

# Alibaba Cloud

## AnalyticDB for PostgreSQL Data

Document Version: 20220707

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

# Table of Contents

1.Manage databases	05
2.Manage schemas	06
3.Manage tables	08
4.Define table distribution	11
5.Define table partitioning	16
6.Define storage models for tables	20
7.Manage indexes	22
8.Manage views	25
9.Manage materialized views	26
10.Real-time materialized views	29
11.Query rewrite for materialized views	32
12.Transaction management	42
13.Manage users and permissions	44

# 1. Manage databases

A database is a collection of tables, indexes, views, stored procedures, and operators. You can create more than one database in an AnalyticDB for PostgreSQL instance. However, one client program can only connect to and access one database at a time. This means that you cannot query data across databases.

## Create a database

Execute the `CREATE DATABASE` statement to create a database. The syntax is as follows:

```
CREATE DATABASE <dbname> [ [WITH] [OWNER [=] <dbowner>] ]  
[ENCODING [=] <encoding>]
```

### Parameter description:

- `<dbname>`: the name of the database you want to create.
- `<dbowner>`: the username of the account who owns the database. By default, the user who executes the statement owns the database.
- `<encoding>`: the character set encoding to use in the database. You must specify a string constant (such as 'SQL\_ASCII') and an integer code number (UTF-8 by default).

### Example:

```
CREATE DATABASE mygpdb;
```

## Delete a database

Execute the `DROP DATABASE` statement to delete a database. This statement deletes the metadata of the database along with the directory of the database on the disk and the data contained in the database. The syntax is as follows:

```
DROP DATABASE <dbname>
```

### Parameter description:

`<dbname>`: the name of the database you want to delete.

### Example:

```
DROP DATABASE mygpdb;
```

## References

For more information, visit [Pivotal Greenplum documentation](#).

## 2. Manage schemas

A schema is the namespace of a database. It is a set of objects in a database. These objects include tables, indexes, views, stored procedures, and operators. A schema is unique to each database. Each database has a default schema named public.

If no schemas are created, objects are created in the public schema. All database roles (users) have CREATE and USAGE permissions in the public schema.

### Create a schema

Execute the `CREATE SCHEMA` statement to create a schema. The syntax is as follows:

```
CREATE SCHEMA <schema_name> [AUTHORIZATION <username>]
```

#### Note

- <schema\_name>: the name of the schema.
- <username>: the name of the role that owns the schema. If this parameter is not specified, the role that executes the statement owns the schema.

#### Example:

```
CREATE SCHEMA myschema;
```

### Set a path to search for schemas

The search\_path parameter specifies the order in which schemas are searched for.

You can use the `ALTER DATABASE` statement to set a search path. Example:

```
ALTER DATABASE mydatabase SET search_path TO myschema, public, pg_catalog;
```

You can also use the `ALTER ROLE` statement to set a search path for a specific role (user). Example:

```
ALTER ROLE sally SET search_path TO myschema, public, pg_catalog;
```

### View the current schema

Execute the `current_schema()` function to view the current schema. Example:

```
SELECT current_schema();
```


Execute the `SHOW` statement to view the current search path. Example:

```
SHOW search_path;
```

### Delete a schema

Execute the `DROP SCHEMA` statement to delete a schema. Example:

```
DROP SCHEMA myschema;
```

 **Note** By default, you can only delete a schema if it is empty.

To delete a schema and all objects (such as tables, data, and functions) in it, execute the following statement:

```
DROP SCHEMA myschema CASCADE;
```

## References

For more information, visit [CREATE SCHEMA](#).

## 3. Manage tables

Tables are similar to tables in relational databases, except that table rows are distributed across compute nodes. The distribution of rows in a table is determined by the distribution policy of the table.

### Create a standard table

The CREATE TABLE statement can be used to create a table. When you create a table, you can define the following items:

- Columns of the table and their [Data types](#)
- [Define constraints](#)
- [Define table distribution](#)
- [Storage model of the table](#)
- [Partitioning strategy of the table](#)

Execute the `CREATE TABLE` statement to create a table. The following syntax is used:

```
CREATE TABLE table_name (
  [ { column_name data_type [ DEFAULT default_expr ]      -- Define a column for the table.
    [column_constraint [ ... ]                             -- Define a constraint for the column.
  ]
    | table_constraint                                     -- Define a constraint for the table.

  ])
  [ WITH ( storage_parameter=value [, ... ] )             -- Define the storage model for the table.
  [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ] -- Define a distribution key for the table.
  [ partition clause]                                     -- Define a partitioning strategy for the table.
```

### Example:

In this example, `trans_id` is used as the distribution key, and `date`-based range partitioning is specified.


```
CREATE TABLE sales (
  trans_id int,
  date date,
  amount decimal(9,2),
  region text)
DISTRIBUTED BY (trans_id)
PARTITION BY RANGE(date)
(start (date '2018-01-01') inclusive
 end (date '2019-01-01') exclusive every (interval '1 month'),
 default partition outlying_dates);
```

### Create a temporary table



Temporary tables are used to store temporary and intermediate data. They are automatically deleted at the end of a session or deleted at the end of the current transaction based on user-defined configurations. The following statement can be used to create a temporary table:

```
CREATE TEMPORARY TABLE table_name(...)  
    [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
```

 **Note** You can use the ON COMMIT clause to determine the operation to be performed on a table at the end of the current transaction.

- PRESERVE ROWS: Data is retained at the end of the current transaction. This is the default operation.
- DELETE ROWS: All rows are deleted at the end of the current transaction.
- DROP: Temporary tables are deleted at the end of the current transaction.

### Example:

Create a temporary table that is to be deleted at the end of the current transaction.

```
CREATE TEMPORARY TABLE temp_foo (a int, b text) ON COMMIT DROP;
```

## Define constraints

You can define table or column constraints to restrict data in your tables. When you define constraints, take note of the following items:

- CHECK constraints can reference only columns in the table on which the constraints are defined.
- UNIQUE and PRIMARY KEY constraints must contain the distribution key. Such constraints are not allowed on append-optimized (AO) or column-oriented tables.
- FOREIGN KEY constraints are allowed but not enforced.
- Constraints that you define on one partition of a table are also used for the other partitions of the table. Constraint definitions cannot be limited to individual partitions.

The following syntax is used:

```
UNIQUE ( column_name [, ... ] )  
| PRIMARY KEY ( column_name [, ... ] )  
| CHECK ( expression )  
| FOREIGN KEY ( column_name [, ... ] )  
    REFERENCES table_name [ ( column_name [, ... ] ) ]  
    [ key_match_type ]  
    [ key_action ]  
    [ key_checking_mode ]
```

## CHECK constraints

You can use a CHECK constraint to specify a column that satisfies a Boolean expression. Example:

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

## NOT NULL constraints

You can use a NOT NULL constraint to specify a column that does not contain NULL values. Example:

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

## UNIQUE constraints

You can use a UNIQUE constraint to ensure that the data contained in a column or a group of columns in a table is unique among all the rows in the table. The table that contains a UNIQUE constraint must be hash-distributed, and the constraint columns must contain the distribution key. Example:


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

 **Note** does not support UNIQUE constraints.

## PRIMARY KEY constraints

A PRIMARY KEY constraint consists of a UNIQUE constraint and a NOT NULL constraint. The table that contains a PRIMARY KEY constraint must be hash-distributed, and the constraint columns must contain the distribution key. By default, if a table has a primary key, the primary key column or columns are used as the distribution key of the table. Example:

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

 **Note** does not support primary keys.

## References

For more information, see the [Pivotal Greenplum documentation](#).

# 4. Define table distribution

## Table distribution options

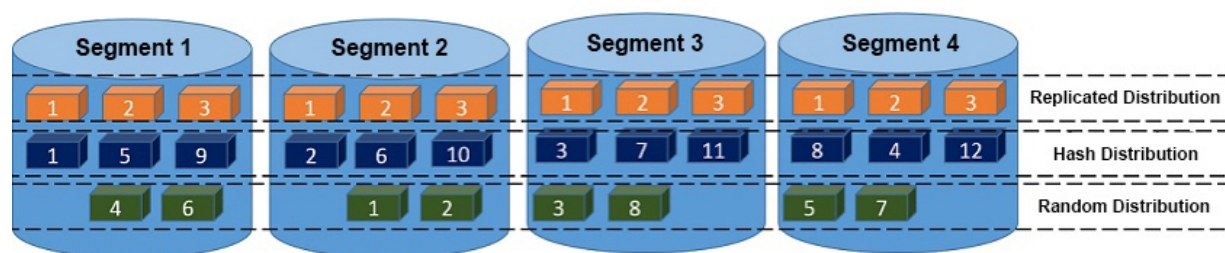
AnalyticDB for PostgreSQL provides three options to distribute the data of a table across compute nodes: hash distribution, random distribution, and replicated distribution.

```
CREATE TABLE <table_name> (...) [ DISTRIBUTED BY (<column> [,..] ) | DISTRIBUTED RANDOMLY
| DISTRIBUTED REPLICATED ]
```

**Note** AnalyticDB for PostgreSQL V4.3 only supports hash distribution and random distribution. Replicated distribution is a new feature in AnalyticDB for PostgreSQL V6.0.

The `CREATE TABLE` statement supports the following clauses that specify table distribution options:

- `DISTRIBUTED BY (column, [ ... ])` : specifies hash distribution. The rows of the table are distributed across compute nodes based on their hash values in the distribution column selected as the distribution key. Each row is assigned to one compute node. Rows with identical values are always assigned to the same compute node. You can choose a unique distribution key (for example, the primary key of the table) to ensure even distribution of data. The default table distribution option is hash distribution. If you do not specify a `DISTRIBUTED` clause, the table uses its primary key or the first identified suitable distribution column as the distribution key. If no suitable distribution column is identified, the system uses random distribution.
- `DISTRIBUTED RANDOMLY` : specifies random distribution. The rows of the table are evenly distributed across all compute nodes by using a round-robin algorithm. Rows with identical values may be assigned to different compute nodes. We recommend that you only use random distribution when no suitable distribution column is identified.
- `DISTRIBUTED REPLICATED` : specifies replicated distribution. All data of the table is stored on all compute nodes. This means that each compute node stores the same rows. If you want to join large and small tables, you can specify replicated distribution for small tables to increase join performance.



### Examples:

In the following example, a table that uses hash distribution is created. Each row is assigned to one compute node based on its hash value.

```
CREATE TABLE products (name varchar(40),
                        prod_id integer,
                        supplier_id integer)
DISTRIBUTED BY (prod_id);
```

In the following example, a table that uses random distribution is created. The rows of the table are distributed across all compute nodes by using a round-robin algorithm. If no suitable distribution column is identified, we recommend that you use random distribution.

```
CREATE TABLE random_stuff (things text,
                           doodads text,
                           etc text)
                           DISTRIBUTED RANDOMLY;
```

In the following example, a table that uses replicated distribution is created. All data of the table is stored on all compute nodes.

```
CREATE TABLE replicated_stuff (things text,
                               doodads text,
                               etc text)
                               DISTRIBUTED REPLICATED;
```

For simple queries that use a distribution key, AnalyticDB for PostgreSQL filters compute nodes based on the distribution key before sending query requests to them. Such simple queries include those initiated by UPDATE and DELETE statements. For example, if you query data from the products table that uses prod\_id as the distribution key, your query is only sent to the compute nodes whose values of prod\_id are 101. This increases your query performance.

```
select * from products where prod_id = 101;
```

## Hash keys

To increase query performance, we recommend that you choose a distribution column as the distribution key for a table based on the following rules:

- Choose one or more distribution columns with data distributed evenly. If the distribution columns you choose have unevenly distributed data, they may skew the data in the table. Tables with skewed data have one or more compute nodes with a disproportionate number of rows. In this situation, some compute nodes finish their portion of a parallel query before the others. However, based on the Cannikin Law, the query cannot be completed until all compute nodes finish processing. As a result, the query is only as fast as the slowest compute node. Therefore, we recommend that you do not choose distribution columns with Boolean or date values.
- Choose a distribution column that is frequently used in JOIN clauses. This way, you can join two tables by using a **collocated join**, as shown in the following figures. If the join key is the same as the distribution key, the join can be completed within the associated compute nodes without data movement. If you do not choose a distribution column that is frequently used in JOIN clauses, you must redistribute (**redistribute motion**) the larger one of the two tables you want to join and then perform a **redistributed join**. You also have the option to broadcast (**broadcast motion**) the smaller one of the two tables and then perform a **broadcast join**. Both the **redistribute** and **broadcast motions increase network overheads**.
- Choose a frequently used query criterion as the distribution key. This enables AnalyticDB for PostgreSQL to filter compute nodes based on the distribution key before it sends query requests to them.
- If you do not specify a distribution key, the primary key of the table is used as the distribution key. In addition, if the table does not have a primary key, the first column is used as the distribution key.

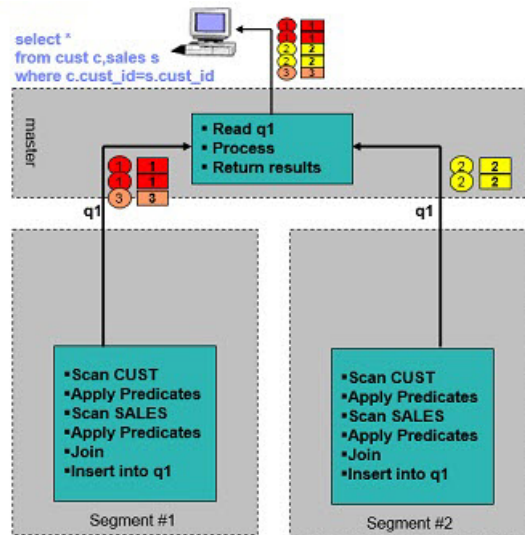
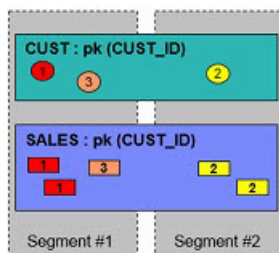
- The distribution key can be defined from one or more columns. Example:

```
create table t1(c1 int, c2 int) distributed by (c1,c2);
```

- Exercise caution when you choose random distribution because it does not support collocated joins or compute node filtering.

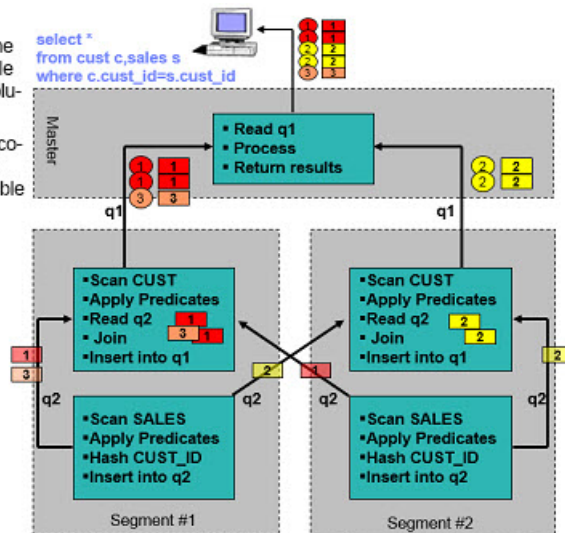
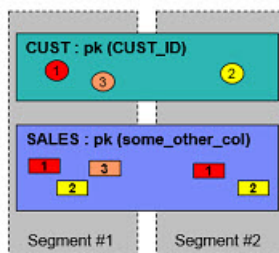
#### Collocated Join

- Both CUST and SALES tables use CUST\_ID as the distribution column
- The JOIN can be completed in each segment without data movement



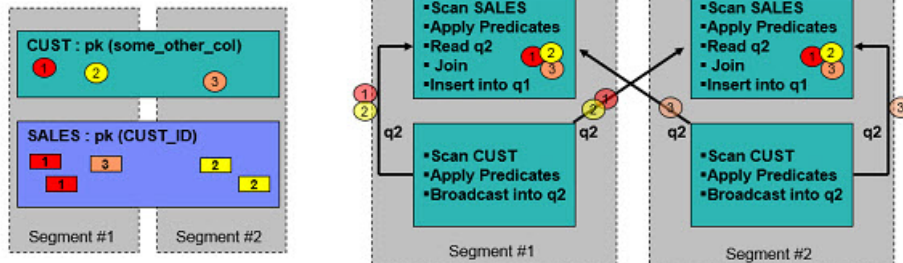
#### Redistributed Join

- The CUST table uses the CUST\_ID as the distribution column, while the SALES table uses other columns as the distribution column
- The SALES table is sent to segments according to the HASH value of the CUST\_ID column, and then JOIN with the CUST table in each segment



#### Broadcast Join

- The SALES table uses the CUST\_ID as the distribution column, while the CUST table uses other columns as the distribution column
- The CUST table is sent to all segments, and then JOIN with the SALES table in each segment



## Limits on distribution keys

- A column defined as the distribution key of a table cannot be updated.
- The distribution key of a table must be either the primary key or a unique key. Example:

```
create table t1(c1 int, c2 int, primary key (c1)) distributed by (c2);
```

**Note** In this example, the primary key c1 differs from the distribution key c2. As a result, the execution of the statement fails and the system reports the following error:

```
ERROR: PRIMARY KEY and DISTRIBUTED BY definitions incompatible
```

- A column with Geometry values or any other custom data type cannot be used as the distribution key of a table.

## Troubleshooting for data skew

If the query performance of a table is poor, check whether an inappropriate distribution key is specified. Example:

```
create table t1(c1 int, c2 int) distributed by (c1);
```

For this example, execute the following statement to check for data skew in the table:

```
select gp_segment_id, count(1) from t1 group by 1 order by 2 desc;
gp_segment_id | count
-----+-----
2 | 131191
0 | 72
1 | 68
(3 rows)
```

If you find that some compute nodes store more rows than the others, data skew occurs. We recommend that you define a column with evenly distributed data as the distribution column. For the following example, execute the `ALTER TABLE` statement to specify the c2 column as the distribution column:

```
alter table t1 set distributed by (c2);
```

The distribution key of the t1 table is changed to the c2 column. After the t1 table is redistributed based on the c2 column, its data is no longer skewed.

# 5. Define table partitioning

allows you to divide a large table into partitions. When you use conditions to query data, the system scans only the partitions that meet the specified conditions. This prevents full table scans and improves query performance.

## Supported partitioning types

- **Range partitioning:** Data is divided based on a numeric range, such as date.
- **List partitioning:** Data is divided based on a list of values, such as city attributes.
- **Multi-level partitioning:** Data is divided based on a numeric range and a list of values.

## Create a range partitioned table

You can have AnalyticDB for PostgreSQL automatically generate partitions by specifying a START value, an END value, and an EVERY clause that defines the interval within the range. By default, the START value is inclusive and the END value is exclusive.

Create a table that is range-partitioned by date. Sample statement:

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( START (date '2016-01-01') INCLUSIVE
  END (date '2017-01-01') EXCLUSIVE
  EVERY (INTERVAL '1 day') );
```

Create a table that is range-partitioned by number. For example, a column of the INT data type can be used as the partition key. Sample statement:

```
CREATE TABLE rank (id int, rank int, year int, gender char(1), count int)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
( START (2006) END (2016) EVERY (1),
  DEFAULT PARTITION extra );
```

## Create a list partitioned table

When you create a list partitioned table, you can set the partition key to any column whose data type allows value comparison, and you must declare a description for each specified value of the partition key.

Create a list partitioned table. Sample statement:

```
CREATE TABLE rank (id int, rank int, year int, gender
char(1), count int )
DISTRIBUTED BY (id)
PARTITION BY LIST (gender)
( PARTITION girls VALUES ('F'),
  PARTITION boys VALUES ('M'),
  DEFAULT PARTITION other );
```



## Create a multi-level partitioned table

allows you to create a table that has multi-level partitions. The following example demonstrates how to create a three-level partitioned table. Data in level-1 partitions is range-partitioned by year, data in level-2 partitions is range-partitioned by month, and data in level-3 partitions is list-partitioned by region. The level-2 and level-3 partitions are called subpartitions.

```
CREATE TABLE sales (id int, year int, month int, day int,
region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
    SUBPARTITION BY RANGE (month)
        SUBPARTITION TEMPLATE (
            START (1) END (13) EVERY (1),
            DEFAULT SUBPARTITION other_months )
        SUBPARTITION BY LIST (region)
            SUBPARTITION TEMPLATE (
                SUBPARTITION usa VALUES ('usa'),
                SUBPARTITION europe VALUES ('europe'),
                SUBPARTITION asia VALUES ('asia'),
                DEFAULT SUBPARTITION other_regions )
    ( START (2002) END (2012) EVERY (1),
      DEFAULT PARTITION outlying_years );
```

## Add a partition

You can execute an ALTER TABLE statement to add a partition to a partitioned table. If a subpartition template is used when the partitioned table is created, the added partition is also subpartitioned accordingly. The following example demonstrates how to add a partition:


```
ALTER TABLE sales ADD PARTITION
    START (date '2017-02-01') INCLUSIVE
    END (date '2017-03-01') EXCLUSIVE;
```

If no subpartition template is used when the partitioned table is created, you can define subpartitions when you add a partition. The following example demonstrates how to add a partition and define its subpartitions:

```
ALTER TABLE sales ADD PARTITION
    START (date '2017-02-01') INCLUSIVE
    END (date '2017-03-01') EXCLUSIVE
    ( SUBPARTITION usa VALUES ('usa'),
      SUBPARTITION asia VALUES ('asia'),
      SUBPARTITION europe VALUES ('europe') );
```

You can also use an ALTER TABLE statement to subpartition an existing partition. Sample statement:

```
ALTER TABLE sales ALTER PARTITION FOR (RANK(12))
    ADD PARTITION africa VALUES ('africa');
```

 **Notice** Partitions cannot be added to a partitioned table that has a default partition. To add partitions to such a partitioned table, you can split the default partition. For more information, see the "[Split a partition](#)" section of this topic.

## Split a partition

You can execute an ALTER TABLE statement to split a partition into two partitions. Partition splitting is subject to the following limits:

- If subpartitions exist, only the lowest level of subpartitions can be split.
- The split value specified in the AT clause of the partition splitting statement is assigned to the second partition.

For example, assume that a partition that contains the data of January 2017 is split into two partitions. The first partition contains the data of January 1 to January 15 and the second partition contains the data of January 16 to January 31. Sample statement:

```
ALTER TABLE sales SPLIT PARTITION FOR ('2017-01-01')
AT ('2017-01-16')
INTO (PARTITION jan171to15, PARTITION jan1716to31);
```

If your partitioned table has a default partition, you can add a partition by splitting the default partition. In the INTO clause, you must specify the default partition as the second partition. Sample statement:

```
ALTER TABLE sales SPLIT DEFAULT PARTITION
START ('2017-01-01') INCLUSIVE
END ('2017-02-01') EXCLUSIVE
INTO (PARTITION jan17, default partition);
```

## Determine the partition granularity

When you use partitioned tables, you need to determine the partition granularity. For example, to partition a table by time, you may choose a granularity of day, week, or month. A finer granularity results in less data in each partition but a larger number of partitions. The number of partitions is not measured by an absolute standard. We recommend that you assign no more than 200 partitions in each table. A large number of partitions may reduce database performance. For example, the query optimizer takes a long time to generate execution plans or the VACUUM operation takes a long time to complete.

## Optimize partitioned table queries

supports partition pruning for partitioned tables to improve query performance. When partition pruning is enabled, the system scans only the required partitions based on query conditions, instead of scanning the entire table. Example:

```
EXPLAIN
SELECT * FROM sales
WHERE year = 2008
      AND month = 1
      AND day = 3
      AND region = 'usa';
```

In the preceding example, the query conditions fall on the level-3 partition usa in the level-2 partition 1 in the level-1 partition 2008. Therefore, only the data in the level-3 partition usa is scanned during the query. The following execution plan shows that only one of the 468 level-3 partitions needs to be scanned.

```
Gather Motion 4:1 (slice1; segments: 4) (cost=0.00..431.00 rows=1 width=24)
  -> Sequence (cost=0.00..431.00 rows=1 width=24)
    -> Partition Selector for sales (dynamic scan id: 1) (cost=10.00..100.00 rows=25 width=4)
      Filter: year = 2008 AND month = 1 AND region = 'usa'::text
      Partitions selected: 1 (out of 468)
    -> Dynamic Table Scan on sales (dynamic scan id: 1) (cost=0.00..431.00 rows=1 width=24)
      Filter: year = 2008 AND month = 1 AND day = 3 AND region = 'usa'::text
```

## Query partition definitions

You can execute the following SQL statement to query the definitions of all partitions in a table:

```
SELECT
  partitionboundary,
  partitiontablename,
  partitionname,
  partitionlevel,
  partitionrank
FROM pg_partitions
WHERE tablename='sales';
```

## Maintain partitioned tables

You can manage partitions in partitioned tables. For example, you can add, remove, rename, truncate, exchange, and split partitions. For more information, see [Partitioning Large Tables](#).

# 6. Define storage models for tables

AnalyticDB for PostgreSQL supports two storage models for tables: row-oriented storage and column-oriented storage.


## Row-oriented table

By default, AnalyticDB for PostgreSQL uses the heap storage model in PostgreSQL to create row-oriented heap tables. Row-oriented tables are used for data that needs to be updated at a high frequency or written in real time by using the INSERT statement. A row-oriented table with a B-tree index provides high data retrieval performance when you query a small amount of data at a time.

### Example:

The following statement creates a row-oriented heap table:

```
CREATE TABLE foo (a int, b text) DISTRIBUTED BY (a);
```

 **Note** When you use Data Transmission Service (DTS) to write data into your AnalyticDB for PostgreSQL instance, the destination tables must be row-oriented tables. DTS allows data synchronization in near real time. In addition to data inserted using the INSERT statement, DTS can synchronize data updated using SQL statements such as UPDATE and DELETE.

## Column-oriented table

Data within a column-oriented table is stored by column. When you access data, only relevant columns are read. Column-oriented tables are used in data warehousing scenarios such as data queries and aggregations of a small number of columns. In these scenarios, column-oriented tables provide efficient I/O. However, column-oriented tables are less efficient in scenarios that require frequent update operations or a large number of INSERT operations. We recommend that you use a batch loading method such as COPY to insert data into column-oriented tables. Column-oriented tables provide a data compression ratio three to five times higher than that provided by row-oriented tables.

### Example:

Column-oriented tables must be append-optimized tables. This means that you must set the appendonly parameter to true for the column-oriented table you want to create.

```
CREATE TABLE bar (a int, b text)
  WITH (appendonly=true, orientation=column)
  DISTRIBUTED BY (a);
```


## Data compression

Data compression is used for column-oriented tables or for append-optimized row-oriented tables whose appendonly parameter is set to true. There are two compression types:

- Table-level compression.
- Column-level compression. You can use a unique compression algorithm for each column.

AnalyticDB for PostgreSQL only supports the following compression algorithms:

- AnalyticDB for PostgreSQL V4.3 supports zlib and RLE\_TYPE.
- AnalyticDB for PostgreSQL V6.0 supports Zstandard (zstd), zlib, RLE\_TYPE, and lz4.

 **Note** If you specify the QuickLZ compression algorithm, zlib is used instead. RLE\_TYPE is only used for column-oriented tables.

### Examples:

Create a column-oriented table that uses the zlib compression algorithm with a compression level of 5.


```
CREATE TABLE foo (a int, b text)
  WITH (appendonly=true, orientation=column, compresstype=zlib, compresslevel=5);
```

Create a column-oriented table that uses the zstd compression algorithm with a compression level of 9.

```
CREATE TABLE foo (a int, b text)
  WITH (appendonly=true, orientation=column, compresstype=zstd, compresslevel=9);
```

# 7.Manage indexes


This topic describes the index types of and their related operations.

 **Note** does not support indexes.

## Index types


supports the following index types:

- B-tree index (default index type)
- Bit map index

 **Note**

Bit map indexes enable AnalyticDB for PostgreSQL to store bit maps that each contain the values of a key. Bit map indexes serve the same purpose as a conventional index but occupy less storage space. In scenarios where indexed columns consist of 100 to 100,000 distinct values and are frequently queried in conjunction with other indexed columns, bit map indexes perform better than other index types.

- BRIN index (available only for of minor version 20210324 or later)
- GIN index (available only for )
- GiST index(available only for )

 **Note** does not support hash indexes.

## Principles for indexing

Scenarios in which to create indexes:

- Small datasets are returned from a query.

Indexes help increase the performance of queries on single data records or small datasets. Such queries include online transaction processing (OLTP) queries.

- Compressed tables are used.

On a compressed append-optimized (AO) table, indexes help increase the performance of queries because only the involved rows are decompressed.

Methods to select index types:

- Create a B-tree index on a column that has a high selectivity.

For example, in a table that has 1,000 rows, if you create an index on a column that has 800 distinct values, the selectivity of the index is 0.8. The selectivity of an index created on a column that has the same value in all rows is always 1.0.

- Create a bitmap index on a column that has a low selectivity.

In scenarios where indexed columns consist of 100 to 100,000 distinct values, bit map indexes perform better than other index types.

- Create a BRIN index if a large amount of data is sequentially distributed and if filter conditions such

as `<` , `<=` , `=` , `>=` , and `>` are used to filter data.

When large datasets are involved, BRIN indexes can provide similar performance as B-tree indexes but occupy less space.

Methods to select appropriate columns to create indexes:

- Create an index on a column that is frequently used for joins with other tables.

For example, create an index on a column used as the foreign key. This enables the query optimizer to use more join methods and therefore increases join performance.

- Create an index on a column that is frequently referenced in predicates.

The most suitable column is the one that is frequently referenced in WHERE clauses.

- Do not create an index on a frequently updated column.

If you create an index on a column that is updated frequently, the amount of data that needs to be read and written for column updates increases.

Best practices for using indexes:

- Do not create redundant indexes.

If an index is created on more than one column, indexes that have the same leading column are redundant.

- Delete indexes before you batch load data.

If you want to load a large amount of data to a table, we recommend that you delete all existing indexes on the data, load the data, and then recreate indexes on the table. This is faster than updating the indexes.

- Test and compare the performance of queries that use and do not use indexes.

Create indexes only when the performance of queries on the indexed columns improves.

- Execute the ANALYZE statement on a table after you create an index.

## Create an index

You can execute the `CREATE INDEX` statement to create an index on a table. Examples:

- B-tree index

Create a B-tree index on the gender column of the employee table.

```
CREATE INDEX gender_idx ON employee (gender);
```

- Bitmap index

Create a bitmap index on the title column of the films table.

```
CREATE INDEX title_bmp_idx ON films USING bitmap (title);
```

- BRIN index

Create a BRIN index on the c\_custkey column of the customer table.

```
CREATE INDEX c_custkey_brin_idx ON customer USING brin(c_custkey) with(pages_per_range=2)
;
```

- GIN index

Create a GIN index on the `l_comment` column of the `lineitem` table. Only supports GIN indexes.

```
CREATE INDEX lineitem_idx ON lineitem USING gin(to_tsvector('english', l_comment));
```

- GIN index

Create a GIN index on the `intarray` column of the `arrayt` table. Only supports GIN indexes.

```
CREATE INDEX arrayt_idx ON arrayt USING gin(intarray);
```

- GiST index

Create a GiST index on the `c_comment` column of the `customer` table. Only supports GiST indexes.

```
CREATE INDEX customer_idx ON customer USING gist(to_tsvector('english', c_comment));
```

## Recreate an index

You can execute the `REINDEX INDEX` statement to recreate an index on a table. Examples:

- Recreate the `my_index` index.

```
REINDEX INDEX my_index;
```


- Recreate all indexes on the `my_table` table.

```
REINDEX TABLE my_table;
```

## Delete an index

You can execute the `DROP INDEX` statement to delete an index from a table. For example, execute the following statement to delete the `title_idx` index:

```
DROP INDEX title_idx;
```

 **Note** If you want to load a large amount of data to a table, we recommend that you delete all existing indexes on the data, load the data, and then recreate indexes on the table. This is faster than updating the indexes.

## Collect indexed data

You can execute the `VACUUM` statement to collect indexed data. For example, execute the following statement to collect indexed data from the `customer` table:

```
VACUUM customer;
```

 **Note** Indexed data collection is available only for BRIN indexes.

## References

For more information about indexes, see the [Pivotal Greenplum documentation](#).



# 8. Manage views


This topic describes how to manage views for both simple and complex queries in AnalyticDB for PostgreSQL. Views are not stored on physical devices. Each view you access runs as a subquery.

## Create a view

Execute a `CREATE VIEW` statement to create a view.

**Example:**

```
CREATE VIEW myview AS SELECT * FROM products WHERE kind = 'food';
```

 **Note** Operations specified by ORDER BY and SORT clauses in a view are ignored.

## Delete a view

Execute a `DROP VIEW` statement to delete a view.

**Example:**

```
DROP VIEW myview;
```

## References

For more information, visit [Pivotal Greenplum documentation](#).

## 9. Manage materialized views

Materialized views are similar to views and allow you to save frequently used or complex queries. Materialized views are different from views in that materialized views are based on physical storage. You cannot write data to materialized views. When a query accesses a materialized view, the system returns the data that is stored in the materialized view. Data in materialized views is not automatically updated and may become obsolete. However, you can retrieve data stored in materialized views faster than retrieving the same data by using base tables or views of the base tables. Therefore, materialized views have significant performance advantages if you accept periodic data updates.

### Create a materialized view

Execute the `CREATE MATERIALIZED VIEW` statement to create a materialized view.

```
CREATE MATERIALIZED VIEW my_materialized_view as
  SELECT * FROM people WHERE age > 40
  DISTRIBUTED BY (id);

SELECT * from my_materialized_view ORDER BY age;
```

The following result is returned:

id	name	city	age
004	zhaoyi	zhennzhou	44
005	xuliui	jiaxing	54
006	maodi	shanghai	55

(3 rows)

The query defined in a materialized view is used only to populate the materialized view. The only difference between a materialized view and a table is that object identifiers (OIDs) are not automatically generated in a materialized view. The `DISTRIBUTED BY` clause is optional when you create a materialized view. If the `DISTRIBUTED BY` clause is not specified, the first column of the table is used as the distribution key.

#### Note

If a materialized view query contains an `ORDER BY` or `SORT` clause, the data may not be ordered or sorted.

### Refresh or disable a materialized view

Execute the `REFRESH MATERIALIZED VIEW` statement to update data in a materialized view.

```
INSERT INTO people VALUES ('007', 'sunshen', 'shenzhen', 60);

SELECT * from my_materialized_view ORDER BY age;
```

The following result is returned:

id	name	city	age
004	zhaoyi	zhennzhou	44
005	xuliui	jiaxing	54
006	maodi	shanghai	55

(3 rows)

```
REFRESH MATERIALIZED VIEW my_materialized_view;

SELECT * from my_materialized_view ORDER BY age;
```

The following result is returned:

id	name	city	age
004	zhaoyi	zhennzhou	44
005	xuliui	jiaxing	54
006	maodi	shanghai	55
007	sunshen	shenzhen	60

(4 rows)

If you include the `WITH NO DATA` clause in the `REFRESH MATERIALIZED VIEW` statement, the materialized view is not populated with data and cannot be scanned after the statement is executed. If a query attempts to access a materialized view that cannot be scanned, an error is returned.

```
REFRESH MATERIALIZED VIEW my_materialized_view With NO DATA;

SELECT * from my_materialized_view ORDER BY age;
ERROR:  materialized view "my_materialized_view" has not been populated
HINT:   Use the REFRESH MATERIALIZED VIEW command.

REFRESH MATERIALIZED VIEW my_materialized_view;

SELECT * from my_materialized_view ORDER BY age;
```

The following result is returned:

id	name	city	age
004	zhaoyi	zhennzhou	44
005	xuliui	jiaxing	54
006	maodi	shanghai	55
007	sunshen	shenzhen	60

(4 rows)

## Delete a materialized view

Execute the `DROP MATERIALIZED VIEW` statement to delete a materialized view.

```
CREATE MATERIALIZED VIEW depend_materialized_view as
  SELECT * FROM my_materialized_view WHERE age > 50
  DISTRIBUTED BY (id);

DROP MATERIALIZED VIEW depend_materialized_view;
```

`DROP MATERIALIZED VIEW ... CASCADE` allows you to delete all objects that depend on the materialized view. If the materialized view that you want to delete has dependent views, the materialized views are also deleted.

#### Notice

You must specify the **CASCADE** option in the `DROP MATERIALIZED VIEW` statement when you delete a materialized view that has dependent views. Otherwise, an error is returned.

```
CREATE MATERIALIZED VIEW depend_materialized_view as
  SELECT * FROM my_materialized_view WHERE age > 50
  DISTRIBUTED BY (id);

DROP MATERIALIZED VIEW my_materialized_view;
ERROR:  cannot drop materialized view my_materialized_view because other objects depend on it
DETAIL:  materialized view depend_materialized_view depends on materialized view my_materialized_view
HINT:   Use DROP ... CASCADE to drop the dependent objects too.

DROP MATERIALIZED VIEW my_materialized_view CASCADE;
```

## Scenarios

- Materialized views can be used for queries that are not time-sensitive.
- Materialized views can be used for frequently used or complex queries.
- To implement fast queries and analysis, you can create materialized views based on external data sources, such as the external tables of Object Storage Service (OSS) or MaxCompute. You can use the materialized views to store external data to on-premises storage. You can also create indexes for the materialized views.


## References

For more information, see the [Pivotal Greenplum documentation](#).

# 10. Real-time materialized views

provides real-time materialized views. Compared with non-real-time materialized views, real-time materialized views can implement automatic refresh in response to data changes without the need to execute REFRESH statements.

A real-time materialized view can be automatically updated with changes made to its base tables. Real-time materialized views support only statement-level automatic refresh. When an INSERT, COPY, UPDATE, or DELETE statement is executed on a base table, real-time materialized views created on this base table are updated in real time to ensure strong data consistency.

 **Note** In this mode, the write performance of base tables may be reduced. We recommend that you do not create more than five real-time materialized views on the same base table.

For more information about non-real-time materialized views, see [Manage materialized views](#).

## Statement-level refresh

When a statement is executed on a base table, real-time materialized views created on this base table are updated in real time to ensure data consistency. Such statements include INSERT, COPY, UPDATE, and DELETE. Real-time materialized views are updated based on the following logic:

- The database engine first updates base tables, and then updates their materialized views. If base tables fail to be updated, no data changes are made to their materialized views.
- If materialized views fail to be updated, no data changes are made to their base tables, and an error is returned for the executed statement.

For an explicit transaction that starts with the BEGIN TRANSACTION statement and ends with the COMMIT statement, after base tables are updated, data changes are also made to their materialized views in this transaction.

- If uses the default READ COMMITTED isolation level, before this transaction is committed, updates of materialized views are invisible to other transactions.
- If this transaction is rolled back, base tables and their materialized views are also rolled back.

## Limits

imposes the following limits on the query statements that are used to create real-time materialized views:

- If a query statement contains JOIN operations, only INNER JOIN can be used to join tables. OUTER JOIN or SELF JOIN is not supported.
- Query statements can contain most filtering and projection operations.
- Only the following aggregate operations are supported in a query statement: COUNT, SUM, AVG, MAX, and MIN. The HAVING clause is not supported.
- Complex statements such as those that contain subqueries and common table expressions (CTEs) are not supported.

After you create a real-time materialized view on a base table, DDL statements that you execute on the base table are subject to the following limits:

- When you execute the TRUNCATE statement on the base table, the real-time materialized view is not synchronously updated. You must manually refresh the real-time materialized view or create another materialized view.

- You must specify the CASCADE option to execute the DROP TABLE statement on the base table.
- ALTER TABLE statements on the base table cannot be used to delete or modify fields referenced by the materialized view.

The real-time materialized view feature has the following limits:

- Real-time materialized views support only standard and partitioned heap tables. Append-optimized (AO) tables are not supported.

## Scenarios

We recommend that you use real-time materialized views in the following scenarios:

- The number of rows or columns in the query results is much smaller than that of the base tables. For example, the query statement may contain a WHERE clause that effectively narrows down the results or may contain an aggregate function that consolidates multiple values into a single value.
- Large amounts of computations are required to obtain the query results of the following operations:
  - Semi-structured data analysis
  - Aggregate operations that take a long time to complete
- Base tables are not frequently changed.

Real-time materialized views can be used in all scenarios where materialized views are suitable. Compared with non-real-time materialized views, real-time materialized views are highly consistent with their base tables. When a base table changes, its real-time materialized views synchronously change with minimal performance cost. However, when you use non-real-time materialized views, you must manually update them each time their base tables change. Therefore, when large amounts of data are changed on a base table or a streaming update is performed, real-time materialized views have great advantages over non-real-time materialized views.

## Disadvantages of real-time materialized views

Real-time materialized views are similar to indexes maintained in real time. They can significantly optimize query performance but also reduce write performance.

When you create a real-time materialized view that contains only a single table, the write performance of the related database is reduced because the data in the materialized view must be synchronously updated. The write latency can be up to three times longer compared with writing data to the base table but not updating the materialized view. We recommend that you do not create more than five real-time materialized views on the same base table.

Batch data writes help reduce the maintenance overhead of real-time materialized views. When you execute COPY or INSERT statements, we recommend that you increase the number of batch processed rows within a single statement. This significantly reduces the maintenance overhead of real-time materialized views.

When the query statement used to create a real-time materialized view contains a JOIN clause to join two tables, you must optimize the write performance of the real-time materialized view. If you do not have relevant experience or encounter low performance when you test the execution, we recommend that you use a real-time materialized view that contains only a single table. The following suggestions are available in scenarios where two tables are joined:

- Use the join key of each base table as their respective distribution keys.
- Create an index for the join key of each base table.

## Create or delete a real-time materialized view

- Execute the following `CREATE INCREMENTAL MATERIALIZED VIEW` statement to create a real-time materialized view named `mv` :

```
CREATE INCREMENTAL MATERIALIZED VIEW mv AS SELECT * FROM base WHERE id > 40;
```

- Execute the following `DROP MATERIALIZED VIEW` statement to delete the `mv` materialized view:

```
DROP MATERIALIZED VIEW mv;
```

## Examples

1. Execute the following statement to create a base table:

```
CREATE TABLE test (a int, b int) DISTRIBUTED BY (a);
```

2. Execute the following statement to create a real-time materialized view:

```
CREATE INCREMENTAL MATERIALIZED VIEW mv AS SELECT * FROM TEST WHERE b > 40 DISTRIBUTED BY (a);
```

3. Execute the following statement to insert data to the base table:

```
INSERT INTO test VALUES (1, 30), (2, 40), (3, 50), (4, 60);
```

4. Execute the following statement to view data in the base table:

```
SELECT * FROM test;
```

The following result is returned:

```
a | b
---+---
1 | 30
2 | 40
3 | 50
4 | 60
(4 rows)
```

5. Execute the following statement to view data in the materialized view:

```
SELECT * FROM mv;
```

The following result is returned, which indicates that the materialized view is updated:

```
a | b
---+---
3 | 50
4 | 60
(2 rows)
```

# 11. Query rewrite for materialized views

provides the query rewrite feature for standard and real-time materialized views. This feature can significantly improve performance for JOIN operations, aggregate functions, subqueries, common table expressions (CTEs), and high-concurrency SQL statements.

Best practices: [Use real-time materialized views to accelerate queries that contain variable parameters](#)

## Features

Before query rewrite is supported, SELECT query statements must be manually modified to specify the use of materialized views. After query rewrite is supported, SELECT query statements can be automatically rewritten to use materialized views, even if the SELECT query statements reference base tables but not materialized views. This accelerates the execution of SELECT query statements.

- If a SELECT query statement is completely identical to the SELECT statement in a CREATE MATERIALIZED VIEW statement, triggers query rewrite to accelerate the query by using data in the materialized view. For more information, see the ["Complete match"](#) section of this topic.
- If a SELECT query statement is partially identical to the SELECT statement in a CREATE MATERIALIZED VIEW statement, query rewrite supplements the SELECT statement in the CREATE MATERIALIZED VIEW statement. For more information, see the [Query supplement](#) section of this topic.

## Limits

- Query rewrite is not supported for the SELECT FOR UPDATE statement.
- Query rewrite is not supported for statements that contain recursive CTEs.
- Query rewrite is not supported for queries that contain random functions such as `RANDOM()` and `NOW()`.
- If a SELECT query statement is partially identical to the SELECT statement in a CREATE MATERIALIZED VIEW statement but the requirements for query supplement are not met, query rewrite is not supported. For more information about query supplement, see the ["Query supplement"](#) section of this topic.
- Query rewrite is supported only when the minor version of AnalyticDB for PostgreSQL is V6.3.6.0 or later.

### Note

- For more information about how to view the minor version of an instance, see [View the minor engine version](#).
- For more information about how to update the minor version of an instance, see [Update the minor engine version](#).

## Enable or disable query rewrite

- Real-time materialized views

By default, query rewrite is enabled for real-time materialized views. You can execute the following statement to disable the feature:




```
SET enable_incremental_matview_query_rewrite TO off;
```

- **Standard materialized views**

By default, query rewrite is disabled for standard materialized views. You can execute the following statement to enable the feature:

```
SET enable_matview_query_rewrite TO on;
```

 **Note** You cannot enable or disable this feature for specific instances. To enable or disable query rewrite for a specific instance, .

## Complete match

In , query rewrite checks whether the syntax tree is a complete match between a SELECT query statement and the SELECT statement in a CREATE MATERIALIZED VIEW statement regardless of spaces, line breaks, comments, or aliases. If a SELECT query statement is completely identical to the SELECT statement in a CREATE MATERIALIZED VIEW statement, the materialized view is preferentially used to accelerate queries.

## Query supplement

In , query rewrite can be used if a SELECT query statement is partially identical to the SELECT statement in a CREATE MATERIALIZED VIEW statement. In this scenario, query rewrite supplements the SELECT statement in the CREATE MATERIALIZED VIEW statement and returns query results based on the materialized view.

Query supplement is supported only for the following parts in a SELECT query statement: SELECT columns, JOIN tables, GROUP BY columns, WHERE clause, HAVING clause, ORDER BY columns, and LIMIT clause. To meet the requirements for query rewrite, make sure that other parts in a SELECT query statement are completely identical to those in the SELECT statement of a CREATE MATERIALIZED VIEW statement.

- **SELECT columns**

When the SELECT columns in a SELECT query statement are partially identical to those in the SELECT statement of a CREATE MATERIALIZED VIEW statement, the following rules apply to query rewrite:

- Query rewrite is supported if the order of SELECT columns in a SELECT query statement is different from that in the SELECT statement of a CREATE MATERIALIZED VIEW statement.
- Query rewrite is supported if the SELECT columns in the SELECT statement of a CREATE MATERIALIZED VIEW statement are not included in a SELECT query statement.
- Query rewrite is supported if the SELECT columns in a SELECT query statement are not included in the SELECT statement of a CREATE MATERIALIZED VIEW statement but can be calculated from those in the SELECT statement of the CREATE MATERIALIZED VIEW statement.
- Query rewrite is not supported if the SELECT columns in a SELECT query statement are not included in the SELECT statement of a CREATE MATERIALIZED VIEW statement and cannot be calculated from those in the SELECT statement of the CREATE MATERIALIZED VIEW statement.

- **GROUP BY columns**

When the GROUP BY columns in a SELECT query statement are partially identical to those in the SELECT statement of a CREATE MATERIALIZED VIEW statement, the following rules apply to query rewrite:

- The SELECT statement in a CREATE MATERIALIZED VIEW statement does not include GROUP BY columns or aggregate functions:
  - Query rewrite is supported if a SELECT query statement includes aggregate functions.
  - Query rewrite is supported if a SELECT query statement includes GROUP BY columns.
  - Query rewrite is supported if a SELECT query statement includes GROUP BY columns and aggregate functions.
- The SELECT statement in a CREATE MATERIALIZED VIEW statement includes GROUP BY columns but not aggregate functions:
  - Query rewrite is supported if the GROUP BY columns in the SELECT statement of a CREATE MATERIALIZED VIEW statement are not included in a SELECT query statement.
  - Query rewrite is not supported if the GROUP BY columns in a SELECT query statement are not included in the SELECT statement of a CREATE MATERIALIZED VIEW statement.
  - Query rewrite is supported if the aggregate function in a SELECT query statement is `count(distinct)`.
- The SELECT statement in a CREATE MATERIALIZED VIEW statement does not include GROUP BY columns but includes aggregate functions:
  - Query rewrite is not supported if a SELECT query statement includes GROUP BY columns.
- The SELECT statement in a CREATE MATERIALIZED VIEW statement includes GROUP BY columns and aggregate functions:
  - Query rewrite is supported if the GROUP BY columns in the SELECT statement of a CREATE MATERIALIZED VIEW statement are not included in a SELECT query statement.
  - Query rewrite is not supported if the GROUP BY columns in a SELECT query statement are not included in the SELECT statement of a CREATE MATERIALIZED VIEW statement.

#### Note

- If a SELECT query statement includes fewer GROUP BY columns than the SELECT statement in a CREATE MATERIALIZED VIEW statement, query rewrite supplements the SELECT statement in the CREATE MATERIALIZED VIEW statement by performing re-aggregation on aggregate functions. The following aggregate functions are supported for re-aggregation: `COUNT`, `SUM`, `MAX`, `MIN`, and `AVG`. Query rewrite is not supported if a SELECT query statement includes other aggregate functions.
- If a SELECT query statement includes a HAVING clause, GROUP BY columns cannot be used for supplement.

#### • JOIN tables

When the JOIN tables or conditions in a SELECT query statement are partially identical to those in the SELECT statement of a CREATE MATERIALIZED VIEW statement, the following rules apply to query rewrite:

- INNER JOIN tables can be interchanged, and additional JOIN tables or conditions can be used to supplement the SELECT statement in a CREATE MATERIALIZED VIEW statement.
- LEFT OUTER JOIN and RIGHT OUTER JOIN can be converted to each other, and the left and right tables of FULL OUTER JOIN can be interchanged. Additional JOIN tables or conditions cannot be used to supplement the SELECT statement in a CREATE MATERIALIZED VIEW statement.

When the JOIN tables in a SELECT query statement are completely identical to those in the SELECT statement of a CREATE MATERIALIZED VIEW statement, the following rules apply to query rewrite:

- INNER JOIN tables can be interchanged. Examples:
  - SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT * FROM a, b WHERE a.i = b.i;
```

SELECT query statements that support query rewrite:

```
SELECT * FROM b, a WHERE a.i = b.i;
SELECT * FROM a INNER JOIN b ON a.i = b.i;
```

- SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT * FROM a INNER JOIN b ON a.i = b.i;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM b INNER JOIN a ON a.i = b.i;
```

- LEFT OUTER JOIN and RIGHT OUTER JOIN can be converted to each other. Examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT * FROM a LEFT JOIN b ON a.i = b.i;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM b RIGHT JOIN a ON b.i = a.i;
```

- The left and right tables of FULL OUTER JOIN can be interchanged. Examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT * FROM a FULL OUTER JOIN b ON a.i = b.i;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM b FULL OUTER JOIN a ON b.i = a.i;
```

When the JOIN tables in a SELECT query statement are partially identical to those in the SELECT statement of a CREATE MATERIALIZED VIEW statement, the following rules apply to query rewrite:

Additional INNER JOIN tables can be supplemented. In this scenario, INNER JOIN or COMMON JOIN tables can be interchanged, and INNER JOIN and COMMON JOIN can be converted to each other. Examples:

- SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT * FROM a, b;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM a, b, c;
```

- SELECT statement in a CREATE MATERIALIZED VIEW statement :

```
SELECT * FROM a INNER JOIN b ON a.i = b.i;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM a INNER JOIN b ON a.i = b.i INNER JOIN c ON a.i = c.i;
```

- **WHERE clause**

When the WHERE clause in a SELECT query statement is partially identical to that in the SELECT statement of a CREATE MATERIALIZED VIEW statement, the following rules apply to query rewrite:

- Both a SELECT query statement and the SELECT statement in a CREATE MATERIALIZED VIEW statement use AND to join multiple WHERE conditions:

- Query rewrite is supported if the order of WHERE conditions in a SELECT query statement is different from that in the SELECT statement of a CREATE MATERIALIZED VIEW statement.

Examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement :

```
SELECT * FROM t WHERE a > 100 AND a < 200;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM t WHERE a < 200 AND a > 100;
```

- Query rewrite is supported if a WHERE clause in a SELECT query statement is not included in the SELECT statement of a CREATE MATERIALIZED VIEW statement. In this scenario, query rewrite supplements the missing WHERE clause. The columns referenced in the WHERE clause to be supplemented must exist in the materialized view. Examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement :

```
SELECT * FROM t WHERE a > 100;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM t WHERE b > 200 AND a > 100;
```

- Both a SELECT query statement and the SELECT statement in a CREATE MATERIALIZED VIEW statement use OR to join multiple WHERE conditions:
  - Query rewrite is supported if the order of WHERE conditions in a SELECT query statement is different from that in the SELECT statement of a CREATE MATERIALIZED VIEW statement. The columns referenced in all WHERE clauses must exist in the materialized view. Examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT * FROM t WHERE a > 100 OR a < 200;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM t WHERE a < 200 OR a > 100;
```

- Query rewrite is supported if a WHERE clause in a SELECT query statement is not included in the SELECT statement of a CREATE MATERIALIZED VIEW statement. The columns referenced in all WHERE clauses must exist in the materialized view. Examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT * FROM t WHERE a > 100 OR a < 200;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM t WHERE a < 200;
```

- The WHERE clause in the SELECT statement of a CREATE MATERIALIZED VIEW statement includes that in a SELECT query statement:
  - Query rewrite is supported if the WHERE clause in a SELECT query statement consists of an equality condition but a range is specified in the WHERE clause in the SELECT statement of a CREATE MATERIALIZED VIEW statement. The columns referenced in all WHERE clauses must exist in the materialized view. Examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT * FROM t WHERE a < 200 AND a >= 100;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM t WHERE a = 102;
```

- Query rewrite is supported if a range is specified in the WHERE clauses of both a SELECT query statement and the SELECT statement in a CREATE MATERIALIZED VIEW statement. The columns referenced in all WHERE clauses must exist in the materialized view. Examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT * FROM t WHERE a < 200 AND a >= 0;
```

SELECT query statement that supports query rewrite:

```
SELECT * FROM t WHERE a <= 100 AND a > 50;
```

- HAVING clause**

When the HAVING clause in a SELECT query statement is partially identical to that in the SELECT statement of a CREATE MATERIALIZED VIEW statement, the following rules apply to query rewrite:

- If GROUP BY columns do not need to be supplemented, query rewrite supplements the HAVING clause in a manner similar to when it supplements the WHERE clause. In the SELECT statement of a CREATE MATERIALIZED VIEW statement, missing AND conditions can be supplemented, additional OR conditions can be removed, and the range can be narrowed down.
- If GROUP BY columns need to be supplemented, query rewrite is supported in the scenario where a SELECT query statement includes a HAVING clause but the SELECT statement in a CREATE MATERIALIZED VIEW statement does not.

- **ORDER BY columns**

Regardless of whether the SELECT statement in a CREATE MATERIALIZED VIEW statement includes ORDER BY columns, query rewrite attempts to supplement ORDER BY columns. To meet the requirements for query rewrite, make sure that ORDER BY columns of a SELECT query statement are included in those of the SELECT statement in a CREATE MATERIALIZED VIEW statement.

- **LIMIT clause**

If the SELECT statement in a CREATE MATERIALIZED VIEW statement does not include a LIMIT clause, query rewrite supplements the LIMIT clause. If the SELECT statement in a CREATE MATERIALIZED VIEW statement includes a LIMIT clause, the SELECT query statement must be completely identical to the SELECT statement in the CREATE MATERIALIZED VIEW statement.

- **Expression supplement**

If the ordinary expression or aggregate function expression in a SELECT query statement does not match the expression in the SELECT statement of a CREATE MATERIALIZED VIEW statement, sub-expressions in the SELECT query statement are used from the top down to find the closest match. Examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT a+b, c FROM t;
```

SELECT query statements that support query rewrite:

```
SELECT a+b, (a+b)+c, mod(a+b, c) FROM t;  
SELECT sum((a+b)*c) FROM t;
```

If aggregate function expressions are included, the following rules apply to query rewrite:

- Aggregate functions `SUM()` and `COUNT()` in the SELECT statement of a CREATE MATERIALIZED VIEW statement can be calculated into an `AVG()` aggregate function.
- Aggregate functions `COUNT(*)` and `COUNT(1)` can be interchanged between a SELECT query statement and the SELECT statement in a CREATE MATERIALIZED VIEW statement.

Aggregate function expression examples:

SELECT statement in a CREATE MATERIALIZED VIEW statement:

```
SELECT sum(a), count(a), count(*) FROM t;
```

SELECT query statement that supports query rewrite:

```
SELECT avg(a), count(1) FROM;
```

## CTEs and subqueries

If CTEs and subqueries are included, the following rules apply to query rewrite based on primary queries and subqueries. A CTE in a WITH clause is equivalent to a subquery.

- A SELECT query statement includes only a single subquery:
  - If the primary query and subquery in a SELECT query statement are completely identical to those in the SELECT statement of a CREATE MATERIALIZED VIEW statement, query rewrite replaces the SELECT statement of the CREATE MATERIALIZED VIEW statement by using the complete match method.
  - If the subquery in a SELECT query statement is completely identical to that in the SELECT statement of a CREATE MATERIALIZED VIEW statement but the primary query is different, query rewrite supplements the SELECT statement of the CREATE MATERIALIZED VIEW statement. For more information about query supplement, see the "Query supplement" section of this topic.
  - A SELECT query statement includes a subquery but the SELECT statement in a CREATE MATERIALIZED VIEW statement does not:
    - If the SELECT statement in a CREATE MATERIALIZED VIEW statement is identical to the primary query in a SELECT query statement or can be supplemented, query rewrite supplements a subquery to the SELECT statement of the CREATE MATERIALIZED VIEW statement. Associated subqueries cannot be supplemented.
    - If the SELECT statement of a CREATE MATERIALIZED VIEW statement is identical to the subquery in a SELECT query statement or can be supplemented, query rewrite replaces the subquery in the SELECT query statement.
    - If the primary query or subquery in a SELECT query statement is replaced with a materialized view, query rewrite continues to replace other parts of the SELECT query statement.
  - Query rewrite does not support replacement for recursive CTEs.
- A SELECT query statement includes multiple subqueries:

If the primary query or a subquery in a SELECT query statement is rewritten, query rewrite continues to rewrite other parts of the SELECT query statement based on the preceding rules.

## UNION, EXCEPT, and INTERSECT

- If both a SELECT query statement and the SELECT statement in a CREATE MATERIALIZED VIEW statement include UNION, EXCEPT, or INTERSECT, two queries before and after UNION or INTERSECT can be interchanged, and the UNION or INTERSECT clause can be supplemented. The EXCEPT clause can be supplemented, but two queries before and after EXCEPT cannot be interchanged.
- If a SELECT query statement includes UNION, EXCEPT, or INTERSECT but the SELECT statement in a CREATE MATERIALIZED VIEW statement does not, a UNION, EXCEPT, or INTERSECT clause can be supplemented to join multiple materialized views.

## Match multiple materialized views

If a SELECT query statement matches multiple materialized views, query rewrite selects materialized views based on the following rules:

- The materialized view that completely matches the SELECT query statement is preferentially selected. If this materialized view does not exist, a materialized view that can be supplemented is selected.
- If multiple materialized views that can be supplemented exist, the materialized view that matches the most tables with those of the SELECT query statement is preferentially selected.

- If multiple materialized views that can be supplemented reference the same number of tables as the SELECT query statement, the materialized view that references the least data is preferentially selected.

## Examples

- Example 1:

- Execute the following statement to create a base table:

```
CREATE TABLE t1 (a int, b int) DISTRIBUTED BY (a);
```

- Execute the following statement to insert data to the base table:

```
INSERT INTO t1 VALUES (generate_series(1, 10), generate_series(1, 2));
```

- Execute the following statement to create a materialized view:

```
CREATE INCREMENTAL MATERIALIZED VIEW mv AS SELECT count(a), b FROM t1 GROUP BY b DISTRIBUTED BY (b);
```

- Execute the following statement to run a query plan:

```
EXPLAIN SELECT count(a), b FROM t1 GROUP BY b;
```

The following result is returned. Query rewrite uses the complete match method to replace the SELECT statement in the CREATE MATERIALIZED VIEW statement and returns the data of the materialized view mv.

```

                                QUERY PLAN
-----
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..2.02 rows=2 width=12)
    -> Seq Scan on mv  (cost=0.00..2.02 rows=1 width=12)
    Optimizer: Postgres query optimizer
    (3 rows)

```

- Example 2:

- Execute the following statements to create two base tables:

```
CREATE TABLE t1 (a int, b int) DISTRIBUTED BY (a);
CREATE TABLE t2 (i int, j int) DISTRIBUTED BY (i);
```

- Execute the following statements to insert data to the base tables:

```
INSERT INTO t1 VALUES (generate_series(1, 10), generate_series(1, 2));
INSERT INTO t2 VALUES (generate_series(1, 10), generate_series(1, 2));
```

- Execute the following statement to create a materialized view:

```
CREATE INCREMENTAL MATERIALIZED VIEW mv AS SELECT count(a), a, b FROM t1 GROUP BY a, b DISTRIBUTED BY (a);
```

- Execute the following statement to run a query plan:

```
EXPLAIN SELECT count(a) FROM t1 JOIN t2 ON t1.a = t2.i WHERE b > 3 GROUP BY a;
```

The following result is returned. Query rewrite returns the data of the materialized view mv after supplementing the JOIN and WHERE clauses and removing the GROUP BY clause.



```
QUERY PLAN
-----
Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..437.00 rows=1 width=8)
  -> Result  (cost=0.00..437.00 rows=1 width=8)
    -> GroupAggregate  (cost=0.00..437.00 rows=1 width=8)
      Group Key: mv.a
      -> Sort  (cost=0.00..437.00 rows=1 width=12)
        Sort Key: mv.a
        -> Hash Join  (cost=0.00..437.00 rows=1 width=12)
          Hash Cond: (mv.a = t2.i)
          -> Index Scan using mv_index on mv  (cost=0.00..6.00 rows=1 width=12)
            Index Cond: (b > 3)
          -> Hash  (cost=431.00..431.00 rows=4 width=4)
            -> Seq Scan on t2  (cost=0.00..431.00 rows=4 width=4)
Optimizer: Pivotal Optimizer (GPORCA) version 3.86.0
(13 rows)
```

# 12.Transaction management

AnalyticDB for PostgreSQL supports standard attributes of database transactions and three isolation levels. These attributes include atomicity, consistency, isolation, and durability, which are collectively referred to as ACID. AnalyticDB for PostgreSQL uses a distributed massively parallel processing (MPP) architecture to horizontally scale nodes and ensure transaction consistency between nodes. This topic describes the transaction isolation levels and transaction-related operations supported by AnalyticDB for PostgreSQL.

## Isolation levels

AnalyticDB for PostgreSQL provides the following three transaction isolation levels:

- **READ UNCOMMITTED**: follows standard SQL syntax. However, this isolation level is implemented the same as the **READ COMMITTED** isolation level in AnalyticDB for PostgreSQL.
- **READ COMMITTED**: follows standard SQL syntax and is implemented the same as the **READ COMMITTED** isolation level in AnalyticDB for PostgreSQL.
- **SERIALIZABLE**: follows standard SQL syntax. However, this isolation level is implemented the same as the **REPEATABLE READ** isolation level in AnalyticDB for PostgreSQL.

### Example:

Execute the following statements to start a transaction block with the **SERIALIZABLE** isolation level:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN;  
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

## SQL statements supported

AnalyticDB for PostgreSQL provides the following SQL statements for you to manage transactions:

- **BEGIN** and **START**: each start a transaction block.
- **END** and **COMMIT**: each commit a transaction.
- **ROLLBACK**: rolls back a transaction with no changes retained.
- **SAVEPOINT**: creates a savepoint within a transaction. You can revoke the SQL statements executed after the savepoint was created.
- **ROLLBACK TO SAVEPOINT**: rolls back a transaction to a savepoint.
- **RELEASE SAVEPOINT**: releases a savepoint from a transaction.

### Examples:

Execute the following statements to create a savepoint in a transaction and revoke the SQL statements executed after the savepoint is created:

```
BEGIN;  
  INSERT INTO table1 VALUES (1);  
  SAVEPOINT my_savepoint;  
  INSERT INTO table1 VALUES (2);  
  ROLLBACK TO SAVEPOINT my_savepoint;  
  INSERT INTO table1 VALUES (3);  
COMMIT;
```

In this example, the values 1 and 3 are inserted, but the value 2 is not.

Execute the following statements to create a savepoint in a transaction and then release the savepoint:


```
BEGIN;  
  INSERT INTO table1 VALUES (3);  
  SAVEPOINT my_savepoint;  
  INSERT INTO table1 VALUES (4);  
  RELEASE SAVEPOINT my_savepoint;  
COMMIT;
```

In this example, the values 3 and 4 are inserted.

# 13. Manage users and permissions

## Manage users

When you create an instance, the system prompts you to specify an initial username and password. This initial user is the root user. After the instance is created, you can use the credentials of the root user to connect to a database on that instance. After you use the psql CLI client of PostgreSQL or Greenplum to connect to a database on your instance, you can run the `\du+` command to view the information of all the users. Example:

 **Notice** In addition to the root user, other users are also created to manage databases.

```
postgres=> \du+
                                List of roles
  Role name  |          Attributes          | Member of | Description
-----+-----+-----+-----
 root_user  |                                |           | rds_superuser
 ...
```

AnalyticDB for PostgreSQL does not provide a superuser, which is equivalent to the RDS\_SUPERUSER role. This is the same in ApsaraDB RDS for PostgreSQL. However, you can grant the RDS\_SUPERUSER role to the root user, for example, the root\_user created in the preceding example. You can only check whether the root user has this role based on the user description. The root user has the following permissions:

- Creates databases and accounts and logs on to databases, but does not have the credentials of a superuser.
- Views and modifies the tables created by users other than a superuser, changes the owners of tables, and performs operations such as SELECT, UPDATE, and DELETE.
- Views connections to users other than a superuser, cancels their SQL statements, and terminates their connections.
- Executes CREATE EXTENSION and DROP EXTENSION statements to create and delete extensions.
- Creates users who have the RDS\_SUPERUSER role. Example:

```
CREATE ROLE root_user2 RDS_SUPERUSER LOGIN PASSWORD 'xyz';
```

## Manage permissions

You can manage permissions at the database, schema, and table levels. For example, if you want to grant read permissions on a table to a user and revoke write permissions, execute the following statements:

```
GRANT SELECT ON TABLE t1 TO normal_user1;
REVOKE UPDATE ON TABLE t1 FROM normal_user1;
REVOKE DELETE ON TABLE t1 FROM normal_user1;
```

## References

For more information, visit [Managing Roles and Privileges](#).