LightDB-A Migration Manual

发布 *22.4*

LightDB

2023 年 05 月 09 日

目录:

1 从 Oracle 迁移到 LightDB-A

我们可以使用 ora2pg 工具来迁移。ora2pg 同时支持 DDL 和数据的迁移。

如果您的表数据量很大，则可以考虑使用 ora2pg 迁移 DDL, 数据迁移则使用 `sqluldr` 工具导出然后使用 `lt_bulkload` 工具导入。

以下讲述使用 ora2pg 工具进行迁移的步骤和注意要点

2 ora2pg 安装和基本使用

ora2pg 的安装和基本使用方法可参考 Oracle 迁移 LightDB 实施手册 ,

此处不再赘述。

3 分布式数据库特性介绍

从 Oracle 迁移到 LightDB-A 不仅仅是两个异构数据库之间的迁移，也是集中式数据库到分布式数据库的迁移，所以在迁移过程需要根据应用业务特点并结合分布式数据特性调整 DDL 语句，以确保有好的性能。

3.1 分布式数据库性能因素

为了实现性能最大化，在分布式数据库需要考虑如下几个因素:

1. **均衡分布数据**

   应该尽量让所有 Segment 包含等量的数据，这样执行查询操作时，每个 Segment 的负载大致相等，性能可以达到最大化。如果数据分布不平衡，可能会导致数据量大的 segment 成为瓶颈。

   例如：如果一个用户表以用户所属的省份 ID 作为分布式键，但是在该系统中某一个省份的用户数占比很高，则数据分布就会不均衡。

   一般情况下建议使用主键或者唯一键作为分布式键。

2. **本地化处理**

   在 Segment 层面上，如果与连接、排序或者聚合操作相关的工作在本地完成则性能最好。

   如果数据分布不合适，则在连接执行时，需要把数据重新分发到其他节点上。

3. **均衡分布负载**

   应该尽力让所有的 Segment 处理等量的查询负载。

   如果一个表的数据分布策略与查询谓词匹配不好，查询负载可能会倾斜。例如: 假定一个销售事务表按照客户 ID 作为分布式键。如果某个查询如果限定一个客户 ID(如: `WHERE client_id=123`)，该查询处理工作只会在一个 Segment 上执行。

3.2 分布策略介绍

LightDB 支持 3 种分布策略: DISTRIBUTED BY，DISTRIBUTED RANDOMLY，或 DISTRIBUTED REPLICATED。

1. **DISTRIBUTED BY**

   hash 分布策略。hash 分布策略需要有个分布式键，相同分布式键的数据会在相同的 segment 上。

   如果在建表时不指定分布策略，则默认使用 hash 分布，并且选择主键或者唯一键作为分布式键，如果没有主键和唯一键，则使用第一个字段作为分布式键。注意默认行为受 GUC 参数: `gp_create_table_random_default_distribution` 当该参数为 on 时，默认分布策略为随机分布。

```
postgres=# create table t1(c1 int, c2 int);
NOTICE:  Table doesn't have 'DISTRIBUTED BY' clause -- Using column named 'c1' as␣
↪the LightDB-A Database data distribution key for this table.
HINT:  The 'DISTRIBUTED BY' clause determines the distribution of data. Make sure␣
↪column(s) chosen are the optimal data distribution key to minimize skew.
CREATE TABLE
```

   不支持多个唯一键 (含主键)

```
postgres=# create table t3(c1 int unique, c2 int unique);
ERROR:  UNIQUE or PRIMARY KEY definitions are incompatible with each other
HINT:  When there are multiple PRIMARY KEY / UNIQUE constraints, they must have␣
↪at least one column in common.

postgres=# create table t3(c1 int primary key , c2 int unique);
ERROR:  UNIQUE or PRIMARY KEY definitions are incompatible with each other
HINT:  When there are multiple PRIMARY KEY / UNIQUE constraints, they must have␣
↪at least one column in common.
```

   支持联合主键或者唯一键作为分布式键

```
# 默认选择主键
postgres=# create table t3(c1 int, c2 int, primary key(c1,c2));
CREATE TABLE

postgres=# create table t5(c1 int, c2 int, unique(c1,c2)) distributed by(c1,c2);
CREATE TABLE
```

2. **DISTRIBUTED RANDOMLY**

   随机分布，如果不能确定一张表的 hash 分布式键，或者不存在合理的避免数据偏斜的分布式键。则可以使用随机分布。数据库会采用循环的方式将一次插入的数据存储到不同的节点上。

```
postgres=# create table t7(id int , name varchar) DISTRIBUTED RANDOMLY;
CREATE TABLE
```

   不支持唯一约束和主键约束

```
postgres=# create table t9(id int unique , name varchar) DISTRIBUTED RANDOMLY;
ERROR:  UNIQUE and DISTRIBUTED RANDOMLY are incompatible
postgres=# create table t9(id int primary key  , name varchar) DISTRIBUTED␣
↪RANDOMLY;
ERROR:  PRIMARY KEY and DISTRIBUTED RANDOMLY are incompatible
```

需要注意的是：随机分布仅在单个 SQL 语句中生效，如果每次仅插入一行数据，则最终的数据会全部保存在第一个节点上。

```
postgres=# insert into t7(id,name) values(1,'abc');
INSERT 0 1
postgres=# insert into t7(id,name) values(2,'abc');
INSERT 0 1
postgres=# insert into t7(id,name) values(4,'abc');
INSERT 0 1

# 数据分布在同一个节点上
postgres=# select gp_segment_id,* from t7;
gp_segment_id | id | name
---------------+----+------
             3 |  1 | abc
             3 |  2 | abc
             3 |  4 | abc
(3 rows)
```

3. **DISTRIBUTED REPLICATED**

   复制表，在每个 segment 上都有一份全量的数据，

```
create table t10(id int , name varchar) DISTRIBUTED REPLICATED;
```

   复制表可以避免分布式查询计划：如果一张表的数据在各个 segment 上都有拷贝，那么就可以生成本地连接计划，而避免数据在集群的不同节点间移动。如果用复制表存储数据量比较小的表（譬如数千行），那么性能有明显的提升。数据量大的表不适合使用复制表模式。

   另外复制表还可以解决 segment 在执行函数时不能访问其他表的限制

```
postgres=# create function tf2() returns bigint as $$ select count(*) from t8; $$␣
↪language sql;
postgres=# select tf2() from t7;;
ERROR:  function cannot execute on a QE slice because it accesses relation
↪"public.t8"  (seg1 slice1 10.20.137.130:49002 pid=3401826)
CONTEXT:  SQL function "tf2" during startup
```

3.3 分布式执行计划介绍

LightDB-A 使用 `Motion` 操作符让执行计划支持并行。`Motion` 操作符有三种:

1. **Broadcast motion**

   每个 segment 把自己的数据广播到其他 segment 中，这样每个 segment 都有该表的全量数据。

   执行计划出现 Broadcast motion 时性能可能不是最优的，一般情况下只有小表会出现，大表广播的代价会比较大。

2. **Redistribute motion**

   每个节点根据关联列重新计算 hash，然后把合适的数据发送到对应的 segment 上。

3. **Gather motion**

   汇总所有 segment 的结果，一般执行计划最后一步都是这个。

以下创建两个简单的分布式表用于验证:

```
create table t11(v1 int , v2 int) distributed by(v1);
insert into t11(v1,v2) select v, v+10 from generate_series(1,10000000) as v;

create table t12(v1 int , v2 int) distributed by(v1);
insert into t12(v1,v2) select v, v+10 from generate_series(1,10000000) as v;

-- 小表
create table t13(v1 int , v2 int) distributed by(v1);
insert into t13(v1,v2) select v, v+10 from generate_series(1,10) as v;
```

1. 分布式键关联查询

   在关联键都是分布式键的情况下，关联操作都可以在 segment 本地完成，不需要数据重分布。

```
postgres=# explain select * from t11 inner join t12 on t11.v1 = t12.v1;
                                        QUERY PLAN
-------------------------------------------------------------------------------------
↪-------
Gather Motion 24:1  (slice1; segments: 24)  (cost=0.00..1489.46 rows=10000000␣
↪width=16)
   ->  Hash Join  (cost=0.00..1106.93 rows=416667 width=16)
         Hash Cond: (t11.v1 = t12.v1)
         ->  Seq Scan on t11  (cost=0.00..439.71 rows=416667 width=8)
         ->  Hash  (cost=439.71..439.71 rows=416667 width=8)
               ->  Seq Scan on t12  (cost=0.00..439.71 rows=416667 width=8)
Optimizer: Pivotal Optimizer (GPORCA)
(7 rows)
```

2. 分布式键和非分布式键关联查询

```
postgres=# explain select * from t11 inner join t12 on t11.v1 = t12.v2;
                                                QUERY PLAN
-------------------------------------------------------------------------------------
↪----------------------------
Gather Motion 24:1  (slice1; segments: 24)  (cost=0.00..1499.90 rows=10000000␣
↪width=16)
   ->  Hash Join  (cost=0.00..1117.36 rows=416667 width=16)
         Hash Cond: (t11.v1 = t12.v2)
         ->  Seq Scan on t11  (cost=0.00..439.71 rows=416667 width=8)
         ->  Hash  (cost=456.34..456.34 rows=416667 width=8)
               ->  Redistribute Motion 24:24  (slice2; segments: 24)  (cost=0.00..
↪456.34 rows=416667 width=8)
                     Hash Key: t12.v2
                     ->  Seq Scan on t12  (cost=0.00..439.71 rows=416667 width=8)
Optimizer: Pivotal Optimizer (GPORCA)
(9 rows)
```

   t12 表的关联键 v2 不是分布式键，所以需要先把 t12 表根据 v2 列重分布 (使用 Redistribute Motion 节点)，然后在各个 segment 进行本地联表操作

3. 非分布式键关联查询

   关联键都是不是分布式键，两张表都根据关联键做数据重分布 (两个 Redistribute Motion)，然后在各个 segment 进行本地联表操作。

```
postgres=# explain select * from t11 inner join t12 on t11.v2 = t12.v2;
                                                QUERY PLAN
-------------------------------------------------------------------------------------
↪----------------------------
```

(下页继续)

(续上页)

```
Gather Motion 24:1  (slice1; segments: 24)  (cost=0.00..1510.33 rows=10000000␣
↪width=16)
  ->  Hash Join  (cost=0.00..1127.80 rows=416667 width=16)
        Hash Cond: (t11.v2 = t12.v2)
        ->  Redistribute Motion 24:24  (slice2; segments: 24)  (cost=0.00..456.
↪34 rows=416667 width=8)
              Hash Key: t11.v2
              ->  Seq Scan on t11  (cost=0.00..439.71 rows=416667 width=8)
        ->  Hash  (cost=456.34..456.34 rows=416667 width=8)
              ->  Redistribute Motion 24:24  (slice3; segments: 24)  (cost=0.00..
↪456.34 rows=416667 width=8)
                    Hash Key: t12.v2
                    ->  Seq Scan on t12  (cost=0.00..439.71 rows=416667 width=8)
Optimizer: Pivotal Optimizer (GPORCA)
(11 rows)
```

但如果其中一张表是小表，则小表使用 Broadcast motion 进行广播效率更好。

```
postgres=# explain select * from t11 inner join t13 on t11.v2 = t13.v2;
                                        QUERY PLAN
------------------------------------------------------------------------------------
↪--------------------
Gather Motion 24:1  (slice1; segments: 24)  (cost=0.00..951.36 rows=1 width=16)
  ->  Hash Join  (cost=0.00..951.36 rows=1 width=16)
        Hash Cond: (t11.v2 = t13.v2)
        ->  Seq Scan on t11  (cost=0.00..439.71 rows=416667 width=8)
        ->  Hash  (cost=431.00..431.00 rows=1 width=8)
              ->  Broadcast Motion 24:24  (slice2; segments: 24)  (cost=0.00..
↪431.00 rows=1 width=8)
                    ->  Seq Scan on t13  (cost=0.00..431.00 rows=1 width=8)
Optimizer: Pivotal Optimizer (GPORCA)
(8 rows)
```

4 DDL 语句语法调整

因为 ora2pg 导出的部分语法和 LightDB-A 的特性不兼容，所以导出后，部分语句需要改写，目前已经发现的如下:

4.1 search_path 修改

LightDB 中 oracle 兼容相关的类型和函数都在 oracle schema 下面，在 ora2pg 的导出文件中修改 search_path，加上 oracle。使得后续的 DDL 语句能够使用 oralce 兼容特性。

```
-- 加上oracle
SET search_path = riskdata,public,Oracle,lt_catalog,pg_catalog;
```

4.2 default 语法错误

```
imp_time char(19) DEFAULT 'TO_CHAR(LOCALTIMESTAMP,'YYYY-MM-DD HH24:MI:SS')',

-- 改写成

imp_time char(19) DEFAULT TO_CHAR(LOCALTIMESTAMP,'YYYY-MM-DD HH24:MI:SS'),
```

需要去掉多余的单引号

4.3 decimal 精度不兼容

```
v decimal(21,40)

-- 会报错
-- ERROR:  NUMERIC scale 40 must be between 0 and precision 21
-- 可以改写成更高进度

v decimal(40,40)
```

SQL 标准规定 scale 大于 precision, 但 oracle 支持 scale 大于 precision。迁移过来时需要调整精度

4.4 ignore nulls 不支持

```
CREATE OR REPLACE VIEW vw_fmt_rating (innercode, chinabond_rating_issernm,␣
↪chinabond_rating_issuer, moody_issuer_rating, fitch_issuer_rating, sp_
↪issuer_rating) AS SELECT    distinct   t.innercode  ,
   last_value(t.chinabond_rating_issernm) ignore nulls over (    partition ␣
↪by  t.innercode    ) chinabond_rating_issernm ,
   last_value(t.chinabond_rating_issuer) ignore nulls  over (    partition ␣
↪by  t.innercode    ) chinabond_rating_issuer,
last_value(t.moody_issuer_rating) ignore nulls  over (    partition  by  t.
↪innercode   )  moody_issuer_rating,
last_value(t.fitch_issuer_rating) ignore nulls  over (    partition  by  t.
↪innercode    ) fitch_issuer_rating,
   last_value(t.sp_issuer_rating) ignore nulls  over (    partition  by  t.
↪innercode    )  sp_issuer_rating
FROM RISKDATA.T_FMT_RATING t
where   t.rating_object_type = 'COMPANY'
```

去掉 ignore nulls，语义保持不变

5 SQL 语法调整

5.1 MERGE INTO

由于 LightDB-A 目前不支持 oracle merge into 语法，因此需要进行手动修改，确保插入或者更新的准确性。

可参考如下改写方法

1. **场景 1：包含 merge_update_clause，不包含 merge_insert_clause** 此时的语义是更新符合条件的数据，所以总是可以改写成等价的 update 语句。

原始语句:

```
MERGE INTO schema.t_table t_alias
USING (schema.s_table | query)  s_alias ON condition1
WHEN MATCHED THEN UPDATE SET column=expr … WHERE condition2;
```

改写后:

```
UPDATE schema.t_table t_alias
SET column=expr …
FROM (schema.s_table | query)  s_alias
WHERE (condition1) AND (condition2)
```

2. 场景 2：不包含 merge_update_clause，包含 merge_insert_clause

原始语句:

```
    MERGE INTO target_table t_alias
    USING (source_table | query) s_alias
    ON condition1
    WHEN NOT MATCHED THEN INSERT (column_list)
    VALUES(values_list) WHERE condition2;

改写后
```

```
INSERT INTO target_table t_alias (column_list)
SELECT values_list
FROM (source_table | query) s_alias
WHERE (NOT EXISTS SELECT 1 FROM target_table t_alias WHERE condition1) AND␣
↪(condition2)
```

3. 场景 3： 包含 merge_update_clause，包含 merge_insert_clause

   在满足如下条件的情况下，可使用 insert onconflict 改写:

   1. **on 条件是一个唯一索引冲突，** 例如 on (t1.a = t2.a AND t1.b = t2.b) 且在 target 表中，a 和 b 是有唯一约束 (多列).
   2. insert 子句没有 where 条件。
   3. update 语句不修改分布式键中的列

   另外要注意：在 LightDB-A 中唯一约束只允许在分布式键中存在

原始语句:

```
MERGE INTO t1  pt
USING (select * from t2 ti) ps
ON (pt.person_id = ps.person_id)
WHEN MATCHED THEN UPDATE
   SET pt.first_name = ps.first_name,
      pt.last_name = ps.last_name,
      pt.title = ps.title
   WHERE ps.person_id <> 2
WHEN NOT MATCHED THEN
   INSERT (person_id, first_name,last_name,title)
   VALUES(ps.person_id,ps.first_name,ps.last_name,ps.title);
```

改写后:

```
INSERT INTO t1 AS pt (person_id, first_name,last_name,title)
    SELECT ps.person_id,ps.first_name,ps.last_name,ps.title FROM
    (select * from t2 ti) ps ON conflict(person_id)
    do UPDATE  SET first_name = excluded.first_name,
      last_name = excluded.last_name,
      title = excluded.title
  WHERE pt.person_id <> 2;
```

4. 场景 4：包含 merge_update_clause，包含 merge_insert_clause

   在 LightDB-A 下需要改成两个语句，一个 insert 语句，一个 update 语句，条件相反。

5.2 CONNECT BY

通过 LightDB-A CTE 语法结构将 oracle connect by 的作用进行替代。主要有以下三种情况：

1. prior 在表达式左边，例如 *prior id = parent*；

   Oracle connect by 语法：

```
select * from sr_menu
start with id = 1
connect by prior id = parent;
```

   LightDB-A CTE 语法：

```
with recursive cte_connect_by as (
select  s.* from sr_menu s where id = 1
union all
select s.* from cte_connect_by r inner join sr_menu s on  r.id = s.parent
)
select * from cte_connect_by;
```

2. prior 在表达式右边，例如 prior id = parent；

   Oracle connect by 语法：

```
select * from sr_menu
start with id = 1
connect by id = prior parent;
```

   LightDB-A CTE 语法：

```
with recursive cte_connect_by as (
select s.* from sr_menu s where id = 1
union all
select  s.* from cte_connect_by r inner join sr_menu s on s.id =  r.parent
)
select * from cte_connect_by ;
```

3. 含 connect_by_root

   Oracle connect by 语法：

```
select connect_by_root id, parent, title from sr_menu
start with id = 1
connect by prior id = parent
```

LightDB-A CTE 语法：

```
with recursive cte_connect_by(id, connect_by_root_id, parent, title) as (
select id, id as connect_by_root_id, parent, title from sr_menu s where id = 1
union all
select s.id, r.connect_by_root_id, s.parent, s.title from cte_connect_by r inner␣
↪join sr_menu s on  r.id = s.parent
)
select id, connect_by_root_id, parent, title from cte_connect_by ;
```

5.3 ROWNUM

Rownum 是 oracle 的伪列，多用于进行分页查询。LightDB-A 未兼容该伪列。

可以使用 row_number() over () 分析函数替代 oracle rownum 伪列，并且使用 rownum 作为别名。

5.4 table() 函数

Oracle table 函数将嵌套表的结果，转换成表格形式，显示出结果。

LightDB-A 不支持创建 table 函数，因为 table 在 LightDB-A 作为保留关键字，不能被作为函数名称使用。LightDB-A 存在函数 unnest()，功能与 Oracle 相同，存在调用 table 的地方使用 unnest 替换。

5.5 exception 类型

在 Oracle plsql 中支持用户自定义异常，并对异常进行初始化，赋错误码 sqlcode。Oracle 使用方法如下：

```
errrecord exception;
exception_int(errrecord, -30001);
```

之后使用 raise errrecoed；抛出错误。postgres plpgsql 不支持 oracle 以相同的方式进行自定义异常，因此在迁移的过程中我们需要对这部分内容进行手动修改，以达到相同的目的。LightDB-A 支持直接使用 raise exception 抛出异常，因此，我们在处理时通常转换为以下形式：

```
raise EXCEPTION '(%) An attached analytic workspace is blocking this command',
↪'errrecode';
```

errrecode 为使用 exception 声明的变量。

5.6 INSERT ALL

Oracle 支持复合相同条件的元组，同时插入到不同的表中。目前 LightDB-A 不支持 insert all 语法。

LightDB-A 将会在 2023 年二季度支持 INSERT ALL 语法，目前可以使用如下方案

Oracle insert all 语法：

```
insert all
into t1(object_name,object_id)
into t2(object_name,object_id)
select * from t;
```

LightDB-A insert 语法：

```
with temp as (select * from t;),
Ins1 as (insert into t1(object_name,object_id) select * from temp)
insert into t2(object_name,object_id) select * from temp;
```

5.7 (+) 外关联

LightDB-A 将会在 2023 年二季度支持 (+) 语法

Oracle 支持（+）表示连接。目前 LightDB-A 不支持 oracle (+) 语法。

实际使用过程中，使用 left join 或者 right join 进行替换。如下面方式进行修改：Oracle oracle (+) 语法：

```
Select a.Pro_lightdb_version_number,
b.Pro_em_release_date,
c.Pro_O45_publisher,
from hs_lightDB a, hs_em b, hs_O45 c
where 1 = 1
and a.Pro_lightdb_version_number(+) = c.Pro_O45_version_number
and b.Pro_em_publisher = c. Pro_O45_publisher(+);
```

LightDB-A oracle (+) 语法：

```
select a.Pro_lightdb_version_number,
b.Pro_em_release_date,
c.Pro_O45_publisher
from hs_lightDB a
right join
hs_O45 c on a.Pro_lightdb_version_number = c.Pro_O45_version_number
right join
hs_em b on b.Pro_em_publisher = c.Pro_O45_publisher
where 1 = 1
order by a.Pro_lightdb_version_number asc;
```

转换时需要加上排序条件。

5.8 SQLCODE

Oracle 支持使用 SQLCODE 返回错误码，放回类型为 int，且 SQLCODE 使用场景不局限于 exception when 结构体内，初始化值为 0。目前 LightDB-A 不支持 SQLCODE，但提供 SQLSTATE 作为错误码的返回值，其类型为字符类型，并且使用范围局限于 exception when 结构体内。

实际使用过程中，可以使用 SQLSTATE 替换 SQLCODE。

5.9 MINUS

Oracle 支持使用 MINUS 去做结果集的减法。A minus B 就意味着将结果集 A 去除结果集 B 中所包含的所有记录后的结果，即在 A 中存在，而在 B 中不存在的记录。Oracle 的 minus 是按列进行比较的，所以 A 能够 minus B 的前提条件是结果集 A 和结果集 B 需要有相同的列数，且相同列索引的列具有相同的数据类型。Oracle 会对 minus 后的结果集进行去重，即如果 A 中原本多条相同的记录数在进行 A minus B 后将会只剩一条对应的记录。目前 LightDB-A 不支持 MINUS，但提供 EXCEPT 支持该功能，

实际使用过程中，使用 EXCEPT 替换 MINUS

5.10 sys_guid() 函数

Oracle 支持使用 sys_guid() 去产生并返回一个全球唯一的标识符。目前 LightDB-A 不支持 sys_guid()，插件 uuid-ossp 提供函数 uuid_generate_v4()，

实际使用过程中，使用 uuid_generate_v4 替换 sys_guid。

5.11 q 转义字符

Oracle 支持使用 q 转义字符进行字符转义。目前 LightDB-A 不支持 q 转义字符。

实际使用过程中，使用 E'' 替换 q'[]'。Oracle q 转义字符：

```
select q'[this isn't a good news  $$$$]' from dual;
```

LightDB-A q 转义字符：

```
select e'this isn\'t a good news  $$$$';
```

可以看到除不可见字符，LightDB-A 需要使用进行转义外，单引号 (‘) 本身也需要使用进行转换。

5.12 嵌套表

Oracle 支持使用嵌套表，常见的使用场景有 2 种：

1. 在存储过程使用；
2. 在 ddl 种使用，作为类型使用。

Oracle 嵌套表单一类型：

```
CREATE OR REPLACE TYPE type1 AS TABLE OF VARCHAR2(30);
/

CREATE TABLE nested_table (id NUMBER, col1 type1)
NESTED TABLE col1 STORE AS col1_tab;

INSERT INTO nested_table VALUES (1, type1('A'));
INSERT INTO nested_table VALUES (2, type1('B', 'C'));
INSERT INTO nested_table VALUES (3, type1('D', 'E', 'F'));
```

LightDB-A 兼容嵌套表单一类型：

```
create domain type1 as varchar2(30);
--数组
create table tt1 (id int, info text, nst type1[]);
insert into tt1 values (1,'test',array['abcde'::type1, 'abcde123'::type1]);
```

Oracle 嵌套表复合类型：

```
--创建对象
CREATE TYPE animal_ty AS OBJECT (
breed varchar2(25),
name varchar2(25),
birthdate date);
/
```

(下页继续)

(续上页)

```
--创建类型
CREATE TYPE animals_nt as table of animal_ty;
/
--创建表
create table breeder
(breedername varchar2(25),
animals animals_nt)
nested table animals store as animals_nt_tab;
--插入数据
insert into breeder
values('mary', animals_nt(animal_ty('dog','butch','31-MAR-97'),
animal_ty('dog','rover','31-MAR-97'),
animal_ty('dog','julio','31-MAR-97')));
--查询
select * from table(select animals from breeder);

-- 再次插入：
insert into breeder
values('mary', animals_nt(animal_ty('dog','butch','31-MAR-97'),
animal_ty('dog','rover','31-MAR-97'),
animal_ty('dog','julio','31-MAR-97')));
报错：single-row subquery returns more than one row
```

LightDB-A 使用数组加上复合类型的方法实现嵌套表

```
--复合类型
create type type1 as (c1 int, c2 int, c3 text, c4 timestamp);
--复合类型数组
create table tt1 (id int, info text, nst type1[]);
--插入数组
insert into tt1 values (1,'test',array['(1,2,"abcde","2018-01-01 12:00:00")'::type1,
↪'(2,3,"abcde123","2018-01-01 12:00:00")'::type1]);
select * from unnest((select nst from tt1 limit 1)::type1[]);
```

查询多列报错，与 oracle 一致，只能查询单列。

5.13 嵌套表 extend 方法

Oracle 支持使用 extend 方法，扩充嵌套表的。目前 LightDB-A 不支持嵌套表 extend 方法。

由嵌套表的转换方案可知，LightDB-A 通过数组的形式来兼容 oracle 嵌套表的，且没有设置数组长度，在内存满足的情况下，数据总能存储到数组中，因此不需要使用 extend 进行扩容。删掉嵌套表 extend 方法。

5.14 嵌套表 count 方法

Oracle 支持使用 count 方法，记录当前嵌套表的长度，常用来插入最新的行。目前 LightDB-A 不支持嵌套表 count 方法。

Oracle 嵌套表 count 方法：

```
--创建嵌套表
CREATE OR REPLACE TYPE TP_STRING is table of VARCHAR2(1000);
/
```

(下页继续)

(续上页)

```
--创建函数
CREATE OR REPLACE FUNCTION F_PUB_GET_STRLIST
(
p_inlist_string    char  DEFAULT  ' ',
p_separator_str    char DEFAULT  ','
)
return tp_string
as
v_inlist_string  char(256)  := nvl(trim(p_inlist_string),' ');  --in字符串列表
v_separator_str  char(256)   := nvl(trim(p_separator_str),',');  --分隔符
v_sstr  char(256);
v_data  tp_string;
v_i  int;
v_j  int;
v_len  int;
v_len2  int;
begin
v_sstr  := ' ' ;
v_data  := tp_string() ;
v_i  := 1 ;
v_j  := 0 ;
v_len  := 0 ;
v_len2  := 0 ;
--20150807 zhangxd modify for 支持分隔符以变量的形式传入
v_len  := length(v_inlist_string);
v_len2 := length(v_separator_str);
while v_i <= v_len + 1 loop
   v_j := instr(v_inlist_string, v_separator_str, v_i);
   if v_j = 0 then
      v_j := v_len + 1;
   end if;
   v_sstr := substr(v_inlist_string, v_i, v_j - v_i);
   v_i    := v_j + v_len2;
   dbms_output.put_line(v_sstr);
   v_data.extend;
   v_data(v_data.count) := v_sstr;
end loop;
return v_data;
end f_pub_get_strlist;
/
;
```

LightDB-A 嵌套表 count 方法：

```
--创建数组
create domain TP_STRING as VARCHAR(1000)[];
--创建函数
CREATE OR REPLACE FUNCTION F_PUB_GET_STRLIST
(
p_inlist_string    char(256)  DEFAULT  ' ',  --in字符串列表
p_separator_str    char(256) DEFAULT  ',' --分隔符
)
returns tp_string
as $$
declare
v_inlist_string  char(256) = nvl(trim(p_inlist_string),' ');  --in字符串列表
```

(下页继续)

(续上页)

```
v_separator_str  char(256)   := nvl(trim(p_separator_str),',');  --分隔符
v_sstr  char(256);
v_data  tp_string;
v_i  int;
v_j  int;
v_len  int;
v_len2  int;
begin
v_sstr  := ' ' ;
v_i  := 1 ;
v_j  := 0 ;
v_len  := 0 ;
v_len2  := 0 ;
--20150807 zhangxd modify for 支持分隔符以变量的形式传入
v_len  := length(v_inlist_string);
v_len2 := length(v_separator_str);
while v_i <= v_len + 1 loop
   v_j := instr(v_inlist_string, v_separator_str, v_i);
   if v_j = 0 then
      v_j := v_len + 1;
   end if;
   v_sstr := substr(v_inlist_string, v_i, v_j - v_i);
   v_i    := v_j + v_len2;
   raise notice '%',v_sstr;
   v_data = array_append(v_data, v_sstr::VARCHAR);
   raise notice '%',v_data[1];
end loop;
return v_data;
end;
$$ language plpgsql
;
```

使用表达式 `v_data = array_append(v_data, v_sstr::VARCHAR);` 去替换 `v_data(v_data.count) := v_sstr;` 注意使用 array_append 函数时，需要保证参数左右两边基础类型相同，可以使用:: 去进行强制转换。如定义数组时使用的 varchar，array_append 右边参数也为 varchar.

5.15 unpivot 行转列

Oracle 支持使用 unpivot 行转列。目前 LightDB-A 不支持 unpivot 行转列。转换方案如下：

```
create table hs_unpivot(name varchar(40),chinese int,math int);
insert into hs_unpivot values('zhangsan',90,100);
insert into hs_unpivot values('lisi',88,99);
```

Oracle unpivot 行转列：

```
select * from hs_unpivot unpivot (score for course in(chinese,math));
```

LightDB-A unpivot 行转列：

```
SELECT name, score, course
FROM hs_unpivot,
LATERAL (
VALUES ('chinese', chinese), ('math', math)
) AS unpiv(score, course);
```

5.16 pivot 列转行

Oracle 支持使用 pivot 列转行。目前 LightDB-A 不支持 pivot 列转行。转换方案如下：

```
create table hs_pivot(name varchar(40),chinese int,math int);
insert into hs_pivot values('zhangsan',90,100);
insert into hs_pivot values('lisi',88,99);
```

Oracle pivot 列转行：

```
select * from hs_pivot pivot (sum(score) for course in('chinese','math'));
```

LightDB-A pivot 列转行：

```
select name,
sum(case when course = 'chinese' then score end)  chinese,
sum(case when course = 'math' then score end) math
   from hs_pivot
where course in('chinese','math') group by name;
```

5.17 goto

Oracle 支持使用 goto，跳转到指定的标签。目前 LightDB-A 不支持 goto 跳转到指定的标签。转换方案 Oracle goto:

```
create or replace function hs_goto(i int) return int as
ind int :=0;
begin
ind := i +1;
if i = 1 then
   goto flag;
end if;
   ind := i+2;
<<flag>>
dbms_output.put_line(ind);
return ind;
end;
/
select hs_goto(-1) from dual;

select hs_goto(1) from dual;

Postgres:
create or replace function hs_goto(i int) return int as
goto_flag int := 0;
ind int := 0;
begin
ind := i +1;
if i = 1 then
   goto_flag := 1;
end if;
if goto_flag != 1 then
   ind := i+2;
end if;
dbms_output.put_line(ind);
return ind;
```

(下页继续)

(续上页)

```
end;
/
select hs_goto(-1) from dual;

select hs_goto(1) from dual;
```

解决方案思路：使用变量 goto_flag 去替换 goto 语句，goto 到标签范围内的代码，使用 if 条件进行判断。!goto_flag 为真，则执行该范围内的代码。

6 DDL 语句分布式改造

ora2pg 导出的 DDL 语句都不包含分布式设置，默认情况下使用 hash 分布表，并取主键或者唯一键，或者该表的第一列作为分布式列，需要根据实际情况对 DDL 做分布式改造。

1. 参考分布式数据库特性介绍 为每个表选择合适的分布策略。
2. 分布式表对索引约束有限制，可能需要对原来的索引进行调整。目前主要有以下两个限制:
    1. hash 分布表不支持多个唯一约束 (含主键)
    2. random 分布表不支持唯一约束 (含主键)

分布式改造需要您对实际业务场景、应用的 SQL、数据库性能情况有较深入的了解。