ALIBABA CLOUD

Alibaba Cloud

AnalyticDB for PostgreSQL

Data

Document Version: 20200904

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.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 Pivotal Greenplum documentation.

3.Create and manage tables

AnalyticDB for PostgreSQL tables are similar to tables in a relational database, except that table rows are distributed across segments. The distribution of the rows in a table is determined by the distribution policy of the table.

Create a standard table

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

- Columns of the table and their data types. For more information about data types, see Data types.
- Constraints of the table.
- Define table distribution
- Storage model of the table.
- Partitioning strategy of the table.

Execute the `CREATE TABLE` statement to create a table. The syntax is as follows:

```
CREATE TABLE table_name (
[ { column_name data_type [ DEFAULT default_expr ] -- Defines a column for the table.
[column_constraint [ ... ] -- Defines a constraint for the column of the table.
]
| table_constraint -- Defines a constraint for the table.
])
[ WITH ( storage_parameter=value [, ... ] ) -- Defines a storage model for the table.
[ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ] -- Defines a distribution key for the table.
[ partition clause] -- Defines 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 automatically deleted at the end of a session or deleted at the end of the current transaction based on user-defined configuration. Temporary tables are used to store temporary, intermediate data. The statement to create a temporary table is as follows:

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

> **Note** You can use the ON COMMIT clause to control 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, but there are the following limits:

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

The syntax is as follows:

```
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 constraint

A CHECK constraint allows you to specify that the value in a specific column must satisfy a Boolean expression. Example:

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

NOT NULL constraint

A NOT NULL constraint allows you to specify that a column cannot contain NULL values. Example:

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

UNIQUE constraint

A UNIQUE constraint ensures 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);
```

PRIMARY KEY constraint

A PRIMARY KEY constraint is a combination of a UNIQUE constraint and a NOT NULL constraint. The table on which a PRIMARY KEY constraint is defined must be hash-distributed, and the constraint columns must contain the distribution key. By default, if a table has a primary key, the column or a group of columns that correspond to the primary key are used as the distribution key for the table. Example:

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

References

For more information, visit Pivotal Greenplum documentation.

4.Define table distribution

Table distribution options

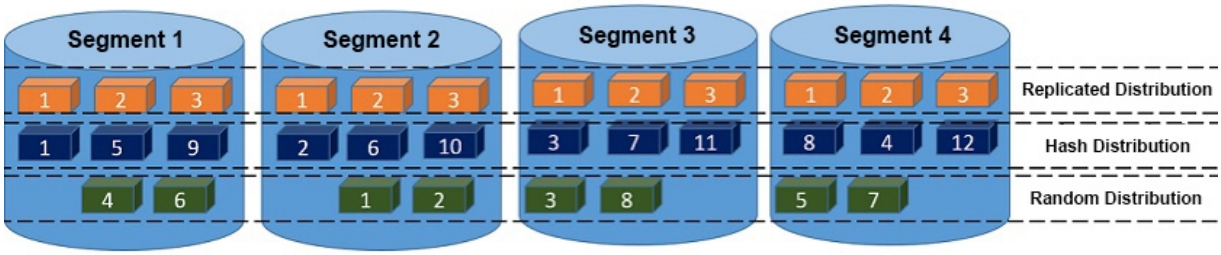
AnalyticDB for PostgreSQL provides three options to distribute the data of a table across compute nodes: hash distribution, random distribution, and replicated distribution.

```
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:

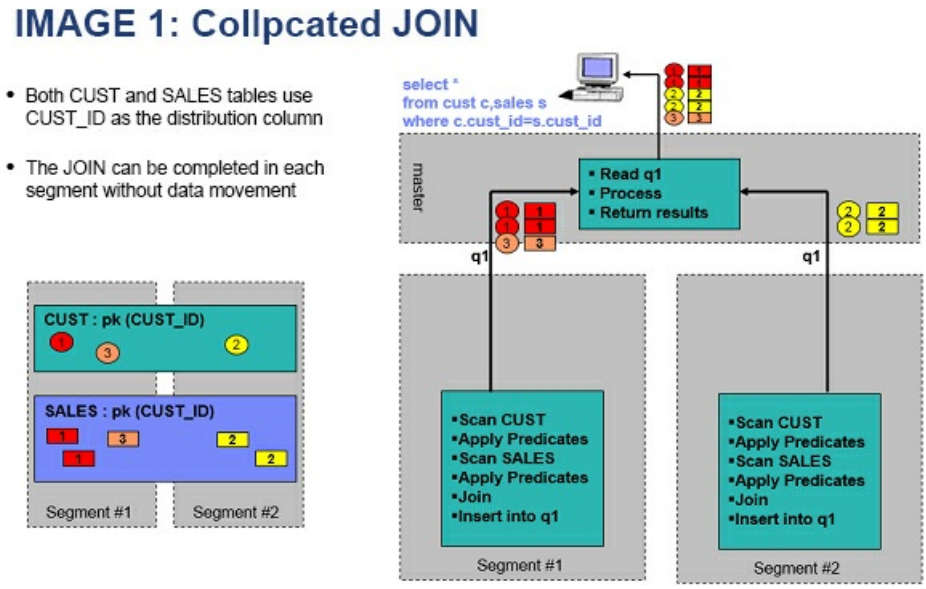
- Choose one or more distribution columns with data distributed evenly. If the distribution columns you choose have unevenly distributed data, they may skew the data in the table. Tables with skewed data have one or more compute nodes with a disproportionate number of rows. In this situation, some compute 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

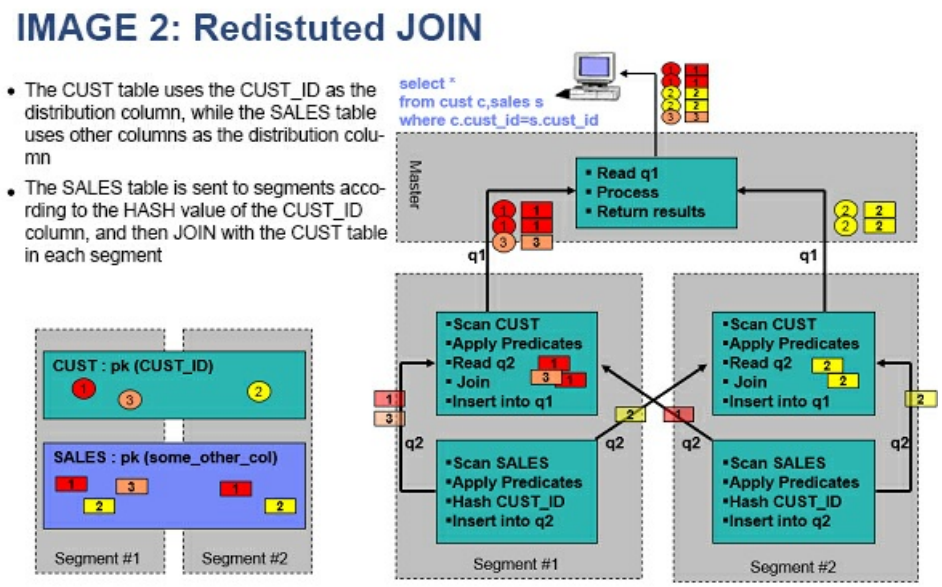
without data movement. If you do not choose a distribution column that is frequently used in JOIN clauses, you must redistribute (**redistribute motion**) the larger one of the two tables you want to join and then perform a **redistributed join**. You also have the option to broadcast (**broadcast motion**) the smaller one of the two tables and then perform a **broadcast join**. **Both the redistribute and broadcast motions increase network overheads.**

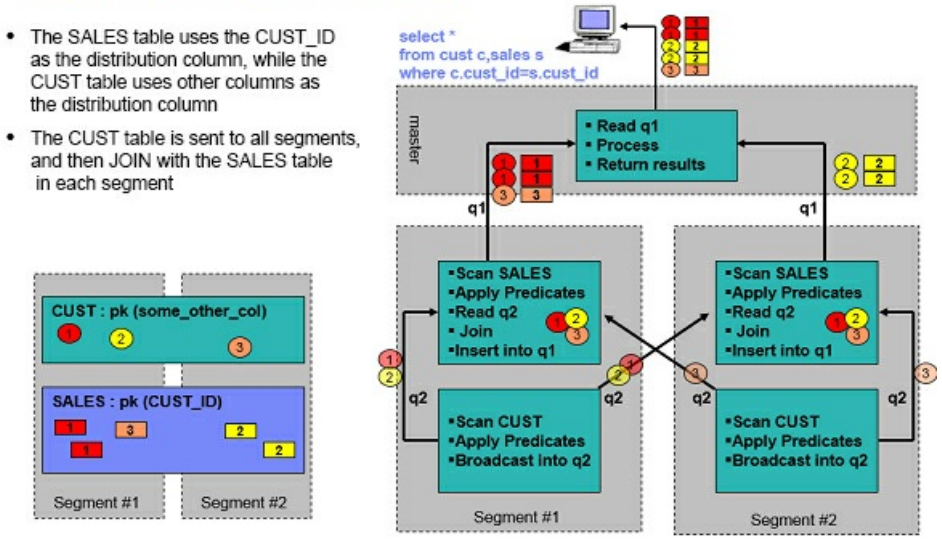
- Choose a frequently used query criterion as the distribution key. This enables AnalyticDB for PostgreSQL to filter compute nodes based on the distribution key before it sends query requests to them.
- If you do not specify a distribution key, the primary key of the table is used as the distribution 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.







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);
```

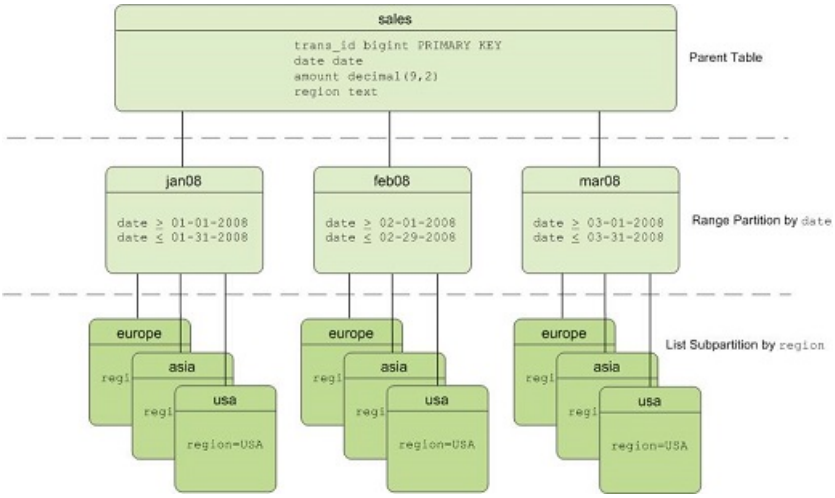
The distribution key of the t1 table is changed to the c2 column. After the t1 table is redistributed based on the c2 column, its data is no longer skewed.

5.Table partitioning

A large table can be divided into smaller storage units called partitions. After you specify query criteria, only the data in partitions that meet the criteria is scanned. This improves query performance.

Table partitioning types supported

- **Range partitioning:** Data is divided based on a numerical 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 both a numerical range and a list of values.



The preceding figure shows an example of a multi-level partitioned table. Data in level-1 partitions is divided by month, and that in level-2 partitions is divided based on the values of regions.

Range partitioned table

You can have AnalyticDB for PostgreSQL automatically generate partitions by giving a START value, an END value, and an EVERY clause that defines the partition increment value. By default, START values are always inclusive and END values are always exclusive. Example:

```
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') );
```

You can create a range partitioned table that uses a single numeric data type column as the partition key. Example:

```
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 );
```

List partitioned table

A list partitioned table can use any data type column that allows equality comparisons as its partition key. For list partitions, you must declare a partition specification for every partition (list value) you want to create. Example:

```
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 );
```

Multi-level partitioned table

You can create a table with multi-level partitions. The following example shows a three-level partition design where the sales table is partitioned by year, month, and then region.

```
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 (2008) END (2016) EVERY (1),
DEFAULT PARTITION outlying_years);
```

Granularity of table partitioning

For a table partitioned by time, the granularity may be a day, week, or month. A finer granularity results in a smaller amount of data in each partition but a larger number of partitions. There are no absolute standards regarding the number of partitions. In most cases, the number of partitions is about 200. This number is relatively large. A large number of partitions has many impacts. For example, it slows operations of the query optimizer and many maintenance operations (such as VACUUM).

Optimization of queries on partitioned tables

AnalyticDB for PostgreSQL supports partition truncating for partitioned tables. Only required partitions are scanned based on query criteria, which improves query performance. The following query is used as an example:

```
explain
select * from sales
where year = 2008
and month = 1
and day = 3
and region = 'usa';
```

The query criteria fall on the level-3 subpartition 'usa' of the level-2 subpartition 1 of level-1 partition 2008. Therefore, only the data in the level-3 subpartition 'usa' is scanned during query. As shown in the following query plan, only one of the 468 level-3 subpartitions is read.

```
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
```

Partition definition query

You can use 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';
```

References

AnalyticDB for PostgreSQL supports diverse partition-related operations such as adding, dropping, clearing, and splitting. For more information, visit Greenplum documentation.

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

Index types

AnalyticDB for PostgreSQL supports B-tree and bitmap indexes, but does not support hash indexes. Only AnalyticDB for PostgreSQL V6.0 supports GIN indexes and GiST indexes.

> **Note** The default index type is B-tree. Bitmap indexes enable AnalyticDB for PostgreSQL to store bitmaps that each contain the values of a key. Bitmap indexes provide the same functions as a conventional index while occupying less storage space. Bitmap indexes perform best for columns that have 100 to 100,000 distinct values and when indexed columns are often queried in conjunction with other indexed columns.

Create an index

Execute a `CREATE INDEX` statement to create a B-tree index.

Examples:

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

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

Execute the following statement to create a bitmap index on the title column of the films table:

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

Execute the following statement to create a GIN index on the l_comment column of the lineitem table. Note that only AnalyticDB for PostgreSQL V6.0 supports GIN indexes.

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

Execute the following statement to create a GIN index on the intarray column of the arrayt table. Note that only AnalyticDB for PostgreSQL V6.0 supports GIN indexes.

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

Execute the following statement to create a GiST index on the c_comment column of the customer table. Note that only AnalyticDB for PostgreSQL V6.0 supports GiST indexes.

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

Rebuild an index

Execute a `REINDEX INDEX` statement to rebuild an index.

Examples:

Execute the following statement to rebuild the my_index index:

```
REINDEX INDEX my_index;
```

Execute the following statement to rebuild all indexes on the my_table table:

```
REINDEX TABLE my_table;
```

Delete an index from a table

Execute a `DROP INDEX` statement to delete an index.

Example:

```
DROP INDEX title_idx;
```

> **Note** If you want to load a large volume of data to a table, we recommend that you first delete all indexes on the data and then load the data to quickly rebuild the indexes on the table.

Principles for indexing

- Create an index based on query loads.

  A query load-based index helps increase the performance of queries for single data records or small data sets. Such queries include online transaction processing (OLTP) queries.

- Create an index on a compressed table.

  On a compressed append-optimized table, an index helps increase the performance of queries for single rows because only the involved rows are decompressed.

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

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

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

  For example, if a table has 1,000 rows and 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 that has a low selectivity. Bitmap indexes perform best for columns that have 100 to 100,000 distinct values.
- 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.

  For example, create an index on a column that is frequently referenced in WHERE clauses.

- Do not create redundant indexes.

For example, if an index is created on more than one column, indexes with the same leading column are redundant.

- Delete indexes before you load data.

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

- Test and compare the performance of queries with and without indexes used.

  Do not create indexes unless the performance of queries for indexed columns increases.

- Execute the ANALYZE statement after you create or update an index.

References

For more information, visit CREATE INDEX in 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. Different from views, 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. The data in materialized views is not automatically updated and may become obsolete. However, you can retrieve data stored in the materialized views faster than retrieving the same data from underlying tables or using views to retrieve the same data from underlying tables. Therefore, if you accept periodic data updates, materialized views have significant performance advantages.

Create a materialized view

Run the `CREATE MATERIALIZED VIEW` command to create a materialized view of a query.

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

SELECT * from my_materialized_view ORDER BY age;
id | name | city | age
------------+------------+------------+-----
004 | zhaoyi | zhenzhou | 44
005 | xuliui | jiaxing | 54
006 | maodi | shanghai | 55
(3 rows)
```

The query defined in a materialized view is executed and used to populate the view at the time the command is issued. A materialized view has many of the same properties as a table, except that object identifiers (OIDs) cannot be automatically generated. 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

Run the `REFRESH MATERIALIZED VIEW` command to update data in a materialized view.

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

SELECT * from my_materialized_view ORDER BY age;
id | name | city | age
------------+------------+------------+-----
004 | zhaoyi | zhenzhou | 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;
id | name | city | age
------------+------------+------------+-----
004 | zhaoyi | zhenzhou | 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 command, the materialized view is not populated with data and is flagged as unscannable. If a query attempts to access an unscannable materialized view, 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;
id | name | city | age
------------+------------+------------+-----
004 | zhaoyi | zhenzhou | 44
005 | xuliui | jiaxing | 54
006 | maodi | shanghai | 55
007 | sunshen | shenzhen | 60
(4 rows)
```

Remove a materialized view

Run the `DROP MATERIALIZED VIEW` command to remove 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;
```

You can use `DROP MATERIALIZED VIEW ... CASCADE` to remove objects that depend on the materialized view. To be specific, the other materialized views that depend on the materialized view to be removed are also removed.

> **Notice** You must specify the **CASCADE** option in the DROP MATERIALIZED VIEW command when you remove 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_v
iew
HINT: Use DROP ... CASCADE to drop the dependent objects too.

DROP MATERIALIZED VIEW my_materialized_view CASCADE;
```

Scenarios

- Materialized views apply to queries that are not time-sensitive.
- Materialized views apply to 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) and 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.Space reclaim

Background information

After data in a table is deleted or updated, the physical storage layer does not delete the data but marks it as invisible. In this situation, a number of holes are left in data pages. When the system reads data, these holes are loaded with the data pages. This slows down data scans. Therefore, you must reclaim space regularly.

How to reclaim space

Use a `VACUUM` statement to rearrange the data in a table and reclaim disk space for better data read performance. The statement syntax is as follows:

```
VACUUM [FULL] [FREEZE] [VERBOSE] [table];
```

A VACUUM statement organizes data on data pages, whereas a VACUUM FULL statement moves data across data pages. A VACUUM operation runs faster. A VACUUM FULL operation reclaims more space but requests exclusive locks. We recommend that you perform a VACUUM operation on system tables once a week.

Precautions

VACUUM:

- VACUUM statements cannot reclaim all space of a table.
- For tables whose data is frequently updated in real time, execute a VACUUM statement once a day.
- If data is updated in batches once a day, you can execute the statement once after the batch update is complete.
- The system does not lock tables on which VACUUM statements are being executed. You can read data from the tables or write data into them while the statements are executed. However, VACUUM statements increase CPU utilization and I/O usage, which may affect query performance.

VACUUM FULL:

- VACUUM FULL statements reclaim space by rearranging data in tables and can reclaim all space occupied by holes. For tables with most data updates, we recommend that you execute VACUUM FULL statements as soon as possible.
- VACUUM FULL statements must be executed at least once a week. If the majority of your data is updated every day, execute a VACUUM FULL statement once a day.
- The system locks tables on which VACUUM FULL statements are being executed. You can neither read data from the tables nor write data into them while the statements are executed. VACUUM FULL statements also increase CPU utilization and I/O usage. We recommend that you execute VACUUM FULL statements in the maintenance window.

Query the tables for which VACUUM statements are required

AnalyticDB for PostgreSQL provides a gp_bloat_diag view to measure the ratio of actual pages to expected pages. After you use an ANALYZE statement to collect statistics on tables, execute the following statement to check this view:

```
gpadmin=# SELECT * FROM gp_toolkit.gp_bloat_diag;
bdirelid | bdinspname | bdirelname | bdirelpages | bdiexppages | bdidiag
----------+------------+------------+-------------+-------------+-----------------------------------------
21488 | public | t1 | 97 | 1 | significant amount of bloat suspected
(1 row)
```

The results only include tables with moderate or significant data bloat. If the ratio of actual pages to expected pages is greater than 4 but less than 10, moderate data bloat is reported. If the ratio is greater than 10, significant data bloat is reported. For these tables, we recommend that you use VACUUM FULL statements to reclaim space.

When to use VACUUM FREEZE

Each transaction executed by AnalyticDB for PostgreSQL has a unique transaction ID (XID). XIDs monotonically increase, with an upper limit of two billion.

If the XID of a database approaches the value of the xid_stop_limit-xid_warn_limit parameter (500000000 by default), AnalyticDB for PostgreSQL returns a warning to the SQL statement that executes the transaction that corresponds to the XID to ensure that the XID does not exceed the limit.

```
WARNING: database "database_name" must be vacuumed within number_of_transactions transactions
```

In this situation, you can manually execute a VACUUM FREEZE statement on the current database to reduce the XID.

If you ignore this warning and XIDs continue to increase to a value greater than the value of the xid_stop_limit parameter (1000000000 by default), AnalyticDB for PostgreSQL rejects new transactions and returns the following message:

```
FATAL: database is not accepting commands to avoid wraparound data loss in database "database_n
ame"
```

In this situation, you must submit a ticket to contact Alibaba Cloud technical support and resolve the issue.