

Data Warehouse Service

Hybrid Data Warehouse

Issue 03
Date 2024-01-25



Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2024. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

Trademarks and Permissions



HUAWEI 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.

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: <https://www.huaweicloud.com/intl/en-us/>

Contents

1 Introduction to Hybrid Data Warehouse.....	1
2 Support and Constraints.....	6
3 Hybrid Data Warehouse Syntax.....	8
3.1 CREATE TABLE.....	8
3.2 INSERT.....	12
3.3 DELETE.....	13
3.4 UPDATE.....	14
3.5 UPSERT.....	16
3.6 MERGE INTO.....	17
3.7 SELECT.....	19
3.8 ALTER TABLE.....	21
4 Hybrid Data Warehouse Functions.....	23
5 Hybrid Data Warehouse GUC Parameters.....	24

1 Introduction to Hybrid Data Warehouse

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 details about how to create a hybrid data warehouse, see [Creating a GaussDB\(DWS\) 2.0 Cluster](#).
- Hot and cold data management is supported for HStore tables. For details, see [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

The hybrid data warehouse and standard data warehouse are two different types of GaussDB(DWS) products and have different usages. For details, see [Table 1-1](#).

Table 1-1 Comparison between hybrid and standard data warehouses

Type	Standard Data Warehouse	Hybrid Data Warehouse
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.
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.

Table Type	Row-Store	Column-Store	HStore
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.
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 merge, 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 .

Table Type	Row-Store	Column-Store	HStore
Application scenario	<ol style="list-style-type: none">1. OLTP transactions with frequent update and deletion operations2. Point queries (simple queries that are based on indexes and return a small amount of data)	<ol style="list-style-type: none">1. OLAP query and analysis2. A large volume of data is imported, and is rarely updated or deleted after the import.	<ol style="list-style-type: none">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. Currently, HStore and column storage do not support the use of VACUUM to clear dirty index data, and frequent updates may cause index bloat. This function will be supported in later versions.

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**.

Precautions

- 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
Table structure	Same as that defined for the column-store primary table.	Different from that defined for the primary table.
Function	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.
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.

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**
 - 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.
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**
 - 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.

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.

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_NAME   VARCHAR(20)  ,  
)WITH(ORIENTATION=COLUMN, ENABLE_HSTORE=ON);
```

Insert a record into a table.

```
INSERT INTO reason_t1(TABLE_SK, TABLE_ID, TABLE_NAME) 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.

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.

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_NAME 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_NAME = '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.

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 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.


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.

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.

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)

Description: This function is used to manually perform full cleanup on HStore tables.

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.

Example:

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

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_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: $262143 - \text{max_inner_tool_connections} - \text{max_connections} - \text{job_queue_processes} - \text{auxiliary threads} - \text{Number of autovacuum launcher threads} - 1$. 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: 3

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: 0

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

enable_hstore_lightupdate

Parameter description: Specifies whether to enable lightweight UPDATE for an HStore table. (When an UPDATE operation is performed on an HStore table, the system automatically determines whether lightweight UPDATE is required.)

Type: SIGHUP

Value range: Boolean

- **on** indicates that lightweight UPDATE is enabled for hstore tables.
- **off** indicates that lightweight UPDATE is disabled for hstore tables.

Default value: off

enable_hstore_merge_keepgtm

Parameter description: Specifies whether the MERGE in the autovacuum operation on column-store and hstore tables occupies slots in the GTM.

Type: SIGHUP

Value range: Boolean

- **true** indicates that it occupies slots in the GTM.
- **false** indicates that it does not occupy slots in the GTM.

Default value: 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 10,000,000

Default value: 100,000