Data Lake Insight

Flink SQL Syntax

Issue 01

Date 2025-01-10

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions

Notice

Security Declaration

Vulnerability

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

Contents

1 Flink OpenSource SQL 1.15 Syntax Reference....1
1.1 Constraints and Definitions.... 1
1.1.1 Supported Data Types.... 1
1.1.2 Reserved Keywords.... 1
1.1.3 Data Definition Language (DDL).... 3
1.1.3.1 CREATE TABLE.... 3
1.1.3.2 CREATE CATALOG.... 6
1.1.3.3 CREATE DATABASE.... 6
1.1.3.4 CREATE VIEW.... 7
1.1.3.5 CREATE FUNCTION.... 7
1.1.4 Data Manipulation Language (DML).... 8
1.2 Overview.... 10
1.3 Flink OpenSource SQL 1.15 Usage.... 12
1.4 Formats.... 13
1.4.1 Overview.... 13
1.4.2 Avro.... 13
1.4.3 Canal.... 16
1.4.4 Confluent Avro.... 21
1.4.5 CSV.... 25
1.4.6 Debezium.... 28
1.4.7 JSON.... 35
1.4.8 Maxwell.... 39
1.4.9 Ogg.... 44
1.4.10 ORC.... 48
1.4.11 Parquet.... 50
1.4.12 Raw.... 53
1.5 Connectors.... 56
1.5.1 Overview.... 56
1.5.2 BlackHole.... 57
1.5.3 ClickHouse.... 59
1.5.4 DataGen.... 66
1.5.5 Doris.... 70
1.5.5.1 Overview.... 70

1.5.5.2 Source Table....70
1.5.5.3 Result Table....74
1.5.5.4 Dimension Table....79
1.5.6 GaussDB(DWS)....83
1.5.6.1 Overview....83
1.5.6.2 GaussDB(DWS) Source Table (Not Recommended)....84
1.5.6.3 GaussDB(DWS) Result Table (Not Recommended)....89
1.5.6.4 GaussDB(DWS) Dimension Table (Not Recommended)....96
1.5.7 Elasticsearch....102
1.5.8 OBS....110
1.5.8.1 OBS Source Table....110
1.5.8.2 OBS Result Table....113
1.5.9 HBase....117
1.5.9.1 Source Table....118
1.5.9.2 HBase Result Table....123
1.5.9.3 Dimension Table....130
1.5.10 Hive....136
1.5.10.1 Creating a Hive Catalog....136
1.5.10.2 Hive Dialect....139
1.5.10.3 Hive Source Table....140
1.5.10.4 Result Table....147
1.5.10.5 Hive Dimension Table....149
1.5.10.6 Using Temporal Join to Associate the Latest Partition of a Dimension Table....150
1.5.10.7 Using Temporal Join to Associate the Latest Version of a Dimension Table....153
1.5.11 JDBC....155
1.5.12 Kafka....164
1.5.13 MySQL CDC....182
1.5.14 Print....196
1.5.15 Redis....199
1.5.15.1 Source Table....199
1.5.15.2 Result Table....206
1.5.15.3 Dimension Table....218
1.5.16 Upsert Kafka....225
1.6 DML Snytax....233
1.6.1 SELECT....233
1.6.2 INSERT INTO....237
1.6.3 Set Operations....239
1.6.4 Window....240
1.6.4.1 Window Functions....248
1.6.4.2 Window Aggregation....256
1.6.4.3 Window Top-N....259
1.6.4.4 Window Deduplication....261

1.6.4.5 Window Join 263
1.6.5 Group Aggregation 265
1.6.6 Over Aggregation 267
1.6.7 JOIN 269
1.6.8 OrderBy & Limit 272
1.6.9 Top-N 272
1.6.10 Deduplication 273
1.7 Functions 274
1.7.1 UDFs 274
1.7.2 Type Inference 278
1.7.3 Parameter Transfer 279
1.7.4 Built-In Functions 283
1.7.4.1 Comparison Functions 283
1.7.4.2 Logical Functions 287
1.7.4.3 Arithmetic Functions 288
1.7.4.4 String Functions 291
1.7.4.5 Temporal Functions 296
1.7.4.6 Conditional Functions 323
1.7.4.7 Type Conversion Functions 324
1.7.4.8 Collection Functions 327
1.7.4.9 JSON Functions 327
1.7.4.10 Value Construction Functions 333
1.7.4.11 Value Retrieval Functions 334
1.7.4.12 Grouping Functions 334
1.7.4.13 Hash Functions 335
1.7.4.14 Aggregate Functions 335
1.7.4.15 Table-Valued Functions 337
1.7.4.15.1 string_split 337
2 Flink OpenSource SQL 1.12 Syntax Reference 341
2.1 Constraints and Definitions 341
2.1.1 Supported Data Types 341
2.1.2 Syntax 341
2.1.2.1 Data Definition Language (DDL) 341
2.1.2.1.1 CREATE TABLE 341
2.1.2.1.2 CREATE VIEW 344
2.1.2.1.3 CREATE FUNCTION 344
2.1.2.2 Data Manipulation Language (DML) 345
2.2 Overview 347
2.3 DDL Syntax 348
2.3.1 Creating Source Tables 349
2.3.1.1 DataGen Source Table 349
2.3.1.2 GaussDB(DWS) Source Table 352

2.3.1.3 HBase Source Table 357
2.3.1.4 JDBC Source Table 361
2.3.1.5 Kafka Source Table 367
2.3.1.6 MySQL CDC Source Table 380
2.3.1.7 Postgres CDC Source Table 385
2.3.1.8 Redis Source Table 390
2.3.1.9 Upsert Kafka Source Table 397
2.3.1.10 FileSystem Source Table 402
2.3.2 Creating Result Tables 403
2.3.2.1 BlackHole Result Table 403
2.3.2.2 ClickHouse Result Table 404
2.3.2.3 GaussDB(DWS) Result Table 409
2.3.2.4 Elasticsearch Result Table 415
2.3.2.5 HBase Result Table 422
2.3.2.6 JDBC Result Table 429
2.3.2.7 Kafka Result Table 434
2.3.2.8 Print Result Table 445
2.3.2.9 Redis Result Table 447
2.3.2.10 Upsert Kafka Result Table 458
2.3.2.11 FileSystem Result Table 463
2.3.3 Creating Dimension Tables 468
2.3.3.1 GaussDB(DWS) Dimension Table 468
2.3.3.2 HBase Dimension Table 474
2.3.3.3 JDBC Dimension Table 480
2.3.3.4 Redis Dimension Table 486
2.3.4 Format 493
2.3.4.1 Avro 493
2.3.4.2 Canal 496
2.3.4.3 Confluent Avro 499
2.3.4.4 CSV 502
2.3.4.5 Debezium 504
2.3.4.6 JSON 507
2.3.4.7 Maxwell 511
2.3.4.8 Raw 514
2.4 DML Snytax 515
2.4.1 SELECT 515
2.4.2 Set Operations 519
2.4.3 Window 520
2.4.4 JOIN 529
2.4.5 OrderBy & Limit 531
2.4.6 Top-N 532
2.4.7 Deduplication 533

2.5 Functions.......534
2.5.1 User-Defined Functions (UDFs).......534
2.5.2 Type Inference.......537
2.5.3 Parameter Transfer.......539
2.5.4 Built-In Functions.......543
2.5.4.1 Mathematical Operation Functions.......543
2.5.4.2 String Functions.......551
2.5.4.3 Temporal Functions.......557
2.5.4.4 Conditional Functions.......580
2.5.4.5 Type Conversion Functions.......581
2.5.4.6 Collection Functions.......584
2.5.4.7 Value Construction Functions.......584
2.5.4.8 Value Access Functions.......585
2.5.4.9 Hash Functions.......585
2.5.4.10 Aggregate Functions.......586
2.5.4.11 Table-Valued Functions.......587
2.5.4.11.1 string_split.......587
3 Flink Opensource SQL 1.10 Syntax Reference.......590
3.1 Constraints and Definitions.......590
3.1.1 Supported Data Types.......590
3.1.2 Syntax Definition.......590
3.1.2.1 Data Definition Language (DDL).......590
3.1.2.1.1 CREATE TABLE.......590
3.1.2.1.2 CREATE VIEW.......593
3.1.2.1.3 CREATE FUNCTION.......593
3.1.2.2 Data Manipulation Language (DML).......594
3.2 Flink OpenSource SQL 1.10 Syntax.......596
3.3 Data Definition Language (DDL).......597
3.3.1 Creating a Source Table.......597
3.3.1.1 Kafka Source Table.......597
3.3.1.2 DIS Source Table.......600
3.3.1.3 JDBC Source Table.......603
3.3.1.4 GaussDB(DWS) Source Table.......605
3.3.1.5 Redis Source Table.......608
3.3.1.6 HBase Source Table.......609
3.3.1.7 userDefined Source Table.......611
3.3.2 Creating a Result Table.......612
3.3.2.1 ClickHouse Result Table.......613
3.3.2.2 Kafka Result Table.......616
3.3.2.3 Upsert Kafka Result Table.......617
3.3.2.4 DIS Result Table.......619
3.3.2.5 JDBC Result Table.......621

3.3.2.6 GaussDB(DWS) Result Table....622
3.3.2.7 Redis Result Table....626
3.3.2.8 SMN Result Table....629
3.3.2.9 HBase Result Table....631
3.3.2.10 Elasticsearch Result Table....633
3.3.2.11 OpenTSDB Result Table....635
3.3.2.12 User-defined Result Table....638
3.3.2.13 Print Result Table....640
3.3.2.14 File System Result Table....641
3.3.3 Creating a Dimension Table....644
3.3.3.1 JDBC Dimension Table....644
3.3.3.2 GaussDB(DWS) Dimension Table....647
3.3.3.3 HBase Dimension Table....650
3.4 Data Manipulation Language (DML)....651
3.4.1 SELECT....651
3.4.2 Set Operations....655
3.4.3 Window....656
3.4.4 JOIN....662
3.4.5 OrderBy & Limit....664
3.4.6 Top-N....665
3.4.7 Deduplication....666
3.5 Functions....667
3.5.1 User-Defined Functions....667
3.5.2 Built-In Functions....671
3.5.2.1 Mathematical Operation Functions....671
3.5.2.2 String Functions....679
3.5.2.3 Temporal Functions....686
3.5.2.4 Conditional Functions....710
3.5.2.5 Type Conversion Function....711
3.5.2.6 Collection Functions....713
3.5.2.7 Value Construction Functions....713
3.5.2.8 Value Access Functions....714
3.5.2.9 Hash Functions....714
3.5.2.10 Aggregate Function....715
3.5.2.11 Table-Valued Functions....716
3.5.2.11.1 split_cursor....716
3.5.2.11.2 string_split....717

1 Flink OpenSource SQL 1.15 Syntax Reference

1.1 Constraints and Definitions

1.1.1 Supported Data Types

DLI supports the following data types:

CHAR, VARCHAR, STRING, BOOLEAN, BINARY, VARBINARY, BYTES, DECIMAL, TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DATE, TIME, TIMESTAMP, TIMESTAMP_LTZ, INTERVAL, ARRAY, MULTISET, MAP, ROW, and RAW.

In the SQL syntax, these types are used to define the data types of columns within a table.

1.1.2 Reserved Keywords

Certain combinations of strings have been reserved as keywords for future use.

If you use any of the following strings as field names, enclose them in backticks when using them, for example: \`value\`, \`count\`.

A, ABS, ABSOLUTE, ACTION, ADA, ADD, ADMIN, AFTER, ALL, ALLOCATE, ALLOW, ALTER, ALWAYS, AND, ANY, ARE, ARRAY, AS, ASC, ASENSITIVE, ASSERTION, ASSIGNMENT, ASYMMETRIC, AT, ATOMIC, ATTRIBUTE, ATTRIBUTES, AUTHORIZATION, AVG, BEFORE, BEGIN, BERNOULLI, BETWEEN, BIGINT, BINARY, BIT, BLOB, BOOLEAN, BOTH, BREADTH, BY, BYTES, C, CALL, CALLED, CARDINALITY, CASCADE, CASCADED, CASE, CAST, CATALOG, CATALOG_NAME, CEIL, CEILING, CENTURY, CHAIN, CHAR, CHARACTER, CHARACTERISTICS, CHARACTERS, CHARACTER_LENGTH, CHARACTER_SET_CATALOG, CHARACTER_SET_NAME, CHARACTER_SET_SCHEMA, CHAR_LENGTH, CHECK, CLASS_ORIGIN, CLOB, CLOSE, COALESCE, COBOL, COLLATE, COLLATION, COLLATION_CATALOG, COLLATION_NAME, COLLATION_SCHEMA, COLLECT, COLUMN, COLUMNS, COLUMN_NAME, COMMAND_FUNCTION, COMMAND_FUNCTION_CODE, COMMIT, COMMITTED, CONDITION, CONDITION_NUMBER, CONNECT, CONNECTION, CONNECTION_NAME, CONSTRAINT, CONSTRAINTS,

CONSTRAINT_CATALOG, CONSTRAINT_NAME, CONSTRAINT_SCHEMA, CONSTRUCTOR, CONTAINS, CONTINUE, CONVERT, CORR, CORRESPONDING, COUNT, COVAR_POP, COVAR_SAMP, CREATE, CROSS, CUBE, CUME_DIST, CURRENT, CURRENT_CATALOG, CURRENT_DATE, CURRENT_DEFAULT_TRANSFORM_GROUP, CURRENT_PATH, CURRENT_ROLE, CURRENT_SCHEMA, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_TRANSFORM_GROUP_FOR_TYPE, CURRENT_USER, CURSOR, CURSOR_NAME, CYCLE, DATA, DATABASE, DATE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, DAY, DEALLOCATE, DEC, DECADE, DECIMAL, DECLARE, DEFAULT, DEFAULTS, DEFERRABLE, DEFERRED, DEFINED, DEFINER, DEGREE, DELETE, DENSE_RANK, DEPTH, DEREF, DERIVED, DESC, DESCRIBE, DESCRIPTION, DESCRIPTOR, DETERMINISTIC, DIAGNOSTICS, DISALLOW, DISCONNECT, DISPATCH, DISTINCT, DOMAIN, DOUBLE, DOW, DOY, DROP, DYNAMIC, DYNAMIC_FUNCTION, DYNAMIC_FUNCTION_CODE, EACH, ELEMENT, ELSE, END, END-EXEC, EPOCH, EQUALS, ESCAPE, EVERY, EXCEPT, EXCEPTION, EXCLUDE, EXCLUDING, EXEC, EXECUTE, EXISTS, EXP, EXPLAIN, EXTEND, EXTERNAL, EXTRACT, FALSE, FETCH, FILTER, FINAL, FIRST, FIRST_VALUE, FLOAT, FLOOR, FOLLOWING, FOR, FOREIGN, FORTRAN, FOUND, FRAC_SECOND, FREE, FROM, FULL, FUNCTION, FUSION, G, GENERAL, GENERATED, GET, GLOBAL, GO, GOTO, GRANT, GRANTED, GROUP, GROUPING, HAVING, HIERARCHY, HOLD, HOUR, IDENTITY, IMMEDIATE, IMPLEMENTATION, IMPORT, IN, INCLUDING, INCREMENT, INDICATOR, INITIALLY, INNER, INOUT, INPUT, INSENSITIVE, INSERT, INSTANCE, INSTANTIABLE, INT, INTEGER, INTERSECT, INTERSECTION, INTERVAL, INTO, INVOKER, IS, ISOLATION, JAVA, JOIN, K, KEY, KEY_MEMBER, KEY_TYPE, LABEL, LANGUAGE, LARGE, LAST, LAST_VALUE, LATERAL, LEADING, LEFT, LENGTH, LEVEL, LIBRARY, LIKE, LIMIT, LN, LOCAL, LOCALTIME, LOCALTIMESTAMP, LOCATOR, LOWER, M, MAP, MATCH, MATCHED, MAX, MAXVALUE, MEMBER, MERGE, MESSAGE_LENGTH, MESSAGE_OCTET_LENGTH, MESSAGE_TEXT, METHOD, MICROSECOND, MILLENNIUM, MIN, MINUTE, MINVALUE, MOD, MODIFIES, MODULE, MODULES, MONTH, MORE, MULTISET, MUMPS, NAME, NAMES, NATIONAL, NATURAL, NCHAR, NCLOB, NESTING, NEW, NEXT, NO, NONE, NORMALIZE, NORMALIZED, NOT, NULL, NULLABLE, NULLIF, NULLS, NUMBER, NUMERIC, OBJECT, OCTETS, OCTET_LENGTH, OF, OFFSET, OLD, ON, ONLY, OPEN, OPTION, OPTIONS, OR, ORDER, ORDERING, ORDINALITY, OTHERS, OUT, OUTER, OUTPUT, OVER, OVERLAPS, OVERLAY, OVERRIDING, PAD, PARAMETER, PARAMETER_MODE, PARAMETER_NAME, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_NAME, PARAMETER_SPECIFIC_SCHEMA, PARTIAL, PARTITION, PASCAL, PASSTHROUGH, PATH, PERCENTILE_CONT, PERCENTILE_DISC, PERCENT_RANK, PLACING, PLAN, PLI, POSITION, POWER, PRECEDING, PRECISION, PREPARE, PRESERVE, PRIMARY, PRIOR, PRIVILEGES, PROCEDURE, PUBLIC, QUARTER, RANGE, RANK, RAW, READ, READS, REAL, RECURSIVE, REF, REFERENCES, REFERENCING, 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, ROLLUP, ROUTINE, ROUTINE_CATALOG, ROUTINE_NAME, ROUTINE_SCHEMA, ROW, ROWS, ROW_COUNT, ROW_NUMBER, SAVEPOINT, SCALE, SCHEMA, 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, SIZE, 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, STATE, STATEMENT, STATIC, STDDEV_POP, STDDEV_SAMP, STREAM, STRING, STRUCTURE, STYLE, SUBCLASS_ORIGIN, SUBMULTISET, SUBSTITUTE, SUBSTRING, SUM, SYMMETRIC, SYSTEM, SYSTEM_USER, TABLE, TABLESAMPLE, TABLE_NAME, TEMPORARY, THEN, TIES, TIME, TIMESTAMP, TIMESTAMPADD, TIMESTAMPDIFF, TIMEZONE_HOUR, TIMEZONE_MINUTE, TINYINT, TO, TOP_LEVEL_COUNT, TRAILING, TRANSACTION, TRANSACTIONS_ACTIVE, TRANSACTIONS_COMMITTED, TRANSACTIONS_ROLLED_BACK, TRANSFORM, TRANSFORMS, TRANSLATE, TRANSLATION, TREAT, TRIGGER, TRIGGER_CATALOG, TRIGGER_NAME, TRIGGER_SCHEMA, TRIM, TRUE, TYPE, UESCAPE, UNBOUNDED, UNCOMMITTED, UNDER, UNION, UNIQUE, UNKNOWN, UNNAMED, UNNEST, UPDATE, UPPER, UPSERT, USAGE, USER, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_CODE, USER_DEFINED_TYPE_NAME, USER_DEFINED_TYPE_SCHEMA, USING, VALUE, VALUES, VARBINARY, VARCHAR, VARYING, VAR_POP, VAR_SAMP, VERSION, VIEW, WEEK, WHEN, WHENEVER, WHERE, WIDTH_BUCKET, WINDOW, WITH, WITHIN, WITHOUT, WORK, WRAPPER, WRITE, XML, YEAR, ZONE

1.1.3 Data Definition Language (DDL)

1.1.3.1 CREATE TABLE

Function

This statement creates a table using the specified table name. However, if a table with the same name already exists in the catalog, the registration process will fail.

Syntax

```
CREATE TABLE [IF NOT EXISTS] [catalog_name.][db_name.]table_name
  (
    { <physical_column_definition> | <metadata_column_definition> | <computed_column_definition> }[ , ...n]
    [ <watermark_definition> ]
    [ <table_constraint> ][ , ...n]
  )
  [COMMENT table_comment]
  [PARTITIONED BY (partition_column_name1, partition_column_name2, ...)]
  WITH (key1=val1, key2=val2, ...)
  [ LIKE source_table [( <like_options> )] ]

<physical_column_definition>:
  column_name column_type [ <column_constraint> ] [COMMENT column_comment]

<column_constraint>:
  [CONSTRAINT constraint_name] PRIMARY KEY NOT ENFORCED

<table_constraint>:
  [CONSTRAINT constraint_name] PRIMARY KEY (column_name, ...) NOT ENFORCED

<metadata_column_definition>:
  column_name column_type METADATA [ FROM metadata_key ] [ VIRTUAL ]

<computed_column_definition>:
  column_name AS computed_column_expression [COMMENT column_comment]

<watermark_definition>:
  WATERMARK FOR rowtime_column_name AS watermark_strategy_expression
```

```
<source_table>:
  [catalog_name.][db_name.]table_name

<like_options>:
{
   { INCLUDING | EXCLUDING } { ALL | CONSTRAINTS | PARTITIONS }
 | { INCLUDING | EXCLUDING | OVERWRITING } { GENERATED | OPTIONS | WATERMARKS }
}[, ...]
```

Description

COMPUTED COLUMN

A computed column is a virtual column generated using **column_name AS computed_column_expression**. A computed column evaluates an expression that can reference other columns declared in the same table. The column itself is not physically stored within the table. A computed column could be defined using **cost AS price * quantity**. This expression can contain any combination of physical columns, constants, functions, or variables, but cannot contain any subquery.

In Flink, a computed column is used to define the time attribute in **CREATE TABLE** statements. A processing time attribute can be defined easily via **proc AS PROCTIME()** using the system's **PROCTIME()** function. The event time column may be obtained from an existing field. In this case, you can use the computed column to obtain event time. For example, if the original field is not of the **TIMESTAMP(3)** type or is nested in a JSON string, you can use computed columns.

Note:

- An expression that defines a computed column in a source table is calculated after data is read from the data source. The column can be used in the **SELECT** statement.
- A computed column cannot be the target of an **INSERT** statement. In an **INSERT** statement, the schema of the **SELECT** statement must be the same as that of the target table that does not have a computed column.

WATERMARK

The **WATERMARK** clause defines the event time attribute of a table and takes the form **WATERMARK FOR rowtime_column_name AS watermark_strategy_expression**.

rowtime_column_name defines an existing column that is marked as the event time attribute of the table. The column must be of the **TIMESTAMP(3)** type and must be the top-level column in the schema. It can also be a computed column.

watermark_strategy_expression defines the watermark generation strategy. It allows arbitrary non-query expressions, including computed columns, to calculate the watermark. The expression return type must be **TIMESTAMP(3)**, which represents the timestamp since the Epoch. The returned watermark will be emitted only if it is non-null and its value is greater than the previously emitted local watermark (to preserve the contract of ascending watermarks). The watermark generation expression is evaluated by the framework for every record. The framework will periodically emit the largest generated watermark. If the current watermark is still identical to the previous one, or is null, or the value of the returned watermark is smaller than that of the last emitted one, then no new watermark will be emitted. A watermark is emitted in an interval defined by **pipeline.auto-watermark-interval**. If the watermark interval is 0 ms, a watermark will be emitted per record if it is not null and greater than the last emitted one.

When using event time semantics, tables must contain an event time attribute and watermark strategy.

Flink provides several commonly used watermark strategies.

- Strictly ascending timestamps: **WATERMARK FOR rowtime_column AS rowtime_column**

  Emits a watermark of the maximum observed timestamp so far. Rows that have a timestamp bigger than the maximum timestamp are not late.
- Ascending timestamps: **WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL '0.001' SECOND**

  Emits a watermark of the maximum observed timestamp so far minus 1. Rows that have a timestamp bigger than or equal to the maximum timestamp are not late.
- Bounded out-of-order timestamps: **WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL 'string' timeUnit**

  Emits a watermark, which is the maximum observed timestamp minus the specified delay, for example, **WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL '5' SECOND** is a 5-second delayed watermark strategy.

  ```
  CREATE TABLE Orders (
      user BIGINT,
      product STRING,
      order_time TIMESTAMP(3),
      WATERMARK FOR order_time AS order_time - INTERVAL '5' SECOND
  ) WITH ( . . . );
  ```

PRIMARY KEY

The primary key constraint is a hint for Flink to leverage for optimizations. It tells that a column or a set of columns of a table or a view are unique and they do not contain null. Neither of columns in a primary can be nullable. The primary key therefore uniquely identifies a row in a table.

The primary key constraint can be either declared along with a column definition (a column constraint) or as a single line (a table constraint). For both cases, it should only be declared as a singleton. If you define multiple primary key constraints at the same time, an exception would be thrown.

Validity Check

SQL standard specifies that a constraint can either be **ENFORCED** or **NOT ENFORCED**. This controls if the constraint checks are performed on the incoming/ outgoing data. Flink does not own the data and therefore the only mode we want to support is the **NOT ENFORCED** mode. It is up to the user to ensure that the query enforces key integrity.

Flink will assume correctness of the primary key by assuming that the columns nullability is aligned with the columns in the primary key. Connectors should ensure those are aligned.

Note: In a **CREATE TABLE** statement, creating a primary key constraint will alter the columns nullability, which means, a column with a primary key constraint is not nullable.

PARTITIONED BY

Partition the created table by the specified columns. A directory is created for each partition if this table is used as a file system sink.

WITH OPTIONS

Table properties used to create a table source/sink. The properties are usually used to find and create the underlying connector.

The key and value of expression **key1=val1** should both be string literal.

Note: The table name can be in any of the following formats: 1. catalog_name.db_name.table_name 2. db_name.table_name 3. table_name. Tables named in the **catalog_name.db_name.table_name** format are registered with metastore along with the catalog named **catalog_name** and the database named **db_name**. Tables named in the **uses db_name.table_name** format will be registered with the current table environment's catalog and the database will be named **db_name**. Tables named in the **table_name** format will be registered with the running catalog and database.

Note: Tables registered using the **CREATE TABLE** statement can be used as both the table source and table sink. We cannot decide if it is used as a source or sink until it is referenced in the DMLs.

1.1.3.2 CREATE CATALOG

Function

This statement creates a catalog using specified attributes. If a catalog with the same name already exists, an exception is thrown.

Syntax

```
CREATE CATALOG catalog_name
  WITH (key1=val1, key2=val2, ...)
```

Description

WITH OPTIONS

Catalog attributes typically store additional information about the catalog.

The key and value of the **key1=val1** expression are string literals.

1.1.3.3 CREATE DATABASE

Function

This statement creates a database using specified table attributes. If a table with the same name already exists in the database, an exception is thrown.

Syntax

```
CREATE DATABASE [IF NOT EXISTS] [catalog_name.]db_name
  [COMMENT database_comment]
  WITH (key1=val1, key2=val2, ...)
```

Description

IF NOT EXISTS

If the database already exists, no operation is performed.

WITH OPTIONS

Database attributes typically store additional information about the database.

The key and value of the **key1=val1** expression are string literals.

1.1.3.4 CREATE VIEW

Function

This statement creates a view based on the given query statement. If a view with the same name already exists in the database, an exception is thrown.

Syntax

```
CREATE [TEMPORARY] VIEW [IF NOT EXISTS] [catalog_name.][db_name.]view_name
  [{columnName [, columnName ]* }] [COMMENT view_comment]
  AS query_expression
```

Description

TEMPORARY

Create a temporary view with catalogs and database namespaces and overwrite the original view.

IF NOT EXISTS

If the view already exists, nothing happens.

Example

Create a view named **viewName**.

```
create view viewName as select * from dataSource
```

1.1.3.5 CREATE FUNCTION

Function

To create a catalog function with a catalog and a database namespace, you will need to specify an identifier. You can specify a language tag. You cannot register the function if a function with the same name has already been registered in the catalog. If the language tag is **JAVA** or **SCALA**, the identifier is the fully qualified name of the UDF implementation class.

For details about how to create a UDF, see **UDFs**.

Syntax

```
CREATE [TEMPORARY|TEMPORARY SYSTEM] FUNCTION
 [IF NOT EXISTS] [[catalog_name.]db_name.]function_name
 AS identifier [LANGUAGE JAVA|SCALA]
```

Description

TEMPORARY

Create a temporary catalog function with catalogs and database namespaces and overwrite the original catalog function.

TEMPORARY SYSTEM

Create a temporary system catalog function without database namespaces and overwrite the system's built-in functions.

IF NOT EXISTS

If the function already exists, nothing happens.

LANGUAGE JAVA|SCALA

The language tag is used to instruct Flink runtime how to execute the function. Currently, only **JAVA** and **SCALA** language tags are supported, and the default language for a function is **JAVA**.

Example

Create a function named **STRINGBACK**.

```
create function STRINGBACK as 'com.dli.StringBack'
```

1.1.4 Data Manipulation Language (DML)

Constraints and Limitations

- Flink SQL uses a lexical policy for identifier (table, attribute, function names) similar to Java:
  - The case of identifiers is preserved whether they are quoted.
  - Identifiers are matched case-sensitively.
  - Unlike Java, back-ticks allow identifiers to contain non-alphanumeric characters (for example, **SELECT a AS `my field` FROM t**).
- String literals must be enclosed in single quotes (for example, **SELECT'Hello World'**). Duplicate a single quote for escaping (for example, **SELECT 'It''s me.'**). Unicode characters are supported in string literals. If explicit Unicode points are required, use the following syntax:
  - Use the backslash (\) as an escaping character (default): **SELECT U&'\263A'**
  - Use a custom escaping character: **SELECT U&'#263A' UESCAPE '#'**

Syntax

```
INSERT INTO table_name [PARTITION part_spec] query

part_spec:  (part_col_name1=val1 [, part_col_name2=val2, ...])

query:
  values
  | {
      select
      | selectWithoutFrom
      | query UNION [ ALL ] query
      | query EXCEPT query
      | query INTERSECT query
    }
    [ ORDER BY orderItem [, orderItem ]* ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start { ROW | ROWS } ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY]

orderItem:
  expression [ ASC | DESC ]

select:
  SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
  [ WINDOW windowName AS windowSpec [, windowName AS windowSpec ]* ]

selectWithoutFrom:
  SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }

projectItem:
  expression [ [ AS ] columnAlias ]
  | tableAlias . *

tableExpression:
  tableReference [, tableReference ]*
  | tableExpression [ NATURAL ] [ LEFT | RIGHT | FULL ] JOIN tableExpression [ joinCondition ]

joinCondition:
  ON booleanExpression
  | USING '(' column [, column ]* ')'

tableReference:
  tablePrimary
  [ matchRecognize ]
  [ [ 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 ]* ')'

windowRef:
```

```
  windowName
 | windowSpec

windowSpec:
   [ windowName ]
   '('
   [ ORDER BY orderItem [, orderItem ]* ]
   [ PARTITION BY expression [, expression ]* ]
   [
       RANGE numericOrIntervalExpression {PRECEDING}
     | ROWS numericExpression {PRECEDING}
   ]
   ')'

matchRecognize:
      MATCH_RECOGNIZE '('
      [ PARTITION BY expression [, expression ]* ]
      [ ORDER BY orderItem [, orderItem ]* ]
      [ MEASURES measureColumn [, measureColumn ]* ]
      [ ONE ROW 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 ]*
      ')'

measureColumn:
      expression AS alias

pattern:
      patternTerm [ '|' patternTerm ]*

patternTerm:
      patternFactor [ patternFactor ]*

patternFactor:
      variable [ patternQuantifier ]

patternQuantifier:
      '*'
  |   '*?'
  |   '+'
  |   '+?'
  |   '?'
  |   '??'
  |   '{' { [ minRepeat ], [ maxRepeat ] } '}' ['?']
  |   '{' repeat '}'
```

1.2 Overview

This section describes the Flink OpenSource SQL 1.15 syntax supported by DLI. For details about the parameters and examples, see the syntax description.

Creating Tables

Table 1-1 Syntax for creating tables

Classification	Function
Format	**Avro**
	Canal
	Confluent Avro
	CSV
	Debezium
	JSON
	Maxwell
	Ogg
	Orc
	Parquet
	Raw
Connectors	**BlackHole**
	ClickHouse
	DataGen
	Doris
	DWS
	Elasticsearch
	FileSystem
	Hbase
	Hive
	JDBC
	Kafka
	Print
	Redis
	Upsert Kafka

1.3 Flink OpenSource SQL 1.15 Usage

When switching from Flink 1.12 to Flink 1.15 for job execution, keep in mind the following considerations when utilizing Flink OpenSource SQL 1.15:

- Flink SQL utilizes a SQL client submission method. To configure this submission method in Flink 1.15, you need to use the **SET 'key'='value'** command in your SQL script. This is different from the optimization parameters used in Flink 1.12. For details about the syntax, see **SQL Client Configuration**.
- The following Flink connectors are added to Flink 1.15: Doris Connector and Hive Connector. For details, see **Overview**.
- In Flink 1.15, you need to configure a custom agency on the tenant plane and configure agency information in the job. The permissions included an agency should be configured based on the specific service scenario requirements of the job. For details, see **DLI Custom Agency**.
- Methods to manage credentials for Flink 1.15 jobs:
  - You are advised to use DEW to manage access credentials, such as passwords and keys, in Flink OpenSource SQL. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
  - Manage fixed AKs/SKs used by Flink Jar jobs to access OBS, temporary AKs/SKs used by Flink Jar jobs to obtain agencies, and temporary AKs/SKs used by Flink SQL UDFs to obtain agencies. For details, see **Flink Job Agencies**.
- There are differences in the way Flink 1.15 Jar reads custom configuration files compared to Flink 1.12. For details, see **Writing Data to OBS Using Flink Jar**.
- The Flink 1.15 Jar program uses a child-first reverse class loading mechanism. By setting the **parent.first.classloader.jars** parameter to include the names of the desired jars, for example, **test1.jar,test2.jar**, certain dependency packages can be loaded by the parent class loader.
- For the built-in JAR file list of Flink 1.15 Jar, obtain information about Flink 1.15 dependency packages from Flink job logs.
  a. Check the logs of a Flink job.
     i. Log in to the DLI management console. In the navigation pane on the left, choose **Job Management** > **Flink Jobs**.
     ii. Click the name of the desired job. On the displayed page, click the **Run Log** tab.
     iii. Check the latest run logs. For more logs, check the OBS bucket where the job logs are stored.
  b. Search for dependency information in the logs.

     Search for **Classpath:** in the logs to check the dependencies.
- Flink 1.15 no longer supports DLI package management. To upload dependency packages and files, select the OBS path directly when editing the job.

1.4 Formats

1.4.1 Overview

Flink provides a set of table formats that can be used with table connectors.

A table format is a storage format that defines how to map binary data onto table columns.

Table 1-2 Formats supported by Flink

Formats	Supported Connectors
CSV	**Kafka**, **Upsert Kafka**, **FileSystem**
JSON	**Kafka**, **Upsert Kafka**, **FileSystem**, **Elasticsearch**
Avro	**Kafka**, **Upsert Kafka**, **FileSystem**
Confluent Avro	**Kafka**, **Upsert Kafka**
Debezium	**Kafka**, **FileSystem**
Canal	**Kafka**, **FileSystem**
Maxwell	**Kafka**, **FileSystem**
Ogg	**Kafka**, **FileSystem**
Orc	**FileSystem**
Parquet	**FileSystem**
Raw	**Kafka**, **Upsert Kafka**, **FileSystem**

1.4.2 Avro

Function

Apache Avro is supported for you to read and write Avro data based on an Avro schema with Flink. The Avro schema is derived from the table schema.

For details, see **Avro Format**.

Supported Connectors

- Kafka
- Upsert Kafka
- FileSystem

Parameters

Table 1-3 Parameters

Parameter	Mandatory	Default value	Type	Description
format	Yes	None	String	Format to be used. Set the value to **avro**.
avro.codec	No	None	String	For Filesystem only, the compression codec for avro. Snappy compression as default. The valid enumerations are: **null**, **deflate**, **snappy**, **bzip2**, and **xz**.

Data Type Mapping

Currently, the Avro schema is derived from the table schema and cannot be explicitly defined. The following table lists mappings between Flink to Avro types.

In addition to the following types, Flink supports reading/writing nullable types. Flink maps nullable types to Avro **union(something, null)**, where **something** is an Avro type converted from Flink type.

Table 1-4 Data type mapping

Flink SQL Type	Avro Type	Avro Logical Type
CHAR/VARCHAR/STRING	String	-
BOOLEAN	Boolean	-
BINARY/VARBINARY	bytes	-
DECIMAL	fixed	decimal
TINYINT	int	-
SMALLINT	int	-
INT	int	-
BIGINT	long	-

Flink SQL Type	Avro Type	Avro Logical Type
FLOAT	float	-
DOUBLE	double	-
DATE	int	date
TIME	int	time-millis
TIMESTAMP	long	timestamp-millis
ARRAY	array	-
MAP (keys must be of the string, char, or varchar type.)	map	-
MULTISET (elements must be of the string, char, or varchar type.)	map	-
ROW	record	-

Example

Read data from Kafka, deserialize the data to the Avro format, and outputs the data to Print.

Step 1 Create a datasource connection for access to the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.15. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'avro'
);
```

```
CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);
insert into printSink select * from kafkaSource;
```

Step 3 Insert the following data to Kafka using Avro data serialization:

```
{"order_id":"202103241000000001","order_channel":"webShop","order_time":"2021-03-24
10:00:00","pay_amount":100.0,"real_pay":100.0,"pay_time":"2021-03-24
10:02:03","user_id":"0001","user_name":"Alice","area_id":"330106"}

{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106"}
```

Step 4 Perform the following operations to view the data result in the **taskmanager.out** file:

1. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
2. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
3. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **.out** file, and view result logs.

```
+I[202103241000000001, webShop, 2021-03-24 10:00:00, 100.0, 100.0, 2021-03-24 10:02:03, 0001, Alice,
330106]
+I[202103241606060001, appShop, 2021-03-24 16:06:06, 200.0, 180.0, 2021-03-24 16:10:06, 0001, Alice,
330106]
```

----End

1.4.3 Canal

Function

Canal is a Changelog Data Capture (CDC) tool that can stream changes in real-time from MySQL into other systems. Canal provides a unified format schema for changelog and supports to serialize messages using JSON and protobuf (the default format for Canal).

Flink supports to interpret Canal JSON messages as INSERT, UPDATE, and DELETE messages into the Flink SQL system. This is useful in many cases to leverage this feature, such as:

- synchronizing incremental data from databases to other systems
- Auditing logs
- Real-time materialized view on databases
- Temporal join changing history of a database table, etc.

Flink also supports to encode the INSERT, UPDATE, and DELETE messages in Flink SQL as Canal JSON messages, and emit to storage like Kafka. However, currently Flink cannot combine UPDATE_BEFORE and UPDATE_AFTER into a single UPDATE message. Therefore, Flink encodes UPDATE_BEFORE and UPDATE_AFTER as DELETE and INSERT Canal messages.

For details, see **Canal Format**.

Supported Connectors

- Kafka
- FileSystem

Parameters

Table 1-5 Parameter description

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. In this example.Set this parameter to **canal-json**.
canal-json.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. The default value is **false**, indicating that an error will be thrown. Fields are set to null in case of errors.
canal-json.timestamp-format.standard	No	'SQL'	String	Input and output timestamp formats. Currently supported values are **SQL** and **ISO-8601**: • **SQL** will parse input timestamp in "yyyy-MM-dd HH:mm:ss.s{precision}" format, for example **2020-12-30 12:13:14.123** and output timestamp in the same format. • **ISO-8601** will parse input timestamp in "yyyy-MM-ddTHH:mm:ss.s{precision}" format, for example **2020-12-30T12:13:14.123** and output timestamp in the same format.

Parameter	Mandatory	Default Value	Type	Description
canal-json.map-null-key.mode	No	'FALL'	String	Handling mode when serializing null keys for map data. Available values are as follows: ● **FAIL** will throw exception when encountering map value with null key. ● **DROP** will drop null key entries for map data. ● **LITERAL** replaces the empty key value in the map with a string constant. The string literal is defined by **canal-json.map-null-key.literal** option.
canal-json.map-null-key.literal	No	'null'	String	String literal to replace null key when **canal-json.map-null-key.mode** is **LITERAL**.
canal-json.encode.decimal-as-plain-number	No	false	Boolean	Encode all decimals as plain numbers instead of possible scientific notations. By default, decimals may be written using scientific notation. For example, **0.000000027** is encoded as **2.7E-8** by default, and will be written as **0.000000027** if set this parameter to **true**.
canal-json.database.include	No	None	String	An optional regular expression to only read the specific databases changelog rows by regular matching the "database" meta field in the Canal record. The pattern string is compatible with Java's **Pattern**.
canal-json.table.include	No	None	String	An optional regular expression to only read the specific tables changelog rows by regular matching the "table" meta field in the Canal record. The pattern string is compatible with Java's **Pattern**.

Metadata

The following format metadata can be exposed as read-only (VIRTUAL) columns in DDL.

Format metadata fields are only available if the corresponding connector forwards format metadata. Currently, only the Kafka connector is able to expose metadata fields for its value format.

Table 1-6 Metadata

Key	Data Type	Description
database	STRING NULL	The originating database. Corresponds to the **database** field in the Canal record if available.
table	STRING NULL	The originating database table. Corresponds to the **table** field in the Canal record if available.
sql-type	MAP<STRING, INT> NULL	Map of various sql types. Corresponds to the **sqlType** field in the Canal record if available.
pk-names	ARRAY<STRING> NULL	Array of primary key names. Corresponds to the **pkNames** field in the Canal record if available.
ingestion-timestamp	TIMESTAMP_LTZ(3) NULL	The timestamp at which the connector processed the event. Corresponds to the **ts** field in the Canal record.

The following example shows how to access Canal metadata fields in Kafka:

```
CREATE TABLE KafkaTable (
  origin_database STRING METADATA FROM 'value.database' VIRTUAL,
  origin_table STRING METADATA FROM 'value.table' VIRTUAL,
  origin_sql_type MAP<STRING, INT> METADATA FROM 'value.sql-type' VIRTUAL,
  origin_pk_names ARRAY<STRING> METADATA FROM 'value.pk-names' VIRTUAL,
  origin_ts TIMESTAMP(3) METADATA FROM 'value.ingestion-timestamp' VIRTUAL,
  user_id BIGINT,
  item_id BIGINT,
  behavior STRING
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'earliest-offset',
  'value.format' = 'canal-json'
);
```


Example

Use canal-json to read Canal records in Kafka and output them to Print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.15. Copy the following statement and submit the job:

```
create table kafkaSource(
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
  ) with (
    'connector' = 'kafka',
    'topic' = '<yourTopic>',
    'properties.group.id' = '<yourGroupId>',
    'properties.bootstrap.servers' = '<yourKafkaAddress>:<yourKafkaPort>',
    'scan.startup.mode' = 'latest-offset',
    'format' = 'canal-json'
);
create table printSink(
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
   ) with (
     'connector' = 'print'
   );
insert into printSink select * from kafkaSource;
```

Step 3 Insert the data below into the appropriate Kafka topics. The data shows that the MySQL products table has four columns: **id**, **name**, **description**, and **weight**. This JSON message is an update event on the products table, indicating that the value of the **weight** field has changed from 5.15 to 5.18 for the row with id = 111.

```
{
  "data": [
    {
      "id": "111",
      "name": "scooter",
      "description": "Big 2-wheel scooter",
      "weight": "5.18"
    }
  ],
  "database": "inventory",
  "es": 1589373560000,
  "id": 9,
  "isDdl": false,
  "mysqlType": {
    "id": "INTEGER",
    "name": "VARCHAR(255)",
    "description": "VARCHAR(512)",
    "weight": "FLOAT"
  },
  "old": [
    {
      "weight": "5.15"
    }
  ],
```

```
 "pkNames": [
   "id"
 ],
 "sql": "",
 "sqlType": {
   "id": 4,
   "name": 12,
   "description": 12,
   "weight": 7
 },
 "table": "products",
 "ts": 1589373560798,
 "type": "UPDATE"
}
```

Step 4 Perform the following operations to view the data result in the **taskmanager.out** file:

1. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
2. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
3. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **.out** file, and view result logs.

```
-U[111, scooter, Big 2-wheel scooter, 5.15]
+U[111, scooter, Big 2-wheel scooter, 5.18]
```

----End

1.4.4 Confluent Avro

Function

The Avro Schema Registry (**avro-confluent**) format allows you to read records that were serialized by the **io.confluent.kafka.serializers.KafkaAvroSerializer** and to write records that can in turn be read by the **io.confluent.kafka.serializers.KafkaAvroDeserializer**.

When reading (deserializing) a record with this format the Avro writer schema is fetched from the configured Confluent Schema Registry based on the schema version ID encoded in the record while the reader schema is inferred from table schema.

When writing (serializing) a record with this format the Avro schema is inferred from the table schema and used to retrieve a schema ID to be encoded with the data The lookup is performed with in the configured Confluent Schema Registry under the **subject**. The subject is specified by the avro-confluent.subject parameter.

Supported Connectors

- kafka
- upsert kafka

Parameters

Table 1-7 Parameter description

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. Set this parameter to **'avro-confluent'**.
avro-confluent.basic-auth.credentials-source	No	None	String	Basic auth credentials source for Schema Registry
avro-confluent.basic-auth.user-info	No	None	String	Basic auth user info for schema registry
avro-confluent.bearer-auth.credentials-source	No	None	String	Bearer auth credentials source for Schema Registry
avro-confluent.bearer-auth.token	No	None	String	Bearer auth token for Schema Registry
avro-confluent.properties	No	None	Map	Properties map that is forwarded to the underlying Schema Registry. This is useful for options that are not officially exposed via Flink config options. However, note that Flink options have higher precedence.
avro-confluent.ssl.keystore.location	No	None	String	Location/File of SSL keystore
avro-confluent.ssl.keystore.password	No	None	String	Password for SSL keystore
avro-confluent.ssl.truststore.location	No	None	String	Location/File of SSL truststore

Parameter	Mandatory	Default Value	Type	Description
avro-confluent.ssl.truststore.password	No	None	String	Password for SSL truststore
avro-confluent.subject	No	None	String	The Confluent Schema Registry subject under which to register the schema used by this format during serialization. By default, 'kafka' and 'upsert-kafka' connectors use '<topic_name>-value' or '<topic_name>-key' as the default subject name if this format is used as the value or key format. But for other connectors (e.g. 'filesystem'), the subject option is required when used as sink.
avro-confluent.url	No	None	String	The URL of the Confluent Schema Registry to fetch/register schemas.

Data Type Mapping

Currently, Apache Flink always uses the table schema to derive the Avro reader schema during deserialization and Avro writer schema during serialization. Explicitly defining an Avro schema is not supported yet. See **Avro** for the mapping between Avro and Flink DataTypes.

In addition to the types listed there, Flink supports reading/writing nullable types. Flink maps nullable types to Avro **union(something, null)**, where **something** is the Avro type converted from Flink type.

Example

Read JSON data from the source topic in Kafka and write the data in Confluent Avro format to the sink topic.

1. Create a datasource connection for the communication with the VPC and subnet where Kafka and ECS locate and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka and ECS IP addresses. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
2. Purchase an ECS cluster, download **Confluent** 5.5.2 and jdk1.8.0_232, and upload them to the ECS cluster. Run the following command to decompress the packages (assume that the decompression directories are **confluent-5.5.2** and **jdk1.8.0_232**):
```
tar zxvf confluent-5.5.2-2.11.tar.gz
tar zxvf jdk1.8.0_232.tar.gz
```

3. Run the following commands to install jdk1.8.0_232 in the current ECS cluster. You can run the **pwd** command in the **jdk1.8.0_232 folder** to view the value of **yourJdkPath**.
```
export JAVA_HOME=<yourJdkPath>
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
```
4. Go to the **confluent-5.5.2/etc/schema-registry/** directory and modify the following configuration items in the **schema-registry.properties** file:
```
listeners=http://<yourEcsIp>:8081
kafkastore.bootstrap.servers=<yourKafkaAddress1>:<yourKafkaPort>,<yourKafkaAddress2>:<yourKafkaPort>
```
5. Switch to the **confluent-5.5.2** directory and run the following command to start Confluent:
```
bin/schema-registry-start etc/schema-registry/schema-registry.properties
```
6. Create a Flink OpenSource SQL job, select the Flink 1.15 version, and allow DLI to save job logs in OBS. Add the following statement to the job and submit it:
```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaSourceTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE kafkaSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaSinkTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'avro-confluent',
  'avro-confluent.url' = 'http://EcsIp:8081'
);
insert into kafkaSink select * from kafkaSource;
```
7. Insert the following data into Kafka:
```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}
```
8. Read the data of the sink Kafka topic. You will find that the data has been written and the schema has been saved to the **_schema** topic of Kafka.

1.4.5 CSV

Function

The CSV format allows you to read and write CSV data based on a CSV schema. Currently, the CSV schema is derived from table schema. For details, see **CSV Format**.

Supported Connectors

- Kafka
- Upsert Kafka
- FileSystem

Parameters

Table 1-8 Description

Parameter	Mandatory	Default value	Type	Description
format	Yes	None	String	Format to be used. Set the value to **csv**.
csv.field-delimiter	No	,	String	Field delimiter character, which must be a single character. You can use backslash to specify special characters, for example, **\t** represents the tab character. You can also use unicode to specify them in plain SQL, for example, **'csv.field-delimiter' = U&'\0001'** represents the 0x01 character.
csv.disable-quote-character	No	false	Boolean	Disabled quote character for enclosing field values. If you set this parameter to **true**, **csv.quote-character** cannot be set.
csv.quote-character	No	"	String	Quote character for enclosing field values.
csv.allow-comments	No	false	Boolean	Ignore comment lines that start with **#**. If you set this parameter to **true**, make sure to also ignore parse errors to allow empty rows.
csv.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. The default value is **false**, indicating that an error will be thrown. Fields are set to null in case of errors.

Parameter	Mandatory	Default value	Type	Description
csv.array-element-delimiter	No	;	String	Array element delimiter string for separating array and row element values.
csv.escape-character	No	None	String	Escape character for escaping values
csv.null-literal	No	None	String	Null literal string that is interpreted as a null value.

Data Type Mapping

Currently, the CSV schema is always derived from table schema. Explicitly defining a CSV schema is not supported yet. Flink CSV format uses **jackson databind API** to parse and generate CSV string.

Table 1-9 Data type mapping

Flink SQL Type	CSV Type
CHAR/VARCHAR/STRING	String
BOOLEAN	Boolean
BINARY/VARBINARY	string with encoding: base64
DECIMAL	Number
TINYINT	Number
SMALLINT	Number
INT	Number
BIGINT	Number
FLOAT	Number
DOUBLE	Number
DATE	string with format: date
TIME	string with format: time
TIMESTAMP	string with format: date-time
INTERVAL	Number
ARRAY	array
ROW	object

Example

Use Kafka to send data and output the data to Print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'csv'
);


CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);
insert into printSink select * from kafkaSource;
```

Step 3 Insert the following data into the source Kafka topic:

```
202103251505050001,appShop,2021-03-25 15:05:05,500.00,400.00,2021-03-25 15:10:00,0003,Cindy,330108

202103241606060001,appShop,2021-03-24 16:06:06,200.00,180.00,2021-03-24 16:10:06,0001,Alice,330106
```

Step 4 Perform the following operations to view the data result in the **taskmanager.out** file:

1. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
2. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
3. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **.out** file, and view result logs.

```
+I[202103251505050001, appShop, 2021-03-25 15:05:05, 500.0, 400.0, 2021-03-25 15:10:00, 0003, Cindy,
330108]
+I[202103241606060001, appShop, 2021-03-24 16:06:06, 200.0, 180.0, 2021-03-24 16:10:06, 0001, Alice,
330106]
```

----End

1.4.6 Debezium

Function

Debezium is a Changelog Data Capture (CDC) tool that can stream changes in real-time from MySQL, PostgreSQL, Oracle, Microsoft SQL Server and many other databases into Kafka. Debezium provides a unified format schema for changelog and supports to serialize messages using JSON and Apache Avro.

Flink supports to interpret Debezium JSON and Avro messages as INSERT/UPDATE/DELETE messages into Flink SQL system. This is useful in many cases to leverage this feature, such as

- Synchronizing incremental data from databases to other systems
- Auditing logs
- Real-time materialized views on databases
- Temporal join changing history of a database table

Flink also supports to encode the INSERT/UPDATE/DELETE messages in Flink SQL as Debezium JSON or Avro messages, and emit to external systems like Kafka. However, currently Flink cannot combine UPDATE_BEFORE and UPDATE_AFTER into a single UPDATE message. Therefore, Flink encodes UPDATE_BEFORE and UDPATE_AFTER as DELETE and INSERT Debezium messages.

For details, see **Debezium Format**.

Supported Connectors

- Kafka
- FileSystem

Caveats

- Duplicate change events

  Under normal operating scenarios, the Debezium application delivers every change event exactly-once. Flink works pretty well when consuming Debezium produced events in this situation. However, Debezium application works in at-least-once delivery if any failover happens. That means, in the abnormal situations, Debezium may deliver duplicate change events to Kafka and Flink will get the duplicate events. This may cause Flink query to get wrong results or unexpected exceptions.

  Solution: Set **table.exec.source.cdc-events-duplicate** to **true** and define a primary key on this source.

  Framework will generate an additional stateful operator, and use the primary key to deduplicate the change events and produce a normalized changelog stream.

  For more information, see **Debezium documentation**.

- Consuming data produced by Debezium Postgres Connector

  If you are using Debezium Connector for PostgreSQL to capture the changes to Kafka, please make sure the **REPLICA IDENTITY** configuration of the monitored PostgreSQL table has been set to **FULL** which is by default **DEFAULT**. Otherwise, Flink SQL currently will fail to interpret the Debezium data.

  In FULL strategy, the UPDATE and DELETE events will contain the previous values of all the table's columns.

  In other strategies, the **before** field of UPDATE and DELETE events will only contain primary key columns or null if no primary key.

  You can change the **REPLICA IDENTITY** by running **ALTER TABLE <your-table-name> REPLICA IDENTITY FULL**.

Parameter Description

Flink provides **debezium-avro-confluent** and **debezium-json** formats to interpret Avro or Json messages produced by Debezium. Use format **debezium-avro-confluent** to interpret Debezium Avro messages and format **debezium-json** to interpret Debezium Json messages.

Table 1-10 Debezium Avro parameters

Parameter	Mandatory	Default Value	Data Type	Description
format	Yes	None	String	Format to be used. Set this parameter to **'debezium-avro-confluent'**.
debezium-avro-confluent.basic-auth.credentials-source	No	None	String	Basic auth credentials source for Schema Registry
debezium-avro-confluent.basic-auth.user-info	No	None	String	Basic auth user info for schema registry
debezium-avro-confluent.bearer-auth.credentials-source	No	None	String	Bearer auth credentials source for Schema Registry

Parameter	Mandatory	Default Value	Data Type	Description
debezium-avro-confluent.bearer-auth.token	No	None	String	Bearer auth token for Schema Registry
debezium-avro-confluent.properties	No	None	Map	Properties map that is forwarded to the underlying Schema Registry. This is useful for options that are not officially exposed via Flink config options. However, note that Flink options have higher precedence.
debezium-avro-confluent.ssl.keystore.location	No	None	String	Location/File of SSL keystore
debezium-avro-confluent.ssl.keystore.password	No	None	String	Password for SSL keystore
debezium-avro-confluent.ssl.truststore.location	No	None	String	Location/File of SSL truststore
debezium-avro-confluent.ssl.truststore.password	No	None	String	Password for SSL truststore
debezium-avro-confluent.subject	No	None	String	The Confluent Schema Registry subject under which to register the schema used by this format during serialization. By default, 'kafka' and 'upsert-kafka' connectors use '<topic_name>-value' or '<topic_name>-key' as the default subject name if this format is used as the value or key format. But for other connectors (e.g. 'filesystem'), the subject option is required when used as sink.
debezium-avro-confluent.url	No	None	String	The URL of the Confluent Schema Registry to fetch/register schemas.

Table 1-11 Debezium JSON parameters

Parameter	Mandatory	Default Value	Mandatory	Description
format	Yes	None	String	Format to be used. In this example.Set this parameter to **debezium-json**.
debezium-json.schema-include	No	false	Boolean	Whether the Debezium JSON messages contain the schema. When setting up Debezium Kafka Connect, enable the Kafka configuration **value.converter.schemas.enable** to include the schema in the message.
debezium-json.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. The default value is **false**, indicating that an error will be thrown. Fields are set to null in case of errors.
debezium-json.timestamp-format.standard	No	'SQL'	String	Input and output timestamp formats. Currently supported values are **SQL** and **ISO-8601**. • **SQL** will parse input timestamp in "yyyy-MM-dd HH:mm:ss.s{precision}" format, for example **2020-12-30 12:13:14.123** and output timestamp in the same format. • **ISO-8601** will parse input timestamp in "yyyy-MM-ddTHH:mm:ss.s{precision}" format, for example **2020-12-30T12:13:14.123** and output timestamp in the same format.

Parameter	Mandatory	Default Value	Mandatory	Description
debezium-json.map-null-key.mode	No	'FAIL'	String	Handling mode when serializing null keys for map data. Available values are as follows: ● **FAIL** will throw exception when encountering map value with null key. ● **DROP** will drop null key entries for map data. ● **LITERAL** replaces the empty key value in the map with a string constant. The string literal is defined by **debezium-json.map-null-key.literal** option.
debezium-json.map-null-key.literal	No	'null'	String	String literal to replace null key when **debezium-json.map-null-key.mode** is **LITERAL**.
debezium-json.encode.decimal-as-plain-number	No	false	Boolean	Encode all decimals as plain numbers instead of possible scientific notations. For example, **0.000000027** is encoded as **2.7E-8** by default, and will be written as **0.000000027** if set this parameter to **true**.

Metadata

The following format metadata can be exposed as read-only (VIRTUAL) columns in DDL.

Format metadata fields are only available if the corresponding connector forwards format metadata. Currently, only the Kafka connector is able to expose metadata fields for its value format.

Table 1-12 Metadata

Key	Data Type	Description
schema	STRING NULL	JSON string describing the schema of the payload. **Null** if the schema is not included in the Debezium record.

Key	Data Type	Description
ingestion-timestamp	TIMESTAMP_LTZ(3) NULL	The timestamp at which the connector processed the event. Corresponds to the **ts_ms** field in the Debezium record.
source.timestamp	TIMESTAMP_LTZ(3) NULL	The timestamp at which the source system created the event. Corresponds to the **source.ts_ms** field in the Debezium record.
source.database	STRING NULL	The originating database. Corresponds to the **source.db** field in the Debezium record if available.
source.schema	STRING NULL	The originating database schema. Corresponds to the **source.schema** field in the Debezium record if available.
source.table	STRING NULL	The originating database table. Corresponds to the **source.table** or **source.collection** field in the Debezium record if available.
source.properties	MAP<STRING, STRING> NULL	Map of various source properties. Corresponds to the **source** field in the Debezium record.

The following example shows how to access Canal metadata fields in Kafka:

```
CREATE TABLE KafkaTable (
  origin_ts TIMESTAMP(3) METADATA FROM 'value.ingestion-timestamp' VIRTUAL,
  event_time TIMESTAMP(3) METADATA FROM 'value.source.timestamp' VIRTUAL,
  origin_database STRING METADATA FROM 'value.source.database' VIRTUAL,
  origin_schema STRING METADATA FROM 'value.source.schema' VIRTUAL,
  origin_table STRING METADATA FROM 'value.source.table' VIRTUAL,
  origin_properties MAP<STRING, STRING> METADATA FROM 'value.source.properties' VIRTUAL,
  user_id BIGINT,
  item_id BIGINT,
  behavior STRING
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'earliest-offset',
```

```
  'value.format' = 'debezium-json'
);
```

Example

Use Kafka to parse Debezium JSON data and output the result to Print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.15. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'debezium-json'
);
CREATE TABLE printSink (
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
) WITH (
  'connector' = 'print'
);
insert into printSink select * from kafkaSource;
```

Step 3 Insert the data below into the appropriate Kafka topics. The data shows that the MySQL products table has four columns: **id**, **name**, **description**, and **weight**. This JSON message represents an update event on the products table, where the **weight** value of the row with id = 111 has been changed from 5.18 to 5.15.

```
{
  "before": {
    "id": 111,
    "name": "scooter",
    "description": "Big 2-wheel scooter",
    "weight": 5.18
  },
  "after": {
    "id": 111,
    "name": "scooter",
    "description": "Big 2-wheel scooter",
    "weight": 5.15
  },
  "source": {
    "version": "0.9.5.Final",
    "connector": "mysql",
    "name": "fullfillment",
    "server_id" :1,
    "ts_sec": 1629607909,
    "gtid": "mysql-bin.000001",
```

```
    "pos": 2238,"row": 0,
    "snapshot": false,
    "thread": 7,
    "db": "inventory",
    "table": "test",
    "query": null},
  "op": "u",
  "ts_ms": 1589362330904,
  "transaction": null
}
```

Step 4 Perform the following operations to view the data result in the **taskmanager.out** file:

1. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
2. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
3. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **.out** file, and view result logs.

```
-U[111, scooter, Big 2-wheel scooter, 5.18]
+U[111, scooter, Big 2-wheel scooter, 5.15]
```

----End

1.4.7 JSON

Function

The JSON format allows you to read and write JSON data based on a JSON schema. Currently, the JSON schema is derived from table schema. For details, see **JSON Format**.

Supported Connectors

- Kafka
- Upsert Kafka
- Elasticsearch

Parameters

Table 1-13

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. Set this parameter to **json**.
json.fail-on-missing-field	No	false	Boolean	Whether missing fields and rows will be skipped or failed. The default value is **false**, indicating that an error will be thrown.

Parameter	Mandatory	Default Value	Type	Description
json.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. The default value is **false**, indicating that an error will be thrown. Fields are set to null in case of errors.
json.timestamp-format.standard	No	'SQL'	String	Specify the input and output timestamp format for **TIMESTAMP** and **TIMESTAMP_LTZ** type. Currently supported values are **SQL** and **ISO-8601**: ● Option **SQL** will parse input **TIMESTAMP** values in "yyyy-MM-dd HH:mm:ss.s{precision}" format, e.g "2020-12-30 12:13:14.123", parse input **TIMESTAMP_LTZ** values in "yyyy-MM-dd HH:mm:ss.s{precision}'Z'" format, e.g "2020-12-30 12:13:14.123Z", and output timestamp in the same format. ● Option **ISO-8601** will parse input **TIMESTAMP** in "yyyy-MM-ddTHH:mm:ss.s{precision}" format, e.g "2020-12-30T12:13:14.123", parse input **TIMESTAMP_LTZ** in "yyyy-MM-ddTHH:mm:ss.s{precision}'Z'" format, e.g "2020-12-30T12:13:14.123Z", and output timestamp in the same format.

Parameter	Mandatory	Default Value	Type	Description
json.map-null-key.mode	No	'FALL'	String	Handling mode when serializing null keys for map data. Available values are as follows: • **FAIL** will throw exception when encountering map value with null key. • **DROP** will drop null key entries for map data. • **LITERAL** replaces the empty key value in the map with a string constant. The string literal is defined by **json.map-null-key.literal** option.
json.map-null-key.literal	No	'null'	String	String literal to replace null key when **json.map-null-key.mode** is **LITERAL**.
json.encode.decimal-as-plain-number	No	false	Boolean	Encode all decimals as plain numbers instead of possible scientific notations. For example, **0.000000027** is encoded as **2.7E-8** by default, and will be written as **0.000000027** if set this parameter to **true**.

Data Type Mapping

Currently, the JSON schema is always derived from table schema. Explicitly defining a JSON schema is not supported yet.

Flink JSON format uses **jackson databind API** to parse and generate JSON string.

The following table lists the type mapping from Flink type to JSON type.

Table 1-14 Data type mapping

Flink SQL Type	JSON Type
CHAR/VARCHAR/STRING	String
BOOLEAN	Boolean
BINARY/VARBINARY	string with encoding: base64
DECIMAL	Number
TINYINT	Number

Flink SQL Type	JSON Type
SMALLINT	Number
INT	Number
BIGINT	Number
FLOAT	Number
DOUBLE	Number
DATE	string with format: date
TIME	string with format: time
TIMESTAMP	string with format: date-time
TIMESTAMP_WITH_LOCAL_TIME_ZONE	string with format: date-time (with UTC time zone)
INTERVAL	Number
ARRAY	array
MAP / MULTISET	object
ROW	object

Example

In this example, data is read from a topic and written to another using a Kafka sink.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set an inbound rule for the security group to allow access of the queue and test the connectivity using the Kafka address. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job, select Flink 1.15, and allow DLI to save job logs in OBS. Use the following statement in the job and submit it:

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);
```

```
CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);
insert into printSink select * from kafkaSource;
```

Step 3 Insert the following data into the source Kafka topic:

```
{"order_id":"202103241000000001","order_channel":"webShop","order_time":"2021-03-24
10:00:00","pay_amount":100.0,"real_pay":100.0,"pay_time":"2021-03-24
10:02:03","user_id":"0001","user_name":"Alice","area_id":"330106"}

{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106"}
```

Step 4 Perform the following operations to view the data result in the **taskmanager.out** file:

1. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
2. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
3. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **.out** file, and view result logs.

```
+I[202103241000000001, webShop, 2021-03-24 10:00:00, 100.0, 100.0, 2021-03-24 10:02:03, 0001,
Alice, 330106]
+I[202103241606060001, appShop11, 2021-03-24 16:06:06, 200.0, 180.0, 2021-03-24 16:10:06, 0001,
Alice, 330106]
```

----End

1.4.8 Maxwell

Function

Maxwell is a Changelog Data Capture (CDC) tool that can stream changes in real-time from MySQL into Kafka and other streaming connectors. Maxwell provides a unified format schema for changelog and supports to serialize messages using JSON.

Flink supports to interpret Maxwell JSON messages as INSERT/UPDATE/DELETE messages into Flink SQL system. This is useful in many cases to leverage this feature,

such as:

- Synchronizing incremental data from databases to other systems
- Auditing logs
- Real-time materialized views on databases

- Temporal join changing history of a database table and so on

Flink also supports to encode the INSERT/UPDATE/DELETE messages in Flink SQL as Maxwell JSON messages, and emit to external systems like Kafka. However, currently Flink cannot combine UPDATE_BEFORE and UPDATE_AFTER into a single UPDATE message. Therefore, Flink encodes UPDATE_BEFORE and UDPATE_AFTER as DELETE and INSERT Maxwell messages.

For details, see **Maxwell Format**.

Supported Connectors

- Kafka
- FileSystem

Caveats

The Maxwell application allows to deliver every change event exactly-once. Flink works pretty well when consuming Maxwell produced events in this situation. If Maxwell application works in at-least-once delivery, it may deliver duplicate change events to Kafka and Flink will get the duplicate events. This may cause Flink query to get wrong results or unexpected exceptions. Thus, it is recommended setting job configuration **table.exec.source.cdc-events-duplicate** to **true** and define **PRIMARY KEY** on the source in this situation. Framework will generate an additional stateful operator, and use the primary key to deduplicate the change events and produce a normalized changelog stream.

Parameters

Table 1-15 Parameters

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. Set this parameter to **'maxwell-json'**.
maxwell-json.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. Fields are set to null in case of errors.

Parameter	Mandatory	Default Value	Type	Description
maxwell-json.timestamp-format.standard	No	'SQL'	String	Specify the input and output timestamp format. Currently supported values are **SQL** and **ISO-8601**: ● **SQL** will parse input timestamp in "yyyy-MM-dd HH:mm:ss.s{precision}" format, e.g '2020-12-30 12:13:14.123' and output timestamp in the same format. ● **ISO-8601** will parse input timestamp in "yyyy-MM-ddTHH:mm:ss.s{precision}" format, e.g '2020-12-30T12:13:14.123' and output timestamp in the same format.
maxwell-json.map-null-key.mode	No	'FAIL'	String	Specify the handling mode when serializing null keys for map data. Currently supported values are **FAIL**, **DROP**, and **LITERAL**: ● **FAIL** will throw exception when encountering map with null key. ● **DROP** will drop null key entries for map data. ● **LITERAL** will replace null key with string literal. The string literal is defined by **maxwell-json.map-null-key.literal**.
maxwell-json.map-null-key.literal	No	'null'	String	Specify string literal to replace null key when **maxwell-json.map-null-key.mode** is **LITERAL**.
maxwell-json.encode.decimal-as-plain-number	No	false	Boolean	Encode all decimals as plain numbers instead of possible scientific notations. By default, decimals may be written using scientific notation. For example, **0.000000027** is encoded as **2.7E-8** by default, and will be written as **0.000000027** if set this parameter to **true**.

Metadata

The following format metadata can be exposed as read-only (VIRTUAL) columns in DDL.

Table 1-16 Metadata

Key	Data Type	Description
database	STRING NULL	The originating database. Corresponds to the **database** field in the Maxwell record if available.
table	STRING NULL	The originating database table. Corresponds to the **table** field in the Maxwell record if available.
primary-key-columns	ARRAY<STRING> NULL	Array of primary key names. Corresponds to the **primary_key_columns** field in the Maxwell record if available.
ingestion-timestamp	TIMESTAMP_LTZ(3) NULL	The timestamp at which the connector processed the event. Corresponds to the **ts** field in the Maxwell record.

The following is an example of using metadata:

```
CREATE TABLE KafkaTable (
  origin_database STRING METADATA FROM 'value.database' VIRTUAL,
  origin_table STRING METADATA FROM 'value.table' VIRTUAL,
  origin_primary_key_columns ARRAY<STRING> METADATA FROM 'value.primary-key-columns' VIRTUAL,
  origin_ts TIMESTAMP(3) METADATA FROM 'value.ingestion-timestamp' VIRTUAL,
  user_id BIGINT,
  item_id BIGINT,
  behavior STRING
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'earliest-offset',
  'value.format' = 'maxwell-json'
);
```

Example

Use Kafka to send data and output the data to Print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the

Operation column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.15. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'maxwell-json'
);


CREATE TABLE printSink (
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
) WITH (
  'connector' = 'print'
);
insert into printSink select * from kafkaSource;
```

Step 3 Insert the data below into the appropriate Kafka topics (for details about the meaning of each field, see **Maxwell documentation**):

```
{
   "database":"test",
   "table":"e",
   "type":"insert",
   "ts":1477053217,
   "xid":23396,
   "commit":true,
   "position":"master.000006:800911",
   "server_id":23042,
   "thread_id":108,
   "primary_key": [1, "2016-10-21 05:33:37.523000"],
   "primary_key_columns": ["id", "c"],
   "data":{
     "id":111,
     "name":"scooter",
     "description":"Big 2-wheel scooter",
     "weight":5.15
   },
   "old":{
     "weight":5.18
   }
}
```

Step 4 Perform the following operations to view the data result in the **taskmanager.out** file:

1. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
2. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
3. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **.out** file, and view result logs.

```
+I[111, scooter, Big 2-wheel scooter, 5.15]
```

----End

1.4.9 Ogg

Function

Oracle GoldenGate (a.k.a ogg) is a comprehensive software package for real-time data capture and replication in heterogeneous IT environments. The product set enables high availability solutions, real-time data integration, transactional change data capture, data replication, transformations, and verification between operational and analytical enterprise systems. Ogg provides a format schema for changelog and supports to serialize messages using JSON.

Flink supports to interpret Ogg JSON as INSERT/UPDATE/DELETE messages into Flink SQL system. This is useful in many cases to leverage this feature, such as:

- Synchronizing incremental data from databases to other systems
- Auditing logs
- Real-time materialized views on databases
- Temporal join changing history of a database table and so on.

Flink also supports to encode the INSERT/UPDATE/DELETE messages in Flink SQL as Ogg JSON, and emit to external systems like Kafka. However, currently Flink cannot combine UPDATE_BEFORE and UPDATE_AFTER into a single UPDATE message. Therefore, Flink encodes UPDATE_BEFORE and UPDATE_AFTER as DELETE and INSERT Ogg messages.

Supported Connectors

- Kafka
- FileSystem

Parameter Description

Table 1-17 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
format	Yes	(none)	String	Specify what format to use, here should be **ogg-json**.
ogg-json.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. The default value is **false**, indicating that an error will be thrown. Fields are set to null in case of errors.

Parameter	Mandatory	Default Value	Data Type	Description
debezium-json.timestamp-format.standard	No	'SQL'	String	Input and output timestamp formats. Currently supported values are **SQL** and **ISO-8601**: ● **SQL** will parse input timestamp in "yyyy-MM-dd HH:mm:ss.s{precision}" format, for example **2020-12-30 12:13:14.123** and output timestamp in the same format. ● **ISO-8601** will parse input timestamp in "yyyy-MM-ddTHH:mm:ss.s{precision}" format, for example **2020-12-30T12:13:14.123** and output timestamp in the same format.
ogg-json.map-null-key.mode	No	'FAIL'	String	Handling mode when serializing null keys for map data. Currently supported values are **FAIL**, **DROP**, and **LITERAL**: ● Option **FAIL** will throw exception when encountering map with null key. ● Option **DROP** will drop null key entries for map data. ● Option **LITERAL** will replace null key with string literal. The string literal is defined by **ogg-json.map-null-key.literal**.
ogg-json.map-null-key.literal	No	'null'	String	Specify string literal to replace null key when **ogg-json.map-null-key.mode** is **LITERAL**.

Metadata

Table 1-18 Metadata

Key	Data Type	Description
table	STRING NULL	Contains fully qualified table name. The format of the fully qualified table name is **CATALOG NAME.SCHEMA NAME.TABLE NAME**.

Key	Data Type	Description
primary-keys	ARRAY<STRING> NULL	An array variable holding the column names of the primary keys of the source table. The **primary-keys** field is only included in the JSON output if the **includePrimaryKeys** configuration property is set to **true**.
ingestion-timestamp	TIMESTAMP_LTZ(6) NULL	The timestamp at which the connector processed the event. Corresponds to the **current_ts** field in the Ogg record.
event-timestamp	TIMESTAMP_LTZ(6) NULL	The timestamp at which the source system created the event. Corresponds to the **op_ts** field in the Ogg record.

The following example shows how to access Canal metadata fields in Kafka:

```
CREATE TABLE KafkaTable (
  origin_ts TIMESTAMP(3) METADATA FROM 'value.ingestion-timestamp' VIRTUAL,
  event_time TIMESTAMP(3) METADATA FROM 'value.event-timestamp' VIRTUAL,
  origin_table STRING METADATA FROM 'value.table' VIRTUAL,
  primary_keys ARRAY<STRING> METADATA FROM 'value.primary_keys' VIRTUAL,
  user_id BIGINT,
  item_id BIGINT,
  behavior STRING
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'earliest-offset',
  'value.format' = 'ogg-json'
);
```

Example

Use ogg-json to read Ogg records in Kafka and output them to Print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.15. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
) WITH (
  'connector' = 'kafka',
```

```
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'ogg-json'
);

CREATE TABLE printSink (
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
) WITH (
  'connector' = 'print'
);
insert into printSink select * from kafkaSource;
```

Step 3 Insert the data below into the appropriate Kafka topics. The data shows that the Oracle PRODUCTS table has four columns: **id**, **name**, **description**, and **weight**. This JSON message represents an update event on the PRODUCTS table, where the **weight** value of the row with id = 111 has been changed from 5.18 to 5.15.

```
{
  "before": {
    "id": 111,
    "name": "scooter",
    "description": "Big 2-wheel scooter",
    "weight": 5.18
  },
  "after": {
    "id": 111,
    "name": "scooter",
    "description": "Big 2-wheel scooter",
    "weight": 5.15
  },
  "op_type": "U",
  "op_ts": "2020-05-13 15:40:06.000000",
  "current_ts": "2020-05-13 15:40:07.000000",
  "primary_keys": [
    "id"
  ],
  "pos": "00000000000000000000143",
  "table": "PRODUCTS"
}
```

Step 4 Perform the following operations to view the data result in the **taskmanager.out** file:

1. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
2. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
3. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **.out** file, and view result logs.

```
-U[111, scooter, Big 2-wheel scooter, 5.18]
+U[111, scooter, Big 2-wheel scooter, 5.15]
```

----End

1.4.10 ORC

Function

The Apache ORC format allows to read and write ORC data. For details, see **ORC Format**.

Supported Connectors

- FileSystem

Parameter Description

Table 1-19 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
format	Yes	None	String	Specify what format to use, here should be **orc**.

ORC format also supports table properties from **Table properties**. For example, you can configure **orc.compress=SNAPPY** to enable snappy compression.

Data Type Mapping

ORC format type mapping is compatible with Apache Hive. The following table lists the type mapping from Flink type to ORC type.

Table 1-20 Data type mapping

Flink SQL Type	ORC Physical Type	ORC Logical Type
CHAR	bytes	CHAR
VARCHAR	bytes	VARCHAR
STRING	bytes	STRING
BOOLEAN	long	BOOLEAN
BYTES	bytes	BINARY
DECIMAL	decimal	DECIMAL
TINYINT	long	BYTE
SMALLINT	long	SHORT
INT	long	INT

Flink SQL Type	ORC Physical Type	ORC Logical Type
BIGINT	long	LONG
FLOAT	double	FLOAT
DOUBLE	double	DOUBLE
DATE	long	DATE
TIMESTAMP	timestamp	TIMESTAMP
ARRAY	-	LIST
MAP	-	MAP
ROW	-	STRUCT

Example

Use Kafka to send data and output the data to Print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and enable checkpointing. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic-pattern' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'csv'
);


CREATE TABLE sink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
```

```
  area_id string
) WITH (
  'connector' = 'filesystem',
  'format' = 'orc',
  'path' = 'obs://xx'
);
insert into sink select * from kafkaSource;
```

Step 3 Insert the following data into the source Kafka topic:

```
202103251505050001,appshop,2021-03-25 15:05:05,500.00,400.00,2021-03-25 15:10:00,0003,Cindy,330108

202103241606060001,appShop,2021-03-24 16:06:06,200.00,180.00,2021-03-24 16:10:06,0001,Alice,330106
```

Step 4 Read the ORC file in the OBS path configured in the sink table. The data results are as follows:

```
202103251202020001, miniAppShop, 2021-03-25 12:02:02, 60.0, 60.0, 2021-03-25 12:03:00, 0002, Bob,
330110

202103241606060001, appShop, 2021-03-24 16:06:06, 200.0, 180.0, 2021-03-24 16:10:06, 0001, Alice,
330106
```

----End

1.4.11 Parquet

Function

The Apache Parquet format allows to read and write Parquet data. For details, see **Parquet Format**.

Supported Connectors

- FileSystem

Parameter Description

Table 1-21 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
format	Yes	None	String	Specify what format to use, here should be **parquet**.

Parameter	Mandatory	Default Value	Data Type	Description
parquet.utc-timezone	No	false	Boolean	Use UTC timezone or local timezone to the conversion between epoch time and LocalDateTime. Hive 0.x/1.x/2.x use local timezone. But Hive 3.*x* use UTC timezone.

Data Type Mapping

Currently, Parquet format type mapping is compatible with Apache Hive, but different with Apache Spark:

- Timestamp: mapping timestamp type to int96 whatever the precision is.
- Decimal: mapping decimal type to fixed length byte array according to the precision.

The following table lists the type mapping from Flink type to Parquet type.

Note that currently only writing is supported for composite data types (Array, Map, and Row), while reading is not supported.

Table 1-22 Data type mapping

Flink SQL Type	Parquet Type	Parquet Logical Type
CHAR/VARCHAR/STRING	BINARY	UTF8
BOOLEAN	BOOLEAN	-
BINARY/VARBINARY	BINARY	-
DECIMAL	FIXED_LEN_BYTE_ARRAY	DECIMAL
TINYINT	INT32	INT_8
SMALLINT	INT32	INT_16
INT	INT32	-
BIGINT	INT64	-

Flink SQL Type	Parquet Type	Parquet Logical Type
FLOAT	FLOAT	-
DOUBLE	DOUBLE	-
DATE	INT32	DATE
TIME	INT32	TIME_MILLIS
TIMESTAMP	INT96	-
ARRAY	-	LIST
MAP	-	MAP
ROW	-	STRUCT

Example

Use Kafka to send data and output the data to Print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and enable checkpointing. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic-pattern' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);


CREATE TABLE sink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
```

```
  area_id string
) WITH (
  'connector' = 'filesystem',
  'format' = 'parquet',
  'path' = 'obs://xx'
);
insert into sink select * from kafkaSource;
```

Step 3 Insert the following data into the source Kafka topic:

```
202103251505050001,appShop,2021-03-25 15:05:05,500.00,400.00,2021-03-25 15:10:00,0003,Cindy,330108

202103241606060001,appShop,2021-03-24 16:06:06,200.00,180.00,2021-03-24 16:10:06,0001,Alice,330106
```

Step 4 Read the Parquet file in the OBS path configured in the sink table. The data results are as follows:

```
202103251202020001, miniAppShop, 2021-03-25 12:02:02, 60.0, 60.0, 2021-03-25 12:03:00, 0002, Bob,
330110

202103241606060001, appShop, 2021-03-24 16:06:06, 200.0, 180.0, 2021-03-24 16:10:06, 0001, Alice,
330106
```

----End

1.4.12 Raw

Function

The Raw format allows to read and write raw (byte based) values as a single column.

NOTE

- This format encodes null values as null of byte[] type. This may have limitation when used in **upsert-kafka**, because **upsert-kafka** treats null values as a tombstone message (DELETE on the key). Therefore, we recommend avoiding using **upsert-kafka** connector and the **raw** format as a **value.format** if the field can have a null value.
- The raw format connector is built-in, no additional dependencies are required. For details, see **Raw Format**.

Supported Connectors

- Kafka
- Upsert Kafka
- FileSystem

Parameter Description

Table 1-23

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. Set this parameter to **raw**.

Parameter	Mandatory	Default Value	Type	Description
raw.charset	No	UTF-8	String	Charset to encode the text string.
raw.endianness	No	big-endian	String	Endianness to encode the bytes of numeric value. Valid values are **big-endian** and **little-endian**. You can search for **endianness** for more details.

Data Type Mapping

The table below details the SQL types the format supports, including details of the serializer and deserializer class for encoding and decoding.

Table 1-24 Data type mapping

Flink SQL Type	Value
CHAR/VARCHAR/STRING	A UTF-8 (by default) encoded text string. The encoding charset can be configured by **raw.charse**.
BINARY / VARBINARY / BYTES	The sequence of bytes itself.
BOOLEAN	A single byte to indicate boolean value, **0** means **false**, **1** means **true**.
TINYINT	A single byte of the signed number value.
SMALLINT	Two bytes with big-endian (by default) encoding. The endianness can be configured by **raw.endianness**.
INT	Four bytes with big-endian (by default) encoding. The endianness can be configured by **raw.endianness**.
BIGINT	Eight bytes with big-endian (by default) encoding. The endianness can be configured by **raw.endianness**.

Flink SQL Type	Value
FLOAT	Four bytes with IEEE 754 format and big-endian (by default) encoding. The endianness can be configured by **raw.endianness**.
DOUBLE	Eight bytes with IEEE 754 format and big-endian (by default) encoding. The endianness can be configured by **raw.endianness**.
RAW	The sequence of bytes serialized by the underlying TypeSerializer of the RAW type.

Example

Use Kafka to send data and output the data to Print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.15. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  log string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'raw'
);

CREATE TABLE printSink (
  log string
) WITH (
  'connector' = 'print'
);
insert into printSink select * from kafkaSource;
```

Step 3 Insert the following data to the corresponding topic in Kafka:

```
47.29.201.179 - - [28/Feb/2019:13:17:10 +0000] "GET /?p=1 HTTP/2.0" 200 5316 "https://domain.com/?
p=1" "Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.119
Safari/537.36" "2.75"
```

Step 4 Perform the following operations to view the data result in the **taskmanager.out** file:

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

2. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
3. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **.out** file, and view result logs.

```
+I[47.29.201.179 - - [28/Feb/2019:13:17:10 +0000] "GET /?p=1 HTTP/2.0" 200 5316 "https://domain.com/?
p=1" "Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.119
Safari/537.36" "2.75"]
```

----End

1.5 Connectors

1.5.1 Overview

Table Type

- Source table: A source table is the data input table for Flink jobs, such as real-time streaming data input from Kafka, etc.
- Dimension Table: An auxiliary table for the data source table, used to enrich and expand the data in the source table. In Flink jobs, because the data collected by the data collection end is often limited, the required dimension information needs to be completed before data analysis can be performed, and the dimension table represents the data source that stores dimension information. Common user dimension tables include MySQL, Redis, etc.
- Result table: The result data table output by the Flink job, which writes each real-time processed data into the target storage, such as MySQL, HBase, and other databases.

Example:

Flink real-time consumes user order data from the Kafka source table, associates the product ID with the dimension table through Redis to obtain the product category, calculates the sales amount of different categories of products, and writes the calculation results into the RDS (Relational Database Service, such as MySQL) result table.

Table information is as follows:

- Source table: Order data table, including user ID, product ID, order ID, order amount, and other information.
- Dimension table: User information table, including product ID and product category information.
- Result table: Statistics of order sales amount data by product category.

The job first reads real-time order data from the order data source table, associates the order data stream with the product category information dimension table, then aggregates and calculates the total order amount, and finally writes the statistical results into the result table.

In this example, the order table serves as the driving source table input, the product category information table serves as the static dimension table, and the statistical result table serves as the final output of the job.

Supported Connectors

Table 1-25 Supported connectors

Connector	Source Table	Dimension Table	Result Table
BlackHole	Not supported	Not supported	Supported
ClickHouse	Not supported	Not supported	Supported
DataGen	Supported	Not supported	Not supported
Doris	Supported	Supported	Supported
GaussDB(DWS)	Supported	Supported	Supported
Elasticsearch	Not supported	Not supported	Supported
FileSystem	Supported	Not supported	Supported
HBase	Supported	Supported	Supported
Hive	Supported	Supported	Supported
JDBC	Supported	Supported	Supported
Kafka	Supported	Not supported	Supported
Print	Not supported	Not supported	Supported
Redis	Supported	Supported	Supported
Upsert Kafka	Supported	Not supported	Supported

1.5.2 BlackHole

Function

The BlackHole connector allows for swallowing all input records. It is designed for high-performance testing and UDF output. It is not a substantive sink. The BlackHole result table is a built-in connector.

For example, if an error is reported when you register a result table of another type, but you are not sure whether it is caused by a system fault or an invalid setting of the **WITH** parameter for the result table, you can change the value of **connector** to **blackhole** and click **Run**. If no error is reported, the system is normal. You must check the settings of the **WITH** parameter.

Table 1-26 Supported types

Type	Description
Supported Table Types	Result table

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.

Syntax

```
create table blackhole_table (
 attr_name attr_type (',' attr_name attr_type) *
) with (
 'connector' = 'blackhole'
);
```

Parameter Description

Table 1-27 Parameter

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **blackhole**.

Example

The DataGen source table generates data, and the BlackHole result table receives the data.

```
create table datagenSource (
 user_id string,
 user_name string,
 user_age int
) with (
 'connector' = 'datagen',
 'rows-per-second'='1'
);
create table blackholeSink (
 user_id string,
 user_name string,
 user_age int
) with (
 'connector' = 'blackhole'
);
insert into blackholeSink select * from datagenSource;
```

1.5.3 ClickHouse

Function

DLI has the capability to export data from Flink jobs to the ClickHouse database. However, it only supports exporting data to result tables.

ClickHouse is a column-based database oriented to online analysis and processing. It supports SQL query and provides good query performance. The aggregation analysis and query performance based on large and wide tables is excellent, which is one order of magnitude faster than other analytical databases. For details, see **Using ClickHouse from Scratch**.

Table 1-28 Supported types

Type	Description
Supported Table Types	Result table

Prerequisites

- Your jobs are running on a dedicated queue of DLI.
- You have established an enhanced datasource connection to ClickHouse and set the port in the security group rule of the ClickHouse cluster as needed.

  For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.

  For details about how to configure security group rules, see **Security Group Overview**.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- When you create a ClickHouse cluster for MRS, set the cluster version to MRS 3.1.0 or later.
- The ClickHouse result table does not support table data deletion.
- Flink supports the following data types: string, tinyint, smallint, int, bigint, float, double, date, timestamp, decimal, and array.

  The array supports only the int, bigint, string, float, and double data types.

Syntax

```
create table clickhouseSink (
  attr_name attr_type
```

```
  (',' attr_name attr_type)*
)
with (
  'type' = 'clickhouse',
  'url' = '',
  'table-name' = ''
);
```

Parameter Description

Table 1-29 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Result table type. Set this parameter to **clickhouse**.

Parameter	Mandatory	Default Value	Data Type	Description
url	Yes	None	String	ClickHouse URL It is in the format of **jdbc:clickhouse://***ClickHouseBalancer instance service IP address 1***:***ClickHouseBalancer port***,***ClickHouseBalancer instance service IP address 2***:***ClickHouseBalancer port***/***Database name*. ● IP address of a ClickHouseBalancer instance: Log in to the MRS console and choose **Clusters** > **Active Clusters** in the navigation pane. Click a cluster name, and choose **Components** > **ClickHouse** > **Instances** to obtain the business IP address of the ClickHouseBalancer instance. ● ClickHouseBalancer port number: Log in to the MRS console and choose **Clusters** > **Active Clusters** in the navigation pane. Click a cluster name, and choose **Components** > **ClickHouse** > **Service Configuration**. On the **Service Configuration** page, select **ClickHouseBalancer** from the **All Roles** drop-down list. If the MRS cluster does not have Kerberos authentication enabled, search for **lb_http_port** and set it (defaults to **21425**). If Kerberos authentication is enabled, search for **lb_https_port** and set it (defaults to **21426**). ● The database name is the name of the database created for the ClickHouse cluster. If the database name does not exist, there is no need to specify it. ● You can configure multiple IP addresses for ClickHouseBalancer instances to avoid single points of failure (SPOFs) of the instances. ● If the MRS cluster has Kerberos authentication enabled, you also

Parameter	Mandatory	Default Value	Data Type	Description
				need to add the **ssl** and **sslmode** request parameters to the URL, setting **ssl** to **true** and **sslmode** to **none**. Refer to **Example 2** for an example.
table-name	Yes	None	String	ClickHouse table name.
driver	No	ru.yandex.clickhouse.ClickHouseDriver	String	Driver required for connecting to the database. If you do not set this parameter, the automatically extracted driver will be used, which defaults to **ru.yandex.clickhouse.ClickHouseDriver**.
username	No	None	String	Username for accessing the ClickHouse database. This parameter is mandatory when Kerberos authentication is enabled for the MRS cluster.
password	No	None	String	Password for accessing the ClickHouse database. This parameter is mandatory when Kerberos authentication is enabled for the MRS cluster.
sink.buffer-flush.max-rows	No	100	Integer	Maximum number of rows to be updated when data is written. The default value is **100**.
sink.buffer-flush.interval	No	1s	Duration	Interval for data update. The unit can be ms, milli, millisecond/s, sec, second/min, or minute. The default value is **1s**. Value **0** indicates that data is not updated.
sink.max-retries	No	3	Integer	Maximum number of retries for writing data to the result table. The default value is **3**.

Example

- **Example 1: Read data from Kafka and insert the data into ClickHouse. (The ClickHouse version is 21.3.4.25 of MRS, and Kerberos authentication is not enabled for the MRS cluster):**
    a. Create an enhanced datasource connection in the VPC and subnet where ClickHouse and Kafka clusters locate, and bind the connection to the

required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.

b. Set ClickHouse and Kafka cluster security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the ClickHouse address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.

c. Use the ClickHouse client to connect to the ClickHouse server and run the following command to query other environment parameters such as the cluster identifier:

For details, see **Using ClickHouse from Scratch**.

```
select cluster,shard_num,replica_num,host_name from system.clusters;
```

The following information is displayed:

```
┌─cluster────────┬────────┬─shard_num─┐
│ default_cluster │    1   │         1 │
│ default_cluster │    1   │         2 │
└────────────────┴────────┴───────────┘
```

Run the following command to create database **flink** on a node of the ClickHouse cluster based on the obtained cluster ID, for example, **default_cluster**:

```
CREATE DATABASE flink ON CLUSTER default_cluster;
```

d. Run the following command to create the ReplicatedMergeTree table named **order** on the node of cluster **default_cluster** and on database **flink**:

```
CREATE TABLE flink.order ON CLUSTER default_cluster(order_id String,order_channel String,order_time String,pay_amount Float64,real_pay Float64,pay_time String,user_id String,user_name String,area_id String) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/flink/order', '{replica}')ORDER BY order_id;
```

e. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses the DMS Kafka data source and the ClickHouse result table.

Change the values of the parameters in bold as needed in the following script.

```
create table orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

create table clickhouseSink(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
```

```
  user_name string,
  area_id string
) with (
  'connector' = 'clickhouse',
  'url' = 'jdbc:clickhouse://ClickhouseAddress1:ClickhousePort,ClickhouseAddress2:ClickhousePort/flink',
  'username' = 'username',
  'password' = 'password',
  'table-name' = 'order',
  'sink.buffer-flush.max-rows' = '10',
  'sink.buffer-flush.interval' = '3s'
);

insert into clickhouseSink select * from orders;
```

f. Connect to the Kafka cluster and insert the following test data into DMS Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

g. Use the ClickHouse client to connect to the ClickHouse and run the following command to query the data written to table **order** in database **flink**:

```
select * from flink.order;
```

The query result is as follows:

```
202103241000000001 webShop 2021-03-24 10:00:00 100 100 2021-03-24 10:02:03 0001 Alice 330106

202103241606060001 appShop 2021-03-24 16:06:06 200 180 2021-03-24 16:10:06 0001 Alice 330106

202103251202020001 miniAppShop 2021-03-25 12:02:02 60 60 2021-03-25 12:03:00 0002 Bob 330110
```

- **Example 2: Read data from Kafka and insert the data into ClickHouse. The procedure is as follows (The ClickHouse version is 21.3.4.25 of MRS, and Kerberos authentication is enabled for the MRS cluster):**

a. Create an enhanced datasource connection in the VPC and subnet where ClickHouse and Kafka clusters locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.

b. Set ClickHouse and Kafka cluster security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the ClickHouse address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.

c. Use the ClickHouse client to connect to the ClickHouse server and run the following command to query other environment parameters such as the cluster identifier:

Refer to **Using ClickHouse from Scratch** for more details.

```
select cluster,shard_num,replica_num,host_name from system.clusters;
```

The following information is displayed:

```
┌─cluster─────────┬─────┬─shard_num─┐
│ default_cluster │  1  │         1 │
```

```
| default_cluster |    1   |          2 |
└───────────────┴─────┴─────────┘
```

Run the following command to create database **flink** on a node of the ClickHouse cluster based on the obtained cluster ID, for example, **default_cluster**:

```
CREATE DATABASE flink ON CLUSTER default_cluster;
```

d. Run the following command to create the ReplicatedMergeTree table named **order** on the node of cluster **default_cluster** and on database **flink**:

```
CREATE TABLE flink.order ON CLUSTER default_cluster(order_id String,order_channel
String,order_time String,pay_amount Float64,real_pay Float64,pay_time String,user_id
String,user_name String,area_id String) ENGINE = ReplicatedMergeTree('/clickhouse/tables/
{shard}/flink/order', '{replica}')ORDER BY order_id;
```

e. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses the Kafka data source and the ClickHouse result table.

Change the values of the parameters in bold as needed in the following script.

```
create table orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

create table clickhouseSink(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) with (
  'connector' = 'clickhouse',
  'url' = 'jdbc:clickhouse://
ClickhouseAddress1:ClickhousePort,ClickhouseAddress2:ClickhousePort/flink?
ssl=true&sslmode=none',
  'table-name' = 'order',
  'username' = 'username',
  'password' = 'password', --Key in the DEW secret
  'sink.buffer-flush.max-rows' = '10',
  'sink.buffer-flush.interval' = '3s',
  'dew.endpoint'='kms.xx.myhuaweicloud.com', --Endpoint information for the DEW service
being used
  'dew.csms.secretName'='xx', --Name of the DEW shared secret
  'dew.csms.decrypt.fields'='password', --The password field value must be decrypted and
replaced using DEW secret management.
  'dew.csms.version'='v1'
);
```

```
insert into clickhouseSink select * from orders;
```

f. Connect to the Kafka cluster and insert the following test data into Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

g. Use the ClickHouse client to connect to the ClickHouse and run the following command to query the data written to table **order** in database **flink**:

```
select * from flink.order;
```

The query result is as follows:

```
202103241000000001 webShop 2021-03-24 10:00:00 100 100 2021-03-24 10:02:03 0001 Alice 330106

202103241606060001 appShop 2021-03-24 16:06:06 200 180 2021-03-24 16:10:06 0001 Alice 330106

202103251202020001 miniAppShop 2021-03-25 12:02:02 60 60 2021-03-25 12:03:00 0002 Bob 330110
```

1.5.4 DataGen

Function

DataGen is used to generate random data for debugging and testing.

Table 1-30 Supported types

Type	Description
Supported Table Types	Source table

Caveats

- When you create a DataGen table, the table field type cannot be Array, Map, or Row. You can use **COMPUTED COLUMN** in **CREATE TABLE** to construct similar functions.
- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.

Syntax

```
create table dataGenSource(
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' WATERMARK FOR rowtime_column_name AS watermark-strategy_expression)
)
with (
  'connector' = 'datagen'
);
```

Parameter Description

Table 1-31 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **datagen**.
rows-per-second	No	10000	Long	Rows per second to control the emit rate.
number-of-rows	No	None	Long	The total number of rows to emit. By default, the total number of rows of generated data is not limited. If the generator type is a sequence generator, data generation will stop when either the maximum number of rows has been reached or the sequence number has reached its end value.

Parameter	Mandatory	Default Value	Data Type	Description
fields.#.kind	No	random	String	Generator of the **#** field. The **#** field must be an actual field in the DataGen table. Replace **#** with the corresponding field name. The meanings of the **#** field for other parameters are the same. The value can be **sequence** or **random**. ● **random** is the default value, indicating an unbounded random generator. You can use the **fields.#.max** and **fields.#.min** parameters to specify the maximum and minimum values that are randomly generated. If the specified field type is char, varchar, or string, you can also use the **fields.#.length** parameter to specify the length. If the specified field type is a timestamp, you can use the **fields.#.max-past** parameter to specify the maximum offset from the current time towards the past. ● **sequence** represents a bounded sequence generator. You can specify the start and end values of the sequence using **fields.#.start** and **fields.#.end**. Once the sequence number reaches the end value, no more data will be generated.
fields.#.min	No	Minimum value of the field type specified by **#**	Field type specified by **#**	This parameter is valid only when **fields.#.kind** is set to **random**. Minimum value of the random generator. It applies only to numeric field types specified by **#**.
fields.#.max	No	Maximum value of the field type specified by **#**	Field type specified by **#**	This parameter is valid only when **fields.#.kind** is set to **random**. Maximum value of the random number. It applies only to numeric field types specified by **#**.

Parameter	Mandatory	Default Value	Data Type	Description
fields.#.max-past	No	0	Duration	This parameter is valid only when **fields.#.kind** is set to **random**. The random generator generates a maximum offset from the current time towards the past. The **#** specified field is only applicable to timestamp types.
fields.#.length	No	100	Integer	This parameter is valid only when **fields.#.kind** is set to **random**. Length of the characters generated by the random generator. It applies only to char, varchar, and string types specified by **#**.
fields.#.start	No	None	Field type specified by **#**	This parameter is valid only when **fields.#.kind** is set to **sequence**. Start value of a sequence generator.
fields.#.end	No	None	Field type specified by **#**	This parameter is valid only when **fields.#.kind** is set to **sequence**. End value of a sequence generator.

Example

Create a Flink OpenSource SQL job. Run the following script to generate random data through the DataGen table and output the data to the Print result table.

```
create table dataGenSource(
  user_id string,
  amount int
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1', --Generates a piece of data per second.
  'fields.user_id.kind' = 'random', --Specifies a random generator for the user_id field.
  'fields.user_id.length' = '3' --Limits the length of the user_id field to 3.
  'fields.amount.kind' = 'sequence', --Specify a sequence generator for the amount field.
  'fields.amount.start' = '1', --Start value of the amount field
  'fields.amount.end' = '1000' --End value of the amount field
);

create table printSink(
  user_id string,
  amount int
) with (
  'connector' = 'print'
);

insert into printSink select * from dataGenSource;
```

After the job is submitted, the job status changes to **Running**. You can perform the following operations of either method to view the output result:

- Method 1:
  a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  b. Locate the row that contains the target Flink job, and choose **More** > **FlinkUI** in the **Operation** column.
  c. On the Flink UI, choose **Task Managers**, click the task name, and select **Stdout** to view job logs.
- Method 2: If you select **Save Job Log** on the **Running Parameters** tab before submitting the job, perform the following operations:
  a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
  c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the file whose name contains **taskmanager.out**, and view result logs.

1.5.5 Doris

1.5.5.1 Overview

The Flink Doris Connector can support operations (read, insert, modify, delete) data stored in Doris through Flink. For details, see **Flink Doris Connector**.

NOTE

Only tables in the Unique Key model can be modified or deleted.

Table 1-32 Supported types

Type	Description
Supported Table Types	Source table, dimension table, and result table

1.5.5.2 Source Table

Function

Flink SQL jobs read from the Doris source table.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to Doris, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.

  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.
  - For details about how to configure security group rules, see **Security Group Overview**.
- **If MRS Doris is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.**

  For details, see **Modifying Host Information**.
- Kerberos authentication is disabled for the cluster (the cluster is in normal mode)

  After connecting to Doris as user **admin**, create a role with administrator permissions, and bind the role to the user.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- Kerberos authentication is disabled for the cluster (the cluster is in normal mode)
- Doris table names are case sensitive.
- When Doris of CloudTable is used, set the port number in the **fenodes** field to **8030**, for example, *xx*:**8030**. In addition, enable ports **8030**, **8040**, and **9030** in the security group.
- After HTTPS is enabled, add the following configuration parameters to the **with** clause for creating a table:
  - **'doris.enable.https' = 'true'**
  - **'doris.ignore.https.ca' = 'true'**

Syntax

```
create table dorisSource (
  attr_name attr_type
  (',' attr_name attr_type)*
 )
with (
  'connector' = 'doris',
  'fenodes' = 'FE_IP:PORT,FE_IP:PORT,FE_IP:PORT',
  'table.identifier' = 'database.table',
  'username' = 'dorisUsername',
  'password' = 'dorisPassword'
);
```

Parameter Description

Shared configuration

Parameter	Default Value	Mandatory	Parameter Type Description
fenodes	--	Yes	IP address and port number of the Doris FE. Use commas (,) to separate them for multiple instances. To obtain the port number, log in to FusionInsight Manager, choose **Cluster** > **Services** > **Doris** > **Configurations**, and search for **http**. Search for **https** instead if HTTPS is enabled.
table.identifier	--	Yes	Doris table name, for example, **db.tbl**.
username	--	Yes	User name for accessing Doris.
password	--	Yes	Password for accessing Doris.
doris.request.retries	3	No	Number of retry times for sending requests to Doris.
doris.request.connect.timeout.ms	30000	No	Connection timeout interval for sending requests to Doris.
doris.request.read.timeout.ms	30000	No	Read timeout interval for sending requests to Doris.
doris.request.query.timeout.s	3600	No	Timeout interval for querying Doris. The default value is **1** hour. The value **-1** indicates that there is no timeout limit.
doris.request.tablet.size	Integer.MAX_VALUE	No	Number of Doris Tablets corresponding to a partition. The smaller the value set, the more partitions will be generated. This increases the degree of parallelism in Flink, but puts more pressure on Doris.
doris.batch.size	1024	No	Maximum number of rows to read from BE at a time. Increasing this value reduces the number of times Flink needs to establish a connection with Doris when reading data from BE, thereby reducing the additional time overhead caused by network latency.
doris.exec.mem.limit	2147483648	No	Memory limit for a single query, with a default value of 2 GB, in bytes.

Parameter	Default Value	Mandatory	Parameter Type Description
doris.deserialize.arrow.async	FALSE	No	Whether to support asynchronous conversion of Arrow format to RowBatch required for flink-doris-connector iteration.
doris.deserialize.queue.size	64	No	The internal processing queue for asynchronous conversion of Arrow format is effective when **doris.deserialize.arrow.async** is set to **true**.
doris.read.field	--	No	The column name list for reading from Doris tables, with multiple columns separated by commas.
doris.filter.query	--	No	The expression used to filter the data to be read, which is passed through to Doris. Doris uses this expression to filter the source data.

Example

This example reads data from a Doris source table and inputs it into the Print connector.

1. Create an enhanced datasource connection in the VPC and subnet where Doris locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**. Add MRS host information for the enhanced datasource connection. For details, see **Modifying Host Information**.
2. Set Doris security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Doris address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
3. Create a Doris table and insert 10 data records into the table. The creation statement is as follows:
```
CREATE TABLE IF NOT EXISTS dorisdemo
(
  `user_id` varchar(10) NOT NULL,
  `city` varchar(10),
  `age` int,
  `gender` int
)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 10;

INSERT INTO dorisdemo VALUES ('user1', 'city1', 20, 1);
INSERT INTO dorisdemo VALUES ('user2', 'city2', 21, 0);
INSERT INTO dorisdemo VALUES ('user3', 'city3', 22, 1);
INSERT INTO dorisdemo VALUES ('user4', 'city4', 23, 0);
INSERT INTO dorisdemo VALUES ('user5', 'city5', 24, 1);
INSERT INTO dorisdemo VALUES ('user6', 'city6', 25, 0);
INSERT INTO dorisdemo VALUES ('user7', 'city7', 26, 1);
INSERT INTO dorisdemo VALUES ('user8', 'city8', 27, 0);
```

```
INSERT INTO dorisdemo VALUES ('user9', 'city9', 28, 1);
INSERT INTO dorisdemo VALUES ('user10', 'city10', 29, 0);
```

4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script reads from the Doris table and prints the output.

```
CREATE TABLE dorisDemo (
  `user_id` String NOT NULL,
  `city` String,
  `age` int,
  `gender` int
) with (
  'connector' = 'doris',
  'fenodes' = 'FE_IP:PORT,FE_IP:PORT,FE_IP:PORT',
  'table.identifier' = 'demo.dorisdemo',
  'username' = 'dorisUser',
  'password' = 'dorisPassword',
  'doris.request.retries'='3',
  'doris.batch.size' = '100'
);

CREATE TABLE print (
  `user_id` String NOT NULL,
  `city` String,
  `age` int,
  `gender` int
) with (
  'connector' = 'print'
);

insert into print select * from dorisDemo;
```

5. View the data in the Print result table.

```
+I[user5, city5, 24, 1]
+I[user4, city4, 23, 0]
+I[user3, city3, 22, 1]
+I[user10, city10, 29, 0]
+I[user6, city6, 25, 0]
+I[user1, city1, 20, 1]
+I[user9, city9, 28, 1]
+I[user7, city7, 26, 1]
+I[user8, city8, 27, 0]
+I[user2, city2, 21, 0]
```

1.5.5.3 Result Table

Function

Flink SQL jobs write to the Doris result table.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to Doris, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.
  - For details about how to configure security group rules, see **Security Group Overview**.
- **If MRS Doris is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.**

  For details, see **Modifying Host Information**.

- Kerberos authentication is disabled for the cluster (the cluster is in normal mode)

  After connecting to Doris as user **admin**, create a role with administrator permissions, and bind the role to the user.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- Kerberos authentication is disabled for the cluster (the cluster is in normal mode)
- Doris table names are case sensitive.
- When Doris of CloudTable is used, set the port number in the **fenodes** field to **8030**, for example, *xx*:**8030**. In addition, enable ports **8030**, **8040**, and **9030** in the security group.
- After HTTPS is enabled, add the following configuration parameters to the **with** clause for creating a table:
  - **'doris.enable.https' = 'true'**
  - **'doris.ignore.https.ca' = 'true'**
- On the **Running Parameters** tab of the Flink job editing page, check **Enable Checkpointing**. Otherwise, data can be written to the Doris result table, and the delay in writing to Doris depends on the value set for **Checkpoint Interval**.

Syntax

```
create table dorisSource (
  attr_name attr_type
  (',' attr_name attr_type)*
 )
with (
  'connector' = 'doris',
  'fenodes' = 'FE_IP:PORT,FE_IP:PORT,FE_IP:PORT',
  'table.identifier' = 'database.table',
  'username' = 'dorisUsername',
  'password' = 'dorisPassword'
);
```

Parameter Description

Shared configuration

Parameter	Default Value	Mandatory	Parameter Type Description
fenodes	--	Yes	IP address and port number of the Doris FE. Use commas (,) to separate them for multiple instances. To obtain the port number, log in to FusionInsight Manager, choose **Cluster** > **Services** > **Doris** > **Configurations**, and search for **http**. Search for **https** instead if HTTPS is enabled.
table.identifier	--	Yes	Doris table name, for example, **db.tbl**.
username	--	Yes	User name for accessing Doris.
password	--	Yes	Password for accessing Doris.
sink.label-prefix	""	Yes	Label prefix used for Stream load import. It must be globally unique in two-phase commit (2pc) scenarios to ensure Flink's EOS semantics.
sink.enable-2pc	TRUE	No	Whether to enable 2pc for ensuring Exactly-Once semantics. The default value is **true**. Refer to this **link** for more information on 2pc.

Parameter	Default Value	Mandatory	Parameter Type Description
sink.check-interval	10000	No	Interval for checking exceptions during loading.
sink.max-retries	3	No	Maximum number of retries when writing records to the database fails.
sink.buffer-size	256 * 1024	No	Buffer size for caching data during Stream load.
sink.buffer-count	3	No	Buffer count for caching data during Stream load.
sink.enable-delete	TRUE	No	Whether to enable deletion. This option requires batch deletion to be enabled for the Doris table (default in Doris 0.15 or later for Unique model only).

Parameter	Default Value	Mandatory	Parameter Type Description
sink.properties.*	--	No	Import parameters for Stream load. For example, **'sink.properties.column_separator' = ','** defines the column separator, and **'sink.properties.escape_delimiters' = 'true'** treats special characters as separators, where **'\x01'** is converted to binary **0x01**. JSON format import 'sink.properties.format' = 'json' 'sink.properties.read_json_by_line' = 'true'

Example

In this example, data is read from the DataGen data source and written to the Doris result table.

1. Create an enhanced datasource connection in the VPC and subnet where Doris locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**. Add MRS host information for the enhanced datasource connection. For details, see **Modifying Host Information**.
2. Set Doris security groups and add inbound rules to allow access from the Flink queue. Test the queue connectivity based on the Doris address. If the connection passes the test, it is bound to the queue.

   For details, see **Testing Address Connectivity**.
3. Create a Doris table by referring to *MRS Doris Usage Guide*. The creation statement is as follows:

```
CREATE TABLE IF NOT EXISTS dorisdemo
(
  `user_id` varchar(10) NOT NULL,
  `city` varchar(10),
  `age` int,
  `gender` int
)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 10
```

4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses DataGen as the data source and writes data to as a Doris result table.

```
create table student_datagen_source(
  `user_id` String NOT NULL,
  `city` String,
  `age` int,
  `gender` int
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1',
  'fields.user_id.kind' = 'random',
  'fields.user_id.length' = '7',
  'fields.city.kind' = 'random',
  'fields.city.length' = '7'
);


CREATE TABLE dorisDemo (
  `user_id` String NOT NULL,
  `city` String,
  `age` int,
  `gender` int
) with (
  'connector' = 'doris',
  'fenodes' = 'FE_IP:PORT',
  'table.identifier' = 'demo.dorisdemo',
  'username' = 'dorisUser',
  'password' = 'dorisPassword',
  'sink.label-prefix' = 'demo',
  'sink.enable-2pc' = 'true',
  'sink.buffer-count' = '10'
);

insert into dorisDemo select * from student_datagen_source
```

5. Check whether data is successfully written to the Doris result table.

user_id	city	age	gender
50aff04	93406c5	12	1
681a230	1f27d06	16	1
006eff4	3521ded	18	0

1.5.5.4 Dimension Table

Function

Create a Doris dimension table to connect to the source streams for wide table generation.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.

  – For details about how to configure security group rules, see **Security Group Overview**.

- **If MRS Doris is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.**

  For details, see **Modifying Host Information**.

- Kerberos authentication is disabled for the cluster (the cluster is in normal mode).

  After connecting to Doris as user **admin**, create a role with administrator permissions, and bind the role to the user.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- Kerberos authentication is disabled for the cluster (the cluster is in normal mode).
- Doris table names are case sensitive.
- When Doris of CloudTable is used, set the port number in the **fenodes** field to **8030**, for example, *xx*:**8030**. In addition, enable ports **8030**, **8040**, and **9030** in the security group.
- After HTTPS is enabled, add the following configuration parameters to the **with** clause for creating a table:
  – **'doris.enable.https' = 'true'**
  – **'doris.ignore.https.ca' = 'true'**

Syntax

```
create table hbaseSource (
  attr_name attr_type
  (',' attr_name attr_type)*
 )
with (
  'connector' = 'doris',
  'fenodes' = 'FE_IP:PORT,FE_IP:PORT,FE_IP:PORT',
  'table.identifier' = 'database.table',
  'username' = 'dorisUsername',
  'password' = 'dorisPassword'
);
```

Parameter Description

Shared configuration

Parameter	Default Value	Mandatory	Parameter Type Description
fenodes	--	Y	IP address and port number of the Doris FE. Use commas (,) to separate them for multiple instances. To obtain the port number, log in to FusionInsight Manager, choose **Cluster** > **Services** > **Doris** > **Configurations**, and search for **http**. Search for **https** instead if HTTPS is enabled.
table.identifier	--	Y	Doris table name, for example, **db.tbl**.
username	--	Y	User name for accessing Doris.
password	--	Y	Password for accessing Doris.
lookup.cache.max-rows	-1L	N	Maximum number of rows to search in the cache, where the oldest row will be deleted if this value is exceeded. To enable cache configuration, both the **cache.max-rows** and **cache.ttl** options must be specified.
lookup.cache.ttl	10s	N	Cache lifespan.
lookup.max-retries	3	N	Maximum number of retry attempts when a database lookup fails.

Example

This example reads data from a Doris source table and inputs it into the Print connector.

1. Create an enhanced datasource connection in the VPC and subnet where Doris locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**. Add MRS host information for the enhanced datasource connection. For details, see **Modifying Host Information**.
2. Set Doris and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Doris and Kafka addresses by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
3. Create a Doris table and insert 10 data records by referring to *MRS Doris Usage Guide*. The creation statement is as follows:
```
CREATE TABLE IF NOT EXISTS dorisdemo
(
  `user_id` varchar(10) NOT NULL,
  `city` varchar(10),
  `age` int,
  `gender` int
)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 10;

INSERT INTO dorisdemo VALUES ('user1', 'city1', 20, 1);
INSERT INTO dorisdemo VALUES ('user2', 'city2', 21, 0);
INSERT INTO dorisdemo VALUES ('user3', 'city3', 22, 1);
INSERT INTO dorisdemo VALUES ('user4', 'city4', 23, 0);
INSERT INTO dorisdemo VALUES ('user5', 'city5', 24, 1);
INSERT INTO dorisdemo VALUES ('user6', 'city6', 25, 0);
INSERT INTO dorisdemo VALUES ('user7', 'city7', 26, 1);
INSERT INTO dorisdemo VALUES ('user8', 'city8', 27, 0);
INSERT INTO dorisdemo VALUES ('user9', 'city9', 28, 1);
INSERT INTO dorisdemo VALUES ('user10', 'city10', 29, 0);
```
4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. This job simulates reading data from Kafka, performs a join with a Doris dimension table to denormalize the data, and outputs it to Print.
```
CREATE TABLE ordersSource (
  user_id string,
  user_name string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafka-topic',
  'properties.bootstrap.servers' = 'kafkaIp:port,kafkaIp:port,kafkaIp:port',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE dorisDemo (
  `user_id` String NOT NULL,
  `city` String,
  `age` int,
  `gender` int
) with (
  'connector' = 'doris',
  'fenodes' = 'IP address of the FE instance:Port number',
  'table.identifier' = 'demo.dorisdemo',
```

```
  'username' = 'dorisUsername',
  'password' = 'dorisPassword',
  'lookup.cache.ttl'='10 m',
  'lookup.cache.max-rows' = '100'
);

CREATE TABLE print (
  user_id string,
  user_name string,
  `city` String,
  `age` int,
  `gender` int
) WITH (
  'connector' = 'print'
);

insert into print
select
  orders.user_id,
  orders.user_name,
  dim.city,
  dim.age,
  dim.sex
from ordersSource orders
left join dorisDemo for system_time as of orders.proctime as dim on orders.user_id = dim.user_id;
```

5. Write two data records to the Kafka data source.
```
{"user_id": "user1", "user_name": "name1"}
{"user_id": "user2", "user_name": "name2"}
```
6. View the data in the Print result table.
```
+I[user1, name1, city1, 20, 1]
+I[user2, name2, city2, 21, 0]
```

1.5.6 GaussDB(DWS)

1.5.6.1 Overview

GaussDB(DWS) is an online data processing database based on the cloud infrastructure and platform and helps you mine and analyze massive sets of data. DLI reads data of Flink jobs from GaussDB(DWS). GaussDB(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.

For more information about GaussDB(DWS), see **Data Warehouse Service Management Guide**.

DLI Flink 1.15 now offers two GaussDB(DWS) connector options for accessing GaussDB data:

- **GaussDB(DWS)'s self-developed GaussDB(DWS) connector (recommended)**: This option focuses on the performance of and direct interaction with GaussDB(DWS), allowing users to easily and flexibly read and write data.

  You can use GaussDB(DWS)'s self-developed GaussDB(DWS) connector by creating UDFs. For details about how to create a UDF, see **UDFs**.

  For details about how to use the GaussDB(DWS) connector, see **dws-connector-flink**.

- **DLI's GaussDB(DWS) connector (discarded and not recommended)**: This option allows users to customize sink and source functions to meet specific data read and write needs.

  For how to use DLI's GaussDB(DWS) connector, see **Table 1-33**.

Table 1-33 Supported GaussDB(DWS) connector types

Type	Instruction
Source table	**GaussDB(DWS) Source Table (Not Recommended)**
Result table	**GaussDB(DWS) Result Table (Not Recommended)**
Dimension table	**GaussDB(DWS) Dimension Table (Not Recommended)**

1.5.6.2 GaussDB(DWS) Source Table (Not Recommended)

Function

DLI reads data of Flink jobs from GaussDB(DWS). GaussDB(DWS) database kernel is compliant with PostgreSQL. The PostgreSQL database can store data of more complex types and deliver space information services, multi-version concurrent control (MVCC), and high concurrency. It applies to location applications, financial insurance, and e-Commerce.

GaussDB(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 GaussDB(DWS), see **Data Warehouse Service Management Guide**.

NOTE

You are advised to use GaussDB(DWS) self-developed GaussDB(DWS) connector.

For how to use the GaussDB(DWS) connector, see **dws-connector-flink**.

Prerequisites

- You have created a GaussDB(DWS) cluster.

  For details about how to create a GaussDB(DWS) cluster, see **Creating a Cluster** in the *Data Warehouse Service Management Guide*.
- You have created a GaussDB(DWS) database table.
- An enhanced datasource connection has been created for DLI to connect to GaussDB(DWS) clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- Fields in the **with** parameter can only be enclosed in single quotes.

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
)
with (
  'connector' = 'gaussdb',
  'url' = '',
  'table-name' = '',
  'username' = '',
  'password' = ''
);
```

Parameters

Table 1-34 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **gaussdb**.

Parameter	Mandatory	Default Value	Data Type	Description
url	Yes	None	String	JDBC connection address. Set the IP address in this parameter to the internal IP address of GaussDB(DWS). If you use the gsjdbc4 driver, set the value in jdbc:postgresql://${ip}:${port}/${dbName} format. If you use the gsjdbc200 driver, set the value in jdbc:gaussdb://${ip}:${port}/${dbName} format.
table-name	Yes	None	String	Name of the GaussDB(DWS) table to be operated. If the GaussDB(DWS) table is in a schema, refer to the description of **GaussDB(DWS) table in a schema**.
driver	No	org.postgresql.Driver	String	JDBC connection driver. The default value is **org.postgresql.Driver**. ● If you use the gsjdbc4 driver for connection, set this parameter to **org.postgresql.Driver**. ● If you use the gsjdbc200 driver for connection, set this parameter to **com.huawei.gauss200.jdbc.Driver**.
username	No	None	String	Username for GaussDB(DWS) database authentication. This parameter must be configured in pair with **password**.
password	No	None	String	Password for GaussDB(DWS) database authentication. This parameter must be configured in pair with **username**.
scan.partition.column	No	None	String	Name of the column used to partition the input. Note: This parameter must be used together with **scan.partition.lower-bound**, **scan.partition.upper-bound**, and **scan.partition.num**.
scan.partition.lower-bound	No	None	Integer	Lower bound of values to be fetched for the first partition. This parameter must be used together with **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.num**.

Parameter	Mandatory	Default Value	Data Type	Description
scan.partition.upper-bound	No	None	Integer	Upper bound of values to be fetched for the last partition. This parameter must be used together with **scan.partition.column**, **scan.partition.lower-bound**, and **scan.partition.num**.
scan.partition.num	No	None	Integer	Number of partitions to be created. This parameter must be used together with **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.upper-bound**.
scan.fetch-size	No	0	Integer	Number of rows fetched from the database each time. The default value is **0**, indicating that the number of rows is not limited.

Example

In this example, data is read from the GaussDB(DWS) data source and written to the Print result table. The procedure is as follows:

1. Create a table named **dws_order** in GaussDB(DWS).
```
create table public.dws_order(
  order_id VARCHAR,
  order_channel VARCHAR,
  order_time VARCHAR,
  pay_amount FLOAT8,
  real_pay FLOAT8,
  pay_time VARCHAR,
  user_id VARCHAR,
  user_name VARCHAR,
  area_id VARCHAR);
```
Insert data into the **dws_order** table.
```
insert into public.dws_order
  (order_id,
  order_channel,
  order_time,
  pay_amount,
  real_pay,
  pay_time,
  user_id,
  user_name,
  area_id) values
  ('202103241000000001', 'webShop', '2021-03-24 10:00:00', '100.00', '100.00', '2021-03-24 10:02:03',
'0001', 'Alice', '330106'),
  ('202103251202020001', 'miniAppShop', '2021-03-25 12:02:02', '60.00', '60.00', '2021-03-25 12:03:00',
'0002', 'Bob', '330110');
```
2. Create an enhanced datasource connection in the VPC and subnet where GaussDB(DWS) locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
3. Set GaussDB(DWS) security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the GaussDB(DWS) address

by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses the GaussDB(DWS) data source and the Print result table.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE dwsSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'gaussdb',
  'url' = 'jdbc:postgresql://DWSIP:DWSPort/DWSdbName',
  'table-name' = 'dws_order',
  'driver' = 'org.postgresql.Driver',
  'username' = 'DWSUserName',
  'password' = 'DWSPassword'
);

CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);

insert into printSink select * from dwsSource;
```

5. Perform the following operations to view the data result in the **taskmanager.out** file:

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

   b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.

   c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I(202103241000000001,webShop,2021-03-24 10:00:00,100.0,100.0,2021-03-24 
10:02:03,0001,Alice,330106)
+I(202103251202020001,miniAppShop,2021-03-25 12:02:02,60.0,60.0,2021-03-25 
12:03:00,0002,Bob,330110)
```


FAQ

- Q: What should I do if the job execution fails and the log contains the following error information?

  ```
  java.io.IOException: unable to open JDBC writer
  ...
  Caused by: org.postgresql.util.PSQLException: The connection attempt failed.
  ...
  Caused by: java.net.SocketTimeoutException: connect timed out
  ```

  A: The datasource connection is not bound or the binding fails.

  – To reconfigure datasource connections, refer to **Enhanced Datasource Connection**. Rectify the fault by referring to **DLI Failed to Connect to GaussDB(DWS) Through an Enhanced Datasource Connection**.

- Q: How can I configure a GaussDB(DWS) table that is in a schema?

  A: The following provides an example of configuring the **dws_order** table in the **dbuser2** schema:

  ```
  CREATE TABLE dwsSource (
    order_id string,
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
    pay_time string,
    user_id string,
    user_name string,
    area_id string
  ) WITH (
    'connector' = 'gaussdb',
    'url' = 'jdbc:postgresql://DWSIP:DWSPort/DWSdbName',
    'table-name' = 'dbuser2.dws_order',
    'driver' = 'org.postgresql.Driver',
    'username' = 'DWSUserName',
    'password' = 'DWSPassword'
  );
  ```

1.5.6.3 GaussDB(DWS) Result Table (Not Recommended)

Function

DLI outputs the Flink job output data to GaussDB(DWS). GaussDB(DWS) database kernel is compliant with PostgreSQL. The PostgreSQL database can store data of more complex types and deliver space information services, multi-version concurrent control (MVCC), and high concurrency. It applies to location applications, financial insurance, and e-Commerce.

GaussDB(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 GaussDB(DWS), see the **Data Warehouse Service Management Guide**.

NOTE

You are advised to use GaussDB(DWS) self-developed GaussDB(DWS) connector.

For how to use the GaussDB(DWS) connector, see **dws-connector-flink**.

Prerequisites

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- You have created a GaussDB(DWS) cluster. For details about how to create a GaussDB(DWS) cluster, see **Creating a Cluster** in the *Data Warehouse Service Management Guide*.
- You have created a GaussDB(DWS) database table.
- An enhanced datasource connection has been created for DLI to connect to GaussDB(DWS) clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- Fields in the **with** parameter can only be enclosed in single quotes.
- To use the upsert mode, you must define the primary key for both the GaussDB(DWS) result table and the GaussDB(DWS) table connected to the result table.
- If tables with the same name exist in different GaussDB(DWS) schemas, you need to specify the schemas in the Flink open source SQL statements.
- If you use the gsjdbc4 driver for connection, set **driver** to **org.postgresql.Driver**. You can omit this parameter because the gsjdbc4 driver is the default one.

  For example, run the following statements to use the gsjdbc4 driver to write data to GaussDB(DWS) in upsert mode:

```
create table dwsSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
  'connector' = 'gaussdb',
  'url' = 'jdbc:postgresql://DwsAddress:DwsPort/DwsDatabase',
```

```
  'table-name' = 'car_info',
  'username' = 'DwsUserName',
  'password' = 'DwsPasswrod',
  'write.mode' = 'upsert'
);
```

- If you use the gsjdbc200 driver for connection, set **driver** to **com.huawei.gauss200.jdbc.Driver**.

  For example, run the following statements to write data to GaussDB(DWS) result table **test** that is in schema **ads_game_sdk_base**:

```
create table dwsSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
  'connector' = 'gaussdb',
  'table-name' = 'ads_game_sdk_base.test',
  'driver' = 'com.huawei.gauss200.jdbc.Driver',
  'url' = 'jdbc:gaussdb://DwsAddress:DwsPort/DwsDatabase',
  'username' = 'DwsUserName',
  'password' = 'DwsPasswrod',
  'write.mode' = 'upsert'
);
```

Syntax

NOTE

Do not set all attributes in a GaussDB(DWS) result table to **PRIMARY KEY**.

```
create table dwsSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'gaussdb',
  'url' = '',
  'table-name' = '',
  'driver' = '',
  'username' = '',
  'password' = ''
);
```

Parameters

Table 1-35 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **gaussdb**.

Parameter	Mandatory	Default Value	Data Type	Description
url	Yes	None	String	JDBC connection address. If you use the gsjdbc4 driver, set the value in jdbc:postgresql://${ip}:${port}/${dbName} format. If you use the gsjdbc200 driver, set the value in jdbc:gaussdb://${ip}:${port}/${dbName} format.
table-name	Yes	None	String	Name of the table to be operated. If the GaussDB(DWS) table is in a schema, the format is **schema\".\"***Table name*. For details, see **FAQ**.
driver	No	org.postgresql.Driver	String	JDBC connection driver. The default value is **org.postgresql.Driver**. • If you use the gsjdbc4 driver for connection, set this parameter to **org.postgresql.Driver**. • If you use the gsjdbc200 driver for connection, set this parameter to **com.huawei.gauss200.jdbc.Driver**.
username	No	None	String	Username for GaussDB(DWS) database authentication. This parameter must be configured in pair with **password**.
password	No	None	String	Password for GaussDB(DWS) database authentication. This parameter must be configured in pair with **username**.
write.mode	No	None	String	Data write mode. The value can be **copy**, **insert**, or **upsert**. The default value is **upsert**. This parameter must be configured depending on **primary key**. • If **primary key** is not configured, data can be appended in **copy** and **insert** modes. • If **primary key** is configured, all the three modes are available. Note: GaussDB(DWS) does not support the update of distribution columns. The primary keys of columns to be updated must cover all distribution columns defined in the GaussDB(DWS) table.

Parameter	Mandatory	Default Value	Data Type	Description
sink.buffer-flush.max-rows	No	100	Integer	Maximum number of rows to buffer for each write request. It can improve the performance of writing data, but may increase the latency. You can set this parameter to **0** to disable it.
sink.buffer-flush.interval	No	1s	Duration	Interval for refreshing the buffer, during which data is refreshed by asynchronous threads. It can improve the performance of writing data to the database, but may increase the latency. You can set this parameter to **0** to disable it. Note: If **sink.buffer-flush.max-size** and **sink.buffer-flush.max-rows** are both set to **0** and the buffer refresh interval is configured, the buffer is asynchronously refreshed. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit).
sink.max-retries	No	3	Integer	Maximum number of write retries.
write.escape-string-value	No	false	Boolean	Whether to escape values of the string type. This parameter is used only when **write.mode** is set to **copy**.
key-by-before-sink	No	false	Boolean	Whether to partition by the specified primary key before the sink operator This parameter aims to solve the problem of interlocking between two subtasks when they acquire row locks based on the primary key from GaussDB(DWS), multiple concurrent writes occur, and **write.mode** is **upsert**. This happens when a batch of data written to the sink by multiple subtasks has more than one record with the same primary key, and the order of these records with the same primary key is inconsistent.

Example

In this example, data is read from the Kafka data source and written to the GaussDB(DWS) result table in insert mode. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where GaussDB(DWS) and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set GaussDB(DWS) and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the GaussDB(DWS) and Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Connect to the GaussDB(DWS) database and create a table named **dws_order**.
```
create table public.dws_order(
  order_id VARCHAR,
  order_channel VARCHAR,
  order_time VARCHAR,
  pay_amount FLOAT8,
  real_pay FLOAT8,
  pay_time VARCHAR,
  user_id VARCHAR,
  user_name VARCHAR,
  area_id VARCHAR);
```
4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses the Kafka data source and the GaussDB(DWS) result table.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**
```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE dwsSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
```

```
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'gaussdb',
  'url' = 'jdbc:postgresql://DWSAddress:DWSPort/DWSdbName',
  'table-name' = 'dws_order',
  'driver' = 'org.postgresql.Driver',
  'username' = 'DWSUserName',
  'password' = 'DWSPassword',
  'write.mode' = 'insert'
);

insert into dwsSink select * from kafkaSource;
```

5. Connect to the Kafka cluster and enter the following test data to Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}
```

6. Run the following SQL statement in GaussDB(DWS) to view the data result:

```
 select * from dws_order
```

The data result is as follows:

```
202103241000000001    webShop    2021-03-24 10:00:00    100.0    100.0    2021-03-24 10:02:03
0001    Alice    330106
```

FAQ

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
java.io.IOException: unable to open JDBC writer
...
Caused by: org.postgresql.util.PSQLException: The connection attempt failed.
...
Caused by: java.net.SocketTimeoutException: connect timed out
```

  A: The datasource connection is not bound or the binding fails.

  – To reconfigure datasource connections, refer to **Enhanced Datasource Connection**. Rectify the fault by referring to **DLI Failed to Connect to GaussDB(DWS) Through an Enhanced Datasource Connection**.

- Q: How can I configure a GaussDB(DWS) table that is in a schema?

  A: When GaussDB(DWS) table **test** is in schema **ads_game_sdk_base**, refer to the **'table-name'** parameter setting in the following example:

```
CREATE TABLE ads_rpt_game_sdk_realtime_ada_reg_user_pay_mm (
  ddate DATE,
  dmin TIMESTAMP(3),
  game_appkey VARCHAR,
  channel_id VARCHAR,
  pay_user_num_1m bigint,
  pay_amt_1m bigint,
  PRIMARY KEY (ddate, dmin, game_appkey, channel_id) NOT ENFORCED
) WITH (
  'connector' = 'gaussdb',
  'url' = 'jdbc:postgresql://<yourDwsAddress>:<yourDwsPort>/dws_bigdata_db',
  'table-name' = 'ads_game_sdk_base.test',
  'username' = '<yourUsername>',
  'password' = '<yourPassword>',
  'write.mode' = 'upsert'
);
```

- Q: What can I do if a job is running properly but there is no data in GaussDB(DWS)?

  A: Check the following items:

- Check whether the JobManager and TaskManager logs contain error information. To view logs, perform the following steps:
  i. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  ii. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
  iii. Go to the folder of the date, find the folder whose name contains **taskmanager** or **jobmanager**, download the **taskmanager.out** or **jobmanager.out** file, and view result logs.
- Check whether the datasource connection is correctly bound and whether a security group rule allows access of the queue.
- Check whether the GaussDB(DWS) table to which data is to be written exists in multiple schemas. If it does, specify the schemas in the Flink job.

1.5.6.4 GaussDB(DWS) Dimension Table (Not Recommended)

Function

Create a GaussDB(DWS) table to connect to source streams for wide table generation.

NOTE

You are advised to use GaussDB(DWS) self-developed GaussDB(DWS) connector.

For how to use the GaussDB(DWS) connector, see **dws-connector-flink**.

Prerequisites

- Ensure that you have created a GaussDB(DWS) cluster using your account.
  For details about how to create a DWS cluster, see **Creating a Cluster** in the *Data Warehouse Service Management Guide*.
- A DWS database table has been created.
- An enhanced datasource connection has been created for DLI to connect to DWS clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- Fields in the **with** parameter can only be enclosed in single quotes.

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
)
with (
  'connector' = 'gaussdb',
  'url' = '',
  'table-name' = '',
  'username' = '',
  'password' = ''
);
```

Parameters

Table 1-36 Parameter description

Parameter	Mandatory	Default Value	Data Types	Description
connector	Yes	None	String	Connector type. Set this parameter to **gaussdb**.
url	Yes	None	String	JDBC connection address. If you use the gsjdbc4 driver, set the value in jdbc:postgresql://${ip}:${port}/${dbName} format. If you use the gsjdbc200 driver, set the value in jdbc:gaussdb://${ip}:${port}/${dbName} format.
table-name	Yes	None	String	Name of the table where the data will be read from the database

Parameter	Mandatory	Default Value	Data Types	Description
driver	No	None	String	JDBC connection driver. The default value is **org.postgresql.Driver**. ● If you use the gsjdbc4 driver for connection, set **connector.driver** to **org.postgresql.Driver**. ● If you use the gsjdbc200 driver for connection, set **connector.driver** to **com.huawei.gauss200.jdbc.Driver**.
username	No	None	String	Database authentication user name. This parameter must be configured in pair with **password**.
password	No	None	String	Database authentication password. This parameter must be configured in pair with **username**.
scan.partition.column	No	None	String	Name of the column used to partition the input This parameter must be set when **scan.partition.lower-bound**, **scan.partition.upper-bound**, and **scan.partition.num** are all configured, and should not be set when other three parameters are not.
scan.partition.lower-bound	No	None	Integer	Lower bound of values to be fetched for the first partition This parameter must be set when **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.num** are all configured, and should not be set when other three parameters are not.
scan.partition.upper-bound	No	None	Integer	Upper bound of values to be fetched for the last partition This parameter must be set when **scan.partition.column**, **scan.partition.lower-bound**, and **scan.partition.num** are all configured, and should not be set when other three parameters are not.

Parameter	Mandatory	Default Value	Data Types	Description
scan.partition.num	No	None	Integer	Number of partitions to be created This parameter must be set when **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.upper-bound** are all configured, and should not be set when other three parameters are not.
scan.fetch-size	No	0	Integer	Number of rows fetched from the database each time. The default value **0** indicates that the number of rows is not limited.
scan.auto-commit	No	true	Boolean	Automatic commit flag. It determines whether each statement is committed in a transaction automatically.
lookup.cache.max-rows	No	None	Integer	Maximum number of cached rows in a dimension table. When the rows exceed this value, the data that is added first will be marked as expired. Lookup cache is disabled by default.
lookup.cache.ttl	No	None	Duration	Maximum time to live (TTL) of for every rows in lookup cache. Caches exceeding the TTL will be expired. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit). Lookup cache is disabled by default.
lookup.max-retries	No	3	Integer	Maximum retry times if lookup database failed.

Example

Read data from a Kafka source table, use a GaussDB(DWS) table as the dimension table. Write wide table information generated by the source and dimension tables to a Kafka result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where DWS and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set GaussDB(DWS) and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the DWS and Kafka

address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.

3. Connect to the GaussDB(DWS) database instance, create a table as a dimension table, and name the table **area_info**. Example SQL statements are as follows:

```
create table public.area_info(
  area_id VARCHAR,
  area_province_name VARCHAR,
  area_city_name VARCHAR,
  area_county_name VARCHAR,
  area_street_name VARCHAR,
  region_name VARCHAR);
```

4. Connect to the database and run the following statement to insert test data into the dimension table **area_info**:

```
 insert into area_info
 (area_id, area_province_name, area_city_name, area_county_name, area_street_name, region_name)
 values
 ('330102', 'a1', 'b1', 'c1', 'd1', 'e1'),
 ('330106', 'a1', 'b1', 'c2', 'd2', 'e1'),
 ('330108', 'a1', 'b1', 'c3', 'd3', 'e1'),
 ('330110', 'a1', 'b1', 'c4', 'd4', 'e1');
```

5. Create a Flink OpenSource SQL job Enter the following job script and submit the job. The job script uses Kafka as the data source and a GaussDB(DWS) table as the dimension table. Data is output to a Kafka result table.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Set the values of the parameters in bold in the following script as needed.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaSourceTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'dws-order',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

-- Create an address dimension table
create table area_info (
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) WITH (
  'connector' = 'gaussdb',
  'driver' = 'org.postgresql.Driver',
  'url' = 'jdbc:postgresql://DwsAddress:DwsPort/DwsDbName',
  'table-name' = 'area_info',
  'username' = 'DwsUserName',
  'password' = 'DwsPassword',
  'lookup.cache.max-rows' = '10000',
  'lookup.cache.ttl' = '2h'
```

```
);

-- Generate a wide table based on the address dimension table containing detailed order information.
create table order_detail(
    order_id string,
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
    pay_time string,
    user_id string,
    user_name string,
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) with (
  'connector' = 'kafka',
  'topic' = 'KafkaSinkTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'json'
);

insert into order_detail
    select orders.order_id, orders.order_channel, orders.order_time, orders.pay_amount, orders.real_pay,
orders.pay_time, orders.user_id, orders.user_name,
           area.area_id, area.area_province_name, area.area_city_name, area.area_county_name,
           area.area_street_name, area.region_name  from orders
    left join area_info for system_time as of orders.proctime as area on orders.area_id = area.area_id;
```

6. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

```
{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202103251505050001", "order_channel":"qqShop", "order_time":"2021-03-25 15:05:05", "pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2021-03-25 15:10:00", "user_id":"0003", "user_name":"Cindy", "area_id":"330108"}
```

7. Connect to the Kafka cluster and read data from the sink topic of Kafka. The result is as follows:

```
{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24 16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24 16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106","area_province_name":"a1","area_city_name":"b1","area_county_name":"c2","area_street_name":"d2","region_name":"e1"}

{"order_id":"202103251202020001","order_channel":"miniAppShop","order_time":"2021-03-25 12:02:02","pay_amount":60.0,"real_pay":60.0,"pay_time":"2021-03-25 12:03:00","user_id":"0002","user_name":"Bob","area_id":"330110","area_province_name":"a1","area_city_name":"b1","area_county_name":"c4","area_street_name":"d4","region_name":"e1"}

{"order_id":"202103251505050001","order_channel":"qqShop","order_time":"2021-03-25 15:05:05","pay_amount":500.0,"real_pay":400.0,"pay_time":"2021-03-25 15:10:00","user_id":"0003","user_name":"Cindy","area_id":"330108","area_province_name":"a1","area_city_name":"b1","area_county_name":"c3","area_street_name":"d3","region_name":"e1"}
```

FAQs

- Q: What should I do if Flink job logs contain the following error information?

```
java.io.IOException: unable to open JDBC writer
...
Caused by: org.postgresql.util.PSQLException: The connection attempt failed.
```

```
...
Caused by: java.net.SocketTimeoutException: connect timed out
```

A: The datasource connection is not bound or the binding fails.

- To reconfigure datasource connections, refer to **Enhanced Datasource Connection**. Rectify the fault by referring to **DLI Failed to Connect to GaussDB(DWS) Through an Enhanced Datasource Connection**.

- Q: How can I configure a GaussDB(DWS) table that is in a schema?

  A: In the following example configures the **area_info** table in the **dbuser2** schema.

```
-- Create an address dimension table
create table area_info (
   area_id string,
   area_province_name string,
   area_city_name string,
   area_county_name string,
   area_street_name string,
   region_name string
) WITH (
 'connector' = 'gaussdb',
  'driver' = 'org.postgresql.Driver',
  'url' = 'jdbc:postgresql://DwsAddress:DwsPort/DwsDbname',
  'table-name' = 'dbuser2.area_info',
  'username' = 'DwsUserName',
  'password' = 'DwsPassword',
  'lookup.cache.max-rows' = '10000',
  'lookup.cache.ttl' = '2h'
);
```

1.5.7 Elasticsearch

Function

DLI outputs the output data of the Flink job to an index in the Elasticsearch engine of the 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.

For more information about CSS, see **Cloud Search Service User Guide**.

For details, see **Elasticsearch SQL Connector**.

Table 1-37 Supported types

Type	Description
Supported Table Types	Result table
Supported Data Formats	**JSON**

Prerequisites

- Ensure that you have created a cluster on CSS using your account. For details about how to create a cluster, see **Creating a Cluster**.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- Fields in the **with** parameter can only be enclosed in single quotes.
- Only CSS 7.*X* or later clusters are currently supported.
- If security mode is enabled and HTTPS is enabled, you need to configure the username, password, and certificate location. Note that the **hosts** field value in this scenario starts with **https**.
- ICMP must be enabled for the security group inbound rule of the CSS cluster.
- Fields in the **with** parameter can only be enclosed in single quotes.
- For details about how to use data types, see section **Format**.

Syntax

```
create table esSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'elasticsearch-7',
  'hosts' = '',
  'index' = ''
);
```

Parameter Description

Table 1-38 Elasticsearch result table parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Specify what connector to use. Set this parameter to **elasticsearch-7**. This indicates connecting to Elasticsearch 7.*x* cluster.

Parameter	Mandatory	Default Value	Data Type	Description
hosts	Yes	None	String	Host name of the cluster where Elasticsearch is located. Use semicolons (;) to separate multiple host names.
index	Yes	None	String	Elasticsearch index for every record. The index can be a static index (for example, **'myIndex'**) or a dynamic index (for example, **'index-{log_ts|yyyy-MM-dd}'**). See the following **Dynamic Index** section for more details.
username	No	None	String	Username of the cluster where Elasticsearch locates. This parameter must be configured in pair with **password**.
password	No	None	String	Password of the cluster where Elasticsearch locates. This parameter must be configured in pair with **username**.
document-id.key-delimiter	No	_	String	Delimiter for composite keys ("_" by default), e.g., **$** would result in IDs **KEY1$KEY2$KEY3**.
failure-handler	No	fail	String	Failure handling strategy in case a request to Elasticsearch fails. Valid strategies are: ● **fail**: throws an exception if a request fails and causes a job failure. ● **ignore**: ignores failures and drops the request. ● **retry-rejected**: Requests that failed due to saturated queue are re-added. ● **custom class name**: The subclass of ActionRequestFailureHandler is used to handle failures.
sink.flush-on-checkpoint	No	true	Boolean	Flush on checkpoint or not. When disabled, a sink will not wait for all pending action requests to be acknowledged by Elasticsearch on checkpoints. Thus, a sink does not provide any strong guarantees for at-least-once delivery of action requests.

Parameter	Mandatory	Default Value	Data Type	Description
sink.bulk-flush.max-actions	No	1000	Interger	Maximum number of buffered actions per bulk request. You can set this parameter to **0** to disable it.
sink.bulk-flush.max-size	No	2mb	MemorySize	Maximum size in memory of buffered actions per bulk request. Must be in MB granularity. Can be set to **0** to disable it.
sink.bulk-flush.interval	No	1s	Duration	The interval to flush buffered actions. Can be set to **0** to disable it. Note, both **sink.bulk-flush.max-size** and **sink.bulk-flush.max-actions** can be set to **0** with the flush interval set allowing for complete async processing of buffered actions.
sink.bulk-flush.backoff.strategy	No	DISABLED	String	Specify how to perform retries if any flush actions failed due to a temporary request error. Valid strategies are: ● **DISABLED**: no retry performed, i.e. fail after the first request error. ● **CONSTANT**: wait for backoff delay between retries. ● **EXPONENTIAL**: initially wait for backoff delay and increase exponentially between retries.
sink.bulk-flush.backoff.max-retries	No	None	Integer	Maximum number of rollback retries.
sink.bulk-flush.backoff.delay	No	None	Duration	Delay between each backoff attempt. For **CONSTANT** backoff, this is simply the delay between each retry. For **EXPONENTIAL** backoff, this is the initial base delay.
connection.path-prefix	No	None	String	Prefix string added to each REST communication, for example, **'/v1'**.
connection.request-timeout	No	None	Duration	The timeout in milliseconds for requesting a connection from the connection manager. The timeout must be larger than or equal to 0. A timeout value of zero is interpreted as an infinite timeout.

Parameter	Mandatory	Default Value	Data Type	Description
connection.timeout	No	None	Duration	The timeout in milliseconds for establishing a connection. The timeout must be larger than or equal to 0. A timeout value of zero is interpreted as an infinite timeout.
socket.timeout	No	None	Duration	The socket timeout (SO_TIMEOUT) for waiting for data. The timeout must be larger than or equal to 0. A timeout value of zero is interpreted as an infinite timeout.
format	No	json	String	Elasticsearch connector supports to specify a format. The format must produce a valid JSON document. By default, the built-in JSON format is used. Refer to **Format** for more details and format parameters.
certificate	No	None	String	Location of the Elasticsearch cluster certificate in OBS. This parameter is required only when the security mode and HTTPS are enabled. Download the certificate from the CSS management console and upload the certificate to OBS. This parameter specifies the OBS address. Example: **obs://bucket/path/CloudSearchService.cer**

Key Handling

The Elasticsearch sink can work in either upsert mode or append mode, depending on whether a primary key is defined.

- If a primary key is defined, the Elasticsearch sink works in upsert mode which can consume queries containing UPDATE/DELETE messages.
- If a primary key is not defined, the Elasticsearch sink works in append mode which can only consume queries containing INSERT only messages.

In the Elasticsearch connector, the primary key is used to calculate the Elasticsearch document ID, which is a string of up to 512 bytes. It cannot have whitespaces.

The Elasticsearch connector generates a document ID string for every row by concatenating all primary key fields in the order defined in the DDL using a key delimiter specified by **document-id.key-delimiter**. Certain types are not allowed as a primary key field as they do not have a good string representation, e.g. **BYTES**, **ROW**, **ARRAY**, **MAP**, etc.

If no primary key is specified, Elasticsearch will generate a document ID automatically.

Dynamic Index

The Elasticsearch sink supports both static index and dynamic index.

- If you want to have a static index, the index option value should be a plain string, e.g. **myusers**, all the records will be consistently written into **myusers** index.
- If you want to have a dynamic index, you can use **{field_name}** to reference a field value in the record to dynamically generate a target index.
  - You can use **{field_name|date_format_string}** to convert a field value of TIMESTAMP/DATE/TIME type into the format specified by the **date_format_string**. The **date_format_string** is compatible with Java's **DateTimeFormatter**. For example, if the option value is **myusers-{log_ts|yyyy-MM-dd}**, then a record with **log_ts** field value **2020-03-27 12:25:55** will be written into **myusers-2020-03-27** index.
  - You can use **{now()|date_format_string}** to convert the current system time to the format specified by **date_format_string**. The corresponding time type of **now()** is **TIMESTAMP_WITH_LTZ**. When formatting the system time as a string, the time zone configured in the **session** through **table.local-time-zone** will be used. You can use **NOW()**, **now()**, **CURRENT_TIMESTAMP**, or **current_timestamp**.

> **CAUTION**
>
> When using the dynamic index generated by the current system time, for changelog stream, there is no guarantee that the records with the same primary key can generate the same index name. Therefore, the dynamic index based on the system time can only support append only stream.

Example

In this example, data is read from the Kafka data source and written to the Elasticsearch result table (Elasticsearch 7.10.2). The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where Elasticsearch and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see .
2. Set Elasticsearch and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Elasticsearch and Kafka addresses by referring to . If the connection passes the test, it is bound to the queue.
3. Log in to Kibana of the Elasticsearch cluster, select Dev Tools, enter and execute the following statement to create an index whose value is **orders**:
```
PUT /orders
{
```

```
 "settings": {
   "number_of_shards": 1
 },
   "mappings": {
     "properties": {
       "order_id": {
         "type": "text"
       },
       "order_channel": {
         "type": "text"
       },
       "order_time": {
         "type": "text"
       },
       "pay_amount": {
         "type": "double"
       },
       "real_pay": {
         "type": "double"
       },
       "pay_time": {
         "type": "text"
       },
       "user_id": {
         "type": "text"
       },
       "user_name": {
         "type": "text"
       },
       "area_id": {
         "type": "text"
       }
     }
   }
}
```

4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   Change the values of the parameters in bold as needed in the following script.

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE elasticsearchSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
```

```
  'connector' = 'elasticsearch-7',
  'hosts' = 'ElasticsearchAddress:ElasticsearchPort',
  'index' = 'orders'
);
insert into elasticsearchSink select * from kafkaSource;
```

5. Connect to the Kafka cluster and insert the following test data into Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}
```

6. Enter the following statement in Kibana of the Elasticsearch cluster and view the result:

```
GET orders/_search
```

```
{
  "took" : 201,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "orders",
        "_type" : "_doc",
        "_id" : "fopyx4sBUuT2wThgYGcp",
        "_score" : 1.0,
        "_source" : {
          "order_id" : "202103241606060001",
          "order_channel" : "appShop",
          "order_time" : "2021-03-24 16:06:06",
          "pay_amount" : 200.0,
          "real_pay" : 180.0,
          "pay_time" : "2021-03-24 16:10:06",
          "user_id" : "0001",
          "user_name" : "Alice",
          "area_id" : "330106"
        }
      },
      {
        "_index" : "orders",
        "_type" : "_doc",
        "_id" : "f4pyx4sBUuT2wThgYGcr",
        "_score" : 1.0,
        "_source" : {
          "order_id" : "202103241000000001",
          "order_channel" : "webShop",
          "order_time" : "2021-03-24 10:00:00",
          "pay_amount" : 100.0,
          "real_pay" : 100.0,
          "pay_time" : "2021-03-24 10:02:03",
          "user_id" : "0001",
          "user_name" : "Alice",
          "area_id" : "330106"
        }
      }
    ]
```

```
 }
}
```

1.5.8 OBS

1.5.8.1 OBS Source Table

Function

The file system connector can be used to read single files or entire directories into a single table.

When using a directory as the source path, there is no defined order of ingestion for the files inside the directory. For more information, see **FileSystem SQL Connector**.

Syntax

```
CREATE TABLE sink_table (
   name string,
   num INT,
   p_day string,
   p_hour string
) partitioned by (p_day, p_hour) WITH (
   'connector' = 'filesystem',
   'path' = 'obs://*** ',
   'format' = 'parquet',
   'source.monitor-interval'=''
);
```

Parameter Description

- **Directory watching**

  By default, the file system connector is bounded, that is it will scan the configured path once and then close itself.

  You can enable continuous directory watching by configuring the **source.monitor-interval** parameter:

Key	Default Value	Data Type	Description
source.monitor-interval	None	Duration	The interval in which the source checks for new files. The interval must be greater than 0. Each file is uniquely identified by its path, and will be processed once, as soon as it is discovered. The set of files already processed is kept in state during the whole lifecycle of the source, so it's persisted in checkpoints and savepoints together with the source state. Shorter intervals mean that files are discovered more quickly, but also imply more frequent listing or directory traversal of the file system/ object store. If this config option is not set, the provided path will be scanned once, hence the source will be bounded.

- **Available Metadata**

  The following connector metadata can be accessed as metadata columns in a table definition. All the metadata are read only.

Key	Data Type	Description
file.path	STRING NOT NULL	Full path of the input file
file.name	STRING NOT NULL	Name of the file, that is the farthest element from the root of the filepath
file.size	STRING NOT NULL	Byte count of the file
file.modification-time	TIMESTAMP_LTZ(3) NOT NULL	Modification time of the file

Example

Read data from the OBS table as the data source and output it to the Print connector.

```
CREATE TABLE obs_source(
   name string,
   num INT,
   `file.path` STRING NOT NULL METADATA
) WITH (
   'connector' = 'filesystem',
   'path' = 'obs://demo/sink_parquent_obs',
   'format' = 'parquet',
   'source.monitor-interval'='1 h'
);


CREATE TABLE print (
   name string,
   num INT,
   path  STRING
) WITH (
   'connector' = 'print'
);

insert into print
select * from obs_source;
```

Print result:

```
+I[0e72e, 841255524, /spark.db/sink_parquent_obs/compacted-part-fd4d4cc8-8b18-42d5-b522-9b524500fa23-0-0]
+I[53524, -2032270969, /spark.db/sink_parquent_obs/compacted-part-fd4d4cc8-8b18-42d5-b522-9b524500fa23-0-0]
+I[77225, 245599258, /spark.db/sink_parquent_obs/compacted-part-fd4d4cc8-8b18-42d5-b522-9b524500fa23-0-0]
+I[fc202, -545621464, /spark.db/sink_parquent_obs/compacted-part-fd4d4cc8-8b18-42d5-b522-9b524500fa23-0-0]
+I[07e9d, 1511139764, /spark.db/sink_parquent_obs/compacted-part-fd4d4cc8-8b18-42d5-b522-9b524500fa23-0-0]
+I[4e48b, 278014413, /spark.db/sink_parquent_obs/compacted-part-fd4d4cc8-8b18-42d5-b522-9b524500fa23-0-0]
```

1.5.8.2 OBS Result Table

Function

The FileSystem result (sink) table is used to export data to the HDFS or OBS file system. It is applicable to scenarios such as data dumping, big data analysis, data backup, and active, deep, or cold archiving.

Considering that the input stream can be unbounded, you can put the data in each bucket into **part** files of a limited size. Data can be written into a bucket based on time. For example, you can write data into a bucket every hour. This bucket contains the records received within one hour, and

data in the bucket directory is split into multiple **part** files. Each sink bucket that receives data contains at least one **part** file for each subtask. Other **part** files are created based on the configured rolling policy. For Row Formats, the default rolling policy is based on the **part** file size. You need to specify the maximum timeout period for opening a file and the timeout period for the inactive state after closing a file. Bulk Formats are rolled each time a checkpoint is created. You can add other rolling conditions based on size or time. For more information, see **FileSystem SQL Connector**.

NOTE

- To use FileSink in STREAMING mode, you need to enable the checkpoint function. **Part** files are generated only when the checkpoint is successful. If the checkpoint function is not enabled, the files remain in the in-progress or pending state, and downstream systems cannot securely read the file data.
- The number recorded by the sink end operator is the number of checkpoints, not the actual volume of the sent data. For the actual volume, see the number recorded by the streaming-writer or StreamingFileWriter operator.

Caveats

On the Flink job's editing page, select **Enable Checkpointing** on the **Running Parameters** tab. Otherwise, data cannot be written to the FileSystem result table.

Syntax

```
CREATE TABLE sink_table (
   name string,
   num INT,
   p_day string,
   p_hour string
) partitioned by (p_day, p_hour) WITH (
   'connector' = 'filesystem',
   'path' = 'obs://*** ',
   'format' = 'parquet',
   'auto-compaction' = 'true'
);
```

Usage

- **Rolling Policy**

  The Rolling Policy determines when to close the current in-progress part file and transition it from the in-progress state to the pending state, and then to the finished state. Part files in the "finished" state are the ones that are ready for viewing and are guaranteed to contain valid data that will not be reverted in case of failure.

In STREAMING mode, the Rolling Policy in combination with the checkpointing interval (pending files become finished on the next checkpoint) control how quickly part files become available for downstream readers and also the size and number of these parts. For details, see **Parameter Description**.

- **Part File Lifecycle**

  To use the output of the FileSink in downstream systems, we need to understand the naming and lifecycle of the output files produced.

  Part files can be in one of three states:

  - **In-progress**: The part file that is currently being written to is in-progress.
  - **Pending**: Closed (due to the specified rolling policy) in-progress files that are waiting to be committed.
  - **Finished**: On successful checkpoints (STREAMING) or at the end of input (BATCH) pending files transition to **Finished**

  Only finished files are safe to read by downstream systems as those are guaranteed to not be modified later.

  By default, the file naming strategy is as follows:

  - **In-progress / Pending**: part-<uid>-<partFileIndex>.inprogress.uid
  - **Finished**: part-<uid>-<partFileIndex>

  uid is a random ID assigned to a subtask of the sink when the subtask is instantiated. This **uid** is not fault-tolerant so it is regenerated when the subtask recovers from a failure.

- **Compaction**

  FileSink supports compaction of the pending files, which allows the application to have smaller checkpoint interval without generating a lot of small files.

  Once enabled, the compaction happens between the files become pending and get committed. The pending files will be first committed to temporary files whose path starts with a dot (.). Then these files will be compacted according to the strategy by the compactor specified by the users, and the new compacted pending files will be generated. Then these pending files will be emitted to the committer to be committed to the formal files. After that, the source files will be removed.

- **Partitions**

  Filesystem sink supports the partitioning function. Partitions are generated based on the selected fields by using the **partitioned by** syntax. The following is an example:

```
path
└── datetime=2022-06-25
    └── hour=10
        ├── part-0.parquet
        ├── part-1.parquet
└── datetime=2022-06-26
    └── hour=16
        ├── part-0.parquet
    └── hour=17
        ├── part-0.parquet
```

Similar to files, partitions also need to be submitted to notify downstream applications that files in the partitions can be securely read. Filesystem sink provides multiple configuration submission policies.

Parameter Description

Table 1-39 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	The value is fixed at **filesystem**.
path	Yes	None	String	OBS path
format	Yes	None	String	File format Available values are: **csv** and **parquet**
sink.rolling-policy.file-size	No	128MB	MemorySize	Maximum size of a part file. If the size of a part file exceeds this value, a new file will be generated. **NOTE** The Rolling Policy determines when to close the current in-progress part file and transition it from the in-progress state to the pending state, and then to the finished state. Part files in the "finished" state are the ones that are ready for viewing and are guaranteed to contain valid data that will not be reverted in case of failure. In STREAMING mode, the Rolling Policy in combination with the checkpointing interval (pending files become finished on the next checkpoint) control how quickly part files become available for downstream readers and also the size and number of these parts.

Parameter	Mandatory	Default Value	Data Type	Description
sink.rolling-policy.rollover-interval	No	30 min	Duration	Maximum duration that a part file can be opened. If a part file is opened longer than the maximum duration, a new file will be generated in rolling mode. The default value is 30 minutes so that there will not be a large number of small files. The check frequency is specified by **sink.rolling-policy.check-interval**. **NOTE** There must be a space between the number and the unit. The supported time units include **d**, **h**, **min**, **s**, and **ms**. For bulk files (parquet, orc, and avro), the checkpoint interval also controls the maximum open duration of a part file.
sink.rolling-policy.check-interval	No	1 min	Duration	Check interval of the time-based rolling policy This parameter controls the frequency of checking whether a file should be rolled based on **sink.rolling-policy.rollover-interval**.
auto-compaction	No	false	Boolean	Whether automatic compaction is enabled for the streaming sink. Data is first written to temporary files. After the checkpoint is complete, the temporary files generated by the checkpoint are compacted.
compaction.file-size	No	Size of **sink.rolling-policy.file-size**	MemorySize	Size of the files that will be compacted. The default value is the size of the files that will be rolled. **NOTE** ● Only files in the same checkpoint are compacted. The final files must be more than or equal to the number of checkpoints. ● If the compaction takes a long time, back pressure may occur and the checkpointing may be prolonged. ● After this function is enabled, final files are generated during checkpoint and a new file is opened to receive the data generated at the next checkpoint.

Example 1

Use datagen to randomly generate data and write the data into the **fileName** directory in the OBS bucket **bucketName**. The generation time of a file is related to the checkpoint. If the file is open for 30 minutes or more, or if it exceeds 128 MB in size, a new file will be created.

```
create table orders(
  name string,
  num INT
) with (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'fields.name.kind' = 'random',
  'fields.name.length' = '5'
);

CREATE TABLE sink_table (
   name string,
   num INT
) WITH (
   'connector' = 'filesystem',
   'path' = 'obs://bucketName/fileName',
   'format' = 'csv',
   'sink.rolling-policy.file-size'='128m',
   'sink.rolling-policy.rollover-interval'='30 min'
);
INSERT into sink_table SELECT * from orders;
```

Example 2

Use datagen to randomly generate data and write the data into the **fileName** directory in the OBS bucket **bucketName**. The file generation time is relevant to the checkpoint. When the checkpoint interval is reached or the file size reaches 100 MB, a new file is generated.

```
create table orders(
  name string,
  num INT
) with (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'fields.name.kind' = 'random',
  'fields.name.length' = '5'
);

CREATE TABLE sink_table (
   name string,
   num INT
) WITH (
   'connector' = 'filesystem',
   'path' = 'obs://bucketName/fileName',
   'format' = 'parquet',
   'sink.rolling-policy.file-size'='128m',
   'sink.rolling-policy.rollover-interval'='30 min',
   'auto-compaction'='true',
   'compaction.file-size'='100m'

);
INSERT into sink_table SELECT * from orders;
```

1.5.9 HBase

The HBase connector allows for reading from and writing to an HBase cluster. This section describes how to set up the HBase Connector to run SQL queries against HBase.

HBase always works in upsert mode for exchange changelog messages with the external system using a primary key defined on the DDL. The primary key must be defined on the HBase rowkey field (rowkey field must be declared). If the PRIMARY KEY clause is not declared, the HBase connector will take rowkey as the primary key by default. For details, see **HBase SQL Connector**.

1.5.9.1 Source Table

Function

Create a source stream to obtain data from HBase as input for jobs. HBase is a column-oriented distributed cloud storage system that features enhanced reliability, excellent performance, and elastic scalability. It applies to the storage of massive amounts of data and distributed computing. You can use HBase to build a storage system capable of storing TB- or even PB-level data. With HBase, you can filter and analyze data with ease and get responses in milliseconds, rapidly mining data value. DLI can read data from HBase for filtering, analysis, and data dumping.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.
  - For details about how to configure security group rules, see **Security Group Overview**.
- **If MRS HBase is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.**

  For details, see **Modifying Host Information**.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- The column families in created HBase source table must be declared as the ROW type, the field names map the column family names, and the nested field names map the column qualifier names.

  There is no need to declare all the families and qualifiers in the schema. Users can declare what is used in the query. Except the ROW type fields, the single atomic type field (for example, STRING or BIGINT) will be recognized as the HBase rowkey. The rowkey field can be an arbitrary name, but should be quoted using backticks if it is a reserved keyword.

Syntax

```
create table hbaseSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
  ','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'hbase-2.2',
  'table-name' = '',
  'zookeeper.quorum' = ''
);
```

Parameters

Table 1-40 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **hbase-2.2**.
table-name	Yes	None	String	Name of the HBase table to connect.
zookeeper.quorum	Yes	None	String	HBase ZooKeeper quorum, in the format of "ZookeeperAddress:ZookeeperPort". The following uses an MRS HBase cluster as an example to describe how to obtain the IP address and port number of ZooKeeper used by this parameter: ● On MRS Manager, choose **Cluster** and click the name of the desired cluster. Choose **Services** > **ZooKeeper** > **Instance**, and obtain the IP address of the ZooKeeper instance. ● On MRS Manager, choose **Cluster** and click the name of the desired cluster. Choose **Services** > **ZooKeeper** > **Configurations** > **All Configurations**, search for the **clientPort** parameter, and obtain its value, that is, the ZooKeeper port number.
zookeeper.znode.parent	No	/hbase	String	Root directory in ZooKeeper. The default value is **/hbase**.
null-string-literal	No	None	String	Representation for null values for string fields. HBase source encodes/decodes empty bytes as null values for all types except the string type.

Parameter	Mandatory	Default Value	Data Type	Description
krb_auth_name	No	None	String	Name of datasource authentication of the Kerberos type created on DLI. **Creating a Datasource Authentication**

Data Type Mapping

HBase stores all data as byte arrays. The data needs to be serialized and deserialized during read and write operations.

When serializing and de-serializing, Flink HBase connector uses utility class **org.apache.hadoop.hbase.util.Bytes** provided by HBase (Hadoop) to convert Flink data types to and from byte arrays.

Flink HBase connector encodes null values to empty bytes, and decodes empty bytes to null values for all data types except the string type. For string type, the null literal is determined by the **null-string-literal** option.

Table 1-41 Data type mapping

Flink SQL Type	HBase Conversion
CHAR/VARCHAR/STRING	byte[] toBytes(String s) String toString(byte[] b)
BOOLEAN	byte[] toBytes(boolean b) boolean toBoolean(byte[] b)
BINARY/VARBINARY	Returns byte[] as is.
DECIMAL	byte[] toBytes(BigDecimal v) BigDecimal toBigDecimal(byte[] b)
TINYINT	new byte[] { val } bytes[0] // returns first and only byte from bytes
SMALLINT	byte[] toBytes(short val) short toShort(byte[] bytes)
INT	byte[] toBytes(int val) int toInt(byte[] bytes)
BIGINT	byte[] toBytes(long val) long toLong(byte[] bytes)
FLOAT	byte[] toBytes(float val) float toFloat(byte[] bytes)

Flink SQL Type	HBase Conversion
DOUBLE	byte[] toBytes(double val) double toDouble(byte[] bytes)
DATE	Stores the number of days since epoch as an int value.
TIME	Stores the number of milliseconds of the day as an int value.
TIMESTAMP	Stores the milliseconds since epoch as a long value.
ARRAY	Not supported
MAP/MULTISET	Not supported
ROW	Not supported

Example

In this example, data is read from the HBase data source and written to the Print result table. (The HBase version used in this example is 2.2.3.)

1. Create an enhanced datasource connection in the VPC and subnet where HBase locates, and bind the connection to the required Flink queue. For details, see **Enhanced Datasource Connections**. Add MRS host information for the enhanced datasource connection by referring to **Modifying Host Information**.
2. Set HBase cluster security groups and add inbound rules to allow access from the Flink job queue. Test the connectivity using the HBase address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
3. Use the HBase shell to create HBase table **order** that has only one column family **detail**. For details, see **Using HBase from Scratch**. The creation statement is as follows:
```
create 'order', {NAME => 'detail'}
```
4. Run the following command in the HBase shell to insert a data record:
```
put 'order', '202103241000000001', 'detail:order_channel','webShop'
put 'order', '202103241000000001', 'detail:order_time','2021-03-24 10:00:00'
put 'order', '202103241000000001', 'detail:pay_amount','100.00'
put 'order', '202103241000000001', 'detail:real_pay','100.00'
put 'order', '202103241000000001', 'detail:pay_time','2021-03-24 10:02:03'
put 'order', '202103241000000001', 'detail:user_id','0001'
put 'order', '202103241000000001', 'detail:user_name','Alice'
put 'order', '202103241000000001', 'detail:area_id','330106'
```
5. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses the HBase data source and the Print result table.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**
```
create table hbaseSource (
  order_id string,-- Indicates the unique rowkey.
```

```
  detail Row( -- Indicates the column family.
    order_channel string,
    order_time string,
    pay_amount string,
    real_pay string,
    pay_time string,
    user_id string,
    user_name string,
    area_id string),
  primary key (order_id) not enforced
) with (
  'connector' = 'hbase-2.2',
   'table-name' = 'order',
   'zookeeper.quorum' = 'ZookeeperAddress:ZookeeperPort'
) ;

create table printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount string,
  real_pay string,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) with (
 'connector' = 'print'
);

insert into printSink select order_id,
detail.order_channel,detail.order_time,detail.pay_amount,detail.real_pay,
detail.pay_time,detail.user_id,detail.user_name,detail.area_id from hbaseSource;
```

6. Perform the following operations to view the data result in the **taskmanager.out** file:
   a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
   b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
   c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I(202103241000000001,webShop,2021-03-24 10:00:00,100.00,100.00,2021-03-24
10:02:03,0001,Alice,330106)
```

FAQ

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
java.lang.IllegalArgumentException: offset (0) + length (8) exceed the capacity of the array: 6
```

  A: If data in the HBase table is imported in other modes, the data is represented in the string format. Therefore, this error is reported when other data formats are used. Change the type of the non-string fields in the HBase source table created by Flink to the string format.

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
org.apache.zookeeper.ClientCnxn$SessionTimeoutException: Client session timed out, have not heard
from server in 90069ms for connection id 0x0
```


A: The datasource connection is not bound, the binding fails, or the security group of the HBase cluster is not configured to allow access from the network segment of the DLI queue. Configure the datasource connection by referring to **Enhanced Datasource Connection** or configure the security group of the HBase cluster to allow access from the DLI queue.

1.5.9.2 HBase Result Table

Function

DLI outputs the job data to HBase. HBase is a column-oriented distributed cloud storage system that features enhanced reliability, excellent performance, and elastic scalability. It applies to the storage of massive amounts of data and distributed computing. You can use HBase to build a storage system capable of storing TB- or even PB-level data. With HBase, you can filter and analyze data with ease and get responses in milliseconds, rapidly mining data value. Structured and semi-structured key-value data can be stored, including messages, reports, recommendation data, risk control data, logs, and orders. With DLI, you can write massive volumes of data to HBase at a high speed and with low latency.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- **If MRS HBase is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.**

  For details, see **Modifying Host Information**.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- The column families in created HBase result table must be declared as the ROW type, the field names map the column family names, and the nested field names map the column qualifier names. There is no need to declare all the families and qualifiers in the schema. Users can declare what is used in the query. Except the ROW type fields, the single atomic type field (for example, STRING or BIGINT) will be recognized as the HBase rowkey. The rowkey field can be an arbitrary name, but should be quoted using backticks if it is a reserved keyword.

Syntax

```
create table hbaseSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  ','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
) with (
  'connector' = 'hbase-2.2',
  'table-name' = '',
  'zookeeper.quorum' = ''
);
```

Parameters

Table 1-42 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **hbase-2.2**.
table-name	Yes	None	String	Name of the HBase table to connect.
zookeeper.quorum	Yes	None	String	HBase ZooKeeper instance information, in the format of ZookeeperAddress:ZookeeperPort. The following uses an MRS HBase cluster as an example to describe how to obtain the IP address and port number of ZooKeeper used by this parameter: ● On MRS Manager, choose **Cluster** and click the name of the desired cluster. Choose **Services** > **ZooKeeper** > **Instance**, and obtain the IP address of the ZooKeeper instance. ● On MRS Manager, choose **Cluster** and click the name of the desired cluster. Choose **Services** > **ZooKeeper** > **Configurations** > **All Configurations**, search for the **clientPort** parameter, and obtain its value, that is, the ZooKeeper port number.
zookeeper.znode.parent	No	/hbase	String	Root directory in ZooKeeper. The default value is **/hbase**.

Parameter	Mandatory	Default Value	Data Type	Description
null-string-literal	No	null	String	Representation for null values for string fields. The HBase sink encodes/decodes empty bytes as null values for all types except the string type.
sink.buffer-flush.max-size	No	2mb	MemorySize	Maximum size in memory of buffered rows for each write request. This can improve performance for writing data to the HBase database, but may increase the latency. You can set this parameter to **0** to disable it.
sink.buffer-flush.max-rows	No	1000	Integer	Maximum number of rows to buffer for each write request. This can improve performance for writing data to the HBase database, but may increase the latency. You can set this parameter to **0** to disable it.
sink.buffer-flush.interval	No	1s	Duration	Interval for refreshing the buffer, during which data is refreshed by asynchronous threads. This can improve performance for writing data to the HBase database, but may increase the latency. You can set this parameter to **0** to disable it. Note: If **sink.buffer-flush.max-size** and **sink.buffer-flush.max-rows** are both set to **0** and the buffer refresh interval is configured, the buffer is asynchronously refreshed. The format is *{length value}{time unit label}*, for example, **123ms, 321s**. The supported time units include **d**, **h**, **min**, **s**, and **ms** (default unit).
sink.parallelism	No	None	Integer	Defines the parallelism of the HBase sink operator. By default, the parallelism is determined by the framework: using the same parallelism as the upstream join operator.

Parameter	Mandatory	Default Value	Data Type	Description
properties.connector.auth.open	No	None	Boolean	**true** indicates that Kerberos authentication is enabled for the HBase cluster. This parameter is mandatory if Kerberos authentication is enabled.
properties.connector.kerberos.principal	No	None	String	Username for logging in to the security cluster. This parameter is mandatory if Kerberos authentication is enabled.
properties.connector.kerberos.keytab	No	None	String	OBS path to which the **user.keytab** file is uploaded. This parameter is mandatory if Kerberos authentication is enabled.
properties.connector.kerberos.krb5	No	None	String	OBS path to which the **krb5.conf** file is uploaded. This parameter is mandatory if Kerberos authentication is enabled. Note: The **renew_lifetime** configuration item under **[libdefaults]** must be removed from **krb5.conf**. Otherwise, the "Message stream modified (41)" error may occur.

Data Type Mapping

HBase stores all data as byte arrays. The data needs to be serialized and deserialized during read and write operations.

When serializing and de-serializing, Flink HBase connector uses utility class **org.apache.hadoop.hbase.util.Bytes** provided by HBase (Hadoop) to convert Flink data types to and from byte arrays.

Flink HBase connector encodes null values to empty bytes, and decodes empty bytes to null values for all data types except the string type. For string type, the null literal is determined by the **null-string-literal** option.

Table 1-43 Data type mapping

Flink SQL Type	HBase Conversion
CHAR/VARCHAR/STRING	byte[] toBytes(String s) String toString(byte[] b)
BOOLEAN	byte[] toBytes(boolean b) boolean toBoolean(byte[] b)
BINARY/VARBINARY	Returns byte[] as is.

Flink SQL Type	HBase Conversion
DECIMAL	byte[] toBytes(BigDecimal v) BigDecimal toBigDecimal(byte[] b)
TINYINT	new byte[] { val } bytes[0] // returns first and only byte from bytes
SMALLINT	byte[] toBytes(short val) short toShort(byte[] bytes)
INT	byte[] toBytes(int val) int toInt(byte[] bytes)
BIGINT	byte[] toBytes(long val) long toLong(byte[] bytes)
FLOAT	byte[] toBytes(float val) float toFloat(byte[] bytes)
DOUBLE	byte[] toBytes(double val) double toDouble(byte[] bytes)
DATE	Stores the number of days since epoch as an int value.
TIME	Stores the number of milliseconds of the day as an int value.
TIMESTAMP	Stores the milliseconds since epoch as a long value.
ARRAY	Not supported
MAP / MULTISET	Not supported
ROW	Not supported

Example

In this example, data is read from the Kafka data source and written to the HBase result table. The procedure is as follows (the HBase version used in this example is 2.2.3):

1. Create an enhanced datasource connection in the VPC and subnet where HBase and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**. Add MRS host information for the enhanced datasource connection by referring to **Modifying Host Information**.
2. Set HBase and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the HBase and Kafka

addresses by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

3. Use the HBase shell to create HBase table **order** that has only one column family **detail**. For details, see **Using HBase from Scratch**.

```
create 'order', {NAME => 'detail'}
```

4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses Kafka as the data source and HBase as the result table (the Rowkey is **order_id** and the column family name is **detail**).

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

create table hbaseSink(
  order_id string,
  detail Row(
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
    pay_time string,
    user_id string,
    user_name string,
    area_id string)
) with (
  'connector' = 'hbase-2.2',
  'table-name' = 'order',
  'zookeeper.quorum' = 'ZookeeperAddress:ZookeeperPort',
  'sink.buffer-flush.max-rows' = '1'
);

insert into hbaseSink select order_id,
Row(order_channel,order_time,pay_amount,real_pay,pay_time,user_id,user_name,area_id) from orders;
```

5. Connect to the Kafka cluster and enter the following data to Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

6. Run the following statement on the HBase shell to view the data result:
```
scan 'order'
```
The data result is as follows:
```
202103241000000001   column=detail:area_id, timestamp=2021-12-16T21:30:37.954, value=330106

202103241000000001   column=detail:order_channel, timestamp=2021-12-16T21:30:37.954, 
value=webShop

202103241000000001   column=detail:order_time, timestamp=2021-12-16T21:30:37.954, 
value=2021-03-24 10:00:00

202103241000000001   column=detail:pay_amount, timestamp=2021-12-16T21:30:37.954, value=@Y
\x00\x00\x00\x00\x00\x00

202103241000000001   column=detail:pay_time, timestamp=2021-12-16T21:30:37.954, 
value=2021-03-24 10:02:03

202103241000000001   column=detail:real_pay, timestamp=2021-12-16T21:30:37.954, value=@Y
\x00\x00\x00\x00\x00\x00

202103241000000001   column=detail:user_id, timestamp=2021-12-16T21:30:37.954, value=0001

202103241000000001   column=detail:user_name, timestamp=2021-12-16T21:30:37.954, value=Alice

202103241606060001   column=detail:area_id, timestamp=2021-12-16T21:30:44.842, value=330106

202103241606060001   column=detail:order_channel, timestamp=2021-12-16T21:30:44.842, 
value=appShop

202103241606060001   column=detail:order_time, timestamp=2021-12-16T21:30:44.842, 
value=2021-03-24 16:06:06

202103241606060001   column=detail:pay_amount, timestamp=2021-12-16T21:30:44.842, value=@i
\x00\x00\x00\x00\x00\x00

202103241606060001   column=detail:pay_time, timestamp=2021-12-16T21:30:44.842, 
value=2021-03-24 16:10:06

202103241606060001   column=detail:real_pay, timestamp=2021-12-16T21:30:44.842, value=@f
\x80\x00\x00\x00\x00\x00

202103241606060001   column=detail:user_id, timestamp=2021-12-16T21:30:44.842, value=0001

202103241606060001   column=detail:user_name, timestamp=2021-12-16T21:30:44.842, value=Alice

202103251202020001   column=detail:area_id, timestamp=2021-12-16T21:30:52.181, value=330110

202103251202020001   column=detail:order_channel, timestamp=2021-12-16T21:30:52.181, 
value=miniAppShop

202103251202020001   column=detail:order_time, timestamp=2021-12-16T21:30:52.181, 
value=2021-03-25 12:02:02

202103251202020001   column=detail:pay_amount, timestamp=2021-12-16T21:30:52.181, value=@N
\x00\x00\x00\x00\x00\x00

202103251202020001   column=detail:pay_time, timestamp=2021-12-16T21:30:52.181, 
value=2021-03-25 12:03:00

202103251202020001   column=detail:real_pay, timestamp=2021-12-16T21:30:52.181, value=@N
\x00\x00\x00\x00\x00\x00

202103251202020001   column=detail:user_id, timestamp=2021-12-16T21:30:52.181, value=0002

202103251202020001   column=detail:user_name, timestamp=2021-12-16T21:30:52.181, value=Bob
```

FAQ

Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
org.apache.zookeeper.ClientCnxn$SessionTimeoutException: Client session timed out, have not heard from server in 90069ms for connection id 0x0
```

A: The datasource connection is not bound or the binding fails. Configure the datasource connection by referring to **Enhanced Datasource Connection** or configure the security group of the Kafka cluster to allow access from the DLI queue.

1.5.9.3 Dimension Table

Function

Create an HBase dimension table to connect to the source streams for wide table generation.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- **If MRS HBase is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.**

  For details, see **Modifying Host Information**.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- All the column families in HBase table must be declared as ROW type, the field name maps to the column family name, and the nested field names map to the column qualifier names. There is no need to declare all the families and qualifiers in the schema, users can declare what is used in the query. Except the ROW type fields, the single atomic type field (for example, STRING, BIGINT) will be recognized as HBase rowkey. The rowkey field can be an arbitrary name, but should be quoted using backticks if it is a reserved keyword.

Syntax

```
create table hbaseSource (
  attr_name attr_type
  (',' attr_name attr_type)*
 )
with (
  'connector' = 'hbase-2.2',
  'table-name' = '',
  'zookeeper.quorum' = ''
);
```

Parameters

Table 1-44 Parameter description

Parameter	Mandatory	Default Value	Type	Description
connector	Yes	None	String	Connector type. Set this parameter to **hbase-2.2**.
table-name	Yes	None	String	Name of the HBase table
zookeeper.quorum	Yes	None	String	HBase Zookeeper quorum. The format is ZookeeperAddress:ZookeeperPort. The following describes how to obtain the ZooKeeper IP address and port number: • On the MRS Manager console, choose **Cluster** > *Name of the desired cluster* > **Service** > **ZooKeeper** > **Instance**. On the displayed page, obtain the IP address of the ZooKeeper instance. • On the MRS Manager console, choose **Cluster** > *Name of the desired cluster* > **Service** > **ZooKeeper** > **Configuration**, and click **All Configurations**. Search for the **clientPort** parameter, and obtain the ZooKeeper port number.
zookeeper.znode.parent	No	/hbase	String	Root directory in ZooKeeper for the HBase cluster.
lookup.async	No	false	Boolean	Whether async lookup is enabled.

Parameter	Mandatory	Default Value	Type	Description
lookup.cache.max-rows	No	-1	Long	Maximum number of cached rows in a dimension table. When the rows exceed this value, the first item added to the cache will be marked as expired. Lookup cache is disabled by default.
lookup.cache.ttl	No	-1	Long	Maximum time to live (TTL) for each row in lookup cache. Caches exceeding the TTL will be expired. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit). Lookup cache is disabled by default.
lookup.max-retries	No	3	Integer	Maximum retry times if lookup database failed.
krb_auth_name	No	None	String	Name of datasource authentication of the Kerberos type created on DLI. **Creating a Datasource Authentication**

Data Type Mapping

HBase stores all data as byte arrays. The data needs to be serialized and deserialized during read and write operations.

When serializing and de-serializing, Flink HBase connector uses utility class **org.apache.hadoop.hbase.util.Bytes** provided by HBase (Hadoop) to convert Flink data types to and from byte arrays.

Flink HBase connector encodes null values to empty bytes, and decodes empty bytes to null values for all data types except the string type. For string type, the null literal is determined by the **null-string-literal** option.

Table 1-45 Data type mapping

Flink SQL Type	HBase Conversion
CHAR/VARCHAR/STRING	byte[] toBytes(String s) String toString(byte[] b)

Flink SQL Type	HBase Conversion
BOOLEAN	byte[] toBytes(boolean b) boolean toBoolean(byte[] b)
BINARY/VARBINARY	Returns byte[] as is.
DECIMAL	byte[] toBytes(BigDecimal v) BigDecimal toBigDecimal(byte[] b)
TINYINT	new byte[] { val } bytes[0] // returns first and only byte from bytes
SMALLINT	byte[] toBytes(short val) short toShort(byte[] bytes)
INT	byte[] toBytes(int val) int toInt(byte[] bytes)
BIGINT	byte[] toBytes(long val) long toLong(byte[] bytes)
FLOAT	byte[] toBytes(float val) float toFloat(byte[] bytes)
DOUBLE	byte[] toBytes(double val) double toDouble(byte[] bytes)
DATE	Number of days since 1970-01-01 00:00:00 UTC. The value is an integer.
TIME	Number of milliseconds since 1970-01-01 00:00:00 UTC. The value is an integer.
TIMESTAMP	Number of milliseconds since 1970-01-01 00:00:00 UTC. The value is of the long type.
ARRAY	Not supported
MAP / MULTISET	Not supported
ROW	Not supported

Example

In this example, data is read from a DMS Kafka data source, an HBase table is used as a dimension table to generate a wide table, and the result is written to a Kafka result table. The procedure is as follows (the HBase version in this example is 2.2.3):

1. Create an enhanced datasource connection in the VPC and subnet where HBase and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**. Add MRS host information for the enhanced datasource connection. For details, see **Modifying Host Information**.
2. Set HBase and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the HBase and Kafka addresses by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Create an HBase table and name it **area_info** using the HBase shell. The table has only one column family **detail**. For details, see **Using HBase from Scratch**. The creation statement is as follows:
```
create 'area_info', {NAME => 'detail'}
```
4. Run the following statement in the HBase shell to insert dimension table data:
```
put 'area_info', '330106', 'detail:area_province_name', 'a1'
put 'area_info', '330106', 'detail:area_city_name', 'b1'
put 'area_info', '330106', 'detail:area_county_name', 'c2'
put 'area_info', '330106', 'detail:area_street_name', 'd2'
put 'area_info', '330106', 'detail:region_name', 'e1'

put 'area_info', '330110', 'detail:area_province_name', 'a1'
put 'area_info', '330110', 'detail:area_city_name', 'b1'
put 'area_info', '330110', 'detail:area_county_name', 'c4'
put 'area_info', '330110', 'detail:area_street_name', 'd4'
put 'area_info', '330110', 'detail:region_name', 'e1'
```
5. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses Kafka as the data source and an HBase table as the dimension table. Data is output to a Kafka result table.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Set the values of the parameters in bold in the following script as needed.**
```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaSourceTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

-- Create an address dimension table
create table area_info (
  area_id string,
  detail row(
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string)
) WITH (
```

```
  'connector' = 'hbase-2.2',
  'table-name' = 'area_info',
  'zookeeper.quorum' = 'ZookeeperAddress:ZookeeperPort',
  'lookup.async' = 'true',
  'lookup.cache.max-rows' = '10000',
  'lookup.cache.ttl' = '2h'
);

-- Generate a wide table based on the address dimension table containing detailed order information.
create table order_detail(
    order_id string,
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
    pay_time string,
    user_id string,
    user_name string,
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) with (
  'connector' = 'kafka',
  'topic' = '<yourSinkTopic>',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'json'
);

insert into order_detail
    select orders.order_id, orders.order_channel, orders.order_time, orders.pay_amount, orders.real_pay,
orders.pay_time, orders.user_id, orders.user_name,
           area.area_id, area.area_province_name, area.area_city_name, area.area_county_name,
           area.area_street_name, area.region_name  from orders
    left join area_info for system_time as of orders.proctime as area on orders.area_id = area.area_id;
```

6. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

7. Connect to the Kafka cluster and read data from the sink topic of Kafka. The result data is as follows:

```
{"order_id":"202103241000000001","order_channel":"webShop","order_time":"2021-03-24
10:00:00","pay_amount":100.0,"real_pay":100.0,"pay_time":"2021-03-24
10:02:03","user_id":"0001","user_name":"Alice","area_id":"330106","area_province_name":"a1","area_ci
ty_name":"b1","area_county_name":"c2","area_street_name":"d2","region_name":"e1"}

{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106","area_province_name":"a1","area_ci
ty_name":"b1","area_county_name":"c2","area_street_name":"d2","region_name":"e1"}

{"order_id":"202103251202020001","order_channel":"miniAppShop","order_time":"2021-03-25
12:02:02","pay_amount":60.0,"real_pay":60.0,"pay_time":"2021-03-25
12:03:00","user_id":"0002","user_name":"Bob","area_id":"330110","area_province_name":"a1","area_cit
y_name":"b1","area_county_name":"c4","area_street_name":"d4","region_name":"e1"}
```

FAQs

Q: What should I do if Flink job logs contain the following error information?

```
org.apache.zookeeper.ClientCnxn$SessionTimeoutException: Client session timed out, have not heard from server in 90069ms for connection id 0x0
```

A: The datasource connection is not bound or the binding fails. Configure the datasource connection by referring to **Enhanced Datasource Connection** or configure the security group of the Kafka cluster to allow access from the DLI queue.

1.5.10 Hive

1.5.10.1 Creating a Hive Catalog

Introduction

Catalogs provide metadata, such as databases, tables, partitions, views, and functions and information needed to access data stored in a database or other external systems.

One of the most crucial aspects of data processing is managing metadata. It may be transient metadata like temporary tables, or UDFs registered against the table environment. Or permanent metadata, like that in a Hive Metastore. Catalogs provide a unified API for managing metadata and making it accessible from the Table API and SQL Queries. For details, see **Apache Flink Catalogs**.

Function

The HiveCatalog serves two purposes; as persistent storage for pure Flink metadata, and as an interface for reading and writing existing Hive metadata.

Flink's **Hive documentation** provides full details on setting up the catalog and interfacing with an existing Hive installation. For details, see **Apache Flink Hive Catalog**.

HiveCatalog can be used to handle two kinds of tables: Hive-compatible tables and generic tables.

- Hive-compatible tables are those stored in a Hive-compatible way, in terms of both metadata and data in the storage layer. Therefore, Hive-compatible tables created via Flink can be queried from Hive side.
- Generic tables, on the other hand, are specific to Flink. When creating generic tables with HiveCatalog, we're just using HMS to persist the metadata. While these tables are visible to Hive, it is unlikely Hive is able to understand the metadata. And therefore using such tables in Hive leads to undefined behavior.

You are advised to switch to Hive dialect to create Hive-compatible tables. If you want to create Hive-compatible tables with default dialect, make sure to set **'connector'='hive'** in your table properties, otherwise a table is considered generic by default in HiveCatalog. Note that the connector property is not required if you use Hive dialect. Refer to **Hive Dialect**.

Caveats

- Warning: The Hive Metastore stores all meta-object names in lower case.
- If a directory with the same name already exists, an exception is thrown.
- To use Hudi tables, you need to use the Hudi catalog, which is not compatible with the Hive catalog.
- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.

Syntax

```
CREATE CATALOG myhive
WITH (
    'type' = 'hive',
    'default-database' = 'default',
    'hive-conf-dir' = '/opt/flink/conf'
);

USE CATALOG myhive;
```

Parameter Description

Table 1-46 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
type	Yes	None	String	Catalog type. Set this parameter to **hive** when creating a HiveCatalog.
hive-conf-dir	Yes	None	String	This refers to the URI that points to the directory containing the **hive-site.xml** file. The value is fixed at **'hive-conf-dir' = '/opt/flink/conf'**.
default-database	No	default	String	When a catalog is set as the current catalog, the default current database is used.

Supported Types

The HiveCatalog supports universal tables for all Flink types.

For Hive-compatible tables, the HiveCatalog needs to map Flink data types to their corresponding Hive types.

Table 1-47 Data type mapping

Flink SQL Type	Hive Data Type
CHAR(p)	CHAR(p)
VARCHAR(p)	VARCHAR(p)
STRING	STRING
BOOLEAN	BOOLEAN
TINYINT	TINYINT
SMALLINT	SMALLINT
INT	INT
BIGINT	LONG
FLOAT	FLOAT
DOUBLE	DOUBLE
DECIMAL(p, s)	DECIMAL(p, s)
DATE	DATE
TIMESTAMP(9)	TIMESTAMP
BYTES	BINARY
ARRAY<T>	LIST<T>
MAP	MAP
ROW	STRUCT

NOTE

- The maximum length for Hive's CHAR(p) is 255.
- The maximum length for Hive's VARCHAR(p) is 65535.
- Hive's MAP only supports primitive types as keys, while Flink's MAP can be any data type.
- Hive does not support UNION types.
- Hive's TIMESTAMP always has a precision of 9 and does not support other precisions. However, Hive UDFs can handle TIMESTAMP values with precision <= 9.
- Hive does not support Flink's **TIMESTAMP_WITH_TIME_ZONE**, **TIMESTAMP_WITH_LOCAL_TIME_ZONE**, and **MULTISET**.
- Flink's INTERVAL type cannot yet be mapped to Hive's INTERVAL type.

Example

1. Create a catalog named **myhive** in the Flink OpenSource SQL job and use it to manage metadata.
```
CREATE CATALOG myhive WITH (
    'type' = 'hive'
```

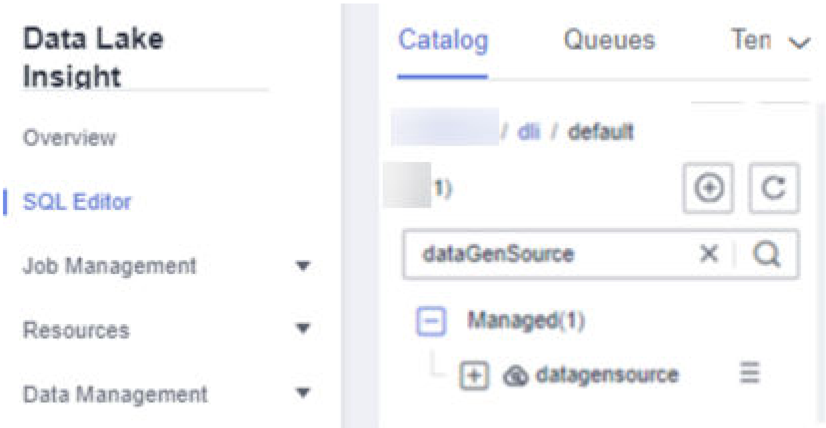
```
    ,'hive-conf-dir' = '/opt/flink/conf'
);

USE CATALOG myhive;

create table dataGenSource(
  user_id string,
  amount int
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1', --Generates a piece of data per second.
  'fields.user_id.kind' = 'random', --Specifies a random generator for the user_id field.
  'fields.user_id.length' = '3' --Limits the length of user_id to 3.
);

create table printSink(
  user_id string,
  amount int
) with (
  'connector' = 'print'
);

insert into printSink select * from dataGenSource;
```

2. Check if the **dataGenSource** and **printSink** tables exist in the **default** database.

   NOTE

   The Hive Metastore stores all meta-object names in lower case.

   Figure 1-1 Checking the default database

3. Create a new Flink OpenSource SQL job using the metadata in the catalog named **myhive**.

```
CREATE CATALOG myhive WITH (
    'type' = 'hive'
    ,'hive-conf-dir' = '/opt/flink/conf'
);

USE CATALOG myhive;

insert into printSink select * from dataGenSource;
```

1.5.10.2 Hive Dialect

Introduction

Starting from 1.11.0, Flink allows users to write SQL statements in Hive syntax when Hive dialect is used. By providing compatibility with Hive syntax, we aim to improve the interoperability with Hive and reduce the scenarios when users need to switch between Flink and Hive in order to execute different statements. For details, see **Apache Flink Hive Dialect**.

Function

Flink currently supports two SQL dialects: default and hive. You need to switch to Hive dialect before you can write in Hive syntax. The following describes how to set dialect with SQL Client.

Also notice that you can dynamically switch dialect for each statement you execute. There's no need to restart a session to use a different dialect.

Syntax

SQL dialect can be specified via the **table.sql-dialect** property.

```
set table.sql-dialect=hive;
```

Caveats

- Hive dialect should only be used to process Hive meta objects, and requires the current catalog to be a **HiveCatalog**.
- Hive dialect only supports 2-part identifiers, so you can not specify catalog for an identifier. Refer to **Apache Flink Hive Read & Write** for more details.

  While all Hive versions support the same syntax, whether a specific feature is available still depends on the **Hive version** you use.

  For example, updating database location is only supported in Hive-2.4.0 or later.
- Use **HiveModule** to run DML and DQL.
- Since Flink 1.15 you need to swap flink-table-planner-loader located in **/lib** with **flink-table-planner_2.12** located in **/opt** to avoid the following exception. Please see **FLINK-25128** for more details.

1.5.10.3 Hive Source Table

Introduction

Apache Hive has established itself as a focal point of the data warehousing ecosystem. It serves as not only a SQL engine for big data analytics and ETL, but also a data management platform, where data is discovered, defined, and evolved.

Flink offers a two-fold integration with Hive. The first is to leverage Hive's Metastore as a persistent catalog. The second is to offer Flink as an alternative engine for reading and writing Hive tables. **Overview | Apache Flink**

Starting from 1.11.0, Flink allows users to write SQL statements in Hive syntax when Hive dialect is used. By providing compatibility with Hive syntax, we aim to improve the interoperability with Hive and reduce the scenarios when users need to switch between Flink and Hive in order to execute different statements. For details, see **Apache Flink Hive Dialect**.

Using the HiveCatalog, Apache Flink can be used for unified BATCH and STREAM processing of Apache Hive Tables. This means Flink can be used as a more performant alternative to Hive's batch engine, or to continuously read and write data into and out of Hive tables to power real-time data warehousing applications. **Apache Flink Hive Read & Write**

Function

This section describes how to use Flink to read and write Hive tables, the definition of the Hive source table, parameters used for creating the source table, and sample code. For details, see **Apache Flink Hive Read & Write**.

Flink supports reading data from Hive in both **BATCH** and **STREAMING** modes. When running as a **BATCH** application, Flink will execute its query over the state of the table at the point in time when the query is executed. **STREAMING** reads will continuously monitor the table and incrementally fetch new data as it is made available. Flink will read tables as bounded by default.

STREAMING reads support consuming both partitioned and non-partitioned tables. For partitioned tables, Flink will monitor the generation of new partitions, and read them incrementally when available. For non-partitioned tables, Flink will monitor the generation of new files in the folder and read new files incrementally.

Prerequisites

To create a FileSystem source table, an enhanced datasource connection is required. You can set security group rules as required when you configure the connection.

- For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.
- For details about how to configure security group rules, see **Security Group Overview**.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- For details about how to use data types, see **Format**.
- Flink 1.15 currently only supports creating OBS tables and DLI lakehouse tables using Hive syntax, which is supported by Hive dialect DDL statements.
  - To create an OBS table using Hive syntax:
    - For the default dialect, set **hive.is-external** to **true** in the with properties.
    - For the Hive dialect, use the **EXTERNAL** keyword in the create table statement.
  - To create a DLI lakehouse table using Hive syntax:
    - For the Hive dialect, add **'is_lakehouse'='true'** to the table properties.
- Enable checkpointing.
- You are advised to switch to Hive dialect to create Hive-compatible tables. If you want to create Hive-compatible tables with default dialect, make sure to

set **'connector'='hive'** in your table properties, otherwise a table is considered generic by default in HiveCatalog. Note that the connector property is not required if you use Hive dialect.

- Monitor strategy is to scan all directories/files currently in the location path. Many partitions may cause performance degradation.
- Streaming reads for non-partitioned tables requires that each file be written atomically into the target directory.
- Streaming reading for partitioned tables requires that each partition should be added atomically in the view of hive metastore. If not, new data added to an existing partition will be consumed.
- Streaming reads do not support watermark grammar in Flink DDL. These tables cannot be used for window operators.

Syntax

```
CREATE EXTERNAL TABLE [IF NOT EXISTS] table_name
  [(col_name data_type [column_constraint] [COMMENT col_comment], ... [table_constraint])]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
  [
    [ROW FORMAT row_format]
    [STORED AS file_format]
  ]
  [LOCATION obs_path]
  [TBLPROPERTIES (property_name=property_value, ...)]

row_format:
  : 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]
  | SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, ...)]

file_format:
  : SEQUENCEFILE
  | TEXTFILE
  | RCFILE
  | ORC
  | PARQUET
  | AVRO
  | INPUTFORMAT input_format_classname OUTPUTFORMAT output_format_classname

column_constraint:
  : NOT NULL [[ENABLE|DISABLE] [VALIDATE|NOVALIDATE] [RELY|NORELY]]

table_constraint:
  : [CONSTRAINT constraint_name] PRIMARY KEY (col_name, ...) [[ENABLE|DISABLE] [VALIDATE|
NOVALIDATE] [RELY|NORELY]]
```

Parameter Description

For the semantics of each DDL statement, see **Creating an OBS Table Using the Hive Syntax** and **Hive documentation**.

Table 1-48 TBLPROPERTIES parameters

Parameter	Mandatory	Default Value	Data Type	Description
streaming-source.enable	No	false	Boolean	Enable streaming source or not. Note: Make sure that each partition/file should be written atomically, otherwise the reader may get incomplete data.
streaming-source.partition.include	No	all	String	Option to set the partitions to read, supported options are **all** and **latest**. By default, this parameter is set to **all**. **all** means read all partitions. **latest** only works when the streaming hive source table used as temporal table. **latest** means reading latest partition in order of **streaming-source.partition.order**. Flink supports temporal join the latest hive partition by enabling **streaming-source.enable** and setting **streaming-source.partition.include** to **latest**. At the same time, user can assign the partition compare order and data update interval by configuring following partition-related options.
streaming-source.monitor-interval	No	None	Duration	Time interval for consecutively monitoring partition/file. Notes: The default interval for hive streaming reading is '1 m', the default interval for hive streaming temporal join is '60 m', this is because there's one framework limitation that every TM will visit the Hive metaStore in current hive streaming temporal join implementation which may produce pressure to metaStore, this will improve in the future.

Parameter	Mandatory	Default Value	Data Type	Description
streaming-source.partition-order	No	partition-name	String	The partition order of streaming source, supporting **create-time**, **partition-time**, and **partition-name**. **create-time** compares partition/file creation time, this is not the partition create time in Hive metaStore, but the folder/file modification time in filesystem, if the partition folder somehow gets updated, e.g. add new file into folder, it can affect how the data is consumed. **partition-time** compares the time extracted from partition name. **partition-name** compares partition name's alphabetical order. For a non-partition table, this value should always be **create-time**. By default the value is **partition-name**. The option is equality with deprecated option **streaming-source.consume-order**.
streaming-source.consume-start-offset	No	None	String	Start offset for streaming consuming. How to parse and compare offsets depends on your order. For **create-time** and **partition-time**, should be a timestamp string (yyyy-[m]m-[d]d [hh:mm:ss]). For **partition-time**, will use partition time extractor to extract time from partition. For **partition-name**, is the partition name string (e.g. pt_year=2020/pt_mon=10/pt_day=01).
is_lakehouse	No	None	Boolean	If DLI lakehouse tables using Hive syntax are used, set this parameter to **true**.

- **Source Parallelism Inference**

  By default, Flink infers the hive source parallelism based on the number of splits, and the number of splits is based on the number of files and the number of blocks in the files.

  Flink allows you to flexibly configure the policy of parallelism inference. You can configure the following parameters in TableConfig (note that these parameters affect all sources of the job):

Key	Default	Type	Description
table.exec.hive.infer-source-parallelism	true	Boolean	If it is **true**, source parallelism is inferred according to splits number. If it is **false**, parallelism of source is set by **config**.
table.exec.hive.infer-source-parallelism.max	1000	Integer	Sets max infer parallelism for source operator.

- **Load Partition Splits**

  Multi-thread is used to split hive's partitions. You can use **table.exec.hive.load-partition-splits.thread-num** to configure the thread number. The default value is **3** and the configured value should be greater than 0.

Key	Default	Type	Description
table.exec.hive.load-partition-splits.thread-num	3	Integer	The configured value should be greater than 0.

SQL hints can be used to apply configurations to a Hive table without changing its definition in the Hive metastore. See **Hints | Apache Flink**.

- **Vectorized Optimization upon Read**

  Flink will automatically used vectorized reads of Hive tables when the following conditions are met:

  – Format: ORC or Parquet.

  – Columns without complex data type, like hive types: List, Map, Struct, Union.

    This feature is enabled by default. It may be disabled with the following configuration.

```
table.exec.hive.fallback-mapred-reader=true
```

- **Reading Hive Views**

  Flink is able to read from Hive defined views, but some limitations apply:

  – The Hive catalog must be set as the current catalog before you can query the view. This can be done by either **tableEnv.useCatalog(...)** in Table API or **USE CATALOG ...** in SQL Client.

  – Hive and Flink SQL have different syntax, e.g. different reserved keywords and literals. Make sure the view's query is compatible with Flink grammar.

Example

1. Create an OBS table in Hive syntax using Spark SQL and insert 10 data records. Simulate the data source.

```
CREATE TABLE IF NOT EXISTS demo.student(
  name STRING,
 score DOUBLE)
PARTITIONED BY (classNo INT)
STORED AS PARQUET
LOCATION 'obs://demo/spark.db/student';

INSERT INTO demo.student PARTITION(classNo=1) VALUES ('Alice', 90.0), ('Bob', 80.0), ('Charlie', 
70.0), ('David', 60.0), ('Eve', 50.0), ('Frank', 40.0), ('Grace', 30.0), ('Hank', 20.0), ('Ivy', 10.0), ('Jack', 0.0);
```

2. Demonstrate batch processing using Flink SQL to read data from the Hive syntax OBS table demo.student in batch mode and print it out. Checkpointing is required.

```
CREATE CATALOG myhive WITH (
    'type' = 'hive',
    'default-database' = 'demo',
     'hive-conf-dir' = '/opt/flink/conf'
);

USE CATALOG myhive;

create table if not exists print (
    name STRING,
    score DOUBLE,
    classNo INT)
with ('connector' = 'print');

insert into print
select * from student;
```

Result (out log of TaskManager):

```
+I[Alice, 90.0, 1]
+I[Bob, 80.0, 1]
+I[Charlie, 70.0, 1]
+I[David, 60.0, 1]
+I[Eve, 50.0, 1]
+I[Frank, 40.0, 1]
+I[Grace, 30.0, 1]
+I[Hank, 20.0, 1]
+I[Ivy, 10.0, 1]
+I[Jack, 0.0, 1]
```

3. Demonstrate stream processing by using Flink SQL to read data from the Hive syntax OBS table demo.student in stream mode and print it out.

```
CREATE CATALOG myhive WITH (
    'type' = 'hive' ,
    'default-database' = 'demo',
     'hive-conf-dir' = '/opt/flink/conf'
);

USE CATALOG myhive;

create table if not exists print (
    name STRING,
    score DOUBLE,
    classNo INT)
with ('connector' = 'print');

insert into print
select * from student /*+ OPTIONS('streaming-source.enable' = 'true', 'streaming-source.monitor-
interval' = '3 m') */;
```


The SQL hints function is used. SQL hints can be used to apply configurations to a Hive table without changing its definition in the Hive metastore. For details, see **SQL Hints**.

1.5.10.4 Result Table

Function

This section describes how to use Flink to write Hive tables, the definition of the Hive result table, parameters used for creating the result table, and sample code. For details, see **Apache Flink Hive Read & Write**.

Flink supports writing data to Hive in both **BATCH** and **STREAMING** modes.

- When run as a BATCH application, Flink will write to a Hive table only making those records visible when the Job finishes. BATCH writes support both appending to and overwriting existing tables.
- **STREAMING** writes continuously adding new data to Hive, committing records - making them visible - incrementally. Users control when/how to trigger commits with several properties. Insert overwrite is not supported for streaming write. Please see the **streaming sink** for a full list of available configurations.

Prerequisites

To create a FileSystem source table, an enhanced datasource connection is required. You can set security group rules as required when you configure the connection.

- For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.
- For details about how to configure security group rules, see **Security Group Overview**.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- For details about how to use data types, see **Format**.
- Flink 1.15 currently only supports creating OBS tables and DLI lakehouse tables using Hive syntax, which is supported by Hive dialect DDL statements.
  - To create an OBS table using Hive syntax:
    - For the default dialect, set **hive.is-external** to **true** in the with properties.
    - For the Hive dialect, use the **EXTERNAL** keyword in the create table statement.
  - To create a DLI lakehouse table using Hive syntax:
    - For the Hive dialect, add **'is_lakehouse'='true'** to the table properties.

- When creating a Flink OpenSource SQL job, enable checkpointing in the job editing interface.

Syntax

```
CREATE EXTERNAL TABLE [IF NOT EXISTS] table_name
  [(col_name data_type [column_constraint] [COMMENT col_comment], ... [table_constraint])]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
  [
    [ROW FORMAT row_format]
    [STORED AS file_format]
  ]
  [LOCATION obs_path]
  [TBLPROPERTIES (property_name=property_value, ...)]

row_format:
  : 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]
  | SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, ...)]

file_format:
  : SEQUENCEFILE
  | TEXTFILE
  | RCFILE
  | ORC
  | PARQUET
  | AVRO
  | INPUTFORMAT input_format_classname OUTPUTFORMAT output_format_classname

column_constraint:
  : NOT NULL [[ENABLE|DISABLE] [VALIDATE|NOVALIDATE] [RELY|NORELY]]

table_constraint:
  : [CONSTRAINT constraint_name] PRIMARY KEY (col_name, ...) [[ENABLE|DISABLE] [VALIDATE|
NOVALIDATE] [RELY|NORELY]]
```

Parameter Description

For the semantics of each DDL statement, see **Creating an OBS Table Using the Hive Syntax** and **Hive documentation**.

Please see the **streaming sink** for a full list of available configurations.

Example

The following example demonstrates how to use Datagen to write to a Hive table with partition submission functionality.

```
CREATE CATALOG myhive WITH (
    'type' = 'hive' ,
    'default-database' = 'demo',
    'hive-conf-dir' = '/opt/flink/conf'
);

USE CATALOG myhive;

SET table.sql-dialect=hive;

-- drop table demo.student_hive_sink;
CREATE EXTERNAL TABLE IF NOT EXISTS demo.student_hive_sink(
  name STRING,
 score DOUBLE)
PARTITIONED BY (classNo INT)
```



```
STORED AS PARQUET
LOCATION 'obs://demo/spark.db/student_hive_sink'
TBLPROPERTIES (
   'sink.partition-commit.policy.kind'='metastore,success-file'
 );

SET table.sql-dialect=default;
create table if not exists student_datagen_source(
  name STRING,
  score DOUBLE,
  classNo INT
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1', --Generates a piece of data per second.
  'fields.name.kind' = 'random', --Specifies a random generator for the user_id field.
  'fields.name.length' = '7', --Limits the user_id length to 7.
  'fields.classNo.kind' ='random',
  'fields.classNo.min' = '1',
  'fields.classNo.max' = '10'
);

insert into student_hive_sink select * from student_datagen_source;
```

Query the result table using Spark SQL.

```
select * from  demo.student_hive_sink where classNo > 0 limit 10
```

Figure 1-2 Query result table

name	score	classno
75e336b	2.4012401223362918e+307	9
b9ba493	3.6761005680423526e+307	9
4452975	9.094319166166612e+307	9
b6045c2	9.381453353163197e+307	8
875b407	1.2156965381708504e+308	8
4234a3f	1.218424475141448e+308	5

1.5.10.5 Hive Dimension Table

Function

You can use Hive tables as temporal tables and associate them through temporal joins. For more information on temporal joins, refer to **temporal join**.

Flink supports processing-time temporal joins with Hive tables, which always join the latest version of the temporal table. Flink supports temporary joins with both partitioned and non-partitioned Hive tables. For partitioned tables, Flink automatically tracks the latest partition of the Hive table. For details, see **Apache Flink Hive Read & Write**.

Caveats

- Currently, Flink does not support event-time temporal joins with Hive tables.
- The "Temporal Join The Latest Partition" feature is only supported in Flink STREAMING mode.
- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- For details about how to use data types, see **Format**.

- Flink 1.15 currently only supports creating OBS tables and DLI lakehouse tables using Hive syntax, which is supported by Hive dialect DDL statements.
  – To create an OBS table using Hive syntax:
    - For the default dialect, set **hive.is-external** to **true** in the with properties.
    - For the Hive dialect, use the **EXTERNAL** keyword in the create table statement.
  – To create a DLI lakehouse table using Hive syntax:
    - For the Hive dialect, add **'is_lakehouse'='true'** to the table properties.
- When creating a Flink OpenSource SQL job, enable checkpointing in the job editing interface.

Syntax Format and Parameter Description

For details, see the syntax format and parameter description in **Hive Source Table**.

1.5.10.6 Using Temporal Join to Associate the Latest Partition of a Dimension Table

Function

For partitioned tables that change over time, we can read them as unbounded streams. If each partition contains a complete set of data for a certain version, the partition can be considered as a version of the temporal table, which retains the data of the partition. Flink supports automatically tracking the latest partition (version) of the temporal table in processing-time joins.

The latest partition (version) is defined by the **streaming-source.partition-order** parameter.

This is the most common use case for using Hive tables as dimension tables in Flink streaming applications.

Caveats

Using Temporal join to associate the latest partition of a dimension table is only supported in Flink STREAMING mode.

Example

The following example shows a classic business pipeline where the dimension table comes from Hive and is updated once a day through batch processing or Flink jobs. The Kafka stream comes from real-time online business data or logs and needs to be joined with the dimension table to expand the stream.

1. Create a Hive OBS external table using Spark SQL and insert data.

```
CREATE TABLE if not exists dimension_hive_table (
  product_id STRING,
  product_name STRING,
  unit_price DECIMAL(10, 4),
```

```
  pv_count BIGINT,
  like_count BIGINT,
  comment_count BIGINT,
  update_time TIMESTAMP,
  update_user STRING
)
STORED AS PARQUET
LOCATION 'obs://demo/spark.db/dimension_hive_table'
PARTITIONED BY (
    create_time   STRING
);
```

```
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_11', 'product_name_11', 1.2345, 100, 50, 20, '2023-11-25 02:10:58', 'update_user_1');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_12', 'product_name_12', 2.3456, 200, 100, 40, '2023-11-25 02:10:58', 'update_user_2');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_13', 'product_name_13', 3.4567, 300, 150, 60, '2023-11-25 02:10:58', 'update_user_3');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_14', 'product_name_14', 4.5678, 400, 200, 80, '2023-11-25 02:10:58', 'update_user_4');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_15', 'product_name_15', 5.6789, 500, 250, 100, '2023-11-25 02:10:58', 'update_user_5');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_16', 'product_name_16', 6.7890, 600, 300, 120, '2023-11-25 02:10:58', 'update_user_6');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_17', 'product_name_17', 7.8901, 700, 350, 140, '2023-11-25 02:10:58', 'update_user_7');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_18', 'product_name_18', 8.9012, 800, 400, 160, '2023-11-25 02:10:58', 'update_user_8');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_19', 'product_name_19', 9.0123, 900, 450, 180, '2023-11-25 02:10:58', 'update_user_9');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_10', 'product_name_10', 10.1234, 1000, 500, 200, '2023-11-25 02:10:58', 'update_user_10');
```

2. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. This job simulates reading data from Kafka, performs a join with a Hive dimension table to denormalize the data, and outputs it to Print.

   Change the values of the parameters in bold as needed in the following script.

```
CREATE CATALOG myhive WITH (
    'type' = 'hive' ,
    'default-database' = 'demo',
     'hive-conf-dir' = '/opt/flink/conf'
);

USE CATALOG myhive;

CREATE TABLE if not exists ordersSource (
  product_id STRING,
  user_name string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'TOPIC',
  'properties.bootstrap.servers' = 'KafkaIP:PROT,KafkaIP:PROT,KafkaIP:PROT',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

create table if not exists print (
  product_id STRING,
  user_name string,
  product_name STRING,
  unit_price DECIMAL(10, 4),
  pv_count BIGINT,
  like_count BIGINT,
  comment_count BIGINT,
  update_time TIMESTAMP,
  update_user STRING,
  create_time   STRING
```

```
) with (
  'connector' = 'print'
);

insert into print
select
  orders.product_id,
  orders.user_name,
  dim.product_name,
  dim.unit_price,
  dim.pv_count,
  dim.like_count,
  dim.comment_count,
  dim.update_time,
  dim.update_user,
  dim.create_time
from ordersSource orders
left join dimension_hive_table /*+ OPTIONS('streaming-source.enable'='true',
   'streaming-source.partition.include' = 'latest', 'streaming-source.monitor-interval' = '10 m') */
   for system_time as of orders.proctime as dim on orders.product_id = dim.product_id;
```

3. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

```
{"product_id": "product_id_11", "user_name": "name11"}
{"product_id": "product_id_12", "user_name": "name12"}
```

4. View the data in the Print result table.

```
+I[product_id_11, name11, product_name_11, 1.2345, 100, 50, 20, 2023-11-24T18:10:58,
update_user_1, create_time_1]
+I[product_id_12, name12, product_name_12, 2.3456, 200, 100, 40, 2023-11-24T18:10:58,
update_user_2, create_time_1]
```

5. Simulate inserting new partition data into the Hive dimension table.

```
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_21', 'product_name_21', 1.2345, 100, 50, 20, '2023-11-25 02:10:58', 'update_user_1');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_22', 'product_name_22', 2.3456, 200, 100, 40, '2023-11-25 02:10:58', 'update_user_2');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_23', 'product_name_23', 3.4567, 300, 150, 60, '2023-11-25 02:10:58', 'update_user_3');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_24', 'product_name_24', 4.5678, 400, 200, 80, '2023-11-25 02:10:58', 'update_user_4');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_25', 'product_name_25', 5.6789, 500, 250, 100, '2023-11-25 02:10:58', 'update_user_5');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_26', 'product_name_26', 6.7890, 600, 300, 120, '2023-11-25 02:10:58', 'update_user_6');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_27', 'product_name_27', 7.8901, 700, 350, 140, '2023-11-25 02:10:58', 'update_user_7');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_28', 'product_name_28', 8.9012, 800, 400, 160, '2023-11-25 02:10:58', 'update_user_8');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_29', 'product_name_29', 9.0123, 900, 450, 180, '2023-11-25 02:10:58', 'update_user_9');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_2') VALUES
('product_id_20', 'product_name_20', 10.1234, 1000, 500, 200, '2023-11-25 02:10:58', 'update_user_10');
```

6. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka. Associate the data from the previous partition with **create_time='create_time_1'**:

```
 {"product_id": "product_id_13", "user_name": "name13"}
```

7. View the data in the Print result table. The data of the previous partition **create_time='create_time_1'** in the Hive dimension table has been deleted.

```
 +I[product_id_13, name13, null, null, null, null, null, null, null, null]
```

8. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka. Associate the latest partition data with **create_time='create_time_2'**:

```
 {"product_id": "product_id_21", "user_name": "name21"}
```

9. View the data in the Print result table. The Hive dimension table retains the data of the latest partition with **create_time='create_time_2'**.

```
+I[product_id_21, name21, product_name_21, 1.2345, 100, 50, 20, 2023-11-24T18:10:58,
update_user_1, create_time_2]
```

1.5.10.7 Using Temporal Join to Associate the Latest Version of a Dimension Table

Function

For Hive tables, we can read them as bounded streams. In this case, the Hive table can only track its latest version when queried. The latest version of the table retains all the data of the Hive table.

Caveats

- Each joining subtask needs to keep its own cache of the Hive table. Make sure the Hive table can fit into the memory of a TM task slot.
- It is encouraged to set a relatively large value both for **streaming-source.monitor-interval** (latest partition as temporal table) or **lookup.join.cache.ttl** (all partitions as temporal table). Otherwise, jobs are prone to performance issues as the table needs to be updated and reloaded too frequently.
- Currently we simply load the whole Hive table whenever the cache needs refreshing. There's no way to differentiate new data from the old.

Parameter Description

When performing the temporal join the latest Hive table, the Hive table will be cached in Slot memory and each record from the stream is joined against the table by key to decide whether a match is found. Using the latest Hive table as a temporal table does not require any additional configuration. Optionally, you can configure the TTL of the Hive table cache with the following property. After the cache expires, the Hive table will be scanned again to load the latest data.

Parameter	Default Value	Data Type	Description
lookup.join.cache.ttl	60 min	Duration	The cache TTL (e.g. 10 min) for the build table in lookup join. By default the TTL is 60 minutes. The option only works when looking up bounded hive table source, if you are using streaming hive source as temporal table, use **streaming-source.monitor-interval** to configure the interval of data update.

Example

The example shows a classic business pipeline where the dimension table comes from Hive and is updated once a day through batch processing or Flink jobs. The

Kafka stream comes from real-time online business data or logs and needs to be joined with the dimension table to expand the stream.

1. Create a Hive OBS external table using Spark SQL and insert data.

```
CREATE TABLE if not exists dimension_hive_table (
  product_id STRING,
  product_name STRING,
  unit_price DECIMAL(10, 4),
  pv_count BIGINT,
  like_count BIGINT,
  comment_count BIGINT,
  update_time TIMESTAMP,
  update_user STRING
)
STORED AS PARQUET
LOCATION 'obs://demo/spark.db/dimension_hive_table'
PARTITIONED BY (
   create_time   STRING
);
```

```
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_11', 'product_name_11', 1.2345, 100, 50, 20, '2023-11-25 02:10:58', 'update_user_1');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_12', 'product_name_12', 2.3456, 200, 100, 40, '2023-11-25 02:10:58', 'update_user_2');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_13', 'product_name_13', 3.4567, 300, 150, 60, '2023-11-25 02:10:58', 'update_user_3');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_14', 'product_name_14', 4.5678, 400, 200, 80, '2023-11-25 02:10:58', 'update_user_4');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_15', 'product_name_15', 5.6789, 500, 250, 100, '2023-11-25 02:10:58', 'update_user_5');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_16', 'product_name_16', 6.7890, 600, 300, 120, '2023-11-25 02:10:58', 'update_user_6');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_17', 'product_name_17', 7.8901, 700, 350, 140, '2023-11-25 02:10:58', 'update_user_7');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_18', 'product_name_18', 8.9012, 800, 400, 160, '2023-11-25 02:10:58', 'update_user_8');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_19', 'product_name_19', 9.0123, 900, 450, 180, '2023-11-25 02:10:58', 'update_user_9');
INSERT INTO dimension_hive_table PARTITION (create_time='create_time_1') VALUES
('product_id_10', 'product_name_10', 10.1234, 1000, 500, 200, '2023-11-25 02:10:58', 'update_user_10');
```

2. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. This job simulates reading data from Kafka, performs a join with a Hive dimension table to denormalize the data, and outputs it to Print.

   Change the values of the parameters in bold as needed in the following script.

```
CREATE CATALOG myhive WITH (
    'type' = 'hive' ,
    'default-database' = 'demo',
     'hive-conf-dir' = '/opt/flink/conf'
);

USE CATALOG myhive;

CREATE TABLE if not exists ordersSource (
  product_id STRING,
  user_name string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'TOPIC',
  'properties.bootstrap.servers' = 'KafkaIP:PROT,KafkaIP:PROT,KafkaIP:PROT',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

create table if not exists print (
  product_id STRING,
  user_name string,
```

```
  product_name STRING,
  unit_price DECIMAL(10, 4),
  pv_count BIGINT,
  like_count BIGINT,
  comment_count BIGINT,
  update_time TIMESTAMP,
  update_user STRING,
  create_time   STRING
) with (
  'connector' = 'print'
);

insert into print
select
  orders.product_id,
  orders.user_name,
  dim.product_name,
  dim.unit_price,
  dim.pv_count,
  dim.like_count,
  dim.comment_count,
  dim.update_time,
  dim.update_user,
  dim.create_time
from ordersSource orders
left join dimension_hive_table /*+ OPTIONS('lookup.join.cache.ttl'='60 m') */
   for system_time as of orders.proctime as dim on orders.product_id = dim.product_id;
```

3. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

```
{"product_id": "product_id_11", "user_name": "name11"}
{"product_id": "product_id_12", "user_name": "name12"}
```

4. View the data in the Print result table.

```
+I[product_id_11, name11, product_name_11, 1.2345, 100, 50, 20, 2023-11-24T18:10:58, 
update_user_1, create_time_1]
+I[product_id_12, name12, product_name_12, 2.3456, 200, 100, 40, 2023-11-24T18:10:58, 
update_user_2, create_time_1]
```

1.5.11 JDBC

Function

The JDBC connector is provided by Apache Flink and can be used to read data from and write data to common databases, such as MySQL and PostgreSQL. Source tables, result tables, and dimension tables are supported.

Table 1-49 Supported types

Type	Description
Supported Table Types	Source table, dimension table, and result table

Prerequisites

- An enhanced datasource connection with the database has been established, so that you can configure security group rules as required.
- For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.
- For details about how to configure security group rules, see **Security Group Overview**.

Caveats

- The JDBC sink operates in upsert mode for exchanging UPDATE/DELETE messages with the external system if a primary key is defined on the DDL, otherwise, it operates in append mode and does not support to consume UPDATE/DELETE messages.
- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.

Syntax

```
create table jbdcTable (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
) with (
  'connector' = 'jdbc',
  'url' = '',
  'table-name' = '',
  'username' = '',
  'password' = ''
);
```

Description

Table 1-50 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **jdbc**.
url	Yes	None	String	Database URL • To connect to a MySQL database, the format is **jdbc:mysql://** *MySQL address*:*MySQL port*/*Database name*. • To connect to a PostgreSQL database, the format is **jdbc:postgresql://** *PostgreSQL address*:*PostgreSQL port*/*Database name*.
table-name	Yes	None	String	Name of the table where the data will be read from the database

Parameter	Mandatory	Default Value	Data Type	Description
driver	No	None	String	Driver required for connecting to the database. If you do not set this parameter, the automatically extracted URL will be used. ● The default driver of the MySQL database is **com.mysql.jdbc.Driver**. ● The default driver of the PostgreSQL database is **org.postgresql.Driver**.
username	No	None	String	Database authentication user name. This parameter must be configured in pair with **password**.
password	No	None	String	Database authentication password. This parameter must be configured in pair with **username**.
connection.max-retry-timeout	No	60s	Duration	Maximum timeout between retries. The timeout should be in second granularity and should not be smaller than 1 second.
scan.partition.column	No	None	String	Name of the column used to partition the input. For details, see **Partitioned Scan**.
scan.partition.num	No	None	Integer	Number of partitions to be created. For details, see **Partitioned Scan**.
scan.partition.lower-bound	No	None	Integer	Lower bound of values to be fetched for the first partition. For details, see **Partitioned Scan**.
scan.partition.upper-bound	No	None	Integer	Upper bound of values to be fetched for the last partition. For details, see **Partitioned Scan**.
scan.fetch-size	No	0	Integer	Number of rows fetched from the database each time. If this parameter is set to **0**, the SQL hint is ignored.
scan.auto-commit	No	true	Boolean	Whether each statement is committed in a transaction automatically.
lookup.cache.max-rows	No	None	Integer	Maximum number of rows in the lookup cache. When the rows exceed this value, the first item added to the cache will be marked as expired. By default, the lookup cache is not enabled. For details, see **Lookup Cache Functions**.

Parameter	Mandatory	Default Value	Data Type	Description
lookup.cache.ttl	No	None	Duration	Maximum survival time of each record in the lookup cache. When the rows exceed this value, the first item added to the cache will be marked as expired. By default, the lookup cache is not enabled. For details, see **Lookup Cache Functions**.
lookup.cache.caching-missing-key	No	true	Boolean	Whether to cache empty query results. The default value is **true**. For details, see **Lookup Cache Functions**.
lookup.max-retries	No	3	Integer	Maximum number of retry attempts when a database query fails.
sink.buffer-flush.max-rows	No	100	Integer	Maximum number of cached records before flushing, which can be set to **0** to disable it.
sink.buffer-flush.interval	No	1s	Duration	The interval for flushing, after which the asynchronous thread will flush the data. Can be set to **0** to disable it. To fully handle the flush events of the cache asynchronously, **sink.buffer-flush.max-rows** can be set to **0** and an appropriate flush time interval can be configured.
sink.max-retries	No	3	Integer	Maximum number of retries after a failed attempt to write records to the database.
sink.parallelism	No	None	Integer	Defines the parallelism of the JDBC sink operator. By default, the parallelism is determined by the framework: using the same parallelism as the upstream chained operator.

Partitioned Scan

To accelerate reading data in parallel Source task instances, Flink provides the partitioned scan feature for the JDBC table. The following parameters describe how to partition the table when reading in parallel from multiple tasks.

- **scan.partition.column**: name of the column used to partition the input. The data type of the column must be number, date, or timestamp.
- **scan.partition.num**: number of partitions.
- **scan.partition.lower-bound**: minimum value of the first partition.
- **scan.partition.upper-bound**: maximum value of the last partition.

NOTE

- **When a table is created, the preceding partitioned scan parameters must all be specified if any of them is specified.**
- The **scan.partition.lower-bound** and **scan.partition.upper-bound** parameters are used to decide the partition stride instead of filtering rows in the table. All rows in the table are partitioned and returned.

Lookup Cache Functions

The JDBC connector can be used as a lookup dimension table in temporal table joins, and currently only supports synchronous lookup mode.

By default, lookup cache is disabled. Therefore, all requests are sent to the external database. You can set **lookup.cache.max-rows** and **lookup.cache.ttl** to enable this feature. The main purpose of the lookup cache is to improve the performance of the JDBC connector in temporal table joins.

When the lookup cache is enabled, each process (i.e. TaskManager) will maintain a cache. Flink will first look up the cache, and only when the cache is not found will it send a request to the external database and update the cache with the returned data. When the cache hits the maximum cache rows **lookup.cache.max-rows** or when the rows exceed the maximum survival time **lookup.cache.ttl**, the first item added to the cache will be marked as expired. The records in the cache may not be the latest, and users can set **lookup.cache.ttl** to a smaller value to get better data refresh, but this may increase the number of requests sent to the database. Therefore, a balance between throughput and correctness should be maintained.

By default, Flink caches empty query results for primary keys, but you can switch this behavior by setting **lookup.cache.caching-missing-key** to **false**.

Data Type Mapping

Table 1-51 Data type mapping

MySQL Type	PostgreSQL Type	Flink SQL Type
TINYINT	-	TINYINT
SMALLINT TINYINT UNSIGNED	SMALLINT INT2 SMALLSERIAL SERIAL2	SMALLINT
INT MEDIUMINT SMALLINT UNSIGNED	INTEGER SERIAL	INT
BIGINT INT UNSIGNED	BIGINT BIGSERIAL	BIGINT

MySQL Type	PostgreSQL Type	Flink SQL Type
BIGINT UNSIGNED	-	DECIMAL(20, 0)
BIGINT	BIGINT	BIGINT
FLOAT	REAL FLOAT4	FLOAT
DOUBLE DOUBLE PRECISION	FLOAT8 DOUBLE PRECISION	DOUBLE
NUMERIC(p, s) DECIMAL(p, s)	NUMERIC(p, s) DECIMAL(p, s)	DECIMAL(p, s)
BOOLEAN TINYINT(1)	BOOLEAN	BOOLEAN
DATE	DATE	DATE
TIME [(p)]	TIME [(p)] [WITHOUT TIMEZONE]	TIME [(p)] [WITHOUT TIMEZONE]
DATETIME [(p)]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]
CHAR(n) VARCHAR(n) TEXT	CHAR(n) CHARACTER(n) VARCHAR(n) CHARACTER VARYING(n) TEXT	STRING
BINARY VARBINARY BLOB	BYTEA	BYTES
-	ARRAY	ARRAY

Example

- **Example 1: Use JDBC as the data source and Print as the result table to read data from an RDS MySQL database and write it into the Print result table.**

a. Create an enhanced datasource connection in the VPC and subnet where RDS MySQL locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.

b. Set RDS MySQL security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the RDS address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.

c. Log in to the RDS MySQL database, create table **orders** in the Flink database, and insert data. For details about how to create a database, see **Creating a Database**.

   Create table **orders** in the Flink database.

```
CREATE TABLE `flink`.`orders` (
    `order_id` VARCHAR(32) NOT NULL,
    `order_channel` VARCHAR(32) NULL,
    PRIMARY KEY (`order_id`)
)   ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4
    COLLATE = utf8mb4_general_ci;
```

   Insert data into the table.

```
insert into orders(
  order_id,
  order_channel
) values
  ('1', 'webShop'),
  ('2', 'miniAppShop');
```

d. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

   Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.

```
CREATE TABLE jdbcSource (
  order_id string,
  order_channel string
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://MySQLAddress:MySQLPort/flink',--flink is the database name created in
RDS MySQL.
  'table-name' = 'orders',
  'username' = 'MySQLUsername',
  'password' = 'MySQLPassword',
  'scan.fetch-size' = '10',
  'scan.auto-commit' = 'true'
);

CREATE TABLE printSink (
  order_id string,
  order_channel string
) WITH (
  'connector' = 'print'
);

insert into printSink select * from jdbcSource;
```

e. View the data result in the **taskmanager.out** file. The data result is as follows:

```
+I(1,webShop)
+I(2,miniAppShop)
```

- **Example 2: Send data using the DataGen source table and output data to a MySQL database through the JDBC result table.**

a. Create an enhanced datasource connection in the VPC and subnet where RDS MySQL locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.

b. Set RDS MySQL security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the RDS address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.

c. Log in to the RDS MySQL database, create table **orders** in the Flink database, and insert data. For details about how to create a database, see **Creating a Database**.

Create table **orders** in the Flink database.

```
CREATE TABLE `flink`.`orders` (
    `order_id` VARCHAR(32) NOT NULL,
    `order_channel` VARCHAR(32) NULL,
    PRIMARY KEY (`order_id`)
)    ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4
    COLLATE = utf8mb4_general_ci;
```

d. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE dataGenSource (
  order_id string,
  order_channel string
) WITH (
  'connector' = 'datagen',
  'fields.order_id.kind' = 'sequence',
  'fields.order_id.start' = '1',
  'fields.order_id.end' = '1000',
  'fields.order_channel.kind' = 'random',
  'fields.order_channel.length' = '5'
);

CREATE TABLE jdbcSink (
  order_id string,
  order_channel string,
  PRIMARY KEY(order_id) NOT ENFORCED
) WITH (
  'connector' = 'jdbc',
  'url? = 'jdbc:mysql://MySQLAddress:MySQLPort/flink',-- flink is the MySQL database where the orders table locates.
  'table-name' = 'orders',
  'username' = 'MySQLUsername',
  'password' = 'MySQLPassword',
  'sink.buffer-flush.max-rows' = '1'
);

insert into jdbcSink select * from dataGenSource;
```

e. Run the SQL statement in the MySQL database to view data in the table:

```
select * from orders;
```

- **Example 3: Read data from the DataGen source table, use the JDBC table as the dimension table, and write the table information generated by both into the Print result table.**
  a. Create an enhanced datasource connection in the VPC and subnet where RDS MySQL locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
  b. Set RDS MySQL security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the RDS address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
  c. Log in to the RDS MySQL database, create table **orders** in the Flink database, and insert data. For details about how to create a database, see **Creating a Database**.

     Create table **orders** in the Flink database.

```
CREATE TABLE `flink`.`orders` (
    `order_id` VARCHAR(32) NOT NULL,
    `order_channel` VARCHAR(32) NULL,
    PRIMARY KEY (`order_id`)
)    ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4
    COLLATE = utf8mb4_general_ci;
```

     Insert data into the table.

```
insert into orders(
  order_id,
  order_channel
) values
  ('1', 'webShop'),
  ('2', 'miniAppShop');
```

  d. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. This job script uses DataGen as the data source and JDBC as the dimension table to write data into the Print result table.

     When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE dataGenSource (
  order_id string,
  order_time timestamp,
  proctime as Proctime()
) WITH (
  'connector' = 'datagen',
  'fields.order_id.kind' = 'sequence',
  'fields.order_id.start' = '1',
  'fields.order_id.end' = '2'
);

--Creating a dimension table
CREATE TABLE jdbcTable (
  order_id string,
  order_channel string
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://JDBC address:JDBC port/flink',--flink is the name of the database where the orders table of RDS for MySQL is located.
  'table-name' = 'orders',
  'username' = 'JDBCUserName',
  'password' = 'JDBCPassWord',
  'lookup.cache.max-rows' = '100',
  'lookup.cache.ttl' = '1000',
```

```
  'lookup.cache.caching-missing-key' = 'false',
  'lookup.max-retries' = '5'
);

CREATE TABLE printSink (
  order_id string,
  order_time timestamp,
  order_channel string
) WITH (
  'connector' = 'print'
);

insert into
  printSink
SELECT
  dataGenSource.order_id, dataGenSource.order_time, jdbcTable.order_channel
from
  dataGenSource
  left join jdbcTable for system_time as of dataGenSource.proctime on dataGenSource.order_id =
jdbcTable.order_id;
```

e. View the data result in the **taskmanager.out** file. The data result is as follows:

```
+I(1, xxx, webShop)
+I(2, xxx, miniAppShop)
```

FAQ

None

1.5.12 Kafka

Function

The Kafka connector allows for reading data from and writing data into Kafka topics.

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.

Table 1-52 Supported types

Type	Description
Supported Table Types	Source table and result table

Type	Description
Supported Data Formats	**CSV** **JSON** **Apache Avro** **Confluent Avro** **Debezium CDC** **Canal CDC** **Maxwell CDC** **OGG CDC** **Raw**

Prerequisites

- You have created a Kafka cluster.
- An enhanced datasource connection has been created for DLI to connect to Kafka clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.
  - For details about how to configure security group rules, see **Security Group Overview**.

Caveats

- For details, see **Apache Kafka SQL Connector**.
- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- Fields in the **with** parameter can only be enclosed in single quotes.
- For details about how to use data types when creating tables, see **Format**.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.

Syntax

```
create table kafkaSource(
  attr_name attr_type
```

```
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' WATERMARK FOR rowtime_column_name AS watermark-strategy_expression)
)
with (
  'connector' = 'kafka',
  'topic' = '',
  'properties.bootstrap.servers' = '',
  'properties.group.id' = '',
  'scan.startup.mode' = '',
  'format' = ''
);
```

Source Table Parameter Description

Table 1-53 Source table parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Specify what connector to use, for Kafka use **kafka**.
topic	No	None	String	Topic name(s) to read data from when the table is used as source. It also supports topic list for source by separating topic by semicolon like **topic-1;topic-2**. Note, only one of **topic-pattern** and **topic** can be specified for sources. When the table is used as sink, the topic name is the topic to write data to. Note topic list is not supported for sinks.
topic-pattern	No	None	String	The regular expression for a pattern of topic names to read from. All topics with names that match the specified regular expression will be subscribed by the consumer when the job starts running. Note, only one of **topic-pattern** and **topic** can be specified for sources. For more information, see **Topic and Partition Discovery**.
properties.bootstrap.servers	Yes	None	String	Comma separated list of Kafka brokers.

Parameter	Mandatory	Default Value	Data Type	Description
properties.group.id	optional for source, not applicable for sink	None	String	The ID of the consumer group for Kafka source. If group ID is not specified, an automatically generated ID **KafkaSource-***{tableIdentifier}* will be used.
properties.*	No	None	String	This can set and pass arbitrary Kafka configurations. • Suffix names must match the configuration key defined in **Apache Kafka**. Flink will remove the **properties.** key prefix and pass the transformed key and values to the underlying KafkaClient. For example, you can disable automatic topic creation via **'properties.allow.auto.create.topics' = 'false'**. • But there are some configurations that do not support to set, because Flink will override them, e.g. **key.deserializer** and **value.deserializer**.
format	Yes	None	String	The format used to deserialize and serialize the value part of Kafka messages. Either this parameter or the **value.format** parameter is required. • For details about the message key and body of Kafka messages, see **Key and Value Formats**. • Refer to **Format** for more details and format parameters.

Parameter	Mandatory	Default Value	Data Type	Description
key.format	No	None	String	The format used to deserialize and serialize the key part of Kafka messages. • If a key format is defined, the **key.fields** parameter is required as well. Otherwise the Kafka records will have an empty key. • Refer to **Format** for more details and format parameters.
key.fields	No	[]	List<String>	Defines an explicit list of physical columns from the table schema that configure the data type for the key format. By default, this list is empty and thus a key is undefined. The list should look like **field1;field2**.
key.fields-prefix	No	None	String	Defines a custom prefix for all fields of the key format to avoid name clashes with fields of the value format. By default, the prefix is empty. If a custom prefix is defined, both the table schema and **key.fields** will work with prefixed names. When constructing the data type of the key format, the prefix will be removed and the non-prefixed names will be used within the key format. Note that this parameter requires that **value.fields-include** must be set to **EXCEPT_KEY**.
value.format	No	None	String	The format used to deserialize and serialize the value part of Kafka messages. • Either this parameter or the **format** parameter is required. If two parameters are configured, a conflict occurs. • Refer to **Format** for more details and format parameters.

Parameter	Mandatory	Default Value	Data Type	Description
value.fields-include	No	ALL	Enum Possible values: [ALL, EXCEPT_KEY]	Defines a strategy how to deal with key columns in the data type of the value format. By default, **ALL** physical columns of the table schema will be included in the value format which means that key columns appear in the data type for both the key and value format.
scan.startup.mode	No	group-offsets	String	Startup mode for Kafka consumer. Valid values are: • **earliest-offset**: start from the earliest offset possible. • **latest-offset**: start from the latest offset. • **group-offsets**: start from committed offsets in ZooKeeper/Kafka brokers of a specific consumer group. • **timestamp**: start from user-supplied timestamp for each partition. • **specific-offsets**: start from user-supplied specific offsets for each partition, and the position is specified by **scan.startup.specific-offsets**.
scan.startup.specific-offsets	No	None	String	Specify offsets for each partition in case of **specific-offsets** startup mode, e.g. **partition:0,offset:42;partition:1,offset:300**.
scan.startup.timestamp-millis	No	None	Long	Start from the specified epoch timestamp (milliseconds) used in case of **timestamp** startup mode.
scan.topic-partition-discovery.interval	No	None	Duration	Interval for consumer to discover dynamically created Kafka topics and partitions periodically.

Result Table Parameters

Table 1-54 Result table parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Specify what connector to use, for Kafka use **kafka**.
topic	No	None	String	Topic name(s) to read data from when the table is used as source. It also supports topic list for source by separating topic by semicolon like **topic-1;topic-2**. Note, only one of **topic-pattern** and **topic** can be specified for sources. When the table is used as sink, the topic name is the topic to write data to. Note topic list is not supported for sinks.
properties.bootstrap.servers	Yes	None	String	Comma separated list of Kafka brokers.
properties.*	No	None	String	This can set and pass arbitrary Kafka configurations. • Suffix names must match the configuration key defined in **Apache Kafka**. Flink will remove the **properties.** key prefix and pass the transformed key and values to the underlying KafkaClient. For example, you can disable automatic topic creation via **'properties.allow.auto.create.topics' = 'false'**. • But there are some configurations that do not support to set, because Flink will override them, e.g. **key.deserializer** and **value.deserializer**.

Parameter	Mandatory	Default Value	Data Type	Description
format	Yes	None	String	The format used to deserialize and serialize the value part of Kafka messages. Note, either this parameter or the **value.format** parameter is required. ● For details about the message key and body of Kafka messages, see **Key and Value Formats**. ● Refer to **Format** for more details and format parameters.
key.format	No	None	String	The format used to deserialize and serialize the key part of Kafka messages. ● If a key format is defined, the **key.fields** parameter is required as well. Otherwise the Kafka records will have an empty key. ● Refer to **Format** for more details and format parameters.
key.fields	No	[]	List<String>	Defines an explicit list of physical columns from the table schema that configure the data type for the key format. By default, this list is empty and thus a key is undefined. The list should look like **field1;field2**.
key.fields-prefix	No	None	String	Defines a custom prefix for all fields of the key format to avoid name clashes with fields of the value format. By default, the prefix is empty. If a custom prefix is defined, both the table schema and **key.fields** will work with prefixed names. When constructing the data type of the key format, the prefix will be removed and the non-prefixed names will be used within the key format. Note that this parameter requires that **value.fields-include** must be set to **EXCEPT_KEY**.

Parameter	Mandatory	Default Value	Data Type	Description
value.format	No	None	String	The format used to deserialize and serialize the value part of Kafka messages. • Either this parameter or the **format** parameter is required. If two parameters are configured, a conflict occurs. • Refer to **Format** for more details and format parameters.
value.fields-include	No	ALL	Enum Possible values: [ALL, EXCEPT_KEY]	Defines a strategy how to deal with key columns in the data type of the value format. By default, **ALL** physical columns of the table schema will be included in the value format which means that key columns appear in the data type for both the key and value format.
sink.partitioner	No	'default'	String	Output partitioning from Flink's partitions into Kafka's partitions. Valid values are: • **default**: use the kafka default partitioner to partition records. • **fixed**: each Flink partition ends up in at most one Kafka partition. • **round-robin**: a Flink partition is distributed to Kafka partitions sticky round-robin. It only works when record's keys are not specified. • Custom **FlinkKafkaPartitioner** subclass: e.g. **org.mycompany.MyPartitioner**.
sink.semantic	No	at-least-once	String	Defines the delivery semantic for the Kafka sink. Valid enumerationns are **at-least-once**, **exactly-once**, and **none**.
sink.parallelism	No	None	Integer	Defines the parallelism of the Kafka sink operator. By default, the parallelism is determined by the framework: using the same parallelism as the upstream chained operator.

Metadata

You can define metadata in the source table to obtain the metadata of Kafka messages.

For example, if multiple topics are defined in the **WITH** parameter and metadata is defined in the Kafka source table, the data read by Flink is labeled with the topic from which the data is read.

Table 1-55 Metadata

Key	Data Type	R/W	Description
topic	STRING NOT NULL	R	Topic name of the Kafka record.
partition	INT NOT NULL	R	Partition ID of the Kafka record.
headers	MAP<STRING, BYTES> NOT NULL	R/W	Headers of the Kafka record as a map of raw bytes.
leader-epoch	INT NULL	R	Leader epoch of the Kafka record if available.
offset	BIGINT NOT NULL	R	Offset of the Kafka record in the partition.
timestamp	TIMESTAMP(3) WITH LOCAL TIME ZONE NOT NULL	R/W	Timestamp of the Kafka record.
timestamp-type	STRING NOT NULL	R	Timestamp type of the Kafka record. • **NoTimestampType**: No timestamp is defined in the message. • **CreateTime**: time when the message is generated. • **LogAppendTime**: time when the message is added to the Kafka broker.

Key and Value Formats

Both the key and value part of a Kafka record can be serialized to and deserialized from raw bytes using one of the given **formats**.

- **Value Format**

  Since a key is optional in Kafka records, the following statement reads and writes records with a configured value format but without a key format. The **format** parameter is a synonym for **value.format**. All format options are prefixed with the format identifier.

```
CREATE TABLE KafkaTable (
  `ts` TIMESTAMP(3) METADATA FROM 'timestamp',
  `user_id` BIGINT,
  `item_id` BIGINT,
  `behavior` STRING
) WITH (
  'connector' = 'kafka',
  ...

  'format' = 'json',
  'json.ignore-parse-errors' = 'true'
)
```

  The value format will be configured with the following data type:

```
ROW<`user_id` BIGINT, `item_id` BIGINT, `behavior` STRING>
```

- **Key and Value Format**

  The following example shows how to specify and configure key and value formats. The format options are prefixed with either the **key** or **value** plus format identifier.

```
CREATE TABLE KafkaTable (
  `ts` TIMESTAMP(3) METADATA FROM 'timestamp',
  `user_id` BIGINT,
  `item_id` BIGINT,
  `behavior` STRING
) WITH (
  'connector' = 'kafka',
  ...

  'key.format' = 'json',
  'key.json.ignore-parse-errors' = 'true',
  'key.fields' = 'user_id;item_id',

  'value.format' = 'json',
  'value.json.fail-on-missing-field' = 'false',
  'value.fields-include' = 'ALL'
)
```

  The key format includes the fields listed in **key.fields** (using **;** as the delimiter) in the same order. Thus, it will be configured with the following data type:

```
ROW<`user_id` BIGINT, `item_id` BIGINT>
```

  Since the value format is configured with **'value.fields-include' = 'ALL'**, key fields will also end up in the value format's data type:

```
ROW<`user_id` BIGINT, `item_id` BIGINT, `behavior` STRING>
```

- **Overlapping Format Fields**

  The connector cannot split the table's columns into key and value fields based on schema information if both key and value formats contain fields of the same name. The **key.fields-prefix** parameter allows to give key columns a unique name in the table schema while keeping the original names when configuring the key format.

  The following example shows a key and value format that both contain a version field:

```
CREATE TABLE KafkaTable (
  `k_version` INT,
  `k_user_id` BIGINT,
```

```
  `k_item_id` BIGINT,
  `version` INT,
  `behavior` STRING
) WITH (
  'connector' = 'kafka',
  ...

  'key.format' = 'json',
  'key.fields-prefix' = 'k_',
  'key.fields' = 'k_version;k_user_id;k_item_id',

  'value.format' = 'json',
  'value.fields-include' = 'EXCEPT_KEY'
)
```

The value format must be configured in **EXCEPT_KEY** mode. The formats will be configured with the following data types:

```
Key format:
ROW<`version` INT, `user_id` BIGINT, `item_id` BIGINT>

Value format:
ROW<`version` INT, `behavior` STRING>
```

Topic and Partition Discovery

The config parameters **topic** and **topic-pattern** specify the topics or topic pattern to consume for source. The config parameter **topic** can accept topic list using semicolon separator like **topic-1;topic-2**. The config parameter **topic-pattern** will use regular expression to discover the matched topic. For example, if the **topic-pattern** is **test-topic-[0-9]**, then all topics with names that match the specified regular expression (starting with test-topic- and ending with a single digit)) will be subscribed by the consumer when the job starts running.

To allow the consumer to discover dynamically created topics after the job started running, set a non-negative value for **scan.topic-partition-discovery.interval**. This allows the consumer to discover partitions of new topics with names that also match the specified pattern.

NOTE

Note that topic list and topic pattern only work in sources. In sinks, Flink currently only supports a single topic.

Example 1: Reading DMS Kafka Metadata in CSV Format and Outputting It to a Kafka Sink (Applicable for Kafka Clusters Without SASL_SSL Enabled)

1. Create an enhanced datasource connection in the VPC and subnet where Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see .
2. Set Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to . If the connection passes the test, it is bound to the queue.
3. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE kafkaSource(
  `topic` String metadata virtual,
  `partition` int metadata virtual,
  `headers` MAP<STRING, BYTES> metadata virtual,
  `leader-epoch` INT metadata virtual,
  `offset` bigint metadata virtual,
  `timestamp-type` string metadata virtual,
  `event_time` TIMESTAMP(3) metadata FROM 'timestamp',
  `message` string
) WITH (
  'connector' = 'kafka',
  'topic' = 'SourceKafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'csv',
  'csv.field-delimiter' = '\u0001',
  'csv.quote-character' = ''''
);

CREATE TABLE kafkaSink (
  `topic` String,
  `partition` int,
  `headers` MAP<STRING, BYTES>,
  `leader-epoch` INT,
  `offset` bigint,
  `timestampType` string,
  `event_time` TIMESTAMP(3),
  `message` string -- Indicates that data written by users is read from Kafka.
) WITH (
  'connector' = 'kafka',
  'topic' = 'SinkKafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'json'
);
insert into kafkaSink select * from kafkaSource;
```

4. Send the following data to the topic of the source table in Kafka. The Kafka topic is kafkaSource.

   For details, see **Configuring Kafka Clients in Java**.

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

5. Read the topic of the Kafka result table. The Kafka topic is kafkaSink.

   For details, see **Configuring Kafka Clients in Java**.

```
{"topic":"kafkaSource","partition":1,"headers":{},"leader-epoch":0,"offset":4,"timestampType":"LogAppendTime","event_time":"2023-11-16 11:16:30.369","message":"{\"order_id\":\"202103251202020001\", \"order_channel\":\"miniAppShop\", \"order_time\":\"2021-03-25 12:02:02\", \"pay_amount\":\"60.00\", \"real_pay\":\"60.00\", \"pay_time\":\"2021-03-25 12:03:00\", \"user_id\":\"0002\", \"user_name\":\"Bob\", \"area_id\":\"330110\"}"}

{"topic":"kafkaSource","partition":0,"headers":{},"leader-epoch":0,"offset":6,"timestampType":"LogAppendTime","event_time":"2023-11-16 11:16:30.367","message":"{\"order_id\":\"202103241000000001\",\"order_channel\":\"webShop\",\"order_time\":\"2021-03-24 10:00:00\",\"pay_amount\":100.0,\"real_pay\":100.0,\"pay_time\":\"2021-03-24 10:02:03\",\"user_id\":\"0001\",\"user_name\":\"Alice\",\"area_id\":\"330106\"}"}

{"topic":"kafkaSource","partition":2,"headers":{},"leader-epoch":0,"offset":5,"timestampType":"LogAppendTime","event_time":"2023-11-16
```

```
11:16:30.368","message":"{\"order_id\":\"202103241606060001\",\"order_channel\":\"appShop
\",\"order_time\":\"2021-03-24 16:06:06\",\"pay_amount\":200.0,\"real_pay\":180.0,\"pay_time
\":\"2021-03-24 16:10:06\",\"user_id\":\"0001\",\"user_name\":\"Alice\",\"area_id\":\"330106\"}"}
```

Example 2: Using DMS Kafka in JSON Format as the Source Table and Outputting It to a Kafka Sink (Applicable for Kafka Clusters Without SASL_SSL Enabled)

Use the Kafka source table and Kafka result table to read JSON data from Kafka and output it to the log file.

1. Create an enhanced datasource connection in the VPC and subnet where Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see .
2. Set Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to . If the connection passes the test, it is bound to the queue.
3. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE kafkaSource(
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaSourceTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE kafkaSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaSinkTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'json'
);
insert into kafkaSink select * from kafkaSource;
```

4. Send the following data to the topic of the source table in Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
```

```
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

5. Read the topic of the Kafka result table. The data results are as follows:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

Example 3: Using DMS Kafka as the Source Table and Print as the Result Table (Applicable for Kafka Clusters with SASL_SSL Enabled)

Create a Kafka cluster for DMS, enable SASL_SSL, download the SSL certificate, and upload the downloaded certificate **client.jks** to an OBS bucket.

The **properties.sasl.jaas.config** field contains account passwords encrypted using DEW.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:9093,KafkaAddress2:9093',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.connector.auth.open' = 'true',
  'properties.ssl.truststore.location' = 'obs://xx/client.jks',  -- Location where the user uploads the certificate
to
  'properties.sasl.mechanism' = 'PLAIN',
  'properties.security.protocol' = 'SASL_SSL',
  'properties.sasl.jaas.config' = 'xx',  -- Key in DEW secret management, whose value is like
org.apache.kafka.common.security.plain.PlainLoginModule required username=xx password=xx;
  'format' = 'json',
  'dew.endpoint' = 'kms.xx.com', --Endpoint information for the DEW service being used
  'dew.csms.secretName' = 'xx', --Name of the DEW shared secret
  'dew.csms.decrypt.fields' = 'properties.sasl.jaas.config', --The properties.sasl.jaas.config field value must
be decrypted and replaced using DEW secret management.
  'dew.csms.version' = 'v1'
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
```

```
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);
 insert into ordersSink select * from ordersSource;
```

Example 4: Using Kafka (MRS Cluster) as the Source Table and Print as the Result Table (Applicable for Kafka with SASL_SSL Enabled and MRS Using Kerberos Authentication)

- Enable Kerberos authentication for the MRS cluster.
- Click the **Components** tab and click **Kafka**. On the displayed page, click the **Service Configuration** tab, locate the **ssl.mode.enable**, set it to **true**, and restart Kafka.
- Download the user credential. Log in to the FusionInsight Manager of the MRS cluster and choose **System** > **Permission** > **User**. Locate the row that contains the target user, click **More**, and select **Download Authentication Credential**.

  Obtain the **truststore.jks** file using the authentication credential and store the credential and **truststore.jks** file in OBS.
- If "Message stream modified (41)" is displayed, the JDK version may be incorrect. Change the JDK version in the sample code to a version earlier than 8u_242 or delete the **renew_lifetime = 0m** configuration item from the **krb5.conf** configuration file.
- Set the port to the **sasl_ssl.port** configured in the Kafka service configuration. The default value is **21009**.
- In the following statements, set **security.protocol** to **SASL_SSL**.
- The **properties.ssl.truststore.password** field in the **with** parameter is encrypted using DEW.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.sasl.kerberos.service.name' = 'kafka', -- Value configured in the MRS cluster
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx', --Username
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.security.protocol' = 'SASL_SSL',
  'properties.ssl.truststore.location' = 'obs://xx/truststore.jks',
  'properties.ssl.truststore.password' = 'xx',  -- Key in the DEW secret
  'properties.sasl.mechanism' = 'GSSAPI',
  'format' = 'json',
```

```
  'dew.endpoint'='kms.xx.myhuaweicloud.com', --Endpoint information for the DEW service being used
  'dew.csms.secretName'='xx', --Name of the DEW shared secret
  'dew.csms.decrypt.fields'='properties.ssl.truststore.password', --The properties.ssl.truststore.password field value must be decrypted and replaced using DEW secret management.
  'dew.csms.version'='v1'
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);
 insert into ordersSink select * from ordersSource;
```

Example 5: Using Kafka (MRS Cluster) as the Source Table and Print as the Result Table (Applicable for Kafka with SASL_SSL Enabled and MRS Using SASL_PLAINTEXT with Kerberos Authentication)

- Enable Kerberos authentication for the MRS cluster.
- Click the **Components** tab and click **Kafka**. On the displayed page, click the **Service Configuration** tab, locate the **ssl.mode.enable**, set it to **true**, and restart Kafka.
- Log in to the FusionInsight Manager of the MRS cluster and download the user credential. Choose **System** > **Permission** > **User**. Locate the row that contains the target user, choose **More** > **Download Authentication Credential**. Upload the credential to OBS.
- If error message "Message stream modified (41)" is displayed, the JDK version may be incorrect. Change the JDK version in the sample code to a version earlier than 8u_242 or delete the **renew_lifetime = 0m** configuration item from the **krb5.conf** configuration file.
- Set the port to the **sasl.port** configured in the Kafka service configuration. The default value is **21007**.
- In the following statements, set **security.protocol** to **SASL_PLAINTEXT**.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.sasl.kerberos.service.name' = 'kafka', -- Configured in the MRS cluster
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx',
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
```

```
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'format' = 'json'
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);
 insert into ordersSink select * from ordersSource;
```

Example 6: Using Kafka (MRS Cluster) as the Source Table and Print as the Result Table (Applicable for Kafka with SSL Enabled and MRS Without Kerberos Authentication Enabled)

- Do not enable Kerberos authentication for the MRS cluster.
- Download the user credential. Log in to the FusionInsight Manager of the MRS cluster and choose **System** > **Permission** > **User**. Locate the row that contains the target user, click **More**, and select **Download Authentication Credential**.

  Obtain the **truststore.jks** file using the authentication credential and store the credential and **truststore.jks** file in OBS.
- Set the port to the **ssl.port** configured in the Kafka service configuration. The default value is **9093**.
- Set **security.protocol** in the **with** parameter to **SSL**.
- In the Kafka configuration of the MRS cluster, set **ssl.mode.enable** to **true** and restart Kafka.
- The **properties.ssl.truststore.password** field in the **with** parameter is encrypted using DEW.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.connector.auth.open' = 'true',
  'properties.ssl.truststore.location' = 'obs://xx/truststore.jks',
  'properties.ssl.truststore.password' = 'xx',  -- Key for DEW secret management, whose value is the
password set when generating truststore.jks
  'properties.security.protocol' = 'SSL',
  'format' = 'json',
```

```
  'dew.endpoint' = 'kms.xx.com', --Endpoint information for the DEW service being used
  'dew.csms.secretName' = 'xx', --Name of the DEW shared secret
  'dew.csms.decrypt.fields' = 'properties.ssl.truststore.password', --The
properties.ssl.truststore.password field value must be decrypted and replaced using DEW secret
management.
  'dew.csms.version' = 'v1'
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);
 insert into ordersSink select * from ordersSource;
```

FAQ

- **Q: What should I do if the Flink job execution fails and the log contains the following error information?**

  ```
  org.apache.kafka.common.errors.TimeoutException: Timeout expired while fetching topic metadata
  ```

  A: The datasource connection is not bound, the binding fails, or the security group of the Kafka cluster is not configured to allow access from the network segment of the DLI queue. Reconfigure the datasource connection or configure the security group of the Kafka cluster to allow access from the DLI queue.

  For details, see **Enhanced Datasource Connections**.

- **Q: What should I do if the Flink job execution fails and the log contains the following error information?**

  ```
  Caused by: java.lang.RuntimeException: RealLine:45;Table 'default_catalog.default_database.printSink'
  declares persistable metadata columns, but the underlying DynamicTableSink doesn't implement the
  SupportsWritingMetadata interface. If the column should not be persisted, it can be declared with the
  VIRTUAL keyword.
  ```

  A: The metadata type is defined in the sink table, but the Print connector does not support deletion of matadata from the sink table.

1.5.13 MySQL CDC

Function

The MySQL CDC source table, that is, the MySQL streaming source table, reads all historical data in the database first and then smoothly switches data read to the Binlog to ensure data integrity.

Table 1-56 Supported types

Type	Description
Supported Table Types	Source table

Prerequisites

- MySQL CDC requires MySQL 5.6, 5.7, or 8.0.*x*.
- Fields in the **with** parameter can only be enclosed in single quotes.
- An enhanced datasource connection has been created for DLI to connect to the MySQL database, so that you can configure security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.
  - For details about how to configure security group rules, see **Security Group Overview**.
- Binlog is enabled for MySQL, and **binlog_row_image** is set to **FULL**.
- A MySQL user has been created and granted the **SELECT**, **SHOW DATABASES**, **REPLICATION SLAVE**, and **REPLICATION CLIENT** permissions. Note: When the **scan.incremental.snapshot.enabled** parameter is enabled (enabled by default), there is no need to grant the reload permission.

```
GRANT SELECT, SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'user'
IDENTIFIED BY 'password';
```

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- Set a different SERVER ID for each reader.
  - Each MySQL client used for reading Binlog should have a unique server ID to ensure that the MySQL server can distinguish between different clients and maintain their respective Binlog reading positions.
  - Sharing the same server ID among different jobs may lead to reading data from incorrect Binlog positions, resulting in data inconsistency.
  - You can assign a unique server ID to each source reader through SQL hints, such as using **SELECT * FROM source_table /*+ OPTIONS('server-id'='5401-5404') */ ;** to allocate unique server IDs for four source readers.
- Set up MySQL session timeouts.

  When an initial consistent snapshot is made for large databases, your established connection could time out while the tables are being read. You can prevent this behavior by configuring interactive_timeout and wait_timeout in your MySQL configuration file.
  - **interactive_timeout**: The number of seconds the server waits for activity on an interactive connection before closing it. See **MySQL documentations**.
  - **wait_timeout**: The number of seconds the server waits for activity on a noninteractive connection before closing it. See **MySQL documentations**.

- Precautions when using tables without primary keys:
  - To use a table without primary keys, you must configure the **scan.incremental.snapshot.chunk.key-column** parameter and specify one non-null field.
  - If there is an index in the table, use a column which is contained in the index in **scan.incremental.snapshot.chunk.key-column**. This will increase the speed of select statement.

    The processing semantics of a MySQL CDC table without primary keys is determined based on the behavior of the column that is specified by the **scan.incremental.snapshot.chunk.key-column**.

    - If no update operation is performed on the specified column, the exactly-once semantics is ensured.
    - If the update operation is performed on the specified column, only the at-least-once semantics is ensured. However, you can specify primary keys at downstream and perform the idempotence operation to ensure data correctness.
- Watermarks cannot be defined for MySQL CDC source tables. For details about window aggregation, see **FAQ**.
- If you connect to a sink source that supports upsert, such as GaussDB(DWS) and MySQL, you need to define the primary key in the statement for creating the sink table. For details, see the printSink table creation statement in **Example**.

Features

- Incremental snapshot reading

  Incremental snapshot reading is a new mechanism to read snapshot of a table. Compared to the old snapshot mechanism, the incremental snapshot has many advantages, including:
  - MySQL CDC Source can be parallel during snapshot reading
  - MySQL CDC Source can perform checkpoints in the chunk granularity during snapshot reading
  - MySQL CDC Source does not need to acquire global read lock (FLUSH TABLES WITH READ LOCK) before snapshot reading

  If you would like the source run in parallel, each parallel reader should have a unique server ID, so the **server-id** must be a range like **5400-6400**, and the range must be larger than the parallelism. During the incremental snapshot reading, the MySQL CDC Source firstly splits snapshot chunks (splits) by primary key of table, and then MySQL CDC Source assigns the chunks to multiple readers to read the data of snapshot chunk.
- Lock-free

  The MySQL CDC source uses incremental snapshot algorithm, which avoids acquiring global read lock (FLUSH TABLES WITH READ LOCK) and thus does not need **RELOAD** permission.
- Controlling parallelism

  Incremental snapshot reading provides the ability to read snapshot data parallelly.

- Checkpoint

  Incremental snapshot reading provides the ability to perform checkpoint in chunk level. It resolves the checkpoint timeout problem in previous version with old snapshot reading mechanism.

Syntax

```
create table mySqlCdcSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'mysql-cdc',
  'hostname' = 'mysqlHostname',
  'username' = 'mysqlUsername',
  'password' = 'mysqlPassword',
  'database-name' = 'mysqlDatabaseName',
  'table-name' = 'mysqlTableName'
);
```

Parameter Description

Table 1-57 Source table parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Specify what connector to use, here should be **mysql-cdc**.
hostname	Yes	None	String	IP address or hostname of the MySQL database server.
username	Yes	None	String	Name of the MySQL database to use when connecting to the MySQL database server.
password	Yes	None	String	Password to use when connecting to the MySQL database server.

Parameter	Mandatory	Default Value	Data Type	Description
database-name	Yes	None	String	Database name of the MySQL server to monitor. The **database-name** also supports regular expressions to monitor multiple tables match the regular expression. ● Prefix matching: **^(test).*** matches database names with the prefix **test**, for example, **test1** and **test2**. ● Suffix matching: **.*[p$]** matches database names with the suffix **p**, for example, **cdcp** and **edcp**. ● Specific matching: **txc** matches a specific database name.
table-name	Yes	None	String	Table name of the MySQL database to monitor. The table-name also supports regular expressions to monitor multiple tables that satisfy the regular expressions. **NOTE** When the MySQL CDC connector regularly matches the table name, it will concat the database-name and table-name filled in by the user through the string `\\.` to form a full-path regular expression, and then use the regular expression to match the fully qualified name of the table in the MySQL database. ● Prefix matching: **^(test).*** matches table names with the prefix **test**, for example, **test1** and **test2**. ● Suffix matching: **.*[p$]** matches table names with the suffix **p**, for example, **cdcp** and **edcp**. ● Specific matching: **txc** matches a specific table name.
port	No	3306	Integer	Integer port number of the MySQL database server.

Parameter	Mandatory	Default Value	Data Type	Description
server-id	No	None	String	A numeric ID or a numeric ID range of this database client. The numeric ID syntax is like **5400**, the numeric ID range syntax is like **5400-5408**. The numeric ID range syntax is recommended when **scan.incremental.snapshot.enabled** enabled. Every ID must be unique across all currently-running database processes in the MySQL cluster. This connector joins the MySQL cluster as another server (with this unique ID) so it can read the binlog. By default, a random number is generated between 5400 and 6400, though we recommend setting an explicit value.
scan.incremental.snapshot.enabled	No	true	Boolean	Incremental snapshot is a new mechanism to read snapshot of a table. Compared to the old snapshot mechanism, the incremental snapshot has many advantages, including: ● MySQL CDC Source can be parallel during snapshot reading ● MySQL CDC Source can perform checkpoints in the chunk granularity during snapshot reading ● MySQL CDC Source does not need to acquire global read lock (FLUSH TABLES WITH READ LOCK) before snapshot reading If you would like the source run in parallel, each parallel reader should have a unique server ID, so the **server-id** must be a range like **5400-6400**, and the range must be larger than the parallelism.

Parameter	Mandatory	Default Value	Data Type	Description
scan.incremental.snapshot.chunk.size	No	8096	Integer	The chunk size (number of rows) of table snapshot, captured tables are split into multiple chunks when reading the snapshot of table.
scan.snapshot.fetch.size	No	1024	Integer	The maximum fetch size for per poll when read table snapshot.
scan.startup.mode	No	initial	String	Optional startup mode for MySQL CDC consumer, valid enumerations are **initial**, **earliest-offset**, **latest-offset**, **specific-offset**, and **timestamp**. ● **initial** (default): Perform an initial snapshot on the monitored database tables upon first startup, and continue to read the latest binlog. ● **earliest-offset**: Skip snapshot phase and start reading binlog events from the earliest accessible binlog offset. ● **latest-offset**: Never to perform snapshot on the monitored database tables upon first startup, just read from the end of the binlog which means only have the changes since the connector was started. ● **specific-offset**: Skip snapshot phase and start reading binlog events from a specific offset. The offset could be specified with binlog filename and position, or a GTID set if GTID is enabled on server. ● **timestamp**: Skip snapshot phase and start reading binlog events from a specific timestamp.
scan.startup.specific-offset.file	No	None	String	Optional binlog file name used in case of **specific-offset** startup mode

Parameter	Mandatory	Default Value	Data Type	Description
scan.startup.specific-offset.pos	No	None	Long	Optional binlog file position used in case of **specific-offset** startup mode
scan.startup.specific-offset.gtid-set	No	None	String	Optional GTID set used in case of **specific-offset** startup mode
scan.startup.specific-offset.skip-events	No	None	Long	Optional number of events to skip after the specific starting offset
scan.startup.specific-offset.skip-rows	No	None	Long	Optional number of rows to skip after the specific starting offset
server-time-zone	No	None	String	The session time zone in database server, e.g. **Asia/Shanghai**. It controls how the TIMESTAMP type in MYSQL converted to STRING. If not set, then **ZoneId.systemDefault()** is used to determine the server time zone.
debezium.min.row.count.to.stream.result	No	1000	Integer	During a snapshot operation, the connector will query each included table to produce a read event for all rows in that table. This parameter determines whether the MySQL connection will pull all results for a table into memory (which is fast but requires large amounts of memory), or whether the results will instead be streamed (can be slower, but will work for very large tables). The value specifies the minimum number of rows a table must contain before the connector will stream results, and defaults to **1000**. Set this parameter to **0** to skip all table size checks and always stream all results during a snapshot.

Parameter	Mandatory	Default Value	Data Type	Description
connect.timeout	No	30s	Duration	The maximum time that the connector should wait after trying to connect to the MySQL database server before timing out.
connect.max-retries	No	3	Integer	The max retry times that the connector should retry to build MySQL database server connection.
connection.pool.size	No	20	Integer	The connection pool size.
jdbc.properties.*	No	None	String	Option to pass custom JDBC URL properties. User can pass custom properties like **'jdbc.properties.useSSL' = 'false'**.
heartbeat.interval	No	30s	Duration	The interval of sending heartbeat event for tracing the latest available binlog offsets.
debezium.*	No	None	String	Pass-through Debezium's properties to Debezium Embedded Engine which is used to capture data changes from MySQL server. For example: **'debezium.snapshot.mode' = 'never'**. See more about the **Debezium's MySQL Connector properties**.
scan.incremental.close-idle-reader.enabled	No	false	Boolean	Whether to close idle readers at the end of the snapshot phase. This feature requires that **execution.checkpointing.checkpoints-after-tasks-finish.enabled** be set to **true**.

Metadata

The following format metadata can be exposed as read-only (VIRTUAL) columns in DDL.

Table 1-58 Metadata

Key	Data Type	Description
table_name	STRING NOT NULL	Name of the table that contains the row.
database_name	STRING NOT NULL	Name of the database that contains the row.
op_ts	TIMESTAMP_LTZ(3) NOT NULL	It indicates the time that the change was made in the database. If the record is read from snapshot of the table instead of the binlog, the value is always **0**.

Data Type Mapping

Table 1-59 Data type mapping

MySQL Type	Flink SQL Type	Remarks
TINYINT	TINYINT	-
SMALLINT TINYINT UNSIGNED TINYINT UNSIGNED ZEROFILL	SMALLINT	-
INT MEDIUMINT SMALLINT UNSIGNED SMALLINT UNSIGNED ZEROFILL	INT	-
BIGINT INT UNSIGNED INT UNSIGNED ZEROFILL MEDIUMINT UNSIGNED MEDIUMINT UNSIGNED ZEROFILL	BIGINT	-
BIGINT UNSIGNED BIGINT UNSIGNED ZEROFILL SERIAL	DECIMAL(20, 0)	-

MySQL Type	Flink SQL Type	Remarks
FLOAT FLOAT UNSIGNED FLOAT UNSIGNED ZEROFILL	FLOAT	-
REAL REAL UNSIGNED REAL UNSIGNED ZEROFILL DOUBLE DOUBLE UNSIGNED DOUBLE UNSIGNED ZEROFILL DOUBLE PRECISION DOUBLE PRECISION UNSIGNED DOUBLE PRECISION UNSIGNED ZEROFILL	DOUBLE	-
NUMERIC(p, s) NUMERIC(p, s) UNSIGNED NUMERIC(p, s) UNSIGNED ZEROFILL DECIMAL(p, s) DECIMAL(p, s) UNSIGNE DDECIMAL(p, s) UNSIGNED ZEROFILL FIXED(p, s) FIXED(p, s) UNSIGNED FIXED(p, s) UNSIGNED ZEROFILL where p <= 38	DECIMAL(p, s)	-

MySQL Type	Flink SQL Type	Remarks
NUMERIC(p, s) NUMERIC(p, s) UNSIGNED NUMERIC(p, s) UNSIGNED ZEROFILL DECIMAL(p, s) DECIMAL(p, s) UNSIGNED DECIMAL(p, s) UNSIGNED ZEROFILL FIXED(p, s) FIXED(p, s) UNSIGNED FIXED(p, s) UNSIGNED ZEROFILL where 38 < p <= 65	STRING	The precision for DECIMAL data type is up to 65 in MySQL, but the precision for DECIMAL is limited to 38 in Flink. So if you define a decimal column whose precision is greater than 38, you should map it to STRING to avoid precision loss.
BOOLEAN TINYINT(1) BIT(1)	BOOLEAN	-
DATE	DATE	-
TIME [(p)]	TIME [(p)]	-
TIMESTAMP [(p)] DATETIME [(p)]	TIMESTAMP [(p)]	-
CHAR(n)	CHAR(n)	-
VARCHAR(n)	VARCHAR(n)	-
BIT(n)	BINARY(⌈n/8⌉)	-
BINARY(n)	BINARY(n)	-
VARBINARY(N)	VARBINARY(N)	-
TINYTEXT TEXT MEDIUMTEXT LONGTEXT	STRING	-
TINYBLOB BLOB MEDIUMBLOB LONGBLOB	BYTES	Currently, for BLOB data type in MySQL, only the blob whose length is not greater than 2,147,483,647(2 ** 31 – 1) is supported.
YEAR	INT	-
ENUM	STRING	-

MySQL Type	Flink SQL Type	Remarks
JSON	STRING	The JSON data type will be converted into STRING with JSON format in Flink.
SET	ARRAY<STRING>	As the SET data type in MySQL is a string object that can have zero or more values, it should always be mapped to an array of string.
GEOMETRY POINT LINESTRING POLYGON MULTIPOINT MULTILINESTRING MULTIPOLYGON GEOMETRYCOLLECTION	STRING	The spatial data types in MySQL will be converted into STRING with a fixed Json format.

Example

This example demonstrates the use of MySQL-CDC to read data and metadata from an RDS for MySQL database in real-time and write it to a Print result table.

In this example, the RDS for MySQL database engine version is MySQL 5.7.33.

1. Create an enhanced datasource connection in the VPC and subnet where MySQL locates, and bind the connection to the required Flink elastic resource pool. For details, see .
2. Set MySQL security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the MySQL address by referring to . If the connection passes the test, it is bound to the queue.
3. Create the **test** user in MySQL and grant them permissions. The SQL statements are as follows:
```
CREATE USER 'test'@'%' IDENTIFIED BY 'xxx';
GRANT SELECT, SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'test';
FLUSH PRIVILEGES;
```
4. Create a table named **cdc_order** in the Flink database of MySQL. The SQL statement is as follows (this statement requires the user to have the **CREATE** permission):
```
CREATE TABLE `flink`.`cdc_order` (
    `order_id` VARCHAR(32) NOT NULL,
    `order_channel` VARCHAR(32) NULL,
    `order_time` VARCHAR(32) NULL,
    `pay_amount` DOUBLE  NULL,
    `real_pay` DOUBLE  NULL,
    `pay_time` VARCHAR(32) NULL,
    `user_id` VARCHAR(32) NULL,
```

```
  `user_name` VARCHAR(32) NULL,
  `area_id` VARCHAR(32) NULL,
  PRIMARY KEY (`order_id`)
)   ENGINE = InnoDB
  DEFAULT CHARACTER SET = utf8mb4
  COLLATE = utf8mb4_general_ci;
```

5. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
create table mysqlCdcSource(
  database_name STRING METADATA VIRTUAL,
  table_name STRING METADATA VIRTUAL,
  operation_ts TIMESTAMP_LTZ(3) METADATA FROM 'op_ts' VIRTUAL,
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING,
  primary key(order_id) not enforced
) with (
  'connector' = 'mysql-cdc',
  'hostname' = 'mysqlHostname',
  'username' = 'mysqlUsername',
  'password' = 'mysqlPassword',
  'database-name' = 'mysqlDatabaseName',
  'table-name' = 'mysqlTableName'
);

create table printSink(
  database_name string,
  table_name string,
  operation_ts TIMESTAMP_LTZ(3),
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING,
  primary key(order_id) not enforced
) with (
  'connector' = 'print'
);
insert into printSink select * from mysqlCdcSource;
```

6. Run the following commands in MySQL to insert test data (this statement requires the user to have the corresponding permission):

```
insert into flink.cdc_order values
('202103241000000001','webShop','2021-03-24 10:00:00','100.00','100.00','2021-03-24
10:02:03','0001','Alice','330106'),
('202103241606060001','appShop','2021-03-24 16:06:06','200.00','180.00','2021-03-24
16:10:06','0001','Alice','330106');

delete from flink.cdc_order  where order_channel = 'webShop';
insert into flink.cdc_order values('202103251202020001','miniAppShop','2021-03-25
12:02:02','60.00','60.00','2021-03-25 12:03:00','0002','Bob','330110');
```


7. Perform the following operations to view the data result in the **taskmanager.out** file:
   a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
   b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
   c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I[flink, cdc_order, 2023-11-10T07:41:12Z, 202103241000000001, webShop, 2021-03-24 10:00:00, 100.0, 100.0, 2021-03-24 10:02:03, 0001, Alice, 330106]
+I[flink, cdc_order, 2023-11-10T07:41:12Z, 202103241606060001, appShop, 2021-03-24 16:06:06, 200.0, 180.0, 2021-03-24 16:10:06, 0001, Alice, 330106]
-D[flink, cdc_order, 2023-11-10T07:41:59Z, 202103241000000001, webShop, 2021-03-24 10:00:00, 100.0, 100.0, 2021-03-24 10:02:03, 0001, Alice, 330106]
+I[flink, cdc_order, 2023-11-10T07:42:00Z, 202103251202020001, miniAppShop, 2021-03-25 12:02:02, 60.0, 60.0, 2021-03-25 12:03:00, 0002, Bob, 330110]
```

FAQ

Q: How do I perform window aggregation if the MySQL CDC source table does not support definition of watermarks?

A: You can use the non-window aggregation method. That is, convert the time field into a window value, and then use **GROUP BY** to perform aggregation based on the window value.

For example, you can use the following script to collect statistics on the number of orders per minute (**order_time** indicates the order time, in the string format):

```
insert into printSink select DATE_FORMAT(order_time, 'yyyy-MM-dd HH:mm'), count(*) from mysqlCdcSource group by DATE_FORMAT(order_time, 'yyyy-MM-dd HH:mm');
```

1.5.14 Print

Function

The Print connector is used to print output data to the error file or out file in the TaskManager, making it easier for you to view the result in code debugging.

Prerequisites

None

Caveats

- The Print result table supports the following output formats:

Print	Condition 1	Condition 2
Identifier:Task ID> Output data	A print identifier prefix must be provided. That is, you must specify **print-identifier** in the **WITH** parameter when creating the Print result table.	parallelism > 1
Identifier> Output data	A print identifier prefix must be provided. That is, you must specify **print-identifier** in the **WITH** parameter when creating the Print result table.	parallelism == 1
Task ID> Output data	A print identifier prefix is not needed. That is, you do not specify **print-identifier** in the **WITH** parameter when creating the Print result table.	parallelism > 1
Output data	A print identifier prefix is not needed. That is, you do not specify **print-identifier** in the **WITH** parameter when creating the Print result table.	parallelism == 1

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.

Syntax

```
create table printSink (
  attr_name attr_type
  (',' attr_name attr_type) *
  (',' PRIMARY KEY (attr_name,...) NOT ENFORCED)
) with (
  'connector' = 'print',
  'print-identifier' = '',
  'standard-error' = ''
);
```

Parameter Description

Table 1-60 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **print**.
print-identifier	No	None	String	Message that identifies print and is prefixed to the output of the value.
standard-error	No	false	Boolean	The value can be only **true** or **false**. The default value is **false**. ● If the value is **true**, data is output to the error file of the TaskManager. ● If the value is **false**, data is output to the **out** file of the TaskManager.
sink.parallelism	No	None	Integer	Defines the parallelism of the Print result table. By default, the parallelism is determined by the framework: using the same parallelism as the upstream chained operator.

Example

Create a Flink OpenSource SQL job. Run the following script to generate random data through the DataGen table and output the data to the Print result table.

```
create table dataGenSource(
  user_id string,
  amount int
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1', --Generates a piece of data per second.
  'fields.user_id.kind' = 'random', --Specifies a random generator for the user_id field.
  'fields.user_id.length' = '3' --Limits the length of user_id to 3.
);

create table printSink(
  user_id string,
  amount int
) with (
  'connector' = 'print',
  'print-identifier' = '', --Configure the data prefix.
  'standard-error' = 'false', --Output data to the out file of TaskManager.
  'sink.parallelism' = '2' --Configure the parallelism.
);

insert into printSink select * from dataGenSource;
```

After the job is submitted, the job status changes to **Running**. You can perform the following operations of either method to view the output result:

- Method 1:
  a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  b. Locate the row that contains the target Flink job, and choose **More** > **FlinkUI** in the **Operation** column.
  c. On the Flink UI, choose **Task Managers**, click the task name, and select **Stdout** to view job logs.
- Method 2: If you select **Save Job Log** on the **Running Parameters** tab before submitting the job, perform the following operations:
  a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
  c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.
- Method 3: If the queue is a new version, perform the following operations:
  a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  b. In the job list, click the name of your desired Flink job. On the displayed page, click the **Logs** tab.
  c. Select the corresponding TaskManager name from the drop-down list in the upper left corner and click the **taskmanager.out** file to view its result log.

1.5.15 Redis

1.5.15.1 Source Table

Function

Create a source stream to obtain data from Redis as input for jobs.

Prerequisites

An enhanced datasource connection has been created for DLI to connect to the Redis database, so that you can configure security group rules as required.

- For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.

- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- To obtain the key values, you can set the primary key in Flink. The primary key maps to the Redis key.
- The primary key cannot be a composite primary key, and only can be one field.
- Constraints on **schema-syntax**:
  - If **schema-syntax** is **map** or **array**, there can be only one non-primary key and it must be of the same **map** or **array** type.
  - If **schema-syntax** is **fields-scores**, the number of non-primary keys must be an even number, and the second key of every two keys except the primary key must be of the **double** type. The **double** value is the score of the previous key. The following is an example:
```
CREATE TABLE redisSource (
  redisKey string,
  order_id string,
  score1 double,
  order_channel string,
  score2 double,
  order_time string,
  score3 double,
  pay_amount double,
  score4 double,
  real_pay double,
  score5 double,
  pay_time string,
  score6 double,
  user_id string,
  score7 double,
  user_name string,
  score8 double,
  area_id string,
  score9 double,
  primary key (redisKey) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields-scores'
);
```
- Restrictions on **data-type**:
  - When **data-type** is **set**, the types of non-primary keys defined in Flink must be the same.
  - If **data-type** is **sorted-set** and **schema-syntax** is **fields** or **array**, only **sorted-set** values can be read from Redis, and the **score** value cannot be read.
  - If **data-type** is **string**, only one non-primary key field is allowed.
  - If **data-type** is **sorted-set** and **schema-syntax** is **map**, only one non-primary key field is allowed besides the primary key field.

This non-primary key field must be of the **map** type. The map value of the field must be of the **double** type, indicating the score. The map key of the field indicates the value in the Redis set.

– If **data-type** is **sorted-set** and **schema-syntax** is **array-scores**, only two non-primary keys are allowed and must be of the **array** type.

The first key indicates values in the Redis set. The second key is of the **array<double>** type, indicating index scores. The following is an example:

```
CREATE TABLE redisSink (
  order_id string,
  arrayField Array<String>,
  arrayScore array<double>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  ''default-score'' = '3',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'array-scores'
);
```

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
  ,PRIMARY KEY (attr_name, ...) NOT ENFORCED
)
with (
  'connector' = 'redis',
  'host' = ''
);
```

Parameters

Table 1-61 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **redis**.
host	Yes	None	String	Redis connector address.
port	No	6379	Integer	Redis connector port.
password	No	None	String	Redis authentication password.
namespace	No	None	String	Redis key namespace.
delimiter	No	:	String	Delimiter between the Redis key and namespace.

Parameter	Mandatory	Default Value	Data Type	Description
data-type	No	hash	String	Redis data type. Available values are as follows: • hash • list • set • sorted-set • string For details about the constraints, see **Constraints on data-type**.
schema-syntax	No	fields	String	Redis schema semantics. Available values are as follows (for details, see **Caveats** and **FAQ**): • **fields**: applicable to all data types • **fields-scores**: applicable to **sorted-set** data • **array**: applicable to **list**, **set**, and **sorted-set** data • **array-scores**: applicable to **sorted-set** data • **map**: applicable to **hash** and **sorted-set** data For details about the constraints, see **Constraints on schema-syntax**.
deploy-mode	No	standalone	String	Deployment mode of the Redis cluster. The value can be **standalone**, **master-replica**, or **cluster**. The default value is **standalone**. The deployment mode varies depending on the Redis instance type. Select **standalone** for single-node, master/standby, and Proxy Cluster instances. For a cluster instance, select **cluster**.
retry-count	No	5	Integer	Number of attempts to connect to the Redis cluster.

Parameter	Mandatory	Default Value	Data Type	Description
connection-timeout-millis	No	10000	Integer	Maximum timeout for connecting to the Redis cluster.
commands-timeout-millis	No	2000	Integer	Maximum time for waiting for a completion response.
rebalancing-timeout-millis	No	15000	Integer	Sleep time when the Redis cluster fails.
scan-keys-count	No	1000	Integer	Number of data records read in each scan.
default-score	No	0	Double	Default score when **data-type** is **sorted-set**.
deserialize-error-policy	No	fail-job	Enum	Policy of how to process a data parsing failure. Available values are as follows: ● **fail-job**: Fail the job. ● **skip-row**: Skip the current data. ● **null-field**: Set the current data to null.
skip-null-values	No	true	Boolean	Whether null values will be skipped.
ignore-retractions	No	false	Boolean	The connector should ignore retraction messages in the update insert/withdraw flow mode.
key-column	No	None	String	Schema key of the Redis table.
source.parallelism	No	None	int	Defines the custom parallelism of the source. By default, if this option is not defined, the parallelism from the global configuration is used.

Example

In this example, data is read from the DCS Redis data source and written to the Print result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where Redis locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.

2. Set Redis security groups and add inbound rules to allow access from the Flink queue.

   Test the connectivity using the Redis address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

3. Run the following commands on the Redis client to insert data into different keys and store the data in hash format:

```
HMSET redisSource order_id 202103241000000001 order_channel webShop order_time "2021-03-24 10:00:00" pay_amount 100.00 real_pay 100.00 pay_time "2021-03-24 10:02:03" user_id 0001 user_name Alice area_id 330106

HMSET redisSource1 order_id 202103241606060001 order_channel appShop order_time "2021-03-24 16:06:06" pay_amount 200.00 real_pay 180.00 pay_time "2021-03-24 16:10:06" user_id 0001 user_name Alice area_id 330106

HMSET redisSource2 order_id 202103251202020001 order_channel miniAppShop order_time "2021-03-25 12:02:02" pay_amount 60.00 real_pay 60.00 pay_time "2021-03-25 12:03:00" user_id 0002 user_name Bob area_id 330110
```

4. Create a Flink OpenSource SQL job. Enter the following job script to read data in hash format from Redis.

   Change the values of the parameters in bold as needed in the following script.

```
CREATE TABLE redisSource (
  redisKey string,
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  primary key (redisKey) not enforced  --Obtains the key value from Redis.
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'hash',
  'deploy-mode' = 'master-replica'
);

CREATE TABLE printSink (
  redisKey string,
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);

insert into printSink select * from redisSource;
```

5. Perform the following operations to view the data result in the **taskmanager.out** file:

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

b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.

c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

The data result is as follows:

```
+I(redisSource1,202103241606060001,appShop,2021-03-24 16:06:06,200.0,180.0,2021-03-24
16:10:06,0001,Alice,330106)
+I(redisSource,202103241000000001,webShop,2021-03-24 10:00:00,100.0,100.0,2021-03-24
10:02:03,0001,Alice,330106)
+I(redisSource2,202103251202020001,miniAppShop,2021-03-25 12:02:02,60.0,60.0,2021-03-25
12:03:00,0002,Bob,330110)
```

FAQ

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
Caused by: org.apache.flink.client.program.ProgramInvocationException: The main method caused an error: RealLine:36;Usage of 'set' data-type and 'fields' schema syntax in source Redis connector with multiple non-key column types. As 'set' in Redis is not sorted, it's not possible to map 'set's values to table schema with different types.
```

  A: If **data-type** is **set**, the data types of non-primary key fields in Flink are different. As a result, this error is reported. When **data-type** is **set**, the types of non-primary keys defined in Flink must be the same.

- Q: If **data-type** is **hash**, what are the differences between **schema-syntax** set to **fields** and that to **map**?

  A: When **schema-syntax** is set to **fields**, the hash value in the Redis key is assigned to the field with the same name in Flink. When **schema-syntax** is set to **map**, the hash key and hash value of each hash in Redis are put into a map, which represents the value of the corresponding Flink field. Specifically, this map contains all hash keys and hash values of a key in Redis.

  – For **fields**:

    i. Insert the following data into Redis:

```
HMSET redisSource order_id 202103241000000001 order_channel webShop order_time "2021-03-24 10:00:00" pay_amount 100.00 real_pay 100.00 pay_time "2021-03-24 10:02:03" user_id 0001 user_name Alice area_id 330106
```

    ii. When **schema-syntax** is set to **fields**, use the following job script:

```
CREATE TABLE redisSource (
  redisKey string,
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  primary key (redisKey) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'hash',
  'deploy-mode' = 'master-replica'
);

CREATE TABLE printSink (
  redisKey string,
```

```
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);

insert into printSink select * from redisSource;
```

iii. The job execution result is as follows:

```
+I(redisSource,202103241000000001,webShop,2021-03-24
10:00:00,100.0,100.0,2021-03-24 10:02:03,0001,Alice,330106)
```

– For **map**:

i. Insert the following data into Redis:

```
HMSET redisSource order_id 202103241000000001 order_channel webShop order_time
"2021-03-24 10:00:00" pay_amount 100.00 real_pay 100.00 pay_time "2021-03-24
10:02:03" user_id 0001 user_name Alice area_id 330106
```

ii. When **schema-syntax** is set to **map**, use the following job script:

```
CREATE TABLE redisSource (
  redisKey string,
  order_result map<string, string>,
  primary key (redisKey) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'hash',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'map'
);

CREATE TABLE printSink (
  redisKey string,
  order_result map<string, string>
) WITH (
  'connector' = 'print'
);

insert into printSink select * from redisSource;
```

iii. The job execution result is as follows:

```
+I(redisSource,{user_id=0001, user_name=Alice, pay_amount=100.00, real_pay=100.00,
order_time=2021-03-24 10:00:00, area_id=330106, order_id=202103241000000001,
order_channel=webShop, pay_time=2021-03-24 10:02:03})
```

1.5.15.2 Result Table

Function

DLI outputs the Flink job output data to Redis. Redis is a key-value storage system that supports multiple types of data structures. It can be used in scenarios such as caching, event publish/subscribe, and high-speed queuing. Redis supports direct read/write of strings, hashes, lists, queues, and sets. Redis works with in-memory datasets and provides persistence. For more information about Redis, visit **https://redis.io/**.

Prerequisites

An enhanced datasource connection with Redis has been established, so that you can configure security group rules as required.

- For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- If the Redis key field is not defined in the statement for creating the Redis result table, the generated UUID is used as the key.
- To specify a key in Redis, you need to define a primary key in the Redis result table of Flink. The value of the primary key is the Redis key.
- If the primary key defined for the Redis result table, it cannot be a composite primary key and only can be one field.
- Constraints on **schema-syntax**:
  - If **schema-syntax** is **map** or **array**, there can be only one non-primary key and it must be of the same **map** or **array** type.
  - If **schema-syntax** is **fields-scores**, the number of non-primary keys must be an even number, and the second key of every two keys except the primary key must be of the **double** type. The **double** value is the score of the previous key. The following is an example:

```
CREATE TABLE redisSink (
  order_id string,
  order_channel string,
  order_time double,
  pay_amount STRING,
  real_pay double,
  pay_time string,
  user_id double,
  user_name string,
  area_id double,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields-scores'
);
```

- Restrictions on **data-type**:
  - If **data-type** is **string**, only one non-primary key field is allowed.

- If **data-type** is **sorted-set** and **schema-syntax** is **fields** or **array**, **default-score** is used as the score.
- If **data-type** is **sorted-set** and **schema-syntax** is **map**, there can be only one non-primary key in addition to the primary key and the non-primary key must be of the **map** type. The **map** values of the non-primary key must be of the **double** type, indicating the score. The keys in the map are the values in the Redis set.
- If **data-type** is **sorted-set** and **schema-syntax** is **array-scores**, only two non-primary keys are allowed and must be of the **array** type.

  The first key indicates values in the Redis set. The second key is of the **array<double>** type, indicating index scores. The following is an example:

```
CREATE TABLE redisSink (
  order_id string,
  arrayField Array<String>,
  arrayScore array<double>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  "default-score" = '3',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'array-scores'
);
```

Syntax

```
create table dwsSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'redis',
  'host' = ''
);
```

Parameters

Table 1-62 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **redis**.
host	Yes	None	String	Redis connector address.
port	No	6379	Integer	Redis connector port.
password	No	None	String	Redis authentication password.

Parameter	Mandatory	Default Value	Data Type	Description
namespace	No	None	String	Redis key namespace. For example, if the value is set to "person" and the key is "jack", the value in the Redis is person:jack.
delimiter	No	:	String	Delimiter between the Redis key and namespace.
data-type	No	hash	String	Redis data type. Available values are as follows: • hash • list • set • sorted-set • string For details about the constraints, see **Constraints on data-type**.
schema-syntax	No	fields	String	Redis schema semantics. Available values are as follows: • **fields**: applicable to all data types. This value indicates that multiple fields can be set and the value of each field is read when data is written. • **fields-scores**: applicable to **sorted-set** data, indicating that each field is read as an independent score. • **array**: applicable to **list**, **set**, and **sorted-set** data. • **array-scores**: applicable to **sorted-set** data. • **map**: applicable to **hash** and **sorted-set** data. For details about the constraints, see **Constraints on schema-syntax**.

Parameter	Mandatory	Default Value	Data Type	Description
deploy-mode	No	standalone	String	Deployment mode of the Redis cluster. The value can be **standalone**, **master-replica**, or **cluster**. The default value is **standalone**. For details about the setting, see the instance type description of the Redis cluster.
retry-count	No	5	Integer	Number of attempts to connect to the Redis cluster.
connection-timeout-millis	No	10000	Integer	Maximum timeout for connecting to the Redis cluster.
commands-timeout-millis	No	2000	Integer	Maximum time for waiting for a completion response.
rebalancing-timeout-millis	No	15000	Integer	Sleep time when the Redis cluster fails.
default-score	No	0	Double	Default score when **data-type** is **sorted-set**.
ignore-retraction	No	false	Boolean	Whether to ignore Retract messages.
skip-null-values	No	true	Boolean	Whether null values will be skipped. If this parameter is **false**, **null** will be assigned for null values.
ignore-retractions	No	false	Boolean	The connector should ignore retraction messages in the update insert/withdraw flow mode.
key-column	No	None	String	Schema key of the Redis table.

Parameter	Mandatory	Default Value	Data Type	Description
sink.delivery-guarantee	No	at-least-once	String	• **exactly-once**: Each record is delivered only once, even in the event of a failover. To create a complete exactly-once pipeline, both the source and the sink must support exactly-once and be properly configured. • **at-least-once**: Records are definitely to be delivered, but may be delivered multiple times. This mode is typically faster than exactly-once. • **none**: Records are delivered on a best-effort basis. This is often the fastest way to process records, but may result in lost or duplicate records.
sink.parallelism	No	None	int	Defines the custom parallelism of the sink. If this parameter is not defined, the planner will derive the parallelism for each statement separately by considering the global configuration.
key-ttl-mode	No	no-ttl	String	Whether the Redis sink TTL function will be enabled. The value can be **no-ttl**, **expire-msec**, **expire-at-date** or **expire-at-timestamp**. • **no-ttl**: No expiration time is set. • **expire-msec**: validity period of the key. The parameter is a long string, in milliseconds. • **expire-at-date**: Date and time when the key expires. The value is in UTC time format. • **expire-at-timestamp**: Timestamp when the key expires.

Parameter	Mandatory	Default Value	Data Type	Description
key-ttl	No	None	String	Supplementary parameter of **key-ttl-mode**. Available values are as follows: ● If **key-ttl-mode** is **no-ttl**, this parameter does not need to be configured. ● If **key-ttl-mode** is **expire-msec**, set this parameter to a string that can be parsed into the Long type. For example, **5000** indicates that the key will expire in 5000 ms. ● If **key-ttl-mode** is **expire-at-date**, set this parameter to a date. For example, **2011-12-03T10:15:30** indicates that the expiration time is 2011-12-03 18:15:30 (UTC+8). ● If **key-ttl-mode** is **expire-at-timestamp**, set this parameter to a timestamp, in milliseconds. For example, **1679385600000** indicates that the expiration time is 2023-03-21 16:00:00.

Example

In this example, data is read from the Kafka data source and written to the Redis result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where Redis locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set Redis security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Redis address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   Change the values of the parameters in bold as needed in the following script.

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
```

```
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);
--In the following redisSink table, data-type is set to default value hash, schema-syntax is fields, and
order_id is defined as the primary key. Therefore, the value of this field is used as the Redis key.
CREATE TABLE redisSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = '<yourRedis>',
  'password' = '<yourPassword>',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields'
);

insert into redisSink select * from orders;
```

4. Connect to the Kafka cluster and insert the following test data into Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}
```

5. Run the following commands in Redis and view the result:
   – Obtain the result whose key is **202103241606060001**.

     Run following command:

     ```
     HGETALL 202103241606060001
     ```

     Command output:

     ```
      1) "user_id"
      2) "0001"
      3) "user_name"
      4) "Alice"
      5) "pay_amount"
      6) "200.0"
      7) "real_pay"
      8) "180.0"
      9) "order_time"
     10) "2021-03-24 16:06:06"
     11) "area_id"
     12) "330106"
     13) "order_channel"
     14) "appShop"
     15) "pay_time"
     16) "2021-03-24 16:10:06"
     ```

   – Obtain the result whose key is **202103241000000001**.

     Run following command:

```
HGETALL 202103241000000001
```

Command output:

```
 1) "user_id"
 2) "0001"
 3) "user_name"
 4) "Alice"
 5) "pay_amount"
 6) "100.0"
 7) "real_pay"
 8) "100.0"
 9) "order_time"
10) "2021-03-24 10:00:00"
11) "area_id"
12) "330106"
13) "order_channel"
14) "webShop"
15) "pay_time"
16) "2021-03-24 10:02:03"
```

FAQ

- Q: When data-type is **set**, why is the final result data less than the input data?

  A: This is because the input data contains duplicate data. Deduplication is performed in the Redis set, and the number of records in the result decreases.

- Q: What should I do if Flink job logs contain the following error information?

  ```
  org.apache.flink.table.api.ValidationException: SQL validation failed. From line 1, column 40 to line 1, column 105: Parameters must be of the same type
  ```

  A: The array type is used. However, the types of fields in the array are different. You need to ensure that the types of fields in the array in Redis are the same.

- Q: What should I do if Flink job logs contain the following error information?

  ```
  org.apache.flink.addons.redis.core.exception.RedisConnectorException: Wrong Redis schema for 'map' syntax: There should be a key (possibly) and 1 MAP non-key column.
  ```

  A: When **schema-syntax** is **map**, the table creation statement in Flink can contain only one non-primary key column, and the column type must be **map**.

- Q: What should I do if Flink job logs contain the following error information?

  ```
  org.apache.flink.addons.redis.core.exception.RedisConnectorException: Wrong Redis schema for 'array' syntax: There should be a key (possibly) and 1 ARRAY non-key column.
  ```

  A: When **schema-syntax** is **array**, the table creation statement in Flink can contain only one non-primary key column, and the column type must be **array**.

- Q: What is the function of **schema-syntax** since **data-type** has been set?

  A: **schema-syntax** is used to process special types, such as **map** and **array**.

  – If it is set to **fields**, the value of each field is processed. If it is set to **array** or **map**, each element in the field is processed. For **fields**, the field value of the **map** or **array** type is directly used as a value in Redis.

  – For **array** or **map**, each value in the array is used as a Redis value, and the field value of the map is used as the Redis value. **array-scores** is used to process the **sorted-set** data type. It indicates that two array fields are used, the first one is the value in the set, and the second one is the score. **fields-scores** is used to process the **sorted-set** data type, indicating that the score is derived from the defined field. The field of an odd number except the primary key indicates the value in the set, and its next field indicates its score. Therefore, its next field must be of the **double** type.

- Q: If **data-type** is **hash**, what are the differences between **schema-syntax** set to **fields** and that to **map**?

  A: When **fields** is used, the field name in Flink is used as the Redis field of the hash data type, and the value of that field is used as the value of the hash data type in Redis. When **map** is used, the field key in Flink is used as the Redis field of the hash data type, and the value of that field is used as the value of the hash data type in Redis. The following is an example:

  – For **fields**:

    i. The execution script of the Flink job is as follows:

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE redisSink (
  order_id string,
  maptest Map<string, String>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields'
);

insert into redisSink select order_id, Map[user_id, area_id] from orders;
```

    ii. Connect to the Kafka cluster and insert the following test data into the Kafka topic:

```
{"order_id":"202103241000000001", "order_channel":"webShop",
"order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00",
"pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice",
"area_id":"330106"}
```

    iii. In the Redis, the result is as follows:

```
1) "maptest"
2) "{0001=330106}"
```

  – For **map**:

    i. The execution script of the Flink job is as follows:

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
```

```
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE redisSink (
  order_id string,
  maptest Map<string, String>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'map'
);

insert into redisSink select order_id, Map[user_id, area_id] from orders;
```

ii. Connect to the Kafka cluster and insert the following test data into the Kafka topic:

```
{"order_id":"202103241000000001", "order_channel":"webShop",
"order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00",
"pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice",
"area_id":"330106"}
```

iii. In the Redis, the result is as follows:

```
1) "0001"
2) "330106"
```

- Q: If **data-type** is **list**, what are the differences between **schema-syntax** set to **fields** and that to **array**?

  A: The setting to **fields** or **array** does not result in different results. The only difference is that in the Flink table creation statement. **fields** can be multiple fields. However, **array** requires that the field is of the **array** type and the data types in the array must be the same. Therefore, **fields** are more flexible.

  – For **fields**:

  i. The execution script of the Flink job is as follows:

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE redisSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
```

```
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'list',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields'
);

insert into redisSink select * from orders;
```

ii. Connect to the Kafka cluster and insert the following test data into the Kafka topic:

```
{"order_id":"202103241000000001", "order_channel":"webShop",
"order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00",
"pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice",
"area_id":"330106"}
```

iii. View the result.

Run the following command in Redis:

```
LRANGE 202103241000000001 0 8
```

The command output is as follows:

```
1) "webShop"
2) "2021-03-24 10:00:00"
3) "100.0"
4) "100.0"
5) "2021-03-24 10:02:03"
6) "0001"
7) "Alice"
8) "330106"
```

– For **array**:

i. The execution script of the Flink job is as follows:

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE redisSink (
  order_id string,
  arraytest Array<String>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'list',
  'deploy-mode' = 'master-replica',
```

```
  'schema-syntax' = 'array'
);

insert into redisSink select order_id,
array[order_channel,order_time,pay_time,user_id,user_name,area_id] from orders;
```

ii. Connect to the Kafka cluster and insert the following test data into the Kafka topic:

```
{"order_id":"202103241000000001", "order_channel":"webShop",
"order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00",
"pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice",
"area_id":"330106"}
```

iii. In Redis, view the result. (The result is different from that of **fields** because data of the **double** type is not added to the table creation statement of the sink in Flink. Therefore, two values are missing. This is not caused by the difference between **fields** and **array**.)

```
1) "webShop"
2) "2021-03-24 10:00:00"
3) "2021-03-24 10:02:03"
4) "0001"
5) "Alice"
6) "330106"
```

1.5.15.3 Dimension Table

Function

Create a Redis table to connect to source streams for wide table generation.

Prerequisites

- An enhanced datasource connection with Redis has been established, so that you can configure security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- To obtain the key values, you can set the primary key in Flink. The primary key maps to the Redis key.
- If the primary key cannot be a composite primary key, and only can be one field.
- Constraints on **schema-syntax**:
  - If **schema-syntax** is **map** or **array**, there can be only one non-primary key and it must be of the same **map** or **array** type.

– If **schema-syntax** is **fields-scores**, the number of non-primary keys must be an even number, and the second key of every two keys except the primary key must be of the **double** type. The **double** value is the score of the previous key. The following is an example:

```
CREATE TABLE redisSource (
  redisKey string,
  order_id string,
  score1 double,
  order_channel string,
  score2 double,
  order_time string,
  score3 double,
  pay_amount double,
  score4 double,
  real_pay double,
  score5 double,
  pay_time string,
  score6 double,
  user_id string,
  score7 double,
  user_name string,
  score8 double,
  area_id string,
  score9 double,
  primary key (redisKey) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields-scores'
);
```

- Restrictions on **data-type**:
  – When **data-type** is **set**, the types of non-primary keys defined in Flink must be the same.
  – If **data-type** is **sorted-set** and **schema-syntax** is **fields** or **array**, only **sorted set** values can be read from Redis, and the **score** value cannot be read.
  – If **data-type** is **string**, only one non-primary key field is allowed.
  – If **data-type** is **sorted-set** and **schema-syntax** is **map**, there can be only one non-primary key in addition to the primary key and the non-primary key must be of the **map** type. The **map** values of the non-primary key must be of the **double** type, indicating the score. The keys in the map are the values in the Redis set.
  – If **data-type** is **sorted-set** and **schema-syntax** is **array-scores**, only two non-primary keys are allowed and must be of the **array** type.

    The first key indicates values in the Redis set. The second key is of the **array<double>** type, indicating index scores. The following is an example:

```
CREATE TABLE redisSink (
  order_id string,
  arrayField Array<String>,
  arrayScore array<double>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  "default-score" = '3',
  'deploy-mode' = 'master-replica',
```

```
  'schema-syntax' = 'array-scores'
);
```

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
  ,PRIMARY KEY (attr_name, ...) NOT ENFORCED
)
with (
  'connector' = 'redis',
  'host' = ''
);
```

Parameters

Table 1-63 Parameter description

Parameter	Mandatory	Default Value	Data Types	Description
connector	Yes	None	String	Connector type. Set this parameter to **redis**.
host	Yes	None	String	Redis connector address
port	No	6379	Integer	Redis connector port
password	No	None	String	Redis authentication password
namespace	No	None	String	Redis key namespace
delimiter	No	:	String	Delimiter between the Redis key and namespace
data-type	No	hash	String	Redis data type. Available values are as follows: ● hash ● list ● set ● sorted-set ● string For details about the constraints, see **Constraints on data-type**.

Parameter	Mandatory	Default Value	Data Types	Description
schema-syntax	No	fields	String	Redis schema semantics. Available values are as follows: • **fields**: applicable to all data types • **fields-scores**: applicable to **sorted set** data • **array**: applicable to **list**, **set**, and **sorted set** data • **array-scores**: applicable to **sorted set** data • **map**: applicable to **hash** and **sorted set** data For details about the constraints, see **Constraints on schema-syntax**.
deploy-mode	No	standalone	String	Deployment mode of the Redis cluster. The value can be **standalone**, **master-replica**, or **cluster**. The default value is **standalone**.
retry-count	Yes	5	Integer	Size of each connection request queue. If the number of connection requests in a queue exceeds the queue size, command calling will cause RedisException. Setting **requestQueueSize** to a small value will cause exceptions to occur earlier during overload or disconnection. A larger value indicates more time required to reach the boundary, but more requests may be queued and more heap space may be used. The default value is **2147483647**.
connection-timeout-millis	No	10000	Integer	Maximum timeout for connecting to the Redis cluster
commands-timeout-millis	No	2000	Integer	Maximum time for waiting for a completion response
rebalancing-timeout-millis	No	15000	Integer	Sleep time when the Redis cluster fails

Parameter	Mandatory	Default Value	Data Types	Description
scan-keys-count	No	1000	Integer	Number of data records read in each scan
default-score	No	0	Double	Default score when **data-type** is **sorted-set**
deserialize-error-policy	No	fail-job	Enum	How to process a data parsing failure Available values are as follows: ● **fail-job**: Fail the job ● **skip-row**: Skip the current data. ● **null-field**: Set the current data to null.
skip-null-values	No	true	Boolean	Whether null values will be skipped
lookup.async	No	false	Boolean	Whether asynchronous I/O will be used when this table is used as a dimension table
lookup.parallelism	No	None	int	Defines the custom parallelism of the lookup join operator. If this parameter is not defined, the planner will derive the parallelism by considering the global configuration (if the **lookup.parallelism** parameter is defined) or the parallelism of the input operator.
lookup.batch.interval	No	1s	Duration	Batch lookup join can buffer input records with a maximum delay. Batch lookup join can buffer input records with a maximum delay.
lookup.batch.size	No	100L	long	Maximum number of input records that can be buffered for batch lookup join.

Parameter	Mandatory	Default Value	Data Types	Description
lookup.batch	No	false	Boolean	Whether to enable batch lookup optimization. If enabled, the user must set both the **lookup.batch.interval** and **lookup.batch.size** parameters. Additionally, due to the implementation of the underlying batch processing interval interference mechanism, the user must explicitly enable the **table.exec.batch-lookup.enabled** parameter in the Flink configuration.
ignore-retractions	No	false	Boolean	The connector should ignore retraction messages in the update insert/withdraw flow mode.
key-column	No	None	String	Schema key of the Redis table.

Example

Read data from a Kafka source table, use a Redis table as the dimension table. Write wide table information generated by the source and dimension tables to a Kafka result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where Redis and Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set Redis and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Redis address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
3. Run the following commands on the Redis client to send data to Redis:
```
HMSET 330102  area_province_name a1 area_province_name b1 area_county_name c1
area_street_name d1 region_name e1

HMSET 330106  area_province_name a1 area_province_name b1 area_county_name c2
area_street_name d2 region_name e1

HMSET 330108  area_province_name a1 area_province_name b1 area_county_name c3
area_street_name d3 region_name e1

HMSET 330110  area_province_name a1 area_province_name b1 area_county_name c4
area_street_name d4 region_name e1
```
4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses Kafka as the data source and a Redis table as the dimension table. Data is output to a Kafka result table.

   Change the values of the parameters in bold as needed in the following script.

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaSourceTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

-- Create an address dimension table
create table area_info (
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string,
    primary key (area_id) not enforced -- Redis key
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'hash',
  'deploy-mode' = 'master-replica'
);

-- Generate a wide table based on the address dimension table containing detailed order information.
create table order_detail(
    order_id string,
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
    pay_time string,
    user_id string,
    user_name string,
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) with (
  'connector' = 'kafka',
  'topic' = 'kafkaSinkTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'json'
);

insert into order_detail
    select orders.order_id, orders.order_channel, orders.order_time, orders.pay_amount, orders.real_pay,
orders.pay_time, orders.user_id, orders.user_name,
           area.area_id, area.area_province_name, area.area_city_name, area.area_county_name,
           area.area_street_name, area.region_name  from orders
    left join area_info for system_time as of orders.proctime as area on orders.area_id = area.area_id;
```

5. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202103251505050001", "order_channel":"appShop", "order_time":"2021-03-25 15:05:05", "pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2021-03-25 15:10:00", "user_id":"0003", "user_name":"Cindy", "area_id":"330108"}

6. Connect to the Kafka cluster and read data from the sink topic of Kafka. The result data is as follows:

{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24 16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24 16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106","area_province_name":"a1","area_city_name":"b1","area_county_name":"c2","area_street_name":"d2","region_name":"e1"}

{"order_id":"202103251202020001","order_channel":"miniAppShop","order_time":"2021-03-25 12:02:02","pay_amount":60.0,"real_pay":60.0,"pay_time":"2021-03-25 12:03:00","user_id":"0002","user_name":"Bob","area_id":"330110","area_province_name":"a1","area_city_name":"b1","area_county_name":"c4","area_street_name":"d4","region_name":"e1"}

{"order_id":"202103251505050001","order_channel":"appshop","order_time":"2021-03-25 15:05:05","pay_amount":500.0,"real_pay":400.0,"pay_time":"2021-03-25 15:10:00","user_id":"0003","user_name":"Cindy","area_id":"330108","area_province_name":"a1","area_city_name":"b1","area_county_name":"c3","area_street_name":"d3","region_name":"e1"}

FAQs

If Chinese characters are written to the Redis in the Windows environment, an exception will occur during data writing.

1.5.16 Upsert Kafka

Function

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. The Upsert Kafka connector allows for reading data from and writing data into Kafka topics in the upsert fashion. Source tables and result tables are supported.

- As a source, the upsert-kafka connector produces a changelog stream, where each data record represents an update or delete event.

  The value in a data record is interpreted as an UPDATE of the last value for the same key, if any (if a corresponding key does not exist yet, the UPDATE will be considered an INSERT). Using the table analogy, a data record in a changelog stream is interpreted as an UPSERT, also known as INSERT/UPDATE, because any existing row with the same key is overwritten. Also, null values are interpreted in a special way: A record with a null value represents a DELETE.
- As a sink, the upsert-kafka connector can consume a changelog stream. It will write INSERT/UPDATE_AFTER data as normal Kafka messages value, and write DELETE data as Kafka messages with null values (indicate tombstone for the key). Flink will guarantee the message ordering on the primary key by partition data on the values of the primary key columns, so the UPDATE/DELETE messages on the same key will fall into the same partition.

Table 1-64 Supported types

Type	Description
Supported Table Types	Source table and result table

Prerequisites

An enhanced datasource connection has been created for DLI to connect to Kafka clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.

- For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections**.
- For details about how to configure security group rules, see **Security Group Overview**.

Caveats

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- Storing authentication credentials such as usernames and passwords in code or plaintext poses significant security risks. It is recommended using DEW to manage credentials instead. Storing encrypted credentials in configuration files or environment variables and decrypting them when needed ensures security. For details, see **Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials**.
- The Upsert Kafka always works in the upsert fashion and requires to define the primary key in the DDL. With the assumption that records with the same key should be ordered in the same partition, the primary key semantic on the changelog source means the materialized changelog is unique on the primary keys. The primary key definition will also control which fields should end up in Kafka's key.
- Because the connector is working in upsert mode, the last record on the same key will take effect when reading back as a source.
- For details about how to use data types, see **Format**.

Syntax

```
create table kafkaTable(
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'upsert-kafka',
  'topic' = '',
  'properties.bootstrap.servers' = '',
  'key.format' = '',
  'value.format' = ''
);
```

Parameter Description

Table 1-65 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. For the Upsert Kafka connector, set this parameter to **upsert-kafka**.
topic	Yes	None	String	Kafka topic name
properties.bootstrap.servers	Yes	None	String	Comma separated list of Kafka brokers
key.format	Yes	None	String	Format used to deserialize and serialize the key part of Kafka messages. The key fields are specified by the **PRIMARY KEY** syntax. The following formats are supported: ● csv ● json ● avro Refer to **Format** for more details and format parameters.
key.fields-prefix	No	None	String	Defines a custom prefix for all fields of the key format to avoid name clashes with fields of the value format. By default, the prefix is empty. If a custom prefix is defined, both the table schema and **key.fields** will work with prefixed names. When constructing the data type of the key format, the prefix will be removed and the non-prefixed names will be used within the key format. Note that this option requires that **value.fields-include** be set to **EXCEPT_KEY**.

Parameter	Mandatory	Default Value	Data Type	Description
value.format	Yes	None	String	Format used to deserialize and serialize the value part of Kafka messages. The following formats are supported: ● csv ● json ● avro Refer to **Format** for more details and format parameters.
value.fields-include	Yes	ALL	String	Controls which fields should appear in the value part. Possible values are: ● **ALL**: All fields in the schema, including the primary key field, are included in the value part. ● **EXCEPT_KEY**: All the fields of the table schema are included, except the primary key field.
properties.*	No	None	String	This option can set and pass arbitrary Kafka configurations. The suffix to **properties.** must match the parameter defined in **Kafka Configuration documentation**. Flink will remove the **properties.** key prefix and pass the transformed key and value to the underlying KafkaClient. For example, you can disable automatic topic creation via **'properties.allow.auto.create.topics' = 'false'**. But there are some configurations that do not support to set, because Flink will override them, for example, **'key.deserializer'** and **'value.deserializer'**.
sink.parallelism	No	None	Integer	Defines the parallelism of the Upsert Kafka sink operator. By default, the parallelism is determined by the framework: using the same parallelism as the upstream join operator.

Parameter	Mandatory	Default Value	Data Type	Description
sink.buffer-flush.max-rows	No	0	Integer	The max size of buffered records before flushing. When the sink receives many updates on the same key, the buffer will retain the last record of the same key. This can help to reduce data shuffling and avoid possible tombstone messages to Kafka topic. Can be set to **0** to disable it. By default, this is disabled. Note both **sink.buffer-flush.max-rows** and **sink.buffer-flush.interval** must be set to be greater than zero to enable sink buffer flushing.
sink.buffer-flush.interval	No	0	Duration	The flush interval mills, over this time, asynchronous threads will flush data. The unit can be millisecond (ms), second (s), minute (min), or hour (h). For example, **'sink.buffer-flush.interval'='10 ms'**. By default, this is disabled. Note both **sink.buffer-flush.max-rows** and **sink.buffer-flush.interval** must be set to be greater than zero to enable sink buffer flushing.

Metadata

For a list of available metadata fields, see **Kafka Connector**.

Example

- **Example 1: This example reads data from a DMS Kafka data source and writes it to the Print result table.**
  a. Create an enhanced datasource connection in the VPC and subnet where Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
  b. Set Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
  c. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

     When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE upsertKafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  PRIMARY KEY (order_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' =  'KafkaAddress1:KafkaPort,KafkAddress2:KafkaPort',
  'key.format' = 'csv',
  'value.format' = 'json'
);

CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  PRIMARY KEY (order_id) NOT ENFORCED
) WITH (
  'connector' = 'print'
);

INSERT INTO printSink SELECT * FROM upsertKafkaSource;
```

d. Insert the following data to the specified topics in Kafka. (Note: Specify the key when inserting data to Kafka.)

```
{"order_id":"202303251202020001", "order_channel":"miniAppShop", "order_time":"2023-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2023-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202303251505050001", "order_channel":"appshop", "order_time":"2023-03-25
15:05:05", "pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2023-03-25 15:10:00",
"user_id":"0003", "user_name":"Cindy", "area_id":"330108"}

{"order_id":"202303251202020001", "order_channel":"miniAppShop", "order_time":"2023-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2023-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330111"}
```

e. View the **out** file of the TaskManager. The data results are as follows:

```
+I(202303251202020001,miniAppShop,2023-03-2512:02:02,60.0,60.0,2023-03-2512:03:00,0002,B
ob,330110)
+I(202303251505050001,appshop,2023-03-25
15:05:05,500.0,400.0,2023-03-2515:10:00,0003,Cindy,330108)
-
U(202303251202020001,miniAppShop,2023-03-2512:02:02,60.0,60.0,2023-03-2512:03:00,0002,B
ob,330110)
+U(202303251202020001,miniAppShop,2023-03-2512:02:02,60.0,60.0,2023-03-2512:03:00,0002,
Bob,330111)
```

- **Example 2: This example retrieves DMS Kafka source topic data from a Kafka source table and writes it to a Kafka sink topic using Upsert Kafka result table.**

a. Create an enhanced datasource connection in the VPC and subnet where Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.

b. Set Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.

c. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE upsertKafkaSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  PRIMARY KEY(order_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' =  'KafkaAddress1:KafkaPort,KafkAddress2:KafkaPort',
  'key.format' = 'csv',
  'value.format' = 'json'
);

insert into upsertKafkaSink select * from orders;
```

d. Connect to the Kafka cluster and send the following test data to the Kafka source topic:

```
{"order_id":"202303251202020001", "order_channel":"miniAppShop", "order_time":"2023-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2023-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202303251505050001", "order_channel":"appshop", "order_time":"2023-03-25 15:05:05", "pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2023-03-25 15:10:00", "user_id":"0003", "user_name":"Cindy", "area_id":"330108"}

{"order_id":"202303251202020001", "order_channel":"miniAppShop", "order_time":"2023-03-25
```

```
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2023-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330111"}
```

e. Connect to the Kafka cluster and read data from the Kafka sink topic. The result is as follows:

```
{"order_id":"202303251202020001", "order_channel":"miniAppShop", "order_time":"2023-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2023-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202303251505050001", "order_channel":"appshop", "order_time":"2023-03-25
15:05:05", "pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2023-03-25 15:10:00",
"user_id":"0003", "user_name":"Cindy", "area_id":"330108"}

{"order_id":"202303251202020001", "order_channel":"miniAppShop", "order_time":"2023-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2023-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330111"}
```

- **Example 3: In this scenario, the MRS cluster has enabled Kerberos authentication and Kafka is using the SASL_PLAINTEXT protocol. Data is retrieved from a Kafka source table and written to the Print result table.**

  a. Create an enhanced datasource connection in the VPC and subnet where the MRS cluster locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.

  b. Set MRS cluster security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.

  c. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

  When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE upsertKafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  PRIMARY KEY(order_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' =  'KafkaAddress1:KafkaPort,KafkAddress2:KafkaPort',
  'key.format' = 'csv',
  'value.format' = 'json',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.kerberos.service.name' = 'kafka', -- Configured in MRS
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'username', --Username
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf', --krb5_conf path
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab' --keytab path
);

CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
```

```
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  PRIMARY KEY (order_id) NOT ENFORCED
) WITH (
  'connector' = 'print'
);

INSERT INTO printSink SELECT * FROM upsertKafkaSource;
```

d. Insert the following data to the specified topics in Kafka. (Note: Specify the key when inserting data to Kafka.)

```
{"order_id":"202303251202020001", "order_channel":"miniAppShop", "order_time":"2023-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2023-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202303251505050001", "order_channel":"appshop", "order_time":"2023-03-25 15:05:05", "pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2023-03-25 15:10:00", "user_id":"0003", "user_name":"Cindy", "area_id":"330108"}

{"order_id":"202303251202020001", "order_channel":"miniAppShop", "order_time":"2023-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2023-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330111"}
```

e. View the **out** file of the TaskManager. The data results are as follows:

```
+I(202303251202020001,miniAppShop,2023-03-2512:02:02,60.0,60.0,2023-03-2512:03:00,0002,Bob,330110)
+I(202303251505050001,appshop,2023-03-2515:05:05,500.0,400.0,2023-03-2515:10:00,0003,Cindy,330108)
-U(202303251202020001,miniAppShop,2023-03-2512:02:02,60.0,60.0,2023-03-2512:03:00,0002,Bob,330110)
+U(202303251202020001,miniAppShop,2023-03-2512:02:02,60.0,60.0,2023-03-2512:03:00,0002,Bob,330111)
```

FAQ

None

1.6 DML Snytax

1.6.1 SELECT

SELECT

Syntax

```
SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
```

Description

SELECT is used to select data from a table.

ALL indicates that all results are returned.

DISTINCT indicates that the duplicated results are removed.

Precautions

- The to-be-queried table must exist. Otherwise, an error is reported.
- WHERE is used to specify the search 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

Syntax

```
SELECT   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
```

Description

This clause 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

Search 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

Function

This clause is used to search for the query results that meet the search condition.

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. You can use GROUP BY for grouping and then use HAVING for filtering. Arithmetic operations and aggregate functions are supported in the HAVING clause.

Precautions

If the filtering condition is subject to the results of GROUP BY, the HAVING clause, rather than the WHERE clause, must be used for search.

Example

Group the **student** table according to the **name** field and search for the records in which the maximum score is higher than 95 in the group.

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

Column-Based GROUP BY

Function

This clause 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

GroupBy generates update results in the stream processing table.

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 clause is used to group streams 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);
```

Grouping sets, Rollup, Cube

Function

- The GROUP BY GROUPING SETS generates a result set equivalent to that generated by multiple simple GROUP BY UNION ALL statements. Using GROUPING SETS is more efficient.
- The ROLLUP and CUBE generate multiple groups based on certain rules and then collect statistics by group.
- The result set generated by CUBE contains all the combinations of values in the selected columns.
- The result set generated by ROLLUP contains the combinations of a certain layer structure in the selected columns.

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY groupingItem]
```

Description

Values of **groupingItem** can be **Grouping sets(columnName [, columnName]*)**, **Rollup(columnName [, columnName]*)**, and **Cube(columnName [, columnName]*)**.

Precautions

None

Example

Return the results generated based on **user** and **product**.

```
INSERT INTO temp SELECT SUM(amount)
FROM Orders
GROUP BY GROUPING SETS ((user), (product));
```

GROUP BY Using HAVING

Function

This clause 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. You can use GROUP BY for grouping and the HAVING for filtering.

Precautions

- If the filtering condition is subject to the results of GROUP BY, the HAVING clause, rather than the WHERE clause, must be used for search. HAVING and GROUP BY are used together. Use GROUP BY for grouping and the HAVING 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** by **num**, use the HAVING clause to search for 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;
```

1.6.2 INSERT INTO

This section describes how to use the **INSERT INTO** statement to write job results to a sink table.

Writing Data to a Sink Table

- **Syntax**
  ```
  INSERT INTO your_sink
  SELECT ... FROM your_source WHERE ...
  ```
- **Example**

  In this example, two tables **my_source** and **my_sink** are defined, and the **INSERT INTO** statement is used to select data from the source table and insert the data to the sink table.
  ```
  -- Use the datagen connector to create the source table my_source.
  CREATE TABLE my_source (
    name VARCHAR,
    age BIGINT
  ) WITH (
    'connector' = 'datagen');

  -- Use the JDBC connector to create the sink table my_sink.
  CREATE TABLE my_sink (
    name VARCHAR,
    age BIGINT
  ) WITH (
    'connector' = 'jdbc',
    'url' = 'jdbc:mysql://xxx/your-database',
  ```

```
  'table-name' = 'your-table',
  'username' = 'your-username',
  'password' = 'your-password'
);

-- Run the INSERT INTO statement to select data from the my_source table and insert the data into
the my_sink table.
INSERT INTO my_sink
SELECT name, age
FROM my_source;
```

Writing Data to Multiple Sink Tables

EXECUTE STATEMENT SET BEGIN... END; is a required statement for writing data to multiple sink tables. It is used to define multiple data insertion operations in the same job.

> **CAUTION**
>
> **EXECUTE STATEMENT SET BEGIN... END;** is required when data is written to multiple sink tables.

- **Syntax**

```
EXECUTE STATEMENT SET BEGIN
-- First DML statement
  INSERT INTO your_sink1
  SELECT ... FROM your_source WHERE ...;

-- Second DML statement
  INSERT INTO your_sink2
  SELECT ... FROM your_source WHERE ...

...
END;
```

- **Example**

In this example, the source table **datagen_source** and sink tables **print_sinkA** and **print_sinkB** are defined. **EXECUTE STATEMENT** is used to execute two **INSERT INTO** statements to write the converted data to two different sinks.

```
-- Use the datagen connector to create the source table datagen_source.
CREATE TABLE datagen_source (
  name VARCHAR,
  age BIGINT
) WITH (
  'connector' = 'datagen'
);

-- Use the print connector to create the result tables print_sinkA and print_sinkB.

CREATE TABLE print_sinkA(
  name VARCHAR,
  age BIGINT
) WITH (
  'connector' = 'print'
);

CREATE TABLE print_sinkB(
  name VARCHAR,
  age BIGINT
) WITH (
  'connector' = 'print'
);
```



```
-- Use EXECUTE STATEMENT SET BEGIN to execute two INSERT INTO statements.
-- The first INSERT INTO statement converts the data in the datagen_source table as needed and writes the converted data to print_sinkA.
-- The second INSERT INTO statement converts data as needed and writes the converted data to print_sinkB.
EXECUTE  STATEMENT SET BEGIN
INSERT INTO print_sinkA
  SELECT UPPER(name), min(age)
  FROM datagen_source
  GROUP BY UPPER(name);
INSERT INTO print_sinkB
  SELECT LOWER(name), max(age)
  FROM datagen_source
  GROUP BY LOWER(name);
END;
```

1.6.3 Set Operations

Union/Union ALL/Intersect/Except

Syntax

```
query UNION [ ALL ] | Intersect | Except query
```

Description

- UNION is used to return the union set of multiple query results.
- INTERSECT is used to return the intersection of multiple query results.
- EXCEPT is used to return the difference set of multiple query results.

Precautions

- Set operations 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, UNION takes only distinct records while UNION ALL does not remove duplicates from the result.

Example

Output distinct records found in either Orders1 and Orders2 tables.

```
insert into temp SELECT  * FROM Orders1
  UNION SELECT  * FROM Orders2;
```

IN

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  WHERE column_name IN (value (, value)* ) | query
```

Description

The IN operator allows multiple values to be specified in the WHERE clause. It returns true if the expression exists in the given table subquery.

Precautions

The subquery table must consist of a single column, and the data type of the column must be the same as that of the expression.

Example

Return **user** and **amount** information of the products in **NewProducts** of the **Orders** table.

```
insert into temp SELECT user, amount
FROM Orders
WHERE product IN (
   SELECT product FROM NewProducts
);
```

1.6.4 Window

GROUP WINDOW

Description

Group Window is defined in GROUP BY. One record is generated from each group. Group Window involves the following functions:

- Grouping functions

CAUTION

In streaming mode, the **time_attr** argument of the group window function must refer to a valid time attribute that specifies the processing time or event time of rows.

- **event-time**: The type is timestamp(3).
- **processing-time**: No need to specify the type.

In batch mode, the **time_attr** argument of the group window function must be an attribute of type timestamp.

Table 1-66 Grouping functions

Grouping Window Function	Description
TUMBLE(time_attr, interval)	Defines a tumbling time window. A tumbling time window assigns rows to non-overlapping, continuous windows with a fixed duration (interval). For example, a tumbling window of 5 minutes groups rows in 5 minutes intervals. Tumbling windows can be defined on event-time (stream + batch) or processing-time (stream).

Grouping Window Function	Description
HOP(time_attr, interval, interval)	Defines a hopping time window (called sliding window in the Table API). A hopping time window has a fixed duration (second interval parameter) and hops by a specified hop interval (first interval parameter). If the hop interval is smaller than the window size, hopping windows are overlapping. Thus, rows can be assigned to multiple windows. For example, a hopping window of 15 minutes size and 5 minute hop interval assigns each row to 3 different windows of 15 minute size, which are evaluated in an interval of 5 minutes. Hopping windows can be defined on event-time (stream + batch) or processing-time (stream).
SESSION(time_attr, interval)	Defines a session time window. Session time windows do not have a fixed duration but their bounds are defined by a time interval of inactivity, i.e., a session window is closed if no event appears for a defined gap period. For example a session window with a 30 minute gap starts when a row is observed after 30 minutes inactivity (otherwise the row would be added to an existing window) and is closed if no row is added within 30 minutes. Session windows can work on event-time (stream + batch) or processing-time (stream).

- Window helper functions

  You can use the following helper functions to select the start and end timestamps, as well as the time attribute, for grouping windows.

  > ⚠ CAUTION
  >
  > When calling helper functions, it is important to use the same parameters as those used in the **GROUP BY** clause for grouping window functions.

Table 1-67 Window helper functions

Helper Function	Description
TUMBLE_START(time_attr, interval) HOP_START(time_attr, interval, interval) SESSION_START(time_attr, interval)	Returns the timestamp of the inclusive lower bound of the corresponding tumbling, hopping, or session window.
TUMBLE_END(time_attr, interval) HOP_END(time_attr, interval, interval) SESSION_END(time_attr, interval)	Returns the timestamp of the **exclusive** upper bound of the corresponding tumbling, hopping, or session window. The exclusive upper bound timestamp **cannot** be used as a rowtime attribute in subsequent time-based operations, such as interval joins and group window or over window aggregations.
TUMBLE_ROWTIME(time_attr, interval) HOP_ROWTIME(time_attr, interval, interval) SESSION_ROWTIME(time_attr, interval)	Returns the timestamp of the inclusive upper bound of the corresponding tumbling, hopping, or session window. The resulting attribute is a rowtime attribute that can be used in subsequent time-based operations such as interval joins and group window or over window aggregations.
TUMBLE_PROCTIME(time_attr, interval) HOP_PROCTIME(time_attr, interval, interval) SESSION_PROCTIME(time_attr, interval)	Returns a processing time attribute that can be used in subsequent time-based operations such as interval joins and group window or over window aggregations.

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;
```

TUMBLE WINDOW Extension

Function

The extension functions of the DLI tumbling window are as follows:

- A tumbling window is triggered periodically to reduce latency.

  Before the tumbling window ends, the window can be periodically triggered based on the configured frequency. The compute result from the start to the current time is output, which does not affect the final output. The latest result can be viewed in each period before the window ends.

- Custom latency for higher data accuracy

  You can set a latency for the end of the window. The output of the window is updated according to the configured latency each time a piece of late data reaches.

Caveats

- If you use the INSERT statement to write results to a sink, it must support the upsert mode. Ensure that the result table supports upsert operations and the primary key is defined.
- Latency settings only take effect for event time and not for processing time.
- Helper functions must be called with the same parameters as the grouping window functions in the GROUP BY clause.
- If event time is used, watermark must be used. The code is as follows (**order_time** is identified as the event time column and watermark is set to 3 seconds):

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  watermark for order_time as order_time - INTERVAL '3' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);
```

- If the processing time is used, you need to use the computed column. The code is as follows (**proc** is the processing time column):

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  proc as proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);
```

Syntax

```
TUMBLE(time_attr, window_interval, period_interval, lateness_interval)
```

Example

The current time attribute column is **testtime**, the window interval is 10 seconds, and the latency is 10 seconds.

```
TUMBLE(testtime, INTERVAL '10' SECOND, INTERVAL '10' SECOND, INTERVAL '10' SECOND)
```

Parameter description

Table 1-68 Parameters

<table>
<tr><th>Parameter</th><th>Description</th><th>Format</th></tr>
<tr><td>time_attr</td><td>Event time or processing time attribute column
● event-time: The type is timestamp(3).
● processing-time: No need to specify the type.</td><td>-</td></tr>
<tr><td>window_interval</td><td>Duration of the window</td><td rowspan="2">● Format 1: INTERVAL '10' SECOND
The window interval is 10 seconds. You can change the value as needed.
● Format 2: INTERVAL '10' MINUTE
The window interval is 10 minutes. You can change the value as needed.
● Format 3: INTERVAL '10' DAY
The window interval is 10 days. You can change the value as needed.</td></tr>
<tr><td>period_interval</td><td>Frequency of periodic triggering within the window range. That is, before the window ends, the output result is updated at an interval specified by period_interval from the time when the window starts. If this parameter is not set, the periodic triggering policy is not used by default.</td></tr>
</table>

Parameter	Description	Format
lateness_interval	Time to postpone the end of the window. The system continues to collect the data that reaches the window within **lateness_interval** after the window ends. The output is updated for each data that reaches the window within **lateness_interval**. **NOTE** If the time window is for processing time, **lateness_interval** does not take effect.	

NOTE

Values of **period_interval** and **lateness_interval** cannot be negative numbers.

- If **period_interval** is set to **0**, periodic triggering is disabled for the window.
- If **lateness_interval** is set to **0**, the latency after the window ends is disabled.
- If neither of the two parameters is set, both periodic triggering and latency are disabled and only the regular tumbling window functions are available .
- If only the latency function needs to be used, set period_interval **INTERVAL '0' SECOND**.

Helper functions

Table 1-69 Helper functions

Helper Function	Description
TUMBLE_START(time_attr, window_interval, period_interval, lateness_interval)	Returns the timestamp of the inclusive lower bound of the corresponding tumbling window.
TUMBLE_END(time_attr, window_interval, period_interval, lateness_interval)	Returns the timestamp of the exclusive upper bound of the corresponding tumbling window.

Example

1. The Kafka is used as the data source table containing the order information, and the JDBC is used as the data result table for statistics on the number of orders settled by a user within 30 seconds. The order ID and window opening time are used as primary keys to collect result statistics in real time to JDBC.

Step 1 Create a datasource connection for the communication with the VPC and subnet where MySQL and Kafka locate and bind the connection to the queue. Set an inbound rule for the security group to allow access of the queue, and test the connectivity of the queue using the MySQL and Kafka addresses. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Run the following statement to create the **order_count** table in the MySQL Flink database:

```
CREATE TABLE `flink`.`order_count` (
    `user_id` VARCHAR(32) NOT NULL,
    `window_start` TIMESTAMP NOT NULL,
    `window_end` TIMESTAMP NULL,
    `total_num` BIGINT UNSIGNED NULL,
    PRIMARY KEY (`user_id`, `window_start`)
)    ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4
    COLLATE = utf8mb4_general_ci;
```

Step 3 Create a Flink OpenSource SQL job and submit the job. In this example, the window size is 30 seconds, the triggering period is 10 seconds, and the latency is 5 seconds. That is, if the result is updated before the window ends, the intermediate result will be output every 10 seconds. After the watermark is reached and the window ends, the data whose event time is within 5 seconds of the watermark will still be processed and counted in the current window. If the event time exceeds 5 seconds of the watermark, the data will be discarded.

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  watermark for order_time as order_time - INTERVAL '3' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE jdbcSink (
  user_id string,
  window_start timestamp(3),
  window_end timestamp(3),
  total_num BIGINT,
  primary key (user_id, window_start) not enforced
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://<yourMySQL>:3306/flink',
  'table-name' = 'order_count',
  'username' = '<yourUserName>',
  'password' = '<yourPassword>',
  'sink.buffer-flush.max-rows' = '1'
);

insert into jdbcSink select
    order_id,
    TUMBLE_START(order_time, INTERVAL '30' SECOND, INTERVAL '10' SECOND, INTERVAL '5' SECOND),
    TUMBLE_END(order_time, INTERVAL '30' SECOND, INTERVAL '10' SECOND, INTERVAL '5' SECOND),
    COUNT(*) from orders
    GROUP BY user_id, TUMBLE(order_time, INTERVAL '30' SECOND, INTERVAL '10' SECOND, INTERVAL '5'
SECOND);
```

Step 4 Insert data to Kafka. Assume that orders are settled at different time and the order data at 10:00:13 arrives late.

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
```

```
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241000000002", "order_channel":"webShop", "order_time":"2021-03-24 10:00:20",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241000000003", "order_channel":"webShop", "order_time":"2021-03-24 10:00:33",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241000000004", "order_channel":"webShop", "order_time":"2021-03-24 10:00:13",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}
```

Step 5 Run the following statement in the MySQL database to view the output result. The final result is displayed as follows because the periodic output result cannot be collected:

```
select * from order_count
user_id      window_start         window_end        total_num
0001      2021-03-24 10:00:00  2021-03-24 10:00:30    3
0001      2021-03-24 10:00:30  2021-03-24 10:01:00    1
```

----End

OVER WINDOW

The difference between Over Window and Group Window is that one record is generated from one row in Over Window.

Syntax

```
SELECT agg1(attr1) OVER (
  [PARTITION BY partition_name]
  ORDER BY proctime|rowtime
  ROWS
 BETWEEN (UNBOUNDED|rowCOUNT) PRECEDING AND CURRENT ROW FROM TABLENAME

SELECT agg1(attr1) OVER (
  [PARTITION BY partition_name]
  ORDER BY proctime|rowtime
  RANGE
  BETWEEN (UNBOUNDED|timeInterval) PRECEDING AND CURRENT ROW FROM TABLENAME
```

Description

Table 1-70 Parameters

Parameter	Description
PARTITION BY	Primary key of the specified group. Each group separately performs calculation.
ORDER BY	Processing time or event time as the timestamp for data.
ROWS	Count window.
RANGE	Time window.

Caveats

- All aggregates must be defined in the same window, that is, in the same partition, sort, and range.
- Currently, only windows from PRECEDING (unbounded or bounded) to CURRENT ROW are supported. The range described by FOLLOWING is not supported.
- ORDER BY must be specified for a single time attribute.

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;
```

1.6.4.1 Window Functions

Windowing Table-Valued Functions (Windowing TVFs)

Windows are at the heart of processing infinite streams. Windows split the stream into "buckets" of finite size, over which we can apply computations.

Apache Flink provides several **window table-valued functions (TVF)** to divide the elements of your table into windows, including:

- Tumble Windows
- Hop Windows
- Cumulate Windows

Note that each element can logically belong to more than one window, depending on the windowing table-valued function you use. For example, HOP windowing creates overlapping windows wherein a single element can be assigned to multiple windows.

Windowing TVFs are Flink defined Polymorphic Table Functions (abbreviated PTF). PTF is part of the SQL 2016 standard, a special table-function, but can have a table as a parameter.

Windowing TVFs is a replacement of legacy Grouped Window Functions. Windowing TVFs is more SQL standard compliant and more powerful to support complex window-based computations, e.g. Window TopN, Window Join. However, Grouped Window Functions can only support Window Aggregation.

For more information, see **Window Functions**.

Window Functions

Apache Flink provides 3 built-in windowing TVFs: **TUMBLE**, **HOP** and **CUMULATE**.

The return value of windowing TVF is a new relation that includes all columns of original relation as well as additional 3 columns named "window_start", "window_end", "window_time" to indicate the assigned window.

In batch mode, the "window_time" field is an attribute of type **TIMESTAMP** or **TIMESTAMP_LTZ** based on input time field type. The "window_time" field can be used in subsequent time-based operations, e.g. another windowing TVF, or interval joins, over aggregations. The value of window_time always equal to window_end – 1 ms.

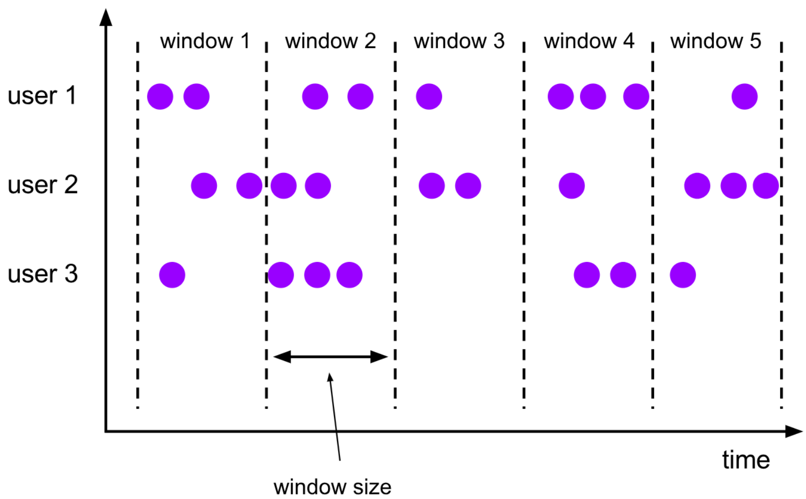
TUMBLE

- **Function**

  The **TUMBLE** function assigns each element to a window of specified window size. Tumbling windows have a fixed size and do not overlap.

  For example, suppose you specify a tumbling window with a size of 5 minutes. In that case, Flink will evaluate the current window, and a new window started every five minutes.

  Figure 1-3 Tumbling window



- **Description**

  The **TUMBLE** function assigns a window for each row of a relation based on a time attribute field. In streaming mode, the time attribute field must be either event or processing time attributes. In batch mode, the time attribute field of window table function must be an attribute of type **TIMESTAMP** or **TIMESTAMP_LTZ**.

  The return value of **TUMBLE** is a new relation that includes all columns of original relation as well as additional 3 columns named "window_start", "window_end", "window_time" to indicate the assigned window. The original time attribute "timecol" will be a regular timestamp column after window TVF.

```
TUMBLE(TABLE data, DESCRIPTOR(timecol), size [, offset ])
```

Table 1-71 TUMBLE function parameters

Parameter	Mandatory	Description
data	Yes	A table parameter that can be any relation with a time attribute column.
timecol	Yes	A column descriptor indicating which time attributes column of data should be mapped to tumbling windows.
size	Yes	A duration specifying the width of the tumbling windows.
offset	No	Offset which window start would be shifted by.

- **Example**

```
-- tables must have time attribute, e.g. `bidtime` in this table
Flink SQL> desc Bid;
+-------------+------------------------+------+-----+--------+---------------------------------+
|        name |                   type | null | key | extras |                       watermark |
+-------------+------------------------+------+-----+--------+---------------------------------+
|     bidtime | TIMESTAMP(3) *ROWTIME* | true |     |        | `bidtime` - INTERVAL '1' SECOND |
|       price |         DECIMAL(10, 2) | true |     |        |                                 |
|        item |                 STRING | true |     |        |                                 |
+-------------+------------------------+------+-----+--------+---------------------------------+

Flink SQL> SELECT * FROM Bid;
+------------------+-------+------+
|          bidtime | price | item |
+------------------+-------+------+
| 2020-04-15 08:05 |  4.00 | C    |
| 2020-04-15 08:07 |  2.00 | A    |
| 2020-04-15 08:09 |  5.00 | D    |
| 2020-04-15 08:11 |  3.00 | B    |
| 2020-04-15 08:13 |  1.00 | E    |
| 2020-04-15 08:17 |  6.00 | F    |
+------------------+-------+------+

Flink SQL> SELECT * FROM TABLE(
   TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES));
-- or with the named params
-- note: the DATA param must be the first
Flink SQL> SELECT * FROM TABLE(
   TUMBLE(
     DATA => TABLE Bid,
     TIMECOL => DESCRIPTOR(bidtime),
     SIZE => INTERVAL '10' MINUTES));
+------------------+-------+------+------------------+------------------+-------------------------+
|          bidtime | price | item |     window_start |       window_end |            window_time  |
+------------------+-------+------+------------------+------------------+-------------------------+
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 |
| 2020-04-15 08:07 |  2.00 | A    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 |
| 2020-04-15 08:09 |  5.00 | D    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 |
| 2020-04-15 08:13 |  1.00 | E    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 |
| 2020-04-15 08:17 |  6.00 | F    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 |
+------------------+-------+------+------------------+------------------+-------------------------+

-- apply aggregation on the tumbling windowed table
Flink SQL> SELECT window_start, window_end, SUM(price)
```

```
 FROM TABLE(
   TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))
 GROUP BY window_start, window_end;
+------------------+------------------+-------+
|     window_start |       window_end | price |
+------------------+------------------+-------+
| 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 |
+------------------+------------------+-------+
```

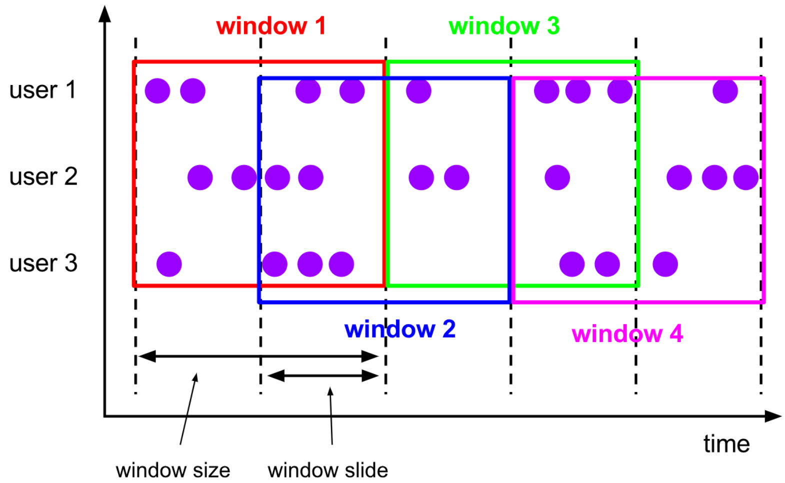
HOP

- **Function**

  The **HOP** function assigns elements to windows of fixed length. Like a **TUMBLE** windowing function, the size of the windows is configured by the window size parameter. An additional window slide parameter controls how frequently a hopping window is started. Hence, hopping windows can be overlapping if the slide is smaller than the window size. In this case, elements are assigned to multiple windows.

  For example, you could have windows of size 10 minutes that slides by 5 minutes. With this, you get every 5 minutes a window that contains the events that arrived during the last 10 minutes, as depicted by the following figure.

  Figure 1-4 Hopping window



- **Description**

  The **HOP** function assigns windows that cover rows within the interval of size and shifting every slide based on a time attribute field. In streaming mode, the time attribute field must be either event or processing time attributes. In batch mode, the time attribute field of window table function must be an attribute of type **TIMESTAMP** or **TIMESTAMP_LTZ**.

  The return value of **HOP** is a new relation that includes all columns of original relation as well as additional 3 columns named "window_start", "window_end", "window_time" to indicate the assigned window. The original time attribute "timecol" will be a regular timestamp column after window TVF.

```
HOP(TABLE data, DESCRIPTOR(timecol), slide, size [, offset ])
```

Table 1-72 HOP function parameters

Parameter	Mandatory	Description
data	Yes	A table parameter that can be any relation with a time attribute column.
timecol	Yes	A column descriptor indicating which time attributes column of data should be mapped to tumbling windows.
slide	Yes	A duration specifying the duration between the start of sequential hopping windows.
size	Yes	A duration specifying the width of the hopping windows.
offset	No	Offset which window start would be shifted by.

- **Example**

```
> SELECT * FROM TABLE(
    HOP(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES));
-- or with the named params
-- note: the DATA param must be the first
> SELECT * FROM TABLE(
    HOP(
      DATA => TABLE Bid,
      TIMECOL => DESCRIPTOR(bidtime),
      SLIDE => INTERVAL '5' MINUTES,
      SIZE => INTERVAL '10' MINUTES));
+------------------+-------+------+------------------+------------------+-------------------------+
|          bidtime | price | item |     window_start |       window_end |           window_time   |
+------------------+-------+------+------------------+------------------+-------------------------+
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 |
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 |
| 2020-04-15 08:07 |  2.00 | A    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 |
| 2020-04-15 08:07 |  2.00 | A    | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 |
| 2020-04-15 08:09 |  5.00 | D    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 |
| 2020-04-15 08:09 |  5.00 | D    | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 |
| 2020-04-15 08:13 |  1.00 | E    | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 |
| 2020-04-15 08:13 |  1.00 | E    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 |
| 2020-04-15 08:17 |  6.00 | F    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 |
| 2020-04-15 08:17 |  6.00 | F    | 2020-04-15 08:15 | 2020-04-15 08:25 | 2020-04-15 08:24:59.999 |
+------------------+-------+------+------------------+------------------+-------------------------+

-- apply aggregation on the hopping windowed table
> SELECT window_start, window_end, SUM(price)
  FROM TABLE(
    HOP(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES))
  GROUP BY window_start, window_end;
+------------------+------------------+-------+
|     window_start |       window_end | price |
+------------------+------------------+-------+
| 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 |
| 2020-04-15 08:05 | 2020-04-15 08:15 | 15.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 |
| 2020-04-15 08:15 | 2020-04-15 08:25 |  6.00 |
+------------------+------------------+-------+
```

CUMULATE

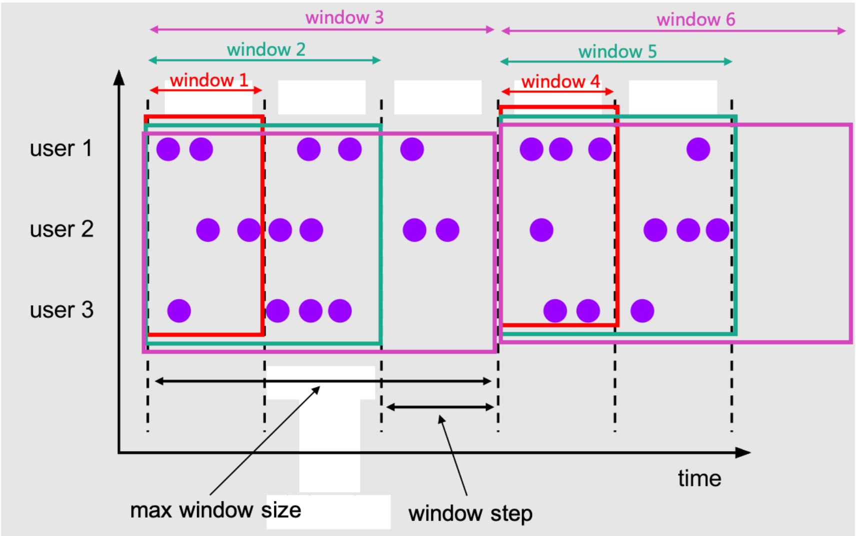
- Function

  Cumulating windows are very useful in some scenarios, such as tumbling windows with early firing in a fixed window interval. For example, a daily dashboard draws cumulative UVs from 00:00 to every minute, the UV at 10:00 represents the total number of UV from 00:00 to 10:00. This can be easily and efficiently implemented by CUMULATE windowing.

  The **CUMULATE** function assigns elements to windows that cover rows within an initial interval of step size and expand to one more step size (keep window start fixed) every step until the max window size. You can think **CUMULATE** function as applying **TUMBLE** windowing with max window size first, and split each tumbling windows into several windows with same window start and window ends of step-size difference. So cumulating windows do overlap and do not have a fixed size.

  For example, you could have a cumulating window for 1 hour step and 1 day max size, and you will get windows: [00:00, 01:00), [00:00, 02:00), [00:00, 03:00), …, [00:00, 24:00) for every day.

  Figure 1-5 Cumulating window

- **Description**

  The **CUMULATE** functions assigns windows based on a time attribute column. In streaming mode, the time attribute field must be either event or processing time attributes. In batch mode, the time attribute field of window table function must be an attribute of type **TIMESTAMP** or **TIMESTAMP_LTZ**.

  The return value of **CUMULATE** is a new relation that includes all columns of original relation as well as additional 3 columns named "window_start", "window_end", "window_time" to indicate the assigned window. The original time attribute "timecol" will be a regular timestamp column after window TVF.

```
CUMULATE(TABLE data, DESCRIPTOR(timecol), step, size)
```


Table 1-73 CUMULATE function parameters

Parameter	Mandatory	Description
data	Yes	A table parameter that can be any relation with a time attribute column.
timecol	Yes	A column descriptor indicating which time attributes column of data should be mapped to cumulating windows.
step	Yes	A duration specifying the increased window size between the end of sequential cumulating windows.
size	Yes	A duration specifying the width of the cumulating windows.
offset	No	Offset which window start would be shifted by.

- **Example**

```
> SELECT * FROM TABLE(
    CUMULATE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '2' MINUTES, INTERVAL '10' MINUTES));
-- or with the named params
-- note: the DATA param must be the first
> SELECT * FROM TABLE(
    CUMULATE(
      DATA => TABLE Bid,
      TIMECOL => DESCRIPTOR(bidtime),
      STEP => INTERVAL '2' MINUTES,
      SIZE => INTERVAL '10' MINUTES));
+------------------+-------+------+------------------+------------------+-------------------------+
|          bidtime | price | item |     window_start |       window_end |             window_time  |
+------------------+-------+------+------------------+------------------+-------------------------+
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:00 | 2020-04-15 08:06 | 2020-04-15 08:05:59.999 |
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:00 | 2020-04-15 08:08 | 2020-04-15 08:07:59.999 |
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 |
| 2020-04-15 08:07 |  2.00 | A    | 2020-04-15 08:00 | 2020-04-15 08:08 | 2020-04-15 08:07:59.999 |
| 2020-04-15 08:07 |  2.00 | A    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 |
| 2020-04-15 08:09 |  5.00 | D    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:10 | 2020-04-15 08:12 | 2020-04-15 08:11:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:10 | 2020-04-15 08:14 | 2020-04-15 08:13:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:10 | 2020-04-15 08:16 | 2020-04-15 08:15:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:10 | 2020-04-15 08:18 | 2020-04-15 08:17:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 |
| 2020-04-15 08:13 |  1.00 | E    | 2020-04-15 08:10 | 2020-04-15 08:14 | 2020-04-15 08:13:59.999 |
| 2020-04-15 08:13 |  1.00 | E    | 2020-04-15 08:10 | 2020-04-15 08:16 | 2020-04-15 08:15:59.999 |
| 2020-04-15 08:13 |  1.00 | E    | 2020-04-15 08:10 | 2020-04-15 08:18 | 2020-04-15 08:17:59.999 |
| 2020-04-15 08:13 |  1.00 | E    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 |
| 2020-04-15 08:17 |  6.00 | F    | 2020-04-15 08:10 | 2020-04-15 08:18 | 2020-04-15 08:17:59.999 |
| 2020-04-15 08:17 |  6.00 | F    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 |
+------------------+-------+------+------------------+------------------+-------------------------+

-- apply aggregation on the cumulating windowed table
> SELECT window_start, window_end, SUM(price)
  FROM TABLE(
    CUMULATE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '2' MINUTES, INTERVAL '10' MINUTES))
  GROUP BY window_start, window_end;
+------------------+------------------+-------+
|     window_start |       window_end | price |
+------------------+------------------+-------+
| 2020-04-15 08:00 | 2020-04-15 08:06 |  4.00 |
| 2020-04-15 08:00 | 2020-04-15 08:08 |  6.00 |
```



```
| 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 |
| 2020-04-15 08:10 | 2020-04-15 08:12 |  3.00 |
| 2020-04-15 08:10 | 2020-04-15 08:14 |  4.00 |
| 2020-04-15 08:10 | 2020-04-15 08:16 |  4.00 |
| 2020-04-15 08:10 | 2020-04-15 08:18 | 10.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 |
+------------------+------------------+-------+
```

Window Offset

Offset is an optional parameter which could be used to change the window assignment. It could be positive duration and negative duration. Default values for window offset is **0**. The same record maybe assigned to the different window if set different offset value. For example, which window would be assigned to for a record with timestamp 2021-06-30 00:00:04 for a Tumble window with 10 MINUTE as size?

- If **offset** value is **-16** MINUTE, the record assigns to window [2021-06-29 23:54:00, 2021-06-30 00:04:00).
- If **offset** value is **-6** MINUTE, the record assigns to window [2021-06-29 23:54:00, 2021-06-30 00:04:00).
- If **offset** is **-4** MINUTE, the record assigns to window [2021-06-29 23:56:00, 2021-06-30 00:06:00).
- If **offset** is **0**, the record assigns to window [2021-06-30 00:00:00, 2021-06-30 00:10:00).
- If **offset** value is **4** MINUTE, the record assigns to window [2021-06-29 23:54:00, 2021-06-30 00:04:00).
- If **offset** is **6** MINUTE, the record assigns to window [2021-06-29 23:56:00, 2021-06-30 00:06:00).
- If **offset** is **16** MINUTE, the record assigns to window [2021-06-29 23:56:00, 2021-06-30 00:06:00). We could find that, some windows offset parameters may have same effect on the assignment of windows. In the above case, **-16** MINUTE, **-6** MINUTE and **4** MINUTE have same effect for a Tumble window with 10 MINUTE as size.

NOTE

The effect of window offset is just for updating window assignment, it has no effect on Watermark.

```
-- NOTE: Currently Flink doesn't support evaluating individual window table-valued function,
--  window table-valued function should be used with aggregate operation,
--  this example is just used for explaining the syntax and the data produced by table-valued function.
Flink SQL> SELECT * FROM TABLE(
   TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES, INTERVAL '1' MINUTES));
-- or with the named params
-- note: the DATA param must be the first
Flink SQL> SELECT * FROM TABLE(
   TUMBLE(
     DATA => TABLE Bid,
     TIMECOL => DESCRIPTOR(bidtime),
     SIZE => INTERVAL '10' MINUTES,
     OFFSET => INTERVAL '1' MINUTES));
+------------------+-------+------+------------------+------------------+-------------------------+
|          bidtime | price | item |     window_start |       window_end |             window_time  |
+------------------+-------+------+------------------+------------------+-------------------------+
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:01 | 2020-04-15 08:11 | 2020-04-15 08:10:59.999 |
| 2020-04-15 08:07 |  2.00 | A    | 2020-04-15 08:01 | 2020-04-15 08:11 | 2020-04-15 08:10:59.999 |
| 2020-04-15 08:09 |  5.00 | D    | 2020-04-15 08:01 | 2020-04-15 08:11 | 2020-04-15 08:10:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:11 | 2020-04-15 08:21 | 2020-04-15 08:20:59.999 |
```

```
| 2020-04-15 08:13 |  1.00 | E    | 2020-04-15 08:11 | 2020-04-15 08:21 | 2020-04-15 08:20:59.999 |
| 2020-04-15 08:17 |  6.00 | F    | 2020-04-15 08:11 | 2020-04-15 08:21 | 2020-04-15 08:20:59.999 |
+------------------+-------+------+------------------+------------------+-------------------------+

-- apply aggregation on the tumbling windowed table
Flink SQL> SELECT window_start, window_end, SUM(price)
  FROM TABLE(
    TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES, INTERVAL '1' MINUTES))
  GROUP BY window_start, window_end;
+------------------+------------------+-------+
|     window_start |       window_end | price |
+------------------+------------------+-------+
| 2020-04-15 08:01 | 2020-04-15 08:11 | 11.00 |
| 2020-04-15 08:11 | 2020-04-15 08:21 | 10.00 |
+------------------+------------------+-------+
```

1.6.4.2 Window Aggregation

Window TVF Aggregation

Window aggregations are defined in the **GROUP BY** clause contains "window_start" and "window_end" columns of the relation applied **Windowing TVF**. Just like queries with regular **GROUP BY** clauses, queries with a group by window aggregation will compute a single result row per group. Unlike other aggregations on continuous tables, window aggregation do not emit intermediate results but only a final result, the total aggregation at the end of the window. Moreover, window aggregations purge all intermediate state when no longer needed.

For more information, see **Window Aggregation**.

NOTE

The start and end timestamps of group windows can be selected with the grouped **window_start** and **window_end** columns.

- **Windowing TVFs**

  Flink supports **TUMBLE**, **HOP** and **CUMULATE** types of window aggregations.

  - In streaming mode, the time attribute field of a window table-valued function must be on either event or processing time attributes. See **Windowing TVF** for more windowing functions information.
  - In batch mode, the time attribute field of a window table-valued function must be an attribute of type **TIMESTAMP** or **TIMESTAMP_LTZ**.

```
-- tables must have time attribute, e.g. `bidtime` in this table
Flink SQL> desc Bid;
+-------------+------------------------+------+-----+--------+---------------------------------+
|        name |                   type | null | key | extras |                       watermark |
+-------------+------------------------+------+-----+--------+---------------------------------+
|     bidtime | TIMESTAMP(3) *ROWTIME* | true |     |        | `bidtime` - INTERVAL '1' SECOND |
|       price |         DECIMAL(10, 2) | true |     |        |                                 |
|        item |                 STRING | true |     |        |                                 |
| supplier_id |                 STRING | true |     |        |                                 |
+-------------+------------------------+------+-----+--------+---------------------------------+

Flink SQL> SELECT * FROM Bid;
+------------------+-------+------+-------------+
|          bidtime | price | item | supplier_id |
+------------------+-------+------+-------------+
| 2020-04-15 08:05 | 4.00  | C    | supplier1   |
| 2020-04-15 08:07 | 2.00  | A    | supplier1   |
| 2020-04-15 08:09 | 5.00  | D    | supplier2   |
```

```
| 2020-04-15 08:11 | 3.00  | B    | supplier2   |
| 2020-04-15 08:13 | 1.00  | E    | supplier1   |
| 2020-04-15 08:17 | 6.00  | F    | supplier2   |
+------------------+-------+------+-------------+

-- tumbling window aggregation
Flink SQL> SELECT window_start, window_end, SUM(price)
  FROM TABLE(
    TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))
  GROUP BY window_start, window_end;
+------------------+------------------+-------+
|     window_start |       window_end | price |
+------------------+------------------+-------+
| 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 |
+------------------+------------------+-------+

-- hopping window aggregation
Flink SQL> SELECT window_start, window_end, SUM(price)
  FROM TABLE(
    HOP(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES))
  GROUP BY window_start, window_end;
+------------------+------------------+-------+
|     window_start |       window_end | price |
+------------------+------------------+-------+
| 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 |
| 2020-04-15 08:05 | 2020-04-15 08:15 | 15.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 |
| 2020-04-15 08:15 | 2020-04-15 08:25 | 6.00  |
+------------------+------------------+-------+

-- cumulative window aggregation
Flink SQL> SELECT window_start, window_end, SUM(price)
  FROM TABLE(
    CUMULATE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '2' MINUTES, INTERVAL '10' MINUTES))
  GROUP BY window_start, window_end;
+------------------+------------------+-------+
|     window_start |       window_end | price |
+------------------+------------------+-------+
| 2020-04-15 08:00 | 2020-04-15 08:06 | 4.00  |
| 2020-04-15 08:00 | 2020-04-15 08:08 | 6.00  |
| 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 |
| 2020-04-15 08:10 | 2020-04-15 08:12 | 3.00  |
| 2020-04-15 08:10 | 2020-04-15 08:14 | 4.00  |
| 2020-04-15 08:10 | 2020-04-15 08:16 | 4.00  |
| 2020-04-15 08:10 | 2020-04-15 08:18 | 10.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 |
```

- **GROUPING SETS**

  Window aggregations also support **GROUPING SETS** syntax. Grouping sets allow for more complex grouping operations than those describable by a standard **GROUP BY**. Rows are grouped separately by each specified grouping set and aggregates are computed for each group just as for simple **GROUP BY** clauses.

  Window aggregations with **GROUPING SETS** require both the **window_start** and **window_end** columns have to be in the **GROUP BY** clause, but not in the **GROUPING SETS** clause.

```
Flink SQL> SELECT window_start, window_end, supplier_id, SUM(price) as price
  FROM TABLE(
    TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))
  GROUP BY window_start, window_end, GROUPING SETS ((supplier_id), ());
+------------------+------------------+-------------+-------+
|     window_start |       window_end | supplier_id | price |
+------------------+------------------+-------------+-------+
| 2020-04-15 08:00 | 2020-04-15 08:10 |      (NULL) | 11.00 |
| 2020-04-15 08:00 | 2020-04-15 08:10 |   supplier2 |  5.00 |
```

```
| 2020-04-15 08:00 | 2020-04-15 08:10 |   supplier1 |  6.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 |      (NULL) | 10.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 |   supplier2 |  9.00 |
| 2020-04-15 08:10 | 2020-04-15 08:20 |   supplier1 |  1.00 |
+------------------+------------------+-------------+-------+
```

Each sublist of **GROUPING SETS** may specify zero or more columns or expressions and is interpreted the same way as though used directly in the **GROUP BY** clause. An empty grouping set means that all rows are aggregated down to a single group, which is output even if no input rows were present.

References to the grouping columns or expressions are replaced by null values in result rows for grouping sets in which those columns do not appear. For example, **()** in **GROUPING SETS ((supplier_id), ())** in the preceding example is an empty sublist, and the **supplier_id** column in the corresponding result data is filled with **NULL**.

- **ROLLUP**

  ROLLUP is a shorthand notation for specifying a common type of grouping set. It represents the given list of expressions and all prefixes of the list, including the empty list.

  For example, **ROLLUP (one,two)** is equivalent to **GROUPING SET((one,two),(one),())**.

  Window aggregations with **ROLLUP** requires both the **window_start** and **window_end** columns have to be in the **GROUP BY** clause, but not in the **ROLLUP** clause.

  For example, the following query is equivalent to the one above.

```
SELECT window_start, window_end, supplier_id, SUM(price) as price
FROM TABLE(
    TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))
GROUP BY window_start, window_end, ROLLUP (supplier_id);
```

- **CUBE**

  CUBE is a shorthand notation for specifying a common type of grouping set. It represents the given list and all of its possible subsets - the power set.

  Window aggregations with **CUBE** requires both the **window_start** and **window_end** columns have to be in the **GROUP BY** clause, but not in the **CUBE** clause.

  For example, the following two queries are equivalent.

```
SELECT window_start, window_end, item, supplier_id, SUM(price) as price
  FROM TABLE(
    TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))
  GROUP BY window_start, window_end, CUBE (supplier_id, item);

SELECT window_start, window_end, item, supplier_id, SUM(price) as price
  FROM TABLE(
    TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))
  GROUP BY window_start, window_end, GROUPING SETS (
      (supplier_id, item),
      (supplier_id       ),
      (              item),
      (                  )
)
```

- **Cascading Window Aggregation**

  The **window_start** and **window_end** columns are regular timestamp columns, not time attributes. Thus they can not be used as time attributes in subsequent time-based operations.

To propagate time attributes, you need to additionally add **window_time** column into **GROUP BY** clause. The **window_time** is the third column produced by **Windowing Table-Valued Functions (Windowing TVFs)** which is a time attribute of the assigned window. Adding **window_time** into **GROUP BY** clause makes **window_time** also to be group key that can be selected. Then following queries can use this column for subsequent time-based operations, such as cascading window aggregations and Window TopN.

The following shows a cascading window aggregation where the first window aggregation propagates the time attribute for the second window aggregation.

```
-- tumbling 5 minutes for each supplier_id
CREATE VIEW window1 AS
-- Note: The window start and window end fields of inner Window TVF are optional in the select clause. However, if they appear in the clause, they need to be aliased to prevent name conflicting with the window start and window end of the outer Window TVF.
SELECT window_start as window_5mintumble_start, window_end as window_5mintumble_end, window_time as rowtime, SUM(price) as partial_price
  FROM TABLE(
    TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES))
  GROUP BY supplier_id, window_start, window_end, window_time;

-- tumbling 10 minutes on the first window
SELECT window_start, window_end, SUM(partial_price) as total_price
  FROM TABLE(
      TUMBLE(TABLE window1, DESCRIPTOR(rowtime), INTERVAL '10' MINUTES))
  GROUP BY window_start, window_end;
```

1.6.4.3 Window Top-N

Function

Window Top-N is a special Top-N which returns the N smallest or largest values for each window and other partitioned keys.

Unlike regular Top-N on continuous tables, window Top-N does not emit intermediate results but only a final result, the total top N records at the end of the window. Moreover, window Top-N purges all intermediate state when no longer needed.

Window Top-N queries have better performance if users do not need results updated per record. Usually, Window Top-N is used with **Windowing Table-Valued Functions (Windowing TVFs)** directly. Besides, Window Top-N could be used with other operations based on **Windowing Table-Valued Functions (Windowing TVFs)**, such as Window Aggregation, Window TopN and Window Join.

Window Top-N can be defined in the same syntax as regular Top-N, see Top-N documentation for more information. Besides that, Window Top-N requires the **PARTITION BY** clause contains **window_start** and **window_end** columns of the relation applied Windowing TVF or Window Aggregation. Otherwise, the optimizer will not be able to translate the query.

For more information, see **Window Top-N**.

Syntax

```
SELECT [column_list]
FROM (
```

```
  SELECT [column_list],
    ROW_NUMBER() OVER (PARTITION BY window_start, window_end [, col_key1...]
      ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum
  FROM table_name) -- relation applied windowing TVF
WHERE rownum <= N [AND conditions]
```

Caveats

Flink only supports Window Top-N follows after Windowing TVF with Tumble Windows, Hop Windows and Cumulate Windows.

Example

Window Top-N follows after Window Aggregation

The following example shows how to calculate Top 3 suppliers who have the highest sales for every tumbling 10 minutes window.

```
-- tables must have time attribute, e.g. `bidtime` in this table
Flink SQL> desc Bid;
+-------------+------------------------+------+-----+--------+---------------------------------+
|        name |                   type | null | key | extras |                       watermark |
+-------------+------------------------+------+-----+--------+---------------------------------+
|     bidtime | TIMESTAMP(3) *ROWTIME* | true |     |        | `bidtime` - INTERVAL '1' SECOND |
|       price |         DECIMAL(10, 2) | true |     |        |                                 |
|        item |                 STRING | true |     |        |                                 |
| supplier_id |                 STRING | true |     |        |                                 |
+-------------+------------------------+------+-----+--------+---------------------------------+

Flink SQL> SELECT * FROM Bid;
+------------------+-------+------+-------------+
|          bidtime | price | item | supplier_id |
+------------------+-------+------+-------------+
| 2020-04-15 08:05 |  4.00 |    A |   supplier1 |
| 2020-04-15 08:06 |  4.00 |    C |   supplier2 |
| 2020-04-15 08:07 |  2.00 |    G |   supplier1 |
| 2020-04-15 08:08 |  2.00 |    B |   supplier3 |
| 2020-04-15 08:09 |  5.00 |    D |   supplier4 |
| 2020-04-15 08:11 |  2.00 |    B |   supplier3 |
| 2020-04-15 08:13 |  1.00 |    E |   supplier1 |
| 2020-04-15 08:15 |  3.00 |    H |   supplier2 |
| 2020-04-15 08:17 |  6.00 |    F |   supplier5 |
+------------------+-------+------+-------------+

Flink SQL> SELECT *
  FROM (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY window_start, window_end ORDER BY price DESC) as
rownum
    FROM (
      SELECT window_start, window_end, supplier_id, SUM(price) as price, COUNT(*) as cnt
      FROM TABLE(
        TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))
      GROUP BY window_start, window_end, supplier_id
    )
  ) WHERE rownum <= 3;
+------------------+------------------+-------------+-------+-----+--------+
|     window_start |       window_end | supplier_id | price | cnt | rownum |
+------------------+------------------+-------------+-------+-----+--------+
| 2020-04-15 08:00 | 2020-04-15 08:10 |   supplier1 |  6.00 |   2 |      1 |
| 2020-04-15 08:00 | 2020-04-15 08:10 |   supplier4 |  5.00 |   1 |      2 |
| 2020-04-15 08:00 | 2020-04-15 08:10 |   supplier2 |  4.00 |   1 |      3 |
| 2020-04-15 08:10 | 2020-04-15 08:20 |   supplier5 |  6.00 |   1 |      1 |
| 2020-04-15 08:10 | 2020-04-15 08:20 |   supplier2 |  3.00 |   1 |      2 |
| 2020-04-15 08:10 | 2020-04-15 08:20 |   supplier3 |  2.00 |   1 |      3 |
+------------------+------------------+-------------+-------+-----+--------+
```

Window Top-N follows after Windowing TVF

The following example shows how to calculate Top 3 items which have the highest price for every tumbling 10 minutes window.

```
Flink SQL> SELECT *
  FROM (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY window_start, window_end ORDER BY price DESC) as
rownum
    FROM TABLE(
               TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))
  ) WHERE rownum <= 3;
+------------------+-------+------+-------------+------------------+------------------+--------+
|          bidtime | price | item | supplier_id |     window_start |       window_end | rownum |
+------------------+-------+------+-------------+------------------+------------------+--------+
| 2020-04-15 08:05 |  4.00 |    A |   supplier1 | 2020-04-15 08:00 | 2020-04-15 08:10 |      2 |
| 2020-04-15 08:06 |  4.00 |    C |   supplier2 | 2020-04-15 08:00 | 2020-04-15 08:10 |      3 |
| 2020-04-15 08:09 |  5.00 |    D |   supplier4 | 2020-04-15 08:00 | 2020-04-15 08:10 |      1 |
| 2020-04-15 08:11 |  2.00 |    B |   supplier3 | 2020-04-15 08:10 | 2020-04-15 08:20 |      3 |
| 2020-04-15 08:15 |  3.00 |    H |   supplier2 | 2020-04-15 08:10 | 2020-04-15 08:20 |      2 |
| 2020-04-15 08:17 |  6.00 |    F |   supplier5 | 2020-04-15 08:10 | 2020-04-15 08:20 |      1 |
+------------------+-------+------+-------------+------------------+------------------+--------+
```

1.6.4.4 Window Deduplication

Function

Window Deduplication is a special Deduplication which removes rows that duplicate over a set of columns, keeping the first one or the last one for each window and partitioned keys.

For streaming queries, unlike regular Deduplicate on continuous tables, Window Deduplication does not emit intermediate results but only a final result at the end of the window. Moreover, window Deduplication purges all intermediate state when no longer needed. Therefore, Window Deduplication queries have better performance if users do not need results updated per record. Usually, Window Deduplication is used with Windowing TVF directly. Besides, Window Deduplication could be used with other operations based on Windowing TVF, such as Window Aggregation, Window TopN and Window Join.

Window Top-N can be defined in the same syntax as regular Top-N, see Top-N documentation for more information. Besides that, Window Deduplication requires the **PARTITION BY** clause contains **window_start** and **window_end** columns of the relation. Otherwise, the optimizer will not be able to translate the query.

Flink uses **ROW_NUMBER()** to remove duplicates, just like the way of Window Top-N query. In theory, Window Deduplication is a special case of Window Top-N in which the N is one and order by the processing time or event time.

For more information, see **Window Deduplication**.

Syntax

```
SELECT [column_list]
FROM (
   SELECT [column_list],
     ROW_NUMBER() OVER (PARTITION BY window_start, window_end [, col_key1...]
       ORDER BY time_attr [asc|desc]) AS rownum
   FROM table_name) -- relation applied windowing TVF
WHERE (rownum = 1 | rownum <=1 | rownum < 2) [AND conditions]
```

Parameter description:

- **ROW_NUMBER()**: Assigns an unique, sequential number to each row, starting with one.
- **PARTITION BY window_start, window_end [, col_key1...]**: Specifies the partition columns which contain **window_start**, **window_end** and other partition keys.
- **ORDER BY time_attr [asc|desc]**: Specifies the ordering column, it must be a time attribute. Currently Flink supports processing time attribute and event time attribute. Ordering by ASC means keeping the first row, ordering by DESC means keeping the last row.
- **WHERE (rownum = 1 | rownum <=1 | rownum < 2)**: The **rownum = 1 | rownum <=1 | rownum < 2** is required for the optimizer to recognize the query could be translated to Window Deduplication.

Caveats

- Flink can only perform window deduplication on window table value functions that are based on tumble, hop, or cumulate windows.
- Window deduplication is only supported when sorting based on the event time attribute.

Example

The following example shows how to keep last record for every 10 minutes tumbling window.

```
-- tables must have time attribute, e.g. `bidtime` in this table
Flink SQL> DESC Bid;
+-------------+------------------------+------+-----+--------+---------------------------------+
|        name |                   type | null | key | extras |                       watermark |
+-------------+------------------------+------+-----+--------+---------------------------------+
|     bidtime | TIMESTAMP(3) *ROWTIME* | true |     |        | `bidtime` - INTERVAL '1' SECOND |
|       price |         DECIMAL(10, 2) | true |     |        |                                 |
|        item |                 STRING | true |     |        |                                 |
+-------------+------------------------+------+-----+--------+---------------------------------+

Flink SQL> SELECT * FROM Bid;
+------------------+-------+------+
|          bidtime | price | item |
+------------------+-------+------+
| 2020-04-15 08:05 |  4.00 | C    |
| 2020-04-15 08:07 |  2.00 | A    |
| 2020-04-15 08:09 |  5.00 | D    |
| 2020-04-15 08:11 |  3.00 | B    |
| 2020-04-15 08:13 |  1.00 | E    |
| 2020-04-15 08:17 |  6.00 | F    |
+------------------+-------+------+

Flink SQL> SELECT *
  FROM (
    SELECT bidtime, price, item, supplier_id, window_start, window_end,
      ROW_NUMBER() OVER (PARTITION BY window_start, window_end ORDER BY bidtime DESC) AS
rownum
    FROM TABLE(
             TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))
  ) WHERE rownum <= 1;
+------------------+-------+------+-------------+------------------+------------------+--------+
|          bidtime | price | item | supplier_id |     window_start |       window_end | rownum |
+------------------+-------+------+-------------+------------------+------------------+--------+
| 2020-04-15 08:09 |  5.00 |    D |   supplier4 | 2020-04-15 08:00 | 2020-04-15 08:10 |      1 |
| 2020-04-15 08:17 |  6.00 |    F |   supplier5 | 2020-04-15 08:10 | 2020-04-15 08:20 |      1 |
+------------------+-------+------+-------------+------------------+------------------+--------+
```

1.6.4.5 Window Join

A window join adds the dimension of time into the join criteria themselves. By doing so, the window join joins the elements of two streams that share a common key and are in the same window. The semantic of window join is same to the **DataStream window join**.

For streaming queries, unlike other joins on continuous tables, window join does not emit intermediate results but only emits final results at the end of the window. Moreover, window join purge all intermediate state when no longer needed. Usually, Window Join is used with Windowing TVF. Besides, Window Join could follow after other operations based on Windowing TVF, such as Window Aggregation, Window TopN and Window Join. Currently, Window Join requires the join on condition contains window starts equality of input tables and window ends equality of input tables. Window Join supports **INNER**/**LEFT**/**RIGHT**/**FULL OUTER**/**ANTI**/**SEMI JOIN**.

For more information, see **Window Join**.

Caveats

- Currently, the window join requires the join on condition contains window starts equality of input tables and window ends equality of input tables.
- Currently, the windowing TVFs must be the same of left and right inputs.
- Currently, if Window Join follows after Windowing TVF, the Windowing TVF has to be with Tumble Windows, Hop Windows or Cumulate Windows instead of Session windows.

INNER/LEFT/RIGHT/FULL OUTER

The syntax of INNER/LEFT/RIGHT/FULL OUTER WINDOW JOIN are very similar with each other, we only give an example for FULL OUTER JOIN here. When performing a window join, all elements with a common key and a common tumbling window are joined together. We only give an example for a Window Join which works on a Tumble Window TVF. By scoping the region of time for the join into fixed five-minute intervals, we chopped our datasets into two distinct windows of time: [12:00, 12:05) and [12:05, 12:10). The L2 and R2 rows could not join together because they fell into separate windows.

Syntax

```
SELECT ...
FROM L [LEFT|RIGHT|FULL OUTER] JOIN R -- L and R are relations applied windowing TVF
ON L.window_start = R.window_start AND L.window_end = R.window_end AND ...
```

Example

When performing a window join, all elements with a common key and a common tumbling window are joined together. We only give an example for a Window Join which works on a Tumble Window TVF. By scoping the region of time for the join into fixed five-minute intervals, we chopped our datasets into two distinct windows of time: [12:00, 12:05) and [12:05, 12:10). The L2 and R2 rows could not join together because they fell into separate windows.

```
Flink SQL> desc LeftTable;
+----------+------------------------+------+-----+--------+----------------------------------+
|     name |                   type | null | key | extras |                        watermark |
```

```
+----------+-----------------------+------+-----+--------+----------------------------------+
| row_time | TIMESTAMP(3) *ROWTIME* | true |     |        | `row_time` - INTERVAL '1' SECOND |
|      num |                    INT | true |     |        |                                  |
|       id |                 STRING | true |     |        |                                  |
+----------+-----------------------+------+-----+--------+----------------------------------+

Flink SQL> SELECT * FROM LeftTable;
+------------------+-----+----+
|         row_time | num | id |
+------------------+-----+----+
| 2020-04-15 12:02 |   1 | L1 |
| 2020-04-15 12:06 |   2 | L2 |
| 2020-04-15 12:03 |   3 | L3 |
+------------------+-----+----+

Flink SQL> desc RightTable;
+----------+-----------------------+------+-----+--------+----------------------------------+
|     name |                   type | null | key | extras |                        watermark |
+----------+-----------------------+------+-----+--------+----------------------------------+
| row_time | TIMESTAMP(3) *ROWTIME* | true |     |        | `row_time` - INTERVAL '1' SECOND |
|      num |                    INT | true |     |        |                                  |
|       id |                 STRING | true |     |        |                                  |
+----------+-----------------------+------+-----+--------+----------------------------------+

Flink SQL> SELECT * FROM RightTable;
+------------------+-----+----+
|         row_time | num | id |
+------------------+-----+----+
| 2020-04-15 12:01 |   2 | R2 |
| 2020-04-15 12:04 |   3 | R3 |
| 2020-04-15 12:05 |   4 | R4 |
+------------------+-----+----+

Flink SQL> SELECT L.num as L_Num, L.id as L_Id, R.num as R_Num, R.id as R_Id,
           COALESCE(L.window_start, R.window_start) as window_start,
           COALESCE(L.window_end, R.window_end) as window_end
           FROM (
               SELECT * FROM TABLE(TUMBLE(TABLE LeftTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
           ) L
           FULL JOIN (
               SELECT * FROM TABLE(TUMBLE(TABLE RightTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
           ) R
           ON L.num = R.num AND L.window_start = R.window_start AND L.window_end = R.window_end;
+-------+------+-------+------+------------------+------------------+
| L_Num | L_Id | R_Num | R_Id |     window_start |       window_end |
+-------+------+-------+------+------------------+------------------+
|     1 |   L1 |  null | null | 2020-04-15 12:00 | 2020-04-15 12:05 |
|  null | null |     2 |   R2 | 2020-04-15 12:00 | 2020-04-15 12:05 |
|     3 |   L3 |     3 |   R3 | 2020-04-15 12:00 | 2020-04-15 12:05 |
|     2 |   L2 |  null | null | 2020-04-15 12:05 | 2020-04-15 12:10 |
|  null | null |     4 |   R4 | 2020-04-15 12:05 | 2020-04-15 12:10 |
+-------+------+-------+------+------------------+------------------+
```

SEMI

Semi Window Joins returns a row from one left record if there is at least one matching row on the right side within the common window.

```
Flink SQL> SELECT *
           FROM (
               SELECT * FROM TABLE(TUMBLE(TABLE LeftTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
           ) L WHERE L.num IN (
             SELECT num FROM (
               SELECT * FROM TABLE(TUMBLE(TABLE RightTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
             ) R WHERE L.window_start = R.window_start AND L.window_end = R.window_end);
```

```
+------------------+-----+----+------------------+------------------+-------------------------+
|         row_time | num | id |     window_start |       window_end |            window_time  |
+------------------+-----+----+------------------+------------------+-------------------------+
| 2020-04-15 12:03 |   3 | L3 | 2020-04-15 12:00 | 2020-04-15 12:05 | 2020-04-15 12:04:59.999 |
+------------------+-----+----+------------------+------------------+-------------------------+

Flink SQL> SELECT *
           FROM (
               SELECT * FROM TABLE(TUMBLE(TABLE LeftTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
           ) L WHERE EXISTS (
             SELECT * FROM (
               SELECT * FROM TABLE(TUMBLE(TABLE RightTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
             ) R WHERE L.num = R.num AND L.window_start = R.window_start AND L.window_end =
R.window_end);
+------------------+-----+----+------------------+------------------+-------------------------+
|         row_time | num | id |     window_start |       window_end |            window_time  |
+------------------+-----+----+------------------+------------------+-------------------------+
| 2020-04-15 12:03 |   3 | L3 | 2020-04-15 12:00 | 2020-04-15 12:05 | 2020-04-15 12:04:59.999 |
+------------------+-----+----+------------------+------------------+-------------------------+
```

ANTI

Anti Window Joins are the obverse of the Inner Window Join: they contain all of the unjoined rows within each common window.

```
Flink SQL> SELECT *
           FROM (
               SELECT * FROM TABLE(TUMBLE(TABLE LeftTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
           ) L WHERE L.num NOT IN (
             SELECT num FROM (
               SELECT * FROM TABLE(TUMBLE(TABLE RightTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
             ) R WHERE L.window_start = R.window_start AND L.window_end = R.window_end);
+------------------+-----+----+------------------+------------------+-------------------------+
|         row_time | num | id |     window_start |       window_end |            window_time  |
+------------------+-----+----+------------------+------------------+-------------------------+
| 2020-04-15 12:02 |   1 | L1 | 2020-04-15 12:00 | 2020-04-15 12:05 | 2020-04-15 12:04:59.999 |
| 2020-04-15 12:06 |   2 | L2 | 2020-04-15 12:05 | 2020-04-15 12:10 | 2020-04-15 12:09:59.999 |
+------------------+-----+----+------------------+------------------+-------------------------+

Flink SQL> SELECT *
           FROM (
               SELECT * FROM TABLE(TUMBLE(TABLE LeftTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
           ) L WHERE NOT EXISTS (
             SELECT * FROM (
               SELECT * FROM TABLE(TUMBLE(TABLE RightTable, DESCRIPTOR(row_time), INTERVAL '5'
MINUTES))
             ) R WHERE L.num = R.num AND L.window_start = R.window_start AND L.window_end =
R.window_end);
+------------------+-----+----+------------------+------------------+-------------------------+
|         row_time | num | id |     window_start |       window_end |            window_time  |
+------------------+-----+----+------------------+------------------+-------------------------+
| 2020-04-15 12:02 |   1 | L1 | 2020-04-15 12:00 | 2020-04-15 12:05 | 2020-04-15 12:04:59.999 |
| 2020-04-15 12:06 |   2 | L2 | 2020-04-15 12:05 | 2020-04-15 12:10 | 2020-04-15 12:09:59.999 |
+------------------+-----+----+------------------+------------------+-------------------------+
```

1.6.5 Group Aggregation

An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the **COUNT**, **SUM**, **AVG** (average), **MAX** (maximum) and **MIN** (minimum) over a set of rows.

For streaming queries, the required state for computing the query result might grow infinitely. State size depends on the number of groups and the number and type of aggregation functions. For example MIN/MAX are heavy on state size while COUNT is cheap. You can provide a query configuration with an appropriate state time-to-live (TTL) to prevent excessive state size. Note that this might affect the correctness of the query result.

For more information, see **Group Aggregation**.

DISTINCT Aggregation

Distinct aggregates remove duplicate values before applying an aggregation function. The following example counts the number of distinct order_ids instead of the total number of rows in the **Orders** table.

```
SELECT COUNT(DISTINCT order_id) FROM Orders
```

GROUPING SETS

Grouping sets allow for more complex grouping operations than those describable by a standard **GROUP BY**. Rows are grouped separately by each specified grouping set and aggregates are computed for each group just as for simple **GROUP BY** clauses.

Each sublist of **GROUPING SETS** may specify zero or more columns or expressions and is interpreted the same way as though used directly in the **GROUP BY** clause. An empty grouping set means that all rows are aggregated down to a single group, which is output even if no input rows were present.

References to the grouping columns or expressions are replaced by null values in result rows for grouping sets in which those columns do not appear.

```
SELECT supplier_id, rating, COUNT(*) AS total
FROM (VALUES
    ('supplier1', 'product1', 4),
    ('supplier1', 'product2', 3),
    ('supplier2', 'product3', 3),
    ('supplier2', 'product4', 4))
AS Products(supplier_id, product_id, rating)
GROUP BY GROUPING SETS ((supplier_id, rating), (supplier_id), ())
```

ROLLUP

ROLLUP is a shorthand notation for specifying a common type of grouping set. It represents the given list of expressions and all prefixes of the list, including the empty list.

```
SELECT supplier_id, rating, COUNT(*)
FROM (VALUES
    ('supplier1', 'product1', 4),
    ('supplier1', 'product2', 3),
    ('supplier2', 'product3', 3),
    ('supplier2', 'product4', 4))
AS Products(supplier_id, product_id, rating)
GROUP BY ROLLUP (supplier_id, rating)
```

CUBE

CUBE is a shorthand notation for specifying a common type of grouping set. It represents the given list and all of its possible subsets - the power set.

For example, the following two queries are equivalent.

```
SELECT supplier_id, rating, product_id, COUNT(*)
FROM (VALUES
    ('supplier1', 'product1', 4),
    ('supplier1', 'product2', 3),
    ('supplier2', 'product3', 3),
    ('supplier2', 'product4', 4))
AS Products(supplier_id, product_id, rating)
GROUP BY CUBE (supplier_id, rating, product_id)

SELECT supplier_id, rating, product_id, COUNT(*)
FROM (VALUES
    ('supplier1', 'product1', 4),
    ('supplier1', 'product2', 3),
    ('supplier2', 'product3', 3),
    ('supplier2', 'product4', 4))
AS Products(supplier_id, product_id, rating)
GROUP BY GROUPING SET (
    ( supplier_id, product_id, rating ),
    ( supplier_id, product_id         ),
    ( supplier_id,             rating ),
    ( supplier_id                     ),
    (              product_id, rating ),
    (              product_id         ),
    (                          rating ),
    (                                 )
)
```

HAVING

HAVING eliminates group rows that do not satisfy the condition. **HAVING** is different from **WHERE**: **WHERE** filters individual rows before the **GROUP BY** while **HAVING** filters group rows created by **GROUP BY**. Each column referenced in condition must unambiguously reference a grouping column unless it appears within an aggregate function.

The presence of **HAVING** turns a query into a grouped query even if there is no **GROUP BY** clause. It is the same as what happens when the query contains aggregate functions but no **GROUP BY** clause. The query considers all selected rows to form a single group, and the **SELECT** list and **HAVING** clause can only reference table columns from within aggregate functions. Such a query will emit a single row if the **HAVING** condition is true, zero rows if it is not true.

```
SELECT SUM(amount)
FROM Orders
GROUP BY users
HAVING SUM(amount) > 50
```

1.6.6 Over Aggregation

OVER aggregates compute an aggregated value for every input row over a range of ordered rows. In contrast to **GROUP BY** aggregates, **OVER** aggregates do not reduce the number of result rows to a single row for every group. Instead **OVER** aggregates produce an aggregated value for every input row.

For more information, see **Over Aggregation**.

Syntax

```
SELECT
  agg_func(agg_col) OVER (
    [PARTITION BY col1[, col2, ...]]
```

```
  ORDER BY time_col
  range_definition),
 ...
FROM ...
```

Caveats

- Currently, only windows from **PRECEDING** (unbounded or bounded) to **CURRENT ROW** are supported. The range described by **FOLLOWING** is not supported.
- **ORDER BY** must be specified for a single time attribute.
- You can define multiple **OVER** window aggregates in a **SELECT** clause. However, for streaming queries, the **OVER** windows for all aggregates must be identical due to current limitation.
- **OVER** windows are defined on an ordered sequence of rows. Since tables do not have an inherent order, the **ORDER BY** clause is mandatory. For streaming queries, Flink currently only supports **OVER** windows that are defined with an ascending time attributes order. Additional orderings are not supported.

Description

```
SELECT order_id, order_time, amount,
  SUM(amount) OVER w AS sum_amount,
  AVG(amount) OVER w AS avg_amount
FROM Orders
WINDOW w AS (
  PARTITION BY product
  ORDER BY order_time
  RANGE BETWEEN INTERVAL '1' HOUR PRECEDING AND CURRENT ROW)
```

- **ORDER BY**: **OVER** windows are defined on an ordered sequence of rows. Since tables do not have an inherent order, the **ORDER BY** clause is mandatory. For streaming queries, Flink currently only supports **OVER** windows that are defined with an ascending time attributes order. Additional orderings are not supported.
- **PARTITION BY**: **OVER** windows can be defined on a partitioned table. In presence of a **PARTITION BY** clause, the aggregate is computed for each input row only over the rows of its partition.
- **Range Definitions**: The range definition specifies how many rows are included in the aggregate. The range is defined with a **BETWEEN** clause that defines a lower and an upper boundary. All rows between these boundaries are included in the aggregate. Flink only supports **CURRENT ROW** as the upper boundary. There are two options to define the range, **ROWS** intervals and **RANGE** intervals.

  a. **RANGE intervals**

     A **RANGE** interval is defined on the values of the **ORDER BY** column, which is in case of Flink always a time attribute. The following **RANGE** interval defines that all rows with a time attribute of at most 30 minutes less than the current row are included in the aggregate.

     ```
     RANGE BETWEEN INTERVAL '30' MINUTE PRECEDING AND CURRENT ROW
     ```

  b. **ROW intervals**

     A **ROWS** interval is a count-based interval. It defines exactly how many rows are included in the aggregate. The following **ROWS** interval defines that the 10 rows preceding the current row and the current row (so 11 rows in total) are included in the aggregate.

```
ROWS BETWEEN 10 PRECEDING AND CURRENT ROW
```

- **WINDOW**: The **WINDOW** clause can be used to define an **OVER** window outside of the **SELECT** clause. It can make queries more readable and also allows us to reuse the window definition for multiple aggregates.

Example

The following query computes for every order the sum of amounts of all orders for the same product that were received within one hour before the current order.

```
SELECT order_id, order_time, amount,
  SUM(amount) OVER (
    PARTITION BY product
    ORDER BY order_time
    RANGE BETWEEN INTERVAL '1' HOUR PRECEDING AND CURRENT ROW
  ) AS one_hour_prod_amount_sum
FROM Orders
```

1.6.7 JOIN

Equi-join

Syntax

```
FROM tableExpression INNER | LEFT | RIGHT | FULL JOIN tableExpression
  ON value11 = value21 [ AND value12 = value22]
```

Precautions

- Currently, only equi-joins are supported, for example, joins that have at least one conjunctive condition with an equality predicate. Arbitrary cross or theta joins are not supported.
- Tables are joined in the order in which they are specified in the FROM clause. Make sure to specify tables in an order that does not yield a cross join (Cartesian product), which are not supported and would cause a query to fail.
- For streaming queries the required state to compute the query result might grow infinitely depending on the type of aggregation and the number of distinct grouping keys. Provide a query configuration with valid retention interval to prevent excessive state size.

Example

```
SELECT *
FROM Orders INNER JOIN Product ON Orders.productId = Product.id;

SELECT *
FROM Orders LEFT JOIN Product ON Orders.productId = Product.id;

SELECT *
FROM Orders RIGHT JOIN Product ON Orders.productId = Product.id;

SELECT *
FROM Orders FULL OUTER JOIN Product ON Orders.productId = Product.id;
```

Time-windowed Join

Function

Each piece of data in a stream is joined with data in different time zones in another stream.

Syntax

```
from t1 JOIN t2 ON t1.key = t2.key AND TIMEBOUND_EXPRESSIO
```

Description

TIMEBOUND_EXPRESSION can be in either of the following formats:

- L.time between LowerBound(R.time) and UpperBound(R.time)
- R.time between LowerBound(L.time) and UpperBound(L.time)
- Comparison expression with the time attributes (L.time/R.time)

Precautions

A time window join requires at least one equi join predicate and a join condition that limits the time of both streams.

For example, use two range predicates (<, <=, >=, or >), a BETWEEN predicate, or an equal predicate that compares the same type of time attributes (such as processing time and event time) in two input tables.

For example, the following predicate is a valid window join condition:

- ltime = rtime
- ltime >= rtime AND ltime < rtime + INTERVAL '10' MINUTE
- ltime BETWEEN rtime - INTERVAL '10' SECOND AND rtime + INTERVAL '5' SECOND

Example

Join all orders shipped within 4 hours with their associated shipments.

```
SELECT *
FROM Orders o, Shipments s
WHERE o.id = s.orderId AND
      o.ordertime BETWEEN s.shiptime - INTERVAL '4' HOUR AND s.shiptime;
```

Expanding arrays into a relation

Precautions

This clause is used to return a new row for each element in the given array. Unnesting WITH ORDINALITY is not yet supported.

Example

```
SELECT users, tag
FROM Orders CROSS JOIN UNNEST(tags) AS t (tag);
```

User-Defined Table Functions

Function

This clause is used to join a table with the results of a table function. ach row of the left (outer) table is joined with all rows produced by the corresponding call of the table function.

Precautions

A left outer join against a lateral table requires a TRUE literal in the ON clause.

Example

The row of the left (outer) table is dropped, if its table function call returns an empty result.

```
SELECT users, tag
FROM Orders, LATERAL TABLE(unnest_udtf(tags)) t AS tag;
```

If a table function call returns an empty result, the corresponding outer row is preserved, and the result padded with null values.

```
SELECT users, tag
FROM Orders LEFT JOIN LATERAL TABLE(unnest_udtf(tags)) t AS tag ON TRUE;
```

Join Temporal Table Function

Function

Precautions

Currently only inner join and left outer join with temporal tables are supported.

Example

Assuming Rates is a temporal table function, the join can be expressed in SQL as follows:

```
SELECT
  o_amount, r_rate
FROM
  Orders,
  LATERAL TABLE (Rates(o_proctime))
WHERE
  r_currency = o_currency;
```

Join Temporal Tables

Function

This clause is used to join the Temporal table.

Syntax

```
SELECT column-names
FROM table1  [AS <alias1>]
[LEFT] JOIN table2 FOR SYSTEM_TIME AS OF table1.proctime [AS <alias2>]
ON table1.column-name1 = table2.key-name1
```

Description

- **table1.proctime** indicates the processing time attribute (computed column) of **table1**.
- **FOR SYSTEM_TIME AS OF table1.proctime** indicates that when the records in the left table are joined with the dimension table on the right, only the snapshot data is used for matching the current processing time dimension table.

Precautions

Only inner and left joins are supported for temporal tables with processing time attributes.

Example

LatestRates is a dimension table (such as HBase table) that is materialized with the latest rate.

```
SELECT
  o.amout, o.currency, r.rate, o.amount * r.rate
FROM
  Orders AS o
  JOIN LatestRates FOR SYSTEM_TIME AS OF o.proctime AS r
  ON r.currency = o.currency;
```

1.6.8 OrderBy & Limit

OrderBy

Function

This clause is used to sort data in ascending order on a time attribute.

Precautions

Currently, only sorting by time attribute is supported.

Example

Sort data in ascending order on the time attribute.

```
SELECT *
FROM Orders
ORDER BY orderTime;
```

Limit

Function

This clause is used to constrain the number of rows returned.

Precautions

This clause is used in conjunction with ORDER BY to ensure that the results are deterministic.

Example

```
SELECT *
FROM Orders
ORDER BY orderTime
LIMIT 3;
```

1.6.9 Top-N

Function

Top-N queries ask for the N smallest or largest values ordered by columns. Both smallest and largest values sets are considered Top-N queries. Top-N queries are useful in cases where the need is to display only the N bottom-most or the N top-most records from batch/streaming table on a condition.

Syntax

```
SELECT [column_list]
FROM (
```

```
  SELECT [column_list],
    ROW_NUMBER() OVER ([PARTITION BY col1[, col2...]]
      ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum
  FROM table_name)
WHERE rownum <= N [AND conditions]
```

Description

- ROW_NUMBER(): Allocate a unique and consecutive number to each line starting from the first line in the current partition. Currently, we only support ROW_NUMBER as the over window function. In the future, we will support RANK() and DENSE_RANK().
- PARTITION BY col1[, col2...]: Specifies the partition columns. Each partition will have a Top-N result.
- ORDER BY col1 [asc|desc][, col2 [asc|desc]...]: Specifies the ordering columns. The ordering directions can be different on different columns.
- WHERE rownum <= N: The rownum <= N is required for Flink to recognize this query is a Top-N query. The N represents the N smallest or largest records will be retained.
- [AND conditions]: It is free to add other conditions in the where clause, but the other conditions can only be combined with rownum <= N using AND conjunction.

Precautions

- The TopN query is Result Updating.
- Flink SQL will sort the input data stream according to the order key,
- so if the top N records have been changed, the changed ones will be sent as retraction/update records to downstream.
- If the top N records need to be stored in external storage, the result table should have the same unique key with the Top-N query.

Example

This is an example to get the top five products per category that have the maximum sales in realtime.

```
SELECT *
  FROM (
    SELECT *,
        ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) as row_num
    FROM ShopSales)
  WHERE row_num <= 5;
```

1.6.10 Deduplication

Function

Deduplication removes rows that duplicate over a set of columns, keeping only the first one or the last one.

Syntax

```
SELECT [column_list]
FROM (
```

```
  SELECT [column_list],
    ROW_NUMBER() OVER ([PARTITION BY col1[, col2...]]
      ORDER BY time_attr [asc|desc]) AS rownum
  FROM table_name)
WHERE rownum = 1
```

Description

- ROW_NUMBER(): Assigns a unique, sequential number to each row, starting with one.
- PARTITION BY col1[, col2...]: Specifies the partition columns, i.e. the deduplicate key.
- ORDER BY time_attr [asc|desc]: Specifies the ordering column, it must be a time attribute. Currently Flink supports proctime only. Ascending (ASC) sorting keeps only the first row, while descending (DESC) sorting keeps only the last row.
- WHERE rownum = 1: The rownum = 1 is required for Flink to recognize this query is deduplication.

Precautions

None

Example

The following examples show how to remove duplicate rows on **order_id**. The proctime is an event time attribute.

```
SELECT order_id, user, product, number
  FROM (
     SELECT *,
         ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY proctime ASC) as row_num
     FROM Orders)
  WHERE row_num = 1;
```

1.7 Functions

1.7.1 UDFs

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

Currently, UDF, UDTF, or UDAF custom functions cannot be written using Python.

POM Dependency

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-common</artifactId>
    <version>1.15.0</version>
    <scope>provided</scope>
</dependency>
```

Using UDFs

1. Encapsulate the implemented UDFs into a JAR package and upload the package to OBS.
2. In the navigation pane of the DLI management console, choose **Data Management** > **Package Management**. On the displayed page, click **Create** and use the JAR package uploaded to OBS to create a package.
3. In the left navigation, choose **Job Management** and click **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. Click the **Running Parameters** tab of your job, select the UDF JAR and click **Save**.
5. Add the following statement to the SQL statements to use the functions:
```
CREATE FUNCTION udf_test AS 'com.huaweicompany.udf.UdfScalarFunction';
```

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.huaweicompany.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;

public class UdfTableFunction extends TableFunction<Row> {
    /**
   * (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.huaweicompany.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.huaweicompany.udf.UdfAggFunction';
INSERT INTO sink_stream SELECT udaf_test(attr2) FROM source_stream GROUP BY attr1;
```


1.7.2 Type Inference

Scenario

Type inference summarizes the logic for validating input arguments and deriving data types for both the parameters and the result of a function. From a logical perspective, the planner needs information about expected types, precision, and scale. From a JVM perspective, the planner needs information about how internal data structures are represented as JVM objects when calling a user-defined function.

Flink's user-defined functions implement an automatic type inference extraction that derives data types from the function's class and its evaluation methods via reflection. However, this implicit reflective extraction approach is not always successful, for example, the Row type commonly used in UDTF cannot be extracted.

Flink 1.11 introduced a UDF registration interface and used a type inference approach, which does not support **getResultType** overload to declare the returned type in Flink 1.10. If you use this approach, the following exception will be thrown:

```
Caused by: org.apache.flink.table.api.ValidationException: Cannot extract a data type from a pure
'org.apache.flink.types.Row' class. Please use annotations to define field names and field types.
```

With Flink 1.15, the extraction process can be supported by annotating affected parameters, classes, or methods with @DataTypeHint and @FunctionHint.

Code Samples

The table ecosystem (similar to the SQL standard) is a strongly typed API. Therefore, both function parameters and return types must be mapped to a **data type**.

If more advanced type inference logic is required, an implementer can explicitly override the **getTypeInference()** method in every user-defined function.

However, the annotation approach is recommended because it keeps custom type inference logic close to the affected locations and falls back to the default behavior for the remaining implementation.

```
importorg.apache.flink.table.annotation.DataTypeHint;
importorg.apache.flink.table.annotation.FunctionHint;
importorg.apache.flink.table.functions.FunctionContext;
importorg.apache.flink.table.functions.TableFunction;
importorg.apache.flink.types.Row;
publicclassUdfTableFunctionextendsTableFunction<Row>{
    /**
     * Initialization, which is optional
     *@paramcontext
     */
    @Override
     public void open(FunctionContextcontext) {  }

    @FunctionHint(output=@DataTypeHint("ROW<s STRING, i INT>"))
    publicvoideval(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);
        }
```

```
  }
  /**
  * The following is optional.
  */
 @Override
 public void close() {}
}
```

Use 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.huaweicompany.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;
```

1.7.3 Parameter Transfer

Scenario

A UDF can be used in many jobs, and some parameter values vary with jobs. To easily modify the parameter values, you can set **pipeline.global-job-parameters** in the **Runtime Configuration** tab on the Flink OpenSource SQL editing page, and then get the parameter values in the UDF code and use the values as you need. You only need to change the parameter values in the runtime configuration tab to pass the new values to the UDF.

Procedure

Use the open(FunctionContext context) method in your UDF to pass parameters through a FunctionContext object. To pass parameters to a job, perform the following steps:

1. Add **pipeline.global-job-parameters** to **Runtime Configuration** on the Flink OpenSource SQL editing page. The format is as follows:
```
pipeline.global-job-parameters=k1:v1,"k2:v1,v2",k3:"str:ing","k4:str""ing"
```
   This configuration defines a map as shown in **Table 1-74**

   Table 1-74 Examples for pipeline.global-job-parameters

Key	Value
k1	v1
k2	v1,v2
k3	str:ing

Key	Value
k4	str""ing

NOTE

- **FunctionContext#getJobParameter** obtains only the value of **pipeline.global-job-parameters**. You need to add all key-value pairs that will be used in the UDF to **pipeline.global-job-parameters**.
- Keys and values are separated by colons (:). All key-values are connected by commas (,).
- If the key or value contains commas (,), use double quotation marks (") to enclose key or value, for example, **"v1,v2"**.
- If the key or value contains colons (:), use double quotation marks (") to enclose the key or value, for example, **"str:ing"**.
- If the key or value contains a double quotation mark("), use another double quotation mark ("") to escape the first one, and use double quotation marks (") to enclose the key or value, for example, **"str""ing"**.

2. In your UDF code, use **FunctionContext#getJobParameter** to obtain the key-value pairs you set. The code example is as follows:

```
context.getJobParameter("url","jdbc:mysql://xx.xx.xx.xx:3306/table");
context.getJobParameter("driver","com.mysql.jdbc.Driver");
context.getJobParameter("user","user");
context.getJobParameter("password","password");
```

Code Samples

The following sample UDF uses **pipeline.global-job-parameters** to pass parameters such as **url**, **user**, and **password** required for connecting to the database, obtains the **udf_info** table data, and combines this data with the stream data into JSON output.

Table 1-75 udf_info

key	value
class	class-4

SimpleJsonBuild.java

```
package udf;

import com.fasterxml.jackson.databind.ObjectMapper;

import org.apache.flink.table.functions.FunctionContext;
import org.apache.flink.table.functions.ScalarFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```
import java.util.HashMap;
import java.util.Map;

public class SimpleJsonBuild extends ScalarFunction {
   private static final Logger LOG = LoggerFactory.getLogger(SimpleJsonBuild.class);
   String remainedKey;
   String remainedValue;

   private Connection initConnection(Map<String, String> userParasMap) {
       String url = userParasMap.get("url");
       String driver = userParasMap.get("driver");
       String user = userParasMap.get("user");
       String password = userParasMap.get("password");
       Connection conn = null;
       try {
           Class.forName(driver);
           conn = DriverManager.getConnection(url, user, password);
           LOG.info("connect successfully");
      } catch (Exception e) {
           LOG.error(String.valueOf(e));
      }
       return conn;
  }

   @Override
   public void open(FunctionContext context) throws Exception {
       Map<String, String> userParasMap = new HashMap<>();
       Connection connection;
       PreparedStatement pstmt;
       ResultSet rs;

       String url = context.getJobParameter("url","jdbc:mysql://xx.xx.xx.xx:3306/table");
       String driver = context.getJobParameter("driver","com.mysql.jdbc.Driver");
       String user = context.getJobParameter("user","user");
       String password = context.getJobParameter("password","password");

       userParasMap.put("url", url);
       userParasMap.put("driver", driver);
       userParasMap.put("user", user);
       userParasMap.put("password", password);

       connection = initConnection(userParasMap);
       String sql = "select `key`, `value` from udf_info";
       pstmt = connection.prepareStatement(sql);
       rs = pstmt.executeQuery();

       while (rs.next()) {
           remainedKey = rs.getString(1);
           remainedValue = rs.getString(2);
      }
  }

   public String eval(String... params) throws IOException {
       if (params != null && params.length != 0 && params.length % 2 <= 0) {
           HashMap<String, String> hashMap = new HashMap();
           for (int i = 0; i < params.length; i += 2) {
               hashMap.put(params[i], params[i + 1]);
               LOG.debug("now the key is " + params[i].toString() + "; now the value is " + params[i + 
1].toString());
          }
           hashMap.put(remainedKey, remainedValue);
           ObjectMapper mapper = new ObjectMapper();
           String result = "{}";
           try {
               result = mapper.writeValueAsString(hashMap);
          } catch (Exception ex) {
               LOG.error("Get result failed." + ex.getMessage());
          }
           LOG.debug(result);
```

```
            return result;
        } else {
            return "{}";
        }
    }

    public static void main(String[] args) throws IOException {
        SimpleJsonBuild  sjb = new SimpleJsonBuild();
        System.out.println(sjb.eval("json1", "json2", "json3", "json4"));
    }
}
```

Add **pipeline.global-job-parameters** to **Runtime Configuration** on the Flink OpenSource SQL editing page. The format is as follows:

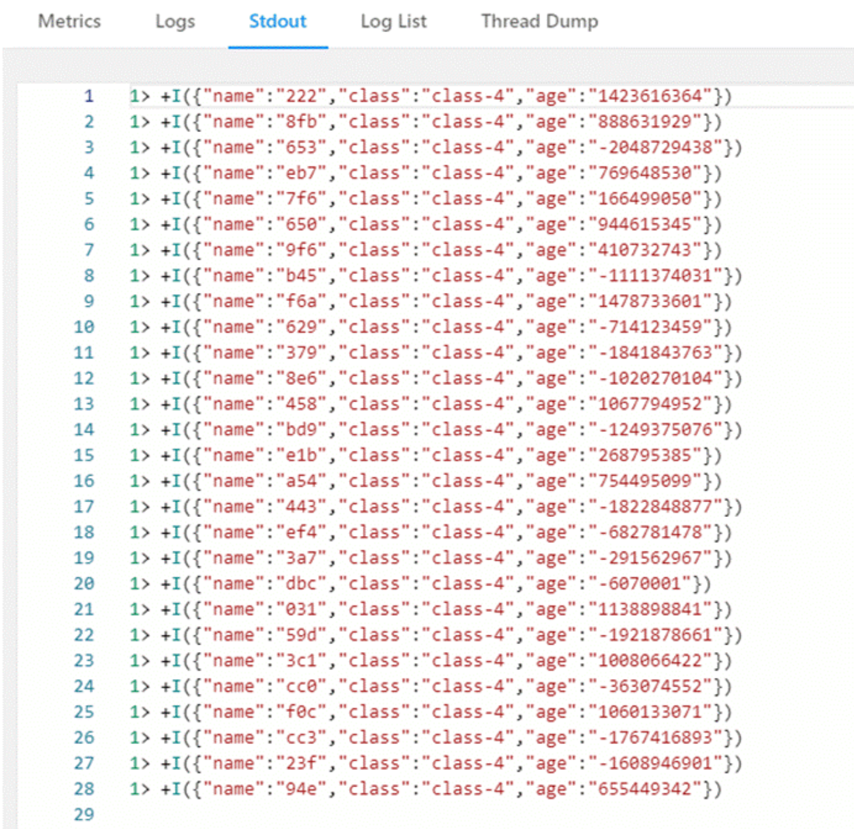
```
pipeline.global-job-parameters=url:'jdbc:mysql://x.x.x.x:xxxx/
test',driver:com.mysql.jdbc.Driver,user:xxx,password:xxx
```

Flink OpenSource SQL

```
create function SimpleJsonBuild AS 'udf.SimpleJsonBuild';
create table dataGenSource(user_id string, amount int) with (
 'connector' = 'datagen',
 'rows-per-second' = '1', --Generate a piece of data per second.
 'fields.user_id.kind' = 'random', --Specify a random generator for the user_id field.
 'fields.user_id.length' = '3' --Limit the length of user_id to 3.
);
create table printSink(message STRING) with ('connector' = 'print');
insert into
printSink
SELECT
SimpleJsonBuild("name", user_id, "age", cast(amount as string))
from
dataGenSource;
```

Output

On the Flin Jobs page, locate your job, and click **More** > **FlinkUI** in the **Operation** column. On the displayed page, click **Task Managers** > **Stdout** to view the job output.



1.7.4 Built-In Functions

For details, see **Built-in Functions**.

1.7.4.1 Comparison Functions

Table 1-76 Comparison functions

SQL Function	Return Type	Description
value1 = value2	BOOLEAN	Returns **TRUE** if **value1** equals **value2**; returns **UNKNOWN** if either **value1** or **value2** is **NULL**.
value1 <> value2	BOOLEAN	Returns **TRUE** if **value1** does not equal **value2**; returns **UNKNOWN** if either **value1** or **value2** is **NULL**.
value1 > value2	BOOLEAN	Returns **TRUE** if **value1** is greater than **value2**; returns **UNKNOWN** if either **value1** or **value2** is **NULL**.

SQL Function	Return Type	Description
value1 >= value2	BOOLEAN	Returns **TRUE** if **value1** is greater than or equal to **value2**; returns **UNKNOWN** if either **value1** or **value2** is **NULL**.
value1 < value2	BOOLEAN	Returns **TRUE** if **value1** is less than **value2**; returns **UNKNOWN** if either **value1** or **value2** is **NULL**.
value1 <= value2	BOOLEAN	Returns **TRUE** if **value1** is less than or equal to **value2**; returns **UNKNOWN** if either **value1** or **value2** is **NULL**.
value IS NULL	BOOLEAN	Returns **TRUE** if **value** is **NULL**.
value IS NOT NULL	BOOLEAN	Returns **TRUE** if **value** is not **NULL**.
value1 IS DISTINCT FROM value2	BOOLEAN	Returns **TRUE** if **value1** and **value2** have different data types or values; returns **FALSE** if they have the same data types and values. Treats **NULL** as the same. For example: **1 IS DISTINCT FROM NULL** returns **TRUE**; **NULL IS DISTINCT FROM NULL** returns **FALSE**.
value1 IS NOT DISTINCT FROM value2	BOOLEAN	Returns **TRUE** if they have the same data types and values; returns **FALSE** if **value1** and **value2** have different data types or values. Treats **NULL** as the same. For example: **1 IS NOT DISTINCT FROM NULL** returns **FALSE**; **NULL IS NOT DISTINCT FROM NULL** returns **TRUE**.

SQL Function	Return Type	Description
value1 BETWEEN [ASYMMETRIC | SYMMETRIC] value2 AND value3	BOOLEAN	Returns **TRUE** if **value1** is greater than or equal to **value2** and less than or equal to **value3**, using the default or **ASYMMETRIC** keyword. If **SYMMETRIC** is used, returns **TRUE** if **value1** is inclusively between **value2** and **value3**. Returns **FALSE** or **UNKNOWN** if **value2** or **value3** is **NULL**. For example: ● **12 BETWEEN 15 AND 12** returns **FALSE**; ● **12 BETWEEN SYMMETRIC 15 AND 12** returns **TRUE**; ● **12 BETWEEN 10 AND NULL** returns **UNKNOWN**; ● **12 BETWEEN NULL AND 10** returns **FALSE**; ● **12 BETWEEN SYMMETRIC NULL AND 12** returns **UNKNOWN**.
value1 NOT BETWEEN [ASYMMETRIC | SYMMETRIC] value2 AND value3	BOOLEAN	Returns **TRUE** if **value1** is less than **value2** or greater than **value3**, using the default or **ASYMMETRIC** keyword. If **SYMMETRIC** is used, returns **TRUE** if **value1** is not between **value2** and **value3**. Returns **TRUE** or **UNKNOWN** if **value2** or **value3** is **NULL**. For example: ● **12 NOT BETWEEN 15 AND 12** returns **TRUE**; ● **12 NOT BETWEEN SYMMETRIC 15 AND 12** returns **FALSE**; ● **12 NOT BETWEEN NULL AND 15** returns **UNKNOWN**; ● **12 NOT BETWEEN 15 AND NULL** returns **TRUE**; ● **12 NOT BETWEEN SYMMETRIC 12 AND NULL** returns **UNKNOWN**.
string1 LIKE string2 [ESCAPE char]	BOOLEAN	Returns **TRUE** if **string1** matches **string2**; returns **UNKNOWN** if either **string1** or **string2** is **NULL**. Escape characters can be defined if needed, but they are not currently supported.

SQL Function	Return Type	Description
string1 NOT LIKE string2 [ESCAPE char]	BOOLEAN	Returns **TRUE** if **string1** does not match **string2**; returns **UNKNOWN** if either **string1** or **string2** is **NULL**. Escape characters can be defined if needed, but they are not currently supported.
string1 SIMILAR TO string2 [ESCAPE char]	BOOLEAN	Returns **TRUE** if **string1** matches the SQL regular expression **string2**; returns **UNKNOWN** if either **string1** or **string2** is **NULL**. Escape characters can be defined if needed, but they are not currently supported.
string1 NOT SIMILAR TO string2 [ESCAPE char]	BOOLEAN	Returns **TRUE** if **string1** does not match the SQL regular expression **string2**; returns **UNKNOWN** if either **string1** or **string2** is **NULL**. Escape characters can be defined if needed, but they are not currently supported.
value1 IN (value2 [, value3]*)	BOOLEAN	Returns **TRUE** if **value1** exists in the given list (value2, value3, …); returns **TRUE** if the list contains **NULL** and **value1** can be found, otherwise returns **UNKNOWN**. Always returns **UNKNOWN** if **value1** is **NULL**. For example: • **4 IN (1, 2, 3)** returns **FALSE**; • **1 IN (1, 2, NULL)** returns **TRUE**; • **4 IN (1, 2, NULL)** returns **UNKNOWN**.
value1 NOT IN (value2 [, value3]*)	BOOLEAN	Returns **TRUE** if **value1** does not exist in the given list (value2, value3, …); returns **FALSE** if the list contains **NULL** and **value1** can be found, otherwise returns **UNKNOWN**. Always returns **UNKNOWN** if **value1** is **NULL**. For example: • **4 NOT IN (1, 2, 3)** returns **TRUE**; • **1 NOT IN (1, 2, NULL)** returns **FALSE**; • **4 NOT IN (1, 2, NULL)** returns **UNKNOWN**.

SQL Function	Return Type	Description
EXISTS (sub-query)	BOOLEAN	Returns **TRUE** if the subquery returns at least one row. Only operations that can be overridden in join and grouping operations are supported. For streaming queries, this operation is rewritten in joins and grouping. The calculation of the query result required state may increase indefinitely based on the number of input rows. Provide a query configuration with effective retention intervals to prevent excessive state.
value IN (sub-query)	BOOLEAN	Returns **TRUE** if **value** is equal to one row in the subquery result set.
value NOT IN (sub-query)	BOOLEAN	Returns **TRUE** if **value** is not contained in the rows returned by the subquery.

1.7.4.2 Logical Functions

Table 1-77 Logical functions

SQL Function	Return Type	Description
boolean1 OR boolean2	BOOLEAN	Returns **TRUE** if **boolean1** or **boolean2** is **TRUE**. Supports three-valued logic. For example, **true || Null(BOOLEAN)** returns **TRUE**.
boolean1 AND boolean2	BOOLEAN	Returns **TRUE** if both **boolean1** and **boolean2** are **TRUE**. Supports three-valued logic. For example, **true && Null(BOOLEAN)** returns **UNKNOWN**.
NOT boolean	BOOLEAN	Returns **TRUE** if the **boolean** value is **FALSE**; returns **FALSE** if the **boolean** value is **TRUE**; returns **UNKNOWN** if the **boolean** value is **UNKNOWN**.
boolean IS FALSE	BOOLEAN	Returns **TRUE** if the **boolean** value is **FALSE**; returns **FALSE** if **boolean** is **TRUE** or **UNKNOWN**.
boolean IS NOT FALSE	BOOLEAN	Returns **TRUE** if **boolean** is **TRUE** or **UNKNOWN**; returns **FALSE** if **boolean** is **FALSE**.
boolean IS TRUE	BOOLEAN	Returns **TRUE** if **boolean** is **TRUE**; returns **FALSE** if **boolean** is **FALSE** or **UNKNOWN**.
boolean IS NOT TRUE	BOOLEAN	Returns **TRUE** if **boolean** is **FALSE** or **UNKNOWN**; returns **FALSE** if **boolean** is **TRUE**.

SQL Function	Return Type	Description
boolean IS UNKNOWN	BOOLEAN	Returns **TRUE** if the **boolean** value is **UNKNOWN**; returns **FALSE** if **boolean** is **TRUE** or **FALSE**.
boolean IS NOT UNKNOWN	BOOLEAN	Returns **TRUE** if **boolean** is **TRUE** or **FALSE**; returns **FALSE** if the **boolean** value is **UNKNOWN**.

1.7.4.3 Arithmetic Functions

Table 1-78 Arithmetic functions

Operator	Description
+ numeric	Returns a numeric.
- numeric	Returns the opposite of a numeric.
numeric1 + numeric2	Returns the sum of **numeric1** and **numeric2**.
numeric1 - numeric2	Returns the difference between **numeric1** and **numeric2**.
numeric1 * numberic2	Returns the product of **numeric1** and **numeric2**.
numeric1 / numeric2	Returns the quotient of **numeric1** divided by **numeric2**.
numeric1 % numeric2	Returns the remainder (modulus) of **numeric1** divided by **numeric2**. The result is negative only if **numeric1** is negative.
POWER(numeric1, numeric2)	Returns **numeric1** raised to the power of **numeric2**.
ABS(numeric)	Returns the absolute value of **numeric**.
SQRT(numeric)	Returns the square root of **numeric**.
LN(numeric)	Returns the natural logarithm (base e) of **numeric**.
LOG10(numeric)	Returns the logarithm (base 10) of **numeric**.
LOG2(numeric)	Returns the logarithm (base 2) of **numeric**.

Operator	Description
LOG(numeric2) LOG(numeric1, numeric2)	When called with one argument, returns the natural logarithm of **numeric2**. When called with two arguments, returns the logarithm of **numeric2** with base **numeric1**. **Numeric2** must be greater than 0 and **numeric1** must be greater than 1.
EXP(numeric)	Returns **e** raised to the power of **numeric**.
CEIL(numeric) CEILING(numeric)	Rounds up and returns the smallest integer greater than or equal to **numeric**.
FLOOR(numeric)	Rounds down and returns the largest integer less than or equal to **numeric**.
SIN(numeric)	Returns the sine of **numeric**.
SINH(numeric)	Returns the hyperbolic sine of **numeric**. The return type is **DOUBLE**.
COS(numeric)	Returns the tangent of **numeric**.
TAN(numeric)	Calculates the tangent of given A.
TANH(numeric)	Returns the hyperbolic tangent of **numeric**. The return type is **DOUBLE**.
COT(numeric)	Returns the cotangent of **numeric**.
ASIN(numeric)	Returns the inverse sine of **numeric**.
ACOS(numeric)	Returns the inverse cosine of **numeric**.
ATAN(numeric)	Returns the inverse tangent of **numeric**.
ATAN2(numeric1, numeric2)	Returns the inverse tangent of the coordinate (**numeric1**, **numeric2**).
COSH(numeric)	Returns the hyperbolic cosine of **numeric**. The return type is **DOUBLE**.
DEGREES(numeric)	Returns the degree representation of the radian **numeric**.
RADIANS(numeric)	Returns the radian representation of the degree **numeric**.
SIGN(numeric)	Returns the sign of **numeric**.
ROUND(numeric, INT)	Returns the value of **numeric** rounded to **INT** decimal places.
PI()	Returns a value very close to **pi**.
E()	Returns a value very close to **e**.

Operator	Description
RAND()	Returns a pseudo-random double-precision value within the range of [0.0, 1.0).
RAND(INT)	Returns a pseudo-random double-precision value within the range of [0.0, 1.0) with an initial seed of **INT**. If two RAND functions have the same initial seed, they will return the same sequence of numbers.
RAND_INTEGER(INT)	Returns a pseudo-random integer within the range of [0, INT).
RAND_INTEGER(INT1, INT2)	Returns a pseudo-random integer within the range of [0, INT2) with an initial seed of **INT1**. If two RAND_INTEGER functions have the same initial seed and boundary, they will return the same sequence of numbers.
UUID()	Returns a universally unique identifier (UUID) string based on RFC 4122 type 4 (pseudo-random generated). For example, **3d3c68f7-f608-473f-b60c-b0c44ad4cc4e** is generated using a cryptographically strong pseudo-random number generator.
BIN(INT)	Returns the string representation of **INTEGER** in binary format. If **INTEGER** is **NULL**, returns **NULL**. For example, **4.bin()** returns **"100"**, and **12.bin()** returns **"1100"**.
HEX(numeric) HEX(string)	Returns the string representation of the **numeric** value or **STRING** in hexadecimal format. If the parameter is **NULL**, returns **NULL**. For example, the number 20 returns **"14"**, the number 100 returns **"64"**, and the string "hello,world" returns **"68656C6C6F2C776F726C64"**.
TRUNCATE(numeric1, integer2)	Returns the number with **integer2** decimal places truncated. If **numeric1** or **integer2** is **NULL**, returns **NULL**. If **integer2** is **0**, the result has no decimal point or decimal part. **integer2** can be negative, making the **integer2** digits to the left of the decimal point zero. This function can also be called with only one **numeric1** parameter and without setting **integer2**. If **integer2** is not set, it defaults to **0**. For example, **42.324.truncate(2)** is **42.32**, and **42.324.truncate()** is **42.0**.

1.7.4.4 String Functions

Table 1-79 String functions

<table>
<tr><th>SQL Function</th><th>Description</th></tr>
<tr><td>string1 || string2</td><td>Returns the concatenation of STRING1 and STRING2.</td></tr>
<tr><td>CHAR_LENGTH(string)
CHARACTER_LENGTH(string)</td><td>Returns the number of characters in a string.</td></tr>
<tr><td>UPPER(string)</td><td>Returns a string in uppercase.</td></tr>
<tr><td>LOWER(string)</td><td>Returns a string in lowercase.</td></tr>
<tr><td>POSITION(string1 IN string2)</td><td>Returns the position (starting from 1) of the first occurrence of STRING1 in STRING2.
Returns 0 if STRING1 is not found in STRING2.</td></tr>
<tr><td>TRIM([BOTH | LEADING | TRAILING] string1 FROM string2)</td><td>Returns the result of removing the string that starts/ends/starts and ends with STRING2 from STRING1. By default, both sides' spaces will be removed.</td></tr>
<tr><td>LTRIM(string)</td><td>Returns the string with left spaces removed from STRING.
For example, ' This is a test String.'.ltrim() returns 'This is a test String.'.</td></tr>
<tr><td>RTRIM(string)</td><td>Returns the string with right spaces removed from STRING.
For example, 'This is a test String. '.ltrim() returns 'This is a test String.'.</td></tr>
<tr><td>REPEAT(string, int)</td><td>Returns a string that is the concatenation of INT number of strings.
For example, REPEAT('This is a test String.', 2) returns "This is a test String.This is a test String.".</td></tr>
<tr><td>REGEXP_REPLACE(string1, string2, string3)</td><td>Returns a string where all substrings in STRING1 that match the regular expression STRING2 are replaced with STRING3.
For example, 'foobar'.regexpReplace('oo|ar', '') returns "fb".</td></tr>
<tr><td>OVERLAY(string1 PLACING string2 FROM integer1 [FOR integer2])</td><td>Returns a string that replaces INT2 (STRING2's length by default) characters of STRING1 with STRING2 from position INT1.
For example, 'xxxxxtest'.overlay('xxxx', 6) returns "xxxxxxxxx", and 'xxxxxtest'.overlay('xxxx', 6, 2) returns "xxxxxxxxxst".</td></tr>
</table>

SQL Function	Description
SUBSTRING(string FROM integer1 [FOR integer2])	Returns a substring of **STRING** starting from position **INT1** with length **INT2** (default to the end).
REPLACE(string1, string2, string3)	Returns a new string where all occurrences of **STRING2** in **STRING1** are replaced with **STRING3** (non-overlapping). For example, **'hello world'.replace('world', 'flink')** returns **'hello flink'**; **'ababab'.replace('abab', 'z')** returns **'zab'**.
REGEXP_EXTRACT(string1, string2[, integer])	Splits the string **STRING1** according to the regular expression rule **STRING2** and returns the string at the specified position **INTEGER1**. The regular expression match group index starts at 1, with 0 indicating the entire regular expression match. In addition, the regular expression match group index should not exceed the defined number of groups. For example, **REGEXP_EXTRACT('foothebar', 'foo(.*?)(bar)', 2)** returns **"bar"**.
INITCAP(string)	Returns a new string where the first character of each word is capitalized and the rest are lowercase. Here, a word is defined as a sequence of alphanumeric characters.
CONCAT(string1, string2, ...)	Returns a string that concatenates string1, string2, ..., together. If any parameter is **NULL**, **NULL** is returned. For example, **CONCAT('AA', 'BB', 'CC')** returns **"AABBCC"**.
CONCAT_WS(string1, string2, string3, ...)	Returns a string that concatenates **STRING2**, **STRING3**, ..., together with the separator **STRING1**. A separator is added between each string to be concatenated. If **STRING1** is **NULL**, **NULL** is returned. Compared to **concat()**, **concat_ws()** automatically skips NULL parameters. For example, **concat_ws('~', 'AA', Null(STRING), 'BB', '', 'CC')** returns **"AA~BB~~CC"**.
LPAD(string1, integer, string2)	Returns a new string where **string2** is left-padded to the length of **INT**. If the length of **string1** is less than the value of **INT**, **string1** is returned shortened to an integer character. For example, **LPAD('hi', 4, '??')** returns **"??hi"**; **LPAD('hi', 1, '??')** returns **"h"**.

SQL Function	Description
RPAD(string1, integer, string2)	Returns a new string where **string2** is right-padded to the length of **INT**. If the length of **string1** is less than the value of **INT**, returns a new string where **string1** is shortened to a length of **INT**. For example, **RPAD('hi', 4, '??')** returns **"hi??"**; **RPAD('hi', 1, '??')** returns **"h"**.
FROM_BASE64(string)	Returns the result of decoding the base64-encoded **string1**. If the string is **NULL**, **NULL** is returned. For example, **FROM_BASE64('aGVsbG8gd29ybGQ=')** returns **"hello world"**.
TO_BASE64(string)	Returns the result of encoding the string to base64. If the string is **NULL**, **NULL** is returned. For example, **TO_BASE64('hello world')** returns **"aGVsbG8gd29ybGQ="**.
ASCII(string)	Returns the numeric value of the first character in the string. If the string is **NULL**, **NULL** is returned. For example, **ascii('abc')** returns **97**, and **ascii(CAST(NULL AS VARCHAR))** returns **NULL**.
CHR(integer)	Returns the ASCII character that corresponds to the binary value of the **integer**. If the integer is greater than 255, we first take the modulo of the integer with 255 and return the CHR of the modulo. If the integer is **NULL**, **NULL** is returned. For example, **chr(97) returns 'a', chr(353)** returns **'a'**, and **chr(CAST(NULL AS VARCHAR))** returns **NULL**.
DECODE(binary, string)	Decodes using the provided character set ('US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If any of the parameters are empty, the result will also be empty.
ENCODE(string1, string2)	Encodes using the provided character set ('US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If any of the parameters are empty, the result will also be empty.
INSTR(string1, string2)	Returns the position of the first occurrence of **string2** in **string1**. Returns **NULL** if the value of any parameter is **NULL**.
LEFT(string, integer)	Returns the leftmost substring of the string with a length equal to the **integer** value. If the **integer** is negative, an empty string is returned. Returns **NULL** if the value of any parameter is **NULL**.

SQL Function	Description
RIGHT(string, integer)	Returns the rightmost substring of the string with a length equal to the **integer** value. If the **integer** is negative, an empty string is returned. Returns **NULL** if the value of any parameter is **NULL**.
LOCATE(string1, string2[, integer])	Returns the position of the first occurrence of **string1** after position **integer** in **string2**. If not found, returns **0**. Returns **NULL** if either parameter is **NULL**.
PARSE_URL(string1, string2[, string3])	Returns a specified part from a URL. The valid values for **string2** include **"HOST"**, **"PATH"**, **"QUERY"**, **"REF"**, **"PROTOCOL"**, **"AUTHORITY"**, **"FILE"**, and **"USERINFO"**. Returns **NULL** if the value of any parameter is **NULL**. For example, **parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST')** returns **'facebook.com'**. You can also extract the value of a specific key in the QUERY by providing a keyword **string3** as the third parameter. For example, **parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1')** returns **'v1'**.
REGEXP(string1, string2)	Returns **TRUE** if any (possibly empty) substring of **string1** matches the Java regular expression **string2**, otherwise it returns **FALSE**. Returns **NULL** if the value of any parameter is **NULL**.
REVERSE(string)	Returns the reversed string. If the string is **NULL**, returns **NULL**.
SPLIT_INDEX(string1, string2, integer1)	Splits **string1** by the delimiter **string2** and returns the integer-th (starting from zero) split string. If the integer is negative, returns **NULL**. Returns **NULL** if the value of any parameter is **NULL**.
STR_TO_MAP(string1[, string2, string3])	Splits **string1** into key-value pairs using a separator and returns a map. **string2** is the pair separator, and the default separator is a comma (,). **string3** is the key-value separator, and the default separator is an equal sign (=). Both separators are regular expressions, so special characters should be escaped beforehand, such as **<([{\^-=$!|]})?*+.>**.
SUBSTR(string[, integer1[, integer2]])	Returns a substring of a string starting from position **integer1** with a length of **integer2** (default to the end).

SQL Function	Description
JSON_VAL(STRING json_string, STRING json_path)	Returns the value of the specified **json_path** from the **json_string**. For details about how to use the functions, see **JSON_VAL Function**. **NOTE** The following rules are listed in descending order of priority. 1. The two arguments **json_string** and **json_path** cannot be **NULL**. 2. The value of **json_string** must be a valid JSON string. Otherwise, the function returns **NULL**. 3. If **json_string** is an empty string, the function returns an empty string. 4. If **json_path** is an empty string or the path does not exist, the function returns **NULL**.

JSON_VAL Function

- Syntax

```
STRING JSON_VAL(STRING json_string, STRING json_path)
```

Table 1-80 Parameters

Parameter	Data Types	Description
json_string	STRING	JSON object to be parsed
json_path	STRING	Path expression for parsing the JSON string For the supported expressions, see **Table 1-81**.

Table 1-81 Expressions supported

Expression	Description
$	Root node in the path
[]	Access array elements
*	Array wildcard
.	Access child elements

- Example
  a. Test input data.
     Test the data source kafka. The message content is as follows:
     ```
     {"name":"James","age":24,"gender":"male","grade":{"math":95,"science":[80,85],"english":100}}
     ```
  b. Use JSON_VAL in SQL statements.

```
CREATE TABLE kafkaSource (
  message string
) WITH (
  'connector' = 'kafka',
  'topic-pattern' = '<yourSinkTopic>',
  'properties.bootstrap.servers' =
'<yourKafkaAddress1>:<yourKafkaPort>,<yourKafkaAddress2>:<yourKafkaPort>',
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'csv',
  'csv.field-delimiter' = '\u0001',
  'csv.quote-character' = ''''
);


CREATE TABLE printSink (
  message1 STRING,
  message2 STRING,
  message3 STRING,
  message4 STRING,
  message5 STRING,
  message6 STRING
) WITH (
  'connector' = 'print'
);
insert into printSink select
JSON_VAL(message,''),
JSON_VAL(message,'$.name'),
JSON_VAL(message,'$.grade.science'),
JSON_VAL(message,'$.grade.science[*]'),
JSON_VAL(message,'$.grade.science[1]'),
JSON_VAL(message,'$.grade.dddd')
from kafkaSource;
```

c. Check the output of the **out** file of the taskmanager.

```
+I[null, James, [80,85], [80,85], 85, null]
```

1.7.4.5 Temporal Functions

Table 1-82 lists the time functions supported by Flink OpenSource SQL.

Description

Table 1-82 Temporal Functions

Function	Return Type	Description
DATE string	DATE	Returns the SQL date parsed from a string in the format of "yyyy-MM-dd".
DATE_ADD	STRING	Resulting date after adding a certain number of days to a specified date. The data type is **STRING**.
DATE_SUB	STRING	Resulting date after subtracting a certain number of days from a specified date. The data type is **STRING**.
TIME string	TIME	Returns the SQL time parsed from a string in the format of "HH:mm:ss".

Function	Return Type	Description
TIMESTAMP string	TIMESTAMP	Returns the SQL timestamp parsed from a string in the format of "yyyy-MM-dd HH:mm:ss[.SSS]".
INTERVAL string range	INTERVAL	Parses the SQL millisecond interval from a string in the format of "dd hh:mm:ss.fff" or the SQL month interval from a string in the format of "yyyy-mm". The interval range can be **DAY**, **MINUTE**, **DAY TO HOUR**, or **DAY TO SECOND**, with the interval in milliseconds; **YEAR** or **YEAR TO MONTH** represents the interval in months. For example, **INTERVAL '10 00:00:00.004' DAY TO SECOND**, **INTERVAL '10' DAY**, or **INTERVAL '2-10' YEAR TO MONTH** returns the interval.
CURRENT_DATE	DATE	Returns the current SQL date in the local time zone. In streaming mode, it is evaluated for each record. In batch processing mode, it is evaluated once at the beginning of the query and the same result is used for each row.
CURRENT_TIME	TIME	Returns the current SQL time in the local time zone, which is a synonym for **LOCAL_TIME**.
CURRENT_TIMESTAMP	TIMESTAMP	Returns the current SQL timestamp in the local time zone, with the return type of **TIMESTAMP_LTZ(3)**. In streaming mode, it is evaluated for each record. In batch processing mode, it is evaluated once at the beginning of the query and the same result is used for each row.
LOCALTIME	TIME	Returns the current SQL time in the local time zone, with the return type of **TIME(0)**. In streaming mode, it is evaluated for each record. In batch processing mode, it is evaluated once at the beginning of the query and the same result is used for each row.

Function	Return Type	Description
LOCALTIMESTAMP	TIMESTAMP	Returns the current SQL timestamp in the local time zone, with the return type of **TIMESTAMP(3)**. In streaming mode, it is evaluated for each record. In batch processing mode, it is evaluated once at the beginning of the query and the same result is used for each row.
NOW()	TIMESTAMP	Returns the current SQL timestamp in the local time zone, which is a synonym for **CURRENT_TIMESTAMP**.
CURRENT_ROW_TIMESTAMP()	TIMESTAMP_LTZ(3)	Returns the current SQL timestamp in the local time zone, with the return type of **TIMESTAMP_LTZ(3)**. It is evaluated for each record, regardless of whether it is in batch processing mode or streaming mode.
EXTRACT(timeintervalunit FROM temporal)	BIGINT	Returns the long value extracted from the time interval unit part of the time. For example, **EXTRACT(DAY FROM DATE '2006-06-05')** returns **5**.
YEAR(date)	BIGINT	Returns the year from the SQL date, which is equivalent to **EXTRACT(YEAR FROM date)**. For example, **YEAR(DATE '1994-09-27')** returns **1994**.
QUARTER(date)	BIGINT	Returns the quarter of the year from the SQL date, which is an integer between 1 and 4, equivalent to **EXTRACT(QUARTER FROM date)**. For example, **QUARTER(DATE '1994-09-27')** returns **3**.
MONTH(date)	BIGINT	Returns the month of the year from the SQL date, which is an integer between 1 and 12, equivalent to **EXTRACT(MONTH FROM date)**. For example, **MONTH(DATE '1994-09-27')** returns **9**.
WEEK(date)	BIGINT	Returns the week of the year from the SQL date, which is an integer between 1 and 53, equivalent to **EXTRACT(WEEK FROM date)**. For example, **WEEK(DATE '1994-09-27')** returns **39**.

Function	Return Type	Description
DAYOFYEAR(date)	BIGINT	Returns the day of the year from the SQL date, which is an integer between 1 and 366, equivalent to **EXTRACT(DOY FROM date)**. For example, **DAYOFYEAR(DATE '1994-09-27')** returns **270**.
DAYOFMONTH(date)	BIGINT	Returns the day of the month from the SQL date, which is an integer between 1 and 31, equivalent to **EXTRACT(DAY FROM date)**. For example, **DAYOFWEEK(DATE '1994-09-27')** returns **3**.
DAYOFWEEK(date)	BIGINT	Calculates which day of the week the current date is, with Sunday being **1**. For example, **DAYOFWEEK(DATE'1994-09-27')** returns **3**.
HOUR(timestamp)	BIGINT	Returns the hour part of the hour unit from the SQL timestamp, which is an integer between 0 and 23, equivalent to **EXTRACT(HOUR FROM timestamp)**. For example, **MINUTE(TIMESTAMP '1994-09-27 13:14:15')** returns **14**.
MINUTE(timestamp)	BIGINT	Returns the minute part of the minute unit from the SQL timestamp, which is an integer between 0 and 59, equivalent to **EXTRACT(MINUTE FROM timestamp)**. For example, **MINUTE(TIMESTAMP '1994-09-27 13:14:15')** returns **14**.
SECOND(timestamp)	BIGINT	Returns the second part of the second unit from the SQL timestamp, which is an integer between 0 and 59, equivalent to **EXTRACT(SECOND FROM timestamp)**. For example, **SECOND(TIMESTAMP '1994-09-27 13:14:15')** returns **15**.
FLOOR(timepoint TO timeintervalunit)	TIME	Returns the value of **timepoint** rounded down to the time interval unit **timeintervalunit**. For example, **FLOOR(TIME '12:44:31' TO MINUTE)** returns **12:44:00**.
CEIL(timepoint TO timeintervalunit)	TIME	Return the value of **timepoint** rounded up to the time interval unit **timeintervalunit**. For example, **CEIL(TIME '12:44:31' TO MINUTE)** returns **12:45:00**.

Function	Return Type	Description
(timepoint1, temporal1) OVERLAPS (timepoint2, temporal2)	BOOLEAN	Returns **TRUE** if the two time intervals defined by (timepoint1, temporal1) and (timepoint2, temporal2) overlap. The time value can be a time point or a time interval. For example, **(TIME '2:55:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR)** returns **TRUE**; **(TIME '9:00:00', TIME '10:00:00') OVERLAPS (TIME '10:15:00', INTERVAL '3' HOUR)** returns **FALSE**.
DATE_FORMAT(timestamp, string)	STRING	Converts the timestamp to a string value in the specified date format string. The format string is compatible with Java's SimpleDateFormat.
TIMESTAMPADD(timeintervalunit, interval, timepoint)	TIMESTAMP/ DATE/ TIME	Adds the result of combining **interval** with **timeintervalunit** to a **timepoint** that includes a date or datetime, and returns the resulting datetime. For example, **TIMESTAMPADD(WEEK, 1, DATE '2003-01-02')** returns **2003-01-09**.
TIMESTAMPDIFF(timepointunit, timepoint1, timepoint2)	INT	Returns the time interval between **timepoint1** and **timepoint2**. The unit of the interval is given by the first parameter, which should be one of the following values: **SECOND**, **MINUTE**, **HOUR**, **DAY**, **MONTH**, or **YEAR**.
CONVERT_TZ(string1, string2, string3)	TIMESTAMP	Convert the datetime **string1** (with the default ISO timestamp format 'yyyy-MM-dd HH:mm:ss') from time zone **string2** to the value in time zone **string3**. The format of time zone should be either an abbreviation such as **PST**, a full name such as **Country A/City A**, or a custom ID such as **GMT-08:00**. For example, **CONVERT_TZ('1970-01-01 00:00:00', 'UTC', 'Country A/City A')** returns **'1969-12-31 16:00:00'**.

Function	Return Type	Description
FROM_UNIXTIME(numeric[, string])	STRING	Returns the representation of the numeric parameter **numberic** in the string format (default is **yyyy-MM-dd HH:mm:ss**). Numeric is an internal timestamp value that represents the number of seconds since '1970-01-01 00:00:00' UTC, generated by the **UNIX_TIMESTAMP()** function. The return value is represented in the session time zone (specified in TableConfig). For example, if in the UTC time zone, **FROM_UNIXTIME(44)** returns **1970-01-01 00:00:44**, and if in the Asia/Tokyo time zone, it returns **1970-01-01 09:00:44**.
UNIX_TIMESTAMP()	BIGINT	Gets the current Unix timestamp in seconds. This function is non-deterministic, meaning it will be recomputed for each record.
UNIX_TIMESTAMP(string1[, string2])	BIGINT	Converts the datetime **string1** in the format of **string2** (default is 'yyyy-MM-dd HH:mm:ss') to a Unix timestamp in seconds, using the time zone specified in the table configuration.
TO_DATE(string1[, string2])	DATE	Converts the **string1** in the format of **string2** (default is **yyyy-MM-dd**) to a date.
TO_TIMESTAMP_LTZ(numeric, precision)	TIMESTAMP_LTZ	Converts the epoch seconds or epoch milliseconds to **TIMESTAMP_LTZ**, with a valid precision of 0 or 3, where 0 represents **TO_TIMESTAMP_LTZ**(epochSeconds, 0) and 3 represents **TO_TIMESTAMP_LTZ**(epochMilliseconds, 3).
TO_TIMESTAMP(string1[, string2])	TIMESTAMP	Converts the **string1** in the format of **string2** (default is **yyyy-MM-dd HH:mm:ss**) in the UTC+0 time zone to a timestamp.

Function	Return Type	Description
CURRENT_WATERMARK(rowtime)	-	Returns the current watermark of the given time column attribute **rowtime**. If there is no common watermark available from upstream operations in the pipeline, the function returns **NULL**. The return type of the function is inferred to match the provided time column attribute, but with an adjusted precision of 3. For example, if the time column attribute is **TIMESTAMP_LTZ(9)**, the function returns **TIMESTAMP_LTZ(3)**. Note that this function can return **NULL**, which you may need to consider. For example, if you want to filter out late data, you can use: `WHERE CURRENT_WATERMARK(ts) IS NULL OR ts > CURRENT_WATERMARK(ts)`

DATE

- **Function**

  Returns a SQL date parsed from string in form of **yyyy-MM-dd**.

- **Description**

```
DATE DATE string
```

- **Input parameters**

Parameter	Data Type	Description
string	STRING	String in the SQL date format. Note that the string must be in the **yyyy-MM-dd** format. Otherwise, an error will be reported.

- **Example**
  - Test statement

```
SELECT
    DATE "2021-08-19" AS `result`
FROM
    testtable;
```

  - Test Result

result
2021-08-19

DATE_ADD

- **Function**

  Returns the resulting date after adding a certain number of days to a specified date.

- **Description**

```
DATE_ADD(string startdate, int days)
```

- **Input parameters**
  - **startdate**: specified date. The data type is **TIMESTAMP** or **STRING**.

    NOTE

    The date format for the **STRING** type is **yyyy-MM-dd HH:mm:ss**.

    The function supports the special case where this parameter value is **NULL**.
  - **days**: target number of days. The data type is **INT**.

- **Return values**

  Resulting date after adding a certain number of days to a specified date. The data type is **STRING**.

- **Example**

  Submit a Flink SQL statement.

```
CREATE TABLE source (
  time1 TIMESTAMP
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
create table Sink (
  date1 string,
  date2 string,
  date3 string
) with ('connector' = 'print');
INSERT into
  Sink
select
  DATE_ADD(time1, 30) as date1,
  DATE_ADD('2017-09-15 00:00:00', 30) as date2,
  DATE_ADD(cast(null as timestamp),30) as date3
FROM source
```

  Test result

date1 (string)	date2 (string)	date3 (string)
2024-06-28	2017-10-15	null

DATE_SUB

- **Function**

  Returns the resulting date after subtracting a certain number of days from a specified date.

- **Description**

```
DATE_SUB(string startdate, int days)
```

- **Input parameters**
  - **startdate**: specified date. The data type is **TIMESTAMP** or **STRING**.

NOTE

The date format for the **STRING** type is **yyyy-MM-dd HH:mm:ss**.

The function supports the special case where this parameter value is **NULL**.

  – **days**: target number of days. The data type is **INT**.

- **Return values**

  Resulting date after subtracting a certain number of days from a specified date. The data type is **STRING**.

- **Example**

  Submit a Flink SQL statement.

```
CREATE TABLE source (
  time1 TIMESTAMP
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
create table Sink (
  date1 string,
  date2 string,
  date3 string
) with ('connector' = 'print');
INSERT into
  Sink
select
  DATE_SUB(time1,30) as date1,
  DATE_SUB('2017-09-15 00:00:00', 30) as date2,
  DATE_SUB(cast(null as timestamp),30) as date3
FROM source
```

  Test result

date1 (string)	date2 (string)	date3 (string)
2024-04-29	2017-08-16	null

TIME

- **Function**

  Returns a SQL time parsed from string in form of **HH:mm:ss[.fff]**.

- **Description**

```
TIME TIME string
```

- **Input parameters**

Parameter	Data Type	Description
string	STRING	Time Note that the string must be in the format of **HH:mm:ss[.fff]**. Otherwise, an error will be reported.

- **Example**

  – Test statement

```
SELECT
    TIME "10:11:12" AS `result`,
        TIME "10:11:12.032" AS `result2`
FROM
    testtable;
```

– Test result

result	result2
10:11:12	10:11:12.032

TIMESTAMP

- **Function**

  Converts the time string into timestamp. The time string format is **yyyy-MM-dd HH:mm:ss[.fff]**. The return value is of the **TIMESTAMP(3)** type.

- **Description**

```
TIMESTAMP(3) TIMESTAMP string
```

- **Input parameters**

Parameter	Data Type	Description
string	STRING	Time Note that the string must be in the format of **yyyy-MM-dd HH:mm:ss[.fff]**. Otherwise, an error will be reported.

- **Example**
  – Test statement

```
SELECT
    TIMESTAMP "1997-04-25 13:14:15" AS `result`,
        TIMESTAMP "1997-04-25 13:14:15.032" AS `result2`
FROM
    testtable;
```

  – Test result

result	result2
1997-04-25 13:14:15	1997-04-25 13:14:15.032

INTERVAL

- **Function**

  Parses an interval string.

- **Description**

```
INTERVAL INTERVAL string range
```

- **Input parameters**

Parameter	Data Type	Description
string	STRING	Timestamp string used together with the **range** parameter. The string is in either of the following two formats: ● **yyyy-MM** for SQL intervals of months. An interval range might be **YEAR** or **YEAR TO MONTH** for intervals of months. ● **dd hh:mm:ss.fff** for SQL intervals of milliseconds. An interval range might be **DAY**, **MINUTE**, **DAY TO HOUR**, or **DAY TO SECOND**.
range	INTERVAL	Interval range. This parameter is used together with the **string** parameter. Available values are as follows: **YEAR, YEAR To Month, DAY, MINUTE, DAY TO HOUR and DAY TO SECOND**.

- **Example**

  Test statement

```
-- indicates that the interval is 10 days and 4 milliseconds.
INTERVAL '10 00:00:00.004' DAY TO second
-- The interval is 10 days.
INTERVAL '10'
-- The interval is 2 years and 10 months.
INTERVAL '2-10' YEAR TO MONTH
```

CURRENT_DATE

- **Function**

  Returns the current SQL time (**yyyy-MM-dd**) in the local time zone. The return value is of the **DATE** type.

- **Description**

```
DATE CURRENT_DATE
```

- **Input parameters**

  None

- **Example**
  - Test statement

```
SELECT
    CURRENT_DATE AS `result`
FROM
    testtable;
```

  - Test result

result
2021-10-28

CURRENT_TIME

- **Function**

  Returns the current SQL time (**HH:mm:sss.fff**) in the local time zone. The return value is of the **TIME** type.

- **Description**

```
TIME CURRENT_TIME
```

- **Input parameters**

  None

- **Example**
  - Test statement

```
SELECT
   CURRENT_TIME AS `result`
FROM
   testtable;
```

  - Test Result

result
08:29:19.289

CURRENT_TIMESTAMP

- **Function**

  Returns the current SQL timestamp in the local time zone. The return value is of the **TIMESTAMP(3)** type.

- **Description**

```
TIMESTAMP(3) CURRENT_TIMESTAMP
```

- **Input parameters**

  None

- **Example**
  - Test statement

```
SELECT
   CURRENT_TIMESTAMP AS `result`
FROM
   testtable;
```

  - Test Result

result
2021-10-28 08:33:51.606

LOCALTIME

- **Function**

  Returns the current SQL time in the local time zone. The return value is of the **TIME** type.

- **Description**

```
TIME LOCALTIME
```

- **Input parameters**

  None

- **Example**
  - Test statement

```
SELECT
    LOCALTIME AS `result`
FROM
    testtable;
```

  - Test Result

result
16:39:37.706

LOCALTIMESTAMP

- **Function**

  Returns the current SQL timestamp in the local time zone. The return value is of the **TIMESTAMP(3)** type.

- **Description**

```
TIMESTAMP(3) LOCALTIMESTAMP
```

- **Input parameters**

  None

- **Example**
  - Test statement

```
SELECT
    LOCALTIMESTAMP AS `result`
FROM
    testtable;
```

  - Test Result

result
2021-10-28 16:43:17.625

EXTRACT

- **Function**

  Returns a value extracted from the **timeintervalunit** part of temporal. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT EXTRACT(timeinteravlunit FROM temporal)
```

- **Input parameters**

Parameter	Data Type	Description
timeinteravlunit	TIMEUNIT	Time unit to be extracted from a time point or interval. The value can be **YEAR**, **QUARTER**, **MONTH**, **WEEK**, **DAY**, **DOY**, **HOUR**, **MINUTE**, **SECOND**.

Parameter	Data Type	Description
temporal	DATE/TIME/ TIMESTAMP/ INTERVAL	Time point or interval

CAUTION

Do not specify a time unit that is not of any time points or intervals. Otherwise, the job fails to be submitted.

For example, an error message is displayed when the following statement is executed because **YEAR** cannot be extracted from **TIME**.

```
SELECT
   EXTRACT(YEAR FROM TIME '12:44:31' ) AS `result`
FROM
   testtable;
```

- **Example**
  - Test statement

```
SELECT
   EXTRACT(YEAR FROM DATE '1997-04-25' ) AS `result`,
      EXTRACT(MINUTE FROM TIME '12:44:31') AS `result2`,
      EXTRACT(SECOND FROM TIMESTAMP '1997-04-25 13:14:15') AS `result3`,
      EXTRACT(YEAR FROM INTERVAL '2-10' YEAR TO MONTH) AS `result4`,
FROM
   testtable;
```

  - Test result

result	result2	result3	result4
1997	44	15	2

YEAR

- **Function**

  Returns the year from a SQL date date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT YEAR(date)
```

- **Input parameters**

Parameter	Data Type	Description
date	DATE	SQL date

- **Example**
  - Test statement

```
SELECT
   YEAR(DATE '1997-04-25' ) AS `result`
FROM
   testtable;
```

– Test result

result
1997

QUARTER

- **Function**

  Returns the quarter of a year (an integer between 1 and 4) from a SQL date date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT QUARTER(date)
```

- **Input parameters**

Parameter	Data Type	Description
date	DATE	SQL date

- **Example**
  – Test statement

```
SELECT
   QUARTER(DATE '1997-04-25' ) AS `result`
FROM
   testtable;
```

  – Test result

result
2

MONTH

- **Function**

  Returns the month of a year (an integer between 1 and 12) from a SQL date date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT MONTH(date)
```

- **Input parameters**

Parameter	Data Type	Description
date	DATE	SQL date

- **Example**
  – Test statement

```
SELECT
   MONTH(DATE '1997-04-25' ) AS `result`
FROM
   testtable;
```


- Test result

result
4

WEEK

- **Function**

  Returns the week of a year from a SQL date date. The return value is of the **BIGINT** type.
- **Description**
  ```
  BIGINT WEEK(date)
  ```
- **Input parameters**

Parameter	Data Type	Description
date	DATE	SQL date

- **Example**
  - Test statement
    ```
    SELECT
        WEEK(DATE '1997-04-25' ) AS `result`
    FROM
        testtable;
    ```
  - Test result

result
17

DAYOFYEAR

- **Function**

  Returns the day of a year (an integer between 1 and 366) from SQL date date. The return value is of the **BIGINT** type.
- **Description**
  ```
  BIGINT DAYOFYEAR(date)
  ```
- **Input parameters**

Parameter	Data Type	Description
date	DATE	SQL date

- **Example**
  - Test statement
    ```
    SELECT
        DAYOFYEAR(DATE '1997-04-25' ) AS `result`
    FROM
        testtable;
    ```

– Test Result

result
115

DAYOFMONTH

- **Function**
  Returns the day of a month (an integer between 1 and 31) from a SQL date date. The return value is of the **BIGINT** type.
- **Description**
  ```
  BIGINT DAYOFMONTH(date)
  ```
- **Input parameters**

Parameter	Data Type	Description
date	DATE	SQL date

- **Example**
  – Test statement
  ```
  SELECT
      DAYOFMONTH(DATE '1997-04-25' ) AS `result`
  FROM
      testtable;
  ```
  – Test Result

result
25

DAYOFWEEK

- **Function**
  Returns the day of a week (an integer between 1 and 7) from a SQL date date. The return value is of the **BIGINT** type.

  NOTE

  Note that the start day of a week is Sunday.
- **Description**
  ```
  BIGINT DAYOFWEEK(date)
  ```
- **Input parameters**

Parameter	Data Type	Description
date	DATE	SQL date

- **Example**
  – Test statement

```
SELECT
    DAYOFWEEK(DATE '1997-04-25') AS `result`
FROM
    testtable;
```

   – Test Result

result
6

HOUR

- **Function**

  Returns the hour of a day (an integer between 0 and 23) from SQL timestamp timestamp. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT HOUR(timestamp)
```

- **Input parameters**

Parameter	Data Type	Description
timestamp	TIMESTAMP	SQL timestamp

- **Example**
  – Test statement

```
SELECT
    HOUR(TIMESTAMP '1997-04-25 10:11:12') AS `result`
FROM
    testtable;
```

   – Test Result

result
10

MINUTE

- **Function**

  Returns the minute of an hour (an integer between 0 and 59) from a SQL timestamp. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT MINUTE(timestamp)
```

- **Input parameters**

Parameter	Data Type	Description
timestamp	TIMESTAMP	SQL timestamp

- **Example**
  – Test statement

```
SELECT
    MINUTE(TIMESTAMP '1997-04-25 10:11:12') AS `result`
FROM
    testtable;
```

– Test Result

result
11

SECOND

- **Function**

  Returns the second of an hour (an integer between 0 and 59) from a SQL timestamp. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT SECOND(timestamp)
```

- **Input parameters**

Parameter	Data Type	Description
timestamp	TIMESTAMP	SQL timestamp

- **Example**
  – Test statement

```
SELECT
    SECOND(TIMESTAMP '1997-04-25 10:11:12') AS `result`
FROM
    testtable;
```

  – Test result

result
12

FLOOR

- **Function**

  Returns a value that rounds **timepoint** down to the time unit **timeintervalunit**.

- **Description**

```
TIME/TIMESTAMP(3) FLOOR(timepoint TO timeintervalunit)
```

- **Input parameters**

Parameter	Data Type	Description
timepoint	TIMESTAMP /TIME	SQL time or SQL timestamp

Parameter	Data Type	Description
timeintervalunit	TIMEUNIT	Time unit. The value can be **YEAR**, **QUARTER**, **MONTH**, **WEEK**, **DAY**, **DOY**, **HOUR**, **MINUTE**, or **SECOND**.

- **Example**
  - Test statement
```
SELECT
    FLOOR(TIME '13:14:15' TO MINUTE) AS `result`
        FLOOR(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result2`,
        FLOOR(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result3`
FROM    testtable;
```
  - Test result

message	message2	message3
13:14	13:14	1997-04-25T13:14

CEIL

- **Function**

  Returns a value that rounds **timepoint** up to the time unit **timeintervalunit**.
- **Description**
```
TIME/TIMESTAMP(3) CEIL(timepoint TO timeintervalunit)
```
- **Input parameters**

Parameter	Data Type	Description
timepoint	TIMESTAMP /TIME	SQL time or SQL timestamp
timeintervalunit	TIMEUNIT	Time unit. The value can be **YEAR**, **QUARTER**, **MONTH**, **WEEK**, **DAY**, **DOY**, **HOUR**, **MINUTE**, or **SECOND**.

- **Example**
  - Test statement
```
SELECT
    CEIL(TIME '13:14:15' TO MINUTE) AS `result`
        CEIL(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result2`,
        CEIL(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result3`
FROM    testtable;
```
  - Test Result

result	result2	result3
13:15	13:15	1997-04-25T13:15

OVERLAPS

- **Function**

  Returns **TRUE** if two time intervals overlap; returns **FALSE** otherwise.

- **Description**
  ```
  BOOLEAN (timepoint1, temporal1) OVERLAPS (timepoint2, temporal2)
  ```
- **Input parameters**

Parameter	Data Type	Description
timepoint1/ timepoint2	DATE/TIME/ TIMESTAMP	Time point
temporal1/ temporal2	DATE/TIME/ TIMESTAMP/ INTERVAL	Time point or interval

NOTE

- **(timepoint, temporal)** is a closed interval.
- The temporal can be of the **DATE**, **TIME**, **TIMESTAMP**, or **INTERVAL** type.
  - When th temporal is **DATE**, **TIME**, or **TIMESTAMP**, **(timepoint, temporal)** indicates an interval between **timepoint** and **temporal**. The temporal can be earlier than the value of **timepoint**, for example, **(DATE '1997-04-25', DATE '1997-04-23')**.
  - When the temporal is **INTERVAL**, **(timepoint, temporal)** indicates an interval between **timepoint** and **timepoint + temporal**.
- Ensure that **(timepoint1, temporal1)** and **(timepoint2, temporal2)** are intervals of the same data type.

- **Example**
  - Test statement
  ```
  SELECT
     (TIME '2:55:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR) AS
  `result`,
        (TIME '2:30:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR) AS
  `result2`,
     (TIME '2:30:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:31:00', INTERVAL '2' HOUR) AS
  `result3`,
     (TIME '9:00:00', TIME '10:00:00') OVERLAPS (TIME '10:00:00', INTERVAL '3' HOUR) AS
  `result4`,
     (TIMESTAMP '1997-04-25 12:00:00', TIMESTAMP '1997-04-25 12:20:00') OVERLAPS
  (TIMESTAMP '1997-04-25 13:00:00', INTERVAL '2' HOUR) AS `result5`,
     (DATE '1997-04-23', INTERVAL '2' DAY) OVERLAPS (DATE '1997-04-25', INTERVAL '2' DAY)
  AS `result6`,
     (DATE '1997-04-25', DATE '1997-04-23') OVERLAPS (DATE '1997-04-25', INTERVAL '2' DAY)
  AS `result7`
  FROM
     testtable;
  ```
  - Test Result

result	result2	result3	result4	result5	result6	result7
true	true	false	true	false	true	true

DATE_FORMAT

- **Function**

  Converts a timestamp to a value of string in the format specified by the date format string.

- **Description**

```
STRING DATE_FORMAT(timestamp, dateformat)
```

- **Input parameters**

Parameter	Data Type	Description
timestamp	TIMESTAMP/STRING	Time point
dateformat	STRING	String in the date format

- **Example**
  - Test statement

```
SELECT
    DATE_FORMAT(TIMESTAMP '1997-04-25 10:11:12', 'yyyy-MM-dd HH:mm:ss') AS `result`,
        DATE_FORMAT(TIMESTAMP '1997-04-25 10:11:12', 'yyyy-MM-dd') AS `result2`,
    DATE_FORMAT(TIMESTAMP '1997-04-25 10:11:12', 'yy/MM/dd HH:mm') AS `result3`,
        DATE_FORMAT('1997-04-25 10:11:12', 'yyyy-MM-dd') AS `result4`
FROM    testtable;
```

  - Test Result

result	result2	result3	result4
1997-04-25 10:11:12	1997-04-25	97/04/25 10:11	1997-04-25

TIMESTAMPADD

- **Function**

  Returns the date and time by combining **interval** and **timeintervalunit** and adding the combination to **timepoint**.

  NOTE

  The return value of **TIMESTAMPADD** is the value of **timepoint**. An exception is that if the input **timepoint** is of the **TIMESTAMP** type, the return value can be inserted into a table field of the **DATE** type.

- **Description**

```
TIMESTAMP(3)/DATE/TIME TIMESTAMPADD(timeintervalunit, interval, timepoint)
```

- **Input parameters**

Parameter	Data Type	Description
timeintervalunit	TIMEUNIT	Time unit.
interval	INT	Interval
timepoint	TIMESTAMP/DATE/TIME	Time point

- **Example**
  - Test statement

```
SELECT
    TIMESTAMPADD(WEEK, 1, DATE '1997-04-25') AS `result`,
        TIMESTAMPADD(QUARTER, 1, TIMESTAMP '1997-04-25 10:11:12') AS `result2`,
    TIMESTAMPADD(SECOND, 2, TIME '10:11:12') AS `result3`
FROM    testtable;
```

  - Test Result

result	result2	result3
1997-05-02	• If this field is inserted into a table field of the **TIMESTAMP** type, **1997-07-25T10:11:12** is returned. • If this field is inserted into a table field of the TIMESTAMP type, 1997-07-25 is returned.	10:11:14

TIMESTAMPDIFF

- **Function**

  Returns the (signed) number of **timepointunit** between **timepoint1** and **timepoint2**. The unit for the interval is given by the first argument.

- **Description**

```
INT TIMESTAMPDIFF(timepointunit, timepoint1, timepoint2)
```

- **Input parameters**

Parameter	Data Type	Description
timepointunit	TIMEUNIT	Time unit. The value can be **SECOND**, **MINUTE**, **HOUR**, **DAY**, **MONTH** or **YEAR**.
timepoint1/timepoint2	TIMESTAMP/DATE	Time point

- **Example**
  - Test statement

```
SELECT
    TIMESTAMPDIFF(DAY, TIMESTAMP '1997-04-25 10:00:00', TIMESTAMP '1997-04-28 10:00:00')
AS `result`,
        TIMESTAMPDIFF(DAY, DATE '1997-04-25', DATE '1997-04-28') AS `result2`,
    TIMESTAMPDIFF(DAY, TIMESTAMP '1997-04-27 10:00:20', TIMESTAMP '1997-04-25 10:00:00')
AS `result3`
FROM     testtable;
```

  - Test result

result	result2	result3
3	3	-2

CONVERT_TZ

- **Function**

  Converts a datetime **string1** (with default ISO timestamp format **'yyyy-MM-dd HH:mm:ss'**) from time zone **string2** to time zone **string3**.

- **Description**

```
STRING CONVERT_TZ(string1, string2, string3)
```

- **Input parameters**

Parameter	Data Type	Description
string1	STRING	SQL timestamp. If the value does not meet the format requirements, **NULL** is returned.
string2	STRING	Time zone before conversion. The format of time zone should be either an abbreviation such as **PST**, a full name such as **Country A/City A**, or a custom ID such as **GMT-08:00**.
string3	STRING	Time zone after conversion. The format of time zone should be either an abbreviation such as **PST**, a full name such as **Country A/City A**, or a custom ID such as **GMT-08:00**.

- **Example**
  - Test statement

```
SELECT
    CONVERT_TZ(1970-01-01 00:00:00, UTC, Country A/City A) AS `result`,
        CONVERT_TZ(1997-04-25 10:00:00, UTC, GMT-08:00) AS `result2`
FROM     testtable;
```

  - Test Result

result	result2
1969-12-31 16:00:00	1997-04-25 02:00:00

FROM_UNIXTIME

- **Function**

  Returns a representation of the **numeric** argument as a value in string format.

- **Description**

```
STRING FROM_UNIXTIME(numeric[, string])
```

- **Input parameters**

Parameter	Data Type	Description
numeric	BIGINT	An internal timestamp representing the number of seconds since 1970-01-01 00:00:00 UTC. The value can be generated by the **UNIX_TIMESTAMP()** function.
string	STRING	Time. If this parameter is not specified, the default time format is **yyyy-MM-dd HH:mm:ss** format.

- **Example**
  - Test statement

```
SELECT
   FROM_UNIXTIME(44) AS `result`,
       FROM_UNIXTIME(44, 'yyyy:MM:dd') AS `result2`
FROM    testtable;
```

  - Test Result

result	result2
1970-01-01 08:00:44	1970:01:01

UNIX_TIMESTAMP

- **Function**

  Gets current Unix timestamp in seconds. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT UNIX_TIMESTAMP()
```

- **Input parameters**

  None

- **Example**
  - Test statement

```
SELECT
   UNIX_TIMESTAMP() AS `result`
FROM
   table;
```

  - Test result

result
1635401982

UNIX_TIMESTAMP(string1[, string2])

- **Function**

  Converts date time **string1** in format **string2** to Unix timestamp (in seconds). The return value is of the **BIGINT** type.

- **Description**

```
BIGINT UNIX_TIMESTAMP(string1[, string2])
```

- **Input parameters**

Parameter	Data Type	Description
string1	STRING	SQL timestamp string. An error is reported if the value does not comply with the **string2** format.
string2	STRING	Time. If this parameter is not specified, the default time format is **yyyy-MM-dd HH:mm:ss**.

- **Example**
  - Test statement

```
SELECT
   UNIX_TIMESTAMP('1997-04-25', 'yyyy-MM-dd') AS `result`,
      UNIX_TIMESTAMP('1997-04-25 00:00:10', 'yyyy-MM-dd HH:mm:ss') AS `result2`,
      UNIX_TIMESTAMP('1997-04-25 00:00:00') AS `result3`
FROM
   testtable;
```

  - Test result

result	result2	result3
861897600	861897610	861897600

TO_DATE

- **Function**

  Converts a date **string1** with format **string2** to a date.

- **Description**

```
DATE TO_DATE(string1[, string2])
```

- **Input parameters**

Parameter	Data Type	Description
string1	STRING	SQL timestamp string. If the value is not in the required format, an error is reported.

Parameter	Data Type	Description
string2	STRING	Format. If this parameter is not specified, the default time format is **yyyy-MM-dd**.

- **Example**
  - Test statement

```
SELECT
   TO_DATE('1997-04-25') AS `result`,
      TO_DATE('1997:04:25', 'yyyy-MM-dd') AS `result2`,
      TO_DATE('1997-04-25 00:00:00', 'yyyy-MM-dd HH:mm:ss') AS `result3`
FROM
   testtable;
```

  - Test result

result	result2	result3
1997-04-25	1997-04-25	1997-04-25

TO_TIMESTAMP

- **Function**

  Converts date time **string1** with format **string2** to a timestamp.

- **Description**

```
TIMESTAMP TO_TIMESTAMP(string1[, string2])
```

- **Input parameters**

Parameter	Data Type	Description
string1	STRING	SQL timestamp string. If the value is not in the required format, **NULL** is returned.
string2	STRING	Date format. If this parameter is not specified, the default format is **yyyy-MM-dd HH:mm:ss**.

- **Example**
  - Test statement

```
SELECT
   TO_TIMESTAMP('1997-04-25', 'yyyy-MM-dd') AS `result`,
      TO_TIMESTAMP('1997-04-25 00:00:00') AS `result2`,
      TO_TIMESTAMP('1997-04-25 00:00:00', 'yyyy-MM-dd HH:mm:ss') AS `result3`
FROM
   testtable;
```

  - Test result

result	result2	result3
1997-04-25 00:00	1997-04-25 00:00	1997-04-25 00:00

1.7.4.6 Conditional Functions

Description

Table 1-83 Conditional functions

Conditional Function	Description
CASE value WHEN value1_1 [, value1_2]* THEN result1 [WHEN value2_1 [, value2_2]* THEN result2]* [ELSE resultZ] END	Returns *resultX* when the first value in (valueX_1, valueX_2, ...) is included. If there is no matching value, returns *result_z* if provided, otherwise returns **NULL**.
CASE WHEN condition1 THEN result1 [WHEN condition2 THEN result2]* [ELSE resultZ] END	Returns *resultX* when the first condition X is met. If no conditions are met, returns *result_z* if provided, otherwise returns **NULL**.
NULLIF(value1, value2)	Returns **NULL** if *value1* equals *value2*; otherwise, returns *value1*. For example, **NULLIF(5, 5)** returns **NULL**; **NULLIF(5, 0)** returns **5**.
COALESCE(value1, value2 [, value3]*)	Returns the first non-NULL value from value1, value2, ... For example, **COALESCE(3, 5, 3)** returns **3**.
IF(condition, true_value, false_value)	Returns **true_value** if the condition is met, otherwise returns **false_value**. For example, **IF(5 > 3, 5, 3)** returns **5**.
IFNULL(input, null_replacement)	Returns **null_replacement** if the input is **NULL**; otherwise, returns the input. This function returns a data type that is very clear about whether it is empty or not compared to **COALESCE** or **CASE WHEN**. The returned type is the common type of the two parameters, but can only be empty if **null_replacement** can be empty. This function allows nullable columns to be passed to functions or tables that use **NOT NULL** constraints. For example, **IFNULL(nullable_column, 5)** will never return **NULL**.

Conditional Function	Description
IS_ALPHA(string)	Returns **true** if all characters in the string are letters; otherwise, returns **false**.
IS_DECIMAL(string)	Returns **true** if the string can be parsed as a valid number; otherwise, returns **false**.
IS_DIGIT(string)	Returns **true** if all characters in the string are numbers; otherwise, returns **false**.
GREATEST(value1[, value2]*)	Returns the maximum value of all input parameters. If the input parameters contain **NULL**, returns **NULL**.
LEAST(value1[, value2]*)	Returns the minimum value of all input parameters. If the input parameters contain **NULL**, returns **NULL**.

1.7.4.7 Type Conversion Functions

Table 1-84 Type conversion functions

SQL Function	Description
CAST(value AS type)	Returns a new value that has been converted to the type. For example, **CAST('42' AS INT)** returns **42**; **CAST(NULL AS VARCHAR)** returns **NULL** of type **VARCHAR**.
TYPEOF(input) | TYPEOF(input, force_serializable)	Returns a string representation of the data type of the input expression. By default, the returned string is a summary string that may omit some details for readability. If **force_serializable** is set to **TRUE**, the string representation can preserve the full data type that is stored in the catalog. Note that anonymous inline data types do not have a serializable string representation, and in this case, **NULL** is returned.

CAST Syntax Format

```
CAST(value AS type)
```

CAST Syntax Description

This syntax is used to forcibly convert types.

CAST Caveats

If the input is **NULL**, **NULL** is returned.

CAST Example 1: Converting the Amount Value to an Integer

The following example converts the **amount** value to an integer.

```
insert into temp select cast(amount as INT) from source_stream;
```

Table 1-85 Examples of CAST type conversion functions

Example	Description	Example

cast(v1 as string)	Converts **v1** to a string. The value of **v1** can be of the numeric type or of the timestamp, date, or time type.	Table T1:
		`| content (INT) |`
		`| ------------- |`
		`| 5 |`
		Statement:
		`SELECT cast(content as varchar) FROM T1;`
		Result:
		`"5"`

cast (v1 as int)	Converts **v1** to the **int** type. The value of **v1** can be a number or a character.	Table T1:
		`| content (STRING) |`
		`| ------------- |`
		`| "5" |`
		Statement:
		`SELECT cast(content as int) FROM T1;`
		Result:
		`5`

cast(v1 as timestamp)	Converts **v1** to the **timestamp** type. The value of **v1** can be of the **string**, **date**, or **time** type.	Table T1:
		`| content (STRING) |`
		`| ------------- |`
		`| "2018-01-01 00:00:01" |`
		Statement:
		`SELECT cast(content as timestamp) FROM T1;`
		Result:
		`1514736001000`

Example	Description	Example

cast(v1 as date)	Converts **v1** to the **date** type. The value of **v1** can be of the **string** or **timestamp** type.	Table T1:
		`| content (TIMESTAMP) |`
		`| ------------- |`
		`| 1514736001000 |`
		Statement:
		`SELECT cast(content as date) FROM T1;`
		Result:
		`"2018-01-01"`

NOTE

Flink jobs do not support the conversion of **bigint** to **timestamp** using CAST. You can convert it using **to_timestamp**.

CAST Example 2

1. Create a Flink OpenSource SQL job by referring to **Kafka** and **Print**, enter the following job running script, and submit the job.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. Change the values of the parameters in bold in the following script according to the actual situation.

```
CREATE TABLE kafkaSource (
  cast_int_to_string int,
  cast_String_to_int string,
  case_string_to_timestamp string,
  case_timestamp_to_date timestamp
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  "format" = "json"
);

CREATE TABLE printSink (
  cast_int_to_string string,
  cast_String_to_int int,
  case_string_to_timestamp timestamp,
  case_timestamp_to_date date
) WITH (
  'connector' = 'print'
);

insert into printSink select
  cast(cast_int_to_string as string),
  cast(cast_String_to_int as int),
  cast(case_string_to_timestamp as timestamp),
  cast(case_timestamp_to_date as date)
from kafkaSource;
```

2. Connect to the Kafka cluster and send the following test data to the Kafka topic:

```
{"cast_int_to_string":"1", "cast_String_to_int": "1", "case_string_to_timestamp": "2022-04-02 15:00:00",
"case_timestamp_to_date": "2022-04-02 15:00:00"}
```


3. View output.
   - Method 1:
     i. Log in to the DLI management console and choose Job Management > Flink Streaming Jobs.
     ii. Locate the row that contains the target Flink job, and choose More & > FlinkUI in the Operation column.
     iii. On the Flink UI, choose Task Managers, click the task name, and select Stdout to view the job run logs.
   - Method 2: If you select **Save Job Log** on the **Running Parameters** tab before submitting the job, perform the following operations:
     i. Log in to the DLI management console and choose Job Management > Flink Streaming Jobs.
     ii. Click the name of the corresponding Flink job, choose Run Log, click OBS Bucket, and locate the folder of the corresponding log based on the job running date.
     iii. Go to the folder of the corresponding date, find the folder whose name contains taskmanager, download the taskmanager.out file, and view the result log.

   The query result is as follows:
```
+I(1,1,2022-04-02T15:00,2022-04-02)
```

1.7.4.8 Collection Functions

Description

Table 1-86 Collection functions

Collection Function	Description
CARDINALITY(array)	Returns the number of elements in the array.
array '[' INT ']'	Returns the element at the INT position in the array. The index starts at 1.
ELEMENT(array)	Returns the unique element in the array (with a base of 1); returns **NULL** if the array is empty; throws an exception if there are multiple elements in the array.
CARDINALITY(map)	Returns the number of entries in the map.
map '[' value ']'	Returns the value corresponding to the specified key in the map.

1.7.4.9 JSON Functions

JSON functions use JSON path expressions described in the SQL standard ISO/IEC TR 19075-6. Their syntax is inspired by ECMAScript and adopts many of its features, but is neither a subset nor a superset of it.

There are two types of path expressions: lax mode and strict mode. When omitted, it defaults to strict mode. Strict mode is intended to check data from a schema perspective and will throw an error when data does not conform to the path expression. However, functions like **JSON_VALUE** allow for defining fallback behavior when encountering errors. Lax mode, on the other hand, will convert errors to an empty sequence.

The special character $ represents the root node in a JSON path. Paths can access properties ($.a), array elements ($.a[0].b), or all elements in an array ($.a[*].b).

Known limitations: not all features of lax mode are currently supported correctly.

Table 1-87 JSON functions

SQL Function	Description

IS JSON [{ VALUE | SCALAR | ARRAY | OBJECT }]	Determines whether a given string is a valid JSON string.
	Specifying an optional type parameter will impose constraints on the allowed types of JSON objects. If the string is a valid JSON but not of that type, it returns **false**. The default value is **VALUE**.
	`-- TRUE`
	`'1' IS JSON`
	`'[]' IS JSON`
	`'{}' IS JSON`
	
	`-- TRUE`
	`'"abc"' IS JSON`
	`-- FALSE`
	`'abc' IS JSON`
	`NULL IS JSON`
	
	`-- TRUE`
	`'1' IS JSON SCALAR`
	`-- FALSE`
	`'1' IS JSON ARRAY`
	`-- FALSE`
	`'1' IS JSON OBJECT`
	
	`-- FALSE`
	`'{}' IS JSON SCALAR`
	`-- FALSE`
	`'{}' IS JSON ARRAY`
	`-- TRUE`
	`'{}' IS JSON OBJECT`

<table>
<tr><th>SQL Function</th><th>Description</th></tr>
<tr><td>JSON_EXISTS(jsonValue, path [{ TRUE | FALSE | UNKNOWN | ERROR } ON ERROR])</td><td>Determines whether a JSON string satisfies a given path search condition.
If error behavior is ignored, FALSE ON ERROR is the default value.<pre>
-- TRUE
SELECT JSON_EXISTS('{"a": true}', '$.a');
-- FALSE
SELECT JSON_EXISTS('{"a": true}', '$.b');
-- TRUE
SELECT JSON_EXISTS('{"a": [{ "b": 1 }]}',
  '$.a[0].b');

-- TRUE
SELECT JSON_EXISTS('{"a": true}',
  'strict $.b' TRUE ON ERROR);
-- FALSE
SELECT JSON_EXISTS('{"a": true}',
  'strict $.b' FALSE ON ERROR);
</pre></td></tr>
<tr><td>JSON_STRING(value)</td><td>Serializes a value to JSON.
This function returns a JSON string containing the serialized value. If the value is NULL, the function returns NULL.<pre>
-- NULL
JSON_STRING(CAST(NULL AS INT))

-- '1'
JSON_STRING(1)
-- 'true'
JSON_STRING(TRUE)
-- '"Hello, World!"'
JSON_STRING('Hello, World!')
-- '[1,2]'
JSON_STRING(ARRAY[1, 2])
</pre></td></tr>
</table>

<table>
<tr><th>SQL Function</th><th>Description</th></tr>
<tr><td>JSON_VALUE(jsonValue, path [RETURNING <dataType>] [{ NULL | ERROR | DEFAULT <defaultExpr> } ON EMPTY] [{ NULL | ERROR | DEFAULT <defaultExpr> } ON ERROR])</td><td>Extracts a scalar from a JSON string.

This method searches for the given path expression in the JSON string and returns the value if it is a scalar. If it is not a scalar value, it cannot be returned. By default, the value is returned as a STRING type. The **returnType** can be used to select a different type, supporting the following types:

VARCHAR/STRING

BOOLEAN

INTEGER

DOUBLE

For empty path expressions or errors, it can be defined to return null, throw an error, or return a defined default value. If omitted, the default value is **NULL ON EMPTY** or **NULL ON ERROR**. The default value can be a literal or an expression. If the default value itself causes an error, it will execute the error behavior of **ON EMPTY** and **ON ERROR**.

-- "true"
JSON_VALUE('{"a": true}', '$.a')

-- TRUE
JSON_VALUE('{"a": true}', '$.a' RETURNING BOOLEAN)

-- "false"
JSON_VALUE('{"a": true}', 'lax $.b'
    DEFAULT FALSE ON EMPTY)

-- "false"
JSON_VALUE('{"a": true}', 'strict $.b'
    DEFAULT FALSE ON ERROR)

-- 0.998D
JSON_VALUE('{"a.b": [0.998,0.996]}','$.["a.b"][0]'
    RETURNING DOUBLE)</td></tr>
</table>

<table>
<tr><th>SQL Function</th><th>Description</th></tr>
<tr><td>JSON_QUERY(jsonValue, path [{ WITHOUT | WITH CONDITIONAL | WITH UNCONDITIONAL } [ARRAY] WRAPPER] [{ NULL | EMPTY ARRAY | EMPTY OBJECT | ERROR } ON EMPTY] [{ NULL | EMPTY ARRAY | EMPTY OBJECT | ERROR } ON ERROR])</td><td>Extracts a JSON value from a JSON string.

The result is always returned as a STRING. The RETURNING clause is currently not supported.

The wrappingBehavior determines whether the extracted value should be wrapped in an array unconditionally or only if the value itself is not an array.

The onEmpty and onError determine the behavior when the path expression is empty or throws an error. By default, null is returned in both cases. Other options are to use an empty array, an empty object, or throw an error.<pre>
-- '{ "b": 1 }'
JSON_QUERY('{ "a": { "b": 1 } }', '$.a')
-- '[1, 2]'
JSON_QUERY('[1, 2]', '$')
-- NULL
JSON_QUERY(CAST(NULL AS STRING), '$')
-- '["c1","c2"]'
JSON_QUERY('{"a":[{"c":"c1"},{"c":"c2"}]}',
    'lax $.a[*].c')

-- Wrap result into an array
-- '[{}]'
JSON_QUERY('{}', '$' WITH CONDITIONAL ARRAY WRAPPER)
-- '[1, 2]'
JSON_QUERY('[1, 2]', '$' WITH CONDITIONAL ARRAY WRAPPER)
-- '[[1, 2]]'
JSON_QUERY('[1, 2]', '$' WITH UNCONDITIONAL ARRAY WRAPPER)

-- Scalars must be wrapped to be returned
-- NULL
JSON_QUERY(1, '$')
-- '[1]'
JSON_QUERY(1, '$' WITH CONDITIONAL ARRAY WRAPPER)

-- Behavior if path expression is empty / there is an error
-- '{}'
JSON_QUERY('{}', 'lax $.invalid' EMPTY OBJECT ON EMPTY)
-- '[]'
JSON_QUERY('{}', 'strict $.invalid' EMPTY ARRAY ON ERROR)
</pre></td></tr>
</table>

<table>
<tr><th>SQL Function</th><th>Description</th></tr>
<tr><td>JSON_OBJECT([[KEY] key VALUE value]* [{ NULL | ABSENT } ON NULL])</td><td>
Builds a JSON object string from a list of key-value pairs.
Note that the keys must be non-null string literals, while the values can be any expression.
The function returns a JSON string. The ON NULL behavior defines how to handle NULL values. If omitted, the default is NULL ON NULL.
Values created from another JSON constructor (JSON_OBJECT, JSON_ARRAY) will be inserted directly instead of being inserted as a string. This allows for building nested JSON structures.
<pre><code>-- '{}'
JSON_OBJECT()

-- '{"K1":"V1","K2":"V2"}'
JSON_OBJECT('K1' VALUE 'V1', 'K2' VALUE 'V2')

-- Expressions as values
JSON_OBJECT('orderNo' VALUE orders.orderId)

-- ON NULL
JSON_OBJECT(KEY 'K1' VALUE CAST(NULL AS STRING) NULL ON NULL)   --
'{"K1":null}'
JSON_OBJECT(KEY 'K1' VALUE CAST(NULL AS STRING) ABSENT ON NULL) -- '{}'

-- '{"K1":{"K2":"V"}}'
JSON_OBJECT(
  KEY 'K1'
  VALUE JSON_OBJECT(
    KEY 'K2'
    VALUE 'V'
)
)</code></pre>
</td></tr>
<tr><td>JSON_OBJECTAGG([KEY] key VALUE value [{ NULL | ABSENT } ON NULL])</td><td>
Builds a JSON object string by aggregating key-value expressions into a single JSON object.
The key expression must return a non-null string. The value expression can be anything, including other JSON functions. If the value is NULL, the ON NULL behavior defines what to do. If omitted, the default is NULL ON NULL.
Note that keys must be unique. If a key appears multiple times, an error will be thrown.
This feature is currently not supported in OVER windows.
<pre><code>-- '{"Apple":2,"Banana":17,"Orange":0}'
SELECT
  JSON_OBJECTAGG(KEY product VALUE cnt)
FROM orders</code></pre>
</td></tr>
</table>

<table>
<tr><th>SQL Function</th><th>Description</th></tr>
<tr><td>JSON_ARRAY([value]* [{ NULL | ABSENT } ON NULL])</td><td>Builds a JSON array string from a list of values.

The function returns a JSON string. These values can be any expression. The ON NULL behavior defines how to handle NULL values. If omitted, the default is ABSENT ON NULL.

Elements created from another JSON constructor (JSON_OBJECT, JSON_ARRAY) will be inserted directly instead of being inserted as a string. This allows for building nested JSON structures.<pre>-- '[]'
JSON_ARRAY()
-- '[1,"2"]'
JSON_ARRAY(1, '2')

-- Expressions as values
JSON_ARRAY(orders.orderId)

-- ON NULL
JSON_ARRAY(CAST(NULL AS STRING) NULL ON NULL) -- '[null]'
JSON_ARRAY(CAST(NULL AS STRING) ABSENT ON NULL) -- '[]'

-- '[[1]]'
JSON_ARRAY(JSON_ARRAY(1))</pre></td></tr>
<tr><td>JSON_ARRAYAGG(items [{ NULL | ABSENT } ON NULL])</td><td>Builds a JSON object string by aggregating elements into an array.

The element expression can be anything, including other JSON functions. If the value is NULL, the ON NULL behavior defines what to do. If omitted, the default is ABSENT ON NULL.

This feature is currently not supported in OVER windows, unbounded session windows, or hop windows.<pre>-- '["Apple","Banana","Orange"]'
SELECT
  JSON_ARRAYAGG(product)
FROM orders</pre></td></tr>
</table>

1.7.4.10 Value Construction Functions

Description

Table 1-88 Value construction functions

Value Construction Function	Description
-- implicit constructor with parenthesis (value1 [, value2]*)	Returns a row created from a list of values (value1, value2, ...). The implicit row constructor supports any expression as a field, but requires at least two fields. The explicit row constructor can handle any number of fields, but currently does not support all types of field expressions well.

Value Construction Function	Description
ARRAY '[' value1 [, value2]* ']'	Returns an array created from a list of values (value1, value2, ...).
MAP '[' value1, value2 [, value3, value4]* ']'	Returns a map created from a list of key-value pairs ((value1, value2), (value3, value4), ...).

1.7.4.11 Value Retrieval Functions

Table 1-89 Value retrieval functions

SQL Function	Description
tableName.compositeType.field	Returns the value of a field from a Flink composite type (e.g. Tuple, POJO) by name.
tableName.compositeType.*	Returns the flattened representation of a Flink composite type (e.g. Tuple, POJO), converting each of its immediate subtypes into a separate field. In most cases, the fields in the flattened representation have similar names to the original fields, but with a $ separator (e.g. mypojo$mytuple$f0).

1.7.4.12 Grouping Functions

Table 1-90 Grouping functions

SQL Function	Description
GROUP_ID()	Returns an integer that uniquely identifies the combination of grouping keys.
GROUPING(expression1 [, expression2]*) | GROUPING_ID(expression1 [, expression2]*)	Returns a bit vector for the given grouping expression.

1.7.4.13 Hash Functions

Table 1-91 Hash functions

Hash Function	Description
MD5(string)	Returns the MD5 hash value of the string as a 32-digit hexadecimal string and returns **NULL** if the string is **NULL**.
SHA1(string)	Returns the SHA-1 hash value of the string as a 40-digit hexadecimal string and returns **NULL** if the string is **NULL**.
SHA224(string)	Returns the SHA-224 hash value of the string as a 56-digit hexadecimal string and returns **NULL** if the string is **NULL**.
SHA256(string)	Returns the SHA-256 hash value of the string as a 64-digit hexadecimal string and returns **NULL** if the string is **NULL**.
SHA384(string)	Returns the SHA-384 hash value of the string as a 96-digit hexadecimal string and returns **NULL** if the string is **NULL**.
SHA512(string)	Returns the SHA-512 hash value of the string as a 128-digit hexadecimal string and returns **NULL** if the string is **NULL**.
SHA2(string, hashLength)	Uses the SHA-2 series hash function (SHA-224, SHA-256, SHA-384, or SHA-512) to return the hash value. The first parameter is the string to be hashed, and the second parameter **hashLength** is the length of the result in bits (224, 256, 384, or 512). Returns **NULL** if **string** or **hashLength** is **NULL**.

1.7.4.14 Aggregate Functions

Aggregate functions process all rows as input and produce a single aggregate value as the output.

Table 1-92 Aggregate functions

Function	Description
COUNT([ALL] expression | DISTINCT expression1 [, expression2]*)	By default or with the keyword **ALL**, returns the number of input rows where the expression is not **NULL**. Using **DISTINCT** calculates the count after removing duplicates.
COUNT(*) | COUNT(1)	Returns the number of input rows.

Function	Description
AVG([ALL | DISTINCT] expression)	By default or with the keyword **ALL**, returns the average value (arithmetic mean) of the expression across all input rows. Using **DISTINCT** calculates the average after removing duplicates.
SUM([ALL | DISTINCT] expression)	By default or with the keyword **ALL**, returns the sum of the expression across all input rows. Using **DISTINCT** calculates the sum after removing duplicates.
MAX([ALL | DISTINCT] expression)	By default or with the keyword **ALL**, returns the maximum value of the expression across all input rows. Using **DISTINCT** calculates the maximum after removing duplicates.
MIN([ALL | DISTINCT] expression)	By default or with the keyword **ALL**, returns the minimum value of the expression across all input rows. Using **DISTINCT** calculates the minimum after removing duplicates.
STDDEV_POP([ALL | DISTINCT] expression)	By default or with the keyword **ALL**, returns the population standard deviation of the expression across all input rows. Using **DISTINCT** calculates the standard deviation after removing duplicates.
STDDEV_SAMP([ALL | DISTINCT] expression)	By default or with the keyword **ALL**, returns the sample standard deviation of the expression across all input rows. Using **DISTINCT** calculates the standard deviation after removing duplicates.
VAR_POP([ALL | DISTINCT] expression)	By default or with the keyword **ALL**, returns the population variance (square of the population standard deviation) of the expression across all input rows. Using **DISTINCT** calculates the variance after removing duplicates.
VAR_SAMP([ALL | DISTINCT] expression)	By default or with the keyword **ALL**, returns the sample variance (square of the sample standard deviation) of the expression across all input rows. Using **DISTINCT** calculates the variance after removing duplicates.
COLLECT([ALL | DISTINCT] expression)	By default or with the keyword **ALL**, returns multiple sets of expressions across all input rows. NULL values are ignored. Using **DISTINCT** calculates the sets after removing duplicates.
VARIANCE([ALL | DISTINCT] expression)	A synonym for **VAR_SAMP()**.

Function	Description
RANK()	Returns the rank of a value within a set of values. The result is 1 plus the number of rows preceding or equal to the current row in the ordering of the partition. The rank may not be consecutive in the sequence.
DENSE_RANK()	Returns the rank of a value within a set of values. The result is one plus the previously assigned rank value. Unlike the rank function, dense_rank does not leave gaps in the ranking sequence.
ROW_NUMBER()	Assigns a unique sequential number to each row within a window partition based on the ordering of rows by rows. ROW_NUMBER is similar to RANK. ROW_NUMBER numbers all rows sequentially (for example, 1, 2, 3, 4, 5). RANK provides the same sequence value for equal rows (for example, 1, 2, 2, 4, 5).
LEAD(expression [, offset] [, default])	Returns the value of the expression at the offset-th row after the current row in the window. The default value of **offset** is **1**, and the default value of **default** is **NULL**.
LAG(expression [, offset] [, default])	Returns the value of the expression at the offset-th row before the current row in the window. The default value of **offset** is **1**, and the default value of **default** is **NULL**.
FIRST_VALUE(expression)	Returns the first value in a set of ordered values.
LAST_VALUE(expression)	Returns the last value in a set of ordered values.
LISTAGG(expression [, separator])	Concatenates the values of string expressions and places a separator value between them. The default separator value is **,** if no separator is added at the end of the string.

1.7.4.15 Table-Valued Functions

1.7.4.15.1 string_split

The **string_split** function splits a target string into substrings based on the specified separator and returns a substring list.

Description

```
string_split(target, separator)
```

Table 1-93 string_split parameters

Parameter	Data Types	Description
target	STRING	Target string to be processed **NOTE** • If **target** is **NULL**, an empty line is returned. • If **target** contains two or more consecutive separators, an empty substring is returned. • If **target** does not contain a specified separator, the original string passed to **target** is returned.
separator	VARCHAR	Separator. Currently, only single-character separators are supported.

Example

1. Create a Flink OpenSource SQL job by referring to **Kafka** and **Print**, enter the following job running script, and submit the job.

   When you create a job, set **Flink Version** to **1.15** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE kafkaSource (
  target STRING,
  separator  VARCHAR
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE printSink (
  target STRING,
  item STRING
) WITH (
  'connector' = 'print'
);
insert into printSink select target,  item from kafkaSource, lateral table(string_split(target, separator))
as T(item);
```

2. Connect to the Kafka cluster and send the following test data to the Kafka topic:

```
{"target":"test-flink","separator":"-"}
{"target":"flink","separator":"-"}
{"target":"one-two-ww-three","separator":"-"}
```

   The data is as follows:

Table 1-94 Test table data

target (STRING)	separator (VARCHAR)
test-flink	-
flink	-
one-two-ww-three	-

3. View output.
   - Method 1:
     i. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
     ii. Locate the row that contains the target Flink job, and choose **More** > **FlinkUI** in the **Operation** column.
     iii. On the Flink UI, choose **Task Managers**, click the task name, and select **Stdout** to view job logs.
   - Method 2: If you select **Save Job Log** on the **Running Parameters** tab before submitting the job, perform the following operations:
     i. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
     ii. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
     iii. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

The query result is as follows:

```
+I(test-flink,test)
+I(test-flink,flink)
+I(flink,flink)
+I(one-two-ww-three,one)
+I(one-two-ww-three,two)
+I(one-two-ww-three,ww)
+I(one-two-ww-three,three)
```

The output data is as follows:

Table 1-95 Result table data

target (STRING)	item (STRING)
test-flink	test
test-flink	flink
flink	flink
one-two-ww-three	one
one-two-ww-three	two
one-two-ww-three	ww

target (STRING)	item (STRING)
one-two-ww-three	three

2 Flink OpenSource SQL 1.12 Syntax Reference

2.1 Constraints and Definitions

2.1.1 Supported Data Types

The DLI SQL syntax supports the following data types:

STRING, BOOLEAN, BYTES, DECIMAL, TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DATE, TIME, TIMESTAMP, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL, ARRAY, MULTISET, MAP, ROW

In the SQL syntax, these types are used to define the data types of columns within a table.

2.1.2 Syntax

2.1.2.1 Data Definition Language (DDL)

2.1.2.1.1 CREATE TABLE

Syntax

```
CREATE TABLE table_name
  (
    { <column_definition> | <computed_column_definition> }[ , ...n]
    [ <watermark_definition> ]
    [ <table_constraint> ][ , ...n]
  )
  [COMMENT table_comment]
  [PARTITIONED BY (partition_column_name1, partition_column_name2, ...)]
  WITH (key1=val1, key2=val2, ...)

<column_definition>:
  column_name column_type [ <column_constraint> ] [COMMENT column_comment]

<column_constraint>:
```

```
  [CONSTRAINT constraint_name] PRIMARY KEY NOT ENFORCED

<table_constraint>:
  [CONSTRAINT constraint_name] PRIMARY KEY (column_name, ...) NOT ENFORCED

<computed_column_definition>:
  column_name AS computed_column_expression [COMMENT column_comment]

<watermark_definition>:
  WATERMARK FOR rowtime_column_name AS watermark_strategy_expression

<source_table>:
  [catalog_name.][db_name.]table_name
```

Function

Create a table with a specified name.

Description

COMPUTED COLUMN

A computed column is a virtual column generated using **column_name AS computed_column_expression**. A computed column evaluates an expression that can reference other columns declared in the same table. The column itself is not physically stored within the table. A computed column could be defined using **cost AS price * quantity**. This expression can contain any combination of physical columns, constants, functions, or variables, but cannot contain any subquery.

In Flink, a computed column is used to define the time attribute in **CREATE TABLE** statements. A processing time attribute can be defined easily via **proc AS PROCTIME()** using the system's **PROCTIME()** function. The event time column may be obtained from an existing field. In this case, you can use the computed column to obtain event time. For example, if the original field is not of the **TIMESTAMP(3)** type or is nested in a JSON string, you can use computed columns.

Note:

- An expression that defines a computed column in a source table is calculated after data is read from the data source. The column can be used in the **SELECT** statement.
- A computed column cannot be the target of an **INSERT** statement. In an **INSERT** statement, the schema of the **SELECT** statement must be the same as that of the target table that does not have a computed column.

WATERMARK

The **WATERMARK** clause defines the event time attribute of a table and takes the form **WATERMARK FOR rowtime_column_name AS watermark_strategy_expression**.

rowtime_column_name defines an existing column that is marked as the event time attribute of the table. The column must be of the **TIMESTAMP(3)** type and must be the top-level column in the schema. It can also be a computed column.

watermark_strategy_expression defines the watermark generation strategy. It allows arbitrary non-query expressions, including computed columns, to calculate the watermark. The expression return type must be **TIMESTAMP(3)**, which represents the timestamp since the Epoch. The returned watermark will be emitted only if it is non-null and its value is greater than the previously emitted local watermark (to preserve the contract of ascending watermarks). The watermark generation expression is evaluated by the framework for every record. The framework will periodically emit the largest generated watermark. If the current watermark is still identical to the previous one, or is null, or the value of the returned watermark is smaller than that of the last emitted one, then no new watermark will be emitted. A watermark is emitted in an interval defined by **pipeline.auto-watermark-interval**. If the watermark interval is 0 ms, a watermark will be emitted per record if it is not null and greater than the last emitted one.

When using event time semantics, tables must contain an event time attribute and watermark strategy.

Flink provides several commonly used watermark strategies.

- Strictly ascending timestamps: **WATERMARK FOR rowtime_column AS rowtime_column**

  Emits a watermark of the maximum observed timestamp so far. Rows that have a timestamp bigger than the maximum timestamp are not late.
- Ascending timestamps: **WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL '0.001' SECOND**

  Emits a watermark of the maximum observed timestamp so far minus 1. Rows that have a timestamp bigger than or equal to the maximum timestamp are not late.
- Bounded out-of-order timestamps: **WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL 'string' timeUnit**

  Emits a watermark, which is the maximum observed timestamp minus the specified delay, for example, **WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL '5' SECOND** is a 5-second delayed watermark strategy.

  ```
  CREATE TABLE Orders (
      user BIGINT,
      product STRING,
      order_time TIMESTAMP(3),
      WATERMARK FOR order_time AS order_time - INTERVAL '5' SECOND
  ) WITH ( . . . );
  ```

PRIMARY KEY

The primary key constraint is a hint for Flink to leverage for optimizations. It tells that a column or a set of columns of a table or a view are unique and they do not contain null. Neither of columns in a primary can be nullable. The primary key therefore uniquely identifies a row in a table.

The primary key constraint can be either declared along with a column definition (a column constraint) or as a single line (a table constraint). For both cases, it should only be declared as a singleton. If you define multiple primary key constraints at the same time, an exception would be thrown.

Validity Check

SQL standard specifies that a constraint can either be **ENFORCED** or **NOT ENFORCED**. This controls if the constraint checks are performed on the incoming/outgoing data. Flink does not own the data and therefore the only mode we want to support is the **NOT ENFORCED** mode. It is up to the user to ensure that the query enforces key integrity.

Flink will assume correctness of the primary key by assuming that the columns nullability is aligned with the columns in the primary key. Connectors should ensure those are aligned.

Note: In a **CREATE TABLE** statement, creating a primary key constraint will alter the columns nullability, which means, a column with a primary key constraint is not nullable.

PARTITIONED BY

Partition the created table by the specified columns. A directory is created for each partition if this table is used as a file system sink.

WITH OPTIONS

Table properties used to create a table source/sink. The properties are usually used to find and create the underlying connector.

The key and value of expression **key1=val1** should both be string literal.

Note: The table registered with the **CREATE TABLE** statement can be used as both the table source and table sink. We cannot decide if it is used as a source or sink until it is referenced in the DMLs.

2.1.2.1.2 CREATE VIEW

Syntax

```
CREATE VIEW [IF NOT EXISTS] view_name
  [{columnName [, columnName ]* }] [COMMENT view_comment]
  AS query_expression
```

Function

Create a view with multiple layers nested in it to simplify the development process.

Description

IF NOT EXISTS

If the view already exists, nothing happens.

Example

Create a view named **viewName**.

```
create view viewName as select * from dataSource
```

2.1.2.1.3 CREATE FUNCTION

Syntax

```
CREATE FUNCTION
  [IF NOT EXISTS] function_name
  AS identifier [LANGUAGE JAVA|SCALA]
```

Function

Create a user-defined function.

For details about how to create a user-defined function, see **User-Defined Functions (UDFs)**.

Description

IF NOT EXISTS

If the function already exists, nothing happens.

LANGUAGE JAVA|SCALA

The language tag is used to instruct Flink runtime how to execute the function. Currently, only **JAVA** and **SCALA** language tags are supported, the default language for a function is **JAVA**.

Example

Create a function named **STRINGBACK**.

```
create function STRINGBACK as 'com.dli.StringBack'
```

2.1.2.2 Data Manipulation Language (DML)

DML Statements

Syntax

```
INSERT INTO table_name [PARTITION part_spec] query

part_spec:  (part_col_name1=val1 [, part_col_name2=val2, ...])

query:
  values
  | {
      select
      | selectWithoutFrom
      | query UNION [ ALL ] query
      | query EXCEPT query
      | query INTERSECT query
    }
    [ ORDER BY orderItem [, orderItem ]* ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start { ROW | ROWS } ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY]

orderItem:
  expression [ ASC | DESC ]

select:
  SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
  [ WINDOW windowName AS windowSpec [, windowName AS windowSpec ]* ]

selectWithoutFrom:
  SELECT [ ALL | DISTINCT ]
```

```
 { * | projectItem [, projectItem ]* }

projectItem:
  expression [ [ AS ] columnAlias ]
  | tableAlias . *

tableExpression:
  tableReference [, tableReference ]*
  | tableExpression [ NATURAL ] [ LEFT | RIGHT | FULL ] JOIN tableExpression [ joinCondition ]

joinCondition:
  ON booleanExpression
  | USING '(' column [, column ]* ')'

tableReference:
  tablePrimary
  [ matchRecognize ]
  [ [ 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 ]* ')'

windowRef:
    windowName
  | windowSpec

windowSpec:
    [ windowName ]
    '('
    [ ORDER BY orderItem [, orderItem ]* ]
    [ PARTITION BY expression [, expression ]* ]
    [
        RANGE numericOrIntervalExpression {PRECEDING}
      | ROWS numericExpression {PRECEDING}
    ]
    ')'

matchRecognize:
      MATCH_RECOGNIZE '('
      [ PARTITION BY expression [, expression ]* ]
      [ ORDER BY orderItem [, orderItem ]* ]
      [ MEASURES measureColumn [, measureColumn ]* ]
      [ ONE ROW 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 ]*
      ')'

measureColumn:
```

```
    expression AS alias

pattern:
    patternTerm [ '|' patternTerm ]*

patternTerm:
    patternFactor [ patternFactor ]*

patternFactor:
    variable [ patternQuantifier ]

patternQuantifier:
    '*'
  |  '*?'
  |  '+'
  |  '+?'
  |  '?'
  |  '??'
  |  '{' { [ minRepeat ], [ maxRepeat ] } '}' ['?']
  |  '{' repeat '}'
```

Precautions

Flink SQL uses a lexical policy for identifier (table, attribute, function names) similar to Java:

- The case of identifiers is preserved whether they are quoted.
- Identifiers are matched case-sensitively.
- Unlike Java, back-ticks allow identifiers to contain non-alphanumeric characters (for example, **SELECT a AS `my field` FROM t**).

String literals must be enclosed in single quotes (for example, **SELECT'Hello World'**). Duplicate a single quote for escaping (for example, **SELECT 'It''s me.'**). Unicode characters are supported in string literals. If explicit Unicode points are required, use the following syntax:

- Use the backslash (\) as an escaping character (default): **SELECT U&'\263A'**
- Use a custom escaping character: **SELECT U&'#263A' UESCAPE '#'**

2.2 Overview

This section describes the Flink open source SQL 1.12 syntax supported by DLI. For details about the parameters and examples, see the syntax description.

Creating Tables

Table 2-1 Syntax for creating tables

Classification	Function
Creating a source table	**DataGen Source Table**
	GaussDB(DWS) Source Table
	HBase Source Table
	JDBC Source Table

Classification	Function
	Kafka Source Table
	MySQL CDC Source Table
	Postgres CDC Source Table
	Redis Source Table
	Upsert Kafka Source Table
Creating a result table	BlackHole Result Table
	ClickHouse Result Table
	GaussDB(DWS) Result Table
	Elasticsearch Result Table
	HBase Result Table
	JDBC Result Table
	Kafka Result Table
	Print Result Table
	Redis Result Table
	Upsert Kafka Result Table
Creating a dimension table	GaussDB(DWS) Dimension Table
	HBase Dimension Table
	JDBC Dimension Table
	Redis Dimension Table
Format	Avro
	Canal
	Confluent Avro
	CSV
	Debezium
	JSON
	Maxwell
	Raw

2.3 DDL Syntax

2.3.1 Creating Source Tables

2.3.1.1 DataGen Source Table

Function

DataGen is used to generate random data for debugging and testing.

Prerequisites

None

Precautions

- When you create a DataGen table, the table field type cannot be Array, Map, or Row. You can use **COMPUTED COLUMN** in **CREATE TABLE** to construct similar functions.
- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.

Syntax

```
create table dataGenSource(
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' WATERMARK FOR rowtime_column_name AS watermark-strategy_expression)
)
with (
  'connector' = 'datagen'
);
```

Parameters

Table 2-2 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **datagen**.
rows-per-second	No	10000	Long	Number of rows generated per second, which is used to control the emit rate.

Parameter	Mandatory	Default Value	Data Type	Description
fields.#.kind	No	random	String	Generator of the **#** field. The **#** field must be an actual field in the DataGen table. Replace **#** with the corresponding field name. The meanings of the **#** field for other parameters are the same. The value can be **sequence** or **random**. ● **random** is the default generator. You can use the **fields.#.max** and **fields.#.min** parameters to specify the maximum and minimum values that are randomly generated. If the specified field type is char, varchar, or string, you can also use the **fields.#.length** field to specify the length. A random generator is an unbounded generator. ● Sequence generator. You can use **fields.#.start** and **fields.#.end** to specify the start and end values of a sequence. A sequence generator is a bounded generator. When the sequence number reaches the end value, the reading ends.
fields.#.min	No	Minimum value of the field type specified by **#**	Field type specified by **#**	This parameter is valid only when **fields.#.kind** is set to **random**. Minimum value of the random generator. It applies only to numeric field types specified by **#**.
fields.#.max	No	Maximum value of the field type specified by **#**	Field type specified by **#**	This parameter is valid only when **fields.#.kind** is set to **random**. Maximum value of the random generator. It applies only to numeric field types specified by **#**.
fields.#.length	No	100	Integer	This parameter is valid only when **fields.#.kind** is set to **random**. Length of the characters generated by the random generator. It applies only to char, varchar, and string types specified by **#**.

Parameter	Mandatory	Default Value	Data Type	Description
fields.#.start	No	None	Field type specified by **#**	This parameter is valid only when **fields.#.kind** is set to **sequence**. Start value of a sequence generator.
fields.#.end	No	None	Field type specified by **#**	This parameter is valid only when **fields.#.kind** is set to **sequence**. End value of a sequence generator.

Example

Create a Flink OpenSource SQL job. Run the following script to generate random data through the DataGen table and output the data to the Print result table.

When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.

```
create table dataGenSource(
  user_id string,
  amount int
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1', --Generates a piece of data per second.
  'fields.user_id.kind' = 'random', --Specifies a random generator for the user_id field.
  'fields.user_id.length' = '3' --Limits the length of user_id to 3.
);

create table printSink(
  user_id string,
  amount int
) with (
  'connector' = 'print'
);

insert into printSink select * from dataGenSource;
```

After the job is submitted, the job status changes to **Running**. You can perform the following operations to view the output result:

- Method 1:
  a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  b. Locate the row that contains the target Flink job, and choose **More** > **FlinkUI** in the **Operation** column.
  c. On the Flink UI, choose **Task Managers**, click the task name, and select **Stdout** to view job logs.
- Method 2: If you select **Save Job Log** on the **Running Parameters** tab before submitting the job, perform the following operations:

a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

2.3.1.2 GaussDB(DWS) Source Table

Function

DLI reads data of Flink jobs from GaussDB(DWS). GaussDB(DWS) database kernel is compliant with PostgreSQL. The PostgreSQL database can store data of more complex types and deliver space information services, multi-version concurrent control (MVCC), and high concurrency. It applies to location applications, financial insurance, and e-Commerce.

GaussDB(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 GaussDB(DWS), see **Data Warehouse Service Management Guide**.

Prerequisites

- You have created a GaussDB(DWS) cluster.

  For details about how to create a GaussDB(DWS) cluster, see **Creating a Cluster** in the *Data Warehouse Service Management Guide*.
- You have created a GaussDB(DWS) database table.
- An enhanced datasource connection has been created for DLI to connect to GaussDB(DWS) clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.

Syntax

```
create table dwsSource (
  attr_name attr_type
```

```
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
)
with (
  'connector' = 'gaussdb',
  'url' = '',
  'table-name' = '',
  'username' = '',
  'password' = ''
);
```

Parameters

Table 2-3 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **gaussdb**.
url	Yes	None	String	JDBC connection address. Set the IP address in this parameter to the internal IP address of GaussDB(DWS). If you use the gsjdbc4 driver, set the value in jdbc:postgresql://${ip}:${port}/${dbName} format. If you use the gsjdbc200 driver, set the value in jdbc:gaussdb://${ip}:${port}/${dbName} format.
table-name	Yes	None	String	Name of the GaussDB(DWS) table to be operated. If the GaussDB(DWS) table is in a schema, refer to the description of **GaussDB(DWS) table in a schema**.
driver	No	org.postgresql.Driver	String	JDBC connection driver. The default value is **org.postgresql.Driver**. ● If you use the gsjdbc4 driver for connection, set this parameter to **org.postgresql.Driver**. ● If you use the gsjdbc200 driver for connection, set this parameter to **com.huawei.gauss200.jdbc.Driver**.
username	No	None	String	Username for GaussDB(DWS) database authentication. This parameter must be configured in pair with **password**.
password	No	None	String	Password for GaussDB(DWS) database authentication. This parameter must be configured in pair with **username**.

Parameter	Mandatory	Default Value	Data Type	Description
scan.partition.column	No	None	String	Name of the column used to partition the input. Note: This parameter must be used together with **scan.partition.lower-bound**, **scan.partition.upper-bound**, and **scan.partition.num**.
scan.partition.lower-bound	No	None	Integer	Lower bound of values to be fetched for the first partition. This parameter must be used together with **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.num**.
scan.partition.upper-bound	No	None	Integer	Upper bound of values to be fetched for the last partition. This parameter must be used together with **scan.partition.column**, **scan.partition.lower-bound**, and **scan.partition.num**.
scan.partition.num	No	None	Integer	Number of partitions to be created. This parameter must be used together with **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.upper-bound**.
scan.fetch-size	No	0	Integer	Number of rows fetched from the database each time. The default value is **0**, indicating that the number of rows is not limited.
pwd_auth_name	No	None	String	Name of password datasource authentication created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.

Example

In this example, data is read from the GaussDB(DWS) data source and written to the Print result table. The procedure is as follows:

1. Create a table named **dws_order** in GaussDB(DWS).
```
create table public.dws_order(
  order_id VARCHAR,
  order_channel VARCHAR,
  order_time VARCHAR,
  pay_amount FLOAT8,
  real_pay FLOAT8,
  pay_time VARCHAR,
  user_id VARCHAR,
```

```
user_name VARCHAR,
area_id VARCHAR);
```

Insert data into the **dws_order** table.

```
insert into public.dws_order
  (order_id,
  order_channel,
  order_time,
  pay_amount,
  real_pay,
  pay_time,
  user_id,
  user_name,
  area_id) values
  ('202103241000000001', 'webShop', '2021-03-24 10:00:00', '100.00', '100.00', '2021-03-24 10:02:03',
'0001', 'Alice', '330106'),
  ('202103251202020001', 'miniAppShop', '2021-03-25 12:02:02', '60.00', '60.00', '2021-03-25 12:03:00',
'0002', 'Bob', '330110');
```

2. Create an enhanced datasource connection in the VPC and subnet where GaussDB(DWS) locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
3. Set GaussDB(DWS) security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the GaussDB(DWS) address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses the GaussDB(DWS) data source and the Print result table.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE dwsSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'gaussdb',
  'url' = 'jdbc:postgresql://DWSIP:DWSPort/DWSdbName',
  'table-name' = 'dws_order',
  'driver' = 'org.postgresql.Driver',
  'username' = 'DWSUserName',
  'password' = 'DWSPassword'
);

CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);
```

```
insert into printSink select * from dwsSource;
```

5. Perform the following operations to view the data result in the **taskmanager.out** file:
   a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
   b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
   c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I(202103241000000001,webShop,2021-03-24 10:00:00,100.0,100.0,2021-03-24
10:02:03,0001,Alice,330106)
+I(202103251202020001,miniAppShop,2021-03-25 12:02:02,60.0,60.0,2021-03-25
12:03:00,0002,Bob,330110)
```

FAQ

- Q: What should I do if the job execution fails and the log contains the following error information?

```
java.io.IOException: unable to open JDBC writer
...
Caused by: org.postgresql.util.PSQLException: The connection attempt failed.
...
Caused by: java.net.SocketTimeoutException: connect timed out
```

  A: The datasource connection is not bound or the binding fails.

  – To reconfigure datasource connections, refer to **Enhanced Datasource Connection**. Rectify the fault by referring to **DLI Failed to Connect to GaussDB(DWS) Through an Enhanced Datasource Connection**.

- Q: How can I configure a GaussDB(DWS) table that is in a schema?

  A: The following provides an example of configuring the **dws_order** table in the **dbuser2** schema:

```
CREATE TABLE dwsSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'gaussdb',
  'url' = 'jdbc:postgresql://DWSIP:DWSPort/DWSdbName',
  'table-name' = 'dbuser2\".\"dws_order',
  'driver' = 'org.postgresql.Driver',
  'username' = 'DWSUserName',
  'password' = 'DWSPassword'
);
```

2.3.1.3 HBase Source Table

Function

Create a source stream to obtain data from HBase as input for jobs. HBase is a column-oriented distributed cloud storage system that features enhanced reliability, excellent performance, and elastic scalability. It applies to the storage of massive amounts of data and distributed computing. You can use HBase to build a storage system capable of storing TB- or even PB-level data. With HBase, you can filter and analyze data with ease and get responses in milliseconds, rapidly mining data value. DLI can read data from HBase for filtering, analysis, and data dumping.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- If MRS HBase is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.

  For details, see **Modifying Host Information** in the *Data Lake Insight User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- The column families in created HBase source table must be declared as the ROW type, the field names map the column family names, and the nested field names map the column qualifier names.

  There is no need to declare all the families and qualifiers in the schema. Users can declare what is used in the query. Except the ROW type fields, the single atomic type field (for example, STRING or BIGINT) will be recognized as the HBase rowkey. The rowkey field can be an arbitrary name, but should be quoted using backticks if it is a reserved keyword.

Syntax

```
create table hbaseSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
```

```
  ','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'hbase-2.2',
  'table-name' = '',
  'zookeeper.quorum' = ''
);
```

Parameters

Table 2-4 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **hbase-2.2**.
table-name	Yes	None	String	Name of the HBase table to connect.
zookeeper.quorum	Yes	None	String	HBase ZooKeeper quorum, in the format of "ZookeeperAddress:ZookeeperPort". The following uses an MRS HBase cluster as an example to describe how to obtain the IP address and port number of ZooKeeper used by this parameter: • On MRS Manager, choose **Cluster** and click the name of the desired cluster. Choose **Services** > **ZooKeeper** > **Instance**, and obtain the IP address of the ZooKeeper instance. • On MRS Manager, choose **Cluster** and click the name of the desired cluster. Choose **Services** > **ZooKeeper** > **Configurations** > **All Configurations**, search for the **clientPort** parameter, and obtain its value, that is, the ZooKeeper port number.
zookeeper.znode.parent	No	/hbase	String	Root directory in ZooKeeper. The default value is **/hbase**.
null-string-literal	No	None	String	Representation for null values for string fields. HBase source encodes/decodes empty bytes as null values for all types except the string type.
krb_auth_name	No	None	String	Name of datasource authentication of the Kerberos type created on DLI.

Data Type Mapping

HBase stores all data as byte arrays. The data needs to be serialized and deserialized during read and write operations.

When serializing and de-serializing, Flink HBase connector uses utility class **org.apache.hadoop.hbase.util.Bytes** provided by HBase (Hadoop) to convert Flink data types to and from byte arrays.

Flink HBase connector encodes null values to empty bytes, and decode empty bytes to null values for all data types except the string type. For the string type, the null literal is determined by the **null-string-literal** option.

Table 2-5 Data type mapping

Flink SQL Type	HBase Conversion
CHAR/VARCHAR/STRING	byte[] toBytes(String s) String toString(byte[] b)
BOOLEAN	byte[] toBytes(boolean b) boolean toBoolean(byte[] b)
BINARY/VARBINARY	Returns byte[] as is.
DECIMAL	byte[] toBytes(BigDecimal v) BigDecimal toBigDecimal(byte[] b)
TINYINT	new byte[] { val } bytes[0] // returns first and only byte from bytes
SMALLINT	byte[] toBytes(short val) short toShort(byte[] bytes)
INT	byte[] toBytes(int val) int toInt(byte[] bytes)
BIGINT	byte[] toBytes(long val) long toLong(byte[] bytes)
FLOAT	byte[] toBytes(float val) float toFloat(byte[] bytes)
DOUBLE	byte[] toBytes(double val) double toDouble(byte[] bytes)
DATE	Stores the number of days since epoch as an int value.
TIME	Stores the number of milliseconds of the day as an int value.
TIMESTAMP	Stores the milliseconds since epoch as a long value.

Flink SQL Type	HBase Conversion
ARRAY	Not supported
MAP/MULTISET	Not supported
ROW	Not supported

Example

In this example, data is read from the HBase data source and written to the Print result table. The procedure is as follows (the HBase versions used in this example are 1.3.1, 2.1.1, and 2.2.3):

1. Create an enhanced datasource connection in the VPC and subnet where HBase locates, and bind the connection to the required Flink queue. For details, see **Enhanced Datasource Connections**. Add MRS host information for the enhanced datasource connection**Modifying Host Information**.
2. Set HBase cluster security groups and add inbound rules to allow access from the Flink job queue. Test the connectivity using the HBase address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Use the HBase shell to create HBase table **order** that has only one column family **detail**. For details, see **Using HBase from Scratch**. The creation statement is as follows:
```
create 'order', {NAME => 'detail'}
```
4. Run the following command in the HBase shell to insert a data record:
```
put 'order', '202103241000000001', 'detail:order_channel','webShop'
put 'order', '202103241000000001', 'detail:order_time','2021-03-24 10:00:00'
put 'order', '202103241000000001', 'detail:pay_amount','100.00'
put 'order', '202103241000000001', 'detail:real_pay','100.00'
put 'order', '202103241000000001', 'detail:pay_time','2021-03-24 10:02:03'
put 'order', '202103241000000001', 'detail:user_id','0001'
put 'order', '202103241000000001', 'detail:user_name','Alice'
put 'order', '202103241000000001', 'detail:area_id','330106'
```
5. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses the HBase data source and the Print result table.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**
```
create table hbaseSource (
  order_id string,-- Indicates the unique rowkey.
  detail Row( -- Indicates the column family.
    order_channel string,
    order_time string,
    pay_amount string,
    real_pay string,
    pay_time string,
    user_id string,
    user_name string,
    area_id string),
  primary key (order_id) not enforced
) with (
  'connector' = 'hbase-2.2',
   'table-name' = 'order',
   'zookeeper.quorum' = 'ZookeeperAddress:ZookeeperPort'
```

```
) ;

create table printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount string,
  real_pay string,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) with (
 'connector' = 'print'
);

insert into printSink select order_id,
detail.order_channel,detail.order_time,detail.pay_amount,detail.real_pay,
detail.pay_time,detail.user_id,detail.user_name,detail.area_id from hbaseSource;
```

6. Perform the following operations to view the data result in the **taskmanager.out** file:
   a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
   b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
   c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I(202103241000000001,webShop,2021-03-24 10:00:00,100.00,100.00,2021-03-24
10:02:03,0001,Alice,330106)
```

FAQ

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
java.lang.IllegalArgumentException: offset (0) + length (8) exceed the capacity of the array: 6
```

  A: If data in the HBase table is imported in other modes, the data is represented in the string format. Therefore, this error is reported when other data formats are used. Change the type of the non-string fields in the HBase source table created by Flink to the string format.

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
org.apache.zookeeper.ClientCnxn$SessionTimeoutException: Client session timed out, have not heard
from server in 90069ms for connection id 0x0
```

  A: The datasource connection is not bound, the binding fails, or the security group of the HBase cluster is not configured to allow access from the network segment of the DLI queue. Configure the datasource connection by referring to **Enhanced Datasource Connection** or configure the security group of the HBase cluster to allow access from the DLI queue.

2.3.1.4 JDBC Source Table

Function

The JDBC connector is a Flink's built-in connector to read data from a database.

Prerequisites

- An enhanced datasource connection with the instances has been established, so that you can configure security group rules as required.
- For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.

Syntax

```
create table jbdcSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
) with (
  'connector' = 'jdbc',
  'url' = '',
  'table-name' = '',
  'username' = '',
  'password' = ''
);
```

Parameters

Table 2-6 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **jdbc**.
url	Yes	None	String	Database URL.
table-name	Yes	None	String	Name of the table where the data will be read from the database.

Parameter	Mandatory	Default Value	Data Type	Description
driver	No	None	String	Driver required for connecting to the database. If you do not set this parameter, it will be automatically derived from the URL.
username	No	None	String	Database authentication username. This parameter must be configured in pair with **password**.
password	No	None	String	Database authentication password. This parameter must be configured in pair with **username**.
scan.partition.column	No	None	String	Name of the column used to partition the input. For details, see **Partitioned Scan**.
scan.partition.num	No	None	Integer	Number of partitions to be created. For details, see **Partitioned Scan**.
scan.partition.lower-bound	No	None	Integer	Lower bound of values to be fetched for the first partition. For details, see **Partitioned Scan**.
scan.partition.upper-bound	No	None	Integer	Upper bound of values to be fetched for the last partition. For details, see **Partitioned Scan**.
scan.fetch-size	No	0	Integer	Number of rows fetched from the database each time. If this parameter is set to **0**, the SQL hint is ignored.
scan.auto-commit	No	true	Boolean	Whether each statement is committed in a transaction automatically.
pwd_auth_name	No	None	String	Name of datasource authentication of the password type created on DLI. If this parameter is set, you do not need to set the username and password in SQL statements.

Partitioned Scan

To accelerate reading data in parallel Source task instances, Flink provides the partitioned scan feature for the JDBC table. The following parameters describe how to partition the table when reading in parallel from multiple tasks.

- **scan.partition.column**: name of the column used to partition the input. The data type of the column must be number, date, or timestamp.

- **scan.partition.num**: number of partitions.
- **scan.partition.lower-bound**: minimum value of the first partition.
- **scan.partition.upper-bound**: maximum value of the last partition.

NOTE

- **When a table is created, the preceding partitioned scan parameters must all be specified if any of them is specified.**
- The **scan.partition.lower-bound** and **scan.partition.upper-bound** parameters are used to decide the partition stride instead of filtering rows in the table. All rows in the table are partitioned and returned.

Data Type Mapping

Table 2-7 Data type mapping

MySQL Type	PostgreSQL Type	Flink SQL Type
TINYINT	-	TINYINT
SMALLINT TINYINT UNSIGNED	SMALLINT INT2 SMALLSERIAL SERIAL2	SMALLINT
INT MEDIUMINT SMALLINT UNSIGNED	INTEGER SERIAL	INT
BIGINT INT UNSIGNED	BIGINT BIGSERIAL	BIGINT
BIGINT UNSIGNED	-	DECIMAL(20, 0)
BIGINT	BIGINT	BIGINT
FLOAT	REAL FLOAT4	FLOAT
DOUBLE DOUBLE PRECISION	FLOAT8 DOUBLE PRECISION	DOUBLE
NUMERIC(p, s) DECIMAL(p, s)	NUMERIC(p, s) DECIMAL(p, s)	DECIMAL(p, s)
BOOLEAN TINYINT(1)	BOOLEAN	BOOLEAN
DATE	DATE	DATE

MySQL Type	PostgreSQL Type	Flink SQL Type
TIME [(p)]	TIME [(p)] [WITHOUT TIMEZONE]	TIME [(p)] [WITHOUT TIMEZONE]
DATETIME [(p)]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]
CHAR(n) VARCHAR(n) TEXT	CHAR(n) CHARACTER(n) VARCHAR(n) CHARACTER VARYING(n) TEXT	STRING
BINARY VARBINARY BLOB	BYTEA	BYTES
-	ARRAY	ARRAY

Example

This example uses JDBC as the data source and Print as the sink to read data from the RDS MySQL database and write the data to the Print result table.

1. Create an enhanced datasource connection in the VPC and subnet where RDS MySQL locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set RDS MySQL security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the RDS address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Log in to the RDS MySQL database, create table **orders** in the Flink database, and insert data.

   Create table **orders** in the Flink database.

```
CREATE TABLE `flink`.`orders` (
    `order_id` VARCHAR(32) NOT NULL,
    `order_channel` VARCHAR(32) NULL,
    `order_time` VARCHAR(32) NULL,
    `pay_amount` DOUBLE UNSIGNED NOT NULL,
    `real_pay` DOUBLE UNSIGNED NULL,
    `pay_time` VARCHAR(32) NULL,
    `user_id` VARCHAR(32) NULL,
    `user_name` VARCHAR(32) NULL,
    `area_id` VARCHAR(32) NULL,
    PRIMARY KEY (`order_id`)
```

```
)    ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4
    COLLATE = utf8mb4_general_ci;
```

Insert data into the table.

```
insert into orders(
  order_id,
  order_channel,
  order_time,
  pay_amount,
  real_pay,
  pay_time,
  user_id,
  user_name,
  area_id) values
  ('202103241000000001', 'webShop', '2021-03-24 10:00:00', '100.00', '100.00', '2021-03-24 10:02:03',
'0001', 'Alice', '330106'),
  ('202103251202020001', 'miniAppShop', '2021-03-25 12:02:02', '60.00', '60.00', '2021-03-25 12:03:00',
'0002', 'Bob', '330110');
```

4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE jdbcSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://MySQLAddress:MySQLPort/flink',--flink is the database name created in RDS
MySQL.
  'table-name' = 'orders',
  'username' = 'MySQLUsername',
  'password' = 'MySQLPassword'
);

CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);

insert into printSink select * from jdbcSource;
```

5. Perform the following operations to view the data result in the **taskmanager.out** file:

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

b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.

c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

The data result is as follows:

```
+I(202103241000000001,webShop,2021-03-24 10:00:00,100.0,100.0,2021-03-24
10:02:03,0001,Alice,330106)
+I(202103251202020001,miniAppShop,2021-03-25 12:02:02,60.0,60.0,2021-03-25
12:03:00,0002,Bob,330110)
```

FAQ

None

2.3.1.5 Kafka Source Table

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

- You have created a Kafka cluster.
- An enhanced datasource connection has been created for DLI to connect to Kafka clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- For details about how to use data types when creating tables, see **Format**.

Syntax

```
create table kafkaSource(
  attr_name attr_type
```

```
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' WATERMARK FOR rowtime_column_name AS watermark-strategy_expression)
)
with (
  'connector' = 'kafka',
  'topic' = '',
  'properties.bootstrap.servers' = '',
  'properties.group.id' = '',
  'scan.startup.mode' = '',
  'format' = ''
);
```

Parameters

Table 2-8 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **kafka**.
topic	Yes	None	String	Topic name of the Kafka record. Note: • Only one of **topic** and **topic-pattern** can be specified. • If there are multiple topics, separate them with semicolons (;), for example, **topic-1;topic-2**.
topic-pattern	No	None	String	Regular expression for a pattern of topic names to read from. Only one of **topic** and **topic-pattern** can be specified. For example: 'topic.*' '(topic-c|topic-d)' '(topic-a|topic-b|topic-\\d*)' '(topic-a|topic-b|topic-[0-9]*)'
properties.bootstrap.servers	Yes	None	String	Comma separated list of Kafka brokers.
properties.group.id	Yes	None	String	ID of the consumer group for the Kafka source.

Parameter	Mandatory	Default Value	Data Type	Description
properties.*	No	None	String	This parameter can set and pass arbitrary Kafka configurations. Note: • The suffix to **properties.** must match the configuration key in **Apache Kafka**. For example, you can disable automatic topic creation via **'properties.allow.auto.create.topics' = 'false'**. • Some configurations are not supported, for example, **'key.deserializer'** and **'value.deserializer'**.
format	Yes	None	String	Format used to deserialize and serialize the value part of Kafka messages. Note: Either this parameter or the **value.format** parameter is required. Refer to **Format** for more details and format parameters.
key.format	No	None	String	Format used to deserialize and serialize the key part of Kafka messages. Note: • If a key format is defined, the **key.fields** parameter is required as well. Otherwise, the Kafka records will have an empty key. • Refer to **Format** for more details and format parameters.
key.fields	No	[]	List<String>	Defines the columns in the table as the list of keys. This parameter must be configured in pair with **key.format**. This parameter is left empty by default. Therefore, no key is defined. The format is like **field1;field2**.
key.fields-prefix	No	None	String	Defines a custom prefix for all fields of the key format to avoid name clashes with fields of the value format.

Parameter	Mandatory	Default Value	Data Type	Description
value.format	Yes	None	String	Format used to deserialize and serialize the value part of Kafka messages. Note: ● Either this parameter or the **format** parameter is required. If two parameters are configured, a conflict occurs. ● Refer to **Format** for more details and format parameters.
value.fields-include	No	ALL	Enum Possible values: [ALL, EXCEPT_KEY]	Whether to contain the key field when parsing the message body. Possible values are: ● **ALL** (default): All defined fields are included in the value of Kafka messages. ● **EXCEPT_KEY**: All the fields except those defined by **key.fields** are included in the value of Kafka messages.
scan.startup.mode	No	group-offsets	String	Start position for Kafka to read data. Possible values are: ● **earliest-offset**: Data is read from the earliest Kafka offset. ● **latest-offset**: Data is read from the latest Kafka offset. ● **group-offsets** (default): Data is read based on the consumer group. ● **timestamp**: Data is read from a user-supplied timestamp. When setting this option, you also need to specify **scan.startup.timestamp-millis** in **WITH**. ● **specific-offsets**: Data is read from user-supplied specific offsets for each partition. When setting this option, you also need to specify **scan.startup.specific-offsets** in **WITH**.

Parameter	Mandatory	Default Value	Data Type	Description
scan.startup.specific-offsets	No	None	String	This parameter takes effect only when **scan.startup.mode** is set to **specific-offsets**. It specifies the offsets for each partition, for example, **partition:0,offset:42;partition:1,offset:300**.
scan.startup.timestamp-millis	No	None	Long	Startup timestamp. This parameter takes effect when **scan.startup.mode** is set to **timestamp**.
scan.topic-partition-discovery.interval	No	None	Duration	Interval for a consumer to periodically discover dynamically created Kafka topics and partitions.
ssl_auth_name	No	None	String	Name of datasource authentication of the Kafka_SSL type created on DLI. This configuration is used when SSL is configured for Kafka. Note: If only the SSL type is used, you need to set **properties.security.protocol** to **SSL**. If SASL_SSL is used, set the following parameters: ● 'properties.security.protocol' = 'SASL_SSL'; ● 'properties.sasl.mechanism' = 'GSSAPI or PLAIN'; ● 'properties.sasl.jaas.config' = 'org.apache.kafka.common.security.plain.PlainLoginModule required username=\"xxx\" password=\"xxx\";'
krb_auth_name	No	None	String	Name of datasource authentication of the Kerberos type created on DLI. This configuration is used when SASL is configured for Kafka. Note: If the SASL_PLAINTEXT type and Kerberos authentication are used, you need to set **properties.sasl.mechanism** to **GSSAPI** and **properties.security.protocol** to **SASL_PLAINTEXT**.

Metadata Column

You can define metadata columns in the source table to obtain the metadata of Kafka messages. For example, if multiple topics are defined in the **WITH** parameter and the metadata column is defined in the Kafka source table, the data read by Flink is labeled with the topic from which the data is read.

Table 2-9 Metadata column

Key	Data Type	R/W	Description
topic	STRING NOT NULL	R	Topic name of the Kafka record.
partition	INT NOT NULL	R	Partition ID of the Kafka record.
headers	MAP<STRING, BYTES> NOT NULL	R/W	Headers of Kafka messages.
leader-epoch	INT NULL	R	Leader epoch of the Kafka record. **For details, see example 1.**
offset	BIGINT NOT NULL	R	Offset of the Kafka record.
timestamp	TIMESTAMP(3) WITH LOCAL TIME ZONE NOT NULL	R/W	Timestamp of the Kafka record.
timestamp-type	STRING NOT NULL	R	Timestamp type of the Kafka record. The options are as follows: • **NoTimestampType**: No timestamp is defined in the message. • **CreateTime**: time when the message is generated. • **LogAppendTime**: time when the message is added to the Kafka broker. **For details, see example 1.**

Example (SASL_SSL Disabled for the Kafka Cluster)

- **Example 1: Read data from the Kafka metadata column and write it to the Print sink.**
  a. Create an enhanced datasource connection in the VPC and subnet where Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
  b. Set Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
  c. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

     When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE orders (
  `topic` String metadata,
  `partition` int metadata,
  `headers` MAP<STRING, BYTES> metadata,
  `leaderEpoch` INT metadata from 'leader-epoch',
  `offset` bigint metadata,
  `timestamp` TIMESTAMP(3) metadata,
  `timestampType` string metadata from 'timestamp-type',
  `message` string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  "format" = "csv",
  "csv.field-delimiter" = "\u0001",
  "csv.quote-character" = "''"
);

CREATE TABLE printSink (
  `topic` String,
  `partition` int,
  `headers` MAP<STRING, BYTES>,
  `leaderEpoch` INT,
  `offset` bigint,
  `timestamp` TIMESTAMP(3),
  `timestampType` string,
  `message` string -- Indicates that data written by users is read from Kafka.
) WITH (
  'connector' = 'print'
);

insert into printSink select * from orders;
```

     If you need to read the value of each field instead of the entire message, use the following statements:

```
CREATE TABLE orders (
  `topic` String metadata,
  `partition` int metadata,
  `headers` MAP<STRING, BYTES> metadata,
  `leaderEpoch` INT metadata from 'leader-epoch',
  `offset` bigint metadata,
  `timestamp` TIMESTAMP(3) metadata,
  `timestampType` string metadata from 'timestamp-type',
  order_id string,
```

```
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourTopic>',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  "format" = "json"
);

CREATE TABLE printSink (
  `topic` String,
  `partition` int,
  `headers` MAP<STRING, BYTES>,
  `leaderEpoch` INT,
  `offset` bigint,
  `timestamp` TIMESTAMP(3),
  `timestampType` string,
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);

insert into printSink select * from orders;
```

d. Send the following data to the corresponding topics in Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24
10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03",
"user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24
16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06",
"user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

e. Perform the following operations to view the output:

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

   ii. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.

   iii. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

The data result is as follows:

```
+I(fz-source-json,0,{},0,243,2021-12-27T09:23:32.253,CreateTime,
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24
```

```
10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03",
"user_id":"0001", "user_name":"Alice", "area_id":"330106"})
+I(fz-source-json,0,{},0,244,2021-12-27T09:23:39.655,CreateTime,
{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24
16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06",
"user_id":"0001", "user_name":"Alice", "area_id":"330106"})
+I(fz-source-json,0,{},0,245,2021-12-27T09:23:48.405,CreateTime,
{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"})
```

- **Example 2: Use the Kafka source table and Print result table to read JSON data from Kafka and output it to the log file.**
  a. Create an enhanced datasource connection in the VPC and subnet where Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
  b. Set Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
  c. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

     When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourTopic>',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  "format" = "json"
);

CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);

insert into printSink select * from orders;
```

  d. Send the following test data to the corresponding topics in Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24
10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03",
```



```
"user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24
16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06",
"user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

e. Perform the following operations to view the output:

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

   ii. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.

   iii. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I(202103241000000001,webShop,2021-03-24T10:00,100.0,100.0,2021-03-2410:02:03,0001,Alice,
330106)
+I(202103241606060001,appShop,2021-03-24T16:06:06,200.0,180.0,2021-03-2416:10:06,0001,Ali
ce,330106)
+I(202103251202020001,miniAppShop,2021-03-25T12:02:02,60.0,60.0,2021-03-2512:03:00,0002,
Bob,330110)
```

Example (SASL_SSL Enabled for the Kafka Cluster)

- **Example 1: Enable SASL_SSL authentication for the DMS cluster.**

  Create a Kafka cluster for DMS, enable SASL_SSL, download the SSL certificate, and upload the downloaded certificate **client.jks** to an OBS bucket.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:9093,xx:9093,xx:9093',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.connector.auth.open' = 'true',
  'properties.ssl.truststore.location' = 'obs://xx/xx.jks',  -- Location where the user uploads the
certificate to
  'properties.sasl.mechanism' = 'PLAIN',  --  Value format: SASL_PLAINTEXT
  'properties.security.protocol' = 'SASL_SSL',
  'properties.sasl.jaas.config' = 'org.apache.kafka.common.security.plain.PlainLoginModule required
username=\"xx\" password=\"xx\";', -- Account and password set when the Kafka cluster is created
  "format" = "json"
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
```

```
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:9093,xx:9093,xx:9093',
  'properties.connector.auth.open' = 'true',
  'properties.ssl.truststore.location' = 'obs://xx/xx.jks',
  'properties.sasl.mechanism' = 'PLAIN',
  'properties.security.protocol' = 'SASL_SSL',
  'properties.sasl.jaas.config' = 'org.apache.kafka.common.security.plain.PlainLoginModule required
username=\"xx\" password=\"xx\";',
  "format" = "json"
);

insert into ordersSink select * from ordersSource;
```

- **Example 2: Enable Kafka SASL_SSL authentication for the MRS cluster.**
  – Enable Kerberos authentication for the MRS cluster.
  – Click the **Components** tab and click **Kafka**. In the displayed page, click the **Service Configuration** tab, locate the **security.protocol**, and set it to **SASL_SSL**.
  – Download the user credential. Log in to the FusionInsight Manager of the MRS cluster and choose **System** > **Permission** > **User**. Locate the row that contains the target user, click **More**, and select **Download Authentication Credential**.

    Obtain the **truststore.jks** file using the authentication credential and store the credential and **truststore.jks** file in OBS.
  – If "Message stream modified (41)" is displayed, the JDK version may be incorrect. Change the JDK version in the sample code to a version earlier than 8u_242 or delete the **renew_lifetime = 0m** configuration item from the **krb5.conf** configuration file.
  – Set the port to the **sasl_ssl.port** configured in the Kafka service configuration.
  – In the following statements, set **security.protocol** to **SASL_SSL**.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:21009,xx:21009',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx', --Username
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.security.protocol' = 'SASL_SSL',
```

```
  'properties.ssl.truststore.location' = 'obs://xx/truststore.jks',
  'properties.ssl.truststore.password' = 'xx',  -- Password set for generating truststore.jks
  'properties.sasl.mechanism' = 'GSSAPI',
  "format" = "json"
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:21009,xx:21009',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx',
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.ssl.truststore.location' = 'obs://xx/truststore.jks',
  'properties.ssl.truststore.password' = 'xx',
  'properties.security.protocol' = 'SASL_SSL',
  'properties.sasl.mechanism' = 'GSSAPI',
  "format" = "json"
);

insert into ordersSink select * from ordersSource;
```

- **Example 3: Enable Kerberos SASL_PAINTEXT authentication for the MRS cluster**
  - Enable Kerberos authentication for the MRS cluster.
  - Click the **Components** tab and click **Kafka**. In the displayed page, click the **Service Configuration** tab, locate the **security.protocol**, and set it to **SASL_PLAINTEXT**.
  - Log in to the FusionInsight Manager of the MRS cluster and download the user credential. Choose **System** > **Permission** > **User**. Locate the row that contains the target user, choose **More** > **Download Authentication Credential**. Upload the credential to OBS.
  - If error message "Message stream modified (41)" is displayed, the JDK version may be incorrect. Change the JDK version in the sample code to a version earlier than 8u_242 or delete the **renew_lifetime = 0m** configuration item from the **krb5.conf** configuration file.
  - Set the port to the **sasl.port** configured in the Kafka service configuration.
  - In the following statements, set **security.protocol** to **SASL_PLAINTEXT**.

```
CREATE TABLE ordersSources (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
```



```
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:21007,xx:21007',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx',
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  "format" = "json"
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:21007,xx:21007',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx',
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  "format" = "json"
);

insert into ordersSink select * from ordersSource;
```

- **Example 4: Use SSL for the MRS cluster**
  - Do not enable Kerberos authentication for the MRS cluster.
  - Download the user credential. Log in to the FusionInsight Manager of the MRS cluster and choose **System** > **Permission** > **User**. Locate the row that contains the target user, click **More**, and select **Download Authentication Credential**.

    Obtain the **truststore.jks** file using the authentication credential and store the credential and **truststore.jks** file in OBS.
  - Set the port to the **ssl.port** configured in the Kafka service configuration.
  - In the following statements, set **security.protocol** to **SSL**.
  - Set **ssl.mode.enable** to **true**.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
```

```
  'properties.bootstrap.servers' = 'xx:9093,xx:9093,xx:9093',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.connector.auth.open' = 'true',
  'properties.ssl.truststore.location' = 'obs://xx/truststore.jks',
  'properties.ssl.truststore.password' = 'xx',  -- Password set for generating truststore.jks
  'properties.security.protocol' = 'SSL',
  "format" = "json"
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);

insert into ordersSink select * from ordersSource;
```

FAQ

- **Q: What should I do if the Flink job execution fails and the log contains the following error information?**

  ```
  org.apache.kafka.common.errors.TimeoutException: Timeout expired while fetching topic metadata
  ```

  A: The datasource connection is not bound, the binding fails, or the security group of the Kafka cluster is not configured to allow access from the network segment of the DLI queue. Configure the datasource connection by referring to **Enhanced Datasource Connection** or configure the security group of the Kafka cluster to allow access from the DLI queue.

- **Q: What should I do if the Flink job execution fails and the log contains the following error information?**

  ```
  Caused by: java.lang.RuntimeException: RealLine:45;Table 'default_catalog.default_database.printSink'
  declares persistable metadata columns, but the underlying DynamicTableSink doesn't implement the
  SupportsWritingMetadata interface. If the column should not be persisted, it can be declared with the
  VIRTUAL keyword.
  ```

  A: The metadata type is defined in the sink table, but the Print connector does not support deletion of matadata from the sink table.

2.3.1.6 MySQL CDC Source Table

Function

The MySQL CDC source table, that is, the MySQL streaming source table, reads all historical data in the database first and then smoothly switches data read to the Binlog to ensure data integrity.

Prerequisites

- MySQL CDC requires MySQL 5.7 or 8.0.*x*.
- An enhanced datasource connection has been created for DLI to connect to the MySQL database, so that you can configure security group rules as required.

  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.
- Binlog is enabled for MySQL, and **binlog_row_image** is set to **FULL**.
- A MySQL user has been created and granted the **SELECT**, **SHOW DATABASES**, **REPLICATION SLAVE**, and **REPLICATION CLIENT** permissions.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- Each client that synchronizes database data has a unique ID, that is, the server ID. You are advised to configure a unique server ID for each MySQL CDC job in the same database.

  Main reasons are as follows:
  - The MySQL server maintains the network connection and Binlog location based on the ID. Therefore, if a large number of clients with the same server ID connect to the MySQL server, the CPU usage of the MySQL server may increase sharply, affecting the stability of online services.
  - If multiple jobs share the same server ID, Binlog locations will be disordered, making data read inaccurate. Therefore, you are advised to configure different server IDs for each MySQL CDC job.
- Watermarks cannot be defined for MySQL CDC source tables. For details about window aggregation, see **FAQ**.
- If you connect to a sink source that supports upsert, such as GaussDB(DWS) and MySQL, you need to define the primary key in the statement for creating the sink table. For details, see the printSink table creation statement in **Example**.
- When using the MySQL CDM source table, do not manually disable **debezium.connect.keep.alive** in the source table parameters. Make sure that **debezium.connect.keep.alive** is set to **true** (default value).

  If **debezium.connect.keep.alive** is manually disabled and there is a connection exception between the Binlog thread and the MySQL server, the Binlog thread will not attempt to reconnect automatically. This may result in the inability to properly pull binlog logs from the source.

Syntax

```
create table mySqlCdcSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
```

```
  'connector' = 'mysql-cdc',
  'hostname' = 'mysqlHostname',
  'username' = 'mysqlUsername',
  'password' = 'mysqlPassword',
  'database-name' = 'mysqlDatabaseName',
  'table-name' = 'mysqlTableName'
);
```

Parameters

Table 2-10 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **mysql-cdc**.
hostname	Yes	None	String	IP address or hostname of the MySQL database.
username	Yes	None	String	Username of the MySQL database.
password	Yes	None	String	Password of the MySQL database.
database-name	Yes	None	String	Name of the database to connect. The database name supports regular expressions to read data from multiple databases. For example, **flink(.)*** indicates all database names starting with **flink**.
table-name	Yes	None	String	Name of the table to read data from. The table name supports regular expressions to read data from multiple tables. For example, **cdc_order(.)*** indicates all table names starting with **cdc_order**.
port	No	3306	Integer	Port number of the MySQL database.
server-id	No	A random value from 5400 to 6000	String	A numeric ID of the database client, which must be globally unique in the MySQL cluster. You are advised to set a unique ID for each job in the same database. By default, a random value ranging from 5400 to 6400 is generated.

Parameter	Mandatory	Default Value	Data Type	Description
scan.startup.mode	No	initial	String	Startup mode for consuming data. ● **initial** (default): In the first startup, the database scans all historical data and then reads the latest Binlog data. ● **latest-offset**: In the first startup, the database reads data directly from the end of the Binlog (the latest Binlog) instead of scanning all historical data. That is, it reads only the latest changes after the connector is started.
server-time-zone	No	None	String	Time zone of the session used by the database. For example, **Asia/Shanghai**.
pwd_auth_name	No	None	String	Name of datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.

Example

In this example, MySQL-CDC is used to read data from RDS for MySQL in real time and write the data to the Print result table. The procedure is as follows (MySQL 5.7.32 is used in this example):

1. Create an enhanced datasource connection in the VPC and subnet where MySQL locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set MySQL security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the MySQL address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Create a table named **cdc_order** in database **flink** of the MySQL database.

```
CREATE TABLE `flink`.`cdc_order` (
    `order_id` VARCHAR(32) NOT NULL,
    `order_channel` VARCHAR(32) NULL,
    `order_time` VARCHAR(32) NULL,
    `pay_amount` DOUBLE  NULL,
    `real_pay` DOUBLE  NULL,
    `pay_time` VARCHAR(32) NULL,
    `user_id` VARCHAR(32) NULL,
    `user_name` VARCHAR(32) NULL,
    `area_id` VARCHAR(32) NULL,
    PRIMARY KEY (`order_id`)
)   ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4
    COLLATE = utf8mb4_general_ci;
```


4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
create table mysqlCdcSource(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING
) with (
  'connector' = 'mysql-cdc',
  'hostname' = 'mysqlHostname',
  'username' = 'mysqlUsername',
  'password' = 'mysqlPassword',
  'database-name' = 'mysqlDatabaseName',
  'table-name' = 'mysqlTableName'
);

create table printSink(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING,
  primary key(order_id) not enforced
) with (
  'connector' = 'print'
);

insert into printSink select * from mysqlCdcSource;
```

5. Insert test data in MySQL.

```
insert into cdc_order values
('202103241000000001','webShop','2021-03-24 10:00:00','100.00','100.00','2021-03-24
10:02:03','0001','Alice','330106'),
('202103241606060001','appShop','2021-03-24 16:06:06','200.00','180.00','2021-03-24
16:10:06','0001','Alice','330106');

delete from cdc_order  where order_channel = 'webShop';

insert into cdc_order values('202103251202020001','miniAppShop','2021-03-25
12:02:02','60.00','60.00','2021-03-25 12:03:00','0002','Bob','330110');
```

6. Perform the following operations to view the data result in the **taskmanager.out** file:
   a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
   b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
   c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I(202103241000000001,webShop,2021-03-2410:00:00,100.0,100.0,2021-03-2410:02:03,0001,Alice,330
106)
+I(202103241606060001,appShop,2021-03-2416:06:06,200.0,180.0,2021-03-2416:10:06,0001,Alice,3301
06)
-
D(202103241000000001,webShop,2021-03-2410:00:00,100.0,100.0,2021-03-2410:02:03,0001,Alice,330
106)
+I(202103251202020001,miniAppShop,2021-03-2512:02:02,60.0,60.0,2021-03-2512:03:00,0002,Bob,330
110)
```

FAQ

Q: How do I perform window aggregation if the MySQL CDC source table does not support definition of watermarks?

A: You can use the non-window aggregation method. That is, convert the time field into a window value, and then use **GROUP BY** to perform aggregation based on the window value.

For example, you can use the following script to collect statistics on the number of orders per minute (**order_time** indicates the order time, in the string format):

```
insert into printSink select DATE_FORMAT(order_time, 'yyyy-MM-dd HH:mm'), count(*) from
mysqlCdcSource group by DATE_FORMAT(order_time, 'yyyy-MM-dd HH:mm');
```

2.3.1.7 Postgres CDC Source Table

Function

The Postgres CDC source table, that is, Postgres streaming source table, is used to read the full snapshot data and changed data of the PostgreSQL database in sequence. The exactly-once processing semantics is used to ensure data accuracy even if a failure occurs.

Prerequisites

- The PostgreSQL version be 9.6, 10, 11, or 12.
- An enhanced datasource connection with the database has been established, so that you can configure security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- The PostgreSQL version cannot be earlier than PostgreSQL 11.

- If operations such as update will be performed on the Postgres table, you need to run the following statement in PostgreSQL. Note: Replace **test.cdc_order** with the actual database and table.
  ```
  ALTER TABLE test.cdc_order REPLICA IDENTITY FULL
  ```
- Before creating the PostgreSQL CDC source table, check whether the current PostgreSQL contains the default plug-in. You can run the following statement in PostgreSQL to query the current plug-ins:
  ```
  SELECT name FROM pg_available_extensions;
  ```
  If the default plug-in **decoderbufs** is not available, you need to set the **decoding.plugin.name** parameter to specify an existing plug-in in PostgreSQL when creating the PostgreSQL CDC source table.

Syntax

```
create table postgresCdcSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'postgres-cdc',
  'hostname' = 'PostgresHostname',
  'username' = 'PostgresUsername',
  'password' = 'PostgresPassword',
  'database-name' = 'PostgresDatabaseName',
  'schema-name' = 'PostgresSchemaName',
  'table-name' = 'PostgresTableName'
);
```

Parameters

Table 2-11 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **postgres-cdc**.
hostname	Yes	None	String	IP address or hostname of the Postgres database.
username	Yes	None	String	Username of the Postgres database.
password	Yes	None	String	Password of the Postgres database.
database-name	Yes	None	String	Database name.

Parameter	Mandatory	Default Value	Data Type	Description
schema-name	Yes	None	String	Postgres schema name. The schema name supports regular expressions to read data from multiple schemas. For example, **test(.)*** indicates all schema names starting with **test**.
table-name	Yes	None	String	Postgres table name. The table name supports regular expressions to read data from multiple tables. For example, **cdc_order(.)*** indicates all table names starting with **cdc_order**.
port	No	5432	Integer	Port number of the Postgres database.
decoding.plugin.name	No	decoderbufs	String	Determined based on the plug-in that is installed in the PostgreSQL database. The value can be: ● decoderbufs (default) ● wal2json ● wal2json_rds ● wal2json_streaming ● wal2json_rds_streaming ● pgoutput
debezium.*	No	None	String	Fine-grained control over the behavior of Debezium clients, for example, **'debezium.snapshot.mode' = 'never'**. You are advised to set the **debezium.slot.name** parameter for each table to avoid the following error: "PSQLException: ERROR: replication slot "debezium" is active for PID 974"
pwd_auth_name	No	None	String	Name of datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.

Example

In this example, Postgres-CDC is used to read data from RDS for PostgreSQL in real time and write the data to the Print result table. The procedure is as follows (PostgreSQL 11.11 is used in this example):

1. Create an enhanced datasource connection in the VPC and subnet where PostgreSQL locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set PostgreSQL security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the PostgreSQL address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. In PostgreSQL, create database **flink** and schema **test**.
4. Create table **cdc_order** in the schema **test** of database **flink** in PostgreSQL.
```
create table test.cdc_order(
  order_id VARCHAR,
  order_channel VARCHAR,
  order_time VARCHAR,
  pay_amount FLOAT8,
  real_pay FLOAT8,
  pay_time VARCHAR,
  user_id VARCHAR,
  user_name VARCHAR,
  area_id VARCHAR,
  primary key(order_id)
);
```
5. Run the following SQL statement in PostgreSQL. If you do not run this statement, an error will be reported when the Flink job is executed. For details, see the error message in **FAQ**.
```
ALTER TABLE test.cdc_order REPLICA IDENTITY FULL
```
6. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**
```
create table postgresCdcSource(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING,
  primary key (order_id) not enforced
) with (
  'connector' = 'postgres-cdc',
  'hostname' = 'PostgresHostname',
  'username' = 'PostgresUsername',
  'password' = 'PostgresPassword',
  'database-name' = 'flink',
  'schema-name' = 'test',
  'table-name' = 'cdc_order'
);

create table printSink(
  order_id string,
  order_channel string,
```

```
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING,
  primary key(order_id) not enforced
) with (
  'connector' = 'print'
);

insert into printSink select * from postgresCdcSource;
```

7. Run the following command in PostgreSQL:

```
insert into test.cdc_order
  (order_id,
  order_channel,
  order_time,
  pay_amount,
  real_pay,
  pay_time,
  user_id,
  user_name,
  area_id) values
  ('202103241000000001', 'webShop', '2021-03-24 10:00:00', '100.00', '100.00', '2021-03-24 10:02:03',
'0001', 'Alice', '330106'),
  ('202103251202020001', 'miniAppShop', '2021-03-25 12:02:02', '60.00', '60.00', '2021-03-25 12:03:00',
'0002', 'Bob', '330110');

update test.cdc_order set order_channel = 'webShop' where order_id = '202103251202020001';

delete from test.cdc_order where order_id = '202103241000000001';
```

8. Perform the following operations to view the data result in the **taskmanager.out** file:
   a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
   b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
   c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I(202103241000000001,webShop,2021-03-24 10:00:00,100.0,100.0,2021-03-24
10:02:03,0001,Alice,330106)
+I(202103251202020001,miniAppShop,2021-03-25 12:02:02,60.0,60.0,2021-03-25
12:03:00,0002,Bob,330110)
-U(202103251202020001,miniAppShop,2021-03-25 12:02:02,60.0,60.0,2021-03-25
12:03:00,0002,Bob,330110)
+U(202103251202020001,webShop,2021-03-25 12:02:02,60.0,60.0,2021-03-25
12:03:00,0002,Bob,330110)
-D(202103241000000001,webShop,2021-03-24 10:00:00,100.0,100.0,2021-03-24
10:02:03,0001,Alice,330106)
```

FAQ

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
org.postgresql.util.PSQLException: ERROR: logical decoding requires wal_level >= logical
```

- A: Change the value of **wal_level** to **logical** and restart the PostgreSQL database.

After modifying the PostgreSQL parameter, restart the RDS PostgreSQL instance for the modification to take effect.

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

  ```
  java.lang.IllegalStateException: The "before" field of UPDATE/DELETE message is null, please check the Postgres table has been set REPLICA IDENTITY to FULL level. You can update the setting by running the command in Postgres 'ALTER TABLE test.cdc_order REPLICA IDENTITY FULL'.
  ```

  A: If a similar error is reported in the run log, run the **ALTER TABLE test.cdc_order REPLICA IDENTITY FULL** statement in PostgreSQL.

2.3.1.8 Redis Source Table

Function

Create a source stream to obtain data from Redis as input for jobs.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to the Redis database, so that you can configure security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- To obtain the key values, you can set the primary key in Flink. The primary key maps to the Redis key.
- The primary key cannot be a composite primary key, and only can be one field.
- Constraints on **schema-syntax**:
  - If **schema-syntax** is **map** or **array**, there can be only one non-primary key and it must be of the same **map** or **array** type.
  - If **schema-syntax** is **fields-scores**, the number of non-primary keys must be an even number, and the second key of every two keys except the primary key must be of the **double** type. The **double** value is the score of the previous key. The following is an example:

    ```
    CREATE TABLE redisSource (
      redisKey string,
      order_id string,
      score1 double,
      order_channel string,
      score2 double,
    ```

```
  order_time string,
  score3 double,
  pay_amount double,
  score4 double,
  real_pay double,
  score5 double,
  pay_time string,
  score6 double,
  user_id string,
  score7 double,
  user_name string,
  score8 double,
  area_id string,
  score9 double,
  primary key (redisKey) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields-scores'
);
```

- Restrictions on **data-type**:
  - When **data-type** is **set**, the types of non-primary keys defined in Flink must be the same.
  - If **data-type** is **sorted-set** and **schema-syntax** is **fields** or **array**, only **sorted-set** values can be read from Redis, and the **score** value cannot be read.
  - If **data-type** is **string**, only one non-primary key field is allowed.
  - If **data-type** is **sorted-set** and **schema-syntax** is **map**, only one non-primary key field is allowed besides the primary key field.

    This non-primary key field must be of the **map** type. The map value of the field must be of the **double** type, indicating the score. The map key of the field indicates the value in the Redis set.
  - If **data-type** is **sorted-set** and **schema-syntax** is **array-scores**, only two non-primary keys are allowed and must be of the **array** type.

    The first key indicates values in the Redis set. The second key is of the **array<double>** type, indicating index scores. The following is an example:

```
CREATE TABLE redisSink (
  order_id string,
  arrayField Array<String>,
  arrayScore array<double>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  "default-score" = '3',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'array-scores'
);
```

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
  ,PRIMARY KEY (attr_name, ...) NOT ENFORCED
```

```
)
with (
  'connector' = 'redis',
  'host' = ''
);
```

Parameters

Table 2-12 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **redis**.
host	Yes	None	String	Redis connector address.
port	No	6379	Integer	Redis connector port.
password	No	None	String	Redis authentication password.
namespace	No	None	String	Redis key namespace.
delimiter	No	:	String	Delimiter between the Redis key and namespace.
data-type	No	hash	String	Redis data type. Available values are as follows: ● hash ● list ● set ● sorted-set ● string For details about the constraints, see **Constraints on data-type**.

Parameter	Mandatory	Default Value	Data Type	Description
schema-syntax	No	fields	String	Redis schema semantics. Available values are as follows (for details, see **Precautions** and **FAQ**): ● **fields**: applicable to all data types ● **fields-scores**: applicable to **sorted-set** data ● **array**: applicable to **list**, **set**, and **sorted-set** data ● **array-scores**: applicable to **sorted-set** data ● **map**: applicable to **hash** and **sorted-set** data For details about the constraints, see **Constraints on schema-syntax**.
deploy-mode	No	standalone	String	Deployment mode of the Redis cluster. The value can be **standalone**, **master-replica**, or **cluster**. The default value is **standalone**.
retry-count	No	5	Integer	Number of attempts to connect to the Redis cluster.
connection-timeout-millis	No	10000	Integer	Maximum timeout for connecting to the Redis cluster.
commands-timeout-millis	No	2000	Integer	Maximum time for waiting for a completion response.
rebalancing-timeout-millis	No	15000	Integer	Sleep time when the Redis cluster fails.
scan-keys-count	No	1000	Integer	Number of data records read in each scan.
default-score	No	0	Double	Default score when **data-type** is **sorted-set**.

Parameter	Mandatory	Default Value	Data Type	Description
deserialize-error-policy	No	fail-job	Enum	Policy of how to process a data parsing failure. Available values are as follows: ● **fail-job**: Fail the job. ● **skip-row**: Skip the current data. ● **null-field**: Set the current data to null.
skip-null-values	No	true	Boolean	Whether null values will be skipped.
pwd_auth_name	No	None	String	Name of datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.

Example

In this example, data is read from the DCS Redis data source and written to the Print result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where Redis locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set Redis security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Redis address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Run the following commands on the Redis client to insert data into different keys and store the data in hash format:

```
HMSET redisSource order_id 202103241000000001 order_channel webShop order_time "2021-03-24 10:00:00" pay_amount 100.00 real_pay 100.00 pay_time "2021-03-24 10:02:03" user_id 0001 user_name Alice area_id 330106

HMSET redisSource1 order_id 202103241606060001 order_channel appShop order_time "2021-03-24 16:06:06" pay_amount 200.00 real_pay 180.00 pay_time "2021-03-24 16:10:06" user_id 0001 user_name Alice area_id 330106

HMSET redisSource2 order_id 202103251202020001 order_channel miniAppShop order_time "2021-03-25 12:02:02" pay_amount 60.00 real_pay 60.00 pay_time "2021-03-25 12:03:00" user_id 0002 user_name Bob area_id 330110
```

4. Create a Flink OpenSource SQL job. Enter the following job script to read data in hash format from Redis.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.

Change the values of the parameters in bold as needed in the following script.

```
CREATE TABLE redisSource (
  redisKey string,
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  primary key (redisKey) not enforced  --Obtains the key value from Redis.
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'hash',
  'deploy-mode' = 'master-replica'
);

CREATE TABLE printSink (
  redisKey string,
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);

insert into printSink select * from redisSource;
```

5. Perform the following operations to view the data result in the **taskmanager.out** file:
   a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
   b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
   c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I(redisSource1,202103241606060001,appShop,2021-03-24 16:06:06,200.0,180.0,2021-03-24
16:10:06,0001,Alice,330106)
+I(redisSource,202103241000000001,webShop,2021-03-24 10:00:00,100.0,100.0,2021-03-24
10:02:03,0001,Alice,330106)
+I(redisSource2,202103251202020001,miniAppShop,2021-03-25 12:02:02,60.0,60.0,2021-03-25
12:03:00,0002,Bob,330110)
```

FAQ

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
Caused by: org.apache.flink.client.program.ProgramInvocationException: The main method caused an
error: RealLine:36;Usage of 'set' data-type and 'fields' schema syntax in source Redis connector with
multiple non-key column types. As 'set' in Redis is not sorted, it's not possible to map 'set's values to
table schema with different types.
```

A: If **data-type** is **set**, the data types of non-primary key fields in Flink are different. As a result, this error is reported. When **data-type** is **set**, the types of non-primary keys defined in Flink must be the same.

- Q: If **data-type** is **hash**, what are the differences between **schema-syntax** set to **fields** and that to **map**?

  A: When **schema-syntax** is set to **fields**, the hash value in the Redis key is assigned to the field with the same name in Flink. When **schema-syntax** is set to **map**, the hash key and hash value of each hash in Redis are put into a map, which represents the value of the corresponding Flink field. Specifically, this map contains all hash keys and hash values of a key in Redis.

  - For **fields**:

    i. Insert the following data into Redis:
    ```
    HMSET redisSource order_id 202103241000000001 order_channel webShop order_time "2021-03-24 10:00:00" pay_amount 100.00 real_pay 100.00 pay_time "2021-03-24 10:02:03" user_id 0001 user_name Alice area_id 330106
    ```

    ii. When **schema-syntax** is set to **fields**, use the following job script:
    ```
    CREATE TABLE redisSource (
      redisKey string,
      order_id string,
      order_channel string,
      order_time string,
      pay_amount double,
      real_pay double,
      pay_time string,
      user_id string,
      user_name string,
      area_id string,
      primary key (redisKey) not enforced
    ) WITH (
      'connector' = 'redis',
      'host' = 'RedisIP',
      'password' = 'RedisPassword',
      'data-type' = 'hash',
      'deploy-mode' = 'master-replica'
    );

    CREATE TABLE printSink (
      redisKey string,
      order_id string,
      order_channel string,
      order_time string,
      pay_amount double,
      real_pay double,
      pay_time string,
      user_id string,
      user_name string,
      area_id string
    ) WITH (
      'connector' = 'print'
    );

    insert into printSink select * from redisSource;
    ```

    iii. The job execution result is as follows:
    ```
    +I(redisSource,202103241000000001,webShop,2021-03-24 10:00:00,100.0,100.0,2021-03-24 10:02:03,0001,Alice,330106)
    ```

  - For **map**:

    i. Insert the following data into Redis:
    ```
    HMSET redisSource order_id 202103241000000001 order_channel webShop order_time "2021-03-24 10:00:00" pay_amount 100.00 real_pay 100.00 pay_time "2021-03-24 10:02:03" user_id 0001 user_name Alice area_id 330106
    ```

ii. When **schema-syntax** is set to **map**, use the following job script:

```
CREATE TABLE redisSource (
  redisKey string,
  order_result map<string, string>,
  primary key (redisKey) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'hash',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'map'
);

CREATE TABLE printSink (
  redisKey string,
  order_result map<string, string>
) WITH (
  'connector' = 'print'
);

insert into printSink select * from redisSource;
```

iii. The job execution result is as follows:

```
+I(redisSource,{user_id=0001, user_name=Alice, pay_amount=100.00, real_pay=100.00,
order_time=2021-03-24 10:00:00, area_id=330106, order_id=202103241000000001,
order_channel=webShop, pay_time=2021-03-24 10:02:03})
```

2.3.1.9 Upsert Kafka Source Table

Function

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.

As a source, the upsert-kafka connector produces a changelog stream, where each data record represents an update or delete event. More precisely, the value in a data record is interpreted as an UPDATE of the last value for the same key, if any (if a corresponding key does not exist yet, the UPDATE will be considered an INSERT). Using the table analogy, a data record in a changelog stream is interpreted as an UPSERT, also known as INSERT/UPDATE, because any existing row with the same key is overwritten. Also, null values are interpreted in a special way: A record with a null value represents a DELETE.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to Kafka clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- The Upsert Kafka always works in the upsert fashion and requires to define the primary key in the DDL. With the assumption that records with the same key should be ordered in the same partition, the primary key semantic on the changelog source means the materialized changelog is unique on the primary keys. The primary key definition will also control which fields should end up in Kafka's key.
- Because the connector is working in upsert mode, the last record on the same key will take effect when reading back as a source.
- For details about how to use data types, see section **Format**.

Syntax

```
create table kafkaSource(
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'upsert-kafka',
  'topic' = '',
  'properties.bootstrap.servers' = '',
  'key.format' = '',
  'value.format' = ''
);
```

Parameters

Table 2-13 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **upsert-kafka**.
topic	Yes	None	String	Kafka topic name.
properties.bootstrap.servers	Yes	None	String	Comma separated list of Kafka brokers.

Parameter	Mandatory	Default Value	Data Type	Description
key.format	Yes	None	String	Format used to deserialize and serialize the key part of Kafka messages. The key fields are specified by the **PRIMARY KEY** syntax. The following formats are supported: • csv • json • avro Refer to **Format** for more details and format parameters.
key.fields-prefix	No	None	String	Defines a custom prefix for all fields of the key format to avoid name clashes with fields of the value format. By default, the prefix is empty. If a custom prefix is defined, both the table schema and **key.fields** will work with prefixed names. When constructing the data type of the key format, the prefix will be removed and the non-prefixed names will be used within the key format. Note that this option requires that **value.fields-include** must be set to **EXCEPT_KEY**.
value.format	Yes	None	String	Format used to deserialize and serialize the value part of Kafka messages. The following formats are supported: • csv • json • avro Refer to **Format** for more details and format parameters.
value.fields-include	Yes	ALL	String	Controls which fields should appear in the value part. Possible values are: • **ALL**: All fields in the schema, including the primary key field, are included in the value part. • **EXCEPT_KEY**: All the fields of the table schema are included, except the primary key field.

Parameter	Mandatory	Default Value	Data Type	Description
properties.*	No	None	String	This option can set and pass arbitrary Kafka configurations. The suffix to **properties.** must match the parameter defined in **Kafka Configuration documentation**. Flink will remove the **properties.** key prefix and pass the transformed key and value to the underlying KafkaClient. For example, you can disable automatic topic creation via **'properties.allow.auto.create.topics' = 'false'**. But there are some configurations that do not support to set, because Flink will override them, for example, **'key.deserializer'** and **'value.deserializer'**.
ssl_auth_name	No	None	String	Name of datasource authentication of the Kafka_SSL type created on DLI. This configuration is used when SSL is configured for Kafka. Note: If only the SSL type is used, you need to set **properties.security.protocol** to **SSL**. If the SASL_SSL type is used, you need to set **properties.security.protocol** to **SASL_SSL**, **properties.sasl.mechanism** to **GSSAPI** or **PLAIN**, and **properties.sasl.jaas.config** to **org.apache.kafka.common.security.plain.PlainLoginModule required username=\"xxx\" password=\"xxx\";**.
krb_auth_name	No	None	String	Name of datasource authentication of the Kerberos type created on DLI. This configuration is used when SASL is configured for Kafka. Note: If the SASL_PLAINTEXT type and Kerberos authentication are used, you need to set **properties.sasl.mechanism** to **GSSAPI** and **properties.security.protocol** to **SASL_PLAINTEXT**.

Example

In this example, data is read from the Kafka data source and written to the Print result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE upsertKafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  PRIMARY KEY (order_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' =  'KafkaAddress1:KafkaPort,KafkAddress2:KafkaPort',
  'key.format' = 'csv',
  'value.format' = 'json'
);

CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  PRIMARY KEY (order_id) NOT ENFORCED
) WITH (
  'connector' = 'print'
);

INSERT INTO printSink
SELECT * FROM upsertKafkaSource;
```

4. Insert the following data to the specified topics in Kafka. (Note: Specify the key when inserting data to Kafka.)

```
{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", 
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202103251505050001", "order_channel":"qqShop", "order_time":"2021-03-25 15:05:05", 
"pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2021-03-25 15:10:00", "user_id":"0003", 
"user_name":"Cindy", "area_id":"330108"}
```

```
{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

5. Perform the following operations to view the output:
   a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
   b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
   c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

   The data result is as follows:

```
+I(202103251202020001,miniAppShop,2021-03-2512:02:02,60.0,60.0,2021-03-2512:03:00,0002,Bob,330110)
+I(202103251505050001,qqShop,2021-03-2515:05:05,500.0,400.0,2021-03-2515:10:00,0003,Cindy,330108)
-U(202103251202020001,miniAppShop,2021-03-2512:02:02,60.0,60.0,2021-03-2512:03:00,0002,Bob,330110)
+U(202103251202020001,miniAppShop,2021-03-2512:02:02,60.0,60.0,2021-03-2512:03:00,0002,Bob,330110)
```

FAQ

None

2.3.1.10 FileSystem Source Table

Function

This section describes the definition of the FileSystem source table, parameters used for creating the source table, and sample code.

Prerequisites

To create a FileSystem source table, an enhanced datasource connection is required. You can set security group rules as required when you configure the connection.

- You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Precautions

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- For details about how to use data types, see section **Format**.

Syntax

```
create table dataGenSource(
   amount int
) with (
   'connector' = 'filesystem',
   'path' = 'obs://longyuan/source-files',
   'format' = 'csv'
);
```

Parameters

Table 2-14 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	The value is fixed at **filesystem**.
path	Yes	None	String	OBS path
format	Yes	None	String	File format Available values are: **csv** and **parquet**

FAQ

None

2.3.2 Creating Result Tables

2.3.2.1 BlackHole Result Table

Function

The BlackHole connector allows for swallowing all input records. It is designed for high-performance testing and UDF output. It is not a substantive sink. The BlackHole result table is a built-in connector.

For example, if an error is reported when you register a result table of another type, but you are not sure whether it is caused by a system fault or an invalid setting of the **WITH** parameter for the result table, you can change the value of **connector** to **blackhole** and click **Run**. If no error is reported, the system is normal. You must check the settings of the **WITH** parameter.

Prerequisites

None

Precautions

When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.

Syntax

```
create table blackhole_table (
 attr_name attr_type (',' attr_name attr_type) *
) with (
 'connector' = 'blackhole'
);
```

Parameters

Table 2-15

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **blackhole**.

Example

The DataGen source table generates data, and the BlackHole result table receives the data.

```
create table datagenSource (
 user_id string,
 user_name string,
 user_age int
) with (
 'connector' = 'datagen',
 'rows-per-second'='1'
);
create table blackholeSink (
 user_id string,
 user_name string,
 user_age int
) with (
 'connector' = 'blackhole'
);
insert into blackholeSink select * from datagenSource;
```

2.3.2.2 ClickHouse Result Table

Function

DLI can output Flink job data to the ClickHouse database. ClickHouse is a column-based database oriented to online analysis and processing. It supports SQL query and provides good query performance. The aggregation analysis and query performance based on large and wide tables is excellent, which is one order of magnitude faster than other analytical databases. For details, see **Using ClickHouse from Scratch**.

Prerequisites

- You have established an enhanced datasource connection to ClickHouse and set the port in the security group rule of the ClickHouse cluster as needed.

  For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.

  For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- When you create a ClickHouse cluster for MRS, set the cluster version to MRS 3.1.0 or later and do not enable Kerberos authentication.
- The ClickHouse result table does not support table data deletion.
- Flink supports the following data types: string, tinyint, smallint, int, long, float, double, date, timestamp, decimal, and array.

  The array supports only the int, bigint, string, float, and double data types.

Syntax

```
create table clickhouseSink (
  attr_name attr_type
  (',' attr_name attr_type)*
)
with (
  'connector.type' = clickhouse,
  'connector.url' = '',
  'connector.table' = ''
);
```

Parameters

Table 2-16 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector.type	Yes	None	String	Result table type. Set this parameter to **clickhouse**.

Parameter	Mandatory	Default Value	Data Type	Description
connector.url	Yes	None	String	ClickHouse URL. Parameter format: **jdbc:clickhouse://***ClickHouseBalancer instance IP address***:***HTTP port number for ClickHouseBalancer instances***/***Database name* • IP address of a ClickHouseBalancer instance: Log in to the MRS console and choose **Clusters** > **Active Clusters** in the navigation pane. Click a cluster name, and choose **Components** > **ClickHouse** > **Instances** to obtain the business IP address of the ClickHouseBalancer instance. • HTTP port of a ClickHouseBalancer instance: Log in to the MRS console and choose **Clusters** > **Active Clusters** in the navigation pane. Click a cluster name, and choose **Components** > **ClickHouse** > **Service Configuration**. On the **Service Configuration** page, select **ClickHouseBalancer** from the **All Roles** drop-down list, search for **lb_http_port**, and obtain the parameter value. The default value is **21425**. • The database name is the name of the database created for the ClickHouse cluster.
connector.table	Yes	None	String	Name of the ClickHouse table to be created.

Parameter	Mandatory	Default Value	Data Type	Description
connector.driver	No	ru.yandex.clickhouse.ClickHouseDriver	String	Driver required for connecting to the database. • If this parameter is not specified during table creation, the driver automatically extracts the value from the ClickHouse URL. • If this parameter is specified during table creation, the value must be **ru.yandex.clickhouse.ClickHouseDriver**.
connector.username	No	None	String	Username for connecting to the ClickHouse database.
connector.password	No	None	String	Password for connecting to the ClickHouse database.
connector.write.flush.max-rows	No	5000	Integer	Maximum number of rows to be updated when data is written. The default value is **5000**.
connector.write.flush.interval	No	0	Duration	Interval for data update. The unit can be ms, milli, millisecond/s, sec, second/min, or minute. Value **0** indicates that data is not updated.
connector.write.max-retries	No	3	Integer	Maximum number of retries for writing data to the result table. The default value is **3**.

Example

In this example, data is from Kafka and inserted to table **order** in ClickHouse database **flink**. The procedure is as follows (the ClickHouse version is 21.3.4.25 in MRS):

1. Create an enhanced datasource connection in the VPC and subnet where ClickHouse and Kafka clusters locate, and bind the connection to the required Flink queue. For details, see **Enhanced Datasource Connections**.
2. Set ClickHouse and Kafka cluster security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the ClickHouse address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Use the ClickHouse client to connect to the ClickHouse server by referring to **Using ClickHouse from Scratch** and run the following command to query other environment parameters such as the cluster ID:

```
select cluster,shard_num,replica_num,host_name from system.clusters;
```

The following information is displayed:

```
┌─cluster───────┬────────┬─shard_num─┐
│ default_cluster │    1    │          1 │
│ default_cluster │    1    │          2 │
└───────────── ┴────────┴────────── ┘
```

4. Run the following command to create database **flink** on a node of the ClickHouse cluster based on the obtained cluster ID, for example, **default_cluster**:

```
CREATE DATABASE flink ON CLUSTER default_cluster;
```

5. Run the following command to create the ReplicatedMergeTree table named **order** on the node of cluster **default_cluster** and on database **flink**:

```
CREATE TABLE flink.order ON CLUSTER default_cluster(order_id String,order_channel
String,order_time String,pay_amount Float64,real_pay Float64,pay_time String,user_id
String,user_name String,area_id String) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/
flink/order', '{replica}')ORDER BY order_id;
```

6. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses the Kafka data source and the ClickHouse result table.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

create table clickhouseSink(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) with (
  'connector.type' = 'clickhouse',
  'connector.url' = 'jdbc:clickhouse://ClickhouseAddress:ClickhousePort/flink',
  'connector.table' = 'order',
  'connector.write.flush.max-rows' = '1'
);

insert into clickhouseSink select * from orders;
```

7. Connect to the Kafka cluster and insert the following test data into Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
```

```
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

8. Use the ClickHouse client to connect to the ClickHouse and run the following command to query the data written to table **order** in database **flink**:

```
select * from flink.order;
```

The query result is as follows:

```
202103241000000001 webShop 2021-03-24 10:00:00 100 100 2021-03-24 10:02:03 0001 Alice 330106

202103241606060001 appShop 2021-03-24 16:06:06 200 180 2021-03-24 16:10:06 0001 Alice 330106

202103251202020001 miniAppShop 2021-03-25 12:02:02 60 60 2021-03-25 12:03:00 0002 Bob 330110
```

FAQ

None

2.3.2.3 GaussDB(DWS) Result Table

Function

DLI outputs the Flink job output data to GaussDB(DWS). GaussDB(DWS) database kernel is compliant with PostgreSQL. The PostgreSQL database can store data of more complex types and deliver space information services, multi-version concurrent control (MVCC), and high concurrency. It applies to location applications, financial insurance, and e-Commerce.

GaussDB(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 GaussDB(DWS), see the **Data Warehouse Service Management Guide**.

Prerequisites

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- You have created a GaussDB(DWS) cluster. For details about how to create a GaussDB(DWS) cluster, see **Creating a Cluster** in the *Data Warehouse Service Management Guide*.
- You have created a GaussDB(DWS) database table.
- An enhanced datasource connection has been created for DLI to connect to GaussDB(DWS) clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- To use the upsert mode, you must define the primary key for both the GaussDB(DWS) result table and the GaussDB(DWS) table connected to the result table.
- If tables with the same name exist in different GaussDB(DWS) schemas, you need to specify the schemas in the Flink open source SQL statements.
- Before submitting a Flink job, you are advised to select **Save Job Log** and set the OBS bucket for saving job logs. This helps you view logs and locate faults when the job fails to be submitted or runs abnormally.
- If you use the gsjdbc4 driver for connection, set **driver** to **org.postgresql.Driver**. You can omit this parameter because the gsjdbc4 driver is the default one.

  For example, run the following statements to use the gsjdbc4 driver to write data to GaussDB(DWS) in upsert mode:

```
create table dwsSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
  'connector' = 'gaussdb',
  'url' = 'jdbc:postgresql://DwsAddress:DwsPort/DwsDatabase',
  'table-name' = 'car_info',
  'username' = 'DwsUserName',
  'password' = 'DwsPasswrod',
  'write.mode' = 'upsert'
);
```

- If you use the gsjdbc200 driver for connection, set **driver** to **com.huawei.gauss200.jdbc.Driver**.

  For example, run the following statements to write data to GaussDB(DWS) result table **test** that is in schema **ads_game_sdk_base**:

```
create table dwsSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
  'connector' = 'gaussdb',
  'table-name' = 'ads_game_sdk_base\".\"test',
  'driver' = 'com.huawei.gauss200.jdbc.Driver',
  'url' = 'jdbc:gaussdb://DwsAddress:DwsPort/DwsDatabase',
  'username' = 'DwsUserName',
  'password' = 'DwsPasswrod',
  'write.mode' = 'upsert'
);
```

Syntax

NOTE

Do not set all attributes in a GaussDB(DWS) result table to **PRIMARY KEY**.

```
create table dwsSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'gaussdb',
  'url' = '',
  'table-name' = '',
  'driver' = '',
  'username' = '',
  'password' = ''
);
```

Parameters

Table 2-17 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **gaussdb**.
url	Yes	None	String	JDBC connection address. If you use the gsjdbc4 driver, set the value in jdbc:postgresql://${ip}:${port}/${dbName} format. If you use the gsjdbc200 driver, set the value in jdbc:gaussdb://${ip}:${port}/${dbName} format.
table-name	Yes	None	String	Name of the table to be operated. If the GaussDB(DWS) table is in a schema, the format is **schema\".\"***Table name*. For details, see **FAQ**.
driver	No	org.postgresql.Driver	String	JDBC connection driver. The default value is **org.postgresql.Driver**. • If you use the gsjdbc4 driver for connection, set this parameter to **org.postgresql.Driver**. • If you use the gsjdbc200 driver for connection, set this parameter to **com.huawei.gauss200.jdbc.Driver**.
username	No	None	String	Username for GaussDB(DWS) database authentication. This parameter must be configured in pair with **password**.
password	No	None	String	Password for GaussDB(DWS) database authentication. This parameter must be configured in pair with **username**.

Parameter	Mandatory	Default Value	Data Type	Description
write.mode	No	None	String	Data write mode. The value can be **copy**, **insert**, or **upsert**. The default value is **upsert**. This parameter must be configured depending on **primary key**. ● If **primary key** is not configured, data can be appended in **copy** and **insert** modes. ● If **primary key** is configured, all the three modes are available. Note: GaussDB(DWS) does not support the update of distribution columns. The primary keys of columns to be updated must cover all distribution columns defined in the GaussDB(DWS) table.
sink.buffer-flush.max-rows	No	100	Integer	Maximum number of rows to buffer for each write request. It can improve the performance of writing data, but may increase the latency. You can set this parameter to **0** to disable it.
sink.buffer-flush.interval	No	1s	Duration	Interval for refreshing the buffer, during which data is refreshed by asynchronous threads. It can improve the performance of writing data to the database, but may increase the latency. You can set this parameter to **0** to disable it. Note: If **sink.buffer-flush.max-size** and **sink.buffer-flush.max-rows** are both set to **0** and the buffer refresh interval is configured, the buffer is asynchronously refreshed. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit).
sink.max-retries	No	3	Integer	Maximum number of write retries.

Parameter	Mandatory	Default Value	Data Type	Description
write.escape-string-value	No	false	Boolean	Whether to escape values of the string type. This parameter is used only when **write.mode** is set to **copy**.
pwd_auth_name	No	None	String	Name of datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.
key-by-before-sink	No	false	Boolean	Whether to partition by the specified primary key before the sink operator This parameter aims to solve the problem of interlocking between two subtasks when they acquire row locks based on the primary key from GaussDB(DWS), multiple concurrent writes occur, and **write.mode** is **upsert**. This happens when a batch of data written to the sink by multiple subtasks has more than one record with the same primary key, and the order of these records with the same primary key is inconsistent.

Example

In this example, data is read from the Kafka data source and written to the GaussDB(DWS) result table in insert mode. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where GaussDB(DWS) and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set GaussDB(DWS) and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the GaussDB(DWS) and Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Connect to the GaussDB(DWS) database and create a table named **dws_order**.
```
create table public.dws_order(
  order_id VARCHAR,
  order_channel VARCHAR,
  order_time VARCHAR,
  pay_amount FLOAT8,
  real_pay FLOAT8,
  pay_time VARCHAR,
  user_id VARCHAR,
```

```
user_name VARCHAR,
area_id VARCHAR);
```

4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses the Kafka data source and the GaussDB(DWS) result table.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE dwsSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'gaussdb',
  'url' = 'jdbc:postgresql://DWSAddress:DWSPort/DWSdbName',
  'table-name' = 'dws_order',
  'driver' = 'org.postgresql.Driver',
  'username' = 'DWSUserName',
  'password' = 'DWSPassword',
  'write.mode' = 'insert'
);

insert into dwsSink select * from kafkaSource;
```

5. Connect to the Kafka cluster and enter the following test data to Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}
```

6. Run the following SQL statement in GaussDB(DWS) to view the data result:

```
select * from dws_order
```

   The data result is as follows:

```
202103241000000001    webShop    2021-03-24 10:00:00    100.0    100.0    2021-03-24 10:02:03
0001    Alice    330106
```

FAQ

- Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
java.io.IOException: unable to open JDBC writer
...
Caused by: org.postgresql.util.PSQLException: The connection attempt failed.
...
Caused by: java.net.SocketTimeoutException: connect timed out
```

A: The datasource connection is not bound or the binding fails.

– To reconfigure datasource connections, refer to **Enhanced Datasource Connection**. Rectify the fault by referring to **DLI Failed to Connect to GaussDB(DWS) Through an Enhanced Datasource Connection**.

- Q: How can I configure a GaussDB(DWS) table that is in a schema?

  A: When GaussDB(DWS) table **test** is in schema **ads_game_sdk_base**, refer to the **'table-name'** parameter setting in the following example:

```
CREATE TABLE ads_rpt_game_sdk_realtime_ada_reg_user_pay_mm (
  ddate DATE,
  dmin TIMESTAMP(3),
  game_appkey VARCHAR,
  channel_id VARCHAR,
  pay_user_num_1m bigint,
  pay_amt_1m bigint,
  PRIMARY KEY (ddate, dmin, game_appkey, channel_id) NOT ENFORCED
) WITH (
  'connector' = 'gaussdb',
  'url' = 'jdbc:postgresql://<yourDwsAddress>:<yourDwsPort>/dws_bigdata_db',
  'table-name' = 'ads_game_sdk_base\".\"test',
  'username' = '<yourUsername>',
  'password' = '<yourPassword>',
  'write.mode' = 'upsert'
);
```

- Q: What can I do if a job is running properly but there is no data in GaussDB(DWS)?

  A: Check the following items:

  – Check whether the JobManager and TaskManager logs contain error information. To view logs, perform the following steps:

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

    ii. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.

    iii. Go to the folder of the date, find the folder whose name contains **taskmanager** or **jobmanager**, download the **taskmanager.out** or **jobmanager.out** file, and view result logs.

  – Check whether the datasource connection is correctly bound and whether a security group rule allows access of the queue.

  – Check whether the GaussDB(DWS) table to which data is to be written exists in multiple schemas. If it does, specify the schemas in the Flink job.

2.3.2.4 Elasticsearch Result Table

Function

DLI outputs 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.

For more information about CSS, see **Cloud Search Service User Guide**.

Prerequisites

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- You have created a cluster on CSS. For details about how to create a cluster, see **Creating a Cluster** in the *Cloud Search Service User Guide*
- An enhanced datasource connection has been created for DLI to connect to CSS, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- Currently, only CSS 7.X and later versions are supported. Version 7.6.2 is recommended.
- ICMP must be enabled for the security group inbound rules of the CSS cluster.
- For details about how to use data types, see section **Format**.
- Before submitting a Flink job, you are advised to select **Save Job Log** and set the OBS bucket for saving job logs. This helps you view logs and locate faults when the job fails to be submitted or runs abnormally.
- The Elasticsearch sink can work in either upsert mode or append mode, depending on whether a primary key is defined.
  - If a primary key is defined, the Elasticsearch sink works in upsert mode, which can consume queries containing UPDATE and DELETE messages.
  - If a primary key is not defined, the Elasticsearch sink works in append mode which can only consume queries containing INSERT messages.

  In the Elasticsearch result table, the primary key is used to calculate the Elasticsearch document ID. The document ID is a string of up to 512 bytes. It cannot have spaces. The Elasticsearch result table generates a document ID string for every row by concatenating all primary key fields in the order defined in the DDL using a key delimiter specified by **document-id.key-delimiter**. Certain types are not allowed as a primary key field as they do not have a good string representation, for example, BYTES, ROW, ARRAY, and MAP. If no primary key is specified, Elasticsearch will generate a document ID automatically.

- The Elasticsearch result table supports both static index and dynamic index.
  - If you want to have a static index, the index option value should be a plain string, such as **myusers**, all the records will be consistently written into the **myusers** index.
  - If you want to have a dynamic index, you can use **{field_name}** to reference a field value in the record to dynamically generate a target index. You can also use **{field_name|date_format_string}** to convert a field value of the TIMESTAMP, DATE, or TIME type into the format specified by **date_format_string**. **date_format_string** is compatible with Java's **DateTimeFormatter**. For example, if the option value is **myusers-{log_ts|yyyy-MM-dd}**, then a record with **log_ts** field value **2020-03-27 12:25:55** will be written into the **myusers-2020-03-27** index.

Syntax

```
create table esSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'elasticsearch-7',
  'hosts' = '',
  'index' = ''
);
```

Parameters

Table 2-18 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **elasticsearch-7**, indicating to connect to a cluster of Elasticsearch 7.x or later.
hosts	Yes	None	String	Host name of the cluster where Elasticsearch is located. Use semicolons (;) to separate multiple host names.
index	Yes	None	String	Elasticsearch index for every record. The index can be a static index (for example, **'myIndex'**) or a dynamic index (for example, **'index-{log_ts|yyyy-MM-dd}'**).

Parameter	Mandatory	Default Value	Data Type	Description
username	No	None	String	Username of the cluster where Elasticsearch locates. This parameter must be configured in pair with **password**.
password	No	None	String	Password of the cluster where Elasticsearch locates. This parameter must be configured in pair with **username**.
document-id.key-delimiter	No	_	String	Delimiter of composite primary keys. The default value is _.
failure-handler	No	fail	String	Failure handling strategy in case a request to Elasticsearch fails. Valid strategies are: ● **fail**: throws an exception if a request fails and thus causes a job failure. ● **ignore**: ignores failures and drops the request. ● **retry-rejected**: re-adds requests that have failed due to queue capacity saturation. ● **Custom class name**: for failure handling with an **ActionRequestFailureHandler** subclass.
sink.flush-on-checkpoint	No	true	Boolean	Whether to flush on checkpoint. If this parameter is set to **false**, the connector will not wait for all pending action requests to be acknowledged by Elasticsearch on checkpoints. Therefore, the connector does not provide any strong guarantees for at-least-once delivery of action requests.
sink.bulk-flush.max-actions	No	1000	Interger	Maximum number of buffered actions per bulk request. You can set this parameter to **0** to disable it.
sink.bulk-flush.max-size	No	2mb	MemorySize	Maximum size in memory of buffered actions per bulk request. It must be in MB granularity. You can set this parameter to **0** to disable it.

Parameter	Mandatory	Default Value	Data Type	Description
sink.bulk-flush.interval	No	1s	Duration	Interval for flushing buffered actions. You can set this parameter to **0** to disable it. Note: Both **sink.bulk-flush.max-size** and **sink.bulk-flush.max-actions** can be set to **0** with the flush interval set allowing for complete asynchronous processing of buffered actions.
sink.bulk-flush.backoff.strategy	No	DISABLED	String	Specifies how to perform retries if any flush actions failed due to a temporary request error. Valid strategies are: ● **DISABLED**: no retry performed, that is, fail after the first request error. ● **CONSTANT**: wait for backoff delay between retries. ● **EXPONENTIAL**: initially wait for backoff delay and increase exponentially between retries.
sink.bulk-flush.backoff.max-retries	No	8	Integer	Maximum number of backoff retries.
sink.bulk-flush.backoff.delay	No	50ms	Duration	Delay between each backoff attempt. For **CONSTANT** backoff, this is simply the delay between each retry. For **EXPONENTIAL** backoff, this is the initial base delay.
connection.max-retry-timeout	No	None	Duration	Maximum timeout between retries.
connection.path-prefix	No	None	String	Prefix string to be added to every REST communication, for example, **'/v1'**.
format	No	json	String	The Elasticsearch connector supports to specify a format. The format must produce a valid JSON document. By default, the built-in JSON format is used. Refer to **Format** for more details and format parameters.

Parameter	Mandatory	Default Value	Data Type	Description
pwd_auth_name	No	None	String	Datasource authentication name of the password type. ● Set this parameter only when datasource authentication of the CSS type is used. ● Set either **es_auth_name** or this parameter.

Example

In this example, data is read from the Kafka data source and written to the Elasticsearch result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where Elasticsearch and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set Elasticsearch and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Elasticsearch and Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Log in to Kibana of the Elasticsearch cluster, select Dev Tools, enter and execute the following statement to create an index whose value is **orders**:
```
PUT /orders
{
  "settings": {
    "number_of_shards": 1
  },
    "mappings": {
      "properties": {
        "order_id": {
          "type": "text"
        },
        "order_channel": {
          "type": "text"
        },
        "order_time": {
          "type": "text"
        },
        "pay_amount": {
          "type": "double"
        },
        "real_pay": {
          "type": "double"
        },
        "pay_time": {
          "type": "text"
        },
        "user_id": {
          "type": "text"
        },
        "user_name": {
          "type": "text"
        },
```

```
      "area_id": {
        "type": "text"
      }
    }
  }
}
```

4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  "format" = "json"
);

CREATE TABLE elasticsearchSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'elasticsearch-7',
  'hosts' = 'ElasticsearchAddress:ElasticsearchPort',
  'index' = 'orders'
);

insert into elasticsearchSink select * from kafkaSource;
```

5. Connect to the Kafka cluster and insert the following test data into Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}
```

6. Enter the following statement in Kibana of the Elasticsearch cluster and view the result:

```
GET orders/_search
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
```

```
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "orders",
        "_type" : "_doc",
        "_id" : "ae7wpH4B1dV9conjpXeB",
        "_score" : 1.0,
        "_source" : {
          "order_id" : "202103241000000001",
          "order_channel" : "webShop",
          "order_time" : "2021-03-24 10:00:00",
          "pay_amount" : 100.0,
          "real_pay" : 100.0,
          "pay_time" : "2021-03-24 10:02:03",
          "user_id" : "0001",
          "user_name" : "Alice",
          "area_id" : "330106"
        }
      },
      {
        "_index" : "orders",
        "_type" : "_doc",
        "_id" : "au7xpH4B1dV9conjn3er",
        "_score" : 1.0,
        "_source" : {
          "order_id" : "202103241606060001",
          "order_channel" : "appShop",
          "order_time" : "2021-03-24 16:06:06",
          "pay_amount" : 200.0,
          "real_pay" : 180.0,
          "pay_time" : "2021-03-24 16:10:06",
          "user_id" : "0001",
          "user_name" : "Alice",
          "area_id" : "330106"
        }
      }
    ]
  }
}
```


2.3.2.5 HBase Result Table

Function

DLI outputs the job data to HBase. HBase is a column-oriented distributed cloud storage system that features enhanced reliability, excellent performance, and elastic scalability. It applies to the storage of massive amounts of data and distributed computing. You can use HBase to build a storage system capable of storing TB- or even PB-level data. With HBase, you can filter and analyze data with ease and get responses in milliseconds, rapidly mining data value. Structured and semi-structured key-value data can be stored, including messages, reports, recommendation data, risk control data, logs, and orders. With DLI, you can write massive volumes of data to HBase at a high speed and with low latency.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- If MRS HBase is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.

  For details, see **Modifying Host Information** in the *Data Lake Insight User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- The column families in created HBase result table must be declared as the ROW type, the field names map the column family names, and the nested field names map the column qualifier names. There is no need to declare all the families and qualifiers in the schema. Users can declare what is used in the query. Except the ROW type fields, the single atomic type field (for example, STRING or BIGINT) will be recognized as the HBase rowkey. The rowkey field can be an arbitrary name, but should be quoted using backticks if it is a reserved keyword.

Syntax

```
create table hbaseSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  ','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
) with (
  'connector' = 'hbase-2.2',
  'table-name' = '',
  'zookeeper.quorum' = ''
);
```

Parameters

Table 2-19 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **hbase-2.2**.
table-name	Yes	None	String	Name of the HBase table to connect.
zookeeper.quorum	Yes	None	String	HBase ZooKeeper instance information, in the format of ZookeeperAddress:ZookeeperPort. The following uses an MRS HBase cluster as an example to describe how to obtain the IP address and port number of ZooKeeper used by this parameter: ● On MRS Manager, choose **Cluster** and click the name of the desired cluster. Choose **Services** > **ZooKeeper** > **Instance**, and obtain the IP address of the ZooKeeper instance. ● On MRS Manager, choose **Cluster** and click the name of the desired cluster. Choose **Services** > **ZooKeeper** > **Configurations** > **All Configurations**, search for the **clientPort** parameter, and obtain its value, that is, the ZooKeeper port number.
zookeeper.znode.parent	No	/hbase	String	Root directory in ZooKeeper. The default value is **/hbase**.
null-string-literal	No	null	String	Representation for null values for string fields. The HBase sink encodes/decodes empty bytes as null values for all types except the string type.
sink.buffer-flush.max-size	No	2mb	MemorySize	Maximum size in memory of buffered rows for each write request. This can improve performance for writing data to the HBase database, but may increase the latency. You can set this parameter to **0** to disable it.

Parameter	Mandatory	Default Value	Data Type	Description
sink.buffer-flush.max-rows	No	1000	Integer	Maximum number of rows to buffer for each write request. This can improve performance for writing data to the HBase database, but may increase the latency. You can set this parameter to **0** to disable it.
sink.buffer-flush.interval	No	1s	Duration	Interval for refreshing the buffer, during which data is refreshed by asynchronous threads. This can improve performance for writing data to the HBase database, but may increase the latency. You can set this parameter to **0** to disable it. Note: If **sink.buffer-flush.max-size** and **sink.buffer-flush.max-rows** are both set to **0** and the buffer refresh interval is configured, the buffer is asynchronously refreshed. The format is *{length value}{time unit label}*, for example, **123ms, 321s**. The supported time units include **d**, **h**, **min**, **s**, and **ms** (default unit).
sink.parallelism	No	None	Integer	Defines the parallelism for the HBase sink operator. By default, the parallelism is determined by the framework: using the same parallelism as the upstream join operator.
krb_auth_name	No	None	String	Name of datasource authentication of the Kerberos type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.

Data Type Mapping

HBase stores all data as byte arrays. The data needs to be serialized and deserialized during read and write operations.

When serializing and de-serializing, Flink HBase connector uses utility class **org.apache.hadoop.hbase.util.Bytes** provided by HBase (Hadoop) to convert Flink data types to and from byte arrays.

Flink HBase connector encodes null values to empty bytes, and decode empty bytes to null values for all data types except the string type. For the string type, the null literal is determined by the **null-string-literal** option.

Table 2-20 Data type mapping

Flink SQL Type	HBase Conversion
CHAR / VARCHAR / STRING	byte[] toBytes(String s) String toString(byte[] b)
BOOLEAN	byte[] toBytes(boolean b) boolean toBoolean(byte[] b)
BINARY / VARBINARY	Returns byte[] as is.
DECIMAL	byte[] toBytes(BigDecimal v) BigDecimal toBigDecimal(byte[] b)
TINYINT	new byte[] { val } bytes[0] // returns first and only byte from bytes
SMALLINT	byte[] toBytes(short val) short toShort(byte[] bytes)
INT	byte[] toBytes(int val) int toInt(byte[] bytes)
BIGINT	byte[] toBytes(long val) long toLong(byte[] bytes)
FLOAT	byte[] toBytes(float val) float toFloat(byte[] bytes)
DOUBLE	byte[] toBytes(double val) double toDouble(byte[] bytes)
DATE	Stores the number of days since epoch as an int value.
TIME	Stores the number of milliseconds of the day as an int value.
TIMESTAMP	Stores the milliseconds since epoch as a long value.
ARRAY	Not supported
MAP / MULTISET	Not supported

Flink SQL Type	HBase Conversion
ROW	Not supported

Example

In this example, data is read from the Kafka data source and written to the HBase result table. The procedure is as follows (the HBase versions used in this example are 1.3.1 and 2.2.3):

1. Create an enhanced datasource connection in the VPC and subnet where HBase and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**. Add MRS host information for the enhanced datasource connection**Modifying Host Information**.
2. Set HBase and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the HBase and Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Use the HBase shell to create HBase table **order** that has only one column family **detail**. For details, see **Using HBase from Scratch**.
```
create 'order', {NAME => 'detail'}
```
4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job. The job script uses Kafka as the data source and HBase as the result table (the Rowkey is **order_id** and the column family name is **detail**).

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**
```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

create table hbaseSink(
  order_id string,
  detail Row(
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
    pay_time string,
    user_id string,
```

```
    user_name string,
    area_id string)
) with (
  'connector' = 'hbase-2.2',
  'table-name' = 'order',
  'zookeeper.quorum' = 'ZookeeperAddress:ZookeeperPort',
  'sink.buffer-flush.max-rows' = '1'
);

insert into hbaseSink select order_id,
Row(order_channel,order_time,pay_amount,real_pay,pay_time,user_id,user_name,area_id) from orders;
```

5. Connect to the Kafka cluster and enter the following data to Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

6. Run the following statement on the HBase shell to view the data result:

```
 scan 'order'
```

The data result is as follows:

```
202103241000000001    column=detail:area_id, timestamp=2021-12-16T21:30:37.954, value=330106

202103241000000001    column=detail:order_channel, timestamp=2021-12-16T21:30:37.954,
value=webShop

202103241000000001    column=detail:order_time, timestamp=2021-12-16T21:30:37.954,
value=2021-03-24 10:00:00

202103241000000001    column=detail:pay_amount, timestamp=2021-12-16T21:30:37.954, value=@Y
\x00\x00\x00\x00\x00\x00

202103241000000001    column=detail:pay_time, timestamp=2021-12-16T21:30:37.954,
value=2021-03-24 10:02:03

202103241000000001    column=detail:real_pay, timestamp=2021-12-16T21:30:37.954, value=@Y
\x00\x00\x00\x00\x00\x00

202103241000000001    column=detail:user_id, timestamp=2021-12-16T21:30:37.954, value=0001

202103241000000001    column=detail:user_name, timestamp=2021-12-16T21:30:37.954, value=Alice

202103241606060001    column=detail:area_id, timestamp=2021-12-16T21:30:44.842, value=330106

202103241606060001    column=detail:order_channel, timestamp=2021-12-16T21:30:44.842,
value=appShop

202103241606060001    column=detail:order_time, timestamp=2021-12-16T21:30:44.842,
value=2021-03-24 16:06:06

202103241606060001    column=detail:pay_amount, timestamp=2021-12-16T21:30:44.842, value=@i
\x00\x00\x00\x00\x00\x00

202103241606060001    column=detail:pay_time, timestamp=2021-12-16T21:30:44.842,
value=2021-03-24 16:10:06

202103241606060001    column=detail:real_pay, timestamp=2021-12-16T21:30:44.842, value=@f
\x80\x00\x00\x00\x00\x00

202103241606060001    column=detail:user_id, timestamp=2021-12-16T21:30:44.842, value=0001

202103241606060001    column=detail:user_name, timestamp=2021-12-16T21:30:44.842, value=Alice
```

```
202103251202020001   column=detail:area_id, timestamp=2021-12-16T21:30:52.181, value=330110

202103251202020001   column=detail:order_channel, timestamp=2021-12-16T21:30:52.181, 
value=miniAppShop

202103251202020001   column=detail:order_time, timestamp=2021-12-16T21:30:52.181, 
value=2021-03-25 12:02:02

202103251202020001   column=detail:pay_amount, timestamp=2021-12-16T21:30:52.181, value=@N
\x00\x00\x00\x00\x00\x00

202103251202020001   column=detail:pay_time, timestamp=2021-12-16T21:30:52.181, 
value=2021-03-25 12:03:00

202103251202020001   column=detail:real_pay, timestamp=2021-12-16T21:30:52.181, value=@N
\x00\x00\x00\x00\x00\x00

202103251202020001   column=detail:user_id, timestamp=2021-12-16T21:30:52.181, value=0002

202103251202020001   column=detail:user_name, timestamp=2021-12-16T21:30:52.181, value=Bob
```

FAQ

Q: What should I do if the Flink job execution fails and the log contains the following error information?

```
org.apache.zookeeper.ClientCnxn$SessionTimeoutException: Client session timed out, have not heard from 
server in 90069ms for connection id 0x0
```

A: The datasource connection is not bound or the binding fails. Configure the datasource connection by referring to **Enhanced Datasource Connection** or configure the security group of the Kafka cluster to allow access from the DLI queue.

2.3.2.6 JDBC Result Table

Function

DLI outputs the Flink job output data to RDS through the JDBC result table.

Prerequisites

- An enhanced datasource connection with the instances has been established, so that you can configure security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- The connector operates in upsert mode if the primary key was defined; otherwise, the connector operates in append mode.
  - In upsert mode, Flink will insert a new row or update the existing row according to the primary key. Flink can ensure the idempotence in this way. To guarantee the output result is as expected, it is recommended to define a primary key for the table.
  - In append mode, Flink will interpret all records as INSERT messages. The INSERT operation may fail if a primary key or unique constraint violation happens in the underlying database.

Syntax

```
create table jdbcSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'jdbc',
  'url' = '',
  'table-name' = '',
  'driver' = '',
  'username' = '',
  'password' = ''
);
```

Parameters

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **jdbc**.
url	Yes	None	String	Database URL.
table-name	Yes	None	String	Name of the table where the data will be read from the database.
driver	No	None	String	Driver required for connecting to the database. If you do not set this parameter, it will be automatically derived from the URL.
username	No	None	String	Database authentication username. This parameter must be configured in pair with **password**.

Parameter	Mandatory	Default Value	Data Type	Description
password	No	None	String	Database authentication password. This parameter must be configured in pair with **username**.
sink.buffer-flush.max-rows	No	100	Integer	Maximum number of rows to buffer for each write request. It can improve the performance of writing data, but may increase the latency. You can set this parameter to **0** to disable it.
sink.buffer-flush.interval	No	1s	Duration	Interval for refreshing the buffer, during which data is refreshed by asynchronous threads. It can improve the performance of writing data, but may increase the latency. You can set this parameter to **0** to disable it. Note: If **sink.buffer-flush.max-rows** is set to **0** and the buffer refresh interval is configured, the buffer is asynchronously refreshed. The format is *{length value}{time unit label}*, for example, **123ms, 321s**. The supported time units include **d**, **h**, **min**, **s**, and **ms** (default unit).
sink.max-retries	No	3	Integer	Maximum number of retries if writing records to the database failed.
pwd_auth_name	No	None	String	Name of datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.

Data Type Mapping

Table 2-21 Data type mapping

MySQL Type	PostgreSQL Type	Flink SQL Type
TINYINT	-	TINYINT

MySQL Type	PostgreSQL Type	Flink SQL Type
SMALLINT TINYINT UNSIGNED	SMALLINT INT2 SMALLSERIAL SERIAL2	SMALLINT
INT MEDIUMINT SMALLINT UNSIGNED	INTEGER SERIAL	INT
BIGINT INT UNSIGNED	BIGINT BIGSERIAL	BIGINT
BIGINT UNSIGNED	-	DECIMAL(20, 0)
BIGINT	BIGINT	BIGINT
FLOAT	REAL FLOAT4	FLOAT
DOUBLE DOUBLE PRECISION	FLOAT8 DOUBLE PRECISION	DOUBLE
NUMERIC(p, s) DECIMAL(p, s)	NUMERIC(p, s) DECIMAL(p, s)	DECIMAL(p, s)
BOOLEAN TINYINT(1)	BOOLEAN	BOOLEAN
DATE	DATE	DATE
TIME [(p)]	TIME [(p)] [WITHOUT TIMEZONE]	TIME [(p)] [WITHOUT TIMEZONE]
DATETIME [(p)]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]
CHAR(n) VARCHAR(n) TEXT	CHAR(n) CHARACTER(n) VARCHAR(n) CHARACTER VARYING(n) TEXT	STRING

MySQL Type	PostgreSQL Type	Flink SQL Type
BINARY VARBINARY BLOB	BYTEA	BYTES
-	ARRAY	ARRAY

Example

In this example, Kafka is used to send data, and Kafka data is written to the MySQL database through the JDBC result table.

1. Create an enhanced datasource connection in the VPC and subnet where MySQL and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set MySQL and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the MySQL and Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Log in to the MySQL database and create table **orders** in database **flink**.

```
CREATE TABLE `flink`.`orders` (
    `order_id` VARCHAR(32) NOT NULL,
    `order_channel` VARCHAR(32) NULL,
    `order_time` VARCHAR(32) NULL,
    `pay_amount` DOUBLE UNSIGNED NOT NULL,
    `real_pay` DOUBLE UNSIGNED NULL,
    `pay_time` VARCHAR(32) NULL,
    `user_id` VARCHAR(32) NULL,
    `user_name` VARCHAR(32) NULL,
    `area_id` VARCHAR(32) NULL,
    PRIMARY KEY (`order_id`)
)    ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4
    COLLATE = utf8mb4_general_ci;
```

4. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
```

```
  'format' = 'json'
);

CREATE TABLE jdbcSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'jdbc',
  'url? = 'jdbc:mysql://MySQLAddress:MySQLPort/flink',-- flink is the MySQL database where the orders table locates.
  'table-name' = 'orders',
  'username' = 'MySQLUsername',
  'password' = 'MySQLPassword',
  'sink.buffer-flush.max-rows' = '1'
);

insert into jdbcSink select * from kafkaSource;
```

5. Connect to the Kafka cluster and send the following test data to the Kafka topics:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}
```

6. Run the SQL statement in the MySQL database to view data in the table:

```
select * from orders;
```

The following is an example of the result (note that the following data is replicated from the MySQL database but not the data style in the MySQL database):

```
202103241000000001,webShop,2021-03-24 10:00:00,100.0,100.0,2021-03-24 10:02:03,0001,Alice,330106
202103241606060001,appShop,2021-03-24 16:06:06,200.0,180.0,2021-03-24 16:10:06,0001,Alice,330106
```

FAQ

None

2.3.2.7 Kafka Result Table

Function

DLI outputs the Flink job output data to Kafka through the Kafka result table.

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

- You have created a Kafka cluster.

- An enhanced datasource connection has been created for DLI to connect to Kafka clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- For details about how to use data types, see section **Format**.

Syntax

```
create table kafkaSink(
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'kafka',
  'topic' = '',
  'properties.bootstrap.servers' = '',
  'format' = ''
);
```

Parameters

Table 2-22 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	string	Connector to be used. Set this parameter to **kafka**.
topic	Yes	None	string	Topic name of the Kafka result table.
properties.bootstrap.servers	Yes	None	string	Kafka broker address. The value is in the format of **host:port,host:port,host:port**. Multiple **host:port** pairs are separated with commas (,).

Parameter	Mandatory	Default Value	Data Type	Description
format	Yes	None	string	Format used by the Flink Kafka connector to serialize Kafka messages. Either this parameter or the **value.format** parameter is required. The following formats are supported: ● csv ● json ● avro Refer to **Format** for more details and format parameters.
topic-pattern	No	None	String	Regular expression for matching the Kafka topic name. Only one of **topic** and **topic-pattern** can be specified. Example: 'topic.*' '(topic-c|topic-d)' '(topic-a|topic-b|topic-\\d*)' '(topic-a|topic-b|topic-[0-9]*)'
properties.*	No	None	String	This parameter can set and pass arbitrary Kafka configurations. Note: ● Suffix names must match the configuration key defined in **Apache Kafka**. For example, you can disable automatic topic creation via **'properties.allow.auto.create.topics' = 'false'**. ● Some configurations are not supported, for example, **'key.deserializer'** and **'value.deserializer'**.

Parameter	Mandatory	Default Value	Data Type	Description
key.format	No	None	String	Format used to deserialize and serialize the key part of Kafka messages. Note: • If a key format is defined, the **key.fields** parameter is required as well. Otherwise, the Kafka records will have an empty key. • Possible values are: csv json avro debezium-json canal-json maxwell-json avro-confluent raw Refer to **Format** for more details and format parameters.
key.fields	No	[]	List<String>	Defines the columns in the table as the list of keys. This parameter must be configured in pair with **key.format**. This parameter is left empty by default. Therefore, no key is defined. The format is like **field1;field2**.
key.fields-prefix	No	None	String	Defines a custom prefix for all fields of the key format to avoid name clashes with fields of the value format.

Parameter	Mandatory	Default Value	Data Type	Description
value.format	Yes	None	String	Format used to deserialize and serialize the value part of Kafka messages. Note: • Either this parameter or the **format** parameter is required. If two parameters are configured, a conflict occurs. • Refer to **Format** for more details and format parameters.
value.fields-include	No	ALL	Enum Possible values: [ALL, EXCEPT_KEY]	Whether to contain the key field when parsing the message body. Possible values are: • **ALL** (default): All defined fields are included in the value of Kafka messages. • **EXCEPT_KEY**: All the fields except those defined by **key.fields** are included in the value of Kafka messages.
sink.partitioner	No	None	string	Mapping from Flink's partitions into Kafka's partitions. Valid values are as follows: • **fixed** (default): Each Flink partition ends up in at most one Kafka partition. • **round-robin**: A Flink partition is distributed to Kafka partitions in a round-robin manner. • **Custom FlinkKafkaPartitioner subclass**: If **fixed** and **round-robin** do not meet your requirements, you can create subclass **FlinkKafkaPartitioner** to customize the partition mapping, for example, **org.mycompany.MyPartitioner**.

Parameter	Mandatory	Default Value	Data Type	Description
sink.semantic	No	at-least-once	String	Defines the delivery semantic for the Kafka sink. Valid values are as follows: ● at-least-once ● exactly-once ● none
sink.parallelism	No	None	Integer	Defines the parallelism of the Kafka sink operator. By default, the parallelism is determined by the framework using the same parallelism of the upstream chained operator.
ssl_auth_name	No	None	String	Name of datasource authentication of the Kafka_SSL type created on DLI. This configuration is used when SSL is configured for Kafka. Note: If only the SSL type is used, you need to set **properties.security.protocol** to **SSL**. If the SASL_SSL type is used, you need to set **properties.security.protocol** to **SASL_SSL**, **properties.sasl.mechanism** to **GSSAPI** or **PLAIN**, and **properties.sasl.jaas.config** to **org.apache.kafka.common.security.plain.PlainLoginModule required username=\"xxx\" password=\"xxx\";**.
krb_auth_name	No	None	String	Name of datasource authentication of the Kerberos type created on DLI. This configuration is used when SASL is configured for Kafka. Note: If the SASL_PLAINTEXT type and Kerberos authentication are used, you need to set **properties.sasl.mechanism** to **GSSAPI** and **properties.security.protocol** to **SASL_PLAINTEXT**.

Example (SASL_SSL Disabled for the Kafka Cluster)

In this example, data is read from a Kafka topic and written to another using a Kafka result table.

1. Create an enhanced datasource connection in the VPC and subnet where Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  "format" = "json"
);

CREATE TABLE kafkaSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaSinkTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  "format" = "json"
);

insert into kafkaSink select * from kafkaSource;
```

4. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

```
{"order_id":"202103241000000001","order_channel":"webShop","order_time":"2021-03-24 10:00:00","pay_amount":100.0,"real_pay":100.0,"pay_time":"2021-03-24 10:02:03","user_id":"0001","user_name":"Alice","area_id":"330106"}
```

```
{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106"}
```

5. Connect to the Kafka cluster and read data from the sink topic of Kafka.

```
{"order_id":"202103241000000001","order_channel":"webShop","order_time":"2021-03-24
10:00:00","pay_amount":100.0,"real_pay":100.0,"pay_time":"2021-03-24
10:02:03","user_id":"0001","user_name":"Alice","area_id":"330106"}

{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106"}
```

Example (SASL_SSL Enabled for the Kafka Cluster)

- **Example 1: Enable SASL_SSL authentication for the DMS cluster.**

  Create a Kafka cluster for DMS, enable SASL_SSL, download the SSL certificate, and upload the downloaded certificate **client.jks** to an OBS bucket.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:9093,xx:9093,xx:9093',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.connector.auth.open' = 'true',
  'properties.ssl.truststore.location' = 'obs://xx/xx.jks',  -- Location where the user uploads the
certificate to
  'properties.sasl.mechanism' = 'PLAIN',  --  Value format: SASL_PLAINTEXT
  'properties.security.protocol' = 'SASL_SSL',
  'properties.sasl.jaas.config' = 'org.apache.kafka.common.security.plain.PlainLoginModule required
username=\"xx\" password=\"xx\";', -- Account and password set when the Kafka cluster is created
  "format" = "json"
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:9093,xx:9093,xx:9093',
  'properties.connector.auth.open' = 'true',
  'properties.ssl.truststore.location' = 'obs://xx/xx.jks',
  'properties.sasl.mechanism' = 'PLAIN',
  'properties.security.protocol' = 'SASL_SSL',
  'properties.sasl.jaas.config' = 'org.apache.kafka.common.security.plain.PlainLoginModule required
username=\"xx\" password=\"xx\";',
  "format" = "json"
);
```

```
insert into ordersSink select * from ordersSource;
```

- **Example 2: Enable Kafka SASL_SSL authentication for the MRS cluster.**
  - Enable Kerberos authentication for the MRS cluster.
  - Click the **Components** tab and click **Kafka**. In the displayed page, click the **Service Configuration** tab, locate the **security.protocol**, and set it to **SASL_SSL**.
  - Download the user credential. Log in to the FusionInsight Manager of the MRS cluster and choose **System** > **Permission** > **User**. Locate the row that contains the target user, click **More**, and select **Download Authentication Credential**.

    Obtain the **truststore.jks** file using the authentication credential and store the credential and **truststore.jks** file in OBS.
  - If "Message stream modified (41)" is displayed, the JDK version may be incorrect. Change the JDK version in the sample code to a version earlier than 8u_242 or delete the **renew_lifetime = 0m** configuration item from the **krb5.conf** configuration file.
  - Set the port to the **sasl_ssl.port** configured in the Kafka service configuration.
  - In the following statements, set **security.protocol** to **SASL_SSL**.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:21009,xx:21009',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx', --Username
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.security.protocol' = 'SASL_SSL',
  'properties.ssl.truststore.location' = 'obs://xx/truststore.jks',
  'properties.ssl.truststore.password' = 'xx',  -- Password set for generating truststore.jks
  'properties.sasl.mechanism' = 'GSSAPI',
  "format" = "json"
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
```

```
  'properties.bootstrap.servers' = 'xx:21009,xx:21009',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx',
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.ssl.truststore.location' = 'obs://xx/truststore.jks',
  'properties.ssl.truststore.password' = 'xx',
  'properties.security.protocol' = 'SASL_SSL',
  'properties.sasl.mechanism' = 'GSSAPI',
  "format" = "json"
);

insert into ordersSink select * from ordersSource;
```

- **Example 3: Enable Kerberos SASL_PAINTEXT authentication for the MRS cluster**
  – Enable Kerberos authentication for the MRS cluster.
  – Click the **Components** tab and click **Kafka**. In the displayed page, click the **Service Configuration** tab, locate the **security.protocol**, and set it to **SASL_PLAINTEXT**.
  – Log in to the FusionInsight Manager of the MRS cluster and download the user credential. Choose **System** > **Permission** > **User**. Locate the row that contains the target user, choose **More** > **Download Authentication Credential**. Upload the credential to OBS.
  – If error message "Message stream modified (41)" is displayed, the JDK version may be incorrect. Change the JDK version in the sample code to a version earlier than 8u_242 or delete the **renew_lifetime = 0m** configuration item from the **krb5.conf** configuration file.
  – Set the port to the **sasl.port** configured in the Kafka service configuration.
  – In the following statements, set **security.protocol** to **SASL_PLAINTEXT**.

```
CREATE TABLE ordersSources (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:21007,xx:21007',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx',
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  "format" = "json"
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
```

```
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:21007,xx:21007',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.connector.auth.open' = 'true',
  'properties.connector.kerberos.principal' = 'xx',
  'properties.connector.kerberos.krb5' = 'obs://xx/krb5.conf',
  'properties.connector.kerberos.keytab' = 'obs://xx/user.keytab',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  "format" = "json"
);

insert into ordersSink select * from ordersSource;
```

- **Example 4: Use SSL for the MRS cluster**
  - Do not enable Kerberos authentication for the MRS cluster.
  - Download the user credential. Log in to the FusionInsight Manager of the MRS cluster and choose **System** > **Permission** > **User**. Locate the row that contains the target user, click **More**, and select **Download Authentication Credential**.

    Obtain the **truststore.jks** file using the authentication credential and store the credential and **truststore.jks** file in OBS.
  - Set the port to the **ssl.port** configured in the Kafka service configuration.
  - In the following statements, set **security.protocol** to **SSL**.
  - Set **ssl.mode.enable** to **true**.

```
CREATE TABLE ordersSource (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'xx',
  'properties.bootstrap.servers' = 'xx:9093,xx:9093,xx:9093',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'properties.connector.auth.open' = 'true',
  'properties.ssl.truststore.location' = 'obs://xx/truststore.jks',
  'properties.ssl.truststore.password' = 'xx',  -- Password set for generating truststore.jks
  'properties.security.protocol' = 'SSL',
  "format" = "json"
);

CREATE TABLE ordersSink (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
```

```
  area_id string
) WITH (
  'connector' = 'print'
);

insert into ordersSink select * from ordersSource;
```

2.3.2.8 Print Result Table

Function

The Print connector is used to print output data to the error file or TaskManager file, making it easier for you to view the result in code debugging.

Prerequisites

None

Precautions

- The Print result table supports the following output formats:

Print	Condition 1	Condition 2
Identifier:Task ID> Output data	A print identifier prefix must be provided. That is, you must specify **print-identifier** in the **WITH** parameter when creating the Print result table.	parallelism > 1
Identifier> Output data	A print identifier prefix must be provided. That is, you must specify **print-identifier** in the **WITH** parameter when creating the Print result table.	parallelism == 1
Task ID> Output data	A print identifier prefix is not needed. That is, you do not specify **print-identifier** in the **WITH** parameter when creating the Print result table.	parallelism > 1
Output data	A print identifier prefix is not needed. That is, you do not specify **print-identifier** in the **WITH** parameter when creating the Print result table.	parallelism == 1

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.

Syntax

```
create table printSink (
  attr_name attr_type
  (',' attr_name attr_type) *
  (',' PRIMARY KEY (attr_name,...) NOT ENFORCED)
) with (
  'connector' = 'print',
  'print-identifier' = '',
  'standard-error' = ''
);
```

Parameters

Table 2-23 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **print**.
print-identifier	No	None	String	Message that identifies print and is prefixed to the output of the value.
standard-error	No	false	Boolean	The value can be only **true** or **false**. The default value is **false**. • If the value is **true**, data is output to the error file of the TaskManager. • If the value is **false**, data is output to the **out** file of the TaskManager.

Example

Create a Flink OpenSource SQL job. Run the following script to generate random data through the DataGen table and output the data to the Print result table.

When you create a job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.

```
create table dataGenSource(
  user_id string,
  amount int
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1', --Generates a piece of data per second.
  'fields.user_id.kind' = 'random', --Specifies a random generator for the user_id field.
  'fields.user_id.length' = '3' --Limits the length of user_id to 3.
);

create table printSink(
  user_id string,
```

```
  amount int
) with (
  'connector' = 'print'
);

insert into printSink select * from dataGenSource;
```

After the job is submitted, the job status changes to **Running**. You can perform the following operations to view the output result:

- Method 1:
  a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  b. Locate the row that contains the target Flink job, click **More** in the **Operation** column, and select **FlinkUI**.
  c. On the Flink UI, choose **Task Managers**, click the task name, and select **Stdout** to view job logs.
- Method 2: If you select **Save Job Log** on the **Running Parameters** tab before submitting the job, perform the following operations:
  a. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  b. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
  c. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

2.3.2.9 Redis Result Table

Function

DLI outputs the Flink job output data to Redis. Redis is a key-value storage system that supports multiple types of data structures. It can be used in scenarios such as caching, event publish/subscribe, and high-speed queuing. Redis supports direct read/write of strings, hashes, lists, queues, and sets. Redis works with in-memory datasets and provides persistence. For more information about Redis, visit **https://redis.io/**.

Prerequisites

- An enhanced datasource connection with Redis has been established, so that you can configure security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- If the Redis key field is not defined in the statement for creating the Redis result table, the generated UUID is used as the key.
- To specify a key in Redis, you need to define a primary key in the Redis result table of Flink. The value of the primary key is the Redis key.
- If the primary key defined for the Redis result table, it cannot be a composite primary key and only can be one field.
- Constraints on **schema-syntax**:
  - If **schema-syntax** is **map** or **array**, there can be only one non-primary key and it must be of the same **map** or **array** type.
  - If **schema-syntax** is **fields-scores**, the number of non-primary keys must be an even number, and the second key of every two keys except the primary key must be of the **double** type. The **double** value is the score of the previous key. The following is an example:
```
CREATE TABLE redisSink (
  order_id string,
  order_channel string,
  order_time double,
  pay_amount STRING,
  real_pay double,
  pay_time string,
  user_id double,
  user_name string,
  area_id double,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields-scores'
);
```
- Restrictions on **data-type**:
  - If **data-type** is **string**, only one non-primary key field is allowed.
  - If **data-type** is **sorted-set** and **schema-syntax** is **fields** or **array**, **default-score** is used as the score.
  - If **data-type** is **sorted-set** and **schema-syntax** is **map**, there can be only one non-primary key in addition to the primary key and the non-primary key must be of the **map** type. The **map** values of the non-primary key must be of the **double** type, indicating the score. The keys in the map are the values in the Redis set.
  - If **data-type** is **sorted-set** and **schema-syntax** is **array-scores**, only two non-primary keys are allowed and must be of the **array** type.

    The first key indicates values in the Redis set. The second key is of the **array<double>** type, indicating index scores. The following is an example:
```
CREATE TABLE redisSink (
  order_id string,
  arrayField Array<String>,
  arrayScore array<double>,
  primary key (order_id) not enforced
) WITH (
```

```
 'connector' = 'redis',
 'host' = 'RedisIP',
 'password' = 'RedisPassword',
 'data-type' = 'sorted-set',
 "default-score" = '3',
 'deploy-mode' = 'master-replica',
 'schema-syntax' = 'array-scores'
);
```

Syntax

```
create table dwsSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name) NOT ENFORCED)
)
with (
  'connector' = 'redis',
  'host' = ''
);
```

Parameters

Table 2-24 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	None	String	Connector to be used. Set this parameter to **redis**.
host	Yes	None	String	Redis connector address.
port	No	6379	Integer	Redis connector port.
password	No	None	String	Redis authentication password.
namespace	No	None	String	Redis key namespace. For example, if the value is set to "person" and the key is "jack", the value in the Redis is person:jack.
delimiter	No	:	String	Delimiter between the Redis key and namespace.
data-type	No	hash	String	Redis data type. Available values are as follows: ● hash ● list ● set ● sorted-set ● string For details about the constraints, see **Constraints on data-type**.

Parameter	Mandatory	Default Value	Data Type	Description
schema-syntax	No	fields	String	Redis schema semantics. Available values are as follows: • **fields**: applicable to all data types. This value indicates that multiple fields can be set and the value of each field is read when data is written. • **fields-scores**: applicable to **sorted-set** data, indicating that each field is read as an independent score. • **array**: applicable to **list**, **set**, and **sorted-set** data. • **array-scores**: applicable to **sorted-set** data. • **map**: applicable to **hash** and **sorted-set** data. For details about the constraints, see **Constraints on schema-syntax**.
deploy-mode	No	standalone	String	Deployment mode of the Redis cluster. The value can be **standalone**, **master-replica**, or **cluster**. The default value is **standalone**. For details about the setting, see the instance type description of the Redis cluster.
retry-count	No	5	Integer	Number of attempts to connect to the Redis cluster.
connection-timeout-millis	No	10000	Integer	Maximum timeout for connecting to the Redis cluster.
commands-timeout-millis	No	2000	Integer	Maximum time for waiting for a completion response.
rebalancing-timeout-millis	No	15000	Integer	Sleep time when the Redis cluster fails.
default-score	No	0	Double	Default score when **data-type** is **sorted-set**.
ignore-retraction	No	false	Boolean	Whether to ignore Retract messages.

Parameter	Mandatory	Default Value	Data Type	Description
skip-null-values	No	true	Boolean	Whether null values will be skipped. If this parameter is **false**, **null** will be assigned for null values.
pwd_auth_name	No	None	String	Name of datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.
key-ttl-mode	No	no-ttl	String	Whether the Redis sink TTL function will be enabled. The value can be **no-ttl**, **expire-msec**, **expire-at-date** or **expire-at-timestamp**. ● **no-ttl**: No expiration time is set. ● **expire-msec**: validity period of the key. The parameter is a long string, in milliseconds. ● **expire-at-date**: Date and time when the key expires. The value is in UTC time format. ● **expire-at-timestamp**: Timestamp when the key expires.

Parameter	Mandatory	Default Value	Data Type	Description
key-ttl	No	None	String	Supplementary parameter of **key-ttl-mode**. Available values are as follows: ● If **key-ttl-mode** is **no-ttl**, this parameter does not need to be configured. ● If **key-ttl-mode** is **expire-msec**, set this parameter to a string that can be parsed into the Long type. For example, **5000** indicates that the key will expire in 5000 ms. ● If **key-ttl-mode** is **expire-at-date**, set this parameter to a date. For example, **2011-12-03T10:15:30** indicates that the expiration time is 2011-12-03 18:15:30 (UTC+8). ● If **key-ttl-mode** is **expire-at-timestamp**, set this parameter to a timestamp, in milliseconds. For example, **1679385600000** indicates that the expiration time is 2023-03-21 16:00:00.

Example

In this example, data is read from the Kafka data source and written to the Redis result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where Redis locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set Redis security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Redis address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
```



```
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourTopic>',
  'properties.bootstrap.servers' = '<yourKafka>:<port>',
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);
--In the following redisSink table, data-type is set to default value hash, schema-syntax is fields, and order_id is defined as the primary key. Therefore, the value of this field is used as the Redis key.
CREATE TABLE redisSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = '<yourRedis>',
  'password' = '<yourPassword>',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields'
);

insert into redisSink select * from orders;
```

4. Connect to the Kafka cluster and insert the following test data into Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}
```

5. Run the following commands in Redis and view the result:
   - Obtain the result whose key is **202103241606060001**.

     Run following command:

     ```
     HGETALL 202103241606060001
     ```

     Command output:

     ```
      1) "user_id"
      2) "0001"
      3) "user_name"
      4) "Alice"
      5) "pay_amount"
      6) "200.0"
      7) "real_pay"
      8) "180.0"
      9) "order_time"
     10) "2021-03-24 16:06:06"
     11) "area_id"
     12) "330106"
     13) "order_channel"
     14) "appShop"
     15) "pay_time"
     16) "2021-03-24 16:10:06"
     ```

– Obtain the result whose key is **202103241000000001**.

Run following command:

```
HGETALL 202103241000000001
```

Command output:

```
 1) "user_id"
 2) "0001"
 3) "user_name"
 4) "Alice"
 5) "pay_amount"
 6) "100.0"
 7) "real_pay"
 8) "100.0"
 9) "order_time"
10) "2021-03-24 10:00:00"
11) "area_id"
12) "330106"
13) "order_channel"
14) "webShop"
15) "pay_time"
16) "2021-03-24 10:02:03"
```

FAQ

- Q: When data-type is **set**, why is the final result data less than the input data?

  A: This is because the input data contains duplicate data. Deduplication is performed in the Redis set, and the number of records in the result decreases.

- Q: What should I do if Flink job logs contain the following error information?

  ```
  org.apache.flink.table.api.ValidationException: SQL validation failed. From line 1, column 40 to line 1, column 105: Parameters must be of the same type
  ```

  A: The array type is used. However, the types of fields in the array are different. You need to ensure that the types of fields in the array in Redis are the same.

- Q: What should I do if Flink job logs contain the following error information?

  ```
  org.apache.flink.addons.redis.core.exception.RedisConnectorException: Wrong Redis schema for 'map' syntax: There should be a key (possibly) and 1 MAP non-key column.
  ```

  A: When **schema-syntax** is **map**, the table creation statement in Flink can contain only one non-primary key column, and the column type must be **map**.

- Q: What should I do if Flink job logs contain the following error information?

  ```
  org.apache.flink.addons.redis.core.exception.RedisConnectorException: Wrong Redis schema for 'array' syntax: There should be a key (possibly) and 1 ARRAY non-key column.
  ```

  A: When **schema-syntax** is **array**, the table creation statement in Flink can contain only one non-primary key column, and the column type must be **array**.

- Q: What is the function of **schema-syntax** since **data-type** has been set?

  A: **schema-syntax** is used to process special types, such as **map** and **array**.

  – If it is set to **fields**, the value of each field is processed. If it is set to **array** or **map**, each element in the field is processed. For **fields**, the field value of the **map** or **array** type is directly used as a value in Redis.

  – For **array** or **map**, each value in the array is used as a Redis value, and the field value of the map is used as the Redis value. **array-scores** is used to process the **sorted-set** data type. It indicates that two array fields are used, the first one is the value in the set, and the second one is the score. **fields-scores** is used to process the **sorted-set** data type, indicating that the score is derived from the defined field. The field of an odd number except the primary key indicates the value in the set, and its next field indicates its score. Therefore, its next field must be of the **double** type.

- Q: If **data-type** is **hash**, what are the differences between **schema-syntax** set to **fields** and that to **map**?

  A: When **fields** is used, the field name in Flink is used as the Redis field of the hash data type, and the value of that field is used as the value of the hash data type in Redis. When **map** is used, the field key in Flink is used as the Redis field of the hash data type, and the value of that field is used as the value of the hash data type in Redis. The following is an example:

  – For **fields**:

    i. The execution script of the Flink job is as follows:
```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE redisSink (
  order_id string,
  maptest Map<string, String>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields'
);

insert into redisSink select order_id, Map[user_id, area_id] from orders;
```

    ii. Connect to the Kafka cluster and insert the following test data into the Kafka topic:
```
{"order_id":"202103241000000001", "order_channel":"webShop",
"order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00",
"pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice",
"area_id":"330106"}
```

    iii. In the Redis, the result is as follows:
```
1) "maptest"
2) "{0001=330106}"
```

  – For **map**:

    i. The execution script of the Flink job is as follows:
```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
```

```
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE redisSink (
  order_id string,
  maptest Map<string, String>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'map'
);

insert into redisSink select order_id, Map[user_id, area_id] from orders;
```

ii. Connect to the Kafka cluster and insert the following test data into the Kafka topic:

```
{"order_id":"202103241000000001", "order_channel":"webShop",
"order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00",
"pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice",
"area_id":"330106"}
```

iii. In the Redis, the result is as follows:

```
1) "0001"
2) "330106"
```

- Q: If **data-type** is **list**, what are the differences between **schema-syntax** set to **fields** and that to **array**?

  A: The setting to **fields** or **array** does not result in different results. The only difference is that in the Flink table creation statement. **fields** can be multiple fields. However, **array** requires that the field is of the **array** type and the data types in the array must be the same. Therefore, **fields** are more flexible.

  – For **fields**:

    i. The execution script of the Flink job is as follows:

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE redisSink (
```

```
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'list',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields'
);

insert into redisSink select * from orders;
```

ii. Connect to the Kafka cluster and insert the following test data into the Kafka topic:

```
{"order_id":"202103241000000001", "order_channel":"webShop",
"order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00",
"pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice",
"area_id":"330106"}
```

iii. View the result.

Run the following command in Redis:

```
LRANGE 202103241000000001 0 8
```

The command output is as follows:

```
1) "webShop"
2) "2021-03-24 10:00:00"
3) "100.0"
4) "100.0"
5) "2021-03-24 10:02:03"
6) "0001"
7) "Alice"
8) "330106"
```

– For **array**:

i. The execution script of the Flink job is as follows:

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE redisSink (
  order_id string,
  arraytest Array<String>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
```

```
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'list',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'array'
);

insert into redisSink select order_id,
array[order_channel,order_time,pay_time,user_id,user_name,area_id] from orders;
```

ii. Connect to the Kafka cluster and insert the following test data into the Kafka topic:

```
{"order_id":"202103241000000001", "order_channel":"webShop",
"order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00",
"pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice",
"area_id":"330106"}
```

iii. In Redis, view the result. (The result is different from that of **fields** because data of the **double** type is not added to the table creation statement of the sink in Flink. Therefore, two values are missing. This is not caused by the difference between **fields** and **array**.)

```
1) "webShop"
2) "2021-03-24 10:00:00"
3) "2021-03-24 10:02:03"
4) "0001"
5) "Alice"
6) "330106"
```

2.3.2.10 Upsert Kafka Result Table

Function

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. DLI outputs the Flink job output data to Kafka in upsert mode.

The Upsert Kafka connector allows for reading data from and writing data into Kafka topics in the upsert fashion.

As a sink, the Upsert Kafka connector can consume a changelog stream. It will write INSERT/UPDATE_AFTER data as normal Kafka messages value, and write DELETE data as Kafka messages with null values (indicate tombstone for the key). Flink will guarantee the message ordering on the primary key by partition data on the values of the primary key columns, so the UPDATE/DELETE messages on the same key will fall into the same partition.

Prerequisites

- You have created a Kafka cluster.
- An enhanced datasource connection has been created for DLI to connect to Kafka clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When creating a Flink OpenSource SQL job, you need to set **Flink Version** to **1.12** on the **Running Parameters** tab of the job editing page, select **Save Job Log**, and set the OBS bucket for saving job logs.
- For details about how to use data types, see section **Format**.
- The Upsert Kafka always works in the upsert fashion and requires to define the primary key in the DDL.
- By default, an Upsert Kafka sink ingests data with at-least-once guarantees into a Kafka topic if the query is executed with checkpointing enabled. This means that Flink may write duplicate records with the same key into the Kafka topic. Therefore, the Upsert Kafka connector achieves idempotent writes.

Syntax

```
create table kafkaSource(
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector' = 'upsert-kafka',
  'topic' = '',
  'properties.bootstrap.servers' = '',
  'key.format' = '',
  'value.format' = ''
);
```

Parameters

Table 2-25 Parameter description

Parameter	Mandatory	Default Value	Data Type	Description
connector	Yes	(none)	String	Connector to be used. Set this parameter to **upsert-kafka**.
topic	Yes	(none)	String	Kafka topic name.
properties.bootstrap.servers	Yes	(none)	String	Comma separated list of Kafka brokers.

Parameter	Mandatory	Default Value	Data Type	Description
key.format	Yes	(none)	String	Format used to deserialize and serialize the key part of Kafka messages. The key fields are specified by the **PRIMARY KEY** syntax. The following formats are supported: ● csv ● json ● avro Refer to **Format** for more details and format parameters.
key.fields-prefix	No	(none)	String	Defines a custom prefix for all fields of the key format to avoid name clashes with fields of the value format. By default, the prefix is empty. If a custom prefix is defined, both the table schema and **key.fields** will work with prefixed names. When constructing the data type of the key format, the prefix will be removed and the non-prefixed names will be used within the key format. Note that this option requires that **value.fields-include** must be set to **EXCEPT_KEY**.
value.format	Yes	(none)	String	Format used to deserialize and serialize the value part of Kafka messages. The following formats are supported: ● csv ● json ● avro Refer to **Format** for more details and format parameters.
value.fields-include	No	'ALL'	String	Controls which fields should appear in the value part. Options: ● **ALL**: All fields in the schema, including the primary key field, are included in the value part. ● **EXCEPT_KEY**: All the fields of the table schema are included, except the primary key field.

Parameter	Mandatory	Default Value	Data Type	Description
sink.parallelism	No	(none)	Interger	Defines the parallelism of the Upsert Kafka sink operator. By default, the parallelism is determined by the framework using the same parallelism of the upstream chained operator.
properties.*	No	(none)	String	This option can set and pass arbitrary Kafka configurations. The suffix of this parameter must match the parameter defined in **Kafka Configuration documentation**. Flink will remove the **properties.** key prefix and pass the transformed key and value to the underlying KafkaClient. For example, you can disable automatic topic creation via **'properties.allow.auto.create.topics' = 'false'**. But there are some configurations that do not support to set, because Flink will override them, for example, **'key.deserializer'** and **'value.deserializer'**.
ssl_auth_name	No	None	String	Name of datasource authentication of the Kafka_SSL type created on DLI. This configuration is used when SSL is configured for Kafka. Note: If only the SSL type is used, you need to set **properties.security.protocol** to **SSL**. If the SASL_SSL type is used, you need to set **properties.security.protocol** to **SASL_SSL**, **properties.sasl.mechanism** to **GSSAPI** or **PLAIN**, and **properties.sasl.jaas.config** to **org.apache.kafka.common.security.plain.PlainLoginModule required username=\"xxx\" password=\"xxx\";**.

Parameter	Mandatory	Default Value	Data Type	Description
krb_auth_name	No	None	String	Name of datasource authentication of the Kerberos type created on DLI. This configuration is used when SASL is configured for Kafka. Note: If the SASL_PLAINTEXT type and Kerberos authentication are used, you need to set **properties.sasl.mechanism** to **GSSAPI** and **properties.security.protocol** to **SASL_PLAINTEXT**.

Example

In this example, Kafka source topic data is read from the Kafka source table and written to the Kafka sink topic through the Upsert Kafka result table.

1. Create an enhanced datasource connection in the VPC and subnet where Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Kafka address by referring to **Testing Address Connectivity**. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.
3. Create a Flink OpenSource SQL job. Enter the following job script and submit the job.

   When you create a job, set **Flink Version** to **1.12** on the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  "format" = "json"
);
CREATE TABLE UPSERTKAFKASINK (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
```

```
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  PRIMARY KEY (order_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' =  'KafkaAddress1:KafkaPort,KafkAddress2:KafkaPort',
  'key.format' = 'json',
  'value.format' = 'json'
);
insert into UPSERTKAFKASINK
select * from orders;
```

4. Connect to the Kafka cluster and send the following test data to the Kafka source topic:

```
{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202103251505050001", "order_channel":"qqShop", "order_time":"2021-03-25 15:05:05", "pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2021-03-25 15:10:00", "user_id":"0003", "user_name":"Cindy", "area_id":"330108"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

5. Connect to the Kafka cluster and read data from the Kafka sink topic. The result is as follows:

```
{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202103251505050001", "order_channel":"qqShop", "order_time":"2021-03-25 15:05:05", "pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2021-03-25 15:10:00", "user_id":"0003", "user_name":"Cindy", "area_id":"330108"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

FAQ

None

2.3.2.11 FileSystem Result Table

Function

The FileSystem result (sink) table is used to export data to the HDFS or OBS file system. It is applicable to scenarios such as data dumping, big data analysis, data backup, and active, deep, or cold archiving.

Considering that the input stream can be unbounded, you can put the data in each bucket into **part** files of a limited size. Data can be written into a bucket based on time. For example, you can write data into a bucket every hour. This bucket contains the records received within one hour, and

data in the bucket directory is split into multiple **part** files. Each sink bucket that receives data contains at least one **part** file for each subtask. Other **part** files are created based on the configured rolling policy. For Row Formats, the default rolling policy is based on the **part** file size. You need to specify the maximum timeout period for opening a file and the timeout period for the inactive state after closing a file. Bulk Formats are rolled each time a checkpoint is created. You can add other rolling conditions based on size or time.

NOTE

- To use FileSink in STREAMING mode, you need to enable the checkpoint function. **Part** files are generated only when the checkpoint is successful. If the checkpoint function is not enabled, the files remain in the in-progress or pending state, and downstream systems cannot securely read the file data.
- The number recorded by the sink end operator is the number of checkpoints, not the actual volume of the sent data. For the actual volume, see the number recorded by the streaming-writer or StreamingFileWriter operator.

Syntax

```
CREATE TABLE sink_table (
   name string,
   num INT,
   p_day string,
   p_hour string
) partitioned by (p_day, p_hour) WITH (
   'connector' = 'filesystem',
   'path' = 'obs://*** ',
   'format' = 'parquet',
   'auto-compaction' = 'true'
);
```

Usage

- **Rolling Policy**

  The Rolling Policy determines when to close the current in-progress part file and transition it from the in-progress state to the pending state, and then to the finished state. Part files in the "finished" state are the ones that are ready for viewing and are guaranteed to contain valid data that will not be reverted in case of failure.

  In STREAMING mode, the Rolling Policy in combination with the checkpointing interval (pending files become finished on the next checkpoint) control how quickly part files become available for downstream readers and also the size and number of these parts. For details, see **Parameters**.

- **Part File Lifecycle**

  To use the output of the FileSink in downstream systems, we need to understand the naming and lifecycle of the output files produced.

  Part files can be in one of three states:

  – **In-progress**: The part file that is currently being written to is in-progress.

  – **Pending**: Closed (due to the specified rolling policy) in-progress files that are waiting to be committed.

  – **Finished**: On successful checkpoints (STREAMING) or at the end of input (BATCH) pending files transition to **Finished**

  Only finished files are safe to read by downstream systems as those are guaranteed to not be modified later.

  By default, the file naming strategy is as follows:

- **In-progress / Pending**: part-<uid>-<partFileIndex>.inprogress.uid
- **Finished**: part-<uid>-<partFileIndex>

uid is a random ID assigned to a subtask of the sink when the subtask is instantiated. This **uid** is not fault-tolerant so it is regenerated when the subtask recovers from a failure.

- **Compaction**

  FileSink supports compaction of the pending files, which allows the application to have smaller checkpoint interval without generating a lot of small files.

  Once enabled, the compaction happens between the files become pending and get committed. The pending files will be first committed to temporary files whose path starts with a dot (.). Then these files will be compacted according to the strategy by the compactor specified by the users, and the new compacted pending files will be generated. Then these pending files will be emitted to the committer to be committed to the formal files. After that, the source files will be removed.

- **Partitions**

  Filesystem sink supports the partitioning function. Partitions are generated based on the selected fields by using the **partitioned by** syntax. The following is an example:

```
path
└── datetime=2022-06-25
    └── hour=10
        ├── part-0.parquet
        ├── part-1.parquet
└── datetime=2022-06-26
    └── hour=16
        ├── part-0.parquet
    └── hour=17
        ├── part-0.parquet
```

  Similar to files, partitions also need to be submitted to notify downstream applications that files in the partitions can be securely read. Filesystem sink provides multiple configuration submission policies.

Parameters

Table 2-26 Parameter description

Parameter	Mandatory	Default Value	Type	Description
connector	Yes	None	String	The value is fixed at **filesystem**.
path	Yes	None	String	OBS path
format	Yes	None	String	File format Available values are: **csv** and **parquet**

Parameter	Mandatory	Default Value	Type	Description
sink.rolling-policy.file-size	No	128 MB	MemorySize	Maximum size of a part file. If the size of a part file exceeds this value, a new file will be generated. **NOTE** The Rolling Policy determines when to close the current in-progress part file and transition it from the in-progress state to the pending state, and then to the finished state. Part files in the "finished" state are the ones that are ready for viewing and are guaranteed to contain valid data that will not be reverted in case of failure. In STREAMING mode, the Rolling Policy in combination with the checkpointing interval (pending files become finished on the next checkpoint) control how quickly part files become available for downstream readers and also the size and number of these parts.
sink.rolling-policy.rollover-interval	No	30 min	Duration	Maximum duration that a part file can be opened. If a part file is opened longer than the maximum duration, a new file will be generated in rolling mode. The default value is 30 minutes so that there will not be a large number of small files. The check frequency is specified by **sink.rolling-policy.check-interval**. **NOTE** There must be a space between the number and the unit. The supported time units include **d**, **h**, **min**, **s**, and **ms**. For bulk files (parquet, orc, and avro), the checkpoint interval also controls the maximum open duration of a part file.
sink.rolling-policy.check-interval	No	1 min	Duration	Check interval of the time-based rolling policy This parameter controls the frequency of checking whether a file should be rolled based on **sink.rolling-policy.rollover-interval**.

Parameter	Mandatory	Default Value	Type	Description
auto-compaction	No	false	Boolean	Whether automatic compaction is enabled for the streaming sink. Data is first written to temporary files. After the checkpoint is complete, the temporary files generated by the checkpoint are compacted.
compaction.file-size	No	Size of **sink.rolling-policy.file-size**	MemorySize	Size of the files that will be compacted. The default value is the size of the files that will be rolled. **NOTE** • Only files in the same checkpoint are compacted. The final files must be more than or equal to the number of checkpoints. • If the compaction takes a long time, back pressure may occur and the checkpointing may be prolonged. • After this function is enabled, final files are generated during checkpoint and a new file is opened to receive the data generated at the next checkpoint.

Example 1

Use datagen to randomly generate data and write the data into the **fileName** directory in the OBS bucket **bucketName**. The file generation time is irrelevant to the checkpoint. When the file is opened more than 30 minutes or is bigger than 128 MB, a new file is generated.

```
create table orders(
  name string,
  num INT
) with (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'fields.name.kind' = 'random',
  'fields.name.length' = '5'
);

CREATE TABLE sink_table (
   name string,
   num INT
) WITH (
   'connector' = 'filesystem',
   'path' = 'obs://bucketName/fileName',
   'format' = 'csv',
   'sink.rolling-policy.file-size'='128m',
   'sink.rolling-policy.rollover-interval'='30 min'
);
INSERT into sink_table SELECT * from orders;
```

Example 2

Use datagen to randomly generate data and write the data into the **fileName** directory in the OBS bucket **bucketName**. The file generation time is relevant to the checkpoint. When the checkpoint interval is reached or the file size reaches 100 MB, a new file is generated.

```
create table orders(
  name string,
  num INT
) with (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'fields.name.kind' = 'random',
  'fields.name.length' = '5'
);

CREATE TABLE sink_table (
   name string,
   num INT
) WITH (
   'connector' = 'filesystem',
   'path' = 'obs://bucketName/fileName',
   'format' = 'csv',
   'sink.rolling-policy.file-size'='128m',
   'sink.rolling-policy.rollover-interval'='30 min',
   'auto-compaction'='true',
   'compaction.file-size'='100m'

);
INSERT into sink_table SELECT * from orders;
```

2.3.3 Creating Dimension Tables

2.3.3.1 GaussDB(DWS) Dimension Table

Function

Create a GaussDB(DWS) table to connect to source streams for wide table generation.

Prerequisites

- Ensure that you have created a GaussDB(DWS) cluster using your account. For details about how to create a DWS cluster, see **Creating a Cluster** in the *Data Warehouse Service Management Guide*.
- A DWS database table has been created.
- An enhanced datasource connection has been created for DLI to connect to DWS clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

When you create a Flink OpenSource SQL job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
)
with (
  'connector' = 'gaussdb',
  'url' = '',
  'table-name' = '',
  'username' = '',
  'password' = ''
);
```

Parameters

Table 2-27 Parameter description

Parameter	Mandatory	Default Value	Data Types	Description
connector	Yes	None	String	Connector type. Set this parameter to **gaussdb**.
url	Yes	None	String	JDBC connection address. If you use the gsjdbc4 driver, set the value in jdbc:postgresql://${ip}:${port}/${dbName} format. If you use the gsjdbc200 driver, set the value in jdbc:gaussdb://${ip}:${port}/${dbName} format.
table-name	Yes	None	String	Name of the table where the data will be read from the database
driver	No	None	String	JDBC connection driver. The default value is **org.postgresql.Driver**. ● If you use the gsjdbc4 driver for connection, set **connector.driver** to **org.postgresql.Driver**. ● If you use the gsjdbc200 driver for connection, set **connector.driver** to **com.huawei.gauss200.jdbc.Driver**.

Parameter	Mandatory	Default Value	Data Types	Description
username	No	None	String	Database authentication user name. This parameter must be configured in pair with **password**.
password	No	None	String	Database authentication password. This parameter must be configured in pair with **username**.
scan.partition.column	No	None	String	Name of the column used to partition the input This parameter must be set when **scan.partition.lower-bound**, **scan.partition.upper-bound**, and **scan.partition.num** are all configured, and should not be set when other three parameters are not.
scan.partition.lower-bound	No	None	Integer	Lower bound of values to be fetched for the first partition This parameter must be set when **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.num** are all configured, and should not be set when other three parameters are not.
scan.partition.upper-bound	No	None	Integer	Upper bound of values to be fetched for the last partition This parameter must be set when **scan.partition.column**, **scan.partition.lower-bound**, and **scan.partition.num** are all configured, and should not be set when other three parameters are not.
scan.partition.num	No	None	Integer	Number of partitions to be created This parameter must be set when **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.upper-bound** are all configured, and should not be set when other three parameters are not.
scan.fetch-size	No	0	Integer	Number of rows fetched from the database each time. The default value **0** indicates that the number of rows is not limited.

Parameter	Mandatory	Default Value	Data Types	Description
scan.auto-commit	No	true	Boolean	Automatic commit flag. It determines whether each statement is committed in a transaction automatically.
lookup.cache.max-rows	No	None	Integer	Maximum number of cached rows in a dimension table. When the rows exceed this value, the data that is added first will be marked as expired. Lookup cache is disabled by default.
lookup.cache.ttl	No	None	Duration	Maximum time to live (TTL) of for every rows in lookup cache. Caches exceeding the TTL will be expired. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit). Lookup cache is disabled by default.
lookup.max-retries	No	3	Integer	Maximum retry times if lookup database failed.
pwd_auth_name	No	None	String	Name of datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.

Example

Read data from a Kafka source table, use a GaussDB(DWS) table as the dimension table. Write wide table information generated by the source and dimension tables to a Kafka result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where DWS and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set GaussDB(DWS) and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the DWS and Kafka address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
3. Connect to the GaussDB(DWS) database instance, create a table as a dimension table, and name the table **area_info**. Example SQL statements are as follows:
```
create table public.area_info(
  area_id VARCHAR,
```

```
area_province_name VARCHAR,
area_city_name VARCHAR,
area_county_name VARCHAR,
area_street_name VARCHAR,
region_name VARCHAR);
```

4. Connect to the database and run the following statement to insert test data into the dimension table **area_info**:

```
insert into area_info
(area_id, area_province_name, area_city_name, area_county_name, area_street_name, region_name)
values
('330102', 'a1', 'b1', 'c1', 'd1', 'e1'),
('330106', 'a1', 'b1', 'c2', 'd2', 'e1'),
('330108', 'a1', 'b1', 'c3', 'd3', 'e1'),
('330110', 'a1', 'b1', 'c4', 'd4', 'e1');
```

5. Create a Flink OpenSource SQL job Enter the following job script and submit the job. The job script uses Kafka as the data source and a GaussDB(DWS) table as the dimension table. Data is output to a Kafka result table.

   When you create a job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Set the values of the parameters in bold in the following script as needed.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaSourceTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'dws-order',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

-- Create an address dimension table
create table area_info (
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) WITH (
  'connector' = 'gaussdb',
  'driver' = 'org.postgresql.Driver',
  'url' = 'jdbc:gaussdb://DwsAddress:DwsPort/DwsDbName',
  'table-name' = 'area_info',
  'username' = 'DwsUserName',
  'password' = 'DwsPassword',
  'lookup.cache.max-rows' = '10000',
  'lookup.cache.ttl' = '2h'
);

-- Generate a wide table based on the address dimension table containing detailed order information.
create table order_detail(
    order_id string,
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
```

```
    pay_time string,
    user_id string,
    user_name string,
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) with (
  'connector' = 'kafka',
  'topic' = 'KafkaSinkTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'json'
);

insert into order_detail
    select orders.order_id, orders.order_channel, orders.order_time, orders.pay_amount, orders.real_pay,
orders.pay_time, orders.user_id, orders.user_name,
           area.area_id, area.area_province_name, area.area_city_name, area.area_county_name,
           area.area_street_name, area.region_name  from orders
    left join area_info for system_time as of orders.proctime as area on orders.area_id = area.area_id;
```

6. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

```
{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202103251505050001", "order_channel":"qqShop", "order_time":"2021-03-25 15:05:05",
"pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2021-03-25 15:10:00", "user_id":"0003",
"user_name":"Cindy", "area_id":"330108"}
```

7. Connect to the Kafka cluster and read data from the sink topic of Kafka. The result is as follows:

```
{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106","area_province_name":"a1","area_ci
ty_name":"b1","area_county_name":"c2","area_street_name":"d2","region_name":"e1"}

{"order_id":"202103251202020001","order_channel":"miniAppShop","order_time":"2021-03-25
12:02:02","pay_amount":60.0,"real_pay":60.0,"pay_time":"2021-03-25
12:03:00","user_id":"0002","user_name":"Bob","area_id":"330110","area_province_name":"a1","area_cit
y_name":"b1","area_county_name":"c4","area_street_name":"d4","region_name":"e1"}

{"order_id":"202103251505050001","order_channel":"qqShop","order_time":"2021-03-25
15:05:05","pay_amount":500.0,"real_pay":400.0,"pay_time":"2021-03-25
15:10:00","user_id":"0003","user_name":"Cindy","area_id":"330108","area_province_name":"a1","area_c
ity_name":"b1","area_county_name":"c3","area_street_name":"d3","region_name":"e1"}
```

FAQs

- Q: What should I do if Flink job logs contain the following error information?

```
java.io.IOException: unable to open JDBC writer
...
Caused by: org.postgresql.util.PSQLException: The connection attempt failed.
...
Caused by: java.net.SocketTimeoutException: connect timed out
```

  A: The datasource connection is not bound or the binding fails.

  – To reconfigure datasource connections, refer to **Enhanced Datasource Connection**. Rectify the fault by referring to **DLI Failed to Connect to GaussDB(DWS) Through an Enhanced Datasource Connection**.

- Q: How can I configure a GaussDB(DWS) table that is in a schema?

  A: In the following example configures the **area_info** table in the **dbuser2** schema.

```
-- Create an address dimension table
create table area_info (
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) WITH (
 'connector' = 'gaussdb',
  'driver' = 'org.postgresql.Driver',
  'url' = 'jdbc:postgresql://DwsAddress:DwsPort/DwsDbname',
  'table-name' = 'dbuser2.area_info',
  'username' = 'DwsUserName',
  'password' = 'DwsPassword',
  'lookup.cache.max-rows' = '10000',
  'lookup.cache.ttl' = '2h'
);
```

2.3.3.2 HBase Dimension Table

Function

Create a Hbase dimension table to connect to the source streams for wide table generation.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- If MRS HBase is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.

  For details, see **Modifying the Host Information** in the Data Lake Insight User Guide.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- All the column families in HBase table must be declared as ROW type, the field name maps to the column family name, and the nested field names map

to the column qualifier names. There is no need to declare all the families and qualifiers in the schema, users can declare what is used in the query. Except the ROW type fields, the single atomic type field (for example, STRING, BIGINT) will be recognized as HBase rowkey. The rowkey field can be an arbitrary name, but should be quoted using backticks if it is a reserved keyword.

Syntax

```
create table hbaseSource (
  attr_name attr_type
  (',' attr_name attr_type)*
 )
with (
  'connector' = 'hbase-2.2',
  'table-name' = '',
  'zookeeper.quorum' = ''
);
```

Parameters

Table 2-28 Parameter description

Parameter	Mandatory	Default Value	Type	Description
connector	Yes	None	String	Connector type. Set this parameter to **hbase-2.2**.
table-name	Yes	None	String	Name of the HBase table

Parameter	Mandatory	Default Value	Type	Description
zookeeper.quorum	Yes	None	String	HBase Zookeeper quorum. The format is ZookeeperAddress:ZookeeperPort. The following describes how to obtain the ZooKeeper IP address and port number: ● On the MRS Manager console, choose **Cluster** > *Name of the desired cluster* > **Service** > **ZooKeeper** > **Instance**. On the displayed page, obtain the IP address of the ZooKeeper instance. ● On the MRS Manager console, choose **Cluster** > *Name of the desired cluster* > **Service** > **ZooKeeper** > **Configuration**, and click **All Configurations**. Search for the **clientPort** parameter, and obtain the ZooKeeper port number.
zookeeper.znode.parent	No	/hbase	String	Root directory in ZooKeeper for the HBase cluster.
lookup.async	No	false	Boolean	Whether async lookup is enabled.
lookup.cache.max-rows	No	-1	Long	Maximum number of cached rows in a dimension table. When the rows exceed this value, the data that is added first will be marked as expired. Lookup cache is disabled by default.
lookup.cache.ttl	No	-1	Long	Maximum time to live (TTL) of for every rows in lookup cache. Caches exceeding the TTL will be expired. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit). Lookup cache is disabled by default.
lookup.max-retries	No	3	Integer	Maximum retry times if lookup database failed.

Parameter	Mandatory	Default Value	Type	Description
krb_auth_name	No	None	String	Name of datasource authentication of the Kerberos type created on DLI.

Data Type Mapping

HBase stores all data as byte arrays. The data needs to be serialized and deserialized during read and write operation.

When serializing and de-serializing, Flink HBase connector uses utility class **org.apache.hadoop.hbase.util.Bytes** provided by HBase (Hadoop) to convert Flink data types to and from byte arrays.

Flink HBase connector encodes null values to empty bytes, and decode empty bytes to null values for all data types except string type. For string type, the null literal is determined by null-string-literal option.

Table 2-29 Data type mapping

Flink SQL Type	HBase Conversion
CHAR / VARCHAR / STRING	byte[] toBytes(String s) String toString(byte[] b)
BOOLEAN	byte[] toBytes(boolean b) boolean toBoolean(byte[] b)
BINARY / VARBINARY	Return byte[] as is.
DECIMAL	byte[] toBytes(BigDecimal v) BigDecimal toBigDecimal(byte[] b)
TINYINT	new byte[] { val } bytes[0] // returns first and only byte from bytes
SMALLINT	byte[] toBytes(short val) short toShort(byte[] bytes)
INT	byte[] toBytes(int val) int toInt(byte[] bytes)
BIGINT	byte[] toBytes(long val) long toLong(byte[] bytes)
FLOAT	byte[] toBytes(float val) float toFloat(byte[] bytes)

Flink SQL Type	HBase Conversion
DOUBLE	byte[] toBytes(double val) double toDouble(byte[] bytes)
DATE	Number of days since 1970-01-01 00:00:00 UTC. The value is an integer.
TIME	Number of milliseconds since 1970-01-01 00:00:00 UTC. The value is an integer.
TIMESTAMP	Number of milliseconds since 1970-01-01 00:00:00 UTC. The value is of the long type.
ARRAY	Not supported
MAP / MULTISET	Not supported
ROW	Not supported

Example

In this example, data is read from a Kafka data source, an HBase table is used as a dimension table to generate a wide table, and the result is written to a Kafka result table. The procedure is as follows (the HBase versions in this example are 1.3.1 and 2.2.3):

1. Create an enhanced datasource connection in the VPC and subnet where HBase and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**. Add MRS host information for the enhanced datasource connection. For details, see **Modifying Host Information**.
2. Set HBase and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the HBase and Kafka address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
3. Create a HBase table and name it **area_info** using the HBase shell. The table has only one column family **detail**. For details, see **Using HBase from Scratch**. The creation statement is as follows:
```
create 'area_info', {NAME => 'detail'}
```
4. Run the following statement in the HBase shell to insert dimension table data:
```
put 'area_info', '330106', 'detail:area_province_name', 'a1'
put 'area_info', '330106', 'detail:area_city_name', 'b1'
put 'area_info', '330106', 'detail:area_county_name', 'c2'
put 'area_info', '330106', 'detail:area_street_name', 'd2'
put 'area_info', '330106', 'detail:region_name', 'e1'

put 'area_info', '330110', 'detail:area_province_name', 'a1'
put 'area_info', '330110', 'detail:area_city_name', 'b1'
put 'area_info', '330110', 'detail:area_county_name', 'c4'
put 'area_info', '330110', 'detail:area_street_name', 'd4'
put 'area_info', '330110', 'detail:region_name', 'e1'
```

5. Create a Flink OpenSource SQL job Enter the following job script and submit the job. The job script uses Kafka as the data source and an HBase table as the dimension table. Data is output to a Kafka result table.

   When you create a job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Set the values of the parameters in bold in the following script as needed.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaSourceTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

-- Create an address dimension table
create table area_info (
  area_id string,
  detail row(
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string)
) WITH (
  'connector' = 'hbase-2.2',
  'table-name' = 'area_info',
  'zookeeper.quorum' = 'ZookeeperAddress:ZookeeperPort',
  'lookup.async' = 'true',
  'lookup.cache.max-rows' = '10000',
  'lookup.cache.ttl' = '2h'
);

-- Generate a wide table based on the address dimension table containing detailed order information.
create table order_detail(
    order_id string,
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
    pay_time string,
    user_id string,
    user_name string,
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) with (
  'connector' = 'kafka',
  'topic' = '<yourSinkTopic>',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'json'
);
```

```
insert into order_detail
    select orders.order_id, orders.order_channel, orders.order_time, orders.pay_amount, orders.real_pay,
orders.pay_time, orders.user_id, orders.user_name,
           area.area_id, area.area_province_name, area.area_city_name, area.area_county_name,
           area.area_street_name, area.region_name  from orders
    left join area_info for system_time as of orders.proctime as area on orders.area_id = area.area_id;
```

6. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}
```

7. Connect to the Kafka cluster and read data from the sink topic of Kafka. The result data is as follows:

```
{"order_id":"202103241000000001","order_channel":"webShop","order_time":"2021-03-24
10:00:00","pay_amount":100.0,"real_pay":100.0,"pay_time":"2021-03-24
10:02:03","user_id":"0001","user_name":"Alice","area_id":"330106","area_province_name":"a1","area_ci
ty_name":"b1","area_county_name":"c2","area_street_name":"d2","region_name":"e1"}

{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106","area_province_name":"a1","area_ci
ty_name":"b1","area_county_name":"c2","area_street_name":"d2","region_name":"e1"}

{"order_id":"202103251202020001","order_channel":"miniAppShop","order_time":"2021-03-25
12:02:02","pay_amount":60.0,"real_pay":60.0,"pay_time":"2021-03-25
12:03:00","user_id":"0002","user_name":"Bob","area_id":"330110","area_province_name":"a1","area_cit
y_name":"b1","area_county_name":"c4","area_street_name":"d4","region_name":"e1"}
```

FAQs

Q: What should I do if Flink job logs contain the following error information?

```
org.apache.zookeeper.ClientCnxn$SessionTimeoutException: Client session timed out, have not heard from
server in 90069ms for connection id 0x0
```

A: The datasource connection is not bound or the binding fails. Configure the datasource connection by referring to **Enhanced Datasource Connection** or configure the security group of the Kafka cluster to allow access from the DLI queue.

2.3.3.3 JDBC Dimension Table

Create a JDBC dimension table to connect to the source stream.

Prerequisites

You have created a JDBC instance for your account.

Precautions

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.

- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Syntax

```
CREATE TABLE  table_id (
  attr_name attr_type
  (',' attr_name attr_type)*
)
  WITH (
  'connector' = 'jdbc',
  'url' = '',
  'table-name' = '',
  'driver' = '',
  'username' = '',
  'password' = ''
);
```

Parameters

Table 2-30 Parameter descriptions

Parameter	Mandatory	Description
connector	Yes	Data source type. The value is fixed to **jdbc**.
url	Yes	Database URL
table-name	Yes	Name of the table where the data will be read from the database
driver	No	Driver required for connecting to the database. If you do not set this parameter, the automatically extracted URL will be used.
username	No	Database authentication user name. This parameter must be configured in pair with **password**.
password	No	Database authentication password. This parameter must be configured in pair with **username**.
scan.partition.column	No	Name of the column used to partition the input This parameter must be set when **scan.partition.lower-bound**, **scan.partition.upper-bound**, and **scan.partition.num** are all configured, and should not be set when other three parameters are not.

Parameter	Mandatory	Description
scan.partition.lower-bound	No	Lower bound of values to be fetched for the first partition This parameter must be set when **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.num** are all configured, and should not be set when other three parameters are not.
scan.partition.upper-bound	No	Upper bound of values to be fetched for the last partition This parameter must be set when **scan.partition.column**, **scan.partition.lower-bound**, and **scan.partition.num** are all configured, and should not be set when other three parameters are not.
scan.partition.num	No	Number of partitions to be created This parameter must be set when **scan.partition.column**, **scan.partition.upper-bound**, and **scan.partition.upper-bound** are all configured, and should not be set when other three parameters are not.
scan.fetch-size	No	Number of rows fetched from the database each time. The default value is **0**, indicating the hint is ignored.
lookup.cache.max-rows	No	Maximum number of cached rows in a dimension table. When the rows exceed this value, the data that is added first will be marked as expired. The value **-1** indicates that data cache disabled.
lookup.cache.ttl	No	Maximum time to live (TTL) of for every rows in lookup cache. Caches exceeding the TTL will be expired. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit).
lookup.max-retries	No	Maximum number of attempts to obtain data from the dimension table. The default value is **3**.
pwd_auth_name	No	Name of datasource authentication of the password type created on DLI.

Data Type Mapping

Table 2-31 Data type mapping

MySQL Type	PostgreSQL Type	Flink SQL Type
TINYINT	-	TINYINT
SMALLINT TINYINT UNSIGNED	SMALLINT INT2 SMALLSERIAL SERIAL2	SMALLINT
INT MEDIUMINT SMALLINT UNSIGNED	INTEGER SERIAL	INT
BIGINT INT UNSIGNED	BIGINT BIGSERIAL	BIGINT
BIGINT UNSIGNED	-	DECIMAL(20, 0)
BIGINT	BIGINT	BIGINT
FLOAT	REAL FLOAT4	FLOAT
DOUBLE DOUBLE PRECISION	FLOAT8 DOUBLE PRECISION	DOUBLE
NUMERIC(p, s) DECIMAL(p, s)	NUMERIC(p, s) DECIMAL(p, s)	DECIMAL(p, s)
BOOLEAN TINYINT(1)	BOOLEAN	BOOLEAN
DATE	DATE	DATE
TIME [(p)]	TIME [(p)] [WITHOUT TIMEZONE]	TIME [(p)] [WITHOUT TIMEZONE]
DATETIME [(p)]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]

MySQL Type	PostgreSQL Type	Flink SQL Type
CHAR(n) VARCHAR(n) TEXT	CHAR(n) CHARACTER(n) VARCHAR(n) CHARACTER VARYING(n) TEXT	STRING
BINARY VARBINARY BLOB	BYTEA	BYTES
-	ARRAY	ARRAY

Example

Read data from a Kafka source table, use a JDBC table as the dimension table. Write table information generated by the source and dimension tables to a Kafka result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where MySQL and Kafka locate, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set MySQL and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the MySQL and Kafka address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
3. Connect to the MySQL database instance, create a table in the flink database as a dimension table, and name the table **area_info**. Example SQL statements are as follows:
```
CREATE TABLE `flink`.`area_info` (
  `area_id` VARCHAR(32) NOT NULL,
  `area_province_name` VARCHAR(32) NOT NULL,
  `area_city_name` VARCHAR(32) NOT NULL,
  `area_county_name` VARCHAR(32) NOT NULL,
  `area_street_name` VARCHAR(32) NOT NULL,
  `region_name` VARCHAR(32) NOT NULL,
  PRIMARY KEY (`area_id`)
)   ENGINE = InnoDB
  DEFAULT CHARACTER SET = utf8mb4
  COLLATE = utf8mb4_general_ci;
```
4. Connect to the MySQL database and run the following statement to insert test data into the JDBC dimension table **area_info**:
```
insert into flink.area_info
  (area_id, area_province_name, area_city_name, area_county_name, area_street_name, region_name)
  values
  ('330102', 'a1', 'b1', 'c1', 'd1', 'e1'),
  ('330106', 'a1', 'b1', 'c2', 'd2', 'e1'),
  ('330108', 'a1', 'b1', 'c3', 'd3', 'e1'),  ('330110', 'a1', 'b1', 'c4', 'd4', 'e1');
```
5. Create a Flink OpenSource SQL job Enter the following job script and submit the job. The job script uses Kafka as the data source and a JDBC table as the dimension table. Data is output to a Kafka result table.

When you create a job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Set the values of the parameters in bold in the following script as needed.**

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaSourceTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'jdbc-order',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

-- Create an address dimension table
create table area_info (
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://JDBCAddress:JDBCPort/flink',--flink is the MySQL database where the area_info
table locates.
  'table-name' = 'area_info',
  'username' = 'JDBCUserName',
  'password' = 'JDBCPassWord'
);

-- Generate a wide table based on the address dimension table containing detailed order information.
create table order_detail(
    order_id string,
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
    pay_time string,
    user_id string,
    user_name string,
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) with (
  'connector' = 'kafka',
  'topic' = 'KafkaSinkTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'json'
);

insert into order_detail
    select orders.order_id, orders.order_channel, orders.order_time, orders.pay_amount, orders.real_pay,
orders.pay_time, orders.user_id, orders.user_name,
           area.area_id, area.area_province_name, area.area_city_name, area.area_county_name,
           area.area_street_name, area.region_name  from orders
```

```
        left join area_info for system_time as of orders.proctime as area on orders.area_id =
area.area_id;
```

6. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

```
{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25 12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00", "user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202103251505050001", "order_channel":"qqShop", "order_time":"2021-03-25 15:05:05", "pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2021-03-25 15:10:00", "user_id":"0003", "user_name":"Cindy", "area_id":"330108"}
```

7. Connect to the Kafka cluster and read data from the sink topic of Kafka.

```
{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24 16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24 16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106","area_province_name":"a1","area_city_name":"b1","area_county_name":"c2","area_street_name":"d2","region_name":"e1"}

{"order_id":"202103251202020001","order_channel":"miniAppShop","order_time":"2021-03-25 12:02:02","pay_amount":60.0,"real_pay":60.0,"pay_time":"2021-03-25 12:03:00","user_id":"0002","user_name":"Bob","area_id":"330110","area_province_name":"a1","area_city_name":"b1","area_county_name":"c4","area_street_name":"d4","region_name":"e1"}

{"order_id":"202103251505050001","order_channel":"qqShop","order_time":"2021-03-25 15:05:05","pay_amount":500.0,"real_pay":400.0,"pay_time":"2021-03-25 15:10:00","user_id":"0003","user_name":"Cindy","area_id":"330108","area_province_name":"a1","area_city_name":"b1","area_county_name":"c3","area_street_name":"d3","region_name":"e1"}
```

FAQs

None

2.3.3.4 Redis Dimension Table

Function

Create a Redis table to connect to source streams for wide table generation.

Prerequisites

- An enhanced datasource connection with Redis has been established, so that you can configure security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- In Flink cross-source development scenarios, there is a risk of password leakage if datasource authentication information is directly configured. You are advised to use the datasource authentication provided by DLI.

  For details about datasource authentication, see **Introduction to Datasource Authentication**.

Precautions

- When you create a Flink OpenSource SQL job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs.
- To obtain the key values, you can set the primary key in Flink. The primary key maps to the Redis key.
- If the primary key cannot be a composite primary key, and only can be one field.
- Constraints on **schema-syntax**:
  - If **schema-syntax** is **map** or **array**, there can be only one non-primary key and it must be of the same **map** or **array** type.
  - If **schema-syntax** is **fields-scores**, the number of non-primary keys must be an even number, and the second key of every two keys except the primary key must be of the **double** type. The **double** value is the score of the previous key. The following is an example:

```
CREATE TABLE redisSource (
  redisKey string,
  order_id string,
  score1 double,
  order_channel string,
  score2 double,
  order_time string,
  score3 double,
  pay_amount double,
  score4 double,
  real_pay double,
  score5 double,
  pay_time string,
  score6 double,
  user_id string,
  score7 double,
  user_name string,
  score8 double,
  area_id string,
  score9 double,
  primary key (redisKey) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'fields-scores'
);
```

- Restrictions on **data-type**:
  - When **data-type** is **set**, the types of non-primary keys defined in Flink must be the same.
  - If **data-type** is **sorted-set** and **schema-syntax** is **fields** or **array**, only **sorted set** values can be read from Redis, and the **score** value cannot be read.
  - If **data-type** is **string**, only one non-primary key field is allowed.
  - If **data-type** is **sorted-set** and **schema-syntax** is **map**, there can be only one non-primary key in addition to the primary key and the non-primary key must be of the **map** type. The **map** values of the non-primary key must be of the **double** type, indicating the score. The keys in the map are the values in the Redis set.

– If **data-type** is **sorted-set** and **schema-syntax** is **array-scores**, only two non-primary keys are allowed and must be of the **array** type.

The first key indicates values in the Redis set. The second key is of the **array<double>** type, indicating index scores. The following is an example:

```
CREATE TABLE redisSink (
  order_id string,
  arrayField Array<String>,
  arrayScore array<double>,
  primary key (order_id) not enforced
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'sorted-set',
  "default-score" = '3',
  'deploy-mode' = 'master-replica',
  'schema-syntax' = 'array-scores'
);
```

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
  ,PRIMARY KEY (attr_name, ...) NOT ENFORCED
)
with (
  'connector' = 'redis',
  'host' = ''
);
```

Parameters

Table 2-32 Parameter description

Parameter	Mandatory	Default Value	Data Types	Description
connector	Yes	None	String	Connector type. Set this parameter to **redis**.
host	Yes	None	String	Redis connector address
port	No	6379	Integer	Redis connector port
password	No	None	String	Redis authentication password
namespace	No	None	String	Redis key namespace
delimiter	No	:	String	Delimiter between the Redis key and namespace

Parameter	Mandatory	Default Value	Data Types	Description
data-type	No	hash	String	Redis data type. Available values are as follows: ● hash ● list ● set ● sorted-set ● string For details about the constraints, see **Constraints on data-type**.
schema-syntax	No	fields	String	Redis schema semantics. Available values are as follows: ● **fields**: applicable to all data types ● **fields-scores**: applicable to **sorted set** data ● **array**: applicable to **list**, **set**, and **sorted set** data ● **array-scores**: applicable to **sorted set** data ● **map**: applicable to **hash** and **sorted set** data For details about the constraints, see **Constraints on schema-syntax**.
deploy-mode	No	standalone	String	Deployment mode of the Redis cluster. The value can be **standalone**, **master-replica**, or **cluster**. The default value is **standalone**.

Parameter	Mandatory	Default Value	Data Types	Description
retry-count	Yes	5	Integer	Size of each connection request queue. If the number of connection requests in a queue exceeds the queue size, command calling will cause RedisException. Setting **requestQueueSize** to a small value will cause exceptions to occur earlier during overload or disconnection. A larger value indicates more time required to reach the boundary, but more requests may be queued and more heap space may be used. The default value is **2147483647**.
connection-timeout-millis	No	10000	Integer	Maximum timeout for connecting to the Redis cluster
commands-timeout-millis	No	2000	Integer	Maximum time for waiting for a completion response
rebalancing-timeout-millis	No	15000	Integer	Sleep time when the Redis cluster fails
scan-keys-count	No	1000	Integer	Number of data records read in each scan
default-score	No	0	Double	Default score when **data-type** is **sorted-set**
deserialize-error-policy	No	fail-job	Enum	How to process a data parsing failure Available values are as follows: ● **fail-job**: Fail the job ● **skip-row**: Skip the current data. ● **null-field**: Set the current data to null.
skip-null-values	No	true	Boolean	Whether null values will be skipped
lookup.async	No	false	Boolean	Whether asynchronous I/O will be used when this table is used as a dimension table

Parameter	Mandatory	Default Value	Data Types	Description
pwd_auth_name	No	None	String	Name of datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.

Example

Read data from a Kafka source table, use a Redis table as the dimension table. Write wide table information generated by the source and dimension tables to a Kafka result table. The procedure is as follows:

1. Create an enhanced datasource connection in the VPC and subnet where Redis and Kafka locates, and bind the connection to the required Flink elastic resource pool. For details, see **Enhanced Datasource Connections**.
2. Set Redis and Kafka security groups and add inbound rules to allow access from the Flink queue. Test the connectivity using the Redis address by referring to **Testing Address Connectivity**. If the connection passes the test, it is bound to the queue.
3. Run the following commands on the Redis client to send data to Redis:
```
HMSET 330102  area_province_name a1 area_province_name b1 area_county_name c1 
area_street_name d1 region_name e1

HMSET 330106  area_province_name a1 area_province_name b1 area_county_name c2 
area_street_name d2 region_name e1

HMSET 330108  area_province_name a1 area_province_name b1 area_county_name c3 
area_street_name d3 region_name e1

HMSET 330110  area_province_name a1 area_province_name b1 area_county_name c4 
area_street_name d4 region_name e1
```
4. Create a Flink OpenSource SQL job Enter the following job script and submit the job. The job script uses Kafka as the data source and a Redis table as the dimension table. Data is output to a Kafka result table.

   When you create a job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Set the values of the parameters in bold in the following script as needed.**
```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  proctime as Proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'kafkaSourceTopic',
```

```
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

-- Create an address dimension table
create table area_info (
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string,
    primary key (area_id) not enforced -- Redis key
) WITH (
  'connector' = 'redis',
  'host' = 'RedisIP',
  'password' = 'RedisPassword',
  'data-type' = 'hash',
  'deploy-mode' = 'master-replica'
);

-- Generate a wide table based on the address dimension table containing detailed order information.
create table order_detail(
    order_id string,
    order_channel string,
    order_time string,
    pay_amount double,
    real_pay double,
    pay_time string,
    user_id string,
    user_name string,
    area_id string,
    area_province_name string,
    area_city_name string,
    area_county_name string,
    area_street_name string,
    region_name string
) with (
  'connector' = 'kafka',
  'topic' = 'kafkaSinkTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'format' = 'json'
);

insert into order_detail
    select orders.order_id, orders.order_channel, orders.order_time, orders.pay_amount, orders.real_pay,
orders.pay_time, orders.user_id, orders.user_name,
           area.area_id, area.area_province_name, area.area_city_name, area.area_county_name,
           area.area_street_name, area.region_name  from orders
    left join area_info for system_time as of orders.proctime as area on orders.area_id = area.area_id;
```

5. Connect to the Kafka cluster and insert the following test data into the source topic in Kafka:

```
{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06",
"pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103251202020001", "order_channel":"miniAppShop", "order_time":"2021-03-25
12:02:02", "pay_amount":"60.00", "real_pay":"60.00", "pay_time":"2021-03-25 12:03:00",
"user_id":"0002", "user_name":"Bob", "area_id":"330110"}

{"order_id":"202103251505050001", "order_channel":"qqShop", "order_time":"2021-03-25 15:05:05",
"pay_amount":"500.00", "real_pay":"400.00", "pay_time":"2021-03-25 15:10:00", "user_id":"0003",
"user_name":"Cindy", "area_id":"330108"}
```

6. Connect to the Kafka cluster and read data from the sink topic of Kafka. The result data is as follows:

```
{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106","area_province_name":"a1","area_ci
ty_name":"b1","area_county_name":"c2","area_street_name":"d2","region_name":"e1"}

{"order_id":"202103251202020001","order_channel":"miniAppShop","order_time":"2021-03-25
12:02:02","pay_amount":60.0,"real_pay":60.0,"pay_time":"2021-03-25
12:03:00","user_id":"0002","user_name":"Bob","area_id":"330110","area_province_name":"a1","area_cit
y_name":"b1","area_county_name":"c4","area_street_name":"d4","region_name":"e1"}

{"order_id":"202103251505050001","order_channel":"qqShop","order_time":"2021-03-25
15:05:05","pay_amount":500.0,"real_pay":400.0,"pay_time":"2021-03-25
15:10:00","user_id":"0003","user_name":"Cindy","area_id":"330108","area_province_name":"a1","area_c
ity_name":"b1","area_county_name":"c3","area_street_name":"d3","region_name":"e1"}
```

FAQs

If Chinese characters are written to the Redis in the Windows environment, an exception will occur during data writing.

2.3.4 Format

2.3.4.1 Avro

Function

Apache Avro is supported for you to read and write Avro data based on an Avro schema with Flink. The Avro schema is derived from the table schema.

Supported Connectors

- Kafka
- Upsert Kafka

Parameters

Table 2-33 Parameters

Parameter	Mandatory	Default value	Type	Description
format	Yes	None	String	Format to be used. Set the value to **avro**.

Parameter	Mandatory	Default value	Type	Description
avro.codec	No	None	String	Avro compression codec for the file system only The codec is disabled by default. Available values are **deflate**, **snappy**, **bzip2**, and **xz**.

Data Type Mapping

Currently, the Avro schema is derived from the table schema and cannot be explicitly defined. The following table lists mappings between Flink to Avro types.

In addition to the following types, Flink supports reading/writing nullable types. Flink maps nullable types to Avro **union(something, null)**, where **something** is an Avro type converted from Flink type.

Table 2-34 Data type mapping

Flink SQL Type	Avro Type	Avro Logical Type
CHAR / VARCHAR / STRING	string	-
BOOLEAN	boolean	-
BINARY / VARBINARY	bytes	-
DECIMAL	fixed	decimal
TINYINT	int	-
SMALLINT	int	-
INT	int	-
BIGINT	long	-
FLOAT	float	-
DOUBLE	double	-
DATE	int	date
TIME	int	time-millis

Flink SQL Type	Avro Type	Avro Logical Type
TIMESTAMP	long	timestamp-millis
ARRAY	array	-
MAP (keys must be of the string, char, or varchar type.)	map	-
MULTISET (elements must be of the string, char, or varchar type.)	map	-
ROW	record	-

Example

Read data from Kafka, deserialize the data to the Avro format, and outputs the data to print.

Step 1 Create a datasource connection for access to the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.12. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourTopic>',
  'properties.bootstrap.servers' =
'<yourKafkaAddress1>:<yourKafkaPort>,<yourKafkaAddress2>:<yourKafkaPort>,<yourKafkaAddress3>:<yourK
afkaPort>',
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  "format" = "avro"
);

CREATE TABLE printSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
```

```
  user_name string,
  area_id string
) WITH (
  'connector' = 'print'
);

insert into printSink select * from kafkaSource;
```

Step 3 Insert the following data to Kafka using Avro data serialization:

```
{"order_id":"202103241000000001","order_channel":"webShop","order_time":"2021-03-24
10:00:00","pay_amount":100.0,"real_pay":100.0,"pay_time":"2021-03-24
10:02:03","user_id":"0001","user_name":"Alice","area_id":"330106"}

{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106"}
```

Step 4 Perform the following operations to view the output:

- Method 1: Locate the job and click **More** > **FlinkUI**. Choose **Task Managers** > **Stdout**.
- Method 2: If you allow DLI to save job logs in OBS, view the output in the **taskmanager.out** file.

```
+I(202103241000000001,webShop,2021-03-2410:00:00,100.0,100.0,2021-03-2410:02:03,0001,Alice,330
106)
+I(202103241606060001,appShop,2021-03-2416:06:06,200.0,180.0,2021-03-2416:10:06,0001,Alice,3301
06)
```

----End

2.3.4.2 Canal

Function

Canal is a Changelog Data Capture (CDC) tool that can stream changes in real-time from MySQL into other systems. Canal provides a unified format schema for changelog and supports to serialize messages using JSON and protobuf (the default format for Canal).

Flink supports to interpret Canal JSON messages as INSERT, UPDATE, and DELETE messages into the Flink SQL system. This is useful in many cases to leverage this feature, such as:

- synchronizing incremental data from databases to other systems
- Auditing logs
- Real-time materialized view on databases
- Temporal join changing history of a database table, etc.

Flink also supports to encode the INSERT, UPDATE, and DELETE messages in Flink SQL as Canal JSON messages, and emit to storage like Kafka. However, currently Flink cannot combine UPDATE_BEFORE and UPDATE_AFTER into a single UPDATE message. Therefore, Flink encodes UPDATE_BEFORE and UPDATE_AFTER as DELETE and INSERT Canal messages.

Parameters

Table 2-35 Parameter description

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. In this example.Set this parameter to **canal-json**.
canal-json.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. The default value is **false**, indicating that an error will be thrown. Fields are set to null in case of errors.
canal-json.timestamp-format.standard	No	'SQL'	String	Input and output timestamp formats. Currently supported values are **SQL** and **ISO-8601**: • **SQL** will parse input timestamp in "yyyy-MM-dd HH:mm:ss.s{precision}" format, for example **2020-12-30 12:13:14.123** and output timestamp in the same format. • **ISO-8601** will parse input timestamp in "yyyy-MM-ddTHH:mm:ss.s{precision}" format, for example **2020-12-30T12:13:14.123** and output timestamp in the same format.
canal-json.map-null-key.mode	No	'FALL'	String	Handling mode when serializing null keys for map data. Available values are as follows: • **FAIL** will throw exception when encountering map value with null key. • **DROP** will drop null key entries for map data. • **LITERAL** replaces the empty key value in the map with a string constant. The string literal is defined by **canal-json.map-null-key.literal** option.

Parameter	Mandatory	Default Value	Type	Description
canal-json.map-null-key.literal	No	'null'	String	String literal to replace null key when **canal-json.map-null-key.mode** is **LITERAL**.
canal-json.database.include	No	None	String	An optional regular expression to only read the specific databases changelog rows by regular matching the **database** meta field in the Canal record.
canal-json.table.include	No	None	String	An optional regular expression to only read the specific tables changelog rows by regular matching the **table** meta field in the Canal record.

Supported Connectors

- Kafka

Example

Use Kafka to send data and output the data to print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.12. Copy the following statement and submit the job:

```
create table kafkaSource(
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
  ) with (
    'connector' = 'kafka',
    'topic' = '<yourTopic>',
    'properties.group.id' = '<yourGroupId>',
    'properties.bootstrap.servers' = '<yourKafkaAddress>:<yourKafkaPort>',
    'scan.startup.mode' = 'latest-offset',
    'format' = 'canal-json'
);
create table printSink(
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
```

```
  ) with (
    'connector' = 'print'
  );
insert into printSink select * from kafkaSource;
```

Step 3 Insert the following data to the corresponding topic in Kafka:

```
{
  "data": [
    {
      "id": "111",
      "name": "scooter",
      "description": "Big 2-wheel scooter",
      "weight": "5.18"
    }
  ],
  "database": "inventory",
  "es": 1589373560000,
  "id": 9,
  "isDdl": false,
  "mysqlType": {
    "id": "INTEGER",
    "name": "VARCHAR(255)",
    "description": "VARCHAR(512)",
    "weight": "FLOAT"
  },
  "old": [
    {
      "weight": "5.15"
    }
  ],
  "pkNames": [
    "id"
  ],
  "sql": "",
  "sqlType": {
    "id": 4,
    "name": 12,
    "description": 12,
    "weight": 7
  },
  "table": "products",
  "ts": 1589373560798,
  "type": "UPDATE"
}
```

Step 4 View the output through either of the following methods:

- Method 1: Locate the job and click **More** > **FlinkUI**. Choose **Task Managers** > **Stdout**.
- Method 2: If you allow DLI to save job logs in OBS, view the output in the **taskmanager.out** file.

```
-U(111,scooter,Big2-wheel scooter,5.15)
+U(111,scooter,Big2-wheel scooter,5.18)
```

----End

2.3.4.3 Confluent Avro

Function

The Avro Schema Registry (**avro-confluent**) format allows you to read records that were serialized by the **io.confluent.kafka.serializers.KafkaAvroSerializer** and to write records that can in turn be read by the **io.confluent.kafka.serializers.KafkaAvroDeserializer**.

When reading (deserializing) a record with this format the Avro writer schema is fetched from the configured Confluent Schema Registry based on the schema version ID encoded in the record while the reader schema is inferred from table schema.

When writing (serializing) a record with this format the Avro schema is inferred from the table schema and used to retrieve a schema ID to be encoded with the data The lookup is performed with in the configured Confluent Schema Registry under the **subject**. The subject is specified by **avro-confluent.schema-registry.subject**.

Supported Connectors

- kafka
- upsert kafka

Parameters

Table 2-36 Parameter description

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. Set this parameter to **avro-confluent**.
avro-confluent.schema-registry.subject	No	None	String	The Confluent Schema Registry subject under which to register the schema used by this format during serialization. By default, **kafka** and **upsert-kafka** connectors use **<topic_name>-value** or **<topic_name>-key** as the default subject name if this format is used as the value or key format.
avro-confluent.schema-registry.url	Yes	None	String	URL of the Confluent Schema Registry to fetch/register schemas.

Example

1. Read JSON data from the source topic in Kafka and write the data in Confluent Avro format to the sink topic.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka and ECS locate and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka and ECS IP addresses. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Purchase an ECS cluster, download Confluent 5.5.2 (**https://packages.confluent.io/archive/5.5/**) and jdk1.8.0_232, and upload them to the ECS cluster. Run the following command to decompress the packages (assume that the decompression directories are **confluent-5.5.2** and **jdk1.8.0_232**):

```
tar zxvf confluent-5.5.2-2.11.tar.gz
tar zxvf jdk1.8.0_232.tar.gz
```

Step 3 Run the following commands to install jdk1.8.0_232 in the current ECS cluster. You can run the **pwd** command in the **jdk1.8.0_232 folder** to view the value of **yourJdkPath**.

```
export JAVA_HOME=<yourJdkPath>
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
```

Step 4 Go to the **confluent-5.5.2/etc/schema-registry/** directory and modify the following configuration items in the **schema-registry.properties** file:

```
listeners=http://<yourEcsIp>:8081
kafkastore.bootstrap.servers=<yourKafkaAddress1>:<yourKafkaPort>,<yourKafkaAddress2>:<yourKafkaPort>
```

Step 5 Switch to the **confluent-5.5.2** directory and run the following command to start Confluent:

```
bin/schema-registry-start etc/schema-registry/schema-registry.properties
```

Step 6 Create a Flink opensource SQL job, select the Flink 1.12 version, and allow DLI to save job logs in OBS. Add the following statement to the job and submit it:

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'properties.bootstrap.servers' =
'<yourKafkaAddress1>:<yourKafkaPort>,<yourKafkaAddress2>:<yourKafkaPort>',
  'topic' = '<yourSourceTopic>',
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);
CREATE TABLE kafkaSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'properties.bootstrap.servers' =
'<yourKafkaAddress1>:<yourKafkaPort>,<yourKafkaAddress2>:<yourKafkaPort>',
  'topic' = '<yourSinkTopic>',
```

```
  'format' = 'avro-confluent',
  'avro-confluent.schema-registry.url' = 'http://<yourEcsIp>:8081',
  'avro-confluent.schema-registry.subject' = '<yourSubject>'
);
insert into kafkaSink select * from kafkaSource;
```

Step 7 Insert the following data into Kafka:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}
```

Step 8 Read the data of the sink Kafka topic. You will find that the data has been written and the schema has been saved to the **_schema** topic of Kafka.

----End

2.3.4.4 CSV

Function

The CSV format allows you to read and write CSV data based on a CSV schema. Currently, the CSV schema is derived from table schema.

Supported Connectors

- Kafka
- Upsert Kafka

Parameters

Table 2-37

Parameter	Mandatory	Default value	Type	Description
format	Yes	None	String	Format to be used. Set the value to **csv**.
csv.field-delimiter	No	,	String	Field delimiter character, which must be a single character. You can use backslash to specify special characters, for example, **\t** represents the tab character. You can also use unicode to specify them in plain SQL, for example, **'csv.field-delimiter' = '\u0001'** represents the 0x01 character.

Parameter	Mandatory	Default value	Type	Description
csv.disable-quote-character	No	false	Boolean	Disabled quote character for enclosing field values. If you set this parameter to **true**, **csv.quote-character** cannot be set.
csv.quote-character	No	''	String	Quote character for enclosing field values.
csv.allow-comments	No	false	Boolean	Ignore comment lines that start with **#**. If you set this parameter to **true**, make sure to also ignore parse errors to allow empty rows.
csv.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. The default value is **false**, indicating that an error will be thrown. Fields are set to null in case of errors.
csv.array-element-delimiter	No	;	String	Array element delimiter string for separating array and row element values.
csv.escape-character	No	None	String	Escape character for escaping values
csv.null-literal	No	None	String	Null literal string that is interpreted as a null value.

Example

Use Kafka to send data and output the data to print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job. Copy the following statement and submit the job:

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
```

```
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourSourceTopic>',
  'properties.bootstrap.servers' = '<yourKafkaAddress>:<yourKafkaPort>',
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  "format" = "csv"
);

CREATE TABLE kafkaSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourSinkTopic>',
  'properties.bootstrap.servers' = '<yourKafkaAddress>:<yourKafkaPort>',
  "format" = "csv"
);

insert into kafkaSink select * from kafkaSource;
```

Step 3 Insert the following data into the source Kafka topic:

```
202103251505050001,qqShop,2021-03-25 15:05:05,500.00,400.00,2021-03-25 15:10:00,0003,Cindy,330108

202103241606060001,appShop,2021-03-24 16:06:06,200.00,180.00,2021-03-24 16:10:06,0001,Alice,330106
```

Step 4 Read data from the sink Kafka topic. The result is as follows:

```
202103251505050001,qqShop,"2021-03-25 15:05:05",500.0,400.0,"2021-03-25 15:10:00",0003,Cindy,330108

202103241606060001,appShop,"2021-03-24 16:06:06",200.0,180.0,"2021-03-24 16:10:06",0001,Alice,330106
```

----End

2.3.4.5 Debezium

Function

Debezium is a Changelog Data Capture (CDC) tool that can stream changes in real-time from other databases into Kafka. Debezium provides a unified format schema for changelog and supports to serialize messages using JSON.

Flink supports to interpret Debezium JSON and Avro messages as INSERT/UPDATE/DELETE messages into Flink SQL system. This is useful in many cases to leverage this feature, such as:

- synchronizing incremental data from databases to other systems
- Auditing logs
- Real-time materialized view on databases
- Temporal join changing history of a database table, etc.

Parameters

Table 2-38

Parameter	Mandatory	Default Value	Mandatory	Description
format	Yes	None	String	Format to be used. In this example.Set this parameter to **debezium-json**.
debezium-json.schema-include	No	false	Boolean	Whether the Debezium JSON messages contain the schema. When setting up Debezium Kafka Connect, enable the Kafka configuration **value.converter.schemas.enable** to include the schema in the message.
debezium-json.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. The default value is **false**, indicating that an error will be thrown. Fields are set to null in case of errors.
debezium-json.timestamp-format.standard	No	'SQL'	String	Input and output timestamp formats. Currently supported values are **SQL** and **ISO-8601**. • **SQL** will parse input timestamp in "yyyy-MM-dd HH:mm:ss.s{precision}" format, for example **2020-12-30 12:13:14.123** and output timestamp in the same format. • **ISO-8601** will parse input timestamp in "yyyy-MM-ddTHH:mm:ss.s{precision}" format, for example **2020-12-30T12:13:14.123** and output timestamp in the same format.

Parameter	Mandatory	Default Value	Mandatory	Description
debezium-json.map-null-key.mode	No	'FAIL'	String	Handling mode when serializing null keys for map data. Available values are as follows: ● **FAIL** will throw exception when encountering map value with null key. ● **DROP** will drop null key entries for map data. ● **LITERAL** replaces the empty key value in the map with a string constant. The string literal is defined by **debezium-json.map-null-key.literal** option.
debezium-json.map-null-key.literal	No	'null'	String	String literal to replace null key when **debezium-json.map-null-key.mode** is **LITERAL**.

Supported Connectors

- Kafka

Example

Use Kafka to send data and output the data to print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job. Copy the following statement and submit the job:

```
create table kafkaSource(
  id BIGINT,
  name STRING,
  description STRING,
  weight DECIMAL(10, 2)
  ) with (
    'connector' = 'kafka',
    'topic' = '<yourTopic>',
    'properties.group.id' = '<yourGroupId>',
    'properties.bootstrap.servers' = '<yourKafkaAddress>:<yourKafkaPort>',
    'scan.startup.mode' = 'latest-offset',
    'format' = 'debezium-json'
);
create table printSink(
```

```
  id BIGINT,
  name STRING,
  description STRING,
  weight DECIMAL(10, 2)
   ) with (
     'connector' = 'print'
   );
insert into printSink select * from kafkaSource;
```

Step 3 Insert the following data to the corresponding topic in Kafka:

```
{
  "before": {
    "id": 111,
    "name": "scooter",
    "description": "Big 2-wheel scooter",
    "weight": 5.18
  },
  "after": {
    "id": 111,
    "name": "scooter",
    "description": "Big 2-wheel scooter",
    "weight": 5.15
  },
  "source": {
    "version": "0.9.5.Final",
    "connector": "mysql",
    "name": "fullfillment",
    "server_id" :1,
    "ts_sec": 1629607909,
    "gtid": "mysql-bin.000001",
    "pos": 2238,"row": 0,
    "snapshot": false,
    "thread": 7,
    "db": "inventory",
    "table": "test",
    "query": null},
  "op": "u",
  "ts_ms": 1589362330904,
  "transaction": null
}
```

Step 4 View the output through either of the following methods:

- Method 1: Locate the job and click **More** > **FlinkUI**. Choose **Task Managers** > **Stdout**.
- Method 2: If you allow DLI to save job logs in OBS, view the output in the **taskmanager.out** file.

```
-U(111,scooter,Big2-wheel scooter,5.18)
+U(111,scooter,Big2-wheel scooter,5.15)
```

----End

2.3.4.6 JSON

Function

The JSON format allows you to read and write JSON data based on a JSON schema. Currently, the JSON schema is derived from table schema.

Supported Connectors

- Kafka
- Upsert Kafka

- Elasticsearch

Parameters

Table 2-39

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. Set this parameter to **json**.
json.fail-on-missing-field	No	false	Boolean	Whether to skip the field or row or throws an error when a field to be parsed is missing. The default value is **false**, indicating that no error will be thrown.
json.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. The default value is **false**, indicating that an error will be thrown. Fields are set to null in case of errors.

Parameter	Mandatory	Default Value	Type	Description
json.timestamp-format.standard	No	'SQL'	String	Input and output timestamp format for TIMESTAMP and TIMESTAMP WITH LOCAL TIME ZONE. Currently supported values are **SQL** and **ISO-8601**: ● **SQL** will parse the input TIMESTAMP values in "yyyy-MM-dd HH:mm:ss.s{precision}" format, for example, **2020-12-30 12:13:14.123**, parse TIMESTAMP WITH LOCAL TIME ZONE values in "yyyy-MM-dd HH:mm:ss.s{precision}'Z'" format, for example, **2020-12-30 12:13:14.123Z** and output timestamp in the same format. ● **ISO-8601** will parse the input TIMESTAMP values in "yyyy-MM-ddTHH:mm:ss.s{precision}" format, for example, **2020-12-30T12:13:14.123** parse TIMESTAMP WITH LOCAL TIME ZONE values in "yyyy-MM-ddTHH:mm:ss.s{precision}'Z'" format, for example, **2020-12-30T12:13:14.123Z** and output timestamp in the same format.

Parameter	Mandatory	Default Value	Type	Description
json.map-null-key.mode	No	'FALL'	String	Handling mode when serializing null keys for map data. Available values are as follows: ● **FAIL** will throw exception when encountering map value with null key. ● **DROP** will drop null key entries for map data. ● **LITERAL** replaces the empty key value in the map with a string constant. The string literal is defined by **json.map-null-key.literal** option.
json.map-null-key.literal	No	'null'	String	String literal to replace null key when **json.map-null-key.mode** is **LITERAL**.

Example

In this example, data is read from a topic and written to another using a Kafka sink.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set an inbound rule for the security group to allow access of the queue and test the connectivity using the Kafka address. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job, select Flink 1.12, and allow DLI to save job logs in OBS. Use the following statement in the job and submit it:

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourSourceTopic>',
  'properties.bootstrap.servers' = '<yourKafkaAddress>:<yourKafkaPort>',
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  "format" = "json"
);

CREATE TABLE kafkaSink (
  order_id string,
```

```
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourSinkTopic>',
  'properties.bootstrap.servers' = '<yourKafkaAddress>:<yourKafkaPort>',
  "format" = "json"
);

insert into kafkaSink select * from kafkaSource;
```

Step 3 Insert the following data into the source Kafka topic:

```
{"order_id":"202103241000000001","order_channel":"webShop","order_time":"2021-03-24
10:00:00","pay_amount":100.0,"real_pay":100.0,"pay_time":"2021-03-24
10:02:03","user_id":"0001","user_name":"Alice","area_id":"330106"}

{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106"}
```

Step 4 Read data from the sink topic. The result is as follows:

```
{"order_id":"202103241000000001","order_channel":"webShop","order_time":"2021-03-24
10:00:00","pay_amount":100.0,"real_pay":100.0,"pay_time":"2021-03-24
10:02:03","user_id":"0001","user_name":"Alice","area_id":"330106"}

{"order_id":"202103241606060001","order_channel":"appShop","order_time":"2021-03-24
16:06:06","pay_amount":200.0,"real_pay":180.0,"pay_time":"2021-03-24
16:10:06","user_id":"0001","user_name":"Alice","area_id":"330106"}
```

----End

2.3.4.7 Maxwell

Function

Flink supports to interpret Maxwell JSON messages as INSERT/UPDATE/DELETE messages into Flink SQL system. This is useful in many cases to leverage this feature,

such as:

- Synchronizing incremental data from databases to other systems
- Auditing logs
- Real-time materialized views on databases
- Temporal join changing history of a database table and so on

Flink also supports to encode the INSERT/UPDATE/DELETE messages in Flink SQL as Maxwell JSON messages, and emit to external systems like Kafka. However, currently Flink cannot combine UPDATE_BEFORE and UPDATE_AFTER into a single UPDATE message. Therefore, Flink encodes UPDATE_BEFORE and UDPATE_AFTER as DELETE and INSERT Maxwell messages.

Parameters

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. Set this parameter to **maxwell-json**.
maxwell-json.ignore-parse-errors	No	false	Boolean	Whether fields and rows with parse errors will be skipped or failed. Fields are set to null in case of errors.
maxwell-json.timestamp-format.standard	No	'SQL'	String	Input and output timestamp formats. Currently supported values are **SQL** and **ISO-8601**: **SQL** will parse input timestamp in "yyyy-MM-dd HH:mm:ss.s{precision}" format, for example, **2020-12-30 12:13:14.123** and output timestamp in the same format. **ISO-8601** will parse input timestamp in "yyyy-MM-ddTHH:mm:ss.s{precision}" format, for example **2020-12-30T12:13:14.123** and output timestamp in the same format.
maxwell-json.map-null-key.mode	No	'FAIL'	String	Handling mode when serializing null keys for map data. Currently supported values are 'FAIL', 'DROP' and 'LITERAL': **FAIL** will throw exception when encountering map with null key. **DROP** will drop null key entries for map data. **LITERAL** will replace null key with string literal. The string literal is defined by **maxwell-json.map-null-key.literal** option.
maxwell-json.map-null-key.literal	No	'null'	String	String literal to replace null key when **maxwell-json.map-null-key.mode** is **LITERAL**.

Supported Connectors

- Kafka

Example

Use Kafka to send data and output the data to print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.12. Copy the following statement and submit the job:

```
create table kafkaSource(
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
  ) with (
    'connector' = 'kafka',
    'topic' = '<yourTopic>',
    'properties.group.id' = '<yourGroupId>',
    'properties.bootstrap.servers' =
'<yourKafkaAddress1>:<yourKafkaPort>,<yourKafkaAddress2>:<yourKafkaPort>',
    'scan.startup.mode' = 'latest-offset',
    'format' = 'maxwell-json'
);
create table printSink(
  id bigint,
  name string,
  description string,
  weight DECIMAL(10, 2)
   ) with (
     'connector' = 'print'
   );
insert into printSink select * from kafkaSource;
```

Step 3 Insert the following data to the corresponding topic in Kafka:

```
{
   "database":"test",
   "table":"e",
   "type":"insert",
   "ts":1477053217,
   "xid":23396,
   "commit":true,
   "position":"master.000006:800911",
   "server_id":23042,
   "thread_id":108,
   "primary_key": [1, "2016-10-21 05:33:37.523000"],
   "primary_key_columns": ["id", "c"],
   "data":{
     "id":111,
     "name":"scooter",
     "description":"Big 2-wheel scooter",
     "weight":5.15
   },
   "old":{
     "weight":5.18
   }
}
```

Step 4 View the output through either of the following methods:

- Method 1: Locate the job and click **More** > **FlinkUI**. Choose **Task Managers** > **Stdout**.
- Method 2: If you allow DLI to save job logs in OBS, view the output in the **taskmanager.out** file.

```
+I(111,scooter,Big 2-wheel scooter,5.15)
```

----End

2.3.4.8 Raw

Function

The raw format allows you to read and write raw (byte based) values as a single column.

Note: This format encodes null values as **null** of the **byte[]** type. This may have limitation when used in **upsert-kafka**, because **upsert-kafka** treats null values as a tombstone message (DELETE on the key). Therefore, we recommend avoiding using **upsert-kafka** connector and the **raw** format as a **value.format** if the field can have a null value.

The raw format connector is built-in, no additional dependencies are required.

Parameters

Table 2-40

Parameter	Mandatory	Default Value	Type	Description
format	Yes	None	String	Format to be used. Set this parameter to **raw**.
raw.charset	No	UTF-8	String	Charset to encode the text string.
raw.endianness	No	big-endian	String	Endianness to encode the bytes of numeric value. Valid values are **big-endian** and **little-endian**. You can search for endianness for more details.

Supported Connectors

- Kafka

- UpsertKafka

Example

Use Kafka to send data and output the data to print.

Step 1 Create a datasource connection for the communication with the VPC and subnet where Kafka locates and bind the connection to the queue. Set a security group and inbound rule to allow access of the queue and test the connectivity of the queue using the Kafka IP address. For example, locate a general-purpose queue where the job runs and choose **More** > **Test Address Connectivity** in the **Operation** column. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Create a Flink OpenSource SQL job and select Flink 1.12. Copy the following statement and submit the job:

```
create table kafkaSource(
  log string
  ) with (
    'connector' = 'kafka',
    'topic' = '<yourTopic>',
    'properties.group.id' = '<yourGroupId>',
    'properties.bootstrap.servers' = '<yourKafkaAddress>:<yourKafkaPort>',
    'scan.startup.mode' = 'latest-offset',
    'format' = 'raw'
);
create table printSink(
  log string
   ) with (
     'connector' = 'print'
   );
insert into printSink select * from kafkaSource;
```

Step 3 Insert the following data to the corresponding topic in Kafka:

```
47.29.201.179 - - [28/Feb/2019:13:17:10 +0000] "GET /?p=1 HTTP/2.0" 200 5316 "https://domain.com/?
p=1" "Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.119
Safari/537.36" "2.75"
```

Step 4 View the output through either of the following methods:

- Method 1: Locate the job and click **More** > **FlinkUI**. Choose **Task Managers** > **Stdout**.
- Method 2: If you allow DLI to save job logs in OBS, view the output in the **taskmanager.out** file.

```
+I(47.29.201.179 - - [28/Feb/2019:13:17:10 +0000] "GET /?p=1 HTTP/2.0"2005316"https://domain.com/?
p=1"
"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.119 Safari/
537.36" "2.75")
```

----End

2.4 DML Snytax

2.4.1 SELECT

SELECT

Syntax

```
SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
```

Description

SELECT is used to select data from a table.

ALL indicates that all results are returned.

DISTINCT indicates that the duplicated results are removed.

Precautions

- The to-be-queried table must exist. Otherwise, an error is reported.
- WHERE is used to specify the search 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

Syntax

```
SELECT   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
```

Description

This clause 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

Search 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

Function

This clause is used to search for the query results that meet the search condition.

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. You can use GROUP BY for grouping and then use HAVING for filtering. Arithmetic operations and aggregate functions are supported in the HAVING clause.

Precautions

If the filtering condition is subject to the results of GROUP BY, the HAVING clause, rather than the WHERE clause, must be used for search.

Example

Group the **student** table according to the **name** field and search for the records in which the maximum score is higher than 95 in the group.

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

Column-Based GROUP BY

Function

This clause 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

GroupBy generates update results in the stream processing table.

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 clause is used to group streams 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);
```

Grouping sets, Rollup, Cube

Function

- The GROUP BY GROUPING SETS generates a result set equivalent to that generated by multiple simple GROUP BY UNION ALL statements. Using GROUPING SETS is more efficient.
- The ROLLUP and CUBE generate multiple groups based on certain rules and then collect statistics by group.
- The result set generated by CUBE contains all the combinations of values in the selected columns.
- The result set generated by ROLLUP contains the combinations of a certain layer structure in the selected columns.

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY groupingItem]
```

Description

Values of **groupingItem** can be **Grouping sets(columnName [, columnName]*)**, **Rollup(columnName [, columnName]*)**, and **Cube(columnName [, columnName]*)**.

Precautions

None

Example

Return the results generated based on **user** and **product**.

```
INSERT INTO temp SELECT SUM(amount)
FROM Orders
GROUP BY GROUPING SETS ((user), (product));
```

GROUP BY Using HAVING

Function

This clause 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. You can use GROUP BY for grouping and the HAVING for filtering.

Precautions

- If the filtering condition is subject to the results of GROUP BY, the HAVING clause, rather than the WHERE clause, must be used for search. HAVING and GROUP BY are used together. Use GROUP BY for grouping and the HAVING 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** by **num**, use the HAVING clause to search for 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;
```

2.4.2 Set Operations

Union/Union ALL/Intersect/Except

Syntax

```
query UNION [ ALL ] | Intersect | Except query
```

Description

- UNION is used to return the union set of multiple query results.
- INTERSECT is used to return the intersection of multiple query results.
- EXCEPT is used to return the difference set of multiple query results.

Precautions

- Set operations 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, UNION takes only distinct records while UNION ALL does not remove duplicates from the result.

Example

Output distinct records found in either Orders1 and Orders2 tables.

```
insert into temp SELECT  * FROM Orders1
  UNION SELECT  * FROM Orders2;
```

IN

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  WHERE column_name IN (value (, value)* ) | query
```

Description

The IN operator allows multiple values to be specified in the WHERE clause. It returns true if the expression exists in the given table subquery.

Precautions

The subquery table must consist of a single column, and the data type of the column must be the same as that of the expression.

Example

Return **user** and **amount** information of the products in **NewProducts** of the **Orders** table.

```
insert into temp SELECT user, amount
FROM Orders
WHERE product IN (
    SELECT product FROM NewProducts
);
```

2.4.3 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 2-41 Array functions

Grouping Window Function	Description
TUMBLE(time_attr, interval)	Defines a tumbling time window. A tumbling time window assigns rows to non-overlapping, continuous windows with a fixed duration (interval). For example, a tumbling window of 5 minutes groups rows in 5 minutes intervals. Tumbling windows can be defined on event-time (stream + batch) or processing-time (stream).
HOP(time_attr, interval, interval)	Defines a hopping time window (called sliding window in the Table API). A hopping time window has a fixed duration (second interval parameter) and hops by a specified hop interval (first interval parameter). If the hop interval is smaller than the window size, hopping windows are overlapping. Thus, rows can be assigned to multiple windows. For example, a hopping window of 15 minutes size and 5 minute hop interval assigns each row to 3 different windows of 15 minute size, which are evaluated in an interval of 5 minutes. Hopping windows can be defined on event-time (stream + batch) or processing-time (stream).
SESSION(time_attr, interval)	Defines a session time window. Session time windows do not have a fixed duration but their bounds are defined by a time interval of inactivity, i.e., a session window is closed if no event appears for a defined gap period. For example a session window with a 30 minute gap starts when a row is observed after 30 minutes inactivity (otherwise the row would be added to an existing window) and is closed if no row is added within 30 minutes. Session windows can work on event-time (stream + batch) or processing-time (stream).

CAUTION

In streaming mode, the **time_attr** argument of the group window function must refer to a valid time attribute that specifies the processing time or event time of rows.

- **event-time**: The type is timestamp(3).
- **processing-time**: No need to specify the type.

In batch mode, the **time_attr** argument of the group window function must be an attribute of type timestamp.

- Window helper functions

  You can use the following helper functions to select the start and end timestamps, as well as the time attribute, for grouping windows.

Table 2-42 Window helper functions

Helper Function	Description
TUMBLE_START(time_attr, interval) HOP_START(time_attr, interval, interval) SESSION_START(time_attr, interval)	Returns the timestamp of the inclusive lower bound of the corresponding tumbling, hopping, or session window.
TUMBLE_END(time_attr, interval) HOP_END(time_attr, interval, interval) SESSION_END(time_attr, interval)	Returns the timestamp of the **exclusive** upper bound of the corresponding tumbling, hopping, or session window. Note: The exclusive upper bound timestamp **cannot** be used as a rowtime attribute in subsequent time-based operations, such as interval joins and group window or over window aggregations.
TUMBLE_ROWTIME(time_attr, interval) HOP_ROWTIME(time_attr, interval, interval) SESSION_ROWTIME(time_attr, interval)	Returns the timestamp of the inclusive upper bound of the corresponding tumbling, hopping, or session window. The resulting attribute is a rowtime attribute that can be used in subsequent time-based operations such as interval joins and group window or over window aggregations.

Helper Function	Description
TUMBLE_PROCTIME(time_attr, interval) HOP_PROCTIME(time_attr, interval, interval) SESSION_PROCTIME(time_attr, interval)	Returns a proctime attribute that can be used in subsequent time-based operations such as interval joins and group window or over window aggregations.

Note: When calling helper functions, it is important to use the same parameters as those used in the **GROUP BY** clause for grouping window functions.

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;
```

TUMBLE WINDOW Extension

Function

The extension functions of the DLI tumbling window are as follows:

- A tumbling window is triggered periodically to reduce latency.

  Before the tumbling window ends, the window can be periodically triggered based on the configured frequency. The compute result from the start to the current time is output, which does not affect the final output. The latest result can be viewed in each period before the window ends.

- Data accuracy is improved.

  You can set a latency for the end of the window. The output of the window is updated according to the configured latency each time a piece of late data reaches.

Precautions

- If you use the INSERT statement to write results to a sink, it must support the upsert mode. Ensure that the result table supports upsert operations and the primary key is defined.
- Latency settings only take effect for event time and not for proctime.
- When calling helper functions, it is important to use the same parameters as those used in the **GROUP BY** clause for grouping window functions.
- If event time is used, watermark must be used. The code is as follows (**order_time** is identified as the event time column and watermark is set to 3 seconds):

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  watermark for order_time as order_time - INTERVAL '3' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourTopic>',
  'properties.bootstrap.servers' = '<yourKafka>:<port>',
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);
```

- If the proctime is used, you need to use the computed column. The code is as follows (**proc** is the processing time column):

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  proc as proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourTopic>',
  'properties.bootstrap.servers' = '<yourKafka>:<port>',
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);
```

Syntax

```
TUMBLE(time_attr, window_interval, period_interval, lateness_interval)
```

Example

The current time attribute column is **testtime**, the window interval is 10 seconds, and the latency is 10 seconds.

```
TUMBLE(testtime, INTERVAL '10' SECOND, INTERVAL '10' SECOND, INTERVAL '10' SECOND)
```

Description

Table 2-43 Parameters

Parameter	Description	Format
time_attr	Event time or processing time attribute column ● **event-time**: The type is timestamp(3). ● **processing-time**: No need to specify the type.	-
window_interval	Duration of the window	● Format 1: **INTERVAL** '10' **SECOND** The window interval is 10 seconds. You can change the value as needed. ● Format 2: **INTERVAL** '10' **MINUTE** The window interval is 10 minutes. You can change the value as needed. ● Format 3: **INTERVAL** '10' **DAY** The window interval is 10 days. You can change the value as needed.
period_interval	Frequency of periodic triggering within the window range. That is, before the window ends, the output result is updated at an interval specified by **period_interval** from the time when the window starts. If this parameter is not set, the periodic triggering policy is not used by default.	
lateness_interval	Time to postpone the end of the window. The system continues to collect the data that reaches the window within **lateness_interval** after the window ends. The output is updated for each data that reaches the window within **lateness_interval**. **NOTE** If the time window is for processing time, **lateness_interval** does not take effect.	

NOTE

Values of **period_interval** and **lateness_interval** cannot be negative numbers.

- If **period_interval** is set to **0**, periodic triggering is disabled for the window.
- If **lateness_interval** is set to **0**, the latency after the window ends is disabled.
- If neither of the two parameters is set, both periodic triggering and latency are disabled and only the regular tumbling window functions are available .
- If only the latency function needs to be used, set period_interval **INTERVAL '0' SECOND**.

Helper Functions

Table 2-44 Helper functions

Helper Function	Description
TUMBLE_START(time_attr, window_interval, period_interval, lateness_interval)	Returns the timestamp of the inclusive lower bound of the corresponding tumbling window.
TUMBLE_END(time_attr, window_interval, period_interval, lateness_interval)	Returns the timestamp of the exclusive upper bound of the corresponding tumbling window.

Example

1. The Kafka is used as the data source table containing the order information, and the JDBC is used as the data result table for statistics on the number of orders settled by a user within 30 seconds. The order ID and window opening time are used as primary keys to collect result statistics in real time to JDBC.

Step 1 Create a datasource connection for the communication with the VPC and subnet where MySQL and Kafka locate and bind the connection to the queue. Set an inbound rule for the security group to allow access of the queue, and test the connectivity of the queue using the MySQL and Kafka addresses. If the connection is successful, the datasource is bound to the queue. Otherwise, the binding fails.

Step 2 Run the following statement to create the **order_count** table in the MySQL Flink database:

```
CREATE TABLE `flink`.`order_count` (
    `user_id` VARCHAR(32) NOT NULL,
    `window_start` TIMESTAMP NOT NULL,
    `window_end` TIMESTAMP NULL,
    `total_num` BIGINT UNSIGNED NULL,
    PRIMARY KEY (`user_id`, `window_start`)
)    ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4
    COLLATE = utf8mb4_general_ci;
```

Step 3 Create a Flink OpenSource SQL job and submit the job. In this example, the window size is 30 seconds, the triggering period is 10 seconds, and the latency is 5 seconds. That is, if the result is updated before the window ends, the intermediate result will be output every 10 seconds. After the watermark is reached and the window ends, the data whose event time is within 5 seconds of the watermark will still be processed and counted in the current window. If the event time exceeds 5 seconds of the watermark, the data will be discarded.

```
CREATE TABLE orders (
  order_id string,
  order_channel string,
  order_time timestamp(3),
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string,
  watermark for order_time as order_time - INTERVAL '3' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourTopic>',
  'properties.bootstrap.servers' = '<yourKafka>:<port>',
```

```
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

CREATE TABLE jdbcSink (
  user_id string,
  window_start timestamp(3),
  window_end timestamp(3),
  total_num BIGINT,
  primary key (user_id, window_start) not enforced
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://<yourMySQL>:3306/flink',
  'table-name' = 'order_count',
  'username' = '<yourUserName>',
  'password' = '<yourPassword>',
  'sink.buffer-flush.max-rows' = '1'
);

insert into jdbcSink select
    order_id,
    TUMBLE_START(order_time, INTERVAL '30' SECOND, INTERVAL '10' SECOND, INTERVAL '5' SECOND),
    TUMBLE_END(order_time, INTERVAL '30' SECOND, INTERVAL '10' SECOND, INTERVAL '5' SECOND),
    COUNT(*) from orders
    GROUP BY user_id, TUMBLE(order_time, INTERVAL '30' SECOND, INTERVAL '10' SECOND, INTERVAL '5'
SECOND);
```

Step 4 Insert data to Kafka. Assume that orders are settled at different time and the order data at 10:00:13 arrives late.

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241000000002", "order_channel":"webShop", "order_time":"2021-03-24 10:00:20",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241000000003", "order_channel":"webShop", "order_time":"2021-03-24 10:00:33",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241000000004", "order_channel":"webShop", "order_time":"2021-03-24 10:00:13",
"pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001",
"user_name":"Alice", "area_id":"330106"}
```

Step 5 Run the following statement in the MySQL database to view the output result. The final result is displayed as follows because the periodic output result cannot be collected:

```
select * from order_count
user_id      window_start         window_end        total_num
0001      2021-03-24 10:00:00  2021-03-24 10:00:30    3
0001      2021-03-24 10:00:30  2021-03-24 10:01:00    1
```

----End

OVER WINDOW

The difference between Over Window and Group Window is that one record is generated from one row in Over Window.

Syntax

```
SELECT agg1(attr1) OVER (
  [PARTITION BY partition_name]
  ORDER BY proctime|rowtime
```

```
 ROWS
 BETWEEN (UNBOUNDED|rowCOUNT) PRECEDING AND CURRENT ROW FROM TABLENAME

SELECT agg1(attr1) OVER (
  [PARTITION BY partition_name]
  ORDER BY proctime|rowtime
  RANGE
  BETWEEN (UNBOUNDED|timeInterval) PRECEDING AND CURRENT ROW FROM TABLENAME
```

Description

Table 2-45 Parameters

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

- All aggregates must be defined in the same window, that is, in the same partition, sort, and range.
- Currently, only windows from PRECEDING (unbounded or bounded) to CURRENT ROW are supported. The range described by FOLLOWING is not supported.
- ORDER BY must be specified for a single time attribute.

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;
```

2.4.4 JOIN

Equi-join

Syntax

```
FROM tableExpression INNER | LEFT | RIGHT | FULL JOIN tableExpression
  ON value11 = value21 [ AND value12 = value22]
```

Precautions

- Currently, only equi-joins are supported, for example, joins that have at least one conjunctive condition with an equality predicate. Arbitrary cross or theta joins are not supported.
- Tables are joined in the order in which they are specified in the FROM clause. Make sure to specify tables in an order that does not yield a cross join (Cartesian product), which are not supported and would cause a query to fail.
- For streaming queries the required state to compute the query result might grow infinitely depending on the type of aggregation and the number of distinct grouping keys. Provide a query configuration with valid retention interval to prevent excessive state size.

Example

```
SELECT *
FROM Orders INNER JOIN Product ON Orders.productId = Product.id;

SELECT *
FROM Orders LEFT JOIN Product ON Orders.productId = Product.id;

SELECT *
FROM Orders RIGHT JOIN Product ON Orders.productId = Product.id;

SELECT *
FROM Orders FULL OUTER JOIN Product ON Orders.productId = Product.id;
```

Time-windowed Join

Function

Each piece of data in a stream is joined with data in different time zones in another stream.

Syntax

```
from t1 JOIN t2 ON t1.key = t2.key AND TIMEBOUND_EXPRESSIO
```

Description

TIMEBOUND_EXPRESSION can be in either of the following formats:

- L.time between LowerBound(R.time) and UpperBound(R.time)
- R.time between LowerBound(L.time) and UpperBound(L.time)
- Comparison expression with the time attributes (L.time/R.time)

Precautions

A time window join requires at least one equi join predicate and a join condition that limits the time of both streams.

For example, use two range predicates (<, <=, >=, or >), a BETWEEN predicate, or an equal predicate that compares the same type of time attributes (such as processing time and event time) in two input tables.

For example, the following predicate is a valid window join condition:

- ltime = rtime
- ltime >= rtime AND ltime < rtime + INTERVAL '10' MINUTE
- ltime BETWEEN rtime - INTERVAL '10' SECOND AND rtime + INTERVAL '5' SECOND

Example

Join all orders shipped within 4 hours with their associated shipments.

```
SELECT *
FROM Orders o, Shipments s
WHERE o.id = s.orderId AND
      o.ordertime BETWEEN s.shiptime - INTERVAL '4' HOUR AND s.shiptime;
```

Expanding arrays into a relation

Precautions

This clause is used to return a new row for each element in the given array. Unnesting WITH ORDINALITY is not yet supported.

Example

```
SELECT users, tag
FROM Orders CROSS JOIN UNNEST(tags) AS t (tag);
```

User-Defined Table Functions

Function

This clause is used to join a table with the results of a table function. ach row of the left (outer) table is joined with all rows produced by the corresponding call of the table function.

Precautions

A left outer join against a lateral table requires a TRUE literal in the ON clause.

Example

The row of the left (outer) table is dropped, if its table function call returns an empty result.

```
SELECT users, tag
FROM Orders, LATERAL TABLE(unnest_udtf(tags)) t AS tag;
```

If a table function call returns an empty result, the corresponding outer row is preserved, and the result padded with null values.

```
SELECT users, tag
FROM Orders LEFT JOIN LATERAL TABLE(unnest_udtf(tags)) t AS tag ON TRUE;
```

Join Temporal Table Function

Function

Precautions

Currently only inner join and left outer join with temporal tables are supported.

Example

Assuming Rates is a temporal table function, the join can be expressed in SQL as follows:

```
SELECT
  o_amount, r_rate
FROM
  Orders,
  LATERAL TABLE (Rates(o_proctime))
WHERE
  r_currency = o_currency;
```

Join Temporal Tables

Function

This clause is used to join the Temporal table.

Syntax

```
SELECT column-names
FROM table1  [AS <alias1>]
[LEFT] JOIN table2 FOR SYSTEM_TIME AS OF table1.proctime [AS <alias2>]
ON table1.column-name1 = table2.key-name1
```

Description

- **table1.proctime** indicates the processing time attribute (computed column) of **table1**.
- **FOR SYSTEM_TIME AS OF table1.proctime** indicates that when the records in the left table are joined with the dimension table on the right, only the snapshot data is used for matching the current processing time dimension table.

Precautions

Only inner and left joins are supported for temporal tables with processing time attributes.

Example

LatestRates is a dimension table (such as HBase table) that is materialized with the latest rate.

```
SELECT
  o.amout, o.currency, r.rate, o.amount * r.rate
FROM
  Orders AS o
  JOIN LatestRates FOR SYSTEM_TIME AS OF o.proctime AS r
  ON r.currency = o.currency;
```

2.4.5 OrderBy & Limit

OrderBy

Function

This clause is used to sort data in ascending order on a time attribute.

Precautions

Currently, only sorting by time attribute is supported.

Example

Sort data in ascending order on the time attribute.

```
SELECT *
FROM Orders
ORDER BY orderTime;
```

Limit

Function

This clause is used to constrain the number of rows returned.

Precautions

This clause is used in conjunction with ORDER BY to ensure that the results are deterministic.

Example

```
SELECT *
FROM Orders
ORDER BY orderTime
LIMIT 3;
```

2.4.6 Top-N

Function

Top-N queries ask for the N smallest or largest values ordered by columns. Both smallest and largest values sets are considered Top-N queries. Top-N queries are useful in cases where the need is to display only the N bottom-most or the N top-most records from batch/streaming table on a condition.

Syntax

```
SELECT [column_list]
FROM (
   SELECT [column_list],
     ROW_NUMBER() OVER ([PARTITION BY col1[, col2...]]
       ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum
   FROM table_name)
WHERE rownum <= N [AND conditions]
```

Description

- ROW_NUMBER(): Allocate a unique and consecutive number to each line starting from the first line in the current partition. Currently, we only support ROW_NUMBER as the over window function. In the future, we will support RANK() and DENSE_RANK().
- PARTITION BY col1[, col2...]: Specifies the partition columns. Each partition will have a Top-N result.

- ORDER BY col1 [asc|desc][, col2 [asc|desc]...]: Specifies the ordering columns. The ordering directions can be different on different columns.
- WHERE rownum <= N: The rownum <= N is required for Flink to recognize this query is a Top-N query. The N represents the N smallest or largest records will be retained.
- [AND conditions]: It is free to add other conditions in the where clause, but the other conditions can only be combined with rownum <= N using AND conjunction.

Precautions

- The TopN query is Result Updating.
- Flink SQL will sort the input data stream according to the order key,
- so if the top N records have been changed, the changed ones will be sent as retraction/update records to downstream.
- If the top N records need to be stored in external storage, the result table should have the same unique key with the Top-N query.

Example

This is an example to get the top five products per category that have the maximum sales in realtime.

```
SELECT *
  FROM (
     SELECT *,
         ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) as row_num
     FROM ShopSales)
  WHERE row_num <= 5;
```

2.4.7 Deduplication

Function

Deduplication removes rows that duplicate over a set of columns, keeping only the first one or the last one.

Syntax

```
SELECT [column_list]
FROM (
   SELECT [column_list],
     ROW_NUMBER() OVER ([PARTITION BY col1[, col2...]]
       ORDER BY time_attr [asc|desc]) AS rownum
   FROM table_name)
WHERE rownum = 1
```

Description

- ROW_NUMBER(): Assigns a unique, sequential number to each row, starting with one.
- PARTITION BY col1[, col2...]: Specifies the partition columns, i.e. the deduplicate key.
- ORDER BY time_attr [asc|desc]: Specifies the ordering column, it must be a time attribute. Currently Flink supports proctime only. Ascending (ASC)

sorting keeps only the first row, while descending (DESC) sorting keeps only the last row.

- WHERE rownum = 1: The rownum = 1 is required for Flink to recognize this query is deduplication.

Precautions

None

Example

The following examples show how to remove duplicate rows on **order_id**. The proctime is an event time attribute.

```
SELECT order_id, user, product, number
  FROM (
     SELECT *,
         ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY proctime ASC) as row_num
     FROM Orders)
  WHERE row_num = 1;
```

2.5 Functions

2.5.1 User-Defined Functions (UDFs)

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

- **Currently, UDF, UDTF, or UDAF custom functions cannot be written using Python.**
- If you use a UDF in a Flink OpenSource SQL job, it is not possible to generate a static stream graph.

POM Dependency

```
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-table-common</artifactId>
        <version>1.10.0</version>
        <scope>provided</scope>
</dependency>
```

Using UDFs

1. Encapsulate the implemented UDFs into a JAR package and upload the package to OBS.

2. In the navigation pane of the DLI management console, choose **Data Management** > **Package Management**. On the displayed page, click **Create** and use the JAR package uploaded to OBS to create a package.
3. In the left navigation, choose **Job Management** and click **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. Click the **Running Parameters** tab of your job, select the UDF JAR and click **Save**.
5. Add the following statement to the SQL statements to use the functions:
```
CREATE FUNCTION udf_test AS 'com.huaweicompany.udf.UdfScalarFunction';
```

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.huaweicompany.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.huaweicompany.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.huaweicompany.udf.UdfAggFunction';
INSERT INTO sink_stream SELECT udaf_test(attr2) FROM source_stream GROUP BY attr1;
```

2.5.2 Type Inference

Scenario

Type inference summarizes the logic for validating input arguments and deriving data types for both the parameters and the result of a function. From a logical perspective, the planner needs information about expected types, precision, and scale. From a JVM perspective, the planner needs information about how internal data structures are represented as JVM objects when calling a user-defined function.

Flink's user-defined functions implement an automatic type inference extraction that derives data types from the function's class and its evaluation methods via reflection. However, this implicit reflective extraction approach is not always successful, for example, the Row type commonly used in UDTF cannot be extracted.

Flink 1.11 introduced a UDF registration interface and used a type inference approach, which does not support **getResultType** overload to declare the returned type in Flink 1.10. If you use this approach, the following exception will be thrown:

```
Caused by: org.apache.flink.table.api.ValidationException: Cannot extract a data type from a pure
'org.apache.flink.types.Row' class. Please use annotations to define field names and field types.
```

With Flink 1.12, the extraction process can be supported by annotating affected parameters, classes, or methods with @DataTypeHint and @FunctionHint.

Code Samples

The table ecosystem (similar to the SQL standard) is a strongly typed API. Therefore, both function parameters and return types must be mapped to a **data type**.

If more advanced type inference logic is required, an implementer can explicitly override the **getTypeInference()** method in every user-defined function.

However, the annotation approach is recommended because it keeps custom type inference logic close to the affected locations and falls back to the default behavior for the remaining implementation.

```
importorg.apache.flink.table.annotation.DataTypeHint;
importorg.apache.flink.table.annotation.FunctionHint;
importorg.apache.flink.table.functions.FunctionContext;
importorg.apache.flink.table.functions.TableFunction;
importorg.apache.flink.types.Row;
publicclassUdfTableFunctionextendsTableFunction<Row>{
    /**
     * Initialization, which is optional
     *@paramcontext
     */
    @Override
     public void open(FunctionContextcontext) {  }

    @FunctionHint(output=@DataTypeHint("ROW<s STRING, i INT>"))
    publicvoideval(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);
        }
    }
    /**
    * The following is optional.
    */
   @Override
   public void close() {}
}
```

Use 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.huaweicompany.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;
```

2.5.3 Parameter Transfer

Scenario

A UDF can be used in many jobs, and some parameter values vary with jobs. To easily modify the parameter values, you can set **pipeline.global-job-parameters** in the **Runtime Configuration** tab on the Flink OpenSource SQL editing page, and then get the parameter values in the UDF code and use the values as you need. You only need to change the parameter values in the runtime configuration tab to pass the new values to the UDF.

Procedure

Use the open(FunctionContext context) method in your UDF to pass parameters through a FunctionContext object. To pass parameters to a job, perform the following steps:

1. Add **pipeline.global-job-parameters** to **Runtime Configuration** on the Flink OpenSource SQL editing page. The format is as follows:

```
pipeline.global-job-parameters=k1:v1,"k2:v1,v2",k3:"str:ing","k4:str""ing"
```

This configuration defines a map as shown in **Table 2-46**

Table 2-46 Examples for pipeline.global-job-parameters

Key	Value
k1	v1
k2	v1,v2
k3	str:ing
k4	str""ing

NOTE

- **FunctionContext#getJobParameter** obtains only the value of **pipeline.global-job-parameters**. You need to add all key-value pairs that will be used in the UDF to **pipeline.global-job-parameters**.
- Keys and values are separated by colons (:). All key-values are connected by commas (,).
- If the key or value contains commas (,), use double quotation marks (") to enclose key or value, for example, **"v1,v2"**.
- If the key or value contains colons (:), use double quotation marks (") to enclose the key or value, for example, **"str:ing"**.
- If the key or value contains a double quotation mark("), use another double quotation mark ("") to escape the first one, and use double quotation marks (") to enclose the key or value, for example, **"str""ing"**.

2. In your UDF code, use **FunctionContext#getJobParameter** to obtain the key-value pairs you set. The code example is as follows:

```
context.getJobParameter("url","jdbc:mysql://xx.xx.xx.xx:3306/table");
context.getJobParameter("driver","com.mysql.jdbc.Driver");
context.getJobParameter("user","user");
context.getJobParameter("password","password");
```

Code Samples

The following sample UDF uses **pipeline.global-job-parameters** to pass parameters such as **url**, **user**, and **password** required for connecting to the database, obtains the **udf_info** table data, and combines this data with the stream data into JSON output.

Table 2-47 udf_info

Key	Value
class	class-4

SimpleJsonBuild.java

```
package udf;

import com.fasterxml.jackson.databind.ObjectMapper;

import org.apache.flink.table.functions.FunctionContext;
import org.apache.flink.table.functions.ScalarFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.HashMap;
import java.util.Map;

public class SimpleJsonBuild extends ScalarFunction {
   private static final Logger LOG = LoggerFactory.getLogger(SimpleJsonBuild.class);
   String remainedKey;
   String remainedValue;
```

```
    private Connection initConnection(Map<String, String> userParasMap) {
        String url = userParasMap.get("url");
        String driver = userParasMap.get("driver");
        String user = userParasMap.get("user");
        String password = userParasMap.get("password");
        Connection conn = null;
        try {
            Class.forName(driver);
            conn = DriverManager.getConnection(url, user, password);
            LOG.info("connect successfully");
        } catch (Exception e) {
            LOG.error(String.valueOf(e));
        }
        return conn;
    }

    @Override
    public void open(FunctionContext context) throws Exception {
        Map<String, String> userParasMap = new HashMap<>();
        Connection connection;
        PreparedStatement pstmt;
        ResultSet rs;

        String url = context.getJobParameter("url","jdbc:mysql://xx.xx.xx.xx:3306/table");
        String driver = context.getJobParameter("driver","com.mysql.jdbc.Driver");
        String user = context.getJobParameter("user","user");
        String password = context.getJobParameter("password","password");

        userParasMap.put("url", url);
        userParasMap.put("driver", driver);
        userParasMap.put("user", user);
        userParasMap.put("password", password);

        connection = initConnection(userParasMap);
        String sql = "select `key`, `value` from udf_info";
        pstmt = connection.prepareStatement(sql);
        rs = pstmt.executeQuery();

        while (rs.next()) {
            remainedKey = rs.getString(1);
            remainedValue = rs.getString(2);
        }
    }

    public String eval(String... params) throws IOException {
        if (params != null && params.length != 0 && params.length % 2 <= 0) {
            HashMap<String, String> hashMap = new HashMap();
            for (int i = 0; i < params.length; i += 2) {
                hashMap.put(params[i], params[i + 1]);
                LOG.debug("now the key is " + params[i].toString() + "; now the value is " + params[i +
1].toString());
            }
            hashMap.put(remainedKey, remainedValue);
            ObjectMapper mapper = new ObjectMapper();
            String result = "{}";
            try {
                result = mapper.writeValueAsString(hashMap);
            } catch (Exception ex) {
                LOG.error("Get result failed." + ex.getMessage());
            }
            LOG.debug(result);
            return result;
        } else {
            return "{}";
        }
    }

    public static void main(String[] args) throws IOException {
        SimpleJsonBuild  sjb = new SimpleJsonBuild();
```

```
        System.out.println(sjb.eval("json1", "json2", "json3", "json4"));
    }
}
```

Add **pipeline.global-job-parameters** to **Runtime Configuration** on the Flink OpenSource SQL editing page. The format is as follows:

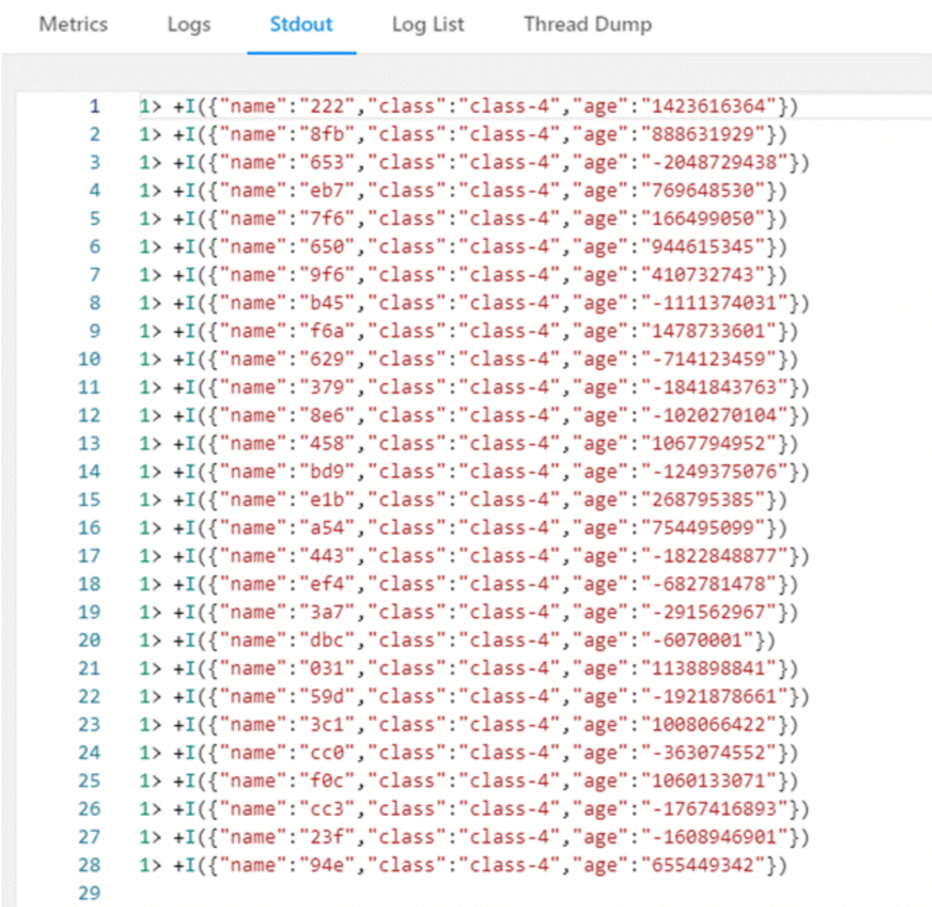
```
pipeline.global-job-parameters=url:'jdbc:mysql://x.x.x.x:xxxx/
swqtest',driver:com.mysql.jdbc.Driver,user:xxx,password:xxx
```

Flink OpenSource SQL

```
create function SimpleJsonBuild AS 'udf.SimpleJsonBuild';
create table dataGenSource(user_id string, amount int) with (
 'connector' = 'datagen',
 'rows-per-second' = '1', --Generate a piece of data per second.
 'fields.user_id.kind' = 'random', --Specify a random generator for the user_id field.
 'fields.user_id.length' = '3' --Limit the length of user_id to 3.
);
create table printSink(message STRING) with ('connector' = 'print');
insert into
printSink
SELECT
SimpleJsonBuild("name", user_id, "age", cast(amount as string))
from
dataGenSource;
```

Output

On the Flin Jobs page, locate your job, and click **More** > **FlinkUI** in the **Operation** column. On the displayed page, click **Task Managers** > **Stdout** to view the job output.



2.5.4 Built-In Functions

2.5.4.1 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 2-48 lists all relational operators supported by Flink SQL.

Table 2-48 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.

Operator	Returned Data Type	Description
A IS NOT DISTINCT FROM B	BOOLEAN	If A is equal to B, **TRUE** is returned. **NULL** indicates A equals B.
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 [ASYMMETRIC | SYMMETRIC]AND C	BOOLEAN	If A is less than B or greater than C, **TRUE** is returned. ● ASYMMETRIC: indicates that B and C are location-related. For example, "A NOT BETWEEN ASYMMETRIC B AND C" is equivalent to "A NOT BETWEEN B AND C". ● SYMMETRIC: indicates that B and C are not location-related. For example, "A NOT BETWEEN SYMMETRIC B AND C" is equivalent to "(A NOT BETWEEN B AND C) OR (A NOT BETWEEN C AND B)".
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.

Operator	Returned Data Type	Description
EXISTS (sub-query)	BOOLEAN	If sub-query returns at least one row, **TRUE** is returned.
value IN (sub-query)	BOOLEAN	If value is equal to a row returned by subquery, **TRUE** is returned.
value NOT IN (sub-query)	BOOLEAN	If value is not equal to a row returned by subquery, **TRUE** is returned.

Precautions

- 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 2-49 lists the calculation rules. A and B indicate logical expressions.

Table 2-49 Logical Operators

Operator	Returned Data 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.

Operator	Returned Data Type	Description
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.

Precautions

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 2-50** lists arithmetic operators supported by Flink SQL.

Table 2-50 Arithmetic Operators

Operator	Returned Data 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.

Operator	Returned Data Type	Description
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).
LOG10(A)	All numeric types	Returns the base 10 logarithms of A.
LOG2(A)	All numeric types	Returns the base 2 logarithm of A.

Operator	Returned Data Type	Description
LOG(B) LOG(A, B)	All numeric types	When called with one argument, returns the natural logarithm of B. When called with two arguments, this function returns the logarithm of B to the base A. B must be greater than 0 and A must be greater than 1.
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.

Operator	Returned Data Type	Description
ACOS(A)	All numeric types	Returns the arc cosine value of A.
ATAN(A)	All numeric types	Returns the arc tangent value of A.
ATAN2(A, B)	All numeric types	Returns the arc tangent of a coordinate (A, B).
COSH(A)	All numeric types	Returns the hyperbolic cosine of A. Return value type is DOUBLE.
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	Returns a number rounded to **d** decimal places for A. For example: round(21.263,2) = 21.26.
PI	All numeric types	Returns the value of **pi**.
E()	All numeric types	Returns the value of **e**.

Operator	Returned Data Type	Description
RAND()	All numeric types	Returns a pseudorandom double value in the range [0.0, 1.0)
RAND(A)	All numeric types	Returns a pseudorandom double value in the range [0.0, 1.0) with an initial seed A. Two RAND functions will return identical sequences of numbers if they have the same initial seed.
RAND_INTEGER(A)	All numeric types	Returns a pseudorandom double value in the range [0.0, A)
RAND_INTEGER(A, B)	All numeric types	Returns a pseudorandom double value in the range [0.0, B) with an initial seed A.
UUID()	All numeric types	Returns a UUID string.
BIN(A)	All numeric types	Returns a string representation of integer A in binary format. Returns NULL if A is NULL.
HEX(A) HEX(B)	All numeric types	Returns a string representation of an integer A value or a string B in hex format. Returns NULL if the argument is NULL.
TRUNCATE(A, d)	All numeric types	Returns a number of truncated to **d** decimal places. Returns NULL if A or **d** is NULL. Example: truncate (42.345, 2) = 42.340 truncate(42.345) = 42.000
PI()	All numeric types	Returns the value of **pi**.

Precautions

Data of the string type is not allowed in arithmetic operations.

2.5.4.2 String Functions

Table 2-51 String Functions

Function	Return Type	Description
string1 || string2	STRING	Returns the concatenation of string1 and string2.
CHAR_LENGTH(string) CHARACTER_LENGTH(string)	INT	Returns the number of characters in the string.
UPPER(string)	STRING	Returns the string in uppercase.
LOWER(string)	STRING	Returns the string in lowercase.
POSITION(string1 IN string2)	INT	Returns the position (start from **1**) of the first occurrence of string1 in string2; returns **0** if string1 cannot be found in string2.
TRIM([BOTH | LEADING | TRAILING] string1 FROM string2)	STRING	Returns a string that removes leading and/or trailing characters string2 from string1.
LTRIM(string)	STRING	Returns a string that removes the left whitespaces from the specified string. For example, **LTRIM(' This is a test String.')** returns **"This is a test String."**.
RTRIM(string)	STRING	Returns a string that removes the right whitespaces from the specified string. For example, **RTRIM('This is a test String. ')** returns **"This is a test String."**.
REPEAT(string, integer)	STRING	Returns a string that repeats the base string integer times. For example, **REPEAT('This is a test String.', 2)** returns **"This is a test String.This is a test String."**.
REGEXP_REPLACE(string1, string2, string3)	STRING	Returns a string from string1 with all the substrings that match a regular expression string2 consecutively being replaced with string3. For example, **REGEXP_REPLACE('foobar', 'oo|ar', '')** returns **"fb"**. **REGEXP_REPLACE('ab\ab', '\\', 'e')** returns **"abeab"**.

Function	Return Type	Description
OVERLAY(string1 PLACING string2 FROM integer1 [FOR integer2])	STRING	Returns a string that replaces integer2 characters of STRING1 with STRING2 from position integer1. The default value of integer2 is the length of string2. For example, **OVERLAY('This is an old string' PLACING ' new' FROM 10 FOR 5)** returns **"This is a new string"**.
SUBSTRING(string FROM integer1 [FOR integer2])	STRING	Returns a substring of the specified string starting from position integer1 with length integer2 (to the end by default). If integer2 is not configured, the substring from integer1 to the end is returned by default.
REPLACE(string1, string2, string3)	STRING	Returns a new string which replaces all the occurrences of string2 with string3 (non-overlapping) from string1. For example, **REPLACE('hello world', 'world', 'flink')** returns **"hello flink"**; **REPLACE('ababab', 'abab', 'z')** returns **"zab"**. **REPLACE('ab\\ab', '\\', 'e')** returns **"abeab"**.
REGEXP_EXTRACT(string1, string2[, integer])	STRING	Returns a string from string1 which extracted with a specified regular expression string2 and a regex match group index integer. Returns NULL, if the parameter is NULL or the regular expression is invalid. For example, **REGEXP_EXTRACT('foothebar', 'foo(.*?)(bar)', 2)"** returns **"bar"**.
INITCAP(string)	STRING	Returns a new form of STRING with the first character of each word converted to uppercase and the rest characters to lowercase.
CONCAT(string1, string2,...)	STRING	Returns a string that concatenates string1, string2, …. For example, **CONCAT('AA', 'BB', 'CC')** returns **"AABBCC"**.
CONCAT_WS(string1, string2, string3,...)	STRING	Returns a string that concatenates string2, string3, … with a separator string1. The separator is added between the strings to be concatenated. Returns NULL if string1 is NULL. If other arguments are NULL, this function automatically skips NULL arguments. For example, **CONCAT_WS('~', 'AA', NULL, 'BB', '', 'CC')** returns **"AA~BB~~CC"**.

Function	Return Type	Description
LPAD(string1, integer, string2)	STRING	Returns a new string from string1 left-padded with string2 to a length of integer characters. If any argument is NULL, NULL is returned. If integer is negative, NULL is returned. If the length of string1 is shorter than integer, returns string1 shortened to integer characters. For example, **LPAD(Symbol,4,Symbol)** returns **"Symbol hi"**. **LPAD('hi',1,'??')** returns **"h"**.
RPAD(string1, integer, string2)	STRING	Returns a new string from string1 right-padded with string2 to a length of integer characters. If any argument is NULL, NULL is returned. If integer is negative, NULL is returned. If the length of string1 is shorter than integer, returns string1 shortened to integer characters. For example, **RPAD('hi',4,'??')** returns **"hi??"**. **RPAD('hi',1,'??')** returns **"h"**.
FROM_BASE64(string)	STRING	Returns the base64-decoded result from string. Returns NULL if string is NULL. For example, **FROM_BASE64('aGVsbG8gd29ybGQ=')** returns **"hello world"**.
TO_BASE64(string)	STRING	Returns the base64-encoded result from string; f string is NULL. Returns NULL if string is NULL. For example, **TO_BASE64(hello world)** returns **"aGVsbG8gd29ybGQ="**.
ASCII(string)	INT	Returns the numeric value of the first character of string. Returns NULL if string is NULL. For example, **ascii('abc')** returns **97.** **ascii(CAST(NULL AS VARCHAR))** returns **NULL**.

Function	Return Type	Description
CHR(integer)	STRING	Returns the ASCII character having the binary equivalent to integer. If integer is larger than 255, we will get the modulus of integer divided by 255 first, and returns CHR of the modulus. Returns NULL if integer is NULL. **chr(97)** returns **a**. **chr(353)** Return **a**.
DECODE(binary, string)	STRING	Decodes the first argument into a String using the provided character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If either argument is NULL, the result will also be NULL.
ENCODE(strinh1, string2)	STRING	Encodes the string1 into a BINARY using the provided string2 character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If either argument is NULL, the result will also be NULL.
INSTR(string1, string2)	INT	Returns the position of the first occurrence of string2 in string1. Returns NULL if any argument is NULL.
LEFT(string, integer)	STRING	Returns the leftmost integer characters from the string. Returns EMPTY String if integer is negative. Returns NULL if any argument is NULL.
RIGHT(string, integer)	STRING	Returns the rightmost integer characters from the string. Returns EMPTY String if integer is negative. Returns NULL if any argument is NULL.
LOCATE(string1, string2[, integer])	INT	Returns the position of the first occurrence of string1 in string2 after position integer. Returns 0 if not found. The value of **integer** defaults to **0**. Returns NULL if any argument is NULL.

Function	Return Type	Description
PARSE_URL(string1, string2[, string3])	STRING	Returns the specified part from the URL. Valid values for string2 include 'HOST', 'PATH', 'QUERY', 'REF', 'PROTOCOL', 'AUTHORITY', 'FILE', and 'USERINFO'. Returns NULL if any argument is NULL. If string2 is QUERY, the key in QUERY can be specified as string3. Example: The **parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST')** returns **'facebook.com'**. **parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1') returns 'v1'**.
REGEXP(string1, string2)	BOOLEAN	Performs a regular expression search on the specified string and returns a BOOLEAN value indicating whether the specified match pattern is found. If it is found, TRUE is returned. string1 indicates the specified string, and string2 indicates the regular expression. Returns NULL if any argument is NULL.
REVERSE(string)	STRING	Returns the reversed string. Returns NULL if any argument is NULL. **NOTE** Note that backquotes must be added to this function, for example, \`REVERSE\`.
SPLIT_INDEX(string1, string2, integer1)	STRING	Splits string1 by the delimiter string2, returns the integerth (zero-based) string of the split strings. Returns NULL if integer is negative. Returns NULL if integer is negative. Returns NULL if any argument is NULL.
STR_TO_MAP(string1[, string2, string3]])	MAP	Returns a map after splitting the string1 into key/value pairs using delimiters. The default value of string2 is **','**. The default value of string3 is **'='**.
SUBSTR(string[, integer1[, integer2]])	STRING	Returns a substring of string starting from position integer1 with length integer2. If integer2 is not specified, the string is truncated to the end.

Function	Return Type	Description
JSON_VAL(STRING json_string, STRING json_path)	STRING	Returns the value of the specified **json_path** from the **json_string**. For details about how to use the functions, see **JSON_VAL Function**. **NOTE** The following rules are listed in descending order of priority. 1. The two arguments **json_string** and **json_path** cannot be **NULL**. 2. The value of **json_string** must be a valid JSON string. Otherwise, the function returns **NULL**. 3. If **json_string** is an empty string, the function returns an empty string. 4. If **json_path** is an empty string or the path does not exist, the function returns **NULL**.

JSON_VAL Function

- Syntax

```
STRING JSON_VAL(STRING json_string, STRING json_path)
```

Table 2-52 Parameters

Parameter	Data Types	Description
json_string	STRING	JSON object to be parsed
json_path	STRING	Path expression for parsing the JSON string For the supported expressions, see **Table 2-53**.

Table 2-53 Expressions supported

Expression	Description
$	Root node in the path
[]	Access array elements
*	Array wildcard
.	Access child elements

- Example
  a. Test input data.
     Test the data source kafka. The message content is as follows:

```
{name:James,age:24,gender:male,grade:{math:95,science:[80,85],english:100}}
{name:James,age:24,gender:male,grade:{math:95,science:[80,85],english:100}]
```

b. Use JSON_VAL in SQL statements.

```
CREATE TABLE kafkaSource (
  `message` string
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourSourceTopic>',
  'properties.bootstrap.servers' =
'<yourKafkaAddress1>:<yourKafkaPort>,<yourKafkaAddress2>:<yourKafkaPort>',
  'properties.group.id' = '<yourGroupId>',
  'scan.startup.mode' = 'latest-offset',
  "format" = "csv",
  "csv.field-delimiter" = "\u0001",
  "csv.quote-character" = "'"
);

CREATE TABLE kafkaSink(
  message1 STRING,
  message2 STRING,
  message3 STRING,
  message4 STRING,
  message5 STRING,
  message6 STRING
) WITH (
  'connector' = 'kafka',
  'topic' = '<yourSinkTopic>',
  'properties.bootstrap.servers' =
'<yourKafkaAddress1>:<yourKafkaPort>,<yourKafkaAddress2>:<yourKafkaPort>',
  "format" = "json"
);

insert into kafkaSink select
JSON_VAL(message,""),
JSON_VAL(message,"$.name"),
JSON_VAL(message,"$.grade.science"),
JSON_VAL(message,"$.grade.science[*]"),
JSON_VAL(message,"$.grade.science[1]"),JSON_VAL(message,"$.grade.dddd")
from kafkaSource;
```

c. Check the output result of the Kafka topic in the sink.

```
{"message1":null,"message2":"swq","message3":"[80,85]","message4":"[80,85]","message5":"85"
,"message6":null}
{"message1":null,"message2":null,"message3":null,"message4":null,"message5":null,"message6":
null}
```

2.5.4.3 Temporal Functions

Table 2-54 lists the time functions supported by Flink OpenSource SQL.

Description

Table 2-54 Temporal Functions

Function	Return Type	Description
DATE string	DATE	Parse the date string (**yyyy-MM-dd**) to a SQL date.
TIME string	TIME	Parse the time string (**HH:mm:ss[.fff]**) to a SQL time.

Function	Return Type	Description
TIMESTAMP string	TIMESTAMP	Convert the time string into a timestamp. The time string format is **yyyy-MM-dd HH:mm:ss[.fff]**.
INTERVAL string range	INTERVAL	interval indicates the interval. There are two forms: • **yyyy-MM** for SQL intervals of months. An interval range might be **YEAR** or **YEAR TO MONTH** for intervals of months. • **dd hh:mm:ss.fff** for SQL intervals of milliseconds. An interval range might be **DAY**, **MINUTE**, **DAY TO HOUR**, or **DAY TO SECOND**. Example: INTERVAL '10 00:00:00.004' DAY TO second indicates that the interval is 10 days and 4 milliseconds. INTERVA**L '10' DAY**: indicates that the interval is 10 days. **INTERVAL '2-10' YEAR TO MONTH** indicates that the interval is two years and ten 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)	BIGINT	Extract part of the time point or interval. Return the part in the int type. For example, extract the date **2006-06-05** and return 5. **EXTRACT(DAY FROM DATE '2006-06-05')** returns **5**.
YEAR(date)	BIGINT	Return the year from SQL date. For example, **YEAR(DATE'1994-09-27')** returns **1994**.

Function	Return Type	Description
QUARTER(date)	BIGINT	Return the quarter of a year (an integer between 1 and 4) from SQL date.
MONTH(date)	BIGINT	Return the month of a year (an integer between 1 and 12) from SQL date. For example, **MONTH(DATE '1994-09-27')** returns **9**.
WEEK(date)	BIGINT	Return the week of a year (an integer between 1 and 53) from SQL date. For example, **WEEK(DATE'1994-09-27')** returns **39**.
DAYOFYEAR(date)	BIGINT	Returns the day of a year (an integer between 1 and 366) from SQL date. For example, **DAYOFYEAR(DATE '1994-09-27')** is **270**.
DAYOFMONTH(date)	BIGINT	Return the day of a month (an integer between 1 and 31) from SQL date. For example, **DAYOFMONTH(DATE'1994-09-27')** returns **27**.
DAYOFWEEK(date)	BIGINT	Return the day of a week (an integer between 1 and 7) from SQL date. Sunday is set to **1**. For example, **DAYOFWEEK(DATE'1994-09-27')** returns **3**.
HOUR(timestamp)	BIGINT	Returns the hour of a day (an integer between 0 and 23) from SQL timestamp. For example, **HOUR(TIMESTAMP '1994-09-27 13:14:15')** returns **13**.
MINUTE(timestamp)	BIGINT	Returns the minute of an hour (an integer between 0 and 59) from SQL timestamp. For example, **MINUTE(TIMESTAMP '1994-09-27 13:14:15')** returns **14**.
SECOND(timestamp)	BIGINT	Returns the second of a minute (an integer between 0 and 59) from SQL timestamp. For example, **SECOND(TIMESTAMP '1994-09-27 13:14:15')** returns **15**.
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, **CEIL(TIME '12:44:31' TO MINUTE)** returns **12:45:00**.
(timepoint1, temporal1) OVERLAPS (timepoint2, temporal2)	BOOLEAN	Return **TRUE** if two time intervals defined by (timepoint1, temporal1) and (timepoint2, temporal2) overlap. Example: **(TIME '2:55:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR)** returns **TRUE**. **(TIME '9:00:00', TIME '10:00:00') OVERLAPS (TIME '10:15:00', INTERVAL '3' HOUR)** returns **FALSE**.
DATE_FORMAT(timestamp, string)	STRING	Convert timestamp to a value of string in the format specified by the date format string.
TIMESTAMPADD(timeintervalunit, interval, timepoint)	TIMESTAMP/ DATE/ TIME	Return the date and time added to **timepoint** based on the result of **interval** and **timeintervalunit**. For example, **TIMESTAMPADD(WEEK, 1, DATE '2003-01-02')** returns **2003-01-09**.
TIMESTAMPDIFF(timepointunit, timepoint1, timepoint2)	INT	Return the (signed) number of timepointunit between **timepoint1** and **timepoint2**. The unit for the interval is given by the first argument, which should be one of the following values: SECOND, MINUTE, HOUR, DAY, MONTH, or YEAR. For example, **TIMESTAMPDIFF(DAY, TIMESTAMP '2003-01-02 10:00:00', TIMESTAMP '2003-01-03 10:00:00')** returns **1**.
CONVERT_TZ(string1, string2, string3)	TIMESTAMP	Convert a datetime **string1** from time zone **string2** to time zone **string3**. For example, **CONVERT_TZ('1970-01-01 00:00:00', 'UTC', 'Country A/City A')** returns **'1969-12-31 16:00:00'**.

Function	Return Type	Description
FROM_UNIXTIME(numeric[, string])	STRING	Return a string representation of the **numeric** argument (in seconds) in the current time zone. The default string format is YYYY-MM-DD hh:mm:ss. For example, **FROM_UNIXTIME(44)** returns **1970-01-01 09:00:44**.
UNIX_TIMESTAMP()	BIGINT	Get current Unix timestamp in seconds.
UNIX_TIMESTAMP(string1[, string2])	BIGINT	Convert date time string **string1** in format **string2** to Unix timestamp (in seconds), using the specified timezone in table config. The default format of **string2** is yyyy-MM-dd HH:mm:ss.
TO_DATE(string1[, string2])	DATE	Convert a date string **string1** with format **string2** to a date. The default format of **string2** is yyyy-MM-dd.
TO_TIMESTAMP(string1[, string2])	TIMESTAMP	Converts date time string string1 with format string2 under the 'UTC+0' time zone to a timestamp. The default format of **string2** is yyyy-MM-dd HH:mm:ss.

DATE

- **Function**

  Returns a SQL date parsed from string in form of **yyyy-MM-dd**.

- **Description**

  ```
  DATE DATE string
  ```

- **Input parameters**

Parameter	Data Types	Parameters
string	STRING	String in the SQL date format. Note that the string must be in the **yyyy-MM-dd** format. Otherwise, an error will be reported.

- **Example**

- Test statement

```
SELECT
   DATE "2021-08-19" AS `result`
FROM
   testtable;
```

- Test Result

result
2021-08-19

TIME

- **Function**

  Returns a SQL time parsed from string in form of **HH:mm:ss[.fff]**.

- **Description**

```
TIME TIME string
```

- **Input parameters**

Parameter	Data Types	Parameters
string	STRING	Time Note that the string must be in the format of **HH:mm:ss[.fff]**. Otherwise, an error will be reported.

- **Example**
  - Test statement

```
SELECT
   TIME "10:11:12" AS `result`,
      TIME "10:11:12.032" AS `result2`
FROM
   testtable;
```

  - Test result

result	result2
10:11:12	10:11:12.032

TIMESTAMP

- **Function**

  Converts the time string into timestamp. The time string format is **yyyy-MM-dd HH:mm:ss[.fff]**. The return value is of the **TIMESTAMP(3)** type.

- **Description**

```
TIMESTAMP(3) TIMESTAMP string
```

- **Input parameters**

Parameter	Data Types	Parameters
string	STRING	Time Note that the string must be in the format of **yyyy-MM-dd HH:mm:ss[.fff]**. Otherwise, an error will be reported.

- **Example**
  - Test statement

```
SELECT
   TIMESTAMP "1997-04-25 13:14:15" AS `result`,
      TIMESTAMP "1997-04-25 13:14:15.032" AS `result2`
FROM
   testtable;
```

  - Test result

result	result2
1997-04-25 13:14:15	1997-04-25 13:14:15.032

INTERVAL

- **Function**
  Parses an interval string.
- **Description**

```
INTERVAL INTERVAL string range
```

- **Input parameters**

Parameter	Data Types	Parameters
string	STRING	Timestamp string used together with the **range** parameter. The string is in either of the following two formats: ● **yyyy-MM** for SQL intervals of months. An interval range might be **YEAR** or **YEAR TO MONTH** for intervals of months. ● **dd hh:mm:ss.fff** for SQL intervals of milliseconds. An interval range might be **DAY**, **MINUTE**, **DAY TO HOUR**, or **DAY TO SECOND**.
range	INTERVAL	Interval range. This parameter is used together with the **string** parameter. Available values are as follows: **YEAR, YEAR To Month, DAY, MINUTE, DAY TO HOUR and DAY TO SECOND**.

- **Example**

Test statement

```
-- indicates that the interval is 10 days and 4 milliseconds.
INTERVAL '10 00:00:00.004' DAY TO second
-- The interval is 10 days.
INTERVAL '10'
-- The interval is 2 years and 10 months.
INTERVAL '2-10' YEAR TO MONTH
```

CURRENT_DATE

- **Function**

  Returns the current SQL time (**yyyy-MM-dd**) in the local time zone. The return value is of the **DATE** type.

- **Description**

  ```
  DATE CURRENT_DATE
  ```

- **Input parameters**

  None

- **Example**
  - Test statement

    ```
    SELECT
       CURRENT_DATE AS `result`
    FROM
       testtable;
    ```

  - Test result

result
2021-10-28

CURRENT_TIME

- **Function**

  Returns the current SQL time (**HH:mm:sss.fff**) in the local time zone. The return value is of the **TIME** type.

- **Description**

  ```
  TIME CURRENT_TIME
  ```

- **Input parameters**

  None

- **Example**
  - Test statement

    ```
    SELECT
       CURRENT_TIME AS `result`
    FROM
       testtable;
    ```

  - Test Result

result
08:29:19.289

CURRENT_TIMESTAMP

- **Function**

  Returns the current SQL timestamp in the local time zone. The return value is of the **TIMESTAMP(3)** type.

- **Description**
  ```
  TIMESTAMP(3) CURRENT_TIMESTAMP
  ```
- **Input parameters**

  None

- **Example**
  - Test statement
    ```
    SELECT
       CURRENT_TIMESTAMP AS `result`
    FROM
       testtable;
    ```
  - Test Result

result
2021-10-28 08:33:51.606

LOCALTIME

- **Function**

  Returns the current SQL time in the local time zone. The return value is of the **TIME** type.

- **Description**
  ```
  TIME LOCALTIME
  ```
- **Input parameters**

  None

- **Example**
  - Test statement
    ```
    SELECT
       LOCALTIME AS `result`
    FROM
       testtable;
    ```
  - Test Result

result
16:39:37.706

LOCALTIMESTAMP

- **Function**

  Returns the current SQL timestamp in the local time zone. The return value is of the **TIMESTAMP(3)** type.

- **Description**
  ```
  TIMESTAMP(3) LOCALTIMESTAMP
  ```

- **Input parameters**

  None

- **Example**
  - Test statement

```
SELECT
   LOCALTIMESTAMP AS `result`
FROM
   testtable;
```

  - Test Result

result
2021-10-28 16:43:17.625

EXTRACT

- **Function**

  Returns a value extracted from the **timeintervalunit** part of temporal. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT EXTRACT(timeinteravlunit FROM temporal)
```

- **Input parameters**

Parameter	Data Types	Parameters
timeinteravlunit	TIMEUNIT	Time unit to be extracted from a time point or interval. The value can be **YEAR**, **QUARTER**, **MONTH**, **WEEK**, **DAY**, **DOY**, **HOUR**, **MINUTE**, **SECOND**.
temporal	DATE/TIME/TIMESTAMP/INTERVAL	Time point or interval

CAUTION

Do not specify a time unit that is not of any time points or intervals. Otherwise, the job fails to be submitted.

For example, an error message is displayed when the following statement is executed because **YEAR** cannot be extracted from **TIME**.

```
SELECT
   EXTRACT(YEAR FROM TIME '12:44:31' ) AS `result`
FROM
   testtable;
```

- **Example**
  - Test statement

```
SELECT
   EXTRACT(YEAR FROM DATE '1997-04-25' ) AS `result`,
      EXTRACT(MINUTE FROM TIME '12:44:31') AS `result2`,
      EXTRACT(SECOND FROM TIMESTAMP '1997-04-25 13:14:15') AS `result3`,
```

```
    EXTRACT(YEAR FROM INTERVAL '2-10' YEAR TO MONTH) AS `result4`,
FROM
  testtable;
```

– Test result

result	result2	result3	result4
1997	44	15	2

YEAR

- **Function**

  Returns the year from a SQL date date. The return value is of the **BIGINT** type.
- **Description**
  ```
  BIGINT YEAR(date)
  ```
- **Input parameters**

Parameter	Data Types	Parameters
date	DATE	SQL date

- **Example**
  – Test statement
  ```
  SELECT
    YEAR(DATE '1997-04-25' ) AS `result`
  FROM
    testtable;
  ```
  – Test result

result
1997

QUARTER

- **Function**

  Returns the quarter of a year (an integer between 1 and 4) from a SQL date date. The return value is of the **BIGINT** type.
- **Description**
  ```
  BIGINT QUARTER(date)
  ```
- **Input parameters**

Parameter	Data Types	Parameters
date	DATE	SQL date

- **Example**
  – Test statement
  ```
  SELECT
    QUARTER(DATE '1997-04-25' ) AS `result`
  ```

```
FROM
    testtable;
```

– Test result

result
2

MONTH

- **Function**

  Returns the month of a year (an integer between 1 and 12) from a SQL date date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT MONTH(date)
```

- **Input parameters**

Parameter	Data Types	Parameters
date	DATE	SQL date

- **Example**
  – Test statement

```
SELECT
    MONTH(DATE '1997-04-25' ) AS `result`
FROM
    testtable;
```

  – Test result

result
4

WEEK

- **Function**

  Returns the week of a year from a SQL date date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT WEEK(date)
```

- **Input parameters**

Parameter	Data Types	Parameters
date	DATE	SQL date

- **Example**
  – Test statement

```
SELECT
    WEEK(DATE '1997-04-25' ) AS `result`
```

```
FROM
    testtable;
```

– Test result

result
17

DAYOFYEAR

- **Function**

  Returns the day of a year (an integer between 1 and 366) from SQL date date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT DAYOFYEAR(date)
```

- **Input parameters**

Parameter	Data Types	Parameters
date	DATE	SQL date

- **Example**
  – Test statement

```
SELECT
    DAYOFYEAR(DATE '1997-04-25' ) AS `result`
FROM
    testtable;
```

  – Test Result

result
115

DAYOFMONTH

- **Function**

  Returns the day of a month (an integer between 1 and 31) from a SQL date date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT DAYOFMONTH(date)
```

- **Input parameters**

Parameter	Data Types	Parameters
date	DATE	SQL date

- **Example**
  – Test statement

```
SELECT
    DAYOFMONTH(DATE '1997-04-25' ) AS `result`
```

```
FROM
    testtable;
```

– Test Result

result
25

DAYOFWEEK

- **Function**

  Returns the day of a week (an integer between 1 and 7) from a SQL date date. The return value is of the **BIGINT** type.

  NOTE

  Note that the start day of a week is Sunday.

- **Description**

```
BIGINT DAYOFWEEK(date)
```

- **Input parameters**

Parameter	Data Types	Parameters
date	DATE	SQL date

- **Example**
  – Test statement

```
SELECT
    DAYOFWEEK(DATE '1997-04-25') AS `result`
FROM
    testtable;
```

  – Test Result

result
6

HOUR

- **Function**

  Returns the hour of a day (an integer between 0 and 23) from SQL timestamp timestamp. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT HOUR(timestamp)
```

- **Input parameters**

Parameter	Data Types	Parameters
timestamp	TIMESTAMP	SQL timestamp

- **Example**
  - Test statement
```
SELECT
   HOUR(TIMESTAMP '1997-04-25 10:11:12') AS `result`
FROM
   testtable;
```
  - Test Result

result
10

MINUTE

- **Function**

  Returns the minute of an hour (an integer between 0 and 59) from a SQL timestamp. The return value is of the **BIGINT** type.
- **Description**
```
BIGINT MINUTE(timestamp)
```
- **Input parameters**

Parameter	Data Types	Parameters
timestamp	TIMESTAMP	SQL timestamp

- **Example**
  - Test statement
```
SELECT
   MINUTE(TIMESTAMP '1997-04-25 10:11:12') AS `result`
FROM
   testtable;
```
  - Test Result

result
11

SECOND

- **Function**

  Returns the second of an hour (an integer between 0 and 59) from a SQL timestamp. The return value is of the **BIGINT** type.
- **Description**
```
BIGINT SECOND(timestamp)
```
- **Input parameters**

Parameter	Data Types	Parameters
timestamp	TIMESTAMP	SQL timestamp

- **Example**
  - Test statement
```
SELECT
    SECOND(TIMESTAMP '1997-04-25 10:11:12') AS `result`
FROM
    testtable;
```
  - Test result

result
12

FLOOR

- **Function**

  Returns a value that rounds **timepoint** down to the time unit **timeintervalunit**.
- **Description**
```
TIME/TIMESTAMP(3) FLOOR(timepoint TO timeintervalunit)
```
- **Input parameters**

Parameter	Data Types	Parameters
timepoint	TIMESTAMP /TIME	SQL time or SQL timestamp
timeintervalunit	TIMEUNIT	Time unit. The value can be **YEAR**, **QUARTER**, **MONTH**, **WEEK**, **DAY**, **DOY**, **HOUR**, **MINUTE**, or **SECOND**.

- **Example**
  - Test statement
```
SELECT
    FLOOR(TIME '13:14:15' TO MINUTE) AS `result`
        FLOOR(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result2`,
        FLOOR(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result3`
FROM    testtable;
```
  - Test result

message	message2	message3
13:14	13:14	1997-04-25T13:14

CEIL

- **Function**

  Returns a value that rounds **timepoint** up to the time unit **timeintervalunit**.
- **Description**
```
TIME/TIMESTAMP(3) CEIL(timepoint TO timeintervalunit)
```
- **Input parameters**

Parameter	Data Types	Parameters
timepoint	TIMESTAMP /TIME	SQL time or SQL timestamp
timeintervalunit	TIMEUNIT	Time unit. The value can be **YEAR**, **QUARTER**, **MONTH**, **WEEK**, **DAY**, **DOY**, **HOUR**, **MINUTE**, or **SECOND**.

- **Example**
  - Test statement

```
SELECT
   CEIL(TIME '13:14:15' TO MINUTE) AS `result`
      CEIL(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result2`,
      CEIL(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result3`
FROM    testtable;
```

  - Test Result

result	result2	result3
13:15	13:15	1997-04-25T13:15

OVERLAPS

- **Function**

  Returns **TRUE** if two time intervals overlap; returns **FALSE** otherwise.
- **Description**

```
BOOLEAN (timepoint1, temporal1) OVERLAPS (timepoint2, temporal2)
```

- **Input parameters**

Parameter	Data Types	Parameters
timepoint1/ timepoint2	DATE/TIME/ TIMESTAMP	Time point
temporal1/ temporal2	DATE/TIME/ TIMESTAMP/ INTERVAL	Time point or interval

NOTE

- **(timepoint, temporal)** is a closed interval.
- The temporal can be of the **DATE**, **TIME**, **TIMESTAMP**, or **INTERVAL** type.
  - When th temporal is **DATE**, **TIME**, or **TIMESTAMP**, **(timepoint, temporal)** indicates an interval between **timepoint** and **temporal**. The temporal can be earlier than the value of **timepoint**, for example, **(DATE '1997-04-25', DATE '1997-04-23')**.
  - When the temporal is **INTERVAL**, **(timepoint, temporal)** indicates an interval between **timepoint** and **timepoint + temporal**.
- Ensure that **(timepoint1, temporal1)** and **(timepoint2, temporal2)** are intervals of the same data type.

- **Example**
  - Test statement

```
SELECT
    (TIME '2:55:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR) AS `result`,
        (TIME '2:30:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR) AS `result2`,
    (TIME '2:30:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:31:00', INTERVAL '2' HOUR) AS `result3`,
    (TIME '9:00:00', TIME '10:00:00') OVERLAPS (TIME '10:00:00', INTERVAL '3' HOUR) AS `result4`,
    (TIMESTAMP '1997-04-25 12:00:00', TIMESTAMP '1997-04-25 12:20:00') OVERLAPS (TIMESTAMP '1997-04-25 13:00:00', INTERVAL '2' HOUR) AS `result5`,
    (DATE '1997-04-23', INTERVAL '2' DAY) OVERLAPS (DATE '1997-04-25', INTERVAL '2' DAY) AS `result6`,
    (DATE '1997-04-25', DATE '1997-04-23') OVERLAPS (DATE '1997-04-25', INTERVAL '2' DAY) AS `result7`
FROM
    testtable;
```

  - Test Result

result	result2	result3	result4	result5	result6	result7
true	true	false	true	false	true	true

DATE_FORMAT

- **Function**

  Converts a timestamp to a value of string in the format specified by the date format string.

- **Description**

```
STRING DATE_FORMAT(timestamp, dateformat)
```

- **Input parameters**

Parameter	Data Types	Parameters
timestamp	TIMESTAMP/ STRING	Time point
dateformat	STRING	String in the date format

- **Example**
  - Test statement

```
SELECT
    DATE_FORMAT(TIMESTAMP '1997-04-25 10:11:12', 'yyyy-MM-dd HH:mm:ss') AS `result`,
        DATE_FORMAT(TIMESTAMP '1997-04-25 10:11:12', 'yyyy-MM-dd') AS `result2`,
    DATE_FORMAT(TIMESTAMP '1997-04-25 10:11:12', 'yy/MM/dd HH:mm') AS `result3`,
        DATE_FORMAT('1997-04-25 10:11:12', 'yyyy-MM-dd') AS `result4`
FROM    testtable;
```

  - Test Result

result	result2	result3	result4
1997-04-25 10:11:12	1997-04-25	97/04/25 10:11	1997-04-25

TIMESTAMPADD

- **Function**

  Returns the date and time by combining **interval** and **timeintervalunit** and adding the combination to **timepoint**.

  NOTE

  The return value of **TIMESTAMPADD** is the value of **timepoint**. An exception is that if the input **timepoint** is of the **TIMESTAMP** type, the return value can be inserted into a table field of the **DATE** type.

- **Description**

```
TIMESTAMP(3)/DATE/TIME TIMESTAMPADD(timeintervalunit, interval, timepoint)
```

- **Input parameters**

Parameter	Data Types	Parameters
timeintervalunit	TIMEUNIT	Time unit.
interval	INT	Interval
timepoint	TIMESTAMP/DATE/TIME	Time point

- **Example**
  - Test statement

```
SELECT
   TIMESTAMPADD(WEEK, 1, DATE '1997-04-25') AS `result`,
      TIMESTAMPADD(QUARTER, 1, TIMESTAMP '1997-04-25 10:11:12') AS `result2`,
   TIMESTAMPADD(SECOND, 2, TIME '10:11:12') AS `result3`
FROM    testtable;
```

  - Test Result

result	result2	result3
1997-05-02	• If this field is inserted into a table field of the **TIMESTAMP** type, **1997-07-25T10:11:12** is returned. • If this field is inserted into a table field of the TIMESTAMP type, 1997-07-25 is returned.	10:11:14

TIMESTAMPDIFF

- **Function**

  Returns the (signed) number of **timepointunit** between **timepoint1** and **timepoint2**. The unit for the interval is given by the first argument.

- **Description**

```
INT TIMESTAMPDIFF(timepointunit, timepoint1, timepoint2)
```

- **Input parameters**

Parameter	Data Types	Parameters
timepointunit	TIMEUNIT	Time unit. The value can be **SECOND**, **MINUTE**, **HOUR**, **DAY**, **MONTH** or **YEAR**.
timepoint1/ timepoint2	TIMESTAMP/ DATE	Time point

- **Example**
  - Test statement

```
SELECT
    TIMESTAMPDIFF(DAY, TIMESTAMP '1997-04-25 10:00:00', TIMESTAMP '1997-04-28 10:00:00')
AS `result`,
        TIMESTAMPDIFF(DAY, DATE '1997-04-25', DATE '1997-04-28') AS `result2`,
    TIMESTAMPDIFF(DAY, TIMESTAMP '1997-04-27 10:00:20', TIMESTAMP '1997-04-25 10:00:00')
AS `result3`
FROM    testtable;
```

  - Test result

result	result2	result3
3	3	-2

CONVERT_TZ

- **Function**

  Converts a datetime **string1** (with default ISO timestamp format **'yyyy-MM-dd HH:mm:ss'**) from time zone **string2** to time zone **string3**.

- **Description**

```
STRING CONVERT_TZ(string1, string2, string3)
```

- **Input parameters**

Parameter	Data Types	Parameters
string1	STRING	SQL timestamp. If the value does not meet the format requirements, **NULL** is returned.

Parameter	Data Types	Parameters
string2	STRING	Time zone before conversion. The format of time zone should be either an abbreviation such as **PST**, a full name such as **Country A/City A**, or a custom ID such as **GMT-08:00**.
string3	STRING	Time zone after conversion. The format of time zone should be either an abbreviation such as **PST**, a full name such as **Country A/City A**, or a custom ID such as **GMT-08:00**.

- **Example**
  - Test statement
```
SELECT
   CONVERT_TZ(1970-01-01 00:00:00, UTC, Country A/City A) AS `result`,
      CONVERT_TZ(1997-04-25 10:00:00, UTC, GMT-08:00) AS `result2`
FROM    testtable;
```
  - Test Result

result	result2
1969-12-31 16:00:00	1997-04-25 02:00:00

FROM_UNIXTIME

- **Function**

  Returns a representation of the **numeric** argument as a value in string format.
- **Description**
```
STRING FROM_UNIXTIME(numeric[, string])
```
- **Input parameters**

Parameter	Data Types	Parameters
numeric	BIGINT	An internal timestamp representing the number of seconds since 1970-01-01 00:00:00 UTC. The value can be generated by the **UNIX_TIMESTAMP()** function.
string	STRING	Time. If this parameter is not specified, the default time format is **yyyy-MM-dd HH:mm:ss** format.

- **Example**
  - Test statement
```
SELECT
   FROM_UNIXTIME(44) AS `result`,
      FROM_UNIXTIME(44, 'yyyy:MM:dd') AS `result2`
FROM    testtable;
```


– Test Result

result	result2
1970-01-01 08:00:44	1970:01:01

UNIX_TIMESTAMP

- **Function**

  Gets current Unix timestamp in seconds. The return value is of the **BIGINT** type.
- **Description**
  ```
  BIGINT UNIX_TIMESTAMP()
  ```
- **Input parameters**

  None
- **Example**
  – Test statement
  ```
  SELECT
     UNIX_TIMESTAMP() AS `result`
  FROM
     table;
  ```
  – Test result

result
1635401982

UNIX_TIMESTAMP(string1[, string2])

- **Function**

  Converts date time **string1** in format **string2** to Unix timestamp (in seconds). The return value is of the **BIGINT** type.
- **Description**
  ```
  BIGINT UNIX_TIMESTAMP(string1[, string2])
  ```
- **Input parameters**

Parameter	Data Types	Parameters
string1	STRING	SQL timestamp string. An error is reported if the value does not comply with the **string2** format.
string2	STRING	Time. If this parameter is not specified, the default time format is **yyyy-MM-dd HH:mm:ss**.

- **Example**
  – Test statement
  ```
  SELECT
     UNIX_TIMESTAMP('1997-04-25', 'yyyy-MM-dd') AS `result`,
  ```

```
        UNIX_TIMESTAMP('1997-04-25 00:00:10', 'yyyy-MM-dd HH:mm:ss') AS `result2`,
        UNIX_TIMESTAMP('1997-04-25 00:00:00') AS `result3`
FROM
    testtable;
```

– Test result

result	result2	result3
861897600	861897610	861897600

TO_DATE

- **Function**

  Converts a date **string1** with format **string2** to a date.

- **Description**

```
DATE TO_DATE(string1[, string2])
```

- **Input parameters**

Parameter	Data Types	Parameters
string1	STRING	SQL timestamp string. If the value is not in the required format, an error is reported.
string2	STRING	Format. If this parameter is not specified, the default time format is **yyyy-MM-dd**.

- **Example**
  – Test statement

```
SELECT
    TO_DATE('1997-04-25') AS `result`,
        TO_DATE('1997:04:25', 'yyyy-MM-dd') AS `result2`,
        TO_DATE('1997-04-25 00:00:00', 'yyyy-MM-dd HH:mm:ss') AS `result3`
FROM
    testtable;
```

  – Test result

result	result2	result3
1997-04-25	1997-04-25	1997-04-25

TO_TIMESTAMP

- **Function**

  Converts date time **string1** with format **string2** to a timestamp.

- **Description**

```
TIMESTAMP TO_TIMESTAMP(string1[, string2])
```

- **Input parameters**

Parameter	Data Types	Parameters
string1	STRING	SQL timestamp string. If the value is not in the required format, **NULL** is returned.
string2	STRING	Date format. If this parameter is not specified, the default format is **yyyy-MM-dd HH:mm:ss**.

- **Example**
  - Test statement

```
SELECT
    TO_TIMESTAMP('1997-04-25', 'yyyy-MM-dd') AS `result`,
        TO_TIMESTAMP('1997-04-25 00:00:00') AS `result2`,
        TO_TIMESTAMP('1997-04-25 00:00:00', 'yyyy-MM-dd HH:mm:ss') AS `result3`
FROM
    testtable;
```

  - Test result

result	result2	result3
1997-04-25 00:00	1997-04-25 00:00	1997-04-25 00:00

2.5.4.4 Conditional Functions

Description

Table 2-55 Conditional Functions

Conditional Functions	Description
CASE value WHEN value1_1 [, value1_2]* THEN result1 [WHEN value2_1 [, value2_2]* THEN result2]* [ELSE resultZ] END	Returns **resultX** when the value is contained in (valueX_1, valueX_2, …). Only the first matched value is returned. When no value matches, returns **result_z** if it is provided and returns **NULL** otherwise.
CASE WHEN condition1 THEN result1 [WHEN condition2 THEN result2]* [ELSE resultZ] END	Returns **resultX** when the first **conditionX** is met. Only the first matched value is returned. When no condition is met, returns **result_z** if it is provided and returns **NULL** otherwise.

Conditional Functions	Description
NULLIF(value1, value2)	Returns NULL if value1 is equal to value2; returns value1 otherwise. For example, **NullIF (5, 5)** returns **NULL**. **NULLIF(5, 0)** returns **5**.
COALESCE(value1, value2 [, value3]*)	Returns the first value (from left to right) that is not NULL from value1, value2, …. For example, **COALESCE(NULL, 5)** returns **5**.
IF(condition, true_value, false_value)	Returns the **true_value** if condition is met, otherwise **false_value**. For example, **IF(5 > 3, 5, 3)** returns **5**.
IS_ALPHA(string)	Returns **TRUE** if all characters in the string are letters, otherwise **FALSE**.
IS_DECIMAL(string)	Returns **TRUE** if string can be parsed to a valid numeric, otherwise **FALSE**.
IS_DIGIT(string)	Returns **TRUE** if all characters in string are digits, otherwise **FALSE**. Otherwise, FALSE is returned.

2.5.4.5 Type Conversion Functions

Syntax

```
CAST(value AS type)
```

Description

This function is used to forcibly convert types.

Precautions

- If the input is **NULL**, **NULL** is returned.
- The **cast** function does not support converting a string to the JSON format.

Example 1: Convert the amount value to an integer.

The following example converts the **amount** value to an integer.

```
insert into temp select cast(amount as INT) from source_stream;
```

Table 2-56 Examples of type conversion functions

Example	Description	Example

cast(v1 as string)	Converts **v1** to a string. The value of **v1** can be of the numeric type or of the timestamp, date, or time type.	Table T1:
		| content (INT) |
		| ------------- |
		| 5 |
		Statement:
		SELECT
		  cast(content as varchar)
		FROM
		  T1;
		Result:
		"5"

cast (v1 as int)	Converts **v1** to the **int** type. The value of **v1** can be a number or a character.	Table T1:
		| content (STRING) |
		| ------------- |
		| "5" |
		Statement:
		SELECT
		  cast(content as int)
		FROM
		  T1;
		Result:
		5

cast(v1 as timestamp)	Converts **v1** to the **timestamp** type. The value of **v1** can be of the **string**, **date**, or **time** type.	Table T1:
		| content (STRING) |
		| ------------- |
		| "2018-01-01 00:00:01" |
		Statement:
		SELECT
		  cast(content as timestamp)
		FROM
		  T1;
		Result:
		1514736001000

cast(v1 as date)	Converts **v1** to the **date** type. The value of **v1** can be of the **string** or **timestamp** type.	Table T1:
		| content (TIMESTAMP) |
		| ------------- |
		| 1514736001000 |
		Statement:
		SELECT
		  cast(content as date)
		FROM
		  T1;
		Result:
		"2018-01-01"

NOTE

Flink jobs do not support the conversion of **bigint** to **timestamp** using CAST. You can convert it using **to_timestamp**.

Example 2:

1. Create a Flink opensource SQL job by referring to **Kafka Source Table** and **Print Result Table**, enter the following job running script, and submit the job.

   Note: When creating a job, set Flink Version to 1.12 in the Running Parameters area on the job editing page, select Save Job Log, and set the OBS bucket for saving job logs to facilitate subsequent job log viewing. Change the values of the parameters in bold in the following script according to the actual situation.

```
CREATE TABLE kafkaSource (
  cast_int_to_string int,
  cast_String_to_int string,
  case_string_to_timestamp string,
  case_timestamp_to_date timestamp
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  "format" = "json"
);

CREATE TABLE printSink (
  cast_int_to_string string,
  cast_String_to_int int,
  case_string_to_timestamp timestamp,
  case_timestamp_to_date date
) WITH (
  'connector' = 'print'
);

insert into printSink select
  cast(cast_int_to_string as string),
  cast(cast_String_to_int as int),
  cast(case_string_to_timestamp as timestamp),
  cast(case_timestamp_to_date as date)
from kafkaSource;
```

2. Connect to the Kafka cluster and send the following test data to the Kafka topic:

```
{"cast_int_to_string":"1", "cast_String_to_int": "1", "case_string_to_timestamp": "2022-04-02 15:00:00",
"case_timestamp_to_date": "2022-04-02 15:00:00"}
```

3. View output.
   - Method 1:
     i. Log in to the DLI management console and choose Job Management > Flink Streaming Jobs.
     ii. Locate the row that contains the target Flink job, and choose More & > FlinkUI in the Operation column.
     iii. On the Flink UI, choose Task Managers, click the task name, and select Stdout to view the job run logs.
   - Method 2: If you select Save Job Log for Running Parameters before submitting the job, perform the following operations:
     i. Log in to the DLI management console and choose Job Management > Flink Streaming Jobs.
     ii. Click the name of the corresponding Flink job, choose Run Log, click OBS Bucket, and locate the folder of the corresponding log based on the job running date.

iii. Go to the folder of the corresponding date, find the folder whose name contains taskmanager, download the taskmanager.out file, and view the result log.

The query result is as follows:

```
+I(1,1,2022-04-02T15:00,2022-04-02)
```

2.5.4.6 Collection Functions

Description

Table 2-57 Collection functions

Collection Functions	Description
CARDINALITY(array)	Returns the number of elements in array.
array '[' integer ']'	Returns the element at position INT in array. The index starts from 1.
ELEMENT(array)	Returns the sole element of array (whose cardinality should be one) Returns **NULL** if array is empty. Throws an exception if array has more than one element.
CARDINALITY(map)	Returns the number of entries in map.
map '[' key ']'	Returns the value specified by key value in map.

2.5.4.7 Value Construction Functions

Description

Table 2-58 Value construction functions

Value Construction Functions	Description
ROW(value1, [, value2]*) (value1, [, value2]*)	Returns a row created from a list of values (value1, value2,…).
ARRAY '[' value1 [, value2]* ']'	Returns an array created from a list of values (value1, value2, …).
MAP '[' key1, value1 [, key2, value2]* ']'	Returns a map created from a list of key-value pairs ((value1, value2), (value3, value4), …). The key-value pair is (key1, value1), (key2, value2).

2.5.4.8 Value Access Functions

Description

Table 2-59 Value access functions

Function	Description
tableName.compositeType.field	Returns the value of a field from a Flink composite type (e.g., Tuple, POJO) by name.
tableName.compositeType.*	Returns a flat representation of a Flink composite type (e.g., Tuple, POJO) that converts each of its direct subtype into a separate field.

2.5.4.9 Hash Functions

Description

Table 2-60 Hash functions

Hash Functions	Description
MD5(string)	Returns the MD5 hash of string as a string of 32 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA1(string)	Returns the SHA-1 hash of string as a string of 40 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA224(string)	Returns the SHA-224 hash of string as a string of 56 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA256(string)	Returns the SHA-256 hash of string as a string of 64 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA384(string)	Returns the SHA-384 hash of string as a string of 96 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA512(string)	Returns the SHA-512 hash of string as a string of 128 hexadecimal digits. Returns **NULL** if **string** is **NULL**.

Hash Functions	Description
SHA2(string, hashLength)	Returns the hash using the SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, or SHA-512). The first argument string is the string to be hashed and the second argument hashLength is the bit length of the result (224, 256, 384, or 512). If either argument is NULL, the result will also be NULL.

2.5.4.10 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 2-61** lists aggregate functions.

Table 2-61 Aggregate functions

Function	Return Type	Description
COUNT([ALL] expression | DISTINCT expression1 [, expression2]*)	BIGINT	Returns the number of input rows for which the expression is not NULL. Use DISTINCT for one unique instance of each value.
COUNT(*) COUNT(1)	BIGINT	Returns the number of input rows.
AVG([ALL | DISTINCT] expression)	DOUBLE	Returns the average (arithmetic mean) of expression across all input rows. Use DISTINCT for one unique instance of each value.
SUM([ALL | DISTINCT] expression)	DOUBLE	Returns the sum of expression across all input rows. Use DISTINCT for one unique instance of each value.
MAX([ALL | DISTINCT] expression)	DOUBLE	Returns the maximum value of expression across all input rows.
MIN([ALL | DISTINCT] expression)	DOUBLE	Returns the minimum value of expression across all input rows.
STDDEV_POP([ALL | DISTINCT] expression)	DOUBLE	Returns the population standard deviation of expression across all input rows.
STDDEV_SAMP([ALL | DISTINCT] expression)	DOUBLE	Returns the sample standard deviation of expression across all input rows.

Function	Return Type	Description
VAR_POP([ALL | DISTINCT] expression)	DOUBLE	Returns the population variance (square of the population standard deviation) of expression across all input rows.
VAR_SAMP([ALL | DISTINCT] expression)	DOUBLE	Returns the sample variance (square of the sample standard deviation) of expression across all input rows.
COLLECT([ALL | DISTINCT] expression)	MULTISET	Returns a multiset of expression across all input rows.
VARIANCE([ALL | DISTINCT] expression)	DOUBLE	Returns the sample variance (square of the sample standard deviation) of expression across all input rows.
FIRST_VALUE(expression)	Actual type	Returns the first value in an ordered set of values.
LAST_VALUE(expression)	Actual type	Returns the last value in an ordered set of values.

2.5.4.11 Table-Valued Functions

2.5.4.11.1 string_split

The **string_split** function splits a target string into substrings based on the specified separator and returns a substring list.

Description

```
string_split(target, separator)
```

Table 2-62 string_split parameters

Parameter	Data Types	Description
target	STRING	Target string to be processed **NOTE** • If **target** is **NULL**, an empty line is returned. • If **target** contains two or more consecutive separators, an empty substring is returned. • If **target** does not contain a specified separator, the original string passed to **target** is returned.

Parameter	Data Types	Description
separator	VARCHAR	Separator. Currently, only single-character separators are supported.

Example

1. Create a Flink OpenSource SQL job by referring to **Kafka Source Table** and **Print Result Table**, enter the following job running script, and submit the job.

   When you create a job, set **Flink Version** to **1.12** in the **Running Parameters** tab. Select **Save Job Log**, and specify the OBS bucket for saving job logs. **Change the values of the parameters in bold as needed in the following script.**

```
CREATE TABLE kafkaSource (
  target STRING,
  separator  VARCHAR
) WITH (
  'connector' = 'kafka',
  'topic' = 'KafkaTopic',
  'properties.bootstrap.servers' = 'KafkaAddress1:KafkaPort,KafkaAddress2:KafkaPort',
  'properties.group.id' = 'GroupId',
  'scan.startup.mode' = 'latest-offset',
  "format" = "json"
);

CREATE TABLE printSink (
  target STRING,
  item STRING
) WITH (
  'connector' = 'print'
);

insert into printSink
  select target,
  item from
  kafkaSource,
  lateral table(string_split(target, separator)) as T(item);
```

2. Connect to the Kafka cluster and send the following test data to the Kafka topic:

```
{"target":"test-flink","separator":"-"}
{"target":"flink","separator":"-"}
{"target":"one-two-ww-three","separator":"-"}
```

   The data is as follows:

Table 2-63 Test table data

target (STRING)	separator (VARCHAR)
test-flink	-
flink	-
one-two-ww-three	-

3. View output.

– Method 1:
  i. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  ii. Locate the row that contains the target Flink job, and choose **More** > **FlinkUI** in the **Operation** column.
  iii. On the Flink UI, choose **Task Managers**, click the task name, and select **Stdout** to view job logs.
– Method 2: If you select **Save Job Log** on the **Running Parameters** tab before submitting the job, perform the following operations:
  i. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Flink Jobs**.
  ii. Click the name of the corresponding Flink job, choose **Run Log**, click **OBS Bucket**, and locate the folder of the log you want to view according to the date.
  iii. Go to the folder of the date, find the folder whose name contains **taskmanager**, download the **taskmanager.out** file, and view result logs.

The query result is as follows:

```
+I(test-flink,test)
+I(test-flink,flink)
+I(flink,flink)
+I(one-two-ww-three,one)
+I(one-two-ww-three,two)
+I(one-two-ww-three,ww)
+I(one-two-ww-three,three)
```

The output data is as follows:

Table 2-64 Result table data

target (STRING)	item (STRING)
test-flink	test
test-flink	flink
flink	flink
one-two-ww-three	one
one-two-ww-three	two
one-two-ww-three	ww
one-two-ww-three	three

3 Flink Opensource SQL 1.10 Syntax Reference

3.1 Constraints and Definitions

3.1.1 Supported Data Types

The DLI SQL syntax supports the following data types:

STRING, BOOLEAN, BYTES, DECIMAL, TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DATE, TIME, TIMESTAMP, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL, ARRAY, MULTISET, MAP, ROW

In the SQL syntax, these types are used to define the data types of columns within a table.

3.1.2 Syntax Definition

3.1.2.1 Data Definition Language (DDL)

3.1.2.1.1 CREATE TABLE

Syntax

```
CREATE TABLE table_name
  (
    { <column_definition> | <computed_column_definition> }[ , ...n]
    [ <watermark_definition> ]
    [ <table_constraint> ][ , ...n]
  )
  [COMMENT table_comment]
  [PARTITIONED BY (partition_column_name1, partition_column_name2, ...)]
  WITH (key1=val1, key2=val2, ...)

<column_definition>:
  column_name column_type [ <column_constraint> ] [COMMENT column_comment]

<column_constraint>:
```

```
  [CONSTRAINT constraint_name] PRIMARY KEY NOT ENFORCED

<table_constraint>:
  [CONSTRAINT constraint_name] PRIMARY KEY (column_name, ...) NOT ENFORCED

<computed_column_definition>:
  column_name AS computed_column_expression [COMMENT column_comment]

<watermark_definition>:
  WATERMARK FOR rowtime_column_name AS watermark_strategy_expression

<source_table>:
  [catalog_name.][db_name.]table_name
```

Function

This clause is used to create a table with a specified name.

Description

COMPUTED COLUMN

A computed column is a virtual column generated using **column_name AS computed_column_expression**. A computed column evaluates an expression that can reference other columns declared in the same table. The column itself is not physically stored within the table. A computed column could be defined using **cost AS price * quantity**. This expression can contain any combination of physical columns, constants, functions, or variables, but cannot contain any subquery.

In Flink, a computed column is used to define the time attribute in **CREATE TABLE** statements. A processing time attribute can be defined easily via **proc AS PROCTIME()** using the system's **PROCTIME()** function. The event time column may be obtained from an existing field. In this case, you can use the computed column to obtain event time. For example, if the original field is not of the **TIMESTAMP(3)** type or is nested in a JSON string, you can use computed columns.

Notes:

- An expression that define a computed column in a source table is calculated after data is read from the data source. The column can be used in the **SELECT** statement.
- A computed column cannot be the target of an **INSERT** statement. In an **INSERT** statement, the schema of the **SELECT** statement must be the same as that of the target table that does not have a computed column.

WATERMARK

The **WATERMARK** clause defines the event time attribute of a table and takes the form **WATERMARK FOR rowtime_column_name AS watermark_strategy_expression**.

rowtime_column_name defines an existing column that is marked as the event time attribute of the table. The column must be of the **TIMESTAMP(3)** type and must be the top-level column in the schema. It can also be a computed column.

watermark_strategy_expression defines the watermark generation strategy. It allows arbitrary non-query expression, including computed columns, to calculate the watermark. The expression return type must be **TIMESTAMP(3)**, which represents the timestamp since the Epoch. The returned watermark will be emitted only if it is non-null and its value is larger than the previously emitted local watermark (to preserve the contract of ascending watermarks). The watermark generation expression is evaluated by the framework for every record. The framework will periodically emit the largest generated watermark. If the current watermark is still identical to the previous one, or is null, or the value of the returned watermark is smaller than that of the last emitted one, then no new watermark will be emitted. Watermark is emitted in an interval defined by **pipeline.auto-watermark-interval** configuration. If watermark interval is 0 ms, the generated watermarks will be emitted per-record if it is not null and greater than the last emitted one.

When using event time semantics, tables must contain an event time attribute and watermarking strategy.

Flink provides several commonly used watermark strategies.

- Strictly ascending timestamps: **WATERMARK FOR rowtime_column AS rowtime_column**.

  Emits a watermark of the maximum observed timestamp so far. Rows that have a timestamp bigger to the max timestamp are not late.

- Ascending timestamps: **WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL '0.001' SECOND**.

  Emits a watermark of the maximum observed timestamp so far minus 1. Rows that have a timestamp bigger or equal to the max timestamp are not late.

- Bounded out of orderness timestamps: **WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL 'string' timeUnit**.

  Emits watermarks, which are the maximum observed timestamp minus the specified delay, for example, **WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL '5' SECOND** is a 5 seconds delayed watermark strategy.

  ```
  CREATE TABLE Orders (
      user BIGINT,
      product STRING,
      order_time TIMESTAMP(3),
      WATERMARK FOR order_time AS order_time - INTERVAL '5' SECOND
  ) WITH ( . . . );
  ```

PRIMARY KEY

Primary key constraint is a hint for Flink to leverage for optimizations. It tells that a column or a set of columns of a table or a view are unique and they do not contain null. Neither of columns in a primary can be nullable. The primary key therefore uniquely identifies a row in a table.

Primary key constraint can be either declared along with a column definition (a column constraint) or as a single line (a table constraint). For both cases, it should only be declared as a singleton. If you define multiple primary key constraints at the same time, an exception would be thrown.

Validity Check

SQL standard specifies that a constraint can either be **ENFORCED** or **NOT ENFORCED**. This controls if the constraint checks are performed on the incoming/ outgoing data. Flink does not own the data therefore the only mode we want to support is the **NOT ENFORCED** mode. It is up to the user to ensure that the query enforces key integrity.

Flink will assume correctness of the primary key by assuming that the columns nullability is aligned with the columns in primary key. Connectors should ensure those are aligned.

Notes: In a **CREATE TABLE** statement, creating a primary key constraint will alter the columns nullability, that means, a column with primary key constraint is not nullable.

PARTITIONED BY

Partition the created table by the specified columns. A directory is created for each partition if this table is used as a filesystem sink.

WITH OPTIONS

Table properties used to create a table source/sink. The properties are usually used to find and create the underlying connector.

The key and value of expression key1=val1 should both be string literal.

Notes: The table registered with CREATE TABLE statement can be used as both table source and table sink. We cannot decide if it is used as a source or sink until it is referenced in the DMLs.

3.1.2.1.2 CREATE VIEW

Syntax

```
CREATE VIEW [IF NOT EXISTS] view_name
  [{columnName [, columnName ]* }] [COMMENT view_comment]
  AS query_expression
```

Function

Create a view with multiple layers nested in it to simplify the development process.

Description

IF NOT EXISTS

If the view already exists, nothing happens.

Example

Create a view named **viewName**.

```
create view viewName as select * from dataSource
```

3.1.2.1.3 CREATE FUNCTION

Syntax

```
CREATE FUNCTION
  [IF NOT EXISTS] function_name
  AS identifier [LANGUAGE JAVA|SCALA]
```

Function

Create a user-defined function.

Description

IF NOT EXISTS

If the function already exists, nothing happens.

LANGUAGE JAVA|SCALA

Language tag is used to instruct Flink runtime how to execute the function. Currently only **JAVA** and **SCALA** are supported, the default language for a function is **JAVA**.

Example

Create a function named **STRINGBACK**.

```
create function STRINGBACK as 'com.dli.StringBack'
```

3.1.2.2 Data Manipulation Language (DML)

Statements

Syntax

```
INSERT INTO table_name [PARTITION part_spec] query

part_spec:  (part_col_name1=val1 [, part_col_name2=val2, ...])

query:
  values
  | {
      select
      | selectWithoutFrom
      | query UNION [ ALL ] query
      | query EXCEPT query
      | query INTERSECT query
    }
    [ ORDER BY orderItem [, orderItem ]* ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start { ROW | ROWS } ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY]

orderItem:
  expression [ ASC | DESC ]

select:
  SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
  [ WINDOW windowName AS windowSpec [, windowName AS windowSpec ]* ]

selectWithoutFrom:
  SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }

projectItem:
```

```
  expression [ [ AS ] columnAlias ]
  | tableAlias . *

tableExpression:
  tableReference [, tableReference ]*
  | tableExpression [ NATURAL ] [ LEFT | RIGHT | FULL ] JOIN tableExpression [ joinCondition ]

joinCondition:
  ON booleanExpression
  | USING '(' column [, column ]* ')'

tableReference:
  tablePrimary
  [ matchRecognize ]
  [ [ 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 ]* ')'

windowRef:
    windowName
  | windowSpec

windowSpec:
    [ windowName ]
    '('
    [ ORDER BY orderItem [, orderItem ]* ]
    [ PARTITION BY expression [, expression ]* ]
    [
        RANGE numericOrIntervalExpression {PRECEDING}
      | ROWS numericExpression {PRECEDING}
    ]
    ')'

matchRecognize:
      MATCH_RECOGNIZE '('
      [ PARTITION BY expression [, expression ]* ]
      [ ORDER BY orderItem [, orderItem ]* ]
      [ MEASURES measureColumn [, measureColumn ]* ]
      [ ONE ROW 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 ]*
      ')'

measureColumn:
      expression AS alias

pattern:
```

```
    patternTerm [ '|' patternTerm ]*

patternTerm:
    patternFactor [ patternFactor ]*

patternFactor:
    variable [ patternQuantifier ]

patternQuantifier:
    '*'
 |  '*?'
 |  '+'
 |  '+?'
 |  '?'
 |  '??'
 |  '{' { [ minRepeat ], [ maxRepeat ] } '}' ['?']
 |  '{' repeat '}'
```

Precautions

Flink SQL uses a lexical policy for identifier (table, attribute, function names) similar to Java:

- The case of identifiers is preserved whether they are quoted.
- Identifiers are matched case-sensitively.
- Unlike Java, back-ticks allow identifiers to contain non-alphanumeric characters (for example **SELECT a AS `my field` FROM t**).

String literals must be enclosed in single quotes (for example, **SELECT'Hello World'**). Two single quotation marks are used for escaping (for example, **SELECT'It''s me.'**). Unicode characters are supported in string literals. If explicit Unicode points are required, use the following syntax:

- Use the backslash (\) as escaping character (default): **SELECT U&'\263A'**
- Use a custom escaping character: **SELECT U&'#263A' UESCAPE '#'**

3.2 Flink OpenSource SQL 1.10 Syntax

This section describes the Flink OpenSource SQL syntax supported by DLI. For details about the parameters and examples, see the syntax description.

Creating Tables

Table 3-1 Syntax for creating tables

Classification	Function
Creating a Source Table	**Kafka Source Table**
	DIS Source Table
	JDBC Source Table
	GaussDB(DWS) Source Table
	Redis Source Table
	HBase Source Table

Classification	Function
	userDefined Source Table
Creating a Result Table	**ClickHouse Result Table**
	Kafka Result Table
	Upsert Kafka Result Table
	DIS Result Table
	JDBC Result Table
	GaussDB(DWS) Result Table
	Redis Result Table
	SMN Result Table
	HBase Result Table
	Elasticsearch Result Table
	User-defined Result Table
Creating a Dimension Table	**JDBC Dimension Table**
	GaussDB(DWS) Dimension Table
	HBase Dimension Table

3.3 Data Definition Language (DDL)

3.3.1 Creating a Source Table

3.3.1.1 Kafka Source Table

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 have built an enhanced datasource connection to connect Flink jobs to Kafka. You have set security group rules as required.

Precautions

SASL_SSL cannot be enabled for the interconnected Kafka cluster.

Syntax

```
create table kafkaSource(
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' WATERMARK FOR rowtime_column_name AS watermark-strategy_expression)
)
with (
  'connector.type' = 'kafka',
  'connector.version' = '',
  'connector.topic' = '',
  'connector.properties.bootstrap.servers' = '',
  'connector.properties.group.id' = '',
  'connector.startup-mode' = '',
  'format.type' = ''
);
```

Parameters

Table 3-2 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **kafka**.
connector.version	Yes	Kafka version. The value can be '**0.10'** or **'0.11'**, which corresponds to Kafka 2.11 to 2.4.0 and other historical versions, respectively.
format.type	Yes	Data deserialization format. The value can be **csv**, **json**, or **avro**.
format.field-delimiter	No	Attribute delimiter. You can customize the attribute delimiter only when the encoding format is CSV. The default delimiter is a comma (,).
connector.topic	Yes	Kafka topic name. Either this parameter or **connector.topic-pattern** is used.
connector.topic-pattern	No	Regular expression for matching the Kafka topic name. Either this parameter or **connector.topic** is used. Example: 'topic.*' '(topic-c|topic-d)' '(topic-a|topic-b|topic-\\d*)' '(topic-a|topic-b|topic-[0-9]*)'

Parameter	Mandatory	Description
connector.properties.bootstrap.servers	Yes	Kafka broker addresses. Use commas (,) to separated them.
connector.properties.group.id	No	Consumer group name
connector.startup-mode	No	Consumer startup mode. The value can be **earliest-offset**, **latest-offset**, **group-offsets**, **specific-offsets** or **timestamp**. The default value is **group-offsets**.
connector.specific-offsets	No	Consumption offset. This parameter is mandatory when **startup-mode** is **specific-offsets**. The value is in the 'partition:0,offset:42;partition:1,offset:300' format.
connector.startup-timestamp-millis	No	Consumption start timestamp. This parameter is mandatory when **startup-mode** is **timestamp**.
connector.properties.*	No	Native Kafka property

Example

- Create table **kafkaSource** and read data encoded in CSV format from Kafka.
```
create table kafkaSource(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT)
with (
  'connector.type' = 'kafka',
  'connector.version' = '0.11',
  'connector.topic' = 'test-topic',
  'connector.properties.bootstrap.servers' = 'xx.xx.xx.xx:9092',
  'connector.properties.group.id' = 'test-group',
  'connector.startup-mode' = 'latest-offset',
  'format.type' = 'csv'
);
```
- Create table **kafkaSource** and read data in non-nested JSON strings from Kafka.

  Assume that the non-nested JSON strings are as follows:
```
{"car_id": 312, "car_owner": "wang", "car_brand": "tang"}
{"car_id": 313, "car_owner": "li", "car_brand": "lin"}
{"car_id": 314, "car_owner": "zhao", "car_brand": "han"}
```
  You can create the table as follows:
```
create table kafkaSource(
  car_id STRING,
  car_owner STRING,
  car_brand STRING
)
with (
  'connector.type' = 'kafka',
  'connector.version' = '0.11',
```

```
  'connector.topic' = 'test-topic',
  'connector.properties.bootstrap.servers' = 'xx.xx.xx.xx:9092',
  'connector.properties.group.id' = 'test-group',
  'connector.startup-mode' = 'latest-offset',
  'format.type' = 'json'
);
```

- Create table **kafkaSource** and read the nested JSON data from Kafka.

  Assume that the JSON data is as follows:

```
{
    "id":"1",
    "type":"online",
    "data":{
        "patient_id":1234,
        "name":"bob1234",
        "age":"Bob",
        "gmt_create":"Bob",
        "gmt_modify":"Bob"
    }
}
```

  You can create the table as follows:

```
CREATE table kafkaSource(
  id STRING,
  type STRING,
  data ROW(
    patient_id STRING,
    name STRING,
    age STRING,
    gmt_create STRING,
    gmt_modify STRING)
)
with (
  'connector.type' = 'kafka',
  'connector.version' = '0.11',
  'connector.topic' = 'test-topic',
  'connector.properties.bootstrap.servers' = 'xx.xx.xx.xx:9092',
  'connector.properties.group.id' = 'test-group',
  'connector.startup-mode' = 'latest-offset',
  'format.type' = 'json'
);
```

3.3.1.2 DIS Source Table

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 *Data Ingestion Service User Guide*.

Syntax

```
create table disSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
)
with (
  'connector.type' = 'dis',
  'connector.region' = '',
  'connector.channel' = '',
  'format-type' = ''
);
```

Parameters

Table 3-3 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Data source type. Set this parameter to **dis**.
connector.region	Yes	Region where DIS for storing the data locates.
connector.ak	No	Access key ID. This parameter must be set in pair with **sk**.
connector.sk	No	Secret access key. This parameter must be set in pair with **ak**.
connector.channel	Yes	Name of the DIS stream where data is located.
connector.partition-count	No	Number of partitions where data will be read. Data in partition 0 to **partition-count** will be read. Neither this parameter or **partition-range** can be configured. If neither of the two parameters is set, all partition data will be read by default.
connector.partition-range	No	Range of partitions where data will be read. Neither this parameter or **partition-count** can be configured. If neither of the two parameters is set, all partition data will be read by default. For example, if you set **partition-range** to **[0:2]**, data in partitions 1, 2, and 3 will be read. The range must be within the DIS stream.
connector.offset	No	Start position from which data will be read. Either this parameter or **start-time** can be configured.

Parameter	Mandatory	Description
connector.start-time	No	Time from which DLI reads 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, the latest data is read.
connector. enable-checkpoint	No	Whether to enable the checkpoint function. The value can be **true** (enabled) or **false** (disabled). The default value is **false**. Do not set this parameter when **offset** or **start-time** is set. If this parameter is set to **true**, **checkpoint-app-name** must be configured.
connector. 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. Do not set this parameter when **offset** or **start-time** is set. If **checkpoint-app-name** is set to **true**, this parameter is mandatory.
connector. checkpoint-interval	No	Interval of checkpoint operations on the DIS source operator. The default value is **60s**. Available value units: d, day/h, hour/min, minute/s, sec, second Do not set this parameter when **offset** or **start-time** is configured.
format.type	Yes	Data coding format. The value can be **csv** or **json**.
format.field-delimiter	No	Attribute delimiter. You can customize the attribute delimiter only when the encoding format is CSV. The default delimiter is a comma (,).

Precautions

None

Example

```
create table disCsvSource (
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed INT,
  total_miles INT)
with (
```



```
  'connector.type' = 'dis',
  'connector.region' = 'ap-southeast-1',
  'connector.channel' = 'disInput',
  'format.type' = 'csv'
);
```

3.3.1.3 JDBC Source Table

Function

The JDBC connector is a Flink's built-in connector to read data from a database.

Prerequisites

- An enhanced datasource connection with the database has been established, so that you can configure security group rules as required.
- You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Syntax

```
create table jbdcSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
)
with (
  'connector.type' = 'jdbc',
  'connector.url' = '',
  'connector.table' = '',
  'connector.username' = '',
  'connector.password' = ''
);
```

Parameters

Table 3-4 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Data source type. Set this parameter to **jdbc**.
connector.url	Yes	Database URL
connector.table	Yes	Name of the table where the data to be read from the database is located
connector.driver	No	Driver required for connecting to the database If you do not set this parameter, the automatically extracted URL will be used.

Parameter	Mandatory	Description
connector.username	No	Database authentication username. This parameter must be configured in pair with **connector.password**.
connector.password	No	Database authentication password. This parameter must be configured in pair with **connector.username**.
connector.read.partition.column	No	Name of the column used to partition the input This parameter is mandatory if **connector.read.partition.lower-bound**, **connector.read.partition.upper-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.lower-bound	No	Lower bound of values to be fetched for the first partition This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.upper-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.upper-bound	No	Upper bound of values to be fetched for the last partition This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.lower-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.num	No	Number of partitions to be created This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.upper-bound**, and **connector.read.partition.upper-bound** are configured.
connector.read.fetch-size	No	Number of rows fetched from the database each time The default value is **0**, indicating the hint is ignored.

Precautions

None

Example

```
create table jdbcSource (
  car_id STRING,
```

```
  car_owner STRING,
  car_age INT,
  average_speed INT,
  total_miles INT)
with (
  'connector.type' = 'jdbc',
  'connector.url' = 'jdbc:mysql://xx.xx.xx.xx:3306/xx',
  'connector.table' = 'jdbc_table_name',
  'connector.driver' = 'com.mysql.jdbc.Driver',
  'connector.username' = 'xxx',
  'connector.password' = 'xxxxxx'
);
```

3.3.1.4 GaussDB(DWS) Source Table

Function

DLI reads data of Flink jobs from GaussDB(DWS). GaussDB(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.

GaussDB(DWS) is an online data processing database based on the cloud infrastructure and platform and helps you mine and analyze massive sets of data.

Prerequisites

- Ensure that you have created a GaussDB(DWS) cluster using your account.

  For details about how to create a GaussDB(DWS) cluster, see "Creating a Cluster" in *Data Warehouse Service Management Guide*.
- A GaussDB(DWS) database table has been created.
- An enhanced datasource connection has been created for DLI to connect to GaussDB(DWS) clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
- You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
)
with (
  'connector.type' = 'gaussdb',
  'connector.url' = '',
  'connector.table' = '',
  'connector.username' = '',
  'connector.password' = ''
);
```


Parameters

Table 3-5 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **gaussdb**.
connector.url	Yes	JDBC connection address. The format is jdbc:postgresql://${ip}:${port}/${dbName}. If the database version is later than 8.1.0, the value format is jdbc:gaussdb://${ip}:${port}/${dbName}.
connector.table	Yes	Name of the table to be operated. If the GaussDB(DWS) table is in a schema, the format is **schema\".\"***Table name*. For details, see the **Example**.
connector.driver	No	JDBC connection driver. The default value is **org.postgresql.Driver**. If the database version is later than 8.1.0, the value is **com.huawei.gauss200.jdbc.Driver**.
connector.username	No	Database authentication user name. This parameter must be configured in pair with **connector.password**.
connector.password	No	Database authentication password. This parameter must be configured in pair with **connector.username**.
connector.read.partition.column	No	Name of the column used to partition the input This parameter is mandatory if **connector.read.partition.lower-bound**, **connector.read.partition.upper-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.lower-bound	No	Lower bound of values to be fetched for the first partition This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.upper-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.upper-bound	No	Upper bound of values to be fetched for the last partition This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.lower-bound**, and **connector.read.partition.num** are configured.

Parameter	Mandatory	Description
connector.read.partition.num	No	Number of partitions to be created This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.upper-bound**, and **connector.read.partition.upper-bound** are configured.
connector.read.fetch-size	No	Number of rows fetched from the database each time The default value is **0**, indicating the hint is ignored.

Example

- If you use the gsjdbc4 driver for connection, set **connector.driver** to **org.postgresql.Driver**. You can omit this parameter because the gsjdbc4 driver is the default one.

  Create table **dwsSource** with data fetched from the **car_info** table that is not in a schema:

```
create table dwsSource(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
  'connector.type' = 'gaussdb',
  'connector.url' = 'jdbc:postgresql://xx.xx.xx.xx:8000/xx',
  'connector.table' = 'car_info',
  'connector.username' = 'xx',
  'connector.password' = 'xx'
);
```

  Create table **dwsSource** with data fetched from GaussDB(DWS) table **test** that is in a schema named **test_schema**:

```
create table dwsSource(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
  'connector.type' = 'gaussdb',
  'connector.url' = 'jdbc:postgresql://xx.xx.xx.xx:8000/xx',
  'connector.table' = 'test_schema\".\"test',
  'connector.username' = 'xx',
  'connector.password' = 'xx'
);
```

- If you use the gsjdbc200 driver for connection, set **connector.driver** to **com.huawei.gauss200.jdbc.Driver**.

  Create table **dwsSource** with data fetched from GaussDB(DWS) table **test** that is in a schema named **ads_game_sdk_base**:

```
create table dwsSource(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
```



```
 'connector.type' = 'gaussdb',
 'connector.table' = 'ads_game_sdk_base\".\"test',
 'connector.driver' = 'com.huawei.gauss200.jdbc.Driver',
 'connector.url' = 'jdbc:gaussdb://xx.xx.xx.xx:8000/xx',
 'connector.username' = 'xx',
 'connector.password' = 'xx'
);
```

3.3.1.5 Redis Source Table

Function

Create a source stream to obtain data from Redis as input for jobs.

Prerequisites

An enhanced datasource connection with Redis has been established, so that you can configure security group rules as required.

- You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
)
with (
  'connector.type' = 'redis',
  'connector.host' = '',
  'connector.port' = ''
);
```

Parameters

Table 3-6 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **redis**.
connector.host	Yes	Redis connector address
connector.port	Yes	Redis connector port
connector.password	No	Redis authentication password
connector.deploy-mode	No	Redis deployment mode. The value can be **standalone** or **cluster**. The default value is **standalone**.

Parameter	Mandatory	Description
connector.table-name	No	Name of the table stored in the Redis. This parameter is mandatory in the Redis Hashmap storage pattern. In this pattern, data is stored to Redis in hashmaps. The hash key is **${table-name}:${ext-key}**, and the field name is the column name. **NOTE** Table storage pattern: **connector.table-name** and **connector.key-column** are used as Redis keys. For the Redis hash type, each key corresponds to a hashmap. A hash key is a field name of the source table, and a hash value is a field value of the source table.
connector.use-internal-schema	No	Whether to use the existing schema in the Redis. This parameter is optional in the Redis Hashmap storage pattern. The default value is **false**.
connector.key-column	No	This parameter is optional in table storage pattern. The value is used as the value of ext-key in the Redis. If this parameter is not set, the value of ext-key is the generated UUID.

Example

Reads data from Redis.

```
create table redisSource(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
 'connector.type' = 'redis',
  'connector.host' = 'xx.xx.xx.xx',
  'connector.port' = '6379',
  'connector.password' = 'xx',
  'connector.table-name' = 'car_info'
);
```

3.3.1.6 HBase Source Table

Function

Create a source stream to obtain data from HBase as input for jobs. HBase is a column-oriented distributed cloud storage system that features enhanced reliability, excellent performance, and elastic scalability. It applies to the storage of massive amounts of data and distributed computing. You can use HBase to build a storage system capable of storing TB- or even PB-level data. With HBase, you can filter and analyze data with ease and get responses in milliseconds, rapidly mining data value. DLI can read data from HBase for filtering, analysis, and data dumping.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
- **If MRS HBase is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.**

  For details, see **Modifying the Host Information** in the Data Lake Insight User Guide.
- You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Syntax

```
create table hbaseSource (
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' watermark for rowtime_column_name as watermark-strategy_expression)
)
with (
  'connector.type' = 'hbase',
  'connector.version' = '1.4.3',
  'connector.table-name' = '',
  'connector.zookeeper.quorum' = ''
);
```

Parameters

Table 3-7 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **hbase**.
connector.version	Yes	The value must be **1.4.3**.
connector.table-name	Yes	HBase table name
connector.zookeeper.quorum	Yes	ZooKeeper address
connector.zookeeper.znode.parent	No	Root directory for ZooKeeper. The default value is **/hbase**.

Parameter	Mandatory	Description
connector.rowkey	No	Content of a compound rowkey to be assigned. The content is assigned to a new field based on the configuration. Example: rowkey1:3,rowkey2:3,... The value 3 indicates the first three bytes of the field. The number cannot be greater than the byte size of the field and cannot be less than 1. **rowkey1:3,rowkey2:3** indicates that the first three bytes of the compound rowkey are assigned to **rowkey1**, and the last three bytes are assigned to **rowkey2**.

Example

```
create table hbaseSource(
  rowkey1 string,
  rowkey2 string,
  info Row<owner string>,
  car ROW<miles string, speed string>
 ) with (
   'connector.type' = 'hbase',
   'connector.version' = '1.4.3',
   'connector.table-name' = 'carinfo',
   'connector.rowkey' = 'rowkey1:1,rowkey2:3',
   'connector.zookeeper.quorum' = 'xxxx:2181'
 );
```

3.3.1.7 userDefined Source Table

Function

You can call APIs to obtain data from the cloud ecosystem or an open source ecosystem and use the obtained data as input of Flink jobs.

Prerequisites

The customized source class needs to inherit the **RichParallelSourceFunction** class and specify the data type as Row.

For example, run **public class MySource extends RichParallelSourceFunction<Row>{}** to declare custom class **MySource**. You need to implement the **open**, **run**, **close**, and **cancel** functions. Encapsulate the class into a JAR file and upload the file through the UDF JAR on the SQL editing page.

Content of the dependent pom configuration file is as follows:

```
<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>
```

Syntax

```
create table userDefinedSource (
  attr_name attr_type
  (',' attr_name attr_type)*
)
with (
  'connector.type' = 'user-defined',
  'connector.class-name' = ''
);
```

Parameters

Table 3-8 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Source type. The value can only be **user-defined**, indicating a custom source.
connector.class-name	Yes	Fully qualified class name of the source class
connector.class-parameter	No	Parameter of the constructor of the source class. Only one parameter of the string type is supported.

Precautions

connector.class-name must be a fully qualified class name.

Example

```
create table userDefinedSource (
  attr1 int,
  attr2 int
)
with (
  'connector.type' = 'user-defined',
  'connector.class-name' = 'xx.xx.MySource'
);
```

3.3.2 Creating a Result Table

3.3.2.1 ClickHouse Result Table

Function

DLI exports Flink job data to ClickHouse result tables.

ClickHouse is a column-based database oriented to online analysis and processing. It supports SQL query and provides good query performance. The aggregation analysis and query performance based on large and wide tables is excellent, which is one order of magnitude faster than other analytical databases. For details, see **Using ClickHouse from Scratch**.

Prerequisites

You have established an enhanced datasource connection to ClickHouse and set the port in the security group rule of the ClickHouse cluster as needed.

For details about how to set up an enhanced datasource connection. For details, see "Enhanced Datasource Connection" in the *Data Lake Insight User Guide*.

For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Precautions

- When you create a ClickHouse cluster for MRS, set the cluster version to MRS 3.1.0 and do not enable Kerberos authentication.
- Do not define a primary key in Flink SQL statements. Do not use any syntax that generates primary keys, such as **insert into clickhouseSink select id, cout(*) from sourceName group by id**.
- Flink supports the following data types: string, tinyint, smallint, int, long, float, double, date, timestamp, decimal, and Array.

  The array supports only the int, bigint, string, float, and double data types.

Syntax

```
create table clickhouseSink (
  attr_name attr_type
  (',' attr_name attr_type)*
)
with (
  'connector.type' = 'clickhouse',
  'connector.url' = '',
  'connector.table' = ''
);
```

Parameters

Table 3-9 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Result table type. Set this parameter to **clickhouse**.

Parameter	Mandatory	Description
connector.url	Yes	ClickHouse URL. Parameter format: **jdbc:clickhouse://** *ClickHouseBalancer instance IP address***:***HTTP port number for ClickHouseBalancer instances***/***Database name* ● IP address of a ClickHouseBalancer instance: Log in to the MRS management console, click a cluster name, and choose **Components** > **ClickHouse** > **Instance** to obtain the service IP address of the ClickHouseBalancer instance. ● HTTP port of a ClickHouseBalancer instance: Log in to the MRS management console, click the target cluster name. On the displayed page, choose **Components** > **ClickHouse**. In the **Service Configuration** tab, choose **ClickHouseBalancer** from the **All Roles** dropdown list and search for **lb_http_port** to configure the parameter. The default value is **21425**. ● The database name is the name of the database created for the ClickHouse cluster.
connector.table	Yes	Name of the ClickHouse table to be created
connector.driver	No	Driver required for connecting to the database ● If this parameter is not specified during table creation, the driver automatically extracts the value from the ClickHouse URL. ● If this parameter is specified during table creation, the value must be **ru.yandex.clickhouse.ClickHouseDriver**.
connector.username	No	Account for connecting the ClickHouse database
connector.password	No	Password for accessing the ClickHouse database
connector.write.flush.max-rows	No	Maximum number of rows to be updated when data is written. The default value is **5000**.
connector.write.flush.interval	No	Interval for data update. The unit can be ms, milli, millisecond/s, sec, second/min or minute.
connector.write.max-retries	No	Maximum number of attempts to write data if failed. The default value is **3**.

Example

Read data from a DIS table and insert the data into the **test** table of ClickHouse database **flinktest**.

1. Create a DIS source table **disSource**.
```
create table disSource(
  attr0 string,
  attr1 TINYINT,
  attr2 smallint,
  attr3 int,
  attr4 bigint,
  attr5 float,
  attr6 double,
  attr7 String,
  attr8 string,
  attr9 timestamp(3),
  attr10 timestamp(3),
  attr11 date,
  attr12 decimal(38, 18),
  attr13 decimal(38, 18)
) with (
  "connector.type" = "dis",
  "connector.region" = "cn-xxxx-x",
  "connector.channel" = "xxxx",
  "format.type" = 'csv'
);
```
2. Create ClickHouse result table **clickhouse** and insert the data from the **disSource** table to the result table.
```
create table clickhouse(
  attr0 string,
  attr1 TINYINT,
  attr2 smallint,
  attr3 int,
  attr4 bigint,
  attr5 float,
  attr6 double,
  attr7 String,
  attr8 string,
  attr9 timestamp(3),
  attr10 timestamp(3),
  attr11 date,
  attr12 decimal(38, 18),
  attr13 decimal(38, 18),
  attr14 array < int >,
  attr15 array < bigint >,
  attr16 array < float >,
  attr17 array < double >,
  attr18 array < varchar >,
  attr19 array < String >
) with (
  'connector.type' = 'clickhouse',
  'connector.url' = 'jdbc:clickhouse://xx.xx.xx.xx:xx/flinktest',
  'connector.table' = 'test'
);

insert into
  clickhouse
select
  attr0,
  attr1,
  attr2,
  attr3,
  attr4,
  attr5,
  attr6,
  attr7,
  attr8,
```

```
  attr9,
  attr10,
  attr11,
  attr12,
  attr13,
  array [attr3, attr3+1],
  array [cast(attr4 as bigint), cast(attr4+1 as bigint)],
  array [cast(attr12 as float), cast(attr12+1 as float)],
  array [cast(attr13 as double), cast(attr13+1 as double)],
  array ['TEST1', 'TEST2'],
  array [attr7, attr7]
from
  disSource;
```

3.3.2.2 Kafka Result Table

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 have built an enhanced datasource connection to connect Flink jobs to Kafka. You have set security group rules as required.

Precautions

SASL_SSL cannot be enabled for the interconnected Kafka cluster.

Syntax

```
create table kafkaSource(
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector.type' = 'kafka',
  'connector.version' = '',
  'connector.topic' = '',
  'connector.properties.bootstrap.servers' = '',
  'format.type' = ''
);
```

Parameters

Table 3-10 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **kafka**.

Parameter	Mandatory	Description
connector.version	No	Kafka version. The value can be '**0.10'** or **'0.11'**, which corresponds to Kafka 2.11 to 2.4.0 and other historical versions, respectively.
format.type	Yes	Data serialization format. The value can be **csv**, **json**, or **avro**.
format.field-delimiter	No	Attribute delimiter. You can customize the attribute delimiter only when the encoding format is CSV. The default delimiter is a comma (,).
connector.topic	Yes	Kafka topic name.
connector.properties.bootstrap.servers	Yes	Kafka broker addresses. Use commas (,) to separated them.
connector.sink-partitioner	No	Partitioner type. The value can be **fixed**, **round-robin**, or **custom**.
connector.sink-partitioner-class	No	Custom partitioner. This parameter is mandatory when **sink-partitioner** is **custom**, for example, **org.mycompany.MyPartitioner**.
update-mode	No	Data update mode. Three write modes are supported: **append**, **retract**, and **upsert**.
connector.properties.*	No	Native properties of Kafka

Example

Output the data in **kafkaSink** to Kafka.

```
create table kafkaSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT)
with (
  'connector.type' = 'kafka',
  'connector.version' = '0.10',
  'connector.topic' = 'test-topic',
  'connector.properties.bootstrap.servers' = 'xx.xx.xx.xx:9092',
  'connector.sink-partitioner' = 'round-robin',
  'format.type' = 'csv'
);
```

3.3.2.3 Upsert Kafka Result Table

Function

DLI exports the output data of the Flink job to Kafka in upsert mode.

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 have built an enhanced datasource connection to connect Flink jobs to Kafka. You have set security group rules as required.

Precautions

SASL_SSL cannot be enabled for the interconnected Kafka cluster.

Syntax

```
create table kafkaSource(
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector.type' = 'upsert-kafka',
  'connector.version' = '',
  'connector.topic' = '',
  'connector.properties.bootstrap.servers' = '',
   'format.type' = ''
);
```

Parameters

Table 3-11 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **upsert-kafka**.
connector.version	No	Kafka version. The value can only be **0.11**.
format.type	Yes	Data serialization format. The value can be **csv**, **json**, or **avro**.
connector.topic	Yes	Kafka topic name
connector.properties.bootstrap.servers	Yes	Kafka broker addresses. Use commas (,) to separated them.
connector.sink-partitioner	No	Partitioner type. The value can be **fixed**, **round-robin**, or **custom**.

Parameter	Mandatory	Description
connector.sink-partitioner-class	No	Custom partitioner. This parameter is mandatory when **sink-partitioner** is **custom**, for example, **org.mycompany.MyPartitioner**.
connector.sink.ignore-retraction	No	Whether to ignore the retraction message. The default value is **false**, indicating that the retraction message is written to Kafka as **null**.
update-mode	No	Data update mode. Three write modes are supported: **append**, **retract**, and **upsert**.
connector.properties.*	No	Native properties of Kafka

Example

```
create table upsertKafkaSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT,
  primary key (car_id) not enforced
)
with (
  'connector.type' = 'upsert-kafka',
  'connector.version' = '0.11',
  'connector.topic' = 'test-topic',
  'connector.properties.bootstrap.servers' = 'xx.xx.xx.xx:9092',
  'format.type' = 'csv'
);
```

3.3.2.4 DIS Result Table

Function

DLI writes the Flink job output data into DIS. The 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 *Data Ingestion Service User Guide*.

Syntax

```
create table disSink (
  attr_name attr_type
  (',' attr_name attr_type)*
```

```
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector.type' = 'dis',
  'connector.region' = '',
  'connector.channel' = '',
  'format.type' = ''
);
```

Parameters

Table 3-12 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Data source type. Set this parameter to **dis**.
connector.region	Yes	Region where DIS for storing the data locates.
connector.ak	No	Access key ID. This parameter must be set in pair with **sk**.
connector.sk	No	Secret access key. This parameter must be set in pair with **ak**.
connector.channel	Yes	Name of the DIS stream where data is located.
format.type	Yes	Data coding format. The value can be **csv** or **json**.
format.field-delimiter	No	Attribute delimiter. You can customize the attribute delimiter only when the encoding format is CSV. The default delimiter is a comma (,).
connector.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.

Precautions

None

Example

Output the data in the **disSink** stream to DIS.

```
create table disSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
)
with (
  'connector.type' = 'dis',
  'connector.region' = 'ap-southeast-1',
```

```
  'connector.channel' = 'disOutput',
  'connector.partition-key' = 'car_id,car_owner',
  'format.type' = 'csv'
);
```

3.3.2.5 JDBC Result Table

Function

DLI exports the output data of the Flink job to RDS.

Prerequisites

- An enhanced datasource connection with the database has been established, so that you can configure security group rules as required.
- You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Syntax

```
create table jdbcSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector.type' = 'jdbc',
  'connector.url' = '',
  'connector.table' = '',
  'connector.driver' = '',
  'connector.username' = '',
  'connector.password' = ''
);
```

Parameters

Table 3-13 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Data source type. Set this parameter to **jdbc**.
connector.url	Yes	Database URL
connector.table	Yes	Name of the table where the data to be read from the database is located
connector.driver	No	Driver required for connecting to the database If you do not set this parameter, the automatically extracted URL will be used.

Parameter	Mandatory	Description
connector.username	No	Username for accessing the database
connector.password	No	Password for accessing the database
connector.write.flush.max-rows	No	Maximum number of rows to be updated when data is written. The default value is **5000**.
connector.write.flush.interval	No	Interval for data update. The unit can be ms, milli, millisecond/s, sec, second/min or minute. If this parameter is not set, the value is not updated based on the interval by default.
connector.write.max-retries	No	Maximum number of attempts to write data if failed. The default value is **3**.
connector.write.exclude-update-columns	No	Columns excluded for data update. The default value is empty, indicating that when data with the same primary key is updated, the update of the specified field is ignored. The primary key column is ignored by default.

Precautions

None

Example

Output data from stream **jdbcSink** to the MySQL database.

```
create table jdbcSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
)
with (
  'connector.type' = 'jdbc',
  'connector.url' = 'jdbc:mysql://xx.xx.xx.xx:3306/xx',
  'connector.table' = 'jdbc_table_name',
  'connector.driver' = 'com.mysql.jdbc.Driver',
  'connector.username' = 'xxx',
  'connector.password' = 'xxxxxx'
);
```

3.3.2.6 GaussDB(DWS) Result Table

Function

DLI outputs the Flink job output data to GaussDB(DWS). GaussDB(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.

GaussDB(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 GaussDB(DWS), see **Data Warehouse Service Management Guide**.

Prerequisites

- Ensure that you have created a GaussDB(DWS) cluster using your account.

  For details about how to create a GaussDB(DWS) cluster, see "Creating a Cluster" in *Data Warehouse Service Management Guide*.
- A GaussDB(DWS) database table has been created.
- An enhanced datasource connection has been created for DLI to connect to GaussDB(DWS) clusters, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
- You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Syntax

NOTE

Do not set all attributes in a GaussDB(DWS) result table to **PRIMARY KEY**.

```
create table dwsSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector.type' = 'gaussdb',
  'connector.url' = '',
  'connector.table' = '',
  'connector.driver' = '',
  'connector.username' = '',
  'connector.password' = ''
);
```

Parameters

Table 3-14 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **gaussdb**.
connector.url	Yes	JDBC connection address. The format is jdbc:postgresql://${ip}:${port}/${dbName}.

Parameter	Mandatory	Description
connector.table	Yes	Name of the table to be operated. If the GaussDB(DWS) table is in a schema, the format is **schema\".\"***Table name*. For details, see the **Example**.
connector.driver	No	JDBC connection driver. The default value is **org.postgresql.Driver**.
connector.username	No	Database authentication user name. This parameter must be configured in pair with **connector.password**.
connector.password	No	Database authentication password. This parameter must be configured in pair with **connector.username**.
connector.write.mode	No	Data write mode. The value can be **copy**, **insert**, or **upsert**. The default value is **upsert**. This parameter must be configured depending on **primary key**. ● If **primary key** is not configured, data can be appended in **copy** and **insert** modes. ● If **primary key** is configured, all the three modes are available. Note: GaussDB(DWS) does not support the update of distribution columns. The primary keys of columns to be updated must cover all distribution columns defined in the GaussDB(DWS) table.
connector.write.flush.max-rows	No	Maximum rows allowed for data flush. If the data size exceeds the value, data flush is triggered. The default value is **5000**.
connector.write.flush.interval	No	Data flush period. Data flush is triggered periodically. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit). If this parameter is not set, the value is not updated based on the interval by default.
connector.write.max-retries	No	Maximum number of attempts to write data. The default value is **3**.
connector.write.merge.filter-key	No	Column to be merged. This parameter takes effects only when PRIMARY KEY is configured and **connector.write.mode** is set to **copy**.
connector.write.escape-string-value	No	Whether to escape values of the string type. The default value is **false**.

Precautions

None

Example

- If you use the gsjdbc4 driver for connection, set **connector.driver** to **org.postgresql.Driver**. You can omit this parameter because the gsjdbc4 driver is the default one.
  - Write data to GaussDB(DWS) in **upsert** mode.

```
create table dwsSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
  'connector.type' = 'gaussdb',
  'connector.url' = 'jdbc:postgresql://xx.xx.xx.xx:8000/xx',
  'connector.table' = 'car_info',
  'connector.username' = 'xx',
  'connector.password' = 'xx',
  'connector.write.mode' = 'upsert',
  'connector.write.flush.interval' = '30s'
);
```

    Create table **dwsSource** with data fetched from GaussDB(DWS) table **test** that is in a schema named **ads_game_sdk_base**:

```
CREATE TABLE ads_rpt_game_sdk_realtime_ada_reg_user_pay_mm (
  ddate DATE,
  dmin TIMESTAMP(3),
  game_appkey VARCHAR,
  channel_id VARCHAR,
  pay_user_num_1m bigint,
  pay_amt_1m bigint,
  PRIMARY KEY (ddate, dmin, game_appkey, channel_id) NOT ENFORCED
) WITH (
  'connector.type' = 'gaussdb',
  'connector.url' = 'jdbc:postgresql://xx.xx.xx.xx:8000/dws_bigdata_db',
  'connector.table' = 'ads_game_sdk_base\".\"test',
  'connector.username' = 'xxxx',
  'connector.password' = 'xxxxx',
  'connector.write.mode' = 'upsert',
  'connector.write.flush.interval' = '30s'
);
```

- If you use the gsjdbc200 driver for connection, set **connector.driver** to **com.huawei.gauss200.jdbc.Driver**.

  Create table **dwsSource** with data fetched from GaussDB(DWS) table **test** that is in a schema named **ads_game_sdk_base**:

```
create table dwsSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
  'connector.type' = 'gaussdb',
  'connector.table' = 'ads_game_sdk_base\".\"test',
  'connector.driver' = 'com.huawei.gauss200.jdbc.Driver',
  'connector.url' = 'jdbc:gaussdb://xx.xx.xx.xx:8000/xx',
  'connector.username' = 'xx',
  'connector.password' = 'xx',
  'connector.write.mode' = 'upsert',
  'connector.write.flush.interval' = '30s'
);
```

3.3.2.7 Redis Result Table

Function

DLI exports the output data of the Flink job to Redis. 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/**.

Prerequisites

An enhanced datasource connection with Redis has been established, so that you can configure security group rules as required.

- You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Syntax

```
create table dwsSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
'connector.type' = 'redis',
  'connector.host' = '',
  'connector.port' = '',
  'connector.password' = '',
  'connector.table-name' = '',
  'connector.key-column' = ''
);
```

Parameters

Table 3-15 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **redis**.
connector.host	Yes	Redis connector address
connector.port	Yes	Redis connector port

Parameter	Mandatory	Description
connector.password	No	Redis authentication password
connector.deploy-mode	No	Redis deployment mode. The value can be **standalone** or **cluster**. The default value is **standalone**.
connector.table-name	No	Name of the table stored in the Redis. This parameter is mandatory in the Redis Hashmap storage pattern. In this pattern, data is stored to Redis in hashmaps. The hash key is **${table-name}:${ext-key}**, and the field name is the column name. **NOTE** Table storage pattern: **connector.table-name** and **connector.key-column** are used as Redis keys. For the Redis hash type, each key corresponds to a hashmap. A hash key is a field name of the source table, and a hash value is a field value of the source table.
connector.key-column	No	This parameter is optional in table storage pattern. The value is used as the value of ext-key in the Redis. If this parameter is not set, the value of ext-key is the generated UUID.
connector.write-schema	No	Whether to write the current schema to the Redis. This parameter is available in table storage pattern. The default value is **false**.
connector.data-type	No	Data types for storage. This parameter is mandatory for a custom storage pattern. Supported values include string, list, hash, and set. In a string, list or set, the number of schema fields must be 2, and the number of hash fields must be 3.
connector.ignore-retraction	No	Whether to ignore the retraction message. The default value is **false**.

Precautions

Either **connector.table-name** or **connector.data-type** must be set.

Example

- Configure the table storage pattern when you configure **connector.table-name**.

  In table storage pattern, data is stored in hash mode, which is different from the basic hash pattern in which the three fields of a table are used as the **key**, **hash_key**, and **hash_value**. The key in table pattern can be specified by **connector.table-name** and **connector.key-column** parameters, all field names in the table are used as **hash_key**, and the field values are written to the hash table as **hash_value**.

```
create table redisSink(
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_speed INT
) with (
  'connector.type' = 'redis',
  'connector.host' = 'xx.xx.xx.xx',
  'connector.port' = '6379',
  'connector.password' = 'xx',
  'connector.table-name'='car_info',
  'connector.key-column'='car_id'
);

insert into redisSink
  (car_id,car_owner,car_brand,car_speed)
  VALUES
  ("A1234","OwnA","A1234",30);
```

- The following example shows how to create a table when **connector.data-type** is set to **string**, **list**, **hash**, or **set**, respectively.
  - String type

    The table contains two columns: key and value.

    ```
    create table redisSink(
      attr1 STRING,
      attr2 STRING
    ) with (
      'connector.type' = 'redis',
      'connector.host' = 'xx.xx.xx.xx',
      'connector.port' = '6379',
      'connector.password' = 'xx',
      'connector.data-type' = 'string'
    );

    insert into redisSink
      (attr1,attr2)
      VALUES
      ("car_id","A1234");
    ```

  - List type

    The table contains two columns: key and value.

    ```
    create table redisSink(
      attr1 STRING,
      attr2 STRING
    ) with (
      'connector.type' = 'redis',
      'connector.host' = 'xx.xx.xx.xx',
      'connector.port' = '6379',
      'connector.password' = 'xx',
      'connector.data-type' = 'list'
    );

    insert into redisSink
      (attr1,attr2)
      VALUES
      ("car_id","A1234");
    ```

  - Set type

    The table contains two columns: key and value.

    ```
    create table redisSink(
      attr1 STRING,
      attr2 STRING
    ) with (
      'connector.type' = 'redis',
      'connector.host' = 'xx.xx.xx.xx',
    ```

```
  'connector.port' = '6379',
  'connector.password' = 'xx',
  'connector.data-type' = 'set'
);

insert into redisSink
  (attr1,attr2)
  VALUES
  ("car_id","A1234");
```

– Hash type

The table contains three columns: key, hash_key, and hash_value.

```
create table redisSink(
  attr1 STRING,
  attr2 STRING,
  attr3 STRING
) with (
  'connector.type' = 'redis',
  'connector.host' = 'xx.xx.xx.xx',
  'connector.port' = '6379',
  'connector.password' = 'xx',
  'connector.data-type' = 'hash'
);

insert into redisSink
  (attr1,attr2,attr3)
  VALUES
  ("car_info","car_id","A1234");
```

3.3.2.8 SMN Result Table

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.

Syntax

```
create table smnSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector.type' = 'smn',
  'connector.region' = '',
  'connector.topic-urn' = '',
  'connector.message-subject' = '',
  'connector.message-column' = ''
);
```

Parameters

Table 3-16 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Sink data type. Set this parameter to **smn**, which means that data is stored to SMN.
connector.region	Yes	Region where SMN belongs
connector.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. Either of **topic_urn** and **urn_column** must be configured. If both of them are configured, the **topic_urn** setting takes precedence.
connector.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.
connector.message-subject	Yes	Message subject sent by SMN. This parameter can be customized.
connector.message-column	Yes	Column name in the current table. Data in this column is the message content and is customized. Currently, only text messages are supported.

Precautions

None

Example

Write the data to the target of SMN topic. The topic of the message sent by SMN is **test**, and the message content is the data in the **attr1** column.

```
create table smnSink (
  attr1 STRING,
  attr2 STRING
)
with (
  'connector.type' = 'smn',
  'connector.region' = 'ap-southeast-1',
```

```
 'connector.topic-urn' = 'xxxxxx',
 'connector.message-subject' = 'test',
 'connector.message-column' = 'attr1'
);
```

3.3.2.9 HBase Result Table

Function

DLI outputs the job data to HBase. HBase is a column-oriented distributed cloud storage system that features enhanced reliability, excellent performance, and elastic scalability. It applies to the storage of massive amounts of data and distributed computing. You can use HBase to build a storage system capable of storing TB- or even PB-level data. With HBase, you can filter and analyze data with ease and get responses in milliseconds, rapidly mining data value. Structured and semi-structured key-value data can be stored, including messages, reports, recommendation data, risk control data, logs, and orders. With DLI, you can write massive volumes of data to HBase at a high speed and with low latency.

Prerequisites

An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.

- **If MRS HBase is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.**

  For details, see **Modifying the Host Information** in the Data Lake Insight User Guide.
- You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
- For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Syntax

```
create table hbaseSink (
  attr_name attr_type
  (',' attr_name attr_type)*
)
with (
  'connector.type' = 'hbase',
  'connector.version' = '1.4.3',
  'connector.table-name' = '',
  'connector.zookeeper.quorum' = ''
);
```

Parameters

Table 3-17 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **hbase**.
connector.version	Yes	The value must be **1.4.3**.
connector.table-name	Yes	HBase table name
connector.zookeeper.quorum	Yes	ZooKeeper address
connector.zookeeper.znode.parent	No	Root directory for ZooKeeper. The default value is **/hbase**.
connector.write.buffer-flush.max-size	No	Maximum buffer size for each data write. The default value is 2 MB. The unit is MB.
connector.write.buffer-flush.max-rows	No	Maximum number of data records that can be updated each time
connector.write.buffer-flush.interval	No	Update time. The default value is **0s**. Example value: **2s**.
connector.rowkey	No	Content of a compound rowkey to be assigned. The content is assigned to a new field based on the configuration. Example: rowkey1:3,rowkey2:3, ... The value 3 indicates the first three bytes of the field. The number cannot be greater than the byte size of the field and cannot be less than 1.

Example

```
create table hbaseSink(
 rowkey string,
 name string,
 i Row<geneder string, age int>,
 j Row<address string>
) with (
```

```
  'connector.type' = 'hbase',
  'connector.version' = '1.4.3',
  'connector.table-name' = 'sink',
  'connector.rowkey' = 'rowkey:1,name:3',
  'connector.write.buffer-flush.max-rows' = '5',
  'connector.zookeeper.quorum' = 'xxxx:2181'
);
```

3.3.2.10 Elasticsearch Result Table

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. For more information about CSS, see **Cloud Search Service User Guide**.

Prerequisites

- Ensure that you have created a cluster on CSS using your account.

  If you need to access Elasticsearch using the cluster username and password, enable the security mode and disable HTTPS for the created CSS 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 CSS. You can also set the security group rules as required.
  - You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Precautions

- Currently, only CSS 7.X and later versions are supported. Version 7.6.2 is recommended.
- Do not enable the security mode for the CSS cluster if **connector.username** and **connector.password** are not configured.
- ICMP must be enabled for the security group inbound rule of the CSS cluster.

Syntax

```
create table esSink (
  attr_name attr_type
  (',' attr_name attr_type)*
  (','PRIMARY KEY (attr_name, ...) NOT ENFORCED)
)
with (
  'connector.type' = 'elasticsearch',
  'connector.version' = '7',
  'connector.hosts' = 'http://xxxx:9200',
```

```
  'connector.index' = '',
  'connector.document-type' = '',
  'update-mode' = '',
  'format.type' = 'json'
);
```

Parameters

Table 3-18 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **elasticsearch**.
connector.version	Yes	Elasticsearch version Currently, only version 7 can be used. That is, the value of this parameter can only be **7**.
connector.hosts	Yes	Host name of the cluster where Elasticsearch locates. Use semicolons (;) to separate multiple host names. Ensure that the host name starts with **http**, for example, **http://**x.x.x.x**:9200**.
connector.index	Yes	Elasticsearch index name
connector.document-type	Yes	Elasticsearch type name This attribute is invalid because Elasticsearch 7 uses the default **_doc** type.
update-mode	Yes	Data update mode of the sink. The value can be **append** or **upsert**.
connector.key-delimiter	No	Delimiter of compound primary keys. The default value is _.
connector.key-null-literal	No	Character used to replace **null** in keys.
connector.failure-handler	No	Policy used when an Elasticsearch request fails. The default value is **fail**. **fail**: An exception is thrown when the request fails and the job fails. **ignore**: The failed request is ignored. **retry-rejected**: If the request fails because the queue running the Elasticsearch node is full, the request is resent and no failure is reported. **custom**: A custom policy is used.
connector.failure-handler-class	No	Custom processing mode used to handle a failure

Parameter	Mandatory	Description
connector.flush-on-checkpoint	No	Whether the connector waits for all pending action requests to be acknowledged by Elasticsearch on checkpoints. The default value **true** indicates that wait for all pending action requests on checkpoints. If you set this parameter to false, the connector will not wait for the requests.
connector.bulk-flush.max-actions	No	Maximum number of records that can be written in a batch
connector.bulk-flush.max-size	No	Maximum total amount of data to be written in batches. Specify the unit when you configure this parameter. The unit is MB.
connector.bulk-flush.interval	No	Update interval for batch writing. The unit is milliseconds and is not required.
format.type	Yes	Data format. Currently, only JSON is supported.
connector.username	No	Account of the cluster where Elasticsearch locates. This parameter and must be configured in pair with **connector.password**. If the account and password are used, the security mode must be enabled and HTTPS must be disabled for the created CSS cluster.
connector.password	No	Password of the cluster where Elasticsearch locates. This parameter must be configured in pair with **connector.username**.

Example

```
create table sink1(
  attr1 string,
  attr2 int
) with (
  'connector.type' = 'elasticsearch',
  'connector.version' = '7',
  'connector.hosts' = 'http://xxxx:9200',
  'connector.index' = 'es',
  'connector.document-type' = 'one',
  'update-mode' = 'append',
  'format.type' = 'json'
);
```

3.3.2.11 OpenTSDB Result Table

Function

OpenTSDB is a distributed, scalable time series database based on HBase. OpenTSDB is designed to collect monitoring information of a large-scale cluster and query data in seconds, facilitating querying and storing massive amounts of monitoring data in common databases. OpenTSDB can be used for system monitoring and measurement as well as collection and monitoring of IoT data, financial data, and scientific experimental results.

DLI uses enhanced datasource connections to write the output of Flink jobs to OpenTSDB.

Prerequisites

- The OpenTSDB service has been enabled.
- An enhanced datasource connection has been created for DLI to connect to OpenTSDB, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - For details about how to set up an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.

Syntax

```
create table tsdbSink (
  attr_name attr_type
  (',' attr_name attr_type)*
)
with (
  'connector.type' = 'opentsdb',
  'connector.region' = '',
  'connector.tsdb-metrics' = '',
  'connector.tsdb-timestamps' = '',
  'connector.tsdb-values' = '',
  'connector.tsdb-tags' = '',
  'connector.tsdb-link-address' = ''
);
```

Parameters

Table 3-19 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **opentsdb**.
connector.region	Yes	Region where OpenTSDB locates
connector.tsdb-metrics	Yes	Metrics of data points, which can be specified through parameter configurations. The number of metrics must be 1 or the same as the number of **connector.tsdb-values**. Use semicolons (;) to separate multiple metrics.

Parameter	Mandatory	Description
connector.tsdb-timestamps	Yes	Timestamps of data points. Only dynamic columns are supported. The data type can be int, bigint, or string. Only numbers are supported. The number of metrics must be 1 or the same as the number of **connector.tsdb-values**. Use semicolons (;) to separate multiple timestamps.
connector.tsdb-values	Yes	Values of data points. You can specify dynamic columns or constant values. Separate multiple values with semicolons (;).
connector.tsdb-tags	Yes	Tags of data points. Each tag contains at least one tag value and a maximum of eight tag values. Separate multiple tags by commas (,). You can specify the tags by parameters. The number of metrics must be 1 or the same as the number of **connector.tsdb-values**. Separate multiple tags with semicolons (;).
connector.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 default value is **8**.
connector.tsdb-link-address	Yes	OpenTSDB address for connecting to the cluster where the data to be inserted belongs.

Precautions

- If your OpenTSDB runs in an MRS cluster, ensure that:
  a. The IP address and port number of OpenTSDB must be obtained from **tsd.network.bind** and **tsd.network.port** in the OpenTSDB service configuration.
  b. If **tsd.https.enabled** is set to **true**, the value format of **connector.tsdb-link-address** in the SQL statement is **https://***ip:port*. If **tsd.https.enabled** is set to **false**, the value of **connector.tsdb-link-address** can be in the format of **http://***ip:port* or *ip:port*.
  c. When establishing an enhanced datasource connection, you need to add the mapping between MRS cluster hosts and IP addresses in **/etc/hosts** to the Host Information parameter.
- 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.

- If dynamic columns are supported, the format must be ${columnName}, where **columnName** indicates a field name.

Example

```
create table sink1(
  attr1 bigint,
  attr2 int,
  attr3 int
) with (
  'connector.type' = 'opentsdb',
  'connector.region' = '',
  'connector.tsdb-metrics' = '',
  'connector.tsdb-timestamps' = '${attr1}',
  'connector.tsdb-values' = '${attr2};10',
  'connector.tsdb-tags' = 'key1:value1,key2:value2;key3:value3',
  'connector.tsdb-link-address' = ''
);
```

3.3.2.12 User-defined Result Table

Function

Write your Java code to insert the processed data into a specified database supported by your cloud service.

Prerequisites

Implement the custom sink class :

The custom sink class is inherited from Flink open-source class **RichSinkFunction**. The data type is **Tuple2<Boolean, Row**>.

For example, define the **MySink** class by **public class MySink extends RichSinkFunction< Tuple2<Boolean, Row>>{}**, and implement the **open**, **invoke**, and **close** functions. A code example is as follows:

```
public class MySink extends RichSinkFunction<Tuple2<Boolean, Row>> {
    // Initialize the object.
    @Override
    public void open(Configuration parameters) throws Exception {}

    @Override
    // Implement the data processing logic.
    /* The in parameter contains two values. The first value is of the Boolean type. The value true indicates the insert or update operation, and the value false indicates the delete operation. If the interconnected sink does not support the delete operation, the deletion will not be executed. The second value indicates the data to be operated.*/
    public void invoke(Tuple2<Boolean, Row> in, Context context) throws Exception {}

    @Override
    public void close() throws Exception {}
}
```

Content of the dependent pom configuration file is as follows:

```
<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>
```

Pack the implemented class and compile it in a JAR file, and upload it using the UDF Jar parameter on the editing page of your Flink OpenSource SQL job.

Syntax

```
create table userDefinedSink (
  attr_name attr_type
  (',' attr_name attr_type)*
)
with (
  'connector.type' = 'user-defined',
  'connector.class-name' = ''
);
```

Parameters

Table 3-20 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. The value can only be a user-defined sink.
connector.class-name	Yes	Fully qualified class name of the sink class. For details about the implementation of the sink class, see **Prerequisites**.
connector.class-parameter	No	Parameter of the constructor of the sink class. Only one parameter of the string type is supported.

Precautions

connector.class-name must be a fully qualified class name.

Example

```
create table userDefinedSink (
  attr1 int,
  attr2 int
)
with (
  'connector.type' = 'user-defined',
  'connector.class-name' = 'xx.xx.MySink'
);
```

3.3.2.13 Print Result Table

Function

The print connector exports your data output to the **error** file or the **out** file of TaskManager. It is mainly used for code debugging and output viewing.

Syntax

```
create table printSink (
  attr_name attr_type (',' attr_name attr_type) * (',' PRIMARY KEY (attr_name,...) NOT ENFORCED)
) with (
  'connector' = 'print',
  'print-identifier' = '',
  'standard-error' = ''
);
```

Parameters

Table 3-21 Parameter description

Parameter	Mandatory	Description
connector	Yes	The value is fixed to **print**.
print-identifier	No	Message that identifies print and is prefixed to the output of the value.
standard-error	No	The value can be only **true** or **false**. The default value is **false**. ● If the value is **true**, data is output to the error file of the TaskManager. ● If the value is **false**, data is output to the **out** file of the TaskManager.

Example

Read data from Kafka and export the data to the **out** file of TaskManager. You can view the output in the exported file.

```
create table kafkaSource(
  attr0 string,
  attr1 boolean,
  attr3 decimal(38, 18),
  attr4 TINYINT,
  attr5 smallint,
  attr6 int,
  attr7 bigint,
  attr8 float,
  attr9 double,
  attr10 date,
  attr11 time,
  attr12 timestamp(3)
) with (
  'connector.type' = 'kafka',
```

```
  'connector.version' = '0.11',
  'connector.topic' = 'test_json',
  'connector.properties.bootstrap.servers' = 'xx.xx.xx.xx:9092',
  'connector.properties.group.id' = 'test_print',
  'connector.startup-mode' = 'latest-offset',
  'format.type' = 'csv'
);

create table printTable(
  attr0 string,
  attr1 boolean,
  attr3 decimal(38,18),
  attr4 TINYINT,
  attr5 smallint,
  attr6 int,
  attr7 bigint,
  attr8 float,
  attr9 double,
  attr10 date,
  attr11 time,
  attr12 timestamp(3),
  attr13 array<string>,
  attr14 row<attr15 float, attr16 timestamp(3)>,
  attr17 map<int, bigint>
) with (
  "connector" = "print"
);

insert into
  printTable
select
  attr0,
  attr1,
  attr3,
  attr4,
  attr5,
  attr6,
  attr7,
  attr8,
  attr9,
  attr10,
  attr11,
  attr12,
  array [cast(attr0 as string), cast(attr0 as string)],
  row(
    cast(attr8 as float),
    cast(attr12 as timestamp(3))
  ),
  map [cast(attr6 as int), cast(attr7 as bigint)]
from
  kafkaSource;
```

3.3.2.14 File System Result Table

Function

You can create a file system result table to export data to a file system such as HDFS or OBS. After the data is generated, a non-DLI 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 partition tables. It is applicable to scenarios such as data dumping, big data analysis, data backup, and active, deep, or cold archiving.

Syntax

```
create table filesystemSink (
  attr_name attr_type (',' attr_name attr_type) *
) with (
  'connector.type' = 'filesystem',
  'connector.file-path' = '',
  'format.type' = ''
);
```

Important Notes

- If the data output directory in the table creation syntax is OBS, the directory must be a parallel file system and cannot be an OBS bucket.
- When using a file system table, you must enable checkpointing to ensure job consistency.
- When **format.type** is **parquet**, the supported data type is string, boolean, tinyint, smallint, int, bigint, float, double, map<string, string>, timestamp(3), and time.
- To avoid data loss or data coverage, you need to enable automatic 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.
- When using HDFS, you need to bind the data source and enter the host information.
- When using HDFS, you need to configure information about the node where the active NameNode locates.

Parameter

Table 3-22 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	The value is fixed to **filesystem**.

Parameter	Mandatory	Description
connector.file-path	Yes	Data output directory. The format is *schema://file.path*. **NOTE** Currently, Schema supports only OBS and HDFS. ● If **schema** is set to **obs**, data is stored to OBS. **Note that OBS directory must be a parallel file system and must not be an OBS bucket.** For example, **obs://***bucketName/fileName* indicates that data is exported to the **fileName** directory in the **bucketName** bucket. ● If **schema** is set to **hdfs**, data is exported to HDFS. Example: **hdfs://node-master1sYAx:9820/user/car_infos**, where **node-master1sYAx:9820** is the name of the node where the NameNode locates.
format.type	Yes	Output data encoding format. Only **parquet** and **csv** 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.
format.field-delimiter	No	Delimiter 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 (**,**).
connector.ak	No	Access key for accessing OBS This parameter is mandatory when data is written to OBS.
connector.sk	No	Secret key for accessing OBS This parameter is mandatory when data is written to OBS.
connector.partitioned-by	No	Partitioning field. Use commas (,) to separate multiple fields.

Example

Read data from Kafka and write the data in Parquet format to the **fileName** directory in the **bucketName** bucket.

```
create table kafkaSource(
  attr0 string,
  attr1 boolean,
  attr2 TINYINT,
  attr3 smallint,
  attr4 int,
```

```
  attr5 bigint,
  attr6 float,
  attr7 double,
  attr8 timestamp(3),
  attr9 time
) with (
  'connector.type' = 'kafka',
  'connector.version' = '0.11',
  'connector.topic' = 'test_json',
  'connector.properties.bootstrap.servers' = 'xx.xx.xx.xx:9092',
  'connector.properties.group.id' = 'test_filesystem',
  'connector.startup-mode' = 'latest-offset',
  'format.type' = 'csv'
);

create table filesystemSink(
  attr0 string,
  attr1 boolean,
  attr2 TINYINT,
  attr3 smallint,
  attr4 int,
  attr5 bigint,
  attr6 float,
  attr7 double,
  attr8 map < string,  string >,
  attr9 timestamp(3),
  attr10 time
) with (
  "connector.type" = "filesystem",
  "connector.file-path" = "obs://bucketName/fileName",
  "format.type" = "parquet",
  "connector.ak" = "xxxx",
  "connector.sk" = "xxxxxx"
);

insert into
  filesystemSink
select
  attr0,
  attr1,
  attr2,
  attr3,
  attr4,
  attr5,
  attr6,
  attr7,
  map [attr0,attr0],
  attr8,
  attr9
from
  kafkaSource;
```

3.3.3 Creating a Dimension Table

3.3.3.1 JDBC Dimension Table

Create a JDBC dimension table to connect to the source stream.

Prerequisites

- You have created a JDBC instance for your account.

Syntax

```
CREATE TABLE  table_id (
  attr_name attr_type
```

```
  (',' attr_name attr_type)*
)
  WITH (
  'connector.type' = 'jdbc',
  'connector.url' = '',
  'connector.table' = '',
  'connector.username' = '',
  'connector.password' = ''
);
```

Parameters

Table 3-23 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Data source type. Set this parameter to **jdbc**.
connector.url	Yes	Database URL
connector.table	Yes	Name of the table where the data to be read from the database is located
connector.driver	No	Driver required for connecting to the database If you do not set this parameter, the automatically extracted URL will be used.
connector.username	No	Database authentication user name. This parameter must be configured in pair with **connector.password**.
connector.password	No	Database authentication password. This parameter must be configured in pair with **connector.username**.
connector.read.partition.column	No	Name of the column used to partition the input This parameter is mandatory if **connector.read.partition.lower-bound**, **connector.read.partition.upper-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.lower-bound	No	Lower bound of values to be fetched for the first partition This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.upper-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.upper-bound	No	Upper bound of values to be fetched for the last partition This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.lower-bound**, and **connector.read.partition.num** are configured.

Parameter	Mandatory	Description
connector.read.partition.num	No	Number of partitions This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.upper-bound**, and **connector.read.partition.upper-bound** are configured.
connector.read.fetch-size	No	Number of rows fetched from the database each time. The default value is **0**, indicating the hint is ignored.
connector.lookup.cache.max-rows	No	Maximum number of cached rows in a dimension table. When the rows exceed this value, the data that is added first will be marked as expired. The value **-1** indicates that data cache disabled.
connector.lookup.cache.ttl	No	Time To Live (TTL) of dimension table cache. Caches exceeding the TTL will be deleted. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit).
connector.lookup.max-retries	No	Maximum number of attempts to obtain data from the dimension table. The default value is **3**.

Example

The RDS table is used to connect to the source stream.

```
CREATE TABLE car_infos (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT,
  proctime as PROCTIME()
)
  WITH (
  'connector.type' = 'dis',
  'connector.region' = 'ap-southeast-1',
  'connector.channel' = 'disInput',
  'format.type' = 'csv'
  );

CREATE TABLE  db_info (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
  'connector.type' = 'jdbc',
  'connector.url' = 'jdbc:mysql://xx.xx.xx.xx:3306/xx',
  'connector.table' = 'jdbc_table_name',
  'connector.driver' = 'com.mysql.jdbc.Driver',
```

```
  'connector.username' = 'xxx',
  'connector.password' = 'xxxxx'
);

CREATE TABLE audi_cheaper_than_30w (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
  'connector.type' = 'dis',
  'connector.region' = 'ap-southeast-1',
  'connector.channel' = 'disOutput',
  'connector.partition-key' = 'car_id,car_owner',
  'format.type' = 'csv'
  );

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 FOR SYSTEM_TIME AS OF a.proctime AS b on a.car_id = b.car_id;
```

3.3.3.2 GaussDB(DWS) Dimension Table

Create a GaussDB(DWS) dimension table to connect to the input stream.

Prerequisites

- You have created a GaussDB(DWS) instance for your account.

Syntax

```
create table dwsSource (
  attr_name attr_type
  (',' attr_name attr_type)*
)
with (
  'connector.type' = 'gaussdb',
  'connector.url' = '',
  'connector.table' = '',
  'connector.username' = '',
  'connector.password' = ''
);
```

Parameters

Table 3-24 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **gaussdb**.
connector.url	Yes	JDBC connection address. The format is **jdbc:postgresql://$**{ip}**:$**{port}**/$**{dbName}.
connector.table	Yes	Name of the table where the data to be read from the database is located

Parameter	Mandatory	Description
connector.driver	No	JDBC connection driver. The default value is **org.postgresql.Driver**.
connector.username	No	Database authentication user name. This parameter must be configured in pair with **connector.password**.
connector.password	No	Database authentication password. This parameter must be configured in pair with **connector.username**.
connector.read.partition.column	No	Name of the column used to partition the input This parameter is mandatory if **connector.read.partition.lower-bound**, **connector.read.partition.upper-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.lower-bound	No	Lower bound of values to be fetched for the first partition This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.upper-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.upper-bound	No	Upper bound of values to be fetched for the last partition This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.lower-bound**, and **connector.read.partition.num** are configured.
connector.read.partition.num	No	Number of partitions This parameter is mandatory if **connector.read.partition.column**, **connector.read.partition.upper-bound**, and **connector.read.partition.upper-bound** are configured.
connector.read.fetch-size	No	Number of rows fetched from the database each time. The default value is **0**, indicating the hint is ignored.
connector.lookup.cache.max-rows	No	Maximum number of cached rows in a dimension table. When the rows exceed this value, the data that is added first will be marked as expired. The value **-1** indicates that data cache disabled.

Parameter	Mandatory	Description
connector.lookup.cache.ttl	No	Time To Live (TTL) of dimension table cache. Caches exceeding the TTL will be deleted. The format is {length value}{time unit label}, for example, **123ms, 321s**. The supported time units include d, h, min, s, and ms (default unit).
connector.lookup.max-retries	No	Maximum number of attempts to obtain data from the dimension table. The default value is **3**.

Example

Use an RDS table to connect to the source stream.

```
CREATE TABLE car_infos (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT,
  proctime as PROCTIME()
)
  WITH (
  'connector.type' = 'dis',
  'connector.region' = 'ap-southeast-1',
  'connector.channel' = 'disInput',
  'format.type' = 'csv'
  );

CREATE TABLE  db_info (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
  'connector.type' = 'gaussdb',
  'connector.driver' = 'org.postgresql.Driver',
  'connector.url' = 'jdbc:gaussdb://xx.xx.xx.xx:8000/xx',
  'connector.table' = 'car_info',
  'connector.username' = 'xx',
  'connector.password' = 'xx',
  'connector.lookup.cache.max-rows' = '10000',
  'connector.lookup.cache.ttl' = '24h'
);

CREATE TABLE audi_cheaper_than_30w (
  car_id STRING,
  car_owner STRING,
  car_brand STRING,
  car_price INT
)
  WITH (
  'connector.type' = 'dis',
  'connector.region' = 'ap-southeast-1',
  'connector.channel' = 'disOutput',
  'connector.partition-key' = 'car_id,car_owner',
  'format.type' = 'csv'
  );

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 FOR SYSTEM_TIME AS OF a.proctime AS b on a.car_id = b.car_id;
```

3.3.3.3 HBase Dimension Table

Function

Create a Hbase dimension table to connect to the source stream.

Prerequisites

- An enhanced datasource connection has been created for DLI to connect to HBase, so that jobs can run on the dedicated queue of DLI and you can set the security group rules as required.
  - You have set up an enhanced datasource connection. For details, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.
  - For details about how to configure security group rules, see **Security Group Overview** in the *Virtual Private Cloud User Guide*.
- **If MRS HBase is used, IP addresses of all hosts in the MRS cluster have been added to host information of the enhanced datasource connection.**

  For details, see **Modifying the Host Information** in the Data Lake Insight User Guide.

Syntax

```
create table hbaseSource (
  attr_name attr_type
  (',' attr_name attr_type)*
 )
with (
  'connector.type' = 'hbase',
  'connector.version' = '1.4.3',
  'connector.table-name' = '',
  'connector.zookeeper.quorum' = ''
);
```

Parameters

Table 3-25 Parameter description

Parameter	Mandatory	Description
connector.type	Yes	Connector type. Set this parameter to **hbase**.
connector.version	Yes	The value must be **1.4.3**.
connector. table-name	Yes	Table name in HBase
connector.zookeeper.quorum	Yes	ZooKeeper address

Parameter	Mandatory	Description
connector.zookeeper.znode.parent	No	Root directory for ZooKeeper. The default value is **/hbase**.

Example

```
create table hbaseSource(
  id string,
  i Row<score string>
 ) with (
   'connector.type' = 'hbase',
   'connector.version' = '1.4.3',
   'connector.table-name' = 'user',
   'connector.zookeeper.quorum' = 'xxxx:2181'
 );
create table source1(
  id string,
  name string,
  geneder string,
  age int,
  address string,
  proctime as PROCTIME()
) with (
  "connector.type" = "dis",
  "connector.region" = "ap-southeast-1",
  "connector.channel" = "read",
  "connector.ak" = "xxxxxx",
  "connector.sk" = "xxxxxx",
  "format.type" = 'csv'
);

 create table hbaseSink(
  rowkey string,
  i Row<name string, geneder string, age int, address string>,
  j ROW<score string>
 ) with (
   'connector.type' = 'hbase',
   'connector.version' = '1.4.3',
   'connector.table-name' = 'score',
   'connector.write.buffer-flush.max-rows' = '1',
   'connector.zookeeper.quorum' = 'xxxx:2181'
 );
 insert into hbaseSink select d.id, ROW(name, geneder,age,address), ROW(score) from source1 as d join
hbaseSource for system_time as of d.proctime as h on d.id = h.id;
```

3.4 Data Manipulation Language (DML)

3.4.1 SELECT

SELECT

Syntax

```
SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
```



```
 [ GROUP BY { groupItem [, groupItem ]* } ]
 [ HAVING booleanExpression ]
```

Description

This clause is used to select data from a table.

ALL indicates that all results are returned.

DISTINCT indicates that the duplicated results are removed.

Precautions

- The to-be-queried table 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 clause 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 clause 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 clause 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

GroupBy generates update results in the stream processing table.

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 clause 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 string and the number of records.

```
insert into temp SELECT substring(name,6),count(name) FROM student
  GROUP BY substring(name,6);
```

Grouping sets, Rollup, Cube

Function

- The GROUP BY GROUPING SETS generates a result set equivalent to that generated by multiple simple GROUP BY UNION ALL statements. Using GROUPING SETS is more efficient.
- The ROLLUP and CUBE generate multiple groups based on certain rules and then collect statistics by group.
- The result set generated by CUBE contains all the combinations of values in the selected columns.
- The result set generated by ROLLUP contains the combinations of a certain layer structure in the selected columns.

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY groupingItem]
```

Description

Values of **groupingItem** can be **Grouping sets(columnName [, columnName]*)**, **Rollup(columnName [, columnName]*)**, and **Cube(columnName [, columnName]*)**.

Precautions

None

Example

Return the results generated based on **user** and **product**.

```
INSERT INTO temp SELECT SUM(amount)
FROM Orders
GROUP BY GROUPING SETS ((user), (product));
```

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;
```

3.4.2 Set Operations

UNION/UNION ALL/INTERSECT/EXCEPT

Syntax

```
query UNION [ ALL ] | Intersect | Except query
```

Description

- UNION is used to return the union set of multiple query results.

- INTERSECT is used to return the intersection of multiple query results.
- EXCEPT is used to return the difference 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 duplicate records returned by UNION are removed. The duplicate 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;
```

IN

Syntax

```
SELECT [ ALL | DISTINCT ]   { * | projectItem [, projectItem ]* }
  FROM tableExpression
  WHERE column_name IN (value (, value)* ) | query
```

Description

The IN operator allows multiple values to be specified in the WHERE clause. It returns true if the expression exists in the given table subquery.

Precautions

The subquery table must consist of a single column, and the data type of the column must be the same as that of the expression.

Example

Return **user** and **amount** information of the products in **NewProducts** of the **Orders** table.

```
insert into temp SELECT user, amount
FROM Orders
WHERE product IN (
    SELECT product FROM NewProducts
);
```

3.4.3 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

CAUTION

- For SQL queries in stream processing tables, the **time_attr** parameter of the grouping window function must reference a valid time attribute, which needs to specify the processing time or event time of the row.
- For batch SQL queries, the **time_attr** parameter of the grouping window function must be a TIMESTAMP type attribute.

Table 3-26 Array functions

Grouping Window Function	Description
TUMBLE(time_attr, interval)	Defines a tumbling time window. A tumbling time window assigns rows to non-overlapping, continuous windows with a fixed duration (interval). For example, a tumbling window of 5 minutes groups rows in 5 minutes intervals. Tumbling windows can be defined on event-time (stream + batch) or processing-time (stream).
HOP(time_attr, interval, interval)	Defines a hopping time window (called sliding window in the Table API). A hopping time window has a fixed duration (second interval parameter) and hops by a specified hop interval (first interval parameter). If the hop interval is smaller than the window size, hopping windows are overlapping. Thus, rows can be assigned to multiple windows. For example, a hopping window of 15 minutes size and 5 minute hop interval assigns each row to 3 different windows of 15 minute size, which are evaluated in an interval of 5 minutes. Hopping windows can be defined on event-time (stream + batch) or processing-time (stream).
SESSION(time_attr, interval)	Defines a session time window. Session time windows do not have a fixed duration but their bounds are defined by a time interval of inactivity, i.e., a session window is closed if no event appears for a defined gap period. For example a session window with a 30 minute gap starts when a row is observed after 30 minutes inactivity (otherwise the row would be added to an existing window) and is closed if no row is added within 30 minutes. Session windows can work on event-time (stream + batch) or processing-time (stream).

- Window auxiliary functions

  You can use the following helper functions to select the start and end timestamps, as well as the time attribute, for grouping windows.

> ⚠ CAUTION
>
> When calling helper functions, it is important to use the same parameters as those used in the **GROUP BY** clause for grouping window functions.

Table 3-27 Window auxiliary functions

Auxiliary Function	Description
TUMBLE_START(time_attr, interval) HOP_START(time_attr, interval, interval) SESSION_START(time_attr, interval)	Returns the timestamp of the inclusive lower bound of the corresponding tumbling, hopping, or session window.
TUMBLE_END(time_attr, interval) HOP_END(time_attr, interval, interval) SESSION_END(time_attr, interval)	Returns the timestamp of the **exclusive** upper bound of the corresponding tumbling, hopping, or session window. Note: The exclusive upper bound timestamp **cannot** be used as a rowtime attribute in subsequent time-based operations, such as interval joins and group window or over window aggregations.
TUMBLE_ROWTIME(time_attr, interval) HOP_ROWTIME(time_attr, interval, interval) SESSION_ROWTIME(time_attr, interval)	Returns the timestamp of the inclusive upper bound of the corresponding tumbling, hopping, or session window. The resulting attribute is a rowtime attribute that can be used in subsequent time-based operations such as interval joins and group window or over window aggregations.
TUMBLE_PROCTIME(time_attr, interval) HOP_PROCTIME(time_attr, interval, interval) SESSION_PROCTIME(time_attr, interval)	Returns a proctime attribute that can be used in subsequent time-based operations such as interval joins and group window or over window aggregations.

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;
```

TUMBLE WINDOW Extension

Function

The extension functions of the DLI tumbling window are as follows:

- Periodical tumbling windows for lower latency

  Before the tumbling window ends, the window can be periodically triggered based on the configured frequency. The compute result from the start to the current time is output, which does not affect the final output. The latest result can be viewed in each period before the window ends.

- Custom latency for higher data accuracy

  You can set a latency for the end of the window. The output of the window is updated according to the configured latency each time a piece of late data reaches.

Precautions

If you use **insert** to write results into the sink, the sink must support the upsert mode.

Syntax

```
TUMBLE(time_attr, window_interval, period_interval, lateness_interval)
```

Example

If the current **time_attr** attribute column is **testtime** and the window interval is 10 seconds, the statement is as follows:

```
TUMBLE(testtime, INTERVAL '10' SECOND, INTERVAL '10' SECOND, INTERVAL '10' SECOND)
```

Description

Table 3-28 Parameter description

Parameter	Description	Format
time_attr	Event time or processing time attribute column	-
window_interval	Duration of the window	● Format 1: **INTERVAL** '10' **SECOND** The window interval is 10 seconds. You can change the value as needed. ● Format 2: **INTERVAL** '10' **MINUTE** The window interval is 10 minutes. You can change the value as needed. ● Format 3: **INTERVAL** '10' **DAY** The window interval is 10 days. You can change the value as needed.
period_interval	Frequency of periodic triggering within the window range. That is, before the window ends, the output result is updated at an interval specified by **period_interval** from the time when the window starts. If this parameter is not set, the periodic triggering policy is not used by default.	
lateness_interval	Time to postpone the end of the window. The system continues to collect the data that reaches the window within **lateness_interval** after the window ends. The output is updated for each data that reaches the window within **lateness_interval**. **NOTE** If the time window is for processing time, **lateness_interval** does not take effect.	

NOTE

Values of **period_interval** and **lateness_interval** cannot be negative numbers.

- If **period_interval** is set to **0**, periodic triggering is disabled for the window.
- If **lateness_interval** is set to **0**, the latency after the window ends is disabled.
- If neither of the two parameters is set, both periodic triggering and latency are disabled and only the regular tumbling window functions are available .
- If only the latency function needs to be used, set period_interval **INTERVAL '0' SECOND**.

OVER WINDOW

The difference between Over Window and Group Window is that one record is generated from one row in Over Window.

Syntax

```
SELECT agg1(attr1) OVER (
  [PARTITION BY partition_name]
  ORDER BY proctime|rowtime
  ROWS
 BETWEEN (UNBOUNDED|rowCOUNT) PRECEDING AND CURRENT ROW FROM TABLENAME
```



```
SELECT agg1(attr1) OVER (
  [PARTITION BY partition_name]
  ORDER BY proctime|rowtime
  RANGE
  BETWEEN (UNBOUNDED|timeInterval) PRECEDING AND CURRENT ROW FROM TABLENAME
```

Description

Table 3-29 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

- All aggregates must be defined in the same window, that is, in the same partition, sort, and range.
- Currently, only windows from PRECEDING (unbounded or bounded) to CURRENT ROW are supported. The range described by FOLLOWING is not supported.
- ORDER BY must be specified for a single time attribute.

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;
```


3.4.4 JOIN

Equi-join

Syntax

```
FROM tableExpression INNER | LEFT | RIGHT | FULL JOIN tableExpression
  ON value11 = value21 [ AND value12 = value22]
```

Precautions

- Currently, only equi-joins are supported, for example, joins that have at least one conjunctive condition with an equality predicate. Arbitrary cross or theta joins are not supported.
- Tables are joined in the order in which they are specified in the FROM clause. Make sure to specify tables in an order that does not yield a cross join (Cartesian product), which are not supported and would cause a query to fail.
- For streaming queries the required state to compute the query result might grow infinitely depending on the type of aggregation and the number of distinct grouping keys. Provide a query configuration with valid retention interval to prevent excessive state size.

Example

```
SELECT *
FROM Orders INNER JOIN Product ON Orders.productId = Product.id;

SELECT *
FROM Orders LEFT JOIN Product ON Orders.productId = Product.id;

SELECT *
FROM Orders RIGHT JOIN Product ON Orders.productId = Product.id;

SELECT *
FROM Orders FULL OUTER JOIN Product ON Orders.productId = Product.id;
```

Time-Windowed Join

Function

Each piece of data in a stream is joined with data in different time zones in another stream.

Syntax

```
from t1 JOIN t2 ON t1.key = t2.key AND TIMEBOUND_EXPRESSIO
```

Description

TIMEBOUND_EXPRESSION can be in either of the following formats:

- L.time between LowerBound(R.time) and UpperBound(R.time)
- R.time between LowerBound(L.time) and UpperBound(L.time)
- Comparison expression with the time attributes (L.time/R.time)

Precautions

A time window join requires at least one equi join predicate and a join condition that limits the time of both streams.

For example, use two range predicates (<, <=, >=, or >), a BETWEEN predicate, or an equal predicate that compares the same type of time attributes (such as processing time and event time) in two input tables.

For example, the following predicate is a valid window join condition:

- ltime = rtime
- ltime >= rtime AND ltime < rtime + INTERVAL '10' MINUTE
- ltime BETWEEN rtime - INTERVAL '10' SECOND AND rtime + INTERVAL '5' SECOND

Example

Join all orders shipped within 4 hours with their associated shipments.

```
SELECT *
FROM Orders o, Shipments s
WHERE o.id = s.orderId AND
      o.ordertime BETWEEN s.shiptime - INTERVAL '4' HOUR AND s.shiptime;
```

Array Expansion

Precautions

This clause is used to return a new row for each element in the given array. Unnesting WITH ORDINALITY is not yet supported.

Example

```
SELECT users, tag
FROM Orders CROSS JOIN UNNEST(tags) AS t (tag);
```

User-Defined Table Functions

Function

This clause is used to join a table with the results of a table function. ach row of the left (outer) table is joined with all rows produced by the corresponding call of the table function.

Precautions

A left outer join against a lateral table requires a TRUE literal in the ON clause.

Example

The row of the left (outer) table is dropped, if its table function call returns an empty result.

```
SELECT users, tag
FROM Orders, LATERAL TABLE(unnest_udtf(tags)) t AS tag;
```

If a table function call returns an empty result, the corresponding outer row is preserved, and the result padded with null values.

```
SELECT users, tag
FROM Orders LEFT JOIN LATERAL TABLE(unnest_udtf(tags)) t AS tag ON TRUE;
```

Temporal Table Function Join

Function

Precautions

Currently only inner join and left outer join with temporal tables are supported.

Example

Assuming Rates is a temporal table function, the join can be expressed in SQL as follows:

```
SELECT
  o_amount, r_rate
FROM
  Orders,
  LATERAL TABLE (Rates(o_proctime))
WHERE
  r_currency = o_currency;
```

Join Temporal Tables

Function

This clause is used to join the Temporal table.

Syntax

```
SELECT column-names
FROM table1  [AS <alias1>]
[LEFT] JOIN table2 FOR SYSTEM_TIME AS OF table1.proctime [AS <alias2>]
ON table1.column-name1 = table2.key-name1
```

Description

- **table1.proctime** indicates the processing time attribute (computed column) of **table1**.
- **FOR SYSTEM_TIME AS OF table1.proctime** indicates that when the records in the left table are joined with the dimension table on the right, only the snapshot data is used for matching the current processing time dimension table.

Precautions

Only inner and left joins are supported for temporal tables with processing time attributes.

Example

LatestRates is a temporal table that is materialized with the latest rate.

```
SELECT
  o.amout, o.currency, r.rate, o.amount * r.rate
FROM
  Orders AS o
  JOIN LatestRates FOR SYSTEM_TIME AS OF o.proctime AS r
  ON r.currency = o.currency;
```

3.4.5 OrderBy & Limit

OrderBy

Function

This clause is used to sort data in ascending order on a time attribute.

Precautions

Currently, only sorting by time attribute is supported.

Example

Sort data in ascending order on the time attribute.

```
SELECT *
FROM Orders
ORDER BY orderTime;
```

Limit

Function

This clause is used to constrain the number of rows returned.

Precautions

This clause is used in conjunction with ORDER BY to ensure that the results are deterministic.

Example

```
SELECT *
FROM Orders
ORDER BY orderTime
LIMIT 3;
```

3.4.6 Top-N

Function

Top-N queries ask for the N smallest or largest values ordered by columns. Both smallest and largest values sets are considered Top-N queries. Top-N queries are useful in cases where the need is to display only the N bottom-most or the N top-most records from batch/streaming table on a condition.

Syntax

```
SELECT [column_list]
FROM (
   SELECT [column_list],
     ROW_NUMBER() OVER ([PARTITION BY col1[, col2...]]
       ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum
   FROM table_name)
WHERE rownum <= N [AND conditions]
```

Description

- ROW_NUMBER(): Allocate a unique and consecutive number to each line starting from the first line in the current partition. Currently, we only support ROW_NUMBER as the over window function. In the future, we will support RANK() and DENSE_RANK().
- PARTITION BY col1[, col2...]: Specifies the partition columns. Each partition will have a Top-N result.
- ORDER BY col1 [asc|desc][, col2 [asc|desc]...]: Specifies the ordering columns. The ordering directions can be different on different columns.

- WHERE rownum <= N: The rownum <= N is required for Flink to recognize this query is a Top-N query. The N represents the N smallest or largest records will be retained.
- [AND conditions]: It is free to add other conditions in the where clause, but the other conditions can only be combined with rownum <= N using AND conjunction.

Important Notes

- The TopN query is Result Updating.
- Flink SQL will sort the input data stream according to the order key,
- so if the top N records have been changed, the changed ones will be sent as retraction/update records to downstream.
- If the top N records need to be stored in external storage, the result table should have the same unique key with the Top-N query.

Example

This is an example to get the top five products per category that have the maximum sales in realtime.

```
SELECT *
  FROM (
     SELECT *,
         ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) as row_num
     FROM ShopSales)
  WHERE row_num <= 5;
```

3.4.7 Deduplication

Function

Deduplication removes rows that duplicate over a set of columns, keeping only the first one or the last one.

Syntax

```
SELECT [column_list]
FROM (
   SELECT [column_list],
     ROW_NUMBER() OVER ([PARTITION BY col1[, col2...]]
       ORDER BY time_attr [asc|desc]) AS rownum
   FROM table_name)
WHERE rownum = 1
```

Description

- ROW_NUMBER(): Assigns a unique, sequential number to each row, starting with one.
- PARTITION BY col1[, col2...]: Specifies the partition columns, for example, the deduplicate key.
- ORDER BY time_attr [asc|desc]: Specifies the ordering column, it must be a time attribute. Currently Flink supports proctime only. Ordering by ASC means to keep the first row, ordering by DESC means to keep the last row.

- WHERE rownum = 1: The rownum = 1 is required for Flink to recognize this query is deduplication.

Precautions

None

Example

The following examples show how to remove duplicate rows on **order_id**. The proctime is an event time attribute.

```
SELECT order_id, user, product, number
  FROM (
     SELECT *,
         ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY proctime ASC) as row_num
     FROM Orders)
  WHERE row_num = 1;
```

3.5 Functions

3.5.1 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.

POM Dependency

```
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-table-common</artifactId>
        <version>1.10.0</version>
        <scope>provided</scope>
</dependency>
```

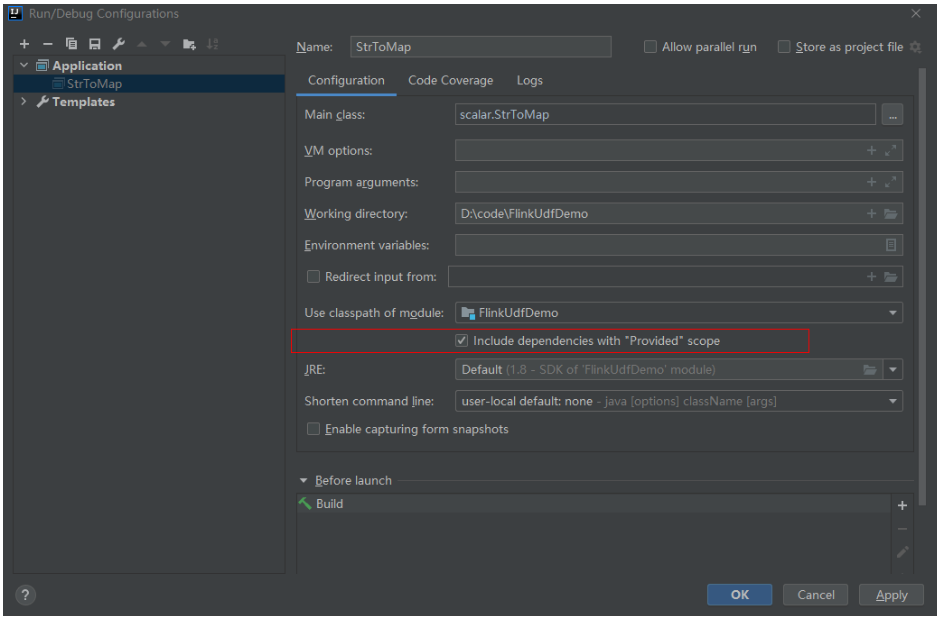
Important Notes

- 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. Encapsulate the implemented UDFs into a JAR package and upload the package to OBS.
2. In the navigation pane of the DLI management console, choose **Data Management** > **Package Management**. On the displayed page, click **Create** and use the JAR package uploaded to OBS to create a package.
3. In the left navigation, choose **Job Management** and click **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. Click the **Running Parameters** tab of your job, select the UDF JAR and click **Save**.
5. Add the following statement to the SQL statements to use the functions:

```
CREATE FUNCTION udf_test AS 'com.huaweicompany.udf.UdfScalarFunction';
```

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.huaweicompany.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.huaweicompany.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.huaweicompany.udf.UdfAggFunction';
INSERT INTO sink_stream SELECT udaf_test(attr2) FROM source_stream GROUP BY attr1;
```

3.5.2 Built-In Functions

3.5.2.1 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 3-30 lists all relational operators supported by Flink SQL.

Table 3-30 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.

Operator	Returned Data Type	Description
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.
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".

Operator	Returned Data Type	Description
A NOT BETWEEN B [ASYMMETRIC | SYMMETRIC]AND C	BOOLEAN	If A is less than B or greater than C, **TRUE** is returned. ● ASYMMETRIC: indicates that B and C are location-related. For example, "A NOT BETWEEN ASYMMETRIC B AND C" is equivalent to "A NOT BETWEEN B AND C". ● SYMMETRIC: indicates that B and C are not location-related. For example, "A NOT BETWEEN SYMMETRIC B AND C" is equivalent to "(A NOT BETWEEN B AND C) OR (A NOT BETWEEN C AND B)".
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.
EXISTS (sub-query)	BOOLEAN	If sub-query returns at least one row, **TRUE** is returned.
value IN (sub-query)	BOOLEAN	If value is equal to a row returned by subquery, **TRUE** is returned.
value NOT IN (sub-query)	BOOLEAN	If value is not equal to a row returned by subquery, **TRUE** is returned.

Precautions

- 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 3-31 lists the calculation rules. A and B indicate logical expressions.

Table 3-31 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.

Precautions

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 3-32** lists arithmetic operators supported by Flink SQL.

Table 3-32 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 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.

Operator	Result Type	Description
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).
LOG10(A)	All numeric types	Returns the base 10 logarithms of A.
LOG2(A)	All numeric types	Returns the base 2 logarithm of A.
LOG(B) LOG(A, B)	All numeric types	When called with one argument, returns the natural logarithm of B. When called with two arguments, this function returns the logarithm of B to the base A. B must be greater than 0 and A must be greater than 1.
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.

Operator	Result Type	Description
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.
ATAN2(A, B)	All numeric types	Returns the arc tangent of a coordinate (A, B).
COSH(A)	All numeric types	Returns the hyperbolic cosine of A. Return value type is DOUBLE.
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.

Operator	Result Type	Description
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	Returns a number rounded to **d** decimal places for A. For example: round(21.263,2) = 21.26.
PI	All numeric types	Returns the value of **pi**.
E()	All numeric types	Returns the value of **e**.
RAND()	All numeric types	Returns a pseudorandom double value in the range [0.0, 1.0)
RAND(A)	All numeric types	Returns a pseudorandom double value in the range [0.0, 1.0) with an initial seed A. Two RAND functions will return identical sequences of numbers if they have the same initial seed.
RAND_INTEGER(A)	All numeric types	Returns a pseudorandom double value in the range [0.0, A)
RAND_INTEGER(A, B)	All numeric types	Returns a pseudorandom double value in the range [0.0, B) with an initial seed A.
UUID()	All numeric types	Returns a UUID string.
BIN(A)	All numeric types	Returns a string representation of integer A in binary format. Returns NULL if A is NULL.

Operator	Result Type	Description
HEX(A) HEX(B)	All numeric types	Returns a string representation of an integer A value or a string B in hex format. Returns NULL if the A or B is NULL.
TRUNCATE(A, d)	All numeric types	Returns a number of truncated to **d** decimal places. Returns NULL if A or **d** is NULL. Example: truncate (42.345, 2) = 42.340 truncate(42.345) = 42.000
PI()	All numeric types	Returns the value of **pi**.

Precautions

Data of the string type is not allowed in arithmetic operations.

3.5.2.2 String Functions

Table 3-33 String functions

SQL Function	Return Type	Description
string1 || string2	STRING	Returns the concatenation of string1 and string2.
CHAR_LENGTH(string) CHARACTER_LENGTH(string)	INT	Returns the number of characters in the string.
UPPER(string)	STRING	Returns the string in uppercase.
LOWER(string)	STRING	Returns the string in lowercase.
POSITION(string1 IN string2)	INT	Returns the position (start from **1**) of the first occurrence of string1 in string2; returns **0** if string1 cannot be found in string2.

SQL Function	Return Type	Description
TRIM([BOTH | LEADING | TRAILING] string1 FROM string2)	STRING	Returns a string that removes leading and/or trailing characters string2 from string1.
LTRIM(string)	STRING	Returns a string that removes the left whitespaces from the specified string. For example, **LTRIM(' This is a test String.')** returns **"This is a test String."**.
RTRIM(string)	STRING	Returns a string that removes the right whitespaces from the specified string. For example, **RTRIM('This is a test String. ')** returns **"This is a test String."**.
REPEAT(string, integer)	STRING	Returns a string that repeats the base string integer times. For example, **REPEAT('This is a test String.', 2)** returns **"This is a test String.This is a test String."**.
REGEXP_REPLACE(string1, string2, string3)	STRING	Returns a string from string1 with all the substrings that match a regular expression string2 consecutively being replaced with string3. For example, **REGEXP_REPLACE('foobar', 'oo|ar', '')** returns **"fb"**. **REGEXP_REPLACE('ab\ab', '\\', 'e')** returns **"abeab"**.
OVERLAY(string1 PLACING string2 FROM integer1 [FOR integer2])	STRING	Returns a string that replaces integer2 characters of STRING1 with STRING2 from position integer1. The default value of integer2 is the length of string2. For example, **OVERLAY('This is an old string' PLACING ' new' FROM 10 FOR 5)** returns **"This is a new string"**.
SUBSTRING(string FROM integer1 [FOR integer2])	STRING	Returns a substring of the specified string starting from position integer1 with length integer2 (to the end by default). If integer2 is not configured, the substring from integer1 to the end is returned by default.

SQL Function	Return Type	Description
REPLACE(string1, string2, string3)	STRING	Returns a new string which replaces all the occurrences of string2 with string3 (non-overlapping) from string1. For example, **REPLACE('hello world', 'world', 'flink')** returns **"hello flink"**; **REPLACE('ababab', 'abab', 'z')** returns **"zab"**. **REPLACE('ab\\ab', '\\', 'e')** returns **"abeab"**.
REGEXP_EXTRACT(string1, string2[, integer])	STRING	Returns a string from string1 which extracted with a specified regular expression string2 and a regex match group index integer. Returns NULL, if the parameter is NULL or the regular expression is invalid. For example, **REGEXP_EXTRACT('foothebar', 'foo(.*?)(bar)', 2)"** returns **"bar"**.
INITCAP(string)	STRING	Returns a new form of STRING with the first character of each word converted to uppercase and the rest characters to lowercase.
CONCAT(string1, string2,...)	STRING	Returns a string that concatenates string1, string2, …. For example, **CONCAT('AA', 'BB', 'CC')** returns **"AABBCC"**.
CONCAT_WS(string1, string2, string3,...)	STRING	Returns a string that concatenates string2, string3, … with a separator string1. The separator is added between the strings to be concatenated. Returns NULL if string1 is NULL. If other arguments are NULL, this function automatically skips NULL arguments. For example, **CONCAT_WS('~', 'AA', NULL, 'BB', '', 'CC')** returns **"AA~BB~~CC"**.
LPAD(string1, integer, string2)	STRING	Returns a new string from string1 left-padded with string2 to a length of integer characters. If any argument is NULL, NULL is returned. If integer is negative, NULL is returned. If the length of string1 is shorter than integer, returns string1 shortened to integer characters. For example, **LPAD(Symbol,4,Symbol)** returns **"Symbol hi"**. **LPAD('hi',1,'??')** returns **"h"**.

SQL Function	Return Type	Description
RPAD(string1, integer, string2)	STRING	Returns a new string from string1 right-padded with string2 to a length of integer characters. If any argument is NULL, NULL is returned. If integer is negative, NULL is returned. If the length of string1 is shorter than integer, returns string1 shortened to integer characters. For example, **RPAD('hi',4,'??')** returns **"hi??"**. **RPAD('hi',1,'??')** returns **"h"**.
FROM_BASE64(string)	STRING	Returns the base64-decoded result from string. Returns NULL if string is NULL. For example, **FROM_BASE64('aGVsbG8gd29ybGQ=')** returns **"hello world"**.
TO_BASE64(string)	STRING	Returns the base64-encoded result from string; f string is NULL. Returns NULL if string is NULL. For example, **TO_BASE64(hello world)** returns **"aGVsbG8gd29ybGQ="**.
ASCII(string)	INT	Returns the numeric value of the first character of string. Returns NULL if string is NULL. For example, **ascii('abc')** returns **97.** **ascii(CAST(NULL AS VARCHAR))** returns **NULL**.
CHR(integer)	STRING	Returns the ASCII character having the binary equivalent to integer. If integer is larger than 255, we will get the modulus of integer divided by 255 first, and returns CHR of the modulus. Returns NULL if integer is NULL. **chr(97)** returns **a**. **chr(353)** Return **a**.
DECODE(binary, string)	STRING	Decodes the first argument into a String using the provided character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If either argument is NULL, the result will also be NULL.

SQL Function	Return Type	Description
ENCODE(strinh1, string2)	STRING	Encodes the string1 into a BINARY using the provided string2 character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If either argument is NULL, the result will also be NULL.
INSTR(string1, string2)	INT	Returns the position of the first occurrence of string2 in string1. Returns NULL if any argument is NULL.
LEFT(string, integer)	STRING	Returns the leftmost integer characters from the string. Returns EMPTY String if integer is negative. Returns NULL if any argument is NULL.
RIGHT(string, integer)	STRING	Returns the rightmost integer characters from the string. Returns EMPTY String if integer is negative. Returns NULL if any argument is NULL.
LOCATE(string1, string2[, integer])	INT	Returns the position of the first occurrence of string1 in string2 after position integer. Returns 0 if not found. The value of **integer** defaults to **0**. Returns NULL if any argument is NULL.
PARSE_URL(string1, string2[, string3])	STRING	Returns the specified part from the URL. Valid values for string2 include 'HOST', 'PATH', 'QUERY', 'REF', 'PROTOCOL', 'AUTHORITY', 'FILE', and 'USERINFO'. Returns NULL if any argument is NULL. If string2 is QUERY, the key in QUERY can be specified as string3. Example: The **parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST')** returns **'facebook.com'**. **parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1') returns 'v1'**.

SQL Function	Return Type	Description
REGEXP(string1, string2)	BOOLEAN	Returns TRUE if any (possibly empty) substring of string1 matches the regular expression string2, otherwise FALSE. If the information is found, TRUE is returned. string1 indicates the specified string, and string2 indicates the regular expression. Returns NULL if any argument is NULL.
REVERSE(string)	STRING	Returns the reversed string. Returns NULL if string is NULL.
SPLIT_INDEX(string1, string2, integer1)	STRING	Splits string1 by the delimiter string2, returns the integer1-th (zero-based) string of the split strings. Returns NULL if integer is negative. Returns NULL if any argument is NULL.
STR_TO_MAP(string1[, string2, string3]])	MAP	Returns a map after splitting the string1 into key/value pairs using delimiters. The default value of string2 is **','**. The default value of string3 is **'='**.
SUBSTR(string[, integer1[, integer2]])	STRING	Returns a substring of string starting from position integer1 with length integer2. If integer2 is not specified, the string is truncated to the end.
JSON_VAL(STRING json_string, STRING json_path)	STRING	Returns the value of the specified **json_path** from the **json_string**. For details about how to use the functions, see **JSON_VAL Function**. **NOTE** The following rules are listed in descending order of priority. 1. The two arguments **json_string** and **json_path** cannot be **NULL**. 2. The value of **json_string** must be a valid JSON string. Otherwise, the function returns **NULL**. 3. If **json_string** is an empty string, the function returns an empty string. 4. If **json_path** is an empty string or the path does not exist, the function returns **NULL**.

JSON_VAL Function

- Syntax

```
STRING JSON_VAL(STRING json_string, STRING json_path)
```

Table 3-34 Parameter description

Parameter	Type	Description
json_string	STRING	JSON object to be parsed
json_path	STRING	Path expression for parsing the JSON string For the supported expressions, see **Table 3-35**.

Table 3-35 Expressions supported

Expression	Description
$	Root node in the path
[]	Access array elements
*	Array wildcard
.	Access child elements

- Example
  a. Test input data.
     Test the data source kafka. The message content is as follows:
```
"{name:James,age:24,gender:male,grade:{math:95,science:[80,85],english:100}}"
"{name:James,age:24,gender:male,grade:{math:95,science:[80,85],english:100}]"
```
  b. Use JSON_VAL in SQL statements.
```
create table kafkaSource(
  message STRING
)
with (
  'connector.type' = 'kafka',
  'connector.version' = '0.11',
  'connector.topic' = 'topic-swq',
  'connector.properties.bootstrap.servers' =
'xxx.xxx.xxx.xxx:9092,yyy.yyy.yyy:9092,zzz.zzz.zzz.zzz:9092',
  'connector.startup-mode' = 'earliest-offset',
  'format.field-delimiter' = '|',
  'format.type' = 'csv'
);

create table kafkaSink(
  message1 STRING,
  message2 STRING,
  message3 STRING,
  message4 STRING,
  message5 STRING,
  message6 STRING
)
with (
  'connector.type' = 'kafka',
  'connector.version' = '0.11',
  'connector.topic' = 'topic-swq-out',
  'connector.properties.bootstrap.servers' =
'xxx.xxx.xxx.xxx:9092,yyy.yyy.yyy:9092,zzz.zzz.zzz.zzz:9092',
  'format.type' = 'json'
);
```

```
INSERT INTO kafkaSink
SELECT
JSON_VAL(message,""),
JSON_VAL(message,"$.name"),
JSON_VAL(message,"$.grade.science"),
JSON_VAL(message,"$.grade.science[*]"),
JSON_VAL(message,"$.grade.science[1]"),
JSON_VAL(message,"$.grade.dddd")
FROM kafkaSource;
```

c. View output.

```
{"message1":null,"message2":"swq","message3":"[80,85]","message4":"[80,85]","message5":"85"
,"message6":null}
{"message1":null,"message2":null,"message3":null,"message4":null,"message5":null,"message6":
null}
```

3.5.2.3 Temporal Functions

Table 3-36 lists the temporal functions supported by Flink OpenSource SQL.

Function Description

Table 3-36 Temporal functions

Function	Return Type	Description
DATE string	DATE	Parse the date string (**yyyy-MM-dd**) to a SQL date.
TIME string	TIME	Parse the time string (**HH:mm:ss[.fff]**) to a SQL time.
TIMESTAMP string	TIMESTAMP	Convert the time string into a timestamp. The time string format is **yyyy-MM-dd HH:mm:ss[.fff]**.

Function	Return Type	Description
INTERVAL string range	INTERVAL	Parse an interval string in the following two forms: • **yyyy-MM** for SQL intervals of months. An interval range might be **YEAR** or **YEAR TO MONTH**. • **dd hh:mm:ss.fff** for SQL intervals of milliseconds. An interval range might be **DAY**, **MINUTE**, **DAY TO HOUR**, or **DAY TO SECOND**. Example: **INTERVAL '10 00:00:00.004' DAY TO second** indicates that the interval is 10 days and 4 milliseconds. **INTERVAL '10' DAY**: indicates that the interval is 10 days. **INTERVAL '2-10' YEAR TO MONTH** indicates that the interval is two years and ten 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 local time zone.
LOCALTIMESTAMP	TIMESTAMP	Return the SQL timestamp of the local time zone.
EXTRACT(timeintervalunit FROM temporal)	BIGINT	Extract part of the time point or interval. Return the part in the int type. For example, extract the date **2006-06-05** and return **5**. **EXTRACT(DAY FROM DATE '2006-06-05')** returns **5**.
YEAR(date)	BIGINT	Return the year from a SQL date. For example, **YEAR(DATE'1994-09-27')** returns **1994**.
QUARTER(date)	BIGINT	Return the quarter of a year from a SQL date.
MONTH(date)	BIGINT	Return the month of a year from a SQL date. For example, **MONTH(DATE '1994-09-27')** returns **9**.

Function	Return Type	Description
WEEK(date)	BIGINT	Return the week of a year from a SQL date. For example, **WEEK(DATE'1994-09-27')** returns **39**.
DAYOFYEAR(date)	BIGINT	Return the day of a year from a SQL date. For example, **DAYOFYEAR(DATE '1994-09-27')** is **270**.
DAYOFMONTH(date)	BIGINT	Return the day of a month from a SQL date. For example, **DAYOFMONTH(DATE'1994-09-27')** returns **27**.
DAYOFWEEK(date)	BIGINT	Return the day of a week from a SQL date. Sunday is set to **1**. For example, **DAYOFWEEK(DATE'1994-09-27')** returns **3**.
HOUR(timestamp)	BIGINT	Return the hour of a day (an integer between 0 and 23) from a SQL timestamp. For example, **HOUR(TIMESTAMP '1994-09-27 13:14:15')** returns **13**.
MINUTE(timestamp)	BIGINT	Return the minute of an hour (an integer between 0 and 59) from a SQL timestamp. For example, **MINUTE(TIMESTAMP '1994-09-27 13:14:15')** returns **14**.
SECOND(timestamp)	BIGINT	Returns the second of a minute (an integer between 0 and 59) from a SQL timestamp. For example, **SECOND(TIMESTAMP '1994-09-27 13:14:15')** returns **15**.
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)**.
CEIL(timepoint TO timeintervalunit)	TIME	Round a time point up to the given unit. For example, **CEIL(TIME '12:44:31' TO MINUTE)** returns **12:45:00**.

Function	Return Type	Description
(timepoint1, temporal1) OVERLAPS (timepoint2, temporal2)	BOOLEAN	Return **TRUE** if two time intervals overlap. Example: **(TIME '2:55:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR)** returns **TRUE**. **(TIME '9:00:00', TIME '10:00:00') OVERLAPS (TIME '10:15:00', INTERVAL '3' HOUR)** returns **FALSE**.
DATE_FORMAT(timestamp, string)	STRING	Convert a timestamp to a value of string in the format specified by the date format string.
TIMESTAMPADD(timeintervalunit, interval, timepoint)	TIMESTAMP/ DATE/ TIME	Return the date and time added to **timepoint** based on the result of **interval** and **timeintervalunit**. For example, **TIMESTAMPADD(WEEK, 1, DATE '2003-01-02')** returns **2003-01-09**.
TIMESTAMPDIFF(timepointunit, timepoint1, timepoint2)	INT	Return the (signed) number of timepointunit between **timepoint1** and **timepoint2**. The unit for the interval is given by the first argument, which should be one of the following values: **SECOND**, **MINUTE**, **HOUR**, **DAY**, **MONTH**, and **YEAR**. For example, **TIMESTAMPDIFF(DAY, TIMESTAMP '2003-01-02 10:00:00', TIMESTAMP '2003-01-03 10:00:00')** returns **1**.
CONVERT_TZ(string1, string2, string3)	TIMESTAMP	Convert a datetime **string1** from time zone **string2** to time zone **string3**. For example, **CONVERT_TZ('1970-01-01 00:00:00', 'UTC', 'Country A/City A')** returns **'1969-12-31 16:00:00'**.
FROM_UNIXTIME(numeric[, string])	STRING	Return a representation of the numeric argument as a value in string format. The default string format is YYYY-MM-DD hh:mm:ss. For example, **FROM_UNIXTIME(44)** returns **1970-01-01 09:00:44**.
UNIX_TIMESTAMP()	BIGINT	Get current Unix timestamp in seconds.

Function	Return Type	Description
UNIX_TIMESTAMP(string1[, string2])	BIGINT	Convert date time string **string1** in format **string2** to Unix timestamp (in seconds), using the specified timezone in table config. The default format of **string2** is yyyy-MM-dd HH:mm:ss.
TO_DATE(string1[, string2])	DATE	Convert a date string **string1** with format **string2** to a date. The default format of **string2** is yyyy-MM-dd.
TO_TIMESTAMP(string1[, string2])	TIMESTAMP	Convert date time string **string1** with format **string2** to a timestamp. The default format of **string2** is yyyy-MM-dd HH:mm:ss.

DATE

- **Function**

  Returns a date parsed from string in form of **yyyy-MM-dd**.

- **Description**

  ```
  DATE DATE string
  ```

- **Input parameters**

Parameter	Type	Description
string	STRING	String in the SQL date format. Note that the string must be in the **yyyy-MM-dd** format. Otherwise, an error will be reported.

- **Example**
  - Test statement

    ```
    SELECT
      DATE "2021-08-19" AS `result`
    FROM
      testtable;
    ```

  - Test result

result
2021-08-19

TIME

- **Function**

  Returns a SQL time parsed from string in form of **HH:mm:ss[.fff]**.

- **Description**

```
TIME TIME string
```

- **Input parameters**

Parameter	Type	Description
string	STRING	Time Note that the string must be in the format of **HH:mm:ss[.fff]**. Otherwise, an error will be reported.

- **Example**
  - Test statement

```
SELECT
    TIME "10:11:12" AS `result`,
        TIME "10:11:12.032" AS `result2`
FROM
    testtable;
```

  - Test result

result	result2
10:11:12	10:11:12.032

TIMESTAMP

- **Function**

  Converts the time string into timestamp. The time string format is **yyyy-MM-dd HH:mm:ss[.fff]**. The return value is of the **TIMESTAMP(3)** type.

- **Description**

```
TIMESTAMP(3) TIMESTAMP string
```

- **Input parameters**

Parameter	Type	Description
string	STRING	Time Note that the string must be in the format of **yyyy-MM-dd HH:mm:ss[.fff]**. Otherwise, an error will be reported.

- **Example**
  - Test statement

```
SELECT
    TIMESTAMP "1997-04-25 13:14:15" AS `result`,
```

```
        TIMESTAMP "1997-04-25 13:14:15.032" AS `result2`
FROM
    testtable;
```

– Test result

result	result2
1997-04-25 13:14:15	1997-04-25 13:14:15.032

INTERVAL

- **Function**

  Parses an interval string.

- **Description**

```
INTERVAL INTERVAL string range
```

- **Input parameters**

Parameter	Type	Description
string	STRING	Timestamp string used together with the **range** parameter. The string is in either of the following two formats: • **yyyy-MM** for SQL intervals of months. An interval range might be **YEAR** or **YEAR TO MONTH** for intervals of months. • **dd hh:mm:ss.fff** for SQL intervals of milliseconds. An interval range might be **DAY**, **MINUTE**, **DAY TO HOUR**, or **DAY TO SECOND**.
range	INTERVAL	Interval range. This parameter is used together with the **string** parameter. Available values are as follows: **YEAR, YEAR To Month, DAY, MINUTE, DAY TO HOUR and DAY TO SECOND**.

- **Example**

  Test statement

```
-- The interval is 10 days and 4 milliseconds.
INTERVAL '10 00:00:00.004' DAY TO second
-- The interval is 10 days.
INTERVAL '10'
-- The interval is 2 years and 10 months.
INTERVAL '2-10' YEAR TO MONTH
```

CURRENT_DATE

- **Function**

  Returns the current SQL time (**yyyy-MM-dd**) in the local time zone. The return value is of the **DATE** type.

- **Description**

```
DATE CURRENT_DATE
```

- **Input parameters**

  N/A

- **Example**
  - Test statement

```
SELECT
   CURRENT_DATE AS `result`
FROM
   testtable;
```

  - Test result

result
2021-10-28

CURRENT_TIME

- **Function**

  Returns the current SQL time (**HH:mm:sss.fff**) in the local time zone. The return value is of the **TIME** type.

- **Description**

```
TIME CURRENT_TIME
```

- **Input parameters**

  N/A

- **Example**
  - Test statement

```
SELECT
   CURRENT_TIME AS `result`
FROM
   testtable;
```

  - Test result

result
08:29:19.289

CURRENT_TIMESTAMP

- **Function**

  Returns the current SQL timestamp in the local time zone. The return value is of the **TIMESTAMP(3)** type.

- **Description**

```
TIMESTAMP(3) CURRENT_TIMESTAMP
```

- **Input parameters**

  N/A

- **Example**
  - Test statement

```
SELECT
    CURRENT_TIMESTAMP AS `result`
FROM
    testtable;
```

– Test result

result
2021-10-28 08:33:51.606

LOCALTIME

- **Function**

  Returns the current SQL time in the local time zone. The return value is of the **TIME** type.

- **Description**

  ```
  TIME LOCALTIME
  ```

- **Input parameters**

  N/A

- **Example**
  – Test statement

  ```
  SELECT
      LOCALTIME AS `result`
  FROM
      testtable;
  ```

  – Test result

result
16:39:37.706

LOCALTIMESTAMP

- **Function**

  Returns the current SQL timestamp in the local time zone. The return value is of the **TIMESTAMP(3)** type.

- **Description**

  ```
  TIMESTAMP(3) LOCALTIMESTAMP
  ```

- **Input parameters**

  N/A

- **Example**
  – Test statement

  ```
  SELECT
      LOCALTIMESTAMP AS `result`
  FROM
      testtable;
  ```

  – Test result

result
2021-10-28 16:43:17.625

EXTRACT

- **Function**

  Returns a value extracted from the **timeintervalunit** part of temporal. The return value is of the **BIGINT** type.

- **Description**

  ```
  BIGINT EXTRACT(timeinteravlunit FROM temporal)
  ```

- **Input parameters**

Parameter	Type	Description
timeinteravlunit	TIMEUNIT	Time unit to be extracted from a time point or interval. The value can be **YEAR**, **QUARTER**, **MONTH**, **WEEK**, **DAY**, **DOY**, **HOUR**, **MINUTE**, **SECOND**.
temporal	DATE/TIME/TIMESTAMP/INTERVAL	Time point or interval.

> **CAUTION**
>
> Do not specify a time unit that is not of any time points or intervals. Otherwise, the job fails to be submitted.
>
> For example, an error message is displayed when the following statement is executed because **YEAR** cannot be extracted from **TIME**.

```
SELECT
   EXTRACT(YEAR FROM TIME '12:44:31' ) AS `result`
FROM
   testtable;
```

- **Example**
  - Test statement

```
SELECT
   EXTRACT(YEAR FROM DATE '1997-04-25' ) AS `result`,
      EXTRACT(MINUTE FROM TIME '12:44:31') AS `result2`,
      EXTRACT(SECOND FROM TIMESTAMP '1997-04-25 13:14:15') AS `result3`,
      EXTRACT(YEAR FROM INTERVAL '2-10' YEAR TO MONTH) AS `result4`,
FROM
   testtable;
```

  - Test result

result	result2	result3	result4
1997	44	15	2

YEAR

- **Function**

  Returns the year from a SQL date. The return value is of the **BIGINT** type.

- **Description**
```
BIGINT YEAR(date)
```
- **Input parameters**

Parameter	Type	Description
date	DATE	SQL date

- **Example**
  - Test statement
```
SELECT
   YEAR(DATE '1997-04-25' ) AS `result`
FROM
   testtable;
```
  - Test result

result
1997

QUARTER

- **Function**

  Returns the quarter of a year (an integer between 1 and 4) from a SQL date. The return value is of the **BIGINT** type.

- **Description**
```
BIGINT QUARTER(date)
```
- **Input parameters**

Parameter	Type	Description
date	DATE	SQL date

- **Example**
  - Test statement
```
SELECT
   QUARTER(DATE '1997-04-25' ) AS `result`
FROM
   testtable;
```
  - Test result

Result
2

MONTH

- **Function**

  Returns the month of a year (an integer between 1 and 12) from a SQL date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT MONTH(date)
```

- **Input parameters**

Parameter	Type	Description
date	DATE	SQL date

- **Example**
  - Test statement

```
SELECT
   MONTH(DATE '1997-04-25' ) AS `result`
FROM
   testtable;
```

  - Test result

result
4

WEEK

- **Function**

  Returns the week of a year from a SQL date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT WEEK(date)
```

- **Input parameters**

Parameter	Type	Description
date	DATE	SQL date

- **Example**
  - Test statement

```
SELECT
   WEEK(DATE '1997-04-25' ) AS `result`
FROM
   testtable;
```

  - Test result

result
17

DAYOFYEAR

- **Function**

  Returns the day of a year (an integer between 1 and 366) from SQL date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT DAYOFYEAR(date)
```

- **Input parameters**

Parameter	Type	Description
date	DATE	SQL date

- **Example**
  - Test statement

```
SELECT
    DAYOFYEAR(DATE '1997-04-25' ) AS `result`
FROM
    testtable;
```

  - Test result

result
115

DAYOFMONTH

- **Function**

  Returns the day of a month (an integer between 1 and 31) from a SQL date. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT DAYOFMONTH(date)
```

- **Input parameters**

Parameter	Type	Description
date	DATE	SQL date

- **Example**
  - Test statement

```
SELECT
    DAYOFMONTH(DATE '1997-04-25' ) AS `result`
FROM
    testtable;
```

  - Test result

Result
25

DAYOFWEEK

- **Function**

  Returns the day of a week (an integer between 1 and 7) from a SQL date. The return value is of the **BIGINT** type.

  NOTE

  Note that the start day of a week is Sunday.

- **Description**

```
BIGINT DAYOFWEEK(date)
```

- **Input parameters**

Parameter	Type	Description
date	DATE	SQL date

- **Example**
  - Test statement

```
SELECT
    DAYOFWEEK(DATE '1997-04-25') AS `result`
FROM
    testtable;
```

  - Test result

result
6

HOUR

- **Function**

  Returns the hour of a day (an integer between 0 and 23) from SQL timestamp. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT HOUR(timestamp)
```

- **Input parameters**

Parameter	Type	Description
timestamp	TIMESTAMP	SQL timestamp

- **Example**
  - Test statement

```
SELECT
    HOUR(TIMESTAMP '1997-04-25 10:11:12') AS `result`
FROM
    testtable;
```

  - Test result

result
10

MINUTE

- **Function**

  Returns the minute of an hour (an integer between 0 and 59) from a SQL timestamp. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT MINUTE(timestamp)
```

- **Input parameters**

Parameter	Type	Description
timestamp	TIMESTAMP	SQL timestamp

- **Example**
  – Test statement

```
SELECT
   MINUTE(TIMESTAMP '1997-04-25 10:11:12') AS `result`
FROM
   testtable;
```

  – Test result

result
11

SECOND

- **Function**

  Returns the second of an hour (an integer between 0 and 59) from a SQL timestamp. The return value is of the **BIGINT** type.

- **Description**

```
BIGINT SECOND(timestamp)
```

- **Input parameters**

Parameter	Type	Description
timestamp	TIMESTAMP	SQL timestamp

- **Example**
  – Test statement

```
SELECT
   SECOND(TIMESTAMP '1997-04-25 10:11:12') AS `result`
FROM
   testtable;
```

  – Test result

result
12

FLOOR

- **Function**

  Returns a value that rounds **timepoint** down to the time unit **timeintervalunit**.
- **Description**
  ```
  TIME/TIMESTAMP(3) FLOOR(timepoint TO timeintervalunit)
  ```
- **Input parameters**

Parameter	Type	Description
timepoint	TIMESTAMP /TIME	SQL time or SQL timestamp
timeintervalunit	TIMEUNIT	Time unit. The value can be **YEAR**, **QUARTER**, **MONTH**, **WEEK**, **DAY**, **DOY**, **HOUR**, **MINUTE**, or **SECOND**.

- **Example**
  - Test statement For details about the syntax of the userDefined result table, see **User-defined Result Table**.
    ```
    create table PrintSink (
      message TIME,
      message2 TIME,
      message3 TIMESTAMP(3)
    )
    with (
      'connector.type' = 'user-defined',
      'connector.class-name' = 'com.swqtest.flink.sink.PrintSink'--Replace the class with a user-defined class. For details, see the syntax description in the userDefined result table.
    );

    INSERT INTO
        PrintSink
    SELECT
        FLOOR(TIME '13:14:15' TO MINUTE) AS `result`
            FLOOR(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result2`,
            FLOOR(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result3`;
    ```
  - Test result

    The values of the fields in the PrintSink table are as follows:

Message	Message 2	Message 3
13:14	13:14	1997-04-25T13:14

CEIL

- **Function**

  Returns a value that rounds **timepoint** up to the time unit **timeintervalunit**.

- **Description**
  TIME/TIMESTAMP(3) **CEIL**(timepoint **TO** timeintervalunit)
- **Input parameters**

Parameter	Type	Description
timepoint	TIMESTAMP /TIME	SQL time or SQL timestamp
timeintervalunit	TIMEUNIT	Time unit. The value can be **YEAR**, **QUARTER**, **MONTH**, **WEEK**, **DAY**, **DOY**, **HOUR**, **MINUTE**, or **SECOND**.

- **Example**
  – Test statement For details about the syntax of the userDefined result table, see **User-defined Result Table**.

```
create table PrintSink (
  message TIME,
  message2 TIME,
  message3 TIMESTAMP(3)
)
with (
  'connector.type' = 'user-defined',
  'connector.class-name' = 'com.swqtest.flink.sink.PrintSink'--Replace the class with a user-defined class. For details, see the syntax description in the userDefined result table.
);

INSERT INTO
    PrintSink
SELECT
    CEIL(TIME '13:14:15' TO MINUTE) AS `result`
        CEIL(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result2`,
        CEIL(TIMESTAMP '1997-04-25 13:14:15' TO MINUTE) AS `result3`;
```

  – Test result

result	result2	result3
13:15	13:15	1997-04-25T13:15

OVERLAPS

- **Function**
  Returns **TRUE** if two time intervals overlap; returns **FALSE** otherwise.
- **Description**
  BOOLEAN (timepoint1, temporal1) **OVERLAPS** (timepoint2, temporal2)
- **Input parameters**

Parameter	Type	Description
timepoint1/ timepoint2	DATE/TIME/ TIMESTAMP	Time point
temporal1/ temporal2	DATE/TIME/ TIMESTAMP/ INTERVAL	Time point or interval

NOTE

- **(timepoint, temporal)** is a closed interval.
- The temporal can be of the **DATE**, **TIME**, **TIMESTAMP**, or **INTERVAL** type.
  - When th temporal is **DATE**, **TIME**, or **TIMESTAMP**, **(timepoint, temporal)** indicates an interval between **timepoint** and **temporal**. The temporal can be earlier than the value of **timepoint**, for example, **(DATE '1997-04-25', DATE '1997-04-23')**.
  - When the temporal is **INTERVAL**, **(timepoint, temporal)** indicates an interval between **timepoint** and **timepoint + temporal**.
- Ensure that **(timepoint1, temporal1)** and **(timepoint2, temporal2)** are intervals of the same data type.

- **Example**
  - Test statement

```
SELECT
    (TIME '2:55:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR) AS
`result`,
        (TIME '2:30:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:30:00', INTERVAL '2' HOUR) AS
`result2`,
    (TIME '2:30:00', INTERVAL '1' HOUR) OVERLAPS (TIME '3:31:00', INTERVAL '2' HOUR) AS
`result3`,
    (TIME '9:00:00', TIME '10:00:00') OVERLAPS (TIME '10:00:00', INTERVAL '3' HOUR) AS
`result4`,
    (TIMESTAMP '1997-04-25 12:00:00', TIMESTAMP '1997-04-25 12:20:00') OVERLAPS
(TIMESTAMP '1997-04-25 13:00:00', INTERVAL '2' HOUR) AS `result5`,
    (DATE '1997-04-23', INTERVAL '2' DAY) OVERLAPS (DATE '1997-04-25', INTERVAL '2' DAY)
AS `result6`,
    (DATE '1997-04-25', DATE '1997-04-23') OVERLAPS (DATE '1997-04-25', INTERVAL '2' DAY)
AS `result7`
FROM
    testtable;
```

  - Test result

result	result2	result3	result4	result5	result6	result7
true	true	false	true	false	true	true

DATE_FORMAT

- **Function**

  Converts a timestamp to a value of string in the format specified by the date format string.

- **Description**

```
STRING DATE_FORMAT(timestamp, dateformat)
```

- **Input parameters**

Parameter	Type	Description
timestamp	TIMESTAMP/ STRING	Time point
dateformat	STRING	String in the date format

- **Example**
  - Test statement
```
SELECT
    DATE_FORMAT(TIMESTAMP '1997-04-25 10:11:12', 'yyyy-MM-dd HH:mm:ss') AS `result`,
        DATE_FORMAT(TIMESTAMP '1997-04-25 10:11:12', 'yyyy-MM-dd') AS `result2`,
    DATE_FORMAT(TIMESTAMP '1997-04-25 10:11:12', 'yy/MM/dd HH:mm') AS `result3`,
        DATE_FORMAT('1997-04-25 10:11:12', 'yyyy-MM-dd') AS `result4`
FROM    testtable;
```
  - Test result

result	result2	result3	result4
1997-04-25 10:11:12	1997-04-25	97/04/25 10:11	1997-04-25

TIMESTAMPADD

- **Function**

  Returns the date and time by combining **interval** and **timeintervalunit** and adding the combination to **timepoint**.

  NOTE

  The return value of **TIMESTAMPADD** is the value of **timepoint**. An exception is that if the input **timepoint** is of the **TIMESTAMP** type, the return value can be inserted into a table field of the **DATE** type.

- **Description**
```
TIMESTAMP(3)/DATE/TIME TIMESTAMPADD(timeintervalunit, interval, timepoint)
```
- **Input parameters**

Parameter	Type	Description
timeintervalunit	TIMEUNIT	Time unit
interval	INT	Interval
timepoint	TIMESTAMP/DATE/TIME	Time point

- **Example**
  - Test statement
```
SELECT
    TIMESTAMPADD(WEEK, 1, DATE '1997-04-25') AS `result`,
        TIMESTAMPADD(QUARTER, 1, TIMESTAMP '1997-04-25 10:11:12') AS `result2`,
    TIMESTAMPADD(SECOND, 2, TIME '10:11:12') AS `result3`
FROM    testtable;
```
  - Test result

result	result2	result3
1997-05-02	● If this field is inserted into a table field of the **TIMESTAMP** type, **1997-07-25T10:11:12** is returned. ● If this field is inserted into a table field of the **DATE** type, **1997-07-25** is returned.	10:11:14

TIMESTAMPDIFF

- **Function**

  Returns the (signed) number of **timepointunit** between **timepoint1** and **timepoint2**. The unit for the interval is given by the first argument.

- **Description**

```
INT TIMESTAMPDIFF(timepointunit, timepoint1, timepoint2)
```

- **Input parameters**

Parameter	Type	Description
timepointunit	TIMEUNIT	Time unit. The value can be **SECOND**, **MINUTE**, **HOUR**, **DAY**, **MONTH** or **YEAR**.
timepoint1/ timepoint2	TIMESTAMP/ DATE	Time point

- **Example**
  - Test statement

```
SELECT
    TIMESTAMPDIFF(DAY, TIMESTAMP '1997-04-25 10:00:00', TIMESTAMP '1997-04-28 10:00:00')
AS `result`,
        TIMESTAMPDIFF(DAY, DATE '1997-04-25', DATE '1997-04-28') AS `result2`,
    TIMESTAMPDIFF(DAY, TIMESTAMP '1997-04-27 10:00:20', TIMESTAMP '1997-04-25 10:00:00')
AS `result3`
FROM     testtable;
```

  - Test result

result	result2	result3
3	3	-2

CONVERT_TZ

- **Function**

  Converts a datetime **string1** (with default ISO timestamp format **'yyyy-MM-dd HH:mm:ss'**) from time zone **string2** to time zone **string3**.

- **Description**

```
STRING CONVERT_TZ(string1, string2, string3)
```

- **Input parameters**

Parameter	Type	Description
string1	STRING	SQL timestamp. If the value does not meet the format requirements, **NULL** is returned.
string2	STRING	Time zone before conversion. The format of time zone should be either an abbreviation such as **PST**, a full name such as **Country A/City A**, or a custom ID such as **GMT-08:00**.
string3	STRING	Time zone after conversion. The format of time zone should be either an abbreviation such as **PST**, a full name such as **Country A/City A**, or a custom ID such as **GMT-08:00**.

- **Example**
  - Test statement

```
SELECT
   CONVERT_TZ(1970-01-01 00:00:00, UTC, Country A/City A) AS `result`,
      CONVERT_TZ(1997-04-25 10:00:00, UTC, GMT-08:00) AS `result2`
FROM    testtable;
```

  - Test result

result	result2
1969-12-31 16:00:00	1997-04-25 02:00:00

FROM_UNIXTIME

- **Function**

  Returns a representation of the **numeric** argument as a value in string format.

- **Description**

```
STRING FROM_UNIXTIME(numeric[, string])
```

- **Input parameters**

Parameter	Type	Description
numeric	BIGINT	An internal timestamp representing the number of seconds since 1970-01-01 00:00:00 UTC. The value can be generated by the **UNIX_TIMESTAMP()** function.

Parameter	Type	Description
string	STRING	Time. If this parameter is not specified, the default time format is **yyyy-MM-dd HH:mm:ss** format.

- **Example**
  - Test statement
```
SELECT
   FROM_UNIXTIME(44) AS `result`,
      FROM_UNIXTIME(44, 'yyyy:MM:dd') AS `result2`
FROM    testtable;
```
  - Test result

result	result2
1970-01-01 08:00:44	1970:01:01

UNIX_TIMESTAMP

- **Function**

  Gets current Unix timestamp in seconds. The return value is of the **BIGINT** type.
- **Description**
```
BIGINT UNIX_TIMESTAMP()
```
- **Input parameters**

  N/A
- **Example**
  - Test statement
```
SELECT
   UNIX_TIMESTAMP() AS `result`
FROM
   table;
```
  - Test result

result
1635401982

UNIX_TIMESTAMP(string1[, string2])

- **Function**

  Converts date time **string1** in format **string2** to Unix timestamp (in seconds). The return value is of the **BIGINT** type.
- **Description**
```
BIGINT UNIX_TIMESTAMP(string1[, string2])
```
- **Input parameters**

Parameter	Type	Description
string1	STRING	SQL timestamp string. An error is reported if the value does not comply with the **string2** format.
string2	STRING	Time. If this parameter is not specified, the default time format is **yyyy-MM-dd HH:mm:ss**.

- **Example**
  - Test statement

```
SELECT
    UNIX_TIMESTAMP('1997-04-25', 'yyyy-MM-dd') AS `result`,
        UNIX_TIMESTAMP('1997-04-25 00:00:10', 'yyyy-MM-dd HH:mm:ss') AS `result2`,
        UNIX_TIMESTAMP('1997-04-25 00:00:00') AS `result3`
FROM
    testtable;
```

  - Test result

result	result2	result3
861897600	861897610	861897600

TO_DATE

- **Function**

  Converts a date **string1** with format **string2** to a date.
- **Description**

```
DATE TO_DATE(string1[, string2])
```

- **Input parameters**

Parameter	Type	Description
string1	STRING	SQL timestamp string. If the value is not in the required format, an error is reported.
string2	STRING	Format. If this parameter is not specified, the default time format is **yyyy-MM-dd**.

- **Example**
  - Test statement

```
SELECT
    TO_DATE('1997-04-25') AS `result`,
        TO_DATE('1997:04:25', 'yyyy-MM-dd') AS `result2`,
        TO_DATE('1997-04-25 00:00:00', 'yyyy-MM-dd HH:mm:ss') AS `result3`
FROM
    testtable;
```

  - Test result

result	result2	result3
1997-04-25	1997-04-25	1997-04-25

TO_TIMESTAMP

- **Function**

  Converts date time **string1** with format **string2** to a timestamp.

- **Description**

```
TIMESTAMP TO_TIMESTAMP(string1[, string2])
```

- **Input parameters**

Parameter	Type	Description
string1	STRING	SQL timestamp string. If the value is not in the required format, **NULL** is returned.
string2	STRING	Date format. If this parameter is not specified, the default format is **yyyy-MM-dd HH:mm:ss**.

- **Example**
  - Test statement

```
SELECT
  TO_TIMESTAMP('1997-04-25', 'yyyy-MM-dd') AS `result`,
    TO_TIMESTAMP('1997-04-25 00:00:00') AS `result2`,
    TO_TIMESTAMP('1997-04-25 00:00:00', 'yyyy-MM-dd HH:mm:ss') AS `result3`
FROM
  testtable;
```

  - Test result

result	result2	result3
1997-04-25 00:00	1997-04-25 00:00	1997-04-25 00:00

3.5.2.4 Conditional Functions

Description

Table 3-37 Conditional functions

Function	Description
CASE value WHEN value1_1 [, value1_2]* THEN result1 [WHEN value2_1 [, value2_2]* THEN result2]* [ELSE resultZ] END	Returns **resultX** when the value is contained in (valueX_1, valueX_2, …). Only the first matched value is returned. When no value matches, returns **resultZ** if it is provided and returns **NULL** otherwise.
CASE WHEN condition1 THEN result1 [WHEN condition2 THEN result2]* [ELSE resultZ] END	Returns **resultX** when the first **conditionX** is met. Only the first matched value is returned. When no condition is met, returns **resultZ** if it is provided and returns **NULL** otherwise.
NULLIF(value1, value2)	Returns NULL if value1 is equal to value2; returns value1 otherwise. For example, **NullIF (5, 5)** returns **NULL**. **NULLIF(5, 0)** returns **5**.
COALESCE(value1, value2 [, value3]*)	Returns the first value (from left to right) that is not NULL from value1, value2, …. For example, **COALESCE(NULL, 5)** returns **5**.
IF(condition, true_value, false_value)	Returns the **true_value** if condition is met, otherwise **false_value**. For example, **IF(5 > 3, 5, 3)** returns **5**.
IS_ALPHA(string)	Returns **TRUE** if all characters in the string are letters, otherwise **FALSE**.
IS_DECIMAL(string)	Returns **TRUE** if string can be parsed to a valid numeric, otherwise **FALSE**.
IS_DIGIT(string)	Returns **TRUE** if all characters in the string are digits, otherwise **FALSE**.

3.5.2.5 Type Conversion Function

Syntax

```
CAST(value AS type)
```

Syntax Description

This function is used to forcibly convert types.

Precautions

If the input is **NULL**, **NULL** is returned.

Example

The following example converts the **amount** value to an integer.

```
insert into temp select cast(amount as INT) from source_stream;
```

Table 3-38 Examples of type conversion functions

Example	Description	Example

cast(v1 as string)	Converts **v1** to a string. The value of **v1** can be of the numeric type or of the timestamp, date, or time type.	Table T1:
		`| content (INT) |`
		`| ------------- |`
		`| 5 |`
		Statement:
		`SELECT`
		`  cast(content as varchar)`
		`FROM`
		`  T1;`
		Result:
		`"5"`

cast (v1 as int)	Converts **v1** to the **int** type. The value of **v1** can be a number or a character.	Table T1:
		`| content (STRING) |`
		`| ------------- |`
		`| "5" |`
		Statement:
		`SELECT`
		`  cast(content as int)`
		`FROM`
		`  T1;`
		Result:
		`5`

Example	Description	Example

cast(v1 as timestamp)	Converts **v1** to the **timestamp** type. The value of **v1** can be of the **string**, **date**, or **time** type.	Table T1:
		`| content (STRING) |`
		`| ------------- |`
		`| "2018-01-01 00:00:01" |`
		Statement:
		`SELECT`
		`  cast(content as timestamp)`
		`FROM`
		`  T1;`
		Result:
		`1514736001000`

cast(v1 as date)	Converts **v1** to the **date** type. The value of **v1** can be of the **string** or **timestamp** type.	Table T1:
		`| content (TIMESTAMP) |`
		`| ------------- |`
		`| 1514736001000 |`
		Statement:
		`SELECT`
		`  cast(content as date)`
		`FROM`
		`  T1;`
		Result:
		`"2018-01-01"`

NOTE

Flink jobs do not support the conversion of **bigint** to **timestamp** using CAST. You can convert it using **to_timestamp**.

Detailed Sample Code

```
/** source **/
CREATE
TABLE car_infos (cast_int_to_string int, cast_String_to_int string,
case_string_to_timestamp string, case_timestamp_to_date timestamp(3)) WITH (
  'connector.type' = 'dis',
  'connector.region' = 'xxxxx',
  'connector.channel' = 'dis-input',
  'format.type' = 'json'
);
/** sink **/
CREATE
TABLE cars_infos_out (cast_int_to_string string, cast_String_to_int
int, case_string_to_timestamp timestamp(3), case_timestamp_to_date date) WITH (
  'connector.type' = 'dis',
  'connector.region' = 'xxxxx',
  'connector.channel' = 'dis-output',
  'format.type' = 'json'
);
/** Statistics on static car information**/
INSERT
INTO
  cars_infos_out
SELECT
  cast(cast_int_to_string as string),
  cast(cast_String_to_int as int),
  cast(case_string_to_timestamp as timestamp),
  cast(case_timestamp_to_date as date)
FROM
  car_infos;
```

3.5.2.6 Collection Functions

Description

Table 3-39 Collection functions

Function	Description
CARDINALITY(array)	Returns the number of elements in array.
array '[' integer ']'	Returns the element at position INT in array. The index starts from 1.
ELEMENT(array)	Returns the sole element of array (whose cardinality should be one) Returns **NULL** if array is empty. Throws an exception if array has more than one element.
CARDINALITY(map)	Returns the number of entries in map.
map '[' key ']'	Returns the value specified by key value in map.

3.5.2.7 Value Construction Functions

Description

Table 3-40 Value construction functions

Function	Description
ROW(value1, [, value2]*) (value1, [, value2]*)	Returns a row created from a list of values (value1, value2,…).
ARRAY '[' value1 [, value2]* ']'	Returns an array created from a list of values (value1, value2, …).
MAP '[' key1, value1 [, key2, value2]* ']'	Returns a map created from a list of key-value pairs ((value1, value2), (value3, value4), …). The key-value pair is (key1, value1), (key2, value2).

3.5.2.8 Value Access Functions

Description

Table 3-41 Value access functions

Function	Description
tableName.compositeType.field	Returns the value of a field from a Flink composite type (e.g., Tuple, POJO) by name.
tableName.compositeType.*	Returns a flat representation of a Flink composite type (e.g., Tuple, POJO) that converts each of its direct subtype into a separate field.

3.5.2.9 Hash Functions

Description

Table 3-42 Hash functions

Function	Description
MD5(string)	Returns the MD5 hash as a string that contains 32 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA1(string)	Returns the SHA-1 hash as a string that contains 40 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA224(string)	Returns the SHA-224 hash as a string that contains 56 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA256(string)	Returns the SHA-256 hash as a string that contains 64 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA384(string)	Returns the SHA-384 hash as a string that contains 96 hexadecimal digits. Returns **NULL** if **string** is **NULL**.
SHA512(string)	Returns the SHA-512 hash as a string that contains 128 hexadecimal digits. Returns **NULL** if **string** is **NULL**.

Function	Description
SHA2(string, hashLength)	Returns the hash using the SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, or SHA-512). The first argument string is the string to be hashed and the second argument hashLength is the bit length of the result (224, 256, 384, or 512). Returns **NULL** if **string** or hashLength is **NULL**.

3.5.2.10 Aggregate Function

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 3-43** lists aggregate functions.

Table 3-43 Aggregate functions

Function	Return Data Type	Description
COUNT([ALL] expression | DISTINCT expression1 [, expression2]*)	BIGINT	Returns the number of input rows for which the expression is not NULL. Use DISTINCT for one unique instance of each value.
COUNT(*) COUNT(1)	BIGINT	Returns the number of input rows.
AVG([ALL | DISTINCT] expression)	DOUBLE	Returns the average (arithmetic mean) of expression across all input rows. Use DISTINCT for one unique instance of each value.
SUM([ALL | DISTINCT] expression)	DOUBLE	Returns the sum of expression across all input rows. Use DISTINCT for one unique instance of each value.
MAX([ALL | DISTINCT] expression)	DOUBLE	Returns the maximum value of expression across all input rows.
MIN([ALL | DISTINCT] expression)	DOUBLE	Returns the minimum value of expression across all input rows.
STDDEV_POP([ALL | DISTINCT] expression)	DOUBLE	Returns the population standard deviation of expression across all input rows.
STDDEV_SAMP([ALL | DISTINCT] expression)	DOUBLE	Returns the sample standard deviation of expression across all input rows.

Function	Return Data Type	Description
VAR_POP([ALL | DISTINCT] expression)	DOUBLE	Returns the population variance (square of the population standard deviation) of expression across all input rows.
VAR_SAMP([ALL | DISTINCT] expression)	DOUBLE	Returns the sample variance (square of the sample standard deviation) of expression across all input rows.
COLLECT([ALL | DISTINCT] expression)	MULTISET	Returns a multiset of expression across all input rows.
VARIANCE([ALL | DISTINCT] expression)	DOUBLE	Returns the sample variance (square of the sample standard deviation) of expression across all input rows.
FIRST_VALUE(expression)	Actual type	Returns the first value in an ordered set of values.
LAST_VALUE(expression)	Actual type	Returns the last value in an ordered set of values.

3.5.2.11 Table-Valued Functions

3.5.2.11.1 split_cursor

The **split_cursor** function 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 3-44 split_cursor function

Function	Return 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 table 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;
```

3.5.2.11.2 string_split

The **string_split** function splits a target string into substrings based on the specified separator and returns a substring list.

Description

```
string_split(target, separator)
```

Table 3-45 string_split parameters

Parameter	Type	Description
target	STRING	Target string to be processed **NOTE** • If **target** is **NULL**, an empty line is returned. • If **target** contains two or more consecutive separators, an empty substring is returned. • If **target** does not contain a specified separator, the original string passed to **target** is returned.
separator	VARCHAR	Delimiter. Currently, only single-character delimiters are supported.

Example

1. Prepare test input data.

Table 3-46 Source table disSource

target (STRING)	separator (VARCHAR)
test-flink	-
flink	-
one-two-ww-three	-

2. Write test SQL statements.

```
create table disSource(
  target STRING,
  separator  VARCHAR
) with (
  "connector.type" = "dis",
  "connector.region" = "xxx",
  "connector.channel" = "ygj-dis-in",
  "format.type" = 'csv'
```



```
);

create table disSink(
  target STRING,
  item STRING
) with (
  'connector.type' = 'dis',
  'connector.region' = 'xxx',
  'connector.channel' = 'ygj-dis-out',
  'format.type' = 'csv'
);

insert into
  disSink
select
  target,
  item
from
  disSource,
lateral table(string_split(target, separator)) as T(item);
```

3. Check test results.

Table 3-47 disSink result table

target (STRING)	item (STRING)
test-flink	test
test-flink	flink
flink	flink
one-two-ww-three	one
one-two-ww-three	two
one-two-ww-three	ww
one-two-ww-three	three