# 3.x Feature Guide for Distributed Instances - Huawei Cloud

|  |  |
|---|---|
| **Issue** | 01 |
| **Date** | 2024-05-17 |

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

## Huawei Cloud Computing Technologies Co., Ltd.

| | |
|---|---|
| Address: | Huawei Cloud Data Center Jiaoxinggong Road<br>Qianzhong Avenue<br>Gui'an New District<br>Gui Zhou 550029<br>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 a common view. A common view is a virtual table and has 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 actually stores the results of the statements executed by the SQL statement, and is used to cache the results.

## 1.1 Complete-Refresh Materialized View

### 1.1.1 Overview

Complete-refresh materialized views can be completely refreshed only. The syntax for creating a complete-refresh materialized view is the same as the CREATE TABLE AS syntax. You cannot specify a node group to create a complete-refresh materialized view.

### 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.
CREATE TABLE t1(c1 int, c2 int);
INSERT INTO t1 VALUES(1, 1);
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 again.
gaussdb=# INSERT INTO t1 VALUES(3, 3);

-- Completely refresh a 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)

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

### 1.1.3 Support and Constraints

#### Supported Scenarios

- Generally, the query scope supported by complete-refresh materialized views is the same as that supported by the CREATE TABLE AS statement.
- The distribution column can be specified when a complete-refresh materialized view is created.
- Indexes can be created in a complete-refresh materialized view.
- ANALYZE and EXPLAIN are supported.

#### Unsupported Scenarios

- Complete-refresh materialized views do not support node groups.
- Materialized views cannot be added, deleted, or modified. Only query statements are supported.

#### Constraints

- The base table used to create a complete-refresh materialized view must be defined on all DNs, and the node group to which the base table belongs must be an installation group.
- 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 incremental and the complete-refresh materialized views is that the fast-refresh materialized view supports 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.
CREATE TABLE t1(c1 int, c2 int);
INSERT INTO t1 VALUES(1, 1);
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 a 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 a 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 a 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.
- Creates an index in the materialized view.
- Performs the Analyze operation in the materialized view.
- Creates a fast-refresh materialized view based on the node group of base tables. (Check whether the base tables are in the same node group and create the fast-refreshmaterialized view based on the node group).

### Unsupported Scenarios

- Materialized views do not support the Stream plan, multi-table join plan, or subquery plan.
- Except for a few ALTER operations, most DDL operations cannot be performed on base tables in materialized views.
- A distribution key of a materialized view cannot be specified when the materialized view is created.
- Materialized views cannot be added, deleted, or modified. Only query statements are supported.
- Materialized views cannot be created using a temporary, hash bucket, unlogged, or partitioned table. Only the hash distribution table is supported.
- 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 is defined as UNION ALL, each subquery must use a different base table and the distribution key of each base table must be the same. The distribution key of the materialized view is automatically deduced and is the same as that of each base table.
- The columns defined in the materialized view must contain all distribution keys in the 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 Setting Encrypted Equality Queries

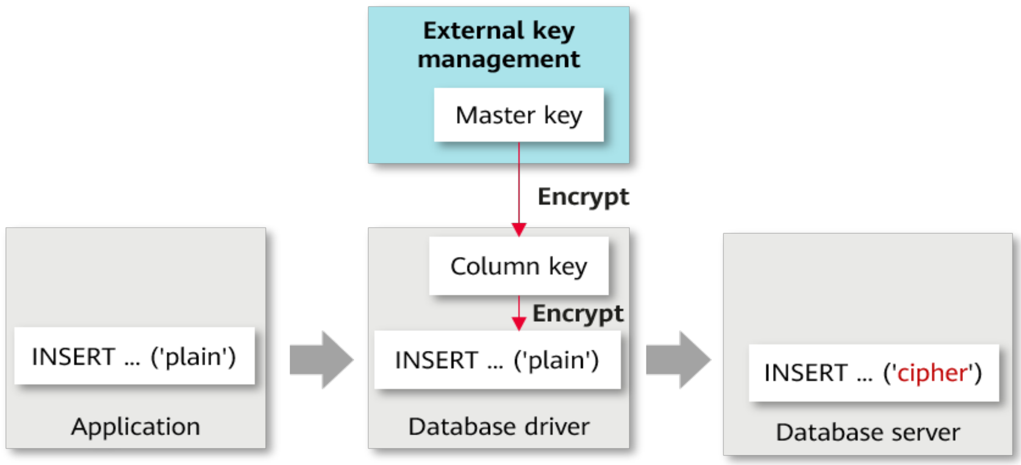
## 2.1 Overview

As enterprise data is migrated to the cloud, data security and privacy protection are facing increasingly severe challenges. The encrypted database will solve the privacy protection issues in the entire data lifecycle, covering network transmission, data storage, and data running status. Furthermore, the encrypted database can implement data privacy permission separation in a cloud scenario, that is, separate data owners from data administrators in terms of the read permission. The encrypted equality query is used to solve equality query issues of ciphertext data.

### Encryption Model

A fully-encrypted database uses a multi-level encryption model. The functions of keys in different encryption scenarios are as follows:

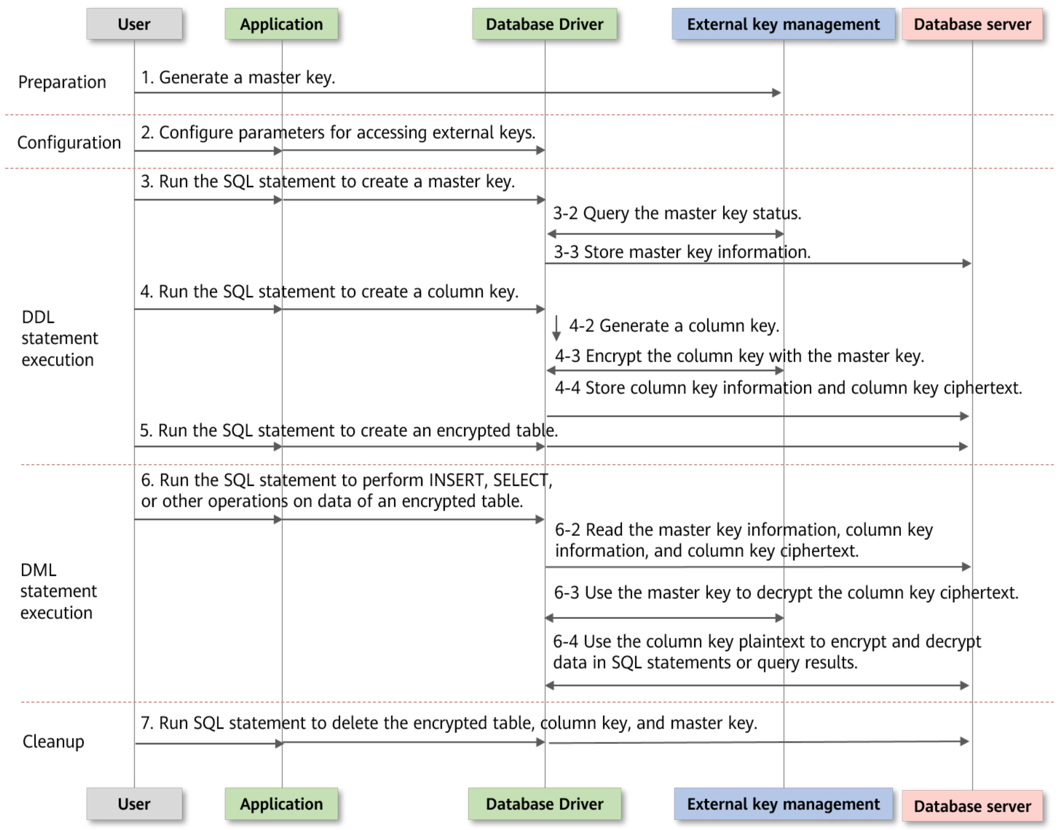
- Data: The encrypted database encrypts data of an encrypted column in SQL statements and decrypts the query result of the encrypted column returned by the database server.
- Column key: Data is encrypted by a column key, and the column key is encrypted by a master key. The column key ciphertext is stored on the database server.
- Master key: It is generated and stored in the external key management service. The database driver automatically accesses the external key management service to encrypt and decrypt column keys.



## Overall Process

The process of using a fully-encrypted database consists of the following five phases. This section describes the overall process. **Using gsql to Operate an Encrypted Database** and **Using JDBC to Operate an Encrypted Database** describe the detailed process.

1. Preparation phase: First, you need to generate a master key in the external key management service. External key management services include Huawei Cloud key management service. Select one of them as required.
2. Configuration phase: In an application, environment variables or database driver parameters are used to set information for accessing external key management service. In subsequent operations, the database driver needs to use the configuration information in this phase to access external key management service.
3. DDL statement execution phase: In this phase, you need to use the key syntax of the encrypted database to define a master key and a column key, define a table, and specify a column in the table as an encrypted column.
4. DML statement execution phase: After an encrypted table is created, you can directly execute syntax including but not limited to INSERT, SELECT, UPDATE, and DELETE. The database driver automatically encrypts and decrypts data of the encrypted column based on the encryption definition in the previous phase.
5. Cleanup phase: You can delete the encrypted table, column key, and master key in sequence.


User
Application
Database Driver
External key management
Database server
Preparation
1. Generate a master key.
Configuration
2. Configure parameters for accessing external keys.
DDL statement execution
3. Run the SQL statement to create a master key.
3-2 Query the master key status.
3-3 Store master key information.
4. Run the SQL statement to create a column key.
4-2 Generate a column key.
4-3 Encrypt the column key with the master key.
4-4 Store column key information and column key ciphertext.
5. Run the SQL statement to create an encrypted table.
DML statement execution
6. Run the SQL statement to perform INSERT, SELECT, or other operations on data of an encrypted table.
6-2 Read the master key information, column key information, and column key ciphertext.
6-3 Use the master key to decrypt the column key ciphertext.
6-4 Use the column key plaintext to encrypt and decrypt data in SQL statements or query results.
Cleanup
7. Run SQL statement to delete the encrypted table, column key, and master key.
User
Application
Database Driver
External key management
Database server

## Preparation Phase

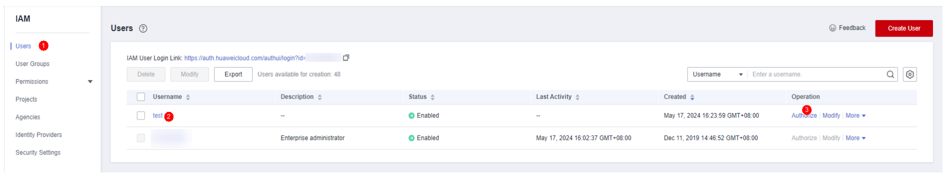
If you use the encrypted database for the first time, you need to perform the preparation. The next time you use the database, you can skip this phase.

The encrypted database can use different external keys to manage the master key. Select one of them as required.

- Huawei Cloud scenario

   a. Register an account with the Huawei Cloud official website and log in to the system.

   b. Search for **Identity and Access Management (IAM)** on the Huawei Cloud official website. On the page that is displayed, click **Create User**, set the IAM password for the IAM user, and grant the data encryption workshop (DEW) permission to the new IAM user.

   IAM User Login Link: https://auth.huaweicloud.com/authui/login?id=[redacted]

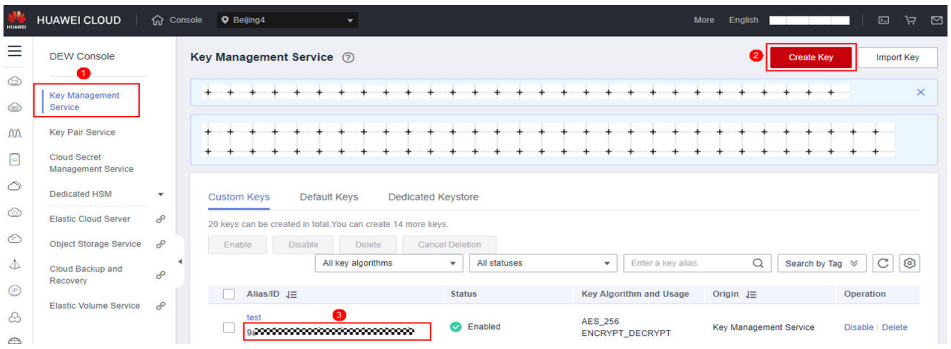
   Users available for creation: 48

   | Username | Description | Status | Last Activity | Created | Operation |
   |---|---|---|---|---|---|
   | test 2 | -- | Enabled | -- | May 17, 2024 16:23:59 GMT+08:00 | 3 Authorize \| Modify \| More |
   | [redacted] | Enterprise administrator | Enabled | May 17, 2024 16:02:37 GMT+08:00 | Dec 11, 2019 14:46:52 GMT+08:00 | Authorize \| Modify \| More |

   c. Go back to the login page, click **IAM User**, and log in to the system as the newly created IAM user. The subsequent operations are all performed by the IAM user.

   d. Search for **Data Encryption Workshop** on Huawei Cloud. On the page that is displayed, click **Key Management Service** and click **Create Key** to create a key. After the key is created, you can see that each key has a key

ID. Remember the key ID, which will be used when you create a master key in the DDL statement execution phase.



e. The key generated in this step is the master key used in the encrypted database. The key is stored in Huawei Cloud key management service. When SQL statements related to encryption and decryption are executed, the database driver automatically accesses the key through the RESTful API of Huawei Cloud.

## Configuration Phase

Configuring Parameters for Accessing External Keys

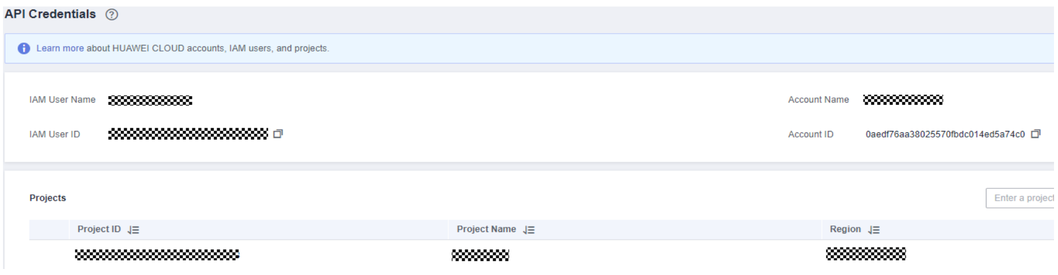
- Huawei Cloud scenario

  Configure the following information through environment variables.

  [terminal] # export HUAWEI_KMS_INFO='iamUrl=https://iam.{*Project*}.myhuaweicloud.com/v3/auth/tokens, iamUser={*IAM username*}, iamPassword={*IAM user key*}, iamDomain={*Account name*}, kmsProject={*Project*}'

  On the Huawei Cloud management console, click the username in the upper right corner and go to the **API Credentials** page. On this page, you can obtain the required parameters, including project, IAM username, and account name. Remember the project ID on this page, which will be used when you create a master key in the DDL statement execution phase.

  **Figure 2-1** Obtaining parameters on the Huawei Cloud page



```
# Example
[terminal] # export HUAWEI_KMS_INFO='iamUrl=https://iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens, iamUser=test_user, iamPassword=**********, iamDomain=test_account, kmsProject=cn-north-4'
```

## 2.2 Using gsql to Operate an Encrypted Database

### Executing SQL Statements

Before running the SQL statements in this section, ensure that the preparation and configuration phases are complete.

This section uses a complete execution process as an example to describe how to use the encrypted database syntax, including three phases: DDL statement execution, DML statement execution, and cleanup.

```
# 1. Connect to the database and use the -C parameter to enable the full encryption function.
[terminal] # gsql -p PORT gaussdb -h HOST -U USER -W PASSWORD -r -C

-- 2. Create a master key.
-- For details about the KEY_PATH format, see "SQL Reference > SQL Syntax > CREATE CLIENT MASTER KEY" in Developer Guide.
-- In the Huawei Cloud scenario, the project ID and key ID are required in KEY_PATH. For details about how to obtain the key ID, see the preparation phase. For details about how to obtain the project ID, see the configuration phase.
gaussdb=#  CREATE CLIENT MASTER KEY cmk1 WITH ( KEY_STORE = huawei_kms , KEY_PATH = 
'https://kms.cn-north-4.myhuaweicloud.com/v1.0/00000000000000000000000000000000/kms/
00000000-0000-0000-0000-00000000000', ALGORITHM = AES_256);
-- 3. Create a column key. The column key is encrypted by the master key created in the previous step. For details about the syntax, see "SQL Reference > SQL Syntax > CREATE COLUMN ENCRYPTION KEY " in Developer Guide.
gaussdb=# CREATE COLUMN ENCRYPTION KEY cek1 WITH VALUES (CLIENT_MASTER_KEY = cmk1, 
ALGORITHM  = AES_256_GCM);

-- 4. Create an encrypted table and use syntax to specify name and credit_card in the table as encrypted columns.
gaussdb=# CREATE TABLE creditcard_info (
  id_number int,
  name text encrypted with (column_encryption_key = cek1, encryption_type = DETERMINISTIC),
  credit_card varchar(19) encrypted with (column_encryption_key = cek1, encryption_type = 
DETERMINISTIC));
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'id_number' as the distribution key by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution key.
CREATE TABLE

-- 5. Write data to the encrypted table.
gaussdb=# INSERT INTO creditcard_info VALUES (1,'joe','6217986500001288393');
INSERT 0 1
gaussdb=# INSERT INTO creditcard_info VALUES (2, 'joy','6219985678349800033');
INSERT 0 1

-- 6. Query data from the encrypted table.
gaussdb=# select * from creditcard_info where name = 'joe';
 id_number | name |     credit_card
-----------+------+---------------------
         1 | joe  | 6217986500001288393

-- 7. Update data in the encrypted table.
gaussdb=# update creditcard_info set credit_card = '80000000011111111' where name = 'joy';
UPDATE 1

-- 8. Other operations: Add an encrypted column to a table.
gaussdb=# ALTER TABLE creditcard_info ADD COLUMN age int ENCRYPTED WITH 
(COLUMN_ENCRYPTION_KEY = cek1, ENCRYPTION_TYPE = DETERMINISTIC);
ALTER TABLE

-- 9. Other operations: Delete an encrypted column from a table.
gaussdb=# ALTER TABLE creditcard_info DROP COLUMN age;
ALTER TABLE
```

```
-- 10. Other operations: Query master key information from the system catalog.
gaussdb=# SELECT * FROM gs_client_global_keys;
 global_key_name | key_namespace | key_owner | key_acl |        create_date
-----------------+---------------+-----------+---------+----------------------------
 cmk1            |          2200 |        10 |         | 2021-04-21 11:04:00.656617
(1 rows)

-- 11. Other operations: Query column key information from the system catalog.
gaussdb=# SELECT column_key_name,column_key_distributed_id ,global_key_id,key_owner  FROM
gs_column_keys;
 column_key_name | column_key_distributed_id | global_key_id | key_owner
-----------------+---------------------------+---------------+-----------
 cek1            |                760411027 |         16392 |        10
(1 rows)

-- 12. Other operations: View the metadata of a column in a table.
gaussdb=# \d creditcard_info
        Table "public.creditcard_info"
   Column    |       Type        | Modifiers
-------------+-------------------+------------
 id_number   | integer           |
 name        | text              |  encrypted
 credit_card | character varying |  encrypted

-- 13. Delete an encrypted table.
gaussdb=# DROP TABLE creditcard_info;
DROP TABLE

-- 14. Delete a column key.
gaussdb=# DROP COLUMN ENCRYPTION KEY cek1;
DROP COLUMN ENCRYPTION KEY

-- 15. Delete a master key.
gaussdb=# DROP CLIENT MASTER KEY cmk1;
DROP CLIENT MASTER KEY
```

## 2.3 Using JDBC to Operate an Encrypted Database

### Obtain the JDBC driver package.

1. Obtain the JDBC driver package. For details about how to obtain and use the JDBC driver, see "Application Development Guide > Development Based on JDBC" in *Developer Guide*.

   The encrypted database supports the **gsjdbc4.jar**, **opengaussjdbc.jar**, and **gscejdbc.jar** driver packages.

   – **gsjdbc4.jar**: The main class name is **org.postgresql.Driver**, and the URL prefix of the database connection is **jdbc:postgresql**.

   – **opengaussjdbc.jar**: The main class name is **com.huawei.opengauss.jdbc.Driver**, and the URL prefix of the database connection is **jdbc:opengauss**.

   – **gscejdbc.jar** (currently, only EulerOS is supported): The main class name is **com.huawei.gaussdb.jdbc.Driver**, and the URL prefix of the database connection is **jdbc:gaussdb**. This driver package is recommended in encrypted scenarios.

2. Configure *LD_LIBRARY_PATH*.

   Before using the JDBC driver package in encrypted scenarios, you need to set the environment variable *LD_LIBRARY_PATH*.

- When the **gscejdbc.jar** driver package is used, the dependent library required by the encrypted database in the **gscejdbc.jar** driver package is automatically copied to the path and loaded when the encrypted database function is enabled to connect to the database.
- When using **opengaussjdbc.jar** or **gsjdbc4.jar**, you need to decompress **GaussDB-Kernel_***Database version number_OS version number***_64bit_libpq.tar.gz** to a specified directory, and add the path of the **lib** folder to the *LD_LIBRARY_PATH* environment variable.

**CAUTION**

To use the JDBC driver package in the full-encryption scenario, you must have the System.loadLibrary permission as well as the read and write permissions on files in the first-priority path of the environment variable *LD_LIBRARY_PATH*. You are advised to use an independent directory to store the full-encryption dependent library. If **java.library.path** is specified during execution, the value must be the same as the first-priority path of *LD_LIBRARY_PATH*.

When **gscejdbc.jar** is used, JVM that loads class files depends on the libstdc++ library of the system. If the encrypted database function is enabled, **gscejdbc.jar** automatically copies the dynamic libraries (including the libstdc++ library) on which the encrypted database depends to the *LD_LIBRARY_PATH* path set by the user. If the version of a dependent library does not match that of the existing system library, only the dependent library is deployed during the first running. After the dependent library is called again, it can be used normally.

## Executing SQL Statements

**Before running the SQL statements in this section, ensure that the preparation and configuration phases are complete.**

This section uses a complete execution process as an example to describe how to use the encrypted database syntax, including three phases: DDL statement execution, DML statement execution, and cleanup.

For details about JDBC development operations that are the same as those in non-encrypted scenarios, see "Application Development Guide > Development Based on JDBC" in *Developer Guide*.

- Connection parameters of an encrypted database

  **enable_ce**: string type. If **enable_ce** is set to **0**, the full encryption function is disabled. If **enable_ce** is set to **1**, the basic capability of encrypted equality query is supported.

```
// The following uses the gscejdbc.jar driver as an example. If other driver packages are used, you only need to change the driver class name and the URL prefix of the database connection.
// gsjdbc4.jar: The main class name is org.postgresql.Driver, and the URL prefix of the database connection is jdbc:postgresql.
// opengaussjdbc.jar: The main class name is com.huawei.opengauss.jdbc.Driver, and the URL prefix of the database connection is jdbc:opengauss.
// gscejdbc.jar: The main class name is com.huawei.gaussdb.jdbc.Driver, and the URL prefix of the database connection is jdbc:gaussdb.

public static void main(String[] args) {
    // Driver class.
    String driver = "com.huawei.gaussdb.jdbc.Driver";
    // Database connection descriptor. If enable_ce is set to 1, the encrypted equality query basic
capability is supported.
    String sourceURL = "jdbc:gaussdb://127.0.0.1:8000/postgres?enable_ce=1";
    // Set the username and password in the environment variables USER and PASSWORD, respectively.
    String username = System.getenv("USER");
    String passwd = System.getenv("PASSWORD");
    Connection conn = null;
    try {
        // Load the driver.
        Class.forName(driver);
        // Create a connection.
        conn = DriverManager.getConnection(sourceURL, username, passwd);
        System.out.println("Connection succeed!");
        // Create a statement object.
        Statement stmt = conn.createStatement();

        // Create a CMK.
        // For details about the KEY_PATH format, see "SQL Reference > SQL Syntax > CREATE CLIENT
MASTER KEY" in Developer Guide.
        // In the Huawei Cloud scenario, the project ID and key ID are required in KEY_PATH. For details
about how to obtain the key ID, see the preparation phase. For details about how to obtain the
project ID, see the configuration phase.
        int rc = stmt.executeUpdate("CREATE CLIENT MASTER KEY ImgCMK1 WITH ( KEY_STORE =
huawei_kms , KEY_PATH = 'https://kms.cn-north-4.myhuaweicloud.com/
v1.0/00000000000000000000000000000000/kms/00000000-0000-0000-0000-00000000000',
ALGORITHM = AES_256);");

        // Create a CEK.
        int rc2 = stmt.executeUpdate("CREATE COLUMN ENCRYPTION KEY ImgCEK1 WITH VALUES
(CLIENT_MASTER_KEY = ImgCMK1, ALGORITHM  = AES_256_GCM);");
        // Create an encrypted table.
        int rc3 = stmt.executeUpdate("CREATE TABLE creditcard_info (id_number int, name varchar(50)
encrypted with (column_encryption_key = ImgCEK1, encryption_type = DETERMINISTIC),credit_card
varchar(19) encrypted with (column_encryption_key = ImgCEK1, encryption_type =
DETERMINISTIC));");
        // Insert data.
        int rc4 = stmt.executeUpdate("INSERT INTO creditcard_info VALUES
(1,'joe','6217986500001288393');");
        // Query the encrypted table.
        ResultSet rs = null;
        rs = stmt.executeQuery("select * from creditcard_info where name = 'joe';");
        // Delete the encrypted table.
        int rc5 = stmt.executeUpdate("DROP TABLE IF EXISTS creditcard_info;");
        // Delete a CEK.
        int rc6 = stmt.executeUpdate("DROP COLUMN ENCRYPTION KEY IF EXISTS ImgCEK1;");
        // Delete the CMK.
        int rc7 = stmt.executeUpdate("DROP CLIENT MASTER KEY IF EXISTS ImgCMK1;");
        // Close the statement object.
        stmt.close();
        // Close the connection.
        conn.close();
    } catch (Exception e) {
        e.printStackTrace();
        return;
    }
}
```

NOTE

[Proposal] When JDBC is used to perform operations on an encrypted database, one database connection object corresponds to one thread. Otherwise, conflicts may occur due to thread changes.

[Proposal] When JDBC is used to perform operations on an encrypted database, different connections change the encrypted configuration data. The client calls the isvalid method to ensure that the connections can hold the changed encrypted configuration data. In this case, the **refreshClientEncryption** parameter must be set to **1** (default value). In a scenario where a single client performs operations on encrypted data, the **refreshClientEncryption** parameter can be set to **0**.

## Example of Calling the IsValid Method to Refresh the Cache

```
// Create a connection conn1.
Connection conn1 = DriverManager.getConnection("url","user","password");
// Create a CMK in another connection conn2.
...
// conn1 calls the IsValid method to refresh the conn1 key cache.
try {
    if (!conn1.isValid(60)) {
        System.out.println("isValid Failed for connection 1");
    }
} catch (SQLException e) {
    e.printStackTrace();
        return null;
}
```

## Decrypting the Encrypted Equality Ciphertext

A decryption API is added to the PgConnection class to decrypt the encrypted equality ciphertext of the fully-encrypted database. After decryption, the plaintext value is returned. The ciphertext column corresponding to the decryption is found based on **schema.table.column** and the original data type is returned.

**Table 2-1** org.postgresql.jdbc.PgConnection function

| Method | Return Type | Support JDBC 4 |
|---|---|---|
| decryptData(String ciphertext, Integer len, String schema, String table, String column) | ClientLogicDecryptRe-sult | Yes |

Parameters:

- **ciphertext**

  Ciphertext to be decrypted.
- **len**

  Ciphertext length. If the value is less than the actual ciphertext length, decryption fails.
- **schema**

  Name of the schema to which the encrypted column belongs.
- **table**

Name of the table to which the encrypted column belongs.

- **column**

  Name of the column to which the encrypted column belongs.

> **NOTE**
>
> Decryption is successful in the following scenarios, but is not recommended:
>
> - The input ciphertext length is longer than the actual ciphertext.
> - The **schema.table.column** points to other encrypted columns. In this case, the original data type of the encrypted column is returned.

**Table 2-2** org.postgresql.jdbc.clientlogic.ClientLogicDecryptResult function

| Method | Return Type | Description | Support JDBC4 (Yes/No) |
|---|---|---|---|
| isFailed() | Boolean | Specifies whether the decryption fails. If the decryption fails, **True** is returned. Otherwise, **False** is returned. | Yes |
| getErrMsg() | String | Obtains error information. | Yes |
| getPlaintext() | String | Obtains the decrypted plaintext. | Yes |
| getPlaintextSize() | Integer | Obtains the length of the decrypted plaintext. | Yes |
| getOriginalType() | String | Obtains the original data type of the encrypted column. | Yes |

```
// After the ciphertext is obtained through non-encrypted connection or logical decoding, this API can be used to decrypt the ciphertext.
import org.postgresql.jdbc.PgConnection;
import org.postgresql.jdbc.clientlogic.ClientLogicDecryptResult;

// conn is an encrypted connection.
// Call the decryptData method of PgConnection to decrypt the ciphertext, locate the encrypted column to which the ciphertext belongs based on the column name, and return the original data type.
ClientLogicDecryptResult decrypt_res = null;
decrypt_res = ((PgConnection)conn).decryptData(ciphertext, ciphertext.length(), schemaname_str,
        tablename_str, colname_str);
// Check whether the decryption of the returned result class is successful. If the decryption fails, obtain the error information. If the decryption is successful, obtain the plaintext, length, and original data type.
if (decrypt_res.isFailed()) {
    System.out.println(String.format("%s\n", decrypt_res.getErrMsg()));
} else {
    System.out.println(String.format("decrypted plaintext: %s size: %d type: %s\n", decrypt_res.getPlaintext(),
        decrypt_res.getPlaintextSize(), decrypt_res.getOriginalType()));
}
```

## Precompiling an Encrypted Table

```
// Create a prepared statement object by calling the prepareStatement method in Connection.
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO creditcard_info VALUES (?, ?, ?);");
```

```
// Set parameters by triggering the setShort method in PreparedStatement.
pstmt.setInt(1, 2);
pstmt.setString(2, "joy");
pstmt.setString(3, "6219985678349800033");
// Execute the precompiled SQL statement by triggering the executeUpdate method in
PreparedStatement.
int rowcount = pstmt.executeUpdate();
// Close the precompiled statement object by calling the close method in PreparedStatement.
pstmt.close();
```

### Batch Processing on an Encrypted Table

```
// Create a prepared statement object by calling the prepareStatement method in Connection.
Connection conn = DriverManager.getConnection("url","user","password");
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO creditcard_info (id_number, name,
credit_card) VALUES (?,?,?)");
// Call the setShort method for each piece of data, and call addBatch to confirm that the setting is
complete.
int loopCount = 20;
 for (int i = 1; i < loopCount + 1; ++i) {
      pstmt.setInt(1, i);
      pstmt.setString(2, "Name " + i);
      pstmt.setString(3, "CreditCard " + i);
      // Add row to the batch.
      pstmt.addBatch();
}
// Execute batch processing by calling the executeBatch method in PreparedStatement.
int[] rowcount = pstmt.executeBatch();
// Close the precompiled statement object by calling the close method in PreparedStatement.
pstmt.close();
```

## 2.4 Enhancing Security in the Configuration Phase

### Setting Environment Variables Securely

Sensitive information exists in *HUAWEI_KMS_INFO*. You are advised to set the environment variables as follows:

1. Set temporary environment variables: When an encrypted database is used, run the **export** command to set environment variables. After the database is used, run the **unset** command to clear environment variables. In this method, OS logs may record sensitive information. You are advised to use process-level environment variables or JDBC APIs to set connection parameters.
2. Set process-level environment variables: In the application code, set environment variables through programming interfaces. The following are examples of setting environment variables in different programming languages:
    a. C/C++: setenv(name, value)
    b. Go: os.Setenv(name, value)
    c. Java does not support the setting of process-level environment variables. Connection parameters can be set only through the JDBC APIs.

### Verifying External Key Management Service Identity

When the database driver accesses Huawei Cloud KMS, to prevent attackers from masquerading as the KMS, the CA certificate can be used to verify the validity of the key server during the establishment of HTTPS connections between the database driver and the KMS. Therefore, you need to configure the CA certificate in advance. If the CA certificate is not configured, the key management service identity will not be verified. The configuration method is as follows:

In the Huawei Cloud scenario, add the following parameters to the environment variables:

```
export HUAWEI_KMS_INFO='Other parameters, iamCaCert=Path/IAM CA certificate file,
kmsCaCert=Path/KMS CA certificate file'
```

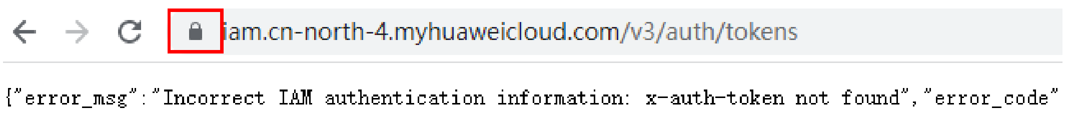
Most browsers automatically download a CA certificate of a website and provide the certificate export function. Some websites (such as **https://www.ssleye.com/ssltool/certs_down.html**) provide the function of automatically downloading CA certificates. However, the CA certificates may be unavailable due to proxy or gateway in the local environment. Therefore, you are advised to use a browser to download the CA certificate. You can perform the following steps:

**CAUTION**

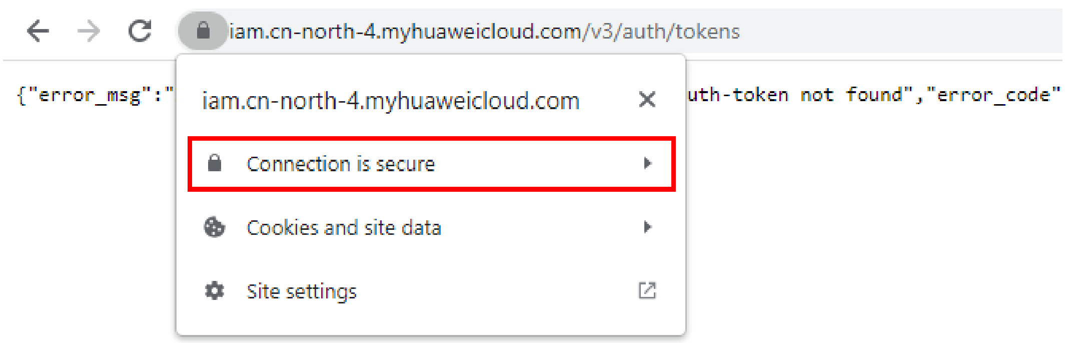
The RESTful API is used to access the KMS. When you enter the URL of the API in the address box of the browser, ignore the failure page in **Step 2**. The browser has automatically downloaded the CA certificate in advance even if the failure page is displayed.

**Step 1** Enter the domain name: Open a browser. In the Huawei Cloud scenario, enter the IAM service domain name (iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens) and KMS domain name (kms.cn-north-4.myhuaweicloud.com/v1.0).

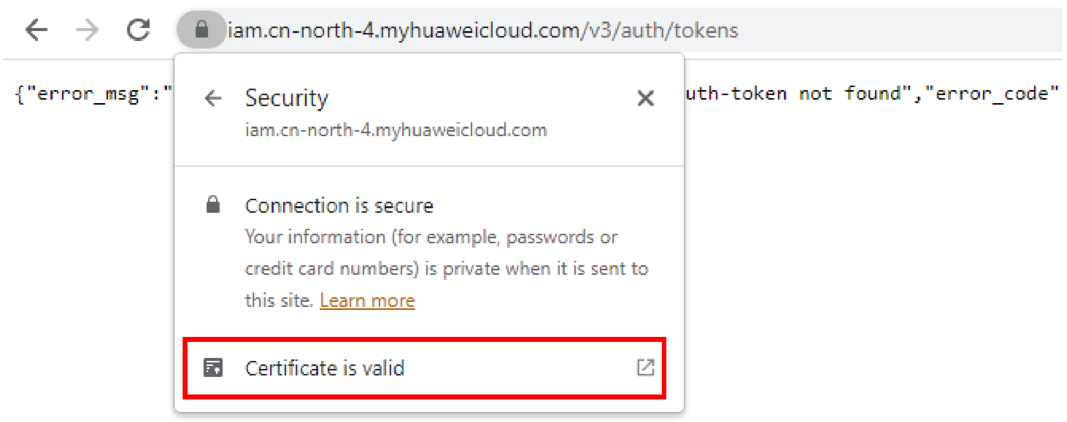
**Step 2** Search for a certificate: Each time you enter a domain name, find the SSL connection information and click the information to view the certificate content.


iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens

```
{"error_msg":"Incorrect IAM authentication information: x-auth-token not found","error_code"
```


iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens

```
{"error_msg":"                                          uth-token not found","error_code"
```


iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens

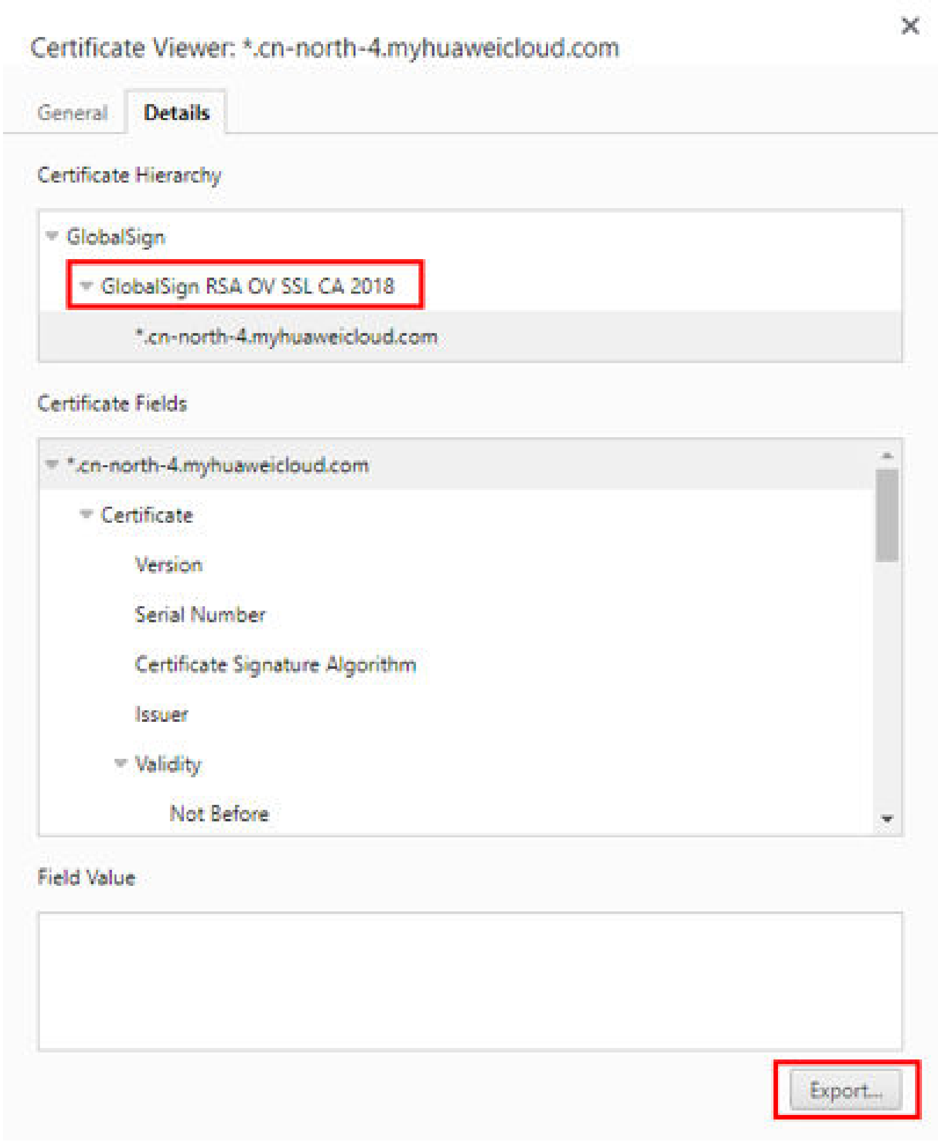
```
{"error_msg":"    uth-token not found","error_code"
```

Security
iam.cn-north-4.myhuaweicloud.com

Connection is secure
Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Learn more

Certificate is valid

**Step 3** Export the certificate. On the **Certificate Viewer** page, certificates may be classified into multiple levels. You only need to select the upper-level certificate of the domain name and click **Export** to generate a certificate file, that is, the required certificate file.


Certificate Viewer: *.cn-north-4.myhuaweicloud.com

General | Details

Certificate Hierarchy
- GlobalSign
  - GlobalSign RSA OV SSL CA 2018
    - *.cn-north-4.myhuaweicloud.com

Certificate Fields
- *.cn-north-4.myhuaweicloud.com
  - Certificate
    - Version
    - Serial Number
    - Certificate Signature Algorithm
    - Issuer
    - Validity
      - Not Before

Field Value

Export...

**Step 4** Upload the certificate: Upload the exported certificate to the application and set the preceding parameters.

**----End**

## 2.5 Encrypted Functions and Stored Procedures

In the current version, only encrypted functions and stored procedures in SQL or PL/pgSQL are supported. Because users are unaware of the creation and execution of functions or stored procedures in an encrypted stored procedure, the syntax has no difference from that of non-encrypted functions and stored procedures.

For details about the syntax of functions and stored procedures, see "User-defined Functions" and "Stored Procedures" in *Developer Guide*.

The **gs_encrypted_proc** system catalog is added to the function or stored procedure for encrypted equality query to store the returned original data type.

For details about the fields in the system catalog, see "System Catalogs and System Views > System Catalogs > GS_ENCRYPTED_PROC" in *Developer Guide*.

### Creating and Executing a Function or Stored Procedure that Involves Encrypted Columns

**Step 1** Create a key. For details, see **Using gsql to Operate an Encrypted Database**.

**Step 2** Create an encrypted table.

```
gaussdb=# CREATE TABLE creditcard_info (
   id_number int,
   name  text,
   credit_card varchar(19) encrypted with (column_encryption_key = cek1, encryption_type =
DETERMINISTIC)
 ) with (orientation=row) distribute by hash(id_number);
CREATE TABLE
```

**Step 3** Insert data.

```
gaussdb=# insert into creditcard_info values(1, 'Avi', '1234567890123456');
INSERT 0 1
gaussdb=# insert into creditcard_info values(2, 'Eli', '2345678901234567');
INSERT 0 1
```

**Step 4** Create a function supporting encrypted equality query.

```
gaussdb=# CREATE FUNCTION f_encrypt_in_sql(val1 text, val2 varchar(19)) RETURNS text AS 'SELECT
name from creditcard_info where name=$1 or credit_card=$2 LIMIT 1' LANGUAGE SQL;
CREATE FUNCTION
gaussdb=# CREATE FUNCTION f_encrypt_in_plpgsql (val1 text, val2 varchar(19), OUT c text) AS $$
 BEGIN
 SELECT into c name from creditcard_info where name=$1 or credit_card =$2 LIMIT 1;
 END; $$
 LANGUAGE plpgsql;
CREATE FUNCTION
```

**Step 5** Execute the function.

```
gaussdb=# SELECT f_encrypt_in_sql('Avi','1234567890123456');
 f_encrypt_in_sql
------------------
 Avi
(1 row)

gaussdb=# SELECT f_encrypt_in_plpgsql('Avi', val2=>'1234567890123456');
```

```
 f_encrypt_in_plpgsql
----------------------
 Avi
(1 row)
```

**----End**

NOTE

1. Because the query, that is, the dynamic query statement executed in a function or stored procedure, is compiled during execution, the table name and column name in the function or stored procedure must be known in the creation phase. The input parameter cannot be used as a table name or column name, or any connection mode.
2. In a function or stored procedure that executes dynamic clauses, data values to be encrypted cannot be contained in the clauses.
3. Among the **RETURNS**, **IN**, and **OUT** parameters, encrypted and non-encrypted parameters cannot be used together. Although the parameter types are all original, the actual types are different.
4. In advanced package interfaces, for example, dbe_output.print_line(), decryption is not performed on the interfaces whose output is printed on the server. This is because when the encrypted data type is forcibly converted into the plaintext original data type, the default value of the data type is printed.
5. In the current version, **LANGUAGE** of functions and stored procedures can only be **SQL** and **PL/pgSQL**, and does not support other procedural languages such as **C** and **Java**.
6. Other functions or stored procedures for querying encrypted columns cannot be executed in a function or stored procedure.
7. In the current version, default values cannot be assigned to variables in DEFAULT or DECLARE statements, and return values in DECLARE statements cannot be decrypted. You can use input parameters and output parameters instead when executing functions.
8. gs_dump cannot be used to back up functions involving encrypted columns.
9. Keys cannot be created in functions or stored procedures.
10. In this version, encrypted functions and stored procedures do not support triggers.
11. Encrypted equality query functions and stored procedures do not support the escape of the PL/pgSQL syntax. The CREATE FUNCTION AS'*Syntax body*' syntax whose syntax body is marked with single quotation marks ('') can be replaced with the CREATE FUNCTION AS $$*Syntax body* $$ syntax.
12. The definition of an encrypted column cannot be modified in an encrypted equality query function or stored procedure, including creating an encrypted table and adding an encrypted column. Because the function is executed on the server, the client cannot determine whether to refresh the cache. The column can be encrypted only after the client is disconnected or the cache of the encrypted column on the client is refreshed.
13. Functions and stored procedures cannot be created using encrypted data types (byteawithoutorderwithequalcol, byteawithoutordercol, _byteawithoutorderwithequalcol or _byteawithoutordercol).
14. If an encrypted function returns a value of an encrypted type, the result cannot be an uncertain row type, for example, RETURN [SETOF] RECORD. You can replace it with a definite row type, for example, RETURN TABLE(columnname typename[, ...]).
15. When an encrypted function is created, the OID of the encrypted column corresponding to a parameter is added to the system catalog gs_encrypted_proc. Therefore, if a table with the same name is deleted and created again, the encrypted function may become invalid and you need to create the encrypted function again.

# 3 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.

## 3.1 Large-Capacity Database

### 3.1.1 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.

### 3.1.2 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:

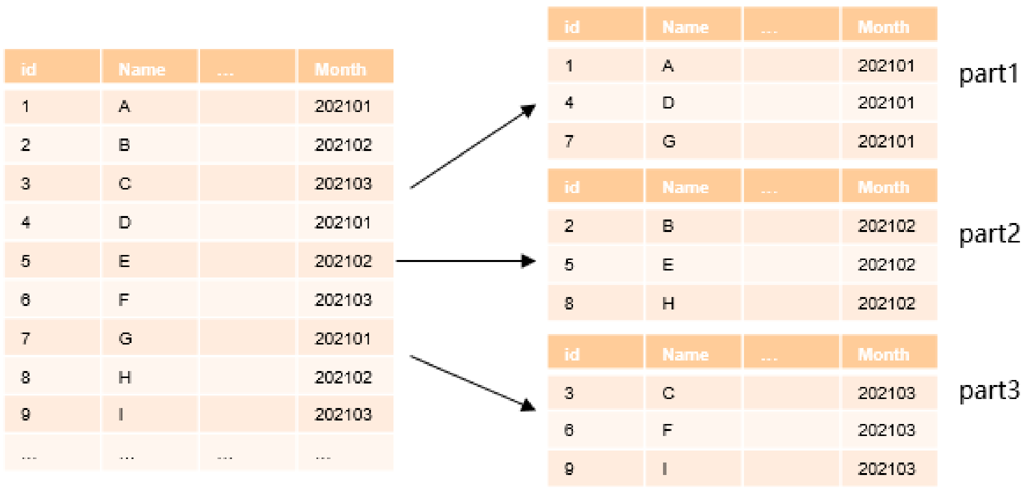
3.x Feature Guide for Distributed Instances - Huawei Cloud

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.

## 3.1.3 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, as shown in **Figure 3-1**, 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 3-2**.

**Figure 3-1** 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

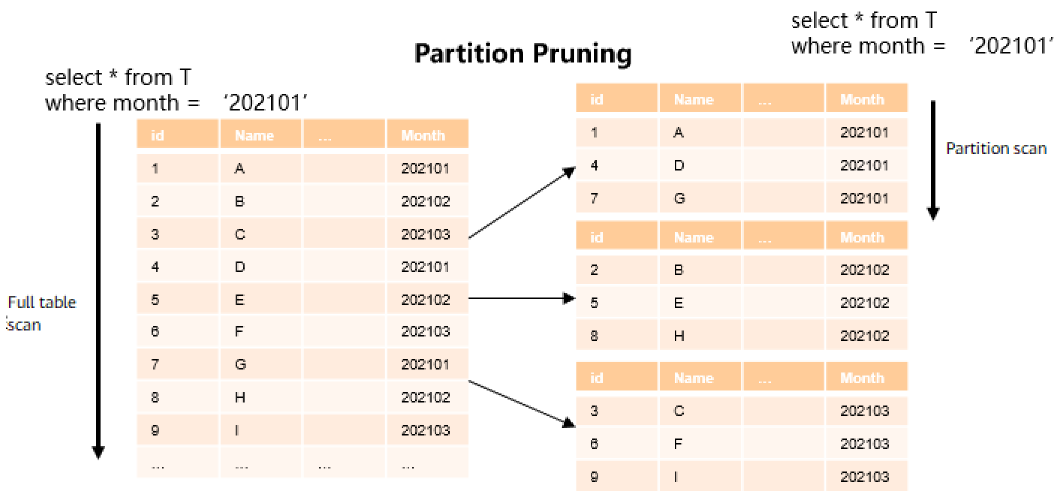
| id | Name | ... | Month |
|---|---|---|---|
| 2 | B | | 202102 |
| 5 | E | | 202102 |
| 8 | H | | 202102 |

part3

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

3.x Feature Guide for Distributed Instances - Huawei Cloud

**Figure 3-2** Example of partition pruning



## 3.1.4 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 dataset to meet differentiated requirements of service scenarios.

## 3.2 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 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 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.

## 3.2.1 Basic Concepts

### 3.2.1.1 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 by 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'**, 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.)
Distribute By: HASH(c1)
Location Nodes: ALL DATANODES
Has OIDs: no
Options: orientation=row, compression=no

-- 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)

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

### 3.2.1.2 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)

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

### 3.2.1.3 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.

## 3.2.2 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 partitioning, hash partitioning based on hash functions, and list partitioning based on data enumeration.

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

### 3.2.2.1 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
   (
       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;
   ```
   **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_with_multiple_keys
   (
       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_with_multiple_keys VALUES(9,5,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(9,20,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(9,21,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(10,5,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(10,15,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(10,20,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(10,21,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(11,5,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(11,20,'a');
gaussdb=# INSERT INTO range_sales_with_multiple_keys VALUES(11,21,'a');

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

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

gaussdb=# SELECT * FROM range_sales_with_multiple_keys 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_with_multiple_keys;
```

**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. If the value of the inserted second column is smaller than the boundary of the next column in the target partition, the values are directly inserted. Otherwise, the comparison of the next columns between the source and target continues.
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)
```

### 3.2.2.2 Hash Partitioning

Hash partitioning uses a hash algorithm to map data to partitions based on partition keys. The GaussDB 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:

```
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)
)
-- 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.

### 3.2.2.3 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 datasets 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 datasets.

**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.

#### 3.2.2.4 Impact of Partitioned Tables on Import Performance

In the GaussDB kernel implementation, compared with the non-partitioned table, the partitioned table has partition routing overheads during data insertion. The overall data insertion overheads include: (1) heap base table insertion and (2) partition routing. 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.

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 with a large amount of data. In the UPDATE scenario, the system searches for the tuple to be updated, deletes the tuple, and then inserts new tuple. Therefore, the performance is not as good as that of a single INSERT statement.

Table 3-1 shows the routing algorithm logic of different partitioning types.

**Table 3-1** 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 |

| Partitioning Type | Routing Algorithm Complexity | Implementation Description |
|---|---|---|
| List partitioning | O(1) | Implemented based on the key-partOid 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.
- In the Arm server scenario, the performance decreases by 20%. 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.

## 3.2.3 Basic Usage of Partitions

### 3.2.3.1 Creating Partitioned Tables

#### Creating Partitioned Tables

The SQL syntax tree is complex due to the powerful and flexible functions of the SQL language. So do partitioned tables. The creation of a partitioned table can be regarded as adding partition attributes to the original non-partitioned table. Therefore, the syntax interface 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 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 */
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. Hash partitioning supports a one-column partition key only.
2. 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.
3. 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.

## 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 section "SQL Reference > SQL Syntax > ALTER TABLE PARTITION" in *Developer Guide*.

### 3.2.3.2 DML Statements for Querying Partitioned Tables

Partitioning is implemented in the database kernel. Therefore, users can query partitioned tables and non-partitioned tables using the same syntax except for querying specified partitions.

For ease of use of partitioned tables, GaussDB allows you to query specified partitions by running **PARTITION** *(partname)* or **PARTITION FOR** *(partvalue)*. 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 partitioned table list_02. */
gaussdb=# CREATE TABLE IF NOT EXISTS list_02
(
    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;
/* Import data. */
INSERT INTO list_02 VALUES(null, 'alice', 'alice data');
INSERT INTO list_02 VALUES(2, null, 'bob data');
INSERT INTO list_02 VALUES(null, null, 'peter data');

/* Query a specified partition. */
-- Query all data in a partitioned table.
gaussdb=# SELECT * FROM list_02 ORDER BY data;
 id | role  |    data
----+-------+------------
    | alice | alice data
  2 |       | bob data
    |       | peter data
(3 rows)
-- Query data in the p_list_2 partition.
gaussdb=# SELECT * FROM list_02 PARTITION (p_list_2) ORDER BY data;
 id | role |   data
----+------+----------
  2 |      | bob data
(1 row)
```

```
-- Query the data of the partition corresponding to (100), that is, partition p_list_4.
gaussdb=# SELECT * FROM list_02 PARTITION FOR (100) ORDER BY data;
 id | role  |    data
----+-------+------------
    | alice | alice data
    |       | peter data
(2 rows)

/* Perform INSERT, UPDATE, and DELETE (IUD) operations on the specified partition. */
-- Delete all data from the p_list_5 partition.
gaussdb=# DELETE FROM list_02 PARTITION (p_list_5);
-- Insert data into the specified partition p_list_7. An error is reported because the data does not comply with the partitioning restrictions.
gaussdb=# INSERT INTO list_02 PARTITION (p_list_7) VALUES(null, 'cherry', 'cherry data');
ERROR:  inserted partition key does not map to the table partition
-- Update the data of the partition to which the partition value 100 belongs, that is, partition p_list_4.
gaussdb=# UPDATE list_02 PARTITION FOR (100) SET data = '';

-- UPSERT
gaussdb=# INSERT INTO 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_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);

-- Drop a table.
gaussdb=#
DROP TABLE list_02;
DROP TABLE list_tmp;
```

## 3.3 Partitioned Table Query Optimization

**NOTE**

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

### 3.3.1 Partition Pruning

#### 3.3.1.1 Static Partition Pruning

For partitioned table query statements with constants 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 types: range partitioning, hash partitioning, and list partitioning.
2. Supported expression types: comparison expression (<, <=, =, >=, >), logical expression, and array expression.

**CAUTION**

- Currently, static pruning does not support subquery expressions.
- 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=# SET max_datanode_for_plan = 1;

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = 1;
                        QUERY PLAN
-------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: datanode1
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = 1

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = 1
 Datanode Name: datanode1
   Partition Iterator
     Output: c1, c2
     Iterations: 1
     ->  Partitioned Seq Scan on public.t1
           Output: c1, c2
           Filter: (t1.c1 = 1)
           Selected Partitions:  1

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 < 1;
                        QUERY PLAN
-------------------------------------------------------------
```

```
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 < 1

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 < 1
 Datanode Name: datanode1
   Partition Iterator
     Output: c1, c2
     Iterations: 1
     ->  Partitioned Seq Scan on public.t1
           Output: c1, c2
           Filter: (t1.c1 < 1)
           Selected Partitions:  1

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 > 11;
                        QUERY PLAN
------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 > 11

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 > 11
 Datanode Name: datanode1
   Partition Iterator
     Output: c1, c2
     Iterations: 2
     ->  Partitioned Seq Scan on public.t1
           Output: c1, c2
           Filter: (t1.c1 > 11)
           Selected Partitions:  2..3

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 is NULL;
                          QUERY PLAN
---------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: datanode1
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 IS NULL

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 IS NULL
 Datanode Name: datanode1
   Partition Iterator
     Output: c1, c2
     Iterations: 1
     ->  Partitioned Seq Scan on public.t1
           Output: c1, c2
           Filter: (t1.c1 IS NULL)
           Selected Partitions:  3

(15 rows)
```

b. Logical expressions

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = 1 AND c2 = 2;
                              QUERY PLAN
----------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: datanode1
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = 1 AND c2 = 2

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = 1 AND c2 = 2
 Datanode Name: datanode1
   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

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = 1 OR c1 = 2;
                              QUERY PLAN
------------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = 1 OR c1 = 2

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = 1 OR c1 = 2
 Datanode Name: datanode1
   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

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE NOT c1 = 1;
                            QUERY PLAN
------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM public.t1 WHERE NOT c1 = 1

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE NOT c1 = 1
 Datanode Name: datanode1
   Partition Iterator
     Output: c1, c2
     Iterations: 3
     ->  Partitioned Seq Scan on public.t1
           Output: c1, c2
           Filter: (t1.c1 <> 1)
           Selected Partitions:  1..3

(15 rows)
```

c. Array expressions

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 IN (1, 2, 3);
                                   QUERY PLAN
---------------------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])
 Datanode Name: datanode1
   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

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = ALL(ARRAY[1,
2, 3]);
                                  QUERY PLAN
--------------------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = ALL (ARRAY[1, 2, 3])

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = ALL (ARRAY[1, 2, 3])
 Datanode Name: datanode1
   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

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 = ANY(ARRAY[1,
2, 3]);
                                  QUERY PLAN
--------------------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])
 Datanode Name: datanode1
   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

(15 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT * FROM t1 WHERE c1 =
SOME(ARRAY[1, 2, 3]);
                                  QUERY PLAN
--------------------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = ANY (ARRAY[1, 2, 3])
 Datanode Name: datanode1
   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

(15 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
---------------------------------------------------------------------------
 Streaming (type: GATHER)
```

```
   Output: public.t1.c1, public.t1.c2
   Node/s: All datanodes
   ->  Partition Iterator
         Output: public.t1.c1, public.t1.c2
         Iterations: 3
         ->  Partitioned Seq Scan on public.t1
               Output: public.t1.c1, public.t1.c2
               Distribute Key: public.t1.c1
               Filter: (SubPlan 1)
               Selected Partitions:  1..3
               SubPlan 1
                 ->  Materialize
                       Output: public.t1.c2
                       ->  Streaming(type: BROADCAST)
                             Output: public.t1.c2
                             Spawn on: All datanodes
                             Consumer Nodes: All datanodes
                             ->  Partition Iterator
                                   Output: public.t1.c2
                                   Iterations: 2
                                   ->  Partitioned Seq Scan on public.t1
                                         Output: public.t1.c2
                                         Distribute Key: public.t1.c1
                                         Filter: (public.t1.c1 > 10)
                                         Selected Partitions:  2..3
(26 rows)

-- Clean up the environment.
gaussdb=# DROP TABLE t1;
```

### 3.3.1.2 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.

#### 3.3.1.2.1 Dynamic PBE Pruning

Dynamic PBE pruning is supported in the following scenarios:

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

- 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)
);

gaussdb=# PREPARE p1(int) AS SELECT * FROM t1 WHERE c1 = $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p1(1);
                         QUERY PLAN
-----------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: datanode1
   Node expr: $1
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = $1

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = $1
 Datanode Name: datanode1
   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)

(16 rows)
```
  b. Logical expressions
```
gaussdb=# PREPARE p2(INT, INT) AS SELECT * FROM t1 WHERE c1 = $1 AND c2 = $2;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p2(1, 2);
                              QUERY PLAN
----------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: datanode1
   Node expr: $1
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = $1 AND c2 = $2

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = $1 AND c2 = $2
 Datanode Name: datanode1
   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)

(16 rows)
```

c. Implicit type conversion

```
gaussdb=# set plan_cache_mode = 'force_generic_plan';
gaussdb=# PREPARE p3(TEXT) AS SELECT * FROM t1 WHERE c1 = $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p3('12');
                             QUERY PLAN
--------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: datanode1
   Node expr: $1
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1 = $1::bigint

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1 = $1::bigint
 Datanode Name: datanode1
   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)

(16 rows)
```

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

a. Subquery expressions

```
gaussdb=# PREPARE p4(INT) AS SELECT * FROM t1 WHERE c1 = ALL(SELECT c2 FROM t1
WHERE c1 > $1);
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p4(1);
                               QUERY PLAN
-------------------------------------------------------------------------
 Streaming (type: GATHER)
   Output: public.t1.c1, public.t1.c2
   Node/s: All datanodes
   ->  Partition Iterator
         Output: public.t1.c1, public.t1.c2
         Iterations: 3
         ->  Partitioned Seq Scan on public.t1
               Output: public.t1.c1, public.t1.c2
               Distribute Key: public.t1.c1
               Filter: (SubPlan 1)
               Selected Partitions:  1..3
               SubPlan 1
                 ->  Materialize
                       Output: public.t1.c2
                       ->  Streaming(type: BROADCAST)
                             Output: public.t1.c2
                             Spawn on: All datanodes
                             Consumer Nodes: All datanodes
                             ->  Partition Iterator
                                   Output: public.t1.c2
                                   Iterations: 3
                                   ->  Partitioned Seq Scan on public.t1
                                         Output: public.t1.c2
                                         Distribute Key: public.t1.c1
                                         Filter: (public.t1.c1 > 1)
                                         Selected Partitions:  1..3
(26 rows)
```

b. Implicit type conversion failure

```
gaussdb=# PREPARE p5(name) AS SELECT * FROM t1 WHERE c1 = $1;
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p5('12');
                              QUERY PLAN
--------------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM public.t1 WHERE c1::text = '12'::text

 Remote SQL: SELECT c1, c2 FROM public.t1 WHERE c1::text = '12'::text
 Datanode Name: datanode1
   Partition Iterator
     Output: c1, c2
     Iterations: 3
     ->  Partitioned Seq Scan on public.t1
           Output: c1, c2
           Filter: ((t1.c1)::text = '12'::text)
           Selected Partitions:  1..3

(15 rows)
```

c. STABLE and VOLATILE functions

```
gaussdb=# create sequence seq;
gaussdb=# PREPARE p6(TEXT) AS SELECT * FROM t1 WHERE c1 = currval($1);-- The VOLATILE
function does not support pruning.
PREPARE
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) EXECUTE p6('seq');
                               QUERY PLAN
--------------------------------------------------------------------------
 Data Node Scan
   Output: t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT c1, c2 FROM ONLY public.t1 WHERE true
   Coordinator quals: ((t1.c1)::numeric = currval(('seq'::text)::regclass))

 Remote SQL: SELECT c1, c2 FROM ONLY public.t1 WHERE true
 Datanode Name: datanode1
   Partition Iterator
     Output: c1, c2
     Iterations: 3
     ->  Partitioned Seq Scan on public.t1
           Output: c1, c2
           Selected Partitions:  1..3

(15 rows)

-- Clean up the environment.
gaussdb=# DROP TABLE t1;
```

### 3.3.1.2.2 Dynamic Parameterized Path Pruning

Dynamic parameterized path pruning is supported in the following scenarios:

1. Supported partitioning types: range partitioning, hash partitioning, and list partitioning.
2. Supported operator types: index scan, index-only scan, and bitmap scan.
3. 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 /*+ nestloop(t1 t2) indexscan(t1)
indexscan(t2) */ * FROM t2 JOIN t1 ON t1.c1 = t2.c1;
                                                                   QUERY
PLAN
------------------------------------------------------------------------------------------------------------------------
-----------------------------------
 Data Node Scan
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT/*+ NestLoop(t1 t2) IndexScan(t1) IndexScan(t2)*/ t2.c1, t2.c2, t1.c1,
t1.c2 FROM public.t2 JOIN public.t1 ON t1.c1 = t2.c1

 Remote SQL: SELECT/*+ NestLoop(t1 t2) IndexScan(t1) IndexScan(t2)*/ t2.c1, t2.c2, t1.c1, t1.c2
FROM public.t2 JOIN public.t1 ON t1.c1 = t2.c1
 Datanode Name: datanode1
   Nested Loop
     Output: t2.c1, t2.c2, t1.c1, t1.c2
     ->  Partition Iterator
           Output: t2.c1, t2.c2
           Iterations: 3
           ->  Partitioned Index Scan using t2_c1 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 t1_c1 on public.t1
                 Output: t1.c1, t1.c2
                 Index Cond: (t1.c1 = t2.c1)
                 Selected Partitions:  1 (ppi-pruning)

(23 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT /*+ nestloop(t1 t2) indexscan(t1)
indexscan(t2) */ * FROM t2 JOIN t1 ON t1.c1 < t2.c1;
                            QUERY PLAN
```

```
--------------------------------------------------------------------
 Streaming (type: GATHER)
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   Node/s: All datanodes
   ->  Nested Loop
         Output: t2.c1, t2.c2, t1.c1, t1.c2
         ->  Streaming(type: BROADCAST)
               Output: t2.c1, t2.c2
               Spawn on: All datanodes
               Consumer Nodes: All datanodes
               ->  Partition Iterator
                     Output: t2.c1, t2.c2
                     Iterations: 3
                     ->  Partitioned Seq Scan on public.t2
                           Output: t2.c1, t2.c2
                           Distribute Key: t2.c1
                           Selected Partitions:  1..3
         ->  Partition Iterator
               Output: t1.c1, t1.c2
               Iterations: PART
               ->  Partitioned Index Scan using t1_c1 on public.t1
                     Output: t1.c1, t1.c2
                     Distribute Key: t1.c1
                     Index Cond: (t1.c1 < t2.c1)
                     Selected Partitions:  1 (ppi-pruning)
(24 rows)

gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT /*+ nestloop(t1 t2) indexscan(t1)
indexscan(t2) */ * FROM t2 JOIN t1 ON t1.c1 < t2.c1;
                            QUERY PLAN
--------------------------------------------------------------------
 Streaming (type: GATHER)
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   Node/s: All datanodes
   ->  Nested Loop
         Output: t2.c1, t2.c2, t1.c1, t1.c2
         ->  Streaming(type: BROADCAST)
               Output: t2.c1, t2.c2
               Spawn on: All datanodes
               Consumer Nodes: All datanodes
               ->  Partition Iterator
                     Output: t2.c1, t2.c2
                     Iterations: 3
                     ->  Partitioned Seq Scan on public.t2
                           Output: t2.c1, t2.c2
                           Distribute Key: t2.c1
                           Selected Partitions:  1..3
         ->  Partition Iterator
               Output: t1.c1, t1.c2
               Iterations: PART
               ->  Partitioned Index Scan using t1_c1 on public.t1
                     Output: t1.c1, t1.c2
                     Distribute Key: t1.c1
                     Index Cond: (t1.c1 > t2.c1)
                     Selected Partitions:  1..3 (ppi-pruning)
(24 rows)
```

b. Logical expressions

```
gaussdb=# EXPLAIN (VERBOSE ON, COSTS OFF) SELECT /*+ nestloop(t1 t2) indexscan(t1)
indexscan(t2) */ * FROM t2 JOIN t1 ON t1.c1 = t2.c1 AND t1.c2 = 2;
                                                                QUERY
PLAN
---------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------
 Data Node Scan
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   Node/s: All datanodes
   Remote query: SELECT/*+ NestLoop(t1 t2) IndexScan(t1) IndexScan(t2)*/ t2.c1, t2.c2, t1.c1,
t1.c2 FROM public.t2 JOIN public.t1 ON t1.c1 = t2.c1 AND t1.c2 = 2
```

```
 Remote SQL: SELECT/*+ NestLoop(t1 t2) IndexScan(t1) IndexScan(t2)*/ t2.c1, t2.c2, t1.c1, t1.c2
FROM public.t2 JOIN public.t1 ON t1.c1 = t2.c1 AND t1.c2 = 2
 Datanode Name: datanode1
   Nested Loop
     Output: t2.c1, t2.c2, t1.c1, t1.c2
     ->  Partition Iterator
           Output: t1.c1, t1.c2
           Iterations: 3
           ->  Partitioned Index Scan using t1_c2 on public.t1
                 Output: t1.c1, t1.c2
                 Index Cond: (t1.c2 = 2)
                 Selected Partitions:  1..3
     ->  Partition Iterator
           Output: t2.c1, t2.c2
           Iterations: PART
           ->  Partitioned Index Scan using t2_c1 on public.t2
                 Output: t2.c1, t2.c2
                 Index Cond: (t2.c1 = t1.c1)
                 Selected Partitions:  1..3 (ppi-pruning)

(24 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 /*+ nestloop(t1 t2) */ * FROM t2 JOIN
t1 ON t1.c1 = t2.c1 OR t1.c2 = 2;
WARNING:  Statistics in some tables or columns(public.t2.c1, public.t1.c1, public.t1.c2) are not
collected.
HINT:  Do analyze for them in order to generate optimized plan.
                              QUERY PLAN
----------------------------------------------------------------------
 Streaming (type: GATHER)
   Output: t2.c1, t2.c2, t1.c1, t1.c2
   Node/s: All datanodes
   ->  Nested Loop
         Output: t2.c1, t2.c2, t1.c1, t1.c2
         ->  Streaming(type: BROADCAST)
               Output: t2.c1, t2.c2
               Spawn on: All datanodes
               Consumer Nodes: All datanodes
               ->  Partition Iterator
                     Output: t2.c1, t2.c2
                     Iterations: 3
                     ->  Partitioned Seq Scan on public.t2
                           Output: t2.c1, t2.c2
                           Distribute Key: t2.c1
                           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
                     Distribute Key: t1.c1
                     Recheck Cond: ((t1.c1 = t2.c1) OR (t1.c2 = 2))
                     Selected Partitions:  1..3
                     ->  BitmapOr
                           ->  Partitioned Bitmap Index Scan on t1_c1
                                 Index Cond: (t1.c1 = t2.c1)
                           ->  Partitioned Bitmap Index Scan on t1_c2
                                 Index Cond: (t1.c2 = 2)
(29 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;
WARNING:  Statistics in some tables or columns(public.t1.c1, public.t3.c1) are not collected.
```

```
HINT:  Do analyze for them in order to generate optimized plan.
                             QUERY PLAN
--------------------------------------------------------------------
 Streaming (type: GATHER)
   Output: t1.c1, t1.c2, t3.c1, t3.c2
   Node/s: All datanodes
   ->  Nested Loop
         Output: t1.c1, t1.c2, t3.c1, t3.c2
         Join Filter: (t1.c1 = ((t3.c1)::bigint))
         ->  Partition Iterator
               Output: t1.c1, t1.c2
               Iterations: 3
               ->  Partitioned Index Scan using t1_c1 on public.t1
                     Output: t1.c1, t1.c2
                     Distribute Key: t1.c1
                     Selected Partitions:  1..3
         ->  Materialize
               Output: t3.c1, t3.c2, ((t3.c1)::bigint)
               ->  Streaming(type: REDISTRIBUTE)
                     Output: t3.c1, t3.c2, ((t3.c1)::bigint)
                     Distribute Key: ((t3.c1)::bigint)
                     Spawn on: All datanodes
                     Consumer Nodes: All datanodes
                     ->  Seq Scan on public.t3
                           Output: t3.c1, t3.c2, t3.c1
                           Distribute Key: t3.c1
(23 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)

-- Clean up the environment.
gaussdb=# DROP TABLE t1;
gaussdb=# DROP TABLE t2;
gaussdb=# DROP TABLE t3;
```

## 3.3.2 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 supports the global non-partitioned index and local partitioned index.

**Figure 3-3** Global non-partitioned index

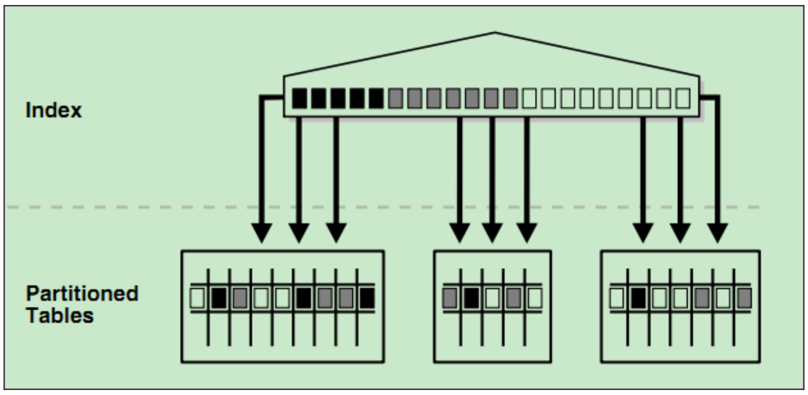


**Figure 3-4** Global partitioned index

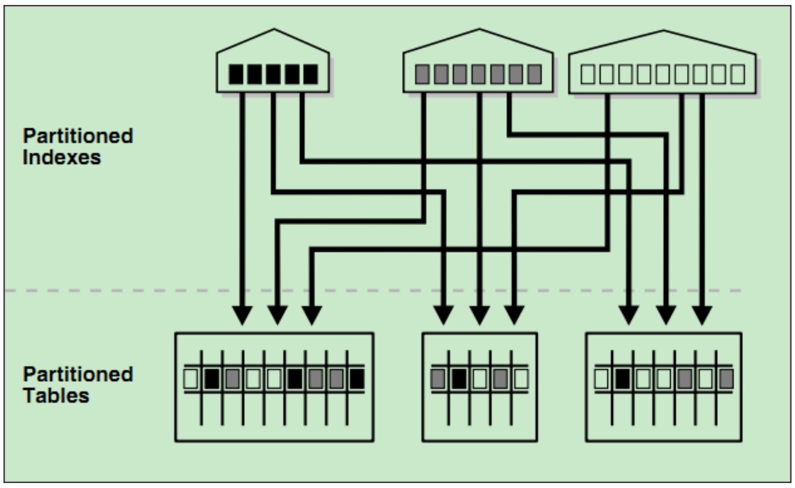
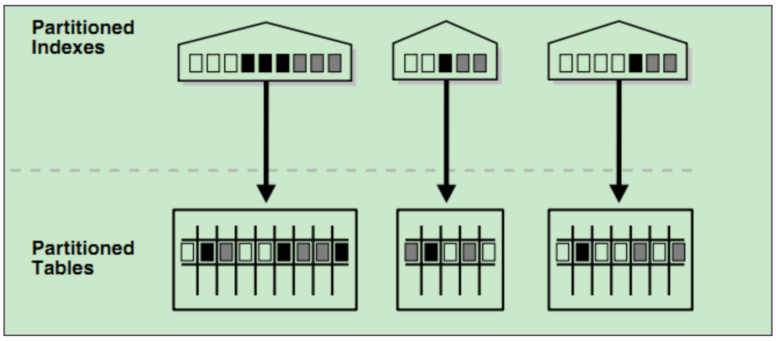


**Figure 3-5** Local partitioned index



## 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 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 test table has been created:
    ```
    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 the name of 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
  ```
  Cleanup example:
  ```
  gaussdb=# DROP TABLE web_returns_p2;
  ```

## 3.4 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 and PARTITION FOR to specify partitions. For PARTITION, you need to specify the partition name. For PARTITION 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: DDL operations on distributed partitioned tables lock the entire table. Cross-partition DDL-DQL/DML concurrency is not supported.

## 3.4.1 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.

### 3.4.1.1 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**.

### 3.4.1.2 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.

## 3.4.2 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.

You can run **ALTER TABLE DROP PARTITION** to delete any partition from a range 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.

### 3.4.3 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 access control constraints.

You can use 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;
```

### 3.4.4 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.

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');
```

### 3.4.5 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.
- 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.

### 3.4.5.1 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.

### 3.4.5.2 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.

## 3.4.6 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 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.

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;
```

## 3.4.7 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.

You can use 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;
```

## 3.4.8 RENAME PARTITION

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

### 3.4.8.1 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;
```

### 3.4.8.2 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.

## 3.4.9 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';
```

## 3.4.10 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.

### 3.4.10.1 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;
```

### 3.4.10.2 Invalidating/Rebuilding Local Indexes of a Partition

- You can run **ALTER INDEX PARTITION** to invalidate or rebuild local indexes of a partition.
- You can run **ALTER TABLE MODIFY PARTITION** to invalidate or rebuild all indexes of a specified partition in a 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;
```

## 3.5 System Views & DFX Related to Partitioned Tables

### 3.5.1 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 section "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_PART_INDEXES: stores information about all local indexes.
   - ADM_IND_PARTITIONS: stores information about all index partitions.
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_PART_INDEXES: stores local index information accessible to the current user.
   - DB_IND_PARTITIONS: stores information about index partitions 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_PART_INDEXES: stores local indexes owned by the current user.
   - MY_IND_PARTITIONS: stores information about index partitions owned by the current user.

### 3.5.2 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)
  ```

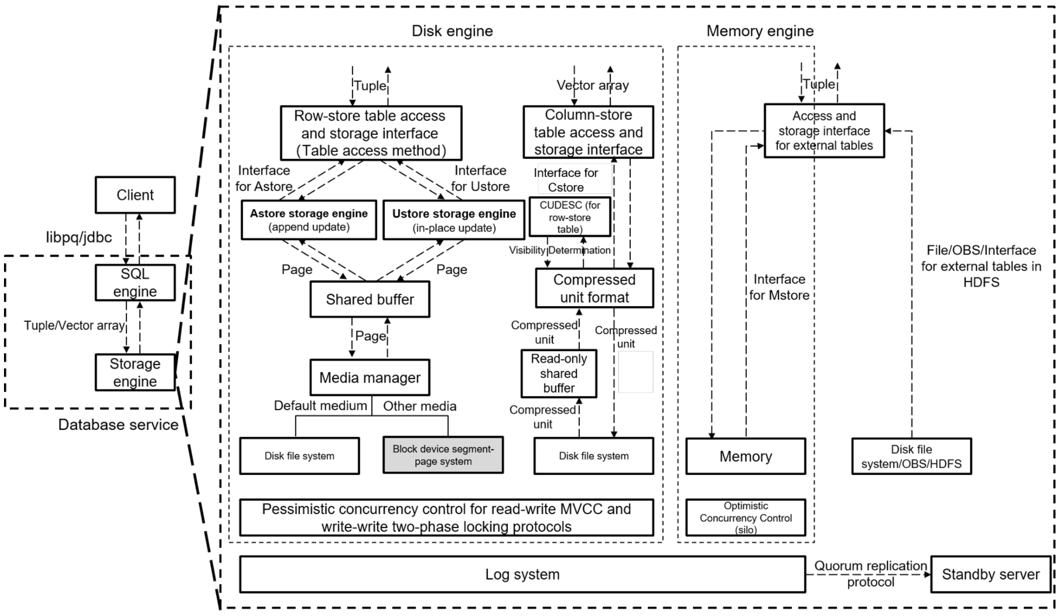
# 4 Storage Engine

## 4.1 Storage Engine Architecture

### 4.1.1 Overview

#### 4.1.1.1 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 interfaces 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.


Disk engine
Memory engine
Tuple
Vector array
Row-store table access and storage interface (Table access method)
Column-store table access and storage interface
Access and storage interface for external tables
Client
libpq/jdbc
SQL engine
Tuple/Vector array
Storage engine
Database service
Interface for Astore
Interface for Ustore
Interface for Cstore
Astore storage engine (append update)
Ustore storage engine (in-place update)
CUDESC (for row-store table)
Visibility Determination
Page
Shared buffer
Compressed unit format
Compressed unit
Read-only shared buffer
Media manager
Default medium
Other media
Disk file system
Block device segment-page system
Interface for Mstore
File/OBS/Interface for external tables in HDFS
Memory
Disk file system/OBS/HDFS
Pessimistic concurrency control for read-write MVCC and write-write two-phase locking protocols
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 upward provide interfaces to read tuples from or write tuples to the SQL engine, downward perform read and write operations 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 interface. 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.

### 4.1.1.2 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.

## 4.1.2 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**. For details about the parameter, see "Configuring Running Parameters > GUC Parameters" in *Administrator Guide*. If this parameter is set to **on**, a Ustore table is created. 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)
```

## 4.1.3 Storage Engine Update Description

### 4.1.3.1 GaussDB 503

- Adapted Ustore to distributed deployment/parallel query/global temporary table/full vacuum/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 Dignose 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.

### 4.1.3.2 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: distributed deployment/parallel query/table sampling/global temporary table/online creation/index rebuild/ultimate RTO/full vacuum/column constraints such as DEFERRABLE and INITIALLY DEFERRED.

## 4.2 Astore Storage Engine

### 4.2.1 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.

## 4.3 Ustore Storage Engine

### 4.3.1 Overview

Unified storage (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.

#### 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).

### 4.3.1.1 Ustore Features and Specifications

#### 4.3.1.1.1 Feature Constraints

| 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 | × |

#### 4.3.1.1.2 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.

### 4.3.1.2 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
```

### 4.3.1.3 Best Practices of Ustore

#### 4.3.1.3.1 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 not advised 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*.

#### 4.3.1.3.2 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.

#### 4.3.1.3.3 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;"
```

#### 4.3.1.3.4 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"
```

#### 4.3.1.3.5 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_increse_speed, where **avg_space_increse_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.

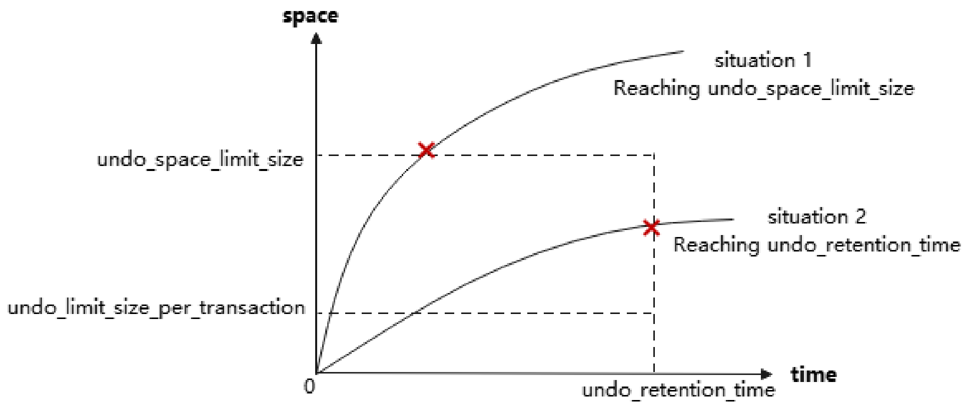
   After adjusting the value of **undo_space_limit_size**, you can increase the value of **undo_limit_size_per_transaction**, which ranges from 2 MB to 16 TB. The default value is **32GB**. It is recommended that the value of **undo_limit_size_per_transaction** be less than or equal to that of **undo_space_limit_size**, 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_increse_speed + curr_used_undo_size, where **avg_space_increse_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.

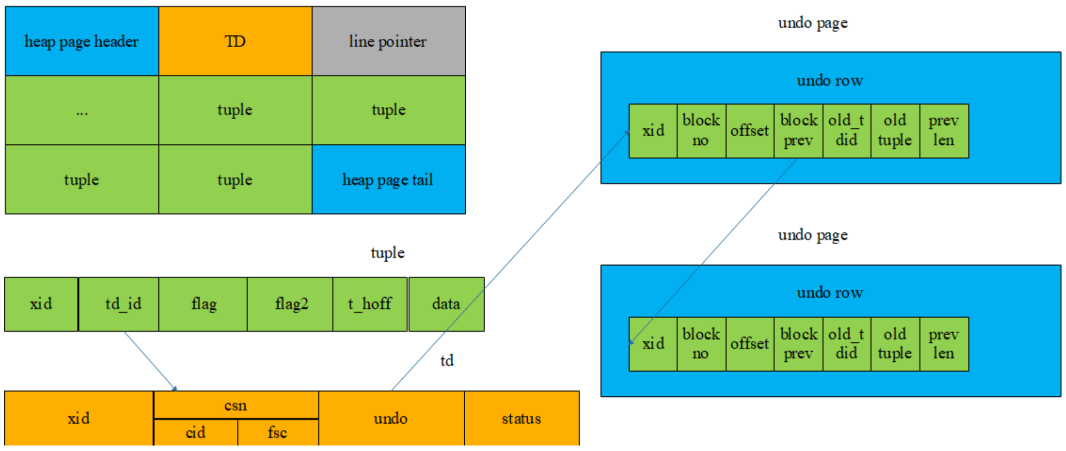


For example, assume that **undo_space_limit_size** is set to **1GB**, and **undo_retention_time** is set to **900s**. If the size of historical data generated within 900s is less than 1 GB x 0.8, the system recycles the data generated within 900s. If the data exceeds 1 GB x 0.8 generated within 900s, 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**.

## 4.3.2 Storage Format

### 4.3.2.1 Relation

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

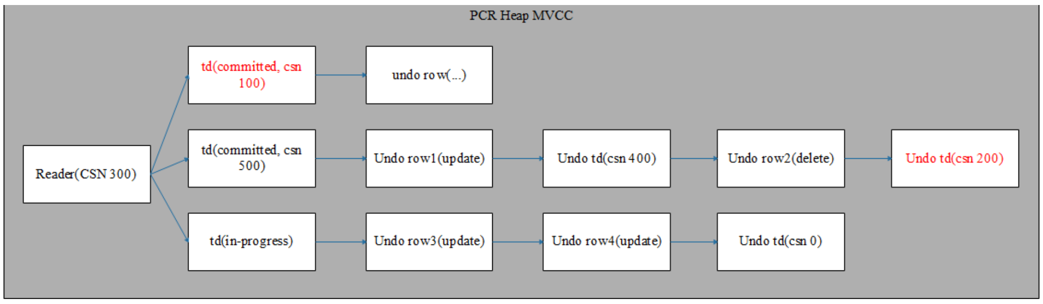


1. The heap multi-version management is row-level multi-version management based on tuples.
2. When a transaction modifies a record, historical data is recorded in an undo row.
3. The address of the generated undo row (zone_id, block no, page offset) is recorded in **td_id** in a tuple.
4. New data is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

3.x Feature Guide for Distributed Instances - Huawei Cloud

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

### 4.3.2.1.2 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. Historical versions of old snapshots can be obtained through undo records.

### 4.3.2.1.3 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 yes, perform the insert operation after the corresponding block number is returned. If no, expand the page logic.

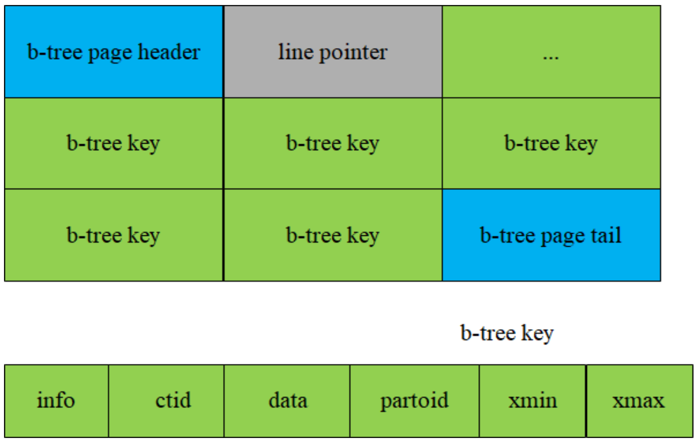
The FSM structure corresponding to each table or partition is stored in an independent FSM file. The FSM file and the table data are stored in the same directory. For example, if the data file corresponding to table **t1** is **32181**, the corresponding FSM file is **32181_fsm**. FSM is stored in the 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.

## 4.3.2.2 Index

The UB-tree is enhanced as follows:

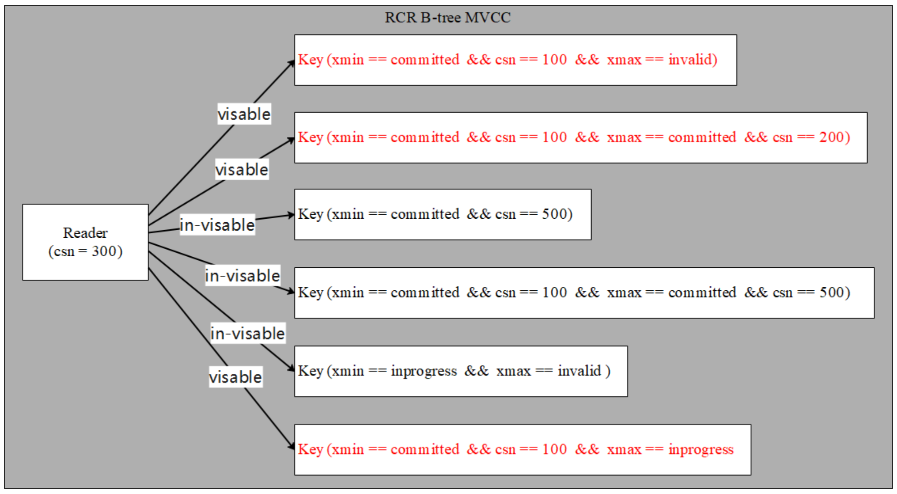
1. Added the MVCC capability.
2. Added the capability of recycling independent empty pages.

#### 4.3.2.2.1 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.

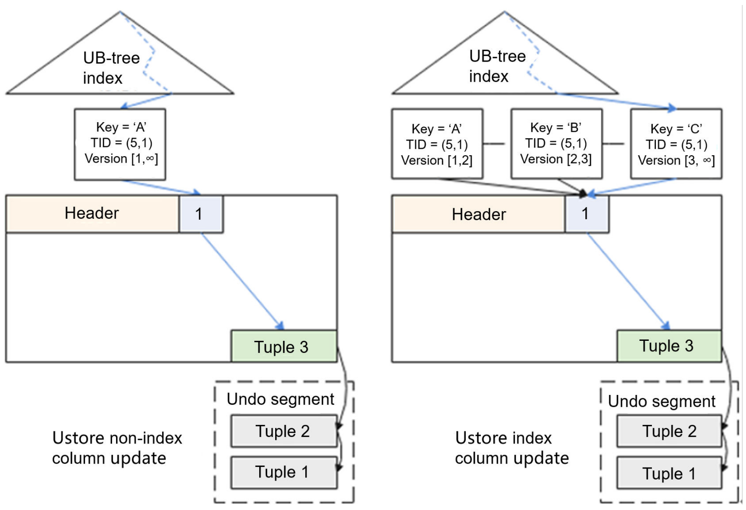
#### 4.3.2.2.2 RCR UB-tree Visibility Mechanism


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 scanning and index-only scanning.
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.

### 4.3.2.2.3 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**: For Ustore, data update operations on UB-tree index columns are different from those on Astore. Data update includes index column update and non-index column update. The following figure shows the processing of UB-tree data update.


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

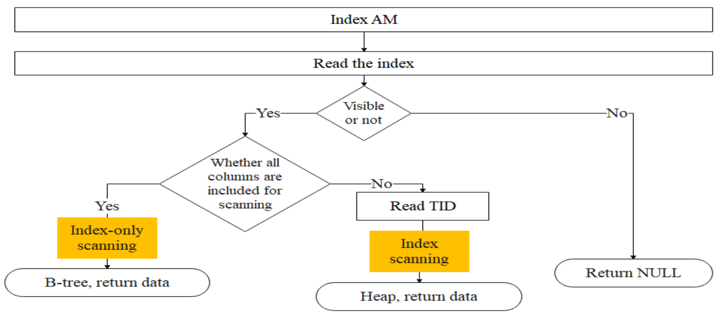
  The preceding figure shows the differences between UB-tree updates in index columns and non-index columns.

  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 scanning and index-only scanning.

  For index scanning:

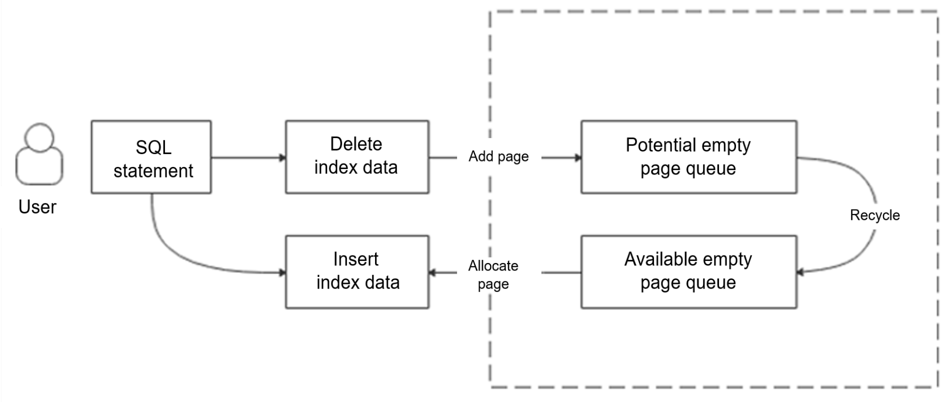
  a. If the index column contains all columns to be scanned (index-only scanning), binary search is performed on indexes based on the scanning conditions. If a tuple that meets the conditions is found, data is returned.
  b. If the index column does not contain all columns to be scanned (index scanning), binary search is performed on indexes based on the scanning 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.



### 4.3.2.2.4 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.

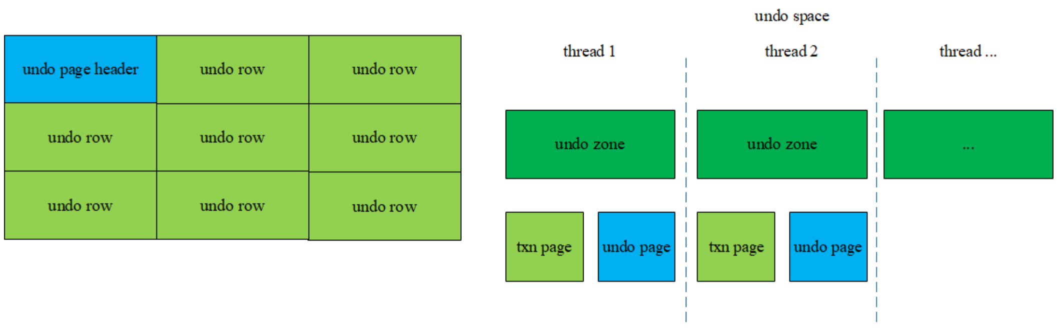
3. **Available queue > Index empty page**

   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.

### 4.3.2.3 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.

#### 4.3.2.3.1 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.

#### 4.3.2.3.2 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
```

#### 4.3.2.3.3 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 server. As for the standby server, 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).

## 4.3.3 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.

### 4.3.3.1 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.

### 4.3.3.2 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 working 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.

## 4.3.4 Flashback Restoration

Flashback restoration 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 modifications can be retrieved only by means of backup/restoration or point-in-time recovery (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**

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

### 4.3.4.1 Flashback Query

## Background

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 an earlier version to obtain the data of the specified 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 historical 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.

Note: 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. In GTM-free mode, there is no globally unified CSN. Therefore, flashback in CSN mode is not supported.

## 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);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'col1' as the distribution column by
default.
```

```
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
-- Query the CSN.
gaussdb=# select int8in(xidout(next_csn)) from gs_get_next_xid_csn();
  int8in
----------
 79351682
 79351682
 79351682
 79351682
 79351682
 79351682
(6 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
```

### 4.3.4.2 Flashback Table

#### Background

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 a table.
gaussdb=# CREATE TABLE "public".flashtest (col1 INT,col2 TEXT) with(storage_type=ustore);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'col1' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
-- Query the CSN.
gaussdb=# select int8in(xidout(next_csn)) from gs_get_next_xid_csn();
  int8in
----------
 79352065
 79352065
 79352065
 79352065
 79352065
 79352065
(6 rows)
-- Query the current timestamp.
gaussdb=# select now();
              now
-------------------------------
 2023-09-13 19:46:34.102863+08
(1 row)
-- View the flashtest table.
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
    1 | INSERT1
    2 | INSERT2
    4 | INSERT4
    5 | INSERT5
    6 | INSERT6
(6 rows)
-- Flash a table back to a specific CSN.
gaussdb=# TIMECAPSULE TABLE flashtest TO CSN 79352065;
TimeCapsule Table
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
------+------
(0 rows)
gaussdb=# select now();
              now
-------------------------------
 2023-09-13 19:52:21.551028+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 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
```

### 4.3.4.3 Flashback DROP/TRUNCATE

#### Background

- 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. For details, contact the administrator.
- 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. For details, contact the administrator.

#### 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
  ```

#### Parameter Description

- 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 objects from the recycle bin.

You can specify either the original user-defined 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 the table you want to retrieve is found.
- When a dropped table is restored, only the base table name is restored, and the names of other objects remain the same as those in the recycle bin. You can run the DDL command to manually change the names of other objects as required.
- The recycle bin does not support write operations such as DML, DCL, and DDL, and does not support DQL query operations (will be 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 exis" is displayed when flashback is performed on DDL operations, such as changing namespaces and table names.
- If the base table has a truncate trigger, the trigger fails when you truncate the target table together. The target table can only be truncated manually.

- 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);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
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 deleted 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 deleted 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);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
-- Create the flashtest_index index for the flashtest table.
gaussdb=# create index flashtest_index on flashtest(id);
CREATE INDEX
-- View basic information about the flashtest table.
gaussdb=# \d+ flashtest
                        Table "public.flashtest"
 Column |  Type   | Modifiers | Storage  | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
 id     | integer |           | plain    |              |
 name   | text    |           | extended |              |
Indexes:
    "flashtest_index" ubtree (id) WITH (storage_type=USTORE) TABLESPACE pg_default
Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, storage_type=ustore, compression=no, toast.storage_type=ustore

-- 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 deleted 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 cleared.
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);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
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 deleted 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 deleted 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 deleted 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
-- Clear the recycle bin.
gaussdb=# PURGE RECYCLEBIN;
PURGE RECYCLEBIN
-- Check the recycle bin. The recycle bin is cleared.
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);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
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
-- Clear the recycle bin.
gaussdb=# PURGE RECYCLEBIN;
PURGE RECYCLEBIN
-- Check the recycle bin. The recycle bin is cleared.
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)
```

## 4.3.5 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<br>• Tuple (non-user data) information<br>• Damaged pages and tuples<br>• Tuple visibility problems<br>• Verification errors | gs_parse_page_bypath |
| | Index recycle queue (URQ) | Parses key information in the URQ. | • UB-tree index space expansion<br>• UB-tree index space recycle exceptions<br>• Verification errors | gs_urq_dump_stat |

| View Type | Type | Function | Application Scenario | Function |
|---|---|---|---|---|
| | Rollback segment (undo) | Parses the specified undo record, excluding the old tuple data. | • Expanded undo space<br>• Undo recycling exceptions<br>• Rollback exceptions<br>• Routine maintenance<br>• Verification errors<br>• Visibility judgment exceptions<br>• 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 |
| | | 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<br>• Log replay errors<br>• 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 |

| View Type | Type | Function | Application Scenario | Function |
|---|---|---|---|---|
| | | 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<br>● Undo resource monitoring | gs_stat_undo |
| | Write ahead log (WAL) | Collects statistics of the memory status table when WALs are written to disks. | ● WAL write/disk flushing monitoring<br>● 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<br>● Visibility issues<br>● Log playback errors | ANALYZE VERIFY |
| | | Checks whether physical files of the current database in the current instance are lost. | Lost files | gs_verify_data_file |

| View Type | Type | Function | Application Scenario | Function |
|---|---|---|---|---|
| | Index recycle (URQ) | Checks whether the data of the URQ (potential queue/available queue/single page) is normal. | ● UB-tree index space expansion<br>● UB-tree index space reclamation exceptions | gs_verify_urq |
| | Rollback segment (undo) | Checks whether undo records are normal offline. | ● Abnormal or damaged undo records<br>● Visibility issues<br>● Abnormal or damaged rollback | gs_verify_undo_record |
| | | Checks whether the transaction slot data is normal offline. | ● Abnormal or damaged undo records<br>● Visibility issues<br>● Abnormal or damaged rollback | gs_verify_undo_slot |
| | | Checks whether the undo metadata is normal offline. | ● Node startup failure caused by undo metadata<br>● Undo space reclamation exceptions<br>● 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 server based on the standby server. | Lost heap tables/ Indexes/undo files | gs_repair_file |
| | Heap table/ Index/ Undo page | Checks and restores damaged pages on the primary server based on the standby server. | Damaged heap tables/ indexes/undo pages | gs_verify_and_tryrepair_page |
| | | Restores the pages of the primary server based on the pages of the standby server. | | 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 |

## 4.3.6 Common Problems and Troubleshooting Methods

### 4.3.6.1 Snapshot Too Old

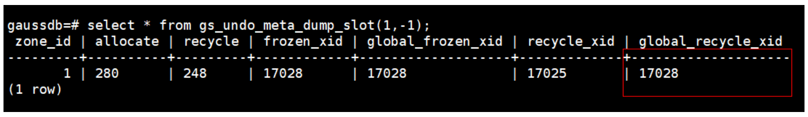
Undo space cannot save historical data if the execution time of the query SQL statement is too long or other reasons. Therefore, an error may be reported if the historical data is forcibly recycled. Generally, the rollback segment space needs to be expanded. However, the specific problem needs to be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

#### 4.3.6.1.1 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.

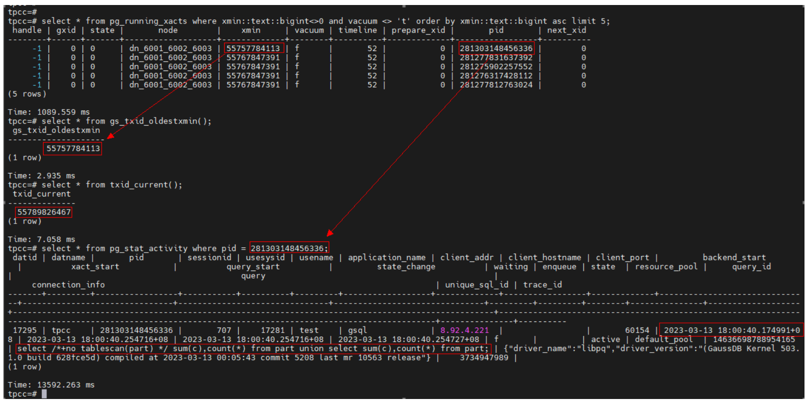
2. The value of **global_recycle_xid** (global recycling XID of the Undo subsystem) does not change for a long time.



```
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** obtained by querying active transactions in pg_running_xacts is the same as that of gs_txid_oldestxmin and the thread execution time obtained by querying pg_stat_activity 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_oldextxmin();
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

Use **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.)

### 4.3.6.1.2 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**.

### 4.3.6.2 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.

### 4.3.6.3 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

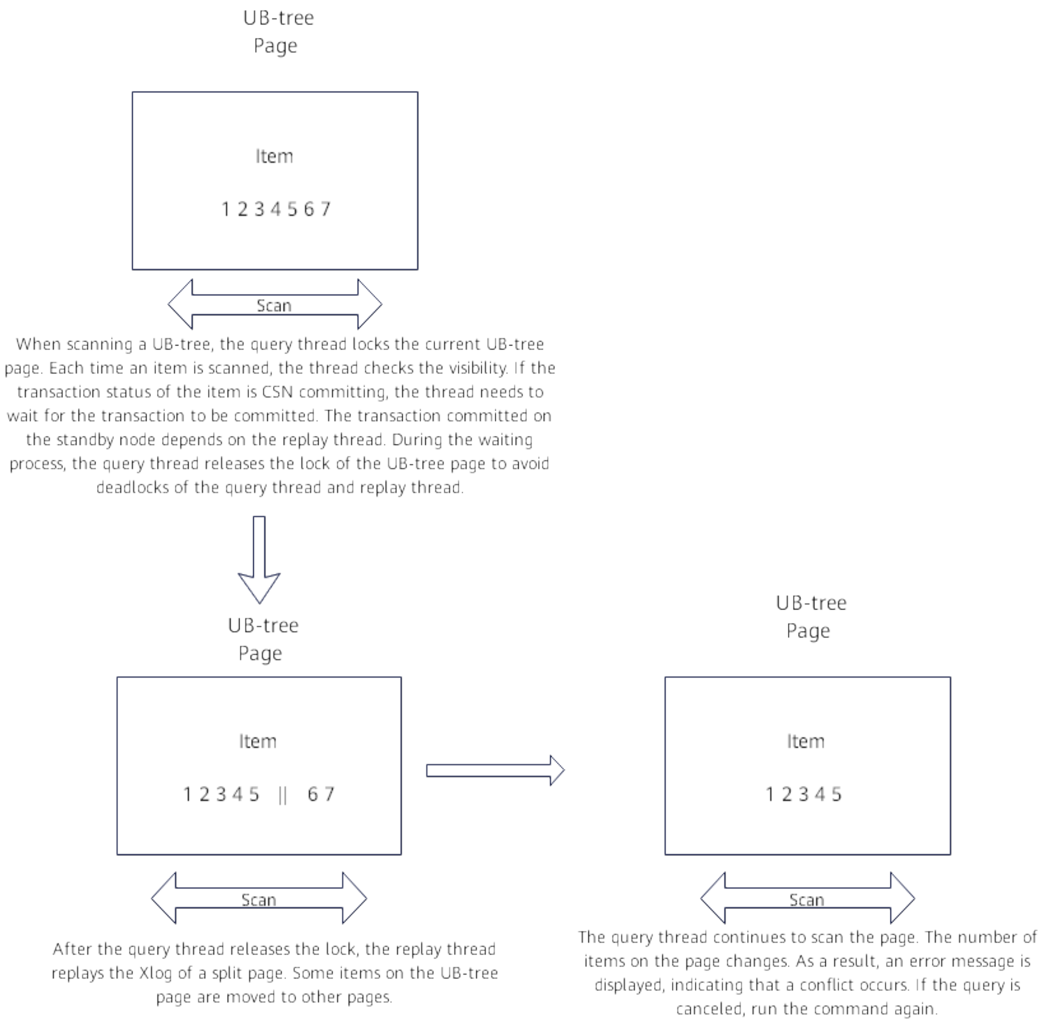
When parallel or serial replay is enabled (if the GUC parameters **recovery_parse_workers** and **recovery_max_workers** are both set to **1**, serial replay is enabled; if **recovery_parse_workers** is set to **1** and **recovery_max_workers** is greater than 1, parallel replay is enabled): If the query 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 4-1** 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.

# 5 FDW

The foreign data wrapper (FDW) of GaussDB can implement cross-database operations between GaussDB databases and remote servers (including databases and file systems). Currently, the supported FDW is file_fdw.

## 5.1 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, 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 when the O&M mode is enabled.

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 of 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

  This 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 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 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**.

## 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** cannot be used for file_fdw.

# 6 Logical Replication

## 6.1 Logical Decoding

### 6.1.1 Overview

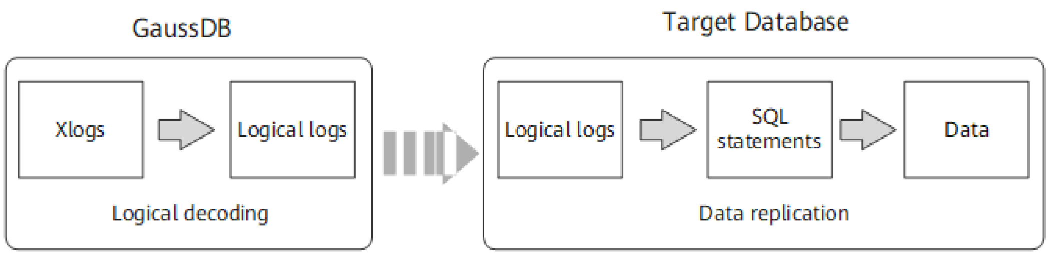
#### Description

In GaussDB:

- Data is periodically synchronized to heterogeneous databases (such as Oracle databases) using a data migration tool. Real-time data replication is not supported. Therefore, the requirements for real-time data synchronization to heterogeneous databases are not satisfied.
- For details about data synchronization for dual-cluster GaussDB DR, see "Server Tools > SyncDataToStby.py" in *Tool Reference*. The standby cluster requires that the numbers of CNs and DNs and the instance deployment mode be consistent with those in the primary cluster. When the standby cluster is restored, read and write operations cannot be performed, and replication latency is relatively high.

Based on the above two points, 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 6-1**. 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 6-1** 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.

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 means a stream of changes that can be replayed in other clusters in the order they were generated in the original cluster. Each owner of logical logs maintains one logical replication slot. 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

- Logical logs are extracted from DNs. If logical replication is required, ensure that the GUC parameter **ssl** on DNs 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 streaming replication slots, backup slots, and logical replication slots required by each DN.

  Physical streaming replication slots provide an automatic method to ensure that Xlogs are not removed from a primary DN before they are received by all the standby DNs. That is, physical replication slots are used to support HA clusters. The number of physical replication slots required by a cluster is equal to the ratio of standby to the primary DN in a ring of DNs. For example, if the cluster has one primary DN and three standby DNs, three physical replication slots are required.

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.

- A user needs to connect to a database through a DN port before using SQL functions to perform logical decoding. If a CN port is used to connect to the database, EXECUTE DIRECT ON (*datanode_name*) *'statement'* is needed to execute SQL functions.
- Only initial users 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

- DDL statement decoding is not supported. When a specific DDL statement (for example, to truncate an ordinary table or exchange a partitioned table) is executed, decoded data may be lost.
- Decoding is not supported for data page replication.
- 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.
- Online cluster scale-out is not allowed during logical decoding.
- 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.
- Decoding compressed tables into DML statements is not supported.
- 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).
- If the SSL connection is required, ensure that the GUC parameter **ssl** is set to **on**.
- The logical replication slot name must contain fewer than 64 characters and contain only one or more types of the following characters: lowercase letters, digits, and underscores (_).
- After the database where a logical replication slot resides is deleted, the replication slot becomes unavailable and needs to be manually deleted.
- Interval partitioned tables cannot be replicated.

- 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.
- After a DDL statement is executed in a transaction, the DDL statement and subsequent statements are not decoded.
- During decoding on the standby node, the decoded data may increase during switchover and 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 hosts.
- 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 use the SQL function pg_terminate_backend (ID of the thread that creates the replication slot) to manually stop the creation.
- 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**. Otherwise, modified rows are not identified in the decoding result of the UPDATE and DELETE statements. For details, see the **REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }** field in "SQL Reference > SQL Syntax > ALTER TABLE" in *Developer Guide*.
- 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.
- 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 replication table exists before scale-in or upgrade, you need to manually set the **logical_repl_node** attribute or reset to the default value for the

replication table. For details, see the usage of the **storage_parameter** parameter and the **logical_repl_node** attribute in "SQL Reference > SQL Syntax > ALTER TABLE" in *Developer Guide*.

- If a transaction has too many sub-transactions, too many files are flushed to disks. To exit decoding, you need to run the SQL function pg_terminate_backend (walsender 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 be reconnected if decoding is disconnected.
- 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. During the observation period before the upgrade is committed, the extended IP address of the DN is used to connect to the logical replication slot created on the DN. Before the upgrade rollback, manually clear the logical replication slot. Otherwise, the DN cannot be directly connected to clear the logical replication slot when the extended IP address feature of the DN is rolled back.
- Logical decoding with strong consistency in a distributed system (with CNs connected) supports only GTM-lite distributed deployment and streaming decoding. It does not support CNs connecting to standby DNs for decoding, SQL logical decoding functions, online scale-out, or global indexes.

- For logical decoding with strong consistency in a distributed system (with CNs connected), the CN HA is switched by the service.
- The **xmin**, **catalog_xmin**, **restart_lsn**, **confirmed_flush**, and **confirmed_csn** columns of the CSN-based logical replication slots on CNs are not displayed because the CSN-based logical replication slots function only as placeholders and do not move with logical decoding or block log recycling.
- If a protocol is used to connect to a CN to create a logical replication slot, only CSN-based replication slots are supported. If a protocol is used to connect to a DN to create a logical replication slot, only LSN-based replication slots are supported.
- For distributed decoding, if an error is reported or the decoding client is manually stopped, wait for 15 seconds and try decoding again. If a replication slot is occupied, run the command **pg_terminate_backend(***ID of the thread that occupies the replication slot***)** to manually release the replication slot.
- If an error is reported when a replication slot fails to be created on a CN, delete the replication slot on the CN and create a replication slot on the CN again.
- When a logical replication slot is deleted from a CN, if the logical replication slot is an LSN-based logical replication slot, only the replication slot of the current node is deleted. Logical replication slots with the same name on other nodes are not affected. When a CSN-based logical replication slot with the same name exists on other nodes, no error is reported because some nodes do not have replication slots. In addition, replication slots with the same name on all nodes are successfully deleted. If no replication slot exists on any node, an error is reported.
- When a CSN-based logical replication slot is created on a CN, if there are residual LSN-based logical replication slots with the same name on some nodes, you need to delete the residual replication slots on these nodes. Otherwise, CSN-based logical replication slots will be created on CNs and primary DNs that do not have replication slots with the same name except the current CN.
- If an LSN-based logical replication slot remains on the current CN and a CSN-based logical replication slot with the same name remains on other nodes, deleting the replication slot on the current CN will delete only the local LSN-based logical replication slot. After the deletion is complete, perform the deletion operation again to delete the replication slots with the same name on other nodes.
- 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.

## 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.

## 6.1.2 Logical Decoding Options

- General options:
  - **include-xids**:

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

    Valid value: **0** and **1**. The default value is **1**.

    - **0**: The decoded **data** column does not contain XID information.
    - **1**: The decoded **data** column contains XID information.
  - **skip-empty-xacts**:

    Specifies whether to ignore empty transaction information during decoding.

    Valid value: **0** and **1**. The default value is **0**.

    - **0**: The empty transaction information is not ignored during decoding.
    - **1**: The empty transaction information is ignored during decoding.
  - **include-timestamp**:

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

    Valid value: **0** and **1**. The default value is **0**.

    - **0**: The decoded information does not contain the **commit** timestamp.
    - **1**: The decoded information contains the **commit** timestamp.
  - **only-local**:

    Specifies whether to decode only local logs.

    Valid value: **0** and **1**. The default value is **1**.

    - **0**: Non-local logs and local logs are decoded.
    - **1**: Only local logs are decoded.
  - **force-binary**:

    Specifies whether to output the decoding result in binary format.

    Value range: **0**

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

– **white-table-list**:

  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.

  Valid value: **0** and **1**. The default value is **0**.

  - **0**: The BEGIN logical log of a transaction does not contain the username of the transaction.
  - **1**: The BEGIN logical log of a transaction records the username of the transaction.

– **exclude-userids**:

  Specifies the OID of a blacklisted user. It can be used only when SQL functions are used for decoding and cannot be specified when a logical decoding task is started.

  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.

  Valid value: **0** and **1**. The default value is **1**.

  - **0**: If the parameter is set to **0**, an error is reported and the logical decoding exits when the decoding detects that the user does not exist in blacklist **exclude-users**.
  - **1**: If the parameter is set to **1**, decoding continues when it detects that the user does not exist in blacklist **exclude-users**.

– **standby-connection**:

  Specifies whether to restrict decoding only on the standby node. **This option is set only for streaming decoding.** It can be used only when SQL functions are used for decoding and cannot be specified when a logical decoding task is started.

  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**.

– **enable-heartbeat**:

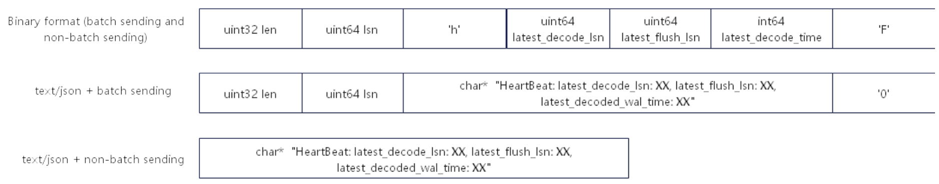
  Specifies whether to generate heartbeat logs. **This option is set only for streaming decoding.**

  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 heartbeat logs can be parsed as follows: For a binary heartbeat log message, it starts with a character 'h' and then the heartbeat log content: an 8-byte uint64 string, an 8-byte uint64 string, and an 8-byte int64 string. For the first 8-byte uint64 string, in the decoding scenario where DNs are directly connected, this string is an LSN, indicating the end position of the WAL read when the heartbeat logical log is sent; in the decoding scenario where distributed strong consistency is required, this string is a CSN, indicating the decoding log transaction CSN that has been sent when the heartbeat logical log is sent. For the second 8-byte uint64 string, in the decoding scenario where DNs are directly connected, this string is an LSN, indicating the location of the WAL that has been flushed to disks when the heartbeat logical log is sent; in the decoding scenario where distributed strong consistency is required, this string is a CSN, indicating the CSN to be obtained by the next transaction committed by the cluster. The last 8-byte int64 string indicates the generation timestamp (starting from January 1, 1970) of the latest decoded transaction log or checkpoint log. 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. The message content is transmitted in big-endian mode. The following figure shows the format. (In consideration of forward compatibility, the LSN naming mode is retained. The actual meaning depends on the specific scenario.)


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**:

  Number of decoder threads for parallel decoding. **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.

  Value range: 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" 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.

NOTICE

- When **output-order** is not configured (that is, the default value **0** is used and the order is based on the COMMIT LSN) or is explicitly configured to **0**, the options in the following "CSN decoding" cannot be configured.
- In streaming decoding scenarios, when a DN receives a logical decoding connection from a CN, the **output-order** option is invalid and CSN decoding is performed by default.

- **auto-advance**:

  Specifies whether to automatically advance logical replication slots. **This parameter is set only for streaming decoding.**

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

  - **true**: The logical replication slot is advanced to the current decoding position when all sent logs are confirmed and there is no transaction to be sent.
  - **false**: The replication service invokes the log confirmation interface 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. Generated columns are not supported in a distributed system and therefore, this parameter has no actual impact. It can be used only when SQL functions are used for decoding and cannot be specified when a logical decoding task is started.

  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.

- CSN decoding:
  - **logical-receiver-num**:

    Number of logical receivers started for distributed decoding. **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.

    Value range: an integer ranging from 1 to 20. The default value is **1**. If this option is set to a value greater than the number of shards in the current cluster, the value is changed to the number of shards.
  - **slice-id**:

ID of the shard where the current DN is located. This option is set only when DNs are connected for decoding. It is used to decode replication tables.

Value range: an integer ranging from 0 to 8192. The default value is **-1**, indicating that the shard ID is not specified. However, an error is reported when the data is decoded to the replication table.

---

⚠ CAUTION

This configuration option is used when the DN attempts to use the CSN logical replication slot (**confirmed_csn** is a non-zero replication slot) for decoding. It is used to indicate the shard ID (that is, the sequence number of the shard. Enter **0** for the first shard). If this option is not set (that is, the default value **-1** is used), an error is reported when data is decoded to the replication table. This option is used when the CN collects decoding results from DNs in distributed decoding mode. You are advised not to manually connect to DNs for decoding in this scenario.

---

– **start-position**:

Filters out transactions whose CSNs are less than the specified CSN, and filters out logs whose LSNs are less than the specified LSN for the transaction with specified CSN. **This option is set only when DNs are connected. BEGIN logs of the transaction with specified CSN must be filtered out.**

Value: a string of two uint64 characters separated by a slash (/). The left and right sides indicate the CSN and LSN, respectively.

---

⚠ CAUTION

This option is used to filter logs that may have been received when the CN sends a decoding request after establishing a connection to the DN during CN decoding. You are advised not to manually connect to DNs for decoding in this scenario.

---

- 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. If **B** is used in the step 3:
   1. The next eight bytes (uint64) indicate the CSN.
   2. The next eight bytes (uint64) indicate **first_lsn**.
   3. (Optional) If the next 1-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 1-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 the step **c**:
   1. (Optional) If the next 1-byte letter is **X**, the following eight bytes (uint64) indicate XID.
   2. (Optional) If the next 1-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 1-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 the step **c**:
   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 1-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 subsequently, 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 decoding scenario, batch sending is limited to a single transaction (that is, if a transaction has multiple small statements, batch sending is used). The batch sending function is not used to send multiple transactions in the same batch.

– **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.

## 6.1.3 Logical Decoding by SQL Function Interfaces

In GaussDB, you can call SQL functions to create, delete, and update logical replication slots, as well as obtain decoded transaction logs.

### Procedure

**Step 1** Log in to any host in the GaussDB cluster as a user with the REPLICATION permission.

**Step 2** Connect to the database through a CN port.

```
gsql -U user1 -d gaussdb -p 40000 -r
```

In the preceding command, **user1** indicates the username, **gaussdb** indicates the name of the database to be connected, and **40000** indicates the database DN port number. You can replace them as required. Replication slots are created on DNs. Therefore, you need to connect to a database through a DN port.

**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** on all DNs. The number of decoded records is 4096.

**NOTE**

For details about the logical decoding options, see **Logical Decoding Options**.

```
gaussdb=# EXECUTE DIRECT ON DATANODES '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** Delete the logical replication slot **slot1**.

```
gaussdb=# SELECT * FROM pg_drop_replication_slot('slot1');
 pg_drop_replication_slot
--------------------------

(1 row)
```

**----End**

## 6.1.4 Logical Data Replication Using Stream 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*.