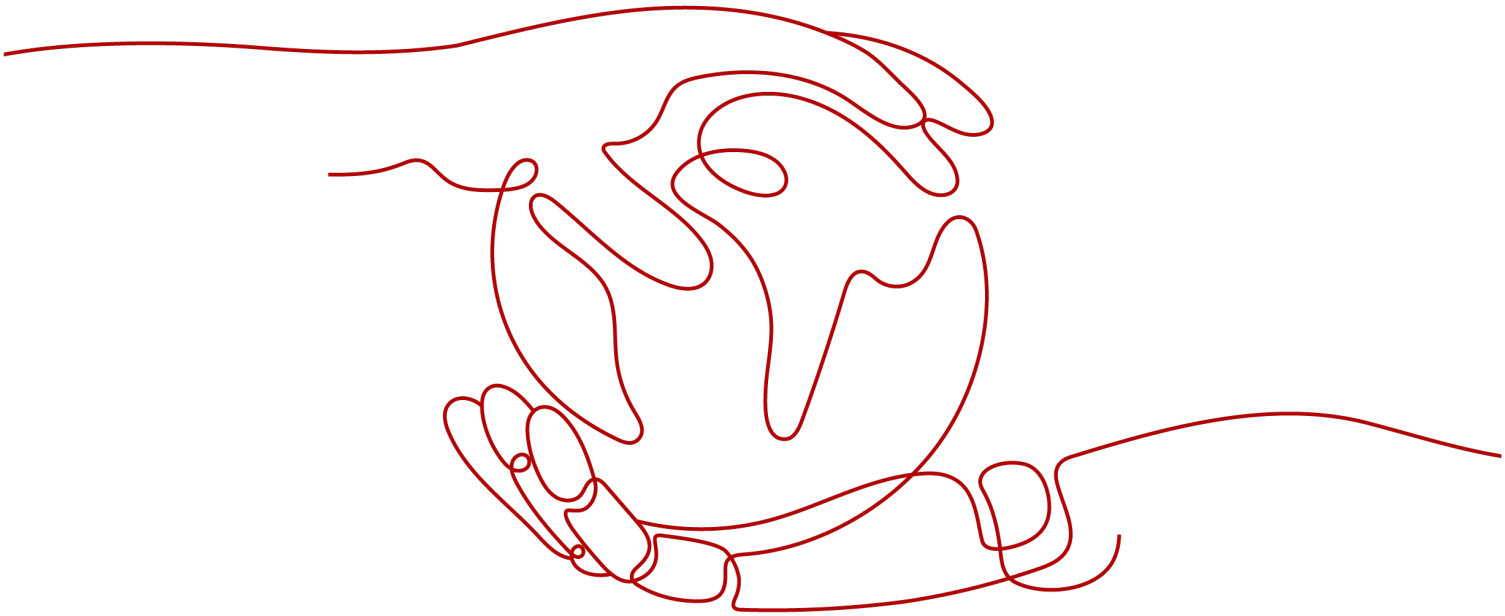
Data Warehouse Service

Hybrid Data Warehouse

Issue	01
Date	2025-01-07



HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Introduction to Hybrid Data Warehouse

Hybrid: A hybrid data warehouse offers both large-scale data query and analysis capabilities, as well as low-cost, high-concurrency, high-performance, and low-latency transaction processing capabilities.

NOTE

- To use hybrid data warehouse capabilities, choose the storage-compute coupled architecture when you create a GaussDB(DWS) cluster on the console and ensure the vCPU to memory ratio is 1:4 when setting up cloud disk flavors. For more information, see **Data Warehouse Flavors**.
- When setting up a GaussDB(DWS) cluster, make sure to have a vCPU to memory ratio of 1:8 for standard data warehouses and a ratio of 1:4 for hybrid data warehouses. You can distinguish a standard data warehouse from a real-time data warehouse by comparing their vCPU to memory ratios.

A hybrid data warehouse needs to work with data sources, such as upstream databases or applications, to insert, upsert, and update data in real time. The data warehouse should also be able to query data shortly after it was imported.

Currently, the existing row-store and column-store tables in a conventional GaussDB(DWS) data warehouse cannot meet real-time data import and query requirements. Row-store tables have strong real-time import capabilities and support highly concurrent updates, but their disk usage is high and query efficiency is low. Column-store tables have high data compression ratio and good OLAP query performance, but do not support concurrent updates. Concurrent import will cause severe lock conflicts.

To solve these problems, we use column storage to reduce the disk usage, support highly concurrency updates, and improve query speed. GaussDB(DWS) hybrid data warehouses use HStore tables to achieve high performance during real-time data import and query, and have the transaction processing capabilities required for traditional OLTP scenarios.

The HStore tables uniquely support single and small-batch real-time IUD operations, as well as regular large-batch import. Data can be queried immediately after being imported. You can deduplicate traditional indexes (such as primary keys) and accelerate point queries. You can further accelerate OLAP queries through partitioning, multi-dimensional dictionaries, and partial sorting. Strong data consistency can be ensured for transactions with heavy workloads, such as TPC-C.

NOTE

- Only clusters 8.2.0.100 and later support the HStore tables of the hybrid data warehouse.
- The hybrid data warehouse is used for both production and analysis. It is applicable to hybrid transaction and analysis scenarios. It can be deployed in single-node or cluster mode. For how to create a hybrid data warehouse, see **Creating a GaussDB(DWS) 2.0 Cluster with Coupled Storage and Compute**.
- Hot and cold data management is supported for HStore tables. For details, see **Best Practices of Hot and Cold Data Management**. This function is supported only by cluster versions 8.2.0.101 and later.
- HStore is a table type designed for the hybrid data warehouse and is irrelevant to the SQL parameter **hstore**.

Differences from Standard Data Warehouses

Hybrid data warehouses and standard data warehouses are two types of GaussDB(DWS) data warehouses with different specifications and usage. For details, see **Table 1-1**.

Table 1-1 Comparison between hybrid and standard data warehouses

Type	Standard Data Warehouse (Compute-Storage Coupled Architecture with 1:8 vCPU to Memory Ratio)	Hybrid Data Warehouse (Compute-Storage Coupled Architecture with 1:4 vCPU to Memory Ratio)
Application scenario	Converged data analysis using OLAP. It is used in sectors such as finance, government and enterprise, e-commerce, and energy.	Real-time data import + Hybrid analysis. Real-time upstream data import + Real-time query after data import. It is mainly used in scenarios that have high requirements on real-time data import, such as e-commerce and finance.
Advantage	It is cost-effective and widely used. Cost effective, both hot and cold data analysis supported, elastic storage and compute capacities.	Hybrid load, high data import performance. It achieves high query efficiency and high data compression ratio that are equivalent to those of column storage. It can also process transactions in traditional OLTP scenarios.

Type	Standard Data Warehouse (Compute-Storage Coupled Architecture with 1:8 vCPU to Memory Ratio)	Hybrid Data Warehouse (Compute-Storage Coupled Architecture with 1:4 vCPU to Memory Ratio)
Features	Excellent performance in interactive analysis and offline processing of massive data, as well as complex data mining.	It supports highly concurrent update operations on massive amounts of data and can achieve high query efficiency. It achieves high performance when processing a large amount of data in scenarios like high-concurrency import and latency-sensitive queries.
SQL syntax	Highly compatible with SQL syntax	Compatible with column-store syntax
GUC parameter	You can configure a wide variety of GUC parameters to tailor your data warehouse environment.	It is compatible with standard data warehouse GUC parameters and supports hybrid data warehouse tuning parameters.

Technical Highlights

- Transaction consistency

  Data can be retrieved for queries immediately after being inserted or updated. After concurrent updates, data is strongly consistent, and there will not be incorrect results caused by wrong update sequence.

- High query performance

  In complex OLAP queries, such as multi-table correlation, the data warehouse achieves high performance through comprehensive distributed query plans and distributed executors. It also supports complex subqueries and stored procedures.

- Quick import

  There will not be lock conflicts on column-store CUs. High-concurrency update and import operations are supported. The concurrent update performance can be over 100 times higher than before in general scenarios.

- High compression

Column storage can achieve a high compression ratio. Data is stored in the column-store primary table through MERGE can be compressed to greatly reduce disk usage and I/O.

- Query acceleration

  You can deduplicate traditional indexes (such as primary keys) and accelerate point queries. You can further accelerate OLAP queries through partitioning, multi-dimensional dictionaries, and partial sorting.

Comparison Between Row-store, Column-store, and HStore Tables

Table 1-2 Comparison between row-store, column-store, and HStore tables

Table Type	Row-Store	Column-Store	HStore
Data storage mode	The attributes of a tuple are stored nearby.	The values of an attribute are stored nearby in the unit of CU.	Data is stored in the column-store primary tables as CUs. Updated columns and data inserted in small batches is serialized and then stored in a newly designed delta table.
Data write	Row-store compression has not been put into commercial use. Data is stored as it is, occupying a large amount of disk space.	In row storage, data with the same attribute value types is easy to compress. Data write consumes much fewer I/O resources and less disk space.	Data inserted in batches is directly written to CUs, which are as easy to compress as column storage. Updated columns and data inserted in small batches are serialized and then compressed. They will also be periodically merged to primary table CUs.
Data update	Data is updated by row, avoiding CU lock conflicts. The performance of concurrent updates (UPDATE/ UPSERT/DELETE) is high.	The entire CU needs to be locked even if only one record in it is updated. Generally, concurrent updates (UPDATE/UPSERT/ DELETE) are not supported.	CU lock conflicts can be avoided. The performance of concurrent updates (UPDATE/UPSERT/ DELETE) is higher than 60% of the row-store update performance.

Table Type	Row-Store	Column-Store	HStore
Data read	Data is read by row. An entire row needs to be retrieved even if only one column in it needs to be accessed. The query performance is low.	When data is read by column, only the CU of a column needs to be accessed. CUs can be easily compressed, occupying less I/O resources, and achieve high read performance.	Data in a column-store primary table is read by column. Updated columns and data inserted in small batches are deserialized and then retrieved. After data is merged to the primary table, the data can be read as easily as that in column storage.
Advantage	The concurrent update performance is high.	The query performance is high, and the disk space usage is small.	The concurrent update performance is high. After data merges, the query and compression performance are the same as those of column storage.
Disadvantage	A large amount of disk space is occupied, and the query performance is low.	Generally, concurrent updates are not supported.	A background permanent thread is required to clear unnecessary HStore table data after merge. Data is merged to the primary table CUs and then cleared. This operation is irrelevant to the SQL syntax **MERGE**.
Application scenario	1. OLTP transactions with frequent update and deletion operations 2. Point queries (simple queries that are based on indexes and return a small amount of data)	1. OLAP query and analysis 2. A large volume of data is imported, and is rarely updated or deleted after the import.	1. Data is concurrently imported to the database in real time. 2. High-concurrency update and import; and high-performance query

2 Support and Constraints

A hybrid data warehouse is compatible with all column-store syntax.

Table 2-1 Supported syntax

Syntax	Supported
CREATE TABLE	Yes
CREATE TABLE LIKE	Yes
DROP TABLE	Yes
INSERT	Yes
COPY	Yes
SELECT	Yes
TRUNCATE	Yes
EXPLAIN	Yes
ANALYZE	Yes
VACUUM	Yes
ALTER TABLE DROP PARTITION	Yes
ALTER TABLE ADD PARTITION	Yes
ALTER TABLE SET WITH OPTION	Yes
ALTER TABLE DROP COLUMN	Yes
ALTER TABLE ADD COLUMN	Yes
ALTER TABLE ADD NODELIST	Yes
ALTER TABLE CHANGE OWNER	Yes
ALTER TABLE RENAME COLUMN	Yes
ALTER TABLE TRUNCATE PARTITION	Yes

Syntax	Supported
CREATE INDEX	Yes
DROP INDEX	Yes
DELETE	Yes
Other ALTER TABLE syntax	Yes
ALTER INDEX	Yes
MERGE	Yes
SELECT INTO	Yes
UPDATE	Yes
CREATE TABLE AS	Yes

Constraints

1. To use HStore tables, use the following parameter settings, or the performance of HStore tables will deteriorate significantly:

   autovacuum_max_workers_hstore=3, **autovacuum_max_workers=6**, and **autovacuum=true**

2. In version 8.2.1 and later, you can now clear the dirty data from column-store indexes. This is especially beneficial when dealing with frequent data updates and imports into the database. By efficiently managing the index space, it improves both the import and query performance.
3. When using HStore asynchronous sorting, pay attention to the following:
   - DML operations on certain data may be blocked during asynchronous sorting. The maximum blocking granularity is the row threshold for asynchronous sorting. This function is not recommended for frequent DML operations.
   - Automatic asynchronous sorting and column-store VACUUM cannot be used together. If the autovacuum process meets the conditions for column-store VACUUM, asynchronous sorting is skipped and will wait for the next trigger. In some cases, column-store VACUUM may be continuously triggered due to a high volume of DML operations, which means asynchronous sorting will never be triggered.

3 Hybrid Data Warehouse Syntax

3.1 CREATE TABLE

Function

Create an HStore table in the current database. The table will be owned by the user who created it.

In a hybrid data warehouse, you can use DDL statements to create HStore tables. To create an HStore table, set **enable_hstore** to **true** and set **orientation** to **column**.

NOTE

- To use hybrid data warehouse capabilities, choose the storage-compute coupled architecture when you create a GaussDB(DWS) cluster on the console and ensure the vCPU to memory ratio is 1:4 when setting up cloud disk flavors. For more information, see **Data Warehouse Flavors**.
- When setting up a GaussDB(DWS) cluster, make sure to have a vCPU to memory ratio of 1:8 for standard data warehouses and a ratio of 1:4 for hybrid data warehouses. You can distinguish a standard data warehouse from a real-time data warehouse by comparing their vCPU to memory ratios.

Precautions

- When creating an HStore table, ensure that the database GUC parameter settings meet the following requirements:
  - **autovacuum** is set to **on**.
  - The value of **autovacuum_max_workers_hstore** is greater than **0**.
  - The value of **autovacuum_max_workers** is greater than that of **autovacuum_max_workers_hstore**.
- To create an HStore table, you must have the **USAGE** permission on schema cstore.
- The table-level parameters **enable_delta** and **enable_hstore** cannot be enabled at the same time. The parameter **enable_delta** is used to enable delta for common column-store tables and conflicts with **enable_hstore**.

- Each HStore table is bound to a delta table. The OID of the delta table is recorded in the **reldeltaidx** field in **pg_class**. (The **reldelta** field is used by the delta table of the column-store table).

Syntax

```
CREATE TABLE [ IF NOT EXISTS ] table_name
({ column_name data_type
   | LIKE source_table [like_option [...] ] }
}
   [, ... ])
[ WITH ( {storage_parameter = value}  [, ... ] ) ]
[ TABLESPACE tablespace_name ]
[ DISTRIBUTE BY  HASH ( column_name [,...])]
[ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]
[ PARTITION BY {
       {RANGE (partition_key) ( partition_less_than_item [, ... ] )}
 } [ { ENABLE | DISABLE } ROW MOVEMENT ] ];
The options for LIKE are as follows:
{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | PARTITION
| RELOPTIONS | DISTRIBUTION | ALL }
```

Differences Between Delta Tables

Table 3-1 Differences between the delta tables of HStore and column-store tables

Type	Column-Store Delta Table	HStore Delta Table	HStore Opt Delta Table
Table structure	Same as that defined for the column-store primary table.	Different from that defined for the primary table.	Different from the definitions of the primary table and but same as the definitions of the HStore table.
Functionality	Used to temporarily store a small batch of inserted data. After the data size reaches the threshold, the data will be merged to the primary table. In this way, data will not be directly inserted to the primary table or generate a large number of small CUs.	Persistently stores UPDATE, DELETE, and INSERT information. It is used to restore the memory structure that manages concurrent updates, such as the memory update chain, in the case of a fault.	Persistently stores UPDATE, DELETE, and INSERT information. It is used to restore the memory structure that manages concurrent updates, such as the memory update chain, in the case of a fault. It is further optimized compared with HStore.

Type	Column-Store Delta Table	HStore Delta Table	HStore Opt Delta Table
Weakness	If data is not merged in a timely manner, the delta table will grow large and affect query performance. In addition, the table cannot solve lock conflicts during concurrent updates.	The merge operation depends on the background AUTOVACUUM.	The merge operation depends on the background AUTOVACUUM.

Type	Column-Store Delta Table	HStore Delta Table	HStore Opt Delta Table
Specification differences	Concurrent requests in the same CU are not supported. It is applicable to the scenario where there are not many concurrent updates.	1. Insertion and update restrictions: • **MERGE INTO** does not support concurrent updates of the same row or repeated updates of the same key. • Concurrent UPDATE or DELETE operations on the same row are not supported. Otherwise, an error is reported. 2. Index and query restrictions: • Indexes do not support array condition filtering, **IN** expression filtering, partial indexes, or expression indexes. • Indexes cannot be invalidated. 3. Table structure and operation restrictions: • Ensure that the tables to be exchanged are HStore tables during partition exchange or relfilenode operations. • The distribution column cannot be modified	1. Insertion and update restrictions: • **MERGE INTO** does not support concurrent updates of the same row or repeated updates of the same key. • Concurrent updates or deletions of the same row is not supported. • **hstore_opt** does not support cross-partition upserts. 2. Index and query restrictions: • Bitmap indexes are supported. • Global dictionaries are supported. • **bitmap_columns** must be specified during table creation and cannot be modified after being set. • The opt version does not support transparent parameter transmission during SMP streaming. In multi-table join queries that require partition pruning, avoid using replicated tables or setting **query_dop**.

Type	Column-Store Delta Table	HStore Delta Table	HStore Opt Delta Table
		using the **UPDATE** command. You are not advised to modify the partition column using the **UPDATE** command. (No error is reported, but the performance is poor.)	3. Table structure and operation restrictions: • Distribution columns and partition columns cannot be modified using **UPDATE**. • The **enable_hstore_opt** attribute must be set when the table is created and cannot be changed after being set.

Type	Column-Store Delta Table	HStore Delta Table	HStore Opt Delta Table
Data import suggestions	1. For optimal data import, query performance, and space utilization, it is recommended to choose the HStore Opt table. In scenarios involving micro-batch copying with high performance demands and no data updates, you can choose the HStore table. 2. Similarities between HStore and HStore Opt tables: • The performance of importing data using **UPDATE** is poor. You are advised to use **UPSERT** to import data. • When using **DELETE** to import data, use index scanning. The **JDBC batch** method is recommended. • Use **MERGE INTO** to import data records to the database when the data volume exceeds 1 million per DN and there is no concurrent data. • Do not modify or add data in cold partitions. 3. Suggestions on HStore table data import using **UPSERT**: • Select a method. Step 1: Select **Method 2** for partial column upsert. For full column upsert (update all columns to new values without expressions when a conflict occurs), go to step 2. Step 2: Check whether data is concurrently updated to the same key when being imported to the database. If no conflict occurs, select **Method 1**. If a conflict occurs, go to step 3. Step 3: If duplicate data exists in the database, select **Method 2**. Otherwise, go to step 4. Step 4: If copying of temporary tables is used for import, select **Method 3**. Otherwise, select **Method 2**. • The methods are as follows: – **Method 1**: Enable **enable_hstore_nonconflict_upsert_optimization** and disable **enable_hstore_partial_upsert_optimization**. – **Method 2**: Disable **enable_hstore_nonconflict_upsert_optimization** and enable **enable_hstore_partial_upsert_optimization**. – **Method 3**: Disable **enable_hstore_nonconflict_upsert_optimization** and **enable_hstore_partial_upsert_optimization**. • Note: If the number of accumulated batches is less than 2,000, import data in batches into the database. For accumulated batches exceeding 2,000, import data into the database by copying temporary tables. 4. Suggestions on HStore Opt table data import using **UPSERT**: If there is no concurrency conflict, enable the **enable_hstore_nonconflict_upsert_optimization** parameter. In other scenarios, disable the parameter. The optimal path is automatically selected.		

Type	Column-Store Delta Table	HStore Delta Table	HStore Opt Delta Table
Point query suggestions	1. Generally, the HStore Opt table is recommended for point queries. 2. Similarities between HStore and HStore Opt tables: Create a level-2 partition on the column where the equal-value filter condition is most frequently used and distinct values are evenly distributed. 3. Suggestions on using HStore tables for point queries: ● Accelerating indexes other than primary keys may have poor effect. You are advised not to enable index acceleration. ● If the data type is numeric or strings less than 16 bytes, Turbo acceleration is recommended. 4. Suggestions on using HStore Opt tables: ● For equal-value filter columns not in level-2 partitions, if the columns involved in the filter criteria are basically fixed in the query, use the CB-tree index. If the columns change continuously, you are advised to use the GIN index. Do not select more than five index columns. ● For all string columns involving equivalent filtering, bitmap indexes can be specified during table creation. The number of columns is not limited, but cannot be modified later. ● Specify columns that can be filtered by time range as the partition columns. ● If the number of returned data records exceeds 100,000 per DN, index scanning may not significantly enhance performance. In this case, you are advised to use the GUC parameter **enable_seqscan** to test the performance then determine which optimization method to use.		

Parameters

- **IF NOT EXISTS**

  If **IF NOT EXISTS** is specified, a table will be created if there is no table using the specified name. If there is already a table using the specified name, no error will be reported. A message will be displayed indicating that the table already exists, and the database will skip table creation.

- **table_name**

  Specifies the name of the table to be created.

  The table name can contain a maximum of 63 characters, including letters, digits, underscores (_), dollar signs ($), and number signs (#). It must start with a letter or underscore (_).

- **column_name**

  Specifies the name of a column to be created in the new table.

  The column name can contain a maximum of 63 characters, including letters, digits, underscores (_), dollar signs ($), and number signs (#). It must start with a letter or underscore (_).

- **data_type**

  Specifies the data type of the column.

- **LIKE source_table [like_option ...]**

  Specifies a table from which the new table automatically copies all column names and their data types.

  The new table and the original table are decoupled after creation is complete. Changes to the original table will not be applied to the new table, and scans on the original table will not be performed on the data of the new table.

  Columns copied by **LIKE** are not merged with the same name. If the same name is specified explicitly or in another **LIKE** clause, an error will be reported.

  HStore tables can be inherited only from HStore tables.

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

  Specifies an optional storage parameter for a table.

  – ORIENTATION

    Specifies the storage mode (time series, row-store, or column-store) of table data. This parameter cannot be modified once it is set. For HStore tables, use the column storage mode and set **enable_hstore** to **on**.

    Options:

    - **TIMESERIES** indicates that the data is stored in time series.
    - **COLUMN** indicates that the data is stored in columns.
    - **ROW** indicates that table data is stored in rows.

    Default value: **ROW**

  – COMPRESSION

    Specifies the compression level of the table data. It determines the compression ratio and time. Generally, a higher compression level indicates a higher compression ratio and a longer compression time, and vice versa. The actual compression ratio depends on the distribution characteristics of loading table data.

    Options:

    - The valid values for HStore tables and column-store tables are **YES**/**NO** and **LOW**/**MIDDLE**/**HIGH**, and the default is **LOW**.
    - The valid values for row-store tables are **YES** and **NO**, and the default is **NO**.

  – COMPRESSLEVEL

    Specifies table data compression rate and duration at the same compression level. This divides a compression level into sub-levels, providing you with more choices for compression ratio and duration. As the value becomes greater, the compression rate becomes higher and duration longer at the same compression level. The parameter is only valid for time series tables and column-store tables.

    Value range: 0 to 3

    Default value: **0**

- MAX_BATCHROW

  Specifies the maximum number of rows in a storage unit during data loading. The parameter is only valid for time series tables and column-store tables.

  Value range: 10000 to 60000

  Default value: **60000**

- PARTIAL_CLUSTER_ROWS

  Specifies the number of records to be partially clustered for storage during data loading. The parameter is only valid for time series tables and column-store tables.

  Value range: 600000 to 2147483647

- enable_delta

  Specifies whether to enable delta tables in column-store tables. This parameter cannot be enabled for HStore tables.

  Default value: **off**

- enable_hstore

  Specifies whether to create a table as an HStore table (based on column-store tables). The parameter is only valid for column-store tables. This parameter is supported by version 8.2.0.100 or later clusters.

  Default value: **off**

  NOTE

  If this parameter is enabled, the following GUC parameters must be set to ensure that HStore tables are cleared.

  autovacuum=**true**, **autovacuum_max_workers**=**6**, **autovacuum_max_workers_hstore**=**3**.

- enable_disaster_cstore

  Specifies whether fine-grained DR will be enabled for column-store tables. This parameter only takes effect on column-store tables whose COLVERSION is 2.0 and cannot be set to **on** if **enable_hstore** is **on**. This parameter is supported by version 8.2.0.100 or later clusters.

  Default value: **off**

  CAUTION

  Before enabling this function, set the GUC parameter **enable_metadata_tracking** to **on**. Otherwise, fine-grained DR may fail to be enabled.

- SUB_PARTITION_COUNT

  Specifies the number of level-2 partitions. This parameter specifies the number of level-2 partitions during data import. This parameter is configured during table creation and cannot be modified after table creation. You are not advised to set the default value, which may affect the import and query performance.

  Value range: 1 to 1024

  Default value: **32**

- DELTAROW_THRESHOLD

  Specifies the maximum number of rows (**SUB_PARTITION_COUNT** x **DELTAROW_THRESHOLD**) to be imported to the delta table.

  Value range: 0 to 60000

  Default value: **60000**

- COLVERSION

  Specifies the version of the storage format. HStore tables support only version 2.0, and **enable_hstore_opt** tables support versions 2.0 and 3.0.

  Options:

  1.0: Each column in a column-store table is stored in a separate file. The file name is **relfilenode.C1.0**, **relfilenode.C2.0**, **relfilenode.C3.0**, or similar.

  2.0: All columns of a column-store table are combined and stored in a file. The file is named **relfilenode.C1.0**.

  Default value: **2.0**

- enable_binlog

  Specifies whether to enable the binlog function for the HStore table. This parameter is supported only by clusters of version 8.3.0.100 or later.

  Value range: **on** and **off**

  Default value: **off**

- enable_binlog_timestamp

  Determines whether to enable the binlog function with timestamps for HStore tables. This parameter and **enable_binlog** cannot be enabled at the same time. Only clusters of 9.1.0.200 and later versions support this parameter.

  Value range: **on** and **off**

  Default value: **off**

- DISTRIBUTE BY

  Specifies how the table is distributed or replicated between DNs.

  Options:

  HASH (column_name): Each row of the table will be placed into all the DNs based on the hash value of the specified column.

- TO { GROUP groupname | NODE (nodename [, ...]) }

  TO GROUP specifies the Node Group in which the table is created. Currently, it cannot be used for HDFS tables. **TO NODE** is used for internal scale-out tools.

- PARTITION BY

  Specifies the initial partition of an HStore table.

- secondary_part_column

  Specifies the name of a level-2 partition column in a column-store table. Only one column can be specified as the level-2 partition column. This parameter applies only to HStore column-store tables. This parameter is supported only by clusters of version 8.3.0 and later. V3 tables do not support this parameter and will use hashbucket pruning.

NOTE

- The column specified as a level-2 partition column cannot be deleted or modified.
- The level-2 partition column can be specified only when a table is created. After a table is created, the level-2 partition column cannot be modified.
- You are not advised to specify a distribution column as a level-2 partition column.
- The level-2 partition column determines how the table is logically split into hash partitions on DNs, which enhances the query performance for that column.

– secondary_part_num

Specifies the number of level-2 partitions in a column-store table. This parameter applies only to HStore column-store tables. This parameter is supported only by clusters of version 8.3.0 and later. V3 tables do not support this parameter and will use hashbucket pruning.

Value range: 1 to 32

Default value: **8**

NOTE

- This parameter can be specified only when **secondary_part_column** is specified.
- The number of level-2 partitions can be specified only when a table is created and cannot be modified after the table is created.
- You are not advised to change the default value, which may affect the import and query performance.

Example

Create a simple HStore table.

```
CREATE TABLE warehouse_t1
(
   W_WAREHOUSE_SK            INTEGER               NOT NULL,
   W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
   W_WAREHOUSE_NAME          VARCHAR(20)                   ,
   W_WAREHOUSE_SQ_FT         INTEGER                       ,
   W_STREET_NUMBER           CHAR(10)                      ,
   W_STREET_NAME             VARCHAR(60)                   ,
   W_STREET_TYPE             CHAR(15)                      ,
   W_SUITE_NUMBER            CHAR(10)                      ,
   W_CITY                    VARCHAR(60)                   ,
   W_COUNTY                  VARCHAR(30)                   ,
   W_STATE                   CHAR(2)                       ,
   W_ZIP                     CHAR(10)                      ,
   W_COUNTRY                 VARCHAR(20)                   ,
   W_GMT_OFFSET              DECIMAL(5,2)
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);

CREATE TABLE warehouse_t2 (LIKE warehouse_t1 INCLUDING ALL);
```

3.2 INSERT

Function

Insert one or more rows of data into an HStore table.

NOTE

- To use hybrid data warehouse capabilities, choose the storage-compute coupled architecture when you create a GaussDB(DWS) cluster on the console and ensure the vCPU to memory ratio is 1:4 when setting up cloud disk flavors. For more information, see **Data Warehouse Flavors**.
- When setting up a GaussDB(DWS) cluster, make sure to have a vCPU to memory ratio of 1:8 for standard data warehouses and a ratio of 1:4 for hybrid data warehouses. You can distinguish a standard data warehouse from a real-time data warehouse by comparing their vCPU to memory ratios.

Precautions

- If the data to be inserted at a time is greater than or equal to the value of the table-level parameter **DELTAROW_THRESHOLD**, the data is directly inserted into the primary table to generate a compression unit (CU).
- If the data to be inserted is smaller than **DELTAROW_THRESHOLD**, a record of the type **I** will be inserted into the delta table. The data will be serialized and stored in the **values** field of the record.
- CUIDs are allocated to the data in the delta table and the primary table in a unified manner.
- The data inserted into the delta table depends on AUTOVACUUM to merge to primary table CUs.

Syntax

```
INSERT [/*+ plan_hint */] [ IGNORE | OVERWRITE ] INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
   { DEFAULT VALUES
   | VALUES {( { expression | DEFAULT } [, ...] ) }[, ...] | query }
```

Parameters

- **table_name**

  Specifies the name of the target table.

  Value range: an existing table name

- **AS**

  Specifies an alias for the target table *table_name*. *alias* indicates the alias name.

- **column_name**

  Specifies the name of a column in a table.

- **query**

  Specifies a query statement (**SELECT** statement) that uses the query result as the inserted data.

Example

Create the **reason_t1** table.

```
-- Create the reason_t1 table.
CREATE TABLE reason_t1
(
   TABLE_SK         INTEGER              ,
   TABLE_ID         VARCHAR(20)          ,
   TABLE_NA         VARCHAR(20)
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);
```


Insert a record into a table.

```
INSERT INTO reason_t1(TABLE_SK, TABLE_ID, TABLE_NA) VALUES (1, 'S01', 'StudentA');
```

Insert records into the table.

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
SELECT * FROM reason_t1 ORDER BY 1;
 TABLE_SK | TABLE_ID | TABLE_NAME
----------+----------+------------
        1 |      S01 |   StudentA
        2 |      T01 |   TeacherA
        3 |      T02 |   TeacherB
(3 rows)
```

3.3 DELETE

Function

Delete data from an HStore table.

NOTE

- To use hybrid data warehouse capabilities, choose the storage-compute coupled architecture when you create a GaussDB(DWS) cluster on the console and ensure the vCPU to memory ratio is 1:4 when setting up cloud disk flavors. For more information, see **Data Warehouse Flavors**.
- When setting up a GaussDB(DWS) cluster, make sure to have a vCPU to memory ratio of 1:8 for standard data warehouses and a ratio of 1:4 for hybrid data warehouses. You can distinguish a standard data warehouse from a real-time data warehouse by comparing their vCPU to memory ratios.

Precautions

- To delete all the data from a table, you are advised to use the **TRUNCATE** syntax to improve performance and reduce table bloating.
- If a single record is deleted from an HStore table, a record of the type **D** will be inserted into the delta table. The memory update chain will also be updated to manage concurrency.
- If multiple records are deleted from an HStore table at a time, a record of the type **D** will be inserted for the consecutive deleted records in each CU.
- In concurrent deletion scenarios, operations on the same CU will get queued in traditional column-store tables and result in low performance. For HStore tables, the operations can be concurrently performed, and the deletion performance can be more than 100 times that of column-store tables.
- The syntax is fully compatible with column storage. For more information, see the **UPDATE** syntax.

Syntax

```
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
   [ USING using_list ]
   [ WHERE condition ]
```

Parameters

- **ONLY**

If **ONLY** is specified, only that table is deleted. If **ONLY** is not specified, this table and all its sub-tables are deleted.

- **table_name**

  Specifies the name (optionally schema-qualified) of a target table.

  Value range: an existing table name

- **alias**

  Specifies the alias for the target table.

  Value range: a string. It must comply with the naming convention.

- **using_list**

  Specifies the **USING** clause.

- **condition**

  Specifies an expression that returns a value of type boolean. Only rows for which this expression returns **true** will be deleted.

Example

Create the **reason_t2** table.

```
CREATE TABLE reason_t2
(
   TABLE_SK         INTEGER              ,
   TABLE_ID         VARCHAR(20)          ,
   TABLE_NA         VARCHAR(20)
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);
INSERT INTO reason_t2 VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
```

Use the **WHERE** condition for deletion.

```
DELETE FROM reason_t2 WHERE TABLE_SK = 2;
DELETE FROM reason_t2 AS rt2 WHERE rt2.TABLE_SK = 2;
```

Use the **IN** syntax for deletion.

```
DELETE FROM reason_t2 WHERE TABLE_SK in (1,3);
```

3.4 UPDATE

Function

Update specified data in an HStore table.

NOTE

- To use hybrid data warehouse capabilities, choose the storage-compute coupled architecture when you create a GaussDB(DWS) cluster on the console and ensure the vCPU to memory ratio is 1:4 when setting up cloud disk flavors. For more information, see **Data Warehouse Flavors**.
- When setting up a GaussDB(DWS) cluster, make sure to have a vCPU to memory ratio of 1:8 for standard data warehouses and a ratio of 1:4 for hybrid data warehouses. You can distinguish a standard data warehouse from a real-time data warehouse by comparing their vCPU to memory ratios.

Precautions

- Similar to column storage, the UPDATE operation on an HStore table in the current version involves DELETE and INSERT. You can configure a global GUC

parameter to control the lightweight UPDATE of HStore. In the current version, the lightweight UPDATE is disabled by default.

- In concurrent update scenarios, operations on the same CU will cause lock conflicts in traditional column-store tables and result in low performance. For HStore tables, the operations can be concurrently performed, and the update performance can be more than 100 times that of column-store tables.

Syntax

```
UPDATE [/*+ plan_hint */] [ ONLY ] table_name [ * ] [ [ AS ] alias ]
SET {column_name = { expression | DEFAULT }
    |( column_name [, ...] ) = {( { expression | DEFAULT } [, ...] ) |sub_query }}[, ...]
    [ FROM from_list] [ WHERE condition ];
```

Parameters

- **plan_hint** clause

  Following the keyword in the **/*+ */** format, hints are used to optimize the plan generated by a specified statement block. For details, see **Hint-based Tuning**.

- **table_name**

  Name (optionally schema-qualified) of the table to be updated.

  Value range: an existing table name

- **alias**

  Specifies the alias for the target table.

  Value range: a string. It must comply with the naming convention.

- **expression**

  Specifies a value assigned to a column or an expression that assigns the value.

- **DEFAULT**

  Sets the column to its default value.

  The value is **NULL** if no specified default value has been assigned to it.

- **from_list**

  A list of table expressions, allowing columns from other tables to appear in the **WHERE** condition and the update expressions. This is similar to the list of tables that can be specified in the **FROM** clause of a **SELECT** statement.

  NOTICE

  Note that the target table must not appear in the **from_list**, unless you intend a self-join (in which case it must appear with an alias in the **from_list**).

- **condition**

  An expression that returns a value of type **boolean**. Only rows for which this expression returns **true** are updated.

Example

Create the **reason_update** table.


```
CREATE TABLE reason_update
(
   TABLE_SK         INTEGER              ,
   TABLE_ID         VARCHAR(20)          ,
   TABLE_NA         VARCHAR(20)
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);
```

Insert data to the **reason_update** table.

```
INSERT INTO reason_update VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
```

Perform the UPDATE operation on the **reason_update** table.

```
UPDATE reason_update SET TABLE_NA = 'TeacherD' where TABLE_SK = 3;
```

3.5 UPSERT

Function

HStore is compatible with the **UPSERT** syntax. You can add one or more rows to a table. When a row duplicates an existing primary key or unique key value, the row will be ignored or updated.

NOTE

- To use hybrid data warehouse capabilities, choose the storage-compute coupled architecture when you create a GaussDB(DWS) cluster on the console and ensure the vCPU to memory ratio is 1:4 when setting up cloud disk flavors. For more information, see **Data Warehouse Flavors**.
- When setting up a GaussDB(DWS) cluster, make sure to have a vCPU to memory ratio of 1:8 for standard data warehouses and a ratio of 1:4 for hybrid data warehouses. You can distinguish a standard data warehouse from a real-time data warehouse by comparing their vCPU to memory ratios.

Precautions

- The **UPSERT** statement of updating data upon conflict can be executed only when the target table contains a primary key or unique index.
- Similar to column storage, an update operation performed using **UPSERT** on an HStore table in the current version involves DELETE and INSERT.
- In concurrent UPSERT scenarios, operations on the same CU will cause lock conflicts in traditional column-store tables and result in low performance. For HStore tables, the operations can be concurrently performed, and the upsert performance can be more than 100 times that of column-store tables.

Syntax

Table 3-2 UPSERT syntax

Syntax	Update Data Upon Conflict	Ignore Data Upon Conflict
Syntax 1: No index is specified.	INSERT INTO ON DUPLICATE KEY UPDATE	INSERT IGNORE INSERT INTO ON CONFLICT DO NOTHING

Syntax	Update Data Upon Conflict	Ignore Data Upon Conflict
Syntax 2: The unique key constraint can be inferred from the specified column name or constraint name.	INSERT INTO ON CONFLICT(...) DO UPDATE SET INSERT INTO ON CONFLICT ON CONSTRAINT con_name DO UPDATE SET	INSERT INTO ON CONFLICT(...) DO NOTHING INSERT INTO ON CONFLICT ON CONSTRAINT con_name DO NOTHING

Parameters

In syntax 1, no index is specified. The system checks for conflicts on all primary keys or unique indexes. If a conflict exists, the system ignores or updates the corresponding data.

In syntax 2, a specified index is used for conflict check. The primary key or unique index is inferred from the column name, the expression that contains column names, or the constraint name specified in the **ON CONFLICT** clause.

- Unique index inference

  Syntax 2 infers the primary key or unique index by specifying the column name or constraint name. You can specify a single column name or multiple column names by using an expression. Example: **column1, column2, column3**

- **UPDATE** clause

  The **UPDATE** clause can use **VALUES(colname)** or **EXCLUDED.colname** to reference inserted data. **EXCLUDED** indicates the rows that should be excluded due to conflicts.

- **WHERE** clause
  - The **WHERE** clause is used to determine whether a specified condition is met when data conflict occurs. If yes, update the conflict data. Otherwise, ignore it.
  - Only syntax 2 of **Update Data Upon Conflict** can specify the **WHERE** clause, that is, **INSERT INTO ON CONFLICT(...) DO UPDATE SET WHERE**.

Example

Create table **reason_upsert** and insert data into it.

```
CREATE TABLE reason_upsert
(
  a    int primary key,
  b    int,
  c    int
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);
INSERT INTO reason_upsert VALUES (1, 2, 3);
```


Ignore conflicting data.

```
INSERT INTO reason_upsert  VALUES (1, 4, 5),(2, 6, 7) ON CONFLICT(a) DO NOTHING;
```

Update conflicting data.

```
INSERT INTO reason_upsert  VALUES (1, 4, 5),(3, 8, 9) ON CONFLICT(a) DO UPDATE SET b = EXCLUDED.b,
c = EXCLUDED.c;
```

3.6 MERGE INTO

Function

The **MERGE INTO** statement is used to conditionally match data in a target table with that in a source table. If data matches, **UPDATE** is executed on the target table; if data does not match, **INSERT** is executed. You can use this syntax to run **UPDATE** and **INSERT** at a time for convenience.

NOTE

- To use hybrid data warehouse capabilities, choose the storage-compute coupled architecture when you create a GaussDB(DWS) cluster on the console and ensure the vCPU to memory ratio is 1:4 when setting up cloud disk flavors. For more information, see **Data Warehouse Flavors**.
- When setting up a GaussDB(DWS) cluster, make sure to have a vCPU to memory ratio of 1:8 for standard data warehouses and a ratio of 1:4 for hybrid data warehouses. You can distinguish a standard data warehouse from a real-time data warehouse by comparing their vCPU to memory ratios.

Precautions

In concurrent **MERGE INTO** scenarios, the update operations triggered on the same CU will cause lock conflicts in traditional column-store tables and result in low performance. For HStore tables, the operations can be concurrently performed, and the **MERGE INTO** performance can be more than 100 times that of column-store tables.

Syntax

```
MERGE INTO table_name [ [ AS ] alias ]
USING { { table_name | view_name } | subquery } [ [ AS ] alias ]
ON ( condition )
[
  WHEN MATCHED THEN
  UPDATE SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
  [ WHERE condition ]
]
[
  WHEN NOT MATCHED THEN
  INSERT { DEFAULT VALUES |
  [ ( column_name [, ...] ) ] VALUES ( { expression | DEFAULT } [, ...] ) [, ...] [ WHERE condition ] }
];
```

Parameters

- **INTO** clause

  Specifies the target table that is being updated or has data being inserted.

  - **table_name**

Specifies the name of the target table.

– **alias**

Specifies the alias for the target table.

Value range: a string. It must comply with the naming convention.

- **USING** clause

  Specifies the source table, which can be a table, view, or subquery.

- **ON** clause

  Specifies the condition used to match data between the source and target tables. Columns in the condition cannot be updated. The **ON** association condition can be **ctid**, **xc_node_id**, or **tableoid**.

- **WHEN MATCHED** clause

  Performs **UPDATE** if data in the source table matches that in the target table based on the condition.

  NOTE

  Distribution columns, system catalogs, and system columns cannot be updated.

- **WHEN NOT MATCHED** clause

  Specifies that the INSERT operation is performed if data in the source table does not match that in the target table based on the condition.

  NOTE

  - An **INSERT** clause can contain only one **VALUES**.
  - The sequence of **WHEN NOT MATCHED** and **WHEN NOT MATCHED** clauses can be exchanged. One of them can be omitted, but they cannot be omitted at the same time.
  - Two **WHEN MATCHED** or **WHEN NOT MATCHED** clauses cannot be specified at the same time.

Example

Create a target for **MERGE INTO**.

```
CREATE TABLE target(a int, b int)WITH(ORIENTATION = COLUMN, ENABLE_HSTORE = ON);
INSERT INTO target VALUES(1, 1),(2, 2);
```

Create a data source table.

```
CREATE TABLE source(a int, b int)WITH(ORIENTATION = COLUMN, ENABLE_HSTORE = ON);
INSERT INTO source VALUES(1, 1),(2, 2),(3, 3),(4, 4),(5, 5);
```

Run the **MERGE INTO** command.

```
MERGE INTO target t
USING source s
ON (t.a = s.a)
WHEN MATCHED THEN
  UPDATE SET t.b = t.b + 1
WHEN NOT MATCHED THEN
  INSERT VALUES (s.a, s.b) WHERE s.b % 2 = 0;
```

3.7 SELECT

Function

Read data from an HStore table.

NOTE

- To use hybrid data warehouse capabilities, choose the storage-compute coupled architecture when you create a GaussDB(DWS) cluster on the console and ensure the vCPU to memory ratio is 1:4 when setting up cloud disk flavors. For more information, see **Data Warehouse Flavors**.
- When setting up a GaussDB(DWS) cluster, make sure to have a vCPU to memory ratio of 1:8 for standard data warehouses and a ratio of 1:4 for hybrid data warehouses. You can distinguish a standard data warehouse from a real-time data warehouse by comparing their vCPU to memory ratios.

Precautions

- Currently, neither column-store tables and HStore tables support the **SELECT FOR UPDATE** syntax.
- When a SELECT query is performed on an HStore table, the system will scan the data in column-store primary table CUs, the delta table, and the update information in each row in the memory. The three types of information will be combined before returned.
- If data is queried based on the primary key index or unique index,

  For traditional column-store tables, the unique index stores both the data location information (blocknum, offset) of the row-store Delta table and the data location information (cuid, offset) of the column-store primary table. After the data is merged to the primary table, a new index tuple will be inserted, and the index will keep bloating.

  For HStore tables, global CUIDs are allocated in a unified manner. Therefore, only cuid and offset are stored in index tuples. After data is merged, no new index tuples will be generated.

Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [/*+ plan_hint */] [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
[ ORDER BY {expression [ [ ASC | DESC | USING operator ] | nlssort_expression_clause ] [ NULLS { FIRST | LAST } ]} [, ...] ]
[ { [ LIMIT { count | ALL } ] [ OFFSET start [ ROW | ROWS ] ] } | { LIMIT start, { count | ALL } } ]
```

Parameters

- **DISTINCT [ON (expression [, ...])]**

  Removes all duplicate rows from the **SELECT** result set.

  ON (expression [, ...]) is only reserved for the first row among all the rows with the same result calculated using given expressions.

- **SELECT list**

  Indicates columns to be queried. Some or all columns (using wildcard character *) can be queried.

  You may use the **AS output_name** clause to give an alias for an output column. The alias is used for the displaying of the output column.

- **FROM** clause

  Indicates one or more source tables for **SELECT**.

  The **FROM** clause can contain the following elements:

- **WHERE clause**

  The **WHERE** clause forms an expression for row selection to narrow down the query range of **SELECT**. The condition is any expression that evaluates to a result of Boolean type. Rows that do not satisfy this condition will be eliminated from the output.

  In the **WHERE** clause, you can use the operator (+) to convert a table join to an outer join. However, this method is not recommended because it is not the standard SQL syntax and may raise syntax compatibility issues during platform migration. There are many restrictions on using the operator (+):

- **GROUP BY clause**

  Condenses query results into a single row all selected rows that share the same values for the grouped expressions.

- **HAVING clause**

  Selects special groups by working with the **GROUP BY** clause. The **HAVING** clause compares some attributes of groups with a constant. Only groups that matching the logical expression in the **HAVING** clause are extracted.

- **ORDER BY** clause

  Sorts data retrieved by **SELECT** in descending or ascending order. If the **ORDER BY** expression contains multiple columns:

Example

Create the **reason_select** table and insert data into the table.

```
CREATE TABLE reason_select
(
  r_reason_sk integer,
  r_reason_id integer,
  r_reason_desc character(100)
)WITH(ORIENTATION = COLUMN, ENABLE_HSTORE=ON);
INSERT INTO reason_select values(3, 1,'reason 1'),(10, 2,'reason 2'),(4, 3,'reason 3'),(10, 4,'reason 4');
```

Perform the GROUP BY operation.

```
SELECT COUNT(*), r_reason_sk FROM reason_select GROUP BY r_reason_sk;
```

Perform the HAVING filtering operation.

```
SELECT COUNT(*) c,r_reason_sk FROM reason_select GROUP BY r_reason_sk HAVING c > 1;
```

Perform the ORDER BY operation.

```
SELECT * FROM reason_select ORDER BY r_reason_sk;
```

3.8 ALTER TABLE

Function

Modify a table, including modifying the definition of a table, renaming a table, renaming a specified column in a table, adding or updating multiple columns, and changing a column-store table to an HStore table.

NOTE

- To use hybrid data warehouse capabilities, choose the storage-compute coupled architecture when you create a GaussDB(DWS) cluster on the console and ensure the vCPU to memory ratio is 1:4 when setting up cloud disk flavors. For more information, see **Data Warehouse Flavors**.
- When setting up a GaussDB(DWS) cluster, make sure to have a vCPU to memory ratio of 1:8 for standard data warehouses and a ratio of 1:4 for hybrid data warehouses. You can distinguish a standard data warehouse from a real-time data warehouse by comparing their vCPU to memory ratios.

Precautions

- You can set **enable_hstore** by using **ALTER** to change a column-store table to an HStore table, or to change it back. If **enable_delta** is set to **on**, **enable_hstore** cannot be set to **on**.
- For some ALTER operations (such as modifying column types, merging partitions, adding NOT NULL constraints, and adding primary key constraints), HStore tables need to merge data to the primary table and then perform ALTER, which may cause extra performance overhead. The impact on performance depends on the data volume in the delta table.
- When you add a column, do not use **ALTER** to specify other operations (for example, modifying the column type). An **ALTER** statement with only the **ADD COLUMN** parameter can achieve high performance, because it does not require FULL MERGE.
- The storage parameter **ORIENTATION** cannot be modified.

Modifying Table Attributes

Syntax:

```
ALTER TABLE [ IF EXISTS ] <table_name> SET  ( {ENABLE_HSTORE = ON} [, ... ] );
```

To change a column-store table to an HStore table, run the following command:

```
CREATE TABLE alter_test(a int, b int) WITH(ORIENTATION = COLUMN);
ALTER TABLE alter_test SET (ENABLE_HSTORE = ON);
```

NOTICE

To use HStore tables, set the following parameters, or the HStore performance will deteriorate severely. The recommended settings are as follows:

autovacuum_max_workers_hstore=3, autovacuum_max_workers=6, autovacuum=true

Adding a Column

Syntax:

```
ALTER TABLE [ IF EXISTS ] <table_name> ADD COLUMN <new_column> <data_type> [ DEFAULT
<default_value>];
```

Example:

Create the **alter_test2** table and add a column to it.


```
CREATE TABLE alter_test2(a int, b int) WITH(ORIENTATION = COLUMN,ENABLE_HSTORE = ON);
ALTER TABLE alter_test ADD COLUMN c int;
```

NOTE

When adding a column, you are not advised to use **ALTER** to specify other operations in the same SQL statement.

Renaming

Syntax:

```
ALTER TABLE [ IF EXISTS ] <table_name> RENAME TO <new_table_name>;
```

Example:

Create table **alter_test3** and rename it as **alter_new**.

```
CREATE TABLE alter_test3(a int, b int) WITH(ORIENTATION = COLUMN,ENABLE_HSTORE = ON);
ALTER TABLE alter_test3 RENAME TO alter_new;
```


4 Hybrid Data Warehouse Functions

hstore_light_merge(rel_name text)

Description: This function is used to manually perform lightweight cleanup on HStore tables and holds the level-3 lock of the target table.

Return type: int

Example:

```
SELECT hstore_light_merge('reason_select');
```

hstore_full_merge(rel_name text, partitionName text)

Description: This function is used to manually perform full cleanup on HStore tables. The second input parameter is optional and is used to specify a single partition for operations.

Return type: int

NOTICE

- This operation forcibly merges all the visible operations of the delta table to the primary table, and then creates an empty delta table. During this period, this operation holds the level-8 lock of the table.
- The duration of this operation depends on the amount of data in the delta table. You must enable the HStore clearing thread to ensure unnecessary data in the HStore table is cleared in a timely manner.
- The second parameter **partitionName** is only supported by clusters of version 8.3.0.100 and later. However, these versions do not allow calling this function via **call** because it lacks reload capability.

Example:

```
SELECT hstore_full_merge('reason_select', 'part1');
```

pgxc_get_small_cu_info(rel_name text, row_count int)

Description: Obtains the small CU information of the target table. The second parameter **row_count** is optional and indicates the small CU threshold. If the

number of live tuples in a CU is fewer than the threshold, the CU is considered as a small CU. The default value is **200**. This function is supported only by clusters of version 8.2.1.300 or later.

Return type: record

Return value:

node_name: DN name.

part_name: partition name. This column is empty for non-partitioned tables.

zero_cu_count: number of 0 CUs. If all data in a CU is deleted, the CU is called 0 CU.

small_cu_count: number of small CUs. When a CU has live data that is less than the threshold, the CU is called a small CU.

total_cu_count: total number of CUs.

sec_part_cu_num: number of CUs in each level-2 partition. This column is displayed only when **secondary_part_column** is specified. This field is available only in clusters of version 8.3.0 or later.

It should be noted that a CU may contain multiple columns.

Example:

```
SELECT * FROM pgxc_get_small_cu_info('hs');
node_name | part_name | zero_cu_count | small_cu_count | total_cu_count |
sec_part_cu_num
-----------+-----------+---------------+---------------+---------------+-----------------------------------------
 datanode1 |           |             0 |             4 |             4 | p1:1 p2:0 p3:1 p4:0 p5:1 p6:0 p7:1 p8:0
 datanode2 |           |             0 |             4 |             4 | p1:0 p2:1 p3:0 p4:1 p5:0 p6:1 p7:0 p8:1
(2 rows)
```

gs_hstore_compaction(rel_name text, row_count int)

Description: Merges small CUs of the target table. The second parameter **row_count** is optional and indicates the small CU threshold. If the number of live tuples in a CU is fewer than the threshold, the CU is considered as a small CU. The default value is **100**. This function is supported only by 8.2.1.300 and later versions.

Return type: int

Return value: **numCompactCU**, which indicates the number of small CUs to be merged.

NOTE

- A CU may contain multiple columns.
- The partition name cannot be input in the function. Currently, a single partition cannot be specified in this function.

Example:

```
SELECT gs_hstore_compaction('hs', 10);
```


pgxc_get_hstore_delta_info(rel_name text)

Description: This function is used to obtain the delta table information of the target table, including the delta table size and the number of **INSERT**, **DELETE**, and **UPDATE** records. This function is supported only by clusters of version 8.2.1.100 or later.

Return type: record

Return value:

node_name: DN name.

part_name: partition name. This column is set to **non-partition table** if the table is not a partitioned table.

live_tup: number of live tuples.

n_ui_type: number of records with a type of *ui* (small CU combination and upsert insertion through update). An **ui** record represents a single or batch insertion. This parameter is supported only by 8.3.0.100 and later versions.

n_i_type: number of records whose type is **i** (insert). An **i** record indicates one insertion, which can be single insertion or batch insertion.

n_d_type: number of records whose type is **d** (delete). One **d** record indicates one deletion, which can be single deletion or batch deletion.

n_x_type: number of records whose type is **x** (deletions generated by update).

n_u_type: number of records whose type is **u** (lightweight update).

n_m_type: number of records whose type is **m** (merge).

data_size: total size of the **delta** table (including the size of the index and **toast** data on the **delta** table).

Example:

```
SELECT * FROM pgxc_get_hstore_delta_info('hs_part');
 node_name | part_name | live_tup | n_ui_type | n_i_type | n_d_type | n_x_type | n_u_type | n_m_type |
data_size
-----------+-----------+----------+-----------+----------+----------+----------+----------+----------+-----------
 dn_1      | p1        |        2 |         0 |        2 |        0 |        0 |        0 |        0 |      8192
 dn_1      | p2        |        2 |         0 |        2 |        0 |        0 |        0 |        0 |      8192
 dn_1      | p3        |        2 |         0 |        2 |        0 |        0 |        0 |        0 |      8192
(3 rows)
```

pgxc_get_binlog_sync_point(rel_name text, slot_name text, checkpoint bool, node_id int)

Description: Obtains the synchronization point information corresponding to a slot from the **pg_binlog_slots** system catalog. This function is applicable only to tables with binlog or binlog timestamp enabled. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: record

Return value:

node_name: DN name

node_id: node ID

last_sync_point: last synchronization point

latest_sync_point: latest synchronization point

xmin: **xmin** corresponding to the synchronization point

Example:

```
SELECT * FROM pg_catalog.pgxc_get_binlog_sync_point('hstore_binlog_source', 'slot1', false, 0);
 node_name |   node_id   | last_sync_point | latest_sync_point | xmin
-----------+-------------+-----------------+-------------------+-------
 dn_2      | -1051926843 |               0 |             10512 | 10507
 dn_1      | -1300059100 |               0 |             10512 | 10508
(2 rows)
```

pgxc_get_binlog_changes(rel_name text, node_id int, start_csn bigint, end_csn bigInt)

Description: Obtains the incremental data of the target table within the specified synchronization point range on a specified DN. If **node_id** is set to **0**, all DNs are specified. This function is applicable only to tables with binlog or binlog timestamp enabled. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: record

Return value:

gs_binlog_sync_point: synchronization point

gs_binlog_event_sequence: sequence in the same transaction

gs_binlog_event_type: binlog type

gs_binlog_timestamp_us: timestamp of the binlog record. For the binlog table whose **enable_binlog_timestamp** is **false**, this column is empty.

value columns: data of each user field in the target table

Example:

```
SELECT * FROM pgxc_get_binlog_changes('hstore_binlog_source', 0, 0 , 9999999999);
 gs_binlog_sync_point | gs_binlog_event_sequence | gs_binlog_event_type | gs_binlog_timestamp_us | c1  | c2
| c3
----------------------+--------------------------+----------------------+------------------------+-----+----+----
               10516 |                        2 | I                    |      1731570520900211 | 100 |  1 |  1
               10517 |                        3 | d                    |      1731570520904425 | 100 |  1 |  1
               10518 |                        2 | I                    |      1731570520909055 | 200 |  1 |  1
               10519 |                        3 | B                    |      1731570520914102 | 200 |  1 |  1
               10519 |                        4 | U                    |      1731570520914154 | 200 |  2 |  1
```

pgxc_register_binlog_sync_point(rel_name text, slot_name text, node_id int, end_csn bigInt, checkpoint bool, xmin bigint)

Description: Registers synchronization points and can be used only for tables with binlog or binlog timestamp enabled. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: int

Return value: number of nodes that are successfully registered

Example:

```
SELECT pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, false, 100);
 pgxc_register_binlog_sync_point
---------------------------------
                               2
(1 row)
```

pgxc_consumed_binlog_records(rel_name text, node_id int)

Description: Obtains the consumption status of the target table on a specified DN. This function can be used only for tables with binlog or binlog timestamp enabled. This function is supported only by clusters of version 9.1.0.200 or later.

Return type: int

Return value: If **0** is returned, the binlog of the target table is not completely consumed (including all slots and checkpoint synchronization points). If **1** is returned, the binlog of the target table is completely consumed.

Example:

```
SELECT * FROM pgxc_consumed_binlog_records('hstore_binlog_source',0);
 pgxc_consumed_binlog_records
------------------------------
                            1
(1 row)
```

pgxc_get_binlog_cursor_by_timestamp(rel_name text, timestamp timestampTz, node_id int)

Description: Obtains information about the first binlog record after a specified time point in the target table. This function can be used only for tables with the binlog timestamp enabled.

This function is supported only by clusters of version 9.1.0.200 or later.

Return type: record

Return value:

node_name: DN name

node_id: node ID

latest_sync_point: latest synchronization point

binlog_sync_point: synchronization point of the first binlog record after the time point

binlog_timestamp_us: timestamp of the first binlog record after the time point

binlog_xmin: **xmin** recorded in the first binlog after the time point

Example:

```
SELECT * FROM pgxc_get_binlog_cursor_by_timestamp('hstore_binlog_source','2024-11-14
15:48:40.900211+08', 0);
 node_name |   node_id   | latest_sync_point | binlog_sync_point | binlog_timestamp_us | binlog_xmin
-----------+-------------+-------------------+-------------------+---------------------+-------------
 dn_2      | -1051926843 |             10532 |             10516 |    1731570520900211 |       10510
```



```
 dn_1      | -1300059100 |          10532 |          10518 |    1731570520909055 |       10510
(2 rows)
```

pgxc_get_binlog_cursor_by_syncpoint(rel_name text, csn int8, node_id int)

Description: Obtains the first binlog record after a specified synchronization point on the target table. This function can be used only for tables with the binlog timestamp enabled.

This function is supported only by clusters of version 9.1.0.200 or later.

Return type: record

Return value:

node_name: DN name

node_id: node ID

latest_sync_point: latest synchronization point

binlog_sync_point: synchronization point of the first binlog record after the time point

binlog_timestamp_us: timestamp of the first binlog record after the time point

binlog_xmin: **xmin** recorded in the first binlog after the time point

Example:

```
SELECT * FROM pgxc_get_binlog_cursor_by_syncpoint('hstore_binlog_source',10516,0);
 node_name |   node_id   | latest_sync_point | binlog_sync_point | binlog_timestamp_us | binlog_xmin
-----------+-------------+-------------------+-------------------+---------------------+-------------
 dn_1      | -1300059100 |          11187 |          10518 |    1731570520909055 |       10510
 dn_2      | -1051926843 |          11187 |          10516 |    1731570520900211 |       10510
(2 rows)
```

pgxc_get_cstore_dirty_ratio(rel_name text, partition_name)

Description: This function is used to obtain the cu, delta, and cudesc dirty page rates and sizes of the target table on each DN. Only **HStore_opt** tables are supported.

The **partition_name** parameter is optional. If the partition name is specified, only the information about the partition is returned. If the partition name is not specified and the table is a primary table, the information about all partitions is returned. It is supported only by clusters of version 9.1.0.100 or later.

Return type: record

Return value:

node_name: DN name

database_name: name of the database where the table is located

rel_name: primary table name

part_name: partition name

cu_dirty_ratio: dirty page rate of CU files

cu_size: CU file size

delta_dirty_ratio: dirty page rate of the delta table

delta_size: delta table size

cudesc_dirty_ratio: dirty page rate of the cudesc table

cudesc_size: cudesc table size

Example:

```
SELECT * FROM pgxc_get_cstore_dirty_ratio('hs_opt_part');
 node_name | database_name |      rel_name       | partition_name | cu_dirty_ratio | cu_size | delta_dirty_ratio
| delta_size | cudesc_dirty_ratio | cudesc_size
-----------+---------------+---------------------+----------------+----------------+---------+-------------------
+------------+--------------------+-------------
 dn_1      | postgres      | public.hs_opt_part | p1             |              0 |       0 |                 0 |      16384
|                  0 |       24576
 dn_1      | postgres      | public.hs_opt_part | p2             |              0 |       0 |                 0 |      16384
|                  0 |       24576
 dn_1      | postgres      | public.hs_opt_part | p3             |              0 |       0 |                 0 |      16384
|                  0 |       24576
 dn_1      | postgres      | public.hs_opt_part | p4             |              0 |       0 |                 0 |      16384
|                  0 |       24576
 dn_1      | postgres      | public.hs_opt_part | other          |              0 | 1105920 |                 0 |     524288
|                  0 |       40960
```


5 Hybrid Data Warehouse GUC Parameters

autovacuum

Parameter description: Specifies whether to start the automatic cleanup process (**autovacuum**).

Type: SIGHUP

Value range: Boolean

- **on** indicates the database automatic cleanup process is enabled.
- **off** indicates that the database automatic cleanup process is disabled.

Default value: **on**

autovacuum_compaction_rows_limit

Parameter description: Specifies the threshold of a small CU. A CU whose number of live tuples is less than the value of this parameter is considered as a small CU. This parameter is supported only by clusters of version 8.2.1.300 or later.

Type: USERSET

Value range: an integer ranging from -1 to 5000

Default value: **2500**

NOTICE

If the version is earlier than 9.1.0.100, do not set this parameter. Otherwise, duplicate primary key data may occur.

NOTE

- If the version is earlier than 9.1.0.100, value **–1** indicates that the 0 CU switch is disabled.
- In version 9.1.0.100, the default value of this parameter is **0**.
- In 9.1.0.200 and later versions, the default value of this parameter is **2500**.
- **You are advised not to modify this parameter. If you do need to modify this parameter, contact technical support.**

autovacuum_compaction_time_limit

Parameter description: Specifies the interval for clearing small CUs. Small CUs are merged at a specified interval. This parameter is supported only by clusters of version 8.2.1.300 or later.

Type: SIGHUP

Value range: an integer ranging from 0 to 10080. The unit is minute.

Default value: **0**

autovacuum_max_workers

Parameter description: Specifies the maximum number of autovacuum worker threads that can run at the same time. The upper limit of this parameter is related to the values of **max_connections** and **job_queue_processes**.

Type: SIGHUP

Value range: an integer

- The minimum value is **0**, indicating that autovacuum is not automatically performed.
- The theoretical maximum value is **262143**, and the actual maximum value dynamically changes. Formula: 262143 - **max_inner_tool_connections** - **max_connections** - **job_queue_processes** - **auxiliary threads** - **Number of autovacuum launcher threads** - 1. The number of auxiliary threads and the number of autovacuum launcher threads are specified by two macros. Their default values in the current version are **20** and **2**, respectively.

Default value: **4**

autovacuum_max_workers_hstore

Parameter description: Specifies the maximum number of concurrent automatic cleanup threads used for hstore tables in **autovacuum_max_workers**.

Type: SIGHUP

Value range: an integer

Default value: **1**

> **NOTE**
>
> To use HStore tables, set the following parameters, or the HStore performance will deteriorate severely. The recommended settings are as follows:
>
> **autovacuum_max_workers_hstore=3, autovacuum_max_workers=6, autovacuum=true**

hstore_buffer_size

Parameter description: Specifies the number of HStore CU slots. The slots are used to store the update chain of each CU, which significantly improves the update and query efficiency.

To prevent excessive memory usage, the system calculates a slot value based on the memory size, compares the slot value with the value of this parameter, and uses the smaller value of the two.

Type: POSTMASTER

Value range: an integer ranging from 100 to 100000

Default value: **100000**

gtm_option

Parameter description: Specifies the GTM running mode in GaussDB(DWS). This parameter is supported by version 8.2.1 or later clusters.

- GTM mode: In this mode, the GTM manages running transactions and allocates XIDs and CSNs in a unified manner.
- GTM-Lite mode: The GTM is only responsible for XID allocation and CSN update, and is no longer responsible for global transaction management. The GTM-Lite mode applies to TP scenarios with high concurrency and short queries. It can improve query performance while ensuring transaction consistency.
- GTM-Free mode: Distributed transactions support only external write consistency and do not support external read consistency. This mode does not take effect in hybrid data warehouses

Type: POSTMASTER

Value range: enumerated values

- **gtm** or **0**: The GTM mode is enabled.
- **gtm-lite** or **1**: The GTM-Lite mode is enabled.
- **gtm-free** or **2**: The GTM-Free mode starts.

Default value: **gtm**

NOTICE

1. Both GaussDB(DWS) and GTM instances have the **gtm_option** parameter with the same meaning. For GTM and GTM-Lite, the same mode must be set in GaussDB(DWS) and GTM. Otherwise, service errors may occur.
2. The GTM-Free mode can be enabled by setting **enable_gtm_free** to **on** or **gtm_option** to **gtm-free**.
3. To set the non-GTM-Free modes, set **enable_gtm_free** to **off**.
4. The GTM-Free mode takes effect only in hybrid cloud and ESL scenarios.

defer_xid_cleanup_time

Parameter description: Specifies the global OldestXmin maintenance period in GTM-Lite mode in the hybrid data warehouse. In each maintenance period, the CCN or FCN collects and delivers the values of global **OldestXmin**. This parameter is supported by version 8.2.1 or later clusters.

This parameter takes effect only in GTM-Lite mode. You are advised not to modify this parameter.

Type: SIGHUP

Value range: an integer ranging from 1 to INT_MAX. The unit is ms.

Default value: **5,000**.

enable_hstore_keyby_upsert

Parameter description: Specifies whether to enable batch upsert optimization for hstore tables, which can boost performance significantly if the front end guarantees that no two upsert operations affect the same row or column at the same time. You should turn on this parameter for such scenarios. This parameter is supported only in cluster 8.3.0 and later versions.

Type: USERSET

Value range: Boolean

on indicates that upsert optimization is enabled for hstore tables.

off indicates that the upsert process optimization on the hstore table is disabled and the old process is used.

Default value: **off**

autovacuum_asyncsort_rows_limit

Parameter description: This parameter specifies the row threshold for automatic asynchronous sorting. This parameter is supported only by clusters of version 9.1.0 or later.

Type: SIGHUP

Value range: an integer ranging from 120000 to 4200000

Default value: **600000**

autovacuum_asyncsort_size_limit

Parameter description: Specifies the space usage threshold for automatic asynchronous sorting. This parameter is supported only by clusters of version 9.1.0 or later.

Type: SIGHUP

Value range: an integer between **1048576** and **104857600**. The unit is KB.

Default value: 10485760 (10 GB)

autovacuum_asyncsort_time_limit

Parameter description: This parameter specifies the interval for automatic asynchronous sorting. Asynchronous sorting is triggered again only when the time elapsed since the last asynchronous sorting exceeds the value of **autovacuum_asyncsort_time_limit**. This parameter is supported only by clusters of version 9.1.0 or later.

Type: SIGHUP

Value range: an integer ranging from 0 to 10080. The unit is minute.

Default value: **10**

enable_hstore_binlog_table

Parameter description: This parameter specifies whether binlog tables can be created.

Type: SIGHUP

Value range: Boolean

- **on** indicates that binlog tables can be created.
- **off** indicates that binlog tables cannot be created.

Default value: **off**

enable_generate_binlog

Parameter description: Specifies whether binlogs are generated for DML operations on binlog tables in the current session. This parameter is supported only by clusters of version 9.1.0.200 or later.

Type: USERSET

Value range: Boolean

- **on** indicates that binlogs are generated.
- **off** indicates that binlogs are not generated.

Default value: **on**

binlog_consume_timeout

Parameter description: This parameter specifies the duration for cyclically determining whether all binlog records are consumed during binlog table scaling or VACUUM FULL operations. This parameter is supported only by 8.3.0.100 and later versions. Unit: second

Type: SIGHUP

Value range: an integer ranging from 0 to 86400

Default value: 3600

enable_hstoreopt_auto_bitmap

Parameter description: This parameter determines whether to automatically set bitmap columns by default when creating HStore Opt tables. This parameter is supported only by version 9.1.0.100 or later.

Type: SIGHUP

Value range: Boolean

- **on** indicates that the bitmap columns option is set by default.
- **off** indicates that the bitmap columns option is not set by default.

Default value: **off**

enable_cu_predicate_pushdown

Parameter description:

1. Function overview: This function is used to control whether to enable filter pushdown. Enabling this will enhance query performance, particularly when working with the **bitmap_columns** column and PCK sorting column. It applies to specific **WHERE**, **IS NULL**, and **IN** conditions. This parameter is supported only in 9.1.0.200 and later versions.
2. Supported column types:
   - Integer type: INT2, INT4, and INT8
   - Date and time type: DATE, TIMESTAMP, and TIMESTAMPTZ
   - String types: VARCHAR and TEXT
   - Numeral type: NUMERIC (a maximum of 19 characters)
3. Query conditions: This function supports multiple **WHERE** expressions, including:
   - **IN** expression: matches multiple values.
   - **IS NULL** / **IS NOT NULL** condition: checks whether the column value is null.
   - Comparison expressions: greater than (>), less than (<), equal to (=), and not equal to (<>), which is used for range query and exact match.

Type: USERSET

Value range: Boolean

- **on** indicates that filter pushdown is enabled.
- **off** indicates that filter pushdown is disabled.

Default value: **on**

enable_hstoreopt_insert_sort

Parameter description: This parameter determines whether to enable sorting (including VACUUM FULL) for importing data to the HStore Opt table.

This parameter is supported only by version 9.1.0.100 or later.

Type: SIGHUP

Value range: Boolean

- **on** indicates that sorting during import is enabled.
- **off** indicates that sorting during import (including VACUUM FULL) is disabled when asynchronous sorting is enabled (the value of **autovacuum_asyncsort_time_limit** is greater than 0). When asynchronous sorting is disabled, sorting during import (including VACUUM FULL) is still enabled.

Default value: **on**

6 Hybrid Data Warehouse Binlog

6.1 Subscribing to Hybrid Data Warehouse Binlog

Binlog Usage

The HStore table within the GaussDB(DWS) hybrid data warehouse offers binlog to facilitate the capture of database events. This enables the export of incremental data to third-party components like Flink. By consuming binlog data, you can synchronize upstream and downstream data, improving data processing efficiency.

Unlike traditional MySQL binlog, which logs all database changes and focuses on data recovery and replication. The GaussDB(DWS) hybrid data warehouse binlog is optimized for real-time data synchronization, recording DML operations—Insert, Delete, Update, and Upsert—while excluding DDL operations.

GaussDB(DWS) Binlog has the following advantages:

- Table-level on-demand switch: enables or disables binlog for specific tables as needed.
- Full incremental integrated consumption: supports full synchronization followed by real-time incremental consumption after a Flink task is started.
- Cleanup upon consumption: allows asynchronous clearing of incremental data after consumption, reducing space usage.

With Flink's real-time processing capabilities and Binlog, you can build a hybrid data warehouse efficiently without additional components like Kafka. The architecture is streamlined, and data flows efficiently, driven by Flink SQL.

Constraints and Limitations

1. Currently, only 8.3.0.100 and later versions support HStore and HStore Opt to record binlogs. V3 tables are in the trial commercial use phase. Before using them, contact technical support for evaluation.
2. Binlog requires a primary key, an HStore or HStore-opt table, and supports only hash distribution.
3. Binlog tables log DML operations like insert, delete, and update (upsert), excluding DDL operations.

4. Binlog tables do not support insert overwrite, altering distribution columns, enabling binlog on temporary tables, or partition operations like exchange, merge, or split.
5. Users can perform certain DDL operations (ADD COLUMN, DROP COLUMN, SET TYPE, VACUUM FULL, TRUNCATE),

   but these will reset incremental data and synchronization details.
6. The system waits for binlog consumption before further scaling. The default wait time is 1 hour, which can be set through the GUC parameter **binlog_consume_timeout**. Timeouts or errors will fail the scaling process.
7. The system waits for the consumption of binlog records before the VACUUM FULL operation is performed on a binlog table. The default wait time is 1 hour, which can be specified by the GUC parameter **binlog_consume_timeout**. Timeouts or errors will fail the VACUUM FULL process. Additionally, even if VACUUM FULL is executed for a partition table, a level-7 lock is added to the primary table of the partition, which blocks the insertion, update, or deletion of the entire table.
8. Binlog tables are backed up as standard HStore tables. Post-restoration, you must restart data synchronization as incremental data and sync details are reset.
9. The Binlog timestamp function is supported. This function can be enabled by activating **enable_binlog_timestamp**. Only the HStore and HStore Opt tables support this function. This constraint is supported only in 9.1.0.200 and later versions.

Binlog Formats and Principles

Table 6-1 binlog fields

Field	Type	Description
gs_binlog_sync_point	BIGINT	Binlog system field, which indicates the synchronization point. In common GTM mode, the value is unique and ordered.
gs_binlog_event_sequence	BIGINT	Binlog system field, which indicates the sequence of operations of the same transaction type.

Field	Type	Description
gs_binlog_event_type	CHAR	Binlog system field, which indicates the operation type of the current record. The options are as follows: • **I** refers to INSERT, indicating that a new record is inserted into the current binlog. • **d** refers to DELETE, indicating that a record is deleted from the current binlog. • **B** refers to BEFORE_UPDATE, indicating that the current binlog is a record before the update. • **U** refers to AFTER_UPDATE, indicating that the current binlog is a record after the update.
gs_binlog_timestamp_us	BIGINT	System field of Binlog, indicating the timestamp when the current record is saved to the database. This field is available only when the Binlog timestamp function is enabled. If the Binlog timestamp function is disabled, this field is left blank. Only 9.1.0.200 and later versions support this function.
user_column_1	User column	User-defined data column
...	...	...
usert_column_n	User column	User-defined data column

NOTE

- For each UPDATE (or UPSERT-triggered update), two binlog records—BEFORE_UPDATE and AFTER_UPDATE—are created. BEFORE_UPDATE verifies the accuracy of data processed by third-party components like Flink.
- During UPDATE and DELETE operations, the GaussDB(DWS) hybrid data warehouse generates BEFORE_UPDATE and DELETE binlogs without querying or populating all user columns, enhancing database import efficiency.
- Enabling binlog for an HStore table in the GaussDB(DWS) hybrid data warehouse is in fact the process of creation of a supplementary table. This table includes three system columns **gs_binlog_event_sync_point**, **gs_binlog_event_event_sequence**, and **gs_binlog_event_type**, and a **value** column that serializes all user columns.
- When the **enable_binlog_timestamp** parameter is enabled, binlog records are retained until the TTL expires, causing extra space overhead proportional to the data volume updated within the TTL. When **enable_binlog** is enabled, binlogs can be cleared asynchronously once consumed by downstream processes, significantly reducing space usage. Only 9.1.0.200 and later versions support this function.

Enabling Binlog

You can specify the table-level parameter **enable_binlog** when creating an HStore table to enable binlog.

```
CREATE TABLE hstore_binlog_source (
    c1  INT PRIMARY KEY,
    c2  INT,
    c3  INT
) WITH (
    ORIENTATION = COLUMN,
    enable_hstore_opt=true,
    enable_binlog=on,
    binlog_ttl = 86400
);
```

NOTE

- Binlog recording begins only after a synchronization point is registered for the task, not during the initial data import. Once binlog synchronization in Flink is activated, it periodically acquires the synchronization point and incremental data, then registers the synchronization point.
- The **binlog_ttl** parameter defaults to 86,400 seconds and is optional. If a registered synchronization point exceeds this TTL without undergoing incremental synchronization, it will be cleared. Subsequently, binlogs before the oldest synchronization point are asynchronously deleted to free up space.
- Space overhead: For a table with common binlog enabled, if incremental data can be consumed by downstream processes in a timely manner, the space can be cleared and reclaimed promptly.

Run the **ALTER** command to enable the binlog function for an existing HStore table.

```
CREATE TABLE hstore_binlog_source (
    c1  INT PRIMARY KEY,
    c2  INT,
    c3  INT
) WITH (
    ORIENTATION = COLUMN,
    enable_hstore_opt=true
);
ALTER TABLE hstore_binlog_source SET (enable_binlog=on);
```

Querying Binlogs

You can use the system functions provided by GaussDB(DWS) to query the binlog information of the target table on a specified DN and check whether the binlog is consumed by downstream processes.

```
-- Simulate Flink to call a system function to obtain the synchronization point. The parameters indicate the table name, slot name, whether the point is a checkpoint, and target DN (0 indicates all DNs).
select * from pg_catalog.pgxc_get_binlog_sync_point('hstore_binlog_source', 'slot1', false, 0);
select * from pg_catalog.pgxc_get_binlog_sync_point('hstore_binlog_source', 'slot1', true, 0);
-- Incremental binlogs are generated after additions, deletions, and modifications.
INSERT INTO hstore_binlog_source VALUES(100, 1, 1);
delete hstore_binlog_source where c1 = 100;
INSERT INTO hstore_binlog_source VALUES(200, 1, 1);
update hstore_binlog_source set c2 =2 where c1 = 200;
-- Simulate Flink to call a system function to query the binlog of a specified CSN range. The parameters indicate the table name, target DN (0 indicates all DNs), start CSN point, and end CSN point.
select * from pgxc_get_binlog_changes('hstore_binlog_source', 0, 0 , 9999999999);
```



```
postgres=# select * from pgxc_get_binlog_changes('hstore_binlog_source', 0, 0 , 9999999999);
 gs_binlog_sync_point | gs_binlog_event_sequence | gs_binlog_event_type | gs_binlog_timestamp_us | c1  | c2  | c3
----------------------+--------------------------+----------------------+------------------------+-----+-----+----
                10241 |                        2 | I                    |                        | 100 |   1 |  1
                10242 |                        3 | d                    |                        | 100 |   1 |  1
                10243 |                        4 | I                    |                        | 100 |   1 |  1
                10245 |                        5 | B                    |                        | 100 |   1 |  1
                10245 |                        6 | U                    |                        | 100 | 100 |  1
(5 rows)
```

Two **INSERT** operations generate two records with **gs_binlog_event_type** as **I**. The **DELETE** operation generates a record whose type is **d**. The **UPDATE** operation generates a **B** record for **BeforeUpdate** and a **U** record for **AfterUpdate**, indicating the values before and after the update.

You can call the system function **pgxc_consumed_binlog_records** to check whether the binlogs of the target table are consumed by all slots. The parameters indicate the target table name and target DN (**0** indicates all DNs).

```
-- Simulate Flink to call the system function to register a synchronization point. The parameters indicate the table name, slot name, registered point, whether the point is a checkpoint, and xmin corresponding to the point (provided when the synchronization point is obtained).
select pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, false, 100);
select pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, true, 100);
-- Check whether all binlogs in the table are consumed. If 1 is returned, all binlogs have been consumed by downstream slots.
select * from pgxc_consumed_binlog_records('hstore_binlog_source',0);
```

```
postgres=# select * from pgxc_consumed_binlog_records('hstore_binlog_source',0);
 pgxc_consumed_binlog_records
------------------------------
                            1
(1 row)
```

Enabling the Binlog Timestamp Function

If you need to read binlogs generated after a specified time point, specify the table-level parameter **enable_binlog_timestamp** when creating an HStore table to enable the binlog timestamp function of the HStore table. Only 9.1.0.200 and later versions support this function.

```
CREATE TABLE hstore_binlog_source(
    c1  INT PRIMARY KEY,
    c2  INT,
    c3  INT
) WITH (
    ORIENTATION = COLUMN,
    enable_hstore_opt=true,
    enable_binlog_timestamp =on,
    binlog_ttl = 86400
);
```

NOTE

- Binlog recording begins only after a synchronization point is registered for the task, not during the initial data import. Once the binlog timestamp is enabled, the system periodically acquires the synchronization point and incremental data, then registers the synchronization point.
- Binlog_ttl is an optional parameter. If not set, the default value is **86400** seconds (i.e., data is retained for one day by default). If the timestamp of the binlog record is greater than the current TTL, the binlog record will be deleted asynchronously.
- Space overhead: For a table with the binlog timestamp enabled, the binlog records recorded in the auxiliary table are retained until the TTL expires. This results in extra space overhead, which is proportional to the amount of data updated and imported into the database within the TTL.

Query the binlog on the table where the binlog timestamp function is enabled.


```
postgres=# select * from pgxc_get_binlog_changes('hstore_binlog_source', 0, 0 , 9999999999);
 gs_binlog_sync_point | gs_binlog_event_sequence | gs_binlog_event_type | gs_binlog_timestamp_us | c1  | c2 | c3
----------------------+--------------------------+----------------------+------------------------+-----+----+----
                10516 |                        2 | I                    |       1731570520900211 | 100 |  1 |  1
                10517 |                        3 | d                    |       1731570520904425 | 100 |  1 |  1
                10518 |                        2 | I                    |       1731570520909055 | 200 |  1 |  1
                10519 |                        3 | B                    |       1731570520914102 | 200 |  1 |  1
                10519 |                        4 | U                    |       173157052091454 | 200 |  2 |  1
(5 rows)
```

Convert **gs_binlog_timestamp_us** from the BigInt type to a readable timestamp.

```
select to_timestamp(1731569598408661/1000000);
```

```
postgres=#  select to_timestamp(1731570520900211/1000000);
         to_timestamp
-------------------------------
 2024-11-14 15:48:40.900211+08
(1 row)
```

To obtain the first binlog information of the target table after the specified time point on each DN (if the value is empty, no binlog exists after the time point).

```
select * from pgxc_get_binlog_cursor_by_timestamp('hstore_binlog_source','2024-11-14 15:33:18.40866+08', 0);
```

```
postgres=# select * from pgxc_get_binlog_cursor_by_timestamp('hstore_binlog_source','2024-11-14 15:48:40.900211+08', 0);
 node_name |   node_id   | latest_sync_point | binlog_sync_point | binlog_timestamp_us | binlog_xmin
-----------+-------------+-------------------+-------------------+---------------------+-------------
 dn_2      | -1051926843 |             10532 |             10516 |    1731570520900211 |       10510
 dn_1      | -1300059100 |             10532 |             10518 |    1731570520909055 |       10510
(2 rows)
```

Obtain the consumption progress of the table for which the binlog timestamp function is enabled.

The returned fields indicate the timestamp of the latest consumed binlog, the latest timestamp on the binlog, the CSN point of the latest consumed binlog, the latest CSN point on the binlog, and the number of unconsumed binlog records.

```
-- Simulate Flink to call the system function to register a synchronization point. The parameters indicate the table name, slot name, registered point, whether the point is a checkpoint, and xmin corresponding to the point (provided when the synchronization point is obtained).
select pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, false, 100);
select pgxc_register_binlog_sync_point('hstore_binlog_source', 'slot1', 0, 9999999999, true, 100);
-- Query the consumption progress of each slot in the target table.
select * from pgxc_get_binlog_consume_progress('hstore_binlog_source', 0);
```

```
postgres=# select * from pgxc_get_binlog_consume_progress('hstore_binlog_source', 0);
 node_name |   node_id   | slot_name | checkpoint | latest_consumed_timestamp |   latest_timestamp    | latest_consumed_csn | latest_csn | unconsumed_binlog_count
-----------+-------------+-----------+------------+---------------------------+-----------------------+---------------------+------------+-------------------------
 dn_1      | -1300059100 | slot1     | f          | 2024-11-14 15:48:40+08    | 2024-11-14 15:48:40+08 |               10519 |      10519 |                       0
 dn_1      | -1300059100 | slot1     | t          | 2024-11-14 15:48:40+08    | 2024-11-14 15:48:40+08 |               10519 |      10519 |                       0
 dn_2      | -1051926843 | slot1     | f          | 2024-11-14 15:48:40+08    | 2024-11-14 15:48:40+08 |               10517 |      10517 |                       0
 dn_2      | -1051926843 | slot1     | t          | 2024-11-14 15:48:40+08    | 2024-11-14 15:48:40+08 |               10517 |      10517 |                       0
(4 rows)
```

Preventing DML from Generating Binlogs

You can set the session-level parameter **enable_generate_binlog** to **off** to control the DML of the current session. When a table for which binlog is enabled is imported to the database, no binlog record is generated.

6.2 Real-Time Binlog Consumption by Flink

Precautions

- Currently, only versions 8.3.0.100 and later support HStore and HStore-opt for recording binlogs. V3 is currently in the trial commercial use phase and needs to be evaluated before being used.

- The Binlog function is only supported for Hstore and HStore-opt tables in GaussDB(DWS). These tables must have primary keys and one of parameters **enable_binlog** and **enable_binlog_timestamp** must be set to **on**.
- The name of the consumed binlog table cannot contain special characters, such as periods (.) and double quotation marks (").
- If multiple tasks consume binlog data of a single table, ensure that **binlogSlotName** of each task is unique.
- For maximum consumption speed, match task concurrency with the number of DNs in your GaussDB(DWS) cluster.
- If you use the sink capability of **dws-connector-flink** to write binlog data, pay attention to the following:
  - To ensure the data write sequence on DNs, set **connectionSize** to **1**.
  - If the primary key is updated on the source end or Flink is required for aggregation calculation, set **ignoreUpdateBefore** to **false**. Otherwise, you are not advised to set **ignoreUpdateBefore** to **false** (the default value is **true**).

Real-Time Binlog Consumption by Flink

Use DWS Connector to consume binlogs in real time. For details, see **DWS-Connector**.

If full data has been synchronized to the target end using other synchronization tools, and only incremental synchronization is required, you can call the following system function to update the synchronization points.

```
SELECT * FROM pg_catalog.pgxc_register_full_sync_point('table_name', 'slot_name');
```

Source Table DDL

The source autonomously assigns the appropriate Flink RowKind type (INSERT, DELETE, UPDATE_BEFORE, or UPDATE_AFTER) to each data row based on the operation type. This mechanism facilitates the synchronization of table data in a mirrored way, akin to the Change Data Capture (CDC) feature in MySQL and PostgreSQL databases.

```
CREATE TABLE test_binlog_source (
   a int,
   b int,
   c int,
   primary key(a) NOT ENFORCED
) with (
   'connector' = 'dws',
   'url' = 'jdbc:gaussdb://ip:port/gaussdb',
   'binlog' = 'true',
   'tableName' = 'test_binlog_source',
   'binlogSlotName' = 'slot',
   'username'='xxx',
   'password'='xxx')
```

Binlog Parameters

The following table describes the parameters involved in binlog consumption.

Table 6-2 Parameters

Parameter	Description	Data Type	Default Value
binlog	Specifies whether to read binlog information.	Boolean	false
binlogSlotName	Slot, which serves as an identifier. Multiple Flink tasks can simultaneously consume binlog data of the same table, so each task's **binlogSlotName** must be unique.	String	Name of the Flink mapping table
binlogBatchReadSize	Rows of binlog data read in batches.	Integer	5000
fullSyncBinlogBatchReadSize	Rows of binlog data fully read.	Integer	50000
binlogReadTimeout	Timeout for incrementally consuming binlog data, in milliseconds.	Integer	600000
fullSyncBinlogReadTimeout	Timeout for fully consuming binlog data, in milliseconds.	Long	1800000
binlogSleepTime	Sleep duration when no real-time binlog data is consumed, in milliseconds. The sleep duration with consecutive read failures is **binlogSleepTime** * failures, up to **binlogMaxSleepTime**. The value is reset after successful data read.	Long	500
binlogMaxSleepTime	Maximum sleep duration when no real-time binlog data is consumed, in milliseconds.	Long	10000
binlogMaxRetryTimes	Maximum number of retries after a binlog data consumption error.	Integer	1
binlogRetryInterval	Interval between retries after a binlog data consumption error, in milliseconds. Sleep duration during retry, which is calculated as **binlogRetryInterval** * (1~**binlogMaxRetryTimes**) + Random(100). The unit is millisecond.	Long	100

Parameter	Description	Data Type	Default Value
binlogParallelNum	Number of threads for consuming binlog data. This parameter is valid only when task concurrency is less than the number of DNs in the GaussDB(DWS) cluster.	Integer	3
connectionPoolSize	Number of connections in the JDBC connection pool.	Integer	5
needRedistribution	Determines compatibility with expansion redistribution. To ensure compatibility, upgrade the kernel to the corresponding version. If the kernel is an older version, set this parameter to **false**. If set to **true**, **restart-strategy** of Flink cannot be set to **none**.	Boolean	true
newSystemValue	Indicates whether to use the new system field when reading binlog data. (The kernel needs to be upgraded to the corresponding version. If the kernel is an older version, set this parameter to **false**.)	Boolean	true
checkNodeChangeInterval	Interval for detecting node changes. This parameter is valid only when **needRedistribution** is set to **true**.	Long	10000
connectionSocket-Timeout	Timeout interval for connection processing, in milliseconds. It can also be considered as the timeout interval for executing SQL statements on the client. The default value is **0**, which means that the timeout interval is not set.	Integer	0
binlogIgnoreUpdateBefore	Determines whether to filter out **before_update** records in binlogs and whether to return only primary key information for **delete** records. This parameter is supported only in 9.1.0.200 and later versions.	Boolean	false

Parameter	Description	Data Type	Default Value
binlogStartTime	Sets the time point from which binlogs are consumed can be set using the format **yyyy-MM-dd hh:mm:ss**. **enable_binlog_timestamp** must be enabled for the table. This parameter is supported only in 9.1.0.200 and later versions.	String	N/A
binlogSyncPointSize	Specifies the size of the synchronization point range for incrementally reading binlogs. This can control data flushing if the data volume is too large. This parameter is supported only in 9.1.0.200 and later versions.	Integer	5000

Data Synchronization Example

- On GaussDB(DWS):

  NOTE

  When creating a binlog table, set **enable_hstore_binlog_table** to **true**. You can run the **show enable_hstore_binlog_table** command to query the binlog table.

  -- Source table (generating binlogs)

  ```
  CREATE TABLE test_binlog_source(a int, b int, c int, primary key(a)) with(orientation=column, enable_hstore_opt=on, enable_binlog=true);
  ```

  -- Target table

  ```
  CREATE TABLE test_binlog_sink(a int, b int, c int, primary key(a)) with(orientation=column, enable_hstore_opt=on);
  ```

- On Flink:

  Run the following commands to perform complete data synchronization:

  ```
  -- Create a mapping table for the source table.
  CREATE TABLE test_binlog_source (
    a int,
    b int,
    c int,
    primary key(a) NOT ENFORCED
  ) with (
    'connector' = 'dws',
    'url' = 'jdbc:gaussdb://ip:port/gaussdb',
    'binlog' = 'true',
    'tableName' = 'test_binlog_source',
    'binlogSlotName' = 'slot',
    'username'='xxx',
    'password'='xxx');

  -- Create a mapping table for the target table:
  CREATE TABLE test_binlog_sink (
    a int,
    b int,
    c int,
  ```



```
  primary key(a) NOT ENFORCED
) with (
  'connector' = 'dws',
  'url' = 'jdbc:gaussdb://ip:port/gaussdb',
  'tableName' = 'test_binlog_sink',
  'ignoreUpdateBefore'='false',
  'connectionSize' = '1',
  'username'='xxx',
  'password'='xxx');

INSERT INTO test_binlog_sink select * from test_binlog_source;
```

Example of Using Java Programs

Create a source table and a target table.

```
-- source
create table binlog_test_source(a int, b int, c int, primary key(a)) with(orientation=column,
enable_hstore_opt=on, enable_binlog=true);
-- sink
create table binlog_test_sink(a int, b int, c int, primary key(a)) with(orientation=column,
enable_hstore_opt=on, enable_binlog=true);
```

Demo program:

```
public class BinlogDemo {

    //Name of the binlog table
    private static final String BINLOG_TABLE_NAME = "binlog_test_source";

    //Slot name of the binlog table
    private static final String BINLOG_SLOT_NAME = "binlog_test_slot";

    //Name of the table to be written
    private static final String SINK_TABLE_NAME = "binlog_test_sink";

    public static void main(String[] args) throws Exception {
        DwsConfig dwsConfig = buildDwsConfig();
        DwsClient dwsClient = new DwsClient(dwsConfig);

        TableSchema sourceTableSchema =
dwsClient.getTableSchema(TableName.valueOf(BINLOG_TABLE_NAME));
        TableSchema sinkTableSchema = dwsClient.getTableSchema(TableName.valueOf(SINK_TABLE_NAME));

        // Columns to be written
        List<String> sinkColumns = sinkTableSchema.getColumnNames();

        // Thread pool
        DwsConnectionPool dwsConnectionPool = new DwsConnectionPool(dwsConfig);
        //Queue for storing data
        BlockingQueue<BinlogRecord> queue = new LinkedBlockingQueue<>();
        //Columns to be synchronized
        List<String> sourceColumnNames = sourceTableSchema.getColumnNames();

        BinlogReader binlogReader = new BinlogReader(dwsConfig, queue, sourceColumnNames,
dwsConnectionPool);

        //Start the read task.
        binlogReader.start();
        binlogReader.getRecords();

        while (binlogReader.isStart()) {
            try {
                while (!queue.isEmpty() && !binlogReader.hasException()) {
                    // Read data.
                    BinlogRecord record = queue.poll();
                    if (Objects.isNull(record)) {
                        continue;
                    }
```



```
                BinlogRecordType type = BinlogRecordType.toBinlogRecordType(record.getType());
                List<Object> columnValues = record.getColumnValues();

                // Write data.
                if (BinlogRecordType.INSERT.equals(type) || BinlogRecordType.UPDATE_AFTER.equals(type)) {
                    Operate upsert = dwsClient.write(sinkTableSchema);
                    for (int i = 0; i < sinkColumns.size(); i++) {
                        upsert.setObject(i, columnValues.get(i), false);
                    }
                    upsert.commit();
                } else if (BinlogRecordType.DELETE.equals(type) ||
BinlogRecordType.UPDATE_BEFORE.equals(type)) {
                    Operate delete = dwsClient.delete(sinkTableSchema);
                    for (int i = 0; i < sinkColumns.size(); i++) {
                        String field = sinkColumns.get(i);
                        if (!sinkTableSchema.isPrimaryKey(field)) {
                            continue;
                        }
                        delete.setObject(i, columnValues.get(i), false);
                    }
                    delete.commit();
                }
            }
            binlogReader.checkException();
        } catch (Exception e) {
            throw new DwsClientException(ExceptionCode.GET_BINLOG_ERROR, "get binlog  has error", e);
        }
    }
  }

  private static DwsConfig buildDwsConfig() {
      //Initialize configuration information. (Only necessary parameters are listed. For more information
about the configuration, see the document.)
      TableConfig tableConfig = new TableConfig().withBinlog(true)
              .withNewSystemValue(true)
              .withNeedRedistribution(false)
              .withBinlogSlotName(BINLOG_SLOT_NAME);
      return DwsConfig.builder()
              .withUrl("Link information")
              .withUsername("Username")
              .withPassword ("Password")
              .withBinlogTableName(BINLOG_TABLE_NAME)
              .withTableConfig(BINLOG_TABLE_NAME, tableConfig)
              .build();
  }
}
```

