ALIBABA CLOUD

# Alibaba Cloud

## AnalyticDB for PostgreSQL

## Best Practices

Document Version: 20200923

Alibaba Cloud

# Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please directly contact Alibaba Cloud for any errors of this document.

# Document conventions

| Style | Description | Example |
| --- | --- | --- |
| Danger | A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results. | Danger: Resetting will result in the loss of user configuration data. |
| Warning | A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results. | Warning: Restarting will cause business interruption. About 10 minutes are required to restart an instance. |
| Notice | A caution notice indicates warning information, supplementary instructions, and other content that the user must understand. | Notice: If the weight is set to 0, the server no longer receives new requests. |
| Note | A note indicates supplemental instructions, best practices, tips, and other content. | Note: You can use Ctrl + A to select all files. |
| > | Closing angle brackets are used to indicate a multi-level menu cascade. | Click **Settings**> **Network**> **Set network type**. |
| **Bold** | Bold formatting is used for buttons , menus, page names, and other UI elements. | Click **OK**. |
| `Courier font` | Courier font is used for commands | Run the `cd /d C:/window` command to enter the Windows system folder. |
| *Italic* | Italic formatting is used for parameters and variables. | `bae log list --instanceid` *Instance_ID* |
| [] or [a\|b] | This format is used for an optional value, where only one item can be selected. | `ipconfig [-all\|-t]` |
| {} or {a\|b} | This format is used for a required value, where only one item can be selected. | `switch {active\|stand}` |

# Table of Contents

# 1.Use pg_stat_activity to analyze and diagnose running PostgreSQL queries

pg_stat_activity is a helpful PostgreSQL system view. You can use pg_stat_activity to analyze and diagnose running PostgreSQL tasks and troubleshoot problems. The pg_stat_activity view shows a row for each server process or connection to the database.

The following table describes the fields of the pg_stat_activity view.

| Field | Type | Description |
| --- | --- | --- |
| datid | oid | The object identifier (OID) of the database to which the backend is connected. |
| datname | name | The name of the database to which the backend is connected. |
| procpid | integer | The ID of the backend process. Note that only AnalyticDB for PostgreSQL V4.3 supports this field. |
| pid | integer | The ID of the backend process. Note that only AnalyticDB for PostgreSQL V6.0 supports this field. |
| sess_id | integer | The session ID of the user who logs on to the backend. |
| usesysid | oid | The OID of the user who logs on to the backend. |
| usename | name | The name of the user who logs on to the backend. |
| current_query | text | Note that only AnalyticDB for PostgreSQL V4.3 supports this field.<br>The currently executing query in the backend. By default, the query text is truncated to 1,024 characters in length. You can specify the track_activity_query_size parameter to change the maximum number of characters. |
| query | text | Note that only AnalyticDB for PostgreSQL V6.0 supports this field.<br>The last executed query in the backend. If the state parameter is set to active, the currently executing query is displayed in the backend. If the state parameter is set to a value other than active, the last executed query is displayed in the backend. By default, the query text is truncated to 1,024 characters in length. You can specify the track_activity_query_size parameter to change the maximum number of characters. |
| waiting | boolean | Indicates whether the current SQL query is waiting on a lock. Valid values: True and False. |
| query_start | | The time when the currently active query was started. If the state field is not set to active, the query_start field indicates the time when the last query was started. |
| backend_start | | The time when the current backend process was started. |

| Field | Type | Description |
| --- | --- | --- |
| client_addr | inet | The IP address of the client that is connected to this backend. If this field is empty, the client is connected through a Unix socket on the server or this is an internal process such as autovacuum. |
| client_port | integer | The TCP port number that the client uses for communication with the backend server. A value of -1 indicates that a Unix socket is used. |
| application_name | text | The name of the application that is connected to this backend. |
| xact_start | | The time when the current transaction of this process was started, or null if no transaction is active. If the current query is the first transaction of the process, this field is equivalent to the query_start field. |
| waiting_reason | text | The reason why the backend is currently waiting. The backend is currently waiting on a lock or on replication of data among nodes. |
| state | text | Note that only AnalyticDB for PostgreSQL V6.0 supports this field.<br>The current state of the backend. Valid values: active, idle, idle in transaction, idle in transaction (aborted), fastpath function call, and disabled. |
| state_change | timestampz | Note that only AnalyticDB for PostgreSQL V6.0 supports this field.<br>The time when the previous state was changed. |

## View connection information

You can execute the following SQL statements to view usernames and the corresponding clients:

```
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)
```

## Query the SQL execution information

You can execute the following SQL statements to view current queries executed by users who connect to the database:

```
 AnalyticDB for PostgreSQL V4.3:
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)

AnalyticDB for PostgreSQL V6.0:
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)
```

You can execute the following SQL statements to view only active queries:

```
AnalyticDB for PostgreSQL V4.3:
SELECT datname,usename,current_query
  FROM pg_stat_activity
  WHERE current_query ! = '<IDLE>' ;

AnalyticDB for PostgreSQL V6.0:
SELECT datname,usename,query
  FROM pg_stat_activity
  WHERE state ! = 'idle' ;
```

## View long-running queries

You can execute the following SQL statements to view long-running queries:

```
AnalyticDB for PostgreSQL V4.3:
select current_timestamp - query_start as runtime, datname, usename, current_query
   from pg_stat_activity
   where current_query ! = '<IDLE>'
   order by 1 desc;

AnalyticDB for PostgreSQL V6.0:
select current_timestamp - query_start as runtime, datname, usename, query
   from pg_stat_activity
   where state ! = 'idle'
   order by 1 desc;
```

Sample responses:

```
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)
```

The first returned query runs for a long time. The first query has been running for 34 seconds and continues to run.

## Diagnose and troubleshoot abnormal SQL queries

If an SQL query has been running for an extended period of time without returning any results, you must check whether the query is running or blocked.

```
AnalyticDB for PostgreSQL V4.3:
SELECT datname,usename,current_query
  FROM pg_stat_activity
  WHERE waiting;

AnalyticDB for PostgreSQL V6.0:
SELECT datname,usename,query
  FROM pg_stat_activity
  WHERE waiting;
```

Note that the output shows only SQL queries that are blocked due to lock waits. In most cases, SQL queries are blocked due to lock waits. However, in some other cases, SQL queries are blocked because they are waiting for I/O operations or timers. If the preceding SQL statements have returned responses, it indicates that specific SQL queries are blocked due to lock waits. You can execute the following statements to view details about these SQL queries:

```
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;
```

The output shows blocked SQL queries and their corresponding process IDs. You can review the details of the blocked SQL queries and then cancel or kill the queries to remove the blocks. You can execute the following statement to cancel a running query:

```
SELECT pg_cancel_backend(pid)
```

The pg_cancel_backend function will take effect only on a session with a running query. The function will not do anything when the session is idle. In addition, canceling the query may take some time depending on the cleanup or rollback of the transactions. pg_terminate_backend can be used to terminate running queries or idle sessions.

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

Connections initiated by the specified user are terminated. We recommend that you do not run the pg_terminate_backend function on processes identified by pid where active queries exist. You must have superuser permissions to perform the operations in this topic.

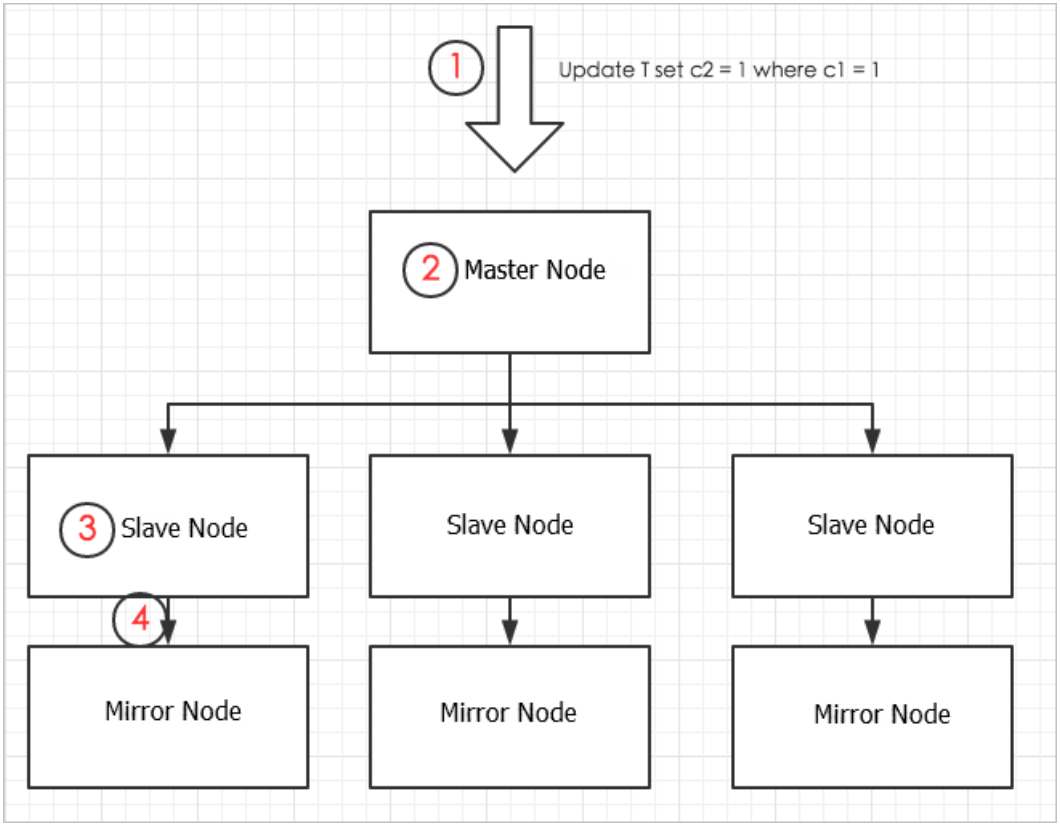
# 2.Bulk update

Update, also called Merge, indicates updating the latest data to AnalyticDB for PostgreSQL. If the updated data already exists, it replaces the old version. If the updated data does not exist, it is inserted to the database. Such data merge is usually completed offline. For example, you can set to update data on a daily basis to AnalyticDB for PostgreSQL. Some users may require real-time updates, that is, the latency is at the minute or second level.

This document describes how to merge data in AnalyticDB for PostgreSQL and explains the principle behind it. In addition, you can learn how to use the bulk operation to update multiple data.

## Simple update

Data merge is about modifying the data, that is, running the Update, Delete, Insert, or Copy operations. Take an Update operation for example, updating the record on a single row in a column-store table. The following figure shows the data updating process in AnalyticDB for PostgreSQL.




The procedure is described as follows:

1. The user sends an Update SQL request to the master node.
2. The master node initiates distributed transactions, locking the table to be updated (AnalyticDB for PostgreSQL does not allow concurrent updates to the same table), and distributing updating requests to matched slave nodes.

3. Slave nodes scan the index to locate the data to update, and update the data. For column-store tables, the updating logic is to delete the old data row and write the new data row at the end of the table. The updated data page in the column-store table is written to the memory cache, and the change in the corresponding table file length (because data is written to the table end, the length of the corresponding table file is increased) is written to the log (xlog file).
4. Before the Update process ends, the updated data page and xlog file in the memory are both synchronized to the mirror node. After the synchronization is complete, the master node ends the distributed transaction, and returns the message about successful execution to the user.

The whole process is long and contains lots of operations, such as SQL statement parsing, transactions distributing, locking, connection establishment between the master node and slave nodes, and the synchronization of data and log between slave nodes and the mirror node. These operations all consume CPU or I/O resources and prolong the response of the request.

Therefore, for AnalyticDB for PostgreSQL, we recommend that you avoid updates to a single data row, and try to update data by using bulk operations as much as possible. That is:

- Put updates in one SQL statement to reduce the overhead for statement parsing, node communications, and data synchronization.
- Put updates in one transaction to avoid unnecessary overhead.

## Bulk update

Follow these steps to use one SQL statement to update multiple independent data rows.

**1. Prepare the target table**

Suppose that the table to be updated is target_table. The target_table is defined as follows.

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

The target table is usually quite big. Suppose that you want to insert 10 million rows of data to target_table. The target_table is indexed to facilitate updates. A primary key is defined and a unique index is included consequently.

**2. Prepare the stage table**

The stage table (source_table in this example) is necessary for bulk update. It is a temporary table created for updating data. To update the data in target_table, you first insert the new data to source_table, and then import the new data by using the Copy command, OSS external table, or other means to target_table.

In the following example, some data is directly generated in source_table.

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

**3. Bulk update**

After the source_table data is ready, run the `update set ... from ... where ..` statement.

> **Note** To utilize the index to a maximum extent, you can use `set optimizer=on` to start the ORCA optimizer before the update operation. If the ORCA optimizer is not started, you can run `set enable_nestloop = on` to use the index.

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

The update operation's query plan is as follows:

```
=> 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
```

From the plan, AnalyticDB for PostgreSQL uses the index. But if you add more data to source_table, the optimizer may deem that using Nest Loop associated method and index scanning is not as efficient as dropping the index. As a result, it may use Hash associated method and table scanning for the execution. For example,

```
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)
```

The bulk update approach described reduces SQL compilation, inter-node communications, transactions, and other overheads, and can greatly boost data updating performance and reduce resource consumption.

## Bulk delete

For delete operations, you can use a stage table similar to that used for bulk update, and use the following delete command with a “Using” clause to delete data by bulk:

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

The bulk delete operation also uses the index.

```
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
```

## Merge data by using Delete and Insert

To merge data, you must first put the data to merge to the stage table.

- If you know in advance that the data to be merged already exists in the target table, you can use update statements to merge the data.
- But in most cases, part of the data to be merged already exists in the target table, and part of it is new, with no matched records in the target table. In this case, you can use a combination of bulk delete and bulk insert. The sample code is as follows.

```
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;
```

## Update data in real time by using Values() expressions

To use the stage table, you must maintain its lifecycle. Some users want to update data to AnalyticDB for PostgreSQL by bulk in real time, that is, to continuously synchronize data or merge data to AnalyticDB for PostgreSQL.

If you use the aforementioned method, you must create and delete (or truncate) the stage table repeatedly. In fact, you can use Values expressions to achieve an effect similar to stage tables, without the effort to maintain the table. The approach is to first splice the data to update into a Values expression, and then run the update or delete commands by using the following method:

```
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
```

**Note** Both `set optimizer=on;` and `set enable_nestloop=on;` generate query plans that use indexes. In complicated cases, however, such as multiple index fields or partition tables are involved, the ORCA optimizer must be used to match the index.

# 3.Optimize query performance

This topic describes how to optimize the query performance of an AnalyticDB for PostgreSQL instance in various scenarios.



## Collect table statistics

The query optimizer of AnalyticDB for PostgreSQL optimizes the plan of a query and estimates its costs based on the statistics of the queried tables. If no statistics are collected from the queried tables or the collected table statistics are outdated, the query optimizer optimizes the query plan based on the default or stale values. As a result, the query optimizer cannot generate an optimal query plan. We recommend that you collect statistics on a table after a large volume of data is loaded or more than 20% of the table data is updated.

The `ANALYZE` statement allows you to collect statistics on all tables, all columns of a table, or specific columns of a table. In most cases, we recommend that you collect statistics on all tables or all columns of a table. However, if you want to have more control over the collection of table statistics, you can choose to only collect statistics of the columns on which join keys, filter conditions, or indexes are created.

**Examples:**

- After a large volume of data is imported, execute the following statement to collects statistics on all tables:

```
ANALYZE;
```

- After a large volume of data is inserted, updated, or deleted in the t table, execute the following statement to collect statistics on all columns of that table:

```
ANALYZE t;
```

- Execute the following statement to collect statistics on the a column of the t table:

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

## Choose a query optimizer

AnalyticDB for PostgreSQL provides two query optimizers: Legacy and ORCA. Each query optimizer has its strengths and weaknesses in various scenarios.

- **Legacy query optimizer**

This is the default query optimizer. The Legacy query optimizer takes a short time to optimize an SQL statement. It is ideal for highly concurrent simple queries that require joins of no more than three tables and for highly concurrent data writes or updates that are executed by using INSERT, UPDATE, or DELETE statements.

- **ORCA query optimizer**

  The ORCA query optimizer is designed to optimize complex queries. It traverses more execution paths, and therefore it takes a longer time than the Legacy query optimizer to generate an optimal plan for each query. We recommend that you choose the ORCA query optimizer for complex queries that require joins of more than three tables to complete extract, transform, load (ETL) and report workloads. In addition, the ORCA query optimizer removes the need to join tables in subqueries and dynamically filters partitions. Therefore, we recommend that you choose the ORCA query optimizer to optimize SQL statements that have subqueries and those used to query data from partitioned tables on which parameter-specified filter conditions are created.

The following example shows how to configure both the Legacy and ORCA optimizers for a session:

```
-- Enable the Legacy query optimizer.
set optimizer = off;
-- Enable the ORCA query optimizer.
set optimizer = on;
```

You can execute the following statement to view the current optimizer:

```
show optimizer;
A value of on indicates that the ORCA query optimizer is used.
A value of off indicates that the Legacy optimizer is used.
```

> **Note**
>
> - By default, AnalyticDB for PostgreSQL V4.3 uses the Legacy optimizer. AnalyticDB for PostgreSQL V6.0 uses the ORCA optimizer.
> - To configure the Legacy and ORCA optimizers for an instance, you must submit a ticket.

## Use indexes to accelerate queries

If a query contains a filter condition used to identify identical values or values within a specific range and only a small volume of data is obtained, you can create an index on the column used as the filter criterion to expedite data scanning. AnalyticDB for PostgreSQL supports the following three types of indexes:

- B-tree indexes: If a column has a large number of unique values and is used to filter, join, or sort data, create a B-tree index.
- Bitmap indexes: If a column has a small number of unique values and more than one filter condition is created on it, create a bitmap index.

- GiST indexes: If you want to query geographic locations, ranges, image features, or Geometry values, create a GiST index.

**Examples:**

If you execute the following statement to query data from a table without an index, the system scans all data of the table and then filters the data based on the filter conditions specified in the query:

```
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)
```

Execute the following statement to create a B-tree index on the b column of the t table:

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

If you execute the following statement to query data from a table with an index, the system obtains data based on the index:

```
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)
```

## View query plans

A query plan is a set of steps that AnalyticDB for PostgreSQL executes to complete the query. It is equivalent to an algorithm. You can analyze query execution processes based on query plans to find out why SQL statements are executed slowly. If you add the keyword EXPLAIN to a query, the system only displays the query plan but does not execute the specified SQL statement. If you add the keyword EXPLAIN ANALYZE to a query, the system executes the specified SQL statement, collects the query execution information, and then displays the information in the query plan.

- The following example shows a query plan with the keyword EXPLAIN added to the query:

```
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)
```

- The following example shows a query plan with the keyword EXPLAIN ANALYZE added to the query:

```
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)
```

A query plan is composed of operators and organizes their information to process data in a logical order.

AnalyticDB for PostgreSQL supports the following types of operators:

- Data scanning operators: Seq Scan, Table Scan, Index Scan, and Bitmap Scan.
- Join operators: Hash Join, Nested Loop, and Merge Join.
- Aggregate operators: Hash Aggregate and Group Aggregate.
- Distribute operators: Redistribute Motion, Broadcast Motion, and Gather Motion.
- Other operators: Hash, Sort, Limit, and Append.

```
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)
```

The preceding query plan is described as follows:

1. The Table Scan operator scans the t1 and t2 tables.
2. The Redistribute Motion operator redistributes the data of the t1 and t2 tables based on the hash values of their b columns to compute nodes.
3. The Hash operator creates a hash key used for joins on the t2 table.
4. The Hash Join operator joins the data of the t1 and t2 tables.
5. The Gather Motion operator transmits the computing result to the coordinator node. The coordinator node then transmits the computing result to the client.

The actual query plan varies based on the SQL statement you specify.

## Remove distribute operators to increase query performance

When you call a join or aggregate operator, AnalyticDB for PostgreSQL adds a distribute operator based on the data distribution to redistribute (Redistribute Motion) or broadcast (Broadcast Motion) data. Distribute operators occupy a large amount of network resources. To increase query performance, we recommend that you create tables and adjust the business logic to remove the needs of distribute operators.

**How it works**

If the distribution keys of the two tables that you want to join do not match the business logic, you can change their distribution keys to remove the need of distribute operators.

**Example:**

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

In this example, the distribution key of the t1 table is the a column.

- If the distribution key of the t2 table is the b column, AnalyticDB for PostgreSQL redistributes the data of the t2 table:

```
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)
```

- If the distribution key of the t2 table is also the a column, AnalyticDB for PostgreSQL joins the t1 and t2 tables without redistributing the data of the t2 table:

```
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)
```

## Optimize the data types of the columns to be joined

Columns you want to join must have the same data type to prevent the explicit or implicit conversion of data types, because data type conversion causes data redistribution.

- **Explicit type conversion**

  In SQL statements, the data types of the columns you want to join may be forcibly converted. This is called an explicit type conversion. For example, the a column of the t table uses the int data type, but it is converted to the numeric data type by a join.

After the data type of a column is explicitly converted, the hash functions or values of the data in that column change. Therefore, we recommend that you avoid data type conversion on columns you want to join.

As shown in the following examples, an explicit type conversion triggers data redistribution:

```
-- Execute a join without a data type conversion.
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)

-- Execute a join with an explicit type conversion.
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)
```

- **Implicit type conversion**

  If you want to join more than one column of two tables but one of the columns uses a distinct data type in each table, the data type in that column needs to be converted. This is called an implicit type conversion.

After the data type of a column is implicitly converted, the hash functions or values in the original data type may differ from those in the new data type. As a result, AnalyticDB for PostgreSQL redistributes the data on that column. Therefore, we recommend that you choose columns of the same data type as the join keys.

In the following example, the a column of the t1 table uses the "timestamp without time zone" data type and the a column of the t2 table uses the "timestamp with time zone" data type. This means that the two a columns use different hash functions. As a result, AnalyticDB for PostgreSQL redistributes their data before joining them.

```
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)
```

## Locate data skew

If your query is abnormally slow or resource usage is uneven, check whether data skew has occurred.

Specifically, check the number of rows distributed to each compute node. If the rows of a table are unevenly distributed among compute nodes, the data of that table is skewed.

```
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)
```

If the data of a table is skewed, we recommend that you use one of the following methods to define a new distribution key for that table:

- Re-create the table and specify a new distribution key.

- Execute the `ALTER TABLE t1 SET DISTRIBUTED BY (b);` statement to change the distribution key.

## View running SQL statements

If a large number of SQL statements are executed concurrently, the concurrent queries are slow and your AnalyticDB for PostgreSQL instance may report insufficient resources.

You can obtain the status of your AnalyticDB for PostgreSQL instance from the pg_stat_activity view. This view lists all concurrent SQL statements. You can determine whether a query took an abnormally long time based on the query_start field in this view.

Example:

```
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)
```

The following section describes crucial fields in the preceding example:

- procpid: the ID of the master process that executed the query.
- usename: the username of the user who executed the query.
- current_query: the query text.
- waiting: indicates whether the query was in the waiting state.
- query_start: the start time of the query.
- backend_start: the time when the process used to execute the query started.
- xact_start: the time when the transaction to which the query belongs started.
- waiting_reason: the reason why the query remained in the waiting state.

In addition, you can add the `current_query ! = '<IDLE>'` condition to the `current_query` field to view the SQL statements that are being executed.

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

Execute the following statement to view the top five SQL statements that took the longest time to execute:

```
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;
```

## Check the status of locks

If an object in your AnalyticDB for PostgreSQL instance remains locked by a query for a long time, the other queries that involve that object may remain in the waiting state and cannot be executed properly. Execute the following statement to view the tables that are locked in your AnalyticDB for PostgreSQL instance:

```
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;
```

If a query does not respond because it is waiting for the lock on a table to be released, you can check the lock on that table. Use one of the following methods to resolve this issue if necessary:

- Cancel this query. If the session indicated by the pid parameter is idle, this method is unsuitable. In addition, you must delete data and roll back the transaction to which the query belongs after you cancel the query.

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

  > **Note** The `pg_cancel_backend` function does not take effect on the session for which the value of the `pg_stat_activity.current_query` parameter is IDLE. In this situation, you can call the `pg_terminate_backend` function to delete data.

- Terminate the session to which the query belongs. After the session is terminated, the

uncommitted transactions in it are rolled back.

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

## Join tables by using nested loops to increase query performance

By default, nested loop joins are disabled in AnalyticDB for PostgreSQL. If only a small volume of data is returned by your query, the query performance may not be optimal.

Example:

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

In the preceding example, the t1 and t2 tables are large. However, the filter condition `(t1.c2 >= '230769548' and t1.c2 < '23432442')` on the t1 table filters out most of the data records and the LIMIT clause further limits qualified data records. As a result, only a small volume of data is queried. In this situation, you can use nested loops to join the two tables.

To join tables by using nested loops, you must execute the SET statement. Example:

```
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)
```

In the preceding example, the t1 and t2 tables are joined by using nested loops to optimize query performance.

# 4.Configure scheduled maintenance tasks

When you perform system update operations such as INSERT VALUES, UPDATE, DELETE, and ALTER TABLE ADD COLUMN, junk data is retained in the system table and updated data table. The junk data diminishes the system performance and consumes a large amount of disk space. Therefore, we recommend that you clear such data on a regular basis. This topic describes how to clear junk data for AnalyticDB for PostgreSQL.

## Clear junk data without locking tables

You can clear some junk data without locking their corresponding tables. The method is as follows:

- Log on to each database as the database owner, and execute the `VACUUM` statement.
- The VACUUM statement must be executed at least once a day.
  - When you perform real-time update operations, such as INSERT VALUES, UPDATE, and DELETE, we recommend that you execute the VACUUM statement once every two hours.
  - If you only perform batch updates once a day, we recommend that you execute the VACUUM statement once after the update.
- The system does not lock tables on which the VACUUM statement is being executed. You can read data from the tables or write data into them when the VACUUM statement is executed. However, this operation increases CPU utilization and I/O usage, and may affect query performance.
- You can run the following Linux Shell script file as a scheduled crontab task:

```
#! /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
```

## Clear junk data during a maintenance window

You can execute the VACUUM FULL statement to clear all junk data in all tables during a maintenance window. The method is as follows:

- Log on to each database as the database owner. The database owner must have full permissions.
  i. Execute the `REINDEX SYSTEM <database name>` statement.
  ii. Execute the `VACUUM FULL <table name>` and `REINDEX TABLE <table name>` statements on each data table.
  iii. If you create and delete system tables and indexes at a high frequency, we recommend that you execute the VACUUM FULL <table name> statement to maintain the tables on a regular basis. The system tables include pg_class, pg_attribute, and pg_index. Note: We recommend that you do not access the database when you execute the VACUUM FULL <table name> statement.
- The VACUUM FULL statement must be executed at least once a week. If the majority of your data is updated every day, execute the VACUUM FULL statement once a day.
- The system locks tables on which the VACUUM FULL or REINDEX statement is being executed and these tables cannot be read or written during the execution. This operation increases CPU utilization and I/O usage.
- You can run the following Linux Shell script file as a scheduled crontab task:

```
#! /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.Pre-process special characters within files to be imported

AnalyticDB for PostgreSQL provides multiple methods to import data. For more information, see Introduction to data migration and synchronization solutions.

Special characters can cause import errors when you import data in parallel by using OSS or when you import data by using the \COPY command. This topic describes how to pre-process special characters in data files to be imported to avoid errors caused by these special characters.

## Method to pre-process special characters when you import data in parallel by using OSS

When you import data in parallel by using OSS, each row is processed as a tuple. You must specify a delimiter in each row to separate data in each column. This section describes how to specify delimiters when you create an OSS external table, the constraints of delimiters, and how to pre-process different types of special characters within columns when you import data in parallel by using OSS.

## Delimiters

In the syntax to create an OSS external table, you can specify DELIMITER after the FORMAT clause, as shown in the following example:

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

- For `FORMAT 'TEXT'`, `DELIMITER` is set to `'\t'` by default.
- For `FORMAT 'CSV'`, `DELIMITER` is set to `','` by default.

When you create an OSS external table, you can also define a custom delimiter that meets the following constraints:

- A delimiter must be a single ASCII character. It cannot be a string of two or more ASCII characters.
- `'\n'` and `'\r'` are not supported.
- Escape characters other than `'\n'` and `'\r'` are supported but must be preceded with "E" or "e" when they are used.
- The escape character `'\t'` is supported if it is not preceded by "E".
- For TEXT files, you can set DELIMITER to OFF and create single-column external tables.

Data in your OSS files must be separated by specified delimiters to ensure that the data can be read.

## Special characters in data

The following section describes scenarios where special characters exist in data to be imported and corresponding pre-processing methods.

- A column contains the same character as a delimiter.

  - You must precede each delimiter with an escape character in TEXT files. You can execute the following statement to specify the escape character. The default escape character is a backslash (\).

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

  - You must precede each delimiter with a double quotation mark (") in CSV files.

- A column contains Chinese characters. OSS external tables support Chinese characters. However, to ensure that data is correctly displayed, you must set the encoding format to UTF-8 when you create an OSS external table.

  ```
  ENCODING 'UTF8'
  ```

- A column contains null data. You can define a matching character for null, so that the specified character is identified as null during data import. For CSV files, the default value is a null value with no quotation marks. For TEXT files, the default value is \N. The following statement maps space to null. If the column is a space, then the value for the column is null in the data imported from the OSS file.

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

- A column contains escape characters. You must precede escape characters with the "ESCAPE" word. You can specify the escape character when you create an external table. The default escape character is a double quotation mark (") in CSV files and a backslash (\) in TEXT files.
  - You can set ESCAPE to a single character. You can use the following statement to set ESCAPE to a backslash (\):

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

  - You can also set ESCAPE to OFF to prevent automatic escape for all characters.

- A column contains single quotation marks or double quotation marks.
  - You must precede a single quotation mark or double quotation mark with the "ESCAPE" word in TEXT files. The default escape character is a backslash (\).
  - You must precede a single quotation mark or double quotation mark with the "ESCAPE" word in CSV files. The default escape character is a double quote ("). You must also enclose the whole column in beginning and ending double quotation marks.

## Method to pre-process special characters when you import data by using the \COPY command

When you use the \COPY command to import data, you can specify delimiters and pre-process special characters in the same way as when you import data in parallel by using OSS. However, the \COPY command and the `CREATE EXTERNAL TABLE` statement are used in slightly different ways. For more information about how to use the \COPY command, see Use the \COPY command.

# 6.Use HyperLogLog for high-performance multi-dimensional data view

This topic uses e-commerce as an example to describe how to use HyperLogLog to precompute data in AnalyticDB for PostgreSQL. Based on precomputation results, you can view data in multiple dimensions within milliseconds . For information about how to use HyperLogLog, see Use HyperLogLog.

## Best practices

The methods provided in this topic are used in the following best practices. If you are familiar with the methods, you can directly apply them to your business with reference to the best practices.

- AnalyticDB for PostgreSQL precomputes data based on query requests. With the precomputation results, you only need to specify the filter criteria for a data view query and AnalyticDB for PostgreSQL responds to your query for data in all dimensions within 100 milliseconds.
- AnalyticDB for PostgreSQL executes a GROUPING SETS statement to collect data statistics in multiple dimensions simultaneously. This reduces repetitive data scanning and computing workloads and increases processing efficiency.
- AnalyticDB for PostgreSQL records user IDs in each dimension as an array. This enables you to not only view data but also spot specific users.
- AnalyticDB for PostgreSQL uses hll fields to store estimated data statistics for complex views. For example, the values of multiple hll fields can be united to analyze unique visitors (UVs) and identify new UVs.
- AnalyticDB for PostgreSQL implements stream processing. It uses PipelineDB to compute data and obtain statistics in real time.
- AnalyticDB for PostgreSQL stores intermediate data (also referred to as raw data) in the OSS_FDW external table and uploads the table to Alibaba Cloud Object Storage Service (OSS). This way, AnalyticDB for PostgreSQL does not need to store intermediate data but only uses it for precomputation. This reduces the volume of data written into AnalyticDB for PostgreSQL and the storage space occupied.
- AnalyticDB for PostgreSQL uses level-1 and level-2 partitions of Greenplum to break a data view query into smaller units. It uses indexed tags to further narrow down the scope of searches. By doing so, AnalyticDB for PostgreSQL responds to your query for data in any dimension and among any volume within 100 milliseconds.
- AnalyticDB for PostgreSQL uses column-oriented storage to increase the data compression ratio and the storage space occupied by data statistics.

## Background information

Typical e-commerce data view businesses use tags such as brand IDs, sales territory IDs, user IDs, and timestamps to compute and analyze data. These tags may be classified in various dimensions. For example, tag 1 is classified into a gender category, tag 2 is classified into an age category, and tag 3 is classified into a hobby category.

A customer is likely to view the user base of their own brand. A typical example is to view the number of users in each sales territory (or channel), in each time range, and in each dimension.

## Preparations

The example in this section is for reference only. Make preparations based on your actual business needs. For this example, define the following data structures:

- t1: the IDs of active users in a sales territory (or channel) on a specific day.

```
t1 (
uid,      -- The ID of an active user.
groupid,  -- The ID of the sales territory (or channel).
day       -- The date.
)
```

- t2: the user base of a brand.

```
t2 (
uid,    -- The ID of a user.
brand   -- The ID of the brand.
)
```

- t3: the tags of a user.

```
t3 (
uid,    -- The ID of the user.
tag1,   -- Tag 1 bound to the user, for example, a hobby tag.
tag2,   -- Tag 2 bound to the user, for example, a gender tag.
tag3,   -- Tag 3 bound to the user, for example, an age tag.
... ,
)
```

Based on the defined data structures, you can view data in brand, sales territory (or channel), tag, and date dimensions. Example:

```
select
  'hobby' 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
```

This type of query requires heavy computing workloads. In addition, you may need to compare and analyze data in different dimensions. Therefore, we recommend that you use precomputation to optimize data retrieval.

## Data retrieval optimization based on precomputation

Use the following methods to optimize data retrieval and expedite queries:

- Use column-oriented storage if your instance runs Greenplum.
- Partition each table by day and then further partition it by brand and group ID.
- Choose random distribution for tables.
- Create an independent index for each `tag?` field.

After the preceding optimization, AnalyticDB for PostgreSQL can respond to a data view query within 100 milliseconds no matter how large the data volume.

The following results are expected after precomputation:

```
t_result (
  day,      -- The date.
  brand,    -- The ID of the brand.
  groupid,  -- The ID of the sales territory (channel) or shop.
  tag1,     -- Tag 1 bound to users.
  tag2,     -- Tag 2 bound to users.
  tag3,     -- Tag 3 bound to users.
  ...       -- Tag n bound to users.
  cnt,      -- The number of users.
  uids,     -- The user IDs in an array. This field is optional. If you do not want user IDs, you can choose n
ot to save them.
  hll_uids  -- The user IDs estimated by HyperLogLog.
)
```

With the precomputation results, you can simplify your query as follows:

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

In the preceding example, AnalyticDB for PostgreSQL filters data in partitions based on the first three filter criteria, `day, brand, and groupid`, and then obtains the expected data based on the `tag?` field.

Precomputation enables AnalyticdB for PostgreSQL to analyze data in more complex dimensions with fewer computing workloads. For example, AnalyticDB for PostgreSQL can identify differentiated users within two days and users to whom more than one tag is bound.

## Precomputation methods

Execute the following SQL statements to precompute data:

```
select
 t1.day,
 t2.brand,
 t1.groupid,
 t3.tag1,
 t3.tag2,
 t3.tag3,
 ...
 count(1) as cnt,
 array_agg(uid) as uids,
 ## Aggregate user IDs into an array.
 hll_add_agg(hll_hash_integer(uid)) as hll_uids
 ## Convert user IDs into hash values of the HLL data type.
from
 t1,
 t2,
 t3
where
 t1.uid = t3.uid
 and t1.uid = t2.uid
group by
 t1.day,
 t2.brand,
 t1.groupid,
 grouping sets (
 ## Execute one GROUPING SETS statement to precompute data in multiple dimensions. This way, you do not need to execute multiple SQL statements, each of which precomputes data from one dimension. After you execute a GROUPING SETS statement, AnalyticDB for PostgreSQL scans data only once and then generates statistics based on each specified tag.
   (t3.tag1),
   (t3.tag2),
   (t3.tag3),
   (...),
   (t3.tagn)
 )
```

## Data view queries based on precomputation results

For complex queries, AnalyticDB for PostgreSQL performs array-based logical operations to obtain a set of user IDs.

**Supported array-based logical operations**

AnalyticDB for PostgreSQL supports the following array-based logical operations:

- Obtain the values that are in array 1 but are not in array 2.

```
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;
```

- Obtain the values that are in both arrays 1 and 2.

```
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;
```

- Obtain the values that are in array 1 and those in array 2.

```
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;
```

**Examples**

The user ID set as of June 24, 2017 is UID1[], and the user ID set since June 25, 2017 when a promotion started is UID2[]. You can run the following command to identify new users attracted by the promotion:

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

## Value-based logical operations by using HyperLogLog

AnalyticDB for PostgreSQL supports the following value-based logical operations:

- Obtain the number of unique values:

```
hll_cardinality(users)
```

- Unite the values of two hll fields to obtain a unique value:

```
hll_union()
```

**Examples**

The user ID set HLL as of June 24, 2017 is uid1_hll, and the user ID set HLL since June 25, 2017 when a promotion started is uid2_hll. You can run the following command to identify new users attracted by the promotion:

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

## Precomputation scheduling

Before precomputation is introduced, AnalyticDB for PostgreSQL returns query results by using joins. After precomputation is introduced, AnalyticDB for PostgreSQL precomputes data to generate statistics. Each precomputation must be scheduled based on the data source and data aggregation method. AnalyticDB for PostgreSQL supports two data aggregation methods: stream computing and batch computing.

**Day-level precomputation**

Historical statistics are not updated and some new statistics are added. New statistics must be written into the `t_result` table in real time.

```
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)
 )
```

**Statistics aggregation**

If data is precomputed at the day level but you want to query data of a specific month or year, AnalyticDB for PostgreSQL aggregates precomputed day-level statistics to respond to your query. AnalyticDB for PostgreSQL supports asynchronous aggregation. The data view returned to you is the aggregation results generated by AnalyticDB for PostgreSQL.

```
t_result_month (
  month,    -- yyyy-mm
  brand,    -- The ID of the brand.
  groupid,  -- The ID of the sales territory (channel) or shop.
  tag1,     -- Tag 1 bound to users.
  tag2,     -- Tag 2 bound to users.
  tag3,     -- Tag 3 bound to users.
  ...       -- Tag n bound to users.
  cnt,      -- The number of users.
  uids,     -- The user IDs in an array. This field is optional. If you do not want user IDs, you can choose n
ot to save them.
  hll_uids  -- The user IDs estimated by HyperLogLog.
)
```

To aggregate data from multiple arrays, define the following function:

```
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)
```

For example, execute the following SQL statements to aggregate data at the month level:

```
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,
 ...
```

Based on the preceding example, you can obtain statistics aggregated at the year level.

## Stream computing

Stream computing is used to aggregate statistics in real time. If the data volume is large, you can use the partition key to distribute data to different stream compute nodes. After the stream compute nodes complete processing, they send processing results to AnalyticDB for PostgreSQL (base on GPDB).

# 7.How do I diagnose and handle locks?

In a database system, locking works with multiversion concurrency control (MVCC) to ensure data consistency. Assume that there are two sessions: A and B. If session A is querying data from a specific object, session B cannot perform data definition language (DDL) operations on that object. Similarly, if session A is updating a data record, session B cannot update or delete that data record.

Locks are controlled by the database system. If an application or SQL script is incorrectly designed, a transaction may remain in a LOCK WAIT state. If a transaction waits indefinitely, a deadlock occurs. AnalyticDB for PostgreSQL provides the following two views from which you can obtain information about lock waits and deadlocks:

- pg_locks: shows lock statistics. Each data record represents a process that holds a lock or is waiting for a lock.
- pg_stat_activity: shows session statistics. Each data record represents a session.

## Create a lock monitoring view

Execute the following statements to create the v_locks_monitor view and query the processes that hold locks or are waiting for locks:

> **Note** All SQL statements described in this topic are executed on the psql CLI client. You must connect to your AnalyticDB for PostgreSQL database by using the psql CLI client.

```
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,objs
ubid,
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 m
ode is null then 'NULL' else mode::text end||' ,
Username: '||case when usename is null then 'NULL' else usename::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_p
ort::text end||' , Application_Name: '||case when application_name is null then 'NULL' else application_n
ame::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()-xa
ct_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;
```

## Query locks

If a lock wait or deadlock occurs, execute the following SQL statement to query the v_locks_monitor view:

```
postgres=# \x

postgres=# select * from v_locks_monitor;
```

## Solution

From the v_locks_monitor view, you can obtain the locks in your database system. Execute the following statement to terminate the process used to run the transactions that triggered the locks:

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

In this example, PID is the Pid value for the data record whose Lock_Granted value is t in the v_locks_monitor view.

```
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
```

Execute the following statement. If the return result is " `0 rows` ", the locks are released. If the return result contains a data record, execute the `select pg_terminate_backend(PID);` statement again to terminate the corresponding process.

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

## References

SQL statements to monitor PostgreSQL lock waits - who blocked whom

# 8.Use a connection pool

AnalyticDB for PostgreSQL is built based on the PostgreSQL kernel, and supports the mainstream connection poolers such as PgBouncer and pgpool-II.

PgBouncer: a lightweight connection pooler for PostgreSQL. For more information, visit PgBouncer.

pgpool-II: a connection pooler that provides abundant features such as connection pooling, load balancing, automatic retrying of tasks, and query caching. For more information, visit pgpool-II.