GaussDB

Feature Guide(Distributed_V2.0-8.x)

Issue 01

Date 2025-12-09

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Huawei Cloud Computing Technologies Co., Ltd.

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

Contents

1 Materialized View...... 1
1.1 Complete-Refresh Materialized View...... 1
1.1.1 Overview...... 1
1.1.2 Support and Constraints...... 2
1.1.3 Usage...... 2
1.2 Fast-Refresh Materialized View...... 3
1.2.1 Overview...... 3
1.2.2 Support and Constraints...... 3
1.2.3 Usage...... 4
2 Setting Encrypted Equality Queries...... 7
2.1 Overview...... 7
2.2 Using gsql to Operate an Encrypted Database...... 11
2.3 Using JDBC to Operate an Encrypted Database...... 13
2.4 Forward Compatibility and Security Enhancement...... 19
2.5 Encrypted Functions and Stored Procedures...... 22
3 Setting a Ledger Database...... 25
3.1 Overview...... 25
3.2 Viewing Historical Operation Records in the Ledger...... 27
3.3 Checking Ledger Data Consistency...... 29
3.4 Archiving a Ledger Database...... 30
3.5 Repairing a Ledger Database...... 32
4 Logical Replication...... 35
4.1 Logical Decoding...... 35
4.1.1 Overview...... 35
4.1.2 Logical Decoding Options...... 44
4.1.3 Logical Decoding by SQL Functions...... 54
4.1.4 Logical Data Replication Using Streaming Decoding...... 55
4.1.5 Logical Decoding of DDL Statements...... 55
4.1.6 Data Restoration by Logical Decoding...... 64
5 Partitioned Table...... 67
5.1 Introduction to Table Partitioning...... 67
5.1.1 Background...... 67

5.1.2 Table Partitioning 67
5.1.3 Optimization Data Query by Partition 68
5.1.4 Data Partition O&M Management 69
5.2 Introduction to Partitioned Tables 69
5.2.1 Basic Concepts 70
5.2.1.1 Partitioned Table 70
5.2.1.2 Partition 70
5.2.1.3 Partition Key 71
5.2.2 Partitioning Policy 72
5.2.2.1 Range Partitioning 72
5.2.2.2 Hash Partitioning 74
5.2.2.3 List Partitioning 75
5.2.2.4 Impact of Partitioned Tables on Import Performance 76
5.2.3 Basic Usage of Partitions 77
5.2.3.1 Creating Partitioned Tables 78
5.2.3.2 Using and Managing Partitioned Tables 79
5.2.3.3 DQL/DML Operations on a Partitioned Table 80
5.3 Partitioned Table Query Optimization 82
5.3.1 Partition Pruning 82
5.3.1.1 Static Partition Pruning 82
5.3.1.2 Dynamic Partition Pruning 87
5.3.1.2.1 Dynamic PBE Pruning 87
5.3.1.2.2 Dynamic Parameterized Path Pruning 90
5.3.2 Partitioned Indexes 94
5.3.3 Collecting Statistics on Partitioned Tables 97
5.3.3.1 Collecting Statistics in Cascading Mode 98
5.3.3.2 Collecting Partition-Level Statistics 100
5.3.4 Partition-Wise Join 105
5.3.4.1 Partition-Wise Join in Non-SMP Scenarios 105
5.3.4.2 Full Partition-Wise Join in the SMP Scenario 106
5.4 Partitioned Table O&M Management 109
5.4.1 ADD PARTITION 109
5.4.1.1 Adding a Partition to a Range Partitioned Table 110
5.4.1.2 Adding a Partition to a List Partitioned Table 110
5.4.2 DROP PARTITION 110
5.4.3 EXCHANGE PARTITION 111
5.4.4 TRUNCATE PARTITION 113
5.4.5 SPLIT PARTITION 114
5.4.5.1 Splitting a Partition for a Range Partitioned Table 114
5.4.5.2 Splitting a Partition for a List Partitioned Table 115
5.4.6 MERGE PARTITION 115
5.4.7 MOVE PARTITION 116

5.4.8 RENAME PARTITION 116
5.4.8.1 Renaming a Partition in a Partitioned Table 116
5.4.8.2 Renaming an Index Partition for a Local Index 117
5.4.9 ALTER TABLE ENABLE/DISABLE ROW MOVEMENT 117
5.4.10 Invalidating/Rebuilding Indexes of a Partition 118
5.4.10.1 Invalidating/Rebuilding Indexes 118
5.4.10.2 Invalidating/Rebuilding Local Indexes of a Partition 118
5.5 Partition Concurrency Control 118
5.5.1 Common Lock Design 119
5.5.2 DQL/DML-DQL/DML Concurrency 120
5.5.3 DQL/DML-DDL Concurrency 120
5.6 System Views & DFX Related to Partitioned Tables 124
5.6.1 System Views Related to Partitioned Tables 124
5.6.2 Built-in Tool Functions Related to Partitioned Tables 125
6 Storage Engine 128
6.1 Storage Engine Architecture 128
6.1.1 Overview 128
6.1.1.1 Static Compilation Architecture 128
6.1.1.2 Database Service Layer 129
6.1.2 Setting Up a Storage Engine 130
6.1.3 Storage Engine Update Description 131
6.1.3.1 GaussDB Kernel 505 131
6.1.3.2 GaussDB Kernel 503 131
6.1.3.3 GaussDB Kernel R2 132
6.2 Astore 132
6.2.1 Overview 132
6.3 Ustore 133
6.3.1 Overview 133
6.3.1.1 Ustore Features and Specifications 133
6.3.1.1.1 Restrictions 133
6.3.1.1.2 Storage Specifications 134
6.3.1.2 Examples 134
6.3.1.3 Best Practices of Ustore 135
6.3.1.3.1 How Can I Configure init_td 135
6.3.1.3.2 How Can I Configure fillfactor 137
6.3.1.3.3 Online Verification 137
6.3.1.3.4 How Can I Configure the Size of Rollback Segments 138
6.3.2 Storage Format 139
6.3.2.1 RCR Uheap 139
6.3.2.1.1 RCR Uheap Multi-Version Management 139
6.3.2.1.2 RCR Uheap Visibility Mechanism 140
6.3.2.1.3 RCR Uheap Free Space Management 140

6.3.2.2 UB-Tree 141
6.3.2.2.1 RCR UB-Tree 141
6.3.2.2.2 PCR UB-Tree 144
6.3.2.3 Undo 146
6.3.2.3.1 Rollback Segment Management 146
6.3.2.3.2 File Structure 146
6.3.2.3.3 Space Management 147
6.3.2.4 Enhanced TOAST 147
6.3.2.4.1 Overview 147
6.3.2.4.2 Enhanced TOAST Storage Structure 147
6.3.2.4.3 Usage of Enhanced TOAST 148
6.3.2.4.4 Adding, Deleting, Modifying, and Querying Enhanced TOAST 148
6.3.2.4.5 DDL Operations Related to Enhanced TOAST 148
6.3.2.4.6 Enhanced TOAST O&M Management 151
6.3.3 Ustore Transaction Model 151
6.3.3.1 Transaction Commit 152
6.3.3.2 Transaction Rollback 152
6.3.4 Flashback 153
6.3.4.1 Flashback Query 153
6.3.4.2 Flashback Table 156
6.3.4.3 Flashback DROP/TRUNCATE 158
6.3.5 Common View Tools 166
6.3.6 Common Problems and Troubleshooting Methods 170
6.3.6.1 Snapshot Too Old 170
6.3.6.1.1 Undo Space Recycling Blocked by Long Transactions 170
6.3.6.1.2 Slow Undo Space Recycling Caused by Many Rollback Transactions 171
6.3.6.2 Storage Test Error 171
6.3.6.3 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 172
6.3.6.4 Write Performance Deteriorates Occasionally When a Large Number of Concurrent Updates Are Performed During Long Query Execution 174
6.4 Data Lifecycle Management: OLTP Table Compression 174
6.4.1 Introduction 174
6.4.2 Restrictions 175
6.4.3 Feature Specifications 175
6.4.4 Usage Guide 176
6.4.5 Setting the Maintenance Window Parameters 181
6.4.6 O&M Command Reference 183
7 Foreign Data Wrapper 191
7.1 file_fdw 191
8 Dynamic Data Masking 193

1 Materialized View

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

Materialized views are classified into complete-refresh materialized view views and fast-refresh materialized views based on creation rules. Complete-refresh materialized views can only be completely refreshed. Fast-refresh materialized views can be completely or fast refreshed. During complete refresh, all data in the base table is refreshed to the materialized view. During fast refresh, only the incremental data generated in the base table during the interval between two refreshes is refreshed to the materialized view.

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

1.1 Complete-Refresh Materialized View

1.1.1 Overview

Complete-refresh materialized view is a materialized view that supports only complete refresh. That is, old data is discarded and the entire table is recalculated for query.

The syntax for creating a complete-refresh materialized view is the same as CREATE TABLE AS. For details, see "SQL Reference > SQL Syntax > CREATE TABLE AS" in *Developer Guide*. You cannot specify a node group for a complete-refresh materialized view. Creating a complete-refresh materialized view inherits the constraints of GTM-Free.

1.1.2 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 key 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.
- Ustore does not support the creation and use of complete-refresh materialized views.

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.

1.1.3 Usage

Syntax

- Create a complete-refresh materialized view.
  ```
  CREATE MATERIALIZED VIEW view_name AS query;
  ```
- 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;
  ```

Parameters

- **view_name**

  Specifies the name of the materialized view to be created.

  Value range: a string. It must comply with the identifier naming convention.
- **AS query**

  Specifies a **SELECT** or **VALUES** command, or an **EXECUTE** command that runs a prepared **SELECT** or **VALUES** query.

Examples

```
-- Change the default type of a table.
gaussdb=#  set enable_default_ustore_table=off;

-- 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 the materialized view and table.
gaussdb=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
gaussdb=# DROP TABLE t1;
DROP TABLE
```

1.2 Fast-Refresh Materialized View

1.2.1 Overview

Fast-refresh materialized view: Views can be incrementally refreshed. You need to manually execute statements to incrementally refresh materialized views in a period of time.

The difference between the fast-refresh and complete-refresh materialized views is that the fast-refresh materialized views support only a small number of scenarios. Currently, only base table scanning statements or UNION ALL can be used to create materialized views.

1.2.2 Support and Constraints

Supported Scenarios

- Supports statements for querying a single table.
- Supports UNION ALL for querying multiple single tables.

- Creates an index in a materialized view.
- Performs the ANALYZE operation in a 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-refresh materialized view based on the node group).

Unsupported Scenarios

- Materialized views do not support the Stream plan, multi-table join plan, or subquery plan.
- Clauses WITH, GROUP BY, ORDER BY, LIMIT, and WINDOW are not supported. Operators DISTINCT and AGG are not supported. Subqueries except UNION ALL are not supported.
- Except for a few ALTER operations, most DDL operations cannot be performed on base tables in materialized views.
- 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 distributed 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.
- Ustore does not support the creation and use of fast-refresh materialized views.

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.

1.2.3 Usage

Syntax

- Create a fast-refresh materialized view.
```
CREATE INCREMENTAL MATERIALIZED VIEW view_name AS query;
```
- 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;
```

Parameters

- **view_name**

  Specifies the name of the materialized view to be created.

  Value range: a string. It must comply with the identifier naming convention.
- **AS query**

  Specifies a **SELECT** or **VALUES** command, or an **EXECUTE** command that runs a prepared **SELECT** or **VALUES** query.

Examples

```
-- Change the default type of a table.
gaussdb=#  SET enable_default_ustore_table=off;

-- 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 the materialized view and table.
```

```
gaussdb=# DROP MATERIALIZED VIEW mv;
DROP MATERIALIZED VIEW
gaussdb=# DROP TABLE t1;
DROP TABLE
```

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.

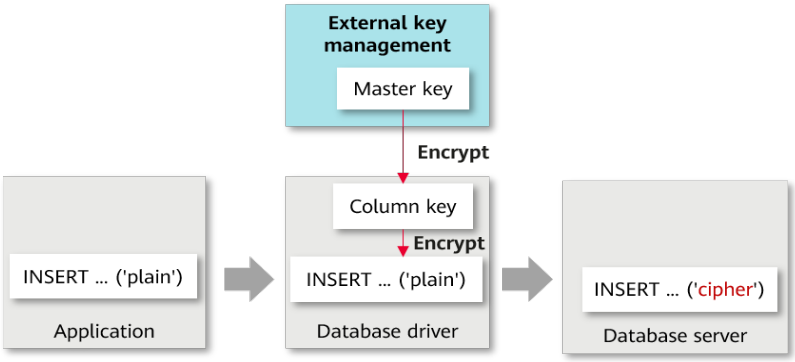
Encryption Model

A fully-encrypted database uses a multi-level encryption model. The encryption model involves three objects: data, column key, and master key, which are described as follows:

- **Data**, including:
  a. Data contained in the SQL syntax. For example, the INSERT... VALUES ('data') syntax contains 'data'.
  b. Query result returned from the database server, for example, the query result returned after the SELECT syntax is executed.

  NOTE

  An encrypted database encrypts data of encrypted columns in the SQL syntax in the driver and decrypts the query result of the encrypted columns returned from the database server.
- **Column key**: Data is encrypted by using column keys. The column keys are generated by the database driver or manually imported by users. The column key ciphertext is stored on the database server.
- **Master key**: Column keys are encrypted by using master keys. The master keys are generated and stored by an external key manager. The database driver automatically accesses the external key manager to encrypt and decrypt column keys.



Overall Process

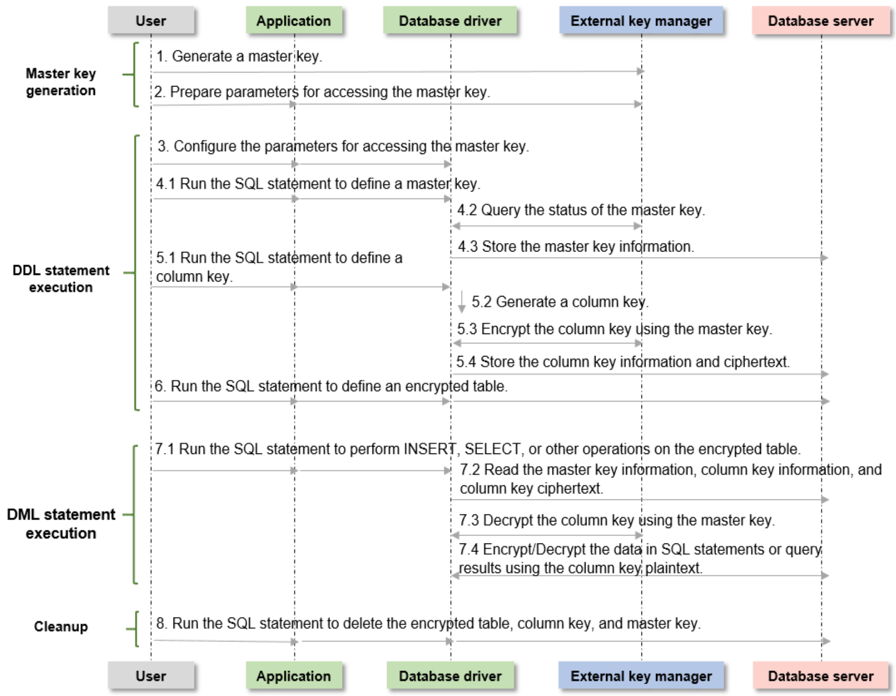
The process of using a fully-encrypted database consists of the following four phases. This section describes the overall process. Sections **Using gsql to Operate an Encrypted Database** and **Using JDBC to Operate an Encrypted Database** describe the detailed usage process.

1. Master key generation phase: First, you need to generate a master key in Huawei Cloud key management service. After the master key is generated, you need to prepare the parameters for accessing the master key.

2. DDL statement execution phase: In this phase, you can use the key syntax of the encrypted database to define the master key and column key in sequence, define the table, and specify a column in the table as an encrypted column. When defining the master key and column key, you need to access the master key generated in the previous phase.

3. 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.

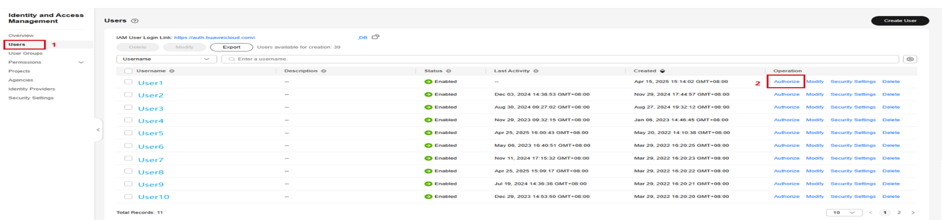
4. Cleanup phase: Delete the encrypted table, column key, and master key in sequence.

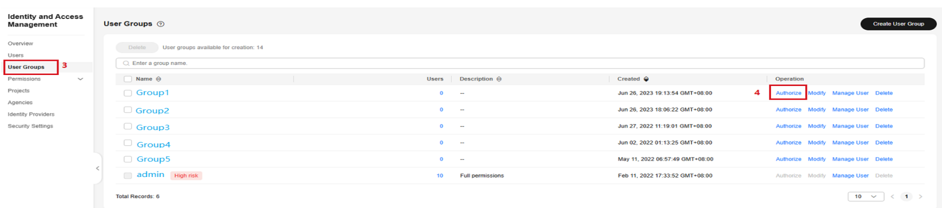


Master Key Generation Phase

When you use an encrypted database for the first time, you need to use an external key manager to generate at least one master key. The operation procedure is as follows:

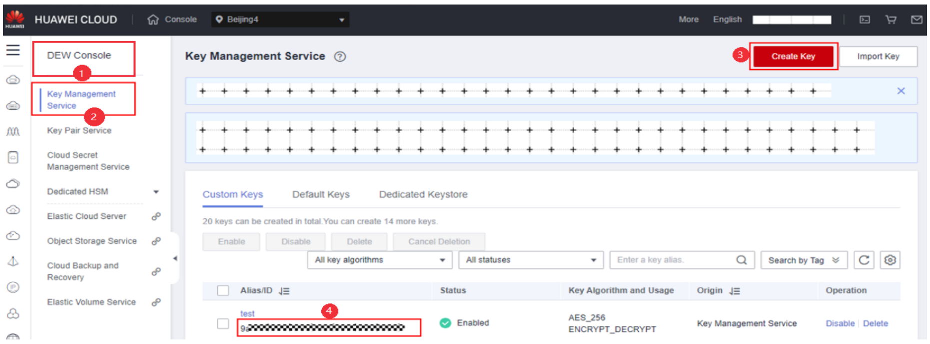
- Huawei Cloud scenario
  a. Access the Huawei Cloud website, register an account, and log in.
  b. Search for and access the IAM service. On the **Users** page, click **Create User** to create an IAM user, set a password for the user, associate the user with a user group, and grant the DEW permission to the user group.





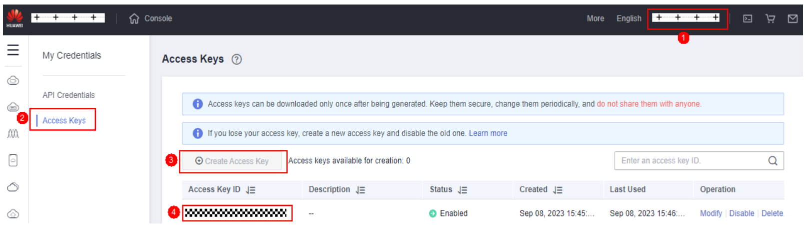
  c. Return to the login page, and select to log in as the IAM user created in the previous step to log in. The subsequent operations are all performed by the IAM user.
  d. Choose **Key Management Service** and click **Create Key** to create at least one key, that is, the master key.

e. Remember the master key ID. Each master key has a key ID. When using encrypted data, you need to configure the master key ID. The database driver accesses the master key through the RESTful API.

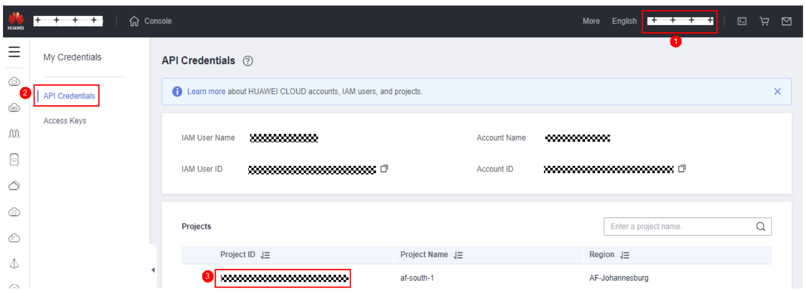


After the master key is generated, you need to prepare parameters for the data driver to access the master key, such as the IAM username and project ID. Huawei Cloud supports two identity authentication modes. The number of parameters and parameter types required by the two authentication modes are different. You can select either of them. To obtain these parameters, perform the following steps:

- **Method 1**: AK/SK authentication
  a. AK and SK: Log in to the Huawei Cloud console, click the username in the upper right corner, and choose **My Credentials**. On the displayed page, click **Access Keys**. Click **Create Access Key** to create an AK/SK pair. After the creation is successful, you can download the AK and SK.



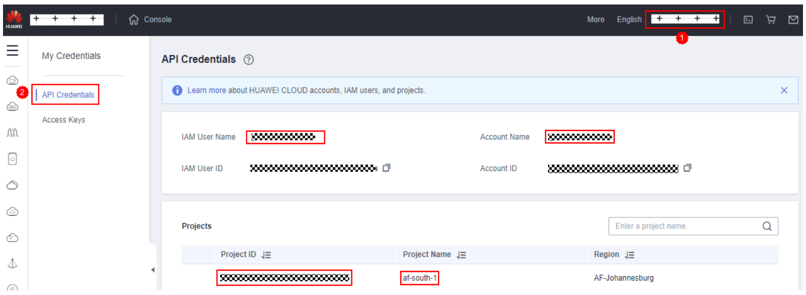
  b. Project ID: On the Huawei Cloud console, click the username in the upper right corner, and choose **My Credentials**. On the displayed page, click **API Credentials** to find the project ID.



  c. KMS server address: https://kms.*Project*.myhuaweicloud.com/v1.0/*Project ID*/kms

- **Method 2**: Account and password authentication
  a. IAM username, account name, project, and project ID: On the Huawei Cloud console, click the username in the upper right corner and choose **My Credentials**. On the displayed page, you can obtain the IAM

username, account name, project, and project ID, as shown in the following figure.



b. IAM server address: https://iam.*Project*.myhuaweicloud.com/v3/auth/tokens

c. IAM user password: Enter the password of the IAM user.

d. KMS server address: https://kms.*Project*.myhuaweicloud.com/v1.0/*Project ID*/kms

2.2 Using gsql to Operate an Encrypted Database

Executing SQL Statements

Before running the SQL statements in this section, ensure that the master key has been generated and the parameters for accessing the master key are clear.

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.

Step 1 Connect to the database and use the **-C** parameter to enable the full encryption function.

```
gsql -p PORT -d DATABASE -h HOST -U USER -W PASSWORD -r -C
```

Step 2 Set parameters for accessing the master key using a meta-command.

Note: There must be no line feed or space in the string starting from **keyType**. Otherwise, the gsql tool cannot identify the entire parameter.

Huawei Cloud supports two authentication modes. The number of parameters and parameter types required by the two authentication modes are different. You can select either of them.

- **Authentication mode 1**: AK/SK authentication

```
gaussdb=# \key_info keyType=huawei_kms,kmsProjectId={Project ID},ak={AK},sk={SK}
```

  Parameters: For details about how to obtain parameters, including the project ID, AK, and SK, see the master key generation phase.

  Example: \key_info keyType=huawei_kms,kmsProjectId=0b59929e8100268a2f22c01429802728,ak=XMAUMJY******DFWLQW,sk=ga6rO8lx1Q4uB*********2gf80muIzUX

- **Authentication mode 2**: Account and password authentication

```
gaussdb=# \key_info keyType=huawei_kms,iamUrl={IAM server address},iamUser={IAM username},iamPassword={IAM user password},iamDomain={Account name},kmsProject={Project}
```

Parameters: For details about how to obtain related parameters, including the IAM server address, IAM username, IAM user password, account name, and project, see the master key generation phase.

Example: \key_info keyType=huawei_kms,iamUrl=https://iam.*example*.com/v3/auth/tokens,iamUser=test,iamPassword=*********,iamDomain=test_account,kmsProject=xxx

Step 3 Define a master key.

In the master key generation phase, the KMS has generated and stored the master key. Running this syntax only stores the master key information in the database for future access. For details about the syntax format, see "SQL Reference > SQL Syntax > CREATE CLIENT MASTER KEY" in *Developer Guide*.

```
gaussdb=# CREATE CLIENT MASTER KEY cmk1 WITH (KEY_STORE = huawei_kms, KEY_PATH = '{KMS server address}/{Key ID}', ALGORITHM = AES_256);
CREATE CLIENT MASTER KEY
```

- Parameters: For details about how to obtain related parameters, including KMS server address and key ID, see the master key generation phase.

  Example of *KEY_PATH*: https://kms.cn-north-4.myhuaweicloud.com/v1.0/0b59929e8100268a2f22c01429802728/kms/9a262917-8b31-41af-a1e0-a53235f32de9

Step 4 Define a column key.

The column key is encrypted by the master key defined 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);
```

Step 5 Define an encrypted table.

In this example, the **name** and **credit_card** columns in the table are specified as encrypted columns by using syntax.

```
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 column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
```

Step 6 Perform other operations on the encrypted table.

```
-- 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

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

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

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

-- Delete an encrypted column from the table.
gaussdb=# ALTER TABLE creditcard_info DROP COLUMN age;
ALTER TABLE

-- 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 row)

-- 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 row)

-- View meta information of a column in the table.
gaussdb=# \d creditcard_info
       Table "public.creditcard_info"
   Column    |       Type        | Modifiers
-------------+-------------------+------------
 id_number   | integer           |
 name        | text              |  encrypted
 credit_card | character varying |  encrypted
```

Step 7 Enter the cleanup phase.

```
-- Delete the encrypted table.
gaussdb=# DROP TABLE creditcard_info;
DROP TABLE

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

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

----End

2.3 Using JDBC to Operate an Encrypted Database

Configuring the JDBC Driver

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" and "Application Development Guide > Compatibility Reference > JDBC Compatibility Package" in *Developer Guide*.

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

   – **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. The Java code examples in this section use the **gscejdbc.jar** package by default.

- **gaussdbjdbc.jar**: The main class name is **com.huawei.gaussdb.jdbc.Driver**. The URL prefix of the database connection is **jdbc:gaussdb**. This driver package does not contain the dependent libraries related to encryption and decryption that need to be loaded to an encrypted database. You need to manually configure the *LD_LIBRARY_PATH* environment variable.
- **gaussdbjdbc-JRE7.jar**: The main class name is **com.huawei.gaussdb.jdbc.Driver**. The URL prefix of the database connection is **jdbc:gaussdb**. The **gaussdbjdbc-JRE7.jar** package is used in the JDK 1.7 environment. This driver package does not contain the dependent libraries related to encryption and decryption that need to be loaded to an encrypted database. You need to manually configure the *LD_LIBRARY_PATH* environment variable.

NOTE

Other compatibility: The encrypted database also supports other compatible JDBC driver packages: **gsjdbc4.jar** and **opengaussjdbc.jar**.

- **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**.

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 connecting to the database with the encrypted database function enabled.
   - When using **gaussdbjdbc.jar**, **gaussdbjdbc-JRE7.jar**, **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 master key has been generated and the parameters for accessing the master key are clear.

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 not set, 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.
// gaussdbjdbc.jar: The main class name is com.huawei.gaussdb.jdbc.Driver, and the URL prefix of the database connection is jdbc:gaussdb.
// gaussdbjdbc-JRE7.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();

       // Set the parameters for accessing the master key.
        // Two methods are provided here. Select either of them.
        // Authentication mode 1: AK/SK authentication (For details about how to obtain parameters, including the project ID, AK, and SK, see the master key generation phase.)
        conn.setClientInfo("key_info", "keyType=huawei_kms, kmsProjectId={Project ID}, ak={AK}, sk={SK}");

        /* Example:
             conn.setClientInfo("key_info",
"keyType=huawei_kms,kmsProjectId=0b59929e8100268a2f22c01429802728," +
                "ak=XMAUMJY******DFWLQW, sk=ga6rO8lx1Q4uB*********2gf80muIzUX,");
        */
        // Authentication mode 2: Account and password authentication (For details about how to obtain related parameters, including the IAM server address, IAM username, IAM user password, account name, and project, see the master key generation phase.)
        conn.setClientInfo("key_info", "keyType=huawei_kms," +
```

```
            "iamUrl={IAM server address}," +
            "iamUser={IAM username}," +
            "iamPassword={IAM user password}," +
            "iamDomain={Account name}," +
            "kmsProject={Project}");
        /* Example:
        conn.setClientInfo("key_info", "keyType=huawei_kms," +
                "iamUrl=https://iam.example.com/v3/auth/tokens," +
                "iamUser=test," +
                "iamPassword=*********," +
                "iamDomain=test_account," +
                "kmsProject=xxx");
        */

        // Define the master key. cmk1 is the master key name, which can be customized.
        // For details about how to obtain the following parameters, including KMS server address and
key ID, see the master key generation phase.
        int rc = stmt.executeUpdate("CREATE CLIENT MASTER KEY ImgCMK1 WITH ( KEY_STORE =
huawei_kms , KEY_PATH = '{KMS server address}/{Key ID}', ALGORITHM = AES_256);");

        /*
             Example of KEY_PATH: https://kms.cn-north-4.myhuaweicloud.com/
v1.0/0b59929e8100268a2f22c01429802728/kms/9a262917-8b31-41af-a1e0-a53235f32de9
             Explanation: In the master key generation phase, the KMS has generated and stored the
master key. Running this syntax only stores the master key information in the database for future
access.
             Note: For details about the KEY_PATH format, see "SQL Reference > SQL Syntax > CREATE
CLIENT MASTER KEY" in Developer Guide.
        */

        // Define a column encryption 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.
        int rc2 = stmt.executeUpdate("CREATE COLUMN ENCRYPTION KEY ImgCEK1 WITH VALUES
(CLIENT_MASTER_KEY = ImgCMK1, ALGORITHM  = AES_256_GCM);");
        // Define 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

- 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.
- 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 com.huawei.gaussdb.jdbc.jdbc.PgConnection function

Method	Return Type	Support JDBC 4 (Yes/No)
decryptData(String ciphertext, Integer len, String schema, String table, String column)	ClientLogicDecryptResult	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 ciphertext length input parameter is longer than the actual ciphertext.
  - **schema.table.column** points to another encrypted column. In this case, the original data type of the encrypted column is returned.

Table 2-2 com.huawei.gaussdb.jdbc.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 com.huawei.gaussdb.jdbc.jdbc.PgConnection;
import com.huawei.gaussdb.jdbc.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()));
}
```

Preparing 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 prepared SQL statement by triggering the executeUpdate method in PreparedStatement.
int rowcount = pstmt.executeUpdate();
// Close the prepared statement object by calling the close method in PreparedStatement.
pstmt.close();
```

Batch Processing on Encrypted Tables

```
// 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 prepared statement object by calling the close method in PreparedStatement.
pstmt.close();
```

2.4 Forward Compatibility and Security Enhancement

Forward Compatibility

In the preceding sections, you can use **key_info** to set parameters for accessing an external key manager.

1. When gsql is used, you can use the meta-command **\key_info** *xxx*.
2. When JDBC is used, you can use the connection parameter **conn.setProperty("key_info","***xxx***")**.

To ensure forward compatibility, you can set parameters for accessing the master key using environment variables.

CAUTION

If you use an encrypted database for the first time, skip the following steps. If you have used any of the following methods to configure the encrypted database, you are advised to use **key_info** instead.

To use system-level environment variables, perform the following steps:

```
export HUAWEI_KMS_INFO='iamUrl=https://iam.{Project}.myhuaweicloud.com/v3/auth/
tokens,iamUser={IAM username},iamPassword={IAM user password},iamDomain={Account
name},kmsProject={Project}'.
# In this method, the OS logs may record sensitive information in environment variables. Delete the
sensitive information in a timely manner.
```

You can also set process-level environment variables using the standard library API. The methods for setting process-level environment variables in different languages are as follows:

1. C/C++
```
setenv("HIS_KMS_INFO", "xxx");
```
2. GO
```
os.Setenv("HIS_KMS_INFO", "xxx");
```

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. This section describes how to download and configure a CA certificate.

Configuration Method

Add certificate-related **key_info** parameters.

- When gsql is used:
```
gaussdb=# \key_info keyType=huawei_kms,iamUrl=https://iam.example.com/v3/auth/tokens,iamUser={IAM username},iamPassword={IAM user password},iamDomain={Account name},kmsProject={Project},iamCaCert=/Path/IAM CA certificate file,kmsCaCert=/Path/KMS CA certificate file

gaussdb=# \key_info keyType=huawei_kms,kmsProjectId={Project ID},ak={AK},sk={SK},kmsCaCert=/Path/KMS CA certificate file
```
- When JDBC is used:
```
conn.setProperty("key_info", "keyType=huawei_kms," +
    "iamUrl=https://iam.{example.com/v3/auth/tokens," +
    "iamUser={IAM username}," +
    "iamPassword={IAM user password}," +
    "iamDomain={Account name}," +
"kmsProject={Project}," +
    "iamCaCert=/Path/IAM CA certificate file," +
    "kmsCaCert=/Path/KMS CA certificate file");

conn.setProperty("key_info", "keyType=huawei_kms, kmsProjectId={Project ID}, ak={AK}, sk={SK}, kmsCaCert=/Path/KMS CA certificate file");
```

Obtaining a Certificate

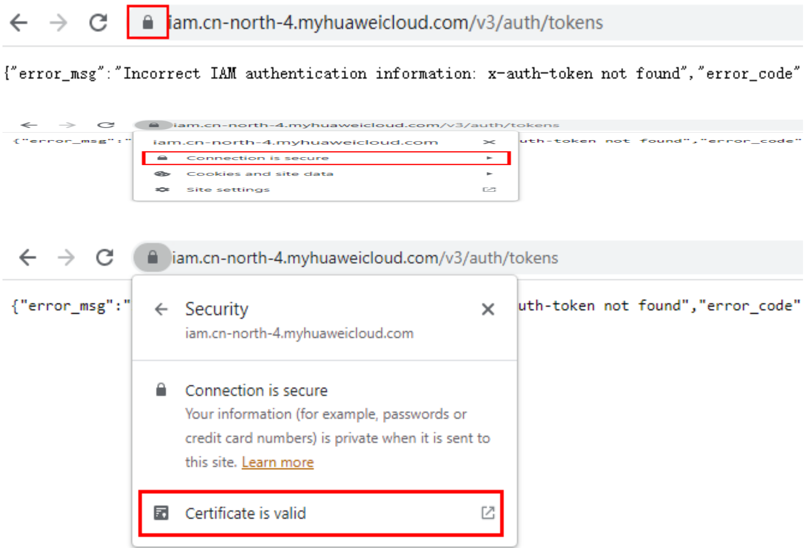
Most browsers automatically download a CA certificate of a website and provide the certificate export function. Although many websites provide the function of automatically downloading CA certificates, these certificates may be unavailable due to proxy or gateway settings in the local environment. Therefore, you are advised to use a browser to download the CA certificate. You can perform the following steps:

CAUTION

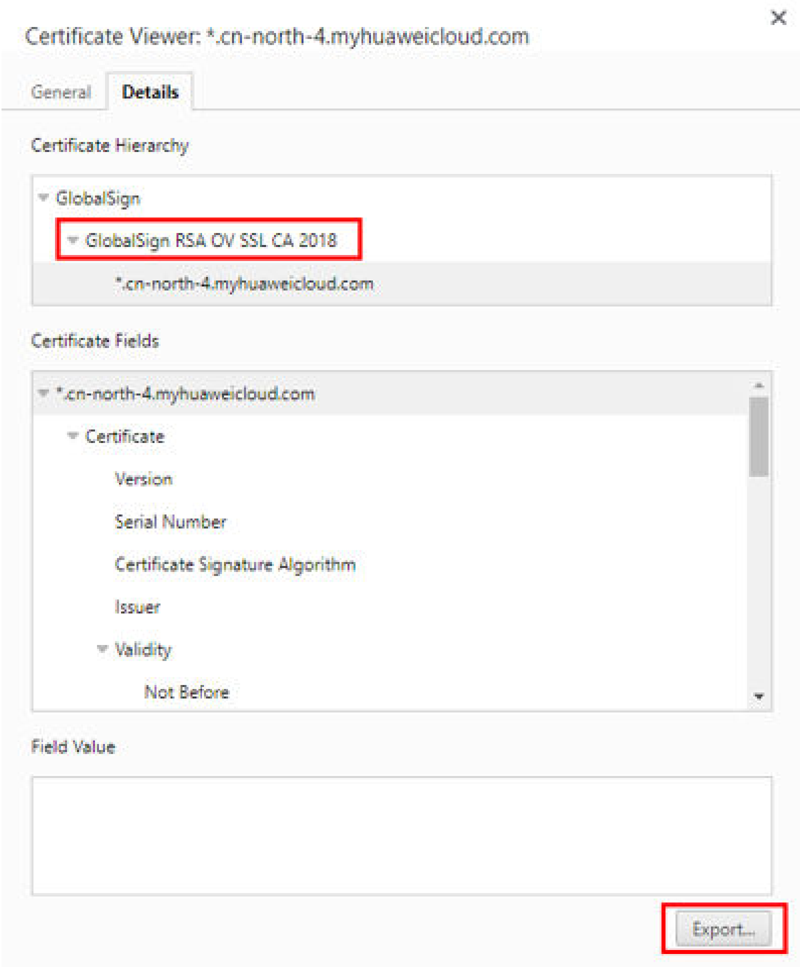
The RESTful API is used to access the KMS. When you enter the URL of the API in the address bar of the browser, ignore the failure page in **Step 2**. The browser has automatically downloaded the CA certificate in advance even if the failure page is displayed.

Step 1 Enter the domain name. Open a browser. In the Huawei Cloud scenario, enter the IP addresses of the IAM and KMS servers. For details about how to obtain the IP addresses, see **Master Key Generation Phase**.

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.



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.



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 APIs, for example, dbe_output.print_line(), decryption is not performed on the APIs 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** or **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 escaping of the PL/pgSQL syntaxes. The CREATE FUNCTION AS '*Syntax body*' syntax whose syntax body is enclosed in single quotation marks (') can be replaced by the CREATE FUNCTION AS $$*Syntax body*$$ syntax.
12. The definitions of encrypted columns 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 columns can be encrypted only after the client is disconnected or the cache of the encrypted columns 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 gs_encrypted_proc system catalog. 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 Setting a Ledger Database

3.1 Overview

Context

The ledger database, which integrates a blockchain idea, records a user operation in two types of historical tables: a user history table and a global blockchain table. When a user creates a tamper-proof user table, the system automatically adds a hash column to the table to save the hash summary of each row of data. In blockchain mode, a user history table is created to record the change behavior of each data record in the user table. The user's modification to the tamper-proof user table will be all recorded in the global blockchain table. Because the history table can only be appended and cannot be modified, the records in the history table are concatenated to form the modification history of the tamper-proof user table.

The name and structure of the user history table are as follows:

Table 3-1 Columns in the blockchain.<*schemaname*>_<*tablename*>_hist user history table

Column Name	Data Type	Description
rec_num	bigint	Sequence number of a row-level modification operation in the history table.
hash_ins	hash16	Hash value of the data row inserted by the INSERT or UPDATE operation.
hash_del	hash16	Hash value of the data row deleted by the DELETE or UPDATE operation.
pre_hash	hash32	Summary of the data in the history table of the current user.

Table 3-2 Mapping between hash_ins and hash_del

-	hash_ins	hash_del
INSERT	(√) Hash value of the inserted row.	Empty.
DELETE	Empty.	(√) Hash value of the deleted row.
UPDATE	(√) Hash value of the newly inserted data.	(√) Hash value of the row before deletion.

Procedure

Step 1 Create a schema in tamper-proof mode.

For example, create **ledgernsp** in tamper-proof mode.

```
gaussdb=# CREATE SCHEMA ledgernsp WITH BLOCKCHAIN;
```

NOTE

To create a table in tamper-proof mode or change the common mode to the tamper-proof mode, set **enable_ledger** to **on**. The default value of **enable_ledger** is **off**.

Step 2 Create a tamper-proof user table in tamper-proof mode.

For example, create a tamper-proof user table **ledgernsp.usertable**.

```
gaussdb=# CREATE TABLE ledgernsp.usertable(id int, name text);
```

Check the structure of the tamper-proof user table and the corresponding user history table.

```
gaussdb=# \d+ ledgernsp.usertable;
gaussdb=# \d+ blockchain.ledgernsp_usertable_hist;
```

The command output is as follows:

```
gaussdb=# \d+ ledgernsp.usertable;
                    Table "ledgernsp.usertable"
 Column     |  Type   | Modifiers | Storage  | Stats target | Description
--------+---------+-----------+----------+--------------+-------------
 id         | integer |           | plain    |              |
 name       | text    |           | extended |              |
 hash_69dd43   | hash16  |           | plain    |              |
Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no
History table name: ledgernsp_usertable_hist

gaussdb=# \d+ blockchain.ledgernsp_usertable_hist;
            Table "blockchain.ledgernsp_usertable_hist"
  Column  |  Type  | Modifiers | Storage | Stats target | Description
----------+--------+-----------+---------+--------------+-------------
 rec_num  | bigint |           | plain   |              |
 hash_ins | hash16 |           | plain   |              |
 hash_del | hash16 |           | plain   |              |
 pre_hash | hash32 |           | plain   |              |
Indexes:
    "gs_hist_69dd43_index" PRIMARY KEY, btree (rec_num int4_ops) TABLESPACE pg_default
Has OIDs: no
```

```
Distribute By: HASH(rec_num)
Location Nodes: ALL DATANODES
Options: internal_mask=263
```

NOTE

1. In tamper-proof mode, only row-store tables are tamper-proof tables. Temporary tables, foreign tables, unlogged tables, and non-row-store tables do not have tamper-proof attributes.
2. When a tamper-proof table is created, a system column used for check is automatically added. Therefore, the maximum number of columns in the tamper-proof table is 1599.

Step 3 Modify the data in the tamper-proof user table.

For example, execute **INSERT**, **UPDATE**, or **DELETE** on the tamper-proof user table.

```
gaussdb=# INSERT INTO ledgernsp.usertable VALUES(1, 'alex'), (2, 'bob'), (3, 'peter');
INSERT 0 3
gaussdb=# SELECT *, hash_69dd43 FROM ledgernsp.usertable ORDER BY id;
 id | name  |       hash_69dd43
----+-------+------------------
  1 | alex  | 1f2e543c580cb8c5
  2 | bob   | 8fcd74a8a6a4b484
  3 | peter | f51b4b1b12d0354b
(3 rows)

gaussdb=# UPDATE ledgernsp.usertable SET name = 'bob2' WHERE id = 2;
UPDATE 1
gaussdb=# SELECT *, hash_69dd43 FROM ledgernsp.usertable ORDER BY id;
 id | name  |       hash_69dd43
----+-------+------------------
  1 | alex  | 1f2e543c580cb8c5
  2 | bob2  | 437761affbb7c605
  3 | peter | f51b4b1b12d0354b
(3 rows)

gaussdb=# DELETE FROM ledgernsp.usertable WHERE id = 3;
DELETE 1
gaussdb=# SELECT *, hash_69dd43 FROM ledgernsp.usertable ORDER BY id;
 id | name |       hash_69dd43
----+------+------------------
  1 | alex | 1f2e543c580cb8c5
  2 | bob2 | 437761affbb7c605
(2 rows)
```

Step 4 Delete tables and schemas.

This step can be performed only after you have completed other operations (if any) on the ledger database.

```
gaussdb=# DROP TABLE ledgernsp.usertable;
DROP TABLE
gaussdb=# DROP SCHEMA ledgernsp;
DROP SCHEMA
```

----End

3.2 Viewing Historical Operation Records in the Ledger

Prerequisites

- You are an audit administrator or a role that has the AUDITADMIN permission.

- The database is running properly, and a series of addition, deletion, and modification operations are performed on the tamper-proof database to ensure that operation records are generated in the ledger for query.
- The global blockchain table of each CN in the database records data separately. The global blockchain table records only the SQL operations performed on the current CN.

Context

- Only users with the AUDITADMIN attribute can view historical operation records in the ledger. For details about database users and how to create users, see "Database Security Management > Managing Users and Their Permissions > Users" in *Developer Guide*.
- To query the global blockchain table gs_global_chain, run the following command:

```
SELECT * FROM gs_global_chain;
```

  This table contains 10 columns. For details about each column, see "System Catalogs and System Views > System Catalogs > GS_GLOBAL_CHAIN" in *Developer Guide*.
- To query the user history table in BLOCKCHAIN schema, the operation is as follows:

  For example, if the schema of the user table is **ledgernsp**, the table name is **usertable**, and the name of the corresponding user history table is **blockchain.ledgernsp_usertable_hist**, you can run the following command:

```
SELECT * FROM blockchain.ledgernsp_usertable_hist;
```

  The user history table contains four fields. For details about the meaning of each field, see **Table 3-1**.

  NOTE

  Generally, the name of a user history table is in the format of blockchain.<*schemaname*>_<*tablename*>_hist. If the schema name or table name of the tamper-proof user table is too long, the length of the table name generated using the preceding format may exceed the upper limit. In this case, the blockchain.<*schema_oid*>_<*table_oid*>_hist format is used to name the table.

Procedure

Step 1 View records in the global blockchain table.

```
gaussdb=# SELECT * FROM gs_global_chain;
```

The query result is as follows:

```
 blocknum |  dbname  | username |          starttime           | relid |  relnsp   |  relname  |     relhash
|            globalhash            |
              txcommand
----------+----------+----------+------------------------------+-------+-----------+-----------+------------------
+----------------------------------+------------------
------------------------------------------------------------
        0 | testdb | omm      | 2021-04-14 07:00:46.32757+08  | 16393 | ledgernsp | usertable |
a41714001181a294 | 6b5624e039e8aee36bff3e8295c75b40 | insert into ledge
rnsp.usertable values(1, 'alex'), (2, 'bob'), (3, 'peter');
        1 | testdb | omm      | 2021-04-14 07:01:19.767799+08 | 16393 | ledgernsp | usertable |
b3a9ed0755131181 | 328b48c4370faed930937869783c23e0 | update ledgernsp.
usertable set name = 'bob2' where id = 2;
        2 | testdb | omm      | 2021-04-14 07:01:29.896148+08 | 16393 | ledgernsp | usertable |
0ae4b4e4ed2fcab5 | aa8f0a236357cac4e5bc1648a739f2ef | delete from ledge
rnsp.usertable where id = 3;
```

The query result indicates that user **omm** has consecutively executed three DML commands: **INSERT**, **UPDATE**, and **DELETE**.

Step 2 View records in the user history table.

```
gaussdb=# SELECT * FROM blockchain.ledgernsp_usertable_hist;
```

The query result is as follows:

```
 rec_num |     hash_ins     |     hash_del     |             pre_hash
---------+------------------+------------------+----------------------------------
       0 | 1f2e543c580cb8c5 |                  | e1b664970d925d09caa295abd38d9b35
       1 | 8fcd74a8a6a4b484 |                  | dad3ed8939a141bf3682043891776b67
       2 | f51b4b1b12d0354b |                  | 53eb887fc7c4302402343c8914e43c69
       3 | 437761affbb7c605 | 8fcd74a8a6a4b484 | c2868c5b49550801d0dbbbaa77a83a10
       4 |                  | f51b4b1b12d0354b | 9c512619f6ffef38c098477933499fe3
(5 rows)
```

The query result shows that user **omm** inserts three rows of data to the **ledgernsp.usertable** table, updates one row of data, deletes one row of data, and leaves two rows of data, and the hash values are **1f2e543c580cb8c5** and **437761affbb7c605**.

Step 3 Query user table data and verification columns.

```
gaussdb=# SELECT *, hash_69dd43 FROM ledgernsp.usertable;
```

The query result is as follows:

```
 id | name |       hash_69dd43
----+------+------------------
  1 | alex | 1f2e543c580cb8c5
  2 | bob2 | 437761affbb7c605
(2 rows)
```

The query result indicates that the remaining two records in the user table are the same as those in step 2.

----End

3.3 Checking Ledger Data Consistency

Prerequisites

The database is running properly, and a series of addition, deletion, and modification operations are performed on the tamper-proof database to ensure that operation records are generated in the ledger for query.

Context

- Currently, the ledger database provides two verification APIs: ledger_hist_check(text, text) and ledger_gchain_check(text, text). When a common user calls a verification API, only the tables that the user has the permission to access can be verified.
- The API for verifying the tamper-proof user table and user history table is pg_catalog.ledger_hist_check. To verify a table, run the following command:

  ```
  SELECT pg_catalog.ledger_hist_check(schema_name text,table_name text);
  ```

  If the verification is successful, the function returns **t**. Otherwise, the function returns **f** and the cause of failure.

- The pg_catalog.ledger_gchain_check API is used to check whether the tamper-proof user table, user history table, and global blockchain table are consistent. To verify consistency, run the following command:

  ```
  SELECT pg_catalog.ledger_gchain_check(schema_name text, table_name text);
  ```

  If the verification is successful, the function returns **t**. Otherwise, the function returns **f** and the cause of failure.

Procedure

Step 1 Check whether the tamper-proof user table **ledgernsp.usertable** is consistent with the corresponding user history table.

```
gaussdb=# SELECT pg_catalog.ledger_hist_check('ledgernsp', 'usertable');
```

The query result is as follows:

```
 ledger_hist_check
-------------------
 t
(1 row)
```

The query result shows that the results recorded in the tamper-proof user table and user history table are consistent.

Step 2 Check whether the records in the tamper-proof table **ledgernsp.usertable** are the same as those in the corresponding user history table and global blockchain table.

```
gaussdb=# SELECT pg_catalog.ledger_gchain_check('ledgernsp', 'usertable');
```

The query result is as follows:

```
 ledger_gchain_check
---------------------
 t
(1 row)
```

The query result shows that the records of **ledgernsp.usertable** in the preceding three tables are consistent and no tampering occurs.

----End

3.4 Archiving a Ledger Database

Prerequisites

- You are an audit administrator or a role that has the AUDITADMIN permission.
- The database is running properly, and a series of addition, deletion, and modification operations are performed on the tamper-proof database to ensure that operation records are generated in the ledger for query.
- The storage path **audit_directory** of audit files has been correctly configured in the database.

Context

- Currently, the ledger database provides two archiving APIs: ledger_hist_archive(text, text) and ledger_gchain_archive(text, text). Only audit administrators can call the ledger database APIs.

- The pg_catalog.ledger_hist_archive API archives the user history table data of the current DN. The operation is as follows:
  ```
  SELECT pg_catalog.ledger_hist_archive(schema_name text,table_name text);
  ```
  If the archiving is successful, the function returns **t**. Otherwise, the function returns **f** and the cause of failure.
- The pg_catalog.ledger_gchain_archive API archives the global historical table data of the current CN. The operation is as follows:
  ```
  SELECT pg_catalog.ledger_gchain_archive();
  ```
  If the archiving is successful, the function returns **t**. Otherwise, the function returns **f** and the cause of failure.

Procedure

Step 1 Run the **EXECUTE DIRECT** statement to archive data on a DN.

```
gaussdb=# EXECUTE DIRECT ON (datanode1) 'select pg_catalog.ledger_hist_archive(''ledgernsp'',
''usertable'');';
```

The query result is as follows:

```
 ledger_hist_archive
---------------------
 t
(1 row)
```

The user history table is archived as a record:

```
gaussdb=# EXECUTE DIRECT ON (datanode1) 'SELECT * FROM blockchain.ledgernsp_usertable_hist;';
 rec_num |     hash_ins     |     hash_del     |             pre_hash
---------+------------------+------------------+----------------------------------
       3 | e78e75b00d396899 | 8fcd74a8a6a4b484 | fd61cb772033da297d10c4e658e898d7
(1 row)
```

The command output indicates that the user history table of the **datanode1** node is exported successfully.

Step 2 Connect to the CN to export the global blockchain table.

```
gaussdb=# SELECT pg_catalog.ledger_gchain_archive();
```

The query result is as follows:

```
 ledger_gchain_archive
-----------------------
 t
(1 row)
```

The global history table will be archived to *n* (number of user tables) data records by user table:

```
gaussdb=# SELECT * FROM gs_global_chain;
 blocknum |  dbname  | username |           starttime           | relid |  relnsp   |  relname  |     relhash
|            globalhash            | txcommand
----------+----------+----------+-------------------------------+-------+-----------+-----------+------------------
+----------------------------------+-----------
        1 | testdb | libc     | 2021-05-10 19:59:38.619472+08 | 16388 | ledgernsp | usertable |
57c101076694b415 | be82f98ee68b2bc4e375f69209345406 | Archived.
(1 row)
```

The command output indicates that the global blockchain table of the current CN is successfully exported.

----End

3.5 Repairing a Ledger Database

Prerequisites

- You are an audit administrator or a role that has the AUDITADMIN permission.
- The database is running properly, and a series of addition, deletion, and modification operations are performed on the tamper-proof database to ensure that operation records are generated in the ledger for query.

Context

- The current ledger database mechanism is as follows: The global blockchain table is stored on the CN, and data on each CN is independent. User history tables are stored on DNs and record data changes in the tamper-proof tables on DNs. Therefore, when data redistribution is triggered, data in the tamper-proof table may be inconsistent with that in the user history table. In this case, you need to use the ledger_hist_repair(text, text) API to restore the user history table on a specified DN. After the fault is rectified, the result of calling the historical table verification API on the current DN is **true**. When CNs are removed or repaired, data in the global blockchain table may be lost or inconsistent with that in the user history table. In this case, you need to use the ledger_gchain_repair(text, text) API to restore the global blockchain table in the entire cluster. After the fault is rectified, the result of calling the global blockchain table verification API is **true**.
- The API for repairing a user history table is pg_catalog.ledger_hist_repair. To repair the table, run the following command:
  ```
  SELECT pg_catalog.ledger_hist_repair(schema_name text,table_name text);
  ```
  If the repair is successful, the function returns the hash increment of the user history table during the repair.

  Note: When performing flashback DROP on a user table, you can use this function to repair the names of the user table and user history table. For details, see **Repairing the Names of a User table and User History Table**.
- The API for repairing the global blockchain table is pg_catalog.ledger_gchain_repair. To repair the table, run the following command:
  ```
  SELECT pg_catalog.ledger_gchain_repair(schema_name text,table_name text);
  ```
  If the repair is successful, the function returns the total hash value of the specified table in the global blockchain table.

Restoring Data in the User Table and Global Blockchain Table

The **omm** user is used as an example. The procedure is as follows:

Step 1 Log in to the primary database node as the OS user **omm**.

Step 2 Run **EXECUTE DIRECT** to restore a history table on a DN.

```
gaussdb=# EXECUTE DIRECT ON (datanode1) 'select pg_catalog.ledger_hist_repair(''ledgernsp'',
''usertable'');';
```

The query result is as follows:

```
 ledger_hist_repair
--------------------
 84e8bfc3b974e9cf
(1 row)
```

The query result indicates that the user history table on the **datanode1** node is successfully repaired. The hash increment of the user history table is **84e8bfc3b974e9cf**.

Step 3 Connect to the CN to repair the global blockchain table.

```
gaussdb=# SELECT pg_catalog.ledger_gchain_repair('ledgernsp', 'usertable');
```

The query result is as follows:

```
 ledger_gchain_repair
----------------------
 a41714001181a294
(1 row)
```

The command output indicates that the global blockchain table of the current cluster is successfully repaired and a piece of repair data is inserted into the current CN. The hash value of the data is **a41714001181a294**.

----End

Repairing the Names of a User table and User History Table

The flashback DROP function has been enabled by using the **enable_recyclebin** and **recyclebin_retention_time** parameters to repair the names of user tables and user history tables. The following is an example:

- Perform flashback DROP on a user table. Use ledger_hist_repair to repair the names of the user table and user history table.

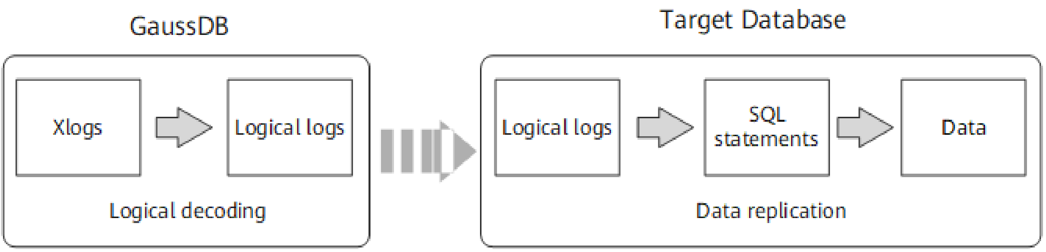
```
-- Perform flashback DROP on the user table and use the ledger_hist_repair API to repair the name of the user history table.
gaussdb=# CREATE TABLE ledgernsp.tab2(a int, b text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'rec_num' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# DROP TABLE ledgernsp.tab2;
DROP TABLE
gaussdb=# SELECT rcyrelid, rcyname, rcyoriginname FROM gs_recyclebin;
 rcyrelid |           rcyname            |    rcyoriginname
----------+------------------------------+---------------------
    32838 | BIN$39B523388046$55C8400==$0 | tab2
    32846 | BIN$39B52338804E$55C90E8==$0 | gs_hist_tab2_index
    32843 | BIN$39B52338804B$55C96A0==$0 | ledgernsp_tab2_hist
    32841 | BIN$39B523388049$55C9EE0==$0 | pg_toast_32838
(4 rows)
-- Perform flashback DROP on the user table.
gaussdb=# TIMECAPSULE TABLE ledgernsp.tab2 TO BEFORE DROP;
TimeCapsule Table
-- Use the ledger_hist_repair API to repair the name of the user history table.
gaussdb=# SELECT ledger_hist_repair('ledgernsp', 'tab2');
 ledger_hist_repair
--------------------
 0000000000000000
(1 row)

gaussdb=# \d+ ledgernsp.tab2;
                        Table "ledgernsp.tab2"
  Column    | Type   | Modifiers | Storage  | Stats target | Description
```

```
-------------+---------+-----------+----------+--------------+-------------
 a           | integer |           | plain    |              |
 b           | text    |           | extended |              |
 hash_1d2d14 | hash16  |           | plain    |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast
History table name: ledgernsp_tab2_hist

-- Perform flashback DROP on the user table and use the ledger_hist_repair API to repair the name of
the user table.
gaussdb=# CREATE TABLE ledgernsp.tab3(a int, b text);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'rec_num' as the distribution column by
default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
gaussdb=# DROP TABLE ledgernsp.tab3;
DROP TABLE
gaussdb=# SELECT rcyrelid, rcyname, rcyoriginname FROM gs_recyclebin;
 rcyrelid |           rcyname            |    rcyoriginname
----------+------------------------------+---------------------
    32952 | BIN$80B6233880B8$FECFF98==$0 | tab3
    32960 | BIN$80B6233880C0$FED0C98==$0 | gs_hist_tab3_index
    32957 | BIN$80B6233880BD$FED1250==$0 | ledgernsp_tab3_hist
    32955 | BIN$80B6233880BB$FED1A00==$0 | pg_toast_32952
(4 rows)
-- Perform flashback DROP on the user history table.
gaussdb=# TIMECAPSULE TABLE blockchain.ledgernsp_tab3_hist TO BEFORE DROP;
TimeCapsule Table
-- Obtain the rcyname corresponding to the user table in the recycle bin and use the
ledger_hist_repair API to repair the name of the user table.
gaussdb=# SELECT ledger_hist_repair('ledgernsp', 'BIN$80B6233880B8$FECFF98==$0');
 ledger_hist_repair
--------------------
 0000000000000000
(1 row)

gaussdb=# \d+ ledgernsp.tab3;
                        Table "ledgernsp.tab3"
   Column    |  Type   | Modifiers | Storage  | Stats target | Description
-------------+---------+-----------+----------+--------------+-------------
 a           | integer |           | plain    |              |
 b           | text    |           | extended |              |
 hash_7a0c87 | hash16  |           | plain    |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast
History table name: ledgernsp_tab3_hist

-- Drop the table.
gaussdb=# DROP TABLE ledgernsp.tab2 PURGE;
DROP TABLE
gaussdb=# DROP TABLE ledgernsp.tab3 PURGE;
DROP TABLE
```

4 Logical Replication

Logical replication consists of logical decoding and data replication. Logical decoding extracts transaction-level logical logs. The service or database middleware parses the logs and replicates data. GaussDB can periodically synchronize data to heterogeneous databases using the data migration tool. However, it does not support real-time data replication and cannot meet the requirements for real-time data synchronization between heterogeneous databases. To this end, GaussDB provides the logical decoding feature to parse Xlogs and generate logical logs for the target database to parse in real time, implementing data replication. This feature lowers the requirements on the target database, supporting data synchronization between heterogeneous databases and between homogeneous but different databases, and allows data to be read and written during data synchronization, achieving low data synchronization latency. **Figure 4-1** shows the implementation. This section describes only logical decoding.

Figure 4-1 Logical Replication



4.1 Logical Decoding

4.1.1 Overview

Description

Logical decoding provides basic transaction decoding capabilities for logical replication. GaussDB uses SQL functions for logical decoding. This method features easy function calling, requires no tools to obtain logical logs, and provides specific

APIs for interconnecting with external replay tools, saving the need of additional adaptation.

Logical logs are generated by transaction and can be output only after a transaction is committed. In addition, logical decoding is driven by users. Therefore, to prevent Xlogs from being recycled by the system at the beginning of a transaction or required transaction information from being recycled by VACUUM, a logical replication slot is added to GaussDB to block Xlog recycling.

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

Prerequisites

- Currently, logical logs are extracted from CNs or DNs. If logical replication is performed, SSL connections must be used. Therefore, ensure that the SSL-related GUC parameter on the corresponding node 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 node.

  NOTE

  - Each logical replication slot decodes modifications to only a single database. To decode multiple databases, you need to create multiple logical replication slots.
  - During multi-channel logical replication and synchronization, the source database needs to create an independent logical replication slot for each logical replication link.
  - A single CN can run a maximum of 10 distributed strong-consistency decoding tasks at the same time. A single DN can run a maximum of 20 decoding tasks at the same time.
- A user needs to connect to a database through a DN port before using SQL functions to perform logical decoding. For details, see **Logical Decoding by SQL Functions**. If a CN port is used to connect to the database, **EXECUTE DIRECT ON (***datanode_name***)** *'statement'* is needed to execute SQL functions.
- Only the initial user and users with the REPLICATION permission can perform this operation. When separation of duties is disabled, database administrators can perform logical replication operations. When separation of duties is enabled, database administrators are not allowed to perform logical replication operations.

Precautions

- Decoding of DML operations on data page replication is not supported.
- Logical replication does not support online cluster scale-out. Before online scale-out, you need to delete existing logical replication slots. After the scale-out is complete, create logical replication slots again.
- The size of a single tuple cannot exceed 1 GB, and decoding results may be larger than inserted data. Therefore, it is recommended that the size of a single tuple be less than or equal to 500 MB.
- GaussDB supports the following types of data to be decoded: INTEGER, BIGINT, SMALLINT, TINYINT, SERIAL, SMALLSERIAL, BIGSERIAL, FLOAT, DOUBLE PRECISION, BOOLEAN, BIT(n), BIT VARYING(n), DATE, TIME[WITHOUT TIME ZONE], TIMESTAMP[WITHOUT TIME ZONE], CHAR(n), VARCHAR(n), TEXT, and CLOB (decoded into the text format).
- For floating-point data in a non-M-compatible database and floating-point data without scale in an M-compatible database, the decoding result precision parameter **extra_float_digits** is set to **3**. For details about this parameter, see the description of the GUC parameter **float_shortest_precision**.
- Decoding of DDL operations on the PUBLIC SCHEMA is not supported.
- Logical decoding of DML and DDL operations on an M-compatible database is supported. The supported data types are as follows:
  - Integer: TINYINT(M), SMALLINT(M), MEDIUMINT(M), BIGINT(M), INT(M)/INTEGER, and BOOL/BOOLEAN.
  - Floating-point: FLOAT(M,D) and DOUBLE(M,D).
  - Fixed-point: DECIMAL(M,D) and NUMERIC(M,N).
  - BIT: BIT.
  - Binary string: CHAR(N), VARCHAR(N), BINARY, and VARBINARY.
  - Text: TINYTEXT, TEXT, MEDIUMTEXT, and LONGTEXT.
  - Date: DATE, TIME, DATETIME, TIMESTAMP, and YEAR.
  - LOB: TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB.
  - Data type attribute: UNSIGNED and ZEROFILL.
- For M-compatible databases, the special restrictions on logical decoding are as follows:
  - Options such as **[partition_options]**, **ENGINE**, **ROW_FORMAT**, **algorithm_option**, and **lock_option** in CREATE TABLE, ALTER TABLE, and DROP TABLE have no syntax effect. Such syntax is not decoded and output.
  - In the ALTER SCHEMA, CREATE SCHEMA, DROP SCHEMA, ALTER DATABASE, CREATE DATABASE, and DROP DATABASE syntaxes, ALTER DATABASE, CREATE DATABASE, and DROP DATABASE are decoded as ALTER SCHEMA, CREATE SCHEMA, and DROP SCHEMA because they are equivalent to schemas in M-compatible mode.
  - Different character setting syntaxes, such as CHARACTER SET, CHAR SET, and CHARSET, are decoded as CHARACTER SET.
  - For the ALTER TABLE tbl_name DROP {INDEX | KEY} index_name syntax, the logical decoding result is DROP INDEX. For the ALTER TABLE tbl_name DROP {primary key | {index | key} index_name} syntax for

deleting a primary key, the logical decoding result is ALTER TABLE tbl_name DROP CONSTRAINT index_name.

  – For the ALTER TABLE tbl_name ADD INDEX syntax, the logical decoding result is CREATE INDEX.
  – During DML statement decoding, if the inserted data is of the time type (time, timestamp, or datetime) with precision, the maximum precision 6 is used for decoding.
  – For data types (BINARY, VARBINARY, TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB) that support binary input, if the data contains **'\0'**, the data will be truncated regardless of output from serial decoding/function decoding or parallel decoding in the JSON/TEXT format. Only the output in BINARY format can be decoded in parallel (that is, the decoding task parameter **decode-style** is set to **'b'**).

- REPLACE is a DML syntax and will be decoded as an actual data operation, such as INSERT or DELETE+INSERT (in case of a primary key or unique key conflict).
- The COPY/LOAD syntax is a DML syntax and is decoded as an actual operation on data, for example, multiple INSERT operations.
- The name of a logical replication slot must be fewer than 64 characters. When you use SQL functions to create or use a replication slot, the replication slot name can contain only lowercase letters, digits, underscores (_), question marks (?), hyphens (-), and periods (.). In addition, a single period (.) or double periods (..) cannot be used as the replication slot name alone. When the JDBC API is called to create or use a replication slot, the replication slot name can contain only lowercase letters, digits, and underscores (_). (Uppercase letters are allowed, but they will be converted into lowercase letters. For example, if you enter the name as **MSLOT**, the actual input is **mslot**.)
- 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 during logical decoding. After forcible switchover, you need to export all data again.
- During decoding on the standby node, the decoded data may increase due to a switchover or failover operation, 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.
- If you need to delete a database during decoding on the standby node, ensure that the database of the corresponding logical replication slot is deleted by the primary node after data migration is complete. Currently, it is not necessary to delete the logical replication slot on a database before the database is deleted. As a result, the database may be deleted from the primary node before the decoding on the standby node is complete. In this case, when the standby node replays to the time when the database is deleted, the standby node exits decoding even if it has not completed all decoding tasks in the current database. If the standby node reconnects to the database, decoding on the standby node cannot be started because the database has been deleted. As a result, decoding tasks are damaged.

- Replication slots can be created or deleted only on CNs and primary DNs. If the replication slot to be deleted is the last one, alarms "replicationSlotMinLSN is INVALID_WAL_REC_PTR!!!" and "replicationSlotMaxLSN is INVALID_WAL_REC_PTR!!!" are generated after the deletion is complete.
- When a replication slot on the primary node is deleted, the replication slot on the standby node may not be deleted in time due to Xlog replay delay on the standby node. If such replication slot is used for decoding on the standby node, a decoding error will be reported when the replication slot is deleted during replay. Before decoding on the standby node, check whether the value of **restart_lsn** of the replication slot on the standby node is the same as that on the primary node. If they are different, the replication slot is a residual replication slot.
- The same replication slot for decoding cannot be used between the primary and standby DNs or between different standby DNs at the same time. Otherwise, data inconsistency or other errors may occur.
- 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 no long transaction is started during the creation of a logical replication slot. If a long transaction is started, the creation of the logical replication slot will be blocked.
- Decoding of DML operations on global temporary tables is not supported.
- Decoding of DML operations on local temporary tables is not supported.
- For the SELECT INTO statement, only the DDL operations for creating the target table are decoded; the DML operations for inserting data will not be decoded.
- To parse the UPDATE and DELETE statements of an Astore table, you need to configure the **REPLICA IDENTITY** attribute for the table. If the table does not have a primary key, set the **REPLICA IDENTITY** attribute to **FULL**. For details about the configuration method, see column **REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }** in "SQL Reference > SQL Syntax > A > 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 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 > A > ALTER TABLE" in *Developer Guide*.
- If a transaction has too many sub-transactions, too many files are flushed to disk. 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 **7FFFFFFF/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 **7FFFFFFF/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 CSN-based logical replication slot on the CN is only a placeholder, its slot number does not update along with logical decoding, and log recycling is not blocked.
- 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. 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.
- Decoding of DML operations on unlogged tables is not supported.
- During backup and restoration, all logical replication slots will be cleared after the instance is restored. If required, rebuild the slots.
- Replication slots must be deleted before the database is deleted. After the database where the logical replication slots reside is deleted, these replication slots become unavailable and need to be manually deleted. Otherwise, WAL recycling will be blocked.
- When a transaction generates a large number of sub-transactions that need to be flushed to disk, the number of opened file handles may exceed the upper limit. In this case, set GUC parameter **max_files_per_process** to a value greater than twice the upper limit of sub-transactions.

- Global secondary indexes are not supported, and decoding of DML statements for modifying distribution keys is not supported.
- The standby cluster of a DR cluster does not support logical decoding using SQL system functions or tools.
- The ledger database function is not supported. In the current version, if DML operations related to the ledger database function exist in the database where decoding is enabled, the decoding result contains hash columns. As a result, the replay fails.
- In scale-out scenarios, if logical replication slots are created in the cluster, the scale-out will fail. Therefore, you need to delete the existing logical replication slots before scale-out.
- Information constraint options (such as **not enforced**) in PRIMARY KEY, FOREIGN KEY, and UNIQUE constraints are not supported. Information constraints in DDL statements are not decoded, while other parts of the DDL statements are decoded as usual. For example, the "CREATE TABLE test(a int primary key not enforced);" statement is decoded as "CREATE TABLE test(a int);".
- When the CREATE TABLE syntax is used to create a table, DDL statements with the **MURMURHASH** option are not decoded.
- Only WALs with **wal_level** set to **logical** can be decoded. For non-logical logs, serial decoding (including function decoding) does not produce corresponding values and types, and parallel decoding does not output the logical logs.
- The logical decoding feature supports the following types of DML operations in Xlogs:
  - Astore: INSERT, DELETE, UPDATE, and MULTI-INSERT.
  - Ustore: INSERT, DELETE, UPDATE, and MULTI-INSERT.
- Compared with the scenario where **wal_level** is set to **hot_standby**, the Xlog data volume increases when **wal_level** is set to **logical**. For example, in the TPC-C scenario, the Astore Xlog bloat rate is about 11%, and the Ustore Xlog bloat rate is about 110%.
- In M-compatible mode, do not perform logical replication between databases with different **lower_case_table_names** parameter settings. Otherwise, data may be lost.
- When decoding is performed in a case-insensitive database, the table name or username in the decoding option whitelist (**white-table-list**) or blacklist (**exclude-users**) must be lowercase regardless of whether the table name or username is created in uppercase or lowercase.
- During cluster scale-out, non-service tables are created and decoded.
- When the gs_logical_decode_start_observe function is used for monitoring, if the replication slot is not in the active state, the error message "invalid slot name" is displayed.
- Function decoding does not support the '\0' character.
- The minimum node specifications supported by logical decoding are 8 vCPUs and 64 GB. If the node specifications are lower than the minimum specifications, logical decoding is not recommended.
- The logical replication slot protects the system catalog records of the version corresponding to the Xlog that is not parsed from being cleared too early. If a

logical replication slot is inactive or the client consumes data slowly, services may be affected as follows:

a. If there are a large number of DDL service statements, there will be too many versions of system catalogs, affecting the SQL command execution efficiency.
b. Due to the limitations of GaussDB data page management, two records whose XID difference exceeds 2^32 cannot be stored on the same page. If there are too many versions of system catalogs, service operations such as DDL and DML may fail to be executed, reporting error code GAUSS-21297.

Solution:

a. Clear the logical replication slots that are no longer used in a timely manner.
b. If the logical replication slot is updated too slowly, rectify the fault by referring to "Common Fault Demarcation and Troubleshooting > Common Logical Decoding Faults > Logical Decoding > Abnormal Logical Decoding Replication Slot" and "Common Fault Demarcation and Troubleshooting > Common Logical Decoding Faults > Logical Decoding > Slow Logical Decoding" in *Troubleshooting*.
c. If the problem in **b** persists, delete the replication slot and re-create the logical decoding task.

SQL Function Decoding Performance

1. In the Benchmarksql-5.0 with 100 warehouses, when pg_logical_slot_get_changes is used:
   - If 4,000 lines of data (about 5 MB to 10 MB of logs) are decoded at a time, the decoding performance ranges from 0.3 MB/s to 0.5 MB/s.
   - If 32,000 lines of data (about 40 MB to 80 MB of logs) are decoded at a time, the decoding performance ranges from 3 MB/s to 5 MB/s.
   - If 256,000 lines of data (about 320 MB to 640 MB of 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.
2. In the Benchmarksql-5.0 with 100 warehouses, when pg_logical_get_area_changes is used:
   - If 4,000 lines of data (about 5 MB to 10 MB of logs) are decoded at a time, the decoding performance ranges from 0.3 MB/s to 0.5 MB/s.
   - If 32,000 lines of data (about 40 MB to 80 MB of logs) are decoded at a time, the decoding performance ranges from 3 MB/s to 5 MB/s.
   - If 256,000 lines of data (about 320 MB to 640 MB of 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.

3. In the Benchmarksql-5.0 with 100 warehouses, when pg_logical_slot_get_binary_changes is used:
   - If 4,000 lines of data (about 5 MB to 10 MB of logs) are decoded at a time, the decoding performance ranges from 0.3 MB/s to 0.5 MB/s.
   - If 32,000 lines of data (about 40 MB to 80 MB of logs) are decoded at a time, the decoding performance ranges from 2 MB/s to 3 MB/s.
   - If 256,000 lines of data (about 320 MB to 640 MB of logs) are decoded at a time, the decoding performance ranges from 2 MB/s to 3 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_binary_changes and pg_replication_slot_advance are used, the decoding performance is 30% to 50% lower than that when pg_logical_slot_get_binary_changes is used.

Streaming Decoding Performance

The decoding performance (Xlog consumption) for connecting to a DN is greater than or equal to 100 MB/s and that for connecting to a CN is greater than or equal to 80 MB/s in the following standard parallel decoding scenario: 16-core CPU, 128 GB memory, network bandwidth > 200 MB/s, 10 to 100 columns in a table, 0.1 KB to 1 KB data in a single row, INSERT-dominant DML operations, fewer than 4,096 statements in a single transaction (that is, no flush to disk is required), **parallel-decode-num** set to **8**, decoding format as **'t'**, and batch sending function enabled. To ensure that the decoding performance meets the requirements and to minimize the impact on services, you are advised to set up only one parallel decoding connection on a standby node to ensure that the CPU, memory, and bandwidth resources are sufficient.

4.1.2 Logical Decoding Options

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

General Options

These options can be configured for both serial decoding and parallel decoding, but may be invalid. For details, see the description of related options.

- **include-xids**:

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

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

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

- **true**: The decoded **data** column contains XID information.

- **skip-empty-xacts**:

  Specifies whether to ignore empty transaction information during decoding.

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

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

- **include-timestamp**:

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

  Value range: Boolean. The default value is **false** in parallel decoding scenarios or **true** in SQL function decoding and serial decoding scenarios.

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

- **only-local**:

  Specifies whether to decode only local logs.

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

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

- **white-table-list**:

  Specifies a whitelist, including the schemas and tables 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.

  For serial decoding, the value range is an integer ranging from 0 to 100. The default value is **0**, indicating that memory control is disabled.

  For parallel decoding, the value ranges from 0 to 25% of the value of **max_process_memory**. The default value is **max_process_memory/4/1024**, where **1024** indicates the conversion from KB to MB. The value **0** indicates that this 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.

  For serial decoding, the value range is an integer ranging from 0 to 100. The default value is **0**, indicating that memory control is disabled.

  For parallel decoding, the value ranges from 0 to 50% of the value of **max_process_memory**. The default value is **max_process_memory/2/1048576**, where **1048576** indicates the conversion from KB to GB. The value **0** indicates that this memory control is disabled.

NOTE

Function decoding is serial decoding. For streaming decoding, setting **parallel-decode-num** to **1** indicates serial decoding; setting it to a value greater than 1 indicates parallel decoding.

- desc-memory-limit

  A memory control parameter, in MB. When the total memory of table metadata maintained by a logical decoding task is greater than the value of this parameter, some table metadata will be evicted.

  Value range: an integer ranging from 10 to 1024. The default value is **100**.

- **include-user**:

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

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

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

- **exclude-userids**:

  OIDs of blacklisted users. This parameter can be configured only for decoding tasks on directly connected DNs and does not apply to decoding tasks on CNs with strong consistency in distributed mode.

  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**:

  Specifies the 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. If no user is created when an Xlog is written, the system considers that the log user does not exist when the Xlog is decoded.

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

  - **false**: An error is reported and logical decoding exits if a user who does not exist in the blacklist specified by **exclude-users** is detected. If the user exists in the blacklist, operations of the user can be filtered out.
  - **true**: No error is reported and the decoding is normal if a user who does not exist in the blacklist specified by **exclude-users** is detected. If the user exists in the blacklist, operations of the user can be filtered out.

- **standby-connection**:

  Specifies whether to restrict decoding only on the standby node. Because no standby CNs are configured, this parameter applies only to DNs.

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.

NOTE

If the resource usage of the primary node is high and services are insensitive to real-time incremental data synchronization, you are advised to perform decoding on the standby node. If services have high requirements on real-time incremental data synchronization and the service pressure on the primary node is low, you are advised to perform decoding on the primary node.

- **sender-timeout**:

  Specifies the heartbeat timeout threshold between the GaussDB and the client. This option is valid 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**. The value **0** indicates that logical decoding does not proactively disconnect from the client. A small value, for example, 1 ms, indicates that decoding tasks may be interrupted.

- **change-log-max-len**:

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

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

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

  Specifies the threshold of the number of cached parallel decoding logs. This option is valid only for parallel decoding of connected DNs and is invalid for distributed strong consistency decoding, serial decoding, and SQL function decoding. If the number of locally cached logs is insufficient, more logs will be retrieved from the global cache.

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

- **enable-heartbeat**:

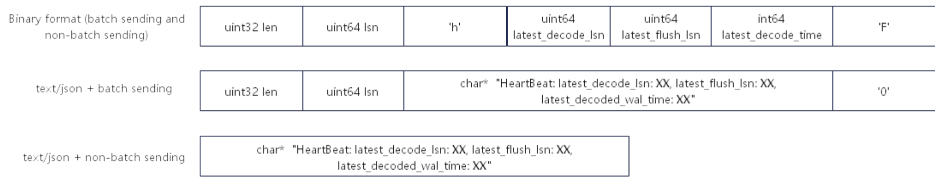
  Specifies whether to generate heartbeat logs. This option is valid only for streaming decoding.

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

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

NOTE

If the heartbeat log output option is enabled, heartbeat logs will be generated. The following uses parallel decoding as an example to describe how to parse heartbeat logs: 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.)



- **parallel-decode-num**:

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

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

NOTICE

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

- **output-order**:

  Specifies whether to output decoding results based on CSNs. This option is valid only for streaming decoding. When the system function is called, this option is invalid and only the value range is verified.

  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 section "Distributed Strong-Consistency 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 update the logical replication slot number. This option is valid 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 calls the log confirmation API to advance the logical replication slot.

- **skip-generated-columns**:

  Specifies whether to skip stored 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 stored generated columns. Stored generated columns are not supported in a distributed system and therefore, this parameter has no actual impact.

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

  - **true**/**on**: The decoding result of stored generated columns is not output.
  - **false**/**off**: The decoding result of stored generated columns is output.

  Virtual generated columns are not controlled by this parameter. The DML decoding result does not output virtual generated columns.

- **enable-ddl-decoding**:

  Specifies whether to enable logical decoding for DDL statements.

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

  - **true**: Logical decoding of DDL statements is enabled.
  - **false**: Logical decoding of DDL statements is disabled.

- **enable-ddl-json-format**:

  Specifies the DDL statement reverse parsing process and output format for logical decoding.

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

  - **true**: The DDL statement reverse parsing result is output in JSON format.
  - **false**: The DDL statement reverse parsing result is output in the format specified by **decode-style**.

- **timezone-is-utc**:

  Specifies the logical decoding control parameter, which is used to control the output of time type data with time zones (for example, timestamptz type in ORA-/MySQL-compatible mode, and timestamp type in M-compatible mode).

This parameter is valid only for streaming decoding and does not take effect for function decoding.

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

- **true**: The time of time zone 0 is forcibly output during decoding.
- **false**: The time of the current database time zone is output during decoding.

- **decode-sequence**:

  Specifies whether to output the decoding result of the change log of the sequence value, which is a logical decoding control parameter.

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

  - The value cannot be set to **true** currently.
  - **false**: The decoding result of the change log of the sequence value is not output.

NOTICE

Currently, **decode-sequence** can only be set to **false**. If **decode-sequence** is set to **true**, an error is reported when decoding is started and the decoding process exits.

- data-limit

  Controls the data volume output by logical decoding.

  When the GUC parameter **logical_decode_options_default** is used, the value is an integer in the range [0,100]. Unit: GB Default value: **10** The value **0** indicates that the size of the decoding result is not limited.

  The GUC parameter setting must be used together with the **data-limit** input parameter of the pg_logical_get_area_changes function. For details, see the description of the pg_logical_get_area_changes function in "SQL Reference > Functions and Operators > System Management Functions > Logical Replication Functions" in *Developer Guide*.

Distributed Strong-Consistency Decoding

- **logical-receiver-num**:

  Specifies the number of logical receivers started for distributed decoding. This option is valid only for streaming decoding. When the system function is called, this option is invalid and only the value range is verified.

  Value range: an integer ranging from 1 to 20. The 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**:

  Specifies the 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.

NOTE

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 parameter cannot be specified when a CN is connected for decoding. The program obtains the DN shard number, and the CN collects only the decoding result of the replication table of the DN shard.

- **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 the specified CSN. This option is set only when DNs are connected. BEGIN logs of the transaction with the specified CSN must be filtered out.

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

  NOTE

  This parameter cannot be specified when a CN is connected for decoding. The program uses this option to filter logs that may have been received when a decoding request is sent after the CN is connected to a DN.

Serial Decoding

- **force-binary**:

  Specifies whether to output the decoding result in binary format and display different behaviors in different scenarios.

  – For system functions pg_logical_slot_get_binary_changes and pg_logical_slot_peek_binary_changes:

    Value range: Boolean. The default value is **false**. The value is meaningless. The decoding result is always output in binary format.

  – For system functions pg_logical_slot_get_changes, pg_logical_slot_peek_changes, and pg_logical_get_area_changes:

    Value range: Boolean. The value is fixed at **false**. The decoding result is always output in text format.

  – For streaming decoding (supported only when DNs are connected):

    Value range: Boolean. The default value is **false**. The value is meaningless. The decoding result is always output in text format.

Parallel Decoding

The following configuration options are set only for streaming decoding:

- decode-style:

  If **enable-ddl-json-format** is set to **true**, the decoding format of DDL statements is controlled by **enable-ddl-json-format**, and **decode-style** specifies only the decoding format of DML statements. If **enable-ddl-json-format** is set to **false**, **decode-style** specifies the decoding format of both DML and DDL statements.

  Value range: **'j'**, **'t'**, or **'b'** of the char type, indicating the JSON, TEXT, or binary format, respectively.

Default value:

– If **decode-style** is not specified:

  For replication slot plug-ins mppdb_decoding and sql_decoding, the default value of **decode-style** is **'b'**, indicating decoding in binary format. For replication slot plug-ins parallel_binary_decoding, parallel_json_decoding, and parallel_text_decoding, the default values of **decode-style** are **'b'**, **'j'**, and **'t'** respectively, indicating decoding in binary, JSON, and TEXT formats, respectively.

– If **decode-style** is specified:

  Decoding is performed based on the specified decoding style.

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).

Take the mppdb_decoding plug-in as an example. When **decode-style** is set to **b**, decoding is performed in binary format. The result is as follows:

```
current_lsn: 0/CFE5C80 BEGIN CSN: 2357 first_lsn: 0/CFE5C80
current_lsn: 0/CFE5D40 INSERT INTO public.test1 new_tuple: {a[typid = 23]: "1", b[typid = 23]: "2"}
current_lsn: 0/CFE5E68 COMMIT xid: 78108
```

When **decode-style** is set to **j**, decoding is performed in JSON format. The result is as follows:

```
BEGIN CSN: 2358 first_lsn: 0/CFE6220
{"table_name":"public.test1","op_type":"INSERT","columns_name":["a","b"],"columns_type":
["integer","integer"],"columns_val":["3","3"],"old_keys_name":[],"old_keys_type":[],"old_keys_val":[]}
COMMIT XID: 78109
```

When **decode-style** is set to **t**, decoding is performed in TEXT format. The result is as follows:

```
BEGIN CSN: 2359 first_lsn: 0/CFE64D0
table public test1 INSERT: a[integer]:3 b[integer]:4
COMMIT XID: 78110
```

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 1-byte letter can be **B**, **C**, **I**, **U**, or **D**, representing BEGIN, COMMIT, INSERT, UPDATE, or DELETE, respectively.
4. If **B** is used in **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 decoding statements in this batch, and **F** indicates that decoding in this batch is complete.
5. If **C** is used in **3**:
   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 **3**:
   1. The next two bytes (uint16) indicate the length of the schema name.
   2. The schema name is read based on the preceding length.
   3. The next two bytes (uint16) indicate the length of the table name.
   4. The table name is read based on the preceding length.
   5. (Optional) If the next 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 following 4 bytes (uint32) indicate the OID of the current column type.
         4. The next 4 bytes (uint32) indicate the length of the value (stored in string format) in the current column. If the value is **0xFFFFFFFF**, it indicates null. If the value is **0**, it indicates a 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 1-byte letter is **P**, it indicates that the batch still needs to be decoded, and if the next 1-byte letter is **F**, it indicates that decoding of the batch ends.

- sending-batch:

  Specifies whether to send messages in batches.

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

  – **0**: The decoding results are sent one by one.

  – **1**: When the accumulated size of decoding results reaches 1 MB, decoding results are sent in batches.

  In the scenario where batch sending is enabled, if the decoding format is 'j' or 't', before each original decoding statement, a uint32 number is added indicating the length of the decoding result (excluding the current uint32 number), and a uint64 number is added indicating the LSN corresponding to the current decoding result.

NOTICE

In the CSN-based decoding scenario (that is, **output-order** is set to **1**), batch sending is limited to a single transaction (that is, if a transaction has multiple small statements, the statements can be batch sent). That is, multiple transactions are not sent in the same batch, and BEGIN and COMMIT statements are not batch sent.

- parallel-queue-size:

  Specifies the length of the queue for interaction between parallel logical decoding threads.

  Value range: an integer ranging from 2 to 1024. The value must be an integer power of 2. The default value is **128**.

  The queue length is positively correlated with the memory usage during decoding.

4.1.3 Logical Decoding by SQL Functions

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 primary DN in the GaussDB cluster as a user with the REPLICATION permission.

Step 2 Connect to the database through a DN port.

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

In the preceding command, *user1* indicates the username, *password* indicates the user password, *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 Connect to the database through the CN port, create table **t** in the database, and insert data into table **t**.

```
gaussdb=> CREATE TABLE t(a int PRIMARY KEY, b int);
gaussdb=> INSERT INTO t VALUES(3,3);
```

Step 5 Connect to the DN by referring to **Step 2** and read the decoding result of replication slot **slot1**. The number of decoded records is 4096.

NOTE

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

```
gaussdb=> SELECT * FROM pg_logical_slot_peek_changes('slot1', NULL, 4096);
location  |  xid  | data
-----------+-------
+-----------------------------------------------------------------------------------------------------------------------------
---------------------
------------------------------------------
 0/601C188 | 1010023 | BEGIN 1010023
 0/601ED60 | 1010023 | COMMIT 1010023 (at 2023-09-14 16:03:51.394287+08) 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 (at 2023-09-14 16:03:57.239821+08) 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

4.1.4 Logical Data Replication Using Streaming Decoding

A third-party replication tool extracts logical logs from GaussDB and replays them on the peer database.

For details about the code of the replication tool that uses JDBC to connect to the database, see "Application Development Guide > Development Based on JDBC > Example: Logical Replication Code" in *Developer Guide*.

4.1.5 Logical Decoding of DDL Statements

DDL statements can be properly executed on a GaussDB host and can be obtained using a logical decoding tool.

Table 4-1 Supported DDL statements

<table>
<tr><th>Table</th><th>Index</th><th>User-defined Function</th><th>User-defined Stored Procedure</th><th>Trigger</th><th>Sequence</th><th>Schema</th><th>Comment on</th></tr>
<tr><td>CREATE TABLE [PARTITION | AS | SUBPARTITION | LIKE]
ALTER TABLE [PARTITION | SUBPARTITION]
DROP TABLE
RENAME TABLE
TRUNCATE</td><td>CREATE INDEX
ALTER INDEX
DROP INDEX
REINDEX</td><td>CREATE FUNCTION
ALTER FUNCTION
DROP FUNCTION</td><td>CREATE PROCEDURE
ALTER PROCEDURE
DROP PROCEDURE</td><td>CREATE TRIGGER
ALTER TRIGGER
DROP TRIGGER</td><td>CREATE SEQUENCE
ALTER SEQUENCE
DROP SEQUENCE</td><td>CREATE SCHEMA
ALTER SCHEMA
DROP SCHEMA</td><td>COMMENT ON</td></tr>
</table>

Table 4-2 DDL statements supported by M-compatible databases

<table>
<tr><th>Table</th><th>Index</th><th>Sequence</th><th>Schema</th><th>Comment on</th></tr>
<tr><td>ALTER TABLE
CREATE TABLE
DROP TABLE
ALTER TABLE PARTITION
CREATE TABLE PARTITION
TRUNCATE</td><td>ALTER INDEX
CREATE INDEX
DROP INDEX
REINDEX
ALTER TABLE DROP INDEX
ALTER TABLE ADD INDEX</td><td>ALTER SEQUENCE
CREATE SEQUENCE
DROP SEQUENCE</td><td>ALTER SCHEMA
CREATE SCHEMA
DROP SCHEMA
ALTER DATABASE
CREATE DATABASE
DROP DATABASE</td><td>COMMENT ON</td></tr>
</table>

Description

When DML statements are executed in the database, the storage engine generates DML logs for restoration. After decoding the DML logs, the storage engine restores the corresponding DML statements and generates logical logs. For DDL statements, the database does not record logs of original DDL statements. Instead, it records DML logs of system catalogs involved in DDL statements. DDL statements are of various types and complex syntax. It is difficult to support DDL statements in logical replication and decode original DDL statements based on DML logs of these system catalogs. DDL logs are added to record original DDL information. During decoding, original DDL statements can be obtained from DDL logs.

During the execution of a DDL statement, the SQL engine parser parses the syntax and lexicon of the original statement and generates a parsing tree. (Different DDL syntaxes generate different types of parsing trees, which contain all information about the DDL statement.) Then, the executor performs the corresponding operation based on the information to generate and modify the corresponding meta information.

In this section, DDL logs are added to support logical decoding of DDL statements. The content of DDL logs is generated based on the parser result (parsing tree) and executor result, and the logs are generated after the execution is complete.

DDL statements are reversely parsed from the syntax tree. In this way, DDL commands are converted to JSON statements and necessary information is provided to rebuild DDL commands in the target location. Compared with the original DDL command strings, the benefits of parsing DDL statements from the syntax tree include:

1. Each parsed database object has a schema. Therefore, if different search paths are used, there is no ambiguity.
2. Structured JSON statements and formatted output support heterogeneous databases. If you are using different database versions and some DDL syntaxes differ, you need to resolve these differences before applying them.

The output result of reverse parsing is in the normalized format. This result is equivalent to the user input but is not necessarily the same. For example:

Example 1: If the function body does not contain single quotation marks ('), the separator $$ of the function body is parsed as single quotation marks (').

Original SQL statement:

```
CREATE FUNCTION func(a INT) RETURNS INT AS
$$
BEGIN
a:= a+1;
CREATE TABLE test(col1 INT);
INSERT INTO test VALUES(1);
DROP TABLE test;
RETURN a;
END;
$$
LANGUAGE plpgsql;
```

Reverse parsing result:

```
CREATE  FUNCTION public.func ( IN a pg_catalog.int4 ) RETURNS pg_catalog.int4 LANGUAGE plpgsql
VOLATILE CALLED ON NULL INPUT SECURITY INVOKER COST 100 AS '
BEGIN
a:= a+1;
CREATE TABLE test(col1 INT);
INSERT INTO test VALUES(1);
DROP TABLE test;
RETURN a;
```

```
END;
';
```

Example 3: "ALTER INDEX "Alter_Index_Index" REBUILD PARTITION "CA_ADDRESS_SK_index2"" will be reversely parsed as "REINDEX INDEX public."Alter_Index_Index" PARTITION "CA_ADDRESS_SK_index2"".

Example 4: Create or modify a range partitioned table. The START END syntax is decoded and converted into the LESS THAN statement.

```
gaussdb=# CREATE TABLE test_create_table_partition2 (c1 INT, c2 INT)
PARTITION BY RANGE (c2) (
    PARTITION p1 START(1) END(1000) EVERY(200) ,
    PARTITION p2 END(2000),
    PARTITION p3 START(2000) END(2500),
    PARTITION p4 START(2500),
    PARTITION p5 START(3000) END(5000) EVERY(1000)
);
```

Reverse parsing result:

```
gaussdb=# CREATE TABLE test_create_table_partition2 (c1 INT, c2 INT)
PARTITION BY RANGE (c2) (
    PARTITION p1_0 VALUES LESS THAN ('1'), PARTITION p1_1 VALUES LESS THAN ('201'), PARTITION p1_2
VALUES LESS THAN ('401'), PARTITION p1_3 VALUES LESS THAN ('601'), PARTITION p1_4 VALUES LESS
THAN ('801'), PARTITION p1_5 VALUES LESS    THAN ('1000'),
    PARTITION p2 VALUES LESS THAN ('2000'),
    PARTITION p3 VALUES LESS THAN ('2500'),
    PARTITION p4 VALUES LESS THAN ('3000'),
    PARTITION p5_1 VALUES LESS THAN ('4000'),
    PARTITION p5_2 VALUES LESS THAN ('5000')
);
```

Example 5: When adding a column to a table, use IF NOT EXISTS for judgment.

- Original SQL statement:
  ```
  gaussdb=# CREATE TABLE IF NOT EXISTS tb5 (c1 int,c2 int) with (ORIENTATION=ROW,
  STORAGE_TYPE=USTORE);
  gaussdb=# ALTER TABLE IF EXISTS tb5 * ADD COLUMN IF NOT EXISTS c2 char(5) after c1; -- Can be
  decoded. Column c2 of the int type exists in the table and is skipped, and the type of column c2 in
  the reverse parsing result remains unchanged.
  ```
  Reverse parsing result:
  ```
  gaussdb=# ALTER TABLE IF EXISTS public.tb5 ADD COLUMN IF NOT EXISTS c2 pg_catalog.int4 AFTER
  c1;
  ```
- Original SQL statement:
  ```
  gaussdb=# ALTER TABLE IF EXISTS tb5 * ADD COLUMN IF NOT EXISTS c2 char(5) after c1,  ADD
  COLUMN IF NOT EXISTS c3 char(5) after c1; -- Decoded. The type of the new column c3 in the
  reverse parsing result is correct.
  ```
  Reverse parsing result:
  ```
  gaussdb=# ALTER TABLE IF EXISTS public.tb5 ADD COLUMN IF NOT EXISTS c2 pg_catalog.int4 AFTER
  c1, ADD COLUMN IF NOT EXISTS c3 pg_catalog.bpchar(5) AFTER c1;
  ```
- Original SQL statement:
  ```
  gaussdb=# ALTER TABLE IF EXISTS tb5 * ADD COLUMN c2 char(5) after c1,  ADD COLUMN IF NOT
  EXISTS c4 int after c1; -- Not decoded. An error occurs when the statement is executed.
  ```

Example 6: When a database object (DROP TABLE/INDEX/SEQUENCE) is defined, the **IFEXISTS** syntax option is added to the decoding result.

- Original SQL statement:
  ```
  gaussdb=# CREATE TABLE IF NOT EXISTS tb6 (c1 int,c2 int) with (ORIENTATION=ROW,
  STORAGE_TYPE=USTORE);
  gaussdb=#   DROP TABLE tb6;
  ```
  Reverse parsing result:
  ```
  gaussdb=# DROP TABLE IF EXISTS public.tb6 RESTRICT;
  ```

Limitations

- DDL specification restrictions for logical decoding:
  - The logical decoding performance in the DDL-only scenario is about 100 MB/s in the standard environment, and that in DDL/DML hybrid transaction scenario is about 100 MB/s in the standard environment.
  - After this function is enabled (by setting **wal_level** to **logical** and **enable_logical_replication_ddl** to **on**), the performance of DDL statements decreases by less than 15%.
- General decoding restrictions (serial and parallel):
  - DDL operations on local temporary objects cannot be decoded.
  - DDL statement decoding in the FOREIGN TABLE scenario is not supported.
  - The DEFAULT of ALTER TABLE ADD COLUMN does not support stable or volatile functions. The CHECK constraint expression of CREATE TABLE and ALTER TABLE regarding columns does not support stable or volatile functions. If ALTER TABLE has multiple clauses and one of them has the preceding two situations, the entire ALTER TABLE statement is not parsed reversely.

    ```
    gaussdb=# ALTER TABLE tbl_28 ADD COLUMN b1 TIMESTAMP DEFAULT NOW(); -- 's' NOT DEPARSE
    gaussdb=# ALTER TABLE tbl_28 ADD COLUMN b2 INT DEFAULT RANDOM(); -- 'v' NOT DEPARSE
    gaussdb=# ALTER TABLE tbl_28 ADD COLUMN b3 INT DEFAULT ABS(1); -- 'i' DEPARSE
    ```
  - In a distributed system, decoding of DDL operation CREATE MATERIALIZED VIEW is not supported.
  - Decoding of DDL statements CREATE/ALTER/DROP VIEW and COMMENT ON VIEW is not supported.
  - Decoding of DDL statements REINDEX DATABASE/SYSTEM is not supported.
  - Decoding of trigger-related DDL statements on views is not supported.
  - Decoding of DDL statements with the CONCURRENTLY keyword is not supported.
  - If IF NOT EXISTS exists in the statement for creating an object and the object already exists, the statement is not decoded. If IF EXISTS exists in the statement for deleting an object but the object does not exist, the statement is not decoded.
  - The ALTER PACKAGE COMPILE statement is not decoded, but the DDL/DML statements contained in the instantiated content are decoded. If the package does not contain instantiated content involving DDL or DML statements, ALTER PACKAGE COMPILE will be ignored by logical decoding.
  - Only the commercial DDL syntax earlier than this version is supported. The following SQL statements do not support logical decoding:
    - Create a row-store table and set the ILM policy.

      Original SQL statement:

      ```
      gaussdb=# CREATE TABLE IF NOT EXISTS tb3 (c1 int) with
      (storage_type=USTORE,ORIENTATION=ROW) ILM ADD POLICY ROW STORE COMPRESS
      ADVANCED ROW AFTER 7 day OF NO MODIFICATION;
      ```

      Reverse parsing result:

```
gaussdb=# CREATE TABLE IF NOT EXISTS public.tb3 (c1 pg_catalog.int4) WITH
(storage_type = 'ustore', orientation = 'row', compression = 'no') NOCOMPRESS;
```

– Logical decoding does not support DDL/DCL/DML hybrid transactions. In hybrid transactions, DML statements after DDL statements cannot be decoded.

```
-- No reverse parsing is performed. DCL statements are not supported and therefore are not
parsed. DML statements after DCL statements are not parsed.
gaussdb=# BEGIN;
gaussdb=# GAINT ALL PRIVILEGES to u01;
gaussdb=# INSERT INTO test1(col1) values(1);
gaussdb=# COMMIT;

-- Only the first and third SQL statements are reversely parsed.
gaussdb=# BEGIN;
gaussdb=# CREATE TABLE mix_tran_t4(id int);
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# CREATE TABLE mix_tran_t5(id int);
gaussdb=# COMMIT;

-- Only the first and second SQL statements are reversely parsed.
gaussdb=# BEGIN;
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# CREATE TABLE mix_tran_t6(id int);
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# COMMIT;

-- Full reverse parsing
gaussdb=# BEGIN;
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# CREATE TABLE mix_tran_t7(id int);
gaussdb=# CREATE TABLE mix_tran_t8(id int);
gaussdb=# COMMIT;

-- Only the first and third SQL statements are reversely parsed.
gaussdb=# BEGIN;
gaussdb=# CREATE TABLE mix_tran_t7(id int);
gaussdb=# CREATE TYPE compfoo AS (f1 int, f2 text);
gaussdb=# CREATE TABLE mix_tran_t8(id int);
gaussdb=# COMMIT;

-- Full reverse parsing
gaussdb=# BEGIN;
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# COMMIT;

-- Only the first SQL statement is reversely parsed.
gaussdb=# BEGIN;
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# CREATE TYPE compfoo AS (f1 int, f2 text);
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# COMMIT;

-- Only the first and third SQL statements are reversely parsed.
gaussdb=# BEGIN;
gaussdb=# INSERT INTO mix_tran_t4 VALUES(111);
gaussdb=# CREATE TYPE compfoo AS (f1 int, f2 text);
gaussdb=# CREATE TABLE mix_tran_t9(id int);
gaussdb=# COMMIT;
```

– For CREATE TABLE AS SELECT, SELECT INTO, and CREATE TABLE AS statements, only CREATE TABLE statements can be decoded and INSERT statements cannot be decoded.

NOTE

For tables created by using CREATE TABLE AS, the ALTER and DROP statements are still decoded.

Example:

Original SQL statement:

```
CREATE TABLE IF NOT EXISTS tb35_2 (c1 int) with (storage_type=USTORE,ORIENTATION=ROW);
INSERT INTO tb35_2 VALUES (6);
CREATE TABLE tb35_1 with (storage_type=USTORE,ORIENTATION=ROW) AS SELECT * FROM
tb35_2;
```

Reverse parsing result of the last SQL statement:

```
 CREATE  TABLE  public.tb35_1 (c1 pg_catalog.int4) WITH (storage_type = 'ustore', orientation =
'row', compression = 'no') NOCOMPRESS;
```

- When a stored procedure, function, or advanced package is executed, if the stored procedure, function, or advanced package contains a DDL/DML hybrid transaction or the stored procedure, function, or advanced package and other statements in the same transaction form a DDL/DML hybrid transaction, decoding is performed based on the hybrid transaction principle.
- Logical decoding does not support the ledger database feature because the decoding result of the DDL statement for creating a ledger database contains hash columns.
  - Original statement:
    ```
    CREATE SCHEMA blockchain_schema WITH BLOCKCHAIN;
    CREATE TABLE blockchain_schema.blockchain_table(mes int);
    ```
  - Decoding result:
    ```
    CREATE SCHEMA blockchain_schema WITH BLOCKCHAIN;
    CREATE TABLE blockchain_schema.blockchain_table (mes pg_catalog.int4, hash_a1d895
    pg_catalog.hash16);  -- The statement cannot be replayed on the target end.
    ```
    You need to manually disable the tamper-proof attribute of blockchain_schema on the target end before replaying data. In this case, blockchain_table on the target end is equivalent to an ordinary table. Then, you can run the DML command again to replay data.

    SQL commands:
    ```
    ALTER SCHEMA blockchain_schema WITHOUT BLOCKCHAIN;
    CREATE TABLE blockchain_schema.blockchain_table (mes pg_catalog.int4, hash_a1d895
    pg_catalog.hash16);
    ```

- In MySQL-compatible mode, the ALTER SCHEMA schema_name WITHOUT/WITH BLOCKCHAIN syntax cannot be parsed.
- DDL-specific constraints for serial logical decoding:
  - The sql_decoding plug-in does not support DDL statements in JSON format.

Decoding Format

- JSON format

  When a DDL statement is input, the SQL engine parser parses the syntax and lexicon of the DDL statement and generates a parsing tree. The parsing tree contains all DDL information and the executor modifies system metadata based on the parsing tree content. After the execution is complete, you can obtain the search path of the DDL objects. After the executor is successfully executed, this feature reversely parses the parsing tree information and executor result to restore all information about the original DDL statement. In this way, the entire DDL statement can be parsed and a DDL statement in JSON format is output to adapt to heterogeneous databases.

Upon lexical and syntax analysis on the CREATE TABLE statement, the corresponding CreateStmt parsing tree node is obtained, containing table information, column information, distribution information (DistributeBy structure), and partition information (PartitionState structure). After reverse parsing, the result is output in JSON format as follows:

```
{"JDDL":{"fmt":"CREATE %{persistence}s TABLE %{if_not_exists}s %{identity}D %{table_elements}s %{with_clause}s %{compression}s","identity":{"object_name":"test_create_table_a","schema_name":"public"},"compression":"NOCOMPRESS","persistence":"","with_clause":{"fmt":"WITH (%{with:, }s)","with":[{"fmt":"%{label}s = %{value}L","label":{"fmt":"%{label}I","label":"orientation"},"value":"row"},{"fmt":"%{label}s = %{value}L","label":{"fmt":"%{label}I","label":"compression"},"value":"no"}]},"if_not_exists":"","table_elements":{"fmt":"(%{elements:, }s)","elements":[{"fmt":"%{name}I %{column_type}T","name":"a","column_type":{"typmod":"","typarray":false,"type_name":"int4","schema_name":"pg_catalog"}}]}}}
```

The output JSON string contains the search path of the object. In the string, the **identity** key indicates that the schema is **public** and the table name is **test_create_table_a**. **%{persistence}s** corresponds to the following field in the SQL statement (This SQL statement does not contain this field and therefore is empty.):

```
[ [ GLOBAL | LOCAL ] [ TEMPORARY | TEMP ] | UNLOGGED ]
```

%{if_not_exists}s corresponds to a field in the SQL statement. (This SQL statement does not contain this field and therefore is empty.)

```
[ IF NOT EXISTS ]
```

%{identity}D corresponds to the following field in the SQL statement:

```
table_name
```

%{table_elements}s corresponds to the following field in the SQL statement:

```
 (column_name data_type)
```

%{with_clause}s corresponds to the following field in the SQL statement:

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
```

%{compression}s corresponds to the following field in the SQL statement:

```
[ COMPRESS | NOCOMPRESS ]
```

- Format specified by **decode-style**

  The output format is controlled by the **decode-style** parameter. For example, when **decode-style** is set to **'j'**, the output format is as follows:

  ```
  {"TDDL":"CREATE  TABLE  public.test_create_table_a (a pg_catalog.int4) WITH (orientation = 'row', compression = 'no') NOCOMPRESS"}
  ```

  The statement also contains the schema name.

API Design

- New control parameters
  a. Logical decoding control parameters are added to control the DDL statement reverse parsing process and output format. To enable them, you can use the JDBC or pg_logical_slot_peek_changes API.
     - **enable-ddl-decoding**: The default value is **false**, indicating that logical decoding of DDL statements is disabled. The value **true** indicates that logical decoding of DDL statements is enabled.

- **enable-ddl-json-format**: The default value is **false**, indicating that the DDL statement reverse parsing result is output in text format. The value **true** indicates that the DDL statement reverse parsing result is output in JSON format.

b. A GUC parameter is added.

- **enable_logical_replication_ddl**: The default value is **ON**. If the value is **ON**, logical replication of DDL statements is supported. Otherwise, logical replication of DDL statements is not supported. The DDL statement execution result is reversely parsed and WALs of the DDL statements are generated only when this parameter is set to **ON**. Otherwise, no reverse parsing is performed and no WAL is generated.

  You can check the operation logs of **enable_logical_replication_ddl** and determine whether logical decoding of DDL statements is not supported because the parameter is modified by users.

- New logs

  The **xl_logical_ddl_message** log is added for DDL statements. The log type is RM_LOGICALDDLMSG_ID. The definition is as follows:

Name	Type	Description
db_id	OID	Database ID
rel_id	OID	Table ID
csn	CommitSeqNo	CSN-based snapshot
cid	CommandId	Command ID
tag_type	NodeTag	DDL type
message_size	Size	Length of the log content
filter_message_size	Size	Length of log information filtered by whitelist
message	char *	DDL content

Procedure

Step 1 For the logical decoding feature, set the **wal_level** GUC parameter to **logical** in advance. This parameter takes effect only after the system is restarted.

```
gs_guc set -Z datanode -D $node_dir -c "wal_level = logical"
```

In the preceding command, *$node_dir* indicates the database node path. Change it based on the actual situation.

Step 2 Log in to the primary node of GaussDB as a user with the REPLICATION permission and run the following command to connect to the database:

```
gsql -U user1 -W password -d db1 -p 16000 -r
```


In the preceding command, *user1* indicates the username, *password* indicates the user password, *db1* indicates the name of the database to be connected, and **16000** indicates the database port number. You can replace them as required.

Step 3 Create a logical replication slot named **slot1**.

```
gaussdb=# SELECT * FROM pg_create_logical_replication_slot('slot1', 'mppdb_decoding');
 slotname | xlog_position
----------+---------------
 slot1    | 0/3764C788
(1 row)
```

Step 4 Create a package in the database.

```
gaussdb=# CREATE OR REPLACE PACKAGE ldp_pkg1 IS
    var1 int:=1;  -- Public variable
    var2 int:=2;
    PROCEDURE testpro1(var3 int);  -- Public stored procedure, which can be called by external systems.
END ldp_pkg1;
/
```

Step 5 Read the decoding result of replication slot 1. You can the JDBC or pg_logical_slot_peek_changes API to update the replication slot number.

NOTE

- For details about the logical decoding options, see **Logical Decoding Options** and new control parameters.
- In parallel decoding, you can change the value of the **decode_style** parameter in the JDBC API to determine the decoding format.

  Configure **decode-style** to specify the decoding format. The value can be **'j'**, **'t'**, or **'b'** of the char type, indicating the JSON, TEXT, or binary format, respectively.

```
gaussdb=# SELECT data FROM pg_logical_slot_peek_changes('slot1', NULL, NULL, 'enable-ddl-decoding', 'true', 'enable-ddl-json-format', 'false') WHERE data not like 'BEGIN%' AND data not like 'COMMIT%' AND data not like '%dbe_pldeveloper.gs_source%';

                                                                   data

------------------------------------------------------------------------------------------------------------------------
-------------------------------------------- {"TDDL":"CREATE OR REPLACE PACKAGE public.ldp_pkg1 AUTHID CURRENT_USER IS var1 int:=1; -- Public variable\n    var2 int:=2;\n
PROCEDURE testpro1(var3 int);  -- Public stored procedure, which can be called externally.\nEND  ldp_pkg1;\n /"}
(1 row)
```

Step 6 Delete the logical replication slot **slot1** and package **ldp_pkg1**.

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

(1 row)

gaussdb=# DROP PACKAGE ldp_pkg1;
NOTICE:  drop cascades to function public.testpro1(integer)
DROP PACKAGE
```

----End

4.1.6 Data Restoration by Logical Decoding

Logical decoding can restore DML data on a single node. Specifically, it restores online or archived WALs.

- Online data: To restore online DML data, you can use the pg_logical_get_area_changes function. For details, see the usage of the pg_logical_get_area_changes function.
- Archived data: OM_Agent uses the pg_logical_get_area_changes function to restore archived logs on OBS. OM_Agent downloads archived logs from OBS to the corresponding node and creates a folder named with the first eight digits of the WAL file name. After the decoding is complete, the restored data is stored in a file and the file path is returned.

Restrictions

1. The log level of parsed WALs is **logical**.
2. The replication identity **FULL** must be specified for copying a data table; otherwise, the copied rows on which UPDATE or DELETE is performed are not complete.
3. After the decoding starts, do not perform the VACUUM FULL operation on the service table for which data modifications are recorded using WALs; otherwise, DML operations before the VACUUM FULL operation cannot be restored.
4. The size of each WAL file cannot exceed 500 MB.
5. Xlogs generated before scale-out cannot be restored.
6. A result file is generated for each shard in the cluster. Files are not merged.
7. Only archived data can be restored. To restore the data, ensure that the archive function is enabled. If data has not been archived, it cannot be restored using this API.
8. Before downloading files, the OM Agent checks whether the used local space is greater than 80% of the total space. If the used space exceeds 80% of the total space, the following error is reported, indicating that extra space is required for storing decoded files: "no enough space left on device, available space must be greater than 20%."
9. If the download or decoding fails, the downloaded WAL files are cleared. If the clearing fails, the program is not forcibly ended and only the error information is recorded in DN logs.
10. The start time cannot exceed the maximum time in the gs_txn_lsn_time system catalog, and the end time cannot exceed the minimum time in the gs_txn_lsn_time system catalog. Otherwise, an error is reported.
11. The data retrieval API cannot be concurrently called on the same node.
12. If a node is replaced, the data of the node before the replacement cannot be restored.
13. Only data restoration through direct connection to DNs is supported.
14. After upgrading from an earlier version to one that supports baseline initialization, or when the current version is 505.2.1 SPC100 or later, baseline initialization must be performed before data restoration can be used. The system can restore data only from Xlog files generated after the baseline. Xlog files created before the upgrade cannot be parsed. You can call the gs_logical_dictionary_baseline() function on each DN to initialize the logical data dictionary baseline.
15. By default, data generated within one year can be restored and those generated earlier will be automatically deleted. To restore data generated

more than one year ago, change the value of the GUC parameter **logical_replication_dictionary_retention_time** before data is cleared.

5 Partitioned Table

This section describes how to optimize query and perform O&M management on partitioned tables in large data scenarios and systematically explains the usage of partitioned tables, covering the semantics, principles, constraints, and limitations.

5.1 Introduction to Table Partitioning

5.1.1 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.

5.1.2 Table Partitioning

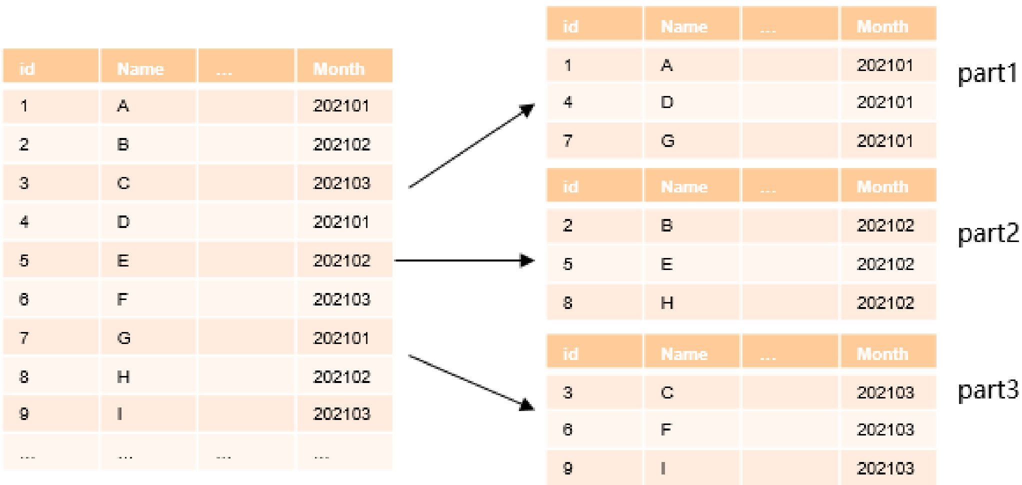
Table partitioning logically divides a large table or index into smaller and easier-to-manage logical units (partitions), minimizing the impact on table query and modification statements. Users can quickly locate a partition where data is located by using a partition key. In this way, users do not need 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:

- Improved query efficiency in large data scenarios: Data in a table is logically partitioned by partition key. During query, only the subset of the related partition is accessed, instead of the entire table. This partition pruning technology can significantly improve query performance and provide performance gains in orders of magnitude.
- Reduced impact caused by concurrent O&M and query operations: Partitioned tables can significantly reduce the mutual impact of DML and DDL statements in concurrent scenarios. This advantage is particularly obvious in scenarios with a large amount of data and partitioning by time. For example, the import of new data partitions and real-time point query operations, as well as O&M operations such as data cleaning and partition merging of old data partitions, can be performed independently without interfering with each other.
- Flexible data O&M management in large data scenarios: Partitioned tables physically isolate data across different partitions. Each partition can be independently set with physical attributes, such as enabling or disabling compression, physical storage settings, and tablespace. In addition, partitioned tables support partition-level data management operations, such as data loading, index creation and rebuild, backup, and restoration, without the need to operate the entire table, greatly reducing the operation time.

5.1.3 Optimization Data Query by Partition

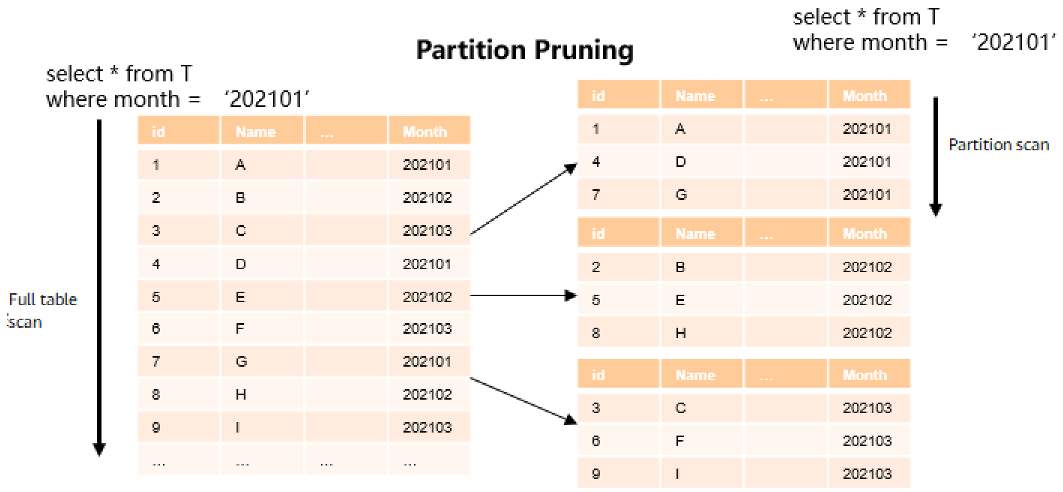
Partitioned tables have advantages in data query when predicate queries are performed on partition keys. Take a table with the month as the partition key as an example, as shown in **Figure 5-1**.

Figure 5-1 Example of a partitioned table



If an ordinary table design is used, a full table scan is required for query. If the table is redesigned with the month as the partition key, the full table scan is optimized to a partition scan. When the table contains a large amount of data and involves a long period, the performance can be significantly improved by reducing the amount of data to be scanned, as shown in **Figure 5-2**.

Figure 5-2 Example of partition pruning



5.1.4 Data Partition O&M Management

Data lifecycle management (DLM) is a set of management processes and policies that run through the entire data lifecycle. One of its core tasks is to match the most suitable and cost-effective storage media for data at different stages. New data that is frequently accessed is stored in the storage layer with fast read/write speed and high availability, while old data that is infrequently accessed is migrated to the storage layer with lower cost and slightly weaker performance. Given that old data is updated infrequently, it is more reasonable to compress it and convert it to read-only.

Partitioned tables create ideal conditions for the implementation of the DLM solution. By using different tablespaces for different partitions, the DLM solution ensures usability and optimizes cost throughout the data lifecycle. The underlying configuration is completed by database O&M personnel on the server side. End users are unaware of the configuration and can still perform query operations on tables based on common logic. In addition, each partition supports independent O&M operations such as backup, restoration, and index rebuild. Different subsets of data sets can be managed separately to meet differentiated requirements of users' service scenarios.

5.2 Introduction to Partitioned Tables

A partitioned table is a table whose data is logically divided into partitions based on partition keys and partitioning policies. Partitioning is essentially a horizontal partitioning strategy. 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 divides tables, indexes, and index-organized tables into smaller units for refined management and access.

GaussDB provides various partitioning policies and extensions to meet the requirements of different service scenarios. Partitioning policies are completely implemented by the database, and users are unaware of the implementation. Therefore, smooth migration can be performed after the partitioning policies are implemented without changing applications. This section describes GaussDB partitioned tables from the following aspects:

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

5.2.1 Basic Concepts

5.2.1.1 Partitioned Table

A partitioned table is a table object on which users perform operations. It supports the SELECT, INSERT, UPDATE, and DELETE operations of the data manipulation language (DML). Generally, the PARTITION BY clause is explicitly used in the table creation DDL statement to define a partitioned table. After a partitioned table is created, a record is added to the pg_class table. The value of the **parttype** field is **'p'** (indicating level-1 partition) or **''** (indicating level-2 partition), indicating that the table corresponding to the record is a partitioned table.

Note that a partitioned table is a logical concept, and no data is really stored in the corresponding physical table.

Example: **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;
```

5.2.1.2 Partition

Partitions are tables that actually store data in a partitioned table. The corresponding entries are usually stored in pg_partition. The **parentid** column of each partition is associated with the OIDs of their partitioned tables in the pg_class table as a foreign key.

Example: **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;
```

5.2.1.3 Partition Key

A partition key consists of one or more columns. The partition key value and the corresponding partitioning strategy 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.

5.2.2 Partitioning Policy

Partitioning policies describe the mapping between data and partition routes in a partitioned table. The PARTITION BY syntax in the DDL statement is used to specify the partitioning policies during table creation.

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) (...)
```

5.2.2.1 Range Partitioning

Range partitioning maps data to partitions based on the partition key value range of each partition. Range partitioning is a common partitioning type in actual production systems. It is usually used in scenarios where data is described by time (date and 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')
);

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, which is used to specify the upper limit of the partition. When the partition key value of the data is greater than or equal to the upper limit of the current partition, the data will be added to the next partition. Except the first partition, all other partitions have a lower limit implicitly specified by the VALUES LESS clause of the previous partition.

You can specify the **MAXVALUE** keyword to define the highest partition. **MAXVALUE** indicates a virtual infinite value. In terms of sorting rules, its priority is higher than that of any other possible value of the partition key, including null values.

– 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)

gaussdb=# DROP TABLE range_sales_with_multiple_keys;
```

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

i. The comparison starts from the first column.

ii. 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.

iii. 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.

iv. 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 a tablespace.
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)

-- Drop the table and schema.
gaussdb=#  DROP TABLE tpcds.startend_pt;
DROP TABLE
gaussdb=# DROP SCHEMA tpcds;
DROP SCHEMA
```

5.2.2.2 Hash Partitioning

Hash partitioning is a method of mapping data to partitions based on the built-in hash algorithm of GaussDB and the partition key. If the partition key value range has no data skew, the hash algorithm can evenly distribute data rows among partitions, ensuring that the sizes of partitions are roughly the same. This is an ideal method for achieving even data distribution 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 there is no obvious partition key available for partitioning. 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
);
-- Drop the table.
gaussdb=# DROP TABLE bmsql_order_line;
```

In the preceding example, the **ol_d_id** column of the **bmsql_order_line** table serves as the basis for partitioning. The **ol_d_id** column is an identifier attribute column. It does not have the time characteristics and cannot distinguish data in a specific dimension. In this case, using the hash partitioning policy to partition the table is an ideal choice.

Compared with other partitioning types, hash partitioning requires only the partition key and the number of partitions to be specified. Before creating a hash partition, ensure that the partition key does not have serious data skew (that is, one or more values are highly repeated). Hash partitioning can ensure that data in each partition is evenly distributed, greatly improving the usability of partitioned tables.

5.2.2.3 List Partitioning

List partitioning allows you to specify a list of discrete values for the partition key in the description of each partition to control how data rows are mapped to partitions. The advantage of this partitioning method is that it can partition data in the form of enumerated partition values, which is especially suitable for grouping and organizing unordered and unrelated data sets.

When processing partition key values, if some values are not defined in the list, you can use the DEFAULT partition to store these data. In this way, no error is generated for rows that cannot be mapped to any other partition. 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)
);
```

```
-- Drop the table.
gaussdb=# DROP TABLE bmsql_order_line;
```

The preceding example is similar to the hash partitioning example provided earlier. The table is partitioned based on the **ol_d_id** column. However, in list partitioning, the possible value range of **ol_d_id** is directly limited. Data that is not in the list enters the **p3** partition (DEFAULT partition).

Compared with hash partitioning, list partitioning has stronger controllability over partition keys and can accurately store target data in the expected partition. However, if there are a large number of values in the list, the partition definition becomes cumbersome. In this case, hash partitioning is recommended.

In general, list partitioning and hash partitioning are often used to group and organize unordered and unrelated data sets.

NOTICE

List partitioning has a clear limit on the number of partition keys. A partition key can contain a maximum of 16 columns. Moreover, the rules for processing null values in the enumerated value list during subpartitioning are different when the number of partition-key columns is different. When a single-column partition key is defined, no null values are allowed in the enumerated value list. When a multi-column partition key is defined, null values are allowed in the enumerated value list.

5.2.2.4 Impact of Partitioned Tables on Import Performance

In GaussDB, compared with a non-partitioned table, a partitioned table has extra overhead of partition routing in a data insertion process.

The data insertion overhead in the partitioned table scenario consists of two parts: (1) Heap-insert: This part addresses the problem how tuples are stored in the corresponding heap table. This operation is common in ordinary tables and partitioned tables. (2) Partition-routing: This part addresses the partition routing problem. That is, tuples need to be inserted into the corresponding partitioned table (partRel).

Therefore, data insertion optimization focuses on the following aspects:

- Heap-insert :
  - The operator noise floor is optimized.
  - Heap data insertion is optimized.
  - Index insertion build (with indexes) is optimized.
- Partition-routing:
  - The logic of the routing search algorithm is optimized.
  - 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 significantly reflected when a single INSERT statement involving a large amount of data is executed. In the UPDATE scenario, the operation logic is more complex. You need to find the tuple to be updated, perform the DELETE operation to remove the tuple, and then perform the INSERT operation to insert a new tuple. Compared with the scenario where a single INSERT statement is used to insert data, the UPDATE scenario has more steps and more complex processes. Therefore, the partition routing performance in the UPDATE scenario is not as good as that in the scenario where a single INSERT statement is used.

Table 5-1 shows the routing algorithm logic of different partitioning types.

Table 5-1 Routing algorithm logic

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

NOTICE

The core processing logic of partition routing is to calculate the partition to which the imported data tuple belongs based on the partition key of the imported data tuple. This is an extra overhead of a partitioned table compared with a non-partitioned table. The impact of the overhead on the final data import performance depends on the CPU processing capability of the server, table width, and actual disk and memory capacity.

Generally, the following estimation can be roughly performed:

- 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 data import performance decreases by about 20%.

The x86 and Arm servers have different performance in importing partitioned tables because partition routing is a memory computing enhancement scenario. The single-core instruction processing capability of the mainstream x86 CPU is slightly stronger than that of the Arm CPU.

5.2.3 Basic Usage of Partitions

5.2.3.1 Creating Partitioned Tables

Creating Partitioned Tables

SQL has powerful and flexible functions, and its syntax tree is complex. This is also true for partitioned tables. Creating a partitioned table can be regarded as adding table partition attributes to a non-partitioned table. Therefore, the syntax of partitioned tables can be regarded as an extension of the CREATE TABLE statement of non-partitioned tables. Specifically, the PARTITION BY clause is added, and the following three core elements related to partitioning are specified:

- **partType**: describes the partitioning policy used by a partitioned table. The partitioning policy can be range partitioning, list partitioning, or hash partitioning.
- **partKey**: determines the partition key of a partitioned table. Currently, range partitioning and list partitioning support multiple columns (no more than 16 columns) as the partition key, while hash partitioning supports only single-column partitioning.
- **partExpr**: describes the partitioning mode of a partitioned table, that is, the mapping between key values and partitions.

The three important elements are reflected in the PARTITION BY clause of the table creation statement. The specific format is PARTITION BY partType (partKey) (partExpr[,partExpr]…). The basic syntax framework is as follows:

```
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) (
    { 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 ];
```

The usage of partitioned tables has the following limitations:

- Range partitioning and list partitioning support a maximum of 16 partition keys, while hash partitioning supports only one partition key.
- The partition key value cannot be null except for hash partitioning. Otherwise, the DML statement reports an error. However, null values can be inserted when the MAXVALUE partition is defined for a range partitioned table or the DEFAULT partition is defined for a list partitioned table.

- The maximum number of partitions is 1,048,575, which can meet the requirements of most service scenarios. As the number of partitions increases, the number of files in the system increases accordingly, affecting system performance. Therefore, it is not recommended that a single table contain more than 200 partitions.

Modifying Partition Attributes

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

- ADD PARTITION
- DROP PARTITION
- TRUNCATE PARTITION
- SPLIT PARTITION
- MERGE PARTITION
- MOVE PARTITION
- EXCHANGE PARTITION
- RENAME PARTITION

The preceding partition attribute change statements are extensions of the ALTER TABLE statement for ordinary tables. The usage of the statements in partitioned tables is similar to that in ordinary tables. The basic syntax framework for changing partition attributes is as follows:

```
/* 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 "SQL Reference > SQL Syntax > ALTER TABLE PARTITION" in *Developer Guide*.

5.2.3.2 Using and Managing Partitioned Tables

Partitioned tables support most functions related to non-partitioned tables. For details, see the syntax related to ordinary tables in *Developer Guide*.

In addition, partitioned tables support many partition-level operation commands, including partition-level DQL/DML operations (such as SELECT, INSERT, UPDATE, DELETE, UPSERT and MERGE INTO), partition-level DDL operations (such as ADD, DROP, TRUNCATE, EXCHANGE, SPLIT, MERGE, MOVE and RENAME), partition-level VACUUM/ANALYZE, and various partitioned indexes. For details about how to use related commands, see **DQL/DML Operations on a Partitioned Table**, **Partitioned Indexes**, **Partitioned Table O&M Management**, and the chapter corresponding to each syntax and command in *Developer Guide*.

A partition-level operation command is generally performed by specifying a partition name or a partition value. For example, the syntax of a command may be as follows:

```
sql_action [ t_name ] { PARTITION | SUBPARTITION } { p_name | (p_name) };
sql_action [ t_name ] { PARTITION | SUBPARTITION } FOR (p_value);
```

You can specify the partition name **p_name** or partition value **p_value** to perform operations on a specific partition. In this case, services apply only to the target partition and do not affect other partitions. If you specify **p_name** to execute a service, the database matches the partition corresponding to **p_name**. If the partition does not exist, an exception is reported. If you specify **p_value** to execute a service, the database matches the partition to which **p_value** belongs.

For example, the following partitioned table is defined:

```
gaussdb=# CREATE TABLE list_01
(
    id   INT,
    role VARCHAR(100),
    data VARCHAR(100)
)
PARTITION BY LIST (id)
(
    PARTITION p_list_1 VALUES(0,1,2,3,4),
    PARTITION p_list_2 VALUES(5,6,7,8,9),
    PARTITION p_list_3 VALUES(DEFAULT)
);

-- Drop the table.
gaussdb=# DROP TABLE list_01;
```

In terms of specifying a partition, PARTITION p_list_1 is equivalent to PARTITION FOR (4), and they actually point to the same partition. Similarly, PARTITION p_list_3 is equivalent to PARTITION FOR (12), and they also represent the same partition.

5.2.3.3 DQL/DML Operations on a Partitioned Table

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

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

The DQL/DML statements for specifying partitions are as follows:

- SELECT
- INSERT
- UPDATE
- DELETE
- UPSERT
- MERGE INTO

The following is an example of a DQL/DML statement 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;
-- Insert 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;
```

5.3 Partitioned Table Query Optimization

NOTE

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

5.3.1 Partition Pruning

Partition pruning is a technology provided by GaussDB to optimize partitioned table queries. The database SQL engine scans only some specific partitions based on query conditions. This optimization action is automatically triggered. When the query conditions of a partitioned table meet the pruning scenario, partition pruning is automatically started.

Partition pruning can be classified into static pruning and dynamic pruning based on the pruning phase.

- Static pruning occurs in the optimizer phase. Before the execution plan is generated, the database has obtained the information about the partitions to be accessed.
- Dynamic pruning is implemented in the executor phase (when the execution starts or is in progress). When an execution plan is generated, the database does not know the partitions to be accessed and only determines that the partition pruning conditions are met. The specific pruning details are determined by the executor.

Note that partition pruning is triggered only by partitioned table page scan and local index scan. Global indexes do not have the concept of partitioning and do not need to be pruned.

5.3.1.1 Static Partition Pruning

For partitioned table query statements with constants in partition keys in the search conditions, the search conditions 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 conditions must contain at least one partition key. For a partitioned table with a multi-column partition key, the search conditions can contain any column of the partition key.

Static pruning is supported in the following scenarios:

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

NOTICE

- 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:
  - 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)
```

– 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)
```

– 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:

  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;
```

5.3.1.2 Dynamic Partition Pruning

If the search condition of a partitioned table query statement contains variables, the optimizer cannot obtain the bound parameters entered by users. Therefore, the optimizer can parse only the search conditions of operators such as index scan, bitmap index scan, and index-only scan. In the executor phase, partition filtering is complete only after the bound parameters are obtained.

The search conditions must contain at least one partition key. For a partitioned table with a multi-column partition key, the search conditions can contain any column of the partition key.

Currently, dynamic partition pruning supports only the parse-bind-execute (PBE) and parameterized path scenarios.

5.3.1.2.1 Dynamic PBE Pruning

Dynamic PBE pruning is supported in the following scenarios:

- Supported partitioning types: range partitioning, hash partitioning, and list partitioning.
- Supported expression types: comparison expression (<, <=, =, >=, >), logical expression, and array expression.
- Supported implicit conversions and functions: some type conversions and the IMMUTABLE function.

NOTICE

- 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 the table to regenerate a Gplan.
- Dynamic PBE pruning is based on a Gplan. 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 a Cplan.

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

```
gaussdb=#
-- Create a partitioned table.
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)
```

– 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)
```

– 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:
  - 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)
```

  - 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)
```

  - 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;
```

5.3.1.2.2 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.

NOTICE

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:
  - 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..3 (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..3 (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)
```

– 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 = (lengthb(t3.c1)))
         ->  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, (lengthb(t3.c1))
               ->  Streaming(type: REDISTRIBUTE)
                     Output: t3.c1, t3.c2, (lengthb(t3.c1))
                     Distribute Key: (lengthb(t3.c1))
                     Spawn on: All datanodes
                     Consumer Nodes: All datanodes
                     ->  Seq Scan on public.t3
                           Output: t3.c1, t3.c2, lengthb(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)

-- Drop a table.
gaussdb=# DROP TABLE t1;
gaussdb=# DROP TABLE t2;
gaussdb=# DROP TABLE t3;
```

5.3.2 Partitioned Indexes

There are three types of indexes on a partitioned table:

- Global non-partitioned index
- Global partitioned index
- Local partitioned index

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

Figure 5-3 Global non-partitioned index

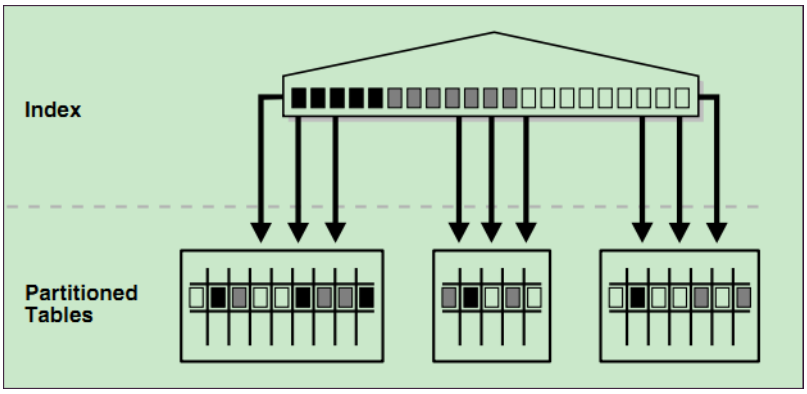


Figure 5-4 Global partitioned index

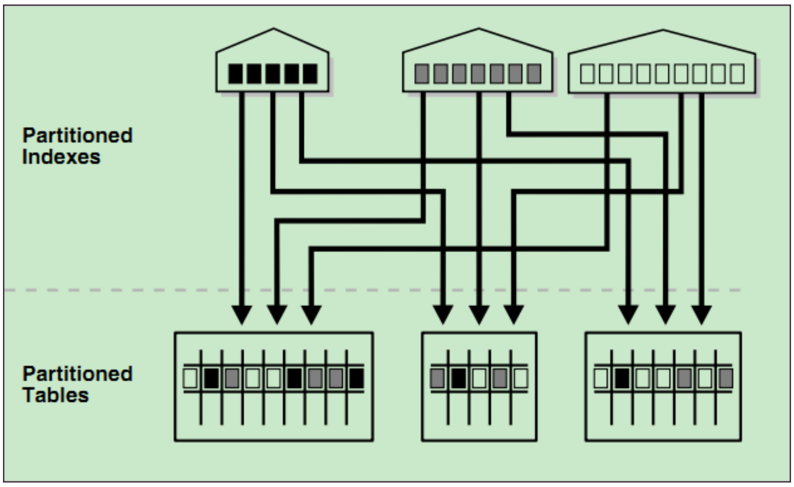
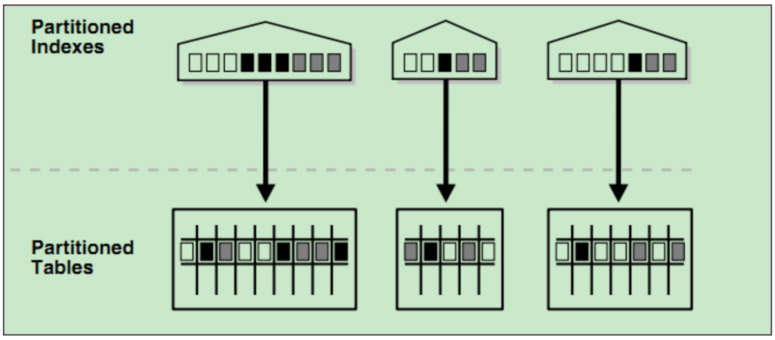


Figure 5-5 Local partitioned index



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.
- When creating an index, you need to determine the index type based on the settings of UNIQUE and PRIMARY KEY constraints. If the constraint keys cover all partition keys, create a local index. Otherwise, create a global index.
- When creating a local index, you can use the FOR { partition_name | (partition_value [,...]) } clause to specify a single partition for index creation. The index created in this way is valid only in the specified partition and does not take effect in other partitions. In addition, when a partition is added, the index is not automatically created for the new partition.
- Currently, the query path of sparsely partitioned indexes can be generated only in the plan that statically prunes data to a single partition.

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 creation is successful:
```
CREATE INDEX
```
  - Create the local index **tpcds_web_returns_p2_index2** with the specified partition name.
```
gaussdb=# CREATE TABLESPACE example2 LOCATION '/home/omm/example2';
gaussdb=# CREATE TABLESPACE example3 LOCATION '/home/omm/example3';
gaussdb=# CREATE TABLESPACE example4 LOCATION '/home/omm/example4';

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


- Create a sparsely partitioned index for a partition.

  Specify the partition name.
  ```
  gaussdb=# CREATE INDEX tpcds_web_returns_for_p1 ON web_returns_p2 (ca_address_id)
  LOCAL(partition ind_part for p1);
  ```
  Specify the value of a partition key.
  ```
  gaussdb=# CREATE INDEX tpcds_web_returns_for_p2 ON web_returns_p2 (ca_address_id)
  LOCAL(partition ind_part for (5000));
  ```
  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
  ```
- Drop a table.
  ```
  gaussdb=# DROP TABLE web_returns_p2;
  ```

5.3.3 Collecting Statistics on Partitioned Tables

For partitioned tables, you can collect partition-level statistics, which can be queried in the pg_partition and pg_statistic system catalogs and in the pg_stats and pg_ext_stats views.

Partition-level statistics apply to the scenario where the scan range is reduced to a single partition after static pruning is performed on a partitioned table. The following statistics can be collected: number of pages and tuples at the partition level, single-column statistics, multi-column statistics, and expression index statistics.

You can collect statistics on partitioned tables in either of the following ways:

- Collect statistics in cascading mode.
- Collect statistics on a specified partition.

5.3.3.1 Collecting Statistics in Cascading Mode

When ANALYZE | ANALYSE is used to analyze a partitioned table, the system automatically collects all partition-level statistics that comply with semantics in the partitioned table based on the specified or default **PARTITION_MODE**. For details about **PARTITION_MODE**, see the **PARTITION_MODE** parameter in "SQL Reference > SQL Syntax > ANALYZE | ANALYSE" in *Developer Guide*.

NOTICE

- When statistics about replication and hash bucket partitioned tables are collected in cascading mode and **PARTITION_MODE** is set to **ALL**, the behavior is converted to the **ALL COMPLETE** mode.
- Cascading collection of partition-level statistics does not support the scenario where the value of **default_statistics_target** is a negative number.

Examples

- Create a partitioned table and insert data.
```
gaussdb=# CREATE TABLE t1_range_int
(
    c1 INT,
    c2 INT,
    c3 INT,
    c4 INT
)
PARTITION BY RANGE(c1)
(
    PARTITION range_p00 VALUES LESS THAN(10),
    PARTITION range_p01 VALUES LESS THAN(20),
    PARTITION range_p02 VALUES LESS THAN(30),
    PARTITION range_p03 VALUES LESS THAN(40),
    PARTITION range_p04 VALUES LESS THAN(50)
);
gaussdb=# INSERT INTO t1_range_int SELECT v,v,v,v FROM generate_series(0, 49) AS v;
```
- Collect statistics in cascading mode.
```
gaussdb=# ANALYZE t1_range_int WITH ALL;
```
- View partition-level statistics.
```
gaussdb=# SELECT relname, parttype, relpages, reltuples FROM pg_partition WHERE
parentid=(SELECT oid FROM pg_class WHERE relname='t1_range_int') ORDER BY relname;
   relname    | parttype | relpages | reltuples
--------------+----------+----------+-----------
 range_p00    | p        |        4 |         9
 range_p01    | p        |        7 |        17
 range_p02    | p        |        6 |        13
 range_p03    | p        |        2 |         5
 range_p04    | p        |        4 |         9
 t1_range_int | r        |        0 |         0
(6 rows)

gaussdb=# SELECT
schemaname,tablename,partitionname,subpartitionname,attname,inherited,null_frac,avg_width,n_disti
nct,n_dndistinct,most_common_vals,most_common_freqs,histogram_bounds FROM pg_stats WHERE
tablename='t1_range_int' ORDER BY tablename, partitionname, attname;
```

```
 schemaname |  tablename   | partitionname | subpartitionname | attname | inherited | null_frac |
avg_width | n_distinct | n_dndistinct | most_common_vals | most_common_freqs
|                                          histogram_bounds

------------+--------------+---------------+------------------+---------+-----------+-----------+-----------
+------------+--------------+------------------+-------------------
+--------------------------------------------------------------------------------------------
---------------------------------------------
 public     | t1_range_int | range_p00     |                  | c1      | f         |         0 |         4 |         -1 |
-1 |              |                  | {0,1,2,3,4,5,6,7,8,9}
 public     | t1_range_int | range_p00     |                  | c2      | f         |         0 |         4 |         -1 |
-1 |              |                  | {0,1,2,3,4,5,6,7,8,9}
 public     | t1_range_int | range_p00     |                  | c3      | f         |         0 |         4 |         -1 |
-1 |              |                  | {0,1,2,3,4,5,6,7,8,9}
 public     | t1_range_int | range_p00     |                  | c4      | f         |         0 |         4 |         -1 |
-1 |              |                  | {0,1,2,3,4,5,6,7,8,9}
 public     | t1_range_int | range_p01     |                  | c1      | f         |         0 |         4 |         -1 |
-1 |              |                  | {10,11,12,13,14,15,16,17,18,19}
 public     | t1_range_int | range_p01     |                  | c2      | f         |         0 |         4 |         -1 |
-1 |              |                  | {10,11,12,13,14,15,16,17,18,19}
 public     | t1_range_int | range_p01     |                  | c3      | f         |         0 |         4 |         -1 |
-1 |              |                  | {10,11,12,13,14,15,16,17,18,19}
 public     | t1_range_int | range_p01     |                  | c4      | f         |         0 |         4 |         -1 |
-1 |              |                  | {10,11,12,13,14,15,16,17,18,19}
 public     | t1_range_int | range_p02     |                  | c1      | f         |         0 |         4 |         -1 |
-1 |              |                  | {20,21,22,23,24,25,26,27,28,29}
 public     | t1_range_int | range_p02     |                  | c2      | f         |         0 |         4 |         -1 |
-1 |              |                  | {20,21,22,23,24,25,26,27,28,29}
 public     | t1_range_int | range_p02     |                  | c3      | f         |         0 |         4 |         -1 |
-1 |              |                  | {20,21,22,23,24,25,26,27,28,29}
 public     | t1_range_int | range_p02     |                  | c4      | f         |         0 |         4 |         -1 |
-1 |              |                  | {20,21,22,23,24,25,26,27,28,29}
 public     | t1_range_int | range_p03     |                  | c1      | f         |         0 |         4 |         -1 |
-1 |              |                  | {30,31,32,33,34,35,36,37,38,39}
 public     | t1_range_int | range_p03     |                  | c2      | f         |         0 |         4 |         -1 |
-1 |              |                  | {30,31,32,33,34,35,36,37,38,39}
 public     | t1_range_int | range_p03     |                  | c3      | f         |         0 |         4 |         -1 |
-1 |              |                  | {30,31,32,33,34,35,36,37,38,39}
 public     | t1_range_int | range_p03     |                  | c4      | f         |         0 |         4 |         -1 |
-1 |              |                  | {30,31,32,33,34,35,36,37,38,39}
 public     | t1_range_int | range_p04     |                  | c1      | f         |         0 |         4 |         -1 |
-1 |              |                  | {40,41,42,43,44,45,46,47,48,49}
 public     | t1_range_int | range_p04     |                  | c2      | f         |         0 |         4 |         -1 |
-1 |              |                  | {40,41,42,43,44,45,46,47,48,49}
 public     | t1_range_int | range_p04     |                  | c3      | f         |         0 |         4 |         -1 |
-1 |              |                  | {40,41,42,43,44,45,46,47,48,49}
 public     | t1_range_int | range_p04     |                  | c4      | f         |         0 |         4 |         -1 |
-1 |              |                  | {40,41,42,43,44,45,46,47,48,49}
 public     | t1_range_int |               |                  | c1      | f         |         0 |         4 |         -1 |          -1
|              |                  |
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49}
 public     | t1_range_int |               |                  | c2      | f         |         0 |         4 |         -1 |          -1
|              |                  |
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49}
 public     | t1_range_int |               |                  | c3      | f         |         0 |         4 |         -1 |          -1
|              |                  |
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49}
 public     | t1_range_int |               |                  | c4      | f         |         0 |         4 |         -1 |          -1
|              |                  |
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49}
(24 rows)
```

- Generate partition-level statistics of data in multiple columns.

```
gaussdb=# ALTER TABLE t1_range_int ADD STATISTICS ((c2, c3));
gaussdb=# ANALYZE t1_range_int WITH ALL;
```

- View partition-level statistics of data in multiple columns.

```
gaussdb=# SELECT
schemaname,tablename,partitionname,subpartitionname,attname,inherited,null_frac,avg_width,n_disti
nct,n_dndistinct,most_common_vals,most_common_freqs,histogram_bounds FROM pg_ext_stats
WHERE tablename='t1_range_int' ORDER BY tablename,partitionname,attname;
 schemaname |   tablename    | partitionname | subpartitionname | attname | inherited | null_frac |
avg_width | n_distinct | n_dndistinct | most_common_vals | most_common_freqs | histogram_bounds
------------+--------------+---------------+------------------+---------+-----------+-----------
+------------+--------------+------------------+-------------------+------------------
 public     | t1_range_int | range_p00     |                  | 2 3     | f         |         0 |         8 |         -1 |
-1 |                  |                   |
 public     | t1_range_int | range_p01     |                  | 2 3     | f         |         0 |         8 |         -1 |
-1 |                  |                   |
 public     | t1_range_int | range_p02     |                  | 2 3     | f         |         0 |         8 |         -1 |
-1 |                  |                   |
 public     | t1_range_int | range_p03     |                  | 2 3     | f         |         0 |         8 |         -1 |
-1 |                  |                   |
 public     | t1_range_int | range_p04     |                  | 2 3     | f         |         0 |         8 |         -1 |
-1 |                  |                   |
 public     | t1_range_int |               |                  | 2 3     | f         |         0 |         8 |         -1 |           -1
|                  |                   |
(6 rows)
```

- Create an expression index and generate partition-level statistics.

```
gaussdb=# CREATE INDEX t1_range_int_index ON t1_range_int(text(c1)) LOCAL;
gaussdb=# ANALYZE t1_range_int WITH ALL;
```

- View the partition-level statistics of the expression index.

```
gaussdb=# SELECT
schemaname,tablename,partitionname,subpartitionname,attname,inherited,null_frac,avg_width,n_disti
nct,n_dndistinct,most_common_vals,most_common_freqs,histogram_bounds FROM pg_stats WHERE
tablename='t1_range_int_index' ORDER BY tablename,partitionname,attname;
 schemaname |      tablename       |    partitionname     | subpartitionname | attname | inherited |
null_frac | avg_width | n_distinct | n_dndistinct | most_common_vals | most_common_freqs
|                                                    histogram_bounds

------------+--------------------+--------------------+------------------+---------+-----------+-----------
+-----------+------------+--------------+------------------+-------------------
+---------------------------------------------------------------------------------
---------------------------------------------------------
 public     | t1_range_int_index | range_p00_text_idx |                  | text    | f         |         0 |         5
|         -1 |            0 |                  |                   | {0,1,2,3,4,5,6,7,8,9}
 public     | t1_range_int_index | range_p01_text_idx |                  | text    | f         |         0 |         6
|         -1 |            0 |                  |                   | {10,11,12,13,14,15,16,17,18,19}
 public     | t1_range_int_index | range_p02_text_idx |                  | text    | f         |         0 |         6
|         -1 |            0 |                  |                   | {20,21,22,23,24,25,26,27,28,29}
 public     | t1_range_int_index | range_p03_text_idx |                  | text    | f         |         0 |         6
|         -1 |            0 |                  |                   | {30,31,32,33,34,35,36,37,38,39}
 public     | t1_range_int_index | range_p04_text_idx |                  | text    | f         |         0 |         6
|         -1 |            0 |                  |                   | {40,41,42,43,44,45,46,47,48,49}
 public     | t1_range_int_index |                    |                  | text    | f         |         0 |         5 |         -1
|            0 |                  |                   |
{0,1,10,11,12,13,14,15,16,17,18,19,2,20,21,22,23,24,25,26,27,28,29,3,30,31,32,33,3
4,35,36,37,38,39,4,40,41,42,43,44,45,46,47,48,49,5,6,7,8,9}
(6 rows)
```

- Delete the partitioned table.

```
gaussdb=# DROP TABLE t1_range_int;
```

5.3.3.2 Collecting Partition-Level Statistics

Collecting Statistics on a Specified Partition

Collecting statistics on a single partition of the current partitioned table is supported. The partitions whose statistics have been collected will be automatically updated and maintained when the statistics are collected again. This function applies to list partitioning, hash partitioning, and range partitioning.

```
gaussdb=# CREATE TABLE only_fisrt_part(id int,name varchar)PARTITION BY RANGE (id)
 (PARTITION id11 VALUES LESS THAN (1000000),
PARTITION id22 VALUES LESS THAN (2000000),
PARTITION max_id1 VALUES LESS THAN (MAXVALUE));

gaussdb=# INSERT INTO only_fisrt_part SELECT generate_series(1,5000),'test';


gaussdb=# ANALYZE only_fisrt_part PARTITION (id11);
gaussdb=# ANALYZE only_fisrt_part PARTITION (id22);
gaussdb=# ANALYZE only_fisrt_part PARTITION (max_id1);

gaussdb=# SELECT relname, relpages, reltuples FROM pg_partition WHERE relname IN ('id11', 'id22', 
'max_id1');
 relname | relpages | reltuples 
---------+----------+-----------
 id11    |     3400 |      5000
 id22    |        0 |         0
 max_id1 |        0 |         0
(3 rows)

gaussdb=# \x
gaussdb=# SELECT * FROM pg_stats WHERE tablename ='only_fisrt_part' AND partitionname ='id11';
-[ RECORD 1 ]----------
+--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
schemaname             | public
tablename              | only_fisrt_part
attname                | name
inherited              | f
null_frac              | 0
avg_width              | 5
n_distinct             | 1
n_dndistinct           | 0
most_common_vals       | {test}
most_common_freqs      | {1}
histogram_bounds       | 
correlation            | 1
most_common_elems      | 
most_common_elem_freqs | 
elem_count_histogram   | 
partitionname          | id11
subpartitionname       | 
-[ RECORD 2 ]----------
+--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------
schemaname             | public
tablename              | only_fisrt_part
attname                | id
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -1
n_dndistinct           | 0
most_common_vals       | 
most_common_freqs      | 
histogram_bounds       | 
{1,50,100,150,200,250,300,350,400,450,500,550,600,650,700,750,800,850,900,950,1000,1050,1100,1150,1200,
1250,1300,1350,1400,1450,1500,1550,1600,1650,1700,1750,1800,1850,1900,1950,2000,2050,2100,2150,2200,
2250,2300,2350,2400,2450,2500,2550,2600,2650,2700,2750,2800,2850,2900,2950,3000,3050,3100,3150,3200,
3250,3300,3350,3400,3450,3500,3550,3600,3650,3700,3750,3800,3850,3900,3950,4000,4050,4100,4150,4200,
4250,4300,4350,4400,4450,4500,4550,4600,4650,4700,4750,4800,4850,4900,4950,5000}
correlation            | 1
most_common_elems      | 
most_common_elem_freqs | 
```

```
elem_count_histogram   |
partitionname          | id11
subpartitionname       |
gaussdb=# q \x
-- Delete the partitioned table.
gaussdb=# DROP TABLE only_fisrt_part;
```

Optimizer Using the Statistics of a Specified Partition

When processing a partitioned table, the optimizer preferentially uses the statistics of the specified partition for query optimization. However, if no statistics have been collected for the specified partition, the optimizer rewrites the partition clause to perform pruning optimization. For details, see **Rewriting a Partition Clause for Pruning Optimization**.

```
gaussdb=# SET enable_fast_query_shipping = off;
gaussdb=#
CREATE TABLE ONLY_FIRST_PART_TWO
(
    C1 INT,
    C2 BIGINT
)
PARTITION BY RANGE(C1)
(
    PARTITION P_1 VALUES LESS THAN (1000),
    PARTITION P_2 VALUES LESS THAN (3000),
    PARTITION P_3 VALUES LESS THAN (MAXVALUE)
);

gaussdb=# INSERT INTO only_first_part_two SELECT generate_series(1,5000), 0;
gaussdb=# EXPLAIN SELECT * FROM only_first_part_two PARTITION (p_2);
                              QUERY PLAN
--------------------------------------------------------------------------------
 Streaming (type: GATHER)  (cost=0.88..2.89 rows=30 width=12)
   Node/s: All datanodes
   ->  Partition Iterator  (cost=0.00..1.14 rows=30 width=12)
         Iterations: 1
         ->  Partitioned Seq Scan on only_first_part_two  (cost=0.00..1.14 rows=30 width=12)
               Selected Partitions:  2
(6 rows)


gaussdb=# EXPLAIN SELECT * FROM only_first_part_two PARTITION (p_1) where c2 = 2;
                               QUERY PLAN
---------------------------------------------------------------------------------
Streaming (type: GATHER)  (cost=0.06..1.30 rows=1 width=12)
   Node/s: All datanodes
   ->  Partition Iterator  (cost=0.00..1.18 rows=1 width=12)
         Iterations: 1
         ->  Partitioned Seq Scan on only_first_part_two  (cost=0.00..1.18 rows=1 width=12)
               Filter: (c2 = 0)
               Selected Partitions:  1
(7 rows)

gaussdb=# DROP TABLE only_fisrt_part_two;
```

Rewriting a Partition Clause for Pruning Optimization

If there is no partition-level statistics, the optimizer logically rewrites the pseudo-predicate of a partition clause, and uses the rewritten pseudo-predicate to affect the selectivity calculation and obtain a more accurate estimated row quantity based on the statistics of the entire table.

NOTE

- It applies only to selectivity calculation.
- Level-2 partitions are not supported.
- Only range partitioning and list partitioning are supported.
- For range partitioning, only single-column partitioning keys can be rewritten. Multi-column partitioning keys cannot be rewritten.
- For list partitioning, to ensure performance, the maximum number of enumerated values for a specified list partition is 40.
  - When the number of enumerated values for a specified list partition exceeds 40, this feature is no longer applicable.
  - For the default partition, the number of enumerated values is the total number of enumerated values in all non-default partitions.

Example 1: Rewriting a range partition

```
gaussdb=# CREATE TABLE test_int4_maxvalue(id INT, name VARCHAR)
PARTITION BY RANGE(id)
(
    PARTITION id1 VALUES LESS THAN(1000),
    PARTITION id2 VALUES LESS THAN(2000),
    PARTITION max_id VALUES LESS THAN(MAXVALUE)
);
gaussdb=# INSERT INTO test_int4_maxvalue SELECT GENERATE_SERIES(1,5000),'test';
gaussdb=# ANALYZE test_int4_maxvalue with global;

-- Query the specified partition id1.
gaussdb=# EXPLAIN SELECT * FROM test_int4_maxvalue PARTITION(id1);
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes

 Remote SQL: SELECT id, name FROM public.test_int4_maxvalue PARTITION (id1)
 Datanode Name: d1_datanode1
   Partition Iterator  (cost=0.00..7.91 rows=491 width=9)
     Iterations: 1
     ->  Partitioned Seq Scan on test_int4_maxvalue  (cost=0.00..7.91 rows=491 width=9)
           Selected Partitions:  1

 Datanode Name: d1_datanode2
   Partition Iterator  (cost=0.00..8.08 rows=508 width=9)
     Iterations: 1
     ->  Partitioned Seq Scan on test_int4_maxvalue  (cost=0.00..8.08 rows=508 width=9)
           Selected Partitions:  1

(16 rows)

-- Query the max_id of the specified partition.
gaussdb=# EXPLAIN SELECT * FROM test_int4_maxvalue PARTITION(max_id);
                                         QUERY PLAN
---------------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes

 Remote SQL: SELECT id, name FROM public.test_int4_maxvalue PARTITION (max_id)
 Datanode Name: d1_datanode1
   Partition Iterator  (cost=0.00..24.46 rows=1546 width=9)
     Iterations: 1
     ->  Partitioned Seq Scan on test_int4_maxvalue  (cost=0.00..24.46 rows=1546 width=9)
           Selected Partitions:  3

 Datanode Name: d1_datanode2
   Partition Iterator  (cost=0.00..23.55 rows=1455 width=9)
     Iterations: 1
```

```
    ->  Partitioned Seq Scan on test_int4_maxvalue  (cost=0.00..23.55 rows=1455 width=9)
          Selected Partitions:  3

(16 rows)

-- Delete the partitioned table.
gaussdb=# DROP TABLE test_int4_maxvalue;
```

Example 2: Rewriting a list partition

```
gaussdb=# CREATE TABLE test_default
(
    c1 INT,
    c2 BIGINT
)
PARTITION BY LIST(c2)
(
    PARTITION p_1 VALUES (10000, 20000),
    PARTITION p_2 VALUES (300000, 400000, 500000),
    PARTITION p_3 VALUES (DEFAULT)
);
gaussdb=# INSERT INTO test_default SELECT GENERATE_SERIES(1, 1000), 10000;
gaussdb=# INSERT INTO test_default SELECT GENERATE_SERIES(1001, 2000), 600000;
gaussdb=# ANALYZE test_default with global;

-- Query the specified partition p_1.
gaussdb=# EXPLAIN SELECT * FROM test_default PARTITION(p_1);
                                    QUERY PLAN
------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes

 Remote SQL: SELECT c1, c2 FROM public.test_default PARTITION (p_1)
 Datanode Name: d1_datanode1
   Partition Iterator  (cost=0.00..7.92 rows=492 width=12)
     Iterations: 1
     ->  Partitioned Seq Scan on test_default  (cost=0.00..7.92 rows=492 width=12)
           Selected Partitions:  1

 Datanode Name: d1_datanode2
   Partition Iterator  (cost=0.00..8.08 rows=508 width=12)
     Iterations: 1
     ->  Partitioned Seq Scan on test_default  (cost=0.00..8.08 rows=508 width=12)
           Selected Partitions:  1

(16 rows)

-- Query the specified partition p_3.
gaussdb=# EXPLAIN SELECT * FROM test_default PARTITION(p_3);
                                    QUERY PLAN
------------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes

 Remote SQL: SELECT c1, c2 FROM public.test_default PARTITION (p_3)
 Datanode Name: d1_datanode1
   Partition Iterator  (cost=0.00..8.24 rows=524 width=12)
     Iterations: 1
     ->  Partitioned Seq Scan on test_default  (cost=0.00..8.24 rows=524 width=12)
           Selected Partitions:  3

 Datanode Name: d1_datanode2
   Partition Iterator  (cost=0.00..7.76 rows=476 width=12)
     Iterations: 1
     ->  Partitioned Seq Scan on test_default  (cost=0.00..7.76 rows=476 width=12)
           Selected Partitions:  3

(16 rows)

-- Delete the partitioned table.
gaussdb=# DROP TABLE test_default;
```

5.3.4 Partition-Wise Join

Partition-wise join is a partition-level parallel optimization technology. It divides a large join into smaller joins between a pair of partitions from the two joined tables when certain conditions are met. This improves the join query performance of partitioned tables by concurrent execution and reducing the amount of data communication.

Partition-wise join is applicable in SMP and non-SMP scenarios.

5.3.4.1 Partition-Wise Join in Non-SMP Scenarios

In non-SMP scenarios, a partition-wise join path is generated based on a rule, that is, a partition-wise join path can be generated as long as a condition is met, without comparing path costs. To enable this function, set the GUC parameter **enable_partitionwise**.

Usage Specifications

Partition-wise join usage specifications in non-SMP scenarios are as follows:

- Only level-1 range partitions are supported.
- Hash Join, Nestloop Join, and Merge Join are supported.
- Only Inner Join is supported.
- **query_dop** must be set to **1**.
- In non-SMP scenarios, partition-wise join selects paths based on a rule. Therefore, using the partition-wise join plan may cause performance deterioration. You can determine whether to enable it based on the actual situation.
- Only FQS plans are supported.

Examples

```
-- Create a range partitioned table.
gaussdb=# CREATE TABLE range_part (
gaussdb(#     a INTEGER,
gaussdb(#     b INTEGER,
gaussdb(#     c INTEGER
gaussdb(# ) PARTITION BY RANGE (a)
gaussdb-# (
gaussdb(# PARTITION range_part_p1 VALUES LESS THAN (10),
gaussdb(# PARTITION range_part_p2 VALUES LESS THAN (20),
gaussdb(# PARTITION range_part_p3 VALUES LESS THAN (30),
gaussdb(# PARTITION range_part_p4 VALUES LESS THAN (40)
gaussdb(# );

-- Use an FQS plan.
gaussdb=# SET enable_fast_query_shipping= ON;
SET

-- Set query_dop to 1 to disable SMP.
gaussdb=# SET query_dop = 1;
SET

-- Disable partition-wise join in a non-SMP scenario.
gaussdb=# SET enable_partitionwise = off;
SET

-- View the non-partition-wise join execution plan.
```

```
gaussdb=# SET max_datanode_for_plan = 1;
SET
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM range_part t1 INNER JOIN range_part t2 ON (t1.a = t2.a);
                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Data Node Scan
   Node/s: All datanodes

 Remote SQL: SELECT t1.a, t1.b, t1.c, t2.a, t2.b, t2.c FROM public.range_part t1 JOIN public.range_part t2 ON
t1.a = t2.a
 Datanode Name: datanode1
   Hash Join
     Hash Cond: (t1.a = t2.a)
     ->  Partition Iterator
           Iterations: 4
           ->  Partitioned Seq Scan on range_part t1
                 Selected Partitions:  1..4
     ->  Hash
           ->  Partition Iterator
                 Iterations: 4
                 ->  Partitioned Seq Scan on range_part t2
                       Selected Partitions:  1..4

(17 rows)

-- Enable partition-wise join in a non-SMP scenario.
gaussdb=# SET enable_partitionwise = on;
SET

-- View the partition-wise join plan in a non-SMP scenario. According to the execution plan, the Partition
Iterator operator is pulled up to the upper layer of the Hash Join operator. In this way, each pair of
partitions is scanned and joined immediately. (Previously, partition join starts after all partitions are
scanned.)
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM range_part t1 INNER JOIN range_part t2 ON (t1.a = t2.a);
                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Data Node Scan
   Node/s: All datanodes

 Remote SQL: SELECT t1.a, t1.b, t1.c, t2.a, t2.b, t2.c FROM public.range_part t1 JOIN public.range_part t2 ON
t1.a = t2.a
 Datanode Name: datanode1
   Result
     ->  Partition Iterator
           Iterations: 4
           ->  Hash Join
                 Hash Cond: (t1.a = t2.a)
                 ->  Partitioned Seq Scan on range_part t1
                       Selected Partitions:  1..4
                 ->  Hash
                       ->  Partitioned Seq Scan on range_part t2
                             Selected Partitions:  1..4

(16 rows)

-- Drop the partitioned table.
gaussdb=# DROP TABLE range_part;
```

5.3.4.2 Full Partition-Wise Join in the SMP Scenario

The partition-wise join plan in the SMP scenario is selected based on costs. In the path generation process, estimated costs of partition-wise join and non-partition-wise join paths are compared, and a path with a lower cost is selected. To enable it, set GUC parameter **enable_smp_partitionwise**.

Full partition-wise join applies to two tables with the same partitioning policy. The condition for generating a full partition-wise join path is that the partition keys of the two tables are a pair of matching join keys.

Usage Specifications

Full partition-wise join usage specifications in the SMP scenario are as follows:

- Level-1 hash partitioned tables and level-1 range partitioned tables are supported.
- For hash partitioned tables, the same partitioning policy means that the partition key types and the number of partitions are the same.
- For range partitioned tables, the same partitioning policy means that the partition key types, number of partitions, number of partition keys, and boundary values of each partition are the same.
- Only stream plans are supported.
- Only the scenario where the partition key and distribution key are the same is supported.
- Only the scenario where the Join operator can complete calculation on a single DN is supported. That is, data of the Join operator does not cross nodes.
- Hash Join and Merge Join are supported.
- Seqscan, Indexscan, Indexonlyscan and Imcvscan are supported. For Indexscan and Indexonlyscan, only partitioned local indexes are supported, and the index type is B-tree or UB-tree.
- Related specifications are inherited from SMP specifications. IUD operations are not supported in the SMP scenario.
- The SMP function must be enabled and the value of **query_dop** must be greater than **1**.

Examples

```
-- Create a hash partitioned table.
gaussdb=# CREATE TABLE hash_part
(
    a INTEGER,
    b INTEGER,
    c INTEGER
)
DISTRIBUTE BY HASH(a)
PARTITION BY HASH(a)
(
    PARTITION p1,
    PARTITION p2,
    PARTITION p3,
    PARTITION p4,
    PARTITION p5
);
CREATE TABLE

-- Use a Stream plan.
gaussdb=# SET enable_fast_query_shipping = off;
SET
gaussdb=# SET enable_stream_operator = on;
SET

-- Set query_dop to 5 to enable SMP.
```

```
gaussdb=# SET query_dop = 5;
SET

-- Disable the partition-wise join function in the SMP scenario.
gaussdb=# SET enable_smp_partitionwise = off;
SET

-- View a non-partition-wise join plan. According to the plan, after data scanning is completed using the
Partition Iterator and Partitioned Seq Scan operators at two layers, data is redistributed using the
Streaming(type: LOCAL REDISTRIBUTE) operator to ensure that data in the Join operator matches each
other.
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM hash_part t1, hash_part t2 WHERE t1.a = t2.a;
                              QUERY PLAN
------------------------------------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Streaming(type: LOCAL GATHER dop: 1/5)
         Spawn on: All datanodes
         ->  Nested Loop
               Join Filter: (t1.a = t2.a)
               ->  Streaming(type: LOCAL REDISTRIBUTE dop: 5/5)
                     Spawn on: All datanodes
                     ->  Partition Iterator
                           Iterations: 5
                           ->  Partitioned Seq Scan on hash_part t1
                                 Selected Partitions:  1..5
               ->  Materialize
                     ->  Streaming(type: LOCAL REDISTRIBUTE dop: 5/5)
                           Spawn on: All datanodes
                           ->  Partition Iterator
                                 Iterations: 5
                                 ->  Partitioned Seq Scan on hash_part t2
                                       Selected Partitions:  1..5
(19 rows)

-- Enable the partition-wise join function in the SMP scenario.
gaussdb=# SET enable_smp_partitionwise = on;
SET

-- View the execution plan of partition-wise join. According to the plan, the partition-wise join plan
eliminates the Streaming operator, that is, data does not need to be redistributed between threads, thereby
reducing overheads of data transfer and improving performance of the Join operation.
gaussdb=# EXPLAIN (COSTS OFF) SELECT * FROM hash_part t1, hash_part t2 WHERE t1.a = t2.a;
                           QUERY PLAN
-------------------------------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Streaming(type: LOCAL GATHER dop: 1/5)
         Spawn on: All datanodes
         ->  Hash Join (Partition-wise Join)
               Hash Cond: (t1.a = t2.a)
               ->  Partition Iterator
                     Iterations: 5
                     ->  Partitioned Seq Scan on hash_part t1
                           Selected Partitions:  1..5
               ->  Hash
                     ->  Partition Iterator
                           Iterations: 5
                           ->  Partitioned Seq Scan on hash_part t2
                                 Selected Partitions:  1..5
(15 rows)

-- Drop the partitioned table.
gaussdb=# DROP TABLE hash_part;
```


NOTE

A partition-wise join message is displayed on the right of the Join operator only for a partition-wise join plan in the SMP scenario. This message is not displayed in non-SMP scenarios.

5.4 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.

  NOTE

  - 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.
  - 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.

5.4.1 ADD PARTITION

You can add partitions to an existing partitioned table to maintain new services. Currently, a partitioned table supports a maximum of 1,048,575 partitions. Once the upper limit is reached, no more partitions can be added.

The memory overhead occupied by partitions needs to be considered in this process. The memory used by a partitioned table is about (Number of partitions x 3/1024) MB. The memory occupied by partitions cannot be greater than the value of **local_syscache_threshold**. In addition, a certain amount of memory space needs to be reserved to ensure that other functions can run properly.

CAUTION

- This command cannot be applied to hash partitions.
- New partitions do not inherit the sparsely partitioned index attribute of the table.

5.4.1.1 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**.

5.4.1.2 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 run the **ALTER TABLE SPLIT PARTITION** command to split partitions.

5.4.2 DROP PARTITION

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

NOTICE

- 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.
- When a partition is deleted, if the partition has a sparsely partitioned index that belongs only to the current partition, the sparsely partitioned index is deleted in cascading mode.

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 running the **ALTER TABLE DROP PARTITION** command.
- If the partitioned table is a hash partitioned table, partitions in the table cannot be deleted by running the **ALTER TABLE DROP PARTITION** command.

5.4.3 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.

NOTICE

- 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.
- 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** ordinary table are provided, and the **DATE_202001** partition is exchanged with the **exchange_sales** table, the following behaviors exist based on the declaration clause:

- If WITHOUT VALIDATION is declared, all data is exchanged to the **DATE_202001** partition. Because **'2020-02-03'** and **'2020-04-08'** do not meet the range constraint of the **DATE_202001** partition, subsequent services may be abnormal.
- If WITH VALIDATION is declared, and **'2020-02-03'** and **'2020-04-08'** do not meet the range constraint of the **DATE_202001** partition, the database reports an error.
- If WITH VALIDATION VERBOSE is declared, the database inserts **'2020-02-03'** into the **DATE_202002** partition, inserts **'2020-04-08'** into the **DATE_202004** partition, and exchanges the remaining data with the **DATE_202001** partition.

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

WARNING

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

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

- The number of columns in an ordinary table is the same as that in a partition, and the information in the corresponding columns is strictly consistent.
- The compression information of the ordinary table and partitioned table is consistent.

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

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;
```

5.4.4 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 use 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');
```


5.4.5 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.
- If the target partition contains a sparsely partitioned index, the partition cannot be split.
- 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.

5.4.5.1 Splitting a Partition for a Range Partitioned Table

You can use 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.

5.4.5.2 Splitting a Partition for a List Partitioned Table

You can use 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.

5.4.6 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.
- If any partition before combination contains a sparsely partitioned index, the partition cannot be combined.

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 use 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;
```

5.4.7 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;
```

5.4.8 RENAME PARTITION

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

5.4.8.1 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;
```

5.4.8.2 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.

5.4.9 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';
```

5.4.10 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.

5.4.10.1 Invalidating/Rebuilding Indexes

You can use 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;
```

5.4.10.2 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;
  ```

5.5 Partition Concurrency Control

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

5.5.1 Common Lock Design

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

Table 5-2 Common lock conflict description

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

Different statements of a partitioned table are applied to the same target partition. The database applies different levels of table locks and partition locks to the target partitioned table and partition to control the concurrency behavior. **Table 5-3** specifies the lock control level of different partitioned table statements. The numbers 1 to 8 indicate the eight levels of common locks specified by **Table 5-2**, which are ACCESS_SHARE, ROW_SHARE, ROW_EXCLUSIVE, SHARE_UPDATE_EXCLUSIVE, SHARE, SHARE_ROW_EXCLUSIVE, EXCLUSIVE, and ACCESS_EXCLUSIVE, respectively.

Table 5-3 Lock control level of different partitioned table statements

Statement	Partitioned Table Lock (Table Lock +Partition Lock)
SELECT	1-1
SELECT FOR UPDATE	2-2
DML statements, including INSERT, UPDATE, DELETE, UPSERT, MERGE INTO, and COPY	3-3
Most partition-level DDL statements, including ADD, DROP, EXCHANGE, TRUNCATE, MOVE and RENAME	4-8
CREATE INDEX (non-sparsely-partitioned index) and REBUILD INDEX	5-5
CREATE INDEX (sparsely partitioned index)	3-5
REBUILD INDEX PARTITION	1-5
ANALYZE and VACUUM	4-4
Other DDL statements on partitioned tables, including partition-level SPLIT and MERGE	8-8

If both the table lock and partition lock applied during service execution meet the lock control requirements specified in **Table 5-2**, concurrent service operations are supported. If either of the table lock and partition lock is not supported, concurrent service operations are not allowed.

NOTE

The table-level lock applied by the DDL operations (ADD/DROP/TRUNCATE/EXCHANGE/MOVE/RENAME) to a partitioned table is controlled by the GUC parameter **enable_partition_ddl_lowlevel_lock**. If **enable_partition_ddl_lowlevel_lock** is set to **on**, a level-4 lock is applied to the table; if it is set to **off**, a level-8 lock is applied.

5.5.2 DQL/DML-DQL/DML Concurrency

Levels 1–3 regular locks are applied to DQL/DML operations on tables and partitions.

DQL/DML operations do not block each other, allowing for concurrent execution of DQL/DML-DQL/DML operations.

5.5.3 DQL/DML-DDL Concurrency

Performing table-level DDL operations or partition-level DDL operations SPLIT or MERGE on a partitioned table will apply a level-8 lock to the partitioned table to block all DQL/DML statements.

Performing partition-level DDL operations ADD, DROP, EXCHANGE, TRUNCATE, MOVE, or RENAME on a partitioned table will apply a level-4 lock to the partitioned table and a level-8 lock to the target partition. When DQL/DML and DDL statements are used in different partitions, concurrent execution is supported. When DQL/DML and DDL statements are used in the same partition, services triggered later will be blocked.

NOTICE

- During partition-level DDL operations, do not perform DQL/DML operations on the target partition at the same time.
- If the target partitions of the concurrent DDL and DQL/DML statements overlap, the DQL/DML statements may occur before or after the DDL statements due to serial blocking. You need to know the possible expected results. For example, when TRUNCATE and INSERT take effect on the same partition, if TRUNCATE is triggered before INSERT, data exists in the target partition after the statements are complete. If TRUNCATE is triggered after INSERT, no data exists in the target partition after the statements are complete.
- If a partitioned table is a bucket table with a global index, the global index is rebuilt when partition-level DDL operations such as MERGE, EXCHANGE, and SET TABLESPACE are performed on the table, blocking DML services during this period.

DQL/DML-DDL Concurrency Across Partitions

GaussDB supports DQL/DML-DDL concurrency across partitions.

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

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

CREATE TABLE temp
(
    product_id     INT4 NOT NULL,
    customer_id    INT4 NOT NULL,
    time_id        DATE,
    channel_id     CHAR(1),
    type_id        INT4,
    quantity_sold  NUMERIC(3),
    amount_sold    NUMERIC(10,2)
);
```

Typically, the partitioned table supports the following concurrent services:


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

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

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

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

DQL/DML-DDL Concurrency on the Same Partition

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

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

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

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

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

- **ADD PARTITION**

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

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

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

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

- **EXCHANGE PARTITION**

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

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

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

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

- **SPLIT PARTITION**

  SPLIT PARTITION blocks DQL/DML operations on the entire table, including those on the target partition. After the SPLIT operation is committed, the DQL/DML operation is performed based on the partitioned table structure after the SPLIT operation is complete.

- **MERGE PARTITION**

  MERGE PARTITION blocks DQL/DML operations on the entire table, including those on the target partition. After the MERGE operation is committed, the DQL/DML operation is performed based on the partitioned table structure after the MERGE operation is complete.

- **RENAME PARTITION**

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

- **MOVE PARTITION**

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

CAUTION

- During the DQL/DML operations, if multiple DDL operations are consecutively performed on the partition where DQL/DML operations are performed, there is a low probability that an error is reported, indicating that the partition cannot be found and has been deleted by a DDL operation.
- There is a low probability that service deadlock or lock timeout occurs when DQL/DML-DDL statements are concurrently executed on the same partition. This is because some DQL/DML operations apply partition locks only on DNs. If the lock sequence of DQL/DML operations on different DNs is different from that of DDL operations, there is a possibility that a deadlock or lock timeout occurs. To avoid deadlocks, you can set the GUC parameter **enable_partition_ddl_lowlevel_lock** to **off**. However, the performance will be affected.

5.6 System Views & DFX Related to Partitioned Tables

5.6.1 System Views Related to Partitioned Tables

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

1. Views related to all partitions:
   - ADM_PART_TABLES: stores information about all partitioned tables.
   - ADM_TAB_PARTITIONS: stores information about all partitions.
   - ADM_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.

5.6.2 Built-in Tool Functions Related to Partitioned Tables

Information About Table Creation

- Create a table.
```
CREATE TABLE test_range_pt (a INT, b INT, c INT)
PARTITION BY RANGE (a)
(
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN (3000),
    partition p3 VALUES LESS THAN (4000),
    partition p4 VALUES LESS THAN (5000),
    partition p5 VALUES LESS THAN (MAXVALUE)
)ENABLE ROW MOVEMENT;
```
- View the OID of the partitioned table.
```
SELECT oid FROM pg_class WHERE relname = 'test_range_pt';
oid
-------
49290
(1 row)
```
- View the partition information.
```
SELECT oid,relname,parttype,parentid,boundaries FROM pg_partition WHERE parentid = 49290;
oid  |    relname    | parttype | parentid | boundaries
-------+---------------+----------+----------+------------
49293 | test_range_pt  | r        |    49290 |
49294 | p1             | p        |    49290 | {2000}
49295 | p2             | p        |    49290 | {3000}
49296 | p3             | p        |    49290 | {4000}
49297 | p4             | p        |    49290 | {5000}
49298 | p5             | p        |    49290 | {NULL}
(6 rows)
```
- Create an index.
```
CREATE INDEX idx_range_a ON test_range_pt(a) LOCAL;
CREATE INDEX
-- Check the OID of the partitioned index.
SELECT oid FROM pg_class WHERE relname = 'idx_range_a';
oid
-------
90250
(1 row)
```
- View the index partition information.
```
SELECT oid,relname,parttype,parentid,boundaries,indextblid FROM pg_partition WHERE parentid = 
90250;
oid  | relname  | parttype | parentid | boundaries | indextblid
-------+----------+----------+----------+------------+------------
90255 | p5_a_idx | x        |    90250 |            |      49298
90254 | p4_a_idx | x        |    90250 |            |      49297
90253 | p3_a_idx | x        |    90250 |            |      49296
90252 | p2_a_idx | x        |    90250 |            |      49295
90251 | p1_a_idx | x        |    90250 |            |      49294
(5 rows)
```

Example of Tool Functions

- **pg_get_tabledef** is used to obtain the definition of a partitioned table. The input parameter can be the table OID or table name.
```
SELECT pg_get_tabledef('test_range_pt');

pg_get_tabledef
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
 SET search_path = 
public;
                                        +
```

```
 CREATE TABLE test_range_pt
(
                                    +
     a
integer,
                                                           +
     b
integer,
                                                           +
     c
integer
                                                           +
 )
                                                               +
 WITH (orientation=row, compression=no, storage_type=USTORE,
segment=off)
             +
 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;
                                                             +
 CREATE INDEX idx_range_a ON test_range_pt USING ubtree (a) LOCAL(PARTITION p1_a_idx,
PARTITION p2_a_idx, PARTITION p3_a_idx, PARTITION p4_a_idx, PARTITION p5_a_idx)  WITH
(storage_type=USTORE) TABLESPACE pg_default;
(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)
```

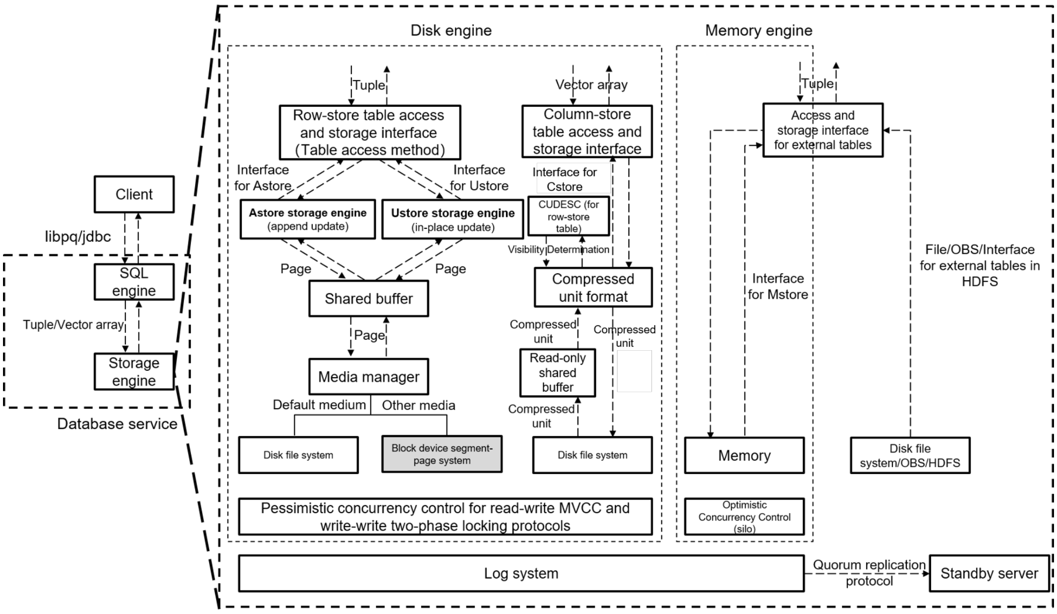

6 Storage Engine

6.1 Storage Engine Architecture

6.1.1 Overview

6.1.1.1 Static Compilation Architecture

From the perspective of the entire database service architecture, the storage engine upward connects to the SQL engine to provide or receive data in a standard format (tuple or vector array) for or from the SQL engine, and downward reads data from or writes data to storage media by a specific data organization mode such as page, compress unit, or other forms through specific APIs provided by the storage media. GaussDB 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.



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

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

6.1.1.2 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 uses physical logs. The write, transmission, and replay operations of physical logs are transparent to the storage engine.

6.1.2 Setting Up a Storage Engine

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

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

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

Example:

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

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, segment=off

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 | test  | table | z7ee88f3a | 0 bytes |
{orientation=row,compression=no,storage_type=USTORE,segment=off} |
 public | test1 | table | z7ee88f3a | 0 bytes |
{orientation=row,storage_type=ustore,compression=no,segment=off} |
 public | test2 | table | z7ee88f3a | 0 bytes | {orientation=row,storage_type=astore,compression=no}             |
 public | test4 | table | z7ee88f3a | 0 bytes |
{orientation=row,compression=no,storage_type=USTORE,segment=off} |
(4 rows)

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

gaussdb=#  DROP TABLE test;
gaussdb=#  DROP TABLE test1;
gaussdb=#  DROP TABLE test2;
gaussdb=#  DROP TABLE test4;
```

6.1.3 Storage Engine Update Description

6.1.3.1 GaussDB Kernel 505

- Ustore supports efficient storage of flexible fields.

- Ustore supports large-scale commercial use of TOAST.

- Added the page recovery and escape technologies to Ustore.

- Ustore supports the SMP technology.

6.1.3.2 GaussDB Kernel 503

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

- Added the online index rebuild function to Ustore.

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

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

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

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

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

- Added the online index creation function for Ustore.

- Adapted Ustore to the flashback function and ultimate RTO.

6.1.3.3 GaussDB Kernel 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, VACUUM FULL, and column constraints DEFERRABLE and INITIALLY DEFERRED.

6.2 Astore

6.2.1 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, because it does not have a rollback segment, old data is stored in the original files. Therefore, when the database crashes, complex restoration is not performed like that performed by a Ustore database.
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 can be completed quickly because data is not deleted.

> ⚠ CAUTION
>
> Rollback is complex. During transaction rollback, the modifications made by the transaction must be undone, the inserted records must be deleted, and the updated records must be rolled back. In addition, a large number of redo logs are generated during the 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.
5. The recycle bin (flashback DROP and flashback TRUNCATE) function is supported.

6.3 Ustore

6.3.1 Overview

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

Ustore Advantages

- 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 versions can be recycled in batches, and space bloat is controllable.
- The transaction information is added to B-tree indexes to perform MVCC independently. This increases the proportion of IndexOnlyScan and greatly reduces the number of times that TABLE ACCESS BY INDEX ROWID is executed.
- VACUUM is not the only way to clear historical data. Spaces are recycled independently. Indexes are decoupled from heap tables and can be cleared independently. The I/O stability is better.
- In the scenario where a large number of concurrent updates are performed on the same row, row ID offset may occur in an Astore table. The in-place update mechanism of Ustore ensures the stability of tuple row IDs and update latency.
- The flashback function is supported.

CAUTION

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

6.3.1.1 Ustore Features and Specifications

6.3.1.1.1 Restrictions

Category	Feature	Supported or Not
Transaction	Serializable	×
	DDL operations on a partitioned table in a transaction block	×
Scalability	Hash bucket	×

Category	Feature	Supported or Not
SQL	Table sampling/Materialized view/Key-value lock	×

6.3.1.1.2 Storage Specifications

1. The maximum number of columns in a data table is 1600.
2. Transaction directory (TD) **init_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**. The value range is [2,128], and the default value is **4**. A single page supports a maximum of 128 concurrent transactions.
3. 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.
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.

6.3.1.2 Examples

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
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, storage_type=ustore, compression=no, segment=off
```

Delete a Ustore table.

```
gaussdb=# DROP TABLE ustore_table;
 DROP TABLE
```

Create an index for a Ustore table.

Currently, Ustore supports only multi-version indexes of the B-tree type. In some scenarios, to distinguish from the B-tree indexes of Astore tables, the multi-version B-tree indexes of Ustore tables are also called UB-tree indexes. (For details about the UB-tree, see **UB-Tree**.) 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.

CAUTION

UB-tree indexes are classified into RCR UB-tree and PCR UB-tree. By default, an RCR UB-tree is created. If WITH option **index_txntype** is set to **pcr** or GUC parameter **index_txntype** is set to **pcr** during index creation, a PCR UB-tree is created.

```
gaussdb=# CREATE TABLE test(a int);
CREATE TABLE
gaussdb=# CREATE INDEX UB_tree_index ON test(a);
CREATE INDEX
gaussdb=# \d+ test
Table "public.test"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 a      | integer |           | plain   |              |
Indexes:
    "ub_tree_index" ubtree (a) WITH (storage_type=USTORE) TABLESPACE pg_default
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off

-- Delete a Ustore table index.
gaussdb=# DROP TABLE test;
DROP TABLE
```

6.3.1.3 Best Practices of Ustore

6.3.1.3.1 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. You can also determine whether to adjust it based on the occurrence frequency of "wait available td" events during service running. Generally, **wait available td** is **0**. If the value of **wait available td** is not **0**, the "wait available td" event exists. In this case, you are advised to increase the value of **init_td** and observe the event for several times. If this case occasionally occurs, you are advised not to adjust the value because the 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. **wait available td** is one of the values of **wait_status**. **wait_status** indicates the waiting status of the current thread, including the waiting status details. You can query the value of **wait_status** in the PG_THREAD_WAIT_STATUS view. The value **none** indicates that the system is not waiting for any event. If there is a wait event, you can view the value of **wait available td**. The following is an example. For details about **init_td**, see "SQL Reference > SQL Syntax > CREATE TABLE" in *Developer Guide*. To view and modify the value of **init_td**, perform the following steps:

```
gaussdb=# CREATE TABLE test1(name varchar) WITH(storage_type = ustore, init_td=2);
gaussdb=# \d+ test1
                         Table "public.test1"
 Column |       Type        | Modifiers | Storage  | Stats target | Description
--------+-------------------+-----------+----------+--------------+-------------
 name   | character varying |           | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, storage_type=ustore, init_td=2, compression=no, segment=off,
toast.storage_type=ustore, toast.toast_storage_type=enhanced_toast

gaussdb=# ALTER TABLE test1 SET(init_td=8);
gaussdb=# \d+ test1
                         Table "public.test1"
 Column |       Type        | Modifiers | Storage  | Stats target | Description
--------+-------------------+-----------+----------+--------------+-------------
 name   | character varying |           | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, storage_type=ustore, compression=no, segment=off, init_td=8,
toast.storage_type=ustore, toast.toast_storage_type=enhanced_toast

gaussdb=# SELECT * FROM pg_thread_wait_status;
 node_name | db_name |          thread_name          |    query_id     |       tid       |    sessionid    | lwtid |
psessionid | tlevel | smpid | wait_status | wait_event | locktag | lo
ckmode | block_sessionid | global_sessionid
-----------+----------+--------------------------------+-----------------+-----------------+-----------------+-------
+------------+--------+-------+-------------+------------+---------+---
-------+-----------------+------------------
 sgnode    |         | PageWriter                     |               0 | 139769678919424 | 139769678919424 | 16915
|            |      0 |     0 | none        | none       |         |
       |                 | 0:0#0
 sgnode    |         | PageWriter                     |               0 | 139769736066816 | 139769736066816 | 16913
|            |      0 |     0 | none        | none       |         |
       |                 | 0:0#0
 sgnode    |         | PageWriter                     |               0 | 139769707755264 | 139769707755264 | 16914
|            |      0 |     0 | none        | none       |         |
       |                 | 0:0#0
 sgnode    |         | PageWriter                     |               0 | 139769761756928 | 139769761756928 | 16912
|            |      0 |     0 | none        | none       |         |
       |                 | 0:0#0
 sgnode    |         | PageWriter                     |               0 | 139769783772928 | 139769783772928 | 16911
|            |      0 |     0 | none        | none       |         |
```

```
      |                 | 0:0#0


gaussdb=# DROP TABLE test1;
DROP TABLE
```

6.3.1.3.2 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 **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 variable-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. To view and modify **fillfactor**, perform the following steps:

```
gaussdb=# CREATE TABLE test(a int) with(fillfactor=100);
gaussdb=# \d+ test
                         Table "public.test"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 a      | integer |           | plain   |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, fillfactor=100, compression=no, storage_type=USTORE, segment=off

gaussdb=# ALTER TABLE test set(fillfactor=92);
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, segment=off, fillfactor=92

gaussdb=# DROP TABLE test;
DROP TABLE
```

6.3.1.3.3 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 for three modules (UPAGE:UBTREE:UNDO). Keep it enabled on the live network, except in performance-sensitive scenarios.

To disable it, run the following command:

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

To enable it, run the following command:

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

6.3.1.3.4 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 specific scenarios and setting methods 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. The valid unit is second, minute, hour, or day. 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 that is more suitable for your service model:

   For the system function gs_stat_undo that collects undo statistics, if the input parameter is **false**, optimization suggestions on the **undo_space_limit_size**, **undo_limit_size_per_transaction** and **undo_retention_time** parameters are provided. For details about the parameter values, see "SQL Reference > Functions and Operators > Undo System Functions" in *Developer Guide*.

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

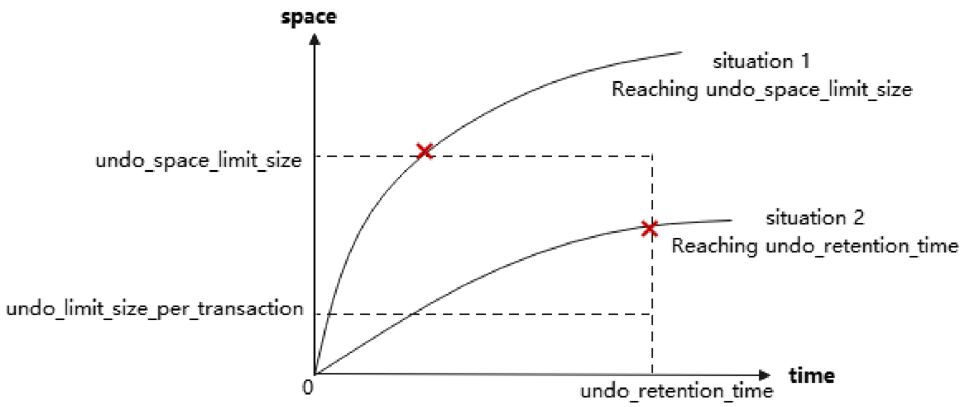
   If long transactions or large transactions exist in your service, undo space may bloat. In this case, you need to increase the value of **undo_space_limit_size**. The default value of **undo_space_limit_size** is **256GB**, and the value ranges from 800 MB to 16 TB.

   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 the **gs_stat_undo()** view, 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.

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.



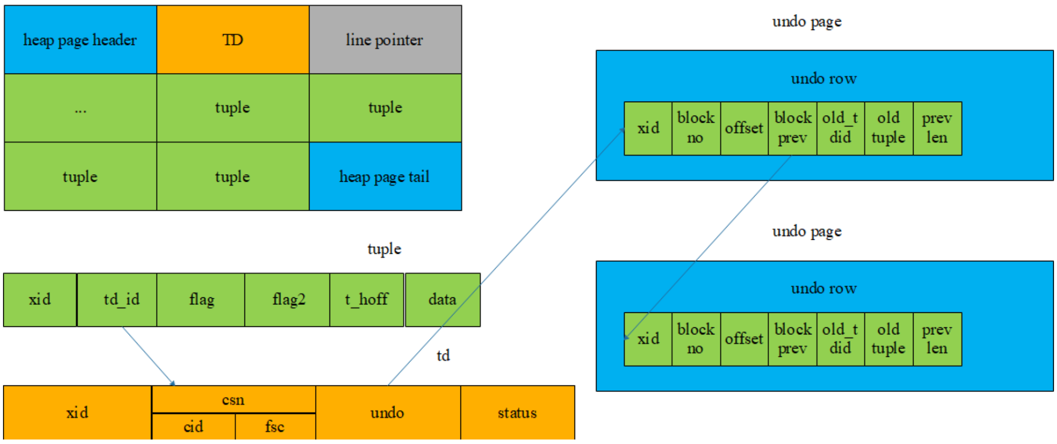
For example, the **undo_space_limit_size** parameter is set to **1 GB**, and the **undo_retention_time** parameter is set to **900s**. If the historical version data generated within 900s is less than 1 GB multiplied by 0.8, data recycling is performed every 900s. Otherwise, data recycling is performed when the volume of generated data reaches 1 GB multiplied by 0.8. 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**.

6.3.2 Storage Format

6.3.2.1 RCR Uheap

6.3.2.1.1 RCR Uheap Multi-Version Management

Ustore has made the following enhancements to the heap it uses, which is referred to as Uheap.



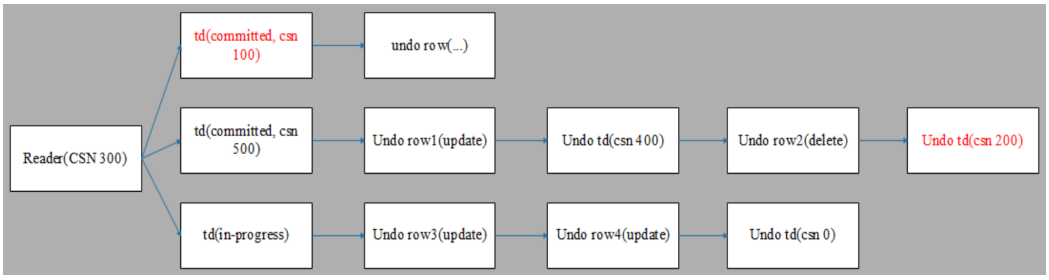
The multi-version management of Ustore row consistency read (RCR) is based on data row levels. Instead of commonly storing XIDs in data rows, Ustore records the XIDs in the transaction directory (TD) area of the page, saving page space. When a transaction modifies a record, historical data is recorded in the undo row. The generated undo row addresses (**zone_id**, **block no**, and **page offset**) are recorded in the TD slot to which **td_id** in tuple points, and new data is overwritten to the page. When a tuple is accessed, the tuple is restored along the version chain until the corresponding version is found.

6.3.2.1.2 RCR Uheap Visibility Mechanism

Ustore visibility is determined by building consistent versions of data rows. Old snapshots can be obtained from undo records. For example:

The figure below illustrates three scenarios of MVCC snapshot visibility checks. A reader thread with a snapshot CSN of 300 determines which version of a row is visible:

1. The TD slot pointed to by the row records a transaction (XID) that is already committed, with a TD CSN less than the snapshot's CSN. This means the TD XID is visible to the snapshot. Furthermore, the transaction XID that generated the current page version of this row must be older than the TD XID and is also committed. Consequently, the latest page version is the visible version for the snapshot.
2. The TD slot records a committed transaction XID, but its TD CSN is greater than the snapshot's CSN, making it invisible. Therefore, the system must traverse the undo chain from newest to oldest to locate a visible version (that is, a version generated by a committed transaction with a CSN less than 300). It ultimately finds the version with a CSN of 200.
3. The transaction that reuses the TD slot is still in progress, making its TD XID invisible to the snapshot. Again, the system traverses the undo chain from newest to oldest to find a visible version. The traversal concludes without finding a corresponding undo record, indicating that even the undo log generated by the transaction that created the current page version has been purged. This implies that all historical versions of this tuple have become visible to all possible snapshots. Therefore, the latest page version is the visible version.



6.3.2.1.3 RCR Uheap Free Space Management

Ustore uses the free space map (FSM) file to record the free space of each data page and organizes them in the tree structure. When you want to perform the 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 in current FSM meets the requirement of the INSERT operation. If so, perform the INSERT operation after the corresponding block number is returned. If not, expand the page.

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 time the

FSM is searched from the root node to leaf nodes till an available page is returned for later service 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 autovacuum. When a user executes an INSERT DML statement, such as INSERT, NON-INPLACE UPDATE (new page), or MULTI INSERT, the FSM structure is queried to find a space where the current record can be inserted. After the DML operation is complete, the system determines whether to update the free space of the current page to the FSM based on the difference between the potential free space and the actual free space of the current page. The larger the difference, that is, the more the potential space is greater than the actual space, the higher the probability that the page is updated to the FSM. FSM records the potential free space of data pages. When a user searches a page to insert data, if the free space of the page is large, the data is directly inserted. Otherwise, if the potential space of the page is large, the page is cleared and then data is inserted. If the space is insufficient, search for the FSM structure again or expand the total number of pages. Updating the FSM structure involves DML statements, page cleaning, vacuum, page expansion, partition merging, and page scanning.

6.3.2.2 UB-Tree

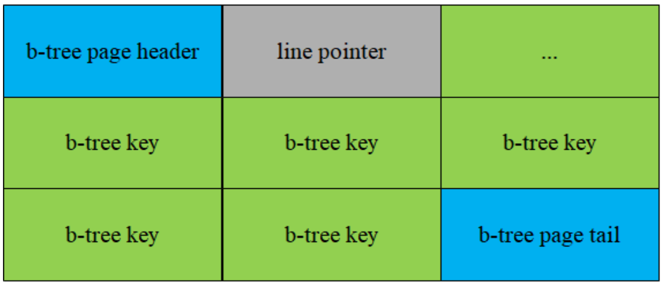
The B-tree is enhanced as follows, which is referred to as UB-tree.

- The transaction information is added to UB-tree indexes to perform MVCC independently. This increases the proportion of IndexOnlyScan and greatly reduces the number of times that TABLE ACCESS BY INDEX ROWID is executed.
- VACUUM is not the only way to clear historical data. Spaces are recycled independently. Indexes are decoupled from heap tables and can be cleared independently. The I/O stability is better.

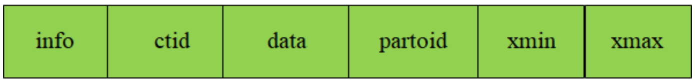
6.3.2.2.1 RCR UB-Tree

RCR UB-Tree Multi-Version Management

The multi-version management of row consistency read (RCR) B-tree is based on data row levels. XIDs are recorded in data rows, which increases the key size, and the index expands by 5% to 20%. The latest and historical versions are on the B-tree, and the index does not record the undo information. Keys are inserted or deleted in the sequence of key + TID. Tuples with the same index column are sorted based on their TIDs as the second sorting keyword, and Xmin and Xmax are added to the end of the key. During index splitting, multi-version information is migrated with key migration.

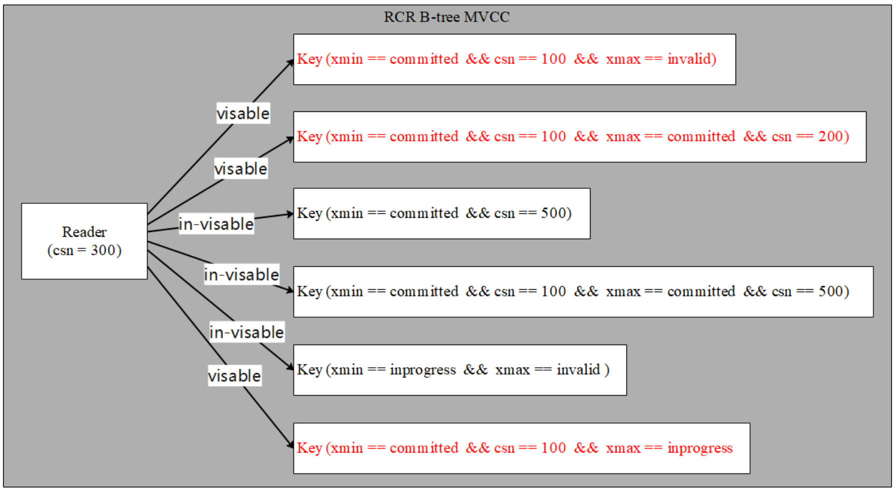


b-tree key



RCR UB-Tree Visibility Mechanism

RCR UB-tree visibility is determined by **xmin/xmax** on the key, which is similar to **xmin/xmax** on Astore heap table data rows.



Adding, Deleting, Modifying, and Querying RCR 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 to the 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, and update **active_tuple_count** on pages. If the value of **active_tuple_count** is reduced to **0**, the system attempts to recycle the page.
- **Update**: In Ustore, data updates require different processing on UB-tree index columns compared to Astore. Data updates include index column updates and non-index column updates. **Figure 6-1** illustrates the UB-tree processing during data updates.

Figure 6-1 UB-tree processing during data updates

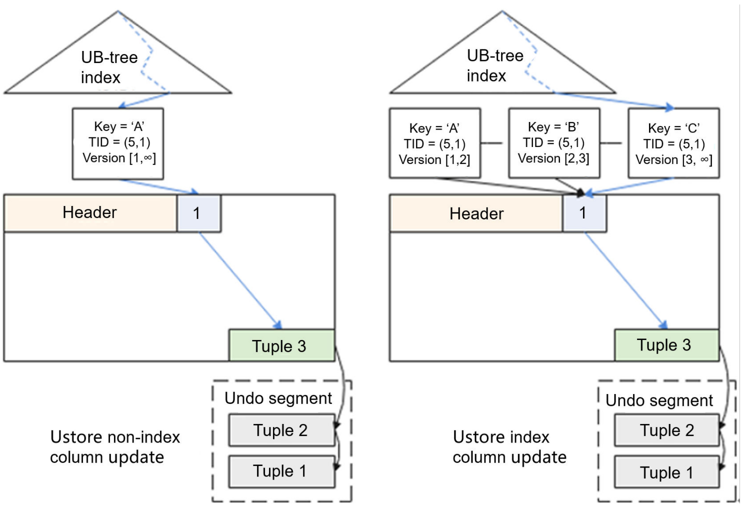


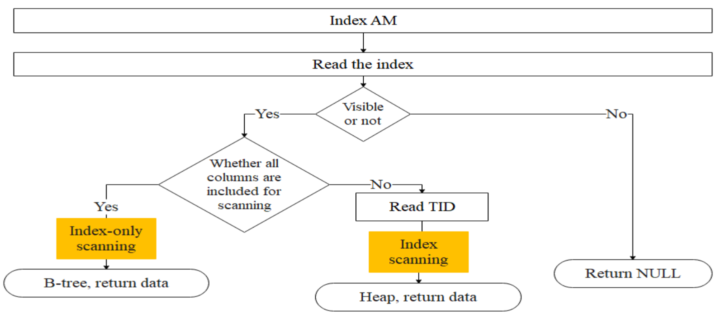
Figure 6-1 shows the differences between UB-tree updates in index columns and non-index columns.

- In the case of non-index column updates, the index does not change. The index tuple still points to the data tuple inserted for the first time. No data tuple is inserted to the Uheap. Instead, the Uheap modifies the current data tuple and saves historical data to the undo segment.
- 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 historical data, you need to read it from the undo segment.

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

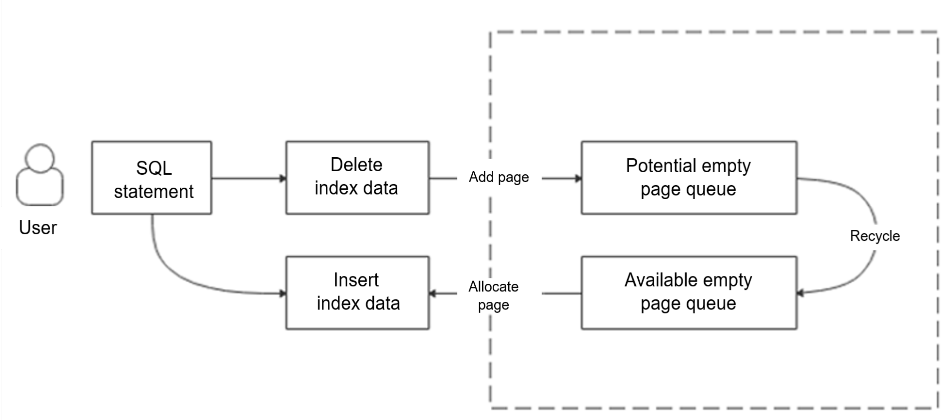
  For index scan:

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



RCR 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. **From an empty page a 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. **From the potential queue to an 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. **From the available queue to an 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.

6.3.2.2.2 PCR UB-Tree

Compared with the RCR UB-tree, the PCR (page consistency read) UB-tree has the following features:

- The transaction information of the index tuple is managed by the TD slot.

- The undo operation is added. Before insertion and deletion, the undo log needs to be written. When a transaction is aborted, the rollback operation needs to be performed.
- Flashback is supported.

When creating an index, you can set WITH option **index_txntype** to **pcr** or set the GUC parameter **index_txntype** to **pcr** to create a PCR UB-tree. If the WITH option or GUC parameter is not specified, an RCR UB-tree is created by default. The PCR UB-tree does not support online creation, ultimate RTO replay, and standby-node read.

In the current version, it may take a long time to roll back a large number of PCR indexes. (The rollback time may increase exponentially as the data volume increases. If the data volume is too large, the rollback may fail to be completely executed.) The rollback time will be optimized in later versions. The following table describes the rollback time specifications in the current version.

Table 6-1 PCR index rollback time specifications

Type/Data Volume	100	1000	10,000	100,000	1 million
Rollback time with PCR indexes	0.692 ms	9.610 ms	544.678 ms	52,963.754 ms	89,440,029.048 ms
Rollback time without PCR indexes	0.226 ms	0.916 ms	8.974 ms	94.903 ms	1206.177 ms
Ratio	3.06	10.49	60.70	558.08	74,151.66

PCR UB-Tree Multi-Version Management

Different from that of RCR UB-tree, page consistency read (PCR) multi-version management is based on pages. Transaction information of all tuples is managed by the TD slot.

PCR UB-Tree Visibility Mechanism

PCR UB-tree visibility is determined by rolling back a page to a moment when the snapshot is visible to obtain a page on which all tuples are visible.

Adding, Deleting, Modifying, and Querying PCR UB-Tree

- **Insert**: The operation is basically the same as that of RCR UB-tree. The difference is that a TD needs to be allocated and undo logs need to be written before insertion.
- **Delete**: The operation is basically the same as that of RCR UB-tree. The difference is that a TD needs to be allocated and undo logs need to be written before deletion.

- **Update**: The operation is the same as that of RCR UB-tree. That is, the operation is converted into a Delete operation and an Insert operation.
- **Scan**: The operation is basically the same as that of RCR UB-tree. The difference is that the query operation needs to copy a CR page and roll back the CR page to the state visible to the snapshot. In this way, the tuples on the entire page are visible to the snapshot.

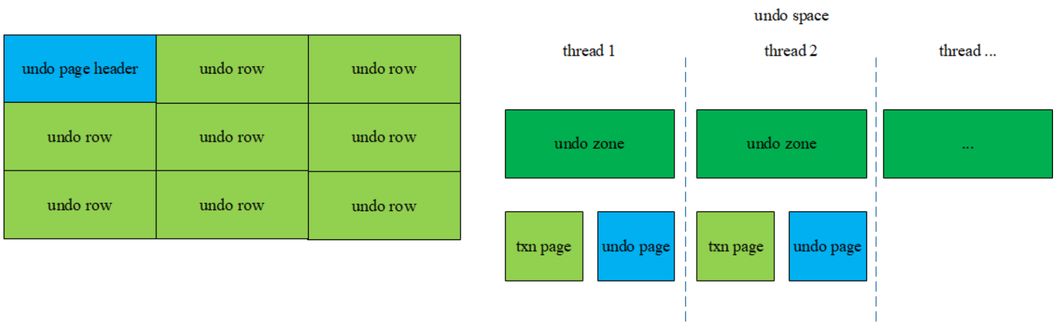
PCR UB-Tree Space Management

The space management operations are basically the same as those of RCR UB-tree. The difference is that PCR UB-tree supports flashback. Therefore, the time point when the page can be recycled is changed from **OldestXmin** to **GlobalRecycleXid**.

6.3.2.3 Undo

Historical data sets are stored in the *$node_dir***/undo** directory, in which *$node_dir* indicates the database node path. The rollback segment log is a collection of all undo logs associated with a single write transaction. Permanent, unlogged, and temp tables are supported.

6.3.2.3.1 Rollback Segment Management



1. In addition to managing some transaction pages (used to store metadata for transaction rollback), each undo zone also manages 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.

6.3.2.3.2 File Structure

To query whether the current rollback segment is stored in page or segment-page mode, query the system catalog. Currently, only the page mode is supported.

Example:

```
gaussdb=# SELECT * FROM gs_global_config where name like '%undostoragetype%';
     name       | value
-----------------+---------
 undostoragetype | page
(1 row)
```

- When the rollback segment is stored by page:

– Structure of the file where the txn page is stored
```
$node_dir/undo/{permanent|unlogged|temp}/$undo_zone_id.meta.$segno
```
– Structure of the file where the undo row is stored
```
$node_dir/undo/{permanent|unlogged|temp}/$undo_zone_id.$segno
```

6.3.2.3.3 Space Management

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

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

6.3.2.4 Enhanced TOAST

6.3.2.4.1 Overview

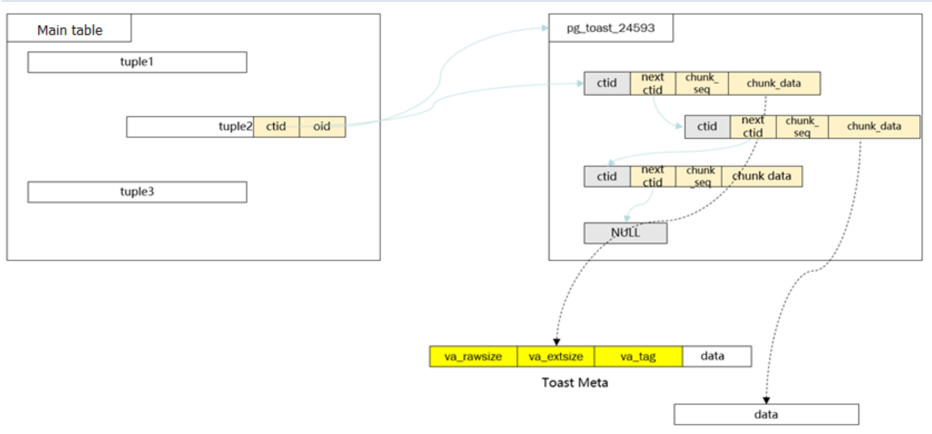
Enhanced TOAST is a technology used to process oversized fields. First, it reduces redundant information in the TOAST pointer so that more than 500 columns of oversized fields can be stored in a single table. Second, it optimizes the mapping between main tables and out-of-line storage tables so that the pg_toast_index table does not need to store such relationship, reducing user storage space. Lastly, the enhanced TOAST technology eliminates the dependency on OID allocation. Instead, it allows split data to be automatically linked, greatly improving write efficiency.

NOTE

- Astore does not support enhanced TOAST.
- No VACUUM FULL operation can be separately performed on out-of-line storage tables of the enhanced TOAST type.

6.3.2.4.2 Enhanced TOAST Storage Structure

The enhanced TOAST technology uses self-linking to handle dependencies between tuples. The out-of-line storage table divides oversized data into linked list blocks by 2 KB. The TOAST pointer of the main table points to the corresponding data linked list header of the out-of-line storage table. In this way, the mapping between the main table and the out-of-line storage table is greatly simplified, effectively improving the data write and query performance.



6.3.2.4.3 Usage of Enhanced TOAST

The new GUC parameter **enable_enhance_toast_table** is used to control the out-of-line storage structure.

- **on**: The enhanced TOAST out-of-line storage table is used.
- **off**: The TOAST out-of-line storage table is used.

```
gs_guc reload -Z coordinator -Z datanode -N all -I all -c "enable_enhance_toast_table=on"
```

6.3.2.4.4 Adding, Deleting, Modifying, and Querying Enhanced TOAST

INSERT: The write conditions for triggering enhanced TOAST are the same as those for triggering TOAST. The INSERT logic remains unchanged except that the linking information is added when data is written.

DELETE: The DELETE process of enhanced TOAST does not depend on the TOAST data index. Instead, data is traversed and deleted based on the linking information between data.

UPDATE: The UPDATE process of enhanced TOAST is the same as that of TOAST.

6.3.2.4.5 DDL Operations Related to Enhanced TOAST

Create an enhanced TOAST table.

Set the table storage type to enhanced TOAST or TOAST during table creation.

```
gaussdb=#  CREATE TABLE test_toast (id int, content text) with(toast.toast_storage_type=toast);
CREATE TABLE
gaussdb=#  \d+ test_toast
                    Table "public.test_toast"
 Column  |  Type   | Modifiers | Storage  | Stats target | Description
---------+---------+-----------+----------+--------------+-------------
 id      | integer |           | plain    |              |
 content | text    |           | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=toast
gaussdb=#  DROP TABLE test_toast;
DROP TABLE
gaussdb=#  CREATE TABLE test_toast (id int, content text) with(toast.toast_storage_type=enhanced_toast);
CREATE TABLE
gaussdb=#  \d+ test_toast
                    Table "public.test_toast"
```

```
 Column  |  Type   | Modifiers | Storage  | Stats target | Description
---------+---------+-----------+----------+--------------+-------------
 id      | integer |           | plain    |              |
 content | text    |           | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast
gaussdb=#  DROP TABLE test_toast;
DROP TABLE
```

If the type of an out-of-line storage table is not specified during table creation, the table type depends on the **enable_enhance_toast_table** parameter.

```
-- Enable GUC parameters by following the instructions provided in section "Using Enhanced TOAST."
gaussdb=#  show enable_enhance_toast_table;
 enable_enhance_toast_table
----------------------------
 on
(1 row)
gaussdb=#  CREATE TABLE test_toast (id int, content text);
CREATE TABLE
gaussdb=#   \d+ test_toast
                       Table "public.test_toast"
 Column  |  Type   | Modifiers | Storage  | Stats target | Description
---------+---------+-----------+----------+--------------+-------------
 id      | integer |           | plain    |              |
 content | text    |           | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast

gaussdb=#   DROP TABLE test_toast;
DROP TABLE
gaussdb=#  SET enable_enhance_toast_table = off;
SET
gaussdb=#  show enable_enhance_toast_table;
 enable_enhance_toast_table
----------------------------
 off
(1 row)
gaussdb=#   CREATE TABLE test_toast (id int, content text);
CREATE TABLE
gaussdb=#  \d+ test_toast
                       Table "public.test_toast"
 Column  |  Type   | Modifiers | Storage  | Stats target | Description
---------+---------+-----------+----------+--------------+-------------
 id      | integer |           | plain    |              |
 content | text    |           | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=toast
gaussdb=#  DROP TABLE test_toast;
DROP TABLE
```

Change the structure of an out-of-line storage table.

When the GUC parameter **enable_enhance_toast_table** is set to **on**, the structure of an out-of-line storage table can be changed to enhanced TOAST through the VACUUM FULL operation.

```
gaussdb=#   CREATE TABLE test_toast (id int, content text);
CREATE TABLE
```

```
gaussdb=#  \d+ test_toast
                        Table "public.test_toast"
 Column  |  Type   | Modifiers | Storage  | Stats target | Description
---------+---------+-----------+----------+--------------+-------------
 id      | integer |           | plain    |              |
 content | text    |           | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=toast
gaussdb=#  VACUUM FULL test_toast;
VACUUM
gaussdb=#  \d+ test_toast
                        Table "public.test_toast"
 Column  |  Type   | Modifiers | Storage  | Stats target | Description
---------+---------+-----------+----------+--------------+-------------
 id      | integer |           | plain    |              |
 content | text    |           | extended |              |
Has OIDs: no
Distribute By: HASH(a)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, storage_type=USTORE, segment=off,
toast.storage_type=USTORE, toast.toast_storage_type=enhanced_toast
gaussdb=#  DROP TABLE test_toast;
DROP TABLE
```

Merge partitioned tables.

Different types of out-of-line storage table partitions can be merged.

NOTE

- For out-of-line storage partitions of the same type, the merge logic remains unchanged, that is, partitions are physically merged.
- For different types of out-of-line storage partitions, the out-of-line storage table generated after partitions are merged is an enhanced TOAST table and is only logically merged. The performance is inferior to that of physical merge.

```
gaussdb=#  CREATE TABLE test_partition_table(a int, b text)PARTITION BY range(a)(partition p1 values less
than (2000),partition p2 values less than (3000));
gaussdb=# SELECT relfilenode FROM pg_partition WHERE relname='p1';
 relfilenode
-------------
      17529
(1 row)

gaussdb=#  \d+ pg_toast.pg_toast_part_17529
TOAST table "pg_toast.pg_toast_part_17529"
   Column   |  Type   | Storage
------------+---------+---------
 chunk_id   | oid     | plain
 chunk_seq  | integer | plain
 chunk_data | bytea   | plain
Options: storage_type=ustore, toast_storage_type=toast

gaussdb=# SELECT relfilenode FROM pg_partition WHERE relname='p2';
 relfilenode
-------------
      17528
(1 row)

gaussdb=#  \d+ pg_toast.pg_toast_part_17528
TOAST table "pg_toast.pg_toast_part_17528"
   Column   |  Type   | Storage
------------+---------+---------
 chunk_seq  | integer | plain
 next_chunk | tid     | plain
```

```
 chunk_data | bytea   | plain
Options: storage_type=ustore, toast_storage_type=enhanced_toast
gaussdb=#  ALTER TABLE test_partition_table MERGE PARTITIONS p1,p2 INTO partition p1_p2;
ALTER TABLE
gaussdb=#  SELECT reltoastrelid::regclass FROM pg_partition where relname='p1_p2';
       reltoastrelid
------------------------------
 pg_toast.pg_toast_part_17559
(1 row)
gaussdb=#  \d+ pg_toast.pg_toast_part_17559
TOAST table "pg_toast.pg_toast_part_17559"
   Column   |  Type   | Storage
------------+---------+---------
 chunk_seq  | integer | plain
 next_chunk | tid     | plain
 chunk_data | bytea   | plain
Options: storage_type=ustore, toast_storage_type=enhanced_toast
gaussdb=# DROP TABLE test_partition_table;
DROP TABLE
```

6.3.2.4.6 Enhanced TOAST O&M Management

Use gs_parse_page_bypath to parse the TOAST pointer information in the main table.

```
gaussdb=#  SELECT ctid,next_chunk,chunk_seq FROM pg_toast.pg_toast_part_17559;
 ctid  | next_chunk | chunk_seq
-------+------------+-----------
 (0,1) | (0,0)      |         1
 (0,2) | (0,1)      |         0
 (0,3) | (0,0)      |         1
 (0,4) | (0,3)      |         0
(4 rows)
gaussdb=#  SELECT gs_parse_page_bypath((SELECT * FROM
pg_relation_filepath('test_toast')),0,'uheap',false);
                    gs_parse_page_bypath
-------------------------------------------------------------------
 ${data_dir}/gs_log/dump/1663_13113_17603_0.page
(1 row)
```

The parsing file **1663_13113_17603_0.page** stores the TOAST pointer information as follows:

```
Toast_Pointer:
    column_index: 1
        toast_relation_oid: 17608    -- OID of the out-of-line storage table
        ctid: (4, 1)                 -- Head pointer of the out-of-line storage data linked list
        bucket id: -1                -- Bucket ID information
    column_index: 2
        toast_relation_oid: 17608
        ctid: (2, 1)
        bucket id: -1
```

Query enhanced TOAST data. You can determine the integrity of the link structure based on the queried enhanced TOAST table data.

```
gaussdb=#  SELECT ctid,next_chunk,chunk_seq FROM pg_toast.pg_toast_part_17559;
 ctid  | next_chunk | chunk_seq
-------+------------+-----------
 (0,1) | (0,0)      |         1
 (0,2) | (0,1)      |         0
 (0,3) | (0,0)      |         1
 (0,4) | (0,3)      |         0
(4 rows)
```

6.3.3 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 corresponds to a unique CSN. The CSN can mark the following states of a transaction: committing, committed, rolled back, and frozen.

6.3.3.1 Transaction Commit

The commit policies for implicit and explicit transactions are as follows:

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.

   A subtransaction must exist in an explicit transaction or stored procedure. The SAVEPOINT statement controls the start of a subtransaction, and the RELEASE SAVEPOINT statement controls the end of a subtransaction. If a subtransaction is not released during transaction committing, commit the subtransaction first. The transaction is not committed until all subtransactions are 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.

6.3.3.2 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 combines synchronous, asynchronous, and page-level rollbacks.

- **Synchronous rollback**

  Transaction rollback is triggered in any of the following scenarios:

  – The ROLLBACK keyword in a transaction block triggers a synchronous rollback.
  – If an error is reported during transaction running, the COMMIT keyword has the same function as ROLLBACK and triggers synchronous rollback.

  – If a fatal/panic error is reported during transaction running, the system attempts to synchronously roll back the transaction bound to the thread before the thread exits.

- **Asynchronous rollback**

  When the synchronous rollback fails or the system is restarted after breakdown, the undo recycling thread initiates an asynchronous rollback task for the transaction that is not rolled back completely and provides services for external systems immediately. The task initiation thread Undo Launch of asynchronous rollback starts the worker thread Undo Worker to execute the rollback task. The undo launch thread can start a maximum of five undo worker threads at the same time.

- **Page-level rollback**

  If a transaction has not been rolled back to the current page, but other transactions need to reuse the TD occupied by this transaction, the page-level rollback operation is performed for all modifications on the current page. Page-level rollback only rolls back modifications on the current page. Other pages are not involved.

  The rollback of a Ustore subtransaction is controlled by the ROLLBACK TO SAVEPOINT statement. After a subtransaction is rolled back, the transaction can continue to run. The rollback of a subtransaction does not affect the status of the transaction. If a subtransaction is not released during transaction rollback, roll back the subtransaction first. The transaction is not rolled back until all subtransactions are rolled back.

6.3.4 Flashback

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

NOTE

- Astore supports only the flashback DROP/TRUNCATE 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.

6.3.4.1 Flashback Query

Context

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

Prerequisites

The overall solution consists of three parts: earlier version retention, snapshot maintenance, and earlier version retrieval. Earlier version retention: The **undo_retention_time** parameter is added to set the retention period of an earlier version. Beyond the retention period, the earlier version will be reclaimed and deleted. To use flashback query, you must set this parameter to a value greater than **0**. Contact the administrator to change the value.

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.

Parameters

- 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 (set **undo_retention_time** to a value greater than **0**):

```
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
```

6.3.4.2 Flashback Table

Context

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

Prerequisites

The overall solution consists of three parts: earlier version retention, snapshot maintenance, and earlier version retrieval. Earlier version retention: The **undo_retention_time** parameter is added to set the retention period of an earlier version. The earlier version will be recycled and deleted after the retention period expires. For details, contact the administrator.

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 back the table to a specific timestamp before the current time.
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 back the table to a specific timestamp after the current time.
gaussdb=# TIMECAPSULE TABLE flashtest TO TIMESTAMP '2023-09-13 20:54:00.641506';
ERROR:  The specified timestamp is invalid.
gaussdb=# SELECT * FROM flashtest;
 col1 | col2
------+------
    3 | INSERT3
    6 | INSERT6
    1 | INSERT1
    2 | INSERT2
```

```
   4 | INSERT4
   5 | INSERT5
(6 rows)
gaussdb=# DROP TABLE IF EXISTS "public".flashtest;
DROP TABLE
```

6.3.4.3 Flashback DROP/TRUNCATE

Context

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

Prerequisites

- The **enable_recyclebin** parameter has been enabled (by modifying the GUC parameter in the **gaussdb.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
```

Parameters

- **DROP/TRUNCATE TABLE table_name PURGE**

  Purges table data in the recycle bin by default.
- **PURGE RECYCLEBIN**

  Purges objects in the recycle bin.

- **TO BEFORE DROP**

  Retrieves dropped tables and their subobjects from the recycle bin.

  You can specify either the original user-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 a name and multiple objects in the recycle bin contain the name, the database retrieves the recently moved objects in the recycle bin. If you want to retrieve tables of earlier versions, do as follows:
    - Specify the system-generated 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 exist" is displayed when flashback is performed on DDL operations, such as changing namespaces and table names.
  – When the **enable_recyclebin** parameter is enabled, if a table has a TRUNCATE trigger, TRUNCATE TABLE cannot activate the trigger.
- **RENAME TO**

  Specifies a new name for the table retrieved from the recycle bin.
- **TO BEFORE TRUNCATE**

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

Examples

```
-- 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 | rcybucket
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------+-----------
(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 | rcybucket
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------+-----------
(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 | rcybucket
-----------+---------+----------+------------------------------+----------------------+--------------+---------
+---------------+-------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------+-----------
     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 |    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 |
(2 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 | rcybucket
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------+-----------
(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, segment=off,toast.storage_type=ustore,
toast.toast_storage_type=enhanced_toast

-- 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 | rcybucket
-----------+---------+----------+------------------------------+---------------------+--------------+---------
+---------------+------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------+-----------
     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 |    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 |
(3 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 | rcybucket
-----------+---------+----------+------------------------------+---------------------+--------------+---------
+---------------+------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------+-----------
     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 |    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 |
(2 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 | rcybucket
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------+-----------
(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 | rcybucket
-----------+---------+----------+------------------------------+----------------------+--------------+---------
+---------------+-------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------+-----------
     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 |    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 |
(2 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.
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 | rcybucket
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
```

```
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------+-----------
(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 | rcybucket
-----------+---------+----------+------------------------------+------------------------------+--------------+---------
+---------------+-------------------------------+--------------+----
----------+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------+-----------
     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 | rcybucket
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------+-----------
(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 | rcybucket
-----------+---------+----------+------------------------------+------------------------------+--------------+---------
+---------------+-------------------------------+--------------+----
----------+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------+-----------
     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 | rcybucket
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------+-----------
(0 rows)
-- Drop the 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 | rcybucket
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
-+----------------+---------------+-------------+--------------+----------------+-----------
(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 | rcybucket
-----------+---------+----------+------------------------------+----------------------+--------------+---------
+---------------+-------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------+-----------
     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 |
(2 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 | rcybucket
-----------+---------+----------+------------------------------+----------------------+--------------+---------
+---------------+-------------------------------+--------------+------------
--+--------------+----------+---------------+----------------+---------------+-------------+--------------
+----------------+-----------
     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 |
(2 rows)

-- Drop the 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 | rcybucket
-----------+---------+----------+---------+---------------+--------------+---------+---------------+----------------
+--------------+--------------+--------------+----------+--------------
```

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

6.3.5 Common View Tools

View Type	Type	Function	Application Scenario	Function
Parsing	All types	Parses a specified table page and returns the path for storing the parsed content.	● Page information viewing ● Tuple (non-user data) information ● Damaged pages and tuples ● Tuple visibility problems ● Verification errors	gs_parse_page_bypath
	Index recycle queue (URQ)	Parses key information in the URQ.	● UB-tree index space expansion ● UB-tree index space reclamation exceptions ● Verification errors	gs_urq_dump_stat

View Type	Type	Function	Application Scenario	Function
	Rollback segment (undo)	Parses the specified undo record, excluding old tuple data.	• Undo space bloat • Undo recycling exceptions • Rollback exceptions • Routine maintenance • Verification errors • Visibility judgment exceptions • Parameter modifications	gs_undo_dump_record
		Parses all undo records generated by a specified transaction, excluding old tuple data.		gs_undo_dump_xid
		Parses all information about transaction slots in a specified undo zone.		gs_undo_translot_dump_slot
		Parses the transaction slot information of a specified transaction, including the XID and the range of undo records generated by the transaction.		gs_undo_translot_dump_xid
		Parses the metadata of a specified undo zone and displays the pointer usage of undo records and transaction slots.		gs_undo_meta_dump_zone
		Parses the undo space metadata corresponding to a specified undo zone and displays the file usage of undo records.		gs_undo_meta_dump_spaces
		Parses the slot space metadata corresponding to a specified undo zone and displays the file usage of transaction slots.		gs_undo_meta_dump_slot
		Parses the data page and all data of historical versions and returns the path for storing the parsed content.		gs_undo_dump_parsepage_mv
	Write ahead log (WAL)	Parses Xlog within the specified LSN range and returns the path for storing parsed content. You can use **pg_current_xlog_location()** to obtain the current Xlog position.	• WAL errors • Log replay errors • Damaged pages	gs_xlogdump_lsn
		Parses Xlog of a specified XID and returns the path for storing parsed content. You can use **txid_current()** to obtain the current XID.		gs_xlogdump_xid
		Parses logs corresponding to a specified table page and returns the path for storing the parsed content.		gs_xlogdump_tablepath

<table>
<tr><th>View Type</th><th>Type</th><th>Function</th><th>Application Scenario</th><th>Function</th></tr>
<tr><td></td><td></td><td>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.</td><td></td><td>gs_xlogdump_parsepage_tablepath</td></tr>
<tr><td rowspan="4">Collecting</td><td>Rollback segment (undo)</td><td>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.</td><td>● Undo space bloat
● Undo resource monitoring</td><td>gs_stat_undo</td></tr>
<tr><td rowspan="3">Write ahead log (WAL)</td><td>Collects statistics of the memory status table when WALs are written to disks.</td><td rowspan="3">● WAL write/disk flushing monitoring
● Suspended WAL write/disk flushing</td><td>gs_stat_wal_entrytable</td></tr>
<tr><td>Collects WAL statistics about the disk flushing status and location.</td><td>gs_walwriter_flush_position</td></tr>
<tr><td>Collects WAL statistic about the frequency of disk flushing, data volume, and flushing files.</td><td>gs_walwriter_flush_stat</td></tr>
<tr><td rowspan="3">Validation</td><td rowspan="2">Heap table/ Index</td><td>Checks whether the disk page data of tables or index files is normal through offline verification.</td><td>● Damaged pages and tuples
● Visibility issues
● Log replay errors</td><td>ANALYZE VERIFY</td></tr>
<tr><td>Checks whether physical files of the current database in the current instance are lost.</td><td>Lost files</td><td>gs_verify_data_file</td></tr>
<tr><td>Index recycle queue (URQ)</td><td>Checks whether the data of the URQ (potential queue/available queue/single page) is normal.</td><td>● UB-tree index space expansion
● UB-tree index space reclamation exceptions</td><td>gs_verify_urq</td></tr>
</table>

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

View Type	Type	Function	Application Scenario	Function
		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

6.3.6 Common Problems and Troubleshooting Methods

6.3.6.1 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.

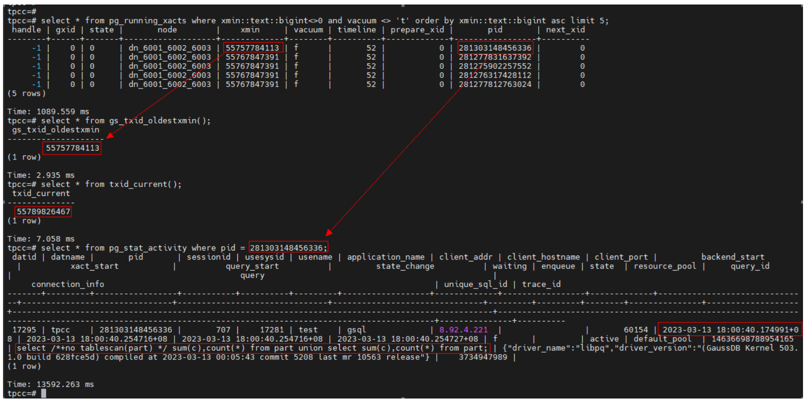
6.3.6.1.1 Undo Space Recycling Blocked by Long Transactions

Symptom

1. The following error information is printed in gs_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 long transactions are suspended and recycled.

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



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.)

6.3.6.1.2 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 **gaussdb.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**.

6.3.6.2 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 **gs_log**. The page is restored to the status before the modification after rollback.

Symptom

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

Solution

Contact Huawei technical support.

6.3.6.3 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

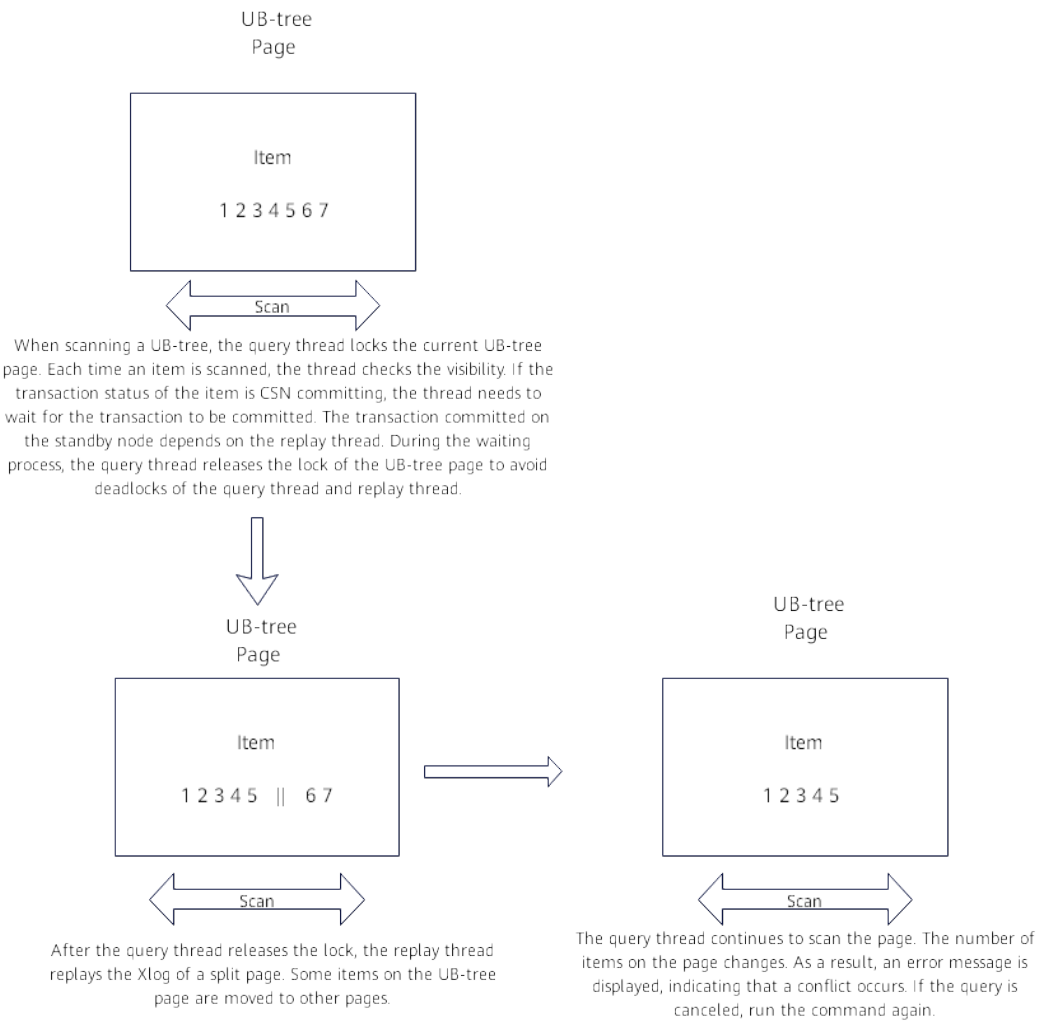
When parallel or serial replay is enabled (if the GUC parameters **recovery_parse_workers** and **recovery_max_workers** are both set to **1**, serial replay is enabled; if **recovery_parse_workers** is set to **1** and **recovery_max_workers** is greater than 1, parallel replay is enabled): If the query 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. Transactions committed on the standby node depend 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. Figure 1 shows the process.

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 6-2 Analysis



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.

6.3.6.4 Write Performance Deteriorates Occasionally When a Large Number of Concurrent Updates Are Performed During Long Query Execution

Symptom

When a table-level full scan long query is executed, a large number of concurrent page updates occur during the scan. As a result, the write performance of some DML statements deteriorates.

Analysis

For a long query (for example, more than two hours) in the full table scan scenario, before a page is scanned, a large number of concurrent updates (for example, more than 100,000 updates) occur on the page. When the page is scanned later, a large number of historical versions need to be accessed to obtain visible tuples (based on the MVCC mechanism). Because a page read lock is held during single-page scan, if the page needs to be written at this time, the write is blocked until read on the page tuple is complete.

Troubleshooting

1. Check whether long queries and DML statements canceled due to timeout exist based on slow SQL alarms and the statement_history view.
2. Check the details of the canceled DML statements queried from statement_history in **1**, use the statement_detail_decode system function to parse the **details** field, and obtain the wait events. If the top wait event is BufferContentLock, there is a high probability that this problem occurs.

Solution

Prevention: Do not perform full table scan long queries on a table involving high-concurrency operations. You are advised to perform long queries on the standby node.

Handling: Check whether this scenario is triggered based on slow SQL alarms. You can interrupt long queries to avoid continuous impact on services.

6.4 Data Lifecycle Management: OLTP Table Compression

6.4.1 Introduction

OLTP table compression is a feature of GaussDB advanced compression.

OLTP table compression allows to selectively compress data rows that are less frequently accessed using common compression algorithms based on user-defined hot and cold separation rules, achieving capacity control with the minimum service intrusion.

6.4.2 Restrictions

- System catalogs, MOTs, global temporary tables, local temporary tables, sequence tables, Ustore segment-page tables, unlogged tables, and compressed TOAST data are not supported.
- This feature is valid only in ORA-compatible, PG-compatible, or B-compatible mode.
- Ustore does not support encoding and decoding, thus offering a compression ratio lower than Astore.
- When compression is enabled for ordinary tables, the space reserved for scale-out must be evaluated based on the size after decompression.
- Compression scheduling is not supported during scale-out.
- Before scale-out, check whether a compression task is being executed. If so, wait until the compression task is complete, or run **DBE_ILM.STOP_ILM** or **DBE_ILM_ADMIN.DISABLE_ILM** to stop the compression task. After the scale-out is complete, run **DBE_ILM_ADMIN.ENABLE_ILM** to enable it.
- The partial decompression feature is added to the current version. As a result, some small tables are not compressed because their compression gain does not meet requirements. However, they can be compressed in earlier versions without the partial decompression feature. For example, a table containing only one column of the int1 type cannot be compressed since the current version.

6.4.3 Feature Specifications

- If the TPC-C policy is enabled but scheduling is disabled, existing services are not affected.
- If the TPC-C compression policy is disabled, existing services are not affected.
- If the ILM policy is set by using **TPCC.bmsql_order_line** (only orders that have been delivered are identified as cold rows) but scheduling is disabled, the tpmC deterioration is not higher than 2% (taking the 56-core CPU, 370 GB memory, 3 TB SSD, and 350 GB shared buffer as an example).
- If the ILM policy is set by using **TPCC.bmsql_order_line** (only orders that have been delivered are identified as cold rows) but parameter scheduling is enabled by default, the tpmC deterioration is not higher than 5% (taking the 56-core CPU, 370 GB memory, 3 TB SSD, and 350 GB shared buffer as an example).
- The processing rate of a single-thread ILM job is about 100 MB/s (taking the 56-core CPU, 370 GB memory, 3 TB SSD, and 350 GB shared buffer as an example).

  The processing rate can be measured based on the start time and end time of compression and the number of compressed pages.
- When GET is used to query data, the performance of accessing compressed data deteriorates compared with that of accessing non-compressed data. The performance deterioration on the driver side is not higher than 10%, and that on the PL/SQL side is not higher than 15% (taking the 32 MB shared buffer and 60,000 data pages as an example).
- When multi-get is used to query data, the performance of accessing compressed data deteriorates compared with that of accessing non-

compressed data. The performance deterioration on the driver side is not higher than 30%, and that on the PL/SQL side is not higher than 40% (taking the 32 MB shared buffer and 60,000 data pages as an example).

- When table-scan is used to query data, the performance of accessing compressed data deteriorates compared with that of accessing non-compressed data. The performance deterioration on the driver side is not higher than 30%, and that on the PL/SQL side is not higher than 40% (taking the 32 MB shared buffer and 60,000 data pages as an example).
- The compression ratio of the **TPCH.lineitem** table (all cold rows) is greater than or equal to 2:1.
- Tests on the **Orderline** table of TPC-C and the **Lineitem**, **Orders**, **Customer**, and **Part tables** of TPC-H show that the compression ratio is higher than that of LZ4 and ZLIB when there are many numeric columns; when there are a large number of text columns, the compression ratio is between that of the compression algorithms of the LZ class and the LZ+Huffman class.

6.4.4 Usage Guide

Step 1 Enable the compression function:

```
gaussdb=# ALTER DATABASE SET ilm = on;
```

Check whether **gsilmpolicy_seq** and **gsilmtask_seq** exist in the public schema of the current database.

```
gaussdb=# \d
                                    List of relations
 Schema |      Name       |   Type   |  Owner   |                    Storage
--------+-----------------+----------+----------+------------------------------------------------------
 public | gsilmpolicy_seq | sequence | omm |
 public | gsilmtask_seq   | sequence | omm |
```

or

```
gaussdb=# SELECT a.oid, a.relname FROM pg_class a inner join pg_namespace b on a.relnamespace = b.oid
    WHERE (a.relname = 'gsilmpolicy_seq' OR a.relname = 'gsilmtask_seq') AND b.nspname = 'public';

  oid  |     relname
-------+-----------------
 17002 | gsilmpolicy_seq
 17004 | gsilmtask_seq
(2 rows)
```

If an exception occurs, a warning is reported.

```
WARNING:  ILM sequences are already existed while initializing
```

Step 2 Add a compression policy for a table.

- Create a table with a policy.
```
gaussdb=# CREATE TABLE ilm_table_1 (col1 int, col2 text)
    ilm add policy row store compress advanced row
    after 3 days of no modification on (col1 < 1000);
```
- Add a policy for an existing table.
```
gaussdb=# CREATE TABLE ilm_table_2 (col1 int, col2 text);
gaussdb=# ALTER TABLE ilm_table_2 ilm add policy row store
    compress advanced row after 3 days of no modification;
```
- Check whether data is added to the policy view.
```
gaussdb=# SELECT * FROM gs_my_ilmpolicies;

 policy_name |  policy_type  | tablespace | enabled | deleted
-------------+---------------+------------+---------+---------
```

```
 p1          | DATA MOVEMENT |             | YES      | NO
 p2          | DATA MOVEMENT |             | YES      | NO
(2 rows)
```

- Check whether the policy that meets the settings is added to the policy details view.

```
gaussdb=# SELECT * FROM gs_my_ilmdatamovementpolicies;

 policy_name | action_type | scope | compression_level | tier_tablespace | tier_status |
condition_type      | condition_days | custom_function | policy_subtype | action_clause | tier_to
-------------+-------------+-------+-------------------+-----------------+-------------+-----------------------
+----------------+----------------+----------------+----------------+---------
 p1          | COMPRESSION | ROW   | ADVANCED          |                 |             | LAST MODIFICATION
TIME |              3 |                 |                |               |
 p2          | COMPRESSION | ROW   | ADVANCED          |                 |             | LAST MODIFICATION
TIME |              3 |                 |                |               |
(2 rows)
```

- Check whether the policy corresponds to the target table.

```
gaussdb=# SELECT * FROM gs_my_ilmobjects;
 policy_name | object_owner | object_name | subobject_name | object_type |     inherited_from     |
tbs_inherited_from | enabled | deleted
-------------+--------------+-------------+----------------+-------------+----------------------
+--------------------+---------+---------
 p1          | public       | ilm_table_1 |                | TABLE       | POLICY NOT INHERITED |                    |
YES      | NO
 p2          | public       | ilm_table_2 |                | TABLE       | POLICY NOT INHERITED |                    |
YES      | NO
(2 rows)
```

Step 3 Perform compression evaluation.

- Manually perform compression evaluation.

CAUTION

To facilitate the test, the **POLICY_TIME** attribute is provided in the environment parameters of this function to determine whether the time unit is day or second. Run the following statement:

```
gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
```

Insert random data for testing:

```
gaussdb=# INSERT INTO ilm_table_1 select *, 'test_data' FROM generate_series(1, 10000);

gaussdb=# DECLARE
    v_taskid number;
gaussdb=# BEGIN
    DBE_ILM.EXECUTE_ILM(OWNER           => 'public',
                        OBJECT_NAME     => 'ilm_table_1',
                        TASK_ID         => v_taskid,
                        SUBOBJECT_NAME  => NULL,
                        POLICY_NAME     => 'ALL POLICIES',
                        EXECUTION_MODE  => 2);
    RAISE INFO 'Task ID is:%', v_taskid;
gaussdb=# END;
/
```

If the input parameter is incorrect, an error message is displayed. If no error occurs, no information is displayed. (The RAISE INFO statement is added to the preceding code segment to print the ID of the current task.)

```
INFO:  Task ID is:1
```

Check the task information.

```
gaussdb=# SELECT * FROM gs_my_ilmtasks;
```

```
task_id | task_owner |   state   |         creation_time         |          start_time           |
completion_time
---------+------------+-----------+-------------------------------+-------------------------------
+-------------------------------
       1 | omm| COMPLETED | 2023-08-29 17:36:38.779555+08 | 2023-08-29 17:36:38.779555+08 |
2023-08-29 17:36:38.879485+08
(1 row)
```

Check the evaluation result.

```
gaussdb=# SELECT * FROM gs_my_ilmevaluationdetails;

 task_id | policy_name | object_owner | object_name | subobject_name | object_type |
selected_for_execution |      job_name      | comments
---------+-------------+--------------+-------------+----------------+-------------+------------------------
+-------------------+----------
       1 | p1          | public       | ilm_table_1 |                | TABLE       | SELECTED FOR EXECUTION | ilmjob
$_postgres1 |
(1 row)
```

Check the compression job information.

```
gaussdb=# SELECT * FROM gs_my_ilmresults;

 task_id |     job_name      |       job_state        |          start_time           |        completion_time        |
comments |                statistics
---------+-------------------+------------------------+-------------------------------
+-------------------------------+----------+---------------------------------------
       1 | ilmjob$_postgres1 | COMPLETED SUCCESSFULLY | 2023-08-29 17:36:38.779555+08 |
2023-08-29 17:36:38.879485+08 |          | SpaceSaving=0,BoundTime=0,LastBlkNum=0
(1 row)
```

- Trigger automatic scheduling evaluation in the background.

  Log in to the **template1** database as the initial user and create a maintenance window.

```
gaussdb=#
DECLARE
    V_HOUR INT := 22;
    V_MINUTE INT := 0;
    V_SECOND INT := 0;
    C_ADO_WINDOW_SCHEDULE_NAME TEXT := 'ado_window_schedule';
    C_ADO_WINDOW_PROGRAM_NAME TEXT := 'ado_window_program';
    C_MAINTENANCE_WINDOW_JOB_NAME TEXT := 'maintenance_window_job';
    V_MAINTENCE_WINDOW_REPEAT TEXT;
    V_MAINTENCE_WINDOW_START TIMESTAMPTZ;
    V_BE_SCHEDULE_ENABLE BOOL;
    V_MAINTENANCE_WINDOW_EXIST INT;
BEGIN
    SELECT COUNT(*) INTO V_MAINTENANCE_WINDOW_EXIST FROM PG_CATALOG.PG_JOB WHERE
JOB_NAME = 'maintenance_window_job' AND DBNAME = 'template1';
    IF CURRENT_DATABASE() != 'template1' THEN
        RAISE EXCEPTION 'Create maintenance_window FAILED, current database is not tempalte1';
    END IF;
    IF V_MAINTENANCE_WINDOW_EXIST = 0 AND CURRENT_DATABASE() = 'template1' THEN
        SELECT
            CASE
                WHEN NOW() < CURRENT_DATE + INTERVAL '22 HOUR' THEN CURRENT_DATE +
INTERVAL '22 HOUR'
                ELSE CURRENT_DATE + INTERVAL '1 DAY 22 HOUR'
            END INTO V_MAINTENCE_WINDOW_START;
        --1. prepare for maintence window schedule
        SELECT 'freq=daily;interval=1;byhour='||V_HOUR||';byminute='||V_MINUTE||';bysecond='||
V_SECOND INTO V_MAINTENCE_WINDOW_REPEAT;
        BEGIN
            SELECT
                CASE
                    WHEN VALUE = 1 THEN TRUE -- DBE_ILM_ADMIN.ILM_ENABLED
                    ELSE FALSE
                END INTO V_BE_SCHEDULE_ENABLE
                FROM PG_CATALOG.GS_ILM_PARAM WHERE IDX = 7; -- DBE_ILM_ADMIN.ENABLED
        EXCEPTION
```

```
            WHEN OTHERS THEN
                V_BE_SCHEDULE_ENABLE := FALSE;
        END;
        --2. Create ado window schedule
        DBE_SCHEDULER.CREATE_SCHEDULE(
            SCHEDULE_NAME => C_ADO_WINDOW_SCHEDULE_NAME,
            START_DATE => '9999-01-01 00:00:01',
            REPEAT_INTERVAL => NULL,
            END_DATE => NULL,
            COMMENTS => 'ado window schedule');
        --3. Create ado window program
        DBE_SCHEDULER.CREATE_PROGRAM(
            PROGRAM_NAME => C_ADO_WINDOW_PROGRAM_NAME,
            PROGRAM_TYPE => 'plsql_block',
            PROGRAM_ACTION => 'call prvt_ilm.be_execute_ilm(0);',
            NUMBER_OF_ARGUMENTS => 0,
            ENABLED => TRUE,
            COMMENTS => NULL);
        --4. Create maintenance window master job
        DBE_SCHEDULER.CREATE_JOB(
            JOB_NAME => C_MAINTENANCE_WINDOW_JOB_NAME,
            START_DATE => V_MAINTENCE_WINDOW_START,
            REPEAT_INTERVAL => V_MAINTENCE_WINDOW_REPEAT,
            END_DATE => NULL,
            JOB_TYPE => 'STORED_PROCEDURE'::TEXT,
            JOB_ACTION => 'prvt_ilm.be_active_ado_window'::TEXT,
            NUMBER_OF_ARGUMENTS => 0,
            ENABLED => V_BE_SCHEDULE_ENABLE,
            AUTO_DROP => FALSE,
            COMMENTS => 'maintenance window job',
            destination_name=> 'CCN');
    ELSE
        RAISE EXCEPTION 'CREATE ILM MAINTENANCE WINDOW FAILED';
    END IF;
END;
```

Automatic scheduling provides the following parameters for adjustment:

```
gaussdb=# SELECT * FROM gs_adm_ilmparameters;
              name               | value
---------------------------------+-------
 EXECUTION_INTERVAL              |    15
 RETENTION_TIME                  |    30
 ENABLED                         |     1
 POLICY_TIME                     |     0
 ABS_JOBLIMIT                    |    10
 JOB_SIZELIMIT                   |  1024
 WIND_DURATION                   |   240
 BLOCK_LIMITS                    |    40
 ENABLE_META_COMPRESSION         |     0
 SAMPLE_MIN                      |    10
 SAMPLE_MAX                      |    10
 CONST_PRIO                      |    40
 CONST_THRESHOLD                 |    90
 EQVALUE_PRIO                    |    60
 EQVALUE_THRESHOLD               |    80
 ENABLE_DELTA_ENCODE_SWITCH      |     1
 LZ4_COMPRESSION_LEVEL           |     0
 ENABLE_LZ4_PARTIAL_DECOMPRESSION |     1
(18 rows)
```

– **EXECUTION_INTERVAL**: interval for executing the automatic scheduling task. By default, the task is executed every 15 minutes.
– **RETENTION_TIME**: interval for deleting historical compression task records. By default, historical compression task records are deleted every 30 days.
– **ENABLED**: specifies whether automatic scheduling is enabled. The default value is **ENABLED**.

- **POLICY_TIME**: time unit for policy evaluation, which is used for tests. The default unit is day.
- **ABS_JOBLIMIT**: maximum number of compression tasks generated in a single evaluation. The default value is **10**.
- **JOB_SIZELIMIT**: I/O upper limit of a single compression task. The default value is 1 GB.
- **WIND_DURATION**: duration of a single maintenance window.
- **BLOCK_LIMITS**: upper limit of the instance-level row-store compression speed. The default value is **40**. The value ranges from 0 to 10000, in block/ms, indicating the maximum number of blocks that can be compressed per millisecond. **0** indicates that there is no speed limit. The maximum speed is calculated as follows: **BLOCK_LIMITS** x 1000 x **BLOCKSIZE**. For example, if the default value is **40**, the maximum speed is 320,000 KB/s (40 x 1000 x 8 KB).
- **ENABLE_META_COMPRESSION**: specifies whether to enable header compression. The default value is **0**. The value can be **0** (disabled) or **1** (enabled).

  NOTE

  If this parameter is set to **1**, the compression ratio is improved for tables with short rows (less data in a single row), but the performance of accessing compressed rows greatly deteriorates. If most tables in the database have long rows, you are advised not to enable this parameter.
- **SAMPLE_MIN**: minimum sampling step for constant encoding and equivalent encoding. The default value is **10**. The value range is [1,100]. Decimals can be entered and will be automatically rounded down.
- **SAMPLE_MAX**: maximum sampling step for constant encoding and equivalent encoding. The default value is **10**. The value range is [1,100]. Decimals can be entered and will be automatically rounded down.
- **CONST_PRIO**: constant encoding priority. The default value is **40**. The value range is [0,100]. The value **100** indicates that constant encoding is disabled. Decimals can be entered and will be automatically rounded down.
- **CONST_THRESHOLD**: constant encoding threshold. The default value is **90**. The value range is [1,100], indicating that constant encoding is performed when the proportion of constant values in a column exceeds the threshold. Decimals can be entered and will be automatically rounded down.
- **EQVALUE_PRIO**: equivalent encoding priority. The default value is **60**. The value range is [0,100]. The value **100** indicates that equivalent encoding is disabled. Decimals can be entered and will be automatically rounded down.
- **EQVALUE_THRESHOLD**: equivalent encoding threshold. The default value is **80**. The value range is [1,100], indicating that equivalent encoding is performed when the ratio of equal values in two columns exceeds the threshold. Decimals can be entered and will be automatically rounded down.
- **ENABLE_DELTA_ENCODE_SWITCH**: specifies whether to enable difference encoding. The default value is **1**. The value **0** indicates that difference encoding is disabled, and the value **1** indicates that difference

encoding is enabled. Decimals can be entered and will be automatically rounded down.

- **LZ4_COMPRESSION_LEVEL**: LZ4 compression level. The default value is **0**. The value range is [0,16]. Decimals can be entered and will be automatically rounded down.
- **ENABLE_LZ4_PARTIAL_DECOMPRESSION**: specifies whether to enable partial decompression. The default value is **1**. The value **0** indicates that partial decompression is disabled, and the value **1** indicates that partial decompression is enabled. Decimals can be entered and will be automatically rounded down.

The preceding parameters can be adjusted through the DBE_ILM_ADMIN.CUSTOMIZE_ILM() API.

By default, the maintenance window is opened at 22:00 (Beijing time) every day. You can use the SET_ATTRIBUTE API provided by the DBE_SCHEDULER to set the maintenance window.

```
\c template1
 CALL DBE_ILM_ADMIN.DISABLE_ILM();
 CALL DBE_ILM_ADMIN.ENABLE_ILM();
DECLARE
    newtime timestamptz := CLOCK_TIMESTAMP() + to_interval('2 seconds');
BEGIN
    DBE_SCHEDULER.set_attribute(
        name          =>          'maintenance_window_job',
        attribute     =>          'start_date',
        value         =>          TO_CHAR(newtime, 'YYYY-MM-DD HH24:MI:SS')
    );
END;
/
```

----End

6.4.5 Setting the Maintenance Window Parameters

- **RETENTION_TIME**: duration for storing evaluation and compression records, in days. The default value is **30**. You can adjust the value based on your storage capacity.
- **EXECUTION_INTERVAL**: execution frequency of an evaluation task, in minutes. The default value is **15**. You can adjust the value based on the service and resource usage during the maintenance window. This parameter and **ABS_JOBLIMIT** affect each other. A single thread can generate a maximum of **WIND_DURATION/EXECUTION_INTERVAL** x **JOB_SIZELIMIT** I/Os per day.
- **JOB_SIZELIMIT**: maximum number of bytes that can be processed by a single compression job. The unit is MB. The default value is **1024**. The compression speed is about 100 MB/s. When the I/O limit of each compression job is 1 GB, the compression job can be completed within 10 seconds. You can adjust the value based on the service off-peak hours and the amount of data to be compressed.
- **ABS_JOBLIMIT**: maximum number of compression jobs that can be generated in an evaluation. You can adjust the value based on the number of partitions and tables in the configured policy. It is recommended that the value be less than or equal to 10. You can run the **select count(*) from gs_adm_ilmobjects where enabled = true** command to query the value.

- **POLICY_TIME**: specifies whether the time unit for determining cold rows is day or second. The time unit second is used only for testing. The value can be **ILM_POLICY_IN_SECONDS** or **ILM_POLICY_IN_DAYS** (default value).
- **WIND_DURATION**: maintenance window duration, in minutes. The default value is 240 minutes (4 hours). By default, the maintenance window lasts 240 minutes from 22:00 (Beijing time). You can adjust the maintenance window based on service off-peak hours.
- **BLOCK_LIMITS**: upper limit of the instance-level row-store compression speed. The default value is **40**. The value ranges from 0 to 10000, in block/ms, indicating the maximum number of blocks that can be compressed per millisecond. **0** indicates that there is no speed limit. The maximum speed is calculated as follows: **BLOCK_LIMITS** x 1000 x **BLOCKSIZE**. For example, if the default value is **40**, the maximum speed is 320,000 KB/s (40 x 1000 x 8 KB).
- **ENABLE_META_COMPRESSION**: specifies whether to enable header compression. The default value is **0**. The value can be **0** (disabled) or **1** (enabled). You can enable or disable this function as required.
- **SAMPLE_MIN**: minimum sampling step for constant encoding and equivalent encoding. The default value is **10**. The value range is [1,100]. Decimals can be entered and will be automatically rounded down. You can set this parameter as required.
- **SAMPLE_MAX**: maximum sampling step for constant encoding and equivalent encoding. The default value is **10**. The value range is [1,100]. Decimals can be entered and will be automatically rounded down. You can set this parameter as required.
- **CONST_PRIO**: constant encoding priority. The default value is **40**. The value range is [0,100]. The value **100** indicates that constant encoding is disabled. Decimals can be entered and will be automatically rounded down. You can set this parameter as required.
- **CONST_THRESHOLD**: constant encoding threshold. The default value is **90**. The value range is [1,100], indicating that constant encoding is performed when the proportion of constant values in a column exceeds the threshold. Decimals can be entered and will be automatically rounded down. You can set this parameter as required.
- **EQVALUE_PRIO**: equivalent encoding priority. The default value is **60**. The value range is [0,100]. The value **100** indicates that equivalent encoding is disabled. Decimals can be entered and will be automatically rounded down. You can set this parameter as required.
- **EQVALUE_THRESHOLD**: equivalent encoding threshold. The default value is **80**. The value range is [1,100], indicating that equivalent encoding is performed when the ratio of equal values in two columns exceeds the threshold. Decimals can be entered and will be automatically rounded down. You can set this parameter as required.
- **ENABLE_DELTA_ENCODE_SWITCH**: specifies whether to enable difference encoding. The default value is **1**. The value **0** indicates that difference encoding is disabled, and the value **1** indicates that difference encoding is enabled. Decimals can be entered and will be automatically rounded down. You can set this parameter as required.
- **LZ4_COMPRESSION_LEVEL**: LZ4 compression level. The default value is **0**. The value range is [0,16]. Decimals can be entered and will be automatically rounded down. You can set this parameter as required.

- **ENABLE_LZ4_PARTIAL_DECOMPRESSION**: specifies whether to enable partial decompression. The default value is **1**. The value **0** indicates that partial decompression is disabled, and the value **1** indicates that partial decompression is enabled. Decimals can be entered and will be automatically rounded down. You can enable or disable this function as required.

Example:

```
EXECUTION_INTERVAL: 15
JOB_SIZELIMIT: 10240
WIND_DURATION: 240
BLOCK_LIMITS: 0
```

In this configuration, for a single table partition, a total of 160 GB (240/15 x 10240 MB) data can be evaluated and compressed during a maintenance window. The compression speed is 100 MB/s, but the actual compression takes only 27 minutes (160 GB/(100 MB/s) = 27 minutes). Services are not affected in other periods. You can adjust the parameters based on the available compression period during off-peak hours.

6.4.6 O&M Command Reference

1. Manually trigger compression once. (In the example, 102400 MB data is compressed once.)
   a. Add the cold and hot separation policy for a table.
   ```
   gaussdb=#  DROP TABLE IF EXISTS ILM_TABLE;
   gaussdb=#  CREATE TABLE ILM_TABLE(a int);
   gaussdb=# ALTER TABLE ILM_TABLE ILM ADD POLICY ROW STORE COMPRESS ADVANCED
      ROW AFTER 3 MONTHS OF NO MODIFICATION;
   ```
   b. Manually trigger compression.
   ```
   DECLARE
     v_taskid number;
   BEGIN
     DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
   DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 102400);
     DBE_ILM.EXECUTE_ILM(OWNER         => '$schema_name',
                     OBJECT_NAME     => 'ilm_table',
                     TASK_ID         => v_taskid,
                     SUBOBJECT_NAME => NULL,
                     POLICY_NAME     => 'ALL POLICIES',
                     EXECUTION_MODE => 2);
     RAISE INFO 'Task ID is:%', v_taskid;
   END;
   /
   ```
   c. Check whether the compression job is complete. You can view the detailed execution information.
   ```
   gaussdb=# SELECT * FROM gs_adm_ilmresults ORDER BY task_id desc;

    task_id |                 job_name                 |     start_time      |   completion_time
   |                        statistics
   ---------+-----------------------------------------+---------------------+---------------------
   +--------------------------------------------------------
      17267 | ilmjob$_2 | 2023-03-29 08:11:25 | 2023-03-29 08:11:25 |
   SpaceSaving=453048,BoundTime=1680145883,LastBlkNum=128
   ```
2. Manually stop compression.
   ```
   gaussdb=# DBE_ILM.STOP_ILM (task_id => V_TASK, p_drop_running_Jobs => FALSE, p_Jobname =>
   V_JOBNAME);
   ```

Table 6-2 DBE_ILM.STOP_ILM input parameters

Name	Description
TASK_ID	Specifies the descriptor ID of the ADO task to be stopped.
P_DROP_RUNNING_JOBS	Determines whether to stop a task that is being executed. The value **true** indicates that the task is forcibly stopped, and the value **false** indicates that the task is not stopped.
P_JOBNAME	Specifies the name of the job to be stopped, which can be queried in the GS_MY_ILMEVALUATIONDETAILS view.

3. Generate a policy for a table and schedule compression tasks in the background.
   a. Add the cold and hot separation policy for a table.
   ```
   gaussdb=# DROP TABLE IF EXISTS ILM_TABLE;
   gaussdb=# CREATE TABLE ILM_TABLE(a int);
   gaussdb=# ALTER TABLE ILM_TABLE ILM ADD POLICY ROW STORE COMPRESS ADVANCED
       ROW AFTER 3 MONTHS OF NO MODIFICATION;
   ```
   b. Set the parameters related to ILM execution.
   ```
   BEGIN
     DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
     DBE_ILM_ADMIN.CUSTOMIZE_ILM(12, 10);
     DBE_ILM_ADMIN.CUSTOMIZE_ILM(1, 1);
     DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 512);
   END;
   /
   ```
   c. Enable scheduled scheduling in the background.
   ```
   gaussdb=# CALL DBE_ILM_ADMIN.DISABLE_ILM();
   gaussdb=# CALL DBE_ILM_ADMIN.ENABLE_ILM();
   ```
   d. You can call DBE_SCHEDULER.set_attribute to set the opening time of the maintenance window as required. By default, this function is enabled at 22:00.
4. Set the parameters related to ILM execution.

   Specifies whether the time unit of ADO is day or second. The time unit second is used only for testing. The setting can be **ILM_POLICY_IN_SECONDS = 1** or **ILM_POLICY_IN_DAYS = 0** (default value).
   ```
   gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(11, 1);
   ```
   Specifies the maximum number of ADO jobs generated by an ADO task. The value is an integer or floating-point number greater than or equal to 0 and less than or equal to 2147483647. The value is rounded down.
   ```
   gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(12, 10);
   ```
   Specifies the frequency of executing an ADO task, in minutes. The default value is **15**. The value is an integer or floating-point number greater than or equal to 1 and less than or equal to 2147483647. The value is rounded down.
   ```
   gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(1, 1);
   ```
   Specifies the maximum number of bytes that can be processed by a single ADO job. The unit is MB. The value is an integer or floating-point number greater than or equal to 0 and less than or equal to 2147483647. The value is rounded down.

```
gaussdb=# CALL DBE_ILM_ADMIN.CUSTOMIZE_ILM(13, 512);
```

5. Check whether a table is suitable for compression and evaluate the compression benefits.

```
DBE_COMPRESSION.GET_COMPRESSION_RATIO(
SCRATCHTBSNAME IN VARCHAR2,
OWNNAME        IN VARCHAR2,
OBJNAME        IN VARCHAR2,
SUBOBJNAME     IN VARCHAR2,
COMPTYPE       IN NUMBER,
BLKCNT_CMP     OUT INTEGER,
BLKCNT_UNCMP   OUT INTEGER,
ROW_CMP        OUT INTEGER,
ROW_UNCMP      OUT INTEGER,
CMP_RATIO      OUT NUMBER,
COMPTYPE_STR   OUT VARCHAR2,
SAMPLE_RATIO   IN NUMBER DEFAULT 20,
OBJTYPE        IN INTEGER DEFAULT 1)
```

Table 6-3 Input parameters of DBE_COMPRESSION.GET_COMPRESSION_RATIO

Name	Description
SCRATCHTBSNAME	Tablespace to which a data object belongs.
OWNNAME	Data object owner (schema to which the data object belongs).
OBJNAME	Data object name.
SUBOBJNAME	Name of a data subobject.
COMPTYPE	● **1**: uncompressed. ● **2**: advanced compression.
SAMPLE_RATIO	Sampling ratio. The value is an integer or floating-point number ranging from 0 to 100, corresponding to the sampling ratio of *N* percent. The default value is **20**, indicating that 20% of the rows are sampled.
OBJTYPE	Object type. The options are as follows: **1**: table object.

Table 6-4 Output parameters of DBE_COMPRESSION.GET_COMPRESSION_RATIO

Name	Description
BLKCNT_CMP	Number of blocks occupied by compressed samples.
BLKCNT_UNCMP	Number of blocks occupied by uncompressed samples.

Name	Description
ROW_CMP	Number of rows that can be contained in a single block after samples are compressed.
ROW_UNCMP	Number of rows that can be contained in a single block when samples are not compressed.
CMP_RATIO	Compression ratio, that is, **blkcnt_uncmp** divided by **blkcnt_cmp**.
COMPTYPE_STR	Character string that describes the compression type.

Example:

```
gaussdb=# ALTER DATABASE set ilm = on;
gaussdb=# CREATE user user1 IDENTIFIED BY '********';
gaussdb=# CREATE user user2 IDENTIFIED BY '********';
gaussdb=# SET ROLE user1 PASSWORD '********';
gaussdb=# CREATE TABLE TEST_DATA (ORDER_ID INT, GOODS_NAME TEXT, CREATE_TIME
TIMESTAMP)
  ILM ADD POLICY ROW STORE COMPRESS ADVANCED ROW AFTER 1 DAYS OF NO MODIFICATION;
INSERT INTO TEST_DATA VALUES (1,'Snack package A', NOW());

DECLARE
o_blkcnt_cmp       integer;
o_blkcnt_uncmp     integer;
o_row_cmp          integer;
o_row_uncmp        integer;
o_cmp_ratio        number;
o_comptype_str     varchar2;
begin
dbe_compression.get_compression_ratio(
    SCRATCHTBSNAME  =>  NULL,
    OWNNAME         =>  'user1',
    OBJNAME         =>  'test_data',
    SUBOBJNAME      =>  NULL,
    COMPTYPE        =>  2,
    BLKCNT_CMP      =>  o_blkcnt_cmp,
    BLKCNT_UNCMP    =>  o_blkcnt_uncmp,
    ROW_CMP         =>  o_row_cmp,
    ROW_UNCMP       =>  o_row_uncmp,
    CMP_RATIO       =>  o_cmp_ratio,
    COMPTYPE_STR    =>  o_comptype_str,
    SAMPLE_RATIO    =>  100,
    OBJTYPE         =>  1);
RAISE INFO 'Number of blocks used by the compressed sample of the object        : %', o_blkcnt_cmp;
RAISE INFO 'Number of blocks used by the uncompressed sample of the object        : %',
o_blkcnt_uncmp;
RAISE INFO 'Number of rows in a block in compressed sample of the object        : %', o_row_cmp;
RAISE INFO 'Number of rows in a block in uncompressed sample of the object        : %', o_row_uncmp;
RAISE INFO 'Estimated Compression Ratio of Sample                              : %', o_cmp_ratio;
RAISE INFO 'Compression Type                                                   : %', o_comptype_str;
end;
/
INFO:  Number of blocks used by the compressed sample of the object        : 0
INFO:  Number of blocks used by the uncompressed sample of the object        : 0
INFO:  Number of rows in a block in compressed sample of the object        : 0
INFO:  Number of rows in a block in uncompressed sample of the object        : 0
INFO:  Estimated Compression Ratio of Sample                              : 1
INFO:  Compression Type                                                   : Compress Advanced
```

6. Query the last modification time of each line.

```
DBE_HEAT_MAP.ROW_HEAT_MAP(
OWNER           IN VARCHAR2,
```

```
SEGMENT_NAME    IN VARCHAR2,
PARTITION_NAME  IN VARCHAR2     DEFAULT NULL,
CTID            IN TEXT,
V_DEBUG         IN BOOL         DEFAULT FALSE);
```

Table 6-5 Input parameters of DBE_HEAT_MAP.ROW_HEAT_MAP

Name	Description
OWNER	Schema to which a data object belongs.
SEGMENT_NAME	Data object name.
PARTITION_NAME	Data object partition name. This parameter is optional. The default value is **NULL**.
CTID	**ctid** of the target row, that is, **block_id** or **row_id**.
V_DEBUG	Debugging. Debug logs are recorded.

Table 6-6 Output parameters of DBE_HEAT_MAP.ROW_HEAT_MAP

Name	Description
OWNER	Owner of a data object.
SEGMENT_NAME	Data object name.
PARTITION_NAME	Name of a data object partition. This parameter is optional.
TABLESPACE_NAME	Name of the tablespace to which data belongs.
FILE_ID	ID of the absolute file to which a row belongs.
RELATIVE_FNO	ID of the relative file to which a row belongs. (GaussDB does not have this logic. Therefore, the value is the same as the preceding value.)
CTID	**ctid** of a row, that is, **block_id** or **row_id**.
WRITETIME	Last modification time of a row.

Example:

```
gaussdb=# ALTER DATABASE set ilm = on;
gaussdb=# CREATE Schema HEAT_MAP_DATA;
gaussdb=# SET current_schema=HEAT_MAP_DATA;

gaussdb=# CREATE TABLESPACE example1 RELATIVE LOCATION 'tablespace1';
gaussdb=# CREATE TABLE HEAT_MAP_DATA.heat_map_table(id INT, value TEXT) TABLESPACE
example1;
gaussdb=# INSERT INTO HEAT_MAP_DATA.heat_map_table VALUES (1, 'test_data_row_1');

gaussdb=# SELECT * from DBE_HEAT_MAP.ROW_HEAT_MAP(
    owner          =>  'heat_map_data',
    segment_name   =>  'heat_map_table',
    partition_name  => NULL,
```

```
   ctid          =>  '(0,1)');
    owner     |  segment_name  | partition_name | tablespace_name | file_id | relative_fno | ctid  |
writetime
---------------+----------------+----------------+-----------------+---------+--------------+-------+-----------
 heat_map_data | heat_map_table |                | example1        |   17291 |        17291 | (0,1) |
(1 row)
```

7. Query the environment parameters related to ILM scheduling and execution.

```
gaussdb=# SELECT * FROM GS_ADM_ILMPARAMETERS;
               name               | value
----------------------------------+-------
 EXECUTION_INTERVAL               |    15
 RETENTION_TIME                   |    30
 ENABLED                          |     1
 POLICY_TIME                      |     0
 ABS_JOBLIMIT                     |    10
 JOB_SIZELIMIT                    |  1024
 WIND_DURATION                    |   240
 BLOCK_LIMITS                     |    40
 ENABLE_META_COMPRESSION          |     0
 SAMPLE_MIN                       |    10
 SAMPLE_MAX                       |    10
 CONST_PRIO                       |    40
 CONST_THRESHOLD                  |    90
 EQVALUE_PRIO                     |    60
 EQVALUE_THRESHOLD                |    80
 ENABLE_DELTA_ENCODE_SWITCH       |     1
 LZ4_COMPRESSION_LEVEL            |     0
 ENABLE_LZ4_PARTIAL_DECOMPRESSION |     1
(18 rows)
```

8. Query the brief information about an ILM policy, including the policy name, type, enabling status, disabling status, and deletion status.

```
gaussdb=# SELECT * FROM GS_ADM_ILMPOLICIES;
policy_name |  policy_type  | tablespace | enabled | deleted
-------------+---------------+------------+---------+---------
 p1          | DATA MOVEMENT |            | YES     | NO


gaussdb=# SELECT * FROM GS_MY_ILMPOLICIES;
 policy_name |  policy_type  | tablespace | enabled | deleted
-------------+---------------+------------+---------+---------
 p1          | DATA MOVEMENT |            | YES     | NO
```

9. Query the brief data movement information about an ILM policy, including the policy name, action type, and condition.

```
gaussdb=# SELECT * FROM GS_ADM_ILMDATAMOVEMENTPOLICIES;
 policy_name | action_type | scope | compression_level | tier_tablespace | tier_status |
condition_type     | condition_days | custom_function | policy_subtype | action_clause | tier_to
-------------+-------------+-------+-------------------+-----------------+-------------+------------------------
+----------------+-----------------+----------------+---------------+---------
 p1          | COMPRESSION | ROW   | ADVANCED          |                 |             | LAST MODIFICATION
TIME |             90 |                 |                |               |

gaussdb=# SELECT * FROM GS_MY_ILMDATAMOVEMENTPOLICIES;
 policy_name | action_type | scope | compression_level | tier_tablespace | tier_status |
condition_type     | condition_days | custom_function | policy_subtype | action_clause | tier_to
-------------+-------------+-------+-------------------+-----------------+-------------+------------------------
+----------------+-----------------+----------------+---------------+---------
 p1          | COMPRESSION | ROW   | ADVANCED          |                 |             | LAST MODIFICATION
TIME |             90 |                 |                |               |
(1 row)
```

10. Query the brief information about all data objects to which ILM policies are applied and the corresponding policies, including the policy name, data object name, policy source, and policy enabling/disabling status.

```
gaussdb=# SELECT * FROM  GS_ADM_ILMOBJECTS;
 policy_name | object_owner | object_name | subobject_name | object_type |    inherited_from    |
tbs_inherited_from | enabled | deleted
-------------+--------------+-------------+----------------+-------------+----------------------
```

```
+--------------------+---------+---------
 p1          | public       | lineitem    |               | TABLE       | POLICY NOT INHERITED |                   |
YES     | NO

gaussdb=# SELECT * FROM  GS_MY_ILMOBJECTS;
 policy_name | object_owner | object_name | subobject_name | object_type |    inherited_from    |
tbs_inherited_from | enabled | deleted
-------------+--------------+-------------+----------------+-------------+----------------------
+--------------------+---------+---------
 p1          | public       | lineitem    |               | TABLE       | POLICY NOT INHERITED |                   |
YES     | NO
```

11. Query the brief information about an ADO task, including the task ID, task owner, status, and time.

```
gaussdb=# SELECT * FROM GS_ADM_ILMTASKS;
 task_id | task_owner |   state   |         creation_time         |          start_time           |
completion_time
---------+------------+-----------+-------------------------------+-------------------------------
+-------------------------------
       1 | omm        | COMPLETED | 2023-10-16 12:03:55.113296+08 | 2023-10-16 12:03:55.113296+08 |
2023-10-16 12:03:56.326864+08
(1 row)

gaussdb=# SELECT * FROM GS_MY_ILMTASKS;
 task_id | task_owner |   state   |         creation_time         |          start_time           |
completion_time
---------+------------+-----------+-------------------------------+-------------------------------
+-------------------------------
       1 | omm        | COMPLETED | 2023-10-16 12:03:55.113296+08 | 2023-10-16 12:03:55.113296+08 |
2023-10-16 12:03:56.326864+08
(1 row)
```

12. Query the evaluation details of an ADO task, including the task ID, policy information, object information, evaluation result, and ADO job name.

```
gaussdb=# SELECT * FROM GS_ADM_ILMEVALUATIONDETAILS;
 task_id | policy_name | object_owner | object_name | subobject_name | object_type |
selected_for_execution |      job_name       | comments

---------+-------------+--------------+-------------+----------------+-------------+------------------------
+-------------------+---------
N/A
       1 | p2          | public       | ilm_table_1 |                | TABLE       | SELECTED FOR EXECUTION | ilmjob
$_postgres1 |
(1 row)

gaussdb=# SELECT * FROM GS_MY_ILMEVALUATIONDETAILS;
 task_id | policy_name | object_owner | object_name | subobject_name | object_type |
selected_for_execution |      job_name       | comments

---------+-------------+--------------+-------------+----------------+-------------+------------------------
+-------------------+---------
N/A
       1 | p2          | public       | ilm_table_1 |                | TABLE       | SELECTED FOR EXECUTION | ilmjob
$_postgres1 |
(1 row)
```

13. Query the execution details of an ADO job, including the task ID, job name, job status, and job time.

```
gaussdb=# SELECT * FROM GS_ADM_ILMRESULTS;
 task_id |      job_name       |       job_state        |          start_time           |        completion_time        |
comments |                   statistics
---------+-------------------+------------------------+-------------------------------
+-------------------------------+----------+-----------------------------------------------
       1 | ilmjob$_postgres1 | COMPLETED SUCCESSFULLY | 2023-10-16 12:03:56.290176+08 |
2023-10-16 12:03:56.319829+08 |          | SpaceSaving=0,BoundTime=1697429033,LastBlkNum=40
(1 row)

gaussdb=# SELECT * FROM GS_MY_ILMRESULTS;
 task_id | job_name | job_state | start_time | completion_time | comments | statistics
---------+----------+-----------+------------+-----------------+----------+------------
```

```
(0 rows)
 task_id |     job_name      |        job_state        |          start_time           |        completion_time        |
comments |                      statistics
---------+-------------------+------------------------+-------------------------------
+-------------------------------+----------+---------------------------------------------------
       1 | ilmjob$_postgres1 | COMPLETED SUCCESSFULLY | 2023-10-16 12:03:56.290176+08 |
2023-10-16 12:03:56.319829+08 |          | SpaceSaving=0,BoundTime=1697429033,LastBlkNum=40
(1 row)
```

7 Foreign Data Wrapper

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

7.1 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, the system administrator, 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 create a foreign table, run **CREATE FOREIGN TABLE**.

  NOTE

  - The structure of the foreign table must be consistent with the data in the specified file.
  - When a foreign table is queried, no write operation is allowed.
- To drop a foreign table, run **DROP FOREIGN TABLE**.
- 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.

8 Dynamic Data Masking

Data masking is an effective database privacy protection solution, which can prevent attackers from snooping on private data. The dynamic data masking mechanism is a technology that protects privacy data by customizing masking policies. It can effectively prevent unauthorized users from accessing sensitive information while retaining original data. After the administrator specifies the object to be masked and customizes a data masking policy, if the database resources queried by a user are associated with a masking policy, data is masked based on the user identity and masking policy to restrict attackers' access to privacy data.

Limitations

- The dynamic data masking policy must be created by a user with the POLADMIN or SYSADMIN attribute, or by the initial user. Common users do not have the permission to access the security policy system catalog and system view.
- Dynamic data masking takes effect only on data tables for which masking policies are configured. Audit logs are not within the effective scope of the masking policies.
- In a masking policy, only one masking mode can be specified for a resource label.
- Multiple masking policies cannot be used to mask the same resource label, except when **FILTER** is used to specify user scenarios where the policies take effect and there is no intersection between user scenarios of different masking policies that contain the same resource label. In this case, you can identify the policy that a resource label is masked by based on the user scenario.
- It is recommended that **APP** in **FILTER** be set to applications in the same trusted domain. Since a client may be forged, a security mechanism must be formed on the client when **APP** is used, to reduce misuse risks. Generally, you are advised not to use it. If it is used, pay attention to the risk of client spoofing.
- For INSERT or MERGE INTO operations with the query clause, if the source table contains masked columns, the inserted or updated result in the preceding two operations is the masked value and cannot be restored.

- When the built-in security policy is enabled, the ALTER TABLE EXCHANGE PARTITION statement fails to be executed if the source table is in the masked column.
- If a dynamic data masking policy is configured for a table, grant the trigger permission of the table to other users with caution to prevent other users from using the trigger to bypass the masking policy.
- A maximum of 98 dynamic data masking policies can be created.
- Only data with the resource labels containing the COLUMN attribute can be masked.
- Only columns in permanent ordinary tables can be masked.
- Only the data directly queried by SELECT can be masked. If you perform secondary processing on the masked result, the masking policy becomes invalid or does not meet the expectation.
- User-defined functions used for dynamic data masking can only be a standard database SQL or PL/SQL function.
- If the user-defined function used for dynamic data masking contains a statement (such as SELECT and INSERT) for accessing database resources, the dynamic data masking result using the user-defined function may not meet the expectation or may cause security risks.
- After a data masking policy is successfully created for a user-defined function used for dynamic data masking, if ALTER or DROP is performed on the data masking column, the data masking policy becomes invalid or does not meet the expectation.
- The user-defined function for dynamic data masking does not support the SECURITY INVOKER function. After a data masking policy is successfully created for a user-defined function that applies to dynamic data masking, no CREATE, ALTER, or DROP operation can be performed on the function.
- User-defined functions used for dynamic data masking can be created only by users with the POLADMIN permission. A user with the POLADMIN permission grants the usage permission for accessing the schema to the **PUBLIC** user. If the user cannot access the user-defined functions due to the GRANT and REVOKE operations, maskall is used.
- The user-defined function used for dynamic data masking must be idempotent. That is, the execution results are the same for multiple times. If the user-defined function is set to non-idempotent, the dynamic data masking result using the user-defined function in distributed scenarios may not meet the expectation.
- The data masking policy created for a user-defined function used for dynamic data masking cannot be applied to system catalogs.
- Taking an IPv4 address as an example, the following formats are supported.

IP Address	Example
Single IP address	127.0.0.1
IP address with mask	127.0.0.1|255.255.255.0

CIDR IP address	127.0.0.1/24
IP address segment	127.0.0.1-127.0.0.5

- Dynamic data masking policies cannot be exported using gs_dump. The system administrator or security policy administrator can access the GS_MASKING_POLICY, GS_MASKING_POLICY_ACTIONS, and GS_MASKING_POLICY_FILTERS system catalogs to query created dynamic data masking policies.

Viewing the Basic Configurations of Dynamic Data Masking

Step 1 Set and check if the dynamic data masking function is enabled.

```
gs_guc reload -Z coordinator -N all -I all -c "enable_security_policy=on"
```

If **enable_security_policy** is set to **on**, the function is enabled. If it is set to **off**, the function is disabled.

```
gaussdb=# SHOW enable_security_policy;
 enable_security_policy
------------------------
 on
(1 row)
```

----End

Creating a Masking Policy

Step 1 Create a table.

```
-- Create table tb_for_masking.
gaussdb=# CREATE TABLE tb_for_masking(idx int, col1 text, col2 text, col3 text, col4 text, col5 text, col6 text, col7 text,col8 text);

-- Insert data into the tb_for_masking table.
gaussdb=# INSERT INTO tb_for_masking VALUES(1, '9876543210', 'usr321usr', 'abc@example.com', 'abc@example.com', '1234-4567-7890-0123', 'abcdef 123456 ui 323 jsfd321 j3k2l3', '4880-9898-4545-2525', 'this is a llt case');

-- View data.
gaussdb=# SELECT * FROM tb_for_masking;
 idx |    col1    |   col2    |      col3       |      col4       |        col5         |                col6
     |        col7         |        col8
-----+------------+-----------+-----------------+-----------------+---------------------+-------------------------------
------+---------------------+--------------------
   1 | 9876543210 | usr321usr | abc@example.com | abc@example.com | 1234-4567-7890-0123 | abcdef 123456 ui 323 jsfd321 j
3k2l3 | 4880-9898-4545-2525 | this is a llt case
(1 row)
```

Step 2 Create a resource label.

```
-- Create a resource label for sensitive column col1.
gaussdb=# CREATE RESOURCE LABEL mask_lb1 ADD COLUMN(tb_for_masking.col1);
CREATE RESOURCE LABEL
gaussdb=# CREATE RESOURCE LABEL mask_lb5 ADD COLUMN(tb_for_masking.col5);
CREATE RESOURCE LABEL
```

Step 3 Create a masking policy.

For details about the syntax format, see "SQL Reference > SQL Syntax > CREATE MASKING POLICY" in *Developer Guide*.

```
-- Create a data masking policy named maskpol1.
gaussdb=# CREATE MASKING POLICY maskpol1 maskall ON LABEL(mask_lb1);
CREATE MASKING POLICY
```

During the configuration of dynamic data masking policies, user-defined functions created by users can be adapted.

```
gaussdb=# create or replace function msk_creditcard(col text) returns TEXT as $$
declare
    result TEXT;
begin
    result := overlay(col placing 'xxxx-xxxx' from 6);
    return result;
end;
$$ language plpgsql security DEFINER;
CREATE FUNCTION
-- Create a data masking policy named maskpol5.
gaussdb=# CREATE MASKING POLICY maskpol5 msk_creditcard ON LABEL(mask_lb5);
CREATE MASKING POLICY
```

Step 4 Query the masked column data in the **tb_for_masking** table.

```
-- Access the tb_for_masking table. The masking policy is triggered for the col1 column.
gaussdb=# SELECT col1 FROM tb_for_masking;
    col5
------------
 xxxxxxxxxx
(1 row)
-- Access the tb_for_masking table. The masking policy is triggered for the col5 column.
gaussdb=# SELECT col5 FROM tb_for_masking;
        col5
---------------------
 1234-xxxx-xxxx-0123
(1 row)
```

Step 5 Clear data.

```
-- Drop masking policies.
gaussdb=# DROP MASKING POLICY maskpol1, maskpol5;
DROP MASKING POLICY
-- Drop resource labels.
gaussdb=# DROP RESOURCE LABEL mask_lb1, mask_lb5;
DROP RESOURCE LABEL
-- Drop the tb_for_masking table.
gaussdb=# DROP TABLE tb_for_masking;
DROP TABLE
```

----End