ALIBABA CLOUD

阿里云

云原生数据仓库 AnalyticDB PostgreSQL 版

最佳实践

文档版本：20200923

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.通过pg_stat_activity分析诊断正在执行的SQL

pg_stat_activity是一个非常有用的视图，可以分析排查当前运行的SQL任务以及一些异常问题。
pg_stat_activity 每行展示的是一个“process” 的相关信息，这里的“process”可以理解为一个用户连接。

pg_stat_activity视图的字段描述如下

字段	类型	描述
datid	oid	这个后端连接到的数据库的OID
datname	name	这个后端连接到的数据库的名称
procpid	integer	这个后端的进程 ID （注：4.3 版本支持的字段定义）
pid	integer	这个后端的进程 ID （注：6.0 版本支持的字段定义）
sess_id	integer	登录到这个后端的用户的session id
usesysid	oid	登录到这个后端的用户的 OID
usename	name	登录到这个后端的用户的名称
current_query	text	注：4.3 版本支持的字段定义 这个域显示当前正在执行的查询。 默认情况下，查询文本被截断为1024个字符；可以通过参数 track_activity_query_size更改此值
query	text	注：6.0版本支持的字段定义 这个后端最近查询的文本。如果state为active，这个域显示当前正在执行的查询。在所有其他状态下，它显示上一个被执行的查询。 默认情况下，查询文本被截断为1024个字符；可以通过参数 track_activity_query_size更改此值
waiting	boolean	True，如果当前SQL在锁等待，否则 false
query_start		当前活动查询被开始的时间，如果state不是active，这个域为上一个查询被开始的时间
backend_start		当前后台进程开始的时间
client_addr	inet	连接到这个后端的客户端的 IP 地址。如果这个域为空，它表示客户端通过服务器机器上的一个 Unix 套接字连接或者这是一个内部进程（如自动清理）。
client_port	integer	客户端用以和这个后端通信的 TCP 端口号，如果使用 Unix 套接字则为-1
application_name	text	连接到这个后端的应用的名称

字段	类型	描述
xact_start		这个进程的当前事务被启动的时间，如果没有活动事务则为空。如果当前查询是它的第一个事务，这一列等于query_start。
waiting_reason	text	当前执行等待的原因，可能是等锁或者等待节点间数据的复制
state	text	注：只有6.0版本支持 这个后端目前的状态，包括 active，idle，idle in transaction，idle in transaction (aborted)，fastpath function call，disabled
state_change	timestampz	注：只有6.0版本支持 上次state状态切换的时间

查看连接信息

通过下述SQL就能确认当前的连接用户和对应的连接机器。

```
postgres=# SELECT datname,usename,client_addr,client_port FROM pg_stat_activity ;
datname  |  usename  |  client_addr   | client_port
----------+-----------+----------------+-------------
 postgres | joe       |  xx.xx.xx.xx  |     60621
 postgres | gpmon     |  xx.xx.xx.xx  |     60312
(9 rows)
```

查看SQL运行信息

获取当前用户执行SQL信息：

```
 4.3 版本：
postgres=# SELECT datname,usename,current_query FROM pg_stat_activity ;
 datname  | usename  |                 current_query
----------+----------+-----------------------------------------------------------------
 postgres | postgres | SELECT datname,usename,current_query FROM pg_stat_activity ;
 postgres | joe      | <IDLE>
(2 rows)

6.0 版本：
postgres=# SELECT datname,usename,query FROM pg_stat_activity ;
 datname  | usename  |                 query
----------+----------+-----------------------------------------------------------------
 postgres | postgres | SELECT datname,usename,query FROM pg_stat_activity ;
 postgres | joe      |
(2 rows)
```

只看当前正在运行的SQL信息：

```
4.3 版本
SELECT datname,usename,current_query
  FROM pg_stat_activity
  WHERE current_query != '<IDLE>' ;

6.0 版本
SELECT datname,usename,query
  FROM pg_stat_activity
  WHERE state != 'idle' ;
```

查看耗时较长的查询

查看当前运行中的耗时较长的SQL语句

```
4.3 版本
select current_timestamp - query_start as runtime, datname, usename, current_query
   from pg_stat_activity
   where current_query != '<IDLE>'
   order by 1 desc;

6.0 版本
select current_timestamp - query_start as runtime, datname, usename, query
   from pg_stat_activity
   where state != 'idle'
   order by 1 desc;
```

例如如下返回：

```
runtime    |   datname    | usename |                    current_query
----------------+---------------+----------+-----------------------------------------------------------------------------
----
 00:00:34.248426 | tpch_1000x_col | postgres | select
                            :       l_returnflag,
                            :       l_linestatus,
                            :       sum(l_quantity) as sum_qty,
                            :       sum(l_extendedprice) as sum_base_price,
                            :       sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
                            :       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
                            :       avg(l_quantity) as avg_qty,
                            :       avg(l_extendedprice) as avg_price,
                            :       avg(l_discount) as avg_disc,
                            :       count(*) as count_order
                            : from
                            :       public.lineitem
                            : where
                            :       l_shipdate <= date '1998-12-01' - interval '93' day
                            : group by
                            :       l_returnflag,
                            :       l_linestatus
                            : order by
                            :       l_returnflag,
                            :       l_linestatus;
 00:00:00      | postgres     | postgres | select
                            :      current_timestamp - query_start as runtime,
                            :      datname,
                            :      usename,
                            :      current_query
                            :    from pg_stat_activity
                            :    where current_query != '<IDLE>'
                            :    order by 1 desc;
(2 rows)
```

可以看到第一个查询耗时较久，已经运行了34s还没有结束。

异常SQL诊断及修复

如果一个SQL运行很长时间没有结果，需要检查该SQL还在运行中还是已经被block了：

```
4.3 版本
SELECT datname,usename,current_query
  FROM pg_stat_activity
  WHERE waiting;

6.0 版本
SELECT datname,usename,query
  FROM pg_stat_activity
  WHERE waiting;
```

需要注意的是这个输出只能获取当前因为lock而被block的SQL，因为其他原因被block的SQL这里获取不到。绝大多数情况下SQL都是因为lock而被block，但也会有一些其他情况例如等待i/o、定时器等。如果上述SQL有结果输出说明有SQL被lock阻塞，进一步明确相互block的SQL信息：

```
SELECT
     w.current_query as waiting_query,
     w.procpid as w_pid,
     w.usename as w_user,
     l.current_query as locking_query,
     l.procpid as l_pid,
     l.usename as l_user,
     t.schemaname || '.' || t.relname as tablename
   from pg_stat_activity w
   join pg_locks l1 on w.procpid = l1.pid and not l1.granted
   join pg_locks l2 on l1.relation = l2.relation and l2.granted
   join pg_stat_activity l on  l2.pid = l.procpid
   join pg_stat_user_tables t on l1.relation = t.relid
   where w.waiting;
```

通过这个SQL的输出信息就能确认相互block的SQL和对应的执行pid。在明确了SQL的阻塞信息后，可以通过cancel或者kill query的方式进行恢复。通过cancel取消一个正在运行的query：

```
SELECT pg_cancel_backend(pid)
```

需要在一个运行query的session中执行，如果session本身就是idle的，执行不起作用。另外取消这个query需要花费一定的时间来做清理和事务的回滚。使用pg_terminate_backend来清理idle session，也可以用来终止query：

```
SELECT pg_terminate_backend(pid);
```

该用户的连接会被断开。尽量避免在正在运行query的进程pid上执行。需要注意的是文中提到操作需要用户有superuser的权限。

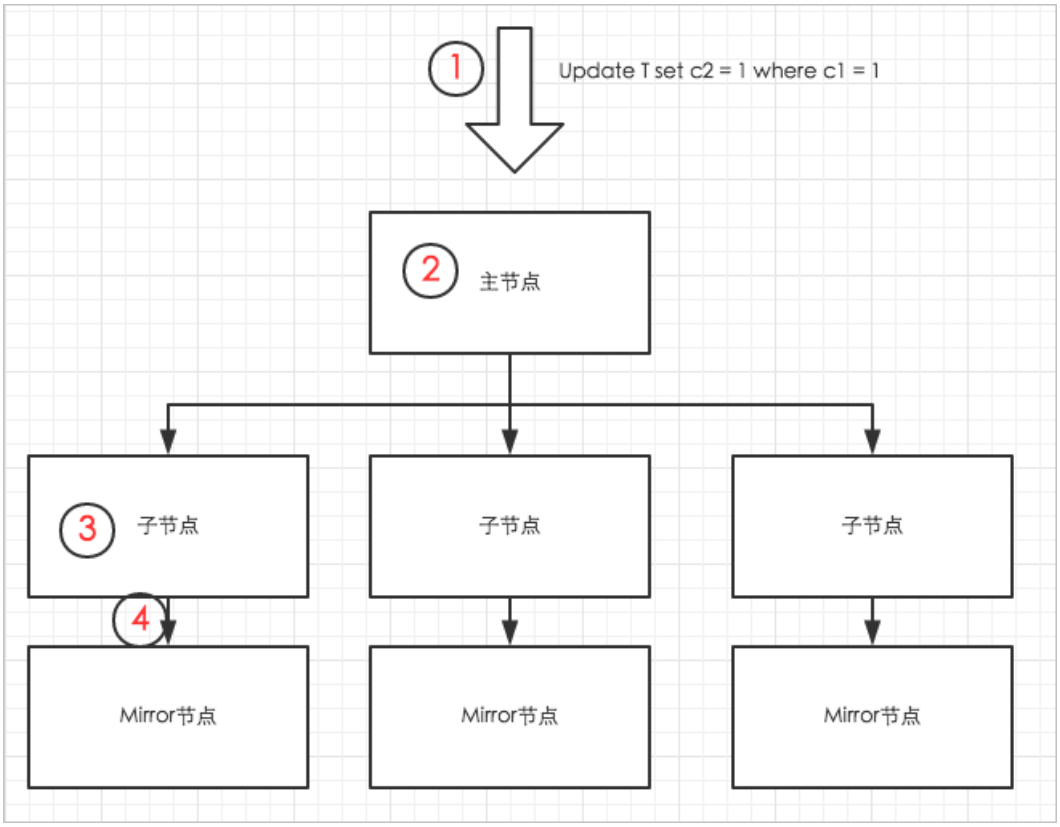
2.使用批量更新

更新，又称为合并（Merge），指把数据最新版本更新到AnalyticDB for PostgreSQL中。如果数据已经存在，则将它们替换为新版本；如果不存在，将它们插入数据库中。这种数据合并一般在离线完成。例如，设置每天一次批量把数据更新到AnalyticDB for PostgreSQL中。也有用户需要实时更新，要求做到分钟级甚至秒级延迟。

本文介绍了AnalyticDB for PostgreSQL中数据合并的方法和背后的原理，进而介绍如何使用批量操作，快速地更新数据。

简单更新过程

无论怎么做数据合并，都是对数据的修改，即执行Update、Delete、Insert、Copy等操作。以用户发起一次Update操作为例（对列存表单行记录的更新），AnalyticDB for PostgreSQL中的数据更新过程如下图所示。



步骤说明如下：

1. 用户发送Update的SQL请求至主节点。
2. 主节点发起分布式事务，对被Update的表加锁（AnalyticDB for PostgreSQL不允许并行Update同一张表），然后把更新请求分发到对应的子节点。
3. 子节点通过索引扫描，定位到要更新的数据，并更新数据。对于列存表，更新逻辑就是删除旧的数据行，并在表的尾端写入新的数据行。列存表中被更新的数据页面会写入内存缓存区，对应的表文件长度的变化（因为尾端写入了数据，所以数据表对应的文件长度增大了）会写入日志（xlog文件）。
4. 在Update命令结束前，内存中被更新的数据页面和xlog日志，都要同步到Mirror节点。同步完成后，主节点结束分布式事务，返回用户执行成功的消息。

可以看出，整个过程的链条很长，SQL语句解析、分布式事务、锁，主节点子节点之间的连接建立、子节点与Mirror数据和日志同步等操作，都会耗费CPU或I/O资源，同时拖慢整个请求的响应时间。因此，对于AnalyticDB for PostgreSQL来说，应该尽量避免单行数据的更新，尽量批量地更新数据，即：

- 把更新放到一个SQL语句，减少语句解析、节点通信、数据同步等开销。
- 把更新放到一个事务，避免不必要的事务开销。

简而言之，就是尽量以“成批”的形式进行数据的合并和更新。

批量Update

怎样用一个SQL实现多个独立数据行的Update？

1.准备目标表

假设有张表需要更新（称为目标表，target_table），这张表的定义如下。

```
create table target_table(c1 int, c2 int, primary key (c1));
insert into target_table select generate_series(1, 10000000);
```

一般目标表都非常大，这里假设往target_table里面插入1千万行数据。为了能快速更新，target_table上要有索引。这里定义了primary key，会隐含地创建一个唯一值索引（unique index）。

2.准备中间表

为了做批量Update，需要用到中间表（Stage Table，示例中的source_table），即为了更新数据临时创建的表。为了更新target_table的数据，可以先把新数据插入到中间表source_table中。然后，把新数据通过COPY命令、OSS外部表等方式导入到source_table。以下示例直接插入了一些数据。

```
create table source_table(c1 int, c2 int);
insert into source_table select generate_series(1, 100), generate_series(1,100);
```

3.批量更新

source_table数据准备好后，执行如下 `update set ... from ... where ..` 语句，即可实现批量的Update。

注意：为了最大限度的使用到索引，在执行Update前，要使用 `set opitimzer=on` 启用ORCA优化器（如果不启用ORCA优化器，则需要执行 `set enable_nestloop = on` 才能使用到索引）。

```
set optimizer=on;
update target_table set c2 = source_table.c2 from source_table where target_table.c1= source_table.c
1;
```

该Update的执行计划如下：

```
=> explain update target_table set c2 = source_table.c2 from source_table where target_table.c1= sou
rce_table.c1;
                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
---
 Update  (cost=0.00..586.10 rows=25 width=1)
   ->  Result  (cost=0.00..581.02 rows=50 width=26)
       ->  Redistribute Motion 4:4  (slice1; segments: 4)  (cost=0.00..581.02 rows=50 width=22)
           Hash Key: public.target_table.c1
           ->  Assert  (cost=0.00..581.01 rows=50 width=22)
               Assert Cond: NOT public.target_table.c1 IS NULL
               ->  Split  (cost=0.00..581.01 rows=50 width=22)
                   ->  Nested Loop  (cost=0.00..581.01 rows=25 width=18)
                       Join Filter: true
                       ->  Table Scan on source_table  (cost=0.00..431.00 rows=25 width=8)
                       ->  Index Scan using target_table_pkey on target_table  (cost=0.00..150.01 rows=1 wi
dth=14)
                           Index Cond: public.target_table.c1 = source_table.c1
```

可以看到，AnalyticDB for PostgreSQL选择了索引。但是，如果往source_table里面加入更多数据，优化器会认为使用Nest Loop关联方法+索引扫描，没有不使用索引高效，而会选取Hash关联方法+表扫描方式来执行。例如：

```
postgres=> insert into source_table select generate_series(1, 1000), generate_series(1,1000);
INSERT 0 1000
postgres=> analyze source_table;
ANALYZE
postgres=> explain update target_table set c2 = source_table.c2 from source_table where target_tabl
e.c1= source_table.c1;
                                QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Update  (cost=0.00..1485.82 rows=275 width=1)
   ->  Result  (cost=0.00..1429.96 rows=550 width=26)
         ->  Assert  (cost=0.00..1429.94 rows=550 width=22)
               Assert Cond: NOT public.target_table.c1 IS NULL
               ->  Split  (cost=0.00..1429.93 rows=550 width=22)
                     ->  Hash Join  (cost=0.00..1429.92 rows=275 width=18)
                           Hash Cond: public.target_table.c1 = source_table.c1
                           ->  Table Scan on target_table  (cost=0.00..477.76 rows=2500659 width=14)
                           ->  Hash  (cost=431.01..431.01 rows=275 width=8)
                                 ->  Table Scan on source_table  (cost=0.00..431.01 rows=275 width=8)
```

上述批量Update方式，减少了SQL编译、节点间通信、事务等开销，可以大大提升数据更新性能并减少对资源的消耗。

批量Delete

对于Delete操作，采用和上述批量Update类似的中间表，然后使用下面的带有“Using”子句的Delete来实现批量删除：

```
delete from target_table using source_table where target_table.c1 = source_table.c1;
```

可以看到，这种批量的Delete同样使用了索引。

```
explain delete from target_table using source_table where target_table.c1 = source_table.c1;
                                QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Delete (slice0; segments: 4)  (rows=50 width=10)
   ->  Nested Loop  (cost=0.00..41124.40 rows=50 width=10)
         ->  Seq Scan on source_table  (cost=0.00..6.00 rows=50 width=4)
         ->  Index Scan using target_table_pkey on target_table  (cost=0.00..205.58 rows=1 width=14)
               Index Cond: target_table.c1 = source_table.c1
```

利用Delete + Insert做数据合并

如何实现批量的数据合并？在做数据合并时，需要先把待合入的数据放入中间表中。

- 如果预先知道待合入的数据，在目标表中都已经有对应的数据行，即可通过Update语句实现数据合入。
- 但多数情况下，待合入的数据中，一部分是在目标表中已存在记录的数据，还有一部分是新增的，目标表中没有对应记录。这种情况下，需要使用一次批量Delete+一次批量Insert。代码示例如下。

```
set optimizer=on;
delete from target_table using source_table where target_table.c1 = source_table.c1;
insert into target_table select * from source_table;
```

利用Values()表达式做实时更新

使用中间表，需要维护中间表生命周期。有的用户想实时批量更新数据到AnalyticDB for PostgreSQL，即持续性地同步数据或合并数据到AnalyticDB for PostgreSQL 。

如果采用上面的方法，需要反复创建、删除（或Truncate）中间表。其实，可以利用Values表达式，达到类似中间表的效果，却不用维护表。方法是先将待更新的数据拼成一个Values表达式，然后按如下方式执行Update或Delete：

```
update target_table set c2 = t.c2 from (values(1,1),(2,2),(3,3),...(2000,2000)) as t(c1,c2) where target_ta
ble.c1=t.c1
delete from target_table using (values(1,1),(2,2),(3,3),...(2000,2000)) as t(c1,c2) where target_table.c1
= t.c1
```

> **注意** 使用 `set optimizer=on;` 或 `set enable_nestloop=on;` 都可以生成使用索引的查询计划。但在比较复杂的情形下，比如索引字段有多个、涉及分区表等，必须要使用ORCA优化器才能匹配上索引。

3.查询性能优化指导

本文介绍在不同操作场景下使用云数据库AnalyticDB for PostgreSQL时的一些具体建议。选择合适的操作实践将有效地帮助您提高AnalyticDB for PostgreSQL的性能。

收集统计信息

AnalyticDB for PostgreSQL的优化器在进行查询优化时，会根据统计信息进行查询代价估算和优化。如果参与查询的表没有收集过统计信息或统计信息过旧，系统将按照默认值或老旧的统计信息进行优化，往往无法生成最优执行计划。所以，建议在大批量数据加载完成，或者有较多数据（超过20%）更新后，进行统计信息收集。

采用 `ANALYZE` 命令收集统计信息时，可以对所有表收集、对某个表的所有列收集或对表的指定列收集。对于大部分用户，建议采用对所有表收集或对表的所有列收集的方式。如果想对统计信息收集环节做精细化控制，可以采用对表的指定列收集方式，针对关联（JOIN）的条件列、过滤条件列、有索引的列进行统计信息收集。

示例

- 收集所有表的统计信息示例（推荐数据大批量入库后使用）：

```
ANALYZE;
```

- 收集表t的所有列的统计信息示例（推荐某个表插入/更新/删除较多数据后使用）：

```
ANALYZE t;
```

- 收集表t的a列的统计信息示例：

```
ANALYZE t(a);
```

两种优化器的选择

AnalyticDB for PostgreSQL有两个SQL优化器可供选择，两个优化器在不同的场景下，各有优势。

- **Legacy优化器**

  Legacy优化器的SQL优化耗时较短，适合高并发的简单查询场景（3表以内关联），或者高并发的数据写入或更新场景（INSERT/UPDATE/DELTE）。

- **ORCA优化器**

ORCA优化器为面向复杂SQL语句的优化器，会遍历更多执行路径，制定最优执行计划，但SQL优化过程相对耗时稍长。建议对复杂查询（3表以上关联为主的场景）为主的 ETL场景和报表场景采用ORCA优化器。此外，ORCA优化器具有相关子查询的解关联优化及动态分区裁剪优化等能力，含有相关子查询的语句及含有带参数化过滤条件的分区表的语句建议使用ORCA优化器。

Session会话级设置方式：

```
-- 使用Legacy优化器
set optimizer = off;
-- 使用ORCA优化器
set optimizer = on;
```

查看当前的优化器的方式：

```
show optimizer;
-- 值为on：表示当前优化器为ORCA优化器
-- 值为off：表示当前优化器为Legacy优化器
```

> 说明
> - ADB PG 4.3版本的默认优化器为Legacy优化器，ADB PG 6.0版本的默认优化器为ORCA优化器。
> - 实例级别设置请提工单。

使用索引加速查询

当查询中有等值过滤条件或范围过滤条件，且过滤后数据量较少时，可考虑在条件列上建立索引，提升数据扫描的速度。AnalyticDB for PostgreSQL目前支持3种索引：

- BTree：适用于唯一值较多的数据列，列上有过滤条件或Join条件，或为排序列。
- Bitmap：适用于唯一值较少的数据列，查询中该列上有多个过滤条件。
- Gist：适用于地理位置、范围数据类型、图像特征值数据、几何类数据等。

示例

无索引时，带条件的表数据获取采用全表扫描再进行过滤的方式：

```
postgres=# EXPLAIN SELECT * FROM t WHERE b = 1;
                        QUERY PLAN
------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..431.00 rows=1 width=16)
   ->  Table Scan on t  (cost=0.00..431.00 rows=1 width=16)
         Filter: b = 1
 Settings:  optimizer=on
 Optimizer status: PQO version 1.609
(5 rows)
```

使用如下的语句在t表的b列上建立BTree索引：

```
postgres=# CREATE INDEX i_t_b ON t USING btree (b);
CREATE INDEX
```

有索引时，带条件的表数据获取采用索引方式：

```
postgres=# EXPLAIN SELECT * FROM t WHERE b = 1;
                        QUERY PLAN
-------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..2.00 rows=1 width=16)
   ->  Index Scan using i_t_b on t  (cost=0.00..2.00 rows=1 width=16)
         Index Cond: b = 1
 Settings:  optimizer=on
 Optimizer status: PQO version 1.609
(5 rows)
```

查看执行计划

执行计划是数据库运行SQL的步骤，相当于算法。查看查询的执行计划有助于分析查询的执行过程，分析慢SQL的瓶颈点，帮助我们明确优化方向。可以通过在查询前加explain关键字，查看查询的执行计划，此时只显示查询的执行计划，而不会执行该语句。也可以在查询前加explain analyze关键字，其会运行该语句，收集查询时真实的执行信息，并在查询计划上显示出来。

- explain示例：

  ```
  postgres=# EXPLAIN SELECT a, b FROM t;
                          QUERY PLAN
  ------------------------------------------------------------------------------
   Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..4.00 rows=100 width=8)
     ->  Seq Scan on t  (cost=0.00..4.00 rows=34 width=8)
   Optimizer status: legacy query optimizer
  (3 rows)
  ```

- explain analyze示例：

```
postgres=# EXPLAIN ANALYZE SELECT a, b FROM t;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
-------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..4.00 rows=100 width=8)
   Rows out:  100 rows at destination with 2.728 ms to first row, 2.838 ms to end, start offset by 0.418
ms.
   ->  Seq Scan on t  (cost=0.00..4.00 rows=34 width=8)
         Rows out:  Avg 33.3 rows x 3 workers.  Max 37 rows (seg2) with 0.088 ms to first row, 0.107 ms t
o end, start offset by 2.887 ms.
 Slice statistics:
   (slice0)    Executor memory: 131K bytes.
   (slice1)    Executor memory: 163K bytes avg x 3 workers, 163K bytes max (seg0).
 Statement statistics:
   Memory used: 128000K bytes
 Optimizer status: legacy query optimizer
 Total runtime: 3.739 ms
(11 rows)
```

执行计划由一系列的算子及其信息按照执行逻辑顺序有机组合在一起，并按此以流水线方式执行，进行数据处理。

算子种类：

- 数据扫描算子：Seq Scan、Table Scan、Index Scan、Bitmap Scan等。
- 连接算子：Hash Join、Nested Loop、Merge Join
- 聚集算子：Hash Aggregate、Group Aggregate
- 分布式算子：Redistribute Motion、Broadcast Motion、Gather Motion
- 其他算子：Hash、Sort、Limit、Append等

```
postgres=# EXPLAIN SELECT * FROM t1, t2 WHERE t1.b = t2.b;
                                QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Gather Motion 3:1  (slice3; segments: 3)  (cost=0.00..862.00 rows=1 width=32)
   ->  Hash Join  (cost=0.00..862.00 rows=1 width=32)
       Hash Cond: t1.b = t2.b
       ->  Redistribute Motion 3:3  (slice1; segments: 3)  (cost=0.00..431.00 rows=1 width=16)
           Hash Key: t1.b
           ->  Table Scan on t1  (cost=0.00..431.00 rows=1 width=16)
       ->  Hash  (cost=431.00..431.00 rows=1 width=16)
           ->  Redistribute Motion 3:3  (slice2; segments: 3)  (cost=0.00..431.00 rows=1 width=16)
               Hash Key: t2.b
               ->  Table Scan on t2  (cost=0.00..431.00 rows=1 width=16)
 Settings:  optimizer=on
 Optimizer status: PQO version 1.609
(12 rows)
```

在上述执行计划中，进行了如下的计算过程：

1. Table Scan算子对表t1和t2进行表扫描操作。
2. Redistribute Motion算子分别按照t1.b和t2.b的Hash值对t1表和t2表进行了数据重分布（Redistribute Motion），将数据重新分配到了各个节点上，以便于进行Join计算。
3. Hash算子在t2表上建了一个用于Join的Hash表。
4. Hash Join算子对两个表的数据做了Join计算。
5. Gather Motion算子将计算结果传输到前端总控节点，进而传输到客户端。

执行计划的整体结构如上所述，具体执行计划会随查询语句的不同发生变化。

消除分布式(Motion)算子提升性能

在进行连接或聚集操作时，AnalyticDB for PostgreSQL会根据数据分布情况添加分布式算子，对数据进行重分布（Redistribute Motion）或广播（Broadcast Motion）。分布式算子会占用大量的网络资源。如果能够通过建表和业务逻辑进行分布式算子的规避，则能够提升数据库查询性能。

基本原理

如果表定义时设置的分布键与业务逻辑并不匹配，则需对分布键进行调整，尽可能减少查询中的分布式算子。

示例

```
SELECT * FROM t1, t2 WHERE t1.a=t2.a;
```

其中，t1表的分布键为t1.a。

- 如果t2表的分布列是t2.b，会出现t2表的重分布。

```
postgres=# EXPLAIN SELECT * FROM t1, t2 WHERE t1.a=t2.a;
                                QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Gather Motion 3:1  (slice2; segments: 3)  (cost=0.00..862.00 rows=1 width=32)
   ->  Hash Join  (cost=0.00..862.00 rows=1 width=32)
         Hash Cond: t1.a = t2.a
         ->  Table Scan on t1  (cost=0.00..431.00 rows=1 width=16)
         ->  Hash  (cost=431.00..431.00 rows=1 width=16)
               ->  Redistribute Motion 3:3  (slice1; segments: 3)  (cost=0.00..431.00 rows=1 width=16)
                     Hash Key: t2.a
                     ->  Table Scan on t2  (cost=0.00..431.00 rows=1 width=16)
 Settings:  optimizer=on
 Optimizer status: PQO version 1.609
(10 rows)
```

- 如果t2表的分布列是t2.a，则无需重分布就可以直接Join。

```
postgres=# EXPLAIN SELECT * FROM t1, t2 WHERE t1.a=t2.a;
                        QUERY PLAN
-----------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..862.00 rows=1 width=32)
   ->  Hash Join  (cost=0.00..862.00 rows=1 width=32)
         Hash Cond: t1.a = t2.a
         ->  Table Scan on t1  (cost=0.00..431.00 rows=1 width=16)
         ->  Hash  (cost=431.00..431.00 rows=1 width=16)
               ->  Table Scan on t2  (cost=0.00..431.00 rows=1 width=16)
 Settings:  optimizer=on
 Optimizer status: PQO version 1.609
(8 rows)
```

优化关联(JOIN)列的数据类型

Join的条件列数据类型应一致，避免因隐式/显式数据类型转换带来数据需要重分布的问题。

- **显式数据类型转换**

  显式数据类型转换是指在SQL语句中，对Join条件列的数据类型进行强制类型转换。比如表t的a列是int类型，但是在Join条件中将其转换为numeric类型（即有t.a::numeric的转换）。

  数据进行类型转换后，其hash函数/hash值会发生变化，SQL语句中应尽量避免在Join条件列上进行类型转换。

  下面例子表明在数据类型转换后，会导致数据的重分布：

```
--无数据类型转换
postgres=# EXPLAIN SELECT * FROM t1, t2 WHERE t1.a=t2.a;
                        QUERY PLAN
------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..862.00 rows=1 width=32)
   ->  Hash Join  (cost=0.00..862.00 rows=1 width=32)
       Hash Cond: t1.a = t2.a
       ->  Table Scan on t1  (cost=0.00..431.00 rows=1 width=16)
       ->  Hash  (cost=431.00..431.00 rows=1 width=16)
           ->  Table Scan on t2  (cost=0.00..431.00 rows=1 width=16)
 Settings:  optimizer=on
 Optimizer status: PQO version 1.609
(8 rows)

--有强制数据类型转换
postgres=# EXPLAIN SELECT * FROM t1, t2 WHERE t1.a=t2.a::numeric;
                                QUERY PLAN
---------------------------------------------------------------------------------------------------------
 Gather Motion 3:1  (slice3; segments: 3)  (cost=0.00..862.00 rows=1 width=32)
   ->  Hash Join  (cost=0.00..862.00 rows=1 width=32)
       Hash Cond: t1.a::numeric = t2.a::numeric
       ->  Redistribute Motion 3:3  (slice1; segments: 3)  (cost=0.00..431.00 rows=1 width=16)
           Hash Key: t1.a::numeric
           ->  Table Scan on t1  (cost=0.00..431.00 rows=1 width=16)
       ->  Hash  (cost=431.00..431.00 rows=1 width=16)
           ->  Redistribute Motion 3:3  (slice2; segments: 3)  (cost=0.00..431.00 rows=1 width=16)
               Hash Key: t2.a::numeric
               ->  Table Scan on t2  (cost=0.00..431.00 rows=1 width=16)
 Settings:  optimizer=on
 Optimizer status: PQO version 1.609
(12 rows)
```

- **隐式数据类型转换**

  隐式数据类型转换是指在Join条件两边数据类型不完全一致，会导致数据库对其中一列的数据类型进行转换。

  数据库对其中一列进行数据类型转换后，可能会引发原始数据类型与新数据类型的Hash函数/Hash值不一致，需要进行重分布。所以，在表设计阶段，参与Join的两个表，Join条件的类型应尽可能统一，避免因为数据类型不同而导致需要额外重分布数据的问题。

  下面的例子中，t1.a为timestamp without time zone类型，t2.a为timestamp with time zone类型，他们的Hash函数不一致，在Join时需要进行重分布后再进行Join。

```
postgres=# CREATE TABLE t1 (a timestamp without time zone);
CREATE TABLE
postgres=# CREATE TABLE t2 (a timestamp with time zone);
CREATE TABLE
postgres=#
postgres=# EXPLAIN SELECT * FROM t1, t2 WHERE t1.a=t2.a;
                            QUERY PLAN
-------------------------------------------------------------------------------------------------
 Gather Motion 3:1  (slice2; segments: 3)  (cost=0.04..0.11 rows=4 width=16)
   ->  Nested Loop  (cost=0.04..0.11 rows=2 width=16)
       Join Filter: t1.a = t2.a
       ->  Seq Scan on t1  (cost=0.00..0.00 rows=1 width=8)
       ->  Materialize  (cost=0.04..0.07 rows=1 width=8)
           ->  Broadcast Motion 3:3  (slice1; segments: 3)  (cost=0.00..0.04 rows=1 width=8)
               ->  Seq Scan on t2  (cost=0.00..0.00 rows=1 width=8)
(7 rows)
```

数据倾斜的检查和处理

如果出现查询异常缓慢，或者资源利用率不均匀的情况，则需要确认是否出现了数据倾斜。

可以通过如下方式检查数据是否发生倾斜：检查某个表在各个节点上的数据分布计数，如果各节点上的数据分布明显不均匀，则需要对该表的分布键进行调整。

```
postgres=# SELECT gp_segment_id, count(1) FROM t1 GROUP BY 1 ORDER BY 2 DESC;
 gp_segment_id | count
---------------+-------
             0 | 16415
             2 |    37
             1 |    32
(3 rows)
```

如果发生倾斜，建议重新合理定义分布键，更改分布键有两种方式：

- 重新建表：新建表时调整分布键。
- 直接更改表的分布键：`ALTER TABLE t1 SET DISTRIBUTED BY (b);`

查看正在运行的语句状态

数据库中正在并发执行的语句过多，会导致系统资源不足，查询执行缓慢。

通过pg_stat_activity视图查看数据库的运行状况，该视图将列出系统中所有正在并发执行的语句，可以通过观察query_start字段（该查询的执行开始时间）来判断某查询是否有执行时长上的异常。

例如：

```
postgres=# SELECT * FROM pg_stat_activity;
 datid | datname  | procpid | sess_id | usesysid |  usename   |        current_query        | waiting |         qu
ery_start         |         backend_start         | client_addr | client_port | application_name |         xact_start
| waiting_reason
-------+----------+---------+---------+----------+------------+-----------------------------+---------+-----------
-------------------+-------------------------------+-------------+-------------+------------------+-------------------
------------+---------------
 10902 | postgres |   53666 |       7 |       10 | yineng.cyn | select * from pg_stat_activity; | f       | 2019-05-13 2
0:27:12.058656+08 | 2019-05-13 20:16:14.179612+08 |             |          -1 | psql             | 2019-05-13 20:27:12.0
58656+08 |
 10902 | postgres |   54158 |       9 |       10 | yineng.cyn | select * from t t1, t t2;       | f       | 2019-05-13 20:26
:28.138641+08 | 2019-05-13 20:17:40.368623+08 |             |          -1 | psql             | 2019-05-13 20:26:28.13864
1+08 |
(2 rows)
```

其中，比较关键的字段为：

- procpid：执行该查询的Master进程号。
- usename：执行该查询的用户名。
- current_query：查询文本。
- waiting：查询是否处于等待状态。
- query_start：查询执行开始时间。
- backend_start：执行该查询的进程启动时间。
- xact_start：该查询所在事务开始时间。
- waiting_reason：查询等待的原因。

此外，可以通过在 `current_query` 列添加 `current_query != '<IDLE>'` 条件，查看正在运行的SQL的信息：

```
SELECT * FROM pg_stat_activity WHERE current_query != '<IDLE>';
```

查看耗时最长的5条语句：

```
SELECT current_timestamp - query_start as runtime
, datname
, usename
, current_query
FROM pg_stat_activity
WHERE current_query != '<IDLE>'
ORDER BY runtime DESC
LIMIT 5;
```

判断当前锁的状况

如果数据库中的对象被加锁，并且长时间没有释放，则可能导致其他查询一直处于等锁状态，会影响其他查询的正常执行。通过如下语句查看数据库中被锁的表：

```
SELECT pgl.locktype AS locktype
, pgl.database AS database
, pgc.relname AS relname
, pgl.relation AS relation
, pgl.transaction AS transaction
, pgl.pid AS pid
, pgl.mode AS mode
, pgl.granted AS granted
, pgsa.current_query AS query
FROM pg_locks pgl
JOIN pg_class pgc ON pgl.relation = pgc.oid
JOIN pg_stat_activity pgsa ON pgl.pid = pgsa.procpid
ORDER BY pgc.relname;
```

如果检查出某个查询被hang住，其原因为在等锁，可查看查询涉及的表被加锁的情况，如有必要可采用如下方式进行人工干预。

- 取消进行中的查询，如果该pid中无查询，为IDLE状态，则执行不起作用。另外取消这个query需要花费一定的时间来做清理和事务的回滚。

  ```
  SELECT pg_cancel_backend(pid);
  ```

  > **说明** `pg_cancel_backend` 对 `pg_stat_activity.current_query` 为<IDLE>状态的session不起作用，可以用 `pg_terminate_backend` 来清理。

- 中断session，如果有未提交事务，也会被回滚掉。

  ```
  SELECT pg_terminate_backend(pid);
  ```

使用 Nest Loop JOIN 提升性能

在实例缺省状态下，AnalyticDB for PostgreSQL没有启用Nested Loop JOIN（嵌套连接）。对于只涉及或返回少部分数据的查询，性能可能不是最优的。

例如下面的SQL语句：

```
SELECT *
FROM t1 join t2 on t1.c1 = t2.c1
WHERE t1.c2 >= '230769548' and t1.c2 < '230769549'
LIMIT 100;
```

其特点是t1和t2表都比较大，t1上的选择条件 `（t1.c2 >= '230769548' and t1.c2 < '23432442'）` 过滤了绝大多数数据记录，且有LIMIT子句，所以查询实际上只涉及总数据量中的一小部分。这种情况下，使用Nested Loop连接方式是较优的。

要使用Nested Loop连接，需要执行一下SET命令，如下所示：

```
show enable_nestloop ;
 enable_nestloop
-----------------
 off
SET enable_nestloop = on ;
show enable_nestloop ;
 enable_nestloop
-----------------
 on
explain SELECT * FROM t1 join t2 on t1.c1 = t2.c1 WHERE t1.c2 >= '230769548' and t1.c2 < '23432442' LIMIT
100;
                                QUERY PLAN
-------------------------------------------------------------------------------------------------
 Limit  (cost=0.26..16.31 rows=1 width=18608)
   ->  Nested Loop  (cost=0.26..16.31 rows=1 width=18608)
         ->  Index Scan using t1 on c2  (cost=0.12..8.14 rows=1 width=12026)
               Filter: ((c2 >= '230769548'::bpchar) AND (c2 < '230769549'::bpchar))
         ->  Index Scan using t2 on c1  (cost=0.14..8.15 rows=1 width=6582)
               Index Cond: ((c1)::text = (T1.c1)::text)
```

可以发现，t1和t2两张表是使用Nested Loop连接的，从而获得了较优的性能。

4.维护定期回收空间任务

当系统有更新操作（包括INSERT VALUES、UPDATE、DELETE、ALTER TABLE ADD COLUMN等），会在系统表和被更新的数据表中留存不再使用的垃圾数据，造成系统性能下降，并占用大量磁盘空间，因此需要定期进行垃圾回收。本文介绍了垃圾回收的方法。

不锁表回收垃圾

在不锁表的情况下，可以回收部分垃圾。具体方式如下：

- 命令：连接每个数据库，以数据库的所有者身份登录，执行 `VACUUM` 命令。
- 频率：
  - 如果有大批量实时更新的情况（即不断执行INSERT VALUES、UPDATE、DELETE等操作），最好每天执行一次，或每周至少一次。
  - 如果更新是每天一次批量进行的，建议每周执行一次，或不要超过一个月执行一次。
- 对系统影响：不会锁表，表可以正常读写。会导致CPU、I/O使用率增加，可能影响查询的性能。
- 示例的脚本文件：可以使用如下的Linux Shell脚本文件，作为crontab定期任务来执行。

```
#!/bin/bash
export PGHOST=myinst.gpdb.rds.tbsite.net
export PGPORT=3432
export PGUSER=myuser
export PGPASSWORD=mypass
#do not echo command, just get a list of db
dblist=`psql -d postgres -c "copy (select datname from pg_stat_database) to stdout"`
for db in $dblist ; do
   #skip the system databases
   if [[ $db == template0 ]] ||  [[ $db == template1 ]] || [[ $db == postgres ]] || [[ $db == gpdb ]] ; then
      continue
   fi
   echo processing $db
   #vacuum all tables (catalog tables/user tables)
   psql -d $db -e -a -c "VACUUM;"
done
```

维护窗口回收垃圾

在业务暂停的维护窗口，可以回收所有垃圾。具体方式如下：

- 命令：连接每个数据库，以数据库的所有者身份登录（需要对所有操作对象有所有者权限）。
  i. 执行 `REINDEX SYSTEM <database name>` 。
  ii. 对每张数据表，执行 `VACUUM FULL <table name>` ， `REINDEX TABLE <table name>` 。
  iii. 对于系统表（包括pg_class, pg_attribute, pg_index等），当有频繁建删表，建删索引等操作时，

也建议执行 VACUUM FULL <table name>进行定期维护。注意：该操作需要业务停止访问数据库。

- 频率：至少每周执行一次。如果每天会更新几乎所有数据，需要每天做一次。
- 对系统影响：会对正在进行VACUUM FULL或REINDEX的表进行锁定，无法读写。会导致CPU、I/O使用率增加。
- 示例的脚本文件：可以使用如下的Linux Shell脚本文件，作为crontab定期任务来执行。

```
#!/bin/bash
export PGHOST=myinst.gpdb.rds.tbsite.net
export PGPORT=3432
export PGUSER=myuser
export PGPASSWORD=mypass
#do not echo command, just get a list of db
dblist=`psql -d postgres -c "copy (select datname from pg_stat_database) to stdout"`
for db in $dblist ; do
   #skip system databases
   if [[ $db == template0 ]] ||  [[ $db == template1 ]] || [[ $db == postgres ]] || [[ $db == gpdb ]] ; then
      continue
   fi
   echo processing db "$db"
   #do a normal vacuum
   psql -d $db -e -a -c "VACUUM;"
   #reindex system tables firstly
   psql -d $db -e -a -c "REINDEX SYSTEM $db;"
   #use a temp file to store the table list, which could be vary large
   cp /dev/null tables.txt
   #query out only the normal user tables, excluding partitions of parent tables
   psql -d $db -c "copy (select '\"'||tables.schemaname||'\".' || '\"'||tables.tablename||'\"' from (select nsp
name as schemaname, relname as tablename from pg_catalog.pg_class, pg_catalog.pg_namespace, pg
_catalog.pg_roles where pg_class.relnamespace = pg_namespace.oid and pg_namespace.nspowner =
pg_roles.oid and pg_class.relkind='r' and (pg_namespace.nspname = 'public' or pg_roles.rolsuper = 'fals
e' ) ) as tables(schemaname, tablename) left join pg_catalog.pg_partitions on pg_partitions.partitionsc
hemaname=tables.schemaname and pg_partitions.partitiontablename=tables.tablename where pg_pa
rtitions.partitiontablename is null) to stdout;" > tables.txt
   while read line; do
      #some table name may contain the $ sign, so escape it
      line=`echo $line |sed 's/\\$/\\\\\\$/g'`
      echo processing table "$line"
      #vacuum full this table, which will lock the table
      psql -d $db -e -a -c "VACUUM FULL $line;"
      #reindex the table to reclaim index space
      psql -d $db -e -a -c "REINDEX TABLE $line;"
   done <tables.txt
done
```

5.导入中特殊符号处理

云数据库AnalyticDB for PostgreSQL支持多种数据导入方法，数据迁移及同步方案综述。

在通过OSS高速并行导入和通过COPY数据导入过程中，经常因为存在特殊字符导致导入失败。本文介绍了预先处理导入数据中的特殊字符的方法，从而消除特殊字符带来的问题。

OSS高速并行导入

在数据导入过程中，一般是将文件的每行作为一个元组，通过在每行中规定分隔符来分割每一列的数据。下文首先介绍分隔符的使用方法和约束，然后介绍在每列中遇到特殊符号的处理方法。

分隔符

在创建OSS外部表语法中，您可以通过在FORMAT子句后面指定DELIMITER分隔符，如下：

```
FORMAT 'TEXT' (DELIMITER ',')
```

- 如果 `FORMAT 'TEXT'` ，则 `DELIMITER` 缺省值为 `'\t'` 。
- 如果 `FORMAT 'CSV'` ，则 `DELIMITER` 缺省值为 `','` 。

您也可以自定义DELIMITER，但是创建外部表语法中自定义的DELIMITER必须满足以下约束：

- 必须是一个ASCII字符，不允许是汉字或者2个以及以上ASCII字符。
- 不支持 `'\n'` 和 `'\r'` 。
- 支持除 `'\n'` 和 `'\r'` 之外的其他转义字符，使用时前面加E或者e。
- 支持前面不加E的转义字符 `'\t'` 。
- 如果是TEXT模式，可以设置DELIMITER为OFF，支持单列外部表。

为了能够正常读取数据，您提供的OSS文件内容必须严格遵守设置的DELIMITER。

数据中的特殊符号

在数据导入过程中，出现特殊符号的场景可以分为以下几种：

- 列中存在和DELIMITER相同的字符。
  - 如果您使用TEXT模式，则需要在每个DELIMITER字符前加ESCAPE符。ESCAPE符可以在创建外表时使用以下命令指定，缺省值为反斜杠（\）。

    ```
    FORMAT 'TEXT' (ESCAPE '\')
    ```

  - 如果您使用的是CSV模式，则需要在每个DELIMITER字符前加双引号（”）。
- 列中存在中文。OSS外表支持中文数据，但是为了保证显示正确，您需要在创建外表时设置如下编码格式：

  ```
  ENCODING 'UTF8'
  ```

- 列中存在null。您可以设置null对应的匹配字符，在导入数据时将对应的字符匹配为null。CSV模式下缺省值为不带引号的空值，TEXT模式下缺省值为\N。以下命令将空格作为null的匹配字符，如果该列为空格，则在使用OSS文件导入的数据中该列值为null。

```
FORMAT 'text' (null ' ')
```

- 列中存在转义字符。您可以在转义字符前增加ESCAPE符。ESCAPE符在创建外表时指定，CSV模式缺省值为双引号（”），TEXT模式缺省值为反斜杠（\）。
  - 您可以自定义ESCAPE为单个字符。例如，以下命令将ESCAPE设置为反斜杠：

    ```
    FORMAT 'csv' (ESCAPE '\')
    ```

  - 您也可以设置ESCAPE为OFF，避免所有字符被自动转义。
- 列中存在单引号或者双引号。
  - 如果您使用TEXT模式，需要在单引号或者双引号前面增加ESCAPE符，默认为反斜杠（\）。
  - 如果您使用CSV模式，需要在单引号或者双引号前面增加ESCAPE符，默认为双引号（”），同时在该列前后加双引号（”），将整列括起来。

COPY数据导入

您在使用\COPY语句导入数据时，分隔符的使用方法和OSS高速并行导入时的使用方法一样，而对数据中出现特殊符号的处理方法也和OSS高速并行导入相类似。不同的是COPY语句和`CREATE EXTERNAL TABLE`语句用法略有不同，COPY语句详细用法见使用COPY命令导入数据。

6.通过 HyperLoglog 实现高性能多维数据透视

本文结和电商类数据透视示例，介绍了使用AnalyticDB for PostgreSQL通过HLL预计算，实现毫秒级多维数据透视的方法。关于HyperLogLog的用法，请参考使用HLL。

实践总结

本文介绍的操作方法，涉及以下最佳实践。如您已了解操作方法，可以直接参考实践总结来应用。

- 对于透视分析需求，使用倒转的方法，将数据按查询需求进行预计算，得到统计结果；从而在透视时仅需查询计算结果，可以做到任意维度透视，都可以在100毫秒以内响应。
- 使用GROUPING SETS，对多个标签维度进行一次性统计，降低数据重复扫描和重复运算，大幅提升处理效率。
- 使用数组，记录每个透视维度的UID，不仅支持透视，也可以满足圈人的需求，同时支持未来更加复杂的透视需求。
- 使用HLL类型来存储估算值，在进行复杂透视时，可以使用HLL。例如，多个HLL的值可以UNION，可以求唯一值个数，通常用于评估UV，新增UV等。
- 使用流计算。如果数据需要实时的统计，那么可以使用pipelineDB进行流式分析，实时计算统计结果。
- 与阿里云云端组件结合，使用OSS对象存储过渡数据（原始数据）。使用OSS_FDW外部表对接OSS，因此过渡数据可以不入库，仅仅用于预计算。大幅降低数据库的写入需求、空间需求。
- 使用Greenplum的一级、二级分区，将透视数据的访问需求打散到更小的单位，然后使用标签索引，再次降低数据搜索的范围，从而做到任意数据量，任意维度透视请求100毫秒以内响应。
- 使用列存储，提升压缩比，节省统计数据的空间占用。

背景

典型的电商类数据透视业务会使用一些用户的标签数据作为透视的语料。例如，包含品牌的ID，销售区域的ID，品牌对应用户的ID，以及若干用户标签字段，时间字段等。在作分析时，标签可能会按不同的维度进行归类。例如，tag1 性别，tag2 年龄段, tag3 兴趣爱好等等。

业务方较多的需求往往是对自有品牌的用户进行透视。例如，一个非常典型的数据透视需求就是统计不同的销售区域（渠道）、时间段、标签维度下的用户数。

准备

作为示例，定义以下数据结构。

- t1：每天所在区域、销售渠道的活跃用户ID。

```
t1 (
uid,     -- 用户 ID
groupid,  -- 销售渠道、区域 ID
day      -- 日期
)
```

- t2：每个品牌的自有用户（维护增量）。

```
t2 (
uid,  -- 用户 ID
brand  -- 品牌
)
```

- t3：用户标签（维护增量）。

```
t3 (
uid,  -- 用户 ID
tag1,  -- 标签1，如兴趣
tag2,  -- 标签2，如性别
tag3,  -- 标签3，如年龄段
... ,
)
```

基于已定义的数据结构，可以按照品牌、销售区域、标签、日期进行透视。例如，

```
select
 '兴趣' as tag,
 t3.tag1 as tag_value,
 count(1) as cnt
from
 t1,
 t2,
 t3
where
 t1.uid = t3.uid
 and t1.uid = t2.uid
 and t2.brand = ?
 and t1.groupid = ?
 AND t1.day = '2017-06-25'
group by t3.tag1
```

可以看出，这类查询的运算量较大。而且，分析师可能需要对不同的维度进行比对分析。因此，建议采用预计算的方法进行优化。

使用预计算优化检索

为实现快速检索，您可以使用以下优化方法：

- 对于Greenplum，使用列存储。
- 表分区按照day范围一级分区，按brand, groupid哈希进行二级分区。
- 数据分布策略选择随机分布。
- 针对每个 `tag?` 字段建立单独索引。

结合以上优化，不管数据量多大，单次透视请求的响应速度都可以控制在100毫秒以内。

通过预计算优化，希望得到以下结果：

```
t_result (
 day,    -- 日期
 brand,   -- 品牌 ID
 groupid, -- 渠道、地区、门店 ID
 tag1,    -- 标签类型1
 tag2,    -- 标签类型2
 tag3,    -- 标签类型3
 ...      -- 标签类型n
 cnt,     -- 用户数
 uids,    -- 用户 ID 数组，这个为可选字段，如果不需要知道 ID 明细，则不需要保存
 hll_uids  -- 用户 HLL 估值
)
```

得到这份结果后，分析师的查询过程可以简化为以下内容：

```
select
 day, brand, groupid, 'tag?' as tag, cnt, uids, hll_uids
from t_result
where
 day =
 and brand =
 and groupid =
 and tag? = ?
```

其中，前三个条件（ `day, brand, groupid` ）通过分区过滤数据，最后根据 `tag?` 的索引快速得到结果。

可以看出，预计算后能够以少量的运算，实现更加复杂的维度分析。例如，可以分析出某两天的差异用户，多个TAG叠加的用户等。

使用预计算的方法

使用如下SQL来产生统计结果。

```
select
 t1.day,
 t2.brand,
 t1.groupid,
 t3.tag1,
 t3.tag2,
 t3.tag3,
 ...
 count(1) as cnt,
 array_agg(uid) as uids,
 ## 将 uid 聚合为数组。
 hll_add_agg(hll_hash_integer(uid)) as hll_uids
 ## 将 UID 转换为 hll hash val，并聚合为 HLL 类型。
from
 t1,
 t2,
 t3
where
 t1.uid = t3.uid
 and t1.uid = t2.uid
group by
 t1.day,
 t2.brand,
 t1.groupid,
 grouping sets (
 ## 为了按每个标签维度进行统计，使用多维分析语法 grouping sets，这样可以不必通过多条 SQL 来实现。结果是数据只扫描一遍，且按每个标签维度进行统计。
   (t3.tag1),
   (t3.tag2),
   (t3.tag3),
   (...),
   (t3.tagn)
 )
```

预计算结果透视查询

如果进行复杂透视，可以对分析结果的不同记录进行数组逻辑运算，得到UID集合结果。

使用数组逻辑运算

您可以使用以下数组逻辑运算。

- 统计在数组1但不在数组2的值。

```
create or replace function arr_miner(anyarray, anyarray) returns anyarray as $$
select array(select * from (select unnest($1) except select unnest($2)) t group by 1);
$$ language sql strict;
```

- 统计数组1和数组2的交集。

```
create or replace function arr_overlap(anyarray, anyarray) returns anyarray as $$
select array(select * from (select unnest($1) intersect select unnest($2)) t group by 1);
$$ language sql strict;
```

- 统计数组1和数组2的并集。

```
create or replace function arr_merge(anyarray, anyarray) returns anyarray as $$
select array(select unnest(array_cat($1,$2)) group by 1);
$$ language sql strict;
```

应用示例

例如，假设促销活动前（2017-06-24）的用户集合为UID1[]，促销活动后（2017-06-25）的用户集合为UID2[]，可以使用以下命令得出促销活动中有哪些新增用户。

```
arr_miner(uid2[], uid1[])
```

使用HLL做数据逻辑计算

您可以使用HLL进行以下逻辑计算。

- 计算唯一值个数。

```
hll_cardinality(users)
```

- 计算两个HLL的并集，得到一个HLL。

```
hll_union()
```

应用示例

例如，假设在促销活动前（2017-06-24）的用户集合HLL为uid1_hll，促销活动后（2017-06-25）的用户集合HLL为uid2_hll，可以使用以下命令得出促销活动中有哪些新增用户。

```
hll_cardinality(uid2_hll) - hll_cardinality(uid1_hll)
```

预计算调度

在优化前，业务通过即时JOIN得到透视结果，而优化后使用事先统计的方法来获得透视结果，而事先统计本身需要调度。调度方法取决于数据的来源以及数据合并的方法（流式增量或批量增量）。

按天统计数据

历史统计数据无更新，只有增量。需要定时将统计结果写入并合并至 `t_result` 结果表中。

```
insert into t_result
select
 t1.day,
 t2.brand,
 t1.groupid,
 t3.tag1,
 t3.tag2,
 t3.tag3,
 ...
 count(1) as cnt,
 array_agg(uid) as uids,
 hll_add_agg(hll_hash_integer(uid)) as hll_uids
from
 t1,
 t2,
 t3
where
 t1.uid = t3.uid
 and t1.uid = t2.uid
group by
 t1.day,
 t2.brand,
 t1.groupid,
 grouping sets (
  (t3.tag1),
  (t3.tag2),
  (t3.tag3),
  (...),
  (t3.tagn)
 )
```

合并统计维度数据

数据结果按天进行统计，但如果要查询按月，或者按年的统计，则需要对按天统计的数据查询并汇聚。业务也能选择异步汇聚，最终用户查询到的是汇聚后的结果。

```
t_result_month (
 month,   -- yyyy-mm
 brand,   -- 品牌 ID
 groupid, -- 渠道、地区、门店 ID
 tag1,    -- 标签类型1
 tag2,    -- 标签类型2
 tag3,    -- 标签类型3
 ...      -- 标签类型n
 cnt,     -- 用户数
 uids,    -- 用户 ID 数组，这个为可选字段，如果不需要知道 ID 明细，则不需要保存
 hll_uids  -- 用户 HLL 估值
)
```

array聚合需要自定义以下聚合函数：

```
postgres=# create aggregate arragg (anyarray) ( sfunc=arr_merge, stype=anyarray);
CREATE AGGREGATE
postgres=# select arragg(c1) from (values (array[1,2,3]),(array[2,5,6])) t (c1);
  arragg
-------------
 {6,3,2,1,5}
(1 row)
```

例如，您可以使用以下SQL，来按月汇聚数据：

```
select
 to_char(day, 'yyyy-mm'),
 brand,
 groupid,
 tag1,
 tag2,
 tag3,
 ...
 array_length(arragg(uid),1) as cnt,
 arragg(uid) as uids,
 hll_union_agg() as hll_uids
from t_result
group by
 to_char(day, 'yyyy-mm'),
 brand,
 groupid,
 tag1,
 tag2,
 tag3,
 ...
```

以此类推，可以得出按年汇聚的结果。

流式调度

如果业务方有实时统计的需求，那么可以使用流式计算的方法，实时进行以上聚合统计。如果数据量非常庞大，可以根据分区键，对数据进行分流，不同的数据落到不同的流计算节点，最后汇总流计算的结果到AnalyticDB for PostgreSQL(base on GPDB)中。

7.如何诊断和处理锁等待

在数据库中，通过锁机制以及多版本并发控制（MVCC）可以保护数据的一致性。例如，会话A正在查询数据，会话B就无法对会话A访问的对象执行DDL。会话A正在更新某条记录，会话B就不能删除或更新这条记录。

锁是由数据库自动控制的，如果应用程序或者SQL脚本设计不当，就可能导致长时间的锁等待或者死锁。AnalyticDB for PostgreSQL提供了两种统计视图，用户可通过这两个视图查询锁等待或者死锁的情况。

- pg_locks：用于展示锁信息，每个被锁的对象或等待锁的对象为一条记录。
- pg_stat_activity：显示所有会话的信息，每个会话为一条记录。

创建锁监控视图

具体如何查询当前的锁等待和持锁信息，您可以通过如下SQL语句创建锁监控视图。

> **说明** 本文的所有SQL命令均在psql客户端中执行，请使用psql连接数据库。

```
create view v_locks_monitor as
with
t_wait as
(
select a.mode,a.locktype,a.database,a.relation,a.page,a.tuple,a.classid,a.granted,
a.objid,a.objsubid,a.pid,a.transactionid,
b.xact_start,b.query_start,b.usename,b.datname,b.client_addr,b.client_port,b.application_name
from pg_locks a,pg_stat_activity b where a.pid=b.procpid and not a.granted
),
t_run as
(
select a.mode,a.locktype,a.database,a.relation,a.page,a.tuple,a.classid,a.granted,
a.objid,a.objsubid,a.pid,a.transactionid,
b.xact_start,b.query_start,b.usename,b.datname,b.client_addr,b.client_port,b.application_name
from pg_locks a,pg_stat_activity b where a.pid=b.procpid and a.granted
),
t_overlap as
(
select r.* from t_wait w join t_run r on
(
r.locktype is not distinct from w.locktype and
r.database is not distinct from w.database and
r.relation is not distinct from w.relation and
r.page is not distinct from w.page and
r.tuple is not distinct from w.tuple and
```

```
r.transactionid is not distinct from w.transactionid and
r.classid is not distinct from w.classid and
r.objid is not distinct from w.objid and
r.objsubid is not distinct from w.objsubid and
r.pid <> w.pid
)
),
t_unionall as
(
select r.* from t_overlap r
union all
select w.* from t_wait w
)
select locktype,datname,relation::regclass,page,tuple,transactionid::text,classid::regclass,objid,objsubid,
string_agg(
'Pid: '||case when pid is null then 'NULL' else pid::text end||chr(10)||
'Lock_Granted: '||case when granted is null then 'NULL' else granted::text end||' , Mode: '||case when mode is null then 'NULL' else mode::text end||' ,
Username: '||case when username is null then 'NULL' else username::text end||' , Database: '||case when datname is null then 'NULL' else datname::text end||' , Client_Addr: '||case when client_addr is null then 'NULL' else client_addr::text end||' , Client_Port: '||case when client_port is null then 'NULL' else client_port::text end||' , Application_Name: '||case when application_name is null then 'NULL' else application_name::text end||chr(10)||
'Xact_Start: '||case when xact_start is null then 'NULL' else xact_start::text end||' , Query_Start: '||case when query_start is null then 'NULL' else query_start::text end||' , Xact_Elapse: '||case when (now()-xact_start) is null then 'NULL' else (now()-xact_start)::text end||' ,
chr(10)||'--------'||chr(10)
order by
( case mode
when 'INVALID' then 0
when 'AccessShareLock' then 1
when 'RowShareLock' then 2
when 'RowExclusiveLock' then 3
when 'ShareUpdateExclusiveLock' then 4
when 'ShareLock' then 5
when 'ShareRowExclusiveLock' then 6
when 'ExclusiveLock' then 7
when 'AccessExclusiveLock' then 8
else 0
end ) desc,
```

```
(case when granted then 0 else 1 end)
) as lock_conflict
from t_unionall
group by
locktype,datname,relation,page,tuple,transactionid::text,classid,objid,objsubid;
```

查询锁信息

当发生锁等待或者死锁时，使用如下SQL语句查询v_locks_monitor的信息。

```
postgres=# \x

postgres=# select * from v_locks_monitor;
```

处理方法

前面SQL查询语句可以清晰地显示目前数据库系统发生的锁的情况，使用如下语句终止对应的进程。

```
postgres=# select pg_terminate_backend(PID);
```

其中PID为v_locks_monitor查询结果中Lock_Granted值为t的记录的Pid，如下图所示。

```
postgres=> \x
Expanded display is on.
postgres=> select * from v_locks_monitor;
-[ RECORD 1
]-+-----------------------------------------------------------------------------------------------
----------------------------------------
locktype      | relation
datname       | postgres
relation      | locktest
page          |
tuple         |
transactionid |
classid       |
objid         |
objsubid      |
lock_conflict | Pid: 3813
              | Lock_Granted: f , Mode: AccessExclusiveLock ,
              | Username: test123 , Database: postgres , Client_Addr:
              , Client_Port: 1712 , Application_Name: psql
              | Xact_Start: 2018-12-25 14:50:21.761477+08 , Query_Sta
              14:50:31.418015+08 , Xact_Elapse: 00:00:23.617841 ,
              | chr(10)||Pid: 52592
              | Lock_Granted: t , Mode: ExclusiveLock ,
              | Username: test123 , Database: postgres , Client_Addr:
              , Client_Port: 12149 , Application_Name: psql
              | Xact_Start: 2018-12-25 14:47:54.268166+08 , Query_Sta
              14:48:28.533436+08 , Xact_Elapse: 00:02:51.111152 ,
              | chr(10)||Pid: 52592
              | Lock_Granted: t , Mode: ExclusiveLock ,
              | Username: test123 , Database: postgres , Client_Addr:
              , Client_Port: 12149 , Application_Name: psql
              | Xact_Start: 2018-12-25 14:47:54.268166+08 , Query_Sta
              14:48:28.533436+08 , Xact_Elapse: 00:02:51.111152 ,
              | chr(10)||Pid: 6734
              | Lock_Granted: f , Mode: RowExclusiveLock ,
              | Username: test123 , Database: postgres , Client_Addr:
              , Client_Port: 20035 , Application_Name: psql
              | Xact_Start: 2018-12-25 14:49:09.0728+08 , Query_Start
              14:49:32.857467+08 , Xact_Elapse: 00:01:36.306518 ,
              | chr(10)||Pid: 52592
              | Lock_Granted: t , Mode: AccessShareLock ,
              | Username: test123 , Database: postgres , Client_Addr:
              , Client_Port: 12149 , Application_Name: psql
              | Xact_Start: 2018-12-25 14:47:54.268166+08 , Query_Sta
```

执行如下语句，如果返回结果为 `0 rows` 表示锁已经清除，如果查询结果还有记录，请再次执行 `select pg_terminate_backend(PID);` 终止对应的PID。

```
postgres=# select * from v_locks_monitor;
```

参考文献

PostgreSQL 锁等待监控珍藏级SQL - 谁堵塞了谁

8.如何使用连接池

AnalyticDB for PostgreSQL基于PostgreSQL内核构建，支持主流的连接池，包括pgbouncer，pgool-II。

pgbouncer：支持数据库连接池功能，pgbouncer

pgpool-II：功能很丰富，除支持数据库连接池，也支持负载均衡，具备任务自动重试，查询缓存等功能pgpool-II