Data Lake Insight

Hudi SQL Syntax Reference

Issue 01

Date 2026-10-15

HUAWEI

HUAWEI TECHNOLOGIES CO., LTD.

Copyright © Huawei Technologies Co., Ltd. 2026. 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 Technologies Co., Ltd.

Trademarks and Permissions

and other Huawei trademarks are trademarks 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 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.

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process.* For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 Hudi Table Overview 1
1.1 Constraints on Hudi Tables 1
1.2 Query Type 3
1.3 Storage Structure 5
2 DLI Hudi Metadata 7
3 DLI Hudi Development Specifications 9
3.1 Overview 9
3.2 Hudi Data Table Design Specifications 10
3.2.1 Hudi Table Model Design Specifications 10
3.2.2 Hudi Table Index Design Specifications 12
3.2.3 Hudi Table Partition Design Specifications 14
3.3 Hudi Data Table Management Operation Specifications 15
3.3.1 Hudi Data Table Compaction Specifications 15
3.3.2 Hudi Data Table Clean Specifications 18
3.3.3 Hudi Data Table Archive Specifications 19
3.4 Spark on Hudi Development Specifications 20
3.4.1 Parameter Specifications for Creating a Hudi Table with SparkSQL 20
3.4.2 Parameter Specifications for Incremental Reading of Hudi Table Data with Spark 21
3.4.3 Parameter Specifications for Spark Asynchronous Task Execution Table Compaction 21
3.4.4 Spark Table Data Maintenance Specifications 22
3.5 Bucket Tuning 22
3.5.1 Tuning a Bucket Index Table 22
3.5.2 Hudi Table Initialization 24
3.5.3 Real-Time Job Ingestion 25
3.5.4 Offline Compaction Configuration 26
4 Using Hudi to Develop Jobs in DLI 28
4.1 Submitting a Spark SQL Job in DLI Using Hudi 28
4.2 Submitting a Spark Jar Job in DLI Using Hudi 29
4.3 Submitting a Flink SQL Job in DLI Using Hudi 32
5 DLI Hudi SQL Syntax Reference 35
5.1 Hudi DDL Syntax 35
5.1.1 CREATE TABLE 35

 Copyright © Huawei Technologies Co., Ltd.

5.1.2 DROP TABLE 38
5.1.3 SHOW TABLE 39
5.1.4 TRUNCATE TABLE 40
5.2 Hudi DML Syntax 40
5.2.1 CREATE TABLE AS SELECT 41
5.2.2 INSERT INTO 42
5.2.3 MERGE INTO 44
5.2.4 UPDATE 46
5.2.5 DELETE 47
5.2.6 COMPACTION 48
5.2.7 ARCHIVELOG 49
5.2.8 CLEAN 50
5.2.9 CLEANARCHIVE 50
5.3 Hudi CALL COMMAND Syntax 52
5.3.1 CLEAN_FILE 52
5.3.2 SHOW_TIME_LINE 53
5.3.3 SHOW_HOODIE_PROPERTIES 54
5.3.4 ROLL_BACK 55
5.3.5 CLUSTERING 56
5.3.6 CLEANING 57
5.3.7 COMPACTION 59
5.3.8 SHOW_COMMIT_FILES 60
5.3.9 SHOW_FS_PATH_DETAIL 61
5.3.10 SHOW_LOG_FILE 63
5.3.11 SHOW_INVALID_PARQUET 64
5.4 Schema Evolution Syntax 65
5.4.1 ALTER COLUMN 66
5.4.2 ADD COLUMNS 67
5.4.3 RENAME COLUMN 69
5.4.4 RENAME TABLE 69
5.4.5 SET 70
5.4.6 DROP COLUMN 71
5.5 Configuring Default Values for Hudi Data Columns 71

6 Spark DataSource API Syntax Reference 74

6.1 API Syntax Description 74
6.2 Hudi Lock Configuration 77

7 Data Management and Maintenance 79

7.1 Hudi Compaction 79
7.2 Hudi Clean 81
7.3 Hudi Archive 81
7.4 Hudi Clustering 82

 Copyright © Huawei Technologies Co., Ltd.

8 Typical Hudi Configuration Parameters.....86

 Copyright © Huawei Technologies Co., Ltd.

1 Hudi Table Overview

1.1 Constraints on Hudi Tables

Hudi Table Type

- **Copy On Write**

  Copy-on-write (COW) tables store data in Parquet files. Internal update operations need to be performed by rewriting the original Parquet files.

  - Advantage: It is efficient because only one data file in the corresponding partition needs to be read.
  - Disadvantage: During data write, a previous copy needs to be copied and then a new data file is generated based on the previous copy. This process is time-consuming. Therefore, the data read by the read request lags behind.

- **Merge On Read**

  Merge-on-read (MOR) tables store data in a hybrid format combining columnar-based Parquet and row-based format Avro. Parquet files are used to store base data, and Avro files (also called log files) are used to store incremental data.

  - Advantage: Data is written to the delta log first, and the delta log size is small. Therefore, the write cost is low.
  - Disadvantage: Files need to be compacted periodically. Otherwise, there are a large number of fragment files. The read performance is poor because delta logs and old data files need to be merged.

Table 1-1 Trade-off of two table types

Trade-off	CopyOnWrite	MergeOnRead
Data latency	High	Low
Query latency	Low	High

 Copyright © Huawei Technologies Co., Ltd.

Trade-off	CopyOnWrite	MergeOnRead
Update cost (I/O)	High (rewriting the entire Parquet file)	Low
Parquet file size	Small (high update cost)	Large (low update cost)
Write amplification	High	Low (depending on the compaction policy)

Notes and Constraints for Using Hudi Tables

- Hudi supports the use of Spark SQL in the DDL/DML syntax. However, when using DLI's metadata to submit SparkSQL jobs, some SQL syntax that directly operates on OBS paths is not supported. For details, refer to **DLI Hudi SQL Syntax Reference**.
- When creating a Hudi table, you must correctly configure **primaryKey** and **preCombineField**. Otherwise, there may be data inconsistency between the expected and actual results.
- **When using the metadata service provided by DLI, creating DLI tables is not supported. Only OBS tables can be created, which means the table path must be configured through the LOCATION parameter.**
- When using the metadata service provided by LakeFormation, both internal and external tables are supported. Note that when dropping an internal table, the data will also be deleted synchronously.
- When submitting Spark SQL or Flink SQL jobs, there is no need to manually configure the **hoodie.write.lock.provider** item for Hudi. However, when submitting Spark Jar jobs, you must manually configure the item. Refer to **Hudi Lock Configuration** for more details.
- The version mapping between Hudi and the queue compute engine is:

Compute Engine	Version	Hudi Version
Spark	3.3.1	0.11.0
Flink	1.15	0.11.0

To determine the compute engine versions supported by a queue, follow these steps: Log in to the DLI console and choose **Resources** > **Queue Management** in the navigation pane on the left. Locate the queue you want to query on the queue management page. Click the icon next to queue name to expand its details. Look for the supported versions in **Supported Versions**. For SQL queues, you cannot switch versions, so checking the default version will indicate the current compute engine version in use.

 Copyright © Huawei Technologies Co., Ltd.

1.2 Query Type

Snapshot Queries

Snapshot queries allow you to read the latest snapshots generated by commit/compaction. For MOR tables, it also merges the content of the latest delta log files in the query, providing near real-time data retrieval.

Incremental Queries

Incremental queries only retrieve data that has been added after a given commit/compaction.

Read Optimized Queries

Read optimized queries are specifically optimized for MOR tables and only read the latest snapshots generated by commit/compaction (excluding delta log files).

Table 1-2 Trade-off between real-time queries and read optimized queries

Trade-off	Real-Time Queries	Read Optimized Queries
Data latency	Low	High
Query latency	Only for MOR tables, high (combining Parquet and delta log files)	Low (Parquet file reading performance)

COW Table Queries

- Real-time view reading (using SparkSQL an example): Directly read the Hudi table stored in the metadata service, where **$*{table_name}*** indicates the table name.

  ```
  select (fields or aggregate functions) from ${table_name};
  ```

- Real-time view reading (using a Spark Jar job as an example):

  Spark Jar jobs can read Hudi tables in two ways: using the Spark datasource API or submitting SQL queries through SparkSession.

  Set the configuration item **hoodie.datasource.query.type** to **snapshot** (which is also the default value).

  ```
  object HudiDemoScala {
    def main(args: Array[String]): Unit = {
      val spark = SparkSession
        .builder()
        .enableHiveSupport()
        .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
        .config("spark.sql.extensions", "org.apache.spark.sql.hudi.HoodieSparkSessionExtension")
        .appName("HudiIncrementalReadDemo")
        .getOrCreate();

      // 1. Read Hudi tables using the Spark datasource API.
      val dataFrame = spark.read.format("hudi")
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
    .option("hoodie.datasource.query.type", "snapshot") // snapshot is also the default value. You can
retain the default value.
    .load("obs://bucket/to_your_table"); // Specify the path of the Hudi table to read. DLI supports
only OBS paths.
   dataFrame.show(100);

   // 2. Read Hudi tables by submitting SQL queries through SparkSession, which requires
interconnection with the metadata service.
   spark.sql("select * from ${table_name}").show(100);
  }
}
```

- Incremental view reading (using Spark SQL as an example):

  Start by configuring:

  ```
  hoodie.${table_name}.consume.mode=INCREMENTAL
  hoodie.${table_name}.consume.start.timestamp=Start commit time
  hoodie.${table_name}.consume.end.timestamp=End commit time
  ```

  Run the following SQL statement:

  ```
  select (fields or aggregate functions) from ${table_name} where `_hoodie_commit_time`>'Start
  commit time' and `_hoodie_commit_time`<='End commit time' //This filtering condition is mandatory.
  ```

- Incremental view reading (using a Spark Jar job as an example):

  The **hoodie.datasource.query.type** configuration item must be set to **incremental**.

  ```
  object HudiDemoScala {
    def main(args: Array[String]): Unit = {
      val spark = SparkSession
        .builder()
        .enableHiveSupport()
        .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
        .config("spark.sql.extensions", "org.apache.spark.sql.hudi.HoodieSparkSessionExtension")
        .appName("HudiIncrementalReadDemo")
        .getOrCreate();

      val startTime = "20240531000000";
      val endTime = "20240531000000";
      spark.read.format("hudi")
        .option("hoodie.datasource.query.type", "incremental") // Specify the query type as incremental
  query.
        .option("hoodie.datasource.read.begin.instanttime", startTime)  // Specify the start commit for
  incremental pull.
        .option("hoodie.datasource.read.end.instanttime", endTime)  // Specify the end commit for
  incremental pull.
        .load("obs://bucket/to_your_table")  // Specify the path of the hudi table to read.
        .createTempView("hudi_incremental_temp_view");  // Register as a temporary Spark table.
      // The results must be filtered based on startTime and endTime. If endTime is not specified,
  filtering only needs to be done based on startTime.
      spark.sql("select * from hudi_incremental_temp_view where
  `_hoodie_commit_time`>'20240531000000' and `_hoodie_commit_time`<='20240531321456'")
        .show(100, false);
    }
  }
  ```

- Read optimized queries: Read optimized queries for COW tables is equivalent to snapshot queries.

MOR Table Queries

When using the metadata service in Spark SQL jobs or configuring HMS synchronization parameters, creating an MOR table will also create two additional tables: **$*{table_name}*_rt** and **$*{table_name}*_ro**. The table with the **rt** suffix represents real-time queries, while the table with the **ro** suffix represents read optimized queries. For example, if you create a Hudi table named **$*{table_name}*** using Spark SQL and synchronize it with the metadata service, two additional tables will be created in the database: **$*{table_name}*_rt** and **$*{table_name}*_ro**.

 Copyright © Huawei Technologies Co., Ltd.

- Real-time view reading (using Spark SQL as an example): Directly read the Hudi table with the **_rt** suffix in the same database.
  ```
  select count(*) from ${table_name}_rt;
  ```
- Real-time view reading (using a Spark Jar job as an example): Same as COW table operations, refer to the relevant COW table operations.
- Incremental view reading (using a Spark SQL job as an example): Same as COW table operations, refer to the relevant COW table operations.
- Incremental view reading (using a Spark Jar job as an example): Same as COW table operations, refer to the relevant COW table operations.
- Read optimized view reading (using a Spark Jar job as an example):

  The **hoodie.datasource.query.type** configuration item must be set to **read_optimized**.

  ```
  object HudiDemoScala {
    def main(args: Array[String]): Unit = {
      val spark = SparkSession.builder
        .enableHiveSupport
        .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
        .config("spark.sql.extensions", "org.apache.spark.sql.hudi.HoodieSparkSessionExtension")
        .appName("HudiIncrementalReadDemo")
        .getOrCreate
      spark.read.format("hudi")
        .option("hoodie.datasource.query.type", "read_optimized") // Specify the query type as read-optimized view.
        .load("obs://bucket/to_your_table") // Specify the path of the hudi table to read.
        .createTempView("hudi_read_optimized_temp_view")
      spark.sql("select * from hudi_read_optimized_temp_view").show(100)
    }
  }
  ```

1.3 Storage Structure

Hudi generates a Hudi table based on the specified storage path, table name, partition structure, and other attributes when writing data.

In the DLI environment, the data files of the Hudi table are stored in OBS, so they can be inspected by viewing the OBS files.

The following shows the storage structure of a Hudi table with multi-level partitioning using the COW method.

```
hudi_table
├── .hoodie  // Metadata folder
│   ├── .aux
│   ├── .cleanData
│   ├── .schema
│   ├── .temp
│   ├── hoodie.properties  // Metadata storage file of the Hudi table
│   ├── 20231030232758468.commit.requested  // Write record metadata into the COW table (representing planned)
│   ├── 20231030232758468.inflight  // Write record metadata into the COW table (representing in progress)
│   ├── 20231030232758468.commit  // Write record metadata into the COW table (representing completed)
│   └── ...
├── 2023  // Multi-level partition directory
│   ├── 10
│   │   ├── 30
│   │   │   ├── .hoodie_partition_metadata
│   │   │   ├── 425388c7-1a84-427e-a7b5-18e6745389b5_1-4-0_20231030232758468.parquet
│   │   │   ├── 425388c7-1a84-427e-a7b5-18e6745389b5_1-4-0_20231030524543412.parquet
│   │   │   ├── ...
```

 Copyright © Huawei Technologies Co., Ltd.

```
│   │   └── 425388c7-1a84-427e-a7b5-18e6745389b5_1-4-0_20231030744231276.parquet
│   └── ...
└── ...
```

2 DLI Hudi Metadata

Hudi Metadata Description

When creating a Hudi table, relevant metadata information about the table will be created in the metadata warehouse.

Hudi supports integration with DLI metadata and LakeFormation metadata (integration with LakeFormation metadata is only supported in Spark 3.3.1 or later), using the same integration method as Spark.

- DLI metadata can be viewed on the **Data Management** > **Databases and Tables** page of the DLI management console.
- LakeFormation metadata can be viewed on the LakeFormation management console.

Related Operations

- To connect a DLI SQL queue to DLI metadata, follow these steps:
  a. On the **SQL Editor** page of the DLI management console, click **dli** in the **Catalog** list.
  b. In the **Databases** drop-down list, select the database in the DLI metadata to connect.
- To connect a DLI general-purpose queue to DLI metadata, follow these steps:
  See **Using Spark Jobs to Access DLI Metadata**.
- To connect a DLI SQL queue to LakeFormation metadata, follow these steps:
  See **Connecting DLI to LakeFormation**.
- To connect a DLI general-purpose queue to LakeFormation metadata, follow these steps:
  See **Connecting DLI to LakeFormation**.
- DLI metadata permissions management
  You can manage the permissions for DLI metadata through DLI SQL permissions management or IAM authentication.
  – DLI SQL permissions management:
    i. Log in to the DLI console. In the navigation pane on the left, choose **Data Management** > **Databases and Tables**. On the displayed page,

 Copyright © Huawei Technologies Co., Ltd.

search for the database you want to authorize or click the name of a database and search for the table you want to authorize.

ii. Locate the database or table you want to authorize and click **Permissions** in the **Operation** column to view or add permissions.

See **Managing Database Resources on the DLI Console**.

- IAM authentication:

See "Application Scenarios of IAM Authentication" in **Permissions Management**.

- LakeFormation metadata permissions management

See **Connecting DLI to LakeFormation**.

 Copyright © Huawei Technologies Co., Ltd.

3 DLI Hudi Development Specifications

3.1 Overview

Scope

This section defines the rules for designing and developing lakehouse and stream and batch processing solutions using DLI-Hudi. These rules are applicable to table design and management, as well as job development in Hudi development scenarios.

It covers:

- Data table design
- Resource configuration
- Performance tuning
- Common troubleshooting
- Typical parameter settings

Terms

This section uses the following terms:

- **Rule**: a principle that must be followed during programming.
- **Suggestion**: a principle that must be considered during programming.
- **Description**: an explanation of the rule or suggestion in question.
- **Example**: positive and negative examples of a rule or suggestion.

Application Scope

- Design, develop, test, and maintain data storage and processing jobs based on DLI-Hudi.
- These design and development specifications are based on Spark 3.3.1 and Hudi 0.11.0.

 Copyright © Huawei Technologies Co., Ltd.

3.2 Hudi Data Table Design Specifications

3.2.1 Hudi Table Model Design Specifications

Rules

- Hudi tables must have a reasonable primary key.

  Hudi tables provide data update and idempotent write capabilities, which require the tables to have a primary key. Setting an inappropriate primary key can result in data duplication. The primary key can be a single key or a composite key, but it must not have null or empty values. Here is an example of setting a primary key.

  SparkSQL:

  ```
  // Specify the primary key using the primaryKey parameter. If the primary key is a composite one, separate them with commas (,).
  create table hudi_table (
  id1 int,
  id2 int,
  name string,
  price double
  ) using hudi
  options (
  primaryKey = 'id1,id2',
  preCombineField = 'price'
  );
  ```

  SparkDataSource:

  ```
  // Specify the primary key using the hoodie.datasource.write.recordkey.field parameter.
  df.write.format("hudi").
  option("hoodie.datasource.write.table.type", COPY_ON_WRITE).
  option("hoodie.datasource.write.precombine.field", "price").
  option("hoodie.datasource.write.recordkey.field", "id1,id2").
  ```

  FlinkSQL:

  ```
  // Specify the primary key using the hoodie.datasource.write.recordkey.field parameter.
  create table hudi_table(
  id1 int,
  id2 int,
  name string,
  price double
  ) partitioned by (name) with (
  'connector' = 'hudi',
  'hoodie.datasource.write.recordkey.field' = 'id1,id2',
  'write.precombine.field' = 'price')
  ```

- The **precombine** field must be configured for Hudi tables.

  During data synchronization, it is inevitable to encounter issues like duplicate data writes and data disorder, such as recovering abnormal data or restarting write programs. By setting a reasonable value for the **precombine** field, data accuracy can be ensured, and old data will not overwrite new data, achieving idempotent writes. The **precombine** field can be a timestamp of updates in the service table or the commit timestamp of the database. The **precombine** field must not have null or empty values. Here is an example of setting the **precombine** field.

  SparkSQL:

  ```
  // Specify the precombine field through the preCombineField parameter.
  create table hudi_table (
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
id1 int,
id2 int,
name string,
price double
) using hudi
options (
primaryKey = 'id1,id2',
preCombineField = 'price'
);
```

SparkDatasource:

```
// Specify the precombine field through the hoodie.datasource.write.precombine.field parameter.
df.write.format("hudi").
option("hoodie.datasource.write.table.type", COPY_ON_WRITE).
option("hoodie.datasource.write.precombine.field", "price").
option("hoodie.datasource.write.recordkey.field", "id1,id2").
```

Flink:

```
// Specify the precombine field through the write.precombine.field parameter.
create table hudi_table(
id1 int,
id2 int,
name string,
price double
) partitioned by (name) with (
'connector' = 'hudi',
'hoodie.datasource.write.recordkey.field' = 'id1,id2',
'write.precombine.field' = 'price')
```

- MOR tables are used for stream computing.

  Stream computing is low-latency real-time processing and requires high-performance streaming read/write capabilities. Among the Merge on Read (MOR) and Copy on Write (COW) models in Hudi tables, MOR tables have better performance for streaming read/write operations. Therefore, MOR tables are preferred for stream computing scenarios. Here is a comparison of the read/write performance between MOR and COW tables.

Dimension	MOR Table	COW Table
Stream write	High	Low
Stream read	High	Low
Batch write	High	Low
Batch read	Low	High

- MOR tables are used for real-time data ingestion.

  Real-time data ingestion usually requires performance within minutes or at a minute level. Considering the comparison between the two table models in Hudi, MOR tables are recommended for real-time data ingestion scenarios.

- Hudi table names and column names should be in lowercase.

  When multiple engines read and write the same Hudi table, using lowercase letters for table and column names can avoid inconsistencies in case sensitivity support between engines.

Recommendations

- For Spark batch processing scenarios with low requirements for write latency, use COW tables.

 Copyright © Huawei Technologies Co., Ltd.

In the COW model, there is a write amplification issue, resulting in slower write speed. However, COW tables have excellent read performance. Since batch processing is not very sensitive to write latency, COW tables can be used.

- Hive metadata synchronization must be enabled for Hudi table write tasks.

  SparkSQL is naturally integrated with Hive, so there is no need to consider metadata issues. This recommendation is for scenarios where Hudi tables are written through the Spark DataSource API or Flink. When writing to Hudi using these two methods, the configuration for synchronizing metadata with Hive needs to be added. The purpose of this configuration is to unify the metadata of Hudi tables in the Hive metadata service, providing convenience for cross-engine data operations and data management in the future.

3.2.2 Hudi Table Index Design Specifications

Rules

- Changing the index type of a table is prohibited.

  The index of a Hudi table determines the way data is stored. Changing the index type without caution can result in data duplication and accuracy issues between existing and new data in the table. Common index types include:

  - Bloom index: A unique index for Spark engine that uses the bloom filter mechanism and writes the bloom index content to the footer of Parquet files.
  - Bucket index: During the data write process, data is bucketed based on the primary key using hash calculation. This index has the fastest write speed but requires proper configuration of the number of buckets. Both Flink and Spark support this index.
  - State index: A unique index for Flink engine that records the storage location of row records in the state backend. During job cold start, it traverses all data storage files to generate index information.

- When using Flink state index, Spark cannot continue writing after Flink writes.

  Flink only generates log files when writing to Hudi's MOR table, and later converts the log files to Parquet files through compaction operations. Spark heavily relies on the existence of Parquet files when updating Hudi tables. Writing with Spark when the current Hudi table writes log files will result in duplicate data. During the batch initialization phase, use Spark for batch writing to Hudi tables first, and then use Flink with Flink state index for writing, as Flink will traverse all data files to generate the state index during cold start.

- In real-time data lake scenarios, Spark engine can use Bucket index, and Flink engine can use either Bucket index or State index.

  Real-time data import to the lake requires high performance within or within minutes. Index selection affects the performance of writing data to the Hudi table. The performance differences between indexes are:

  - Bucket index

    Advantages: During the write process, data is hashed and written into buckets based on the primary key, resulting in high performance. It is not limited by the amount of data in the table. Both Flink and Spark engines support this index, allowing cross-writing on the same table.

 Copyright © Huawei Technologies Co., Ltd.

Disadvantages: The number of buckets cannot be dynamically adjusted. Fluctuations in data volume and continuous increase in overall data volume can result in excessively large data files in individual buckets. Partitioned tables need to be used for improvement.

- Flink state index

  Advantages: The primary key's index information exists in the state backend. Data updates only require querying the state backend, which is quite fast. The size of the generated data files is stable and won't cause issues with small or oversized files.

  Disadvantages: This index is unique to Flink. When the total number of rows in the table reaches billions, it is necessary to optimize the state backend parameters to maintain write performance. Using this index does not support cross-writing between Flink and Spark.

- For tables with continuously increasing data volume, when using the Bucket index, time partitioning must be used, with the partition key based on the data creation time.

  According to the characteristics of the Flink state index, when the data volume of the Hudi table exceeds a certain amount, the pressure on the Flink job's state backend becomes significant, requiring optimization of state backend parameters to maintain performance. Also, since Flink needs to traverse the entire table data during a cold start, a large amount of data will cause the Flink job to start slowly. Therefore, from the perspective of simplifying usage, for tables with large amounts of data, the Bucket index can be used to avoid complex tuning of the state backend.

  If the combination of Bucket indexing and partitioned tables fails to resolve issues related to oversized buckets, you may still opt for Flink state indexing. In such cases, follow the guidelines to optimize the corresponding configuration parameters accordingly.

Recommendations

- For tables based on Flink's streaming write with data volumes exceeding 200 million records, use the bucket index. For volumes within 200 million, the Flink state index can be used.

  According to the characteristics of the Flink state index, when the data volume of the Hudi table exceeds a certain amount, the pressure on the Flink job's state backend becomes significant, requiring optimization of state backend parameters to maintain performance. Also, since Flink needs to traverse the entire table data during a cold start, a large amount of data will cause the Flink job to start slowly. Therefore, from the perspective of simplifying usage, for tables with large amounts of data, the Bucket index can be used to avoid complex tuning of the state backend.

  If the combination of Bucket indexing and partitioned tables fails to resolve issues related to oversized buckets, you may still opt for Flink state indexing. In such cases, follow the guidelines to optimize the corresponding configuration parameters accordingly.

- Design tables based on the bucket index with each bucket containing 2 GB of data.

  To avoid having a single bucket being too large, it is recommended that the data volume of a single bucket not exceed 2 GB (this 2 GB refers to the data

 Copyright © Huawei Technologies Co., Ltd.

content size, not the number of data rows or the size of the Parquet data file). The goal is to control the size of the corresponding bucket's Parquet files within 256 MB (to balance memory consumption and efficient use of HDFS storage). Thus, the 2 GB limit is just an empirical value, as the size of business data varies after columnar compression.

Why is 2 GB recommended?

- After storing 2 GB of data as columnar Parquet files, the data file size is approximately 150 MB to 256 MB. There may be variations for different service data. A single HDFS data block is generally 128 MB, which allows for efficient use of storage space.
- The memory space occupied by data read/write is the original data size (including null values), and 2 GB is within the acceptable range for a single task during big data computation.

What could be the impact if the data volume of a single bucket exceeds this range?

- Read/write tasks might encounter OOM issues, which can be resolved by increasing the memory allocation per task.
- Read/write performance may decrease because the data volume processed by a single task becomes larger, leading to longer processing times.

3.2.3 Hudi Table Partition Design Specifications

Rules

Partition keys cannot be updated:

Hudi has a primary key uniqueness mechanism, but in the context of partitioned tables, it usually can only guarantee uniqueness within a partition. Therefore, if the partition key value changes, it will result in multiple rows with the same primary key. In scenarios where date partitions are used, the creation time of the data can be used as the partition field. Remember not to use the data update time as the partition.

NOTE

When the index type of Hudi is specified as global, Hudi supports data updates across partitions, but global index performance is generally poor and is not recommended.

Recommendations

- Fact tables should use date partitions, while dimension tables should use non-partitioned or coarser-grained date partitions.

  Whether to use partitioned tables depends on the total data volume, increment, and usage. The characteristics of fact tables and dimension tables are:

  - Fact tables: large total data volume, large increment, data reads mostly sliced by date, and data is read for a certain time period.
  - Dimension tables: relatively small total volume, small increment, mostly update operations, data reads involve the whole table or are filtered by the corresponding service ID.

 Copyright © Huawei Technologies Co., Ltd.

Based on the above considerations, using daily partitions for dimension tables will result in too many files, and since the whole table is read, it will cause too many file read tasks. Using coarser-grained date partitions, such as yearly partitions, can effectively reduce the number of partitions and files. For dimension tables with small increments, non-partitioned tables can also be used. If the total data volume or increment of the dimension table is large, consider using a certain business ID for partitioning. In most data processing logic, large dimension tables will have certain business conditions for filtering to improve processing performance. This type of table needs to be optimized based on specific service scenarios and cannot be optimized solely by date partitioning. Fact table reads will be sliced by time periods, such as the last year, last month, or last day, so fact tables should prioritize date partitions.

- Use date fields for partitions, and the granularity of partitioned tables should be determined based on the data update range, neither too large nor too small.

  The granularity of partitions can be yearly, monthly, or daily. The goal of partition granularity is to reduce the number of file buckets written simultaneously, especially when there is a regular pattern of data updates over a certain time range. For example, if the highest proportion of data updates is within the last month, partitions can be created by month; if the highest proportion of data updates is within the last day, partitions can be created by day.

  Using bucket index, writes are scattered by hashing the primary key, and data is evenly written to each bucket under the partition. Since the data volume of each partition can fluctuate, the design of the number of buckets under each partition is usually calculated based on the maximum partition data volume. The finer the partition granularity, the more redundant the number of buckets. For example:

  Using daily partitions, with an average daily data increment of 3 GB and a maximum daily log of 8 GB, the table is created with Bucket count = 8 GB/2 GB = 4. If the highest proportion of data updates is daily and mainly distributed over the last month, this means that the data will be written into all buckets for the entire month, which is 4 x 30 = 120 buckets. If monthly partitions are used, the number of partition buckets = 3 GB x 30/2 GB = 45 buckets, thus reducing the number of data buckets to 45. With limited compute resources, the fewer buckets written, the higher the performance.

3.3 Hudi Data Table Management Operation Specifications

3.3.1 Hudi Data Table Compaction Specifications

MOR tables update data in the form of row logs, which need to be merged by primary key when read, making log read efficiency much lower than Parquet. To solve the log read performance problem, Hudi compresses logs into Parquet files through compaction, significantly improving read performance.

 Copyright © Huawei Technologies Co., Ltd.

Rules

- For tables with continuous data writing, perform compaction at least once every 24 hours.

  For MOR tables, whether streaming or batch writing, it is necessary to ensure that at least one compaction operation is completed daily. If compaction is not performed for a long time, the Hudi table's logs will grow larger, which will lead to the following problems:

  - Hudi table reads become very slow and require a lot of resources. This is because reading MOR tables involves log merging, which requires consuming many resources and is very slow with large logs.
  - Long-duration compaction requires a lot of resources and can easily lead to OOM.
  - Blocks cleaning. If compaction operations do not produce new versions of parquet files, old version files cannot be cleaned, increasing storage pressure.

- When submitting a Spark Jar job, the CPU to memory ratio should be 1:4 to 1:8.

  Compaction jobs merge data in existing parquet files with data in new logs, consuming high memory resources. According to previous table design specifications and actual traffic fluctuations, you are advised to configure the compaction job's CPU to memory ratio as 1:4 to 1:8 to ensure the stable operation of compaction jobs. If compaction encounters OOM issues, increasing the memory proportion can resolve them.

Recommendations

- Improve compaction performance by increasing concurrency.

  A reasonable CPU and memory ratio configuration ensures that the compaction job is stable, achieving stable operation of individual compaction tasks. However, the overall runtime of the compaction depends on the number of files processed in this compaction and the allocated CPU cores (concurrency). Therefore, increasing the number of CPU cores for the compaction job can improve compaction performance (note that increasing CPUs should also maintain the CPU to memory ratio).

- Use asynchronous compaction for Hudi tables.

  To ensure the stable operation of streaming ingestion jobs, it is necessary to ensure that streaming jobs do not perform other tasks during real-time ingestion, such as doing compaction while Flink writes to Hudi. This seems like a good solution as it completes ingestion and compaction. However, compaction operations are very memory and IO-intensive and will impact the streaming ingestion job as follows:

  - Increased end-to-end latency: Compaction amplifies write latency because it is more time-consuming than ingestion.
  - Unstable job: Compaction adds more instability to the ingestion job, and compaction OOM will directly cause the entire job to fail.

- Perform compaction every 2 to 4 hours.

  Compaction is a crucial and necessary maintenance method for MOR tables. For real-time tasks, the compaction merging process must be decoupled from real-time tasks. This is achieved by scheduling Spark tasks periodically for

 Copyright © Huawei Technologies Co., Ltd.

asynchronous compaction. The key to this solution is setting a reasonable period. If the period is too short, Spark tasks may run idle. If too long, many compaction plans may accumulate without being executed, leading to long Spark task durations and high downstream read task latency. Based on this scenario, here are some suggestions: according to cluster resource usage, schedule asynchronous compaction jobs every 2 or 4 hours, which is a basic maintenance plan for MOR tables.

- Perform compaction asynchronously using Spark instead of Flink.

  The recommended approach for Flink writing to Hudi is for Flink to handle data writing and compaction planning only. Submit Spark SQL or Spark Jar jobs asynchronously to perform compaction, clean, and archive tasks. The compaction plan generation is lightweight, with minimal impact on the Flink writing job.

  The specific steps for implementing this plan are as follows:

  - **Flink handles data writing and compaction planning only.**

    Add the following parameters to the Flink stream task's table creation statement/SQL hints to control Flink tasks writing to Hudi to only generate a compaction plan.

    ```
    'compaction.async.enabled' = 'false'       // Disable Flink executing compaction tasks.
    'compaction.schedule.enabled' = 'true' // Enable compaction plan generation.
    'compaction.delta_commits' = '5'           // MOR table defaults to attempt generating a compaction plan every 5 checkpoints; this parameter needs adjustment based on service requirements.
    'clean.async.enabled' = 'false'            // Disable clean operation.
    'hoodie.archive.automatic' = 'false'       // Disable archive operation.
    ```

  - **Perform compaction plans execution, clean, and archive tasks offline with Spark.**

    On the scheduling platform (for example, DataArts), run a scheduled offline task to let Spark complete the Hudi table's compaction plan execution, clean, and archive tasks.

    For SQL jobs, add the following configurations:

    ```
    hoodie.archive.automatic = false;
    hoodie.clean.automatic = false;
    hoodie.compact.inline = true;
    hoodie.run.compact.only.inline=true;
    hoodie.cleaner.commits.retained = 500;  // Clean retains the latest 500 deltacommit data files on the timeline; earlier versions will be cleaned. This value should be greater than the compaction.delta_commits setting and needs adjustment based on service requirements.
    hoodie.keep.max.commits = 700;  // The timeline retains a maximum of 700 delta commits.
    hoodie.keep.min.commits = 501;  // The timeline retains at least 500 delta commits. This value should be greater than hoodie.cleaner.commits.retained and needs adjustment based on service requirements.
    ```

    Then, keep the above configurations and **schedule** the following SQL **in order**:

    ```
    run compaction on <database name>. <table name>;   // Execute the compaction plan.
    run clean on <database name>. <table name>;        // Execute the clean operation.
    run archivelog on <database name>.<table name>;    // Execute the archive operation.
    ```

- Asynchronous compaction can serialize multiple tables into one job, grouping tables with similar resource configurations. The resource requirement for this job group is based on the table with the highest resource consumption.

  For asynchronous compaction tasks mentioned in **Use asynchronous compaction for Hudi tables.** and **Perform compaction asynchronously using Spark instead of Flink.**, here are some development suggestions:

 Copyright © Huawei Technologies Co., Ltd.

- You do not need to develop asynchronous compaction tasks for each Hudi table, as this increases development costs.
- Asynchronous compaction tasks can be completed by submitting Spark SQL jobs or handling compaction, clean, and archive for multiple tables in Spark jar tasks:

```
hoodie.clean.async = true;
hoodie.clean.automatic = false;
hoodie.compact.inline = true;
hoodie.run.compact.only.inline=true;
hoodie.cleaner.commits.retained = 500;
hoodie.keep.min.commits = 501;
hoodie.keep.max.commits = 700;
```

Schedule the following SQL **in order**:

```
run compaction on <database name>. <table1>;
run clean on <database name>. <table1>;
run archivelog on <database name>.<table1>;
run compaction on <database name>.<table2>;
run clean on <database name>.<table2>;
run archivelog on <database name>.<table2>;
```

3.3.2 Hudi Data Table Clean Specifications

Clean is also one of the maintenance operations for Hudi tables and needs to be executed for both MOR and COW tables. The purpose of the Clean operation is to remove old version files (data files no longer used by Hudi), which not only saves time during the Hudi table list process but also alleviates storage pressure.

Rules

Clean operations must be performed for Hudi tables.

For Hudi's MOR and COW tables, clean must be enabled.

- When writing data to Hudi tables, the system will automatically determine whether to execute clean, as the clean function is enabled by default (**hoodie.clean.automatic** is **true**).
- The clean operation is not triggered every time data is written; at least two conditions must be met:
  a. There must be old version table files in Hudi. For COW tables, as long as the data has been updated, old version files will exist. For MOR tables, the data must have been updated and compaction must have been performed to have old version files.
  b. The Hudi table meets the threshold specified by **hoodie.cleaner.commits.retained**. If writing to Hudi with Flink, at least the submitted checkpoint must exceed this threshold; if writing to Hudi in batches, the number of batches must exceed this threshold.

Recommendations

- For MOR tables downstream using batch read mode, the number of clean versions should be compaction versions plus 1.

  MOR tables must ensure the compaction plan is successfully executed. The compaction plan only records which log files and which Parquet files in the Hudi table should be merged, so the most important point is to ensure that the files to be merged exist when executing the compaction plan. Since only

 Copyright © Huawei Technologies Co., Ltd.

the clean operation can clean files in the Hudi table, it is recommended that the clean trigger threshold (**hoodie.cleaner.commits.retained** value) should be greater than the compaction trigger threshold (**compaction.delta_commits** value for Flink tasks).

- For MOR tables with downstream stream computing, retain historical versions at the hourly level.

  If the downstream of the MOR table is stream computing, such as Flink streaming reads, retain hourly historical versions as needed. This way, incremental data within the last few hours can be read through log files. If the retention period is too short, in case of downstream Flink job restarts or interruptions, the upstream incremental data might already be Cleaned, and Flink will need to read incremental data from Parquet files, which will reduce performance. If the retention period is too long, it will lead to redundant storage of historical data in logs.

  To retain 2 hours of historical version data, use the following formula:

  Number of versions = 3600 x 2/Version interval time, where the version interval time comes from the Flink job's checkpoint cycle or the upstream batch write cycle.

- For COW tables without special requirements for retaining historical version data, set the number of retained versions to **1**.

  Each version of the COW table contains the entire table data, so retaining multiple versions leads to redundancy. Therefore, if there is no need for historical data retrieval, set the number of retained versions to **1**, retaining only the latest version.

- Execute clean jobs at least once a day, or every 2 to 4 hours.

  Both Hudi's MOR and COW tables need to ensure at least one clean operation daily. The clean operation for MOR tables can be executed asynchronously with compaction, as referenced in section 2.2.1.6. For COW tables, clean can be automatically determined during data writing.

3.3.3 Hudi Data Table Archive Specifications

Archive is intended to alleviate the metadata read and write pressure on Hudi. All metadata is stored in this path: *the root directory of the Hudi table***/.***hoodie directory*. If the number of files in the .hoodie directory exceeds 10,000, there will be very noticeable read and write delays in the Hudi table.

Rules

Archive operations must be performed for Hudi tables.

For Hudi's MOR and COW tables, archive must be enabled.

- When writing data to Hudi tables, the system will automatically determine whether to execute archive, as the archive function is enabled by default (**hoodie.archive.automatic** is **true**).
- The archive operation is not triggered every time data is written; at least two conditions must be met:
  - The Hudi table meets the threshold specified by **hoodie.keep.max.commits**. If writing to Hudi with Flink, at least the submitted checkpoint must exceed this threshold; if writing to Hudi with

 Copyright © Huawei Technologies Co., Ltd.

Spark, the number of times data is written to Hudi must exceed this threshold.

- Hudi tables must have undergone clean; if clean has not been performed, archive will not be executed.

Recommendations

Execute archive jobs at least once a day, or every 2 to 4 hours.

Both Hudi's MOR and COW tables need to ensure at least one archive operation daily. The archive operation for MOR tables can be executed asynchronously with compaction, as referenced in section 2.2.1.6. For COW tables, archive can be automatically determined during data writing.

3.4 Spark on Hudi Development Specifications

3.4.1 Parameter Specifications for Creating a Hudi Table with SparkSQL

Rules

- When creating a table, you must specify the **primaryKey** and **preCombineField**.

  Hudi tables provide the capability for data updates and idempotent writes, which require data records to have a primary key to identify duplicate data and perform update operations. Not specifying a primary key will result in the loss of data update capabilities for the table, and not specifying **preCombineField** will lead to primary key duplication.

Parameter	Description	Value	Remarks
primaryKey	Primary key of the Hudi table	As needed	(Mandatory) It can be a composite primary key but must be globally unique.
preCombineField	Pre-merge key. Multiple data records with the same primary key are merged based on this field.	As needed	(Mandatory) Data with the same primary key is merged based on this field. Multiple fields cannot be specified.

- Do not set **hoodie.datasource.hive_sync.enable** to **false** when creating a table.

  Setting it to **false** will prevent new partitions from being synchronized to the Hive Metastore. Missing new partition information will result in data loss when the query engine reads it.

 Copyright © Huawei Technologies Co., Ltd.

- Do not set the Hudi index type to **INMEMORY**.

  This index is for testing purposes only. Using this index in a production environment will lead to data duplication.

Example of Creating a Table

```
create table data_partition(id int, comb int, col0 int,yy int, mm int, dd int)
using hudi                          --Specify a Hudi data source.
partitioned by(yy,mm,dd)                --Specify one or multiple partitions.
location 'obs://bucket/path/data_partition'       --Specify a path. When using the metadata service provided by DLI, only OBS tables can be created.
options(
type='mor',                         --Table type: mor or cow
primaryKey='id',                     -- Primary key, which can be a composite one but must be globally unique.
preCombineField='comb'                  --Pre-merge field. Data with the same primary key is merged based on this field. Currently, multiple fields cannot be specified.
)
```

3.4.2 Parameter Specifications for Incremental Reading of Hudi Table Data with Spark

Rules

Before performing an incremental query, you must set the current table's query mode to incremental query and reset the table's query mode after the query is completed.

If the table's query mode is not reset after the incremental query, subsequent real-time queries will be affected.

Example

The following uses a SQL job as an example:

Set parameters.

```
hoodie.tableName.consume.mode=INCREMENTAL // Set the current table to be read in incremental mode.
hoodie.tableName.consume.start.timestamp=20201227153030 // Specify the initial incremental pull commit.
hoodie.tableName.consume.end.timestamp=20210308212318 // Specify the incremental pull end commit. If not specified, the latest commit is used.
```

Run the following SQL statement:

```
select * from tableName where `_hoodie_commit_time`>'20201227153030' and `_hoodie_commit_time`<='20210308212318'; // The results must be filtered based on start.timestamp and end.timestamp. If end.timestamp is not specified, then filtering should only be done based on start.timestamp.
```

When submitting other SQL statements, you need to clear the preceding configuration parameters to avoid affecting the execution results of other tasks.

3.4.3 Parameter Specifications for Spark Asynchronous Task Execution Table Compaction

- Do not manually execute the **run schedule** command to generate a compaction plan when the writing job has not stopped.

  Incorrect example:

  ```
  run schedule on dsrTable
  ```

 Copyright © Huawei Technologies Co., Ltd.

If there are other tasks writing to this table, executing this operation will result in data loss.

- When executing the **run compaction** command, do not set **hoodie.run.compact.only.inline** to **false**. Instead, set it to **true**.

  Incorrect example:

  Set parameters.

  ```
  hoodie.run.compact.only.inline=false
  ```

  Run the following SQL statement:

  ```
  run compaction on dsrTable;
  ```

  If there are other tasks writing to this table, executing these operations will result in data loss.

  Correct example: asynchronous compaction

  ```
  hoodie.compact.inline = true
  hoodie.run.compact.only.inline=true
  ```

  Run the following SQL statement:

  ```
  run compaction on dsrTable;
  ```

3.4.4 Spark Table Data Maintenance Specifications

Do not modify key table attribute information using the alter command: **type/primaryKey/preCombineField/hoodie.index.type**.

Error example: executing the following statements to modify key table attributes:

```
alter table dsrTable set tblproperties('type'='xx');
alter table dsrTable set tblproperties('primaryKey'='xx');
alter table dsrTable set tblproperties('preCombineField'='xx');
alter table dsrTable set tblproperties('hoodie.index.type'='xx');
```

Other engines aside from Spark can also modify Hudi table metadata, but such modifications can lead to data duplication or even data corruption in the entire Hudi table; therefore, modification of the above attributes is prohibited.

3.5 Bucket Tuning

3.5.1 Tuning a Bucket Index Table

Tuning a Bucket Index Table

Common parameters for bucket indexes:

- Spark:
  ```
  hoodie.index.type=BUCKET
  hoodie.bucket.index.num.buckets=5
  ```
- Flink:
  ```
  index.type=BUCKET
  hoodie.bucket.index.num.buckets=5
  ```

Determining Whether to Use Partitioned or Non-Partitioned Tables

There are two types of tables based on usage scenarios: fact tables and dimension tables.

- Fact tables typically have a larger amount of data, primarily new data, with a small proportion of updated data. Most of the updated data falls within a recent time range (year, month, or day). When a downstream system reads the table for ETL calculations, it typically uses a time range for clipping (e.g., last day, month, year). Such tables can usually be partitioned by the creation time of the data to ensure optimal read/write performance.
- Dimension tables generally have a smaller amount of data, primarily updated data, with fewer new entries. The table data size is relatively stable, and full reads are usually required for join-like ETL calculations. Therefore, using non-partitioned tables usually results in better performance.
- **The partition key for partitioned tables is not allowed to update; otherwise, it will create duplicate data.**

Exceptional scenarios: super large dimension tables and very small fact tables

Special cases such as **dimension tables with continuous large amounts of new data** (table data size over 200 GB or daily growth exceeding 60 MB) or very small fact tables (table data size less than 10 GB and will not exceed 10 GB even after 3 to 5 years of growth) need to be handled on a case-by-case basis:

- Dimension tables with continuous large amounts of new data
  - Method 1: Reserve bucket numbers. If using a non-partitioned table, pre-increase the bucket numbers by estimating the data increment over a long period. **The disadvantage is that files will continue to swell as data grows.**
  - Method 2: Large-granularity partitions (recommended). If using a partitioned table, calculate the growth of the data to determine the partitioning, for example, using yearly partitions. **This method is relatively cumbersome but avoids re-importing the table after many years.**
  - Method 3: Data aging. Analyze the business logic to see if the large dimension table can reduce data scale by aging out and cleaning up invalid dimension data.
- Very small fact tables

  Using non-partitioned tables with a slightly more generous number of buckets can improve read/write performance, based on the premise of estimating long-term data growth.

Determining the Number of Buckets in a Table

Setting the number of buckets for Hudi tables is crucial for performance and requires special attention.

The following are key points to confirm before creating a table:

- Non-partitioned tables
  - Total number of records in a single table = **select count(1) from tablename** (provided during ingestion).
  - Size of a single record = average 1 KB (suggest using **select * from tablename limit 100**, calculate the size of 100 records, and divide by 100 for the average size).

 Copyright © Huawei Technologies Co., Ltd.

  - Size of single table data (GB) = Total number of records x Size of a single record/1024/1024.
  - Number of buckets for non-partitioned tables = MAX(Total size of single table data (GB)/2 GB x 2, round up to nearest even number, 4).
- Partitioned tables
  - **Total number of records** in the partition with the largest data volume in the last month = Consult the product line before ingestion.
  - Size of a single record = average 1 KB (suggest using **select * from tablename limit 100**, calculate the size of 100 records, and divide by 100 for the average size).
  - Size of single partition data (GB) = Total number of records in the largest data volume partition in the last month x Size of a single record/1024/1024.
  - Number of buckets for partitioned tables = MAX(Total size of single partition data (GB)/2 GB, round up to nearest even number, 1).

CAUTION

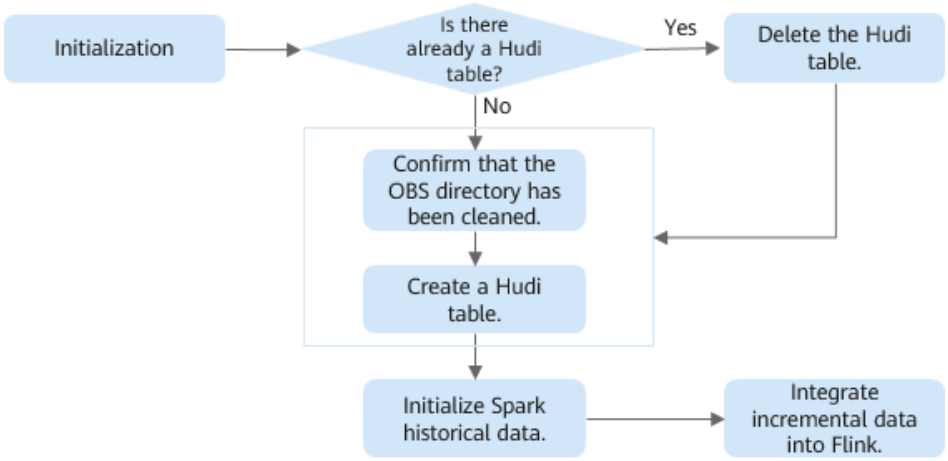
- Use the total data size of the table, not the compressed file size.
- It is best to set an even number of buckets, with a minimum of 4 for non-partitioned tables and a minimum of 1 for partitioned tables.

3.5.2 Hudi Table Initialization

- The initial import of existing data is usually done by Spark jobs. Since the initial data volume is typically large, you are advised to use APIs to allocate sufficient resources.
- In scenarios where real-time writing by Flink or Spark streaming jobs is required after batch initialization, you are advised to control the amount of duplicated data access by filtering messages from a specified time range (for example, after Spark initialization is completed, Flink filters out data from 2 hours ago when consuming from Kafka). If filtering Kafka messages is not possible, you can consider ingesting data in real-time to generate an offset, truncating the table, then performing a historical import, and finally starting real-time ingestion.

 Copyright © Huawei Technologies Co., Ltd.

Figure 3-1 Initialization flowchart



NOTE

- If data already exists in the table before batch initialization without truncating the table, it will result in very large log files, putting significant pressure on subsequent compaction and requiring more resources to complete.
- In the Hive metadata, a Hudi table should have one internal table (manually created) and two external tables (automatically created after writing data).
- The two external tables, **_ro** (user-read-only merged Parquet file, i.e., read-optimized view table) and **_rt** (read latest version of real-time written data, i.e., real-time view table).

3.5.3 Real-Time Job Ingestion

Real-time jobs are generally completed using Flink SQL or Spark Streaming. Stream-based real-time jobs are typically configured to synchronously generate compaction plans and asynchronously execute the plans.

- Sink Hudi table configuration in Flink SQL jobs:

```
create table hudi_sink_table (
  // table columns...
) PARTITIONED BY (
  years,
  months,
  days
) with (
  'connector' = 'hudi',                                          // Specify the Hudi table to be written.
  'path' = 'obs://bucket/path/hudi_sink_table',                  // Specify the storage path for the
Hudi table.
  'table.type' = 'MERGE_ON_READ',                                // Hudi table type
  'hoodie.datasource.write.recordkey.field' = 'id',              // Primary key
  'write.precombine.field' = 'vin',                              // Field for pre-combining
  'write.tasks' = '10',                                          // Flink write parallelism
  'hoodie.datasource.write.keygenerator.type' = 'COMPLEX',       // Specify KeyGenerator,
consistent with the type of the Hudi table created by Spark.
  'hoodie.datasource.write.hive_style_partitioning' = 'true',    // Use a Hive-compatible
partitioning format.
  'read.streaming.enabled' = 'true',                             // Enable stream read.
  'read.streaming.check-interval' = '60',                        // Checkpoint interval, in seconds.
  'index.type'='BUCKET',                                         // Specify the Hudi table index type as
BUCKET.
  'hoodie.bucket.index.num.buckets'='10',                        // Specify the number of buckets.
  'compaction.delta_commits' = '3',                              // Interval for compaction commits
  'compaction.async.enabled' = 'false',                          // Disable asynchronous compaction.
```

 Copyright © Huawei Technologies Co., Ltd.

```
 'compaction.schedule.enabled' = 'true',                          // Enable synchronous compaction
scheduling.
 'clean.async.enabled' = 'false',                                 // Disable asynchronous cleaning.
 'hoodie.archive.automatic' = 'false',                            // Disable automatic archiving.
 'hoodie.clean.automatic' = 'false',                              // Disable automatic cleaning.
 'hive_sync.enable' = 'true',                                     // Enable automatic metadata
synchronization.
 'hive_sync.mode' = 'hms',                                        // Use HMS for metadata synchronization.
 'hive_sync.db' = 'default',                                      // Database for metadata synchronization
 'hive_sync.table' = 'hudi_sink_table',                           // Table name for metadata
synchronization
 'hive_sync.support_timestamp' = 'true',                          // Support timestamp format for Hive
table synchronization
 'hive_sync.partition_extractor_class' = 'org.apache.hudi.hive.MultiPartKeysValueExtractor'          //
Extractor class for Hive table synchronization
);
```

- Common parameters for writing Hudi tables with Spark Streaming (The meanings of the parameters are similar to those in Flink, so they will not be annotated again):

```
hoodie.table.name=
hoodie.index.type=BUCKET
hoodie.bucket.index.num.buckets=3
hoodie.datasource.write.precombine.field=
hoodie.datasource.write.recordkey.field=
hoodie.datasource.write.partitionpath.field=
hoodie.datasource.write.table.type= MERGE_ON_READ
hoodie.datasource.write.hive_style_partitioning=true
hoodie.compact.inline=true
hoodie.schedule.compact.only.inline=true
hoodie.run.compact.only.inline=false
hoodie.clean.automatic=false
hoodie.clean.async=false
hoodie.archive.async=false
hoodie.archive.automatic=false
hoodie.compact.inline.max.delta.commits=50
hoodie.datasource.hive_sync.enable=true
hoodie.datasource.hive_sync.partition_fields=
hoodie.datasource.hive_sync.database=
hoodie.datasource.hive_sync.table=
hoodie.datasource.hive_sync.partition_extractor_class=org.apache.hudi.hive.MultiPartKeysValueExtracto
r
```

3.5.4 Offline Compaction Configuration

For real-time operations of MOR tables, it is common to synchronously generate compaction plans during writing. Therefore, additional scheduling using DataArts Studio or scripts to execute the already generated compaction plans with SparkSQL is needed.

- Execution parameters

```
set hoodie.compact.inline = true;               // Enable compaction.
set hoodie.run.compact.only.inline = true;      // Compaction only executes the generated plans
without creating new plans.
set hoodie.cleaner.commits.retained = 120;      // Clear commits and retain 120 commits.
set hoodie.keep.max.commits = 140;              // Maximum of 140 commits to be archived
set hoodie.keep.min.commits = 121;              // Minimum of 121 commits to be archived
set hoodie.clean.async = false;                 // Disable asynchronous cleaning.
set hoodie.clean.automatic = false;             // Disable automatic cleaning to prevent compaction
operations from triggering clean.

run compaction on $tablename;                   // Execute the compaction plan.
run clean on $tablename;                        // Execute the clean operation to remove redundant
versions.
run archivelog on $tablename;                   // Execute archivelog to merge and clean metadata files.
```

 Copyright © Huawei Technologies Co., Ltd.

CAUTION

- The values for cleanup and archival parameters should not be set too high, as it can affect the performance of Hudi tables. It is generally recommended:
  - hoodie.cleaner.commits.retained = 2 x Number of commits required for compaction
  - hoodie.keep.min.commits = hoodie.cleaner.commits.retained + 1
  - hoodie.keep.max.commits = hoodie.keep.min.commits + 20
- Execute clean and archive after compaction. Since clean and archivelog require fewer resources, to avoid resource waste, you can use DataArts Studio to schedule compaction as one task, and clean and archive as another task with different resources.

- Execution resources
  - The interval for scheduling compaction should be less than the interval for generating compaction plans. For example, if a compaction plan is generated approximately every hour, the scheduling task for executing the compaction plan should be scheduled at least once every half hour.
  - The resources configured for compaction jobs should have a number of vCPUs at least equal to the number of buckets in a single partition. The ratio of vCPUs to memory should be 1:4, that is, 1 vCPU with 4 GB of memory.

 Copyright © Huawei Technologies Co., Ltd.

4 Using Hudi to Develop Jobs in DLI

4.1 Submitting a Spark SQL Job in DLI Using Hudi

Log in to the DLI management console. In the navigation pane on the left, choose **SQL Editor**. When submitting a SQL job, select a Spark SQL queue that supports Hudi.

Step 1 Create a Hudi table.

Paste the following table creation statements to the edit area of the DLI SQL editor, replace the **LOCATION** value with the actual path, set **Engine** to **Spark**, configure **Queues**, **Catalog**, and **Databases**, and click **Execute** in the upper right corner to submit the job.

Note: Hudi internal tables cannot be created when using DLI as the metadata service. So, you must set **LOCATION** to an OBS path.

```
CREATE TABLE
  hudi_table (id int, comb long, name string, dt date) USING hudi PARTITIONED BY (dt) OPTIONS (
    type = 'cow',
    primaryKey = 'id',
    preCombineField = 'comb'
  ) LOCATION 'obs://bucket/path/hudi_table';
```

Wait for the execution history below to show that the job is successfully executed, which means that the table is created. At this point, a COW partition table of Hudi is created.

You can run **SHOW TABLES** to check if the table is successfully created.

```
SHOW TABLES;
```

Step 2 Run the following SQL statement to write data to the created Hudi table:

```
INSERT INTO hudi_table VALUES (1, 100, 'aaa', '2021-08-28'), (2, 200, 'bbb', '2021-08-28');
```

To check the execution result, navigate to the **Executed Queries (Last Day)** tab at the bottom of the editor. Alternatively, you can choose **Job Management** > **SQL Jobs** on the left navigation pane to view the status of the SQL job.

Step 3 Set Hudi parameters when running SQL statements.

NOTE

DLI does not support setting parameters by running SET statements.

 Copyright © Huawei Technologies Co., Ltd.

Click **Settings**. In the **Parameter Settings** area, set the key and value. The parameters configured here will take effect when a SQL job is submitted.

In the navigation pane on the left, choose **Job Management** > **SQL Jobs**. Locate the job that is being executed and expand its details. In **Parameter Settings**, check its parameter settings.

Step 4 Run the following SQL statement to query the written content:

```
select id,comb,name,dt from hudi_table where dt='2021-08-28';
```

You can view the query result in the lower pane of the editor.

Step 5 Delete the Hudi table you created.

If a foreign table is created, only the metadata of the Hudi table is deleted when the SQL statement is executed to delete the table, and the data still exists in the OBS bucket and needs to be manually deleted.

```
DROP TABLE IF EXISTS hudi_table;
```

----End

4.2 Submitting a Spark Jar Job in DLI Using Hudi

To submit a Spark Jar job, you need to manually configure the Hudi lock provider when using LakeFormation as the metadata service. Refer to **Hudi Lock Configuration**.

Step 1 Log in to the DLI management console and choose **Job Management** > **Spark Jobs**.

To submit a Spark Jar job related to Hudi, select Spark 3.3.1 and make sure that the general-purpose queue supports Hudi.

Step 2 Click **Create Job** in the upper right corner.

Step 3 Write and package the Spark JAR file (using a Maven project as an example).

Create or use an existing Maven Java project, and introduce dependencies for Scala 2.12, Spark 3.3.1, and Hudi 0.11.0 in the **pom.xml** file. Since the DLI environment already provides the required dependencies, you can set the **scope** to **provided**.

```
<dependencies>
 <dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala-library</artifactId>
  <version>2.12.15</version>
 </dependency>
 <dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.12</artifactId>
  <version>3.3.1</version>
  <scope>provided</scope>
 </dependency>
 <dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.12</artifactId>
  <version>3.3.1</version>
  <scope>provided</scope>
 </dependency>
 <dependency>
```

 Copyright © Huawei Technologies Co., Ltd.

```
    <groupId>org.apache.hudi</groupId>
    <artifactId>hudi-spark3-bundle_2.12</artifactId>
    <version>0.11.0</version>
    <scope>provided</scope>
  </dependency>
  <!-- ... -->
</dependencies>
```

Configure scala-maven-plugin for compilation and packaging.

```
<build>
  <plugins>
    <plugin>
      <groupId>net.alchim31.maven</groupId>
      <artifactId>scala-maven-plugin</artifactId>
      <version>3.3.1</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>testCompile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!-- ... -->
  </plugins>
  <!-- ... -->
</build>
```

Then, create a Scala directory under the main directory, and create a package within it. Inside the package directory, create a Scala file and write the following:

```
import org.apache.spark.sql.catalyst.expressions.GenericRowWithSchema
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
import org.apache.spark.sql.types.{DataTypes, StructField, StructType}

import java.util.{ArrayList, List => JList}

object HudiScalaDemo {
  def main(args: Array[String]): Unit = {
    // Step 1: Obtain or create a SparkSession instance.
    val spark = SparkSession.builder
      .enableHiveSupport
      .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
      .config("spark.sql.extensions", "org.apache.spark.sql.hudi.HoodieSparkSessionExtension")
      .appName("spark_jar_hudi_demo")
      .getOrCreate

    // Step 2: Construct DataFrame data for writing.
    val schema = StructType(Array(
      StructField("id", DataTypes.IntegerType),
      StructField("name", DataTypes.StringType),
      StructField("update_time", DataTypes.StringType),
      StructField("create_time", DataTypes.StringType)
    ))
    val data: JList[Row] = new ArrayList[Row]()
    data.add(new GenericRowWithSchema(Array(1, "Alice", "2024-08-05 09:00:00", "2024-08-01"), schema))
    data.add(new GenericRowWithSchema(Array(2, "Bob", "2024-08-05 09:00:00", "2024-08-02"), schema))
    data.add(new GenericRowWithSchema(Array(3, "Charlie", "2024-08-05 09:00:00", "2024-08-03"),
schema))
    val df = spark.createDataFrame(data, schema)

    // Step 3: Configure the table name and OBS path.
    val databaseName = "default"
    val tableName = "hudi_table"
    val basePath = "obs://bucket/path/hudi_table"

    // Step 4: Write data and synchronize the metadata service provided by DLI to create a table.
    df.write.format("hudi")
```

 Copyright © Huawei Technologies Co., Ltd.

```
      .option("hoodie.table.name", tableName)
      .option("hoodie.datasource.write.table.type", "COPY_ON_WRITE")
      .option("hoodie.datasource.write.recordkey.field", schema.fields(0).name)  // Primary key, which is
mandatory.
      .option("hoodie.datasource.write.precombine.field", schema.fields(2).name)  // Pre-aggregation key,
which is mandatory. If not needed, configure the same column as the primary key.
      .option("hoodie.datasource.write.partitionpath.field", schema.fields(3).name)  // Partition column.
Multiple partitions can be configured and separated by commas (,).
      .option("hoodie.datasource.write.keygenerator.class", "org.apache.hudi.keygen.ComplexKeyGenerator")
      // Enable synchronization.
      .option("hoodie.datasource.hive_sync.enable", "true")
      .option("hoodie.datasource.hive_sync.partition_fields", schema.fields(3).name)
// Set this parameter based on the actual partition field. For a non-partitioned table, select
org.apache.hudi.hive.NonPartitionedExtractor.
      .option("hoodie.datasource.hive_sync.partition_extractor_class",
"org.apache.hudi.hive.MultiPartKeysValueExtractor")
      .option("hoodie.datasource.hive_sync.use_jdbc", "false")
      .option("hoodie.datasource.hive_sync.table", tableName)
      .option("hoodie.datasource.hive_sync.database", databaseName)
      // Select a save mode as needed.
      .mode(SaveMode.Overwrite)
      .save(basePath)

    // Step 5: Run the following SQL statement to query the table:
    spark.sql(s"select id,name,update_time,create_time from ${databaseName}.${tableName} where
create_time='2024-08-01'")
      .show(100)
  }
}
```

Run the Maven packaging command to obtain the JAR file from the target directory and upload it to the OBS directory.

```
mvn clean install
```

Step 4 Submit the Spark Jar job.

Log in to the DLI management console. In the navigation pane on the left, choose **Job Management** > **Spark Jobs**. On the displayed page, click **Create Job** in the upper right corner.

- Select a queue for **Queues** and set **Spark Version** to **3.3.1** or later.
- You can configure the job name to facilitate identification and filtering.
- Set **Application**. The path points to the Spark JAR file uploaded to OBS in the previous step.
- Configure an agency. Select the agency required for submitting DLI jobs.
- Set **Main Class(--class) (Optional)** to the full name of the class that contains the main function to be executed.
- You must configure Hudi lock parameters in **Spark Arguments(--conf)** and add the prefix **spark.hadoop.**, for example:

  ```
  spark.hadoop.hoodie.write.lock.provider=com.huawei.luxor.hudi.util.DliCatalogBasedLockProvider
  ```

  For additional parameter details, refer to **Hudi Lock Configuration**.

 Copyright © Huawei Technologies Co., Ltd.

> **CAUTION**
>
> - Disabling Hudi locks or using alternative lock implementation classes may lead to data loss or anomalies.
> - Under no circumstances will DLI be held responsible for any direct or indirect losses or damages resulting from disabling Hudi locks or using lock implementation classes incompatible with metadata services. This includes, but is not limited to, lost business profits, service disruptions, data loss, or other financial damages.

- Set **Access Metadata** to **Yes**. You are advised to use the metadata service to manage Hudi tables. The configuration in the previous step contains the configuration item for synchronizing metadata.

Click **Execute** in the upper right corner to submit the job.

Step 5 Execute the job and check the logs. (Note: Log archiving may take some time. Logs are typically archived within 1 to 5 minutes after the job execution.)

After you click **Execute**, the **Spark Jobs** page is displayed, where you can view the job execution status. Click **More** in the **Operation** column of the job and select an operation.

- **View Log**: Redirects to the OBS page where you can see the complete log archive addresses of the job, including commit logs, driver logs, and executor logs. You can download the logs here.
- **Commit Logs**: Redirects to the aggregated commit log display page where you can view log information during job submission.
- **Driver Logs**: Redirects to the aggregated display page for driver logs, sequentially displaying **spark.log**, **stderr.log**, and **stdout.log** from top to bottom.

Then select **Driver Logs**. If the logs are not yet aggregated, wait a few minutes and check again. You can see the result of the **select** statement printed by the sample program in **stdout.log** at the bottom of the logs.

----End

4.3 Submitting a Flink SQL Job in DLI Using Hudi

This section describes how to submit a Flink SQL job in DLI using Hudi.

For details about the syntax, see **Flink OpenSource SQL 1.15 Syntax Overview**.

Log in to the DLI management console. In the navigation pane on the left, choose **Job Management** > **Flink Jobs**.

Step 1 Create a Flink job. Specifically, click **Create Job** in the upper right corner. In the dialog box that appears, set **Type** to **Flink OpenSource SQL** and enter a name.

Step 2 Write Flink SQL statements (without using catalogs).

Here, the sink table points to the Hudi table path by creating a temporary table to write data, and the table parameters are configured with **hive_sync** related

 Copyright © Huawei Technologies Co., Ltd.

parameters to synchronize metadata to the metadata service provided by DLI in real-time.

For details about the parameters, see **Hudi Result Table**.

Modify the path parameter of the sink table in the job to the desired OBS path for saving the Hudi table.

```
-- Temporary table as the source, using datagen to mock data
create table
  orderSource (
    order_id STRING,
    order_name STRING,
    order_time TIMESTAMP(3)
  )
with
  ('connector' = 'datagen', 'rows-per-second' = '1');

-- Create a Hudi temporary table as the sink, configuring hms to synchronize the table to DLI metadata service.
CREATE TABLE
  hudi_table (
    order_id STRING PRIMARY KEY NOT ENFORCED,
    order_name STRING,
    order_time TIMESTAMP(3),
    order_date String
  ) PARTITIONED BY (order_date)
WITH
  (
    'connector' = 'hudi',
    'path' = 'obs://bucket/path/hudi_table',
    'table.type' = 'MERGE_ON_READ',
    'hoodie.datasource.write.recordkey.field' = 'order_id',
    'write.precombine.field' = 'order_time',
    'hive_sync.enable' = 'true',
    'hive_sync.mode' = 'hms',
    'hive_sync.table' = 'hudi_table',
    'hive_sync.db' = 'default'
  );

-- Execute the insert statement to read from the source table and write to the sink.
insert into
  hudi_table
select
  order_id,
  order_name,
  order_time,
  DATE_FORMAT (order_time, 'yyyyMMdd')
from
  orderSource;
```

Step 3 Set job running parameters.

- Select a queue and configure the Flink version to at least 1.15.
- Configure an agency with sufficient permissions.
- Configure an OBS bucket.
- Enable checkpointing. Checkpointing must be enabled when using Hudi.

Step 4 Submit the job and check the FlinkUI and logs.

Click **Submit** in the upper right corner of the page. Confirm the parameters are correct on the displayed page, then click **Start** at the bottom. After submission, the Flink job page is displayed, where you can filter the recently submitted Flink job and check its execution status.

Click the job name to navigate to the job page. Here you can click the **Commit Logs** or **Run Log** tab to check the aggregated logs. You can also directly click the

 Copyright © Huawei Technologies Co., Ltd.

Logs tab, select **JobManager** or **TaskManager**, and download the corresponding logs.

Click **More** in the upper right corner of the page and select **FlinkUI** to navigate to the FlinkUI for the job.

----End

 Copyright © Huawei Technologies Co., Ltd.

5 DLI Hudi SQL Syntax Reference

5.1 Hudi DDL Syntax

5.1.1 CREATE TABLE

Function

This command is used to create a Hudi table by specifying the list of fields along with the table options. When using the metadata service provided by DLI, only foreign tables can be created, meaning you need to specify the table path through **LOCATION**.

Syntax

CREATE TABLE *[IF NOT EXISTS] [database_name.]table_name*

[(columnTypeList)]

USING hudi

[COMMENT table_comment]

[LOCATION location_path]

[OPTIONS (options_list)]

Parameter Description

Table 5-1 Parameter descriptions

Parameter	Description
database_name	Database name that contains letters, digits, and underscores (_).

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
table_name	Database table name that contains letters, digits, and underscores (_).
columnTypeList	List of comma-separated columns with data types. The column name contains letters, digits, and underscores (_).
using	Uses **hudi** to define and create a Hudi table.
table_comment	Description of the table.
location_path	OBS path. If specified, the Hudi table will be created as a foreign table.
options_list	List of Hudi table options.

Table 5-2 Table options

Parameter	Description
primaryKey	Mandatory. Primary key name. Separate multiple primary key names with commas (,).
type	Type of the table. **'cow'** indicates a copy-on-write (COW) table, and **'mor'** indicates a merge-on-read (MOR) table. If this parameter is not specified, the default value is **'cow'**.
preCombineField	(Mandatory) Table's preCombine field. When pre-aggregating data before writing, if the primary keys are the same, the preCombine field will be used for comparison.
payloadClass	Logic that uses **preCombineField** for data filtering. **DefaultHoodieRecordPayload** is used by default. In addition, multiple preset payloads are provided, such as **OverwriteNonDefaultsWithLatestAvroPayload**, **OverwriteWithLatestAvroPayload**, and **EmptyHoodieRecordPayload**.
useCache	Whether to cache table relationships in Spark. This parameter does not need to be configured. This parameter is set to **false** by default to support the incremental view query of the COW table in Spark SQL.

Example

- **Create a non-partitioned table.**

```
create table if not exists hudi_table0 (
id int,
name string,
price double
```

```
) using hudi
options (
type = 'cow',
primaryKey = 'id',
preCombineField = 'price'
);
```

- **Create a partitioned table.**

```
create table if not exists hudi_table_p0 (
id bigint,
name string,
ts bigint,
dt string,
hh string
) using hudi
options (
type = 'cow',
primaryKey = 'id',
preCombineField = 'ts'
)
partitioned by (dt, hh);
```

- **Create a table in a specified path.**

```
create table if not exists h3(
id bigint,
name string,
price double
) using hudi

options (
primaryKey = 'id',
preCombineField = 'price'
)
location 'obs://bucket/path/to/hudi/h3';
```

- **Specify table properties when creating a table.** (This operation is supported but not recommended, as writing properties in the table creation statement makes future modifications inconvenient).

```
create table if not exists h3(
id bigint,
name string,
price double
) using hudi
options (
primaryKey = 'id',
type = 'mor',
hoodie.cleaner.fileversions.retained = '20',
hoodie.keep.max.commits = '20'
);
```

Caveats

- Currently, Hudi does not support CHAR, VARCHAR, TINYINT, and SMALLINT types; you are advised to use string or INT types.
- Currently, only **int**, **bigint**, **float**, **double**, **decimal**, **string**, **date**, **timestamp**, **boolean**, and **binary** types in Hudi support setting default values.
- You must specify **primaryKey** and **preCombineField** for Hudi tables.
- When creating a table at a specified path, if a Hudi table already exists at the path, you do not need to specify columns during table creation, and you cannot modify the table's original properties.

Permission Requirements

Metadata service provided by DLI

 Copyright © Huawei Technologies Co., Ltd.

- SQL permissions:

database	table
CREATE_TABLE	None

- Fine-grained permission: **dli:database:createTable**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

The table is successfully created. The created Hudi table can be accessed by entering the DLI console, choosing **Data Management** > **Databases and Tables** from the left navigation pane, and then clicking the name of the database where the table is created.

5.1.2 DROP TABLE

Function

This command is used to delete an existing table.

Syntax

DROP TABLE *[IF EXISTS] [db_name.]table_name;*

Parameter Description

Table 5-3 Parameter descriptions

Parameter	Description
db_name	Database name. If this parameter is not specified, the current database is selected.
table_name	Name of the table to be deleted.

Caveats

- In this command, **IF EXISTS** and **db_name** are optional.
- When this statement is used to drop a foreign table, the data in the OBS directory is not automatically deleted.
- When deleting an MOR table, the tables with the **_rt** and **_ro** suffixes are not automatically deleted. To delete them, you need to execute a DROP statement separately.

Example

DROP TABLE IF EXISTS hudidb.h1;

 Copyright © Huawei Technologies Co., Ltd.

Permission Requirements

Metadata service provided by DLI

- SQL permissions:

database	table
DROP_TABLE	None

- Fine-grained permission: **dli:table:dropTable**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

The table will be deleted.

5.1.3 SHOW TABLE

Function

This command is used to display all tables in current database or all tables in a specific database.

Syntax

SHOW TABLES *[IN db_name];*

Parameter Description

Table 5-4 Parameter descriptions

Parameter	Description
IN db_name	Name of the specific database whose tables need to be all displayed.

Caveats

IN db_Name is optional. It is required only when you need to display all tables of a specific database.

Example

```
SHOW TABLES IN hudidb;
```

Permission Requirements

Metadata service provided by DLI

 Copyright © Huawei Technologies Co., Ltd.

- SQL permissions:

database	table
LIST_TABLES, DISPLAY_ALL_TABLES, SELECT (only one of them required)	None

- Fine-grained permission: **dli:database:displayAllTables**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

You can directly view the job result on the job submission page, or go to the **SQL Jobs** page, locate the job, click **More** in the Operation column, and select **View Result**.

5.1.4 TRUNCATE TABLE

Function

This command is used to clear all data in a specific table.

Syntax

TRUNCATE TABLE *tableIdentifier*

Parameter Description

Table 5-5 Parameter descriptions

Parameter	Description
tableIdentifier	Table name.

Example

```
truncate table h0_1;
```

System Response

Data in the table is cleared. You can run the **QUERY** statement to check whether data in the table has been deleted.

5.2 Hudi DML Syntax

 Copyright © Huawei Technologies Co., Ltd.

5.2.1 CREATE TABLE AS SELECT

Function

This command creates a Hudi table by specifying a list of fields with table attributes. When using the metadata service provided by DLI, only foreign tables can be created, meaning you need to specify the table path through **LOCATION**.

Syntax

CREATE TABLE *[IF NOT EXISTS] [database_name.]table_name*

USING hudi

[COMMENT table_comment]

[LOCATION location_path]

[OPTIONS (options_list)]

[AS query_statement]

Parameter Description

Table 5-6 Parameter descriptions of CREATE TABLE As SELECT

Parameter	Description
database_name	Name of the database, consisting of letters, numbers, and underscores (_)
table_name	Name of the table in the database, consisting of letters, numbers, and underscores (_)
using	Uses **hudi** to define and create a Hudi table.
table_comment	Description of the table
location_path	OBS path. If specified, the Hudi table will be created as a foreign table.
options_list	List of Hudi table options.
query_statement	SELECT statement

Example

- Create a partitioned table.

```
create table h2 using hudi
options (type = 'cow', primaryKey = 'id', preCombineField = 'dt')
partitioned by (dt)
as
select 1 as id, 'a1' as name, 10 as price, 1000 as dt;
```

- Create a non-partitioned table.

```
create table h3 using hudi
options (type = 'cow', primaryKey = 'id', preCombineField = 'dt')
```

 Copyright © Huawei Technologies Co., Ltd.

```
as
select 1 as id, 'a1' as name, 10 as price, 1000 as dt;

Load data from a Parquet table to the Hudi table.
# Create a Parquet table.
create table parquet_mngd using parquet options(path='obs://bucket/path/parquet_dataset/
*.parquet');

# Create a Hudi table by Creating a Table from Query Results (CTAS).
create table hudi_tbl using hudi location 'obs://bucket/path/hudi_tbl/' options (
type = 'cow',
primaryKey = 'id',
preCombineField = 'ts'
)
partitioned by (datestr) as select * from parquet_mngd;
```

Caveats

CTAS uses **bulk insert** to write data for improved loading performance.

Permission Requirements

Metadata service provided by DLI

- SQL permissions:

database	table
CREATE_TABLE	Source table: SELECT

- Fine-grained permissions: **dli:table:createTable** and **dli:table:select**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

The table is successfully created. The created Hudi table can be accessed by entering the DLI console, choosing **Data Management** > **Databases and Tables** from the left navigation pane, and then clicking the name of the database where the table is created.

5.2.2 INSERT INTO

Function

This command is used to insert the output of the **SELECT** statement to a Hudi table.

Syntax

INSERT INTO *tableIndentifier select query;*

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

Table 5-7 Parameter descriptions

Parameter	Description
tableIndentifier	Name of the Hudi table.
select query	SELECT statement.

Caveats

- Insert mode: Hudi supports three insert modes for tables with primary keys. You can set **hoodie.sql.insert.mode** to specify the insert mode. The default value is **upsert**.
  ```
  hoodie.sql.insert.mode = upsert
  ```
  - In strict mode, the INSERT statement retains the primary key constraint of COW tables and is not allowed to insert duplicate records. If a record already exists during data insertion, HoodieDuplicateKeyException is thrown for COW tables. For MOR tables, the behavior in this mode is the same as that in upsert mode.
  - In non-strict mode, records are inserted to primary key tables.
  - In upsert mode, duplicate values in the primary key table are updated.
- When submitting a Spark SQL job, you can configure the following parameters to switch bulk insert as the writing method for INSERT statements.
  ```
  hoodie.sql.bulk.insert.enable = true
  hoodie.sql.insert.mode = non-strict
  ```
- Alternatively, you can set **hoodie.datasource.write.operation** to control the writing method of INSERT statements, with options including **bulk_insert**, **insert**, and **upsert**. (Note: This will override the result of the configured **hoodie.sql.insert.mode**).
  ```
  hoodie.datasource.write.operation = upsert
  ```

Example

```
insert into h0 select 1, 'a1', 20;

-- insert static partition
insert into h_p0 partition(dt = '2021-01-02') select 1, 'a1';

-- insert dynamic partition
insert into h_p0 select 1, 'a1', dt;

-- insert dynamic partition
insert into h_p1 select 1 as id, 'a1', '2021-01-03' as dt, '19' as hh;

-- insert overwrite table
insert overwrite table h0 select 1, 'a1', 20;

-- insert overwrite table with static partition
insert overwrite h_p0 partition(dt = '2021-01-02') select 1, 'a1';

-- insert overwrite table with dynamic partition
insert overwrite table h_p1 select 2 as id, 'a2', '2021-01-03' as dt, '19' as hh;
```

 Copyright © Huawei Technologies Co., Ltd.

Permission Requirements

Metadata service provided by DLI

- SQL permissions:

database	table
None	INSERT_INTO_TABLE

- Fine-grained permission: **dli:table:insertIntoTable**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

You can confirm the job status as successful and run a QUERY statement to view the data written in the table.

5.2.3 MERGE INTO

Function

This command is used to query another table based on the join condition of a table or subquery. If **UPDATE** or **DELETE** is executed for the table matching the join condition, and **INSERT** is executed if the join condition is not met. This command completes the synchronization requiring only one full table scan, delivering higher efficiency than **INSERT** plus **UPDATE**.

Syntax

MERGE INTO *tableIdentifier AS target_alias*

USING *(sub_query | tableIdentifier) AS source_alias*

ON *<merge_condition>*

[WHEN MATCHED [AND <condition>] THEN <matched_action>]

[WHEN MATCHED [AND <condition>] THEN <matched_action>]

[WHEN NOT MATCHED [AND <condition>] THEN <not_matched_action>]

<merge_condition> =A equal bool condition

<matched_action> =

DELETE |

*UPDATE SET * |*

UPDATE SET column1 = expression1 [, column2 = expression2 ...]

<not_matched_action> =

*INSERT * |*

 Copyright © Huawei Technologies Co., Ltd.

INSERT (column1 [, column2 ...]) VALUES (value1 [, value2 ...])

Parameter Description

Table 5-8 Parameter descriptions

Parameter	Description
tableIdentifier	Name of the Hudi table.
target_alias	Alias of the target table.
sub_query	Subquery.
source_alias	Alias of the source table or source expression.
merge_condition	Condition for associating the source table or expression with the target table.
condition	Filtering condition. This parameter is optional.
matched_action	DELETE or UPDATE operation to be performed when conditions are met.
not_matched_action	INSERT operation to be performed when conditions are not met.

Caveats

1. The merge-on condition supports only primary key columns currently.
2. Currently, only some fields in the COW table can be updated, and the updated values must contain the pre-merged columns. All fields in the MOR table must be provided in the Update statement.

Example

- Update some fields.

```
create table h0(id int, comb int, name string, price int) using hudi options(primaryKey = 'id',
preCombineField = 'comb') LOCATION 'obs://bucket/path/h0';
create table s0(id int, comb int, name string, price int) using hudi options(primaryKey = 'id',
preCombineField = 'comb') LOCATION 'obs://bucket/path/s0';
insert into h0 values(1, 1, 1, 1);
insert into s0 values(1, 1, 1, 1);
insert into s0 values(2, 2, 2, 2);
// Method 1
merge into h0 using s0
on h0.id = s0.id
when matched then update set h0.id = s0.id, h0.comb = s0.comb, price = s0.price * 2;
// Method 2
merge into h0 using s0
on h0.id = s0.id
when matched then update set id = s0.id,
name = h0.name,
comb = s0.comb + h0.comb,
price = s0.price + h0.price;
```

- Update and insert default fields.

```
create table h0(id int, comb int, name string, price int, flag boolean) using hudi
 options(primaryKey = 'id', preCombineField = 'comb') LOCATION 'obs://bucket/path/h0';
```

 Copyright © Huawei Technologies Co., Ltd.

```
create table s0(id int, comb int, name string, price int, flag boolean) using hudi
  options(primaryKey = 'id', preCombineField = 'comb') LOCATION 'obs://bucket/path/s0';
insert into h0 values(1, 1, 1, 1, false);
insert into s0 values(1, 2, 1, 1, true);
insert into s0 values(2, 2, 2, 2, false);

merge into h0 as target
using (
select id, comb, name, price, flag from s0
) source
on target.id = source.id
when matched then update set *
when not matched then insert *;
```

- Update and delete data based on multiple conditions.

```
create table h0(id int, comb int, name string, price int, flag boolean) using hudi
  options(primaryKey = 'id', preCombineField = 'comb') LOCATION 'obs://bucket/path/h0';
create table s0(id int, comb int, name string, price int, flag boolean) using hudi
   options(primaryKey = 'id', preCombineField = 'comb') LOCATION 'obs://bucket/path/s0';
insert into h0 values(1, 1, 1, 1, false);
insert into h0 values(2, 2, 1, 1, false);
insert into s0 values(1, 1, 1, 1, true);
insert into s0 values(2, 2, 2, 2, false);
insert into s0 values(3, 3, 3, 3, false);

merge into h0
using (
select id, comb, name, price, flag from s0
) source
on h0.id = source.id
when matched and flag = false then update set id = source.id, comb = h0.comb + source.comb, price =
source.price * 2
when matched and flag = true then delete
when not matched then insert *;
```

System Response

You can check if the job status is successful and review the job logs to confirm if there are any exceptions.

5.2.4 UPDATE

Function

This command is used to update the Hudi table based on the column expression and optional filtering conditions.

Syntax

UPDATE *tableIdentifier SET column = EXPRESSION(,column = EXPRESSION) [WHERE boolExpression]*

Parameter Description

Table 5-9 Parameter descriptions

Parameter	Description
tableIdentifier	Name of the Hudi table to be updated.
column	Target column to be updated.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
EXPRESSION	Expression of the source table column to be updated in the target table.
boolExpression	Filtering condition expression.

Example

```
update h0 set price = price + 20 where id = 1;
update h0 set price = price *2, name = 'a2' where id = 2;
```

System Response

You can confirm the job status as successful and run a QUERY statement to verify that the data in the table has been updated.

5.2.5 DELETE

Function

This command is used to delete records from a Hudi table.

Syntax

DELETE from *tableIdentifier [WHERE boolExpression]*

Parameter Description

Table 5-10 Parameter descriptions

Parameter	Description
tableIdentifier	Name of the Hudi table to delete records.
boolExpression	Filtering conditions for deleting records.

Example

- Example 1:
  ```
  delete from h0 where column1 = 'country';
  ```
- Example 2:
  ```
  delete from h0 where column1 IN ('country1', 'country2');
  ```
- Example 3:
  ```
  delete from h0 where column1 IN (select column11 from sourceTable2);
  ```
- Example 4:
  ```
  delete from h0 where column1 IN (select column11 from sourceTable2 where column1 = 'xxx');
  ```
- Example 5:
  ```
  delete from h0;
  ```

 Copyright © Huawei Technologies Co., Ltd.

System Response

You can confirm the job status as successful and run a QUERY statement to verify that the corresponding data in the table has been deleted.

5.2.6 COMPACTION

Function

This command is used to convert row-based log files in MOR tables into column-based data files in parquet tables to accelerate record search.

Syntax

SCHEDULE COMPACTION on *tableIdentifier* |tablelocation;

SHOW COMPACTION on *tableIdentifier* |tablelocation;

RUN COMPACTION on *tableIdentifier* |tablelocation [at instant-time];

Parameter Description

Table 5-11 Parameter descriptions

Parameter	Description
tableIdentifier	Name of the Hudi table to convert log files.
tablelocation	Storage path of the Hudi table.
instant-time	Time to run the command. You can run the **show compaction** command to view the **instant-time** value.

Example

```
schedule compaction  on h1;
show compaction on h1;
run compaction on h1 at 20210915170758;

schedule compaction  on 'obs://bucket/path/h1';
run compaction on 'obs://bucket/path/h1';
```

Caveats

- When triggering compaction for Hudi tables created via SQL using the API method, you need to set **hoodie.payload.ordering.field** to the value of **preCombineField**.
- When using the metadata service provided by DLI, this command does not support OBS paths.

 Copyright © Huawei Technologies Co., Ltd.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.2.7 ARCHIVELOG

Function

Archives instants on the Timeline based on configurations and deletes archived instants from the Timeline to reduce the operation pressure on the Timeline.

Syntax

RUN ARCHIVELOG ON tableIdentifier;

RUN ARCHIVELOG ON tablelocation;

Parameter Description

Table 5-12 Parameter descriptions

Parameter	Description
tableIdentifier	Name of the Hudi table
tablelocation	Storage path of the Hudi table

Example

```
run archivelog on h1;
run archivelog on "obs://bucket/path/h1";
```

Caveats

- First, execute the clean command. After the clean command has cleaned up the historical data files, only the corresponding instants on the timeline can be archived.
- No matter whether the compaction operation is performed, at least *x* (*x* indicates the value of **hoodie.compact.inline.max.delta.commits**) instants are retained and not archived to ensure that there are enough instants to trigger the compaction schedule.
- When using the metadata service provided by DLI, this command does not support OBS paths.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

 Copyright © Huawei Technologies Co., Ltd.

5.2.8 CLEAN

Function

Cleans instants on the Timeline based on configurations and deletes historical version files to reduce the data storage and read/write pressure of Hudi tables.

Syntax

RUN CLEAN ON tableIdentifier;

RUN CLEAN ON tablelocation;

Parameter Description

Table 5-13 Parameter descriptions

Parameter	Description
tableIdentifier	Name of the Hudi table
tablelocation	Storage path of the Hudi table

Example

```
run clean on h1;
run clean on 'obs://bucket/path/h1';
```

Caveats

- The clean operation can only be executed by the owner of the table.
- To modify the default parameters of the clean command, you need to configure the parameters such as the number of commits to be retained in the settings when executing the SQL command. Refer to **Typical Hudi Configuration Parameters**.
- When using the metadata service provided by DLI, this command does not support OBS paths.

System Response

You can check whether the job status is successful, and view the job log to confirm whether there is any exception.

5.2.9 CLEANARCHIVE

Function

This command is used to clean up archive files of Hudi tables to reduce data storage and read/write pressure on Hudi tables.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

- To clean up based on file size, you need to configure parameters:

```
hoodie.archive.file.cleaner.policy = KEEP_ARCHIVED_FILES_BY_SIZE;
hoodie.archive.file.cleaner.size.retained = 5368709120;
```

  Submitting SQL statements

```
run cleanarchive on tableIdentifier/tablelocation;
```

- To clean up based on retention time, you need to configure parameters:

```
hoodie.archive.file.cleaner.policy = KEEP_ARCHIVED_FILES_BY_DAYS;
hoodie.archive.file.cleaner.days.retained = 30;
```

  Submitting SQL statements

```
run cleanarchive on tableIdentifier/tablelocation;
```

Parameter Description

Table 5-14 Parameter descriptions

Parameter	Description
tableIdentifier	Name of the Hudi table
tablelocation	Storage path of the Hudi table
hoodie.archive.file.cleaner.policy	Policy for clearing archived files: Currently, only the **KEEP_ARCHIVED_FILES_BY_SIZE** and **KEEP_ARCHIVED_FILES_BY_DAYS** policies are supported. The default policy is **KEEP_ARCHIVED_FILES_BY_DAYS**. • **KEEP_ARCHIVED_FILES_BY_SIZE**: used to configure the storage capacity that can be used by archived files. • **KEEP_ARCHIVED_FILES_BY_DAYS**: used to delete archived files beyond a specified time point.
hoodie.archive.file.cleaner.size.retained	When the deletion policy is **KEEP_ARCHIVED_FILES_BY_SIZE**, this parameter specifies the number of bytes of archived files to be retained. The default value is **5368709120** bytes (5 GB).
hoodie.archive.file.cleaner.days.retained	When the deletion policy is **KEEP_ARCHIVED_FILES_BY_DAYS**, this parameter specifies the number of days for storing archived files. The default value is **30** days.

Caveats

- Archived files do not have backups and cannot be restored after deletion.
- When using the metadata service provided by DLI, this command does not support OBS paths.

 Copyright © Huawei Technologies Co., Ltd.

System Response

You can check whether the job status is successful, and view the job log to confirm whether there is any exception.

5.3 Hudi CALL COMMAND Syntax

5.3.1 CLEAN_FILE

Function

Cleans invalid data files from the Hudi table directory.

Syntax

call clean_file(table => '[table_name]', mode=>'[op_type]', backup_path=>'[backup_path]', start_instant_time=>'[start_time]', end_instant_time=>'[end_time]');

Parameter Description

Table 5-15 Parameter descriptions

Parameter	Description
table_name	Mandatory. Name of the Hudi table from which invalid data files are to be deleted.
op_type	Optional. Command running mode. The default value is **dry_run**. Value options are **dry_run**, **repair**, **undo**, and **query**. **dry_run**: displays invalid data files to be cleaned. **repair**: displays and cleans invalid data files. **undo**: restores deleted data files. **query**: displays the backup directories that have been cleaned.
backup_path	Mandatory. Backup directory of the data files to be restored. This parameter is available only when the running mode is **undo**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
start_time	Optional. Start time for generating invalid data files. This parameter is available only when the running mode is **dry_run** or **repair**. The start time is not limited by default.
end_time	Optional. End time for generating invalid data files. This parameter is available only when the running mode is **dry_run** or **repair**. The end time is not limited by default.

Example

```
call clean_file(table => 'h1', mode=>'repair');
call clean_file(table => 'h1', mode=>'dry_run');
call clean_file(table => 'h1', mode=>'query');
call clean_file(table => 'h1', mode=>'undo', backup_path=>'obs://bucket/hudi/h1/.hoodie/.cleanbackup/
hoodie_repair_backup_20230527');
```

Caveats

The command cleans only invalid Parquet files.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.3.2 SHOW_TIME_LINE

Function

Displays the effective or archived Hudi timelines and details of a specified instant time.

Syntax

- Viewing the list of effective timelines of a table:

  call show_active_instant_list(table => '[table_name]');
- Viewing the list of effective timelines after a timestamp in a table:

  call show_active_instant_list(table => '[table_name]', instant => '[instant]');
- Viewing information about an instant that takes effect in a table:

  call show_active_instant_detail(table => '[table_name]', instant => '[instant]');
- Viewing the list of archived instant timelines in a table:

  call show_archived_instant_list(table => '[table_name]');

 Copyright © Huawei Technologies Co., Ltd.

- Viewing the list of archived instant timelines after a timestamp in a table:
  call show_archived_instant_list(table => '[table_name]', instant => '[instant]');
- Viewing information about archived instants in a table:
  call show_archived_instant_detail(table => '[table_name]', instant => '[instant]');

Parameter Description

Table 5-16 Parameter descriptions

Parameter	Description
table_name	Name of the table to be queried. The value can be in the **database.tablename** format.
instant	Instant timestamp to be queried

Example

```
call show_active_instant_detail(table => 'hudi_table1', instant => '20230913144936897');
```

Permission Requirements

Metadata service provided by DLI

- SQL permissions:

database	table
None	SELECT

- Fine-grained permission: **dli:table:select**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.3.3 SHOW_HOODIE_PROPERTIES

Function

Displays the configuration in the **hoodie.properties** file of a specified Hudi table.

Syntax

call show_hoodie_properties(table => '[table_name]');

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

Table 5-17 Parameter descriptions

Parameter	Description
table_name	Name of the table to be queried. The value can be in the **database.tablename** format.

Example

```
call show_hoodie_properties(table => "hudi_table5");
```

Permission Requirements

Metadata service provided by DLI

- SQL permissions:

database	table
None	SELECT

- Fine-grained permission: **dli:table:select**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.3.4 ROLL_BACK

Function

Rolls back a specified commit.

Syntax

call rollback_to_instant(table => '[table_name]', instant_time => '[instant]');

Parameter Description

Table 5-18 Parameter descriptions

Parameter	Description
table_name	Mandatory. Name of the Hudi table to be rolled back.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
instant	Mandatory. Commit instant timestamp of the Hudi table to be rolled back.

Example

```
call rollback_to_instant(table => 'h1', instant_time=>'20220915113127525');
```

Caveats

- You can only roll back the latest commit timestamp sequentially. You can check the latest instant time using the **SHOW_TIME_LINE** command.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.3.5 CLUSTERING

Function

Performs the clustering operation on Hudi tables. For details, see **Hudi Clustering**.

Syntax

- Performing clustering:

  call run_clustering(table=>'[table]', path=>'[path]', predicate=>'[predicate]', order=>'[order]');

- Viewing the clustering plan:

  call show_clustering(table=>'[table]', path=>'[path]', limit=>[limit]);

Parameter Description

Table 5-19 Parameter descriptions

Parameter	Description	Mandatory
table	Name of the table to be queried. The value can be in the **database.tablename** format.	Either **table** or **path** must be set.
path	Path of the table to be queried	Either **table** or **path** must be set.
predicate	Predicate to be defined, which is used to filter partitions to be clustered	No

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Mandatory
order	Sorting field for clustering	No
limit	Number of query results to display	No

Example

```
call show_clustering(table => 'hudi_table1');

call run_clustering(table => 'hudi_table1', predicate => '(ts >= 1006L and ts < 1008L) or ts >= 1009L', order => 'ts');

call run_clustering(path => 'obs://bucket/path/hudi_test2', predicate => "dt = '2021-08-28'", order => 'id');
```

Caveats

- Either **table** or **path** must exist. Otherwise, the Hudi table to be clustered cannot be determined.
- When using the metadata service provided by DLI, this command only supports configuring the **table** parameter and does not support configuring the **path** parameter.
- To cluster a specified partition, refer to the format **predicate => "dt = '2023-08-28'"**.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.3.6 CLEANING

Function

Cleans Hudi tables. For details, see **Hudi Clean**.

Syntax

call run_clean(table=>'[table]', clean_policy=>'[clean_policy]', retain_commits=>'[retain_commits]', hours_retained=> '[hours_retained]', file_versions_retained=> '[file_versions_retained]');

 Copyright © Huawei Technologies Co., Ltd.

Parameter Description

Table 5-20 Parameter descriptions

Parameter	Description	Mandatory
table	Name of the table to be queried. The value can be in the *database.tablename* format.	Yes
clean_policy	Policy for deleting data files of an earlier version. The default value is **KEEP_LATEST_COMMITS**.	No
retain_commits	This parameter is available only when **clean_policy** is set to **KEEP_LATEST_COMMITS**.	No
hours_retained	This parameter is available only when **clean_policy** is set to **KEEP_LATEST_BY_HOURS**.	No
file_version_retained	This parameter is available only when **clean_policy** is set to **KEEP_LATEST_FILE_VERSIONS**.	No

Example

```
call run_clean(table => 'hudi_table1');

call run_clean(table => 'hudi_table1', retain_commits => 2);

call run_clean(table => 'hudi_table1', clean_policy => 'KEEP_LATEST_FILE_VERSIONS', file_version_retained
=> 1);
```

Caveats

The cleaning operation cleans data files of an earlier version in partitions only when trigger conditions are met. If trigger conditions are not met, this operation does not clean the data files even if the command is successfully executed.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

 Copyright © Huawei Technologies Co., Ltd.

5.3.7 COMPACTION

Function

Compacts Hudi tables. For details, see **Hudi Compaction**.

Syntax

```
call run_compaction(op => '[op]', table=>'[table]', path=>'[path]', timestamp=>'[timestamp]');
```

Parameter Description

Table 5-21 Parameter descriptions

Parameter	Description	Mandatory
op	Set this parameter to **schedule** to generate a compaction plan or to **run** to execute a generated compaction plan.	Yes
table	Name of the table to be queried. The value can be in the *database.tablename* format.	Either **table** or **path** must be set.
path	Path of the table to be queried	Either **table** or **path** must be set.
timestamp	When **op** is set to **run**, you can specify **timestamp** to execute the compaction plan corresponding to the timestamp and the compaction plan that is not executed before the timestamp.	No

Example

```
call run_compaction(table => 'hudi_table1', op => 'schedule');
call run_compaction(table => 'hudi_table1', op => 'run');
call run_compaction(table => 'hudi_table1', op => 'run', timestamp => 'xxx');
call run_compaction(path => 'obs://bucket/path/hudi_table1', op => 'run', timestamp => 'xxx');
```

Caveats

Only MOR tables can be compacted.

 Copyright © Huawei Technologies Co., Ltd.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.3.8 SHOW_COMMIT_FILES

Function

Checks whether multiple files are updated in or inserted to a specified instant.

Syntax

call show_commit_files(table=>'[table]', instant_time=>'[instant_time]', limit=>[limit]);

Parameter Description

Table 5-22 Parameter descriptions

Parameter	Description	Mandatory
table	Name of the table to be queried. The value can be in the *database*.*tablename* format.	Yes
instant_time	Timestamp corresponding to a commit operation	Yes
limit	Number of returned items	No

Example

```
call show_commit_files(table=>'hudi_mor', instant_time=>'20230216144548249');
call show_commit_files(table=>'hudi_mor', instant_time=>'20230216144548249', limit=>1);
```

Returned Result

Parameter	Description
action	Action type of the commit operation corresponding to **instant_time**, such as **compaction**, **deltacommit**, and **clean**
partition_path	Partition where the file updated in or inserted to a specified instant is located
file_id	ID of the file updated in or inserted to the specified instant

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
previous_commit	Timestamp in the file name of the file updated in or inserted to the specified instant
total_records_updated	Number of updated records in the file
total_records_written	Number of records inserted into the file
total_bytes_written	Number of bytes of data added to the file
total_errors	Total number of errors reported during the update in or insertion to the specified instant
file_size	File size, in bytes

Permission Requirements

Metadata service provided by DLI

- SQL permissions:

database	table
None	SELECT

- Fine-grained permission: **dli:table:select**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.3.9 SHOW_FS_PATH_DETAIL

Function

Checks statistics about a specified FS path.

Syntax

```
call show_fs_path_detail(path=>'[path]', is_sub=>'[is_sub]', sort=>'[sort]');
```

Parameter Description

Table 5-23 Parameter descriptions

Parameter	Description	Mandatory
path	Path of the FS to be queried	Yes
is_sub	The default value is **false**, indicating that statistics about a specified directory is collected. The value **true** indicates that statistics about subdirectories in a specified directory is collected.	No
sort	The default value is **true**, indicating that the results are sorted based on **storage_size**. The value **false** indicates that the results are sorted based on the number of files.	No

Example

```
call show_fs_path_detail(path=>'obs://bucket/path/hudi_mor/dt=2021-08-28', is_sub=>false, sort=>true);
```

Returned Result

Parameter	Description
path_num	Number of subdirectories in a specified directory
file_num	Number of files in a specified directory
storage_size	Size of the directory, in bytes
storage_size(unit)	Size of the directory, in KB
storage_path	Complete FS absolute path of the specified directory
space_consumed	Actual space occupied by the returned files/directories in the cluster, that is, the replication factor set for the cluster is considered.
quota	Name quota, which is a mandatory restriction on the number of files and directory names in the current directory tree

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
space_quota	Space quota, which is a mandatory restriction on the number of bytes used by files in the current directory tree

Caveats

- This command is not supported when the metadata service provided by DLI is used.

Permission Requirements

Metadata service provided by DLI

- SQL permissions: not supported.

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.3.10 SHOW_LOG_FILE

Function

This command is used to view the meta and record information in log files.

Syntax

- Viewing meta information:

  call show_logfile_metadata(table => '[table]', log_file_path_pattern => '[log_file_path_pattern]', limit => [limit])

- Viewing record information:

  call show_logfile_records(table => '[table]', log_file_path_pattern => '[log_file_path_pattern]', merge => '[merge]', limit => [limit])

Parameter Description

Table 5-24 Parameter descriptions

Parameter	Description	Mandatory
table	Name of the table to be queried. The value can be in the *database*.*tablename* format.	Yes

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Mandatory
log_file_path_pattern	Path of log files. Regular expression matching is supported.	No
merge	When **show_logfile_records** is executed, this parameter is used to specify whether to combine records in multiple log files and return them together.	No
limit	Number of returned items	No

Example

```
call show_logfile_metadata(table => 'hudi_mor', log_file_path_pattern => 'obs://bucket/path/hudi_mor/dt=2021-08-28/.*?log.*?');
call show_logfile_records(table => 'hudi_mor', log_file_path_pattern => 'obs://bucket/path/hudi_mor/dt=2021-08-28/.*?log.*?', merge => false, limit => 1);
```

Caveats

- This command is used only for MOR tables.

Permission Requirements

Metadata service provided by DLI

- SQL permissions:

database	table
None	SELECT

- Fine-grained permission: **dli:table:select**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.3.11 SHOW_INVALID_PARQUET

Function

Checks the damaged parquet file in the execution path.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

call show_invalid_parquet(path => 'path')

Parameter Description

Table 5-25 Parameter descriptions

Parameter	Description	Mandatory
path	Path of the FS to be queried	Yes

Example

```
call show_invalid_parquet(path => 'obs://path/hudi_table/dt=2021-08-28');
```

Caveats

- This command is not supported when the metadata service provided by DLI is used.

Permission Requirements

Metadata service provided by DLI

- SQL permissions:

database	table
None	SELECT

- Fine-grained permission: **dli:table:select**

Metadata services provided by LakeFormation. Refer to the LakeFormation documentation for details on permission configuration.

System Response

You can check if the job status is successful, view the job result, and review the job logs to confirm if there are any exceptions.

5.4 Schema Evolution Syntax

Function

This capability supports column alterations on Hudi tables with SparkSQL. Schema evolution must be enabled before using this capability.

Scope of Schema Evolution

The scope of schema evolution includes:

 Copyright © Huawei Technologies Co., Ltd.

- Column operations: Supports adding, deleting, modifying, and adjusting the position of columns (including nested columns).
- Partition columns: Does not support schema evolution on partition columns.
- Array type columns: Does not support adding, deleting, or altering nested columns of Array type.

5.4.1 ALTER COLUMN

Enabling Schema Evolution

Set the following parameter:

```
hoodie.schema.evolution.enable=true
```

Function

The **ALTER TABLE ... ALTER COLUMN** command is used to change the attributes of a column, such as the column type, position, and comment.

Syntax

ALTER TABLE *Table name* **ALTER**

[COLUMN] *col_old_name* **TYPE** *column_type*

[COMMENT] *col_comment*

[FIRST|AFTER] *column_name*

Parameter Description

Table 5-26 ALTER COLUMN parameters

Parameter	Description
tableName	Table name.
col_old_name	Name of the column to be altered.
column_type	Type of the target column.
col_comment	Column comment.
column_name	New position to place the target column. For example, **AFTER column_name** indicates that the target column is placed after **column_name**.

Example

- Changing the column type
  ```
  ALTER TABLE table1 ALTER COLUMN a.b.c TYPE bigint
  ```
  a.b.c indicates the full path of a nested column. For details about the nested column rules, see **ADD COLUMNS**.

 Copyright © Huawei Technologies Co., Ltd.

The following changes on column types are supported:

- int => long/float/double/string/decimal
- long => float/double/string/decimal
- float => double/String/decimal
- From double to string or decimal
- From decimal to decimal or string
- From string to date or decimal
- From date to string

- Altering other attributes

```
ALTER TABLE table1 ALTER COLUMN a.b.c DROP NOT NULL
ALTER TABLE table1 ALTER COLUMN a.b.c COMMENT 'new comment'
ALTER TABLE table1 ALTER COLUMN a.b.c FIRST
ALTER TABLE table1 ALTER COLUMN a.b.c AFTER x
```

a.b.c indicates the full path of a nested column. For details about the nested column rules, see **ADD COLUMNS**.

System Response

You can run the **DESCRIBE** command to view the modified column.

5.4.2 ADD COLUMNS

Enabling Schema Evolution

Set the following parameter:

```
hoodie.schema.evolution.enable=true
```

Function

The **ADD COLUMNS** command is used to add a column to an existing table.

Syntax

ALTER TABLE *Table name* **ADD COLUMNS***(col_spec[, col_spec ...])*

Parameter Description

Table 5-27 ADD COLUMNS parameters

Parameter	Description
tableName	Table name.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
col_spec	Column specifications, consisting of five fields, **col_name**, **col_type**, **nullable**, **comment**, and **col_position**. • **col_name**: name of the new column. It is mandatory. To add a sub-column to a nested column, specify the full name of the sub-column in this field. For example: – To add sub-column **col1** to a nested struct type column **users struct<name: string, age: int>**, set this field to **users.col1**. – To add sub-column **col1** to a nested map type column **member map<string, struct<n: string, a: int>>**, set this field to **member.value.col1**. – To add sub-column **col2** to a nested array type column **arraylike array<struct<a1: string, a2: int>>**, set this field to **arraylike.element.col2**. • **col_type**: type of the new column. It is mandatory. • **nullable**: whether the new column can be null. The value can be left empty. • **comment**: comment of the new column. The value can be left empty. • **col_position**: position where the new column is added. The value can be **FIRST** or **AFTER origin_col**. If it is set to **FIRST**, the new column will be added to the first column of the table. If it is set to **AFTER origin_col**, the new column will be added after original column **origin_col**. The value can be left empty. **FIRST** can be used only when new sub-columns are added to nested columns. Do not use **FIRST** in top-level columns. There are no restrictions about the usage of **AFTER**.

Example

```
alter table h0 add columns(ext0 string);
alter table h0 add columns(new_col int not null comment 'add new column' after col1);
alter table complex_table add columns(col_struct.col_name string comment 'add new column to a struct
col' after col_from_col_struct);
```

System Response

You can run the **DESCRIBE** command to view the new column.

 Copyright © Huawei Technologies Co., Ltd.

5.4.3 RENAME COLUMN

Enabling Schema Evolution

Set the following parameter:

```
hoodie.schema.evolution.enable=true
```

Function

The **ALTER TABLE ... RENAME COLUMN** command is used to change the column name.

Syntax

ALTER TABLE *tableName* **RENAME COLUMN** *old_columnName* **TO** *new_columnName*

Parameter Description

Table 5-28 Parameter descriptions

Parameter	Description
tableName	Table name.
old_columnName	Old column name.
new_columnName	New column name.

Example

```
ALTER TABLE table1 RENAME COLUMN a.b.c TO x
```

a.b.c indicates the full path of a nested column. For details about the nested column rules, see **ADD COLUMNS**.

NOTE

After the column name is changed, the change is automatically synchronized to the column comment. The comment is in **rename oldName to newName** format.

System Response

You can run the **SHOW CREATE TABLE** command to view the new column name.

5.4.4 RENAME TABLE

Enabling Schema Evolution

Set the following parameter:

```
hoodie.schema.evolution.enable=true
```

 Copyright © Huawei Technologies Co., Ltd.

Function

The **ALTER TABLE ... RENAME** command is used to change the table name.

Syntax

ALTER TABLE *tableName* **RENAME TO** *newTableName*

Parameter Description

Table 5-29 RENAME parameters

Parameter	Description
tableName	Table name.
newTableName	New table name.

Example

```
ALTER TABLE table1 RENAME TO table2
```

System Response

You can run the **SHOW TABLES** command to view the new table name.

5.4.5 SET

Enabling Schema Evolution

Set the following parameter:

```
hoodie.schema.evolution.enable=true
```

Function

The **ALTER TABLE ... SET|UNSET** command is used to modify table properties.

Syntax

ALTER TABLE *Table name* **SET|UNSET** *tblproperties*

Parameter Description

Table 5-30 Parameter descriptions

Parameter	Description
tableName	Table name.
tblproperties	Table properties.

 Copyright © Huawei Technologies Co., Ltd.

Example

```
ALTER TABLE table SET TBLPROPERTIES ('table_property' = 'property_value')
ALTER TABLE table UNSET TBLPROPERTIES [IF EXISTS] ('comment', 'key')
```

System Response

You can run the **DESCRIBE** command to view new table properties.

5.4.6 DROP COLUMN

Enabling Schema Evolution

Set the following parameter:

```
hoodie.schema.evolution.enable=true
```

Function

The **ALTER TABLE ... DROP COLUMN** command is used to delete a column.

Syntax

ALTER TABLE *tableName* **DROP COLUMN|COLUMNS** *cols*

Parameter Description

Table 5-31 DROP COLUMN parameters

Parameter	Description
tableName	Table name.
cols	Columns to be deleted. You can specify multiple columns.

Example

```
ALTER TABLE table1 DROP COLUMN a.b.c
ALTER TABLE table1 DROP COLUMNS a.b.c, x, y
```

a.b.c indicates the full path of a nested column. For details about the nested column rules, see **ADD COLUMNS**.

System Response

You can run the **DESCRIBE** command to check which column is deleted.

5.5 Configuring Default Values for Hudi Data Columns

This feature allows you to set default values for columns when you add columns to a table. When you query historical data, the default value is returned for the new column.

 Copyright © Huawei Technologies Co., Ltd.

Constraints

- If data has been rewritten before default values are set for a new column, the default values of the column cannot be returned when historical data is queried. In this case, NULL values are returned. Some or all data will be rewritten when data is imported to the database, updated, compacted, or clustered.
- The default values of a column must match the column type. If they do not match, the type will be forcibly converted. This loses the precision of the default values or changes the default values to NULL.
- The default values of historical data are the default values set for the column for the first time. Changing the default values of a column for multiple times does not affect the query result of historical data.
- After the default value is set, it cannot be rolled back.
- Currently, Spark SQL does not support the function of viewing default column values. You can run the **show create table** SQL statements to view default column values.
- The default method of writing missing columns is not supported. **Column names must be specified** when writing.

Scope

Currently, only the **int**, **bigint**, **float**, **double**, **decimal**, **string**, **date**, **timestamp**, **boolean**, and **binary** data types are supported.

Table 5-32 Engines supported

Engine	DDL Operation Support	Write Operation Support	Read Operation Support
SparkSQL	Y	Y	Y
Spark DataSource	N	N	Y
Flink	N	N	Y
Hive	N	N	Y

Example

For details about the SQL syntax, see **DLI Hudi SQL Syntax Reference**.

Example:

- Create a table and specify default values for columns.

```
create table if not exists h3(
id bigint,
name string,
price double default 12.34
) using hudi
options (
primaryKey = 'id',
type = 'mor',
preCombineField = 'name'
);
```

 Copyright © Huawei Technologies Co., Ltd.

- Add columns and specify default values for the columns.
```
alter table h3 add columns(col1 string default 'col1_value');
alter table h3 add columns(col2 string default 'col2_value', col3 int default 1);
```
- Change default values of columns.
```
alter table h3 alter column price set default 14.56;
```
- When inserting data using column default values, you need to specify the column names being written, corresponding one-to-one with the inserted data.
```
insert into h3(id, name) values(1, 'aaa');
insert into h3(id, name, price) select 2, 'bbb', 12.5;
```

 Copyright © Huawei Technologies Co., Ltd.

6 Spark DataSource API Syntax Reference

For how to submit a Spark Jar job, see **Submitting a Spark Jar Job in DLI Using Hudi**.

6.1 API Syntax Description

Setting Write Modes

Hudi uses the **hoodie.datasource.write.operation** parameter to set the write mode.

- insert: This operation does not require querying specific update file partitions through the index, so it is faster than upsert. You are advised to use this operation when there is no update data. If there is update data, using this operation may result in duplicate data.
- bulk_insert: This operation sorts the primary key and inserts it into the Hudi table as a regular Parquet table. It has the highest performance but cannot control small files, whereas upsert and insert can control small files well.
- upsert: The default operation type. Hudi determines whether the data to be inserted contains update data based on the primary key. If it contains updates, it performs an upsert; otherwise, it performs an insert.

NOTE

- Because insert does not sort primary keys, you are advised not to use insert when initializing the dataset. Use bulk_insert instead.
- You are advised to use insert when it is confirmed that all data is new, and upsert when there is update data.

Example: bulk_insert writing a COW non-partitioned table

```
df.write.format("org.apache.hudi").
option("hoodie.datasource.write.table.type", COW_TABLE_TYPE_OPT_VAL).
option("hoodie.datasource.write.precombine.field", "update_time").
option("hoodie.datasource.write.recordkey.field", "id").
option("hoodie.datasource.write.partitionpath.field", "").
option("hoodie.datasource.write.operation", "bulk_insert").
option("hoodie.table.name", tableName).
option("hoodie.write.lock.provider", "com.huawei.luxor.hudi.util.DliCatalogBasedLockProvider").
option("hoodie.datasource.write.keygenerator.class",
"org.apache.hudi.keygen.NonpartitionedKeyGenerator").
```

 Copyright © Huawei Technologies Co., Ltd.

```
option("hoodie.datasource.hive_sync.enable", "true").
option("hoodie.datasource.hive_sync.partition_fields", "").
option("hoodie.datasource.hive_sync.partition_extractor_class",
"org.apache.hudi.hive.NonPartitionedExtractor").
option("hoodie.datasource.hive_sync.database", databaseName).
option("hoodie.datasource.hive_sync.table", tableName).
option("hoodie.datasource.hive_sync.use_jdbc", "false").
option("hoodie.bulkinsert.shuffle.parallelism", 4).
mode(SaveMode.Overwrite).
save(basePath)
```

Setting Partitions

- Multi-level partitioning

Parameter	Description
hoodie.datasource.write.partitionpath.field	Set it to multiple service fields separated by commas (,).
hoodie.datasource.hive_sync.partition_fields	Set it to the same as **hoodie.datasource.write.partitionpath.field**.
hoodie.datasource.write.keygenerator.class	Set it to **org.apache.hudi.keygen.ComplexKeyGenerator**.
hoodie.datasource.hive_sync.partition_extractor_class	Set it to **org.apache.hudi.hive.MultiPartKeysValueExtractor**.

Example: Creating a multi-level partitioned COW table with partitions p1/p2/p3

```
df.write.format("org.apache.hudi").
option("hoodie.datasource.write.table.type", COW_TABLE_TYPE_OPT_VAL).
option("hoodie.datasource.write.precombine.field", "update_time").
option("hoodie.datasource.write.recordkey.field", "id").
option("hoodie.datasource.write.partitionpath.field", "year,month,day").
option("hoodie.datasource.write.operation", "bulk_insert").
option("hoodie.table.name", tableName).
option("hoodie.write.lock.provider", "com.huawei.luxor.hudi.util.DliCatalogBasedLockProvider").
option("hoodie.datasource.write.keygenerator.class",
"org.apache.hudi.keygen.ComplexKeyGenerator").
option("hoodie.datasource.hive_sync.enable", "true").
option("hoodie.datasource.hive_sync.partition_fields", "year,month,day").
option("hoodie.datasource.hive_sync.partition_extractor_class",
"org.apache.hudi.hive.MultiPartKeysValueExtractor").
option("hoodie.datasource.hive_sync.database", databaseName).
option("hoodie.datasource.hive_sync.table", tableName).
option("hoodie.datasource.hive_sync.use_jdbc", "false").
mode(SaveMode.Overwrite).
save(basePath)
```

- Single-level partitioning

Parameter	Description
hoodie.datasource.write.partitionpath.field	Set it to a service field.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
hoodie.datasource.hive_sync.partition_fields	Set it to the same as **hoodie.datasource.write.partitionpath.field**.
hoodie.datasource.write.keygenerator.class	The default value is **org.apache.hudi.keygen.ComplexKeyGenerator**. You can also set it to **org.apache.hudi.keygen.SimpleKeyGenerator**. If left unspecified, the system automatically uses the default value.
hoodie.datasource.hive_sync.partition_extractor_class	Set it to **org.apache.hudi.hive.MultiPartKeysValueExtractor**.

Example: Creating a single partitioned MOR table with partition p1

```
df.write.format("org.apache.hudi").
option("hoodie.datasource.write.table.type", MOR_TABLE_TYPE_OPT_VAL).
option("hoodie.datasource.write.precombine.field", "update_time").
option("hoodie.datasource.write.recordkey.field", "id").
option("hoodie.datasource.write.partitionpath.field", "create_time").
option("hoodie.datasource.write.operation", "bulk_insert").
option("hoodie.table.name", tableName).
option("hoodie.write.lock.provider", "com.huawei.luxor.hudi.util.DliCatalogBasedLockProvider").
option("hoodie.datasource.write.keygenerator.class",
"org.apache.hudi.keygen.ComplexKeyGenerator").
option("hoodie.datasource.hive_sync.enable", "true").
option("hoodie.datasource.hive_sync.partition_fields", "create_time").
option("hoodie.datasource.hive_sync.partition_extractor_class",
"org.apache.hudi.hive.MultiPartKeysValueExtractor").
option("hoodie.datasource.hive_sync.database", databaseName).
option("hoodie.datasource.hive_sync.table", tableName).
option("hoodie.datasource.hive_sync.use_jdbc", "false").
mode(SaveMode.Overwrite).
save(basePath)
```

- Non-partitioning

Parameter	Description
hoodie.datasource.write.partitionpath.field	Set it to an empty string.
hoodie.datasource.hive_sync.partition_fields	Set it to an empty string.
hoodie.datasource.write.keygenerator.class	Set it to **org.apache.hudi.keygen.NonpartitionedKeyGenerator**.
hoodie.datasource.hive_sync.partition_extractor_class	Set it to **org.apache.hudi.hive.NonPartitionedExtractor**.

 Copyright © Huawei Technologies Co., Ltd.

Example: Creating a non-partitioned COW table

```
df.write.format("org.apache.hudi").
option("hoodie.datasource.write.table.type", COW_TABLE_TYPE_OPT_VAL).
option("hoodie.datasource.write.precombine.field", "update_time").
option("hoodie.datasource.write.recordkey.field", "id").
option("hoodie.datasource.write.partitionpath.field", "").
option("hoodie.datasource.write.operation", "bulk_insert").
option("hoodie.table.name", tableName).
option("hoodie.write.lock.provider", "com.huawei.luxor.hudi.util.DliCatalogBasedLockProvider").
option("hoodie.datasource.write.keygenerator.class",
"org.apache.hudi.keygen.NonpartitionedKeyGenerator").
option("hoodie.datasource.hive_sync.enable", "true").
option("hoodie.datasource.hive_sync.partition_fields", "").
option("hoodie.datasource.hive_sync.partition_extractor_class",
"org.apache.hudi.hive.NonPartitionedExtractor").
option("hoodie.datasource.hive_sync.database", databaseName).
option("hoodie.datasource.hive_sync.table", tableName).
option("hoodie.datasource.hive_sync.use_jdbc", "false").
mode(SaveMode.Overwrite).
save(basePath)
```

- Time-date partitioning

Parameter	Description
hoodie.datasource.write.partitionpath.field	Set it to a value of the date type, in the format of **yyyy/mm/dd**.
hoodie.datasource.hive_sync.partition_fields	Set it to the same as **hoodie.datasource.write.partitionpath.field**.
hoodie.datasource.write.keygenerator.class	The default value is **org.apache.hudi.keygen.SimpleKeyGenerator**. You can set it to **org.apache.hudi.keygen.ComplexKeyGenerator**.
hoodie.datasource.hive_sync.partition_extractor_class	Set it to **org.apache.hudi.hive.SlashEncodedDayPartitionValueExtractor**.

NOTE

The SlashEncodedDayPartitionValueExtractor has the following constraint: The date format must be *yyyy/mm/dd*.

6.2 Hudi Lock Configuration

When submitting a Spark JAR job, you need to manually configure the Hudi lock.

When using the metadata service hosted by DLI, the Hudi lock must be enabled, and the implementation class of the Hudi lock provided by DLI must be configured.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Value
hoodie.write.lock.provider	com.huawei.luxor.hudi.util.DliCatalogBasedLockProvider

Similarly, when using the metadata service provided by LakeFormation, the implementation class of the Hudi lock provided by LakeFormation must be configured and used.

Parameter	Value
hoodie.write.lock.provider	org.apache.hudi.lakeformation.LakeCatMetastoreBasedLockProvider

CAUTION

- Disabling the Hudi lock or using other lock implementation classes poses a risk of data loss or anomalies.
- Under any circumstances, DLI is not responsible for any form of loss or damage, direct or indirect, caused by disabling the Hudi lock or using lock implementation classes that do not match the metadata service. This includes but is not limited to, loss of business profits, service interruption, data loss, or other financial losses.

 Copyright © Huawei Technologies Co., Ltd.

7 Data Management and Maintenance

7.1 Hudi Compaction

What Is Compaction?

Compaction is used to merge the base and log files of merge-on-read (MOR) tables. Compaction involves two processes: Schedule and Run. The schedule process generates a compaction plan in the timeline, which records which Parquet files will be merged with which log files. However, this is just a plan and does not perform the merge. The run process executes all the compaction plans in the timeline one by one until all are completed.

For MOR tables, data is stored using columnar Parquet files and row-based Avro files. Updates are recorded in incremental files, and then synchronous/asynchronous compaction is performed to generate new versions of the columnar files. MOR tables can reduce data ingestion latency, making asynchronous compaction that does not block ingestion meaningful.

How to Execute Compaction?

1. Schedule only
   - Spark SQL (Set the following parameters, trigger on data write)

     ```
     hoodie.compact.inline=true
     hoodie.schedule.compact.only.inline=true
     hoodie.run.compact.only.inline=false
     hoodie.compact.inline.max.delta.commits=5 // The default value is 5, but you can adjust it based on the service scenario.
     ```

     After executing any write SQL, compaction will be triggered when the condition is met (for example, there are 5 delta log files under the same file slice).
   - Spark SQL (Set the following parameters, manually trigger once)

     ```
     hoodie.compact.inline=true
     hoodie.schedule.compact.only.inline=true
     hoodie.run.compact.only.inline=false
     hoodie.compact.inline.max.delta.commits=5 // The default value is 5, but you can adjust it based on the service scenario.
     ```

     Then manually execute SQL:

     ```
     schedule compaction on ${table_name}
     ```

 Copyright © Huawei Technologies Co., Ltd.

- SparkDataSource (Set the following parameters in the option, trigger on data write)

  hoodie.compact.inline=true

  hoodie.schedule.compact.only.inline=true

  hoodie.run.compact.only.inline=false

  hoodie.compact.inline.max.delta.commits=5 // The default value is **5**, but you can adjust it based on the service scenario.

- Flink (Set the following parameters in the with attribute, trigger on data write)

  compaction.async.enabled=false

  compaction.schedule.enabled=true

  compaction.delta_commits=5 // The default value is **5**, but you can adjust it based on the service scenario.

2. Run only
   - Spark SQL (Set the following parameters, manually trigger once)

     ```
     hoodie.compact.inline=true
     hoodie.schedule.compact.only.inline=false
     hoodie.run.compact.only.inline=true
     ```

     Then execute the following SQL:

     ```
     run compaction on ${table_name}
     ```

3. Execute Schedule and Run together

   If there is no compaction plan in the timeline, it will attempt to generate and execute a compaction plan.

   - Spark SQL (Set the following parameters, trigger on data write when the condition is met)

     ```
     hoodie.compact.inline=true
     hoodie.schedule.compact.only.inline=false
     hoodie.run.compact.only.inline=false
     hoodie.compact.inline.max.delta.commits=5 // The default value is 5, but you can adjust it based on the service scenario.
     ```

   - SparkDataSource (Set the following parameters in the option, trigger on data write)

     hoodie.compact.inline=true

     hoodie.schedule.compact.only.inline=false

     hoodie.run.compact.only.inline=false

     hoodie.compact.inline.max.delta.commits=5 // The default value is **5**, but you can adjust it based on the service scenario.

   - Flink (Set the following parameters in the with attribute, trigger on data write)

     compaction.async.enabled=true

     compaction.schedule.enabled=false

     compaction.delta_commits=5 // The default value is **5**, but you can adjust it based on the service scenario.

4. Recommended approach
   - Spark/Flink Streaming jobs: Execute only Schedule, and then set up a separate Spark SQL job to execute Run at regular intervals.

 Copyright © Huawei Technologies Co., Ltd.

- Spark batch jobs: Execute Schedule and Run together directly.

  NOTE

  To ensure the highest efficiency of data lake ingestion, you are advised to produce the compaction schedule plan synchronously and execute the compaction schedule plan asynchronously.

7.2 Hudi Clean

What Is Clean?

Cleaning is used to remove **old version data files (Parquet files or log files) in Hudi tables** that are no longer needed. This reduces storage pressure and improves the efficiency of list operations.

How to Execute Clean?

1. Clean after writing data.
   - Spark SQL (Set the following parameters, trigger on data write when the condition is met)

     ```
     hoodie.clean.automatic=true
     hoodie.cleaner.commits.retained=10 // The default value is 10, but you can adjust it based on the service scenario.
     ```

   - SparkDataSource (Set the following parameters in the option, trigger on data write)

     hoodie.clean.automatic=true

     hoodie.cleaner.commits.retained=10 // The default value is **10**, but you can adjust it based on the service scenario.

   - Flink (Set the following parameters in the with attribute, trigger on data write)

     clean.async.enabled=true

     clean.retain_commits=10 // The default value is **10**, but you can adjust it based on the service scenario.
2. Manually trigger clean once.
   - Spark SQL (Set the following parameters, manually trigger once)

     ```
     hoodie.clean.automatic=true
     hoodie.cleaner.commits.retained=10 // The default value is 10, but you can adjust it based on the service scenario.
     ```

     Execute the SQL. When there are more than 10 instant records in the timeline, the clean operation is triggered:

     ```
     run clean on ${table_name}
     ```

7.3 Hudi Archive

What Is Archive?

Archiving is used to clean up metadata files in Hudi tables (located in the **.hoodie** directory, formatted as **${timestamp}.${operation_type}.${operation_status}**, for example, **20240622143023546.deltacommit.request**). Each operation on a Hudi

table generates metadata files, and too many metadata files can lead to performance issues. Therefore, it is best to keep the number of metadata files under 1,000.

How to Execute Archive?

1. Archive after writing data.
   - Spark SQL (Set the following parameters, trigger on data write)

     ```
     hoodie.archive.automatic=true
     hoodie.keep.max.commits=30 // The default value is 30, but you can adjust it based on the service scenario.
     hoodie.keep.min.commits=20 // The default value is 20, but you can adjust it based on the service scenario.
     ```

   - SparkDataSource (Set the following parameters in the option, trigger on data write)

     hoodie.archive.automatic=true

     hoodie.keep.max.commits=30 // The default value is **30**, but you can adjust it based on the service scenario.

     hoodie.keep.min.commits=20 // The default value is **20**, but you can adjust it based on the service scenario.

   - Flink (Set the following parameters in the with attribute, trigger on data write)

     hoodie.archive.automatic=true

     archive.max_commits=30 // The default value is **30**, but you can adjust it based on the service scenario.

     archive.min_commits=20 // The default value is **20**, but you can adjust it based on the service scenario.

2. Manually trigger archive once.
   - Spark SQL (Set the following parameters, manually trigger once)

     hoodie.archive.automatic=true
     hoodie.keep.max.commits=30 // The default value is **30**, but you can adjust it based on the service scenario.
     hoodie.keep.min.commits=20 // The default value is **20**, but you can adjust it based on the service scenario.

     Execute the SQL. When a clean operation has been performed and the timeline contains instant records of data files that have already been cleaned, and the total number of instant records exceeds 30, the archive operation is triggered.

     run archivelog on *${table_name}*

7.4 Hudi Clustering

Introduction

Clustering reorganizes data layout to improve query performance without affecting the ingestion speed.

Architecture

Hudi provides different operations, such as **insert**, **upsert**, and **bulk_insert**, through its write client API to write data to a Hudi table. To weight between file

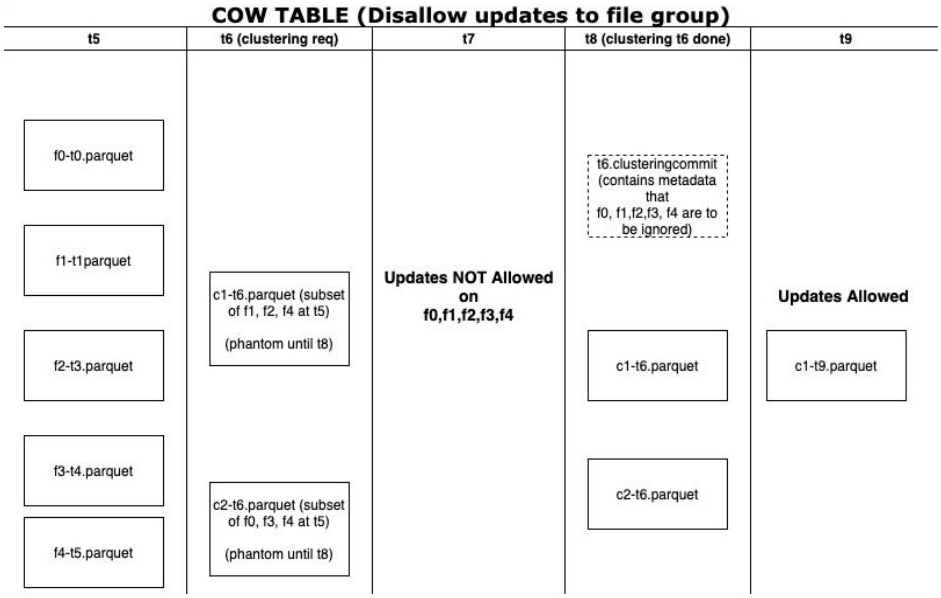
 Copyright © Huawei Technologies Co., Ltd.

size and speed of importing data into the data lake, Hudi provides **hoodie.parquet.small.file.limit** to configure the minimum file size. You can set it to **0** to force new data to be written to new file groups, or to a higher value to ensure that new data is "padded" to existing small file groups until it reaches the specified size, but this increases ingestion latency.

To support fast ingestion without affecting query performance, the clustering service is introduced to rewrite data to optimize the layout of Hudi data lake files.

The clustering service can run asynchronously or synchronously. It adds a new operation type called **REPLACE**, which will mark the clustering operation in the Hudi metadata timeline.

Clustering service is based on the MVCC design of Hudi to allow new data to be inserted. Clustering operations run in the background to reformat data layout, ensuring snapshot isolation between concurrent readers and writers.



Clustering is divided into two parts:

- Scheduling clustering: Create a clustering plan using a pluggable clustering strategy.
  a. Identify files that are eligible for clustering: Depending on the selected clustering strategy, the scheduling logic will identify the files eligible for clustering.
  b. Group files that are eligible for clustering based on specific criteria. The data size of each group must be a multiple of **targetFileSize**. Grouping is a part of the strategy defined in the plan. Additionally, there is an option to control group size to improve parallelism and avoid shuffling large volumes of data.
  c. Save the clustering plan to the timeline in Avro metadata format.
- Execute clustering: Process the plan using an execution strategy to create new files and replace old files.
  a. Read the clustering plan and obtain **clusteringGroups** that marks the file groups to be clustered.

 Copyright © Huawei Technologies Co., Ltd.

b. Instantiate appropriate strategy class for each group using **strategyParams** (for example, **sortColumns**) and apply the strategy to rewrite data.

c. Create a **REPLACE** commit and update the metadata in HoodieReplaceCommitMetadata.

How to Execute Clustering

1. Spark SQL (Set the following parameters, trigger on data write)

   hoodie.clustering.inline=true // The default value is **false**, meaning clustering is disabled by default.
   hoodie.clustering.inline.max.commits=4 // The default value is **4**, but you can adjust it based on the service scenario.
   hoodie.clustering.plan.strategy.max.bytes.per.group=2147483648 // The default value is 2 GB, but you can adjust it based on the service scenario. Generally, no need to specify as the normal data amount under each file group does not exceed 2 GB.
   hoodie.clustering.plan.strategy.max.num.groups=30 // The default value is **30**, but you can adjust it based on the service scenario. Usually, adjust this parameter to adjust the data amount for each clustering plan (**max.bytes.per.group** x **max.num.groups**).
   hoodie.clustering.plan.strategy.partition.regex.pattern=${Regular expression} // No default value. If not specified, clustering will reorganize data across all partitions.
   hoodie.clustering.plan.strategy.small.file.limit=314572800 // The default value is 300 MB, but you can adjust it based on the service scenario. Files smaller than 300 MB in each partition will be selected for clustering.
   hoodie.clustering.plan.strategy.sort.columns=${Column 1,...,Column N} // No default value, but you can set it based on the service scenario. Specify columns frequently used in queries and do not contain null.
   hoodie.clustering.plan.strategy.target.file.max.bytes=1073741824 // The default value is 1 GB, but you can adjust it based on the service scenario. It is used to set the maximum size of files generated by clustering.

2. SparkDataSource (Set the following parameters in the option, trigger on data write)

   option("hoodie.clustering.inline", "true").

   option("hoodie.clustering.inline.max.commits", 4).

   option("hoodie.clustering.plan.strategy.max.bytes.per.group", 2147483648).

   option("hoodie.clustering.plan.strategy.max.num.groups", 30).

   option("hoodie.clustering.plan.strategy.partition.regex.pattern", "${Regular expression}").

   option("hoodie.clustering.plan.strategy.small.file.limit", 314572800).

   option("hoodie.clustering.plan.strategy.sort.columns", "${Column 1,.....,Column N}").

   option("hoodie.clustering.plan.strategy.target.file.max.bytes", 1073741824).

3. Manually trigger clustering once.

   – Spark SQL (Set the following parameters, manually trigger once)

   hoodie.clustering.inline=true
   hoodie.clustering.inline.max.commits=4 // The default value is **4**, but you can adjust it based on the service scenario.
   hoodie.clustering.plan.strategy.max.bytes.per.group=2147483648 // The default value is 2 GB, but you can adjust it based on the service scenario. Generally, no need to specify as the normal data amount under each file group does not exceed 2 GB.
   hoodie.clustering.plan.strategy.max.num.groups=30 // The default value is **30**, but you can adjust it based on the service scenario. Usually, adjust this parameter to adjust the data amount for each clustering plan (**max.bytes.per.group** x **max.num.groups**).
   hoodie.clustering.plan.strategy.partition.regex.pattern=${Regular expression} // No default value. If not specified, clustering will reorganize data across all partitions.
   hoodie.clustering.plan.strategy.small.file.limit=314572800 // The default value is 300 MB, but you

 Copyright © Huawei Technologies Co., Ltd.

can adjust it based on the service scenario. Files smaller than 300 MB in each partition will be selected for clustering.
hoodie.clustering.plan.strategy.sort.columns=${Column 1,...,Column N} // No default value, but you can set it based on the service scenario. Specify columns frequently used in queries and do not contain null.
hoodie.clustering.plan.strategy.target.file.max.bytes=1073741824 // The default value is 1 GB, but you can adjust it based on the service scenario. It is used to set the maximum size of files generated by clustering.

Run the following SQL statement:

```
call run_clustering (table =>'${Table name}')
```

CAUTION

1. Clustering sort columns cannot contain null values, due to Spark RDD limitations.
2. When the value of **target.file.max.bytes** is large, increase **--executor-memory** to avoid executor memory overflow.
3. Clean does not support cleaning residual files after clustering failures.
4. New files generated after clustering may vary in size, which could cause data skew.
5. Clustering does not support concurrency with upsert (write operation updates files waiting for clustering). If clustering is inflight, files under that FileGroup cannot be updated.
6. If there are unfinished clustering plans, subsequent write triggers generating compaction plans may fail. Execute clustering plans promptly.

 Copyright © Huawei Technologies Co., Ltd.

8 Typical Hudi Configuration Parameters

This section describes important Hudi configurations. For details, visit the Hudi official website at **https://hudi.apache.org/cn/docs/0.11.0/configurations/**.

- To set Hudi parameters while submitting a DLI Spark SQL job, access the **SQL Editor** page and click **Settings** in the upper right corner. In the **Parameter Settings** area, set the parameters.
- When submitting DLI Spark jar jobs, Hudi parameters can be configured through the Spark datasource API's options.

  Alternatively, you can configure them in **Spark Arguments(--conf)** when submitting the job. Note that when configuring parameters here, the key needs to have the prefix **spark.hadoop.**, for example, **spark.hadoop.hoodie.compact.inline=true**.

Write Configuration

Table 8-1 Write configuration parameters

Parameter	Description	Default Value
hoodie.datasource.write.table.name	Name of the Hudi table to write to	None

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.datasource.write.operation	Operation type for writing to the Hudi table. Currently, **upsert**, **delete**, **insert**, and **bulk_insert** are supported. • **upsert**: updates and inserts data. • **delete**: deletes data. • **insert**: inserts data. • **bulk_insert**: imports data during initial table creation. Do not use **upsert** or **insert** during initial table creation. • **insert_overwrite**: performs insert and overwrite operations on static partitions. • **insert_overwrite_table**: performs insert and overwrite operations on dynamic partitions. It does not immediately delete the entire table or overwrite the table. Instead, it overwrites the metadata of the Hudi table logically, and Hudi deletes useless data through the clean mechanism. More efficient than bulk_insert + overwrite.	upsert
hoodie.datasource.write.table.type	Type of Hudi table. Once specified, this parameter cannot be modified later. Option: **MERGE_ON_READ**.	COPY_ON_WRITE
hoodie.datasource.write.precombine.field	Merges and reduplicates rows with the same key before write.	A specific table field
hoodie.datasource.write.payload.class	Class used to merge the records to be updated and the updated records during update. This parameter can be customized. You can compile it to implement your merge logic.	org.apache.hudi.common.model.DefaultHoodieRecordPayload
hoodie.datasource.write.recordkey.field	Unique primary key for the Hudi table	A specific table field

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.datasource.write.partitionpath.field	Partition key. This parameter can be used together with **hoodie.datasource.write.keygenerator.class** to meet different partition needs.	None
hoodie.datasource.write.hive_style_partitioning	Whether to specify a partition mode that is the same as that of Hive. Set it to **true**.	true
hoodie.datasource.write.keygenerator.class	Used with **hoodie.datasource.write.partitionpath.field** and **hoodie.datasource.write.recordkey.field** to generate the primary key and partition mode. **NOTE** If the value of this parameter is different from that saved in the table, a message is displayed, indicating that the value must be the same.	org.apache.hudi.keygen.ComplexKeyGenerator

Configuration of Hive Table Synchronization

The metadata service provided by DLI is a Hive Metastore service (HMS), so the following parameters are related to synchronizing the metadata service.

Table 8-2 Parameters for synchronizing Hive tables

Parameter	Description	Default Value
hoodie.datasource.hive_sync.enable	Whether to synchronize Hudi tables to Hive. When using the metadata service provided by DLI, configuring this parameter means synchronizing to the metadata of DLI. **CAUTION** You are advised to set it to **true** to use the metadata service to manage Hudi tables.	true
hoodie.datasource.hive_sync.database	Name of the database to be synchronized to Hive	default

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.datasource.hive_sync.table	Name of the table to be synchronized to Hive. Set it to the value of **hoodie.datasource.write.table.name**.	unknown
hoodie.datasource.hive_sync.partition_fields	Hive partition columns	""
hoodie.datasource.hive_sync.partition_extractor_class	Class used to extract Hudi partition column values and convert them into Hive partition columns.	org.apache.hudi.hive.SlashEncodedDayPartitionValueExtractor
hoodie.datasource.hive_sync.support_timestamp	If the Hudi table contains a field of the timestamp type, set this parameter to **true** to synchronize the timestamp type to the Hive metadata. The default value is **false**, indicating that the timestamp type is converted to bigint during synchronization by default. In this case, an error may occur when you query a Hudi table that contains a field of the timestamp type using SQL statements.	true
hoodie.datasource.hive_sync.fast_sync	Methods for synchronizing Hudi partitions to Hive. ● If this parameter is set to **true**, **add partition if not exist** is used to synchronize Hive tables based on the partitions filtered out in the Hudi table. ● If this parameter is set to **false**, check whether the partitions to be synchronized exist in the Hive table. If not, synchronize the partitions into the Hive table.	true

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.datasource.hive_sync.mode	The method for synchronizing Hudi tables to Hive tables. When using the metadata service provided by DLI, this must be set to **hms**: • **hms**: synchronizes metadata through the Hive metastore client. • **jdbc**: synchronizes metadata through the Hive JDBC mode. (Not supported by the DLI metadata service.) • **hiveql**: synchronizes metadata by running the **hive ql** command. (Not supported by the DLI metadata service.)	hms
hoodie.datasource.hive_sync.username	Username specified when synchronizing Hive using JDBC	hive
hoodie.datasource.hive_sync.password	Password specified when synchronizing Hive using JDBC	hive
hoodie.datasource.hive_sync.jdbcurl	JDBC URL specified for connecting to Hive	""
hoodie.datasource.hive_sync.use_jdbc	Whether to use Hive JDBC to synchronize Hudi table information to Hive. You are advised to set this parameter to **false**. When set to **false**, the JDBC connection-related configuration will be invalid.	true

Index Configuration

Table 8-3 Index parameters

Parameter	Description	Default Value
hoodie.index.class	Full path of a user-defined index, which must be a subclass of HoodieIndex. When this parameter is specified, the configuration takes precedence over that of **hoodie.index.type**.	""

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.index.type	Index type. The default value is **BLOOM**. Possible options are **BLOOM**, **GLOBAL_BLOOM**, **SIMPLE**, and **GLOBAL_SIMPLE**. The Bloom filter eliminates the dependency on an external system and is stored in the footer of a Parquet data file.	BLOOM
hoodie.index.bloom.num_entries	Number of entries stored in the Bloom filter. Assuming **maxParquetFileSize** is 128 MB and **averageRecordSize** is 1,024 bytes, the total number of records in a file is about 130 KB. The default value (**60000**) is about half of this approximation. **CAUTION** Setting this value too low will result in many false positives, and the index lookup will need to scan more files than necessary. Setting it too high will linearly increase the size of each data file (approximately 4 KB for every 50,000 entries).	60000
hoodie.index.bloom.fpp	The allowed error rate based on the number of entries. Used to calculate the number of bits to allocate for the Bloom filter and the number of hash functions. This value is typically set very low (default: **0.000000001**) to trade off disk space for a lower false positive rate.	0.000000001
hoodie.bloom.index.parallelism	Parallelism for index lookup involving Spark Shuffle. By default, it is automatically calculated based on input workload characteristics.	0
hoodie.bloom.index.prune.by.ranges	When set to **true**, file range information can speed up index lookups. This is particularly useful if the keys have a monotonically increasing prefix, such as timestamps.	true

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.bloom.index.use.caching	When set to **true**, the input RDD is cached to speed up index lookups by reducing IO required for calculating parallelism or affected partitions.	true
hoodie.bloom.index.use.treebased.filter	When set to **true**, tree-based file filter optimization is enabled. Compared to brute force, this mode speeds up file filtering based on key ranges.	true
hoodie.bloom.index.bucketized.checking	When set to **true**, bucketized Bloom filtering is enabled. This reduces bias seen in sort-based Bloom index lookups.	true
hoodie.bloom.index.keys.per.bucket	This parameter is available only when **bloomIndexBucketizedChecking** is enabled and the index type is **BLOOM**. This configuration controls the size of the "bucket", which tracks the number of record key checks performed on a single file and serves as the work unit assigned to each partition executing Bloom filter lookups. Higher values will amortize the fixed cost of reading the Bloom filter into memory.	10000000
hoodie.bloom.index.update.partition.path	This parameter is applicable only when the index type is **GLOBAL_BLOOM**. When set to **true**, updating a record that includes a partition path will insert the new record into the new partition and delete the original record from the old partition. When set to **false**, only the original record in the old partition is updated.	true

 Copyright © Huawei Technologies Co., Ltd.

Storage Configuration

Table 8-4 Storage parameter configuration

Parameter	Description	Default Value
hoodie.parquet.max.file.size	Target size of the Parquet files generated during the Hudi write phase. For DFS, this should align with the underlying file system block size for optimal performance.	120 * 1024 * 1024 byte
hoodie.parquet.block.size	Parquet page size, which is the read unit in a parquet file. Pages within a block are compressed separately.	120 * 1024 * 1024 byte
hoodie.parquet.compression.ratio	Expected compression ratio for Parquet data when Hudi tries to size new parquet files. Increase this value if the files generated by **bulk_insert** are smaller than expected.	0.1
hoodie.parquet.compression.codec	Name of the parquet compression codec. Default is **gzip**. Possible options are **gzip**, **snappy**, **uncompressed**, and **lzo**.	snappy
hoodie.logfile.max.size	Maximum size of the LogFile. This is the maximum size allowed before rolling over to a new version of the log file.	1 GB
hoodie.logfile.data.block.max.size	Maximum size of the LogFile data block. This is the maximum size of a single data block appended to the log file. It helps ensure that data appended to the log file is broken down into manageable blocks to prevent OOM errors. This size should be greater than the JVM memory.	256 MB
hoodie.logfile.to.parquet.compression.ratio	Expected compression ratio as records move from log files to Parquet. Used in MOR storage to control the size of compressed Parquet files.	0.35

 Copyright © Huawei Technologies Co., Ltd.

Compaction and Cleaning Configuration

Table 8-5 Compaction and cleaning parameters

Parameter	Description	Default Value
hoodie.clean.automatic	Whether to perform automatic cleaning	true
hoodie.cleaner.policy	Cleaning policy to use. Hudi will remove old versions of Parquet files to reclaim space. Any query or computation referencing this version will fail. Ensure data retention exceeds the maximum query execution time.	KEEP_LATEST_COMMITS
hoodie.cleaner.commits.retained	Number of commits to retain. Data will be retained for **num_of_commits * time_between_commits** (planned), which directly translates to the number of incremental pulls on this dataset.	10
hoodie.keep.max.commits	Threshold for the number of commits to trigger archival.	30
hoodie.keep.min.commits	Number of commits to retain for archival	20
hoodie.commits.archival.batch	Controls the number of commit instants to read and archive together in a batch.	10
hoodie.parquet.small.file.limit	Should be less than **maxFileSize**. If set to **0**, this function is disabled. Due to the large number of records inserted into partitions in batch processing, small files will always appear. Hudi provides an option to address the small file problem by treating inserts into the partition as updates to existing small files. The size here is the minimum file size considered a "small file size".	104857600 bytes

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.copyonwrite.insert.split.size	Parallelism for insert writes. Total number of inserts for a single partition. Writing out 100 MB files, with records at least 1 KB in size, means 100 KB records per file. Default is over-provisioned to 500 KB. Adjust this to match the number of records in a single file to improve insert latency. Setting this value smaller results in smaller files (especially if **compactionSmallFileSize** is **0**).	500000
hoodie.copyonwrite.insert.auto.split	Whether Hudi should dynamically calculate **insertSplitSize** based on the last 24 commits' metadata. Default is **false**.	true
hoodie.copyonwrite.record.size.estimate	Average record size. If specified, Hudi will use it instead of dynamically calculating based on the last 24 commits' metadata. No default value. Crucial for calculating insert parallelism and packing inserts into small files.	1024
hoodie.compact.inline	When set to **true**, compaction is triggered by the ingestion itself immediately after the **insert**, **upsert**, **bulk insert**, or **incremental commit** operations.	true
hoodie.compact.inline.max.delta.commits	Maximum number of delta commits to retain before triggering inline compaction.	5
hoodie.compaction.lazy.block.read	Helps choose whether to delay reading log blocks when **CompactedLogScanner** merges all log files. Set it to **true** for I/O-intensive delayed block reading (low memory usage), or **false** for memory-intensive immediate block reading (high memory usage).	true

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.compaction.reverse.log.read	**HoodieLogFormatReader** reads log files forward from **pos=0** to **pos=file_length**. If set to **true**, the reader reads log files backward from **pos=file_length** to **pos=0**.	false
hoodie.cleaner.parallelism	Increase this value if cleaning is slow.	200
hoodie.compaction.strategy	Strategy to determine which file groups to compact during each compaction run. By default, Hudi selects log files with the most unmerged data accumulated.	org.apache.hudi.table.action.compact.strategy. LogFileSizeBasedCompactionStrategy
hoodie.compaction.target.io	Amount of MB to spend during the compaction run in **LogFileSizeBasedCompactionStrategy**. This value helps limit ingestion delays when compaction runs in inline mode.	500 * 1024 MB
hoodie.compaction.daybased.target.partitions	Used by **org.apache.hudi.io.compact.strategy.DayBasedCompactionStrategy**, representing the latest number of partitions to compact during the compaction run.	10
hoodie.compaction.payload.class	Needs to be the same class used during insert/upsert operations. Like writing, compaction uses the record payload class to merge records from the logs with each other, then with the base file again, and generate the final records to be written post-compaction.	org.apache.hudi.common.model.Defaulthoodierecordpayload
hoodie.schedule.compact.only.inline	Whether to only generate a compaction plan during write operations. Valid when **hoodie.compact.inline** is **true**.	false

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.run.compact.only.inline	Whether to only perform compaction operations when executing the run compaction command through SQL. If the compaction plan does not exist, it exits directly.	false

Single-Table Concurrency Control

Table 8-6 Single-table concurrency control configuration

Parameter	Description	Default Value
hoodie.write.lock.provider	Lock provider when metadata is hosted by DLI Recommended value: **com.huawei.luxor.hudi.util.DliCatalogBasedLockProvider**	Spark SQL and Flink SQL jobs will switch to the corresponding implementation class based on the metadata service. For scenarios where metadata is managed by DLI, use **com.huawei.luxor.hudi.util.DliCatalogBasedLockProvider**.
hoodie.write.lock.hivemetastore.database	Database in the HMS service	None
hoodie.write.lock.hivemetastore.table	Table name in the HMS service	None
hoodie.write.lock.client.num_retries	Number of retries	10
hoodie.write.lock.client.wait_time_ms_between_retry	Retry interval	10000
hoodie.write.lock.conflict.resolution.strategy	Lock provider class, must be a subclass of **ConflictResolutionStrategy**.	org.apache.hudi.client.transaction.SimpleConcurrentFileWritesConflictResolutionStrategy

 Copyright © Huawei Technologies Co., Ltd.

Clustering Configuration

NOTE

There are two strategies in clustering: **hoodie.clustering.plan.strategy.class** and **hoodie.clustering.execution.strategy.class**. Typically, when **hoodie.clustering.plan.strategy.class** is set to **SparkRecentDaysClusteringPlanStrategy** or **SparkSizeBasedClusteringPlanStrategy**, there is no need to specify **hoodie.clustering.execution.strategy.class**. However, when **hoodie.clustering.plan.strategy.class** is **SparkSingleFileSortPlanStrategy**, **hoodie.clustering.execution.strategy.class** should be set to **SparkSingleFileSortExecutionStrategy**.

Table 8-7 Clustering parameters

Parameter	Description	Default Value
hoodie.clustering.inline	Whether to execute clustering synchronously	false
hoodie.clustering.inline.max.commits	Number of commits to trigger clustering	4
hoodie.clustering.async.enabled	Whether to enable asynchronous clustering	false
hoodie.clustering.async.max.commits	Number of commits to trigger asynchronous clustering	4
hoodie.clustering.plan.strategy.target.file.max.bytes	Maximum file size after clustering	1024 * 1024 * 1024 byte
hoodie.clustering.plan.strategy.small.file.limit	Files smaller than this size will be clustered	300 * 1024 * 1024 byte
hoodie.clustering.plan.strategy.sort.columns	Columns used for sorting in clustering	None
hoodie.layout.optimize.strategy	Clustering execution strategy. The options are **linear**, **z-order**, and **hilbert**.	linear
hoodie.layout.optimize.enable	Set this parameter to **true** when **z-order** or **hilbert** is used.	false
hoodie.clustering.plan.strategy.class	Strategy class for filtering file groups for clustering. By default, files smaller than the value of **hoodie.clustering.plan.strategy.small.file.limit** are filtered.	org.apache.hudi.client.clustering.plan.strategy.SparkSizeBasedClusteringPlanStrategy

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
hoodie.clustering.execution.strategy.class	Strategy class for executing clustering (subclass of **RunClusteringStrategy**), which defines how to execute a clustering plan. The default class sorts the file groups in the plan by specified columns while meeting the target file size configuration.	org.apache.hudi.client.clustering.run.strategy.SparkSortAndSizeExecutionStrategy
hoodie.clustering.plan.strategy.max.num.groups	Maximum number of FileGroups to select for clustering at execution. Higher values increase concurrency.	30
hoodie.clustering.plan.strategy.max.bytes.per.group	Maximum data per FileGroup to participate in clustering at execution	2 * 1024 * 1024 * 1024 byte

 Copyright © Huawei Technologies Co., Ltd.