AI DataLake

Spark Syntax Reference

Issue	01
Date	2026-11-05



HUAWEI TECHNOLOGIES CO., LTD.

Copyright © Huawei Technologies Co., Ltd. 2026. All rights reserved.

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

Trademarks and Permissions


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

Notice

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

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

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: https://www.huawei.com

Email: support@huawei.com

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 Before You Start 1
2 Database-Related 2
2.1 Create Database 2
2.2 Drop Database 4
2.3 Describe Specific Database 5
2.4 Show All Databases 7
2.5 Alter Database Properties 9
3 Table Related 12
3.1 Table Type Description 12
3.2 Create Table 14
3.2.1 Create Managed Table Using DataSource Syntax 14
3.2.2 Create Managed Table Using Hive Syntax 17
3.2.3 Create External Table Using DataSource Syntax 20
3.2.4 Create External Table Using Hive Syntax 25
3.3 Drop Table 30
3.4 View Table 31
3.4.1 Show All Tables 31
3.4.2 Show Create Table 32
3.4.3 Show Table Properties 34
3.4.4 Show All Columns of Specified Table 36
3.4.5 Show All Partitions of Specified Table 37
3.4.6 Show Table Statistics 39
3.5 Alter Table 41
3.5.1 Add Columns 41
3.5.2 Modify Column Comment 42
3.6 Table Alias 44
3.7 Partition Related 44
3.7.1 Add Partition 44
3.7.2 Rename Partition 46
3.7.3 Drop Partition 48
3.7.4 Modify Table Partition Location 49
3.7.5 Update Table Partition Information 50

 Copyright © Huawei Technologies Co., Ltd.

4 Import/Export Data Syntax 52
4.1 Load Data 52
4.2 Insert Data 56
4.3 Reuse Subquery 59
4.4 Truncate Data 61
5 Export Query Results 62
6 View Syntax 64
6.1 Create View 64
6.2 Drop View 65
6.3 Alter View 65
6.4 Show Views 67
7 Materialized Views 70
7.1 Use Cases 70
7.2 Configuration Parameters 73
7.3 Regular Materialized Views 74
7.3.1 Syntax Reference 74
7.3.2 Permission Management 76
7.3.3 Automatic Rewriting 76
7.3.4 Metadata 77
7.3.5 Notes and Constraints 77
7.4 Intelligent Materialized Views 78
7.4.1 Syntax Reference 78
7.4.2 Permission Management 78
7.4.3 View Refresh 79
7.4.3.1 Refresh Strategies 79
7.4.3.2 Refresh Schedules 80
7.4.3.3 View Metadata 80
7.4.4 CTE Materialization 80
7.4.5 Notes and Constraints 81
8 View Plan 82
9 Data Permissions 83
9.1 Data Permission List 83
9.2 Create Role 86
9.3 Show Roles 86
9.4 Delete Role 87
9.5 Bind Role 87
9.6 Unbind Role 87
9.7 Grant Permissions 88
9.8 Revoke Permissions 89
9.9 Show Granted Permissions 90

 Copyright © Huawei Technologies Co., Ltd.

9.10 Show All Role and User Bindings....90
10 Data Types....92
10.1 Overview....92
10.2 Primitive Data Types....92
10.3 Complex Data Types....98
11 Iceberg Table Related Syntax....101
12 Function Syntax....102
12.1 SQL User-Defined Function....102
12.2 Create Function....104
12.3 Drop Function....109
12.4 Show Function Details....109
12.5 Show All Functions....110
12.6 DESCRIBE FUNCTION....111
13 Query Syntax....113
13.1 SELECT....113
13.2 Sorting....116
13.2.1 ORDER BY....116
13.2.2 SORT BY....118
13.2.3 CLUSTER BY....119
13.2.4 DISTRIBUTE BY....119
13.3 Grouping....120
13.3.1 GROUP BY by Column....120
13.3.2 GROUP BY Expression....121
13.3.3 Using HAVING in GROUP BY....122
13.3.4 ROLLUP....122
13.3.5 GROUPING SETS....124
13.4 JOINS....125
13.4.1 INNER JOIN....125
13.4.2 LEFT OUTER JOIN....126
13.4.3 RIGHT OUTER JOIN....127
13.4.4 FULL OUTER JOIN....128
13.4.5 IMPLICIT JOIN....129
13.4.6 CROSS JOIN (Cartesian Join)....130
13.4.7 LEFT SEMI JOIN....131
13.4.8 NON-EQUI JOIN....132
13.5 Clauses....133
13.5.1 FROM....133
13.5.2 OVER....134
13.5.3 WHERE....137
13.5.4 HAVING....138
13.5.5 Multi-Level Nested Subquery....139

 Copyright © Huawei Technologies Co., Ltd.

13.6 Alias 140
13.6.1 Column Alias 140
13.7 Set Operations 141
13.7.1 UNION 141
13.7.2 INTERSECT 142
13.7.3 EXCEPT 143
13.8 CASE...WHEN 144
13.9 Pipe Syntax (|>) 146
14 Exporting Query Results 150
15 Built-In Functions 152
15.1 Date Functions 152
15.1.1 add_months 152
15.1.2 current_date 153
15.1.3 current_timestamp 154
15.1.4 date_add 154
15.1.5 dateadd1 156
15.1.6 date_sub 157
15.1.7 date_format 159
15.1.8 datediff 160
15.1.9 datediff1 161
15.1.10 datepart1 163
15.1.11 datetrunc 164
15.1.12 day/dayofmonth 166
15.1.13 from_unixtime 167
15.1.14 from_utc_timestamp 168
15.1.15 getdate 169
15.1.16 hour 169
15.1.17 isdate 170
15.1.18 last_day 171
15.1.19 last_day1 172
15.1.20 minute 173
15.1.21 month 174
15.1.22 months_between 175
15.1.23 next_day 177
15.1.24 quarter 178
15.1.25 second 179
15.1.26 to_char1 180
15.1.27 to_date 182
15.1.28 to_date1 182
15.1.29 to_utc_timestamp 184
15.1.30 trunc 185
15.1.31 trunc_numeric 186

 Copyright © Huawei Technologies Co., Ltd.

15.1.32 unix_timestamp 187
15.1.33 weekday 188
15.1.34 weekofyear 189
15.1.35 year 190
15.2 String Functions 190
15.2.1 ascii 190
15.2.2 concat 191
15.2.3 concat_ws 192
15.2.4 char_matchcount 193
15.2.5 encode 194
15.2.6 find_in_set 195
15.2.7 get_json_object 196
15.2.8 initcap 198
15.2.9 instr 198
15.2.10 instr1 199
15.2.11 keyvalue 200
15.2.12 length 201
15.2.13 lengthb 202
15.2.14 levenshtein 203
15.2.15 locate 204
15.2.16 lower/lcase 205
15.2.17 lpad 205
15.2.18 ltrim 206
15.2.19 parse_url 208
15.2.20 regexp_count1 209
15.2.21 regexp_extract 210
15.2.22 regexp_instr1 211
15.2.23 regexp_replace 212
15.2.24 regexp_replace1 213
15.2.25 regexp_substr1 214
15.2.26 repeat 215
15.2.27 replace 216
15.2.28 reverse 216
15.2.29 rpad 217
15.2.30 rtrim 218
15.2.31 space 219
15.2.32 split_part1 220
15.2.33 substr/substring 221
15.2.34 substring_index 222
15.2.35 translate 223
15.2.36 trim 224
15.2.37 upper/ucase 225

 Copyright © Huawei Technologies Co., Ltd.

15.2.38 url_decode1 226
15.2.39 url_encode1 227
15.2.40 printf 227
15.3 Mathematical Functions 228
15.3.1 abs 228
15.3.2 acos 229
15.3.3 asin 230
15.3.4 atan 230
15.3.5 bin 231
15.3.6 bround 232
15.3.7 cbrt 233
15.3.8 ceil 234
15.3.9 conv 235
15.3.10 cos 236
15.3.11 cot1 237
15.3.12 degrees 238
15.3.13 e 239
15.3.14 exp 239
15.3.15 factorial 240
15.3.16 floor 241
15.3.17 greatest 242
15.3.18 hex 243
15.3.19 least 244
15.3.20 ln 245
15.3.21 log 246
15.3.22 log10 247
15.3.23 log2 247
15.3.24 negative 248
15.3.25 pi 249
15.3.26 pmod 249
15.3.27 positive 250
15.3.28 pow 251
15.3.29 radians 252
15.3.30 rand 253
15.3.31 round 253
15.3.32 shiftleft 254
15.3.33 shiftright 255
15.3.34 shiftrightunsigned 256
15.3.35 sign 257
15.3.36 sin 258
15.3.37 soundex 259
15.3.38 sqrt 260

 Copyright © Huawei Technologies Co., Ltd.

15.3.39 tan 261
15.4 Aggregate Functions 261
15.4.1 avg 261
15.4.2 corr 262
15.4.3 count 263
15.4.4 covar_pop 264
15.4.5 covar_samp 265
15.4.6 max 266
15.4.7 min 267
15.4.8 percentile 268
15.4.9 percentile_approx 269
15.4.10 stddev_pop 270
15.4.11 stddev_samp 271
15.4.12 sum 272
15.4.13 variance/var_pop 273
15.4.14 var_samp 273
15.4.15 median 274
15.5 Window Functions 275
15.5.1 cume_dist 275
15.5.2 first_value 277
15.5.3 last_value 279
15.5.4 lag 280
15.5.5 lead 283
15.5.6 percent_rank 285
15.5.7 rank 286
15.5.8 row_number 287
15.6 Other Functions 289
15.6.1 decode1 289
15.6.2 ordinal 290
15.6.3 trans_array 291
16 Identifiers 294
16.1 aggregate_func 296
16.2 alias 296
16.3 attr_expr 296
16.4 attr_expr_list 298
16.5 attrs_value_set_expr 298
16.6 boolean_expression 299
16.7 class_name 299
16.8 col 300
16.9 col_comment 300
16.10 col_name 300
16.11 col_name_list 300

 Copyright © Huawei Technologies Co., Ltd.

16.12 condition 301
16.13 condition_list 302
16.14 cte_name 303
16.15 data_type 303
16.16 db_comment 304
16.17 db_name 304
16.18 else_result_expression 304
16.19 file_format 304
16.20 file_path 305
16.21 function_name 305
16.22 groupby_expression 305
16.23 having_condition 306
16.24 hdfs_path 307
16.25 input_expression 307
16.26 input_format_classname 307
16.27 jar_path 308
16.28 join_condition 308
16.29 non_equi_join_condition 309
16.30 number 309
16.31 num_buckets 310
16.32 output_format_classname 310
16.33 partition_col_name 310
16.34 partition_col_value 310
16.35 partition_specs 311
16.36 property_name 311
16.37 property_value 311
16.38 regex_expression 311
16.39 result_expression 312
16.40 row_format 312
16.41 select_statement 313
16.42 separator 313
16.43 serde_name 313
16.44 sql_containing_cte_name 313
16.45 sub_query 313
16.46 table_comment 314
16.47 table_name 314
16.48 table_properties 314
16.49 table_reference 314
16.50 view_name 315
16.51 view_properties 315
16.52 when_expression 315
16.53 where_condition 315

 Copyright © Huawei Technologies Co., Ltd.

16.54 window_function ... 317
17 Operators ... 318
17.1 Relational Operators ... 318
17.2 Arithmetic Operators ... 320
17.3 Logical Operators ... 322
18 Datasets ... 323
18.1 Overview ... 323
18.2 Dataset Syntax ... 323
18.2.1 Creating a Dataset ... 323
18.2.2 Altering a Dataset ... 324
18.2.3 Updating Dataset Format ... 325
18.2.4 Setting or Updating a Dataset Comment ... 325
18.2.5 Soft Deleting a Dataset ... 325
18.2.6 Listing All Datasets ... 326
18.2.7 Displaying Dataset Details ... 326
18.2.8 Reading Files of a Specified Format in a Dataset ... 327
18.3 Dataset Permissions ... 327
18.3.1 Overview ... 327
18.3.2 Granting User Access ... 328
18.3.3 Revoking User Access ... 329
18.3.4 Showing All Permissions ... 329

 Copyright © Huawei Technologies Co., Ltd.

1 Before You Start

This document is the AI DataLake Spark SQL syntax reference, covering database management, table operations, data read/write, query syntax, built-in functions, identifiers, operators, and more. Before using this document for development, read the following notes and constraints.

Identifier Case Sensitivity

During development, you are advised to use case-insensitive formats where possible. You are advised to use lowercase letters consistently for database object names (such as table names, column names, function names, field names, etc.). Avoid inconsistent casing for the same object in different locations to prevent errors such as "object not found" caused by identifier case sensitivity.

Notes and Constraints

Currently, both standard format tables and Iceberg tables do not support executing the following SQL statements:

- SET
- USE
- USE DATABASE
- REFRESH RESOURCE
- CACHE TABLE
- UNCACHE TABLE
- CLEAR CACHE
- REFRESH TABLE
- RESET
- LIST JARS
- LIST FILES
- ADD JAR
- ADD FILE

 Copyright © Huawei Technologies Co., Ltd.

2 Database-Related

2.1 Create Database
2.2 Drop Database
2.3 Describe Specific Database
2.4 Show All Databases
2.5 Alter Database Properties

2.1 Create Database

Function

Creates a database.

Syntax

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

Keywords

- **IF NOT EXISTS**: Does not raise an error if the database to be created already exists; silently skips.
- **COMMENT**: A description of the database.
- **LOCATION**: Specifies the storage path of the database in the file system. Must be specified and in OBS format, e.g., **obs://bucket-name/db-path/**. An error is reported if **LOCATION** is not specified or does not meet the format requirements.
- **WITH DBPROPERTIES**: Properties of the database, with property names and values appearing in pairs.

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 2-1 Parameter descriptions

Parameter	Description
db_name	Database name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with a digit or underscore.
db_comment	Database description.
obs_path	Database storage path, must be in **obs://** format, pointing to a valid object storage path.
property_name	Database property name.
property_value	Database property value.

Precautions

- **DATABASE** and **SCHEMA** have no difference and can be used interchangeably. **DATABASE** is recommended.
- **default** is a built-in database. A database named **default** cannot be created.
- Use **DESCRIBE DATABASE EXTENDED** to view database property information.

Example 1: Create a Database

Create database **testdb**. If **testdb** already exists, no error is raised.

```
CREATE DATABASE IF NOT EXISTS testdb
  LOCATION 'obs://my-bucket/testdb/';
```

Example 2: Create a Database with Comment and Properties

Create database **customer_db** with comment and property information.

```
CREATE DATABASE IF NOT EXISTS customer_db
  COMMENT 'This is customer database'
  LOCATION 'obs://my-bucket/customer_db/'
  WITH DBPROPERTIES (ID=001, Name='John');
```

Example 3: Verify Database Properties

Use **DESCRIBE DATABASE EXTENDED** to view detailed database information, including properties.

```
DESCRIBE DATABASE EXTENDED customer_db;
```

Example result:

 Copyright © Huawei Technologies Co., Ltd.

Table 2-2 Example result

database_description_item	database_description_value
Database Name	customer_db
Description	This is customer database
Location	obs://my-bucket/customer_db/
Properties	((ID,001), (Name,John))

Example 4: Create a Database Using the SCHEMA Keyword

SCHEMA is equivalent to **DATABASE** and can be used interchangeably.

```
CREATE SCHEMA IF NOT EXISTS payroll_db
  LOCATION 'obs://my-bucket/payroll_db/';
```

2.2 Drop Database

Function

Drops a database and its associated file system directory. If the database does not exist, the system throws an exception.

Syntax

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

Keywords

IF EXISTS: Used when the database to be dropped does not exist, to prevent the system from reporting an error.

Precautions

- **DATABASE** and **SCHEMA** have no difference and can be used interchangeably. **DATABASE** is recommended.
- **RESTRICT**: If the database is not empty (has tables), the **DROP** operation reports an error and fails. **RESTRICT** is the default logic.
- **CASCADE**: Even if the database is not empty (has tables), **DROP** will cascade and delete all tables under it. Use this feature with caution.
- After dropping a database, its storage path on OBS is not automatically deleted and needs to be cleaned up manually.

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 2-3 Parameter description

Parameter	Description
db_name	Database name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with a digit or underscore.

Example 1: Drop an Empty Database

Drop database **inventory_db**. If the database is not empty, the operation fails.

```
DROP DATABASE inventory_db;
```

Example 2: Drop a Non-Empty Database (Cascade Drop)

Drop database **inventory_db** and all tables within it.

```
DROP DATABASE inventory_db CASCADE;
```

Example 3: Safely Drop Using IF EXISTS

Drop database **inventory_db**. If the database does not exist, no error is raised.

```
DROP DATABASE IF EXISTS inventory_db CASCADE;
```

Example 4: Drop a Database Using the SCHEMA Keyword

SCHEMA is equivalent to **DATABASE** and can be used interchangeably.

```
DROP SCHEMA IF EXISTS inventory_db;
```

2.3 Describe Specific Database

Function

Returns metadata information of a specified database, including the database name, database description, storage path on the file system, etc. If the **EXTENDED** option is specified, database property information is also returned.

Syntax

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

Keywords

EXTENDED: In addition to displaying the above information, also displays database properties (**DBPROPERTIES**).

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 2-4 Parameter description

Parameter	Description
db_name	Database name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with a digit or underscore.

Precautions

- If the database to be described does not exist, the system reports an error.
- The **EXTENDED** option additionally displays property information defined in **DBPROPERTIES**.
- **DESC** is equivalent to **DESCRIBE** and can be used interchangeably.
- **DATABASE** is equivalent to **SCHEMA** and can be used interchangeably.

Example 1: Describe Basic Database Information

Describe the name, description, and storage path of the **employees** database.

```
DESCRIBE DATABASE employees;
```

Example result:

database_description_item	database_description_value
Database Name	employees
Description	For software companies
Location	obs://my-bucket/employees/

Example 2: Describe Detailed Database Information (Including Properties)

Use the **EXTENDED** option to view database property information.

```
DESCRIBE DATABASE EXTENDED employees;
```

Example result:

database_description_item	database_description_value
Database Name	employees
Description	For software companies
Location	obs://my-bucket/employees/
Properties	((Create-by,kevin), (Create-date,09/01/2019))

 Copyright © Huawei Technologies Co., Ltd.

Example 3: Use DESC Abbreviation

DESC is the abbreviation form of **DESCRIBE**.

```
DESC DATABASE employees;
```

Example 4: Describe a Database Created Using the SCHEMA Keyword

DATABASE is equivalent to **SCHEMA**.

```
DESC DATABASE deployment;
```

Example result:

database_description_item	database_description_value
Database Name	deployment
Description	Deployment environment
Location	obs://my-bucket/deployment/

2.4 Show All Databases

Function

Shows all databases under the current project. The **LIKE** clause can be used to filter database names by regular expression pattern. If no pattern is provided, all databases in the system are listed.

Syntax

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

Keywords

- **LIKE**: Filter results using a regular expression pattern.
- **DATABASES**/**SCHEMAS**: Both are equivalent and can be used.

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 2-5 Parameter description

Parameter	Description
regex_expression	A regular expression used to filter results. Except for * and |, the pattern works the same as a regular expression. * can be used to match zero or more characters, and | is used to separate multiple different regular expressions. Pattern matching is case-insensitive.

Precautions

DATABASES and **SCHEMAS** are equivalent and both return all database names.

Example 1: List All Databases

View all databases in the current instance.

```
SHOW DATABASES;
```

Table 2-6 Example result

databaseName
default
payments_db
payroll_db

Example 2: Filter Databases by Pattern

View all database names starting with **pay**.

```
SHOW DATABASES LIKE 'pay*';
```

Table 2-7 Example result

databaseName
payments_db
payroll_db

Example 3: Use the SCHEMAS Keyword

SCHEMAS is equivalent to **DATABASES** and can be used interchangeably.

 Copyright © Huawei Technologies Co., Ltd.

```
SHOW SCHEMAS;
```

Example 4: Filter Using Multiple Patterns

View databases starting with **pay** or **hr**.

```
SHOW DATABASES LIKE 'pay*|hr*';
```

2.5 Alter Database Properties

Function

The **ALTER DATABASE** statement is used to modify the properties or location of a database. The **DATABASE**, **SCHEMA**, and **NAMESPACE** keywords are interchangeable. If the database does not exist, an error is reported.

Syntax

Set properties

```
ALTER { DATABASE | SCHEMA | NAMESPACE } database_name
    SET { DBPROPERTIES | PROPERTIES } ( property_name = property_value [, ...] )
```

Unset properties

```
ALTER { DATABASE | SCHEMA | NAMESPACE } database_name
    UNSET { DBPROPERTIES | PROPERTIES } ( property_name [, ...] )
```

Modify location

```
ALTER { DATABASE | SCHEMA | NAMESPACE } database_name
    SET LOCATION 'new_location'
```

Keywords

- **DATABASE**/**SCHEMA**/**NAMESPACE**: All three are equivalent and interchangeable. **DATABASE** is recommended.
- **SET DBPROPERTIES**/**SET PROPERTIES**: Sets database properties. **DBPROPERTIES** and **PROPERTIES** are equivalent.
- **UNSET DBPROPERTIES**/**UNSET PROPERTIES**: Removes database properties. If the specified property does not exist, the command ignores it and executes successfully.
- **SET LOCATION**: Modifies the default storage location of the database.

Parameters

Table 2-8 Parameter descriptions

Parameter	Mandatory	Description
database_name	Yes	The name of the database to alter.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
SET DBPROPERTIES / SET PROPERTIES	Yes	Sets database properties. The specified property value overwrites the existing property value with the same name. Primarily used to record database metadata information.
UNSET DBPROPERTIES / UNSET PROPERTIES	Yes	Removes database properties. If the specified property does not exist, the command ignores it and executes successfully.
property_name	Yes	Property name.
property_value	Yes (for SET)	Property value.
SET LOCATION	Yes	Modifies the default storage location of the database. Does not move data from the current database directory to the new location, and does not affect the location of existing tables/partitions under the database.

Precautions

- Modifying the database location does not automatically migrate existing data. It only affects the default location for newly created tables in the future.
- The path for **SET LOCATION** must be in OBS format (e.g., **obs://bucket-name/path/**).
- The **DATABASE**, **SCHEMA**, and **NAMESPACE** keywords are interchangeable. **DATABASE** is recommended.
- The **DBPROPERTIES** and **PROPERTIES** keywords are interchangeable.
- Use **DESCRIBE DATABASE EXTENDED** to verify the results of property and location modifications.

Example 1: Set Database Properties

Create database **inventory** and set properties.

```
CREATE DATABASE IF NOT EXISTS inventory
  LOCATION 'obs://my-bucket/inventory/';

ALTER DATABASE inventory SET DBPROPERTIES ('Edited-by' = 'John', 'Edit-date' = '01/01/2001');
```

Verify properties:

```
DESCRIBE DATABASE EXTENDED inventory;
```

Example result:

 Copyright © Huawei Technologies Co., Ltd.

database_description_item	database_description_value
Database Name	inventory
Location	obs://my-bucket/inventory/
Properties	((Edit-date,01/01/2001), (Edited-by,John))

Example 2: Modify Database Location

```
ALTER DATABASE inventory SET LOCATION 'obs://my-bucket/new-inventory/';
```

Verify new location:

```
DESCRIBE DATABASE EXTENDED inventory;
```

Example 3: Remove Database Properties

```
ALTER DATABASE inventory UNSET DBPROPERTIES ('Edited-by');
```

Removing a non-existent property is ignored and the command executes successfully:

```
ALTER DATABASE inventory UNSET DBPROPERTIES ('non-existent');
```

Example 4: Use SCHEMA/NAMESPACE Keywords

SCHEMA and **NAMESPACE** are equivalent to **DATABASE** and can be used interchangeably.

```
ALTER SCHEMA inventory SET DBPROPERTIES ('Owner' = 'Admin');
ALTER NAMESPACE inventory SET DBPROPERTIES ('Department' = 'Engineering');
```

 Copyright © Huawei Technologies Co., Ltd.

3 Table Related

3.1 Table Type Description
3.2 Create Table
3.3 Drop Table
3.4 View Table
3.5 Alter Table
3.6 Table Alias
3.7 Partition Related

3.1 Table Type Description

Comparison Between External Table and Managed Table

AI DataLake Spark SQL endpoints support two table types: external table and managed table.

Feature	External Table	Managed Table
Data Storage Location	External OBS path (specified via **LOCATION**)	Platform internal storage
Drop Table Behavior	Only metadata is deleted. Data remains in OBS.	Both metadata and data are deleted.
Table Creation Keyword	Does not use the **EXTERNAL** keyword. Identified by specifying **LOCATION**.	**LOCATION** is not specified.
Data Lifecycle	OBS data is managed by the user.	Managed by the platform
Use Cases	Data already exists in OBS and needs to be shared across services.	Temporary data, intermediate results, and platform-managed data

 Copyright © Huawei Technologies Co., Ltd.

Feature	External Table	Managed Table
Difference from Open-Source Spark	Open-source Spark uses **CREATE EXTERNAL TABLE**. The platform does not use the **EXTERNAL** keyword.	Consistent with open-source Spark
Supported Table Creation Syntax	DataSource syntax and Hive syntax	DataSource syntax and Hive syntax

NOTE

The external table (OBS table) in AI DataLake is conceptually consistent with the external table in open-source Spark. However, the table creation syntax does not use the **EXTERNAL** keyword, which is the main difference from open-source Spark. External tables are created by specifying a **LOCATION 'obs://...'** path.

CREATE TABLE Default Storage Engine Change

In Spark 3.*x*, when **CREATE TABLE** does not specify a **USING** clause, the Hive storage format is used by default.

In Spark 4.0, the default storage format is determined by the **spark.sql.sources.default** configuration, which is typically **parquet**.

Table 3-1 Behavior change comparison

Scenario	Spark 3.x	Spark 4.0
CREATE TABLE t (id INT)	Default Hive format	Default Parquet format
CREATE TABLE t (id INT) USING hive	Hive format	Hive format (explicitly specified)
CREATE TABLE t (id INT) USING parquet	Parquet format	Parquet format (explicitly specified)

- **Usage recommendations:**
  - Explicitly specify the storage format in all **CREATE TABLE** statements to avoid relying on default values. For **CREATE TABLE** statements without **USING**, confirm the expected format.
  - AI DataLake recommends that all **CREATE TABLE** statements explicitly include a **USING** clause.
  - For managed tables in AI DataLake, **USING parquet** or **USING orc** is recommended.
- **Recommended syntax: explicitly specify the format**

```
-- Create a managed table using the Parquet format.
CREATE TABLE t1 (id INT, name STRING) USING parquet;
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- Create an OBS table using the Parquet format.
CREATE TABLE t3 (id INT, name STRING) USING parquet
LOCATION 'obs://bucket/path';
```

3.2 Create Table

3.2.1 Create Managed Table Using DataSource Syntax

Function

Create a managed table using DataSource syntax. The main differences between DataSource syntax and Hive syntax lie in the supported table data storage format range and the supported number of partitions. For details, refer to the syntax format and precautions.

Precautions

- CTAS (CREATE TABLE AS SELECT) statements cannot specify table properties.
- If no delimiter is specified, the default is a comma (,).
- Usage notes for partitioned tables:
  - When creating a partitioned table, the partition columns specified in **PARTITIONED BY** must be columns in the table, and their types must be specified in the column list. Partition columns only support the following types: STRING, BOOLEAN, TINYINT, SMALLINT, SHORT, INT, BIGINT, LONG, DECIMAL, FLOAT, DOUBLE, DATE, TIMESTAMP.
  - When creating a partitioned table, the partition columns must be the last column or columns in the table field list, and the order of multiple partition columns must also correspond. Otherwise, an error will occur.
  - The maximum number of partitions allowed for a single table is 200,000.

Syntax

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

Keywords

- **IF NOT EXISTS**: Specify this keyword to avoid an error if the table already exists.
- **USING**: Specify the storage format.
- **OPTIONS**: Specify property names and values when creating the table.
- **COMMENT**: Column or table description.
- **PARTITIONED BY**: Specify partition columns.
- **AS**: Create a table using CTAS.

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 3-2 Basic parameters

Parameter	Mandatory	Description
db_name	No	Database name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with an underscore.
table_name	Yes	Table name in the database. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with an underscore. The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. Special characters must be enclosed in single quotes (''). Table names are case-insensitive.
col_name	Yes	Comma-separated column names with data types. Column names consist of letters, digits, and underscores (_). It cannot be purely numeric and must contain at least one letter. Column names are case-insensitive.
col_type	Yes	Data type of the column field. The data type is a primitive type. Refer to Primitive Data Types.
col_comment	No	Column field description. Only string constants are supported.
file_format	Yes	Managed table data storage format. **PARQUET**, **ORC**, and **JSON** formats are supported.
table_comment	No	Table description. Only string constants are supported.
select_statement	No	Used for **CTAS** commands to insert the **SELECT** query results or specific data from the source table into the newly created managed table.

 Copyright © Huawei Technologies Co., Ltd.

Table 3-3 OPTIONS parameters

Parameter	Mandatory	Description	Default Value
multiLevelDirEnable	No	Whether to iteratively query data in subdirectories. When set to **true**, all files in the table path, including files in subdirectories, are read iteratively when querying the table.	false
compression	No	Specify the compression format. Typically specified for the Parquet format. The **zstd** compression format is recommended.	-

Example 1: Create a Basic Managed Table

Create a non-partitioned managed table named **table1**, specifying the storage format as ORC using the **USING** keyword.

```
CREATE TABLE IF NOT EXISTS table1 (
   col_1 STRING,
   col_2 INT
)
USING orc;
```

Example 2: Create a Managed Table with Default Values and Comments

Create a managed table named **employee** with column comments and a table comment.

```
CREATE TABLE IF NOT EXISTS employee (
   id      INT COMMENT 'Employee ID',
   name    STRING COMMENT 'Employee name',
   salary  DECIMAL(10,2) COMMENT 'Employee salary'
)
USING parquet
COMMENT 'Employee information management table',
```

Example 3: Create Managed Tables Using Different Formats (ORC, JSON)

Create managed tables using different data formats.

```
-- Using ORC format
CREATE TABLE IF NOT EXISTS table_orc (
   id      INT,
   data    STRING
)
USING orc;

-- Using JSON format
CREATE TABLE IF NOT EXISTS table_json (
   id      INT,
   info    STRING
)
USING json;
```

 Copyright © Huawei Technologies Co., Ltd.

Example 4: Create a Partitioned Managed Table

Create a partitioned managed table named **student**, partitioned by faculty number (**facultyNo**) and class number (**classNo**), with **OPTIONS** properties configured.

```
CREATE TABLE IF NOT EXISTS student (
    Name        STRING,
    facultyNo   INT,
    classNo     INT
)
USING parquet
PARTITIONED BY (facultyNo, classNo)
OPTIONS (
    compression = 'zstd'
);
```

3.2.2 Create Managed Table Using Hive Syntax

Function

Create a managed table using Hive syntax. The main differences between DataSource syntax and Hive syntax lie in the supported table data storage format range and the supported number of partitions. For details, refer to the syntax format and precautions.

Precautions

- CTAS (CREATE TABLE AS SELECT) statements cannot specify table properties.
- Hive managed tables do not support specifying multi-character delimiters when creating the table.
- Usage notes for partitioned tables:
  - When creating a partitioned table, the partition columns specified in **PARTITIONED BY** must be columns not in the table, and their data types must be specified. Partition columns support types supported by open-source Hive, such as STRING, BOOLEAN, TINYINT, SMALLINT, SHORT, INT, BIGINT, LONG, DECIMAL, FLOAT, DOUBLE, DATE, TIMESTAMP.
  - Multiple partition columns can be specified. Partition columns only need to be specified after the **PARTITIONED BY** keyword and cannot be specified after the table name like regular columns. Otherwise, an error will occur.
  - The maximum number of partitions allowed for a single table is 200,000.

Syntax

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

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

 Copyright © Huawei Technologies Co., Ltd.

```
[MAP KEYS TERMINATED BY char]
[LINES TERMINATED BY char]
[NULL DEFINED AS char]
```

Keywords

- **IF NOT EXISTS**: Specify this keyword to avoid an error if the table already exists.
- **COMMENT**: Column or table description.
- **PARTITIONED BY**: Specify partition columns.
- **ROW FORMAT**: Row data format.
- **STORED AS**: Specify the stored file format. Currently, this keyword only supports the following formats: TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, PARQUET. This keyword must be specified when creating a managed table.
- **TBLPROPERTIES**: Used to add key/value properties to the table. When the table storage format is PARQUET, the table compression format can be specified as zstd by specifying **TBLPROPERTIES(parquet.compression = 'zstd')**.
- **AS**: Create a table using CTAS.

Parameters

Table 3-4 Basic parameters

Parameter	Mandatory	Description
db_name	No	Database name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with an underscore.
table_name	Yes	Table name in the database. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with an underscore. The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. Special characters must be enclosed in single quotes ('). Table names are case-insensitive.
col_name	Yes	Column field name. Column field names consist of letters, digits, and underscores (_). It cannot be purely numeric and must contain at least one letter. Column names are case-insensitive.
col_type	Yes	Data type of the column field. The data type is a primitive type. Refer to Primitive Data Types.
col_comment	No	Column field description. Only string constants are supported.
row_format	No	Row data format. The **ROW FORMAT** function only supports tables of type TEXTFILE.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
file_format	Yes	Managed table data storage format. Supports TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, PARQUET.
table_comment	No	Table description. Only string constants are supported.
key = value	No	Set specific properties and values for **TBLPROPERTIES**. When the table storage format is PARQUET, the table compression format can be specified as zstd by specifying **TBLPROPERTIES(parquet.compression = 'zstd')**.
select_statement	No	Used for **CTAS** commands to insert the **SELECT** query results or specific data from the source table into the newly created managed table.

Example 1: Create a Basic Hive Format Managed Table

Create a non-partitioned managed table named **table1**, specifying the storage format as ORC using the **STORED AS** keyword.

```
CREATE TABLE IF NOT EXISTS table1 (
   col_1   STRING,
   col_2   INT
)
STORED AS orc;
```

Example 2: Create a Managed Table with STORED AS PARQUET

Create a PARQUET format managed table named **table_parquet**, specifying the compression format through **TBLPROPERTIES**.

```
CREATE TABLE IF NOT EXISTS table_parquet (
   id      INT,
   name    STRING,
   score   DOUBLE
)
STORED AS parquet
TBLPROPERTIES (
   parquet.compression = 'zstd'
);
```

Example 3: Create a Managed Table with ROW FORMAT DELIMITED

Create a TEXTFILE type managed table named **table4** with **ROW FORMAT** set.

```
CREATE TABLE IF NOT EXISTS table4 (
   col_1   STRING,
   col_2   INT
)
STORED AS textfile
ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '/'
   COLLECTION ITEMS TERMINATED BY '$'
   MAP KEYS TERMINATED BY '#'
```

 Copyright © Huawei Technologies Co., Ltd.

```
  LINES TERMINATED BY '\n'
  NULL DEFINED AS 'NULL';
```

Example 4: Create a Managed Table with SERDE

Create a managed table named **table_serde** using a custom SERDE class.

```
CREATE TABLE IF NOT EXISTS table_serde (
  id      INT,
  data    STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES (
  'field.delim' = ',',
  'serialization.format' = ','
)
STORED AS textfile;
```

3.2.3 Create External Table Using DataSource Syntax

Function

Create an external table using DataSource syntax. The main differences between DataSource syntax and Hive syntax lie in the supported table data storage format range and the supported number of partitions. For details, refer to the syntax format and precautions.

You are advised to use the OBS parallel file system for storage. The parallel file system is a high-performance file system that provides millisecond-level access latency, TB/s-level bandwidth, and millions of IOPS, making it suitable for interactive big data analytics scenarios.

Precautions

- To view the table size, check it through OBS.
- CTAS (CREATE TABLE AS SELECT) statements cannot specify table properties.
- Scenario where the OBS directory contains subdirectories: When creating a table, if the specified path is a directory on OBS and that directory contains subdirectories (or nested subdirectories), all file types and their contents under the subdirectories are also part of the table content. You must ensure that all file types under the specified directory and its subdirectories are consistent with the storage format specified in the **CREATE TABLE** statement, and that all file contents are consistent with the fields in the table. Otherwise, the query will report an error. You can set **multiLevelDirEnable** to **true** in the **OPTIONS** of the **CREATE TABLE** statement to query content under subdirectories. The default value for this parameter is **false** (Note: This configuration item is a table property; configure it with caution. Hive tables do not support this configuration item).
- Usage notes for partitioned tables:
  - When creating a partitioned table, the partition columns specified in **PARTITIONED BY** must be columns in the table, and their types must be specified in the column list. Partition columns only support the following types: STRING, BOOLEAN, TINYINT, SMALLINT, SHORT, INT, BIGINT, LONG, DECIMAL, FLOAT, DOUBLE, DATE, TIMESTAMP.

 Copyright © Huawei Technologies Co., Ltd.

  - When creating a partitioned table, the partition columns must be the last column or columns in the table field list, and the order of multiple partition columns must also correspond. Otherwise, an error will occur.
  - The maximum number of partitions allowed for a single table is 200,000.
- Differences between external table and managed table creation:

Difference Item	External Table	Managed Table
LOCATION	Mandatory. Specifies the OBS storage path.	Not specified
Data deletion	**DROP TABLE** only deletes metadata.	**DROP TABLE** deletes both metadata and data.
EXTERNAL keyword	Not used (different from open-source)	Not used
USING / STORED AS	Must be specified.	Must be specified.

Difference from open-source Spark: Open-source Spark uses **CREATE EXTERNAL TABLE ... LOCATION** to create external tables. The platform does not use the **EXTERNAL** keyword. Instead, it distinguishes external tables from managed tables by specifying a **LOCATION** path.

Syntax

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

Keywords

- **IF NOT EXISTS**: Specify this keyword to avoid an error if the table already exists.
- **USING**: Specify the storage format.
- **OPTIONS**: Specify property names and values when creating the table.
- **COMMENT**: Column or table description.
- **PARTITIONED BY**: Specify partition columns.
- **AS**: Create a table using CTAS.

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 3-5 Basic parameters

Parameter	Mandatory	Description
db_name	No	Database name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with an underscore.
table_name	Yes	Table name to be created in the database. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with a digit or underscore. The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. Special characters must be enclosed in single quotes (''). Table names are case-insensitive.
col_name	Yes	Comma-separated column names with data types. Column names consist of letters, digits, and underscores (_). It cannot be purely numeric and must contain at least one letter. Column names are case-insensitive.
col_type	Yes	Data type of the column. The data type is a primitive type. Refer to Primitive Data Types.
col_comment	No	Column field description. Only string constants are supported.
file_format	Yes	Input format for creating the table. Supports ORC, PARQUET, JSON, CSV, AVRO types.
path	Yes	OBS storage path where the data file is located. You are advised to use the OBS parallel file system for storage. Format: **obs://bucketName/tblPath**, where **bucketName** is the bucket name and **tblPath** is the directory name. The file name does not need to be specified after the directory. When a folder and a file have the same name under an OBS directory, the path pointed to by the OBS table creation will prioritize the file over the folder.
table_comment	No	Table description information. Only string constants are supported.
select_statement	No	Used for **CTAS** commands to insert the **SELECT** query results or specific data from the source table into the newly created OBS table.

 Copyright © Huawei Technologies Co., Ltd.

Table 3-6 OPTIONS parameters

Parameter	Mandatory	Description	Default Value
path	No	Specified table path, i.e., the OBS storage path.	-
multiLevelDirEnable	No	In nested subdirectory scenarios, whether to iteratively query data in subdirectories. When set to **true**, all files in the table path, including files in subdirectories, are read iteratively when querying the table.	false
dataDelegated	No	Whether to clear data under the **path** when deleting a table or partition.	false
compression	No	Specify the compression format. Typically specified for the Parquet format. The **zstd** compression format is recommended.	-

Table 3-7 CSV data format OPTIONS parameters

Parameter	Mandatory	Description	Default Value
delimiter	No	Data delimiter.	,
quote	No	Quote character.	"
escape	No	Escape character.	\
multiLine	No	Whether column data contains carriage returns or line breaks. **true** means it contains them; **false** means it does not.	false
dateFormat	No	Specifies the date format for **DATE** fields in the CSV file.	yyyy-MM-dd
timestampFormat	No	Specifies the date format for **TIMESTAMP** fields in the CSV file.	yyyy-MM-dd HH:mm:ss

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description	Default Value
mode	No	Specifies the mode for parsing CSV. **PERMISSIVE**: Permissive mode, sets erroneous fields to NULL. **DROPMALFORMED**: Drops the entire row when encountering erroneous fields. **FAILFAST**: Error mode, reports an error directly when encountering erroneous fields.	PERMISSIVE
header	No	Whether the CSV contains a header. **true** means it contains a header; **false** means it does not.	false
nullValue	No	Sets the character representing NULL. For example, **nullValue="nl"** means **nl** represents NULL.	-
comment	No	Sets the character representing the start of a comment. For example, **comment='#'** means lines starting with **#** are comments.	-
compression	No	Sets the compression format for the data. Currently supports gzip, bzip2, deflate compression formats. Enter **none** if compression is not desired.	none
encoding	No	Encoding format for the data. Supports utf-8, gb2312, gbk.	utf-8

Example 1: Create a Basic OBS Non-Partitioned Table

Create a non-partitioned OBS table named **table1**, specifying the storage format as ORC using the **USING** keyword.

```
CREATE TABLE IF NOT EXISTS table1 (
   col_1   STRING,
   col_2   INT
)
USING orc
OPTIONS (path 'obs://bucketName/filePath');
```

 Copyright © Huawei Technologies Co., Ltd.

Example 2: Create an OBS Partitioned Table

Create a partitioned table named **student**, partitioned by faculty number (**facultyNo**) and class number (**classNo**).

```
CREATE TABLE IF NOT EXISTS student (
    Name        STRING,
    facultyNo   INT,
    classNo     INT
)
USING csv
OPTIONS (path 'obs://bucketName/filePath')
PARTITIONED BY (facultyNo, classNo);
```

Example 3: Create an OBS Partitioned Table with Comments and Properties

Create an OBS partitioned table named **table3** with column comments, table comments, and OPTIONS property configuration.

```
CREATE TABLE IF NOT EXISTS table3 (
    col_1   STRING COMMENT 'Username',
    col_2   INT COMMENT 'User ID'
)
USING parquet
OPTIONS (
    path 'obs://bucketName/filePath',
    multiLevelDirEnable = true,
    dataDelegated = true,
    compression = 'zstd'
)
PARTITIONED BY (col_2)
COMMENT 'User information table'
```

Example 4: Create OBS Tables Using Different Data Formats

Create OBS tables using different data formats such as JSON, AVRO.

```
-- Using JSON format
CREATE TABLE IF NOT EXISTS table_json (
    id      INT,
    name    STRING
)
USING json
OPTIONS (path 'obs://bucketName/jsonPath');

-- Using AVRO format
CREATE TABLE IF NOT EXISTS table_avro (
    id      INT,
    data    STRING
)
USING avro
OPTIONS (path 'obs://bucketName/avroPath');
```

3.2.4 Create External Table Using Hive Syntax

Function

Create an OBS table using Hive syntax. The main differences between DataSource syntax and Hive syntax lie in the supported table data storage format range and the supported number of partitions. For details, refer to the syntax format and precautions.

You are advised to use the OBS parallel file system for storage. The parallel file system is a high-performance file system that provides millisecond-level access

 Copyright © Huawei Technologies Co., Ltd.

latency, TB/s-level bandwidth, and millions of IOPS, making it suitable for interactive big data analytics scenarios.

Precautions

- To view the table size, check it through OBS.
- CTAS (CREATE TABLE AS SELECT) statements cannot specify table properties.
- **Usage notes for partitioned tables:**
  - When creating a partitioned table, the partition columns specified in **PARTITIONED BY** must be columns not in the table, and their data types must be specified. Partition columns support types supported by open-source Hive, such as STRING, BOOLEAN, TINYINT, SMALLINT, SHORT, INT, BIGINT, LONG, DECIMAL, FLOAT, DOUBLE, DATE, TIMESTAMP.
  - Multiple partition columns can be specified. Partition columns only need to be specified after the **PARTITIONED BY** keyword and cannot be specified after the table name like regular columns. Otherwise, an error will occur.
  - The maximum number of partitions allowed for a single table is 200,000.
- **Setting multi-character delimiters when creating a table:**
  - Multi-character field delimiters are only supported when **ROW FORMAT SERDE** is specified as **org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe**.
  - Only Hive OBS tables support specifying multi-character delimiters when creating the table. Hive managed tables do not support specifying multi-character delimiters when creating the table.
  - Tables with multi-character delimiters do not support write statements such as **INSERT** and **IMPORT**. To add data, place the data file directly in the corresponding OBS path of the table.
- **Differences between external table and managed table creation:**

Difference Item	External Table	Managed Table
LOCATION	Mandatory. Specifies the OBS storage path.	Not specified
Data deletion	**DROP TABLE** only deletes metadata.	**DROP TABLE** deletes both metadata and data.
EXTERNAL keyword	Not used (different from open-source)	Not used
STORED AS	Must be specified.	Must be specified.

*Difference from open-source Spark: Open-source Spark uses **CREATE EXTERNAL TABLE ... LOCATION** to create external tables. The platform does not use the **EXTERNAL** keyword. Instead, it distinguishes external tables from managed tables by specifying a **LOCATION** path.*

 Copyright © Huawei Technologies Co., Ltd.

Syntax

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

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

Keywords

- **IF NOT EXISTS**: Specify this keyword to avoid an error if the table already exists.
- **COMMENT**: Column or table description.
- **PARTITIONED BY**: Specify partition columns.
- **ROW FORMAT**: Row data format.
- **STORED AS**: Specify the stored file format. Currently, this keyword only supports the following formats: TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, PARQUET.
- **LOCATION**: Specify the OBS path. This keyword must be specified when creating an OBS table.
- **TBLPROPERTIES**: Allows users to add key/value properties to the table.
- **AS**: Create a table using CTAS.

Parameters

Table 3-8 Basic parameters

Parameter	Mandatory	Description
db_name	No	Database name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with a digit or underscore.
table_name	Yes	Table name in the database. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with a digit or underscore. The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. Special characters must be enclosed in single quotes (''). Table names are case-insensitive.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
col_name	Yes	Column field name. Column field names consist of letters, digits, and underscores (_). It cannot be purely numeric and must contain at least one letter. Column names are case-insensitive.
col_type	Yes	Data type of the column. The data type is a primitive type. Refer to Primitive Data Types.
col_comment	No	Column field description. Only string constants are supported.
row_format	No	Row data format. The **ROW FORMAT** function only supports tables of type TEXTFILE.
file_format	Yes	OBS table storage format. Supports TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, PARQUET.
table_comment	No	Table description. Only string constants are supported.
obs_path	Yes	OBS storage path where the data file is located. You are advised to use the OBS parallel file system for storage. Format: **obs://bucketName/tblPath**, where **bucketName** is the bucket name and **tblPath** is the directory name. The file name does not need to be specified after the directory. The file name does not need to be specified after the directory. When a folder and a file have the same name under an OBS directory, the path pointed to by the OBS table creation will prioritize the file over the folder.
key = value	No	Set specific properties and values for **TBLPROPERTIES**.
select_statement	No	Used for **CTAS** commands to insert the **SELECT** query results or specific data from the source table into the newly created OBS table.

Table 3-9 TBLPROPERTIES main parameters

Parameter	Mandatory	Description
comment	No	Table description information.
orc.compress	No	A property for ORC storage format tables, used to specify the compression method for ORC storage. Supported values: ZLIB, SNAPPY, NONE.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
auto.purge	No	When set to **true**, deleted or overwritten data will be directly deleted without going through the recycle bin.

Example 1: Create a Basic Hive Format OBS Non-Partitioned Table

Create a non-partitioned OBS table named **table1**, specifying the storage format as ORC using the **STORED AS** keyword.

```
CREATE TABLE IF NOT EXISTS table1 (
   col_1   STRING,
   col_2   INT
)
STORED AS orc
LOCATION 'obs://bucketName/filePath';
```

Example 2: Create an OBS Table with STORED AS TEXTFILE

Create a TEXTFILE format OBS table named **table_text**.

```
CREATE TABLE IF NOT EXISTS table_text (
   id      INT,
   name    STRING
)
STORED AS textfile
LOCATION 'obs://bucketName/textFilePath';
```

Example 3: Create an OBS Table with ROW FORMAT

Create a TEXTFILE type non-partitioned OBS table named **table4** with **ROW FORMAT** set.

```
CREATE TABLE IF NOT EXISTS table4 (
   col_1   STRING,
   col_2   INT
)
STORED AS textfile
LOCATION 'obs://bucketName/filePath'
ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '/'
   COLLECTION ITEMS TERMINATED BY '$'
   MAP KEYS TERMINATED BY '#'
   LINES TERMINATED BY '\n'
   NULL DEFINED AS 'null';
```

Example 4: Create an OBS Table with SERDE

Create an OBS table named **table5** using a custom SERDE with multi-character delimiters.

```
CREATE TABLE IF NOT EXISTS table5 (
   col_1   STRING,
   col_2   INT
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe'
WITH SERDEPROPERTIES (
   'field.delim' = '/#'
```

 Copyright © Huawei Technologies Co., Ltd.

```
)
STORED AS textfile
LOCATION 'obs://bucketName/filePath';
```

3.3 Drop Table

Function

Drop a table. **DROP TABLE** is used to drop an existing table and its metadata.

- OBS table (external table): Only metadata information is deleted. The data stored on OBS is not deleted.
- Management table: Both data and corresponding metadata information are deleted.

Syntax

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

Keywords

IF EXISTS: If specified, no error is reported when the table does not exist. If not specified and the table does not exist, an exception is thrown.

Parameters

Parameter	Mandatory	Description
db_name	No	Database name. If not specified, the current database is used by default. Naming rule: Only letters, digits, and underscores (_) are allowed. Cannot be purely numeric and cannot start with a digit or underscore.
table_name	Yes	Name of the table to be dropped.

Precautions

- The table to be dropped must exist in the current database. Otherwise, an error will occur. Adding **IF EXISTS** can prevent the error.
- For OBS tables (external tables), **DROP TABLE** only deletes metadata. The data files under the OBS path are retained. For managed tables, both metadata and data are deleted.

Example 1: Drop Table

Drop the table named **employee** in the current database.

```
DROP TABLE employee;
```

 Copyright © Huawei Technologies Co., Ltd.

Example 2: Use IF EXISTS to Avoid Errors

Drop the table named **employee**. If the table does not exist, no error is reported.

```
DROP TABLE IF EXISTS employee;
```

Example 3: Drop a Table in a Specified Database

Drop the table named **employee** in the **company_db** database.

```
DROP TABLE IF EXISTS company_db.employee;
```

Example 4: Difference Between OBS Table and Managed Table

The following example demonstrates the behavioral difference when dropping an OBS table versus a managed table.

```
-- Drop a managed table: Both metadata and data are deleted.
DROP TABLE IF EXISTS managed_table;

-- Drop an OBS table (external table): Only metadata is deleted. Data files on OBS are retained.
DROP TABLE IF EXISTS obs_table;
```

To clean up the data files corresponding to an OBS table, manually delete the files under the OBS path, for example, ***obs://bucket-name/path/to/data/****.*

3.4 View Table

3.4.1 Show All Tables

Function

View all tables and views in the current database. Use **IN**/**FROM** to specify a database, and use **LIKE** for pattern matching filtering.

Syntax

```
SHOW TABLES [ { FROM | IN } db_name ] [ LIKE regex_expression ]
```

Parameters

Parameter	Mandatory	Description
FROM / IN	No	Specify the database name to view tables and views under a specific database. **FROM** and **IN** have the same function and are interchangeable.
db_name	No	Database name. If not specified, the current database is viewed by default. Naming rule: Only letters, digits, and underscores (_) are allowed. Cannot be purely numeric and cannot start with a digit or underscore.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
LIKE regex_expression	No	Uses a regular expression for pattern matching filtering of table names. Supports wildcards such as ***** (matches any sequence of characters) and **|** (OR).

Precautions

None

Example 1: View All Tables and Views in the Current Database

```
SHOW TABLES;
```

Example result:

```
+---------+---------+-----------+
|database |tableName|isTemporary|
+---------+---------+-----------+
|default  |employee |false      |
|default  |student  |false      |
|         |temp_view|true       |
+---------+---------+-----------+
```

Example 2: View All Tables in a Specified Database

View all tables in the **company_db** database.

```
SHOW TABLES FROM company_db;
```

Example 3: Use LIKE Pattern to Filter Table Names

View all tables in the current database whose names start with **emp**.

```
SHOW TABLES LIKE "emp*";
```

Example 4: View Tables with a Specific Prefix in a Specified Database

View all tables in the **company_db** database whose names start with **dept**.

```
SHOW TABLES IN company_db LIKE "dept*";
```

3.4.2 Show Create Table

Function

Return the `CREATE TABLE` statement for the corresponding table. **SHOW CREATE TABLE** is used to view the complete definition statement of a table.

- By default, the DataSource format **CREATE TABLE** statement is returned.
- Using the **AS SERDE** clause returns the Hive format **CREATE TABLE** statement, suitable for Hive SerDe tables.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
SHOW CREATE TABLE [ db_name. ] table_name [ AS SERDE ]
```

Parameters

Parameter	Mandatory	Description
db_name	No	Database name. If not specified, the current database is used by default.
table_name	Yes	Table name. The specified table must exist. Otherwise, an error will occur.
AS SERDE	No	Used to generate a Hive SerDe format **CREATE TABLE** statement, suitable for tables using Hive storage format. If not specified, the DataSource format **CREATE TABLE** statement is generated by default.

Precautions

- The specified table must exist. Otherwise, an error will occur.
- Without **AS SERDE**, the DataSource format **CREATE TABLE** statement is returned (recommended).
- With **AS SERDE**, the Hive SerDe format **CREATE TABLE** statement is returned, including Hive-specific properties such as **ROW FORMAT SERDE**, **INPUTFORMAT**, **OUTPUTFORMAT**.

Example 1: Show Create Table Statement

View the **CREATE TABLE** statement for the **employee** table.

```
SHOW CREATE TABLE employee;
```

Example result (DataSource format):

```
+----------------------+
|createtab_stmt      |
+----------------------+
|CREATE TABLE employee (
  id INT,
  name STRING,
  salary DOUBLE)
USING parquet
TBLPROPERTIES (
  'transient_lastDdlTime' = '1718534400')
+----------------------+
```

Example 2: View the Create Table Statement for a DataSource Table

Create a DataSource format table and view its **CREATE TABLE** statement.

```
-- Create a DataSource format table.
CREATE TABLE sales (order_id INT, amount DOUBLE, order_date DATE)
  USING parquet
  LOCATION 'obs://my-bucket/warehouse/sales';
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- View the Create Table statement.
SHOW CREATE TABLE sales;
```

Example 3: View the Create Table Statement for a Hive Format Table (AS SERDE)

Use **AS SERDE** to view the Hive format **CREATE TABLE** statement.

```
-- Create a Hive format table.
CREATE TABLE logs (id INT, message STRING)
  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
  STORED AS TEXTFILE;

-- View the CREATE TABLE statement in Hive SerDe format.
SHOW CREATE TABLE logs AS SERDE;
```

Example result (Hive SerDe format):

```
+----------------------+
|createtab_stmt        |
+----------------------+
|CREATE TABLE logs (
  id INT,
  message STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
STORED AS
  INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
  OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
TBLPROPERTIES (
  'serialization.format' = '1',
  'transient_lastDdlTime' = '1718534400')
+----------------------+
```

Example 4: View the Create Table Statement for a Table in a Specified Database

View the **CREATE TABLE** statement for the **employee** table in the **company_db** database.

```
SHOW CREATE TABLE company_db.employee;
```

3.4.3 Show Table Properties

Function

View table properties. **SHOW TBLPROPERTIES** is used to return key-value pair information of table properties. You can view all properties or the value of a specific property.

Syntax

```
SHOW TBLPROPERTIES [ db_name. ] table_name [ ( 'property_name' ) ]
```

Keywords

SHOW TBLPROPERTIES: Keyword for viewing table properties. **TBLPROPERTIES** and **PROPERTIES** are interchangeable.

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Description
db_name	No	Database name. If not specified, the current database is used by default.
table_name	Yes	Table name.
property_name	No	Property name, must be enclosed in single quotes. If not specified, all properties and their values are returned. If specified, only the value of that property is returned. Note: **property_name** is case-sensitive. Multiple **property_name** values cannot be specified simultaneously. Otherwise, an error will occur.

Precautions

- **property_name** is case-sensitive. For example, **owner** and **Owner** are different properties.
- Multiple **property_name** values cannot be specified simultaneously. Otherwise, an error will occur.
- If the specified **property_name** does not exist, a null value is returned.

Example 1: View All Properties of a Table.

View all properties and their values for the **employee** table.

```
SHOW TBLPROPERTIES employee;
```

Example result:

```
+--------------------+--------------------+
|key                 |value               |
+--------------------+--------------------+
|owner               |spark               |
|transient_lastDdlTime|1718534400         |
+--------------------+--------------------+
```

Example 2: View the Value of a Specified Property

View the value of the property **owner** in the **employee** table.

```
SHOW TBLPROPERTIES employee ('owner');
```

Example result:

```
+-----+
|value|
+-----+
|spark|
+-----+
```

 Copyright © Huawei Technologies Co., Ltd.

Example 3: View Properties of a Table with Custom Properties

First create a table with custom **TBLPROPERTIES**, then view its properties.

```
-- Create a table with custom properties.
CREATE TABLE employee (id INT, name STRING)
  TBLPROPERTIES ('owner' = 'admin', 'department' = 'hr', 'created_by' = 'spark_sql');

-- View all properties.
SHOW TBLPROPERTIES employee;
```

Example 4: Case Sensitivity of property_name

property_name is case-sensitive. The following queries will return different results.

```
-- View the value of the lowercase property name owner (correct match).
SHOW TBLPROPERTIES employee ('owner');

-- View the value of the uppercase property name Owner (different from owner, may return null).
SHOW TBLPROPERTIES employee ('Owner');
```

owner and **Owner** are treated as different property names. Ensure the case matches the definition when using.

3.4.4 Show All Columns of Specified Table

Function

View all columns in a specified table. Use **FROM**/**IN** to specify the database to which the table belongs.

Syntax

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

Parameters

Parameter	Mandatory	Description
table_name	Yes	Table name.
db_name	No	Database name. Used to qualify the database to which the table belongs.
FROM / IN	Yes (first group)	Specify the table whose columns are to be viewed. **FROM** and **IN** have the same function and are interchangeable.
FROM / IN	No (second group)	Specify the database to which the table belongs. Same function as the first group of keywords.

 Copyright © Huawei Technologies Co., Ltd.

Precautions

- The specified table must exist in the database. Otherwise, an error will occur.
- The first group of **FROM**/**IN** is followed by the table name. The second group of **FROM**/**IN** is followed by the database name. Both have the same function but different positional meanings.

Example 1: View All Columns of a Table

View all columns in the **student** table.

```
SHOW COLUMNS FROM student;
```

Example result:

```
+---------+
|col_name |
+---------+
|id       |
|name     |
|age      |
|score    |
+---------+
```

Example 2: View Columns of a Table in a Specified Database

View all columns in the **student** table in the **school_db** database.

```
SHOW COLUMNS FROM student FROM school_db;
```

Example 3: View Columns of a Table Using the IN Keyword

View all columns in the **student** table using the **IN** keyword.

```
SHOW COLUMNS IN student;
```

Example 4: Specify a Database Using the IN Keyword

View all columns in the **student** table in the **school_db** database using the **IN** keyword.

```
SHOW COLUMNS IN student IN school_db;
```

3.4.5 Show All Partitions of Specified Table

Function

View all partitions of a specified table. Use the **PARTITION** clause to specify partition conditions for filtering. Partial partition key matching is supported.

Syntax

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

The syntax for **partition_specs** is:

```
partition_specs = ( partition_col_name = partition_col_value [ , partition_col_name =
partition_col_value , ... ] )
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Description
db_name	No	Database name. If not specified, the current database is used by default. Naming rule: Only letters, digits, and underscores (_) are allowed. Cannot be purely numeric and cannot start with an underscore.
table_name	Yes	Table name. The table being viewed must be a partitioned table. Otherwise, an error will occur. Naming rule: Only letters, digits, and underscores (_) are allowed. Cannot be purely numeric and cannot start with an underscore. The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If it contains special characters, it must be enclosed in backticks (``).
PARTITION partition_specs	No	Partition filter condition, in the format of **key=value**. If the partition has multiple fields, not all fields need to be included. The system will return all matching partition information.

Precautions

- The table whose partitions are to be viewed must exist and be a partitioned table. Otherwise, an error will occur.
- In **partition_specs**, only some partition keys can be specified. The system will return all matching partitions. For example, for a three-level partitioned table **(year, month, day)**, specifying only **year=2024** will return all partitions under that year.

Example 1: View All Partitions of a Table

View all partitions under the **student** table.

```
SHOW PARTITIONS student;
```

Example result:

```
+-------------------+
|partition          |
+-------------------+
|year=2024/month=01 |
|year=2024/month=02 |
|year=2024/month=03 |
+-------------------+
```

Example 2: View a Specific Partition

View the partition in the **student** table where **year=2024** and **month=01**.

 Copyright © Huawei Technologies Co., Ltd.

```
SHOW PARTITIONS student PARTITION(year='2024', month='01');
```

Example 3: Partial Partition Key Matching

View all partitions in the **student** table where **year=2024** (without specifying **month**, all monthly partitions under that year will be returned).

```
SHOW PARTITIONS student PARTITION(year='2024');
```

Example result:

```
+-------------------+
|partition          |
+-------------------+
|year=2024/month=01 |
|year=2024/month=02 |
|year=2024/month=03 |
+-------------------+
```

Example 4: View Partitions of a Table in a Specified Database

View all partitions of the **student** table in the **school_db** database.

```
SHOW PARTITIONS school_db.student;
```

3.4.6 Show Table Statistics

Function

View table statistics. By default, the column names and column data types of all columns are returned. Use **EXTENDED** or **FORMATTED** to view detailed table metadata information.

DESCRIBE and the abbreviation **DESC** are equivalent.

Syntax

{ DESCRIBE | DESC } [EXTENDED | FORMATTED] [db_name.] table_name

Parameters

Parameter	Mandatory	Description
DESCRIBE / DESC	Yes	Keyword for viewing table statistics. Both are equivalent. **DESC** is the abbreviation for **DESCRIBE**.
EXTENDED	No	Displays all metadata information of the table, including detailed table statistics and configuration information. Typically used during debugging.
FORMATTED	No	Displays all metadata information in a formatted table. Similar to **EXTENDED** but with more readable output.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
db_name	No	Database name. If not specified, the current database is used by default. Naming rule: Only letters, digits, and underscores (_) are allowed. Cannot be purely numeric and cannot start with an underscore.
table_name	Yes	Table name. Naming rule: Only letters, digits, and underscores (_) are allowed. Cannot be purely numeric and cannot start with an underscore. The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. If it contains special characters, it must be enclosed in backticks (``).

Precautions

- If the table being viewed does not exist, an error will occur.
- By default, only column names and data types are returned. Use **EXTENDED** or **FORMATTED** to view complete metadata including partition information, storage format, table properties, etc.

Example 1: Basic DESCRIBE to View Table Information

View the column names and column data types of all columns in the **student** table.

```
DESCRIBE student;
```

Example result:

```
+---------+----------+-------+
|col_name |data_type |comment|
+---------+----------+-------+
|id       |int       |NULL   |
|name     |string    |NULL   |
|age      |int       |NULL   |
|score    |double    |NULL   |
+---------+----------+-------+
```

Example 2: Use the DESC Abbreviation

Use the **DESC** abbreviation to view column information of the **student** table. Equivalent to **DESCRIBE**.

```
DESC student;
```

Example 3: Use EXTENDED to View Detailed Information

View detailed metadata information of the **student** table, including partitions, storage format, table properties, etc.

```
DESCRIBE EXTENDED student;
```

The returned result will include:

 Copyright © Huawei Technologies Co., Ltd.

- Basic column information (column name, data type, comment)
- Detailed table metadata (database, owner, creation time, storage format, table properties, etc.)

Example 4: View a Table in a Specified Database

View column information of the **student** table in the **school_db** database.

```
DESCRIBE school_db.student;
```

3.5 Alter Table

3.5.1 Add Columns

Function

Adds one or more new columns to a table.

Syntax

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

Keywords

- **ADD COLUMNS**: Adds columns.
- **COMMENT**: Describe columns.

Parameters

Parameter	Mandatory	Description
db_name	No	Database name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with an underscore.
table_name	Yes	Table name.
col_name	Yes	Column field name.
col_type	Yes	Column field type.
col_comment	No	Column description. Only string constants are supported.

Precautions

This SQL statement is not recommended for concurrent execution. In concurrent scenarios, the results of adding columns may overwrite each other.

 Copyright © Huawei Technologies Co., Ltd.

Example 1: Add a Single Column

Add an INT type column **age** to table **t1**.

```
ALTER TABLE t1 ADD COLUMNS (age INT);
```

Example 2: Add Multiple Columns

Add two columns, **column2** and **column3**, to table **t1** simultaneously.

```
ALTER TABLE t1 ADD COLUMNS (column2 INT, column3 STRING);
```

Example 3: Add a Column with a Comment

Add a STRING type column **address** to table **t1** with a column comment.

```
ALTER TABLE t1 ADD COLUMNS (address STRING COMMENT 'User address information');
```

Example 4: Add a Column to a Table in a Specified Database

Add a column **salary** to the table **employee** in database **mydb**.

```
ALTER TABLE mydb.employee ADD COLUMNS (salary DECIMAL(10,2) COMMENT 'Employee salary');
```

3.5.2 Modify Column Comment

Function

Modify the column comment of a non-partitioned table or a partitioned table. Spark 4.0 recommends using the **ALTER COLUMN** syntax and is also compatible with the **CHANGE COLUMN** syntax.

Syntax

- **Syntax 1 (recommended, Spark 4.0):**
  ```
  ALTER TABLE [db_name.]table_name ALTER COLUMN col_name COMMENT 'col_comment';
  ```
- **Syntax 2 (compatible):**
  ```
  ALTER TABLE [db_name.]table_name CHANGE COLUMN col_name col_name col_type COMMENT 'col_comment';
  ```

Keywords

- **ALTER COLUMN**: Modify column properties (recommended syntax for Spark 4.0).
- **CHANGE COLUMN**: Modify column (compatible syntax, requires specifying both column name and type).
- **COMMENT**: Describe columns.

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Description
db_name	No	Database name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with an underscore.
table_name	Yes	Table name.
col_name	Yes	Column field name. Must be an existing column.
col_type	Mandatory only for **CHANGE COLUMN** syntax	Column data type. The **CHANGE COLUMN** syntax does not support modifying the column data type. Specify the data type used when creating the table. The **ALTER COLUMN** syntax does not require this.
col_comment	Yes	Modified column comment. The comment content is a valid string with a length not exceeding 1,024 bytes.

Precautions

- Spark 4.0 recommends using the **ALTER COLUMN** syntax to modify column comments. This syntax is more concise and does not require repeatedly specifying the column data type.
- The **CHANGE COLUMN** syntax is a compatible syntax. When using it, you must specify the same column data type as when the table was created.

Example 1: Modify Comment Using ALTER COLUMN (Recommended)

Use the **ALTER COLUMN** syntax recommended by Spark 4.0 to modify the comment of column **c1** in table **t1**.

```
ALTER TABLE t1 ALTER COLUMN c1 COMMENT 'the new comment';
```

Example 2: Modify Comment Using CHANGE COLUMN (Compatible)

Use the compatible syntax to modify the comment of column **c1** in table **t1**, requiring both the column name and data type.

```
ALTER TABLE t1 CHANGE COLUMN c1 c1 STRING COMMENT 'the new comment';
```

Example 3: Modify Column Comment in a Specified Database

Modify the comment of the **name** column in the **employee** table of database **mydb**.

```
ALTER TABLE mydb.employee ALTER COLUMN name COMMENT 'Employee name';
```

 Copyright © Huawei Technologies Co., Ltd.

Example 4: Modify Comments for Multiple Columns

Modify the comments for multiple columns in table **t1** separately.

```
ALTER TABLE t1 ALTER COLUMN c1 COMMENT 'Description for column 1';
ALTER TABLE t1 ALTER COLUMN c2 COMMENT 'Description for column 2';
```

3.6 Table Alias

Function

Assign an alias to a table or a subquery result.

Syntax

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

Keywords

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

Precautions

- The table being queried must already exist. Otherwise, an error will occur.
- The alias must be named before it is used. Otherwise, an error will occur. Additionally, you are advised to avoid duplicate names.

Examples

- Assign the alias **n** to the table **simple_table** and use **n.name** to access the **name** field in **simple_table**.
  ```
  SELECT n.score FROM simple_table n WHERE n.name = "leilei";
  ```
- Name the result of a subquery as **m** and use **SELECT * FROM m** to return all results from the subquery.
  ```
  SELECT * FROM (SELECT * FROM simple_table WHERE score > 90) AS m;
  ```

3.7 Partition Related

3.7.1 Add Partition

Function

Add one or more partitions to an existing OBS partitioned table. This command registers the partition metadata information in the metadata database, enabling subsequent queries to retrieve data based on partition columns.

There are two main scenarios for generating partition information in an OBS partitioned table:

 Copyright © Huawei Technologies Co., Ltd.

- Insert corresponding partition data into the OBS partitioned table. After the data is inserted successfully, the partition metadata information is automatically generated.
- Manually copy the partition directory and data to the OBS partitioned table path, then execute this command to add the partition metadata information.

Syntax

```
ALTER TABLE table_name ADD [IF NOT EXISTS]
  PARTITION partition_specs1 [LOCATION 'location_path1']
  PARTITION partition_specs2 [LOCATION 'location_path2']
  ...;
```

Parameters

Parameter	Description
table_name	Table name.
IF NOT EXISTS	Optional keyword. When specified, no error is reported if the partition already exists. It is silently skipped.
partition_specs	Partition specification, in the format **(col1=val1, col2=val2, ...)**. If the table has multiple partition columns, all partition column values must be specified when adding a partition. The order can be arbitrary.
location_path	Optional. The storage path of the partition. If specified, the path must already exist. Otherwise, an error will be reported.

Precautions

- When adding a partition to a table, the table and partition columns (columns specified by **PARTITIONED BY** when creating the table) must already exist. The partition to be added must not be a duplicate. Otherwise, an error will occur. Use **IF NOT EXISTS** to avoid errors for existing partitions.
- If the partitioned table is partitioned by multiple fields, all partition fields must be specified when adding a partition. The order of the specified fields can be arbitrary.
- Parameters in **partition_specs** are enclosed in parentheses by default, for example: **PARTITION (dt='2009-09-09', city='xxx')**.
- If a path is specified when adding a partition, the path must already exist; otherwise, an error will occur.
- When adding multiple partitions, separate each group of **PARTITION partition_specs LOCATION 'location_path'** with a space.
- If the path specified for a new partition contains subdirectories (or nested subdirectories), all file types and contents under the subdirectories will also be considered records of that partition. Ensure that all file types and file contents under the partition directory are consistent with the table fields; otherwise, the query will report an error.

 Copyright © Huawei Technologies Co., Ltd.

Example 1: Add a Single Partition

Add a partition to the table **student** with partition columns **facultyNo=10, classNo=101**.

```
ALTER TABLE student ADD
  PARTITION (facultyNo=10, classNo=101);
```

Example 2: Add Multiple Partitions

Add two partitions to the table **student** simultaneously.

```
ALTER TABLE student ADD
  PARTITION (facultyNo=10, classNo=101)
  PARTITION (facultyNo=20, classNo=102);
```

Example 3: Add a Partition Using IF NOT EXISTS

Use **IF NOT EXISTS** when adding a partition to avoid errors if the partition already exists.

```
ALTER TABLE student ADD IF NOT EXISTS
  PARTITION (facultyNo=10, classNo=101);
```

Example 4: Add a Partition and Specify LOCATION

Add a partition to an OBS table and specify the storage path for the partition data.

```
ALTER TABLE testobstable ADD
  PARTITION (external_data='22')
  LOCATION 'obs://bucketName/datapath/external_data=22';
```

3.7.2 Rename Partition

Function

Modify the partition value of a partition in an OBS partitioned table. This command only supports OBS tables and does not support managed tables.

Syntax

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

Parameters

Parameter	Description
table_name	Table name.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
partition_specs	Partition specification, in the format **(col1=val1, col2=val2, ...)**. The value before **RENAME TO** is the original partition value, and the value after is the new partition value. If the partitioned table has multiple partition columns, all partition column values must be specified. The order can be arbitrary.

Precautions

- This command only supports OBS tables and does not support managed tables.
- The table and partition to be renamed must already exist. Otherwise, an error will occur. The new partition name must not duplicate any existing partition name. Otherwise, an error will occur.
- If the partitioned table is partitioned by multiple fields, all partition fields must be specified when renaming a partition. The order of the specified fields can be arbitrary.
- Parameters in **partition_specs** are enclosed in parentheses by default, for example: **PARTITION (dt='2009-09-09', city='xxx')**.

Example 1: Rename a Single Partition Column

Rename the partition **year=2023** to **year=2024** in the table **log_data**.

```
ALTER TABLE log_data
  PARTITION (year='2023')
  RENAME TO PARTITION (year='2024');
```

Example 2: Rename Multiple Partition Columns

Rename the partition **city='city1', dt='2023-01-01'** to **city='city2', dt='2023-06-01'** in the table **student**.

```
ALTER TABLE student
  PARTITION (city='city1', dt='2023-01-01')
  RENAME TO PARTITION (city='city2', dt='2023-06-01');
```

Example 3: Rename a Date Partition

Rename the date partition **dt='2023-12-31'** to **dt='2024-01-01'** in the table **sales**.

```
ALTER TABLE sales
  PARTITION (dt='2023-12-31')
  RENAME TO PARTITION (dt='2024-01-01');
```

Example 4: Change Partition Value (Modify Only Some Partition Column Values)

Rename the partition **region='east', type='click'` to `region='west', type='click'** in the table **event_log**, modifying only the **region** column value.

 Copyright © Huawei Technologies Co., Ltd.

```
ALTER TABLE event_log
  PARTITION (region='east', type='click')
  RENAME TO PARTITION (region='west', type='click');
```

3.7.3 Drop Partition

Function

Drop one or more partitions from a partitioned table. Both managed tables and OBS tables support using this command to drop partitions. For OBS tables, you can also use the specified filter condition to drop partitions for batch deletion based on conditions.

Syntax

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

Parameters

Parameter	Description
db_name	Optional. Database name, consisting of letters, digits, and underscores (_). It cannot be purely numeric and cannot start with an underscore.
table_name	Table name. Only letters, digits, and underscores (_) are allowed. It cannot be purely numeric and cannot start with an underscore. The matching rule is **^(?!_)(?![0-9]+$)[A-Za-z0-9_$]*$**. Special characters must be enclosed in single quotes (').
IF EXISTS	Optional keyword. When specified, no error is reported if the partition does not exist. It is silently skipped.
partition_spec	Partition specification, in the format **(col1=val1, col2=val2, ...)**. If the partition has multiple fields, not all fields need to be included. All matching partitions will be dropped.

Precautions

- The table from which partitions are to be dropped must already exist. Otherwise, an error will occur.
- The partition to be dropped must already exist. Otherwise, an error will occur. Add **IF EXISTS** to the statement to avoid this error.
- For OBS tables, dropping a partition only deletes the metadata. The data files in the OBS directory are not automatically deleted.

Example 1: Drop a Specific Partition

Drop the partition **facultyNo=20, classNo=103** from the table **student**.

```
ALTER TABLE student DROP
  PARTITION (facultyNo=20, classNo=103);
```

 Copyright © Huawei Technologies Co., Ltd.

Example 2: Drop a Partition Using IF EXISTS

Use **IF EXISTS** to avoid an error if the partition does not exist.

```
ALTER TABLE student DROP IF EXISTS
  PARTITION (facultyNo=99, classNo=999);
```

Example 3: Drop Multiple Partitions Simultaneously

Drop multiple partitions in a single statement.

```
ALTER TABLE student DROP IF EXISTS PARTITION (facultyNo=99, classNo=999), PARTITION (facultyNo=100, classNo=100);
```

Example 4: Drop a Partition from a Table in a Specified Database

Drop a partition from the table **student** in the database **academy**.

```
ALTER TABLE academy.student DROP IF EXISTS
  PARTITION (facultyNo=20, classNo=103);
```

3.7.4 Modify Table Partition Location

Function

Modify the storage location of an OBS table partition. This command only supports OBS tables (external tables) and does not support managed tables.

Syntax

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

Keywords

- **PARTITION**: Specify the partition whose location is to be modified.
- **SET LOCATION**: Set the new storage path for the partition.
- **LOCATION**: Partition path, must be in OBS format.

Parameters

Parameter	Mandatory	Description
table_name	Yes	Table name.
partition_specs	Yes	Partition specification, in the format **(col1=val1, col2=val2, ...)**. The partition must already exist.
obs_path	Yes	New OBS storage path, which must be an existing absolute path.

 Copyright © Huawei Technologies Co., Ltd.

Precautions

- This command only supports OBS tables (external tables) and does not support managed tables.
- The table partition whose location is to be modified must already exist. Otherwise, an error will be reported.
- The specified new OBS path must be an existing absolute path. Otherwise, an error will be reported.
- If the new path contains subdirectories, all file types and contents under the subdirectories will also be considered records of that partition. Ensure consistency with the table fields.
- Modifying the partition location does not automatically migrate existing data.

Example 1: Modify the Partition Location for a Single Partition Column

```
ALTER TABLE student
  PARTITION (dt='2008-08-08')
  SET LOCATION 'obs://my-bucket/student/dt=2008-08-08/';
```

Example 2: Modify the Partition Location for Multiple Partition Columns

```
ALTER TABLE student
  PARTITION (dt='2008-08-08', city='beijing')
  SET LOCATION 'obs://my-bucket/student/dt=2008-08-08/city=beijing/';
```

Example 3: Verify Partition Location After Modification

```
ALTER TABLE student PARTITION (dt='2008-08-08') SET LOCATION 'obs://my-bucket/student/new_path/';

SHOW PARTITIONS student;
```

Example 4: Modify the Partition Location for a Table in a Specified Database

```
ALTER TABLE myschool.student
  PARTITION (class='101')
  SET LOCATION 'obs://my-bucket/myschool/student/class=101/';
```

3.7.5 Update Table Partition Information

Function

Recover (re-register) all partition information in a partitioned table that has not yet been registered in the metadata database. Typical application scenario: After manually adding a partition directory on OBS, use this command to refresh the newly added partition information into the metadata database. The new partitions can then be viewed using **SHOW PARTITIONS**.

Spark SQL provides two equivalent syntaxes: **MSCK REPAIR TABLE** and **ALTER TABLE ... RECOVER PARTITIONS**.

Syntax

- Method 1: MSCK REPAIR TABLE
  ```
  MSCK REPAIR TABLE table_name;
  ```
- Method 2: RECOVER PARTITIONS
  ```
  ALTER TABLE table_name RECOVER PARTITIONS;
  ```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Description
table_name	Table name. The name of the partitioned table. The table must already exist.

Precautions

- The partition directory name must be entered in the specified format, i.e., **tablepath/partition_column_name=partition_column_value**.
- The two syntaxes are functionally equivalent. Either one can be used.
- This command is mainly used for OBS partitioned table scenarios, to refresh metadata after manually adding a partition directory on OBS.

Example 1: Update Partition Information Using MSCK REPAIR TABLE

Update the partition information of table **ptable** in the metadata database.

```
MSCK REPAIR TABLE ptable;
```

Example 2: Update Partition Information Using RECOVER PARTITIONS

Use the equivalent syntax to update the partition information of table **ptable**.

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

Example 3: Add an OBS Directory and Then Repair Partitions

Assuming the partition directory **obs://bucketName/datapath/dt=2024-01-01** has been manually created on OBS, execute the following command to register the new partition in the metadata.

```
MSCK REPAIR TABLE sales_data;
```

Example 4: View the Partition List After Repairing Partitions

After repairing the partition information, use **SHOW PARTITIONS** to verify that the new partition has been registered.

```
-- Repair partitions.
ALTER TABLE sales_data RECOVER PARTITIONS;

-- View all partitions.
SHOW PARTITIONS sales_data;
```

 Copyright © Huawei Technologies Co., Ltd.

4 Import/Export Data Syntax

4.1 Load Data
4.2 Insert Data
4.3 Reuse Subquery
4.4 Truncate Data

4.1 Load Data

Function

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

Usage Restrictions

Datasource tables and Iceberg tables do not support the **LOAD DATA** operation.

Syntax

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

Keywords

- **INPATH**: Data path.
- **OPTIONS**: A list of properties.

Parameters

Parameter	Mandatory	Description
folder_path	Yes	OBS path to the source data folder or file.
db_name	No	Database name. If not specified, the current database is used.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
table_name	Yes	Name of the managed table into which data will be imported.

The following configuration options are available when importing data:

- **DATA_TYPE**: Specifies the data type to import. Currently supports CSV, Parquet, ORC, JSON, and Avro types. The default value is **CSV**.

  Configuration: **OPTIONS ('DATA_TYPE'='CSV')**

  When importing CSV and JSON files, three modes are available:

  - **PERMISSIVE**: When a column's data type does not match the target table column's data type, the entire row is set to null.
  - **DROPMALFORMED**: When a column's data type does not match, the row is not imported.
  - **FAILFAST**: When a column's type does not match, an exception is thrown and the import fails.

  The mode can be set by adding **OPTIONS('MODE'='PERMISSIVE')** to the **OPTIONS** clause.
- **DELIMITER**: Specifies the delimiter for the import command. The default value is **,**.

  Configuration: **OPTIONS('DELIMITER'=',')**

  For CSV data, the following delimiters are supported:

  - Tab character, e.g., **'DELIMITER'='\t'**.
  - Any binary character, e.g., **'DELIMITER'='\u0001(^A)'**.
  - Single quote (`'`), which must be enclosed in double quotes (" "), e.g., **'DELIMITER'= "'"**.
  - Managed tables also support **\001 (^A)** and **\017 (^Q)**, e.g., **'DELIMITER'='\001(^A)'**, **'DELIMITER'='\017(^Q)'**.
- **QUOTECHAR**: Specifies the quote character for the import command. The default value is **"**.

  Configuration: **OPTIONS('QUOTECHAR'='"')**
- **COMMENTCHAR**: Specifies the comment character for the import command. During the import operation, if a comment character is encountered at the beginning of a line, that line is treated as a comment and is not imported. The default value is **#**.

  Configuration: **OPTIONS('COMMENTCHAR'='#')**
- **HEADER**: Indicates whether the source file has a header. Valid values are **true** and **false**. **true** means there is a header, and **false** means there is no header. The default value is **false**. If there is no header, the **FILEHEADER** parameter can be used in the import command to provide one.

  Configuration: **OPTIONS('HEADER'='true')**
- **FILEHEADER**: If the source file does not have a header, a header can be provided in the **LOAD DATA** command.

  OPTIONS('FILEHEADER'='column1,column2')

 Copyright © Huawei Technologies Co., Ltd.

- **ESCAPECHAR**: If strict validation of escape characters on CSV is required, an escape character can be provided. The default value is \\.

  Configuration: **OPTIONS('ESCAPECHAR'='\\')**

  NOTE

  If **ESCAPECHAR** is entered in CSV data, it must be enclosed in double quotes (" "), e.g., **"a\b"**.

- **MAXCOLUMNS**: This optional parameter specifies the maximum number of columns the CSV parser parses in a single row.

  Configuration: **OPTIONS('MAXCOLUMNS'='400')**

  Table 4-1 MAXCOLUMNS parameters

Optional Parameter Name	Default Value	Maximum Value
MAXCOLUMNS	2000	20000

  NOTE

  After setting the **MAXCOLUMNS** option, importing data will impose memory requirements on the executor. Therefore, the import may fail due to insufficient executor memory.

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

  OPTIONS('DATEFORMAT'='dateFormat')

  NOTE

  - The default value is **yyyy-MM-dd**.
  - The date format is specified by Java's date pattern string. In Java's date and time pattern strings, characters **A** to **Z** and **a** to **z** that are not enclosed in single quotes (') are interpreted as pattern characters, representing date or time string elements. If a pattern character is enclosed in single quotes ('), it is treated as literal text during parsing and is not interpreted. Refer to **Table 4-2** for Java pattern character definitions.

  Table 4-2 Date and time pattern character definitions

Pattern Character	Date or Time Element	Example
G	Era designator	AD
y	Year	1996; 96
M	Month in year	July; Jul; 07
w	Week in year	27 (27th week of the year)
W	Week in month	2 (2nd week of the month)

 Copyright © Huawei Technologies Co., Ltd.

Pattern Character	Date or Time Element	Example
D	Day in year	189 (189th day of the year)
d	Day in month	10 (10th day of the month)
u	Day number of week	1 = Monday, ..., 7 = Sunday
a	Am/pm marker	pm (12:00–24:00)
H	Hour in day (0–23)	2
h	Hour in am/pm (1–12)	12
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	978
z	Time zone	Pacific Standard Time; PST; GMT-08:00

- **TIMESTAMPFORMAT**: Specifies the timestamp format for a column.

  OPTIONS('TIMESTAMPFORMAT'='timestampFormat')

  NOTE

  - Default value: **yyyy-MM-dd HH:mm:ss**.
  - The timestamp format is specified by Java's time pattern string. Refer to **Table 4-2** for Java time pattern string definitions.

- **Mode**: Specifies the handling mode for error records during the import process. Three options are supported: **PERMISSIVE**, **DROPMALFORMED**, and **FAILFAST**.

  OPTIONS('MODE'='permissive')

  NOTE

  - **PERMISSIVE (default)**: Parses bad records as much as possible. If a field cannot be converted, the entire row is set to null.
  - **DROPMALFORMED**: Ignores bad records that cannot be parsed.
  - **FAILFAST**: Throws an exception and causes the job to fail when encountering a record that cannot be parsed.

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

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

  NOTE

  When this option is configured, **MODE** cannot be set and is fixed to **DROPMALFORMED**. Records that are successfully converted are imported into the target table, while records that fail conversion are stored in the specified error record storage directory.

 Copyright © Huawei Technologies Co., Ltd.

Precautions

- When importing into an OBS table, the path specified when creating the OBS table must be a folder. If the table creation path is a file, the data import will fail.
- Only raw data located on an OBS path can be imported.
- Concurrent data imports on the same table are not recommended, as there is a certain probability of concurrency conflicts leading to import failure.
- Only one path can be specified when importing data. The path must not contain commas.
- When a folder and a file have the same name in an OBS bucket directory, the data import will prioritize the file under that path over the folder.
- When importing data in PARQUET, ORC, and JSON formats, the **DATA_TYPE** option must be specified. Otherwise, the data will be parsed in the default **CSV** format, resulting in incorrect data formatting.
- When importing CSV and JSON data that includes date and time columns, the **DATEFORMAT** and **TIMESTAMPFORMAT** options must be specified. Otherwise, the default date and timestamp formats will be used for parsing.

Examples

- The following statement can be used to import a CSV file into a managed table named **t**.

```
LOAD DATA INPATH 'obs://aidatalake/data.csv' INTO TABLE t
  OPTIONS('DELIMITER=',' , 'QUOTECHAR='"','COMMENTCHAR='#','HEADER='false');
```

- The following statement can be used to import a JSON file into a managed table named **jsontb**.

```
LOAD DATA INPATH 'obs://aidatalake/alltype.json' into table jsontb
  OPTIONS('DATA_TYPE'='json','DATEFORMAT'='yyyy/MM/dd','TIMESTAMPFORMAT'='yyyy/MM/dd
HH:mm:ss');
```

4.2 Insert Data

Function

Inserts the result of a SELECT query or specific data values into a table.

Notes and Constraints

- The **INSERT OVERWRITE** syntax is not applicable for self-read-write scenarios on the same table (including both partitioned and non-partitioned tables). Executing **INSERT OVERWRITE** directly on the original table may lead to data loss or data inconsistency.

  To perform data operations in a self-read-write scenario, you are advised to use a temporary table to process the data.

  Self-read-write means the destination table and the source data table are the same table. For example, the following statement extracts student information where **class_no = 1** from the **student** table and overwrites the original table. This is a typical example of a self-read-write scenario.

```
INSERT OVERWRITE TABLE student SELECT name FROM student WHERE class_no = 1;
```

 Copyright © Huawei Technologies Co., Ltd.

- When using Hive and Datasource (excluding Hudi) tables to execute data modification commands (e.g., **INSERT INTO**, **LOAD DATA**), because the data source does not support transactions, system failures or endpoint resource restarts may lead to data duplication or data inconsistency.

  To avoid this, you are advised to prioritize data sources that support transactions, such as the Hudi data source. These data sources have atomicity, consistency, isolation, durability (ACID) capabilities, which help ensure data consistency and accuracy.

Syntax

- Insert the result of a SELECT query into a table:

```
INSERT INTO [TABLE] [db_name.]table_name
  [PARTITION part_spec] select_statement;
INSERT OVERWRITE TABLE [db_name.]table_name
  [PARTITION part_spec] select_statement;
part_spec:
   : (part_col_name1=val1 [, part_col_name2=val2, ...])
```

- Insert specific data values into a table:

```
INSERT INTO [TABLE] [db_name.]table_name
  [PARTITION part_spec] VALUES values_row [, values_row ...];
INSERT OVERWRITE TABLE [db_name.]table_name
  [PARTITION part_spec] VALUES values_row [, values_row ...];
values_row:
  : (val1 [, val2, ...])
```

Keywords

Table 4-3 INSERT keywords

Parameter	Description
db_name	The name of the database containing the table on which the INSERT command is to be executed.
table_name	The name of the table on which the INSERT command is to be executed.
part_spec	Specifies detailed partition information. If the partition field has multiple fields, all fields must be included, but the corresponding values may be omitted. The system will match the corresponding partition. The maximum number of partitions per table is 100,000.
select_statement	A SELECT query on the source table (supports managed tables and OBS tables).
values_row	The values to be inserted into the table, with columns separated by commas.

Precautions

- The table must already exist.
- If dynamic partitioning does not require specifying a partition, place **part_spec** as a regular field in the SELECT statement.

 Copyright © Huawei Technologies Co., Ltd.

- The OBS table being inserted into must specify a folder path during table creation.
- The data types and number of columns in the source and target tables should match. Otherwise, the insertion will fail.
- Concurrent data insertion into the same table is not recommended, as concurrency conflicts may lead to abnormal insertion results.
- The **INSERT INTO** command appends the query results to the target table.
- The **INSERT OVERWRITE** command overwrites existing data in the target table.
- **INSERT INTO** commands can be executed in parallel. **INSERT OVERWRITE** commands can only be executed in parallel on partitioned tables when inserting into different static partitions.
- The result of executing **INSERT INTO** and **INSERT OVERWRITE** commands simultaneously is unknown.
- During the process of inserting data from a source table into a target table, data cannot be imported or updated in the source table.
- For dynamic **INSERT OVERWRITE** on Hive partitioned tables, only the data in the affected partitions is overwritten. Overwriting the entire table is not supported.
- To overwrite data in a specified partition of a Datasource table, the parameter **spark.sql.dynamicPartitionOverwrite.enabled** must be set to **true** first, and then the **INSERT OVERWRITE** statement is used. The default value of **spark.sql.dynamicPartitionOverwrite.enabled** is **false**, meaning the entire table is overwritten. For example:
```
insert overwrite table tb1 partition(part1='v1', part2='v2') select * from ...
```
- By configuring the **spark.sql.shuffle.partitions** parameter, the number of files inserted into an OBS bucket for non-managed tables can be set. Additionally, to avoid data skew, **distribute by rand()** can be added after the INSERT statement to increase the concurrency of the processing job. For example:
```
insert into table table_target select * from table_source distribute by cast(rand() * N as int);
```

Examples

NOTE

Before importing data, refer to the examples in "Creating an OBS Table" or "Creating a Managed Table" to create the corresponding tables.

- Example 1: Insert the result of a SELECT query into a table.
  - Create a partitioned table in Parquet format using Datasource syntax:
    ```
    CREATE TABLE data_source_tab1 (col1 INT, p1 INT, p2 INT)
      USING PARQUET PARTITIONED BY (p1, p2);
    ```
  - Insert query results into partition **(p1 = 3, p2 = 4)**:
    ```
    INSERT INTO data_source_tab1 PARTITION (p1 = 3, p2 = 4)
      SELECT id FROM RANGE(1, 3);
    ```
  - Insert new query results into partition **(p1 = 3, p2 = 4)**:
    ```
    INSERT OVERWRITE TABLE data_source_tab1 PARTITION (p1 = 3, p2 = 4)
      SELECT id FROM RANGE(3, 5);
    ```
- Example 2: Insert specific data values into a table.
  - Create a partitioned table in Parquet format using Hive syntax:
    ```
    CREATE TABLE hive_serde_tab1 (col1 INT, p1 INT, p2 INT)
      USING HIVE OPTIONS(fileFormat 'PARQUET') PARTITIONED BY (p1, p2);
    ```

 Copyright © Huawei Technologies Co., Ltd.

- Insert two data rows into partition **(p1 = 3, p2 = 4)**:
```
INSERT INTO hive_serde_tab1 PARTITION (p1 = 3, p2 = 4)
  VALUES (1), (2);
```
- Insert new data into partition **(p1 = 3, p2 = 4)**:
```
INSERT OVERWRITE TABLE hive_serde_tab1 PARTITION (p1 = 3, p2 = 4)
  VALUES (3), (4);
```

How Do I Handle Duplicate Records After Executing INSERT INTO?

- Symptom

When using Hive and Datasource (excluding Hudi) tables to execute data modification commands (e.g., **INSERT INTO**, **LOAD DATA**), because the data source does not support transactions, system failures or endpoint resource restarts may lead to data duplication or data inconsistency.

- Cause Analysis

If an endpoint resource restart occurs during the commit phase of data writing, the data may have already been written to the final directory. If an **INSERT INTO** statement is being executed, a retry triggered after the resource restart may result in duplicate data writes.

- Solution

1. Hudi data sources that support ACID properties are recommended.
2. You are advised to use idempotent syntax like **INSERT OVERWRITE** instead of non-idempotent syntax like **INSERT INTO** for data insertion whenever possible.
3. If strict requirements prevent data duplication, you are advised to perform a deduplication operation on the table data after executing **INSERT INTO** to prevent duplicate data.

4.3 Reuse Subquery

Function

A common table expression (CTE) defines a named temporary result set using a **WITH ... AS** statement. This result set can be reused in different parts of a query, avoiding repeated computation of the same subquery, thereby improving query performance and readability.

Syntax

Single subquery:

```
WITH cte_name AS (
    SELECT ...
    FROM table_name
    WHERE ...
)
SELECT ...
FROM cte_name
WHERE ...;
```

Multiple subqueries:

 Copyright © Huawei Technologies Co., Ltd.

```
WITH cte_name1 AS (
    SELECT ...
    FROM table_name
    WHERE ...
),
cte_name2 AS (
    SELECT ...
    FROM table_name
    WHERE ...
)
SELECT ...
FROM cte_name1, cte_name2
WHERE ...;
```

Parameters

Parameter	Description
cte_name	A user-defined name for the subquery. It must be unique within the same **WITH** statement.
table_name	The name of the table on which the subquery command is executed. The table must already exist.
WITH	The keyword for defining a CTE, followed by one or more named subqueries.
RECURSIVE	The keyword for a recursive CTE, allowing the CTE to reference itself to implement a recursive query.
AS	The keyword used to assign a name to the subquery.

Precautions

- The table being queried must already exist. Otherwise, an error will occur.
- Multiple CTEs are separated by commas (,).
- CTE definitions must precede the main query.
- A CTE is only valid within the **SELECT** statement in which it is defined and cannot be used across different queries.
- A recursive CTE must consist of an anchor query and a recursive query, connected by **UNION ALL**.

Examples

- Example 1: Define a subquery **sales_data** based on the **sales** table to query products with a sales amount greater than 100, and then query all data from this subquery.

  ```
  WITH sales_data AS (
    SELECT product_name, sales_amount
    FROM sales
    WHERE sales_amount > 100
  )
  SELECT * FROM sales_data;
  ```

- Example 2: Define subqueries **sales_data1** and **sales_data2** based on the **sales** table. **sales_data1** queries products with a sales amount greater than 100, and **sales_data2** queries products with a sales amount less than 20. Finally, combine the data from both subqueries.

 Copyright © Huawei Technologies Co., Ltd.

```
WITH sales_data1 AS (
  SELECT product_name, sales_amount
  FROM sales
  WHERE sales_amount > 100
),
sales_data2 AS (
  SELECT product_name, sales_amount
  FROM sales
  WHERE sales_amount < 20
)
SELECT * FROM sales_data1
UNION ALL
SELECT * FROM sales_data2;
```

4.4 Truncate Data

Function

Truncate data from a managed table or an OBS table.

Syntax

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

Keywords

Table 4-4 Keywords

Parameter	Description
tablename	The name of the managed table or OBS table on which the TRUNCATE command is to be executed.
partcol1	The name of the partition in the managed table or OBS table to be truncated.

Precautions

Only supports clearing data from managed tables or OBS tables.

Examples

```
TRUNCATE TABLE test PARTITION (class = 'test');
```

 Copyright © Huawei Technologies Co., Ltd.

5 Export Query Results

Function

INSERT OVERWRITE DIRECTORY is used to write query results directly to a specified directory. It supports storage in CSV, Parquet, ORC, JSON, and Avro formats.

Syntax

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

Keywords

- **USING**: Specifies the storage format.
- **OPTIONS**: A list of properties for the export, optional.

Parameters

Table 5-1 INSERT OVERWRITE DIRECTORY parameter descriptions

Parameter	Description
path	The OBS path to which the query results will be written.
file_format	The file format for writing. Supports CSV, Parquet, ORC, JSON, and Avro formats.

Precautions

- By configuring the **spark.sql.shuffle.partitions** parameter, the number of files inserted into an OBS bucket for non-managed tables can be set. Additionally, to avoid data skew, **distribute by rand()** can be added after the INSERT statement to increase the concurrency of the processing job. For example:
  ```
  insert overwrite directory 'obs://bucket/dir' using csv select * from table_source distribute by cast(rand() * N as int);
  ```

 Copyright © Huawei Technologies Co., Ltd.

- When the configuration is **OPTIONS('DELIMITER'=',')**, the delimiter can be specified. The default value is **,**.

  For CSV data, the following delimiters are supported:

  - Tab character, e.g., **'DELIMITER'='\t'**.
  - Specifying a delimiter using a Unicode encoding, e.g., **'DELIMITER'='\u0001'**.
  - Single quote (`'`), which must be enclosed in double quotes (" "), e.g., **'DELIMITER'= "'"**.

Examples

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

 Copyright © Huawei Technologies Co., Ltd.

6 View Syntax

6.1 Create View
6.2 Drop View
6.3 Alter View
6.4 Show Views

6.1 Create View

Function

Create a view.

Syntax

```
CREATE [OR REPLACE] VIEW view_name
  [WITH SCHEMA { BINDING | COMPENSATION | EVOLUTION | TYPE EVOLUTION }]
  AS select_statement;
```

Keywords

- **CREATE VIEW**: Create a view based on the given SELECT statement. The results of the SELECT statement are not written to disk.
- **OR REPLACE**: When this keyword is specified, no error is raised if the view already exists, and the view definition is updated according to the SELECT statement.
- **WITH SCHEMA**: Specifies the schema management strategy for the view.
- **BINDING**: Strict binding. The view becomes invalid when the underlying table structure changes.
- **COMPENSATION**: Compensation mode (default). Tolerates column type changes.
- **EVOLUTION**: Evolution mode. Automatically follows changes in the underlying table schema.
- **TYPE EVOLUTION**: Type evolution. Only allows column type evolution.

 Copyright © Huawei Technologies Co., Ltd.

Precautions

- The view to be created must not already exist in the current database. Otherwise, an error will occur. When the view exists, the error can be avoided by adding the **OR REPLACE** keyword.
- The information of tables or views included in the view cannot be changed. Any changes may cause query failures.
- If the compute engines used for creating the table and the view are inconsistent, view queries may fail due to incompatible varchar types.

Examples

Query the **id** and **name** data from the **student** table, and create the view **student_view** based on the query results.

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

Create a view with a schema evolution strategy:

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

6.2 Drop View

Function

Drop a view.

Syntax

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

Keywords

DROP: Drop the metadata of the specified view. Although views and tables share many commonalities, **DROP TABLE** cannot be used to drop a view.

Precautions

The view to be dropped must already exist. Otherwise, an error will occur. This error can be avoided by using **IF EXISTS**.

Examples

Drop the view named **student_view**.

```
DROP VIEW student_view;
```

6.3 Alter View

Function

The **ALTER VIEW** statement is used to modify the metadata of a view, including renaming the view, and setting or dropping view properties.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

- Rename a view.

  CAUTION

  Iceberg, Lance, and Hive tables do not support renaming views.

  ```
  ALTER VIEW view_identifier RENAME TO view_identifier
  ```

- Set view properties.
  ```
  ALTER VIEW view_identifier SET TBLPROPERTIES ( property_key = property_val [, ...] )
  ```
- Drop view properties.
  ```
  ALTER VIEW view_identifier UNSET TBLPROPERTIES [ IF EXISTS ] ( property_key [, ...] )
  ```

Parameters

Parameter	Description
view_identifier	The view name, which can be qualified using **database_name.view_name**.
RENAME TO	Rename the view. The source and target databases must be the same. Moving views across databases is not supported.
SET TBLPROPERTIES	Set view properties. If the property key already exists, its value is replaced. If it does not exist, it is added.
UNSET TBLPROPERTIES	Drop view properties. If **IF EXISTS** is specified, no error is raised if the property does not exist.
property_key	The property key name, which can consist of multiple dot-separated parts, e.g., **key_part1.key_part2**.

Examples

Rename a view.

```
ALTER VIEW tempdb1.v1 RENAME TO tempdb1.v2;
```

Set view properties.

```
ALTER VIEW tempdb1.v2 SET TBLPROPERTIES ('created.by.user' = 'John', 'created.date' = '01-01-2001');
```

Drop view properties.

```
-- Drop existing properties.
ALTER VIEW tempdb1.v2 UNSET TBLPROPERTIES ('created.by.user', 'created.date');
-- Use IF EXISTS to avoid an error if the property does not exist.
ALTER VIEW tempdb1.v2 UNSET TBLPROPERTIES IF EXISTS ('non-existent');
```

Precautions

- **ALTER VIEW** only alters the metadata of the view and does not affect the underlying data.

 Copyright © Huawei Technologies Co., Ltd.

- When renaming a view, if the new view name already exists, a **TableAlreadyExistsException** is thrown.
- Renaming a view clears the cached data for that view and its dependencies.
- **ALTER VIEW** does not support **SET SERDE** or **SET SERDEPROPERTIES**.
- **WITH SCHEMA** is not supported for **TEMPORARY** views.

6.4 Show Views

Function

The **SHOW VIEWS** statement is used to list all views in a specified database. View names can be filtered using pattern matching. If no database is specified, views from the current database are listed. If the specified database is the global temporary view database, global temporary views are listed. Note: Local temporary views are always listed, regardless of the database specified.

Syntax

```
SHOW VIEWS [ { FROM | IN } database_name ] [ LIKE regex_pattern ]
```

Parameters

Parameter	Description
FROM / IN database_name	Optional. Specifies the database name for which to view views. If omitted, the current database is used.
LIKE regex_pattern	Optional. Filters view names using a regular expression. Matches zero or more characters.

Examples

List all views in the current database.

```
SHOW VIEWS;
```

Result:

namespace	viewName	isTemporary
default	sam	false
default	sam1	false
default	suj	false
default	temp2	true

 Copyright © Huawei Technologies Co., Ltd.

List views in a specified database.

```
SHOW VIEWS FROM userdb;
```

Result:

namespace	viewName	isTemporary
userdb	user1	false
userdb	user2	false
userdb	temp2	true

List global temporary views.

```
SHOW VIEWS IN global_temp;
```

Result:

namespace	viewName	isTemporary
global_temp	temp1	true
global_temp	temp2	true

Filter views using pattern matching.

```
SHOW VIEWS FROM default LIKE 'sam*';
```

Result:

namespace	viewName	isTemporary
default	sam	false
default	sam1	false

Use multiple patterns for matching.

```
SHOW VIEWS LIKE 'sam|suj|temp*';
```

Result:

namespace	viewName	isTemporary
default	sam	false
default	suj	false
default	temp2	true

 Copyright © Huawei Technologies Co., Ltd.

Precautions

- This statement only displays views, not regular tables. To view all tables, use **SHOW TABLES**.
- Local temporary views always appear in the results, regardless of the database specified.
- The ***** in the regex pattern is equivalent to **%** in SQL, and **|** represents logical OR.
- If the database does not exist, an error is raised.

 Copyright © Huawei Technologies Co., Ltd.

7 Materialized Views

7.1 Use Cases
7.2 Configuration Parameters
7.3 Regular Materialized Views
7.4 Intelligent Materialized Views

7.1 Use Cases

Materialized views physically store the results of a SQL query, allowing future queries to read the precomputed results directly instead of recomputing the same logic each time. This can significantly improve query performance, especially for frequently run queries, particularly those involving joins and aggregate operations.

When materialized views are enabled, the optimizer automatically rewrites eligible queries to use the materialized view instead of scanning the base tables.

Spark materialized views support the following use cases:

- Aggregation scenarios: Materialized views precompute and store aggregation results, such as SUM, COUNT, and AVG.

  Example of creating a materialized view:

  ```
  CREATE MATERIALIZED VIEW mv AS SELECT deptno, SUM(salary) AS s FROM emps WHERE deptno > 5 GROUP BY deptno;
  ```

  Original query statement:

  ```
  SELECT deptno FROM emps
  WHERE deptno > 10 GROUP BY deptno
  ```

  Rewritten query statement:

  ```
  SELECT deptno FROM mv
  WHERE deptno > 10
  GROUP BY deptno
  ```

- Join scenarios: The query's table join method matches the materialized view, or the materialized view can compensate for the join method.

  Example of creating a materialized view:

  ```
  CREATE MATERIALIZED VIEW mv AS
  SELECT o.order_id, c.customer_id
  FROM orders o INNER JOIN customers c ON o.cust_id = c.customer_id;
  ```

 Copyright © Huawei Technologies Co., Ltd.

Original query statement:

```
SELECT order_id FROM orders INNER JOIN customers
ON orders.cust_id = customers.customer_id;
```

Rewritten query statement:

```
SELECT order_id FROM mv
```

- Predicate/Filter scenarios: If a query contains filter conditions that can be pushed down, the materialized view can precompute and store the results matching those conditions.

  Example of creating a materialized view:

  ```
  CREATE MATERIALIZED VIEW mv AS
  SELECT * FROM sales WHERE region = 'ASIA';
  ```

  Original query statement:

  ```
  SELECT customer_id
  FROM sales WHERE region = 'ASIA' AND revenue > 1000;
  ```

  Rewritten query statement:

  ```
  SELECT customer_id from mv WHERE revenue > 1000;
  ```

  The optimizer can use the materialized view and apply additional filter conditions to filter data where **revenue > 1000**.

- Aggregate Rollup scenarios: When a materialized view contains finer-grained pre-aggregated data, the query can perform a roll up.

  Original query statement:

  ```
  SELECT deptno, COUNT(*) AS c, SUM(salary) AS s
  FROM emps
  GROUP BY deptno
  ```

  Example of creating a materialized view:

  ```
  CREATE MATERIALIZED VIEW mv AS
  SELECT empid, deptno, COUNT(*) AS c, SUM(salary) AS s
  FROM emps
  GROUP BY empid, deptno
  ```

  Rewritten query statement:

  ```
  SELECT deptno, SUM(c), SUM(s)
  FROM mv
  GROUP BY deptno
  ```

- Partial join scenarios: The joined tables in the query are a superset of the joined tables in the materialized view.

  Example 1

  Example of creating a materialized view:

  ```
  CREATE MATERIALIZED VIEW join_mv1 AS
  SELECT lo_orderkey, lo_linenumber, lo_revenue, lo_partkey, c_name, c_address
  FROM lineorder INNER JOIN customer
  ON lo_custkey = c_custkey;
  ```

  Original query statement:

  ```
  SELECT lo_orderkey, lo_linenumber, lo_revenue, c_name, c_address, p_name
  FROM lineorder INNER JOIN customer ON lo_custkey = c_custkey
  INNER JOIN part ON lo_partkey = p_partkey;
  ```

  Rewritten query statement:

  ```
  SELECT lo_orderkey, lo_linenumber, lo_revenue, c_name, c_address, p_name
  FROM join_mv1 INNER JOIN part ON lo_partkey = p_partkey;
  ```

  Example 2

  Example of creating a materialized view:

  ```
  CREATE MATERIALIZED VIEW mv AS
  SELECT empid, deptno, state, SUM(salary) AS s
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
FROM emps
JOIN locations ON emps.locationid = locations.locationid
GROUP BY empid, deptno, state
```

Original query statement:

```
SELECT deptname, state, SUM(salary) AS s
FROM emps
JOIN depts ON emps.deptno = depts.deptno
JOIN locations ON emps.locationid = locations.locationid
GROUP BY deptname, state
```

Rewritten query statement:

```
SELECT deptname, state, SUM(s)
FROM mv
JOIN depts ON mv.deptno = depts.deptno
GROUP BY deptname, state;
```

- Expression scenarios: Rewrite join queries with complex expressions (e.g., arithmetic operations, string functions, and OR predicates).

  Example of creating a materialized view:

  ```
  CREATE MATERIALIZED VIEW join_mv1
  AS
  SELECT lo_orderkey, lo_linenumber, lo_revenue, lo_partkey, c_name, c_address
  FROM lineorder INNER JOIN customer
  ON lo_custkey = c_custkey;
  ```

  Original query statement:

  ```
  SELECT
  lo_orderkey,
  lo_linenumber,
  (2 * lo_revenue + 1) * lo_linenumber,
  upper(c_name),
  substr(c_address, 3)
  FROM lineorder INNER JOIN customer
  ON lo_custkey = c_custkey;
  ```

- Join transitivity scenarios: This refers to a scenario where the join type in the materialized view differs from the join type in the query, but the join result of the materialized view contains the join result of the query.

  Example 1

  Original query statement:

  ```
  SELECT lo_orderkey, lo_linenumber, lo_revenue, c_custkey, c_address
  FROM lineorder LEFT OUTER JOIN customer
  ON lo_custkey = c_custkey
  WHERE customer.c_address = 'Sb4gxKs7' and customer.c_address is not null;
  ```

  Example of creating a materialized view:

  ```
  CREATE MATERIALIZED VIEW join_mv4
  AS
  SELECT lo_orderkey, lo_linenumber, lo_revenue, c_custkey, c_address
  FROM lineorder INNER JOIN customer
  ON lo_custkey = c_custkey;
  ```

  Example 2

  Original query statement:

  ```
  SELECT lo_orderkey, lo_linenumber, lo_revenue, c_custkey, c_address
  FROM lineorder INNER JOIN customer
  ON lo_custkey = c_custkey WHERE  c_custkey IS NOT NULL;
  ```

  Example of creating a materialized view:

  ```
  CREATE MATERIALIZED VIEW join_mv3
  AS
  SELECT lo_orderkey, lo_linenumber, lo_revenue, c_custkey, c_address
  FROM lineorder LEFT OUTER JOIN customer
  ON lo_custkey = c_custkey WHERE c_custkey IS NOT NULL;
  ```

 Copyright © Huawei Technologies Co., Ltd.

For exact rewrite matching, the following string transformation functions, aggregation functions, and join types are supported:

- **Aggregation functions**: SUM, COUNT, MIN, MAX, AVG, STDDEV_SAMP, and STDDEV_POP
- **String transformation functions:** LOWER, UPPER, SUBSTR, CONCAT, and CONCAT_WS
- **Join Types**: INNER, LEFT, RIGHT, and FULL

7.2 Configuration Parameters

Parameter	Description	Default Value
spark.sql.materialized.views.location	Specifies the storage location for regular materialized views. Does not apply to intelligent materialized views. If this parameter is not specified, Spark uses the default storage location.	-
spark.sql.materialized.views.rewrite.enabled	Controls whether materialized view query rewriting is enabled. Query rewriting means that when executing a query, Spark attempts to rewrite the query to use a materialized view to improve query performance. • **true**: Enables materialized view query rewriting. • **false**: Does not attempt to rewrite queries to use materialized views.	false
spark.sql.materialized.views.calcite.timeoutSeconds	Controls the maximum timeout for the Calcite planner during query rewriting, in seconds. If the Calcite planner does not complete query rewriting within the specified time, Spark abandons using the materialized view and executes the query as-is.	10

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
spark.sql.cte.mv.enabled	Eligible CTE definitions will be materialized asynchronously after query execution. If a structurally identical CTE body query appears later, it will be replaced by a simple scan of the materialized view instead of recomputation. (Session-level enabling/disabling is currently not supported but will be in the future.) ● **true**: Enable CTE materialization. ● **false**: Disable CTE materialization.	false
spark.sql.cte.mv.viewPrefix	The prefix for temporary view names created by the CTE materialization feature.	cte
spark.sql.cte.mv.database	The database name for storing CTE materialized views.	default
spark.sql.materialized.views.cte.hash.match	Controls whether hash matching mode is enabled for materialized view query rewriting. Hash matching mode can hit materialized views faster, but the disadvantage is that the rewritten SQL must exactly match the logical execution plan of the materialized view. Otherwise, it cannot be hit. ● **true**: Enable hash matching mode. ● **false**: Disable hash matching mode.	false

7.3 Regular Materialized Views

7.3.1 Syntax Reference

- Create a regular materialized view.

  Create a materialized view based on a SQL query.

  ```
  CREATE MATERIALIZED VIEW [IF NOT EXISTS] view_name
  [(<col_name> [COMMENT <col_comment>], ...)]
  [COMMENT '...']
  [PARTITIONED BY col1, col2, ...]
  [ CLUSTERED BY (col_name3, col_name4, ...)
  [ SORTED BY (col_name [ ASC | DESC], ...)]
  INTO num_buckets BUCKETS]
  [TBLPROPERTIES ('key'='value', ...)]
  AS <SELECT query>;
  ```

  Examples:

 Copyright © Huawei Technologies Co., Ltd.

```
CREATE MATERIALIZED VIEW mv_partitioned
PARTITIONED BY (region, product)
AS SELECT region, product, SUM(amount) AS total
FROM sales GROUP BY region, product
CREATE MATERIALIZED VIEW mv_sorted_bucketed
CLUSTERED BY (id) SORTED BY (amount ASC) INTO 2 BUCKETS
AS SELECT * FROM sales
CREATE MATERIALIZED VIEW mv_sales_with_product AS
SELECT s.region, p.product_name, SUM(s.amount) AS total_amount
FROM sales s
INNER JOIN products p ON s.product_id = p.product_id
GROUP BY s.region, p.product_name
```

- Delete a materialized view.

  ```
  DROP MATERIALIZED VIEW [IF EXISTS] mv_revenue_per_customer;
  ```

  Example:

  ```
  DROP MATERIALIZED VIEW mv_sales
  ```

- Query materialized views.

  Query all materialized views in the current database.

  ```
  SHOW MATERIALIZED VIEWS;
  ```

  Query all materialized views in a specific database.

  ```
  SHOW MATERIALIZED VIEWS FROM example_db;
  ```

  Query materialized views matching a specific pattern.

  ```
  SHOW MATERIALIZED VIEWS LIKE 'mv_*';
  ```

- Refresh a materialized view.

  Recompute the data of the materialized view from the base tables.

  ```
  REFRESH MATERIALIZED VIEW mv_revenue_per_customer;
  ```

- Alter materialized view properties.

  ```
  ALTER MATERIALIZED VIEW mv_name SET TBLPROPERTIES ('key1' = 'val1');
  ```

- Drop materialized view properties.

  ```
  ALTER MATERIALIZED VIEW mv_name UNSET TBLPROPERTIES ('key1' = 'val1');
  ```

  The following properties cannot be altered using the ALTER command:

  - **mv.archived**: Whether the view is archived.
  - **mv.base.table.valid**: Whether the view can be used for SQL rewrite.
  - **mv.base.tables**: Base tables of the materialized view.
  - **mv.base.table.names**: Materialized view.
  - **mv.select.statement.spark**: SELECT statement corresponding to this materialized view in the Spark engine.
  - **mv.base.tables.last.update.time**: Last refresh time of the base tables of the materialized view.
  - **mv.last.refresh.time**: Last refresh time of the materialized view.
  - **viewQuery**: Query statement of the materialized view.
  - **spark.mv.type**: Type of the materialized view.

  The following property can be altered using the ALTER command:

  mv.refresh.policy: Refresh policy of the materialized view.

  Use the following syntax to alter the refresh policy:

  ```
  ALTER MATERIALIZED VIEW mv_altr ADD SCHEDULE REFRESH EVERY 3 DAYS
  ALTER MATERIALIZED VIEW mv_altr ALTER SCHEDULE REFRESH EVERY 2 DAYS
  ALTER MATERIALIZED VIEW mv_altr DROP SCHEDULE;
  ```

 Copyright © Huawei Technologies Co., Ltd.

7.3.2 Permission Management

Materialized views can be considered a special type of table for independent authorization management in LakeFormation, including permissions for reading, deleting, and modifying.

- Create a view

  Requires **CREATE_TABLE** and **DESCRIBE** permissions on the database, plus **SELECT**, **DESCRIBE**, and **ALTER** permissions on all source tables.

- Query a view

  A view can be considered a special table type. The view creator automatically has all permissions on the view. For other users to query the view, **SELECT** and **DESCRIBE** permissions must be granted separately in LakeFormation.

- Drop a view

  The view owner automatically has delete permission on the view. Other users who need to delete the view must be granted **DESCRIBE** and **DROP** permissions separately.

- Refresh a view

  The view owner automatically has refresh permission on the view. Other users who need to refresh the view must be granted **INSERT**, **UPDATE**, **DELETE**, **DESCRIBE**, and **ALTER** permissions separately.

- Automatic rewriting

  Requires **SELECT** and **DESCRIBE** permissions on all source tables, plus **SELECT** and **DESCRIBE** permissions on the materialized view.

7.3.3 Automatic Rewriting

Automatic rewriting of materialized views can match the execution plan during the logical plan generation phase and automatically rewrite it into a materialized view query statement.

- Automatic rewriting authorization

  During automatic rewriting, permissions on all source tables and the materialized view must be verified simultaneously for the rewriting to occur.

- Exact match (hash match)

  The hash value of the logical execution plan is stored in the table properties. The hash value is directly used for comparison to determine if there is a match. If a match is found, the query statement is rewritten.

- Fuzzy match (plan match)

  The view's SQL statement is retrieved and re-parsed to generate a logical plan. All nodes of the execution plan are compared in detail in the background. If the execution plan is a subtree of the materialized view's execution plan, a match is found, and the query statement is rewritten.

  Compared to exact match, fuzzy match can cover a wider range. Even if the execution plans are not identical, there is a possibility of a match.

 Copyright © Huawei Technologies Co., Ltd.

7.3.4 Metadata

When a materialized view is created, some new parameters are added to the table properties of the source tables and the materialized view to record information about the associated materialized views and source tables.

Table 7-1

Property	Type	Description
mv.archived	Boolean	Whether the view is archived. Materialized views allow creating only the table schema without populating data. Data can be filled in later during a refresh.
mv.base.table.valid	Boolean	Whether the view is valid. When the view is invalid, automatic rewriting is not allowed.
mv.base.tables	Array	Stores the source tables associated with the view.
mv.select.statement.spark	String	Stores the SELECT statement for Spark materialized views.

7.3.5 Notes and Constraints

- Mode

  Currently, only the Spark-SQL mode is supported. The Spark-JOB mode is not supported at this time.
- Supported source table types

  MANAGED_TABLE , EXTERNAL_TABLE
- Table formats
  - Managed materialized views are created as Iceberg tables.
  - View source tables can support all available formats in AI DataLake, such as Hive, Datasource, and Iceberg.
- Refresh strategy

  Only full view refresh is supported. Incremental refresh is not supported.
- Use cases
  - Queries containing non-deterministic functions, such as rand, random, and uuid, cannot be rewritten.
  - A materialized view cannot be used as a source table in the statement for creating a new materialized view.
  - Queries containing window functions cannot be rewritten.
  - Queries containing GROUPING SETS, WITH CUBE, or WITH ROLLUP cannot be rewritten.
  - Queries containing scalar subqueries or correlated subqueries that reference outer columns cannot be rewritten.

 Copyright © Huawei Technologies Co., Ltd.

7.4 Intelligent Materialized Views

Intelligent materialized views provide more advanced features, including more types of refresh strategies, more flexible refresh schedules, and CTE materialization.

7.4.1 Syntax Reference

- Create an intelligent materialized view.

  Create a materialized view based on a SQL query.

  ```
  CREATE LAKE MATERIALIZED VIEW [IF NOT EXISTS] view_name
  [(<col_name> [COMMENT <col_comment>], ...)]
  [COMMENT '...']
  [PARTITIONED BY col1, col2, ...]
  [ CLUSTERED BY (col_name3, col_name4, ...)
  [ SORTED BY (col_name [ ASC | DESC], ...)]
  INTO num_buckets BUCKETS]
  [TBLPROPERTIES ('key'='value', ...)]
  REFRESH POLICY AUTO | INCREMENTAL | INCREMENTAL STRICT | FULL
  SCHEDULE EVERY  MINUTES| HOURS | DAYS
  AS <SELECT query>;
  ```

  REFRESH POLICY AUTO | INCREMENTAL | INCREMENTAL STRICT | FULL specifies the refresh strategy.

  SCHEDULE EVERY MINUTES| HOURS | DAYS specifies the refresh schedule. Parameter limits are as follows:

  - MINUTES: 1 <= M <= 60
  - HOURS: 1 <= H <= 72
  - DAYS: 1 <= D <= 7

7.4.2 Permission Management

- Create a view.

  The creation location is within the LakeFormation internal OBS bucket. Requires **CREATE_TABLE** and **DESCRIBE** permissions on the database, plus **SELECT**, **DESCRIBE**, and **ALTER** permissions on all source tables.

- Query a view.

  For user convenience, LakeFormation enables permission cascade authorization by default for intelligent materialized views in the background.

  If a user has **SELECT** and **DESCRIBE** permissions on the source tables of a materialized view, when a new view is created by another user, LakeFormation automatically grants the **SELECT** and **DESCRIBE** permissions on the view to that user in the background.

- Drop a view.

  The view owner automatically has delete permission on the view. Other users who need to delete the view must be granted **DESCRIBE** and **DROP** permissions separately.

- Refresh a view.

  The view owner automatically has refresh permission on the view. Other users who need to refresh the view must be granted **INSERT**, **UPDATE**, **DELETE**, **DESCRIBE**, and **ALTER** permissions separately.

 Copyright © Huawei Technologies Co., Ltd.

Intelligent materialized views support scheduled background refresh by LakeFormation. Therefore, refresh permissions need to be granted to the LakeFormation internal user.

- Automatic rewriting

  Requires **SELECT** and **DESCRIBE** permissions on all source tables, plus **SELECT** and **DESCRIBE** permissions on the materialized view.

7.4.3 View Refresh

7.4.3.1 Refresh Strategies

- Incremental refresh

  Incremental refresh allows materialized views to be updated **by processing only the data that has changed in the source tables**, rather than recomputing the entire view. This significantly reduces refresh time and resource consumption for large datasets.

  Using incremental refresh requires that the source tables are Apache Iceberg tables with the Iceberg snapshot feature enabled.

- Restrictions on incremental refresh
  - Supported aggregate functions: SUM, COUNT(*), COUNT(col), MIN, and MAX
  - Supported joins: INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN
  - Supported queries: Simple SELECT (Projection), GROUP BY Aggregation, UNION ALL, and JOIN + Aggregation
  - Unsupported scenarios: The source table is not an Iceberg table, or non-equi JOINs are used.
- Four refresh strategies

  As shown in the table below, intelligent materialized views support four refresh strategies. The strategy can be specified using the **REFRESH POLICY** keyword in the **CREATE** statement. If not specified, the default value is **AUTO**. The strategy can also be modified after view creation using an **ALTER** statement.

Refresh Strategy	Description
INCREMENTAL	Attempts incremental refresh first. Falls back to full refresh if not supported.
INCREMENTAL STRICT	Uses only incremental refresh. Reports an error or fails if not supported.
FULL	Always performs a full refresh (recomputes the entire view).
AUTO (Default)	Prefers incremental refresh. Falls back to full refresh if incremental conditions are not met.

 Copyright © Huawei Technologies Co., Ltd.

7.4.3.2 Refresh Schedules

Intelligent materialized views offer multiple refresh schedules for user selection.

- Manual refresh

  A user executes the **REFRESH MATERIALIZED VIEW mv_name** command.

- Scheduled refresh

  The refresh interval field is set in the table properties, and a scheduled task is set up by the LakeFormation backend.

  Users are allowed to execute **ALTER** commands to modify the refresh interval field. The Spark kernel calls the LakeFormation backend interface to modify the period of the scheduled task.

- Real-time refresh

  LakeFormation monitors the freshness of the view. Once it detects that the view is not fresh, it triggers an asynchronous refresh task in the background.

7.4.3.3 View Metadata

Property	Type	Description
mv.base.tables.last.update.time	Long	The UTC timestamp of the last update to the source tables.
mv.last.refresh.time	Long	The UTC timestamp of the last refresh of the materialized view. The type is long. The default value is the creation time of the materialized view.
mv.refresh.policy	String	Enumerated values: **FULL**, **INCREMENTAL**, **INCREMENTAL STRICT**, and **AUTO**.
mv.refresh.interval	Integer	Unit: minutes. The minimum value is 1 minute, and the maximum value is 10,080 minutes (7 days).

7.4.4 CTE Materialization

In some cases, users may execute repeated queries containing the same CTE subquery. Additionally, different users may be unaware that they are using the same computation logic. To improve user computing efficiency, reduce redundant computation, and accelerate queries, AI DataLake provides the CTE materialization feature, which can automatically create materialized views for CTEs.

- Enabling CTE materialization

  After enabling this feature in a SQL statement, the system will perform the following operations:

  a. CTE subquery identification and compliance check: Identifies CTE subqueries in the query and extracts only those that meet the criteria for materialization. Refer to section "Limitations" for specific qualifying criteria.

 Copyright © Huawei Technologies Co., Ltd.

b. Automatic creation of materialized views: Based on the identification results from the first step, materializes the qualifying CTE subqueries.

c. Query rewriting: Uses the query rewriting function of the CTE MV to rewrite the CTE subquery to read data directly from the CTE materialized view table created above.

7.4.5 Notes and Constraints

- Mode

  Currently, only the Spark-SQL mode is supported. The Spark-JOB mode is not supported at this time.

- Supported source table types

  MANAGED_TABLE

- Table formats
  - Lake materialized views are created as Iceberg tables.
  - View source tables only support Iceberg tables.
- Refresh strategy

  All four refresh strategies are supported.

- Use cases
  - Queries containing non-deterministic functions, such as rand, random, and uuid, cannot be rewritten.
  - A materialized view cannot be used as a source table in the statement for creating a new materialized view.
  - Queries containing window functions cannot be rewritten.
  - Queries containing GROUPING SETS, WITH CUBE, or WITH ROLLUP cannot be rewritten.
  - Queries containing scalar subqueries or correlated subqueries that reference outer columns cannot be rewritten.
  - CTE self-reference is not supported.

 Copyright © Huawei Technologies Co., Ltd.

8 View Plan

Function

Executing this statement returns the physical execution plan for the specified SQL statement. When the **EXTENDED** keyword is specified, both the logical plan and the physical execution plan are output.

Syntax

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

Keywords

EXTENDED: When this keyword is specified, both the logical plan and the physical execution plan are output.

CODEGEN: When this keyword is specified, code generated by codegen is also output, if any.

Precautions

None.

Example

Return the logical plan and physical execution plan for the SQL statement **SELECT * FROM test**.

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

 Copyright © Huawei Technologies Co., Ltd.

9 Data Permissions

9.1 Data Permission List
9.2 Create Role
9.3 Show Roles
9.4 Delete Role
9.5 Bind Role
9.6 Unbind Role
9.7 Grant Permissions
9.8 Revoke Permissions
9.9 Show Granted Permissions
9.10 Show All Role and User Bindings

9.1 Data Permission List

Permission Matrix

Table 9-1 and **Table 9-2** describe the permission matrix related to SQL statements, databases, tables, and roles in the platform.

Table 9-1 Supported permission matrix

Authorization Type	LakeFormation Permission Type	Description
database	ALL	All operation permissions on the database.
	ALTER	Modify the database.
	DROP	Delete the database.

 Copyright © Huawei Technologies Co., Ltd.

Authorization Type	LakeFormation Permission Type	Description
	DESCRIBE	View database metadata information or switch databases.
	LIST_TABLE	View the list of resources under the database.
	LIST_FUNC	View functions under a specific database.
	CREATE_TABLE	Create a table in the database.
	CREATE_FUNC	Create a function in the database.
table	ALL	All operation permissions on the table.
	ALTER	Modify the table.
	DROP	Delete the table.
	DESCRIBE	View table metadata information.
	UPDATE	Update table data.
	INSERT	Insert table data.
	SELECT	Query data in the table.
	DELETE	Delete data from the table.
column	SELECT	Query column data in the table.
function	ALL	All operation permissions on the function.
	ALTER	Modify the function.
	DROP	Delete the function.
	DESCRIBE	View function metadata information.
	EXEC	Execute the function.

Table 9-2 Common SQL permission descriptions

Category	SQL Statement	Required Permission
Database	DROP DATABASE db1	**DROP** permission on **db1**.
	CREATE TABLE tb1(...)	**CREATE_TABLE** permission under **db1**.
	CREATE VIEW v1	**CREATE_TABLE** permission under **db1**.
Table	SHOW CREATE TABLE tb1	**DESCRIBE** permission on **tb1**.

 Copyright © Huawei Technologies Co., Ltd.

Category	SQL Statement	Required Permission
	DESCRIBE [EXTENDED| FORMATTED] tb1	**DESCRIBE** permission on **tb1**.
	DROP TABLE [IF EXISTS] tb1	**DROP** permission on **tb1**.
	SELECT count(*) FROM tb1	**SELECT** permission on **tb1**.
	SELECT * FROM view1	**SELECT** permission on **view1**.
	SELECT count(*) FROM view1	**SELECT** permission on **view1**.
	LOAD DATA INPATH ... INTO TABLE	**INSERT** permission on **tb1**.
	INSERT INTO TABLE	**INSERT** permission on **tb1**.
	INSERT OVERWRITE TABLE	**INSERT** permission on **tb1**.
	ALTER TABLE ADD COLUMNS	**ALTER** permission on **tb1**.
	ALTER TABLE RENAME	**ALTER** permission on **tb1**.
column	SELECT user_id FROM db.users;	**SELECT** permission on the **user_id** column in the **db.users** table.
function	ALTER FUNCTION db.calc_tax RENAME TO db.calc_tax_v2;	**ALTER** permission on the **calc_tax** function under **db**.
function	DROP FUNCTION IF EXISTS db.calc_tax;	**DROP** permission on the **calc_tax** function under **db**.
function	DESCRIBE FUNCTION EXTENDED db.calc_tax;	**DESCRIBE** permission on the **calc_tax** function under **db**.
function	SELECT db.calc_tax("xxx");	**EXEC** permission on the **calc_tax** function under **db**.

Precautions

- The essence of permission operations is that they are implemented through the platform-managed APIs, rather than by directly executing **GRANT**/ **REVOKE** statements in the Spark SQL engine.
- Tables and views under the same database share a permission namespace.

 Copyright © Huawei Technologies Co., Ltd.

- Users with **GRANT_PRIVILEGE** permission can grant permissions to other users.
- Users with **REVOKE_PRIVILEGE** permission can revoke permissions from other users.

9.2 Create Role

Function

- Creates a new role.
- Only users with **CREATE_ROLE** permission on the database can create roles. For example: administrator users, database owner users, and other users who have been granted the **CREATE_ROLE** permission.
- Each role must belong to only one database.

Syntax

```
CREATE ROLE role_name;
```

Keywords

None.

Examples

```
CREATE ROLE role1;
```

9.3 Show Roles

Function

Displays all roles or display the roles bound to a specified **user_name** in the current database.

Syntax

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

Parameters

ALL: Displays all roles.

Precautions

The **ALL** keyword and **user_name** cannot be used simultaneously.

Examples

- Display all roles under the project.
  ```
  SHOW ALL ROLES;
  ```

 Copyright © Huawei Technologies Co., Ltd.

NOTE

Only administrators have permission to execute the **SHOW ALL ROLES** statement.

- Display all roles bound to the user named **user_name1**.
  ```
  SHOW ROLES user_name1;
  ```

9.4 Delete Role

Function

Deletes a role.

Syntax

```
DROP ROLE role_name;
```

Keywords

None.

Example

```
DROP ROLE role1;
```

9.5 Bind Role

Function

Binds a user and a role.

Syntax

```
GRANT (role_name,...) TO (user_name,...);
```

Keywords

None.

Precautions

The **role_name** and **user_name** must exist. Otherwise, an error is reported.

Example

```
GRANT role1 TO user_name1;
```

9.6 Unbind Role

Function

Cancels the binding between a user and a role.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
REVOKE (role_name,...) FROM (user_name,...);
```

Keywords

None.

Precautions

The **role_name** and **user_name** must exist, and the **user_name** must be bound to the **role_name**.

Example

Cancel the binding between user **user_name1** and role **role1**.

```
REVOKE role1 FROM user_name1;
```

9.7 Grant Permissions

Function

Grants permissions to a user or role.

Syntax

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

Keywords

- **ROLE**: Specifies that the following **role_name** is a role.
- **USER**: Specifies that the following **user_name** is a user.

Precautions

- **privilege** must be one of the grantable permissions. If the grantee already has the corresponding permission on the **resource** or a parent **resource**, the grant fails. For supported privilege types, see **9.1 Data Permission List**.
- **resource** can be a database, table, view, column, or function. The formats are as follows:
  - Database format: **databases.db_name**.

    For supported privilege types for databases, see **9.1 Data Permission List**.
  - Table format: **databases.db_name.tables.table_name**.

    For supported privilege types for tables, see **9.1 Data Permission List**.
  - View format: **databases.db_name.tables.view_name**.

    Views support the same privilege types as tables. For details, refer to the table permission list description in **9.1 Data Permission List**.
  - Column format: **databases.db_name.tables.table_name.columns.column_name**.

 Copyright © Huawei Technologies Co., Ltd.

The only supported privilege type for columns is **SELECT**.

- Function format: **databases.db_name.functions.function_name**.

  For supported privilege types for functions, see **9.1 Data Permission List**.

Examples

- Grant the DROP database permission on database **db1** to user **user_name1**.
  ```
  GRANT DROP ON databases.db1 TO USER user_name1;
  ```
- Grant the SELECT permission on table **tb1** in database **db1** to user **user_name1**.
  ```
  GRANT SELECT ON databases.db1.tables.tb1 TO USER user_name1;
  ```
- Grant the SELECT permission on table **tb1** in database **db1** to role **role_name**.
  ```
  GRANT SELECT ON databases.db1.tables.tb1 TO ROLE role_name;
  ```

9.8 Revoke Permissions

Function

Revokes permissions that have been granted to a user or role.

Syntax

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

Keywords

- **ROLE**: Specifies that the following **role_name** is a role.
- **USER**: Specifies that the following **user_name** is a user.

Precautions

- **privilege** must be a permission that has been granted to the grantee on the **resource**. Otherwise, the revocation fails. For supported privilege types, see **9.1 Data Permission List**.
- **resource** can be a database, table, view, column, or function. The formats are as follows:
  - Database format: **databases.db_name**.
  - Table format: **databases.db_name.tables.table_name**.
  - View format: **databases.db_name.tables.view_name**.
  - Column format: **databases.db_name.tables.table_name.columns.column_name**.
  - Function format: **databases.db_name.functions.function_name**.

Examples

- Revoke the DROP database permission on database **db1** from user **user_name1**.
  ```
  REVOKE DROP ON databases.db1 FROM USER user_name1;
  ```
- Revoke the SELECT permission on table **tb1** in database **db1** from user **user_name1**.

 Copyright © Huawei Technologies Co., Ltd.

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

- Revoke the SELECT permission on table **tb1** in database **db1** from role **role_name**.

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

9.9 Show Granted Permissions

Function

Displays the permissions that have been granted to a user on a resource.

Syntax

```
SHOW GRANT USER user_name ON resource;
```

Keywords

USER: Specifies that the following **user_name** is a user.

Precautions

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

- Database format: **databases.db_name**.
- Table format: **databases.db_name.tables.table_name**.
- Column format: **databases.db_name.tables.table_name.columns.column_name**.
- View format: **databases.db_name.tables.view_name**.
- Function format: **databases.db_name.functions.function_name**.

Example

Display the permissions of user **user_name1** on database **db1**.

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

9.10 Show All Role and User Bindings

Function

Displays all users bound to the specified role.

Syntax

```
SHOW PRINCIPALS ROLE;
```

Keywords

None.

 Copyright © Huawei Technologies Co., Ltd.

Precautions

The role name must exist.

Example

```
SHOW PRINCIPALS role1;
```

10 Data Types

10.1 Overview

10.2 Primitive Data Types

10.3 Complex Data Types

10.1 Overview

A data type is a fundamental attribute of data, used to distinguish between different types of data. Different data types occupy different storage spaces and support different operations.

Data in a database is stored in tables. Each column in a table defines a data type. When storing data, users must comply with the attributes of these data types. Otherwise, errors may occur.

10.2 Primitive Data Types

Table 10-1 lists primitive data types that AI DataLake supports.

Table 10-1 Primitive data types

Data Type	Description	Storage Space	Value Range	OBS Table Support	Managed Table Support
INT	Signed integer	4 bytes	–2147483648 to 2147483647	Yes	Yes
STRING	String	-	-	Yes	Yes

 Copyright © Huawei Technologies Co., Ltd.

Data Type	Description	Storage Space	Value Range	OBS Table Support	Managed Table Support
FLOAT	Single-precision floating point	4 bytes	-	Yes	Yes
DOUBLE	Double-precision floating-point	8 bytes	-	Yes	Yes
DECIMAL(precision,scale)	Decimal exact numeric type. A data type with fixed significant digits and decimal places, e.g., 3.5. **precision**: indicates the maximum number of digits that can be represented. **scale**: indicates the number of digits in the fractional part.	-	1 <= precision <= 38 0 <= scale <= 38 If **precision** and **scale** are not specified, the default is decimal(10,0).	Yes	Yes
BOOLEAN	Boolean	1 byte	TRUE/ FALSE	Yes	Yes
SMALLINT/SHORT	Signed integer	2 bytes	–32768 to 32767	Yes	Yes
TINYINT	Signed integer	1 byte	–128 to 127	Yes	No

 Copyright © Huawei Technologies Co., Ltd.

Data Type	Description	Storage Space	Value Range	OBS Table Support	Managed Table Support
BIGINT/ LONG	Signed integer	8 bytes	–9223372036854775808 to 9223372036854775807	Yes	Yes
TIMESTAMP	Timestamp, representing date and time, in raw data format, e.g., **2021-05-19 22:22:11**	-	-	Yes	Yes
TIMESTAMP_NTZ(p)	Timestamp without time zone, **p** is the fractional seconds precision (0-6), without time zone information	-	-	Yes	Yes
TIMESTAMP_LTZ(p)	Timestamp with time zone, **p** is the fractional seconds precision (0-6), with time zone information	-	-	Yes	Yes
CHAR	Fixed-length string	-	-	Yes	Yes

 Copyright © Huawei Technologies Co., Ltd.

Data Type	Description	Storage Space	Value Range	OBS Table Support	Managed Table Support
VARCHAR	Variable-length string	-	-	Yes	Yes
DATE	Date type, describes a specific year, month, and day, expressed in *yyyy-mm-dd* format, e.g., **2014-05-29**	-	The **DATE** type does not include time. The range of dates that can be represented is **0000-01-01** to **9999-12-31**.	Yes	Yes
VARIANT	Suitable for semi-structured data processing. Only supports Iceberg tables.	-	-	Yes	Yes

NOTE

- VARCHAR and CHAR are stored as the STRING type in practice.
- FLOAT is a single-precision floating-point type, occupying 4 bytes of storage, with approximately 7 significant digits.
- Spark Native already supports the VARIANT type.
- Only v3 tables support the VARIANT data type.

INT

A signed integer with a storage space of 4 bytes, ranging from –2147483648 to 2147483647. When NULL, the default value is **0**.

STRING

A string type.

 Copyright © Huawei Technologies Co., Ltd.

FLOAT

A single-precision floating-point type with a storage space of 4 bytes. When NULL, the default value for calculation is **0**.

Due to the limitations of how floating-point data is stored in computers, when comparing whether two floating-point data values are equal, direct comparison using **a == b** is not recommended due to precision issues. You are advised to use the method **abs(a - b) <= EPSILON** for comparison, where **EPSILON** is the allowable error range, typically **1.19209290E-07F**. If the absolute difference between two floating-point numbers is within this range, they are considered equal.

DOUBLE

A double-precision floating-point type with a storage space of 8 bytes. When NULL, the default value for calculation is **0**.

Due to the limitations of how floating-point data is stored in computers, when comparing whether two floating-point data values are equal, direct comparison using **a == b** is not recommended due to precision issues. You are advised to use the method **abs(a - b) <= EPSILON** for comparison, where **EPSILON** is the allowable error range, typically **2.2204460492503131E-16**. If the absolute difference between two floating-point numbers is within this range, they are considered equal.

DECIMAL

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

BOOLEAN

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

SMALLINT/SHORT

Signed integer with a storage space of 2 bytes. Its value ranges from –32768 to 32767. When NULL, the default value for calculation is **0**.

TINYINT

Signed integer with a storage space of 1 byte. Its value ranges from –128 to 127. When NULL, the default value for calculation is **0**.

 Copyright © Huawei Technologies Co., Ltd.

BIGINT/LONG

A signed integer with a storage space of 8 bytes, ranging from –9223372036854775808 to 9223372036854775807. Scientific notation is not supported. When NULL, the default value for calculation is **0**.

TIMESTAMP

Legacy UNIX TIMESTAMP is supported, providing the precision up to the microsecond level. **TIMESTAMP** is defined by the difference between the specified time and UNIX epoch (UNIX epoch time: 1970-01-01 00:00:00) in seconds. Data types that can be implicitly converted to TIMESTAMP include STRING (must have the format **yyyy-MM-dd HH:mm:ss[.ffffff]**. The precision after the decimal point is optional).

CHAR

The length of CHAR is fixed, using a fixed-length string of the specified length. It is stored as the STRING type in practice.

VARCHAR

VARCHAR is declared with a length that indicates the maximum number of characters in a string. Similar to **STRING**, the spaces at the end of **VARCHAR** are meaningful and affect the comparison result. It is stored as the STRING type in practice.

DATE

The DATE type can only be explicitly cast to and from DATE, TIMESTAMP, and STRING, as shown in **Table 10-2**.

Table 10-2 cast function conversion

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

 Copyright © Huawei Technologies Co., Ltd.

Explicit Conversion	Conversion Result
cast(date as string)	A STRING in the **yyyy-MM-dd** format is generated based on the date (yyyy-mm-dd) specified by **DATE** and returned as the value of **DATE**.

10.3 Complex Data Types

Table 10-3 lists complex data types that Spark SQL supports.

Table 10-3 Complex data types

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

Restrictions

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

 Copyright © Huawei Technologies Co., Ltd.

Example of ARRAY

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

1. Create a table.

   CREATE TABLE array_test(name STRING, id ARRAY < INT >) USING PARQUET;

2. Run the following statements to insert test data:

   INSERT INTO array_test VALUES ('test',array(1,2,3,4));

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

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

3. Query the result.

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

   SELECT * FROM array_test;

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

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

   SELECT id[0] FROM array_test;

   ```
   7
   4
   1
   ```

Example of Map

Create the table **map_test**, define the **score** parameter as the **map<STRING,INT>** data type (key is STRING type, value is INT type). After the table is successfully created, insert test data into **map_test**. The procedure is as follows:

1. Create a table.

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

2. Run the following statements to insert test data:

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

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

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

3. Query the result.

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

   SELECT * FROM map_test;

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

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

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

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

 Copyright © Huawei Technologies Co., Ltd.

Example of STRUCT

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

1. Create a table.

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

2. Run the following statements to insert test data:

   INSERT INTO struct_test VALUES (8, struct('user1',23));

   INSERT INTO struct_test VALUES (9, struct('user2',25));

   INSERT INTO struct_test VALUES (10, struct('user3',26));

3. Query the result.

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

   SELECT * FROM struct_test;

   ```
   8{"name":"user1","age":23}
   10{"name":"user3","age":26}
   9{"name":"user2","age":25}
   ```

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

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

   ```
   8       user1   23
   10      user3   26
   9    user2    25
   ```

 Copyright © Huawei Technologies Co., Ltd.

11 Iceberg Table Related Syntax

Apache Iceberg is an open table format for massive analytical datasets. It adds table functionality to compute engines through a high-performance table format, and its usage is fully consistent with SQL tables.

Iceberg is built for extremely large tables and has been applied in real-world business environments. A single table can hold tens of petabytes of data, and even such a massive table can be read without requiring a distributed SQL engine.

The Spark engine supports Iceberg syntax. For more information on Iceberg table-related syntax, see **Iceberg on Spark**.

 Copyright © Huawei Technologies Co., Ltd.

12 Function Syntax

12.1 SQL User-Defined Function

12.2 Create Function

12.3 Drop Function

12.4 Show Function Details

12.5 Show All Functions

12.6 DESCRIBE FUNCTION

12.1 SQL User-Defined Function

Function

SQL user-defined functions allow defining function logic using SQL expressions. They support scalar functions (returning a single value) and table-valued functions (returning a table result set).

Syntax

- **Creating a Scalar function**

```
CREATE [OR REPLACE] FUNCTION function_name ([parameter_name data_type [{ DEFAULT | = } default_value]] [, ...])
  RETURNS return_data_type
  RETURN expression;
```

- **Creating a table-valued function**

```
CREATE [OR REPLACE] FUNCTION function_name ([parameter_name data_type [{ DEFAULT | = } default_value]] [, ...])
  RETURNS TABLE (col_name col_type [, ...])
  RETURN query_statement;
```

- **Deleting a function**

```
DROP FUNCTION [IF EXISTS] function_name;
```

- **Viewing the function definition**

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

 Copyright © Huawei Technologies Co., Ltd.

Keywords

- **OR REPLACE**: Replaces the function if it already exists.
- **RETURNS**: Specifies the return value type.
- **RETURN**: The function body, a SQL expression or query statement.
- **DEFAULT / =**: Specifies a default value for a parameter.

Parameters

Parameter	Mandatory	Description
function_name	Yes	Function name.
parameter_name	No	Parameter name.
data_type	No	The parameter data type. If omitted, the system infers the type from the default value.
default_value	No	The default value for the parameter.
return_data_type	Yes	The return value data type.
expression	Yes	The scalar function body, a SQL expression.
query_statement	Yes	The table-valued function body, a SELECT query statement.

Examples

- **Creating a simple scalar function**

```
CREATE FUNCTION hello() RETURNS STRING RETURN 'Hello World';

SELECT hello();
-- Output: Hello World
```

- **Creating a scalar function with parameters**

```
CREATE FUNCTION add_tax(price DOUBLE) RETURNS DOUBLE RETURN price * 1.13;

SELECT add_tax(100.0);
-- Output: 113.0
```

- **Creating a function with default parameters**

```
CREATE FUNCTION greet(name STRING DEFAULT 'World') RETURNS STRING
  RETURN CONCAT('Hello, ', name, '!');

SELECT greet();
-- Output: Hello, World!

SELECT greet('Spark');
-- Output: Hello, Spark!
```

- **Creating a table-valued function**

```
CREATE FUNCTION range_n(n INT)
  RETURNS TABLE (id INT, value STRING)
  RETURN SELECT id, CONCAT('val_', id) FROM EXPLODE(SEQUENCE(1, n)) AS t(id);
```

 Copyright © Huawei Technologies Co., Ltd.

```
SELECT * FROM range_n(3);
-- Output: (1, val_1), (2, val_2), (3, val_3)
```

- **Replacing an existing function**

```
CREATE OR REPLACE FUNCTION add_tax(price DOUBLE) RETURNS DOUBLE
  RETURN price * 1.2;
```

- **Deleting a function**

```
DROP FUNCTION IF EXISTS add_tax;
```

Precautions

- SQL user-defined functions are a new feature in Spark 4.0.
- For scalar functions, **RETURN** is followed by a single SQL expression.
- For table-valued functions, **RETURN** is followed by a SELECT query statement.
- For functions with **DEFAULT** parameters, the parameter can be omitted when calling.
- Declared session variables can be referenced within the function body.

12.2 Create Function

Function

The platform supports creating custom functions such as user-defined functions (UDFs) and user-defined table functions (UDTFs) for use in Spark job development.

Syntax

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

resource:
  : JAR file_uri
```

or

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

resource:
  : JAR file_uri
```

Precautions

- If a function with the same name already exists in the database, the system reports an error.
- Only Hive syntax is supported for creating functions.
- Be aware of the following scenario: If a custom function **F1** is created specifying class **C1** and package **JAR1**, and another custom function **F2** is created also specifying class **C1** but package **JAR2**, the functions will conflict because **F2** and **F1** use the same class name, affecting job execution.
- The UDF hot-reload feature is in a restricted usage phase. To use it, submit a service ticket to request activation.

 Copyright © Huawei Technologies Co., Ltd.

Keywords

- **USING**: The resources to be loaded. Can be a list of JARs, files, or URIs.
- **OR REPLACE**: Supports the custom function hot-reload feature.
  - If **OR REPLACE** is not used when creating a custom function, pay attention to the following scenarios:

Table 12-1 Scenarios without OR REPLACE

Scenario	Example	Effective Mechanism	Operational Impact
Scenario 1: The implementation logic of the original package class is modified. The JAR package name and class name specified for the newly created function remain the same as the original.	UDF **F1** has been created under a Spark SQL endpoint, specifying class name **C1** and JAR file name **J1**. The function implementation in the **J1** package is later modified. A new function **F2** is created, specifying class name **C1** and JAR package name **J1**. Note: If step 2 continues to use the function name **F1**, creation will fail due to duplicate function name. In this case, consider using **OR REPLACE**, or replacing all occurrences of function **F1** in jobs with **F2**.	The Spark SQL endpoint needs to be restarted for the newly created UDF **F2** to take effect.	The Spark SQL endpoint needs to be restarted, affecting currently running jobs. After the endpoint is restarted, the original functionality of **F1** is affected, and F1's functionality becomes the same as **F2**.
Scenario 2: A new class is added to the original package. The newly created function specifies the new class, and the package name remains unchanged.	UDF **F1** has been created under a Spark SQL endpoint, specifying class name **C1** and JAR file name **J1**. A new class **C2** is added to the **J1** JAR package, while **C1** remains unchanged. A new UDF **F2** is created, specifying class name **C2** and JAR package name **J1**.	The Spark SQL endpoint needs to be restarted for the newly created UDF **F2** to take effect.	The Spark SQL endpoint needs to be restarted, affecting currently running jobs. After the endpoint is restarted, the functionality of **F1** remains unchanged.

 Copyright © Huawei Technologies Co., Ltd.

Scenario	Example	Effective Mechanism	Operational Impact
Scenario 3: The implementation logic of the original package class remains unchanged, but the package is repackaged with a new name. The newly created function specifies the new JAR package name, and the class name remains unchanged.	UDF **F1** has been created under a Spark SQL endpoint, specifying class name **C1** and JAR file name **J1**. The JAR package is repackaged as **J2** with unchanged logic. A new UDF **F2** is created, specifying class name **C1** and JAR package name **J2**.	The newly created UDF F2 takes effect immediately.	No impact.

- If **OR REPLACE** is used when creating a custom function, it indicates that the existing function content needs to be functionally replaced and take effect in real-time.

  This feature is in a restricted usage phase. To use it, submit a service ticket to request activation.

  - To make it take effect immediately on all Spark SQL endpoints, you need to execute **CREATE OR REPLACE** *xxx* **FUNCTION ...** on each Spark SQL endpoint separately. Otherwise, endpoints where it was not executed may experience a delay of 0–12 hours before taking effect.
  - If a running job uses UDF **F1**, which specifies class name **C1** and package name **J1**, and after the job has run halfway, the logic of the **J1** package is modified and **CREATE OR REPLACE FUNCTION F1** is executed, subsequent jobs will use the new **F1** function logic, but the job that has already executed halfway will use the old function logic.

 Copyright © Huawei Technologies Co., Ltd.

Table 12-2 Scenarios with OR REPLACE

Scenario	Example	Effective Mechanism	Operational Impact
Scenario 1: The implementation logic of the original package class is modified. The JAR package name and class name specified for the newly created function remain the same as the original.	UDF **F1** has been created under a Spark SQL endpoint, specifying class name **C1** and JAR file name **J1**. The function implementation in the **J1** package is later modified. A new function **F1** is created, specifying class name **C1** and JAR package name **J1**.	Takes effect immediately.	The functionality of **F1** becomes the newly modified logic.
Scenario 2: A new class is added to the original package. The newly created function specifies the new class, and the package name remains unchanged.	UDF **F1** has been created under a Spark SQL endpoint, specifying class name **C1** and JAR file name **J1**. A new class **C2** is added to the **J1** JAR package, while **C1** remains unchanged. A new UDF **F2** is created, specifying class name **C2** and JAR package name **J1**. Since this is the first time creating a function specifying **C2**, function **F2** cannot take effect immediately.	Two ways to take effect: 1. Execute the **CREATE OR REPLACE FUNCTION F2** statement again. 2. Restart the Spark SQL endpoint.	If the Spark SQL endpoint is restarted, currently running jobs will be affected.

 Copyright © Huawei Technologies Co., Ltd.

Scenario	Example	Effective Mechanism	Operational Impact
Scenario 3: The class name in the original package is modified. The newly created function specifies the new class name, and the package name remains unchanged.	UDF **F1** has been created under a Spark SQL endpoint, specifying class name **C1** and JAR file name **J1**. The original **C1** in the **J1** JAR package is modified to **C2**, and **C1** no longer exists. A new UDF **F2** is created, specifying class name **C2** and JAR package name **J1**.	Takes effect immediately.	The functionality of **F1** becomes invalid, and the functionality of **F2** takes effect immediately.
Scenario 4: The implementation logic of the original package class remains unchanged, but the package is repackaged with a new name. The newly created function specifies the new JAR package name, and the class name remains unchanged.	UDF **F1** has been created under a Spark SQL endpoint, specifying class name **C1** and JAR file name **J1**. The JAR package is repackaged as **J2** with unchanged logic. A new UDF **F2** is created, specifying class name **C1** and JAR package name **J2**.	Takes effect immediately.	No impact.

Examples

Create the function **mergeBill**.

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

 Copyright © Huawei Technologies Co., Ltd.

12.3 Drop Function

Function

Deletes a function.

Syntax

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

Keywords

- **TEMPORARY**: Specifies to delete a temporary function.
- **IF EXISTS**: Used when the function to be deleted does not exist to avoid a system error.

Precautions

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

Example

Delete the function **mergeBill**.

```
DROP FUNCTION mergeBill;
```

12.4 Show Function Details

Function

Views information about a specified function.

Syntax

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

Keywords

EXTENDED: Displays extended usage information.

Precautions

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

Example

View information about the **mergeBill** function.

```
DESCRIBE FUNCTION mergeBill;
```

 Copyright © Huawei Technologies Co., Ltd.

12.5 Show All Functions

Function

Views all functions in the current project.

Syntax

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

Where **regex** is a regular expression. Refer to the following examples:

Table 12-3 regex parameter examples

Regular Expression	Matching Meaning
'xpath*'	Matches all function names starting with **xpath**. For example: **SHOW FUNCTIONS LIKE 'xpath*'** matches **xpath**, **xpath_int**, **xpath_string**, and other functions starting with **xpath**.
'x[a-z]+'	Matches function names starting with **x**, followed by one or more characters in the range **a** to **z**. For example: **xpath**, **xtest**, etc.
'x.*h'	Matches function names starting with **x**, ending with **h**, with any number of characters (including zero) in between. For example: **xpath**, **xtesth**, etc.

Keywords

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

Precautions

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

Example

View all current functions.

```
SHOW FUNCTIONS;
```

 Copyright © Huawei Technologies Co., Ltd.

12.6 DESCRIBE FUNCTION

Function

The **DESCRIBE FUNCTION** statement is used to view metadata information about a function, including the function name, implementation class, and usage description. If the **EXTENDED** option is specified, extended usage information and examples are also displayed.

Syntax

```
{ DESC | DESCRIBE } FUNCTION [ EXTENDED ] function_name
```

Parameters

Parameter	Description
EXTENDED	Optional. Displays extended information about the function, including usage methods and examples.
function_name	The function name. Can be a built-in function or a custom function. Can be qualified using **database_name.function_name**. If the database name is not specified, the current database is used.

Precautions

- When **EXTENDED** is not used, only the function name, implementation class, and basic usage are displayed.
- When **EXTENDED** is used, examples and detailed usage instructions are additionally displayed.
- If the function does not exist, an error is reported.
- **DESC** is an abbreviation of **DESCRIBE**.

Examples

- **Viewing a built-in scalar function**

```
DESC FUNCTION abs;
-- +----------------------------------------------------------------+
-- |function_desc                                                   |
-- +----------------------------------------------------------------+
-- |Function: abs                                                   |
-- |Class: org.apache.spark.sql.catalyst.expressions.Abs            |
-- |Usage: abs(expr) - Returns the absolute value of the numeric value.|
-- +----------------------------------------------------------------+
```

- **Viewing a built-in scalar function (extended information)**

```
DESC FUNCTION EXTENDED abs;
-- +----------------------------------------------------------------+
-- |function_desc                                                   |
-- +----------------------------------------------------------------+
-- |Function: abs                                                   |
-- |Class: org.apache.spark.sql.catalyst.expressions.Abs            |
-- |Usage: abs(expr) - Returns the absolute value of the numeric value.|
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- |Extended Usage:                                                  |
-- |    Examples:                                                    |
-- |      > SELECT abs(-1);                                          |
-- |       1                                                         |
-- +-----------------------------------------------------------------+
```

- **Viewing a built-in aggregate function**

```
DESC FUNCTION max;
-- +-----------------------------------------------------------------+
-- |function_desc                                                    |
-- +-----------------------------------------------------------------+
-- |Function: max                                                    |
-- |Class: org.apache.spark.sql.catalyst.expressions.aggregate.Max|
-- |Usage: max(expr) - Returns the maximum value of `expr`.       |
-- +-----------------------------------------------------------------+
```

- **Viewing a generator function (extended information)**

```
DESC FUNCTION EXTENDED explode;
-- +-----------------------------------------------------------------+
-- |function_desc                                                    |
-- +-----------------------------------------------------------------+
-- |Function: explode                                                |
-- |Class: org.apache.spark.sql.catalyst.expressions.Explode       |
-- |Usage: explode(expr) - Separates the elements of array `expr`  |
-- |  into multiple rows, or the elements of map `expr` into      |
-- |  multiple rows and columns.                                     |
-- |Extended Usage:                                                  |
-- |    Examples:                                                    |
-- |      > SELECT explode(array(10, 20));                           |
-- |       10                                                        |
-- |       20                                                        |
-- +-----------------------------------------------------------------+
```

13 Query Syntax

13.1 SELECT
13.2 Sorting
13.3 Grouping
13.4 JOINS
13.5 Clauses
13.6 Alias
13.7 Set Operations
13.8 CASE...WHEN
13.9 Pipe Syntax (|>)

13.1 SELECT

Function

The **SELECT** statement is used to query data from tables and is the most basic query statement in Spark SQL. Spark 4.0 supports the complete SELECT syntax, including LATERAL, PIVOT/UNPIVOT, OFFSET pagination, and other features.

Syntax

```
[ WITH with_query [, ...] ]
SELECT [ ALL | DISTINCT ] { * | projectItem [, ...] }
  FROM table_reference [, ...]
  [ WHERE boolean_expression ]
  [ GROUP BY { expression | GROUPING SETS ( ... ) | ROLLUP ( ... ) | CUBE ( ... ) } [, ...] ]
  [ HAVING boolean_expression ]
  [ PIVOT ( aggregate_expr FOR pivot_column IN ( value1 [, ...] ) ) ]
  [ UNPIVOT ( value_column FOR name_column IN ( column1 [, ...] ) ) ]
  [ LATERAL VIEW function_expr AS alias [, ...] ]
  [ LATERAL ( subquery ) ]

  [ ORDER BY { expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] } [, ...] ]
  [ SORT BY { expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] } [, ...] ]
  [ CLUSTER BY expression [, ...] ]
  [ DISTRIBUTE BY expression [, ...] ]
```

 Copyright © Huawei Technologies Co., Ltd.

```
[ OFFSET integer_expression ]
[ LIMIT integer_expression ]
```

Where **projectItem** is:

```
expression [ [ AS ] alias ]
```

Where **table_reference** is:

```
{ table_name [ TABLE ] [ alias ]
  | join_table  | LATERAL ( subquery ) [ alias ]
  | ( subquery ) [ alias ]
  | VALUES ( { expression } [, ...] ) [, ...] [ alias ]
  | UNNEST(expression) [ alias ] }
```

Parameters

Table 13-1 Parameter descriptions

Parameter	Description
WITH with_query	Defines a CTE to simplify complex queries and improve readability. See WITH...AS for details.
ALL	Returns all rows, including duplicate rows. This is the default option.
DISTINCT	Removes duplicate rows from the result set.
*	Queries all columns.
projectItem	Specifies the columns or expressions to query. An alias can be specified using **AS**.
FROM table_reference	Specifies the data source for the query. Can be a table name, subquery, JOIN result, LATERAL subquery, etc. See FROM for details.
WHERE	Specifies filter conditions. Only rows that satisfy the conditions are returned. See WHERE for details.
GROUP BY	Groups rows by the specified expression, typically used with aggregate functions. Supports GROUPING SETS, ROLLUP, and CUBE extensions.
HAVING	Filters the results of GROUP BY. See HAVING for details.
PIVOT	Rotates rows to columns. Converts row data into columns based on the values of a specified column.
UNPIVOT	Rotates columns to rows. Converts column data into rows.
LATERAL VIEW	Used with table-generating functions (such as **explode**) to expand function output into multiple rows.
LATERAL (subquery)	A LATERAL subquery allows the subquery to reference columns from the outer query.
ORDER BY	Global sorting. See ORDER BY for details.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
SORT BY	Local sorting. Sorts within each Reducer. See SORT BY for details.
CLUSTER BY	Buckets and sorts within buckets. Equivalent to **DISTRIBUTE BY** + **SORT BY** on the same field. See CLUSTER BY for details.
DISTRIBUTE BY	Buckets data by the specified field, distributing rows with the same key value to the same Reducer. See DISTRIBUTE BY for details.
OFFSET	Skips the first *N* rows of results. Used for pagination queries.
LIMIT	Limits the number of rows returned.
NULLS FIRST / NULLS LAST	Specifies the position of NULL values in the sort result. **NULLS FIRST** places NULLs first, and **NULLS LAST** places NULLs last.

Precautions

- The table being queried must already exist. Otherwise, a query error is reported.
- When using **NOT IN** in a **WHERE** clause, ensure that the subquery result set does not contain any NULL values. If a NULL value exists, the entire query result will be an empty set. See the FAQ below for details.

Examples

- Basic query:
```
SELECT * FROM student;
```
- Query with DISTINCT deduplication:
```
SELECT DISTINCT name FROM student;
```
- Query with WHERE condition:
```
SELECT name, score FROM student WHERE score > 90;
```
- Query with GROUP BY and HAVING:
```
SELECT class_id, AVG(score) AS avg_score FROM student
  GROUP BY class_id
  HAVING AVG(score) > 80;
```
- Query with OFFSET pagination (Spark 4.0):
```
SELECT * FROM student ORDER BY score DESC LIMIT 10 OFFSET 20;
```
- Query with PIVOT row-to-column (Spark 4.0):
```
SELECT * FROM sales
  PIVOT (SUM(amount) FOR year IN (2023, 2024, 2025));
```
- Query with LATERAL subquery (Spark 4.0):
```
SELECT * FROM orders, LATERAL (SELECT * FROM items WHERE items.order_id = orders.id);
```

:: Type Cast

```
SELECT '123' :: INT;
 -- Equivalent to CAST('123' AS INT)
```

COLLATE Collation

```
SELECT name FROM t ORDER BY name COLLATE unicode_ci;
```

 Copyright © Huawei Technologies Co., Ltd.

FAQ: What Can I Do If the Query Result Is Empty When the Not In Subquery Contains Null Values?

- **Issue overview**: In SQL syntax, if a **NOT IN** subquery contains a single NULL value, the entire query result will be an empty set (no data returned).
- **Example**: In the following example, the expected result is **3**, but an empty set is returned.

```
-- Table A       -- Table B
id          value
---         -----
1           1
2           2
3
```

  Run the following SQL statement:

```
SELECT * FROM A
WHERE id NOT IN (SELECT value FROM B);
```

- **Cause analysis**:
  id NOT IN (1, 2, NULL) is equivalent to **id <> 1 AND id <> 2 AND id <> NULL**.

  id <> NULL evaluates to **UNKNOWN** (SQL's three-valued logic), making the entire expression **FALSE**. Therefore, no data is returned.
- **Solutions**:

  Solution 1: Use **NOT EXISTS** instead of **NOT IN**.

```
SELECT * FROM A
WHERE NOT EXISTS (
  SELECT 1 FROM B WHERE B.value = A.id
);
```

  Solution 2: Filter out NULL values in the subquery.

```
SELECT * FROM A
WHERE id NOT IN (
  SELECT value FROM B WHERE value IS NOT NULL
);
```

13.2 Sorting

13.2.1 ORDER BY

Function

The **ORDER BY** clause is used to perform global sorting on query results. Spark 4.0 enhances the sorting functionality by supporting **NULLS FIRST/LAST** to specify the sort position of NULL values and can be used with **OFFSET** for pagination.

Syntax

```
SELECT projectItem [, ...] FROM table_reference
  ORDER BY { expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] } [, ...]
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Description
expression	The sort expression. Can be a column name, column alias, or expression.
ASC	Sorts in ascending order. This is the default option.
DESC	Sorts in descending order.
NULLS FIRST	Places NULL values at the beginning of the sort result.
NULLS LAST	Places NULL values at the end of the sort result. By default, NULLs are placed last in ascending order and first in descending order.

Precautions

- **ORDER BY** performs global sorting. All data is gathered into a single Reducer for processing.
- When used with **GROUP BY**, **ORDER BY** can be followed by aggregate functions.
- **ORDER BY** can reference aliases from the SELECT list.
- Can be used with **LIMIT** and **OFFSET** for pagination queries.

Examples

Sort by a single column in ascending order:

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

Sort by a single column in descending order:

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

Sort by multiple columns:

```
SELECT * FROM student ORDER BY class_id ASC, score DESC;
```

Specify NULL value sort position:

```
SELECT * FROM student ORDER BY score DESC NULLS LAST;
```

Sort using a column alias:

```
SELECT name, score * 1.5 AS weighted_score FROM student   ORDER BY weighted_score DESC;
```

Pagination query (with OFFSET, Spark 4.0):

```
SELECT * FROM student ORDER BY score DESC LIMIT 10 OFFSET 20;
```

 Copyright © Huawei Technologies Co., Ltd.

13.2.2 SORT BY

Function

The **SORT BY** clause is used to perform local sorting of data within each Reducer. Unlike **ORDER BY** which performs global sorting, **SORT BY** only guarantees that the data within each Reducer is sorted.

Syntax

```
SELECT projectItem [, ...] FROM
  table_reference SORT BY { expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] } [, ...]
```

Parameters

Parameter	Description
expression	The sort expression. Can be a column name or expression.
ASC	Sorts in ascending order. This is the default option.
DESC	Sorts in descending order.
NULLS FIRST	Places NULL values at the beginning of the sort result.
NULLS LAST	Places NULL values at the end of the sort result.

Precautions

- **SORT BY** is a local sort. It only guarantees that data within each Reducer is sorted, not globally.
- For global sorting, use **ORDER BY**.
- **SORT BY** is typically used with **DISTRIBUTE BY** to first bucket data and then sort within buckets.
- **SORT BY** supports **NULLS FIRST/LAST** to specify the sort position of NULL values.

Examples

Sort by score in ascending order within Reducers:

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

Sort by score in descending order within Reducers:

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

Use with DISTRIBUTE BY:

```
SELECT * FROM student DISTRIBUTE BY class_id SORT BY score DESC;
```

Sort multiple fields and specify NULL value position:

```
SELECT * FROM student SORT BY class_id ASC NULLS LAST, score DESC NULLS LAST;
```

 Copyright © Huawei Technologies Co., Ltd.

13.2.3 CLUSTER BY

Function

The **CLUSTER BY** clause is used to bucket data by the specified fields and sort within the buckets by the same fields. **CLUSTER BY** is equivalent to using **DISTRIBUTE BY** and **SORT BY** with the same fields.

Syntax

```
SELECT projectItem [, ...] FROM table_reference   CLUSTER BY expression [, ...]
```

Parameters

Parameter	Description
expression	The bucketing and sorting expression. Supports single and multiple fields. **CLUSTER BY** distributes data into different buckets based on the hash value of the specified fields and sorts within the buckets by the same fields.

Precautions

- **CLUSTER BY** can only sort in ascending order. **ASC** or **DESC** cannot be specified. To specify a sort direction, use the **DISTRIBUTE BY** + **SORT BY** combination.
- **CLUSTER BY** is equivalent to using **DISTRIBUTE BY** and **SORT BY** with the same fields.

Examples

Bucket and sort by the **score** field:

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

Bucket and sort by multiple fields:

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

To specify descending order, use **DISTRIBUTE BY** + **SORT BY**:

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

13.2.4 DISTRIBUTE BY

Function

The **DISTRIBUTE BY** clause is used to bucket (repartition) data by the specified fields, distributing rows with the same key value to the same Reducer. **DISTRIBUTE BY** does not perform sorting within the buckets.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
SELECT projectItem [, ...] FROM table_reference
  DISTRIBUTE BY expression [, ...]
```

Parameters

Parameter	Description
expression	The bucketing expression. Supports single and multiple fields. Data is distributed to different Reducers based on the hash value of the expression. Rows with the same key value are guaranteed to be in the same Reducer.

Precautions

- **DISTRIBUTE BY** is only responsible for bucketing, not for sorting within buckets.
- To sort after bucketing, use **SORT BY** in conjunction. **DISTRIBUTE BY** and **SORT BY** can use different fields.
- **CLUSTER BY** is a shorthand for **DISTRIBUTE BY** and **SORT BY** using the same fields.

Examples

Bucket by the **score** field:

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

Bucket and then sort:

```
SELECT * FROM student DISTRIBUTE BY class_id SORT BY score DESC;
```

Bucket by multiple fields:

```
SELECT * FROM student DISTRIBUTE BY class_id, name;
```

13.3 Grouping

13.3.1 GROUP BY by Column

Function

Groups a table by columns.

Syntax

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

 Copyright © Huawei Technologies Co., Ltd.

Keywords

GROUP BY: Can be divided into single-column GROUP BY and multi-column GROUP BY based on the columns.

- Single-column GROUP BY: The GROUP BY clause contains only one column. The fields in **col_name_list** must appear in the fields of **attr_expr_list**. Multiple aggregate functions, such as **count()**, **sum()**, can be used in **attr_expr_list**, and the aggregate functions can contain other fields.
- Multi-column GROUP BY: The GROUP BY clause contains more than one column. The query statement will group by all fields in the GROUP BY clause. Records with the same values for all fields will be placed in the same group. Similarly, the fields appearing in GROUP BY must be within the fields of **attr_expr_list**, and aggregate functions can also be used in **attr_expr_list**.

Precautions

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

Example

Group the **student** table by the **score** and **name** fields, and return the grouped results.

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

13.3.2 GROUP BY Expression

Function

Groups a table by an expression.

Syntax

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

Keywords

groupby_expression: Can be a single field, multiple fields, or an aggregate function, string function, etc.

Precautions

- The table to be grouped must already exist. Otherwise, an error is reported.
- Similar to single-column grouping, the fields appearing in GROUP BY must be included in the fields of **attr_expr_list**. Expressions support built-in functions, user-defined functions, etc.

Example

Use the **substr** function to extract a substring of the **name** field, group by that substring, and return each substring and its corresponding record count.

 Copyright © Huawei Technologies Co., Ltd.

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

13.3.3 Using HAVING in GROUP BY

Function

Uses the HAVING clause to filter after grouping a table.

Syntax

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

Keywords

groupby_expression: Can be a single field, multiple fields, or an aggregate function, string function, etc.

Precautions

- The table to be grouped must already exist. Otherwise, an error is reported.
- If the filter condition is affected by the query results of GROUP BY, the WHERE clause cannot be used for filtering. Instead, the HAVING clause must be used. If HAVING is used in conjunction with GROUP BY, the data is first grouped by GROUP BY, and then filtered in the HAVING clause. The HAVING clause supports arithmetic operations, aggregate functions, etc.

Example

First, group the **transactions** table by **num**. Then, use the HAVING clause to filter the query results. Records where the maximum value of the product of **price** and **amount** is greater than 5000 will be selected. Return the corresponding **num** and the maximum value of the product of **price** and **amount**.

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

13.3.4 ROLLUP

Function

ROLLUP is an extension of GROUP BY, used to generate multi-level aggregate rows, super-aggregate rows, and grand total rows. ROLLUP decreases the grouping granularity from right to left, achieving hierarchical aggregate statistics.

Syntax

```
SELECT projectItem [, ...] FROM table_reference
  GROUP BY ROLLUP ( expression [, ...] )
```

Or using the WITH syntax:

 Copyright © Huawei Technologies Co., Ltd.

```
SELECT projectItem [, ...] FROM table_reference
  GROUP BY expression [, ...] WITH ROLLUP;
```

Parameters

Parameter	Description
ROLLUP	Groups and aggregates by a hierarchical structure that decreases from right to left based on the specified columns. For example, ROLLUP(a, b, c) is equivalent to GROUPING SETS ((a, b, c), (a, b), (a), ()).
expression	The grouping expression. Can be a column name or expression.

ROLLUP Equivalent Transformation

GROUP BY ROLLUP(a, b, c) is equivalent to the UNION ALL of the following four grouping queries:

- Group subtotal for (a, b, c):
  ```
  SELECT a, b, c, sum(expression) FROM table GROUP BY a, b, c;
  ```
- Group subtotal for (a, b):
  ```
  SELECT a, b, NULL, sum(expression) FROM table GROUP BY a, b;
  ```
- Group subtotal for (a):
  ```
  SELECT a, NULL, NULL, sum(expression) FROM table GROUP BY a;
  ```
- Grand total for ():
  ```
  SELECT NULL, NULL, NULL, sum(expression) FROM table;
  ```

Precautions

- The table to be grouped must already exist. Otherwise, an error is reported.
- The grouping granularity of ROLLUP decreases from right to left. The rightmost column is removed first.
- The **GROUPING()** function can be used to determine whether a column participated in the grouping of the current row.

Examples

Generate multi-level aggregates based on **group_id** and **job**:

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

Equivalent WITH syntax:

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

Use the **GROUPING()** function to identify the grouping level:

```
SELECT group_id, job, SUM(salary),
  GROUPING(group_id) AS g1, GROUPING(job) AS g2
  FROM group_test
  GROUP BY ROLLUP (group_id, job);
```

 Copyright © Huawei Technologies Co., Ltd.

13.3.5 GROUPING SETS

Function

GROUPING SETS is an extension of GROUP BY, used to implement multiple sets of grouping statistics with different granularities in a single query. It is equivalent to the result of merging multiple GROUP BY queries using UNION ALL.

Syntax

```
SELECT projectItem [, ...] FROM table_reference
  GROUP BY { expression | GROUPING SETS ( grouping_set [, ...] ) } [, ...]
```

Where **grouping_set** is:

```
{ ( expression [, ...] ) | expression }
```

Parameters

Parameter	Description
GROUPING SETS	Specifies multiple grouping sets. Each grouping set defines a set of columns for GROUP BY. Columns not participating in the grouping are filled with NULL in the result.
grouping_set	A grouping set, represented by a combination of columns enclosed in parentheses. Parentheses can be omitted for a single column.

GROUPING SETS Equivalent Transformation

- **GROUP BY a, b GROUPING SETS ((a,b))** is equivalent to:
  ```
  SELECT a, b, sum(expression) FROM table GROUP BY a, b;
  ```
- **GROUP BY a, b GROUPING SETS (a, b)** is equivalent to:
  ```
  SELECT a, NULL, sum(expression) FROM table GROUP BY a
  UNION ALL
  SELECT NULL, b, sum(expression) FROM table GROUP BY b;
  ```
- **GROUP BY a, b GROUPING SETS ((a,b), a)** is equivalent to:
  ```
  SELECT a, b, sum(expression) FROM table GROUP BY a, b
  UNION ALL
  SELECT a, NULL, sum(expression) FROM table GROUP BY a;
  ```
- **GROUP BY a, b GROUPING SETS ((a,b), a, b, ())** is equivalent to:
  ```
  SELECT a, b, sum(expression) FROM table GROUP BY a, b
  UNION ALL
  SELECT a, NULL, sum(expression) FROM table GROUP BY a
  UNION ALL
  SELECT NULL, b, sum(expression) FROM table GROUP BY b
  UNION ALL
  SELECT NULL, NULL, sum(expression) FROM table;
  ```

Precautions

- The table to be grouped must already exist. Otherwise, an error is reported.
- Columns not participating in the grouping are filled with NULL in the result.
- The **GROUPING()** function can be used to determine whether a column participated in the grouping of the current row.

 Copyright © Huawei Technologies Co., Ltd.

Example

Generate cross-table rows based on the **group_id** and **job** fields:

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

Multiple grouping sets including a grand total:

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

13.4 JOINS

13.4.1 INNER JOIN

Function

An INNER JOIN combines rows from two tables that satisfy the join condition into a result set. Only records that meet the JOIN condition in both tables are returned.

Syntax

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

Parameters

Parameter	Description
attr_expr_list	The list of query column expressions, specifying the columns to be returned.
table_reference	The table name or subquery to be joined. Must be an existing table. Otherwise, an error is reported.
JOIN / INNER JOIN	Inner join keywords. Only records that satisfy the JOIN condition in the participating tables are displayed. Both have the same semantics.
join_condition	The join condition, typically an equality comparison expression between columns of the two tables.
ON	The keyword specifying the join condition.

Precautions

- The tables to be joined must already exist. Otherwise, an error is reported.
- More than two tables can be joined in a single query by using multiple JOIN clauses sequentially.

 Copyright © Huawei Technologies Co., Ltd.

Example

Join the **student_info** and **course_info** tables by matching course IDs to view student names and their selected course names.

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

Equivalent syntax:

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

13.4.2 LEFT OUTER JOIN

Function

A LEFT OUTER JOIN matches records from the left table against the right table and returns all records from the left table. For records in the right table that have no matching value, the corresponding columns in the result set return NULL.

Syntax

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

Parameters

Parameter	Description
attr_expr_list	The list of query column expressions, specifying the columns to be returned.
table_reference	The table name or subquery to be joined. Must be an existing table. Otherwise, an error is reported.
LEFT OUTER JOIN	LEFT OUTER JOIN keyword. Returns all records from the left table. Records in the right table without a matching value return NULL. **LEFT JOIN** is an abbreviation of **LEFT OUTER JOIN**.
join_condition	The join condition, typically an equality comparison expression between columns of the two tables.
ON	The keyword specifying the join condition.

Precautions

- The tables to be joined must already exist. Otherwise, an error is reported.
- **LEFT JOIN** is an abbreviation of **LEFT OUTER JOIN**. Both have identical semantics.

 Copyright © Huawei Technologies Co., Ltd.

Example

In a LEFT OUTER JOIN, match **courseId** from **student_info** with **courseId** from **course_info**. Return the names of students who have selected courses and their selected course names. Records in the right table without a matching value will return NULL.

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

Equivalent syntax:

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

13.4.3 RIGHT OUTER JOIN

Function

A RIGHT OUTER JOIN matches records from the right table against the left table and returns all records from the right table. For records in the left table that have no matching value, the corresponding columns in the result set return NULL.

Syntax

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

Parameters

Parameter	Description
attr_expr_list	The list of query column expressions, specifying the columns to be returned.
table_reference	The table name or subquery to be joined. Must be an existing table. Otherwise, an error is reported.
RIGHT OUTER JOIN	RIGHT OUTER JOIN keyword. Returns all records from the right table. Records in the left table without a matching value return NULL. **RIGHT JOIN** is an abbreviation of **RIGHT OUTER JOIN**.
join_condition	The join condition, typically an equality comparison expression between columns of the two tables.
ON	The keyword specifying the join condition.

Precautions

- The tables to be joined must already exist. Otherwise, an error is reported.
- **RIGHT JOIN** is an abbreviation of **RIGHT OUTER JOIN**. Both have identical semantics.
- A RIGHT OUTER JOIN is symmetric to a LEFT OUTER JOIN: **A RIGHT OUTER JOIN B ON ...** is equivalent to **B LEFT OUTER JOIN A ON ...**.

 Copyright © Huawei Technologies Co., Ltd.

Example

A RIGHT OUTER JOIN is similar to a LEFT OUTER JOIN, but it returns all records from the right table (**course_info**). Records in the left table without a matching value will return NULL.

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

Equivalent syntax:

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

13.4.4 FULL OUTER JOIN

Function

A full outer join (FULL OUTER JOIN) matches all records from the left table against all records from the right table and returns all records from both tables. For records without a matching value, the corresponding columns in the result set return NULL.

Syntax

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

Parameters

Parameter	Description
attr_expr_list	The list of query column expressions, specifying the columns to be returned.
table_reference	The table name or subquery to be joined. Must be an existing table. Otherwise, an error is reported.
FULL OUTER JOIN	FULL OUTER JOIN keyword. Returns all records from both the left and right tables. Records without a matching value return NULL. **FULL JOIN** is an abbreviation of **FULL OUTER JOIN**.
join_condition	The join condition, typically an equality comparison expression between columns of the two tables.
ON	The keyword specifying the join condition.

Precautions

- The tables to be joined must already exist. Otherwise, an error is reported.
- **FULL JOIN** is an abbreviation of **FULL OUTER JOIN**. Both have identical semantics.
- The result set of a full outer join is equal to the union of a LEFT OUTER JOIN and a RIGHT OUTER JOIN.

 Copyright © Huawei Technologies Co., Ltd.

Example

Using a FULL OUTER JOIN, all records from both tables can be returned. Records in the left or right table without a matching value will return NULL.

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

Equivalent syntax:

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

13.4.5 IMPLICIT JOIN

Function

An IMPLICIT JOIN has the same function as an INNER JOIN. It returns the result set from two tables that satisfy the WHERE condition, but it does not use the JOIN keyword to explicitly specify the join condition. Instead, it is implemented through the WHERE clause.

Syntax

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

Parameters

Parameter	Description
col_name	The column name, specifying the columns to be returned.
table_reference	The table name to be joined. Must be an existing table. Otherwise, an error is reported.
WHERE	An IMPLICIT JOIN uses the WHERE condition to achieve a join similar to **JOIN ... ON ...**, returning matching records. Supports both equi-join and non-equi-join conditions.

Precautions

- The tables to be joined must already exist. Otherwise, an error is reported.
- The IMPLICIT JOIN command does not contain the **JOIN ... ON ...** keywords. Instead, it uses the WHERE clause as the join condition to connect the two tables.
- An implicit join is equivalent to an INNER JOIN. You are advised to use the explicit JOIN syntax for better readability.

Examples

Return student names and course names where the **courseId** matches.

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

 Copyright © Huawei Technologies Co., Ltd.

Example of a non-equi implicit join:

```
SELECT a.name, b.name FROM student_info_1 a, student_info_2 b WHERE a.score > b.score;
```

13.4.6 CROSS JOIN (Cartesian Join)

Function

A CROSS JOIN combines each record from the first table with all records from the second table. If the first table has **m** records and the second table has **n** records, the result will have **m * n** records.

Syntax

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

SELECT attr_expr_list FROM table_reference
  CROSS JOIN table_reference;
```

Parameters

Parameter	Description
attr_expr_list	The list of query column expressions, specifying the columns to be returned.
table_reference	The table name or subquery to be joined. Must be an existing table. Otherwise, an error is reported.
CROSS JOIN	The CROSS JOIN keyword, the standard way to compute a Cartesian product.
join_condition	An optional join condition. If this condition is always true (e.g., 1 = 1), the join is a Cartesian join. A **CROSS JOIN** without an **ON** clause also represents a Cartesian product.

Precautions

- The tables to be joined must already exist. Otherwise, an error is reported.
- **CROSS JOIN** is the standard way to compute a Cartesian product. If the **ON** clause is omitted, the Cartesian product of the two tables is computed directly.
- Cartesian joins can produce a large number of records. Use with caution.

Example

Return all combinations of student names and course names from the **student_info** and **course_info** tables.

```
SELECT student_info.name, course_info.courseName FROM student_info CROSS JOIN course_info;
```

Equivalent syntax (using an always-true ON condition):

 Copyright © Huawei Technologies Co., Ltd.

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

13.4.7 LEFT SEMI JOIN

Function

A LEFT SEMI JOIN is used to view records in the left table that satisfy the JOIN condition. A LEFT SEMI JOIN only returns records from the left table and does not return columns from the right table.

Syntax

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

Parameters

Parameter	Description
attr_expr_list	The list of query column expressions, specifying the columns to be returned. The fields involved can only be fields from the left table. Otherwise, an error is reported.
table_reference	The table name or subquery to be joined. Must be an existing table. Otherwise, an error is reported.
LEFT SEMI JOIN	LEFT SEMI JOIN keyword. Only displays records from the left table that satisfy the JOIN condition.
join_condition	The join condition, typically an equality comparison expression between columns of the two tables.
ON	The keyword specifying the join condition.

Precautions

- The tables to be joined must already exist. Otherwise, an error is reported.
- The fields involved in **attr_expr_list** can only be fields from the left table. Otherwise, an error is reported.
- Difference between LEFT SEMI JOIN and LEFT OUTER JOIN: A LEFT SEMI JOIN only returns records from the left table that satisfy the JOIN condition, while a LEFT OUTER JOIN returns all records from the left table. For records that do not match the JOIN condition, the columns from the right table return NULL.
- The effect of a LEFT SEMI JOIN is equivalent to a **WHERE ... IN** or **WHERE EXISTS** subquery.

Example

Return the names of students who have selected courses and their selected course IDs.

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

 Copyright © Huawei Technologies Co., Ltd.

Equivalent syntax (using **WHERE IN**):

```
SELECT name, courseId FROM student_info WHERE courseId IN (SELECT courseId FROM course_info);
```

Equivalent syntax (using **WHERE EXISTS**):

```
SELECT name, courseId FROM student_info WHERE EXISTS (SELECT 1 FROM course_info WHERE
student_info.courseId = course_info.courseId);
```

13.4.8 NON-EQUI JOIN

Function

In a NON-EQUI JOIN, multiple tables are joined using NON-EQUAL JOIN values, and the result set satisfying the inequality condition is returned.

Syntax

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

Parameters

Parameter	Description
attr_expr_list	The list of query column expressions, specifying the columns to be returned.
table_reference	The table name or subquery to be joined. Must be an existing table. Otherwise, an error is reported.
JOIN	The JOIN keyword. Can be **INNER JOIN**, **LEFT OUTER JOIN**, etc.
non_equi_join_condition	The non-equi join condition. Uses inequality operators (such as **<>**, **<**, **>**, **<=**, **>=**) instead of the equals sign for joining.

Precautions

- The tables to be joined must already exist. Otherwise, an error is reported.
- The join conditions for NON-EQUI JOINS are all inequality conditions, which is different from EQUI-JOINS (which use **=** for comparison).
- NON-EQUI JOINS may result in a large result set. Use with caution.

Examples

Return all combinations of student name pairs from the **student_info_1** and **student_info_2** tables, excluding name pairs with the same name.

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

Example of a NON-EQUI JOIN using a range condition:

```
SELECT a.name, b.name FROM student_info_1 a JOIN student_info_2 b ON a.score > b.score;
```

 Copyright © Huawei Technologies Co., Ltd.

13.5 Clauses

13.5.1 FROM

Function

The **FROM** clause is used to specify the data source for a query. Spark 4.0 enhances the **FROM** clause, supporting **LATERAL** subqueries, the **TABLE** keyword, **VALUES** row constructors, and other data source forms.

Syntax

```
FROM table_reference [, ...]
```

Where **table_reference** is:

```
{ table_name [ TABLE ] [ alias ]
  | join_table
  | LATERAL ( subquery ) [ alias ]
  | ( subquery ) [ alias ]
  | VALUES ( { expression } [, ...] ) [, ...] [ alias ]
  }
```

join_table is:

```
table_reference [ join_type ] JOIN table_reference [ ON join_condition | USING ( column [, ...] ) ]
```

join_type is:

```
{ INNER | CROSS | LEFT [ OUTER ] | RIGHT [ OUTER ] | FULL [ OUTER ] | LEFT SEMI | LEFT ANTI }
```

Parameters

Parameter	Description
table_name	The table name, specifying the table to be queried. The **TABLE** keyword can be used to reference a table value.
alias	Specifies an alias for a table or subquery. A subquery must specify an alias.
LATERAL (subquery)	A LATERAL subquery allows the subquery to reference columns from tables that appear earlier in the same **FROM** clause. Enhanced support in Spark 4.0.
(subquery)	A nested subquery. The subquery result serves as an intermediate temporary table. A subquery must specify an alias.
VALUES	A row constructor. Directly constructs row data as a data source.
join_type	The join type, including **INNER**, **CROSS**, **LEFT OUTER**, **RIGHT OUTER**, **FULL OUTER**, **LEFT SEMI**, and **LEFT ANTI**.
ON join_condition	The JOIN condition.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
USING (column)	A shorthand for performing a JOIN using columns with the same name.

Precautions

- The table being queried must already exist. Otherwise, an error will occur.
- In a **FROM** nested subquery, the subquery must specify an alias.
- A **LATERAL** subquery can reference columns from tables to its left in the same **FROM** clause. A regular subquery cannot reference outer columns.

Examples

Subquery as a data source:

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

LATERAL subquery referencing an outer column:

```
SELECT * FROM orders o,   LATERAL (SELECT * FROM order_items WHERE order_id = o.id) items;
```

VALUES constructing data:

```
SELECT * FROM VALUES (1, 'a'), (2, 'b'), (3, 'c') AS t(id, name);
```

LATERAL VIEW with explode:

```
SELECT name, course FROM student   LATERAL VIEW EXPLODE(courses) t AS course;
```

13.5.2 OVER

Function

The **OVER** clause is used with window functions to define the partitioning, ordering, and range of a window. A window function calculates a value for each row based on its window, unlike regular aggregate functions which merge multiple rows into one.

Syntax

```
window_func ( [ expression [, ...] ] ) OVER
 ( [ PARTITION BY expression [, ...] ]
   [ ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] ]
   [ window_frame ] )
```

Where **window_frame** is:

```
{ ROWS | RANGE } BETWEEN frame_start AND frame_end
```

frame_start and **frame_end** are:

```
{ CURRENT ROW
   | UNBOUNDED PRECEDING
   | UNBOUNDED FOLLOWING
   | offset PRECEDING
   | offset FOLLOWING }
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Description
window_func	The window function, including aggregate window functions (**SUM**, **AVG**, **COUNT**, **MAX**, **MIN**, etc.) and ranking window functions (**RANK**, **DENSE_RANK**, **ROW_NUMBER**, **NTILE**, **LEAD**, **LAG**, **FIRST_VALUE**, **LAST_VALUE**, **NTH_VALUE**, etc.).
PARTITION BY	Partitions data by one or more expressions. The window function is calculated independently within each partition. Similar to **GROUP BY**, but does not merge rows.
ORDER BY	Determines the order for window function calculation. **ASC** or **DESC** can be used to specify ascending or descending order.
ROWS	A physical window. Defines the window range based on row number offset.
RANGE	A logical window. Defines the window range based on the value offset of the **ORDER BY** expression.
CURRENT ROW	The current row.
UNBOUNDED PRECEDING	The window start, from the first row of the partition.
UNBOUNDED FOLLOWING	The window end, to the last row of the partition.
offset PRECEDING	Moves **offset** rows (**ROWS**) or values (**RANGE**) forward from the current row.
offset FOLLOWING	Moves **offset** rows (**ROWS**) or values (**RANGE**) backward from the current row.
NULLS FIRST	Places NULL values first.
NULLS LAST	Places NULL values last.

Window Range Examples

Window Range	Description
ROWS BETWEEN CURRENT ROW AND CURRENT ROW	The window contains only the current row.
ROWS BETWEEN 3 PRECEDING AND 5 FOLLOWING	The window spans from 3 rows before to 5 rows after the current row.
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	The window spans from the beginning of the partition to the current row (default window).

 Copyright © Huawei Technologies Co., Ltd.

Window Range	Description
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING	The window spans from the current row to the end of the partition.
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	The window covers the entire partition.

Differences Between ROWS and RANGE

- **ROWS** is a physical window, calculated based on row number offset, independent of the value of the current row.
- **RANGE** is a logical window, calculated based on the value offset of the **ORDER BY** expression, including all rows whose values fall within the range.

Precautions

- The **OVER** clause includes three parts: **PARTITION BY**, **ORDER BY**, and the window range. They can be used in combination.
- When **ORDER BY** is not specified, the window defaults to covering the entire partition.
- When the window range is not specified, the default window is **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**.
- When the **OVER** clause is empty, the window represents the entire table.

Examples

Cumulative count:

```
SELECT id, COUNT(id) OVER (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
  FROM over_test;
```

Partitioned ranking:

```
SELECT name, class_id, score,
  RANK() OVER (PARTITION BY class_id ORDER BY score DESC) AS class_rank
  FROM student;
```

Moving average:

```
SELECT date, amount,
  AVG(amount) OVER (ORDER BY date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving_avg
  FROM sales;
```

Using a window alias (Spark 4.0):

```
SELECT name, score,
  RANK() OVER w AS rnk,
  SUM(score) OVER w AS total
  FROM student
  WINDOW w AS (PARTITION BY class_id ORDER BY score DESC);
```

 Copyright © Huawei Technologies Co., Ltd.

13.5.3 WHERE

Function

The **WHERE** clause is used to specify filter conditions, returning only the rows that satisfy the conditions. Spark 4.0 enhances the subquery capabilities within **WHERE**, supporting **LATERAL** subqueries and correlated subqueries.

Syntax

```
SELECT [ ALL | DISTINCT ] projectItem [, ...] FROM table_reference [, ...]
  WHERE boolean_expression
```

Where **boolean_expression** can include subqueries:

```
{ expression operator ( subquery )
  | [ NOT ] EXISTS ( subquery )
  | expression [ NOT ] IN ( subquery | value [, ...] ) }
```

Parameters

Parameter	Description
ALL	Returns all rows, including duplicate rows. This is the default option.
DISTINCT	Removes duplicate rows from the result set.
boolean_expression	The filter condition expression, which evaluates to a boolean value.
operator	A comparison operator, including **=**, **!=**, **<**, **>**, **<=**, **>=**, etc.
IN / NOT IN	Determines if a value is in the subquery result or value list. When using a subquery, the subquery must return a single column.
EXISTS / NOT EXISTS	Determines if the subquery returns any rows. **EXISTS** is true when the subquery returns at least one row. The subquery can reference columns from the outer query (correlated subquery).

Precautions

- The table being queried must already exist. Otherwise, an error will occur.
- When using **IN** or **NOT IN**, the subquery must return a single column.
- When using **EXISTS** or **NOT EXISTS**, the subquery can reference columns from the outer query to form a correlated subquery.

Examples

Basic condition filtering:

```
SELECT name, score FROM student WHERE score > 90;
```

Subquery as a filter condition:

 Copyright © Huawei Technologies Co., Ltd.

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

Using an **IN** subquery:

```
SELECT name FROM student_info
  WHERE courseId IN (SELECT courseId FROM course_info WHERE credits > 3);
```

Using an **EXISTS** correlated subquery:

```
SELECT name FROM student_info s
  WHERE EXISTS (SELECT 1 FROM course_info c WHERE c.courseId = s.courseId AND c.courseName =
'Biology');
```

Using **NOT EXISTS**:

```
SELECT name FROM student_info s
  WHERE NOT EXISTS (SELECT 1 FROM course_info c WHERE c.courseId = s.courseId);
```

13.5.4 HAVING

Function

The **HAVING** clause is used to filter the results after a **GROUP BY** operation. Unlike **WHERE**, **HAVING** can use aggregate functions as filter conditions. Spark 4.0 enhances the subquery capabilities within the **HAVING** clause.

Syntax

```
SELECT [ ALL | DISTINCT ] projectItem [, ...] FROM table_reference
  GROUP BY { expression [, ...] | GROUPING SETS (...) | ROLLUP (...) | CUBE (...) }
  HAVING boolean_expression
```

Parameters

Parameter	Description
ALL	Returns all rows, including duplicate rows. This is the default option.
DISTINCT	Removes duplicate rows from the result set.
GROUP BY	Groups by the specified expression. Supports **GROUPING SETS**, **ROLLUP**, and **CUBE** extensions.
HAVING	Filters the grouped results. Conditions can use aggregate functions and subqueries.
boolean_expression	The filter condition expression, typically containing aggregate functions. Can include subqueries.

Precautions

- The table being queried must already exist. Otherwise, an error will occur.
- The position of the subquery and the aggregate function in the **HAVING** clause cannot be swapped. The aggregate function can appear on either side of the comparison operator (e.g., **AVG(score) > 80** or **80 < AVG(score)**), and their positions are interchangeable.

 Copyright © Huawei Technologies Co., Ltd.

- Difference between **HAVING** and **WHERE**: **WHERE** filters rows before grouping, while **HAVING** filters groups after grouping.

Examples

Basic **HAVING** filtering:

```
SELECT class_id, AVG(score) AS avg_score FROM student
  GROUP BY class_id
  HAVING AVG(score) > 80;
```

Subquery as a **HAVING** condition:

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

Used with **ROLLUP**:

```
SELECT group_id, job, SUM(salary) FROM group_test
  GROUP BY ROLLUP (group_id, job)
  HAVING SUM(salary) > 10000;
```

13.5.5 Multi-Level Nested Subquery

Function

A multi-level nested subquery refers to nesting subqueries within subqueries. That is, using another **SELECT** statement as a data source within the **FROM** clause of a **SELECT** statement. This can be nested across multiple levels.

Syntax

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

LATERAL subquery syntax (supported by Spark 4.0):

```
SELECT attr_expr FROM table_reference
  LATERAL VIEW OUTER explode(array_col) alias AS col_name;

SELECT attr_expr FROM table_reference,
  LATERAL (SELECT ... FROM ... WHERE ...) AS alias;
```

Parameters

Parameter	Description
attr_expr	The query column expression, specifying the columns to be returned.
alias	The alias for the subquery. In a nested query, each subquery must be assigned an alias. Otherwise, an error is reported.
LATERAL VIEW	The **LATERAL VIEW** keyword is used to expand a column in a table into multiple rows. It is often used with table-generating functions such as **explode** and **json_tuple**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
LATERAL (subquery)	A LATERAL subquery allows referencing columns from preceding tables in the **FROM** clause. Newly supported in Spark 4.0.
OUTER	An optional keyword. When the result of a **LATERAL VIEW** expansion is empty, the original row is retained and the expanded column is set to **NULL**.

Precautions

- The table being queried must already exist. Otherwise, an error will occur.
- In a nested query, an alias must be specified for the subquery. Otherwise, an error is reported.
- An alias must be defined before it is referenced. It is recommended that aliases be unique.
- A **LATERAL** subquery can reference columns from tables that appear before it in the **FROM** clause.

Examples

Through three levels of subqueries, ultimately return the **name** field from **user_info**.

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

LATERAL VIEW example, expanding an array column into multiple rows:

```
SELECT name, course
FROM student_info
LATERAL VIEW OUTER explode(courses) courses AS course;
```

LATERAL subquery example (Spark 4.0):

```
SELECT o.order_id, s.total
FROM orders o,
  LATERAL (SELECT SUM(amount) AS total FROM order_items i WHERE i.order_id = o.order_id) s;
```

13.6 Alias

13.6.1 Column Alias

Function

Assigns an alias to a column.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

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

Keywords

- **alias**: Used to assign an alias to a field name in **attr_expr**.
- **AS**: Whether this keyword is added does not affect the result.

Precautions

- The table being queried must already exist. Otherwise, an error will occur.
- The alias must be named before it is used. Otherwise, an error will occur. Additionally, you are advised to avoid duplicate names.

Example

First, obtain a result through a subquery **SELECT name AS n FROM simple_table WHERE score > 90**. The alias **n** assigned to **name** in the subquery can be directly used in the outer **SELECT** statement.

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

13.7 Set Operations

13.7.1 UNION

Function

UNION returns the union of the results of multiple queries. **UNION** removes duplicate rows by default, while **UNION ALL** retains duplicate rows.

Syntax

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

Parameters

Parameter	Description
UNION	Set union operation. Merges the results of multiple queries. Removes duplicate rows by default.
ALL	When the **ALL** keyword is added, duplicate rows are retained (i.e., no deduplication). The performance of **UNION ALL** is typically better than **UNION** because no deduplication operation is required.

Precautions

- Each **SELECT** statement must return the same number of columns.

 Copyright © Huawei Technologies Co., Ltd.

- The data types of corresponding columns must be compatible.
- The column names of the result set are determined by the first **SELECT** statement.
- **UNION** removes duplicates by default. **UNION ALL** does not. In scenarios where deduplication is not required, you are advised to use **UNION ALL** for better performance.
- Parentheses can be used to control the precedence of multiple set operations (**UNION**, **INTERSECT**, **EXCEPT**) (Spark 4.0).

Examples

UNION with deduplication:

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

UNION ALL without deduplication:

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

Using parentheses to control precedence (Spark 4.0):

```
(SELECT * FROM student_1 UNION ALL SELECT * FROM student_2)
  INTERSECT SELECT * FROM student_3;
```

Multi-table UNION:

```
SELECT name FROM student_1
  UNION ALL
SELECT name FROM student_2
  UNION ALL
SELECT name FROM student_3;
```

13.7.2 INTERSECT

Function

INTERSECT returns the intersection of the results of multiple queries, i.e., rows that exist in all query results. **INTERSECT** removes duplicate rows by default, while **INTERSECT ALL** retains duplicate rows.

Syntax

```
select_statement INTERSECT [ ALL ] select_statement;
```

Parameters

Parameter	Description
INTERSECT	Set intersection operation. Returns rows that exist in both the left and right query results.
ALL	When the **ALL** keyword is added, duplicate rows are retained. Without **ALL**, duplicate rows are removed by default.

 Copyright © Huawei Technologies Co., Ltd.

Precautions

- Both **SELECT** statements must return the same number of columns.
- The data types of corresponding columns must be compatible.
- Spark 4.0 supports the **INTERSECT ALL** syntax, which retains duplicate rows.
- Parentheses can be used to control the precedence of multiple set operations (**UNION**, **INTERSECT**, **EXCEPT**).

Examples

Basic INTERSECT intersection (with deduplication):

```
SELECT * FROM student_1 INTERSECT SELECT * FROM student_2;
```

INTERSECT ALL retaining duplicate rows (Spark 4.0):

```
SELECT * FROM student_1 INTERSECT ALL SELECT * FROM student_2;
```

Using parentheses to control precedence (Spark 4.0):

```
(SELECT * FROM student_1 INTERSECT SELECT * FROM student_2)  UNION SELECT * FROM student_3;
```

13.7.3 EXCEPT

Function

EXCEPT returns the set difference of two query results, i.e., rows that exist in the first query result but not in the second. **EXCEPT** removes duplicate rows by default, while **EXCEPT ALL** retains duplicate rows.

Syntax

```
select_statement EXCEPT [ ALL ] select_statement;
```

Parameters

Parameter	Description
EXCEPT	Set difference operation. Returns rows that exist in the left query result but not in the right query result.
ALL	When the **ALL** keyword is added, duplicate rows are retained. Without **ALL**, duplicate rows are removed by default.

Precautions

- Each **SELECT** statement must return the same number of columns.
- The data types of corresponding columns must be compatible.
- Spark 4.0 supports the **EXCEPT ALL** syntax, which retains duplicate rows.
- Parentheses can be used to control the precedence of multiple set operations (**UNION**, **INTERSECT**, **EXCEPT**).

 Copyright © Huawei Technologies Co., Ltd.

Examples

Basic EXCEPT difference (with deduplication):

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

EXCEPT ALL retaining duplicate rows (Spark 4.0):

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

Using parentheses to control precedence (Spark 4.0):

```
(SELECT * FROM student_1 EXCEPT SELECT * FROM student_2)
  UNION SELECT * FROM student_3;
```

13.8 CASE...WHEN

Function

CASE...WHEN is a conditional expression used to return different values based on conditions. Spark SQL supports two forms: the simple **CASE** function and the **CASE** search function.

Syntax

Simple CASE function: Jumps to the corresponding **result1** based on the match between **expression** and **value1**.

```
CASE expression
   WHEN value1 THEN result1
   [ WHEN value2 THEN result2 ] ...
   [ ELSE else_result ]
 END
```

CASE search function: Evaluates **condition1** for each **WHEN** clause in the specified order. Returns **result1** for the first **condition1** that evaluates to **TRUE**.

```
CASE
  WHEN condition1 THEN result1
  [ WHEN condition2 THEN result2 ] ...
  [ ELSE else_result ]
END
```

Parameters

Parameter	Description
expression	The expression being compared in the simple **CASE**.
WHEN value	In the simple **CASE**, returns the corresponding **THEN** result when **expression** equals **value**.
WHEN condition	In the **CASE** search function, returns the corresponding **THEN** result when **condition** is true.
THEN result	The result returned when the condition is met. The types of all **THEN** results should be consistent.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description
ELSE else_result	The result returned when none of the **WHEN** conditions are met. If **ELSE** is omitted, **NULL** is returned.

Precautions

- The simple **CASE** function uses equality comparison (=), while the **CASE** search function supports any Boolean condition.
- The **CASE** expression evaluates conditions from top to bottom and returns the **THEN** result of the first satisfied condition. Subsequent conditions are not evaluated.
- The types of all **THEN** and **ELSE** return values should be consistent or implicitly convertible.
- **CASE** expressions can be used in any place where an expression is needed, such as the **SELECT** list, **WHERE** conditions, **ORDER BY**, **GROUP BY**, etc.

Examples

- **Simple CASE function:**

  Returns the **name** field from the **student** table and a string corresponding to **class_id**.

```
SELECT name,
  CASE class_id
    WHEN 1 THEN 'Class A'
    WHEN 2 THEN 'Class B'
    WHEN 3 THEN 'Class C'
    ELSE 'Other'
  END AS class_name
  FROM student;
```

- **CASE search function:**

  Queries the **student** table, returning the **name** field and a result corresponding to **score**. Returns **EXCELLENT** if **score >= 90**, **GOOD** if **80 < score < 90**, and **BAD** otherwise.

```
SELECT name,
 CASE
  WHEN score >= 90 THEN 'EXCELLENT'
  WHEN 80 < score AND score < 90 THEN 'GOOD'
  ELSE 'BAD'
 END AS level
 FROM student;
```

- **CASE used in ORDER BY:**

```
SELECT name, score FROM student
  ORDER BY CASE WHEN score >= 60 THEN 0 ELSE 1 END, score DESC;
```

- **CASE used in aggregation:**

```
SELECT class_id,
  COUNT(CASE WHEN score >= 90 THEN 1 END) AS excellent_count,
  COUNT(CASE WHEN score < 60 THEN 1 END) AS fail_count
  FROM student
  GROUP BY class_id;
```

 Copyright © Huawei Technologies Co., Ltd.

13.9 Pipe Syntax (|>)

Function

The Pipe syntax (pipeline operator **|>**) is a SQL syntax extension in Spark 4.0 that allows organizing SQL queries in a pipeline fashion. The Pipe syntax changes the query style from nested function calls to a top-to-bottom linear flow, making complex queries easier to read and write.

The Pipe syntax is fully compatible with existing SQL syntax and can be mixed in any query.

Syntax

```
FROM table_source
|> WHERE condition
|> SELECT select_expr [, ...]
|> ORDER BY col_name [ASC|DESC] [, ...]
|> LIMIT num_rows
```

Parameters

Operator	Syntax	Description
FROM / TABLE	**FROM** or **TABLE**	Starts from the source table, returning all rows.
SELECT	|> SELECT [[AS] alias], ...	Computes expressions and generates new columns (does not support aggregate functions; use **AGGREGATE** for aggregation).
EXTEND	|> EXTEND [[AS] alias], ...	Appends new columns to the input table.
SET	|> SET col_name = expression [, col_name = expression, ...]	Replaces the values of existing columns.
DROP	|> DROP , ...	Deletes specified columns.
AS	|> AS	Introduces a new alias for the input table.
WHERE	|> WHERE	Filters rows (no separate **HAVING** or **QUALIFY** needed).

 Copyright © Huawei Technologies Co., Ltd.

Operator	Syntax	Description
LIMIT	|> [LIMIT] [OFFSET]	Limits the number of rows returned and skips rows.
AGGREGATE	|> AGGREGATE <agg_expr> GROUP BY <group_expr>	Performs aggregation (full table or grouped).
JOIN	|> [LEFT|RIGHT|FULL|CROSS|SEMI|ANTI] JOIN ON	Performs a join operation.
ORDER BY	|> ORDER BY [ASC|DESC], ...	Performs sorting.
UNION ALL	|> UNION ALL	Merges result sets.
TABLESAMPLE	|> TABLESAMPLE ({ROWS|PERCENT})	Performs sampling.
PIVOT	|> PIVOT (agg_expr FOR col IN (val1, ...))	Rotates rows to columns.
UNPIVOT	|> UNPIVOT (value_col FOR key_col IN (col1, ...))	Rotates columns to rows.

Precautions

- The Pipe syntax starts with a **FROM** clause.
- Each **|>** operator processes the output of the previous step, forming a pipeline chain.
- The **WHERE** operator can be used anywhere. No separate **HAVING** or **QUALIFY** is needed.
- The Pipe syntax does not change the semantics of the query; it only provides a different way of writing it.
- The **EXTEND**/**JOIN** operators may trigger a **toAttribute on unresolved object** internal error on simple inline queries. You are advised to use a sub-table alias approach like **SELECT * FROM (subquery) AS alias |> ...**.
- The **|> SET** operator only supports single-segment keys (e.g., **key_name**), not multi-segment configuration keys (e.g., **spark.sql.ansi.enabled**).

Example 1: Basic Pipeline Query

```
FROM employees
|> WHERE department = 'Engineering'
|> SELECT name, salary
|> ORDER BY salary DESC
|> LIMIT 10;
```

Equivalent to:

```
SELECT name, salary
FROM employees
```

 Copyright © Huawei Technologies Co., Ltd.

```
WHERE department = 'Engineering'
ORDER BY salary DESC
LIMIT 10;
```

Example 2: Using EXTEND to Add a Computed Column

```
VALUES (0), (1) tab(col)
|> EXTEND col * 2 AS result;
```

Result:

col	result
0	0
1	2

Example 3: Using SET to Modify Column Values

```
VALUES (0), (1) tab(col)
|> SET col = col * 2;
```

Result:

col
0
2

Example 4: Using AGGREGATE for Grouped Aggregation

```
-- Full table aggregation
VALUES (0), (1) tab(col)
|> AGGREGATE COUNT(col) AS count;
-- Grouped aggregation
VALUES (0, 1), (0, 2) tab(col1, col2)
|> AGGREGATE COUNT(col2) AS count GROUP BY col1;
```

Example 5: Using JOIN

```
SELECT 0 AS a, 1 AS b
|> AS lhs
|> JOIN VALUES (0, 2) rhs(a, b) ON (lhs.a = rhs.a);
```

Example 6: Complex Pipeline (TPC-H Q13 Rewritten)

Standard SQL:

```
SELECT c_count, COUNT(*) AS custdist
FROM
  (SELECT c_custkey, COUNT(o_orderkey) c_count
   FROM customer
   LEFT OUTER JOIN orders ON c_custkey = o_custkey
     AND o_comment NOT LIKE '%unusual%packages%' GROUP BY c_custkey
   ) AS c_orders
GROUP BY c_count
ORDER BY custdist DESC, c_count DESC;
```

 Copyright © Huawei Technologies Co., Ltd.

Pipeline syntax:

```
FROM customer
|> LEFT OUTER JOIN orders ON c_custkey = o_custkey    AND o_comment NOT LIKE '%unusual%packages
%'
|> AGGREGATE COUNT(o_orderkey) c_count    GROUP BY c_custkey
|> AGGREGATE COUNT(*) AS custdist    GROUP BY c_count
|> ORDER BY custdist DESC, c_count DESC;
```

Core Features

Independence and Interoperability

- Each pipeline operator receives an input table and independently operates on the rows.
- For any valid chain of **N** pipeline operators, the first **M** operators (**M ≤ N**) also form a valid query.
- Pipeline operators can be appended to any valid standard SQL query.
- Table subqueries can be written using standard SQL syntax or pipeline syntax.
- Views, DDL, and DML statements can all contain pipeline syntax queries.

Projection operations

- **SELECT**: Computes new expressions and generates a new table.
- **EXTEND**: Appends new columns while retaining the original columns.
- **SET**: Replaces the values of existing columns.
- **DROP**: Deletes specified columns.

Aggregation operations

- **AGGREGATE** combines the functionality of **GROUP BY** + **SELECT**, eliminating the need to repeat expressions.
- Grouping expressions support specifying aliases directly in the **GROUP BY** clause.
- The output columns automatically include the grouping columns and the aggregate columns.

 Copyright © Huawei Technologies Co., Ltd.

14 Exporting Query Results

Function

INSERT OVERWRITE DIRECTORY is used to write query results directly to a specified directory. It supports storage in CSV, Parquet, ORC, JSON, and Avro formats.

Syntax

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

Keywords

- **USING**: Specifies the storage format.
- **OPTIONS**: A list of properties for the export, optional.

Parameters

Table 14-1 INSERT OVERWRITE DIRECTORY parameter descriptions

Parameter	Description
path	The OBS path to which the query results will be written.
file_format	The file format for writing. Supports CSV, Parquet, ORC, JSON, and Avro formats.

Precautions

- By configuring the **spark.sql.shuffle.partitions** parameter, the number of files inserted into an OBS bucket for non-managed tables can be set. Additionally, to avoid data skew, **distribute by rand()** can be added after the INSERT statement to increase the concurrency of the processing job. For example:
  `INSERT INTO TABLE table_target SELECT * FROM table_source DISTRIBUTE BY cast(rand() * N AS INT);`
- When the configuration is **OPTIONS('DELIMITER'=',')**, the delimiter can be specified. The default value is **,**.

 Copyright © Huawei Technologies Co., Ltd.

For CSV data, the following delimiters are supported:

- Tab character, e.g., **'DELIMITER'='\t'**.
- Specifying a delimiter using a Unicode encoding, e.g., **'DELIMITER'='\u0001'**.
- Single quote (`'`), which must be enclosed in double quotes (" "), e.g., **'DELIMITER'= "'"**.

Example

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

 Copyright © Huawei Technologies Co., Ltd.

15 Built-In Functions

15.1 Date Functions
15.2 String Functions
15.3 Mathematical Functions
15.4 Aggregate Functions
15.5 Window Functions
15.6 Other Functions

15.1 Date Functions

15.1.1 add_months

The **add_months** function is used to calculate the date obtained by adding a specified number of months to a date value. That is, the date that is **num_months** months after **start_date**.

Syntax

```
add_months(string start_date, int num_months)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 15-1 Parameter descriptions

Parameter	Mandatory	Type	Description
start_date	Yes	DATE or STRING	The start date. Supports the following formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
num_months	Yes	INT	The number of months to add.

Return Values

Returns the date obtained by adding **num_months** months to the **start_date**. The return value format is **yyyy-mm-dd**.

Returns a date value of type DATE.

NOTE

- If **start_date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **start_date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **start_date** value is **NULL**, an error is returned.
- If the **num_months** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
  ```
  select add_months('2023-02-26',3);
  ```
  Returns **2023-05-26**.
- Example 2
  ```
  select add_months('2023-02-14 21:30:00',3);
  ```
  Returns **2023-05-14**.
- Example 3
  ```
  select add_months('2023-08-15 20:00:00',null);
  ```
  Returns **NULL**.

15.1.2 current_date

The **current_date** function is used to return the current date value. The return value format is **yyyy-mm-dd**.

Similar function: **15.1.15 getdate**. The **getdate** function returns the current system time. The return value format is **yyyy-mm-dd hh:mi:ss**.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
current_date()
```

Parameters

None

Return Values

Returns a date value of type DATE, in the format **yyyy-mm-dd**.

Example Code

```
select current_date();
```

Returns **2023-08-16**.

15.1.3 current_timestamp

The **current_timestamp** function is used to return the current timestamp.

Syntax

```
current_timestamp()
```

Parameters

None

Return Values

Returns a timestamp of type TIMESTAMP.

Example Code

```
select current_timestamp();
```

Returns **1692002816300**.

15.1.4 date_add

The **date_add** function is used to calculate the date obtained by incrementing the **start_date** by the specified number of **days**.

To obtain a date with a specified offset from the current date, it can be used together with **15.1.2 current_date** or **15.1.15 getdate**.

Note that the **date_add** function has the opposite logic to **15.1.6 date_sub**.

Syntax

```
date_add(string startdate, int days)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 15-2 Parameter descriptions

Parameter	Mandatory	Type	Description
start_date	Yes	DATE or STRING	The start date. Supports the following formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
days	Yes	BIGINT	The number of days to add. • If **days** is greater than 0, days are added. • If **days** is less than 0, days are subtracted. • If **days** equals 0, 0 days are added and the date remains unchanged. • If the **days** value is **NULL**, **NULL** is returned.

Return Values

Returns a date value of type DATE, in the format **yyyy-mm-dd**.

NOTE

- If **start_date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **start_date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **start_date** value is **NULL**, **NULL** is returned.
- If the **days** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
  ```
  select date_add('2023-02-28 00:00:00', 1);
  ```
  Returns **2023-03-01**. Adding one day results in a date beyond the last day of February of that year, so the actual value is the first day of the next month.
- Example 2
  ```
  select date_add(date '2023-02-28', -1);
  ```
  Returns **2023-02-27**. Subtracts one day.
- Example 3
  ```
  select date_add('2023-02-28 00:00:00', 20);
  ```
  Returns **2023-03-20**.

 Copyright © Huawei Technologies Co., Ltd.

- Example 4
  ```
  select date_add(getdate(),-1);
  ```
  Assuming the current time is **2023-08-14 16:00:00**, returns **2023-08-13**.
- Example 5
  ```
  select date_add('2023-02-28 00:00:00', null);
  ```
  Returns **NULL**.

15.1.5 dateadd1

The **dateadd1** function is used to modify the **date** value by the specified **delta** according to the specified unit **datepart**.

To obtain a date with a specified offset from the current date, it can be used together with **15.1.2 current_date** or **15.1.15 getdate**.

Syntax

```
dateadd1(string date, bigint delta, string datepart)
```

Parameters

Table 15-3 Parameter descriptions

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The start date. Supports the following formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
delta	Yes	BIGINT	The modification magnitude.
datepart	Yes	STRING	The time unit for modification. The **datepart** parameter supports extended date formats: year, month or mon, day, and hour. • **yyyy** indicates year. • **mm** indicates month. • **dd** indicates day. • **hh** indicates hour. • **mi** indicates minute. • **ss** indicates second.

Return Values

Returns a date value of type STRING.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.
- If the **delta** or **datepart** value is **NULL**, **NULL** is returned.

Example Code

Returns **2023-08-15 17:00:00** after one day is added.

```
select dateadd1( '2023-08-14 17:00:00', 1, 'dd');
```

Returns **2025-04-14 17:00:00** after 20 months are added. Month overflow causes the year to increase by 2.

```
select dateadd1('2023-08-14 17:00:00', 20, 'mm');
```

Returns **2023-09-14 17:00:00**.

```
select dateadd1('2023-08-14 17:00:00', 1, 'mm');
```

Returns **2023-09-14**.

```
select dateadd1('2023-08-14', 1, 'mm');
```

Assuming the current time is **2023-08-14 17:00:00**, returns **2023-08-13 17:00:00**.

```
select dateadd1(getdate(),-1,'dd');
```

Returns **NULL**.

```
select dateadd1(date '2023-08-14', 1, null);
```

15.1.6 date_sub

The **date_sub** function is used to decrement the **start_date** by the specified number of **days**.

To obtain a date with a specified offset from the current date, it can be used together with **15.1.2 current_date** or **15.1.15 getdate**.

Note that the **date_sub** function has the opposite logic to **15.1.4 date_add**.

Syntax

```
date_sub(string startdate, int days)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 15-4 Parameter descriptions

Parameter	Mandatory	Type	Description
start_date	Yes	DATE or STRING	The start date. Supports the following formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
days	Yes	INT	The number of days to subtract. • When **days** is greater than 0, the specified number of days is subtracted from **start_date**. • When **days** is less than 0, the specified number of days is added to **start_date**. • When **days** equals 0, 0 days are added and the date remains unchanged. • When the **days** value is **NULL**, **NULL** is returned.

Return Values

Returns a date value of type DATE, in the format **yyyy-mm-dd**.

NOTE

- If **start_date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **start_date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **start_date** value is **NULL**, **NULL** is returned.
- If the **days** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
```
select date_sub('2023-08-14 17:00:00', 2);
```
Returns **2023-08-12** after two days are subtracted.
- Example 2
```
select date_sub(date'2023-08-14', -1);
```
Returns **2023-08-15** after one day is added.
- Example 3
```
select date_sub(getdate(),1);
```

 Copyright © Huawei Technologies Co., Ltd.

Assuming the current time is **2023-08-14 17:00:00**, returns **2023-08-13**.

- Example 4
```
select date_sub('2023-08-14 17:00:00', null);
```
Returns **NULL**.

15.1.7 date_format

The **date_format** function is used to convert a date to a string in the format specified by **format**.

Syntax

```
date_format(string date, string format)
```

Parameters

Table 15-5 Parameter descriptions

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be converted. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
format	Yes	STRING	The target format. The format is a combination of time units representing year, month, day, hour, minute, and second, along with any characters. • **yyyy** indicates year. • **MM** indicates month. • **dd** indicates day. • **HH** indicates hour in 24-hour format. • **hh** indicates hour in 12-hour format. • **mm** indicates minute. • **ss** indicates second.

Return Values

Returns a date of type STRING according to the specified format.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.
- If the **format** value is **NULL**, **NULL** is returned.

Example Code

Returns **2023-08-14**.

```
select date_format('2023-08-14','yyyy-MM-dd');
```

Returns **2023-08**.

```
select date_format('2023-08-14','yyyy-MM')
```

Returns **20230814**.

```
select date_format('2023-08-14','yyyyMMdd')
```

15.1.8 datediff

The **datediff** function is used to calculate the date difference between two times, **date1** and **date2**.

Similar function: **15.1.9 datediff1**. The **datediff1** function calculates the difference between two times, **date1** and **date2**, and expresses the difference in the specified time unit **datepart**.

Syntax

```
datediff(string date1, string date2)
```

Parameters

Table 15-6 Parameter descriptions

Parameter	Mandatory	Type	Description
date1	Yes	DATE or STRING	The minuend date. Formats: yyyy-mm-dd yyyy-mm-dd hh:mi:ss yyyy-mm-dd hh:mi:ss.ff3

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Description
date2	Yes	DATE or STRING	The subtrahend date. Formats: yyyy-mm-dd yyyy-mm-dd hh:mi:ss yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type BIGINT.

NOTE

- If **date1** or **date2** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date1** or **date2** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If **date1** is less than **date2**, the return value is negative.
- If the **date1** or **date2** value is **NULL**, **NULL** is returned.

Example Code

Returns **10**.

```
select datediff('2023-06-30 00:00:00', '2023-06-20 00:00:00');
```

Returns **11**.

```
select datediff(date '2023-05-21', date '2023-05-10');
```

Returns **NULL**.

```
select datediff(date '2023-05-21', null);
```

15.1.9 datediff1

The **datediff1** function is used to calculate the difference between two times, **date1** and **date2**, and expresses the difference in the specified time unit **datepart**.

Similar function: **15.1.8 datediff**. The **datediff** function calculates the date difference between two times, **date1** and **date2**, and does not support specifying the return time unit.

Syntax

```
datediff1(string date1, string date2, string datepart)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 15-7 Parameter descriptions

Parameter	Mandatory	Type	Description
date1	Yes	DATE or STRING	The minuend in the date difference calculation between two times, **date1** and **date2**. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
date2	Yes	DATE or STRING	The subtrahend in the date difference calculation between two times, **date1** and **date2**. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
datepart	Yes	STRING	The time unit to return. The **datepart** parameter supports extended date formats: year, month or mon, day, and hour. • **yyyy** indicates year. • **mm** indicates month. • **dd** indicates day. • **hh** indicates hour. • **mi** indicates minute. • **ss** indicates second.

Return Values

Returns a value of type BIGINT.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **date1** or **date2** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date1** or **date2** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If **date1** is less than **date2**, the return value is negative.
- If the **date1** or **date2** value is **NULL**, **NULL** is returned.
- If the **datepart** value is **NULL**, **NULL** is returned.

Example Code

Returns **14400**.

```
select datediff1('2023-06-30 00:00:00', '2023-06-20 00:00:00', 'mi');
```

Returns **10**.

```
select datediff1(date '2023-06-21', date '2023-06-11', 'dd');
```

Returns **NULL**.

```
select datediff1(date '2023-05-21', date '2023-05-10', null);
```

Returns **NULL**.

```
select datediff1(date '2023-05-21', null, 'dd');
```

15.1.10 datepart1

The **datepart1** function is used to calculate the value from the date that matches the specified time unit **datepart**.

Syntax

```
datepart1(string date,string datepart)
```

Parameters

Table 15-8 Parameter descriptions

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The start date. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Description
datepart	Yes	STRING	The time unit to return. The **datepart** parameter supports extended date formats: year, month or mon, day, and hour. • **yyyy** indicates year. • **mm** indicates month. • **dd** indicates day. • **hh** indicates hour. • **mi** indicates minute. • **ss** indicates second.

Return Values

Returns a value of type BIGINT.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.
- If the **datepart** value is **NULL**, **NULL** is returned.

Example Code

Returns **2023**.

```
select datepart1(date '2023-08-14 17:00:00', 'yyyy');
```

Returns **2023**.

```
select datepart1('2023-08-14 17:00:00', 'yyyy');
```

Returns **59**.

```
select datepart1('2023-08-14 17:59:59', 'mi')
```

Returns **NULL**.

```
select datepart1(date '2023-08-14', null);
```

15.1.11 datetrunc

The **datetrunc** function is used to calculate the date value obtained by truncating the date according to the time unit specified by **datepart**.

The portion before the **datepart** is truncated, and the truncated portion is automatically filled with default values. For details, see **Example Code**.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
datetrunc (string date, string datepart)
```

Parameters

Table 15-9 Parameter descriptions

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The start date. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
datepart	Yes	STRING	The time unit to return. The **datepart** parameter supports extended date formats: year, month or mon, day, and hour. • **yyyy** indicates year. • **mm** indicates month. • **dd** indicates day. • **hh** indicates hour. • **mi** indicates minute. • **ss** indicates second.

Return Values

Returns a date value of type DATE or STRING.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **datepart** value is **NULL**, **NULL** is returned.
- Truncation is performed according to the specified time unit (e.g., **hh** truncates by hour).

Example Code

Example static data

- Example 1:
  ```
  select datetrunc('2023-08-14 17:00:00', 'yyyy');
  ```
  Returns **2023-01-01 00:00:00**.

 Copyright © Huawei Technologies Co., Ltd.

- Example 2:
  ```
  select datetrunc('2023-08-14 17:00:00', 'month');
  ```
  Returns **2023-08-01 00:00:00**.
- Example 3:
  ```
  select datetrunc('2023-08-14 17:00:00', 'DD');
  ```
  Returns **2023-08-14 00:00:00**.
- Example 4:
  ```
  select datetrunc('2023-08-14', 'yyyy');
  ```
  Returns **2023-01-01**.
- Example 5:
  ```
  select datetrunc('2023-08-14 17:11:11', 'hh');
  ```
  Returns **2023-08-14 17:00:00**.
- Example 6:
  ```
  select datetrunc('2023-08-14', null);
  ```
  Returns **NULL**.

15.1.12 day/dayofmonth

The **day** function is used to return the day of the specified date.

Syntax

```
day(string date), dayofmonth(string date)
```

Parameters

Table 15-10 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type INT.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

 Copyright © Huawei Technologies Co., Ltd.

Example Code

- **Example 1**
  ```
  select day('2023-08-01');
  ```
  Returns **1**.
- **Example 2**
  ```
  select day(null);
  ```
  Returns **NULL**.

15.1.13 from_unixtime

The **from_unixtime** function is used to convert a timestamp represented by a numeric UNIX value into a date value.

Syntax

```
from_unixtime(bigint unixtime)
```

Parameters

Table 15-11 Parameter description

Parameter	Mandatory	Type	Description
unixtime	Yes	BIGINT	A timestamp in UNIX format. Represents the timestamp to be converted. The parameter should be filled with the first ten digits of a normal UNIX format timestamp.

Return Values

Converts the timestamp to a time format and returns a date value of type STRING, in the format **yyyy-MM-dd HH:mm:ss**, **yyyyMMddHHmmss.uuuuuu**, or **yyyyMMdd**.

NOTE

If the **unixtime** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
  ```
  select from_unixtime(1692149997);
  ```
  Returns **2023-08-16 09:39:57**.
- Example 2
  ```
  select from_unixtime(NULL);
  ```
  Returns **NULL**.

 Copyright © Huawei Technologies Co., Ltd.

15.1.14 from_utc_timestamp

The **from_utc_timestamp** function is used to convert a UTC timestamp to the local timestamp corresponding to the specified **timezone**.

Syntax

```
from_utc_timestamp(string timestamp, string timezone)
```

Parameters

Table 15-12 Parameter descriptions

Parameter	Mandatory	Type	Description
timestamp	Yes	DATE, TIMESTAMP, or STRING	UTC timestamp. Formats: yyyy-MM-dd yyyy-MM-dd HH:mm:ss yyyy-MM-dd HH:mm:ss.SSS
timezone	Yes	STRING	The target time zone for conversion.

Return Values

Returns a timestamp of type TIMESTAMP.

NOTE

- If **timestamp** is not of type DATE, TIMESTAMP, or STRING, an error is returned with the message "data type mismatch".
- If **timestamp** is of type DATE, TIMESTAMP, or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **timestamp** value is **NULL**, **NULL** is returned.
- If the **timezone** value is **NULL**, **NULL** is returned.

Example Code

Returns **1691978400000** (representing 2023-08-14 10:00:00).

```
select from_utc_timestamp('2023-08-14 17:00:00','PST');
```

Returns **1691917200000000** (representing 2023-08-13 17:00:00).

```
select from_utc_timestamp(date '2023-08-14 00:00:00','PST');
```

Returns **NULL**.

```
select from_utc_timestamp('2023-08-13',null);
```

 Copyright © Huawei Technologies Co., Ltd.

15.1.15 getdate

The **getdate** function is used to return the current system time. The return value format is **yyyy-mm-dd hh:mi:ss**.

Similar function: **15.1.2 current_date**. The **current_date** function returns the current date value. The return value format is **yyyy-mm-dd**.

Syntax

```
getdate()
```

Parameters

None

Return Values

Returns a date value of type STRING, in the format **yyyy-mm-dd hh:mi:ss**.

Example Code

Assuming the current time is **2023-08-10 10:54:00**, returns **2023-08-10 10:54:00**.

```
select getdate();
```

15.1.16 hour

The **hour** function is used to return the hour of the specified time, ranging from 0 to 23.

Syntax

```
hour(string date)
```

Parameters

Table 15-13 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type INT.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
  ```
  select hour('2023-08-10 10:54:00');
  ```
  Returns **10**.
- Example 2
  ```
  select hour('12:00:00');
  ```
  Returns **12**.
- Example 3
  ```
  select hour(null);
  ```
- Returns **NULL**.

15.1.17 isdate

The **isdate** function is used to determine whether a date string can be converted to a date value according to the specified format.

Syntax

```
isdate(string date , string format)
```

Parameters

Table 15-14 Parameter descriptions

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The string to be evaluated. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before computation. The **date** parameter can be in any string format.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Description
format	Yes	STRING	The target date format for conversion. A STRING type constant. Extended date formats are not supported. The format is a combination of time units representing year, month, day, hour, minute, and second, along with any characters. • **yyyy** indicates year. • **mm** indicates month. • **dd** indicates day. • **hh** indicates hour. • **mi** indicates minute. • **ss** indicates second.

Return Values

Returns a value of type BOOLEAN.

NOTE

If the **date** or **format** value is **NULL**, **NULL** is returned.

Example Code

Returns **true**.

```
select isdate('2023-08-10','yyyy-mm-dd');
```

Returns **false**.

```
select isdate(123456789,'yyyy-mm-dd');
```

15.1.18 last_day

The **last_day** function is used to return the last day of the month containing date.

Similar function: **15.1.19 last_day1**. The **last_day1** function returns the last day of the month containing date, truncated to the day, with the hour, minute, and second portion set to 00:00:00.

Syntax

```
last_day(string date)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 15-15 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a date value of type DATE, in the format **yyyy-mm-dd**.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
  ```
  select last_day('2023-08-15');
  ```
  Returns **2023-08-31**.
- Example 2
  ```
  select last_day('2023-08-10 10:54:00');
  ```
  Returns **2023-08-31**.

15.1.19 last_day1

The **last_day1** function is used to return the last day of the month containing date, truncated to the day, with the hour, minute, and second portion set to 00:00:00.

Similar function: **15.1.18 last_day**. The **last_day** function returns the last day of the month containing date. The return value format is **yyyy-mm-dd**.

Syntax

```
last_day1(string date)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 15-16 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a date value of type STRING, in the format **yyyy-mm-dd hh:mi:ss**.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

Example Code

- Returns **2023-08-31**.
```
select last_day1('2023-08-10');
```
- Returns **2023-08-31 00:00:00**.
```
select last_day1('2023-08-10 10:54:00');
```
- Returns **NULL**.
```
select last_day1(null);
```

15.1.20 minute

The **minute** function is used to return the minute of the specified time, ranging from 0 to 59.

Syntax

```
minute(string date)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 15-17 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type INT.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
  ```
  select minute('2023-08-10 10:54:00');
  ```
  Returns **54**.
- Example 2
  ```
  select minute('10:54:00');
  ```
  Returns **54**.
- Example 3
  ```
  select minute(null);
  ```
  Returns **NULL**.

15.1.21 month

The **month** function is used to return the month of the specified time, ranging from 1 to 12.

Syntax

```
month(string date)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 15-18 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. When **date** is of type STRING, it must contain at least **yyyy-mm-dd** and must not contain extra characters. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type INT.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
  ```
  select month('2023-08-10 10:54:00');
  ```
  Returns **8**.
- Example 2
  ```
  select month(null);
  ```
  Returns **NULL**.

15.1.22 months_between

The **months_between** function is used to return the number of months between **date1** and **date2**.

Syntax

```
months_between(string date1, string date2[, roundOff])
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 15-19 Parameter descriptions

Parameter	Mandatory	Type	Description
date1	Yes	DATE or STRING	The minuend. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
date2	Yes	DATE or STRING	The subtrahend. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
roundOff	No	BOOLEAN	Whether to round the result. The default value is **false**. This parameter is currently not effective.

Return Values

Returns a value of type DOUBLE.

NOTE

- If **date1** or **date2** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date1** or **date2** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- When **date1** is later than **date2**, the return value is positive. When **date2** is later than **date1**, the return value is negative.
- When **date1** and **date2** correspond to the last day of their respective months, an integer number of months is returned. Otherwise, the calculation is the number of days from **date1** minus **date2** divided by 31 days, and whether the result is rounded is determined by the **roundOff** parameter.
- If the **date1** or **date2** value is **NULL**, **NULL** is returned.

Example Code

- Returns **0.0563172**.
```
select months_between('2023-08-16 10:54:00', '2023-08-14 17:00:00');
```
- Returns **0.06451613**.
```
select months_between('2023-08-16','2023-08-14');
```
- Returns **NULL**.
```
select months_between('2023-08-16',null);
```

 Copyright © Huawei Technologies Co., Ltd.

- Returns **3.0**.

  Note: May 31, 2025 and February 28, 2025 are the last days of their respective months. Therefore, the integer month **3** is returned.

  ```
  select months_between('2025-05-31','2025-02-28')
  ```

- Returns **3.0**.

  Note: The difference in days between **2025-05-28** and **2025-02-28** is 89 days.

  ```
  select months_between('2025-05-28','2025-02-28')
  ```

15.1.23 next_day

The **next_day** function is used to return the date closest to the specified day of the week after the start date.

Syntax

```
next_day(string start_date, string day_of_week)
```

Parameters

Table 15-20 Parameter descriptions

Parameter	Mandatory	Type	Description
start_date	Yes	DATE or STRING	The start date. When **start_date** is of type STRING, it must contain at least **yyyy-mm-dd** and must not contain extra characters. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
day_of_week	Yes	STRING	The two or three letter abbreviation of the day name, or the full day name. For example, **TU** indicates Tuesday.

Return Values

Returns a date value of type DATE, in the format **yyyy-mm-dd**.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **start_date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **start_date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **start_date** value is **NULL**, **NULL** is returned.
- If the **day_of_week** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
  ```
  select next_day('2023-08-16','TU');
  ```
  Returns **2023-08-22**.
- Example 2
  ```
  select next_day('2023-08-16 10:54:00','TU');
  ```
  Returns **2023-08-22**.
- Example 3
  ```
  select next_day('2023-08-16 10:54:00','WE');
  ```
  Returns **2023-08-23**.
- Example 4
  ```
  select next_day('20230816 20:00:00',null);
  ```
  Returns **NULL**.

15.1.24 quarter

The **quarter** function is used to return the quarter in which the date falls, ranging from 1 to 4.

Syntax

```
quarter(string date)
```

Parameters

Table 15-21 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type INT.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

Example Code

Returns **3**.

```
select quarter('2023-08-16 10:54:00');
```

Returns **3**.

```
select quarter('2023-08-16');
```

Returns **NULL**.

```
select quarter(null);
```

15.1.25 second

The **second** function is used to return the second of the specified time, ranging from 0 to 59.

Syntax

```
second(string date)
```

Parameters

Table 15-22 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type INT.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

Example Code

- Example 1
  ```
  select second('2023-08-16 10:54:36');
  ```
  Returns **36**.
- Example 2
  ```
  select second('10:54:36');
  ```
  Returns **36**.
- Example 3
  ```
  select second(null);
  ```
  Returns **NULL**.

15.1.26 to_char1

The **to_char1** function is used to convert a date to a string according to the specified format.

Syntax

```
to_char1(string date, string format)
```

Parameters

Table 15-23 Parameter descriptions

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Description
format	Yes	STRING	The target date format for conversion. A STRING type constant. Extended date formats are not supported. The format is a combination of time units representing year, month, day, hour, minute, and second, along with any characters. • **yyyy** indicates year. • **mm** indicates month. • **dd** indicates day. • **hh** indicates hour. • **mi** indicates minute. • **ss** indicates second.

Return Values

Returns a value of type STRING.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **format** value is **NULL**, **NULL** is returned.

Example Code

Returns static data example **2023-08*16**.

```
select to_char1('2023-08-16 10:54:36', 'Static data example yyyy-mm*dd');
```

Returns **20230816**.

```
select to_char1('2023-08-16 10:54:36', 'yyyymmdd');
```

Returns **NULL**.

```
select to_char1('Static data example 2023-08-16', 'Static data example yyyy-mm*dd');
```

Returns **NULL**.

```
select to_char1('20230816', 'yyyy');
```

Returns **NULL**.

```
select to_char1('2023-08-16 10:54:36', null);
```

 Copyright © Huawei Technologies Co., Ltd.

15.1.27 to_date

The **to_date** function is used to return the year, month, and day from a time.

Similar function: **15.1.28 to_date1**. The **to_date1** function converts a string in the specified format to a date value, supporting the specification of the target date format.

Syntax

```
to_date(string timestamp)
```

Parameters

Table 15-24 Parameter description

Parameter	Mandatory	Type	Description
timestamp	Yes	DATE or STRING	The time to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a date value of type DATE, in the format **yyyy-mm-dd**.

NOTE

- If **timestamp** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **timestamp** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.

Example Code

Returns **2023-08-16**.

```
select to_date('2023-08-16 10:54:36');
```

Returns **NULL**.

```
select to_date(null);
```

15.1.28 to_date1

The **to_date1** function is used to convert a string in the specified format to a date value.

Similar function: **15.1.27 to_date**. The **to_date** function returns the year, month, and day from a time and does not support specifying the target date format.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
to_date1(string date, string format)
```

Parameters

Table 15-25 Parameter descriptions

Parameter	Mandatory	Type	Description
date	Yes	STRING	The string to be converted. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
format	Yes	STRING	The target date format for conversion. A STRING type constant. Extended date formats are not supported. The **format** is a combination of time units representing year, month, day, hour, minute, and second, along with any characters. • **yyyy** indicates year. • **mm** indicates month. • **dd** indicates day. • **hh** indicates hour. • **mi** indicates minute. • **ss** indicates second.

Return Values

Returns a date string of type STRING in the format **yyyy-mm-dd hh:mi:ss** (including hours, minutes, and seconds).

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.
- If the **format** value is **NULL**, a date value in the format **yyyy-mm-dd** is returned.

Example Code

Returns **2023-08-16 10:54:36**.

 Copyright © Huawei Technologies Co., Ltd.

```
select to_date1('2023-08-16 10:54:36','yyyy-mm-dd hh:mi:ss');
```

Returns **2023-08-16 00:00:00**.

```
select to_date1('2023-08-16','yyyy-mm-dd');
```

Returns **NULL**.

```
select to_date1(null);
```

Returns **2023-08-16**.

```
select to_date1('2023-08-16 10:54:36');
```

15.1.29 to_utc_timestamp

The **to_utc_timestamp** function is used to convert a timestamp corresponding to the specified timezone to a UTC timestamp.

Syntax

```
to_utc_timestamp(string timestamp, string timezone)
```

Parameters

Table 15-26 Parameter descriptions

Parameter	Mandatory	Type	Description
timestamp	Yes	DATE, TIMESTAMP, or STRING	The timestamp to be converted Formats: yyyy-MM-dd yyyy-MM-dd HH:mm:ss yyyy-MM-dd HH:mm:ss.SSS
timezone	Yes	STRING	The target time zone for conversion.

Return Values

Returns a value of type BIGINT.

NOTE

- If **timestamp** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **timestamp** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **timestamp** value is **NULL**, **NULL** is returned.
- If the **timezone** value is **NULL**, **NULL** is returned.

Example Code

Returns **1692028800000**.

 Copyright © Huawei Technologies Co., Ltd.

```
select to_utc_timestamp('2023-08-14 17:00:00','PST');
```

Returns **NULL**.

```
select to_utc_timestamp(null, 'PST');
```

15.1.30 trunc

The **trunc** function is used to truncate a date according to a specific format.

Truncation means returning the default value. The default value for year, month, and day is **01**. The default value for hour, minute, second, and millisecond is **00**.

Syntax

```
trunc(string date, string format)
```

Parameters

Table 15-27 Parameter descriptions

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
format	Yes	STRING	The target date format for conversion. The **format** is a combination of time units representing year, month, day, hour, minute, and second, along with any characters. • **yyyy** indicates year. • **MM** indicates month.

Return Values

Returns a date value of type DATE, in the format **yyyy-mm-dd**.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.
- If the **format** value is **NULL**, **NULL** is returned.

 Copyright © Huawei Technologies Co., Ltd.

Example Code

Returns **2023-08-01**.

```
select trunc('2023-08-16', 'MM');
```

Returns **2023-08-01**.

```
select trunc('2023-08-16 10:54:36', 'MM');
```

Returns **NULL**.

```
select trunc(null, 'MM');
```

15.1.31 trunc_numeric

The **trunc_numeric** function is used to truncate the input number to the specified number of decimal places.

Syntax

```
trunc_numeric(<number>[, bigint<decimal_places>])
```

Parameters

Table 15-28 Parameter descriptions

Parameter	Mandatory	Type	Description
number	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The data to be truncated.
decimal_places	No	BIGINT	The default value is **0**. Truncates to the *N*th decimal place.

Return Values

Returns a value of type DOUBLE or DECIMAL.

NOTE

Return rules are as follows:

- When **number** is of type DOUBLE or DECIMAL, the corresponding type is returned.
- When **number** is of type STRING or BIGINT, a DOUBLE type is returned.
- If **decimal_places** is not of type BIGINT, an error is returned.
- If the **number** value is **NULL**, **NULL** is returned.

Example Code

- Example 1

  Returns **3.141**.

  ```
  select trunc_numeric(3.1415926, 3);
  ```

 Copyright © Huawei Technologies Co., Ltd.

- Example 2

  Returns **3**.

```
select trunc_numeric(3.1415926);
```

15.1.32 unix_timestamp

The **unix_timestamp** function is used to convert a date value to a numeric UNIX format date value.

The function returns the first ten digits of a normal UNIX format timestamp.

Syntax

```
unix_timestamp(string timestamp, string pattern)
```

Parameters

Table 15-29 Parameter descriptions

Parameter	Mandatory	Type	Description
timestamp	No	DATE or STRING	The date value to be converted. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3
pattern	No	STRING	The target format. When **pattern** is empty, the default format is **yyyy-mm-dd hh:mi:ss**. The **format** is a combination of time units representing year, month, day, hour, minute, and second, along with any characters. • **yyyy** indicates year. • **mm** indicates month. • **dd** indicates day. • **hh** indicates hour. • **mi** indicates minute. • **ss** indicates second.

Return Values

Returns a value of type BIGINT.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If the **timestamp** value is **NULL**, **NULL** is returned.
- When both **timestamp** and **pattern** are empty, the timestamp representing the number of seconds from **1970-01-01 00:00:00** to the present is returned.

Example Code

- Returns **1692149997**.
  ```
  select unix_timestamp('2023-08-16 09:39:57')
  ```
- Assuming the current system time is **2023-08-16 10:23:16**, returns **1692152596**.
  ```
  select unix_timestamp();
  ```

15.1.33 weekday

The **weekday** function is used to return the day of the week for the date value (Monday is 0).

Syntax

```
weekday (string date)
```

Parameters

Table 15-30 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type INT.

NOTE

- Monday is considered the first day of the week and returns **0**. Other days increment sequentially, and Sunday returns **6**.
- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

 Copyright © Huawei Technologies Co., Ltd.

Example Code

Returns **2**.

```
select weekday ('2023-08-16 10:54:36');
```

Returns **NULL**.

```
select weekday (null);
```

15.1.34 weekofyear

The **weekofyear** function is used to return the week number of the year for the specified date, ranging from 0 to 53.

Syntax

```
weekofyear(string date)
```

Parameters

Table 15-31 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type INT.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

Example Code

- Returns **33**.
  ```
  select weekofyear('2023-08-16 10:54:36');
  ```
- Returns **NULL**.
  ```
  select weekofyear(null);
  ```

 Copyright © Huawei Technologies Co., Ltd.

15.1.35 year

The **year** function is used to return the year from the specified date.

Syntax

```
year(string date)
```

Parameters

Table 15-32 Parameter description

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	The date to be processed. Formats: • yyyy-mm-dd • yyyy-mm-dd hh:mi:ss • yyyy-mm-dd hh:mi:ss.ff3

Return Values

Returns a value of type INT.

NOTE

- If **date** is not of type DATE or STRING, an error is returned with the message "data type mismatch".
- If **date** is of type DATE or STRING but does not conform to the input format for date values, **NULL** is returned.
- If the **date** value is **NULL**, **NULL** is returned.

Example Code

- Returns **2023**.
  ```
  select year('2023-08-16 10:54:36');
  ```
- Returns **NULL**.
  ```
  select year(null);
  ```

15.2 String Functions

15.2.1 ascii

The **ascii** function is used to return the ASCII code of the first character of the string **str**.

Syntax

```
ascii(string )
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is automatically converted to STRING type before evaluation. For example, **ABC**.

Return Values

Returns a value of type BIGINT.

NOTE

- If **str** is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If **str** is NULL, NULL is returned.

Example Code

- Return the ASCII code of the first character of the string **ABC**. The command example is as follows.

  Returns **65**.

  ```
  select ascii('ABC');
  ```

- The input parameter is NULL. The command example is as follows.

  Returns **NULL**.

  ```
  select ascii(null);
  ```

15.2.2 concat

The **concat** function is used to concatenate arrays or strings.

Syntax

Input is an array: Concatenates all elements from multiple arrays together to generate a new array.

```
concat(array , array [,...])
```

Input is a string: Concatenates multiple strings together to generate a new string.

```
concat(string , string [,...])
```

Parameters

Input is an array

Parameter	Mandatory	Type	Description
a, b	Yes	ARRAY	Arrays. **T** in **array<T>** refers to the data type of the elements in the array. Elements in the array can be of any type. The data types of elements in **a** and **b** must be consistent. NULL values in the array participate in the operation.

Input is a string

Parameter	Mandatory	Type	Description
str1, str2	Yes	STRING	Strings. If the input parameter is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is automatically converted to STRING type before evaluation. Other types will return an error.

Return Values

Returns a value of type ARRAY or STRING.

NOTE

- Returns ARRAY type. If any input array is NULL, the return result is NULL.
- Returns STRING type. If there are no parameters or any parameter is NULL, the return result is NULL.

Example Code

- Concatenate arrays **array(1, 2)** and **array(2, -2)**. The command example is as follows.

  Returns **[1,2,2,-2]**.

  ```
  select concat(array(1, 2), array(2, -2));
  ```

- Concatenate strings **ABC** and **DEF**. The command example is as follows.

  Returns **ABCDEF**.

  ```
  select concat('ABC','DEF');
  ```

- The input is empty. The command example is as follows.

  Returns **NULL**.

  ```
  select concat();
  ```

- Any input string is NULL. The command example is as follows.

  Returns **NULL**.

  ```
  select concat('abc', 'def', null);
  ```

15.2.3 concat_ws

Concatenates multiple strings, separating them with a specified delimiter.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
concat_ws(string <separator>, string <str1>, string <str2>[,...])
```

or

```
concat_ws(string <separator>, array<string> <a>)
```

Parameters

Parameter	Mandatory	Type	Description
separator	Yes	STRING	The delimiter.
str1, str2	Yes	STRING	At least 2 strings must be specified. If the input is of type BIGINT, DECIMAL, DOUBLE, or DATETIME, it is implicitly converted to STRING type before evaluation.
a	Yes	ARRAY	The elements in the array are of type STRING.

Return Values

Returns a value of type STRING.

NOTE

- If **str1** or **str2** is not of type STRING, BIGINT, DECIMAL, DOUBLE, or DATETIME, an error is returned.
- If a parameter (the string to be concatenated) is NULL, that parameter is ignored.
- If there are no input parameters (strings to be concatenated), NULL is returned.

Example Code

Concatenate strings **ABC** and **DEF** using **:**. Return **ABC:DEF**.

```
select concat_ws(':','ABC','DEF');
```

Any input parameter is NULL. Return **avg:18**.

```
select concat_ws(':','avg',null,'18');
```

Concatenate elements in the array **array('name', 'lilei')** using **:**. Return **name:lilei**.

```
select concat_ws(':',array('name', 'lilei'));
```

15.2.4 char_matchcount

The **char_matchcount** function is used to calculate how many characters in **str1** appear in **str2**.

Syntax

```
char_matchcount(string , string )
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
str1, str2	Yes	STRING	The strings to be calculated, **str1** and **str2**.

Return Values

Returns a value of type BIGINT.

NOTE

If **str1** or **str2** is NULL, NULL is returned.

Example Code

Returns **3**.

```
select char_matchcount('abcz','abcde');
```

Returns **NULL**.

```
select char_matchcount(null,'abcde');
```

15.2.5 encode

The **encode** function is used to encode **str** using the encoding method specified by **charset**.

Syntax

```
encode(string , string )
```

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	String to be encoded. Type STRING. If the input is of type BIGINT, DECIMAL, DOUBLE, or DATETIME, it is implicitly converted to STRING type before evaluation.
charset	Yes	STRING	The encoding format. Valid values are: **UTF-8**, **UTF-16**, **UTF-16LE**, **UTF-16BE**, **ISO-8859-1**, **US-ASCII**.

Return Values

Returns a value of type BINARY.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

If **str** or **charset** is NULL, NULL is returned.

Example Code

- Encode the string **abc** using UTF-8 format. The command example is as follows.

  Returns **YWJj**.

  ```
  select encode("abc", "UTF-8");
  ```

- Any input parameter is NULL. The command example is as follows.

  Returns **NULL**.

  ```
  select encode("abc", null);
  ```

15.2.6 find_in_set

The **find_in_set** function is used to find the position of string **str1** in the comma (**,**) separated string **str2**, counting from 1.

Syntax

```
find_in_set(string , string )
```

Parameters

Parameter	Mandatory	Type	Description
str1	Yes	STRING	The string to find.
str2	Yes	STRING	A string separated by commas (`,`).

Return Values

Returns a value of type BIGINT.

NOTE

- When **str1** cannot be matched in **str2** or **str1** contains a comma (**,**), 0 is returned.
- When **str1** or **str2** is NULL, NULL is returned.

Example Code

- Find the position of string **ab** in the string **abc,123,ab,c**. The command example is as follows.

  Returns **3**.

  ```
  select find_in_set('ab', 'abc,123,ab,c');
  ```

 Copyright © Huawei Technologies Co., Ltd.

- Find the position of string **hi** in the string **abc,123,ab,c**. The command example is as follows.

  Returns **0**.

  ```
  select find_in_set('hi', 'abc,123,ab,c');
  ```

- Any input parameter is NULL. The command example is as follows.

  Returns **NULL**.

  ```
  select find_in_set(null, 'abc,123,ab,c');
  ```

15.2.7 get_json_object

The **get_json_object** function is used to parse a JSON object based on the given path. If the JSON object is invalid, **NULL** is returned.

Syntax

```
get_json_object( json_string , path  )
```

Parameters

Parameter	Mandatory	Type	Description
json_string	Yes	STRING	A standard JSON format object, in the format **{Key:Value, Key:Value,...}**.
path	Yes	STRING	A JSONPath expression used to specify the location of the element to extract. The meanings of different characters are as follows: ● **$** indicates the root object. ● **.key** indicates the property of the accessed object. ● **[]**: The subscript operator, used to extract a specific element from a JSON array (counting starts from 0), for example, **$.tags[1]**.

Return Values

Returns a value of type STRING.

NOTE

- If **json** is empty or in an invalid JSON format, NULL is returned.
- If **json** is valid and **path** exists, the corresponding string is returned.

Example Code

- Extract information from the JSON object **src_json.json**. Example data is as follows.

  ```
  jsonString = {"store": {"fruit":[{"weight":8,"type":"apple"},{"weight":9,"type":"pear"}], "bicycle":
  {"price":19.95,"color":"red"} }, "email":"amy@only_for_json_udf_test.net", "owner":"Tony" }
  ```

  Extract the **owner** field information from the above example. Return **Tony**.

 Copyright © Huawei Technologies Co., Ltd.

```
select get_json_object(jsonString, '$.owner');
```

Extract the first array information from the **store.fruit** field. Return **{"weight":8,"type":"apple"}**.

```
select get_json_object(jsonString, '$.store.fruit[0]');
```

Extract information from a non-existent field. Return **NULL**.

```
select get_json_object(jsonString, '$.non_exist_key');
```

- Extract information from an array-type JSON object. The command example is as follows.

  Returns **22**.

  ```
  select get_json_object('{"array":[["a",11],["b",22],["c",33]]}','$.array[1][1]');
  ```

  Returns **["h00","h11","h22"]**.

  ```
  select get_json_object('{"a":"b","c":{"d":"e","f":"g","h":["h00","h11","h22"]},"i":"j"}','$.c.h[*]');
  ```

  Returns **["h00","h11","h22"]**.

  ```
  select get_json_object('{"a":"b","c":{"d":"e","f":"g","h":["h00","h11","h22"]},"i":"j"}','$.c.h');
  ```

  Returns **h11**.

  ```
  select get_json_object('{"a":"b","c":{"d":"e","f":"g","h":["h00","h11","h22"]},"i":"j"}','$.c.h[1]');
  ```

- Extract information from a JSON object containing **.**. The command example is as follows.

  Create a table.

  ```
  create table json_table (id string, json string);
  ```

  Insert data into the table. The key contains **.**.

  ```
  insert into table json_table (id, json) values ("1", "{\"city1\":{\"region\":{\"rid\":6}}}");
  ```

  Insert data into the table. The key does not contain **.**.

  ```
  insert into table json_table (id, json) values ("2", "{\"city1\":{\"region\":{\"rid\":7}}}");
  ```

  Get the value of **rid** and query the key as **city1**. Return **6**. Since it contains **.**, it can only be parsed using **['']**.

  ```
  select get_json_object(json, "$['city1'].region['rid']") from json_table where id =1;
  ```

  Get the value of **rid** and query the key as **city1**. Return **7**. There are two query methods.

  ```
  select get_json_object(json, "$['city1'].region['rid']") from json_table where id =2;  select get_json_object(json, "$.city1.region['rid']") from json_table where id =2;
  ```

- JSON input is empty or in an invalid format. The command example is as follows.

  Returns **NULL**.

  ```
  select get_json_object('','$.array[2]');
  ```

  Returns **NULL**.

  ```
  select get_json_object('"array":["a",1],"b":["c",3]','$.array[1][1]');
  ```

- JSON string involves escape characters. The command example is as follows.

  Returns **3**.

  ```
  select get_json_object('{"a":"\\"3\\"","b":"6"}', '$.a');
  ```

  Returns **3**.

  ```
  select get_json_object('{"a":"\'3\'","b":"6"}', '$.a');
  ```

- A JSON object can contain the same Key, and it can be parsed successfully.

  Returns **1**.

  ```
  select get_json_object('{"b":"1","b":"2"}', '$.b');
  ```

 Copyright © Huawei Technologies Co., Ltd.

- The output result is output in the original sorting order of the JSON string.
  Returns **{"b":"3","a":"4"}**.

```
select get_json_object('{"b":"3","a":"4"}', '$');
```

15.2.8 initcap

The **initcap** function is used to convert a text string to a form where the first letter of each word is capitalized and the remaining letters are in lowercase.

Syntax

```
initcap(string A)
```

Parameters

Parameter	Mandatory	Type	Description
A	Yes	STRING	The text string to be converted.

Return Values

Returns a STRING type string where the first letter of each word is capitalized and the rest are in lowercase.

Example Code

Returns **Fli Sql**.

```
SELECT initcap("fLI sql");
```

15.2.9 instr

The **instr** function is used to return the earliest occurrence index of **substr** in **str**.

When a parameter is **NULL**, **NULL** is returned. When **substr** does not exist in **str**, 0 is returned. Note that the index starts from 1.

Similar function: **instr1**. The **instr1** function calculates the position of substring **str2** in string **str1**. The **instr1** function supports specifying a starting search position and the occurrence count.

Syntax

```
instr(string , string )
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The target string to be searched. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before evaluation. Other types will return an error.
substr	Yes	STRING	The substring to be matched. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before evaluation. Other types will return an error.

Return Values

Returns a value of type BIGINT.

NOTE

- If **str2** is not found in **str1**, 0 is returned.
- If **str2** is an empty string, it always matches successfully. For example, **select instr('abc','');** returns 1.
- If **str1** or **str2** is NULL, NULL is returned.

Example Code

- Calculate the position of character **b** in the string **abc**. The command example is as follows.

  Returns **2**.

  ```
  select instr('abc', 'b');
  ```

- Any input parameter is NULL. The command example is as follows.

  Returns **NULL**.

  ```
  select instr('abc', null)
  ```

15.2.10 instr1

The **instr1** function is used to calculate the position of substring **str2** in string **str1**.

Similar function: **instr**. The **instr** function returns the earliest occurrence index of **substr** in **str**. However, **instr** does not support specifying a starting search position or the occurrence count.

Syntax

```
instr1(string <str1>, string <str2>[, bigint <start_position>[, bigint <nth_appearance>]])
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 1 Parameter descriptions

Parameter	Mandatory	Type	Description
str1	Yes	STRING	The target string to be searched. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before evaluation. Other types will return an error.
str2	Yes	STRING	The substring to be matched. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before evaluation. Other types will return an error.
start_position	No	BIGINT	Indicates the character position in **str1** to start the search from. The default starting position is the first character position, 1. Specifying a negative number is not supported.
nth_appearance	No	BIGINT	Indicates the position of the **nth_appearance** match of **str2** in **str1**. If **nth_appearance** is of another type or is less than or equal to 0, an error is returned.

Return Values

Returns a value of type BIGINT.

NOTE

- If **str2** is not found in **str1**, 0 is returned.
- If **str2** is an empty string, it always matches successfully.
- If **str1**, **str2**, **start_position**, or **nth_appearance** is NULL, NULL is returned.

Example Code

Returns **10**.

```
select instr1('Tech on the net', 'h', 5, 1);
```

Returns **2**.

```
select instr1('abc', 'b');
```

Returns **NULL**.

```
select instr1('abc', null);
```

15.2.11 keyvalue

The **keyvalue** function is used to split the string **str** by **split1**, then split each group into key-value pairs by **split2**, and return the value corresponding to the specified key.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
keyvalue(string <str>,[string <split1>,string <split2>,] string <key>)
```

Parameters

Table 1 Parameter descriptions

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string to be split.
split1, split2	No	STRING	The strings used as delimiters. The source string is split according to the two specified delimiters. If these two items are not specified in the expression, the default **split1** is **;** and **split2** is **:**. When a string split by **split1** contains multiple **split2** characters, the return result is undefined.
key	Yes	STRING	After splitting the string by **split1** and **split2**, returns the value corresponding to the **key** value.

Return Values

Returns STRING type.

NOTE

- If **split1** or **split2** is NULL, NULL is returned.
- If **str** or **key** is NULL, or if no matching key is found, NULL is returned.
- If there are multiple key-value matches, the value corresponding to the first matching key is returned.

Example Code

Returns **2**.

```
select keyvalue('a:1;b:2', 'b');
```

15.2.12 length

The **length** function is used to return the length of a string.

Similar function: **lengthb**. The **lengthb** function calculates the byte length of a string **str**.

Syntax

```
length(string )
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string whose length is to be calculated. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before evaluation. Other types will return an error.

Return Values

Returns a value of type BIGINT.

NOTE

- If **str** is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If **str** is NULL, NULL is returned.

Example Code

- Calculate the length of the string **abc**. The command example is as follows.

  Returns **3**.

  ```
  select length('abc');
  ```

- The input parameter is NULL. The command example is as follows.

  Returns **NULL**.

  ```
  select length(null);
  ```

15.2.13 lengthb

The **lengthb** function is used to calculate the byte length of a string **str**.

Similar function: length. The **length** function returns the length of a string.

Syntax

```
lengthb(string )
```

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The input string.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type BIGINT.

NOTE

- If **str** is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If **str** is NULL, NULL is returned.

Example Code

Returns **5**.

```
select lengthb('hello');
```

Returns **NULL**.

```
select lengthb(null);
```

15.2.14 levenshtein

The **levenshtein** function is used to return the Levenshtein distance between two strings. For example, **levenshtein('kitten','sitting') = 3**.

NOTE

The Levenshtein distance is a type of edit distance. It refers to the minimum number of single-character edits required to change one string into the other.

Syntax

```
levenshtein(string A, string B)
```

Parameters

Parameter	Mandatory	Type	Description
A, B	Yes	STRING	The strings required for calculating the Levenshtein distance.

Return Values

Returns a value of type INT.

Example Code

Returns **3**.

```
SELECT levenshtein('kitten','sitting');
```

 Copyright © Huawei Technologies Co., Ltd.

15.2.15 locate

The **locate** function is used to find the position of **substr** in **str**, counting from 1. The search can start from a specified position using **start_pos**.

Syntax

```
locate(string <substr>, string <str>[, bigint <start_pos>])
```

Parameters

Parameter	Mandatory	Type	Description
substr	Yes	STRING	The substring to be matched. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before evaluation.
str	Yes	STRING	The target string to be searched. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before evaluation.
start_pos	No	BIGINT	Specifies the starting position for the search, starting from 1. When not specified, the default is 1.

Return Values

Returns a value of type INT.

NOTE

- If **substr** cannot be matched in **str**, 0 is returned.
- If **str** or **substr** is NULL, NULL is returned.
- If **start_pos** is NULL, 0 is returned.

Example Code

Find the position of string **ab** in string **abhiab**. Return **1**.

```
select locate('ab', 'abhiab');
```

Start searching from the 2nd position. Return **5**.

```
select locate('ab', 'abhiab', 2);
```

If **start_pos** is NULL, 0 is returned.

```
select locate('ab', 'abhiab', null);
```

Find the position of string **hi** in string **hanmeimei and lilei**. Return **0**.

```
select locate('hi', 'hanmeimei and lilei');
```

 Copyright © Huawei Technologies Co., Ltd.

15.2.16 lower/lcase

The **lower** function is used to convert a text string to all lowercase letters.

Syntax

```
lower(string A) / lcase(string A)
```

Parameters

Parameter	Mandatory	Type	Description
A	Yes	STRING	The text string to be converted.

Return Values

Returns a value of type STRING.

NOTE

- If the input parameter is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If the input parameter is NULL, NULL is returned.

Example Code

Convert uppercase characters in the string to lowercase characters. The command example is as follows.

Returns **abc**.

```
select lower('ABC');
```

The input parameter is NULL. The command example is as follows.

Returns **NULL**.

```
select lower(null);
```

15.2.17 lpad

Returns a string of a specified length. When the length of the given string **str** is less than the specified length **len**, the string is padded on the left with the specified character **pad**. When the length of the given string **str** is greater than the specified length **len**, the string is truncated from the left to **len** characters.

Syntax

```
lpad(string <str>, int <len>[, string <pad>])
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string to be padded.
len	Yes	INT	The length of the result string.
pad	No	STRING	The string used for padding. When not specified, the default is a space.

Return Values

Returns a value of type STRING.

NOTE

- If **len** is less than the number of characters in **str**, the string is truncated from the left to **len** characters.
- If **len** is 0, an empty string is returned.
- If there are no input parameters or any input parameter is NULL, NULL is returned.

Example Code

Pad the string **abcdefgh** on the left with the string **ZZ** to a length of 10. Return **ZZabcdefgh**.

```
select lpad('abcdefgh', 10, 'ZZ');
```

Pad the string **abcdefgh** on the left with the string **ZZ** to a length of 5. Return **abcde**.

```
select lpad('abcdefgh', 5, 'ZZ');
```

length is 0. Return an empty string.

```
select lpad('abcdefgh', 0, 'ZZ');
```

Any input parameter is NULL. Return **NULL**.

```
select lpad(null, 0, 'ZZ');
```

15.2.18 ltrim

The **ltrim** function is used to remove characters from the left end of **str**:

- If **trimChars** is not specified, spaces are removed by default.
- If **trimChars** is specified, the characters in **trimChars** are treated as a set. The longest possible substring from the left end of **str** where all characters are in the **trimChars** set is removed.

Similar functions:

- **rtrim**: The **rtrim** function removes characters from the right end of **str**.

 Copyright © Huawei Technologies Co., Ltd.

- **trim**: The **trim** function removes characters from both the left and right ends of **str**.

Syntax

```
ltrim([<trimChars>,] string <str>)
```

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string from which to remove characters from the left end. If the input is of type BIGINT, DECIMAL, DOUBLE, or DATETIME, it is implicitly converted to STRING type before evaluation.
trimChars	No	STRING	The characters to be removed.

Return Values

Returns a value of type STRING.

NOTE

- If **str** is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If **str** or **trimChars** is NULL, NULL is returned.

Example Code

- Remove the left spaces from the string **abc**. The command example is as follows.

  Returns the string **abc**.

  ```
  select ltrim('     abc');
  ```

  Equivalent to the following statement.

  ```
  select trim(leading from '     abc');
  ```

  leading indicates removing spaces from the beginning of the string.
- The input parameter is NULL. The command example is as follows.

  Returns **NULL**.

  ```
  select ltrim(null);     select ltrim('xy', null);     select ltrim(null, 'xy');
  ```
- Remove the substring from the left end of **yxlucyxx** where all characters are in the set **xy**.

  Returns **lucyxx**. Whenever an **x** or **y** is encountered at the left end, it is removed.

  ```
  select ltrim( 'xy','yxlucyxx');
  ```

  Equivalent to the following statement.

 Copyright © Huawei Technologies Co., Ltd.

```
select trim(leading 'xy' from 'yxlucyxx');
```

15.2.19 parse_url

Returns the specified part of a given URL. Valid values for **partToExtract** are **HOST**, **PATH**, **QUERY**, **REF**, **PROTOCOL**, **AUTHORITY**, **FILE**, and **USERINFO**. When the second parameter is **QUERY**, the third parameter can be used to extract the value of a specific parameter.

Syntax

```
parse_url(string <urlString>, string <partToExtract [, string <keyToExtract>])
```

Parameters

Parameter	Mandatory	Type	Description
urlString	Yes	STRING	URL.
partToExtract	Yes	STRING	Valid values are **HOST**, **PATH**, **QUERY**, **REF**, **PROTOCOL**, **AUTHORITY**, **FILE**, and **USERINFO**. Case-insensitive.
keyToExtract	No	STRING	When **partToExtract** is **QUERY**, extracts the corresponding value based on the key.

Return Values

Returns a value of type STRING.

NOTE

- If **urlString**, **partToExtract**, or **keyToExtract** is NULL, NULL is returned.
- If **partToExtract** is not a valid value, NULL is returned.

Example Code

Returns **example.com**.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'HOST');
```

Returns **/over/there/index.dtb**.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'PATH');
```

Returns **animal**.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'QUERY', 'type');
```

Returns **nose**.

 Copyright © Huawei Technologies Co., Ltd.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'REF');
```

Returns **file**.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'PROTOCOL');
```

Returns **username@*example*.com:666**.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'AUTHORITY');
```

Returns **username**.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'USERINFO');
```

15.2.20 regexp_count1

Counts the number of substrings matching the specified **pattern** in **source**, starting from the **start_position**.

Syntax

```
regexp_count1(string <source>, string <pattern>[, bigint <start_position>])
```

Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	The string to be searched.
pattern	Yes	STRING	The regular expression. An error is returned if **pattern** is an empty string.
start_position	No	BIGINT	The starting position for the search. Must be greater than 0. When not specified, the default is **1**, meaning the match starts from the first character of **source**.

Return Values

Returns a value of type BIGINT.

NOTE

- If no match is found, 0 is returned.
- If **source** or **pattern** is NULL, NULL is returned.

Example Code

Returns **4**.

```
select regexp_count1('ab0a1a2b3c', '[0-9]');
```

Returns **3**.

 Copyright © Huawei Technologies Co., Ltd.

```
select regexp_count1('ab0a1a2b3c', '[0-9]', 4);
```

Returns **NULL**.

```
select regexp_count1('ab0a1a2b3c', null);
```

15.2.21 regexp_extract

Matches the string **source** against the grouping rules of **pattern** and returns the string content matched by the **groupid**-th group.

Syntax

```
regexp_extract(string <source>, string <pattern>[, bigint <groupid>])
```

Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	The string to be matched.
pattern	Yes	STRING	The regular expression. The pattern to be matched.
groupid	No	BIGINT	A BIGINT type constant, must be greater than or equal to 0. When not specified, the default is **1**, meaning the first group is returned. If **groupid** is **0**, the substring matching the entire **pattern** is returned.

Return Values

Returns a value of type STRING.

NOTE

- If **pattern** is an empty string or **pattern** contains no groups, an error is returned.
- If **groupid** is not of type BIGINT or is less than 0, an error is reported.
- If **source**, **pattern**, or **groupid** is NULL, NULL is returned.

Example Code

Split **basketball** by **bas(.*?)(ball)** and return **ket**.

```
select regexp_extract('basketball', 'bas(.*?)(ball)');
```

Returns **basketball**.

```
select regexp_extract('basketball', 'bas(.*?)(ball)', 0);
```

Returns **99**. In SQL, two **** characters are needed as escape characters.

```
select regexp_extract('8d99d8', '8d(\\d+)d8');
```

 Copyright © Huawei Technologies Co., Ltd.

15.2.22 regexp_instr1

Calculates the starting or ending position of the substring in **source** that matches **pattern** for the **occurrence**-th time, starting from **start_position**.

Syntax

```
regexp_instr1(string <source>, string <pattern>[, bigint <start_position>[, bigint <occurrence>[, bigint <return_option>]]])
```

Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	The source string.
pattern	Yes	STRING	The regular expression. The pattern to be matched. An error is returned if **pattern** is an empty string.
start_position	No	BIGINT	The starting position for the search. When not specified, the default value is **1**.
occurrence	No	BIGINT	Specifies the occurrence count. When not specified, the default value is **1**, meaning the first occurrence is searched.
return_option	No	BIGINT	Specifies the type of position to return. Valid values are **0** or **1**. When not specified, the default value is **0**. **0** indicates returning the starting position of the match, and **1** indicates returning the ending position of the match.

Return Values

Returns a value of type BIGINT. **return_option** specifies whether the starting or ending position of the matched substring in **source** is returned.

NOTE

- If **pattern** is an empty string, an error is reported.
- If **start_position** or **occurrence** is not of type BIGINT or is less than or equal to 0, an error is returned.
- If **source**, **pattern**, **start_position**, **occurrence**, or **return_option** is NULL, NULL is returned.

Example Code

Returns **6**.

```
select regexp_instr1('a1b2c3d4', '[0-9]', 3, 2);
```

Returns **NULL**.

 Copyright © Huawei Technologies Co., Ltd.

```
select regexp_instr1('a1b2c3d4', null, 3, 2);
```

15.2.23 regexp_replace

The **regexp_replace** function replaces substrings in **source** that match **pattern** with the specified string **replacement** and returns the result string. Spark 4.0 supports the **position** parameter, which allows specifying the match occurrence from which to start replacing.

Syntax

```
regexp_replace(str, pattern, replacement[, position])
```

Parameters

Table 1 Parameter descriptions

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string to be replaced.
pattern	Yes	STRING	The regular expression pattern. An error is returned if **pattern** is an empty string.
replacement	Yes	STRING	The string to replace the substrings matching **pattern**.
position	No	BIGINT	Specifies to replace all matches starting from the **position**-th match. If **0** or **1**, all matches are replaced. If another type or less than 0, an error is returned. The default value is **0** (replace all matches).

Return Values

Returns a value of type STRING.

NOTE

- If **pattern** is an empty string, an error is reported.
- When a non-existent group is referenced, no replacement is performed.
- If **replacement** is NULL and **pattern** has a match, NULL is returned.
- If **str**, **pattern**, or **position** is NULL, NULL is returned.

Example Code

Replace all matches. Return **num-num**.

```
SELECT regexp_replace('100-200', '(\\d+)', 'num');
```

Replace all matches. Return **2222**.

```
SELECT regexp_replace('abcd', '[a-z]', '2');
```

 Copyright © Huawei Technologies Co., Ltd.

Start replacing from the 2nd match. Return **a222**.

```
SELECT regexp_replace('abcd', '[a-z]', '2', 2);
```

Start replacing from the 3rd match. Return **ab22**.

```
SELECT regexp_replace('abcd', '[a-z]', '2', 3);
```

Start replacing from the 4th match. Return **abc2**.

```
SELECT regexp_replace('abcd', '[a-z]', '2', 4);
```

15.2.24 regexp_replace1

The **regexp_replace1** function is used to replace the substring in **source** that matches **pattern** for the **occurrence**-th time with the specified string **replace_string** and return the result string.

NOTE

The **regexp_replace1** function is only applicable to Spark 2.4.5 or earlier.

Similar function: **regexp_replace**. The **regexp_replace** function has slightly different functionalities depending on the Spark version. Refer to **regexp_replace** for detailed functional descriptions.

Syntax

```
regexp_replace1(string , string  , string [, bigint ])
```

Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	The string to be replaced.
pattern	Yes	STRING	A STRING type constant or regular expression. The pattern to be matched. For more regular expression writing specifications, see the regular expression specifications. An error is returned if **pattern** is an empty string.
replace_string	Yes	STRING	The string to replace the matched substring.
occurrence	No	BIGINT	Must be greater than or equal to 1. Indicates that the substring matching the **occurrence**-th time is replaced with **replace_string**. If 1, all matching substrings are replaced. If another type or less than 1, an error is returned. The default value is **1**.

Return Values

Returns a value of type STRING.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- When a non-existent group is referenced, no replacement is performed.
- If **replace_string** is NULL and **pattern** has a match, NULL is returned.
- If **replace_string** is NULL but **pattern** does not match, NULL is returned.
- If **source**, **pattern**, or **occurrence** is NULL, NULL is returned.

Example Code

Returns **2222**.

```
select regexp_replace1('abcd', '[a-z]', '2');
```

Returns **2bcd**.

```
select regexp_replace1('abcd', '[a-z]', '2', 1);
```

Returns **a2cd**.

```
select regexp_replace1('abcd', '[a-z]', '2', 2);
```

Returns **ab2d**.

```
select regexp_replace1('abcd', '[a-z]', '2', 3);
```

Returns **abc2**.

```
select regexp_replace1('abcd', '[a-z]', '2', 4);
```

15.2.25 regexp_substr1

Calculates the substring in **source** that matches the specified **pattern** for the **occurrence**-th time, starting from the **start_position**.

Syntax

```
regexp_substr1(string <source>, string <pattern>[, bigint <start_position>[, bigint <occurrence>]])
```

Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	The string to be searched.
pattern	Yes	STRING	The regular expression. The pattern to be matched. An error is returned if **pattern** is an empty string.
start_position	No	BIGINT	The starting position. Must be greater than 0. When not specified, the default is **1**, meaning the match starts from the first character of **source**.
occurrence	No	BIGINT	The occurrence count. Must be greater than 0. When not specified, the default is **1**, meaning the first matching substring is returned.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type STRING.

NOTE

- If **pattern** is an empty string, an error is reported.
- If no match is found, NULL is returned.
- If **start_position** or **occurrence** is not of type BIGINT or is less than or equal to 0, an error is returned.
- If **source**, **pattern**, **start_position**, or **occurrence** is NULL, NULL is returned.

Example Code

Returns **a**.

```
select regexp_substr1('a1b2c3', '[a-z]');
```

Returns **b**.

```
select regexp_substr1('a1b2c3', '[a-z]', 2, 1);
```

Returns **c**.

```
select regexp_substr1('a1b2c3', '[a-z]', 2, 2);
```

Returns **NULL**.

```
select regexp_substr1('a1b2c3', null);
```

15.2.26 repeat

The **repeat** function is used to return a string consisting of **str** repeated **n** times.

Syntax

```
repeat(string , bigint )
```

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string to be repeated. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before evaluation.
n	Yes	BIGINT	The number of repetitions.

Return Values

Returns STRING type.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **str** is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If **n** is empty, an error is reported.
- If **str** or **n** is NULL, NULL is returned.

Example Code

Repeat the string **123** 2 times. Return **123123**.

```
SELECT repeat('123', 2);
```

15.2.27 replace

Replaces all occurrences of the **old** string in the **str** string with the **new** string and returns the replaced string. If no overlapping string is found, the original **str** is returned.

Syntax

```
replace(string <str>, string <old>, string <new>)
```

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string to be replaced.
old	Yes	STRING	The string to be compared.
new	Yes	STRING	The replacement string.

Return Values

Returns a value of type STRING.

NOTE

If any input parameter is NULL, NULL is returned.

Example Code

Returns **AA123AA**.

```
SELECT replace('abc123abc', 'abc', 'AA');
```

Returns **NULL**.

```
SELECT replace('abc123abc', null, 'AA');
```

15.2.28 reverse

The **reverse** function is used to return the reversed string.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
reverse(string )
```

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string to be reversed. If the input is of type BIGINT, DOUBLE, DECIMAL, or DATETIME, it is implicitly converted to STRING type before evaluation.

Return Values

Returns STRING type.

NOTE

- If **str** is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If **str** is NULL, NULL is returned.

Example Code

Returns **LQS krapS**.

```
SELECT reverse('Spark SQL');
```

Returns **[3,4,1,2]**.

```
SELECT reverse(array(2, 1, 4, 3));
```

15.2.29 rpad

Right-pads the string **str1** to a length of **length** using the string **str2**.

Syntax

```
rpad(string <str1>, int <length>, string <str2>)
```

Parameters

Parameter	Mandatory	Type	Description
str1	Yes	STRING	The string to be right-padded.
length	Yes	INT	The target length after right-padding.
str2	Yes	STRING	The string used for padding.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type STRING.

NOTE

- If **length** is less than the length of **str1**, the string is truncated from the left to **length** characters.
- If **length** is 0, an empty string is returned.
- If any input parameter is NULL, NULL is returned.

Example Code

Returns **hi???**.

```
SELECT rpad('hi', 5, '??');
```

Returns **h**.

```
SELECT rpad('hi', 1, '??');
```

15.2.30 rtrim

The **rtrim** function is used to remove characters from the right end of **str**:

- If **trimChars** is not specified, space characters are removed by default.
- If **trimChars** is specified, the characters in **trimChars** are treated as a set. The longest possible substring from the right end of **str** where all characters are in the **trimChars** set is removed.

Similar functions:

- **ltrim**: The **ltrim** function removes characters from the left end of **str**.
- **trim**: The **trim** function removes characters from both the left and right ends of **str**.

Syntax

```
rtrim([<trimChars>, ]string <str>)
```

or

```
rtrim(trailing [] from )
```

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string from which to remove characters from the right end. If the input is of type BIGINT, DECIMAL, DOUBLE, or DATETIME, it is implicitly converted to STRING type before evaluation.
trimChars	No	STRING	The characters to be removed.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type STRING.

NOTE

- If **str** is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If **str** or **trimChars** is NULL, NULL is returned.

Example Code

- Remove the right spaces from the string **yxabcxx** . The command example is as follows.

  Returns the string **yxabcxx**.

  ```
  select rtrim('yxabcxx     ');
  ```

  Equivalent to the following statement.

  ```
  select trim(trailing from ' yxabcxx ');
  ```

- Remove the substring from the right end of **yxabcxx** where all characters are in the set **xy**.

  Returns **yxabc**. Whenever an **x** or **y** is encountered at the right end, it is removed.

  ```
  select rtrim('xy', 'yxabcxx');
  ```

  Equivalent to the following statement.

  ```
  select trim(trailing 'xy' from 'yxabcxx');
  ```

- The input parameter is NULL. The command example is as follows.

  Returns **NULL**.

  ```
  select rtrim(null);      select rtrim('yxabcxx', 'null');
  ```

15.2.31 space

The **space** function returns a string consisting of a specified number of spaces.

Syntax

```
space(bigint )
```

Parameters

Parameter	Mandatory	Type	Description
n	Yes	BIGINT	Specifies the number of spaces.

Return Values

Returns STRING type.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **n** is empty, an error is reported.
- If **n** is NULL, NULL is returned.

Example Code

Returns **6**.

```
select length(space(6));
```

15.2.32 split_part1

The **split_part1** function is used to split the string **str** by the delimiter **separator** and return the substring from the **start** part to the end part (inclusive).

Syntax

```
split_part1(string <str>, string <separator>, bigint <start>[, bigint <end>])
```

Parameters

Table 1 Parameter descriptions

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string to be split.
separator	Yes	STRING	A STRING type constant. The delimiter used for splitting. Can be a single character or a string.
start	Yes	BIGINT	A BIGINT type constant, must be greater than 0. Indicates the starting part number (starting from 1).
end	No	BIGINT	A BIGINT type constant, greater than or equal to **start**. Indicates the ending part number. Can be omitted. When omitted, it is equal to the value of **start**, returning the part specified by **start**.

Return Values

Returns a value of type STRING.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If the value of **start** is greater than the actual number of parts after splitting, for example, the string is split into 4 parts and **start** is greater than 4, an empty string is returned.
- If **separator** does not exist in **str** and **start** is specified as 1, the entire **str** is returned. If **str** is an empty string, an empty string is returned.
- If **separator** is an empty string, the original string **str** is returned.
- If **end** is greater than the number of parts, the substring starting from **start** is returned.
- If **str** is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If any parameter value is NULL (except **separator**), NULL is returned.

Example Code

Returns **aa**.

```
select split_part1('aa,bb,cc,dd', ',', 1);
```

Returns **aa,bb**.

```
select split_part1('aa,bb,cc,dd', ',', 1, 2);
```

Returns an empty string.

```
select split_part1('aa,bb,cc,dd', ',', 10);
```

Returns **aa,bb,cc,dd**.

```
select split_part1('aa,bb,cc,dd', ':', 1);
```

Returns an empty string.

```
select split_part1('aa,bb,cc,dd', ':', 2);
```

Returns **aa,bb,cc,dd**.

```
select split_part1('aa,bb,cc,dd', '', 1);
```

Returns **bb,cc,dd**.

```
select split_part1('aa,bb,cc,dd', ',', 2, 6);
```

Returns **NULL**.

```
select split_part1('aa,bb,cc,dd', ',', null);
```

15.2.33 substr/substring

Returns a substring of **str** starting from **start_position** with a length of **length**. **substr** and **substring** have the same functionality.

Syntax

```
substr(string <str>, bigint <start_position>[, bigint <length>])
```

or

```
substring(string <str>, bigint <start_position>[, bigint <length>])
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string from which to extract the substring.
start_position	Yes	BIGINT	The starting position, starting from 1. A positive value starts from the left, and a negative value starts from the right.
length	No	BIGINT	The length of the substring. Must be greater than 0. When omitted, the substring to the end of **str** is returned.

Return Values

Returns a value of type STRING.

NOTE

- When **length** is omitted, the substring to the end of **str** is returned.
- If **str**, **start_position**, or **length** is NULL, NULL is returned.

Example Code

Returns **k SQL**.

```
SELECT substr('Spark SQL', 5);
```

Returns **SQL**.

```
SELECT substr('Spark SQL', -3);
```

Returns **k**.

```
SELECT substr('Spark SQL', 5, 1);
```

15.2.34 substring_index

The **substring_index** function is used to truncate the string **str** before the **count**-th occurrence of the **separator**. If **count** is positive, truncation starts from the left. If **count** is negative, truncation starts from the right.

Syntax

```
substring_index(string <str>, string <separator>, int <count>)
```

Parameters

Table 1 Parameter descriptions

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string to be truncated.
separator	Yes	STRING	A STRING type delimiter.
count	Yes	INT	Specifies the delimiter position.

Return Values

Returns STRING type.

NOTE

If any input parameter is NULL, NULL is returned.

Example Code

Returns **hello.world**.

```
SELECT substring_index('hello.world.people', '.', 2);
```

Returns **world.people**.

```
select substring_index('hello.world.people', '.', -2);
```

15.2.35 translate

Replaces characters in the **input** string that appear in **from** with the corresponding characters at the same position in **to**.

For example: Replace **bcd** in **abcde** with **BCD**:

```
translate("abcde", "bcd", "BCD")
```

Syntax

```
translate(string <input>, string <from>, string <to>)
```

Parameters

Parameter	Mandatory	Type	Description
input	Yes	STRING	The string to be replaced.
from	Yes	STRING	The set of characters to be replaced.
to	Yes	STRING	The set of replacement characters.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type STRING.

NOTE

If any input parameter is NULL, NULL is returned.

Example Code

Returns **A1B2C3**.

```
SELECT translate('AaBbCc', 'abc', '123');
```

15.2.36 trim

The **trim** function is used to remove characters from both the left and right ends of **str**:

- If **trimChars** is not specified, space characters are removed by default.
- If **trimChars** is specified, the characters in **trimChars** are treated as a set. The longest possible substring from both ends of **str** where all characters are in the **trimChars** set is removed.

Similar functions:

- **ltrim**: The **ltrim** function removes characters from the left end of **str**.
- **rtrim**: The **rtrim** function removes characters from the right end of **str**.

Syntax

```
trim([,]string )
```

or

```
trim([BOTH] [] from )
```

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string from which to remove characters from both ends. If the input is of type BIGINT, DECIMAL, DOUBLE, or DATETIME, it is implicitly converted to STRING type before evaluation.
trimChars	No	STRING	The characters to be removed.

Return Values

Returns a value of type STRING.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **str** is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If **str** or **trimChars** is NULL, NULL is returned.

Example Code

- Remove the left and right spaces from the string **yxabcxx** . The command example is as follows.

  Returns the string **yxabcxx**.

  ```
  select trim(' yxabcxx ');
  ```

  Equivalent to the following statement.

  ```
  select trim(both from ' yxabcxx ');
  ```

- Remove the substring from both ends of **yxabcxx** where all characters are in the set **xy**.

  Returns **abc**. Whenever an **x** or **y** is encountered at either end, it is removed.

  ```
  select trim('xy', 'yxabcxx');
  ```

  Equivalent to the following statement.

  ```
  select trim(both 'xy' from 'yxabcxx');
  ```

- The input parameter is NULL. The command example is as follows.

  Returns **NULL**.

  ```
  select trim(null);      select trim(null, 'yxabcxx');
  ```

15.2.37 upper/ucase

The **upper** function is used to convert a text string to all uppercase letters.

Syntax

```
upper(string A)
```

or

```
ucase(string A)
```

Parameters

Parameter	Mandatory	Type	Description
A	Yes	STRING	The text string to be converted.

Return Values

Returns STRING type.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If the input parameter is not of type STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME, an error is returned.
- If the input parameter is NULL, NULL is returned.

Example Code

Convert lowercase characters in the string to uppercase characters. The command example is as follows.

Returns **ABC**.

```
select upper('abc');
```

The input parameter is NULL. The command example is as follows.

Returns **NULL**.

```
select upper(null);
```

15.2.38 url_decode1

The **url_decode1** function is used to decode a string from **application/x-www-form-urlencoded** MIME format to regular characters.

Syntax

```
url_decode1(string [, string ])
```

Parameters

Parameter	Mandatory	Type	Description
input	Yes	STRING	The input string.
encoding	No	STRING	Specifies the encoding format. Supports standard encoding formats such as GBK or UTF-8. If not specified, the default is **UTF-8**.

Return Values

Returns a value of type STRING.

NOTE

A STRING type UTF-8 encoded string.

Example Code

Returns **Example for URL_DECODE:// dsf (fasfs)**.

```
select url_decode1('Example+for+url_decode+%3A%2F%2F+dsf%28fasfs%29', 'GBK');
```

 Copyright © Huawei Technologies Co., Ltd.

15.2.39 url_encode1

The **url_encode1** function encodes a string to **application/x-www-form-urlencoded** MIME format.

Syntax

```
url_encode1(string [, string ])
```

Parameters

Parameter	Mandatory	Type	Description
input	Yes	STRING	The input string.
encoding	No	STRING	Specifies the encoding format. Supports standard encoding formats such as GBK or UTF-8. If not specified, the default is **UTF-8**.

Return Values

Returns a value of type STRING.

NOTE

If **input** or **encoding** is NULL, NULL is returned.

Example Code

Returns **Example+for+url_encode+%3A%2F%2F+dsf%28fasfs%29**.

```
select url_encode1('Example for url_encode:// dsf(fasfs)', 'GBK');
```

15.2.40 printf

The **printf** function is used to format and print the input according to a specific format.

Syntax

```
printf(String format, Obj... args)
```

Parameters

Parameter	Mandatory	Type	Description
format	Yes	STRING	Defines the output format.
Obj	No	STRING	Other input parameters.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type STRING.

Fills the parameters from **Obj** into **format** and prints the output.

Example Code

Returns the string **name: user1, age: 20, gender: female, place of origin: city 1**.

```
SELECT printf('Name: %s, Age: %d, Gender: %s, Place of origin: %s', "user1", 20, "Female", "City 1");
```

15.3 Mathematical Functions

15.3.1 abs

The **abs** function is used to calculate the absolute value of the input parameter.

Syntax

abs(DOUBLE a)

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE, INT, or DECIMAL.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **NULL**.

 Copyright © Huawei Technologies Co., Ltd.

```
select abs(null);
```

Returns **1**.

```
select abs(-1);
```

Returns **3.1415926**.

```
select abs(-3.1415926);
```

15.3.2 acos

The **acos** function is used to return the arccosine of the given numeric value **a**.

Syntax

```
acos(DOUBLE a)
```

Parameters

Table 15-33 Parameter descriptions

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value range of parameter **a** is [-1,1]. The format of **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE, in the range of 0 to π.

NOTE

- If the value of **a** is not within the range [-1, 1], NaN is returned.
- If **a** is NULL, NULL is returned.

Example Code

Returns **3.141592653589793**.

```
select acos(-1);
```

Returns **0**.

```
select acos(1);
```

Returns **NULL**.

```
select acos(null);
```

Returns **NaN**.

 Copyright © Huawei Technologies Co., Ltd.

```
select acos(10);
```

15.3.3 asin

The **asin** function is used to return the arcsine of the given numeric value **a**.

Syntax

```
asin(DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value range of parameter **a** is [-1,1]. The format of **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE, in the range of -π/2 to π/2.

NOTE

- If the value of **a** is not within the range [-1, 1], NaN is returned.
- If **a** is NULL, NULL is returned.

Example Code

Returns **1.5707963267948966**.

```
select asin(1);
```

Returns **0.6435011087932844**.

```
select asin(0.6);
```

Returns **NULL**.

```
select asin(null);
```

Returns **NaN**.

```
select asin(10);
```

15.3.4 atan

The **atan** function is used to return the arctangent of the given numeric value **a**.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
atan(DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE, in the range of -π/2 to π/2.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **0.7853981633974483**.

```
select atan(1);
```

Returns **0.5404195002705842**.

```
select atan(0.6);
```

Returns **NULL**.

```
select atan(null);
```

15.3.5 bin

The **bin** function is used to return the binary representation of **a**.

Syntax

```
bin(BIGINT a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** is an integer format. If parameter **a** is not of type BIGINT, it is implicitly converted to BIGINT type before evaluation.

Return Values

Returns STRING type.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **1**.

```
select bin(1);
```

Returns **NULL**.

```
select bin(null);
```

Returns **1000**.

```
select bin(8);
```

Returns **1000**.

```
select bin(8.123456);
```

15.3.6 bround

The **bround** function is used to return a value that is the result of rounding the input to the specified number of decimal places **d** using banker's rounding.

Syntax

```
bround(DOUBLE a, INT d)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. Represents the value to be rounded. The difference between this command and traditional rounding is in the handling of the digit 5: the preceding digit determines the rounding. If the preceding digit is odd, it rounds up. If the preceding digit is even, it rounds down. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.
d	No	DOUBLE, BIGINT, DECIMAL, or STRING	Represents the number of decimal places to round to. If parameter **d** is not of type INT, it is implicitly converted to INT type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** or **d** is NULL, NULL is returned.

Example Code

Returns **123.4**.

```
select bround(123.45,1);
```

Returns **123.6**.

```
select bround(123.55,1);
```

Returns **NULL**.

```
select bround(null);
```

Returns **123.457**.

```
select bround(123.456789,3.123456);
```

15.3.7 cbrt

The **cbrt** function is used to return the cube root of **a**.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
cbrt(DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **3**.

```
select cbrt(27);
```

Returns **3.3019272488946267**.

```
select cbrt(36);
```

Returns **NULL**.

```
select cbrt(null);
```

15.3.8 ceil

The **ceil** function is used to return the smallest integer greater than or equal to **a**.

Syntax

```
ceil(DOUBLE a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DECIMAL.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **2**.

```
select ceil(1.3);
```

Returns **-1**.

```
select ceil(-1.3);
```

Returns **NULL**.

```
select ceil(null);
```

15.3.9 conv

The **conv** function is used to convert a number **num** from base **from_base** to base **to_base**.

Syntax

```
conv(BIGINT num, INT from_base, INT to_base)
```

Parameters

Parameter	Mandatory	Type	Description
num	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The number to be converted. The format of parameter **num** includes floating-point format, integer format, and string format.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Description
from_base	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The source base **from_base**. The format of parameter **from_base** includes floating-point format, integer format, and string format.
to_base	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The target base **to_base**. The format of parameter **to_base** includes floating-point format, integer format, and string format.

Return Values

Returns STRING type.

NOTE

- If **num**, **from_base**, or **to_base** is NULL, NULL is returned.
- The conversion process works with 64-bit precision. NULL is returned on overflow.
- If **num** is a decimal number, it is converted to an integer value before the base conversion. The fractional part is discarded.

Example Code

Returns **8**.

```
select conv('1000', 2, 10);
```

Returns **B**.

```
select conv('1011', 2, 16);
```

Returns **703710**.

```
select conv('ABCDE', 16, 10);
```

Returns **27**.

```
select conv(1000.123456, 3.123456, 10.123456);
```

Returns **18446744073709551589**.

```
select conv(-1000.123456, 3.123456, 10.123456);
```

Returns **NULL**.

```
select conv('1100', null, 10);
```

15.3.10 cos

The **cos** function is used to calculate the cosine of **a**, where the input is in radians.

Syntax

```
cos(DOUBLE a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **-0.9999999999999986**.

```
select cos(3.1415926);
```

Returns **NULL**.

```
select cos(null);
```

15.3.11 cot1

The **cot1** function is used to calculate the cotangent of **a**, where the input is in radians.

Syntax

```
cot1(DOUBLE a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE or DECIMAL.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **1.0000000000000002**.

```
select cot1(pi()/4);
```

Returns **NULL**.

```
select cot1(null);
```

15.3.12 degrees

The **degrees** function is used to convert radians to degrees.

Syntax

```
degrees(DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **90.0**.

```
select degrees(1.5707963267948966);
```

Returns **0**.

```
select degrees(0);
```

Returns **NULL**.

```
select degrees(null);
```

15.3.13 e

The **e** function is used to return the value of the natural constant e.

Syntax

```
e()
```

Return Values

Returns a value of type DOUBLE.

Example Code

Returns **2.718281828459045**.

```
select e();
```

15.3.14 exp

The **exp** function is used to calculate the value of e raised to the power of **a**.

Syntax

```
exp(DOUBLE a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **7.38905609893065**.

```
select exp(2);
```

Returns **20.085536923187668**.

```
select exp(3);
```

Returns **NULL**.

```
select exp(null);
```

15.3.15 factorial

The **factorial** function is used to return the factorial of **a**.

Syntax

```
factorial(INT a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	BIGINT, INT, SMALLINT, or TINYINT	The format of parameter **a** is an integer format. If parameter **a** is not of type INT, it is implicitly converted to INT type before evaluation. Strings are converted to their corresponding ASCII codes.

Return Values

Returns a value of type BIGINT.

NOTE

- If **a** is 0, 1 is returned.
- If **a** is NULL or outside the range [0,20], NULL is returned.

Example Code

Returns **720**.

```
select factorial(6);
```

Returns **1**.

```
select factorial(1);
```

Returns **120**.

```
select factorial(5.123456);
```

Returns **NULL**.

```
select factorial(null);
```

Returns **NULL**.

```
select factorial(21);
```

15.3.16 floor

The **floor** function is used to return the largest integer less than or equal to **a**.

Syntax

```
floor(DOUBLE a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type BIGINT.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **1**.

```
select floor(1.2);
```

Returns **-2**.

```
select floor(-1.2);
```

Returns **NULL**.

```
select floor(null);
```

15.3.17 greatest

The **greatest** function is used to return the maximum value from a list.

Syntax

```
greatest(T v1, T v2, ...)
```

Parameters

Parameter	Mandatory	Type	Description
v1	Yes	DOUBLE, BIGINT, or DECIMAL	The format of parameter **v1** includes floating-point format, integer format, and DECIMAL format.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Description
v2	Yes	DOUBLE, BIGINT, or DECIMAL	The format of parameter **v2** includes floating-point format, integer format, and DECIMAL format.

Return Values

Returns a value of type DOUBLE.

NOTE

If any parameter is NULL, the NULL is ignored and the maximum value is returned.

Example Code

Returns **4.0**.

```
select greatest(1,2.0,3,4.0);
```

Returns **4**.

```
select greatest(null,4);
```

15.3.18 hex

The **hex** function is used to convert an integer or character to its hexadecimal representation.

Syntax

```
hex(BIGINT a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type BIGINT, it is implicitly converted to BIGINT type before evaluation. Strings are converted to their corresponding ASCII codes.

Return Values

Returns STRING type.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **a** is 0, 0 is returned.
- If **a** is NULL, NULL is returned.

Example Code

Returns **0**.

```
select hex(0);
```

Returns **61**.

```
select hex('a');
```

Returns **10**.

```
select hex(16);
```

Returns **31**.

```
select hex('1');
```

Returns **3136**.

```
select hex('16');
```

Returns **NULL**.

```
select hex(null);
```

15.3.19 least

The **least** function is used to return the minimum value from a list.

Syntax

```
least(T v1, T v2, ...)
```

Parameters

Parameter	Mandatory	Type	Description
v1	Yes	DOUBLE, BIGINT, or DECIMAL	The format of parameter **v1** includes floating-point format, integer format, and DECIMAL format.
v2	Yes	DOUBLE, BIGINT, or DECIMAL	The format of parameter **v2** includes floating-point format, integer format, and DECIMAL format.

Return Values

Returns a value of type DOUBLE.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If **v1** or **v2** is of type STRING, an error is returned.
- If all parameters are NULL, NULL is returned.

Example Code

Returns **1.0**.

```
select least(1,2.0,3,4.0);
```

Returns **NULL**.

```
select least(null,null);
```

15.3.20 ln

The **ln** function is used to return the natural logarithm of the given numeric value.

Syntax

```
ln(DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

- If **a** is negative or 0, NULL is returned.
- If **a** is NULL, NULL is returned.

Example Code

Returns **1.144729868791239**.

```
select ln(3.1415926);
```

Returns **1**.

```
select ln(2.718281828459045);
```

Returns **NULL**.

 Copyright © Huawei Technologies Co., Ltd.

```
select ln(null);
```

15.3.21 log

The **log** function is used to return the logarithm of a number **a** with a given base **base**.

Syntax

```
log(DOUBLE base, DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
base	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter `base` includes floating-point format, integer format, and string format. If parameter **base** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

- If **base** or **a** is NULL, NULL is returned.
- If **base** or **a** is negative or 0, NULL is returned.
- If **base** is 1 (which would cause a division by zero), NULL is returned.

Example Code

Returns **2**.

```
select log(2, 4);
```

Returns **NULL**.

```
select log(2, null);
```

Returns **NULL**.

```
select log(null, 4);
```

 Copyright © Huawei Technologies Co., Ltd.

15.3.22 log10

The **log10** function is used to return the base-10 logarithm (common logarithm) of the given numeric value.

Syntax

```
log10(DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is **0**, negative, or NULL, NULL is returned.

Example Code

Returns **NULL**.

```
select log10(null);
```

Returns **NULL**.

```
select log10(0);
```

Returns **0.9542425094393249**.

```
select log10(9);
```

Returns **1**.

```
select log10(10);
```

15.3.23 log2

The **log2** function is used to return the base-2 logarithm of the given numeric value.

Syntax

```
log2(DOUBLE a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is **0**, negative, or NULL, NULL is returned.

Example Code

Returns **NULL**.

```
select log2(null);
```

Returns **NULL**.

```
select log2(0);
```

Returns **3.1699250014423126**.

```
select log2(9);
```

Returns **4**.

```
select log2(16);
```

15.3.24 negative

The **negative** function is used to return the negation of **a**.

Syntax

```
negative(INT a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format.

Return Values

Returns a value of type DECIMAL or INT.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **-1**.

```
SELECT negative(1);
```

Returns **3**.

```
SELECT negative(-3);
```

15.3.25 pi

The **pi** function is used to return the value of π (pi).

Syntax

```
pi()
```

Return Values

Returns a value of type DOUBLE.

Example Code

Returns **3.141592653589793**.

```
select pi();
```

15.3.26 pmod

The **pmod** function is used to return the positive modulus of **a** divided by **b**.

Syntax

```
pmod(INT a, INT b)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format.
b	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **b** includes floating-point format, integer format, and string format.

Return Values

Returns a value of type DECIMAL or INT.

NOTE

- If **a** or **b** is NULL, NULL is returned.
- If **b** is 0, NULL is returned.

Example Code

Returns **2**.

```
select pmod(2,5);
```

Returns **1**.

```
select pmod(5,2);
```

Returns **0.877**.

```
select pmod(5.123,2.123);
```

15.3.27 positive

The **positive** function is used to return the value of **a**.

Syntax

```
positive(INT a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type DECIMAL, DOUBLE, or INT.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **3**.

```
SELECT positive(3);
```

Returns **-3**.

```
SELECT positive(-3);
```

Returns **123**.

```
SELECT positive('123');
```

15.3.28 pow

The **pow** function is used to calculate **a** raised to the power of **p**.

Syntax

```
pow(DOUBLE a, DOUBLE p),
power(DOUBLE a, DOUBLE p)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.
p	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **p** includes floating-point format, integer format, and string format. If parameter **p** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** or **p** is NULL, NULL is returned.

Example Code

Returns **16**.

```
select pow(2, 4);
```

Returns **NULL**.

```
select pow(2, null);
```

Returns **17.429460393524256**.

```
select pow(2, 4.123456);
```

15.3.29 radians

The **radians** function is used to convert degrees to radians.

Syntax

```
radians(DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **1.0471975511965976**.

```
select radians(60);
```

 Copyright © Huawei Technologies Co., Ltd.

Returns **0**.

```
select radians(0);
```

Returns **NULL**.

```
select radians(null);
```

15.3.30 rand

The **rand** function is used to return a uniformly distributed random number greater than or equal to 0 and less than 1.

Syntax

```
rand(INT seed)
```

Parameters

Parameter	Mandatory	Type	Description
seed	No	INT	The format of parameter **seed** is an integer format. If a seed is specified, a stable sequence of random numbers will be generated under the same runtime environment.

Return Values

Returns a value of type DOUBLE.

Example Code

Returns **0.3668915240363728**.

```
select rand();
```

Returns **0.25738143505962285**.

```
select rand(3);
```

15.3.31 round

The **round** function is used to calculate the value of **a** rounded to **d** decimal places.

Syntax

```
round(DOUBLE a, INT d)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	Represents the value to be rounded. The format of parameter **a** includes floating-point format, integer format, and string format.
d	No	INT	Default value: **0**. Represents the number of decimal places to round to. If parameter **d** is not of type INT, it is implicitly converted to INT type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

- If **d** is negative, rounding is performed on the |d|-th digit to the left of the decimal point.
- If **a** or **d** is NULL, NULL is returned.

Example Code

Returns **123.0**.

```
select round(123.321);
```

Returns **123.4**.

```
select round(123.396, 1);
```

Returns **NULL**.

```
select round (NULL);
```

15.3.32 shiftleft

The **shiftleft** function is used to perform a signed left shift, shifting the binary representation of **a** left by **b** bits.

Syntax

```
shiftleft(BIGINT a, BIGINT b)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type BIGINT, it is implicitly converted to BIGINT type before evaluation.
b	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **b** includes floating-point format, integer format, and string format. If parameter **b** is not of type BIGINT, it is implicitly converted to BIGINT type before evaluation.

Return Values

Returns a value of type INT.

NOTE

If **a** or **b** is NULL, NULL is returned.

Example Code

Returns **8**.

```
select shiftleft(1,3);
```

Returns **48**.

```
select shiftleft(6,3);
```

Returns **48**.

```
select shiftleft(6.123456,3.123456);
```

Returns **NULL**.

```
select shiftleft(null,3);
```

15.3.33 shiftright

The **shiftright** function is used to perform a signed right shift, shifting the binary representation of **a** right by **b** bits.

Syntax

```
shiftright(BIGINT a, BIGINT b)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type BIGINT, it is implicitly converted to BIGINT type before evaluation.
b	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **b** includes floating-point format, integer format, and string format. If parameter **b** is not of type BIGINT, it is implicitly converted to BIGINT type before evaluation.

Return Values

Returns a value of type INT.

NOTE

If **a** or **b** is NULL, NULL is returned.

Example Code

Returns **2**.

```
select shiftright(16,3);
```

Returns **4**.

```
select shiftright(36,3);
```

Returns **4**.

```
select shiftright(36.123456,3.123456);
```

Returns **NULL**.

```
select shiftright(null,3);
```

15.3.34 shiftrightunsigned

The **shiftrightunsigned** function is used to perform an unsigned right shift, shifting the binary representation of **a** right by **b** bits.

Syntax

```
shiftrightunsigned(BIGINT a, BIGINT b)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type BIGINT, it is implicitly converted to BIGINT type before evaluation.
b	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **b** includes floating-point format, integer format, and string format. If parameter **b** is not of type BIGINT, it is implicitly converted to BIGINT type before evaluation.

Return Values

Returns a value of type INT.

NOTE

If **a** or **b** is NULL, NULL is returned.

Example Code

Returns **2**.

```
select shiftrightunsigned(16,3);
```

Returns **536870910**.

```
select shiftrightunsigned(-16,3);
```

Returns **2**.

```
select shiftrightunsigned(16.123456,3.123456);
```

Returns **NULL**.

```
select shiftrightunsigned(null,3);
```

15.3.35 sign

The **sign** function is used to return the sign of **a**.

Syntax

```
sign(DOUBLE a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format.

Return Values

Returns a value of type DOUBLE.

NOTE

- If **a** is positive, 1 is returned.
- If **a** is negative, -1 is returned.
- If **a** is 0, 0 is returned.
- If **a** is NULL, NULL is returned.

Example Code

Returns **-1**.

```
select sign(-3);
```

Returns **1**.

```
select sign(3);
```

Returns **0**.

```
select sign(0);
```

Returns **1**.

```
select sign(3.1415926);
```

Returns **NULL**.

```
select sign(null);
```

15.3.36 sin

The **sin** function is used to calculate the sine of **a**, where the input is in radians.

Syntax

```
sin(DOUBLE a)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **1**.

```
select sin(pi()/2);
```

Returns **NULL**.

```
select sin(null);
```

15.3.37 soundex

The **soundex** function is used to return a Soundex string from **str**. For example, **soundex('Miller') = M460**.

Syntax

```
soundex(string str)
```

Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	The string to be converted.

Return Values

Returns a value of type STRING.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

If **str** is NULL, NULL is returned.

Example Code

Returns **M460**.

```
SELECT soundex('Miller');
```

15.3.38 sqrt

The **sqrt** function is used to return the square root of a numeric value.

Syntax

```
sqrt(DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **2.8284271247461903**.

```
select sqrt(8);
```

Returns **4**.

```
select sqrt(16);
```

Returns **NULL**.

```
select sqrt(null);
```

 Copyright © Huawei Technologies Co., Ltd.

15.3.39 tan

The **tan** function is used to calculate the tangent of **a**, where the input is in radians.

Syntax

```
tan(DOUBLE a)
```

Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The format of parameter **a** includes floating-point format, integer format, and string format. If parameter **a** is not of type DOUBLE, it is implicitly converted to DOUBLE type before evaluation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **a** is NULL, NULL is returned.

Example Code

Returns **0.9999999999999999**.

```
select tan(pi()/4);
```

Returns **NULL**.

```
select tan(null);
```

15.4 Aggregate Functions

15.4.1 avg

Function

The **avg** function is used to calculate the average value.

Syntax

```
avg(col), avg(DISTINCT col)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Description
col	Yes	Column values support all data types and can be converted to DOUBLE type for computation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **col** is NULL, the column is excluded from the calculation.

Examples

- Calculate the average number of items across all warehouses. Example command:
  ```
  select avg(items) from warehouse;
  ```
  Result:
  ```
  _c0
  100.0
  ```
- Used with GROUP BY to calculate the average inventory of all items in each warehouse. Example command:
  ```
  select warehouseId, avg(items) from warehouse group by warehouseId;
  ```
  Result:
  ```
  warehouseId _c1
  city1    155
  city2    101
  city3    194
  ```

15.4.2 corr

Function

The **corr** function is used to return the correlation coefficient of two numeric columns.

Syntax

```
corr(col1, col2)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
col1	Yes	DOUBLE, BIGINT, INT, SMALLINT, TINYINT, FLOAT, or DECIMAL	A column with numeric data type. Other types return NULL.
col2	Yes	DOUBLE, BIGINT, INT, SMALLINT, TINYINT, FLOAT, or DECIMAL	A column with numeric data type. Other types return NULL.

Return Values

Returns a value of type DOUBLE.

Examples

- Calculate the correlation coefficient between all items inventory and price. Example command:

```
select corr(items,price) from warehouse;
```

  Result:

```
_c0
1.242355
```

- Used with GROUP BY to group all items by warehouse (warehouseId) and calculate the correlation coefficient between items inventory and price within the same group. Example command:

```
select warehouseId, corr(items,price) from warehouse group by warehouseId;
```

  Result:

```
warehouseId _c1
city1   0.43124
city2   0.53344
city3   0.73425
```

15.4.3 count

Function

The **count** function is used to return the number of records.

Syntax

```
count([distinct|all] <colname>)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Description
distinct or all	No	Specifies whether to remove duplicate records when counting. Default is **all**, which counts all records. If **distinct** is specified, only unique values are counted.
colname	Yes	Column values can be of any type. **colname** can be *, i.e., **count(*)**, which returns the total number of rows.

Return Values

Returns a value of type BIGINT.

NOTE

If **colname** is NULL, that row is excluded from the calculation.

Examples

- Count the number of records in the warehouse table. Example command:
  ```
  select count(*) from warehouse;
  ```
  Result:
  ```
  _c0
  10
  ```
- Used with GROUP BY to group all items by warehouse (warehouseId) and count the number of items in each warehouse. Example command:
  ```
  select warehouseId, count(*) from warehouse group by warehouseId;
  ```
  Result:
  ```
  warehouseId _c1
  city1   6
  city2   5
  city3   6
  ```
- Use distinct to deduplicate and count the number of warehouses. Example command:
  ```
  select count(distinct warehouseId) from warehouse;
  ```
  Result:
  ```
  _c0
  3
  ```

15.4.4 covar_pop

Function

The **covar_pop** function is used to return the population covariance of two numeric columns.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
covar_pop(col1, col2)
```

Parameters

Parameter	Mandatory	Description
col1	Yes	A column with numeric data type. Other types return NULL.
col2	Yes	A column with numeric data type. Other types return NULL.

Return Values

Returns a value of type DOUBLE.

Examples

- Calculate the population covariance between all items inventory and price. Example command:
  ```
  select covar_pop(items,price) from warehouse;
  ```
  Result:
  ```
  _c0
  1.242355
  ```
- Used with GROUP BY to group all items by warehouse (warehouseId) and calculate the population covariance between items inventory and price within the same group. Example command:
  ```
  select warehouseId, covar_pop(items,price) from warehouse group by warehouseId;
  ```
  Result:
  ```
  warehouseId _c1
  city1    1.13124
  city2    1.13344
  city3    1.53425
  ```

15.4.5 covar_samp

Function

The **covar_samp** function is used to return the sample covariance of two numeric columns.

Syntax

```
covar_samp(col1, col2)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Description
col1	Yes	A column with numeric data type. Other types return NULL.
col2	Yes	A column with numeric data type. Other types return NULL.

Return Values

Returns a value of type DOUBLE.

Examples

- Calculate the sample covariance between all items inventory and price. Example command:
  ```
  select covar_samp(items,price) from warehouse;
  ```
  Result:
  ```
  _c0
  1.242355
  ```
- Used with GROUP BY to group all items by warehouse (warehouseId) and calculate the sample covariance between items inventory and price within the same group. Example command:
  ```
  select warehouseId, covar_samp(items,price) from warehouse group by warehouseId;
  ```
  Result:
  ```
  warehouseId _c1
  city1    1.03124
  city2    1.03344
  city3    1.33425
  ```

15.4.6 max

Function

The **max** function is used to return the maximum value.

Syntax

```
max(col)
```

Parameters

Parameter	Mandatory	Description
col	Yes	Column values can be of any type except BOOLEAN.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type DOUBLE.

NOTE

The return value type is the same as the **col** type. Return rules are as follows:

- If **col** is NULL, that row is excluded from the calculation.
- If **col** is of type BOOLEAN, it is not allowed in the computation.

Examples

- Calculate the maximum inventory (items) across all items. Example command:

```
select max(items) from warehouse;
```

Result:

```
_c0
900
```

- Used with GROUP BY to find the maximum inventory for each warehouse. Example command:

```
select warehouseId, max(items) from warehouse group by warehouseId;
```

Result:

```
warehouseId _c1
city1    200
city2    300
city3    400
```

15.4.7 min

Function

The **min** function is used to return the minimum value.

Syntax

```
min(col)
```

Parameters

Parameter	Mandatory	Type	Description
col	Yes	Any type except BOOLEAN	Column values can be of any type except BOOLEAN.

Return Values

Returns a value of type DOUBLE.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

The return value type is the same as the **col** type. Return rules are as follows:

- If **col** is NULL, that row is excluded from the calculation.
- If **col** is of type BOOLEAN, it is not allowed in the computation.

Examples

- Calculate the minimum inventory (items) across all items. Example command:
  ```
  select min(items) from warehouse;
  ```
  Result:
  ```
  _c0
  600
  ```
- Used with GROUP BY to find the minimum inventory for each warehouse. Example command:
  ```
  select warehouseId, min(items) from warehouse group by warehouseId;
  ```
  Result:
  ```
  warehouseId _c1
  city1	15
  city2	10
  city3	19
  ```

15.4.8 percentile

Function

The **percentile** function is used to calculate the exact percentile, suitable for small data volumes. It first sorts the specified column in ascending order, then returns the exact value at the p-th percentile.

Syntax

```
percentile(colname,DOUBLE p)
```

Parameters

Parameter	Mandatory	Type	Description
colname	Yes	STRING	Represents the column name to be sorted. Elements in the column must be of integer type.
p	Yes	DOUBLE	The range of **p** is 0–1. The format of parameter **p** includes floating-point format.

Return Values

Returns a value of type DOUBLE or ARRAY.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If the column name does not exist, an error is returned.
- If **p** is NULL or outside [0, 1], an error is returned.

Examples

Assume the elements in column **int_test** are 1, 2, 3, 4, of type INT.

Returns **3.0999999999999996**.

```
select percentile(int_test,0.7) FROM int_test;
```

Returns **2.5**.

```
select percentile(int_test,0.5) FROM int_test;
```

Returns **[1.3, 1.9, 2.5, 2.8, 3.7]**.

```
select percentile (int_test,ARRAY(0.1,0.3,0.5,0.6,0.9)) FROM int_test;
```

15.4.9 percentile_approx

Function

The **percentile_approx** function is used to calculate the approximate percentile, suitable for large data volumes. It first sorts the specified column in ascending order, then returns the value closest to the p-th percentile.

Syntax

```
percentile_approx(DOUBLE col, p [, B])
```

Parameters

Parameter	Mandatory	Description
col	Yes	A column with numeric data type. Other types return NULL.
p	Yes	0 <= P <= 1, otherwise returns NULL.
B	No	Parameter **B** controls the approximation accuracy. The larger the **B** value, the higher the accuracy. The default value is **10000**. When the number of distinct values in the column is less than **B**, the exact percentile is returned.

Return Values

Returns a value of type DOUBLE.

Examples

- Calculate the 0.5 percentile of all items inventory with accuracy 100. Example command: Assume the elements in column int_test are 1, 2, 3, 4, of type INT.

 Copyright © Huawei Technologies Co., Ltd.

```
select PERCENTILE_APPROX(items,0.5,100) from warehouse;
```

Result:

```
+------------+
| _c0        |
+------------+
| 10         |
+------------+
```

- Used with GROUP BY to group all items by warehouse (warehouseId) and calculate the 0.5 percentile of items inventory within the same group with accuracy 100. Example command:

```
select warehouseId, PERCENTILE_APPROX(items, 0.5, 100) from warehouse group by warehouseId;
```

Result:

```
+------------+------------+
| warehouseId| _c1        |
+------------+------------+
| city1    | 3     |
| city2    | 20     |
| city3    | 200     |
+------------+------------+
```

15.4.10 stddev_pop

Function

The **stddev_pop** function is used to return the population standard deviation of the specified column.

Syntax

```
stddev_pop(col)
```

Parameters

Parameter	Mandatory	Description
col	Yes	A column with numeric data type. Other types return NULL.

Return Values

Returns a value of type DOUBLE.

Examples

- Calculate the population standard deviation of all items inventory. Example command:

```
select stddev_pop(items) from warehouse;
```

Result:

```
_c0
91.49358659976605
```

- Used with GROUP BY to group all items by warehouse (warehouseId) and calculate the population standard deviation of items inventory within the same group. Example command:

 Copyright © Huawei Technologies Co., Ltd.

```
select warehouseId, stddev_pop(items) from warehouse group by warehouseId;
```

Result:

```
warehouseId _c1
city1    1.4142135623730951
city2    11.180339887498949
city3    81.64965809277261
```

15.4.11 stddev_samp

Function

The **stddev_samp** function is used to return the sample standard deviation of the specified column.

Syntax

```
stddev_samp(col)
```

Parameters

Parameter	Mandatory	Description
col	Yes	A column with numeric data type. Other types return NULL.

Return Values

Returns a value of type DOUBLE.

Examples

- Calculate the sample standard deviation of all items inventory. Example command:
  ```
  select stddev_samp(items) from warehouse;
  ```

  Result:

  ```
  +------------+
  | _c0        |
  +------------+
  | 1.342355   |
  +------------+
  ```

- Used with GROUP BY to group all items by warehouse (warehouseId) and calculate the sample standard deviation of items inventory within the same group. Example command:
  ```
  select warehouseId, stddev_samp(items) from warehouse group by warehouseId;
  ```

  Result:

  ```
  +------------+------------+
  | warehouseId| _c1        |
  +------------+------------+
  | city1    | 1.23124   |
  | city2    | 1.23344   |
  | city3    | 1.43425   |
  +------------+------------+
  ```

 Copyright © Huawei Technologies Co., Ltd.

15.4.12 sum

Function

The **sum** function is used to calculate the sum.

Syntax

```
sum(col),
sum(DISTINCT col)
```

Parameters

Parameter	Mandatory	Description
col	Yes	Column values support all data types and can be converted to DOUBLE type for computation. Column values can be of type DOUBLE, DECIMAL, BIGINT, etc. If the input is of type STRING, it is implicitly converted to DOUBLE type before computation.

Return Values

Returns a value of type DOUBLE.

NOTE

If **col** is NULL, that row is excluded from the calculation.

Examples

- Calculate the total sum of items across all warehouses. Example command:
  ```
  select sum(items) from warehouse;
  ```
  Result:
  ```
  _c0
  55357
  ```
- Used with GROUP BY to group all items by warehouse (warehouseId) and calculate the total sum of items for each warehouse. Example command:
  ```
  select warehouseId, sum(items) from warehouse group by warehouseId;
  ```
  Result:
  ```
  warehouseId| _c1
  city1    15500
  city2    10175
  city3    19400
  ```

 Copyright © Huawei Technologies Co., Ltd.

15.4.13 variance/var_pop

Function

The **variance/var_pop** function is used to return the population variance of a column.

Syntax

```
variance(col),
var_pop(col)
```

Parameters

Parameter	Mandatory	Description
col	Yes	A column with numeric data type. Columns of other types return NULL.

Return Values

Returns a value of type DOUBLE.

Examples

- Calculate the variance of all items inventory. Example command:

  ```
  select variance(items) from warehouse;
  -- Equivalent to the following statement.
  select var_pop(items) from warehouse;
  ```

  Result:

  ```
  _c0
  203.42352
  ```

- Used with GROUP BY to group all items by warehouse (warehouseId) and calculate the variance of items inventory within the same group. Example command:

  ```
  select warehouseId, variance(items) from warehouse group by warehouseId;
  -- Equivalent to the following statement.
  select warehouseId, var_pop(items) from warehouse group by warehouseId;
  ```

  Result:

  ```
  warehouseId _c1
  city1    19.23124
  city2    17.23344
  city3    12.43425
  ```

15.4.14 var_samp

Function

The **var_samp** function is used to return the sample variance of the specified column.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
var_samp(col)
```

Parameters

Parameter	Mandatory	Description
col	Yes	A column with numeric data type. Other types return NULL.

Return Values

Returns a value of type DOUBLE.

Examples

- Calculate the sample variance of all items inventory. Example command:
  ```
  select var_samp(items) from warehouse;
  ```
  Result:
  ```
  _c0
  294.342355
  ```
- Used with GROUP BY to group all items by warehouse (warehouseId) and calculate the sample variance of items inventory within the same group. Example command:
  ```
  select warehouseId, var_samp(items) from warehouse group by warehouseId;
  ```
  Result:
  ```
  warehouseId _c1
  city1    18.23124
  city2    16.23344
  city3    11.43425
  ```

15.4.15 median

Function

The **median** function is used to calculate the median of the input parameters.

Syntax

```
median(colname)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
colname	Yes	DOUBLE, DECIMAL, STRING, or BIGINT	Represents the column name to be sorted. Elements in the column are of type DOUBLE. When elements in the column are not of type DOUBLE, they are implicitly converted to DOUBLE type before computation.

Return Values

Returns a value of type DOUBLE or DECIMAL.

NOTE

If the column name does not exist, an error is returned.

Examples

Assume the elements in column **int_test** are 1, 2, 3, 4, of type INT.

Returns **2.5**.

```
select median(int_test) FROM int_test;
```

15.5 Window Functions

15.5.1 cume_dist

Function

The **cume_dist** function is used to calculate the cumulative distribution, which is equivalent to calculating the proportion of data greater than or equal to, or less than or equal to the current row within a partition.

Usage Restrictions

The usage restrictions for window functions are as follows:

- Window functions can only appear in SELECT statements.
- Window functions cannot be nested with other window functions or aggregate functions.
- Window functions cannot be used together with aggregate functions at the same level.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
cume_dist() over([partition_clause] [orderby_clause])
```

Parameters

Parameter	Mandatory	Description
partition_clause	No	Specifies the partition. Rows with the same value in the partition column are considered to be within the same window.
orderby_clause	No	Specifies how data is sorted within a window.

Return Values

Returns a value of type DOUBLE.

Examples

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the **salary** table and add data with the following example command:

```
CREATE EXTERNAL TABLE salary (
dept STRING, -- <department-name>
userid string, -- <employee-ID>
sal INT -- <salary>
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
stored as textfile;
```

Add data as follows:

```
d1,user1,1000
d1,user2,2000
d1,user3,3000
d2,user4,4000
d2,user5,5000
```

- Calculate the proportion of people with salaries less than or equal to the current salary.

  ```
  select dept, userid, sal,
        cume_dist() over(order by sal) as cume1
  from salary;
  --  Result:
  d1 user1 1000 0.2
  d1 user2 2000 0.4
  d1 user3 3000 0.6
  d2 user4 4000 0.8
  d2 user5 5000 1.0
  ```

- Group by department and calculate the proportion of employees with salaries less than or equal to the current salary within each department.

  ```
  select dept, userid, sal,
        cume_dist() over (partition by dept order by sal) as cume2
  from salary;
  -- Result:
  d1 user1 1000 0.3333333333333333
  d1 user2 2000 0.6666666666666666
  d1 user3 3000 1.0
  ```

 Copyright © Huawei Technologies Co., Ltd.

```
d2 user4 4000 0.5
d2 user5 5000 1.0
```

- After sorting by **sal** in descending order, the result calculates the proportion of employees with salaries greater than or equal to the current salary.

```
select dept, userid, sal,
       cume_dist() over(order by sal desc) as cume3
from salary;
 -- Result:
d2 user5 5000 0.2
d2 user4 4000 0.4
d1 user3 3000 0.6
d1 user2 2000 0.8
d1 user1 1000 1.0 select dept, userid, sal,
       cume_dist() over(partition by dept order by sal desc) as cume4
from salary;
 -- Result:
d1 user3 3000 0.3333333333333333
d1 user2 2000 0.6666666666666666
d1 user1 1000 1.0
d2 user5 5000 0.5
d2 user4 4000 1.0
```

15.5.2 first_value

Function

The **first_value** function is used to retrieve the value of the first row in the window corresponding to the current row.

Usage Restrictions

The usage restrictions for window functions are as follows:

- Window functions can only appear in SELECT statements.
- Window functions cannot be nested with other window functions or aggregate functions.
- Window functions cannot be used together with aggregate functions at the same level.

Syntax

```
first_value(expr [, <ignore_nulls>]) over ([partition_clause] [orderby_clause] [frame_clause])'
```

Parameters

Parameter	Mandatory	Description
expr	Yes	The expression for which the result is to be computed and returned.
ignore_nulls	No	BOOLEAN type, specifies whether to ignore NULL values. The default value is **False**. When the parameter value is **True**, the first non-NULL value in the window is returned.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
partition_clause	No	Specifies the partition. Rows with the same value in the partition column are considered to be within the same window.
orderby_clause	No	Specifies how data is sorted within a window.
frame_clause	No	Used to determine the data boundaries.

Return Values

The data type of the parameter.

Examples

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the **logs** table and add data with the following example command:

```
create table logs(
  cookieid string,
  createtime string,
  url string )
STORED AS parquet;
```

Add data as follows:

```
cookie1 2015-04-10 10:00:02 url2
cookie1 2015-04-10 10:00:00 url1
cookie1 2015-04-10 10:03:04 url3
cookie1 2015-04-10 10:50:05 url6
cookie1 2015-04-10 11:00:00 url7
cookie1 2015-04-10 10:10:00 url4
cookie1 2015-04-10 10:50:01 url5
cookie2 2015-04-10 10:00:02 url22
cookie2 2015-04-10 10:00:00 url11
cookie2 2015-04-10 10:03:04 url33
cookie2 2015-04-10 10:50:05 url66
cookie2 2015-04-10 11:00:00 url77
cookie2 2015-04-10 10:10:00 url44
cookie2 2015-04-10 10:50:01 url55
```

Example: Group all records by **cookieid**, sort by **createtime** in ascending order, and return the first row of data in each group. Example command:

```
SELECT cookieid, createtime, url,
       FIRST_VALUE(url) OVER (PARTITION BY cookieid ORDER BY createtime) AS first
FROM logs;
Result:
cookieid createtime         url  first
cookie1 2015-04-10 10:00:00 url1 url1
cookie1 2015-04-10 10:00:02 url2 url1
cookie1 2015-04-10 10:03:04 url3 url1
cookie1 2015-04-10 10:10:00 url4 url1
cookie1 2015-04-10 10:50:01 url5 url1
cookie1 2015-04-10 10:50:05 url6 url1
cookie1 2015-04-10 11:00:00 url7 url1
cookie2 2015-04-10 10:00:00 url11 url11
cookie2 2015-04-10 10:00:02 url22 url11
cookie2 2015-04-10 10:03:04 url33 url11
cookie2 2015-04-10 10:10:00 url44 url11
```

 Copyright © Huawei Technologies Co., Ltd.

```
cookie2 2015-04-10 10:50:01 url55 url11
cookie2 2015-04-10 10:50:05 url66 url11
cookie2 2015-04-10 11:00:00 url77 url11
```

15.5.3 last_value

Function

The **last_value** function is used to retrieve the value of the last row in the window corresponding to the current row.

Usage Restrictions

The usage restrictions for window functions are as follows:

- Window functions can only appear in SELECT statements.
- Window functions cannot be nested with other window functions or aggregate functions.
- Window functions cannot be used together with aggregate functions at the same level.

Syntax

```
last_value(<expr>[, <ignore_nulls>]) over ([partition_clause] [orderby_clause] [frame_clause])
```

Parameters

Parameter	Mandatory	Description
expr	Yes	The expression for which the result is to be computed and returned.
ignore_nulls	No	BOOLEAN type, specifies whether to ignore NULL values. The default value is **False**. When the parameter value is **True**, the last non-NULL value in the window is returned.
partition_clause	No	Specifies the partition. Rows with the same value in the partition column are considered to be within the same window.
orderby_clause	No	Specifies how data is sorted within a window.
frame_clause	No	Used to determine the data boundaries.

Return Values

The data type of the parameter.

 Copyright © Huawei Technologies Co., Ltd.

Examples

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the logs table and add data with the following example command:

```
create table logs(
  cookieid string,
  createtime string,
  url string
)
STORED AS parquet;
```

Add data as follows:

```
cookie1 2015-04-10 10:00:02 url2
cookie1 2015-04-10 10:00:00 url1
cookie1 2015-04-10 10:03:04 url3
cookie1 2015-04-10 10:50:05 url6
cookie1 2015-04-10 11:00:00 url7
cookie1 2015-04-10 10:10:00 url4
cookie1 2015-04-10 10:50:01 url5
cookie2 2015-04-10 10:00:02 url22
cookie2 2015-04-10 10:00:00 url11
cookie2 2015-04-10 10:03:04 url33
cookie2 2015-04-10 10:50:05 url66
cookie2 2015-04-10 11:00:00 url77
cookie2 2015-04-10 10:10:00 url44
cookie2 2015-04-10 10:50:01 url55
```

Example: Group all records by **cookieid**, sort by **createtime** in ascending order, and return the last row of data in each group. Example command:

```
SELECT cookieid, createtime, url,
       LAST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS last
FROM logs;
-- Result:
cookieid createtime         url last
cookie1 2015-04-10 10:00:00 url1 url1
cookie1 2015-04-10 10:00:02 url2 url2
cookie1 2015-04-10 10:03:04 url3 url3
cookie1 2015-04-10 10:10:00 url4 url4
cookie1 2015-04-10 10:50:01 url5 url5
cookie1 2015-04-10 10:50:05 url6 url6
cookie1 2015-04-10 11:00:00 url7 url7
cookie2 2015-04-10 10:00:00 url11 url11
cookie2 2015-04-10 10:00:02 url22 url22
cookie2 2015-04-10 10:03:04 url33 url33
cookie2 2015-04-10 10:10:00 url44 url44
cookie2 2015-04-10 10:50:01 url55 url55
cookie2 2015-04-10 10:50:05 url66 url66
cookie2 2015-04-10 11:00:00 url77 url77
```

Note: The last value up to the current row is essentially the current row itself.

15.5.4 lag

Function

The **lag** function is used to retrieve the value from the *n*th row above the current row within the window.

Usage Restrictions

The usage restrictions for window functions are as follows:

 Copyright © Huawei Technologies Co., Ltd.

- Window functions can only appear in SELECT statements.
- Window functions cannot be nested with other window functions or aggregate functions.
- Window functions cannot be used together with aggregate functions at the same level.

Syntax

```
lag(<expr>[, bigint <offset>[, <default>]]) over([partition_clause] orderby_clause)
```

Parameters

Parameter	Mandatory	Description
expr	Yes	The expression for which the result is to be computed and returned.
offset	No	Offset, a BIGINT type constant with a value greater than or equal to 0. A value of **0** indicates the current row, **1** indicates the previous row, and so on. The default value is **1**. If the input value is of type STRING or DOUBLE, it is implicitly converted to BIGINT type before computation.
default	No	A constant, default value is NULL. The default value when the range specified by **offset** exceeds the boundary. Must be of the same data type as **expr**. If **expr** is not a constant, it is evaluated based on the current row.
partition_clause	No	Specifies the partition. Rows with the same value in the partition column are considered to be within the same window.
orderby_clause	No	Specifies how data is sorted within a window.

Return Values

The data type of the parameter.

Examples

Example data:

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the logs table and add data with the following example command:

```
create table logs(
  cookieid string,
  createtime string,
  url string
```

 Copyright © Huawei Technologies Co., Ltd.

```
)
STORED AS parquet;
```

Add data as follows:

```
cookie1 2015-04-10 10:00:02 url2
cookie1 2015-04-10 10:00:00 url1
cookie1 2015-04-10 10:03:04 url3
cookie1 2015-04-10 10:50:05 url6
cookie1 2015-04-10 11:00:00 url7
cookie1 2015-04-10 10:10:00 url4
cookie1 2015-04-10 10:50:01 url5
cookie2 2015-04-10 10:00:02 url22
cookie2 2015-04-10 10:00:00 url11
cookie2 2015-04-10 10:03:04 url33
cookie2 2015-04-10 10:50:05 url66
cookie2 2015-04-10 11:00:00 url77
cookie2 2015-04-10 10:10:00 url44
cookie2 2015-04-10 10:50:01 url55
```

Group all records by **cookieid**, sort by **createtime** in ascending order, and return the value from the 2nd row above the current row within the window. Example command:

Example 1:

```
SELECT cookieid, createtime, url,
       LAG(createtime, 2) OVER (PARTITION BY cookieid ORDER BY createtime) AS last_2_time
FROM logs;
-- Result:
cookieid createtime         url  last_2_time
cookie1 2015-04-10 10:00:00 url1 NULL
cookie1 2015-04-10 10:00:02 url2 NULL
cookie1 2015-04-10 10:03:04 url3 2015-04-10 10:00:00
cookie1 2015-04-10 10:10:00 url4 2015-04-10 10:00:02
cookie1 2015-04-10 10:50:01 url5 2015-04-10 10:03:04
cookie1 2015-04-10 10:50:05 url6 2015-04-10 10:10:00
cookie1 2015-04-10 11:00:00 url7 2015-04-10 10:50:01
cookie2 2015-04-10 10:00:00 url11 NULL
cookie2 2015-04-10 10:00:02 url22 NULL
cookie2 2015-04-10 10:03:04 url33 2015-04-10 10:00:00
cookie2 2015-04-10 10:10:00 url44 2015-04-10 10:00:02
cookie2 2015-04-10 10:50:01 url55 2015-04-10 10:03:04
cookie2 2015-04-10 10:50:05 url66 2015-04-10 10:10:00
cookie2 2015-04-10 11:00:00 url77 2015-04-10 10:50:01
```

Note: Because no default value was set, NULL is displayed when there are no two rows above.

Example 2:

```
SELECT cookieid, createtime, url,
       LAG(createtime,1,'1970-01-01 00:00:00') OVER (PARTITION BY cookieid ORDER BY createtime) AS
last_1_time
FROM logs;
-- Result:
cookieid createtime         url last_1_time cookie1 2015-04-10 10:00:00 url1 1970-01-01 00:00:00 (default
value displayed)
cookie1 2015-04-10 10:00:02 url2 2015-04-10 10:00:00
cookie1 2015-04-10 10:03:04 url3 2015-04-10 10:00:02
cookie1 2015-04-10 10:10:00 url4 2015-04-10 10:03:04
cookie1 2015-04-10 10:50:01 url5 2015-04-10 10:10:00
cookie1 2015-04-10 10:50:05 url6 2015-04-10 10:50:01
cookie1 2015-04-10 11:00:00 url7 2015-04-10 10:50:05
cookie2 2015-04-10 10:00:00 url11 1970-01-01 00:00:00 (default value displayed)
cookie2 2015-04-10 10:00:02 url22 2015-04-10 10:00:00
cookie2 2015-04-10 10:03:04 url33 2015-04-10 10:00:02
cookie2 2015-04-10 10:10:00 url44 2015-04-10 10:03:04
cookie2 2015-04-10 10:50:01 url55 2015-04-10 10:10:00
```

 Copyright © Huawei Technologies Co., Ltd.

```
cookie2 2015-04-10 10:50:05 url66 2015-04-10 10:50:01
cookie2 2015-04-10 11:00:00 url77 2015-04-10 10:50:05
```

15.5.5 lead

Function

The **lead** function is used to retrieve the value from the *n*th row below the current row within the window.

Usage Restrictions

The usage restrictions for window functions are as follows:

- Window functions can only appear in SELECT statements.
- Window functions cannot be nested with other window functions or aggregate functions.
- Window functions cannot be used together with aggregate functions at the same level.

Syntax

```
lead(<expr>[, bigint <offset>[, <default>]]) over([partition_clause] orderby_clause)
```

Parameters

Parameter	Mandatory	Description
expr	Yes	The expression for which the result is to be computed and returned.
offset	No	Offset, a BIGINT type constant with a value greater than or equal to 0. A value of **0** indicates the current row, **1** indicates the next row, and so on. The default value is **1**. If the input value is of type STRING or DOUBLE, it is implicitly converted to BIGINT type before computation.
default	No	A constant, default value is NULL. The default value when the range specified by **offset** exceeds the boundary. Must be of the same data type as **expr**. If **expr** is not a constant, it is evaluated based on the current row.
partition_clause	No	Specifies the partition. Rows with the same value in the partition column are considered to be within the same window.
orderby_clause	No	Specifies how data is sorted within a window.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

The data type of the parameter.

Examples

Example data:

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the logs table and add data with the following example command:

```
create table logs(
  cookieid string,
  createtime string,
  url string )
STORED AS parquet;
```

Add data as follows:

```
cookie1 2015-04-10 10:00:02 url2
cookie1 2015-04-10 10:00:00 url1
cookie1 2015-04-10 10:03:04 url3
cookie1 2015-04-10 10:50:05 url6
cookie1 2015-04-10 11:00:00 url7
cookie1 2015-04-10 10:10:00 url4
cookie1 2015-04-10 10:50:01 url5
cookie2 2015-04-10 10:00:02 url22
cookie2 2015-04-10 10:00:00 url11
cookie2 2015-04-10 10:03:04 url33
cookie2 2015-04-10 10:50:05 url66
cookie2 2015-04-10 11:00:00 url77
cookie2 2015-04-10 10:10:00 url44
cookie2 2015-04-10 10:50:01 url55
```

Group all records by **cookieid**, sort by **createtime** in ascending order, and return the values from the 2nd row and 1st row below the current row within the window. Example command:

```
SELECT cookieid, createtime, url,
      LEAD(createtime, 2) OVER(PARTITION BY cookieid ORDER BY createtime) AS next_2_time,
      LEAD(createtime, 1, '1970-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY createtime) AS
next_1_time
FROM logs;

-- Result:
cookieid createtime         url  next_2_time          next_1_time
cookie1 2015-04-10 10:00:00 url1 2015-04-10 10:03:04  2015-04-10 10:00:02
cookie1 2015-04-10 10:00:02 url2 2015-04-10 10:10:00  2015-04-10 10:03:04
cookie1 2015-04-10 10:03:04 url3 2015-04-10 10:50:01  2015-04-10 10:10:00
cookie1 2015-04-10 10:10:00 url4 2015-04-10 10:50:05  2015-04-10 10:50:01
cookie1 2015-04-10 10:50:01 url5 2015-04-10 11:00:00  2015-04-10 10:50:05
cookie1 2015-04-10 10:50:05 url6 NULL                 2015-04-10 11:00:00
cookie1 2015-04-10 11:00:00 url7 NULL                 1970-01-01 00:00:00
cookie2 2015-04-10 10:00:00 url11 2015-04-10 10:03:04 2015-04-10 10:00:02
cookie2 2015-04-10 10:00:02 url22 2015-04-10 10:10:00 2015-04-10 10:03:04
cookie2 2015-04-10 10:03:04 url33 2015-04-10 10:50:01 2015-04-10 10:10:00
cookie2 2015-04-10 10:10:00 url44 2015-04-10 10:50:05 2015-04-10 10:50:01
cookie2 2015-04-10 10:50:01 url55 2015-04-10 11:00:00 2015-04-10 10:50:05
cookie2 2015-04-10 10:50:05 url66 NULL                 2015-04-10 11:00:00
cookie2 2015-04-10 11:00:00 url77 NULL                 1970-01-01 00:00:00
```

 Copyright © Huawei Technologies Co., Ltd.

15.5.6 percent_rank

Function

The **percent_rank** function is used to return the value of the column specified in the ORDER BY clause of the window, but expressed as a decimal between 0 and 1. The calculation method is (RANK value of the current row within the group – 1)/(Total number of rows within the group – 1).

Usage Restrictions

The usage restrictions for window functions are as follows:

- Window functions can only appear in SELECT statements.
- Window functions cannot be nested with other window functions or aggregate functions.
- Window functions cannot be used together with aggregate functions at the same level.

Syntax

```
percent_rank() over([partition_clause] [orderby_clause])
```

Parameters

Parameter	Mandatory	Description
partition_clause	No	Specifies the partition. Rows with the same value in the partition column are considered to be within the same window.
orderby_clause	No	Specifies how data is sorted within a window.

Return Values

Returns a value of type DOUBLE.

Examples

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the **salary** table and add data with the following example command:

```
CREATE EXTERNAL TABLE salary (
dept STRING, -- <department-name>
userid string, -- <employee-ID>
sal INT -- <salary>
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
stored as textfile;
```

Add data as follows:

 Copyright © Huawei Technologies Co., Ltd.

```
d1,user1,1000
d1,user2,2000
d1,user3,3000
d2,user4,4000
d2,user5,5000
```

Example: Calculate the percentage rank of employee salaries within their departments.

```
select dept, userid, sal,
      percent_rank() over(partition by dept order by sal) as pr2
from salary;
-- Result analysis:
d1 user1 1000 0.0    -- (1-1)/(3-1)=0.0
d1 user2 2000 0.5    -- (2-1)/(3-1)=0.5
d1 user3 3000 1.0    -- (3-1)/(3-1)=1.0
d2 user4 4000 0.0    -- (1-1)/(2-1)=0.0
d2 user5 5000 1.0    -- (2-1)/(2-1)=1.0
```

15.5.7 rank

Function

The **rank** function is used to calculate the rank of a value within a set of values. If there are ties, the **rank** function leaves gaps in the ranking sequence.

Usage Restrictions

The usage restrictions for window functions are as follows:

- Window functions can only appear in SELECT statements.
- Window functions cannot be nested with other window functions or aggregate functions.
- Window functions cannot be used together with aggregate functions at the same level.

Syntax

```
rank() over ([partition_clause] [orderby_clause])
```

Parameters

Parameter	Mandatory	Description
partition_clause	No	Specifies the partition. Rows with the same value in the partition column are considered to be within the same window.
orderby_clause	No	Specifies how data is sorted within a window.

 Copyright © Huawei Technologies Co., Ltd.

Return Values

Returns a value of type INT.

Examples

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the logs table and add data with the following example command:

```
CREATE TABLE logs (
cookieid string,
createtime string,
pv INT
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
stored as textfile;
```

Add data as follows:

```
cookie1 2015-04-10 1
cookie1 2015-04-11 5
cookie1 2015-04-12 7
cookie1 2015-04-13 3
cookie1 2015-04-14 2
cookie1 2015-04-15 4
cookie1 2015-04-16 4
cookie2 2015-04-10 2
cookie2 2015-04-11 3
cookie2 2015-04-12 5
cookie2 2015-04-13 6
cookie2 2015-04-14 3
cookie2 2015-04-15 9
cookie2 2015-04-16 7
```

Example: Group all records by **cookieid**, sort by **pv** in descending order, and return the sequence number for each row within the group. Example command:

```
select cookieid, createtime, pv,
      rank() over(partition by cookieid order by pv desc) as rank
from logs
where cookieid = 'cookie1';
-- Result:
cookie1 2015-04-12 7 1
cookie1 2015-04-11 5 2
cookie1 2015-04-16 4 3 (tied for third)
cookie1 2015-04-15 4 3
cookie1 2015-04-13 3 5 (skips 4, starts from 5)
cookie1 2015-04-14 2 6
cookie1 2015-04-10 1 7
```

15.5.8 row_number

Function

The **row_number** function is used to calculate the row number. It increments starting from 1.

Usage Restrictions

The usage restrictions for window functions are as follows:

- Window functions can only appear in SELECT statements.

 Copyright © Huawei Technologies Co., Ltd.

- Window functions cannot be nested with other window functions or aggregate functions.
- Window functions cannot be used together with aggregate functions at the same level.

Syntax

```
row_number() over([partition_clause] [orderby_clause])
```

Parameters

Parameter	Mandatory	Description
partition_clause	No	Specifies the partition. Rows with the same value in the partition column are considered to be within the same window.
orderby_clause	No	Specifies how data is sorted within a window.

Return Values

Returns a value of type INT.

Examples

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the logs table and add data with the following example command:

```
CREATE TABLE logs (
cookieid string,
createtime string,
pv INT
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
stored as textfile;
```

Add data as follows:

```
cookie1 2015-04-10 1
cookie1 2015-04-11 5
cookie1 2015-04-12 7
cookie1 2015-04-13 3
cookie1 2015-04-14 2
cookie1 2015-04-15 4
cookie1 2015-04-16 4
cookie2 2015-04-10 2
cookie2 2015-04-11 3
cookie2 2015-04-12 5
cookie2 2015-04-13 6
cookie2 2015-04-14 3
cookie2 2015-04-15 9
cookie2 2015-04-16 7
```

Example: Group all records by **cookieid**, sort by **pv** in descending order, and return the sequence number for each row within the group. Example command:

 Copyright © Huawei Technologies Co., Ltd.

```
select cookieid, createtime, pv,
      row_number() over (partition by cookieid order by pv desc) as index
from logs;

-- Result:
cookie1 2015-04-12 7 1
cookie1 2015-04-11 5 2
cookie1 2015-04-16 4 3
cookie1 2015-04-15 4 4
cookie1 2015-04-13 3 5
cookie1 2015-04-14 2 6
cookie1 2015-04-10 1 7
cookie2 2015-04-15 9 1
cookie2 2015-04-16 7 2
cookie2 2015-04-13 6 3
cookie2 2015-04-12 5 4
cookie2 2015-04-11 3 5
cookie2 2015-04-14 3 6
cookie2 2015-04-10 2 7
```

15.6 Other Functions

15.6.1 decode1

Function

The **decode1** function implements if-then-else branch selection functionality.

Syntax

```
decode1(expression, search, result [, search, result]...[, default])
```

Parameters

Parameter	Mandatory	Type	Description
expression	Yes	All data types	The expression to be compared.
search	Yes	Consistent with **expression**	The search term to compare with the **expression**.
result	Yes	All data types	The return value when the **search** and **expression** values match.
default	No	Consistent with **result**	The default value returned if no search term matches. If not specified, NULL is returned.

Return Values

result and **default** are the return values, supporting all data types.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- If a match is found, **result** is returned.
- If no match is found, **default** is returned.
- If **default** is not specified, **NULL** is returned.
- If search options are duplicated and a match occurs, the first value is returned.

Example Code

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the **salary** table and add data with the following example command:

```
CREATE EXTERNAL TABLE IF NOT EXISTS salary (
  dept_id STRING,  -- <department-ID>
  userid STRING,   -- <employee-ID>
  sal INT          -- <salary>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION 'obs://your-bucket-name/aidatalake/salary/';-- [Mandatory] Replace with OBS bucket path (required when using external tables on the platform)
```

Add data as follows:

```
INSERT INTO salary VALUES
('d1','user1',1000),
('d1','user2',2000),
('d1','user3',3000),
('d2','user4',4000),
('d2','user5',5000),
('d3','user6',6000),
('d4','user7',7000),
('d5','user8',8000);
```

Command Output

Return the name for each department.

When the value of **dept_id** is **d1**, return **Platform**. When the value is **d2**, return **MRS**. For all other cases, return **Others**.

```
select dept_id,
decode1(dept_id, 'd1', 'Platform', 'd2', 'MRS', 'Others') as result from salary
GROUP BY dept_id;  -- Deduplicate by department ID, ensuring only one row per department is returned
```

Result:

```
+-----+-------+
| dept_id| result|
+-----+-------+
|   d1|    Platform|
|   d2|    MRS|
|   d3| Others|
|   d4| Others|
|   d5| Others|
+-----+-------+
```

15.6.2 ordinal

The **ordinal** function is used to sort the input variables in ascending order and return the value at the *n*th specified position.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

```
ordinal(bigint nth, var1, var2 [, ...])
```

Parameters

Parameter	Mandatory	Type	Description
nth	Yes	BIGINT	Specifies the position value to return.
var	Yes	BIGINT, DOUBLE, DATETIME, or STRING	The values to be sorted.

Return Values

Returns a value of type DOUBLE or DECIMAL.

NOTE

- Returns the value at the *n*th position. When no implicit conversion exists, the return value has the same data type as the input parameters.
- When type conversion occurs, conversion between DOUBLE, BIGINT, and STRING returns a DOUBLE type. Conversion between STRING and DATETIME returns a DATETIME type. Other implicit conversions are not allowed.
- NULL is treated as the minimum value.

Example Code

Returns **2**.

```
select ordinal(3, 1, 3, 2, 5, 2, 4, 9);
```

15.6.3 trans_array

The **trans_array** function is a UDTF that converts one row of data into multiple rows, transforming arrays stored in columns with a fixed delimiter format into multiple rows.

Usage Restrictions

- All key columns must be placed in front, while the columns to be transposed must be placed at the back.
- Only one UDTF is allowed in a single SELECT statement, and no other columns can appear.
- Cannot be used with **GROUP BY**, **CLUSTER BY**, **DISTRIBUTE BY**, or **SORT BY**.

Syntax

```
trans_array(num_keys, separator, key1, ..., col1, col2, ...) as (key1, ..., col1, col2, ...)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Parameter	Mandatory	Type	Description
num_keys	Yes	BIGINT	A BIGINT type constant, value must be >= 0. Specifies the number of columns used as keys for transposition when converting to multiple rows.
separator	Yes	STRING	A STRING type constant used as the delimiter to split a string into multiple elements. An error is returned if empty.
keys	Yes	STRING	The columns used as keys during transposition. The number is specified by **num_keys**. If **num_keys** specifies all columns as keys (i.e., **num_keys** equals the total number of columns), only one row is returned.
cols	Yes	STRING	The arrays to be converted into rows. All columns after the **keys** are treated as arrays to be transposed and must be of STRING type.

Return Values

The data type of the parameter.

NOTE

- Returns the transposed rows, with new column names specified by the AS clause.
- The data type of key columns remains unchanged, while all other columns are of STRING type.
- The number of split rows is determined by the array with the most elements. Insufficient elements are filled with NULL.

Example Code

To facilitate understanding of the function's usage, source data is provided along with function-related examples. Create the **salary** table and add data with the following example command:

```
CREATE EXTERNAL TABLE salary (
dept_id STRING, -- <department>
user_id STRING, -- <employee>
sal STRING -- <salary>
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
stored as textfile;
```

Add data as follows:

```
d1,user1/user4,1000/6000
d1,user2/user5,2000/7000
d1,user3/user6,3000
d2,user4/user7,4000
d2,user5/user8,5000/8000
```

 Copyright © Huawei Technologies Co., Ltd.

Execute SQL (When using a slash (/) in SQL statements, the platform console automatically handles escaping, but when calling via API, you need to manually escape it. Otherwise, a parsing failure error will occur.)

```
select trans_array(1, "/", dept_id, user_id, sal) as (dept_id, user_id, sal) from salary;
```

Result:

```
d1,user1,1000/6000
d1,user4,1000/6000
d1,user2,2000/7000
d1,user5,2000/7000
d1,user3,3000
d1,user6,3000
d2,user4,4000
d2,user7,4000
d2,user5,5000/6000
d2,user8,5000/6000
```

 Copyright © Huawei Technologies Co., Ltd.

16 Identifiers

16.1 aggregate_func
16.2 alias
16.3 attr_expr
16.4 attr_expr_list
16.5 attrs_value_set_expr
16.6 boolean_expression
16.7 class_name
16.8 col
16.9 col_comment
16.10 col_name
16.11 col_name_list
16.12 condition
16.13 condition_list
16.14 cte_name
16.15 data_type
16.16 db_comment
16.17 db_name
16.18 else_result_expression
16.19 file_format
16.20 file_path
16.21 function_name
16.22 groupby_expression
16.23 having_condition

 Copyright © Huawei Technologies Co., Ltd.

16.24 hdfs_path
16.25 input_expression
16.26 input_format_classname
16.27 jar_path
16.28 join_condition
16.29 non_equi_join_condition
16.30 number
16.31 num_buckets
16.32 output_format_classname
16.33 partition_col_name
16.34 partition_col_value
16.35 partition_specs
16.36 property_name
16.37 property_value
16.38 regex_expression
16.39 result_expression
16.40 row_format
16.41 select_statement
16.42 separator
16.43 serde_name
16.44 sql_containing_cte_name
16.45 sub_query
16.46 table_comment
16.47 table_name
16.48 table_properties
16.49 table_reference
16.50 view_name
16.51 view_properties
16.52 when_expression
16.53 where_condition
16.54 window_function

 Copyright © Huawei Technologies Co., Ltd.

16.1 aggregate_func

Syntax

None.

Description

Aggregate function.

aggregate_func is typically used in database queries to perform calculations on a set of values and return a single result.

16.2 alias

Syntax

None.

Description

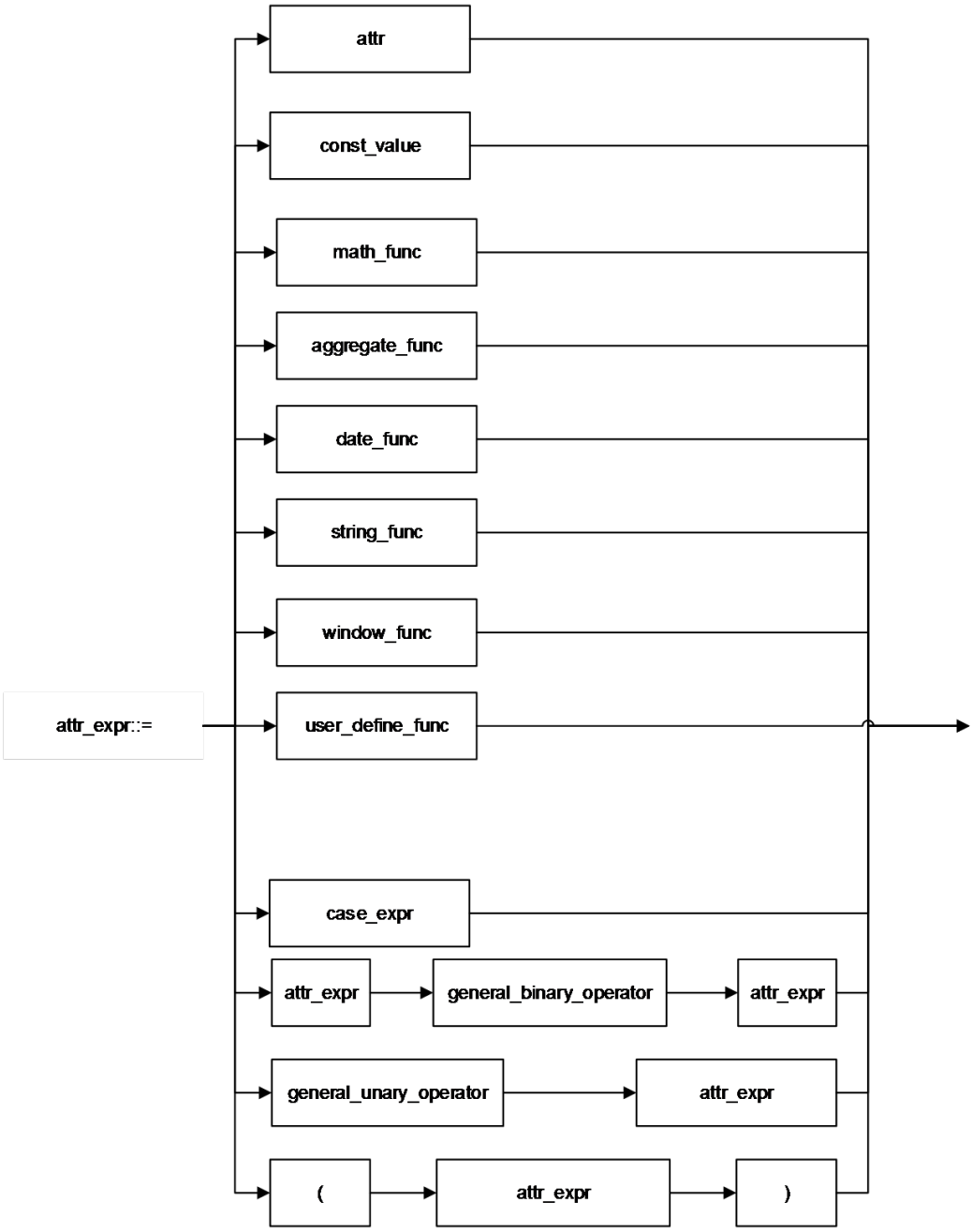
Alias. Can be used to assign an alias to a field, table, view, or subquery. Only supports string type.

16.3 attr_expr

Attribute expression, used in SELECT lists, WHERE conditions, and other scenarios. Can include: field names (attr), constants (const_value), CASE expressions, function calls (mathematical/date/string/aggregate/window/UDF), binary/unary operators, and sub-expressions (enclosed in parentheses).

 Copyright © Huawei Technologies Co., Ltd.

Syntax



Description

Syntax	Description
attr_expr	Attribute expression.
attr	Table field, same as **col_name**.
const_value	Constant value.
case_expr	CASE expression.

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Description
math_func	Mathematical function.
date_func	Date function.
string_func	String function.
aggregate_func	Aggregate function.
window_func	Window function.
user_define_func	User-defined function.
general_binary_operator	General binary operator.
general_unary_operator	General unary operator.
(	Start of a sub-attribute expression.
)	End of a sub-attribute expression.

16.4 attr_expr_list

Attribute expression list.

Syntax

```
attr_expr [, attr_expr ...]
```

Description

A list of **attr_expr**, with multiple attribute expressions separated by commas.

16.5 attrs_value_set_expr

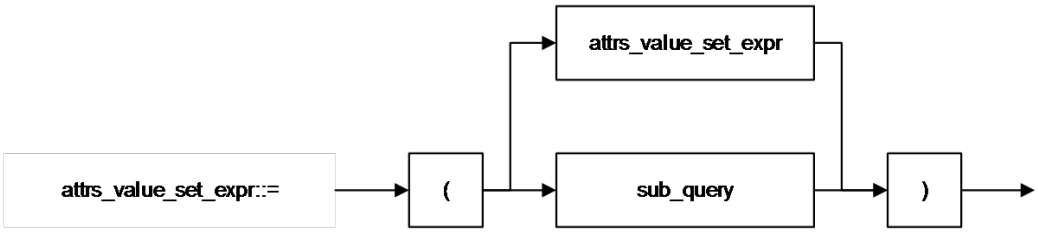
Attribute value set expression, used in the VALUES clause of an INSERT statement. This expression can be either a value set enclosed in parentheses **(value1, value2, ...)** or a subquery **(sub_query)**. When it is a subquery, it must be enclosed in parentheses.

Syntax

Each set of values is enclosed in parentheses, with multiple sets separated by commas.

```
(value1, value2, ...) [, (value1, value2, ...)]
```

 Copyright © Huawei Technologies Co., Ltd.



Description

Syntax	Description
attrs_value_set_expr	Attribute value set.
sub_query	Subquery statement.
(	Start of a subquery expression.
)	End of a subquery expression.

16.6 boolean_expression

Boolean expression, which returns TRUE, FALSE, or NULL.

Syntax

Can include: comparison operators, logical operators (AND/OR/NOT), IN/EXISTS subqueries, IS NULL, etc.

Description

An expression that returns a boolean type.

16.7 class_name

The fully qualified class name of a Java class, used for UDF or SerDe.

Syntax

For example, **org.apache.spark.sql.udf.MyUDF**.

Description

The class name that the function depends on. Note that the class name must include the full path of the package containing the class.

 Copyright © Huawei Technologies Co., Ltd.

16.8 col

Table field, same as **col_name**.

Syntax

None.

Description

A formal parameter in a function call, typically the field name.

16.9 col_comment

Column comment, which is a description of the column.

Syntax

A string constant enclosed in single quotes.

Description

A description of a column (field). Only supports string type. The description length cannot exceed 256 bytes.

16.10 col_name

Syntax

None.

Description

Column name, i.e., the field name. Only supports string type. The name length cannot exceed 128 bytes.

16.11 col_name_list

Syntax

```
col_name [, col_name ...]
```

Description

A field list, consisting of one or more **col_name** values, with multiple **col_name** values separated by commas.

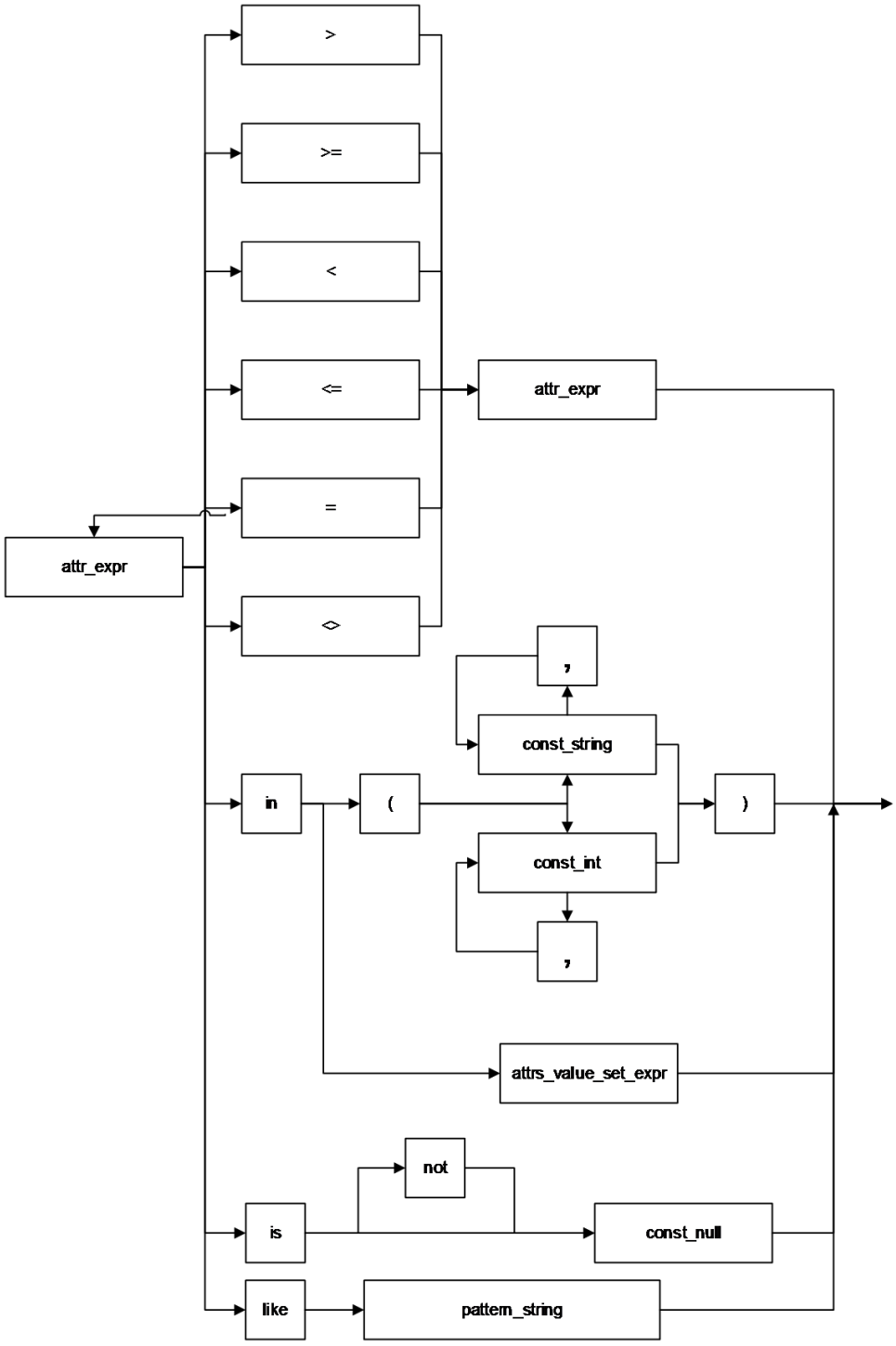
 Copyright © Huawei Technologies Co., Ltd.

16.12 condition

Condition expression, used in WHERE, HAVING, and other clauses.

Can include: comparison operations, logical operations, BETWEEN, IN, LIKE, IS NULL, EXISTS subqueries, etc.

Syntax



 Copyright © Huawei Technologies Co., Ltd.

Description

Syntax	Description
condition	Judgment condition.
>	Relational operator: >
>=	Relational operator: ≥
<	Relational operator: <
<=	Relational operator: ≤
=	Relational operator: =
<>	Relational operator: <>
is	Relational operator: is
is not	Relational operator: is not
const_null	Constant value: null
like	Relational operator: used for wildcard matching.
pattern_string	A pattern matching string that supports wildcard matching. When filtering using the **LIKE** condition, the SQL wildcards **%** and _ are supported, where **%** indicates one or more characters, and _ indicates exactly one character.
attr_expr	Attribute expression.
attrs_value_set_expr	Collection of attribute values.
in	Keyword used to determine whether attributes are in the same collection.
const_string	String constant.
const_int	Integer constant.
(	Start of the specified constant collection.
)	End of the specified constant collection.
,	Separator comma (,)

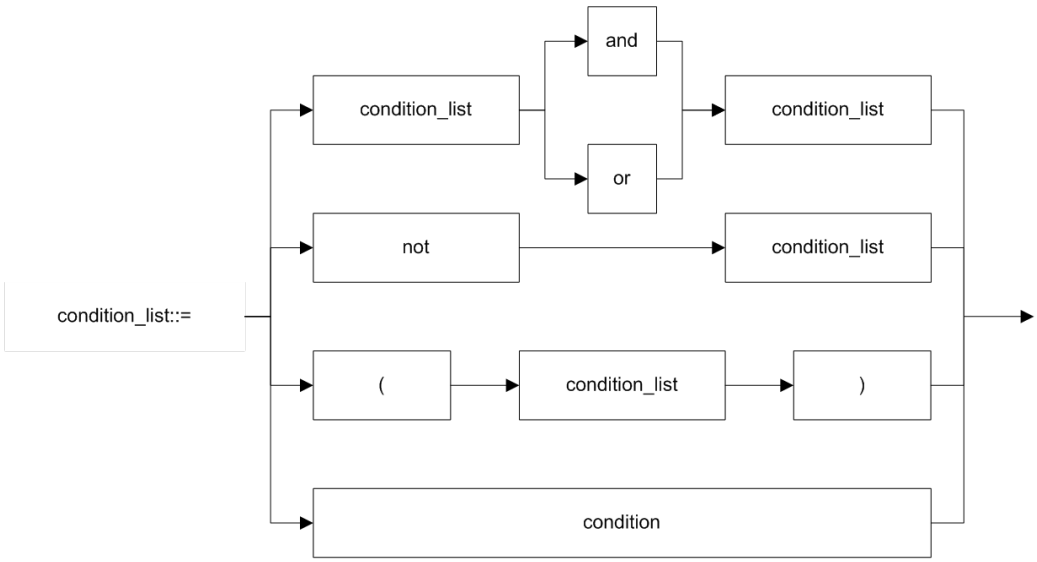
16.13 condition_list

Condition list, with multiple conditions connected by AND/OR.

Syntax

```
condition [AND|OR condition ...]
```

 Copyright © Huawei Technologies Co., Ltd.



Description

Syntax	Description
condition_list	Logical judgment condition list.
and	Logical operator: AND.
or	Logical operator: OR.
not	Logical operator: NOT.
(	Start of a sub-logical judgment condition.
)	End of a sub-logical judgment condition.
condition	Logical judgment condition.

16.14 cte_name

Syntax

None.

Description

The name of a CTE.

16.15 data_type

Data type, specifies the data type of a column or expression. For example, INT, BIGINT, DOUBLE, STRING, DATE, TIMESTAMP, etc.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

None.

Description

Data types include primitive types and complex types (ARRAY, MAP, STRUCT). For details, see **10.3 Complex Data Types**.

16.16 db_comment

Syntax

None.

Description

A description of the database. Only supports string type. The description length cannot exceed 256 bytes.

16.17 db_name

Syntax

None.

Description

Database name. Only supports string type. The name length cannot exceed 128 bytes.

16.18 else_result_expression

Syntax

None.

Description

The return result after the ELSE clause in a CASE WHEN statement.

16.19 file_format

Data file format.

Syntax

| AVRO

 Copyright © Huawei Technologies Co., Ltd.

| CSV

| JSON

| ORC

| PARQUET

Description

- The above five formats are currently supported.
- There are two ways to specify the data format: one is **USING**, which can specify any of the above five data formats; the other is **STORED AS**, which can only specify ORC and PARQUET.
- ORC optimizes RCFile and provides an efficient method for storing Hive data.
- PARQUET is a columnar storage format designed for analytical workloads.

16.20 file_path

Syntax

None.

Description

File path, indicating the storage location of the data file. This path is an OBS path.

16.21 function_name

Syntax

None.

Description

Function name. Only supports string type.

16.22 groupby_expression

Syntax

None.

Description

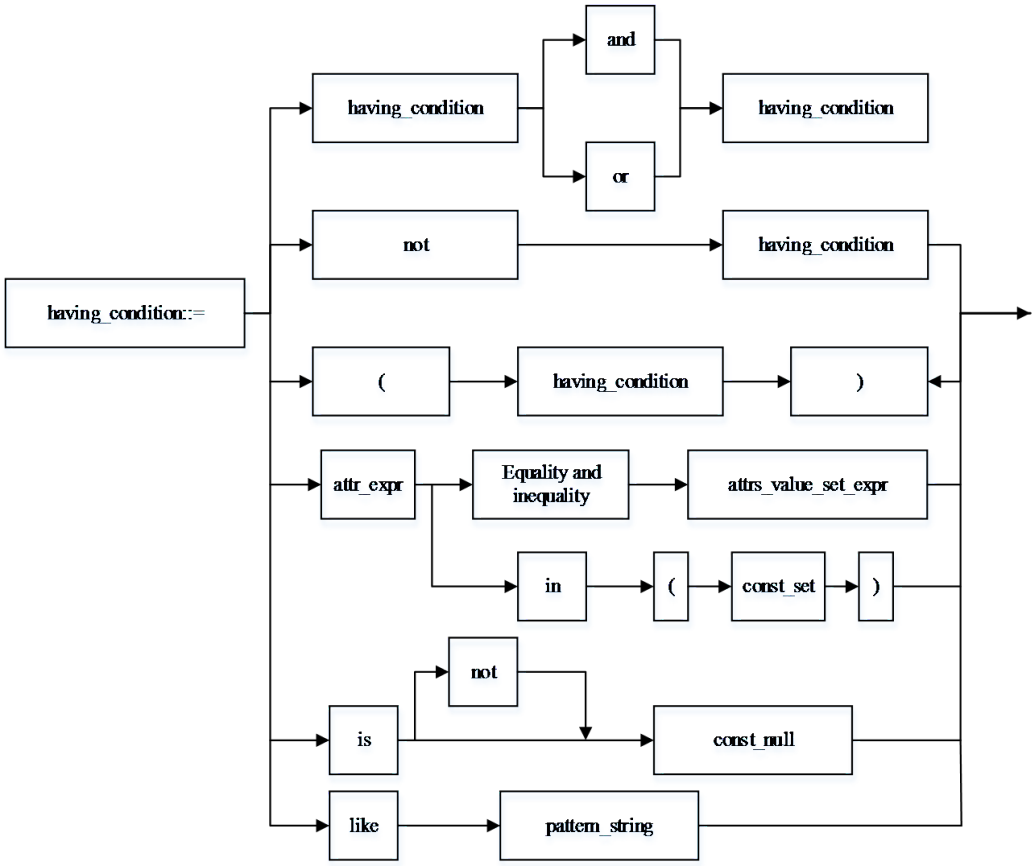
An expression that includes GROUP BY.

 Copyright © Huawei Technologies Co., Ltd.

16.23 having_condition

HAVING condition, used to filter grouped results.

Syntax



Description

Syntax	Description
having_condition	Judgment condition of **having**.
and	Logical operator: AND
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjudgment condition.
)	End of the subjudgment condition.
condition	Judgment condition.
const_set	Collection of constants, which are separated by using comma (,).

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Description
in	Keyword used to determine whether attributes are in the same collection.
attrs_value_set_expr	Collection of attribute values.
attr_expr	Attribute expression.
Equality and inequality	Equality and inequality operators. For details, see **17.1 Relational Operators**.
pattern_string	Pattern matching string, which supports wildcard matching. When filtering using the **LIKE** condition, the SQL wildcards **%** and _ are supported, where **%** indicates one or more characters, and _ indicates exactly one character.
like	Relational operator: used for wildcard matching.

16.24 hdfs_path

Syntax

None.

Description

HDFS path, e.g., **hdfs:///tmp**.

16.25 input_expression

Syntax

None.

Description

The input expression of a CASE WHEN statement.

16.26 input_format_classname

Syntax

None.

 Copyright © Huawei Technologies Co., Ltd.

Description

Specifies the class name of the input format, e.g., **org.apache.hadoop.mapred.TextInputFormat**.

16.27 jar_path

Syntax

None.

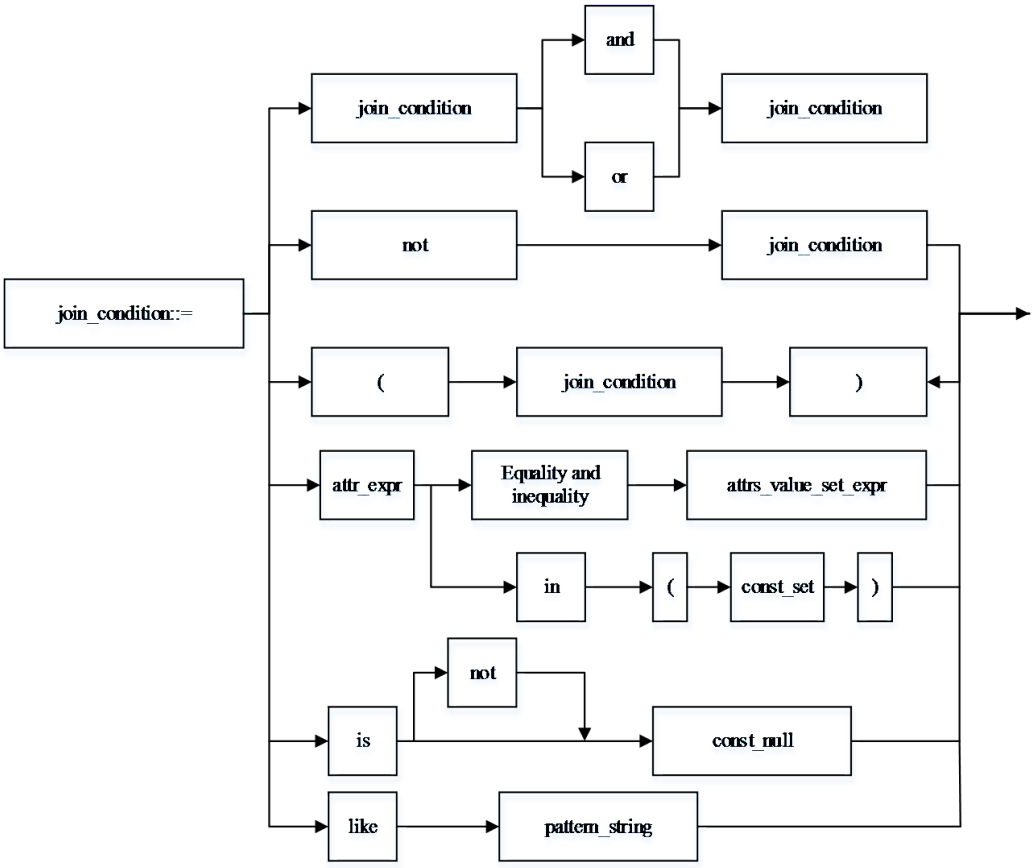
Description

JAR package path. This path can be a local path or an HDFS path.

16.28 join_condition

JOIN condition, which specifies the association rule between two tables.

Syntax



 Copyright © Huawei Technologies Co., Ltd.

Description

Syntax	Description
join_condition	Judgment condition of **join**.
and	Logical operator: AND
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjudgment condition.
)	End of the subjudgment condition.
condition	Judgment condition.
const_set	Collection of constants, which are separated by using comma (,).
in	Keyword used to determine whether attributes are in the same collection.
attrs_value_set_expr	Collection of attribute values.
attr_expr	Attribute expression.
Equality and inequality	Equality and inequality operators. For details, see **17.1 Relational Operators**.
pattern_string	Pattern matching string, which supports wildcard matching. When filtering using the **LIKE** condition, the SQL wildcards **%** and _ are supported, where **%** indicates one or more characters, and _ indicates exactly one character.

16.29 non_equi_join_condition

Syntax

None.

Description

Refers to a non-equi join condition.

A JOIN condition that includes comparison operators such as <, >, <=, >=, <>.

16.30 number

Syntax

None.

 Copyright © Huawei Technologies Co., Ltd.

Description

LIMIT restricts the number of output rows. Only supports INT type.

16.31 num_buckets

Syntax

None.

Description

The number of buckets. Only supports INT type.

Used in **CLUSTERED BY ... INTO N BUCKETS**.

16.32 output_format_classname

Syntax

None.

Description

Specifies the class name of the output format, e.g., **org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat**.

16.33 partition_col_name

Syntax

None.

Description

Partition column name, i.e., the partition field name. Only supports string type.

16.34 partition_col_value

Syntax

None.

Description

Partition column value, i.e., the value of the partition field.

A constant value, enclosed in single quotes (string type) or written directly (numeric type).

 Copyright © Huawei Technologies Co., Ltd.

16.35 partition_specs

Syntax

```
partition_specs : (partition_col_name = partition_col_value, partition_col_name = partition_col_value, ...);
```

Description

The partition list of a table, expressed in key=value pairs, where **key** is **partition_col_name** and **value** is **partition_col_value**. If multiple partition fields exist, each group of key=value is separated by commas (,).

16.36 property_name

Syntax

None.

Description

Property name, used for TBLPROPERTIES or DBPROPERTIES.

Only supports string type.

16.37 property_value

Syntax

None.

Description

Property value. Only supports string type.

16.38 regex_expression

Syntax

None.

Description

Pattern matching string, which supports regular expression matching.

 Copyright © Huawei Technologies Co., Ltd.

16.39 result_expression

Syntax

None.

Description

The return result after the THEN clause in a CASE WHEN statement.

16.40 row_format

Row format, which specifies the row serialization/deserialization method for a Hive table.

Syntax

ROW FORMAT DELIMITED

[FIELDS TERMINATED BY separator]

[COLLECTION ITEMS TERMINATED BY separator]

[MAP KEYS TERMINATED BY separator] [LINES TERMINATED BY separator]

[NULL DEFINED AS separator]

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

Description

- **separator** refers to the delimiter or substitute character in the syntax. Only supports the CHAR type.
- **FIELDS TERMINATED BY** specifies the field-level delimiter in the table. Only supports the CHAR type.
- **COLLECTION ITEMS TERMINATED BY** specifies the collection-level delimiter. Only supports the CHAR type.
- **MAP KEYS TERMINATED BY** is only used to specify the delimiter between key and value in MAP types. Only supports the CHAR type.
- **LINES TERMINATED BY** specifies the delimiter between rows. Currently only supports **\n**.
- The **NULL DEFINED AS** clause can be used to specify the **NULL** format.
- The SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, property_name=property_value, ...)] You can use the following statement to convert the NULL value to an empty string.

  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe' with serdeproperties('serialization.null.format' = '')

 Copyright © Huawei Technologies Co., Ltd.

16.41 select_statement

Syntax

None.

Description

Basic SELECT statement, i.e., a query statement.

16.42 separator

Syntax

None.

Description

Delimiter. Only supports the CHAR type. Supports user-defined values, such as commas (,), semicolons (;), colons (:), etc.

16.43 serde_name

Syntax

None.

Description

The fully qualified class name of the SerDe class.

Specifies the name of the SerDe. For example, **org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe**.

16.44 sql_containing_cte_name

Syntax

```
WITH cte_name AS (...) SELECT ...
```

Description

A SQL statement that includes a CTE defined by **cte_name**.

16.45 sub_query

Subquery, which is a SELECT statement nested within another query.

 Copyright © Huawei Technologies Co., Ltd.

Syntax

A SELECT statement enclosed in parentheses.

Description

Subquery.

16.46 table_comment

Syntax

None.

Description

A description of the table. Only supports string type. The description length cannot exceed 256 bytes.

16.47 table_name

Syntax

None

Description

Table name, which cannot exceed 128 bytes. The string type and "$" symbol are supported.

The database can be specified using the **db_name.table_name** format.

16.48 table_properties

Syntax

```
('property_name' = 'property_value' [, ...])
```

Description

The property list of a table, expressed in key=value pairs, where **key** is **property_name** and **value** is **property_value**. Each group of key=value in the list is separated by commas (,).

16.49 table_reference

Syntax

None.

 Copyright © Huawei Technologies Co., Ltd.

Description

Table reference, which is the data source referenced in the FROM clause.

This is the name of a table or view. It supports string types or subqueries. When a subquery is used, an alias is recommended.

16.50 view_name

Syntax

None.

Description

View name. Only supports string type. The name length cannot exceed 128 bytes.

16.51 view_properties

Syntax

```
('property_name' = 'property_value' [, ...])
```

Description

The property list of a view, expressed in key=value pairs, where **key** is **property_name** and **value** is **property_value**. Each group of key=value in the list is separated by commas (,).

16.52 when_expression

Syntax

None.

Description

The condition or value after WHEN in a CASE statement.

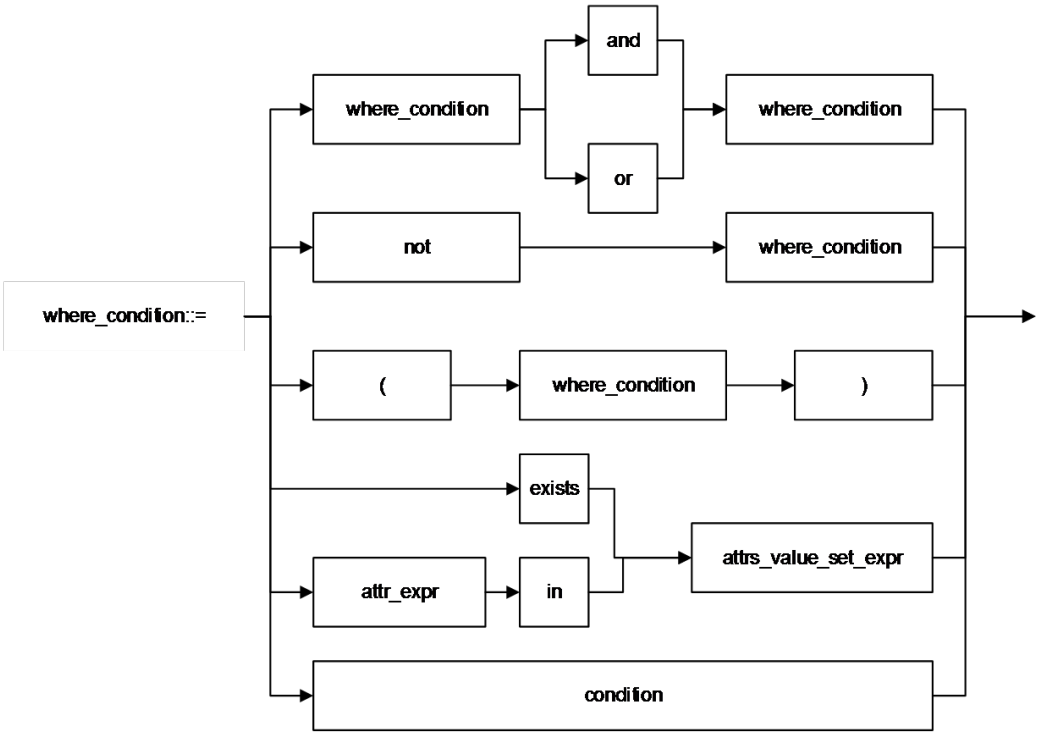
Simple CASE: the value compared with input_expression. Searched CASE: a boolean condition.

16.53 where_condition

WHERE condition, which is used to filter rows.

 Copyright © Huawei Technologies Co., Ltd.

Syntax



Description

Syntax	Description
where_condition	WHERE logical judgment condition.
and	Logical operator: AND.
or	Logical operator: OR.
not	Logical operator: NOT.
(	Start of a sub-logical judgment condition.
)	End of a sub-logical judgment condition.
condition	Logical judgment condition.
exists	Keyword, used to determine if a non-empty set exists. If **exists** is followed by a subquery, the subquery must contain a logical judgment condition.
in	Keyword, used to determine if an attribute is in a set.
attrs_value_set_expr	Attribute value set.
attr_expr	Attribute expression.

 Copyright © Huawei Technologies Co., Ltd.

16.54 window_function

Syntax

None

Description

Window function, which performs calculations over an OVER window.

Examples include **row_number**, **rank**, **lag**, **lead**, etc.

 Copyright © Huawei Technologies Co., Ltd.

17 Operators

17.1 Relational Operators

17.2 Arithmetic Operators

17.3 Logical Operators

17.1 Relational Operators

All basic data types can be compared using relational operators, which return a value of type BOOLEAN.

Relational operators are all binary operators. The two data types being compared must be the same data type or types that can be implicitly converted.

Table 17-1 Relational operators

Operator	Return Type	Description
A = B	BOOLEAN	Returns **TRUE** if A equals B. Otherwise returns **FALSE**.
A == B	BOOLEAN	Returns **TRUE** if A equals B. Otherwise returns **FALSE**. Cannot be used for assignment operations.
A <=> B	BOOLEAN	Returns **TRUE** if A equals B. Otherwise returns **FALSE**. Returns **TRUE** if both A and B are NULL. Returns **FALSE** if one of them is NULL.
A <> B	BOOLEAN	Returns **TRUE** if A is not equal to B. Otherwise returns **FALSE**. Returns NULL if A or B is NULL. Standard SQL syntax.
A != B	BOOLEAN	Same as the <> logical operator. SQL Server syntax.
A < B	BOOLEAN	Returns **TRUE** if A is less than B. Otherwise returns **FALSE**. Returns NULL if A or B is NULL.

 Copyright © Huawei Technologies Co., Ltd.

Operator	Return Type	Description
A <= B	BOOLEAN	Returns **TRUE** if A is less than or equal to B. Otherwise returns **FALSE**. Returns NULL if A or B is NULL.
A > B	BOOLEAN	Returns **TRUE** if A is greater than B. Otherwise returns **FALSE**. Returns NULL if A or B is NULL.
A >= B	BOOLEAN	Returns **TRUE** if A is greater than or equal to B. Otherwise returns **FALSE**. Returns NULL if A or B is NULL.
A BETWEEN B AND C	BOOLEAN	Returns **TRUE** if A is greater than or equal to B and less than or equal to C. Otherwise returns **FALSE**. Returns NULL if any of A, B, or C is NULL.
A NOT BETWEEN B AND C	BOOLEAN	Returns **TRUE** if A is less than B or greater than C. Otherwise returns **FALSE**. Returns NULL if any of A, B, or C is NULL.
A IS NULL	BOOLEAN	Returns **TRUE** if A is NULL. Otherwise returns **FALSE**.
A IS NOT NULL	BOOLEAN	Returns **TRUE** if A is not NULL. Otherwise returns **FALSE**.
A LIKE B	BOOLEAN	Returns **TRUE** if string A matches string B. Otherwise returns **FALSE**. Returns NULL if A or B is NULL.
A NOT LIKE B	BOOLEAN	Returns **TRUE** if string A does not match string B. Otherwise returns **FALSE**. Returns NULL if A or B is NULL.
A RLIKE B	BOOLEAN	Java regular expression matching. Returns **TRUE** if A or any of its substrings matches B. Otherwise returns **FALSE**. Returns NULL if A or B is NULL.
A REGEXP B	BOOLEAN	Same result as A RLIKE B.

Precautions

- Both **=** and **==** are equality comparison operators with the same functionality. **=** is the standard SQL syntax, while **==** is a compatible alternative. Neither is used for assignment.
- **<=>** is a NULL-safe equality operator. Returns **TRUE** when both sides are NULL.
- **<>** is the standard SQL not-equal operator, while **!=** is SQL Server syntax. Both have the same logic.
- In ANSI mode (enabled by default in Spark 4.0), comparisons involving NULL return NULL rather than being silently handled.

 Copyright © Huawei Technologies Co., Ltd.

Example 1: Equality Comparison

```
SELECT 1 = 1;       -- TRUE
SELECT 1 == 2;      -- FALSE
SELECT 1 <=> 1;     -- TRUE
SELECT NULL <=> NULL; -- TRUE
```

Example 2: Inequality Comparison

```
SELECT 1 <> 2;      -- TRUE
SELECT 1 != 1;      -- FALSE
SELECT 1 < 2;       -- TRUE
SELECT 2 >= 2;      -- TRUE
```

Example 3: Range Comparison and NULL Checking

```
SELECT 2 BETWEEN 1 AND 3;    -- TRUE
SELECT 5 NOT BETWEEN 1 AND 3; -- TRUE
SELECT NULL IS NULL;         -- TRUE
SELECT 1 IS NOT NULL;        -- TRUE
```

Example 4: Fuzzy Matching

```
SELECT 'hello' LIKE 'h%';     -- TRUE
SELECT 'hello' NOT LIKE 'a%';  -- TRUE
SELECT 'hello' RLIKE 'h.*o';   -- TRUE
SELECT 'hello' REGEXP 'h.*o';  -- TRUE
```

17.2 Arithmetic Operators

Arithmetic operators include binary operations and unary operations. These operators all return numeric types.

Table 17-2 Arithmetic operators

Operator	Result Type	Description
A + B	All numeric types	Adds A and B. The result data type is related to the operand data types. For example, adding an integer type data to a floating-point type data results in a floating-point type value.
A - B	All numeric types	Subtracts B from A. The result data type is related to the operand data types.
A * B	All numeric types	Multiplies A and B. The result data type is related to the operand data types.
A / B	All numeric types	Divides A by B. The result is a double-precision numeric value.
A % B	All numeric types	Returns the remainder of A divided by B. The result data type is related to the operand data types.

 Copyright © Huawei Technologies Co., Ltd.

Operator	Result Type	Description
A & B	All numeric types	Performs a bitwise AND operation on the binary representations of the two parameters. If a bit is 1 in both expressions, the corresponding bit in the result is 1. Otherwise, the corresponding bit in the result is 0.
A | B	All numeric types	Performs a bitwise OR operation on the binary representations of the two parameters. If a bit is 1 in either expression, the corresponding bit in the result is 1. Otherwise, the corresponding bit in the result is 0.
A ^ B	All numeric types	Performs a bitwise XOR operation on the binary representations of the two parameters. If and only if a bit is 1 in exactly one expression, the corresponding bit in the result is 1. Otherwise, the corresponding bit in the result is 0.
~A	All numeric types	Performs a bitwise NOT operation (inversion) on an expression.

Precautions

- Integer division uses the **div** function. The **/** operator always returns a DOUBLE type.
- In ANSI mode (enabled by default in Spark 4.0), integer division by zero throws an exception rather than returning NULL.
- Bitwise operators (&, |, ^, ~) only apply to integer types.
- The **%** modulo operator is equivalent to the **mod()** function.

Example 1: Basic Arithmetic Operations

```
SELECT 1 + 2;    -- 3
SELECT 5 - 3;    -- 2
SELECT 2 * 3;    -- 6
SELECT 10 / 3;   -- 3.333...
```

Example 2: Modulo Operation

```
SELECT 10 % 3;   -- 1
SELECT 7 % 2;    -- 1
```

Example 3: Bitwise Operations

```
SELECT 5 & 3;    -- 1   (0101 & 0011 = 0001)
SELECT 5 | 3;    -- 7   (0101 | 0011 = 0111)
SELECT 5 ^ 3;    -- 6   (0101 ^ 0011 = 0110)
SELECT ~5;       -- -6  (bitwise NOT)
```

Example 4: Mixed Type Operations

```
SELECT 1 + 2.5;      -- 3.5 (integer + floating-point = floating-point)
SELECT 10 / 3;       -- 3.333... (division returns DOUBLE)
SELECT CAST(10 AS DOUBLE) / 3; -- 3.333...
```

 Copyright © Huawei Technologies Co., Ltd.

17.3 Logical Operators

Commonly used logical operators include **AND**, **OR**, and **NOT**. Their operation results have three possible values: **TRUE**, **FALSE**, and **NULL**, where **NULL** represents unknown. The precedence order is: **NOT** > **AND** > **OR**.

For operation rules, see **Table 17-3**, where A and B in the table represent logical expressions.

Table 17-3 Logical operators

Operator	Result Type	Description
A AND B	BOOLEAN	Returns **TRUE** if both A and B are **TRUE**. Returns **FALSE** if either A or B is **FALSE**. Otherwise, returns **NULL**.
A OR B	BOOLEAN	Returns **TRUE** if either A or B is **TRUE**. Otherwise returns **FALSE**. Returns **NULL** if A or B is **NULL**. Returns **TRUE** if one is **TRUE** and the other is **NULL**.
NOT A	BOOLEAN	Returns **TRUE** if A is **FALSE**. Returns **NULL** if A is **NULL**. Otherwise returns **FALSE**.
! A	BOOLEAN	Same as **NOT A**.
A IN (val1, val2, ...)	BOOLEAN	Returns **TRUE** if A equals any value in (val1, val2, ...). Otherwise returns **FALSE**.
A NOT IN (val1, val2, ...)	BOOLEAN	Returns **TRUE** if A does not equal any value in (val1, val2, ...). Otherwise returns **FALSE**.
EXISTS (subquery)	BOOLEAN	Returns **TRUE** if the subquery returns at least one row. Otherwise returns **FALSE**.
NOT EXISTS (subquery)	BOOLEAN	Returns **TRUE** if the subquery returns no rows. Otherwise returns **FALSE**.

 Copyright © Huawei Technologies Co., Ltd.

18 Datasets

18.1 Overview

18.2 Dataset Syntax

18.3 Dataset Permissions

18.1 Overview

Standard Spark SQL and LakeFormation natively support the management of table structures (database, table, view), but lack a secure, native abstraction for raw file directories. Standard Spark cannot apply SQL-based access control lists (ACLs) to raw directory paths, which forces users to manage complex, cloud-specific IAM roles for different user groups.

A dataset supported by LakeFormation is an entity that can be used to manage raw file directories.

Datasets support a three-level namespace in the format **catalog.schema.dataset**.

DDL commands are supported for datasets (**CREATE DATASET**, **DESCRIBE DATASET**, **DROP DATASET**, **ALTER DATASET**, **SHOW DATASETS**, and **COMMENT ON DATASET**).

Files within a dataset location can be queried.

DCL commands are supported for datasets, including **GRANT**, **REVOKE**, and **SHOW GRANTS**.

18.2 Dataset Syntax

18.2.1 Creating a Dataset

Syntax

```
CREATE [EXTERNAL] DATASET [IF NOT EXISTS] dataset_name
[LOCATION location_path ]
```

 Copyright © Huawei Technologies Co., Ltd.

```
[COMMENT comment ]
[SET (FORMAT <format> | FORMAT <format> CONTENT_TYPE (<content_type>*))]
```

- **EXTERNAL**: Creates an external dataset. An external dataset must be registered to a directory within an external location. When **EXTERNAL** is not specified, the dataset type is **MANAGED**.
- **IF NOT EXISTS**: If specified, no error is thrown when the dataset already exists.
- **dataset_name**: Name of the dataset to be created. A dataset can only be created within a LakeFormation schema. A dataset must not have a naming conflict with other tables or datasets in the same database; otherwise, creation will fail.
- **LOCATION location_path**: **location_path** is a string literal that defines the file system path for creating the specified external dataset. If the specified path does not exist in the underlying file system, a directory will be created at that path. A location must be specified when creating an external dataset. Managed datasets do not support **location_path**. The location of a managed dataset is automatically generated.
- **comment**: Optional description of the dataset.
- **format**: Data format enumeration: **ROW**/**TEXT**/**IMAGE**/**AUDIO**/**VIDEO**/**CUSTOM**.
- **content_type**: When the format is **ROW**, optional values for **content_type** include: **CSV** (comma-separated values file), **JSONL** (JSON object lines file), and **AVRO** (AVRO row-based file).
  - For image file formats, this is set when the format is **IMAGE**. Optional values for **content_type** include: **JPG** (JPG image), **PNG** (PNG image), and **TIFF** (TIFF image).
  - **content_type** has no restrictions. Any string is allowed as content. **format** and **content_type** here serve as annotations to help users filter when listing datasets.

Sample Code

```
CREATE EXTERNAL DATASET my_catalog.my_schema.my_external_dataset LOCATION 'obs://my-bucket/my-location/my-path' COMMENT 'This is my example external dataset on OBS'
```

```
CREATE EXTERNAL DATASET my_catalog.my_schema.my_external_dataset LOCATION 'obs://my-bucket/my-location/my-path' COMMENT 'This is my example external dataset on OBS' SET FORMAT IMAGE CONTENT_TYPE ("JPG", "PNG")
```

18.2.2 Altering a Dataset

```
ALTER DATASET dataset_name
      { SET TAGS clause |
       UNSET TAGS clause |
         SET (
          (FORMAT <format>) |
            (FORMAT <format> CONTENT_TYPE (<content_type>*)))
}
```

- **dataset_name**: Name of the dataset to be altered.
- **SET TAGS ({ tag_name = tag_value } [, ...])**: Applies tags to the dataset. These tags are stored as properties in the dataset.
- **UNSET TAGS (tag_name [, ...])**: Removes tags from the dataset.

 Copyright © Huawei Technologies Co., Ltd.

- **tag_name**: Tag value name. **tag_name** must be unique among all tags associated with the specified dataset.
- **tag_value**: Tag value.

18.2.3 Updating Dataset Format

Syntax

```
ALTER DATASET dataset_name SET FORMAT <format> CONTENT_TYPE(<content_type>*))
```

- **format**: Data format enumeration: **ROW**/**TEXT**/**IMAGE**/**AUDIO**/**VIDEO**/**CUSTOM**.
- **content_type**: When the format is **ROW**, optional values for **content_type** include: **CSV** (comma-separated values file), **JSONL** (JSON object lines file), and **AVRO** (AVRO row-based file). For image file formats, this is set when the format is **IMAGE**. Optional values for **content_type** include: **JPG** (JPG image), **PNG** (PNG image), and **TIFF** (TIFF image). **content_type** has no restrictions. Any string is allowed as content.

Sample Code

```
ALTER DATASET my_dataset SET TAGS ('tag1' = 'val1', 'tag2' = 'val2', 'tag3' = 'val3');
ALTER DATASET my_dataset UNSET TAGS ('tag1', 'tag2', 'tag3');
```

```
ALTER DATASET dataset_name SET FORMAT IMAGE
ALTER DATASET dataset_name SET FORMAT IMAGE CONTENT_TYPE ("JPG", "PNG")
```

18.2.4 Setting or Updating a Dataset Comment

Syntax

```
COMMENT ON DATASET dataset_name IS comment
```

- **dataset_name**: Name of the dataset to be commented on.
- **comment**: A string literal or **NULL**. If **NULL** is specified, any existing comment is removed.

Sample Code

```
COMMENT ON DATASET my_dataset IS 'Huge dataset';
COMMENT ON DATASET my_dataset IS NULL;
```

18.2.5 Soft Deleting a Dataset

Syntax

```
DROP DATASET [IF EXISTS] dataset_name [PURGE]
```

- **PURGE**: The dataset data is deleted.
- **IF EXISTS**: If specified, no error is thrown when the dataset does not exist.
- **dataset_name**: Name of the dataset to be dropped. If **EXISTS** is not specified and the dataset is not found, Spark will throw an error.

For managed datasets:

PURGE specified: Updates the metadata entry and moves the underlying data files to the recycle bin.

 Copyright © Huawei Technologies Co., Ltd.

PURGE not specified: Only updates the metadata entry.

For external datasets:

Only the metadata entry is updated. The underlying files are never deleted. Therefore, specifying **PURGE** or not has no effect on external datasets.

A soft-deleted dataset can be restored or permanently deleted.

If a dataset exists in a database, dropping the database will fail. When dropping a database, either the database must be empty (containing no datasets), or the **cascade** parameter must be passed to recursively delete all entities within the database.

Sample Code

```
DROP DATASET IF EXISTS my_dataset
DROP DATASET IF EXISTS my_dataset PURGE
```

18.2.6 Listing All Datasets

Syntax

```
SHOW DATASETS [ { FROM | IN } schema_name ] [ [ LIKE ] regex_pattern ] [FORMAT format]
```

- **schema_name**: The schema for which to list datasets.
- **regex_pattern**: A string literal containing a regular expression pattern used to filter the results of the statement.
- **format**: The format for filtering datasets.

Returns **(database: string, datasetName: string, description: string)**.

Sample Code

```
SHOW DATASETS IN machine_learning LIKE 'a*'
SHOW DATASETS IN machine_learning LIKE 'a*' FORMAT IMAGE
```

18.2.7 Displaying Dataset Details

Syntax

```
DESCRIBE DATASET dataset_name
```

dataset_name: Name of an existing dataset in LakeFormation. If the name does not exist, an exception is thrown.

Returns a single row containing:

- **name**, **catalog**, **database**, **owner**, **storageLocation**, **datasetType** (MANAGED/EXTERNAL), **comment**, **format**, **contentTypes**, **tags**
- **name** STRING NOT NULL: Name of the dataset, used to identify the dataset object.
- **catalog** STRING NOT NULL: Name of the catalog containing the dataset.
- **database** STRING NOT NULL: Name of the schema containing the dataset.
- **owner** STRING NOT NULL: Entity that owns the dataset.
- **storageLocation** STRING: Path of the external location for an external dataset. For managed datasets, this value is **NULL**.

 Copyright © Huawei Technologies Co., Ltd.

- **datasetType** STRING NOT NULL: Dataset type. The value can be **EXTERNAL** or **MANAGED**.
- **comment** STRING: Comment for the dataset.
- **format** STRING: File format of the dataset.
- **contentTypes** STRING: Content types of the dataset.
- **tags** STRING: Tags or properties set on the dataset.

Sample Code

```
DESCRIBE DATASET my_external_dataset
```

18.2.8 Reading Files of a Specified Format in a Dataset

Syntax

```
SELECT * FROM <fileformat>.`/Datasets/<catalog>/<schema>/<dataset>/<relative_path_to_file>`
```

- **fileformat**: indicates the file format being read.
- **Datasets** is a prefix used to identify a dataset path.
- **CATALOG**, **SCHEMA**, and **DATASET** are used to identify the dataset to which this relative file path belongs. The location is obtained from the dataset.
- The remaining part (**relative_path_to_file.parquet**) is the relative file path resolved relative to the location obtained from the dataset.

Sample Code

```
SELECT * FROM parquet.`/Datasets/cat1/sch1/dataset1/file.parquet`
```

18.3 Dataset Permissions

18.3.1 Overview

The following are the LakeFormation permissions required for end users to create or manage datasets.

Permission	Enum Value	Grant Level	Use Case	Corresponding Access Label	Corresponding Spark SQL
ALL	PermissionType.ALL	Dataset	All permissions (including READ_DATASET + WRITE_DATASET)	W	ALTER DATASET, DROP DATASET, DESCRIBE DATASET, COMMENT ON DATASET

 Copyright © Huawei Technologies Co., Ltd.

Permission	Enum Value	Grant Level	Use Case	Corresponding Access Label	Corresponding Spark SQL
READ_DATASET	PermissionType.READ_DATASET	Dataset	View Dataset Metadata	R	DESCRIBE DATASET
WRITE_DATASET	PermissionType.WRITE_DATASET	Dataset	Modify Dataset, Delete Dataset	W	DROP DATASET, COMMENT ON DATASET
CREATE_DATASET	PermissionType.CREATE_DATASET	Database	Create a Dataset under Database	—	CREATE DATASET
LIST_DATASET	PermissionType.LIST_DATASET	Database	List Datasets under Database	—	SHOW DATASETS

Listing all datasets:

If a database does not have the **LIST_DATASET** permission, datasets with the **READ_DATASET** permission are returned. If a database has the **LIST_DATASET** permission, all datasets are returned.

Query support:

When a user reads files from a specific dataset in a specified format, the user must have the **READ_DATASET** permission on that dataset. The user should also have URI permissions on the dataset location to read files within the dataset location.

18.3.2 Granting User Access

Syntax

```
GRANT <privilege>* ON <resource>* TO <principal>*
```

- **privilege**: The permission to grant to the principal on the resource.
- **resource**: The entity on which the permission is granted.
- **principal**: The user or role to which the permission is granted.

Sample Code

```
Read/Write/All (read + write) - Dataset-level authorization:
GRANT READ_DATASET ON databases.demo_db1.datasets.demo_ds1 TO USER aid_developer03;
GRANT WRITE_DATASET ON databases.demo_db1.datasets.demo_ds1 TO USER aid_developer03;
GRANT ALL ON databases.demo_db1.datasets.demo_ds1 TO USER aid_developer03;
GRANT READ_DATASET ON databases.demo_db1.datasets.demo_ds1 TO ROLE nit_role3;
```

 Copyright © Huawei Technologies Co., Ltd.

```
GRANT WRITE_DATASET ON databases.demo_db1.datasets.demo_ds1 TO ROLE nit_role3;
GRANT ALL ON databases.demo_db1.datasets.demo_ds1 TO ROLE nit_role3;

Create/List - Database-level authorization:
GRANT CREATE_DATASET ON databases.demo_db1 TO USER aid_developer03;
GRANT LIST_DATASET ON databases.demo_db1 TO USER aid_developer03;
GRANT CREATE_DATASET ON databases.demo_db1 TO ROLE nit_role3;
GRANT LIST_DATASET ON databases.demo_db1 TO ROLE nit_role3
```

18.3.3 Revoking User Access

Syntax

```
REVOKE <privilege>* ON <resource>* FROM <principal>*
```

- **privilege**: The permission to revoke from the principal on the resource.
- **resource**: The entity from which the permission is revoked.
- **principal**: The user or role from which the permission is revoked.

Sample Code

```
Read/Write/All (read + write) - Dataset-level authorization:
REVOKE READ_DATASET ON databases.demo_db1.datasets.demo_ds1 FROM USER normal_user1;
REVOKE WRITE_DATASET ON databases.demo_db1.datasets.demo_ds1 FROM USER normal_user1;
REVOKE ALL ON databases.demo_db1.datasets.demo_ds1 FROM USER normal_user1;
REVOKE READ_DATASET ON databases.demo_db1.datasets.demo_ds1 FROM ROLE nit_role3;
REVOKE WRITE_DATASET ON databases.demo_db1.datasets.demo_ds1 FROM ROLE nit_role3;
REVOKE ALL ON databases.demo_db1.datasets.demo_ds1 FROM ROLE nit_role3;

Create/List - Database-level authorization:
REVOKE CREATE_DATASET ON databases.demo_db1 FROM USER normal_user1;
REVOKE LIST_DATASET ON databases.demo_db1 FROM USER normal_user1;
REVOKE CREATE_DATASET ON databases.demo_db1 FROM ROLE nit_role3;
REVOKE LIST_DATASET ON databases.demo_db1 FROM ROLE nit_role3;
```

18.3.4 Showing All Permissions

Syntax

```
SHOW GRANT <principal>  ON <resource>
```

- **principal**: Lists permissions for a specific user or role.
- **resource**: Lists permissions for a specific entity.

Sample Code

```
SHOW GRANT USER aid_developer03 ON databases.demo_db1.datasets.demo_ds1;
SHOW GRANT ROLE nit_role3 ON databases.demo_db1.datasets.demo_ds1;
SHOW GRANT USER aid_developer03 ON databases.demo_db1;
SHOW GRANT ROLE nit_role3 ON databases.demo_db1;
```

 Copyright © Huawei Technologies Co., Ltd.