GaussDB

Feature Guide(Centralized_V2.0-3.x)

Issue	01
Date	2025-08-22

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Huawei Cloud Computing Technologies Co., Ltd.

Address:	Huawei Cloud Data Center Jiaoxinggong Road Qianzhong Avenue Gui'an New District Gui Zhou 550029 People's Republic of China
Website:	https://www.huaweicloud.com/intl/en-us/

Contents

1 Materialized View

A materialized view is a special physical table, which is relative to an ordinary view. An ordinary view is a virtual table with many application limitations. Any query on a view is actually converted into a query on an SQL statement, and performance is not actually improved. The materialized view serves as a cache to store the results of the statements executed by SQL.

Currently, Ustore does not support the creation or use of materialized views.

1.1 Complete-Refresh Materialized View

1.1.1 Overview

Complete-refresh materialized views refer to the created materialized views that support only complete refresh but do not support fast refresh. The syntax for creating a complete-refresh materialized view is similar to the CREATE TABLE AS syntax.

1.1.2 Usage

Syntax

- Create a complete-refresh materialized view.
  ```
  CREATE MATERIALIZED VIEW [ view_name ] AS { query_block };
  ```
- Refresh a complete-refresh materialized view.
  ```
  REFRESH MATERIALIZED VIEW [ view_name ];
  ```
- Drop a materialized view.
  ```
  DROP MATERIALIZED VIEW [ view_name ];
  ```
- Query a materialized view.
  ```
  SELECT * FROM [ view_name ];
  ```

Examples

```
-- Prepare data.
gaussdb=# CREATE TABLE t1(c1 int, c2 int);
gaussdb=# INSERT INTO t1 VALUES(1, 1);
gaussdb=# INSERT INTO t1 VALUES(2, 2);

-- Create a complete-refresh materialized view.
```

```
gaussdb=# CREATE MATERIALIZED VIEW mv AS select count(*) from t1;
CREATE MATERIALIZED VIEW

-- Query the materialized view result.
gaussdb=# SELECT * FROM mv;
 count
-------
     2
(1 row)

-- Insert data into the base table in the materialized view.
gaussdb=# INSERT INTO t1 VALUES(3, 3);
INSERT 0 1

-- Fully refresh the complete-refresh materialized view.
gaussdb=# REFRESH MATERIALIZED VIEW mv;
REFRESH MATERIALIZED VIEW

-- Query the materialized view result.
gaussdb=# SELECT * FROM mv;
 count
-------
     3
(1 row)

-- Delete the materialized view.
gaussdb=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
```

1.1.3 Support and Constraints

Supported Scenarios

- Supports the same query scope as the CREATE TABLE AS statement does.
- Supports index creation in complete-refresh materialized views.
- Supports ANALYZE and EXPLAIN.

Unsupported Scenarios

Materialized views cannot be added, deleted, or modified. They support only query statements.

Constraints

When a complete-refresh materialized view is refreshed or deleted, a high-level lock is added to the base table. If the definition of a materialized view involves multiple tables, pay attention to the service logic to avoid deadlock.

1.2 Fast-Refresh Materialized View

1.2.1 Overview

Fast-refresh materialized views can be incrementally refreshed. You need to manually execute statements to incrementally refresh materialized views in a period of time. The difference between the fast-refresh and complete-refresh materialized views is that the fast-refresh materialized views support only a small number of scenarios. Currently, only base table scanning statements or UNION ALL can be used to create materialized views.

1.2.2 Usage

Syntax

- Create a fast-refresh materialized view.
  ```
  CREATE INCREMENTAL MATERIALIZED VIEW [ view_name ] AS { query_block };
  ```
- Completely refresh a materialized view.
  ```
  REFRESH MATERIALIZED VIEW [ view_name ];
  ```
- Fast refresh a materialized view.
  ```
  REFRESH INCREMENTAL MATERIALIZED VIEW [ view_name ];
  ```
- Drop a materialized view.
  ```
  DROP MATERIALIZED VIEW [ view_name ];
  ```
- Query a materialized view.
  ```
  SELECT * FROM [ view_name ];
  ```

Examples

```
-- Prepare data.
gaussdb=# CREATE TABLE t1(c1 int, c2 int);
gaussdb=# INSERT INTO t1 VALUES(1, 1);
gaussdb=# INSERT INTO t1 VALUES(2, 2);

-- Create a fast-refresh materialized view.
gaussdb=# CREATE INCREMENTAL MATERIALIZED VIEW mv AS SELECT * FROM t1;
CREATE MATERIALIZED VIEW

-- Insert data.
gaussdb=# INSERT INTO t1 VALUES(3, 3);
INSERT 0 1

-- Fast refresh the materialized view.
gaussdb=# REFRESH INCREMENTAL MATERIALIZED VIEW mv;
REFRESH MATERIALIZED VIEW

-- Query the materialized view result.
gaussdb=# SELECT * FROM mv;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
(3 rows)

-- Insert data.
gaussdb=# INSERT INTO t1 VALUES(4, 4);
INSERT 0 1

-- Completely refresh the materialized view.
gaussdb=# REFRESH MATERIALIZED VIEW mv;
REFRESH MATERIALIZED VIEW

-- Query the materialized view result.
gaussdb=# select * from mv;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)

-- Drop the materialized view.
gaussdb=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
```

1.2.3 Support and Constraints

Supported Scenarios

- Supports statements for querying a single table.
- Supports UNION ALL for querying multiple single tables.
- Supports index creation in materialized views.
- Supports the Analyze operation in materialized views.

Unsupported Scenarios

- Multi-table join plans and subquery plans are not supported in materialized views.
- Clauses WITH, GROUP BY, ORDER BY, LIMIT, and WINDOW are not supported. Operators DISTINCT and AGG are not supported. Subqueries except UNION ALL are not supported.
- Except for a few ALTER operations, most DDL operations cannot be performed on base tables in materialized views.
- Materialized views cannot be added, deleted, or modified. They support only query statements.
- Temporary, hash bucket, unlogged, or partitioned tables cannot be used to create materialized views.
- Materialized views cannot be created in nested mode (that is, a materialized view cannot be created in another materialized view).
- Materialized views of the UNLOGGED type are not supported, and the WITH syntax is not supported.

Constraints

- If the materialized view definition is UNION ALL, each subquery needs to use a different base table.
- When a fast-refresh materialized view is created, fully refreshed, or deleted, a high-level lock is added to the base table. If the materialized view is defined as UNION ALL, pay attention to the service logic to avoid deadlock.

2 Partitioned Table

This chapter describes how to perform query optimization and O&M management on stored data in partitioned tables in scenarios with a large amount of data, including semantics, principles, and constraints.

2.1 Large-Capacity Database

2.1.1 Background

With the increasing amount of data to be processed and diversified application scenarios, databases are facing more and more scenarios with large capacity and diversified data. In the past 20 years, the data volume has gradually increased from MB- and GB-level to TB-level. Facing such a large amount of data, the database management system (DBMS) has higher requirements on data query and management. Objectively, the database must support multiple optimization search policies and O&M methods.

In classic algorithms of computer science, people usually use the Divide and Conquer method to solve problems in large-scale scenarios. The basic idea is to divide a complex problem into two or more same or similar problems. These problems are divided into smaller problems until they can be solved directly. The solution of the original problem can be regarded as the combination of the solutions to all small problems. In a large-capacity data scenario, the database provides a Divide and Conquer method, that is, partitioning. The logical database or its components are divided into different independent partitions. Each partition maintains data with similar attributes logically. In this way, the large amount of data is divided, facilitating data management, search, and maintenance.

2.1.2 Table Partitioning

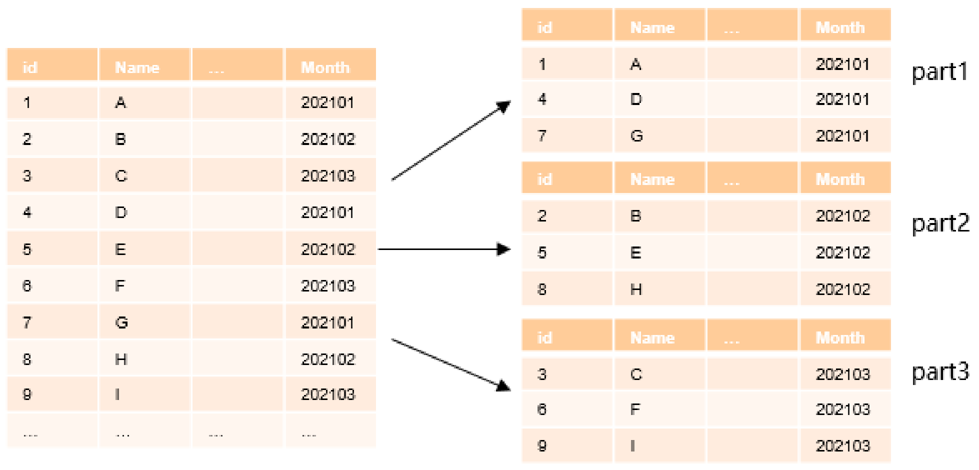
Table partitioning logically divides a large table or index into smaller and easier-to-manage logical units (partitions), minimizing the impact on table query and modification statements. Users can quickly locate a partition where data is located by using a partition key. In this way, users do not need to scan all large tables in the database and can concurrently perform DDL and DML operations on different partitions. Table partitioning provides users with the following capabilities:

1. Improve query efficiency in large-capacity data scenarios: Because data in a table is logically partitioned by partition key, the query result can be implemented by accessing a partition subset instead of the entire table. This partition pruning technique can provide an order of magnitude performance gain.
2. Reduce the impact of parallel O&M and query operations. The mutual impact of parallel DML and DDL statements is reduced, which is obvious in scenarios where a large amount of data is partitioned by time. For example, new data partitions are imported to the database and queried in real time, and old data partitions are cleaned and merged.
3. Provide flexible data O&M management in large-capacity scenarios: Partitioned tables physically isolate data in different partitions at the table file level. Each partition can have independent physical attributes, such as data compression, physical storage settings, and tablespaces. In addition, it supports data management operations, such as data loading, index creation and rebuilding, and partition-level backup and restoration, instead of performing operations on the entire table, reducing operation time.

2.1.3 Data Partition Query Optimization

Partitioned tables help you query data by using predicates based on partition keys. For example, if a table uses month as the partition key and the table structure is designed as an ordinary table, as shown in **Figure 2-1**, you need to access all data in the table (full table scan). If the table is redesigned based on date, the original full table scan is optimized to partition scan. When the table contains a large amount of data and has a long historical period, the performance is greatly improved due to data reduction, as shown in **Figure 2-2**.

Figure 2-1 Example of a partitioned table



id	Name	...	Month
1	A		202101
2	B		202102
3	C		202103
4	D		202101
5	E		202102
6	F		202103
7	G		202101
8	H		202102
9	I		202103
...	...	...	...

part1

id	Name	...	Month
1	A		202101
4	D		202101
7	G		202101

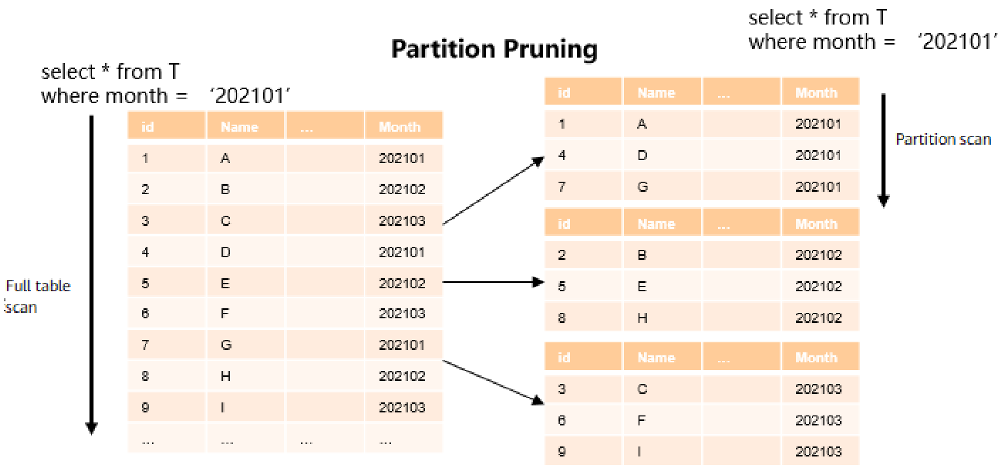
part2

id	Name	...	Month
2	B		202102
5	E		202102
8	H		202102

part3

id	Name	...	Month
3	C		202103
6	F		202103
9	I		202103

Figure 2-2 Example of partition pruning



id	Name	...	Month
1	A		202101
2	B		202102
3	C		202103
4	D		202101
5	E		202102
6	F		202103
7	G		202101
8	H		202102
9	I		202103
...	...	...	...

id	Name	...	Month
1	A		202101
4	D		202101
7	G		202101

id	Name	...	Month
2	B		202102
5	E		202102
8	H		202102

id	Name	...	Month
3	C		202103
6	F		202103
9	I		202103

2.1.4 Data Partition O&M Management

A partitioned table provides flexible support for data lifecycle management (DLM). DLM is a set of processes and policies used to manage data throughout the service life of data. An important component is to determine the most appropriate and cost-effective medium for storing data at any point in the data lifecycle. New data used in daily operations is stored on the fastest and most available storage tier, while old data that is infrequently accessed may be stored on a less costly and inefficient storage tier. Old data may also be updated less frequently, so it makes sense to compress the data and store it as read-only.

Partitioned tables provide an ideal environment for implementing the DLM solution. Different partitions use different tablespaces, maximizing usability and reducing costs in the data lifecycle. The settings are performed by database O&M personnel on the server. Actually, users are unaware of the optimization settings. Logically, users still query the same table. In addition, O&M operations, such as backup, restoration, and index rebuilding, can be performed on different partitions. The Divide and Conquer method is implemented on different subsets of a single data set to meet differentiated requirements of service scenarios.

2.2 Introduction to Partitioned Tables

A partitioned table logically divides table data on a single node based on the partition key and the partition policy related to the partition key. From the perspective of data partitioning, it is a horizontal partitioning policy. Partitioned tables enhance the performance, manageability, and usability of database applications, and help reduce the total cost of ownership (TCO) for storing large amounts of data. Partitioning allows tables, indexes, and index-organized tables to be further divided into smaller parts, enabling these database objects to be managed and accessed at a finer granularity level. GaussDB Kernel provides various partitioning policies and extensions to meet the requirements of different service scenarios. The partitioning policy is implemented inside the database and is transparent to users. Therefore, it enables smooth data migration after the partitioning optimization policy is implemented, without the need to change applications that consume workforce and material resources. This section describes GaussDB Kernel partitioned tables from the following aspects:

1. Basic concepts of partitioned tables: catalog storage and its principle.
2. Partitioning policies: basic partitioning types, and features, optimization, and effects of each partitioning type.

2.2.1 Basic Concepts

2.2.1.1 Partitioned Table

A table that is displayed to users. Users can add, delete, query, and modify data in the table using common DML statements. Generally, it is defined by explicitly using the PARTITION BY statement when DDL statements are used for creating a table. After the table is created, an entry is added to the pg_class table, and the content in the **parttype** column is **'p'** (level-1 partition) or **'s'** (level-2 partition), indicating that the entry is a partitioned table. The partitioned table is usually a logical form, and does not store any data.

Example 1: **t1_hash** is a partitioned table whose partitioning type is hash.

```
gaussdb=# CREATE TABLE t1_hash (c1 INT, c2 INT, c3 INT)
PARTITION BY HASH(c1)
(
    PARTITION p0,
    PARTITION p1,
    PARTITION p2,
    PARTITION p3,
    PARTITION p4,
    PARTITION p5,
    PARTITION p6,
    PARTITION p7,
    PARTITION p8,
    PARTITION p9
);
gaussdb=# \d+ t1_hash
                         Table "public.t1_hash"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 c1     | integer |           | plain   |              |
 c2     | integer |           | plain   |              |
 c3     | integer |           | plain   |              |
Partition By HASH(c1)
Number of partitions: 10 (View pg_partition to check each partition range.)
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE, segment=off

-- Query the partitioning type of table t1_hash.
gaussdb=# SELECT relname, parttype FROM pg_class WHERE relname = 't1_hash';
 relname | parttype
---------+----------
 t1_hash | p
(1 row)

-- Cleanup example
gaussdb=# DROP TABLE t1_hash;
```

Example 2: **t1_sub_rr** is a level-2 partitioned table whose partitioning type is range-list.

```
gaussdb=# CREATE TABLE t1_sub_rr (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY RANGE (c1)
SUBPARTITION BY LIST (c2)
(
    PARTITION p_2021 VALUES LESS THAN (2022) (
        SUBPARTITION p_2021_1 VALUES (1),
        SUBPARTITION p_2021_2 VALUES (2),
        SUBPARTITION p_2021_3 VALUES (3)
    ),
    PARTITION p_2022 VALUES LESS THAN (2023) (
        SUBPARTITION p_2022_1 VALUES (1),
        SUBPARTITION p_2022_2 VALUES (2),
        SUBPARTITION p_2022_3 VALUES (3)
    ),
    PARTITION p_2023 VALUES LESS THAN (2024) (
        SUBPARTITION p_2023_1 VALUES (1),
        SUBPARTITION p_2023_2 VALUES (2),
        SUBPARTITION p_2023_3 VALUES (3)
    ),
    PARTITION p_2024 VALUES LESS THAN (2025) (
        SUBPARTITION p_2024_1 VALUES (1),
        SUBPARTITION p_2024_2 VALUES (2),
        SUBPARTITION p_2024_3 VALUES (3)
    ),
    PARTITION p_2025 VALUES LESS THAN (2026) (
        SUBPARTITION p_2025_1 VALUES (1),
        SUBPARTITION p_2025_2 VALUES (2),
        SUBPARTITION p_2025_3 VALUES (3)
    ),
    PARTITION p_2026 VALUES LESS THAN (2027) (
        SUBPARTITION p_2026_1 VALUES (1),
        SUBPARTITION p_2026_2 VALUES (2),
        SUBPARTITION p_2026_3 VALUES (3)
    )
);

gaussdb=# \d+ t1_sub_rr
                    Table "public.t1_sub_rr"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 c1     | integer |           | plain   |              |
 c2     | integer |           | plain   |              |
 c3     | integer |           | plain   |              |
Partition By RANGE(c1) Subpartition By LIST(c2)
Number of partitions: 6 (View pg_partition to check each partition range.)
Number of subpartitions: 18 (View pg_partition to check each subpartition range.)
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE, segment=off

-- Query the partitioning type of table t1_sub_rr.
gaussdb=# SELECT relname, parttype FROM pg_class WHERE relname = 't1_sub_rr';
  relname  | parttype
-----------+----------
 t1_sub_rr | s
(1 row)

-- Cleanup example
gaussdb=# DROP TABLE t1_sub_rr;
```

2.2.1.2 Partition

A partition stores data actually. The corresponding entry is usually stored in **pg_partition**. The **parentid** of each partition is used as a foreign key to associate with the **oid** column of its partitioned table in the pg_class table.

Example 1: **t1_hash** is a partitioned table.

```
gaussdb=# CREATE TABLE t1_hash (c1 INT, c2 INT, c3 INT)
PARTITION BY HASH(c1)
(
    PARTITION p0,
    PARTITION p1,
    PARTITION p2,
    PARTITION p3,
    PARTITION p4,
    PARTITION p5,
    PARTITION p6,
    PARTITION p7,
    PARTITION p8,
    PARTITION p9
);

-- Query the partitioning type of table t1_hash.
gaussdb=# SELECT oid, relname, parttype FROM pg_class WHERE relname = 't1_hash';
  oid  | relname | parttype
-------+---------+----------
 16685 | t1_hash | p
(1 row)

-- Query the partition information about table t1_hash.
gaussdb=# SELECT oid, relname, parttype, parentid FROM pg_partition WHERE parentid = 16685;
  oid  | relname | parttype | parentid
-------+---------+----------+----------
 16688 | t1_hash | r        |    16685
 16689 | p0      | p        |    16685
 16690 | p1      | p        |    16685
 16691 | p2      | p        |    16685
 16692 | p3      | p        |    16685
 16693 | p4      | p        |    16685
 16694 | p5      | p        |    16685
 16695 | p6      | p        |    16685
 16696 | p7      | p        |    16685
 16697 | p8      | p        |    16685
 16698 | p9      | p        |    16685
(11 rows)

-- Cleanup example: Drop table t1_hash.
gaussdb=# DROP TABLE t1_hash;
```

2.2.1.3 Partition Key

A partition key consists of one or more columns. The partition key value and the corresponding partitioning method can uniquely identify the partition where a tuple is located. Generally, the partition key value is specified by the PARTITION BY clause during table creation.

```
CREATE TABLE table_name (...) PARTITION BY part_strategy (partition_key) (...)
```

NOTICE

Range partitioned tables and list partitioned tables support a partition key with up to 16 columns. Other partitioned tables support a one-column partition key only.

2.2.2 Partitioning Policy

A partitioning policy is specified by the syntax of the PARTITION BY statement when DDL statements are used to create tables. A partitioning policy describes the mapping between data in a partitioned table and partition routes. Common partitioning types include condition-based range/interval partitioning, hash partitioning based on hash functions, and list partitioning based on data enumeration.

```
CREATE TABLE table_name (...) PARTITION BY partition_strategy (partition_key) (...)
```

2.2.2.1 Range Partitioning

Range partitioning maps data to partitions based on the value range of the partition key created for each partition. Range partitioning is the most common partitioning type in production systems and is usually used in scenarios where data is described by date or timestamp. There are two syntax formats for range partitioning. The following is an example:

1. VALUES LESS THAN

   If the VALUE LESS THAN clause is used, a range partitioning policy supports a partition key with up to 16 columns.

   – The following is an example of a single-column partition key:

```
gaussdb=# CREATE TABLE range_sales_single_key
(
    product_id      INT4 NOT NULL,
    customer_id     INT4 NOT NULL,
    time            DATE,
    channel_id      CHAR(1),
    type_id         INT4,
    quantity_sold   NUMERIC(3),
    amount_sold     NUMERIC(10,2)
)
PARTITION BY RANGE (time)
(
    PARTITION date_202001 VALUES LESS THAN ('2020-02-01'),
    PARTITION date_202002 VALUES LESS THAN ('2020-03-01'),
    PARTITION date_202003 VALUES LESS THAN ('2020-04-01'),
    PARTITION date_202004 VALUES LESS THAN ('2020-05-01')
);

-- Cleanup example
gaussdb=# DROP TABLE range_sales_single_key;
```

   date_202002 indicates the partition of February 2020, which contains the data of the partition key from February 1, 2020 to February 29, 2020.

   Each partition has a VALUES LESS clause that specifies the upper limit (excluded) of the partition. Any value greater than or equal to that partition key will be added to the next partition. Except the first partition, all partitions have an implicit lower limit specified by the VALUES LESS clause of the previous partition. You can define the MAXVALUE keyword for the last partition. MAXVALUE represents a virtual infinite value that is prior to any other possible value (including null) of the partition key.

   – The following is an example of a multi-column partition key:

```
gaussdb=# CREATE TABLE range_sales
(
    c1      INT4 NOT NULL,
    c2      INT4 NOT NULL,
    c3      CHAR(1)
)
PARTITION BY RANGE (c1,c2)
(
    PARTITION p1 VALUES LESS THAN (10,10),
    PARTITION p2 VALUES LESS THAN (10,20),
    PARTITION p3 VALUES LESS THAN (20,10)
);
gaussdb=# INSERT INTO range_sales VALUES(9,5,'a');
gaussdb=# INSERT INTO range_sales VALUES(9,20,'a');
gaussdb=# INSERT INTO range_sales VALUES(9,21,'a');
gaussdb=# INSERT INTO range_sales VALUES(10,5,'a');
gaussdb=# INSERT INTO range_sales VALUES(10,15,'a');
gaussdb=# INSERT INTO range_sales VALUES(10,20,'a');
gaussdb=# INSERT INTO range_sales VALUES(10,21,'a');
gaussdb=# INSERT INTO range_sales VALUES(11,5,'a');
```

```
gaussdb=# INSERT INTO range_sales VALUES(11,20,'a');
gaussdb=# INSERT INTO range_sales VALUES(11,21,'a');

gaussdb=# SELECT * FROM range_sales PARTITION (p1);
 c1 | c2 | c3
----+----+----
  9 |  5 | a
  9 | 20 | a
  9 | 21 | a
 10 |  5 | a
(4 rows)

gaussdb=# SELECT * FROM range_sales PARTITION (p2);
 c1 | c2 | c3
----+----+----
 10 | 15 | a
(1 row)

gaussdb=# SELECT * FROM range_sales PARTITION (p3);
 c1 | c2 | c3
----+----+----
 10 | 20 | a
 10 | 21 | a
 11 |  5 | a
 11 | 20 | a
 11 | 21 | a
(5 rows)

-- Cleanup example
gaussdb=# DROP TABLE range_sales;
```

NOTE

The partitioning rules for multi-column partition keys are as follows:

1. The comparison starts from the first column.
2. If the value of the inserted first column is smaller than the boundary value of the current column in the target partition, the values are directly inserted.
3. If the value of the inserted first column is equal to the boundary of the current column in the target partition, compare the value of the inserted second column with the boundary of the next column in the target partition.
4. If the value of the inserted first column is greater than the boundary of the current column in the target partition, compare the value with that in the next partition.

2. START END

   If the START END clause is used, a range partitioning policy supports only a one-column partition key.

   Example:

```
-- Create tablespaces.
gaussdb=# CREATE TABLESPACE startend_tbs1 LOCATION '/home/omm/startend_tbs1';
gaussdb=# CREATE TABLESPACE startend_tbs2 LOCATION '/home/omm/startend_tbs2';
gaussdb=# CREATE TABLESPACE startend_tbs3 LOCATION '/home/omm/startend_tbs3';
gaussdb=# CREATE TABLESPACE startend_tbs4 LOCATION '/home/omm/startend_tbs4';

-- Create a temporary schema.
gaussdb=# CREATE SCHEMA tpcds;
gaussdb=# SET CURRENT_SCHEMA TO tpcds;

-- Create a partitioned table with the partition key of the integer type.
gaussdb=# CREATE TABLE tpcds.startend_pt (c1 INT, c2 INT)
TABLESPACE startend_tbs1
PARTITION BY RANGE (c2) (
    PARTITION p1 START(1) END(1000) EVERY(200) TABLESPACE startend_tbs2,
    PARTITION p2 END(2000),
    PARTITION p3 START(2000) END(2500) TABLESPACE startend_tbs3,
    PARTITION p4 START(2500),
    PARTITION p5 START(3000) END(5000) EVERY(1000) TABLESPACE startend_tbs4
)
ENABLE ROW MOVEMENT;

-- View the information of the partitioned table.
gaussdb=# SELECT relname, boundaries, spcname FROM pg_partition p JOIN pg_tablespace t ON
    p.reltablespace=t.oid and p.parentid='tpcds.startend_pt'::regclass ORDER BY 1;
     relname | boundaries | spcname
-------------+------------+---------------
        p1_0 | {1}        | startend_tbs2
        p1_1 | {201}      | startend_tbs2
        p1_2 | {401}      | startend_tbs2
        p1_3 | {601}      | startend_tbs2
        p1_4 | {801}      | startend_tbs2
        p1_5 | {1000}     | startend_tbs2
          p2 | {2000}     | startend_tbs1
          p3 | {2500}     | startend_tbs3
          p4 | {3000}     | startend_tbs1
        p5_1 | {4000}     | startend_tbs4
        p5_2 | {5000}     | startend_tbs4
 startend_pt |            | startend_tbs1
(12 rows)

-- Cleanup example
gaussdb=# DROP TABLE tpcds.startend_pt;
DROP TABLE
gaussdb=# DROP SCHEMA tpcds;
DROP SCHEMA
```

2.2.2.2 Interval Partitioning

Interval partitioning is an enhancement and extension of range partitioning. When interval partitions are defined, the upper and lower limits do not need to be specified for each new partition. After a partition length is determined, partitions are automatically created and expanded during insertion. At least one range partition must be specified when an interval partition is created. The range partitioning key value determines the high value of the range partitions, which is called the transition point, and the database creates interval partitions for data with values that are beyond that transition point. The lower boundary of every interval partition is the non-inclusive upper boundary of the previous range or interval partition. Example:

```
gaussdb=# CREATE TABLE interval_sales
(
    prod_id       NUMBER(6),
    cust_id       NUMBER,
    time_id       DATE,
    channel_id    CHAR(1),
    promo_id      NUMBER(6),
    quantity_sold NUMBER(3),
    amount_sold   NUMBER(10, 2)
)
PARTITION BY RANGE (time_id) INTERVAL ('1 month')
(
    PARTITION date_2015 VALUES LESS THAN ('2016-01-01'),
    PARTITION date_2016 VALUES LESS THAN ('2017-01-01'),
    PARTITION date_2017 VALUES LESS THAN ('2018-01-01'),
    PARTITION date_2018 VALUES LESS THAN ('2019-01-01'),
    PARTITION date_2019 VALUES LESS THAN ('2020-01-01')
);

-- Cleanup example
gaussdb=# DROP TABLE interval_sales;
```

In the preceding example, partitions are created by year from 2015 to 2019. When data after 2020-01-01 is inserted, a partition is automatically created because the data exceeds the upper boundary of the predefined range partition.

CAUTION

Interval partitioning supports only the date and time types, such as date, time, and timestamp.

2.2.2.3 Hash Partitioning

Hash partitioning uses a hash algorithm to map data to partitions based on partition keys. The GaussDB Kernel built-in hash algorithm is used. When the value range of partition keys has no data skew, the hash algorithm evenly distributes rows among partitions to ensure that the partition sizes are roughly the same. Therefore, hash partitioning is an ideal method for evenly distributing data among partitions. Hash partitioning is also an easy-to-use alternative to range partitioning, especially when the data to be partitioned is not historical data or has no obvious partition key. The following is an example:

```
CREATE TABLE bmsql_order_line (
    ol_w_id         INTEGER   NOT NULL,
    ol_d_id         INTEGER   NOT NULL,
    ol_o_id         INTEGER   NOT NULL,
    ol_number       INTEGER   NOT NULL,
    ol_i_id         INTEGER   NOT NULL,
    ol_delivery_d   TIMESTAMP,
    ol_amount       DECIMAL(6,2),
    ol_supply_w_id  INTEGER,
    ol_quantity     INTEGER,
    ol_dist_info    CHAR(24)
)
-- Define 100 partitions.
PARTITION BY HASH(ol_d_id)
(
    PARTITION p0,
    PARTITION p1,
    PARTITION p2,
    …
    PARTITION p99
);
```

In the preceding example, the **ol_d_id** column in the **bmsql_order_line** table is partitioned. The **ol_d_id** column is an identifier attribute column and does not distinguish time or a specific dimension. Using the hash partitioning policy to divide a table is an ideal choice. Compared with operations of other partitioning types, when creating partitions, you only need to specify the partition key and the number of partitions on the basis that the partition key does not have too much data skew (one or more values are highly repeated). In addition, data in each partition is evenly distributed, improving usability of partitioned tables.

2.2.2.4 List Partitioning

List partitioning can explicitly control how rows are mapped to partitions by specifying a list of discrete values for the partition key in the description for each partition. The advantages of list partitioning are that data can be partitioned by enumerating partition values, and unordered and irrelevant data sets can be grouped and organized. For partition key values that are not defined in the list, you can use the default partition (DEFAULT) to save data. In this way, all rows that are not mapped to any other partition do not generate errors. Example:

```
gaussdb=# CREATE TABLE bmsql_order_line (
    ol_w_id          INTEGER   NOT NULL,
    ol_d_id          INTEGER   NOT NULL,
    ol_o_id          INTEGER   NOT NULL,
    ol_number        INTEGER   NOT NULL,
    ol_i_id          INTEGER   NOT NULL,
    ol_delivery_d    TIMESTAMP,
    ol_amount        DECIMAL(6,2),
    ol_supply_w_id   INTEGER,
    ol_quantity      INTEGER,
    ol_dist_info     CHAR(24)
)
PARTITION BY LIST(ol_d_id)
(
    PARTITION p0 VALUES (1,4,7),
    PARTITION p1 VALUES (2,5,8),
    PARTITION p2 VALUES (3,6,9),
    PARTITION p3 VALUES (DEFAULT)
);

-- Cleanup example
gaussdb=# DROP TABLE bmsql_order_line;
```

The preceding example is similar to that of hash partitioning. The **ol_d_id** column is used for partitioning. However, list partitioning limits a possible range of **ol_d_id** values, and data that is not in the list enters the **p3** partition (DEFAULT). Compared with hash partitioning, list partitioning has better control over partition keys and can accurately store target data in the expected partitions. However, if there are a large number of list values, it is difficult to define partitions. In this case, hash partitioning is recommended. List partitioning and hash partitioning are used to group and organize unordered and irrelevant data sets.

CAUTION

List partitioning supports a partition key with up to 16 columns. For one-column partition keys, the enumerated values in the list cannot be NULL during partition defining. For multi-column partition keys, the enumerated values in the list can be NULL during partition defining.

2.2.2.5 Subpartitioning

Subpartitioning (also referred to as composite partitioning) is a combination of basic data partitioning types. A table is partitioned by one data distribution method and then each partition is further subdivided into new partitions using a second data distribution method. All new partitions of a given partition represent a logical subset of the data. Common types of composite partitioning are as follows:

1. Range-range
2. Range-list
3. Range-hash
4. List-range

5. List-list
6. List-hash
7. Hash-range
8. Hash-list
9. Hash-hash

Example:

```
-- Range-range
gaussdb=# CREATE TABLE t_range_range (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY RANGE (c1)
SUBPARTITION BY RANGE (c2)
(
    PARTITION p1 VALUES LESS THAN (10) (
        SUBPARTITION p1sp1 VALUES LESS THAN (5),
        SUBPARTITION p1sp2 VALUES LESS THAN (10)
    ),
    PARTITION p2 VALUES LESS THAN (20) (
        SUBPARTITION p2sp1 VALUES LESS THAN (15),
        SUBPARTITION p2sp2 VALUES LESS THAN (20)
    )
);
gaussdb=# DROP TABLE t_range_range;

-- Range-list
gaussdb=# CREATE TABLE t_range_list (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY RANGE (c1)
SUBPARTITION BY LIST (c2)
(
    PARTITION p1 VALUES LESS THAN (10) (
        SUBPARTITION p1sp1 VALUES (1, 2),
        SUBPARTITION p1sp2 VALUES (3, 4)
    ),
    PARTITION p2 VALUES LESS THAN (20) (
        SUBPARTITION p2sp1 VALUES (1, 2),
        SUBPARTITION p2sp2 VALUES (3, 4)
    )
);
gaussdb=# DROP TABLE t_range_list;

-- Range-hash
gaussdb=# CREATE TABLE t_range_hash (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY RANGE (c1)
SUBPARTITION BY HASH (c2)
SUBPARTITIONS 2
(
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN (20)
);
gaussdb=# DROP TABLE t_range_hash;

-- List-range
gaussdb=# CREATE TABLE t_list_range (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY LIST (c1)
SUBPARTITION BY RANGE (c2)
(
    PARTITION p1 VALUES (1, 2) (
        SUBPARTITION p1sp1 VALUES LESS THAN (5),
        SUBPARTITION p1sp2 VALUES LESS THAN (10)
    ),
    PARTITION p2 VALUES (3, 4) (
        SUBPARTITION p2sp1 VALUES LESS THAN (5),
        SUBPARTITION p2sp2 VALUES LESS THAN (10)
    )
);
gaussdb=# DROP TABLE t_list_range;

-- List-list
gaussdb=# CREATE TABLE t_list_list (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY LIST (c1)
SUBPARTITION BY LIST (c2)
(
    PARTITION p1 VALUES (1, 2) (
        SUBPARTITION p1sp1 VALUES (1, 2),
        SUBPARTITION p1sp2 VALUES (3, 4)
    ),
    PARTITION p2 VALUES (3, 4) (
        SUBPARTITION p2sp1 VALUES (1, 2),
        SUBPARTITION p2sp2 VALUES (3, 4)
    )
);
gaussdb=# DROP TABLE t_list_list;

-- List-hash
gaussdb=# CREATE TABLE t_list_hash (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY LIST (c1)
SUBPARTITION BY HASH (c2)
SUBPARTITIONS 2
(
    PARTITION p1 VALUES (1, 2),
    PARTITION p2 VALUES (3, 4)
);
gaussdb=# DROP TABLE t_list_hash;

-- Hash-range
gaussdb=# CREATE TABLE t_hash_range (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY HASH (c1)
PARTITIONS 2
SUBPARTITION BY RANGE (c2)
(
    PARTITION p1 (
        SUBPARTITION p1sp1 VALUES LESS THAN (5),
        SUBPARTITION p1sp2 VALUES LESS THAN (10)
    ),
    PARTITION p2 (
        SUBPARTITION p2sp1 VALUES LESS THAN (5),
        SUBPARTITION p2sp2 VALUES LESS THAN (10)
    )
```

```
);
gaussdb=# DROP TABLE t_hash_range;

-- Hash-list
gaussdb=# CREATE TABLE t_hash_list (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY HASH (c1)
PARTITIONS 2
SUBPARTITION BY LIST (c2)
(
    PARTITION p1 (
        SUBPARTITION p1sp1 VALUES (1, 2),
        SUBPARTITION p1sp2 VALUES (3, 4)
    ),
    PARTITION p2 (
        SUBPARTITION p2sp1 VALUES (1, 2),
        SUBPARTITION p2sp2 VALUES (3, 4)
    )
);
gaussdb=# DROP TABLE t_hash_list;

-- Hash-hash
gaussdb=# CREATE TABLE t_hash_hash (
    c1 INT,
    c2 INT,
    c3 INT
)
PARTITION BY HASH (c1)
PARTITIONS 2
SUBPARTITION BY HASH (c2)
SUBPARTITIONS 2
(
    PARTITION p1,
    PARTITION p2
);
gaussdb=# DROP TABLE t_hash_hash;
```

CAUTION

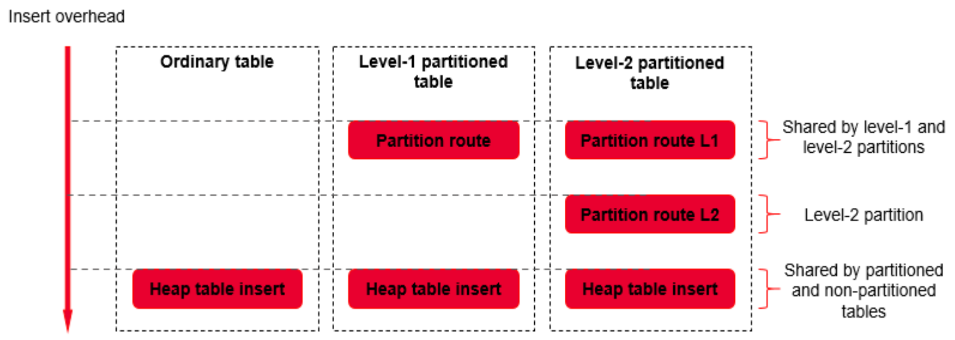
Interval partitioning is a special form of range partitioning. Currently, interval partitioning cannot be defined in subpartitioning.

The partitions and level-2 partitions of a level-2 partitioned table support a one-column partition key only.

2.2.2.6 Impact of Partitioned Tables on Import Performance

In the GaussDB Kernel kernel implementation, compared with a non-partitioned table, a partitioned table has partition routing overheads during data insertion. The overall data insertion overheads include: (1) heap base table insertion and (2) partition routing, as shown in **Figure 2-3**. The heap base table insertion solves the problem of importing tuples to the corresponding heap table and is shared by ordinary tables and partitioned tables. The partition routing solves the problem that the tuple is inserted into the corresponding partRel. In addition, the partition routing algorithm is shared by partitions and level-2 partitions. The difference is that the level-2 partition has one more routing operation than the partition, and calls the routing algorithm twice.

Figure 2-3 Inserting data into ordinary tables and partitioned tables



Therefore, data insertion optimization focuses on the following aspects:

1. Heap base table insertion in a partitioned table:
   a. The operator noise floor is optimized.
   b. Heap data insertion is optimized.
   c. Index insertion build (with indexes) is optimized.
2. Partition routing in a partitioned table:
   a. The logic of the routing search algorithm is optimized.
   b. The routing noise floor is optimized, including enabling the partRel handle of the partitioned table and adding the logic overhead of function calling.

NOTE

The performance of partition routing is reflected by a single INSERT statement involving a large amount of data. In the UPDATE scenario, the system searches for the tuples to be updated, deletes the tuples, and then inserts new tuples. Therefore, the performance is not as good as that of a single INSERT statement.

Table 2-1 shows the routing algorithm logic of different partitioning types.

Table 2-1 Routing algorithm logic

Partitioning Type	Routing Algorithm Complexity	Implementation Description
Range partitioning	O(logN)	Implemented based on binary search
Interval partitioning	O(logN)	Implemented based on binary search
Hash partitioning	O(1)	Implemented based on the key-partOid hash table
List partitioning	O(1)	Implemented based on the key-partOid hash table
List-list partitioning	O(1) + O(1)	Implemented based on a hash table and another hash table

Partitioning Type	Routing Algorithm Complexity	Implementation Description
List-range partitioning	O(1) + O(1) = O(1)	Implemented based on a hash table and binary search
List-hash partitioning	O(1) + O(1) = O(1)	Implemented based on a hash table and another hash table
Range-list partitioning	O(1) + O(1) = O(1)	Implemented based binary search and a hash table
Range-range partitioning	O(1) + O(1) = O(1)	Implemented based on binary search and another binary search
Range-hash partitioning	O(1) + O(1) = O(1)	Implemented based binary search and a hash table
Hash-list partitioning	O(1) + O(1) = O(1)	Implemented based on a hash table and another hash table
Hash-range partitioning	O(1) + O(1) = O(1)	Implemented based on a hash table and binary search
Hash-hash partitioning	O(1) + O(1) = O(1)	Implemented based on a hash table and another hash table

CAUTION

The main processing logic of routing is to calculate the partition where the imported data tuple is located based on the partition key. Compared with a non-partitioned table, this part is an extra overhead. The performance loss caused by this overhead in the final data import is related to the CPU processing capability of the server, table width, and actual disk/memory capacity. Generally, it can be roughly considered that:

- In the x86 server scenario, the import performance of a partitioned table is 10% lower than that of an ordinary table, and the import performance of a level-2 partitioned table is 20% lower than that of an ordinary table.
- In the Arm server scenario, the performance decreases by 20% and 30% respectively. The main reason is that routing is performed in the in-memory computing enhancement scenario. The single-core instruction processing capability of mainstream x86 CPUs is slightly better than that of Arm CPUs.

2.2.3 Basic Usage of Partitions

2.2.3.1 Creating Partitioned Tables

Creating a Partitioned Table

The SQL syntax tree is complex due to the powerful and flexible functions of the SQL language. So do partitioned tables. Creating a partitioned table can be regarded as adding partition attributes to the original non-partitioned table. Therefore, the syntax API of a partitioned table can be regarded to extend the CREATE TABLE statement of a non-partitioned table with a PARTITION BY clause and specify the following three core elements related to the partition:

1. **partType**: describes the partitioning policy of a partitioned table. The options are **RANGE**, **INTERVAL**, **LIST**, and **HASH**.
2. **partKey**: describes the partition key of a partitioned table. Currently, range and list partitioning supports a partition key with up to 16 columns, while interval and hash partitioning supports a one-column partition key only.
3. **partExpr**: describes the specific partitioning type of a partitioned table, that is, the mapping between key values and partitions.

The three elements are reflected in the PARTITION BY clause of the CREATE TABLE statement, for example, **PARTITION BY** *partType* (*partKey*) (*partExpr[,partExpr]*...). Example:

```
CREATE TABLE [ IF NOT EXISTS ] partition_table_name
(
    [ /* Inherited from the CREATE TABLE statement of an ordinary table */
    { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option [...] ] }[, ... ]
    ]
)
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ COMPRESS | NOCOMPRESS ]
[ TABLESPACE tablespace_name ]
/* Range partitioning. If the INTERVAL clause is declared, interval partitioning is used. */
PARTITION BY RANGE (partKey) [ INTERVAL ('interval_expr') [ STORE IN (tablespace_name [, ... ] ) ] ] (
    partition_start_end_item [, ... ]
    partition_less_then_item [, ... ]
)
/* List partitioning */
PARTITION BY LIST (partKey)
(
    PARTITION partition_name VALUES (list_values_clause) [ TABLESPACE tablespace_name [, ... ] ]
...
)
/* Hash partitioning */
PARTITION BY HASH (partKey) (
    PARTITION partition_name [ TABLESPACE tablespace_name [, ... ] ]
...
)
/* Enable or disable row migration for a partitioned table. */
[ { ENABLE | DISABLE } ROW MOVEMENT ];
```

Restrictions

1. Range and list partitioning supports a partition key with up to 16 columns. Interval and hash partitioning supports a one-column partition key only. All subpartitioning types support a one-column partition key only.
2. Interval partitioning supports only partition keys of the time/date data type and interval partitions cannot be created in a level-2 partitioned table.
3. The partition key value cannot be null except for hash partitioning. Otherwise, the DML statement reports an error. The only exception is the MAXVALUE partition defined for a range partitioned table and the DEFAULT partition defined for a list partitioned table.
4. The maximum number of partitions is 1048575, which can meet the requirements of most service scenarios. If the number of partitions increases, the number of files in the system increases, which affects the system

performance. It is recommended that the number of partitions for a single table be less than or equal to 200.

Creating a Level-2 Partitioned Table

The level-2 partitioned table may be considered as an extension of the partitioned table. In the level-2 partitioned table, the partition is a logical table and does not actually store data, and the data is actually stored on the level-2 partition node. The subpartitioning solution is implemented by nesting two partitions. For details about the partitioning solution, see "CREATE TABLE PARTITION." Common subpartitioning solutions include range-range partitioning, range-list partitioning, range-hash partitioning, list-range partitioning, list-list partitioning, list-hash partitioning, hash-range partitioning, hash-list partitioning, and hash-hash partitioning. Currently, subpartitioning is only applicable to row-store tables. The following is an example of creating a level-2 partition:

```
CREATE TABLE [ IF NOT EXISTS ] subpartition_table_name
(
    [ /* Inherited from the CREATE TABLE statement of an ordinary table */
    { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option [...] ] } [, ... ]
    ]
)
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ COMPRESS | NOCOMPRESS ]
[ TABLESPACE tablespace_name ]
/* Level-2 partition definition */
PARTITION BY {RANGE | LIST | HASH} SUBPARTITOIN BY {RANGE | LIST | HASH}
(
    PARTITION partition_name partExpr…  /* Partition */
    (
        SUBPARTITION partition_name partExpr … /* Subpartition */
        SUBPARTITION partition_name partExpr … /* Subpartition */
    ),
    PARTITION partition_name partExpr…  /* Partition */
    (
        SUBPARTITION partition_name partExpr … /* Subpartition */
        SUBPARTITION partition_name partExpr … /* Subpartition */
    ),
    …
)
[ { ENABLE | DISABLE } ROW MOVEMENT ];
```

Restrictions

1. Subpartitioning support a combination of any two of the list, hash, and range partitioning methods.
2. Subpartitioning supports only a single partition key.
3. Subpartitioning does not support interval partitions.
4. Subpartitioning supports a maximum of 1048575 partitions.

Modifying Partition Attributes

You can run the **ALTER TABLE** command similar to that of a non-partitioned table to modify attributes related to partitioned tables and partitions. Common statements for modifying partition attributes are as follows:

1. ADD PARTITION
2. DROP PARTITION

3. TRUNCATE PARTITION
4. SPLIT PARTITION
5. MERGE PARTITION
6. MOVE PARTITION
7. EXCHANGE PARTITION
8. RENAME PARTITION

The preceding statements for modifying partition attributes are extended based on the ALTER TABLE statement of an ordinary table. Most of the statements are used in a similar way. The following is an example of the basic syntax framework for modifying partitioned table attributes:

```
/* Basic ALTER TABLE syntax */
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name )}
action [, ... ];
```

For details about how to use the ALTER TABLE statement, see **Partitioned Table O&M Management** and sections "ALTER TABLE PARTITION and ALTER TABLE SUBPARTITION" in "SQL Reference > SQL Syntax" of *Developer Guide*.

2.2.3.2 DML Statements for Querying Partitioned Tables

Partitioning is implemented in the database kernel. Therefore, DQL/DML statements for partitioned tables are the same as those for non-partitioned tables in syntax.

For ease of use of partitioned tables, GaussDB Kernel allows you to query specified partitions by running **PARTITION** *(partname)* or **PARTITION FOR** *(partvalue)*. For level-2 partitioned tables, you can run **SUBPARTITION** *(subpartname)* or **SUBPARTITION FOR** *(subpartvalue)* to specify a specific level-2 partition. When DQL/DML statements are executed on a specified partition, if the inserted data does not belong to the target partition, an error is reported. If the queried data does not belong to the target partition, the data is skipped.

The DML statements for specifying partitions are as follows:

1. SELECT
2. INSERT
3. UPDATE
4. DELETE
5. UPSERT
6. MERGE INTO

The following is an example of DML statements for specifying partitions:

```
/* Create a level-2 partitioned table list_list_02. */
gaussdb=# CREATE TABLE IF NOT EXISTS list_list_02
(
    id   INT,
    role VARCHAR(100),
    data VARCHAR(100)
)
PARTITION BY LIST (id) SUBPARTITION BY LIST (role)
(
    PARTITION p_list_2 VALUES(0,1,2,3,4,5,6,7,8,9)
    (
        SUBPARTITION p_list_2_1 VALUES ( 0,1,2,3,4,5,6,7,8,9 ),
        SUBPARTITION p_list_2_2 VALUES ( DEFAULT ),
        SUBPARTITION p_list_2_3 VALUES ( 10,11,12,13,14,15,16,17,18,19),
        SUBPARTITION p_list_2_4 VALUES ( 20,21,22,23,24,25,26,27,28,29 ),
        SUBPARTITION p_list_2_5 VALUES ( 30,31,32,33,34,35,36,37,38,39 )
    ),
    PARTITION p_list_3 VALUES(10,11,12,13,14,15,16,17,18,19)
    (
        SUBPARTITION p_list_3_2 VALUES ( DEFAULT )
    ),
    PARTITION p_list_4 VALUES( DEFAULT ),
    PARTITION p_list_5 VALUES(20,21,22,23,24,25,26,27,28,29)
    (
        SUBPARTITION p_list_5_1 VALUES ( 0,1,2,3,4,5,6,7,8,9 ),
        SUBPARTITION p_list_5_2 VALUES ( DEFAULT ),
        SUBPARTITION p_list_5_3 VALUES ( 10,11,12,13,14,15,16,17,18,19),
        SUBPARTITION p_list_5_4 VALUES ( 20,21,22,23,24,25,26,27,28,29 ),
        SUBPARTITION p_list_5_5 VALUES ( 30,31,32,33,34,35,36,37,38,39 )
    ),
    PARTITION p_list_6 VALUES(30,31,32,33,34,35,36,37,38,39),
    PARTITION p_list_7 VALUES(40,41,42,43,44,45,46,47,48,49)
    (
        SUBPARTITION p_list_7_1 VALUES ( DEFAULT )
    )
) ENABLE ROW MOVEMENT;
/* Import data. */
INSERT INTO list_list_02 VALUES(null, 'alice', 'alice data');
INSERT INTO list_list_02 VALUES(2, null, 'bob data');
INSERT INTO list_list_02 VALUES(null, null, 'peter data');

/* Query a specified partition. */
-- Query all data in a partitioned table.
gaussdb=# SELECT * FROM list_list_02 ORDER BY data;
 id | role  |    data
----+-------+------------
    | alice | alice data
2   |       | bob data
    |       | peter data
(3 rows)
-- Query data in the p_list_4 partition.
gaussdb=# SELECT * FROM list_list_02 PARTITION (p_list_4) ORDER BY data;
 id | role  |    data
----+-------+------------
    | alice | alice data
    |       | peter data
(2 rows)
-- Query the data of the level-2 partition corresponding to (100, 100), that is, the level-2 partition
p_list_4_subpartdefault1.
gaussdb=# SELECT * FROM list_list_02 SUBPARTITION FOR(100, 100) ORDER BY data;
 id | role  |    data
----+-------+------------
    | alice | alice data
    |       | peter data
(2 rows)
-- Query data in the p_list_2 partition.
gaussdb=# SELECT * FROM list_list_02 PARTITION (p_list_2) ORDER BY data;
 id | role |   data
----+------+----------
  2 |      | bob data
(1 row)
-- Query the data of the level-2 partition corresponding to (0, 100), that is, the level-2 partition p_list_2_2.
gaussdb=# SELECT * FROM list_list_02 SUBPARTITION FOR (0, 100) ORDER BY data;
 id | role |   data
----+------+----------
  2 |      | bob data
(1 row)

/* Perform INSERT, UPDATE, and DELETE (IUD) operations on the specified partition. */
-- Delete all data from the p_list_5 partition.
gaussdb=# DELETE FROM list_list_02 PARTITION (p_list_5);
```

```
-- Insert data into the specified partition p_list_7_1. An error is reported because the data does not comply
with the partitioning restrictions.
gaussdb=# INSERT INTO list_list_02 SUBPARTITION (p_list_7_1) VALUES(null, 'cherry', 'cherry data');
ERROR:  inserted subpartition key does not map to the table subpartition
-- Update data of a partition to which the partition value 100 belongs.
gaussdb=# UPDATE list_list_02 PARTITION FOR (100) SET id = 1;

--upsert
gaussdb=# INSERT INTO list_list_02 (id, role, data) VALUES (1, 'test', 'testdata') ON DUPLICATE KEY UPDATE
role = VALUES(role), data = VALUES(data);

--merge into
gaussdb=# CREATE TABLE IF NOT EXISTS list_tmp
(
    id   INT,
    role VARCHAR(100),
    data VARCHAR(100)
)
PARTITION BY LIST (id)
(
    PARTITION p_list_2 VALUES(0,1,2,3,4,5,6,7,8,9),
    PARTITION p_list_3 VALUES(10,11,12,13,14,15,16,17,18,19),
    PARTITION p_list_4 VALUES( DEFAULT ),
    PARTITION p_list_5 VALUES(20,21,22,23,24,25,26,27,28,29),
    PARTITION p_list_6 VALUES(30,31,32,33,34,35,36,37,38,39),
    PARTITION p_list_7 VALUES(40,41,42,43,44,45,46,47,48,49)) ENABLE ROW MOVEMENT;

gaussdb=# MERGE INTO list_tmp target
USING list_list_02 source
ON (target.id = source.id)
WHEN MATCHED THEN
  UPDATE SET target.data = source.data,
             target.role = source.role
WHEN NOT MATCHED THEN
  INSERT (id, role, data)
  VALUES (source.id, source.role, source.data);

-- Cleanup example
gaussdb=# DROP TABLE list_tmp;
gaussdb=# DROP TABLE list_list_02;
```

2.3 Partitioned Table Query Optimization

NOTE

In this example, **explain_perf_mode** is set to **normal**.

2.3.1 Partition Pruning

2.3.1.1 Static Partition Pruning

For partitioned table query statements with constants in partition keys in the search criteria, the search criteria contained in operators such as index scan, bitmap index scan, and index-only scan are used as pruning conditions in the optimizer phase to filter partitions. The search criteria must contain at least one partition key. For a partitioned table with a multi-column partition key, the search criteria can contain any column of the partition key.

Static pruning is supported in the following scenarios:

1. Supported partitioning levels: level-1 partition and level-2 partition.

2. Supported partitioning types: range partitioning, interval partitioning, hash partitioning, and list partitioning.
3. Supported expression types: comparison expression (<, <=, =, >=, >), logical expression, and array expression.

CAUTION

- Currently, static pruning does not support subquery expressions.
- Query statements that specify level-1 partitions in level-2 partitioned tables cannot prune the filter conditions of the level-2 partition keys.
- To support partitioned table pruning, the filter condition on the partition key is forcibly converted to the partition key type when the plan is generated. This operation is different from the implicit type conversion rule. As a result, an error may be reported when the same condition is converted on the partition key, and no error is reported for non-partition keys.

- Typical scenarios where static pruning is supported are as follows:
  a. Comparison expressions

```
-- Create a partitioned table.
gaussdb=# CREATE TABLE t1 (c1 int, c2 int)
PARTITION BY RANGE (c1)
(
    PARTITION p1 VALUES LESS THAN(10),
    PARTITION p2 VALUES LESS THAN(20),
    PARTITION p3 VALUES LESS THAN(MAXVALUE)
);

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = 1;
               QUERY PLAN
-----------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 1
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = 1)
         Selected Partitions:  1
(7 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 < 1;
               QUERY PLAN
-----------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 1
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 < 1)
         Selected Partitions:  1
(7 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 > 11;
               QUERY PLAN
-----------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 2
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 > 11)
         Selected Partitions:  2..3
```

```
(7 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 is NULL;
             QUERY PLAN
-----------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 1
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 IS NULL)
         Selected Partitions:  3
(7 rows)
```

b. Logical expressions

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = 1 AND c2 = 2;
                QUERY PLAN
-----------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 1
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: ((t1.c1 = 1) AND (t1.c2 = 2))
         Selected Partitions:  1
(7 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = 1 OR c1 = 2;
                QUERY PLAN
----------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 1
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: ((t1.c1 = 1) OR (t1.c1 = 2))
         Selected Partitions:  1
(7 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE NOT c1 = 1;
             QUERY PLAN
-----------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 3
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 <> 1)
         Selected Partitions:  1..3
(7 rows)
```

c. Array expressions

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 IN (1, 2, 3);
                   QUERY PLAN
-----------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 1
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = ANY ('{1,2,3}'::integer[]))
         Selected Partitions:  1
(7 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = ALL(ARRAY[1,
2, 3]);
                   QUERY PLAN
-----------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 0
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = ALL ('{1,2,3}'::integer[]))
         Selected Partitions:  NONE
(7 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = ANY(ARRAY[1,
2, 3]);
                      QUERY PLAN
-------------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 1
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = ANY ('{1,2,3}'::integer[]))
         Selected Partitions:  1
(7 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 =
SOME(ARRAY[1, 2, 3]);
                      QUERY PLAN
-------------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 1
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = ANY ('{1,2,3}'::integer[]))
         Selected Partitions:  1
(7 rows)
```

- Typical scenarios where static pruning is not supported are as follows:
   a. Subquery expressions

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = ALL(SELECT c2
FROM t1 WHERE c1 > 10);
                         QUERY PLAN
-------------------------------------------------------------
 Partition Iterator
   Output: public.t1.c1, public.t1.c2
   Iterations: 3
   ->  Partitioned Seq Scan on public.t1
         Output: public.t1.c1, public.t1.c2
         Filter: (SubPlan 1)
         Selected Partitions:  1..3
(7 rows)

-- Cleanup example
gaussdb=# DROP TABLE t1;
```

2.3.1.2 Dynamic Partition Pruning

If a partitioned table query statement with variables exists in the search criteria, the optimizer cannot obtain the bound parameters of the user. Therefore, only the search criteria of operators such as index scan, bitmap index scan, and index-only scan can be parsed in the optimizer phase. After the bound parameters are obtained in the executor phase, the partition filtering is complete. The search criteria must contain at least one partition key. For a partitioned table with a multi-column partition key, the search criteria can contain any column of the partition key. Currently, dynamic partition pruning supports only the parse-bind-execute (PBE) and parameterized path scenarios.

2.3.1.2.1 Dynamic PBE Pruning

Dynamic PBE pruning is supported in the following scenarios:

1. Supported partitioning levels: level-1 partition and level-2 partition
2. Supported partitioning types: range partitioning, interval partitioning, hash partitioning, and list partitioning.
3. Supported expression types: comparison expression (<, <=, =, >=, >), logical expression, and array expression.
4. Supported conversions and functions: some implicit type conversions and the IMMUTABLE function.

CAUTION

- Dynamic PBE pruning supports expressions, implicit conversions, the IMMUTABLE function, and the STABLE function, but does not support subquery expressions or VOLATILE function. For the STABLE function, type conversion functions such as to_timestamp may be affected by GUC parameters and lead to different pruning results. To ensure performance optimization, you can analyze table to regenerate a Gplan.
- Dynamic PBE pruning is based on the generic plan. Therefore, when determining whether a statement can be dynamically pruned, you need to set **plan_cache_mode** to **'force_generic_plan'** to eliminate the interference of the custom plan.
- Query statements that specify level-1 partitions in level-2 partitioned tables cannot prune the filter conditions of the level-2 partition keys.

- Typical scenarios where dynamic PBE pruning is supported are as follows:
  a. Comparison expressions

```
-- Create a partitioned table.
gaussdb=# CREATE TABLE t1 (c1 int, c2 int)
PARTITION BY RANGE (c1)
(
    PARTITION p1 VALUES LESS THAN(10),
    PARTITION p2 VALUES LESS THAN(20),
    PARTITION p3 VALUES LESS THAN(MAXVALUE)
);
-- Set parameters.
gaussdb=# set plan_cache_mode = 'force_generic_plan';

gaussdb=# PREPARE p1(int) AS SELECT * FROM t1 WHERE c1 = $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p1(1);
              QUERY PLAN
---------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = $1)
         Selected Partitions:  1 (pbe-pruning)
(7 rows)

gaussdb=# PREPARE p2(int) AS SELECT * FROM t1 WHERE c1 < $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p2(1);
```

```
               QUERY PLAN
-----------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 < $1)
         Selected Partitions:  1 (pbe-pruning)
(7 rows)

gaussdb=# PREPARE p3(int) AS SELECT * FROM t1 WHERE c1 > $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p3(1);
               QUERY PLAN
-----------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 > $1)
         Selected Partitions:  1..3 (pbe-pruning)
(7 rows)
```

b. Logical expressions

```
gaussdb=# PREPARE p5(INT, INT) AS SELECT * FROM t1 WHERE c1 = $1 AND c2 = $2;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p5(1, 2);
                   QUERY PLAN
-------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: ((t1.c1 = $1) AND (t1.c2 = $2))
         Selected Partitions:  1 (pbe-pruning)
(7 rows)

gaussdb=# PREPARE p6(INT, INT) AS SELECT * FROM t1 WHERE c1 = $1 OR c2 = $2;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p6(1, 2);
                   QUERY PLAN
------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: ((t1.c1 = $1) OR (t1.c2 = $2))
         Selected Partitions:  1..3 (pbe-pruning)
(7 rows)
gaussdb=# PREPARE p7(INT) AS SELECT * FROM t1 WHERE NOT c1 = $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) execute p7(1);
               QUERY PLAN
-----------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 <> $1)
         Selected Partitions:  1..3 (pbe-pruning)
(7 rows)
```

c. Array expressions

```
gaussdb=# PREPARE p8(INT, INT, INT) AS SELECT * FROM t1 WHERE c1 IN ($1, $2, $3);
PREPARE
```

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p8(1, 2, 3);
                QUERY PLAN
---------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = ANY (ARRAY[$1, $2, $3]))
         Selected Partitions:  1 (pbe-pruning)
(7 rows)
gaussdb=# PREPARE p9(INT, INT, INT) AS SELECT * FROM t1 WHERE c1 NOT IN ($1, $2, $3);
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p9(1, 2, 3);
                 QUERY PLAN
----------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 <> ALL (ARRAY[$1, $2, $3]))
         Selected Partitions:  1..3 (pbe-pruning)
(7 rows)
gaussdb=# PREPARE p10(INT, INT, INT) AS SELECT * FROM t1 WHERE c1 = ALL(ARRAY[$1, $2,
$3]);
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p10(1, 2, 3);
                QUERY PLAN
---------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = ALL (ARRAY[$1, $2, $3]))
         Selected Partitions:  NONE (pbe-pruning)
(7 rows)
gaussdb=# PREPARE p11(INT, INT, INT) AS SELECT * FROM t1 WHERE c1 = ANY(ARRAY[$1, $2,
$3]);
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p11(1, 2, 3);
                QUERY PLAN
---------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = ANY (ARRAY[$1, $2, $3]))
         Selected Partitions:  1 (pbe-pruning)
(7 rows)
gaussdb=# PREPARE p12(INT, INT, INT) AS SELECT * FROM t1 WHERE c1 = SOME(ARRAY[$1,
$2, $3]);
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p12(1, 2, 3);
                QUERY PLAN
---------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = ANY (ARRAY[$1, $2, $3]))
         Selected Partitions:  1 (pbe-pruning)
(7 rows)
```

d. Implicit type conversion

```
gaussdb=# set plan_cache_mode = 'force_generic_plan';
gaussdb=# PREPARE p13(TEXT) AS SELECT * FROM t1 WHERE c1 = $1;
```

```
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p13('12');
                  QUERY PLAN
--------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = ($1)::bigint)
         Selected Partitions:  2 (pbe-pruning)
(7 rows)
```

e. IMMUTABLE function

```
gaussdb=# PREPARE p14(TEXT) AS SELECT * FROM t1 WHERE c1 = LENGTHB($1);
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p14('hello');
                  QUERY PLAN
--------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: PART
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: (t1.c1 = lengthb($1))
         Selected Partitions:  1 (pbe-pruning)
(7 rows)
```

- Typical scenarios where dynamic PBE pruning is not supported are as follows:

a. Subquery expressions

```
gaussdb=# PREPARE p15(INT) AS SELECT * FROM t1 WHERE c1 = ALL(SELECT c2 FROM t1
WHERE c1 > $1);
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p15(1);
                       QUERY PLAN
-------------------------------------------------------------
 Partition Iterator
   Output: public.t1.c1, public.t1.c2
   Iterations: 3
   ->  Partitioned Seq Scan on public.t1
         Output: public.t1.c1, public.t1.c2
         Filter: (SubPlan 1)
         Selected Partitions:  1..3
(7 rows)
```

b. Implicit type conversion failure

```
gaussdb=# PREPARE p16(name) AS SELECT * FROM t1 WHERE c1 = $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p16('12');
                 QUERY PLAN
---------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 3
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: ((t1.c1)::text = ($1)::text)
         Selected Partitions:  1..3
(7 rows)
```

c. STABLE and VOLATILE functions

```
gaussdb=# create sequence seq;
gaussdb=# PREPARE p17(TEXT) AS SELECT * FROM t1 WHERE c1 = currval($1);-- The VOLATILE
function does not support pruning.
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p17('seq');
                        QUERY PLAN
-------------------------------------------------------------
 Partition Iterator
   Output: c1, c2
   Iterations: 3
   ->  Partitioned Seq Scan on public.t1
         Output: c1, c2
         Filter: ((t1.c1)::numeric = currval(($1)::regclass))
         Selected Partitions:  1..3
(7 rows)

-- Cleanup example
gaussdb=# DROP TABLE t1;
```

2.3.1.2.2 Dynamic Parameterized Path Pruning

Dynamic parameterized path pruning is supported in the following scenarios:

1. Supported partitioning levels: level-1 partition and level-2 partition
2. Supported partitioning types: range partitioning, interval partitioning, hash partitioning, and list partitioning.
3. Supported operator types: index scan, index-only scan, and bitmap scan.
4. Supported expression types: comparison expression (<, <=, =, >=, >) and logical expression.

CAUTION

Dynamic parameterized path pruning does not support subquery expressions, STABLE and VOLATILE functions, cross-QueryBlock parameterized paths, BitmapOr operator, or BitmapAnd operator.

- Typical scenarios where dynamic parameterized path pruning is supported are as follows:

  a. Comparison expressions

```
-- Create partitioned tables and indexes.
gaussdb=# CREATE TABLE t1 (c1 INT, c2 INT)
PARTITION BY RANGE (c1)
(
    PARTITION p1 VALUES LESS THAN(10),
    PARTITION p2 VALUES LESS THAN(20),
    PARTITION p3 VALUES LESS THAN(MAXVALUE)
);
gaussdb=# CREATE TABLE t2 (c1 INT, c2 INT)
PARTITION BY RANGE (c1)
(
    PARTITION p1 VALUES LESS THAN(10),
    PARTITION p2 VALUES LESS THAN(20),
    PARTITION p3 VALUES LESS THAN(MAXVALUE)
);
gaussdb=# CREATE INDEX t1_c1 ON t1(c1) LOCAL;
gaussdb=# CREATE INDEX t2_c1 ON t2(c1) LOCAL;
gaussdb=# CREATE INDEX t1_c2 ON t1(c2) LOCAL;
gaussdb=# CREATE INDEX t2_c2 ON t2(c2) LOCAL;

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t2 JOIN t1 ON t1.c1 = t2.c2;
                     QUERY PLAN
--------------------------------------------------------------
 Hash Join
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   Hash Cond: (t2.c2 = t1.c1)
   ->  Partition Iterator
         Output: t2.c1, t2.c2
         Iterations: 3
         ->  Partitioned Seq Scan on public.t2
               Output: t2.c1, t2.c2
               Selected Partitions:  1..3
   ->  Hash
         Output: t1.c1, t1.c2
         ->  Partition Iterator
               Output: t1.c1, t1.c2
               Iterations: 3
               ->  Partitioned Seq Scan on public.t1
                     Output: t1.c1, t1.c2
                     Selected Partitions:  1..3
(17 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t2 JOIN t1 ON t1.c1 < t2.c2;
                        QUERY PLAN
-------------------------------------------------------------
 Nested Loop
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   ->  Partition Iterator
         Output: t2.c1, t2.c2
         Iterations: 3
         ->  Partitioned Seq Scan on public.t2
               Output: t2.c1, t2.c2
               Selected Partitions:  1..3
   ->  Partition Iterator
         Output: t1.c1, t1.c2
         Iterations: PART
         ->  Partitioned Index Scan using t2_c1 on public.t1
               Output: t1.c1, t1.c2
               Index Cond: (t1.c1 < t2.c2)
               Selected Partitions:  1..3 (ppi-pruning)
(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t2 JOIN t1 ON t1.c1 > t2.c2;
                        QUERY PLAN
-------------------------------------------------------------
 Nested Loop
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   ->  Partition Iterator
         Output: t2.c1, t2.c2
         Iterations: 3
         ->  Partitioned Seq Scan on public.t2
               Output: t2.c1, t2.c2
               Selected Partitions:  1..3
   ->  Partition Iterator
         Output: t1.c1, t1.c2
         Iterations: PART
         ->  Partitioned Index Scan using t2_c1 on public.t1
               Output: t1.c1, t1.c2
               Index Cond: (t1.c1 > t2.c2)
               Selected Partitions:  1..3 (ppi-pruning)
(15 rows)
```

b. Logical expressions

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t2 JOIN t1 ON t1.c1 = t2.c2 
AND t1.c2 = 2;
                        QUERY PLAN
-------------------------------------------------------------
 Hash Join
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   Hash Cond: (t2.c2 = t1.c1)
   ->  Partition Iterator
         Output: t2.c1, t2.c2
         Iterations: 3
         ->  Partitioned Seq Scan on public.t2
               Output: t2.c1, t2.c2
               Selected Partitions:  1..3
   ->  Hash
         Output: t1.c1, t1.c2
         ->  Partition Iterator
               Output: t1.c1, t1.c2
               Iterations: 3
               ->  Partitioned Bitmap Heap Scan on public.t1
                     Output: t1.c1, t1.c2
                     Recheck Cond: (t1.c2 = 2)
                     Selected Partitions:  1..3
                     ->  Partitioned Bitmap Index Scan on t1_c2
                           Index Cond: (t1.c2 = 2)
(20 rows)
```

- Typical scenarios where dynamic parameterized path pruning is not supported are as follows:

  a. BitmapOr and BitmapAnd operators

```
gaussdb=# SET enable_seqscan=off;
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t2 JOIN t1 ON t1.c1 = t2.c2 OR
t1.c1 = 2;
                           QUERY PLAN
-----------------------------------------------------------------
 Nested Loop
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   ->  Partition Iterator
         Output: t2.c1, t2.c2
         Iterations: 3
         ->  Partitioned Seq Scan on public.t2
               Output: t2.c1, t2.c2
               Selected Partitions:  1..3
   ->  Partition Iterator
         Output: t1.c1, t1.c2
         Iterations: 3
         ->  Partitioned Bitmap Heap Scan on public.t1
               Output: t1.c1, t1.c2
               Recheck Cond: ((t1.c1 = t2.c2) OR (t1.c1 = 2))
               Selected Partitions:  1..3
               ->  BitmapOr
                     ->  Partitioned Bitmap Index Scan on t1_c1
                           Index Cond: (t1.c1 = t2.c2)
                     ->  Partitioned Bitmap Index Scan on t1_c1
                           Index Cond: (t1.c1 = 2)
(20 rows)
```

  b. Implicit conversion

```
gaussdb=# CREATE TABLE t3(c1 TEXT, c2 INT);
CREATE TABLE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 JOIN t3 ON t1.c1 = t3.c1;
                         QUERY PLAN
-------------------------------------------------------------
 Nested Loop
   Output: t1.c1, t1.c2, t3.c1, t3.c2
   ->  Seq Scan on public.t3
         Output: t3.c1, t3.c2
   ->  Partition Iterator
         Output: t1.c1, t1.c2
         Iterations: 3
         ->  Partitioned Index Scan using t1_c1 on public.t1
               Output: t1.c1, t1.c2
               Index Cond: (t1.c1 = (t3.c1)::bigint)
               Selected Partitions:  1..3
(11 rows)
```

  c. Functions

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 JOIN t3 ON t1.c1 =
LENGTHB(t3.c1);
                         QUERY PLAN
-------------------------------------------------------------
 Nested Loop
   Output: t1.c1, t1.c2, t3.c1, t3.c2
   ->  Seq Scan on public.t3
         Output: t3.c1, t3.c2
   ->  Partition Iterator
         Output: t1.c1, t1.c2
         Iterations: 3
         ->  Partitioned Index Scan using t1_c1 on public.t1
```

```
              Output: t1.c1, t1.c2
              Index Cond: (t1.c1 = lengthb(t3.c1))
              Selected Partitions:  1..3
(11 rows)

-- Cleanup example
gaussdb=# DROP TABLE t1;
gaussdb=# DROP TABLE t2;
gaussdb=# DROP TABLE t3;
```

2.3.2 Optimizing Partition Operator Execution

2.3.2.1 Elimination of the Partition Iterator Operator

Scenario

In the current partitioned table architecture, the executor iteratively accesses each partition by using the Partition Iterator (PI) operator. When the partition pruning result has only one partition, the PI operator has lost its function as an iterator. In this case, eliminating the PI operator can avoid some unnecessary overheads during execution. Due to the PIPELINE architecture of the executor, the PI operator is executed repeatedly. In scenarios with a large amount of data, the benefits of eliminating the PI operator are considerable.

Example

The PI elimination takes effect only after the GUC parameter **partition_iterator_elimination** is enabled. The following is an example:

```
gaussdb=# CREATE TABLE test_range_pt (a INT, b INT, c INT)
PARTITION BY RANGE (a)
(
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN (3000),
    PARTITION p3 VALUES LESS THAN (4000),
    PARTITION p4 VALUES LESS THAN (5000),
    PARTITION p5 VALUES LESS THAN (MAXVALUE)
)ENABLE ROW MOVEMENT;

gaussdb=# EXPLAIN SELECT * FROM test_range_pt WHERE a = 3000;
                                  QUERY PLAN
-----------------------------------------------------------------------------------
 Partition Iterator  (cost=0.00..25.31 rows=10 width=12)
   Iterations: 1
   ->  Partitioned Seq Scan on test_range_pt  (cost=0.00..25.31 rows=10 width=12)
         Filter: (a = 3000)
         Selected Partitions:  3
(5 rows)

gaussdb=# SET partition_iterator_elimination = on;
SET
gaussdb=# EXPLAIN SELECT * FROM test_range_pt WHERE a = 3000;
                               QUERY PLAN
-----------------------------------------------------------------------------
 Partitioned Seq Scan on test_range_pt  (cost=0.00..25.31 rows=10 width=12)
   Filter: (a = 3000)
   Selected Partitions:  3
(3 rows)

-- Cleanup example
gaussdb=# DROP TABLE test_range_pt;
```

Precautions and Constraints

1. The optimization in the target scenario takes effect only when the GUC parameter **partition_iterator_elimination** is enabled and the optimizer pruning result contains only one partition.
2. The PI operator does not support level-2 partitioned tables.
3. Cplan and some Gplan scenarios are supported, for example, the partition key **a = $1** (that is, the scenario where data can be pruned to one partition in the optimizer phase).
4. The SeqScan, Indexscan, Indexonlyscan, Bitmapscan, RowToVec, and Tidscan operators are supported.
5. Row store, Astore, Ustore, and SQLBypass are supported.

2.3.2.2 Merge Append

Scenario

To globally sort a partitioned table, the SQL engine uses the PI operator and PartitionScan to perform a full scan on the partitioned table before sorting. In this case, it is difficult to perform global sorting based on the data partition algorithm. If the **ORDER BY** *column* contains indexes, the existing order cannot be used. To solve this problem, partitioned tables support Merge Append to improve the sorting mechanism.

Example

The following is an example of executing MergeAppend.

```
gaussdb=# CREATE TABLE test_range_pt (a INT, b INT, c INT)
PARTITION BY RANGE(a)
(
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN (3000),
    PARTITION p3 VALUES LESS THAN (4000),
    PARTITION p4 VALUES LESS THAN (5000),
    PARTITION p5 VALUES LESS THAN (MAXVALUE)
)ENABLE ROW MOVEMENT;
gaussdb=# INSERT INTO test_range_pt VALUES
(generate_series(1,10000),generate_series(1,10000),generate_series(1,10000));
gaussdb=# CREATE INDEX idx_range_b ON test_range_pt(b) LOCAL;
gaussdb=# ANALYZE test_range_pt;

gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_range_pt WHERE b >10 AND b < 5000 ORDER BY b
LIMIT 10;
                                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
--------------------------------
 Limit  (cost=0.06..1.02 rows=10 width=12) (actual time=0.990..1.041 rows=10 loops=1)
   ->  Result  (cost=0.06..480.32 rows=10 width=12) (actual time=0.988..1.036 rows=10 loops=1)
         ->  Merge Append  (cost=0.06..480.32 rows=10 width=12) (actual time=0.985..1.026 rows=10 loops=1)
               Sort Key: b
               ->  Partitioned Index Scan using idx_range_b on test_range_pt  (cost=0.00..44.61 rows=998
width=12) (actual time=0.256..0.284 rows=10 loops=1)
                     Index Cond: ((b > 10) AND (b < 5000))
                     Selected Partitions:  1
               ->  Partitioned Index Scan using idx_range_b on test_range_pt  (cost=0.00..44.61 rows=998
width=12) (actual time=0.208..0.208 rows=1 loops=1)
                     Index Cond: ((b > 10) AND (b < 5000))
                     Selected Partitions:  2
               ->  Partitioned Index Scan using idx_range_b on test_range_pt  (cost=0.00..44.61 rows=998
width=12) (actual time=0.205..0.205 rows=1 loops=1)
                     Index Cond: ((b > 10) AND (b < 5000))
                     Selected Partitions:  3
               ->  Partitioned Index Scan using idx_range_b on test_range_pt  (cost=0.00..44.61 rows=998 
width=12) (actual time=0.212..0.212 rows=1 loops=1)
                     Index Cond: ((b > 10) AND (b < 5000))
                     Selected Partitions:  4
               ->  Partitioned Index Scan using idx_range_b on test_range_pt  (cost=0.00..44.61 rows=998 
width=12) (actual time=0.092..0.092 rows=0 loops=1)
                     Index Cond: ((b > 10) AND (b < 5000))
                     Selected Partitions:  5
 Total runtime: 1.656 ms
(20 rows)

-- Disable the MergeAppend operator of a partitioned table.
gaussdb=# SET sql_beta_feature = 'disable_merge_append_partition';
SET
gaussdb=# EXPLAIN ANALYZE SELECT * FROM test_range_pt WHERE b >10 AND b < 5000 ORDER BY b 
LIMIT 10;
                                                                   QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
------------------
 Limit  (cost=296.85..296.88 rows=10 width=12) (actual time=33.559..33.565 rows=10 loops=1)
   ->  Sort  (cost=296.85..309.33 rows=10 width=12) (actual time=33.555..33.557 rows=10 loops=1)
         Sort Key: b
         Sort Method: top-N heapsort  Memory: 26kB
         ->  Partition Iterator  (cost=0.00..189.00 rows=4991 width=12) (actual time=0.352..27.176 rows=4989 
loops=1)
               Iterations: 5
               ->  Partitioned Seq Scan on test_range_pt  (cost=0.00..189.00 rows=4991 width=12) (actual 
time=16.874..25.637 rows=4989 loops=5)
                     Filter: ((b > 10) AND (b < 5000))
                     Rows Removed by Filter: 5011
                     Selected Partitions:  1..5
 Total runtime: 33.877 ms
(11 rows)

-- Cleanup example
gaussdb=# DROP TABLE test_range_pt;
```

Executing MergeAppend consumes much less resources than the common execution mode.

Precautions and Constraints

1. MergeAppend can be executed only when the partition scan path is index or index-only.
2. MergeAppend can be executed only when the partition pruning result is greater than 1.
3. MergeAppend can be executed only when all partitioned indexes are valid and are B-tree indexes.
4. MergeAppend can be executed only when the SQL statement contains the LIMIT clause.
5. MergeAppend cannot be executed when a filter exists during partition scan.
6. The MergeAppend path is no longer generated when the GUC parameter **sql_beta_feature** is set to **'disable_merge_append_partition'**.

2.3.2.3 Max/Min

Scenario

When the min/max function is used for a partitioned table, the SQL engine uses PI and PartitionScan to perform a full scan on the partitioned table and then performs the Sort and Limit operations. If index scan is used to scan the partition, you can perform the Limit operation on each partition to calculate the min/max value, and then perform the Sort and Limit operations on the partitioned table. In this way, when the partitioned table is sorted, the amount of data to be sorted is the same as the number of partitions because the max/min values have been calculated for each partition, so that the sorting overhead is greatly reduced.

Example

The following is an example of executing the max/min function on a partitioned table.

```
gaussdb=# CREATE TABLE test_range_pt (a INT, b INT, c INT)
PARTITION BY RANGE(a)
(
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN (3000),
    PARTITION p3 VALUES LESS THAN (4000),
    PARTITION p4 VALUES LESS THAN (5000),
    PARTITION p5 VALUES LESS THAN (MAXVALUE)
)ENABLE ROW MOVEMENT;
gaussdb=# CREATE INDEX idx_range_b ON test_range_pt(b) LOCAL;
gaussdb=# INSERT INTO test_range_pt VALUES(generate_series(1,10000), generate_series(1,10000),
generate_series(1,10000));
```

Before optimization:

```
gaussdb=# explain  analyze select min(b) from test_range_pt;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
-----------
Aggregate  (cost=164.00..164.01 rows=1 width=8) (actual time=6.779..6.780 rows=1 loops=1)
  ->  Partition Iterator  (cost=0.00..139.00 rows=10000 width=4) (actual time=0.099..4.588 rows=10000
loops=1)
      Iterations: 5
      ->  Partitioned Seq Scan on test_range_pt  (cost=0.00..139.00 rows=10000 width=4) (actual
time=0.326..3.516 rows=10000 loops=5)
          Selected Partitions:  1..5
 Total runtime: 6.942 ms
(6 rows)
```

After optimization:

```
gaussdb=# explain  analyze select min(b) from test_range_pt;
                                                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------
 Result  (cost=441.25..441.26 rows=1 width=0) (actual time=0.554..0.555 rows=1 loops=1)
   InitPlan 1 (returns $2)
     ->  Limit  (cost=441.25..441.25 rows=1 width=4) (actual time=0.547..0.547 rows=1 loops=1)
           ->  Sort  (cost=441.25..466.25 rows=1 width=4) (actual time=0.544..0.544 rows=1 loops=1)
                 Sort Key: public.test_range_pt.b
                 Sort Method: top-N heapsort  Memory: 25kB
                 ->  Partition Iterator  (cost=0.00..391.25 rows=10000 width=4) (actual time=0.135..0.502 rows=5
loops=1)
                       Iterations: 5
                       ->  Limit  (cost=0.00..0.04 rows=1 width=4) (actual time=0.322..0.322 rows=5 loops=5)
                             ->  Partitioned Index Only Scan using idx_range_b on test_range_pt  (cost=0.00..391.25
rows=1 width=4) (actual time=0.319..0.319 rows=5 loops=5)
                                   Index Cond: (b IS NOT NULL)
                                   Heap Fetches: 5
```

```
                                  Selected Partitions:  1..5
 Total runtime: 0.838 ms
(14 rows)
```

The time consumed after the optimization is much shorter than that before the optimization.

```
-- Cleanup example
gaussdb=# DROP TABLE test_range_pt;
```

Precautions and Constraints

1. The max/min function is supported only when the partition scan path is index or index only.
2. The max/min function is supported only when all partitioned indexes are valid and are B-tree indexes.

2.3.2.4 Optimizing Performance of Importing Data to Partitions

Scenario

When data is inserted into a partitioned table, if the inserted data is of simple types such as constants, parameters, and expressions, the INSERT operator is automatically optimized (FastPath). You can determine whether the operator optimization is triggered based on the execution plan. When the operator optimization is triggered, the keyword **FastPath** is added before the INSERT plan.

Example

```
gaussdb=# CREATE TABLE fastpath_t1
(
    col1 int,
    col2 text
)
PARTITION BY RANGE(col1)
(
    PARTITION p1 VALUES LESS THAN(10),
    PARTITION p2 VALUES LESS THAN(MAXVALUE)
);

-- Insert a constant and execute FastPath.
gaussdb=# EXPLAIN INSERT into fastpath_t1 values (0, 'test_insert');
                           QUERY PLAN
-----------------------------------------------------------------
 FastPath Insert on fastpath_t1  (cost=0.00..0.01 rows=1 width=0)
   ->  Result  (cost=0.00..0.01 rows=1 width=0)
(2 rows)

-- Insert an expression with parameters or a simple expression and execute FastPath.
gaussdb=# prepare insert_t1 as insert into fastpath_t1 values($1 + 1 + $2, $2);
PREPARE
gaussdb=# explain execute insert_t1(10, '0');
                           QUERY PLAN
-----------------------------------------------------------------
 FastPath Insert on fastpath_t1  (cost=0.00..0.02 rows=1 width=0)
   ->  Result  (cost=0.00..0.02 rows=1 width=0)
(2 rows)

-- Insert a subquery. FastPath cannot be executed. The standard executor is used.
gaussdb=# create table test_1(col1 int, col3 text);
gaussdb=# explain insert into fastpath_t1 select * from test_1;
                          QUERY PLAN
-----------------------------------------------------------------
```

```
 Insert on fastpath_t1  (cost=0.00..22.38 rows=1238 width=36)
   ->  Seq Scan on test_1  (cost=0.00..22.38 rows=1238 width=36)
(2 rows)

-- Cleanup example
gaussdb=# DROP TABLE fastpath_t1;
gaussdb=# DROP TABLE test_1;
```

Precautions and Constraints

1. FastPath can only be executed under the INSERT VALUES statement, and the data following the VALUES clause must be of the constant, parameter, or expression type.
2. FastPath can only be executed for row-store tables.
3. FastPath does not support triggers.
4. FastPath cannot be executed under the UPSERT statement.
5. The performance can be better improved in the case of CPU resource bottleneck.

2.3.3 Partitioned Indexes

There are three types of indexes on a partitioned table:

1. Global non-partitioned index
2. Global partitioned index
3. Local partitioned index

Currently, GaussDB Kernel supports the global non-partitioned index and local partitioned index.

Figure 2-4 Global non-partitioned index

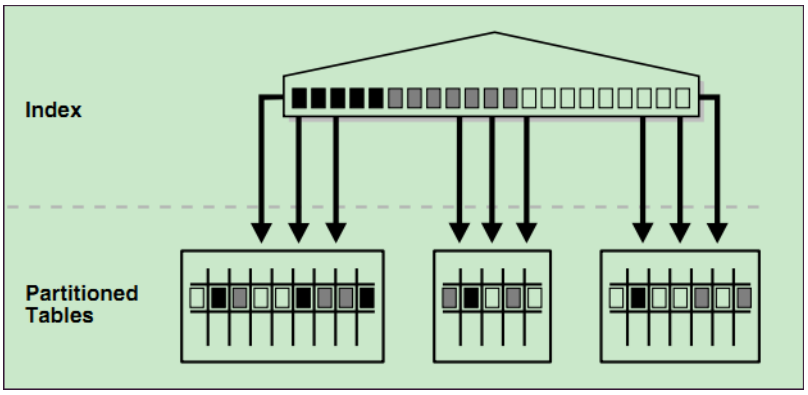


Figure 2-5 Global partitioned index

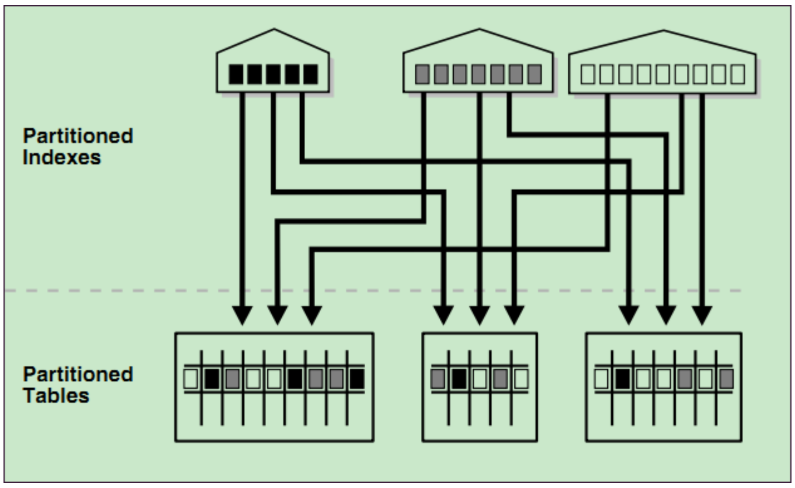
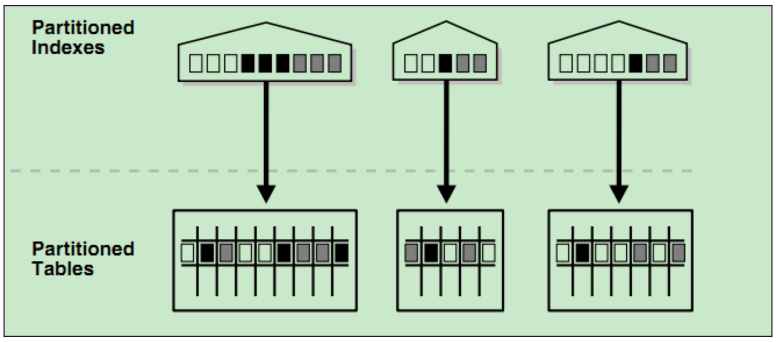


Figure 2-6 Local partitioned index



Constraints

- Partitioned indexes are classified into local indexes and global indexes. A local index binds to a specific partition, and a global index corresponds to the entire partitioned table.
- If the constraint key of the unique constraint and primary key constraint contains all partition keys, a local index is created for the constraints. Otherwise, a global index is created.

NOTE

If the query statement involves multiple target partitions, you are advised to use the global index. Otherwise, you are advised to use the local index. However, note that the global index has extra overhead in the partition maintenance syntax.

Examples

- Create a table.

```
gaussdb=# CREATE TABLE web_returns_p2
(
    ca_address_sk INTEGER NOT NULL ,
    ca_address_id CHARACTER(16) NOT NULL ,
    ca_street_number CHARACTER(10) ,
    ca_street_name CHARACTER VARYING(60) ,
    ca_street_type CHARACTER(15) ,
    ca_suite_number CHARACTER(10) ,
    ca_city CHARACTER VARYING(60) ,
    ca_county CHARACTER VARYING(30) ,
    ca_state CHARACTER(2) ,
    ca_zip CHARACTER(10) ,
    ca_country CHARACTER VARYING(20) ,
    ca_gmt_offset NUMERIC(5,2) ,
    ca_location_type CHARACTER(20)
)
PARTITION BY RANGE (ca_address_sk)
(
    PARTITION P1 VALUES LESS THAN(5000),
    PARTITION P2 VALUES LESS THAN(10000),
    PARTITION P3 VALUES LESS THAN(15000),
    PARTITION P4 VALUES LESS THAN(20000),
    PARTITION P5 VALUES LESS THAN(25000),
    PARTITION P6 VALUES LESS THAN(30000),
    PARTITION P7 VALUES LESS THAN(40000),
    PARTITION P8 VALUES LESS THAN(MAXVALUE)
)
ENABLE ROW MOVEMENT;
```

- Create an index.
  - Create the local index **tpcds_web_returns_p2_index1** without specifying the partition name.
    ```
    gaussdb=# CREATE INDEX tpcds_web_returns_p2_index1 ON web_returns_p2 (ca_address_id)
    LOCAL;
    ```
    If the following information is displayed, the creation is successful:
    ```
    CREATE INDEX
    ```
  - Create the local index **tpcds_web_returns_p2_index2** with the specified partition name.
    ```
    gaussdb=# CREATE TABLESPACE example2 LOCATION '/home/omm/example2';
    gaussdb=# CREATE TABLESPACE example3 LOCATION '/home/omm/example3';
    gaussdb=# CREATE TABLESPACE example4 LOCATION '/home/omm/example4';

    gaussdb=# CREATE INDEX tpcds_web_returns_p2_index2 ON web_returns_p2 (ca_address_sk)
    LOCAL
    (
        PARTITION web_returns_p2_P1_index,
        PARTITION web_returns_p2_P2_index TABLESPACE example3,
        PARTITION web_returns_p2_P3_index TABLESPACE example4,
        PARTITION web_returns_p2_P4_index,
        PARTITION web_returns_p2_P5_index,
        PARTITION web_returns_p2_P6_index,
        PARTITION web_returns_p2_P7_index,
        PARTITION web_returns_p2_P8_index
    ) TABLESPACE example2;
    ```
    If the following information is displayed, the creation is successful:
    ```
    CREATE INDEX
    ```
  - Create the global index **tpcds_web_returns_p2_global_index** for a partitioned table.
    ```
    gaussdb=# CREATE INDEX tpcds_web_returns_p2_global_index ON web_returns_p2
    (ca_street_number) GLOBAL;
    ```
    If the following information is displayed, the creation is successful:
    ```
    CREATE INDEX
    ```
- Modify the tablespace of an index partition.
  - Change the tablespace of index partition **web_returns_p2_P2_index** to **example1**.
    ```
    gaussdb=# ALTER INDEX tpcds_web_returns_p2_index2 MOVE PARTITION
    web_returns_p2_P2_index TABLESPACE example1;
    ```
    If the following information is displayed, the modification is successful:
    ```
    ALTER INDEX
    ```
  - Change the tablespace of index partition **web_returns_p2_P3_index** to **example2**.

```
gaussdb=# ALTER INDEX tpcds_web_returns_p2_index2 MOVE PARTITION
web_returns_p2_P3_index TABLESPACE example2;
```

If the following information is displayed, the modification is successful:

```
ALTER INDEX
```

- Rename an index partition.
  - Rename index partition **web_returns_p2_P8_index** to **web_returns_p2_P8_index_new**.

    ```
    gaussdb=# ALTER INDEX tpcds_web_returns_p2_index2 RENAME PARTITION
    web_returns_p2_P8_index TO web_returns_p2_P8_index_new;
    ```

    If the following information is displayed, the renaming is successful:

    ```
    ALTER INDEX
    ```
- Query indexes.
  - Query all indexes defined by the system and users.

    ```
    gaussdb=# SELECT RELNAME FROM PG_CLASS WHERE RELKIND='i' or RELKIND='I';
    ```
  - Query information about a specified index.

    ```
    gaussdb=# \di+ tpcds_web_returns_p2_index2
    ```
- Drop an index.

  ```
  gaussdb=# DROP INDEX tpcds_web_returns_p2_index1;
  ```

  If the following information is displayed, the deletion is successful:

  ```
  DROP INDEX
  ```

  Perform the cleanup operation.

  ```
  -- Cleanup example
  gaussdb=# DROP TABLE web_returns_p2;
  ```

2.4 Partitioned Table O&M Management

Partitioned table O&M management includes partition management, partitioned table management, partitioned index management, and partitioned table statement concurrency support.

- Partition management: also known as partition-level DDL operations, including ADD, DROP, EXCHANGE, TRUNCATE, SPLIT, MERGE, MOVE, and RENAME.

CAUTION

- For hash partitions, operations involving partition quantity change will cause data re-shuffling, including ADD, DROP, SPLIT, and MERGE. Therefore, GaussDB does not support these operations.
- Operations involving partition data change will invalidate global indexes, including DROP, EXCHANGE, TRUNCATE, SPLIT, and MERGE. You can use the UPDATE GLOBAL INDEX clause to update global indexes synchronously.

NOTE

- Most partition DDL operations use PARTITION/SUBPARTITION and PARTITION/SUBPARTITION FOR to specify partitions. For PARTITION/SUBPARTITION, you need to specify the partition name. For PARTITION/SUBPARTITION FOR, you need to specify any partition value within the partition range. For example, if the range of partition **part1** is defined as [100, 200), **partition part1** and **partition for(150)** function the same.
- The DDL execution cost varies depending on the partition. The target partition will be locked during DDL execution. Therefore, you need to evaluate the cost and impact on services. Generally, the execution cost of splitting and merging is much greater than that of other partition DDL operations and is positively correlated with the size of the source partition. The cost of exchanging is mainly caused by global index rebuilding and validation. The cost of moving is limited by disk I/O. The execution cost of other partition DDL operations is low.

- Partitioned table management: In addition to the functions inherited from ordinary tables, you can enable or disable row migration for partitioned tables.
- Partitioned index management: You can invalidate indexes or index partitions or rebuild invalid indexes or index partitions. For example, global indexes become invalid due to partition management operations.
- Partitioned table statement concurrency support: When partition-level DDL operations and partition-level DQL/DML operations are applied to different partitions, concurrency at the execution layer is supported.

2.4.1 ADD PARTITION

You can add partitions to an existing partitioned table to maintain new services. Currently, a partitioned table can contain a maximum of 1048575 partitions. If the number of partitions reaches the upper limit, no more partitions can be added. In addition, the memory usage of partitions must be considered. Typically, the memory usage of a partitioned table is about (Number of partitions x 3/1024) MB. The memory usage of a partition cannot be greater than the value of **local_syscache_threshold**. In addition, some space must be reserved for other functions.

CAUTION

- This command cannot be applied to hash partitions.

2.4.1.1 Adding a Partition to a Range Partitioned Table

You can run **ALTER TABLE ADD PARTITION** to add a partition to the end of an existing partitioned table. The upper boundary of the new partition must be greater than that of the last partition.

For example, add a partition to the range partitioned table **range_sales**.

```
ALTER TABLE range_sales ADD PARTITION date_202005 VALUES LESS THAN ('2020-06-01') TABLESPACE tb1;
```

NOTICE

If a range partitioned table has the MAXVALUE partition, partitions cannot be added. You can run the **ALTER TABLE SPLIT PARTITION** command to split partitions. Partition splitting is also applicable to the scenario where partitions need to be added before or in the middle of an existing partitioned table. For details, see **Splitting a Partition for a Range Partitioned Table**.

2.4.1.2 Adding a Partition to an Interval Partitioned Table

Partitions cannot be added to an interval partitioned table by running the **ALTER TABLE ADD PARTITION** statement. If the data inserted by a user exceeds the range of the existing interval partitioned table, the database automatically creates a partition based on the **INTERVAL** value of the interval partitioned table.

For example, after the following data is inserted into the interval partitioned table **interval_sales**, the database creates a partition whose range is ['2020-07-01', '2020-08-01'). The new partition names start from **sys_p1** in ascending order.

```
INSERT INTO interval_sales VALUES (263722,42819872,'2020-07-09','E',432072,213,17);
```

2.4.1.3 Adding a Partition to a List Partitioned Table

You can run **ALTER TABLE ADD PARTITION** to add a partition to a list partitioned table. The enumerated values of the new partition cannot be the same as those of any existing partition.

For example, add a partition to the list partitioned table **list_sales**.

```
ALTER TABLE list_sales ADD PARTITION channel5 VALUES ('X') TABLESPACE tb1;
```

NOTICE

If a list partitioned table has the DEFAULT partition, partitions cannot be added. You can use the ALTER TABLE SPLIT PARTITION statement to split partitions.

2.4.1.4 Adding a Partition to a Level-2 Partitioned Table

You can run **ALTER TABLE ADD PARTITION** to add a range or list partition to a level-2 partitioned table. If a level-2 partition definition is declared under the new partition, the database creates the corresponding level-2 partition based on the definition. If no level-2 partition definition is declared under the new partition, the database automatically creates a default level-2 partition.

For example, add a partition to the level-2 partitioned table **range_list_sales** and create four level-2 partitions.

```
ALTER TABLE range_list_sales ADD PARTITION date_202005 VALUES LESS THAN ('2020-06-01')
TABLESPACE tb1
(
    SUBPARTITION date_202005_channel1 VALUES ('0', '1', '2'),
    SUBPARTITION date_202005_channel2 VALUES ('3', '4', '5') TABLESPACE tb2,
    SUBPARTITION date_202005_channel3 VALUES ('6', '7'),
    SUBPARTITION date_202005_channel4 VALUES ('8', '9')
);
```

Alternatively, add only a partition to the level-2 partitioned table **range_list_sales**.

```
ALTER TABLE range_list_sales ADD PARTITION date_202005 VALUES LESS THAN ('2020-06-01')
TABLESPACE tb1;
```

The preceding statement is equivalent to the following SQL statement:

```
ALTER TABLE range_list_sales ADD PARTITION date_202005 VALUES LESS THAN ('2020-06-01')
TABLESPACE tb1
(
   SUBPARTITION date_202005_channel1 VALUES (DEFAULT)
);
```

NOTICE

If the level-1 partitioning policy of a level-2 partitioned table is HASH, the partition cannot be added using ALTER TABLE ADD PARTITION.

2.4.1.5 Adding a Level-2 Partition to a Level-2 Partitioned Table

You can run **ALTER TABLE MODIFY PARTITION ADD SUBPARTITION** to add a level-2 range or list partition to a level-2 partitioned table.

For example, add a level-2 partition named **date_202004** to the level-2 partitioned table **range_list_sales**.

```
ALTER TABLE range_list_sales MODIFY PARTITION date_202004 ADD SUBPARTITION date_202004_channel5
VALUES ('X') TABLESPACE tb2;
```

NOTICE

If the level-2 partitioning policy of a level-2 partitioned table is HASH, the level-2 partition cannot be added using ALTER TABLE MODIFY PARTITION ADD SUBPARTITION.

2.4.2 DROP PARTITION

You can run this command to remove unnecessary partitions. You can delete a partition by specifying the partition name or partition value.

CAUTION

- This command cannot be applied to hash partitions.
- Running this command will invalidate the global index. You can use the UPDATE GLOBAL INDEX clause to update the global index synchronously or rebuild the global index.

2.4.2.1 Deleting a Partition from a Partitioned Table

You can run **ALTER TABLE DROP PARTITION** to delete any partition from a range partitioned table, interval partitioned table, or list partitioned table.

For example, delete the partition **date_202005** from the range partitioned table **range_sales** by specifying the partition name and update the global index.

```
ALTER TABLE range_sales DROP PARTITION date_202005 UPDATE GLOBAL INDEX;
```

Alternatively, delete the partition corresponding to the partition value **'2020-05-08'** in the range partitioned table **range_sales**. Global indexes become invalid after this command is executed because the UPDATE GLOBAL INDEX clause is not used.

```
ALTER TABLE range_sales DROP PARTITION FOR ('2020-05-08');
```

NOTICE

- If a partitioned table has only one partition, the partition cannot be deleted by using the ALTER TABLE DROP PARTITION statement.
- If the partitioned table is a hash partitioned table, partitions in the table cannot be deleted by using the ALTER TABLE DROP PARTITION statement.

2.4.2.2 Deleting a Partition from a Level-2 Partitioned Table

You can run **ALTER TABLE DROP PARTITION** to delete a range or list partition from a level-2 partitioned table. The database deletes the partition and all level-2 partitions under the partition.

For example, delete the partition **date_202005** from the level-2 partitioned table **range_list_sales** by specifying the partition name and update the global index.

```
ALTER TABLE range_list_sales DROP PARTITION date_202005 UPDATE GLOBAL INDEX;
```

Alternatively, delete a partition corresponding to the partition value **('2020-05-08')** in the level-2 partitioned table **range_list_sales**. Global indexes become invalid after this command is executed because the UPDATE GLOBAL INDEX clause is not used.

```
ALTER TABLE range_list_sales DROP PARTITION FOR ('2020-05-08');
```

NOTICE

- If a level-2 partitioned table has only one partition, the partition cannot be deleted using the ALTER TABLE DROP PARTITION statement.
- If the level-1 partition policy of a level-2 partitioned table is HASH, the partition cannot be deleted using the ALTER TABLE DROP PARTITION statement.

2.4.2.3 Deleting a Level-2 Partition from a Level-2 Partitioned Table

You can run **ALTER TABLE DROP SUBPARTITION** to delete a level-2 range or list partition from a level-2 partitioned table.

For example, delete the level-2 partition **date_202005_channel1** from the level-2 partitioned table **range_list_sales** by specifying the partition name and update the global index.

```
ALTER TABLE range_list_sales DROP SUBPARTITION date_202005_channel1 UPDATE GLOBAL INDEX;
```

Alternatively, delete a level-2 partition corresponding to the partition value **('2020-05-08', '0')** in the level-2 partitioned table **range_list_sales**. Global indexes become invalid after this command is executed because the UPDATE GLOBAL INDEX clause is not used.

```
ALTER TABLE range_list_sales DROP SUBPARTITION FOR ('2020-05-08', '0');
```

NOTICE

- If the level-2 partitioned table has only one level-2 partition, the level-2 partition cannot be deleted using the ALTER TABLE DROP SUBPARTITION statement.
- If the level-2 partition policy of a level-2 partitioned table is HASH, the level-2 partition cannot be deleted using the ALTER TABLE DROP SUBPARTITION statement.

2.4.3 EXCHANGE PARTITION

You can run this command to exchange the data in a partition with that in an ordinary table. This command can quickly import data to or export data from a partitioned table, achieving efficient data loading. In service migration scenarios, using EXCHANGE PARTITION is much faster than using common import operation. You can exchange a partition by specifying the partition name or partition value.

CAUTION

- Running this command will invalidate the global index. You can use the UPDATE GLOBAL INDEX clause to update the global index synchronously or rebuild the global index.

NOTICE

- When exchanging partitions, you can declare WITH/WITHOUT VALIDATION, indicating whether to validate that ordinary table data meets the partition key constraint rules of the target partition (validated by default). The overhead of data validation is high. If you ensure that the exchanged data belongs to the target partition, you can declare WITHOUT VALIDATION to improve the exchange performance.
- You can declare WITH VALIDATION VERBOSE. In this case, the database validates each row of the ordinary table, inserts the data that does not meet the partition key constraint of the target partition to other partitions of the partitioned table, and exchanges the ordinary table with the target partition.

For example, if the following partition definition and data distribution of the **exchange_sales** table are provided, and the **DATE_202001** partition is exchanged with the **exchange_sales** table, the following behaviors exist based on the declaration clause:

- If WITHOUT VALIDATION is declared, all data is exchanged to the **DATE_202001** partition. Because **'2020-02-03**' and **'2020-04-08'** do not meet the range constraint of the **DATE_202001** partition, subsequent services may be abnormal.
- If WITH VALIDATION is declared, and **'2020-02-03'** and **'2020-04-08'** do not meet the range constraint of the **DATE_202001** partition, the database reports an error.

- If WITH VALIDATION VERBOSE is declared, the database inserts **'2020-02-03'** into the **DATE_202002** partition, inserts **'2020-04-08'** into the **DATE_202004** partition, and exchanges the remaining data with the **DATE_202001** partition.

```
-- Partition definition
PARTITION DATE_202001 VALUES LESS THAN ('2020-02-01'),
PARTITION DATE_202002 VALUES LESS THAN ('2020-03-01'),
PARTITION DATE_202003 VALUES LESS THAN ('2020-04-01'),
PARTITION DATE_202004 VALUES LESS THAN ('2020-05-01')
-- Data distribution of exchange_sales
('2020-01-15', '2020-01-17', '2020-01-23', '2020-02-03', '2020-04-08')
```

WARNING

If the data to be exchanged does not completely belong to the target partition, do not declare WITHOUT VALIDATION. Otherwise, the partition constraint rules will be damaged, and subsequent DML statement results of the partitioned table will be abnormal.

The ordinary table and partition whose data is to be exchanged must meet the following requirements:

- The number of columns in an ordinary table is the same as that in a partition, and the information in the corresponding columns is strictly consistent.
- The compression information of the ordinary table and partitioned table is consistent.
- The number of ordinary table indexes is the same as that of local indexes of the partition, and the index information is the same.
- The number and information of constraints of the ordinary table and partition are consistent.
- The ordinary table is not a temporary table.
- The ordinary table and partitioned table do not support dynamic data masking and row-level security constraints.

2.4.3.1 Exchanging Partitions for a Partitioned Table

You can run **ALTER TABLE EXCHANGE PARTITION** to exchange partitions for a partitioned table.

For example, exchange the partition **date_202001** of the partitioned table **range_sales** with the ordinary table **exchange_sales** by specifying the partition name without validating the partition key, and update the global index.

```
ALTER TABLE range_sales EXCHANGE PARTITION (date_202001) WITH TABLE exchange_sales WITHOUT
VALIDATION UPDATE GLOBAL INDEX;
```

Alternatively, exchange the partition corresponding to **'2020-01-08'** in the range partitioned table **range_sales** with the ordinary table **exchange_sales** by specifying a partition value, validate the partition, and insert data that does not meet the target partition constraints into another partition of the partitioned table. Global indexes become invalid after this command is executed because the UPDATE GLOBAL INDEX clause is not used.

```
ALTER TABLE range_sales EXCHANGE PARTITION FOR ('2020-01-08') WITH TABLE exchange_sales WITH
VALIDATION VERBOSE;
```

2.4.3.2 Exchanging Level-2 Partitions for a Level-2 Partitioned Table

You can run **ALTER TABLE EXCHANGE SUBPARTITION** to exchange level-2 partitions in a level-2 partitioned table.

For example, exchange the level-2 partition **date_202001_channel1** of the level-2 partitioned table **range_list_sales** with the ordinary table **exchange_sales** by specifying the partition name without validating the partition key, and update the global index.

```
ALTER TABLE range_list_sales EXCHANGE SUBPARTITION (date_202001_channel1) WITH TABLE
exchange_sales WITHOUT VALIDATION UPDATE GLOBAL INDEX;
```

Alternatively, exchange the level-2 partition corresponding to **('2020-01-08', '0')** in the level-2 partitioned table **range_list_sales** with the ordinary table **exchange_sales** by specifying a partition value, validate the partition, and insert data that does not meet the target partition constraints into another partition of the partitioned table. Global indexes become invalid after this command is executed because the UPDATE GLOBAL INDEX clause is not used.

```
ALTER TABLE range_list_sales EXCHANGE SUBPARTITION FOR ('2020-01-08', '0') WITH TABLE
exchange_sales WITH VALIDATION VERBOSE;
```

NOTICE

Partitions in a level-2 partitioned table cannot be exchanged.

2.4.4 TRUNCATE PARTITION

You can run this command to quickly clear data in a partition. The function is similar to that of DROP PARTITION. The difference is that TRUNCATE PARTITION deletes only data in a partition, and the definition and physical files of the partition are retained. You can clear a partition by specifying the partition name or partition value.

CAUTION

- Running this command will invalidate the global index. You can use the UPDATE GLOBAL INDEX clause to update the global index synchronously or rebuild the global index.

2.4.4.1 Clearing Partitions from a Partitioned Table

You can run **ALTER TABLE TRUNCATE PARTITION** to clear any partition in a specified partitioned table.

For example, truncate the partition **date_202005** in the range partitioned table **range_sales** by specifying the partition name and update the global index.

```
ALTER TABLE range_sales TRUNCATE PARTITION date_202005 UPDATE GLOBAL INDEX;
```

Alternatively, truncate the partition corresponding to the partition value **'2020-05-08'** in the range partitioned table **range_sales**. Global indexes become invalid after this command is executed because the UPDATE GLOBAL INDEX clause is not used.

```
ALTER TABLE range_sales TRUNCATE PARTITION FOR ('2020-05-08');
```

2.4.4.2 Clearing Partitions from a Level-2 Partitioned Table

You can run **ALTER TABLE TRUNCATE PARTITION** to clear a partition in a level-2 partitioned table. The database clears all level-2 partitions under the partition.

For example, truncate the partition **date_202005** in the level-2 partitioned table **range_list_sales** by specifying the partition name and update the global index.

```
ALTER TABLE range_list_sales TRUNCATE PARTITION date_202005 UPDATE GLOBAL INDEX;
```

Alternatively, truncate a partition corresponding to the partition value **('2020-05-08')** in the level-2 partitioned table **range_list_sales**. Global indexes become invalid after this command is executed because the UPDATE GLOBAL INDEX clause is not used.

```
ALTER TABLE range_list_sales TRUNCATE PARTITION FOR ('2020-05-08');
```

2.4.4.3 Clearing Level-2 Partitions from a Level-2 Partitioned Table

You can run **ALTER TABLE TRUNCATE SUBPARTITION** to clear a level-2 partition in a level-2 partitioned table.

For example, truncate the level-2 partition **date_202005_channel1** in the level-2 partitioned table **range_list_sales** by specifying the partition name and update the global index.

```
ALTER TABLE range_list_sales TRUNCATE SUBPARTITION date_202005_channel1 UPDATE GLOBAL INDEX;
```

Alternatively, truncate a level-2 partition corresponding to the partition value **('2020-05-08', '0')** in the level-2 partitioned table **range_list_sales**. Global indexes become invalid after this command is executed because the UPDATE GLOBAL INDEX clause is not used.

```
ALTER TABLE range_list_sales TRUNCATE SUBPARTITION FOR ('2020-05-08', '0');
```

2.4.5 SPLIT PARTITION

You can run this command to split a partition into two or more partitions. This operation is considered when the partition data is too large or you need to add a partition to a range partition with MAXVALUE or a list partition with DEFAULT. You can specify a split point to split a partition into two partitions, or split a partition into multiple partitions without specifying a split point. You can split a partition by specifying the partition name or partition value.

CAUTION

- This command cannot be applied to hash partitions.
- Partitions in a level-2 partitioned table cannot be split.
- Running this command will invalidate the global index. You can use the UPDATE GLOBAL INDEX clause to update the global index synchronously or rebuild the global index.

NOTICE

The names of the new partitions can be the same as that of the source partition. For example, partition **p1** is split into **p1** and **p2**. However, the database does not consider the partitions with the same name before and after the splitting as the same partition, which affects the query of the source partition **p1** during the splitting. For details, see **DQL/DML-DDL Concurrency**.

2.4.5.1 Splitting a Partition for a Range Partitioned Table

You can run **ALTER TABLE SPLIT PARTITION** to split a partition for a range partitioned table.

For example, the range of the **date_202001** partition in the range partitioned table **range_sales** is ['2020-01-01', '2020-02-01'). You can specify the split point **'2020-01-16'** to split the **date_202001** partition into two partitions and update the global index.

```
ALTER TABLE range_sales SPLIT PARTITION date_202001 AT ('2020-01-16') INTO
(
    PARTITION date_202001_p1, -- The upper boundary of the first partition is '2020-01-16'.
    PARTITION date_202001_p2  -- The upper boundary of the second partition is '2020-02-01'.
) UPDATE GLOBAL INDEX;
```

Alternatively, split the partition **date_202001** into multiple partitions without specifying a split point, and update the global index.

```
ALTER TABLE range_sales SPLIT PARTITION date_202001 INTO
(
    PARTITION date_202001_p1 VALUES LESS THAN ('2020-01-11'),
    PARTITION date_202001_p2 VALUES LESS THAN ('2020-01-21'),
    PARTITION date_202001_p3 -- The upper boundary of the third partition is '2020-02-01'.
)UPDATE GLOBAL INDEX;
```

Alternatively, split the partition by specifying the partition value instead of the partition name.

```
ALTER TABLE range_sales SPLIT PARTITION FOR ('2020-01-15') AT ('2020-01-16') INTO
(
    PARTITION date_202001_p1, -- The upper boundary of the first partition is '2020-01-16'.
    PARTITION date_202001_p2  -- The upper boundary of the second partition is '2020-02-01'.
) UPDATE GLOBAL INDEX;
```

NOTICE

If the MAXVALUE partition is split, the MAXVALUE range cannot be declared for the first several partitions, and the last partition inherits the MAXVALUE range.

2.4.5.2 Splitting a Partition for an Interval Partitioned Table

You can run **ALTER TABLE SPLIT PARTITION** to split a partition for an interval partitioned table.

NOTICE

After an interval partition is split, the interval partition before the split partition becomes a range partition.

For example, create the following interval partitioned table and add three partitions: **sys_p1**, **sys_p2**, and **sys_p3**.

```
CREATE TABLE interval_sales
(
    prod_id       NUMBER(6),
    cust_id       NUMBER,
    time_id       DATE,
    channel_id    CHAR(1),
    promo_id      NUMBER(6),
    quantity_sold NUMBER(3),
    amount_sold   NUMBER(10, 2)
)
PARTITION BY RANGE (TIME_ID) INTERVAL ('1 MONTH')
(
    PARTITION date_2015 VALUES LESS THAN ('2016-01-01'),
    PARTITION date_2016 VALUES LESS THAN ('2017-01-01'),
    PARTITION date_2017 VALUES LESS THAN ('2018-01-01'),
    PARTITION date_2018 VALUES LESS THAN ('2019-01-01'),
    PARTITION date_2019 VALUES LESS THAN ('2020-01-01')
);
INSERT INTO interval_sales VALUES (263722,42819872,'2020-07-09','E',432072,213,17); -- The sys_p1
partition is added.
INSERT INTO interval_sales VALUES (345724,72651233,'2021-03-05','A',352451,146,9);  -- The sys_p2
partition is added.
INSERT INTO interval_sales VALUES (153241,65143129,'2021-05-07','H',864134,89,34);  -- The sys_p3
partition is added.
```

If the **sys_p2** partition is split, the **sys_p1** partition is changed to a range partition, and the lower boundary of the partition range depends on the upper boundary of the previous partition instead of the interval partition value. That is, the partition range changes from ['2020-07-01', '2020-08-01') to ['2020-01-01', '2020-08-01'). The **sys_p3** partition is still an interval partition, and its partition range is ['2021-05-01', '2021-06-01').

2.4.5.3 Splitting a Partition for a List Partitioned Table

You can run **ALTER TABLE SPLIT PARTITION** to split a partition for a list partitioned table.

For example, assume that the range defined for the partition **channel2** of the list partitioned table **list_sales** is ('6', '7', '8', '9'). You can specify the split point **('6', '7')** to split the **channel2** partition into two partitions and update the global index.

```
ALTER TABLE list_sales SPLIT PARTITION channel2 VALUES ('6', '7') INTO
(
    PARTITION channel2_1, -- The first partition range is ('6', '7').
    PARTITION channel2_2  -- The second partition range is ('8', '9').
) UPDATE GLOBAL INDEX;
```

Alternatively, split the partition **channel2** into multiple partitions without specifying a split point, and update the global index.

```
ALTER TABLE list_sales SPLIT PARTITION channel2 INTO
(
    PARTITION channel2_1 VALUES ('6'),
    PARTITION channel2_2 VALUES ('8'),
    PARTITION channel2_3 -- The third partition range is ('7', '9').
)UPDATE GLOBAL INDEX;
```

Alternatively, split the partition by specifying the partition value instead of the partition name.

```
ALTER TABLE list_sales SPLIT PARTITION FOR ('6') VALUES ('6', '7') INTO
(
    PARTITION channel2_1, -- The first partition range is ('6', '7').
    PARTITION channel2_2  -- The second partition range is ('8', '9').
) UPDATE GLOBAL INDEX;
```

CAUTION

If the DEFAULT partition is split, the DEFAULT range cannot be declared for the first several partitions, and the last partition inherits the DEFAULT range.

2.4.5.4 Splitting a Level-2 Partition for a Level-2 *-Range Partitioned Table

You can run **ALTER TABLE SPLIT SUBPARTITION** to split a level-2 partition for a level-2 *-range partitioned table.

For example, assume that the defined range of the level-2 partition **channel1_customer4** of a level-2 *-range partitioned table **list_range_sales** is [1000, MAXVALUE). You can specify the split point **1200** to split the **channel1_customer4** level-2 partition into two partitions and update the global index.

```
ALTER TABLE list_range_sales SPLIT SUBPARTITION channel1_customer4 AT (1200) INTO
(
   SUBPARTITION channel1_customer4_p1, -- The upper boundary of the first partition is 1200.
   SUBPARTITION channel1_customer4_p2  -- The upper boundary of the second partition is MAXVALUE.
) UPDATE GLOBAL INDEX;
```

Alternatively, split the partition **channel1_customer4** into multiple partitions without specifying a split point, and update the global index.

```
ALTER TABLE list_range_sales SPLIT SUBPARTITION channel1_customer4 INTO
(
   SUBPARTITION channel1_customer4_p1 VALUES LESS THAN (1200),
   SUBPARTITION channel1_customer4_p2 VALUES LESS THAN (1400),
   SUBPARTITION channel1_customer4_p3 -- The upper boundary of the third partition is MAXVALUE.
)UPDATE GLOBAL INDEX;
```

Alternatively, split the partition by specifying the partition value instead of the partition name.

```
ALTER TABLE range_sales SPLIT SUBPARTITION FOR ('1', 1200) AT (1200) INTO
(
   PARTITION channel1_customer4_p1,
   PARTITION channel1_customer4_p2
) UPDATE GLOBAL INDEX;
```

NOTICE

If the MAXVALUE partition is split, the MAXVALUE range cannot be declared for the first several partitions, and the last partition inherits the MAXVALUE range.

2.4.5.5 Splitting a Level-2 Partition for a Level-2 *-List Partitioned Table

You can run **ALTER TABLE SPLIT SUBPARTITION** to split a level-2 partition for a level-2 *-list partitioned table.

For example, assume that the defined range of the level-2 partition **product2_channel2** of a level-2 *-list partitioned table **hash_list_sales** is DEFAULT. You can specify a split point to split the level-2 partition into two partitions and update the global index.

```
ALTER TABLE hash_list_sales SPLIT SUBPARTITION product2_channel2 VALUES ('6', '7', '8', '9') INTO
(
   SUBPARTITION product2_channel2_p1, -- The first partition range is ('6', '7', '8', '9').
   SUBPARTITION product2_channel2_p2  -- The second partition range is DEFAULT.
) UPDATE GLOBAL INDEX;
```

Alternatively, split the partition **product2_channel2** into multiple partitions without specifying a split point, and update the global index.

```
ALTER TABLE hash_list_sales SPLIT SUBPARTITION product2_channel2 INTO
(
    SUBPARTITION product2_channel2_p1 VALUES ('6', '7', '8'),
    SUBPARTITION product2_channel2_p2 VALUES ('9', '10'),
    SUBPARTITION product2_channel2_p3 -- The third partition range is DEFAULT.
) UPDATE GLOBAL INDEX;
```

Alternatively, split the partition by specifying the partition value instead of the partition name.

```
ALTER TABLE hash_list_sales SPLIT SUBPARTITION FOR (1200, '6') VALUES ('6', '7', '8', '9') INTO
(
    SUBPARTITION product2_channel2_p1, -- The first partition range is ('6', '7', '8', '9').
    SUBPARTITION product2_channel2_p2  -- The second partition range is DEFAULT.
) UPDATE GLOBAL INDEX;
```

CAUTION

If the DEFAULT partition is split, the DEFAULT range cannot be declared for the first several partitions, and the last partition inherits the DEFAULT range.

2.4.6 MERGE PARTITION

You can run this command to merge multiple partitions into one partition. Partitions can be merged only by specifying partition names, instead of partition values.

CAUTION

- This command cannot be applied to hash partitions.
- Running this command will invalidate the global index. You can use the UPDATE GLOBAL INDEX clause to update the global index synchronously or rebuild the global index.

NOTICE

For a range or interval partition, the name of the new partition can be the same as that of the last source partition. For example, partitions **p1** and **p2** can be merged into **p2**. For a list partition, the name of the new partition can be the same as that of any source partition. For example, **p1** and **p2** can be merged into **p1**.

If the name of the new partition is the same as that of the source partition, the database considers the new partition as inheritance of the source partition, which affects the query of the source partition during the merging. For details, see **DQL/DML-DDL Concurrency**.

2.4.6.1 Merging Partitions for a Partitioned Table

You can run **ALTER TABLE MERGE PARTITIONS** to merge multiple partitions into one partition.

For example, merge the partitions **date_202001** and **date_202002** of the range partitioned table **range_sales** into a new partition and update the global index.

```
ALTER TABLE range_sales MERGE PARTITIONS date_202001, date_202002 INTO
    PARTITION date_2020_old UPDATE GLOBAL INDEX;
```

NOTICE

After interval partitions are merged, the interval partition before the merged partitions becomes a range partition.

2.4.6.2 Merging Level-2 Partitions for a Level-2 Partitioned Table

You can run **ALTER TABLE MERGE SUBPARTITIONS** to merge multiple level-2 partitions into one level-2 partition.

For example, merge the level-2 partitions **product1_channel1**, **product1_channel2** and **product1_channel3** of the level-2 partitioned table **hash_list_sales** into a new level-2 partition and update the global index.

```
ALTER TABLE hash_list_sales MERGE SUBPARTITIONS product1_channel1, product1_channel2,
product1_channel3 INTO
    SUBPARTITION product1_channel1 UPDATE GLOBAL INDEX;
```

2.4.7 MOVE PARTITION

You can run this command to move a partition to a new tablespace. You can move a partition by specifying the partition name or partition value.

2.4.7.1 Moving Partitions for a Partitioned Table

You can run **ALTER TABLE MOVE PARTITION** to move partitions in a partitioned table.

For example, move the partition **date_202001** from the range partitioned table **range_sales** to the tablespace **tb1** by specifying the partition name.

```
ALTER TABLE range_sales MOVE PARTITION date_202001 TABLESPACE tb1;
```

Alternatively, move the partition corresponding to **'0'** in the list partitioned table **list_sales** to the tablespace **tb1** by specifying a partition value.

```
ALTER TABLE list_sales MOVE PARTITION FOR ('0') TABLESPACE tb1;
```

2.4.7.2 Moving Level-2 Partitions for a Level-2 Partitioned Table

You can run **ALTER TABLE MOVE SUBPARTITION** to move level-2 partitions in a level-2 partitioned table.

For example, move the partition **date_202001_channel1** from the level-2 partitioned table **range_list_sales** to the tablespace **tb1** by specifying the partition name.

```
ALTER TABLE range_list_sales MOVE SUBPARTITION date_202001_channel1 TABLESPACE tb1;
```

Alternatively, move the partition corresponding to the partition value **('2020-01-08', '0')** from the level-2 partitioned table **range_list_sales** to the tablespace **tb1**.

```
ALTER TABLE range_list_sales MOVE SUBPARTITION FOR ('2020-01-08', '0') TABLESPACE tb1;
```

2.4.8 RENAME PARTITION

You can run this command to rename a partition. You can rename a partition by specifying the partition name or partition value.

2.4.8.1 Renaming a Partition in a Partitioned Table

You can run **ALTER TABLE RENAME PARTITION** to rename a partition in a partitioned table.

For example, rename the partition **date_202001** in the range partitioned table **range_sales** by specifying the partition name.

```
ALTER TABLE range_sales RENAME PARTITION date_202001 TO date_202001_new;
```

Alternatively, rename the partition corresponding to **'0'** in the list partitioned table **list_sales** by specifying a partition value.

```
ALTER TABLE list_sales RENAME PARTITION FOR ('0') TO channel_new;
```

2.4.8.2 Renaming a Partition in a Level-2 Partitioned Table

You can run **ALTER TABLE RENAME PARTITION** to rename a partition in a level-2 partitioned table. The specific method is the same as that of the partitioned table.

2.4.8.3 Renaming a Level-2 Partition in a Level-2 Partitioned Table

You can run **ALTER TABLE RENAME SUBPARTITION** to rename a level-2 partition in a level-2 partitioned table.

For example, rename the partition **date_202001_channel1** in the level-2 partitioned table **range_list_sales** by specifying the partition name.

```
ALTER TABLE range_list_sales RENAME SUBPARTITION date_202001_channel1 TO date_202001_channelnew;
```

Alternatively, rename the partition corresponding to the partition value **('2020-01-08', '0')** in the level-2 partitioned table **range_list_sales**.

```
ALTER TABLE range_list_sales RENAME SUBPARTITION FOR ('2020-01-08', '0') TO date_202001_channelnew;
```

2.4.8.4 Renaming an Index Partition for a Local Index

You can run **ALTER INDEX RENAME PARTITION** to rename an index partition for a local index. The method is the same as that for renaming a partition in a partitioned table.

2.4.9 ALTER TABLE ENABLE/DISABLE ROW MOVEMENT

You can run this command to enable or disable row movement for a partitioned table.

When row migration is enabled, data in a partition can be migrated to another partition through an UPDATE operation. When row migration is disabled, if such an UPDATE operation occurs, a service error is reported.

NOTICE

If you are not allowed to update the column where the partition key is located, you are advised to disable row migration.

For example, if you create a list partitioned table and enable row migration, you can update the column where the partition key is located across partitions. If you disable row migration, an error is reported when you update the column where the partition key is located across partitions.

```
CREATE TABLE list_sales
(
    product_id     INT4 NOT NULL,
    customer_id    INT4 PRIMARY KEY,
    time_id        DATE,
    channel_id     CHAR(1),
    type_id        INT4,
    quantity_sold  NUMERIC(3),
    amount_sold    NUMERIC(10,2)
)
PARTITION BY LIST (channel_id)
(
    PARTITION channel1 VALUES ('0', '1', '2'),
    PARTITION channel2 VALUES ('3', '4', '5'),
    PARTITION channel3 VALUES ('6', '7'),
    PARTITION channel4 VALUES ('8', '9')
) ENABLE ROW MOVEMENT;
INSERT INTO list_sales VALUES (153241,65143129,'2021-05-07','0',864134,89,34);
-- The cross-partition update is successful, and data is migrated from partition channel1 to partition
channel2.
UPDATE list_sales SET channel_id = '3' WHERE channel_id = '0';
-- Disable row migration for the partitioned table.
ALTER TABLE list_sales DISABLE ROW MOVEMENT;
-- The cross-partition update fails, and an error is reported: fail to update partitioned table "list_sales".
UPDATE list_sales SET channel_id = '0' WHERE channel_id = '3';
-- The update in the partition is still successful.
UPDATE list_sales SET channel_id = '4' WHERE channel_id = '3';
```

2.4.10 Invalidating/Rebuilding Indexes of a Partition

You can run commands to invalidate or rebuild a partitioned index or an index partition. In this case, the index or index partition is no longer maintained. You can rebuild a partitioned index to restore the index function.

In addition, some partition-level DDL operations also invalidate global indexes, including DROP, EXCHANGE, TRUNCATE, SPLIT, and MERGE. You can use the UPDATE GLOBAL INDEX clause to update the global index synchronously. Otherwise, you need to rebuild the index.

2.4.10.1 Invalidating/Rebuilding Indexes

You can run **ALTER INDEX** to invalidate or rebuild indexes.

For example, if the **range_sales_idx** index exists in the **range_sales** partitioned table, run the following command to invalidate the index:

```
ALTER INDEX range_sales_idx UNUSABLE;
```

Run the following command to rebuild the **range_sales_idx** index:

```
ALTER INDEX range_sales_idx REBUILD;
```

2.4.10.2 Invalidating/Rebuilding Local Indexes of a Partition

- You can run **ALTER INDEX PARTITION** to invalidate or rebuild local indexes of a partition.
- You can use ALTER TABLE MODIFY PARTITION to invalidate or rebuild all indexes of a specified partition in a partitioned table. If this syntax is applied to the partition of a level-2 partitioned table, this command takes effect on all level-2 partitions of the partition.

- You can use ALTER TABLE MODIFY SUBPARTITION to invalidate or rebuild all indexes of a specified level-2 partition in a level-2 partitioned table.

For example, assume that the partitioned table **range_sales** has two local indexes **range_sales_idx1** and **range_sales_idx2**, and the corresponding indexes on the partition **date_202001** are **range_sales_idx1_part1** and **range_sales_idx2_part1**.

The syntax for maintaining partitioned indexes of a partitioned table is as follows:

- Run the following command to disable all indexes on the **date_202001** partition:
  ```
  ALTER TABLE range_sales MODIFY PARTITION date_202001 UNUSABLE LOCAL INDEXES;
  ```
- Alternatively, run the following command to disable the index **range_sales_idx1_part1** on the **date_202001** partition:
  ```
  ALTER INDEX range_sales_idx1 MODIFY PARTITION range_sales_idx1_part1 UNUSABLE;
  ```
- Run the following command to rebuild all indexes on the **date_202001** partition:
  ```
  ALTER TABLE range_sales MODIFY PARTITION date_202001 REBUILD UNUSABLE LOCAL INDEXES;
  ```
- Alternatively, run the following command to rebuild the index **range_sales_idx1_part1** on the **date_202001** partition:
  ```
  ALTER INDEX range_sales_idx1 REBUILD PARTITION range_sales_idx1_part1;
  ```

Assume that the level-2 partitioned table **list_range_sales** has two local indexes: **list_range_sales_idx1** and **list_range_sales_idx2**. The table has a partition **channel1** and its level-2 partitions **channel1_product1**, **channel1_product2** and **channel1_product3**. The indexes corresponding to level-2 partition **channel1_product1** are **channel1_product1_idx1** and **channel1_product1_idx2**.

The syntax for maintaining the partitioned indexes of a level-2 partitioned table is as follows:

- Run the following command to disable all indexes on the level-2 partitions of partition **channel1**, including level-2 partitions **channel1_product1**, **channel1_product2** and **channel1_product3**:
  ```
  ALTER TABLE list_range_sales MODIFY PARTITION channel1 UNUSABLE LOCAL INDEXES;
  ```
- Run the following command to rebuild all indexes on the level-2 partitions under partition **channel1**:
  ```
  ALTER TABLE list_range_sales MODIFY PARTITION channel1 REBUILD UNUSABLE LOCAL INDEXES;
  ```

The syntax for maintaining the level-2 partitioned indexes of a level-2 partitioned table is as follows:

- Run the following command to disable all indexes on the level-2 partition **channel1_product1**:
  ```
  ALTER TABLE list_range_sales MODIFY SUBPARTITION channel1_product1 UNUSABLE LOCAL INDEXES;
  ```
- Run the following command to rebuild all indexes on the level-2 partition **channel1_product1**:
  ```
  ALTER TABLE list_range_sales MODIFY SUBPARTITION channel1_product1 REBUILD UNUSABLE LOCAL
  INDEXES;
  ```
- Alternatively, run the following command to disable the index **channel1_product1_idx1** on the level-2 partition **channel1_product1**:
  ```
  ALTER INDEX list_range_sales_idx1 MODIFY PARTITION channel1_product1_idx1 UNUSABLE;
  ```
- Run the following command to rebuild the index **channel1_product1_idx1** on the level-2 partition **channel1_product1**:
  ```
  ALTER INDEX list_range_sales_idx1 REBUILD PARTITION channel1_product1_idx1;
  ```

2.5 Partition Concurrency Control

Partition concurrency control limits the behavior specifications during concurrent DQL, DML, and DDL operations on partitioned tables. You can refer to this section when designing concurrent statements for partitioned tables, especially when maintaining partitions.

2.5.1 Common Lock Design

Partitioned tables use table locks and partition locks. Eight common locks of different levels are applied to tables and partitions to ensure proper behavior control during concurrent DQL, DML, and DDL operations. The following table lists the mutually exclusive behavior of locks at different levels. Every two types of common locks marked with √ do not block each other and can be executed concurrently.

Table 2-2 Common lock behavior

-	ACCESS_SHARE	ROW_SHARE	ROW_EXCLUSIVE	SHARE_UPDATE_EXCLUSIVE	SHARE	SHARE_ROW_EXCLUSIVE	EXCLUSIVE	ACCESS_EXCLUSIVE
ACCESS_SHARE	√	√	√	√	√	√	√	×
ROW_SHARE	√	√	√	√	√	√	×	×
ROW_EXCLUSIVE	√	√	√	√	×	×	×	×
SHARE_UPDATE_EXCLUSIVE	√	√	√	×	×	×	×	×
SHARE	√	√	×	×	√	×	×	×
SHARE_ROW_EXCLUSIVE	√	√	×	×	×	×	×	×
EXCLUSIVE	√	×	×	×	×	×	×	×
ACCESS_EXCLUSIVE	×	×	×	×	×	×	×	×

Different statements of a partitioned table are applied to the same target partition. The database applies different levels of table locks and partition locks to the target partitioned table and partition to control the concurrency behavior. The following table lists the lock control level for different statements. Numbers 1 to 8 indicate the eight common locks listed in the preceding table.

Table 2-3 Lock control level of different partitioned table statements

Statement	Partitioned Table Lock (Table Lock + Partition Lock)	Level-2 Partitioned Table Lock (Table Lock + Partition Lock + Level-2 Partition Lock)
SELECT	1-1	1-1-1
SELECT FOR UPDATE	2-2	2-2-2
DML statements, including INSERT, UPDATE, DELETE, UPSERT, MERGE INTO, and COPY	3-3	3-3-3
Partition-level DDL statements, including ADD, DROP, EXCHANGE, TRUNCATE, SPLIT, MERGE, MOVE, and RENAME	4-8	4-8-8 (used for partitions in a level-2 partitioned table) 4-4-8 (used for level-2 partitions in a level-2 partitioned table)
CREATE INDEX and REBUILD INDEX	5-5	5-5-5
REBUILD INDEX PARTITION	1-5	1-1-5
Other partitioned table-level DDL statements	8-8	8-8-8

2.5.2 DQL/DML-DQL/DML Concurrency

Level 1–3 locks will be used for DQL/DML statements on tables and partitions. DQL and DML statements do not block each other and DQL/DML-DQL/DML concurrency is supported.

CAUTION

Adding partitions to an interval partitioned table using statements, such as INSERT, UPDATE, UPSERT, MERGE INTO, and COPY, is regarded as a partition-level DDL operation.

2.5.3 DQL/DML-DDL Concurrency

Level-8 locks will be used for table-level DDL statements on a partitioned table. All DQL/DML statements are blocked.

Level-4 locks will be used for partition-level DDL statements on a partitioned table and level-8 locks will be used for the target partition. When DQL/DML and DDL statements are used in different partitions, concurrent execution is supported. When DQL/DML and DDL statements are used in the same partition, statements triggered later will be blocked.

NOTICE

If the target partitions of the concurrent DDL and DQL/DML statements overlap, the DQL/DML statements may occur before or after the DDL statements due to serial blocking. You need to know the possible expected results. For example, when TRUNCATE and INSERT take effect on the same partition, if TRUNCATE is triggered before INSERT, data exists in the target partition after the statements are complete. If TRUNCATE is triggered after INSERT, no data exists in the target partition after the statements are complete.

WARNING

During partition-level DDL operations, do not perform DQL/DML operations on the target partition at the same time.

DQL/DML-DDL Concurrency Across Partitions

GaussDB supports DQL/DML-DDL concurrency across partitions.

The following provides some examples of supporting concurrency in the partitioned table **range_sales**.

```
CREATE TABLE range_sales
(
    product_id     INT4 NOT NULL,
    customer_id    INT4 NOT NULL,
    time_id        DATE,
    channel_id     CHAR(1),
    type_id        INT4,
    quantity_sold  NUMERIC(3),
    amount_sold    NUMERIC(10,2)
)
PARTITION BY RANGE (time_id)
(
    PARTITION time_2008 VALUES LESS THAN ('2009-01-01'),
    PARTITION time_2009 VALUES LESS THAN ('2010-01-01'),
    PARTITION time_2010 VALUES LESS THAN ('2011-01-01'),
    PARTITION time_2011 VALUES LESS THAN ('2012-01-01')
);
```

Partitioned tables support the following concurrent statements:

```
-- In case 1, inserting partition time_2011 and truncating partition time_2008 do not block each other.
\parallel on
INSERT INTO range_sales VALUES (455124, 92121433, '2011-09-17', 'X', 4513, 7, 17);
ALTER TABLE range_sales TRUNCATE PARTITION time_2008 UPDATE GLOBAL INDEX;
\parallel off
```

```
-- In case 2, querying partition time_2010 and exchanging partition time_2009 do not block each other.
\parallel on
SELECT COUNT(*) FROM range_sales PARTITION (time_2010);
ALTER TABLE range_sales EXCHANGE PARTITION (time_2009) WITH TABLE temp UPDATE GLOBAL INDEX;
\parallel off

-- In case 3, updating partitioned table range_sales and dropping partition time_2008 do not block each
other. This is because the SQL statement with a condition (partition pruning) updates the time_2010 and
time_2011 partitions only.
\parallel on
UPDATE range_sales SET channel_id = 'T' WHERE channel_id = 'X' AND time_id > '2010-06-01';
ALTER TABLE range_sales DROP PARTITION time_2008 UPDATE GLOBAL INDEX;
\parallel off

-- In case 4, any DQL/DML statement of partitioned table range_sales and adding partition time_2012 do
not block each other. This is because ADD PARTITION is invisible to other statements.
\parallel on
DELETE FROM range_sales WHERE channel_id = 'T';
ALTER TABLE range_sales ADD PARTITION time_2012 VALUES LESS THAN ('2013-01-01');
\parallel off
```

DQL/DML-DDL Concurrency on the Same Partition

GaussDB does not support DQL/DML-DDL concurrency on the same partition. A triggered statement will block the subsequent statements.

In principle, you are advised not to perform DQL/DML operations on a partition when performing DDL operations on the partition. This is because the status of the target partition changes abruptly, which may cause unexpected statement query results.

If the DQL/DML and DDL target partitions overlap due to improper statements or pruning failures, consider the following two scenarios:

Scenario 1: If DQL/DML statements are triggered before DDL statements, DDL statements are blocked until DQL/DML statements are committed.

Scenario 2: if DDL statements are triggered before DQL/DML statements, DQL/DML statements are blocked and are executed after DDL statements are committed. The result may be unexpected. To ensure data consistency, the expected result is formulated based on the following rules:

- **ADD PARTITION**

  During ADD PARTITION, a new partition is generated and is invisible to the triggered DQL/DML statements. There is no blocking.

- **DROP PARTITION**

  During DROP PARTITION, an existing partition is dropped, and the DQL/DML statements triggered on the target partition are blocked. After the blocking is complete, the processing on the partition will be skipped.

- **TRUNCATE PARTITION**

  During TRUNCATE PARTITION, data is cleared from an existing partition, and the DQL/DML statements triggered on the target partition are blocked. After the blocking is complete, the processing on the partition continues.

  Note that no data can be queried in the target partition during this period because no data exists in the target partition after the TRUNCATE operation is committed.

- **EXCHANGE PARTITION**

  The EXCHANGE PARTITION exchanges an existing partition with an ordinary table. During this period, the DQL/DML statements on the target partition are blocked. After the blocking is complete, the partition processing continues. The actual data of the partition corresponds to the original ordinary table.

  Exception: If the global index exists in the partitioned table, the EXCHANGE statement contains the UPDATE GLOBAL INDEX clause, and the partitioned table query triggered during this period uses the global index, the data in the partition after the exchange cannot be queried. As a result, an error is reported during the query after the blocking is complete.

  ERROR: partition xxxxxx does not exist on relation "xxxxxx"

  DETAIL: this partition may have already been dropped by cocurrent DDL operations EXCHANGE PARTITION

- **SPLIT PARTITION**

  The SPLIT PARTITION splits a partition into multiple partitions. Even if a new partition has the same name as the source partition, the new partition is regarded as a different partition. During this period, the DQL/DML statements on the target partition are blocked. After the blocking is complete, an error is reported.

  ERROR: partition xxxxxx does not exist on relation "xxxxxx"

  DETAIL: this partition may have already been dropped by cocurrent DDL operations SPLIT PARTITION

- **MERGE PARTITION**

  The MERGE PARTITION merges multiple partitions into one partition. If the name of the merged partition is the same as that of any of the source partitions, the merged partition is logically considered the same as the source partition. The DQL/DML statements on the target partition triggered during this period are blocked. After the blocking is complete, the system determines whether the target partition is the specified source partition based on the target partitioning type. If the target partition is the specified source partition, the statements take effect on the new partition. If the target partition is another source partition, an error is reported.

  ERROR: partition xxxxxx does not exist on relation "xxxxxx"

  DETAIL: this partition may have already been dropped by cocurrent DDL operations MERGE PARTITION

- **RENAME PARTITION**

  The RENAME PARTITION does not change the partition structure information. The DQL/DML statements triggered during this period do not encounter any exception but are blocked until the RENAME operation is committed.

- **MOVE PARTITION**

  The MOVE PARTITION does not change the partition structure information. The DQL/DML statements triggered during this period do not encounter any exception but are blocked until the MOVE operation is committed.

2.5.4 DDL-DDL Concurrency

GaussDB Kernel does not support concurrent DDL statements. DDL statements triggered later will be blocked.

2.6 System Views & DFX Related to Partitioned Tables

2.6.1 System Views Related to Partitioned Tables

The system views related to partitioned tables are classified into three types based on permissions. For details about the columns, see "System Catalogs and System Views > System Views" in *Developer Guide*.

1. Views related to all partitions:
   - ADM_PART_TABLES: stores information about all partitioned tables.
   - ADM_TAB_PARTITIONS: stores information about all partitions.
   - ADM_TAB_SUBPARTITIONS: stores information about all level-2 partitions.
   - ADM_PART_INDEXES: stores information about all local indexes.
   - ADM_IND_PARTITIONS: stores index partition information about all partitioned tables.
   - ADM_IND_SUBPARTITIONS: stores index partition information about all level-2 partitioned tables.
2. Views accessible to the current user:
   - DB_PART_TABLES: stores information about partitioned tables accessible to the current user.
   - DB_TAB_PARTITIONS: stores information about partitions accessible to the current user.
   - DB_TAB_SUBPARTITIONS: stores information about level-2 partitions accessible to the current user.
   - DB_PART_INDEXES: stores local index information accessible to the current user.
   - DB_IND_PARTITIONS: stores index partition information about partitioned tables accessible to the current user.
   - DB_IND_SUBPARTITIONS: stores index partition information about level-2 partitioned tables accessible to the current user.
3. Views owned by the current user:
   - MY_PART_TABLES: stores information about partitioned tables owned by the current user.
   - MY_TAB_PARTITIONS: stores information about partitions owned by the current user.
   - MY_TAB_SUBPARTITIONS: stores information about level-2 partitions owned by the current user.
   - MY_PART_INDEXES: stores information about local indexes owned by the current user.
   - MY_IND_PARTITIONS: stores index partition information about partitioned tables owned by the current user.
   - MY_IND_SUBPARTITIONS: stores index partition information about level-2 partitioned tables owned by the current user.

2.6.2 Built-in Tool Functions Related to Partitioned Tables

Information About Table Creation

- Create a table.

```
CREATE TABLE test_range_pt (a INT, b INT, c INT)
PARTITION BY RANGE (a)
(
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN (3000),
    PARTITION p3 VALUES LESS THAN (4000),
    PARTITION p4 VALUES LESS THAN (5000),
    PARTITION p5 VALUES LESS THAN (MAXVALUE)
)ENABLE ROW MOVEMENT;
```

- View the OID of the partitioned table.

```
SELECT oid FROM pg_class WHERE relname = 'test_range_pt';
  oid
-------
 49290
(1 row)
```

- View the partition information.

```
SELECT oid,relname,parttype,parentid,boundaries FROM pg_partition WHERE parentid = 49290;
  oid  |    relname    | parttype | parentid | boundaries
-------+---------------+----------+----------+------------
 49293 | test_range_pt | r        |    49290 |
 49294 | p1            | p        |    49290 | {2000}
 49295 | p2            | p        |    49290 | {3000}
 49296 | p3            | p        |    49290 | {4000}
 49297 | p4            | p        |    49290 | {5000}
 49298 | p5            | p        |    49290 | {NULL}
(6 rows)
```

- Create an index.

```
CREATE INDEX idx_range_a ON test_range_pt(a) LOCAL;
CREATE INDEX
-- Check the OID of the partitioned index.
SELECT oid FROM pg_class WHERE relname = 'idx_range_a';
  oid
-------
 90250
(1 row)
```

- View the index partition information.

```
SELECT oid,relname,parttype,parentid,boundaries,indextblid FROM pg_partition WHERE parentid =
90250;
  oid  | relname  | parttype | parentid | boundaries | indextblid
-------+----------+----------+----------+------------+------------
 90255 | p5_a_idx | x        |    90250 |            |      49298
 90254 | p4_a_idx | x        |    90250 |            |      49297
 90253 | p3_a_idx | x        |    90250 |            |      49296
 90252 | p2_a_idx | x        |    90250 |            |      49295
 90251 | p1_a_idx | x        |    90250 |            |      49294
(5 rows)
```

Example of Tool Functions

- **pg_get_tabledef** is used to obtain the definition of a partitioned table. The input parameter can be the table OID or table name.

```
SELECT pg_get_tabledef('test_range_pt');
                        pg_get_tabledef
--------------------------------------------------------------------
 SET search_path = public;                                        +
 CREATE TABLE test_range_pt (                                     +
     a integer,                                           +
     b integer,                                           +
     c integer                                            +
 )                                                    +
 WITH (orientation=row, compression=no)                          +
 PARTITION BY RANGE (a)                                    +
 (                                                    +
     PARTITION p1 VALUES LESS THAN (2000) TABLESPACE pg_default,   +
     PARTITION p2 VALUES LESS THAN (3000) TABLESPACE pg_default,   +
     PARTITION p3 VALUES LESS THAN (4000) TABLESPACE pg_default,   +
     PARTITION p4 VALUES LESS THAN (5000) TABLESPACE pg_default,   +
     PARTITION p5 VALUES LESS THAN (MAXVALUE) TABLESPACE pg_default+
 )                                                    +
 ENABLE ROW MOVEMENT;
(1 row)
```

- **pg_stat_get_partition_tuples_hot_updated** is used to return the number of hot updated tuples in a partition with a specified partition ID.

  Insert 10 data records into partition **p1** and update the data. Count the number of hot updated tuples in partition **p1**.

```
INSERT INTO test_range_pt VALUES(generate_series(1,10),1,1);
INSERT 0 10
SELECT pg_stat_get_partition_tuples_hot_updated(49294);
pg_stat_get_partition_tuples_hot_updated
------------------------------------------
0
(1 row)
UPDATE test_range_pt SET b = 2;
UPDATE 10
SELECT pg_stat_get_partition_tuples_hot_updated(49294);
pg_stat_get_partition_tuples_hot_updated
------------------------------------------
10
(1 row)
```

- **pg_partition_size(oid,oid)** is used to specify the disk space used by the partition with a specified OID. The first **oid** is the OID of the table and the second **oid** is the OID of the partition.

  Check the disk space of partition **p1**.

```
SELECT pg_partition_size(49290, 49294);
pg_partition_size
-------------------
90112
(1 row)
```

- **pg_partition_size(text, text)** is used to specify the disk space used by the partition with a specified name. The first **text** is the table name and the second **text** is the partition name.

  Check the disk space of partition **p1**.

```
SELECT pg_partition_size('test_range_pt', 'p1');
pg_partition_size
-------------------
90112
(1 row)
```

- **pg_partition_indexes_size(oid,oid)** is used to specify the disk space used by the index of the partition with a specified OID. The first **oid** is the OID of the table and the second **oid** is the OID of the partition.

  Check the disk space of the index partition of partition **p1**.

```
SELECT pg_partition_indexes_size(49290, 49294);
pg_partition_indexes_size
---------------------------
204800
(1 row)
```

- **pg_partition_indexes_size(text,text)** is used to specify the disk space used by the index of the partition with a specified name. The first **text** is the table name and the second **text** is the partition name.

  Check the disk space of the index partition of partition **p1**.

  ```
  SELECT pg_partition_indexes_size('test_range_pt', 'p1');
  pg_partition_indexes_size
  ---------------------------
  204800
  (1 row)
  ```

- **pg_partition_filenode(partition_oid)** is used to obtain the file node corresponding to the OID of the specified partitioned table.

  Check the file node of partition **p1**.

  ```
  SELECT pg_partition_filenode(49294);
  pg_partition_filenode
  -----------------------
  49294
  (1 row)
  ```

- **pg_partition_filepath(partition_oid)** is used to specify the file path name of the partition.

  Check the file path of partition **p1**.

  ```
  SELECT pg_partition_filepath(49294);
  pg_partition_filepath
  -----------------------
  base/16521/49294
  (1 row)
  ```

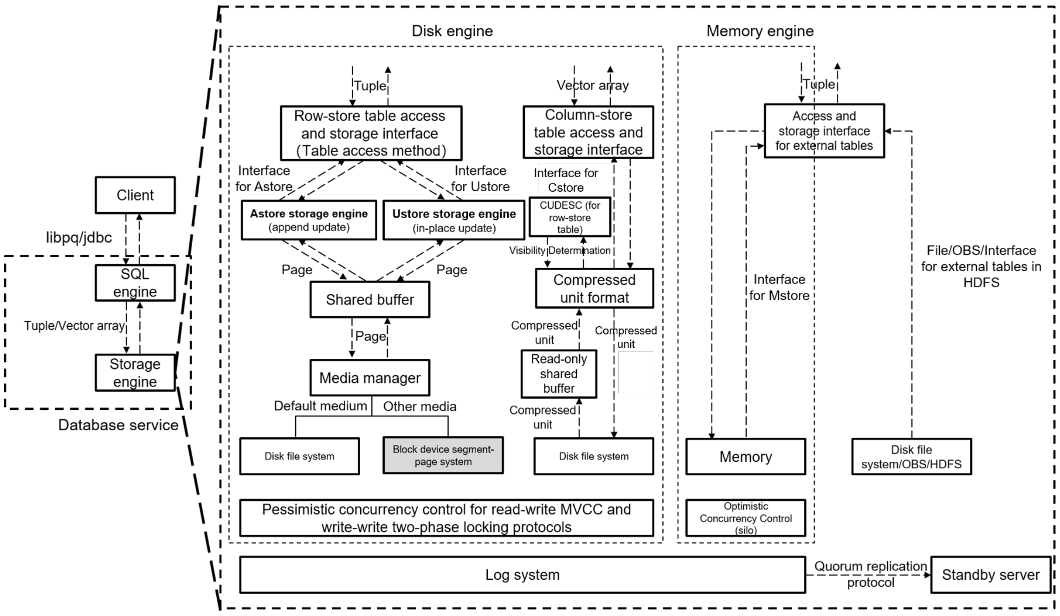
3 Storage Engine

3.1 Storage Engine Architecture

3.1.1 Overview

3.1.1.1 Static Compilation Architecture

From the perspective of the entire database service architecture, the storage engine upward connects to the SQL engine to provide or receive data in a standard format (tuple or vector array) for or from the SQL engine, and downward reads data from or writes data to storage media by a specific data organization mode such as page, compress unit, or other forms through specific APIs provided by the storage media. GaussDB Kernel enables database professionals to select dedicated storage engines for meeting specific application requirements through static compilation. To reduce interference to the execution engines, the row-store table access method (TableAM) layer is provided to shield the differences caused by the underlying row-store engines so that different row-store engines can evolve independently. See the following figure.


Client
libpq/jdbc
SQL engine
Tuple/Vector array
Storage engine
Database service
Disk engine
Memory engine
Tuple
Row-store table access and storage interface (Table access method)
Interface for Astore
Interface for Ustore
Astore storage engine (append update)
Ustore storage engine (in-place update)
Page
Shared buffer
Page
Media manager
Default medium
Other media
Disk file system
Block device segment-page system
Vector array
Column-store table access and storage interface
Interface for Cstore
CUDESC (for row-store table)
Visibility Determination
Compressed unit format
Compressed unit
Read-only shared buffer
Compressed unit
Disk file system
Pessimistic concurrency control for read-write MVCC and write-write two-phase locking protocols
Access and storage interface for external tables
Interface for Mstore
File/OBS/Interface for external tables in HDFS
Memory
Disk file system/OBS/HDFS
Optimistic Concurrency Control (silo)
Log system
Quorum replication protocol
Standby server

On this basis, the storage engines provide data persistence and reliability capabilities through the log system. The concurrency control (transaction) system ensures atomicity, consistency, and isolation between multiple read and write operations that are executed at the same time. The index system provides accelerated addressing and query capabilities for specific data. The primary/ standby replication system provides high availability of the entire database service.

Row-store engines are oriented to online transaction processing (OLTP) scenarios, which are suitable for highly concurrent read and write operations on a small amount of data at a single point or within a small range. Row-store engines provide APIs upward to read tuples from or write tuples to the SQL engine, perform read and write operations downward on storage media by page through an extensible media manager, and improve read and write operation efficiency in the shared buffer by page. For concurrent read and write operations, multi-version concurrency control (MVCC) is used. For concurrent write and write operations, pessimistic concurrency control (PCC) based on the two-phase locking (2PL) protocol is used. Currently, the default media manager of row-store engines uses the disk file system API. Other types of storage media such as block devices will be supported in the future. The GaussDB Kernel row-store engine can be the append update-based Astore or in-place update-based Ustore.

3.1.1.2 Database Service Layer

From the technical perspective, a storage engine requires some infrastructure components.

Concurrency: The overhead of a storage engine can be reduced by properly employing locks, so as to improve overall performance. In addition, it provides functions such as multi-version concurrency control and snapshot reading.

Transaction: All transactions must meet the ACID requirements and their statuses can be queried.

Memory cache: Typically, storage engines cache indexes and data when accessing them. You can directly process common data in the cache pool, which facilitates the handling speed.

Checkpoint: Though storage engines are different, they all support incremental checkpoint/double write and full checkpoint/full page write. For different applications, you can select incremental checkpoint/double write or full checkpoint/full page write based on different conditions, which is transparent to storage engines.

Log: GaussDB Kernel uses physical logs. The write, transmission, and replay operations of physical logs are transparent to the storage engine.

3.1.2 Setting Up a Storage Engine

The storage engine has a great impact on the overall efficiency and performance of the database. Select a proper storage engine based on the actual requirements. You can run **WITH ([ORIENTATION | STORAGE_TYPE] [= value] [, ...])** to specify an optional storage parameter for a table or index. The parameters are described as follows.

ORIENTATION	STORAGE_TYPE
ROW (default value): The data will be stored in rows.	[USTORE (default value)|ASTORE|Null]

If **ORIENTATION** is set to **ROW** and **STORAGE_TYPE** is left empty, the type of the created table is determined by the value of the **enable_default_ustore_table** parameter. The parameter value can be **on** or **off**. The default value is **on**. If this parameter is set to **off**, an Astore table is created.

Example:

```
gaussdb=# CREATE TABLE TEST(a int);
gaussdb=# \d+ test
                     Table "public.test"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 a      | integer |           | plain   |              |
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE

gaussdb=# CREATE TABLE TEST1(a int) with(orientation=row, storage_type=ustore);
gaussdb=# \d+ test1
Table "public.test1"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 a      | integer |           | plain   |              |
Has OIDs: no
Options: orientation=row, storage_type=ustore, compression=no

gaussdb=# CREATE TABLE TEST2(a int) with(orientation=row, storage_type=astore);
gaussdb=# \d+ test2
Table "public.test2"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 a      | integer |           | plain   |              |
Has OIDs: no
Options: orientation=row, storage_type=astore, compression=no

gaussdb=# create table test4(a int) with(orientation=row);
gaussdb=# \d+
                                   List of relations
 Schema | Name  | Type  |   Owner   |  Size   |                  Storage                  | Description
```

```
--------+-------+-------+-----------+---------+-------------------------------------------------------+-------------
 public | test4 | table | l30048445 | 0 bytes | {orientation=row,compression=no,storage_type=USTORE} |
(1 row)

gaussdb=# show enable_default_ustore_table;
 enable_default_ustore_table
-----------------------------
 on
(1 row)
```

3.1.3 Storage Engine Update Description

3.1.3.1 GaussDB 503

- Adapted Ustore to distributed deployment, parallel query, global temporary tables, VACUUM FULL, and column constraints DEFERRABLE and INITIALLY DEFERRED.

- Added the online index rebuild function to Ustore.

- Enhanced B-tree empty page estimation for Ustore to improve the cost estimation accuracy of an optimizer.

- Added the storage engine reliability verification framework Diagnose Page/Page Verify to Ustore.

- Enhanced the view parsing, detection, and repair related to Ustore.

- Enhanced the WAL locating capability for Ustore. The gs_redo_upage system view is added to support constant replay of a single page and obtain and print any historical page, accelerating fault locating for damaged pages.

- Extended the Ustore transaction directory's physical format for transaction slots for space reuse within a transaction.

- Added the online index creation function for Ustore.

- Adapted Ustore to the flashback function and ultimate RTO.

3.1.3.2 GaussDB R2

- Added the Ustore row storage engine based on in-place update to implement separate storage of new and old data.

- Added rollback segments to Ustore.

- Added the synchronous, asynchronous, and in-page rollback to Ustore.

- Enhanced Ustore B-tree indexes for transactions.

- Added the flashback function to Astore to support table flashback, flashback query, flashback DROP, and flashback TRUNCATE.

- Ustore does not support the following features: parallel query, table sampling, global temporary table, online creation, index rebuild, ultimate RTO, VACUUM FULL, and column constraints DEFERRABLE and INITIALLY DEFERRED.

3.2 Astore

3.2.1 Overview

The biggest difference between Astore and Ustore lies in whether the latest data and historical data are stored separately. Astore does not perform separated storage. Ustore only separates data, but does not separate indexes.

Astore Advantages

1. Astore does not have rollback segments, but Ustore does. For Ustore, rollback segments are very important. If rollback segments are damaged, data will be lost or even the database cannot be started. In addition, redo and undo operations are required for Ustore restoration. For Astore, it does not have a rollback segment, therefore, old data is stored in the original files, whose restoration is not as complex as that of Ustore.
2. Besides, the error "Snapshot Too Old" is not frequently reported, because old data is directly recorded in data files instead of rollback segments.
3. The rollback operation can be completed quickly since no data needs to be deleted. However, the rollback operation is complex, because the modifications and the inserted records must be deleted, and the updated records must be undone. In addition, a large number of redo logs are generated during rollback.
4. WAL in Astore is simpler than that in Ustore. Only data file changes need to be recorded in WALs. Rollback segment changes do not need to be recorded.

3.3 Ustore

3.3.1 Overview

Ustore is an in-place update storage engine launched by GaussDB. The biggest difference between Ustore and Astore lies in that, the latest data and historical data (excluding indexes) are stored separately. Now, Ustore is a default row-store engine of GaussDB.

Ustore Advantages

1. The latest data and historical data are stored separately. Compared with Astore, Ustore has a smaller scanning scope. The HOT chain of Astore is removed. Non-index columns, index columns, and heaps can be updated in-place without change to row IDs. Historical data can be recycled in batches, which is friendly to the expansion of the latest data.
2. If the same row is updated in a large concurrency, the in-place update mechanism of Ustore ensures the stability of tuple row IDs and update latency.
3. VACUUM is not the only way to clear historical data. Indexes are decoupled from heaps and can be cleared separately with good I/O stability.

4. The flashback function is supported.

However, in addition to modifying data pages, Ustore DML operations also modify undo logs. Therefore, the update overhead is higher. In addition, the scanning overhead of a single tuple is high because of replication (Astore returns pointers).

3.3.1.1 Ustore Features and Specifications

3.3.1.1.1 Restrictions

Category	Feature	Supported or Not
Transaction	Serializable	×
	DDL operations on a partitioned table in a transaction block	×
Scalability	Hash bucket	×
SQL	Table sampling/Materialized view/Key-value lock	×

3.3.1.1.2 Storage Specifications

1. The maximum number of columns in a data table is 1600.
2. The maximum tuple length of a Ustore table (excluding toast) cannot exceed 8192 – MAXALIGN(56 + init_td x 26 + 4), where **MAXALIGN** indicates 8-byte alignment. When the length of the inserted data exceeds the threshold, you will receive an error reporting that the tuple is too long to be inserted. The impact of **init_td** on the tuple length is as follows:
   – If the value of **init_td** is the minimum value **2**, the tuple length cannot exceed 8192 – MAXALIGN(56 + 2 x 26 + 4) = 8080 bytes.
   – If the value of **init_td** is the default value **4**, the tuple length cannot exceed 8192 – MAXALIGN(56 + 4 x 26 + 4) = 8024 bytes.
   – If the value of **init_td** is the maximum value **128**, the tuple length cannot exceed 8192 – MAXALIGN(56 + 128 x 26 + 4) = 4800 bytes.
3. The value range of **init_td** is [2,128], and the default value is **4**. A single page supports a maximum of 128 concurrent transactions.
4. The maximum number of index columns is 32. The maximum number of columns in a global partitioned index is 31.
5. The length of an index tuple cannot exceed (8192 – MAXALIGN(28 + 3 x 4 + 3 x 10) – MAXALIGN(42))/3, where **MAXALIGN** indicates 8-byte alignment. When the length of the inserted data exceeds the threshold, you will receive an error reporting that the tuple is too long to be inserted. As for the threshold, the index page header is 28 bytes, row pointer is 4 bytes, tuple CTID+INFO flag is 10 bytes, and page tail is 42 bytes.
6. The maximum rollback segment size is 16 TB.

3.3.1.2 Example

Create a Ustore table.

Run the **CREATE TABLE** statement to create a Ustore table.

```
gaussdb=# CREATE TABLE ustore_table(a INT PRIMARY KEY, b CHAR (20)) WITH (STORAGE_TYPE=USTORE);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "ustore_table_pkey" for table
"ustore_table"
CREATE TABLE
gaussdb=# \d+ ustore_table
Table "public.ustore_table"
Column |     Type      | Modifiers | Storage  | Stats target | Description
--------+---------------+-----------+----------+--------------+-------------
a      | integer       | not null  | plain    |              |
b      | character(20) |           | extended |              |
Indexes:
"ustore_table_pkey" PRIMARY KEY, ubtree (a) WITH (storage_type=USTORE) TABLESPACE pg_default
Has OIDs: no
Options: orientation=row, storage_type=ustore, compression=no
```

Create an index for a Ustore table.

Currently, Ustore supports only multi-version B-tree indexes. In some scenarios, to distinguish them from Astore B-tree indexes, a multi-version B-tree index of the Ustore table is also called a Ustore B-tree or UB-tree. For details about UB-tree, see **Index**. You can run the **CREATE INDEX** statement to create a UB-tree index for the "a" attribute of a Ustore table.

If no index type is specified for a Ustore table, a UB-tree index is created by default.

```
gaussdb=# CREATE INDEX UB-tree_index ON ustore_table(a);
CREATE INDEX
gaussdb=# \d+ ustore_table
Table "public.ustore_table"
Column |     Type      | Modifiers | Storage  | Stats target | Description
--------+---------------+-----------+----------+--------------+-------------
a      | integer       | not null  | plain    |              |
b      | character(20) |           | extended |              |
Indexes:
"ustore_table_pkey" PRIMARY KEY, ubtree (a) WITH (storage_type=USTORE) TABLESPACE pg_default
"ubtree_index" ubtree (a) WITH (storage_type=USTORE) TABLESPACE pg_default
Has OIDs: no
Options: orientation=row, storage_type=ustore, compression=no
```

3.3.1.3 Best Practices of Ustore

3.3.1.3.1 How Can I Configure init_td

Transaction directory (TD) is a unique structure used by Ustore tables to store page transaction information. The number of TDs determines the maximum number of concurrent transactions supported on a page. When creating a table or index, you can specify the initial TD size **init_td**, whose default value is **4**. That is, four concurrent transactions are supported to modify the page. The maximum value of **init_td** is **128**.

You can configure **init_td** based on the service concurrency requirements. Besides, you can also configure it based on the occurrence frequency of **wait available td** events during service running. Generally, the value of **wait available td** is **0**. If the value of **wait available td** is not **0**, there are events waiting for available TDs. In this case, you are advised to increase the value of **init_td**. If the value **0** is an occasional situation, you are advised not to adjust **init_td** because extra TD slots occupy more space. You are advised to gradually increase the value in ascending order, such as 8, 16, 32, 48, ..., and 128, and check whether the number of wait events decreases significantly in this process. Use the minimum value of **init_td** with few wait events as the default value to save space. For details about how to configure and modify **init_td**, see "SQL Reference > SQL Syntax > CREATE TABLE" in *Developer Guide*.

3.3.1.3.2 How Can I Configure fillfactor

fillfactor is a parameter used to describe the page filling rate and is directly related to the number and size of tuples that can be stored on a page and the physical space of a table. The default page filling rate of Ustore tables is 92%. The reserved 8% space is used for page update and TD list expansion. For details about how to configure and modify **fillfactor**, see "SQL Reference > SQL Syntax > CREATE TABLE" in *Developer Guide*.

You can configure **fillfactor** after analyzing services. If only query or fixed-length update operations are performed after table data is imported, you can increase the page filling rate to 100%. If a large number of fixed-length updates are performed after data is imported, you are advised to retain or decrease the page filling rate to reduce performance loss caused by cross-page update.

3.3.1.3.3 Collecting Statistics

Clearing invalid tuples in Ustore tables depends on the accuracy of statistics. Disabling **track_counts** and **track_activities** will cause tablespace bloat. By default, they are enabled. You are advised to enable them, except in performance-sensitive scenarios.

To enable them, run the following commands:

```
gs_guc reload -Z datanode -N all -I all -c "track_counts=on;"
gs_guc reload -Z datanode -N all -I all -c "track_activities=on;"
```

To disable them, run the following commands:

```
gs_guc reload -Z datanode -N all -I all -c "track_counts=off;"
gs_guc reload -Z datanode -N all -I all -c "track_activities=off;"
```

3.3.1.3.4 Online Verification

Online verification is unique to Ustore. It can effectively prevent logic damage on a page caused by encoding logic errors during running. By default, it is enabled. Keep it enabled on the live network, except in performance-sensitive scenarios.

To disable it, run the following command:

```
gs_guc reload -Z datanode -N all -I all -c "ustore_attr='';"
```

To enable it, run the following command:

```
gs_guc reload -Z datanode -N all -I all -c
"ustore_attr='ustore_verify_level=fast;ustore_verify_module=upage:ubtree:undo"
```

3.3.1.3.5 How Can I Configure the Size of Rollback Segments

Generally, use the default size of rollback segments. To achieve optimal performance, you can adjust the parameters related to the rollback segment size in some scenarios. The scenarios and corresponding configurations are as follows:

1. Historical data within a specified period needs to be retained.

   To use flashback or locate faults, you can change the value of **undo_retention_time** to retain more historical data. The default value of **undo_retention_time** is **0**. The value ranges from 0 to 3 days.

   You are advised to set it to **900s**. Note that a larger value of **undo_retention_time** indicates more undo space usage and data space bloat, which further affects the data scanning and update performance. When flashback is not used, you are advised to set **undo_retention_time** to a smaller value to reduce the disk space occupied by historical data and achieve optimal performance. You can use the following method to determine the new value of **undo_retention_time** that is more suitable for your service model:

   new_val = 0.5 x (undo_space_limit_size x 0.8 – curr_used_undo_size)/ avg_space_increase_speed, where **avg_space_increase_speed** is the recent average growth speed of the undo space and **curr_used_undo_size** is the current undo space and both of them can be queried in the gs_stat_undo view.

2. Historical data within a specified size needs to be retained.

   If long transactions or large transactions exist in your service, undo space may bloat. In this case, you need to increase the value of **undo_space_limit_size**. The default value of **undo_space_limit_size** is **256GB**, and the value ranges from 800 MB to 16 TB.

   If the disk space is sufficient, you are advised to double the value of **undo_space_limit_size**. In addition, a larger value of **undo_space_limit_size** indicates more disk space occupation and deteriorated performance. If no undo space bloat is found by querying **curr_used_undo_size** of **gs_stat_undo()**, you can restore the value to the original value.

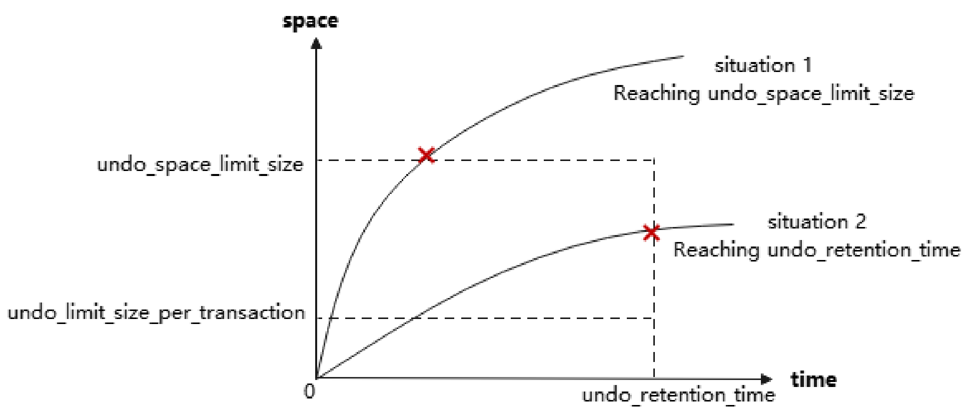
   After adjusting the value of **undo_space_limit_size**, you can increase the value of **undo_limit_size_per_transaction**, which ranges from 2 MB to 16 TB. The default value is **32GB**. It is recommended that the value of **undo_limit_size_per_transaction** be less than or equal to that of **undo_space_limit_size**, that is, the threshold of the undo space allocated to a single transaction be less than or equal to the threshold of the total undo space.

   To accurately set this parameter to achieve optimal performance, you are advised to determine the new value by using the following methods:

   – **undo_space_limit_size**: new_val = 86400 x 30 x avg_space_increase_speed + curr_used_undo_size, where **avg_space_increase_speed** and **curr_used_undo_size** can be queried in the gs_stat_undo view.
   – **undo_limit_size_per_transaction**: new_val = 10 x max_xact_space, where **max_xact_space** indicates the maximum undo space occupied by a single transaction and can be queried in the gs_stat_undo() view in the 503.2 version.

3. The parameter adjustment priority is retained for historical data.

   If any of **undo_retention_time**, **undo_space_limit_size** and **undo_limit_size_per_transaction** is reached, the corresponding restriction is triggered.

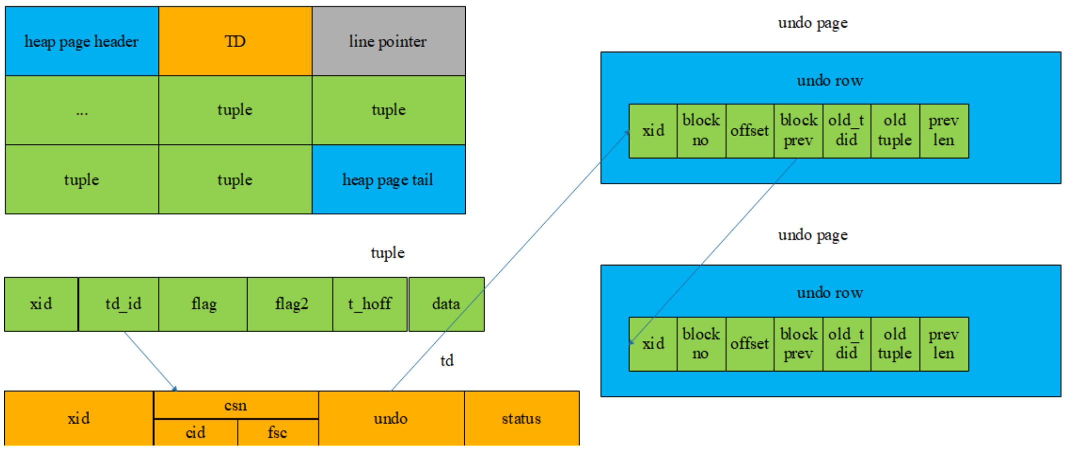


For example, assume that **undo_space_limit_size** is set to **1GB**, and **undo_retention_time** is set to **900s**. If the size of historical data generated within 900s is less than 1 GB x 0.8, the system recycles the data generated within 900s. If the data exceeds 1 GB x 0.8 generated within 900s, only 1 GB x 0.8 data will be recycled. In this case, if the disk space is sufficient, you can increase the value of **undo_space_limit_size**. If not, decrease the value of **undo_retention_time**.

3.3.2 Storage Format

3.3.2.1 Relation

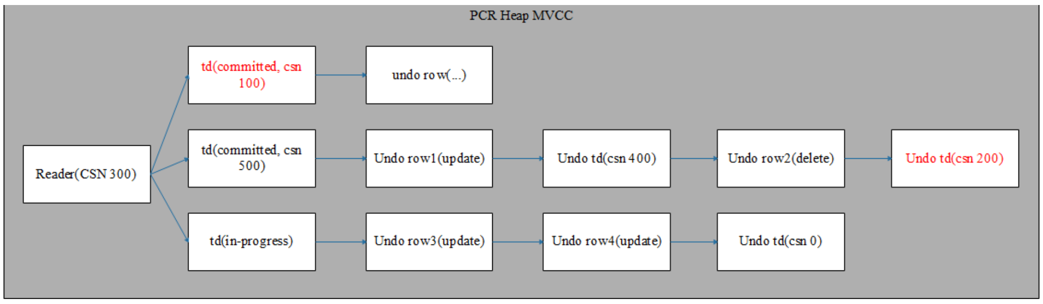
3.3.2.1.1 Page-based Row Consistency Read (PbRCR) Heap Multi-Version Management



1. The heap multi-version management is row-level multi-version management based on tuples.
2. When a transaction modifies a record, historical data is recorded in an undo row.
3. The address of the generated undo row (zone_id, block no, page offset) is recorded in **td_id** in a tuple.
4. New data is overwritten to the heap page.

5. Each data modification generates an undo row. Undo rows with the same record is connected by **block_prev**.

3.3.2.1.2 PbPCR Heap Visibility Mechanism



1. Currently, only row consistency read is supported. In the future, CR page construction and page consistency read will be supported, greatly improving the sequence scanning efficiency.
2. Space can be reused after data deletion transactions are committed without waiting for oldestxmin, increasing the space utilization. Old snapshots can be obtained from undo records.

3.3.2.1.3 Heap Space Management

Ustore uses the free space map (FSM) file to record the free space of each data page and organizes it in the tree structure. When you want to perform insert operations or non-in-place update operations on a table, search an FSM file corresponding to the table to check whether the maximum free space recorded in current FSM file meets the requirement of the insert operation. If so, perform the insert operation after the corresponding block number is returned. If not, expand the page logic.

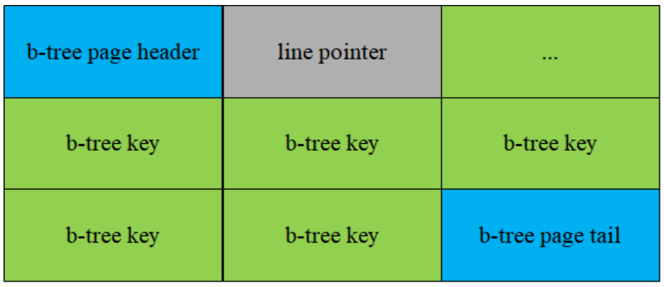
The FSM structure corresponding to each table or partition is stored in an independent FSM file. The FSM file and the table data are stored in the same directory. For example, if the data file corresponding to table **t1** is **32181**, the corresponding FSM file is **32181_fsm**. FSM is stored in the format of data blocks, which are called FSM block. The logical structure among FSM blocks is a tree with three layers of nodes. The nodes of the tree in logic are max heaps. Each searching on FSM starts from the root node to leaf nodes to search for and return an available page for the following operations. This structure may not keep real-time consistency with the actual available space of data pages and is maintained during DML execution. Ustore occasionally repairs and rebuilds FSM during the automatic vacuum process.

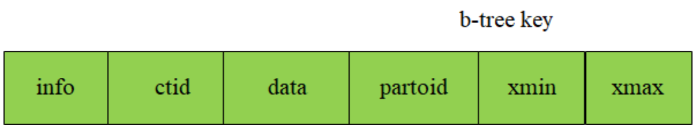
3.3.2.2 Index

The UB-tree is enhanced as follows:

1. Added the MVCC capability.
2. Added the capability of recycling independent empty pages.

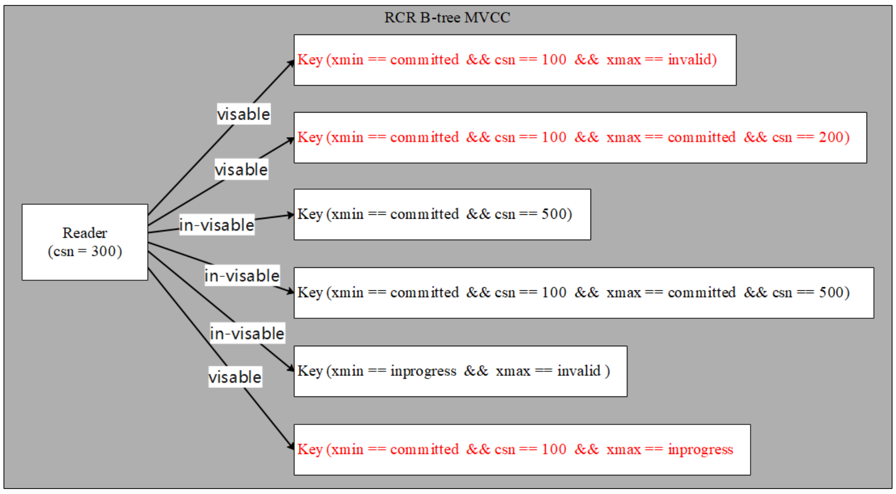
3.3.2.2.1 Row Consistency Read (RCR) UB-Tree Multi-Version Management





1. The UB-tree multi-version management adopts the key-based multi-version management. The latest version and historical versions are both on UB-tree.
2. To save the space, xmin/xmax is expressed in xid-base + delta. The 64-bit xid-base is stored on pages and the 32-bit delta is stored on tuples. The xid-base on pages also needs to be maintained through additional logic.
3. Keys are inserted into or deleted from the UB-tree in the sequence of key + TID. Tuples with the same index column are sorted based on their TIDs as the second keywords. The **xmin** and **xmax** are added to the end of the key.
4. During index splitting, multi-version information is migrated with key migration.

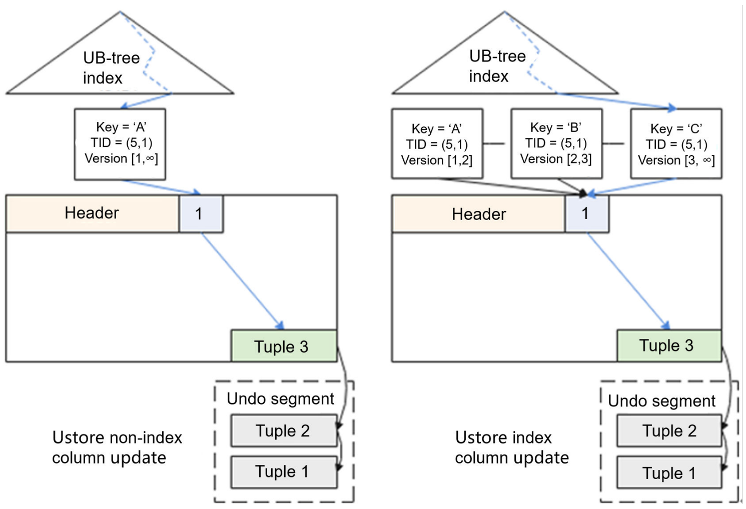
3.3.2.2.2 RCR UB-Tree Visibility Mechanism


RCR B-tree MVCC
Reader (csn = 300)
visable
visable
in-visable
in-visable
in-visable
visable
Key (xmin == committed && csn == 100 && xmax == invalid)
Key (xmin == committed && csn == 100 && xmax == committed && csn == 200)
Key (xmin == committed && csn == 500)
Key (xmin == committed && csn == 100 && xmax == committed && csn == 500)
Key (xmin == inprogress && xmax == invalid)
Key (xmin == committed && csn == 100 && xmax == inprogress

1. Multi-version management and visibility check of index data are supported to identify tuples of historical versions and recycle them. In addition, the visibility check at the index layer greatly improves the probability of index scan and index-only scan.
2. In addition to the index insertion operation, an index deletion operation is added to mark an index tuple corresponding to a deleted or modified tuple.

3.3.2.2.3 Inserting, Deleting, Updating, and Scanning UB-Tree

- **Insert**: The insertion logic of UB-tree is basically not changed, except that you need to directly obtain the transaction information and fill in the **xmin** column during index insertion.
- **Delete**: The index deletion process is added for UB-tree. The main procedure of index deletion is similar to that of index insertion. That is, obtain the transaction information, fill in the xmax column (The B-tree index does not maintain the version information so that the deletion operation is required.), and update **active_tuple_count** on pages. If **active_tuple_count** is reduced to **0**, the system attempts to recycle the page.
- **Update**: In Ustore, data updates require different processing on UB-tree index columns compared to Astore. Data updates include index column updates and non-index column updates. The following figure illustrates the UB-tree processing during data updates.


UB-tree index
Key = 'A' TID = (5,1) Version [1,∞]
Header
1
Tuple 3
Undo segment
Tuple 2
Tuple 1
Ustore non-index column update
UB-tree index
Key = 'A' TID = (5,1) Version [1,2]
Key = 'B' TID = (5,1) Version [2,3]
Key = 'C' TID = (5,1) Version [3, ∞]
Header
1
Tuple 3
Undo segment
Tuple 2
Tuple 1
Ustore index column update

The preceding figure shows the differences between UB-tree updates in index columns and non-index columns.

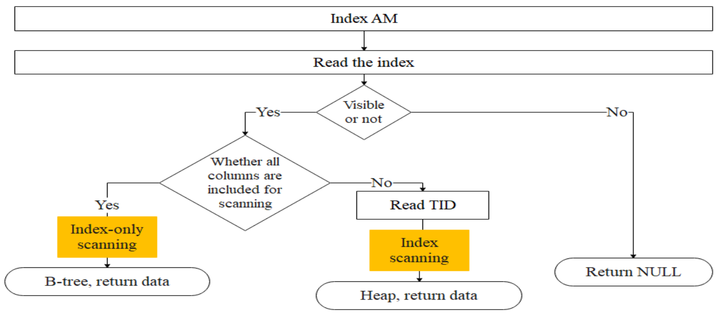
a. When a non-index column is updated, the index does not change. The index tuple points to the data tuple inserted at the first time. The Uheap does not insert a new data tuple. Instead, the Uheap modifies the current data tuple and saves historical data to the undo segment.
b. When the index column is updated, a new index tuple is inserted into UB-tree and points to the same data linepointer and data tuple. To scan the data of historical versions, you need to read it from the undo segment.

- **Scan**: When reading data, you can use index to speed up scanning. UB-tree supports multi-version management and visibility check of index data. The visibility check at the index layer improves the performance of index scan and index-only scan.

  For index scan:

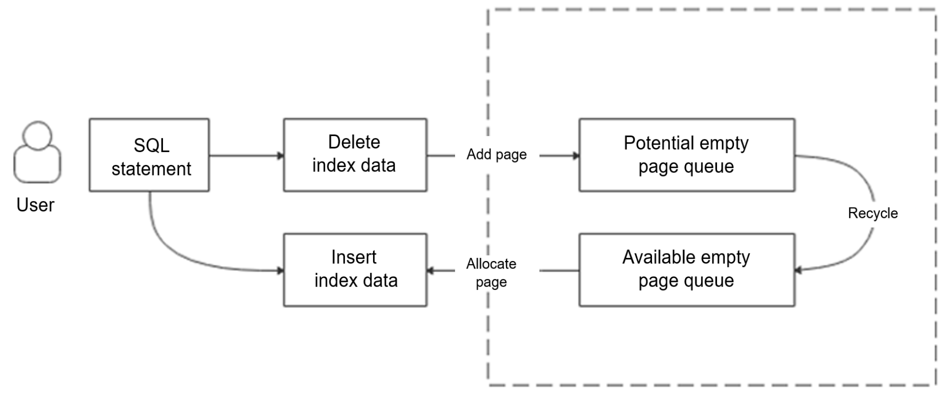
  a. If the index column contains all columns to be scanned (index-only scan), binary search is performed on indexes based on the scan conditions. If a tuple that meets the conditions is found, data is returned.
  b. If the index column does not contain all columns to be scanned (index scan), binary search is performed on indexes based on the scan conditions to find TIDs of the tuples that meet the conditions, and then

the corresponding data tuples are found in data tables based on the TIDs. See the following figure.



3.3.2.2.4 UB-Tree Space Management

Currently, Astore indexes depend on AutoVacuum and FSM for space management. The space may not be recycled in a timely manner. However, Ustore indexes use the UB-tree recycle queue (URQ) to manage idle index space. The URQ contains two circular queues: potential empty page queue and available empty page queue. Completing space management of indexes in a DML process can effectively alleviate the sharp space expansion caused during the DML process. Index recycle queues are separately stored in FSM files corresponding to the B-tree indexes.



As shown in the preceding figure, the index page flow in the URQ is as follows:

1. **Index empty page > Potential queue**

   The index page tail column records the number of active tuples (activeTupleCount) on the page. During the DML process, all tuples on a page are deleted, that is, when **activeTupleCount** is set to **0**, the index page is placed in the potential queue.

2. **Potential queue > Available queue**

   The flow from a potential queue to an available queue mainly achieves an income and expense balance for the potential queue and ensure that pages are available for the available queue. That is, after an index empty page is used up in an available queue, at least one index page is transferred from a potential queue to the available queue. Besides, if a new index page is added to a potential queue, at least one index page can be removed from the potential queue and inserted into the available queue.

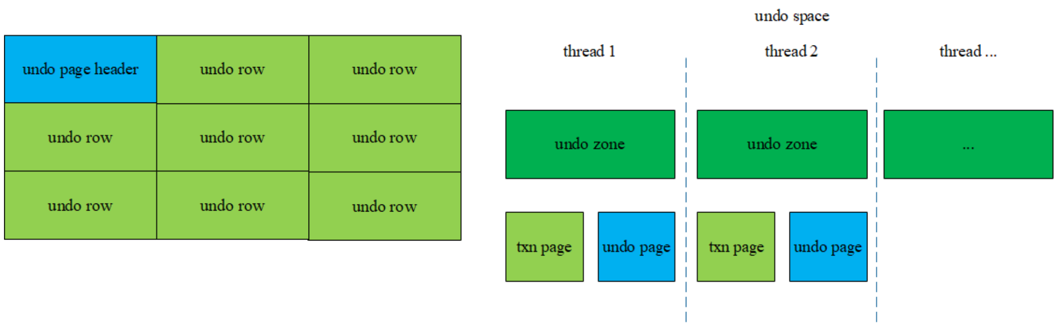
3. **Available queue > Index empty page**

   When an empty index page is obtained during index splitting, the system first searches an available queue for an index page that can be reused. If such index page is found, it is directly reused. If no index page can be reused, physical page expansion is performed.

3.3.2.3 Undo

Data of historical versions is stored in the **$GAUSS_HOME/undo** directory. The rollback segment log is a collection of all undo logs associated with a single write transaction. Permanent, unlogged, and temp tables are supported.

3.3.2.3.1 Rollback Segment Management



1. Each undo zone manages some txn pages and undo pages.
2. Undo rows are stored on undo pages. Therefore, the modified data of historical versions is recorded on the undo pages.
3. Records on the undo pages are also data. Therefore, modifications on the undo pages are also recorded on the redo pages.

3.3.2.3.2 File Structure

Structure of the file where the txn page is stored

```
$GAUSS_HOME/undo/{permanent|unlogged|temp}/$undo_zone_id.meta.$segno
```

Structure of the file where the undo row is stored

```
$GAUSS_HOME/undo/{permanent|unlogged|temp}/$undo_zone_id.$segno
```

3.3.2.3.3 Undo Space Management

The undo subsystem relies on the backend recycle thread to recycle free space. It recycles the space of the undo module on the primary node. As for the standby node, it recycles the space by replaying the Xlog. The recycle thread traverses the undo zones in use. The txn pages in the undo zone are scanned in the ascending order of XIDs. The transactions that have been committed or rolled back are also recycled. The commit time of transactions must be earlier than *$* (current_time – undo_retention_time). For a transaction that needs to be rolled back during a traversal, the recycle thread adds an asynchronous rollback task for the transaction.

When the database has transactions that run for a long time and contain a large amount of modified data, or it takes a long time to enable flashback, the undo space may continuously expand. When the undo space is close to the value specified by **undo_space_limit_size**, forcible recycling is triggered. As long as a transaction has been committed or rolled back, the transaction may be recycled even if it is committed later than *$* (current_time – undo_retention_time).

3.3.3 Ustore Transaction Model

GaussDB transaction basis:

1. An XID is not automatically allocated when a transaction is started, unless the first DML/DDL statement in the transaction is executed.
2. When a transaction ends, a commit log (Clog) indicating the transaction commit state is generated. The states can be IN_PROGRESS, COMMITTED, ABORTED, or SUB_COMMITTED. Each transaction has two Clog status bits. Each byte on the Clog page indicates four transaction commit states.
3. When a transaction ends, a commit sequence number (CSN) is generated, which is an instance-level variable. Each XID has its unique CSN. The CSN can mark the following transaction states: IN_PROGRESS, COMMITTED, ABORTED, or SUB_COMMITTED.

3.3.3.1 Transaction Commit

1. Implicit transaction. A single DML/DDL statement can automatically trigger an implicit transaction, which does not have explicit transaction block control statements (such as START TRANSACTION/BEGIN/COMMIT/END). After a DML/DDL statement ends, the transaction is automatically committed.
2. Explicit transaction. An explicit transaction uses an explicit statement, such as START TRANSACTION or BEGIN, to control the start of the transaction. The COMMIT and END statements control the commit of a transaction.

   Sub-transactions must be in explicit transactions or stored procedures. The SAVEPOINT statement controls the start of sub-transactions, and the RELEASE SAVEPOINT statement controls the end of sub-transactions. If sub-transactions that are not released during transaction committing, the sub-transactions are committed before the transaction is committed.

   Ustore supports READ COMMITTED. At the beginning of statement execution, the current system CSN is obtained for querying the current statement. The visible result of the entire statement is determined at the beginning of statement execution and is not affected by subsequent transaction modifications. By default, READ COMMITTED in the Ustore is consistent. Ustore also supports standard 2PC transactions.

3.3.3.2 Transaction Rollback

Rollback is a process in which a transaction cannot be executed if a fault occurs during transaction running. In this case, the system needs to cancel the modification operations that have been completed in the transaction. Astore and UB-tree do not have rollback segments. Therefore, there is no dedicated rollback operation. To ensure performance, the Ustore rollback process supports synchronous, asynchronous, and in-page instant rollback.

1. **Synchronous rollback.**

   Transaction rollback is triggered in any of the following scenarios:

   a. The ROLLBACK keyword in a transaction block triggers a synchronous rollback.
   b. If an error is reported during transaction running, the COMMIT keyword has the same function as ROLLBACK and triggers synchronous rollback.
   c. If a fatal/panic error is reported during transaction running, the system attempts to roll back the transaction bound to the thread before exiting the thread.

2. **Asynchronous rollback.** When the synchronous rollback fails or the system is restarted after breakdown, the undo recycling thread initiates an asynchronous rollback task for the transaction that is not rolled back completely and provides services for external systems immediately. The task initiation thread Undo Launch of asynchronous rollback starts the worker thread Undo Worker to execute the rollback task. The Undo Launch thread can start a maximum of five Undo Worker threads at the same time.
3. **In-page rollback.** If the rollback operation of a transaction page is not completed, but other transactions need to reuse the TD occupied by this transaction, the in-page rollback operation is performed for all modifications on the current page. In-page rollback only rolls back modifications on the current page. Other pages are not involved.

   The rollback of a Ustore sub-transaction is controlled by the ROLLBACK TO SAVEPOINT statement. After a sub-transaction is rolled back, the parent transaction can continue to run. The rollback of a sub-transaction does not affect the transaction status of the parent transaction. If sub-transactions that are not released during the parent transaction rollback, the sub-transactions are rolled back before the parent transaction is rolled back.

3.3.4 Flashback

Flashback is a part of the database recovery technology. It can be used to selectively cancel the impact of a committed transaction and restore data from incorrect manual operations. Before the flashback technology is used, the committed database modification can be retrieved only by means of restoring backup and PITR. The restoration takes several minutes or even hours. After the flashback technology is used, it takes only seconds to restore the DROP/TRUNCATE data committed in the database through FLASHBACK DROP and FLASHBACK TRUNCATE. In addition, the restoration time is irrelevant to the database size.

NOTE

- Astore does not support the flashback function.
- Standby database instances do not support the flashback function.
- You can enable the flashback function as required. Note that enabling this function will cause performance deterioration.

3.3.4.1 Flashback Query

Context

Flashback query enables you to query a snapshot of a table at a certain time point in the past. This feature can be used to view and logically rebuild damaged data that is accidentally deleted or modified. The flashback query is based on the MVCC mechanism. You can retrieve and query the earlier version to obtain the data of the specified earlier version.

Prerequisites

The **undo_retention_time** parameter has been set for specifying the retention period of undo logs.

Syntax

```
{[ ONLY ] table_name [ * ] [ partition_clause ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
[ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ] ]
[TIMECAPSULE { TIMESTAMP | CSN } expression ]
|( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
|with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
|function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] | column_definition [, ...] ) ]
|function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
|from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]}
```

In the syntax tree, **TIMECAPSULE {TIMESTAMP | CSN} expression** is a new expression for the flashback function. **TIMECAPSULE** indicates that the flashback function is used. **TIMESTAMP** and **CSN** indicate that the flashback function uses specific time point information or commit sequence number (CSN) information.

Parameter Description

- TIMESTAMP
  – Specifies a history time point of the table data to be queried.
- CSN
  – Specifies a logical commit time point of the data in the entire database to be queried. Each CSN in the database represents a consistency point of the entire database. To query the data under a CSN means to query the data related to the consistency point in the database through SQL statements.

CAUTION

When the time point is used for flashback, there may be a 3s error. To flash back to an operation point exactly, you need to use CSN for flashback.

Examples

- Example:

```
gaussdb=# drop TABLE IF EXISTS "public".flashtest;
NOTICE:  table "flashtest" does not exist, skipping
DROP TABLE
-- Create the flashtest table.
```

```
gaussdb=# CREATE TABLE "public".flashtest (col1 INT,col2 TEXT) with(storage_type=ustore);
CREATE TABLE
-- Query the CSN.
gaussdb=# select int8in(xidout(next_csn)) from gs_get_next_xid_csn();
  int8in
----------
 79351682
(1 rows)
-- Query the current timestamp.
gaussdb=# select now();
              now
-------------------------------
 2023-09-13 19:35:26.011986+08
(1 row)
-- Insert data.
gaussdb=# INSERT INTO flashtest VALUES(1,'INSERT1'),(2,'INSERT2'),(3,'INSERT3'),(4,'INSERT4'),
(5,'INSERT5'),(6,'INSERT6');
INSERT 0 6
gaussdb=# SELECT * FROM flashtest;
 col1 |  col2
------+---------
    3 | INSERT3
    1 | INSERT1
    2 | INSERT2
    4 | INSERT4
    5 | INSERT5
    6 | INSERT6
(6 rows)
-- Use flashback query to query the table at a CSN.
gaussdb=# SELECT * FROM flashtest TIMECAPSULE CSN 79351682;
 col1 | col2
------+------
(0 rows)
gaussdb=# SELECT * FROM flashtest;
 col1 |  col2
------+---------
    1 | INSERT1
    2 | INSERT2
    4 | INSERT4
    5 | INSERT5
    3 | INSERT3
    6 | INSERT6
(6 rows)
-- Use flashback query to query the table at a timestamp.
gaussdb=# SELECT * FROM flashtest TIMECAPSULE TIMESTAMP '2023-09-13 19:35:26.011986';
 col1 | col2
------+------
(0 rows)
gaussdb=# SELECT * FROM flashtest;
 col1 |  col2
------+---------
    1 | INSERT1
    2 | INSERT2
    4 | INSERT4
    5 | INSERT5
    3 | INSERT3
    6 | INSERT6
(6 rows)
-- Use flashback query to query the table at a timestamp.
gaussdb=# SELECT * FROM flashtest TIMECAPSULE TIMESTAMP to_timestamp ('2023-09-13
19:35:26.011986', 'YYYY-MM-DD HH24:MI:SS.FF');
 col1 | col2
------+------
(0 rows)
-- Use flashback query to query the table at a CSN and rename the table.
gaussdb=# SELECT * FROM flashtest AS ft TIMECAPSULE CSN 79351682;
 col1 | col2
------+------
(0 rows)
```

```
gaussdb=# drop TABLE IF EXISTS "public".flashtest;
DROP TABLE
```

3.3.4.2 Flashback Table

Context

Flashback table enables you to restore a table to a specific point in time. When only one table or a group of tables are logically damaged instead of the entire database, this feature can be used to quickly restore the table data. Based on the MVCC mechanism, the flashback table deletes incremental data at a specified time point and after the specified time point and retrieves the data deleted at the specified time point and the current time point to restore table-level data.

Prerequisites

The **undo_retention_time** parameter has been set for specifying the retention period of undo logs.

Syntax

```
TIMECAPSULE TABLE table_name TO { TIMESTAMP | CSN } expression
```

Examples

```
gaussdb=# drop TABLE IF EXISTS "public".flashtest;
NOTICE:  table "flashtest" does not exist, skipping
DROP TABLE
-- Create the flashtest table.
gaussdb=# CREATE TABLE "public".flashtest (col1 INT,col2 TEXT) with(storage_type=ustore);
CREATE TABLE
-- Query the CSN.
gaussdb=# select int8in(xidout(next_csn)) from gs_get_next_xid_csn();
  int8in
----------
 79352065
(1 rows)
-- Query the current timestamp.
gaussdb=# select now();
              now
-------------------------------
 2023-09-13 19:46:34.102863+08
(1 row)
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
------+------
(0 rows)
-- Insert data.
gaussdb=# INSERT INTO flashtest VALUES(1,'INSERT1'),(2,'INSERT2'),(3,'INSERT3'),(4,'INSERT4'),
(5,'INSERT5'),(6,'INSERT6');
INSERT 0 6
gaussdb=# SELECT * FROM flashtest;
 col1 |  col2
------+---------
    3 | INSERT3
    6 | INSERT6
    1 | INSERT1
    2 | INSERT2
    4 | INSERT4
    5 | INSERT5
(6 rows)
-- Flash a table back to a specific timestamp.
gaussdb=# TIMECAPSULE TABLE flashtest TO TIMESTAMP to_timestamp ('2023-09-13 19:52:21.551028',
'YYYY-MM-DD HH24:MI:SS.FF');
TimeCapsule Table
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
------+------
(0 rows)
gaussdb=# select now();
              now
-------------------------------
 2023-09-13 19:54:00.641506+08
(1 row)
-- Insert data.
gaussdb=# INSERT INTO flashtest VALUES(1,'INSERT1'),(2,'INSERT2'),(3,'INSERT3'),(4,'INSERT4'),
(5,'INSERT5'),(6,'INSERT6');
INSERT 0 6
gaussdb=# SELECT * FROM flashtest;
 col1 |  col2
------+---------
    3 | INSERT3
    6 | INSERT6
    1 | INSERT1
    2 | INSERT2
    4 | INSERT4
    5 | INSERT5
(6 rows)
-- Flash a table back to a specific timestamp.
gaussdb=# TIMECAPSULE TABLE flashtest TO TIMESTAMP '2023-09-13 19:54:00.641506';
TimeCapsule Table
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
------+------
(0 rows)
gaussdb=# drop TABLE IF EXISTS "public".flashtest;
DROP TABLE
```

3.3.4.3 DROP/TRUNCATE Flashback

Context

- Flashback DROP enables you to restore tables that are dropped by mistake and their auxiliary structures, such as indexes and table constraints, from the recycle bin. Flashback DROP is based on the recycle bin mechanism. You can restore physical table files recorded in the recycle bin to restore dropped tables.
- Flashback TRUNCATE enables you to restore tables that are truncated by mistake and restore the physical data of the truncated tables and indexes from the recycle bin. Flashback TRUNCATE is based on the recycle bin mechanism. You can restore physical table files recorded in the recycle bin to restore truncated tables.

Prerequisites

- The **enable_recyclebin** parameter has been enabled (by modifying the GUC parameter in the **postgresql.conf** file) to enable the recycle bin. Contact an administrator to modify the parameter.
- The **recyclebin_retention_time** parameter has been set for specifying the retention period of objects in the recycle bin. The objects will be automatically deleted after the retention period expires. Contact an administrator to modify the parameter.

Syntax

- Drop a table.
  ```
  DROP TABLE table_name [PURGE]
  ```
- Purge objects in the recycle bin.
  ```
  PURGE { TABLE { table_name }
  | INDEX { index_name }
  | RECYCLEBIN
  }
  ```
- Flash back a dropped table.
  ```
  TIMECAPSULE TABLE { table_name } TO BEFORE DROP [RENAME TO new_tablename]
  ```
- Truncate a table.
  ```
  TRUNCATE TABLE { table_name } [ PURGE ]
  ```
- Flash back a truncated table.
  ```
  TIMECAPSULE TABLE { table_name } TO BEFORE TRUNCATE
  ```

Parameters

- DROP/TRUNCATE TABLE table_name PURGE
  – Purges table data in the recycle bin by default.
- PURGE RECYCLEBIN
  – Purges objects in the recycle bin.
- **TO BEFORE DROP**

  Retrieves dropped tables and their subobjects from the recycle bin.

  You can specify either the original user-specified name of the table or the system-generated name assigned to the object when it was dropped.
  – System-generated recycle bin object names are unique. Therefore, if you specify the system-generated name, the database retrieves that specified object. To see the content in your recycle bin, run **select * from gs_recyclebin;**.
  – If you specify the user-specified name and the recycle bin contains more than one object of that name, the database retrieves the object that was moved to the recycle bin most recently. If you want to retrieve an older version of the table, then do one of these things:
    ▪ Specify the system-generated recycle bin name of the table you want to retrieve.
    ▪ Run the **TIMECAPSULE TABLE ... TO BEFORE DROP** statement until you retrieve the table you want.
  – When a dropped table is restored, only the base table name is restored, and the names of other subobjects remain the same as those in the recycle bin. You can run the DDL command to manually change the names of subobjects as required.
  – The recycle bin does not support write operations such as DML, DCL, and DDL, and does not support DQL query operations (supported in later versions).
  – Between the flashback point and the current point, a statement has been executed to modify the table structure or to affect the physical structure. Therefore, the flashback fails. The error message "ERROR: The table definition of %s has been changed." is displayed when flashback is

performed on a table where DDL operations have been performed. The error message "ERROR: recycle object %s desired does not exist" is displayed when flashback is performed on DDL operations, such as changing namespaces and table names.

- When the **enable_recyclebin** parameter is enabled, if a table has a TRUNCATE trigger, TRUNCATE TABLE cannot activate the trigger.

- **RENAME TO**

  Specifies a new name for the table retrieved from the recycle bin.

- **TO BEFORE TRUNCATE**

  Flashes back to the point in time before the TRUNCATE operation.

Syntax Example

```
-- PURGE TABLE table_name; --
-- Check the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------
(0 rows)

gaussdb=# drop table if EXISTS flashtest;
NOTICE:  table "flashtest" does not exist, skipping
DROP TABLE
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------
(0 rows)
-- Create the flashtest table.
gaussdb=# create table if not EXISTS flashtest(id int, name text) with (storage_type = ustore);
CREATE TABLE
-- Insert data.
gaussdb=# insert into flashtest values(1, 'A');
INSERT 0 1
gaussdb=# select * from flashtest;
 id | name
----+------
  1 | A
(1 row)

-- Drop the flashtest table.
gaussdb=# drop table if EXISTS flashtest;
DROP TABLE
-- Check the recycle bin. The dropped table is moved to the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid |           rcyname            |    rcyoriginname     | rcyoperation | rcytype |
rcyrecyclecsn |        rcyrecycletime         | rcycreatecsn | rcychangecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid |
rcyfrozenxid64
-----------+---------+----------+------------------------------+----------------------+--------------+---------
+---------------+------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------
     18591 |   12737 |    18585 | BIN$31C14EB4899$9737$0==$0   | flashtest            | d            |       0 |
79352606 | 2023-09-13 20:01:28.640664+08 |     79352595 |     7935259
5 |         2200 |       10 |             0 |          18585 | t             | t           | 225492       |         225492
     18591 |   12737 |    18590 | BIN$31C14EB489E$12D1B978==$0 | pg_toast_18585_index | d            |       3
```

```
|      79352606 | 2023-09-13 20:01:28.64093+08  |      79352595 |      7935259
5 |            99 |        10 |              0 |           18590 | f              | f            | 0            |              0
     18591 |   12737 |    18588 | BIN$31C14EB489C$12D1BF60==$0 | pg_toast_18585       | d            |        2
|      79352606 | 2023-09-13 20:01:28.641018+08 |             0 |
0 |            99 |        10 |              0 |           18588 | f              | f            | 225492       |          225492
(3 rows)
-- Check the flashtest table. The table does not exist.
gaussdb=# select * from flashtest;
ERROR:  relation "flashtest" does not exist
LINE 1: select * from flashtest;
                      ^
-- Purge the table from the recycle bin.
gaussdb=# PURGE TABLE flashtest;
PURGE TABLE
-- Check the recycle bin. The table is purged from the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------
(0 rows)


-- PURGE INDEX index_name; --
gaussdb=# drop table if EXISTS flashtest;
NOTICE:  table "flashtest" does not exist, skipping
DROP TABLE
-- Create the flashtest table.
gaussdb=# create table if not EXISTS flashtest(id int, name text) with (storage_type = ustore);
CREATE TABLE
-- Create the flashtest_index index for the flashtest table.
gaussdb=# create index flashtest_index on flashtest(id);
CREATE INDEX
-- Drop the table.
gaussdb=# drop table if EXISTS flashtest;
DROP TABLE
-- Check the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid |            rcyname             |    rcyoriginname      | rcyoperation | rcytype |
rcyrecyclecsn |         rcyrecycletime          | rcycreatecsn | rcychangecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid |
rcyfrozenxid64
-----------+---------+----------+------------------------------+----------------------+--------------+---------
+---------------+-------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------
     18648 |   12737 |    18641 | BIN$31C14EB48D1$9A85$0==$0   | flashtest            | d            |        0 |
79354509 | 2023-09-13 20:40:11.360638+08 |      79354506 |      7935450
8 |          2200 |        10 |              0 |           18641 | t              | t            | 226642       |          226642
     18648 |   12737 |    18646 | BIN$31C14EB48D6$12E230B8==$0 | pg_toast_18641_index | d            |        3
|      79354509 | 2023-09-13 20:40:11.361034+08 |      79354506 |      7935450
6 |            99 |        10 |              0 |           18646 | f              | f            | 0            |              0
     18648 |   12737 |    18644 | BIN$31C14EB48D4$12E236A0==$0 | pg_toast_18641       | d            |        2
|      79354509 | 2023-09-13 20:40:11.36112+08  |             0 |
0 |            99 |        10 |              0 |           18644 | f              | f            | 226642       |          226642
     18648 |   12737 |    18647 | BIN$31C14EB48D7$9A85$0==$0   | flashtest_index      | d            |        1 |
79354509 | 2023-09-13 20:40:11.361246+08 |      79354508 |      7935450
8 |          2200 |        10 |              0 |           18647 | f              | t            | 0            |              0
(4 rows)

--Purge the flashtest_index index.
gaussdb=# PURGE index flashtest_index;
PURGE INDEX
-- Check the recycle bin. The flashtest_index index is purged from the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid |            rcyname             |    rcyoriginname      | rcyoperation | rcytype |
rcyrecyclecsn |         rcyrecycletime          | rcycreatecsn | rcychangecs
```

```
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid |
rcyfrozenxid64
-----------+---------+----------+-----------------------------+----------------------+--------------+---------
+---------------+------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------
    18648 |   12737 |    18641 | BIN$31C14EB48D1$9A85$0==$0   | flashtest            | d            |       0 |
79354509 | 2023-09-13 20:40:11.360638+08 |     79354506 |     7935450
8 |         2200 |        10 |             0 |          18641 | t             | t           | 226642       |         226642
    18648 |   12737 |    18646 | BIN$31C14EB48D6$12E230B8==$0 | pg_toast_18641_index | d            |       3
|      79354509 | 2023-09-13 20:40:11.361034+08 |     79354506 |     7935450
6 |           99 |        10 |             0 |          18646 | f             | f           | 0            |             0
    18648 |   12737 |    18644 | BIN$31C14EB48D4$12E236A0==$0 | pg_toast_18641       | d            |       2
|      79354509 | 2023-09-13 20:40:11.36112+08  |            0 |
0 |           99 |        10 |             0 |          18644 | f             | f           | 226642       |         226642
(3 rows)

-- PURGE RECYCLEBIN --
-- Purge the recycle bin.
gaussdb=# PURGE RECYCLEBIN;
PURGE RECYCLEBIN
-- Check the recycle bin. The recycle bin is purged.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------
(0 rows)

-- TIMECAPSULE TABLE { table_name } TO BEFORE DROP [RENAME TO new_tablename] --
gaussdb=# drop table if EXISTS flashtest;
NOTICE:  table "flashtest" does not exist, skipping
DROP TABLE
-- Create the flashtest table.
gaussdb=# create table if not EXISTS flashtest(id int, name text) with (storage_type = ustore);
CREATE TABLE
-- Insert data.
gaussdb=# insert into flashtest values(1, 'A');
INSERT 0 1
gaussdb=# select * from flashtest;
 id | name
----+------
  1 | A
(1 row)

-- Drop the table.
gaussdb=# drop table if EXISTS flashtest;
DROP TABLE
-- Check the recycle bin. The table is moved to the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid |           rcyname            |    rcyoriginname     | rcyoperation | rcytype |
rcyrecyclecsn |        rcyrecycletime         | rcycreatecsn | rcychangecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid |
rcyfrozenxid64
-----------+---------+----------+-----------------------------+----------------------+--------------+---------
+---------------+------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------
    18658 |   12737 |    18652 | BIN$31C14EB48DC$9B2B$0==$0   | flashtest            | d            |       0 |
79354760 | 2023-09-13 20:47:57.075907+08 |     79354753 |     7935475
3 |         2200 |        10 |             0 |          18652 | t             | t           | 226824       |         226824
    18658 |   12737 |    18657 | BIN$31C14EB48E1$12E45E00==$0 | pg_toast_18652_index | d            |       3
|      79354760 | 2023-09-13 20:47:57.076129+08 |     79354753 |     7935475
3 |           99 |        10 |             0 |          18657 | f             | f           | 0            |             0
    18658 |   12737 |    18655 | BIN$31C14EB48DF$12E46400==$0 | pg_toast_18652       | d            |       2
|      79354760 | 2023-09-13 20:47:57.07621+08  |            0 |
0 |           99 |        10 |             0 |          18655 | f             | f           | 226824       |         226824
```

```
(3 rows)

-- Check the table. The table does not exist.
gaussdb=# select * from flashtest;
ERROR:  relation "flashtest" does not exist
LINE 1: select * from flashtest;
                      ^
-- Flash back a dropped table.
gaussdb=# timecapsule table flashtest to before drop;
TimeCapsule Table
-- Check the table. The table is restored to the state before the DROP operation.
gaussdb=# select * from flashtest;
 id | name
----+------
  1 | A
(1 row)

-- Check the recycle bin. The table is removed from the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------
(0 rows)

-- Drop the table.
gaussdb=# drop table if EXISTS flashtest;
DROP TABLE
gaussdb=# select * from flashtest;
ERROR:  relation "flashtest" does not exist
LINE 1: select * from flashtest;
                      ^
-- Check the recycle bin. The table is moved to the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid |           rcyname            |        rcyoriginname         | rcyoperation | rcytype |
rcyrecyclecsn |        rcyrecycletime         | rcycreatecsn | rcy
changecsn | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge |
rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+------------------------------+------------------------------+--------------+---------
+---------------+------------------------------+--------------+----
----------+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------
     18664 |   12737 |    18652 | BIN$31C14EB48DC$9B4E$0==$0   | flashtest                    | d            |       0
|      79354845 | 2023-09-13 20:49:17.762977+08 |     79354753 |
 79354753 |         2200 |       10 |             0 |          18652 | t             | t           | 226824       |         226824
     18664 |   12737 |    18657 | BIN$31C14EB48E1$12E680A8==$0 | BIN$31C14EB48E1$12E45E00==$0 |
d            |       3 |      79354845 | 2023-09-13 20:49:17.763271+08 |     79354753 |
 79354753 |           99 |       10 |             0 |          18657 | f             | f           | 0            |              0
     18664 |   12737 |    18655 | BIN$31C14EB48DF$12E68698==$0 | BIN$31C14EB48DF$12E46400==$0 |
d            |       2 |      79354845 | 2023-09-13 20:49:17.763343+08 |            0 |
        0 |           99 |       10 |             0 |          18655 | f             | f           | 226824       |         226824
(3 rows)

-- Flash back the dropped table. The table name is rcyname in the recycle bin.
gaussdb=# timecapsule table "BIN$31C14EB48DC$9B4E$0==$0" to before drop;
TimeCapsule Table
-- Check the recycle bin. The table is removed from the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------
(0 rows)

gaussdb=# select * from flashtest;
```

```
 id | name
----+------
  1 | A
(1 row)

-- Drop the table.
gaussdb=# drop table if EXISTS flashtest;
DROP TABLE
-- Check the recycle bin. The table is moved to the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid |           rcyname            |        rcyoriginname         | rcyoperation | rcytype |
rcyrecyclecsn |        rcyrecycletime         | rcycreatecsn | rcy
changecsn | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge |
rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+------------------------------+------------------------------+--------------+---------
+---------------+------------------------------+--------------+----
----------+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------
     18667 |   12737 |    18652 | BIN$31C14EB48DC$9B8D$0==$0   | flashtest                    | d            |       0
|      79354943 | 2023-09-13 20:52:14.525946+08 |     79354753 |
 79354753 |         2200 |       10 |             0 |          18652 | t             | t           | 226824       |         226824
     18667 |   12737 |    18657 | BIN$31C14EB48E1$1320B4F0==$0 | BIN$31C14EB48E1$12E680A8==$0 |
d            |       3 |      79354943 | 2023-09-13 20:52:14.526319+08 |     79354753 |
 79354753 |           99 |       10 |             0 |          18657 | f             | f           | 0            |             0
     18667 |   12737 |    18655 | BIN$31C14EB48DF$1320BAE0==$0 | BIN$31C14EB48DF$12E68698==$0 |
d            |       2 |      79354943 | 2023-09-13 20:52:14.526423+08 |            0 |
        0 |           99 |       10 |             0 |          18655 | f             | f           | 226824       |         226824
(3 rows)

-- Check the table. The table does not exist.
gaussdb=# select * from flashtest;
ERROR:  relation "flashtest" does not exist
LINE 1: select * from flashtest;
                      ^
-- Flash back the dropped table and rename the table.
gaussdb=# timecapsule table flashtest to before drop rename to flashtest_rename;
TimeCapsule Table
-- Check the original table. The table does not exist.
gaussdb=# select * from flashtest;
ERROR:  relation "flashtest" does not exist
LINE 1: select * from flashtest;
                      ^
-- Check the renamed table. The table exists.
gaussdb=# select * from flashtest_rename;
 id | name
----+------
  1 | A
(1 row)

-- Check the recycle bin. The table is removed from the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------
(0 rows)
-- Drop a table.
gaussdb=# drop table if EXISTS flashtest_rename;
DROP TABLE
-- Purge the recycle bin.
gaussdb=# PURGE RECYCLEBIN;
PURGE RECYCLEBIN
-- Check the recycle bin. The recycle bin is purged.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64
```

```
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+---------------+---------------+-------------+--------------+----------------
(0 rows)


-- TIMECAPSULE TABLE { table_name } TO BEFORE TRUNCATE --
gaussdb=# drop table if EXISTS flashtest;
NOTICE:  table "flashtest" does not exist, skipping
DROP TABLE
-- Create the flashtest table.
gaussdb=# create table if not EXISTS flashtest(id int, name text) with (storage_type = ustore);
CREATE TABLE
-- Insert data.
gaussdb=# insert into flashtest values(1, 'A');
INSERT 0 1
gaussdb=# select * from flashtest;
 id | name
----+------
  1 | A
(1 row)

-- Truncate a table.
gaussdb=# truncate table flashtest;
TRUNCATE TABLE
-- Check the recycle bin. The table data is moved to the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid |            rcyname             |    rcyoriginname     | rcyoperation | rcytype |
rcyrecyclecsn |         rcyrecycletime          | rcycreatecsn | rcychangecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid |
rcyfrozenxid64
-----------+---------+----------+-----------------------------+----------------------+--------------+---------
+---------------+------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------
     18703 |   12737 |    18697 | BIN$31C14EB4909$9E4C$0==$0   | flashtest            | t            |       0 |
79356608 | 2023-09-13 21:24:42.819863+08 |     79356606 |     7935660
6 |          2200 |        10 |              0 |           18697 | t              | t            | 227927       |         227927
     18703 |   12737 |    18700 | BIN$31C14EB490C$132FE3F0==$0 | pg_toast_18697       | t            |       2
|      79356608 | 2023-09-13 21:24:42.820358+08 |             0 |
0 |            99 |        10 |              0 |           18700 | f              | f            | 227927       |         227927
     18703 |   12737 |    18702 | BIN$31C14EB490E$132FEA40==$0 | pg_toast_18697_index | t            |       3
|      79356608 | 2023-09-13 21:24:42.821012+08 |     79356606 |     7935660
6 |            99 |        10 |              0 |           18702 | f              | f            | 0            |              0
(3 rows)

-- Check the table. The table is empty.
gaussdb=# select * from flashtest;
 id | name
----+------
(0 rows)

-- Flash back a truncated table.
gaussdb=# timecapsule table flashtest to before truncate;
TimeCapsule Table
-- Check the table. The data in the table is restored.
gaussdb=# select * from flashtest;
 id | name
----+------
  1 | A
(1 row)

-- Check the recycle bin.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid |            rcyname             |    rcyoriginname     | rcyoperation | rcytype |
rcyrecyclecsn |         rcyrecycletime          | rcycreatecsn | rcychangecs
n | rcynamespace | rcyowner | rcytablespace | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid |
rcyfrozenxid64
-----------+---------+----------+-----------------------------+----------------------+--------------+---------
```

```
+----------------+--------------------------------+---------------+-------------
--+---------------+-----------+----------------+-----------------+----------------+--------------+---------------
+-----------------
     18703 |    12737 |     18702 | BIN$31C14EB490E$132FFC38==$0 | pg_toast_18697_index | t             |        3
|       79356610 | 2023-09-13 21:24:42.872654+08 |       79356606 |       7935660
6 |             99 |         10 |               0 |            18708 | f               | f            | 0              |               0
     18703 |    12737 |     18700 | BIN$31C14EB490C$13300228==$0 | pg_toast_18697        | t             |        2
|       79356610 | 2023-09-13 21:24:42.872732+08 |               0 |
0 |             99 |         10 |               0 |            18706 | f               | f            | 0              |          227928
     18703 |    12737 |     18697 | BIN$31C14EB4909$9E4D$0==$0    | flashtest              | t             |        0 |
79356610 | 2023-09-13 21:24:42.872792+08 |       79356606 |       7935660
6 |           2200 |         10 |               0 |            18704 | t               | t            | 0              |          227928
(3 rows)

-- Drop a table.
gaussdb=# drop table if EXISTS flashtest;
DROP TABLE
-- Purge the recycle bin.
gaussdb=# PURGE RECYCLEBIN;
PURGE RECYCLEBIN
-- Check the recycle bin. The recycle bin is purged.
gaussdb=# select * from gs_recyclebin;
 rcybaseid | rcydbid | rcyrelid | rcyname | rcyoriginname | rcyoperation | rcytype | rcyrecyclecsn |
rcyrecycletime | rcycreatecsn | rcychangecsn | rcynamespace | rcyowner | rcytablespace
 | rcyrelfilenode | rcycanrestore | rcycanpurge | rcyfrozenxid | rcyfrozenxid64
-----------+---------+----------+---------+---------------+--------------+---------+---------------+-----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------
(0 rows)
```

3.3.5 Common View Tools

View Type	Type	Function	Application Scenario	Function
Parsing	All types	Parses a specified table page and returns the path for storing the parsed content.	• Page information viewing • Tuple (non-user data) information • Damaged pages and tuples • Tuple visibility problems • Verification errors	gs_parse_page_bypath

View Type	Type	Function	Application Scenario	Function
	Index recycle queue (URQ)	Parses key information in the URQ.	• UB-tree index space expansion • UB-tree index space recycle exceptions • Verification errors	gs_urq_dump_stat
	Rollback segment (undo)	Parses the specified undo record, excluding the old tuple data.	• Expanded undo space • Undo recycling exceptions • Rollback exceptions • Routine maintenance • Verification errors • Visibility judgment exceptions • Parameter modifications	gs_undo_dump_record
		Parses all undo records generated by a specified transaction, excluding old tuple data.		gs_undo_dump_xid
		Parses all information about transaction slots in a specified undo zone.		gs_undo_translot_dump_slot
		Parses the transaction slot information of a specified transaction, including the XID and the range of undo records generated by the transaction.		gs_undo_translot_dump_xid
		Parses the metadata of a specified undo zone and displays the pointer usage of undo records and transaction slots.		gs_undo_meta_dump_zone
		Parses the undo space metadata corresponding to a specified undo zone and displays the file usage of undo records.		gs_undo_meta_dump_spaces
		Parses the slot space metadata corresponding to a specified undo zone and displays the file usage of transaction slots.		gs_undo_meta_dump_slot

View Type	Type	Function	Application Scenario	Function
		Parses the data page and all data of historical versions and returns the path for storing the parsed content.		gs_undo_dump_parsepage_mv
	Write ahead log (WAL)	Parses Xlog within the specified LSN range and returns the path for storing parsed content. You can use **pg_current_xlog_location()** to obtain the current Xlog position.	• WAL errors • Log replay errors • Damaged pages	gs_xlogdump_lsn
		Parses Xlog of a specified XID and returns the path for storing parsed content. You can use **txid_current()** to obtain the current XID.		gs_xlogdump_xid
		Parses logs corresponding to a specified table page and returns the path for storing the parsed content.		gs_xlogdump_tablepath
		Parses the specified table page and logs corresponding to the table page and returns the path for storing the parsed content. It can be regarded as one execution of **gs_parse_page_bypath** and **gs_xlogdump_tablepath**. The prerequisite for executing this function is that the table file exists. To view logs of deleted tables, call **gs_xlogdump_tablepath**.		gs_xlogdump_parsepage_tablepath
Collecting	Rollback segment (undo)	Displays the statistics of the Undo module, including the usage of undo zones and undo links, creation and deletion of undo module files, and recommended values of undo module parameters.	• Undo space expansion • Undo resource monitoring	gs_stat_undo

View Type	Type	Function	Application Scenario	Function
	Write ahead log (WAL)	Collects statistics of the memory status table when WALs are written to disks.	• WAL write/disk flushing monitoring • Suspended WAL write/disk flushing	gs_stat_wal_entrytable
		Collects WAL statistics about the disk flushing status and location.		gs_walwriter_flush_position
		Collects WAL statistic about the frequency of disk flushing, data volume, and flushing files.		gs_walwriter_flush_stat
Validation	Heap table/Index	Checks whether the disk page data of tables or index files is normal offline.	• Damaged pages and tuples • Visibility issues • Log replay errors	ANALYZE VERIFY
		Checks whether physical files of the current database in the current instance are lost.	Lost files	gs_verify_data_file
	Index recycle (URQ)	Checks whether the data of the URQ (potential queue/available queue/single page) is normal.	• UB-tree index space expansion • UB-tree index space reclamation exceptions	gs_verify_urq

View Type	Type	Function	Application Scenario	Function
	Rollback segment (undo)	Checks whether undo records are normal offline.	• Abnormal or damaged undo records • Visibility issues • Abnormal or damaged rollback	gs_verify_undo_record
		Checks whether the transaction slot data is normal offline.	• Abnormal or damaged undo records • Visibility issues • Abnormal or damaged rollback	gs_verify_undo_slot
		Checks whether the undo metadata is normal offline.	• Node startup failure caused by undo metadata • Undo space reclamation exceptions • Outdated snapshots	gs_verify_undo_meta

View Type	Type	Function	Application Scenario	Function
Restoration	Heap table/ Index/ Undo file	Restores lost physical files on the primary node based on the standby node.	Lost heap tables/ Indexes/ undo files	gs_repair_file
	Heap table/ Index/ Undo page	Checks and restores damaged pages on the primary node based on the standby node.	Damaged heap tables/ indexes/ undo pages	gs_verify_and_tryrepair_page
		Restores the pages of the primary node based on the pages of the standby node.		gs_repair_page
		Modifies the bytes of the page backup based on the offset.		gs_edit_page_bypath
		Overwrites the modified page to the target page.		gs_repair_page_bypath
	Rollback segment (undo)	Rebuilds undo metadata. If the undo metadata is proper, rebuilding is not required.	Abnormal or damaged undo metadata	gs_repair_undo_byzone
	Index recycle queue (URQ)	Rebuilds the URQ.	Abnormal or damaged URQ	gs_repair_urq

3.3.6 Common Problems and Troubleshooting Methods

3.3.6.1 Snapshot Too Old

Undo space cannot save historical data if the execution time of the query SQL statement is too long or other reasons. Therefore, an error may be reported if the historical data is forcibly recycled. Generally, the rollback segment space needs to be expanded. However, the specific problem needs to be analyzed.

3.3.6.1.1 Undo Space Recycling Blocked by Long Transactions

Symptom

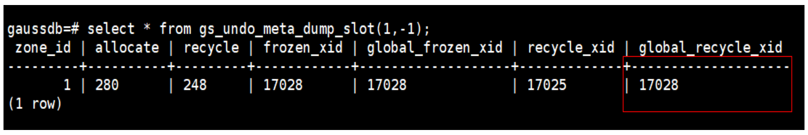
1. The following error information is printed in **pg_log**:

```
snapshot too old! the undo record has been forcibly discarded
```

```
xid xxx, the undo size xxx of the transaction exceeds the threshold xxx. trans_undo_threshold_size
xxx,undo_space_limit_size xxx.
```

In the actual error information, *xxx* indicates the actual data.

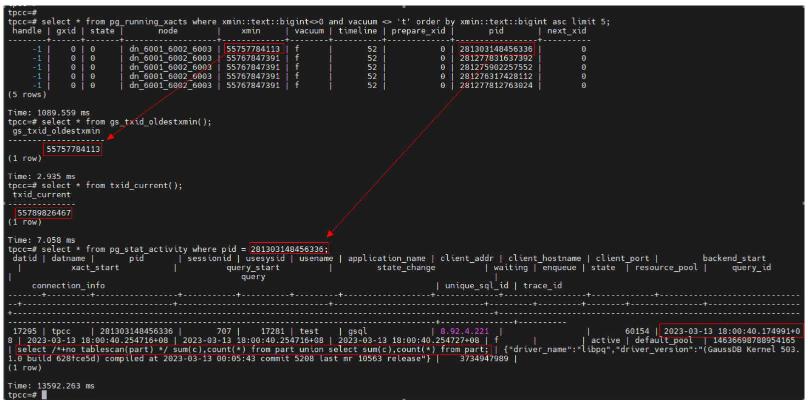
2. The value of **global_recycle_xid** (global recycling XID of the Undo subsystem) does not change for a long time.



```
gaussdb=# select * from gs_undo_meta_dump_slot(1,-1);
 zone_id | allocate | recycle | frozen_xid | global_frozen_xid | recycle_xid | global_recycle_xid
---------+----------+---------+------------+-------------------+-------------+-------------------
       1 | 280      | 248     | 17028      | 17028             | 17025       | 17028
(1 row)
```

3. Long transactions exist in the pg_running_xacts and pg_stat_activity views, blocking the progress of **oldestxmin** and **global_recycle_xid**. If the value of **xmin** for querying active transactions in pg_running_xacts is the same as that of gs_txid_oldestxmin and the execution time of the pg_stat_activity query thread based on a PID is too long, the recycling is suspended by a long transaction.

```
select * from pg_running_xacts where xmin::text::bigint<>0 and vacuum <> 't' order by
xmin::text::bigint asc      limit 5;
select * from gs_txid_oldestxmin();
select * from pg_stat_activity where pid = Thread PID where the long transaction exists
```



```
tpcc=#
tpcc=# select * from pg_running_xacts where xmin::text::bigint<>0 and vacuum <> 't' order by xmin::text::bigint asc limit 5;
 handle | gxid | state |        node        |     xmin     | vacuum | timeline | prepare_xid |       pid       | next_xid
--------+------+-------+--------------------+--------------+--------+----------+-------------+-----------------+----------
     -1 |    0 | 0     | dn_6001_6002_6003  | 55757784113  | f      |       52 |           0 | 281303148456336 |        0
     -1 |    0 | 0     | dn_6001_6002_6003  | 55767847391  | f      |       52 |           0 | 281277831637392 |        0
     -1 |    0 | 0     | dn_6001_6002_6003  | 55767847391  | f      |       52 |           0 | 281275902257552 |        0
     -1 |    0 | 0     | dn_6001_6002_6003  | 55767847391  | f      |       52 |           0 | 281276317428112 |        0
     -1 |    0 | 0     | dn_6001_6002_6003  | 55767847391  | f      |       52 |           0 | 281277812763024 |        0
(5 rows)

Time: 1089.559 ms
tpcc=# select * from gs_txid_oldestxmin();
 gs_txid_oldestxmin
--------------------
        55757784113
(1 row)

Time: 2.935 ms
tpcc=# select * from txid_current();
 txid_current
--------------
  55789826467
(1 row)

Time: 7.058 ms
tpcc=# select * from pg_stat_activity where pid = 281303148456336;
 datid | datname |       pid       | sessionid | usesysid | usename | application_name | client_addr | client_hostname | client_port |         backend_start
  |          xact_start           |          query_start          |         state_change          | waiting | enqueue | state  | resource_pool |     query_id
|                                                       query
     connection_info                                                                     | unique_sql_id | trace_id
-------+---------+-----------------+-----------+----------+---------+------------------+-------------+-----------------+-------------+-------------------------------
--+-------------------------------+-------------------------------+-------------------------------+---------+---------+--------+---------------+-------------------
+----------------------------------------------------------------------------------------------------+-----------------------------------------------------------
---------------------------------------------------------------------------------------+---------------+----------
 17295 | tpcc    | 281303148456336 |       707 |    17281 | test    | gsql             | 8.92.4.221  |                 |       60154 | 2023-03-13 18:00:40.174991+0
8 | 2023-03-13 18:00:40.254716+08 | 2023-03-13 18:00:40.254716+08 | 2023-03-13 18:00:40.254727+08 | f       |         | active | default_pool  | 14636698788954165
| select /*+no tablescan(part) */ sum(c),count(*) from part union select sum(c),count(*) from part; | {"driver_name":"libpq","driver_version":"(GaussDB Kernel 503.
1.0 build 628fce5d) compiled at 2023-03-13 00:05:43 commit 5208 last mr 10563 release"} |    3734947989 |
(1 row)

Time: 13592.263 ms
tpcc=#
```

Solution

Call pg_terminate_session(*pid*, *sessionid*) to terminate the sessions of the long transactions. (Note: There is no fixed quick restoration method for long transactions. Forcibly ending the execution of SQL statements is a common but high-risk operation. Exercise caution when performing this operation. Before performing this operation, please confirm with the administrator and Huawei technical personnel to prevent service failures or errors.)

3.3.6.1.2 Slow Undo Space Recycling Caused by Many Rollback Transactions

Symptom

The **gs_async_rollback_xact_status** view shows that there are a large number of transactions to be rolled back, and the number of transactions to be rolled back remains unchanged or keeps increasing.

```
select * from gs_async_rollback_xact_status();
```

Solution

Increase the number of asynchronous rollback threads in either of the following ways:

Method 1: Configure **max_undo_workers** in **postgresql.conf** and restart the node.

Method 2: Restart the instance using **gs_guc reload -Z NODE-TYPE [-N NODE-NAME] [-I INSTANCE-NAME | -D DATADIR] -c max_undo_workers=100**.

3.3.6.2 Storage Test Error

During service execution, if a data page, index, or undo page changes, logic damage detection is performed before the page is locked. If a page damage is detected, log information containing the keyword "storage test error" is exported to the database running log file **pg_log**. The page is restored to the status before the modification after rollback.

Symptom

The keyword "storage test error" is printed in **pg_log**.

Solution

Contact Huawei technical support.

3.3.6.3 An Error "UBTreeSearch::read_page has conflict with recovery, please try again later" Is Reported when a Service Uses a Standby Node to Read Data

Symptom

When the service uses the standby node to read data, an error (error code 43244) is reported. The error information contains "UBTreeSearch::read_page has conflict with recovery, please try again later."

Analysis

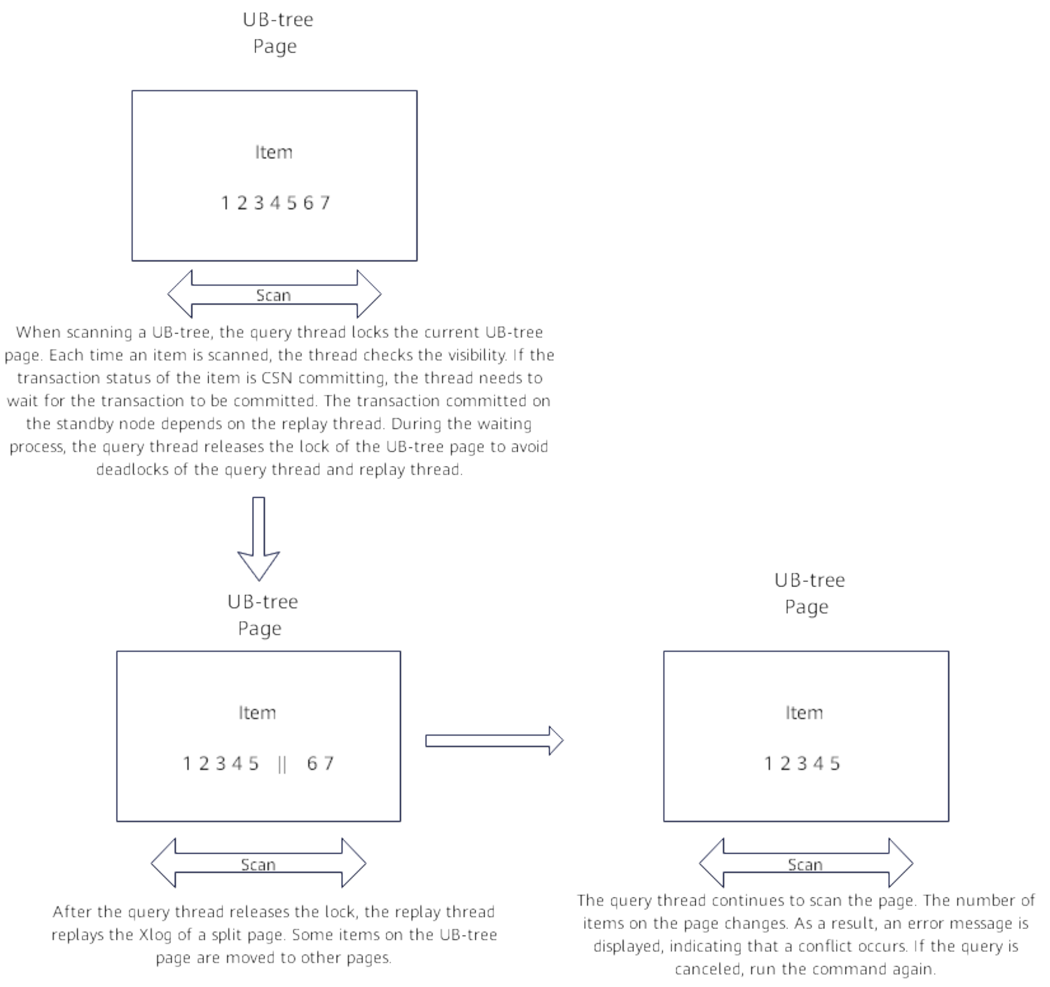
When parallel or serial replay is enabled (if the GUC parameters **recovery_parse_workers** and **recovery_max_workers** are both set to **1**, serial replay is enabled; if **recovery_parse_workers** is set to **1** and **recovery_max_workers** is greater than 1, parallel replay is enabled): If the query thread of the standby node scans indexes, a read lock is added to the index page. Each time a tuple is scanned, the visibility is checked. If the transaction corresponding to the tuple is in the committing state, the visibility is checked after the transaction is committed. Transaction committed on the standby node depends on the log replay thread. During this process, the index page is modified. Therefore, a lock is required. The query thread releases the lock of the index page during waiting. Otherwise, the query thread waits for the replay thread to commit the transaction, and the replay thread waits for the query thread to release the lock.

This error occurs only when the same index page needs to be accessed during query and replay. When the query thread releases the lock and waits for the transaction to end, the accessed page is modified.

NOTE

- When scanning tuples in the committing state, the standby node needs to wait for transactions to be committed because the transaction committing sequence and log generation sequence may be out of order. For example, the transaction **tx_1** on the primary node is committed earlier than transaction **tx_2**, the commit log of **tx_1** on the standby node is replayed after the commit log of **tx_2**. According to the transaction committing sequence, **tx_1** should be visible to **tx_2**. Therefore, you need to wait for the transaction to be committed.
- When the standby node scans the index page, it is found that the number of tuples (including dead tuples) on the page changes and cannot be retried. This is because the scanning may be forward or reverse scanning. For example, after the page is split, some tuples are moved to the right page. In the case of reverse scanning, even if the retry is performed, the tuples can only be read from the left, the correctness of the result cannot be ensured, and the split or insertion cannot be distinguished. Therefore, retry is not allowed.

Figure 3-1 Analysis


UB-tree Page
Item
1 2 3 4 5 6 7
Scan
When scanning a UB-tree, the query thread locks the current UB-tree page. Each time an item is scanned, the thread checks the visibility. If the transaction status of the item is CSN committing, the thread needs to wait for the transaction to be committed. The transaction committed on the standby node depends on the replay thread. During the waiting process, the query thread releases the lock of the UB-tree page to avoid deadlocks of the query thread and replay thread.
UB-tree Page
Item
1 2 3 4 5 || 6 7
Scan
After the query thread releases the lock, the replay thread replays the Xlog of a split page. Some items on the UB-tree page are moved to other pages.
UB-tree Page
Item
1 2 3 4 5
Scan
The query thread continues to scan the page. The number of items on the page changes. As a result, an error message is displayed, indicating that a conflict occurs. If the query is canceled, run the command again.

Solution

If an error is reported, you are advised to retry the query. In addition, you are advised to select index columns that are not frequently updated and use the soft deletion mode (physical deletion is performed during off-peak hours) to reduce the probability of this error.

4 Foreign Data Wrapper

Foreign data wrappers (FDWs) enable cross-database operations between GaussDB databases and remote servers (including databases and file systems). Currently, the following FDWs are supported: file_fdw.

4.1 file_fdw

The file_fdw module provides the foreign data wrapper file_fdw, which can be used to access data files in the file system of a server. The data file must be readable by COPY FROM. For details, see "SQL Reference > SQL Syntax > COPY" in *Developer Guide*. file_fdw is only used to access readable data files, but cannot write data to the data files.

By default, the file_fdw is compiled in GaussDB. During database initialization, the plug-in is created in the pg_catalog schema.

The server and foreign table corresponding to file_fdw can be created only by the initial user of the database or the O&M administrator who enables the O&M mode.

When you create a foreign table using file_fdw, you can add the following options:

- filename

  File to be read. This parameter is required and must be an absolute path.
- format

  File format of the remote server, which is the same as the **FORMAT** option in the COPY statement. The value can be **text**, **csv**, or **binary**.
- header

  Specifies whether a specified file has a header, which is the same as the **HEADER** option of the COPY statement.
- delimiter

  File delimiter, which is the same as the **DELIMITER** option of the COPY statement.
- quote

  Quote character of a file, which is the same as the **QUOTE** option of the COPY statement.

- escape

  Escape character of a file, which is the same as the **ESCAPE** option of the COPY statement.
- null

  Null string of a file, which is the same as the **NULL** option of the COPY statement.
- encoding

  Encoding of a file, which is the same as the **ENCODING** option of the COPY statement.
- force_not_null

  File-level null option, which is a Boolean option. If it is true, the value of the declared field cannot be an empty string. This option is the same as the **FORCE_NOT_NULL** option of the COPY statement.

NOTE

- file_fdw does not support the **OIDS** and **FORCE_QUOTE** options of the COPY statement.
- These options can only be declared for a foreign table or the columns of the foreign table, not for the file_fdw itself, nor for the server or user mapping that uses file_fdw.
- To modify table-level options, you must obtain the system administrator role permissions. For security reasons, only the system administrator can determine the files to be read.
- For a foreign table that uses file_fdw, running **EXPLAIN** displays the name and size (in bytes) of the file to be read. If the keyword **COSTS OFF** is specified, the file size is not displayed.

Using file_fdw

- To create a server object, run **CREATE SERVER**.
- To create a user mapping, run **CREATE USER MAPPING**.
- To drop a user mapping, run **DROP USER MAPPING**.
- To drop a server object, run **DROP SERVER**.

Examples

```
-- Create a server.
gaussdb=# CREATE SERVER file_server FOREIGN DATA WRAPPER file_fdw;
CREATE SERVER

-- Create a foreign table.
gaussdb=# CREATE FOREIGN TABLE file_ft(id int, name text) SERVER file_server OPTIONS(filename '/tmp/
1.csv', format 'csv', delimiter ',');
CREATE FOREIGN TABLE

-- Drop a foreign table.
gaussdb=# DROP FOREIGN TABLE file_ft;
DROP FOREIGN TABLE

-- Drop a server.
gaussdb=# DROP SERVER file_server;
DROP SERVER
```

Precautions

- To use file_fdw, you need to specify the file to be read. Prepare the file and grant the read permission on the file for the database to access the file.

- **DROP EXTENSION file_fdw** is not supported.
- The extended function is for internal use only. You are advised not to use it.

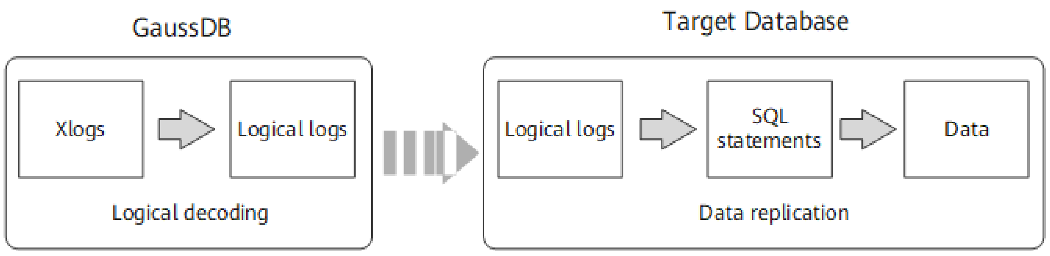
5 Logical Replication

The data replication capabilities supported by GaussDB are as follows:

Data is periodically synchronized to heterogeneous databases (such as Oracle Database) using a data migration tool. Real-time data replication is not supported. Therefore, real-time data synchronization to heterogeneous databases is not supported.

GaussDB provides the logical decoding function to generate logical logs by decoding Xlogs. A target database parses logical logs to replicate data in real time. For details, see **Figure 5-1**. Logical replication reduces the restrictions on target databases, allowing for data synchronization between heterogeneous databases and homogeneous databases with different forms. It allows data to be read and written during data synchronization on a target database, reducing the data synchronization latency.

Figure 5-1 Logical Replication



Logical replication consists of logical decoding and data replication. Logical decoding outputs logical logs by transaction. The database service or middleware parses the logical logs to implement data replication. Currently, GaussDB supports only logical decoding. Therefore, this section involves only logical decoding.

5.1 Logical Decoding

5.1.1 Overview

Description

Logical decoding provides basic transaction decoding capabilities for logical replication. GaussDB uses SQL functions for logical decoding. This method features easy function calling, requires no tools to obtain logical logs, and provides specific APIs for interconnecting with external replay tools, saving the need of additional adaptation.

Logical logs are output only after transactions are committed because they use transactions as the unit and logical decoding is driven by users. Therefore, to prevent Xlogs from being recycled by the system when transactions start and prevent required transaction information from being recycled by VACUUM, GaussDB introduces logical replication slots to block Xlog recycling.

A logical replication slot represents a stream of changes that can be re-executed in other databases in the order they were generated in the original database. Each logical replication slot is maintained by the person who obtains the corresponding logical logs. If the database where the logical replication slot in streaming decoding resides does not have services, the replication slot is updated based on the log location of other databases. The LSN-based logical replication slot in the active state may be updated based on the LSN of the current log when processing the active transaction snapshot log. The CSN-based logical replication slot in the active state may be updated based on the CSN of the current log when processing the virtual transaction log.

Prerequisites

- Currently, logical logs are extracted from DNs. If logical replication is performed, SSL connections must be used. Therefore, ensure that the **ssl** parameter on the corresponding DN is set to **on**.

  NOTE

  For security purposes, ensure that SSL connections are enabled.
- The GUC parameter **wal_level** is set to **logical**.
- The GUC parameter **max_replication_slots** is set to a value greater than or equal to the number of physical replication slots plus backup slots and logical replication slots required by each node.

  NOTE

  Plan the number of logical replication slots as follows:
  - A logical replication slot can carry changes of only one database for decoding. If multiple databases are involved, create multiple logical replication slots.
  - If logical replication is needed by multiple target databases, create multiple logical replication slots in the source database. Each logical replication slot corresponds to one logical replication link.
  - A maximum of 20 logical replication slots can be enabled for decoding on the same instance.
- Only the initial user and users with the REPLICATION permission can perform this operation. When separation of duties is disabled, database administrators can perform logical replication operations. When separation of duties is

enabled, database administrators are not allowed to perform logical replication operations.

Precautions

- Logical decoding does not support DDL statements.
- Decoded data may be lost when a specific DDL statement (for example, to truncate an ordinary table or exchange data between partitioned tables) is executed.
- Decoding of DML operations on data page replication is not supported.
- After a DDL statement (for example, ALTER TABLE) is executed, the physical logs that are not decoded before the DDL statement execution may be lost.
- The size of a single tuple cannot exceed 1 GB, and decoding results may be larger than inserted data. Therefore, it is recommended that the size of a single tuple be less than or equal to 500 MB.
- GaussDB supports the following types of data to be decoded: INTEGER, BIGINT, SMALLINT, TINYINT, SERIAL, SMALLSERIAL, BIGSERIAL, FLOAT, DOUBLE PRECISION, BOOLEAN, BIT(n), BIT VARYING(n), DATE, TIME[WITHOUT TIME ZONE], TIMESTAMP[WITHOUT TIME ZONE], CHAR(n), VARCHAR(n), TEXT, and CLOB (decoded into the text format).
- The name of a logical replication slot can contain no more than 64 characters, including lowercase letters, digits, underscores (_), question marks (?), hyphens (-), and periods (.). In addition, periods (. or ..) cannot be used as the name of a logical replication slot independently.
- After the database where a logical replication slot resides is deleted, the replication slot becomes unavailable and needs to be manually deleted.
- To decode multiple databases, you need to create a streaming replication slot in each database and start decoding. Logs need to be scanned for decoding of each database.
- Forcible switchover is not supported. After forcible switchover, you need to export all data again.
- During decoding on the standby node, the decoded data may increase due to switchover or failover, which needs to be manually filtered out. When the quorum protocol is used, switchover and failover should be performed on the standby node that is to be promoted to primary, and logs must be synchronized from the primary node to the standby node.
- The same replication slot for decoding cannot be used between the primary node and standby node or between different standby nodes at the same time. Otherwise, data inconsistency occurs.
- Replication slots can only be created or deleted on the primary node.
- After the database is restarted due to a fault or the logical replication process is restarted, duplicate decoded data may exist. You need to filter out the duplicate data.
- If the computer kernel is faulty, garbled characters may be displayed during decoding, which need to be manually or automatically filtered out.
- Ensure that the long transaction is not started during the creation of the logical replication slot. If the long transaction is started, the creation of the logical replication slot will be blocked. If the creation of a replication slot is blocked due to a long transaction, you can call the SQL function

pg_terminate_backend (*ID of the thread that creates the replication slot*) to manually stop the creation.

- Interval partitioned tables cannot be replicated.
- Decoding of DML operations on global temporary tables is not supported.
- After a DDL/DCL statement is executed in a transaction, the DDL/DCL statement and subsequent statements are not decoded.
- Do not perform operations on the replication slot on other nodes when the logical replication slot is in use. To delete a replication slot, stop decoding in the replication slot first.
- To parse the UPDATE and DELETE statements of an Astore table, you need to configure the REPLICA IDENTITY attribute for the table. If the table does not have a primary key, set the **REPLICA IDENTITY** attribute to **FULL**. For details about the configuration method, see column **REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }** in "SQL Reference > SQL Syntax > ALTER TABLE" in *Developer Guide*.
- Considering that the target database may require the system status information of the source database, logical decoding automatically filters only logical logs of system catalogs whose OIDs are less than 16384 in the pg_catalog and pg_toast schemas. If the target database does not need to copy the content of other related system catalogs, the related system catalogs need to be filtered during logical log replay.
- When logical replication is enabled, if you need to create a primary key index that contains system columns, you must set the **REPLICA IDENTITY** attribute of the table to **FULL** or use USING INDEX to specify a unique, non-local, non-deferrable index that does not contain system columns and contains only columns marked **NOT NULL**.
- If a transaction has too many sub-transactions, too many files are flushed to disks. To exit decoding, you need to call the SQL function pg_terminate_backend(*WAL sender thread ID for logical decoding*) to manually stop decoding. In addition, the exit delay increases by about 1 minute per 300,000 sub-transactions. Therefore, when logical decoding is enabled, if the number of sub-transactions of a transaction reaches 50,000, a WARNING log is generated.
- When a logical replication slot is inactive, GUC parameters **enable_xlog_prune** is set to **on**, **enable_logicalrepl_xlog_prune** is set to **on**, and **max_size_for_xlog_retention** is set to a non-zero value, the number of retained log segments caused by the backup slot or logical replication slot exceeds the value of **wal_keep_segments**, and other replication slots do not cause more retained log segments, if the value of **max_size_for_xlog_retention** is greater than 0 and the number of retained log segments (the size of each log segment is 16 MB) caused by the current logical replication slot exceeds the value of **max_size_for_xlog_retention**, or if the value of **max_size_for_xlog_retention** is less than 0 and the disk usage reaches the value of **–max_size_for_xlog_retention**/**100**, the logical replication slot is forcibly invalidated and **restart_lsn** is set to **FFFFFFFF/FFFFFFFF**. Logical replication slots in this state do not participate in the recycling of blocked logs or historical system catalogs, but the limitation on the maximum number of replication slots still takes effect. In this case, you need to manually delete them.

- After the standby node starts decoding and sends an instruction of updating the replication slot number to the primary node, the standby node occupies a corresponding logical replication slot (identified as an active state) on the primary node. Before that, the corresponding logical replication slot on the primary node is inactive. In this state, if the condition for forcibly invalidating the logical replication slot is met, the logical replication slot is marked as invalid (that is, **restart_lsn** is set to **FFFFFFFF/FFFFFFFF**). As a result, the standby node cannot update the replication slot on the primary node. In addition, after the standby node replays the logs indicating that the replication slot is invalid, the standby node of the current replication slot cannot reconnect to the primary node if decoding is interrupted.
- Inactive logical replication slots block WAL recycling and historical system catalog tuple clearing. As a result, disk logs are accumulated and system catalog scanning performance deteriorates. Therefore, you need to clear logical replication slots that are no longer used in time.
- Only LSN-based logical replication slots can be created by connecting to DNs using protocols.
- When the JSON format is used for decoding, the data column cannot contain special characters (such as the null character '\0'). Otherwise, the content in the decoding output column will be truncated.
- When a transaction generates a large number of sub-transactions that need to be flushed to disks, the number of opened file handles may exceed the upper limit. In this case, set **max_files_per_process** to a value greater than twice the upper limit of sub-transactions.
- sql_decoding decodes the UPDATE statement as a "DELETE+INSERT" operation.
- In the massive sub-transaction scenarios, the logical decoding memory control mechanism flushes logical logs to disks based on the threshold set by users. The number of flushed files is positively correlated with the number of sub-transactions. In this case, the disk read/write speed becomes the logical decoding bottleneck. Pay attention to the flushing latency in such scenarios. When data is stored remotely, the read latency overhead increases exponentially, and the write latency increases by multiple times. Therefore, massive sub-transactions need to be avoided at the service layer.

  You are advised to take the following measures:

  a. Check the sub-transaction service scenario and determine the number of sub-transactions.
  b. Modify parameters such as **max-changes-in-memory**, **max-reorderbuffer-in-memory**, and **max-txn-in-memory** to increase the threshold for generating temporary files.
  c. Reduce the number of sub-transactions based on service requirements. It is recommended that the number of sub-transactions be less than or equal to 1000.

SQL Function Decoding Performance

In the Benchmarksql-5.0 with 100 warehouses, when pg_logical_slot_get_changes is used:

- If 4000 lines of data (about 5 MB to 10 MB logs) are decoded at a time, the decoding performance ranges from 0.3 MB/s to 0.5 MB/s.

- If 32000 lines of data (about 40 MB to 80 MB logs) are decoded at a time, the decoding performance ranges from 3 MB/s to 5 MB/s.
- If 256000 lines of data (about 320 MB to 640 MB logs) are decoded at a time, the decoding performance ranges from 3 MB/s to 5 MB/s.
- If the amount of data to be decoded at a time still increases, the decoding performance is not significantly improved.

If pg_logical_slot_peek_changes and pg_replication_slot_advance are used, the decoding performance is 30% to 50% lower than that when pg_logical_slot_get_changes is used.

5.1.2 Logical Decoding Options

Logical decoding options can provide a restriction on or additional functions for the current logical decoding, for example, specifying whether the decoding result includes a transaction number or whether empty transactions are ignored during decoding. For details about the configuration method and SQL function decoding, see the optional input parameters **options_name** and **options_value** of the pg_logical_slot_peek_changes function in "SQL Reference > Functions and Operators > System Administration Functions > Logical Replication Functions" in *Developer Guide*. For details about JDBC streaming decoding, see the usage of the withSlotOption function in the sample code in "Application Development Guide > Development Based on JDBC > Example: Logical Replication Code Example" in *Developer Guide*.

General Options (Both serial decoding and parallel decoding can be configured, but the settings may be invalid. For details, see the description of related options.)

- **include-xids**:

  Specifies whether the decoded **data** column contains XID information.

  Value range: Boolean. The default value is **true**.

  - **false**: The decoded **data** column does not contain XID information.
  - **true**: The decoded **data** column contains XID information.

- **skip-empty-xacts**:

  Specifies whether to ignore empty transaction information during decoding.

  Value range: Boolean. The default value is **false**.

  - **false**: The empty transaction information is not ignored during decoding.
  - **true**: The empty transaction information is ignored during decoding.

- **include-timestamp**:

  Specifies whether decoded information contains the **commit** timestamp.

  Value range: Boolean. The default value is **false**.

  - **false**: The decoded information does not contain the **commit** timestamp.
  - **true**: The decoded information contains the **commit** timestamp.

- **only-local**:

  Specifies whether to decode only local logs.

  Value range: Boolean. The default value is **true**.

  - **false**: Non-local logs and local logs are decoded.
  - **true**: Only local logs are decoded.
- **force-binary**:

  Specifies whether to output the decoding result in binary format.

  Value range: **0**. The default value is **0**.

  - **0**: The decoding result is output in text format.
- **white-table-list**:

  Specifies the whitelist parameter, including the schema and table name to be decoded.

  Value range: a string that contains table names in the whitelist. Different tables are separated by commas (,). An asterisk (*) is used to fuzzily match all tables. Schema names and table names are separated by periods (.). No space character is allowed. For example:

  ```
  select * from pg_logical_slot_peek_changes('slot1', NULL, 4096, 'white-table-list',
  'public.t1,public.t2,*.t3,my_schema.*');
  ```
- **max-txn-in-memory**:

  Memory control parameter. The unit is MB. If the memory occupied by a single transaction is greater than the value of this parameter, data is flushed to disks.

  Value range: an integer ranging from 0 to 100. The default value is **0**, indicating that memory control is disabled.
- **max-reorderbuffer-in-memory:**

  Memory control parameter. The unit is GB. If the total memory (including the cache) of transactions being concatenated in the sender thread is greater than the value of this parameter, the current decoding transaction is flushed to disks.

  Value range: an integer ranging from 0 to 100. The default value is **0**, indicating that memory control is disabled.
- **include-user**:

  Specifies whether the BEGIN logical log of a transaction records the username of the transaction. The username of a transaction refers to the authorized user, that is, the login user who executes the session corresponding to the transaction. The username does not change during the execution of the transaction.

  Value range: Boolean. The default value is **false**.

  - **false**: The BEGIN logical log of a transaction does not record the username of the transaction.
  - **true**: The BEGIN logical log of a transaction records the username of the transaction.
- **exclude-userids**:

  Specifies the OID of a blacklisted user.

  Value range: a string, which specifies the OIDs of blacklisted users. Multiple OIDs are separated by commas (,). The system does not check whether the OIDs exist.
- **exclude-users**:

  Name list of blacklisted users.

Value range: a string, which specifies the names of blacklisted users. Multiple names are separated by commas (,). The system does not check whether the names exist.

- **dynamic-resolution**:

  Specifies whether to dynamically parse the names of blacklisted users.

  Value range: Boolean. The default value is **true**.

  – **false**: An error is reported and the logical decoding exits when the decoding detects that a user does not exist in the blacklist specified by **exclude-users**.

  – **true**: Decoding continues when it detects that a user does not exist in the blacklist specified by **exclude-users**.

- **standby-connection**:

  Specifies whether to restrict decoding only on the standby node. **This option is set only for streaming decoding.**

  Value range: Boolean. The default value is **false**.

  – **true**: Only the standby node can be connected for decoding. When the primary node is connected for decoding, an error is reported and the system exits.

  – **false**: The primary or standby node can be connected for decoding.

- **sender-timeout**:

  Heartbeat timeout threshold between the kernel and the client. **This option is set only for streaming decoding.** If no message is received from the client within the period, the logical decoding stops and disconnects from the client. The unit is ms.

  Value range: an integer ranging from 0 to 2147483647. The default value depends on the value of the GUC parameter **logical_sender_timeout**.

- change-log-max-len:

  Specifies the maximum length of the logical log buffer, in bytes. **This parameter is valid only for parallel decoding and is invalid for serial decoding and SQL function decoding.** If the length of a single decoding result exceeds the upper limit, the memory will be destroyed and another memory whose size is 1024 bytes is allocated for caching. If the value is too large, the memory usage increases. If the value is too small, the memory allocation and release operations are frequently triggered. Therefore, you are advised not to set it to a value less than 1024.

  Value range: 1 to 65535. The default value is **4096**.

- max-decode-to-sender-cache-num:

  Specifies the threshold of the number of cached parallel decoding logs. **This parameter is valid only for parallel decoding and is invalid for serial decoding and SQL function decoding.** If the number of cached logs does not exceed the threshold, the used decoding logs are stored in the cache. Otherwise, the cache is released directly.

  Value range: 1 to 65535. The default value is **4096**.

- **enable-heartbeat**:

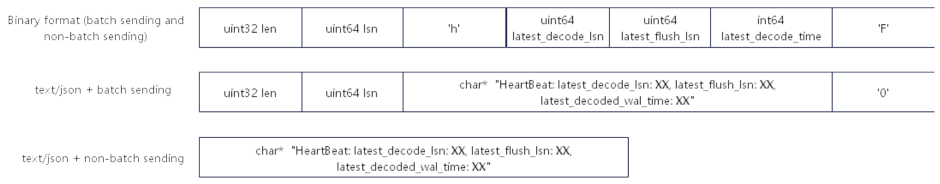
  Specifies whether to generate heartbeat logs. **This option is valid only for streaming decoding.**

  Value range: Boolean. The default value is **false**.

- **true**: Heartbeat logs are generated.
- **false**: Heartbeat logs are not generated.

NOTE

If the heartbeat log output option is enabled, heartbeat logs will be generated. The following uses parallel decoding as an example: The heartbeat logs can be in the binary format. For a binary heartbeat log message, it starts with a character 'h' and then the heartbeat log content: an 8-byte uint64 LSN indicating the end position of WAL reading when the heartbeat logical log is sent, an 8-byte uint64 LSN indicating the location of the WAL that has been flushed to disks when the heartbeat logical log is sent, and an 8-byte int64 timestamp (starting from January 1, 1970) indicating the timestamp when the latest decoded transaction log or checkpoint log is generated. Then, it ends with character 'F'. TEXT/JSON heartbeat log messages that are sent in batches end with '0'. There is no such terminator for each TEXT/JSON heartbeat log message. For details, see the following figure.


Binary format (batch sending and non-batch sending)
uint32 len
uint64 lsn
'h'
uint64 latest_decode_lsn
uint64 latest_flush_lsn
int64 latest_decode_time
'F'
text/json + batch sending
uint32 len
uint64 lsn
char* "HeartBeat: latest_decode_lsn: XX, latest_flush_lsn: XX, latest_decoded_wal_time: XX"
'0'
text/json + non-batch sending
char* "HeartBeat: latest_decode_lsn: XX, latest_flush_lsn: XX, latest_decoded_wal_time: XX"

- **parallel-decode-num**:

  Specifies the number of decoder threads for parallel decoding. **This option is valid only for streaming decoding.** When the system function is called, this option is invalid and only the value range is verified.

  Value range: an integer ranging from 1 to 20. The value **1** indicates that decoding is performed based on the original serial logic. Other values indicate that parallel decoding is enabled. The default value is **1**.

  NOTICE

  If **parallel-decode-num** is not set (the default value is **1**) or is explicitly set to **1**, the options in the following "Parallel Decoding" section cannot be configured.

- **output-order**:

  Specifies whether to use the CSN sequence to output decoding results. **This option is set only for streaming decoding.** When the system function is called, this option is invalid and only the value range is verified.

  Valid value: **0** or **1** of the int type. The default value is **0**.

  - **0**: The decoding results are sorted by transaction COMMIT LSN. This mode can be used only when the value of **confirmed_csn** of the decoding replication slot is set to **0** (not displayed). Otherwise, an error is reported.
  - **1**: The decoding results are sorted by transaction CSN. This mode can be used only when the value of **confirmed_csn** of the decoding replication slot is not set to **0**. Otherwise, an error is reported.

- **auto-advance**:

  Specifies whether to automatically update the logical replication slot number. **This parameter is valid only for streaming decoding.**

Value range: Boolean. The default value is **false**.

- **true**: The logical replication slot number is updated to the current decoding position when all sent logs are confirmed and there is no transaction to be sent.
- **false**: The replication service calls the log confirmation API to advance the logical replication slot.

- skip-generated-columns:

  Specifies whether to skip generated columns in the logical decoding result. This parameter is invalid for UPDATE and DELETE on old tuples, and the corresponding tuples always output the generated columns.

  Value range: Boolean. The default value is **false**.

  - **true**: The decoding result of generated columns is not output.
  - **false**: The decoding result of generated columns is output.

Serial Decoding

- **force-binary**:

  Specifies whether to output the decoding result in binary format and display different behaviors in different scenarios.

  - For system functions pg_logical_slot_get_binary_changes and pg_logical_slot_peek_binary_changes:

    Value range: Boolean. The default value is **false**. The value is meaningless. The decoding result is always output in binary format.

  - For system functions pg_logical_slot_get_changes, pg_logical_slot_peek_changes, and pg_logical_get_area_changes:

    Value range: Boolean. The value is fixed at **false**. The decoding result is always output in text format.

  - For streaming decoding:

    Value range: Boolean. The default value is **false**. The value is meaningless. The decoding result is always output in text format.

Parallel Decoding

The following configuration options are set only for streaming decoding.

- **decode-style**:

  Specifies the decoding format.

  Valid value: **'j'**, **'t'**, or **'b'** of the char type, indicating the JSON, TEXT, or binary format, respectively. The default value is **'b'**, indicating binary decoding.

  For the JSON and TEXT formats, in the decoding result sent in batches, the uint32 consisting of the first four bytes of each decoding statement indicates the total number of bytes of the statement (the four bytes occupied by the uint32 are excluded, and **0** indicates that the decoding of this batch ends). The 8-byte uint64 indicates the corresponding LSN (**begin** corresponds to **first_lsn**, **commit** corresponds to **end_lsn**, and other values correspond to the LSN of the statement).

NOTE

The binary encoding rules are as follows:

1. The first four bytes represent the total number of bytes of the decoding result of statements following the statement-level delimiter letter P (excluded) or the batch end character F (excluded). If the value is **0**, the decoding of this batch ends.
2. The next eight bytes (uint64) indicate the corresponding LSN (**begin** corresponds to **first_lsn**, **commit** corresponds to **end_lsn**, and other values correspond to the LSN of the statement).
3. The next one-byte letter can be **B**, **C**, **I**, **U**, or **D**, representing BEGIN, COMMIT, INSERT, UPDATE, or DELETE.
4. The letter in step **3** is **B**.
   1. The next eight bytes (uint64) indicate the CSN.
   2. The next eight bytes (uint64) indicate **first_lsn**.
   3. (Optional) If the next one-byte letter is **T**, the following four bytes (uint32) indicate the timestamp length for committing the transaction. The following characters with the same length are the timestamp character string.
   4. (Optional) If the next one-byte letter is **N**, the following four bytes (uint32) indicate the length of the transaction username. The following characters with the same length are the transaction username.
   5. Because there may still be a decoding statement subsequently, a one-byte letter **P** or **F** is used as a separator between statements. **P** indicates that there are still decoded statements in this batch, and **F** indicates that this batch is completed.
5. If **C** is used in step **3**:
   1. (Optional) If the next one-byte letter is **X**, the following eight bytes (uint64) indicate the XID.
   2. (Optional) If the next one-byte letter is **T**, the following four bytes (uint32) indicate the timestamp length. The following characters with the same length are the timestamp character string.
   3. When logs are sent in batches, decoding results of other transactions may still exist after a COMMIT log is decoded. If the next one-byte letter is **P**, the batch still needs to be decoded. If the letter is **F**, the batch decoding ends.
6. If **I**, **U**, or **D** is used in step **3**:
   1. The next two bytes (uint16) indicate the length of the schema name.
   2. The schema name is read based on the preceding length.
   3. The next two bytes (uint16) indicate the length of the table name.
   4. The table name is read based on the preceding length.
   5. (Optional) If the next one-byte letter is **N**, it indicates a new tuple. If the letter is **O**, it indicates an old tuple. In this case, the new tuple is sent first.
      1. The following two bytes (uint16) indicate the number of columns to be decoded for the tuple, which is recorded as **attrnum**.
      2. The following procedure is repeated for *attrnum* times.
         1. The next two bytes (uint16) indicate the length of the column name.
         2. The column name is read based on the preceding length.
         3. The next four bytes (uint32) indicate the OID of the current column type.
         4. The next four bytes (uint32) indicate the length of the value (stored in the character string format) in the current column. If the value is **0xFFFFFFFF**, it indicates null. If the value is **0**, it indicates a character string whose length is 0.
         5. The column value is read based on the preceding length.

6. Because there may still be a decoding statement after, if the next one-byte letter is **P**, it indicates that the batch still needs to be decoded, and if the next one-byte letter is **F**, it indicates that decoding of the batch ends.

- **sending-batch**:

  Specifies whether to send messages in batches.

  Valid value: **0** or **1** of the int type. The default value is **0**.

  – **0**: The decoding results are sent one by one.

  – **1**: When the accumulated size of decoding results reaches 1 MB, decoding results are sent in batches.

  In the scenario where batch sending is enabled, if the decoding format is 'j' or 't', before each original decoding statement, a uint32 number is added indicating the length of the decoding result (excluding the current uint32 number), and a uint64 number is added indicating the LSN corresponding to the current decoding result.

  NOTICE

  In the CSN-based decoding scenario (that is, **output-order** is set to **1**), batch sending is limited to a single transaction (that is, if a transaction has multiple small statements, the statements can be batch sent). That is, multiple transactions are not sent in the same batch, and BEGIN and COMMIT statements are not batch sent.

- **parallel-queue-size**:

  Specifies the length of the queue for interaction between parallel logical decoding threads.

  Value range: an integer ranging from 2 to 1024. The value must be an integer power of 2. The default value is **128**.

  The queue length is positively correlated with the memory usage during decoding.

- logical-reader-bind-cpu:

  Specifies the CPU core ID bound to the reader thread.

  Value range: –1 to 65535. If this parameter is not specified, cores are not bound.

  The default value is **–1**, indicating that cores are not bound. The value **–1** cannot be manually set. Ensure that the core ID is within the total number of logical cores of the machine. Otherwise, an error is reported. If multiple threads are bound to the same core, the load of the core increases and the performance deteriorates.

- **logical-decoder-bind-cpu-index**:

  Specifies the CPU core ID bound to the logical decoder thread.

  Value range: –1 to 65535. If this parameter is not specified, cores are not bound.

  The default value is **–1**, indicating that cores are not bound. The value **–1** cannot be manually set. Ensure that the core ID is within the total number of logical cores of the machine and is less than the value of [Number of CPU cores – Number of parallel logical decoders]. Otherwise, an error is reported.

Starting from the specified core ID, the number of newly started threads increases by one.

If multiple threads are bound to the same core, the load of the core increases and the performance deteriorates.

NOTE

When the GaussDB performs logical decoding and replays logs, a large number of CPU resources are occupied. Related threads such as WAL writer, WAL sender, WAL receiver and PageRedo are in the performance bottleneck. If these threads can be bound to a fixed CPU, frequent CPU switchovers caused by OS scheduling threads can be reduced. In this way, the performance overhead caused by cache miss is also reduced, improving the process handling speed. If the user scenario has performance requirements, you can optimize the configuration by referring to the following core binding example.

- The following is an example of setting parameters:
  1. walwriter_cpu_bind=1
  2. walwriteraux_bind_cpu=2
  3. wal_receiver_bind_cpu=4
  4. wal_rec_writer_bind_cpu=5
  5. wal_sender_bind_cpu_attr='cpuorderbind:7-14'
  6. redo_bind_cpu_attr='cpuorderbind:16-19'
  7. logical-reader-bind-cpu=20
  8. logical-decoder-bind-cpu-index=21
- In the example, cores 1, 2, 3, 4, 5, and 6 are bound using the GUC tool. The command is as follows:

  gs_guc set -Z datanode -N all -I all -c "walwriter_cpu_bind=1"

  In the example, cores 7 and 8 are bound when a decoding request is initiated by a JDBC client.
- In the example, **walwriter_cpu_bind=1** indicates that the thread can run on CPU core 1.

  cpuorderbind:7-14 indicates that each started thread is bound to CPU cores 7 to 14 in sequence. If the CPU cores in the range are used up, the newly started threads do not participate in core binding.

  logical-decoder-bind-cpu-index indicates that the started threads are bound to CPU cores 21, 22, 23 and so on.
- The core binding principle is that one thread occupies one CPU core.
- Inappropriate core binding, for example, binding multiple threads to one CPU core, may cause performance deterioration.
- You can run the **lscpu** command to view **CPU(s)**, that is, the number of logical CPU cores in your environment.

If the number of logical CPU cores is less than 36, you are advised not to use this core binding policy. In this case, you are advised to use the default configuration (no parameter setting).

5.1.3 Logical Decoding by SQL Function Interfaces

In GaussDB, you can call SQL functions to create, delete, and push logical replication slots, as well as obtain decoded transaction logs.

Procedure

Step 1 Log in to the primary node of the GaussDB database as a user who has the REPLICATION permission.

Step 2 Run the following command to connect to the database:

```
gsql -U user1 -d gaussdb -p 16000 -r
```

In the preceding command, **user1** indicates the username, **gaussdb** indicates the name of the database to be connected, and **16000** indicates the database port number. You can replace them as required.

Step 3 Create a logical replication slot named **slot1**.

```
gaussdb=# SELECT * FROM pg_create_logical_replication_slot('slot1', 'mppdb_decoding');
slotname | xlog_position
----------+---------------
slot1    | 0/601C150
(1 row)
```

Step 4 Create a table **t** in the database and insert data into it.

```
gaussdb=# CREATE TABLE t(a int PRIMARY KEY, b int);
gaussdb=# INSERT INTO t VALUES(3,3);
```

Step 5 Read the decoding result of **slot1**. The number of decoded records is 4096.

NOTE

For details about the logical decoding options, see **Logical Decoding Options**.

```
gaussdb=# SELECT * FROM pg_logical_slot_peek_changes('slot1', NULL, 4096);
location  |  xid  | data
-----------+-------
+-------------------------------------------------------------------------------------------------------------------------------
---------------------
-------------------------------------------
 0/601C188 | 1010023 | BEGIN 1010023
 0/601ED60 | 1010023 | COMMIT 1010023 CSN 1010022
 0/601ED60 | 1010024 | BEGIN 1010024
 0/601ED60 | 1010024 | {"table_name":"public.t","op_type":"INSERT","columns_name":
["a","b"],"columns_type":["integer","integer"],"columns_val":["3","3"],"old_keys_name":[],"old_keys_type":
[],"old_keys_val":[]}
 0/601EED8 | 1010024 | COMMIT 1010024 CSN 1010023
(5 rows)
```

Step 6 Drop the logical replication slot **slot1**.

```
gaussdb=#  SELECT * FROM pg_drop_replication_slot('slot1');
 pg_drop_replication_slot
--------------------------

(1 row)
```

----End

5.1.4 Logical Data Replication Using Streaming Decoding

A third-party replication tool extracts logical logs from GaussDB and replays them on the peer database. For details about the code of the replication tool that uses JDBC to connect to the database, see "Application Development Guide > Development Based on JDBC > Example: Logic Replication Code" in *Developer Guide*.