Data Lake Insight

SQL Syntax Reference

Issue 01

Date 2022-08-16

HUAWEI TECHNOLOGIES CO., LTD.

Contents

1 Spark SQL Syntax Reference

1.1 Common Configuration Items of Batch SQL Jobs

This section describes the common configuration items of the SQL syntax for DLI batch jobs.

Table 1-1 Common configuration items

Item	Default Value	Description
spark.sql.files.maxRecordsPerFile	0	Maximum number of records to be written into a single file. If the value is zero or negative, there is no limit.
spark.sql.autoBroadcastJoinThreshold	20971520 0	Maximum size of the table that displays all working nodes when a connection is executed. You can set this parameter to **-1** to disable the display. **NOTE** Currently, only the configuration unit metastore table that runs the **ANALYZE TABLE COMPUTE statistics noscan** command and the file-based data source table that directly calculates statistics based on data files are supported.
spark.sql.shuffle.partitions	200	Default number of partitions used to filter data for join or aggregation.

Item	Default Value	Description
spark.sql.dynamicPartitionOverwrite.enabled	false	Whether DLI overwrites the partitions where data will be written into during runtime. If you set this parameter to **false**, all partitions that meet the specified condition will be deleted before data overwrite starts. For example, if you set **false** and use INSERT OVERWRITE to write partition 2021-02 to a partitioned table that has the 2021-01 partition, this partition will be deleted. If you set this parameter to **true**, DLI does not delete partitions before overwrite starts.
spark.sql.files.maxPartitionBytes	134217728	Maximum number of bytes to be packed into a single partition when a file is read.
spark.sql.badRecordsPath	-	Path of bad records.

1.2 SQL Syntax Overview of Batch Jobs

This section describes the Spark SQL syntax list provided by DLI. For details about the parameters and examples, see the syntax description.

Table 1-2 SQL syntax of batch jobs

Classification	Function
Database-related Syntax	**Creating a Database**
	Deleting a Database
	Viewing a Specified Database
	Viewing All Databases
Syntax for Creating an OBS Table	**Creating an OBS Table Using the Datasource Syntax**
	Creating an OBS Table Using the Hive Syntax
Syntax for Deleting a Table	**Deleting a Table**
Syntax for Viewing a Table	**Viewing All Tables**
	Viewing Table Creation Statements

Classification	Function
	Viewing Table Properties
	Viewing All Columns in a Specified Table
	Viewing All Partitions in a Specified Table
	Viewing Table Statistics
Syntax for Modifying a Table	Adding a Column
Syntax for Partitioning a Table	Adding a Partition (Only OBS Tables Supported)
	Renaming a Partition
	Deleting a Partition
	Altering the Partition Location of a Table (Only OBS Tables Supported)
	Updating Partitioned Table Data (Only OBS Tables Supported)
Syntax for Importing Data	Importing Data
Syntax for Inserting Data	Inserting Data
Syntax for Clearing Data	Clearing Data
Syntax for Exporting Query Results	Exporting Query Result
Syntax for Datasource Connection to an HBase Table	Creating a Table and Associating It with HBase
	Inserting Data to an HBase Table
	Querying an HBase Table
Syntax for Datasource Connection to an OpenTSDB Table	Creating a Table and Associating It with OpenTSDB
	Inserting Data to an OpenTSDB Table
	Querying an OpenTSDB Table
Syntax for Datasource Connection to a DWS Table	Creating a Table and Associating It with DWS
	Inserting Data to a DWS Table
	Querying a DWS Table
Syntax for Datasource Connection to an RDS Table	Creating a Table and Associating It with RDS
	Inserting Data to an RDS Table
	Querying an RDS Table

Classification	Function
Syntax for Datasource Connection to a CSS Table	Creating a Table and Associating It with CSS
	Inserting Data to a CSS Table
	Querying a CSS Table
Syntax for Datasource Connection to a DCS Table	Creating a Table and Associating It with DCS
	Inserting Data to a DCS Table
	Querying a DCS Table
Syntax for Datasource Connection to a DDS Table	Creating a Table and Associating It with DDS
	Inserting Data to a DDS Table
	Querying a DDS Table
View-related Syntax	Creating a View
	Deleting a View
Syntax for Viewing the Execution Plan	Viewing the Execution Plan
Syntax Related to Data Permissions	Creating a Role
	Deleting a Role
	Binding a Role
	Unbinding a Role
	Displaying a Role
	Granting a Permission
	Revoking a Permission
	Displaying the Granted Permissions
	Displaying the Binding Relationship Between All Roles and Users
UDF-related Syntax	Creating a Function
	Deleting a Function
	Displaying Function Details
	Displaying All Functions

Classification	Function
Multiversion-related Syntax	**Enabling Multiversion Backup When Creating an OBS Table** **Enabling or Disabling Multiversion Backup When Modifying a Table** **Setting the Retention Period for Multiversion Backup Data** **Viewing Multiversion Backup Data** **Restoring Multiversion Backup Data** **Configuring the Trash Bin for Expired Multiversion Data** **Deleting Multiversion Backup Data**

1.3 Databases

1.3.1 Creating a Database

Function

This statement is used to create a database.

Syntax

```
CREATE [DATABASE | SCHEMA] [IF NOT EXISTS] db_name
  [COMMENT db_comment]
  [WITH DBPROPERTIES (property_name=property_value, ...)];
```

Keyword

- **IF NOT EXISTS**: Prevents system errors if the database to be created exists.
- **COMMENT**: Describes a database.
- **DBPROPERTIES**: Specifies database attributes. The attribute name and attribute value appear in pairs.

Parameters

Table 1-3 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
db_comment	Database description
property_name	Database property name

Parameter	Description
property_value	Database property value

Precautions

- **DATABASE** and **SCHEMA** can be used interchangeably. You are advised to use **DATABASE**.
- The **default** database is a built-in database. You cannot create a database named **default**.

Example

1. Create a queue. A queue is the basis for using DLI. Before executing SQL statements, you need to create a queue.
2. On the DLI management console, click **SQL Editor** in the navigation pane on the left. The **SQL Editor** page is displayed.
3. In the editing window on the right of the **SQL Editor** page, enter the following SQL statement for creating a database and click **Execute**. Read and agree to the privacy agreement, and click **OK**.

   If database **testdb** does not exist, run the following statement to create database **testdb**:

   ```
   CREATE DATABASE IF NOT EXISTS testdb;
   ```

1.3.2 Deleting a Database

Function

This statement is used to delete a database.

Syntax

```
DROP [DATABASE | SCHEMA] [IF EXISTS] db_name [RESTRICT|CASCADE];
```

Keyword

IF EXISTS: Prevents system errors if the database to be deleted does not exist.

Precautions

- **DATABASE** and **SCHEMA** can be used interchangeably. You are advised to use **DATABASE**.
- **RESTRICT**: If the database is not empty (tables exist), an error is reported and the **DROP** operation fails. **RESTRICT** is the default logic.
- **CASCADE**: Even if the database is not empty (tables exist), the **DROP** will delete all the tables in the database. Therefore, exercise caution when using this function.

Parameters

Table 1-4 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).

Example

1. Create a database, for example, **testdb**, by referring to **Example**.
2. Run the following statement to delete database **testdb** if it exists:

```
DROP DATABASE IF EXISTS testdb;
```

1.3.3 Viewing a Specified Database

Function

This syntax is used to view the information about a specified database, including the database name and database description.

Syntax

```
DESCRIBE DATABASE [EXTENDED] db_name;
```

Keyword

EXTENDED: Displays the database properties.

Parameters

Table 1-5 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).

Precautions

If the database to be viewed does not exist, the system reports an error.

Example

1. Create a database, for example, **testdb**, by referring to **Example**.
2. Run the following statement to query information about the **testdb** database:

```
DESCRIBE DATABASE testdb;
```

1.3.4 Viewing All Databases

Function

This syntax is used to query all current databases.

Syntax

```
SHOW [DATABASES | SCHEMAS] [LIKE regex_expression];
```

Keyword

None

Parameters

Table 1-6 Parameter description

Parameter	Description
regex_expression	Database name

Precautions

Keyword DATABASES is equivalent to SCHEMAS. You can use either of them in this statement.

Example

View all the current databases.

```
SHOW DATABASES;
```

View all databases whose names start with **test**.

```
SHOW DATABASES LIKE "test.*";
```

1.4 Creating an OBS Table

1.4.1 Creating an OBS Table Using the DataSource Syntax

Function

This statement is used to create an OBS table using the DataSource syntax. The main differences between the DataSource and the Hive syntax lie in the supported data formats and the number of supported partitions. For details, see syntax and precautions.

Usage

- The size of the table will not be calculated during table creation.
- When data is added, the table size will be changed to 0.
- You can view the table size on OBS.

Syntax

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name1 col_type1 [COMMENT col_comment1], ...)]
  USING file_format
  [OPTIONS (path 'obs_path', key1=val1, key2=val2, ...)]
  [PARTITIONED BY (col_name1, col_name2, ...)]
  [COMMENT table_comment]
  [AS select_statement];
```

Keyword

- IF NOT EXISTS: Prevents system errors when the created table exists.
- USING: Specifies the storage format.
- OPTIONS: Specifies the attribute name and attribute value when a table is created.
- COMMENT: Field or table description.
- PARTITIONED BY: Partition field.
- AS: Run the CREATE TABLE AS statement to create a table.

Parameter

Table 1-7 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). The value cannot contain only digits and cannot start with a digit or underscore (_).
table_name	Table name of a database that contains letters, digits, and underscores (_). The value cannot contain only digits and cannot start with a digit or underscore (_). The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If special characters are required, use single quotation marks ('') to enclose them.
col_name	Column names with data types separated by commas (,). The column name contains letters, digits, and underscores (_). It cannot contain only digits and must contain at least one letter.
col_type	Field type
col_comment	Field description
file_format	Table format stored in an OBS bucket. The value can be ORC, Parquet, JSON, CSV, or Avro.
path	Data storage path

Parameter	Description
table_comment	Table description
select_statement	The CREATE TABLE AS statement is used to insert the SELECT query result of the source table or a data record to a new table in OBS bucket.

Table 1-8 OPTIONS parameter description

Parameter	Description	Default Value
path	Specified table storage location. Currently, only OBS is supported.	-
multiLevelDirEnable	Whether to iteratively query data in subdirectories. When this parameter is set to **true**, all files in the table path, including files in subdirectories, are iteratively read when a table is queried.	false
dataDelegated	Whether to clear data in the path when deleting a table or a partition.	false
compression	Specified compression format. Generally, you need to set this parameter to **zstd** for parquet files.	-

When the file format is set to **CSV**, you can set the following OPTIONS parameters:

Table 1-9 OPTIONS parameter description of the CSV data format

Parameter	Description	Default Value
delimiter	Data separator	Comma (,)
quote	Quotation character	Double quotation marks (" ")
escape	Escape character	Backslash (\)
multiLine	Whether the column data contains carriage return characters or transfer characters. The value **true** indicates yes and the value **false** indicates no.	false
dateFormat	Date format of the **date** field in a CSV file	yyyy-MM-dd

Parameter	Description	Default Value
timestampFormat	Date format of the **timestamp** field in a CSV file	yyyy-MM-dd HH:mm:ss
mode	Mode for parsing CSV files. The options are as follows: ● PERMISSIVE: Permissive mode. When an incorrect field is encountered, set the line to **Null**. ● DROPMALFORMED: When an incorrect field is encountered, the entire line is discarded. ● FAILFAST: Error mode. If an error occurs, it is automatically reported.	PERMISSIVE
header	Whether CSV contains header information. The value **true** indicates that the table header information is contained, and the value **false** indicates that the information is not included.	false
nullValue	Character that represents the null value. For example, **nullValue= "\\N"** indicates that **\N** represents the null value.	-
comment	Character that indicates the beginning of the comment. For example, **comment= '#'** indicates that the line starting with **#** is a comment.	-
compression	Data compression format. Currently, **gzip**, **bzip2**, and **deflate** are supported. If you do not want to compress data, enter **none**.	none
encoding	Data encoding format. Available values are **utf-8**, **gb2312**, and **gbk**. Value **utf-8** will be used if this parameter is left empty.	utf-8

Precautions

- The table and column names are case-insensitive.
- Descriptions of table names and column names support only string constants.
- During table creation, you need to specify the column name and corresponding data type. The data type is primitive type.
- If a folder and a file have the same name in the OBS directory, the file is preferred as the path when creating an OBS table.
- During table creation, if the specified path is an OBS directory and it contains subdirectories (or nested subdirectories), all file types and content in the subdirectories are considered table content. Ensure that all file types in the specified directory and its subdirectories are consistent with the storage

format specified in the table creation statement. All file content must be consistent with the fields in the table. Otherwise, errors will be reported in the query. You can set **multiLevelDirEnable** to **true** in the **OPTIONS** statement to query the content in the subdirectory. The default value is **false** (Note that this configuration item is a table attribute, exercise caution when performing this operation). Hive tables do not support this configuration item.

- The OBS storage path must be a directory on the OBS. The directory must be created in advance and be empty.
- When a partitioned table is created, the column specified in PARTITIONED BY must be a column in the table, and the partition type must be specified. The partition column supports only the **string**, **boolean**, **tinyint**, **smallint**, **short**, **int**, **bigint**, **long**, **decimal**, **float**, **double**, **date**, and **timestamp** type.
- When a partitioned table is created, the partition field must be the last one or several fields of the table field, and the sequence of the partition fields must be the same. Otherwise, an error occurs.
- A maximum of 7,000 partitions can be created in a single table.
- The CREATE TABLE AS statement cannot specify table attributes or create partitioned tables.

Example

- Create a **parquetTable** OBS table.

```
CREATE TABLE parquetTable (name string, id int) USING parquet OPTIONS (path "obs://bucketName/filePath");
```

- Create a **parquetZstdTable** OBS table and set the compression format to **zstd**.

```
CREATE TABLE parquetZstdTable (name string, id string) USING parquet OPTIONS (path "obs://bucketName/filePath",compression='zstd');
```

- Create a **student** table that has two fields **name** and **score**and partition the table by **classNo**.

```
CREATE TABLE IF NOT EXISTS student(name STRING, score DOUBLE, classNo INT) USING csv OPTIONS (PATH 'obs://bucketName/filePath') PARTITIONED BY (classNo);
```

NOTE

The **classNo** field is a partition field and must be placed at the end of the table field, that is, **student(name STRING, score DOUBLE, classNo INT)**.

- To create table **t1** and insert data of table **t2** into table **t1**, run the following statement:

```
CREATE TABLE t1 USING parquet OPTIONS(path 'obs://bucketName/tblPath') AS select * from t2;
```

1.4.2 Creating an OBS Table Using the Hive Syntax

Function

This statement is used to create an OBS table using the Hive syntax. The main differences between the DataSource and the Hive syntax lie in the supported data formats and the number of supported partitions. For details, see syntax and precautions.

Usage

- The size of the table will be calculated during creation.

- When data is added, the table size will not be changed to 0.
- You can view the table size on OBS.

Syntax

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name1 col_type1 [COMMENT col_comment1], ...)]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name2 col_type2, [COMMENT col_comment2], ...)]
  [ROW FORMAT row_format]
  [STORED AS file_format]
  LOCATION 'obs_path'
  [TBLPROPERTIES (key = value)]
  [AS select_statement];

row_format:
  : SERDE serde_cls [WITH SERDEPROPERTIES (key1=val1, key2=val2, ...)]
  | DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]]
      [COLLECTION ITEMS TERMINATED BY char]
      [MAP KEYS TERMINATED BY char]
      [LINES TERMINATED BY char]
      [NULL DEFINED AS char]
```

Keyword

- EXTERNAL: Creates an OBS table.
- IF NOT EXISTS: Prevents system errors when the created table exists.
- COMMENT: Field or table description.
- PARTITIONED BY: Partition field.
- ROW FORMAT: Row data format.
- STORED AS: Specifies the format of the file to be stored. Currently, only the TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, and PARQUET format are supported.
- LOCATION: Specifies the path of OBS. This keyword is mandatory when you create OBS tables.
- TBLPROPERTIES: Allows you to add the **key/value** properties to a table.

  For example, you can use this statement to enable the multiversion function to back up and restore table data. After the multiversion function is enabled, the system automatically backs up table data when you delete or modify the data using **insert overwrite** or **truncate**, and retains the data for a certain period. You can quickly restore data within the retention period. For details about SQL syntax related to the multiversion function, see **Enabling or Disabling Multiversion Backup** and **Backing Up and Restoring Data of Multiple Versions**.

  When creating an OBS table, you can use **TBLPROPERTIES ("dli.multi.version.enable"="true")** to enable multiversion. For details, see the following example.

Table 1-10 TBLPROPERTIES parameters

Key	Value
dli.multi.version.enable	• **true**: Enable the multiversion backup function. • **false**: Disable the multiversion backup function.

Key	Value
comment	Description of the table
orc.compress	An attribute of the ORC table, which specifies the compression mode of the ORC storage. Available values are as follows: ● **ZLIB** ● **SNAPPY** ● **NONE**
auto.purge	If this parameter is set to **true**, the deleted or overwritten data is removed and will not be dumped to the recycle bin.

- AS: You can run the CREATE TABLE AS statement to create a table.

Parameter

Table 1-11 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). The value cannot contain only digits and cannot start with a digit or underscore (_).
table_name	Table name of a database that contains letters, digits, and underscores (_). The value cannot contain only digits and cannot start with a digit or underscore (_). The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If special characters are required, use single quotation marks ('') to enclose them.
col_name	Field name
col_type	Field type
col_comment	Field description
row_format	Line data format
file_format	OBS table storage format. TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, and PARQUET are supported.
table_comment	Table description
obs_path	OBS path
key = value	Set table properties and values. For example, if you want to enable multiversion, you can set **"dli.multi.version.enable"="true"**.

Parameter	Description
select_statement	The CREATE TABLE AS statement is used to insert the SELECT query result of the source table or a data record to a new table in OBS bucket.

Precautions

- The table and column names are case-insensitive.
- Descriptions of table names and column names support only string constants.
- During table creation, you need to specify the column name and corresponding data type. The data type is primitive type.
- If a folder and a file have the same name in the OBS directory, the file is preferred as the path when creating an OBS table.
- When you create a partitioned table, ensure that the specified column in **PARTITIONED BY** is not a column in the table and the data type is specified. The partition column supports only the open-source Hive table types including **string**, **boolean**, **tinyint**, **smallint**, **short**, **int**, **bigint**, **long**, **decimal**, **float**, **double**, **date**, and **timestamp**.
- Multiple partition fields can be specified. The partition fields need to be specified after the **PARTITIONED BY** keyword, instead of the table name. Otherwise, an error occurs.
- A maximum of 100,000 partitions can be created in a single table.
- The CREATE TABLE AS statement cannot specify table attributes or create partitioned tables.

Example

- To create a Parquet table named **student**, in which the **id**, **name**, and **score** fields are contained and the data types of the respective fields are INT, STRING, and FLOAT, run the following statement:
```
CREATE TABLE student (id INT, name STRING, score FLOAT) STORED AS PARQUET LOCATION 'obs://bucketName/filePath';
```
- To create a table named **student**, for which **classNo** is the partition field and two fields **name** and **score** are specified, run the following statement:
```
CREATE TABLE IF NOT EXISTS student(name STRING, score DOUBLE) PARTITIONED BY (classNo INT) STORED AS PARQUET LOCATION 'obs://bucketName/filePath';
```

  NOTE

  classNo is a partition field and must be specified after the PARTITIONED BY keyword, that is, **PARTITIONED BY (classNo INT)**. It cannot be specified after the table name as a table field.
- To create table **t1** and insert data of table **t2** into table **t1** by using the Hive syntax, run the following statement:
```
CREATE TABLE t1 STORED AS parquet LOCATION 'obs://bucketName/filePath' as select * from t2;
```
- Create the **student** table and enable multiversion by using the Hive syntax.
```
CREATE TABLE student (id INT, name STRING, score FLOAT) STORED AS PARQUET LOCATION 'obs://bucketName/filePath' TBLPROPERTIES ("dli.multi.version.enable"="true");
```


1.5 Creating a DLI Table

1.5.1 Creating a DLI Table Using the DataSource Syntax

Function

This DataSource syntax can be used to create a DLI table. The main differences between the DataSource and the Hive syntax lie in the supported data formats and the number of supported partitions. For details, see syntax and precautions.

Syntax

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name1 col_type1 [COMMENT col_comment1], ...)]
  USING file_format
  [OPTIONS (key1=val1, key2=val2, ...)]
  [PARTITIONED BY (col_name1, col_name2, ...)]
  [COMMENT table_comment]
  [AS select_statement];
```

Keyword

- IF NOT EXISTS: Prevents system errors when the created table exists.
- USING: Specifies the storage format.
- OPTIONS: Specifies the attribute name and attribute value when a table is created.
- COMMENT: Field or table description.
- PARTITIONED BY: Partition field.
- AS: Run the CREATE TABLE AS statement to create a table.

Parameter Description

Table 1-12 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). The value cannot contain only digits and cannot start with a digit or underscore (_).
table_name	Table name of a database that contains letters, digits, and underscores (_). The value cannot contain only digits and cannot start with a digit or underscore (_). The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If special characters are required, use single quotation marks ('') to enclose them.
col_name	Column names with data types separated by commas (,). The column name contains letters, digits, and underscores (_). It cannot contain only digits and must contain at least one letter.
col_type	Field type

Parameter	Description
col_comment	Field description
file_format	Data storage format of DLI tables. The value can be **parquet** only.
table_comment	Table description
select_statement	The CREATE TABLE AS statement is used to insert the SELECT query result of the source table or a data record to a newly created DLI table.

Table 1-13 OPTIONS parameter description

Parameter	Description	Default Value
multiLevelDirEnable	Whether to iteratively query data in subdirectories. When this parameter is set to **true**, all files in the table path, including files in subdirectories, are iteratively read when a table is queried.	false
compression	Specified compression format. Generally, you need to set this parameter to **zstd** for parquet files.	-

Precautions

- If no delimiter is specified, the comma (,) is used by default.
- When a partitioned table is created, the column specified in PARTITIONED BY must be a column in the table, and the partition type must be specified. The partition column supports only the **string**, **boolean**, **tinyint**, **smallint**, **short**, **int**, **bigint**, **long**, **decimal**, **float**, **double**, **date**, and **timestamp** type.
- When a partitioned table is created, the partition field must be the last one or several fields of the table field, and the sequence of the partition fields must be the same. Otherwise, an error occurs.
- A maximum of 7,000 partitions can be created in a single table.
- The CREATE TABLE AS statement cannot specify table attributes or create partitioned tables.

Example

- Create a **src** table that has two columns **key** and **value** in INT and STRING types respectively, and set the compression format to **zstd**.
```
CREATE TABLE src(key INT, value STRING) USING PARQUET OPTIONS(compression = 'zstd');
```
- Create a **student** table that has **name**, **score**, and **classNo** columns and stores data in **Parquet** format. Partition the table by **classNo**.


```
CREATE TABLE student(name STRING, score INT, classNo INT) USING PARQUET OPTIONS('key1' =
'value1') PARTITIONED BY(classNo) ;
```

NOTE

classNo is the partition field, which must be placed at the end of the table field, that is, **student(name STRING, score INT, classNo INT)**.

- Create table **t1** and insert **t2** data into table **t1**.

```
CREATE TABLE t1 USING parquet AS select * from t2;
```

1.5.2 Creating a DLI Table Using the Hive Syntax

Function

This Hive syntax is used to create a DLI table. The main differences between the DataSource and the Hive syntax lie in the supported data formats and the number of supported partitions. For details, see syntax and precautions.

Syntax

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name1 col_type1 [COMMENT col_comment1], ...)]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name2 col_type2, [COMMENT col_comment2], ...)]
  [ROW FORMAT row_format]
  STORED AS file_format
  [TBLPROPERTIES (key1=val1, key2=val2, ...)]
  [AS select_statement];

row_format:
  : SERDE serde_cls [WITH SERDEPROPERTIES (key1=val1, key2=val2, ...)]
  | DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]]
      [COLLECTION ITEMS TERMINATED BY char]
      [MAP KEYS TERMINATED BY char]
      [LINES TERMINATED BY char]
      [NULL DEFINED AS char]
```

Keyword

- IF NOT EXISTS: Prevents system errors when the created table exists.
- COMMENT: Field or table description.
- PARTITIONED BY: Partition field.
- ROW FORMAT: Row data format.
- STORED AS: Specifies the format of the file to be stored. Currently, only the TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, and PARQUET format are supported. This keyword is mandatory when you create DLI tables.
- TBLPROPERTIES: The TBLPROPERTIES clause allows you to add the **key/value** attribute to a table.

  For example, if the table storage format is Parquet, you can use **TBLPROPERTIES(parquet.compression = 'zstd')** to set the table compression format to **zstd**.
- AS: Run the CREATE TABLE AS statement to create a table.

Parameter Description

Table 1-14 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). The value cannot contain only digits and cannot start with a digit or underscore (_).
table_name	Table name of a database that contains letters, digits, and underscores (_). The value cannot contain only digits and cannot start with a digit or underscore (_). The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If special characters are required, use single quotation marks ('') to enclose them.
col_name	Column names with data types separated by commas (,). The column name contains letters, digits, and underscores (_). It cannot contain only digits and must contain at least one letter.
col_type	Field type
col_comment	Field description
row_format	Line data format
file_format	Data storage format: TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, PARQUET.
table_comment	Table description
select_stateme nt	The CREATE TABLE AS statement is used to insert the SELECT query result of the source table or a data record to a newly created DLI table.

Precautions

- When you create a partitioned table, ensure that the specified column in **PARTITIONED BY** is not a column in the table and the data type is specified. The partition column supports only the open-source Hive table types including **string**, **boolean**, **tinyint**, **smallint**, **short**, **int**, **bigint**, **long**, **decimal**, **float**, **double**, **date**, and **timestamp**.
- Multiple partition fields can be specified. The partition fields need to be specified after the **PARTITIONED BY** keyword, instead of the table name. Otherwise, an error occurs.
- A maximum of 100,000 partitions can be created in a single table.
- The CREATE TABLE AS statement cannot specify table attributes or create partitioned tables.

Example

- Create a **src** table that has **key** and **value** columns in INT and STRING types respectively, and specify a property as required.

```
CREATE TABLE src
  (key INT, value STRING)
  STORED AS PARQUET
  TBLPROPERTIES('key1' = 'value1');
```

- Create a **student** table that has **name**, **score**, and **classNo** columns, and partition the table by **classNo**.
```
CREATE TABLE student
  (name STRING, score INT)
  STORED AS PARQUET
  TBLPROPERTIES(parquet.compression = 'zstd') PARTITIONED BY(classNo INT);
```
- Create table **t1** and insert **t2** data into table **t1**.
```
CREATE TABLE t1
  STORED AS PARQUET
  AS select * from t2;
```

1.6 Deleting a Table

Function

This statement is used to delete tables.

Syntax

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

Keyword

- If the table is stored in OBS, only the metadata is deleted. The data stored on OBS is not deleted.

Parameters

Table 1-15 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
table_name	Table name

Precautions

The to-be-deleted table must exist in the current database. Otherwise, an error is reported. To avoid this error, add **IF EXISTS** in this statement.

Example

1. Create an OBS or DLI table, for example, **student**. For details, see **Creating an OBS Table** or **Creating a DLI Table**.

2. Run the following statement to delete a table named **student** in the current database:
```
DROP TABLE IF EXISTS student;
```

1.7 Viewing Tables

1.7.1 Viewing All Tables

Function

This statement is used to query and show information about all tables and views in the current database.

Syntax

```
SHOW TABLES [IN | FROM db_name] [LIKE regex_expression];
```

Keyword

FROM/IN: followed by the name of a database whose tables and views will be displayed.

Parameters

Table 1-16 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
regex_expression	Name of a database table.

Precautions

None

Example

1. Create an OBS or DLI table. For details, see **Creating an OBS Table** or **Creating a DLI Table**.
2. To show all tables and views in the current database, run the following statement:
```
SHOW TABLES;
```
3. To show all tables started with **test** in the **testdb** database, run the following statement:
```
SHOW TABLES IN testdb LIKE "test*";
```


1.7.2 Viewing Table Creation Statements

Function

This statement is used to show the statements for creating a table.

Syntax

```
SHOW CREATE TABLE table_name;
```

Keyword

CREATE TABLE: statement for creating a table

Parameters

Table 1-17 Parameter description

Parameter	Description
table_name	Table name

Precautions

The table specified in this statement must exist. Otherwise, an error will occur.

Example

1. Create an OBS or DLI table, for example, **test**. For details, see **Creating an OBS Table** or **Creating a DLI Table**.
2. Run the following statement to view the statement that is used to create table **test**:
   ```
   SHOW CREATE TABLE test;
   ```

1.7.3 Viewing Table Properties

Function

Check the properties of a table.

Syntax

```
SHOW TBLPROPERTIES table_name [('property_name')];
```

Keyword

TBLPROPERTIES: This statement allows you to add a **key/value** property to a table.

Parameters

Table 1-18 Parameter description

Parameter	Description
table_name	Table name
property_name	• If this parameter is not specified, all properties and their values are returned. • If a property name is specified, only the specified property and its value are returned.

Precautions

property_name is case sensitive. You cannot specify multiple **property_name** attributes at the same time. Otherwise, an error occurs.

Example

To return the value of **property_key1** in the test table, run the following statement:

```
SHOW TBLPROPERTIES test ('property_key1');
```

1.7.4 Viewing All Columns in a Specified Table

Function

This statement is used to query all columns in a specified table.

Syntax

```
SHOW COLUMNS {FROM | IN} table_name [{FROM | IN} db_name];
```

Keyword

- COLUMNS: columns in the current table
- FROM/IN: followed by the name of a database whose tables and views will be displayed. Keyword FROM is equivalent to IN. You can use either of them in a statement.

Parameters

Table 1-19 Parameter description

Parameter	Description
table_name	Table name
db_name	Database name

Precautions

The specified table must exist in the database. If the table does not exist, an error is reported.

Example

Run the following statement to view all columns in the **student** table.

```
SHOW COLUMNS IN student;
```

1.7.5 Viewing All Partitions in a Specified Table

Function

This statement is used to view all partitions in a specified table.

Syntax

```
SHOW PARTITIONS [db_name.]table_name
  [PARTITION partition_specs];
```

Keyword

- PARTITIONS: partitions in a specified table
- PARTITION: a specified partition

Parameters

Table 1-20 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). It cannot contain only digits and cannot start with an underscore (_).

Parameter	Description
table_name	Table name of a database that contains letters, digits, and underscores (_). It cannot contain only digits and cannot start with an underscore (_). The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If special characters are required, use single quotation marks ('') to enclose them.
partition_specs	Partition information, in the format of "key=value", where **key** indicates the partition field and **value** indicates the partition value. If a partition field contains multiple fields, the system displays all partition information that matches the partition field.

Precautions

The table specified in this statement must exist and must be a partitioned table. Otherwise, an error is reported.

Example

- To show all partitions in the **student** table, run the following statement:
  ```
  SHOW PARTITIONS student;
  ```
- Check the **dt='2010-10-10'** partition in the **student** table, run the following statement:
  ```
  SHOW PARTITIONS student PARTITION(dt='2010-10-10')
  ```

1.7.6 Viewing Table Statistics

Function

This statement is used to view the table statistics. The names and data types of all columns in a specified table will be returned.

Syntax

```
DESCRIBE [EXTENDED|FORMATTED] [db_name.]table_name;
```

Keyword

- EXTENDED: displays all metadata of the specified table. It is used during debugging in general.
- FORMATTED: displays all metadata of the specified table in a form.

Parameters

Table 1-21 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_).
table_name	Table name of a database that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_). The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If special characters are required, use single quotation marks ('') to enclose them.

Precautions

The to-be-queried table must exist. If this statement is used to query the information about a table that does not exist, an error is reported.

Example

To query the names and data types of all columns in the **student** table, run the following statement:

```
DESCRIBE student;
```

1.8 Modifying a Table

1.8.1 Adding a Column

Function

This statement is used to add one or more new columns to a table.

Syntax

```
ALTER TABLE [db_name.]table_name ADD COLUMNS (col_name1 col_type1 [COMMENT
col_comment1], ...);
```

Keyword

- ADD COLUMNS: columns to add
- COMMENT: column description

Parameters

Table 1-22 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_).
table_name	Table name
col_name	Column name
col_type	Field type
col_comment	Column description

Precautions

None

Example

```
ALTER TABLE t1 ADD COLUMNS (column2 int, column3 string);
```

1.8.2 Enabling or Disabling Multiversion Backup

Function

DLI controls multiple versions of backup data for restoration. After the multiversion function is enabled, the system automatically backs up table data when you delete or modify the data using **insert overwrite** or **truncate**, and retains the data for a certain period. You can quickly restore data within the retention period. For details about the syntax related to the multiversion function, see **Backing Up and Restoring Data of Multiple Versions**.

Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.

Syntax

- Enable the multiversion function.
```
ALTER TABLE [db_name.]table_name
SET TBLPROPERTIES ("dli.multi.version.enable"="true");
```
- Disable the multiversion function.
```
ALTER TABLE [db_name.]table_name
UNSET TBLPROPERTIES ("dli.multi.version.enable");
```
After multiversion is enabled, data of different versions is automatically stored in the OBS storage directory when **insert overwrite** or **truncate** is executed. After multiversion is disabled, run the following statement to restore the multiversion backup data directory:

```
RESTORE TABLE [db_name.]table_name TO initial layout;
```

Keyword

- SET TBLPROPERTIES: Used to set table properties and enable multiversion.
- UNSET TBLPROPERTIES: Used to unset table properties and disable multiversion.

Parameter

Table 1-23 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_).
table_name	Table name

Precautions

Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.

Example

- Modify the **test_table** table to enable multiversion.
```
ALTER TABLE test_table
SET TBLPROPERTIES ("dli.multi.version.enable"="true");
```
- Modify the **test_table** table to disable multiversion.
```
ALTER TABLE test_table
UNSET TBLPROPERTIES ("dli.multi.version.enable");
```
Restore the multiversion backup data directory.
```
RESTORE TABLE test_table TO initial layout;
```

1.9 Syntax for Partitioning a Table

1.9.1 Adding Partition Data (Only OBS Tables Supported)

Function

After an OBS partitioned table is created, no partition information is generated for the table. Partition information is generated only after you:

- Insert data to the OBS partitioned table. After the data is inserted successfully, the partition metadata can be queried, for example, by partition columns.
- Copy the partition directory and data into the OBS path of the partitioned table, and run the partition adding statements described in this section to

generate partition metadata. Then you can perform operations such as table query by partition columns.

The following describes how to use the **ALTER TABLE** statement to add a partition.

Syntax

```
ALTER TABLE table_name ADD [IF NOT EXISTS]
  PARTITION partition_specs1
  [LOCATION 'obs_path1']
  PARTITION partition_specs2
  [LOCATION 'obs_path2'];
```

Keyword

- IF NOT EXISTS: prevents errors when partitions are repeatedly added.
- PARTITION: specifies a partition.
- LOCATION: specifies the partition path.

Parameters

Table 1-24 Parameter description

Parameter	Description
table_name	Table name
partition_specs	Partition fields
obs_path	OBS path

Precautions

- When you add a partition to a table, the table and the partition column (specified by PARTITIONED BY during table creation) must exist, and the partition to be added cannot be added repeatedly. Otherwise, an error is reported. You can use **IF NOT EXISTS** to avoid errors if the partition does not exist.
- If tables are partitioned by multiple fields, you need to specify all partitioning fields in any sequence when adding partitions.
- By default, parameters in **partition_specs** contain parentheses (). For example: **PARTITION (dt='2009-09-09',city='xxx')**.
- If you need to specify an OBS path when adding a partition, the OBS path must exist. Otherwise, an error occurs.
- To add multiple partitions, you need to use spaces to separate each set of **LOCATION 'obs_path'** in the **PARTITION partition_specs**. The following is an example:

  PARTITION partition_specs LOCATION 'obs_path' PARTITION partition_specs LOCATION 'obs_path'

- If the path specified in the new partition contains subdirectories (or nested subdirectories), all file types and content in the subdirectories are considered partition records. Ensure that all file types and file content in the partition directory are the same as those in the table. Otherwise, an error is reported.

Example

- The following example shows you how to add partition data when the OBS table is partitioned by a single column.
  a. Use the DataSource syntax to create an OBS table, and partition the table by column **external_data**. The partition data is stored in **obs://bucketName/datapath**.
  ```
  create table testobstable(id varchar(128), external_data varchar(16)) using JSON OPTIONS
  (path 'obs://bucketName/datapath') PARTITIONED by (external_data);
  ```
  b. Copy the partition directory to **obs://bucketName/datapath**. In this example, copy all files in the partition column **external_data=22** to **obs://bucketName/datapath**.
  c. Run the following command to add partition data:
  ```
  ALTER TABLE testobstable ADD
    PARTITION (external_data='22')
    LOCATION 'obs://bucketName/datapath/external_data=22';
  ```
  d. After the partition data is added successfully, you can perform operations such as data query based on the partition column.
  ```
  select * from testobstable where external_data='22';
  ```
- The following example shows you how to add partition data when the OBS table is partitioned by multiple columns.
  a. Use the DataSource syntax to create an OBS table, and partition the table by columns **external_data** and **dt**. The partition data is stored in **obs://bucketName/datapath**.
  ```
  create table testobstable(
    id varchar(128),
    external_data varchar(16),
    dt varchar(16)
  ) using JSON OPTIONS (path 'obs://bucketName/datapath') PARTITIONED by (external_data,
  dt);
  ```
  b. Copy the partition directories to **obs://bucketName/datapath**. In this example, copy files in **external_data=22** and its subdirectory **dt=2021-07-27** to **obs://bucketName/datapath**.
  c. Run the following command to add partition data:
  ```
  ALTER TABLE
    testobstable
  ADD
    PARTITION (external_data = '22', dt = '2021-07-27') LOCATION 'obs://bucketName/datapath/
  external_data=22/dt=2021-07-27';
  ```
  d. After the partition data is added successfully, you can perform operations such as data query based on the partition columns.
  ```
  select * from testobstable where external_data = '22';
  select * from testobstable where external_data = '22' and dt='2021-07-27';
  ```

1.9.2 Renaming a Partition (Only OBS Tables Supported)

Function

This statement is used to rename partitions.

Syntax

```
ALTER TABLE table_name
  PARTITION partition_specs
  RENAME TO PARTITION partition_specs;
```

Keyword

- PARTITION: a specified partition
- RENAME: new name of the partition

Parameters

Table 1-25 Parameter description

Parameter	Description
table_name	Table name
partition_specs	Partition fields

Precautions

- **This statement is used for OBS table operations.**
- The table and partition to be renamed must exist. Otherwise, an error occurs. The name of the new partition must be unique. Otherwise, an error occurs.
- If a table is partitioned using multiple fields, you are required to specify all the fields of a partition (at random order) when renaming the partition.
- By default, the **partition_specs** parameter contains **()**. For example: **PARTITION (dt='2009-09-09',city='xxx')**

Example

To modify the name of the **city='xxx',dt='2008-08-08'** partition in the **student** table to **city='xxx',dt='2009-09-09'**, run the following statement:

```
ALTER TABLE student
  PARTITION (city='xxx',dt='2008-08-08')
  RENAME TO PARTITION (city='xxx',dt='2009-09-09');
```

1.9.3 Deleting a Partition

Function

Deletes one or more partitions from a partitioned table.

Precautions

- The table in which partitions are to be deleted must exist. Otherwise, an error is reported.
- The to-be-deleted partition must exist. Otherwise, an error is reported. To avoid this error, add **IF EXISTS** in this statement.

Syntax

```
ALTER TABLE [db_name.]table_name
  DROP [IF EXISTS]
  PARTITION partition_spec1[,PARTITION partition_spec2,...];
```

Keyword

- DROP: deletes a partition.
- IF EXISTS: The partition to be deleted must exist. Otherwise, an error is reported.
- PARTITION: specifies the partition to be deleted

Parameters

Table 1-26 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). It cannot contain only digits and cannot start with an underscore (_).
table_name	Table name of a database that contains letters, digits, and underscores (_). It cannot contain only digits and cannot start with an underscore (_). The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If special characters are required, use single quotation marks ('') to enclose them.
partition_specs	Partition information, in the format of "key=value", where **key** indicates the partition field and **value** indicates the partition value. In a table partitioned using multiple fields, if you specify all the fields of a partition name, only the partition is deleted; if you specify only some fields of a partition name, all matching partitions will be deleted. By default, the **partition_specs** parameter contains **()**. For example: **PARTITION (dt='2009-09-09',city='xxx')**

Example

To delete the **dt = '2008-08-08', city = 'xxx'** partition in the **student** table, run the following statement:

```
ALTER TABLE student
  DROP
  PARTITION (dt = '2008-08-08', city = 'xxx');
```

1.9.4 Deleting Partitions by Specifying Filter Criteria (Only OBS Tables Supported)

Function

This statement is used to delete one or more partitions based on specified conditions.

Precautions

- **This statement is used for OBS table operations only.**
- The table in which partitions are to be deleted must exist. Otherwise, an error is reported.
- The to-be-deleted partition must exist. Otherwise, an error is reported. To avoid this error, add **IF EXISTS** in this statement.

Syntax

```
ALTER TABLE [db_name.]table_name
  DROP [IF EXISTS]
  PARTITIONS partition_filtercondition;
```

Keyword

- DROP: deletes specified partitions.
- IF EXISTS: Partitions to be deleted must exist. Otherwise, an error is reported.
- PARTITIONS: specifies partitions meeting the conditions

Parameters

Table 1-27 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_).
table_name	Table name of a database that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_). The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If special characters are required, use single quotation marks ('') to enclose them. **This statement is used for OBS table operations.**

Parameter	Description
partition_filter condition	Condition used to search partitions to be deleted. The format is as follows: ● *Partition column name* **Operator** *Value to compare* Example: start_date < '201911' ● <partition_filtercondition1> AND|OR <partition_filtercondition2> Example: start_date < '201911' OR start_date >= '202006' ● (<partition_filtercondition1>) [,partitions (<partition_filtercondition2>), ...] Example: (start_date <> '202007'), partitions(start_date < '201912')

Example

You can run the following statements to delete the **dt** partition of the **student** table using different conditions:

```
alter table student drop partitions(start_date < '201911');
alter table student drop partitions(start_date >= '202007');
alter table student drop partitions(start_date BETWEEN '202001' AND '202007');
alter table student drop partitions(start_date < '201912' OR start_date >= '202006');
alter table student drop partitions(start_date > '201912' AND start_date <= '202004');
alter table student drop partitions(start_date != '202007');
alter table student drop partitions(start_date <> '202007');
alter table student drop partitions(start_date <> '202007'), partitions(start_date < '201912');
```

1.9.5 Altering the Partition Location of a Table (Only OBS Tables Supported)

Function

This statement is used to modify the positions of table partitions.

Syntax

```
ALTER TABLE table_name
  PARTITION partition_specs
  SET LOCATION obs_path;
```

Keyword

- PARTITION: a specified partition
- LOCATION: path of the partition

Parameters

Table 1-28 Parameter description

Parameter	Description
table_name	Table name
partition_specs	Partition fields
obs_path	OBS path

Precautions

- For a table partition whose position is to be modified, the table and partition must exist. Otherwise, an error is reported.
- By default, the **partition_specs** parameter contains **()**. For example: **PARTITION (dt='2009-09-09',city='xxx')**
- The specified OBS path must be an absolute path. Otherwise, an error is reported.
- If the path specified in the new partition contains subdirectories (or nested subdirectories), all file types and content in the subdirectories are considered partition records. Ensure that all file types and file content in the partition directory are the same as those in the table. Otherwise, an error is reported.

Example

To set the OBS path of partition **dt='2008-08-08',city='xxx'** in table **student** to **obs://bucketName/fileName/student/dt=2008-08-08/city=xxx**, run the following statement:

```
ALTER TABLE student
  PARTITION(dt='2008-08-08',city='xxx')
  SET LOCATION 'obs://bucketName/fileName/student/dt=2008-08-08/city=xxx';
```

1.9.6 Updating Partitioned Table Data (Only OBS Tables Supported)

Function

This statement is used to update the partition information about a table in the Metastore.

Syntax

```
MSCK REPAIR TABLE table_name;
```

Or

```
ALTER TABLE table_name RECOVER PARTITIONS;
```


Keyword

- PARTITIONS: partition information
- SERDEPROPERTIES: Serde attribute

Parameters

Table 1-29 Parameter description

Parameter	Description
table_name	Table name
partition_specs	Partition fields
obs_path	OBS path

Precautions

- This statement is applied only to partitioned tables. After you manually add partition directories to OBS, run this statement to update the newly added partition information in the metastore. The **SHOW PARTITIONS table_name** statement can be used to query the newly-added partitions.
- The partition directory name must be in the specified format, that is, **tablepath/partition_column_name=partition_column_value**.

Example

Run the following statements to update the partition information about table **ptable** in the Metastore:

```
MSCK REPAIR TABLE ptable;
```

Or

```
ALTER TABLE ptable RECOVER PARTITIONS;
```

1.9.7 Updating Table Metadata with REFRESH TABLE

Function

Spark caches Parquet metadata to improve performance. If you update a Parquet table, the cached metadata is not updated. Spark SQL cannot find the newly inserted data and an error similar with the following is reported:

```
DLI.0002: FileNotFoundException: getFileStatus on  error message
```

You can use REFRESH TABLE to solve this problem. REFRESH TABLE reorganizes files of a partition and reuses the original table metadata information to detect the increase or decrease of table fields. This statement is mainly used when the metadata in a table is not modified but the table data is modified.

Syntax

```
REFRESH TABLE [db_name.]table_name;
```

Keyword

None

Parameter

Table 1-30 Parameter description

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_).
table_name	Table name of a database that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_). The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If special characters are required, use single quotation marks ('') to enclose them.

Precautions

None

Example

Update metadata of the **test** table.

```
REFRESH TABLE test;
```

1.10 Importing Data to the Table

Function

The **LOAD DATA** function can be used to import data in **CSV**, **Parquet**, **ORC**, **JSON**, and **Avro** formats. The data is converted into the **Parquet** data format for storage.

Syntax

```
LOAD DATA INPATH 'folder_path' INTO TABLE [db_name.]table_name
  OPTIONS(property_name=property_value, ...);
```

Keyword

- INPATH: path of data to be imported
- OPTIONS: list of properties

Parameter

Table 1-31 Parameter description

Parameter	Description
folder_path	OBS path of the file or folder used for storing the raw data.
db_name	Enter the database name. If this parameter is not specified, the current database is used.
table_name	Name of the table to which data is to be imported.

The following configuration options can be used during data import:

- DATA_TYPE: specifies the type of data to be imported. Currently, **CSV**, **Parquet**, **ORC**, **JSON**, and **Avro** are supported. The default value is **CSV**.

  The configuration item is **OPTIONS** ('DATA_TYPE' = 'CSV').

  When importing a **CSV** file or a **JSON** file, you can select one of the following modes:

  - **PERMISSIVE**: When the **PERMISSIVE** mode is selected, the data of a column is set to **null** if its data type does not match that of the target table column.
  - **DROPMALFORMED**: When the **DROPMALFORMED** mode is selected, the data of a column s not imported if its data type does not match that of the target table column.
  - **FAILFAST**: When the **FAILFAST** mode is selected, exceptions might occur and the import may fail if a column type does not match.

  You can set the mode by adding **OPTIONS ('MODE' = 'PERMISSIVE')** to the **OPTIONS** parameter.

- **DELIMITER**: You can specify a separator in the import statement. The default value is **,**.

  The configuration item is **OPTIONS('DELIMITER'=',')**.

  For CSV data, the following delimiters are supported:

  - Tab character, for example, **'DELIMITER'='\t'**.
  - Any binary character, for example, **'DELIMITER'='\u0001(^A)'**.
  - Single quotation mark ('). A single quotation mark must be enclosed in double quotation marks (" "). For example, **'DELIMITER'= "'"**.

- **QUOTECHAR**: You can specify quotation marks in the import statement. The default value is double quotation marks (**"**).

  The configuration item is **OPTIONS('QUOTECHAR'='"')**.

- **COMMENTCHAR**: You can specify the comment character in the import statement. During the import operation, if a comment character is at the beginning of a row, the row is considered as a comment and will not be imported. The default value is a pound key (#).

  The configuration item is *OPTIONS('COMMENTCHAR'='#')*.

- **HEADER**: Indicates whether the source file contains a header. Possible values can be **true** and **false**. **true** indicates that the source file contains a header, and **false** indicates that the source file does not contain a header. The default value is **false**. If no header exists, specify the **FILEHEADER** parameter in the **LOAD DATA** statement to add a header.

  The configuration item is *OPTIONS('HEADER'='true')*.
- **FILEHEADER**: If the source file does not contain any header, add a header to the **LOAD DATA** statement.

  OPTIONS('FILEHEADER'='column1,column2')
- **ESCAPECHAR**: Is used to perform strict verification of the escape character on CSV files. The default value is a slash (\\).

  The configuration item is OPTIONS. (ESCAPECHAR?=?\\?)

  NOTE

  Enter **ESCAPECHAR** in the CSV data. **ESCAPECHAR** must be enclosed in double quotation marks (" "). For example, "a\b".
- **MAXCOLUMNS**: This parameter is optional and specifies the maximum number of columns parsed by a CSV parser in a line.

  The configuration item is *OPTIONS('MAXCOLUMNS'='400')*.

Table 1-32 MAXCOLUMNS

Name of the Optional Parameter	Default Value	Maximum Value
MAXCOLUMNS	2000	20000

NOTE

After the value of **MAXCOLUMNS Option** is set, data import will require the memory of **executor**. As a result, data may fail to be imported due to insufficient **executor** memory.

- **DATEFORMAT**: Specifies the date format of a column.

  OPTIONS('DATEFORMAT'='dateFormat')

  NOTE

  - The default value is yyyy-MM-dd.
  - The date format is specified by the date mode string of **Java**. For the Java strings describing date and time pattern, characters **A** to **Z** and **a** to **z** without single quotation marks (') are interpreted as pattern characters , which are used to represent date or time string elements. If the pattern character is quoted by single quotation marks ('), text matching rather than parsing is performed. For the definition of pattern characters in Java, see **Table 1-33**.

Table 1-33 Definition of characters involved in the date and time patterns

Character	Date or Time Element	Example
G	Epoch ID	AD
y	Year	1996; 96
M	Month	July; Jul; 07
w	Number of the week in a year	27 (the twenty-seventh week of the year)
W	Number of the week in a month	2 (the second week of the month)
D	Number of the day in a year	189 (the 189th day of the year)
d	Number of the day in a month	10 (the tenth day of the month)
u	Number of the day in a week	1 (Monday), ..., 7 (Sunday)
a	am/pm flag	pm (12:00-24:00)
H	Hour time (0-23)	2
h	Hour time (1-12)	12
m	Number of minutes	30
s	Number of seconds	55
S	Number of milliseconds	978
z	Time zone	Pacific Standard Time; PST; GMT-08:00

- **TIMESTAMPFORMAT**: Specifies the timestamp format of a column.
  OPTIONS('TIMESTAMPFORMAT'='timestampFormat')

  NOTE

  - Default value: yyyy-MM-dd HH:mm:ss.
  - The timestamp format is specified by the Java time pattern character string. For details, see **Table 3 Definition of date and time pattern characters**.

- **Mode**: Specifies the processing mode of error records while importing. The options are as follows: **PERMISSIVE**, **DROPMALFORMED**, and **FAILFAST**.
  OPTIONS('MODE'='permissive')

NOTE

- **PERMISSIVE (default)**: Parse bad records as much as possible. If a field cannot be converted, the entire row is null.
- **DROPMALFORMED**: Ignore the **bad records** that cannot be parsed.
- **FAILFAST**: If a record cannot be parsed, an exception is thrown and the job fails.

- **BADRECORDSPATH**: Specifies the directory for storing error records during the import.

  OPTIONS('BADRECORDSPATH'='obs://bucket/path')

  NOTE

  It is recommended that this option be used together with the **DROPMALFORMED** pattern to import the records that can be successfully converted into the target table and store the records that fail to be converted to the specified error record storage directory.

Precautions

- When importing or creating an OBS table, you must specify a folder as the directory. If a file is specified, data import may be failed.
- Only the raw data stored in the OBS path can be imported.
- You are advised not to concurrently import data in to a table. If you concurrently import data into a table, there is a possibility that conflicts occur, leading to failed data import.
- Only one path can be specified during data import. The path cannot contain commas (,).
- If a folder and a file with the same name exist in the OBS bucket directory, the data is preferentially to be imported directed to the file rather than the folder.
- When importing data of the PARQUET, ORC, or JSON format, you must specify *DATA_TYPE*. Otherwise, the data is parsed into the default format **CSV**. In this case, the format of the imported data is incorrect.
- If the data to be imported is in the CSV or JSON format and contains the date and columns, you need to specify *DATEFORMAT* and *TIMESTAMPFORMAT*. Otherwise, the data will be parsed into the default date and timestamp formats.

Example

NOTE

Before importing data, you must create an OBS table or DLI table. For details, see **Creating an OBS Table** or **Creating a DLI Table**.

- To import a CSV file to a table named **t**, run the following statement:
```
LOAD DATA INPATH 'obs://dli/data.csv' INTO TABLE t
  OPTIONS('DELIMITER'=',' , 'QUOTECHAR'='"','COMMENTCHAR'='#','HEADER'='false');
```
- To import a JSON file to a table named **jsontb**, run the following statement:
```
LOAD DATA INPATH 'obs://dli/alltype.json' into table jsontb
  OPTIONS('DATA_TYPE'='json','DATEFORMAT'='yyyy/MM/dd','TIMESTAMPFORMAT'='yyyy/MM/dd
HH:mm:ss');
```

1.11 Inserting Data

Function

This statement is used to insert the SELECT query result or a certain data record into a table.

Syntax

- Insert the SELECT query result into a table.
```
INSERT INTO [TABLE] [db_name.]table_name
  [PARTITION part_spec] select_statement;
INSERT OVERWRITE TABLE [db_name.]table_name
  [PARTITION part_spec] select_statement;
part_spec:
  : (part_col_name1=val1 [, part_col_name2=val2, ...])
```
- Insert a data record into a table.
```
INSERT INTO [TABLE] [db_name.]table_name
  [PARTITION part_spec] VALUES values_row [, values_row ...];
INSERT OVERWRITE TABLE [db_name.]table_name
  [PARTITION part_spec] VALUES values_row [, values_row ...];
values_row:
  : (val1 [, val2, ...])
```

Keyword

Table 1-34 INSERT parameter description

Parameter	Description
db_name	Name of the database where the target table resides.
table_name	Name of the target table.
part_spec	Detailed partition information. If there are multiple partition fields, all fields must be contained, but the corresponding values are optional. The system matches the corresponding partition. A maximum of 100,000 partitions can be created in a single table.
select_statement	SELECT query on the source table.
values_row	Value to be inserted to a table. Use commas (,) to separate columns.

Precautions

- The target DLI table must exist.
- If no partition needs to be specified for dynamic partitioning, place **part_spec** in the SELECT statement as a common field.
- During creation of the target OBS table, only the folder path can be specified.
- The source table and the target table must have the same data types and column field quantity. Otherwise, data insertion fails.

- You are advised not to concurrently insert data into a table. If you concurrently insert data into a table, there is a possibility that conflicts occur, leading to failed data insertion.
- The **INSERT INTO** statement is used to add the query result to the target table.
- The **INSERT OVERWRITE** statement is used to overwrite existing data in the source table.
- The **INSERT INTO** statement can be batch executed, but the **INSERT OVERWRITE** statement can be batch executed only when data of different partitioned tables is inserted to different static partitions.
- The **INSERT INTO** and **INSERT OVERWRITE** statements can be executed at the same time. However, the result is unknown.
- When you insert data of the source table to the target table, you cannot import or update data of the source table.
- The dynamic INSERT OVERWRITE statement of Hive partitioned tables can overwrite the involved partition data but cannot overwrite the entire table data.
- To overwrite data in a specified partition of the datasource table, set **dli.sql.dynamicPartitionOverwrite.enabled** to **true** and run the **insert overwrite** statement. The default value of **dli.sql.dynamicPartitionOverwrite.enabled** is **false**, indicating that data in the entire table is overwritten. The following is an example:
```
insert overwrite table tb1 partition(part1='v1', part2='v2') select * from ...
```

  NOTE

  On the DLI management console, click **SQL Editor**. In the upper right corner of the editing window, click **Settings** to configure parameters.

- You can configure the **spark.sql.shuffle.partitions** parameter to set the number of files to be inserted into the OBS bucket in the table. In addition, to avoid data skew, you can add **distribute by rand()** to the end of the INSERT statement to increase the number of concurrent jobs. The following is an example:
```
insert into table table_target select * from table_source distribute by cast(rand() * N as int);
```

Example

NOTE

Before importing data, you must create an OBS table or DLI table. For details, see **Creating an OBS Table** or **Creating a DLI Table**.

- Insert the SELECT query result into a table.
  - Use the DataSource syntax to create a parquet partitioned table.
```
CREATE TABLE data_source_tab1 (col1 INT, p1 INT, p2 INT)
  USING PARQUET PARTITIONED BY (p1, p2);
```
  - Insert the query result to the partition (p1 = 3, p2 = 4).
```
INSERT INTO data_source_tab1 PARTITION (p1 = 3, p2 = 4)
  SELECT id FROM RANGE(1, 3);
```
  - Insert the new query result to the partition (p1 = 3, p2 = 4).
```
INSERT OVERWRITE TABLE data_source_tab1 PARTITION (p1 = 3, p2 = 4)
  SELECT id FROM RANGE(3, 5);
```
- Insert a data record into a table.

- Create a Parquet partitioned table with Hive format
```
CREATE TABLE hive_serde_tab1 (col1 INT, p1 INT, p2 INT)
  USING HIVE OPTIONS(fileFormat 'PARQUET') PARTITIONED BY (p1, p2);
```
- Insert two data records into the partition (p1 = 3, p2 = 4).
```
INSERT INTO hive_serde_tab1 PARTITION (p1 = 3, p2 = 4)
  VALUES (1), (2);
```
- Insert new data to the partition (p1 = 3, p2 = 4).
```
INSERT OVERWRITE TABLE hive_serde_tab1 PARTITION (p1 = 3, p2 = 4)
  VALUES (3), (4);
```

1.12 Clearing Data

Function

This statement is used to delete data from the table.

Syntax

```
TRUNCATE TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)];
```

Keyword

Table 1-35 Parameter

Parameter	Description
tablename	Name of the target table that runs the **Truncate** statement.
partcol1	Partition name of the table to be deleted.

Precautions

Only data in the table can be deleted.

Example

```
truncate table test PARTITION (class = 'test');
```

1.13 Exporting Search Results

Function

This statement is used to directly write query results to a specified directory. The query results can be stored in CSV, Parquet, ORC, JSON, or Avro format.

Syntax

```
INSERT OVERWRITE DIRECTORY path
  USING file_format
  [OPTIONS(key1=value1)]
  select_statement;
```

Keyword

- USING: Specifies the storage format.
- OPTIONS: Specifies the list of attributes to be exported. This parameter is optional.

Parameter

Table 1-36 INSERT OVERWRITE DIRECTORY parameter description

Parameter	Description
path	The OBS path to which the query result is to be written.
file_format	Format of the file to be written. The value can be CSV, Parquet, ORC, JSON, or Avro.

NOTE

If the file format is set to **CSV**, see the **Table 1-9** for the OPTIONS parameters.

Precautions

- You can configure the **spark.sql.shuffle.partitions** parameter to set the number of files to be inserted into the OBS bucket in the table. In addition, to avoid data skew, you can add **distribute by rand()** to the end of the INSERT statement to increase the number of concurrent jobs. The following is an example:
```
insert into table table_target select * from table_source distribute by cast(rand() * N as int);
```
- When the configuration item is **OPTIONS('DELIMITER'=',')**, you can specify a separator. The default value is **,**.

  For CSV data, the following delimiters are supported:
  - Tab character, for example, **'DELIMITER'='\t'**.
  - Any binary character, for example, **'DELIMITER'='\u0001(^A)'**.
  - Single quotation mark ('). A single quotation mark must be enclosed in double quotation marks (" "). For example, **'DELIMITER'= "'"**.

Example

```
INSERT OVERWRITE DIRECTORY 'obs://bucket/dir'
 USING csv
 OPTIONS(key1=value1)
 select * from db1.tb1;
```

1.14 Backing Up and Restoring Data of Multiple Versions

1.14.1 Setting the Retention Period for Multiversion Backup Data

Function

After multiversion is enabled, backup data is retained for seven days by default. You can change the retention period by setting system parameter **dli.multi.version.retention.days**. Multiversion data out of the retention period will be automatically deleted when the **insert overwrite** or **truncate** statement is executed. You can also set table attribute **dli.multi.version.retention.days** to adjust the retention period when adding a column or modifying a partitioned table. For details about the syntax for enabling or disabling the multiversion function, see **Enabling or Disabling Multiversion Backup**.

Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.

Syntax

```
ALTER TABLE [db_name.]table_name
SET TBLPROPERTIES ("dli.multi.version.retention.days"="days");
```

Keyword

- TBLPROPERTIES: This keyword is used to add a **key/value** property to a table.

Parameter

Table 1-37 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
table_name	Table name
days	Date when the multiversion backup data is reserved. The default value is 7 days. The value ranges from 1 to 7 days.

Precautions

Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.

Example

Set the retention period of multiversion backup data to 5 days.

```
ALTER TABLE test_table
SET TBLPROPERTIES ("dli.multi.version.retention.days"="5");
```

1.14.2 Viewing Multiversion Backup Data

Function

After the multiversion function is enabled, you can run the **SHOW HISTORY** command to view the backup data of a table. For details about the syntax for enabling or disabling the multiversion function, see **Enabling or Disabling Multiversion Backup**.

Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.

Syntax

- View the backup data of a non-partitioned table.
  ```
  SHOW HISTORY FOR TABLE [db_name.]table_name;
  ```
- View the backup data of a specified partition.
  ```
  SHOW HISTORY FOR TABLE [db_name.]table_name PARTITION (column = value, ...);
  ```

Keyword

- SHOW HISTORY FOR TABLE: Used to view backup data
- PARTITION: Used to specify the partition column

Parameter

Table 1-38 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
table_name	Table name
column	Partition column name
value	Value corresponding to the partition column name

Precautions

Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.

Example

- View multiversion backup data of the **test_table** table.
  ```
  SHOW HISTORY FOR TABLE test_table;
  ```
- View multiversion backup data of the **dt** partition in the **test_table** partitioned table.

```
SHOW HISTORY FOR TABLE test_table PARTITION (dt='2021-07-27');
```

1.14.3 Restoring Multiversion Backup Data

Function

After the multiversion function is enabled, you can run the **RESTORE TABLE** statement to restore a table or partition of a specified version. For details about the syntax for enabling or disabling the multiversion function, see **Enabling or Disabling Multiversion Backup**.

Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.

Syntax

- Restore the non-partitioned table data to the backup data of a specified version.
```
RESTORE TABLE [db_name.]table_name TO VERSION 'version_id';
```
- Restore the data of a single partition in a partitioned table to the backup data of a specified version.
```
RESTORE TABLE [db_name.]table_name PARTITION (column = value, ...) TO VERSION 'version_id';
```

Keyword

- RESTORE TABLE: Used to restore backup data
- PARTITION: Used to specify the partition column
- TO VERSION: Used to specify the version number You can run the **SHOW HISTORY** command to obtain the version number. For details, see **Viewing Multiversion Backup Data**.

Parameter

Table 1-39 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
table_name	Table name
column	Partition column name
value	Value corresponding to the partition column name
version_id	Target version of the backup data to be restored You can run the **SHOW HISTORY** command to obtain the version number. For details, see **Viewing Multiversion Backup Data**.

Precautions

Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.

Example

- Restore the data in non-partitioned table **test_table** to version 20210930.
```
RESTORE TABLE test_table TO VERSION '20210930';
```
- Restore the data of partition **dt** in partitioned table **test_table** to version 20210930.
```
RESTORE TABLE test_table PARTITION (dt='2021-07-27') TO VERSION '20210930';
```

1.14.4 Configuring the Trash Bin for Expired Multiversion Data

Function

After the multiversion function is enabled, expired backup data will be directly deleted by the system when the **insert overwrite** or **truncate** statement is executed. You can configure the trash bin of the OBS parallel file system to accelerate the deletion of expired backup data. To enable the trash bin, add **dli.multi.version.trash.dir** to the table properties. For details about the syntax for enabling or disabling the multiversion function, see **Enabling or Disabling Multiversion Backup**.

Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.

Syntax

```
ALTER TABLE [db_name.]table_name
SET TBLPROPERTIES ("dli.multi.version.trash.dir"="OBS bucket for expired multiversion backup data");
```

Keyword

- TBLPROPERTIES: This keyword is used to add a **key/value** property to a table.

Parameter

Table 1-40 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
table_name	Table name

Parameter	Description
OBS bucket for expired multiversion backup data	A directory in the bucket where the current OBS table locates. You can change the directory path as needed. For example, if the current OBS table directory is **obs://bucketName/filePath** and a **Trash** directory has been created in the OBS table directory, you can set the trash bin directory to **obs://bucketName/filePath/Trash**.

Precautions

- Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.
- To automatically empty the trash bin, you need to configure a lifecycle rule for the bucket of the OBS parallel file system. The procedure is as follows:
  a. On the OBS console, choose **Parallel File System** in the left navigation pane. Click the name of the target file system. The **Overview** page is displayed.
  b. In the left navigation pane, choose **Basic Configurations** > **Lifecycle Rules** to create a lifecycle rule.

Example

Configure the trash bin to accelerate the deletion of expired backup data. The data is dumped to the **/.Trash** directory in OBS.

```
ALTER TABLE test_table
SET TBLPROPERTIES ("dli.multi.version.trash.dir"="/.Trash");
```

1.14.5 Deleting Multiversion Backup Data

Function

The retention period of multiversion backup data takes effect each time the **insert overwrite** or **truncate** statement is executed. If neither statement is executed for the table, multiversion backup data out of the retention period will not be automatically deleted. You can run the SQL commands described in this section to manually delete multiversion backup data.

Syntax

Delete multiversion backup data out of the retention period.

```
clear history for table [db_name.]table_name older_than 'timestamp';
```

Keyword

- clear history for table: Used to delete multiversion backup data
- older_than: Used to specify the time range for deleting multiversion backup data

Parameter

Table 1-41 Parameter description

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
table_name	Table name
Timestamp	Multiversion backup data generated before the timestamp will be deleted. Timestamp format: yyyy-MM-dd HH:mm:ss

Precautions

- Currently, the multiversion function supports only OBS tables created using the Hive syntax. For details about the syntax for creating a table, see **Creating an OBS Table Using the Hive Syntax**.
- This statement does not delete the backup data of the current version.

Example

Delete the multiversion backup data generated before 2021-09-25 23:59:59 in the **dliTable** table. When the multiversion backup data is generated, a timestamp is generated.

```
clear history for table dliTable older_than '2021-09-25 23:59:59';
```

1.15 Creating a Datasource Connection with an HBase Table

1.15.1 Creating a Table and Associating It with HBase

Function

This statement is used to create a table and associate it with an existing HBase table.

Prerequisites

- Before creating a table and associating it with HBase, you need to create a datasource connection. For details about operations on the management console, see
- Ensure that the **/etc/hosts** information of the master node in the MRS cluster is added to the host file of the DLI queue.
- The syntax is not supported for security clusters.

Syntax

- Single row key

```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME (
  ATTR1 TYPE,
  ATTR2 TYPE,
  ATTR3 TYPE)
  USING [HBASE] OPTIONS (
  'ZKHost'='xx',
  'TableName'='TABLE_IN_HBASE',
  'RowKey'='ATTR1',
  'Cols'='ATTR2:CF1.C1, ATTR3:CF1.C2');
```

- Combined row key

```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME (
  ATTR1 String,
  ATTR2 String,
  ATTR3 TYPE)
  USING [HBASE] OPTIONS (
  'ZKHost'='xx',
  'TableName'='TABLE_IN_HBASE',
  'RowKey'='ATTR1:2, ATTR2:10',
  'Cols'='ATTR2:CF1.C1, ATTR3:CF1.C2'
```

Keyword

Table 1-42 CREATE TABLE parameter description

Parameter	Description
USING [HBASE]	Specify the HBase datasource. The value is case insensitive.
ZKHost	ZooKeeper IP address of the HBase cluster. Before obtaining the ZooKeeper IP address, you need to create a datasource connection first.. ● To access the MRS cluster, enter the IP address of the node where the ZooKeeper is located and the external port number of the ZooKeeper. The format is **ZK_IP1:ZK_PORT1,ZK_IP2:ZK_PORT2**. **NOTE**
TableName	Specifies the name of a table that has been created in the HBase cluster.
RowKey	Specifies the row key field of the table connected to DLI. The single and composite row keys are supported. A single row key can be of the numeric or string type. The length does not need to be specified. The composite row key supports only fixed-length data of the string type. The format is ***attribute name 1*:*Length*, *attribute name 2*:*length***.
Cols	Provides mappings between fields in the table and columns in the HBase table. The mappings are separated by commas (,). In a mapping, the field in the table is located before the colon (:) and information about the HBase table follows the colon (:). In the HBase table information, the column family and column name are separated using a dot (.).

Precautions

- If the to-be-created table exists, an error is reported. To avoid such error, add **IF NOT EXISTS** in this statement.
- All parameters in OPTIONS are mandatory. Parameter names are case-insensitive, while parameter values are case-sensitive.
- In OPTIONS, spaces are not allowed before or after the value in the quotation marks because spaces are also considered as a part of the value.
- Descriptions of table names and column names support only string constants.
- When creating a table, specify the column name and the corresponding data types. Currently, supported data types include Boolean, short, int, long, float, double, and string.
- The value of **row key** (for example, ATTR1) cannot be null, and its length must be greater than 0 and less than or equal to 32767.
- The total number of fields in **Cols** and **row key** must be the same as that in the table. Specifically, all fields in the table are mapped to **Cols** and **row key** without sequence requirements specified.
- The combined row key only supports data of the string type. If the combined row key is used, the length must follow each attribute name. If only one field is specified as the row key, the field type can be any supported data type and you do not need to specify the length.
- If the combined row key is used:
  - When the row key is inserted, if the actual attribute length is shorter than the specified length when the attribute is used as the row key, add **\0** after the attribute. If it is longer, the attribute will be truncated when it is inserted into HBase.
  - When reading the **row key** field in HBase, if the actual data length of an attribute is shorter than that specified when the attribute is used as the **row key**, an error message (**OutofBoundException**) is reported. If it is longer, the attribute will be truncated during data reading.

Example

```
CREATE TABLE test_hbase(
ATTR1 int,
ATTR2 int,
ATTR3 string)
using hbase OPTIONS (
'ZKHost'='to-hbase-1174405101-CE1bDm5B.datasource.com:2181',
'TableName'='HBASE_TABLE',
'RowKey'='ATTR1',
'Cols'='ATTR2:CF1.C1, ATTR3:CF1.C2');
```

1.15.2 Inserting Data to an HBase Table

Function

This statement is used to insert data in a table to the associated HBase table.

Syntax

- Insert the SELECT query result into a table.

```
INSERT INTO DLI_TABLE
  SELECT field1,field2...
```

```
[FROM DLI_TEST]
[WHERE where_condition]
[LIMIT num]
[GROUP BY field]
[ORDER BY field] ...;
```

- Insert a data record into a table.
```
INSERT INTO DLI_TABLE
  VALUES values_row [, values_row ...];
```

Keywords

For details about the SELECT keywords, see **Basic SELECT Statements**.

Parameter description

Table 1-43 Parameter description

Parameter	Description
DLI_TABLE	Name of the table for which a datasource connection has been created.
DLI_TEST	indicates the table that contains the data to be queried.
field1,field2..., field	Column values in the DLI_TEST table must match the column values and types in the DLI_TABLE table.
where_condition	Query condition.
num	Limit the query result. The num parameter supports only the INT type.
values_row	Value to be inserted to a table. Use commas (,) to separate columns.

Precautions

- The target table must exist.
- In the column family created in **Creating a Table and Associating It with HBase**, if the column family specified by **Cols** in **OPTIONS** does not exist, an error is reported when **INSERT INTO** is executed.
- If the row key, column family, or column you need to insert to the HBase table already exists, the existing data in HBase table will be overwritten.
- You are advised not to concurrently insert data into a table. If you concurrently insert data into a table, there is a possibility that conflicts occur, leading to failed data insertion.
- **INSERT OVERWRITE** is not supported.

Example

- Query data in the user table and insert the data into the test table.
```
INSERT INTO test
  SELECT ATTR_EXPR
```

```
FROM user
WHERE user_name='cyz'
LIMIT 3
GROUP BY user_age
```

- Insert data 1 into the test table.
```
INSERT INTO test
  VALUES (1);
```

1.15.3 Querying an HBase Table

This statement is used to query data in an HBase table.

Syntax

```
SELECT * FROM table_name LIMIT number;
```

Keyword

LIMIT is used to limit the query results. Only INT type is supported by the **number** parameter.

Precautions

The table to be queried must exist. Otherwise, an error is reported.

Example

Query data in the **test_ct** table.

```
SELECT * FROM test_hbase limit 100;
```

Query Pushdown

Query pushdown implements data filtering using HBase. Specifically, the HBase Client sends filtering conditions to the HBase server, and the HBase server returns only the required data, speeding up your Spark SQL queries. For the filter criteria that HBase does not support, for example, query with the composite row key, Spark SQL performs data filtering.

- Scenarios where query pushdown is supported
  - Query pushdown can be performed on data of the following types:
    - Int
    - boolean
    - short
    - long
    - double
    - string

    NOTE

    Data of the float type does not support query pushdown.

- Query pushdown is not supported for the following filter criteria:
  - >, <, >=, <=, =, !=, and, or

    The following is an example:

    ```
    select * from tableName where (column1 >= value1 and column2<= value2) or column3 !
    = value3
    ```

  - The filtering conditions are **like** and **not like**. The prefix, suffix, and inclusion match are supported.

    The following is an example:

    ```
    select * from tableName where column1 like "%value" or column2 like "value%" or
    column3 like "%value%"
    ```

  - IsNotNull()

    The following is an example:

    ```
    select * from tableName where IsNotNull(column)
    ```

  - in and not in

    The following is an example:

    ```
    select * from tableName where column1 in (value1,value2,value3)  and column2 not in
    (value4,value5,value6)
    ```

  - between _ and _

    The following is an example:

    ```
    select * from tableName where column1 between value1 and value2
    ```

  - Filtering of the row sub-keys in the composite row key

    For example, to perform row sub-key query on the composite row key **column1+column2+column3**, run the following statement:

    ```
    select * from tableName where column1= value1
    ```

- Scenarios where query pushdown is not supported
  - Query pushdown can be performed on data of the following types:

    Except for the preceding data types where query pushdown is supported, data of other types does not support query pushdown.
  - Query pushdown is not supported for the following filter criteria:
    - Length, count, max, min, join, groupby, orderby, limit, and avg
    - Column comparison

      The following is an example:

      ```
      select * from tableName where column1 > (column2+column3)
      ```

1.16 Creating a Datasource Connection with an OpenTSDB Table

1.16.1 Creating a Table and Associating It with OpenTSDB

Function

Run the CREATE TABLE statement to create the table and associate it with the existing metric in OpenTSDB. This syntax supports the OpenTSDB of MRS.

Prerequisites

Before creating a table and associating it with OpenTSDB, you need to create a datasource connection. For details about operations on the management console, see

Syntax

```
CREATE TABLE [IF NOT EXISTS] UQUERY_OPENTSDB_TABLE_NAME
  USING OPENTSDB OPTIONS (
  'host' = 'xx;xx',
  'metric' = 'METRIC_NAME',
  'tags' = 'TAG1,TAG2');
```

Keyword

Table 1-44 CREATE TABLE parameter description

Parameter	Description
host	OpenTSDB IP address. Before obtaining the OpenTSDB IP address, you need to create a datasource connection first.. ● You can also access the MRS OpenTSDB. If you have created an enhanced datasource connection, enter the IP address and port number of the node where the OpenTSDB is located. The format is **IP:PORT**. If the OpenTSDB has multiple nodes, enter one of the node IP addresses.
metric	Name of the metric in OpenTSDB corresponding to the table to be created.
tags	Tags corresponding to the metric. The tags are used for classification, filtering, and quick retrieval. You can set 1 to 8 tags, which are separated by commas (,). The parameter value includes values of all tagKs in the corresponding metric.

Precautions

When creating a table, you do not need to specify the **timestamp** and **value** fields. The system automatically builds the following fields based on the specified tags. The fields **TAG1** and **TAG2** are specified by tags.

- TAG1 String

- TAG2 String
- timestamp Timestamp
- value double

Example

```
CREATE table opentsdb_table
  USING OPENTSDB OPTIONS (
  'host' = 'opentsdb-3xcl8dir15m58z3.com:4242',
  'metric' = 'city.temp',
  'tags' = 'city,location');
```

1.16.2 Inserting Data to the OpenTSDB Table

Function

Run the **INSERT INTO** statement to insert the data in the table to the associated **OpenTSDB metric**.

NOTE

If no metric exists on the OpenTSDB, a new metric is automatically created on the OpenTSDB when data is inserted.

Syntax

```
INSERT INTO TABLE TABLE_NAME SELECT * FROM DLI_TABLE;
INSERT INTO TABLE TABLE_NAME VALUES(XXX);
```

Keyword

Table 1-45 INSERT INTO parameter description

Parameter	Description
TABLE_NAME	Name of the associated OpenTSDB table.
DLI_TABLE	Name of the table created.

Precautions

- The inserted data cannot be **null**. If the inserted data is the same as the original data or only the **value** is different, the inserted data overwrites the original data.
- **INSERT OVERWRITE** is not supported.
- You are advised not to concurrently insert data into a table. If you concurrently insert data into a table, there is a possibility that conflicts occur, leading to failed data insertion.
- The **TIMESTAMP** format supports only yyyy-MM-dd hh:mm:ss.

Example

```
INSERT INTO TABLE opentsdb_table VALUES('xxx','xxx','2018-05-03 00:00:00',21);
```

1.16.3 Querying an OpenTSDB Table

This **SELECT** command is used to query data in an OpenTSDB table.

NOTE

- If no metric exists in OpenTSDB, an error will be reported when the corresponding table is queried.
- If the security mode is enabled, you need to set **conf:dli.sql.mrs.opentsdb.ssl.enabled** to **true** when connecting to OpenTSDB.

Syntax

```
SELECT * FROM table_name LIMIT number;
```

Keyword

LIMIT is used to limit the query results. Only INT type is supported by the **number** parameter.

Precautions

The table to be queried must exist. Otherwise, an error is reported.

Example

Query data in the **opentsdb_table** table.

```
SELECT * FROM opentsdb_table limit 100;
```

1.17 Creating a Datasource Connection with a DWS table

1.17.1 Creating a Table and Associating It with DWS

Function

This statement is used to create a table and associate it with an existing DWS table.

Prerequisites

Before creating a table and associating it with DWS, you need to create a datasource connection. For details about operations on the management console, see

Syntax

```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME
  USING JDBC OPTIONS (
  'url'='xx',
  'dbtable'='db_name_in_DWS.table_name_in_DWS',
```

```
'passwdauth' = 'xxx',
'encryption' = 'true');
```

Keyword

Table 1-46 CREATE TABLE parameter description

Parameter	Description
url	Before obtaining the DWS IP address, you need to create a datasource connection first.. If you have created an enhanced datasource connection, you can use the **JDBC Connection String (intranet)** provided by DWS or the intranet address and port number to access DWS. The format is ***protocol header*: //*Internal IP address*:*Internal network port*/ *Database name***, for example: **jdbc:postgresql:// 192.168.0.77:8000/postgres**. **NOTE** The DWS IP address is in the following format: ***protocol header*://*IP address*:*port number*/*database name*** The following is an example: jdbc:postgresql://to-dws-1174405119-ihlUr78j.datasource.com:8000/ postgres If you want to connect to a database created in DWS, change **postgres** to the corresponding database name in this connection.
dbtable	Specifies the name or **Schema name.Table name** of the table that is associated with the DWS. For example: **public.table_name**.
user	(Discarded) DWS username.
password	User password of the DWS cluster.
passwdauth	Datasource password authentication name. For details about how to create datasource authentication, see in the *Data Lake Insight User Guide*.
encryption	Set this parameter to **true** when datasource password authentication is used.
partitionColumn	This parameter is used to set the numeric field used concurrently when data is read. **NOTE** ● The **partitionColumn**, **lowerBound**, **upperBound**, and **numPartitions** parameters must be set at the same time. ● To improve the concurrent read performance, you are advised to use auto-increment columns.
lowerBound	Minimum value of a column specified by **partitionColumn**. The value is contained in the returned result.
upperBound	Maximum value of a column specified by **partitionColumn**. The value is not contained in the returned result.

Parameter	Description
numPartitions	Number of concurrent read operations. **NOTE** When data is read, the number of concurrent operations are evenly allocated to each task according to the **lowerBound** and **upperBound** to obtain data. The following is an example: 'partitionColumn'='id', 'lowerBound'='0', 'upperBound'='100', 'numPartitions'='2' Two concurrent tasks are started in DLI. The execution ID of one task is greater than or equal to **0** and the ID is less than **50**, and the execution ID of the other task is greater than or equal to **50** and the ID is less than **100**.
fetchsize	Number of data records obtained in each batch during data reading. The default value is **1000**. If this parameter is set to a large value, the performance is good but more memory is occupied. If this parameter is set to a large value, memory overflow may occur.
batchsize	Number of data records written in each batch. The default value is **1000**. If this parameter is set to a large value, the performance is good but more memory is occupied. If this parameter is set to a large value, memory overflow may occur.
truncate	Indicates whether to clear the table without deleting the original table when **overwrite** is executed. The options are as follows: ● true ● false The default value is **false**, indicating that the original table is deleted and then a new table is created when the **overwrite** operation is performed.
isolationLevel	Transaction isolation level. The options are as follows: ● NONE ● READ_UNCOMMITTED ● READ_COMMITTED ● REPEATABLE_READ ● SERIALIZABLE The default value is **READ_UNCOMMITTED**.

Precautions

When creating a table associated with DWS, you do not need to specify the **Schema** of the associated table. DLI automatically obtains the schema of the table in the **dbtable** parameter of DWS.

Example

```
CREATE TABLE IF NOT EXISTS dli_to_dws
  USING JDBC OPTIONS (
```



```
'url'='jdbc:postgresql://to-dws-1174405119-ih1Ur78j.datasource.com:8000/postgres',
'dbtable'='test_dws',
'passwdauth' = 'xxx',
'encryption' = 'true');
```

1.17.2 Inserting Data to the DWS Table

Function

This statement is used to insert data in a table to the associated DWS table.

Syntax

- Insert the SELECT query result into a table.

```
INSERT INTO DLI_TABLE
  SELECT field1,field2...
  [FROM DLI_TEST]
  [WHERE where_condition]
  [LIMIT num]
  [GROUP BY field]
  [ORDER BY field] ...;
```

- Insert a data record into a table.

```
INSERT INTO DLI_TABLE
  VALUES values_row [, values_row ...];
```

Keywords

For details about the SELECT keywords, see **Basic SELECT Statements**.

Parameter description

Table 1-47 Parameter description

Parameter	Description
DLI_TABLE	Name of the table for which a datasource connection has been created.
DLI_TEST	indicates the table that contains the data to be queried.
field1,field2..., field	Column values in the DLI_TEST table must match the column values and types in the DLI_TABLE table.
where_condition	Query condition.
num	Limit the query result. The num parameter supports only the INT type.
values_row	Value to be inserted to a table. Use commas (,) to separate columns.

Precautions

- The target table must exist.

- When creating the table, you do not need to specify the **Schema** information. The **Schema** information complies with that in the DWS table. If the number and type of fields selected in the **SELECT** clause do not match the **Schema** information in the DWS table, the system reports an error.
- You are advised not to concurrently insert data into a table. If you concurrently insert data into a table, there is a possibility that conflicts occur, leading to failed data insertion.

Example

- Query data in the user table and insert the data into the test table.
```
INSERT INTO test
  SELECT ATTR_EXPR
  FROM user
  WHERE user_name='cyz'
  LIMIT 3
  GROUP BY user_age
```
- Insert data 1 into the test table.
```
INSERT INTO test
  VALUES (1);
```

1.17.3 Querying the DWS Table

This statement is used to query data in a DWS table.

Syntax

```
SELECT * FROM table_name LIMIT number;
```

Keyword

LIMIT is used to limit the query results. Only INT type is supported by the **number** parameter.

Precautions

The table to be queried must exist. Otherwise, an error is reported.

Example

To query data in the **dli_to_dws** table, enter the following statement:

```
SELECT * FROM dli_to_dws limit 100;
```

1.18 Creating a Datasource Connection with an RDS Table

1.18.1 Creating a Table and Associating It with RDS

Function

This statement is used to create a table and associate it with an existing RDS table. This function supports access to the MySQL and PostgreSQL clusters of RDS.

Prerequisites

Before creating a table and associating it with RDS, you need to create a datasource connection. For details about operations on the management console, see

Syntax

```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME
  USING JDBC OPTIONS (
  'url'='xx',
  'driver'='DRIVER_NAME',
  'dbtable'='db_name_in_RDS.table_name_in_RDS',
  'passwdauth' = 'xxx',
  'encryption' = 'true');
```

Keywords

Table 1-48 CREATE TABLE parameter description

Parameter	Description
url	Before obtaining the RDS IP address, you need to create a datasource connection first.. After an enhanced datasource connection is created, use the internal network domain name or internal network address and database port number provided by RDS to connect to DLI. If MySQL is used, the format is ***protocol header://internal IP address:internal network port number***. If PostgreSQL is used, the format is ***protocol header://internal IP address:internal network port number/database name***. For example: **jdbc:mysql://192.168.0.193:3306** or **jdbc:postgresql://192.168.0.193:3306/postgres**.
driver	JDBC driver class name. To connect to a MySQL cluster, enter **com.mysql.jdbc.Driver**. To connect to a PostgreSQL cluster, enter **org.postgresql.Driver**.
dbtable	• To access the MySQL cluster, enter ***Database name.Table name***. **CAUTION** The name of the RDS database cannot contain hyphens (-) or ^. Otherwise, the table fails to be created. • To access the PostGre cluster, enter ***Schema name.Table name*** **NOTE** The schema name is the name of the database schema. A schema is a collection of database objects, including tables and views.
user	(Discarded) Specifies the RDS username.
password	(Discarded) Specifies the RDS username and password.

Parameter	Description
passwdauth	Datasource password authentication name. For details about how to create datasource authentication, see in the *Data Lake Insight User Guide*.
encryption	Set this parameter to **true** when datasource password authentication is used.
partitionColumn	This parameter is used to set the numeric field used concurrently when data is read. **NOTE** ● The **partitionColumn**, **lowerBound**, **upperBound**, and **numPartitions** parameters must be set at the same time. ● To improve the concurrent read performance, you are advised to use auto-increment columns.
lowerBound	Minimum value of a column specified by **partitionColumn**. The value is contained in the returned result.
upperBound	Maximum value of a column specified by **partitionColumn**. The value is not contained in the returned result.

numPartitions	Number of concurrent read operations.
	NOTE
	When data is read, the number of concurrent operations are evenly allocated to each task according to the **lowerBound** and **upperBound** to obtain data. The following is an example:
	`'partitionColumn'='id',`
	`'lowerBound'='0',`
	`'upperBound'='100',`
	`'numPartitions'='2'`
	Two concurrent tasks are started in DLI. The execution ID of one task is greater than or equal to **0** and the ID is less than **50**, and the execution ID of the other task is greater than or equal to **50** and the ID is less than **100**.

fetchsize	Number of data records obtained in each batch during data reading. The default value is **1000**. If this parameter is set to a large value, the performance is good but more memory is occupied. If this parameter is set to a large value, memory overflow may occur.
batchsize	Number of data records written in each batch. The default value is **1000**. If this parameter is set to a large value, the performance is good but more memory is occupied. If this parameter is set to a large value, memory overflow may occur.
truncate	Indicates whether to clear the table without deleting the original table when **overwrite** is executed. The options are as follows: ● true ● false The default value is **false**, indicating that the original table is deleted and then a new table is created when the **overwrite** operation is performed.

Parameter	Description
isolationLevel	Transaction isolation level. The options are as follows: ● NONE ● READ_UNCOMMITTED ● READ_COMMITTED ● REPEATABLE_READ ● SERIALIZABLE The default value is **READ_UNCOMMITTED**.

Precautions

When creating a table associated with RDS, you do not need to specify the **Schema** of the associated table. DLI automatically obtains the schema of the table in the **dbtable** parameter of RDS.

Example

Accessing MySQL:

```
CREATE TABLE IF NOT EXISTS dli_to_rds
  USING JDBC OPTIONS (
  'url'='jdbc:mysql://to-rds-117405104-3eAHxnlz.datasource.com:3306',
  'driver'='com.mysql.jdbc.Driver',
  'dbtable'='rds_test.test1',
  'passwdauth' = 'xxx',
  'encryption' = 'true');
```

Accessing PostgreSQL:

```
CREATE TABLE IF NOT EXISTS dli_to_rds
  USING JDBC OPTIONS (
  'url'='jdbc:postgresql://to-rds-1174405119-oLRHAGE7.datasource.com:3306/postgreDB',
  'driver'='org.postgresql.Driver',
  'dbtable'='pg_schema.test1',
  'passwdauth' = 'xxx',
  'encryption' = 'true');
```

1.18.2 Inserting Data to the RDS Table

Function

This statement is used to insert data in a table to the associated RDS table.

Syntax

- Insert the SELECT query result into a table.

```
INSERT INTO DLI_TABLE
  SELECT field1,field2...
  [FROM DLI_TEST]
  [WHERE where_condition]
  [LIMIT num]
  [GROUP BY field]
  [ORDER BY field] ...;
```

- Insert a data record into a table.
```
INSERT INTO DLI_TABLE
  VALUES values_row [, values_row ...];
```

Keywords

For details about the SELECT keywords, see **Basic SELECT Statements**.

Parameter description

Table 1-49 Parameter description

Parameter	Description
DLI_TABLE	Name of the table for which a datasource connection has been created.
DLI_TEST	indicates the table that contains the data to be queried.
field1,field2..., field	Column values in the DLI_TEST table must match the column values and types in the DLI_TABLE table.
where_condition	Query condition.
num	Limit the query result. The num parameter supports only the INT type.
values_row	Value to be inserted to a table. Use commas (,) to separate columns.

Precautions

- The target table must exist.
- When creating the table, you do not need to specify the **Schema** information. The **Schema** information complies with that in the RDS table. If the number and type of fields selected in the **SELECT** clause do not match the **Schema** information in the RDS table, the system reports an error.
- You are advised not to concurrently insert data into a table. If you concurrently insert data into a table, there is a possibility that conflicts occur, leading to failed data insertion.

Example

- Query data in the user table and insert the data into the test table.
```
INSERT INTO test
  SELECT ATTR_EXPR
  FROM user
  WHERE user_name='cyz'
  LIMIT 3
  GROUP BY user_age
```
- Insert data 1 into the test table.
```
INSERT INTO test
  VALUES (1);
```


1.18.3 Querying the RDS Table

This statement is used to query data in an RDS table.

Syntax

```
SELECT * FROM table_name LIMIT number;
```

Keyword

LIMIT is used to limit the query results. Only INT type is supported by the **number** parameter.

Precautions

The table to be queried must exist. Otherwise, an error is reported.

Example

Query data in the **test_ct** table.

```
SELECT * FROM dli_to_rds limit 100;
```

1.19 Creating a Datasource Connection with a CSS Table

1.19.1 Creating a Table and Associating It with CSS

Function

This statement is used to create a table and associate it with an existing CSS table.

Prerequisites

Before creating a table and associating it with CSS, you need to create a datasource connection. For details about operations on the management console, see

Syntax

```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME(
  FIELDNAME1 FIELDTYPE1,
  FIELDNAME2 FIELDTYPE2)
  USING CSS OPTIONS (
  'es.nodes'='xx',
  'resource'='type_path_in_CSS',
  'pushdown'='true',
  'strict'='false',
  'batch.size.entries'= '1000',
  'batch.size.bytes'= '1mb',
  'es.nodes.wan.only' = 'true',
  'es.mapping.id' = 'FIELDNAME');
```


Keyword

Table 1-50 CREATE TABLE parameter description

Parameter	Description
es.nodes	Before obtaining the CSS IP address, you need to create a datasource connection first.. If you have created an enhanced datasource connection, you can use the internal IP address provided by CSS. The format is ***IP1*:*PORT1*,*IP2*:*PORT2***.
resource	The **resource** is used to specify the CSS datasource connection name. You can use **/index/type** to specify the resource location (for easier understanding, the **index** can be seen as **database** and **type** as **table**). **NOTE** ● In ES 6.X, a single index supports only one type, and the type name can be customized. ● In ES 7.X, a single index uses **_doc** as the type name and cannot be customized. To access ES 7.X, set this parameter to **index**.
pushdown	Indicates whether the press function of CSS is enabled. The default value is set to **true**. If there are a large number of I/O transfer tables, the **pushdown** can be enabled to reduce I/Os when the **where** filtering conditions are met.
strict	Indicates whether the CSS **pushdown** is strict. The default value is set to **false**. In exact match scenarios, more I/Os are reduced than **pushdown**.
batch.size.entries	Maximum number of entries that can be inserted to a batch processing. The default value is **1000**. If the size of a single data record is so large that the number of data records in the bulk storage reaches the upper limit of the data amount of a single batch processing, the system stops storing data and submits the data based on the **batch.size.bytes**.
batch.size.bytes	Maximum amount of data in a single batch processing. The default value is 1 MB. If the size of a single data record is so small that the number of data records in the bulk storage reaches the upper limit of the data amount of a single batch processing, the system stops storing data and submits the data based on the **batch.size.entries**.
es.nodes.wan.only	Indicates whether to access the Elasticsearch node using only the domain name. The default value is **false**. If the original internal IP address provided by CSS is used as the **es.nodes**, you do not need to set this parameter or set it to **false**.

Parameter	Description
es.mapping.id	Specifies a field whose value is used as the document ID in the Elasticsearch node. **NOTE** ● The document ID in the same **/index/type** is unique. If a field that functions as a document ID has duplicate values, the document with the duplicate ID will be overwritten when the ES is inserted. ● This feature can be used as a fault tolerance solution. When data is being inserted, the DLI job fails and some data has been inserted into Elasticsearch. The data is redundant. If Document id is set, the last redundant data will be overwritten when the DLI job is executed again.
es.net.ssl	Whether to connect to the secure CSS cluster. The default value is **false**.
es.certificate.name	Name of the datasource authentication used to connect to the secure CSS cluster. For details about how to create datasource authentication, see in the *Data Lake Insight User Guide*.

NOTE

batch.size.entries and **batch.size.bytes** limit the number of data records and data volume respectively.

Example

```
CREATE TABLE IF NOT EXISTS dli_to_css (doc_id String, name string, age int)
  USING CSS OPTIONS (
  es.nodes 'to-css-1174404703-LzwpJEyx.datasource.com:9200',
  resource '/dli_index/dli_type',
  pushdown 'false',
  strict 'true',
  es.nodes.wan.only 'true',
  es.mapping.id 'doc_id');
```

1.19.2 Inserting Data to the CSS Table

Function

This statement is used to insert data in a table to the associated CSS table.

Syntax

- Insert the SELECT query result into a table.
```
INSERT INTO DLI_TABLE
  SELECT field1,field2...
  [FROM DLI_TEST]
  [WHERE where_condition]
  [LIMIT num]
  [GROUP BY field]
  [ORDER BY field] ...;
```
- Insert a data record into a table.
```
INSERT INTO DLI_TABLE
  VALUES values_row [, values_row ...];
```

Keywords

For details about the SELECT keywords, see **Basic SELECT Statements**.

Parameter description

Table 1-51 Parameter description

Parameter	Description
DLI_TABLE	Name of the table for which a datasource connection has been created.
DLI_TEST	indicates the table that contains the data to be queried.
field1,field2..., field	Column values in the DLI_TEST table must match the column values and types in the DLI_TABLE table.
where_condition	Query condition.
num	Limit the query result. The num parameter supports only the INT type.
values_row	Value to be inserted to a table. Use commas (,) to separate columns.

Precautions

- The target table must exist.
- When creating the table, you need to specify the **schema** information. If the number and type of fields selected in the **SELECT** clause or in **Values** do not match the **Schema** information in the CSS table, the system reports an error.
- Inconsistent types may not always cause error reports. For example, if the data of the **int** type is inserted, but the **text** type is saved in the CSS **Schema**, the **int** type will be converted to the **text** type and no error will be reported.
- You are advised not to concurrently insert data into a table. If you concurrently insert data into a table, there is a possibility that conflicts occur, leading to failed data insertion.

Example

- Query data in the user table and insert the data into the test table.
```
INSERT INTO test
  SELECT ATTR_EXPR
  FROM user
  WHERE user_name='cyz'
  LIMIT 3
  GROUP BY user_age
```
- Insert data 1 into the test table.
```
INSERT INTO test
  VALUES (1);
```

1.19.3 Querying the CSS Table

This statement is used to query data in a CSS table.

Syntax

```
SELECT * FROM table_name LIMIT number;
```

Keyword

LIMIT is used to limit the query results. Only INT type is supported by the **number** parameter.

Precautions

The table to be queried must exist. Otherwise, an error is reported.

Example

To query data in the **dli_to_css** table, enter the following statement:

```
SELECT * FROM dli_to_css limit 100;
```

1.20 Creating a Datasource Connection with a DCS Table

1.20.1 Creating a Table and Associating It with DCS

Function

This statement is used to create a table and associate it with an existing DCS key.

Prerequisites

Before creating a table and associating it with DCS, you need to create a datasource connection and bind it to a queue. For details about operations on the management console, see

Syntax

- Specified key
```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME(
    FIELDNAME1 FIELDTYPE1,
    FIELDNAME2 FIELDTYPE2)
  USING REDIS OPTIONS (
  'host'='xx',
  'port'='xx',
  'passwdauth' = 'xxx',
  'encryption' = 'true',
  'table'='namespace_in_redis:key_in_redis',
  'key.column'= 'FIELDNAME1'
);
```
- Wildcard key

```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME(
    FIELDNAME1 FIELDTYPE1,
    FIELDNAME2 FIELDTYPE2)
  USING REDIS OPTIONS (
  'host'='xx',
  'port'='xx',
  'passwdauth' = 'xxx',
  'encryption' = 'true',
  'keys.pattern'='key*:*',
  'key.column'= 'FIELDNAME1'
);
```

Keyword

Table 1-52 CREATE TABLE parameter description

Parameter	Description
host	To connect to DCS, you need to create a datasource connection first. After creating an enhanced datasource connection, use the connection address provided by DCS. If there are multiple connection addresses, select one of them. **NOTE** Currently, only enhanced datasource is supported.
port	DCS connection port, for example, 6379.
password	Password entered during DCS cluster creation. You do not need to set this parameter when accessing a non-secure Redis cluster.
passwdauth	Datasource password authentication name. For details about how to create datasource authentication, see in the *Data Lake Insight User Guide*.
encryption	Set this parameter to **true** when datasource password authentication is used.
table	The key or hash key in Redis. ● This parameter is mandatory when Redis data is inserted. ● Either this parameter or the **keys.pattern** parameter when Redis data is queried.
keys.pattern	Use a regular expression to match multiple keys or hash keys. This parameter is used only for query. Either this parameter or **table** is used to query Redis data.
key.column	(Optional) Specifies a field in the schema as the key ID in Redis. This parameter is used together with the **table** parameter when data is inserted.
partitions.number	Number of concurrent tasks during data reading.

Parameter	Description
scan.count	Number of data records read in each batch. The default value is **100**. If the CPU usage of the Redis cluster still needs to be improved during data reading, increase the value of this parameter.
iterator.grouping.size	Number of data records inserted in each batch. The default value is **100**. If the CPU usage of the Redis cluster still needs to be improved during the insertion, increase the value of this parameter.
timeout	Timeout interval for connecting to the Redis, in milliseconds. The default value is **2000** (2 seconds).

NOTE

When connecting to DCS, complex data types such as Array, Struct, and Map are not supported.

The following methods can be used to process complex data:

- Place the fields of the next level in the Schema field of the same level.
- Write and read data in binary mode, and encode and decode it using user-defined functions.

Example

- Specified table

```
create table test_redis(name string, age int) using redis options(
  'host' = '192.168.4.199',
  'port' = '6379',
  'passwdauth' = 'xxx',
  'encryption' = 'true',
  'table' = 'person'
);
```

- Wildcard table name

```
create table test_redis_keys_patten(id string, name string, age int) using redis options(
  'host' = '192.168.4.199',
  'port' = '6379',
  'passwdauth' = 'xxx',
  'encryption' = 'true',
  'keys.pattern' = 'p*:*',
  'key.column' = 'id'
);
```

1.20.2 Inserting Data to a DCS Table

Function

This statement is used to insert data in a table to the DCS key.

Syntax

- Insert the SELECT query result into a table.

```
INSERT INTO DLI_TABLE
  SELECT field1,field2...
```

```
[FROM DLI_TEST]
[WHERE where_condition]
[LIMIT num]
[GROUP BY field]
[ORDER BY field] ...;
```

- Insert a data record into a table.

```
INSERT INTO DLI_TABLE
  VALUES values_row [, values_row ...];
```

Keywords

For details about the SELECT keywords, see **Basic SELECT Statements**.

Parameter description

Table 1-53 Parameter description

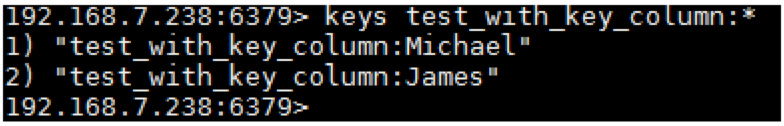
Parameter	Description
DLI_TABLE	Name of the table for which a datasource connection has been created.
DLI_TEST	indicates the table that contains the data to be queried.
field1,field2..., field	Column values in the DLI_TEST table must match the column values and types in the DLI_TABLE table.
where_condition	Query condition.
num	Limit the query result. The num parameter supports only the INT type.
values_row	Value to be inserted to a table. Use commas (,) to separate columns.

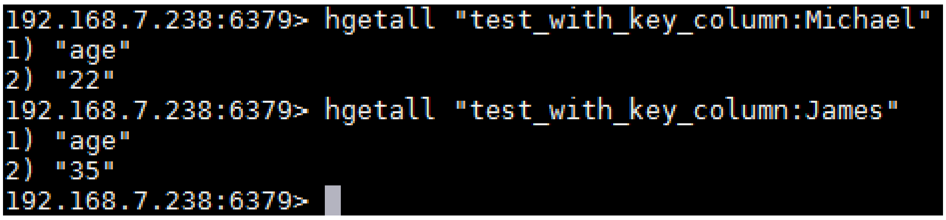
Precautions

- The target table must exist.
- When creating a table, you need to specify the schema information.
- If **key.column** is specified during table creation, the value of the specified field is used as a part of the Redis key name. The following is an example:

```
create table test_redis(name string, age int) using redis options(
  'host' = '192.168.4.199',
  'port' = '6379',
  'password' = '******',
  'table' = 'test_with_key_column',
  'key.column' = 'name'
);
insert into test_redis values("James", 35), ("Michael", 22);
```

The Redis database contains two tables, naming **test_with_key_column:James** and **test_with_key_column:Michael** respectively.



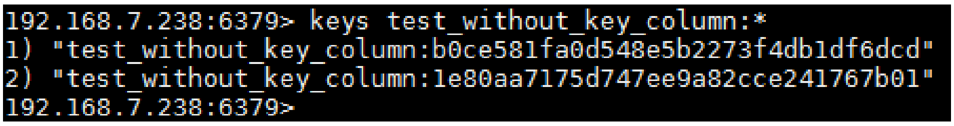


```
192.168.7.238:6379> hgetall "test_with_key_column:Michael"
1) "age"
2) "22"
192.168.7.238:6379> hgetall "test_with_key_column:James"
1) "age"
2) "35"
192.168.7.238:6379>
```

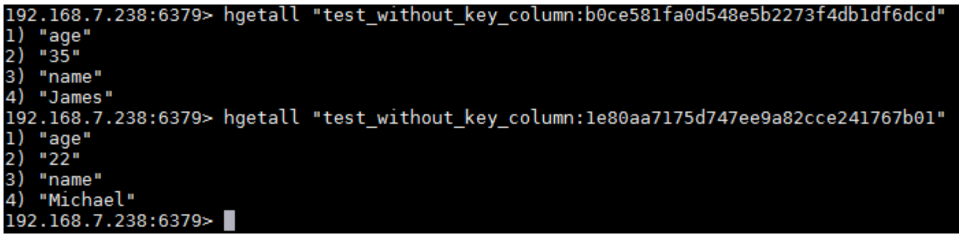
- If **key.column** is not specified during table creation, the key name in Redis uses the UUID. The following is an example:

```
create table test_redis(name string, age int) using redis options(
  'host' = '192.168.7.238',
  'port' = '6379',
  'password' = '******',
  'table' = 'test_without_key_column'
);
insert into test_redis values("James", 35), ("Michael", 22);
```

In Redis, there are two tables named **test_without_key_column:uuid**.



```
192.168.7.238:6379> keys test_without_key_column:*
1) "test_without_key_column:b0ce581fa0d548e5b2273f4db1df6dcd"
2) "test_without_key_column:1e80aa7175d747ee9a82cce241767b01"
192.168.7.238:6379>
```



```
192.168.7.238:6379> hgetall "test_without_key_column:b0ce581fa0d548e5b2273f4db1df6dcd"
1) "age"
2) "35"
3) "name"
4) "James"
192.168.7.238:6379> hgetall "test_without_key_column:1e80aa7175d747ee9a82cce241767b01"
1) "age"
2) "22"
3) "name"
4) "Michael"
192.168.7.238:6379>
```

Example

```
INSERT INTO test_redis
  VALUES("James", 35), ("Michael", 22);
```

1.20.3 Querying the DCS Table

This statement is used to query data in a DCS table.

Syntax

```
SELECT * FROM table_name LIMIT number;
```

Keyword

LIMIT is used to limit the query results. Only INT type is supported by the **number** parameter.

Example

Query data in the **test_redis** table.

```
SELECT * FROM test_redis limit 100;
```


1.21 Creating a Datasource Connection with a DDS Table

1.21.1 Creating a Table and Associating It with DDS

Function

This statement is used to create a table and associate it with an existing DDS collection.

Prerequisites

Before creating a table and associating it with DDS, you need to create a datasource connection and bind it to a queue. For details about operations on the management console, see

Syntax

```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME(
    FIELDNAME1 FIELDTYPE1,
    FIELDNAME2 FIELDTYPE2)
  USING MONGO OPTIONS (
  'url'='IP:PORT[,IP:PORT]/[DATABASE][.COLLECTION][AUTH_PROPERTIES]',
  'database'='xx',
  'collection'='xx',
  'passwdauth' = 'xxx',
  'encryption' = 'true'
);
```

Keyword

Table 1-54 CREATE TABLE parameter description

Parameter	Description
url	Before obtaining the DDS IP address, you need to create a datasource connection first.. After creating an enhanced datasource connection, use the random connection address provided by DDS. The format is as follows: "IP:PORT[,IP:PORT]/[DATABASE][.COLLECTION][AUTH_PROPERTIES]" Example: "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
database	DDS database name. If the database name is specified in the URL, the database name in the URL does not take effect.
collection	Collection name in the DDS. If the collection is specified in the URL, the collection in the URL does not take effect.

Parameter	Description
user	(Discarded) Username for accessing the DDS cluster.
password	(Discarded) Password for accessing the DDS cluster.
passwdauth	Datasource password authentication name. For details about how to create datasource authentication, see in the *Data Lake Insight User Guide*.
encryption	Set this parameter to **true** when datasource password authentication is used.

NOTE

If a collection already exists in DDS, you do not need to specify schema information when creating a table. DLI automatically generates schema information based on data in the collection.

Example

```
create table 1_datasource_mongo.test_mongo(id string, name string, age int) using mongo options(
  'url' = '192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin',
  'database' = 'test',
  'collection' = 'test',
  'passwdauth' = 'xxx',
  'encryption' = 'true');
```

1.21.2 Inserting Data to the DDS Table

Function

This statement is used to insert data in a table to the associated DDS table.

Syntax

- Insert the SELECT query result into a table.
```
INSERT INTO DLI_TABLE
  SELECT field1,field2...
  [FROM DLI_TEST]
  [WHERE where_condition]
  [LIMIT num]
  [GROUP BY field]
  [ORDER BY field] ...;
```
- Insert a data record into a table.
```
INSERT INTO DLI_TABLE
  VALUES values_row [, values_row ...];
```
- Overwriting the inserted data
```
INSERT OVERWRITE TABLE DLI_TABLE
  SELECT field1,field2...
  [FROM DLI_TEST]
  [WHERE where_condition]
  [LIMIT num]
  [GROUP BY field]
  [ORDER BY field] ...;
```

Keywords

For details about the SELECT keywords, see **Basic SELECT Statements**.

Parameter description

Table 1-55 Parameter description

Parameter	Description
DLI_TABLE	Name of the table for which a datasource connection has been created.
DLI_TEST	indicates the table that contains the data to be queried.
field1,field2..., field	Column values in the DLI_TEST table must match the column values and types in the DLI_TABLE table.
where_condition	Query condition.
num	Limit the query result. The num parameter supports only the INT type.
values_row	Value to be inserted to a table. Use commas (,) to separate columns.

Precautions

The target table must exist.

Example

- Query data in the user table and insert the data into the test table.
```
INSERT INTO test
  SELECT ATTR_EXPR
  FROM user
  WHERE user_name='cyz'
  LIMIT 3
  GROUP BY user_age
```
- Insert data 1 into the test table.
```
INSERT INTO test
  VALUES (1);
```

1.21.3 Querying the DDS Table

This statement is used to query data in a DDS table.

Syntax

```
SELECT * FROM table_name LIMIT number;
```

Keyword

LIMIT is used to limit the query results. Only INT type is supported by the **number** parameter.

Precautions

If schema information is not specified during table creation, the query result contains the **_id** field for storing **_id** in the DOC file. For example:

Example

Query data in the **test_mongo** table.

```
SELECT * FROM test_mongo limit 100;
```

1.22 Views

1.22.1 Creating a View

Function

This statement is used to create views.

Syntax

```
CREATE [OR REPLACE] VIEW view_name AS select_statement;
```

Keyword

- CREATE VIEW: creates views based on the given select statement. The result of the select statement will not be written into the disk.
- OR REPLACE: updates views using the select statement. No error is reported and the view definition is updated using the SELECT statement if a view exists.

Precautions

- The view to be created must not exist in the current database. Otherwise, an error will be reported. When the view exists, you can add keyword **OR REPLACE** to avoid the error message.
- The table or view information contained in the view cannot be modified. If the table or view information is modified, the query may fail.

Example

To create a view named **student_view** for the queried ID and name of the **student** table, run the following statement:

```
CREATE VIEW student_view AS SELECT id, name FROM student;
```

1.22.2 Deleting a View

Function

This statement is used to delete views.

Syntax

```
DROP VIEW [IF EXISTS] [db_name.]view_name;
```

Keyword

DROP: Deletes the metadata of a specified view. Although views and tables have many common points, the DROP TABLE statement cannot be used to delete views.

Precautions

The to-be-deleted view must exist. If you run this statement to delete a view that does not exist, an error is reported. To avoid such an error, you can add **IF EXISTS** in this statement.

Example

To delete a view named **student_view**, run the following statement:

```
DROP VIEW student_view;
```

1.23 Viewing the Execution Plan

Function

This statement returns the logical plan and physical execution plan for the SQL statement.

Syntax

```
EXPLAIN [EXTENDED | CODEGEN] statement;
```

Keyword

EXTENDED: After this keyword is specified, the logical and physical plans are outputted at the same time.

CODEGEN: After this keyword is specified, code generated by using the Codegen is also outputted.

Precautions

None

Example

To return the logical and physical plans of **SELECT * FROM test**, run the following statement:

```
EXPLAIN EXTENDED select * from test;
```

1.24 Data Permissions Management

1.24.1 Data Permissions List

Table 1-56 describes the SQL statement permission matrix in DLI in terms of permissions on databases, tables, and roles.

Table 1-56 Permission matrix

Category	SQL statement	Permission	Description
Database	DROP DATABASE db1	The **DROP_DATABASE** permission of **database.db1**	-
	CREATE TABLE tb1(...)	The **CREATE_TABLE** permission of **database.db1**	-
	CREATE VIEW v1	The **CREATE_VIEW** permission of **database.db1**	-
	EXPLAIN query	The **EXPLAIN** permission of **database.db1**	Depending on the permissions required by **query** statements.
Table	SHOW CREATE TABLE tb1	The **SHOW_CREATE_TABLE** permission of **database.db1.tables.tb1**	-
	DESCRIBE [EXTENDED|FORMATTED] tb1	The **DESCRIBE_TABLE** permission of **databases.db1.tables.tb1**	-
	DROP TABLE [IF EXISTS] tb1	The **DROP_TABLE** permission of **database.db1.tables.tb1**	-
	SELECT * FROM tb1	The **SELECT** permission of **database.db1.tables.tb1**	-
	SELECT count(*) FROM tb1	The **SELECT** permission of **database.db1.tables.tb1**	-
	SELECT * FROM view1	The **SELECT** permission of **database.db1.tables.view1**	-
	SELECT count(*) FROM view1	The **SELECT** permission of **database.db1.tables.view1**	-
	LOAD DLI TABLE	The **INSERT_INTO_TABLE** permission of **database.db1.tables.tb1**	-

Category	SQL statement	Permission	Description
	INSERT INTO TABLE	The **INSERT_INTO_TABLE** permission of **database.db1.tables.tb1**	-
	INSERT OVERWRITE TABLE	The **INSERT_OVERWRITE_TABLE** permission of **database.db1.tables.tb1**	-
	ALTER TABLE ADD COLUMNS	The **ALTER_TABLE_ADD_COLUMNS** permission of **database.db1.tables.tb1**	-
	ALTER TABLE RENAME	The **ALTER_TABLE_RENAME** permission of **database.db1.tables.tb1**	-
ROLE&PRIVILEGE	CREATE ROLE	The **CREATE_ROLE** permission of **db**	-
	DROP ROLE	The **DROP_ROLE** permission of **db**	-
	SHOW ROLES	The **SHOW_ROLES** permission of **db**	-
	GRANT ROLES	The **GRANT_ROLE** permission of **db**	-
	REVOKE ROLES	The **REVOKE_ROLE** permission of **db**	-
	GRANT PRIVILEGE	The **GRANT_PRIVILEGE** permission of **db** or **table**	-
	REVOKE PRIVILEGE	The **REVOKE_PRIVILEGE** permission of **db** or **table**	-
	SHOW GRANT	The **SHOW_GRANT** permission of **db** or **table**	-

For privilege granting or revocation on databases and tables, DLI supports the following permissions:

- Permissions that can be assigned or revoked on databases are as follows:
  - DROP_DATABASE (Deleting a database)
  - CREATE_TABLE (Creating a table)
  - CREATE_VIEW (Creating a view)
  - EXPLAIN (Explaining a SQL statement as an execution plan)

  - CREATE_ROLE (Creating a role)
  - DROP_ROLE (Deleting a role)
  - SHOW_ROLES (Displaying a role)
  - GRANT_ROLE (Bounding a role)
  - REVOKE_ROLE (Unbinding a role)
  - DESCRIBE_TABLE (Describing a table)
  - DROP_TABLE (Deleting a table)
  - Select (Querying a table)
  - INSERT_INTO_TABLE (Inserting)
  - INSERT_OVERWRITE_TABLE (Overwriting)
  - GRANT_PRIVILEGE (Granting permissions to a database)
  - REVOKE_PRIVILEGE (Revoking permissions from a database)
  - SHOW_PRIVILEGES (Viewing the database permissions of other users)
  - ALTER_TABLE_ADD_PARTITION (Adding partitions to a partitioned table)
  - ALTER_TABLE_DROP_PARTITION (Deleting partitions from a partitioned table)
  - ALTER_TABLE_RENAME_PARTITION (Renaming table partitions)
  - ALTER_TABLE_RECOVER_PARTITION (Restoring table partitions)
  - ALTER_TABLE_SET_LOCATION (Setting the path of a partition)
  - SHOW_PARTITIONS (Displaying all partitions)
  - SHOW_CREATE_TABLE (Viewing table creation statements)
- Permissions that can be assigned or revoked on tables are as follows:
  - DESCRIBE_TABLE (Describing a table)
  - DROP_TABLE (Deleting a table)
  - Select (Querying a table)
  - INSERT_INTO_TABLE (Inserting)
  - INSERT_OVERWRITE_TABLE (Overwriting)
  - GRANT_PRIVILEGE (Granting permissions to a table)
  - REVOKE_PRIVILEGE (Revoking permissions from a table)
  - SHOW_PRIVILEGES (Viewing the table permissions of other users)
  - ALTER_TABLE_ADD_COLUMNS (Adding a column)
  - ALTER_TABLE_RENAME (Renaming a table)
  - ALTER_TABLE_ADD_PARTITION (Adding partitions to a partitioned table)
  - ALTER_TABLE_DROP_PARTITION (Deleting partitions from a partitioned table)
  - ALTER_TABLE_RENAME_PARTITION (Renaming table partitions)
  - ALTER_TABLE_RECOVER_PARTITION (Restoring table partitions)
  - ALTER_TABLE_SET_LOCATION (Setting the path of a partition)
  - SHOW_PARTITIONS (Displaying all partitions)
  - SHOW_CREATE_TABLE (Viewing table creation statements)

1.24.2 Creating a Role

Function

- This statement is used to create a role in the current database or a specified database.
- Only users with the CREATE_ROLE permission on the database can create roles. For example, the administrator, database owner, and other users with the CREATE_ROLE permission.
- Each role must belong to only one database.

Syntax

```
CREATE ROLE [db_name].role_name;
```

Keyword

None

Precautions

- The **role_name** to be created must not exist in the current database or the specified database. Otherwise, an error will be reported.
- If **db_name** is not specified, the role is created in the current database.

Example

```
CREATE ROLE role1;
```

1.24.3 Deleting a Role

Function

This statement is used to delete a role in the current database or a specified database.

Syntax

```
DROP ROLE [db_name].role_name;
```

Keyword

None

Precautions

- The **role_name** to be deleted must exist in the current database or the specified database. Otherwise, an error will be reported.
- If **db_name** is not specified, the role is deleted in the current database.

Example

```
DROP ROLE role1;
```

1.24.4 Binding a Role

Function

This statement is used to bind a user with a role.

Syntax

```
GRANT ([db_name].role_name,...) TO (user_name,...);
```

Keyword

None

Precautions

The **role_name** and **username** must exist. Otherwise, an error will be reported.

Example

```
GRANT role1 TO user_name1;
```

1.24.5 Unbinding a Role

Function

This statement is used to unbind the user with the role.

Syntax

```
REVOKE ([db_name].role_name,...) FROM (user_name,...);
```

Keyword

None

Precautions

role_name and user_name must exist and user_name has been bound to role_name.

Example

To unbind the user_name1 from role1, run the following statement:

```
REVOKE role1 FROM user_name1;
```

1.24.6 Displaying a Role

Function

This statement is used to display all roles or roles bound to the **user_name** in the current database.

Syntax

```
SHOW [ALL] ROLES [user_name];
```

Keyword

ALL: Displays all roles.

Precautions

Keywords ALL and user_name cannot coexist.

Example

- To display all roles bound to the user, run the following statement:
  ```
  SHOW ROLES;
  ```
- To display all roles in the project, run the following statement:
  ```
  SHOW ALL ROLES;
  ```

  NOTE

  Only the administrator has the permission to run the **show all roles** statement.
- To display all roles bound to the user named **user_name1**, run the following statement:
  ```
  SHOW ROLES user_name1;
  ```

1.24.7 Granting a Permission

Function

This statement is used to grant permissions to a user or role.

Syntax

```
GRANT (privilege,...) ON (resource,..) TO ((ROLE [db_name].role_name) | (USER user_name)),...);
```

Keyword

ROLE: The subsequent **role_name** must be a role.

USER: The subsequent **user_name** must be a user.

Precautions

- The privilege must be one of the authorizable permissions. If the object has the corresponding permission on the resource or the upper-level resource, the permission fails to be granted. For details about the permission types supported by the privilege, see **Data Permissions List**.
- The resource can be a queue, database, table, view, or column. The formats are as follows:
  - Queue format: queues.queue_name

    The following table lists the permission types supported by a queue.

Operation	Description
DROP_QUEUE	Deleting a queue
SUBMIT_JOB	Submitting a job
CANCEL_JOB	Cancel a job
RESTART	Restarting a queue
SCALE_QUEUE	Scaling out/in a queue
GRANT_PRIVILEGE	Granting queue permissions
REVOKE_PRIVILEGE	Revoking queue permissions
SHOW_PRIVILEGES	Viewing queue permissions of other users

- Database format: databases.db_name

  For details about the permission types supported by a database, see **Data Permissions List**.
- Table format: databases.db_name.tables.table_name

  For details about the permission types supported by a table, see **Data Permissions List**.
- View format: databases.db_name.tables.view_name

  Permission types supported by a view are the same as those supported by a table. For details, see table permissions in **Data Permissions List**.
- Column format: databases.db_name.tables.table_name.columns.column_name

  Columns support only the SELECT permission.

Example

Run the following statement to grant user_name1 the permission to delete the **db1** database:

```
GRANT DROP_DATABASE ON databases.db1 TO USER user_name1;
```

Run the following statement to grant user_name1 the SELECT permission of data table **tb1** in the **db1** database:

```
GRANT SELECT ON databases.db1.tables.tb1 TO USER user_name1;
```

Run the following statement to grant **role_name** the SELECT permission of data table **tb1** in the **db1** database:

```
GRANT SELECT ON databases.db1.tables.tb1 TO ROLE role_name;
```

1.24.8 Revoking a Permission

Function

This statement is used to revoke permissions granted to a user or role.

Syntax

```
REVOKE (privilege,...) ON (resource,..) FROM ((ROLE [db_name].role_name) | (USER user_name)),...);
```

Keyword

ROLE: The subsequent **role_name** must be a role.

USER: The subsequent **user_name** must be a user.

Precautions

- The privilege must be the granted permissions of the authorized object in the resource. Otherwise, the permission fails to be revoked. For details about the permission types supported by the privilege, see **Data Permissions List**.
- The resource can be a queue, database, table, view, or column. The formats are as follows:
  - Queue format: queues.queue_name
  - Database format: databases.db_name
  - Table format: databases.db_name.tables.table_name
  - View format: databases.db_name.tables.view_name
  - Column format: databases.db_name.tables.table_name.columns.column_name

Example

To revoke the permission of user **user_name1** to delete database **db1**, run the following statement:

```
REVOKE DROP_DATABASE ON databases.db1 FROM USER user_name1;
```

To revoke the SELECT permission of user **user_name1** on table **tb1** in database **db1**, run the following statement:

```
REVOKE SELECT ON databases.db1.tables.tb1 FROM USER user_name1;
```

To revoke the SELECT permission of role **role_name** on table **tb1** in database **db1**, run the following statement:

```
REVOKE SELECT ON databases.db1.tables.tb1 FROM ROLE role_name;
```

1.24.9 Displaying the Granted Permissions

Function

This statement is used to show the permissions granted to a user or role in the resource.

Syntax

```
SHOW GRANT ((ROLE [db_name].role_name) | (USER user_name)) ON resource;
```

Keyword

ROLE: The subsequent **role_name** must be a role.

USER: The subsequent **user_name** must be a user.

Precautions

The resource can be a queue, database, table, view, or column. The formats are as follows:

- Queue format: queues.queue_name
- Database format: databases.db_name
- Table format: databases.db_name.tables.table_name
- Column format: databases.db_name.tables.table_name.columns.column_name
- View format: databases.db_name.tables.view_name

Example

Run the following statement to show permissions of **user_name1** in the **db1** database:

```
SHOW GRANT USER user_name1 ON databases.db1;
```

Run the following statement to show permissions of **role_name** on table **tb1** in the **db1** database:

```
SHOW GRANT ROLE role_name ON databases.db1.tables.tb1;
```

1.24.10 Displaying the Binding Relationship Between All Roles and Users

Function

This statement is used to display the binding relationship between roles and a user in the current database.

Syntax

```
SHOW PRINCIPALS ROLE;
```

Keyword

None

Precautions

The ROLE variable must exist.

Example

```
SHOW PRINCIPALS role1;
```

1.25 Data Types

1.25.1 Overview

Data type is a basic attribute of data. It is used to distinguish different types of data. Different data types occupy different storage space and support different operations. Data is stored in data tables in the database. A data type is specified for each column of a data table. Therefore, data to be stored in a data table must comply with the attribute of the specific data type. Otherwise, errors may occur.

DLI only supports primitive data types.

1.25.2 Primitive Data Types

Table 1-57 lists the primitive data types supported by DLI.

Table 1-57 Primitive data types

Data Type	Description	Storage Space	Value Range	Support by OBS Table
INT	Signed integer	4 bytes	–2147483648 to 2147483647	Yes
STRING	Character string	-	-	Yes
FLOAT	Single-precision floating point	4 bytes	-	Yes
DOUBLE	Double-precision floating-point	8 bytes	-	Yes
DECIMAL(precision,scale)	Decimal number. Data type of valid fixed places and decimal places, for example, 3.5. • **precision**: indicates the maximum number of digits that can be displayed. • **scale**: indicates the number of decimal places.	-	1<=precision<=38 0<=scale<=38 If **precision** and **scale** are not specified, **DECIMAL (38,38)** is used by default.	Yes
BOOLEAN	Boolean	1 byte	TRUE/FALSE	Yes
SMALLINT/SHORT	Signed integer	2 bytes	-32768~32767	Yes
TINYINT	Signed integer	1 byte	-128~127	Yes

Data Type	Description	Storage Space	Value Range	Support by OBS Table
BIGINT/ LONG	Signed integer	8 bytes	–9223372036854775808 to 9223372036854775807	Yes
TIMESTAMP	Timestamp in raw data format, indicating the date and time Example: 1621434131222	-	-	Yes
CHAR	Fixed-length character string	-	-	Yes
VARCHAR	Variable-length character string	-	-	Yes
DATE	Date type in the format of ***yyyy-mm-dd***, for example, **2014-05-29**	-	**DATE** does not contain time information. Its value ranges from **0000-01-01** to **9999-12-31**.	Yes

NOTE

- VARCHAR and CHAR data is stored in STRING type on DLI. Therefore, the string that exceeds the specified length will not be truncated.
- FLOAT data is stored as DOUBLE data on DLI.

INT

Signed integer with a storage space of 4 bytes. Its value ranges from –2147483648 to 2147483647. If this field is NULL, value 0 is used by default.

STRING

Character string.

FLOAT

Single-precision floating point with a storage space of 4 bytes. If this field is NULL, value 0 is used by default.

Due to the limitation of storage methods of floating point data, do not use the formula a==b to check whether two floating point values are the same. You are advised to use the formula: absolute value of (a-b) <= EPSILON. EPSILON indicates the allowed error range which is usually 1.19209290E-07F. If the formula is satisfied, the compared two floating point values are considered the same.

DOUBLE

Double-precision floating point with a storage space of 8 bytes. If this field is NULL, value 0 is used by default.

Due to the limitation of storage methods of floating point data, do not use the formula a==b to check whether two floating point values are the same. You are advised to use the formula: absolute value of (a-b) <= EPSILON. EPSILON indicates the allowed error range which is usually 2.2204460492503131E-16. If the formula is satisfied, the compared two floating point values are considered the same.

DECIMAL

Decimal(p,s) indicates that the total digit length is **p**, including **p – s** integer digits and **s** fractional digits. **p** indicates the maximum number of decimal digits that can be stored, including the digits to both the left and right of the decimal point. The value of **p** ranges from 1 to 38. **s** indicates the maximum number of decimal digits that can be stored to the right of the decimal point. The fractional digits must be values ranging from 0 to **p**. The fractional digits can be specified only after significant digits are specified. Therefore, the following inequality is concluded: 0 ≤ **s** ≤ **p**. For example, decimal (10,6) indicates that the value contains 10 digits, in which there are four integer digits and six fractional digits.

BOOLEAN

Boolean, which can be **TRUE** or **FALSE**.

SMALLINT/SHORT

Signed integer with a storage space of 2 bytes. Its value ranges from –32768 to 32767. If this field is NULL, value 0 is used by default.

TINYINT

Signed integer with a storage space of 1 byte. Its value ranges from –128 to 127. If this field is NULL, value 0 is used by default.

BIGINT/LONG

Signed integer with a storage space of 8 bytes. Its value ranges from –9223372036854775808 to 9223372036854775807. It does not support scientific notation. If this field is NULL, value 0 is used by default.

TIMESTAMP

Legacy UNIX TIMESTAMP is supported, providing the precision up to the microsecond level. **TIMESTAMP** is defined by the difference between the specified time and UNIX epoch (UNIX epoch time: 1970-01-01 00:00:00) in seconds. Data of the STRING type supports implicit conversion to TIMESTAMP. (The STRING must in the yyyy-MM-dd HH:MM:SS[.ffffff] format. The precision after the decimal point is optional.)

CHAR

Character string with a fixed length. In DLI, the STRING type is used.

VARCHAR

VARCHAR is declared with a length that indicates the maximum number of characters in a string. During conversion from **STRING** to **VARCHAR**, if the number of characters in **STRING** exceeds the specified length, the excess characters of **STRING** are automatically trimmed. Similar to **STRING**, the spaces at the end of **VARCHAR** are meaningful and affect the comparison result. In DLI, the STRING type is used.

DATE

DATE supports only explicit conversion (cast) with **DATE**, **TIMESTAMP**, and **STRING**. For details, see **Table 1-58**.

Table 1-58 cast function conversion

Explicit Conversion	Conversion Result
cast(date as date)	Same as value of **DATE**.
cast(timestamp as date)	The date (yyyy-mm-dd) is obtained from **TIMESTAMP** based on the local time zone and returned as the value of **DATE**.
cast(string as date)	If the STRING is in the **yyyy-MM-dd** format, the corresponding date (yyyy-mm-dd) is returned as the value of **DATE**. If the STRING is not in the **yyyy-MM-dd** format, **NULL** is returned.
cast(date as timestamp)	Timestamp that maps to the zero hour of the date (yyyy-mm-dd) specified by **DATE** is generated based on the local time zone and returned as the value of **DATE**.
cast(date as string)	A STRING in the **yyyy-MM-dd** format is generated based on the date (yyyy-mm-dd) specified by **DATE** and returned as the value of **DATE**.

1.25.3 Complex Data Types

Spark SQL supports complex data types, as shown in **Table 1-59**.

Table 1-59 Complex data types

Data Type	Description	Syntax
ARRAY	A set of ordered fields that construct an ARRAY with the specified values. The value can be of any type and the data type of all fields must be the same.	array(<value>,<value>[, ...]) For details, see **Example of ARRAY**.
MAP	A group of unordered key/value pairs used to generate a MAP. The key must be native data type, but the value can be either native data type or complex data type. The type of the same MAP key, as well as the MAP value, must be the same.	map(K <key1>, V <value1>, K <key2>, V <value2>[, ...]) For details, see **Example of Map**.
STRUCT	Indicates a group of named fields. The data types of the fields can be different.	struct(<value1>,<value2>[, ...]) For details, see **Example of STRUCT**.

Restrictions

- When a table containing fields of the complex data type is created, the storage format of this table cannot be CSV (txt).
- If a table contains fields of the complex data type, data in CSV (txt) files cannot be imported to the table.
- When creating a table of the MAP data type, you must specify the schema and do not support the **date**, **short**, and **timestamp** data types.
- For the OBS table in JSON format, the key type of the MAP supports only the STRING type.
- The key of the MAP type cannot be **NULL**. Therefore, the MAP key does not support implicit conversion between inserted data formats where NULL values are allowed. For example, the STRING type cannot be converted to other native types, the FLOAT type cannot be converted to the TIMESTAMP type, and other native types cannot be converted to the DECIMAL type.
- Values of the **double** or **boolean** data type cannot be included in the **STRUCT** data type does not support the.

Example of ARRAY

Create an **array_test** table, set **id** to **ARRAY<INT>**, and **name** to **STRING**. After the table is created, insert test data into **array_test**. The procedure is as follows:

1. Create a table.
   CREATE TABLE array_test(name STRING, id ARRAY < INT >) USING PARQUET;
2. Run the following statements to insert test data:
   INSERT INTO array_test VALUES ('test',array(1,2,3,4));

INSERT INTO array_test VALUES ('test2',array(4,5,6,7))

INSERT INTO array_test VALUES ('test3',array(7,8,9,0));

3. Query the result.

   To query all data in the **array_test** table, run the following statement:

   SELECT * FROM array_test;

```
test3    [7,8,9,0]
test2    [4,5,6,7]
test    [1,2,3,4]
```

   To query the data of element **0** in the **id** array in the **array_test** table, run the following statement:

   SELECT id[0] FROM array_test;

```
7
4
1
```

Example of Map

Create the **map_test** table and set **score** to **map<STRING,INT>**. The key is of the **STRING** type and the value is of the **INT** type. After the table is created, insert test data to **map_test**. The procedure is as follows:

1. Create a table.

   CREATE TABLE map_test(id STRING, score map<STRING,INT>) USING PARQUET;

2. Run the following statements to insert test data:

   INSERT INTO map_test VALUES ('test4',map('math',70,'chemistry',84));

   INSERT INTO map_test VALUES ('test5',map('math',85,'chemistry',97));

   INSERT INTO map_test VALUES ('test6',map('math',88,'chemistry',80));

3. Query the result.

   To query all data in the **map_test** table, run the following statement:

   SELECT * FROM map_test;

```
test6    {"chemistry":80,"math":88}
test5    {"chemistry":97,"math":85}
test4    {"chemistry":84,"math":70}
```

   To query the math score in the **map_test** table, run the following statement:

   SELECT id, score['Math'] FROM map_test;

```
test6    88
test5    85
test4    70
```

Example of STRUCT

Create a **struct_test** table and set **info** to the **STRUCT<name:STRING, age:INT>** data type (the field consists of **name** and **age**, where the type of **name** is **STRING** and **age** is **INT**). After the table is created, insert test data into the **struct_test** table. The procedure is as follows:

1. Create a table.

   CREATE TABLE struct_test(id INT, info STRUCT<name:STRING,age:INT>) USING PARQUET;

2. Run the following statements to insert test data:

   INSERT INTO struct_test VALUES (8, struct('zhang',23));

   INSERT INTO struct_test VALUES (9, struct('li',25));

   INSERT INTO struct_test VALUES (10, struct('wang',26));

3. Query the result.

   To query all data in the **struct_test** table, run the following statement:

   SELECT * FROM struct_test;

```
8   {"name":"zhang","age":23}
10   {"name":"wang","age":26}
9   {"name":"li","age":25}
```

   Query **name** and **age** in the **struct_test** table.

   SELECT id,info.name,info.age FROM struct_test;

```
8   zhang   23
10   wang   26
9   li   25
```

1.26 User-Defined Functions

1.26.1 Creating a Function

Function

DLI allows you to create and use user-defined functions (UDF) and user-defined table functions (UDTF) in Spark jobs.

Syntax

```
CREATE [TEMPORARY] FUNCTION [db_name.]function_name AS class_name
  [USING resource,...]

resource:
  : (JAR|FILE|ARCHIVE)file_uri
```

Precautions

- If a function with the same name exists in the database, the system reports an error.
- Only the Hive syntax can be used to create functions.
- If you specify the same class name for two UDFs, the functions conflict though the package names are different. Avoid this problem because it causes failure of job execution.

Keywords

- TEMPORARY: The created function is available only in the current session and is not persisted to the underlying metabase, if any. The database name cannot be specified for a temporary function.
- USING <resources>: resources to be loaded. It can be a list of JARs, files, or URIs.

Example

Create the mergeBill function.

```
CREATE FUNCTION mergeBill AS 'com.xxx.hiveudf.MergeBill'
  using jar 'obs://onlyci-7/udf/MergeBill.jar';
```

1.26.2 Deleting a Function

Function

This statement is used to delete functions.

Syntax

```
DROP [TEMPORARY] FUNCTION [IF EXISTS] [db_name.] function_name;
```

Keywords

- TEMPORARY: Indicates whether the function to be deleted is a temporary function.
- IF EXISTS: Used when the function to be deleted does not exist to avoid system errors.

Precautions

- An existing function is deleted. If the function to be deleted does not exist, the system reports an error.
- Only the HIVE syntax is supported.

Example

The mergeBill function is deleted.

```
DROP FUNCTION mergeBill;
```

1.26.3 Displaying Function Details

Function

Displays information about a specified function.

Syntax

```
DESCRIBE FUNCTION [EXTENDED] [db_name.] function_name;
```

Keywords

EXTENDED: displays extended usage information.

Precautions

The metadata (implementation class and usage) of an existing function is returned. If the function does not exist, the system reports an error.

Example

Displays information about the mergeBill function.

```
DESCRIBE FUNCTION mergeBill;
```

1.26.4 Displaying All Functions

Function

View all functions in the current project.

Syntax

```
SHOW [USER|SYSTEM|ALL] FUNCTIONS ([LIKE] regex | [db_name.] function_name);
```

In the preceding statement, regex is a regular expression. For details about its parameters, see **Table 1-60**.

Table 1-60 Parameter examples

Expression	Description
'xpath*'	Matches all functions whose names start with **xpath**. Example: SHOW FUNCTIONS LIKE'xpath* ; Matches functions whose names start with **xpath**, including **xpath**, **xpath_int**, and **xpath_string**.
'x[a-z]+'	Matches functions whose names start with **x** and is followed by one or more characters from a to z. For example, **xpath** and **xtest** can be matched.
'x.*h'	Matches functions whose names start with **x**, end with **h**, and contain one or more characters in the middle. For example, **xpath** and **xtesth** can be matched.

For details about other expressions, see the official website.

Keywords

LIKE: This qualifier is used only for compatibility and has no actual effect.

Precautions

The function that matches the given regular expression or function name are displayed. If no regular expression or name is provided, all functions are displayed. If USER or SYSTEM is specified, user-defined Spark SQL functions and system-defined Spark SQL functions are displayed, respectively.

Example

This statement is used to view all functions.

```
SHOW FUNCTIONS;
```

1.27 Built-in Functions

1.27.1 Mathematical Functions

Table 1-61 lists the mathematical functions supported in DLI.

Table 1-61 Mathematical functions

Function	Return Type	Description
round(DOUBLE a)	DOUBLE	Round **a**.
round(DOUBLE a, INT d)	DOUBLE	Round **a** to **d** decimal places. Example: round(21.263,2) = 21.26.
bround(DOUBLE a)	DOUBLE	Round off **a** figure using the HALF_EVEN rounding mode. If the figure to be rounded off ends in **5**, the HALF_EVEN rounding mode is as follows: ● Round up if the digit in the place preceding **5** is odd. ● Round down if the digit in the place preceding **5** is even. Example: bround(7.5) = 8.0, bround(6.5) = 6.0.
bround(DOUBLE a, INT d)	DOUBLE	Retain **d** decimal places and round the d+1 decimal place using the HALF_EVEN rounding mode. If the figure to be rounded off ends in **5**, it will be rounded off as follows: ● Round up if the d decimal digit is odd. ● Round down if the d decimal digit is even. Example: bround(8.25, 1) = 8.2, bround(8.35, 1) = 8.4. Example: bround(8.25, 1) = 8.2, bround(8.35, 1) = 8.4.
floor(DOUBLE a)	BIGINT	Return the largest integer that is less than or equal to **a**. Example: floor(21.2) = 21.
ceil(DOUBLE a), ceiling(DOUBLE a)	BIGINT	Return the smallest integer that is greater than or equal to **a**. Example: ceil(21.2) = 22.
rand(), rand(INT seed)	DOUBLE	Return a random number that is distributed uniformly from 0 through 1 (1 is exclusive). If the seed is specified, a stable random number sequence is displayed.

Function	Return Type	Description
exp(DOUBLE a), exp(DECIMAL a)	DOUBLE	Return the value of e raised to the power of **a**.
ln(DOUBLE a), ln(DECIMAL a)	DOUBLE	Return the natural logarithm of the argument **a**.
log10(DOUBLE a), log10(DECIMAL a)	DOUBLE	Return the base 10 logarithm of the argument **a**.
log2(DOUBLE a), log2(DECIMAL a)	DOUBLE	Return the base 2 logarithm of the argument **a**.
log(DOUBLE base, DOUBLE a) log(DECIMAL base, DECIMAL a)	DOUBLE	Return the base **base** logarithm of the argument **a**.
pow(DOUBLE a, DOUBLE p), power(DOUBLE a, DOUBLE p)	DOUBLE	Return the value of **a** raised to the power of **p**.
sqrt(DOUBLE a), sqrt(DECIMAL a)	DOUBLE	Return the square root of **a**.
bin(BIGINT a)	STRING	Return a number in binary format.
hex(BIGINT a) hex(STRING a)	STRING	Convert an integer or character to its hexadecimal representation.
conv(BIGINT num, INT from_base, INT to_base), conv(STRING num, INT from_base, INT to_base)	STRING	Convert a number from the base **from_base** to the base **to_base**. Example: Convert 5 from decimal to quaternary using conv(5,10,4) = 11.
abs(DOUBLE a)	DOUBLE	Return the absolute value.
pmod(INT a, INT b), pmod(DOUBLE a, DOUBLE b)	INT or DOUBLE	Return the positive value of the remainder after division of **a** by **b**.
sin(DOUBLE a), sin(DECIMAL a)	DOUBLE	Return the sine value of **a**.
asin(DOUBLE a), asin(DECIMAL a)	DOUBLE	Return the arc sine value of **a**.
cos(DOUBLE a), cos(DECIMAL a)	DOUBLE	Return the cosine value of **a**.

Function	Return Type	Description
acos(DOUBLE a), acos(DECIMAL a)	DOUBLE	Return the arc cosine value of **a**.
tan(DOUBLE a), tan(DECIMAL a)	DOUBLE	Return the tangent value of **a**.
atan(DOUBLE a), atan(DECIMAL a)	DOUBLE	Return the arc tangent value of **a**.
degrees(DOUBLE a), degrees(DECIMAL a)	DOUBLE	Convert the value of **a** from radians to degrees.
radians(DOUBLE a), radians(DECIMAL a)	DOUBLE	Convert the value of **a** from degrees to radians.
positive(INT a), positive(DOUBLE a)	INT or DOUBLE	Return **a**. Example: positive(2) = 2.
negative(INT a), negative(DOUBLE a)	INT or DOUBLE	Return **–a**. Example: negative(2) = –2.
sign(DOUBLE a), sign(DECIMAL a)	DOUBLE or INT	Return the sign of **a**. 1.0 is returned if **a** is positive. –1.0 is returned if **a** is negative. Otherwise, 0.0 is returned.
e()	DOUBLE	Return the value of **e**.
pi()	DOUBLE	Return the value of **pi**.
factorial(INT a)	BIGINT	Return the factorial of **a**.
cbrt(DOUBLE a)	DOUBLE	Return the cube root of **a**.
shiftleft(TINYINT|SMALLINT|INT a, INT b) shiftleft(BIGINT a, INT b)	INT BIGINT	Bitwise signed left shift. Interpret **a** as a binary number and shift the binary number **b** positions to the left.
shiftright(TINYINT|SMALLINT|INT a, INT b) shiftright(BIGINT a, INT b)	INT BIGINT	Bitwise signed right shift. Interpret **a** as a binary number and shift the binary number **b** positions to the right.

Function	Return Type	Description
shiftrightunsigned(TINYINT|SMALLINT|INT a, INT b), shiftrightunsigned(BIGINT a, INT b)	INT BIGINT	Bitwise unsigned right shift. Interpret **a** as a binary number and shift the binary number **b** positions to the right.
greatest(T v1, T v2, ...)	T	Return the maximum value of a list of values.
least(T v1, T v2, ...)	T	Return the minimum value of a list of values.

1.27.2 Date Functions

Table 1-62 lists the date functions supported in DLI.

Table 1-62 Date/time functions

Function	Return Type	Description
from_unixtime(bigint unixtime[, string format])	STRING	Convert a timestamp to the time format "yyyy-MM-dd HH:mm:ss" or "yyyyMMddHHmmss.uuuuuu". For example, select FROM_UNIXTIME(1608135036,'yyyy-MM-dd HH:mm:ss').
unix_timestamp()	BIGINT	Return a Unix timestamp (the number of seconds that have elapsed since **1970-01-01 00:00:00**) represented by an unsigned integer when the function is called without arguments.
unix_timestamp(string date)	BIGINT	Return the number of seconds between a specified date and **1970-01-01 00:00:00**.
unix_timestamp(string date, string pattern)	BIGINT	Convert a time string with a given pattern to a Unix timestamp. Example: unix_timestamp("2009-03-20", "yyyy-MM-dd") = 1237532400.
to_date(string timestamp)	STRING	Return the date part of a time string. Example: to_date("1970-01-01 00:00:00") = "1970-01-01".
year(string date)	INT	Return the year part of a date.

Function	Return Type	Description
quarter(string date/timestamp/ string)	INT	Return the quarter of the year for a date, timestamp, or string. Example: quarter('2015-04-01')=2.
month(string date)	INT	Return the month (from 1 to 12) part of a date.
day(string date) dayofmonth(string date)	INT	Return the day part of a date.
hour(string date)	INT	Return the hour (from 0 to 23) part of a date.
minute(string date)	INT	Return the minute (from 0 to 59) part of a date.
second(string date)	INT	Return the second (from 0 to 59) part of a date.
weekofyear(string date)	INT	Return the week number (from 0 to 53) of a date.
datediff(string enddate, string startdate)	INT	Return the number of days from **startdate** to **enddate**.
date_add(string startdate, int days)	STRING	Add a number of days to a date.
date_sub(string startdate, int days)	STRING	Subtract a number of days from a date.
from_utc_timestamp(string timestamp, string timezone)	TIMESTAMP	Convert a UTC timestamp to a timestamp in a given time zone. For example, from_utc_timestamp('1970-01-01 08:00:00','PST') returns 1970-01-01 00:00:00.
to_utc_timestamp(string timestamp, string timezone)	TIMESTAMP	Convert a timestamp in a given time zone to a UTC timestamp. For example, to_utc_timestamp('1970-01-01 00:00:00','PST') returns 1970-01-01 08:00:00.
current_date()	DATE	Return the current date, for example, 2016-07-04.
current_timestamp()	TIMESTAMP	Return the current time, for example, 2016-07-04 11:18:11.685.
add_months(string start_date, int num_months)	STRING	Return the date that is **num_months** after **start_date**.

Function	Return Type	Description
last_day(string date)	STRING	Return the last day of the month to which a date belongs. The returned date is in the format of yyyy-MM-dd, for example, 2015-08-31.
next_day(string start_date, string day_of_week)	STRING	Return the first date that is later than **start_date** and nearest to **day_of_week**. The returned date in the format of yyyy-MM-dd. **day_of_week** specifies a day of a week. For example, the value of **day_of_week** can be Monday or FRIDAY.
trunc(string date, string format)	STRING	Reset the date in a specified format. Supported formats are MONTH/MON/MM and YEAR/YYYY/YY. Example: trunc('2015-03-17', 'MM') = 2015-03-01.
months_between(string date1, string date2)	DOUBLE	Return number of months between dates **date1** and **date2**.
date_format(date/timestamp/string ts, string fmt)	STRING	Return the formatted value of **date/timestamp/string**. The Java SimpleDateFormat format is supported. Example: date_format('2015-04-08', 'y') = '2015'. In the format, **y** indicates the year. **Y** indicates the year when the current week is located. A week starts from Sunday and ends on Saturday. If a week crosses years, this week is counted as the next year.

1.27.3 String Functions

Table 1-63 lists the string functions supported by DLI.

Table 1-63 String functions

Function	Return Type	Description
ascii(string str)	INT	Returns the numeric value of the first character in a string.
concat(string A, string B...)	STRING	Return a string resulting from concatenating the input strings. This function can take any number of input strings.

Function	Return Type	Description
concat_ws(string SEP, string A, string B...)	STRING	Return a string resulting from concatenating the input strings, which are separated by specified separators.
encode(string src, string charset)	BINARY	Encode src in the encoding mode specified by charset.
find_in_set(string str, string strList)	INT	Return the position of the first occurrence of str in strList. If the value of any parameter is **NULL**, **NULL** is returned. If the first parameter contains a comma (,), **0** is returned.
get_json_object(string json_string, string path)	STRING	Parse the JSON object in a specified JSON path. The function will return **NULL** if the JSON object is invalid.
instr(string str, string substr)	INT	Return the position of the first occurrence of substr in str. Return **NULL** if **NULL** is contained in the parameters and return **0** if substr does not exist in str. Note that the subscripts start from 1.
length(string A)	INT	Return the length of a string.
locate(string substr, string str[, int pos])	INT	Return the position of the first occurrence of substr in str after position pos (starting from 1).
lower(string A) lcase(string A)	STRING	Convert all characters of a string to lower case.
lpad(string str, int len, string pad)	STRING	Return a string of a specified length. If the length of the given string (str) is shorter than the specified length (len), the given string is left-padded with pad to the specified length.
ltrim(string A)	STRING	Trim spaces from the left hand side of a string.

Function	Return Type	Description
parse_url(string urlString, string partToExtract [, string keyToExtract])	STRING	Return the specified part of the specified URL. Valid values of partToExtract include HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, and USERINFO. For example, **parse_url ('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1 ',' HOST ')** returns **'facebook.com'**. When the second parameter is QUERY, the third parameter can be used to extract the value of a specific parameter. For example, **parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1')** returns **'v1'**.
printf(String format, Obj... args)	STRING	Print the input according to a specified format.
regexp_extract(string subject, string pattern, int index)	STRING	Extract the string specified by the regular expression. **regexp_extract ('foothebar ',' foo (.*?) (bar) '2)** returns **'bar.'**
regexp_replace(string A, string B, string C)	STRING	Replace character B in string A with character C.
repeat(string str, int n)	STRING	Repeat a string *N* times.
reverse(string A)	STRING	Return the reversed string.
rpad(string str, int len, string pad)	STRING	Return a string of a specified length. If the length of the given string (str) is shorter than the specified length (len), the given string is right-padded with pad to the specified length.
rtrim(string A)	STRING	Trim spaces from the right hand side of a string.
space(int n)	STRING	Returns a specified number of spaces.
substr(string A, int start) substring(string A, int start)	STRING	Return the substring starting from the specified start position in string A till the end of the string.

Function	Return Type	Description
substr(string A, int start, int len) substring(string A, int start, int len)	STRING	Return the substring of a specified length starting from the specified start position in A string.
substring_index(string A, string delim, int count)	STRING	Return the substring from string A before count occurrences of the delimiter delim.
translate(string|char|varchar input, string|char|varchar from, string|char|varchar to)	STRING	Translate the input string by replacing the characters or string specified by from with the characters or string specified by to. For example, replace bcd in abcde with BCD using **translate ("abcde", "bcd", "BCD")**.
trim(string A)	STRING	Trim spaces from both ends of a string.
upper(string A) ucase(string A)	STRING	Convert all characters of a string to upper case.
initcap(string A)	STRING	Convert the first letter of each word of a string to upper case and all other letters to lower case.
levenshtein(string A, string B)	INT	Return the Levenshtein distance between two strings. Example: **levenshtein ('kitten ',' sitting ') = 3**.
soundex(string A)	STRING	Return the soundex string from str. Example: **soundex ('Miller ') = M460**.

1.27.4 Aggregate Functions

An aggregate function performs a calculation operation on a set of input values and returns a value. For example, the COUNT function counts the number of rows retrieved by an SQL statement. **Table 1-64** lists aggregate functions.

Table 1-64 Aggregate functions

Function	Return Type	Description
count(*), count(expr), count(DISTINCT expr[, expr...])	BIGINT	Return the total number of retrieved records.
sum(col), sum(DISTINCT col)	DOUBLE	Return the sum of the values in a column.

Function	Return Type	Description
avg(col), avg(DISTINCT col)	DOUBLE	Return the average of the values in a column.
min(col)	DOUBLE	Return the minimum value of a column.
max(col)	DOUBLE	Return the maximum value of a column.
variance(col), var_pop(col)	DOUBLE	Return the variance of a numeric column.
var_samp(col)	DOUBLE	Return the sample variance of a numeric column.
stddev_pop(col)	DOUBLE	Return the deviation of a numeric column.
stddev_samp(col)	DOUBLE	Return the sample deviation of a numeric column.
covar_pop(col1, col2)	DOUBLE	Return the covariance of a pair of numeric columns.
covar_samp(col1, col2)	DOUBLE	Return the sample covariance of a pair of numeric columns.
corr(col1, col2)	DOUBLE	Return the coefficient of correlation of a pair of numeric columns.
percentile(BIGINT col, p)	DOUBLE	Return the exact pth percentile of a column. **p** must be between 0 and 1. Otherwise, this function returns null. This function does not work with floating point types.
percentile_approx(DOUBLE col, p [, B])	DOUBLE	Return an approximate pth percentile of a numeric column (including floating point types) in a group. **p** must be between 0 and 1. **B** controls approximation accuracy. Higher values of **B** mean better approximations, and the default value is 10,000. When the number of distinct values in the numeric column is smaller than **B**, an exact percentile value is returned.

NOTE

Functions such as var_pop, stddev_pop, var_samp, stddev_samp, covar_pop, covar_samp, corr, and percentile_approx, do not support non-numeral data types, such as TimeStamp.

1.27.5 Window Functions

A window function performs a calculation operation on a set of values related to the current value. A window function can be an aggregate function used in the GROUP BY clause, such as sum, max, min, count, and avg functions. The window functions also include the functions listed in **Table 1-65**. A window contains multiple rows defined by an OVER clause. A window function works on one window.

Table 1-65 Functions

Function	Return Type	Description
first_value(col)	Data type of the argument.	Return the value of the first data record from a column.
last_value(col)	Data type of the argument.	Return the value of the last data record from a column.
lag (col,n,DEFAULT)	Data type of the argument.	Return the value from the *n*th row preceding the current row. The first argument specifies the column name. The second argument specifies the *n*th row preceding the current row. The configuration of the second argument is optional, and the default argument value is **1** if the argument is not specified. The third argument is set to a default value. If the *n*th row preceding the current row is **null**, the default value is used. The default value of the third argument is **NULL** if the argument is not specified.
lead (col,n,DEFAULT)	Data type of the argument.	Return the value from the *n*th row following the current row. The first argument specifies the column name. The second argument specifies the *n*th row following the current row. The configuration of the second argument is optional, and the default argument value is **1** if the argument is not specified. The third argument is set to a default value. If the *n*th row following the current row is **null**, the default value is used. The default value of the third argument is **NULL** if the argument is not specified.
row_number() over (order by col_1[,col_2 ...])	INT	Assign a unique number to each row.
rank()	INT	Return the rank of a value in a set of values. When multiple values share the same rank, the next rank in the sequence is not consecutive.
cume_dist()	DOUBLE	Calculate the relative position of a value in a row.

Function	Return Type	Description
percent_rank()	DOUBLE	Return the rank of a value from the column specified by the ORDER BY clause of the window. The return value is a decimal between 0 and 1, which is calculated using (RANK - 1)/(- 1).

1.28 Basic SELECT Statements

Function

This statement is a basic query statement and is used to return the query results.

Syntax

```
SELECT [ALL | DISTINCT] attr_expr_list FROM table_reference
  [WHERE where_condition]
  [GROUP BY col_name_list]
  [ORDER BY col_name_list][ASC | DESC]
  [CLUSTER BY col_name_list | DISTRIBUTE BY col_name_list]
  [SORT BY col_name_list]]
  [LIMIT number];
```

Keyword

Table 1-66 SELECT parameter description

Parameter	Description
ALL	Returns duplicate rows. By default, all repeated rows are returned. It is followed by asterisks (*) only. Otherwise, an error will occur.
DISTINCT	Removes duplicate rows from the result set.
WHERE	Specifies the filter criteria for a query. Arithmetic operators, relational operators, and logical operators are supported.
where_condition	Filter criteria.
GROUP BY	Specifies the grouping field. Single-field grouping and multi-field grouping are supported.
col_name_list	Field list
ORDER BY	Sort the query results.
ASC/DESC	ASC sorts from the lowest value to the highest value. DESC sorts from the highest value to the lowest value. ASC is the default sort order.

Parameter	Description
CLUSTER BY	CLUSTER BY is used to bucket the table according to the bucketing fields and then sort within the bucketed table. If the field of DISTRIBUTE BY is the same as the field of SORT BY and the sorting is in descending order, the combination of DISTRIBUTE BY and SORT BY achieves the same function as CLUSTER BY.
DISTRIBUTE BY	Specifies the bucketing fields without sorting the table.
SORT BY	The objects will be sorted in the bucket.
LIMIT	LIMIT is used to limit the query results. Only INT type is supported by the **number** parameter.

Precautions

The table to be queried must exist. Otherwise, an error is reported.

Example

To filter the record, in which the name is Mike, from the **student** table and sort the results in ascending order of score, run the following statement:

```
SELECT * FROM student
  WHERE name = 'Mike'
  ORDER BY score;
```

1.29 Filtering

1.29.1 WHERE Filtering Clause

Function

This statement is used to filter the query results using the WHERE clause.

Syntax

```
SELECT [ALL | DISTINCT] attr_expr_list FROM table_reference
  WHERE where_condition;
```

Keyword

- All is used to return repeated rows. By default, all repeated rows are returned. It is followed by asterisks (*) only. Otherwise, an error will occur.
- DISTINCT is used to remove the repeated line from the result.
- WHERE is used to filter out records that do not meet the condition and return records that meet the condition.

Precautions

The to-be-queried table must exist.

Example

To filter the records in which the scores are higher than 90 and lower than 95 in the **student** table, run the following statement:

```
SELECT * FROM student
  WHERE score > 90 AND score < 95;
```

1.29.2 HAVING Filtering Clause

Function

This statement is used to filter the query results using the HAVING clause.

Syntax

```
SELECT [ALL | DISTINCT] attr_expr_list FROM table_reference
  [WHERE where_condition]
  [GROUP BY col_name_list]
  HAVING having_condition;
```

Keyword

- All is used to return repeated rows. By default, all repeated rows are returned. It is followed by asterisks (*) only. Otherwise, an error will occur.
- DISTINCT is used to remove the repeated line from the result.
- Generally, HAVING and GROUP BY are used together. GROUP BY applies first for grouping and HAVING then applies for filtering. The arithmetic operation and aggregate function are supported by the HAVING clause.

Precautions

- The to-be-queried table must exist.
- If the filtering condition is subject to the query results of GROUP BY, the HAVING clause, rather than the WHERE clause, must be used for filtering.

Example

Group the **student** table according to the **name** field and filter the records in which the maximum score is higher than 95 based on groups.

```
SELECT name, max(score) FROM student
  GROUP BY name
  HAVING max(score) >95;
```

1.30 Sorting

1.30.1 ORDER BY

Function

This statement is used to order the result set of a query by the specified field.

Syntax

```
SELECT attr_expr_list FROM table_reference
  ORDER BY col_name
  [ASC | DESC] [,col_name [ASC | DESC],...];
```

Keyword

- **ASC/DESC**: ASC sorts from the lowest value to the highest value. DESC sorts from the highest value to the lowest value. **ASC** is the default sort order.
- **ORDER BY**: specifies that the values in one or more columns should be sorted globally. When **ORDER BY** is used with **GROUP BY**, **ORDER BY** can be followed by the aggregate function.

Precautions

The to-be-sorted table must exist. If this statement is used to sort a table that does not exist, an error is reported.

Example

To sort table **student** in ascending order according to field **score** and return the sorting result, run the following statement:

```
SELECT * FROM student
  ORDER BY score;
```

1.30.2 SORT BY

Function

This statement is used to achieve the partial sorting of tables according to fields.

Syntax

```
SELECT attr_expr_list FROM table_reference
  SORT BY col_name
  [ASC | DESC] [,col_name [ASC | DESC],...];
```

Keyword

- ASC/DESC: ASC sorts from the lowest value to the highest value. DESC sorts from the highest value to the lowest value. ASC is the default sort order.
- SORT BY: Used together with GROUP BY to perform local sorting of a single column or multiple columns for PARTITION.

Precautions

The to-be-sorted table must exist. If this statement is used to sort a table that does not exist, an error is reported.

Example

To sort the **student** table in ascending order of the **score** field in Reducer, run the following statement:

```
SELECT * FROM student
  SORT BY score;
```

1.30.3 CLUSTER BY

Function

This statement is used to bucket a table and sort the table within buckets.

Syntax

```
SELECT attr_expr_list FROM table_reference
  CLUSTER BY col_name [,col_name ,...];
```

Keyword

CLUSTER BY: Buckets are created based on specified fields. Single fields and multiple fields are supported, and data is sorted in buckets.

Precautions

The to-be-sorted table must exist. If this statement is used to sort a table that does not exist, an error is reported.

Example

To bucket the **student** table according to the **score** field and sort tables within buckets in descending order, run the following statement:

```
SELECT * FROM student
  CLUSTER BY score;
```

1.30.4 DISTRIBUTE BY

Function

This statement is used to bucket a table according to the field.

Syntax

```
SELECT attr_expr_list FROM table_reference
  DISTRIBUTE BY col_name [,col_name ,...];
```

Keyword

DISTRIBUTE BY: Buckets are created based on specified fields. A single field or multiple fields are supported, and the fields are not sorted in the bucket. This parameter is used together with SORT BY to sort data after bucket division.

Precautions

The to-be-sorted table must exist. If this statement is used to sort a table that does not exist, an error is reported.

Example Value

To bucket the **student** table according to the **score** field, run the following statement:

```
SELECT * FROM student
  DISTRIBUTE BY score;
```

1.31 Grouping

1.31.1 Column-Based GROUP BY

Function

This statement is used to group a table based on columns.

Syntax

```
SELECT attr_expr_list FROM table_reference
  GROUP BY col_name_list;
```

Keyword

Column-based GROUP BY can be categorized into single-column GROUP BY and multi-column GROUP BY.

- Single-column GROUP BY indicates that the GROUP BY clause contains only one column. The fields in **col_name_list** must exist in **attr_expr_list**. The aggregate function, **count()** and **sum()** for example, is supported in **attr_expr_list**. The aggregate function can contain other fields.
- Multi-column GROUP BY indicates that there is more than one column in the GROUP BY clause. The query statement is grouped according to all the fields in the GROUP BY clause. The records with the same fields are put in the same group. Similarly, the fields in the GROUP BY clause must be in the fields in **attr_expr_list**. The **attr_expr_list** field can also use the aggregate function.

Precautions

The to-be-grouped table must exist. Otherwise, an error is reported.

Example

Group the **student** table according to the score and name fields and return the grouping results.

```
SELECT score, count(name) FROM student
  GROUP BY score,name;
```

1.31.2 Expression-Based GROUP BY

Function

This statement is used to group a table according to expressions.

Syntax

```
SELECT attr_expr_list FROM table_reference
  GROUP BY groupby_expression [, groupby_expression, ...];
```

Keyword

The **groupby_expression** can contain a single field or multiple fields, and also can call aggregate functions or string functions.

Precautions

- The to-be-grouped table must exist. Otherwise, an error is reported.
- In the same single-column group, built-in functions and self-defined functions are supported in the expression in the GRUOP BY fields that must exit in **attr_expr_list**.

Example

To use the **substr** function to obtain the character string from the **name** field, group the student table according to the obtained character string, and return each sub character string and the number of records, run the following statement:

```
SELECT substr(name,6),count(name) FROM student
  GROUP BY substr(name,6);
```

1.31.3 GROUP BY Using HAVING

Function

This statement filters a table after grouping it using the HAVING clause.

Syntax

```
SELECT attr_expr_list FROM table_reference
  GROUP BY groupby_expression [, groupby_expression...]
  HAVING having_expression;
```

Keyword

The groupby_expression can contain a single field or multiple fields, and can also call aggregate functions or string functions.

Precautions

- The to-be-grouped table must exist. Otherwise, an error is reported.
- If the filtering condition is subject to the query results of GROUP BY, the HAVING clause, rather than the WHERE clause, must be used for filtering. If HAVING and GROUP BY are used together, GROUP BY applies first for grouping and HAVING then applies for filtering. The arithmetic operation and aggregate function are supported by the HAVING clause.

Example

Group the **transactions** according to **num**, use the HAVING clause to filter the records in which the maximum value derived from multiplying **price** with **amount** is higher than 5000, and return the filtered results.

```
SELECT num, max(price*amount) FROM transactions
  WHERE time > '2016-06-01'
  GROUP BY num
  HAVING max(price*amount)>5000;
```

1.31.4 ROLLUP

Function

This statement is used to generate the aggregate row, super-aggregate row, and the total row. The statement can achieve multi-layer statistics from right to left and display the aggregation of a certain layer.

Syntax

```
SELECT attr_expr_list FROM table_reference
  GROUP BY col_name_list
  WITH ROLLUP;
```

Keyword

ROLLUP is the expansion of GROUP BY. For example, ***SELECT a, b, c, SUM(expression) FROM table GROUP BY a, b, c WITH ROLLUP;*** can be transformed into the following query statements:

- Counting the (a, b, c) combinations
```
SELECT a, b, c, sum(expression) FROM table
  GROUP BY a, b, c;
```
- Counting the (a, b) combinations
```
SELECT a, b, NULL, sum(expression) FROM table
  GROUP BY a, b;
```
- Counting the (a) combinations
```
SELECT a, NULL, NULL, sum(expression) FROM table
  GROUP BY a;
```
- Total
```
SELECT NULL, NULL, NULL, sum(expression) FROM table;
```

Precautions

The to-be-grouped table must exist. If this statement is used to group a table that does not exist, an error is reported.

Example

To generate the aggregate row, super-aggregate row, and total row according to the group_id and job fields and return the total salary on each aggregation condition, run the following statement:

```
SELECT group_id, job, SUM(salary) FROM group_test
  GROUP BY group_id, job
  WITH ROLLUP;
```

1.31.5 GROUPING SETS

Function

This statement is used to generate the cross-table row and achieve the cross-statistics of the GROUP BY field.

Syntax

```
SELECT attr_expr_list FROM table_reference
  GROUP BY col_name_list
  GROUPING SETS(col_name_list);
```

Keyword

GROUPING SETS is the expansion of GROUP BY. For example:

- ***SELECT a, b, sum(expression) FROM table GROUP BY a, b GROUPING SETS((a,b));***

  It can be converted to the following query:

  ```
  SELECT a, b, sum(expression) FROM table
    GROUP BY a, b;
  ```

- ***SELECT a, b, sum(expression) FROM table GROUP BY a, b GROUPING SETS(a,b);***

  It can be converted to the following two queries:

  ```
  SELECT a, NULL, sum(expression) FROM table GROUP BY a;
  UNION
  SELECT NULL, b, sum(expression) FROM table GROUP BY b;
  ```

- ***SELECT a, b, sum(expression) FROM table GROUP BY a, b GROUPING SETS((a,b), a);***

  It can be converted to the following two queries:

  ```
  SELECT a, b, sum(expression) FROM table GROUP BY a, b;
  UNION
  SELECT a, NULL, sum(expression) FROM table GROUP BY a;
  ```

- ***SELECT a, b, sum(expression) FROM table GROUP BY a, b GROUPING SETS((a,b), a, b, ());***

  It can be converted to the following four queries:

  ```
  SELECT a, b, sum(expression) FROM table GROUP BY a, b;
  UNION
  SELECT a, NULL, sum(expression) FROM table GROUP BY a, NULL;
  UNION
  SELECT NULL, b, sum(expression) FROM table GROUP BY NULL, b;
  UNION
  SELECT NULL, NULL, sum(expression) FROM table;
  ```


Precautions

- The to-be-grouped table must exist. Otherwise, an error is reported.
- Different from ROLLUP, there is only one syntax for GROUPING SETS.

Example

To generate the cross-table row according to the **group_id** and **job** fields and return the total salary on each aggregation condition, run the following statement:

```
SELECT group_id, job, SUM(salary) FROM group_test
  GROUP BY group_id, job
  GROUPING SETS (group_id, job);
```

1.32 JOIN

1.32.1 INNER JOIN

Function

This statement is used to join and return the rows that meet the JOIN conditions from two tables as the result set.

Syntax

```
SELECT attr_expr_list FROM table_reference
  {JOIN | INNER JOIN} table_reference ON join_condition;
```

Keyword

JOIN/INNER JOIN: Only the records that meet the JOIN conditions in joined tables will be displayed.

Precautions

- The to-be-joined table must exist. Otherwise, an error is reported.
- INNER JOIN can join more than two tables at one query.

Example

To join the course IDs from the **student_info** and **course_info** tables and check the mapping between student names and courses, run the following statement:

```
SELECT student_info.name, course_info.courseName FROM student_info
  JOIN course_info ON (student_info.courseId = course_info.courseId);
```

1.32.2 LEFT OUTER JOIN

Function

Join the left table with the right table and return all joined records of the left table. If no joined record is found, NULL will be returned.

Syntax

```
SELECT attr_expr_list FROM table_reference
  LEFT OUTER JOIN table_reference ON join_condition;
```

Keyword

LEFT OUTER JOIN: Returns all joined records of the left table. If no record is matched, NULL is returned.

Precautions

The to-be-joined table must exist. Otherwise, an error is reported.

Example

To join the courseId from the **student_info** table to the **courseId** from the **course_info** table for inner join and return the name of the students who have selected course, run the following statement. If no joined record is found, NULL will be returned.

```
SELECT student_info.name, course_info.courseName FROM student_info
  LEFT OUTER JOIN course_info ON (student_info.courseId = course_info.courseId);
```

1.32.3 RIGHT OUTER JOIN

Function

Match the right table with the left table and return all matched records of the right table. If no matched record is found, NULL will be returned.

Syntax

```
SELECT attr_expr_list FROM table_reference
  RIGHT OUTER JOIN table_reference ON join_condition;
```

Keyword

RIGHT OUTER JOIN: Return all matched records of the right table. If no record is matched, NULL is returned.

Precautions

The to-be-joined table must exist. Otherwise, an error is reported.

Example

To join the courseId from the **course_info** table to the **courseId** from the **student_info** table for inner join and return the records in the **course_info** table, run the following statement. If no joined record is found, NULL will be returned.

```
SELECT student_info.name, course_info.courseName FROM student_info
  RIGHT OUTER JOIN course_info ON (student_info.courseId = course_info.courseId);
```

1.32.4 FULL OUTER JOIN

Function

Join all records from the right table and the left table and return all joined records. If no joined record is found, NULL will be returned.

Syntax

```
SELECT attr_expr_list FROM table_reference
  FULL OUTER JOIN table_reference ON join_condition;
```

Keyword

FULL OUTER JOIN: Matches all records in the left and right tables. If no record is matched, NULL is returned.

Precautions

The to-be-joined table must exist. Otherwise, an error is reported.

Example

To join all records from the right table and the left table and return all joined records, run the following statement. If no joined record is found, NULL will be returned.

```
SELECT student_info.name, course_info.courseName FROM student_info
  FULL OUTER JOIN course_info ON (student_info.courseId = course_info.courseId);
```

1.32.5 IMPLICIT JOIN

Function

This statement has the same function as INNER JOIN, that is, the result set that meet the WHERE condition is returned. However, IMPLICIT JOIN does not use the condition specified by JOIN.

Syntax

```
SELECT table_reference.col_name, table_reference.col_name, ... FROM table_reference, table_reference
  WHERE table_reference.col_name = table_reference.col_name;
```

Keyword

The keyword WHERE achieves the same function as JOIN...ON... and the mapped records will be returned. **Syntax** shows the WHERE filtering according to an equation. The WHERE filtering according to an inequation is also supported.

Precautions

- The to-be-joined table must exist. Otherwise, an error is reported.
- The statement of IMPLICIT JOIN does not contain keywords JOIN...ON.... Instead, the WHERE clause is used as the condition to join two tables.

Example

To return the student names and course names that match **courseId**, run the following statement:

```
SELECT student_info.name, course_info.courseName FROM student_info,course_info
  WHERE student_info.courseId = course_info.courseId;
```

1.32.6 Cartesian JOIN

Function

Cartesian JOIN joins each record of table A with all records in table B. For example, if there are *m* records in table A and *n* records in table B, *m* x *n* records will be generated by Cartesian JOIN.

Syntax

```
SELECT attr_expr_list FROM table_reference
  CROSS JOIN table_reference ON join_condition;
```

Keyword

The join_condition is the condition for joining. If join_condition is always true, for example **1=1**, the join is Cartesian JOIN. Therefore, the number of records output by Cartesian join is equal to the product of the number of records in the joined table. If Cartesian join is required, use the special keyword CROSS JOIN. CROSS JOIN is the standard way to calculate Cartesian product.

Precautions

The to-be-joined table must exist. Otherwise, an error is reported.

Example

To return all the JOIN results of the student name and course name from the **student_info** and **course_info** tables, run the following statement:

```
SELECT student_info.name, course_info.courseName FROM student_info
  CROSS JOIN course_info ON (1 = 1);
```

1.32.7 LEFT SEMI JOIN

Function

This statement is used to query the records that meet the JOIN condition from the left table.

Syntax

```
SELECT attr_expr_list FROM table_reference
  LEFT SEMI JOIN table_reference ON join_condition;
```

Keyword

LEFT SEMI JOIN: Indicates to only return the records from the left table. LEFT SEMI JOIN can be achieved by nesting subqueries in LEFT SEMI JOIN, WHERE...IN, or

WHERE EXISTS. LEFT SEMI JOIN returns the records that meet the JOIN condition from the left table, while LEFT OUTER JOIN returns all the records from the left table or NULL if no records that meet the JOIN condition are found.

Precautions

- The to-be-joined table must exist. Otherwise, an error is reported.
- he fields in attr_expr_list must be the fields in the left table. Otherwise, an error is reported.

Example

To return the names of students who select the courses and the course IDs, run the following statement:

```
SELECT student_info.name, student_info.courseId FROM student_info
  LEFT SEMI JOIN course_info ON (student_info.courseId = course_info.courseId);
```

1.32.8 NON-EQUIJOIN

Function

This statement is used to join multiple tables using unequal values and return the result set that meet the condition.

Syntax

```
SELECT attr_expr_list FROM table_reference
  JOIN table reference ON non_equi_join_condition;
```

Keyword

The **non_equi_join_condition** is similar to **join_condition**. The only difference is that the JOIN condition is inequation.

Precautions

The to-be-joined table must exist. Otherwise, an error is reported.

Example

To return all the pairs of different student names from the **student_info_1** and **student_info_2** tables, run the following statement:

```
SELECT student_info_1.name, student_info_2.name FROM student_info_1
  JOIN student_info_2 ON (student_info_1. name <> student_info_2. name);
```

1.33 Subquery

1.33.1 Subquery Nested by WHERE

Function

Subqueries are nested in the WHERE clause, and the subquery result is used as the filtering condition.

Syntax

```
SELECT [ALL | DISTINCT] attr_expr_list FROM table_reference
  WHERE {col_name operator (sub_query) | [NOT] EXISTS sub_query};
```

Keyword

- All is used to return repeated rows. By default, all repeated rows are returned. It is followed by asterisks (*) only. Otherwise, an error will occur.
- DISTINCT is used to remove the repeated line from the result.
- The subquery results are used as the filter condition in the subquery nested by WHERE.
- The operator includes the equation and inequation operators, and IN, NOT IN, EXISTS, and NOT EXISTS operators.
  - If the operator is IN or NOT IN, the returned records are in a single column.
  - If the operator is EXISTS or NOT EXISTS, the subquery must contain WHERE. If any a field in the subquery is the same as that in the external query, add the table name before the field in the subquery.

Precautions

The to-be-queried table must exist. If this statement is used to query a table that does not exist, an error is reported.

Example

To query the courseId of Biology from the course_info table, and then query the student name matched the courseId from the student_info table, run the following statement:

```
SELECT name FROM student_info
  WHERE courseId = (SELECT courseId FROM course_info WHERE courseName = 'Biology');
```

1.33.2 Subquery Nested by FROM

Function

This statement is used to nest subquery by FROM and use the subquery results as the data source of the external SELECT statement.

Syntax

```
SELECT [ALL | DISTINCT] attr_expr_list FROM (sub_query) [alias];
```

Keyword

- All is used to return repeated rows. By default, all repeated rows are returned. It is followed by asterisks (*) only. Otherwise, an error will occur.
- DISTINCT is used to remove the repeated line from the result.

Precautions

- The to-be-queried table must exist. If this statement is used to query a table that does not exist, an error is reported.
- The subquery nested in FROM must have an alias. The alias must be specified before the running of the statement. Otherwise, an error is reported. It is advised to specify a unique alias.
- The subquery results sequent to FROM must be followed by the specified alias. Otherwise, an error is reported.

Example

To return the names of students who select the courses in the **course_info** table and remove the repeated records using DISTINCT, run the following statement:

```
SELECT DISTINCT name FROM (SELECT name FROM student_info
  JOIN course_info ON student_info.courseId = course_info.courseId) temp;
```

1.33.3 Subquery Nested by HAVING

Function

This statement is used to embed a subquery in the HAVING clause. The subquery result is used as a part of the HAVING clause.

Syntax

```
SELECT [ALL | DISTINCT] attr_expr_list FROM table_reference
  GROUP BY groupby_expression
  HAVING aggregate_func(col_name) operator (sub_query);
```

Keyword

- All is used to return repeated rows. By default, all repeated rows are returned. It is followed by asterisks (*) only. Otherwise, an error will occur.
- DISTINCT is used to remove the repeated line from the result.
- The **groupby_expression** can contain a single field or multiple fields, and also can call aggregate functions or string functions.
- The operator includes the equation and inequation operators, and IN and NOT IN operators.

Precautions

- The to-be-queried table must exist. If this statement is used to query a table that does not exist, an error is reported.
- The sequence of **sub_query** and the aggregate function cannot be changed.

Example

To group the **student_info** table according to the name field, count the records of each group, and return the number of records in which the name fields in the **student_info** table equal to the name fields in the **course_info** table if the two tables have the same number of records, run the following statement:

```
SELECT name FROM student_info
  GROUP BY name
  HAVING count(name) = (SELECT count(*) FROM course_info);
```

1.33.4 Multi-Layer Nested Subquery

Function

This statement is used to nest queries in the subquery.

Syntax

```
SELECT attr_expr FROM ( SELECT attr_expr FROM ( SELECT attr_expr FROM... ... ) [alias] ) [alias];
```

Keyword

- All is used to return repeated rows. By default, all repeated rows are returned. It is followed by asterisks (*) only. Otherwise, an error will occur.
- DISTINCT is used to remove the repeated line from the result.

Precautions

- The to-be-queried table must exist. If this statement is used to query a table that does not exist, an error is reported.
- The alias of the subquery must be specified in the nested query. Otherwise, an error is reported.
- The alias must be specified before the running of the statement. Otherwise, an error is reported. It is advised to specify a unique alias.

Example

To return the name field from the **user_info** table after three queries, run the following statement:

```
SELECT name FROM ( SELECT name, acc_num FROM ( SELECT name, acc_num, password FROM ( SELECT
name, acc_num, password, bank_acc FROM user_info) a ) b ) c;
```

1.34 Alias

1.34.1 AS for Table

Function

This statement is used to specify an alias for a table or the subquery result.

Syntax

```
SELECT attr_expr_list FROM table_reference [AS] alias;
```

Keyword

- table_reference: Can be a table, view, or subquery.
- As: Is used to connect to table_reference and alias. Whether this keyword is added or not does not affect the command execution result.

Precautions

- The to-be-queried table must exist. Otherwise, an error is reported.
- The alias must be specified before execution of the statement. Otherwise, an error is reported. You are advised to specify a unique alias.

Example

- To specify alias **n** for table **simple_table** and visit the name field in table **simple_table** by using n.name, run the following statement:
```
SELECT n.score FROM simple_table n WHERE n.name = "leilei";
```
- To specify alias **m** for the subquery result and return all the query results using **SELECT * FROM m**, run the following statement:
```
SELECT * FROM (SELECT * FROM simple_table WHERE score > 90) AS m;
```

1.34.2 AS for Column

Function

This statement is used to specify an alias for a column.

Syntax

```
SELECT attr_expr [AS] alias, attr_expr [AS] alias, ... FROM table_reference;
```

Keyword

- alias: gives an alias for the **attr_expr** field.
- AS: Whether to add AS does not affect the result.

Precautions

- The to-be-queried table must exist. Otherwise, an error is reported.
- The alias must be specified before execution of the statement. Otherwise, an error is reported. You are advised to specify a unique alias.

Example

Run **SELECT name AS n FROM simple_table WHERE score > 90** to obtain the subquery result. The alias **n** for **name** can be used by external SELECT statement.

```
SELECT n FROM (SELECT name AS n FROM simple_table WHERE score > 90) m WHERE n = "xiaoming";
```


1.35 Set Operations

1.35.1 UNION

Function

This statement is used to return the union set of multiple query results.

Syntax

```
select_statement UNION [ALL] select_statement;
```

Keyword

UNION: The set operation is used to join the head and tail of a table based on certain conditions. The number of columns returned by each SELECT statement must be the same. The column type and column name may not be the same.

Precautions

- By default, the repeated records returned by UNION are removed. The repeated records returned by UNION ALL are not removed.
- Do not add brackets between multiple set operations, such as UNION, INTERSECT, and EXCEPT. Otherwise, an error is reported.

Example

To return the union set of the query results of the **SELECT * FROM student _1** and **SELECT * FROM student _2** commands with the repeated records removed, run the following statement:

```
SELECT * FROM student_1 UNION SELECT * FROM student_2;
```

1.35.2 INTERSECT

Function

This statement is used to return the intersection set of multiple query results.

Syntax

```
select_statement INTERSECT select_statement;
```

Keyword

INTERSECT returns the intersection of multiple query results. The number of columns returned by each SELECT statement must be the same. The column type and column name may not be the same. By default, INTERSECT deduplication is used.

Precautions

Do not add brackets between multiple set operations, such as UNION, INTERSECT, and EXCEPT. Otherwise, an error is reported.

Example

To return the intersection set of the query results of the **SELECT * FROM student _1** and **SELECT * FROM student _2** commands with the repeated records removed, run the following statement:

```
SELECT * FROM student _1 INTERSECT SELECT * FROM student _2;
```

1.35.3 EXCEPT

Function

This statement is used to return the difference set of two query results.

Syntax

```
select_statement EXCEPT select_statement;
```

Keyword

EXCEPT minus the sets. A EXCEPT B indicates to remove the records that exist in both A and B from A and return the results. The repeated records returned by EXCEPT are not removed by default. The number of columns returned by each SELECT statement must be the same. The types and names of columns do not have to be the same.

Precautions

Do not add brackets between multiple set operations, such as UNION, INTERSECT, and EXCEPT. Otherwise, an error is reported.

Example

To remove the records that exist in both **SELECT * FROM student_1** and **SELECT * FROM student_2** from **SELECT * FROM student_1** and return the results, run the following statement:

```
SELECT * FROM student_1 EXCEPT SELECT * FROM student_2;
```

1.36 WITH...AS

Function

This statement is used to define the common table expression (CTE) using WITH...AS to simplify the query and make the result easier to read and maintain.

Syntax

```
WITH cte_name AS (select_statement) sql_containing_cte_name;
```

Keyword

- cte_name: Name of a public expression. The name must be unique.
- select_statement: complete SELECT clause.
- sql_containing_cte_name: SQL statement containing the defined common expression.

Precautions

- A CTE must be used immediately after it is defined. Otherwise, the definition becomes invalid.
- Multiple CTEs can be defined by WITH at a time. The CTEs are separated by commas and the CTEs defined later can quote the CTEs defined earlier.

Example

Define **SELECT courseId FROM course_info WHERE courseName = 'Biology'** as CTE **nv** and use **nv** as the SELECT statement in future queries.

```
WITH nv AS (SELECT courseId FROM course_info WHERE courseName = 'Biology') SELECT DISTINCT
courseId FROM nv;
```

1.37 CASE...WHEN

1.37.1 Basic CASE Statement

Function

This statement is used to display **result_expression** according to the joined results of **input_expression** and **when_expression**.

Syntax

```
CASE input_expression WHEN when_expression THEN result_expression [...n] [ELSE else_result_expression]
END;
```

Keyword

CASE: Subquery is supported in basic CASE statement. However, input_expression and when_expression must be joinable.

Precautions

If there is no input_expression = when_expression with the TRUE value, else_result_expression will be returned when the ELSE clause is specified. If the ELSE clause is not specified, NULL will be returned.

Example

To return the name field and the character that is matched to id from the student table with the following matching rules, run the following statement:

- If id is 1, 'a' is returned.
- If id is 2, 'b' is returned.
- If id is 3, 'c' is returned.
- Otherwise, **NULL** is returned.

```
SELECT name, CASE id WHEN 1 THEN 'a' WHEN 2 THEN 'b' WHEN 3 THEN 'c' ELSE NULL END FROM student;
```

1.37.2 CASE Query Statement

Function

This statement is used to obtain the value of **boolean_expression** for each WHEN statement in a specified order. Then return the first **result_expression** with the value **TRUE** of **boolean_expression**.

Syntax

```
CASE WHEN boolean_expression THEN result_expression [...n] [ELSE else_result_expression] END;
```

Keyword

boolean_expression: can include subquery. However, the return value of boolean_expression can only be of Boolean type.

Precautions

If there is no Boolean_expression with the TRUE value, else_result_expression will be returned when the ELSE clause is specified. If the ELSE clause is not specified, NULL will be returned.

Example

To query the student table and return the related results for the name and score fields: EXCELLENT if the score is higher than 90, GOOD if the score ranges from 80 to 90, and BAD if the score is lower than 80, run the following statement:

```
SELECT name, CASE WHEN score >= 90 THEN 'EXCELLENT' WHEN 80 < score AND score < 90 THEN 'GOOD' ELSE 'BAD' END AS level FROM student;
```

1.38 OVER Clause

Function

This statement is used together with the window function. The OVER statement is used to group data and sort the data within the group. The window function is used to generate serial numbers for values within the group.

Syntax

```
SELECT window_func(args) OVER
  ([PARTITION BY col_name, col_name, ...]
   [ORDER BY col_name, col_name, ...]
   [ROWS | RANGE BETWEEN (CURRENT ROW | (UNBOUNDED |[num]) PRECEDING)
  AND (CURRENT ROW | ( UNBOUNDED | [num]) FOLLOWING)]);
```

Keyword

- PARTITION BY: used to partition a table with one or multiple fields. Similar to GROUP BY, PARTITION BY is used to partition table by fields and each partition is a window. The window function can apply to the entire table or specific partitions. A maximum of 7,000 partitions can be created in a single table.
- ORDER BY: used to specify the order for the window function to obtain the value. ORDER BY can be used to sort table with one or multiple fields. The sorting order can be ascending (specified by **ASC**) or descending (specified by **DESC**). The window is specified by WINDOW. If the window is not specified, the default window is ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. In other words, the window starts from the head of the table or partition (if PARTITION BY is used in the OVER clause) to the current row.
- WINDOW: used to define the window by specifying a range of rows.
- CURRENT ROW: indicates the current row.
- num PRECEDING: used to specify the start of the defined window. The window starts from the num row precedes the current row.
- UNBOUNDED PRECEDING: used to indicate that there is no start of the window.
- num FOLLOWING: used to specify the end of the defined window. The window ends from the num row following the current row.
- UNBOUNDED FOLLOWING: used to indicate that there is no end of the window.
- The differences between ROWS BETWEEN... and RANGE BETWEEN... are as follows:
  - ROWS refers to the physical window. After the data is sorted, the physical window starts at the *n*th row in front of the current row and ends at the *m*th row following the current row.
  - RANGE refers to the logic window. The column of the logic window is determined by the values rather than the location of rows.
- The scenarios of the window are as follows:
  - The window only contains the current row.
    ```
    ROWS BETWEEN CURRENT ROW AND CURRENT ROW
    ```
  - The window starts from three rows precede the current row and ends at the fifth row follows the current row.
    ```
    ROWS BETWEEN 3 PRECEDING AND 5 FOLLOWING
    ```
  - The window starts from the beginning of the table or partition and ends at the current row.
    ```
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ```
  - The window starts from the current window and ends at the end of the table or partition.
    ```
    ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
    ```
  - The window starts from the beginning of the table or partition and ends at the end of the table or partition.
    ```
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ```

Precautions

The three options of the OVER clause are PARTITION BY, ORDER BY, and WINDOW. They are optional and can be used together. If the OVER clause is empty, the window is the entire table.

Example

To start the window from the beginning of the table or partition and end the window at the current row, sort the over_test table according to the id field, and return the sorted id fields and corresponding serial numbers, run the following statement:

```
SELECT id, count(id) OVER (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT
ROW) FROM over_test;
```


2 Flink SQL Syntax

2.1 SQL Syntax Constraints and Definitions

Syntax Constraints

- Currently, Flink SQL only supports the following operations: SELECT, FROM, WHERE, UNION, aggregation, window, JOIN between stream and table data, and JOIN between streams.
- Data cannot be inserted into the source stream.
- The sink stream cannot be used to perform query operations.

Data Types Supported by Syntax

- Basic data types: VARCHAR, STRING, BOOLEAN, TINYINT, SMALLINT, INTEGER/INT, BIGINT, REAL/FLOAT, DOUBLE, DECIMAL, DATE, TIME, and TIMESTAMP
- Array: Square brackets ([]) are used to quote fields. The following is an example:
```
insert into temp select CARDINALITY(ARRAY[1,2,3]) FROM OrderA;
```

Syntax Definition

```
INSERT INTO stream_name query;
query:
  values
  | {
      select
      | selectWithoutFrom
      | query UNION [ ALL ] query
    }

orderItem:
  expression [ ASC | DESC ]

select:
  SELECT
  { * | projectItem [, projectItem ]* }
  FROM tableExpression [ JOIN tableExpression ]
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
```

```
selectWithoutFrom:
  SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }

projectItem:
  expression [ [ AS ] columnAlias ]
  | tableAlias . *

tableExpression:
  tableReference

tableReference:
  tablePrimary
  [ [ AS ] alias [ '(' columnAlias [, columnAlias ]* ')' ] ]

tablePrimary:
  [ TABLE ] [ [ catalogName . ] schemaName . ] tableName
  | LATERAL TABLE '(' functionName '(' expression [, expression ]* ')' ')'
  | UNNEST '(' expression ')'

values:
  VALUES expression [, expression ]*

groupItem:
  expression
  | '(' ')'
  | '(' expression [, expression ]* ')'
  | CUBE '(' expression [, expression ]* ')'
  | ROLLUP '(' expression [, expression ]* ')'
  | GROUPING SETS '(' groupItem [, groupItem ]* ')'
```

2.2 SQL Syntax Overview of Stream Jobs

This section describes the Flink SQL syntax list provided by DLI. For details about the parameters and examples, see the syntax description.

Table 2-1 SQL Syntax of stream jobs

Classification	Function
Creating a Source Stream	**DIS Source Stream**
	DMS Source Stream
Creating a Source Stream	**MRS Kafka Source Stream**
	Open-Source Kafka Source Stream
	OBS Source Stream
Creating a Sink Stream	**CSS Elasticsearch Sink Stream**
	DCS Sink Stream
	DDS Sink Stream
	DIS Sink Stream
	DMS Sink Stream
	DWS Sink Stream (JDBC Mode)

Classification	Function
	DWS Sink Stream (OBS-based Dumping)
Creating a Sink Stream	MRS HBase Sink Stream
	MRS Kafka Sink Stream
	Open-Source Kafka Sink Stream
	OBS Sink Stream
	RDS Sink Stream
Creating a Sink Stream	SMN Sink Stream
	File System Sink Stream (Recommended)
Creating a Temporary Stream	Creating a Temporary Stream
Creating a Dimension Table	Creating a Redis Table
	Creating an RDS Table
Custom Stream Ecosystem	Custom Source Stream
	Custom Sink Stream

2.3 Creating a Source Stream

2.3.1 DIS Source Stream

Function

Create a source stream to read data from DIS. DIS accesses user data and Flink job reads data from the DIS stream as input data for jobs. Flink jobs can quickly remove data from producers using DIS source sources for continuous processing. Flink jobs are applicable to scenarios where data outside the cloud service is imported to the cloud service for filtering, real-time analysis, monitoring reports, and dumping.

DIS addresses the challenge of transmitting data outside cloud services to cloud services. DIS builds data intake streams for custom applications capable of processing or analyzing streaming data. DIS continuously captures, transmits, and stores terabytes of data from hundreds of thousands of sources every hour, such as logs, Internet of Things (IoT) data, social media feeds, website clickstreams, and location-tracking events. For more information about DIS, see the .

Syntax

```
CREATE SOURCE STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
 WITH (
  type = "dis",
```

```
region = "",
channel = "",
partition_count = "",
encode = "",
field_delimiter = "",
offset= "");
```

Keyword

Table 2-2 Keyword description

Parameter	Mandatory	Description
type	Yes	Data source type. **dis** indicates that the data source is DIS.
region	Yes	Region where DIS for storing the data is located.
ak	No	Access Key ID (AK).
sk	No	Specifies the secret access key used together with the ID of the access key.
channel	Yes	Name of the DIS stream where data is located.
partition_count	No	Number of partitions of the DIS stream where data is located. This parameter and **partition_range** cannot be configured at the same time. If this parameter is not specified, data of all partitions is read by default.
partition_range	No	Range of partitions of a DIS stream, data in which is ingested by the DLI job. This parameter and **partition_count** cannot be configured at the same time. If this parameter is not specified, data of all partitions is read by default. If you set this parameter to **[0:2]**, data will be read from partitions 1, 2, and 3.
encode	Yes	Data encoding format. The value can be **csv**, **json**, **xml**, **email**, **blob**, or **user_defined**. ● **field_delimiter** must be specified if this parameter is set to **csv**. ● **json_config** must be specified if this parameter is set to **json**. ● **xml_config** must be specified if this parameter is set to **xml**. ● **email_key** must be specified if this parameter is set to **email**. ● If this parameter is set to **blob**, the received data is not parsed, only one stream attribute exists, and the data format is ARRAY[TINYINT]. ● **encode_class_name** and **encode_class_parameter** must be specified if this parameter is set to **user_defined**.

<table>
<tr><th>Parameter</th><th>Mandatory</th><th>Description</th></tr>
<tr><td>field_delimiter</td><td>No</td><td>Attribute delimiter. This parameter is mandatory only when the CSV encoding format is used. You can set this parameter, for example, to a comma (,).</td></tr>
<tr><td>quote</td><td>No</td><td>Quoted symbol in a data format. The attribute delimiters between two quoted symbols are treated as common characters.
If double quotation marks are used as the quoted symbol, set this parameter to \u005c\u0022 for character conversion.
If a single quotation mark is used as the quoted symbol, set this parameter to a single quotation mark (').
NOTE
Currently, only the CSV format is supported.
After this parameter is specified, ensure that each field does not contain quoted symbols or contains an even number of quoted symbols. Otherwise, parsing will fail.</td></tr>
<tr><td>json_config</td><td>No</td><td>When the encoding format is JSON, you need to use this parameter to specify the mapping between JSON fields and stream definition fields. The format is field1=data_json.field1; field2=data_json.field2; field3=$, where <b>field3=$ indicates that the content of field3 is the entire JSON string.</td></tr>
<tr><td>xml_config</td><td>No</td><td>If encode is set to xml, you need to set this parameter to specify the mapping between the xml field and the stream definition field. An example of the format is as follows: field1=data_xml.field1; field2=data_xml.field2.</td></tr>
<tr><td>email_key</td><td>No</td><td>If encode is set to email, you need to set the parameter to specify the information to be extracted. You need to list the key values that correspond to stream definition fields. Multiple key values are separated by commas (,), for example, "Message-ID, Date, Subject, body". There is no keyword in the email body and DLI specifies "body" as the keyword.</td></tr>
<tr><td>encode_class_name</td><td>No</td><td>If encode is set to user_defined, you need to set this parameter to the name of the user-defined decoding class (including the complete package path). The class must inherit the DeserializationSchema class.</td></tr>
<tr><td>encode_class_parameter</td><td>No</td><td>If encode is set to user_defined, you can set this parameter to specify the input parameter of the user-defined decoding class. Only one parameter of the string type is supported.</td></tr>
</table>

Parameter	Mandatory	Description
offset	No	● If data is imported to the DIS stream after the job is started, this parameter will become invalid. ● If the job is started after data is imported to the DIS stream, you can set the parameter as required. For example, if **offset** is set to **100**, DLI starts from the 100th data record in DIS.
start_time	No	Start time for reading DIS data. ● If this parameter is specified, DLI reads data read from the specified time. The format is **yyyy-MM-dd HH:mm:ss**. ● If neither **start_time** nor **offset** is specified, DLI reads the latest data. ● If **start_time** is not specified but **offset** is specified, DLI reads data from the data record specified by **offset**.
enable_checkpoint	No	Whether to enable the checkpoint function. The value can be **true** (enabled) or **false** (disabled). The default value is **false**.
checkpoint_app_name	No	ID of a DIS consumer. If a DIS stream is consumed by different jobs, you need to configure the consumer ID for each job to avoid checkpoint confusion.
checkpoint_interval	No	Interval of checkpoint operations on the DIS source operator. The value is in the unit of seconds. The default value is **60**.

Precautions

When creating a source stream, you can specify a time model for subsequent calculation. Currently, DLI supports two time models: Processing Time and Event Time. For details about the syntax, see **Configuring Time Models**.

Example

- In CSV encoding format, DLI reads data from the DIS stream and records it as codes in CSV format. The codes are separated by commas (,).

```
CREATE SOURCE STREAM car_infos (
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed INT,
  total_miles INT,
  car_timestamp LONG
)
  WITH (
    type = "dis",
    region = "xxx",
    channel = "dliinput",
    encode = "csv",
```

```
    field_delimiter = ","
);
```

- In JSON encoding format, DLI reads data from the DIS stream and records it as codes in JSON format. For example, {"car":{"car_id":"ZJA710XC", "car_owner":"coco", "car_age":5, "average_speed":80, "total_miles":15000, "car_timestamp":1526438880}}

```
CREATE SOURCE STREAM car_infos (
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed INT,
  total_miles INT,
  car_timestamp LONG
)
  WITH (
    type = "dis",
    region = "xxx",
    channel = "dliinput",
    encode = "json",
    json_config = "car_id=car.car_id;car_owner =car.car_owner;car_age=car.car_age;average_speed
=car.average_speed ;total_miles=car.total_miles;"
);
```

- In XML encoding format, DLI reads data from the DIS stream and records it as codes in XML format.

```
CREATE SOURCE STREAM person_infos (
    pid BIGINT,
    pname STRING,
    page int,
    plocation STRING,
    pbir DATE,
    phealthy BOOLEAN,
    pgrade ARRAY[STRING]
)
  WITH (
    type = "dis",
    region = "xxx",
    channel = "dis-dli-input",
    encode = "xml",
    field_delimiter = ",",
    xml_config =
"pid=person.pid;page=person.page;pname=person.pname;plocation=person.plocation;pbir=person.pbir;
pgrade=person.pgrade;phealthy=person.phealthy"
);
```

An example of XML data is as follows:

```
<?xml version="1.0" encodeing="utf-8"?>

<root>
  <person>
    <pid>362305199010025042</pid>
    <pname>xiaoming</pname>
    <page>28</page>
    <plocation>Shangdu County, Jining District, Ulanchap, Inner Mongolia</plocation>
    <pbir>1990-10-02</pbir>
    <phealthy>true</phealthy>
    <pgrade>[A,B,C]</pgrade>
  </person>
</root>
```

- In EMAIL encoding format, DLI reads data from the DIS stream and records it as a complete Email.

```
CREATE SOURCE STREAM email_infos (
  Event_ID String,
  Event_Time Date,
  Subject String,
  From_Email String,
  To_EMAIL String,
```

```
  CC_EMAIL Array[String],
  BCC_EMAIL String,
  MessageBody String,
  Mime_Version String,
  Content_Type String,
  charset String,
  Content_Transfer_Encoding String
)
  WITH (
    type = "dis",
    region = "xxx",
    channel = "dliinput",
    encode = "email",
    email_key = "Message-ID, Date, Subject, From, To, CC, BCC, Body, Mime-Version, Content-Type, 
charset, Content_Transfer_Encoding"
);
```

An example of email data is as follows:

```
Message-ID: <200906291839032504254@sample.com>
Date: Fri, 11 May 2001 09:54:00 -0700 (PDT)
From: zhangsan@sample.com
To: lisi@sample.com, wangwu@sample.com
Subject:  "Hello World"
Cc: lilei@sample.com, hanmei@sample.com
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
Bcc: jack@sample.com, lily@sample.com
X-From: Zhang San
X-To: Li Si, Wang Wu
X-cc: Li Lei, Han Mei
X-bcc: 
X-Folder: \Li_Si_June2001\Notes Folders\Notes inbox
X-Origin: Lucy
X-FileName: sample.nsf

Dear Associate / Analyst Committee:

Hello World!

Thank you,

Associate / Analyst Program
zhangsan
```

2.3.2 DMS Source Stream

DMS (Distributed Message Service) is a message middleware service based on distributed, high-availability clustering technology. It provides reliable, scalable, fully managed queues for sending, receiving, and storing messages. DMS for Kafka is a message queuing service based on Apache Kafka. This service provides Kafka premium instances.

The source stream can read data from a Kafka instance as the input data of jobs. The syntax for creating a Kafka source stream is the same as that for creating an open source Apache Kafka source stream. For details, see **Open-Source Kafka Source Stream**.

2.3.3 MRS Kafka Source Stream

Function

Create a source stream to obtain data from Kafka as input data for jobs.

Apache Kafka is a fast, scalable, and fault-tolerant distributed message publishing and subscription system. It delivers high throughput and built-in partitions and provides data replicas and fault tolerance. Apache Kafka is applicable to scenarios of handling massive messages. Kafka clusters are deployed and hosted on MRS that is powered on Apache Kafka.

Prerequisites

- Kafka is an offline cluster. You need to use the enhanced datasource connection function to connect Flink jobs to Kafka. You can also set security group rules as required.

Syntax

```
CREATE SOURCE STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "kafka",
    kafka_bootstrap_servers = "",
    kafka_group_id = "",
    kafka_topic = "",
    encode = "json"
  );
```

Keyword

Table 2-3 Keyword description

Parameter	Mandatory	Description
type	Yes	Data source type. **Kafka** indicates that the data source is Kafka.
kafka_bootstrap_servers	Yes	Port that connects DLI to Kafka. Use enhanced datasource connections to connect DLI queues with Kafka clusters.
kafka_group_id	No	Group ID
kafka_topic	Yes	Kafka topic to be read. Currently, only one topic can be read at a time.
encode	Yes	Data encoding format. The value can be **csv**, **json**, **blob**, or **user_defined**. • **field_delimiter** must be specified if this parameter is set to **csv**. • **json_config** must be specified if this parameter is set to **json**. • If this parameter is set to **blob**, the received data is not parsed, only one stream attribute exists, and the stream attribute is of the Array[TINYINT] type. • **encode_class_name** and **encode_class_parameter** must be specified if this parameter is set to **user_defined**.

Parameter	Mandatory	Description
encode_class_name	No	If **encode** is set to **user_defined**, you need to set this parameter to the name of the user-defined decoding class (including the complete package path). The class must inherit the **Deserialization-Schema** class.
encode_class_parameter	No	If **encode** is set to **user_defined**, you can set this parameter to specify the input parameter of the user-defined decoding class. Only one parameter of the string type is supported.
krb_auth	No	The authentication name for creating a datasource connection authentication. This parameter is mandatory when Kerberos authentication is enabled. **NOTE** Ensure that the **/etc/hosts** information of the master node in the MRS cluster is added to the host file of the DLI queue.
json_config	No	If **encode** is set to **json**, you can use this parameter to specify the mapping between JSON fields and stream attributes. The format is field1=json_field1;field2=json_field2. **field1** and **field2** indicate the names of the created table fields. **json_field1** and **json_field2** are key fields of the JSON strings in the Kafka input data. For details, see the **example**.
field_delimiter	No	If **encode** is set to **csv**, you can use this parameter to specify the separator between CSV fields. By default, the comma (,) is used.
quote	No	Quoted symbol in a data format. The attribute delimiters between two quoted symbols are treated as common characters. ● If double quotation marks are used as the quoted symbol, set this parameter to **\u005c \u0022** for character conversion. ● If a single quotation mark is used as the quoted symbol, set this parameter to a single quotation mark ('). **NOTE** ● Currently, only the CSV format is supported. ● After this parameter is specified, ensure that each field does not contain quoted symbols or contains an even number of quoted symbols. Otherwise, parsing will fail.

Parameter	Mandatory	Description
start_time	No	Start time when Kafka data is ingested. If this parameter is specified, DLI reads data read from the specified time. The format is **yyyy-MM-dd HH:mm:ss**. Ensure that the value of **start_time** is not later than the current time. Otherwise, no data will be obtained.
kafka_properties	No	This parameter is used to configure the native attributes of Kafka. The format is **key1=value1;key2=value2**.
kafka_certificate_name	No	Specifies the name of the datasource authentication information. This parameter is valid only when the datasource authentication type is set to **Kafka_SSL**. **NOTE** ● If this parameter is specified, the service loads only the specified file and password under the authentication. The system automatically sets this parameter to **kafka_properties**. ● Other configuration information required for Kafka SSL authentication needs to be manually configured in the **kafka_properties** attribute.

Precautions

When creating a source stream, you can specify a time model for subsequent calculation. Currently, DLI supports two time models: Processing Time and Event Time. For details about the syntax, see **Configuring Time Models**.

Example

- Read data from the Kafka topic **test**.
```
CREATE SOURCE STREAM kafka_source (
  name STRING,
  age int
 )
  WITH (
    type = "kafka",
    kafka_bootstrap_servers = "ip1:port1,ip2:port2",
    kafka_group_id = "sourcegroup1",
    kafka_topic = "test",
    encode = "json"
);
```
- Read the topic whose object is **test** from Kafka and use **json_config** to map JSON data to table fields.

  The data encoding format is non-nested JSON.
```
{"attr1": "lilei", "attr2": 18}
```
  The table creation statement is as follows:
```
CREATE SOURCE STREAM kafka_source (name STRING, age int)
WITH (
```

```
  type = "kafka",
  kafka_bootstrap_servers = "ip1:port1,ip2:port2",
  kafka_group_id = "sourcegroup1",
  kafka_topic = "test",
  encode = "json",
  json_config = "name=attr1;age=attr2"
);
```

2.3.4 Open-Source Kafka Source Stream

Function

Create a source stream to obtain data from Kafka as input data for jobs.

Apache Kafka is a fast, scalable, and fault-tolerant distributed message publishing and subscription system. It delivers high throughput and built-in partitions and provides data replicas and fault tolerance. Apache Kafka is applicable to scenarios of handling massive messages.

Prerequisites

- Kafka is an offline cluster. You need to use the enhanced datasource connection function to connect Flink jobs to Kafka. You can also set security group rules as required.

Syntax

```
CREATE SOURCE STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "kafka",
    kafka_bootstrap_servers = "",
    kafka_group_id = "",
    kafka_topic = "",
    encode = "json",
    json_config=""
  );
```

Keywords

Table 2-4 Keyword description

Parameter	Mandatory	Description
type	Yes	Data source type. **Kafka** indicates that the data source is Kafka.
kafka_bootstrap_servers	Yes	Port that connects DLI to Kafka. Use enhanced datasource connections to connect DLI queues with Kafka clusters.
kafka_group_id	No	Group ID.
kafka_topic	Yes	Kafka topic to be read. Currently, only one topic can be read at a time.

Parameter	Mandatory	Description
encode	Yes	Data encoding format. The value can be **csv**, **json**, **blob**, or **user_defined**. ● **field_delimiter** must be specified if this parameter is set to **csv**. ● **json_config** must be specified if this parameter is set to **json**. ● If this parameter is set to **blob**, the received data will not be parsed, and only one Array[TINYINT] field exists in the table. ● **encode_class_name** and **encode_class_parameter** must be specified if this parameter is set to **user_defined**.
encode_class_name	No	If **encode** is set to **user_defined**, you need to set this parameter to the name of the user-defined decoding class (including the complete package path). The class must inherit the **Deserialization-Schema** class.
encode_class_parameter	No	If **encode** is set to **user_defined**, you can set this parameter to specify the input parameter of the user-defined decoding class. Only one parameter of the string type is supported.
json_config	No	If **encode** is set to **json**, you can use this parameter to specify the mapping between JSON fields and stream attributes. The format is field1=json_field1;field2=json_field2. **field1** and **field2** indicate the names of the created table fields. **json_field1** and **json_field2** are key fields of the JSON strings in the Kafka input data. For details, see **Example**. **NOTE** If the attribute names in the source stream are the same as those in JSON fields, you do not need to set this parameter.
field_delimiter	No	If **encode** is set to **csv**, you can use this parameter to specify the separator between CSV fields. By default, the comma (,) is used.

Parameter	Mandatory	Description
quote	No	Quoted symbol in a data format. The attribute delimiters between two quoted symbols are treated as common characters. • If double quotation marks are used as the quoted symbol, set this parameter to **\u005c \u0022** for character conversion. • If a single quotation mark is used as the quoted symbol, set this parameter to a single quotation mark ('). **NOTE** • Currently, only the CSV format is supported. • After this parameter is specified, ensure that each field does not contain quoted symbols or contains an even number of quoted symbols. Otherwise, parsing will fail.
start_time	No	Start time when Kafka data is ingested. If this parameter is specified, DLI reads data read from the specified time. The format is **yyyy-MM-dd HH:mm:ss**. Ensure that the value of **start_time** is not later than the current time. Otherwise, no data will be obtained. If you set this parameter, only the data generated after the specified time for the Kafka topic will be read.
kafka_properties	No	Native properties of Kafka. The format is **key1=value1;key2=value2**. For details about the property values, see the description in **Apache Kafka**.
kafka_certificate_name	No	Name of the datasource authentication information. This parameter is valid only when the datasource authentication type is set to **Kafka_SSL**. **NOTE** • If this parameter is specified, the service loads only the specified file and password under the authentication. The system automatically sets this parameter to **kafka_properties**. • Other configuration information required for Kafka SSL authentication needs to be manually configured in the **kafka_properties** attribute.

Precautions

When creating a source stream, you can specify a time model for subsequent calculation. Currently, DLI supports two time models: Processing Time and Event Time. For details about the syntax, see **Configuring Time Models**.

Example

- Read Kafka topic **test**. The data encoding format is non-nested JSON, for example, {"attr1": "lilei", "attr2": 18}.
```
CREATE SOURCE STREAM kafka_source (name STRING, age int)
WITH (
  type = "kafka",
  kafka_bootstrap_servers = "ip1:port1,ip2:port2",
  kafka_group_id = "sourcegroup1",
  kafka_topic = "test",
  encode = "json",
  json_config = "name=attr1;age=attr2"
);
```
- Read Kafka topic **test**. The data is encoded in JSON format and nested. This example uses the complex data type ROW. For details about the syntax of ROW, see **Data Type**.

  The test data is as follows:
```
{
    "id":"1",
    "type2":"online",
    "data":{
        "patient_id":1234,
        "name":"bob1234"
    }
}
```
  An example of the table creation statements is as follows:
```
CREATE SOURCE STREAM kafka_source
(
  id STRING,
  type2 STRING,
  data ROW<
    patient_id STRING,
    name STRING>
)
WITH (
  type = "kafka",
  kafka_bootstrap_servers = "ip1:port1,ip2:port2",
  kafka_group_id = "sourcegroup1",
  kafka_topic = "test",
  encode = "json"
);

CREATE SINK STREAM kafka_sink
(
  id STRING,
  type2 STRING,
  patient_id STRING,
  name STRING
)
  WITH (
    type="kafka",
    kafka_bootstrap_servers =  "ip1:port1,ip2:port2",
    kafka_topic = "testsink",
    encode = "csv"
  );

INSERT INTO kafka_sink select id, type2, data.patient_id, data.name from kafka_source;
```

2.3.5 OBS Source Stream

Function

Create a source stream to obtain data from OBS. DLI reads data stored by users in OBS as input data for jobs. OBS applies to various scenarios, such as big data analysis, cloud-native application program data, static website hosting, backup/active archive, and deep/cold archive.

OBS is an object-based storage service. It provides massive, secure, highly reliable, and low-cost data storage capabilities. For more information about OBS, see the .

Syntax

```
CREATE SOURCE STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "obs",
    region = "",
    bucket = "",
    object_name = "",
    row_delimiter = "\n",
    field_delimiter = '',
    version_id = ""
  );
```

Keyword

Table 2-5 Keyword description

Parameter	Mandatory	Description
type	Yes	Data source type. **obs** indicates that the data source is OBS.
region	Yes	Region to which OBS belongs.
encode	No	Data encoding format. The value can be **csv** or **json**. The default value is **csv**.
ak	No	Access Key ID (AK).
sk	No	Secret access key used together with the ID of the access key.
bucket	Yes	Name of the OBS bucket where data is located.
object_name	Yes	Name of the object stored in the OBS bucket where data is located. If the object is not in the OBS root directory, you need to specify the folder name, for example, **test/test.csv**. For the object file format, see the **encode** parameter.
row_delimiter	Yes	Separator used to separate every two rows.
field_delimiter	No	Separator used to separate every two attributes. ● This parameter is mandatory when **encode** is **csv**. You use custom attribute separators. ● If **encode** is **json**, you do not need to set this parameter.

Parameter	Mandatory	Description
quote	No	Quoted symbol in a data format. The attribute delimiters between two quoted symbols are treated as common characters. ● If double quotation marks are used as the quoted symbol, set this parameter to **\u005c\u0022** for character conversion. ● If a single quotation mark is used as the quoted symbol, set this parameter to a single quotation mark ('). **NOTE** ● Currently, only the CSV format is supported. ● After this parameter is specified, ensure that each field does not contain quoted symbols or contains an even number of quoted symbols. Otherwise, parsing will fail.
version_id	No	Version number. This parameter is optional and required only when the OBS bucket or object has version settings.

Precautions

When creating a source stream, you can specify a time model for subsequent calculation. Currently, DLI supports two time models: Processing Time and Event Time. For details about the syntax, see **Configuring Time Models**.

Example

- The **input.csv** file is read from the OBS bucket. Rows are separated by **'\n'** and columns are separated by **','**.

  To use the test data, create an **input.txt** file, copy and paste the following text data, and save the file as **input.csv**. Upload the **input.csv** file to the target OBS bucket directory. For example, upload the file to the **dli-test-obs01** bucket directory.

  ```
  1,2,3,4,1403149534
  5,6,7,8,1403149535
  ```

  The following is an example for creating the table:

  ```
  CREATE SOURCE STREAM car_infos (
    car_id STRING,
    car_owner STRING,
    car_brand STRING,
    car_price INT,
    car_timestamp LONG
  )
    WITH (
      type = "obs",
      bucket = "dli-test-obs01",
      region = "xxx",
      object_name = "input.csv",
      row_delimiter = "\n",
      field_delimiter = ","
  );
  ```

- The **input.json** file is read from the OBS bucket. Rows are separated by **'\n'**.

```
CREATE SOURCE STREAM obs_source (
  str STRING
)
  WITH (
    type = "obs",
    bucket = "obssource",
    region = "xxx",
    encode = "json",
    row_delimiter = "\n",
    object_name = "input.json"
);
```

2.4 Creating a Sink Stream

2.4.1 MRS OpenTSDB Sink Stream

Function

DLI exports the output data of the Flink job to OpenTSDB of MRS.

Prerequisites

- OpenTSDB has been installed in the MRS cluster.
- In this scenario, jobs must run on the dedicated queue of DLI. Therefore, DLI must interconnect with the enhanced datasource connection that has been connected with MRS clusters. You can also set the security group rules as required.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "opentsdb",
    region = "",
    tsdb_metrics = "",
    tsdb_timestamps = "",
    tsdb_values = "",
    tsdb_tags = "",
    batch_insert_data_num = ""
  )
```

Keywords

Table 2-6 Keyword description

Parameter	Mandatory	Description
type	Yes	Sink channel type. **opentsdb** indicates that data is exported to OpenTSDB of MRS.
region	Yes	Region where MRS resides.

Parameter	Mandatory	Description
tsdb_link_address	Yes	Service address of the OpenTSDB instance in MRS. The format is **http://ip:port** or **https://ip:port**. **NOTE** If **tsd.https.enabled** is set to **true**, HTTPS must be used. Note that HTTPS does not support certificate authentication.
tsdb_metrics	Yes	Metric of a data point, which can be specified through parameter configurations.
tsdb_timestamps	Yes	Timestamp of a data point. The data type can be LONG, INT, SHORT, or STRING. Only dynamic columns are supported.
tsdb_values	Yes	Value of a data point. The data type can be SHORT, INT, LONG, FLOAT, DOUBLE, or STRING. Dynamic columns or constant values are supported.
tsdb_tags	Yes	Tags of a data point. Each of tags contains at least one tag value and up to eight tag values. Tags of the data point can be specified through parameter configurations.
batch_insert_data_num	No	Number of data records to be written in batches at a time. The value must be a positive integer. The upper limit is **65536**. The default value is **8**.

Precautions

If a configuration item can be specified through parameter configurations, one or more columns in the record can be used as part of the configuration item. For example, if the configuration item is set to **car_$ {car_brand}** and the value of **car_brand** in a record is **BMW**, the value of this configuration item is **car_BMW** in the record.

Example

Output data of stream **weather_out** to OpenTSDB of MRS.

```
CREATE SINK STREAM weather_out (
  timestamp_value LONG, /* Time */
  temperature FLOAT, /* Temperature value */
  humidity FLOAT, /* Humidity */
  location STRING /* Location */
)
  WITH (
    type = "opentsdb",
    region = "xxx",
    tsdb_link_address = "https://x.x.x.x:4242",
    tsdb_metrics = "weather",
    tsdb_timestamps = "${timestamp_value}",
    tsdb_values = "${temperature}; ${humidity}",
    tsdb_tags = "location:${location},signify:temperature; location:${location},signify:humidity",
    batch_insert_data_num = "10"
);
```

2.4.2 CSS Elasticsearch Sink Stream

Function

DLI exports Flink job output data to Elasticsearch of Cloud Search Service (CSS). Elasticsearch is a popular enterprise-class Lucene-powered search server and provides the distributed multi-user capabilities. It delivers multiple functions, including full-text retrieval, structured search, analytics, aggregation, and highlighting. With Elasticsearch, you can achieve stable, reliable, real-time search. Elasticsearch applies to diversified scenarios, such as log analysis and site search.

CSS is a fully managed, distributed search service. It is fully compatible with open-source Elasticsearch and provides DLI with structured and unstructured data search, statistics, and report capabilities.

NOTE

If the security mode is enabled when you create a CSS cluster, it cannot be undone.

Prerequisites

- In this scenario, jobs must run on the dedicated queue of DLI. Therefore, DLI must interconnect with the enhanced datasource connection that has been connected with CSS. You can also set the security group rules as required.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
 WITH (
   type = "es",
   region = "",
   cluster_address = "",
   es_index = "",
   es_type= "",
   es_fields= "",
   batch_insert_data_num= ""
 );
```

Keyword

Table 2-7 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **es** indicates that data is exported to CSS.
region	Yes	Region where CSS is located.
cluster_address	Yes	Private access address of the CSS cluster, for example: **x.x.x.x:x**. Use commas (,) to separate multiple addresses.

Parameter	Mandatory	Description
es_index	Yes	Index of the data to be inserted. This parameter corresponds to CSS index.
es_type	Yes	Type of the document to which data is to be inserted. This parameter corresponds to the CSS type. If the Elasticsearch version is 6.x, the value cannot start with an underscore (_). If the Elasticsearch version is 7.x and the type of CSS is preset, the value must be **_doc**. Otherwise, the value must comply with CSS specifications.
es_fields	Yes	Key of the data field to be inserted. The format is **id,f1,f2,f3,f4**. Ensure that the key corresponds to the data column in the sink. If a random attribute field instead of a key is used, the keyword **id** does not need to be used, for example, **f1,f2,f3,f4,f5**. This parameter corresponds to the CSS filed.
batch_insert_data_num	Yes	Amount of data to be written in batches at a time. The value must be a positive integer. The unit is 10 records. The maximum value allowed is **65536**, and the default value is **10**.
action	No	If the value is **add**, data is forcibly overwritten when the same ID is encountered. If the value is **upsert**, data is updated when the same ID is encountered. (If **upsert** is selected, **id** in the **es_fields** field must be specified.) The default value is **add**.
enable_output_null	No	This parameter is used to configure whether to generate an empty field. If this parameter is set to **true**, an empty field (the value is **null**) is generated. If set to **false**, no empty field is generated. The default value is **false**.
max_record_num_cache	No	Maximum number of records that can be cached.
es_certificate_name	No	Name of the datasource authentication information If the security mode is enabled and HTTPS is used by the Elasticsearch cluster, the certificate is required for access. In this case, set the datasource authentication type to **CSS**. If the security mode is enabled for the Elasticsearch cluster but HTTP is disabled, the username and password are required for access. In this case, set the datasource authentication type to **Password**.

Precautions

If a configuration item can be specified through parameter configurations, one or more columns in the record can be used as part of the configuration item. For example, if the configuration item is set to **car_$ {car_brand}** and the value of **car_brand** in a record is **BMW**, the value of this configuration item is **car_BMW** in the record.

Example

Data of stream **qualified_cars** is exported to the cluster on CSS.

```
CREATE SINK STREAM qualified_cars (
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed INT,
  total_miles INT
)
  WITH (
    type = "es",
    region = "xxx",
    cluster_address = "192.168.0.212:9200",
    es_index = "car",
    es_type = "information",
    es_fields = "id,owner,age,speed,miles",
    batch_insert_data_num = "10"
);
```

2.4.3 DCS Sink Stream

Function

DLI exports the Flink job output data to Redis of DCS. Redis is a storage system that supports multiple types of data structures such as key-value. It can be used in scenarios such as caching, event pub/sub, and high-speed queuing. Redis supports direct read/write of strings, hashes, lists, queues, and sets. Redis works with in-memory dataset and provides persistence. For more information about Redis, visit **https://redis.io/**.

DCS provides Redis-compatible, secure, reliable, out-of-the-box, distributed cache capabilities allowing elastic scaling and convenient management. It meets users' requirements for high concurrency and fast data access.

Prerequisites

- Ensure that You have created a Redis cache instance on DCS using your account.
- In this scenario, jobs must run on the dedicated queue of DLI. Therefore, DLI must be interconnected with the DCS clusters. You can also set the security group rules as required.
- If you use a VPC peering connection to access a DCS instance, the following restrictions also apply:
  - If network segment 172.16.0.0/12~24 is used during DCS instance creation, the DLI queue cannot be in any of the following network segments: 192.168.1.0/24, 192.168.2.0/24, and 192.168.3.0/24.

- If network segment 192.168.0.0/16~24 is used during DCS instance creation, the DLI queue cannot be in any of the following network segments: 172.31.1.0/24, 172.31.2.0/24, and 172.31.3.0/24.
- If network segment 10.0.0.0/8~24 is used during DCS instance creation, the DLI queue cannot be in any of the following network segments: 172.31.1.0/24, 172.31.2.0/24, and 172.31.3.0/24.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "dcs_redis",
    region = "",
    cluster_address = "",
    password = "",
    value_type= "",key_value= ""
  );
```

Keyword

Table 2-8 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **dcs_redis** indicates that data is exported to DCS Redis.
region	Yes	Region where DCS for storing the data is located.
cluster_address	Yes	Redis instance connection address.
password	No	Redis instance connection password. This parameter is not required if password-free access is used.
value_type	Yes	This parameter can be set to any or the combination of the following options: ● Data types, including **string**, **list**, **hash**, **set**, and **zset** ● Commands used to set the expiration time of a key, including **expire**, **pexpire**, **expireAt**, and **pexpireAt** ● Commands used to delete a key, including **del** and **hdel** Use commas (,) to separate multiple commands.
key_value	Yes	Key and value. The number of key_value pairs must be the same as the number of types specified by value_type, and key_value pairs are separated by semicolons (;). Both key and value can be specified through parameter configurations. The dynamic column name is represented by ${column name}.

Precautions

- If a configuration item can be specified through parameter configurations, one or more columns in the record can be used as part of the configuration item. For example, if the configuration item is set to **car_$ {car_brand}** and the value of **car_brand** in a record is **BMW**, the value of this configuration item is **car_BMW** in the record.
- Characters ":", ",", ";", "$", "{", and "}" have been used as special separators without the escape function. These characters cannot be used in key and value as common characters. Otherwise, parsing will be affected and the program exceptions will occur.

Example

Data of stream **qualified_cars** is exported to the Redis cache instance on DCS.

```
CREATE SINK STREAM qualified_cars (
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed DOUBLE,
  total_miles DOUBLE
)
  WITH (
    type = "dcs_redis",
    cluster_address = "192.168.0.34:6379",
    password = "xxxxxxxx",
    value_type = "string; list; hash; set; zset",
    key_value = "${car_id}_str: ${car_owner}; name_list: ${car_owner}; ${car_id}_hash: {name:${car_owner}, age: ${car_age}}; name_set:   ${car_owner}; math_zset: {${car_owner}:${average_speed}}"
  );
```

2.4.4 DDS Sink Stream

Function

DLI outputs the job output data to Document Database Service (DDS).

DDS is compatible with the MongoDB protocol and is secure, highly available, reliable, scalable, and easy to use. It provides DB instance creation, scaling, redundancy, backup, restoration, monitoring, and alarm reporting functions with just a few clicks on the DDS console.

Prerequisites

- Ensure that you have created a DDS instance on DDS using your account.

  For details about how to create a DDS instance, see **Buying a DDS DB Instance** in the .
- Currently, only cluster instances with SSL authentication disabled are supported. Replica set and single node instances are not supported.
- In this scenario, jobs must run on the dedicated queue of DLI. Ensure that the dedicated queue of DLI has been created.
- Ensure that a datasource connection has been set up between the DLI dedicated queue and the DDS cluster, and security group rules have been configured based on the site requirements.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "dds",
    username = "",
    password = "",
    db_url = "",
    field_names = ""
  );
```

Keyword

Table 2-9 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **dds** indicates that data is exported to DDS.
username	Yes	Username for connecting to a database.
password	Yes	Password for connecting to a database.
db_url	Yes	DDS instance access address, for example, **ip1:port,ip2:port/database/collection**.
field_names	Yes	Key of the data field to be inserted. The format is **f1,f2,f3**. Ensure that the key corresponds to the data column in the sink stream.
batch_insert_data_num	No	Amount of data to be written in batches at a time. The value must be a positive integer. The default value is **10**.

Example

Output data in the **qualified_cars** stream to the **collectionTest** DDS DB.

```
CREATE SINK STREAM qualified_cars (
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed INT,
  total_miles INT
)
  WITH (
    type = "dds",
    region = "xxx",
    db_url = "192.168.0.8:8635,192.168.0.130:8635/dbtest/collectionTest",
    username = "xxxxxxxxxx",
    password =  "xxxxxxxxxx",
    field_names = "car_id,car_owner,car_age,average_speed,total_miles",
    batch_insert_data_num = "10"
  );
```


2.4.5 DIS Sink Stream

Function

DLI writes the Flink job output data into DIS. This cloud ecosystem is applicable to scenarios where data is filtered and imported to the DIS stream for future processing.

DIS addresses the challenge of transmitting data outside cloud services to cloud services. DIS builds data intake streams for custom applications capable of processing or analyzing streaming data. DIS continuously captures, transmits, and stores terabytes of data from hundreds of thousands of sources every hour, such as logs, Internet of Things (IoT) data, social media feeds, website clickstreams, and location-tracking events. For more information about DIS, see the .

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "dis",
    region = "",
    channel = "",
    partition_key = "",
    encode= "",
    field_delimiter= ""
  );
```

Keyword

Table 2-10 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **dis** indicates that data is exported to DIS.
region	Yes	Region where DIS for storing the data is located.
ak	No	Access Key ID (AK).
sk	No	Specifies the secret access key used together with the ID of the access key.
channel	Yes	DIS stream.
partition_key	No	Group primary key. Multiple primary keys are separated by commas (,). If this parameter is not specified, data is randomly written to DIS partitions.

Parameter	Mandatory	Description
encode	Yes	Data encoding format. The value can be **csv**, **json**, or **user_defined**. **NOTE** • **field_delimiter** must be specified if this parameter is set to **csv**. • If the encoding format is **json**, you need to configure **enable_output_null** to determine whether to generate an empty field. For details, see the examples. • **encode_class_name** and **encode_class_parameter** must be specified if this parameter is set to **user_defined**.
field_delimiter	Yes	Separator used to separate every two attributes. • This parameter needs to be configured if the CSV encoding format is adopted. It can be user-defined, for example, a comma (**,**). • This parameter is not required if the JSON encoding format is adopted.
json_config	No	If **encode** is set to **json**, you can set this parameter to specify the mapping between the JSON field and the stream definition field. An example of the format is as follows: field1=data_json.field1; field2=data_json.field2.
enable_output_null	No	If **encode** is set to **json**, you need to specify this parameter to determine whether to generate an empty field. If this parameter is set to **true**, an empty field (the value is **null**) is generated. If set to **false**, no empty field is generated. The default value is **true**.
encode_class_name	No	If **encode** is set to **user_defined**, you need to set this parameter to the name of the user-defined decoding class (including the complete package path). The class must inherit the **Deserialization-Schema** class.
encode_class_parameter	No	If **encode** is set to **user_defined**, you can set this parameter to specify the input parameter of the user-defined decoding class. Only one parameter of the string type is supported.

Precautions

None

Example

- CSV: Data is written to the DIS stream and encoded using CSV. CSV fields are separated by commas (,). If there are multiple partitions, car_owner is used as the key to distribute data to different partitions. An example is as follows: "ZJA710XC", "lilei", "BMW", 700000.

```
CREATE SINK STREAM audi_cheaper_than_30w (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
    type = "dis",
    region = "xxx",
    channel = "dlioutput",
    encode = "csv",
    field_delimiter = ","
);
```

- JSON: Data is written to the DIS stream and encoded using JSON. If there are multiple partitions, car_owner and car_brand are used as the keys to distribute data to different partitions. If **enableOutputNull** is set to **true**, an empty field (the value is **null**) is generated. If set to **false**, no empty field is generated. An example is as follows: "car_id ":"ZJA710XC", "car_owner ":"lilei", "car_brand ":"BMW", "car_price ":700000.

```
CREATE SINK STREAM audi_cheaper_than_30w (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
    type = "dis",
    channel = "dlioutput",
    region = "xxx",
    partition_key = "car_owner,car_brand",
    encode = "json",
    enable_output_null = "false"
);
```

2.4.6 DMS Sink Stream

DMS (Distributed Message Service) is a message middleware service based on distributed, high-availability clustering technology. It provides reliable, scalable, fully managed queues for sending, receiving, and storing messages. DMS for Kafka is a message queuing service based on Apache Kafka. This service provides Kafka premium instances.

DLI can write the job output data into the Kafka instance. The syntax for creating a Kafka sink stream is the same as that for creating an open source Apache Kafka sink stream. For details, see **MRS Kafka Sink Stream**.

2.4.7 DWS Sink Stream (JDBC Mode)

Function

DLI outputs the Flink job output data to Data Warehouse Service (DWS). DWS database kernel is compliant with PostgreSQL. The PostgreSQL database can store data of more complex types and delivers space information services, multi-version concurrent control (MVCC), and high concurrency. It applies to location applications, financial insurance, and e-commerce.

DWS is an online data processing database based on the cloud infrastructure and platform and helps you mine and analyze massive sets of data. For more information about DWS, see the .

Prerequisites

- Ensure that you have created a DWS cluster on DWS using your account.

  For details about how to create a DWS cluster, see **Creating a Cluster** in the *Data Warehouse Service Management Guide*.
- Ensure that a DWS database table has been created.
- In this scenario, jobs must run on the dedicated queue of DLI. Therefore, DLI must interconnect with the enhanced datasource connection that has been connected with DWS clusters. You can also set the security group rules as required.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "rds",
    username = "",
    password = "",
    db_url = "",
    table_name = ""
  );
```

Keyword

Table 2-11 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **rds** indicates that data is exported to RDS or DWS.
username	Yes	Username for connecting to a database.
password	Yes	Password for connecting to a database.
db_url	Yes	Database connection address, for example, **postgresql://ip:port/database**.
table_name	Yes	Name of the table where data will be inserted. You need to create the database table in advance.

<table>
<tr><th>Parameter</th><th>Mandatory</th><th>Description</th></tr>
<tr><td>db_columns</td><td>No</td><td>Mapping between attributes in the output stream and those in the database table. This parameter must be configured based on the sequence of attributes in the output stream.

Example:<pre>create sink stream a3(student_name string, student_age int)
 with (
  type = "rds",
  username = "root",
  password = "xxxxxxxx",
  db_url = "postgresql://192.168.0.102:8000/test1",
  db_columns = "name,age",
  table_name = "t1"
 );</pre>In the example, student_name corresponds to the name attribute in the database, and student_age corresponds to the age attribute in the database.

NOTE
If db_columns is not configured, it is normal that the number of attributes in the output stream is less than that of attributes in the database table and the extra attributes in the database table are all nullable or have default values.</td></tr>
<tr><td>primary_key</td><td>No</td><td>To update data in the table in real time by using the primary key, add the primary_key configuration item (c_timeminute in the following example) when creating a table. During the data writing operation, data is updated if the specified primary_key exists. Otherwise, data is inserted.

Example:<pre>CREATE SINK STREAM test(c_timeminute LONG, c_cnt LONG)
 WITH (
  type = "rds",
  username = "root",
  password = "xxxxxxxx",
  db_url = "postgresql://192.168.0.12:8000/test",
  table_name = "test",
  primary_key = "c_timeminute"
 );</pre></td></tr>
</table>

Precautions

The stream format defined by **stream_id** must be the same as the table format.

Example

Data of stream **audi_cheaper_than_30w** is exported to the **audi_cheaper_than_30w** table in the **test** database.

```
CREATE SINK STREAM audi_cheaper_than_30w (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
```

```
 WITH (
   type = "rds",
   username = "root",
   password = "xxxxxx",
   db_url = "postgresql://192.168.1.1:8000/test",
   table_name = "audi_cheaper_than_30w"
 );

insert into audi_cheaper_than_30w select "1","2","3",4;
```

2.4.8 DWS Sink Stream (OBS-based Dumping)

Function

Create a sink stream to export Flink job data to DWS through OBS-based dumping, specifically, output Flink job data to OBS and then import data from OBS to DWS. For details about how to import OBS data to DWS, see **Concurrently Importing Data from OBS** in the .

DWS is an online data processing database based on the cloud infrastructure and platform and helps you mine and analyze massive sets of data. For more information about DWS, see the .

Precautions

- OBS-based dumping supports intermediate files of the following two types:
  - ORC: The ORC format does not support array data type. If the ORC format is used, create a foreign server in DWS. For details, see **Creating a Foreign Server** in the .
  - CSV: By default, the line break is used as the record separator. If the line break is contained in the attribute content, you are advised to configure quote. For details, see **Table 2-12**.
- If the target table does not exist, a table is automatically created. DLI data of the SQL type does not support **text**. If a long text exists, you are advised to create a table in the database.
- When **encode** uses the ORC format to create a DWS table, if the field attribute of the SQL stream is defined as the **String** type, the field attribute of the DWS table cannot use the **varchar** type. Instead, a specific text type must be used. If the SQL stream field attribute is defined as the **Integer** type, the DWS table field must use the **Integer** type.

Prerequisites

- Ensure that OBS buckets and folders have been created.
- In this scenario, jobs must run on the dedicated queue of DLI. Therefore, DLI must interconnect with the enhanced datasource connection that has been connected with DWS clusters. You can also set the security group rules as required.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
 WITH (
        type = "dws",
        region = "",
        ak = "",
```

```
        sk = "",
        encode = "",
        field_delimiter = "",
        quote = "",
        db_obs_server = "",
        obs_dir = "",
        username = "",
        password =  "",
        db_url = "",
        table_name = "",
        max_record_num_per_file = "",
        dump_interval = ""
);
```

Keyword

Table 2-12 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **dws** indicates that data is exported to DWS.
region	Yes	Region where DWS is located.
ak	Yes	Access Key ID (AK).
sk	Yes	Secret access key used together with the ID of the AK.
encode	Yes	Encoding format. Currently, CSV and ORC are supported.
field_delimiter	No	Separator used to separate every two attributes. This parameter needs to be configured if the CSV encoding mode is used. It is recommended that you use invisible characters as separators, for example, **\u0006\u0002**.
quote	No	Single byte. It is recommended that invisible characters be used, for example, **u0007**.
db_obs_server	No	Foreign server (for example, **obs_server**) that has been created in the database. You need to specify this parameter if the ORC encoding mode is adopted.
obs_dir	Yes	Directory for storing intermediate files. The directory is in the format of {Bucket name}/{Directory name}, for example, obs-a1/dir1/subdir.
username	Yes	Username for connecting to a database.
password	Yes	Password for connecting to a database.
db_url	Yes	Database connection address. The format is /ip:port/database, for example, **192.168.1.21:8000/test1**.

Parameter	Mandatory	Description
table_name	Yes	Data table name. If no table is available, a table is automatically created.
max_record_num_per_file	Yes	Maximum number of records that can be stored in a file. If the number of records in a file is less than the maximum value, the file will be dumped to OBS after one dumping period.
dump_interval	Yes	Dumping period. The unit is second.
delete_obs_temp_file	No	Whether to delete temporary files on OBS. The default value is **true**. If this parameter is set to **false**, files on OBS will not be deleted. You need to manually clear the files.
max_dump_file_num	No	Maximum number of files that can be dumped at a time. If the number of files to be dumped is less than the maximum value, the files will be dumped to OBS after one dumping period.

Example

- Dump files in CSV format.
```
CREATE SINK STREAM car_infos (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT,
  car_timestamp LONG
)
  WITH (
    type = "dws",
    region = "xxx",
    ak = "",
    sk = "",
    encode = "csv",
    field_delimiter = "\u0006\u0006\u0002",
    quote = "\u0007",
    obs_dir = "dli-append-2/dws",
    username = "",
    password =  "",
    db_url = "192.168.1.12:8000/test1",
    table_name = "table1",
    max_record_num_per_file = "100",
    dump_interval = "10"
  );
```
- Dump files in ORC format.
```
CREATE SINK STREAM car_infos (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT,
  car_timestamp LONG
)
  WITH (
    type = "dws",
```

```
  region = "xxx",
  ak = "",
  sk = "",
  encode = "orc",
  db_obs_server = "obs_server",
  obs_dir = "dli-append-2/dws",
  username = "",
  password =  "",
  db_url = "192.168.1.12:8000/test1",
  table_name = "table1",
  max_record_num_per_file = "100",
  dump_interval = "10"
);
```

2.4.9 MRS HBase Sink Stream

Function

DLI exports the output data of the Flink job to HBase of MRS.

Prerequisites

- An MRS cluster has been created by using your account. DLI can interconnect with HBase clusters with Kerberos enabled.
- In this scenario, jobs must run on the dedicated queue of DLI. Ensure that the dedicated queue of DLI has been created.
- Ensure that a datasource connection has been set up between the DLI dedicated queue and the MRS cluster, and security group rules have been configured based on the site requirements.
- **If you use MRS HBase, ensure that you have added IP addresses of all hosts in the MRS cluster for the enhanced datasource connection.**

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "mrs_hbase",
    region = "",
    cluster_address = "",
    table_name = "",
    table_columns = "",
    illegal_data_table = "",
    batch_insert_data_num = "",
    action = ""
)
```

Keyword

Table 2-13 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **mrs_hbase** indicates that data is exported to HBase of MRS.
region	Yes	Region where MRS resides.

Parameter	Mandatory	Description
cluster_address	Yes	ZooKeeper address of the cluster to which the data table to be inserted belongs. The format is **ip1,ip2:port**.
table_name	Yes	Name of the table where data is to be inserted. It can be specified through parameter configurations. For example, if you want one or more certain columns as part of the table name, use **car_pass_inspect_with_age_${car_age}**, where **car_age** is the column name.
table_columns	Yes	Columns to be inserted. The format is **rowKey, f1:c1, f1:c2, f2:c1**, where **rowKey** must be specified. If you do not want to add a column (for example, the third column) to the database, set this parameter to **rowKey,f1:c1,,f2:c1**.
illegal_data_table	No	If this parameter is specified, abnormal data (for example, **rowKey** does not exist) will be written into the table. If not specified, abnormal data will be discarded. The **rowKey** value is **taskNo**_*Timestamp* followed by six random digits, and the schema is info:data, info:reason.
batch_insert_data_num	No	Number of data records to be written in batches at a time. The value must be a positive integer. The upper limit is **1000**. The default value is **10**.
action	No	Whether data is added or deleted. Available options include **add** and **delete**. The default value is **add**.
krb_auth	No	Authentication name for creating a datasource connection authentication. This parameter is mandatory when Kerberos authentication is enabled. Set this parameter to the corresponding cross-source authentication name. **NOTE** Ensure that the **/etc/hosts** information of the master node in the MRS cluster is added to the host file of the DLI queue.

Precautions

None

Example

Output data to HBase of MRS.

```
CREATE SINK STREAM qualified_cars (
  car_id STRING,
```

```
  car_owner STRING,
  car_age INT,
  average_speed INT,
  total_miles INT
)
  WITH (
    type = "mrs_hbase",
    region = "xxx",
    cluster_address = "192.16.0.88,192.87.3.88:2181",
    table_name = "car_pass_inspect_with_age_${car_age}",
    table_columns = "rowKey,info:owner,,car:speed,car:miles",
    illegal_data_table = "illegal_data",
    batch_insert_data_num = "20",
    action = "add",
    krb_auth = "KRB_AUTH_NAME"
  );
```

2.4.10 MRS Kafka Sink Stream

Function

DLI exports the output data of the Flink job to Kafka.

Apache Kafka is a fast, scalable, and fault-tolerant distributed message publishing and subscription system. It delivers high throughput and built-in partitions and provides data replicas and fault tolerance. Apache Kafka is applicable to scenarios of handling massive messages. Kafka clusters are deployed and hosted on MRS that is powered on Apache Kafka.

Prerequisites

- Kafka is an offline cluster. You need to use the enhanced datasource connection function to connect Flink jobs to Kafka. You can also set security group rules as required.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH(
    type = "kafka",
    kafka_bootstrap_servers = "",
    kafka_topic = "",
    encode = "json"
)
```

Keyword

Table 2-14 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **kafka** indicates that data is exported to Kafka.

Parameter	Mandatory	Description
kafka_bootstrap_servers	Yes	Port that connects DLI to Kafka. Use enhanced datasource connections to connect DLI queues with Kafka clusters.
kafka_topic	Yes	Kafka topic into which DLI writes data.
encode	Yes	Encoding format. Currently, **json** and **user_defined** are supported. **encode_class_name** and **encode_class_parameter** must be specified if this parameter is set to **user_defined**.
encode_class_name	No	If **encode** is set to **user_defined**, you need to set this parameter to the name of the user-defined decoding class (including the complete package path). The class must inherit the **DeserializationSchema** class.
encode_class_parameter	No	If **encode** is set to **user_defined**, you can set this parameter to specify the input parameter of the user-defined decoding class. Only one parameter of the string type is supported.
krb_auth	No	Authentication name for creating a datasource connection authentication. This parameter is mandatory when Kerberos authentication is enabled. If Kerberos authentication is not enabled for the created MRS cluster, ensure that the **/etc/hosts** information of the master node in the MRS cluster is added to the host file of the DLI queue.
kafka_properties	No	This parameter is used to configure the native attributes of Kafka. The format is **key1=value1;key2=value2**.
kafka_certificate_name	No	Specifies the name of the datasource authentication information. This parameter is valid only when the datasource authentication type is set to **Kafka_SSL**. **NOTE** ● If this parameter is specified, the service loads only the specified file and password under the authentication. The system automatically sets this parameter to **kafka_properties**. ● Other configuration information required for Kafka SSL authentication needs to be manually configured in the **kafka_properties** attribute.

Precautions

None

Example

Output data to Kafka.

- Example 1:
```
CREATE SINK STREAM kafka_sink (name STRING)
  WITH (
    type="kafka",
    kafka_bootstrap_servers =  "ip1:port1,ip2:port2",
    kafka_topic = "testsink",
    encode = "json"
  );
```
- Example 2:
```
CREATE SINK STREAM kafka_sink (
  a1 string,
  a2 string,
  a3 string,
  a4 INT
  ) // Output Field
  WITH (
    type="kafka",
    kafka_bootstrap_servers =  "192.x.x.x:9093, 192.x.x.x:9093, 192.x.x.x:9093",
kafka_topic = "testflink", // Written topic
  encode = "csv", // Encoding format, which can be JSON or CSV.
    kafka_certificate_name = "Flink",
    kafka_properties_delimiter = ",",
    kafka_properties = "sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
required username=\"xxx\" password=\"xxx\";,sasl.mechanism=PLAIN,security.protocol=SASL_SSL"
  );
```

2.4.11 Open-Source Kafka Sink Stream

Function

DLI exports the output data of the Flink job to Kafka.

Apache Kafka is a fast, scalable, and fault-tolerant distributed message publishing and subscription system. It delivers high throughput and built-in partitions and provides data replicas and fault tolerance. Apache Kafka is applicable to scenarios of handling massive messages.

Prerequisites

- Kafka is an offline cluster. You need to use the enhanced datasource connection function to connect Flink jobs to Kafka. You can also set security group rules as required.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH(
    type = "kafka",
    kafka_bootstrap_servers = "",
    kafka_topic = "",
    encode = "json"
  )
```

Keyword

Table 2-15 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **kafka** indicates that data is exported to Kafka.
kafka_bootstrap_servers	Yes	Port that connects DLI to Kafka. Use enhanced datasource connections to connect DLI queues with Kafka clusters.
kafka_topic	Yes	Kafka topic into which DLI writes data.
encode	Yes	Data encoding format. The value can be **csv**, **json**, or **user_defined**. • **field_delimiter** must be specified if this parameter is set to **csv**. • **encode_class_name** and **encode_class_parameter** must be specified if this parameter is set to **user_defined**.
filed_delimiter	No	If **encode** is set to **csv**, you can use this parameter to specify the separator between CSV fields. By default, the comma (,) is used.
encode_class_name	No	If **encode** is set to **user_defined**, you need to set this parameter to the name of the user-defined decoding class (including the complete package path). The class must inherit the **DeserializationSchema** class.
encode_class_parameter	No	If **encode** is set to **user_defined**, you can set this parameter to specify the input parameter of the user-defined decoding class. Only one parameter of the string type is supported.
kafka_properties	No	This parameter is used to configure the native attributes of Kafka. The format is **key1=value1;key2=value2**.
kafka_certificate_name	No	Name of the datasource authentication information. This parameter is valid only when the datasource authentication type is set to **Kafka_SSL**. **NOTE** • If this parameter is specified, the service loads only the specified file and password under the authentication. The system automatically sets this parameter to **kafka_properties**. • Other configuration information required for Kafka SSL authentication needs to be manually configured in the **kafka_properties** attribute.

Precautions

None

Example

Output the data in the kafka_sink stream to Kafka.

```
CREATE SINK STREAM kafka_sink (name STRING)
  WITH (
    type="kafka",
    kafka_bootstrap_servers =  "ip1:port1,ip2:port2",
    kafka_topic = "testsink",
    encode = "json"
  );
```

2.4.12 File System Sink Stream (Recommended)

Function

You can create a sink stream to export data to a file system such as HDFS or OBS. After the data is generated, a table can be created directly according to the generated directory. The table can be processed through DLI SQL, and the output data directory can be stored in partitioned tables. It is applicable to scenarios such as data dumping, big data analysis, data backup, and active, deep, or cold archiving.

OBS is an object-based storage service. It provides massive, secure, highly reliable, and low-cost data storage capabilities.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  [PARTITIONED BY (attr_name (',' attr_name)*]
  WITH (
    type = "filesystem",
    file.path = "obs://bucket/xx",
    encode = "parquet",
    ak = "",
    sk = ""
  );
```

Keywords

Table 2-16 Keyword description

Parameter	Mandatory	Description
type	Yes	Output stream type. If **type** is set to **filesystem**, data is exported to the file system.

Parameter	Mandatory	Description
file.path	Yes	Output directory in the form: **schema://file.path**. Currently, Schema supports only OBS and HDFS. ● If **schema** is set to **obs**, data is stored to OBS. ● If **schema** is set to **hdfs**, data is exported to HDFS. A proxy user needs to be configured for HDFS. For details, see **HDFS Proxy User Configuration**. Example: **hdfs://node-master1sYAx:9820/user/car_infos**, where **node-master1sYAx:9820** is the name of the node where the NameNode is located.
encode	Yes	Output data encoding format. Currently, only the **parquet** and **csv** formats are supported. ● When **schema** is set to **obs**, the encoding format of the output data can only be **parquet**. ● When **schema** is set to **hdfs**, the output data can be encoded in **Parquet** or **CSV** format.
ak	No	Access key. This parameter is mandatory when data is exported to OBS. Global variables can be used to mask the access key used for OBS authentication.
sk	No	Secret access key. This parameter is mandatory when data is exported to OBS. Secret key for accessing OBS authentication. Global variables can be used to mask sensitive information.
krb_auth	No	Authentication name for creating a datasource connection authentication. This parameter is mandatory when Kerberos authentication is enabled. If Kerberos authentication is not enabled for the created MRS cluster, ensure that the **/etc/hosts** information of the master node in the MRS cluster is added to the host file of the DLI queue.
field_delimiter	No	Separator used to separate every two attributes. This parameter needs to be configured if the CSV encoding format is adopted. It can be user-defined, for example, a comma (**,**).

Precautions

- To ensure job consistency, enable checkpointing if the Flink job uses the file system output stream.
- To avoid data loss or data coverage, you need to enable automatic or manual restart upon job exceptions. Enable the **Restore Job from Checkpoint**.
- Set the checkpoint interval after weighing between real-time output file, file size, and recovery time, such as 10 minutes.
- Two modes are supported.

- **At least once**: Events are processed at least once.
- **Exactly once**: Events are processed only once.

- When you use sink streams of a file system to write data into OBS, do not use multiple jobs for the same directory.
  - The default behavior of an OBS bucket is overwriting, which may cause data loss.
  - The default behavior of the OBS parallel file system bucket is appending, which may cause data confusion.

  You should carefully select the OBS bucket because of the preceding behavior differences. Data exceptions may occur after abnormal job restart.

HDFS Proxy User Configuration

1. Log in to the MRS management page.
2. Select the HDFS NameNode configuration of MRS and add configuration parameters in the **Customization** area.

   In the preceding information, **myname** in the **core-site** values **hadoop.proxyuser.myname.hosts** and **hadoop.proxyuser.myname.groups** is the name of the krb authentication user.

   NOTE

   Ensure that the permission on the HDFS data write path is **777**.
3. After the configuration is complete, click **Save**.

Example

- Example 1:

  The following example dumps the **car_info** data to OBS, with the **buyday** field as the partition field and **parquet** as the encoding format.

```
create sink stream car_infos (
  carId string,
  carOwner string,
  average_speed double,
  buyday string
  ) partitioned by (buyday)
  with (
    type = "filesystem",
    file.path = "obs://obs-sink/car_infos",
    encode = "parquet",
    ak = "{{myAk}}",
    sk = "{{mySk}}"
);
```

  The data is ultimately stored in OBS. Directory: **obs://obs-sink/car_infos/buyday=xx/part-x-x**.

  After the data is generated, the OBS partitioned table can be established for subsequent batch processing through the following SQL statements:

  a. Create an OBS partitioned table.

```
create table car_infos (
  carId string,
  carOwner string,
  average_speed double
)
  partitioned by (buyday string)
```

```
stored as parquet
location 'obs://obs-sink/car_infos';
```

b. Restore partition information from the associated OBS path.

```
alter table car_infos recover partitions;
```

- Example 2:

  The following example dumps the **car_info** data to HDFS, with the **buyday** field as the partition field and **csv** as the encoding format.

```
create sink stream car_infos (
  carId string,
  carOwner string,
  average_speed double,
  buyday string
  ) partitioned by (buyday)
  with (
    type = "filesystem",
    file.path = "hdfs://node-master1sYAx:9820/user/car_infos",
    encode = "csv",
    field_delimiter = ","
);
```

  The data is ultimately stored in HDFS. Directory: **/user/car_infos/buyday=xx/part-x-x**.

2.4.13 OBS Sink Stream

Function

Create a sink stream to export DLI data to OBS. DLI can export the job analysis results to OBS. OBS applies to various scenarios, such as big data analysis, cloud-native application program data, static website hosting, backup/active archive, and deep/cold archive.

OBS is an object-based storage service. It provides massive, secure, highly reliable, and low-cost data storage capabilities. For more information about OBS, see the .

NOTE

You are advised to use the **File System Sink Stream (Recommended)**.

Prerequisites

Before data exporting, check the version of the OBS bucket. The OBS sink stream supports data exporting to an OBS bucket running OBS 3.0 or a later version.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
         type = "obs",
         region = "",
         encode = "",
         field_delimiter = "",
         row_delimiter = "",
         obs_dir = "",
         file_prefix = "",
         rolling_size = "",
         rolling_interval = "",
         quote = "",
         array_bracket = "",
         append = "",
```

```
        max_record_num_per_file = "",
        dump_interval = "",
        dis_notice_channel = ""
)
```

Keyword

Table 2-17 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **obs** indicates that data is exported to OBS.
region	Yes	Region to which OBS belongs.
ak	No	Access Key ID (AK).
sk	No	Secret access key used together with the ID of the access key.
encode	Yes	Encoding format. Currently, formats CSV, JSON, ORC, Avro, Avro-Merge, and Parquet are supported.
field_delimiter	No	Separator used to separate every two attributes. This parameter is mandatory only when the CSV encoding format is adopted. If this parameter is not specified, the default separator comma (,) is used.
row_delimiter	No	Row delimiter. This parameter does not need to be configured if the CSV or JSON encoding format is adopted.
json_config	No	If **encode** is set to **json**, you can set this parameter to specify the mapping between the JSON field and the stream definition field. An example of the format is as follows: field1=data_json.field1;field2=data_json.field2.
obs_dir	Yes	Directory for storing files. The directory is in the format of {Bucket name}/{Directory name}, for example, obs-a1/dir1/subdir. If **encode** is set to **csv** (**append** is **false**), **json** (**append** is **false**), **avro_merge**, or **parquet**, parameterization is supported.
file_prefix	No	Prefix of the data export file name. The generated file is named in the format of file_prefix.*x*, for example, file_prefix.1 and file_prefix.2. If this parameter is not specified, the file prefix is **temp** by default.

Parameter	Mandatory	Description
rolling_size	No	Maximum size of a file. **NOTE** • One or both of **rolling_size** and **rolling_interval** must be configured. • When the size of a file exceeds the specified size, a new file is generated. • The unit can be KB, MB, or GB. If no unit is specified, the byte unit is used. • This parameter does not need to be configured if the ORC encoding format is adopted.
rolling_interval	No	Time mode, in which data is saved to the corresponding directory. **NOTE** • One or both of **rolling_size** and **rolling_interval** must be configured. • After this parameter is specified, data is written to the corresponding directories according to the output time. • The parameter value can be in the format of yyyy/MM/dd/HH/mm, which is case sensitive. The minimum unit is minute. If this parameter is set to **yyyy/MM/dd/HH**, data is written to the directory that is generated at the hour time. For example, data generated at 2018-09-10 16:00 will be written to the **{obs_dir}/2018-09-10_16** directory. • If both **rolling_size** and **rolling_interval** are set, a new file is generated when the size of a single file exceeds the specified size in the directory corresponding to each time point.
quote	No	Modifier, which is added before and after each attribute only when the CSV encoding format is adopted. You are advised to use invisible characters, such as **u0007**, as the parameter value.
array_bracket	No	Array bracket, which can be configured only when the CSV encoding format is adopted. The available options are **()**, **{}**, and **[]**. For example, if you set this parameter to **{}**, the array output format is {a1, a2}.
append	No	The value can be **true** or **false**. The default value is **true**. If OBS does not support the append mode and the encoding format is CSV or JSON, set this parameter to **false**. If **Append** is set to **false**, **max_record_num_per_file** and **dump_interval** must be set.

Parameter	Mandatory	Description
max_record_num_per_file	No	Maximum number of records in a file. This parameter needs to be set if **encode** is **csv** (**append** is **false**), **json** (**append** is **false**), **orc**, **avro**, **avro_merge**, or **parquet**. If the maximum number of records has been reached, a new file is generated.
dump_interval	No	Triggering period. This parameter needs to be configured when the ORC encoding format is adopted or notification to DIS is enabled. ● If the ORC encoding format is specified, this parameter indicates that files will be uploaded to OBS when the triggering period arrives even if the number of file records does not reach the maximum value. ● In notification to DIS is enabled, this parameter specifies that a notification is sent to DIS every period to indicate that no more files will be generated in the directory.
dis_notice_channel	No	DIS channel where DLI sends the record that contains the OBS directory DLI periodically sends the DIS channel a record, which contains the OBS directory, indicating that no more new files will be generated in the directory.
encoded_data	No	Data to be encoded. This parameter is set if **encode** is **json** (**append** is **false**), **avro_merge**, or **parquet**. The format is **${field_name}**, indicating that the stream field content is encoded as a complete record.

Precautions

If a configuration item can be specified through parameter configurations, one or more columns in the record can be used as part of the configuration item. For example, if the configuration item is set to **car_$ {car_brand}** and the value of **car_brand** in a record is **BMW**, the value of this configuration item is **car_BMW** in the record.

Example

- Export the **car_infos** data to the **obs-sink** bucket in OBS. The output directory is **car_infos**. The output file uses **greater_30** as the file name prefix. The maximum size of a single file is 100 MB. If the data size exceeds 100 MB, another new file is generated. The data is encoded in CSV format, the comma (,) is used as the attribute delimiter, and the line break is used as the line separator.
```
CREATE SINK STREAM car_infos (
  car_id STRING,
  car_owner STRING,
```

```
  car_brand STRING,
  car_price INT,
  car_timestamp LONG
)
  WITH (
    type = "obs",
    encode = "csv",
    region = "xxx",
    field_delimiter = ",",
    row_delimiter = "\n",
    obs_dir = "obs-sink/car_infos",
    file_prefix = "greater_30",
    rolling_size = "100m"
);
```

- Example of the ORC encoding format

```
CREATE SINK STREAM car_infos (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT,
  car_timestamp LONG
)
  WITH (
    type = "obs",
    region = "xxx",
    encode = "orc",
    obs_dir = "dli-append-2/obsorc",
    FILE_PREFIX = "es_info",
    max_record_num_per_file = "100000",
    dump_interval = "60"
);
```

- For details about the parquet encoding example, see the example in **File System Sink Stream (Recommended)**.

2.4.14 RDS Sink Stream

Function

DLI outputs the Flink job output data to RDS. Currently, PostgreSQL and MySQL databases are supported. The PostgreSQL database can store data of more complex types and delivers space information services, multi-version concurrent control (MVCC), and high concurrency. It applies to location applications, financial insurance, and e-commerce. The MySQL database reduces IT deployment and maintenance costs in various scenarios, such as web applications, e-commerce, enterprise applications, and mobile applications.

RDS is a cloud-based web service. RDS includes databases of the following types: MySQL, PostgreSQL, and Microsoft SQL Server.

Prerequisites

- Ensure that you have created a PostgreSQL or MySQL RDS instance in RDS.
- In this scenario, jobs must run on the dedicated queue of DLI. Therefore, DLI must interconnect with the enhanced datasource connection that has been connected with RDS instance. You can also set the security group rules as required.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
```

```
  type = "rds",
  username = "",
  password = "",
  db_url = "",
  table_name = ""
);
```

Keyword

Table 2-18 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **rds** indicates that data is exported to RDS.
username	Yes	Username for connecting to a database.
password	Yes	Password for connecting to a database.
db_url	Yes	Database connection address, for example, **{database_type}://ip:port/database**. Currently, two types of database connections are supported: MySQL and PostgreSQL. ● MySQL: 'mysql://ip:port/database' ● PostgreSQL: 'postgresql://ip:port/database'
table_name	Yes	Name of the table where data will be inserted.

db_columns	No	Mapping between attributes in the output stream and those in the database table. This parameter must be configured based on the sequence of attributes in the output stream.
		Example:
		`create sink stream a3(student_name  string, student_age int) with (`
		`type = "rds",`
		`username = "root",`
		`password = "xxxxxxxx",`
		`db_url = "mysql://192.168.0.102:8635/test1",`
		`db_columns = "name,age",`
		`table_name = "t1"`
		`);`
		In the example, **student_name** corresponds to the name attribute in the database, and **student_age** corresponds to the age attribute in the database.
		NOTE
		● If **db_columns** is not configured, it is normal that the number of attributes in the output stream is less than that of attributes in the database table and the extra attributes in the database table are all nullable or have default values.

Parameter	Mandatory	Description

primary_key	No	To update data in the table in real time by using the primary key, add the **primary_key** configuration item (**c_timeminute** in the following example) when creating a table. During the data writing operation, data is updated if the specified **primary_key** exists. Otherwise, data is inserted.
		Example:
		`CREATE SINK STREAM test(c_timeminute LONG, c_cnt LONG)`
		`WITH (`
		`type = "rds",`
		`username = "root",`
		`password = "xxxxxxxx",`
		`db_url = "mysql://192.168.0.12:8635/test",`
		`table_name = "test",`
		`primary_key = "c_timeminute");`

operation_field	No	Processing method of specified data in the format of ${field_name}. The value of **field_name** must be a string. If **field_name** indicates D or DELETE, this record is deleted from the database and data is inserted by default.

Precautions

The stream format defined by **stream_id** must be the same as the table format.

Example

Data of stream **audi_cheaper_than_30w** is exported to the **audi_cheaper_than_30w** table in the **test** database.

```
CREATE SINK STREAM audi_cheaper_than_30w (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
    type = "rds",
    username = "root",
    password = "xxxxxx",
    db_url = "mysql://192.168.1.1:8635/test",
    table_name = "audi_cheaper_than_30w"
);
```

2.4.15 SMN Sink Stream

Function

DLI exports Flink job output data to SMN.

SMN provides reliable and flexible large-scale message notification services to DLI. It significantly simplifies system coupling and pushes messages to subscription endpoints based on requirements. SMN can be connected to other cloud services or integrated with any application that uses or generates message notifications to push messages over multiple protocols.

For more information about SMN, see the .

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH(
    type = "smn",
    region = "",
    topic_urn = "",
    urn_column = "",
    message_subject = "",
    message_column = ""
  )
```

Keyword

Table 2-19 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. **smn** indicates that data is exported to SMN.
region	Yes	Region to which SMN belongs.
topic_urn	No	URN of an SMN topic, which is used for the static topic URN configuration. The SMN topic serves as the destination for short message notification and needs to be created in SMN. One of **topic_urn** and **urn_column** must be configured. If both of them are configured, the **topic_urn** setting takes precedence.
urn_column	No	Field name of the topic URN content, which is used for the dynamic topic URN configuration. One of **topic_urn** and **urn_column** must be configured. If both of them are configured, the **topic_urn** setting takes precedence.
message_subject	Yes	Message subject sent to SMN. This parameter can be user-defined.
message_column	Yes	Field name in the sink stream. Contents of the field name serve as the message contents, which are user-defined. Currently, only text messages (default) are supported.

Precautions

None

Example

Data of stream **over_speed_warning** is exported to SMN.

```
//Static topic configuration
CREATE SINK STREAM over_speed_warning (
 over_speed_message STRING /* over speed message */
)
 WITH (
  type = "smn",
  region = "xxx",
  topic_Urn = "xxx",
  message_subject = "message title",
  message_column = "over_speed_message"
 );
//Dynamic topic configuration
CREATE SINK STREAM over_speed_warning2 (
  over_speed_message STRING, /* over speed message */
  over_speed_urn STRING
)
 WITH (
  type = "smn",
  region = "xxx",
  urn_column = "over_speed_urn",
  message_subject = "message title",
  message_column = "over_speed_message"
 );
```

2.5 Creating a Temporary Stream

Function

The temporary stream is used to simplify SQL logic. If complex SQL logic is followed, write SQL statements concatenated with temporary streams. The temporary stream is just a logical concept and does not generate any data.

Syntax

```
CREATE TEMP STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
```

Example

```
create temp stream a2(attr1 int, attr2 string);
```

2.6 Creating a Dimension Table

2.6.1 Creating a Redis Table

Create a Redis table to connect to the source stream.

For details about the JOIN syntax, see **JOIN Between Stream Data and Table Data**.

Syntax

```
CREATE TABLE table_id (key_attr_name STRING(, hash_key_attr_name STRING)?, value_attr_name STRING)
 WITH (
  type = "dcs_redis",
```



```
  cluster_address = ""(,password = "")?,
  value_type= "",
  key_column= ""(,hash_key_column="")?);
```

Keyword

Table 2-20 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. Value **dcs_redis** indicates that data is exported to DCS Redis.
cluster_address	Yes	Redis instance connection address.
password	No	Redis instance connection password. This parameter is not required if password-free access is used.
value_type	Yes	Indicates the field data type. Supported data types include string, list, hash, set, and zset.
key_column	Yes	Indicates the column name of the Redis key attribute.
hash_key_column	No	If **value_type** is set to **hash**, this field must be specified as the column name of the level-2 key attribute.
cache_max_num	No	Indicates the maximum number of cached query results. The default value is **32768**.
cache_time	No	Indicates the maximum duration for caching database query results in the memory. The unit is millisecond. The default value is **10000**. The value **0** indicates that caching is disabled.

Precautions

- Redis clusters are not supported.
- Ensure that You have created a Redis cache instance on DCS using your account.
- In this scenario, jobs must run on the dedicated queue of DLI. Therefore, DLI must interconnect with the enhanced datasource connection that has been connected with DCS instance. You can also set the security group rules as required.

Example

The Redis table is used to connect to the source stream.

```
CREATE TABLE table_a (attr1 string, attr2 string, attr3 string)
  WITH (
    type = "dcs_redis",
    value_type = "hash",
    key_column = "attr1",
    hash_key_column = "attr2",
```

```
  cluster_address = "192.168.1.238:6379",
  password = "xxxxxxxx"
);
```

2.6.2 Creating an RDS Table

Create an RDS/DWS table to connect to the source stream.

For details about the JOIN syntax, see **JOIN**.

Prerequisites

- Ensure that you have created a PostgreSQL or MySQL RDS instance in RDS.
- In this scenario, jobs must run on the dedicated queue of DLI. Therefore, DLI must interconnect with the enhanced datasource connection that has been connected with RDS instance. You can also set the security group rules as required.

Syntax

```
CREATE TABLE  table_id (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
    type = "rds",
    username = "",
    password = "",
    db_url = "",
    table_name = ""
  );
```

Keyword

Table 2-21 Keyword description

Parameter	Mandatory	Description
type	Yes	Output channel type. Value **rds** indicates that data is stored to RDS.
username	Yes	Username for connecting to a database.
password	Yes	Password for connecting to a database.

Parameter	Mandatory	Description
db_url	Yes	Database connection address, for example, **{database_type}://ip:port/database**. Currently, two types of database connections are supported: MySQL and PostgreSQL. ● MySQL: 'mysql://ip:port/database' ● PostgreSQL: 'postgresql://ip:port/database' **NOTE** To create a DWS dimension table, set the database connection address to a DWS database address. If the DWS database version is later than 8.1.0, the open-source PostgreSQL driver cannot be used for connection. You need to use the GaussDB driver for connection.
table_name	Yes	Indicates the name of the database table for data query.
db_columns	No	Indicates the mapping of stream attribute fields between the sink stream and database table. This parameter is mandatory when the stream attribute fields in the sink stream do not match those in the database table. The parameter value is in the format of dbtable_attr1,dbtable_attr2,dbtable_attr3.
cache_max_num	No	Indicates the maximum number of cached query results. The default value is **32768**.
cache_time	No	Indicates the maximum duration for caching database query results in the memory. The unit is millisecond. The default value is **10000**. The value **0** indicates that caching is disabled.

Example

The RDS table is used to connect to the source stream.

```
CREATE SOURCE STREAM car_infos (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
    type = "dis",
    region = "",
    channel = "dliinput",
    encode = "csv",
    field_delimiter = ","
  );

CREATE TABLE  db_info (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
```

```
  WITH (
    type = "rds",
    username = "root",
    password = "******",
    db_url = "postgresql://192.168.0.0:2000/test1",
    table_name = "car"
);

CREATE SINK STREAM audi_cheaper_than_30w (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
    type = "dis",
    region = "",
    channel = "dlioutput",
    partition_key = "car_owner",
    encode = "csv",
    field_delimiter = ","
  );

INSERT INTO audi_cheaper_than_30w
SELECT a.car_id, b.car_owner, b.car_brand, b.car_price
FROM car_infos as a join db_info as b on a.car_id = b.car_id;
```

NOTE

To create a DWS dimension table, set the database connection address to a DWS database address. If the DWS database version is later than 8.1.0, the open-source PostgreSQL driver cannot be used for connection. You need to use the GaussDB driver for connection.

2.7 Custom Stream Ecosystem

2.7.1 Custom Source Stream

Compile code to obtain data from the desired cloud ecosystem or open-source ecosystem as the input data of Flink jobs.

Syntax

```
CREATE SOURCE STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "user_defined",
    type_class_name = "",
    type_class_parameter = ""
  )
  (TIMESTAMP BY timeindicator (',' timeindicator)?);timeindicator:PROCTIME '.' PROCTIME| ID '.' ROWTIME
```

Keyword

Table 2-22 Keyword description

Parameter	Mandatory	Description
type	Yes	Data source type. The value **user_defined** indicates that the data source is a user-defined data source.
type_class_name	Yes	Name of the source class for obtaining source data. The value must contain the complete package path.
type_class_parameter	Yes	Input parameter of the user-defined source class. Only one parameter of the string type is supported.

Precautions

The user-defined source class needs to inherit the **RichParallelSourceFunction** class and specify the data type as Row. For example, define MySource class: **public class MySource extends RichParallelSourceFunction<Row>{}**. It aims to implement the **open**, **run**, and **close** functions.

Dependency pom:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_2.11</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-core</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
</dependency>
```

Example

A data record is generated in each period. The data record contains only one field of the INT type. The initial value is 1 and the period is 60 seconds. The period is specified by an input parameter.

```
CREATE SOURCE STREAM user_in_data (
    count INT
     )
  WITH (
    type = "user_defined",
    type_class_name = "mySourceSink.MySource",
    type_class_parameter = "60"
      )
      TIMESTAMP BY car_timestamp.rowtime;
```

NOTE

To customize the implementation of the source class, you need to pack the class in a JAR package and upload the UDF function on the SQL editing page.

2.7.2 Custom Sink Stream

Compile code to write the data processed by DLI to a specified cloud ecosystem or open-source ecosystem.

Syntax

```
CREATE SINK STREAM stream_id (attr_name attr_type (',' attr_name attr_type)* )
  WITH (
    type = "user_defined",
    type_class_name = "",
    type_class_parameter = ""
  );
```

Keyword

Table 2-23 Keyword description

Parameter	Mandatory	Description
type	Yes	Data source type. The value **user_defined** indicates that the data source is a user-defined data source.
type_class_name	Yes	Name of the sink class for obtaining source data. The value must contain the complete package path.
type_class_parameter	Yes	Input parameter of the user-defined sink class. Only one parameter of the string type is supported.

Precautions

The user-defined sink class needs to inherit the **RichSinkFunction** class and specify the data type as Row. For example, define MySink class: **public class MySink extends RichSinkFunction<Row>{}**. It aims to implement the **open**, **invoke**, and **close** functions.

Dependency pom:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_2.11</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-core</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
</dependency>
```

Example

Writing data encoded in CSV format to a DIS stream is used as an example.

```
CREATE SINK STREAM user_out_data (
    count INT
```

```
)
 WITH (
  type = "user_defined",
  type_class_name = "mySourceSink.MySink",
  type_class_parameter = ""
    );
```

NOTE

To customize the implementation of the sink class, you need to pack the class in a JAR package and upload the UDF function on the SQL editing page.

2.8 Data Type

Overview

Data type is a basic attribute of data and used to distinguish different types of data. Different data types occupy different storage space and support different operations. Data is stored in data tables in the database. Each column of a data table defines the data type. During storage, data must be stored according to data types.

Similar to the open source community, Flink SQL of the big data platform supports both native data types and complex data types.

Primitive Data Types

Table 2-24 lists native data types supported by Flink SQL.

Table 2-24 Primitive Data Types

Data Type	Description	Storage Space	Value Range
VARCHAR	Character with a variable length	-	-
BOOLEAN	Boolean	-	TRUE/FALSE
TINYINT	Signed integer	1 byte	-128-127
SMALLINT	Signed integer	2 bytes	-32768-32767
INT	Signed integer	4 bytes	–2147483648 to 2147483647
INTEGER	Signed integer	4 bytes	–2147483648 to 2147483647
BIGINT	Signed integer	8 bytes	–9223372036854775808 to 9223372036854775807
REAL	Single-precision floating point	4 bytes	-

Data Type	Description	Storage Space	Value Range
FLOAT	Single-precision floating point	4 bytes	-
DOUBLE	Double-precision floating-point	8 bytes	-
DECIMAL	Data type of valid fixed places and decimal places	-	-
DATE	Date type in the format of yyyy-MM-dd, for example, 2014-05-29	-	**DATE** does not contain time information. Its value ranges from 0000-01-01 to 9999-12-31.
TIME	Time type in the format of HH:MM:SS For example, 20:17:40	-	-
TIMESTAMP(3)	Timestamp of date and time For example, 1969-07-20 20:17:40	-	-
INTERVAL timeUnit [TO timeUnit]	Time interval For example, INTERVAL '1:5' YEAR TO MONTH, INTERVAL '45' DAY	-	-

Complex Data Types

Flink SQL supports complex data types and complex type nesting. **Table 2-25** describes complex data types.

Table 2-25 Complex Data Types

Data Type	Description	Declaration Method	Reference Method	Construction Method
ARRAY	Indicates a group of ordered fields that are of the same data type.	ARRAY[TYPE]	Variable name **[subscript]**. The subscript starts from 1, for example, **v1[1]**.	Array[value1, value2, ...] as v1

Data Type	Description	Declaration Method	Reference Method	Construction Method
MAP	Indicates a group of unordered key/value pairs. The key must be native data type, but the value can be either native data type or complex data type. The type of the same MAP key, as well as the MAP value, must be the same.	**MAP [TYPE, TYPE]**	Variable name **[key]**, for example, **v1[key]**	Map[key, value, key2, value2, key3, value3.......] as v1
ROW	Indicates a group of named fields. The data types of the fields can be different.	ROW<a1 TYPE1, a2 TYPE2>	Variable name. Field name, for example, **v1.a1**.	Row('1',2) as v1

Here is a sample code:

```
CREATE SOURCE STREAM car_infos (
  car_id STRING,
  address ROW<city STRING, province STRING, country STRING>,
  average_speed MAP[STRING, LONG],
  speeds ARRAY[LONG]
)
  WITH (
    type = "dis",
    region = "xxx",
    channel = "dliinput",
    encode = "json"
);

CREATE temp STREAM car_speed_infos (
  car_id STRING,
  province STRING,
  average_speed LONG,
  start_speed LONG
);

INSERT INTO car_speed_infos SELECT
   car_id,
   address.province,
   average_speed[address.city],
   speeds[1]
FROM car_infos;
```

Complex Type Nesting

- JSON format enhancement

  The following uses Source as an example. The method of using Sink is the same.

– **json_schema** can be configured.

After **json_schema** is configured, fields in DDL can be automatically generated from **json_schema** without declaration. Here is a sample code:

```
CREATE SOURCE STREAM data_with_schema WITH (
       type = "dis",
       region = "xxx",
       channel = "dis-in",
       encode = "json",
       json_schema = '{"definitions":{"address":{"type":"object","properties":{"street_address":
{"type":"string"},"city":{"type":"string"},"state":{"type":"string"}},"required":
["street_address","city","state"]}},"type":"object","properties":{"billing_address":{"$ref":"#/
definitions/address"},"shipping_address":{"$ref":"#/definitions/address"},"optional_address":
{"oneOf":[{"type":"null"},{"$ref":"#/definitions/address"}]}}}'
     );

     CREATE SINK STREAM buy_infos (
       billing_address_city STRING,
       shipping_address_state string
     ) WITH (
       type = "obs",
       encode = "csv",
       region = "xxx" ,
       field_delimiter = ",",
       row_delimiter = "\n",
       obs_dir = "bucket/car_infos",
       file_prefix = "over",
       rolling_size = "100m"
     );

     insert into buy_infos select billing_address.city, shipping_address.state from
data_with_schema;
```

Example data

```
{
 "billing_address":
  {
   "street_address":"xxx",
   "city":"xxx",
   "state":"xxx"
   },
 "shipping_address":
  {
   "street_address":"xxx",
   "city":"xxx",
   "state":"xxx"
  }
}
```

– The **json_schema** and **json_config** parameters can be left empty. For details about how to use **json_config**, see the example in **Open-Source Kafka Source Stream**.

In this case, the attribute name in the DDL is used as the JSON key for parsing by default.

The following is example data. It contains nested JSON fields, such as **billing_address** and **shipping_address**, and non-nested fields **id** and **type2**.

```
{
 "id":"1",
 "type2":"online",
 "billing_address":
  {
   "street_address":"xxx",
   "city":"xxx",
   "state":"xxx"
   },
```

```
 "shipping_address":
  {
   "street_address":"xxx",
   "city":"xxx",
   "state":"xxx"
  }
}
```

The table creation and usage examples are as follows:

```
CREATE SOURCE STREAM car_info_data (
       id STRING,
       type2 STRING,
       billing_address Row<street_address string, city string, state string>,
       shipping_address Row<street_address string, city string, state string>,
       optional_address Row<street_address string, city string, state string>
     ) WITH (
       type = "dis",
       region = "xxx",
       channel = "dis-in",
       encode = "json"
     );

    CREATE SINK STREAM buy_infos (
       id STRING,
       type2 STRING,
       billing_address_city STRING,
       shipping_address_state string
     ) WITH (
       type = "obs",
       encode = "csv",
       region = "xxx",
       field_delimiter = ",",
       row_delimiter = "\n",
       obs_dir = "bucket/car_infos",
       file_prefix = "over",
       rolling_size = "100m"
     );

     insert into buy_infos select id, type2, billing_address.city, shipping_address.state from
car_info_data;
```

- Complex data types supported by sink serialization
  - Currently, only the CSV and JSON formats support complex data types.
  - For details about the JSON format, see **Json format enhancement**.
  - There is no standard format for CSV files. Therefore, only sink parsing is supported.
  - Output format: It is recommended that the output format be the same as that of the native Flink.

    Map: {key1=Value1, key2=Value2}

    Row: Attributes are separated by commas (,), for example, **Row(1,'2') => 1,'2'**.

2.9 Built-In Functions

2.9.1 Mathematical Operation Functions

Relational Operators

All data types can be compared by using relational operators and the result is returned as a BOOLEAN value.

Relationship operators are binary operators. Two compared data types must be of the same type or they must support implicit conversion.

Table 2-26 lists all relational operators supported by Flink SQL.

Table 2-26 Relational Operators

Operator	Returned Data Type	Description
A = B	BOOLEAN	If A is equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. This operator is used for value assignment.
A <> B	BOOLEAN	If A is not equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned. This operator follows the standard SQL syntax.
A < B	BOOLEAN	If A is less than B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A <= B	BOOLEAN	If A is less than or equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A > B	BOOLEAN	If A is greater than B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A >= B	BOOLEAN	If A is greater than or equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A IS NULL	BOOLEAN	If A is **NULL**, then **TRUE** is returned. Otherwise, **FALSE** is returned.
A IS NOT NULL	BOOLEAN	If A is not **NULL**, then **TRUE** is returned. Otherwise, **FALSE** is returned.
A IS DISTINCT FROM B	BOOLEAN	If A is not equal to B, **TRUE** is returned. **NULL** indicates A equals B.
A IS NOT DISTINCT FROM B	BOOLEAN	If A is equal to B, **TRUE** is returned. **NULL** indicates A equals B.

Operator	Returned Data Type	Description
A BETWEEN [ASYMMETRIC | SYMMETRIC] B AND C	BOOLEAN	If A is greater than or equal to B but less than or equal to C, **TRUE** is returned. ● ASYMMETRIC: indicates that B and C are location-related. For example, "A BETWEEN ASYMMETRIC B AND C" is equivalent to "A BETWEEN B AND C". ● SYMMETRIC: indicates that B and C are not location-related. For example, "A BETWEEN SYMMETRIC B AND C" is equivalent to "A BETWEEN B AND C) OR (A BETWEEN C AND B".
A NOT BETWEEN B AND C	BOOLEAN	If A is less than B or greater than C, **TRUE** is returned.
A LIKE B [ESCAPE C]	BOOLEAN	If A matches pattern B, **TRUE** is returned. The escape character C can be defined as required.
A NOT LIKE B [ESCAPE C]	BOOLEAN	If A does not match pattern B, **TRUE** is returned. The escape character C can be defined as required.
A SIMILAR TO B [ESCAPE C]	BOOLEAN	If A matches regular expression B, **TRUE** is returned. The escape character C can be defined as required.
A NOT SIMILAR TO B [ESCAPE C]	BOOLEAN	If A does not match regular expression B, **TRUE** is returned. The escape character C can be defined as required.
value IN (value [, value]*)	BOOLEAN	If the value is equal to any value in the list, **TRUE** is returned.
value NOT IN (value [, value]*)	BOOLEAN	If the value is not equal to any value in the list, **TRUE** is returned.

NOTE

- Values of the double, real, and float types may be different in precision. The equal sign (=) is not recommended for comparing two values of the double type. You are advised to obtain the absolute value by subtracting these two values of the double type and determine whether they are the same based on the absolute value. If the absolute value is small enough, the two values of the double data type are regarded equal. For example:
```
abs(0.9999999999 - 1.0000000000) < 0.000000001 //The precision decimal places of 0.9999999999 and 1.0000000000 are 10, while the precision decimal place of 0.000000001 is 9. Therefore, 0.9999999999 can be regarded equal to 1.0000000000.
```
- Comparison between data of the numeric type and character strings is allowed. During comparison using relational operators, including >, <, ≤, and ≥, data of the string type is converted to numeric type by default. No characters other than numeric characters are allowed.
- Character strings can be compared using relational operators.

Logical Operators

Common logical operators are AND, OR, and NOT. Their priority order is NOT > AND > OR.

Table 2-27 lists the calculation rules. A and B indicate logical expressions.

Table 2-27 Logical Operators

Operator	Result Type	Description
A OR B	BOOLEAN	If A or B is TRUE, **TRUE** is returned. Three-valued logic is supported.
A AND B	BOOLEAN	If both A and B are TRUE, **TRUE** is returned. Three-valued logic is supported.
NOT A	BOOLEAN	If A is not TRUE, **TRUE** is returned. If A is UNKNOWN, **UNKNOWN** is returned.
A IS FALSE	BOOLEAN	If A is TRUE, **TRUE** is returned. If A is UNKNOWN, **FALSE** is returned.
A IS NOT FALSE	BOOLEAN	If A is not FALSE, **TRUE** is returned. If A is UNKNOWN, **TRUE** is returned.
A IS TRUE	BOOLEAN	If A is TRUE, **TRUE** is returned. If A is UNKNOWN, **FALSE** is returned.
A IS NOT TRUE	BOOLEAN	If A is not TRUE, **TRUE** is returned. If A is UNKNOWN, **TRUE** is returned.
A IS UNKNOWN	BOOLEAN	If A is UNKNOWN, **TRUE** is returned.
A IS NOT UNKNOWN	BOOLEAN	If A is not UNKNOWN, **TRUE** is returned.

NOTE

Only data of the Boolean type can be used for calculation using logical operators. Implicit type conversion is not supported.

Arithmetic Operators

Arithmetic operators include binary operators and unary operators, for all of which, the returned results are of the numeric type. Table 2-28 lists arithmetic operators supported by Flink SQL.

Table 2-28 Arithmetic Operators

Operator	Result Type	Description
+ numeric	All numeric types	Returns numbers.
- numeric	All numeric types	Returns negative numbers.
A + B	All numeric types	A plus B. The result type is associated with the operation data type. For example, if floating-point number is added to an integer, the result will be a floating-point number.
A - B	All numeric types	A minus B. The result type is associated with the operation data type.
A * B	All numeric types	Multiply A and B. The result type is associated with the operation data type.
A / B	All numeric types	Divide A by B. The result is a number of the double type (double-precision number).
POWER(A, B)	All numeric types	Returns the value of A raised to the power B.
ABS(numeric)	All numeric types	Returns the absolute value of a specified value.
MOD(A, B)	All numeric types	Returns the remainder (modulus) of A divided by B. A negative value is returned only when A is a negative value.
SQRT(A)	All numeric types	Returns the square root of A.
LN(A)	All numeric types	Returns the nature logarithm of A (base e).

Operator	Result Type	Description
LOG10(A)	All numeric types	Returns the base 10 logarithms of A.
EXP(A)	All numeric types	Return the value of e raised to the power of **a**.
CEIL(A) CEILING(A)	All numeric types	Return the smallest integer that is greater than or equal to **a**. For example: ceil(21.2) = 22.
FLOOR(A)	All numeric types	Return the largest integer that is less than or equal to **a**. For example: floor(21.2) = 21.
SIN(A)	All numeric types	Returns the sine value of A.
COS(A)	All numeric types	Returns the cosine value of A.
TAN(A)	All numeric types	Returns the tangent value of A.
COT(A)	All numeric types	Returns the cotangent value of A.
ASIN(A)	All numeric types	Returns the arc sine value of A.
ACOS(A)	All numeric types	Returns the arc cosine value of A.
ATAN(A)	All numeric types	Returns the arc tangent value of A.
DEGREES(A)	All numeric types	Convert the value of **a** from radians to degrees.
RADIANS(A)	All numeric types	Convert the value of **a** from degrees to radians.
SIGN(A)	All numeric types	Returns the sign of A. **1** is returned if A is positive. **–1** is returned if A is negative. Otherwise, **0** is returned.
ROUND(A, d)	All numeric types	Round A to d places right to the decimal point. d is an int type. For example: round(21.263,2) = 21.26.
PI()	All numeric types	Return the value of **pi**.

NOTE

Data of the string type is not allowed in arithmetic operations.

2.9.2 String Functions

The common character string functions of DLI are as follows:

Table 2-29 String Operators

Operator	Returned Data Type	Description
||	VARCHAR	Concatenates two strings.
CHAR_LENGTH	INT	Returns the number of characters in a string.
CHARACTER_LENGTH	INT	Returns the number of characters in a string.
CONCAT	VARCHAR	Concatenates two or more string values to form a new string. If the value of any parameter is **NULL**, skip this parameter.
CONCAT_WS	VARCHAR	Concatenates each parameter value and the separator specified by the first parameter separator to form a new string. The length and type of the new string depend on the input value.
HASH_CODE	INT	Returns the absolute value of **HASH_CODE()** of a string. In addition to **string**, **int**, **bigint**, **float**, and **double** are also supported.
INITCAP	VARCHAR	Returns a string whose first letter is in uppercase and the other letters in lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.
IS_ALPHA	BOOLEAN	Checks whether a string contains only letters.
IS_DIGITS	BOOLEAN	Checks whether a string contains only digits.
IS_NUMBER	BOOLEAN	Checks whether a string is numeric.
IS_URL	BOOLEAN	Checks whether a string is a valid URL.
JSON_VALUE	VARCHAR	Obtains the value of a specified path in a JSON string.
KEY_VALUE	VARCHAR	Obtains the value of a key in a key-value pair string.

Operator	Returned Data Type	Description
LOWER	VARCHAR	Returns a string of lowercase characters.
LPAD	VARCHAR	Concatenates the pad string to the left of the str string until the length of the new string reaches the specified length len.
MD5	VARCHAR	Returns the MD5 value of a string. If the parameter is an empty string (that is, the parameter is **''**), an empty string is returned.
OVERLAY	VARCHAR	Replaces the substring of **x** with **y**. Replace length +1 characters starting from **start_position**.
POSITION	INT	Returns the position of the first occurrence of the target string **x** in the queried string **y**. If the target string **x** does not exist in the queried string **y**, **0** is returned.
REPLACE	VARCHAR	Replaces all **str2** in the **str1** string with **str3**. ● **str1**: original character. ● **str2**: target character. ● **str3**: replacement character.
RPAD	VARCHAR	Concatenates the pad string to the right of the str string until the length of the new string reaches the specified length len.
SHA1	STRING	Returns the SHA1 value of the **expr** string.
SHA256	STRING	Returns the SHA256 value of the **expr** string.
STRING_TO_ARRAY	ARRAY[STRING]	Separates the **value** string as string arrays by using the delimiter.
SUBSTRING	VARCHAR	Returns the substring starting from a fixed position of A. The start position starts from 1.
TRIM	STRING	Removes A at the start position, or end position, or both the start and end positions from B. By default, string expressions A at both the start and end positions are removed.
UPPER	VARCHAR	Returns a string converted to uppercase characters.

||

- Function

  Concatenates two character strings.

- Syntax
```
VARCHAR VARCHAR a || VARCHAR b
```
- Parameter description
  - **a**: character string.
  - **b**: character string.
- Example
  - Test statement
```
SELECT "hello" || "world";
```
  - Test result
```
"helloworld"
```

CHAR_LENGTH

- Function

  Returns the number of characters in a string.
- Syntax
```
INT CHAR_LENGTH(a)
```
- Parameter description
  - **a**: character string.
- Example
  - Test statement
```
SELECT  CHAR_LENGTH(var1) as aa FROM T1;
```
  - Test data and result

Table 2-30 Test data and result

Test Data (var1)	Test Result (aa)
abcde123	8

CHARACTER_LENGTH

- Function

  Returns the number of characters in a string.
- Syntax
```
INT CHARACTER_LENGTH(a)
```
- Parameter description
  - **a**: character string.
- Example
  - Test statement
```
SELECT  CHARACTER_LENGTH(var1) as aa FROM T1;
```
  - Test data and result

Table 2-31 Test data and result

Test Data (var1)	Test Result (aa)
abcde123	8

CONCAT

- Function

  Concatenates two or more string values to form a new string. If the value of any parameter is NULL, skip this parameter.
- Syntax
```
VARCHAR CONCAT(VARCHAR var1, VARCHAR var2, ...)
```
- Parameter description
  - **var1**: character string
  - **var2**: character string
- Example
  - Test statement
```
SELECT CONCAT("abc", "def", "ghi", "jkl");
```
  - Test result
```
"abcdefghijkl"
```

CONCAT_WS

- Function

  Concatenates each parameter value and the separator specified by the first parameter separator to form a new string. The length and type of the new string depend on the input value.

  NOTE

  If the value of **separator** is **null**, **separator** is combined with an empty string. If other parameters are set to null, the parameters whose values are null are skipped during combination.
- Syntax
```
VARCHAR CONCAT_WS(VARCHAR separator, VARCHAR var1, VARCHAR var2, ...)
```
- Parameter description
  - **separator**: separator.
  - **var1**: character string
  - **var2**: character string
- Example
  - Test statement
```
SELECT CONCAT_WS("-", "abc", "def", "ghi", "jkl");
```
  - Test result
```
"abc-def-ghi-jkl"
```

HASH_CODE

- Function

Returns the absolute value of **HASH_CODE()** of a string. In addition to **string**, **int**, **bigint**, **float**, and **double** are also supported.

- Syntax
```
INT HASH_CODE(VARCHAR str)
```
- Parameter description
  – **str**: character string.
- Example
  – Test statement
```
SELECT HASH_CODE("abc");
```
  – Test result
```
96354
```

INITCAP

- Function

  Return the string whose first letter is in uppercase and the other letters in lowercase. Strings are sequences of alphanumeric characters separated by non-alphanumeric characters.
- Syntax
```
VARCHAR INITCAP(a)
```
- Parameter description
  – **a**: character string.
- Example
  – Test statement
```
SELECT INITCAP(var1)as aa FROM T1;
```
  – Test data and result

Table 2-32 Test data and result

Test Data (var1)	Test Result (aa)
aBCde	Abcde

IS_ALPHA

- Function

  Checks whether a character string contains only letters.
- Syntax
```
BOOLEAN IS_ALPHA(VARCHAR content)
```
- Parameter description
  – **content**: Enter a character string.
- Example
  – Test statement
```
SELECT IS_ALPHA(content)  AS case_result FROM T1;
```
  – Test data and results

Table 2-33 Test data and results

Test Data (content)	Test Result (case_result)
Abc	true
abc1#$	false
null	false
Empty string	false

IS_DIGITS

- Function
  Checks whether a character string contains only digits.
- Syntax
  ```
  BOOLEAN IS_DIGITS(VARCHAR content)
  ```
- Parameter description
  - **content**: Enter a character string.
- Example
  - Test statement
    ```
    SELECT IS_DIGITS(content) AS case_result FROM T1;
    ```
  - Test data and results

Table 2-34 Test data and results

Test Data (content)	Test Result (case_result)
78	true
78.0	false
78a	false
null	false
Empty string	false

IS_NUMBER

- Function
  This function is used to check whether a character string is a numeric string.
- Syntax
  ```
  BOOLEAN IS_NUMBER(VARCHAR content)
  ```
- Parameter description
  - **content**: Enter a character string.
- Example
  - Test statement
    ```
    SELECT IS_NUMBER(content) AS case_result FROM T1;
    ```

– Test data and results

Table 2-35 Test data and results

Test Data (content)	Test Result (case_result)
78	true
78.0	true
78a	false
null	false
Empty string	false

IS_URL

- Function
  This function is used to check whether a character string is a valid URL.
- Syntax
```
BOOLEAN IS_URL(VARCHAR content)
```
- Parameter description
  – **content**: Enter a character string.
- Example
  – Test statement
```
SELECT IS_URL(content) AS case_result FROM T1;
```
  – Test data and results

Table 2-36 Test data and results

Test Data (content)	Test Result (case_result)
https://www.testweb.com	true
https://www.testweb.com:443	true
www.testweb.com:443	false
null	false
Empty string	false

JSON_VALUE

- Function
  Obtains the value of a specified path in a JSON character string.
- Syntax
```
VARCHAR JSON_VALUE(VARCHAR content, VARCHAR path)
```
- Parameter description

– **content**: Enter a character string.
– **path**: path to be obtained.

- Example
  – Test statement
```
SELECT JSON_VALUE(content, path) AS case_result FROM T1;
```
  – Test data and results

Table 2-37 Test data and results

Test Data (content and path)		Test Result (case_result)
{ "a1":"v1","a2":7,"a3": 8.0,"a4": {"a41":"v41","a42": ["v1","v2"]}}	$	{ "a1":"v1","a2":7,"a3":8.0,"a4": {"a41":"v41","a42": ["v1","v2"]}}
{ "a1":"v1","a2":7,"a3": 8.0,"a4": {"a41":"v41","a42": ["v1","v2"]}}	$.a1	v1
{ "a1":"v1","a2":7,"a3": 8.0,"a4": {"a41":"v41","a42": ["v1","v2"]}}	$.a4	{"a41":"v41","a42": ["v1","v2"]}
{ "a1":"v1","a2":7,"a3": 8.0,"a4": {"a41":"v41","a42": ["v1","v2"]}}	$.a4.a42	["v1","v2"]
{ "a1":"v1","a2":7,"a3": 8.0,"a4": {"a41":"v41","a42": ["v1","v2"]}}	$.a4.a42[0]	v1

KEY_VALUE

- Function

  This function is used to obtain the value of a key in a key-value pair string.
- Syntax
```
VARCHAR KEY_VALUE(VARCHAR content, VARCHAR split1, VARCHAR split2, VARCHAR key_name)
```
- Parameter description
  – **content**: Enter a character string.
  – **split1**: separator of multiple key-value pairs.
  – **split2**: separator between the key and value.
  – **key_name**: name of the key to be obtained.
- Example
  – Test statement
```
SELECT KEY_VALUE(content, split1, split2, key_name)  AS case_result FROM T1;
```
  – Test data and results

Table 2-38 Test data and results

Test Data (content, split1, split2, and key_name)				Test Result (case_result)
k1=v1;k2=v2	;	=	k1	v1
null	;	=	k1	null
k1=v1;k2=v2	null	=	k1	null

LOWER

- Function
  Returns a string of lowercase characters.
- Syntax
```
VARCHAR LOWER(A)
```
- Parameter description
  – **A**: character string.
- Example
  – Test statement
```
SELECT LOWER(var1) AS aa FROM T1;
```
  – Test data and result

Table 2-39 Test data and result

Test Data (var1)	Test Result (aa)
ABc	abc

LPAD

- Function
  Concatenates the pad string to the left of the str string until the length of the new string reaches the specified length len.
- Syntax
```
VARCHAR LPAD(VARCHAR str, INT len, VARCHAR pad)
```
- Parameter description
  – **str**: character string before concatenation.
  – **len**: length of the concatenated character string.
  – **pad**: character string to be concatenated.

NOTE

- If any parameter is null, **null** is returned.
- If the value of len is a negative number, value **null** is returned.
- If the value of **len** is less than the length of **str**, the first chunk of **str** characters in **len** length is returned.

- Example
  - Test statement
```
SELECT
  LPAD("adc", 2, "hello"),
  LPAD("adc", -1, "hello"),
  LPAD("adc", 10, "hello");
```
  - Test result
```
"ad",,"helloheadc"
```

MD5

- Function

  Returns the MD5 value of a string. If the parameter is an empty string (that is, the parameter is **''**), an empty string is returned.
- Syntax
```
VARCHAR MD5(VARCHAR str)
```
- Parameter description
  - **str**: character string
- Example
  - Test statement
```
SELECT MD5("abc");
```
  - Test result
```
"900150983cd24fb0d6963f7d28e17f72"
```

OVERLAY

- Function

  Replaces the substring of **x** with **y**. Replaces length+1 characters starting from **start_position**.
- Syntax
```
VARCHAR OVERLAY ( (VARCHAR x PLACING VARCHAR y FROM INT start_position [ FOR INT
length ]) )
```
- Parameter description
  - **x**: character string
  - **y**: character string.
  - **start_position**: start position.
  - **length (optional)**: indicates the character length.
- Example
  - Test statement
```
OVERLAY('abcdefg' PLACING 'xyz' FROM 2 FOR 2) AS result FROM T1;
```
  - Test result

Table 2-40 Test result

result
axyzdefg

POSITION

- Function

  Returns the position of the first occurrence of the target string **x** in the queried string **y**. If the target character string **x** does not exist in the queried character string **y**, **0** is returned.

- Syntax
```
INTEGER POSITION(x IN y)
```
- Parameter description
  - **x**: character string
  - **y**: character string.
- Example
  - Test statement
```
POSITION('in' IN 'chin') AS result FROM T1;
```
  - Test result

Table 2-41 Test result

result
3

REPLACE

- Function

  The character string replacement function is used to replace all **str2** in the **str1** string with **str3**.

- Syntax
```
VARCHAR REPLACE(VARCHAR str1, VARCHAR str2, VARCHAR str3)
```
- Parameter description
  - **str1**: original character.
  - **str2**: target character.
  - **str3**: replacement character.
- Example
  - Test statement
```
SELECT
  replace(
    "hello world hello world hello world",
    "world",
    "hello"
  );
```
  - Test result
```
"hello hello hello hello hello hello"
```

RPAD

- Function

  Concatenates the pad string to the right of the str string until the length of the new string reaches the specified length len.

  - If any parameter is null, **null** is returned.
  - If the value of len is a negative number, value **null** is returned.
  - The value of **pad** is an empty string. If the value of **len** is less than the length of **str**, the string whose length is the same as the length of **str** is returned.

- Syntax
```
VARCHAR RPAD(VARCHAR str, INT len, VARCHAR pad)
```
- Parameter description
  - **str**: start character string.
  - **len**: indicates the length of the new character string.
  - **pad**: character string that needs to be added repeatedly.
- Example
  - Test statement
```
SELECT
  RPAD("adc", 2, "hello"),
  RPAD("adc", -1, "hello"),
  RPAD("adc", 10, "hello");
```
  - Test result
```
"ad",,"adchellohe"
```

SHA1

- Function

  Returns the SHA1 value of the **expr** string.
- Syntax
```
STRING SHA1(STRING expr)
```
- Parameter description
  - **expr**: character string.
- Example
  - Test statement
```
SELECT SHA1("abc");
```
  - Test result
```
"a9993e364706816aba3e25717850c26c9cd0d89d"
```

SHA256

- Function

  Returns the SHA256 value of the expr string.
- Syntax
```
STRING SHA256(STRING expr)
```
- Parameter description
  - **expr**: character string.
- Example
  - Test statement
```
SELECT SHA256("abc");
```
  - Test result
```
"ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"
```

STRING_TO_ARRAY

- Function

  Separates the **value** string as character string arrays by using the delimiter.

  NOTE

  delimiter uses the Java regular expression. If special characters are used, they need to be escaped.

- Syntax
  ```
  ARRAY[String] STRING_TO_ARRAY(STRING value, VARCHAR delimiter)
  ```
- Parameter description
  - **value**: character string.
  - **delimiter**: specifies the delimiter.
- Example
  - Test statement
    ```
    SELECT
      string_to_array("127.0.0.1", "\\."),
      string_to_array("red-black-white-blue", "-");
    ```
  - Test result
    ```
    [127,0,0,1],[red,black,white,blue]
    ```

SUBSTRING

- Function

  Returns the substring that starts from a fixed position of A. The start position starts from 1.

  - If **len** is not specified, the substring from the start position to the end of the string is truncated.
  - If **len** is specified, the substring starting from the position specified by **start** is truncated. The length is specified by **len**.

  NOTE

  The value of **start** starts from **1**. If the value is **0**, it is regarded as **1**. If the value of start is a negative number, the position is calculated from the end of the character string in reverse order.

- Syntax
  ```
  VARCHAR SUBSTRING(STRING A FROM INT start)
  ```
  Or
  ```
  VARCHAR SUBSTRING(STRING A FROM INT start FOR INT len)
  ```
- Parameter description
  - **A**: specified character string.
  - **start**: start position for truncating the character string **A**.
  - **len**: intercepted length.
- Example
  - Test statement 1
    ```
    SELECT SUBSTRING("123456" FROM 2);
    ```
  - Test result 1
    ```
    "23456"
    ```
  - Test statement 2

```
SELECT SUBSTRING("123456" FROM 2 FOR 4);
```

  - Test result 2

```
"2345"
```

TRIM

- Function

  Remove A at the start position, or end position, or both the start and end positions from B. By default, string expressions A at both the start and end positions are removed.

- Syntax

```
STRING TRIM( { BOTH | LEADING | TRAILING } STRING a FROM STRING b)
```

- Parameter description
  - **a**: character string.
  - **b**: character string.
- Example
  - Test statement

```
SELECT TRIM(BOTH " " FROM "  hello world  ");
```

  - Test result

```
"hello world"
```

UPPER

- Function

  Returns a string converted to an uppercase character.

- Syntax

```
VARCHAR UPPER(A)
```

- Parameter description
  - **A**: character string.
- Example
  - Test statement

```
SELECT UPPER("hello world");
```

  - Test result

```
"HELLO WORLD"
```

2.9.3 Temporal Functions

Table 2-42 lists the time functions supported by Flink SQL.

Function Description

Table 2-42 Time Function

Function	Return Type	Description
DATE string	DATE	Parse the date string (**yyyy-MM-dd**) to a SQL date.

Function	Return Type	Description
TIME string	TIME	Parse the time string (**HH:mm:ss**) to the SQL time.
TIMESTAMP string	TIMESTAMP	Convert the time string into timestamp. The time string format is **yyyy-MM-dd HH:mm:ss.fff**.
INTERVAL string range	INTERVAL	There are two types of intervals: **yyyy-MM** and **dd HH:mm:sss.fff'**. The range of yyyy-MM can be YEAR or YEAR TO MONTH, with the precision of month. The range of dd HH:mm:sss.fff' can be DAY TO HOUR, DAY TO MINUTE, DAY TO SECOND, or DAY TO MILLISECONDS, with the precision of millisecond. For example, if the range is DAY TO SECOND, the day, hour, minute, and second are all valid and the precision is second. DAY TO MINUTE indicates that the precision is minute. The following is an example: **INTERVAL '10 00:00:00.004' DAY TO milliseconds** indicates that the interval is 10 days and 4 milliseconds. **INTERVAL '10' DAY** indicates that the interval is 10 days and **INTERVAL '2-10' YEAR TO MONTH** indicates that the interval is 2 years and 10 months.
CURRENT_DATE	DATE	Return the SQL date of UTC time zone.
CURRENT_TIME	TIME	Return the SQL time of UTC time zone.
CURRENT_TIMESTAMP	TIMESTAMP	Return the SQL timestamp of UTC time zone.
LOCALTIME	TIME	Return the SQL time of the current time zone.
LOCALTIMESTAMP	TIMESTAMP	Return the SQL timestamp of the current time zone.
EXTRACT(timeintervalunit FROM temporal)	INT	Extract part of the time point or interval. Return the part in the int type. For example, **5** is returned from **EXTRACT(DAY FROM DATE "2006-06-05")**.
FLOOR(timepoint TO timeintervalunit)	TIME	Round a time point down to the given unit. For example, **12:44:00** is returned from **FLOOR(TIME '12:44:31' TO MINUTE)**.

Function	Return Type	Description
CEIL(timepoint TO timeintervalunit)	TIME	Round a time point up to the given unit. For example, **12:45:00** is returned from CEIL(TIME '12:44:31' TO MINUTE).
QUARTER(date)	INT	Return the quarter from the SQL date.
(timepoint, temporal) OVERLAPS (timepoint, temporal)	BOOLEAN	Check whether two intervals overlap. The time points and time are converted into a time range with a start point and an end point. The function is ***leftEnd >= rightStart && rightEnd >= leftStart***. If leftEnd is greater than or equal to rightStart and rightEnd is greater than or equal to leftStart, **true** is returned. Otherwise, **false** is returned. The following is an example: ● If leftEnd is **3:55:00** (2:55:00+1:00:00), rightStart is **3:30:00**, rightEnd is **5:30:00** (3:30:00+2:00:00), and leftStart is **2:55:00**, **true** will be returned. Specifically, **true** is returned from (TIME '2:55:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR). ● If leftEnd is **10:00:00**, rightStart is **10:15:00**, rightEnd is **13:15:00** (10:15:00+3:00:00), and leftStart is **9:00:00**, **false** will be returned. Specifically, **false** is returned from (TIME '9:00:00', TIME '10:00:00') OVERLAPS (TIME '10:15:00', INTERVAL '3' HOUR).
TO_TIMESTAMP(long expr)	TIMESTAMP	Convert a timestamp to time. The input parameter this function must be of the BIGINT type. Other data types, such as VARCHAR and STRING, are not supported. For example, **TO_TIMESTAMP (1628765159000)** is converted to **2021-08-12 18:45:59**.

Function	Return Type	Description
UNIX_TIMESTAMP	BIGINT	Returns the timestamp of a specified parameter. The timestamp type is BIGINT and the unit is **second**. The following methods are supported: ● UNIX_TIMESTAMP(): returns the timestamp of the current time if no parameter is specified. ● UNIX_TIMESTAMP(STRING datestr): returns the timestamp indicated by the parameter if only one parameter is contained. The format of **datestr** must be yyyy-MM-dd HH:mm:ss. ● UNIX_TIMESTAMP(STRING datestr, STRING format): returns the timestamp indicated by the first parameter if two parameters are contained. The second parameter can specify the format of **datestr**.
UNIX_TIMESTAMP_MS	BIGINT	Returns the timestamp of a specified parameter. The timestamp type is BIGINT and the unit is **millisecond**. The following methods are supported: ● UNIX_TIMESTAMP_MS(): returns the timestamp of the current time if no parameter is specified. ● UNIX_TIMESTAMP_MS(STRING datestr): returns the timestamp indicated by the parameter if only one parameter is contained. The format of **datestr** must be yyyy-MM-dd HH:mm:ss.SSS. ● UNIX_TIMESTAMP_MS(STRING datestr, STRING format): returns the timestamp indicated by the first parameter if two parameters are contained. The second parameter can specify the format of **datestr**.

Precautions

None

Example

```
insert into temp SELECT Date '2015-10-11'  FROM  OrderA;//Date is returned
insert into temp1 SELECT Time '12:14:50'  FROM  OrderA;//Time is returned
insert into temp2 SELECT Timestamp '2015-10-11 12:14:50'  FROM  OrderA;//Timestamp is returned
```

2.9.4 Type Conversion Functions

Syntax

```
CAST(value AS type)
```

Syntax Description

This function is used to forcibly convert types.

Precautions

- If the input is **NULL**, **NULL** is returned.
- Flink jobs do not support the conversion of **bigint** to **timestamp** using CAST. You can convert it using **to_timestamp** or **to_localtimestamp**.

Example

Convert amount into a character string. The specified length of the string is invalid after the conversion.

```
insert into temp select cast(amount as VARCHAR(10)) from source_stream;
```

Common Type Conversion Functions

Table 2-43 Common type conversion functions

Function	Description
cast(v1 as varchar)	Converts **v1** to a string. The value of **v1** can be of the numeric type or of the timestamp, date, or time type.
cast (v1 as int)	Converts **v1** to the **int** type. The value of **v1** can be a number or a character.
cast(v1 as timestamp)	Converts **v1** to the **timestamp** type. The value of **v1** can be of the **string**, **date**, or **time** type.
cast(v1 as date)	Converts **v1** to the **date** type. The value of **v1** can be of the **string** or **timestamp** type.

- cast(v1 as varchar)
  - Test statement
    ```
    SELECT cast(content as varchar) FROM T1;
    ```
  - Test data and result

Table 2-44 T1

content (INT)	varchar
5	"5"

- cast (v1 as int)
  - Test statement
    ```
    SELECT cast(content as int) FROM T1;
    ```
  - Test data and result

Table 2-45 T1

content (STRING)	int
"5"	5

- cast(v1 as timestamp)
  - Test statement
    ```
    SELECT cast(content as timestamp) FROM T1;
    ```
  - Test data and result

Table 2-46 T1

content (STRING)	timestamp
"2018-01-01 00:00:01"	1514736001000

- cast(v1 as date)
  - Test statement
    ```
    SELECT cast(content as date) FROM T1;
    ```
  - Test data and result

Table 2-47 T1

content (TIMESTAMP)	date
1514736001000	"2018-01-01"

Detailed Sample Code

```
/** source **/
CREATE
SOURCE STREAM car_infos (cast_int_to_varchar int, cast_String_to_int string,
case_string_to_timestamp string, case_timestamp_to_date timestamp) WITH (
  type = "dis",
  region = "xxxxx",
  channel = "dis-input",
  partition_count = "1",
  encode = "json",
  offset = "13",
  json_config =
```

```
"cast_int_to_varchar=cast_int_to_varchar;cast_String_to_int=cast_String_to_int;case_string_to_timestamp=cas
e_string_to_timestamp;case_timestamp_to_date=case_timestamp_to_date"

);
/** sink **/
CREATE
SINK STREAM cars_infos_out (cast_int_to_varchar varchar, cast_String_to_int
int, case_string_to_timestamp timestamp, case_timestamp_to_date date) WITH (
  type = "dis",
  region = "xxxxx",
  channel = "dis-output",
  partition_count = "1",
  encode = "json",
  offset = "4",
  json_config =
"cast_int_to_varchar=cast_int_to_varchar;cast_String_to_int=cast_String_to_int;case_string_to_timestamp=cas
e_string_to_timestamp;case_timestamp_to_date=case_timestamp_to_date",
  enable_output_null="true"
);
/** Statistics on static car information**/
INSERT
INTO
  cars_infos_out
SELECT
  cast(cast_int_to_varchar as varchar),
  cast(cast_String_to_int as int),
  cast(case_string_to_timestamp as timestamp),
  cast(case_timestamp_to_date as date)
FROM
  car_infos;
```

Returned data

```
{"case_string_to_timestamp":
1514736001000,"cast_int_to_varchar":"5","case_timestamp_to_date":"2018-01-01","cast_String_to_int":100}
```

2.9.5 Aggregate Functions

An aggregate function performs a calculation operation on a set of input values and returns a value. For example, the COUNT function counts the number of rows retrieved by an SQL statement. **Table 2-48** lists aggregate functions.

Sample data: Table T1

```
|score|
|81   |
|100  |
|60   |
|95   |
|86   |
```

Common Aggregate Functions

Table 2-48 Common aggregation functions

Function	Return Data Type	Description
COUNT(*)	BIGINT	Return count of tuples.
COUNT([ALL] expression...	BIGINT	Returns the number of input rows for which the expression is not NULL. Use DISTINCT for a unique instance of each value.

Function	Return Data Type	Description
AVG(numeric)	DOUBLE	Return average (arithmetic mean) of all input values.
SUM(numeric)	DOUBLE	Return the sum of all input numerical values.
MAX(value)	DOUBLE	Return the maximum value of all input values.
MIN(value)	DOUBLE	Return the minimum value of all input values.
STDDEV_POP(value)	DOUBLE	Return the population standard deviation of all numeric fields of all input values.
STDDEV_SAMP(value)	DOUBLE	Return the sample standard deviation of all numeric fields of all input values.
VAR_POP(value)	DOUBLE	Return the population variance (square of population standard deviation) of numeral fields of all input values.
VAR_SAMP(value)	DOUBLE	Return the sample variance (square of the sample standard deviation) of numeric fields of all input values.

Example

- COUNT(*)
  - Test statement
    ```
    SELECT COUNT(score) FROM T1;
    ```
  - Test data and results

    Table 2-49 T1

Test Data (score)	Test Result
81	5
100	
60	
95	
86	

- COUNT([ALL] expression | DISTINCT expression1 [, expression2]*)
  - Test statement
    ```
    SELECT COUNT(DISTINCT content ) FROM T1;
    ```

– Test data and results

Table 2-50 T1

content (STRING)	Test Result
"hello1 "	2
"hello2 "	
"hello2"	
null	
86	

- AVG(numeric)
  – Test statement
```
SELECT AVG(score) FROM T1;
```
  – Test data and results

Table 2-51 T1

Test Data (score)	Test Result
81	84.0
100	
60	
95	
86	

- SUM(numeric)
  – Test statement
```
SELECT SUM(score) FROM T1;
```
  – Test data and results

Table 2-52 T1

Test Data (score)	Test Result
81	422.0
100	
60	
95	
86	

- MAX(value)
  - Test statement
    ```
    SELECT MAX(score) FROM T1;
    ```
  - Test data and results

Table 2-53 T1

Test Data (score)	Test Result
81	100.0
100	
60	
95	
86	

- MIN(value)
  - Test statement
    ```
    SELECT MIN(score) FROM T1;
    ```
  - Test data and results

Table 2-54 T1

Test Data (score)	Test Result
81	60.0
100	
60	
95	
86	

- STDDEV_POP(value)
  - Test statement
    ```
    SELECT STDDEV_POP(score) FROM T1;
    ```
  - Test data and results

Table 2-55 T1

Test Data (score)	Test Result
81	13.0
100	
60	
95	

Test Data (score)	Test Result
86	

- STDDEV_SAMP(value)
  - Test statement
    ```
    SELECT STDDEV_SAMP(score) FROM T1;
    ```
  - Test data and results

Table 2-56 T1

Test Data (score)	Test Result
81	15.0
100	
60	
95	
86	

- VAR_POP(value)
  - Test statement
    ```
    SELECT VAR_POP(score) FROM T1;
    ```
  - Test data and results

Table 2-57 T1

Test Data (score)	Test Result
81	193.0
100	
60	
95	
86	

- VAR_SAMP(value)
  - Test statement
    ```
    SELECT VAR_SAMP(score) FROM T1;
    ```
  - Test data and results

Table 2-58 T1

Test Data (score)	Test Result
81	241.0

Test Data (score)	Test Result
100	
60	
95	
86	

2.9.6 Table-Valued Functions

Table-valued functions can convert one row of records into multiple rows or convert one column of records into multiple columns. Table-valued functions can only be used in JOIN LATERAL TABLE.

Table 2-59 Table-valued functions

Function	Return Data Type	Description
split_cursor(value, delimiter)	cursor	Separates the "value" string into multiple rows of strings by using the delimiter.

Example

Input one record ("student1", "student2, student3") and output two records ("student1", "student2") and ("student1", "student3").

```
create source stream s1(attr1 string, attr2 string) with (......);
insert into s2 select  attr1, b1 from s1 left join lateral table(split_cursor(attr2, ',')) as T(b1) on true;
```

2.9.7 Other Functions

Array Functions

Table 2-60 Array functions

Function	Return Data Type	Description
CARDINALITY(ARRAY)	INT	Return the element count of an array.
ELEMENT(ARRAY)	-	Return the sole element of an array with a single element. If the array contains no elements, **null** is returned. If the array contains multiple elements, an exception is reported.

Example:

The returned number of elements in the array is 3.

```
insert into temp select CARDINALITY(ARRAY[TRUE, TRUE, FALSE]) from source_stream;
```

HELLO WORLD is returned.

```
insert into temp select ELEMENT(ARRAY['HELLO WORLD']) from source_stream;
```

Attribute Access Functions

Table 2-61 Attribute access functions

Function	Return Data Type	Description
tableName.compositeType.field	-	Select a single field, use the name to access the field of Apache Flink composite types, such as Tuple and POJO, and return the value.
tableName.compositeType.*	-	Select all fields, and convert Apache Flink composite types, such as Tuple and POJO, and all their direct subtypes into a simple table. Each subtype is a separate field.

2.10 User-Defined Functions

Overview

DLI supports the following three types of user-defined functions (UDFs):

- Regular UDF: takes in one or more input parameters and returns a single result.
- User-defined table-generating function (UDTF): takes in one or more input parameters and returns multiple rows or columns.
- User-defined aggregate function (UDAF): aggregates multiple records into one value.

NOTE

UDFs can only be used in dedicated queues.

POM Dependency

```
<dependency>
   <groupId>org.apache.flink</groupId>
   <artifactId>flink-table_2.11</artifactId>
   <version>1.7.2</version>
   <scope>provided</scope>
</dependency>
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-java_2.11</artifactId>
```

```
        <version>1.7.2</version>
        <scope>provided</scope>
</dependency>
```

Precautions

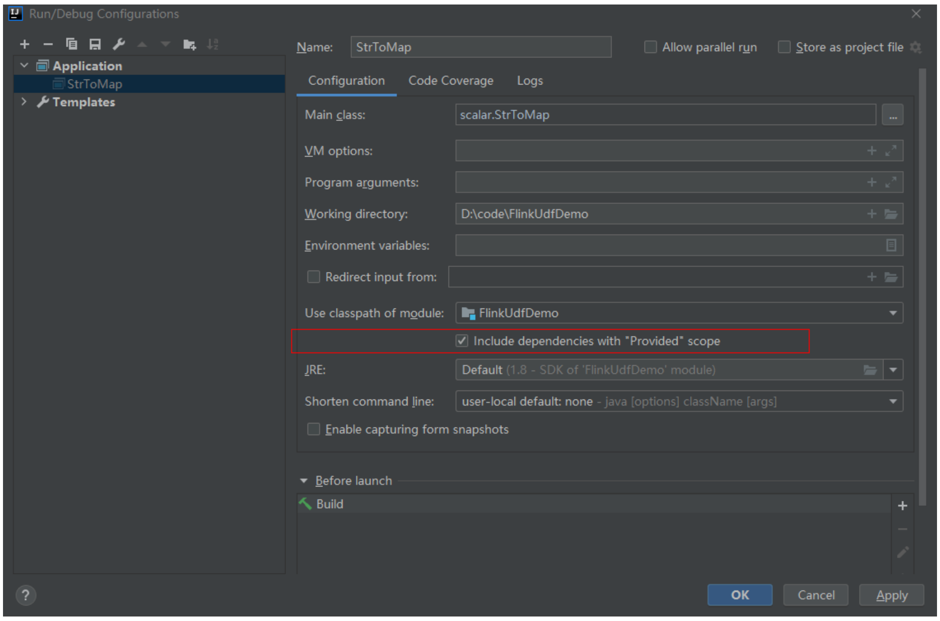
- **Currently, Python is not supported for programming UDFs, UDTFs, and UDAFs.**
- If you use IntelliJ IDEA to debug the created UDF, select **include dependencies with "Provided" scope**. Otherwise, the dependency packages in the POM file cannot be loaded for local debugging.

  The following uses IntelliJ IDEA 2020.2 as an example:

  a. On the IntelliJ IDEA page, select the configuration file you need to debug and click **Edit Configurations**.

  b. On the **Run/Debug Configurations** page, select **include dependencies with "Provided" scope**.

  c. Click **OK**.

Using UDFs

1. Write the code of custom functions. For details about the code examples, see **UDF**, **UDTF**, or **UDAF**.

2. Compile the UDF code, pack it into a JAR package, and upload the package to OBS.
3. In the left navigation pane of the DLI management console, click **Job Management** > **Flink Jobs**. Locate the row where the target resides and click **Edit** in the **Operation** column to switch to the page where you can edit the job.
4. On the **Running Parameters** tab page, select an exclusive queue for **Queue**. The **UDF Jar** parameter is displayed. Select the JAR file stored on OBS and click **Save**.

   NOTE

   Before selecting a user-defined function JAR package, upload the JAR package to the created OBS bucket.

   After the JAR package is selected, add the UDF statement to the SQL statement.

UDF

The regular UDF must inherit the ScalarFunction function and implement the eval method. The open and close functions are optional.

Example code

```
import org.apache.flink.table.functions.FunctionContext;
import org.apache.flink.table.functions.ScalarFunction;
public class UdfScalarFunction extends ScalarFunction {
  private int factor = 12;
  public UdfScalarFunction() {
    this.factor = 12;
  }
  /**
   * (optional) Initialization
   * @param context
   */
  @Override
  public void open(FunctionContext context) {}
  /**
   * Custom logic
   * @param s
   * @return
   */
   public int eval(String s) {
     return s.hashCode() * factor;
   }
   /**
    * Optional
    */
   @Override
   public void close() {}
}
```

Example

```
CREATE FUNCTION udf_test AS 'com.xxx.udf.UdfScalarFunction';
INSERT INTO sink_stream select udf_test(attr) FROM source_stream;
```

UDTF

The UDTF must inherit the TableFunction function and implement the eval method. The open and close functions are optional. If the UDTF needs to return multiple columns, you only need to declare the returned value as **Tuple** or **Row**. If

Row is used, you need to overload the getResultType method to declare the returned field type.

Example code

```
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.table.functions.FunctionContext;
import org.apache.flink.table.functions.TableFunction;
import org.apache.flink.types.Row;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
public class UdfTableFunction extends TableFunction<Row> {
  private Logger log = LoggerFactory.getLogger(TableFunction.class);
  /**
   * (optional) Initialization
   * @param context
   */
  @Override
  public void open(FunctionContext context) {}
  public void eval(String str, String split) {
    for (String s : str.split(split)) {
      Row row = new Row(2);
      row.setField(0, s);
      row.setField(1, s.length());
      collect(row);
    }
  }
  /**
   * Declare the type returned by the function
   * @return
   */
  @Override
  public TypeInformation<Row> getResultType() {
  return Types.ROW(Types.STRING, Types.INT);
  }
  /**
    * Optional
   */
  @Override
  public void close() {}
 }
```

Example

The UDTF supports CROSS JOIN and LEFT JOIN. When the UDTF is used, the **LATERAL** and **TABLE** keywords must be included.

- CROSS JOIN: does not output the data of a row in the left table if the UDTF does not output the result for the data of the row.
- LEFT JOIN: outputs the data of a row in the left table even if the UDTF does not output the result for the data of the row, but pads null with UDTF-related fields.

```
CREATE FUNCTION udtf_test AS 'com.xxx.udf.TableFunction';
// CROSS JOIN
INSERT INTO sink_stream select subValue, length FROM source_stream, LATERAL
TABLE(udtf_test(attr, ',')) as T(subValue, length);
// LEFT JOIN
INSERT INTO sink_stream select subValue, length FROM source_stream LEFT JOIN LATERAL
TABLE(udtf_test(attr, ',')) as T(subValue, length) ON TRUE;
```

UDAF

The UDAF must inherit the AggregateFunction function. You need to create an accumulator for storing the computing result, for example, **WeightedAvgAccum** in the following example code.

Example code

```
public class WeightedAvgAccum {
public long sum = 0;
public int count = 0;
}
```

```
import org.apache.flink.table.functions.AggregateFunction;
import java.util.Iterator;
/**
* The first type variable is the type returned by the aggregation function, and the second type variable is of
the Accumulator type.
* Weighted Average user-defined aggregate function.
*/
public class UdfAggFunction extends AggregateFunction<Long, WeightedAvgAccum> {
// Initialize the accumulator.
  @Override
  public WeightedAvgAccum createAccumulator() {
    return new WeightedAvgAccum();
  }
// Return the intermediate computing value stored in the accumulator.
  @Override
  public Long getValue(WeightedAvgAccum acc) {
    if (acc.count == 0) {
       return null;
    } else {
      return acc.sum / acc.count;
 }
}
// Update the intermediate computing value according to the input.
public void accumulate(WeightedAvgAccum acc, long iValue) {
acc.sum += iValue;
acc.count += 1;
}
// Perform the retraction operation, which is opposite to the accumulate operation.
public void retract(WeightedAvgAccum acc, long iValue) {
acc.sum -= iValue;
acc.count -= 1;
}
// Combine multiple accumulator values.
public void merge(WeightedAvgAccum acc, Iterable<WeightedAvgAccum> it) {
Iterator<WeightedAvgAccum> iter = it.iterator();
while (iter.hasNext()) {
WeightedAvgAccum a = iter.next();
acc.count += a.count;
acc.sum += a.sum;
}
}
// Reset the intermediate computing value.
public void resetAccumulator(WeightedAvgAccum acc) {
acc.count = 0;
acc.sum = 0L;
}
}
```

Example

```
CREATE FUNCTION udaf_test AS 'com.xxx.udf.UdfAggFunction';
INSERT INTO sink_stream SELECT udaf_test(attr2) FROM source_stream GROUP BY attr1;
```

2.11 Geographical Functions

Function description

Table 2-62 describes the basic geospatial geometric elements.

Table 2-62 Basic geospatial geometric element table

Geospatial geometric elements	Description	Example Value
ST_POINT(latitude, longitude)	Indicates a geographical point, including the longitude and latitude.	ST_POINT(1.12012, 1.23401)
ST_LINE(array[point1...pointN])	Indicates a geographical line formed by connecting multiple geographical points (ST_POINT) in sequence. The line can be a polygonal line or a straight line.	ST_LINE(ARRAY[ST_POINT(1.12, 2.23), ST_POINT(1.13, 2.44), ST_POINT(1.13, 2.44)])
ST_POLYGON(array[point1...point1])	Indicates a geographical polygon, which is a closed polygon area formed by connecting multiple geographical points (ST_POINT) with the same start and end points in sequence.	ST_POLYGON(ARRAY[ST_POINT(1.0, 1.0), ST_POINT(2.0, 1.0), ST_POINT(2.0, 2.0), ST_POINT(1.0, 1.0)])
ST_CIRCLE(point, radius)	Indicates a geographical circle that consists of ST_POINT and a radius.	ST_CIRCLE(ST_POINT(1.0, 1.0), 1.234)

You can build complex geospatial geometries based on basic geospatial geometric elements. Table 2-63 describes the related transformation methods.

Table 2-63 Transformation methods for building complex geometric elements based on basic geospatial geometric elements

Transformation Method	Description	Example Value
ST_BUFFER(geometry, distance)	Creates a polygon that surrounds the geospatial geometric elements at a given distance. Generally, this function is used to build the road area of a certain width for yaw detection.	ST_BUFFER(ST_LINE(ARRAY[ST_POINT(1.12, 2.23), ST_POINT(1.13, 2.44), ST_POINT(1.13, 2.44)]),1.0)

Transformation Method	Description	Example Value
ST_INTERSECTION(geometry, geometry)	Creates a polygon that delimits the overlapping area of two given geospatial geometric elements.	ST_INTERSECTION(ST_CIRCLE(ST_POINT(1.0, 1.0), 2.0), ST_CIRCLE(ST_POINT(3.0, 1.0), 1.234))
ST_ENVELOPE(geometry)	Creates the minimal rectangle polygon including the given geospatial geometric elements.	ST_ENVELOPE(ST_CIRCLE(ST_POINT(1.0, 1.0), 2.0))

DLI provides multiple functions used for performing operations on and determining locations of geospatial geometric elements. **Table 2-64** describes the SQL scalar functions.

Table 2-64 SQL scalar function table

Function	Return Type	Description
ST_DISTANCE(point_1, point_2)	DOUBLE	Calculates the Euclidean distance between the two geographical points. The following provides an example: Select ST_DISTANCE(ST_POINT(x1, y1), ST_POINT(x2, y2)) FROM input
ST_GEODESIC_DISTANCE(point_1, point_2)	DOUBLE	Calculates the shortest distance along the surface between two geographical points. The following provides an example: Select ST_GEODESIC_DISTANCE(ST_POINT(x1, y1), ST_POINT(x2, y2)) FROM input
ST_PERIMETER(polygon)	DOUBLE	Calculates the circumference of a polygon. The following provides an example: Select ST_PERIMETER(ST_POLYGON(ARRAY[ST_POINT(x11, y11), ST_POINT(x12, y12), ST_POINT(x11, y11)]) FROM input
ST_AREA(polygon)	DOUBLE	Calculates the area of a polygon. The following provides an example: Select ST_AREA(ST_POLYGON(ARRAY[ST_POINT(x11, y11), ST_POINT(x12, y12), ST_POINT(x11, y11)]) FROM input

Function	Return Type	Description
ST_OVERLAPS(polygon_1, polygon_2)	BOOLEAN	Checks whether one polygon overlaps with another. The following provides an example: SELECT ST_OVERLAPS(ST_POLYGON(ARRAY[ST_POINT(x11, y11), ST_POINT(x12, y12), ST_POINT(x11, y11)]), ST_POLYGON(ARRAY[ST_POINT(x21, y21), ST_POINT(x22, y22), ST_POINT(x23, y23), ST_POINT(x21, y21)])) FROM input
ST_INTERSECT(line1, line2)	BOOLEAN	Checks whether two line segments, rather than the two straight lines where the two line segments are located, intersect each other. The following provides an example: SELECT ST_INTERSECT(ST_LINE(ARRAY[ST_POINT(x11, y11), ST_POINT(x12, y12)]), ST_LINE(ARRAY[ST_POINT(x21, y21), ST_POINT(x22, y22), ST_POINT(x23, y23)])) FROM input
ST_WITHIN(point, polygon)	BOOLEAN	Checks whether one point is contained inside a geometry (polygon or circle). The following provides an example: SELECT ST_WITHIN(ST_POINT(x11, y11), ST_POLYGON(ARRAY[ST_POINT(x21, y21), ST_POINT(x22, y22), ST_POINT(x23, y23), ST_POINT(x21, y21)])) FROM input
ST_CONTAINS(polygon_1, polygon_2)	BOOLEAN	Checks whether the first geometry contains the second geometry. The following provides an example: SELECT ST_CONTAINS(ST_POLYGON(ARRAY[ST_POINT(x11, y11), ST_POINT(x12, y12), ST_POINT(x11, y11)]), ST_POLYGON(ARRAY[ST_POINT(x21, y21), ST_POINT(x22, y22), ST_POINT(x23, y23), ST_POINT(x21, y21)])) FROM input

Function	Return Type	Description
ST_COVERS(polygon_1, polygon_2)	BOOLEAN	Checks whether the first geometry covers the second geometry. This function is similar to ST_CONTAINS except the situation when judging the relationship between a polygon and the boundary line of polygon, for which ST_COVER returns TRUE and ST_CONTAINS returns FALSE. The following provides an example: SELECT ST_COVERS(ST_POLYGON(ARRAY[ST_POINT(x11, y11), ST_POINT(x12, y12), ST_POINT(x11, y11)]), ST_POLYGON([ST_POINT(x21, y21), ST_POINT(x22, y22), ST_POINT(x23, y23), ST_POINT(x21, y21)])) FROM input
ST_DISJOINT(polygon_1, polygon_2)	BOOLEAN	Checks whether one polygon is disjoint (not overlapped) with the other polygon. The following provides an example: SELECT ST_DISJOINT(ST_POLYGON(ARRAY[ST_POINT(x11, y11), ST_POINT(x12, y12), ST_POINT(x11, y11)]), ST_POLYGON(ARRAY[ST_POINT(x21, y21), ST_POINT(x22, y22), ST_POINT(x23, y23), ST_POINT(x21, y21)])) FROM input

The World Geodetic System 1984 (WGS84) is used as the reference coordinate system for geographical functions. Due to offsets, the GPS coordinates cannot be directly used in the Baidu Map (compliant with BD09) and the Google Map (compliant with GCJ02). To implement switchover between different geographical coordinate systems, DLI provides a series of functions related to coordinate system conversion as well as functions related to conversion between geographical distances and the unit meter. For details, see **Table 2-65**.

Table 2-65 Functions for geographical coordinate system conversion and distance-unit conversion

Function	Return Type	Description
WGS84_TO_BD09(geometry)	Geospatial geometric elements in the Baidu Map coordinate system	Converts the geospatial geometric elements in the GPS coordinate system into those in the Baidu Map coordinate system. The following provides an example: WGS84_TO_BD09(ST_CIRCLE(ST_POINT(x, y), r))
WGS84_TO_CJ02(geometry)	Geospatial geometric elements in the Google Map coordinate system	Converts the geospatial geometric elements in the GPS coordinate system into those in the Google Map coordinate system. The following provides an example: WGS84_TO_CJ02(ST_CIRCLE(ST_POINT(x, y), r))
BD09_TO_WGS84(geometry)	Geospatial geometric elements in the GPS coordinate system	Converts the geospatial geometric elements in the Baidu Map coordinate system into those in the GPS coordinate system. The following provides an example: BD09_TO_WGS84(ST_CIRCLE(ST_POINT(x, y), r))
BD09_TO_CJ02(geometry)	Geospatial geometric elements in the Google Map coordinate system	Converts the geospatial geometric elements in the Baidu Map coordinate system into those in the Google Map coordinate system. The following provides an example: BD09_TO_CJ02(ST_CIRCLE(ST_POINT(x, y), r))

Function	Return Type	Description
CJ02_TO_WGS84(geometry)	Geospatial geometric elements in the GPS coordinate system	Converts the geospatial geometric elements in the Google Map coordinate system into those in the GPS coordinate system. The following provides an example: CJ02_TO_WGS84(ST_CIRCLE(ST_POINT(x, y), r))
CJ02_TO_BD09(geometry)	Geospatial geometric elements in the Baidu Map coordinate system	Converts the geospatial geometric elements in the Google Map coordinate system into those in the Baidu Map coordinate system. The following provides an example: CJ02_TO_BD09(ST_CIRCLE(ST_POINT(x, y), r))
DEGREE_TO_METER(distance)	DOUBLE	Converts the distance value of the geographical function to a value in the unit of meter. In the following example, you calculate the circumference of a triangle in the unit of meter. DEGREE_TO_METER(ST_PERIMETER(ST_POLYGON(ARRAY[ST_POINT(x1,y1), ST_POINT(x2,y2), ST_POINT(x3,y3), ST_POINT(x1,y1)])))

Function	Return Type	Description
METER_TO_DEGREE(numerical_value)	DOUBLE	Convert the value in the unit of meter to the distance value that can be calculated using the geographical function. In the following example, you draw a circle which takes a specified geographical point as the center and has a radius of 1 km. ST_CIRCLE(ST_POINT(x,y), METER_TO_DEGREE(1000))

DLI also provides window-based SQL geographical aggregation functions specific for scenarios where SQL logic involves windows and aggregation. For details about the functions, see **Table 2-66**.

Table 2-66 Time-related SQL geographical aggregation function table

Function	Description	Example Value
AGG_DISTANCE(point)	Distance aggregation function, which is used to calculate the total distance of all adjacent geographical points in the window.	SELECT AGG_DISTANCE(ST_POINT(x,y)) FROM input GROUP BY HOP(rowtime, INTERVAL '1' HOUR, INTERVAL '1' DAY)
AVG_SPEED(point)	Average speed aggregation function, which is used to calculate the average speed of moving tracks formed by all geographical points in a window. The average speed is in the unit of m/s.	SELECT AVG_SPEED(ST_POINT(x,y)) FROM input GROUP BY TUMBLE(proctime, INTERVAL '1' DAY)

Precautions

None

Example

Example of yaw detection:

```
INSERT INTO yaw_warning
SELECT "The car is yawing"
FROM driver_behavior
WHERE NOT ST_WITHIN(ST_POINT(cast(Longitude as DOUBLE), cast(Latitude as DOUBLE)),
ST_BUFFER(ST_LINE(ARRAY[ST_POINT(34.585555,105.725221),ST_POINT(34.586729,105.735974),ST_POINT(
34.586492,105.740538),ST_POINT(34.586388,105.741651),ST_POINT(34.586135,105.748712),ST_POINT(34.5
88691,105.74997)]),0.001));
```

IP Functions

NOTE

Currently, only IPv4 addresses are supported.

Table 2-67 IP functions

Function	Return Type	Description
IP_TO_COUNTRY	STRING	Obtains the name of the country where the IP address is located.
IP_TO_PROVINCE	STRING	Obtains the province where the IP address is located. Usage: • IP_TO_PROVINCE(STRING ip): Determines the province where the IP address is located and returns the province name. • IP_TO_PROVINCE(STRING ip, STRING lang): Determines the province where the IP is located and returns the province name of the specified language. **NOTE** • If the province where the IP address is located cannot be obtained through IP address parsing, the country where the IP address is located is returned. If the IP address cannot be parsed, **Unknown** is returned. • The name returned by the function for the province is the short name.
IP_TO_CITY	STRING	Obtains the name of the city where the IP address is located. **NOTE** If the city where the IP address is located cannot be obtained through IP address parsing, the province or the country where the IP address is located is returned. If the IP address cannot be parsed, **Unknown** is returned.

Function	Return Type	Description
IP_TO_CITY_GEO	STRING	Obtains the longitude and latitude of the city where the IP address is located. The parameter value is in the following format: *Latitude*, *Longitude*. Usage: IP_TO_CITY_GEO(STRING ip): Returns the longitude and latitude of the city where the IP address is located.

2.12 SELECT

SELECT

Syntax

```
SELECT [ ALL | DISTINCT ]  { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
```

Description

The SELECT statement is used to select data from a table or insert constant data into a table.

Precautions

- The table to be queried must exist. Otherwise, an error is reported.
- WHERE is used to specify the filtering condition, which can be the arithmetic operator, relational operator, or logical operator.
- GROUP BY is used to specify the grouping field, which can be one or more multiple fields.

Example

Select the order which contains more than 3 pieces of data.

```
insert into temp SELECT  * FROM Orders WHERE units > 3;
```

Insert a group of constant data.

```
insert into temp select 'Lily', 'male', 'student', 17;
```

WHERE Filtering Clause

Syntax

```
SELECT   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
```

Description

This statement is used to filter the query results using the WHERE clause.

Precautions

- The to-be-queried table must exist.
- WHERE filters the records that do not meet the requirements.

Example

Filter orders which contain more than 3 pieces and fewer than 10 pieces of data.

```
insert into temp SELECT  * FROM Orders
  WHERE units > 3 and units < 10;
```

HAVING Filtering Clause

Function

This statement is used to filter the query results using the HAVING clause.

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
```

Description

Generally, HAVING and GROUP BY are used together. GROUP BY applies first for grouping and HAVING then applies for filtering. The arithmetic operation and aggregate function are supported by the HAVING clause.

Precautions

If the filtering condition is subject to the query results of GROUP BY, the HAVING clause, rather than the WHERE clause, must be used for filtering.

Example

Group the **student** table according to the **name** field and filter the records in which the maximum score is higher than 95 based on groups.

```
insert into temp SELECT name, max(score) FROM student
  GROUP BY name
  HAVING max(score) >95
```

Column-Based GROUP BY

Function

This statement is used to group a table based on columns.

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
```

Description

Column-based GROUP BY can be categorized into single-column GROUP BY and multi-column GROUP BY.

- Single-column GROUP BY indicates that the GROUP BY clause contains only one column.
- Multi-column GROUP BY indicates that the GROUP BY clause contains multiple columns. The table will be grouped according to all fields in the GROUP BY clause. The records whose fields are the same are grouped into one group.

Precautions

None

Example

Group the **student** table according to the score and name fields and return the grouping results.

```
insert into temp SELECT name,score, max(score) FROM student
  GROUP BY name,score;
```

Expression-Based GROUP BY

Function

This statement is used to group a table according to expressions.

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
```

Description

groupItem can have one or more fields. The fields can be called by string functions, but cannot be called by aggregate functions.

Precautions

None

Example

Use the substring function to obtain the character string from the name field, group the **student** table according to the obtained character string, and return each sub character string and the number of records.

```
insert into temp SELECT substring(name,6),count(name) FROM student
  GROUP BY substring(name,6);
```

GROUP BY Using HAVING

Function

This statement filters a table after grouping it using the HAVING clause.

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
```

```
[ WHERE booleanExpression ]
[ GROUP BY { groupItem [, groupItem ]* } ]
[ HAVING booleanExpression ]
```

Description

Generally, HAVING and GROUP BY are used together. GROUP BY applies first for grouping and HAVING then applies for filtering.

Precautions

- If the filtering condition is subject to the query results of GROUP BY, the HAVING clause, rather than the WHERE clause, must be used for filtering. HAVING and GROUP BY are used together. GROUP BY applies first for grouping and HAVING then applies for filtering.
- Fields used in HAVING, except for those used for aggregate functions, must exist in GROUP BY.
- The arithmetic operation and aggregate function are supported by the HAVING clause.

Example

Group the **transactions** according to **num**, use the HAVING clause to filter the records in which the maximum value derived from multiplying **price** with **amount** is higher than 5000, and return the filtered results.

```
insert into temp SELECT num, max(price*amount) FROM transactions
  WHERE time > '2016-06-01'
  GROUP BY num
  HAVING max(price*amount)>5000;
```

UNION

Syntax

```
query UNION [ ALL ] query
```

Description

This statement is used to return the union set of multiple query results.

Precautions

- Set operation is to join tables from head to tail under certain conditions. The quantity of columns returned by each SELECT statement must be the same. Column types must be the same. Column names can be different.
- By default, the repeated records returned by UNION are removed. The repeated records returned by UNION ALL are not removed.

Example

Output the union set of Orders1 and Orders2 without duplicate records.

```
insert into temp SELECT  * FROM Orders1
  UNION SELECT  * FROM Orders2;
```

2.13 Condition Expression

CASE Expression

Syntax

```
CASE value WHEN value1 [, value11 ]* THEN result1
  [ WHEN valueN [, valueN1 ]* THEN resultN ]* [ ELSE resultZ ]
  END
```

or

```
CASE WHEN condition1 THEN result1
  [ WHEN conditionN THEN resultN ]* [ ELSE resultZ ]
  END
```

Description

- If the value of **value** is **value1**, **result1** is returned. If the value is not any of the values listed in the clause, **resultZ** is returned. If no else statement is specified, **null** is returned.
- If the value of **condition1** is **true**, **result1** is returned. If the value does not match any condition listed in the clause, **resultZ** is returned. If no else statement is specified, **null** is returned.

Precautions

- All results must be of the same type.
- All conditions must be of the Boolean type.
- If the value does not match any condition, the value of **ELSE** is returned when the else statement is specified, and **null** is returned when no else statement is specified.

Example

If the value of **units** equals **5**, **1** is returned. Otherwise, **0** is returned.

Example 1:

```
insert into temp SELECT  CASE units WHEN 5 THEN 1 ELSE 0 END FROM Orders;
```

Example 2:

```
insert into temp SELECT CASE WHEN units = 5 THEN 1 ELSE 0 END FROM Orders;
```

NULLIF Expression

Syntax

```
NULLIF(value, value)
```

Description

If the values are the same, **NULL** is returned. For example, **NULL** is returned from NULLIF (5,5) and **5** is returned from NULLIF (5,0).

Precautions

None

Example

If the value of **units** equals **3**, **null** is returned. Otherwise, the value of **units** is returned.

```
insert into temp SELECT  NULLIF(units, 3) FROM Orders;
```

COALESCE Expression

Syntax

```
COALESCE(value, value [, value ]* )
```

Description

Return the first value that is not **NULL**, counting from left to right.

Precautions

All values must be of the same type.

Example

5 is returned from the following example:

```
insert into temp SELECT  COALESCE(NULL, 5) FROM Orders;
```

2.14 Window

GROUP WINDOW

Description

Group Window is defined in GROUP BY. One record is generated from each group. Group Window involves the following functions:

- Array functions

Table 2-68 Array functions

Function Name	Description
TUMBLE(time_attr, interval)	Indicates the tumble window. **time_attr** can be set to **processing-time** or **event-time**. **interval** specifies the window period.
HOP(time_attr, interval, interval)	Indicates the extended tumble window (similar to the datastream sliding window). You can set the output triggering cycle and window period.

Function Name	Description
SESSION(time_attr, interval)	Indicates the session window. A session window will be closed if no response is returned within a duration specified by **interval**.

- Window functions

Table 2-69 Window functions

Function Name	Description
TUMBLE_START(time_attr, interval)	Indicates the start time of returning to the tumble window. The parameter is a UTC time zone.
TUMBLE_END(time_attr, interval)	Indicates the end time of returning to the tumble window. The parameter is a UTC time zone.
HOP_START(time_attr, interval, interval)	Indicates the start time of returning to the extended tumble window. The parameter is a UTC time zone.
HOP_END(time_attr, interval, interval)	Indicates the end time of returning to the extended tumble window. The parameter is a UTC time zone.
SESSION_START(time_attr, interval)	Indicates the start time of returning to the session window. The parameter is a UTC time zone.
SESSION_END(time_attr, interval)	Indicates the end time of returning to the session window. The parameter is a UTC time zone.

Example

```
//Calculate the SUM every day (event time).
insert into temp SELECT name,
    TUMBLE_START(ts, INTERVAL '1' DAY) as wStart,
    SUM(amount)
    FROM Orders
    GROUP BY TUMBLE(ts, INTERVAL '1' DAY), name;

//Calculate the SUM every day (processing time).
insert into temp SELECT name,
    SUM(amount)
    FROM Orders
    GROUP BY TUMBLE(proctime, INTERVAL '1' DAY), name;

//Calculate the SUM over the recent 24 hours every hour (event time).
insert into temp SELECT product,
    SUM(amount)
    FROM Orders
    GROUP BY HOP(ts, INTERVAL '1' HOUR, INTERVAL '1' DAY), product;
```

```
//Calculate the SUM of each session and an inactive interval every 12 hours (event time).
insert into temp SELECT name,
    SESSION_START(ts, INTERVAL '12' HOUR) AS sStart,
    SESSION_END(ts, INTERVAL '12' HOUR) AS sEnd,
    SUM(amount)
    FROM Orders
    GROUP BY SESSION(ts, INTERVAL '12' HOUR), name;
```

OVER WINDOW

The difference between Over Window and Group Window is that one record is generated from one row in Over Window.

Syntax

```
OVER (
  [PARTITION BY partition_name]
  ORDER BY proctime|rowtime(ROWS number PRECEDING) |(RANGE (BETWEEN INTERVAL '1' SECOND
PRECEDING AND CURRENT ROW | UNBOUNDED preceding))
)
```

Description

Table 2-70 Parameter description

Parameter	Parameter Description
PARTITION BY	Indicates the primary key of the specified group. Each group separately performs calculation.
ORDER BY	Indicates the processing time or event time as the timestamp for data.
ROWS	Indicates the count window.
RANGE	Indicates the time window.

Precautions

- In the same SELECT statement, windows defined by aggregate functions must be the same.
- Currently, Over Window only supports forward calculation (preceding).
- The value of **ORDER BY** must be specified as **processing time** or **event time**.
- Constants do not support aggregation, such as sum(2).

Example

```
//Calculate the count and total number from syntax rules enabled to now (in proctime).
insert into temp SELECT name,
    count(amount) OVER (PARTITION BY name ORDER BY proctime RANGE UNBOUNDED preceding) as
cnt1,
    sum(amount) OVER (PARTITION BY name ORDER BY proctime RANGE UNBOUNDED preceding) as cnt2
    FROM Orders;

//Calculate the count and total number of the recent four records (in proctime).
insert into temp SELECT name,
    count(amount) OVER (PARTITION BY name ORDER BY proctime ROWS BETWEEN 4 PRECEDING AND
```

```
CURRENT ROW) as cnt1,
   sum(amount) OVER (PARTITION BY name ORDER BY proctime ROWS BETWEEN 4 PRECEDING AND
CURRENT ROW) as cnt2
   FROM Orders;

//Calculate the count and total number last 60s (in eventtime). Process the events based on event time,
which is the timeattr field in Orders.
insert into temp SELECT name,
   count(amount) OVER (PARTITION BY name ORDER BY timeattr RANGE BETWEEN INTERVAL '60'
SECOND PRECEDING AND CURRENT ROW) as cnt1,
   sum(amount) OVER (PARTITION BY name ORDER BY timeattr RANGE BETWEEN INTERVAL '60' SECOND
PRECEDING AND CURRENT ROW) as cnt2
   FROM Orders;
```

2.15 JOIN Between Stream Data and Table Data

The JOIN operation allows you to query data from a table and write the query result to the sink stream. Currently, only RDSs and DCS Redis tables are supported. The ON keyword describes the Key used for data query and then writes the **Value** field to the sink stream.

For details about the data definition statements of RDS tables, see **Creating an RDS Table**.

For details about the data definition statements of Redis tables, see **Creating a Redis Table**.

Syntax

```
FROM tableExpression JOIN tableExpression
  ON value11 = value21 [ AND value12 = value22]
```

Syntax Description

The ON keyword only supports equivalent query of table attributes. If level-2 keys exist (specifically, the Redis value type is HASH), the AND keyword needs to be used to express the equivalent query between Key and Hash Key.

Precautions

None

Example

Perform equivalent JOIN between the vehicle information source stream and the vehicle price table, get the vehicle price data, and write the price data into the vehicle information sink stream.

```
CREATE SOURCE STREAM car_infos (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_detail_type STRING
)
WITH (
  type = "dis",
  region = "",
  channel = "dliinput",
  partition_count = "1",
```

```
  encode = "csv",
  field_delimiter = ","
);

/** Create a data dimension table to connect to the source stream to fulfill field backfill.
  *
  * Reconfigure the following options according to actual conditions:
  * value_type: indicates the value type of the Redis key value. The value can be STRING, HASH, SET, ZSET, or LIST. For the HASH type, you need to specify hash_key_column as the layer-2 primary key. For the SET type, you need to concatenate all queried values using commas (,).
  * key_column: indicates the column name corresponding to the primary key of the dimension table.
  * hash_key_column: indicates the column name corresponding to the KEY of the HASHMAP when value_type is HASH. If value_type is not HASH, you do not need to set this option.
  * cluster_address: indicates the DCS Redis cluster address.
  * password: indicates the DCS Redis cluster password.
  **/
CREATE TABLE car_price_table (
  car_brand STRING,
  car_detail_type STRING,
  car_price STRING
)
WITH (
  type = "dcs_redis",
  value_type = "hash",
  key_column = "car_brand",
  hash_key_column = "car_detail_type",
  cluster_address = "192.168.1.238:6379",
  password = "xxxxxxxx"
);

CREATE SINK STREAM audi_car_owner_info (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_detail_type STRING,
  car_price STRING
)
WITH (
  type = "dis",
  region = "",
  channel = "dlioutput",
  partition_key = "car_owner",
  encode = "csv",
  field_delimiter = ","
);

INSERT INTO audi_car_owner_info
SELECT t1.car_id, t1.car_owner, t2.car_brand, t1.car_detail_type, t2.car_price
FROM car_infos as t1 join car_price_table as t2
ON t2.car_brand = t1.car_brand and t2.car_detail_type = t1.car_detail_type
WHERE t1.car_brand = "audi";
```

2.16 Configuring Time Models

Flink provides two time models: processing time and event time.

DLI allows you to specify the time model during creation of the source stream and temporary stream.

Configuring Processing Time

Processing time refers to the system time, which is irrelevant to the data timestamp.

Syntax

```
CREATE SOURCE STREAM stream_name(...) WITH (...)
TIMESTAMP BY proctime.proctime;
CREATE TEMP STREAM stream_name(...)
TIMESTAMP BY proctime.proctime;
```

Description

To set the processing time, you only need to add proctime.proctime following TIMESTAMP BY. You can directly use the proctime field later.

Precautions

None

Example

```
CREATE SOURCE STREAM student_scores (
  student_number STRING, /* Student ID */
  student_name STRING, /* Name */
  subject STRING, /* Subject */
  score INT /* Score */
)
WITH (
  type = "dis",
  region = "",
  channel = "dliinput",
  partition_count = "1",
  encode = "csv",
  field_delimiter=","
)TIMESTAMP BY proctime.proctime;

INSERT INTO score_greate_90
SELECT student_name, sum(score) over (order by proctime RANGE UNBOUNDED PRECEDING)
FROM student_scores;
```

Configuring Event Time

Event Time refers to the time when an event is generated, that is, the timestamp generated during data generation.

Syntax

```
CREATE SOURCE STREAM stream_name(...) WITH (...)
TIMESTAMP BY {attr_name}.rowtime
SET WATERMARK (RANGE {time_interval} | ROWS {literal}, {time_interval});
```

Description

To set the event time, you need to select a certain attribute in the stream as the timestamp and set the watermark policy.

Out-of-order events or late events may occur due to network faults. The watermark must be configured to trigger the window for calculation after waiting for a certain period of time. Watermarks are mainly used to process out-of-order data before generated events are sent to DLI during stream processing.

The following two watermark policies are available:

- By time interval
  ```
  SET WATERMARK(range interval {time_unit}, interval {time_unit})
  ```
- By event quantity
  ```
  SET WATERMARK(rows literal, interval {time_unit})
  ```

NOTE

Parameters are separated by commas (,). The first parameter indicates the watermark sending interval and the second indicates the maximum event delay.

Precautions

None

Example

- Send a watermark every 10s the **time2** event is generated. The maximum event latency is 20s.

```
CREATE SOURCE STREAM student_scores (
  student_number STRING, /* Student ID */
  student_name STRING, /* Name */
  subject STRING, /* Subject */
  score INT, /* Score */
  time2 TIMESTAMP
)
WITH (
  type = "dis",
  region = "",
  channel = "dliinput",
  partition_count = "1",
  encode = "csv",
  field_delimiter=","
)
TIMESTAMP BY time2.rowtime
SET WATERMARK (RANGE interval 10 second, interval 20 second);

INSERT INTO score_greate_90
SELECT student_name, sum(score) over (order by time2 RANGE UNBOUNDED PRECEDING)
FROM student_scores;
```

- Send the watermark every time when 10 pieces of data are received, and the maximum event latency is 20s.

```
CREATE SOURCE STREAM student_scores (
  student_number STRING, /* Student ID */
  student_name STRING, /* Name */
  subject STRING, /* Subject */
  score INT, /* Score */
  time2 TIMESTAMP
)
WITH (
  type = "dis",
  region = "",
  channel = "dliinput",
  partition_count = "1",
  encode = "csv",
  field_delimiter=","
)
TIMESTAMP BY time2.rowtime
SET WATERMARK (ROWS 10, interval 20 second);

INSERT INTO score_greate_90
SELECT student_name, sum(score) over (order by time2 RANGE UNBOUNDED PRECEDING)
FROM student_scores;
```

2.17 Pattern Matching

Complex event processing (CEP) is used to detect complex patterns in endless data streams so as to identify and search patterns in various data rows. Pattern matching is a powerful aid to complex event handling.

CEP is used in a collection of event-driven business processes, such as abnormal behavior detection in secure applications and the pattern of searching for prices, transaction volume, and other behavior in financial applications. It also applies to fraud detection and sensor data analysis.

Syntax

```
MATCH_RECOGNIZE (
    [ PARTITION BY expression [, expression ]* ]
    [ ORDER BY orderItem [, orderItem ]* ]
    [ MEASURES measureColumn [, measureColumn ]* ]
    [ ONE ROW PER MATCH | ALL ROWS PER MATCH ]
    [ AFTER MATCH
        ( SKIP TO NEXT ROW
        | SKIP PAST LAST ROW
        | SKIP TO FIRST variable
        | SKIP TO LAST variable
        | SKIP TO variable )
    ]
    PATTERN ( pattern )
    [ WITHIN intervalLiteral ]
    DEFINE variable AS condition [, variable AS condition ]*
) MR
```

NOTE

Pattern matching in SQL is performed using the MATCH_RECOGNIZE clause. MATCH_RECOGNIZE enables you to do the following tasks:

- Logically partition and order the data that is used in the MATCH_RECOGNIZE clause with its PARTITION BY and ORDER BY clauses.
- Define patterns of rows to seek using the PATTERN clause of the MATCH_RECOGNIZE clause. These patterns use regular expression syntax.
- Specify the logical conditions required to map a row to a row pattern variable in the DEFINE clause.
- Define measures, which are expressions usable in other parts of the SQL query, in the MEASURES clause.

Syntax description

Table 2-71 Syntax description

Parameter	Mandatory	Description
PARTITION BY	No	Logically divides the rows into groups.
ORDER BY	No	Logically orders the rows in a partition.

<table>
<tr><th>Parameter</th><th>Mandatory</th><th>Description</th></tr>
<tr><td>[ONE ROW | ALL ROWS] PER MATCH</td><td>No</td><td>Chooses summaries or details for each match.
ONE ROW PER MATCH: Each match produces one summary row.
ALL ROWS PER MATCH: A match spanning multiple rows will produce one output row for each row in the match.
The following provides an example:
<pre>SELECT * FROM MyTable MATCH_RECOGNIZE
(
  MEASURES AVG(B.id) as Bid
  ALL ROWS PER MATCH
  PATTERN (A B C)
  DEFINE
    A AS A.name = 'a',
    B AS B.name = 'b',
    C as C.name = 'c'
) MR</pre>
Example description
Assume that the format of MyTable is (id, name) and there are three data records: (1, a), (2, b), and (3, c).
ONE ROW PER MATCH outputs the average value 2 of B.
ALL ROWS PER MATCH outputs each record and the average value of B, specifically, (1,a, null), (2,b,2), (3,c,2).</td></tr>
<tr><td>MEASURES</td><td>No</td><td>Defines calculations for export from the pattern matching.</td></tr>
</table>

Parameter	Mandatory	Description
PATTERN	Yes	Defines the row pattern that will be matched. ● PATTERN (A B C) indicates to detect concatenated events A, B, and C. ● PATTERN (A | B) indicates to detect A or B. ● Modifiers – *: 0 or more iterations. For example, A* indicates to match A for 0 or more times. – +: 1 or more iterations. For example, A+ indicates to match A for 1 or more times. – ? : 0 or 1 iteration. For example, A? indicates to match A for 0 times or once. – {n}: *n* iterations (*n* > 0). For example, A{5} indicates to match A for five times. – {n,}: *n* or more iterations (*n* ≥ 0). For example, A{5,} indicates to match A for five or more times. – {n, m}: between *n* and *m* (inclusive) iterations (0 ≤ *n* ≤ *m*, 0 < *m*). For example, A{3,6} indicates to match A for 3 to 6 times. – {, m}: between 0 and *m* (inclusive) iterations (*m* > 0). For example, A{,4} indicates to match A for 0 to 4 times.
DEFINE	Yes	Defines primary pattern variables.
AFTER MATCH SKIP	No	Defines where to restart the matching process after a match is found. ● SKIP TO NEXT ROW: Resumes pattern matching at the row after the first row of the current match. ● SKIP PAST LAST ROW: Resumes pattern matching at the next row after the last row of the current match. ● SKIP TO FIRST variable: Resumes pattern matching at the first row that is mapped to the pattern variable. ● SKIP TO LAST variable: Resumes pattern matching at the last row that is mapped to the pattern variable. ● SKIP TO variable: Same as SKIP TO LAST variable.

Functions Supported by CEP

Table 2-72 Function description

Function	Description
MATCH_NUMBER()	Finds which rows are in which match. It can be used in the MEASURES and DEFINE clauses.
CLASSIFIER()	Finds which pattern variable applies to which rows. It can be used in the MEASURES and DEFINE clauses.
FIRST()/LAST()	FIRST returns the value of an expression evaluated in the first row of the group of rows mapped to a pattern variable. LAST returns the value of an expression evaluated in the last row of the group of rows mapped to a pattern variable. In PATTERN (A B+ C), FIRST (B.id) indicates the ID of the first B in the match, and LAST (B.id) indicates the ID of the last B in the match.
NEXT()/PREV()	Relative offset, which can be used in DEFINE. For example, PATTERN (A B+) DEFINE B AS B.price > PREV(B.price)
RUNNING/ FINAL	RUNNING indicates to match the middle value, while FINAL indicates to match the final result value. Generally, RUNNING/ FINAL is valid only in ALL ROWS PER MATCH. For example, if there are three records (a, 2), (b, 6), and (c, 12), then the values of RUNNING AVG (A.price) and FINAL AVG (A.price) are (2,6), (4,6), (6,6).
Aggregate functions (COUNT, SUM, AVG, MAX, MIN)	Aggregation operations. These functions can be used in the MEASURES and DEFINE clauses. For details, see **Aggregate Functions**.

Example

- Fake plate vehicle detection

CEP conducts pattern matching based on license plate switchover features on the data of vehicles collected by cameras installed on urban roads or high-speed roads in different areas within 5 minutes.

```
INSERT INTO fake_licensed_car
SELECT * FROM camera_license_data MATCH_RECOGNIZE
(
  PARTITION BY car_license_number
  ORDER BY proctime
  MEASURES A.car_license_number as car_license_number, A.camera_zone_number as first_zone, 
B.camera_zone_number as second_zone
  ONE ROW PER MATCH
  AFTER MATCH SKIP TO LAST C
  PATTERN (A B+ C)
  WITHIN interval '5' minute
  DEFINE
    B AS B.camera_zone_number <> A.camera_zone_number,
```

```
    C AS C.camera_zone_number = A.camera_zone_number
) MR;
```

According to this rule, if a vehicle of a license plate number drives from area A to area B but another vehicle of the same license plate number is detected in area A within 5 minutes, then the vehicle in area A is considered to carry a fake license plate.

Input data:

```
Zhejiang B88888, zone_A
Zhejiang AZ626M, zone_A
Zhejiang B88888, zone_A
Zhejiang AZ626M, zone_A
Zhejiang AZ626M, zone_A
Zhejiang B88888, zone_B
Zhejiang B88888, zone_B
Zhejiang AZ626M, zone_B
Zhejiang AZ626M, zone_B
Zhejiang AZ626M, zone_C
Zhejiang B88888, zone_A
Zhejiang B88888, zone_A
```

The output is as follows:

```
Zhejiang B88888, zone_A, zone_B
```

2.18 StreamingML

2.18.1 Anomaly Detection

Anomaly detection applies to various scenarios, including intrusion detection, financial fraud detection, sensor data monitoring, medical diagnosis, natural data detection, and more. The typical algorithms for anomaly detection include the statistical modeling method, distance-based calculation method, linear model, and nonlinear model.

DLI uses an anomaly detection method based on the random forest, which has the following characteristics:

- The one-pass algorithm is used with O(1) amortized time complexity and O(1) space complexity.
- The random forest structure is constructed only once. The model update operation only updates the node data distribution values.
- The node stores data distribution information of multiple windows, and the algorithm can detect data distribution changes.
- Anomaly detection and model updates are completed in the same code framework.

Syntax

```
SRF_UNSUP(ARRAY[Field 1, Field 2, ...], 'Optional parameter list')
```

NOTE

- The anomaly score returned by the function is a DOUBLE value in the range of [0, 1].
- The field names must be of the same type. If the field types are different, you can use the CAST function to escape the field names, for example, [a, CAST(b as DOUBLE)].
- The syntax of the optional parameter list is as follows: "key1=value,key2=value2,..."

Parameter Description

Table 2-73 Parameter Description

Parameter	Mandatory	Description	Default Value
transientThreshold	No	Threshold for which the histogram change is indicating a change in the data.	5
numTrees	No	Number of trees composing the random forest.	15
maxLeafCount	No	Maximum number of leaf nodes one tree can have.	15
maxTreeHeight	No	Maximum height of the tree.	12
seed	No	Random seed value used by the algorithm.	4010
numClusters	No	Number of types of data to be detected. By default, the following two data types are available: anomalous and normal data.	2
dataViewMode	No	Algorithm learning mode. • Value **history** indicates that all historical data is considered. • Value **horizon** indicates that only historical data of a recent time period (typically a size of 4 windows) is considered.	history

Example

Anomaly detection is conducted on the **c** field in data stream **MyTable**. If the anomaly score is greater than 0.8, then the detection result is considered to be anomaly.

```
SELECT c,
   CASE WHEN SRF_UNSUP(ARRAY[c], "numTrees=15,seed=4010") OVER (ORDER BY proctime RANGE
BETWEEN INTERVAL '99' SECOND PRECEDING AND CURRENT ROW) > 0.8
        THEN 'anomaly'
        ELSE 'not anomaly'
```

```
  END
FROM MyTable
```

2.18.2 Time Series Forecasting

Modeling and forecasting time series is a common task in many business verticals. Modeling is used to extract meaningful statistics and other characteristics of the data. Forecasting is the use of a model to predict future data. DLI provides a series of stochastic linear models to help users conduct online modeling and forecasting in real time.

ARIMA (Non-Seasonal)

Auto-Regressive Integrated Moving Average (ARIMA) is a classical model used for time series forecasting and is closely correlated with the AR, MA, and ARMA models.

- The AR, MA, and ARMA models are applicable to **stationary** sequences.
  - AR(p) is an autoregressive model. An AR(p) is a linear combination of p consecutive values from immediate past. The model can predict the next value by using the weight of linear combination.
  - MA(q) is a moving average model. An MA(q) is a linear combination of q white noise values from the past plus the average value. The model can also predict the next value by using the weight of linear combination.
  - ARMA(p, q) is an autoregressive moving average model, which integrates the advantages of both AR and MA models. In the ARMA model, the autoregressive process is responsible for quantizing the relationship between the current data and the previous data, and the moving average process is responsible for solving problems of random variables. Therefore, the ARMA model is more effective than AR/MA.
- ARIMA is suitable for **non-stationary** series. In ARIMA(p, q, d), **p** indicates the autoregressive order, **q** indicates the moving average order, and **d** indicates the difference order.

Syntax

```
AR_PRED(field, degree): Use the AR model to forecast new data.
AR_COEF(field, degree): Return the weight of the AR model.
ARMA_PRED(field, degree): Use the ARMA model to forecast new data.
ARMA_COEF(field, degree): Return the weight of the ARMA model.
ARIMA_PRED(field, degree, derivativeOrder): Use ARIMA to forecast new data.
```

Table 2-74 Parameter Description

Parameter	Mandatory	Description	Default Value
field	Yes	Name of the field, data in which is used for prediction, in the data stream.	-
degree	No	Defines how many steps in the past are going to be considered for the next prediction. Currently, only "p = q = degree" is allowed.	5

Parameter	Mandatory	Description	Default Value
derivativeOrder	No	Derivative order. Generally, this parameter is set to **1** or **2**.	1

Example

Separately use AR, ARMA, and ARIMA to forecast the time series ordered by rowtime.

```
SELECT b,
   AR_PRED(b) OVER (ORDER BY rowtime ROWS  BETWEEN 5 PRECEDING AND CURRENT ROW) AS ar,
   ARMA_PRED(b) OVER (ORDER BY rowtime ROWS  BETWEEN 5 PRECEDING AND CURRENT ROW) AS
arma,
   ARIMA_PRED(b) OVER (ORDER BY rowtime ROWS  BETWEEN 5 PRECEDING AND CURRENT ROW) AS
arima
FROM MyTable
```

Holt Winters

The Holt-Winters algorithm is one of the Exponential smoothing methods used to forecast **seasonal** data in time series.

Syntax

```
HOLT_WINTERS(field, seasonality, forecastOrder)
```

Table 2-75 Parameter Description

Parameter	Mandatory	Description
field	Yes	Name of the field, data in which is used for prediction, in the data stream.
seasonality	Yes	Seasonality space used to perform the prediction. For example, if data samples are collected daily, and the season space to consider is a week, then **seasonality** is **7**.
forecastOrder	No	Value to be forecast, specifically, the number of steps to be considered in the future for producing the forecast. If **forecastOrder** is set to **1**, the algorithm forecasts the next value. If **forecastOrder** is set to **2**, the algorithm forecasts the value of 2 steps ahead in the future. The default value is **1**. When using this parameter, ensure that the OVER window size is greater than the value of this parameter.

Example

Use Holt-Winters to forecast time series ordered by rowtime.

```
SELECT b,
   HOLT_WINTERS(b, 5) OVER (ORDER BY rowtime ROWS  BETWEEN 5 PRECEDING AND CURRENT ROW)
AS a1,
   HOLT_WINTERS(b, 5, 2) OVER (ORDER BY rowtime ROWS  BETWEEN 5 PRECEDING AND CURRENT
ROW) AS a2
FROM MyTable
```

2.18.3 Real-Time Clustering

Clustering algorithms belong to unsupervised algorithms. K-Means, a clustering algorithm, partitions data points into related clusters by calculating the distance between data points based on the predefined cluster quantity. For offline static datasets, we can determine the clusters based on field knowledge and run K-Means to achieve a better clustering effect. However, online real-time streaming data is always changing and evolving, and the cluster quantity is likely to change. To address clustering issues on online real-time streaming data, DLI provides a low-delay online clustering algorithm that does not require predefined cluster quantity.

The algorithm works as follows: Given a distance function, if the distance between two data points is less than a threshold, both data points will be partitioned into the same cluster. If the distances between a data point and the central data points in several cluster centers are less than the threshold, then related clusters will be merged. When data in a data stream arrives, the algorithm computes the distances between each data point and the central data points of all clusters to determine whether the data point can be partitioned into to an existing or new cluster.

Syntax

```
CENTROID(ARRAY[field_names], distance_threshold): Compute the centroid of the cluster where the
current data point is assigned.
CLUSTER_CENTROIDS(ARRAY[field_names], distance_threshold): Compute all centroids after the data point
is assigned.
ALL_POINTS_OF_CLUSTER(ARRAY[field_names], distance_threshold): Compute all data points in the cluster
where the current data point is assigned.
ALL_CLUSTERS_POINTS(ARRAY[field_names], distance_threshold): Computers all data points in each cluster
after the current data point is assigned.
```

NOTE

- Clustering algorithms can be applied in **unbounded streams**.

Parameter Description

Table 2-76 Parameter Description

Parameter	Mandatory	Description
field_names	Yes	Name of the field where the data is located in the data stream. Multiple fields are separated by commas (,). For example, **ARRAY[a, b, c]**.
distance_threshold	Yes	Distance threshold. When the distance between two data points is less than the threshold, both data points are placed in the same cluster.

Example

Use four functions to compute information related to clusters over windows.

```
SELECT
  CENTROID(ARRAY[c,e], 1.0) OVER (ORDER BY proctime RANGE UNBOUNDED PRECEDING) AS centroid,
  CLUSTER_CENTROIDS(ARRAY[c,e], 1.0) OVER (ORDER BY proctime RANGE UNBOUNDED PRECEDING) AS
centroids
FROM MyTable

SELECT
  CENTROID(ARRAY[c,e], 1.0) OVER (ORDER BY proctime RANGE BETWEEN INTERVAL '60' MINUTE
PRECEDING AND CURRENT ROW) AS centroidCE,
  ALL_POINTS_OF_CLUSTER(ARRAY[c,e], 1.0) OVER (ORDER BY proctime RANGE BETWEEN INTERVAL '60'
MINUTE PRECEDING AND CURRENT ROW) AS itemList,
  ALL_CLUSTERS_POINTS(ARRAY[c,e], 1.0) OVER (ORDER BY proctime RANGE  BETWEEN INTERVAL '60'
MINUTE PRECEDING AND CURRENT ROW) AS listoflistofpoints
FROM MyTable
```

2.18.4 Deep Learning Model Prediction

Deep learning has a wide range of applications in many industries, such as image classification, image recognition, and speech recognition. DLI provides several functions to load deep learning models for prediction.

Currently, models DeepLearning4j and Keras are supported. In Keras, TensorFlow, CNTK, or Theano can serve as the backend engine. With importing of the neural network model from Keras, models of mainstream learning frameworks such as Theano, TensorFlow, Caffe, and CNTK can be imported.

Syntax

```
-- Image classification: returns the predicted category IDs used for image classification.
DL_IMAGE_MAX_PREDICTION_INDEX(field_name, model_path, is_dl4j_model)
DL_IMAGE_MAX_PREDICTION_INDEX(field_name, keras_model_config_path, keras_weights_path) --
Suitable for the Keras model

--Text classification: returns the predicted category IDs used for text classification.
DL_TEXT_MAX_PREDICTION_INDEX(field_name, model_path, is_dl4j_model) -- Use the default word2vec
model.
DL_TEXT_MAX_PREDICTION_INDEX(field_name, word2vec_path, model_path, is_dl4j_model)
```

NOTE

Models and configuration files must be stored on OBS. The path format is obs://**your_ak**:**your_sk**@obs.**your_obs_region**.xxx.com:443/**your_model_path**.

Parameter Description

Table 2-77 Parameter description

Parameter	Mandatory	Description
field_name	Yes	Name of the field, data in which is used for prediction, in the data stream. In image classification, this parameter needs to declare ARRAY[TINYINT]. In image classification, this parameter needs to declare String.
model_path	Yes	Complete save path of the model on OBS, including the model structure and model weight.
is_dl4j_model	Yes	Whether the model is a Deeplearning4j model Value **true** indicates that the model is a Deeplearning4j model, while value **false** indicates that the model is a Keras model.
keras_model_config_path	Yes	Complete save path of the model structure on OBS. In Keras, you can obtain the model structure by using **model.to_json()**.
keras_weights_path	Yes	Complete save path of the model weight on OBS. In Keras, you can obtain the model weight by using **model.save_weights(filepath)**.
word2vec_path	Yes	Complete save path of the word2vec model on OBS.

Example

For prediction in image classification, use the Mnist dataset as the input and load the pre-trained Deeplearning4j model or Keras model to predict the digit representing each image in real time.

```
CREATE SOURCE STREAM Mnist(
    image Array[TINYINT]
)
SELECT DL_IMAGE_MAX_PREDICTION_INDEX(image, 'your_dl4j_model_path', false) FROM Mnist
SELECT DL_IMAGE_MAX_PREDICTION_INDEX(image, 'your_keras_model_path', true) FROM Mnist
SELECT DL_IMAGE_MAX_PREDICTION_INDEX(image, 'your_keras_model_config_path', 'keras_weights_path')
FROM Mnist
```

For prediction in text classification, use data of a group of news titles as the input and load the pre-trained Deeplearning4j model or Keras model to predict the category of each news title in real time, such as economy, sports, and entertainment.

```
CREATE SOURCE STREAM News(
    title String
)
SELECT DL_TEXT_MAX_PREDICTION_INDEX(title, 'your_dl4j_word2vec_model_path','your_dl4j_model_path', false) FROM News
SELECT DL_TEXT_MAX_PREDICTION_INDEX(title,
'your_keras_word2vec_model_path','your_keras_model_path', true) FROM News
SELECT DL_TEXT_MAX_PREDICTION_INDEX(title, 'your_dl4j_model_path', false) FROM New
SELECT DL_TEXT_MAX_PREDICTION_INDEX(title, 'your_keras_model_path', true) FROM New
```

2.19 Reserved Keywords

Flink SQL reserves some strings as keywords. If you want to use the following character strings as field names, ensure that they are enclosed by back quotes, for example, `value` and `count`.

A

- A
- ABS
- ABSOLUTE
- ACTION
- ADA
- ADD
- ADMIN
- AFTER
- AK
- ALL
- ALLOCATE
- ALLOW
- ALTER
- ALWAYS
- AND
- ANY
- APPEND
- APP_ID
- ARE
- ARRAY
- ARRAY_BRACKET
- AS
- ASC
- ASENSITIVE
- ASSERTION
- ASSIGNMENT

- ASYMMETRIC
- AT
- AT_LEAST_ONCE
- ATOMIC
- ATTRIBUTE
- ATTRIBUTES
- AUTHORIZATION
- AVG
- AVRO_CONFIG
- AVRO_DATA
- AVRO_SCHEMA

B

- BATCH_INSERT_DATA_NUM
- BEFORE
- BEGIN
- BERNOULLI
- BETWEEN
- BIGINT
- BINARY
- BIT
- BLOB
- BOOL
- BOOLEAN
- BOTH
- BREADTH
- BUCKET
- BY

C

- C
- CACHE_MAX_NUM
- CACHE_TIME
- CALL
- CALLED
- CARDINALITY
- CASCADE
- CASCADED
- CASE
- CAST
- CATALOG

- CATALOG_NAME
- CEIL
- CEILING
- CENTURY
- CHAIN
- CHANNEL
- CHAR
- CHARACTER
- CHARACTERISTICTS
- CHARACTERS
- CHARACTER_LENGTH
- CHARACTER_SET_CATALOG
- CHARACTER_SET_NAME
- CHARACTER_SET_SCHEMA
- CHAR_LENGTH
- CHECK
- CHECKPOINT_APP_NAME
- CHECKPOINT_INTERVAL
- CHECKPOINTINTERVAL
- CLASS_ORIGIN
- CLOB
- CLOSE
- CLUSTER_ADDRESS
- CLUSTER_ID
- CLUSTER_NAME
- COALESCE
- COBOL
- COLLATE
- COLLATION
- COLLATION_CATALOG
- COLLATION_NAME
- COLLATION_SCHEMA
- COLLECT
- COLUMN
- COLUMN_NAME
- COLUMN_NAME_MAP
- COMMAND_FUNCTION
- COMMAND_FUNCTION_CODE
- COMMIT
- COMMITTED

- CONDITION
- CONDITION_NUMBER
- CONFIGURATION
- CONFLUENT_CERTIFICATE_NAME
- CONFLUENT_PROPERTIES
- CONFLUENT_SCHEMA_FIELD
- CONFLUENT_URL
- CONNECT
- CONNECTION_NAME
- CONSTRAINT
- CONSTRAINTS
- CONSTRAINT_CATALOG
- CONSTRAINT_NAME
- CONSTRAINT_SCHEMA
- CONSTRUCTOR
- CONTAINS
- CONTINUE
- CONVERT
- CORR
- CORRESPONDING
- COUNT
- COVAR_POP
- COVAR_SAMP
- CREATE
- CREATE_IF_NOT_EXIST
- CROSS
- CUBE
- CUME_DIST
- CURRENT
- CURRENT_CATALOG
- CURRENT_DATE
- CURRENT_DEFAULT_TRANSFORM_GROUP
- CURRENT_PATH
- CURRENT_ROLE
- CURRENT_SCHEMA
- CURRENT_TIMESTAMP
- CURRENT_TRANSFORM_GROUP_FOR_TYPE
- CURRENT_USER
- CURSOR
- CURSOR_NAME

- CYCLE

D

- DATE
- DATABASE
- DATE
- DATETIME_INTERVAL_CODE
- DATETIME_INTERVAL_PRECISION
- DAY
- DB_COLUMNS
- DB_URL
- DB_OBS_SERVER
- DB_TYPE
- DEALLOCATE
- DEC
- DECADE
- DECIMAL
- DECLARE
- DEFAULTS
- DEFERRABLE
- DEFERRED
- DEFINER
- DEGREE
- DELETE
- DELETE_OBS_TEMP_FILE
- DENSE_RANK
- DEPTH
- DEREF
- DERIVED
- DESC
- DESCRIBE
- DESCRIPTION
- DESCRIPTOR
- DETERMINISTIC
- DIAGNOSTICS
- DISALLOW
- DISCONNECT
- DIS_NOTICE_CHANNEL
- DISPATCH
- DISTINCT

- DOMAIN
- DOUBLE
- DOW
- DOY
- DRIVER
- DROP
- DUMP_INTERVAL
- DYNAMIC
- DYNAMIC_FUNCTION
- DYNAMIC_FUNCTION_CODE

E

- EACH
- ELEMENT
- ELSE
- EMAIL_KEY
- ENABLECHECKPOINT
- ENABLE_CHECKPOINT
- ENABLE_OUTPUT_NULL
- ENCODE
- ENCODE_CLASS_NAME
- ENCODE_CLASS_PARAMETER
- ENCODED_DATA
- END
- ENDPOINT
- END_EXEC
- EPOCH
- EQUALS
- ESCAPE
- ES_FIELDS
- ES_INDEX
- ES_TYPE
- ESTIMATEMEM
- ESTIMATEPARALLELISM
- EXACTLY_ONCE
- EXCEPT
- EXCEPTION
- EXCLUDE
- EXCLUDING
- EXEC

- EXECUTE
- EXISTS
- EXP
- EXPLAIN
- EXTEND
- EXTERNAL
- EXTRACT
- EVERY

F

- FALSE
- FETCH
- FIELD_DELIMITER
- FIELD_NAMES
- FILE_PREFIX
- FILTER
- FINAL
- FIRST
- FIRST_VALUE
- FLOAT
- FLOOR
- FOLLOWING
- FOR
- FUNCTION
- FOREIGN
- FORTRAN
- FOUND
- FRAC_SECOND
- FREE
- FROM
- FULL
- FUSION

G

- G
- GENERAL
- GENERATED
- GET
- GLOBAL
- GO
- GOTO

- GRANT
- GRANTED
- GROUP
- GROUPING
- GW_URL

H

- HASH_KEY_COLUMN
- HAVING
- HIERARCHY
- HOLD
- HOUR
- HTTPS_PORT

I

- IDENTITY
- ILLEGAL_DATA_TABLE
- IMMEDIATE
- IMPLEMENTATION
- IMPORT
- IN
- INCLUDING
- INCREMENT
- INDICATOR
- INITIALLY
- INNER
- INOUT
- INPUT
- INSENSITIVE
- INSERT
- INSTANCE
- INSTANTIABLE
- INT
- INTEGER
- INTERSECT
- INTERSECTION
- INTERVAL
- INTO
- INVOKER
- IN_WITH_SCHEMA
- IS

- ISOLATION

J

- JAVA
- JOIN
- JSON_CONFIG
- JSON_SCHEMA

K

- K
- KAFKA_BOOTSTRAP_SERVERS
- KAFKA_CERTIFICATE_NAME
- KAFKA_GROUP_ID
- KAFKA_PROPERTIES
- KAFKA_PROPERTIES_DELIMITER
- KAFKA_TOPIC
- KEY
- KEY_COLUMN
- KEY_MEMBER
- KEY_TYPE
- KEY_VALUE
- KRB_AUTH

L

- LABEL
- LANGUAGE
- LARGE
- LAST
- LAST_VALUE
- LATERAL
- LEADING
- LEFT
- LENGTH
- LEVEL
- LIBRARY
- LIKE
- LIMIT
- LONG

M

- M

- MAP
- MATCH
- MATCHED
- MATCHING_COLUMNS
- MATCHING_REGEX
- MAX
- MAXALLOWEDCPU
- MAXALLOWEDMEM
- MAXALLOWEDPARALLELISM
- MAX_DUMP_FILE_NUM
- MAX_RECORD_NUM_CACHE
- MAX_RECORD_NUM_PER_FILE
- MAXVALUE
- MEMBER
- MERGE
- MESSAGE_COLUMN
- MESSAGE_LENGTH
- MESSAGE_OCTET_LENGTH
- MESSAGE_SUBJECT
- MESSAGE_TEXT
- METHOD
- MICROSECOND
- MILLENNIUM
- MIN
- MINUTE
- MINVALUE
- MOD
- MODIFIES
- MODULE
- MONTH
- MORE
- MS
- MULTISET
- MUMPS

N

- NAME
- NAMES
- NATIONAL
- NATURAL

- NCHAR
- NCLOB
- NESTING
- NEW
- NEXT
- NO
- NONE
- NORMALIZE
- NORMALIZED
- NOT
- NULL
- NULLABLE
- NULLIF
- NULLS
- NUMBER
- NUMERIC

O

- OBJECT
- OBJECT_NAME
- OBS_DIR
- OCTETS
- OCTET_LENGTH
- OF
- OFFSET
- OLD
- ON
- ONLY
- OPEN
- OPERATION_FIELD
- OPTION
- OPTIONS
- OR
- ORDER
- ORDERING
- ORDINALITY
- OTHERS
- OUT
- OUTER
- OUTPUT

- OVER
- OVERLAPS
- OVERLAY
- OVERRIDING

P

- PAD
- PARALLELISM
- PARAMETER
- PARAMETER_MODE
- PARAMETER_NAME
- PARAMETER_ORDINAL_POSITION
- PARAMETER_SPECIFIC_CATALOG
- PARAMETER_SPECIFIC_NAME
- PARAMETER_SPECIFIC_SCHEMA
- PARTIAL
- PARTITION
- PARTITION_COUNT
- PARTITION_KEY
- PARTITION_RANGE
- PASCAL
- PASSTHROUGH
- PASSWORD
- PATH
- PERCENTILE_CONT
- PERCENTILE_DISC
- PERCENT_RANK
- PERSIST_SCHEMA
- PIPELINE_ID
- PLACING
- PLAN
- PLI
- POSITION
- POWER
- PRECEDING
- PRECISION
- PREPARE
- PRESERVE
- PRIMARY
- PRIMARY_KEY

- PRIOR
- PRIVILEGES
- PROCEDURE
- PROCTIME
- PROJECT_ID
- PUBLIC

Q

- QUARTER
- QUOTE

R

- RANGE
- RANK
- RAW
- READ
- READS
- READ_ONCE
- REAL
- RECURSIVE
- REF
- REFERENCES
- REFERENCING
- REGION
- REGR_AVGX
- REGR_AVGY
- REGR_COUNT
- REGR_INTERCEPT
- REGR_R2
- REGR_SLOPE
- REGR_SXX
- REGR_SXY
- REGR_SYY
- RELATIVE
- RELEASE
- REPEATABLE
- RESET
- RESTART
- RESTRICT
- RESULT
- RETURN

- RETURNED_CARDINALITY
- RETURNED_LENGTH
- RETURNED_OCTET_LENGTH
- RETURNED_SQLSTATE
- RETURNS
- REVOKE
- RIGHT
- ROLE
- ROLLBACK
- ROLLING_INTERVAL
- ROLLING_SIZE
- ROLLUP
- ROUTINE
- ROUTINE_CATALOG
- ROUTINE_NAME
- ROUTINE_SCHEMA
- ROW
- ROW_COUNT
- ROW_DELIMITER
- ROW_NUMBER
- ROWS
- ROWTIME

S

- SAVEPOINT
- SCALE
- SCHEMA
- SCHEMA_CASE_SENSITIVE
- SCHEMA_NAME
- SCOPE
- SCOPE_CATALOGS
- SCOPE_NAME
- SCOPE_SCHEMA
- SCROLL
- SEARCH
- SECOND
- SECTION
- SECURITY
- SELECT
- SELF

- SENSITIVE
- SEQUENCE
- SERIALIZABLE
- SERVER
- SERVER_NAME
- SESSION
- SESSION_USER
- SET
- SETS
- SIMILAR
- SIMPLE
- SINK
- SIZE
- SK
- SMALLINT
- SOME
- SOURCE
- SPACE
- SPECIFIC
- SPECIFICTYPE
- SPECIFIC_NAME
- SQL
- SQLEXCEPTION
- SQLSTATE
- SQLWARNING
- SQL_TSI_DAY
- SQL_TSI_FRAC_SECOND
- SQL_TSI_HOUR
- SQL_TSI_MICROSECOND
- SQL_TSI_MINUTE
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_SECOND
- SQL_TSI_WEEK
- SQL_TSI_YEAR
- SQRT
- START
- START_TIME
- STATE
- STATEMENT

- STATIC
- STDDEV_POP
- STDDEV_SAMP
- STREAM
- STRING
- STRUCTURE
- STYLE
- SUBCLASS_ORIGIN
- SUBMULTISET
- SUBSTITUTE
- SUBSTRING
- SUM
- SYMMETRIC
- SYSTEM
- SYSTEM_USER

T

- TABLE
- TABLESAMPLE
- TABLE_COLUMNS
- TABLE_NAME
- TABLE_NAME_MAP
- TEMP
- TEMPORARY
- THEN
- TIES
- TIME
- TIMESTAMP
- TIMESTAMPADD
- TIMESTAMPDIFF
- TIMEZONE_HOUR
- TIMEZONE_MINUTE
- TINYINT
- TO
- TOP_LEVEL_COUNT
- TOPIC
- TOPIC_URN
- TRAILING
- TRANSACTION
- TRANSACTIONAL_TABLE

- TRANSACTIONS_ACTIVE
- TRANSACTIONS_COMMITTED
- TRANSACTIONS_ROLLED_BACK
- TRANSFORM
- TRANSFORMS
- TRANSLATE
- TRANSLATION
- TRANX_ID
- TREAT
- TRIGGER
- TRIGGER_CATALOG
- TRIGGER_NAME
- TRIGGER_SCHEMA
- TRIM
- TRUE
- TSDB_LINK_ADDRESS
- TSDB_METRICS
- TSDB_TIMESTAMPS
- TSDB_TAGS
- TSDB_VALUES
- TYPE
- TYPE_CLASS_NAME
- TYPE_CLASS_PARAMETER

U

- UESCAPE
- UNBOUNDED
- UNCOMMITTED
- UNDER
- UNION
- UNIQUE
- UNKNOWN
- UNNAMED
- UNNEST
- UPDATE
- UPPER
- UPSERT
- URN_COLUMN
- USAGE
- USER

- USER_DEFINED_TYPE_CATALOG
- USER_DEFINED_TYPE_CODE
- USER_DEFINED_TYPE_NAME
- USER_DEFINED_TYPE_SCHEMA
- USERNAME
- USING

V

- VALUE
- VALUES
- VALUE_TYPE
- VARBINARY
- VARCHAR
- VARYING
- VAR_POP
- VAR_SAMP
- VERSION
- VERSION_ID
- VIEW

W

- WATERMARK
- WEEK
- WHEN
- WHENEVER
- WHERE
- WIDTH_BUCKET
- WINDOW
- WITH
- WITHIN
- WITHOUT
- WORK
- WRAPPER
- WRITE

X

- XML
- XML_CONFIG

Y

- YEAR

Z

- ZONE

3 Identifiers

3.1 aggregate_func

Syntax

None.

Description

Aggregate function.

3.2 alias

Syntax

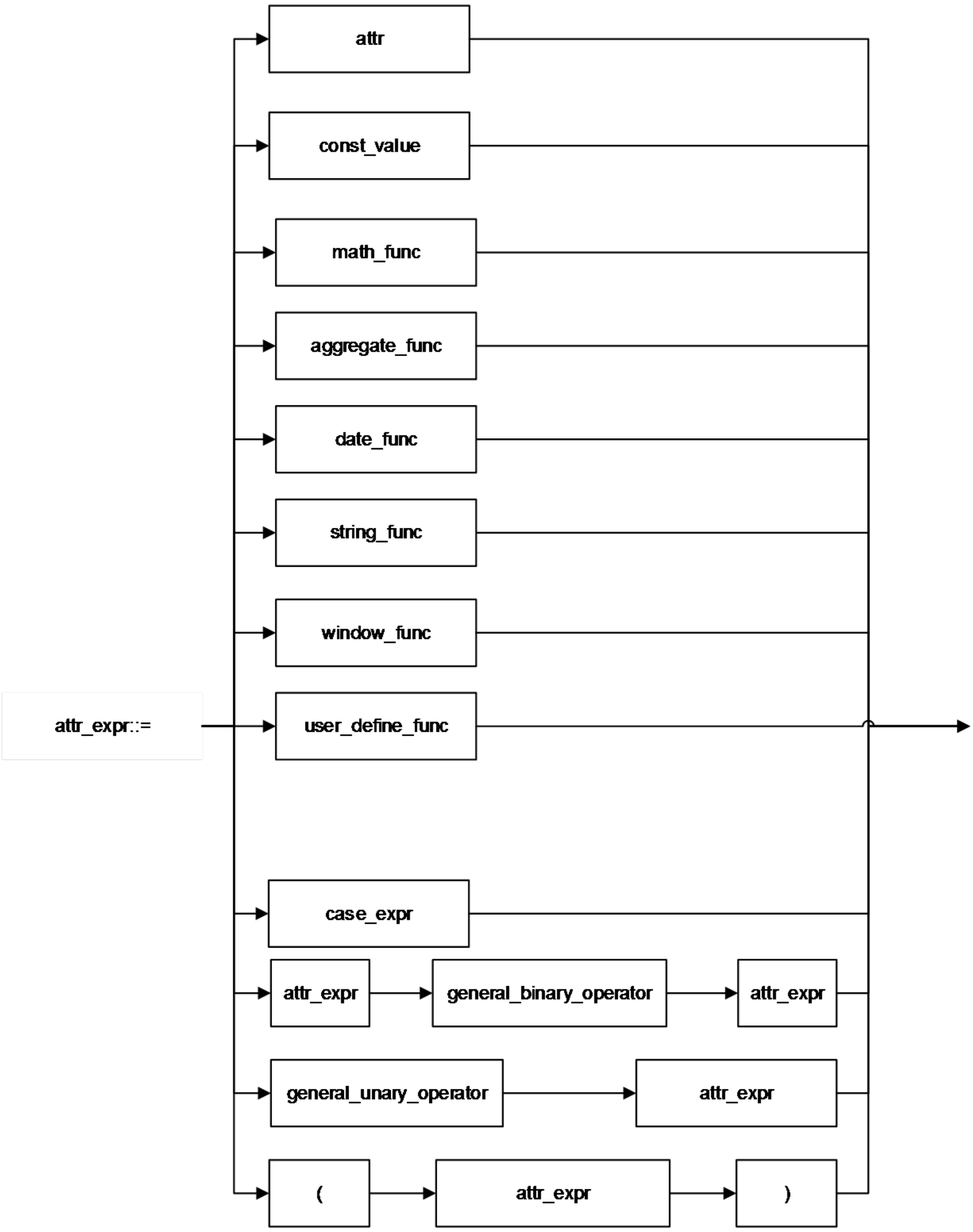
None.

Description

Alias, which must be STRING type. It can be assigned to a field, table, view, or subquery.

3.3 attr_expr

Syntax



Description

Syntax	Description
attr_expr	Attribute expression.

Syntax	Description
attr	Table field, which is the same as col_name.
const_value	Constant value.
case_expr	Case expression.
math_func	Mathematical function.
date_func	Date function.
string_func	String function.
aggregate_func	Aggregate function.
window_func	Analysis window function.
user_define_func	User-defined function.
general_binary_operator	General binary operator.
general_unary_operator	General unary operator.
(	Start of the specified subattribute expression.
)	End of the specified subattribute expression.

3.4 attr_expr_list

Syntax

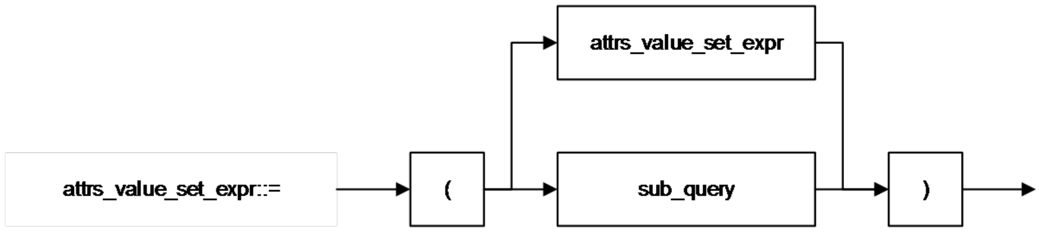
None.

Description

List of attr_expr, which is separated by commas (,).

3.5 attrs_value_set_expr

Syntax



Description

Syntax	Description
attrs_value_set_expr	Collection of attribute values.
sub_query	Subquery clause.
(	Start of the specified subquery expression.
)	End of the specified subquery expression.

3.6 boolean_expression

Syntax

None.

Description

Return a boolean expression.

3.7 col

Syntax

None.

Description

Formal parameter for function call. It is usually a field name, which is the same as **col_name**.

3.8 col_comment

Syntax

None.

Description

Column (field) description, which must be STRING type and cannot exceed 256 bytes.

3.9 col_name

Syntax

None.

Description

Column name, which must be STRING type and cannot exceed 128 bytes.

3.10 col_name_list

Syntax

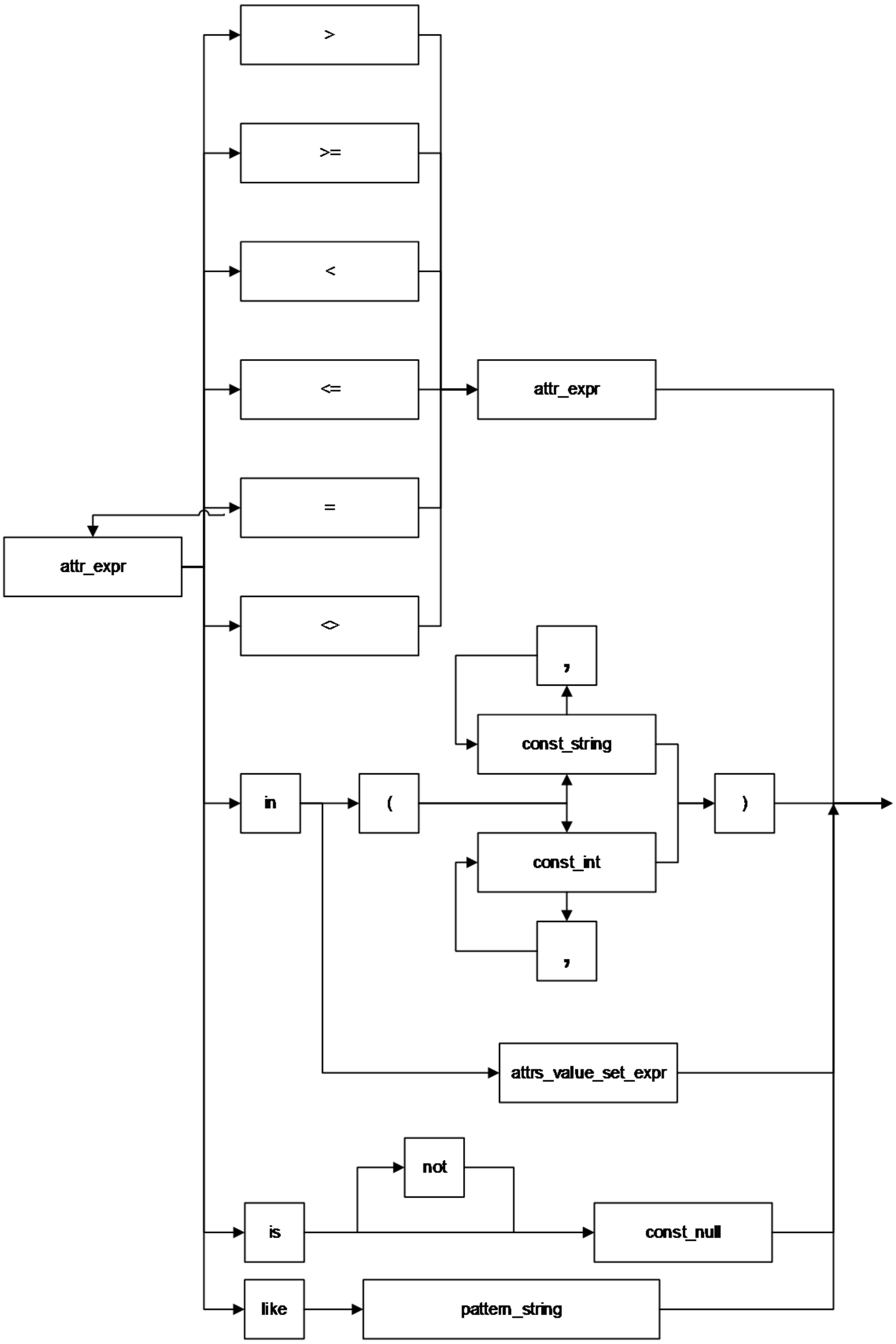
None.

Description

Field list, which consists of one **col_name** or more. If there is more than one **col_name**, separate them by using a comma (,).

3.11 condition

Syntax

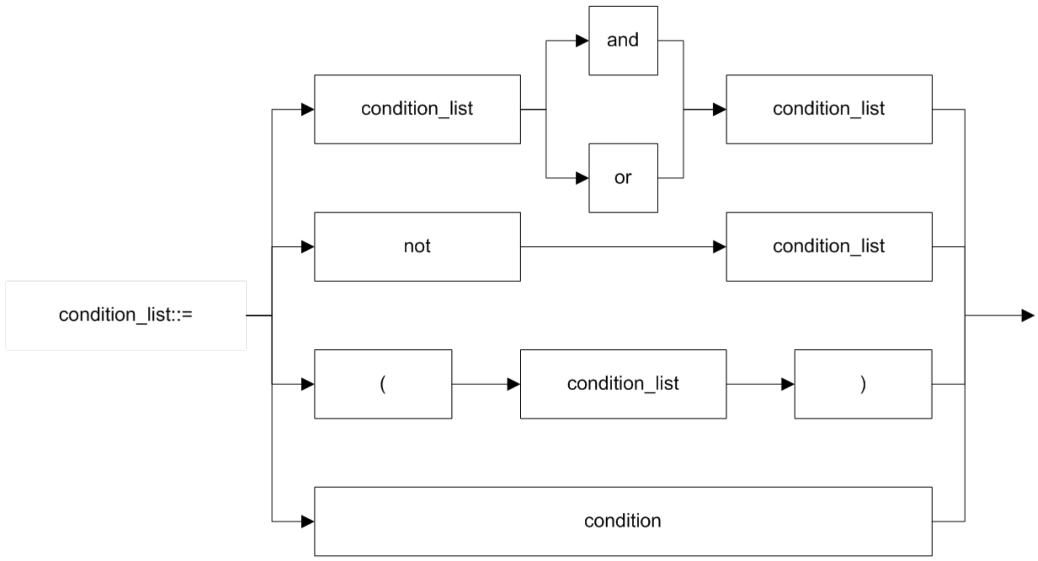


Description

Syntax	Description
condition	Judgment condition.
>	Relational operator: >
>=	Relational operator: ≥
<	Relational operator: <
<=	Relational operator: ≤
=	Relational operator: =
<>	Relational operator: <>
is	Relational operator: is
is not	Relational operator: is not
const_null	Constant value: null
like	Relational operator: used for wildcard matching.
pattern_string	Pattern matching string, which supports wildcard matching. In WHERE LIKE, SQL wildcard characters "**%**" and "_" are supported. "**%**" represents one or more characters. "_" represents only one character.
attr_expr	Attribute expression.
attrs_value_set_expr	Collection of attribute values.
in	Keyword used to determine whether attributes are in the same collection.
const_string	String constant.
const_int	Integer constant.
(	Start of the specified constant collection.
)	End of the specified constant collection.
,	Separator comma (,)

3.12 condition_list

Syntax



Description

Syntax	Description
condition_list	List of judgment conditions.
and	Logical operator: AND
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjudgment condition.
)	End of the subjudgment condition.
condition	Judgment condition.

3.13 cte_name

Syntax

None.

Description

Common expression name.

3.14 data_type

Syntax

None.

Description

Data type. Currently, only the primitive data types are supported.

3.15 db_comment

Syntax

None.

Description

Database description, which must be STRING type and cannot exceed 256 characters.

3.16 db_name

Syntax

None.

Description

Database name, which must be STRING type and cannot exceed 128 bytes.

3.17 else_result_expression

Syntax

None.

Description

Returned result for the **ELSE** clause of the **CASE WHEN** statement.

3.18 file_format

Format

| AVRO

| CSV

| JSON

| ORC

| PARQUET

Description

- Currently, the preceding formats are supported.
- Both **USING** and **STORED AS** can be used for specifying the data format. You can specify the preceding data formats by **USING**, but only the **ORC** and **PARQUET** formats by **STORED AS**.
- **ORC** has optimized **RCFile** to provide an efficient method to store **Hive** data.
- **PARQUET** is an analytical service-oriented and column-based storage format.

3.19 file_path

Syntax

None.

Description

File path, which is the OBS path

3.20 function_name

Syntax

None.

Description

Function name, which must be STRING type.

3.21 groupby_expression

Syntax

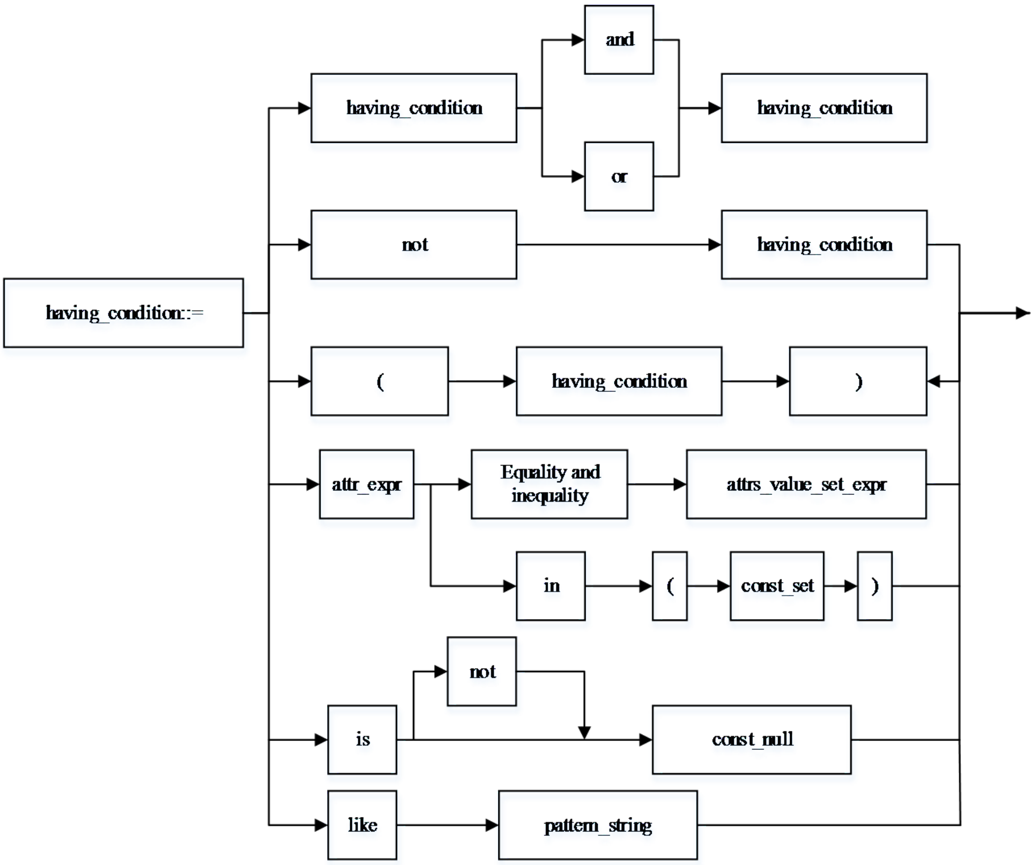
None.

Description

Expression that includes GROUP BY.

3.22 having_condition

Syntax



Description

Syntax	Description
having_condition	Judgment condition of **having**.
and	Logical operator: AND
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjudgment condition.
)	End of the subjudgment condition.
condition	Judgment condition.
const_set	Collection of constants, which are separated by using comma (,).

Syntax	Description
in	Keyword used to determine whether attributes are in the same collection.
attrs_value_set_expr	Collection of attribute values.
attr_expr	Attribute expression.
Equality and inequality	Equation and inequality. For details, see **Relational Operators**.
pattern_string	Pattern matching string, which supports wildcard matching. In WHERE LIKE, SQL wildcard characters "**%**" and "_" are supported. "**%**" represents one or more characters. "_" represents only one character.
like	Relational operator: used for wildcard matching.

3.23 input_expression

Syntax

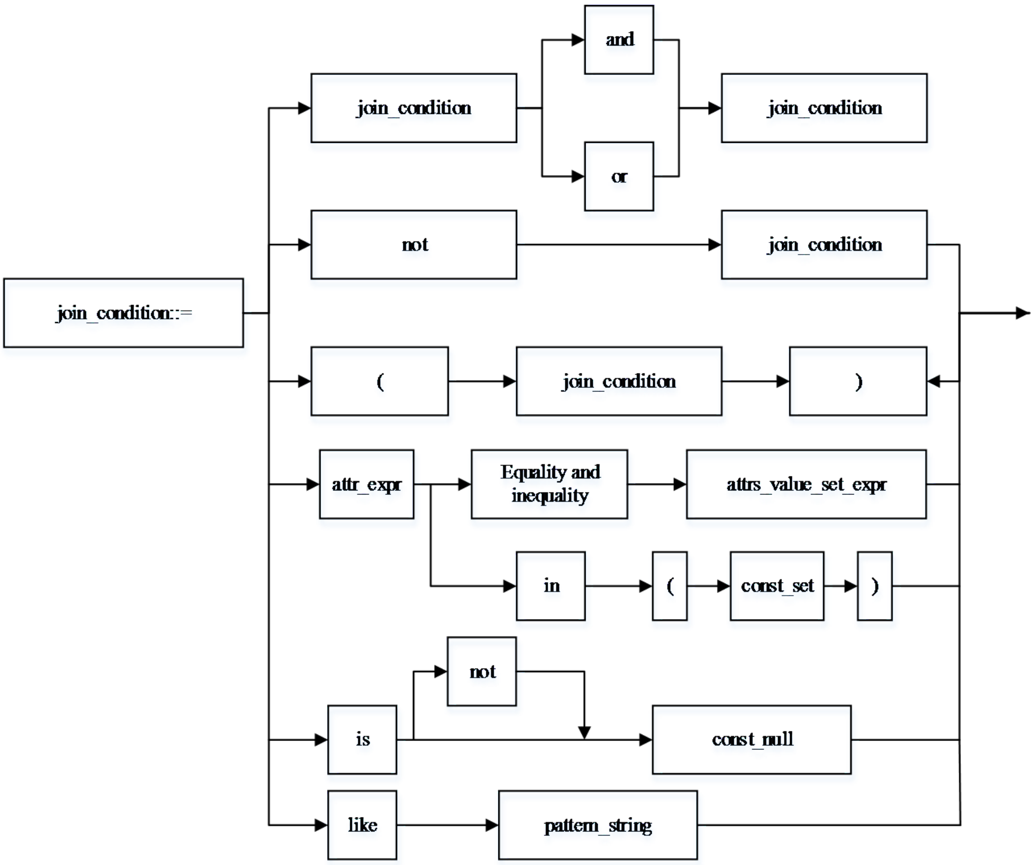
None.

Description

Input expression of the **CASE WHEN** statement.

3.24 join_condition

Syntax



Description

Syntax	Description
join_condition	Judgment condition of **join**.
and	Logical operator: AND
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjudgment condition.
)	End of the subjudgment condition.
condition	Judgment condition.
const_set	Collection of constants, which are separated by using comma (,).

Syntax	Description
in	Keyword used to determine whether attributes are in the same collection.
atrrs_value_set_expr	Collection of attribute values.
attr_expr	Attribute expression.
Equality and inequality	Equation and inequality. For details, see **Relational Operators**.
pattern_string	Pattern matching string, which supports wildcard matching. In WHERE LIKE, SQL wildcard characters "**%**" and "_" are supported. "**%**" represents one or more characters. "_" represents only one character.

3.25 non_equi_join_condition

Syntax

None.

Description

The condition of an inequality join.

3.26 number

Syntax

None.

Description

Maximum number of output lines specified by **LIMIT**. Which must be INT type.

3.27 partition_col_name

Syntax

None.

Description

Partition column name, that is, partition field name, which must be STRING type.

3.28 partition_col_value

Syntax

None.

Description

Partition column value, that is, partition field value.

3.29 partition_specs

Syntax

partition_specs : (partition_col_name = partition_col_value, partition_col_name = partition_col_value, ...);

Description

Table partition list, which is expressed by using key=value pairs, in which **key** represents **partition_col_name**, and **value** represents **partition_col_value**. If there is more than one partition field, separate every two key=value pairs by using a comma (,).

3.30 property_name

Syntax

None.

Description

Property name, which must be STRING type.

3.31 property_value

Syntax

None.

Description

Property value, which must be STRING type.

3.32 regex_expression

Syntax

None.

Description

Pattern matching string, which supports wildcard matching.

3.33 result_expression

Syntax

None.

Description

Returned result for the **THEN** clause of the **CASE WHEN** statement.

3.34 select_statement

Syntax

None.

Description

Query clause for the basic **SELECT** statement.

3.35 separator

Syntax

None.

Description

Separator, which can be customized by users, for example, comma (,), semicolon (;), and colon (:). Which must be CHAR type.

3.36 sql_containing_cte_name

Syntax

None.

Description

SQL statement containing the common expression defined by cte_name.

3.37 sub_query

Syntax

None.

Description

Subquery.

3.38 table_comment

Syntax

None.

Description

Table description, which must be STRING type and cannot exceed 256 bytes.

3.39 table_name

Syntax

None

Description

Table name, which cannot exceed 128 bytes. The string type and "$" symbol are supported.

3.40 table_properties

Syntax

None.

Description

Table property list, which is expressed by using key=value pairs. key represents **property_name**, and value represents **property_value**. If there is more than one key=value pair, separate every two key=value pairs by using a comma (,).

3.41 table_reference

Syntax

None.

Description

Table or view name, which must be STRING type. It can also be a subquery. If it is subquery, an alias must also be provided.

3.42 when_expression

Syntax

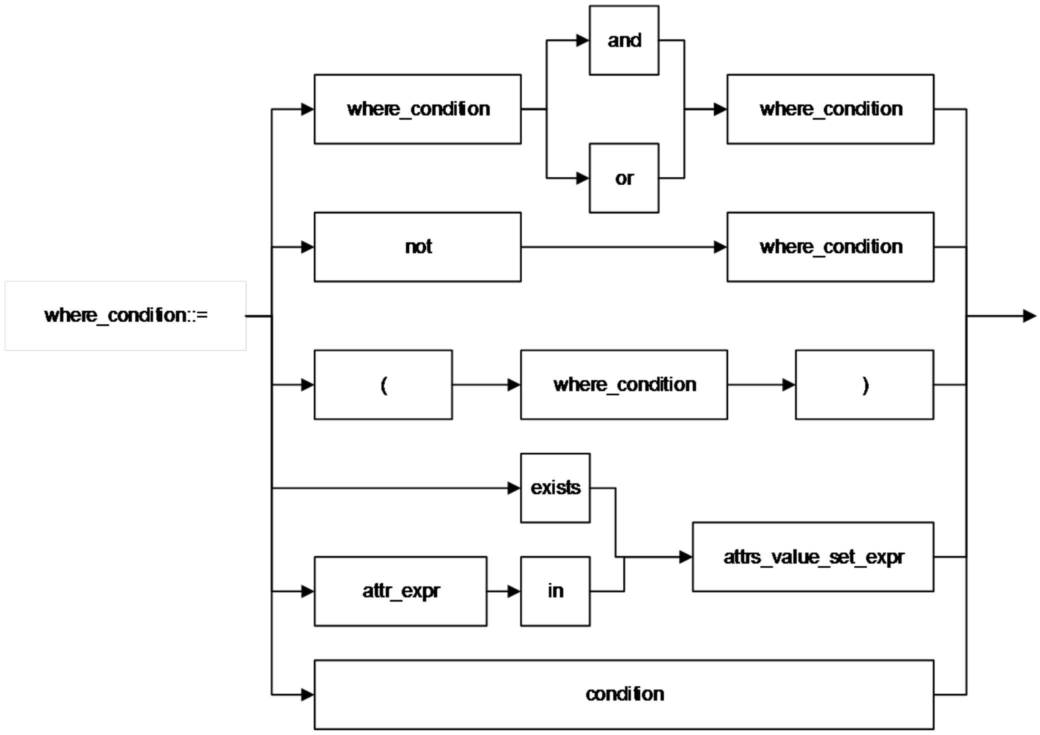
None.

Description

When expression of the **CASE WHEN** statement. It is used for matching with the input expression.

3.43 where_condition

Syntax



Description

Syntax	Description
where_condition	Judgment condition of **where**.
and	Logical operator: AND
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjudgment condition.
)	End of the subjudgment condition.
condition	Judgment condition.
exists	Keyword used to determine whether a non-empty collection exists. If exists is followed by a subquery, then the subquery must contain a judgment condition.
in	Keyword used to determine whether attributes are in the same collection.
attrs_value_set_expr	Collection of attribute values.
attr_expr	Attribute expression.

3.44 window_function

Syntax

None.

Description

Analysis window function. For details, see **Window Functions**.

4 Operators

4.1 Relational Operators

All data types can be compared by using relational operators and the result is returned as a BOOLEAN value.

Relationship operators are binary operators. Two compared data types must be of the same type or they must support implicit conversion.

Table 4-1 lists the relational operators provided by DLI.

Table 4-1 Relational operators

Operator	Result Type	Description
A = B	BOOLEAN	If A is equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. This operator is used for value assignment.
A == B	BOOLEAN	If A is equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. This operator cannot be used for value assignment.
A <=> B	BOOLEAN	If A is equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A and B are **NULL**, then **TRUE** is returned. If A or B is **NULL**, then **FALSE** is returned.
A <> B	BOOLEAN	If A is not equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned. This operator follows the standard SQL syntax.
A != B	BOOLEAN	This operator is the same as the <> logical operator. It follows the SQL Server syntax.

Operator	Result Type	Description
A < B	BOOLEAN	If A is less than B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A <= B	BOOLEAN	If A is less than or equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A > B	BOOLEAN	If A is greater than B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A >= B	BOOLEAN	If A is greater than or equal to B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A BETWEEN B AND C	BOOLEAN	If A is greater than or equal to B and less than or equal to C, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A, B, or C is **NULL**, then **NULL** is returned.
A NOT BETWEEN B AND C	BOOLEAN	If A is less than B or greater than C, **TRUE** is returned; otherwise, **FALSE** is returned. If A, B, or C is **NULL**, then **NULL** is returned.
A IS NULL	BOOLEAN	If A is **NULL**, then **TRUE** is returned. Otherwise, **FALSE** is returned.
A IS NOT NULL	BOOLEAN	If A is not **NULL**, then **TRUE** is returned. Otherwise, **FALSE** is returned.
A LIKE B	BOOLEAN	If A matches B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A NOT LIKE B	BOOLEAN	If A does not match B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A RLIKE B	BOOLEAN	This operator is used for the LIKE operation of JAVA. If A or its substring matches B, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A REGEXP B	BOOLEAN	The result is the same as A RLIKE B.

4.2 Arithmetic Operators

Arithmetic operators include binary operators and unary operators. For both types of operators, the returned results are numbers. **Table 4-2** lists the arithmetic operators supported by DLI.

Table 4-2 Arithmetic operators

Operator	Result Type	Description
A + B	All numeric types	A plus B. The result type is associated with the operation data type. For example, if floating-point number is added to an integer, the result will be a floating-point number.
A–B	All numeric types	A minus B. The result type is associated with the operation data type.
A * B	All numeric types	Multiply A and B. The result type is associated with the operation data type.
A / B	All numeric types	Divide A by B. The result is a number of the double type (double-precision number).
A % B	All numeric types	A on the B Modulo. The result type is associated with the operation data type.
A & B	All numeric types	Check the value of the two parameters in binary expressions and perform the AND operation by bit. If the same bit of both expressions are 1, then the bit is set to 1. Otherwise, the bit is 0.
A | B	All numeric types	Check the value of the two parameters in binary expressions and perform the OR operation by bit. If one bit of either expression is 1, then the bit is set to 1. Otherwise, the bit is set to 0.
A ^ B	All numeric types	Check the value of the two parameters in binary expressions and perform the XOR operation by bit. Only when one bit of either expression is 1, the bit is 1. Otherwise, the bit is 0.
~A	All numeric types	Perform the NOT operation on one expression by bit.

4.3 Logical Operators

Common logical operators include AND, OR, and NOT. The operation result can be TRUE, FALSE, or NULL (which means unknown). The priorities of the operators are as follows: NOT > AND > OR.

Table 4-3 lists the calculation rules, where A and B represent logical expressions.

Table 4-3 Logical operators

Operator	Result Type	Description
A AND B	BOOLEAN	If A and B are **TRUE**, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned.
A OR B	BOOLEAN	If A or B is **TRUE**, then **TRUE** is returned. Otherwise, **FALSE** is returned. If A or B is **NULL**, then **NULL** is returned. If one is **TRUE** and the other is **NULL**, then **TRUE** is returned.
NOT A	BOOLEAN	If A is **FALSE**, then **TRUE** is returned. If A is **NULL**, then **NULL** is returned. Otherwise, **FALSE** is returned.
! A	BOOLEAN	Same as NOT A.
A IN (val1, val2, ...)	BOOLEAN	If A is equal to any value in (val1, val2, ...), then **TRUE** is returned. Otherwise, **FALSE** is returned.
A NOT IN (val1, val2, ...)	BOOLEAN	If A is not equal to any value in (val1, val2, ...), then **TRUE** is returned. Otherwise, **FALSE** is returned.
EXISTS (subquery)	BOOLEAN	If the result of any subquery contains at least one line, then **TRUE** is returned. Otherwise, **FALSE** is returned.
NOT EXISTS (subquery)	BOOLEAN	If the subquery output does not contain any row, **TRUE** is returned; otherwise, **FALSE** is returned.

