于 MySQL<sup>11</sup>。

### H. 54. 1. Installation

要编译 MySQL<sup>12</sup> 外部数据包装器，需要 MySQL 的 C 客户端库，默认版本为 8.0.26。该库可以从官方 MySQL 网站<sup>13</sup>下载。

<sup>11</sup> <https://www.mysql.com/>

<sup>12</sup> <https://www.mysql.com/>

<sup>13</sup> <https://dev.mysql.com/downloads/mysql/>



在库路径中也必须包含 `mysql_config`，它位于 MySQL 的 `lib` 目录中。

```
$ export LD_LIBRARY_PATH=/usr/lib64/mysql/:$LD_LIBRARY_PATH
```

## H. 54. 2. Enhancements

最新版本的 `mysql_fdw` 增加了以下增强功能：

### H. 54. 2. 1. Write-able FDW

以前的版本仅为只读，最新版本提供了写入功能。用户现在可以使用 `mysql_fdw` 对外部表执行插入、更新和删除语句。它使用 PG 类型转换机制来提供 MySQL 和 PG 数据类型之间的相反类型转换。

### H. 54. 2. 2. Connection Pooling

最新版本附带一个连接池，它为同一会话中的所有查询利用相同的 MySQL 数据库连接。以前的版本会为每个查询打开一个新的 MySQL 数据库连接，这是一个性能增强。

### H. 54. 2. 3. WHERE clause push-down

最新版本将外部表的 WHERE 条件下推到外部服务器。外部表上的 WHERE 条件将在外部服务器上执行，因此需要传输到 LightDB 的行数更少。这是一个性能特性。

### H. 54. 2. 4. Column push-down

以前的版本从目标外部表获取所有列。最新版本执行列下推，并仅返回选择目标列表中的列。这是一个性能特性。

### H. 54. 2. 5. Prepared Statement

(重构 `select` 查询以使用预编译语句)

现在，`select` 查询使用预编译语句而不是简单查询协议。

### H. 54. 2. 6. JOIN push-down

`mysql_fdw` 现在还支持 JOIN 下推。来自同一远程 MySQL 服务器的两个外部表之间的连接被推送到远程服务器，而不是获取两个表的所有行并在本地执行连接，从而提高性能。目前，仅涉及关系和算术运算符的连接子句中的连接被下推以避免任何潜在的连接失败。此外，仅支持 INNER 和 LEFT/RIGHT OUTER 连接，不支持 FULL OUTER、SEMI 和 ANTI 连接。这是一个性能特性。

## H. 54. 3. Usage

可以在 MySQL 外部服务器对象上设置以下参数：

- `host`: MySQL 服务器的地址或主机名。默认值为 `127.0.0.1`。
- `port`: MySQL 服务器的端口号。默认值为 `3306`。
- `secure_auth`: 启用或禁用安全身份验证。默认值为 `true`。
- `init_command`: 连接到 MySQL 服务器时要执行的 SQL 语句。
- `use_remote_estimate`: 控制 `mysql_fdw` 是否发出远程 EXPLAIN 命令来获取成本估算。默认值为 `false`。

- `reconnect`: 启用或禁用自动重新连接到 MySQL 服务器的功能，如果发现现有连接已经丢失。默认值为 `false`。
- `ssl_key`: 客户端私钥文件的路径名。
- `ssl_cert`: 客户端公钥证书文件的路径名。
- `ssl_ca`: 证书授权机构 (CA) 证书文件的路径名。如果使用此选项，必须指定与服务器使用的证书相同的证书。
- `ssl_cacpath`: 包含可信 SSL CA 证书文件的目录的路径名。
- `ssl_cipher`: SSL 加密的可允许的密码列表。
- `fetch_size`: 此选项指定 `mysql_fdw` 在每个获取操作中应获取的行数。它可以为外部表或外部服务器指定。在表上指定的选项会覆盖为服务器指定的选项。默认值为 100。

可以在 MySQL 外部表对象上设置以下参数：

- `dbname`: 要查询的 MySQL 数据库的名称。这是一个必选项。
- `table_name`: MySQL 表的名称，默认值与外部表相同。
- `max_blob_size`: 读取而不截断的最大 blob 大小。
- `fetch_size`: 与外部服务器的 `fetch_size` 参数相同。

创建用户映射时需要提供以下参数。

- `username`: 连接到 MySQL 时要使用的用户名。
- `password`: 用于鉴别连接到 MySQL 服务器的密码。

## H. 54. 4. Examples

```

-- load extension first time after install
CREATE EXTENSION mysql_fdw;

-- create server object
CREATE SERVER mysql_server
  FOREIGN DATA WRAPPER mysql_fdw
  OPTIONS (host '127.0.0.1', port '3306');

-- create user mapping
CREATE USER MAPPING FOR LightDB
  SERVER mysql_server
  OPTIONS (username 'foo', password 'bar');

-- create foreign table
CREATE FOREIGN TABLE warehouse
  (
    warehouse_id int,
    warehouse_name text,
    warehouse_created timestamp
  )
  SERVER mysql_server
  OPTIONS (dbname 'db', table_name 'warehouse');

-- insert new rows in table

```

```

INSERT INTO warehouse values (1, 'UPS', current_date);
INSERT INTO warehouse values (2, 'TV', current_date);
INSERT INTO warehouse values (3, 'Table', current_date);

-- select from table
SELECT * FROM warehouse ORDER BY 1;

warehouse_id | warehouse_name | warehouse_created
-----+-----+-----
          1 | UPS            | 10-JUL-20 00:00:00
          2 | TV             | 10-JUL-20 00:00:00
          3 | Table          | 10-JUL-20 00:00:00

-- delete row from table
DELETE FROM warehouse where warehouse_id = 3;

-- update a row of table
UPDATE warehouse set warehouse_name = 'UPS_NEW' where warehouse_id =
1;

-- explain a table with verbose option
EXPLAIN VERBOSE SELECT warehouse_id, warehouse_name FROM warehouse
WHERE warehouse_name LIKE 'TV' limit 1;

```

QUERY PLAN

```

-----
Limit (cost=10.00..11.00 rows=1 width=36)
  Output: warehouse_id, warehouse_name
  -> Foreign Scan on public.warehouse (cost=10.00..1010.00
rows=1000 width=36)
    Output: warehouse_id, warehouse_name
    Local server startup cost: 10
    Remote query: SELECT `warehouse_id`, `warehouse_name` FROM
`db`.`warehouse` WHERE ((`warehouse_name` LIKE BINARY 'TV'))

```

## H. 55. oracle\_fdw

oracle\_fdw 是 LightDB 的一个扩展，提供了一个外部数据包装器，用于轻松高效地访问 Oracle 数据库，包括 WHERE 条件和所需列的下推以及全面的 EXPLAIN 支持。

oracle\_fdw 由 Laurenz Albe 编写，Vincent Mora of Oslandia 和 NTT OSS Center 的 Tatsuro Yamada 也做出了重要贡献。

特别感谢 Christian Ullrich 在 Windows 上的持续帮助。

### H. 55. 1. Cookbook

以下是使用 oracle\_fdw 的简单示例。

更详细的信息将在“选项”和“用法”部分中提供。您还应该阅读 LightDB 关于外部数据和相关命令的文档。

为了本示例，假设您可以使用以下命令作为操作系统用户 lightdb（或启动 LightDB 服务器的任何人）连接：

```
sqlplus orauser/orapwd@//dbserver.mydomain.com:1521/ORADB
```

这意味着 Oracle 客户端和环境已设置正确。

我们想要访问一个定义如下的表：

```
SQL> DESCRIBE oratab
Name                               Null?    Type
-----
ID                                  NOT NULL NUMBER(5)
TEXT                                VARCHAR2(30)
FLOATING                            NOT NULL NUMBER(7,2)
```

然后将 oracle\_fdw 配置为 LightDB 超级用户，如下所示：

```
pgdb=# CREATE EXTENSION oracle_fdw;
pgdb=# CREATE SERVER oradb FOREIGN DATA WRAPPER oracle_fdw
        OPTIONS (dbserver ' //dbserver.mydomain.com:1521/ORADB');
```

（您可以使用其他命名方法或本地连接，请参阅下面选项 dbserver 的描述。）

只有在真正需要的情况下才使用超级用户是个好主意，因此让我们允许普通用户使用外部服务器（这并非本示例工作所必需，但我建议这样做）：

```
pgdb=# GRANT USAGE ON FOREIGN SERVER oradb TO pguser;
```

然后，您可以作为 pguser 连接到 LightDB 并定义：

```
pgdb=> CREATE USER MAPPING FOR pguser SERVER oradb
        OPTIONS (user 'orauser', password 'orapwd');
```

（您可以使用外部身份验证来避免存储 Oracle 密码；请参阅下面的说明。）

```
pgdb=> CREATE FOREIGN TABLE oratab (
        id          integer          OPTIONS (key 'true') NOT
NULL,
        text        character varying(30),
        floating    double precision NOT NULL
) SERVER oradb OPTIONS (schema 'ORAUSER', table 'ORATAB');
```

（请记住表名和模式名 -- 后者是可选的 -- 通常必须大写。）

现在，您可以像使用常规的 LightDB 表一样使用该表。

## H. 55. 2. Objects created by the extension

```
FUNCTION oracle_fdw_handler() RETURNS fdw_handler
```

```
FUNCTION oracle_fdw_validator(text[], oid) RETURNS void
```

这些函数是必要的处理程序和验证器函数，用于创建外部数据包装器。

```
FOREIGN DATA WRAPPER oracle_fdw
HANDLER oracle_fdw_handler
VALIDATOR oracle_fdw_validator
```

扩展自动创建了一个名为 `oracle_fdw` 的外部数据包装器。

通常这就是您所需要的，然后您可以继续定义外部服务器。您可以创建其他 Oracle 外部数据包装器，例如如果您需要设置 `nls_lang` 选项（您可以更改现有的 `oracle_fdw` 包装器，但所有修改将在转储/还原后丢失）。

```
FUNCTION oracle_close_connections() RETURNS void
```

此函数可用于关闭此会话中的所有打开的 Oracle 连接。有关进一步说明，请参见“用法”部分。

```
FUNCTION oracle_diag(name DEFAULT NULL) RETURNS text
```

此函数仅用于诊断目的。

它将返回 `oracle_fdw`、LightDB 服务器和 Oracle 客户端的版本。如果不带参数或为 NULL 调用，还将返回用于建立 Oracle 连接的一些环境变量的值。

```
FUNCTION oracle_execute(server name, stmt text) RETURNS void
```

此函数可用于在远程 Oracle 服务器上执行任意 SQL 语句。仅适用于不返回结果的语句（通常是 DDL 语句）。

在使用此函数时要小心，因为它可能会干扰 `oracle_fdw` 的事务管理。请记住，在 Oracle 中运行 DDL 语句会发出隐式 COMMIT。最好建议您在多语句事务之外使用此函数。

## H. 55. 3. 选项

### H. 55. 3. 1. 外部数据包装器选项

（注意：如果您修改默认的外部数据包装器 `oracle_fdw`，则任何更改都将在转储/还原时丢失。如果要选项持久化，请创建一个新的外部数据包装器。随软件提供的 SQL 脚本中包含可用的 CREATE FOREIGN DATA WRAPPER 语句。）

- `nls_lang`（可选）

设置 Oracle 的 `NLS_LANG` 环境变量为此值。

`NLS_LANG` 格式为“语言\_国家. 字符集”（例如 `AMERICAN_AMERICA.AL32UTF8`）。这必须与您的数据库编码匹配。当此值未设置时，如果 `oracle_fdw` 可以自动完成正确的操作，它将自动执行操作，并在无法执行时发出警告。仅在您知道自己在做什么时才设置此值。请参阅“问题”部分。

### H. 55. 3. 2. 外部服务器选项

- dbserver (必填)

远程数据库的 Oracle 数据库连接字符串。

这可以以Oracle支持的任何形式进行，只要您的Oracle客户端进行了相应的配置。

将此设置为空字符串以进行本地 (“BEQUEATH”) 连接。

- isolation\_level (可选，默认为serializable)

在Oracle数据库中要使用的事务隔离级别。

该值可以为serializable, read\_committed或read\_only 。

请注意，Oracle表可以在单个LightDB语句期间进行多次查询（例如，在嵌套循环连接期间）。为确保不会发生由并发事务的竞争条件引起的不一致，事务隔离级别必须保证读取稳定性。

仅使用Oracle的SERIALIZABLE或READ ONLY隔离级别才能保证这一点。

不幸的是，Oracle的SERIALIZABLE实现相当差，会在意外情况下（例如插入表中）导致序列化错误（ORA-08177）。

使用READ COMMITTED事务可以解决此问题，但是存在不一致的风险。如果要使用它，请检查您的执行计划，以确定外部扫描是否可以执行多次。

- nchar (布尔型，可选，默认为关闭)

将此选项设置为on 将在Oracle端选择更昂贵的字符转换。如果您正在使用单字节Oracle数据库字符集，但具有包含不能在数据库字符集中表示的字符的NCHAR或NVARCHAR2列，则需要此选项。

将nchar设置为on 将对性能产生明显影响，并导致UPDATE语句中设置超过2000个字节的字符串时出现ORA-01461错误（如果您的MAX\_STRING\_SIZE = EXTENDED，则为16383）。此错误似乎是Oracle的一个bug。

### H. 55. 3. 3. 用户映射选项

- user (必填)

会话的Oracle用户名。

如果您不想将Oracle凭据存储在LightDB数据库中（一种简单的方法是使用外部密码存储库），则将其设置为空字符串，以进行外部身份验证。

- password (必填)

Oracle用户的密码。

### H. 55. 3. 4. 外部表选项

- table (必填)

Oracle表名。此名称必须与Oracle系统目录中出现的名称完全一致，因此通常只由大写字母组成。

要基于任意Oracle查询定义外部表，请将此选项设置为括在括号中的查询，例如

```
OPTIONS (table ' (SELECT col FROM tab WHERE val =
''string''))')
```

在这种情况下不要设置schema选项。

如果在简单查询上定义了外部表，则INSERT、UPDATE和DELETE将能够正常工作；如果要避免这种情况（或者在复杂查询的情况下避免混淆Oracle错误消息），请使用表选项readonly。

- dblink（可选）

通过Oracle数据库连接访问表。此名称必须与Oracle系统目录中出现的名称完全一致，因此通常只由大写字母组成。

- schema（可选）

表的模式（或所有者）。有用于访问不属于连接Oracle用户的表。此名称必须与Oracle系统目录中出现的名称完全一致，因此通常只由大写字母组成。

- max\_long（可选，默认为“32767”）

Oracle表中任何LONG、LONG RAW和XMLTYPE列的最大长度。可能的值为1到1073741823之间的整数（LightDB中bytea 的最大大小）。这个内存量至少会分配两次，所以大的值将会消耗大量的内存。

如果max\_long小于检索到的最长值的长度，则会收到错误消息ORA-01406: fetched column value was truncated。

- readonly（可选，默认为“false”）

只有在该选项设置为no/off/false的表上才允许进行INSERT、UPDATE和DELETE操作。

- sample\_percent（可选，默认为“100”）

该选项仅影响ANALYZE处理过程，并可用于在合理的时间内分析非常大的表。

该值必须在0.000001和100之间，并定义了将随机选择计算LightDB表统计信息的Oracle表的百分比。这是通过在Oracle中使用SAMPLE BLOCK (x)子句来实现的。

对于使用Oracle查询定义的表，ANALYZE将失败并显示ORA-00933错误，对于使用复杂的Oracle视图定义的表，ANALYZE可能会失败并显示ORA-01446错误。

- prefetch（可选，默认为“200”）

设置在LightDB与Oracle之间进行外部表扫描期间单次往返获取的行数。这是使用Oracle行预取实现的。该值必须在0和10240之间，其中值为0表示禁用预取。

更高的值可以提高性能，但会在LightDB服务器上使用更多的内存。

## H. 55. 4. 用法

### H. 55. 4. 1. Oracle权限

Oracle用户显然需要CREATE SESSION权限以及选择相关表或视图的权限。

对于EXPLAIN VERBOSE，用户还需要在V\$SQL和V\$SQL\_PLAN上拥有SELECT权限。

### H. 55. 4. 2. 连接

oracle\_fdw会缓存Oracle连接，因为为每个单独的查询创建Oracle会话的成本较高。当LightDB会话结束时，所有连接都会自动关闭。

函数`oracle_close_connections()` 可用于关闭所有缓存的Oracle连接。这对于长时间运行的会话非常有用，因为它们不会一直访问外部表，并且希望避免阻塞打开的Oracle连接所需的资源。

不能在修改Oracle数据的事务中调用此函数。

### H. 55. 4. 3. 列

当您定义外部表时，Oracle表的列按其定义顺序映射到LightDB列。

`oracle_fdw`只会在LightDB查询中实际需要的情况下在Oracle查询中包含这些列。

LightDB表可以比Oracle表具有更多或更少的列。如果它有更多的列，并且这些列被使用，您将收到一个警告并返回NULL值。

如果要进行UPDATE或DELETE操作，请确保所有属于表主键的列的`key`选项都已设置。否则将导致错误。

### H. 55. 4. 4. Oracle权限

必须使用`oracle_fdw`可以转换的数据类型来定义LightDB列（请参见下面的转换表）。只有在实际使用列时才强制执行此限制，因此您可以为无法转换的数据类型定义“虚拟”列，只要您不访问它们即可（此技巧仅适用于SELECT，而不是在修改外部数据时）。如果Oracle值超出LightDB列的大小（例如，`varchar`列的长度或最大整数值），则会收到运行时错误。

这些转换由`oracle_fdw`自动处理：

Oracle type	Possible LightDB types
CHAR	char, varchar, text
NCHAR	char, varchar, text
VARCHAR	char, varchar, text
VARCHAR2	char, varchar, text, json
NVARCHAR2	char, varchar, text
CLOB	char, varchar, text, json
LONG	char, varchar, text
RAW	uuid, bytea
BLOB	bytea
BFILE	bytea (read-only)
LONG RAW	bytea
NUMBER	numeric, float4, float8, char,
varchar, text	
NUMBER(n, m)	numeric, float4, float8, int2, int4, int8,
	boolean, char, varchar, text
FLOAT	numeric, float4, float8, char,
varchar, text	
BINARY_FLOAT	numeric, float4, float8, char,
varchar, text	
BINARY_DOUBLE	numeric, float4, float8, char,
varchar, text	
DATE	date, timestamp, timestamptz, char,
varchar, text	
TIMESTAMP	date, timestamp, timestamptz, char,
varchar, text	
TIMESTAMP WITH TIME ZONE	date, timestamp, timestamptz, char,
varchar, text	



varchar, text	TIMESTAMP WITH LOCAL TIME ZONE	date, timestamp, timestamptz, char,
	INTERVAL YEAR TO MONTH	interval, char, varchar, text
	INTERVAL DAY TO SECOND	interval, char, varchar, text
	XMLTYPE	xml, char, varchar, text
	MDSYS.SDO_GEOMETRY	geometry (see "PostGIS support"

below)

如果将NUMBER转换为布尔值，则0表示false，其他所有值表示true。

插入或更新XMLTYPE只适用于不超过VARCHAR2数据类型最大长度（4000或32767，取决于MAX\_STRING\_SIZE参数）的值。

目前不支持NCLOB，因为Oracle无法自动将其转换为客户端编码。

如果需要超出上述转换的转换，请在Oracle或LightDB中定义适当的视图。

#### H. 55. 4. 5. WHERE条件和ORDER BY子句

LightDB将使用WHERE子句的所有适用部分作为扫描的过滤器。每当这样的条件可以安全地转换为Oracle SQL时，Oracle\_fdw构造的Oracle查询将包含与这些过滤条件相对应的WHERE子句。这个功能也被称为WHERE子句的“下推”，可以大大减少从Oracle检索的行数，可能使Oracle的优化器选择访问所需表的良好计划。

同样地，ORDER BY子句将在可能的情况下下推到Oracle。请注意，不会将按字符串排序的ORDER BY条件下推，因为LightDB和Oracle中的排序顺序不能保证相同。

为了利用这一点，请尽量使用简单的条件来为外部表进行选择。选择与Oracle类型相对应的LightDB列数据类型，否则条件无法转换。

表达式now(), transaction\_timestamp(), current\_timestamp, current\_date和localtimestamp 会被正确转换。

EXPLAIN的输出将显示所使用的Oracle查询，因此您可以看到哪些条件被翻译为Oracle以及如何翻译。

#### H. 55. 4. 6. 外部表之间的连接

oracle\_fdw可以将连接下推到Oracle服务器，也就是两个外部表之间的连接将导致在Oracle端执行连接的单个查询。

在下推连接时，有一些限制：

- 两个表必须在相同的外部服务器上定义。
- 三个或更多表之间的连接不会被下推。
- 连接必须在SELECT语句中。
- oracle\_fdw必须能够将所有连接条件和WHERE子句下推。
- 没有连接条件的交叉连接不会被下推。
- 如果连接被下推，则ORDER BY子句不会被下推。

很重要的是，使用ANALYZE收集了两个外部表的表统计信息，LightDB才能确定最佳的连接策略。

## H. 55. 4. 7. 修改外部数据

oracle\_fdw支持对外部表进行INSERT、UPDATE和DELETE操作。默认情况下（包括从早期的LightDB版本升级的数据库中）允许此操作，并且可以通过设置readonly表选项来禁用。

对于UPDATE和DELETE操作，与Oracle表的主键列对应的列必须设置key列选项。这些列用于标识外部表中的一行，因此请确保在属于主键的所有列上设置了该选项。

如果在INSERT操作中省略了外部表列，那么该列将设置为在LightDB外部表上定义的DEFAULT子句的值（如果没有DEFAULT子句，则为NULL）。不使用相应的Oracle列上的DEFAULT子句。如果LightDB外部表不包括Oracle表的所有列，则Oracle DEFAULT子句将用于未包含在外部表定义中的列。

INSERT、UPDATE和DELETE操作中的RETURNING子句被支持，但对于具有Oracle数据类型LONG和LONG RAW的列除外（Oracle不支持在RETURNING子句中使用这些数据类型）。

支持外部表上的触发器。使用AFTER和FOR EACH ROW定义的触发器要求外部表没有具有Oracle数据类型LONG或LONG RAW的列。这是因为这样的触发器利用了上面提到的RETURNING子句。

虽然修改外部数据是可行的，但在许多行受到影响时，性能并不是特别好，这是由于（由于外部数据包装器的工作方式）每一行都必须单独处理。

事务将被转发到Oracle，因此BEGIN、COMMIT、ROLLBACK和SAVEPOINT按预期工作。不支持涉及Oracle的预处理语句。有关详细信息，请参见内部部分。

由于oracle\_fdw默认使用序列化事务，因此修改数据语句可能导致序列化失败：

```
ORA-08177: can't serialize access for this transaction
```

如果并发事务修改了表，并且在长时间运行的事务中更有可能发生此类情况。这些错误可以通过它们的SQLSTATE（40001）进行识别。使用oracle\_fdw的应用程序应该重试以此错误失败的事务。

可以使用不同的事务隔离级别，有关讨论，请参见Foreign server options。

## H. 55. 4. 8. EXPLAIN

LightDB的EXPLAIN命令将显示实际发送给Oracle的查询语句。使用EXPLAIN VERBOSE命令将显示Oracle的执行计划（不适用于Oracle服务器9i或更早版本，请参阅Problems）。

## H. 55. 4. 9. ANALYZE

您可以使用ANALYZE命令收集外部表的统计信息。这一功能由oracle\_fdw支持。

如果没有统计信息，LightDB无法估计对外部表的查询结果行数，这可能导致选择错误的执行计划。

与普通表不同，LightDB不会自动通过autovacuum守护进程来收集外部表的统计信息，因此在创建外部表之后，以及远程表发生重大变化时，运行ANALYZE命令收集外部表的统计信息尤为重要。

请注意，对Oracle外部表进行分析将导致完整的顺序表扫描。您可以使用表选项sample\_percent，仅使用Oracle表的部分抽样数据以加快速度。

## H. 55. 4. 10. PostGIS支持

当安装了PostGIS时，数据类型geometry才可用。

支持的几何类型包括二维和三维的：POINT、LINE、POLYGON、MULTIPOINT、MULTILINE和MULTIPOLYGON。不支持空的PostGIS几何类型，因为它们在Oracle Spatial中没有等价物。

Oracle SRID的NULL值将转换为0，反之亦然。对于Oracle SRID和PostGIS SRID之间的其他转换，请在LightDB的share 目录中创建一个文件srid.map。该文件的每一行应包含一个Oracle SRID和相应的PostGIS SRID，两者之间用空格分隔。为了获得良好的性能，请保持文件较小。

## H. 55. 4. 11. IMPORT FOREIGN SCHEMA的支持

支持使用IMPORT FOREIGN SCHEMA命令批量导入Oracle模式中所有表的表定义。

除了IMPORT FOREIGN SCHEMA的文档之外，还请考虑以下内容：

- IMPORT FOREIGN SCHEMA会为在ALL\_TAB\_COLUMNS中找到的所有对象创建外部表，包括表、视图和物化视图，但不包括同义词。

- 下列是IMPORT FOREIGN SCHEMA支持的选项：

- case: 控制导入期间表和列名的大小写折叠。

可能的取值包括：

- keep: 保持与Oracle中相同的名称，通常为大写。
- lower: 将所有表和列名转换为小写。
- smart: 仅翻译在Oracle中全部为大写字母的名称（这是默认设置）。

- collation: 用于case选项的lower和 smart的大小写折叠的排序规则。

默认值为default，即使用数据库的默认排序规则。仅支持pg\_catalog 模式中的排序规则。请参阅pg\_collation目录中的collname值，获取可能的取值列表。

- dblink: 访问模式的Oracle数据库链接。

此名称必须与Oracle系统目录中的名称完全相同，通常只包含大写字母。

- readonly: 在所有导入表上设置readonly选项。

有关详细信息，请参阅选项部分。

- max\_long: 在所有导入表上设置max\_long选项。

有关详细信息，请参阅选项部分。

- sample\_percent: 在所有导入表上设置sample\_percent选项。

详见选项部分以获取更多细节。

- prefetch: 对所有导入的表设置 prefetch 选项。

详见选项部分以获取更多细节。

- Oracle 模式名称必须与 Oracle 中完全一致，通常为大写。由于 LightDB 在处理之前将名称转换为小写，因此必须使用双引号保护模式名称（例如“SCOTT”）。

- 在限制（LIMIT TO）或排除（EXCEPT）子句中的表名必须按照上述大小写转换后在 LightDB 中的表现方式进行编写。

请注意，IMPORT FOREIGN SCHEMA 与 Oracle server 8i 不兼容；详见问题部分以获取更多细节。

## H. 55. 5. 内部结构

oracle\_fdw 将 Oracle 会话的 MODULE 设置为 LightDB, ACTION 设置为后端进程号。这有助于识别 Oracle 会话, 并允许您使用 DBMS\_MONITOR.SERV\_MOD\_ACT\_TRACE\_ENABLE 跟踪它。

oracle\_fdw 使用 Oracle 的结果预取以避免不必要的客户端和服务端之间的往返。预取行数可以使用 prefetch 表选项进行配置, 默认设置为 200。

oracle\_fdw 不使用 PLAN\_TABLE 来解释 Oracle 查询 (这需要在 Oracle 数据库中创建这样的表), 而是使用库高速缓存中存储的执行计划。为此, 需要明确地描述一个 Oracle 查询, 这将迫使 Oracle 解析该查询。难点在于在 V\$SQL 中查找语句的 SQL\_ID 和 CHILD\_NUMBER, 因为 SQL\_TEXT 列仅包含查询的前 1000 个字节。因此, oracle\_fdw 向查询中添加了一个包含查询文本的 MD5 哈希值的注释。这用于在 V\$SQL 中搜索。实际的执行计划或成本信息从 V\$SQL\_PLAN 中检索。

oracle\_fdw 在 Oracle 端使用事务隔离级别 SERIALIZABLE, 这对应于 LightDB 的 REPEATABLE READ。这是必要的, 因为单个 LightDB 语句可能导致多个 Oracle 查询 (例如, 在嵌套循环连接期间), 并且结果需要保持一致。

不幸的是, Oracle 实现的 SERIALIZABLE 具有某些怪癖; 详见问题部分以获取更多细节。

在本地事务提交之前, Oracle 事务立即提交, 以便完成的 LightDB 事务保证已完成 Oracle 事务。然而, 即使 Oracle 事务已提交, 仍存在 LightDB 事务无法完成的小概率情况。如果没有使用两阶段事务和事务管理器, 这是无法避免的, 这已经超出了外部数据包装器可以合理提供的范围。

由于相同的原因, 不支持涉及 Oracle 的预处理语句。

## H. 55. 6. 存在的问题

### H. 55. 6. 1. 编码

存储在 Oracle 数据库中的字符, 如果无法转换为 LightDB 数据库编码, Oracle 将默默地替换为 替换字符, 通常是普通或反向问号。您将不会收到警告或错误消息。

如果您使用 Oracle 不知道的 LightDB 数据库编码 (目前有 EUC\_CN、EUC\_KR、LATIN10、MULE\_INTERNAL、WIN874 和 SQL\_ASCII), 则无法正确地转换非 ASCII 字符。在这种情况下, 您将收到警告, 并且字符将被替换为上述替换字符。

您可以设置外部数据包装器的 nls\_lang 选项以强制使用某个 Oracle 编码, 但是生成的字符很可能是不正确的, 并会导致 LightDB 错误消息。如果您知道自己在做什么, 则仅对 SQL\_ASCII 编码有用。请参见选项部分。

### H. 55. 6. 2. 旧版 Oracle 中的功能受限

- Oracle 系统目录 V\$SQL 和 V\$SQL\_PLAN 的定义已经随 Oracle 10.1 更改。在旧版本的 Oracle 服务器上使用 EXPLAIN VERBOSE 将导致出现错误, 例如:

```
ERROR: error describing query: OCISmtExecute failed
to execute
remote query for sql_id
DETAIL: ORA-00904: "LAST_ACTIVE_TIME": invalid
identifier
```

目前没有修复这个问题的计划, 因为自 2010 年以来, Oracle 9i 已经停止了扩展支持, 而且这种功能也不是必要的。

- IMPORT FOREIGN SCHEMA 在 Oracle server 8i 中抛出以下错误:

```

failed to execute          ERROR:  error importing foreign schema: OCISstmtExecute
                           column query
                           DETAIL:  ORA-00904: invalid column name
    
```

这是因为 ALL\_TAB\_COLUMNS 视图缺少列 CHAR\_LENGTH，该列在 Oracle 9i 中添加。

### H. 55.6.3. 序列化错误

在 Oracle 11.2 或更高版本中，使用 oracle\_fdw 向新创建的 Oracle 表插入第一行会导致序列化错误。

这是因为 Oracle 具有一个名为延迟 段创建 的功能，它推迟为新表分配存储空间，直到插入第一行。这会导致可串行化事务出现序列化故障（请参见 Oracle 知识库中的文档 1285464.1）。

这不是一个严重的问题；您可以通过忽略第一个错误或使用 SEGMENT CREATION IMMEDIATE 创建表来解决它。

一种更加严重的问题是，当索引页面与修改可串行化事务同时发生并发插入时，有时可能会导致序列化错误（请参见 Oracle 文档 160593.1）。

Oracle 声称这不是一个 bug，建议的解决方案是重试遇到序列化错误的事务。

### H. 55.6.4. Oracle Bug

这是过去影响 oracle\_fdw 的 Oracle Bug 列表。

Bug 2728408 可能会导致 ORA-8177 cannot serialize access for this transaction，即使没有尝试修改远程数据。

它可能会发生在 Oracle server 8.1.7.4（安装 one-off patch 2728408）或 Oracle server 9.2（安装 Patch Set 9.2.0.4 或更高版本）。

## H. 56. orafce

orafce - Oracle 的兼容性函数和包。

### H. 56.1. 与 Oracle 数据库兼容的特性

提供了与 Oracle 数据库兼容的特性。这些特性使您能够轻松迁移到 LightDB 并减少重新配置应用程序的成本。下表列出了与 Oracle 数据库兼容的特性。

Table H. 85. 数据类型

项目	概述
VARCHAR2	可变长度字符数据类型
NVARCHAR2	可变长度国家字符数据类型
DATE	存储日期和时间的数据类型
NUMBER	NUMBER 是 NUMERIC 的别名
CLOB	存储单字节和多字节字符数据，并且和 `text` 数据类型有基本一致的功能
BLOB	存储无结构的二进制大对象，并且和 `bytea` 数据类型有基本一致的功能

项目	概述
RAW	可变长度十六进制数据类型。
PLS_INTEGER	用于存储有符号整数的 PL/SQL 数据类型。
BINARY_FLOAT	32位浮点数。这种数据类型需要4个字节。
BINARY_DOUBLE	64位浮点数。这种数据类型需要8个字节。
XMLType	可变长度xml数据类型。

在 varchar 和 char 之间, varchar2 和 nvarchar2 的优先级配置, 创建 ora9ce 扩展后, 优先级如下:

numeric > double precision > real > bigint > integer > smallint > text > varchar > varchar2 > nvarchar2 > char

Table H. 86. SQL 查询

项目	概述
DUAL 表	系统提供的表

Table H. 87. 数学函数

项目	概述	并行属性
BIN_TO_NUM	将位向量转换为相应的数字	安全
BITAND	执行按位 AND 操作	安全
BITOR	执行按位 OR 操作	安全
BITXOR	执行按位 XOR 操作	安全
COSH	计算一个数的双曲余弦值	安全
MOD	MOD函数返回参数1除以参数2的余数(模)	安全
REMAINDER	返回 n2 除以 n1 的余数	安全
ROUND_TIES_TO_EVEN	将 n 四舍五入到整数位	安全
SINH	计算一个数的双曲正弦值	安全
TANH	计算一个数的双曲正切值	安全
CEIL	计算大于等于一个数的最小整数, 当输入参数是 interval 类型时, 会先获取其天数并将其转成数值后再计算	安全

Table H. 88. 字符串函数

项目	概述	并行属性
INSTR	返回字符串中一个子字符串的位置	安全
INSTRB	返回字符串中第一个字节为指定子字符串的位置。	安全
LENGTH	返回字符串的字符数长度	安全
LENGTHB	返回字符串的字节数长度	安全

项目	概述	并行属性
LPAD	在字符串左侧填充指定长度的字符序列	安全
LTRIM	从字符串的开头删除指定的字符	安全
NLSSORT	返回用于根据语言环境对字符串进行语言排序的字节字符串	安全
REGEXP_COUNT	搜索字符串中的正则表达式，并返回匹配项的计数	安全
REGEXP_INSTR	返回字符串中匹配模式的起始或结束位置。	安全
REGEXP_LIKE	在查询的WHERE子句中使用，使查询返回与给定模式匹配的行。	安全
REGEXP_SUBSTR	返回与函数调用中指定的模式匹配的字符串。	安全
REGEXP_REPLACE	返回与函数调用中指定的模式匹配的字符串。	安全
REPLACE	将string中每个search_string替换为replacement_string并返回结果字符串。	安全
RPAD	用指定的字符序列将字符串右侧填充到指定长度。	安全
RTRIM	从字符串末尾删除指定的字符。	安全
SOUNDEX	返回一个包含char的语音表示的字符字符串。	安全
SUBSTR	使用字符指定位置和长度，提取字符串的一部分。	安全
SUBSTRB	使用字节指定位置和长度，提取字符串的一部分。	安全
GUID	返回UUID。	安全
CONCAT	连接两个字符串	安全
UPPER	大写所有字母	Safe

Table H. 89. Date/Time Functions

项目	概述	并行属性
ADD_MONTHS	给定日期加上指定的月份。	安全
DBTIMEZONE	返回数据库时区的值。	安全
LENGTHB	返回一个字符串的字节数。	安全
LAST_DAY	返回指定日期所在月份的最后一天。	安全
LTRIM	从字符串的开头删除指定的字符。	安全
MONTHS_BETWEEN	返回两个日期之间的月份数。	安全

项目	概述	并行属性
NEXT_DAY	返回指定日期之后第一个特定星期几的日期。	安全
ROUND	四舍五入一个日期。	安全
SESSIONTIMEZONE	返回会话的时区。	安全
SYSDATE	返回系统日期。	安全
TRUNC	截断一个日期。	安全
TZ_OFFSET	返回时区偏移量。	安全

Table H. 90. 数据类型格式化函数

项目	概述	并行安全
CONVERT	将一个字符集的字符字符串转换为另一个字符集。	安全
NUMTOYMINTERVAL	将数字转换为一个 INTERVAL YEAR TO MONTH 字面量。	安全
TO_BLOB(raw)	将 RAW 值转换为 BLOB 值。	安全
TO_CHAR	将一个值转换为一个字符串。	安全
TO_DATE	将字符串按照指定的格式转换为日期。	安全
TO_MULTI_BYTE	将单字节字符串转换为多字节字符串。	安全
TO_NUMBER	将一个值按照指定的格式转换为一个数字。	安全
TO_SINGLE_BYTE	将多字节字符串转换为单字节字符串。	安全
TO_CLOB	将字符字符串或数值转换为 CLOB 值。	

Table H. 91. 条件表达式

项目	概述	并行属性
DECODE	比较值，如果相等，则返回相应的值。	安全
LNNVL	判断一个值是否为 false 或未知。	安全
NANVL	当值为非数字 (NaN) 时，返回替代值。	安全
NVL	当值为 NULL 时，返回替代值。	安全
NVL2	根据值是否为 NULL，返回替代值。	安全

Table H. 92. 聚合函数

项目	概述	并行属性
ANY_VALUE	返回 expr 的一个单一的非确定性值。	不安全



项目	概述	并行属性
BIT_AND_AGG	返回按位 AND 操作的结果。	不安全
BIT_OR_AGG	返回按位 OR 操作的结果。	不安全
BIT_XOR_AGG	返回按位 XOR 操作的结果。	不安全
KURTOSIS_POP	用于确定给定分布中的异常值特征。	不安全
KURTOSIS_SAMP	用于确定给定分布中的异常值特征。	不安全
LISTAGG	返回连接的、以分隔符分隔的字符串值列表。	不安全
MEDIAN	计算一组值的中位数。	不安全
SKEWNESS_POP	用于确定给定分布的对称性。	不安全
SKEWNESS_SAMP	用于确定给定分布的对称性。	不安全

Table H. 93. 返回内部信息的函数

项目	概述	并行属性
DUMP	返回一个值的内部信息。	安全
NLS_CHARSET_ID	返回与字符集名称字符串对应的字符集ID编号。	安全
NLS_CHARSET_NAME	返回与ID编号对应的字符集名称。	安全
SYS_CONTEXT	返回当前时刻与上下文命名空间相关联的参数值。	安全
USERENV	返回当前会话中与上下文“USERENV”相关联的参数值。	安全

Table H. 94. SQL运算符

项目	概述
日期时间运算符	用于 DATE 类型的日期时间运算符。

Table H. 95. 其他函数

项目	概述	并行属性
EMPTY_CLOB	返回一个空的 CLOB	安全
EMPTY_BLOB	返回一个空的 BLOB	安全
ORA_HASH	ORA_HASH 是一个计算给定表达式的哈希值的函数。	安全
VSIZE	返回表达式的内部表示中的字节数。	安全
DEPS_SAVE_AND_DROP_DEPENDENCIES	保存并删除表的依赖项。	不安全
DEPS_RESTORE_DEPENDENCIES	恢复表的依赖项。	不安全
PLVSTR.IS_PREFIX	如果 a 是 b 的前缀，则返回 true。	安全
PLVDATE.ISLEAPYEAR	判断某个年份是否是闰年。	

Table H. 96. 包

项目	概述
DBMS_ALERT	向多个会话发送警报。
DBMS_ASSERT	验证输入值的属性。
DBMS_LOB	与 DBMS_LOB 包兼容的一些函数。
DBMS_OUTPUT	向客户端发送消息。
DBMS_PIPE	为会话间通信创建一个管道。
DBMS_RANDOM	生成随机数。
DBMS_UTILITY	提供各种实用工具。
DBMS_JOB	在 <code>lt_cron</code> 中安排和管理作业。
DBMS_LOCK	提供与锁管理服务交互的接口。
DBMS_METADATA	提供了一种从数据库字典中检索元数据的方法。
DBMS_OBFUSCATION_TOOLKIT	允许应用程序使用数据加密标准 (DES) 或三重DES算法加密数据。
DBMS_SNAPSHOT	允许您刷新快照 (MVIEW)。
UTL_FILE	允许进行文本文件操作。
UTL_RAW	提供操作 RAW 数据类型的 SQL 函数。
UTL_URL	包含两个函数，提供 URL 字符的转义和反转义机制。
UTL_ENCODE	提供将 RAW 数据编码为标准编码格式的函数。

Table H. 97. 系统视图

项	概述
DBA_SEQUENCES ALL_SEQUENCES USER_SEQUENCES	描述序列。
DBA_SYNONYMS ALL_SYNONYMS USER_SYNONYMS	描述同义词，当前为空。
DBA_TAB_COLS ALL_TAB_COLS USER_TAB_COLS COLS DBA_TAB_COLUMNS ALL_TAB_COLUMNS USER_TAB_COLUMNS	描述表和视图的列。
DBA_TAB_COL_STATISTICS	包含从 "[DBA/ALL/USER]_TAB_COLUMNS" 提取的列统计信息和直方图信息。

项	概述
ALL_TAB_COL_STATISTICS USER_TAB_COL_STATISTICS	
DBA_OBJECTS ALL_OBJECTS USER_OBJECTS OBJ	描述对象。
DBA_CATALOG ALL_CATALOG USER_CATALOG	列出表、视图和序列。
DICTIONARY DICT	包含数据字典表和视图的描述信息。
DBA_DEPENDENCIES ALL_DEPENDENCIES USER_DEPENDENCIES	描述对象之间的依赖关系。
DBA_SOURCE ALL_SOURCE USER_SOURCE	描述存储对象的文本源代码。
DBA_PROCEDURES ALL_PROCEDURES USER_PROCEDURES	列出函数和存储过程。
DBA_TRIGGERS ALL_TRIGGERS USER_TRIGGERS	描述触发器。
DBA_TRIGGER_COLS ALL_TRIGGER_COLS USER_TRIGGER_COLS	描述触发器中列的使用情况。
DBA_TYPES ALL_TYPES USER_TYPES	描述对象类型。
DBA_CONSTRAINTS ALL_CONSTRAINTS USER_CONSTRAINTS	描述约束定义。
DBA_CONS_COLUMNS ALL_CONS_COLUMNS	描述在约束中指定的列。

项	概述
USER_CONS_COLUMNS	
DBA_VIEWS ALL_VIEWS USER_VIEWS	描述视图。
DBA_TABLES ALL_TABLES USER_TABLES TABS DBA_ALL_TABLES ALL_ALL_TABLES USER_ALL_TABLES TAB	描述对象表和关系表
DBA_TAB_STATISTICS ALL_TAB_STATISTICS USER_TAB_STATISTICS	显示表的优化器统计信息
DBA_TAB_COMMENTS ALL_TAB_COMMENTS USER_TAB_COMMENTS	显示表和视图的注释
DBA_COL_COMMENTS ALL_COL_COMMENTS USER_COL_COMMENTS	显示表和视图的列的注释
DBA_TAB_MODIFICATIONS ALL_TAB_MODIFICATIONS USER_TAB_MODIFICATIONS	描述所有表的修改情况
DBA_INDEXES ALL_INDEXES USER_INDEXES IND	描述索引
DBA_INDEX_USAGE ALL_INDEX_USAGE USER_INDEX_USAGE	显示每个索引的累计统计信息。
DBA_IND_COLUMNS ALL_IND_COLUMNS	描述表上索引的列

项	概述
USER_IND_COLUMNS	
DBA_IND_EXPRESSIONS ALL_IND_EXPRESSIONS USER_IND_EXPRESSIONS	描述基于函数的索引的表达式
DBA_IND_STATISTICS ALL_IND_STATISTICS USER_IND_STATISTICS	显示索引的优化器统计信息
DBA_USERS ALL_USERS USER_USERS	描述用户
DBA_ROLES	描述数据库中的所有角色
DBA_ROLE_PRIVS USER_ROLE_PRIVS	描述赋予用户和角色的角色
PRODUCT_COMPONENT_VERSION	包含组件产品的版本和状态信息
PLAN_TABLE	PLAN_TABLE 会自动创建为全局临时表，用于保存所有用户的 EXPLAIN PLAN 语句的输出，目前为空
DBA_DATA_FILES	描述数据库文件
DBA_JOBS ALL_JOBS USER_JOBS	描述作业
DBA_JOBS_RUNNING	列出实例中当前正在运行的所有作业
DBA_MVIEWS ALL_MVIEWS USER_MVIEWS	描述物化视图
DBA_MVIEW_LOGS ALL_MVIEW_LOGS USER_MVIEW_LOGS	描述所有物化视图日志，目前为空
DBA_MVIEW_COMMENTS ALL_MVIEW_COMMENTS USER_MVIEW_COMMENTS	显示物化视图的注释
DBA_TABLESPACES USER_TABLESPACES	描述表空间
NLS_DATABASE_PARAMETERS NLS_INSTANCE_PARAMETERS	列出 NLS 参数

项	概述
NLS_SESSION_PARAMETERS	
DBA_SEGMENTS USER_SEGMENTS	描述分配给段的存储空间
DBA_PART_TABLES ALL_PART_TABLES USER_PART_TABLES	显示分区表的对象级分区信息
DBA_TAB_PARTITIONS ALL_TAB_PARTITIONS USER_TAB_PARTITIONS	显示表分区级别的分区信息、分区存储参数和分区统计信息
DBA_TAB_SUBPARTITIONS ALL_TAB_SUBPARTITIONS USER_TAB_SUBPARTITIONS	显示所有子分区的信息
DBA_PART_KEY_COLUMNS ALL_PART_KEY_COLUMNS USER_PART_KEY_COLUMNS	描述分区对象的分区键列
DBA_SUBPART_KEY_COLUMNS ALL_SUBPART_KEY_COLUMNS USER_SUBPART_KEY_COLUMNS	显示复合分区表(和复合分区表上的本地索引)的子分区键列
DBA_IND_PARTITIONS ALL_IND_PARTITIONS USER_IND_PARTITIONS	描述索引分区
DBA_PART_INDEXES ALL_PART_INDEXES USER_PART_INDEXES	显示分区索引的对象级别分区信息
COL	描述表, 视图的字段信息

以DBA开头的视图显示数据库中的某些内容。以ALL开头的视图目前与以DBA开头的视图相同, 不像Oracle那样进行权限验证。以USER开头的视图显示当前用户拥有的某些内容(实际上是当前模式, 在Oracle中, 用户和模式是一一对应的)。

numeric和boolean类型的隐式转换

Oracle支持numeric和boolean类型的双向隐式转换。

Example

```
CREATE TABLE test(v1 BOOLEAN, v2 NUMERIC);

-- numeric implicit cast to boolean
-- boolean implicit cast to numeric
INSERT INTO test(v1, v2) VALUES(1.1, true);
```

```
SELECT * FROM test WHERE v2 = true;
```

## H. 56. 2. 使用orafce的注意事项

orafce被定义为在创建数据库集群时默认创建的“public”模式中的用户定义函数，因此它们可以对所有用户可用，无需特殊设置。由于这个原因，请确保在search\_path参数中指定的模式搜索路径列表中包含“public”(不带双引号)。

以下由orafce提供的功能是使用不同的外部规范在LightDB和orafce中实现的。在LightDB的默认配置中，LightDB的标准功能优先。

Table H. 98. 数据类型

项目	LightDB的标准功能	orafce添加的兼容性功能
DATE	仅存储日期。	存储日期和时间。

Table H. 99. 函数

项目	LightDB的标准功能	orafce添加的兼容性功能
LENGTH	如果字符串是CHAR类型，则不包括尾随空格长度。	如果字符串是CHAR类型，则包括尾随空格长度。
SUBSTR	如果为开始位置指定了0或负值，则将开始位置减1，位置将向左移动，从那里开始提取。	<ul style="list-style-type: none"> <li>- 如果为开始位置指定0，则从字符串的开头开始提取。</li> <li>- 如果为开始位置指定负值，则从字符串末尾计算的位置开始提取。</li> </ul>
LPAD RPAD	<ul style="list-style-type: none"> <li>- 如果字符串是CHAR类型，则删除尾随空格，然后填充值。</li> <li>- 结果长度被视为字符数。</li> </ul>	<ul style="list-style-type: none"> <li>- 如果字符串是CHAR类型，则在不删除尾随空格的情况下填充值。</li> <li>- 结果长度基于显示字符串的宽度。因此，全角字符使用宽度2处理，半角字符使用宽度1处理。</li> </ul>
LTRIM RTRIM BTRIM (*1)	如果字符串是CHAR类型，则删除尾随空格，然后删除值。	如果字符串是CHAR类型，则在不删除尾随空格的情况下删除值。
TO_DATE	返回值的数据类型为DATE。	返回值的数据类型为TIMESTAMP。

\*1: 对于Oracle数据库，不存在BTRIM函数，但是orafce实现了与TRIM函数相同的外部规范，以对齐其行为。

此外，在LightDB的默认配置中无法使用以下功能。

Table H. 100. 函数

功能
SYSDATE
DBTIMEZONE
SESSIONTIMEZONE

功能
TO_CHAR (日期/时间值)

Table H.101. 运算符

功能
日期时间运算符

要使用这些功能，请在lightdb.conf的“search\_path”参数中设置“oracle”和“pg\_catalog”。在这种情况下，您必须在“pg\_catalog”之前指定“oracle”。

```
search_path = '$user', public, oracle, pg_catalog'
```

#### 信息

- search\_path参数指定搜索模式的顺序。与Oracle数据库兼容的每个功能都在oracle模式中定义。
- 建议在lightdb.conf中设置search\_path。在这种情况下，它将对每个实例有效。
- search\_path的配置可以在用户级别或数据库级别完成。以下是设置示例。
- 如果LightDB的标准功能具有优先级，并且不需要使用默认配置无法使用的功能，则无需更改search\_path的设置。
- 用户级别的设置示例。可以通过执行SQL命令来设置。在这个例子中，user1被用作用户名。

```
ALTER USER user1 SET search_path = '$user', public, oracle, pg_catalog;
```

- 在数据库级别进行设置的示例。可以通过执行SQL命令来设置。在这个例子中，db1被用作数据库名称。您必须在“pg\_catalog”之前指定“oracle”。

```
ALTER DATABASE db1 SET search_path = '$user', public, oracle, pg_catalog;
```

#### 参见

- 有关search\_path的信息，请参考LightDB文档中的“服务器管理” > “客户端连接默认值” > “语句行为”。
- 有关ALTER USER和ALTER DATABASE的信息，请参考LightDB文档中的“参考” > “SQL命令”。

## H.56.3. 数据类型

支持以下数据类型：

- VARCHAR2
- NVARCHAR2
- DATE



- RAW
- PLS\_INTEGER
- BINARY\_FLOAT
- BINARY\_DOUBLE
- XMLType

### H. 56. 3. 1. VARCHAR2

语法

指定VARCHAR2类型如下。

Table H. 102. VARCHAR2语法

数据类型语法	说明
VARCHAR2(len [byte char])	变长字符串，最多可包含len个字符。对于len，请指定大于0的整数。如果省略len，则字符串可以是任意长度。自23.1版本以来，LightDB实现了此语法，以兼容Oracle的varchar2类型。在len后面的关键字byte或char是可选的，仅与Oracle语法兼容，内部实现与只有len而没有byte或char关键字的实现相同。

通用规则

- VARCHAR2是一种字符数据类型。请指定长度所需的字符数。
- 字符串长度可变。指定的值将被存储为原样。此数据类型的上限约为1GB。

注意

VARCHAR2类型不支持排序序列。因此，在需要类似于ORDER BY子句的排序序列时，会发生以下错误。此时，以下提示将提示使用COLLATE子句，但是，由于不支持排序序列，因此无法使用此子句。

```
ERROR: could not determine which collation to use for string comparison
HINT: Use the COLLATE clause to set the collation explicitly.
```

如果显示了上面显示的错误，请将列显式转换为VARCHAR或TEXT类型。

示例

```
lightdb@postgres=# create table t1(name varchar2(10));
CREATE TABLE
lightdb@postgres=#
lightdb@postgres=# \d+ t1
```

Table "public.t1"						
Column	Type	Collation	Nullable	Default	Storage	Stats target
Description						
name	varchar2(10)		NOT NULL			

```

name | varchar2(10) |          |          |          | plain |
|
Access method: heap

lightdb@postgres=#
lightdb@postgres=# create table t2(name varchar2(10 byte));
CREATE TABLE
lightdb@postgres=#
lightdb@postgres=# \d+ t2
                                Table "public.t2"
Column |      Type      | Collation | Nullable | Default | Storage | Stats target
| Description
-----+-----+-----+-----+-----+-----+-----
name   | varchar2(10)   |           |          |          | plain   |
|
Access method: heap

lightdb@postgres=#
lightdb@postgres=# create table t3(name varchar2(10 char));
CREATE TABLE
lightdb@postgres=#
lightdb@postgres=# \d+ t3
                                Table "public.t3"
Column |      Type      | Collation | Nullable | Default | Storage | Stats target
| Description
-----+-----+-----+-----+-----+-----+-----
name   | varchar2(10)   |           |          |          | plain   |
|
Access method: heap

lightdb@postgres=#

```

### H. 56. 3. 2. NVARCHAR2

语法

指定NVARCHAR2类型如下。

Table H. 103. NVARCHAR2语法

数据类型语法	解释
NVARCHAR2(len)	变长国家字符串，最多可包含len个字符。对于len，请指定大于0的整数。如果省略len，则字符串可以是任意长度。

通用规则

- NVARCHAR2是一种国家字符数据类型。请指定长度所需的字符数。
- 字符串长度可变。指定的值将被存储为原样。此数据类型的上限约为1GB。

注意

NVARCHAR2类型不支持排序序列。因此，在需要类似于ORDER BY子句的排序序列时，会发生以下错误。此时，以下提示将提示使用COLLATE子句，但是，由于不支持排序序列，因此无法使用此子句。

ERROR: could not determine which collation to use for string comparison  
 HINT: Use the COLLATE clause to set the collation explicitly.

如果显示了上面显示的错误，请将列显式转换为NCHAR VARYING或TEXT类型。

### H. 56. 3. 3. 日期

语法

指定DATE类型如下。

Table H. 104. 日期语法

数据类型语法	解释
DATE	存储日期和时间

通用规则

- DATE是一种日期/时间数据类型。
- 日期和时间存储在DATE中。时区不会被存储。

注意

如果在DDL语句（如表定义）中使用orafce的DATE类型，请在执行DDL语句之前始终设置search\_path。即使在定义之后更改了search\_path，数据类型也将是LightDB的DATE类型。

信息

orafce的DATE类型等同于LightDB的TIMESTAMP类型。因此，可以使用参数数据类型为TIMESTAMP的现有LightDB函数。

### H. 56. 3. 4. RAW

语法

指定 RAW 类型如下。

Table H. 105. RAW 语法

RAW 类型语法	解释
RAW(len)	存储可变长度的二进制数据。外部表示为十六进制。len 只是为了与 Oracle 兼容，没有实际作用。二进制数据可以是任意长度。

通用规则

- RAW 是一个可变长度的十六进制数据类型。
- 十六进制字符串长度可变。指定的值将被存储为二进制数据。此数据类型的上限约为1GB。

### H. 56. 3. 5. PLS\_INTEGER

语法

指定 PLS\_INTEGER 类型如下。

Table H.106. PLS\_INTEGER 语法

PLS_INTEGER 类型语法	解释
PLS_INTEGER	PLS_INTEGER 数据类型以 32 位表示带符号整数，范围为 -2,147,483,648 到 2,147,483,647。

通用规则

- PLS\_INTEGER 类似于 int4。

### H.56.3.6. BINARY\_FLOAT

语法

指定 BINARY\_FLOAT 类型如下。

Table H.107. BINARY\_FLOAT 语法

BINARY_FLOAT 类型语法	解释
BINARY_FLOAT	BINARY_FLOAT 是一种 32 位单精度浮点数的数据类型。每个 BINARY_FLOAT 值需要 4 个字节。

通用规则

- BINARY\_FLOAT 类似于 float4。
- 与 oracle 相同，都遵循 IEEE754 标准。

例子

```
lightdb@test_o=# create table t1(key1 binary_float);
CREATE TABLE
lightdb@test_o=# \d+ t1
                                Table "public.t1"
 Column |      Type      | Collation | Nullable | Default | Storage | Sta
ts target | Description
-----+-----+-----+-----+-----+-----+-----
 key1   | binary_float   |           |          |         | plain   |
Access method: heap
lightdb@test_o=# insert into t1 values(123456789012345678901234567890123456789);
INSERT 0 1
lightdb@test_o=# insert into t1 values('infinity');
INSERT 0 1
lightdb@test_o=# insert into t1 values('NaN');
INSERT 0 1
lightdb@test_o=# select * from t1;
 key1
-----
1.2345679e+38
Infinity
NaN
(3 rows)
```

### H. 56. 3. 7. BINARY\_DOUBLE

语法

指定 BINARY\_DOUBLE 类型如下。

Table H.108. BINARY\_DOUBLE 语法

BINARY_DOUBLE 类型语法	解释
BINARY_DOUBLE	BINARY_DOUBLE是一种64位双精度浮点数的数据类型。每个BINARY_DOUBLE值需要8个字节。

通用规则

- BINARY\_DOUBLE 类似于 float8。
- 与oracle相同都遵循IEEE754标准。

例子

```
lightdb@test_o=# create table t1(key1 binary_double);
CREATE TABLE
lightdb@test_o=# \d+ t1
                                Table "public.t1"
 Column |      Type      | Collation | Nullable | Default | Storage | Sta
ts target | Description
-----+-----+-----+-----+-----+-----+-----
 key1   | binary_double |           |          |         | plain   |
Access method: heap
lightdb@test_o=# insert into t1 values(123456789012345678901234567890123456789);
INSERT 0 1
lightdb@test_o=# insert into t1 values('infinity');
INSERT 0 1
lightdb@test_o=# insert into t1 values('NaN');
INSERT 0 1
lightdb@test_o=# select * from t1;
          key1
-----
1. 2345678901234568e+38
          Infinity
          NaN
(3 rows)
```

### H. 56. 3. 8. XMLType

语法

按如下方式指定XMLType类型。

Table H.109. XMLType 语法

XMLType 语法	解释
XMLType	XMLType是一个对象类型，可以用来存储XML数据。

通用规则

- XMLType是xml数据类型。
- 支持oracle.XMLType和pg\_catalog.xml之间的隐式转换。

成员函数

- function getClobVal() return CLOB

以文本形式返回XML

例子

```

lightdb@postgres=# create database test_oracle
lightdb_syntax_compatible_type oracle;
CREATE DATABASE

lightdb@postgres=# \c test_oracle
You are now connected to database "test_oracle" as user "lightdb".
compatible type: oracle

lightdb@test_oracle=# select dbms_output.serveroutput(true);
serveroutput
-----

(1 row)

lightdb@test_oracle=# CREATE TABLE xml_table (id NUMBER, xml_column
XMLType );
CREATE TABLE

lightdb@test_oracle=# INSERT INTO xml_table (id, xml_column) VALUES (1,
XMLType('<root><name>John</name></root>'));
INSERT 0 1

lightdb@test_oracle=# DECLARE
lightdb@test_oracle$$ v_clob CLOB;
lightdb@test_oracle$$ BEGIN
lightdb@test_oracle$$ SELECT XMLType.getClobVal(xml_column)
lightdb@test_oracle$$ INTO v_clob
lightdb@test_oracle$$ FROM xml_table
lightdb@test_oracle$$ WHERE id = 1;
lightdb@test_oracle$$ DBMS_OUTPUT.PUT_LINE(' CLOB value: ' || v_clob);
lightdb@test_oracle$$ END;
lightdb@test_oracle$$ /
CLOB value: <root><name>John</name></root>
DO

lightdb@test_oracle=# DECLARE
lightdb@test_oracle$$ xml_data XMLType;
lightdb@test_oracle$$ clob_data CLOB;
lightdb@test_oracle$$ BEGIN
lightdb@test_oracle$$ xml_data := XMLType('<root><name>John Doe</name></
root>');
lightdb@test_oracle$$ clob_data := xml_data.getClobVal();

```

```

lightdb@test_oracle$# DBMS_OUTPUT.PUT_LINE(clob_data);
lightdb@test_oracle$# END;
lightdb@test_oracle$# /
<root><name>John Doe</name></root>
DO

lightdb@test_oracle=# SELECT XMLType.getClobVal(xml_column) from xml_table;
getclobval
-----
<root><name>John</name></root>
(1 row)

```

## H. 56. 4. 查询

支持以下查询：

- DUAL 表

DUAL 表是系统提供的虚拟表。在执行 SQL 时不需要访问基表时使用，例如执行测试以获取结果表达式（如函数和运算符）。

示例

在以下示例中，返回当前系统日期。

```

SELECT CURRENT_DATE "date" FROM DUAL;
      date
-----
2013-05-14
(1 row)

```

## H. 56. 5. SQL 函数参考

### H. 56. 5. 1. 数学函数

支持以下数学函数：

- BIN\_TO\_NUM
- BITAND
- BITOR
- BITXOR
- COSH
- MOD
- REMAINDER
- ROUND\_TIES\_TO\_EVEN
- SINH
- TANH

### H. 56. 5. 1. 1. BIN\_TO\_NUM

#### 描述

BIN\_TO\_NUM 将位向量转换为其等效数字。

#### 语法

```
BIN_TO_NUM(VARIADIC integer []) returns int
```

#### 通用规则

- 此函数的每个参数表示位向量中的一位。
- 此函数以任何整数数据类型或任何可隐式转换为整数的非整数数据类型作为参数。
- BIN\_TO\_NUM 在数据仓库应用程序中很有用，用于使用分组集从物化视图中选择感兴趣的组。

#### 注意

- 如果使用 BIN\_TO\_NUM 进行转换，则需要预先指定 "oracle" 作为 search\_path。

#### 参见

- 有关如何编辑 search\_path 的信息，请参阅 "使用 orafce 的注意事项"。
- 有关如何设置服务器参数的信息，请参阅 LightDB 文档中的 "服务器管理" > "服务器配置" > "设置参数"。

#### 示例

```
SELECT BIN_TO_NUM(1, 0, 1, 0) FROM DUAL;
 bin_to_num
-----
          10
(1 row)
```

### H. 56. 5. 1. 2. BITAND

#### 描述

执行按位 AND 操作。

#### 语法

```
BITAND(int1 integers, int2 integers) returns bigint
```

#### 通用规则

- BITAND 对两个整数的每一位执行 AND 操作，并返回结果。
- 指定整数类型的值。
- 返回值的数据类型为 BIGINT。



示例

在以下示例中，返回数字字面值 5 和 3 的 AND 操作结果。

```
SELECT BITAND(5,3) FROM DUAL;
bitand
-----
      1
(1 row)
```

### H. 56. 5. 1. 3. BITOR

描述

执行按位 OR 操作。

语法

BITOR(int1 bigint, int2 bigint) returns bigint

通用规则

- BITOR 对两个 BIGINT 的每一位执行 OR 操作，并返回结果。
- 指定 BIGINT 类型的值。
- 返回值的数据类型为 BIGINT。

注意

- 如果使用 BITOR，则需要预先指定 "oracle" 作为 search\_path。

参见

- 有关如何编辑 search\_path 的信息，请参阅 "使用 orafce 的注意事项"。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的 "服务器管理" > "服务器配置" > "设置参数"。

示例

在以下示例中，返回数字字面值 5 和 3 的 OR 操作结果。

```
SELECT BITOR(5,3) FROM DUAL;
bitor
-----
      7
(1 row)
```

### H. 56. 5. 1. 4. BITXOR

描述

执行按位异或操作。

语法

BITXOR(int1 bigint, int2 bigint) returns bigint

通用规则

- BITXOR 对两个整数的每一位执行 XOR 操作，并返回结果。
- 指定 BIGINT 类型的值。
- 返回值的数据类型为 BIGINT。

注意

- 如果使用 BITXOR，则需要预先指定 "oracle" 作为 search\_path。

参见

- 有关如何编辑 search\_path 的信息，请参阅 "使用 oracle 的注意事项"。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的 "服务器管理" > "服务器配置" > "设置参数"。

示例

在以下示例中，返回数字字面值 5 和 3 的 XOR 操作结果。

```
SELECT BITXOR(5,3) FROM DUAL;  
bitxor  
-----  
      6  
(1 row)
```

### H. 56. 5. 1. 5. COSH

描述

计算一个数字的双曲余弦。

语法

COSH(num numeric) returns double precision

通用规则

- COSH 返回指定数字的双曲余弦值。
- 数字必须是数值数据类型。
- 返回值的数据类型是 DOUBLE PRECISION。

示例

在以下示例中，返回数字字面值 2.236 的双曲余弦值。

```
SELECT COSH(2.236) FROM DUAL;
```

cosh

```
-----
4.7313591000247
(1 row)
```

#### H. 56. 5. 1. 6. MOD

描述

MOD返回n2除以n1的余数。如果n1为0，则返回n2。

语法

```
MOD(n2 bigint, n1 bigint) returns bigint
MOD(n2 integer, n1 integer) returns integer
MOD(n2 numeric, n1 numeric) returns numeric
MOD(n2 smallint, n1 smallint) returns smallint
```

General rules

- 该函数接受数字作为参数

注意

- mod 函数不支持一个参数是字符串形式的小数，另一个参数是整数的情况。

示例

```
SELECT mod(5, 2) FROM DUAL;
mod
-----
1
```

#### H. 56. 5. 1. 7. REMAINDER

描述

返回n2除以n1的余数。

语法

```
REMAINDER(n2 numeric, n1 numeric) returns numeric
```

通用规则

- 此函数将任何数字数据类型或可隐式转换为数字数据类型的任何非数字数据类型作为参数。
- MOD函数类似于REMAINDER函数，但其公式中使用FLOOR函数，而REMAINDER使用ROUND函数。
- 如果n1 ≠ 0，则余数为n2 - (n1 \* N)，其中N是最接近n2 / n1的整数。如果n2 / n1 等于x.5，则N是最接近的偶数整数。

注意

- 如果使用REMAINDER函数，则需要事先在search\_path中指定“oracle”。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 请参阅LightDB文档中“服务器管理”>“服务器配置”>“设置参数”以获取有关如何设置服务器参数的信息。

示例

```
SELECT REMAINDER(5.5, 2) FROM DUAL;
remainder
-----
          -0.5
(1 row)
```

### H. 56. 5. 1. 8. ROUND\_TIES\_TO\_EVEN

描述

该函数将n四舍五入为整数位。

语法

```
ROUND_TIES_TO_EVEN(n NUMERIC, integer int4 DEFAULT 0) returns numeric
```

通用规则

- 如果整数为正，则n将向小数点右侧的整数位四舍五入。
- 如果未指定整数，则n将四舍五入为0位。
- 如果整数为负，则n将向小数点左侧的整数位四舍五入。

注意

- 如果使用ROUND\_TIES\_TO\_EVEN函数，则需要事先在search\_path中指定“oracle”。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 请参阅LightDB文档中“服务器管理”>“服务器配置”>“设置参数”以获取有关如何设置服务器参数的信息。

示例

以下示例将一个数字向右舍入到小数点后一位：

```
SELECT ROUND_TIES_TO_EVEN(45.177, 1) "ROUND_EVEN" FROM DUAL;
ROUND_EVEN
-----
          45.2
(1 row)
```

以下示例将一个数字向左舍入到小数点前一位：

```
SELECT ROUND_TIES_TO_EVEN(45.177, -1) "ROUND_EVEN" FROM DUAL;
ROUND_EVEN
-----
          50
(1 row)
```

#### H. 56. 5. 1. 9. SINH

描述

计算一个数字的双曲正弦。

语法

SINH(num numeric) returns double precision

通用规则

- SINH函数返回指定数字的双曲正弦。
- 数字必须是数字数据类型。
- 返回值的数据类型是DOUBLE PRECISION。

示例

在以下示例中，返回数字1.414的双曲正弦。

```
SELECT SINH(1.414) FROM DUAL;
sinh
-----
1.93460168824956
(1 row)
```

#### H. 56. 5. 1. 10. TANH

描述

计算一个数字的双曲正切。

语法

TANH(num numeric) returns double precision

通用规则

- TANH函数返回指定数字的双曲正切。
- 数字必须是数字数据类型。

- 返回值的数据类型是DOUBLE PRECISION。

示例

在以下示例中，返回数字3的双曲正切。

```
SELECT TANH(3) FROM DUAL;
      tanh
-----
0.995054753686731
(1 row)
```

#### H. 56. 5. 1. 11. CEIL

描述

计算大于等于一个数的最小整数，当输入参数是interval类型时，会先获取其天数并将其转成数值后再计算。

语法

```
CEIL(i interval) returns number
CEIL(n numeric) returns numeric
CEIL(d double precision) returns numeric
```

通用规则

- CEIL函数计算大于等于一个数的最小整数。
- 参数可以是interval类型、numeric类型或double precision类型。
- 返回值的数据类型是numeric或double precision类型。

示例

在以下示例中，返回大于等于一个数的最小整数，当输入参数是interval类型时，先获取其天数，这是个数值，然后再返回大于等于这个数值的最小整数。

```
select ceil(sysdate - '2023-01-01'::date) FROM dual ;
      ceil
-----
      335
(1 row)

select ceil(sysdate - to_date('2023-11-27','yyyy-mm-dd')) FROM dual;
      ceil
-----
        5
(1 row)

select ceil(sysdate - to_date('2023-01-01','yyyy-mm-dd')) FROM dual;
      ceil
-----
      335
(1 row)
```

## H. 56. 5. 2. 字符串函数

支持以下字符串函数：

- BTRIM
- INSTR
- INSTRB
- LENGTH
- LENGTHB
- LPAD
- LTRIM
- NLSSORT
- REGEXP\_COUNT
- REGEXP\_INSTR
- REGEXP\_LIKE
- REGEXP\_SUBSTR
- REGEXP\_REPLACE
- REPLACE
- RPAD
- RTRIM
- SOUNDEX
- SUBSTR
- SUBSTRB
- CONCAT
- UPPER

### H. 56. 5. 2. 1. BTRIM

描述

从字符串的开头和结尾删除指定的字符。

语法

```
BTRIM(str text) returns text  
BTRIM(str text, trimChars text) returns text  
BTRIM(str clob) returns clob  
BTRIM(str clob, trimChars clob) returns clob
```

### 通用规则

- BTRIM 从字符串 `str` 的开头和结尾删除 `trimChars`，并返回结果字符串。
- 如果指定了多个修剪字符，则删除与修剪字符匹配的所有字符。如果省略了 `trimChars`，则删除所有前导和尾随半角空格。
- 返回值的数据类型为文本类型如果第一个参数是 `text` 类型，返回值类型为 `CLOB` 如果第一个参数是 `clob` 类型。

### 注意

- Oracle 数据库不支持 BTRIM 函数。
- BTRIM 函数的 CHAR 类型规范使用 `orafce` 来定义，其行为与 LightDB 的 BTRIM 函数不同。必须修改 `search_path` 参数才能使其与上述规范描述的行为相同。

### 信息

LightDB 的 BTRIM 通用规则如下：

- 如果字符串是 CHAR 类型，则先删除尾随空格，然后删除修剪字符。

### 参见

- 有关如何编辑 `search_path` 的信息，请参阅“使用 `orafce` 的注意事项”。
- 请参阅 LightDB 文档中的“SQL 语言” > “函数和操作符” > “字符串函数和操作符”中的 BTRIM 了解更多信息。

### 示例

在以下示例中，返回从字符串“aabcaba”的开头和结尾删除“a”后的字符串。

```
SELECT BTRIM(' aabcaba', 'a') FROM DUAL;
 btrim
-----
  bcab
(1 row)
```

## H. 56. 5. 2. 2. INSTR

### 描述

返回字符串中子字符串的位置。

### 语法

```
INSTR(str1 text, str2 text) returns integer
INSTR(str1 text, str2 text, startPos integer) returns integer
INSTR(str1 text, str2 text, startPos integer, occurrences integer) returns integer
```

### 通用规则

- INSTR 函数在字符串 `str1` 中搜索子字符串 `str2`，并返回第一次出现的字符位置（以字符计数）。



- 搜索从字符串 `str1` 的指定起始位置 `startPos` 开始。
- 当 `startPos` 为 0 或负数时，起始位置将从 `str1` 结尾左侧的指定字符数开始，`INSTR` 函数将从该点向后搜索。
- 如果没有指定起始位置，则从 `str1` 的开头开始搜索。
- 如果指定了 `occurrences`，则返回 `str2` 的第 `n` 次出现在 `str1` 中的位置。只能指定正数。
- 如果未指定 `occurrences`，则返回找到的第一个出现位置的起始位置。
- 如果在 `str1` 中未找到 `str2`，则返回 0。
- 对于 `startPos` 和 `occurrences`，请指定 `SMALLINT` 或 `INTEGER` 类型。
- 返回值的数据类型为 `INTEGER`。

示例

在以下示例中，字符串 "ABCACBCAAC" 中找到了字符 "BC"，并返回了这些字符的位置。

```

SELECT INSTR('ABCACBCAAC', 'BC') FROM DUAL;
instr
-----
      2
(1 row)

SELECT INSTR('ABCACBCAAC', 'BC', -1, 2) FROM DUAL;
instr
-----
      2
(1 row)

```

### H. 56. 5. 2. 3. INSTRB

描述

`INSTRB` 函数使用字节搜索字符串中的子字符串，并返回第一个指定出现的子字符串字节所在字符串的位置。

语法

```

INSTRB( p_source text, p_search text, p_position int4 default 1,
p_occurrence int4 default 1) RETURNS integer

```

通用规则

- `INSTRB` 函数在字符串 `p_source` 中搜索子字符串 `p_search`，并返回第一次出现的子字符串字节所在字符串的位置（以字节计数）。
- 搜索从字符串 `p_source` 的指定起始位置 `p_position` 开始。
- 当 `p_position` 为 0 或负数时，将引发错误。这与 Oracle 不同，在 Oracle 中，`p_position` 将从 `p_source` 结尾左侧的指定字节数开始，`INSTRB` 函数将从该点向后搜索。
- 如果未指定起始位置，则从 `p_source` 的开头开始搜索。

- 如果指定了 `p_occurrence`，则返回 `p_search` 的第 `n` 次出现在 `p_source` 中的位置。只能指定正数。
- 如果未指定 `occurrences`，则返回找到的第一个出现位置的起始位置。
- 如果在 `p_source` 中未找到 `p_search`，则返回 0。
- 返回值的数据类型为 `INTEGER`。

注意

- 如果使用 `INSTRB` 函数，则需要事先指定 `search_path` 为 `"oracle"`。

请参见

- 有关如何编辑 `search_path` 的信息，请参阅 `"Notes on Using orafce"`。
- 有关如何设置服务器参数的信息，请参阅 LightDB 文档中的 `"Server Administration" > "Server Configuration" > "Setting Parameters"` 部分。

示例

在下面的例子中，找到字符串 `"hhh测试onhx"` 中的字符 `"on"`，并返回那些字节的位置。

```
SELECT INSTRB('hhh测试onhx', 'on') FROM DUAL;
instrb
-----
      10
(1 row)
```

```
SELECT INSTRB('hhh测试onhx on xxxx', 'on', 7, 2) FROM DUAL;
instrb
-----
      15
(1 row)
```

#### H. 56. 5. 2. 4. LENGTH

描述

返回字符串中字符的数量。

语法

```
LENGTH(str text) returns integer
```

通用规则

- `LENGTH` 返回字符串 `str` 中字符的数量。
- 如果字符串是 `CHAR` 类型，尾随空格也会计入长度。
- 返回值的数据类型是 `INTEGER`。

注意

上面的 `LENGTH` 规范使用 `orafce` 来确定其行为，这与 LightDB 的 `LENGTH` 不同。必须修改 `search_path` 参数，以使其按照 `orafce` 规范运行。

信息

LightDB 的 LENGTH 的通用规则如下：

- 如果字符串是 CHAR 类型，尾随空格不计入长度。

参见

- 有关如何编辑 search\_path 的信息，请参阅“使用 orafce 的说明”。
- 有关 LENGTH 的信息，请参阅 LightDB 文档中的“SQL 语言” > “函数和运算符” > “字符串函数和运算符”。

示例

在以下示例中，返回表 t1 中列 col2（使用 CHAR(10) 定义）中的字符数。

```
SELECT col2,LENGTH(col2) FROM t1 WHERE col1 = '1001';
   col2      | length
-----+-----
   AAAAA    |      10
(1 row)
```

#### H. 56. 5. 2. 5. LENGTHB

描述

返回字符串的字节数。

语法

```
LENGTHB(str text) returns integer
```

通用规则

- LENGTHB 返回字符串 str 中的字节数。
- 如果字符串是 CHAR 类型，尾随空格也会计入长度。
- 返回值的数据类型是 INTEGER。

示例

在以下示例中，返回表 t1 中列 col2（使用 CHAR(10) 定义）中的字节数。请注意，在第二个 SELECT 语句中，“\*” 中的每个字符占用 3 个字节，总共占用 9 个字节，并且为 7 个尾随空格添加了 7 个字节。这将得出结果为 16 个字节。

```
SELECT col2,LENGTHB(col2) FROM t1 WHERE col1 = '1001';
   col2      | lengthb
-----+-----
   AAAAA    |      10
(1 row)

SELECT col2,LENGTHB(col2) FROM t1 WHERE col1 = '1004';
   col2      | lengthb
-----+-----
   ***      |      16
```

(1 row)

### H. 56. 5. 2. 6. LPAD

#### 描述

使用一系列的字符填充一个字符串，使其达到指定的长度。

#### 语法

```
LPAD(str text, len integer) returns text
LPAD(str text, len integer, paddingStr text) returns text
LPAD(str CLOB, len integer, str text) returns clob
```

#### 通用规则

- LPAD 函数将在字符串 str 的开始处反复填充填充字符 paddingStr，直到字符串达到长度 len，然后返回结果。
- 如果字符串是 CHAR 类型，则填充字符将添加到字符串中，而不会删除尾随空格。
- 在结果字符串中，全角字符被认为长度为2，半角字符被认为长度为1。如果一个全角字符不能被包含在结果字符串中，因为只有一个半角字符的空间可用，那么该字符串将用单字节空格填充。
- 返回值的数据类型是 TEXT 如果第一个参数类型是 text 类型，CLOB 如果第一个参数类型是 CLOB 类型。

#### 注意

上面的 LPAD 规范使用 orafce 实现，其行为与 LightDB 的 LPAD 不同。必须修改 search\_path 参数才能使其按照 orafce 规范工作。

#### 信息

LightDB 的 LPAD 的通用规则如下：

- 如果字符串是 CHAR 类型，则删除尾随空格，然后将填充字符添加到字符串中。
- 结果长度是字符数。

#### 参见

- 有关如何编辑 search\_path 的信息，请参阅《使用 orafce 的注意事项》。
- 有关 LPAD 的信息，请参阅 LightDB 文档中的“SQL 语言” > “函数和操作符” > “字符串函数和操作符”。

#### 示例

在下面的示例中，返回一个通过使用“a”填充字符串“abc”的左侧而形成的 10 个字符的字符串。

```
SELECT LPAD(' abc', 10, ' a') FROM DUAL;
      lpad
-----
 aaaaaaaabc
(1 row)
```

### H. 56. 5. 2. 7. LTRIM

#### 描述

从字符串的开头删除指定的字符。

#### 语法

```
LTRIM(str text) returns text
LTRIM(str text, trimChars text) returns text
LTRIM(str clob) returns clob
LTRIM(str clob, trimChars clob) returns clob
```

#### 通用规则

- LTRIM 函数返回一个从字符串 str 的开头删除了 trimChars 的字符串。
- 如果指定了多个修剪字符，则删除与修剪字符匹配的所有字符。如果省略了 trimChars，则删除所有前导半角空格。
- 返回值的数据类型是 TEXT 如果第一个参数类型为 `text` 类型，CLOB 如果第一个参数类型为 `clob` 类型。

#### 注意

上面的 LTRIM 规范使用 orafce 实现，其行为与 LightDB 的 LTRIM 不同。必须修改 search\_path 参数才能使其按照 orafce 规范工作。

#### 信息

LightDB的LTRIM的通用规则如下：

- 如果字符串是CHAR类型，则会先去除尾随空格，然后移除修剪字符。

#### 参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关LTRIM的信息，请参阅LightDB文档中“SQL语言” > “函数和操作符” > “字符串函数和操作符”。

#### 示例

在以下示例中，将返回从“aabcab”的开头中移除“ab”的字符串。

```
SELECT LTRIM(' aabcab', 'ab') FROM DUAL;
 ltrim
-----
 cab
(1 row)
```

### H. 56. 5. 2. 8. NLSSORT

#### 描述

返回表示区域设置（COLLATE）的词法顺序的字节字符串。

## 语法

```
NLSSORT(str text) returns bytea
NLSSORT(str text, locale text) returns bytea
```

## 通用规则

- NLSSORT用于在区域设置（COLLATE）的排序顺序中进行比较和排序，该区域设置与默认区域设置不同。
- 可以为区域设置指定的值根据数据库服务器的操作系统而异。
- 如果省略了区域设置，则需要使用set\_nls\_sort预先设置区域设置。要使用set\_nls\_sort设置区域设置，请执行SELECT语句。
- 返回值的数据类型是BYTEA。

使用SELECT语句设置set\_nls\_sort的示例

```
SELECT set_nls_sort('en_US.UTF8');
```

## 注意

如果指定区域编码，请确保它与数据库编码匹配。

## 参见

有关可以指定的语言环境信息，请参阅LightDB文档中的“服务器管理” > “本地化” > “语言环境支持”部分。

## 示例

Table H. 110. 表(t3)的组成

列1	列2
1001	aabcababc
2001	abcdef
3001	aacbaab

在以下示例中，返回按“da\_DK.UTF8”排序表t3的列col2的结果。

```
SELECT col1,col2 FROM t3 ORDER BY NLSSORT(col2,'da_DK.UTF8');
col1 | col2
-----+-----
2001 | abcdef
1001 | aabcababc
3001 | aacbaab
(3 row)
```

### H. 56. 5. 2. 9. REGEXP\_COUNT

#### 描述

在字符串中搜索正则表达式，并返回匹配项的计数。

## 语法

```

REGEXP_COUNT(string text, pattern text) returns integer
REGEXP_COUNT(string text, pattern text, startPos integer) returns integer
REGEXP_COUNT(string text, pattern text, startPos integer, flags text)
returns integer
    
```

## 通用规则

- REGEXP\_COUNT函数返回pattern在源string中出现的次数。它返回一个整数，表示pattern出现的次数。如果没有找到匹配项，则该函数返回0。如果参数中有一个为空，则返回 NULL，但 'flags' 参数可以为空。
- 搜索从string中指定的起始位置startPos开始，默认从string的开头开始。
- startPos是正整数，不允许使用负值从string的末尾开始搜索。
- flags是一个字符表达式，它允许您改变函数的默认匹配行为。

flags的值可以包括以下一个或多个字符：

- 'i'：不区分大小写匹配。
- 'c'：区分大小写和重音敏感匹配。
- 'n'：句点（.）匹配换行符。默认情况下，句点不匹配换行符。
- 'm'：将源字符串视为多行。
- 'x'：忽略空格字符。默认情况下，空格字符与它们自身匹配。

如果省略了flags，那么：

- 默认情况下区分大小写和重音。
- 句点（.）不匹配换行符。
- 源字符串被视为单行。

## 注意

如果使用REGEXP\_COUNT函数，则需要事先在search\_path中指定“oracle”。

## 参见

有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。

## 示例

```

SELECT REGEXP_COUNT(' a' ||CHR(10)||' d', ' a. d') FROM DUAL;
   regexp_count
-----
              0
(1 row)

SELECT REGEXP_COUNT(' a' ||CHR(10)||' d', ' a. d', 1, 'm') FROM DUAL;
   regexp_count
-----
              0
    
```

```
(1 row)

SELECT REGEXP_COUNT('a' || CHR(10) || 'd', 'a.d', 1, 'n') FROM DUAL;
 regexp_count
-----
          1
(1 row)

SELECT REGEXP_COUNT('a' || CHR(10) || 'd', '^d$', 1, 'm') FROM DUAL;
 regexp_count
-----
          1
(1 row)
```

#### H. 56. 5. 2. 10. REGEXP\_INSTR

##### 描述

返回在字符串中找到模式匹配的开始或结束位置。

##### 语法

```
REGEXP_INSTR(string text, pattern text) returns integer
REGEXP_INSTR(string text, pattern text, startPos integer) returns integer
REGEXP_INSTR(string text, pattern text, startPos integer, occurrence
integer) returns integer
REGEXP_INSTR(string text, pattern text, startPos integer, occurrence
integer) returns integer
REGEXP_INSTR(string text, pattern text, startPos integer, occurrence
integer, return_opt integer) returns integer
REGEXP_INSTR(string text, pattern text, startPos integer, occurrence
integer, return_opt integer, flags text) returns integer
REGEXP_INSTR(string text, pattern text, startPos integer, occurrence
integer, return_opt integer, flags text) returns integer
REGEXP_INSTR(string text, pattern text, startPos integer, occurrence
integer, return_opt integer, flags text, group integer) returns integer
```

##### 通用规则

- REGEXP\_INSTR函数返回一个整数，表示匹配子字符串的开始或结束位置，具体取决于return\_opt参数的值。如果没有找到匹配项，则该函数返回0。
- 搜索从string中指定的起始位置startPos开始，默认从string的开头开始。
- startPos是正整数，不允许使用负值从string的末尾开始搜索。
- occurrence是一个正整数，表示应该搜索string中pattern的第几个出现次数。默认值为1，表示在string中搜索第一个出现的pattern。
- return\_opt允许您指定与出现次数相关的返回内容：
  - 当return\_opt = 0时，返回第一次出现的第一个字符的位置。这是默认值。
  - 当return\_opt = 1时，返回出现后面一个字符的位置。
- flags是一个字符表达式，它允许您改变函数的默认匹配行为。有关详细信息，请参阅REGEXP\_COUNT。



- 对于带有捕获组的pattern，group是一个正整数，表示应该返回pattern中的哪个捕获组。捕获组可以嵌套，它们按其在pattern中出现左括号的顺序进行编号。如果group为零，则返回与模式匹配的整个子字符串的位置。如果group大于pattern中的捕获组数量，则函数返回零。空的group值返回NULL。group的默认值为零。

注意

如果使用REGEXP\_INSTR函数，则需要事先在search\_path中指定“oracle”。

参见

有关如何编辑search\_path的信息，请参阅“使用oracle的注意事项”。

示例

```

SELECT REGEXP_INSTR(' 1234567890', '(123)(4(56)(78))') FROM DUAL;
  regexp_instr
  -----
                1
(1 row)

SELECT REGEXP_INSTR(' 1234567890', '(4(56)(78))', 3) FROM DUAL;
  regexp_instr
  -----
                4
(1 row)

SELECT REGEXP_INSTR(' 123 123456 1234567, 1234567 1234567 12', '[^ ]+', 1, 6)
FROM DUAL;
  regexp_instr
  -----
                37
(1 row)

SELECT REGEXP_INSTR('199 Oretax Prayers, Riffler Stream, CA', '[S|R|P]
[[:alpha:]]{6}', 3, 2, 1) FROM DUAL;
  regexp_instr
  -----
                28
(1 row)

```

#### H. 56. 5. 2. 11. REGEXP\_LIKE

描述

在查询的WHERE子句中使用，使得查询返回与给定模式匹配的行。

语法

```

REGEXP_LIKE(string text, pattern text) returns boolean
REGEXP_LIKE(string text, pattern text, flags text) returns boolean

```

通用规则

- REGEXP\_LIKE类似于LIKE条件，但它执行正则表达式匹配，而不是LIKE执行的简单模式匹配。

- 返回一个布尔值，当模式匹配字符串时为true，否则为false。 如果参数中有一个为空，则返回 NULL，但 'flags' 参数可以为空。
- flags是一个字符表达式，它允许您更改函数的默认匹配行为。有关详细信息，请参见 REGEXP\_COUNT。

注意

如果使用REGEXP\_LIKE函数，需要预先指定“oracle”作为search\_path。

参见

有关如何编辑search\_path的信息，请参见“使用oracle的注意事项”。

示例

```

SELECT REGEXP_LIKE('a' || CHR(10) || 'd', 'a.d', 'm') FROM DUAL;
  regexp_like
-----
f
(1 row)

SELECT REGEXP_LIKE('a' || CHR(10) || 'd', 'a.d', 'n') FROM DUAL;
  regexp_like
-----
t
(1 row)

```

#### H. 56. 5. 2. 12. REGEXP\_SUBSTR

描述

返回与调用函数中指定的模式匹配的字符串。

语法

```

REGEXP_SUBSTR(string text, pattern text) returns text
REGEXP_SUBSTR(string text, pattern text, startPos integer) returns text
REGEXP_SUBSTR(string text, pattern text, startPos integer, occurrence
integer) returns text
REGEXP_SUBSTR(string text, pattern text, startPos integer, occurrence
integer, flags text) returns text
REGEXP_SUBSTR(string text, pattern text, startPos integer, occurrence
integer, flags text, group int) returns text
REGEXP_SUBSTR(string CLOB, pattern text) returns clob

```

通用规则

- REGEXP\_SUBSTR返回与将POSIX正则表达式模式与字符串匹配所得到的匹配子字符串。如果未找到匹配，则该函数返回NULL。 如果参数中有一个为空，则返回 NULL，但 'flags' 参数可以为空。 函数将字符串作为 TEXT 或 CLOB 数据返回，返回的字符集与第一个参数相同。
- 搜索从字符串中指定的起始位置startPos开始，默认从字符串的开头开始。
- startPos是一个正整数，不允许使用负值从字符串的末尾开始搜索。

- occurrence是一个正整数，表示在字符串中搜索第几个模式。默认值为1，表示在字符串中搜索第一个模式。
- flags是一个字符表达式，它允许您更改函数的默认匹配行为。有关详细信息，请参见 REGEXP\_COUNT。
- 对于具有捕获组的模式，\*group\*是一个正整数，表示应由函数返回模式中的哪个捕获组。捕获组可以嵌套，它们按照它们在模式中左括号出现的顺序进行编号。如果group为零，则返回与模式匹配的整个子字符串的位置。如果group值超过模式中捕获组的数量，则函数返回NULL。空的group值返回NULL。group的默认值为零。

注意

如果使用REGEXP\_SUBSTR函数，需要预先指定“oracle”作为search\_path。

参见

有关如何编辑search\_path的信息，请参见“使用oracle的注意事项”。

示例

```
SELECT REGEXP_SUBSTR('number of your street, zipcode town, FR', '[^,]+')
FROM DUAL;
  regexp_substr
  -----
  , zipcode town
(1 row)
```

```
SELECT regexp_substr('number of your street, zipcode town, FR', '[^,]+',
24) FROM DUAL;
  regexp_substr
  -----
  , FR
(1 row)
```

```
SELECT regexp_substr('number of your street, zipcode town, FR', '[^,]+', 1,
2) FROM DUAL;
  regexp_substr
  -----
  , FR
(1 row)
```

```
SELECT regexp_substr('1234567890 1234567890', '(123)(4(56)(78))', 1, 1, 'i',
0) FROM DUAL;
  regexp_substr
  -----
  12345678
(1 row)
```

### H. 56. 5. 2. 13. REGEXP\_REPLACE

描述

返回与调用函数中指定的模式匹配的字符串。

语法

```

REGEXP_REPLACE(string text, pattern text, replace_string text, startPos
integer,
occurrence integer, flags text) returns text
REGEXP_REPLACE(string clob, pattern text, replace_string text) returns clob

```

#### 通用规则

- REGEXP\_REPLACE 返回修改后的源字符串，其中源字符串中出现的 POSIX 正则表达式模式会被替换为指定的替换字符串。如果未找到匹配项或查询的出现次数超过了匹配项的数量，则返回未修改的源字符串。函数返回值为 TEXT 类型如果第一个参数不是 CLOB 类型，CLOB 类型如果第一个参数是 CLOB 类型。
- 搜索和替换从指定的起始位置 startPos 开始，在 string 中进行，默认从 string 的开头开始。
- startPos 是一个正整数，不允许使用负值从 string 的末尾开始搜索。
- occurrence 是一个正整数，表示应搜索和替换 string 中 pattern 的哪个出现次数。默认值为 0，表示替换 string 中所有的 pattern。
- flags 是一个字符表达式，可让您更改函数的默认匹配行为。有关详细信息，请参见 REGEXP\_COUNT。

#### 注意

如果使用 REGEXP\_REPLACE 函数，则需要提前指定 "oracle" 作为 search\_path。

#### 参见

有关如何编辑 search\_path 的信息，请参阅 "使用 orafce 的注释"。

#### 示例

```

SELECT regexp_replace('512.123.4567 612.123.4567', '([[:digit:]]{3})\.[
([[:digit:]]{3})\.[([[:digit:]]{4})', '(\1) \2-\3') FROM DUAL;

```

```

          regexp_replace
-----
(512) 123-4567 (612) 123-4567
(1 row)

```

```

SELECT oracle.REGEXP_REPLACE('number your street, zipcode town,
FR', '() {2,}', ' ', 9);

```

```

          regexp_replace
-----
number your street, zipcode town, FR
(1 row)

```

```

SELECT oracle.REGEXP_REPLACE('number your street, zipcode town,
FR', '() {2,}', ' ', 9, 2);

```

```

          regexp_replace
-----
number your street, zipcode town, FR
(1 row)

```

## H. 56. 5. 2. 14. REPLACE

#### 描述

返回将每个 `search_string` 的出现替换为 `replacement_string` 的字符串。

语法

```
REPLACE(string text, pattern text) returns text
REPLACE(string text, pattern text, replace_string text) returns text
REPLACE(string uuid, pattern text, replace_string text) returns text
REPLACE(string CLOB, pattern text) returns CLOB
REPLACE(string CLOB, pattern text, replace_string text) returns CLOB
```

通用规则

- 如果省略或为 `null`，则将删除 `search_string` 的所有出现。如果 `search_string` 为 `null`，则返回 `string`。
- 函数返回值为 `TEXT` 类型如果第一个参数不是 `CLOB` 类型，`CLOB` 类型如果第一个参数是 `CLOB` 类型。

注意

上述 `REPLACE` 规范使用 `orafce` 来实现其行为，这与 `LightDB` 的 `REPLACE` 不同。必须修改 `search_path` 参数，以使其按照 `orafce` 规范进行操作。

信息

`LightDB` 的 `REPLACE` 的通用规则如下：

- 如果 `replace_string` 或 `search_string` 为 `null`，则返回 `null`。

参见

有关如何编辑 `search_path` 的信息，请参阅“使用 `orafce` 的注释”。

示例

```
SELECT REPLACE('abcdAbcdasd', 'a') FROM DUAL;
  replace
-----
bcdAbcdsd
(1 row)

SELECT REPLACE('abcdAbcdasd', 'a', 'c') FROM DUAL;
  replace
-----
cbcdAbcdsd
(1 row)

SELECT REPLACE('abcdAbcdasd', NULL, 'c') FROM DUAL;
  replace
-----
abcdAbcdasd
(1 row)

SELECT REPLACE('abcdAbcdasd', 'a', NULL) FROM DUAL;
  replace
-----
bcdAbcdsd
```

(1 row)

## H. 56. 5. 2. 15. RPAD

### 描述

使用一系列字符将字符串右侧填充到指定的长度。

### 语法

```
RPAD(str text, len integer) returns text
RPAD(str text, len integer, paddingStr text) returns text
RPAD(str CLOB, len integer, paddingStr text) returns CLOB
```

### 通用规则

- RPAD 返回在使用填充字符 paddingStr 反复填充字符串 str 的末尾直到字符串达到长度 len 后的结果。
- 如果字符串是 CHAR 类型，则添加填充字符时不会删除尾随空格。
- 在结果字符串中，全角字符被认为长度为 2，半角字符长度为 1。如果由于只有一个半角字符的空间可用而无法包含全角字符，则使用单字节空格填充字符串。
- 返回的字符串是 TEXT 类型如果第一个参数是字符类型，CLOB 类型如果第一个参数是 CLOB 类型。

### 注意

上述 RPAD 规范使用 orafce 来实现其行为，这与 LightDB 的 RPAD 不同。必须修改 search\_path 参数，以使其按照 orafce 规范进行操作。

### 信息

LightDB 的 RPAD 的通用规则如下：

- 如果字符串是 CHAR 类型，则删除尾随空格，然后将填充字符添加到字符串中。
- 结果长度是字符数。

### 参见

- 有关如何编辑 search\_path 的信息，请参阅“使用 orafce 的注释”。
- 有关 RPAD 的信息，请参阅 LightDB 文档中的“SQL 语言” > “函数和运算符” > “字符串函数和运算符”。

### 示例

在以下示例中，返回通过使用“a”右侧填充字符串“abc”形成的长度为 10 个字符的字符串。

```
SELECT RPAD(' abc', 10, ' a') FROM DUAL;
      rpad
-----
abcaaaaaaa
(1 row)
```

## H. 56. 5. 2. 16. RTRIM

### 描述

从字符串末尾删除指定的字符。

### 语法

```
RTRIM(str text) returns text
RTRIM(str text, trimChars text) returns text
RTRIM(str clob) returns clob
RTRIM(str clob, trimChars text) returns clob
```

### 通用规则

- RTRIM 返回从字符串 `str` 末尾删除 `trimChars` 后的字符串。
- 如果指定了多个修剪字符，则删除与修剪字符匹配的所有字符。如果省略了 `trimChars`，则删除所有尾随半角空格。
- 返回的字符串是 `TEXT` 类型如果第一个参数是字符类型，`CLOB` 类型如果第一个参数是 `CLOB` 类型。

### 注意

上述 RTRIM 规范使用 `orafce` 来实现其行为，这与 `LightDB` 的 RTRIM 不同。必须修改 `search_path` 参数，以使其与 `orafce` 规范的行为相同。

### 信息

`LightDB` 的 RTRIM 的通用规则如下：

- 如果字符串是 `CHAR` 类型，则删除尾随空格，然后删除修剪字符。

### 请参见

- 参阅“使用`orafce`的注意事项”了解如何编辑`search_path`。
- 请参阅`LightDB`文档中的“SQL语言” > “函数和操作符” > “字符串函数和操作符”，了解有关RTRIM的信息。

### 示例

在下面的示例中，将从“`aabcab`”的结尾中删除“`ab`”字符串并返回。

```
SELECT RTRIM('aabcab','ab') FROM DUAL;
 rtrim
-----
 aabc
(1 row)
```

## H. 56. 5. 2. 17. SOUNDEX

### 描述

SOUNDEX返回一个包含`char`的语音表示的字符字符串。 此函数可让您比较不同拼写但在英语中发音相似的单词。

语法

SOUNDEX(str text) returns text

通用规则

- 语音表示在Donald E. Knuth的《计算机程序设计艺术》第3卷：“排序和搜索”中定义。
- 保留字符串的第一个字母并删除以下字母的所有其他出现次数：  
a, e, h, i, o, u, w, y。
- 将剩下的字母按以下方式分配数字：

```
b, f, p, v = 1
c, g, j, k, q, s, x, z = 2
d, t = 3
l = 4
m, n = 5
r = 6
```

- 如果原始名称（在步骤1之前）中有两个或多个具有相同数字的字母相邻，或者除了任何插入的h和w之外相邻，则保留第一个字母并省略所有具有相同数字的相邻字母的其余部分。
- 返回前4个字节，右侧补0。

注意

如果使用SOUNDEX函数，则需要事先在search\_path中指定“oracle”。

请参见

参阅“使用orafce的注意事项”了解如何编辑search\_path。

示例

在下面的示例中，将从“aabcab”的结尾中删除“ab”字符串并返回。

```
SELECT SOUNDEX('SMYTHE')=SOUNDEX('Smith') as same FROM DUAL;
same
-----
t
(1 row)
```

H. 56. 5. 2. 18. SUBSTR

描述

使用字符指定位置和长度提取字符串的一部分。

语法

```
SUBSTR(str text, startPos integer) returns text
SUBSTR(str text, startPos numeric) returns text
```



```
SUBSTR(str text, startPos integer, len integer) returns text
SUBSTR(str text, startPos numeric, len numeric) returns text
SUBSTR(str CLOB, startPos number) returns CLOB
SUBSTR(str CLOB, startPos number, len number) returns CLOB
```

#### 通用规则

- SUBSTR从字符串str的位置startPos开始提取并返回一个子字符串，长度为len个字符。
- 当startPos为正数时，它将是字符串开头算起的字符数。
- 当startPos为0时，它将被视为1。
- 当startPos为负数时，它将是字符串末尾算起的字符数。
- 当未指定len时，将返回字符串末尾的所有字符。如果len小于1，则返回NULL。
- 对于startPos和len，请指定整数或NUMERIC类型。如果指定了包含小数位的数字，则将其截断为整数。
- 函数返回值为 TEXT 类型如果第一个参数是字符类型，CLOB 类型如果第一个参数是 CLOB 类型。

#### 注意

- 有两种类型的SUBSTR。一种与上述描述相同，另一种与SUBSTRING相同。必须修改search\_path参数才能使其与上述规范的行为相同。
- 如果未实施更改，则SUBSTR与SUBSTRING相同。

#### 信息

LightDB的SUBSTRING的通用规则如下：

- 无论起始位置是正数、0还是负数，起始位置都将从字符串开头算起。
- 当未指定len时，将返回字符串末尾的所有字符。
- 如果未提取字符串或len小于1，则返回空字符串。

#### 请参见

- 请参阅LightDB文档中的“[The SQL Language](#)” > “[Functions and Operators](#)” > “[String Functions and Operators](#)”中的SUBSTRING信息。

#### 示例

在下面的示例中，提取了字符串“ABCDEFGH”的部分内容。

```
SELECT SUBSTR('ABCDEFGH', 3, 4) "Substring" FROM DUAL;
Substring
```

```
-----
CDEF
(1 row)
```

```
SELECT SUBSTR('ABCDEFGH', -5, 4) "Substring" FROM DUAL;
Substring
```

```
-----
CDEF
(1 row)
```

### H. 56. 5. 2. 19. SUBSTRB

#### 描述

使用字节指定位置和长度提取字符串的一部分。

#### 语法

```
SUBSTRB(str text, startPos integer) returns varchar2
SUBSTRB(str text, startPos integer, len integer) returns varchar2
```

#### 通用规则

- SUBSTR从字符串str的字节位置startPos开始提取并返回一个子字符串，长度为len个字节。
- 当startPos为0或负数时，提取从起始位置减1并向左移动相应位置数后找到的位置开始。
- 当未指定len时，将返回字符串末尾的所有字节。
- 如果未提取字符串或len小于1，则返回空字符串。
- 对于startPos和len，请指定SMALLINT或INTEGER类型。
- 返回值的数据类型是VARCHAR2。

#### 注意

SUBSTRB的外部规范与orafce添加的SUBSTR不同，符合LightDB的SUBSTRING。

#### 示例

在下面的示例中，提取了字符串“aaabbbccc”的部分内容。

```
SELECT SUBSTRB('aaabbbccc', 4, 3) FROM DUAL;
substrb
-----
bbb
(1 row)

SELECT SUBSTRB('aaabbbccc', -2, 6) FROM DUAL;
substrb
-----
aaa
(1 row)
```

### H. 56. 5. 2. 20. CONCAT

#### 描述

拼接两个字符串

#### 语法

```
CONCAT(str1 TEXT, str2 TEXT) returns TEXT
```

CONCAT(str1 CLOB, str2 CLOB) returns CLOB

通用规则

- 拼接两个不同类型的值时，该函数返回值类型为导致无损转换的数据类型。
- 该函数和拼接操作符 || 等价。
- 函数返回值为 TEXT 类型如果第一个参数是字符类型，CLOB 类型如果第一个参数是 CLOB 类型。

样例

```
SELECT concat('a', 'b') FROM DUAL;
concat
-----
ab
(1 row)

SELECT concat(to_clob('a'), 'b') FROM DUAL;
concat
-----
ab
(1 row)
```

H. 56. 5. 2. 21. UPPER

描述

UPPER 返回所有字母为大写的字符串

语法

UPPER(string TEXT) returns TEXT  
UPPER(string CLOB) returns CLOB

通用规则

- 函数返回值为 TEXT 类型如果第一个参数是字符类型，CLOB 类型如果第一个参数是 CLOB 类型。

样例

```
SELECT UPPER('hello world') FROM DUAL;
upper
-----
HELLO WORLD
(1 row)

SELECT UPPER(to_clob('hello world')) FROM DUAL;
upper
-----
HELLO WORLD
(1 row)
```

### H. 56. 5. 3. 日期/时间函数

支持以下日期/时间函数：

- ADD\_MONTHS
- DBTIMEZONE
- LAST\_DAY
- MONTHS\_BETWEEN
- NEXT\_DAY
- ROUND
- SESSIONTIMEZONE
- SYSDATE
- TRUNC
- TZ\_OFFSET

注意

如果日期/时间函数中只显示了DATE类型，则这些函数可在orafce和LightDB中使用。

#### H. 56. 5. 3. 1. ADD\_MONTHS

描述

对日期添加月份。

语法

```
ADD_MONTHS(date DATE, months INTEGER) returns DATE
ADD_MONTHS(date TIMESTAMP WITH TIME ZONE, months INTEGER) returns TIMESTAMP
```

通用规则

- ADD\_MONTHS返回date加上months。
- 对于date，请指定DATE或TIMESTAMP WITH TIME ZONE类型。
- 对于months，请指定SMALLINT或INTEGER类型。
- 如果months指定为负值，则从日期中减去月数。
- 返回值的数据类型是DATE或TIMESTAMP。

注意

如果使用orafce的DATE类型，则需要提前指定search\_path为“oracle”。

请参见

有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。

示例

下面的示例显示了在日期2016年5月1日上添加3个月的结果。

```
SELECT ADD_MONTHS (DATE '2016/05/01', 3) FROM DUAL;
      add_months
-----
2016-08-01 00:00:00
(1 row)
```

### H. 56. 5. 3. 2. DBTIMEZONE

#### 描述

返回数据库时区的值。

#### 语法

```
DBTIMEZONE () returns text
```

#### 通用规则

- DBTIMEZONE返回数据库的时区值。
- 返回值的数据类型为TEXT。

#### 注意

- 如果使用DBTIMEZONE，则需要提前在search\_path中指定“oracle”。
- 数据库的时区默认设置为“GMT”。要更改时区，请更改“orafce.timezone”参数。下面是使用SET语句的示例。

```
SET orafce.timezone = 'Japan';
```

- 可以使用任何设置服务器参数的方法来设置orafce.timezone设置。
- 如果使用设置了orafce.timezone的SQL语句，可能会显示以下消息，但是参数设置已启用，所以可以忽略此消息。

```
WARNING: unrecognized configuration parameter "orafce.timezone"
```

- 可以在“orafce.timezone”中设置的时区与“TimeZone”服务器参数的设置相同。

#### 参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关时区的信息，请参阅LightDB文档中的“The SQL Language” > “Data Types” > “Date/Time Types”。

#### 示例

在以下示例中，返回DBTIMEZONE结果。

```
SELECT DBTIMEZONE () FROM DUAL;
      dbtimezone
```

```

-----
GMT
(1 row)

```

### H. 56. 5. 3. 3. LAST\_DAY

#### 描述

返回指定日期所在月份的最后一天。

#### 语法

```

LAST_DAY(date DATE) returns DATE
LAST_DAY(date TIMESTAMPTZ) returns TIMESTAMPTZ

```

#### 通用规则

- LAST\_DAY返回指定日期所在月份的最后一天。
- 对于date，请指定DATE或TIMESTAMPTZ类型。
- 返回值的数据类型为DATE或TIMESTAMPTZ。

#### 注意

如果使用orafce的DATE类型，则需要提前在search\_path中指定“oracle”。

#### 参见

有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。

#### 示例

在以下示例中，返回“2016年2月1日”的最后一天。

```

SELECT LAST_DAY(DATE' 2016/02/01') FROM DUAL;
      last_day
-----
2016-02-29 00:00:00
(1 row)

```

### H. 56. 5. 3. 4. MONTHS\_BETWEEN

#### 描述

返回两个日期之间的月数。

#### 语法

```

MONTHS_BETWEEN(date1 DATE, date2 DATE) returns NUMERIC
MONTHS_BETWEEN(date1 TIMESTAMP WITH TIME ZONE, date2 TIMESTAMP WITH TIME
ZONE) returns NUMERIC

```

#### 通用规则

- MONTHS\_BETWEEN返回date1和date2之间月份的差异。

- 对于date1和date2，请指定DATE或TIMESTAMP TZ类型。
- 如果date2比date1早，返回值将为负数。
- 如果两个日期是同一天，或者这两个日期都是它们所属月份的最后一天，则返回一个整数。如果这两个日期的天数不同，则将一个月视为31天，并返回一个值，该值加上日期差的除以31的结果。
- 返回值的数据类型为DATE或NUMERIC。

注意

如果使用orafce的DATE类型，则需要提前在search\_path中指定“oracle”。

参见

有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。

示例

在以下示例中，返回2016年3月15日和2015年11月15日之间月份的差异。

```
SELECT MONTHS_BETWEEN( DATE' 2016/03/15' , DATE' 2015/11/15' ) FROM DUAL;
months_between
-----
4
(1 row)
```

### H. 56. 5. 3. 5. NEXT\_DAY

描述

返回指定日期之后的第一个特定星期几的日期。

语法

```
NEXT_DAY(date DATE, dayOfWk TEXT) returns DATE
NEXT_DAY(date DATE, dayOfWk INTEGER) returns DATE
NEXT_DAY(date TIMESTAMP WITH TIME ZONE, dayOfWk TEXT) returns TIMESTAMP
NEXT_DAY(date TIMESTAMP WITH TIME ZONE, dayOfWk INTEGER) returns TIMESTAMP
```

通用规则

- NEXT\_DAY 返回符合条件的日期，即在指定日期 date 之后的第一个 dayOfWk 的实例。
- 对于 date，请指定为 DATE 或 TIMESTAMP TZ WITH TIME ZONE 类型。
- 指定表示星期几的数字值或字符串。

可指定的星期几值

Table H.111. NEXT\_DAY 函数中设置 dayOfWk 的值

设置示例	概述
1	可指定数字 1 到 7 表示星期日到星期六
'Sun' 或 'Sunday'	英文显示的星期几
'*'	日文显示的星期几

- 返回值的数据类型为 DATE 或 TIMESTAMP。

注意

- 如果使用 orafce 的 DATE 类型，则需要提前指定 search\_path 为 "oracle"。
- 使用日语输入日期的能力是由 orafce 私有规范提供的。当使用除 NEXT\_DAY 之外的日期/时间函数（如 TO\_DATE）时，无法使用日语输入日期。

参见

有关如何编辑 search\_path 的信息，请参阅 "使用 orafce 的注意事项"。

示例

在下面的示例中，返回 "2016 年 5 月 1 日" 或之后的第一个星期五的日期。

```
SELECT NEXT_DAY(DATE' 2016/05/01', 'Friday') FROM DUAL;
      next_day
-----
2016-05-06 00:00:00
(1 row)
```

#### H. 56. 5. 3. 6. ROUND 函数

描述

对日期进行四舍五入。

语法

```
ROUND(date DATE) returns DATE
ROUND(date DATE, fmt TEXT) returns DATE
ROUND(date TIMESTAMP WITH TIME ZONE) returns TIMESTAMP WITH TIME ZONE
ROUND(date TIMESTAMP WITH TIME ZONE, fmt TEXT) returns TIMESTAMP WITH TIME
ZONE
ROUND(date TIMESTAMP WITHOUT TIME ZONE) returns TIMESTAMP WITHOUT TIME ZONE
ROUND(date TIMESTAMP WITHOUT TIME ZONE, fmt TEXT) returns TIMESTAMP WITHOUT
TIME ZONE
```

通用规则

- ROUND 函数返回按格式模型 fmt 指定的单位进行四舍五入的日期。
- 对于 date，请指定为 DATE 或 TIMESTAMPTZ 类型。
- 以字符串形式指定格式模型。

可指定的格式模型值

Table H. 112. ROUND 函数中设置格式模型

格式模型	舍入单位
Y, YY, YYY, YYYY, SYYYY, YEAR, SYEAR	年



格式模型	舍入单位
I, IY, IYY, IYYY	年（包括符合 ISO 标准的日历周的值）
Q	季度
WW	周（一年的第一天）
IW	周（该周的星期一）
W	周（该月的第一个星期日所在的周）
DAY, DY, D	周（该周的星期日）
MONTH, MON, MM, RM	月
CC, SCC	世纪
DDD, DD, J	日
HH, HH12, HH24	小时
MI	分钟

- 如果小数位数被四舍五入：对于年，舍入边界为 7 月 1 日；对于月份，日期为 16 日；对于周，星期四为舍入边界。
- 如果省略 fmt，则按天四舍五入日期。
- 如果在日期中指定了 LightDB 的 DATE 类型，则返回值的数据类型将为 DATE 类型。如果在日期中指定了 TIMESTAMP 类型，则数据类型将为 TIMESTAMP WITH TIME ZONE，无论是否使用时区。

示例

在下面的示例中，返回按本周的星期进行四舍五入的“2016 年 6 月 20 日 18:00:00”的结果。

```
SELECT ROUND(TIMESTAMP' 2016/06/20 18:00:00', 'DAY') FROM DUAL;
          round
-----
2016-06-19 00:00:00+09
(1 row)
```

### H. 56. 5. 3. 7. SESSIONTIMEZONE

描述

返回会话的时区。

语法

```
SESSIONTIMEZONE() returns text
```

通用规则

- SESSIONTIMEZONE 返回会话之间的时区值。
- 返回值的数据类型为TEXT。

注意

- 如果使用SESSIONTIMEZONE，需要提前将search\_path指定为“oracle”。

- SESSIONTIMEZONE返回的值将成为“TimeZone”服务器参数中设置的值。

参见

有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。

示例

在下面的示例中，返回会话的时区。

```
SELECT SESSIONTIMEZONE() FROM DUAL;
 sessiontimezone
-----
Japan
(1 row)
```

### H. 56. 5. 3. 8. SYSDATE

描述

返回系统日期。

语法

```
SYSDATE() returns oracle.date
```

通用规则

- SYSDATE返回系统日期。
- 返回值的数据类型是orafce的DATE类型。

注意

- 如果使用SYSDATE，需要提前将search\_path指定为“oracle”。
- SYSDATE返回的日期依赖于orafce数据库的时区值。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关数据库时区值的信息，请参阅“DBTIMEZONE”。
- 有关时区的信息，请参阅LightDB文档中的“The SQL Language” > “Data Types” > “Date/Time Types”。

示例

在下面的示例中，返回系统日期。

```
SELECT SYSDATE() FROM DUAL;
 sysdate
-----
2016-06-22 08:06:51
(1 row)
```

### H. 56. 5. 3. 9. TRUNC

#### 描述

截取日期部分。

#### 语法

```
TRUNC(date DATE) returns DATE
TRUNC(date DATE, fmt TEXT) returns DATE
TRUNC(date TIMESTAMP WITH TIME ZONE) returns TIMESTAMP WITH TIME ZONE
TRUNC(date TIMESTAMP WITH TIME ZONE, fmt TEXT) returns TIMESTAMP WITH TIME
ZONE
TRUNC(date TIMESTAMP WITHOUT TIME ZONE) returns TIMESTAMP WITHOUT TIME ZONE
TRUNC(date TIMESTAMP WITHOUT TIME ZONE, fmt TEXT) returns TIMESTAMP WITHOUT
TIME ZONE
```

#### 通用规则

- TRUNC返回按格式模型fmt指定的单位截断的日期。
- 对于date，请指定DATE或TIMESTAMP类型。
- 将格式模型指定为字符串。可以指定的值与ROUND相同。
- 如果省略fmt，则按天截断日期。
- 如果在日期中指定了LightDB的DATE类型，则返回值的数据类型将是该DATE类型。如果在日期中指定了TIMESTAMP类型，则返回值的数据类型为TIMESTAMP WITH TIME ZONE，无论是否使用时区。

#### 参见

有关可以为格式模型指定的值的信息，请参阅“ROUND”。

#### 示例

在下面的示例中，返回截取了日期部分的“August 10, 2016 15:30:00”。

```
SELECT TRUNC(TIMESTAMP' 2016/08/10 15:30:00', 'DDD') FROM DUAL;
      trunc
-----
2016-08-10 00:00:00+09
(1 row)
```

### H. 56. 5. 3. 10. TZ\_OFFSET

#### 描述

返回时区偏移量。

#### 语法

```
TZ_OFFSET(text) returns text
```

#### 通用规则

- 您可以输入一个有效的时区区域名称，或一个距UTC的时区偏移量（仅返回该偏移量本身）。
- 要获取time\_zone\_name的有效值列表，请查询pg\_timezone\_names的name列。

注意

- 如果使用TZ\_OFFSET，需要提前将search\_path指定为“oracle”。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 请参考LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”，了解如何设置服务器参数。

示例

```
SELECT TZ_OFFSET('Asia/Shanghai') FROM DUAL;  
tz_offset  
-----  
08:00:00  
(1 row)  
(1 row)
```

#### H. 56. 5. 4. 数据类型格式化函数

支持以下数据类型格式化函数：

- CONVERT
- NUMTOYMINTERVAL
- TO\_BLOB(raw)
- TO\_CHAR
- TO\_DATE
- TO\_MULTI\_BYTE
- TO\_NUMBER
- TO\_SINGLE\_BYTE
- ASCIISTR
- ASCII
- SQLCODE\_TO\_SQLSTATE
- TO\_CLOB
- RAWTOHEX
- ROWIDTOCHAR

##### H. 56. 5. 4. 1. CONVERT

描述

CONVERT将一个字符串从一个字符集转换为另一个字符集。

语法

```
CONVERT(str TEXT, dest_char_set TEXT, source_char_set TEXT DEFAULT 'utf8')
returns TEXT
```

通用规则

- str参数是要转换的值。
- dest\_char\_set参数是将str转换为的字符集的名称。
- source\_char\_set参数是数据库中存储str的字符集的名称。默认值为数据库字符集。
- 目标字符集和源字符集参数都可以是文字或包含字符集名称的列。
- 为了在字符转换中实现完全对应，目标字符集必须包含源字符集中定义的所有字符的表示。如果目标字符集中不存在某个字符，则会出现替换字符。替换字符可以作为字符集定义的一部分进行定义。

注意

- 如果使用CONVERT转换字符串，则需要事先指定“oracle”用于search\_path。

参见

- 请参考“使用orafce的注意事项”了解如何编辑search\_path。
- 请参考LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”，了解如何设置服务器参数。

示例

```
SELECT CONVERT(' 嫵嫵嫵', 'gbk', 'utf8') from dual; FROM DUAL;
convert
-----
测试
(1 row)
```

#### H. 56. 5. 4. 2. NUMTOYMINTERVAL

描述

NUMTOYMINTERVAL将数字转换为INTERVAL YEAR TO MONTH文字。

语法

```
NUMTOYMINTERVAL(n numeric, interval_unit text) returns interval
```

通用规则

- interval\_unit的值指定n的单位，并且必须解析为以下字符串值之一：'YEAR'，'MONTH'。
- interval\_unit不区分大小写。

注意

- 如果使用NUMTOYMINTERVAL指定日期/时间值，则需要事先指定“oracle”用于search\_path。

参见

- 请参考“使用orafce的注意事项”了解如何编辑search\_path。
- 请参考LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”，了解如何设置服务器参数。

示例

```
SELECT NUMTOYMINTERVAL(1,'YEAR') FROM DUAL;
numtoyminterval
-----
1 year
(1 row)
```

#### H. 56. 5. 4. 3. TO\_BLOB(raw)

描述

TO\_BLOB(raw)将RAW值转换为BLOB值。

语法

TO\_BLOB(raw) returns blob

注意

- 如果使用TO\_BLOB(raw)函数，则需要事先指定“oracle”用于search\_path。

参见

- 请参考“使用orafce的注意事项”了解如何编辑search\_path。
- 请参考LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”，了解如何设置服务器参数。

示例

```
SELECT TO_BLOB(UTL_RAW.CAST_TO_RAW('测试')) FROM DUAL;
to_blob
-----
\xe6b58be8af95
(1 row)
```

#### H. 56. 5. 4. 4. TO\_CHAR

描述

将一个值转换为字符串。

语法

```
TO_CHAR(num SMALLINT) returns TEXT
TO_CHAR(num INTEGER) returns TEXT
TO_CHAR(num BIGINT) returns TEXT
TO_CHAR(num REAL) returns TEXT
TO_CHAR(num DOUBLE PRECISION) returns TEXT
TO_CHAR(num NUMERIC) returns TEXT
TO_CHAR(date TIMESTAMP) returns TEXT
TO_CHAR(TEXT) returns TEXT

TO_CHAR(date TIMESTAMP, fmt TEXT) returns TEXT
TO_CHAR(date TIMESTAMP WITH TIME ZONE, fmt TEXT) returns TEXT

TO_CHAR(date TIMESTAMP, fmt TEXT, nls_date_language TEXT) returns TEXT
TO_CHAR(date TIMESTAMP WITH TIME ZONE, fmt TEXT, nls_date_language TEXT)
returns TEXT
```

#### 通用规则

- TO\_CHAR将特定的数字或日期/时间值转换为字符串。
- 对于num，请指定数字数据类型。
- 对于date，请指定DATE或TIMESTAMP类型。 另外，您可以事先为nls\_date\_format变量设置日期/时间格式，则无需 TO\_CHAR 调用时指定。 以下是使用SET语句进行设置的示例。

```
SET nls_date_format = 'YYYY/MM/DD HH24:MI:SS';
```

无论显示指定还是调用时指定，格式符大小写效果相同。

- 对于fmt，用于设置数据格式。 当第一个参数是数字类型时，请参见数字值的模版模式。当第一个参数是日期类型时，请参见日期值的模板模式。
- 对于nls\_date\_language，用于设置日期的显示语言环境。
- 返回值的数据类型为TEXT。

#### 注意

- 如果使用TO\_CHAR来指定日期/时间值，则需要事先将search\_path设置为“oracle”。
- 可以使用任何设置服务器参数的方法来设置nls\_date\_format设置。
- 如果设置了nls\_date\_format，则在执行SQL语句时可能会显示以下消息，但是参数设置已启用，因此可以忽略此消息。

```
WARNING: unrecognized configuration parameter "nls_date_format"
```

#### 参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”。

#### 示例

在以下示例中，将数字值“123.45”作为字符串返回。

```

SELECT TO_CHAR(123.45) FROM DUAL;
to_char
-----
123.45
(1 row)

SELECT to_char (sysdate, 'YYYY-MM-DD', 'NLS_DATE_LANGUAGE=' SIMPLIFIED
CHINESE'');
to_char
-----
2023-12-04
(1 row)

```

#### H. 56. 5. 4. 5. TO\_DATE

##### 描述

根据指定的格式将字符串转换为日期。

##### 语法

```

TO_DATE(str TEXT) returns TIMESTAMP
TO_DATE(str TEXT, fmt TEXT) returns TIMESTAMP

```

##### 通用规则

- TO\_DATE将字符串str根据指定的格式fmt转换为日期。
- 指定一个表示日期/时间的字符串。
- 指定所需的日期/时间格式。如果省略，则使用nls\_date\_format变量中指定的格式。 如果未设置nls\_date\_format变量，则使用现有的日期/时间输入解释。 以下是使用SET语句进行设置的示例。

```
SET nls_date_format = 'YYYY/MM/DD HH24:MI:SS';
```

- 返回值的数据类型为TIMESTAMP。

##### 注意

- 上述TO\_DATE规范使用orafce来实现，其行为与LightDB的TO\_DATE不同。 必须修改search\_path参数，以使其按照orafce规范运行。
- 可以使用任何设置服务器参数的方法来设置nls\_date\_format设置。
- 如果设置了nls\_date\_format，则在执行SQL语句时可能会显示以下消息，但是参数设置已启用，因此可以忽略此消息。

```
WARNING: unrecognized configuration parameter "nls_date_format"
```

##### 信息



LightDB中TO\_DATE指定数据类型格式的通用规则如下：

- 返回值的数据类型是LightDB的DATE类型。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关LightDB中TO\_DATE的信息，请参阅LightDB文档中的“[The SQL Language](#)” > “[Functions and Operators](#)” > “[Data Type Formatting Functions](#)”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“[服务器管理](#)” > “[服务器配置](#)” > “[设置参数](#)”。
- 有关现有日期/时间输入解释的信息，请参阅LightDB文档中的“[日期/时间支持](#)” > “[日期/时间输入解释](#)”。

示例

在以下示例中，将字符串“2016/12/31”转换为日期并返回。

```
SELECT TO_DATE(' 2016/12/31', 'YYYY/MM/DD') FROM DUAL;
      to_date
-----
2016-12-31 00:00:00
(1 row)
```

#### H. 56. 5. 4. 6. TO\_MULTI\_BYTE

描述

将单字节字符串转换为多字节字符串。

语法

```
TO_MULTI_BYTE(str TEXT) returns TEXT
```

通用规则

- TO\_MULTI\_BYTE将字符串str中的半角字符转换为全角字符，并返回转换后的字符串。
- 指定一个表示日期/时间的字符串。
- 指定所需的日期/时间格式。如果省略，则使用nls\_date\_format变量中指定的格式。 如果未设置nls\_date\_format变量，则使用现有的日期/时间输入解释。 以下是使用SET语句进行设置的示例。

```
Only halfwidth alphanumeric characters, spaces and symbols can be
converted.
```

- 返回值的数据类型是TEXT。

示例

在以下示例中，将“abc123”转换为全角字符并返回。

```
SELECT TO_MULTI_BYTE('abc123') FROM DUAL;
to_multi_byte
-----
*****
(1 row)
```

“\*\*\*\*\*”是多字节的“abc123”。

#### H. 56. 5. 4. 7. TO\_NUMBER

##### 描述

根据指定的格式将值转换为数字。

##### 语法

```
TO_NUMBER(str TEXT) returns NUMERIC
TO_NUMBER(num NUMERIC) returns NUMERIC
TO_NUMBER(num NUMERIC, fmt NUMERIC) returns NUMERIC
```

##### 通用规则

- TO\_NUMBER将指定的值按照指定的格式fmt转换为数字值。
- 对于num，指定一个数字数据类型。
- 对于str，指定一个表示数字值的字符串。数字值必须由可转换字符组成。
- 指定所需的数字数据格式。指定的数字值将被处理为数据类型表达式。
- 返回值的数据类型是 NUMERIC。

##### 详见

有关数字值格式的信息，请参阅LightDB文档中的“The SQL Language” > “Functions and Operators” > “Data Type Formatting Functions”。

##### 示例

在以下示例中，将数字字面值“-130.5”转换为数字值并返回。

```
SELECT TO_NUMBER(-130.5) FROM DUAL;
to_number
-----
-130.5
(1 row)
```

#### H. 56. 5. 4. 8. TO\_SINGLE\_BYTE

##### 描述

将多字节字符串转换为单字节字符串。

##### 语法

```
TO_SINGLE_BYTE(str TEXT) returns TEXT
```

#### 通用规则

- TO\_SINGLE\_BYTE将字符串str中的全角字符转换为半角字符，并返回转换后的字符串。
- 只有可以显示为半角的全角字母数字字符、空格和符号才能被转换。
- 返回值的数据类型为TEXT。

#### 示例

在以下示例中，“\*\*\*\*\*”被转换为半角字符并返回。 “\*\*\*\*\*”是多字节字符串“xyz999”。

```
SELECT TO_SINGLE_BYTE('*****') FROM DUAL;
to_single_byte
-----
xyz999
(1 row)
```

### H. 56. 5. 4. 9. ASCIISTR

#### 描述

ASCIISTR以一个字符串或解析为字符串的表达式作为其参数，并返回数据库字符集中字符串的ASCII版本。

#### 语法

```
ASCIISTR(str TEXT) returns TEXT
```

#### 通用规则

- 非ASCII字符被转换为形式\ xxxx，其中xxxx表示UTF-16代码单元。
- 返回值的数据类型为TEXT。

#### 示例

在以下示例中，“Ä”被转换为“\00C4”。

```
SELECT ASCIISTR(' ABÄCDE') FROM DUAL;
asciistr
-----
AB\00C4CDE
(1 row)
```

### H. 56. 5. 4. 10. ASCII

#### 描述

ASCII以一个字符串或解析为字符串的表达式作为其参数，并返回第一个字符的数据库字符集中的值。

#### 语法

ASCIISTR(str TEXT) returns BIGINT

通用规则

- 如果您的数据库字符集是7位ASCII，则此函数返回ASCII值。如果您的数据库字符集是UTF-8 Code，则此函数返回UTF-8值。
- 返回值的数据类型为BIGINT。

示例

在以下示例中，“Ä”被转换为UTF-8中的“50052”。

```
SELECT ORACLE.ASCII('Ä') FROM DUAL;
  ascii
-----
  50052
(1 row)
```

#### H. 56. 5. 4. 11. SQLCODE\_TO\_SQLSTATE

描述

将整数错误代码转换为长度为5的字符串错误代码。

语法

SQLCODE\_TO\_SQLSTATE(code INTEGER) returns TEXT

通用规则

- 只能转换0和715827882之间的整数错误代码。
- 返回值的数据类型为TEXT。

示例

在以下示例中，转换错误码2。

```
select SQLCODE_TO_SQLSTATE(2);
  sqlcode_to_sqlstate
-----
  20000
(1 row)
```

#### H. 56. 5. 4. 12. TO\_CLOB

描述

将字符字符串或数值转换为CLOB值。

语法

TO\_CLOB(str TEXT) returns CLOB

通用规则

- TO\_CLOB 转换值为CLOB值.

示例

```
CREATE TABLE testorafce_to_clob (
    col_char CHAR(10),
    col_varchar2 VARCHAR2(20),
    col_varchar VARCHAR(20),
    col_nchar NCHAR(10),
    col_nvarchar2 NVARCHAR2(20),
    col_smallint smallint,
    col_integer integer,
    col_bigint bigint,
    col_decimal decimal,
    col_numeric numeric,
    col_real real,
    col_double double precision,
    col_clob CLOB,
    col_raw raw(10)
);
```

```
INSERT INTO testorafce_to_clob (col_char, col_varchar2, col_varchar, col_nchar,
    col_nvarchar2, col_smallint, col_integer, col_bigint, col_decimal, col_numeric,
    col_real, col_double, col_clob, col_raw)
VALUES ('ABC1', 'Hello', 'world', 'hello', 'world', 1, 2, 3, 4, 5, 6, 7, 'This
    is a CLOB', 'AB');
```

```
INSERT INTO testorafce_to_clob (col_char, col_varchar2, col_varchar, col_nchar,
    col_nvarchar2, col_smallint, col_integer, col_bigint, col_decimal, col_numeric,
    col_real, col_double, col_clob, col_raw)
VALUES ('ABC2', 'Hello', 'world', 'hello', 'world', 1, 2, 3, 4, 5, 6, 7, 'This
    is a CLOB', '1');
```

```
INSERT INTO testorafce_to_clob (col_char, col_varchar2, col_varchar, col_nchar,
    col_nvarchar2, col_smallint, col_integer, col_bigint, col_decimal, col_numeric,
    col_real, col_double, col_clob, col_raw)
VALUES ('ABC3', 'Hello', 'world', 'hello', 'world', 1, 2, 3, 4, 5, 6, 7,
    to_clob('This is a CLOB'), '1AB456789');
```

```
SELECT to_clob(col_char) AS clob_char,
    to_clob(col_varchar2) AS clob_varchar2,
    to_clob(col_varchar) AS col_varchar,
    to_clob(col_nchar) AS clob_nchar,
    to_clob(col_nvarchar2) AS clob_nvarchar2,
    to_clob(col_clob) AS clob_clob,
    to_clob(col_smallint) AS col_smallint,
    to_clob(col_integer) AS col_integer,
    to_clob(col_bigint) AS col_bigint,
    to_clob(col_decimal) AS col_decimal,
    to_clob(col_numeric) AS col_numeric,
    to_clob(col_real) AS col_real,
    to_clob(col_double) AS col_double,
    to_clob(col_raw) AS clob_nclob
```

```

FROM testorafce_to_clob order by col_char asc;

clob_char | clob_varchar2 | col_varchar | clob_nchar | clob_nvarchar2 |
  clob_clob | col_smallint | col_integer | col_bigint | col_decimal |
  col_numeric | col_real | col_double | clob_nclob
-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----
ABC1      | Hello          | world        | hello        | world        | This is
a CLOB | 1              | 2            | 3            | 4            | 5          |
6         | 7              | AB           |              |              |            |
ABC2      | Hello          | world        | hello        | world        | This is
a CLOB | 1              | 2            | 3            | 4            | 5          |
6         | 7              | 01          |              |              |            |
ABC3      | Hello          | world        | hello        | world        | This is
a CLOB | 1              | 2            | 3            | 4            | 5          |
6         | 7              | 01AB456789 |              |              |            |
(3 rows)

```

#### H. 56. 5. 4. 13. RAWTOHEX

描述

RAWTOHEX将传入值转换为包含其十六进制表示形式的字符值。

语法

```

RAWTOHEX(test_raw RAW) returns TEXT
RAWTOHEX(test_text TEXT) returns TEXT
RAWTOHEX(test_uuid UUID) returns VARCHAR2

```

通用规则

- 此函数返回一个值，该值具有构成传入值的字节的十六进制表示形式。每个字节由两个十六进制数字表示。

示例

```

select rawtohex(sys_guid());
           rawtohex
-----
6466c654a5dc4755902b70d0e6ea6eb6
(1 row)

```

#### H. 56. 5. 4. 14. ROWIDTOCHAR

描述

ROWIDTOCHAR将传入值ORACLE的tid类型的ROWID转换为varchar2形式的字符值。

语法

```

ROWIDTOCHAR(test_rowid TID) returns VARCHAR2

```

### 通用规则

- 此函数返回一个值，该值具有构成传入值的字节转化为的varchar2的表示形式。最大长度为18位。

### 示例

```
drop TABLE if EXISTS rowidtochar_t1;
create table rowidtochar_t1
(
    id int not null
);
insert into rowidtochar_t1 values(1);
select rowidtochar(rowid) from rowidtochar_t1;
rowidtochar
-----
(0,1)
(1 row)
```

## H. 56. 5. 5. 条件表达式

支持以下用于比较的函数：

- DECODE
- LNNVL
- NANVL
- NVL
- NVL2

### H. 56. 5. 5. 1. DECODE

#### 描述

比较值，如果它们匹配，则返回相应的值。

#### 语法

```
DECODE(expr, srch, result)
DECODE(expr, srch, result, default)
DECODE(expr, srch1, result1, srch2, result2)
DECODE(expr, srch1, result1, srch2, result2, default)
DECODE(expr, srch1, result1, srch2, result2, srch3, result3)
DECODE(expr, srch1, result1, srch2, result2, srch3, result3, default)
```

#### 通用规则

- DECODE按顺序逐个比较要转换的值表达式和搜索值。如果值匹配，则返回相应的结果值。如果没有匹配的值，则返回指定的默认值。如果没有指定默认值，则返回NULL值。
- 如果指定了相同的搜索值超过一次，则返回的结果值是列出的第一个搜索值的结果值。
- 结果值和默认值中可以使用以下数据类型：

- CHAR
- VARCHAR
- VARCHAR2
- NCHAR
- NCHAR VARYING
- NVARCHAR2
- TEXT
- INTEGER
- BIGINT
- NUMERIC
- DATE
- TIME WITHOUT TIME ZONE
- TIMESTAMP WITHOUT TIME ZONE
- TIMESTAMP WITH TIME ZONE
- 要转换的值和搜索值必须指定相同的数据类型。 但是，请注意，如果在搜索值中指定了文字，可以指定不同的数据类型，且要转换的值表达式包含可以转换的数据类型。
- 如果结果值和默认值都是文字，则这些值的数据类型如下所示：
  - 如果所有值都是字符串文字，则所有值都将成为字符类型。
  - 如果有一个或多个数字文字，则所有值都将变为数值类型。
  - 如果有一个或多个文字转换为日期时间/时间类型，则所有值都将变为日期时间/时间类型。
- 如果结果值和默认值包含文字和非文字的混合，则文字将转换为非文字的数据类型。
- 所有结果值和默认值都必须指定相同的数据类型。 但是，如果可以转换结果值或默认值的任何数据类型，则可以指定不同的数据类型-这些数据类型如下所示：

Table H. 113. DECODE可以转换的数据类型组合（摘要）

		其他结果值或默认值		
		数值类型	字符类型	日期/时间类型
结果值（任何）	数值类型	Y	N	N
	字符类型	N	Y	N
	日期/时间类型	N	N	S(*1)

Y: 可以转换

S: 可转换一些数据类型

N: 无法转换

\*1: 可转换日期/时间类型的数据类型如下所示：



Table H. 114. DECODE可转换的结果值和默认值日期/时间数据类型

		其他结果值或默认值			
		DATE	TIME WITHOUT TIME ZONE	TIMESTAMP WITHOUT TIME ZONE	TIMESTAMP WITH TIME ZONE
结果值（任何）	DATE	Y	N	Y	Y
	TIME WITHOUT TIME ZONE	N	Y	N	N
	TIMESTAMP WITHOUT TIME ZONE	Y	N	Y	Y
	TIMESTAMP WITH TIME ZONE	Y	N	Y	Y

Y: 可以转换

N: 无法转换

- 返回值的数据类型将是结果值或默认值中最长且精度最高的数据类型。

示例

在下面的示例中，将比较并转换表t1中col3的值为不同的值。如果col3值匹配搜索值1，则返回的结果值是“one”。如果col3值不与搜索值1、2或3中的任何一个匹配，则返回默认值“other number”。

```

SELECT col1,
       DECODE(col3, 1, 'one',
              2, 'two',
              3, 'three',
              'other number') "num-word"
FROM t1;
col1 | num-word
-----+-----
1001 | one
1002 | two
1003 | three
(3 rows)
    
```

### H. 56. 5. 5. 2. LNNVL

描述

确定指定条件的值是TRUE还是FALSE。

语法

```
LNNVL(cond BOOLEAN) returns BOOLEAN
```

通用规则

- LNNVL确定指定条件的值是TRUE还是FALSE。如果条件的结果是FALSE或NULL，则返回TRUE。如果条件的结果是TRUE，则返回FALSE。

- 在条件中指定了返回TRUE或FALSE的表达式。
- 返回值的数据类型是BOOLEAN。

示例

在以下示例中，当col3的值为2000或更低或为空时，返回表t1的col1和col3。

```
SELECT col1,col3 FROM t1 WHERE LNNVL( col3 > 2000 );
col1 | col3
-----+-----
1001 | 1000
1002 | 2000
2002 |
(3 row)
```

### H. 56. 5. 5. 3. NANVL函数

描述

当值不是数字 (NaN) 时返回替代值。

语法

```
NANVL(expr FLOAT4, substitute FLOAT4) returns FLOAT4
NANVL(expr FLOAT8, substitute FLOAT8) returns FLOAT8
NANVL(expr NUMERIC, substitute NUMERIC) returns NUMERIC
NANVL(expr FLOAT4, substitute VARCHAR) returns FLOAT4
NANVL(expr FLOAT8, substitute VARCHAR) returns FLOAT8
NANVL(expr NUMERIC, substitute VARCHAR) returns NUMERIC
```

通用规则

- NANVL函数在指定的值不是数字 (NaN) 时返回替代值。 替代值可以是数字或可转换为数字的字符串。
- 对于expr和substitute，请指定数字数据类型。 如果expr和substitute具有不同的数据类型， 它们将转换为具有更大长度或精度的数据类型， 并返回该数据类型。
- 对于substitute，您还可以指定表示数字值的字符串。
- 如果为替代值指定了字符串， 则返回值使用的数据类型与expr的数据类型相同。

示例

在以下示例中，如果表t1中的col1的值是NaN值，则返回“0”。

```
SELECT col1, NANVL(col3,0) FROM t1;
col1 | nanvl
-----+-----
2001 | 0
(1 row)
```

### H. 56. 5. 5. 4. NVL函数

描述

当值为NULL时返回替代值。

语法

```
NVL(expr1 anyelement, expr2 anyelement) return anyelement
```

通用规则

- NVL函数在指定的值为NULL时返回替代值。 当expr1为NULL时，返回expr2。 当expr1不为NULL时返回expr1。
- 请为expr1和expr2指定相同的数据类型。 然而，如果在expr2中指定了常量，并且数据类型也可以由expr1转换， 则可以指定不同的数据类型。 在这种情况下，expr2的转换将根据expr1的数据类型进行调整， 因此expr1为NULL值时返回的expr2的值将以expr1的数据类型转换。 对于类型（numeric、int）和（bigint、int）来说，这是不必要的。

示例

在以下示例中，如果表t1中的col1的值是NULL值，则返回“IS NULL”。

```
SELECT col2, NVL(col1,' IS NULL') "nv1" FROM t1;
 col2 |   nv1
-----+-----
  aaa | IS NULL
(1 row)
```

#### H. 56. 5. 5. 5. NVL2函数

描述

根据值是否为NULL或非NULL返回替代值。

语法

```
NVL2(expr anyelement, substitute1 anyelement, substitute2 anyelement)
return anyelement
```

通用规则

- NVL2函数根据指定的值是否为NULL返回替代值。 当expr为NULL时，返回substitute2。 当它不为NULL时，返回substitute1。
- 请为expr、substitute1和substitute2指定相同的数据类型。 然而，如果在substitute1或substitute2中指定了文字，而且数据类型也可以由expr转换，则可以指定不同的数据类型。在这种情况下，substitute1或substitute2将根据expr的数据类型进行调整，因此当expr为NULL值时返回的substitute2的值将以expr的数据类型转换。

示例

在以下示例中，如果表t1中列col1中的值为NULL，则返回“IS NULL”，如果不为NULL，则返回“IS NOT NULL”。

```
SELECT col2, NVL2(col1,' IS NOT NULL',' IS NULL') FROM t1;
 col2 |   nv2
-----+-----
```

```

aaa | IS NULL
bbb | IS NOT NULL
(2 row)

```

## H. 56. 5. 6. 聚合函数

支持以下聚合函数：

- ANY\_VALUE
- BIT\_AND\_AGG
- BIT\_OR\_AGG
- BIT\_XOR\_AGG
- KURTOSIS\_POP
- KURTOSIS\_SAMP
- LISTAGG
- MEDIAN
- SKEWNESS\_POP
- SKEWNESS\_SAMP

### H. 56. 5. 6. 1. ANY\_VALUE

描述

ANY\_VALUE函数返回表达式（expr）的一个非确定性值。

语法

```
ANY_VALUE(expr anyelement) returns anyelement
```

通用规则

- 使用ANY\_VALUE来优化包含GROUP BY子句的查询。ANY\_VALUE返回一组中一个表达式的值。它被优化以返回第一个值。
- 它确保不会为任何传入的行进行比较，并消除了将每个列指定为GROUP BY子句的一部分的必要性。
- 由于它不比较值，在GROUP BY查询中，ANY\_VALUE比MIN或MAX更快地返回一个值。
- 根据GROUP BY规范，返回每个组中的任何值。如果组中所有行的表达式值都为NULL，则返回NULL。
- ANY\_VALUE的结果是非确定性的。

注意

- 如果使用ANY\_VALUE，则需要事先指定search\_path为“oracle”。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。

- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”。

示例

```
create table test_any_value(key1 int, key2 int);
insert into test_any_value values(1, 1);
select any_value(key2) from test_any_value group by key1;
any_value
-----
          1
(1 row)
```

#### H. 56. 5. 6. 2. BIT\_AND\_AGG

描述

BIT\_AND\_AGG是一个位运算聚合函数，返回位与运算的结果。

语法

BIT\_AND\_AGG(numeric) returns int

通用规则

- 您可以将BIT\_AND\_AGG作为GROUP BY查询、窗口函数或分析函数的一部分使用。
- 对于一组给定的值，位运算聚合的结果总是确定性的，与排序无关。
- 返回值的数据类型是int。

注意

- 如果使用BIT\_AND\_AGG，则需要事先指定search\_path为“oracle”。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”。

示例

```
select bit_and_agg(column1) from (values (1), (2), (4), (8)) x;
bit_and_agg
-----
          0
(1 row)
```

#### H. 56. 5. 6. 3. BIT\_OR\_AGG

描述

BIT\_OR\_AGG是一个位运算聚合函数，返回位或运算的结果。

## 语法

`BIT_OR_AGG(numeric)` returns int

## 通用规则

- 您可以将BIT\_OR\_AGG作为GROUP BY查询、窗口函数或分析函数的一部分使用。
- 对于一组给定的值，位运算聚合的结果总是确定性的，与排序无关。
- 返回值的数据类型是int。

## 注意

- 如果使用BIT\_OR\_AGG，则需要事先指定search\_path为“oracle”。

## 参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参考LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”一节。

## 示例

```
select bit_or_agg(column1) from (values (1), (2), (4), (8)) x;
 bit_or_agg
-----
          15
(1 row)
```

### H. 56. 5. 6. 4. BIT\_XOR\_AGG

## 描述

BIT\_XOR\_AGG是一个位运算聚合函数，返回位异或操作的结果。

## 语法

`BIT_XOR_AGG(numeric)` returns int

## 通用规则

- 您可以将BIT\_XOR\_AGG作为GROUP BY查询、窗口函数或分析函数的一部分使用。
- 对于给定的一组值，位聚合的结果始终是确定性的，并且与排序无关。
- 返回值的数据类型是int。

## 注意

- 如果使用BIT\_XOR\_AGG函数，需要提前在search\_path中指定“oracle”。

## 参见

- 有关如何编辑search\_path的信息，请参考“使用orafce的注意事项”。

- 有关如何设置服务器参数的信息，请参考LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”一节。

示例

```
select bit_xor_agg(column1) from (values (1), (3), (4), (8)) x;
      bit_xor_agg
-----
                14
(1 row)
```

#### H. 56. 5. 6. 5. KURTOSIS\_POP

描述

总体峰度函数KURTOSIS\_POP主要用于确定给定分布中的异常值特征。

语法

KURTOSIS\_POP(numeric) returns numeric

通用规则

- expr中的NULL值将被忽略。
- 如果组中所有行的expr值都为NULL，则返回NULL。
- 如果expr中只有一个或两个行，则返回0。
- 对于给定的一组值，总体峰度（KURTOSIS\_POP）和样本峰度（KURTOSIS\_SAMP）的结果始终是确定性的。但是，KURTOSIS\_POP和KURTOSIS\_SAMP的值是不同的。随着数据集中值的数量增加，计算出的KURTOSIS\_SAMP和KURTOSIS\_POP值之间的差异会减小。

注意

- 如果使用KURTOSIS\_POP函数，需要提前在search\_path中指定“oracle”。

参见

- 有关如何编辑search\_path的信息，请参考“使用oracle的注意事项”。
- 有关如何设置服务器参数的信息，请参考LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”一节。

示例

```
select kurtosis_pop(column1) from (values (1), (2), (4), (8)) x;
      kurtosis_pop
-----
-1.09897920604914942667
(1 row)
```

#### H. 56. 5. 6. 6. KURTOSIS\_SAMP

描述

样本峰度(KURTOSIS\_SAMP)函数主要用于确定给定分布中的异常值特征。

语法

```
KURTOSIS_SAMP(numeric) returns numeric
```

总则

- expr中的NULL值将被忽略。
- 如果组中的所有行都具有NULL表达式值，则返回NULL。
- 如果expr中只有一行或两行，则返回0。
- 对于一组给定的值，样本峰度(KURTOSIS\_SAMP)和总体峰度(KURTOSIS\_POP)的结果始终是确定性的。然而，KURTOSIS\_SAMP和KURTOSIS\_POP的值是不同的。随着数据集中值的数量增加，计算出的KURTOSIS\_SAMP和KURTOSIS\_POP的差异减小。

注意

- 如果使用KURTOSIS\_SAMP函数，则需要提前指定“oracle”作为search\_path。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理”>“服务器配置”>“设置参数”。

示例

```
select kurtosis_samp(column1) from (values (1), (2), (4), (8)) x;
 kurtosis_samp
-----
0.7576559546313793
(1 row)
```

## H. 56. 5. 6. 7. LISTAGG

描述

返回一个由字符串值组成的连接的、带分隔符的列表。

语法

```
LISTAGG(strExpr TEXT) returns TEXT
LISTAGG(strExpr TEXT, delimiter TEXT) []returns TEXT
LISTAGG(strExpr TEXT, delimiter TEXT ON OVERFLOW TRUNCATE) []returns TEXT
LISTAGG(strExpr TEXT, delimiter TEXT ON OVERFLOW ERROR) []returns TEXT
```

总则

- LISTAGG连接和分隔一组字符串值，并返回结果。
- 对于delimiter，请指定一个字符串。如果省略了分隔符，则返回一个没有分隔符的字符串列表。



- 在LightDB中, LISTAGG支持在函数中使用distinct和within group子句。 例如: LISTAGG(distinct c1) within group(order by c1)
- 在LightDB中, LISTAGG支持在函数中使用on overflow truncate。 例如: LISTAGG(c1, ':' on overflow truncate)
- 在LightDB中, LISTAGG支持在函数中使用on overflow error。 例如: LISTAGG(c1, ':' on overflow error)
- 在LightDB中, LISTAGG支持在over子句中使用within group子句。 在这种情况下, 我们还不能在函数中使用distinct。 例如: LISTAGG(c1) within group(order by c1) over(partition by c2)
- 返回值的数据类型为TEXT。

#### 示例

在下面的示例中, 返回由表t1中列col2的值组成的、以“:”为分隔符的结果。

```
SELECT LISTAGG(col2,':') FROM t1;
      listagg
-----
AAAAA:BBBBB:CCCCC
(1 row)
```

在下面的示例中, 使用over中的within group。

```
CREATE TABLE EMP
( EMPNO    NUMBER(4, 0),
  ENAME    VARCHAR2(10),
  JOB      VARCHAR2(9),
  MGR      NUMBER(4, 0),
  HIREDATE DATE,
  SAL      NUMBER(7, 2),
  COMM     NUMBER(7, 2),
  DEPTNO   NUMBER(2, 0),
  DNAME    VARCHAR2(100),
  CONSTRAINT PK_EMP PRIMARY KEY (EMPNO)
);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
values (7369, 'SMITH', 'CLERK', 7902, to_date('1980-12-17', 'yyyy-mm-dd'),
800.00, null, 20, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
values (7499, 'ALLEN', 'SALESMAN', 7698, to_date('1981-02-20', 'yyyy-mm-dd'),
1600.00, 300.00, 30, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
values (7521, 'WARD', 'SALESMAN', 7698, to_date('1981-02-22', 'yyyy-mm-dd'),
1250.00, 500.00, 30, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
values (7566, 'JONES', 'MANAGER', 7839, to_date('1981-04-02', 'yyyy-mm-dd'),
2975.00, null, 20, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
values (7654, 'MARTIN', 'SALESMAN', 7698, to_date('1981-09-28', 'yyyy-mm-dd'),
1250.00, 1400.00, 30, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
values (7698, 'BLAKE', 'MANAGER', 7839, to_date('1981-05-01', 'yyyy-mm-dd'),
2850.00, null, 30, null);
```

```

insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
  values (7782, 'CLARK', 'MANAGER', 7839, to_date('1981-06-09', 'yyyy-mm-dd'),
    2450.00, null, 10, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
  values (7788, 'SCOTT', 'ANALYST', 7566, to_date('1987-04-19', 'yyyy-mm-dd'),
    3000.00, null, 20, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
  values (7839, 'KING', 'PRESIDENT', null, to_date('1981-11-17', 'yyyy-mm-dd'),
    5000.00, null, 10, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
  values (7844, 'TURNER', 'SALESMAN', 7698, to_date('1981-09-08', 'yyyy-mm-dd'),
    1500.00, 0.00, 30, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
  values (7876, 'ADAMS', 'CLERK', 7788, to_date('1987-05-23', 'yyyy-mm-dd'),
    1100.00, null, 20, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
  values (7900, 'JAMES', 'CLERK', 7698, to_date('1981-12-03', 'yyyy-mm-dd'),
    950.00, null, 30, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
  values (7902, 'FORD', 'ANALYST', 7566, to_date('1981-12-03', 'yyyy-mm-dd'),
    3000.00, null, 20, null);
insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO, DNAME)
  values (7934, 'MILLER', 'CLERK', 7782, to_date('1982-01-23', 'yyyy-mm-dd'),
    1300.00, null, 10, null);

```

```

select listagg(ename, ',') within group(order by ename) over(partition by
  deptno) as enames,
  deptno,
  ename
from EMP;

```

enames	deptno	ename
CLARK, KING, MILLER	10	CLARK
CLARK, KING, MILLER	10	KING
CLARK, KING, MILLER	10	MILLER
ADAMS, FORD, JONES, SCOTT, SMITH	20	ADAMS
ADAMS, FORD, JONES, SCOTT, SMITH	20	FORD
ADAMS, FORD, JONES, SCOTT, SMITH	20	JONES
ADAMS, FORD, JONES, SCOTT, SMITH	20	SCOTT
ADAMS, FORD, JONES, SCOTT, SMITH	20	SMITH
ALLEN, BLAKE, JAMES, MARTIN, TURNER, WARD	30	ALLEN
ALLEN, BLAKE, JAMES, MARTIN, TURNER, WARD	30	BLAKE
ALLEN, BLAKE, JAMES, MARTIN, TURNER, WARD	30	JAMES
ALLEN, BLAKE, JAMES, MARTIN, TURNER, WARD	30	MARTIN
ALLEN, BLAKE, JAMES, MARTIN, TURNER, WARD	30	TURNER
ALLEN, BLAKE, JAMES, MARTIN, TURNER, WARD	30	WARD

(14 rows)

## H. 56. 5. 6. 8. MEDIAN

描述

计算一组数字的中位数。

语法

MEDIAN(numExpr REAL) returns REAL  
 MEDIAN(numExpr DOUBLE PRECISION) returns DOUBLE PRECISION

#### 通用规则

- MEDIAN返回一组数字的中位数。
- 数字必须是数值数据类型。
- 如果数字是REAL类型，则返回值的数据类型为REAL；如果指定了其他类型，则为DOUBLE PRECISION。

#### 示例

在下面的示例中，返回表t1中列col3的中位数。

```
SELECT MEDIAN(col3) FROM t1;
   median
-----
      2000
(1 row)
```

### H. 56. 5. 6. 9. SKEWNESS\_POP (总体偏度)

#### 描述

SKEWNESS\_POP是一个聚合函数，主要用于确定给定分布中的对称性。

#### 语法

SKEWNESS\_POP(numeric) returns numeric

#### 通用规则

- 忽略expr中的NULL值。
- 如果组中所有行的expr值都为NULL，则返回NULL。
- 如果expr中有一行或两行，则返回0。
- 对于给定的值集，总体偏度 (SKEWNESS\_POP) 和样本偏度 (SKEWNESS\_SAMP) 的结果始终是确定性的。但是，SKEWNESS\_POP和SKEWNESS\_SAMP的值是不同的。随着数据集中值的数量增加，SKEWNESS\_SAMP和SKEWNESS\_POP的计算值之间的差异会减小。

#### 注意

- 如果使用SKEWNESS\_POP函数，则需要事先指定search\_path为“oracle”。

#### 参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”。

#### 示例

```
SELECT SKEWNESS_POP(column1) from (values (1), (2), (4), (8)) x;
      skewness_pop
-----
0.65680773449969915746
(1 row)
```

#### H. 56. 5. 6. 10. SKEWNESS\_SAMP (样本偏度)

##### 描述

SKEWNESS\_SAMP是一个聚合函数，主要用于确定给定分布中的对称性。

##### 语法

SKEWNESS\_SAMP(numeric) returns numeric

##### 通用规则

- 忽略expr中的NULL值。
- 如果组中所有行的expr值都为NULL，则返回NULL。
- 如果expr中有一行或两行，则返回0。
- 对于给定的值集，总体偏度（SKEWNESS\_POP）和样本偏度（SKEWNESS\_SAMP）的结果始终是确定性的。但是，SKEWNESS\_POP和SKEWNESS\_SAMP的值是不同的。随着数据集中值的数量增加，SKEWNESS\_SAMP和SKEWNESS\_POP的计算值之间的差异会减小。

##### 注意

- 如果使用SKEWNESS\_SAMP函数，则需要事先指定search\_path为“oracle”。

##### 参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理”>“服务器配置”>“设置参数”。

##### 示例

```
SELECT SKEWNESS_SAMP(column1) from (values (1), (2), (4), (8)) x;
      skewness_samp
-----
1.13762436695768880892
(1 row)
```

#### H. 56. 5. 7. 返回内部信息的函数

支持以下返回内部信息的函数：

- DUMP
- NLS\_CHARSET\_ID
- NLS\_CHARSET\_NAME
- SYS\_CONTEXT

- USERENV

#### H. 56. 5. 7. 1. DUMP

##### 描述

返回一个值的内部信息。

##### 语法

```
DUMP(expr TEXT) returns VARCHAR
DUMP(expr TEXT, fmt INTEGER) returns VARCHAR
```

##### 通用规则

- DUMP命令以符合输出格式的显示格式返回指定表达式的内部信息。
- 数据类型的内部代码 (Typ)、数据长度 (Len) 和数据的内部表达式将作为内部信息输出。
- 可以为表达式指定任何数据类型。
- 数据的内部表达式的显示格式 (基数n) 由输出格式指定。可以指定的基数为8、10和16。如果省略，则默认使用10。
- 返回值的数据类型为VARCHAR。

##### 注意

DUMP命令输出的信息将是完整的内部信息。因此，由于产品更新等原因，这些值可能会发生变化。

##### 示例

在下面的示例中，将返回表t1中列col1的内部信息。

```
SELECT col1, DUMP(col1) FROM t1;
col1 | dump
-----+-----
1001 | Typ=25 Len=8: 32, 0, 0, 0, 49, 48, 48, 49
1002 | Typ=25 Len=8: 32, 0, 0, 0, 49, 48, 48, 50
1003 | Typ=25 Len=8: 32, 0, 0, 0, 49, 48, 48, 51
(3 row)
```

#### H. 56. 5. 7. 2. NLS\_CHARSET\_ID

##### 描述

NLS\_CHARSET\_ID返回与字符集名称字符串对应的字符集ID编号。

##### 语法

```
NLS_CHARSET_ID(str text) returns integer
```

##### 通用规则

- 无效的字符集名称返回null。

注意

- 字符集名称来自于Oracle，请仅使用此函数进行兼容性。
- 如果使用NLS\_CHARSET\_ID函数，则需要提前指定search\_path为“oracle”。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理”>“服务器配置”>“设置参数”。

示例

下面的示例返回字符集的字符集ID:

```
SELECT NLS_CHARSET_ID('AL32UTF8') FROM DUAL;
nls_charset_id
-----
                873
(1 row)
```

### H. 56. 5. 7. 3. NLS\_CHARSET\_NAME

描述

NLS\_CHARSET\_NAME返回与ID编号对应的字符集名称。

语法

```
NLS_CHARSET_NAME(str text) returns integer
```

通用规则

- 如果number不能被识别为有效的字符集ID，则此函数将返回null。

注意

- 字符集名称来自于Oracle，请仅使用此函数进行兼容性。
- 如果使用NLS\_CHARSET\_NAME函数，则需要提前指定search\_path为“oracle”。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理”>“服务器配置”>“设置参数”。

示例

下面的示例返回与字符集ID编号873对应的字符集:

```
SELECT NLS_CHARSET_NAME(873) FROM DUAL;
nls_charset_name
-----
AL32UTF8
```

(1 row)

#### H. 56. 5. 7. 4. SYS\_CONTEXT

##### 描述

返回与上下文命名空间相关联的参数在当前时刻的值。

##### 语法

```
SYS_CONTEXT(namespace text, parameter text, length int4 default 256)
returns text
```

##### 通用规则

- 对于命名空间和参数，可以指定字符串或解析为指定命名空间或属性的字符串的表达式。
- 上下文命名空间必须已经创建，相关联的参数及其值也必须已设置。
- 命名空间必须是有效的标识符。参数名称可以是任何字符串。 它不区分大小写，但长度不能超过4000字节。

##### 注意

LightDB提供以下内置命名空间：

- USERENV - 描述当前会话。 命名空间USERENV的预定义参数列在表Table H.115中。
- SYS\_SESSION\_ROLES - 指示指定的角色当前是否为会话启用。

Table H.115. 命名空间USERENV的预定义参数

参数	概述
CLIENT_IDENTIFIER	数据库会话所使用的程序的名称
CLIENT_INFO	数据库会话所使用的程序的名称
CLIENT_PROGRAM_NAME	数据库会话所使用的程序的名称
CDB_NAME	当前数据库
CON_ID	始终为1
CON_NAME	当前数据库
CURRENT_SCHEMA	当前模式
CURRENT_SCHEMAID	当前模式ID
CURRENT_USER	当前用户
CURRENT_USERID	当前用户ID
DATABASE_ROLE	角色是以下之一： PRIMARY, PHYSICAL STANDBY
DB_NAME	当前数据库
DB_UNIQUE_NAME	当前数据库
HOST	连接客户端的主机名称
INSTANCE	始终为1
INSTANCE_NAME	始终为 'LightDB'
IP_ADDRESS	连接客户端的机器的IP地址

参数	概述
ISDBA	如果用户已被认证为具有DBA特权，则返回TRUE
LANG	语言的缩写名称，比现有的 'LANGUAGE' 参数更简短
LANGUAGE	您的会话当前使用的语言和区域设置，以及数据库字符集
MODULE	数据库会话所使用的程序的名称
NETWORK_PROTOCOL	用于通信的网络协议
NLS_DATE_FORMAT	会话的日期格式
ORACLE_HOME	数据主目录的完整路径名
PID	当前进程ID
SERVER_HOST	listen_addresses
SERVICE_NAME	当前数据库
SESSION_USER	当前用户
SESSION_USERID	当前用户ID
SESSIONID	当前会话进程ID
SID	当前会话进程ID

注意

- 如果使用 SYS\_CONTEXT，则需要提前指定 search\_path 为 "oracle"。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”。

示例

下面的语句返回登录到数据库的用户的名称：

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER') FROM DUAL;
sys_context
-----
lightdb
(1 row)
```

#### H. 56. 5. 7. 5. USERENV

描述

返回有关当前会话的信息

语法

USERENV(parameter text) returns text

通用规则



- 这些信息对于编写特定于应用程序的审计跟踪表或确定当前会话使用的特定于语言的字符可能很有用。
- 表Table H.11描述了参数参数的值。

注意

- 如果使用USERENV，则需要事先指定“oracle”作为search\_path。

参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”。

示例

下面的示例返回当前会话的LANGUAGE参数：

```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL;
Language
-----
en_US.UTF-8
(1 row)
```

### H. 56. 5. 8. 日期时间操作符

orafce支持以下日期类型的日期时间操作符。

Table H.116. 日期时间操作符

操作	示例	结果
+	DATE' 2016/01/01' + 10	2016-01-11 00:00:00
-	DATE' 2016/03/20' - 35	2016-02-14 00:00:00
-	DATE' 2016/09/01' - DATE' 2015/12/31'	245

Note

如果使用orafce的DATE类型的日期时间操作符，则需要事先指定“oracle”作为search\_path。

参见

有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。

### H. 56. 5. 9. 其他函数

以下函数用于其他目的：

- EMPTY\_CLOB
- EMPTY\_BLOB
- ORA\_HASH
- VSIZE

- DEPS\_SAVE\_AND\_DROP\_DEPENDENCIES/DEPS\_RESTORE\_DEPENDENCIES
- PLVSTR. IS\_PREFIX
- PLVDATE. ISLEAPYEAR

#### H. 56. 5. 9. 1. EMPTY\_CLOB

##### 描述

返回一个空的CLOB

##### 语法

```
EMPTY_CLOB() RETURNS clob
```

##### 通用规则

- 可以用来初始化一个 CLOB 变量，或者在 INSERT 或 UPDATE 语句中初始化 CLOB 列为 EMPTY。EMPTY 意味着该 CLOB 已被初始化但没有被赋值。
- 长度为 0 但不为 NULL。

##### 注意

- 如果使用EMPTY\_CLOB函数，则需要事先指定“oracle”作为search\_path。

##### 参见

- 有关如何编辑search\_path的信息，请参阅“使用orafce的注意事项”。
- 有关如何设置服务器参数的信息，请参阅LightDB文档中的“服务器管理”>“服务器配置”>“设置参数”。

##### 示例

```
select length(empty_clob()) FROM DUAL;
length
-----
      0
(1 row)
```

```
select count(*) from dual where empty_clob() is null;
count
-----
      0
(1 row)
```

```
create table foo (a int, b clob default empty_clob());
insert into foo(a) values(1);
insert into foo values (2, 'hello');
select count(*) from foo where b is not null;
count
-----
      2
```

```
(1 row)
select count(*) from foo where length(b) > 0;
count
-----
      1
(1 row)
```

#### H. 56. 5. 9. 2. EMPTY\_BLOB

描述

返回一个空的 BLOB

语法

EMPTY\_BLOB() RETURNS blob

通用规则

- 可以用来初始化一个 BLOB 变量，或者在 INSERT 或 UPDATE 语句中初始化 BLOB 列为 EMPTY。EMPTY 意味着该 BLOB 已被初始化但没有被赋值。
- 长度为 0 但不为 NULL

样例

```
select length(empty_blob()) FROM DUAL;
length
-----
      0
(1 row)
```

```
select count(*) from dual where empty_blob() is null;
count
-----
      0
(1 row)
```

```
create table foo (a int, b blob default empty_blob());
insert into foo(a) values(1);
insert into foo values (2, to_blob('616263'));
select count(*) from foo where b is not null;
count
-----
      2
(1 row)
select count(*) from foo where length(b) > 0;
count
-----
      1
(1 row)
```

### H. 56. 5. 9. 3. ORA\_HASH

#### 描述

ORA\_HASH是一个计算给定表达式哈希值的函数。此函数可用于操作，例如分析数据子集和生成随机样本。

#### 语法

```
ORA_HASH(p_data anyelement, p_buckets int4, p_seed int4) RETURNS int
```

#### 通用规则

- 参数p\_data确定要计算哈希值的数据。参数expr通常表示列名，对于其长度没有限制。
- 可选参数p\_buckets确定哈希函数返回的最大桶值。您可以指定0到2147483647之间的任何值。默认值为0。
- 可选参数p\_seed使LightDB能够为相同的数据集生成许多不同的结果。LightDB将哈希函数应用于expr和p\_seed的组合。您可以指定0到2147483647之间的任何值。默认值为0。
- 返回值的数据类型为int。

#### 注意

- 如果使用ORA\_HASH，则需要事先指定“oracle”作为search\_path。

#### 参见

- 请参考“使用orafce的注意事项”以获取如何编辑search\_path的信息。
- 请参考LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”章节，以获取如何设置服务器参数的信息。

#### 示例

```
select ora_hash('abcdAbcdasd'::text) FROM DUAL;
ora_hash
-----
303228277
(1 row)
```

### H. 56. 5. 9. 4. VSIZE

#### 描述

VSIZE函数返回expr内部表示的字节数。如果expr为null，则该函数返回null。

#### 语法

```
VSIZE(anyelement) RETURNS integer
```

#### 通用规则

- pg\_column\_size函数与之相同。

#### 注意

- 如果使用VSIZE函数，则需提前将search\_path指定为“oracle”。

参见

- 请参考“使用orafce的注意事项”以获取如何编辑search\_path的信息。
- 请参考LightDB文档中的“服务器管理” > “服务器配置” > “设置参数”章节，以获取如何设置服务器参数的信息。

示例

```
select vsize(123456789) FROM DUAL;
vsize
-----
      4
(1 row)
```

#### H. 56. 5. 9. 5. DEPS\_SAVE\_AND\_DROP\_DEPENDENCIES/DEPS\_RESTORE\_DEPENDENCIES

描述

这些函数用于修改具有视图依赖关系的表。递归备份所有依赖视图，然后修改基表，最后重新创建所有备份视图。

DEPS\_SAVE\_AND\_DROP\_DEPENDENCIES函数用于保存和删除表的依赖关系。

DEPS\_RESTORE\_DEPENDENCIES函数用于恢复表的依赖关系。

语法

```
DEPS_SAVE_AND_DROP_DEPENDENCIES(name, name default current_schema()::name,
jsonb default '{}'::jsonb) RETURNS void
DEPS_RESTORE_DEPENDENCIES(name, name default current_schema()::name, jsonb
default '{}'::jsonb)
```

通用规则

- 在lt\_catalog下创建。
- 依赖关系保存在lt\_catalog.deps\_saved\_ddl表中。
- dry\_run参数用于不实际删除依赖关系运行函数，默认值为false。
- verbose参数用于显示调试日志，默认值为false。
- populate\_materialized\_view参数用于启用或禁用使用WITH [NO] DATA标志刷新创建的物化视图，默认值为false。

注意

- 如果使用DEPS\_SAVE\_AND\_DROP\_DEPENDENCIES/DEPS\_RESTORE\_DEPENDENCIES函数，则无需提前将search\_path指定为“oracle”。

示例

```
create table test_t(key1 int);
create view test_t_v as select * from test_t;
```

```
begin;
select deps_save_and_drop_dependencies('test_t');
alter table test_t modify key1 number;
select * from deps_restore_dependencies('test_t');
commit;
```

#### H. 56. 5. 9. 6. PLVSTR. IS\_PREFIX

##### 描述

PLVSTR.IS\_PREFIX会在一个字符串或数字是另一个字符串或数字的前缀时返回true。

##### 语法

```
PLVSTR.IS_PREFIX(p bigint, prefix bigint) RETURNS boolean
PLVSTR.IS_PREFIX(p integer, prefix integer) RETURNS boolean
PLVSTR.IS_PREFIX(p text, prefix text) RETURNS boolean
```

##### 通用规则

- 这个功能在测试字符串或数字是否以某些内容开头时非常有用。

##### 例子

```
select plvstr.is_prefix(111, 11) FROM DUAL;
is_prefix
-----
t
(1 row)
select plvstr.is_prefix('abc', 'AB') from dual;
is_prefix
-----
f
(1 row)
```

#### H. 56. 5. 9. 7. PLVDATE. ISLEAPYEAR

##### 描述

判断某个年份是否是闰年

##### 语法

```
PLVDATE.ISLEAPYEAR(date) RETURNS boolean
```

##### 示例

```
select plvdate.isleapyear(date '2023-01-01') FROM DUAL;
isleapyear
-----
f
select plvdate.isleapyear(date '2020-01-01') from dual;
```

```
isleapyear
-----
t
```

## H. 56. 6. 包参考

“包”是由模式组合在一起的一组功能，具有单一的功能，并通过从PL/pgSQL调用来使用。

支持以下包：

- DBMS\_ALERT
- DBMS\_ASSERT
- DBMS\_LOB
- DBMS\_OUTPUT
- DBMS\_PIPE
- DBMS\_RANDOM
- DBMS\_UTILITY
- DBMS\_JOB
- DBMS\_LOCK
- DBMS\_METADATA
- DBMS\_OBFUSCATION\_TOOLKIT
- DBMS\_SNAPSHOT
- UTL\_FILE
- UTL\_RAW
- UTL\_URL
- UTL\_ENCODE

要从PL/pgSQL调用不同的功能，请使用PERFORM语句或SELECT语句，并使用包名称限定功能名称。请参考每个包功能的解释，以获取有关调用格式的信息。

### H. 56. 6. 1. DBMS\_ALERT

概述

DBMS\_ALERT包可将警报从一个PL/pgSQL会话发送到多个其他PL/pgSQL会话。

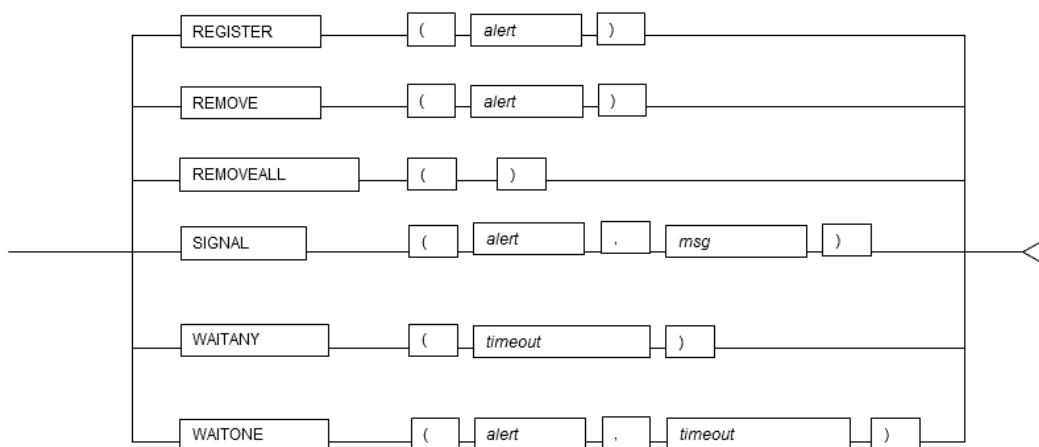
当进行1:N处理时，可以使用此包，例如在同时将警报从给定的PL/pgSQL会话通知到另一个PL/pgSQL会话时。

Table H. 117. DBMS\_ALERT功能

特性	描述
REGISTER	注册指定的警报。
REMOVE	删除指定的警报。

特性	描述
REMOVEALL	从会话中删除所有警报。
SIGNAL	通知警报。
WAITANY	等待会话注册的任何警报的通知。
WAITONE	等待会话注册的特定警报的通知。

语法



#### H. 56. 6. 1. 1. 功能说明

本节将说明DBMS\_ALERT的每个功能。

##### REGISTER

- REGISTER函数将指定的警报注册到会话中。通过将警报注册到会话中，可以接收到SIGNAL通知。
- 指定警报的名称。
- 警报区分大小写。
- 可以在单个会话中注册多个警报。如果注册多个警报，请为每个警报调用REGISTER函数。

示例

```
PERFORM DBMS_ALERT.REGISTER('sample_alert');
```

##### REMOVE

- REMOVE 从会话中移除指定警报。
- 指定警报的名称。
- 警报区分大小写。
- 警报留下的消息将被移除。

示例

```
PERFORM DBMS_ALERT.REMOVE('sample_alert');
```



#### REMOVEALL

- REMOVEALL 从会话中移除所有已注册的警报。
- 所有警报留下的消息都将被移除。

#### 示例

```
PERFORM DBMS_ALERT.REMOVEALL();
```

#### SIGNAL

- SIGNAL 发送指定警报的消息通知。
- 指定要发送消息通知的警报名称。
- 警报区分大小写。
- 在消息中，指定要通知的警报消息。
- 当执行 SIGNAL 时，消息通知并不完整。消息通知是在提交事务时发送的。如果在 SIGNAL 执行后执行回滚，则消息通知会被丢弃。
- 如果来自多个会话的同一警报发送了消息通知，则消息将被累积而不被移除。

#### 示例

```
PERFORM DBMS_ALERT.SIGNAL('ALERT001', 'message001');
```

#### 注意

如果连续发出 SIGNAL，累积的消息超过一定数量，则可能会输出内存不足错误。如果内存不足，请调用 AITANY 或 WAITONE 来接收警报，并减少累积的消息。

#### WAITANY

- WAITANY 等待会话中任何已注册的警报的通知。
- 指定等待警报的最大等待时间 timeout（以秒为单位）。
- 使用 SELECT 语句获取通知信息，该信息存储在名称、消息和状态列中。
- 名称列存储警报名称。名称的数据类型为 TEXT。
- 消息列存储通知警报的消息。消息的数据类型为 TEXT。
- 状态列存储操作返回的状态代码：0-发生警报；1-发生超时。状态的数据类型为 INTEGER。

#### 示例

```
DECLARE
    alert_name      TEXT := 'sample_alert';
    alert_message   TEXT;
    alert_status    INTEGER;
BEGIN
```

```
SELECT name,message,status INTO alert_name,alert_message,alert_status
FROM DBMS_ALERT.WAITANY(60);
```

WAITONE

- WAITONE 等待指定警报的通知。
- 指定要等待的警报名称。
- 警报区分大小写。
- 指定等待警报的最大等待时间 timeout（以秒为单位）。
- 使用 SELECT 语句获取通知信息，该信息存储在消息和状态列中。
- 消息列存储通知警报的消息。消息的数据类型为 TEXT。
- 状态列存储操作返回的状态代码：0-发生警报；1-发生超时。状态的数据类型为 INTEGER。

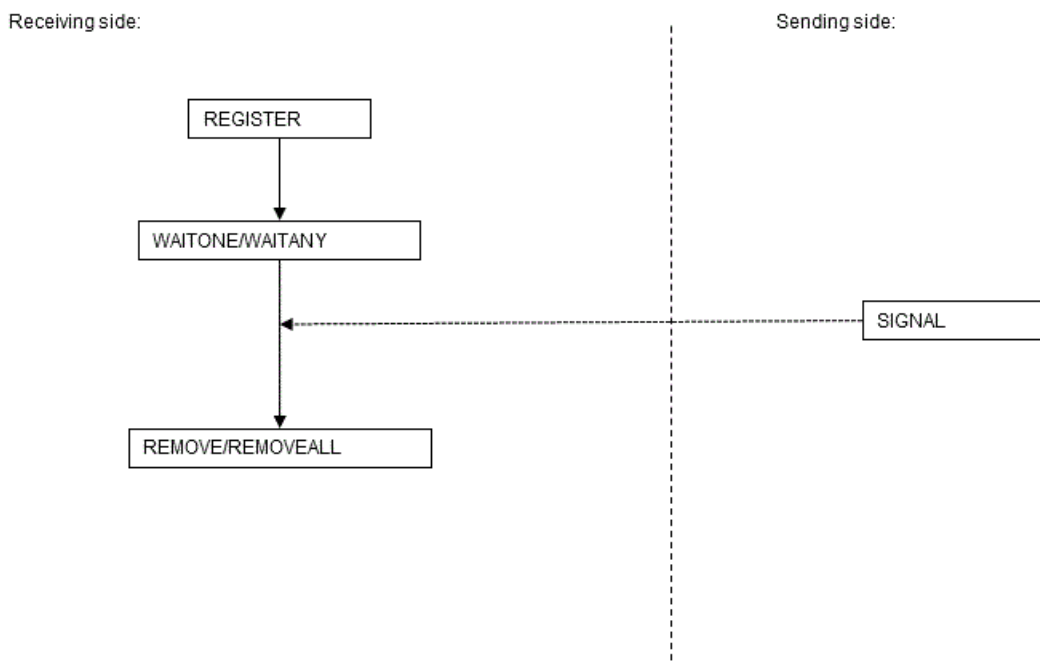
示例

```
DECLARE
    alert_message TEXT;
    alert_status  INTEGER;
BEGIN
    SELECT message,status INTO alert_message,alert_status FROM
    DBMS_ALERT.WAITONE('sample_alert', 60);
```

#### H. 56. 6. 1. 2. 使用示例

下面是 DBMS\_ALERT 处理流程的使用示例。

DBMS\_ALERT 流程



注意

- SIGNAL 发送的消息通知的目标是在执行 SIGNAL 时执行 REGISTER 的会话。
- 在接收端，始终确保使用 REMOVE 或 REMOVEALL 尽快删除警报，以便在不再需要警报时将其删除。如果关闭会话而没有删除警报，则可能无法在另一个会话中接收到同名警报的 SIGNAL。
- DBMS\_ALERT 和 DBMS\_PIPE 使用相同的内存环境。因此，当检测到 DBMS\_PIPE 的内存不足时，也可能会检测到 DBMS\_ALERT 的内存不足。

使用示例

- 发送方

```
CREATE FUNCTION send_dbms_alert_exe() RETURNS VOID AS $$
BEGIN
    PERFORM DBMS_ALERT.SIGNAL('sample_alert','SIGNAL ALERT');
END;
$$ LANGUAGE plpgsql;
SELECT send_dbms_alert_exe();
DROP FUNCTION send_dbms_alert_exe();
```

- 接收方

```
CREATE FUNCTION receive_dbms_alert_exe() RETURNS VOID AS $$
DECLARE
    alert_name TEXT := 'sample_alert';
    alert_message TEXT;
    alert_status INTEGER;
BEGIN
    PERFORM DBMS_ALERT.REGISTER(alert_name);
    SELECT message, status INTO alert_message, alert_status FROM
DBMS_ALERT.WAITONE(alert_name, 300);
    RAISE NOTICE 'Message : %', alert_message;
    RAISE NOTICE 'Status : %', alert_status;
    PERFORM DBMS_ALERT.REMOVE(alert_name);
END;
$$ LANGUAGE plpgsql;
SELECT receive_dbms_alert_exe();
DROP FUNCTION receive_dbms_alert_exe();
```

## H. 56. 6. 2. DBMS\_ASSERT

概述

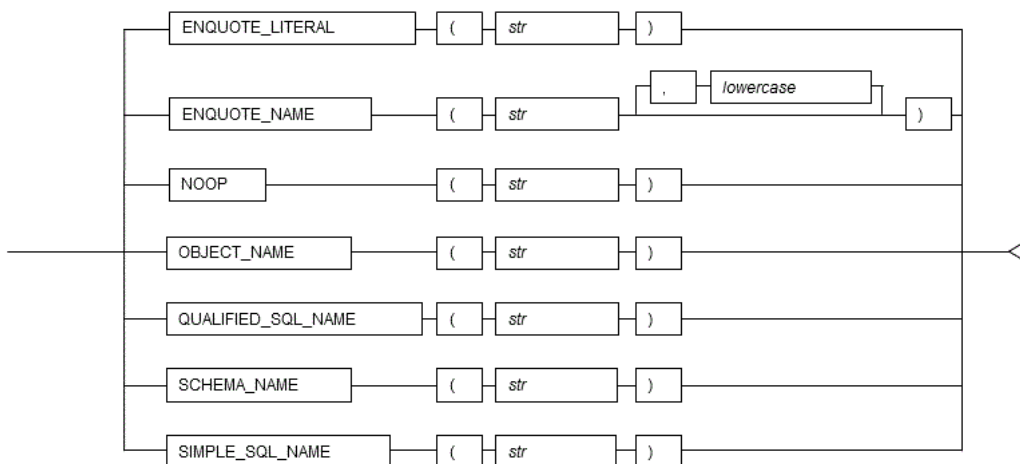
在 PL/pgSQL 中执行输入值属性的验证。

Table H. 118. DBMS\_ASSERT 特性

特性	描述
ENQUOTE_LITERAL	返回用单引号括起来的指定字符串。
ENQUOTE_NAME	返回用双引号括起来的指定字符串。
NOOP	按原样返回指定的字符串。

特性	描述
OBJECT_NAME	验证指定的字符串是否为已定义的标识符。
QUALIFIED_SQL_NAME	验证指定的字符串是否以适当的格式作为标识符。
SCHEMA_NAME	验证指定的字符串是否为已定义的模式。
SIMPLE_SQL_NAME	验证指定的字符串是否以适当的格式作为单个标识符。

语法



### H. 56. 6. 2. 1. 特性描述

本节介绍了 DBMS\_ASSERT 的每个特性。

#### ENQUOTE\_LITERAL

- ENQUOTE\_LITERAL 返回用单引号括起来的指定字符串。
- 指定一个用单引号括起来的字符串。
- 返回值的数据类型是VARCHAR。

示例

```

DECLARE
    q_literal    VARCHAR(256);
BEGIN
    q_literal := DBMS_ASSERT.ENQUOTE_LITERAL(' literal_word');

```

#### ENQUOTE\_NAME

- ENQUOTE\_NAME将指定的字符串用双引号括起来返回。
- 指定用双引号括起来的字符串。
- 对于小写字母转换，请指定TRUE或FALSE。指定TRUE将字符串中的大写字母转换为小写字母。如果指定FALSE，则不会进行小写转换。默认值为TRUE。
- 如果字符串中的所有字符都是小写，则不会用双引号括起来。

- 返回值的数据类型是VARCHAR。

参见

有关布尔类型（TRUE/FALSE）值的信息，请参阅LightDB文档中的“[The Language](#)” > “[Data Types](#)” > “[Boolean Type](#)”。 SQL

示例

```
DECLARE
    dq_literal    VARCHAR(256);
BEGIN
    dq_literal := DBMS_ASSERT.ENQUOTE_NAME(' TBL001');
```

NOOP

- NOOP原样返回指定的字符串。
- 指定一个字符串。
- 返回值的数据类型是VARCHAR。

示例

```
DECLARE
    literal    VARCHAR(256);
BEGIN
    literal := DBMS_ASSERT.NOOP(' NOOP_WORD');
```

OBJECT\_NAME

- OBJECT\_NAME验证指定的字符串是否为已定义的标识符。
- 指定要验证的标识符。如果已定义该标识符，则返回指定的标识符。否则，将出现以下错误。

```
ERROR:  invalid object name
```

- 返回值的数据类型是VARCHAR。

示例

```
DECLARE
    object_name    VARCHAR(256);
BEGIN
    object_name := DBMS_ASSERT.OBJECT_NAME(' SCM001. TBL001');
```

QUALIFIED\_SQL\_NAME

- QUALIFIED\_SQL\_NAME验证指定的字符串是否符合标识符的适当格式。
- 指定要验证的标识符。如果字符串可以用作标识符，则将返回指定的标识符。否则，将出现以下错误。

```
ERROR: string is not qualified SQL name
```

- 返回值的数据类型是VARCHAR。

参见

关于可以用作标识符的格式的信息，请参阅LightDB文档中的“[The Language](#)” > “[Lexical Structure](#)” > “[Identifiers and Key Words](#)”。 SQL

示例

```
DECLARE
    object_name    VARCHAR(256);
BEGIN
    object_name := DBMS_ASSERT.QUALIFIED_SQL_NAME('SCM002.TBL001');
```

SCHEMA\_NAME

- SCHEMA\_NAME验证指定的字符串是否为已定义的模式。
- 指定要验证的模式名称。如果已定义该模式，则返回指定的模式名称。否则，将出现以下错误。

```
ERROR: invalid schema name
```

- 返回值的数据类型是VARCHAR。

示例

```
DECLARE
    schema_name    VARCHAR(256);
BEGIN
    schema_name := DBMS_ASSERT.SCHEMA_NAME('SCM001');
```

SIMPLE\_SQL\_NAME

- SIMPLE\_SQL\_NAME验证指定的字符串是否符合单个标识符的适当格式。
- 指定要验证的标识符。如果指定的字符串可以用作标识符，则返回指定的标识符。否则，将出现以下错误。

```
ERROR: string is not qualified SQL name
```

- 返回值的数据类型是VARCHAR。

参见

关于可以用作标识符的格式的信息，请参阅LightDB文档中的“[The Language](#)” > “[Lexical Structure](#)” > “[Identifiers and Key Words](#)”。请注意，如果指定了使用全角字符的标识符，则会出现错误。如果包含全角字符，请指定带引号的标识符。

示例

```
DECLARE
    simple_name    VARCHAR(256);
BEGIN
    simple_name := DBMS_ASSERT.SIMPLE_SQL_NAME('COL01');
```

### H. 56. 6. 2. 2. 用法示例

下面是 DBMS\_ASSERT 的使用示例。

```
CREATE FUNCTION dbms_assert_exe() RETURNS VOID AS $$
DECLARE
    w_schema VARCHAR(20) := 'public';
    w_table  VARCHAR(20) := 'T1';
    w_object VARCHAR(40);
BEGIN
    PERFORM DBMS_ASSERT.NOOP(w_schema);
    PERFORM DBMS_ASSERT.SIMPLE_SQL_NAME(w_table);
    PERFORM DBMS_ASSERT.SCHEMA_NAME(w_schema);
    w_object := w_schema || '.' || w_table;
    PERFORM DBMS_ASSERT.QUALIFIED_SQL_NAME(w_object);
    PERFORM DBMS_ASSERT.OBJECT_NAME(w_object);
    RAISE NOTICE 'OBJECT      : %', DBMS_ASSERT.ENQUOTE_LITERAL(w_object);
    RAISE NOTICE 'TABLE_NAME : %', DBMS_ASSERT.ENQUOTE_NAME(w_table);
END;
$$
LANGUAGE plpgsql;
SELECT dbms_assert_exe();
DROP FUNCTION dbms_assert_exe();
```

### H. 56. 6. 3. DBMS\_LOB

概述

一些与 DBMS\_LOB 包兼容的函数。

Table H. 119. DBMS\_LOB 功能

功能	描述
INSTR	返回指定 LOB 的起始索引。
SUBSTR	提取 LOB 的子字符串。
GETLENGTH	返回 LOB 中字符的数量。
APPEND	将源 LOB 的内容追加到目标 LOB 中。
CLOSE	关闭之前打开的内部或外部 LOB。(已作废)
COMPARE	比较两个完整的 LOB 或两个 LOB 的部分。
COPY	将源 LOB 的全部或部分内容复制到目标 LOB 中。
CREATETEMPORAR	在用户默认的临时表空间中创建一个临时 BLOB 或 CLOB 及其相应的索引。(已作废)
ERASE	擦除一个LOB的全部或部分。

功能	描述
FREETEMPORARY	释放一个LOB的全部或部分临时空间。
OPEN	在默认的临时表空间中释放临时BLOB或CLOB。（无效）
READ	从指定偏移量开始读取LOB中的数据。
TRIM	将LOB值修剪为指定的较短长度。
WRITE	从指定偏移量开始向LOB写入数据。
WRITEAPPEND	将缓冲区写入LOB的末尾。
COMPARE	比较两个完整的LOB或两个LOB的部分。
COMPARE	比较两个完整的LOB或两个LOB的部分。
COMPARE	比较两个完整的LOB或两个LOB的部分。

### 语法

```

INSTR(STR TEXT, PATT TEXT, START INTEGER DEFAULT 1, NTH INTEGER DEFAULT 1)
RETURNS INTEGER
INSTR(STR BLOB, PATT TEXT, START INTEGER DEFAULT 1, NTH INTEGER DEFAULT 1)
RETURNS INTEGER
SUBSTR(STR CLOB, AMOUNT INTEGER DEFAULT 32767, START_OFFSET INTEGER DEFAULT
1) RETURNS VARCHAR2
SUBSTR(STR BLOB, AMOUNT INTEGER DEFAULT 32767, START_OFFSET INTEGER DEFAULT
1) RETURNS RAW
GETLENGTH(STR CLOB) RETURNS INTEGER
GETLENGTH(STR BLOB) RETURNS INTEGER
APPEND(dest_lob BLOB, src_lob BLOB)/APPEND(CLOB, CLOB)
COMPARE(lob_1 BLOB, lob_2 BLOB, amount INT4 DEFAULT 2147483647, offset_1 INT4
DEFAULT 1, offset_2 INT4 DEFAULT 1) RETURNS INTEGER
COMPARE(CLOB, CLOB, INT4 DEFAULT 2147483647, INT4 DEFAULT 1, INT4 DEFAULT 1)
RETURNS INTEGER
COPY(dest_lob BLOB, src_lob BLOB, amount INT4, dest_offset INT4 DEFAULT
1, src_offset INT4 DEFAULT 1)/COPY(CLOB, CLOB, INT4, INT4 DEFAULT 1, INT4 DEFAULT 1)
ERASE(lob_loc BLOB, amount INT4, p_offset INT4 DEFAULT 1)/
ERASE(CLOB, INT4, INT4 DEFAULT 1)
READ(lob_loc BLOB, amount INT4, p_offset INT4, buffer RAW)/
READ(CLOB, INT4, INT4, TEXT)
TRIM(lob_loc BLOB, newlen INT4)/TRIM(CLOB, INT4)
WRITE(lob_loc BLOB, amount INT4, p_offset INT4, buffer RAW)/
WRITE(CLOB, INT4, INT4, TEXT)
WRITEAPPEND(lob_loc BLOB, amount INT4, buffer RAW)/
WRITEAPPEND(CLOB, INT4, TEXT)

```

### 特性

#### H. 56. 6. 3. 1. 描述

本节介绍了DBMS\_LOB的每个特性。

#### INSTR

- 匹配模式的起始偏移量，以字节或字符为单位。如果未找到模式，则返回0。
- 如果任何一个或多个输入参数为NULL或INVALID，则返回null。
- 如果offset < 1，则返回NULL。



例子

```
select dbms_lob.INSTR('Thomas', 'om') from dual;
select dbms_lob.INSTR('Thomas'::blob, utl_raw.CAST_TO_RAW('om')) from
dual;
```

SUBSTR

- 如果指定了start和count，则从start位置开始提取字符串的子字符串，并在count个字符后停止。
- 如果任何输入参数为NULL，则返回NULL。
- 如果amount < 1，则返回NULL。
- 如果offset < 1，则返回NULL。

例子

```
select dbms_lob.substr('1234567',1,2) from dual;
select dbms_lob.substr('1234567'::blob,1,2) from dual;
```

GETLENGTH

- 文本的长度，以字节或字符的形式表示为INTEGER。
- 如果输入的str为NULL，则返回NULL。

例子

```
select DBMS_LOB.GETLENGTH('abc') from dual;
```

APPEND

- 将源LOB的内容附加到目标LOB中。

例子

```
declare
    l_blob          blob := 'abcd'::blob;
    l_clob          clob := 'abcd'::clob;
begin
    dbms_output.serveroutput(true);
    dbms_lob.append(l_blob, 'ext'::blob);
    dbms_output.put_line(l_blob::text);

    dbms_lob.append(l_clob, 'ext'::clob);
    dbms_output.put_line(l_clob);
end;
/
```

COMPARE

- 比较两个完整的LOB或两个LOB的部分。
- 返回 0 如果 `lob\_1` 和 `lob\_2` 相等，返回 -1 如果 `lob\_1` 比 `lob\_2` 小，返回 1 如果 `lob\_1` 比 `lob\_2` 大。
- 返回 NULL 如果amount、offset\_1或offset\_2不是有效的LOB偏移值。有效偏移值的范围为1至LOBMAXSIZE（包括1和LOBMAXSIZE）。

例子

```

declare
    l_result          int;
    l_blob            blob := 'abcd'::blob;
    l_clob            clob := 'abcd'::clob;
    l_raw_result      raw;
    l_amout           INTEGER := 100;
    l_text_result     text;
begin
    dbms_output.serveroutput(true);
    l_result := dbms_lob.compare('abcd'::blob, 'abcd'::blob);
    dbms_output.put_line(l_result);
    l_result := dbms_lob.compare('abcd'::clob, 'abcd'::clob);
    dbms_output.put_line(l_result);

end;
/

```

COPY

- 将源LOB的全部或部分内容复制到目标LOB中。 您可以指定源LOB和目标LOB的偏移量，以及要复制的字节数或字符数。
- 如果您在目标LOB中指定的偏移量超出了当前LOB中的数据末尾，则会在目标BLOB或CLOB中插入零字节填充或空格。如果偏移量小于目标LOB的当前长度，则现有数据将被覆盖。

例子

```

declare
    l_result          int;
    l_blob            blob := 'abcd'::blob;
    l_clob            clob := 'abcd'::clob;
    l_raw_result      raw;
    l_amout           INTEGER := 100;
    l_text_result     text;
begin
    dbms_output.serveroutput(true);
    dbms_lob.copy(l_blob, 'new_blob'::blob, 100);
    dbms_output.put_line(l_blob::text);
    dbms_lob.copy(l_blob, 'xnew_blob'::blob, 100, 1, 2);
    dbms_output.put_line(l_blob::text);
    dbms_lob.copy(l_blob, 'xnew_blob'::blob, 100, 2, 2);
    dbms_output.put_line(l_blob::text);

end;
/

```

ERASE

- 擦除整个内部LOB或部分内部LOB。
- 当从LOB中间擦除数据时，会写入BLOB或CLOB的零字节填充或空格。
- 如果在擦除指定数量之前到达LOB值的末尾，则实际擦除的字节数或字符数可能与您在amount参数中指定的数量不同。实际擦除的字符或字节数将在amount参数中返回。

例子

```

declare
    l_result      int;
    l_blob        blob := 'abcd'::blob;
    l_clob        clob := 'abcd'::clob;
    l_raw_result  raw;
    l_amout       INTEGER := 100;
    l_text_result text;
begin
    dbms_output.serveroutput(true);

    l_blob := 'abcd'::blob;
    l_amout := 100;
    dbms_output.put_line(l_blob::text);
    dbms_lob.erase(l_blob, l_amout);
    dbms_output.put_line(l_blob::text);

    l_clob := 'abcd'::clob;
    l_amout := 4;
    dbms_lob.erase(l_clob, l_amout);
    dbms_output.put_line(l_clob);
end;
/

```

READ

- 此过程读取LOB的一部分，并将指定的amount从LOB开头的绝对偏移量开始返回到缓冲区参数中。
- 实际读取的字节数或字符数将在amount参数中返回。如果输入的偏移量指向LOB的末尾，则将amount设置为0，并引发NO\_DATA\_FOUND异常。

例子

```

declare
    l_result      int;
    l_blob        blob := 'abcde'::blob;
    l_clob        clob := 'abcde'::clob;
    l_raw_result  raw;
    l_amout       INTEGER := 100;
    l_text_result text;
begin
    dbms_output.serveroutput(true);

    l_amout := 5;
    dbms_lob.read('abcde'::blob, l_amout, 1, l_raw_result);
    dbms_output.put_line(l_raw_result);

```

```

        l_amount := 5;
        dbms_lob.read('abcde'::clob, l_amount, 1, l_text_result);
        dbms_output.put_line(l_text_result);
    end;
/

```

#### TRIM

- 此过程将内部LOB的值修剪为您在newlen参数中指定的长度。对于BLOB，请以字节为单位指定长度；对于CLOB，请以字符为单位指定长度。

#### 例子

```

declare
    l_result          int;
    l_blob            blob := 'abcd'::blob;
    l_clob            clob := 'abcd'::clob;
    l_raw_result     raw;
    l_amount         INTEGER := 100;
    l_text_result    text;
begin
    dbms_output.serveroutput(true);

    l_blob := 'abcde'::blob;
    dbms_lob.trim(l_blob, 1);
    dbms_output.put_line(l_blob::text);

    l_clob := 'abcde'::clob;
    dbms_lob.trim(l_clob, 1);
    dbms_output.put_line(l_clob);
end;
/

```

#### WRITE

- 此过程从LOB开头的绝对偏移量开始，将指定数量的数据从缓冲区参数中写入内部LOB。数据从缓冲区参数中写入。
- WRITE将替换（覆盖）在指定偏移量上已存在的LOB中的任何数据，替换的长度由您指定。

#### 例子

```

declare
    l_result          int;
    l_blob            blob := 'abcd'::blob;
    l_clob            clob := 'abcd'::clob;
    l_raw_result     raw;
    l_amount         INTEGER := 100;
    l_text_result    text;
begin
    dbms_output.serveroutput(true);

    l_blob := 'abcde'::blob;
    dbms_lob.write(l_blob, pg_catalog.length(utl_raw.CAST_TO_RAW('测试')), 2,
    utl_raw.CAST_TO_RAW('测试'));
    dbms_output.put_line(l_blob::text);
end;

```

```

-- write clob
l_clob := 'abcde'::clob;
dbms_lob.write(l_clob, 2, 2, '测试'::text);
dbms_output.put_line(l_clob);
end;
/

```

#### WRITEAPPEND

- 此过程将指定数量的数据从缓冲区参数中写入内部LOB的末尾。

#### 例子

```

declare
    l_result      int;
    l_blob        blob := 'abcd'::blob;
    l_clob        clob := 'abcd'::clob;
    l_raw_result  raw;
    l_amout       INTEGER := 100;
    l_text_result text;
begin
    dbms_output.serveroutput(true);

    -- writeappend blob
    l_blob := 'abcde'::blob;
    dbms_lob.writeappend(l_blob, pg_catalog.length(utl_raw.CAST_TO_RAW('测试')),
utl_raw.CAST_TO_RAW('测试'));
    dbms_output.put_line(l_blob::text);

    -- write clob
    l_clob := 'abcde'::clob;
    dbms_lob.writeappend(l_clob, 2, '测试'::text);
    dbms_output.put_line(l_clob);
end;
/

```

#### H. 56. 6. 3. 2. 使用示例

下面展示了DBMS\_LOB的使用示例。

```

CREATE PROCEDURE dbms_lob_exe() AS $$
DECLARE
    str1 text;
    str2 text;
    pos INTEGER;
    count INTEGER;
BEGIN
    str2 := dbms_lob.substr('1234567', 2, 3);
    pos := dbms_lob.instr('Tech on the net', 'e');
    count := dbms_lob.getlength('1234567');
    raise info 'str2=%, pos=%, count=%', str2, pos, count;
END;
$$ LANGUAGE plpgsql;
call dbms_lob_exe();

```

```
DROP PROCEDURE dbms_lob_exe();
```

### H. 56. 6. 4. DBMS\_OUTPUT

概述

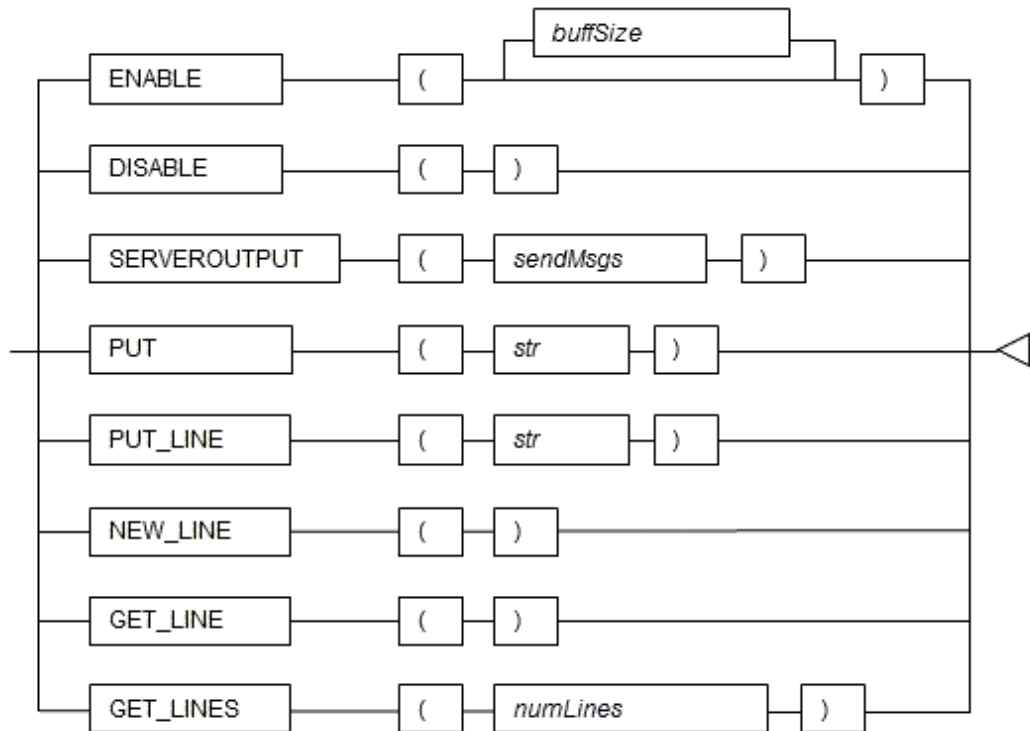
从PL/pgSQL向客户端（如1tsql）发送消息。

特点

Table H. 120. DBMS\_OUTPUT 特点

特点	描述
ENABLE	启用此包的功能。
DISABLE	禁用此包的功能。
SERVEROUTPUT	控制是否发送消息。
PUT	发送消息。
PUT_LINE	发送带换行符的消息。
NEW_LINE	发送一个换行符。不能带参数,支持两种使用方式: dbms_output.new_line和 dbms_output.new_line()。
GET_LINE	从消息缓冲区获取一行。
GET_LINES	从消息缓冲区检索多行。

语法



#### H. 56. 6. 4. 1. 描述

本节详细说明了DBMS\_OUTPUT的每个功能。

#### ENABLE

- ENABLE启用PUT、PUT\_LINE、NEW\_LINE、GET\_LINE和GET\_LINES的使用。
- 在多次执行ENABLE时，最后指定的值是缓冲区大小（以字节为单位）。请指定2000到1000000之间的缓冲区大小。
- 缓冲区大小的默认值为20000。如果将NULL指定为缓冲区大小，则将使用1000000。
- 如果没有执行ENABLE，则即使执行PUT、PUT\_LINE、NEW\_LINE、GET\_LINE和GET\_LINES，它们也会被忽略。

#### 例子

```
PERFORM DBMS_OUTPUT.ENABLE(20000);
```

#### DISABLE

- DISABLE禁用PUT、PUT\_LINE、NEW\_LINE、GET\_LINE和GET\_LINES的使用。
- 剩余的缓冲区信息将被丢弃。

#### 例子

```
PERFORM DBMS_OUTPUT.DISABLE();
```

#### SERVEROUTPUT

- SERVEROUTPUT控制是否发送消息。
- 对于sendMsgs，请指定TRUE或FALSE。
- 如果指定为TRUE，则在执行PUT、PUT\_LINE或NEW\_LINE时，将消息发送到客户端（如1tsql），而不是存储在缓冲区中。
- 如果指定为FALSE，则在执行PUT、PUT\_LINE或NEW\_LINE时，将消息存储在缓冲区中，而不发送到客户端（如1tsql）。

#### 参见

请参阅LightDB文档中的“The SQL Language” > “Data Types” > “Boolean Type”以获取有关布尔类型（TRUE/FALSE）值的信息。

#### 例子

```
PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);
```

#### PUT

- PUT设置要发送的消息。
- 字符串是要发送的消息。
- 当为SERVEROUTPUT指定TRUE时，消息将发送到客户端，例如1tsql。
- 当SERVEROUTPUT指定为FALSE时，消息将保留在缓冲区中。

- PUT不会附加换行符。要附加换行符，请执行NEW\_LINE。
- 如果发送的字符串长度超过了在ENABLE中指定的缓冲区大小，则会发生错误。

示例

```
PERFORM DBMS_OUTPUT.PUT(' abc');
```

PUT\_LINE

- PUT\_LINE将要发送的信息设置为末尾加上一个换行符。
- 字符串是要发送的信息。
- 当在SERVEROUTPUT中指定TRUE时，消息将被发送给客户端，如ltsql。
- 当在SERVEROUTPUT中指定FALSE时，消息将被保留在缓冲区中。
- 如果发送的字符串长度超过了ENABLE中指定的缓冲区大小，就会发生错误。

示例

```
PERFORM DBMS_OUTPUT.PUT_LINE(' abc');
```

NEW\_LINE

- NEW\_LINE在使用PUT创建消息时在末尾添加一个换行符。
- 当在SERVEROUTPUT中指定TRUE时，消息将被发送给客户端，如ltsql。
- 当在SERVEROUTPUT中指定FALSE时，消息将被保留在缓冲区中。

示例

```
PERFORM DBMS_OUTPUT.NEW_LINE();
```

GET\_LINE

- GET\_LINE从消息缓冲区中检索一行。
- 使用SELECT语句获取检索到的行和操作返回的状态码，它们存储在line和status列中。
- line列存储从缓冲区检索到的行。line的数据类型为TEXT。
- status列存储操作返回的状态码：0-成功完成；1-因缓冲区中没有更多行而失败。status的数据类型为INTEGER。
- 如果执行GET\_LINE或GET\_LINES，然后执行PUT、PUT\_LINE或PUT\_LINES，而缓冲区中仍存在未检索的消息，则未从缓冲区检索到的消息将被丢弃。

示例

```
DECLARE  
    buff1    VARCHAR(20);
```



```

        stts1    INTEGER;
BEGIN
    SELECT line,status INTO buff1,stts1 FROM DBMS_OUTPUT.GET_LINE();

```

#### GET\_LINES

- GET\_LINES从消息缓冲区中检索多行信息。
- 指定从缓冲区中检索的行数。
- 使用SELECT语句获取检索到的行和行数，它们存储在lines和numlines列中。
- lines列存储从缓冲区检索到的行。lines的数据类型为TEXT。
- numlines列存储从缓冲区检索到的行数。如果此数字小于所请求的行数，则缓冲区中没有更多行。numlines的数据类型为INTEGER。
- 如果执行GET\_LINE或GET\_LINES，然后执行PUT、PUT\_LINE或NEW\_LINE，而缓冲区中仍存在未检索的消息，则未从缓冲区检索到的消息将被丢弃。

#### 示例

```

DECLARE
    buff    VARCHAR(20) [10];
    stts    INTEGER := 10;
BEGIN
    SELECT lines, numlines INTO buff,stts FROM DBMS_OUTPUT.GET_LINES(stts);

```

#### H. 56. 6. 4. 2. 使用示例

下面是DBMS\_OUTPUT的一个使用示例。

```

CREATE FUNCTION dbms_output_exe() RETURNS VOID AS $$
DECLARE
    buff1 VARCHAR(20);
    buff2 VARCHAR(20);
    stts1 INTEGER;
    stts2 INTEGER;
BEGIN
    PERFORM DBMS_OUTPUT.DISABLE();
    PERFORM DBMS_OUTPUT.ENABLE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    PERFORM DBMS_OUTPUT.PUT('DBMS_OUTPUT TEST 1');
    PERFORM DBMS_OUTPUT.NEW_LINE();
    PERFORM DBMS_OUTPUT.PUT_LINE('DBMS_OUTPUT TEST 2');
    SELECT line,status INTO buff1,stts1 FROM DBMS_OUTPUT.GET_LINE();
    SELECT line,status INTO buff2,stts2 FROM DBMS_OUTPUT.GET_LINE();
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    PERFORM DBMS_OUTPUT.PUT_LINE(buff1);
    PERFORM DBMS_OUTPUT.PUT_LINE(buff2);
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_exe();
DROP FUNCTION dbms_output_exe();

```

## H. 56. 6. 5. DBMS\_PIPE

### 概述

在执行PL / pgSQL的会话之间进行通信。

该软件包可用于1:1通信，例如在执行PL/pgSQL的会话之间交换数据时。

对于管道，有显式管道和隐式管道，并且对于显式管道，您可以选择公共管道和私有管道。每种类型的特征如下：

### 管道类型

Table H. 121. 管道类型

类型	特点
显式管道	<ul style="list-style-type: none"> <li>- 使用 CREATE_PIPE 显式地创建一个管道。</li> <li>- 在创建管道时，可以选择公共管道和私有管道。</li> <li>- 必须使用 REMOVE_PIPE 显式地删除一个管道。</li> </ul>
隐式管道	<ul style="list-style-type: none"> <li>- 在使用 SEND_MESSAGE 和 RECEIVE_MESSAGE 时自动创建。</li> <li>- 创建的管道成为公共管道。</li> <li>- 当使用 RECEIVE_MESSAGE 接收消息时，如果管道中没有其他消息剩余，管道将被自动删除。</li> </ul>
公共管道	<ul style="list-style-type: none"> <li>- 可以作为显式管道或隐式管道创建。</li> <li>- 也可以被除创建者以外的用户使用。</li> </ul>
私有管道	<ul style="list-style-type: none"> <li>- 只能作为显式管道创建。</li> <li>- 只能被其创建者使用。</li> </ul>

### 注意

- 单个实例最多可同时使用50个管道。
- 在频繁创建和删除管道的情况下，请使用公共管道。 如果创建私有管道，即使在管道被删除后，内部信息（私有管道的创建者）仍将保留。 因此，反复创建和删除管道可能会导致内存耗尽。
- 当通过 RECEIVE\_MESSAGE 创建隐式管道时，在未收到消息的情况下发生超时，管道不会被移除。

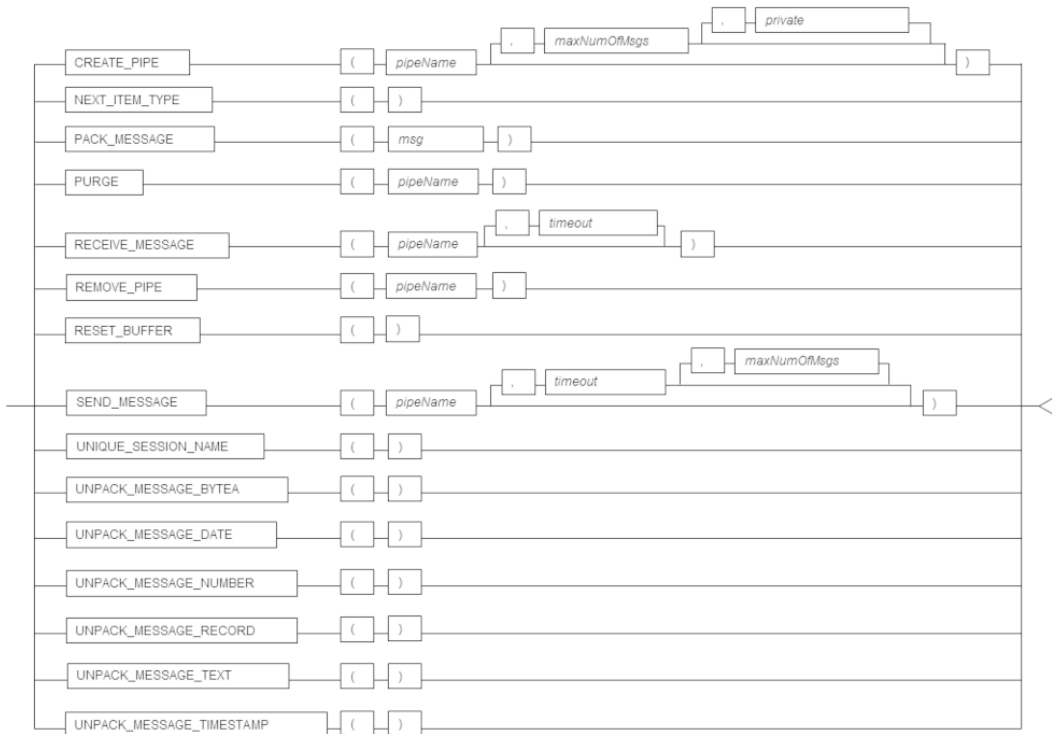
### 特点

Table H. 122. DBMS\_PIPE 特点

特点	描述
CREATE_PIPE	创建公共或私有管道。
NEXT_ITEM_TYPE	确定本地缓冲区中下一个项目的数据类型，并返回该类型。
PACK_MESSAGE	在本地缓冲区中设置一个消息。

特点	描述
PURGE	清空指定管道的内容。
RECEIVE_MESSAGE	在本地缓冲区设置接收到的消息。
REMOVE_PIPE	移除指定的管道。
RESET_BUFFER	重置本地缓冲区的设置位置。
SEND_MESSAGE	发送本地缓冲区的内容。
UNIQUE_SESSION_NAME	返回一个唯一的会话名称。
UNPACK_MESSAGE_BYTEA	在本地缓冲区中以 BYTEA 类型接收一个消息。
UNPACK_MESSAGE_DATE	在本地缓冲区中以 DATE 类型接收一个消息。
UNPACK_MESSAGE_NUMBER	在本地缓冲区中以 NUMERIC 类型接收一个消息。
UNPACK_MESSAGE_RECORD	在本地缓冲区中以 RECORD 类型接收一个消息。
UNPACK_MESSAGE_TEXT	在本地缓冲区中以 TEXT 类型接收一个消息。
UNPACK_MESSAGE_TIMESTAMP	在本地缓冲区中以 TIMESTAMP 类型接收一个消息。

语法



H. 56. 6. 5. 1. 特点描述

本节将解释 DBMS\_PIPE 的每个特点。

CREATE\_PIPE

- CREATE\_PIPE explicitly creates a pipe environment for data communication.

- 指定要创建的管道的名称。
- 管道名称区分大小写。
- 指定可以发送或接收的最大消息数。如果省略，则使用 0（无法发送消息）。指定从 1 到 32767。
- 对于 private，指定 TRUE 或 FALSE。如果指定为 TRUE，则创建私有管道。如果指定为 FALSE，则创建公共管道。默认值为 FALSE。
- 如果已经创建了同名管道，则会发生错误。

参见

有关布尔类型（TRUE/FALSE）的信息，请参阅 LightDB 文档中的 “The SQL Language” > “Data Types” > “Boolean Type”。

示例

```
PERFORM DBMS_PIPE.CREATE_PIPE('P01', 100, FALSE);
```

#### NEXT\_ITEM\_TYPE

- NEXT\_ITEM\_TYPE 返回本地缓冲区中的下一个数据类型。
- 返回值的数据类型为INTEGER。返回以下值之一：

Table H.123. NEXT\_ITEM\_TYPE返回的值

返回值	数据类型
9	NUMERIC类型
11	TEXT类型
12	DATE类型
13	TIMESTAMP类型
23	BYTEA类型
24	RECORD类型
0	缓冲区中没有数据

示例

```
DECLARE
    i_iType    INTEGER;
BEGIN
    i_iType := DBMS_PIPE.NEXT_ITEM_TYPE();
```

#### PACK\_MESSAGE

- PACK\_MESSAGE将指定的消息设置到本地缓冲区中。
- 指定要设置到本地缓冲区中的数据。可以使用以下数据类型：
  - 字符类型 (\*1)
  - 整数类型 (\*2)

- NUMERIC类型
- DATE类型
- TIMESTAMP类型 (\*3)
- BYTEA类型
- RECORD类型

\*1: 字符类型在内部转换为TEXT类型。

\*2: 整数类型在内部转换为NUMERIC类型。

\*3: TIMESTAMP类型在内部转换为带时区的TIMESTAMP类型。

- 每次调用PACK\_MESSAGE时，都会向本地缓冲区添加一个新消息。
- 本地缓冲区的大小约为8KB。但是，每个消息都有开销，因此实际可以存储的总大小小于8KB。要清除本地缓冲区，请发送消息（SEND\_MESSAGE）或将缓冲区（RESET\_BUFFER）重置为其初始状态。

示例

```
PERFORM DBMS_PIPE.PACK_MESSAGE(' Message Test001');
```

PURGE

- PURGE将管道中的消息删除。
- 指定要删除其消息的管道的名称。
- 管道名称区分大小写。

示例

```
PERFORM DBMS_PIPE.PURGE(' P01');
```

注意

执行PURGE操作时，本地缓冲区将用于删除管道中的消息。因此，如果管道中仍有任何消息，本地缓冲区将被PURGE覆盖。

RECEIVE\_MESSAGE

- RECEIVE\_MESSAGE接收指定管道中存在的消息，并将这些消息设置在本地缓冲区中。
- 消息按照发送到管道的单位进行接收，由SEND\_MESSAGE发送。接收到的消息在设置到本地缓冲区之后会从管道中删除。
- 指定要接收消息的管道名称。
- 管道名称区分大小写。
- 指定等待消息的最大等待时间timeout（单位：秒）。如果省略，则默认为31536000秒（1年）。

- 返回值的数据类型为INTEGER。如果成功接收到消息，返回值为0。如果超时，则返回1。

示例

```
DECLARE
    i_Ret    INTEGER;
BEGIN
    i_Ret := DBMS_PIPE.RECEIVE_MESSAGE(' P01', 60);
```

REMOVE\_PIPE

- REMOVE\_PIPE将删除指定的管道。
- 指定要删除的管道的名称。
- 管道名称区分大小写。

示例

```
PERFORM DBMS_PIPE.REMOVE_PIPE(' P01');
```

RESET\_BUFFER

- RESET\_BUFFER重置了本地缓冲区的设置位置。使用此操作可以丢弃本地缓冲区中剩余的任何不必要的数据。

示例

```
PERFORM DBMS_PIPE.RESET_BUFFER();
```

SEND\_MESSAGE

- SEND\_MESSAGE将本地缓冲区中存储的数据发送到指定的管道。
- 指定数据要发送到的管道的名称。
- 管道名称区分大小写。
- 请指定以秒为单位的发送数据的最大等待时间超时时间。如果省略，则默认为31536000秒（1年）。
- 请指定可以发送或接收的最大消息数。如果省略，则使用CREATE\_PIPE中设置的最大消息数。如果在隐式管道中省略，则消息数将不受限制。可以指定1到32767之间的数字。
- 如果在SEND\_MESSAGE和CREATE\_PIPE中都指定了最大消息数，则将使用较大的值。
- 返回值的数据类型为INTEGER。如果成功接收到一条消息，则返回0。如果发生超时，则返回1。

示例

```
DECLARE
    i_Ret    INTEGER;
```

```
BEGIN
    i_Ret := DBMS_PIPE.SEND_MESSAGE(' P01', 10, 20);
```

#### 注意

如果达到最大消息数或正在发送的消息太大，则在发送过程中会发生超时。如果发生超时，请使用RECEIVE\_MESSAGE接收管道中的任何消息。

#### UNIQUE\_SESSION\_NAME

- UNIQUE\_SESSION\_NAME返回一个在所有会话中都唯一的名称。该名称可用作管道名称。
- 同一会话的多个调用始终返回相同的名称。
- 返回值的数据类型为VARCHAR。返回一个长度最多为30个字符的字符串。

#### 示例

```
DECLARE
    p_Name VARCHAR(30);
BEGIN
    p_Name := DBMS_PIPE.UNIQUE_SESSION_NAME();
```

#### UNPACK\_MESSAGE\_BYTEA

- NPACK\_MESSAGE\_BYTEA在本地缓冲区中接收BYTEA类型的消息。
- 通过PACK\_MESSAGE在本地缓冲区中设置单位来接收消息。接收到的消息将从本地缓冲区中移除。
- 返回值的数据类型是BYTEA。
- 如果本地缓冲区中没有消息，则返回NULL值。
- 对于数据类型，必须与PACK\_MESSAGE设置的数据类型对齐。如果数据类型不同，则会出现以下错误。

```
ERROR: datatype mismatch
DETAIL:  unpack unexpected type: xx
```

#### 示例

```
DECLARE
    g_Bytea BYTEA;
BEGIN
    g_Bytea := DBMS_PIPE.UNPACK_MESSAGE_BYTEA();
```

#### UNPACK\_MESSAGE\_DATE

- UNPACK\_MESSAGE\_DATE在本地缓冲区中接收DATE类型的消息。
- 通过PACK\_MESSAGE在本地缓冲区中设置单位来接收消息。接收到的消息将从本地缓冲区中移除。

- 返回值的数据类型是DATE。
- 如果本地缓冲区中没有消息，则返回NULL值。
- 对于数据类型，必须与PACK\_MESSAGE设置的数据类型对齐。如果数据类型不同，则会出现以下错误。

```
ERROR: datatype mismatch
DETAIL: unpack unexpected type: xx
```

#### 示例

```
DECLARE
    g_Date    DATE;
BEGIN
    g_Date := DBMS_PIPE.UNPACK_MESSAGE_DATE();
```

#### 注意

如果在search\_path中设置了“oracle”模式，则将使用orafce的DATE类型，因此接收数据时请使用UNPACK\_MESSAGE\_TIMESTAMP。UNPACK\_MESSAGE\_DATE是LightDB DATE类型的接口。

#### UNPACK\_MESSAGE\_NUMBER

- UNPACK\_MESSAGE\_NUMBER在本地缓冲区中接收NUMERIC类型的消息。
- 通过PACK\_MESSAGE在本地缓冲区中设置单位来接收消息。接收到的消息将从本地缓冲区中移除。
- 返回值的数据类型是NUMERIC。
- 如果本地缓冲区中没有消息，则返回NULL值。
- 对于数据类型，必须与PACK\_MESSAGE设置的数据类型对齐。如果数据类型不同，则会出现以下错误。

```
ERROR: datatype mismatch
DETAIL: unpack unexpected type: xx
```

#### 示例

```
DECLARE
    g_Number    NUMERIC;
BEGIN
    g_Number := DBMS_PIPE.UNPACK_MESSAGE_NUMBER();
```

#### UNPACK\_MESSAGE\_RECORD

- UNPACK\_MESSAGE\_RECORD 接收本地缓冲区中的 RECORD 类型消息。
- 消息由 PACK\_MESSAGE 在本地缓冲区中按照设定的单元接收。接收到的消息将从本地缓冲区中移除。



- 返回值的数据类型是 RECORD。
- 如果本地缓冲区中没有消息，则返回 NULL 值。
- 对于数据类型，需要与 PACK\_MESSAGE 设置的数据类型保持一致。如果数据类型不同，则会出现以下错误：

```
ERROR: datatype mismatch
DETAIL: unpack unexpected type: xx
```

示例

```
DECLARE
    msg1    TEXT;
    status  NUMERIC;
BEGIN
    SELECT col1, col2 INTO msg1, status FROM
DBMS_PIPE.UNPACK_MESSAGE_RECORD();
```

UNPACK\_MESSAGE\_TEXT

- UNPACK\_MESSAGE\_TEXT 接收本地缓冲区中的 TEXT 类型消息。
- 消息由 PACK\_MESSAGE 在本地缓冲区中按照设定的单元接收。接收到的消息将从本地缓冲区中移除。
- 返回值的数据类型是 TEXT。
- 如果本地缓冲区中没有消息，则返回 NULL 值。
- 对于数据类型，需要与 PACK\_MESSAGE 设置的数据类型保持一致。如果数据类型不同，则会出现以下错误。

```
ERROR: datatype mismatch
DETAIL: unpack unexpected type: xx
```

示例

```
DECLARE
    g_Text  TEXT;
BEGIN
    g_Text := DBMS_PIPE.UNPACK_MESSAGE_TEXT();
```

UNPACK\_MESSAGE\_TIMESTAMP

- UNPACK\_MESSAGE\_TIMESTAMP 接收本地缓冲区中的 TIMESTAMP WITH TIME ZONE 类型消息。
- 消息由 PACK\_MESSAGE 在本地缓冲区中按照设定的单元接收。接收到的消息将从本地缓冲区中移除。
- 返回值的数据类型是 TIMESTAMP WITH TIME ZONE。

- 如果本地缓冲区中没有消息，则返回 NULL 值。
- 对于数据类型，需要与 PACK\_MESSAGE 设置的数据类型保持一致。如果数据类型不同，则会出现以下错误。

```
ERROR: datatype mismatch
DETAIL: unpack unexpected type: xx
```

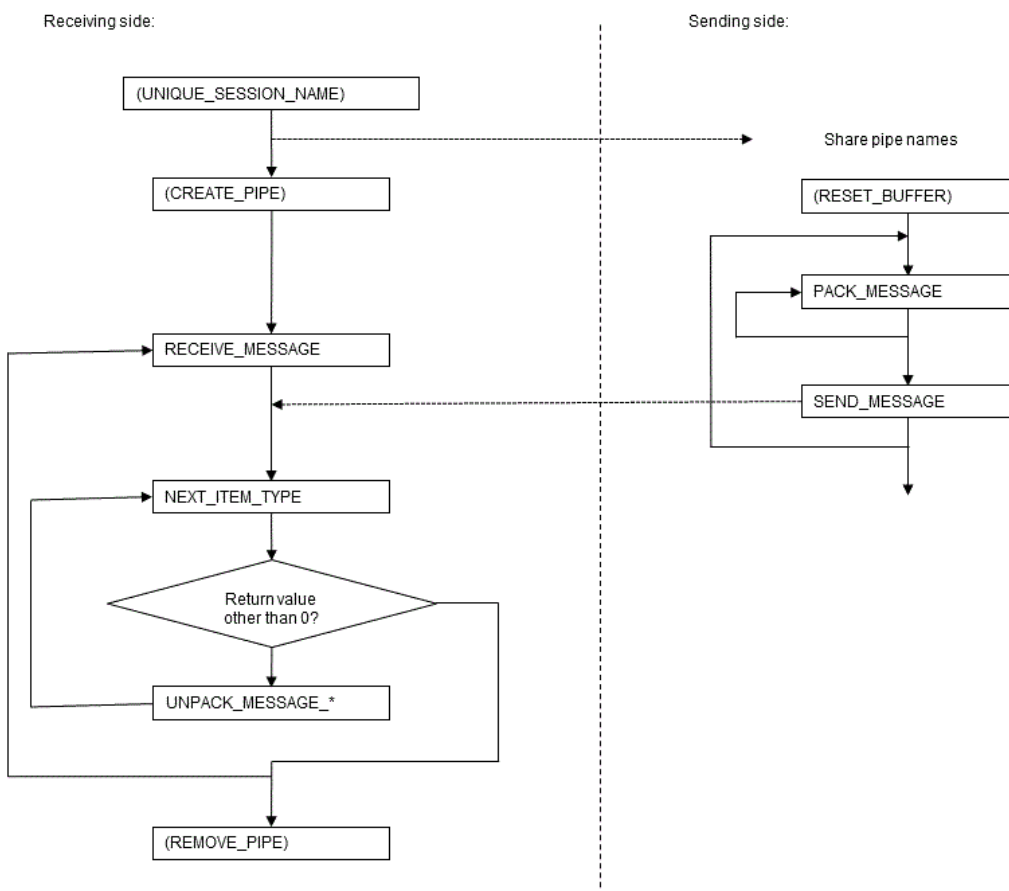
示例

```
DECLARE
    g_Timestampz    TIMESTAMP WITH TIME ZONE;
BEGIN
    g_Timestampz := DBMS_PIPE.UNPACK_MESSAGE_TIMESTAMP();
```

### H. 56. 6. 5. 2. 使用示例

下面是使用 DBMS\_PIPE 处理流程的示例：

DBMS\_PIPE 的流程



注意

- 当使用 CREATE\_PIPE 明确创建管道时，确保使用 REMOVE\_PIPE 命令删除管道。如果未明确删除管道，则一旦创建，它将一直存在，直到实例停止。

- 在流程图中，创建管道和删除管道是在接收方描述的，但是，这些也可以在发送方执行。为了保持一致性，建议在一个方向上创建和删除管道。
- 如果同名管道已经存在，CREATE\_PIPE 将会出现错误。 隐式创建的管道也可以成为 SEND\_MESSAGE 和 RECEIVE\_MESSAGE 的目标，因此在执行 CREATE\_PIPE 时，确保不要先调用 SEND\_MESSAGE 和 RECEIVE\_MESSAGE。
- DBMS\_ALERT 和 DBMS\_PIPE 使用相同的内存环境。因此，当检测到 DBMS\_ALERT 的内存不足时，也可能会检测到 DBMS\_PIPE 的内存不足。

#### 信息

正在使用中的管道信息可以在DBMS\_PIPE.DB\_PIPEES视图中查看。

```
SELECT * from dbms_pipe.db_pipes;
name | items | size | limit | private | owner
-----+-----+-----+-----+-----+-----
P01  |      1 |    18 |    100 | f       |
(1 row)
```

#### 使用示例

- 发送方

```
CREATE FUNCTION send_dbms_pipe_exe(IN pipe_mess text) RETURNS void AS $$
DECLARE
    pipe_name text := 'sample_pipe';
    pipe_time timestamp := current_timestamp;
    pipe_stat int;
BEGIN
    PERFORM DBMS_PIPE.RESET_BUFFER();
    PERFORM DBMS_PIPE.PACK_MESSAGE(pipe_mess);
    PERFORM DBMS_PIPE.PACK_MESSAGE(pipe_time);
    pipe_stat := DBMS_PIPE.SEND_MESSAGE(pipe_name);
    RAISE NOTICE 'PIPE_NAME: % SEND Return Value =%', pipe_name, pipe_stat;
END;
$$ LANGUAGE plpgsql;

SELECT send_dbms_pipe_exe(' Sample Message. ');
DROP FUNCTION send_dbms_pipe_exe(text);
```

- 接收方

```
CREATE FUNCTION receive_dbms_pipe_exe() RETURNS void AS $$
DECLARE
    pipe_name text := 'sample_pipe';
    pipe_text text;
    pipe_nume numeric;
    pipe_date date;
    pipe_time timestamp with time zone;
    pipe_byte bytea;
    pipe_reco record;
    pipe_item int;
    pipe_stat int;
```

```

BEGIN
pipe_stat := DBMS_PIPE.RECEIVE_MESSAGE(pipe_name, 300);
RAISE NOTICE 'Return Value = %', pipe_stat;
    LOOP
        pipe_item := DBMS_PIPE.NEXT_ITEM_TYPE();
        RAISE NOTICE 'Next Item : %', pipe_item;
        IF (pipe_item = 9) THEN
            pipe_nume := DBMS_PIPE.UNPACK_MESSAGE_NUMBER();
            RAISE NOTICE 'Get Message : %' ,pipe_nume;
        ELSIF (pipe_item = 11) THEN
            pipe_text := DBMS_PIPE.UNPACK_MESSAGE_TEXT();
            RAISE NOTICE 'Get Message : %' ,pipe_text;
        ELSIF (pipe_item = 12) THEN
            pipe_date := DBMS_PIPE.UNPACK_MESSAGE_DATE();
            RAISE NOTICE 'Get Message : %' ,pipe_date;
        ELSIF (pipe_item = 13) THEN
            pipe_time := DBMS_PIPE.UNPACK_MESSAGE_TIMESTAMP();
            RAISE NOTICE 'Get Message : %' ,pipe_time;
        ELSIF (pipe_item = 23) THEN
            pipe_byte := DBMS_PIPE.UNPACK_MESSAGE_BYTEA();
            RAISE NOTICE 'Get Message : %' ,pipe_byte;
        ELSIF (pipe_item = 24) THEN
            pipe_reco := DBMS_PIPE.UNPACK_MESSAGE_RECORD();
            RAISE NOTICE 'Get Message : %' ,pipe_reco;
        ELSE
            EXIT;
        END IF;
    END LOOP;
    PERFORM DBMS_PIPE.REMOVE_PIPE(pipe_name);
END;
$$ LANGUAGE plpgsql;

SELECT receive_dbms_pipe_exe();
DROP FUNCTION receive_dbms_pipe_exe();

```

### H. 56. 6. 6. DBMS\_RANDOM

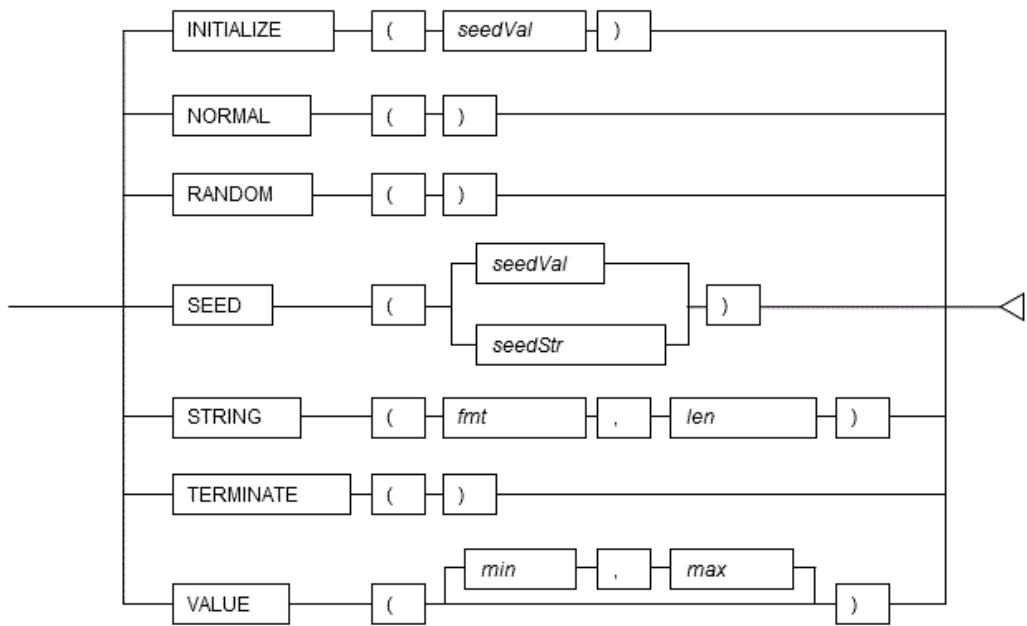
概述

生成在PL/pgSQL中的随机数。

Table H. 124. DBMS\_RANDOM功能

功能	描述
INITIALIZE	初始化生成随机数。
NORMAL	返回正态分布的随机数。
RANDOM	生成随机数。
SEED	重置种子值。
STRING	生成一个随机字符串。
TERMINATE	终止随机数生成。
VALUE	生成一个介于0和1之间的随机十进制数，或介于指定值之间的随机十进制数。

语法



### H. 56. 6. 6. 1. 功能说明

本节介绍DBMS\_RANDOM的每个功能。

#### INITIALIZE

- INITIALIZE使用指定的种子值初始化随机数生成。
- 对于seedVal，请指定SMALLINT或INTEGER类型。

示例

```
PERFORM DBMS_RANDOM. INITIALIZE (999) ;
```

#### NORMAL

- NORMAL生成并返回一个正态分布的随机数。
- 返回值类型为DOUBLE PRECISION。

示例

```
DECLARE
    d_RunNum    DOUBLE PRECISION;
BEGIN
    d_RunNum := DBMS_RANDOM. NORMAL ();
```

#### RANDOM

- RANDOM生成并返回一个随机数。
- 返回值的数据类型为INTEGER。

示例

```

DECLARE
    d_RunInt    INTEGER;
BEGIN
    d_RunInt := DBMS_RANDOM.RANDOM();

```

SEED

- SEED使用指定的种子值或种子字符串初始化随机数生成。
- 对于seedVal，请指定SMALLINT或INTEGER类型。
- 种子字符串可以指定为任何字符串。

示例

```

PERFORM DBMS_RANDOM.SEED('123');

```

STRING

- STRING根据指定的显示格式和字符串长度生成并返回一个随机字符串。
- 对于显示格式fmt，请指定以下任何值。如果指定了其他任何值，则会出现错误。

Table H.125. 可指定的显示格式值

设置值	生成的字符串
'u', 'U'	仅大写字母
'l', 'L'	仅小写字母
'a', 'A'	大写字母和小写字母的混合
'x', 'X'	大写字母和数字
'p', 'P'	任何可显示字符

- 指定要生成的字符串的长度。请指定SMALLINT或INTEGER类型。
- 返回值的数据类型为TEXT。

示例

```

DECLARE
    d_RunStr    TEXT;
BEGIN
    d_RunStr := DBMS_RANDOM.STRING('a', 20);

```

TERMINATE

- 调用TERMINATE以终止随机数生成。

信息

TERMINATE不执行任何操作，但已包含以与Oracle数据库兼容。

示例

```

PERFORM DBMS_RANDOM.TERMINATE();

```

VALUE

- VALUE生成并返回指定范围内的随机数。
- 对于min和max， 请指定数字数据类型。将生成介于最小值和最大值之间（包括最小值和最大值）的随机数。
- 如果省略了最小值和最大值，则将生成介于0和1之间的随机十进制数。
- 返回值的数据类型为DOUBLE PRECISION。

示例

```
DECLARE
    d_RunDb1    DOUBLE PRECISION;
BEGIN
    d_RunDb1 := DBMS_RANDOM.VALUE();
```

### H. 56. 6. 6. 2. 使用示例

下面是DBMS\_RANDOM的使用示例。

```
CREATE FUNCTION dbms_random_exe() RETURNS VOID AS $$
DECLARE
    w_rkey VARCHAR(10) := 'rnd111';
    i_rkey INTEGER := 97310;
BEGIN
    PERFORM DBMS_RANDOM.INITIALIZE(i_rkey);
    RAISE NOTICE 'RANDOM -> NORMAL : %', DBMS_RANDOM.NORMAL();
    RAISE NOTICE 'RANDOM -> RANDOM : %', DBMS_RANDOM.RANDOM();
    RAISE NOTICE 'RANDOM -> STRING : %', DBMS_RANDOM.STRING('a', 10);
    RAISE NOTICE 'RANDOM -> VALUE : %', DBMS_RANDOM.VALUE();
    PERFORM DBMS_RANDOM.SEED(w_rkey);
    RAISE NOTICE 'RANDOM -> NORMAL : %', DBMS_RANDOM.NORMAL();
    RAISE NOTICE 'RANDOM -> RANDOM : %', DBMS_RANDOM.RANDOM();
    RAISE NOTICE 'RANDOM -> STRING : %', DBMS_RANDOM.STRING('p', 10);
    RAISE NOTICE 'RANDOM -> VALUE : %', DBMS_RANDOM.VALUE(1, 100);
    PERFORM DBMS_RANDOM.TERMINATE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_random_exe();
DROP FUNCTION dbms_random_exe();
```

### H. 56. 6. 7. DBMS\_UTILITY

概述

提供PL/pgSQL的实用工具。

Table H. 126. DBMS\_UTILITY功能

功能	描述
FORMAT_CALL_STACK	返回当前调用堆栈。
GET_HASH_VALUE	基于输入字符串的哈希值。
CANONICALIZE	将给定字符串进行规范化。

功能	描述
COMMA_TO_TABLE	将以逗号分隔的名称列表转换为PL/SQL名称表。
TABLE_TO_COMMA	将PL/SQL名称表转换为以逗号分隔的名称列表。
DB_VERSION	返回数据库的版本信息。
EXEC_DDL_STATEMENT	在parse_string中执行DDL语句。
GET_TIME	以百分之一秒为单位查找当前时间。

语法

```

FORMAT_CALL_STACK(fmt text);
FORMAT_CALL_STACK();
GET_HASH_VALUE(name VARCHAR2, base INTEGER, hash_size INTEGER)
CANONICALIZE(name TEXT, canon_name TEXT, canon_len INT4)
COMMA_TO_TABLE(list TEXT, tablen INT4, tab TEXT[])
TABLE_TO_COMMA(tab TEXT[], tablen INT4, list TEXT)
DB_VERSION(version TEXT, compatibility TEXT)
EXEC_DDL_STATEMENT(parse_string TEXT)
    
```

#### H. 56. 6. 7. 1. 功能描述

本节介绍了DBMS\_UTILITY的每个功能。

FORMAT\_CALL\_STACK

- FORMAT\_CALL\_STACK返回PL/pgSQL的当前调用堆栈。
- 对于显示格式fmt，请指定以下任何值。如果指定了其他任何值，则会出现错误。

Table H. 127. 可指定的显示格式值

设置值	显示的内容
'o'	标准格式的调用堆栈显示（带标题）
's'	标准格式的调用堆栈显示（不带标题）
'p'	逗号分隔的调用堆栈显示（不带标题）

- 如果省略了显示格式，则使用显示格式“o”。
- 返回值的数据类型为TEXT。

示例

```

DECLARE
    s_StackTrace TEXT
BEGIN
    s_StackTrace := DBMS_UTILITY.FORMAT_CALL_STACK();
    
```

注意

如果指定了英语以外的语言环境作为消息语言环境，则可能无法正确检索调用堆栈结果。为了正确检索调用堆栈结果，请将英语指定为消息语言环境。

GET\_HASH\_VALUE



- 基于输入字符串的哈希值，哈希值应在base和base + hash\_size -1之间。

示例

```
PERFORM DBMS_UTILITY.GET_HASH_VALUE(' abcdef', 0, 10000);
```

#### CANONICALIZE

- 此过程处理单个保留字或关键字（例如“table”），并为单个标识符删除空格，以便'table'变成TABLE。

示例

```
declare
    l_result      text := '';
begin
    dbms_output.serveroutput(True);
    dbms_utility.canonicalize(' taBlE', l_result, 100);
    dbms_output.put_line(l_result);
end;
/
```

#### COMMA\_TO\_TABLE

- 这些过程将以逗号分隔的名称列表转换为PL/SQL名称表。
- 列表必须是非空的逗号分隔列表：除逗号分隔列表之外的任何内容都会被拒绝。双引号内的逗号不算。

示例

```
declare
    tablen      int4;
    tab         text[];
begin
    dbms_output.serveroutput(True);
    dbms_utility.comma_to_table(' table, "Table", "Table, 122324" , Tablesd',
    tablen, tab);
    dbms_output.put_line(tablen);
    dbms_output.put_line(array_to_string(tab, ','));
end;
/
```

#### TABLE\_TO\_COMMA

- 此过程将PL/SQL名称表转换为逗号分隔的名称列表。

示例

```
declare
    tablen      int4;
    l_str       text;
begin
    dbms_output.serveroutput(True);
```

```

    dbms_utility.table_to_comma(' {table," \"table\"","\"table.1234567890\" ",
tablezx",NULL}'::text[], tablen, l_str);
    dbms_output.put_line(tablen);
    dbms_output.put_line(l_str);

```

```

end;
/

```

#### DB\_VERSION

- 此过程返回数据库的版本信息。

示例

```

declare
    lt_version      text;
    compatibility   text;
begin
    dbms_output.serveroutput(True);
    dbms_utility.db_version(lt_version, compatibility);
    dbms_output.put_line(lt_version);
    dbms_output.put_line(oracle.nvl(compatibility,'NULL'));
end;
/

```

#### EXEC\_DDL\_STATEMENT

- 此过程在parse\_string中执行DDL语句。

示例

```

declare
    l_result        text := '';
begin
    dbms_output.serveroutput(True);
    dbms_utility.exec_ddl_statement('create table if not exists
public.test_exec_ddl_stmt (id int, name text);');
end;
/

```

#### GET\_TIME

- 此函数返回当前时间的百分之一秒。两次调用返回时间之间的差异可测量两个时间点之间经过的总时间（不仅仅是CPU处理时间）

示例

```

DO $$
DECLARE
    start_time integer;
    end_time integer;
BEGIN
    start_time := DBMS_UTILITY.GET_TIME();
    PERFORM pg_sleep(2);
    end_time := DBMS_UTILITY.GET_TIME();

```

```

        -- clamp long runtime on slow build machines to the 2s the testsuite is
        expecting
        IF end_time BETWEEN start_time + 300 AND start_time + 1000 THEN
end_time := start_time + 250; END IF;
        RAISE NOTICE 'Execution time: % seconds', trunc((end_time -
start_time)::numeric/100);
        END
    $$;

```

#### H. 56. 6. 7. 2. 使用示例

下面是DBMS\_UTILITY的一个使用示例。

```

CREATE FUNCTION dbms_utility1_exe() RETURNS VOID AS $$
DECLARE
    s_StackTrace TEXT;
    v_hashvalue numeric(38,0);
BEGIN
    s_StackTrace := DBMS_UTILITY.FORMAT_CALL_STACK();
    RAISE NOTICE '%', s_StackTrace;
    v_hashvalue := DBMS_UTILITY.GET_HASH_VALUE('abcdef', 0, 10000);
    RAISE NOTICE 'v_hashvalue=%', v_hashvalue;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE FUNCTION dbms_utility2_exe() RETURNS VOID AS $$
BEGIN
    PERFORM dbms_utility1_exe();
END;
$$ LANGUAGE plpgsql;

```

```

SELECT dbms_utility2_exe();
DROP FUNCTION dbms_utility2_exe();
DROP FUNCTION dbms_utility1_exe();

```

```

declare
    l_result          text := '';
begin
    dbms_output.serveroutput(True);
    dbms_utility.exec_ddl_statement('create table if not exists
public.test_exec_ddl_stmt (id int, name text);');
    l_result := dbms_metadata.get_ddl('table', 'test_exec_ddl_stmt');
    dbms_output.put_line(l_result);
    dbms_utility.exec_ddl_statement(' truncate table
public.test_exec_ddl_stmt');
    dbms_output.put_line(' truncate SUCCESS');
    dbms_utility.exec_ddl_statement(' drop table if exists
public.test_exec_ddl_stmt ; ');
    dbms_output.put_line(' drop SUCCESS');
end;
/

```

#### H. 56. 6. 8. DBMS\_JOB

概述

在lt\_cron中安排和管理作业。

Table H. 128. DBMS\_JOB功能

功能	描述
BROKEN	禁用作业执行。（无效）
CHANGE	更改与作业关联的任何用户定义参数。
INSTANCE	将作业分配给实例运行。（无效）
INTERVAL	更改指定作业的执行间隔。
NEXT_DATE	更改指定作业的下一个执行时间。（无效）
REMOVE	从作业队列中删除指定的作业。
RUN	强制指定的作业运行。（无效）
SUBMIT	向作业队列提交新作业。
USER_EXPORT	为导出重新创建一个给定的作业。
WHAT	更改指定作业的作业描述。

语法

```

CHANGE(job INT8, what TEXT, next_date TIMESTAMP, interval TEXT, instance INT4
default null, force BOOL default 'false')
INTERVAL(job INT8, interval TEXT)
REMOVE(job INT8)
SUBMIT(job INT8, what TEXT, next_date TIMESTAMP default sysdate, interval TEXT
default 'null', no_parse BOOL default 'false', instance INT4 default null, force
BOOL default 'false')
USER_EXPORT(job INT8, mycall TEXT)
WHAT(job INT8, what TEXT)
    
```

### H. 56. 6. 8. 1. 功能描述

本节解释了dbms\_job的每个功能。

CHANGE

- 此过程更改用户可以在作业中设置的任何字段。

示例

```
dbms_job.change(l_job_id, 'select 3', sysdate, '0 11 * * *');
```

INTERVAL

- 此过程更改作业运行的频率。
- interval参数格式是lt\_cron的格式。

示例

```
dbms_job.interval(l_job_id, '0 11 * * *');
```

REMOVE

- 此过程从作业队列中删除现有作业。

示例

```
dbms_job.remove(2);
```

SUBMIT

- 此过程提交新作业。
- 此过程提交的作业将没有名称。 建议使用“select cron.Schedule (name, xxx)”提交。

示例

```
dbms_job.submit(l_job_id, 'select 1', sysdate, '0 10 * * *');
```

USER\_EXPORT

- 生成重新创建给定作业的调用文本。

示例

```
dbms_job.user_export(2, l_result);
```

WHAT

- 此过程更改现有作业的操作。

示例

```
dbms_job.what(2, 'select 2');
```

## H. 56. 6. 8. 2. 使用示例

下面是DBMS\_JOB的一个使用示例。

```
declare
    l_job_id    int8;
    l_result    text := '';
begin
    dbms_output.serveroutput(true);

    -- use dbms_job.submit will not have jobname
    dbms_job.submit(l_job_id, 'select 1', sysdate, '0 10 * * *');
    dbms_output.put_line('submit job, job id:' || l_job_id);

    dbms_job.user_export(l_job_id, l_result);
    dbms_output.put_line('user_export(job id' || l_job_id || '):' || l_result);

    -- update can't be used for job without name
```

```

dbms_job.what(l_job_id, 'select 2');
dbms_job.user_export(l_job_id, l_result);
dbms_output.put_line('after dbms_job.what user_export(job id: ' ||
l_job_id || '):' || l_result);

dbms_job.interval(l_job_id, '0 11 * * *');
dbms_job.user_export(l_job_id, l_result);
dbms_output.put_line('after dbms_job.interval user_export(job id: ' ||
l_job_id || '):' || l_result);

dbms_job.change(l_job_id, 'select 3', sysdate, '0 11 * * *');
dbms_job.user_export(l_job_id, l_result);
dbms_output.put_line('after dbms_job.change user_export(job id: ' ||
l_job_id || '):' || l_result);

dbms_job.next_date(l_job_id, sysdate);
dbms_job.broken(l_job_id, True);
dbms_job.instance(l_job_id, 10);
dbms_job.run(l_job_id);

dbms_job.remove(l_job_id);

end;
/

```

## H. 56. 6. 9. DBMS\_LOCK

### 概述

提供锁管理服务的接口。

Table H. 129. DBMS\_LOCK功能

功能	描述
sleep	将一个会话置于休眠状态，等待特定时间。

### 语法

SLEEP(pi\_seconds FLOAT8)

### H. 56. 6. 9. 1. 功能描述

本节解释了dbms\_job的每个功能。

#### SLEEP

- 此过程将暂停会话一段指定的时间。
- 暂停会话的时间以秒为单位。

#### 示例

```
dbms_lock.sleep(0.1);
```

## H. 56. 6. 10. DBMS\_METADATA

概述

为您提供了一种从数据库字典中检索元数据的方式。

Table H. 130. DBMS\_METADATA 功能

功能	描述
get_ddl	允许您获取对象的元数据。

语法

```
GET_DDL(object_type VARCHAR, name VARCHAR, schema VARCHAR DEFAULT
current_schema(), VARCHAR DEFAULT 'COMPATIBLE', VARCHAR DEFAULT 'ORACLE', VARCHAR
DEFAULT 'DDL', BOOL DEFAULT 'false') RETURNS TEXT
```

### H. 56. 6. 10. 1. 功能描述

本节介绍了 dbms\_job 的每个功能。

GET\_DDL

- 允许您获取对象的元数据。

示例

```
select dbms_metadata.get_ddl('constraint', 'key1_check') from dual;
```

## H. 56. 6. 11. DBMS\_OBFUSCATION\_TOOLKIT

概述

启用应用程序使用数据加密标准 (DES) 或三重DES算法加密数据。

Table H. 131. DBMS\_OBFUSCATION\_TOOLKIT 功能特性

功能	描述
md5	生成数据的 MD5 哈希值。

语法

```
MD5(input_string TEXT) RETURNS RAW
```

### H. 56. 6. 11. 1. 功能描述

本节介绍 dbms\_job 的每个功能。

MD5

- 生成数据的 MD5 哈希值。

示例

```
select dbms_obfuscation_toolkit.md5(input_string => '测试')::text from dual;
```

## H. 56. 6. 12. DBMS\_SNAPSHOT

概述

启用您刷新快照 (MVIEW)。

Table H. 132. DBMS\_SNAPSHOT 功能特性

功能	描述
refresh	刷新快照列表。

语法

```
REFRESH(list TEXT,method TEXT default 'C')
```

### H. 56. 6. 12. 1. 功能描述

本节介绍 dbms\_job 的每个功能。

REFRESH

- 此过程刷新快照 (MVIEW) 列表。

示例

```
select dbms_snapshot.refresh('mv_test_dbms_snapshot,mv_test_dbms_snapshot1')
from dual;
```

## H. 56. 6. 13. UTL\_FILE

概述

PL/pgSQL 可以写入和读取文本文件。

要执行这些文件操作，操作目标的目录必须预先在 UTL\_FILE.UTL\_FILE\_DIR 表中注册。作为数据库管理员或具有 INSERT 权限的用户，使用 INSERT 语句来注册目录。此外，如果目录不再需要，请从相同的表中删除它。有关如何注册和删除目录的信息，请参阅“注册和删除目录”。

在 PL/pgSQL 中，如下声明之后将解释的文件处理程序：

```
DECLARE f UTL_FILE.FILE_TYPE;
```

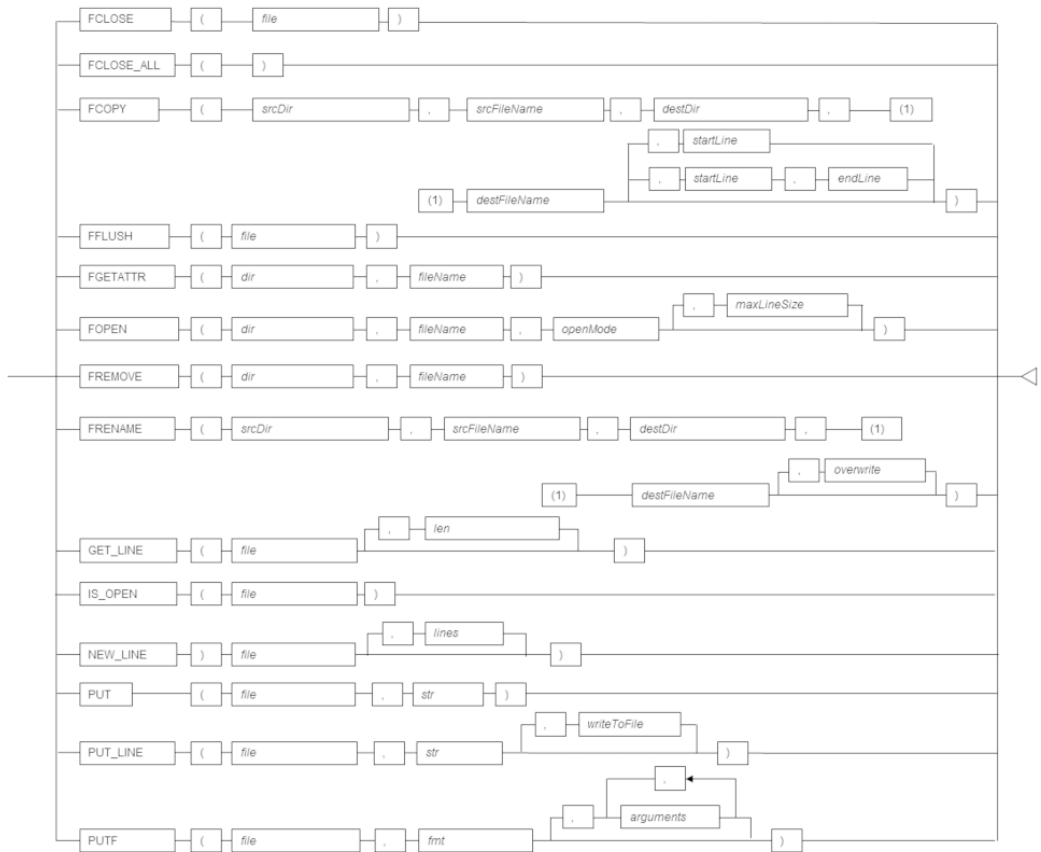
Table H. 133. UTL\_FILE 功能特性

功能	描述
FCLOSE	关闭文件。
FCLOSE_ALL	关闭会话中打开的所有文件。



功能	描述
FCOPY	复制整个文件或其连续部分。
FFLUSH	刷新缓冲区。
FGETATTR	检索文件的属性。
FOPEN	打开文件。
FREMOVE	删除文件。
FRENAME	重命名文件。
GET_LINE	从文本文件中读取一行。
IS_OPEN	检查文件是否已打开。
NEW_LINE	写入换行符。
PUT	写入字符串。
PUT_LINE	将换行符附加到字符串并将该字符串写入。
PUTF	写入格式化字符串。

语法



H. 56. 6. 13. 1. 注册和删除目录

注册目录

1. 检查目录是否已经注册（如果已经注册，则不需要执行第2步）。

```
SELECT * FROM UTL_FILE.UTL_FILE_DIR WHERE dir='/home/pgsql';
```

2. 注册目录。

```
INSERT INTO UTL_FILE.UTL_FILE_DIR VALUES ('/home/pgsql');
```

删除目录

```
DELETE FROM UTL_FILE.UTL_FILE_DIR WHERE dir='/home/lightdb';
```

### H. 56. 6. 13. 2. 描述

本节介绍 UTL\_FILE 的每个功能。

FCLOSE

- FCLOSE 用于关闭打开的文件。
- 指定一个打开的文件句柄。
- 返回值为 NULL 值。

示例

```
f := UTL_FILE.FCLOSE(f);
```

FCLOSE\_ALL

- FCLOSE\_ALL 关闭会话中所有打开的文件。
- 使用 FCLOSE\_ALL 关闭的文件将无法再进行读写操作。

示例

```
PERFORM UTL_FILE.FCLOSE_ALL();
```

FCOPY

- FCOPY 可以复制整个文件或其连续部分。如果没有指定 startLine 和 endLine，则会复制整个文件。
- 指定源文件的目录位置。
- 指定源文件。
- 指定创建目标文件的目录。
- 指定目标文件的名称。
- 指定开始复制的行号。请指定大于0的值。如果没有指定，则使用1。
- 指定复制结束的行号。如果没有指定，则使用文件的最后一行。

示例

```
PERFORM UTL_FILE.FCOPY('/home/lightdb', 'regress_ltsql.txt', '/home/
lightdb', 'regress_ltsql2.txt');
```

#### FFLUSH

- FFLUSH 强制将缓冲数据写入文件。
- 指定一个打开的文件句柄。

#### 示例

```
PERFORM UTL_FILE.FFLUSH(f);
```

#### FGETATTR

- FGETATTR 用于检索文件属性：文件是否存在，文件大小以及有关文件块大小的信息。
- 指定文件所在的目录。
- 指定相关文件名。
- 使用SELECT语句获取文件属性，这些属性存储在“fexists”，“file\_length”和“blocksize”列中。
- “fexists”列存储一个布尔值（TRUE/FALSE）。如果文件存在，则将“fexists”设置为TRUE。如果文件不存在，则将“fexists”设置为FALSE。fexists的数据类型为BOOLEAN。
- “file\_length”列以字节为单位存储文件的长度。如果文件不存在，则“file\_length”为NULL。file\_length的数据类型为INTEGER。
- “blocksize”列以字节为单位存储文件的块大小。如果文件不存在，则“blocksize”为NULL。blocksize的数据类型为INTEGER。

#### 示例

```
SELECT fexists, file_length, blocksize INTO file_flag, file_len, size
FROM UTL_FILE.FGETATTR('/home/lightdb', 'regress_ltsql.txt');
```

#### FOPEN

- FOPEN打开一个文件。
- 指定文件所在的目录。
- 指定文件名。
- 指定打开文件的模式：

r: 读取模式

w: 写入

a: 添加

- 指定可以在一次操作中处理的最大字符串长度（以字节为单位）。 如果省略，默认值为1024。指定1到32767之间的值。
- 每次会话最多可以同时打开50个文件。

示例

```
f := UTL_FILE.FOPEN('/home/pgsql', 'regress_pgsql.txt', 'r', 1024);
```

FREMOVE

- FREMOVE用于删除一个文件。
- 指定文件所在的目录。
- 指定文件名。

示例

```
PERFORM UTL_FILE.FREMOVE('/home/lightdb', 'regress_ltsql.txt');
```

FRENAME

- FRENAME用于重命名文件。
- 指定源文件所在的目录位置。
- 指定要重命名的源文件。
- 指定要创建重命名文件的目录。
- 指定文件的新名称。
- 指定是否覆盖与重命名文件同名并位于同一位置的文件。如果指定为TRUE，则会覆盖现有文件。如果指定为FALSE，则会发生错误。如果省略，则设置为FALSE。

参见

请参考LightDB文档中的“SQL语言” > “数据类型” > “布尔类型”以获取关于布尔类型（TRUE/FALSE）值的信息。

示例

```
PERFORM UTL_FILE.FRENAME('/home/lightdb', 'regress_ltsql.txt', '/home/lightdb', 'regress_ltsql2.txt', TRUE);
```

GET\_LINE

- GET\_LINE用于从文件中读取一行。
- 指定以r（读取）模式返回的FOPEN文件句柄。
- 指定从文件中读取的字节数。如果未指定，则将使用FOPEN中指定的最大字符串长度。
- 返回值是接收从文件中读取的行的缓冲区。
- 换行符不会被加载到缓冲区中。
- 如果加载的是空行，则返回空字符串。
- 指定要读取的数据的最大长度（以字节为单位）。指定1到32767之间的值。如果未指定，则设置为FOPEN中指定的最大字符串长度。如果在FOPEN中未指定最大字符串长度，则设置为1024。

- 如果行长度大于要读取的指定字节数，则剩余部分将在下一次调用中读取。
- 当尝试读取最后一行之后的内容时，将会发生NO\_DATA\_FOUND异常。

示例

```
buff := UTL_FILE.GET_LINE(f);
```

IS\_OPEN

- IS\_OPEN用于检查文件是否已经打开。
- 指定文件句柄。
- 返回值是BOOLEAN类型。TRUE表示打开状态，FALSE表示关闭状态。

参见

请参考LightDB文档中的“SQL语言” > “数据类型” > “布尔类型”以获取关于布尔类型（TRUE/FALSE）值的信息。

示例

```
IF UTL_FILE.IS_OPEN(f) THEN
    PERFORM UTL_FILE.FCLOSE(f);
END IF;
```

NEW\_LINE

- NEW\_LINE用于写入一个或多个换行符。
- 指定一个打开的文件句柄。
- 指定要写入文件的换行符数量。如果省略，则使用“1”。

示例

```
PERFORM UTL_FILE.NEW_LINE(f, 2);
```

PUT

- PUT用于将字符串写入文件。
- 指定使用w（写入）或a（追加）打开的FOPEN文件句柄。
- 指定要写入文件的字符串。
- 要写入的字符串的最大长度（以字节为单位）是在FOPEN中指定的最大字符串长度。
- PUT不会追加换行符。如果要追加换行符，请使用NEW\_LINE命令。

示例

```
PERFORM UTL_FILE.PUT(f, 'ABC');
```

## PUT\_LINE

- PUT\_LINE将一个换行符追加到字符串末尾，并将字符串写入文件。
- 指定使用w（写入）或a（追加）打开的FOPEN文件句柄。
- 指定是否强制写入文件。如果指定为TRUE，则会强制写入文件。如果指定为FALSE，则文件写入是异步的。如果省略，则设置为FALSE。
- 字符串的最大长度（以字节为单位）是在FOPEN中指定的最大字符串长度。

### 示例

```
PERFORM UTL_FILE.PUT_LINE(f, 'ABC', TRUE);
```

## PUTF

- PUTF用于写入格式化的字符串。
- 指定使用w（写入）或a（追加）打开的FOPEN文件句柄。
- 指定格式，格式字符串中包含格式化字符 \n 和 %s。
- 格式字符串中的 \n 表示换行符。
- 输入值的数量必须与格式字符串中 %s 的数量相同。最多可以指定五个输入值。格式字符串中的 %s 将被相应的输入字符替换。如果没有指定与 %s 相应的输入值，则它将被替换为空字符串。

### 示例

```
PERFORM UTL_FILE.PUTF(f, '[1=%s, 2=%s, 3=%s, 4=%s, 5=%s]\n', '1', '2', '3', '4', '5');
```

## H. 56. 6. 13. 3. 使用示例

在使用 UTL\_FILE 时的流程和使用示例如下所示。

### 1. 准备

在开始使用 UTL\_FILE 的新作业之前，请在 UTL\_FILE.UTL\_FILE\_DIR 表中注册目录。

有关如何注册目录的信息，请参见“注册和删除目录”。

### 2. 执行作业

执行使用 UTL\_FILE 的作业。以下是示例。

```
CREATE OR REPLACE FUNCTION gen_file(mydir TEXT, infile TEXT, outfile TEXT,
copyfile TEXT) RETURNS void AS $$
DECLARE
    v1 VARCHAR(32767);
    inf UTL_FILE.FILE_TYPE;
    otf UTL_FILE.FILE_TYPE;
BEGIN
    inf := UTL_FILE.FOPEN(mydir, infile,'r',256);
    otf := UTL_FILE.FOPEN(mydir, outfile,'w');
```

```

v1 := UTL_FILE.GET_LINE(inf, 256);
PERFORM UTL_FILE.PUT_LINE(otf, v1, TRUE);
v1 := UTL_FILE.GET_LINE(inf, 256);
PERFORM UTL_FILE.PUTF(otf, '%s\n', v1);
v1 := UTL_FILE.GET_LINE(inf, 256);
PERFORM UTL_FILE.PUT(otf, v1);
PERFORM UTL_FILE.NEW_LINE(otf);
PERFORM UTL_FILE.FFLUSH(otf);

inf := UTL_FILE.FCLOSE(inf);
otf := UTL_FILE.FCLOSE(otf);

PERFORM UTL_FILE.FCOPY(mydir, outfile, mydir, copyfile, 2, 3);
PERFORM UTL_FILE.FRENAME(mydir, outfile, mydir, 'rename.txt');

END;
$$ LANGUAGE plpgsql;

SELECT gen_file('/home/pgsql', 'input.txt', 'output.txt', 'copyfile.txt');

```

### 3. 后处理

如果你移除使用UTL\_FILE的一个作业，需要从UTL\_FILE.UTL\_FILE\_DIR表中删除该目录的信息。确保在删除之前该目录的信息不再被其他作业使用。

参考“注册和删除目录”以获取有关如何删除该目录的信息。

## H. 56. 6. 14. UTL\_RAW

### 概述

提供用于操作RAW数据类型的SQL函数。

Table H. 134. UTL\_RAW 特性

特性	描述
BIT_AND	对RAW r1和RAW r2中的值执行按位逻辑“与”操作，并返回“与”结果RAW。
BIT_COMPLEMENT	对RAW r中的值执行按位逻辑“补码”操作，并返回“补码”结果RAW。
BIT_OR	对RAW r1和RAW r2中的值执行按位逻辑“或”操作，并返回“或”结果RAW。
BIT_XOR	对RAW r1和RAW r2中的值执行按位逻辑“异或”操作，并返回“异或”结果RAW。
CAST_FROM_BINARY_INTEGER	返回BINARY_INTEGER值的RAW二进制表示。
CAST_FROM_NUMBER	返回一个数字值的原始二进制表示。
CAST_TO_BINARY_INTEGER	将一个BINARY_INTEGER的原始二进制表示转换为BINARY_INTEGER。
CAST_TO_NUMBER	将一个数字的原始二进制表示转换为数字。
CAST_TO_RAW	将一个文本值转换为一个RAW值。
CAST_TO_VARCHAR2	将一个RAW值转换为文本值。
COMPARE	比较RAW r1和RAW r2。
CONCAT	将最多12个RAW连接成一个RAW。

特性	描述
CONVERT	将RAW r从字符集from_charset转换为字符集to_charset, 返回结果RAW。
COPIES	返回n个连接在一起的r的副本。
LENGTH	返回RAW r的字节长度。
OVERLAY	从目标RAW的字节位置pos开始, 使用覆盖RAW覆盖目标RAW的指定部分, 覆盖长度为len个字节。
REVERSE	反转RAW r中的字节序列。
TRANSLITERATE	根据从集合RAW from_set和集合RAW to_set中的字节, 翻译输入RAW r中的字节。
TRANSLATE	根据从集合RAW from_set和集合RAW to_set中的字节, 转换输入RAW r中的字节。
XRANGE	创建一系列RAW值, 从start到end, 步长为step。
XRANGE	返回一个包含所有连续有效1字节编码的RAW, 从start_byte开始, 到end_byte结束。

#### 语法

```

CAST_TO_VARCHAR2(r raw) RETURNS TEXT
CAST_TO_RAW(c TEXT) RETURNS RAW
CONCAT(r1 RAW, r2 RAW, r3 RAW, r4 RAW, r5 RAW, r6 RAW, r7 RAW, r8 RAW, r9 RAW, r10 RAW, r11
RAW, r12 RAW) RETURNS RAW
LENGTH(r RAW) RETURNS INT4
SUBSTR(r RAW, pos INT4, len INT4 DEFAULT null) RETURNS RAW
TRANSLITERATE(r RAW, to_set RAW DEFAULT ''::raw, from_set RAW DEFAULT ''::raw, pad
RAW DEFAULT '00'::raw) RETURNS RAW
TRANSLATE(r RAW, from_set RAW, to_set RAW) RETURNS RAW
COPIES(r RAW, n INT8) RETURNS RAW
OVERLAY(overlay_str RAW, target RAW, pos INT4 DEFAULT 1, len INT4 DEFAULT NULL, pad
RAW DEFAULT '00'::raw) RETURNS RAW
XRANGE(start_byte RAW, end_byte RAW) RETURNS RAW
REVERSE(r RAW) RETURNS RAW
COMPARE(r1 RAW, r2 RAW, pad RAW) RETURNS INT4
CONVERT(r RAW, to_charset TEXT, from_charset TEXT) RETURNS RAW
BIT_AND(r1 RAW, r2 RAW) RETURNS RAW
BIT_OR(r1 RAW, r2 RAW) RETURNS RAW
BIT_XOR(r1 RAW, r2 RAW) RETURNS RAW
BIT_COMPLEMENT(r RAW) RETURNS RAW
CAST_TO_NUMBER(r RAW) RETURNS NUMERIC
CAST_FROM_NUMBER(n NUMERIC) RETURNS RAW
CAST_TO_BINARY_INTEGER(r RAW, endianess INT) RETURNS INT4
CAST_FROM_BINARY_INTEGER(n INT4, endianess INT) RETURNS RAW

```

#### H. 56. 6. 14. 1. 功能描述

本节介绍dbms\_job的每个功能。

##### CAST\_TO\_VARCHAR2

- 这个函数将一个使用一些数据字节表示的RAW值转换为一个具有相同数量数据字节的文本值。



示例

```
select utl_raw.cast_to_varchar2('43616D65726F6E') from dual;
```

CAST\_TO\_RAW

- 这个函数将一个使用一些数据字节表示的文本值转换为一个具有相同数量数据字节的RAW值。数据本身不做任何修改，但其数据类型被重新转换为RAW数据类型。

示例

```
select utl_raw.cast_to_raw('测试') from dual;
```

CONCAT

- 这个函数将最多12个RAW连接成一个RAW。

示例

```
select utl_raw.concat('1', '0102', 'f', '1a2b') from dual;
```

LENGTH

- 这个函数返回RAW r的字节长度。

示例

```
select UTL_RAW.length('FFFF'::raw) from dual;
```

SUBSTR

- 如果pos为正数，则SUBSTR从r的开头开始计数以找到第一个字节。如果pos为负数，则SUBSTR从r的末尾开始向后计数。pos的值可以为0。

示例

```
select utl_raw.substr('0102030405', 3, 2) from dual;
```

TRANSLITERATE

- 这个函数根据从集合RAW from\_set和集合RAW to\_set中的字节，转换输入RAW r中的字节。r中的连续字节被查找from\_set，在查找不到时，未更改地复制到结果RAW中。如果找到，则它们将被对应的to\_set中的字节或填充字节替换为结果RAW中，当没有对应关系时。

示例

```
select utl_raw.transliterate('010203040502', '0709', '01020304', 'ff') from dual;
```

TRANSLATE

- 这个函数根据从集合RAW from\_set和集合RAW to\_set中的字节，转换输入RAW r中的字节。如果r中的一个字节在from\_set中有一个匹配字节，则它将被相应位置to\_set中的字节替换，或删除。

示例

```
select utl_raw.translate( '0102030405', '0304', '09' ) from dual;
```

COPIES

- 这个函数返回n个r的副本，按顺序连接在一起。

示例

```
select utl_raw.copies( '010203', 3 ) from dual;
```

OVERLAY

- OVERLAY函数将目标RAW的指定部分用overlay\_str RAW替换，从目标的字节位置pos开始，继续len个字节。
- 如果overlay\_str少于len个字节，则使用填充字节将其扩展到len个字节。如果overlay\_str超过len个字节，则忽略overlay\_str中的额外字节。如果从目标的位置pos开始的len个字节超过了目标的长度，则目标被扩展以包含整个overlay\_str的长度。

示例

```
select utl_raw.overlay( 'aabb', '010203', 5, 1, 'FF' ) from dual;
```

XRANGE

- 此函数返回一个RAW值，其中包含以指定字节代码开头和结尾的一字节编码的连续。
- 如果start\_byte的值大于end\_byte的值，则结果字节的连续开始于start\_byte，通过x' FF' 回到x' 00'，然后以end\_byte结束。

示例

```
select utl_raw.xrange( 'FA', '01' ) from dual;
```

REVERSE

- 此函数将RAW r的字节序列从末尾到开头进行反转。例如，x' 0102F3' 将被反转为x' F30201'，而' xyz' 将被反转为' zyx'。结果长度与输入RAW长度相同。

示例

```
select utl_raw.reverse( '010203040506070809' ) from dual;
```

COMPARE

- 此函数比较两个RAW值。如果它们的长度不同，则根据可选的填充参数在右侧扩展较短的值。

示例

```
select utl_raw.compare( '010203', '01020304', '04' ) from dual;
```

转换

- 此函数将RAW r从字符集from\_charset转换为字符集to\_charset，并返回转换后的RAW。
- from\_charset和to\_charset必须是LightDB服务器定义的支持字符集。

示例

```
select utl_raw.convert(rawout('测试'::BYTEA)::text::raw,'GBK','UTF8') from dual;
```

BIT\_AND

- 此函数对RAW r1和RAW r2中的值执行位逻辑“与”操作，并返回“与”结果RAW。
- 如果r1和r2长度不同，则两个RAW中长度较短的一个将通过添加'00'来扩展到相同的长度（这与Oracle不同）。

示例

```
select utl_raw.bit_and('1234ffdd','fff1234f') from dual;
```

BIT\_OR

- 此函数对RAW r1和RAW r2中的值执行位逻辑“或”操作，并返回“或”结果RAW。
- 如果r1和r2长度不同，则两个RAW中长度较短的一个将通过添加'00'来扩展到相同的长度（这与Oracle不同）。

示例

```
select utl_raw.bit_or('1234ffdd','fff1234f') from dual;
```

BIT\_XOR

- 此函数对RAW r1和RAW r2中的值执行位逻辑“异或”操作，并返回“异或”结果RAW。
- 如果r1和r2长度不同，则两个RAW中长度较短的一个将通过添加'00'来扩展到相同的长度（这与Oracle不同）。

示例

```
select utl_raw.bit_xor('1234ffdd','1234ffee') from dual;
```

BIT\_COMPLEMENT

- 此函数对RAW r中的值执行位逻辑“补码”操作，并返回补码结果RAW。结果长度等于输入RAW r的长度。

示例

```
select UTL_raw.bit_complement('1122FF') from dual;
```

#### CAST\_TO\_NUMBER

- 此函数将NUMBER的RAW二进制表示转换为NUMBER。

示例

```
select utl_raw.cast_TO_number('C10215') from dual;
```

#### CAST\_FROM\_NUMBER

- 此函数返回NUMBER值的RAW二进制表示。

示例

```
select UTL_raw.cast_from_number(1.2) from dual;
```

#### CAST\_TO\_BINARY\_INTEGER

- 此函数将整数的RAW二进制表示转换为整数。(Oracle中为BINARY\_INTEGER)

示例

```
select utl_raw.cast_to_binary_integer('FF00') from dual;
```

#### CAST\_FROM\_BINARY\_INTEGER

- 此函数返回整数值的RAW二进制表示。(Oracle中为BINARY\_INTEGER)

示例

```
select utl_raw.cast_from_binary_integer(65280) from dual;
```

## H. 56. 6. 15. UTL\_URL

概述

有两个函数提供对URL字符进行转义和反转义的机制。

Table H. 135. UTL\_URL Features

特性	描述
ESCAPE	返回一个URL，其中的非法字符（可选保留字符）使用%2-digit-hex-code格式进行转义。
UNESCAPE	将URL中的转义字符序列恢复为其原始形式。将%XX转义字符序列转换为原始字符。

语法

```
ESCAPE(url TEXT, escape_reserved_chars BOOL DEFAULT FALSE, url_charset TEXT
DEFAULT 'UTF8') RETURNS TEXT
UNESCAPE(url TEXT, url_charset TEXT DEFAULT 'UTF8') RETURNS TEXT
```

### H. 56. 6. 15. 1. 特性描述

本节介绍dbms\_job的每个特性。

ESCAPE

- 此函数返回一个URL，其中的非法字符（可选保留字符）使用%2-digit-hex-code格式进行转义。

示例

```
select utl_url.escape('http://新年好.com') from dual;
```

UNESCAPE

- 此函数将URL中的转义字符序列恢复为其原始形式，将%XX转义字符序列转换为原始字符。

示例

```
select utl_url.unescape('http%3A%2F%2F%E6%96%B0%E5%B9%B4%E5%A5%BD.com', 'utf8')
from dual;
```

### H. 56. 6. 16. UTL\_ENCODE

概述

提供将原始数据编码为标准编码格式的函数。

Table H. 136. UTL\_ENCODE 特性

特性	描述
BASE64_DECODE	读取 base64 编码的原始输入字符串，并将其解码为其原始的原始字符串值。
BASE64_ENCODE	将原始值的二进制表示编码为 base 64 元素，并以原始字符串形式返回它。
MIMEHEADER_DECODE	从 MIME 标头格式解码字符串。
MIMEHEADER_ENCODE	将字符串编码为 MIME 标头格式。
QUOTED_PRINTABLE_DECODE	读取文本 quoted printable 格式的输入字符串，并将其解码为相应的原始字符串。
QUOTED_PRINTABLE_ENCODE	读取原始输入字符串并将其编码为相应的 quoted printable 格式字符串。
TEXT_DECODE	解码一个字符集敏感的文本字符串。
TEXT_ENCODE	编码一个字符集敏感的文本字符串。
UUDECODE	读取原始 uuencode 格式的输入字符串，并将其解码为相应的原始字符串。
UUDECODE	读取原始输入字符串并将其编码为相应的 uuencode 格式字符串。

语法

```

BASE64_DECODE(r RAW) RETURNS RAW
BASE64_ENCODE(r RAW) RETURNS RAW
MIMEHEADER_DECODE(buf TEXT) RETURNS TEXT
MIMEHEADER_ENCODE(buf TEXT, encode_charset TEXT default 'UTF8'::text, encoding
  INT4 default 2::int4) RETURNS TEXT
QUOTED_PRINTABLE_DECODE(r RAW) RETURNS RAW
QUOTED_PRINTABLE_ENCODE(r RAW) RETURNS RAW
TEXT_DECODE(buf TEXT, encode_charset TEXT default 'UTF8'::text, encoding INT4
  default 2::int4) RETURNS TEXT
TEXT_ENCODE(buf TEXT, encode_charset TEXT default 'UTF8'::text, encoding INT4
  default 2::int4) RETURNS TEXT
UUDECODE(r RAW) RETURNS RAW
UUENCODE(r RAW, type INT default 1::int, filename TEXT default
  'uuencode.txt', permission TEXT default '0') RETURNS RAW

```

### H. 56. 6. 16. 1. 特性描述

本节介绍 dbms\_job 的每个特性。

#### BASE64\_DECODE

- 此函数读取 base64 编码的原始输入字符串，并将其解码为其原始的原始字符串值。

示例

```
select utl_encode.base64_decode('3572574C364B2B56') from dual;
```

#### BASE64\_ENCODE

- 此函数将原始值的二进制表示编码为 base 64 元素，并以原始字符串形式返回它。

示例

```
select utl_encode.base64_encode(rawtohex('测试'))::raw from dual;
```

#### MIMEHEADER\_DECODE

- 从 MIME 标头格式(=?<charset> ?<encoding> ?<encoded text>?) 解码字符串。

示例

```
select UTL_ENCODE.mimeheader_decode('=?UTF-8?Q?
What=20is=20the=20date=20=E7=8E=8B=20=3D20=20/=20\=3F?=' ) from dual;
```

#### MIMEHEADER\_ENCODE

- 此函数生成一个形式为(=?<charset> ?<encoding> ?<encoded text>?) 的“编码单词”作为输出。
- ENCODING 输入参数接受 UTL\_ENCODE.QUOTED\_PRINTABLE 或 UTL\_ENCODE.BASE64 或 NULL 作为有效值。如果为 NULL，则选择 quoted-printable 编码作为默认值。

- <charset> 值作为输入参数 encode\_charset 指定。

示例

```
select utl_encode.MIMEHEADER_ENCODE('What is the date 王 =20 / \?',
  encode_charset =>'UTF8', encoding => 1 ) from dual;
```

#### QUOTED\_PRINTABLE\_DECODE

- 该函数读取原始的 quoted printable 格式的输入字符串，并将其解码为相应的原始字符串。

示例

```
select
  utl_encode.quoted_printable_decode(' 3D45353D42453D3830313233345C61732064663D33444142')
  from dual;
```

#### QUOTED\_PRINTABLE\_ENCODE

- 此函数将读取原始输入字符串并将其编码为相应的Quoted-Printable格式字符串。

示例

```
select
  utl_encode.quoted_printable_decode(' 3D45353D42453D3830313233345C61732064663D33444142')
  from dual;
```

#### TEXT\_DECODE

- 此函数将把输入的文本转换为目标字符集，该字符集由encode\_charset参数指定，如果 encode\_charset参数不为NULL。编码后的文本将被转换为数据库的基本字符集。
- 您可以根据每个编码参数选择解码Quoted-Printable或Base64格式。 1表示Base64编码，2表示Quoted-Printable编码，默认为Quoted-Printable格式。

示例

```
select utl_encode.text_decode('=CD=F91234\as df=3DAB', encode_charset =>
  'GBK', encoding => 2) from dual;
```

#### TEXT\_ENCODE

- 此函数将把输入的文本转换为目标字符集，该字符集由encode\_charset参数指定，如果 encode\_charset参数不为NULL。文本将根据encoding参数指定的格式编码为Base64或Quoted-Printable格式。

示例

```
select utl_encode.text_encode(' 往12\as df=AB', encode_charset => 'GBK', encoding
  => 1) from dual;
```

#### UUDECODE

- 此函数将读取RAW Uuencode格式的输入字符串并将其解码为相应的RAW字符串。

示例

```
select UTL_ENCODE.uudecode(rawtohex(' ,6[C9&5FA$R,S0`'))::raw) from dual;
```

UUENCODE

- 此函数将读取原始输入字符串并将其编码为相应的Uuencode格式字符串。

示例

```
select UTL_ENCODE.uencode('5BB8D91959A113233340')::raw) from dual;
```

## H. 56. 7. 事务行为

大多数事务行为都是完全相同的，但以下内容不同。

### H. 56. 7. 1. 处理语句失败。

```
create table t (a int primary key, b int);
begin;
insert into t values(1, 1);
insert into t values(1, 1);
commit;
```

Oracle: 提交操作可能会成功。这之后，t表将有1行记录。

LightDB: 由于第二个插入操作失败，提交失败。因此，t表没有任何记录。

### H. 56. 7. 2. 带子查询的DML

情况1:

```
create table dml(a int, b int);
insert into dml values(1, 1), (2, 2);

-- session 1:
begin;
delete from dml where a in (select min(a) from dml);

--session 2:
delete from dml where a in (select min(a) from dml);

-- session 1:
commit;
```

在Oracle中: sess2中删除了1行记录。因此最终dml中没有任何记录。

在LightDB中: sess2中没有删除任何记录，因此最终dml中有1行记录。

Oracle可能会检测到min(a)已更改，然后回滚/重新运行该语句。



相同的原因也可能导致以下不同之处。

```

create table su (a int, b int);
insert into su values(1, 1);

- session 1:
begin;
update su set b = 2 where b = 1;

- sess 2:
select * from su where a in (select a from su where b = 1) for update;

- sess 1:
commit;
    
```

在Oracle中，未选择任何记录。在LightDB中，选择了1行记录（1, 2）。

最佳实践是在DML和SELECT ... FOR UPDATE中永远不要使用子查询。即使在Oracle中，行为也不一致。在Oracle 11.2.0.1和11.2.0.3之间，可能的行为与LightDB相同，但其他版本则不同。

## H. 56. 8. 系统视图参考

这些视图基于Oracle 11g，但一些视图包含新版本的一些字段。

视图在Oracle和sys模式下创建。

为了与Oracle兼容，以下视图具有与Oracle中相应视图相同的字段，但对于那些无法获取或难以获取其字段值的字段返回NULL值。

支持的字段如下所示。某些字段的含义与Oracle不同。

### H. 56. 8. 1. [DBA/ALL/USER]\_SEQUENCES

- DBA\_SEQUENCES描述了数据库中的所有序列。
- ALL\_SEQUENCES视图目前与DBA\_SEQUENCES视图相同，并且不像Oracle那样进行权限验证。
- USER\_SEQUENCES描述了当前用户拥有的所有序列。

Table H. 137. ALL\_SEQUENCES

列名	描述
SEQUENCE_OWNER	序列所有者
SEQUENCE_NAME	序列名称
MIN_VALUE	序列的最小值
MAX_VALUE	序列的最大值
INCREMENT_BY	序列增加的值
CYCLE_FLAG	指示序列在达到限制时是否循环（Y）或不循环（N）
ORDER_FLAG	指示序列号是否按顺序生成（Y）或不按顺序生成（N）。在LightDB下，始终为Y。
CACHE_SIZE	缓存的序列号数量
LAST_NUMBER	写入磁盘的最后一个序列号。

## H. 56. 8. 2. [DBA/ALL/USER]\_SYNONYMS

- LightDB尚不支持SYNONYMS，因此这些视图目前为空。

## H. 56. 8. 3. [DBA/ALL/USER]\_TAB\_COLS, COLS, [DBA/ALL/USER]\_TAB\_COLUMNS

- DBA\_TAB\_COLS描述了数据库中所有表和视图的列。
- ALL\_TAB\_COLS视图目前与DBA\_TAB\_COLS视图相同，并且不像Oracle那样进行权限验证。
- USER\_TAB\_COLS描述了当前用户拥有的表和视图的列。此视图不显示OWNER列。
- COLS与USER\_TAB\_COLS相同。
- DBA\_TAB\_COLUMNS描述了数据库中所有表和视图的列。
- ALL\_TAB\_COLUMNS视图目前与DBA\_TAB\_COLUMNS视图相同，并且不像Oracle那样进行权限验证。
- USER\_TAB\_COLUMNS描述了当前用户拥有的表和视图的列。此视图不显示OWNER列。
- [DBA/ALL/USER]\_TAB\_COLS视图与[DBA/ALL/USER]\_TAB\_COLUMNS视图不同，因为不会过滤隐藏列  
(HIDDEN\_COLUMN, VIRTUAL\_COLUMN, SEGMENT\_COLUMN\_ID, INTERNAL\_COLUMN\_ID)。

Table H. 138. ALL\_TAB\_COLS

列名	描述
OWNER	表、视图的所有者
TABLE_NAME	表、视图的名称
COLUMN_NAME	列名
DATA_TYPE	列的数据类型
DATA_TYPE_OWNER	列的数据类型的所有者
DATA_LENGTH	列的长度（字符类型单位为字符数；其他类型为字节数，对于number/numeric 类型，长度总是22）
DATA_PRECISION	NUMBER数据类型的小数精度；FLOAT数据类型 的二进制精度；其他数据类型的值为NULL。
DATA_SCALE	小数点右侧的数字位数
NULLABLE	指示列是否允许为空。如果列有NOT NULL约 束或列是主键的一部分，则值为N。
COLUMN_ID	列的创建顺序号
DEFAULT_LENGTH	列默认值的长度
DATA_DEFAULT	列的默认值
NUM_DISTINCT	列中不同值的数量
NUM_NULLS	列中的空值数量
AVG_COL_LEN	列的平均长度（单位为字节）
CHAR_LENGTH	显示列的字符长度
CHAR_USED	指示列使用字节长度语义（B）或字符长度语 义（C），或者数据类型不是以下任何一种 （NULL）

#### H. 56. 8. 4. [DBA/ALL/USER]\_TAB\_COL\_STATISTICS

- DBA\_TAB\_COL\_STATISTICS包含从“DBA\_TAB\_COLUMNS”中提取的列统计信息和直方图信息。它的列与“ALL\_TAB\_COL\_STATISTICS”中的列相同。
- ALL\_TAB\_COL\_STATISTICS视图目前与DBA\_SEQUENCES视图相同，且不像Oracle那样进行权限验证。
- USER\_TAB\_COL\_STATISTICS显示从“USER\_TAB\_COLUMNS”提取的此类信息。该视图不显示OWNER列。

Table H. 139. ALL\_TAB\_COL\_STATISTICS

Column	Description
OWNER	表的所有者
TABLE_NAME	表的名称
COLUMN_NAME	列名称
NUM_DISTINCT	列中不同值的数量
NUM_NULLS	列中NULL的数量
AVG_COL_LEN	列的平均长度（字节为单位）

#### H. 56. 8. 5. [DBA/ALL/USER]\_OBJECTS, OBJ

- DBA\_OBJECTS描述了数据库中的所有对象。其列与“ALL\_OBJECTS”中的列相同。
- ALL\_OBJECTS视图目前与DBA\_OBJECTS视图相同，且不像Oracle那样进行权限验证。
- USER\_OBJECTS描述了当前用户拥有的所有对象。该视图不显示OWNER列。
- OBJ与USER\_OBJECTS相同。

Table H. 140. ALL\_OBJECTS

Column	Description
OWNER	对象的所有者
OBJECT_NAME	对象的名称
SUBOBJECT_NAME	子对象的名称（例如，分区），如果不存在则为NULL
OBJECT_ID	对象的字典对象编号
DATA_OBJECT_ID	包含该对象的段的字典对象编号
OBJECT_TYPE	对象的类型
STATUS	对象的状态：VALID, INVALID
TEMPORARY	表示对象是否为临时对象（当前会话只能看到自己放置在该对象中的数据）（Y）或不是（N）

#### H. 56. 8. 6. [DBA/ALL/USER]\_CATALOG

- DBA\_CATALOG列出数据库中的所有表、视图和序列（不包括聚簇和同义词）。
- ALL\_CATALOG视图目前与DBA\_CATALOG视图相同，且不像Oracle那样进行权限验证。
- USER\_CATALOG 显示当前用户模式中的表、视图和序列。
- Oracle 视图中的所有列都得到支持。

Table H. 141. ALL\_CATALOG

列名	描述
OWNER	表、视图或序列的拥有者
TABLE_NAME	表、视图或序列的名称
TABLE_TYPE	表、视图或序列的类型

### H. 56. 8. 7. DICTIONARY, DICT

- DICTIONARY/DICT 包含数据字典表和视图的描述。

Table H. 142. DICTIONARY

Column	Description
TABLE_NAME	对象的名称
COMMENTS	对象的文本注释

### H. 56. 8. 8. [DBA/ALL/USER]\_DEPENDENCIES

- DBA\_DEPENDENCIES 描述了数据库中所有过程、包、函数、包体和触发器之间的依赖关系，包括对未使用任何数据库链接创建的视图的依赖关系。该视图不会显示 SCHEMAID 列。
- ALL\_DEPENDENCIES视图目前与DBA\_DEPENDENCIES视图相同，并且不像Oracle一样进行权限验证。
- USER\_DEPENDENCIES描述了当前用户模式中对象之间的依赖关系。此视图不显示OWNER列。

Table H. 143. ALL\_DEPENDENCIES

列名	描述
OWNER	对象的所有者
NAME	对象的名称
类型	对象的类型
REFERENCED_OWNER	引用对象的所有者（如果是远程对象，则为远程所有者）
REFERENCED_NAME	引用对象的名称
REFERENCED_TYPE	引用对象的类型
DEPENDENCY_TYPE	依赖关系的类型（oracle: 表示依赖关系是否为REF依赖关系（REF），或者不是（HARD））

### H. 56. 8. 9. [DBA/ALL/USER]\_SOURCE

- DBA\_SOURCE描述数据库中所有存储对象的文本来源。
- ALL\_SOURCE视图目前与DBA\_SOURCE视图相同，且没有像oracle那样进行权限验证。
- USER\_SOURCE描述当前用户所拥有的存储对象的文本来源。此视图不显示OWNER列。

Table H. 144. ALL\_SOURCE

列名	描述
OWNER	对象的所有者

列名	描述
NAME	对象的名称
TYPE	对象类型: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, TRIGGER
LINE	此行源代码的行号
TEXT	存储对象的文本来源

### H. 56. 8. 10. [DBA/ALL/USER]\_PROCEDURES

- DBA\_PROCEDURES列出所有函数和过程，以及相关属性。
- ALL\_PROCEDURES视图目前与DBA\_PROCEDURES视图相同，且没有像oracle那样进行权限验证。
- USER\_PROCEDURES列出当前用户拥有的所有函数和过程，以及相关属性。它不包含OWNER列。
- 这些视图指示函数是否为管道化函数、并行启用或聚合函数。如果函数是管道化函数或聚合函数，则还会确定相关的实现类型（如果有）。

Table H. 145. ALL\_PROCEDURES

列名	描述
OWNER	过程的所有者
OBJECT_NAME	对象名称: 顶层函数、过程或包名称
OBJECT_ID	对象的对象编号
OBJECT_TYPE	对象类型的名称
AGGREGATE	指示过程是否为聚合函数 (YES) 还是不是 (NO)
PARALLEL	指示过程或函数是否启用并行 (YES) 还是不启用 (NO)
DETERMINISTIC	如果过程/函数被声明为确定性的，则为 YES; 否则为NO

### H. 56. 8. 11. [DBA/ALL/USER]\_TRIGGERS

- DBA\_TRIGGERS描述数据库中的所有触发器。
- ALL\_TRIGGERS视图目前与DBA\_TRIGGERS视图相同，且没有像oracle那样进行权限验证。
- USER\_TRIGGERS描述当前用户拥有的触发器。

Table H. 146. ALL\_TRIGGERS

列名	描述
OWNER	触发器的所有者
TRIGGER_NAME	触发器名称
TRIGGER_TYPE	触发器触发的时间: BEFORE STATEMENT, BEFORE EACH ROW, AFTER STATEMENT, AFTER EACH ROW, INSTEAD OF EACH ROW
TRIGGERING_EVENT	触发触发器的DML
TABLE_OWNER	定义触发器的表的所有者

列名	描述
BASE_OBJECT_TYPE	定义触发器的基础对象：TABLE，VIEW
TABLE_NAME	指示定义触发器的表或视图名称
REFERENCING_NAMES	在触发器内部用于引用OLD和NEW列值的名称
WHEN_CLAUSE	必须为TRUE才能执行TRIGGER_BODY
STATUS	指示触发器是否启用（ENABLED）或禁用（DISABLED）；禁用的触发器将不会触发
ACTION_TYPE	触发器主体的操作类型：'PL/SQL'
TRIGGER_BODY	触发器触发时执行的语句
BEFORE_STATEMENT	指示触发器是否有BEFORE STATEMENT部分（YES）或没有（NO）
BEFORE_ROW	指示触发器是否有BEFORE EACH ROW部分（YES）或没有（NO）
AFTER_ROW	指示触发器是否有AFTER EACH ROW部分（YES）或没有（NO）
AFTER_STATEMENT	指示触发器是否有AFTER STATEMENT部分（YES）或没有（NO）
INSTEAD_OF_ROW	指示触发器是否有INSTEAD OF部分（YES）或没有（NO）

#### H. 56. 8. 12. [DBA/ALL/USER]\_TRIGGER\_COLS

- DBA\_TRIGGER\_COLS描述数据库中所有触发器中列的使用情况。
- ALL\_TRIGGER\_COLS视图目前与DBA\_TRIGGER\_COLS视图相同，且没有像oracle那样进行权限验证。
- USER\_TRIGGER\_COLS描述当前用户拥有的触发器中列的使用情况，以及当前用户所拥有的表上的触发器中列的使用情况。

Table H. 147. ALL\_TRIGGER\_COLS

列名	描述
TRIGGER_OWNER	触发器的所有者
TRIGGER_NAME	触发器的名称
TABLE_OWNER	定义触发器的表的所有者
TABLE_NAME	定义触发器的表的名称
COLUMN_NAME	在触发器中使用的列的名称

#### H. 56. 8. 13. [DBA/ALL/USER]\_TYPES

- DBA\_TYPES描述数据库中的所有对象类型。
- ALL\_TYPES视图目前与DBA\_TYPES视图相同，且没有像oracle那样进行权限验证。
- USER\_TYPES描述当前用户所拥有的对象类型。此视图不显示OWNER列。

Table H. 148. ALL\_TYPES

列名	描述
OWNER	类型的所有者

列名	描述
TYPE_NAME	类型名称
TYPE_OID	类型的对象标识符 (OID)
TYPECODE	类型的类型码 (Typecode)
PREDEFINED	指示该类型是否为预定义类型 (YES) 或不是 (NO)

#### H. 56. 8. 14. [DBA/ALL/USER]\_CONSTRAINTS

- DBA\_CONSTRAINTS描述了数据库中所有约束的定义。
- ALL\_CONSTRAINTS视图目前与DBA\_CONSTRAINTS视图相同，并且不像 Oracle 那样进行权限验证。
- USER\_CONSTRAINTS描述当前用户模式中表上的约束定义。

Table H. 149. ALL\_CONSTRAINTS

字段	描述
OWNER	约束定义的所有者
CONSTRAINT_NAME	约束定义的名称
CONSTRAINT_TYPE	约束定义的类型 c - 表上的 Check 约束 p - 主键 u - 唯一键 f - 涉及 REF 列的约束 t - 触发器约束 t - 排它性约束
TABLE_NAME	与约束定义关联的表 (或视图) 的名称
SEARCH_CONDITION	Check 约束的搜索条件文本
R_OWNER	引用约束中所涉及的表的所有者
DELETE_RULE	引用约束的删除规则: 'NO ACTION'、'RESTRICT'、'CASCADE'、'SET NULL'、'SET DEFAULT'
STATUS	约束的执行状态: 'ENABLED'、'DISABLED'
DEFERRABLE	指示约束是否可延迟 (DEFERRABLE) 或不可延迟 (NOT DEFERRABLE)
DEFERRED	指示约束是否最初被延迟 (DEFERRED) 或不是 (IMMEDIATE)
VALIDATED	约束是否已验证? 目前, 仅对外键和 CHECK 约束可以为 false
INDEX_OWNER	拥有该索引的用户名称
INDEX_NAME	索引的名称 (仅对唯一、主键、外键或排它性约束显示)
INVALID	指示约束是否无效 (INVALID) 或不是 (NULL)

### H. 56. 8. 15. [DBA/ALL/USER]\_CONS\_COLUMNS

- DBA\_CONS\_COLUMNS描述了数据库中在约束中指定的所有列。
- ALL\_CONS\_COLUMNS视图目前与DBA\_CONS\_COLUMNS视图相同，并且不像 Oracle 那样进行权限验证。
- USER\_CONS\_COLUMNS描述了当前用户拥有并在约束中指定的列。
- Oracle支持所有列类型。

Table H. 150. ALL\_CONS\_COLUMNS

字段	描述
OWNER	约束定义的所有者
CONSTRAINT_NAME	约束定义的名称
TABLE_NAME	约束定义所在的表名称
COLUMN_NAME	约束定义中指定的列或对象类型列的属性名称
POSITION	列或属性在对象定义中的原始位置

### H. 56. 8. 16. [DBA/ALL/USER]\_VIEWS

- DBA\_VIEWS描述了数据库中的所有视图。
- ALL\_VIEWS视图目前与DBA\_VIEWS视图相同，并且不像 Oracle 那样进行权限验证。
- USER\_VIEWS描述当前用户拥有的视图。该视图不显示OWNER列。

Table H. 151. ALL\_VIEWS

字段	描述
OWNER	视图的所有者
VIEW_NAME	视图的名称
TEXT_LENGTH	视图文本的长度
TEXT	视图文本

### H. 56. 8. 17. [DBA/ALL/USER]\_TABLES, [DBA/ALL/USER]\_ALL\_TABLES, TAB

- DBA\_ALL\_TABLES描述了数据库中的所有对象表和关系表。
- ALL\_ALL\_TABLES视图目前与DBA\_ALL\_TABLES视图相同，并且不像 Oracle 那样进行权限验证。
- USER\_ALL\_TABLES描述当前用户拥有的对象表和关系表。该视图不显示OWNER列。
- [DBA/ALL/USER]\_TABLES视图不显示OBJECT\_ID\_TYPE、TABLE\_TYPE\_OWNER和TABLE\_TYPE列。
- [DBA/ALL/USER]\_TABLES有RESULT\_CACHE列，但[DBA/ALL/USER]\_ALL\_TABLES没有。
- TABS与USER\_TABLES相同。



Table H. 152. ALL\_ALL\_TABLES

字段	描述
OWNER	表的所有者
TABLE_NAME	表的名称
TABLESPACE_NAME	表所在表空间的名称。
LOGGING	表示表的更改是否被记录日志。
NUM_ROWS	表中的行数。
BLOCKS	表中已使用的块数。
AVG_ROW_LEN	平均行长度，包括行开销。
LAST_ANALYZED	表最近分析日期。
PARTITIONED	表示表是否被分区（YES）或未被分区（NO）。
TEMPORARY	表示表是否为临时表（Y）或非临时表（N）。
ROW_MOVEMENT	如果表被分区，表示行移动是否启用（ENABLED）或禁用（DISABLED）。  总是为“ENABLE”。

### H. 56. 8. 18. [DBA/ALL/USER]\_TAB\_STATISTICS

- DBA\_TAB\_STATISTICS 显示数据库中所有表的优化器统计信息。
- ALL\_TAB\_STATISTICS 视图目前与 DBA\_TAB\_STATISTICS 视图相同，而且不像 Oracle 那样进行权限验证。
- USER\_TAB\_STATISTICS 显示当前用户拥有的表的优化器统计信息。此视图不显示 OWNER 列。
- [DBA/ALL/USER]\_TABLES 视图不显示 OBJECT\_ID\_TYPE、TABLE\_TYPE\_OWNER 和 TABLE\_TYPE 列。

Table H. 153. ALL\_TAB\_STATISTICS

列名	描述
OWNER	对象的所有者。
TABLE_NAME	表名。
TABLESPACE_NAME	表所在表空间的名称。
PARTITION_NAME	分区名称。
PARTITION_POSITION	分区在表中的位置。
OBJECT_TYPE	对象类型： TABLE、PARTITION、SUBPARTITION。
NUM_ROWS	对象中的行数。
BLOCKS	对象中已使用的块数。
AVG_ROW_LEN	平均行长度，包括行开销。
LAST_ANALYZED	表最近分析日期。

### H. 56. 8. 19. [DBA/ALL/USER]\_TAB\_COMMENTS

- DBA\_TAB\_COMMENTS 显示数据库中所有表和视图的注释。

- ALL\_TAB\_COMMENTS 视图目前与 DBA\_TAB\_COMMENTS 视图相同，而且不像 Oracle 那样进行权限验证。
- USER\_TAB\_COMMENTS 显示当前用户拥有的表和视图的注释。此视图不显示 OWNER 列。

Table H.154. ALL\_TAB\_COMMENTS

列名	描述
OWNER	对象的所有者。
TABLE_NAME	对象名称。
TABLE_TYPE	对象类型。
COMMENTS	对象注释。

### H. 56. 8. 20. [DBA/ALL/USER]\_COL\_COMMENTS

- DBA\_COL\_COMMENTS 显示数据库中所有表和视图列的注释。
- ALL\_COL\_COMMENTS 视图目前与 DBA\_COL\_COMMENTS 视图相同，且不像 Oracle 一样进行权限验证。
- USER\_COL\_COMMENTS 显示当前用户拥有的表和视图的列的注释。该视图不显示 OWNER 列。

Table H.155. ALL\_COL\_COMMENTS

列名	描述
OWNER	对象的所有者
TABLE_NAME	对象的名称
COLUMN_NAME	列的名称
COMMENTS	列的名称

### H. 56. 8. 21. [DBA/ALL/USER]\_TAB\_MODIFICATIONS

- DBA\_TAB\_MODIFICATIONS 描述数据库中所有表的此类信息。
- ALL\_TAB\_MODIFICATIONS 视图目前与 DBA\_TAB\_MODIFICATIONS 视图相同，且不像 Oracle 一样进行权限验证。
- USER\_TAB\_MODIFICATIONS 描述当前用户拥有的表的此类信息。此视图不会显示 TABLE\_OWNER 列。

Table H.156. ALL\_TAB\_MODIFICATIONS

列名	描述
TABLE_OWNER	被修改表的所有者
TABLE_NAME	被修改表的名称
INSERTS	最近一次收集统计信息以来插入的近似数量
UPDATES	最近一次收集统计信息以来更新的近似数量
DELETES	最近一次收集统计信息以来删除的近似数量
TIMESTAMP	指示表上一次被修改的时间

### H. 56. 8. 22. [DBA/ALL/USER]\_INDEXES, IND

- DBA\_INDEXES 描述数据库中的所有索引。

- ALL\_INDEXES视图当前与DBA\_INDEXES视图相同，且不像oracle那样进行权限验证。
- USER\_INDEXES描述当前用户拥有的索引。该视图不显示OWNER列。
- IND与USER\_INDEXES相同。

Table H.157. ALL\_INDEXES

列	描述
OWNER	索引的所有者
INDEX_NAME	索引的名称
INDEX_TYPE	索引的类型，始终为NORMAL
TABLE_OWNER	索引对象的所有者
TABLE_NAME	索引对象的名称
TABLE_TYPE	索引对象的类型：TABLE, VIEW, INDEX, SEQUENCE
UNIQUENESS	指示索引是否唯一（UNIQUE）或非唯一（NONUNIQUE）
TABLESPACE_NAME	包含该索引的表空间名称
LOGGING	指示索引更改是否记录：'YES', 'NO'
DISTINCT_KEYS	索引值的不同数量
STATUS	指示非分区索引是否有效（VALID）或无用（UNUSABLE）
NUM_ROWS	索引中的行数
LAST_ANALYZED	此索引最近分析的日期
PARTITIONED	指示索引是否已分区（YES）或未分区（NO）
TEMPORARY	指示索引是否在临时表上（Y）或不在（N）

### H. 56. 8. 23. [DBA/ALL/USER]\_INDEX\_USAGE

- DBA\_INDEX\_USAGE 显示每个索引的累计统计信息。
- ALL\_INDEX\_USAGE 视图与 DBA\_INDEX\_USAGE 视图相同（该视图在 Oracle 中不存在）。
- USER\_INDEX\_USAGE 描述当前用户拥有的每个索引的累计统计信息（该视图在 Oracle 中不存在）。此视图不显示 OWNER 列。

Table H.158. DBA\_INDEX\_USAGE

列名	描述
OBJECT_ID	索引的对象 ID
NAME	索引名称
OWNER	索引所有者
TOTAL_ACCESS_COUNT	索引被访问的总次数
TOTAL_ROWS_RETURNED	索引返回的总行数

### H. 56. 8. 24. [DBA/ALL/USER]\_IND\_COLUMNS

- DBA\_IND\_COLUMNS 描述数据库中所有表的索引列。
- ALL\_IND\_COLUMNS 视图目前与 DBA\_IND\_COLUMNS 视图相同，而且与 Oracle 一样，不进行权限验证。

- USER\_IND\_COLUMNS 描述当前用户拥有的索引列以及当前用户拥有的表上的索引列。此视图不显示 INDEX\_OWNER 或 TABLE\_OWNER 列。

Table H. 159. ALL\_IND\_COLUMNS

列名	描述
INDEX_OWNER	索引所有者
INDEX_NAME	索引名称
TABLE_OWNER	表的所有者
TABLE_NAME	表名称
COLUMN_NAME	列名或对象类型列属性
COLUMN_POSITION	列或属性在索引中的位置
COLUMN_LENGTH	列的索引长度, 对于number/numeric 类型, 长度总是22
CHAR_LENGTH	列的最大代码点长度
DESCEND	指示列是否按降序 (DESC) 或升序 (ASC) 排序

#### H. 56. 8. 25. [DBA/ALL/USER]\_IND\_EXPRESSIONS

- DBA\_IND\_EXPRESSIONS 描述数据库中所有基于函数的索引的表达式。
- ALL\_IND\_EXPRESSIONS视图目前与DBA\_IND\_COLUMNS视图相同, 且与Oracle一样不进行权限验证。
- USER\_IND\_EXPRESSIONS描述当前用户拥有的表上基于函数的索引的表达式。该视图不显示 INDEX\_OWNER或TABLE\_OWNER列。

Table H. 160. ALL\_IND\_EXPRESSIONS

列名	描述
INDEX_OWNER	索引所有者
INDEX_NAME	索引名称
TABLE_OWNER	表所有者
TABLE_NAME	表名称
COLUMN_EXPRESSION	定义列的基于函数的索引表达式
COLUMN_POSITION	列或属性在索引中的位置

#### H. 56. 8. 26. [DBA/ALL/USER]\_IND\_STATISTICS

- DBA\_IND\_STATISTICS显示数据库中所有索引的优化器统计信息。
- ALL\_IND\_STATISTICS视图目前与DBA\_IND\_STATISTICS视图相同, 且与Oracle一样不进行权限验证。
- USER\_IND\_STATISTICS显示当前用户拥有的表上索引的优化器统计信息。该视图不显示 OWNER列。

Table H. 161. ALL\_IND\_STATISTICS

列名	描述
OWNER	索引所有者

列名	描述
INDEX_NAME	索引名称
TABLE_OWNER	索引对象所属者
TABLE_NAME	索引对象名称
PARTITION_NAME	定义列的基于函数的索引表达式
PARTITION_POSITION	列或属性在索引中的位置
OBJECT_TYPE	对象类型: INDEX, PARTITION  现在始终为“INDEX”，因为该视图不会获取分区信息
DISTINCT_KEYS	索引中不同键的数量
NUM_ROWS	索引中的行数
LAST_ANALYZED	最近一次分析索引的日期

### H. 56. 8. 27. [DBA/ALL/USER]\_USERS

- DBA\_USERS描述数据库中的所有用户，并包含比ALL\_USERS更多的列。
- ALL\_USERS 列出数据库中的所有用户（USERNAME, USER\_ID, CREATED），但不像 Oracle 那样进行权限验证。
- USER\_USERS 描述当前用户，并包含比 ALL\_USERS 更多的列。

Table H. 162. DBA\_USERS

列名	描述
USERNAME	用户名称
USER_ID	用户ID号
PASSWORD	此列已被弃用，建议使用 AUTHENTICATION_TYPE 列。
ACCOUNT_STATUS	账户状态，始终为“NORMAL”。
EXPIRY_DATE	账户过期日期

Table H. 163. ALL\_USERS

列名	描述
USERNAME	用户名称
USER_ID	用户ID号

### H. 56. 8. 28. DBA\_ROLES

- 描述数据库中所有角色。

Table H. 164. DBA\_ROLES

列名	描述
ROLE	角色名称
PASSWORD_REQUIRED	此列已被弃用，建议使用 AUTHENTICATION_TYPE 列，始终为“NO”。
AUTHENTICATION_TYPE	指示角色的身份验证机制，始终为“NONE”。

列名	描述
	NONE - CREATE ROLE role1;
COMMON	指示给定角色是否为公共角色。始终为“YES”，在 Oracle 11g 中不存在。

### H. 56. 8. 29. [DBA/USER]\_ROLE\_PRIVS

- DBA\_ROLE\_PRIVS 描述了授予数据库中所有用户和角色的角色。
- USER\_ROLE\_PRIVS 描述当前用户被授予的角色。

Table H. 165. DBA\_ROLE\_PRIVS

列名	描述
GRANTEE	授予的用户或角色名称
GRANTED_ROLE	授予的角色名称
ADMIN_OPTION	指示是否使用 ADMIN OPTION 进行了授权 (YES)，还是没有 (NO)。
DEFAULT_ROLE	指示角色是否被指定为用户的默认角色 (YES)，还是没有 (NO)。 始终为“NO”。
COMMON	指示授权方式。 始终为“YES”，在 Oracle 11g 中不存在。

### H. 56. 8. 30. PRODUCT\_COMPONENT\_VERSION

- 包含组件产品的版本和状态信息。

Table H. 166. PRODUCT\_COMPONENT\_VERSION

列名	描述
PRODUCT	产品名称
VERSION	版本号
STATUS	发布状态

### H. 56. 8. 31. PLAN\_TABLE

- PLAN\_TABLE 是自动创建的全局临时表，用于保存所有用户的 EXPLAIN PLAN 语句的输出结果。
- 由于 LightDB 不支持，因此此视图为空，以保证兼容性。

### H. 56. 8. 32. DBA\_DATA\_FILES

- 描述数据库文件。

Table H. 167. DBA\_DATA\_FILES

列名	描述
FILE_NAME	数据库文件名称
FILE_ID	数据库文件的文件标识号

列名	描述
TABSPACE_NAME	数据库文件所属的表空间名称
BYTES	文件的大小（字节）
BLOCKS	文件的大小（Oracle 块）
STATUS	文件状态：AVAILABLE 或 INVALID。 始终为“AVAILABLE”。
RELATIVE_FNO	相对文件编号
AUTOEXTENSIBLE	自动扩展指示器 始终为“YES”。
INCREMENT_BY	作为自动扩展增量使用的 Oracle 块数
ONLINE_STATUS	文件的在线状态 始终为“ONLINE”。

### H. 56. 8. 33. [DBA/ALL/USER]\_JOBS

- 从 lt\_cron 获取信息。
- DBA\_JOBS 描述数据库中的所有作业。
- USER\_JOBS 描述当前用户拥有的作业。
- ALL\_JOBS 与 USER\_JOBS 相同。

Table H. 168. DBA\_JOBS

列名	描述
JOB	作业的标识符
LOG_USER	提交作业时的登录用户
PRIV_USER	默认权限适用于此作业的用户
SCHEMA_USER	用于解析作业的默认模式 始终为'CRON'
LAST_DATE	此作业上一次成功执行的日期
LAST_SEC	与 LAST_DATE 相同。这是上一次成功执行开始的时间。 采用 'HH24:MI:SS' 格式。
THIS_DATE	此作业开始执行的日期（如果未执行则通常为空白）。
THIS_SEC	与 THIS_DATE 相同。这是上一次成功执行开始的时间。 采用 'HH24:MI:SS' 格式。
TOTAL_TIME	系统在上一次执行中为此作业花费的总挂钟时间  它与 Oracle 不同。在 Oracle 中，它表示自此作业首次执行以来系统在此作业上花费的总挂钟时间（以秒为单位），并且此值是累积的。

列名	描述
BROKEN	Y: 不尝试运行此作业, N: 尝试运行此作业始终为“N”。
INTERVAL	该字符串使用标准的 cron 语法, 参考 lt_cron。 与 Oracle 不同。
WHAT	LightDB 命令 与 Oracle 不同。

### H. 56. 8. 34. DBA\_JOBS\_RUNNING

- 从 lt\_cron 获取信息。
- 列出当前在实例中运行的所有作业。

Table H. 169. DBA\_JOBS\_RUNNING

列名	描述
SID	执行该作业的进程的标识符
JOB	作业的标识符。该作业当前正在执行
LAST_DATE	该作业上次成功执行的日期
LAST_SEC	和LAST_DATE相同。这是上次成功执行开始的时间 格式为'HH24:MI:SS'
THIS_DATE	该作业开始执行的日期（如果未执行, 则通常为空白）
THIS_SEC	和THIS_DATE相同。这是上次成功执行开始的时间 格式为'HH24:MI:SS'

### H. 56. 8. 35. [DBA/ALL/USER]\_MVIEWES

- DBA\_MVIEWES描述了数据库中的所有物化视图。
- ALL\_MVIEWES视图当前与DBA\_MVIEWES视图相同, 且不像oracle一样进行权限验证。
- USER\_MVIEWES描述了当前用户拥有的所有物化视图。

Table H. 170. ALL\_MVIEWES

列名	描述
OWNER	创建该物化视图的模式
MVIEW_NAME	物化视图的名称
CONTAINER_NAME	保存物化视图数据的容器的名称。通常与MVIEW_NAME相同。
QUERY	定义物化视图的查询语句
QUERY_LEN	定义查询的长度（以字节为单位）
UPDATABLE	指示物化视图是否可更新（Y）或不可更新（N）



列名	描述
	始终为'N'
REFRESH_MODE	物化视图的刷新模式 始终为'DEMAND'，这意味着当调用适当的刷新过程时，数据库会刷新此物化视图。
REFRESH_METHOD	用于刷新物化视图的默认方法 始终为'COMPLETE'
BUILD_MODE	指示物化视图在创建期间如何填充 始终为'IMMEDIATE'

### H. 56. 8. 36. [DBA/ALL/USER]\_MVIEW\_LOGS

- 不支持此视图，因此可以将其用作空视图以实现兼容性。
- DBA\_MVIEW\_LOGS描述了数据库中的所有物化视图日志。
- ALL\_MVIEW\_LOGS描述了当前用户可以访问的所有物化视图日志。
- USER\_MVIEW\_LOGS描述了当前用户拥有的所有物化视图日志。

### H. 56. 8. 37. [DBA/ALL/USER]\_MVIEW\_COMMENTS

- DBA\_MVIEW\_COMMENTS显示数据库中物化视图的注释。
- 目前ALL\_MVIEW\_COMMENTS视图与DBA\_MVIEW\_COMMENTS视图相同，且像Oracle那样没有进行权限验证。
- USER\_MVIEW\_COMMENTS显示当前用户拥有的物化视图的注释。此视图不显示OWNER列。

Table H. 171. ALL\_MVIEW\_COMMENTS

列名	描述
OWNER	物化视图的所有者
MVIEW_NAME	物化视图的名称
COMMENTS	物化视图的注释

### H. 56. 8. 38. [DBA/USER]\_TABLESPACES

- DBA\_TABLESPACES描述数据库中的所有表空间。
- USER\_TABLESPACES描述当前用户可以访问的表空间。该视图不显示PLUGGED\_IN列。

Table H. 172. DBA\_TABLESPACES

列	描述
TABLESPACE_NAME	表空间名称
BLOCK_SIZE	表空间块大小（以字节为单位）
STATUS	表空间状态 始终为'ONLINE'
CONTENTS	表空间内容

列	描述
	始终为 'PERMANENT'

### H. 56. 8. 39. NLS\_[DATABASE/INSTANCE/SESSION]\_PARAMETERS

- NLS\_DATABASE\_PARAMETERS 列出数据库的永久 NLS 参数。
- NLS\_INSTANCE\_PARAMETERS 列出实例的 NLS 参数。
- NLS\_SESSION\_PARAMETERS 列出用户会话的 NLS 参数。

Table H. 173. NLS\_DATABASE\_PARAMETERS

列	描述
PARAMETER	参数名称
VALUE	参数值

### H. 56. 8. 40. [DBA/USER]\_SEGMENTS

- DBA\_SEGMENTS 描述数据库中所有段分配的存储空间。
- USER\_SEGMENTS 描述当前用户对象所拥有的段分配的存储空间。该视图不会显示 OWNER、HEADER\_FILE、HEADER\_BLOCK 或 RELATIVE\_FNO 列。

Table H. 174. DBA\_SEGMENTS

列	描述
OWNER	段所有者的用户名
SEGMENT_NAME	段名称（如果有）
PARTITION_NAME	对象分区名称（非分区对象设为 NULL）
SEGMENT_TYPE	段类型：'TABLE'、'TABLE PARTITION'、'TABLE SUBPARTITION'、'INDEX'、'INDEX PARTITION'、'INDEX SUBPARTITION'
TABLESPACE_NAME	包含该段的表空间名称
HEADER_FILE	包含该段的表空间名称
BYTES	段的大小（以字节为单位）
BLOCKS	段的大小（以 LightDB 块为单位）

### H. 56. 8. 41. [DBA/USER/ALL]\_PART\_TABLES

- DBA\_PART\_TABLES 显示数据库中所有分区表的对象级分区信息。
- ALL\_PART\_TABLES 视图目前与 DBA\_PART\_TABLES 视图相同，并且不像 Oracle 那样进行权限验证。
- USER\_PART\_TABLES 显示当前用户拥有的分区表的对象级分区信息。该视图不会显示 OWNER 列。

支持的字段如下所示。不支持的字段将填入 NULL。

Table H. 175. ALL\_PART\_TABLES

列	描述
OWNER	分区表的所有者

列	描述
TABLE_NAME	分区表的名称
PARTITIONING_TYPE	分区方法的类型：RANGE、HASH、LIST
SUBPARTITIONING_TYPE	组合分区方法的类型：RANGE、HASH、LIST
PARTITION_COUNT	表中的分区数
PARTITIONING_KEY_COUNT	分区键中的列数
SUBPARTITIONING_KEY_COUNT	对于组合分区表，子分区键中的列数。
DEF_TABLESPACE_NAME	添加分区时要使用的默认表空间
DEF_LOGGING	添加分区时要使用的默认 LOGGING 属性

### H. 56. 8. 42. [DBA/USER/ALL]\_TAB\_PARTITIONS

- DBA\_TAB\_PARTITIONS 显示数据库中所有分区的分区级分区信息、分区存储参数和分区统计信息。
- 目前ALL\_TAB\_PARTITIONS视图与DBA\_TAB\_PARTITIONS视图相同，与Oracle不同的是没有进行权限验证。
- USER\_TAB\_PARTITIONS显示当前用户拥有的所有分区对象的分区信息，该视图不显示TABLE\_OWNER列。
- 通过ADD/CREATE/MODIFY创建分区时，分区名与传入的名字相同，Interval分区自动加前缀“ap”，哈希分区自动添加前缀“p”。

Table H. 176. ALL\_TAB\_PARTITIONS

列名	描述
TABLE_OWNER	表所有者
TABLE_NAME	表名
COMPOSITE	表示表是否为组合分区表（是：YES；否：NO）
PARTITION_NAME	分区名称
SUBPARTITION_COUNT	如果这是一个组合分区表，则为该分区中的子分区数。
PARTITION_POSITION	分区在表中的位置。
TABLESPACE_NAME	包含该分区的表空间名称
LOGGING	表示表格更改是否被记录
NUM_ROWS	分区中的行数
BLOCKS	分区中已使用数据块的数量
LAST_ANALYZED	此分区最近一次分析的日期

### H. 56. 8. 43. [DBA/USER/ALL]\_TAB\_SUBPARTITIONS

- DBA\_TAB\_SUBPARTITIONS显示数据库中所有子分区的子分区名称、所属表格和分区的名称、存储属性以及统计信息。
- ALL\_TAB\_SUBPARTITIONS视图目前与DBA\_TAB\_SUBPARTITIONS视图相同，并且与Oracle一样不进行权限验证。
- USER\_TAB\_SUBPARTITIONS显示当前用户拥有的所有分区对象的子分区信息。此视图不显示TABLE\_OWNER列。

- 添加子分区时，实际的子分区的名字与传入的分区名相同，创建表同时创建子分区时，子分区会自动添加分区名前缀，当创建一个未指定分区名的哈希分区时，分区名加前缀p。

Table H. 177. ALL\_TAB\_SUBPARTITIONS

列名	描述
TABLE_OWNER	表格所有者
TABLE_NAME	表格名称
PARTITION_NAME	分区名称
SUBPARTITION_NAME	子分区名称
PARTITION_POSITION	分区在表格中的位置
SUBPARTITION_POSITION	子分区在分区中的位置
TABLESPACE_NAME	包含该分区的表空间名称
LOGGING	表格更改是否被记录
NUM_ROWS	子分区中的行数
BLOCKS	子分区中的块数
LAST_ANALYZED	此表格最近一次分析的日期

#### H. 56. 8. 44. [DBA/USER/ALL]\_PART\_KEY\_COLUMNS

- DBA\_PART\_KEY\_COLUMNS描述了数据库中所有分区对象的分区键列。
- ALL\_PART\_KEY\_COLUMNS视图目前与DBA\_PART\_KEY\_COLUMNS视图相同，并且与Oracle一样不进行权限验证。
- USER\_PART\_KEY\_COLUMNS描述了当前用户拥有的分区对象的分区键列。此视图不显示OWNER列。

Table H. 178. ALL\_PART\_KEY\_COLUMNS

列名	描述
OWNER	分区表或索引的所有者
NAME	分区表或索引的名称
OBJECT_TYPE	对象类型：TABLE, INDEX
COLUMN_NAME	列的名称
COLUMN_POSITION	列在分区键中的位置

#### H. 56. 8. 45. [DBA/USER/ALL]\_SUBPART\_KEY\_COLUMNS

- DBA\_SUBPART\_KEY\_COLUMNS显示了数据库中所有子分区的复合分区表（以及复合分区表上的本地索引）的子分区键列。
- ALL\_SUBPART\_KEY\_COLUMNS视图目前与DBA\_SUBPART\_KEY\_COLUMNS视图相同，并且与Oracle一样不进行权限验证。
- USER\_SUBPART\_KEY\_COLUMNS显示了当前用户拥有的所有分区对象的子分区的此信息。此视图不显示OWNER列。

Table H. 179. ALL\_SUBPART\_KEY\_COLUMNS

列名	描述
OWNER	分区表或索引的所有者

列名	描述
NAME	分区表或索引的名称
OBJECT_TYPE	对象类型：TABLE, INDEX
COLUMN_NAME	列名
COLUMN_POSITION	列在子分区键中的位置

### H. 56. 8. 46. [DBA/USER/ALL]\_IND\_PARTITIONS

- DBA\_IND\_PARTITIONS描述数据库中的所有索引分区。
- ALL\_IND\_PARTITIONS视图目前与DBA\_IND\_PARTITIONS视图相同，且像Oracle一样不进行权限验证。
- USER\_IND\_PARTITIONS描述当前用户拥有的索引分区。此视图不显示INDEX\_OWNER列。

Table H. 180. ALL\_IND\_PARTITIONS

列名	描述
INDEX_OWNER	索引的所有者
INDEX_NAME	索引的名称
COMPOSITE	指示分区是否属于复合分区表上的本地索引（YES）或不属于（NO）
PARTITION_NAME	分区的名称
SUBPARTITION_COUNT	如果是复合分区表上的本地索引，则为该分区中子分区的数量
PARTITION_POSITION	分区在索引中的位置
TABLESPACE_NAME	包含该分区的表空间名称
LOGGING	指示是否将对索引的更改记录到日志中
NUM_ROWS	返回的行数
LAST_ANALYZED	此分区最近分析的日期

### H. 56. 8. 47. [DBA/USER/ALL]\_PART\_INDEXES

- DBA\_PART\_INDEXES显示数据库中所有分区索引的对象级分区信息。
- ALL\_PART\_INDEXES视图目前与DBA\_PART\_INDEXES视图相同，且像Oracle一样不进行权限验证。
- USER\_PART\_INDEXES显示当前用户拥有的分区索引的对象级分区信息。此视图不显示OWNER列。

Table H. 181. ALL\_PART\_INDEXES

列名	描述
OWNER	分区索引的所有者
INDEX_NAME	分区索引的名称
TABLE_NAME	分区表的名称
PARTITIONING_TYPE	分区方法的类型：RANGE, HASH, LIST
SUBPARTITIONING_TYPE	复合分区方法的类型：RANGE, HASH, LIST
PARTITION_COUNT	索引中的分区数

列名	描述
PARTITIONING_KEY_COUNT	分区键中的列数
SUBPARTITIONING_KEY_COUNT	对于复合分区表，子分区键中的列数
LOCALITY	指示分区索引是本地（LOCAL）还是全局（GLOBAL），始终为“LOCAL”
ALIGNMENT	指示分区索引是前缀（PREFIXED）还是非前缀（NON_PREFIXED）
DEF_TABLESPACE_NAME	对于本地索引，在添加或拆分表分区时要使用的默认表空间
DEF_LOGGING	对于本地索引，在添加表分区时要使用的默认LOGGING属性

## H. 56. 8. 48. COL

- COL 描述了当前用户拥有的表和视图的列。此视图不显示OWNER列。

Table H. 182. COL

Column	Description
TNAME	表、视图的名称
COLNO	列的创建顺序号
CNAME	列名
COLTYPE	列的数据类型
WIDTH	列的长度（字符类型单位为字符数；其他类型为字节数，对于number/numeric 类型，长度总是22）
SCALE	小数点右侧的数字位数
PRECISION	NUMBER数据类型的小数精度；FLOAT数据类型 的二进制精度；其他数据类型的值为NULL
NULLS	指示列是否允许为空。如果列有NOT NULL约 束或列是主键的一部分，则值为 'NOT NULL'
DEFAULTVAL	列的默认值

## H. 57. pageinspect

pageinspect模块提供函数让你从低层次观察数据库页面的内容，这对于调试目的很有用。所有这些函数只能被超级用户使用。

### H. 57. 1. 通用函数

get\_raw\_page(relname text, fork text, blkno int) 返回 bytea

get\_raw\_page读取提及的关系中的指定块并且以一个bytea值的形式返回一个拷贝。这允许得到该块的一个单一的时间一致的拷贝。对于主数据分叉，fork应该是'main'，对于空闲空间映射应该是'fsm'，对于可见性映射应该是'vm'，对于初始化分叉应该是'init'。

get\_raw\_page(relname text, blkno int) 返回 bytea

一个简写版的get\_raw\_page，用于读取主分叉。等效于get\_raw\_page(relname, 'main', blkno)

page\_header(page bytea) 返回 record

page\_header显示所有LightDB堆和索引页面的公共域。

用get\_raw\_page获得的一个页面映像应该作为参数传递。例如：

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
   lsn   | checksum | flags | lower | upper | special | pagesize | version
   | prune_xid
-----+-----+-----+-----+-----+-----+-----+-----
0/24A1B50 |         0 |      1 |   232 |   368 |      8192 |      8192 |         4
   |         0
```

返回的列对应于PageHeaderData结构中的域。详见src/include/storage/bufpage.h。

checksum域是存放在页面中的校验和，如果页面被损坏它可能是不正确的。如果对这个实例没有启用数据校验和，则存储的这个值没有意义。

page\_checksum(page bytea, blkno int4) returns smallint

page\_checksum为页面计算校验和，就像它被放置在给定块上一样。

应该将get\_raw\_page得到的页面映像作为参数传入。例如：

```
test=# SELECT page_checksum(get_raw_page('pg_class', 0), 0);
 page_checksum
-----
          13443
```

注意校验和取决于块号，因此应该将匹配的块号传入（除非在做调试）。

用这个函数计算的校验和可以拿来和函数page\_header的结果域checksum进行比较。如果为这个实例启用了数据校验和，则两个值应该相等。

fsm\_page\_contents(page bytea) returns text

fsm\_page\_contents显示FSM页面的内部节点结构。例如：

```
test=# SELECT fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
```

输出为多行字符串，页面中的二进制树中每个节点有一行。 仅有不为零的节点会被打印。所谓的“next”指针，指向页面中下一个要返回的槽，也会被打印。

FSM页面结构的更多信息可参见 src/backend/storage/freespace/README。

## H. 57.2. Heap Functions

heap\_page\_items(page bytea) 返回 setof record

heap\_page\_items显示一个堆页面上所有的行指针。 对那些使用中的行指针，元组头部和元组原始数据也会被显示。 不管元组对于拷贝原始页面时的 MVCC 快照是否可见，它们都会被显示。

用get\_raw\_page获得的一个堆页面映像应该作为参数传递。例如：

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

返回的域的解释可见src/include/storage/itemid.h和src/include/access/htup\_details.h。

heap\_tuple\_infomask\_flags函数可用于为堆元组解包t\_infomask 和t\_infomask2的标志位。

tuple\_data\_split(rel\_oid oid, t\_data bytea, t\_infomask integer, t\_infomask2 integer, t\_bits text [, do\_detoast bool]) returns bytea[]

tuple\_data\_split以后端内部的相同方式将元组数据拆解成属性。

```
test=# SELECT tuple_data_split('pg_class'::regclass, t_data, t_infomask,
    t_infomask2, t_bits) FROM heap_page_items(get_raw_page('pg_class', 0));
```

应该用与heap\_page\_items的返回属性相同的参数来调用这个函数。

如果do\_detoast是true, 则根据需要把属性解除TOAST。默认值为false。

heap\_page\_item\_attrs(page bytea, rel\_oid regclass [, do\_detoast bool]) returns setof record

heap\_page\_item\_attrs等效于 heap\_page\_items, 不过它会把元组原始数据 返回为属性的数组, 如果do\_detoast为真 ( 默认为false), 这些属性会被反 TOAST。

应该把用get\_raw\_page得到的一个堆页面映像 作为参数传入。例如:

```
test=# SELECT * FROM heap_page_item_attrs(get_raw_page('pg_class', 0),
    'pg_class'::regclass);
```

heap\_tuple\_infomask\_flags(t\_infomask integer, t\_infomask2 integer) returns record

heap\_tuple\_infomask\_flags将heap\_page\_items返回的 t\_infomask和t\_infomask2解码为一组由标志 名称组成的人类可读数组, 一列用于所有标志, 一列用于组合标志。 例如:

```
test=# SELECT t_ctid, raw_flags, combined_flags
    FROM heap_page_items(get_raw_page('pg_class', 0)),
    LATERAL heap_tuple_infomask_flags(t_infomask, t_infomask2)
    WHERE t_infomask IS NOT NULL OR t_infomask2 IS NOT NULL;
```

应该使用与heap\_page\_items的返回属性相同的参数调用此函数。

为考虑多个原始位值的源级宏显示组合标志, 例如HEAP\_XMIN\_FROZEN。

有关返回的标志名称的说明, 请参阅src/include/access/htup\_details.h。

### H. 57. 3. B树函数

bt\_metap(relname text) 返回 record

bt\_metap返回关于一个B树索引元页的信息。例如:

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-----+-----
magic          | 340322
version        | 4
```



```

root          | 1
level         | 0
fastroot     | 1
fastlevel    | 0
oldest_xact  | 582
last_cleanup_num_tuples | 1000
allequalimage | f

```

bt\_page\_stats(relname text, blkno int) 返回 record

bt\_page\_stats返回有关 B-树索引单一页面的总计信息。例如：

```

test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]+-----
blkno      | 1
type       | 1
live_items | 224
dead_items | 0
avg_item_size | 16
page_size  | 8192
free_size  | 3668
btpo_prev  | 0
btpo_next  | 0
btpo       | 0
btpo_flags | 3

```

bt\_page\_items(relname text, blkno int) 返回 setof record

bt\_page\_items返回一个 B-树索引页面上项的所有细节信息。例如：

```

test=# SELECT itemoffset, ctid, itemlen, nulls, vars, data, dead, htid,
tids[0:2] AS some_tids
FROM bt_page_items('tenk2_hundred', 5);
 itemoffset | ctid | itemlen | nulls | vars | data | dead | htid | some_tids |
-----+-----+-----+-----+-----+-----+-----+-----+-----
1 | (16,1) | 16 | f | f | 30 00 00 00 00 00 00 00 | | |
2 | (16,8292) | 616 | f | f | 24 00 00 00 00 00 00 00 | | |
f | (1,6) | {"(1,6)","(10,22)"} | | | | | |
3 | (16,8292) | 616 | f | f | 25 00 00 00 00 00 00 00 | | |
f | (1,18) | {"(1,18)","(4,22)"} | | | | | |
4 | (16,8292) | 616 | f | f | 26 00 00 00 00 00 00 00 | | |
f | (4,18) | {"(4,18)","(6,17)"} | | | | | |
5 | (16,8292) | 616 | f | f | 27 00 00 00 00 00 00 00 | | |
f | (1,2) | {"(1,2)","(1,19)"} | | | | | |
6 | (16,8292) | 616 | f | f | 28 00 00 00 00 00 00 00 | | |
f | (2,24) | {"(2,24)","(4,11)"} | | | | | |
7 | (16,8292) | 616 | f | f | 29 00 00 00 00 00 00 00 | | |
f | (2,17) | {"(2,17)","(11,2)"} | | | | | |
8 | (16,8292) | 616 | f | f | 2a 00 00 00 00 00 00 00 | | |
f | (0,25) | {"(0,25)","(3,20)"} | | | | | |
9 | (16,8292) | 616 | f | f | 2b 00 00 00 00 00 00 00 | | |
f | (0,10) | {"(0,10)","(0,14)"} | | | | | |
10 | (16,8292) | 616 | f | f | 2c 00 00 00 00 00 00 00 | | |
f | (1,3) | {"(1,3)","(3,9)"} | | | | | |

```

```

      11 | (16, 8292) |      616 | f      | f      | 2d 00 00 00 00 00 00 00 |
f      | (6, 28) | {"(6, 28)", "(11, 1)"}
      12 | (16, 8292) |      616 | f      | f      | 2e 00 00 00 00 00 00 00 |
f      | (0, 27) | {"(0, 27)", "(1, 13)"}
      13 | (16, 8292) |      616 | f      | f      | 2f 00 00 00 00 00 00 00 |
f      | (4, 17) | {"(4, 17)", "(4, 21)"}
(13 rows)

```

这是一个 B 树的叶子页面。指向该表的所有元组恰好是发布列表元组（所有这些元组总共存储 100 个 6 字节 TID）。在 itemoffset 编号 1 处还有一个“high key”元组。ctid 用于存储本示例中每个元组的编码信息，尽管叶页元组通常将堆 TID 直接存储在 ctid 字段中，而不是 tids 是存储为发布列表的 TID 列表。

在内部页面（未显示）中，ctid 的块号部分是一个“downlink”，它是索引本身中另一个页面的块号。ctid 的偏移部分（第二个数字）存储有关元组的编码信息，例如存在的列数（后缀截断可能已删除不需要的后缀列）。截断的列被视为具有值“minus infinity”。

htid 显示元组的堆 TID，无论底层元组表示如何。该值可能与 ctid 匹配，或者可以从发布列表元组和来自内部页面的元组所使用的替代表示中解码。内部页面中的元组通常会截断实现级别的堆 TID 列，这表示为 NULLhtid 值。

注意在任何非最右页面（页面的 btpo\_next 域中有非零值）上的第一个项是该页的“high key”，表示它的 data 是作为该页面上所有项的一个上界存在，而它的 ctid 字段不指向另一个块。此外，在内部页面上，第一个真正的数据项（第一个不是高键的项）可靠地将每一列都截断了，在其 data 字段中没有留下任何实际值。不过，这样一个项确实 在其有 ctid 域中有向下的链接。

有关 B 树索引结构的更多详细信息，请参阅 Section 60.4.1 有关重复数据删除和发布列表的更多详细信息，请参阅 Section 60.4.2

bt\_page\_items(page bytea) returns setof record

还可以把一个页面以 bytea 值的形式传递给 bt\_page\_items。应该将 get\_raw\_page 得到的页面映像作为参数传入。因此，上一个例子可以被重写成这样：

```

test=# SELECT itemoffset, ctid, itemlen, nulls, vars, data, dead, htid,
      tids[0:2] AS some_tids
      FROM bt_page_items(get_raw_page('tenk2_hundred', 5));
 itemoffset |   ctid   | itemlen | nulls | vars |          data          |
dead | htid |      some_tids
-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----
      1 | (16, 1) |      16 | f      | f      | 30 00 00 00 00 00 00 00 |
      |      |      |      |      |      |      |
      2 | (16, 8292) |      616 | f      | f      | 24 00 00 00 00 00 00 00 |
f      | (1, 6) | {"(1, 6)", "(10, 22)"}
      3 | (16, 8292) |      616 | f      | f      | 25 00 00 00 00 00 00 00 |
f      | (1, 18) | {"(1, 18)", "(4, 22)"}
      4 | (16, 8292) |      616 | f      | f      | 26 00 00 00 00 00 00 00 |
f      | (4, 18) | {"(4, 18)", "(6, 17)"}
      5 | (16, 8292) |      616 | f      | f      | 27 00 00 00 00 00 00 00 |
f      | (1, 2) | {"(1, 2)", "(1, 19)"}
      6 | (16, 8292) |      616 | f      | f      | 28 00 00 00 00 00 00 00 |
f      | (2, 24) | {"(2, 24)", "(4, 11)"}
      7 | (16, 8292) |      616 | f      | f      | 29 00 00 00 00 00 00 00 |
f      | (2, 17) | {"(2, 17)", "(11, 2)"}
      8 | (16, 8292) |      616 | f      | f      | 2a 00 00 00 00 00 00 00 |
f      | (0, 25) | {"(0, 25)", "(3, 20)"}

```

```

          9 | (16, 8292) |      616 | f      | f      | 2b 00 00 00 00 00 00 00 |
f      | (0, 10) | {"(0, 10)", "(0, 14)"}
          10 | (16, 8292) |      616 | f      | f      | 2c 00 00 00 00 00 00 00 |
f      | (1, 3)  | {"(1, 3)", "(3, 9)"}
          11 | (16, 8292) |      616 | f      | f      | 2d 00 00 00 00 00 00 00 |
f      | (6, 28) | {"(6, 28)", "(11, 1)"}
          12 | (16, 8292) |      616 | f      | f      | 2e 00 00 00 00 00 00 00 |
f      | (0, 27) | {"(0, 27)", "(1, 13)"}
          13 | (16, 8292) |      616 | f      | f      | 2f 00 00 00 00 00 00 00 |
f      | (4, 17) | {"(4, 17)", "(4, 21)"}
(13 rows)

```

所有其他细节和前一项中的解释相同。

## H. 57. 4. GIN函数

`gin_metapage_info(page bytea)` returns record

`gin_metapage_info`返回有关一个 GIN索引元页的信息。例如：

```

test=# SELECT * FROM gin_metapage_info(get_raw_page('gin_index', 0));
-[ RECORD 1 ]-----+-----
pending_head      | 4294967295
pending_tail      | 4294967295
tail_free_size    | 0
n_pending_pages   | 0
n_pending_tuples  | 0
n_total_pages     | 7
n_entry_pages     | 6
n_data_pages      | 0
n_entries         | 693
version           | 2

```

`gin_page_opaque_info(page bytea)` returns record

`gin_page_opaque_info`返回有关一个 GIN索引不透明区域的信息，如页面类型等。例如：

```

test=# SELECT * FROM gin_page_opaque_info(get_raw_page('gin_index', 2));
 rightlink | maxoff | flags
-----+-----+-----
          5 |        0 | {data, leaf, compressed}
(1 row)

```

`gin_leafpage_items(page bytea)` returns setof record

`gin_leafpage_items`返回有关存储在一个 GIN叶子页面中的数据的的信息。例如：

```

test=# SELECT first_tid, nbytes, tids[0:5] AS some_tids
       FROM gin_leafpage_items(get_raw_page('gin_test_idx', 2));
 first_tid | nbytes | some_tids
-----+-----+-----
(8, 41)   |    244 | {"(8, 41)", "(8, 43)", "(8, 44)", "(8, 45)", "(8, 46)"}
(10, 45)  |    248 | {"(10, 45)", "(10, 46)", "(10, 47)", "(10, 48)", "(10, 49)"}
(12, 52)  |    248 | {"(12, 52)", "(12, 53)", "(12, 54)", "(12, 55)", "(12, 56)"}

```

```
(14, 59) | 320 | {"(14, 59)", "(14, 60)", "(14, 61)", "(14, 62)", "(14, 63)"}
(167, 16) | 376 |
{"(167, 16)", "(167, 17)", "(167, 18)", "(167, 19)", "(167, 20)"}
(170, 30) | 376 |
{"(170, 30)", "(170, 31)", "(170, 32)", "(170, 33)", "(170, 34)"}
(173, 44) | 197 |
{"(173, 44)", "(173, 45)", "(173, 46)", "(173, 47)", "(173, 48)"}
(7 rows)
```

## H. 57. 5. Hash函数

hash\_page\_type(page bytea) returns text

hash\_page\_type返回给定的HASH索引页面的页面类型。例如：

```
test=# SELECT hash_page_type(get_raw_page('con_hash_index', 0));
 hash_page_type
-----
metapage
```

hash\_page\_stats(page bytea) returns setof record

hash\_page\_stats返回有关一个HASH索引的桶页或者溢出页的信息。例如：

```
test=# SELECT * FROM hash_page_stats(get_raw_page('con_hash_index', 1));
-[ RECORD 1 ]-----+-----
live_items      | 407
dead_items      | 0
page_size       | 8192
free_size       | 8
hasho_prevblkno | 4096
hasho_nextblkno | 8474
hasho_bucket     | 0
hasho_flag       | 66
hasho_page_id    | 65408
```

hash\_page\_items(page bytea) returns setof record

hash\_page\_items返回有关一个HASH索引的桶页或者溢出页中存储的数据的信息。例如：

```
test=# SELECT * FROM hash_page_items(get_raw_page('con_hash_index', 1)) LIMIT
5;
 itemoffset | ctid | data
-----+-----+-----
          1 | (899, 77) | 1053474816
          2 | (897, 29) | 1053474816
          3 | (894, 207) | 1053474816
          4 | (892, 159) | 1053474816
          5 | (890, 111) | 1053474816
```

hash\_bitmap\_info(index oid, blkno int) returns record

hash\_bitmap\_info返回HASH索引一个特定溢出页在位图页中的位的状态。例如：

```
test=# SELECT * FROM hash_bitmap_info('con_hash_index', 2052);
 bitmapblkno | bitmapbit | bitstatus
```

```
-----+-----+-----
        65 |          3 | t
```

hash\_metapage\_info(page bytea) returns record

hash\_metapage\_info返回一个HASH索引的元页中存放的信息。例如：

```
test=# SELECT magic, version, ntuples, ffactor, bsize, bmsize, bmshift,
test-#      maxbucket, highmask, lowmask, ovflpoint, firstfree, nmaps, procid,
test-#      regexp_replace(spares::text, '(,0)*', ',}') as spares,
test-#      regexp_replace(mapp::text, '(,0)*', ',}') as mapp
test=# FROM hash_metapage_info(get_raw_page('con_hash_index', 0));
-[ RECORD 1 ]-----
magic      | 105121344
version    | 4
ntuples    | 500500
ffactor    | 40
bsize      | 8152
bmsize     | 4096
bmshift    | 15
maxbucket  | 12512
highmask   | 16383
lowmask    | 8191
ovflpoint  | 28
firstfree  | 1204
nmaps      | 1
procid     | 450
spares     | {0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 3, 4, 4, 4, 45, 55, 58, 59,
508, 567, 628, 704, 1193, 1202, 1204}
mapp       | {65}
```

## H. 58. passwordcheck

只要通过CREATE ROLE或ALTER ROLE设置用户， passwordcheck模块会检查用户的口令。如果一个口令被认为太弱，它将被拒绝并且该命令将带着一个错误终止。

要启用这个模块，把 '\$libdir/passwordcheck' 加入到lightdb.conf中的shared\_preload\_libraries，然后重启服务器。

你可以通过修改源代码来按你的需要修改这个模块。例如，你可以使用CrackLib<sup>14</sup>来检查口令 — 这只需要在Makefile中取消两行的注释并且重新编译该模块（由于授权原因，我们不能默认包括CrackLib）。如果没有CrackLib，该模块会对口令强度强制一些简单的规则，你可以自行修改和扩充。

### Caution

要阻止未加密的口令通过网络传送、写入到服务器日志或者被一个数据库管理员窃取，LightDB允许用户提供预加密的口令。很多客户端程序利用这种功能并且在把口令发送给服务器之前加密它。

这限制了passwordcheck模块的有用性，因为这种情况下它只能尝试猜测口令。由于这个原因，如果你的安全性需求很高，我们不推荐passwordcheck。使用一个诸如 GSSAPI（见Chapter 18 的外部认证方法比依赖数据库内的口令更加安全。

<sup>14</sup> <https://github.com/cracklib/cracklib>

此外，你可以修改passwordcheck来拒绝预加密的口令，但是强制用户将口令设置为明文带来了它的安全风险。

## H. 59. postgres\_fdw

postgres\_fdw模块提供了外部数据包装器postgres\_fdw，它可以被用来访问存储在外部LightDB服务器中的数据。

这个模块提供的功能大体上覆盖了较老的dblink模块的功能。但是postgres\_fdw提供了更透明且更兼容标准的语法来访问远程表，并且可以在很多情况下给出更好的性能。

要使用postgres\_fdw来为远程访问做准备：

1. 使用CREATE EXTENSION来安装postgres\_fdw扩展。
2. 使用CREATE SERVER创建一个外部服务器对象，它用来表示你想连接的每一个远程数据库。指定除了user和password之外的连接信息作为该服务器对象的选项。
3. 使用CREATE USER MAPPING创建一个用户映射，每一个用户映射都代表你想允许一个数据库用户访问一个外部服务器。指定远程用户名和口令作为用户映射的user和password选项。
4. 为每一个你想访问的远程表使用CREATE FOREIGN TABLE或者IMPORT FOREIGN SCHEMA创建一个外部表。外部表的列必须匹配被引用的远程表。但是，如果你在外部表对象的选项中指定了正确的远程名称，你可以使用不同于远程表的表名和/或列名。

现在你只需要从一个外部表SELECT来访问存储在它的底层的远程表中的数据。你也可以使用INSERT、UPDATE或DELETE修改远程表（当然，你在你的用户映射中已经指定的远程用户必须具有做这些事情的权利）。

注意当前postgres\_fdw缺少对于带ON CONFLICT DO UPDATE子句的INSERT语句的支持。不过，它支持ON CONFLICT DO NOTHING子句，已提供的唯一索引推断说明会被省略。另请注意postgres\_fdw支持UPDATE语句在分区表上执行的行移动调用，但是它目前不处理这样的情况：选择将移动行插入到的远程分区也是稍后将更新的UPDATE目标分区。

我们通常推荐一个外部表的列被声明为与被引用的远程表列完全相同的数据类型和排序规则（如果可用）。尽管postgres\_fdw目前已经能够容忍在需要时执行数据类型转换，但是当类型或排序规则不匹配时可能会发生奇怪的语义异常，因为远程服务器解释WHERE子句时可能会与本地服务器有所不同。

注意一个外部表可以被声明比底层的远程表较少的列，或者使用一种不同的列序。与远程表的列匹配是通过名字而不是位置进行的。

### H. 59. 1. postgres\_fdw 的 FDW 选项

#### H. 59. 1. 1. 连接选项

一个使用postgres\_fdw外部数据包装器的外部服务器可以使用和libpq在连接字符串中能接受的选项，如Section 32. 1. 所述，除了这些选项不被允许或有特殊处理：

- user、password和sslpassword（应该在用户映射中指定这些，或者使用服务文件）
- client\_encoding（这是自动从本地服务器编码设置）
- fallback\_application\_name（总是设置为postgres\_fdw）
- sslkey和sslcert - 这些可能出现在either or both连接和用户映射中。如果两者都存在，则用户映射设置会覆盖连接设置。

只有超级用户可以使用sslcert或sslkey设置创建或修改用户映射。

只有超级用户可以在不经过口令认证的情况下连接到外部服务器，因此应总是为属于非超级用户的用户映射指定password选项。

超级用户可以通过设置用户映射选项 `password_required 'false'` 在每个用户映射的基础上覆盖此检查，例如，

```
ALTER USER MAPPING FOR some_non_superuser SERVER loopback_nopw
OPTIONS (ADD password_required 'false');
```

为了防止非特权用户利用正在运行的 `lightdb` 服务器升级到超级用户权限的 `unix` 用户的身份验证权限，只有超级用户可以在用户映射上设置此选项。

需要注意确保这不允许映射用户能够根据 `CVE-2007-3278` 和 `CVE-2007-6601` 作为超级用户连接到映射数据库。不要在 `public` 角色上设置 `password_required=false`。请记住，映射的用户可能会使用 `lightdb` 服务器运行的系统用户的 `unix` 主目录中的任何客户端证书，`.pgpass`、`.pg_service.conf`等。他们还可以使用由诸如 `peer` 或 `ident` 身份验证等身份验证模式授予的任何信任关系。

## H. 59. 1. 2. 对象名称选项

这些选项可以被用来控制使用在被发送到远程 `LightDB` 服务器的 `SQL` 语句中使用的名称。当一个外部表被使用不同于底层远程表的名称创建时，就需要这些选项。

`schema_name`

这个选项给出用在远程服务器之上的外部表的模式名称，它可以为一个外部表指定。如果这个选项被忽略，该外部表的模式名称将被使用。

`table_name`

这个选项给出用在远程服务器上的外部表给出表名，它可以为一个外部表指定。如果这个选项被忽略，该外部表的名字将被使用。

`column_name`

这个选项给出用在远程服务器上列的列名，它可以为一个外部表的一个列指定。如果这个选项被忽略，该列的名字将被使用。

## H. 59. 1. 3. 代价估计选项

`postgres_fdw` 通过在远程服务器上执行查询来检索远程数据，因此理想的扫描一个外部表的估计代价应该是在远程服务器上完成它的花销，外加一些通信开销。得到这样一个估计的最可靠的方法是询问远程服务器并加上一些通信开销 — 但是对于简单查询，不值得为获得一个代价估计而额外使用一次远程查询。因此 `postgres_fdw` 提供了下列选项来控制如何完成代价估计：

`use_remote_estimate`

这个选项控制 `postgres_fdw` 是否发出 `EXPLAIN` 命令来获得代价估计，它可以为一个外部表或一个外部服务器指定。一个外部表的设置会覆盖它的服务器的任何设置，但是只用于这个表。默认值是 `false`。

`fdw_startup_cost`

这个选项是一个要被加到那个服务器上所有外部表扫描的估计启动代价的数字值。这表示建立一个连接、在远端解析和规查询的额外负荷等。默认值是 `100`。

`fdw_tuple_cost`

这个选项是一个数字值，它被用作那个服务器上外部表扫描的每元组额外代价，它可以为一个外部服务器指定。这表示在服务器之间数据传输的额外负荷。你可以增加或减少这个数来反映到远程服务器更高或更低的网络延迟。默认值是 `0.01`。

当`use_remote_estimate`为真时，`postgres_fdw`从远程服务器获得行计数和代价估计，然后在代价估计上加上`fdw_startup_cost`和`fdw_tuple_cost`。当`use_remote_estimate`为假时，`postgres_fdw`执行本地行计数和代价估计，并且接着在代价估计上加上`fdw_startup_cost`和`fdw_tuple_cost`。这种本地估计不会很准确，除非有远程表统计数据的本地拷贝可用。在外部表上运行ANALYZE是更新本地统计数据的方法，这将执行远程表的一次扫描并接着计算和存储统计数据，就好像表在本地一样。保留本地统计数据可能是一种有用的方法来减少一个远程表的预查询规划负荷 — 但是如果远程表被频繁更新，本地统计数据将很快就被废弃。

#### H. 59. 1. 4. 远程执行选项

默认情况下，只有使用了内建操作符和函数的WHERE子句才会被考虑在远程服务器上执行。涉及非内建函数的子句将会在取完行后在本地进行检查。如果这类函数在远程服务器上可用并且可以用来产生和本地执行时一样的结果，则可以通过将这种WHERE子句发送到远程执行来提高性能。可以用下面的选项控制这种行为：

`extensions`

这个选项是一个用逗号分隔的已安装的LightDB扩展名称列表，这些扩展在本地和远程服务器上具有兼容的版本。属于一个该列表中扩展的`immutable`函数和操作符将被考虑转移到远程服务器上执行。这个选项只能为外部服务器指定，无法逐个表指定。

在使用`extensions`选项时，用户应该负责确保列出的扩展在本地和远程服务器上都存在且保持一致。否则，远程查询可能失败或者行为异常。

`fetch_size`

这个选项指定在每次获取行的操作中`postgres_fdw`应该得到的行数。可以为一个外部表或者外部服务器指定这个选项。在表上指定的选项将会覆盖在服务器级别上指定的选项。默认值为100。

#### H. 59. 1. 5. 可更新性选项

默认情况下，所有使用`postgres_fdw`的外部表都被假定是可更新的。这可以使用下列选项覆盖：

`updatable`

这个选项控制`postgres_fdw`是否允许外部表被使用INSERT、UPDATE和DELETE命令更新。它可以为一个外部表或一个外部服务器指定。一个表级选项会覆盖一个服务器级选项。默认值是`true`。

当然，如果远程表实际上并非可更新的，将产生一个错误。这个选项的使用主要是允许在不查询远程服务器的情况下在本地抛出错误。但是要注意`information_schema`视图会根据这个选项的设置报告一个`postgres_fdw`外部表是可更新的（或者不可更新），而不需要远程服务器的任何检查。

#### H. 59. 1. 6. 导入选项

`postgres_fdw`能使用IMPORT FOREIGN SCHEMA导入外部表定义。这个命令会在本地服务器上创建外部表定义，这个定义能匹配存在于远程服务器上的表或者视图。如果要被导入的远程表有用户自定义数据类型的列，本地服务器上也必须具有相同名称的兼容类型。

导入行为可以用下列选项自定义（在IMPORT FOREIGN SCHEMA命令中给出）：

`import_collate`

这个选项控制是否在从外部服务器导入的外部表定义中包括列的COLLATE选项。默认是`true`。如果远程服务器具有和本地服务器不同的排序规则名集合，可能需要关闭这个选项，在不同的操作系统上运行时很可能就是这样。



即使将此参数设置为true，导入远程服务器默认排序规则的列也可能存在风险。它们将被导入并使用COLLATE "default"，这将选择本地服务器的默认排序规则，而这可能是不同的。

#### import\_default

这个选项控制是否在从外部服务器导入的外部表定义中包括列的DEFAULT表达式。默认是false。如果启用这个选项，要当心在远程服务器和本地服务器上计算表达式的方式不同，nextval()常会导致这类问题。如果导入的默认值表达式使用了一个本地不存在的函数或者操作符，IMPORT将整个失败。

#### import\_not\_null

这个选项控制是否在从外部服务器导入的外部表定义中包括列的NOT NULL约束。默认是true。

注意除NOT NULL之外的约束将不会从远程表中导入。虽然LightDB确实支持外部表上的CHECK约束，但不会自动导入它们，因为存在本地和远程服务器计算约束表达式方式不同的风险。CHECK约束中的任何这类不一致都可能导致查询优化中很难检测的错误。因此，如果你希望导入CHECK约束，你必须手工来做，并且你应该仔细地验证每一个这种约束的语义。有关处理外部表上CHECK约束的更多细节，请见CREATE FOREIGN TABLE。

自动排除作为其他表的分区的表或外部表。分区表被导入，除非它们不是其他表的分区。由于所有数据都可以通过作为分区层次根的分区的表来访问，所以这种方法应该允许访问所有数据而不创建额外的对象。

## H. 59. 2. 连接管理

postgres\_fdw在第一个使用关联到外部服务器的外部表的查询期间建立一个到外部服务器的连接。这个连接会被保持，并被重用于同一个会话中的后续查询。但是，如果使用了多个用户实体（用户映射）来访问外部服务器，会为每一个用户映射建立一个连接。

## H. 59. 3. 事务管理

在一个引用外部服务器上任何远程表的查询期间，如果还没有根据当前的本地事务打开一个远程事务，postgres\_fdw将在远程服务器上打开一个事务。当本地事务提交或中止时，远程事务也被提交或中止。保存点也相似地采用创建相应的远程保存点来管理。

当本地事务为SERIALIZABLE隔离级别时，远程事务使用SERIALIZABLE隔离级别；否则它使用REPEATABLE READ隔离级别。如果一个查询在远程服务器上执行多个表查询，这种选择保证它将为所有扫描得到快照一致的结果。一种后果是在单一事务中的后继查询将会看到来自远程服务器的相同数据，即便由于其他活动在远程服务器上发生了其他并发更新。如果本地事务使用SERIALIZABLE或REPEATABLE READ隔离级别，这种行为也是可以预期的，但是对于一个READ COMMITTED本地事务它是奇怪的。一个未来的LightDB发布可能会修改这些规则。

请注意postgres\_fdw当前不支持为两阶段提交准备远程事务。

## H. 59. 4. 远程查询优化

postgres\_fdw尝试优化远程查询来减少从外部服务器传来的数据量。这可以通过把查询的WHERE子句发送给远程服务器执行来完成，并且还可以不检索当前查询不需要的表列。为了降低查询被误执行的风险，除非WHERE子句使用的数据类型、操作符和函数都是内建的或者属于列在该外部服务器的extensions选项中的一个扩展，将不会把WHERE子句发送到远程服务器。这些子句中的操作符合函数也必须是IMMUTABLE。对于UPDATE或者DELETE查询，如果没有不能发送给远程服务器的WHERE子句、没有查询的本地连接、目标表上没有本地的行级BEFORE或AFTER触发器或存储生成的列，并且没有来自父视图的CHECK OPTION约束，postgres\_fdw会尝试通过将整个查询发送给远程服务器来优化查询的执行。在UPDATE中，赋值给目标列的表达式只能使用内建数据类型、IMMUTABLE操作符或者IMMUTABLE操作符，这样能降低查询被误执行的风险。

当`postgres_fdw`碰到同一个外部服务器上的外部表之间的连接时，它会把整个连接发送给外部服务器，除非由于某些原因它认为逐个从每一个表取得行的效率更高或者涉及的表引用属于不同的用户映射。在发送JOIN子句时，它也会采取和上述WHERE子句相同的预防措施。

实际被发送到远程服务器执行的查询可以使用`EXPLAIN VERBOSE`来检查。

## H. 59. 5. 远程查询执行环境

在`postgres_fdw`开启的远程会话中，`search_path`参数只被设置为`pg_catalog`，因此只有内建对象可以在无模式限定时可见。这对于`postgres_fdw`本身产生的查询来说不是问题，因为它总是会提供这样的限定。不过，这可能会对在远程服务器上通过触发器或者远程表上的规则执行的函数带来灾难。例如，如果一个远程表实际是一个视图，任何在该视图中使用的函数都将被在这个受限的搜索路径中执行。我们推荐在这类函数中用模式限定所有名称，或者为这类函数附着`SET search_path`选项（见`CREATE FUNCTION`）来建立它们所期望的搜索路径环境。

`postgres_fdw`同样为各种参数建立远程会话设置：

- `TimeZone`设置为UTC
- `DateStyle`设置为ISO
- `IntervalStyle`设置为`postgres`
- 对于远程服务器9.0和更新版本，`extra_float_digits` 设置为3，并且针对更老版本设置为2

这些不如`search_path`有那么多问题，但是如果需要也可以使用函数 `SET`选项来处理。

我们不推荐通过更改这些参数的会话级设置来推翻这种行为，这很可能导致`postgres_fdw`故障。

## H. 59. 6. 跨版本兼容性

`postgres_fdw`能够与最老的是LightDB 8.3 的远程服务器一起使用。只读能力则最低可以在8.1 中使用。但是一个限制是`postgres_fdw`通常假定不变的内建函数和操作符是安全的，如果它们出现在一个外部表的WHERE子句中，它们可以发送给远程服务器执行。因此，由于一个由于远程服务器的发布可能被发送给它来执行而被增加的内建函数，会导致“function does not exist” 或一个类似的错误。这类错误可以通过重写查询来解决，例如通过嵌入在一个带`OFFSET 0`的子SELECT中引用的外部表作为一种优化墙，并且把出问题的函数或操作符放在子SELECT的外部。

## H. 59. 7. 例子

这里是一个用`postgres_fdw`创建外部表的例子。首先安装该扩展：

```
CREATE EXTENSION postgres_fdw;
```

然后使用`CREATE SERVER`创建一个外部服务器。在这个例子中我们希望连接到一个位于主机192.83.123.89上并且监听5432端口的LightDB服务器。在该远程服务器上要连接的数据库名为`foreign_db`：

```
CREATE SERVER foreign_server
    FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');
```

需要用`CREATE USER MAPPING`定义一个用户映射来标识在远程服务器上使用哪个角色：

```
CREATE USER MAPPING FOR local_user
    SERVER foreign_server
    OPTIONS (user 'foreign_user', password 'password');
```

现在就可以使用CREATE FOREIGN TABLE创建外部表了。在这个例子中我们希望访问远程服务器上名为some\_schema.some\_table的表。它的本地名称是foreign\_table:

```
CREATE FOREIGN TABLE foreign_table (
    id integer NOT NULL,
    data text
)
    SERVER foreign_server
    OPTIONS (schema_name 'some_schema', table_name 'some_table');
```

CREATE FOREIGN TABLE中声明的列数据类型和其他性质必须要匹配实际的远程表。列名也必须匹配, 不过也可以为个别列附上column\_name选项以表示它们在远程服务器上对应哪个列。在很多情况中, 要手工构造外部表定义, 使用IMPORT FOREIGN SCHEMA会更好。

## H. 59. 8. 作者

Shigeru Hanada <shigeru.hanada@gmail.com>

## H. 60. pldebugger

LightDB pl/pgsql Debugger API。该模块是一组共享库, 用于实现在 lightDB 13.8-23.2 及以上版本上调试 pl/pgsql 函数的API。

### H. 60. 1. 安装

编辑 lightdb.conf 文件, 修改 shared\_preload\_libraries 配置选项如下:

```
shared_preload_libraries = "$libdir/plugin_debugger"
```

- plugin\_debugger 必须是最后一个扩展名, 否则调试可能无效。
- 新启动 LightDB 以使新设置生效。
- 在希望调试函数的数据库中运行以下命令: CREATE EXTENSION pldbgapi;

### H. 60. 2. 用法

连接 navicat 到包含要调试函数的数据库。右键单击要调试的函数, 选择 Debugging->Debug 以立即执行和调试函数, 或选择 Debugging->Set Global Breakpoint 来在函数上设置断点。这将导致调试器等待另一个会话 (例如为 Web 应用程序提供服务的后端) 来执行函数, 并允许在上下文中进行调试。有关更多信息, 请参阅 navica 文档。

### H. 60. 3. 架构

调试器由三部分组成:

1. 客户端。通常是 GUI, 显示源代码、当前堆栈帧、变量等, 并允许用户设置断点和逐步调试代码。客户端可以驻留在与数据库服务器不同的主机上。
2. 目标后端。这是运行被调试代码的后端。必须将 plugin\_debugger.so 库加载到目标后端中。
3. 调试代理。这是与客户端连接的另一个后端进程。在此后端中运行 pldbgapi.so 库中的 pldbg\_\* API 函数。客户端使用常规的 libpq 连接连接到调试代理。当调试会话处于活动状态时, 代理通过套接字连接到目标。代理与目标后端之间的协议对其他人不可见, 并且可能会发生更改。pldbg\_\* API 函数形成了调试设施的公共接口。

```

调试端 *----- libpq -----* 代理端
(navicat)
|
pldebugger socket 连接
|
*
应用端 *----- libpq -----* 目标端
    
```

## H. 61. rum

### H. 61. 1. 介绍

rum 模块提供了使用 RUM 索引的方法。它基于 GIN 访问方法代码实现。

GIN 索引允许使用 `tsvector` 和 `tsquery` 类型进行快速全文搜索。但是，使用 GIN 索引进行全文搜索存在几个问题：

- 排序慢。需要词元的位置信息才能进行排序。GIN 索引不存储词元的位置。因此，在索引扫描之后，我们需要进行额外的堆扫描来检索词元的位置。
- 使用 GIN 索引进行短语搜索较慢。这个问题与上一个问题有关。执行短语搜索需要位置信息。
- 按时间戳排序慢。GIN 索引无法在词元中存储某些相关信息。因此需要执行额外的堆扫描。

RUM 通过在 `posting` 树中存储附加信息来解决这些问题。例如，词元的位置信息或时间戳。

RUM 的缺点是它的构建和插入时间比 GIN 慢。这是因为我们需要存储键之外的附加信息，并且 RUM 使用通用 WAL 记录。

### H. 61. 2. 常用操作符和函数

rum 模块提供了以下操作符。

操作符	返回值	描述
<code>tsvector &lt;=&gt; tsquery</code>	float4	返回 <code>tsvector</code> 和 <code>tsquery</code> 之间的距离。
<code>timestamp &lt;=&gt; timestamp</code>	float8	返回两个时间戳之间的距离。
<code>timestamp &lt;=  timestamp</code>	float8	仅返回左侧时间戳的距离。
<code>timestamp  =&gt; timestamp</code>	float8	仅返回右侧时间戳的距离。

最后三个操作也适用于以下类型：

`timestamptz`、`int2`、`int4`、`int8`、`float4`、`float8`、`money` 和 `oid`。

### H. 61. 3. 操作符类

rum 提供了以下操作符类。

#### H. 61. 3. 1. rum\_tsvector\_ops

适用类型：`tsvector`

该操作符类存储了带有位置信息的 `tsvector` 词元。支持使用 `<=>` 操作符进行排序和前缀搜索。下面是一个示例。

假设我们有以下表:

```
CREATE TABLE test_run(t text, a tsvector);

CREATE TRIGGER tsvectorupdate
BEFORE UPDATE OR INSERT ON test_run
FOR EACH ROW EXECUTE PROCEDURE tsvector_update_trigger('a',
'pg_catalog.english', 't');

INSERT INTO test_run(t) VALUES ('The situation is most beautiful');
INSERT INTO test_run(t) VALUES ('It is a beautiful');
INSERT INTO test_run(t) VALUES ('It looks like a beautiful place');
```

要创建 rum 索引, 我们需要创建一个扩展:

```
CREATE EXTENSION rum;
```

然后我们可以创建新的索引:

```
CREATE INDEX rumidx ON test_run USING rum (a rum_tsvector_ops);
```

我们可以执行以下查询:

```
SELECT t, a <=> to_tsquery('english', 'beautiful | place') AS rank
FROM test_run
WHERE a @@ to_tsquery('english', 'beautiful | place')
ORDER BY a <=> to_tsquery('english', 'beautiful | place');
      t | rank
-----+-----
It looks like a beautiful place | 8.22467
The situation is most beautiful | 16.4493
It is a beautiful | 16.4493
(3 rows)
```

```
SELECT t, a <=> to_tsquery('english', 'place | situation') AS rank
FROM test_run
WHERE a @@ to_tsquery('english', 'place | situation')
ORDER BY a <=> to_tsquery('english', 'place | situation');
      t | rank
-----+-----
The situation is most beautiful | 16.4493
It looks like a beautiful place | 16.4493
(2 rows)
```

### H. 61. 3. 2. rum\_tsvector\_hash\_ops

适用类型: tsvector

该操作符类存储带有位置信息的 tsvector 词元的哈希值。支持使用 <=> 操作符进行排序。但是, 它不支持前缀搜索。

### H. 61. 3. 3. rum\_TYPE\_ops

适用类型:

int2、int4、int8、float4、float8、money、oid、time、timetz、date、interval、macaddr、inet、cidr

支持的操作：对于所有类型，支持 `<`、`<=`、`=`、`>=`、`>`，对于 `int2`、`int4`、`int8`、`float4`、`float8`、`money`、`oid`、`timestamp` 和 `timestampz` 类型，还支持 `<=>`、`<=|` 和 `|=>`。

支持使用 `<=>`、`<=|` 和 `|=>` 操作符进行排序。可以与 `rum_tsvector_addon_ops`、`rum_tsvector_hash_addon_ops` 和 `rum_anyarray_addon_ops` 操作符类一起使用。

### H. 61. 3. 4. rum\_tsvector\_addon\_ops

适用类型：tsvector

该操作符类可以使用模块支持的任何字段存储带有位置信息的 tsvector 词元。下面是一个示例。

假设我们有以下表：

```
CREATE TABLE tstst (id int, t tsvector, d timestamp);

\copy tstst from 'rum/data/tstst.data'

CREATE INDEX tstst_idx ON tstst USING rum (t rum_tsvector_addon_ops, d)
    WITH (attach = 'd', to = 't');
```

现在我们可以执行以下查询：

```
EXPLAIN (costs off)
    SELECT id, d, d <=> '2016-05-16 14:21:25' FROM tstst WHERE t @@ 'wr&qh' ORDER
    BY d <=> '2016-05-16 14:21:25' LIMIT 5;
                                QUERY PLAN
```

```
-----
Limit
->  Index Scan using tstst_idx on tstst
      Index Cond: (t @@ '''wr''' & '''qh'''::tsquery)
      Order By: (d <=> 'Mon May 16 14:21:25 2016'::timestamp without time
zone)
(4 rows)
```

```
SELECT id, d, d <=> '2016-05-16 14:21:25' FROM tstst WHERE t @@ 'wr&qh' ORDER BY
d <=> '2016-05-16 14:21:25' LIMIT 5;
```

id	d	?column?
355	Mon May 16 14:21:22.326724 2016	2.673276
354	Mon May 16 13:21:22.326724 2016	3602.673276
371	Tue May 17 06:21:22.326724 2016	57597.326724
406	Wed May 18 17:21:22.326724 2016	183597.326724
415	Thu May 19 02:21:22.326724 2016	215997.326724

(5 rows)

警告： 当使用传递引用附加信息的排序创建索引时，目前的 RUM 存在错误行为。这是因为 posting 树具有固定长度的右边界和固定长度的非叶 posting 项。不允许创建这样的索引。

### H. 61. 3. 5. rum\_tsvector\_hash\_addon\_ops

适用类型：tsvector

该操作符类可以使用模块支持的任何字段存储带有位置信息的 tsvector 词元的哈希值。

不支持前缀搜索。

### H. 61. 3. 6. rum\_tsquery\_ops

适用类型: tsquery

将查询树的分支存储在附加信息中。例如，我们有以下表：

```
CREATE TABLE query (q tsquery, tag text);

INSERT INTO query VALUES ('supernova & star', 'sn'),
    ('black', 'color'),
    ('big & bang & black & hole', 'bang'),
    ('spiral & galaxy', 'shape'),
    ('black & hole', 'color');
```

```
CREATE INDEX query_idx ON query USING rum(q);
```

现在我们可以执行以下快速查询：

```
SELECT * FROM query
    WHERE to_tsvector('black holes never exists before we think about them') @@
    q;
```

q	tag
'black'	color
'black' & 'hole'	color

(2 rows)

### H. 61. 3. 7. rum\_anyarray\_ops

对于类型: anyarray

此操作符类使用数组的长度存储anyarray元素。支持运算符&&, @>, <@, =, %。使用<=>运算符进行排序。例如，我们有以下表：

```
CREATE TABLE test_array (i int2[]);

INSERT INTO test_array VALUES ('{}'), ('{0}'), ('{1,2,3,4}'), ('{1,2,3}'),
    ('{1,2}'), ('{1}');
```

```
CREATE INDEX idx_array ON test_array USING rum (i rum_anyarray_ops);
```

现在我们可以使用索引扫描执行查询：

```
SET enable_seqscan TO off;

EXPLAIN (COSTS OFF) SELECT * FROM test_array WHERE i && '{1}' ORDER BY i <=>
    '{1}' ASC;
```

QUERY PLAN

```
-----
Index Scan using idx_array on test_array
  Index Cond: (i && '{1}'::smallint[])
  Order By: (i <=> '{1}'::smallint[])
(3 rows)
```

```
SELECT * FROM test_array WHERE i && '{1}' ORDER BY i <=> '{1}' ASC;
      i
-----
      {1}
      {1,2}
      {1,2,3}
      {1,2,3,4}
(4 rows)
```

### H. 61. 3. 8. rum\_anyarray\_addon\_ops

对于类型: anyarray

此操作符类使用模块支持的任何字段存储anyarray元素。

## H. 62. seg

这个模块为表示线段或者浮点区间实现了一种数据类型seg。 seg可以表示区间端点中的不确定性，这使得它对表达实验室测量特别有用。

该模块被认为是“trusted”，也就是说，它可以由对当前数据库具有 CREATE 权限的非超级用户安装。

### H. 62. 1. 原理

度量的几何结构通常比一个数字连续区中的一个点更复杂。 一个度量通常是具有一些模糊限制的连续区的一个分段。 由于不确定性和随机性，也因为被度量的值可能天然地就是一个指示某种情况的区间（例如一种蛋白质的稳态的温度范围）， 度量呈现为区间的形式。

只用常识，我们就知道将这类数据存储为区间比存储为数字对更加方便。 实际上，这样做在大部分应用中也更有效。

还是根据常识，限度的模糊性意味着使用传统的数字数据类型会导致信息丢失。 试想：你的仪器读到 6.50，并且你把这个读数输入到数据库。在你取出它时会得到什么？看看：

```
test=> select 6.50 :: float8 as "pH";
      pH
-----
      6.5
(1 row)
```

在度量世界里，6.50 和 6.5 并不相同。有时候它们可能很不同。实验者们通常会写下（并且发表）他们信任的数字。6.50 实际上是一个模糊的区间，它被包含于一个更大的而且更模糊的区间 6.5 中，它们的中心点（可能）是它们唯一共享的公共特征。我们绝对不希望这类不同的数据项表现得相同。

结论？一种能够记录具有任意可变精度的区间的限度的特殊数据类型是很好的。 这种意义下，每一个数据元素都记录其自身的精度。

来看看这个：

```
test=> select '6.25 .. 6.50' ::seg as "pH";
      pH
-----
6.25 .. 6.50
```



(1 row)

## H. 62. 2. 语法

一个区间的外部表达由通过范围操作符 (`..` 或者 `...`) 连接的一个或者两个浮点数构成。或者，它也可以被指定为一个中心点加上或者减去一个偏差值。也能够存储可选的确定性指示符 (`<`、`>` 或者 `~`)。不过，所有内建操作符会忽略确定性指示符。Table H. 183给出了所有允许的表达形式，Table H. 184展示了一些例子。

在Table H. 183中，`x`、`y`和`delta`表示浮点数。`x`和`y`可以前置一个确定性指示符，但是`delta`不行。

Table H. 183. `seg`外部表达

<code>x</code>	单一值（零长度区间）
<code>x .. y</code>	从 <code>x</code> 到 <code>y</code> 的区间
<code>x (+-) delta</code>	从 <code>x - delta</code> 到 <code>x + delta</code> 的区间
<code>x ..</code>	下界为 <code>x</code> 的开区间
<code>.. x</code>	上界为 <code>x</code> 的开区间

Table H. 184. 合法`seg`输入的例子

<code>5.0</code>	创建一个零长度的段（一个点）
<code>~5.0</code>	创建一个零长度的段并且在数据中记录 <code>~</code> 。 <code>~</code> 会被 <code>seg</code> 操作忽略，但是会被保留为一个注释。
<code>&lt;5.0</code>	在 <code>5.0</code> 创建一个点。 <code>&lt;</code> 会被忽略，但是被保留为一个注释。
<code>&gt;5.0</code>	在 <code>5.0</code> 创建一个点。 <code>&gt;</code> 会被忽略，但是被保留为一个注释。
<code>5(+-)0.3</code>	创建一个区间 <code>4.7 .. 5.3</code> 。注意 <code>(+)</code> 标记不会被保留。
<code>50 ..</code>	大于或等于 <code>50</code> 的所有东西
<code>.. 0</code>	小于或等于 <code>0</code> 的所有东西
<code>1.5e-2 .. 2E-2</code>	创建一个区间 <code>0.015 .. 0.02</code>
<code>1 ... 2</code>	与 <code>1..2</code> 、 <code>1 .. 2</code> 或者 <code>1..2</code> 相同（范围操作符周围的空格会被忽略）

由于`...`运算符被广泛地用在数据源中，它被允许作为`..`运算符的一种替代。不幸地是，这会带来解析歧义：分不清`0...23`的上界是`23`或者`0.23`。通过要求`seg`输入中所有数字的小数点前至少有一位可以解决这个问题。

作为一种完整性检查，`seg`会拒绝下界大于上界的区间，例如`5 .. 2`。

## H. 62. 3. 精度

`seg`值在内部被存储为一对 32 位浮点数。这意味着具有超过 7 个有效位的数字会被截断。

具有 7 个或者更少有效位的数字会保留它们的原有精度。即，如果你的查询返回 `0.00`，你可以确信拖尾的零不是人工造成的，它们反映了原始数据的精度。前导零的数量不影响精度：值 `0.0067` 被认为只有 2 个有效位。

## H. 62. 4. 用法

`seg`模块包括了用于`seg`值的一个 GiST 索引操作符类。该 GiST 操作符类所支持的操作符在Table H. 185中展示。

Table H.185. Seg GiST 操作符

操作符	简述
<code>seg &lt;&lt; seg</code>	布尔值 第一个seg是否完全位于第二个的左侧? <code>[a, b] &lt;&lt; [c, d]</code> is true if <code>b &lt; c</code> .
<code>seg &gt;&gt; seg</code>	布尔值 第一个seg是否完全位于第二个的右侧? <code>[a, b] &gt;&gt; [c, d]</code> is true if <code>a &gt; d</code> .
<code>seg &amp;&lt; seg</code>	布尔值 第一个seg没有延伸到第二个的右边吗? <code>[a, b] &amp;&lt; [c, d]</code> is true if <code>b &lt;= d</code> .
<code>seg &amp;&gt; seg</code>	布尔值 第一个seg没有延伸到第二个的左边吗? <code>[a, b] &amp;&gt; [c, d]</code> is true if <code>a &gt;= c</code> .
<code>seg = seg</code>	布尔值 两个seg是否相等?
<code>seg &amp;&amp; seg</code>	布尔值 两个seg是否重叠?
<code>seg @&gt; seg</code>	布尔值 第一个seg是否包含第二个?
<code>seg &lt;@ seg</code>	布尔值 第一个seg是否被第二个包含?

(在 LightDB 8.2 之前, 包含运算符 `@>` 和 `<@` 分别被称为 `@` 和 `~`。这些名称仍然可用, 但已被弃用并最终将被淘汰。注意, 旧名称与核心几何数据类型以前遵循的约定相反!)

除上述运算符外, Table 9. 中显示的常用比较运算符也可用于类型 `seg`。这些运算符首先比较 (a) 和 (c), 如果它们相等, 则比较 (b) 和 (d)。在大多数情况下, 这会导致相当好的排序, 如果您希望将 `ORDER BY` 用于此类型, 这将非常有用。

## H. 62. 5. 注解

使用的例子, 请见回归测试 `sql/seg.sql`。

将 `(+)` 转换成常规范范围的机制在确定边界的有效位数时并不完全准确。 例如, 如果结果区间包括一个 `10` 的幂时, 它会加上一个额外的位:

```
lightdb@postgres=> select '10(+)-1'::seg as seg;
seg
-----
9.0 .. 11          -- should be: 9 .. 11
```

一个 R-树索引的性能很大程度上依赖于输入值的初始顺序。 将输入表以 `seg` 列进行排序将会非常有帮助, 例子可见脚本 `sort-segments.pl`。

## H. 62. 6. 开发人员

原始作者: Gene Selkov, Jr. <selkovjr@mcsl.gov>, 阿尔贡国家实验室数学和计算机科学部。

我要感谢 Joe Hellerstein 教授 (<https://dsf.berkeley.edu/jmh/>) 解释了 GiST (<http://gist.cs.berkeley.edu/>)。 我也要向现在和过去的所有 Postgres 开发者致敬, 让我能创造自己的世界并且不受打扰地生活在其中。我也要感谢阿尔贡实验室以及美国能源部多年来对我数据库研究的支持。

## H. 63. sepgsql

sepgsql是一个基于SELinux安全策略的 支持基于标签的强制访问控制（MAC）模块。

### Warning

当前的实现具有明显的限制，并且不支持对所有动作的强制访问控制。详见 Section H.63.7

## H.63.1. 概述

这个模块和SELinux集成在一起在 LightDB提供的安全检查之上提供了一个 额外的安全检查层。从SELinux的角度来看，这个模块允许 LightDB作为一个用户空间对象管理器。对每一次由 DML 查询发起的表或者函数访问将根据系统安全策略进行检查。这种 检查是在LightDB执行的常规 SQL 权限 检查之外进行的。

SELinux访问控制决定是通过使用安全标签 来做出的，安全标签使用system\_u:object\_r:sepgsql\_table\_t:s0 这样的字符串表示。每个访问控制决定涉及两个标签：尝试执行该动作的主体的 标签以及要在其上执行该动作的客体的标签。由于这些标签可以被应用于任何种 类的对象，对于存储在数据库中的对象的（用这个模块做出的）访问控制决定服 从于用于任意其他类型对象（例如文件）的同一种一般准则。这种设计是为了允 许一种中央安全策略来保护信息资产，而不依赖于这些资产是如何存储的。

SECURITY LABEL语句允许为一个数据库对象分配安全标签。

## H.63.2. 安装

sepgsql只能在启用了 SELinux的 Linux 2.6.28 或者更高版本上使用。在任何 其他平台上都无法使用这个模块。你将还需要 libselinux 2.1.10 或者更高版本以及 selinux-policy 3.9.13 或者更高版本（尽管某些发行中可能 把必要的规则逆向移植到较老的策略版本中）。

你可以使用sestatus命令检查 SELinux的状态。一种典型的显示是：

```
$ sestatus
SELinux status:                enabled
SELinuxfs mount:              /selinux
Current mode:                  enforcing
Mode from config file:        enforcing
Policy version:                24
Policy from config file:      targeted
```

如果没有安装或者启用SELinux，你就必须在安装这个模块 之前先安装或者启用它。

要编译这个模块，应该在你的 LightDB configure 命令中包括 选项--with-selinux。还要确定编译时安装了 libselinux-devel RPM 包。

要使用这个模块，你必须在lightdb.conf文件中的 shared\_preload\_libraries参数里包括 sepgsql。如果以其他任何方式载入该模块，它将无法正确地工作。一旦该模块被载入，你应该在每一个数据库中执行 sepgsql.sql。这将会安装安全标签管理所需的函数 并且分配初始的安全标签。

这里有一个展示如何用sepgsql函数和安全标签初始化一个新 数据库集簇的例子（根据你的安装调整其中的路径）：

```
$ export LTDATA=/path/to/data/directory
$ initdb
$ vi $LTDATA/lightdb.conf
```

```
change
    #shared_preload_libraries = ''                # (change requires restart)
to
    shared_preload_libraries = 'sepgsql'        # (change requires restart)
$ for DBNAME in template0 template1 postgres; do
    lightdb --single -F -c exit_on_error=true $DBNAME \
        </usr/local/lightdb-x/share/contrib/sepgsql.sql >/dev/null
done
```

请注意，如果你具有特定版本的libselinux和 selinux-policy，你可能会看到下列提示中的一些或者全部：

```
/etc/selinux/targeted/contexts/sepgsql_contexts: line 33 has invalid object
type db_blobs
/etc/selinux/targeted/contexts/sepgsql_contexts: line 36 has invalid object
type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts: line 37 has invalid object
type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts: line 38 has invalid object
type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts: line 39 has invalid object
type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts: line 40 has invalid object
type db_language
```

这些消息是无害的并且应该被忽略。

如果该安装过程完成时没有出现错误，就可以正常启动服务器了。

### H. 63. 3. 回归测试

由于SELinux的本质，为sepgsql 运行回归测试要求一些额外的配置步骤，某些步骤还需要由 root 来完成。该回归测试 无法通过普通的make check或者make installcheck命令运行， 你必须建立配置并且接着手工调用测试脚本。这些测试必须在一个已配置 LightDB 编译树的contrib/sepgsql目录中运行。尽管它们要求一个编译树，但是 这些测试被设计成在一个已安装服务器上执行，也就是说它们可以比得上 make installcheck（而不是make check）。

首先，根据Section H. 63. 2中的指导在一个工作数据库中设置 sepgsql。注意当前操作系统用户必须能够不使用口令认证作为超级用户连接到该数据库。

第二，为该回归测试编译和安装策略包。sepgsql-regtest策略是一个 特殊的策略包，它提供一组在回归测试浅见要被允许的规则。它应该从策略源文件 sepgsql-regtest.te编译，这需要通过使用 make和一个 SELinux 提供的 Makefile 完成。你将需要 在你自己的系统上找到合适的 Makefile，下面展示的路径只是一个例子。（这个 Makefile 通常由 selinux-policy-devel 或 selinux-policy RPM 提供。）一旦编译好，使用semodule命令安装这个策略包，它会把所提供的策略包载入到 内核中。如果该包被正确地安装，semodule -l应该把 sepgsql-regtest列成一个可用的策略包：

```
$ cd .../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.07
```

第三，打开sepgsql\_regression\_test\_mode。由于安全性的原因， sepgsql-regtest中的规则默认没有被启用。 the sepgsql\_regression\_test\_mode参数会启用启动该回归 测试所需的规则。它可以使用setsebool命令来启用：

```
$ sudo setsebool sepgsql_regression_test_mode on
$ getsebool sepgsql_regression_test_mode
sepgsql_regression_test_mode --> on
```

第四，验证你的 shell 在unconfined\_t域中操作：

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

如果有必要，可以参考Section H.63.8来调整你的工作域。

最后，运行该回归测试脚本：

```
$ ./test_sepgsql
```

这个脚本将尝试验证你已经正确地完成了所有的配置步骤，接下来它将运行 sepgsql模块的回归测试。

完成测试后，推荐你禁用 sepgsql\_regression\_test\_mode参数：

```
$ sudo setsebool sepgsql_regression_test_mode off
```

你可能想要完全移除sepgsql-regtest策略：

```
$ sudo semodule -r sepgsql-regtest
```

## H.63.4. GUC 参数

sepgsql.permissive (boolean)

不管系统设置如何，这个参数让sepgsql在自由模式中运行。默认值为关闭。这个参数只能在lightdb.conf文件中或者 服务器命令行上被设置。

当这个参数为打开时，sepgsql在自由模式中运行，即便 SELinux 运行在强制模式中也是如此。这个参数主要用于测试目的。

sepgsql.debug\_audit (boolean)

不管系统策略设置如何，这个参数启用打印审计消息。默认值为关闭，表示将 根据系统设置打印消息。

SELinux的安全性策略也具有控制是否记录特定访问的 规则。默认情况下，违法访问将会被记录，但是被允许的访问则不会被记录。

这个参数强制打开所有可能的记录而不管该系统策略。

## H.63.5. 特性

### H.63.5.1. 控制对象类

SELinux的安全模型把所有访问控制规则描述为一个 主体（典型的是一个数据库客户端）和一个客体（例如一个数据库对象）之间的关系，每一个这样的关系被一个安全标签标识。如果尝试访问一个未加 标签的客体，会认为该客体被分配了标签unlabeled\_t。

当前，sepgsql允许把安全标签分配给模式、表、列、 序列、视图和函数。在使用sepgsql时，安全标签会 在所支持的数据库对象创建时自动分配给它们。这种标签被称为

默认安全标签 并且根据系统安全性策略决定，默认安全标签被用来输入创建者标签、分配给 新对象父对象的标签以及所构造对象的可选名称。

一个新数据库对象基本上会继承父对象的安全标签，不过当安全策略具有特殊的 类型转换规则时，将会应用一个不同的标签。对于模式，其父对象是当前数据库。对于表、序列、视图和函数，父对象是包含它的模式。对于列，其父对象是包含 它的表。

## H. 63. 5. 2. DML 权限

对于表，根据语句的种类会对所有被引用的目标表检查 db\_table:select、db\_table:insert、 db\_table:update或者db\_table:delete。此外，对于 所有其列被WHERE或RETURNING子句引用、作为 UPDATE的数据源（以及其他情况）的表，都要检查db\_table:select。

对每一个被引用的列也将检查列级权限。不仅在使用SELECT读取列 时会检查db\_column:select，在其他 DML 语句中引用列时也要检查。对于被UPDATE或者INSERT修改的列也将检查 db\_column:update或者db\_column:insert。

例如，考虑：

```
UPDATE t1 SET x = 2, y = func1(y) WHERE z = 100;
```

这里，将对t1.x检查db\_column:update，因为它 被更新。对t1.y将检查db\_column:{select update}，因为它既被更新也被引用。并且会对t1.z检查 db\_column:select，因为它只被更新。还将在表层面上检查 db\_table:{select update}。

对于序列，当我们使用SELECT引用一个序列对象时会检查 db\_sequence:get\_value。不过，我们当前不会检查执行相应 函数（例如lastval()）的权限。

对于视图，将检查db\_view:expand，然后对从视图展开来的任何 对象都会分别检查所需的权限。

对于函数，当用户尝试在一个查询中或者使用快路径调用执行一个函数时会检查 db\_procedure:{execute}。如果该函数是一个可信过程，也会检查 db\_procedure:{entrypoint}权限来看看它能否作为一个可信程序 的入口点来执行。

为了访问任何模式对象，在其所在的模式上需要db\_schema:search 权限。当不用模式限定引用一个对象时，其上没有该权限的模式不会被搜索（就好像该用户在该模式上没有USAGE特权）。如果出现一个显式的模式 限定，当该用户在提及的模式上没有要求的权限时将会发生一个错误。

客户端必须被允许访问所有引用到的表和列，即便它们是由视图扩展得来的。这样 我们可以应用一致的访问控制规则而不管表内容被引用的方式。

默认的数据库特权系统允许数据库超级用户使用 DML 命令修改系统目录并且引用 或者修改 TOAST 表。当sepgsql被启用时，这些操作会被禁止。

## H. 63. 5. 3. DDL 权限

SELinux为每一种对象类型定义了数个权限来控制 常用操作，例如创建、修改、删除以及重新标记安全标签。此外，数种 对象类型具有特殊的权限来控制它们的特性化操作，例如在一个特定模式 中增加或者删除名字项。

创建一个新的数据库对象要求create权限。 SELinux将基于客户端的安全标签来授予或者否决 这个权限并且为新对象提出安全标签。在某些情况下，还需要额外的特权：

- CREATE DATABASE额外要求源数据库或者模板数 据库的getattr权限。
- 创建一个模式对象额外地要求父模式上的add\_name权限。

- 创建一个表额外要求创建单个表列的权限，就好像每一个表列都是一个单独的顶层对象。
- 创建一个被标记为LEAKPROOF的函数额外要求 install 权限（当为一个现有函数设置 LEAKPROOF时也要检查这个权限）。

当执行DROP命令时，在要移除的对象上会检查drop。对于通过CASCADE间接被删除的对象也会检查权限。删除包含在一个特定模式内的对象（表、视图、序列以及过程）额外地要求该模式上的 remove\_name。

在执行ALTER命令时，会在被修改的对象上为每一种对象类型检查 setattr。附属对象（例如一个表的索引或者触发器）除外，这种情况下权限是在父对象上检查的。在某些情况下，还需要额外的权限：

- 将一个对象移动到一个新的模式要求旧模式上的remove\_name 权限以及新模式上的add\_name权限。
- 设置一个函数上的LEAKPROOF属性要求install权限。
- 在一个对象上使用SECURITY LABEL会额外对该对象要求 relabelfrom权限连同它的旧安全标签以及relabelto 权限连同它的新安全标签（在安装多个标签提供者并且用户尝试设置一个不由 SELinux管理的安全标签的情况中，这里只应该检查 setattr。当前由于实现限制没有这样做。）。

## H. 63. 5. 4. 可信过程

可信过程类似于 SECURITY DEFINER 函数或者 setuid 命令。SELinux提供了一个特性来允许可信代码使用一个不同于客户端的安全标签运行，通常这是为了提供对敏感数据的高度控制的访问（例如行可能会被忽略或者存储值的精度可能会被降低）。一个函数是否可以作为可信过程受到其安全标签和操作系统安全性策略的控制。例如：

```
postgres=# CREATE TABLE customer (
           cid      int primary key,
           cname   text,
           credit  text
        );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
           IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
           AS 'SELECT regexp_replace(credit, ''-[0-9]+$$'', ''-xxxx'', ''g'')
           FROM customer WHERE cid = $1'
           LANGUAGE sql;
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
           IS 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

上述的操作应该由一个管理员用户执行。

```
postgres=# SELECT * FROM customer;
ERROR: SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname | show_credit
-----+-----+-----
  1  | taro  | 1111-2222-3333-xxxx
  2  | hanako | 5555-6666-7777-xxxx
```

(2 rows)

在这种情况下，一个常规用户无法直接引用customer.credit，但是一个可信过程show\_credit允许用户在打印客户的信用卡号时 把一些数字掩盖掉。

### H. 63. 5. 5. 动态域转换

如果安全性策略允许，可以使用 SELinux 的动态域转换特性来切换客户端 进程（客户端域）的安全性标签到一个新的上下文。该客户端域需要 setcurrent权限还有从旧的域到新的域的 dyntransition权限。

动态域转换需要被仔细考虑，因为在用户看来，它们允许用户切换其标签，并且因而切换特权，而不是（像可信过程的情况那样）受系统的强制性管理。因此，只有当被用来切换到一个比原来的域具有更少特权的域时， dyntransition才被认为是安全的。例如：

```
regression=# select sepgsql_getcon();
                sepgsql_getcon
-----
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c4');
 sepgsql_setcon
-----
t
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c1023');
ERROR:  SELinux: security policy violation
```

在上面的这个例子中，我们被允许从较大范围的c1.c1023切换到 较小范围的c1.c4，但却禁止切换回去。

动态域转换和可信过程的组合开启了一种有趣的使用案例，它适合典型的 连接池软件的处理生命周期。即便你的连接池软件不被允许运行大部分的 SQL 命令，你可以从一个可信过程中使用 sepgsql\_setcon() 函数允许它切换该客户端的安全标签，这个过程应该采用一些证据来授权该请求切换该客户端标签。之后，这个 会话将会具有目标用户而不是连接池的特权。该连接池之后可以用 NULL参数再次调用sepgsql\_setcon() 逆转这次安全标签改变，当然再次的调用也要在一个可信过程中配合适当的权限检查进行。这里的要点是只有可信过程实际具有权限来更改有效的安全 标签，并且只有在得到适当的证据后才这样做。当然，对于安全操作，必须 保护证据存储（表、过程定义或者其他什么）不会受到未经授权的访问。

### H. 63. 5. 6. 杂项

我们全面拒绝LOAD命令，因为任何模块的装载都 可能很轻易地绕过安全策略的强制保护。

## H. 63. 6. Sepgsql 函数

Table H.18展示了可用的函数。

Table H.186. Sepgsql 函数

函数	简述
sepgsql_getcon ()	→ text



函数	简述
	返回客户端域，即客户端的当前安全标签。
sepgsql_setcon ( text ) → boolean	如果安全策略允许，则将当前会话的客户端域切换到新域。它还接受NULL输入作为转换到客户机原始域的请求。
sepgsql_mcstrans_in ( text ) → text	如果 mcstrans 守护进程正在运行，则将给定的限定 MLS/MCS 范围转换为原始格式。
sepgsql_mcstrans_out ( text ) → text	如果 mcstrans 守护程序正在运行，则将给定的原始 MLS/MCS 范围转换为限定格式。
sepgsql_restorecon ( text ) → boolean	为当前数据库中的所有对象设置初始安全标签。参数可以是NULL，也可以是要用作系统默认值替代项的规范文件的名称。

## H. 63. 7. 限制

数据定义语言 (DDL) 权限

收到实现的限制，一些 DDL 操作无法检查权限。

数据控制语言 (DCL) 权限

由于实现限制，DCL 操作不检查权限。

行级访问控制

LightDB支持行级访问，但是 sepgsql不支持行级访问。

隐蔽通道

sepgsql不会尝试隐藏一个特定对象的存在，即便是 用户不被允许引用该对象。例如，即便我们无法得到一个不可见对象 的内容，我们也可以通过主键冲突、外键违背等等结果来推知该对象 的存在。一个绝密表的存在无法被隐藏，我们只希望能够隐藏其内容。

## H. 63. 8. 外部资源

SE-LightDB 介绍<sup>15</sup>

这个 wiki 页面提供了一个简单的综述、安全性设计、架构、管理和即将到来的特性。

SELinux 用户和管理员指南<sup>16</sup>

这个文档提供了广泛的知识来管理系统上的 SELinux。它主要关注 Red Hat 操作系统，但是并不仅限于此。

Fedora SELinux FAQ<sup>17</sup>

这个文档回答了很多SELinux 的常见问题。它主要关注 Fedora，但是并不仅限于此。

## H. 63. 9. 作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

<sup>15</sup> <https://wiki.postgresql.org/wiki/SEPostgreSQL>

<sup>16</sup> [https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/7/html/selinux\\_users\\_and\\_administrators\\_guide/index](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/index)

<sup>17</sup> [https://fedoraproject.org/wiki/SELinux\\_FAQ](https://fedoraproject.org/wiki/SELinux_FAQ)

## H. 64. spi

spi模块提供了多个可工作的使用服务器编程接口（SPI）和触发器的例子。尽管这些例子的价值只对它们自己合适，它们甚至更有助于作为例子来修改达到你自己的目的。这些函数足够普通，可以与任何表一起使用，但是在创建一个触发器时你必须指定表名和域名（如下所述）。

下面描述的函数组中的每一个都作为一个独立可安装的扩展被提供。

### H. 64. 1. refint — 用于实现参照完整性的函数

`check_primary_key()`和`check_foreign_key()`被用来检查外键约束（当然，这个功能很早以前被内建的外建机制取代了，但是该模块还是可以作为例子）。

`check_primary_key()`检查引用表。用法是使用这个函数在一个引用其他表的表上创建一个BEFORE INSERT OR UPDATE触发器。指定该触发器的参数为：引用表中构成外键的列名、被引用表名称以及在被引用表中构成主键/唯一键的列名。要处理多个外键，请为每一个引用创建一个触发器。

`check_foreign_key()`检查被引用表。用法是使用这个函数在一个被其他表引用的表上创建一个BEFORE DELETE OR UPDATE触发器。指定该触发器的参数为：该函数必须对其执行检查的引用表数量、找到一个引用键后的动作（`cascade` — 删除引用行，`restrict` — 如果引用键存在则中断事务，`setnull` — 设置引用键域为空）、触发器所在表中构成主键/唯一键的列名、引用表名称和列名称（第一个参数指定多少个引用表就重复多少次）。注意主键/唯一键列应该被标记为 NOT NULL 并且应该有一个唯一索引。

`refint.example`中有一些例子。

### H. 64. 2. autoinc — 用于自增域的函数

`autoinc()`是一个将序列的下一个值存储到一个整数域的触发器。这和内建的“序列”特性有些重叠，但是它并不完全一样：`autoinc()`在插入时会覆盖掉给出的不同域值，并且它可被选择用来在更新时增加域。

用法是使用这个函数创建一个BEFORE INSERT（或者BEFORE INSERT OR UPDATE）触发器。指定两个触发器参数：要被修改的整数列名和将提供值的序列对象名（事实上，如果你想要更新多于一个自增列，你可以指定任意数量的这种名称对）。

在`autoinc.example`中有一个例子。

### H. 64. 3. insert\_username — 用于跟踪谁修改了一个表的函数

`insert_username()`是存储当前用户名到一个文本域的触发器。这有助于跟踪是谁最后在一个表中修改了一个特定行。

用法是使用这个函数创建一个BEFORE INSERT以及/或者UPDATE触发器。指定一个触发器参数：要被修改的文本列名。

在`insert_username.example`中有一个例子。

### H. 64. 4. moddatetime — 用于跟踪上一次修改时间的函数

`moddatetime()`是一个存储当前时间到一个timestamp域的触发器。它有助于跟踪一个表中特定行最后一次的修改时间。

用法是使用这个函数创建一个BEFORE UPDATE触发器。指定一个触发器参数：要被修改的列名。该列必须是类型timestamp或者timestamp with time zone。

在moddatetime.example中有一个例子。

## H. 65. sslinfo

在连接到LightDB时，sslinfo模块提供当前客户端提供的 SSL 证书的有关信息。如果当前连接不使用 SSL，这个模块就没有用处（大部分函数将返回 NULL）。

本模块提供的一些信息，也可以使用内置系统视图 pg\_stat\_ssl来获得。

除非安装时用--with-ssl=openssl进行了配置，这个扩展压根就不会被编译。

### H. 65. 1. 提供的函数

ssl\_is\_used() 返回 boolean

如果当前到服务器的连接使用 SSL 则返回 true，否则返回 false。

ssl\_version() 返回 text

返回 SSL 连接使用的协议名称（如 TLSv1.0、TLSv1.1、TLSv1.2 或者 TLSv1.3）。

ssl\_cipher() 返回 text

返回 SSL 连接所用的加密方法名称（如 DHE-RSA-AES256-SHA）。

ssl\_client\_cert\_present() 返回 boolean

如果当前客户端已经向服务器出示了一个合法的 SSL 客户端证书则返回 true，否则返回 false（服务器可能被配置要求一个客户端配置，也可能没有被配置成这样）。

ssl\_client\_serial() 返回 numeric

返回当前客户端证书的序列号。证书序列号和证书发行人的组合被确保可以唯一标识一个证书（但是不能唯一标识其拥有者 — 拥有者应该定期地更换其密钥，并且从发行人那里得到新的证书）。

因此，如果你运行自己的 CA 并且只允许服务器接收来自于这个 CA 的证书，序列号就是最可靠的（虽然并非很好记忆）标识一个用户的方法。

ssl\_client\_dn() 返回 text

返回当前客户端证书的完整主题，并且将字符数据转换成当前数据库的编码。我们假定如果你在证书名中使用非 ASCII 字符，你的数据库也有能力展示这些字符。如果你的数据库使用 SQL\_ASCII 编码，名称中的非 ASCII 字符将被表示为 UTF-8 序列。

结果看起来像/CN=某人 /C=某个国家 /O=某个组织。

ssl\_issuer\_dn() 返回 text

返回当前客户端证书的完整的发行人名称，并把字符数据转换成当前数据库的编码。编码转换以与ssl\_client\_dn相同的方式处理。

这个函数的返回值与证书序列号的组合唯一地标识证书。

如果在服务器的证书授权中心文件中有多于一个的可信 CA 证书，或者如果 CAI 已经发行了某些中间认证授权证书，这个函数就真的很有用。

`ssl_client_dn_field(fieldname text)` 返回 `text`

这个函数返回证书主题中指定域的值，如果域不存在则返回 `NULL`。域的名称是字符串常量，它们被使用OpenSSL对象数据库转换成 ASN1 对象标识符。下列值是可接受的：

```
commonName (alias CN)
surname (alias SN)
name
givenName (alias GN)
countryName (alias C)
localityName (alias L)
stateOrProvinceName (alias ST)
organizationName (alias O)
organizationalUnitName (alias OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

这些域中除了`commonName`都是可选的。它们之中哪些会被包括或者不会被包括完全取决于你的 CA 策略。不过，这些域的含义由 X.500 和 X.509 标准严格地定义，因此你不能只是为它们分配任意含义。

`ssl_issuer_dn_field(fieldname text)` 返回 `text`

和`ssl_client_dn_field`一样，但是用于证书发行人而不是证书主题。

`ssl_extension_info()` 返回 `setof record`

提供有关客户端证书扩展的信息：扩展名、扩展值以及是否为决定性的扩展。

## H. 65. 2. 作者

Victor Wagner <vitus@cryptocom.ru>, Cryptocom LTD

Dmitry Voronin <carriingfate92@yandex.ru>

Cryptocom OpenSSL 开发组的 E-Mail: <openssl@cryptocom.ru>

## H. 66. system\_stats

`system_stats`是LightDB的扩展，提供用于访问系统级别的统计信息的函数，可用于监视。

### H. 66. 1. 安全性

由于这些函数返回的信息的性质，访问仅限超级用户和当扩展安装时创建的 `monitor_system_stats`角色的成员。要允许用户访问这些函数而不授予超级用户访问权限，请将它们添加到`monitor_system_stats`角色中。例如：

```
GRANT monitor_system_stats to nagios;
```

## H. 66. 2. 函数

提供以下函数以获取所有平台的系统级别统计信息。

### H. 66. 2. 1. pg\_sys\_os\_info

此接口允许用户获取操作系统统计信息。

### H. 66. 2. 2. pg\_sys\_cpu\_info

此接口允许用户获取CPU信息。

### H. 66. 2. 3. pg\_sys\_cpu\_usage\_info

此接口允许用户获取CPU使用信息。值是CPU为所有操作花费的时间的百分比。

### H. 66. 2. 4. pg\_sys\_memory\_info

此接口允许用户获取内存使用信息。所有值都以字节为单位。

### H. 66. 2. 5. pg\_sys\_io\_analysis\_info

此接口允许用户获取块设备的I/O分析。

### H. 66. 2. 6. pg\_sys\_disk\_info

此接口允许用户获取磁盘信息。

### H. 66. 2. 7. pg\_sys\_load\_avg\_info

此接口允许用户获取系统在1、5、10和15分钟间隔内的平均负载。

### H. 66. 2. 8. pg\_sys\_process\_info

此接口允许用户获取进程信息。

### H. 66. 2. 9. pg\_sys\_network\_info

此接口允许用户获取网络接口信息。

### H. 66. 2. 10. pg\_sys\_cpu\_memory\_by\_process

此接口允许用户获取每个进程ID的CPU和内存信息。

## H. 66. 3. 每个函数的详细输出

### H. 66. 3. 1. pg\_sys\_os\_info

- 名称
- 版本
- 主机名
- 域名
- 句柄数
- 进程数

- 线程数
- 架构
- 上次启动时间
- 运行时间（秒）

#### H. 66. 3. 2. pg\_sys\_cpu\_info

- 厂商
- 描述
- 型号名称
- 处理器类型
- 逻辑处理器数
- 物理处理器数
- 核心数
- 架构
- 时钟速度（赫兹）
- CPU类型
- CPU家族
- 字节顺序
- L1d缓存大小
- L1i缓存大小
- L2缓存大小
- L3缓存大小

#### H. 66. 3. 3. pg\_sys\_cpu\_usage\_info

- 用户模式普通进程处理所占时间百分比
- 用户模式优先级较低进程处理所占时间百分比
- 内核模式进程处理所占时间百分比
- 空闲模式所占时间百分比
- IO完成所占时间百分比
- 服务中断所占时间百分比
- 服务软件中断所占时间百分比
- 用户时间所占时间百分比
- 处理器时间所占时间百分比
- 特权时间所占时间百分比
- 中断时间所占时间百分比

#### H. 66. 3. 4. pg\_sys\_memory\_info

- 总内存
- 已用内存
- 空闲内存
- 总交换空间
- 已用交换空间
- 空闲交换空间
- 总缓存内存
- 总内核内存
- 内核分页内存
- 内核非分页内存
- 总页面文件
- 可用页面文件

#### H. 66. 3. 5. pg\_sys\_io\_analysis\_info

- 块设备名称
- 总读取次数
- 总写入次数

- 读取字节数
- 写入字节数
- 读取所花费的毫秒数
- 写入所花费的毫秒数

#### H. 66. 3. 6. `pg_sys_disk_info`

- 磁盘的文件系统
- 文件系统类型
- 文件系统的挂载点
- 驱动器盘符
- 驱动器类型
- 总空间（以字节为单位）
- 已用空间（以字节为单位）
- 可用空间（以字节为单位）
- 总 inode 数量
- 已使用的 inode 数量
- 空闲 inode 数量

#### H. 66. 3. 7. `pg_sys_load_avg_info`

- 1 分钟平均负载
- 5 分钟平均负载
- 10 分钟平均负载
- 15 分钟平均负载

#### H. 66. 3. 8. `pg_sys_process_info`

- 总进程数量
- 正在运行的进程数量
- 睡眠中的进程数量
- 停止的进程数量
- 僵尸进程数量

#### H. 66. 3. 9. `pg_sys_network_info`

- 接口名称
- 接口的 IPv4 地址
- 总发送字节数
- 总发送数据包数量
- 此网络设备发送错误的次数
- 传输期间丢失的数据包数量
- 总接收字节数
- 总接收数据包数量
- 此网络设备接收错误的次数
- 此网络设备接收丢弃的数据包数量
- 接口速度（以 Mbps 为单位）

#### H. 66. 3. 10. `pg_sys_cpu_memory_by_process`

- 进程的 PID
- 进程名称
- CPU使用量（以字节为单位）
- 内存使用量（以字节为单位）
- 使用的总内存量（以字节为单位）

### H. 67. `tablefunc`

tablefunc模块包括多个返回表（也就是多行）的函数。这些函数都很有用，并且也可以作为如何编写返回多行的 C 函数的例子。

这个模块被认为是“可信的”，也就是说，它可以由对当前数据库具有CREATE权限的非超级用户安装。

## H. 67. 1. 所提供的函数

Table H. 18总结了tablefunc模块提供的函数。

Table H. 187. tablefunc函数

函数	简述
normal_rand ( numvals integer, mean float8, stddev float8 )	→ 浮点数集 产生一组正态分布的随机值。
crosstab ( sql text )	→ 记录集 生成一个“数据透视表”，其中包含行名称和 N 列值，其中 N 由调用查询中指定的行类型决定。
crosstabN ( sql text )	→ table_crosstab_集N 产生一个包含行名称外加N个值列的“数据透视表”。crosstab2、crosstab3和crosstab4是被预定义的，但你可以按照下文所述创建额外的crosstabN函数。
crosstab ( source_sql text, category_sql text )	→ 记录集 产生一个“数据透视表”，其值列由第二个查询指定。
crosstab ( sql text, N integer )	→ 记录集 crosstab(text)的废弃版本。参数N现在被忽略，因为值列的数量总是由调用查询所决定。
connectby ( relname text, keyid_fld text, parent_keyid_fld text [, orderby_fld text ], start_with text, max_depth integer [, branch_delim text ] )	→ 记录集 产生一个层次树结构的表达。

### H. 67. 1. 1. normal\_rand

normal\_rand(int numvals, float8 mean, float8 stddev) returns setof float8

normal\_rand产生一个正态分布随机值（高斯分布）的集合。

numvals是从该函数返回的值的数量。mean是值的正态分布的均值而stddev是值的正态分布的标准偏差。

例如，这个调用请求 1000 个值，它们具有均值 5 和标准偏差 3:

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
normal_rand
-----
1. 56556322244898
9. 10040991424657
5. 36957140345079
-0. 369151492880995
0. 283600703686639
.
.
```



```

      .
      4. 82992125404908
      9. 71308014517282
      2. 49639286969028
(1000 rows)

```

## H. 67. 1. 2. crosstab(text)

```

crosstab(text sql)
crosstab(text sql, int N)

```

crosstab函数被用来产生“pivot”显示，在其中数据被横布在页面上而不是直接向下列举。例如，我们可能有这样的数据

```

row1  val11
row1  val12
row1  val13
...
row2  val21
row2  val22
row2  val23
...

```

而我们希望显示成这样

```

row1  val11  val12  val13  ...
row2  val21  val22  val23  ...
...

```

crosstab函数会采用一个文本参数，该文本是一个 SQL 查询，它产生按照第一种方式格式化的原始数据，并且产生以第二种方式格式化的一个表。

sql参数是一个产生数据的源集合的 SQL 语句。这个语句必须返回一个row\_name列、一个category列和一个value列。N是一个废弃参数，即使提供也会被忽略（之前这必须匹配输出值列的数目，但是现在这由调用查询决定了）。

例如，所提供的查询可能会产生这样的一个集合：

```

row_name  cat  value
-----+-----+-----
row1     cat1  val1
row1     cat2  val2
row1     cat3  val3
row1     cat4  val4
row2     cat1  val5
row2     cat2  val6
row2     cat3  val7
row2     cat4  val8

```

crosstab函数被声明为返回setof record，因此输出列的实际名称和类型必须定义在调用的SELECT语句的FROM子句中，例如：

```

SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2
text);

```

这个例子产生这样一个集合：

```

<== value columns ==>
row_name  category_1  category_2
-----+-----+-----
row1      val1          val2
row2      val5          val6
    
```

FROM子句必须把输出定义为一个row\_name列（具有 SQL 查询的第一个结果列的相同数据类型），其后跟随着 N 个value列（都具有 SQL 查询的第三个结果列的相同数据类型）。你可以按照你的意愿设置任意多的输出值列。而输出列的名称取决于你。

crosstab函数为具有相同row\_name值的输入行的每一个连续分组产生一个输出行。它使用来自这些行的值域从左至右填充输出的值列。如果一个分组中的行比输出值列少，多余的输出列将被用空值填充。如果行更多，则多余的输入行会被跳过。

事实上，SQL 查询应该总是指定ORDER BY 1, 2来保证输入行被正确地排序，也就是说具有相同row\_name的值会被放在一起并且在行内被正确地排序。注意crosstab本身并不关注查询结果的第二列，它放在那里只是为了被排序，以便控制出现在页面上的第三列值的顺序。

这是一个完整的例子：

```

CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
    'select rowid, attribute, value
    from ct
    where attribute = ''att2'' or attribute = ''att3''
    order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);
    
```

```

row_name | category_1 | category_2 | category_3
-----+-----+-----+-----
test1    | val2       | val3       |
test2    | val6       | val7       |
(2 rows)
    
```

你可以避免总是要写出一个FROM子句来定义输出列，方法是设置一个在其定义中硬编码所期望的输出行类型的自定义 crosstab 函数。这会在下一节中描述。另一种可能性是在一个视图定义中嵌入所需的FROM子句。

**Note**

另见ltsql中的 \crosstabview 命令，它提供了和crosstab()类似的功能。

### H. 67. 1. 3. crosstabN(text)

```
crosstabN(text sql)
```

crosstabN系列函数是如何为普通crosstab 函数设置自定义包装器的例子，这样你不需要在调用的SELECT查询中 写出列名和类型。tablefunc模块包括crosstab2、 crosstab3以及crosstab4，它们的输入行类型被定义为：

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
    .
    .
    .
    category_N TEXT
);
```

因此，当输入查询产生类型为text的列row\_name和value 并且想要 2、3 或 4 个输出值列时，这些函数可以被直接使用。在所有其他方法中，它们的行为都和上面的一般crosstab函数完全相同。

例如，前一节给出的例子也可以这样做

```
SELECT *
FROM crosstab3(
    'select rowid, attribute, value
    from ct
    where attribute = ''att2'' or attribute = ''att3''
    order by 1,2');
```

这些函数主要是出于举例的目的而提供。你可以基于底层的crosstab()函数 创建你自己的返回类型和函数。有两种方法来做：

- 与contrib/tablefunc/tablefunc--1.0.sql中相似，创建一个组合类型来描述所期望的输出列。 然后定义一个唯一的函数名，它接受一个text参数并且返回setof your\_type\_name，但是链接到同样的 底层crosstab C 函数。例如，如果你的源数据产生为text类型的行名称，并且值是float8， 并且你想要 5 个值列：

```
CREATE TYPE my_crosstab_float8_5_cols AS (
    my_row_name text,
    my_category_1 float8,
    my_category_2 float8,
    my_category_3 float8,
    my_category_4 float8,
    my_category_5 float8
);

CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
    RETURNS setof my_crosstab_float8_5_cols
    AS '$libdir/tablefunc', 'crosstab' LANGUAGE C STABLE STRICT;
```

- 使用OUT参数来隐式定义返回类型。同样的例子也可以这样做：

```
CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
    IN text,
    OUT my_row_name text,
```

```

    OUT my_category_1 float8,
    OUT my_category_2 float8,
    OUT my_category_3 float8,
    OUT my_category_4 float8,
    OUT my_category_5 float8)
    RETURNS setof record
    AS '$libdir/tablefunc', 'crosstab' LANGUAGE C STABLE STRICT;

```

### H.67.1.4. crosstab(text, text)

crosstab(text source\_sql, text category\_sql)

crosstab的单一参数形式的主要限制是它把一个组中的所有值都视作相似，并且把每一个值插入到第一个可用的列中。如果你想要值列对应于特定的数据分类，并且某些分组可能没有关于某些分类的数据，这样的形式就无法工作。crosstab的双参数形式通过提供一个对应于输出列的显式分类列表来处理这种情况。

source\_sql是一个产生源数据集的 SQL 语句。这个语句必须返回一个 row\_name列、一个category列以及一个value列。也可以有一个或者多个“extra”列。row\_name列必须是第一个。category和value列必须是按照这个顺序的最后两个列。row\_name和category之间的任何列都被视作“extra”。对于具有相同row\_name值的所有行，其“extra”列都应该相同。

例如，source\_sql可能产生一组这样的东西：

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;
```

row_name	extra_col	cat	value
row1	extra1	cat1	val1
row1	extra1	cat2	val2
row1	extra1	cat4	val4
row2	extra2	cat1	val5
row2	extra2	cat2	val6
row2	extra2	cat3	val7
row2	extra2	cat4	val8

category\_sql是一个产生分类集合的 SQL 语句。这个语句必须只返回一列。它必须产生至少一行，否则会生成一个错误。还有，它不能产生重复值，否则会生成一个错误。category\_sql可能是这样的：

```

SELECT DISTINCT cat FROM foo ORDER BY 1;
   cat
-----
   cat1
   cat2
   cat3
   cat4

```

crosstab函数被声明为返回setof record，这样输出列的实际名称和类型就必须在调用的SELECT语句的FROM子句中被定义，例如：

```

SELECT * FROM crosstab('...', '...')
    AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4
    text);

```

这将产生这样的结果:

	<== value columns ==>				
row_name	extra	cat1	cat2	cat3	cat4
row1	extral	val1	val2		val4
row2	extra2	val5	val6	val7	val8

FROM子句必须定义正确数量的输出列以及正确的数据类型。如果在source\_sql 查询的结果中有N列，其中的前N-2 列必须匹配前N-2 个输出列。剩余的输出列必须具有source\_sql查询结果的最后一列的类型，并且并且它们的数量 必须正好和source\_sql查询结果中的行数相同。

crosstab函数为具有相同row\_name值的输入行形成的每一个连续分组 产生一个输出行。输出的row\_name列外加任意一个“extra”列都是从分组的 第一行复制而来。输出的value列被使用具有匹配的category值的行中的 value域填充。如果一个行的category不匹配category\_sql 查询的任何输出，它的value会被忽略。匹配的分类不出现在于分组中任何输出行中的的 输出列会被用空值填充。

事实上，source\_sql查询应该总是指定ORDER BY 1来保证 具有相同row\_name的值会被放在一起。但是，一个分组内分类的顺序并不重要。 还有，确保category\_sql查询的输出的顺序与指定的输出列顺序匹配是非常重要的。

这里有两个完整的例子:

```

create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
    'select year, month, qty from sales order by 1',
    'select m from generate_series(1,12) m'
) as (
    year int,
    "Jan" int,
    "Feb" int,
    "Mar" int,
    "Apr" int,
    "May" int,
    "Jun" int,
    "Jul" int,
    "Aug" int,
    "Sep" int,
    "Oct" int,
    "Nov" int,
    "Dec" int
);

```

year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2007	1000	1500					500				1500	
2008	1000											

(2 rows)

```
CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March
2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');
```

```
SELECT * FROM crosstab
```

```
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
```

```
AS
```

```
(
  rowid text,
  rowdt timestamp,
  temperature int4,
  test_result text,
  test_startdate timestamp,
  volts float8
);
```

```
rowid |          rowdt          | temperature | test_result |
test_startdate | volts
-----+-----+-----+-----+
+-----+-----+-----+-----+
test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |
      | 2.6987                    |
test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat Mar 01
00:00:00 2003 | 3.1234
```

(2 rows)

你可以创建预定义的函数来避免在每个查询中都必须写出结果列的名称和类型。请参考前一节中的例子。用于这种形式的crosstab的底层 C 函数被命名为crosstab\_hash。

## H. 67. 1. 5. connectby

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

connectby函数产生存储在一个表中的层次数据的显示。该表必须具有一个用以 唯一标识行的键域，以及一个父亲键域用来引用其父亲（如果有）。connectby能 显示从任意行开始向下的子树。

Table H. 188解释了参数。

Table H. 188. connectby 参数

参数	描述
relname	源关系的名称
keyid_fld	键域的名称

参数	描述
parent_keyid_fld	父亲键域的名称
orderby_fld	用于排序兄弟的域的名称（可选）
start_with	起始行的键值
max_depth	要向下的最大深度，零表示无限深度
branch_delim	在分支输出中用于分隔键值的字符串（可选）

键域和父亲键域可以是任意数据类型，但是它们必须是同一类型。 注意start\_with值必须作为一个文本串被输入，而不管键域的类型如何。

connectby函数被声明为返回setof record，因此输出列的实际名称和类型 就必须在调用的SELECT语句的FROM子句中被定义，例如：

```
SELECT * FROM connectby(' connectby_tree', 'keyid', 'parent_keyid', 'pos',
    'row2', 0, '~')
    AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

前两个输出列被用于当前行的键和其父亲行的键，它们必须匹配该表的键域的类型。第三个输出行是该树中的深度， 并且必须是类型integer。如果给定了一个branch\_delim参数，下一个输出列 就是分支显示并且必须是类型text。最后，如果给出了一个orderby\_fld参数，最后一个输出列是一个序号，并且必须是类型integer。

“branch” 输出列显示了用于到达当前行的由键构成的路径。其中的键用指定的branch\_delim 字符串分隔开。如果不需要分支显示，可以在输出列列表中忽略branch\_delim参数和分支列。

如果同一父亲的子女之间的顺序很重要，可以包括orderby\_fld参数以指定用哪个域对兄弟排序。 这个域可以是任何可排序数据类型。当且仅当orderby\_fld被指定时，输出列列表必须包括一个 最终的整数序号列。

表示表和列名的参数会被原样复制到connectby内部生成的 SQL 查询中。因此，如果名称是大小写混合或者包含特殊字符，应包括双引号。你也可能需要用模式限定表名。

在大型的表中，除非在父亲键域上有索引，否则性能会很差。

branch\_delim字符串不出现在任何键值中是很重要的，否则connectby可能会错误地 报告一个无限递归错误。注意如果没有提供branch\_delim，将用一个默认值~来进行递归检测。

这里是一个例子：

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES(' row1', NULL, 0);
INSERT INTO connectby_tree VALUES(' row2', ' row1', 0);
INSERT INTO connectby_tree VALUES(' row3', ' row1', 0);
INSERT INTO connectby_tree VALUES(' row4', ' row2', 1);
INSERT INTO connectby_tree VALUES(' row5', ' row2', 0);
INSERT INTO connectby_tree VALUES(' row6', ' row4', 0);
INSERT INTO connectby_tree VALUES(' row7', ' row3', 0);
INSERT INTO connectby_tree VALUES(' row8', ' row6', 0);
INSERT INTO connectby_tree VALUES(' row9', ' row5', 0);

-- 带有分支，但没有 orderby_fld （不保证结果的顺序）
SELECT * FROM connectby(' connectby_tree', 'keyid', 'parent_keyid', 'row2', 0,
    '~')
```

```
AS t(keyid text, parent_keyid text, level int, branch text);
keyid | parent_keyid | level | branch
-----|-----|-----|-----
row2  |               | 0    | row2
row4  | row2          | 1    | row2~row4
row6  | row4          | 2    | row2~row4~row6
row8  | row6          | 3    | row2~row4~row6~row8
row5  | row2          | 1    | row2~row5
row9  | row5          | 2    | row2~row5~row9
(6 rows)
```

-- 没有分支, 也没有 orderby\_fld (不保证结果的顺序)

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
AS t(keyid text, parent_keyid text, level int);
keyid | parent_keyid | level
-----|-----|-----
row2  |               | 0
row4  | row2          | 1
row6  | row4          | 2
row8  | row6          | 3
row5  | row2          | 1
row9  | row5          | 2
(6 rows)
```

-- 有分支, 有 orderby\_fld (注意 row5 在 row4 前面)

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos',
'row2', 0, '~')
AS t(keyid text, parent_keyid text, level int, branch text, pos int);
keyid | parent_keyid | level | branch | pos
-----|-----|-----|-----|-----
row2  |               | 0    | row2   | 1
row5  | row2          | 1    | row2~row5 | 2
row9  | row5          | 2    | row2~row5~row9 | 3
row4  | row2          | 1    | row2~row4 | 4
row6  | row4          | 2    | row2~row4~row6 | 5
row8  | row6          | 3    | row2~row4~row6~row8 | 6
(6 rows)
```

-- 没有分支, 有 orderby\_fld (注意 row5 在 row4 前面)

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos',
'row2', 0)
AS t(keyid text, parent_keyid text, level int, pos int);
keyid | parent_keyid | level | pos
-----|-----|-----|-----
row2  |               | 0    | 1
row5  | row2          | 1    | 2
row9  | row5          | 2    | 3
row4  | row2          | 1    | 4
row6  | row4          | 2    | 5
row8  | row6          | 3    | 6
(6 rows)
```

## H. 67. 2. 作者

Joe Conway

## H. 68. tcn



tcn模块提供一个触发器函数，它通知监听者有关它所附着的任意表上的改变。它必须被用作一个行级AFTER触发器。

这个模块被认为是“可信的”，也就是说，它可以由对当前数据库具有CREATE权限的非超级用户安装。

在一个CREATE TRIGGER语句中只可以为该函数提供一个参数，并且是可选的。如果提供该参数，它将被作为用于通知的频道名。如果忽略它，频道名将使用tcn。

通知的负载由表名、一个指示所执行操作类型的字母以及用于主键列的列名/值对构成。每一部分都用逗号与下一部分隔开。为了便于解析对正则表达式的使用，表和列名总是被包裹在双引号内，并且数据值总是被包裹在单引号内。嵌入的引号都被双写。

下面是使用该扩展的简单例子。

```
test=# create table tcndata
test=# (
test(#   a int not null,
test(#   b date not null,
test(#   c text,
test(#   primary key (a, b)
test(# );
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test=# after insert or update or delete on tcndata
test=# for each row execute function triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test=#                               (1, date '2012-12-23', 'another'),
test=#                               (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload
""tcndata",I,"a"='1',"b"='2012-12-22'" received from server process with PID
22770.
Asynchronous notification "tcn" with payload
""tcndata",I,"a"='1',"b"='2012-12-23'" received from server process with PID
22770.
Asynchronous notification "tcn" with payload
""tcndata",I,"a"='2',"b"='2012-12-23'" received from server process with PID
22770.
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload
""tcndata",U,"a"='1',"b"='2012-12-22'" received from server process with PID
22770.
Asynchronous notification "tcn" with payload
""tcndata",U,"a"='1',"b"='2012-12-23'" received from server process with PID
22770.
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload
""tcndata",D,"a"='1',"b"='2012-12-22'" received from server process with PID
22770.
```

## H. 69. test\_decoding

test\_decoding是一个逻辑解码输出插件的例子。它其实不做任何特别有用的事情，但是可以作为开发你自己的输出插件的起点。

test\_decoding通过逻辑解码机制接收 WAL 并且把它解码成 被执行的操作的文本表达形式。

这个插件的典型输出（在 SQL 逻辑解码接口上使用）可能是：

```
lightdb@postgres=# SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL,
NULL, 'include-xids', '0');
 lsn      |  xid  | data
-----+-----+-----
0/16D30F8 | 691   | BEGIN
0/16D32A0 | 691   | table public.data: INSERT: id[int4]:2 data[text]:'arg'
0/16D32A0 | 691   | table public.data: INSERT: id[int4]:3 data[text]:'demo'
0/16D32A0 | 691   | COMMIT
0/16D32D8 | 692   | BEGIN
0/16D3398 | 692   | table public.data: DELETE: id[int4]:2
0/16D3398 | 692   | table public.data: DELETE: id[int4]:3
0/16D3398 | 692   | COMMIT
(8 rows)
```

## H. 70. tsm\_system\_rows

tsm\_system\_rows模块提供了表采样方法 SYSTEM\_ROWS，它可以用在SELECT 命令的TABLESAMPLE子句中。

这种表采样方法接受一个整数参数，它是要读取的最大行数。得到的采样将总是包含正好这么多行，除非该表中没有足够的行，在那种情况下整个表都会被选择出来。

和内建的SYSTEM采样方法一样，SYSTEM\_ROWS执行块级别的采样，所以采样不是完全随机的，而是服从于聚簇效果，特别是只要求少量行时。

SYSTEM\_ROWS不支持 REPEATABLE子句。

这个模块被认为是“可信的”，也就是说，它可以由对当前数据库具有CREATE权限的非超级用户安装。

### H. 70. 1. 示例

这里是一个用SYSTEM\_ROWS选择一个表采样的例子。首先安装扩展：

```
CREATE EXTENSION tsm_system_rows;
```

然后就可以在SELECT命令中使用它，例如：

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_ROWS(100);
```

这个命令从表my\_table中返回一个 100 行的采样（除非 该表没有 100 个可见行，那时将会返回其中所有的行）。

## H. 71. tsm\_system\_time

tsm\_system\_time模块提供了表采样方法 SYSTEM\_TIME，它可以用在SELECT 命令的TABLESAMPLE子句中。

这种表采样方法接受一个浮点类型的参数，它是花费在读表上的最大毫秒数。这可以让你直接控制查询进行多久，但付出的代价是很难预测采样的尺寸。得到的采样将包含在指定时间内能读入的那么多行，除非首先已经读入了整个表。

和内建的SYSTEM采样方法一样，SYSTEM\_TIME执行块级别的采样，所以采样不是完全随机的，而是服从于聚簇效果，特别是只选择少量行时。

SYSTEM\_TIME不支持 REPEATABLE子句。

这个模块被认为是“可信的”，也就是说，它可以由对当前数据库具有CREATE权限的非超级用户安装。

## H. 71. 1. 示例

这里是一个用SYSTEM\_TIME选择一个表采样的例子。首先安装扩展：

```
CREATE EXTENSION tsm_system_time;
```

然后就可以在SELECT命令中使用它，例如：

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_TIME(1000);
```

这个命令将返回在 1 秒（1000 毫秒）能读到的my\_table 采样。当然，如果 1 秒内就能读完整个表，所有的行都将被返回。

## H. 72. tsvector2

LightDB的扩展tsvector类型。它的实现旨在提供更好的压缩，并消除原始tsvector类型的1MB大小限制。

它可以作为原始tsvector的透明替代品，并支持其所有函数、运算符和索引类型。函数名称中包含tsvector的函数已更改为tsvector2。这些函数的完整列表在下面指定。

请参考LightDB 文档以获取有关tsvector的详细信息。它们也可以应用于tsvector2。

### H. 72. 1. tsvector2特定函数

- to\_tsvector2 (来自text, json, jsonb类型)
- array\_to\_tsvector2
- tsvector2\_to\_array
- tsvector2\_stat (应该替代ts\_stat)
- jsonb\_to\_tsvector2
- json\_to\_tsvector2
- tsvector2\_update\_trigger
- tsvector2\_update\_trigger\_column

### H. 72. 2. 可安全用于两种类型的常见函数

- strip
- unnest
- length
- setweight
- ts\_rank

- ts\_rank\_cd
- ts\_delete
- ts\_filter

## H. 73. unaccent

unaccent是一个文本搜索字典，它能从词位中移除重音（附加符号）。它是一个过滤词典，这表示它的输出总是会被传递给下一个字典（如果有），这和字典的通常行为不同。这允许为全文搜索做与重音无关的处理。

unaccent的当前实现不能被用作thesaurus字典的正规化字典。

这个模块被认为是“可信的”，也就是说，它可以由对当前数据库具有CREATE权限的非超级用户安装。

### H. 73. 1. 配置

unaccent字典接受下列选项：

- RULES是包含翻译规则列表的文件的基本名。这个文件必须被存储在\$SHAREDIR/tsearch\_data/（这里\$SHAREDIR表示LightDB安装的共享数据目录）中。它的名称必须以.rules（不包含在RULES参数中）结束。

规则文件具有下面的格式：

- 每一行表示一个由带有重音的字符和不带重音的字符构成的对。第一个字符将被翻译成第二个。例如：

```
À      A
Á      A
Â      A
Ã      A
Ä      A
Å      A
Æ      AE
```

两个字符必须由空格分隔，并且一行上的任何前导或尾随空白都将被忽略。

- 或者，如果一行只给出一个字符，则删除该字符的实例；这在用单独的字符表示重音的语言中是有用的。
- 实际上，每个“字符”可以是不包含空格的任何字符串，因此，除了去除变音符之外，unaccent字典也可以用于其他类型的字符串替换。
- 与其他LightDB文本搜索配置文件一样，规则文件必须以UTF-8编码方式存储。加载时，数据将自动转换为当前数据库的编码。任何含有不可翻译字符的行都将被忽略，因此规则文件可以包含当前编码中不适用的规则。

在unaccent.rules中可以找到一个更完整的例子，它可以直接用于大部分欧洲语言，当unaccent模块被安装时，它被安装在\$SHAREDIR/tsearch\_data/中。

### H. 73. 2. 用法

安装unaccent扩展会创建一个文本搜索模板unaccent和一个基于前者的字典unaccent。unaccent字典有默认的参数设置RULES='unaccent'，这会让该字典使用标准的unaccent.rules文件。如果希望修改该参数，可以

```
mydb=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='my_rules');
```

或者基于该模板创建新的字典。

要测试该字典，你可以尝试：

```
mydb=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----
 {Hotel}
(1 row)
```

这里是一个展示把unaccent字典插入到一个文本搜索配置的例子：

```
mydb=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
mydb=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
mydb=# select to_tsvector('fr','Hôtels de la Mer');
 to_tsvector
-----
 'hotel':1 'mer':4
(1 row)

mydb=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
 ?column?
-----
 t
(1 row)

mydb=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
 ts_headline
-----
 <b>Hôtel</b> de la Mer
(1 row)
```

### H. 73. 3. 函数

unaccent()函数从一个给定的字符串中移除重音（附加符号）。基本上，它是unaccent字典的一个包装器，但是它能在普通的文本搜索环境之外使用。

```
unaccent([dictionary regdictionary, ] string text) returns text
```

如果省略dictionary参数，则使用名为unaccent并且与unaccent()函数有相同模式的文本搜索词典。

例如：

```
SELECT unaccent('unaccent', 'Hôtel');
SELECT unaccent('Hôtel');
```

## H. 74. uuid-osp

uuid-osspp模块提供函数使用几种标准算法之一产生通用唯一标识符（UUID）。还提供产生某些特殊 UUID 常量的函数。此模块仅适用于核心LightDB中提供的以外的特殊要求。有关生成UUID的内置方法，请参见Section 9.14

这个模块被认为是“可信的”，也就是说，它可以由对当前数据库具有CREATE权限的非超级用户安装。

## H.74.1. uuid-osspp 函数

Table H.18展示了可用来产生 UUID 的函数。相关标准 ITU-T Rec. X.667、ISO/IEC 9834-8:2005 以及RFC 4122<sup>18</sup>指定了四种用于产生 UUID 的算法，分别用版本号 1、3、4、5 标识（没有版本 2 的算法）。这些算法中的每一个都适合于不同的应用集合。

Table H.189. 用于 UUID 产生的函数

函数	简述
uuid_generate_v1 () → uuid	产生一个版本 1 的 UUID。这涉及到计算机的 MAC 地址和一个时间戳。注意这种 UUID 会泄露产生该标识符的计算机标识以及产生的时间，因此它不适合某些对安全性很敏感的应用。
uuid_generate_v1mc () → uuid	这个函数产生一个版本 1 的 UUID，但是使用一个随机广播 MAC 地址而不是该计算机真实的 MAC 地址。
uuid_generate_v3 ( namespace uuid, name text ) → uuid	使用指定的输入名称在给定的名字空间中产生一个版本 3 的 UUID。该名字空间应该是由uuid_ns_*()函数（如Table H.190所示）产生的特殊常量之一（理论上它可以是任意 UUID）。名称是选择的名称空间中的一个标识符。 例如：  <pre>SELECT uuid_generate_v3(uuid_ns_url(), 'www.hs.net/lightdb');</pre> 名称参数将使用 MD5 进行哈希，因此从产生的 UUID 中得不到明文。采用这种方法的 UUID 生成没有随机性并且不涉及依赖于环境的元素，因此是可以重现的。
uuid_generate_v4 () → uuid	产生一个版本 4 的 UUID，它完全从随机数产生。
sys_guid () → uuid	产生一个版本 4 的 UUID，它完全从随机数产生。和 uuid_generate_v4 等价。
uuid_generate_v5 ( namespace uuid, name text ) → uuid	产生一个版本 5 的 UUID，它和版本 3 的 UUID 相似，但是采用的是 SHA-1 作为哈希方法。版本 5 比版本 3 更好，因为 SHA-1 被认为比 MD5 更安全。

Table H.190. 返回 UUID 常量的函数

函数	简述
uuid_nil () → uuid	返回一个“nil” UUID 常量，它不作为一个真正的 UUID 出现。
uuid_ns_dns () → uuid	返回为 UUID 指定 DNS 名字空间的常量。
uuid_ns_url () → uuid	返回为 UUID 指定 URL 名字空间的常量。

<sup>18</sup> <https://tools.ietf.org/html/rfc4122>

函数	简述
<code>uuid_ns_oid () → uuid</code>	返回为 UUID 指定 ISO 对象标识符 (OID) 名字空间的常量 (这属于 ASN.1 OID, 它与LightDB使用的 OID 无关)。
<code>uuid_ns_x500 () → uuid</code>	返回为 UUID 指定 X.500 可识别名 (DN) 名字空间的常量。

## H. 74. 2. 作者

Peter Eisentraut <peter\_e@gmx.net>

## H. 75. wal2sql

wal2sql 是一个逻辑解码输出扩展。这意味着扩展可以访问 INSERT 和 UPDATE 生成的元组。此外, 根据配置的复制标识符, 可以访问 UPDATE/DELETE 旧行版本。更改可以通过流协议 (逻辑复制槽) 或特殊的 SQL API 进行消费。

格式版本 1 会为每个事务生成一个 JSON 对象。该 JSON 对象中包含所有新旧元组。此外, 还有选项可以包括事务时间戳、模式限定符、数据类型和事务 ID 等属性。

格式版本 2 会为每个元组生成一个 JSON 对象。可以选择为事务的开始和结束生成 JSON 对象。此外, 还有各种选项可用于包括属性。

### H. 75. 1. 配置

您需要在 `lightdb.conf` 中设置至少以下参数

```
# lightdb.conf
wal_level = logical
```

### H. 75. 2. 配置参数

`include-transaction` (boolean)

发出记录, 表示每个事务的开始和结束。默认值为 `true`。

`include-xids` (boolean)

将 XID 添加到每个更改集。默认值为 `false`。

`include-timestamp` (boolean)

将时间戳添加到每个更改集。默认值为 `false`。

`include-pk` (boolean)

将主键信息添加为 `pk`。包括列名和数据类型。默认值为 `false`。

`include-origin` (boolean)

添加数据的来源。默认值为 `false`。

`include-schemas` (boolean)

将模式添加到每个更改中。默认值为 `true`。

`include-types` (boolean)

将类型添加到每个更改中。默认值为 `true`。

`include-type-oids` (boolean)

添加类型 OID。默认值为 `false`。

`include-typmod` (boolean)

对于有修饰符的类型添加修饰符（例如 `varchar(20)` 而不是 `varchar`）。默认值为 `true`。

`include-domain-data-type` (boolean)

将域名替换为底层数据类型。默认值为 `false`。

`include-column-positions` (boolean)

添加列位置 (`pg_attribute.attnum`)。默认值为 `false`。

`include-not-null` (boolean)

将非空信息添加为列选项。默认值为 `false`。

`include-default` (boolean)

添加默认表达式。默认值为 `false`。

`pretty-print` (boolean)

向 JSON 结构添加空格和缩进。默认值为 `false`。

`write-in-chunks` (boolean)

每次更改后写入，而不是每个更改集后写入。仅在格式版本为 1 时使用。默认值为 `false`。

`include-lsn` (boolean)

在每个更改集中添加 `nextlsn`。默认值为 `false`。

`actions` (boolean)

定义将发送哪些操作。默认值为所有操作（插入、更新、删除和截断）。但是，如果您使用格式版本 1，则禁用截断（向后兼容）。

`filter-origins` (boolean)

排除来自指定来源的更改。默认为空，意味着没有来源将被过滤。它是一个逗号分隔的值。

`filter-tables` (boolean)

排除来自指定表的行。默认为空，意味着没有表将被过滤。它是一个逗号分隔的值。表应该被模式限定。 `*.foo` 表示所有模式中的表 `foo`，`bar.*` 表示模式 `bar` 中的所有



表。特殊字符（空格、单引号、逗号、点、星号）必须用反斜杠转义。模式和表区分大小写。表“public”。“Foo bar”应该被指定为 public.Foo\ bar。

add-tables (boolean)

仅包括来自指定表的行。默认是来自所有模式的所有表。它具有与 filter-tables 相同的规则。

filter-msg-prefixes (boolean)

如果前缀在列表中，则排除消息。默认为空，意味着没有消息将被过滤。它是一个逗号分隔的值。

add-msg-prefixes (boolean)

只包括前缀在列表中的消息。默认为所有前缀。这是一个逗号分隔的值。wal2sql在此参数之前应用filter-msg-prefixes。

format-version (integer)

定义要使用的格式。默认为1。

### H. 75. 3. 参数示例

从wal2sql插件中获取更改（JSON对象）有两种方法：通过 ltdts\_recvlogical 调用函数或使用SQL。

#### H. 75. 3. 1. ltdts\_recvlogical

除上述配置外，还需要配置一个复制连接来使用 ltdts\_recvlogical。

在 lt\_hba.conf 中添加复制连接规则。

```
local    mydatabase    myuser                                trust
```

另外，在 lightdb.conf 中设置 max\_wal\_senders:

```
max_wal_senders = 1
```

如果更改了 max\_wal\_senders，需要重新启动。现在可以尝试使用 wal2sql。在一个终端中执行以下命令：

```
# create a logical replication slot name as test_oracle_slot
ltdts_recvlogical -d postgres --slot=test_oracle_slot --create-slot
-P wal2sql

# delete a logical replication slot
ltdts_recvlogical -d postgres --slot=test_oracle_slot --drop-slot

# print output in the first terminal
ltdts_recvlogical -d postgres --slot test_oracle_slot --start -o
pretty-print=1 -o add-msg-prefixes=wal2sql -f -
```

```
# forward output to oracle.
ltdts_recvlogical --tnsname=TEST --oracle-username=scott --oracle-
password=tiger --start --slot=test_oracle_slot -o include-types=false
-o include-type-oids=true -d postgres -h 10.20.30.193 -p 9999 -f
debug.dat
```

### H. 75. 3. 2. SQL

保存以下代码并执行：

```
CREATE TABLE table3_with_pk (a SERIAL, b VARCHAR(30), c TIMESTAMP
NOT NULL, PRIMARY KEY(a, c));
CREATE TABLE table3_without_pk (a SERIAL, b NUMERIC(5,2), c TEXT);

SELECT 'init' FROM pg_create_logical_replication_slot('test_slot',
'wal2sql');

BEGIN;
INSERT INTO table3_with_pk (b, c) VALUES('Backup and Restore',
now());
INSERT INTO table3_with_pk (b, c) VALUES('Tuning', now());
INSERT INTO table3_with_pk (b, c) VALUES('Replication', now());
SELECT pg_logical_emit_message(true, 'wal2sql', 'this message will
be delivered');
DELETE FROM table3_with_pk WHERE a < 3;
SELECT pg_logical_emit_message(false, 'wal2sql', 'this non-
transactional message will be delivered even if you rollback the transaction');

INSERT INTO table3_without_pk (b, c) VALUES(2.34, 'Tapir');
-- it is not added to stream because there isn't a pk or a replica
identity
UPDATE table3_without_pk SET c = 'Anta' WHERE c = 'Tapir';
COMMIT;

SELECT data FROM pg_logical_slot_get_changes('test_slot', NULL,
NULL, 'format-version', '2', 'add-msg-prefixes', 'wal2sql');
SELECT 'stop' FROM pg_drop_replication_slot('test_slot');

DROP TABLE table3_with_pk;
DROP TABLE table3_without_pk;
```

### H. 75. 4. 注意

1. wal2sql 不支持在 龙芯架构<sup>19</sup> 上运行。
2. 在 高可用性 中，请启动 ltdts\_logicalrepl\_copier.sh 将 slot 从主库复制到备份库。
3. 在 高可用性 中，当发生故障转移时，请重新启动新的主库 LightDB 以使 slot 生效。

## H. 76. xml2

xml2模块提供 XPath 查询和 XSLT 功能。

<sup>19</sup> <https://www.loongson.cn/>

## H. 76. 1. 废弃公告

从LightDB 8.3 开始，在核心服务器中就已经有基于 SQL/XML 标准的 XML 相关功能。那些功能覆盖了 XML 语法检查和 XPath 查询，这些本模块也能做，但是其 API 并不是完全兼容。这个模块已经有计划将从 LightDB 的一个未来版本中移除，因此我们鼓励你尝试转换你的应用。如果你发现这个模块的某些功能在更新的 API 中没有合适的形式相对应，请向<pgsql-hackers@lists.postgresql.org>表达你的问题，这样该缺点会被进行改进。

## H. 76. 2. 函数的描述

Table H.19展示了这个模块提供的函数。这些函数提供了直接的 XML 解析和 XPath 查询。

Table H.191.xml2 函数

函数	描述
xml_valid ( document text ) → boolean	解析给定的文档，如果文档是格式良好的 XML，则返回 true。（注意，这是标准 LightDB函数xml_is_well_formed()的别名。名称xml_valid()在技术上是不正确的，因为在 XML 中有效性和结构良好性具有不同的含义。）
xpath_string ( document text, query text ) → text	在提供的文档上评估 XPath 查询，并将结果转换为text。
xpath_number ( document text, query text ) → real	在提供的文档上评估 XPath 查询，并将结果转换为real。
xpath_bool ( document text, query text ) → boolean	在提供的文档上评估 XPath 查询，并将结果转换为boolean。
xpath_nodeset ( document text, query text, toptag text, itemtag text ) → text	这个函数在文档上计算查询并且把结果包装在 XML 标签中。如果结果是多值的，输出看起来是这样：  <pre>&lt;toptag&gt; &lt;itemtag&gt;Value 1 which could be an XML fragment&lt;/itemtag&gt; &lt;itemtag&gt;Value 2....&lt;/itemtag&gt; &lt;/toptag&gt;</pre> 如果toptag或者itemtag是一个空字符串，相关的标签会被忽略。
xpath_nodeset ( document text, query text, itemtag text ) → text	类似于xpath_nodeset(document, query, toptag, itemtag)但结果省略了toptag。
xpath_nodeset ( document text, query text ) → text	与xpath_nodeset(document, query, toptag, itemtag)相似但是结果忽略两种标签。
xpath_list ( document text, query text, separator text ) → text	评估文档上的查询并返回由指定分隔符分隔的多个值，例如Value 1, Value 2, Value 3, 如果separator 是,。
xpath_list ( document text, query text ) → text	这是上述函数的包装器，使用,作为分隔符。

## H. 76. 3. xpath\_table

xpath\_table(text key, text document, text relation, text xpaths, text criteria)  
returns setof record

xpath\_table是一个表函数，它在一组文档中的每一个上计算一组 XPath 查询，并且将结果作为一个表返回。来自原始文档表的主键域被返回为结果的第一列，这样结果集可以被用于连接。其参数在Table H.192中描述。

Table H.192. xpath\_table 参数

参数	描述
key	“key” 域的名称 — 这只是被用作输出表中第一列的域，即它标识每一个输出行是来自于哪个记录（见下面有关多个值的注解）
document	包含 XML 文档的域的名称
relation	包含文档的表或视图的名称
xpaths	一个或多个 XPath 表达式，用 分隔
criteria	WHERE 子句的内容。这不能被忽略，因此如果你想要处理关系中的所有行，可以使用true或1=1

这些参数（除了 XPath 字符串）只是会被替换到一个纯粹的 SQL SELECT 语句中，因此你有一些灵活性 — 该语句是

```
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

因此那些参数可以是那些特定位置上合法的任何东西。来自于这个 SELECT 的结果需要返回正好两列（它确实会这样，除非你尝试为键或文档列出多个域）。注意这种简单方法要求你验证任何用户提供的值来避免 SQL 注入攻击。

该函数必须被使用在一个FROM表达式中，并带有一个AS子句来指定输出列，例如

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date_entered > ''2003-01-01'' ')
AS t(article_id integer, author text, page_count integer, title text);
```

AS子句定义了输出表中列的名称和类型。第一个是“key”域并且剩下的对应于 XPath 查询。如果 XPath 查询比结果列多，额外的查询将被忽略。如果结果列比 XPath 查询多，额外的列将是 NULL。

注意这个例子定义page\_count结果列为一个整数。该函数在内部处理字符串表达，因此当你在输出中想要一个整数时，它将采用 XPath 结果的字符串表达并且使用 LightDB 输入函数来把它转换成一个整数（或者AS子句要求的任何类型）。如果它无法做到这一点将会导致一个错误 — 例如结果是空 — 因此如果你认为你的数据有任何问题，你可能希望坚持用text作为列类型。

调用的SELECT语句不必只是 SELECT \* — 它可以用名称引用输出列或者将它们连接到其他表。该函数会产生一个虚拟表，你可以在其上执行任何所需的操作（例如聚集、连接、排序等）。因此我们也可以有：

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                '/article/title|/article/author/@id',
                'xpath_string(article_xml, '/article/@date') > ''2003-03-20''
                ')
AS t(article_id integer, title text, author_id integer),
```

```
tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

作为一个更复杂的例子。当然，为了便利你也可以把所有这些包装在一个视图中。

### H. 76. 3. 1. 多值结果

xpath\_table函数假定每一个 XPath 查询的结果可能是多值的，因此该函数返回的行数可能与输入文档的数目不同。被返回的第一行包含来自每一个查询的第一个结果，第二行则是来自每一个查询的第二个结果。如果其中一个查询的值比其他查询少，则会为它返回空值。

在某些情况下，一个用户将知道一个给定的 XPath 查询将只返回一个单一结果（可能是一个唯一文档标识符）——如果和一个返回多值的 XPath 查询一起使用，单值结果将只出现在结果的第一行中。对于这种情况的解决方案是使用键域作为针对一个更简单 XPath 查询的连接的一部分。一个例子：

```
CREATE TABLE test (
  id int PRIMARY KEY,
  xml text
);

INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id', 'xml', 'test',
             '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
             'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int, val2 int,
      val3 int)
WHERE id = 1 ORDER BY doc_num, line_num
```

id	doc_num	line_num	val1	val2	val3
1	C1	L1	1	2	3
1		L2	11	22	33

要在每一行上得到doc\_num，解决方案是使用xpath\_table的两个调用并且连接结果：

```
SELECT t.*, i.doc_num FROM
  xpath_table('id', 'xml', 'test',
             '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
             'true')
  AS t(id int, line_num varchar(10), val1 int, val2 int, val3 int),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
  AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;
```

id	line_num	val1	val2	val3	doc_num
1	L1	1	2	3	C1
1	L2	11	22	33	

1	L1	1	2	3	C1
1	L2	11	22	33	C1

(2 rows)

## H. 76. 4. XSLT 函数

如果安装了 `libxslt`，那么可以使用下列函数：

### H. 76. 4. 1. `xslt_process`

`xslt_process(text document, text stylesheet, text paramlist)` returns text

这个函数将 XSL 样式表应用于文档并且返回转换过的结果。`paramlist`是一个被用在转换中的参数赋值列表，以=1,b=2的形式指定。注意参数解析是非常天真的：参数值不能包含逗号！

还有一个双参数版本的`xslt_process`，它不会向转换传递任何参数。

## H. 76. 5. 作者

John Gray <jgray@azuli.co.uk>

这个模块的开发由 Torchbox Ltd. ([www.torchbox.com](http://www.torchbox.com)) 赞助。它使用了和 LightDB 相同的 BSD 许可证。

## H. 77. zhparser

`zhparser` 是 LightDB 的一个扩展，用于中文全文搜索（普通话）。它基于 Simple Chinese Word Segmentation(SCWS) 实现了一个中文语言解析器。

### H. 77. 1. 配置

这些选项用于控制字典加载行为和分词行为。这些选项都不是必需的，并且默认情况下为 `false`（也就是说，如果这些选项在配置文件中未设置，则 `zhparser` 的行为与将以下选项设置为 `false` 相同）。

忽略所有特殊符号（例如标点符号）：`zhparser.punctuation_ignore = f`

空闲词自动分组为两个词语的分词：`zhparser.seg_with_duality = f`

将所有字典加载到内存中：`zhparser.dict_in_memory = f`

短词合并：`zhparser.multi_short = f`

哈希二进制合并：`zhparser.multi_duality = f`

重要词合并：`zhparser.multi_zmain = f`

设置 `zhparser` 的非停用词：有关更多信息，请参见 `lightdb_tsearch_non_stopwords`。

除了 `zhparser` 随附的字典之外，您还可以添加自定义字典，它们优先于本地字典。文件必须存储在 `share/postgresql/tsearch_data` 目录中。`zhparser` 根据文件扩展名确定字典格式。`.txt` 扩展名表示字典采用文本格式，`.xdb` 扩展名表示字典采用 XDB 格式。多个文件使用逗号分隔，并且字典按优先级从高到低分类，例如：

```
zhparser.extra_dicts = 'dict_extra.txt,mydict.xdb'
```

注意: zhparser.extra\_dicts 和 zhparser.dict\_in\_memory 选项需要在后端启动之前设置 (可以在配置文件中更改, 并重新加载, 此后新的连接才会生效), 其他选项可以在会话中随时设置以生效。

## H. 77. 2. 示例

```
-- make test configuration using parser

CREATE TEXT SEARCH CONFIGURATION testzhcfg (PARSER = zhparser);

-- add token mapping

ALTER TEXT SEARCH CONFIGURATION testzhcfg ADD MAPPING FOR
n, v, a, i, e, l WITH simple;

-- ts_parse

SELECT * FROM ts_parse('zhparser', 'hello world! 2010年保障房建设在
全国范围内获全面启动, 从中央到地方纷纷加大 了保障房的建设和投入力度 。2011年, 保
障房进入了更大规模的建设阶段。住房城乡建设部党组书记、部长姜伟新去年底在全国住房
城乡建设工作会议上表示, 要继续推进保障性安居工程建设。');

-- test to_tsvector

SELECT to_tsvector('testzhcfg', '“今年保障房新开工数量虽然有所下调,
但实际的年度在建规模以及竣工规模会超以往年份, 相对应的对资金的需求也会创历>史记
录。”陈国强说。在他看来, 与2011年相比, 2012年的保障房建设在资金配套上的压力将更
为严峻。');

-- test to_tsquery

SELECT to_tsquery('testzhcfg', '保障房资金压力');

-- test lightdb_tsearch_non_stopwords

SET lightdb_tsearch_non_stopwords = '1';
ERROR:  invalid value for parameter "lightdb_tsearch_non_stopwords":
"1"
DETAIL:  lightdb_tsearch_non_stopwords not support '1' as non
stopword.

SET lightdb_tsearch_non_stopwords = '@:_-';
SELECT to_tsvector('testzhcfg', '浙江省杭州市this is from
lt_hs_tab table 滨江区江南大道logged at 2020-12-21 恒生电子 00:22:32,
username=zjh&pwd=balabala');

to_tsvector

-----
'00:22:32':16 '2020-12-21':13 'at':12 'from':5 'is':4 'logged':11
'lt_hs_tab':6 'pwd=balabala':18 'table':7 'this':3 'username=zjh':17 '大道':10
'恒生':14 '杭州市':2 '江南':9 '浙江省':1 '滨江区':8 '电子':15
(1 row)
```

```
SET lightdb_tsearch_non_stopwords = '';
SELECT to_tsvector('testzhcfg', 'this is from lt_hs_tab table logged
at 2020-12-21 00:22:32, username=zjh&pwd=balabala');
```

```
to_tsvector
```

```
-----
'00':11 '12':9 '2020':8 '21':10 '22':12 '32':13 'at':7 'balabala':17
'from':3 'is':2 'logged':6 'lt_hs_tab':4 'pwd':16 'table':5 'this':1
'username':14 'zjh':15
(1 row)
```

### H. 77. 3. 自定义字典

**\*\* TXT 字典目前与 CLI/SCws\_gen\_dict 用于文本字典兼容 \*\***

- 每行一个词条，以 # 或分号开头的注释将被忽略和跳过。

每行由 4 个字段组成，依次为“单词”（由汉字或少于 3 个字母组成）、“TF”、“IDF”和“词性”。字段由空格或制表符分隔。字段数量没有限制，并且可以对齐以美化自己。

除了“单词”以外，其他字段都可以忽略。如果省略，则 TF 和 IDF 默认为 1.0，词性为“@”。

TXT库的动态加载（内部监视文件修改时间自动转换为XDB存储在系统临时目录中），建议TXT字典不要太大。

要删除单词，请将其词性设置为“! ”，这样单词就被设置为无效，即使它存在于其他核心库中。

注意： 1. 自定义字典的格式可以是TXT或二进制XDB格式。 XDB格式更高效，适用于大型字典。您可以使用内置的SCWS工具scws-Gen-dict将文本词典转换为XDB格式。 2. 默认的zhparser词典是简体中文。

### H. 77. 4. 自定义字典 2.1

**\*\* 自定义字典 2.1 增加了自定义字典的可移植性，并与 1.0 提供的功能兼容 \*\***

自定义字典需要超级用户权限。自定义字典是数据库级别的（而非实例级别的）。每个数据库都有自己的自定义分词，并存储在数据目录 base/ 数据库ID 下（版本 2.0 存储在 share/tsearch\_data 下）。

```
test=# SELECT * FROM ts_parse('zhparser', '保障房资金压力');
 tokid | token
-----+-----
    118 | 保障
    110 | 房
    110 | 资金
    110 | 压力

test=# insert into zhparser.zhprs_custom_word values('资金压力');
--Delete the word insert into zhprs_custom_word(word, attr)
values('word', '!');
--\d zhprs_custom_word view its table structure, supported TD, IDF
```



```

test=# select sync_zhprs_custom_word();
 sync_zhprs_custom_word
-----
(1 row)

test=# \q --sync then re-establish the connection
[lzzhang@lzzhang-pc bin]$ ./psql -U lzzhang -d test -p 1600
test=# SELECT * FROM ts_parse('zhparser', '保障房资金压力');
 tokid | token
-----+-----
    118 | 保障
    110 | 房
    120 | 资金压力

```

## H. 78. 1t\_t3sdk

这个LightDB扩展实现了JRES的T3请求

### H. 78. 1. 安装

要安装t3sdk，需要使用CRES的C客户端库，默认版本是CRES2.0.V202302.00.000。这个库可以从恒生下载

下载CRES的SDK并解压，目录如下：

```

acm/
dev/
  client_demo/T3SDK/c++/linux.x64/lib/libt3sdk.so
  client_demo/T3SDK_C/linux.x64/lib/libt3sdk_c.so
docker/
instance/
tools/

```

CRES SDK的libt3sdk.so and libt3sdk\_c.so 必须添加到 LD\_LIBRARY\_PATH.

如果你首次安装，使用 CREATE EXTENSION t3sdk.

如果你已经安装了一个之前的版本，并且仅仅想要升级则使用 ALTER EXTENSION t3sdk UPDATE.

### H. 78. 2. 概述

每一个T3调用都是由VARCHAR请求列表和t3sdk\_response应答组成.

每次向T3的服务请求时，后端进程都会保持连接，直到该进程退出。键是服务器地址和端口，值是连接句柄。

List of functions		
Schema	Name	Result data type
		Argument data types
		Type

```

-----+-----+-----
+-----
public | t3sdk_send | t3sdk_response | t_address character varying,
groups character varying, service character varying, version character varying,
function_id character varying, function_args character varying | func

Composite type "public.t3sdk_response"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----
content | character varying | | |

```

### H. 78. 3. 函数

- t3sdk\_send(t\_address VARCHAR, groups VARCHAR, service VARCHAR, version VARCHAR, function\_id VARCHAR, function\_args VARCHAR) returns t3sdk\_response

默认情况下，发送请求的超时时间为60秒。如果需要不同的超时时间，可以使用毫秒设置以下GUC变量：

```
t3sdk.t3_send_timeout_msec = 60000
```

默认情况下，为请求的接收设置了60秒的超时时间。如果需要不同的超时时间，可以使用毫秒设置以下GUC变量：

```
t3sdk.t3_rcv_timeout_msec = 60000
```

如果需要不同T3的许可证，可以使用以下GUC变量进行设置：

```
t3sdk.t3_license_no = 'HS-HUNDSUN001-
FBASE2-0000-4ePWxzscBVtY9ZKdGDKhSyk2'
```

### H. 78. 4. 例子

```
-- 发送t3请求
select * from t3sdk_send('10.20.30.193:18082', 'g', 'repo-app',
'v', 'getExpressInfoNdb', '');
```

content

```
-----
{"result":
 [{"express_id": "a", "order_id": "1", "express_no": "123456789", "company": "测试公司1", "create_datetime": "20230407010201", "status": "0"},
 {"express_id": "b", "order_id": "2", "express_no": "123456789", "company": "测试公司2", "create_datetime": "20230407010201", "status": "0"}]}
(1 row)
```

## H. 79. lt\_http

这个LightDB扩展实现了http请求。

### H. 79. 1. 安装

如果你首次安装, 使用 `CREATE EXTENSION http`.

如果你已经安装了一个之前的版本, 并且仅仅想要升级则使用 `ALTER EXTENSION http UPDATE`.

### H. 79. 2. 概述

每一个HTTP调用都是由一个http\_request请求和一个 http\_response应答组成.

Composite type "public.http\_request"

Column	Type	Modifiers
method	http_method	
uri	character varying	
headers	http_header[]	
content_type	character varying	
content	character varying	

Composite type "public.http\_response"

Column	Type	Modifiers
status	integer	
content_type	character varying	
headers	http_header[]	
content	character varying	

`http_get()`、`http_post()`、`http_put()`、`http_delete()`和`http_head()`这些实用函数都是围绕着主函数`http(http_request)`的封装, 该函数会返回`http_response`。

请求和响应的headers字段是类型为`http_header`的LightDB数组, 而`http_header`只是一个简单的元组。

Composite type "public.http\_header"

Column	Type	Modifiers
field	character varying	
value	character varying	

如示例中所见, 您可以使用LightDB数组上的`unnest()`函数将`http_header`元组的数组展开为结果集。 从那里您可以选择您感兴趣的特定标头。

### H. 79. 3. 函数

- `http_header(field VARCHAR, value VARCHAR)` 返回 `http_header`
- `http(request http_request)` 返回 `http_response`

- `http_get(uri VARCHAR)` 返回 `http_response`
- `http_get(uri VARCHAR, data JSONB)` 返回 `http_response`
- `http_post(uri VARCHAR, content VARCHAR, content_type VARCHAR)` 返回 `http_response`
- `http_post(uri VARCHAR, data JSONB)` 返回 `http_response`
- `http_put(uri VARCHAR, content VARCHAR, content_type VARCHAR)` 返回 `http_response`
- `http_patch(uri VARCHAR, content VARCHAR, content_type VARCHAR)` 返回 `http_response`
- `http_delete(uri VARCHAR, content VARCHAR, content_type VARCHAR)` 返回 `http_response`
- `http_head(uri VARCHAR)` 返回 `http_response`
- `http_set_curlopt(curlopt VARCHAR, value varchar)` 返回 `boolean`
- `http_reset_curlopt()` 返回 `boolean`
- `http_list_curlopt()` 返回 `setof(curlopt text, value text)`
- `urlencode(string VARCHAR)` 返回 `text`
- `urlencode(data JSONB)` 返回 `text`

## H. 79. 4. CURL选项

可以使用`http_set_curlopt(curlopt VARCHAR, value varchar)`函数来设置CURL选项<sup>20</sup>。

- `CURLOPT_DNS_SERVERS`<sup>21</sup>
- `CURLOPT_PROXY`<sup>22</sup>
- `CURLOPT_PRE_PROXY`<sup>23</sup>
- `CURLOPT_PROXYPORT`<sup>24</sup>
- `CURLOPT_PROXYUSERPWD`<sup>25</sup>
- `CURLOPT_PROXYUSERNAME`<sup>26</sup>
- `CURLOPT_PROXYPASSWORD`<sup>27</sup>
- `CURLOPT_PROXY_TLSAUTH_USERNAME`<sup>28</sup>
- `CURLOPT_PROXY_TLSAUTH_PASSWORD`<sup>29</sup>
- `CURLOPT_PROXY_TLSAUTH_TYPE`<sup>30</sup>
- `CURLOPT_TLSAUTH_USERNAME`<sup>31</sup>

<sup>20</sup> [https://curl.haxx.se/libcurl/c/curl\\_easy\\_setopt.html](https://curl.haxx.se/libcurl/c/curl_easy_setopt.html)

<sup>21</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_DNS\\_SERVERS.html](https://curl.haxx.se/libcurl/c/CURLOPT_DNS_SERVERS.html)

<sup>22</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_PROXY.html](https://curl.haxx.se/libcurl/c/CURLOPT_PROXY.html)

<sup>23</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_PRE\\_PROXY.html](https://curl.haxx.se/libcurl/c/CURLOPT_PRE_PROXY.html)

<sup>24</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_PROXYPORT.html](https://curl.haxx.se/libcurl/c/CURLOPT_PROXYPORT.html)

<sup>25</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_PROXYUSERPWD.html](https://curl.haxx.se/libcurl/c/CURLOPT_PROXYUSERPWD.html)

<sup>26</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_PROXYUSERNAME.html](https://curl.haxx.se/libcurl/c/CURLOPT_PROXYUSERNAME.html)

<sup>27</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_PROXYPASSWORD.html](https://curl.haxx.se/libcurl/c/CURLOPT_PROXYPASSWORD.html)

<sup>28</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_PROXY\\_TLSAUTH\\_USERNAME.html](https://curl.haxx.se/libcurl/c/CURLOPT_PROXY_TLSAUTH_USERNAME.html)

<sup>29</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_PROXY\\_TLSAUTH\\_PASSWORD.html](https://curl.haxx.se/libcurl/c/CURLOPT_PROXY_TLSAUTH_PASSWORD.html)

<sup>30</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_PROXY\\_TLSAUTH\\_TYPE.html](https://curl.haxx.se/libcurl/c/CURLOPT_PROXY_TLSAUTH_TYPE.html)

<sup>31</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_TLSAUTH\\_USERNAME.html](https://curl.haxx.se/libcurl/c/CURLOPT_TLSAUTH_USERNAME.html)

- `CURLOPT_TLSAUTH_PASSWORD`<sup>32</sup>
- `CURLOPT_TLSAUTH_TYPE`<sup>33</sup>
- `CURLOPT_SSL_VERIFYHOST`<sup>34</sup>
- `CURLOPT_SSL_VERIFYPEER`<sup>35</sup>
- `CURLOPT_SSLCERT`<sup>36</sup>
- `CURLOPT_SSLKEY`<sup>37</sup>
- `CURLOPT_SSLCERTTYPE`<sup>38</sup>
- `CURLOPT_CAINFO`<sup>39</sup>
- `CURLOPT_TIMEOUT`<sup>40</sup>
- `CURLOPT_TIMEOUT_MS`<sup>41</sup>
- `CURLOPT_TCP_KEEPALIVE`<sup>42</sup>
- `CURLOPT_TCP_KEEPIDLE`<sup>43</sup>
- `CURLOPT_CONNECTTIMEOUT`<sup>44</sup>
- `CURLOPT_USERAGENT`<sup>45</sup>

## H. 79. 5. 保持连接和超时

推荐使用上面描述的`http_reset_curlopt()`方法。下面的全局变量将被弃用，并随着时间的推移移除。

默认情况下，每个请求都使用一个新连接，并确保在请求完成后关闭连接。这种行为减少了扩展在长时间运行过程中消耗系统资源（套接字）的机会。

高性能的应用程序可能希望启用保持连接和连接持久性，以减少延迟并提高吞吐量。以下GUC变量改变http扩展的行为，以尽可能维持连接：

```
http.keepalive = 'on'
```

默认情况下，请求的完成超时设置为60秒。如果需要不同的超时，可以使用以下GUC变量以毫秒为单位设置：

```
http.timeout_msec = 60000
```

---

<sup>32</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_TLSAUTH\\_PASSWORD.html](https://curl.haxx.se/libcurl/c/CURLOPT_TLSAUTH_PASSWORD.html)

<sup>33</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_TLSAUTH\\_TYPE.html](https://curl.haxx.se/libcurl/c/CURLOPT_TLSAUTH_TYPE.html)

<sup>34</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_SSL\\_VERIFYHOST.html](https://curl.haxx.se/libcurl/c/CURLOPT_SSL_VERIFYHOST.html)

<sup>35</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_SSL\\_VERIFYPEER.html](https://curl.haxx.se/libcurl/c/CURLOPT_SSL_VERIFYPEER.html)

<sup>36</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_SSLCERT.html](https://curl.haxx.se/libcurl/c/CURLOPT_SSLCERT.html)

<sup>37</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_SSLKEY.html](https://curl.haxx.se/libcurl/c/CURLOPT_SSLKEY.html)

<sup>38</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_SSLCERTTYPE.html](https://curl.haxx.se/libcurl/c/CURLOPT_SSLCERTTYPE.html)

<sup>39</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_CAINFO.html](https://curl.haxx.se/libcurl/c/CURLOPT_CAINFO.html)

<sup>40</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_TIMEOUT.html](https://curl.haxx.se/libcurl/c/CURLOPT_TIMEOUT.html)

<sup>41</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_TIMEOUT\\_MS.html](https://curl.haxx.se/libcurl/c/CURLOPT_TIMEOUT_MS.html)

<sup>42</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_TCP\\_KEEPALIVE.html](https://curl.haxx.se/libcurl/c/CURLOPT_TCP_KEEPALIVE.html)

<sup>43</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_TCP\\_KEEPIDLE.html](https://curl.haxx.se/libcurl/c/CURLOPT_TCP_KEEPIDLE.html)

<sup>44</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_CONNECTTIMEOUT.html](https://curl.haxx.se/libcurl/c/CURLOPT_CONNECTTIMEOUT.html)

<sup>45</sup> [https://curl.haxx.se/libcurl/c/CURLOPT\\_USERAGENT.html](https://curl.haxx.se/libcurl/c/CURLOPT_USERAGENT.html)

## H. 79. 6. 例子

-- 对字符串进行URL编码。

```
SELECT urlencode('my special string's and things?');
```

```

                                urlencode
                                -----
my+special+string%27s+and+things%3F
(1 row)

```

-- 对JSON关联数组进行URL编码。

```
SELECT urlencode(jsonb_build_object('name', 'Colin and James'));
```

```

                                urlencode
                                -----
name=Colin+and+James
(1 row)

```

-- 运行GET请求，查看内容。

```
SELECT content
FROM http_get('http://httpbin.org/ip');
```

```

                                content
                                -----
{"origin": "24. 69. 186. 43"}
(1 row)

```

-- 运行带有Authorization标头的GET请求。

```
SELECT content::json->'headers' ->>'Authorization'
FROM http((
    'GET',
    'http://httpbin.org/headers',
    ARRAY[http_header('Authorization', 'Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9')],
    NULL,
    NULL
)::http_request);
```

```

                                content
                                -----
Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
(1 row)

```

-- 从http\_response对象中读取status和content字段。

```
SELECT status, content_type
FROM http_get('http://httpbin.org/');
```

status	content_type
200	text/html; charset=utf-8

(1 row)

-- 显示http\_response对象中的所有http\_header。

```
SELECT (unnest(headers)).*
FROM http_get('http://httpbin.org/');
```

field	value
Connection	close
Server	meinheld/0.6.1
Date	Tue, 09 Jan 2018 18:40:30 GMT
Content-Type	text/html; charset=utf-8
Content-Length	13011
Access-Control-Allow-Origin	*
Access-Control-Allow-Credentials	true
X-Powered-By	Flask
X-Processed-Time	0.0208520889282
Via	1.1 vegur

-- 使用PUT命令将一个简单的文本文件发送到服务器。

```
SELECT status, content_type, content::json->>'data' AS data
FROM http_put('http://httpbin.org/put', 'some text', 'text/plain');
```

status	content_type	data
200	application/json	some text

-- 使用PATCH命令将一个简单的JSON文件发送到服务器。

```
SELECT status, content_type, content::json->>'data' AS data
FROM http_patch('http://httpbin.org/patch', '{"this":"that"}',
'application/json');
```

status	content_type	data
200	application/json	'{"this":"that"}'

-- 使用DELETE命令来请求资源的删除。

```
SELECT status, content_type, content::json->>'url' AS url
FROM http_delete('http://httpbin.org/delete');
```

status	content_type	url
200	application/json	http://httpbin.org/delete

— 作为将数据发送到GET请求的快捷方式，传递一个JSONB数据参数。

```
SELECT status, content::json->'args' AS args
FROM http_get('http://httpbin.org/get',
             jsonb_build_object('myvar', 'myval', 'foo', 'bar'));
```

— 要使用数据有效负载而不是嵌入在URL中的参数将数据POST到URL，请使用JSONB将数据编码为数据有效负载。

```
SELECT status, content::json->'form' AS form
FROM http_post('http://httpbin.org/post',
              jsonb_build_object('myvar', 'myval', 'foo', 'bar'));
```

— 要访问二进制内容，您必须使用textsend函数将内容从默认的varchar表示强制转换为bytea表示。使用默认的varchar::bytea转换将无法正常工作，因为转换将在第一次遇到值为零的字节（二进制数据中很常见）时停止。

```
WITH
  http AS (
    SELECT * FROM http_get('http://httpbin.org/image/png')
  ),
  headers AS (
    SELECT (unnest(headers)).* FROM http
  )
SELECT
  http.content_type,
  length(textsend(http.content)) AS length_binary,
  headers.value AS length_headers
FROM http, headers
WHERE field = 'Content-Length';
```

content_type	length_binary	length_headers
image/png	8090	8090



---

# Appendix I. 额外提供的程序

这个附录及之前的一个包含了在LightDB发布的contrib目录中可以找到的模块的信息。在Appendix I 中有更多关于contrib小节的总体信息，以及contrib中的服务器扩展和插件的详细信息。

这个附录覆盖了contrib中能找到的功能程序。一旦被安装（不管是从源代码安装还是从包系统安装），它们就可以在LightDB安装的bin目录中找到，并且能够和任何其他程序一样使用。

## I. 1. 客户端应用

这一节覆盖了contrib中的LightDB客户端应用。它们可以从任何位置运行，不管数据库服务器运行在哪里。核心LightDB发布中的客户端应用也可见LightDB 客户端应用。

## oid2name

oid2name — 解析一个LightDB数据目录中的 OID 和文件结点。

## Synopsis

```
oid2name [option...]
```

## 描述

oid2name是一个帮助管理员检查被 LightDB 使用的文件结构的工具程序。要使用它，你需要熟悉数据库文件结构（见Chapter 64）。

### Note

名称“oid2name”是有历史原因的，它确实有些误导性，因为在你使用它的大部分时间里，你实际关心的是表的文件结点编号（在数据目录中是可见的文件名）。请确定你理解表 OID 和表文件结点之间的区别！

oid2name连接到一个目标数据库并且抽取 OID、文件节点或者表名信息。你也可以让它显示数据库 OID 或表空间 OID。

## 选项

oid2name接受下列命令行参数：

```
-f filenode
```

```
--filenode=filenode
```

显示具有文件结点的表的信息filenode.

```
-i
```

```
--indexes
```

在列表中包括索引和序列.

```
-o oid
```

```
--oid=oid
```

显示具有OID的表的信息 oid.

```
-q
```

```
--quiet
```

忽略头部（用于脚本）.

```
-s
```

```
--tablespaces
```

显示表空间 OIDs.

```
-S
```

```
--system-objects
```

包括系统对象（位于 information\_schema, pg\_toast 和 pg\_catalog 模式）.

```
-t tablename_pattern
```

```
--table=tablename_pattern
```

显示匹配的表的信息 tablename\_pattern.

-V  
--version

打印oid2name版本并退出。

-x  
--extended

显示关于每个对象所展现的更多信息：表空间名， 模式名， 以及 OID。

-?  
--help

显示有关oid2name命令行参数的帮助并退出。

oid2name也接受下列用于连接参数的命令行参数：

-d database  
--dbname=database

要连接的数据库。

-h host  
--host=host

数据库服务器的主机。

-H host

数据库服务器的主机。这个参数在 LightDB 12中已经不推荐 。

-p port  
--port=port

数据库服务器的端口。

-U username  
--username=username

用于连接的用户名。

要显示特定表，通过使用-o、-f和-t选择要显示哪个表。 -o采用一个 OID， -f采用一个文件节点， 而-t采用一个表名（实际上，它是一个LIKE模式，因此你可以用诸如foo%之类的东西）。 这些选项你想用多少就用多少，最后的列举将包括所有匹配任意一个这些选项的对象。但是注意这些选项只能显示由-d给定的数据库中的对象。

如果你没有给出任何-o、-f或者-t，但是给出了-d，它将列出由-d指定的数据库中的所有表。在这种模式下，-S和-i选项控制什么会被列出。

如果你也没有给出-d，它将显示一个数据库 OID 的列表。你也可以给出-s来得到一个表空间列表。

## 环境

LTHOST  
LTPORT  
LTUSER

默认连接参数。

与大多数其他 LightDB的实用程序一样，这个实用程序也使用libpq 支持的环境变量（参见 Section 32.14）。

环境变量PG\_COLOR指定是否在诊断消息中使用颜色。可能的值是always、auto和 never。

## 注解

oid2name要求一个运行着的数据库服务器并且其系统目录没有损坏。因此它对于数据库损坏的情况用处有限。

## 例子

```
$ # 在这个数据库服务器中到底有什么?
$ oid2name
All databases:
  Oid Database Name Tablespace
-----
 17228      alvherre pg_default
 17255      regression pg_default
 17227      template0  pg_default
 1      templatel pg_default

$ oid2name -s
All tablespaces:
  Oid Tablespace Name
-----
 1663      pg_default
 1664      pg_global
 155151     fastdisk
 155152     bigdisk

$ # OK, 让我们看看数据库 alvherre 里面
$ cd $LTDATA/base/17228

$ # 得到默认表空间中前十个数据库对象, 按尺寸排序
$ ls -lS * | head -10
-rw----- 1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw----- 1 alvherre alvherre 17965056 sep 14 09:51 1155291
-rw----- 1 alvherre alvherre 1204224 sep 14 09:51 16717
-rw----- 1 alvherre alvherre 581632 sep 6 17:51 1255
-rw----- 1 alvherre alvherre 237568 sep 14 09:50 16674
-rw----- 1 alvherre alvherre 212992 sep 14 09:51 1249
-rw----- 1 alvherre alvherre 204800 sep 14 09:51 16684
-rw----- 1 alvherre alvherre 196608 sep 14 09:50 16700
-rw----- 1 alvherre alvherre 163840 sep 14 09:50 16699
-rw----- 1 alvherre alvherre 122880 sep 6 17:51 16751

$ # 我好奇文件 155173 是 ...
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode Table Name
-----
 155173      accounts

$ # 你可以请求多于一个对象
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode Table Name
-----
 155173      accounts
 1155291     accounts_pkey
```

```

$ # 你可以混合选项, 并且用 -x 得到更多细节
$ oid2name -d alvherre -t accounts -f 1155291 -x
From database "alvherre":
  Filenode      Table Name      Oid  Schema  Tablespace
-----
    155173      accounts      155173  public  pg_default
    1155291  accounts_pkey  1155291  public  pg_default

$ # 为每个数据库对象显示磁盘空间
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE      `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16          1155287  branches_pkey
16          1155289  tellers_pkey
17561       1155291  accounts_pkey
...

$ # 相同, 但是按尺寸排序
$ du [0-9]* | sort -rn | while read SIZE FN
> do
>   echo "$SIZE      `oid2name -q -d alvherre -f $FN`"
> done
133466      155173   accounts
17561       1155291  accounts_pkey
1177        16717   pg_proc_proname_args_nsp_index
...

$ # 如果你想看看表空间里有什么, 使用 lt_tblspc 目录
$ cd $LTDATA/lt_tblspc
$ oid2name -s
All tablespaces:
  Oid  Tablespace Name
-----
    1663      pg_default
    1664      pg_global
    155151     fastdisk
    155152     bigdisk

$ # 哪些数据库在表空间 "fastdisk" 中有对象?
$ ls -d 155151/*
155151/17228/ 155151/PG_VERSION

$ # 噢, 什么是数据库 17228 ?
$ oid2name
All databases:
  Oid  Database Name  Tablespace
-----
    17228      alvherre  pg_default
    17255      regression  pg_default
    17227      template0  pg_default
    1         template1  pg_default

$ # 让我们看这个数据库在该表空间中有哪些对象。
$ cd 155151/17228
$ ls -l
total 0

```

```
-rw----- 1 lightdb lightdb 0 sep 13 23:20 155156
```

```
$ # OK, 这是个很小的表, 但是哪一个是它?
```

```
$ oid2name -d alvherre -f 155156
```

```
From database "alvherre":
```

```
  Filenode  Table Name
```

```
-----  
    155156          foo
```

作者

B. Palmer <bpalmer@crimelabs.net>

## vacuumlo

vacuumlo — 从LightDB数据库中移除孤立的大对象

## Synopsis

```
vacuumlo [option...] dbname...
```

## 描述

vacuumlo是一个从LightDB数据库中移除“孤立”大对象的简单使用程序。一个孤立的大对象（LO）是指其OID不出现在数据中任何oid或lo数据列中的LO。

如果你使用该程序，你也许还会对lo模块中的lo\_manage触发器感兴趣。lo\_manage对于避免创建孤立LO有用处。

在命令行中提到的所有数据库都将被处理。

## 选项

vacuumlo接受下列命令行参数：

```
-l limit  
--limit=limit
```

在每一个事务中移除不超过limit个大对象（默认值为1000）。因为移除每一个LO时服务器都将要求一个锁，所以在一个事务中移除过多的LO会有超过max\_locks\_per\_transaction的风险。如果你想在一个事务中就完成所有的移除工作，请将这个限制设置为0。

```
-n  
--dry-run
```

不移除任何东西，只是显示将会做什么。

```
-v  
--verbose
```

写一些进度消息。

```
-V  
--version
```

打印vacuumlo的版本并退出。

```
-?  
--help
```

显示关于vacuumlo的命令行参数，并且退出。

vacuumlo也接受下列命令行参数用于连接：

```
-h host  
--host=host
```

数据库服务器的主机名。

`-p port`  
`--port=port`

数据库服务器的端口。

`-U username`  
`--username=username`

用于连接的用户名。

`-w`  
`--no-password`

不要发出一个口令提示。如果服务器要求口令认证并且没有其他方式可以提供一个口令（例如一个`.pgpass`文件），连接尝试将会失败。这个选项可用于批处理任务以及脚本，因为在这些情况下不会有用户输入口令。

`-W`  
`--password`

强制`vacuumlo`在连接到数据库之前提示要求一个口令。

这个选项不是不可缺少的，因为如果服务器要求口令认证，`vacuumlo`会自动提示要求一个口令。但是，`vacuumlo`将会浪费一次连接尝试来了解到服务器需要口令。在某些情况，值得用`-W`来避免这种额外的连接尝试。

## 环境

`LTHOST`  
`LTPORT`  
`LTUSER`

默认连接参数。

这个实用程序，像其他大多数LightDB实用程序，也还使用 `libpq` 支持的环境变量(参见 Section 32.1)。

环境变量 `PG_COLOR` 指定是否在诊断消息中使用颜色。可能的值是 `always`、`auto` 和 `never`。

## 说明

`vacuumlo`按照下列方法工作：首先`vacuumlo`建立一个临时表，其中包含所选择数据库中所有大对象的OID。然后它会扫描数据库中所有类型为`oid`或`lo`的列，并且从临时表中移除在这些列中出现过的OID（注意：只有类型为这些名字的才被考虑，特别的，在这些类型之上的域是不被考虑的）。临时表中剩下的项就标识了鼓励LO。它们将被移除。

## 作者

Peter Mount <peter@retep.org.uk>

## I. 2. 服务器应用

某些应用程序在LightDB服务器上运行。目前，`contrib`目录中不包含此类应用程序。请参阅 [LightDB 服务器应用](#) 以获取有关作为核心 LightDB 发行版一部分的服务器应用程序的信息。



## lt\_standby

lt\_standby — 支持创建一个LightDB温备服务器

## Synopsis

lt\_standby [选项...] 归档位置 下一个WAL文件 WAL文件路径 [重启WAL文件]

## 描述

lt\_standby支持创建一个“温备”数据库服务器。它被设计为一个生产就绪的程序，同时也是一个可定制的模板，如果您需要特定的修改。

lt\_standby被设计为一个等待restore\_command，这是将标准归档恢复转换为温备操作所需的。其他配置也是必需的，所有这些都主服务器手册中描述（参见Section 24.2）。

要配置一个备用服务器使用lt\_standby，请将以下内容放入其lightdb.conf配置文件中：

```
restore_command = 'lt_standby archiveDir %f %p %r'
```

其中archiveDir是应从中恢复WAL段文件的目录。

如果指定了restartwalfile（通常使用%r宏），则所有逻辑上在此文件之前的WAL文件将从archivelocation中删除。这最小化了需要保留的文件数量，同时保留了崩溃恢复功能。如果archivelocation是特定备用服务器的短暂暂存区，那么使用此参数是适当的，但不适用于archivelocation用作长期WAL存档区域的情况。

lt\_standby假定archivelocation是服务器所有用户可读的目录。如果指定了restartwalfile（或-k），则archivelocation目录也必须是可写的。

当主服务器失败时，有两种方法可以故障切换到“温备”数据库服务器：

### 智能故障切换

在智能故障切换中，应用存档中可用的所有WAL文件后，服务器会启动。这会导致零数据丢失，即使备用服务器已经落后，但如果存在大量未应用的WAL，备用服务器变为就绪之前可能需要很长时间。要触发智能故障切换，请创建一个包含单词smart的触发器文件，或者只需创建它并留空。

### 快速故障切换

在快速故障切换中，服务器会立即启动。存档中尚未应用的任何WAL文件将被忽略，并且这些文件中的所有事务都将丢失。要触发快速故障切换，请创建一个触发器文件并将单词fast写入其中。如果在定义的时间间隔内没有出现新的WAL文件，lt\_standby还可以配置自动执行快速故障切换。

## 选项

lt\_standby接受以下命令行参数：

-c

使用cp或copy命令从存档中恢复WAL文件。这是唯一支持的行为，因此此选项无用。

-d

在stderr上打印大量调试日志输出。

-k

从archivelocation中删除文件，以便在当前文件之前最多不保留这么多WAL文件。零（默认值）表示不从archivelocation中删除任何文件。如果指定了restartwalfile，则将静默忽略此参数，因为该指定方法更准确地确定正确的存档截止点。从LightDB 8.3开始，使用此参数是不推荐的；更安全和更有效的方法是指定restartwalfile参数。设置过小可能导致删除仍然需要重启备用服务器的文件，而设置过大会浪费存档空间。

-r maxretries

设置在失败时重试复制命令的最大次数（默认值为3）。每次失败后，我们都会等待sleep\_time \* num\_retries，以便等待时间逐渐增加。因此，默认情况下，我们将在报告失败之前等待5秒、10秒，然后是15秒。这将被解释为恢复结束，备用将完全上线。

-s sleeptime

设置在测试WAL文件是否已经在存档中可用时，每个测试之间休眠的秒数（最多60秒，默认为5秒）。不一定推荐使用默认设置；请参阅Section 24. 进行讨论。

-t triggerfile

指定一个触发文件，其存在应导致故障切换。建议您使用结构化的文件名，以避免在同一系统上存在多个服务器时混淆触发的服务器；例如/tmp/pgsql.trigger.5432。

-V

--version

打印lt\_standby的版本并退出。

-w maxwaittime

设置等待下一个WAL文件的最大秒数，超过这个时间后将执行快速故障切换。零（默认值）表示无限等待。默认设置不一定推荐；请参阅Section 24. 进行讨论。

-?

--help

显示lt\_standby命令行参数的帮助信息，然后退出。

## 注意事项

lt\_standby旨在与LightDB 8.2及更高版本一起使用。

LightDB 8.3提供%r宏，旨在让lt\_standby知道它需要保留的最后一个文件。在LightDB 8.2中，如果需要进行存档清理，则必须使用-k选项。该选项在8.3中仍然可用，但其使用已被弃用。

LightDB 8.4提供了recovery\_end\_command选项。如果没有该选项，剩余的触发文件可能会有危险。

lt\_standby使用C编写，并且具有易于修改的源代码，其中具有专门指定的部分可供自定义需求进行修改。

## 示例

在Linux或Unix系统上，您可以使用以下命令：

```
archive_command = 'cp %p .../archive/%f'
```

```
restore_command = 'lt_standby -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f  
%p %r 2>>standby.log'
```

```
recovery_end_command = 'rm -f /tmp/pgsql.trigger.5442'
```

其中存档目录物理上位于备用服务器上，以便archive\_command通过NFS访问它，但文件是备用服务器本地的（启用了ln的使用）。这将：

- 在standby.log中生成调试输出
- 检查下一个WAL文件可用性之间暂停2秒
- 仅当出现名为/tmp/pgsql.trigger.5442的触发文件时，才停止等待，并根据其内容执行故障转移
- 恢复结束时删除触发文件
- 从存档目录中删除不再需要的文件

在Windows上，您可以使用以下命令：

```
archive_command = 'copy %p ...\\archive\\%f'
```

```
restore_command = 'lt_standby -d -s 5 -t C:\pgsql.trigger.5442 ...\archive %f %p  
%r 2>>standby.log'
```

```
recovery_end_command = 'del C:\pgsql.trigger.5442'
```

请注意，必须在archive\_command中加倍反斜杠，但在restore\_command或recovery\_end\_command中不需要。这将：

- 使用copy命令从存档中恢复WAL文件
- 在standby.log中生成调试输出
- 检查下一个WAL文件可用性之间暂停5秒
- 仅当出现名为C:\pgsql.trigger.5442的触发文件时，才停止等待，并根据其内容执行故障转移
- 恢复结束时删除触发文件
- 从存档目录中删除不再需要的文件

Windows上的copy命令在文件完全复制之前设置最终文件大小，这通常会令lt\_standby混淆。因此，lt\_standby在看到正确的文件大小后会等待sleeptime秒。GNUWin32的cp仅在文件复制完成后设置文件大小。

由于Windows示例在两端都使用copy，因此任一或两个服务器都可以通过网络访问存档目录。

## 作者

Simon Riggs <simon@2ndquadrant.com>

## 另请参阅

lt\_archivecleanup

---

## Appendix J. 外部项目

LightDB是一项复杂的软件项目，管理它是一项困难的工作。我们发现许多LightDB的提高可以通过独立于核心项目的方式更有效地开发。

### J. 1. 客户端接口

在基本的LightDB发布中仅包含两种客户端接口：

- libpq被包括的原因是它是主要的 C 语言接口，并且许多其它客户端接口都是在它的基础上构建的。
- ECPG被包括的原因是它依赖于服务器端 SQL 语法，并且因此对LightDB自身的变化非常敏感。

除此以外的所有其它语言接口都是外部项目并独立发布。Table J. 包括了其中一些项目的列表。需要注意的是其中一些包的发布许可证与LightDB不同。要了解更多关于每种语言的接口细节（包括许可证条款），请参考它的网站和文档。

Table J. 1. 外部维护的客户端接口

名字	语言	注释	网站
DBD::Pg	Perl	Perl DBI 驱动	<a href="https://metacpan.org/release/DBD-Pg">https://metacpan.org/release/DBD-Pg</a>
JDBC	Java	Type 4 JDBC 驱动	<a href="https://jdbc.postgresql.org/">https://jdbc.postgresql.org/</a>
libpqxx	C++	C++ 接口	<a href="https://pqxx.org/">https://pqxx.org/</a>
node-postgres	JavaScript	Node.js 驱动器	<a href="https://node-postgres.com/">https://node-postgres.com/</a>
Npgsql	.NET	.NET 数据提供者	<a href="https://www.npgsql.org/">https://www.npgsql.org/</a>
pgtcl	Tcl		<a href="https://github.com/flightaware/Pgtcl">https://github.com/flightaware/Pgtcl</a>
pgtclng	Tcl		<a href="https://sourceforge.net/projects/pgtclng/">https://sourceforge.net/projects/pgtclng/</a>
pq	Go	Go的数据库/sql的Pure Go驱动程序	<a href="https://github.com/lib/pq">https://github.com/lib/pq</a>
psqlODBC	ODBC	ODBC 驱动	<a href="https://odbc.postgresql.org/">https://odbc.postgresql.org/</a>
psycopg	Python	DB API 2.0兼容	<a href="https://www.psycopg.org/">https://www.psycopg.org/</a>

### J. 2. 管理工具

有多个管理工具可用于LightDB。最流行的是pgAdmin<sup>1</sup>，并且也有一些商业版的可用。

### J. 3. 过程语言

---

<sup>1</sup> <https://www.pgadmin.org/>

LightDB在基本发布中包括了多种过程语言：PL/pgSQL, PL/oraSQL.

此外，还有一些过程语言是在核心LightDB发布之外被开发和维护的。Table J. 列出了其中的一些包。注意其中某些项目的发行许可证与LightDB不同。要了解每种过程语言的更多信息（包括许可证信息），请参考它的网站和文档。

Table J.2. 外部维护的过程语言

名称	语言	网站
PL/Java	Java	<a href="https://tada.github.io/pljava/">https://tada.github.io/pljava/</a>
PL/Lua	Lua	<a href="https://github.com/pllua/pllua-ng">https://github.com/pllua/pllua-ng</a>
PL/R	R	<a href="https://github.com/postgres-plr/plr">https://github.com/postgres-plr/plr</a>
PL/sh	Unix shell	<a href="https://github.com/petere/plsh">https://github.com/petere/plsh</a>
PL/v8	JavaScript	<a href="https://github.com/plv8/plv8">https://github.com/plv8/plv8</a>

## J.4. 扩展

LightDB被设计成很容易扩展。由于这个原因，被载入到数据库中的扩展可以像内建特性那样运行。contrib/目录中包含了多个扩展的源码，在Appendix 中有相关介绍。其他扩展的开发是独立进行的，例如PostGIS<sup>2</sup>。甚至LightDB的复制解决方案也能在外部开发。例如，Slony-I<sup>3</sup>是一个在核心项目之外独立开发的流行的主/备复制解决方案。

<sup>2</sup> <https://postgis.net/>

<sup>3</sup> <https://www.slony.info>

---

## Appendix K. LightDB限制

Table K. 描述了LightDB的各种硬性限制。但是，在达到绝对硬限制之前，可能会先受到现实的限制，例如性能限制或可用磁盘空间。

Table K.1. LightDB限制

项目	上限值	说明
数据库大小	不限	
数据库数量	4, 294, 950, 911	
每个数据库的关系数量	1, 431, 650, 303	
关系大小	32 TB	基于BLCKSZ为缺省的8192字节
每个表的记录数	受限于可以放到4, 294, 967, 295个页中的元组数	
每个表的列数	1600	同时受限于元组大小，需要能放到单个页中；参考下面的说明
结果集中的列	1664	
字段大小	1 GB	
标识符长度	63字节	可以通过重新编译LightDB增大
每个表的索引数	不限	受每个数据库最大关系数的约束
每个索引的列数	32	可以通过重新编译LightDB增大
分区键数量	32	可以通过重新编译LightDB增大

由于要存储的元组必须适合单个8192字节的堆页面，因此表的最大列数进一步减少。例如，除元组头外，由1600个int列组成的元组将占用6400字节并可以存储在堆页面中，而一个包含1600个bigint列的元组将消耗12800字节，因此无法放入单个堆页面。当类型为比如text, varchar和char的可变长度字段时，如果值的长度足够大，它们的值可以存储在行外的TOAST表中。表堆中的元组中只保留18个字节的指针。对于较短长度的可变长度字段，使用4字节或1字节的字段头，并且该值存储在堆元组内部。

从表中删除的列也会影响最大列限制。此外，尽管在元组的空位图中将新创建的元组的删除列值内部标记为空，但空位图也占用空间。

---

## Appendix L. 首字母缩写词

在LightDB的文档和讨论中有很多常用的首字母缩写词。

ANSI

American National Standards Institute<sup>1</sup>

API

Application Programming Interface<sup>2</sup>

ASCII

American Standard Code for Information Interchange<sup>3</sup>

BKI

Backend Interface

CA

Certificate Authority<sup>4</sup>

CIDR

Classless Inter-Domain Routing<sup>5</sup>

CPAN

Comprehensive Perl Archive Network<sup>6</sup>

CRL

Certificate Revocation List<sup>7</sup>

CSV

Comma Separated Values<sup>8</sup>

CTE

Common Table Expression

CVE

Common Vulnerabilities and Exposures<sup>9</sup>

DBA

Database Administrator<sup>10</sup>

---

<sup>1</sup> [https://en.wikipedia.org/wiki/American\\_National\\_Standards\\_Institute](https://en.wikipedia.org/wiki/American_National_Standards_Institute)

<sup>2</sup> <https://en.wikipedia.org/wiki/API>

<sup>3</sup> <https://en.wikipedia.org/wiki/Ascii>

<sup>4</sup> [https://en.wikipedia.org/wiki/Certificate\\_authority](https://en.wikipedia.org/wiki/Certificate_authority)

<sup>5</sup> [https://en.wikipedia.org/wiki/Classless\\_Inter-Domain\\_Routing](https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing)

<sup>6</sup> <https://www.cpan.org/>

<sup>7</sup> [https://en.wikipedia.org/wiki/Certificate\\_revocation\\_list](https://en.wikipedia.org/wiki/Certificate_revocation_list)

<sup>8</sup> [https://en.wikipedia.org/wiki/Comma-separated\\_values](https://en.wikipedia.org/wiki/Comma-separated_values)

<sup>9</sup> <https://cve.mitre.org/>

<sup>10</sup> [https://en.wikipedia.org/wiki/Database\\_administrator](https://en.wikipedia.org/wiki/Database_administrator)

DBI

Database Interface (Perl)<sup>11</sup>

DBMS

Database Management System<sup>12</sup>

DDL

Data Definition Language<sup>13</sup>, SQL commands such as CREATE TABLE, ALTER USER

DML

Data Manipulation Language<sup>14</sup>, SQL commands such as INSERT, UPDATE, DELETE

DST

Daylight Saving Time<sup>15</sup>

ECPG

Embedded C for LightDB

ESQL

Embedded SQL<sup>16</sup>

FAQ

Frequently Asked Questions<sup>17</sup>

FSM

Free Space Map

GEQO

Genetic Query Optimizer

GIN

Generalized Inverted Index

GiST

Generalized Search Tree

Git

Git<sup>18</sup>

GMT

Greenwich Mean Time<sup>19</sup>

---

<sup>11</sup> <https://dbi.perl.org/>

<sup>12</sup> <https://en.wikipedia.org/wiki/Dbms>

<sup>13</sup> [https://en.wikipedia.org/wiki/Data\\_Definition\\_Language](https://en.wikipedia.org/wiki/Data_Definition_Language)

<sup>14</sup> [https://en.wikipedia.org/wiki/Data\\_Manipulation\\_Language](https://en.wikipedia.org/wiki/Data_Manipulation_Language)

<sup>15</sup> [https://en.wikipedia.org/wiki/Daylight\\_saving\\_time](https://en.wikipedia.org/wiki/Daylight_saving_time)

<sup>16</sup> [https://en.wikipedia.org/wiki/Embedded\\_SQL](https://en.wikipedia.org/wiki/Embedded_SQL)

<sup>17</sup> <https://en.wikipedia.org/wiki/FAQ>

<sup>18</sup> [https://en.wikipedia.org/wiki/Git\\_\(software\)](https://en.wikipedia.org/wiki/Git_(software))

<sup>19</sup> <https://en.wikipedia.org/wiki/GMT>



GSSAPI

Generic Security Services Application Programming Interface<sup>20</sup>

GUC

Grand Unified Configuration, the LightDB subsystem that handles server configuration

HBA

Host-Based Authentication

HOT

Heap-Only Tuples<sup>21</sup>

IEC

International Electrotechnical Commission<sup>22</sup>

IEEE

Institute of Electrical and Electronics Engineers<sup>23</sup>

IPC

Inter-Process Communication<sup>24</sup>

ISO

International Organization for Standardization<sup>25</sup>

ISSN

International Standard Serial Number<sup>26</sup>

JDBC

Java Database Connectivity<sup>27</sup>

JIT

Just-in-Time compilation<sup>28</sup>

JSON

JavaScript Object Notation<sup>29</sup>

LDAP

Lightweight Directory Access Protocol<sup>30</sup>

---

<sup>20</sup> [https://en.wikipedia.org/wiki/Generic\\_Security\\_Services\\_Application\\_Program\\_Interface](https://en.wikipedia.org/wiki/Generic_Security_Services_Application_Program_Interface)

<sup>21</sup> <https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/backend/access/heap/README.HOT;hb=HEAD>

<sup>22</sup> [https://en.wikipedia.org/wiki/International\\_Electrotechnical\\_Commission](https://en.wikipedia.org/wiki/International_Electrotechnical_Commission)

<sup>23</sup> <https://standards.ieee.org/>

<sup>24</sup> [https://en.wikipedia.org/wiki/Inter-process\\_communication](https://en.wikipedia.org/wiki/Inter-process_communication)

<sup>25</sup> <https://www.iso.org/home.html>

<sup>26</sup> <https://en.wikipedia.org/wiki/Issn>

<sup>27</sup> [https://en.wikipedia.org/wiki/Java\\_Database\\_Connectivity](https://en.wikipedia.org/wiki/Java_Database_Connectivity)

<sup>28</sup> [https://en.wikipedia.org/wiki/Just-in-time\\_compilation](https://en.wikipedia.org/wiki/Just-in-time_compilation)

<sup>29</sup> <https://www.json.org>

<sup>30</sup> [https://en.wikipedia.org/wiki/Lightweight\\_Directory\\_Access\\_Protocol](https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol)

LSN

Log Sequence Number, see `pg_lsn` and WAL Internals.

MSVC

Microsoft Visual C<sup>31</sup>

MVCC

Multi-Version Concurrency Control

NLS

National Language Support<sup>32</sup>

ODBC

Open Database Connectivity<sup>33</sup>

OID

Object Identifier

OLAP

Online Analytical Processing<sup>34</sup>

OLTP

Online Transaction Processing<sup>35</sup>

ORDBMS

Object-Relational Database Management System<sup>36</sup>

PAM

Pluggable Authentication Modules<sup>37</sup>

PGSQL

LightDB

PGXS

LightDB Extension System

PID

Process Identifier<sup>38</sup>

PITR

Point-In-Time Recovery (Continuous Archiving)

---

<sup>31</sup> [https://en.wikipedia.org/wiki/Visual\\_C++](https://en.wikipedia.org/wiki/Visual_C++)

<sup>32</sup> [https://en.wikipedia.org/wiki/Internationalization\\_and\\_localization](https://en.wikipedia.org/wiki/Internationalization_and_localization)

<sup>33</sup> [https://en.wikipedia.org/wiki/Open\\_Database\\_Connectivity](https://en.wikipedia.org/wiki/Open_Database_Connectivity)

<sup>34</sup> <https://en.wikipedia.org/wiki/Olap>

<sup>35</sup> <https://en.wikipedia.org/wiki/OLTP>

<sup>36</sup> <https://en.wikipedia.org/wiki/ORDBMS>

<sup>37</sup> [https://en.wikipedia.org/wiki/Pluggable\\_Authentication\\_Modules](https://en.wikipedia.org/wiki/Pluggable_Authentication_Modules)

<sup>38</sup> [https://en.wikipedia.org/wiki/Process\\_identifier](https://en.wikipedia.org/wiki/Process_identifier)

PL

Procedural Languages (server-side)

POSIX

Portable Operating System Interface<sup>39</sup>

RDBMS

Relational Database Management System<sup>40</sup>

RFC

Request For Comments<sup>41</sup>

SGML

Standard Generalized Markup Language<sup>42</sup>

SPI

Server Programming Interface

SQL

Structured Query Language<sup>43</sup>

SRF

Set-Returning Function

SSH

Secure Shell<sup>44</sup>

SSL

Secure Sockets Layer<sup>45</sup>

SSPI

Security Support Provider Interface<sup>46</sup>

SYSV

Unix System V<sup>47</sup>

TCP/IP

Transmission Control Protocol (TCP) / Internet Protocol (IP)<sup>48</sup>

TID

Tuple Identifier

---

<sup>39</sup> <https://en.wikipedia.org/wiki/POSIX>

<sup>40</sup> [https://en.wikipedia.org/wiki/Relational\\_database\\_management\\_system](https://en.wikipedia.org/wiki/Relational_database_management_system)

<sup>41</sup> [https://en.wikipedia.org/wiki/Request\\_for\\_Comments](https://en.wikipedia.org/wiki/Request_for_Comments)

<sup>42</sup> <https://en.wikipedia.org/wiki/SGML>

<sup>43</sup> <https://en.wikipedia.org/wiki/SQL>

<sup>44</sup> [https://en.wikipedia.org/wiki/Secure\\_Shell](https://en.wikipedia.org/wiki/Secure_Shell)

<sup>45</sup> [https://en.wikipedia.org/wiki/Secure\\_Sockets\\_Layer](https://en.wikipedia.org/wiki/Secure_Sockets_Layer)

<sup>46</sup> <https://msdn.microsoft.com/en-us/library/aa380493%28VS.85%29.aspx>

<sup>47</sup> [https://en.wikipedia.org/wiki/System\\_V](https://en.wikipedia.org/wiki/System_V)

<sup>48</sup> [https://en.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://en.wikipedia.org/wiki/Transmission_Control_Protocol)

TLS

Transport Layer Security<sup>49</sup>

TOAST

The Oversized-Attribute Storage Technique

TPC

Transaction Processing Performance Council<sup>50</sup>

URL

Uniform Resource Locator<sup>51</sup>

UTC

Coordinated Universal Time<sup>52</sup>

UTF

Unicode Transformation Format<sup>53</sup>

UTF8

Eight-Bit Unicode Transformation Format<sup>54</sup>

UUID

Universally Unique Identifier

WAL

Write-Ahead Log

XID

Transaction Identifier

XML

Extensible Markup Language<sup>55</sup>

---

<sup>49</sup> [https://en.wikipedia.org/wiki/Transport\\_Layer\\_Security](https://en.wikipedia.org/wiki/Transport_Layer_Security)

<sup>50</sup> <http://www.tpc.org/>

<sup>51</sup> <https://en.wikipedia.org/wiki/URL>

<sup>52</sup> [https://en.wikipedia.org/wiki/Coordinated\\_Universal\\_Time](https://en.wikipedia.org/wiki/Coordinated_Universal_Time)

<sup>53</sup> <https://www.unicode.org/>

<sup>54</sup> <https://en.wikipedia.org/wiki/Utf8>

<sup>55</sup> <https://en.wikipedia.org/wiki/XML>

---

## Appendix M. 术语表

这是一个术语以及它们在LightDB和一般关系数据库系统上下文中的含义的列表。

ACID		原子性(Atomicity), 一致性(Consistency), 隔离性(Isolation), 和持久性(Durability). 这组数据库事务属性是为了保证并发操作的有效性, 并且在发生错误、电源故障等情况下也是如此。
Aggregate (routine)	function	一种结合(aggregates)多个输入值的function, 例如通过计数、平均或相加, 产生一个输出值。  更多信息, 请参见 Section 9.21  See Also Window function (routine).
Analytic function		See Window function (routine).
Analyze (operation)		从tables和其他relations中的数据中收集统计信息的过程, 以帮助 query planner做出关于如何执行queries的决策。  (不要把这个术语与EXPLAIN命令的ANALYZE选项搞混淆。)  更多信息, 请参见ANALYZE.
Atomic		关于 datum: 它的值不能被分解成更小的组件。  关于 database transaction: 参见 atomicity.
Atomicity		transaction的一种属性, 它的所有操作要么作为单个单元完成, 要么不操作。此外, 如果在事务执行过程中发生了系统故障, 恢复后不会看到部分结果。这是ACID的属性之一。
Attribute		tuple中具有特定名称和数据类型的元素。
Autovacuum (process)		一组例行执行vacuum和analyze操作的后台程序。  更多信息, 请参见 Section 22.1.6
Backend (process)		代表client session并处理其请求的instance进程。  (不要把这个词与类似的Background Worker或Background Writer搞混淆)。
Background (process)	worker	instance中的进程, 该进程运行系统或用户提供的代码。作为LightDB中的一些特性的基础架构, 例如logical replication和parallel queries。此外, Extensions可以添加自定义的后台工作者进程。  更多信息, 参见 Chapter 44
Background (process)	writer	将脏data pages从shared memory写入文件系统的进程。它会定期地唤醒, 但只在很短的周期内工作, 以便将大开销的I/O活动分布在一段时间内, 避免产生可能阻塞其他进程的更大的I/O峰值。  更多信息, 参见 Section 17.4.5
Bloat		数据页中不包含当前行版本的空间, 例如未使用(空闲)空间或过时的行版本。

Cast	<p>从当前数据类型到另一数据类型的datum的转换。</p> <p>更多信息，参见 CREATE CAST。</p>
Catalog	<p>SQL标准使用这个术语来表示LightDB术语中所谓的database。</p> <p>(不要把这个术语与 system catalog搞混淆)。</p> <p>更多信息，参见 Section 20.1</p>
Check constraint	<p>一种定义在 relation上面的constraint类型，以限制一个或多个attributes中允许的值。检查约束可以引用关系中的同一行的任何属性，但不能引用同一关系的其他行或者其他关系。</p> <p>更多信息，参见 Section 5.4</p>
Checkpoint	<p>WAL序列中的一个点，在这个点上可以保证堆和索引数据文件已经被更新，包括所有来自在检查点之前被修改过的shared memory的信息；一个checkpoint record被写入并刷新到WAL以标记该点。</p> <p>检查点也是执行到达上述定义的检查点所需的所有操作的行为。当满足预定义的条件时，如已经过了指定的时间，或者已写入了一定数量的记录，则启动此过程；也可以由用户通过命令CHECKPOINT来调用。</p> <p>更多信息，参见 Section 27.4</p>
Checkpointner (process)	负责执行检查点的专用流程。
Class (archaic)	See Relation.
Client (process)	通过connecting到instance与database交互来建立session的任何进程，可能是远程的。
Column	在table或view中找到的attribute。
Commit	<p>在database中完成transaction的行为，以使其他事务可见并确保他的durability。</p> <p>更多信息，参见 COMMIT。</p>
Concurrency	这个概念指在database里面在同一时间发生多个独立的操作。在LightDB中，并发性是由multiversion concurrency control机制来控制的。
Connection	<p>客户端进程和backend进程之间的通讯的一个已建成的线路，通常是基于网络，支持session。这个术语有时候用做为会话的同义词。</p> <p>更多信息，参见 Section 17.3</p>
Consistency	database中的数据的特性总是遵从于integrity constraints。事务可能被允许短暂的违反一些约束，在它被提交之前，这样的事务自动被rolled back。这是ACID属性中的一个。
Constraint	<p>table中数据的值上所允许的限制，或的属性。</p> <p>更多信息，参见 Section 5.4</p>

Data area	See Data directory.
Database	local SQL objects的一个已命名的汇集? 更多信息, 参见 Section 20.1
Database cluster	数据库和全局SQL对象的一个汇集, 以及他们的公用状态和动态元数据。有时被引用为cluster。  在LightDB中, 术语cluster有时也用来指一个实例。(不要将这个术语与SQL命令CLUSTER混淆。)
Database server	See Instance.
Data directory	server文件系统上的基本目录, 包含与database cluster相关联的所有数据文件和子目录 (除了tablespaces和可选的WAL)。环境变量LTDATA通常用于引用数据目录。  cluster的存储空间包括数据目录以及一些额外的表空间。 更多信息, 参见 Section 64.1
Data page	用于存储关系数据的基本结构。所有的页面大小相同。数据页往往存储在磁盘上, 每个数据页都存储在一个特定的文件中, 并且可以被读取到shared buffers, 在那里它们可以被修改, 变为dirty的。它们写入磁盘时变得干净。新页面, 最初只存在于内存中, 在写入之前也是脏的。
Datum	SQL数据类型的一个值的内部表示方式。
Delete	从给定table或relation中删除rows的SQL命令。 更多信息, 参见 DELETE。
Durability	保证一旦transaction被committed, 即使在系统故障或崩溃后变更仍然存在。这是ACID的特性之一。
Epoch	See Transaction ID.
Extension	可以安装在instance上的附加软件包, 以获得额外特性。 更多信息, 参见 Section 36.17
File segment	存储给定relation的数据的一种物理文件。文件段的大小受配置值的限制(通常为1G比特), 所以如果一个关系超过这个大小, 它就被分割成多个段。  更多信息, 参见 Section 64.1  (不要将这个术语与相似的WAL segment混淆)。
Foreign data wrapper	表示本地database中不包含的数据的一种方式, 使其看起来就像在本地table(s)中一样。使用外部数据封装器, 可以定义外部foreign server和foreign tables。  更多信息, 参见 CREATE FOREIGN DATA WRAPPER。
Foreign key	一种定义在table中的一个或多个columns上的constraint类型, 它需要这些columns中的值标识另一个表中的零或一行(或者, 偶尔是相同的)。
Foreign server	foreign tables的命名的集合, 它们都使用相同的foreign data wrapper, 并具有其他共同的配置值。

---

	更多信息，参见 CREATE SERVER。
Foreign table (relation)	<p>一种relation，看起来拥有与常规table相似的rows和columns，但将通过它的foreign data wrapper转发对数据的请求，并将返回根据foreign table定义的结构化的result sets。</p> <p>更多信息，参见CREATE FOREIGN TABLE。</p>
Fork	在被存储的关系中的每个单独分段文件集。main fork是实际数据驻留的地方。对于元数据还存在两个次要分支:free space map和visibility map。Unlogged relations也有一个init fork。
Free space map (fork)	<p>一种存储结构，用于保存关于表的主分支的每个数据页的元数据。每个页面的空闲空间映射条目存储对于未来元组可用的空闲空间数量，并且是结构化的，可以高效地搜索给定大小的新元组的可用空间。</p> <p>更多信息，参见 Section 64.3</p>
Function (routine)	<p>一种例程类型，它接收零个或多个参数，返回零个或多个输出值，并被限制在一个事务中运行。函数被调用作为查询的一部分，例如通过SELECT。某些函数可以返回 sets；这些被称为set-returning functions。</p> <p>函数还可以用于triggers调用。</p> <p>更多信息，参见 CREATE FUNCTION。</p>
Grant	<p>一个SQL命令，用于允许user或role访问database中的特定对象。</p> <p>更多信息，参见 GRANT。</p>
Heap	包含对于关系的row属性(例如数据)的值。堆是在关系的main fork中的一个或多个file segments中实现的。
Host	与其他计算机通过网络通信的计算机。这有时被用作server的同义词。它也用于指运行client processes的计算机。
Index (relation)	<p>包含从table或materialized view中派生的数据的relation。它的内部结构支持对原始数据的快速检索和访问。</p> <p>更多信息，参见 CREATE INDEX。</p>
Insert	<p>用于向table中添加新数据的SQL命令。</p> <p>更多信息，参见 INSERT。</p>
Instance	<p>一组使用公共的共享内存区域进行通信的后端进程和辅助进程。一个 postmaster process进程管理实例；一个实例管理一个database cluster及其所有数据库。只要TCP端口不冲突，许多实例可以在同一台server上运行。</p> <p>实例处理DBMS的所有关键属性：读和写到文件以及共享内存，确保ACID特性，connections到client processes，特权验证，崩溃恢复，复制等等。</p>

---



---

Isolation	在提交之前，事务的效果对concurrent transactions不可见的属性。这是ACID的特性之一。  更多信息，参见 Section 13.2
Join	在queries中用于组合来自多个relations的数据的操作和SQL关键字。
Key	一种标识table或其他relation中的row的方法，通过关系中一个或多个attributes中包含的值。
Lock	允许进程限制或阻止对资源同时进行访问的一种机制。
Log file	日志文件包含关于事件的人类可读的文本行。例如登录失败、长时间运行的查询，等等。  更多信息，参见 Section 22.3
Logged	如果对表的更改被发送到WAL，则认为该table被logged。默认情况下，所有常规表都被记日志。可以将表指定为unlogged，在创建时或通过ALTER TABLE命令。
Logger (process)	如果被激活，该进程将有关数据库事件的信息写入当前log file。当达到某些依赖于时间或卷的标准时，将创建一个新的日志文件。也叫syslogger。  更多信息，参见 Section 17.8
Log record	WAL record的古体术语叫法。
Master (server)	See Primary (server).
Materialized	预先计算并存储一些信息的属性，以供以后使用，而不是在运行中计算。  这个术语在materialized view中使用，意思是从视图的查询中导出的数据并与该数据的来源分开存储在磁盘上。  这个术语也用于代指一些多步骤查询，表示执行给定步骤产生的数据存储在内存中(有溢出到磁盘的可能性)，这样就可以通过另一个步骤多次读取数据。
Materialized (relation)	view 由SELECT语句定义的relation(就像view)，但以与table相同的方式存储数据。它不能通过INSERT, UPDATE, 或DELETE操作进行修改。  更多信息，参见 CREATE MATERIALIZED VIEW。
Multi-version concurrency control (MVCC)	一种机制，用于允许一些transactions读取和写入相同的行，而不会因一个进程导致其他进程停顿。在LightDB中，MVCC是通过在tuples被修改时创建副本(versions)来实现的;在可以看到旧版本终止的事务之后，需要删除那些旧版本。
Null	不存在性(non-existence)的概念，是关系数据库理论的中心原则。它表示没有一个确定的值。
Optimizer	See Query planner.
Parallel query	处理执行部分query的能力，以利用具有多个CPU的服务器上的并行进程。

---

Partition		<p>大的集合中的几个不相交(不重叠)的子集之一。</p> <p>对partitioned table的引用:每个表都包含分区表的部分数据的表之一,称为parent。分区本身是一个表,所以也可以直接查询;同时,分区有时可以是一个分区表,允许创建层次结构。</p> <p>query中有关的window function,分区是一种用户定义的标准,它确定该函数可以考虑query's result set的哪些邻近rows。</p>
Partitioned (relation)	table	<p>语义上与table,相同的relation,但其存储分布在多个partitions上。</p>
Postmaster (process)		<p>instance的第一个过程。它启动和管理其他辅助进程,并根据需要创建backend processes。</p> <p>更多信息,参见 Section 16.3</p>
Primary key		<p>在table或其他relation上定义的unique constraint的一种特殊情况,它也保证primary key内的所有attributes不会有null值。顾名思义,每个表只能有一个主键,但也可能有多个惟一的约束,这些约束也不支持空(null-capable)属性。</p>
Primary (server)		<p>当两个或多个databases通过replication链接时,被认为是权威信息源的server称为primary,也称为master。</p>
Procedure (routine)		<p>一种例程。它们的特点是不返回值,并且允许它们执行如COMMIT 和 ROLLBACK这样的事务语句。它们是通过CALL命令调用的。</p> <p>更多信息,参见 CREATE PROCEDURE。</p>
Query		<p>客户端向backend发送的请求,通常用于返回结果或修改数据库上的数据。</p>
Query planner		<p>LightDB中用于确定(planning)执行queries的最有效方法的部分。也称为query optimizer、optimizer或简称为planner。</p>
Record		<p>See Tuple.</p>
Recycling		<p>See WAL file.</p>
Referential integrity		<p>通过foreign key限制一个relation中的数据的一种方法,使它在另一个relation中必须有匹配的数据。</p>
Relation		<p>database中所有对象的通用术语,有名称和以特定顺序定义的attributes列表。Tables, sequences, views, foreign tables, materialized views、复合类型和indexes都是关系。</p> <p>更通用的来说,关系是一组元组;例如,查询的结果也是一个关系。</p> <p>在LightDB中,Class是relation的古老同义词。</p>
Replica (server)		<p>与primary数据库配对并维护主数据库部分或全部数据副本的database。这样做的最主要的原因是允许对该数据进行更大的访问,并在primary服务器不可用时保持数据的可用性。</p>

---

Replication	<p>将一台server上的数据复制到另一台服务器上的行为称为replica。这可以采用physical replication的形式，其中一个服务器上的所有文件更改都是逐字复制的，或者是logical replication，其中已定义的数据更改子集使用更高层次的表示来传递。</p>
Result set	<p>SQL命令完成后从backend process传输到client的relation，通常是SELECT，但如果指定了RETURNING子句，则可以是INSERT、UPDATE或DELETE命令。</p> <p>结果集是一个关系这一事实意味着一个查询可以用于另一个查询的定义，成为一个subquery。</p>
Revoke	<p>用于防止指定roles列表对指定database对象集的访问的命令。</p> <p>更多信息，参见 REVOKE。</p>
Role	<p>instance的访问权限的集合。角色本身是可以授予其他角色的特权。当多个users需要相同的特权时，这样做通常是为了方便或确保完整性。</p> <p>更多信息，参见 CREATE ROLE。</p>
Rollback	<p>撤销自transaction开始以来执行的所有操作的命令。</p> <p>更多信息，参见 ROLLBACK。</p>
Routine	<p>存储在数据库系统中可被调用以执行的一组已定义的指令集合。例程可以用多种编程语言编写。例程可以是functions(包括返回集的函数和trigger functions)、aggregate functions和procedures。</p> <p>LightDB自身已经定义了许多例程，但是也可以添加用户定义的例程。</p>
Row	See Tuple.
Savepoint	<p>transaction中步骤序列中的特殊标记。在此时间点之后的数据修改可能会被恢复到保存点的时间。</p> <p>更多信息，参见 SAVEPOINT。</p>
Schema	<p>模式是SQL objects的名称空间，这些对象都驻留在同一个database中。每个SQL对象必须确切的驻留在一个模式中。</p> <p>所有系统定义的SQL对象都驻留在模式pg_catalog中。</p> <p>更通常地，schema术语用于表示给定database或其子集的所有数据描述(table定义、constraints、注释等)。</p> <p>更多信息，参见 Section 5.9</p>
Segment	See File segment.
Select	<p>用于从database请求数据的SQL命令。通常，SELECT命令不会以任何方式修改database，但是查询中调用的functions可能会产生修改数据的作用。</p> <p>更多信息，参见 SELECT。</p>

---

Sequence (relation)	一种用于生成值的关系类型。通常生成的值是连续的非重复的数字。它们通常用于生成替代的primary key值。
Server	运行LightDB实例的计算机。术语server表示实际硬件、容器或virtual machine。  这个术语有时用于指实例或主机。
Session	允许客户端和后端进行交互的状态，通过connection通信。
Shared memory	由instance的公共进程使用的RAM。它可以镜像部分database文件，为WAL records提供一个临时区域，并存储其他公共信息。注意，共享内存属于整个实例，而不是单个数据库。  共享内存中最大的部分称为shared buffers，用于镜像组成页面的部分数据文件。当页面被修改时，它被称为脏页面，直到它被写回文件系统。  更多信息，参见 Section 17.4.1
SQL object	任何可以使用CREATE命令创建的对象。大多数对象都特定于一个数据库，通常称为local objects。  大多数本地对象都属于它们所被包含的数据库中的特定schema，比如relations(所有类型)、routines(所有类型)、数据类型等。同一模式中相同类型的对象的名称是强制性要求唯一的。  也存在不属于模式的局部对象；一些例子是extensions、data type casts和foreign data wrappers。数据库中相同类型的对象的名称必须是唯一的。  其他对象类型，如roles、tablespaces、复制源、逻辑复制的订阅和数据库本身都不是本地SQL对象，因为它们完全存在于任何特定的数据库之外；它们被称为global objects。在整个数据库集群中这些对象的名称必须是唯一的。  更多信息，参见 Section 20.1
SQL standard	定义SQL语言的一系列文档。
Standby (server)	See Replica (server).
Stats collector (process)	该流程收集关于instance的活动的统计信息。  更多信息，参见 Section 25.2
System catalog	tables的集合，描述了实例的所有SQL objects的结构。系统目录位于pg_catalog模式中。这些表包含内部表示的数据，并且通常不被认为对用户检查有用；许多用户友好的views(，也在pg_catalog模式中，提供了对其中一些信息的更方便的访问，而在模式information_schema(参见 Chapter 35)中存在其他表和视图，它们公开了SQL standard规定的一些相同和额外的信息  更多信息，参见 Section 5.9

---

Table	<p>具有公共数据结构的tuples集合(相同数量的attributes、以同样的顺序、每个位置具有相同的名称和类型)。表是LightDB中最常见的relation形式。</p> <p>更多信息, 参见 CREATE TABLE。</p>
Tablespace	<p>服务器文件系统上的一个已命名位置。所有需要在它们的系统编目中定义上面存储的SQL objects必须属于单个表空间。最初, 一个数据库集群包含一个可用的表空间, 它被用作于所有SQL对象的默认表空间, 称为pg_default。</p> <p>更多信息, 参见 Section 20.6</p>
Temporary table	<p>在session或transaction的生命周期内存在的Tables, 如创建时指定的那样。其中的数据对其他会话是不可见的, 并且不会被logged。临时表经常用于存储多步骤操作的中间数据。</p> <p>更多信息, 参见 CREATE TABLE。</p>
TOAST	<p>一种将表行的大型属性分割并存储在辅助表中的机制, 该表被称为TOAST table。每个具有大型属性的关系都有自己的TOAST表。</p> <p>更多信息, 参见 Section 64.2</p>
Transaction	<p>必须作为单个atomic命令的命令组合: 它们都作为单个单元成功或失败, 并且它们的效果对其他sessions是不可见的, 直到事务完成, 甚至更晚, 这取决于隔离级别。</p> <p>更多信息, 参见 Section 13.2</p>
Transaction ID	<p>每个事务在第一次导致数据库修改时所接收到的数字的、唯一的、按顺序分配的标识符。常缩写为xid。当存储在磁盘上时, xids仅有有32位宽, 因此只能生成大约40亿个写事务id; 为了允许系统运行的比那个更长, 使用了epochs, 也是32位宽。当计数器达到最大的xid值时, 它从3开始(那以下的值被保留了), epoch增加值为1。在某些上下文中, epoch和xid值一起被视为单个64位值。</p> <p>更多信息, 参见 Section 8.19</p>
Transactions per second (TPS)	<p>每秒执行的平均事务数, 在测量运行的所有活动会话中合计。这是被用做衡量实例的性能特性。</p>
Trigger	<p>当某个操作 (INSERT, UPDATE, DELETE, TRUNCATE) 应用于某个relation时, 可以被定义执行的function。触发器与调用它的语句在同一个transaction中执行, 如果函数失败, 则调用语句也失败。</p> <p>更多信息, 参见 CREATE TRIGGER。</p>
Tuple	<p>按固定顺序排列的attributes集合。该顺序可以由包含元组的table(或其他relation)定义, 在这种情况下, 元组通常被称为一行。它也可以由结果集的结构定义, 在这种情况下, 它有时被称为record。</p>
Unique constraint	<p>一种relation上定义的constraint, 其限制的值允许在一个列或列的组合, 以便每个值或值的组合关系中只能出现一次, 也就是说, 没有其他关系中的行包含值与之相等。</p>

---

---

	因为null values不被认为是相等的，所以允许在不违反唯一约束的情况下存在多个空值行。
Unlogged	<p>某些relations的性质，即它们的变化没有反映在WAL中。这将禁用这些关系的复制和崩溃恢复。</p> <p>非日志表的主要用途是存储必须要跨进程共享的临时工作数据。</p> <p>Temporary tables总是非日志的。</p>
Update	<p>用于修改指定table中可能已经存在的rows的SQL命令。它不能创建或删除行。</p> <p>更多信息，参见 UPDATE。</p>
User	具有LOGIN 特权的role。
User mapping	<p>将本地database中的登录凭据转换为由foreign data wrapper定义的远程数据系统中的凭据。</p> <p>更多信息，参见 CREATE USER MAPPING。</p>
Vacuum	<p>从表或物化视图中删除过时的tuple versions的过程，以及MVCC的LightDB实现所需的其他密切相关的处理过程。这可以通过使用VACUUM命令来启动，但是也可以通过autovacuum进程自动处理。</p> <p>更多信息，参见Section 22.1</p>
View	<p>由SELECT语句定义的relation，但没有自己的存储空间。每当查询引用视图时，视图的定义就会被替换到查询中，就好像用户输入它作为子查询而不是视图的名称。</p> <p>更多信息，参见 CREATE VIEW。</p>
Visibility map (fork)	<p>一种存储结构，用于保存表的主分支的每个数据页的元数据。每个页面的可视性映射条目存储两个比特：第一个(all-visible)表示页面中的所有元组对所有事务都可见。第二个(all-frozen)表示页面中的所有元组都标记为冻结。</p>
WAL	See Write-ahead log.
WAL archiver (process)	<p>一种保存WAL files副本的过程，目的是创建备份或保持当前的replicas。</p> <p>更多信息，参见 Section 23.3</p>
WAL file	<p>又称WAL segment或WAL segment file。为WAL提供存储空间的每个按顺序编号的文件。这些文件都具有相同的预定义大小，并且是按顺序写入的，当它们在多个同时发生的会话中发生时，会将更改穿插在其中。如果系统崩溃，则按顺序读取文件，并重现每个更改，以将系统恢复到崩溃前的状态。</p> <p>当checkpoint把每个WAL文件的所有修改写入相应的数据文件后，就可以释放每个WAL文件。释放文件可以通过删除它，也可以通过更改它的名称以便将来使用，这被称为recycling。</p>

---

	更多信息, 参见Section 27.5
WAL record	对单个数据更改的低级描述。它包含足够的信息, 以便数据更改可以被重新执行(replayed), 在系统故障导致更改丢失的情况下。WAL记录使用不可打印的二进制格式。  更多信息, 参见Section 27.5
WAL segment	See WAL file.
WAL writer (process)	将WAL records从shared memory写入WAL files的进程。  更多信息, 参见 Section 17.5
Window function (routine)	query中使用的一种function, 用于查询result set的一个partition; 该函数的结果基于在相同分区或框架的rows中找到的值。  所有aggregate functions都可以用作窗口函数, 但是窗口函数也可以用来给分区中的每一行排序。也称为analytic functions。  更多信息, 参见Section 3.5
Write-ahead log	当用户和系统调用操作发生时, 跟踪database cluster中的更改的日志。它包含许多顺序写入WAL files的独立WAL records。

---

## Appendix N. 颜色支持

LightDB 软件包中的大多数程序都可以生成彩色控制台输出。本附录介绍如何配置。

### N. 1. 当使用颜色时

要使用颜色输出，设置环境变量PG\_COLOR如下：

1. 如果值为 `always`，则颜色被应用。
2. 如果值为 `auto` 并且标准错误流被关联到终端设备，则颜色被应用。
3. 否则，颜色不被应用。

### N. 2. 配置颜色

要使用的实际颜色通过环境变量PG\_COLORS进行配置(注意复数)。该值是key=value对的冒号分隔的列表。键指定要使用的颜色。值为 SGR（选择图形再现）规范，由终端解释。

下列键当前被使用：

`error`

用于高亮文本 “`error`” 在错误消息中

`warning`

用于高亮文本 “`warning`” 在告警消息中

`locus`

用于在消息中高亮位置信息（例如程序名称和文件名称）

默认值为 `error=01;31:warning=01;35:locus=01` (`01;31` = 粗体红色, `01;35` = 粗体洋红, `01` = 粗体默认颜色)。

#### Tip

这些颜色规定格式也用于其他软件包，例如 GCC, GNU coreutils, 和 GNU grep.



---

## Appendix 0. LightDB JDBC

LightDB JDBC驱动程序允许Java程序使用标准的、与数据库无关的Java代码连接到LightDB。

除了标准连接参数外，驱动程序还支持一些额外的属性，可用于指定与LightDB特定的额外驱动程序行为。

- `lightdbSyntaxCompatibleType` (String) 指定LightDB兼容的类型，有效值为`oracle/mysql/off`，默认值为`off`。它可以与Oracle数据库、MySQL数据库兼容。例如：

```
jdbc:postgresql://localhost:5432/postgres?  
lightdbSyntaxCompatibleType=oracle
```

jdbc可以与Oracle数据库兼容

- `currentSchema` (String) 指定要设置在搜索路径中的模式（或用逗号分隔的多个模式）。此模式将用于解析在此连接上使用的语句中使用的未限定对象名称。例如：

```
jdbc:postgresql://localhost:5432/postgres?currentSchema=  
\"$user\", oracle
```

jdbc将自动在搜索路径中补充`public`。实际的搜索路径是`\"$user\", oracle, public, lt_catalog`

- `options` (String) 指定选项中的guc参数。格式如下：`options=-c guc参数名称=参数值`，其中`options`后面的值需要进行编码。例如：

```
jdbc:postgresql://localhost:5432/postgres?  
options=-c%20search_path=\"$user\",public,oracle,lt_catalog%20-c  
%20statement_timeout=90000
```

`search_path`是guc参数的一部分，因此您也可以设置`search_path`。

# Appendix P. 扩展的钩子列表

扩展可以使用许多钩子来扩展内部功能。钩子可以修改不同位置的行为，包括在运行 SQL 语句时。

下面列出了可用的钩子列表

Table P.1. 钩子列表

位置	钩子	描述
analyze.h	post_parse_analyze_hook	插件在分析解析结束时获得控制的钩子
	lt_post_parse_analyze_stmt	插件在语句分析解析结束时获得控制的钩子
planner.h	planner_hook	插件在 planner() 中获得控制的钩子
	create_upper_paths_hook	插件在 grouping_planner() 规划上层关系时获得控制的钩子
	pre_create_ordinary_grouping_paths_hook	插件在 create_ordinary_grouping_paths 之前获得控制的钩子
	lt_distribute_qual_to_rels	插件在将条件分配给关系时获得控制的钩子
paths.h	set_rel_pathlist_hook	插件在 set_rel_pathlist() 中获得控制的钩子
	set_join_pathlist_hook	插件在 add_paths_to_joinrel() 中获得控制的钩子
	join_search_hook	插件替换 standard_join_search() 的钩子
	pushdown_qual_to_subquery	插件决定是否将 quals 推入子查询的钩子
planmain.h	support_semi_join_to_inner_join	插件转换半连接为内连接时获得控制的钩子
	lt_order_qual_clauses_hook	插件在排序 quals 子句时获得控制的钩子
Executor.h	ExecutorStart_hook	插件在 ExecutorStart() 中获得控制的钩子
	ExecutorRun_hook	插件在 ExecutorRun() 中获得控制的钩子
	ExecutorFinish_hook	插件在 ExecutorFinish() 中获得控制的钩子
	ExecutorEnd_hook	插件在 ExecutorEnd() 中获得控制的钩子
	ExecutorCheckPerms_hook	插件在 ExecCheckRTPerms() 中获得控制的钩子
Utility.h	ProcessUtility_hook	插件在 ProcessUtility() 中获得控制的钩子

扩展的钩子列表

位置	钩子	描述
plancat.h	get_relation_info_hook	插件在 get_relation_info() 中获得控制的钩子
auth.h	ClientAuthentication_hook	插件在 ClientAuthentication() 中获得控制的钩子
explain.h	ExplainOneQuery_hook	插件在 ExplainOneQuery() 中获得控制的钩子
	explain_get_index_name_hook	插件在 explain_get_index_name() 中获得控制的钩子
user.h	check_password_hook	检查 CreateRole() 和 AlterRole() 中的密码的钩子
elog.h	emit_log_hook	在消息发送到服务器日志之前拦截消息的钩子
fmgr.h	needs_fmgr_hook	函数调用的钩子
	fmgr_hook	函数调用的钩子
lsyscache.h	get_attavgwidth_hook	插件在 get_attavgwidth() 中获得控制的钩子
selfuncs.h	get_relation_stats_hook	我们要求统计信息时插件获得控制的钩子
	get_index_stats_hook	我们要求索引统计信息时插件获得控制的钩子
objectaccess.h	object_access_hook	插件提供与此签名匹配的钩子函数
libpq-be.h	openssl_tls_init_hook	SSL 的初始化钩子, 如果适用, 则默认设置密码回调函数
rowsecurity.h	row_security_policy_hook_permit	允许扩展添加自己的安全策略的钩子, 用于添加与其他宽容策略结合的策略, 使用 OR 运算。
ipc.h	shmem_startup_hook	允许扩展设置其 shmem 分配的钩子。
prep.h	lt_judge_can_pull_up_hook	插件决定是否提取子查询的钩子
	lt_judge_sublink_can_pull_up_hook	插件决定是否提取子链接的钩子

---

# 参考书目

一些有关SQL和LightDB的参考和读物。

一些来自于最初的POSTGRES开发团队的白皮书和技术报告可以在加州大学伯克利分校计算机系的网站<sup>1</sup>上找到。

## SQL参考书

- [bowman01] The Practical SQL Handbook. Using SQL Variants. Fourth Edition. Judith Bowman, Sandra Emerson, and Marcy Darnovsky. ISBN 0-201-70309-2. Addison-Wesley Professional. 2001.
- [date97] A Guide to the SQL Standard. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date and Hugh Darwen. ISBN 0-201-96426-0. Addison-Wesley. 1997.
- [date04] An Introduction to Database Systems. Eighth Edition. C. J. Date. ISBN 0-321-19784-4. Addison-Wesley. 2003.
- [elma04] Fundamentals of Database Systems. Fourth Edition. Ramez Elmasri and Shamkant Navathe. ISBN 0-321-12226-7. Addison-Wesley. 2003.
- [melt93] Understanding the New SQL. A complete guide. Jim Melton and Alan R. Simon. ISBN 1-55860-245-3. Morgan Kaufmann. 1993.
- [ull88] Principles of Database and Knowledge-Base Systems. Classical Database Systems. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.
- [sqltr-19075-6] SQL Technical Report. Part 6: SQL support for JavaScript Object Notation (JSON). First Edition. 2017.

## LightDB相关文档

- [sim98] Enhancement of the ANSI SQL Implementation of LightDB. Stefan Simkovic. Department of Information Systems, Vienna University of Technology. Vienna, Austria. November 29, 1998.
- [yu95] The Postgres95. User Manual. A. Yu and J. Chen. University of California. Berkeley, California. Sept. 5, 1995.
- [fong] The design and implementation of the POSTGRES query optimizer<sup>2</sup>. Zelaine Fong. University of California, Berkeley, Computer Science Department.

## 会议记录和期刊论文

- [ports12] “Serializable Snapshot Isolation in LightDB<sup>3</sup>”. D. Ports and K. Grittner. VLDB Conference, August 2012.
- [berenson95] “A Critique of ANSI SQL Isolation Levels<sup>4</sup>”. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil. ACM-SIGMOD Conference on Management of Data, June 1995.

---

<sup>1</sup> <https://dsf.berkeley.edu/papers/>

<sup>2</sup> <https://dsf.berkeley.edu/papers/UCB-MS-zfong.pdf>

<sup>3</sup> <https://arxiv.org/pdf/1208.4179>

<sup>4</sup> <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf>

- [olson93] Partial indexing in POSTGRES: research project. Nels Olson. UCB Engin T7.49.1993 0676. University of California. Berkeley, California. 1993.
- [ong90] “A Unified Framework for Version Modeling Using Production Rules in a Database System”. L. Ong and J. Goh. ERL Technical Memorandum M90/33. University of California. Berkeley, California. April, 1990.
- [rowe87] “The POSTGRES data model<sup>5</sup>”. L. Rowe and M. Stonebraker. VLDB Conference, Sept. 1987.
- [seshadri95] “Generalized Partial Indexes<sup>6</sup>”. P. Seshadri and A. Swami. Eleventh International Conference on Data Engineering, 6 - 10 March 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, California. 1995. 420 - 7.
- [ston86] “The design of POSTGRES<sup>7</sup>”. M. Stonebraker and L. Rowe. ACM-SIGMOD Conference on Management of Data, May 1986.
- [ston87a] “The design of the POSTGRES. rules system”. M. Stonebraker, E. Hanson, and C. H. Hong. IEEE Conference on Data Engineering, Feb. 1987.
- [ston87b] “The design of the POSTGRES storage system<sup>8</sup>”. M. Stonebraker. VLDB Conference, Sept. 1987.
- [ston89] “A commentary on the POSTGRES rules system<sup>9</sup>”. M. Stonebraker, M. Hearst, and S. Potamianos. SIGMOD Record 18(3). Sept. 1989.
- [ston89b] “The case for partial indexes<sup>10</sup>”. M. Stonebraker. SIGMOD Record 18(4). Dec. 1989. 4 - 11.
- [ston90a] “The implementation of POSTGRES<sup>11</sup>”. M. Stonebraker, L. A. Rowe, and M. Hirohama. Transactions on Knowledge and Data Engineering 2(1). IEEE. March 1990.
- [ston90b] “On Rules, Procedures, Caching and Views in Database Systems<sup>12</sup>”. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. ACM-SIGMOD Conference on Management of Data, June 1990.

---

<sup>5</sup> <https://dsf.berkeley.edu/papers/ERL-M87-13.pdf>

<sup>6</sup> <https://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.5740>

<sup>7</sup> <https://dsf.berkeley.edu/papers/ERL-M85-95.pdf>

<sup>8</sup> <https://dsf.berkeley.edu/papers/ERL-M87-06.pdf>

<sup>9</sup> <https://dsf.berkeley.edu/papers/ERL-M89-82.pdf>

<sup>10</sup> <https://dsf.berkeley.edu/papers/ERL-M89-17.pdf>

<sup>11</sup> <https://dsf.berkeley.edu/papers/ERL-M90-34.pdf>

<sup>12</sup> <https://dsf.berkeley.edu/papers/ERL-M90-36.pdf>

---

# Index

## Symbols

- \$, 40
- \$libdir, 984
- \$libdir/plugins, 551, 1682
- \*, 119
- .pgpass, 783
- .pg\_service.conf, 784
- ::, 46
- \_PG\_fini, 984
- \_PG\_init, 984
- \_PG\_output\_plugin\_init, 1286
- 下标, 41
- 不可重复读, 423
- 不带时区的时间, 141
- 不带时区的时间戳, 141
- 不是一个数字
  - 数字 (数据类型), 134
- 事件日志
  - 事件日志, 487
- 事件触发器, 1048
- 事务 ID
  - 回卷, 613
- 事务隔离, 423
- 事务隔离级别, 423
  - 可序列化, 426
  - 可重复读, 425
  - 设置, 1762
  - 读已提交, 424
- 二进制数据, 140
- 交叉连接, 106
- 任意精度数字, 133
- 位串
  - 数据类型, 157
- 信号
  - 后端进程, 341
- 信号量, 471
- 信息模式, 913
- 值表达式, 39
- 儒略日期, 2399
- 全文搜索, 385
  - 函数和操作符, 158
  - 数据类型, 158
- 公共表表达式 (see WITH)
- 共享内存, 471
- 共享库, 991
- 关系, 7
- 关系数据库, 7
- 关闭, 478
- 准备一个查询
  - 在 PL/pgSQL 中, 1133
- 函数
  - 一次调用中的类型决定, 362
  - 位置记号法, 54
  - 命名参数, 967
  - 命名记号法, 55
  - 在FROM子句中, 111
  - 多态, 964
  - 混合记号法, 55
  - 用户定义的
    - in SQL, 966
  - 调用, 42
  - 输出参数, 972
- 函数依赖, 116
- 分组, 114
- 切片面包 (see TOAST)
- 列, 7
- 列引用, 40
- 初始化分支, 2318
- 删除, 103
- 别名
  - 在FROM子句中, 109
  - 在选择列表中, 120
- 加密, 481
- 动态载入, 984
- 匿名代码块, 1600
- 协议
  - 前端-后端, 2180
- 单用户模式, 2078
- 参数
  - 语法, 40
- 双精度, 135
- 口令
  - 认证, 572
- 口令文件, 783
- 只用索引的扫描, 380
- 只读事务
  - 设置, 1762
- 可序列化, 426
- 可序列化快照隔离, 423
- 可延迟事务
  - 设置, 1762
- 可更新视图, 1589
- 可见性映射, 2318
- 右连接, 107
- 同步复制, 633
- 同步提交, 699
- 后台工作者, 1279
- 后备服务器, 633
- 咨询锁, 432
- 回卷
  - 事务 ID 的, 613
- 圆括号, 40
- 域选择, 41
- 基础类型, 963
- 复制, 633
- 复制槽
  - 流复制, 639
  - 逻辑复制, 1284
- 复制源, 1291
- 复制进度追踪, 1291
- 外连接, 106
- 外键

- 自引用, 65
- 多态函数, 964
- 多态类型, 964
- 多版本并发控制, 423
- 子查询, 13, 48, 110
- 字符, 138
- 字符串
  - 数据类型, 138
- 字符集, 600
- 存储参数, 1540
- 客户端认证, 563
- 对象标识符
  - 数据类型, 190
- 层次数据库, 7
- 左连接, 107
- 带时区的时间, 141
- 带时区的时间戳, 141
- 并发, 423
- 并行查询, 455
- 幻读, 423
- 序列化异常, 423, 426
- 库初始化函数, 984, 984
- 异步提交, 699
- 循环
  - 在 PL/oraSQL 中, 1162
- 性能, 435
- 扩展, 1027
  - externally maintained, 3105
- 扩展 SQL, 963
- 报告错误
  - 在 PL/pgSQL 中, 1120
- 排序, 122
- 排序规则, 595
- 接口
  - externally maintained, 3104
- 控制文件, 1028
- 插入, 101
- 搜索路径
  - 对象可见性, 331
- 操作符
  - 一次调用中的类型决定, 359
  - 用户定义的, 1013
  - 调用, 42
- 操作符类, 1018
- 数字 (数据类型), 133
- 数据分区, 633
- 数据区域 (see 数据库集簇)
- 数据库活动
  - 监控, 652
- 数据库集簇, 7, 466
- 数据类型, 131
  - enumerated (enum), 151
  - 内部组织, 985
  - 分类, 359
  - 基础, 963
  - 多态, 964
  - 数字, 132
  - 用户定义的, 1009
  - 组合, 963
  - 转换, 358
- 数组
  - 构造器, 48
  - 用户定义类型的, 1012
- 整数, 133
- 文件系统挂载点, 467
- 文本搜索, 385
  - 函数和操作符, 158
  - 数据类型, 158
  - 索引, 418
- 文档
  - 全文搜索, 385
- 无限大
  - 浮点型, 136
- 日常维护, 610
- 日志传送, 633
- 时区
  - POSIX样式规范, 2397
- 时间跨度, 141
- 更新, 102
- 有序集聚集, 42
- 服务器日志, 526
  - 日志文件维护, 617
- 服务器欺骗, 481
- 权限
  - 与规则, 1077
  - 与视图, 1077
- 条件表达式, 299
- 查询, 9, 105
- 查询树, 1056
- 查询计划, 435
- 标签 (see 别名)
- 标量 (see 表达式)
- 格里高利历, 2398
- 检查点, 700
- 模式, 588
- 死锁, 431
- 比较
  - 子查询结果行, 318
  - 行构造器, 320
  - 逐行, 320
- 注释
  - 关于数据库对象, 337
- 流复制, 633
- 流水线处理
  - 协议规范, 2188
- 浮点, 135
- 清理, 610
- 温备, 633
- 游标
  - 显示查询计划, 1655
- 热备, 633
- 物化视图
  - 通过规则实现, 1064
- 环境变量, 782
- 用户, 582
- 用户变量, 51

- 用户名映射, 569
- 监控
  - 数据库活动, 652
- 目标列表, 1057
- 磁盘用量, 695
- 磁盘驱动器, 703
- 移动聚集模式, 1005
- 空值
  - in DISTINCT, 120
- 空闲空间映射, 2318
- 窗口函数
  - invocation, 44
  - 内建, 316
  - 执行顺序, 119
- 管理工具
  - externally maintained, 3104
- 类型 (see 数据类型)
- 索引, 372
  - GIN, 2306
    - 文本搜索, 418
  - GiST
    - 文本搜索, 418
  - 只用索引的扫描, 380
  - 哈希, 2312
  - 覆盖, 380
  - 锁, 434
- 索引访问方法, 2272
- 级联复制, 633
- 线性回归, 313
- 线段, 154
- 组合类型, 179, 963
  - 常量, 180
  - 构造器, 50
  - 比较, 320
- 统计, 313
- 统计信息
  - of the planner, 446
- 维护, 610
- 编译
  - libpq 应用, 789
- 聚集函数, 12
  - support functions for, 1009
  - 内建, 310
  - 可变, 1006
  - 多态, 1006
  - 有序集, 1007
  - 用户定义的, 1003
  - 移动聚集, 1005
  - 调用, 42
  - 部分聚集, 1008
- 脏读, 423
- 自动提交
  - 大批量载入数据, 452
- 自动清理
  - 一般信息, 615
  - 配置参数, 538
- 自然连接, 107
- 范围表, 1056
- 行, 7
- 行估计
  - 规划器, 2330
- 行类型, 179
  - 构造器, 50
- 表, 7
- 表函数, 111
- 表空间, 591
- 表表达式, 105
- 表访问方法, 2271
- 表达式
  - 计算的顺序, 53
  - 语法, 39
- 表采样方法, 2260
- 覆盖索引, 380
- 规则, 1056
  - 与触发器比较, 1079
  - 和物化视图, 1064
  - 和视图, 1057
- 视图
  - 物化, 1064
  - 通过规则实现, 1057
- 角色, 582, 586
- 触发器
  - 与规则比较, 1079
  - 在 PL/pgSQL 中, 1122
  - 用于更新一个派生的 tsvector 列, 400
  - 触发器函数的参数, 1041
- 证书, 579
- 读已提交, 424
- 谓词锁, 426
- 负载均衡, 633
- 输入函数, 1010
- 输出函数, 1010
- 过程语言, 1082
  - externally maintained, 3104
- 连接, 10, 106
  - 交叉, 106
  - 右, 107
  - 左, 107
  - 控制顺序, 450
  - 自然, 107
- 连接服务文件, 784
- 连锁的事务
  - 在 PL/pgSQL, 1120
- 逆分布, 315
- 透明数据加密, 720
- 逐行比较, 320
- 逻辑解码, 1282, 1284
- 遗传查询优化, 523
- 配置
  - of the server, 488
- 重启点, 701
- 重复, 10, 120
- 重复读, 425
- 重建索引, 616
- 重载
  - 操作符, 1013



- 锁, 428
    - 咨询, 432
  - 错误消息, 743
  - 间隔, 141
  - 集合交, 121
  - 集合差, 121
  - 集合并, 121
  - 集合操作, 121
  - 集合返回函数
    - 函数, 323
  - 集簇
    - 数据库 (see 数据库集簇)
  - 非数字
    - 浮点型, 136
  - 非阻塞连接, 760
  - 面向对象数据库, 7
  - 预备语句
    - 显示查询计划, 1655
  - 颜色, 2343, 3124
  - 高可用, 633
- A**
- abbrev, 262
  - ABORT, 1302
  - abs, 200
  - ACL, 70
  - aclcontains, 330
  - acldefault, 330
  - aclexplode, 330
  - aclitem, 74
  - aclitemeq, 330
  - acos, 202
  - acosd, 202
  - acosh, 204
  - actions 配置参数, 3076
  - add-msg-prefixes 配置参数, 3077
  - add-tables 配置参数, 3077
  - adminpack, 2502
  - age, 240
  - AIX
    - IPC 配置, 473
  - akeys, 2558
  - alias
    - for table name in query, 12
  - ALL, 318, 320
  - allow\_in\_place\_tablespaces配置参数, 557
  - allow\_system\_table\_mods配置参数, 557
  - ALTER AGGREGATE, 1303
  - ALTER COLLATION, 1305
  - ALTER CONVERSION, 1307
  - ALTER DATABASE, 1308
  - ALTER DEFAULT PRIVILEGES, 1310
  - ALTER DOMAIN, 1313
  - ALTER EVENT TRIGGER, 1316
  - ALTER EXTENSION, 1317
  - ALTER FOREIGN DATA WRAPPER, 1320
  - ALTER FOREIGN TABLE, 1322
  - ALTER FUNCTION, 1327
  - ALTER GROUP, 1331
  - ALTER INDEX, 1333
  - ALTER LANGUAGE, 1336
  - ALTER LARGE OBJECT, 1337
  - ALTER MATERIALIZED VIEW, 1338
  - ALTER OPERATOR, 1340
  - ALTER OPERATOR CLASS, 1342
  - ALTER OPERATOR FAMILY, 1343
  - ALTER POLICY, 1347
  - ALTER PROCEDURE, 1349
  - ALTER PUBLICATION, 1352
  - ALTER ROLE, 583, 1354
  - ALTER ROUTINE, 1358
  - ALTER RULE, 1359
  - ALTER SCHEMA, 1360
  - ALTER SEQUENCE, 1361
  - ALTER SERVER, 1364
  - ALTER SESSION, 1366
  - ALTER STATISTICS, 1367
  - ALTER SUBSCRIPTION, 1368
  - ALTER SYSTEM, 1370
  - ALTER TABLE, 1372
  - ALTER TABLESPACE, 1391
  - ALTER TEXT SEARCH CONFIGURATION, 1393
  - ALTER TEXT SEARCH DICTIONARY, 1395
  - ALTER TEXT SEARCH PARSER, 1397
  - ALTER TEXT SEARCH TEMPLATE, 1398
  - ALTER TRIGGER, 1399
  - ALTER TYPE, 1401
  - ALTER USER, 1405
  - ALTER USER MAPPING, 1406
  - ALTER VIEW, 1407
  - amcheck, 2503
  - ANALYZE, 611, 1409
  - AND (操作符), 194
  - any, 192
  - ANY, 312, 318, 320
  - anyarray, 192
  - anycompatible, 192
  - anycompatiblearray, 192
  - anycompatiblenonarray, 192
  - anycompatiblerange, 192
  - anyelement, 192
  - anyenum, 192
  - anynonarray, 192
  - anyrange, 192
  - applicable role, 914
  - application\_name配置参数, 531
  - archive\_cleanup\_command 配置参数, 512
  - archive\_command配置参数, 510
  - archive\_mode配置参数, 510
  - archive\_timeout配置参数, 511
  - area, 259
  - armor, 2764
  - ARRAY, 48
    - determination of result type, 367
  - array, 171
    - accessing, 173

- constant, 171
  - declaration, 171
  - I/O, 179
  - modifying, 175
  - searching, 177
  - array\_agg, 311, 2562
  - array\_append, 307
  - array\_cat, 307
  - array\_dims, 307
  - array\_fill, 307
  - array\_length, 307
  - array\_lower, 307
  - array\_ndims, 307
  - array\_nulls配置参数, 553
  - array\_position, 307
  - array\_positions, 307
  - array\_prepend, 307
  - array\_remove, 307
  - array\_replace, 308
  - array\_to\_json, 285
  - array\_to\_string, 308
  - array\_to\_tsvector, 265
  - array\_upper, 308
  - ascii, 206
  - asin, 203
  - asind, 203
  - asinh, 204
  - ASSERT
    - in PL/pgSQL, 1122
  - assertions
    - in PL/pgSQL, 1122
  - AT TIME ZONE, 249
  - atan, 203
  - atan2, 203
  - atan2d, 203
  - atand, 203
  - atanh, 204
  - authentication\_timeout配置参数, 495
  - auth\_delay, 2506
  - auth\_delay.milliseconds配置参数, 2506
  - auto-increment (see serial)
  - autocommit
    - ltsql, 2004
  - autovacuum\_analyze\_scale\_factor
    - 存储参数, 1541
    - 配置参数, 539
  - autovacuum\_analyze\_threshold
    - 存储参数, 1541
    - 配置参数, 539
  - autovacuum\_enabled 存储参数, 1541
  - autovacuum\_freeze\_max\_age
    - 存储参数, 1542
    - 配置参数, 539
  - autovacuum\_freeze\_min\_age存储参数, 1542
  - autovacuum\_freeze\_table\_age存储参数, 1542
  - autovacuum\_max\_workers配置参数, 538
  - autovacuum\_multixact\_freeze\_max\_age
    - 存储参数, 1542
    - 配置参数, 539
  - autovacuum\_multixact\_freeze\_min\_age 存储参数, 1542
  - autovacuum\_multixact\_freeze\_table\_age 存储参数, 1542
  - autovacuum\_naptime配置参数, 538
  - autovacuum\_vacuum\_cost\_delay
    - 存储参数, 1541
    - 配置参数, 539
  - autovacuum\_vacuum\_cost\_limit
    - 存储参数, 1541
    - 配置参数, 540
  - autovacuum\_vacuum\_insert\_scale\_factor
    - storage parameter, 1541
    - 配置参数, 539
  - autovacuum\_vacuum\_insert\_threshold
    - storage parameter, 1541
    - 配置参数, 539
  - autovacuum\_vacuum\_scale\_factor
    - 存储参数, 1541
    - 配置参数, 539
  - autovacuum\_vacuum\_threshold
    - 存储参数, 1541
    - 配置参数, 538
  - autovacuum\_work\_mem配置参数, 500
  - autovacuum配置参数, 538
  - auto\_explain, 2507
  - auto\_explain.log\_analyze配置参数, 2507
  - auto\_explain.log\_buffers配置参数, 2507
  - auto\_explain.log\_format配置参数, 2508
  - auto\_explain.log\_level配置参数, 2508
  - auto\_explain.log\_min\_duration配置参数, 2507
  - auto\_explain.log\_nested\_statements配置参数, 2508
  - auto\_explain.log\_settings配置参数, 2508
  - auto\_explain.log\_timing配置参数, 2508
  - auto\_explain.log\_triggers配置参数, 2508
  - auto\_explain.log\_verbose配置参数, 2508
  - auto\_explain.log\_wal 配置参数, 2508
  - auto\_explain.sample\_rate 配置参数, 2508
  - avals, 2558
  - average, 311
  - avg, 311
- ## B
- B-tree (see index)
  - backend\_flush\_after configuration parameter, 505
  - backslash escapes, 33
  - backslash\_quote配置参数, 554
  - backtrace\_functions 配置参数, 557
  - backup, 342, 619
  - Backup Manifest, 2339
  - base64 format, 214
  - BASE\_BACKUP, 2199
  - BEGIN, 1412
  - BETWEEN, 196
  - BETWEEN SYMMETRIC, 196

- BGWORKER\_BACKEND\_DATABASE\_CONNECTION, 1279
- BGWORKER\_SHMEM\_ACCESS, 1279
- bgwriter\_delay配置参数, 503
- bgwriter\_flush\_after 配置参数, 503
- bgwriter\_lru\_maxpages配置参数, 503
- bgwriter\_lru\_multiplier配置参数, 503
- bigint, 37, 133
- bigserial, 136
- binary data
  - functions, 212
- binary string
  - concatenation, 212
  - converting to character string, 214
  - length, 213
- bit string
  - constant, 36
  - length, 215
- bit strings
  - functions, 214
- bitmap scan, 375, 520
- bit\_and, 311
- bit\_length, 205, 212, 215
- bit\_or, 311
- BLOB (see large object)
- block\_size配置参数, 556
- bloom, 2509
- bonjour\_name配置参数, 494
- bonjour配置参数, 493
- Boolean
  - operators (see operators, logical)
  - 数据类型, 150
- bool\_and, 311
- bool\_or, 311
- booting
  - starting the server during, 469
- bound\_box, 260
- box, 260
- box (data type), 154
- brin\_desummarize\_range, 350
- brin\_summarize\_new\_values, 350
- brin\_summarize\_range, 350
- broadcast, 263
- BSD 认证, 580
- btree\_gin, 2512
- btree\_gist, 2513
- btrim, 206, 212
- bt\_index\_check, 2504
- bt\_index\_parent\_check, 2504
- bt\_metap, 3028
- bt\_page\_items, 3029, 3030
- bt\_page\_stats, 3029
- buffering 存储参数, 1480
- bytea, 140
- bytea\_output配置参数, 544
- C**
- C, 726, 812
- C++, 1002
- CALL, 1414
- canceling
  - SQL command, 764
- canopy, 2514
- cardinality, 308
- CASCADE
  - with DROP, 99, 100, 104
  - foreign key action, 66
- CASE, 299
  - determination of result type, 367
- case sensitivity
  - of SQL commands, 32
- cast
  - I/O conversion, 1442
- cbirt, 200
- ceil, 200
- ceiling, 200
- center, 259
- chained transactions, 1423, 1715
- char, 138
- character set, 547, 556
- character string
  - concatenation, 205
  - constant, 33
  - converting to binary string, 214
  - length, 205
- character varying, 138
- character\_length, 205
- char\_length, 205
- check constraint, 60
- CHECK OPTION, 1587
- CHECKPOINT, 1415
- checkpoint\_completion\_target配置参数, 509
- checkpoint\_flush\_after 配置参数, 509
- checkpoint\_timeout配置参数, 509
- checkpoint\_warning配置参数, 510
- check\_function\_bodies配置参数, 542
- check\_list, 2084
- chr, 206
- cid, 190
- cidr, 156
- circle, 155, 260
- citext, 2514
- client authentication
  - timeout during, 495
- client\_connection\_check\_interval 配置参数, 495
- client\_encoding配置参数, 547
- client\_min\_messages配置参数, 540
- clock\_timestamp, 241
- CLOSE, 1416
- CLUSTER, 1417
- clusterdb, 1787
- clustering, 633
- cluster\_name 配置参数, 536
- cmax, 68
- cmin, 67
- COALESCE, 301

- COLLATE, 47
- collation
  - in PL/pgSQL, 1091
  - in SQL functions, 981
- COLLATION FOR, 334
- column, 57
  - adding, 68
  - removing, 69
  - renaming, 70
  - system column, 67
- column data type
  - changing, 70
- col\_description, 337
- comment
  - in SQL, 38
- COMMENT, 1419
- COMMIT, 1423
- COMMIT PREPARED, 1424
- commit\_delay配置参数, 509
- commit\_siblings配置参数, 509
- comparison
  - operators, 194
- computed field, 184
- compute\_query\_id 配置参数, 537
- concat, 207
- concat\_ws, 207
- configuration
  - of recovery
    - of a standby server, 511
  - of the server
    - functions, 341
- config\_file配置参数, 492
- conjunction, 194
- connectby, 3060, 3066
- conninfo, 732
- constant, 33
- constraint, 60
  - adding, 69
  - check, 60
  - exclusion, 67
  - foreign key, 64
  - name, 60
  - NOT NULL, 62
  - primary key, 64
  - removing, 69
  - unique, 63
- constraint exclusion, 97, 524
- constraint\_exclusion配置参数, 524
- container type, 963
- CONTINUE
  - 在 PL/oraSQL 中, 1163
  - 在 PL/pgSQL 中, 1107
- continuous archiving, 619
  - in standby, 643
- convert, 214
- convert\_from, 214
- convert\_to, 214
- COPY, 9, 1425
  - with libpq, 767
- corr, 313
- correlation, 313
  - in the query planner, 448
- cos, 203
- cosd, 203
- cosh, 204
- cot, 203
- cotd, 203
- count, 311
- covariance
  - population, 313
  - sample, 314
- covar\_pop, 313
- covar\_samp, 314
- cpu\_index\_tuple\_cost配置参数, 522
- cpu\_operator\_cost配置参数, 522
- cpu\_tuple\_cost配置参数, 522
- CREATE ACCESS METHOD, 1434
- CREATE AGGREGATE, 1435
- CREATE CAST, 1442
- CREATE COLLATION, 1446
- CREATE CONVERSION, 1448
- CREATE DATABASE, 588, 1450
- CREATE DOMAIN, 1456
- CREATE EVENT TRIGGER, 1459
- CREATE EXTENSION, 1461
- CREATE FOREIGN DATA WRAPPER, 1463
- CREATE FOREIGN TABLE, 1465
- CREATE FUNCTION, 1469
- CREATE GROUP, 1476
- CREATE INDEX, 1477
- CREATE LANGUAGE, 1485
- CREATE MATERIALIZED VIEW, 1487
- CREATE OPERATOR, 1489
- CREATE OPERATOR CLASS, 1492
- CREATE OPERATOR FAMILY, 1495
- CREATE POLICY, 1496
- CREATE PROCEDURE, 1501
- CREATE PUBLICATION, 1504
- CREATE ROLE, 582, 1506
- CREATE RULE, 1510
- CREATE SCHEMA, 1513
- CREATE SEQUENCE, 1515
- CREATE SERVER, 1519
- CREATE STATISTICS, 1521
- CREATE SUBSCRIPTION, 1523
- CREATE TABLE, 7, 1526
- CREATE TABLE AS, 1555
- CREATE TABLESPACE, 591, 1558
- CREATE TEXT SEARCH CONFIGURATION, 1560
- CREATE TEXT SEARCH DICTIONARY, 1561
- CREATE TEXT SEARCH PARSER, 1563
- CREATE TEXT SEARCH TEMPLATE, 1565
- CREATE TRANSFORM, 1566
- CREATE TRIGGER, 1568
- CREATE TYPE, 1574
- CREATE USER, 1585

- 
- CREATE USER MAPPING, 1586
  - CREATE VIEW, 1587
  - createdb, 3, 589, 1790
  - createuser, 582, 1797
  - CREATE\_REPLICATION\_SLOT, 2195
  - crosstab, 3061, 3062, 3064
  - crypt, 2760
  - cstring, 192
  - CSV (Comma-Separated Values) format
    - in ltsql, 1998
  - ctid, 68
  - CTID, 1063
  - CUBE, 117
  - cube (扩展), 2516
  - cume\_dist, 317
    - hypothetical, 316
  - current\_catalog, 326
  - current\_database, 326
  - current\_date, 241
  - current\_logfiles
    - and the log\_destination configuration parameter, 526
    - and the pg\_current\_logfile function, 327
  - current\_query, 326
  - current\_role, 326
  - current\_schema, 326
  - current\_schemas, 326
  - current\_setting, 341
  - current\_time, 241
  - current\_timestamp, 241
  - current\_user, 326
  - currval, 298
  - cursor
    - CLOSE, 1416
    - DECLARE, 1593
    - FETCH, 1660
    - MOVE, 1686
    - 在 PL/oraSQL 中, 1168
    - 在 PL/pgSQL 中, 1114
  - cursor\_tuple\_fraction配置参数, 525
  - custom scan provider
    - handler for, 2263
- D**
- data type
    - constant, 37
    - container, 963
    - domain, 190
    - type cast, 46
  - database, 588
    - creating, 3
    - privilege to create, 583
  - data\_checksums配置参数, 556
  - data\_directory\_mode配置参数, 556
  - data\_directory配置参数, 491
  - data\_sync\_retry配置参数, 555
  - date, 141, 143
    - constants, 145
    - current, 250
    - 输出格式, 146
      - (see also formatting)
    - DateStyle配置参数, 546
    - date\_part, 241, 244
    - date\_trunc, 241, 248
    - dblink, 2520, 2525
      - dblink\_build\_sql\_delete, 2547
      - dblink\_build\_sql\_insert, 2545
      - dblink\_build\_sql\_update, 2548
      - dblink\_cancel\_query, 2543
      - dblink\_close, 2534
      - dblink\_connect, 2521
      - dblink\_connect\_u, 2523
      - dblink\_disconnect, 2524
      - dblink\_error\_message, 2537
      - dblink\_exec, 2528
      - dblink\_fetch, 2532
      - dblink\_get\_connections, 2536
      - dblink\_get\_notify, 2540
      - dblink\_get\_pkey, 2544
      - dblink\_get\_result, 2541
      - dblink\_is\_busy, 2539
      - dblink\_open, 2530
      - dblink\_send\_query, 2538
    - db\_user\_namespace配置参数, 495
    - deadlock
      - timeout during, 553
    - deadlock\_timeout配置参数, 553
    - DEALLOCATE, 1592
    - dearmor, 2764
    - debug\_assertions 配置参数, 556
    - debug\_deadlocks配置参数, 559
    - debug\_pretty\_print配置参数, 531
    - debug\_print\_parse配置参数, 531
    - debug\_print\_plan配置参数, 531
    - debug\_print\_rewritten配置参数, 531
    - decimal (see numeric)
    - DECLARE, 1593
    - decode, 214
    - DECODE, 304
    - decrypt, 2767
    - decrypt\_iv, 2767
    - deduplicate\_items storage parameter, 1480
    - default value, 58
      - changing, 70
    - default\_statistics\_target配置参数, 524
    - default\_tablespace配置参数, 541
    - default\_table\_access\_method 配置参数, 541
    - default\_text\_search\_config配置参数, 547
    - default\_transaction\_deferrable配置参数, 542
    - default\_transaction\_isolation配置参数, 542
    - default\_transaction\_read\_only配置参数, 542
    - deferrable transaction, 542
      - setting default, 542
    - defined, 2559
    - degrees, 200
    - delay, 254
-

- DELETE, 14, 103, 1596
    - RETURNING, 103
  - delete, 2559
  - dense\_rank, 317
    - hypothetical, 315
  - diagonal, 259
  - diameter, 259
  - dict\_int, 2549
  - dict\_xsyn, 2550
  - difference, 2554
  - digest, 2759
  - DISCARD, 1599
  - disjunction, 194
  - disk space, 610
  - DISTINCT, 10, 120
  - div, 200
  - dmetaphone, 2555
  - dmetaphone\_alt, 2555
  - DO, 1600
  - dollar quoting, 35
  - domain, 190
  - DROP ACCESS METHOD, 1602
  - DROP AGGREGATE, 1603
  - DROP CAST, 1605
  - DROP COLLATION, 1606
  - DROP CONVERSION, 1607
  - DROP DATABASE, 590, 1608
  - DROP DOMAIN, 1609
  - DROP EVENT TRIGGER, 1610
  - DROP EXTENSION, 1611
  - DROP FOREIGN DATA WRAPPER, 1612
  - DROP FOREIGN TABLE, 1613
  - DROP FUNCTION, 1614
  - DROP GROUP, 1616
  - DROP INDEX, 1617
  - DROP LANGUAGE, 1619
  - DROP MATERIALIZED VIEW, 1620
  - DROP OPERATOR, 1621
  - DROP OPERATOR CLASS, 1623
  - DROP OPERATOR FAMILY, 1625
  - DROP OWNED, 1627
  - DROP POLICY, 1628
  - DROP PROCEDURE, 1629
  - DROP PUBLICATION, 1631
  - DROP ROLE, 582, 1632
  - DROP ROUTINE, 1633
  - DROP RULE, 1634
  - DROP SCHEMA, 1635
  - DROP SEQUENCE, 1636
  - DROP SERVER, 1637
  - DROP STATISTICS, 1638
  - DROP SUBSCRIPTION, 1639
  - DROP TABLE, 8, 1640
  - DROP TABLESPACE, 1642
  - DROP TEXT SEARCH CONFIGURATION, 1643
  - DROP TEXT SEARCH DICTIONARY, 1644
  - DROP TEXT SEARCH PARSER, 1645
  - DROP TEXT SEARCH TEMPLATE, 1646
  - DROP TRANSFORM, 1647
  - DROP TRIGGER, 1648
  - DROP TYPE, 1649
  - DROP USER, 1650
  - DROP USER MAPPING, 1651
  - DROP VIEW, 1652
  - dropdb, 591, 1801
  - dropuser, 582, 1804
  - DROP\_REPLICATION\_SLOT, 2199
  - DTD, 162
  - DTrace, 687
  - dynamic loading, 552
  - dynamic\_library\_path, 984
  - dynamic\_library\_path配置参数, 552
  - dynamic\_shared\_memory\_type配置参数, 501
- ## E
- each, 2559
  - ECPG, 812
  - ecpg, 1807
  - EDITONABLE
    - in views, 1587
  - effective\_cache\_size配置参数, 523
  - effective\_io\_concurrency配置参数, 503
  - elog, 2226
  - embedded SQL
    - in C, 812
  - enabled role, 930
  - enable\_bitmapscan配置参数, 520
  - enable\_gathermerge配置参数, 520
  - enable\_hashagg配置参数, 520
  - enable\_hashjoin配置参数, 520
  - enable\_incremental\_sort配置参数, 520
  - enable\_indexonlyscan配置参数, 520
  - enable\_indexscan配置参数, 520
  - enable\_material配置参数, 520
  - enable\_mergejoin配置参数, 520
  - enable\_nestloop配置参数, 520
  - enable\_parallel\_append配置参数, 520
  - enable\_parallel\_hash配置参数, 520
  - enable\_partitionwise\_aggregate配置参数, 521
  - enable\_partitionwise\_join配置参数, 521
  - enable\_partition\_pruning配置参数, 520
  - enable\_seqscan配置参数, 521
  - enable\_sort配置参数, 521
  - enable\_tidscan配置参数, 521
  - encode, 214
  - encrypt, 2767
  - encryption
    - for specific columns, 2759
  - encrypt\_iv, 2767
  - END, 1653
  - enumerated types, 151
  - enum\_first, 255
  - enum\_last, 255
  - enum\_range, 255
  - ephemeral named relation
    - registering with SPI, 1251, 1253

- unregistering from SPI, 1252
  - ereport, 2226
  - error codes
    - libpq, 751
    - list of, 2384
  - escape format, 214
  - escape string syntax, 33
  - escape\_string\_warning配置参数, 554
  - escaping strings
    - in libpq, 757
  - event trigger
    - in C, 1052
  - event\_source配置参数, 529
  - event\_trigger, 192
  - every, 311
  - EXCEPT, 121
  - exceptions
    - in PL/pgSQL, 1110
    - 在 PL/oraSQL 中, 1168
  - exclusion constraint, 67
  - EXECUTE, 1654
  - executor, 1809
  - exist, 2559
  - EXISTS, 318
  - EXIT
    - 在 PL/oraSQL 中, 1163
    - 在 PL/pgSQL 中, 1106
  - exit\_on\_error配置参数, 555
  - exp, 200
  - EXPLAIN, 435, 1655
  - external\_pid\_file配置参数, 492
  - extract, 242, 244
  - extra\_float\_digits配置参数, 546
- F**
- factorial, 200
  - failover, 633
  - false, 150
  - family, 263
  - fast path, 765
  - fastupdate 存储参数, 1480
  - fdw\_handler, 192
  - FETCH, 1660
  - field
    - computed, 184
  - file\_fdw, 2551
  - fillfactor 存储参数, 1480, 1540
  - FILTER, 42
  - filter-msg-prefixes 配置参数, 3077
  - filter-origins 配置参数, 3076
  - filter-tables 配置参数, 3076
  - first\_value, 317
  - float4 (see real)
  - float8 (see 双精度)
  - floating-point
    - display, 546
  - floor, 200
  - force\_parallel\_mode 配置参数, 525
  - foreign data, 98
  - foreign data wrapper
    - handler for, 2243
  - foreign key, 16, 64
  - foreign table, 98
  - format, 207, 210
    - 在 PL/pgSQL 中使用, 1096
  - format-version 配置参数, 3077
  - formatting, 232
  - format\_type, 332
  - FreeBSD
    - IPC 配置, 473
    - start script, 469
    - 共享库, 991
  - from\_collapse\_limit配置参数, 525
  - FSM (see 空闲空间映射)
  - fsm\_page\_contents, 3027
  - fsync配置参数, 506
  - full\_page\_writes配置参数, 507
  - function, 194
    - default values for arguments, 974
    - internal, 983
    - RETURNS TABLE, 978
    - statistics, 356
    - user-defined, 965
      - in C, 984
    - variadic, 973
    - with SETOF, 975
  - fuzzystrmatch, 2553
- G**
- gcd, 200
  - generated column, 59, 1467, 1535
    - in triggers, 1040
  - generate\_series, 323
  - generate\_subscripts, 324
  - gen\_random\_bytes, 2768
  - gen\_random\_uuid, 269, 2768
  - gen\_salt, 2761
  - GEQO (see 遗传查询优化)
  - geqo\_effort配置参数, 523
  - geqo\_generations配置参数, 524
  - geqo\_pool\_size配置参数, 524
  - geqo\_seed配置参数, 524
  - geqo\_selection\_bias配置参数, 524
  - geqo\_threshold配置参数, 523
  - geqo配置参数, 523
  - get\_bit, 213, 216
  - get\_byte, 213
  - get\_current\_ts\_config, 265
  - get\_raw\_page, 3026
  - GIN (see index)
  - gin\_clean\_pending\_list, 350
  - gin\_fuzzy\_search\_limit配置参数, 552
  - gin\_leafpage\_items, 3031
  - gin\_metapage\_info, 3031
  - gin\_page\_opaque\_info, 3031
  - gin\_pending\_list\_limit

- 存储参数, 1480
  - 配置参数, 544
  - GiST (see index)
  - GOTO
    - in PL/oraSQL, 1164
  - GRANT, 70, 1664
  - GREATEST, 302
    - determination of result type, 367
  - GROUP BY, 13, 114
  - GROUPING, 316
  - GROUPING SETS, 117
  - gssapi, 486
  - GSSAPI, 572
    - with libpq, 737
  - GUID, 161
- H**
- hash (see index)
  - hash\_bitmap\_info, 3032
  - hash\_mem\_multiplier 配置参数, 500
  - hash\_metapage\_info, 3033
  - hash\_page\_items, 3032
  - hash\_page\_stats, 3032
  - hash\_page\_type, 3032
  - has\_any\_column\_privilege, 329
  - has\_column\_privilege, 329
  - has\_database\_privilege, 329
  - has\_foreign\_data\_wrapper\_privilege, 329
  - has\_function\_privilege, 329
  - has\_language\_privilege, 329
  - has\_schema\_privilege, 329
  - has\_sequence\_privilege, 329
  - has\_server\_privilege, 329
  - has\_tablespace\_privilege, 329
  - has\_table\_privilege, 329
  - has\_type\_privilege, 329
  - HAVING, 13, 116
  - hba\_file配置参数, 492
  - heap\_page\_items, 3027
  - heap\_page\_item\_attrs, 3028
  - heap\_tuple\_infomask\_flags, 3028
  - height, 259
  - hex format, 214
  - hmac, 2760
  - host, 263
  - host name, 734
  - hostmask, 263
  - hot\_standby\_feedback配置参数, 518
  - hot\_standby配置参数, 517
  - HP-UX
    - IPC 配置, 473
    - 共享库, 991
  - hstore, 2556, 2558
  - hstore\_to\_array, 2558
  - hstore\_to\_json, 2558
  - hstore\_to\_jsonb, 2559
  - hstore\_to\_jsonb\_loose, 2559
  - hstore\_to\_json\_loose, 2559
  - hstore\_to\_matrix, 2558
  - huge\_pages配置参数, 498
  - hypothetical-set aggregate
    - 内建, 315
- I**
- icount, 2564
  - ICU, 596, 1446
  - ident, 574
  - identifier
    - length, 32
    - syntax of, 31
  - IDENTIFY\_SYSTEM, 2195
  - ident\_file配置参数, 492
  - idle\_in\_transaction\_session\_timeout 配置参数, 543
  - idx, 2564
  - IFNULL, 301
  - ignore\_checksum\_failure配置参数, 559
  - ignore\_invalid\_pages 配置参数, 560
  - ignore\_system\_indexes配置参数, 558
  - IMMUTABLE, 982
  - IMPORT FOREIGN SCHEMA, 1669
  - IN, 318, 320
  - INCLUDE
    - in index definitions, 380
  - include
    - in configuration file, 490
  - include-column-positions 配置参数, 3076
  - include-default 配置参数, 3076
  - include-domain-data-type 配置参数, 3076
  - include-lsn 配置参数, 3076
  - include-not-null 配置参数, 3076
  - include-origin 配置参数, 3075
  - include-pk 配置参数, 3075
  - include-schemas 配置参数, 3076
  - include-timestamp 配置参数, 3075
  - include-transaction 配置参数, 3075
  - include-type-oids 配置参数, 3076
  - include-types 配置参数, 3076
  - include-typmod 配置参数, 3076
  - include-xids 配置参数, 3075
  - include\_dir
    - in configuration file, 490
  - include\_if\_exists
    - in configuration file, 490
  - index, 2781
    - and ORDER BY, 375
    - B-tree, 373
    - B-Tree, 2287
    - building concurrently, 1481
    - combining multiple indexes, 375
    - on expressions, 376
    - for user-defined data type, 1017
    - GIN, 374
    - GiST, 373, 2293
    - hash, 373
    - multicolumn, 374



- partial, 377
  - rebuilding concurrently, 1703
  - unique, 376
  - 检查用量, 384
  - index scan, 520
  - indexam
    - 索引访问方法, 2272
  - index\_am\_handler, 192
  - inet\_client\_addr, 326
  - inet\_client\_port, 326
  - inet\_merge, 263
  - inet\_same\_family, 263
  - inet\_server\_addr, 326
  - inet\_server\_port, 327
  - inet (数据类型), 155
  - infinity
    - numeric (数据类型), 134
  - inheritance, 21, 84
  - initcap, 207
  - initdb, 466, 2028
  - INSERT, 8, 101, 1671
    - RETURNING, 103
  - instr 函数, 1144
  - int2 (see smallint)
  - int4 (see 整数)
  - int8 (see bigint)
  - intagg, 2562
  - intarray, 2563
  - integer, 37
  - integer\_datetimes配置参数, 556
  - internal, 192
  - INTERSECT, 121
  - interval, 148
    - 输出格式, 150
    - (see also formatting)
  - IntervalStyle配置参数, 546
  - intset, 2564
  - int\_array\_aggregate, 2562
  - int\_array\_enum, 2562
  - in\_range support functions, 2288
  - IS DISTINCT FROM, 197, 320
  - IS DOCUMENT, 274
  - IS FALSE, 197
  - IS NOT DISTINCT FROM, 197, 320
  - IS NOT DOCUMENT, 274
  - IS NOT FALSE, 197
  - IS NOT NULL, 197
  - IS NOT TRUE, 197
  - IS NOT UNKNOWN, 197
  - IS NULL, 197, 555
  - IS TRUE, 197
  - IS UNKNOWN, 197
  - isclosed, 259
  - isempty, 310
  - isfinite, 242
  - isn, 2566
  - ISNULL, 197
  - isn\_weak, 2568
  - isopen, 259
  - is\_valid, 2568
- ## J
- JIT, 709
  - jit\_above\_cost配置参数, 523
  - jit\_debugging\_support配置参数, 560
  - jit\_dump\_bitcode配置参数, 560
  - jit\_expressions配置参数, 560
  - jit\_inline\_above\_cost配置参数, 523
  - jit\_optimize\_above\_cost配置参数, 523
  - jit\_profiling\_support配置参数, 560
  - jit\_provider配置参数, 552
  - jit\_tuple deforming配置参数, 560
  - jit配置参数, 525
  - join
    - outer, 11, 106
    - self, 12
  - join\_collapse\_limit配置参数, 525
  - JSON, 163
    - 函数和操作符, 283
  - JSONB, 163
  - jsonb
    - containment, 165
    - existence, 165
    - indexes on, 167
  - jsonb\_agg, 311
  - jsonb\_array\_elements, 286
  - jsonb\_array\_elements\_text, 286
  - jsonb\_array\_length, 286
  - jsonb\_build\_array, 285
  - jsonb\_build\_object, 285
  - jsonb\_each, 287
  - jsonb\_each\_text, 287
  - jsonb\_extract\_path, 287
  - jsonb\_extract\_path\_text, 287
  - jsonb\_insert, 289
  - jsonb\_object, 286
  - jsonb\_object\_agg, 311
  - jsonb\_object\_keys, 287
  - jsonb\_path\_exists, 289
  - jsonb\_path\_exists\_tz, 290
  - jsonb\_path\_match, 290
  - jsonb\_path\_match\_tz, 290
  - jsonb\_path\_query, 290
  - jsonb\_path\_query\_array, 290
  - jsonb\_path\_query\_array\_tz, 290
  - jsonb\_path\_query\_first, 290
  - jsonb\_path\_query\_first\_tz, 290
  - jsonb\_path\_query\_tz, 290
  - jsonb\_populate\_record, 287
  - jsonb\_populate\_recordset, 288
  - jsonb\_pretty, 291
  - jsonb\_set, 289
  - jsonb\_set\_lax, 289
  - jsonb\_strip\_nulls, 289
  - jsonb\_to\_record, 288
  - jsonb\_to\_recordset, 288

- jsonb\_to\_tsvector, 266
  - jsonb\_typeof, 291
  - jsonpath, 169
  - json\_agg, 311
  - json\_array\_elements, 286
  - json\_array\_elements\_text, 286
  - json\_array\_length, 286
  - json\_build\_array, 285
  - json\_build\_object, 285
  - json\_each, 286
  - json\_each\_text, 287
  - json\_extract\_path, 287
  - json\_extract\_path\_text, 287
  - json\_object, 286
  - json\_object\_agg, 311
  - json\_object\_keys, 287
  - json\_populate\_record, 287
  - json\_populate\_recordset, 288
  - json\_strip\_nulls, 289
  - json\_to\_record, 288
  - json\_to\_recordset, 288
  - json\_to\_tsvector, 266
  - json\_typeof, 291
  - Just-In-Time compilation (see JIT)
  - justify\_days, 242
  - justify\_hours, 242
  - justify\_interval, 242
- K**
- key word
    - list of, 2401
    - syntax of, 31
  - krb\_caseins\_users配置参数, 495
  - krb\_server\_keyfile配置参数, 495
- L**
- lag, 317
  - language\_handler, 192
  - large object, 801
  - lastval, 298
  - last\_value, 317
  - LATERAL
    - 在FROM子句中, 112
  - lca, 2781
  - lcm, 200
  - lc\_collate配置参数, 556
  - lc\_ctype配置参数, 556
  - lc\_messages配置参数, 547
  - lc\_monetary配置参数, 547
  - lc\_numeric配置参数, 547
  - lc\_time配置参数, 547
  - LDAP, 575
  - LDAP 连接参数查找, 784
  - lead, 317
  - LEAST, 302
    - determination of result type, 367
  - left, 207
    - length, 207, 213, 215, 259, 265
      - of a binary string (see binary strings, length)
      - of a character string (see character string, length)
    - length(tsvector), 397
    - levenshtein, 2554
    - levenshtein\_less\_equal, 2554
    - libpq, 726
      - single-row mode, 763
    - libpq-fe.h, 726, 740
    - libpq-int.h, 740
    - lightdb, 3, 468, 2075
    - lightdb user, 466
    - lightdb.auto.conf, 489
    - lightdb.conf, 488
    - lightdbService, 1810
    - lightdb\_aborted\_rollback配置参数, 561
    - lightdb\_archive\_dir配置参数, 511
    - lightdb\_archive\_retention\_size配置参数, 511
    - lightdb\_ascii\_zero\_store\_value配置参数, 545
    - lightdb\_best\_practice\_level 配置参数, 2733
    - lightdb\_best\_practice\_mode 配置参数, 2732
    - lightdb\_convert\_chinese\_char配置参数, 546
    - lightdb\_cursor\_after\_commit配置参数, 562
    - lightdb\_dblevel\_syntax\_compatible\_type配置参数, 544
    - lightdb\_enable\_default\_tablespace配置参数 , 561
    - lightdb\_enable\_indexautoanalyze配置参数, 524
    - lightdb\_enable\_sorted\_group配置参数, 520
    - lightdb\_encryption\_key\_command配置参数, 549
    - lightdb\_external\_virtual\_ip配置参数, 561
    - lightdb\_ignore\_template\_compatible\_type\_mismatchconfiguration parameter, 546
    - lightdb\_keep\_temp\_file configuration parameter, 500
    - lightdb\_log\_retention\_age配置参数, 528
    - lightdb\_mysql\_lower\_case\_table\_names配置参数, 545
    - lightdb\_only\_full\_group\_by配置参数, 550
    - lightdb\_oracle\_sql\_mode配置参数, 562
    - lightdb\_sql\_mode配置参数, 561
    - lightdb\_syntax\_compatible\_type配置参数, 544
    - lightdb\_tsearch\_non\_stopwords配置参数, 547
    - lightdb\_tsearch\_word\_superpose配置参数, 548
  - LIKE, 217
    - 与区域, 594
  - LIKE\_REGEX, 231
    - in SQL/JSON, 297
  - LIMIT, 123
  - line, 153, 260
  - Linux
    - IPC 配置, 474
    - 共享库, 992
    - 启动脚本, 469
  - LISTEN, 1680
  - listen\_addresses配置参数, 492

- ln, 200
- lo, 2569
- LOAD, 1682
- locale, 467, 593
- localtime, 242
- localtimestamp, 242
- local\_preload\_libraries配置参数, 551
- LOCK, 428, 1683
- lock
  - monitoring, 680
- lock\_timeout配置参数, 543
- log, 201
- log10, 201
- Logging
  - current\_logfiles file and the pg\_current\_logfile function, 327
  - pg\_current\_logfile function, 327
- logging\_collector配置参数, 526
- logical\_decoding\_work\_mem 配置参数, 500
- login privilege, 583
- log\_autovacuum\_min\_duration
  - 存储参数, 1542
  - 配置参数, 538
- log\_btree\_build\_stats配置参数, 559
- log\_checkpoints配置参数, 531
- log\_connections配置参数, 531
- log\_destination配置参数, 526
- log\_directory配置参数, 527
- log\_disconnections配置参数, 531
- log\_duration配置参数, 531
- log\_error\_verbosity配置参数, 532
- log\_executor\_stats配置参数, 538
- log\_filename配置参数, 527
- log\_file\_mode配置参数, 527
- log\_hostname配置参数, 532
- log\_line\_prefix配置参数, 532
- log\_lock\_waits配置参数, 534
- log\_min\_duration\_sample 配置参数, 530
- log\_min\_duration\_statement配置参数, 529
- log\_min\_error\_statement配置参数, 529
- log\_min\_messages配置参数, 529
- log\_parameter\_max\_length 配置参数, 534
- log\_parameter\_max\_length\_on\_error 配置参数, 534
- log\_parser\_stats配置参数, 538
- log\_planner\_stats配置参数, 538
- log\_recovery\_conflict\_waits 配置参数, 534
- log\_replication\_commands 配置参数, 535
- log\_rotation\_age配置参数, 528
- log\_rotation\_size配置参数, 528
- log\_statement\_sample\_rate 配置参数, 530
- log\_statement\_stats配置参数, 538
- log\_statement配置参数, 534
- log\_temp\_files配置参数, 535
- log\_timezone配置参数, 535
- log\_transaction\_sample\_rate 配置参数, 530
- log\_truncate\_on\_rotation配置参数, 528
- loop
  - 在 PL/pgSQL 中, 1106
- lower, 205, 310
  - 与区域, 594
- lower\_inc, 310
- lower\_inf, 310
- lo\_close, 805
- lo\_compat\_privileges配置参数, 554
- lo\_creat, 801, 805
- lo\_create, 802
- lo\_export, 802, 805
- lo\_from\_bytea, 805
- lo\_get, 805
- lo\_import, 802, 805
- lo\_import\_with\_oid, 802
- lo\_lseek, 803
- lo\_lseek64, 804
- lo\_open, 803
- lo\_put, 805
- lo\_read, 803
- lo\_tell, 804
- lo\_tell64, 804
- lo\_truncate, 804
- lo\_truncate64, 804
- lo\_unlink, 805, 805
- lo\_write, 803
- lpad, 207
- lseg, 154, 260
- LSN, 702
- LTAPPNAME, 783
- ltbench, 1958
- ltcenter, 1975
- LTCHANNELBINDING, 782
- LTCLIENTENCODING, 783
- ltcluster, 2590
- LTCONNECT\_TIMEOUT, 783
- ltcrypto, 2759
- LTDATA, 466
- LTDATABASE, 782
- LTDATESTYLE, 783
- ltdts\_mysql, 1846
- ltdts\_ora, 1843
- ltdts\_recvlogical, 1839
- ltfce, 2771
- LTGEQO, 783
- LTGSSENCMODE, 783
- LTGSSLIB, 783
- LTHOST, 782
- LTHOSTADDR, 782
- LTKRBSRVNAME, 783
- ltldr, 1850
- LTLOCALEDIR, 783
- LTOPTIONS, 783
- LTPASSFILE, 782
- LTPASSWORD, 782
- LTPORT, 782
- ltree, 2778
- ltree2text, 2781
- LTREQUIREPEER, 783

- 
- LTREQUIRESSL, 783
  - ltrim, 207
  - ltrowlocks, 2784
  - LTSERVICE, 782
  - LTSERVICEFILE, 783
  - ltsql, 5, 1977
  - LTSSLCERT, 783
  - LTSSLCOMPRESSION, 783
  - LTSSLCRL, 783
  - LTSSLKEY, 783
  - LTSSLMAXPROTOCOLVERSION, 783
  - LTSSLMINPROTOCOLVERSION, 783
  - LTSSLMODE, 783
  - LTSSLROOTCERT, 783
  - ltstattuple, 2785
  - LTSYSCONFDIR, 783
  - LTTARGETSESSIONATTRS, 783
  - LTTZ, 783
  - ltuldr, 1853
  - LTUSER, 782
  - lt\_archivecleanup, 2042
  - lt\_attri\_ext, 2108
  - lt\_basebackup, 1856
  - lt\_buffercache, 2570
  - lt\_bulkload (批量加载), 2572
  - lt\_check, 2040
  - lt\_checksums, 2044
  - lt\_config, 1863
    - with ecpg, 868
    - with libpq, 790
    - with user-defined C functions, 990
  - lt\_controldata, 2046
  - lt\_createdb, 1794
  - lt\_cron, 2590
  - lt\_ctl, 466, 468, 2047
  - lt\_database\_ext, 2108
  - lt\_distributed\_dump, 1821
  - lt\_distributed\_probackup, 1815
  - lt\_distributed\_probackup.py, 1815
  - lt\_distributed\_restore, 1832
  - lt\_dump, 1866
  - lt\_dumpall, 1879
    - use during upgrade, 480
  - lt\_freemap, 2594
  - lt\_global\_temp\_table\_ext, 2108
  - lt\_hba.conf, 563
  - lt\_hint\_plan, 2595
  - lt\_http, 3087
  - lt\_ident.conf, 570
  - lt\_initdb, 2033
  - lt\_isready, 1886
  - lt\_list\_temp\_file, 2747
  - lt\_ope, 2651
  - lt\_package, 2109
  - lt\_partition\_ext, 2109
  - lt\_patch, 2038, 2668
    - function, 2669
  - lt\_prewarm, 2669
    - lt\_prewarm.autoprewarm 配置参数, 2670
    - lt\_prewarm.autoprewarm\_interval 配置参数, 2670
    - lt\_prewarm.lt\_autoprewarm\_dbs 配置参数, 2670
    - lt\_prewarm.lt\_autoprewarm\_indexes 配置参数, 2671
    - lt\_prewarm.lt\_autoprewarm\_tables 配置参数, 2670
  - lt\_probackup, 1888
  - lt\_proc\_ext, 2110
  - lt\_profile (轻量级开发工具配置文件), 2673
  - lt\_qualstats, 2723
  - lt\_receivewal, 1938
  - lt\_recvlogical, 1942
  - lt\_remove\_temp\_file, 2747
  - lt\_resetwal, 2052
  - lt\_restore, 1946
  - lt\_rewind, 2055
  - lt\_ring\_buffers配置参数, 499
  - lt\_show\_plans, 2727
    - lt\_show\_plans.max\_plan\_length配置参数, 2727
    - lt\_show\_plans.plan\_format配置参数, 2727
    - lt\_show\_plans.show\_level配置参数, 2727
  - lt\_sm, 2729
  - lt\_sql\_inspect, 2732
    - lt\_sql\_inspect.secure\_enable\_db 配置参数, 2735
    - lt\_sql\_inspect.secure\_enable\_time 配置参数, 2735
    - lt\_sql\_inspect.secure\_mode 配置参数, 2735
  - lt\_standby, 3101
  - lt\_standby\_forward, 2739
    - lt\_standby\_forward.enable配置参数, 2739
  - lt\_statement\_rollback\_enable配置参数, 561
  - lt\_stat\_activity, 2740
  - lt\_stat\_statements, 2740
  - lt\_t3sdk, 3085
  - lt\_tempfile, 2746
  - lt\_test\_fsync, 2059
  - lt\_test\_timing, 2060
  - lt\_trgm, 2747
    - lt\_trgm.similarity\_threshold配置参数, 2749
    - lt\_trgm.strict\_word\_similarity\_threshold配置参数, 2749
    - lt\_trgm.word\_similarity\_threshold配置参数, 2749
  - lt\_type\_ext, 2110
  - lt\_upgrade, 2063
  - lt\_upgrade\_check, 2071
  - lt\_variable, 2111
  - lt\_verifybackup, 1955
  - lt\_visibility, 2752
  - lt\_waldump, 2073
  - lt\_walminer, 2789
- M**
- MAC address (EUI-64 format) (see macaddr)
-

- macaddr8 (data type), 157
  - macaddr8\_set7bit, 264
  - macaddr (数据类型), 156
  - macOS
    - IPC 配置, 474
    - 共享库, 992
  - MAC地址 (see macaddr)
  - magic block, 984
  - maintenance\_io\_concurrency 配置参数, 504
  - maintenance\_work\_mem配置参数, 500
  - makeaclitem, 331
  - make\_date, 242
  - make\_interval, 242
  - make\_time, 243
  - make\_timestamp, 243
  - make\_timestamptz, 243
  - make\_valid, 2568
  - masklen, 263
  - materialized views, 2164
  - max, 311
  - max\_connections配置参数, 492
  - max\_files\_per\_process配置参数, 501
  - max\_function\_args配置参数, 556
  - max\_identifier\_length配置参数, 556
  - max\_index\_keys配置参数, 556
  - max\_locks\_per\_transaction配置参数, 553
  - max\_logical\_replication\_workers配置参数, 519
  - max\_parallel\_maintenance\_workers配置参数, 504
  - max\_parallel\_workers\_per\_gather 配置参数, 504
  - max\_parallel\_workers配置参数, 504
  - max\_pred\_locks\_per\_page配置参数, 553
  - max\_pred\_locks\_per\_relation配置参数, 553
  - max\_pred\_locks\_per\_transaction配置参数, 553
  - max\_prepared\_transactions配置参数, 499
  - max\_replication\_slots配置参数, 514
  - max\_slot\_wal\_keep\_size 配置参数, 515
  - max\_stack\_depth配置参数, 501
  - max\_standby\_archive\_delay配置参数, 517
  - max\_standby\_streaming\_delay配置参数, 517
  - max\_sync\_workers\_per\_subscription配置参数, 519
  - max\_wal\_senders配置参数, 514
  - max\_wal\_size 配置参数, 510
  - max\_worker\_processes配置参数, 504
  - md5, 207, 213
  - MD5, 572
  - median, 44
    - (see also percentile)
  - memory context
    - in SPI, 1262
  - memory overcommit, 476
  - metaphone, 2555
  - min, 312
  - min\_parallel\_index\_scan\_size配置参数, 522
  - min\_parallel\_table\_scan\_size配置参数, 522
  - min\_scale, 201
  - min\_wal\_size 配置参数, 510
  - mod, 201
  - mode
    - statistical, 315
  - MOVE, 1686
  - MultiXactId, 615
  - MVCC, 423
  - myfce, 2789
  - mysql, 2351
  - mysql\_fdw, 2844
- ## N
- name
    - qualified, 80
    - syntax of, 31
    - unqualified, 81
  - NaN (see 不是一个数字)
  - negation, 194
  - NetBSD
    - IPC 配置, 473
    - 共享库, 992
    - 启动脚本, 470
  - netmask, 263
  - network, 263
    - 数据类型, 155
  - nextval, 298
  - NFS, 468
  - nlevel, 2781
  - nls\_date\_format, 252
  - nls\_date\_format配置参数, 549
  - nls\_timestamp\_format, 252
  - nls\_timestamp\_format配置参数, 550
  - non-durable, 454
  - nonblocking connection, 728
  - normalize, 205
  - normalized, 205
  - normal\_rand, 3060
  - NOT IN, 318, 320
  - not-null constraint, 62
  - notation
    - functions, 54
  - notice processing
    - in libpq, 775
  - notice processor, 775
  - notice receiver, 775
  - NOTIFY, 1692
    - in libpq, 766
  - NOTNULL, 197
  - NOT (操作符), 194
  - now, 243
  - npoints, 259
  - nth\_value, 317
  - ntile, 317
  - null value
    - with check constraints, 61
    - comparing, 197
    - default value, 58
    - in libpq, 755

with unique constraints, 64  
 NULLIF, 302  
 number  
   constant, 36  
 numeric, 37  
 numnode, 265, 398  
 num\_nonnulls, 198  
 num\_nulls, 198  
 NVL, 301  
 NVL and NVL2, 302

**O**

obj\_description, 337  
 octet\_length, 205, 205, 212, 215  
 OFFSET, 123  
 oid, 190  
 OID  
   in libpq, 757  
 oid2name, 3094  
 old\_snapshot\_threshold 配置参数, 505  
 ON CONFLICT, 1671  
 ONLY, 106  
 OOM, 476  
 OpenBSD  
   IPC 配置, 473  
   共享库, 992  
   启动脚本, 469  
 operator, 194  
   logical, 194  
   precedence, 38  
   syntax, 37  
 operator class, 382  
 operator family, 382, 1023  
 operator\_precedence\_warning 配置参数, 554  
 optimization information  
   for functions, 1003  
   for operators, 1014  
 Oracle  
   从 PL/SQL 移植到 PL/pgSQL, 1137  
 oracle\_fdw, 2847  
 orafce, 2857  
 ORDER BY, 10, 122  
   与区域, 594  
 ordered-set aggregate  
   内建, 315  
 ordering operator, 1025  
 ordinality, 325  
 OR (操作符), 194  
 OVER clause, 44  
 overcommit, 476  
 OVERLAPS, 243  
 overlay, 205, 212, 215  
 overloading  
   functions, 981  
 owner, 70

**P**

pageinspect, 3026  
 page\_checksum, 3027  
 page\_header, 3027  
 pallo, 990  
 PAM, 579  
 parallel\_leader\_participation 配置参数, 525  
 parallel\_setup\_cost 配置参数, 522  
 parallel\_tuple\_cost 配置参数, 522  
 parallel\_workers 存储参数, 1540  
 parse\_ident, 207  
 partition pruning, 96  
 partitioned table, 87  
 partitioning, 87  
 password, 583  
   of the superuser, 467  
 passwordcheck, 3033  
 password\_encryption 配置参数, 495  
 path, 260  
   for schemas, 540  
 path (data type), 154  
 pattern matching, 216  
 patterns  
   in ltsql and lt\_dump, 2003  
 pclose, 259  
 peer, 575  
 percentile  
   continuous, 315  
   discrete, 315  
 percent\_rank, 317  
   hypothetical, 315  
 permission (see privilege)  
 pfree, 990  
 PGcancel, 764  
 PGconn, 726  
 PGEventProc, 778  
 pgp\_armor\_headers, 2764  
 pgp\_key\_id, 2763  
 pgp\_pub\_decrypt, 2763  
 pgp\_pub\_decrypt\_bytea, 2763  
 pgp\_pub\_encrypt, 2763  
 pgp\_pub\_encrypt\_bytea, 2763  
 pgp\_sym\_decrypt, 2763  
 pgp\_sym\_decrypt\_bytea, 2763  
 pgp\_sym\_encrypt, 2763  
 pgp\_sym\_encrypt\_bytea, 2763  
 PGresult, 749  
 pgrowlocks, 2784  
 pgstatginindex, 2787  
 pgstathashindex, 2788  
 pgstatindex, 2786  
 pgstattuple, 2785  
 pgstattuple\_approx, 2788  
 pgxs, 1034  
 pg\_advisory\_lock, 352  
 pg\_advisory\_lock\_shared, 352  
 pg\_advisory\_unlock, 352

---

pg\_advisory\_unlock\_all, 352  
pg\_advisory\_unlock\_shared, 352  
pg\_advisory\_xact\_lock, 352  
pg\_advisory\_xact\_lock\_shared, 352  
pg\_aggregate, 2111  
pg\_am, 2113  
pg\_amop, 2113  
pg\_amproc, 2114  
pg\_attrdef, 2115  
pg\_attribute, 2115  
pg\_authid, 2117  
pg\_auth\_members, 2118  
pg\_available\_extensions, 2158  
pg\_available\_extension\_versions, 2158  
pg\_backend\_pid, 327  
pg\_backup\_start\_time, 343  
pg\_blocking\_pids, 327  
pg\_bufferscache\_pages, 2570  
pg\_cancel\_backend, 341  
pg\_cast, 2118  
pg\_class, 2119  
pg\_client\_encoding, 207  
pg\_collation, 2121  
pg\_collation\_actual\_version, 349  
pg\_collation\_is\_visible, 331  
PG\_COLOR, 3124  
PG\_COLORS, 3124  
pg\_column\_size, 348  
pg\_config, 2159  
pg\_conf\_load\_time, 327  
pg\_constraint, 2122  
pg\_control\_checkpoint, 339  
pg\_control\_init, 339  
pg\_control\_recovery, 339  
pg\_control\_system, 339  
pg\_conversion, 2124  
pg\_conversion\_is\_visible, 331  
pg\_copy\_logical\_replication\_slot, 346  
pg\_copy\_physical\_replication\_slot, 346  
pg\_create\_logical\_replication\_slot, 346  
pg\_create\_physical\_replication\_slot, 346  
pg\_create\_restore\_point, 342  
pg\_current\_logfile, 327  
pg\_current\_snapshot, 338  
pg\_current\_wal\_flush\_lsn, 342  
pg\_current\_wal\_insert\_lsn, 342  
pg\_current\_wal\_lsn, 342  
pg\_current\_xact\_id, 337  
pg\_current\_xact\_id\_if\_assigned, 337  
pg\_cursors, 2159  
pg\_database, 590, 2124  
pg\_database\_size, 348  
pg\_db\_role\_setting, 2125  
pg\_ddl\_command, 192  
pg\_default\_acl, 2126  
pg\_depend, 2126  
pg\_describe\_object, 336  
pg\_description, 2128  
pg\_drop\_replication\_slot, 346  
pg\_enum, 2128  
pg\_event\_trigger, 2129  
pg\_event\_trigger\_ddl\_commands, 354  
pg\_event\_trigger\_dropped\_objects, 354  
pg\_event\_trigger\_table\_rewrite\_oid, 356  
pg\_event\_trigger\_table\_rewrite\_reason, 356  
pg\_export\_snapshot, 345  
pg\_extension, 2129  
pg\_extension\_config\_dump, 1031  
pg\_filenode\_relation, 349  
pg\_file\_rename, 2503  
pg\_file\_settings, 2160  
pg\_file\_sync, 2503  
pg\_file\_unlink, 2503  
pg\_file\_write, 2503  
pg\_foreign\_data\_wrapper, 2130  
pg\_foreign\_server, 2130  
pg\_foreign\_table, 2131  
pg\_freespace, 2594  
pg\_function\_is\_visible, 331  
pg\_get\_constraintdef, 332  
pg\_get\_expr, 332  
pg\_get\_functiondef, 332  
pg\_get\_function\_arguments, 332  
pg\_get\_function\_identity\_arguments, 332  
pg\_get\_function\_result, 332  
pg\_get\_indexdef, 332  
pg\_get\_keywords, 332  
pg\_get\_object\_address, 337  
pg\_get\_ruledef, 333  
pg\_get\_serial\_sequence, 333  
pg\_get\_statisticsobjdef, 333  
pg\_get\_triggerdef, 333  
pg\_get\_userbyid, 333  
pg\_get\_viewdef, 333  
pg\_group, 2161  
pg\_has\_role, 330  
pg\_hba\_file\_rules, 2161  
pg\_identify\_object, 336  
pg\_identify\_object\_as\_address, 336  
pg\_import\_system\_collations, 349  
pg\_index, 2131  
pg\_indexam\_has\_property, 333  
pg\_indexes, 2162  
pg\_indexes\_size, 348  
pg\_index\_column\_has\_property, 333  
pg\_index\_has\_property, 333  
pg\_inherits, 2132  
pg\_init\_privs, 2133  
pg\_is\_in\_backup, 343  
pg\_is\_in\_recovery, 344  
pg\_is\_other\_temp\_schema, 327  
pg\_is\_wal\_replay\_paused, 344  
pg\_jit\_available, 327  
pg\_language, 2133  
pg\_largeobject, 2134  
pg\_largeobject\_metadata, 2134

---

- 
- pg\_last\_committed\_xact, 339
  - pg\_last\_wal\_receive\_lsn, 344
  - pg\_last\_wal\_replay\_lsn, 344
  - pg\_last\_xact\_replay\_timestamp, 344
  - pg\_listening\_channels, 327
  - pg\_locks, 2162
  - pg\_logdir\_ls, 2503
  - pg\_logical\_emit\_message, 347
  - pg\_logical\_slot\_get\_binary\_changes, 346
  - pg\_logical\_slot\_get\_changes, 346
  - pg\_logical\_slot\_peek\_binary\_changes, 347
  - pg\_logical\_slot\_peek\_changes, 346
  - pg\_lsn, 192
  - pg\_ls\_archive\_statusdir, 351
  - pg\_ls\_dir, 350
  - pg\_ls\_logdir, 351
  - pg\_ls\_tmpdir, 351
  - pg\_ls\_waldir, 351
  - pg\_matviews, 2164
  - pg\_mcv\_list\_items, 356
  - pg\_my\_temp\_schema, 327
  - pg\_namespace, 2135
  - pg\_notification\_queue\_usage, 327
  - pg\_notify, 1693
  - pg\_opclass, 2135
  - pg\_opclass\_is\_visible, 331
  - pg\_operator, 2136
  - pg\_operator\_is\_visible, 331
  - pg\_opfamily, 2137
  - pg\_opfamily\_is\_visible, 331
  - pg\_options\_to\_table, 333
  - pg\_partitioned\_table, 2137
  - pg\_partition\_ancestors, 349
  - pg\_partition\_root, 349
  - pg\_partition\_tree, 349
  - pg\_policies, 2165
  - pg\_policy, 2138
  - pg\_postmaster\_start\_time, 327
  - pg\_prepared\_statements, 2165
  - pg\_prepared\_xacts, 2166
  - pg\_proc, 2138
  - pg\_promote, 344
  - pg\_publication, 2140
  - pg\_publication\_rel, 2141
  - pg\_publication\_tables, 2166
  - pg\_range, 2141
  - pg\_read\_binary\_file, 351
  - pg\_read\_file, 351
  - pg\_relation\_filenode, 349
  - pg\_relation\_filepath, 349
  - pg\_relation\_size, 348
  - pg\_reload\_conf, 341
  - pg\_relpages, 2788
  - pg\_replication\_origin, 2142
  - pg\_replication\_origin\_advance, 347
  - pg\_replication\_origin\_create, 347
  - pg\_replication\_origin\_drop, 347
  - pg\_replication\_origin\_oid, 347
  - pg\_replication\_origin\_progress, 347
  - pg\_replication\_origin\_session\_is\_setup, 347
  - pg\_replication\_origin\_session\_progress, 347
  - pg\_replication\_origin\_session\_reset, 347
  - pg\_replication\_origin\_session\_setup, 347
  - pg\_replication\_origin\_status, 2167
  - pg\_replication\_origin\_xact\_reset, 347
  - pg\_replication\_origin\_xact\_setup, 347
  - pg\_replication\_slots, 2167
  - pg\_replication\_slot\_advance, 347
  - pg\_rewrite, 2142
  - pg\_roles, 2168
  - pg\_rotate\_logfile, 341
  - pg\_rules, 2169
  - pg\_safe\_snapshot\_blocking\_pids, 327
  - pg\_seclabel, 2143
  - pg\_seclabels, 2169
  - pg\_sequence, 2143
  - pg\_sequences, 2170
  - pg\_service.conf, 784
  - pg\_settings, 2171
  - pg\_shadow, 2173
  - pg\_shdepend, 2144
  - pg\_shdescription, 2145
  - pg\_shmem\_allocations, 2173
  - pg\_shseclabel, 2145
  - pg\_size\_bytes, 348
  - pg\_size\_pretty, 348
  - pg\_sleep, 254
  - pg\_sleep\_for, 254
  - pg\_sleep\_until, 254
  - pg\_snapshot\_xip, 338
  - pg\_snapshot\_xmax, 338
  - pg\_snapshot\_xmin, 338
  - pg\_start\_backup, 342
  - pg\_statio\_all\_indexes, 656, 677
  - pg\_statio\_all\_sequences, 656, 677
  - pg\_statio\_all\_tables, 656, 676
  - pg\_statio\_sys\_indexes, 656
  - pg\_statio\_sys\_sequences, 656
  - pg\_statio\_sys\_tables, 656
  - pg\_statio\_user\_indexes, 656
  - pg\_statio\_user\_sequences, 656
  - pg\_statio\_user\_tables, 656
  - pg\_statistic, 447, 2146
  - pg\_statistics\_obj\_is\_visible, 331
  - pg\_statistic\_ext, 448, 2147
  - pg\_statistic\_ext\_data, 448, 2147
  - pg\_stats, 447, 2174
  - pg\_stats\_ext, 2175
  - pg\_stat\_activity, 654, 657
  - pg\_stat\_all\_indexes, 655, 675
  - pg\_stat\_all\_tables, 655, 674
  - pg\_stat\_archiver, 655, 671
  - pg\_stat\_bgwriter, 655, 671
  - pg\_stat\_clear\_snapshot, 679
  - pg\_stat\_database, 655, 672
  - pg\_stat\_database\_conflicts, 655, 673
-



- pg\_stat\_file, 351
- pg\_stat\_get\_activity, 679
- pg\_stat\_get\_backend\_activity, 680
- pg\_stat\_get\_backend\_activity\_start, 680
- pg\_stat\_get\_backend\_client\_addr, 680
- pg\_stat\_get\_backend\_client\_port, 680
- pg\_stat\_get\_backend\_dbid, 680
- pg\_stat\_get\_backend\_idset, 680
- pg\_stat\_get\_backend\_pid, 680
- pg\_stat\_get\_backend\_start, 680
- pg\_stat\_get\_backend\_userid, 680
- pg\_stat\_get\_backend\_wait\_event, 680
- pg\_stat\_get\_backend\_wait\_event\_type, 680
- pg\_stat\_get\_backend\_xact\_start, 680
- pg\_stat\_get\_snapshot\_timestamp, 679
- pg\_stat\_gssapi, 654, 671
- pg\_stat\_progress\_analyze, 654
- pg\_stat\_progress\_basebackup, 655
- pg\_stat\_progress\_cluster, 655
- pg\_stat\_progress\_create\_index, 655
- pg\_stat\_progress\_vacuum, 655
- pg\_stat\_replication, 654, 667
- pg\_stat\_reset, 679
- pg\_stat\_reset\_shared, 679
- pg\_stat\_reset\_single\_function\_counters, 679
- pg\_stat\_reset\_single\_table\_counters, 679
- pg\_stat\_slru, 656, 678
- pg\_stat\_ssl, 654, 670
- pg\_stat\_statements
  - 函数, 2743
- pg\_stat\_statements\_reset, 2743
- pg\_stat\_subscription, 654, 669
- pg\_stat\_sys\_indexes, 655
- pg\_stat\_sys\_tables, 655
- pg\_stat\_user\_functions, 656, 677
- pg\_stat\_user\_indexes, 656
- pg\_stat\_user\_tables, 655
- pg\_stat\_wal\_receiver, 654, 668
- pg\_stat\_xact\_all\_tables, 655
- pg\_stat\_xact\_sys\_tables, 655
- pg\_stat\_xact\_user\_functions, 656
- pg\_stat\_xact\_user\_tables, 655
- pg\_stop\_backup, 342
- pg\_subscription, 2148
- pg\_subscription\_rel, 2149
- pg\_switch\_wal, 343
- pg\_tables, 2176
- pg\_tablespace, 2149
- pg\_tablespace\_databases, 334
- pg\_tablespace\_location, 334
- pg\_tablespace\_size, 348
- pg\_table\_is\_visible, 331
- pg\_table\_size, 348
- pg\_temp, 540
  - securing functions, 1474
- pg\_terminate\_backend, 341
- pg\_timezone\_abbrevs, 2176
- pg\_timezone\_names, 2177
- pg\_total\_relation\_size, 348
- pg\_transform, 2149
- pg\_trigger, 2150
- pg\_trigger\_depth, 328
- pg\_try\_advisory\_lock, 352
- pg\_try\_advisory\_lock\_shared, 352
- pg\_try\_advisory\_xact\_lock, 353
- pg\_try\_advisory\_xact\_lock\_shared, 353
- pg\_ts\_config, 2151
- pg\_ts\_config\_is\_visible, 331
- pg\_ts\_config\_map, 2152
- pg\_ts\_dict, 2152
- pg\_ts\_dict\_is\_visible, 331
- pg\_ts\_parser, 2153
- pg\_ts\_parser\_is\_visible, 331
- pg\_ts\_template, 2153
- pg\_ts\_template\_is\_visible, 331
- pg\_type, 2154
- pg\_typeof, 334
- pg\_type\_is\_visible, 332
- pg\_user, 2177
- pg\_user\_mapping, 2157
- pg\_user\_mappings, 2178
- pg\_views, 2178
- pg\_visible\_in\_snapshot, 338
- pg\_walfile\_name, 343
- pg\_walfile\_name\_offset, 343
- pg\_wal\_lsn\_diff, 343
- pg\_wal\_replay\_pause, 344
- pg\_wal\_replay\_resume, 345
- pg\_xact\_commit\_timestamp, 339
- pg\_xact\_status, 337
- phraseto\_tsquery, 265, 393
- pi, 201
- PIC, 991
- PID
  - 确定服务器进程的 PID
    - in libpq, 743
- PITR, 619
- PITR standby, 633
- pkg-config
  - with ecpg, 868
  - with libpq, 790
- PL/oraSQL, 1146
- PL/pgSQL, 1084
- PL/SQL (Oracle)
  - 移植到 PL/pgSQL, 1137
- plainto\_tsquery, 265, 392
- plan\_cache\_mode 配置参数, 526
- pldebugger, 3039
- plpgsql.check\_asserts配置参数, 1122
- plpgsql.variable\_conflict配置参数, 1132
- point, 153, 260
- point-in-time recovery, 619
- policy, 74
- polygon, 155, 261
- popen, 259

- populate\_record, 2559
- port, 735
- port配置参数, 492
- position, 205, 212, 215
- postgres, 589
- postgres\_fdw, 3034
- postmaster, 2082
- post\_auth\_delay配置参数, 558
- power, 201
- PQbackendPID, 743
- PQbinaryTuples, 755
  - with COPY, 767
- PQcancel, 764
- PQclear, 752
- PQclientEncoding, 771
- PQcmdStatus, 756
- PQcmdTuples, 756
- PQconndefaults, 729
- PQconnectdb, 727
- PQconnectdbParams, 726
- PQconnectionNeedsPassword, 743
- PQconnectionUsedPassword, 743
- PQconnectPoll, 728
- PQconnectStart, 728
- PQconnectStartParams, 728
- PQconninfo, 730
- PQconninfoFree, 772
- PQconninfoParse, 730
- PQconsumeInput, 762
- PQcopyResult, 774
- PQdb, 740
- PQdescribePortal, 748
- PQdescribePrepared, 748
- PQencryptPassword, 773
- PQencryptPasswordConn, 773
- PQendcopy, 770
- PQerrorMessage, 743
- PQescapeBytea, 759
- PQescapeByteaConn, 759
- PQescapeIdentifier, 758
- PQescapeLiteral, 757
- PQescapeString, 758
- PQescapeStringConn, 758
- PQexec, 745
- PQexecParams, 746
- PQexecPrepared, 748
- PQfformat, 754
  - with COPY, 767
- PQfinish, 730
- PQfireResultCreateEvents, 773
- PQflush, 763
- PQfmod, 754
- PQfn, 765
- PQfname, 753
- PQfnumber, 753
- PQfreeCancel, 764
- PQfreemem, 772
- PQfsize, 754
- PQftable, 753
- PQftablecol, 754
- PQftype, 754
- PQgetCancel, 764
- PQgetCopyData, 768
- PQgetisnull, 755
- PQgetlength, 755
- PQgetline, 769
- PQgetlineAsync, 769
- PQgetResult, 761
- PQgetssl, 745
- PQgetSSLKeyPassHook\_OpenSSL, 732
- PQgetvalue, 755
- PQhost, 740
- PQhostaddr, 741
- PQinitOpenSSL, 788
- PQinitSSL, 788
- PQinstanceData, 779
- PQisBusy, 762
- PQisnonblocking, 763
- PQisthreadsafe, 789
- PQlibVersion, 775
  - (see also PQserverVersion)
- PQlightdbserverVersion, 743
- PQmakeEmptyPGresult, 773
- PQnfields, 753
  - with COPY, 767
- PQnotifies, 766
- PQnparams, 756
- PQntuples, 753
- PQoidStatus, 757
- PQoidValue, 757
- PQoptions, 741
- PQparameterStatus, 742
- PQparamtype, 756
- PQpass, 740
- PQping, 732
- PQpingParams, 731
- PQport, 741
- PQprepare, 747
- PQprint, 756
- PQprotocolVersion, 742
- PQputCopyData, 768
- PQputCopyEnd, 768
- PQputline, 770
- PQputnbytes, 770
- PQregisterEventProc, 779
- PQrequestCancel, 765
- PQreset, 731
- PQresetPoll, 731
- PQresetStart, 731
- PQresStatus, 750
- PQresultAlloc, 774
- PQresultErrorField, 750
- PQresultErrorMessage, 750
- PQresultInstanceData, 779
- PQresultMemorySize, 775
- PQresultSetInstanceData, 779

- 
- PQresultStatus, 749
  - PQresultVerboseErrorMessage, 750
  - PQsendDescribePortal, 761
  - PQsendDescribePrepared, 761
  - PQsendPrepare, 760
  - PQsendQuery, 760
  - PQsendQueryParams, 760
  - PQsendQueryPrepared, 761
  - PQserverVersion, 742
  - PQsetClientEncoding, 771
  - PQsetdb, 727
  - PQsetdbLogin, 727
  - PQsetErrorContextVisibility, 771
  - PQsetErrorVerbosity, 771
  - PQsetInstanceData, 779
  - PQsetnonblocking, 763
  - PQsetNoticeProcessor, 775
  - PQsetNoticeReceiver, 775
  - PQsetResultAttrs, 774
  - PQsetSingleRowMode, 764
  - PQsetSSLKeyPassHook\_OpenSSL, 732
  - PQsetvalue, 774
  - PQsocket, 743
  - PQsslAttribute, 744
  - PQsslAttributeNames, 744
  - PQsslInUse, 744
  - PQsslStruct, 744
  - PQstatus, 741
  - PQtrace, 772
  - PQtransactionStatus, 741
  - PQtty, 741
  - PQunescapeBytea, 759
  - PQuntrace, 772
  - PQuser, 740
  - PREPARE, 1694
  - PREPARE TRANSACTION, 1697
  - prepared statements
    - creating, 1694
    - executing, 1654
    - removing, 1592
  - pretty-print 配置参数, 3076
  - pre\_auth\_delay配置参数, 558
  - primary key, 64
  - primary\_conninfo 配置参数, 517
  - primary\_slot\_name 配置参数, 517
  - privilege, 70
    - querying, 328
    - for schemas, 83
  - procedural language
    - handler for, 2240
  - procedure
    - user-defined, 966
  - promote\_trigger\_file 配置参数, 517
  - ps
    - to monitor activity, 652
- Q**
- q quoting, 36
  - qualified name, 80
  - querytree, 266, 399
  - quotation marks
    - and identifiers, 32
    - escaping, 33
  - quote\_all\_identifiers配置参数, 554
  - quote\_ident, 208
    - 在 PL/pgSQL 中使用, 1096
  - quote\_literal, 208
    - 在 PL/pgSQL 中使用, 1096
  - quote\_nullable, 208
    - 在 PL/pgSQL 中使用, 1096
- R**
- radians, 201
  - radius, 259
  - RADIUS, 578
  - RAISE
    - 在 PL/pgSQL 中, 1120
  - random, 202
  - random\_page\_cost配置参数, 521
  - range type, 185
    - exclude, 189
    - indexes on, 189
  - range\_merge, 310
  - rank, 317
    - hypothetical, 315
  - read-only transaction, 542
    - setting default, 542
  - real, 135
  - REASSIGN OWNED, 1699
  - record, 192
  - recovery.signal, 511
  - recovery\_end\_command 配置参数, 512
  - recovery\_min\_apply\_delay 配置参数, 518
  - recovery\_target 配置参数, 513
  - recovery\_target\_action 配置参数, 513
  - recovery\_target\_inclusive 配置参数, 513
  - recovery\_target\_lsn 配置参数, 513
  - recovery\_target\_name 配置参数, 513
  - recovery\_target\_time 配置参数, 513
  - recovery\_target\_timeline 配置参数, 513
  - recovery\_target\_xid 配置参数, 513
  - rectangle, 154
  - RECURSIVE
    - in common table expressions, 125
    - in views, 1587
  - referential integrity, 16, 64
  - REFRESH MATERIALIZED VIEW, 1700
  - regclass, 190
  - regcollation, 190
  - regconfig, 190
  - regdictionary, 190
  - regexp\_match, 208, 219
  - regexp\_matches, 208, 219
  - regexp\_replace, 208, 219
  - regexp\_split\_to\_array, 208, 219
  - regexp\_split\_to\_table, 208, 219
-

- regnamespace, 190
  - regoper, 190
  - regoperator, 190
  - regproc, 190
  - regprocedure, 190
  - regression intercept, 314
  - regression slope, 314
  - regression tests, 712
  - regrole, 190
  - regr\_avgx, 314
  - regr\_avgy, 314
  - regr\_count, 314
  - regr\_intercept, 314
  - regr\_r2, 314
  - regr\_slope, 314
  - regr\_sxx, 314
  - regr\_sxy, 314
  - regr\_syy, 314
  - regtype, 190
  - regular expression, 218, 219
    - (see also pattern matching)
  - regular expressions
    - 与区域, 594
  - REINDEX, 1702
  - reindexdb, 2017
  - RELEASE SAVEPOINT, 1706
  - repeat, 208
  - replace, 209
  - REPLACE INTO, 1707
  - RESET, 1710
  - restart\_after\_crash配置参数, 555
  - restore\_command 配置参数, 512
  - RESTRICT
    - with DROP, 99, 100, 104
    - foreign key action, 66
  - RETURN NEXT
    - in PL/pgSQL, 1101
  - RETURN QUERY
    - in PL/pgSQL, 1101
  - RETURNING, 103
  - RETURNING INTO
    - in PL/pgSQL, 1094
    - 在 PL/oraSQL 中, 1155
  - reverse, 209
  - REVOKE, 70, 1711
  - right, 209
  - role
    - applicable, 914
    - enabled, 930
    - membership in, 584
    - privilege to create, 583
    - privilege to initiate replication, 583
  - ROLLBACK, 1715
  - rollback
    - ltsql, 2006
  - ROLLBACK PREPARED, 1716
  - ROLLBACK TO SAVEPOINT, 1717
  - ROLLUP, 117
  - round, 201
  - routine, 966
  - ROW, 50
  - row, 57
  - row estimation
    - multivariate, 2335
  - row-level security, 74
  - rowid, 68
  - row\_number, 317
  - row\_security 配置参数, 541
  - row\_security\_active, 330
  - row\_to\_json, 285
  - rpad, 209
  - rtrim, 209
  - rule
    - for DELETE, 1066
    - for INSERT, 1066
    - for SELECT, 1058
    - for UPDATE, 1066
- ## S
- SAVEPOINT, 1719
  - savepoints
    - defining, 1719
    - releasing, 1706
    - rolling back, 1717
  - scale, 201
  - schema, 80
    - creating, 80
    - current, 82, 326
    - public, 81
    - removing, 81
  - SCRAM, 572
  - search path, 81
    - current, 326
  - search\_path 配置参数
    - use in securing functions, 1474
  - search\_path配置参数, 82, 540
  - SECURITY LABEL, 1721
  - seg, 3044
  - segment\_size配置参数, 556
  - SELECT, 9, 105, 1724
    - determination of result type, 368
    - 选择列表, 119
  - SELECT INTO, 1752
    - in PL/pgSQL, 1094
    - 在 PL/oraSQL 中, 1155
  - sepgsql, 3046
  - sepgsql.debug\_audit 配置参数, 3049
  - sepgsql.permissive 配置参数, 3049
  - sequence, 297
    - and serial type, 136
  - sequential scan, 521
  - seq\_page\_cost配置参数, 521
  - serial, 136
  - serial2, 136
  - serial4, 136
  - serial8, 136

- server\_encoding配置参数, 556
- server\_version\_num配置参数, 557
- server\_version配置参数, 556
- session\_preload\_libraries配置参数, 551
- session\_replication\_role配置参数, 542
- session\_user, 328
- SET, 341, 1754
- SET CONSTRAINTS, 1757
- SET ROLE, 1758
- SET SESSION AUTHORIZATION, 1760
- SET TRANSACTION, 1762
- SET XML OPTION, 544
- setseed, 202
- setval, 298
- setweight, 266, 397
  - setweight for specific lexeme(s), 266
- set\_bit, 213, 216
- set\_byte, 213
- set\_config, 341
- set\_limit, 2748
- set\_masklen, 263
- sha224, 213
- sha256, 213
- sha384, 213
- sha512, 213
- shared\_buffers配置参数, 498
- shared\_memory\_type 配置参数, 501
- shared\_preload\_libraries, 1002
- shared\_preload\_libraries配置参数, 552
- shobj\_description, 337
- SHOW, 341, 1765, 2195
- show\_limit, 2748
- show\_trgm, 2747
- SIGHUP, 489, 567, 570
- SIGINT, 478
- sign, 201
- significant digits, 546
- SIGQUIT, 478
- SIGTERM, 478
- SIMILAR TO, 218
- similarity, 2747
- sin, 203
- sind, 203
- sinh, 204
- skeys, 2558
- sleep, 254
- slice, 2559
- slope, 259
- SLRU, 678
- smallint, 133
- smallserial, 136
- Solaris
  - 共享库, 992
  - 启动脚本, 470
- SOME, 312, 318, 320
- sort, 2564
- sort\_asc, 2564
- sort\_desc, 2564
- soundex, 2553
- SPI, 1220
  - examples, 3054
  - SPI\_commit, 1273
  - SPI\_commit\_and\_chain, 1273
  - SPI\_connect, 1221
  - SPI\_connect\_ext, 1221
  - SPI\_copytuple, 1266
  - SPI\_cursor\_close, 1248
  - SPI\_cursor\_fetch, 1244
  - SPI\_cursor\_find, 1243
  - SPI\_cursor\_move, 1245
  - SPI\_cursor\_open, 1239
  - SPI\_cursor\_open\_with\_args, 1240
  - SPI\_cursor\_open\_with\_paramlist, 1242
  - SPI\_exec, 1226
  - SPI\_execp, 1238
  - SPI\_execute, 1223
  - SPI\_execute\_plan, 1236
  - SPI\_execute\_plan\_with\_paramlist, 1237
  - SPI\_execute\_with\_args, 1227
  - SPI\_finish, 1222
  - SPI\_fname, 1254
  - SPI\_fnumber, 1255
  - SPI\_freeplan, 1272
  - SPI\_freetuple, 1270
  - SPI\_freetuptable, 1271
  - SPI\_getargcount, 1233
  - SPI\_getargtypeid, 1234
  - SPI\_getbinval, 1257
  - SPI\_getnspname, 1261
  - SPI\_getrelname, 1260
  - SPI\_gettype, 1258
  - SPI\_gettypeid, 1259
  - SPI\_getvalue, 1256
  - SPI\_is\_cursor\_plan, 1235
  - SPI\_keepplan, 1249
  - SPI\_modifytuple, 1268
  - SPI\_palloc, 1263
  - SPI\_pfree, 1265
  - SPI\_prepare, 1229
  - SPI\_prepare\_cursor, 1231
  - SPI\_prepare\_params, 1232
  - SPI\_register\_relation, 1251
  - SPI\_register\_trigger\_data, 1253
  - SPI\_realloc, 1264
  - SPI\_result\_code\_string, 1262
  - SPI\_returntuple, 1267
  - SPI\_rollback, 1274
  - SPI\_rollback\_and\_chain, 1274
  - SPI\_saveplan, 1250
  - SPI\_scroll\_cursor\_fetch, 1246
  - SPI\_scroll\_cursor\_move, 1247
  - SPI\_start\_transaction, 1275
  - SPI\_unregister\_relation, 1252
  - split\_part, 209
  - SQL/CLI, 2436
  - SQL/Foundation, 2436

- SQL/Framework, 2436
  - SQL/JRT, 2436
  - SQL/JSON 路径语言, 291
  - SQL/MDA, 2436
  - SQL/MED, 2436
  - SQL/OLB, 2436
  - SQL/PSM, 2436
  - SQL/Schemata, 2436
  - SQL/XML, 2436
    - 限制和符合性, 2457
  - sqrt, 201
  - ssh, 486
  - SSI, 423
  - SSL, 482, 785
    - in libpq, 745
    - with libpq, 738
  - sslinfo, 3055
  - ssl\_ca\_file配置参数, 496
  - ssl\_cert\_file配置参数, 496
  - ssl\_cipher, 3055
  - ssl\_ciphers配置参数, 496
  - ssl\_client\_cert\_present, 3055
  - ssl\_client\_dn, 3055
  - ssl\_client\_dn\_field, 3056
  - ssl\_client\_serial, 3055
  - ssl\_crl\_dir 配置参数, 496
  - ssl\_crl\_file配置参数, 496
  - ssl\_dh\_params\_file配置参数, 498
  - ssl\_ecdh\_curve配置参数, 497
  - ssl\_extension\_info, 3056
  - ssl\_issuer\_dn, 3055
  - ssl\_issuer\_field, 3056
  - ssl\_is\_used, 3055
  - ssl\_key\_file配置参数, 496
  - ssl\_library 配置参数, 557
  - ssl\_max\_protocol\_version 配置参数, 497
  - ssl\_min\_protocol\_version 配置参数, 497
  - ssl\_passphrase\_command\_supports\_reload配置参数, 498
  - ssl\_passphrase\_command配置参数, 498
  - ssl\_prefer\_server\_ciphers配置参数, 497
  - ssl\_version, 3055
  - ssl配置参数, 496
  - SSPI, 574
  - STABLE, 982
  - standard deviation, 314
    - population, 314
    - sample, 314
  - standard\_conforming\_strings配置参数, 554
  - standby.signal, 511, 636, 637
    - lt\_basebackup --write-recovery-conf, 1857
    - 热备, 649
  - START TRANSACTION, 1769
  - starts\_with, 209
  - START\_REPLICATION, 2196
  - statement\_timeout配置参数, 543
  - statement\_timestamp, 243
  - statistics, 653
    - of the planner, 448, 611
  - stats\_temp\_directory配置参数, 537
  - stddev, 314
  - stddev\_pop, 314
  - stddev\_samp, 314
  - STONITH, 633
  - strict\_word\_similarity, 2748
  - string (see 字符串)
  - strings
    - backslash quotes, 554
    - escape warning, 554
    - standard conforming, 554
  - string\_agg, 312
  - string\_to\_array, 308
  - strip, 266, 397
  - strpos, 209
  - subarray, 2564
  - subtree, 2780
  - subpath, 2781
  - subquery, 318
  - substr, 209, 213
  - substring, 206, 212, 216, 218, 219
  - sum, 312
  - superuser, 5, 583
  - superuser\_reserved\_connections配置参数, 493
  - support functions
    - in\_range, 2288
  - suppress\_redundant\_updates\_trigger, 353
  - svals, 2558
  - synchronize\_seqscans配置参数, 554
  - synchronous\_commit配置参数, 506
  - synchronous\_standby\_names配置参数, 515
  - syntax
    - SQL, 31
  - sysdate, 242
  - syslog\_facility配置参数, 528
  - syslog\_ident配置参数, 528
  - syslog\_sequence\_numbers 配置参数, 528
  - syslog\_split\_messages 配置参数, 529
  - system catalog
    - schema, 83
  - systemd, 469
    - RemoveIPC, 475
  - systemtimestamp, 241, 241
- ## T
- table, 57
    - creating, 57
    - inheritance, 84
    - modifying, 68
    - partitioning, 87
    - removing, 58
    - renaming, 70
  - TABLE command, 1724
  - table function
    - XMLTABLE, 276
  - tableam
    - 表访问方法, 2271

- tablefunc, 3059
- tableoid, 67
- TABLESAMPLE 方法, 2260
- tablespace
  - default, 541
  - temporary, 541
- table\_am\_handler, 192
- tan, 203
- tand, 203
- tanh, 204
- tcn, 3068
- tcp\_keepalives\_count配置参数, 494
- tcp\_keepalives\_idle配置参数, 494
- tcp\_keepalives\_interval配置参数, 494
- tcp\_user\_timeout 配置参数, 494
- template0, 589
- template1, 589, 589
- temp\_buffers配置参数, 499
- temp\_file\_limit配置参数, 501
- temp\_tablespaces配置参数, 541
- test, 712
- test\_decoding, 3069
- text, 138, 263
- text2ltree, 2781
- threads
  - with libpq, 789
- tid, 190
- time, 141, 143
  - constants, 145
  - current, 250
  - 输出格式, 146
    - (see also formatting)
- time with time zone, 143
- time without time zone, 143
- time zone, 147, 546
  - conversion, 249
  - input abbreviations, 2395
- time zone names, 546
- timelines, 619
- TIMELINE\_HISTORY, 2195
- timeofday, 243
- timeout
  - client authentication, 495
  - deadlock, 553
- timestamp, 141, 144
- timestamp with time zone, 144
- timestamp without time zone, 144
- timestampz, 141
- timezone\_abbreviations配置参数, 546
- TimeZone配置参数, 546
- TOAST, 2316
  - per-column storage settings, 1377
  - per-type storage settings, 1402
  - versus large objects, 801
  - 以及用户定义的类型, 1012
- toast\_tuple\_target 存储参数, 1540
- token, 31
- to\_ascii, 209
- to\_char, 232
  - 与区域, 594
- to\_date, 232
- to\_hex, 209
- to\_json, 285
- to\_jsonb, 285
- to\_number, 232
- to\_regclass, 334
- to\_regcollation, 334
- to\_regnamespace, 334
- to\_regoper, 334
- to\_regoperator, 335
- to\_regproc, 335
- to\_regprocedure, 335
- to\_regrole, 335
- to\_regtype, 335
- to\_timestamp, 232, 243
- to\_tsquery, 266, 391
- to\_tsvector, 266, 390
- trace\_locks配置参数, 558
- trace\_lock\_oidmin配置参数, 559
- trace\_lock\_table配置参数, 559
- trace\_lwlocks配置参数, 559
- trace\_notify配置参数, 558
- trace\_recovery\_messages配置参数, 558
- trace\_sort配置参数, 558
- trace\_userlocks配置参数, 559
- track\_activities配置参数, 537
- track\_activity\_query\_size配置参数, 537
- track\_commit\_timestamp 配置参数, 515
- track\_counts配置参数, 537
- track\_functions配置参数, 537
- track\_io\_timing配置参数, 537
- transaction, 17
- transaction isolation level, 542
  - setting default, 542
- transaction log (see WAL)
- transaction\_deferrable 配置参数, 542
- transaction\_isolation配置参数, 542
- transaction\_read\_only 配置参数, 542
- transaction\_timestamp, 243
- transform\_null\_equals配置参数, 555
- transition tables, 1568
  - (see also ephemeral named relation)
  - implementation in PLs, 1253
  - referencing from C trigger, 1042
- translate, 209
- transparent huge pages, 499
- trigger, 192, 1039
  - in C, 1042
  - 约束触发器, 1569
- triggered\_change\_notification, 3068
- trim, 206, 212
- trim\_scale, 201
- true, 150
- trunc, 201, 264
- TRUNCATE, 1770
- tsm\_handler, 192

- tsm\_system\_rows, 3070
  - tsm\_system\_time, 3070
  - tsquery\_phrase, 267, 398
  - tsquery (数据类型), 159
  - tsvector 连接, 397
  - tsvector\_to\_array, 268
  - tsvector\_update\_trigger, 353
  - tsvector\_update\_trigger\_column, 353
  - tsvector (数据类型), 158
  - ts\_debug, 268, 414
  - ts\_delete, 267
  - ts\_filter, 267
  - ts\_headline, 267, 396
  - ts\_lexize, 268, 417
  - ts\_parse, 268, 416
  - ts\_rank, 267, 394
  - ts\_rank\_cd, 267, 394
  - ts\_rewrite, 267, 399
  - ts\_stat, 269, 401
  - ts\_token\_type, 269, 416
  - tuple\_data\_split, 3028
  - txid\_current, 338
  - txid\_current\_if\_assigned, 338
  - txid\_current\_snapshot, 338
  - txid\_snapshot\_xip, 339
  - txid\_snapshot\_xmax, 339
  - txid\_snapshot\_xmin, 339
  - txid\_status, 339
  - txid\_visible\_in\_snapshot, 339
  - type cast, 37, 46
- U**
- UESCAPE, 32, 34
  - unaccent, 3072, 3073
  - Unicode escape
    - in identifiers, 32
    - in string constants, 34
  - Unicode normalization, 205, 205
  - UNION, 121
    - determination of result type, 367
  - uniq, 2564
  - unique constraint, 63
  - Unix 域套接字, 734
  - unix\_socket\_directories配置参数, 493
  - unix\_socket\_group配置参数, 493
  - unix\_socket\_permissions配置参数, 493
  - unknown, 192
  - UNLISTEN, 1772
  - unnest, 308
    - for tsvector, 268
  - unqualified name, 81
  - UPDATE, 14, 102, 1774
    - RETURNING, 103
  - update\_process\_title 配置参数, 536
  - upgrading, 479
  - upper, 206, 310
    - 与区域, 594
  - upper\_inc, 310
  - upper\_inf, 310
  - UPSERT, 1671
  - URI, 732
  - user, 328
    - current, 326
  - user mapping, 98
  - user\_catalog\_table 存储参数, 1542
  - UUID, 161
    - 生成, 161
  - uuid-oss, 3073
  - uuid\_generate\_v1, 3074
  - uuid\_generate\_v1mc, 3074
  - uuid\_generate\_v3, 3074
- V**
- VACUUM, 1779
  - vacuumdb, 2021
  - vacuumlo, 3099
  - vacuum\_cost\_delay配置参数, 502
  - vacuum\_cost\_limit配置参数, 502
  - vacuum\_cost\_page\_dirty配置参数, 502
  - vacuum\_cost\_page\_hit配置参数, 502
  - vacuum\_cost\_page\_miss配置参数, 502
  - vacuum\_cost\_page\_miss配置参数, 502
  - vacuum\_defer\_cleanup\_age配置参数, 516
  - vacuum\_freeze\_min\_age配置参数, 543
  - vacuum\_freeze\_table\_age配置参数, 543
  - vacuum\_index\_cleanup 存储参数, 1541
  - vacuum\_multixact\_freeze\_min\_age配置参数, 544
  - vacuum\_multixact\_freeze\_table\_age配置参数, 543
  - vacuum\_truncate 存储参数, 1541
  - VALUES, 123, 1783
    - determination of result type, 367
  - varchar, 138
  - variadic function, 973
  - variance, 314
    - population, 314
    - sample, 314
  - var\_pop, 314
  - var\_samp, 314
  - version, 6, 328
    - compatibility, 479
  - view, 16
    - updating, 1071
  - VM (see 可见性映射)
  - void, 192
  - VOLATILE, 982
  - volatility
    - functions, 982
  - VPATH, 1037
- W**
- WAL, 697
  - wal2sql, 3075
  - wal\_block\_size配置参数, 557
  - wal\_buffers配置参数, 508
  - wal\_compression 配置参数, 508



wal\_consistency\_checking配置参数, 559  
wal\_debug配置参数, 559  
wal\_init\_zero 配置参数, 508  
wal\_keep\_size 配置参数, 514  
wal\_level配置参数, 505  
wal\_log\_hints配置参数, 508  
wal\_receiver\_create\_temp\_slot 配置参数, 518  
wal\_receiver\_status\_interval配置参数, 518  
wal\_receiver\_timeout配置参数, 518  
wal\_recycle 配置参数, 508  
wal\_retrieve\_retry\_interval 配置参数, 518  
wal\_segment\_size配置参数, 557  
wal\_sender\_timeout配置参数, 515  
wal\_skip\_threshold 配置参数, 509  
wal\_sync\_method配置参数, 507  
wal\_writer\_delay配置参数, 508  
wal\_writer\_flush\_after 配置参数, 509  
websearch\_to\_tsquery, 265  
WHERE, 113  
where to log, 526  
WHILE  
    在 PL/oraSQL 中, 1164  
    在 PL/pgSQL 中, 1107  
width, 259  
width\_bucket, 202  
window function, 18  
WITH  
    in SELECT, 124, 1724  
WITH CHECK OPTION, 1587  
WITHIN GROUP, 42  
witness server, 633  
word\_similarity, 2747  
work\_mem配置参数, 500  
wraparound  
    of multixact IDs, 615  
write-in-chunks 配置参数, 3076

## X

xid, 190  
xid8, 190  
xmax, 67  
xmin, 67  
XML, 161  
XML export, 279  
XML option, 162, 544  
XML 函数, 269  
xml2, 3078  
xmlagg, 273, 312  
xmlbinary配置参数, 544  
xmlcomment, 269  
xmlconcat, 270  
xmlelement, 270  
XMLEXISTS, 274  
xmlforest, 272  
xmloption配置参数, 544  
xmlparse, 161  
xmlpi, 272  
xmlroot, 272

xmlserialize, 162  
xmltable, 276  
xml\_is\_well\_formed, 274  
xml\_is\_well\_formed\_content, 274  
xml\_is\_well\_formed\_document, 274  
XPath, 275  
xpath\_exists, 276  
xpath\_table, 3079  
XQuery正则表达式, 231  
xslt\_process, 3082

## Z

zero\_damaged\_pages配置参数, 560  
zhparser, 3082