

**Data Lake Insight**

# **Spark SQL Syntax**

**Issue**            01  
**Date**             2025-08-06



**Copyright © Huawei Technologies Co., Ltd. 2025. 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**



HUAWEI 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](mailto: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

---

<b>1 Common Configuration Items.....</b>	<b>1</b>
<b>2 Spark SQL Syntax.....</b>	<b>7</b>
<b>3 Spark Open Source Commands.....</b>	<b>10</b>
<b>4 Databases.....</b>	<b>14</b>
4.1 Creating a Database.....	14
4.2 Deleting a Database.....	15
4.3 Viewing a Specified Database.....	16
4.4 Viewing All Databases.....	17
<b>5 Tables.....</b>	<b>18</b>
5.1 Creating an OBS Table.....	18
5.1.1 Creating an OBS Table Using the DataSource Syntax.....	18
5.1.2 Creating an OBS Table Using the Hive Syntax.....	26
5.2 Creating a DLI Table.....	33
5.2.1 Creating a DLI Table Using the DataSource Syntax.....	34
5.2.2 Creating a DLI Table Using the Hive Syntax.....	38
5.3 Deleting a Table.....	43
5.4 Viewing a Table.....	44
5.4.1 Viewing All Tables.....	44
5.4.2 Viewing Table Creation Statements.....	45
5.4.3 Viewing Table Properties.....	47
5.4.4 Viewing All Columns in a Specified Table.....	47
5.4.5 Viewing All Partitions in a Specified Table.....	48
5.4.6 Viewing Table Statistics.....	49
5.5 Modifying a Table.....	50
5.5.1 Adding a Column.....	50
5.5.2 Modifying Column Comments.....	51
5.5.3 Enabling or Disabling Data Multi-Versioning (Deprecated, Not Recommended).....	52
5.6 Partition-related Syntax.....	53
5.6.1 Adding Partition Data (Only OBS Tables Supported).....	54
5.6.2 Renaming a Partition (Only OBS Tables Supported).....	56
5.6.3 Deleting a Partition.....	57
5.6.4 Deleting Partitions by Specifying Filter Criteria (Only Supported on OBS Tables).....	60

- 5.6.5 Altering the Partition Location of a Table (Only OBS Tables Supported)..... 65
- 5.6.6 Updating Partitioned Table Data (Only OBS Tables Supported)..... 66
- 5.6.7 Updating Table Metadata with REFRESH TABLE..... 67
- 5.7 Backing Up and Restoring Multi-Versioning Data (Deprecated, Not Recommended)..... 68
  - 5.7.1 Setting the Retention Period of Multi-Versioning Backup Data (Deprecated, Not Recommended)... 68
  - 5.7.2 Viewing Multi-Versioning Backup Data (Deprecated, Not Recommended)..... 69
  - 5.7.3 Restoring Multi-Versioning Backup Data (Deprecated, Not Recommended)..... 70
  - 5.7.4 Configuring the Recycle Bin for Expired Multi-Versioning Data (Deprecated, Not Recommended)... 71
  - 5.7.5 Clearing Multi-Versioning Data (Deprecated, Not Recommended)..... 73
- 5.8 Table Lifecycle Management..... 74
  - 5.8.1 Specifying the Lifecycle of a Table When Creating the Table..... 74
  - 5.8.2 Modifying the Lifecycle of a Table..... 78
  - 5.8.3 Disabling or Restoring the Lifecycle of a Table..... 79
- 6 Data..... 82**
  - 6.1 Importing Data..... 82
  - 6.2 Inserting Data..... 86
  - 6.3 Reusing Results of Subqueries..... 90
  - 6.4 Clearing Data..... 91
- 7 Exporting Query Results..... 92**
- 8 Datasource Connections..... 94**
  - 8.1 Creating a Datasource Connection with an HBase Table..... 94
    - 8.1.1 Creating a DLI Table and Associating It with HBase..... 94
    - 8.1.2 Inserting Data to an HBase Table..... 97
    - 8.1.3 Querying an HBase Table..... 98
  - 8.2 Creating a Datasource Connection with an OpenTSDB Table..... 100
    - 8.2.1 Creating a DLI Table and Associating It with OpenTSDB..... 100
    - 8.2.2 Inserting Data to the OpenTSDB Table..... 101
    - 8.2.3 Querying an OpenTSDB Table..... 102
  - 8.3 Creating a Datasource Connection with a DWS Table..... 103
    - 8.3.1 Creating a DLI Table and Associating It with DWS..... 103
    - 8.3.2 Inserting Data to the DWS Table..... 106
    - 8.3.3 Querying the DWS Table..... 107
  - 8.4 Creating a Datasource Connection with an RDS Table..... 107
    - 8.4.1 Creating a DLI Table and Associating It with RDS..... 107
    - 8.4.2 Inserting Data to the RDS Table..... 111
    - 8.4.3 Querying the RDS Table..... 112
  - 8.5 Creating a Datasource Connection with a CSS Table..... 112
    - 8.5.1 Creating a DLI Table and Associating It with CSS..... 113
    - 8.5.2 Inserting Data to the CSS Table..... 115
    - 8.5.3 Querying the CSS Table..... 116
  - 8.6 Creating a Datasource Connection with a DCS Table..... 117

8.6.1 Creating a DLI Table and Associating It with DCS.....	117
8.6.2 Inserting Data to a DCS Table.....	119
8.6.3 Querying the DCS Table.....	121
8.7 Creating a Datasource Connection with a DDS Table.....	122
8.7.1 Creating a DLI Table and Associating It with DDS.....	122
8.7.2 Inserting Data to the DDS Table.....	123
8.7.3 Querying the DDS Table.....	125
8.8 Creating a Datasource Connection with an Oracle Table.....	125
8.8.1 Creating a DLI Table and Associating It with Oracle.....	125
8.8.2 Inserting Data to an Oracle Table.....	127
8.8.3 Querying an Oracle Table.....	128
<b>9 Views.....</b>	<b>129</b>
9.1 Creating a View.....	129
9.2 Deleting a View.....	130
<b>10 Viewing the Execution Plan.....</b>	<b>131</b>
<b>11 Data Permissions.....</b>	<b>132</b>
11.1 Data Permissions List.....	132
11.2 Creating a Role.....	135
11.3 Deleting a Role.....	136
11.4 Binding a Role.....	136
11.5 Unbinding a Role.....	137
11.6 Displaying a Role.....	137
11.7 Granting a Permission.....	138
11.8 Revoking a Permission.....	139
11.9 Displaying the Granted Permissions.....	140
11.10 Displaying the Binding Relationship Between All Roles and Users.....	141
<b>12 Data Types.....</b>	<b>142</b>
12.1 Overview.....	142
12.2 Primitive Data Types.....	142
12.3 Complex Data Types.....	146
<b>13 User-Defined Functions.....</b>	<b>150</b>
13.1 Creating a Function.....	150
13.2 Deleting a Function.....	158
13.3 Displaying Function Details.....	158
13.4 Displaying All Functions.....	159
<b>14 Built-In Functions.....</b>	<b>161</b>
14.1 Date Functions.....	161
14.1.1 Overview.....	161
14.1.2 add_months.....	165
14.1.3 current_date.....	166

14.1.4	current_timestamp	166
14.1.5	date_add	167
14.1.6	dateadd	168
14.1.7	date_sub	170
14.1.8	date_format	171
14.1.9	datediff	173
14.1.10	datediff1	174
14.1.11	datepart	176
14.1.12	datetrunc	177
14.1.13	day/dayofmonth	179
14.1.14	from_unixtime	179
14.1.15	from_utc_timestamp	180
14.1.16	getdate	181
14.1.17	hour	182
14.1.18	isdate	183
14.1.19	last_day	184
14.1.20	lastday	185
14.1.21	minute	186
14.1.22	month	187
14.1.23	months_between	188
14.1.24	next_day	190
14.1.25	quarter	191
14.1.26	second	192
14.1.27	to_char	193
14.1.28	to_date	194
14.1.29	to_date1	195
14.1.30	to_utc_timestamp	196
14.1.31	trunc	197
14.1.32	unix_timestamp	199
14.1.33	weekday	200
14.1.34	weekofyear	201
14.1.35	year	202
14.2	String Functions	203
14.2.1	Overview	203
14.2.2	ascii	209
14.2.3	concat	210
14.2.4	concat_ws	211
14.2.5	char_matchcount	213
14.2.6	encode	213
14.2.7	find_in_set	214
14.2.8	get_json_object	215
14.2.9	instr	217

14.2.10 instr1.....	218
14.2.11 initcap.....	220
14.2.12 keyvalue.....	220
14.2.13 length.....	221
14.2.14 lengthb.....	222
14.2.15 levenshtein.....	223
14.2.16 locate.....	224
14.2.17 lower/lcase.....	225
14.2.18 lpad.....	226
14.2.19 ltrim.....	227
14.2.20 parse_url.....	228
14.2.21 printf.....	229
14.2.22 regexp_count.....	230
14.2.23 regexp_extract.....	231
14.2.24 replace.....	232
14.2.25 regexp_replace.....	233
14.2.26 regexp_replace1.....	235
14.2.27 regexp_instr.....	236
14.2.28 regexp_substr.....	238
14.2.29 repeat.....	239
14.2.30 reverse.....	240
14.2.31 rpad.....	240
14.2.32 rtrim.....	241
14.2.33 soundex.....	243
14.2.34 space.....	243
14.2.35 substr/substring.....	244
14.2.36 substring_index.....	245
14.2.37 split_part.....	246
14.2.38 translate.....	247
14.2.39 trim.....	248
14.2.40 upper/ucase.....	250
14.3 Mathematical Functions.....	250
14.3.1 Overview.....	250
14.3.2 abs.....	254
14.3.3 acos.....	255
14.3.4 asin.....	256
14.3.5 atan.....	257
14.3.6 bin.....	257
14.3.7 bround.....	258
14.3.8 cbrt.....	259
14.3.9 ceil.....	260
14.3.10 conv.....	261

14.3.11 cos.....	263
14.3.12 cot1.....	263
14.3.13 degrees.....	264
14.3.14 e.....	265
14.3.15 exp.....	265
14.3.16 factorial.....	266
14.3.17 floor.....	267
14.3.18 greatest.....	268
14.3.19 hex.....	269
14.3.20 least.....	270
14.3.21 ln.....	271
14.3.22 log.....	272
14.3.23 log10.....	273
14.3.24 log2.....	273
14.3.25 median.....	274
14.3.26 negative.....	275
14.3.27 percentile.....	276
14.3.28 percentile_approx.....	277
14.3.29 pi.....	278
14.3.30 pmod.....	278
14.3.31 positive.....	279
14.3.32 pow.....	280
14.3.33 radians.....	281
14.3.34 rand.....	282
14.3.35 round.....	283
14.3.36 shiftleft.....	284
14.3.37 shiftright.....	285
14.3.38 shiftrightunsigned.....	286
14.3.39 sign.....	287
14.3.40 sin.....	288
14.3.41 sqrt.....	289
14.3.42 tan.....	290
14.4 Aggregate Functions.....	291
14.4.1 Overview.....	291
14.4.2 avg.....	292
14.4.3 corr.....	293
14.4.4 count.....	294
14.4.5 covar_pop.....	295
14.4.6 covar_samp.....	296
14.4.7 max.....	297
14.4.8 min.....	298
14.4.9 percentile.....	298

14.4.10 percentile_approx.....	299
14.4.11 stddev_pop.....	301
14.4.12 stddev_samp.....	301
14.4.13 sum.....	302
14.4.14 variance/var_pop.....	303
14.4.15 var_samp.....	304
14.5 Window Functions.....	305
14.5.1 Overview.....	305
14.5.2 cume_dist.....	307
14.5.3 first_value.....	309
14.5.4 last_value.....	311
14.5.5 lag.....	313
14.5.6 lead.....	315
14.5.7 percent_rank.....	317
14.5.8 rank.....	318
14.5.9 row_number.....	320
14.6 Other Functions.....	321
14.6.1 Overview.....	322
14.6.2 decode1.....	323
14.6.3 javahash.....	324
14.6.4 max_pt.....	325
14.6.5 ordinal.....	325
14.6.6 trans_array.....	326
14.6.7 trunc_numeric.....	328
14.6.8 url_decode.....	329
14.6.9 url_encode.....	330
<b>15 SELECT.....</b>	<b>331</b>
15.1 Basic Statements.....	331
15.2 Sort.....	332
15.2.1 ORDER BY.....	333
15.2.2 SORT BY.....	333
15.2.3 CLUSTER BY.....	334
15.2.4 DISTRIBUTE BY.....	334
15.3 Grouping.....	335
15.3.1 Column-Based GROUP BY.....	335
15.3.2 Expression-Based GROUP BY.....	336
15.3.3 Using HAVING in GROUP BY.....	336
15.3.4 ROLLUP.....	337
15.3.5 GROUPING SETS.....	338
15.4 Joins.....	339
15.4.1 INNER JOIN.....	339
15.4.2 LEFT OUTER JOIN.....	339

15.4.3 RIGHT OUTER JOIN.....	340
15.4.4 FULL OUTER JOIN.....	341
15.4.5 IMPLICIT JOIN.....	341
15.4.6 Cartesian JOIN.....	342
15.4.7 LEFT SEMI JOIN.....	342
15.4.8 NON-EQUIJOIN.....	343
15.5 Clauses.....	343
15.5.1 FROM.....	344
15.5.2 OVER.....	344
15.5.3 WHERE.....	346
15.5.4 HAVING.....	347
15.5.5 Multi-Layer Nested Subquery.....	347
15.6 Alias.....	348
15.6.1 Table Alias.....	348
15.6.2 Column Alias.....	349
15.7 Set Operations.....	349
15.7.1 UNION.....	349
15.7.2 INTERSECT.....	350
15.7.3 EXCEPT.....	350
15.8 WITH...AS.....	351
15.9 CASE...WHEN.....	351
15.9.1 Basic CASE Statement.....	352
15.9.2 CASE Query Statement.....	352
<b>16 Identifiers.....</b>	<b>354</b>
16.1 aggregate_func.....	354
16.2 alias.....	354
16.3 attr_expr.....	355
16.4 attr_expr_list.....	356
16.5 attrs_value_set_expr.....	357
16.6 boolean_expression.....	357
16.7 class_name.....	357
16.8 col.....	358
16.9 col_comment.....	358
16.10 col_name.....	358
16.11 col_name_list.....	358
16.12 condition.....	359
16.13 condition_list.....	361
16.14 cte_name.....	361
16.15 data_type.....	362
16.16 db_comment.....	362
16.17 db_name.....	362
16.18 else_result_expression.....	362

16.19 file_format.....	362
16.20 file_path.....	363
16.21 function_name.....	363
16.22 groupby_expression.....	363
16.23 having_condition.....	364
16.24 hdfs_path.....	365
16.25 input_expression.....	365
16.26 input_format_classname.....	365
16.27 jar_path.....	366
16.28 join_condition.....	366
16.29 non_equi_join_condition.....	367
16.30 number.....	367
16.31 num_buckets.....	368
16.32 output_format_classname.....	368
16.33 partition_col_name.....	368
16.34 partition_col_value.....	368
16.35 partition_specs.....	369
16.36 property_name.....	369
16.37 property_value.....	369
16.38 regex_expression.....	369
16.39 result_expression.....	370
16.40 row_format.....	370
16.41 select_statement.....	371
16.42 separator.....	371
16.43 serde_name.....	371
16.44 sql_containing_cte_name.....	371
16.45 sub_query.....	371
16.46 table_comment.....	372
16.47 table_name.....	372
16.48 table_properties.....	372
16.49 table_reference.....	372
16.50 view_name.....	373
16.51 view_properties.....	373
16.52 when_expression.....	373
16.53 where_condition.....	374
16.54 window_function.....	375
<b>17 Operators.....</b>	<b>376</b>
17.1 Relational Operators.....	376
17.2 Arithmetic Operators.....	377
17.3 Logical Operators.....	378

# 1 Common Configuration Items

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

**Table 1-1** Common configuration items

Item	Default Value	Description
spark.sql.files.maxRecordsPerFile	0	Maximum number of records to be written into a single file. If the value is zero or negative, there is no limit.
spark.sql.shuffle.partitions	200	Default number of partitions used to filter data for join or aggregation.
spark.sql.dynamicPartitionOverwrite.enabled	false	Whether DLI overwrites the partitions where data will be written into during runtime. If you set this parameter to <b>false</b> , all partitions that meet the specified condition will be deleted before data overwrite starts. For example, if you set <b>false</b> and use INSERT OVERWRITE to write partition 2021-02 to a partitioned table that has the 2021-01 partition, this partition will be deleted.  If you set this parameter to <b>true</b> , DLI does not delete partitions before overwrite starts.
spark.sql.files.maxPartitionBytes	134217728	Maximum number of bytes to be packed into a single partition when a file is read.
spark.sql.badRecordsPath	-	Path of bad records.

Item	Default Value	Description
spark.sql.legacy.correlated.scalar.query.enabled	false	<ul style="list-style-type: none"><li>• <b>If set to true:</b><ul style="list-style-type: none"><li>– When there is no duplicate data in a subquery, executing a correlated subquery does not require deduplication from the subquery's result.</li><li>– If there is duplicate data in a subquery, executing a correlated subquery will result in an error. To resolve this, the subquery's result must be deduplicated using functions such as <b>max()</b> or <b>min()</b>.</li></ul></li><li>• <b>If set to false:</b><p>Regardless of whether there is duplicate data in a subquery, executing a correlated subquery requires deduplicating the subquery's result using functions such as <b>max()</b> or <b>min()</b>. Otherwise, an error will occur.</p></li></ul>

Item	Default Value	Description
spark.sql.keep.distinct.expandThreshold	-	<ul style="list-style-type: none"> <li>● <b>Parameter description:</b> When running queries with multidimensional analysis that include the <b>count(distinct)</b> function using the cube structure in Spark, the typical execution plan involves using the <b>expand</b> operator. However, this operation can cause query inflation. To avoid this issue, you are advised to configure the following settings: <ul style="list-style-type: none"> <li>- <b>spark.sql.keep.distinct.expandThreshold:</b> Default value: <b>-1</b>, indicating that Spark's default <b>expand</b> operator is used. Setting the parameter to a specific value, such as <b>512</b>, defines the threshold for query inflation. If the threshold is exceeded, the <b>count(distinct)</b> function will use the <b>distinct</b> aggregation operator to execute the query instead of the <b>expand</b> operator.</li> <li>- <b>spark.sql.distinct.aggregator.enabled:</b> whether to forcibly use the <b>distinct</b> aggregation operator. If set to <b>true</b>, <b>spark.sql.keep.distinct.expandThreshold</b> is not used.</li> </ul> </li> <li>● <b>Use case:</b> Queries with multidimensional analysis that use the cube structure and may include multiple <b>count(distinct)</b> functions, as well as the <b>cube</b> or <b>rollup</b> operator.</li> <li>● <b>Example of a typical use case:</b> SELECT a1, a2, count(distinct b), count(distinct c) FROM test_distinct group by a1, a2 with cube</li> </ul>
spark.sql.distinct.aggregator.enabled	false	

Item	Default Value	Description
dli.jobs.sql.resubmit.enable	null	<p>You can control whether Spark SQL jobs are resubmitted in the event of driver failure or queue restart by setting this parameter.</p> <ul style="list-style-type: none"> <li>• <b>false</b>: Disables job retry, all types of commands will not be resubmitted, and the job will be marked as failed once the driver fails.</li> <li>• <b>true</b>: Enables job retry, meaning all types of jobs will be resubmitted in the event of driver failure.</li> </ul> <p><b>CAUTION</b>                      If set to <b>true</b>, there may be data consistency issues when performing idempotent operations such as <b>INSERT</b> (for example, <b>insert into</b>, <b>load data</b>, <b>update</b>). This means that if the driver fails and the job is retried, the data that was already inserted before the driver failure may be overwritten again.</p>

Item	Default Value	Description
spark.sql.dli.job.share Level	Queue	<p>This configuration item is used to set the isolation level of SQL statements. Different isolation levels (job, user, project, queue) determine whether SQL jobs are executed by independent Spark Drivers and Executors or share existing ones.</p> <ul style="list-style-type: none"> <li>• <b>job:</b> <ul style="list-style-type: none"> <li>- Each SQL job will independently start a Spark Driver and a set of Executors for execution.</li> <li>- This is suitable for jobs that require complete isolation, ensuring that each job's execution environment is entirely independent.</li> </ul> </li> <li>• <b>user:</b> <ul style="list-style-type: none"> <li>- If a Spark Driver started by this user already exists and can continue submitting tasks, the new SQL job will be submitted to this existing Driver for execution.</li> <li>- If there is no existing Driver or the current Driver cannot continue submitting tasks, a new Spark Driver will be started for this user.</li> <li>- This is suitable for scenarios where multiple jobs from the same user need to share resources.</li> </ul> </li> <li>• <b>project:</b> <ul style="list-style-type: none"> <li>- If a Spark Driver started by this project already exists and can continue submitting tasks, the new SQL job will be submitted to this existing Driver for execution.</li> <li>- If there is no existing Driver or the current Driver cannot continue submitting tasks, a new Spark Driver will be started for this project.</li> <li>- This is suitable for scenarios where multiple jobs within the same project need to share resources.</li> </ul> </li> <li>• <b>queue:</b> <ul style="list-style-type: none"> <li>- If a Spark Driver started by this queue already exists and can continue submitting tasks, the new SQL job will be submitted to this existing Driver for execution.</li> </ul> </li> </ul>

Item	Default Value	Description
		<ul style="list-style-type: none"> <li>- If there is no existing Driver or the current Driver cannot continue submitting tasks, a new Spark Driver will be started for this queue.</li> <li>- This is suitable for scenarios where resources are managed by queues, allowing for more granular control over resource allocation.</li> </ul> <p><b>NOTE</b>                      The maximum number of Spark Drivers that can be started (maximum Spark Driver instances) and the maximum number of concurrent SQL queries that can be executed by each Spark Driver (maximum concurrency per Spark Driver instance) can be configured in the queue properties.</p> <p><a href="#">Setting Queue Properties</a></p>

# 2 Spark SQL Syntax

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

**Table 2-1** SQL syntax of batch jobs

Classification	Reference
Database-related Syntax	<a href="#">Creating a Database</a>
	<a href="#">Deleting a Database</a>
	<a href="#">Checking a Specified Database</a>
	<a href="#">Checking All Databases</a>
Syntax for Creating an OBS Table	<a href="#">Creating an OBS Table Using the Datasource Syntax</a>
	<a href="#">Creating an OBS Table Using the Hive Syntax</a>
Syntax for Creating a DLI Table	<a href="#">Creating a DLI Table Using the Datasource Syntax</a>
	<a href="#">Creating a DLI Table Using the Hive Syntax</a>
Syntax for Deleting a Table	<a href="#">Deleting a Table</a>
Syntax for Checking a Table	<a href="#">Checking All Tables</a>
	<a href="#">Checking Table Creation Statements</a>
	<a href="#">Checking Table Properties</a>
	<a href="#">Checking All Columns in a Specified Table</a>
	<a href="#">Checking All Partitions in a Specified Table</a>
	<a href="#">Checking Table Statistics</a>
Syntax for Modifying a Table	<a href="#">Adding a Column</a>

<b>Classification</b>	<b>Reference</b>
Syntax for Partitioning a Table	<a href="#">Adding a Partition (Only OBS Tables Supported)</a>
	<a href="#">Renaming a Partition</a>
	<a href="#">Deleting a Partition</a>
	<a href="#">Altering the Partition Location of a Table (Only OBS Tables Supported)</a>
	<a href="#">Updating Partitioned Table Data (Only OBS Tables Supported)</a>
Syntax for Importing Data	<a href="#">Importing Data</a>
Syntax for Inserting Data	<a href="#">Inserting Data</a>
Syntax for Clearing Data	<a href="#">Clearing Data</a>
Syntax for Exporting Query Results	<a href="#">Exporting Query Result</a>
Syntax for Datasource Connection to an HBase Table	<a href="#">Creating a Table and Associating It with HBase</a>
	<a href="#">Inserting Data to an HBase Table</a>
	<a href="#">Querying an HBase Table</a>
Syntax for Datasource Connection to an OpenTSDB Table	<a href="#">Creating a Table and Associating It with OpenTSDB</a>
	<a href="#">Inserting Data to an OpenTSDB Table</a>
	<a href="#">Querying an OpenTSDB Table</a>
Syntax for Datasource Connection to a GaussDB(DWS) Table	<a href="#">Creating a Table and Associating It with GaussDB(DWS)</a>
	<a href="#">Inserting Data to a GaussDB(DWS) Table</a>
	<a href="#">Querying a GaussDB(DWS) Table</a>
Syntax for Datasource Connection to an RDS Table	<a href="#">Creating a Table and Associating It with RDS</a>
	<a href="#">Inserting Data to an RDS Table</a>
	<a href="#">Querying an RDS Table</a>
Syntax for Datasource Connection to a CSS Table	<a href="#">Creating a Table and Associating It with CSS</a>
	<a href="#">Inserting Data to a CSS Table</a>
	<a href="#">Querying a CSS Table</a>
Syntax for Datasource Connection to a DCS Table	<a href="#">Creating a Table and Associating It with DCS</a>
	<a href="#">Inserting Data to a DCS Table</a>

Classification	Reference
	<a href="#">Querying a DCS Table</a>
Syntax for Datasource Connection to a DDS Table	<a href="#">Creating a Table and Associating It with DDS</a>
	<a href="#">Inserting Data to a DDS Table</a>
	<a href="#">Querying a DDS Table</a>
View-related Syntax	<a href="#">Creating a View</a>
	<a href="#">Deleting a View</a>
Syntax for Checking the Execution Plan	<a href="#">Checking the Execution Plan</a>
Syntax Related to Data Permissions	<a href="#">Creating a Role</a>
	<a href="#">Deleting a Role</a>
	<a href="#">Binding a Role</a>
	<a href="#">Unbinding a Role</a>
	<a href="#">Displaying a Role</a>
	<a href="#">Granting a Permission</a>
	<a href="#">Revoking a Permission</a>
	<a href="#">Displaying the Granted Permissions</a>
	<a href="#">Displaying the Binding Relationship Between All Roles and Users</a>
UDF-related Syntax	<a href="#">Creating a Function</a>
	<a href="#">Deleting a Function</a>
	<a href="#">Displaying Function Details</a>
	<a href="#">Displaying All Functions</a>
Data Multi-Versioning-related Syntax (Deprecated, Not Recommended)	<a href="#">Enabling Multiversion Backup When Creating an OBS Table</a> <a href="#">Enabling or Disabling Multiversion Backup When Modifying a Table</a> <a href="#">Setting the Retention Period for Multiversion Backup Data</a> <a href="#">Checking Multiversion Backup Data</a> <a href="#">Restoring Multiversion Backup Data</a> <a href="#">Configuring the Trash Bin for Expired Multiversion Data</a> <a href="#">Deleting Multiversion Backup Data</a>

# 3 Spark Open Source Commands

This section describes the open source Spark SQL syntax supported by DLI. For details about the syntax, parameters, and examples, see [Spark SQL Syntax](#).

**Table 3-1** DLI Spark open source commands

Function	Syntax Example	DLI Spark 2.4.5	DLI Spark 3.3.1
Creating a database	CREATE DATABASE testDB;	Supported	Supported
Creating a DLI table	create table if not exists testDB.testTable1( id int, age int, money double );	Supported	Supported
Creating an OBS table	create table if not exists testDB.testTable2( id int, age int, money double ) LOCATION 'obs://bucketName/ filePath' partitioned by (dt string) row format delimited fields terminated by ',' STORED as TEXTFILE;	Supported	Supported

Function	Syntax Example	DLI Spark 2.4.5	DLI Spark 3.3.1
Inserting test data	insert into table testDB.testTable2 values (1, 18, 3.14, "20240101" ), (2, 18, 3.15, "20240102" );	Supported	Supported
Changing database attributes	ALTER DATABASE testDB SET DBPROPERTIES ('Edited-by' = 'John');	Not supported	Not supported
Changing the path for storing database files on OBS	ALTER DATABASE testDB SET LOCATION 'obs://bucketName/filePath';	Not supported	Not supported
Changing a table name	ALTER TABLE testDB.testTable1 RENAME TO testDB.testTable101;	Supported	Supported
Changing the name of a partition in a table	ALTER TABLE testDB.testTable2 PARTITION ( dt='20240101') RENAME TO PARTITION ( dt='20240103');  It only supports renaming partitions of tables stored in OBS, and the storage path of the table in OBS will not change.	Supported	Supported
Adding a column	ALTER TABLE testDB.testTable1 ADD COLUMNS (name string);	Supported	Supported
Dropping a column	ALTER TABLE testDB.testTable1 DROP columns (money);	Not supported	Not supported
Changing the name of a column	ALTER TABLE testDB.testTable1 RENAME COLUMN age TO years_of_age;	Not supported	Not supported
Modifying column comments	ALTER TABLE testDB.testTable1 ALTER COLUMN age COMMENT "new comment";	Not supported	Supported
Replacing a specified column	ALTER TABLE testDB.testTable1 REPLACE COLUMNS (name string, ID int COMMENT 'new comment');	Not supported	Not supported
Customizing table attributes	ALTER TABLE testDB.testTable1 SET TBLPROPERTIES ('aaa' = 'bbb');	Supported	Supported
Adding or modifying table comments	ALTER TABLE testDB.testTable1 SET TBLPROPERTIES ('comment' = 'test');	Supported	Not supported

Function	Syntax Example	DLI Spark 2.4.5	DLI Spark 3.3.1
Changing the storage format of a table	ALTER TABLE testDB.testTable1 SET fileformat csv;	Not supported	Not supported
Deleting a table attribute	ALTER TABLE testDB.testTable1 UNSET TBLPROPERTIES ('test');	Supported	Supported
Deleting table comments	ALTER TABLE testDB.testTable1 UNSET TBLPROPERTIES ('comment');	Supported	Supported
Displaying column information	DESCRIBE database_name.table_name col_name; Example: <b>DESCRIBE testDB.testTable1 id;</b>	Supported	Supported
Displaying column information	DESCRIBE table_name table_name.col_name; Example: <b>DESCRIBE testTable1 testTable1.id;</b> It only supports viewing the column information for tables in the current database.	Supported	Supported
Returning metadata information of a query statement	DESCRIBE QUERY SELECT age, sum(age) FROM testDB.testTable1 GROUP BY age;	Not supported	Supported
Returning metadata information of inserted data	DESCRIBE QUERY VALUES(100, 10, 10000.20D) AS testTable1(id, age, money);	Not supported	Supported
Returning metadata information of a table	DESCRIBE QUERY TABLE testTable1;	Not supported	Supported
Returning metadata information of a column in a table	DESCRIBE FROM testTable1 SELECT age;	Not supported	Supported

Function	Syntax Example	DLI Spark 2.4.5	DLI Spark 3.3.1
Returning the table creation statements for a table	SHOW CREATE TABLE testDB.testTable1 AS SERDE;  When executing this statement in Spark 3.3.1, it only applies to query the table creation statements for Hive tables.	Not supported	Supported
Returning the table creation statements for a table	SHOW CREATE TABLE testDB.testTable1;	Supported	Supported

# 4 Databases

## 4.1 Creating a Database

### Function

This statement is used to create a database.

### Syntax

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

### Keywords

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

### Parameters

Table 4-1 Parameters

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

## Precautions

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

## Example

### NOTE

For details about the complete process for submitting SQL jobs, see [Submitting a SQL Job](#).

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

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

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

## 4.2 Deleting a Database

### Function

This statement is used to delete a database.

### Syntax

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

### Keywords

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

### Precautions

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

## Parameters

**Table 4-2** Parameter

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

## Example

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

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

## 4.3 Viewing a Specified Database

### Function

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

### Syntax

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

### Keywords

EXTENDED: Displays the database properties.

### Parameters

**Table 4-3** Parameter

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

### Precautions

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

### Example

1. Create a database, for example, **testdb**, by referring to [Example](#).

2. Run the following statement to query information about the **testdb** database:  

```
DESCRIBE DATABASE testdb;
```

## 4.4 Viewing All Databases

### Function

This syntax is used to query all current databases.

### Syntax

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

### Keywords

None

### Parameters

**Table 4-4** Parameter

Parameter	Description
regex_expression	Database name

### Precautions

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

### Example

View all the current databases.

```
SHOW DATABASES;
```

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

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

# 5 Tables

---

## 5.1 Creating an OBS Table

### 5.1.1 Creating an OBS Table Using the DataSource Syntax

#### Function

Create an OBS table using the DataSource syntax.

The main differences between the DataSource and the Hive syntax lie in the supported data formats and the number of supported partitions. For details, see syntax and precautions.

#### NOTE

You are advised to use an OBS parallel file system for storage. A parallel file system is a high-performance file system that provides latency in milliseconds, TB/s-level bandwidth, and millions of IOPS. It applies to interactive big data analysis scenarios.

#### Precautions

- The size of a table is not calculated when the table is created.
- When data is added, the table size will be changed to 0.
- You can check the table size on OBS.
- Table properties cannot be specified using CTAS table creation statements.
- **An OBS directory containing subdirectories:**

If you specify an OBS directory that contains subdirectories when creating a table, all file types and content within those subdirectories will also be included as table content.

Ensure that all file types in the specified directory and its subdirectories are consistent with the storage format specified in the table creation statement. All file content must be consistent with the fields in the table. Otherwise, errors will be reported in the query.

You can set **multiLevelDirEnable** to **true** in the **OPTIONS** statement to query the content in the subdirectory. The default value is **false** (Note that this

configuration item is a table attribute, exercise caution when performing this operation. Hive tables do not support this configuration item.)

- **Instructions on using partitioned tables:**
  - When a partitioned table is created, the column specified in PARTITIONED BY must be a column in the table, and the partition type must be specified. The partition column supports only the **string, boolean, tinyint, smallint, short, int, bigint, long, decimal, float, double, date, and timestamp** type.
  - When a partitioned table is created, the partition field must be the last one or several fields of the table field, and the sequence of the partition fields must be the same. Otherwise, an error occurs.
  - A maximum of 200,000 partitions can be created in a single table.
  - Starting from January 2024, new users who register for DLI and use Spark 3.3 or later will be able to use CTAS statements to create partitioned tables when creating tables using the DataSource syntax.

## 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:** Prevents system errors when the created table exists.
- **USING:** Storage format.
- **OPTIONS:** Property name and property value when a table is created.
- **COMMENT:** Field or table description.
- **PARTITIONED BY:** Partition field.
- **AS:** Run the **CREATE TABLE AS** statement to create a table.

## Parameters

Table 5-1 Parameters

Parameter	Mandatory	Description
db_name	No	Database name The value can contain letters, numbers, and underscores (_), but it cannot contain only numbers or start with a number or underscore (_).

Parameter	Mandatory	Description
table_name	Yes	Name of the table to be created in the database The value can contain letters, numbers, and underscores (_), but it cannot contain only numbers or start with a number or underscore (_). The matching rule is <code>^(?!_)(?![0-9]+\$)[A-Za-z0-9_\$]*\$</code> . Special characters must be enclosed in single quotation marks ("). The table name is case insensitive.
col_name	Yes	Column names with data types separated by commas (,) The column name can contain letters, numbers, and underscores (_), but it cannot contain only numbers and must contain at least one letter. The column name is case insensitive.
col_type	Yes	Data type of a column field, which is primitive. For details, see <a href="#">Primitive Data Types</a> .
col_comment	No	Column field description, which can only be string constants.
file_format	Yes	Format of the table to be created, which can be <b>orc</b> , <b>parquet</b> , <b>json</b> , <b>csv</b> , or <b>avro</b> .
path	Yes	OBS storage path where data files are stored. You are advised to use an OBS parallel file system for storage. Format: <b>obs://bucketName/tblPath</b> <i>bucketName</i> : bucket name <i>tblPath</i> : directory name. You do not need to specify the file name following the directory. Refer to <a href="#">Table 5-2</a> for details about property names and values during table creation. Refer to <a href="#">Table 5-2</a> and <a href="#">Table 5-3</a> for details about the table property names and values when <b>file_format</b> is set to <b>csv</b> . If there is a folder and a file with the same name in the OBS directory, the path pointed to by the OBS table will prioritize the file over the folder.
table_comment	No	Table description, which can only be string constants.
select_statement	No	Used in the <b>CREATE TABLE AS</b> statement to insert the <b>SELECT</b> query results of the source table or a data record to a table newly created in the OBS bucket.

**Table 5-2** OPTIONS parameters

Parameter	Mandatory	Description
path	No	Path where the table is stored, which currently can only be an OBS directory
multiLevelDirEnable	No	Whether data in subdirectories is iteratively queried when there are nested subdirectories. When this parameter is set to <b>true</b> , all files in the table path, including files in subdirectories, are iteratively read when a table is queried. Default value: <b>false</b>
dataDelegated	No	Whether data in the path is cleared when deleting a table or partition Default value: <b>false</b>
compression	No	Compression format. This parameter is typically required for Parquet files and is set to <b>zstd</b> .

When **file\_format** is set to **csv**, you can set the following OPTIONS parameters:

**Table 5-3** OPTIONS parameters of the CSV data format

Parameter	Mandatory	Description
delimiter	No	Data separator Default value: comma (,)
quote	No	Quotation character Default value: double quotation marks ("" )
escape	No	Escape character Default value: backslash (\)
multiLine	No	Whether the column data contains carriage return characters or transfer characters. The value <b>true</b> indicates yes and the value <b>false</b> indicates no. Default value: <b>false</b>
dateFormat	No	Date format of the <b>date</b> field in a CSV file Default value: yyyy-MM-dd
timestampFormat	No	Date format of the <b>timestamp</b> field in a CSV file Default value: yyyy-MM-dd HH:mm:ss

Parameter	Mandatory	Description
mode	No	Mode for parsing CSV files. The options are as follows: Default value: <b>PERMISSIVE</b> <ul style="list-style-type: none"><li>● <b>PERMISSIVE</b>: Permissive mode. If an incorrect field is encountered, set the field to <b>Null</b>.</li><li>● <b>DROPMALFORMED</b>: When an incorrect field is encountered, the entire line is discarded.</li><li>● <b>FAILFAST</b>: Error mode. If an error occurs, it is automatically reported.</li></ul>
header	No	Whether the CSV file contains header information. The value <b>true</b> indicates that the table header information is contained, and the value <b>false</b> indicates that the information is not included. Default value: <b>false</b>
nullValue	No	Character that represents the null value. For example, <b>nullValue="nl"</b> indicates that <b>nl</b> represents the null value.
comment	No	Character that indicates the beginning of the comment. For example, <b>comment= '#'</b> indicates that the line starting with <b>#</b> is a comment.
compression	No	Data compression format. Currently, <b>gzip</b> , <b>bzip2</b> , and <b>deflate</b> are supported. If you do not want to compress data, enter <b>none</b> . Default value: <b>none</b>
encoding	No	Data encoding format. Available values are <b>utf-8</b> , <b>gb2312</b> , and <b>gbk</b> . Value <b>utf-8</b> will be used if this parameter is left empty. Default value: <b>utf-8</b>

## Example 1: Creating an OBS Non-Partitioned Table

Example description: Create an OBS non-partitioned table named **table1** and use the **USING** keyword to set the storage format of the table to **orc**.

You can store OBS tables in **parquet**, **json**, or **avro** format.

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

## Example 2: Creating an OBS Partitioned Table

Example description: Create a partitioned table named **student**. The partitioned table is partitioned using **facultyNo** and **classNo**. The **student** table is partitioned by faculty number (**facultyNo**) and class number (**classNo**).

In practice, you can select a proper partitioning field and add it to the brackets following the **PARTITIONED BY** keyword.

```
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: Using CTAS to Create an OBS Non-Partitioned Table Using All or Part of the Data in the Source Table

Example description: Based on the OBS table **table1** created in [Example 1: Creating an OBS Non-Partitioned Table](#), use the CTAS syntax to copy data from **table1** to **table1\_ctas**.

When using CTAS to create a table, you can ignore the syntax used to create the table being copied. This means that regardless of the syntax used to create **table1**, you can use the DataSource syntax to create **table1\_ctas**.

In addition, in this example, the storage format of **table1** is **orc**, and the storage format of **table1\_ctas** may be **parquet**. This means that the storage format of the table created by CTAS may be different from that of the original table.

Use the **SELECT** statement following the **AS** keyword to select required data and insert the data to **table1\_ctas**.

The **SELECT** syntax is as follows: **SELECT** <Column name> **FROM** <Table name> **WHERE** <Related filter criteria>.

- In this example, **SELECT \* FROM table1** is used. \* indicates that all columns are selected from **table1** and all data in **table1** is inserted into **table1\_ctas**.

```
CREATE TABLE IF NOT EXISTS table1_ctas  
USING parquet  
OPTIONS (path 'obs:// bucketName/filePath')  
AS  
SELECT *  
FROM table1;
```

- To filter and insert data into **table1\_ctas** in a customized way, you can use the following **SELECT** statement: **SELECT col\_1 FROM table1 WHERE col\_1 = 'Ann'**. This will allow you to select only **col\_1** from **table1** and insert data into **table1\_ctas** where the value equals 'Ann'.

```
CREATE TABLE IF NOT EXISTS table1_ctas  
USING parquet  
OPTIONS (path 'obs:// bucketName/filePath')  
AS  
SELECT col_1  
FROM table1  
WHERE col_1 = 'Ann';
```

## Example 4: Creating an OBS Non-Partitioned Table and Customizing the Data Type of a Column Field

Example description: Create an OBS non-partitioned table named **table2**. You can customize the native data types of column fields based on service requirements.

- **STRING**, **CHAR**, or **VARCHAR** can be used for text characters.
- **TIMESTAMP** or **DATE** can be used for time characters.
- **INT**, **SMALLINT/SHORT**, **BIGINT/LONG**, or **TINYINT** can be used for integer characters.
- **FLOAT**, **DOUBLE**, or **DECIMAL** can be used for decimal calculation.
- **BOOLEAN** can be used if only logical switches are involved.

For details, see "Data Types" > "Primitive Data Types".

For details, see [Primitive Data Types](#).

```
CREATE TABLE IF NOT EXISTS table2 (  
  col_01 STRING,  
  col_02 CHAR (2),  
  col_03 VARCHAR (32),  
  col_04 TIMESTAMP,  
  col_05 DATE,  
  col_06 INT,  
  col_07 SMALLINT,  
  col_08 BIGINT,  
  col_09 TINYINT,  
  col_10 FLOAT,  
  col_11 DOUBLE,  
  col_12 DECIMAL (10, 3),  
  col_13 BOOLEAN  
)  
USING parquet  
OPTIONS (path 'obs://bucketName/filePath');
```

## Example 5: Creating an OBS Partitioned Table and Customizing OPTIONS Parameters

Example description: When creating an OBS table, you can customize property names and values. For details about OPTIONS parameters, see [Table 5-2](#).

In this example, an OBS partitioned table named **table3** is created and partitioned based on **col\_2**. Configure **path**, **multiLevelDirEnable**, **dataDelegated**, and **compression** in **OPTIONS**.

- **path**: OBS storage path. In this example, the value is **obs://bucketName/filePath**, where *bucketName* indicates the bucket name and *filePath* indicates the actual directory name.
- In big data scenarios, you are advised to use the OBS parallel file system for storage.
- **multiLevelDirEnable**: In this example, set this parameter to **true**, indicating that all files and subdirectories in the table path are read iteratively when the table is queried. If this parameter is not required, set it to **false** or leave it blank (the default value is **false**).
- **dataDelegated**: In this example, set this parameter to **true**, indicating that all data in the path is deleted when a table or partition is deleted. If this

parameter is not required, set it to **false** or leave it blank (the default value is **false**).

- **compression**: If the created OBS table needs to be compressed, you can use the keyword **compression** to configure the compression format. In this example, the **zstd** compression format is used.

```
CREATE TABLE IF NOT EXISTS table3 (  
  col_1 STRING,  
  col_2 int  
)  
USING parquet  
PARTITIONED BY (col_2)  
OPTIONS (  
  path 'obs://bucketName/filePath',  
  multiLevelDirenable = true,  
  dataDelegated = true,  
  compression = 'zstd'  
);
```

## Example 6: Creating an OBS Non-Partitioned Table and Customizing OPTIONS Parameters

Example description: A CSV table is a file format that uses commas to separate data values in plain text. It is commonly used for storing and sharing data, but it is not ideal for complex data types due to its lack of structured data concepts. So, when **file\_format** is set to **csv**, more **OPTIONS** parameters can be configured. For details, see [Table 5-3](#).

In this example, a non-partitioned table named **table4** is created with a **csv** storage format, and additional **OPTIONS** parameters are used to constrain the data.

- **delimiter**: data separator, indicating that commas (,) are used as separators between data
- **quote**: quotation character, indicating that double quotation marks (") are used to quote the reference information in the data
- **escape**: escape character, indicating that backslashes (\) are used as the escape character for data storage
- **multiLine**: This parameter is used to set the column data to be stored does not include carriage return or newline characters.
- **dateFormat**: indicates that the date format of the **data** field in the CSV file is **yyyy-MM-dd**.
- **timestampFormat**: indicates that the timestamp format in the CSV file is **yyyy-MM-dd HH:mm:ss**.
- **header**: indicates that the CSV table contains the table header information.
- **nullValue**: indicates that **null** is set to indicate the null value in the CSV table.
- **comment**: indicates that the CSV table uses a slash (/) to indicate the beginning of a comment.
- **compression**: indicates that the CSV table is compressed in the **gzip**, **bzip2**, or **deflate** format. If compression is not required, set this parameter to **none**.
- **encoding**: indicates that the table uses the **utf-8** encoding format. You can choose **utf-8**, **gb2312**, or **gbk** as needed. The default encoding format is **utf-8**.

```
CREATE TABLE IF NOT EXISTS table4 (  
  col_1 STRING,  
  col_2 INT  
)  
USING csv  
OPTIONS (  
  path 'obs://bucketName/filePath',  
  delimiter = ',',  
  quote = '#',  
  escape = '|',  
  multiline = false,  
  dateFormat = 'yyyy-MM-dd',  
  timestampFormat = 'yyyy-MM-dd HH:mm:ss',  
  mode = 'failfast',  
  header = true,  
  nullValue = 'null',  
  comment = '*',  
  compression = 'deflate',  
  encoding = 'utf - 8'  
);
```

## FAQ

- **What should I do if the error message "xxx dli datasource v2 tables is only supported in spark3.3 or later version." appears when I create a DataSource table using the default queue?**

Ensure that you use Spark 3.3.1 or a later version when creating such a table. If the error message appears, use the Hive syntax to create the table. For details, see [Creating an OBS Table Using the Hive Syntax](#).

- **What should I do if the error message "xxx don't support dli v1 table." appears when I use Spark 3.3.1 to run a Jar job?**

This error message indicates that table operations cannot be performed when Spark 3.3.1 is used to execute the Jar job. Use the Hive syntax to recreate the tables' data structure. For example, you can use **[STORED AS file\_format] CTAS** to recreate the table and then run the job. For details, see [Creating an OBS Table Using the Hive Syntax](#).

## 5.1.2 Creating an OBS Table Using the Hive Syntax

### Function

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

#### NOTE

You are advised to use the OBS parallel file system for storage. A parallel file system is a high-performance file system that provides latency in milliseconds, TB/s-level bandwidth, and millions of IOPS. It applies to interactive big data analysis scenarios.

### Precautions

- The size of a table is calculated when the table is created.
- When data is added, the table size will not be changed.
- You can check the table size on OBS.

- Table properties cannot be specified using CTAS table creation statements.
- **Instructions on using partitioned tables:**
  - When you create a partitioned table, ensure that the specified column in **PARTITIONED BY** is not a column in the table and the data type is specified. The partition column supports only the open-source Hive table types including **string**, **boolean**, **tinyint**, **smallint**, **short**, **int**, **bigint**, **long**, **decimal**, **float**, **double**, **date**, and **timestamp**.
  - Multiple partition fields can be specified. The partition fields need to be specified after the **PARTITIONED BY** keyword, instead of the table name. Otherwise, an error occurs.
  - A maximum of 200,000 partitions can be created in a single table.
  - Spark 3.3 or later supports creating partitioned tables using CTAS statements of the Hive syntax.
- **Instructions on setting the multi-character delimiter during table creation:**
  - The field delimiter can contain multiple characters only when **ROW FORMAT SERDE** is set to **org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe**.
  - You can only specify multi-character delimiters when creating Hive OBS tables.
  - Tables with multi-character delimiters specified do not support data writing statements such as **INSERT** and **IMPORT**. To insert data into these tables, save the data file in the OBS path where the tables are located. For example, in [Example 7: Creating a Table and Setting a Multi-Character Delimiter](#), store the data file in **obs://bucketName/filePath**.

## Syntax

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name1 col_type1 [COMMENT col_comment1], ...)]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name2 col_type2, [COMMENT col_comment2], ...)]
  [ROW FORMAT row_format]
  [STORED AS file_format]
  LOCATION 'obs_path'
  [TBLPROPERTIES (key = value)]
  [AS select_statement]
row_format:
  : SERDE serde_cls [WITH SERDEPROPERTIES (key1=val1, key2=val2, ...)]
  | DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]]
  [COLLECTION ITEMS TERMINATED BY char]
  [MAP KEYS TERMINATED BY char]
  [LINES TERMINATED BY char]
  [NULL DEFINED AS char]
```

## Keywords

- **EXTERNAL:** Creates an OBS table.
- **IF NOT EXISTS:** Prevents system errors when the created table exists.
- **COMMENT:** Field or table description.
- **PARTITIONED BY:** Partition field.
- **ROW FORMAT:** Row data format.

- **STORED AS:** Specifies the format of the file to be stored. Currently, only the TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, and PARQUET format are supported.
- **LOCATION:** Specifies the path of OBS. This keyword is mandatory when you create OBS tables.
- **TBLPROPERTIES:** Allows you to add the **key/value** properties to a table.
  - (The multi-versioning function has been deprecated and is not recommended.) Enables data multi-versioning for table data backup and restoration. After the multiversion function is enabled, the system automatically backs up table data when you delete or modify the data using **insert overwrite** or **truncate**, and retains the data for a certain period. You can quickly restore data within the retention period. For details about the SQL syntax for the multiversion function, see [Enabling or Disabling Data Multi-Versioning \(Deprecated, Not Recommended\)](#) and [Backing Up and Restoring Multi-Versioning Data \(Deprecated, Not Recommended\)](#).

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

**Table 5-4** TBLPROPERTIES parameters

Key	Value
dli.multi.version.enable	<ul style="list-style-type: none"> <li>• <b>true:</b> Enable the multiversion backup function.</li> <li>• <b>false:</b> Disable the multiversion backup function.</li> </ul>
comment	Description of the table
orc.compress	An attribute of the ORC table, which specifies the compression mode of the ORC storage. Available values are as follows: <ul style="list-style-type: none"> <li>• <b>ZLIB</b></li> <li>• <b>SNAPPY</b></li> <li>• <b>NONE</b></li> </ul>
auto.purge	If this parameter is set to <b>true</b> , the deleted or overwritten data is removed and will not be dumped to the recycle bin.

- **AS:** Run the **CREATE TABLE AS** statement to create a table.
- The field delimiter can contain multiple characters only when **ROW FORMAT SERDE** is set to **org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe**. For details, see [Example 7: Creating a Table and Setting a Multi-Character Delimiter](#).

## Parameters

**Table 5-5** Parameters

Parameter	Mandatory	Description
db_name	No	Database name The value can contain letters, numbers, and underscores (_), but it cannot contain only numbers or start with a number or underscore (_).
table_name	Yes	Table name in the database The value can contain letters, numbers, and underscores (_), but it cannot contain only numbers or start with a number or underscore (_). The matching rule is <code>^(?!_)(?![0-9]+\$)[A-Za-z0-9_]*\$</code> . Special characters must be enclosed in single quotation marks ("). The table name is case insensitive.
col_name	Yes	Name of a column field The column field can contain letters, numbers, and underscores (_), but it cannot contain only numbers and must contain at least one letter. The column name is case insensitive.
col_type	Yes	Data type of a column field, which is primitive. For details, see <a href="#">Primitive Data Types</a> .
col_comment	No	Column field description, which can only be string constants.
row_format	Yes	Row data format This function is available only for textfile tables.
file_format	Yes	OBS table storage format, which can be <b>TEXTFILE</b> , <b>AVRO</b> , <b>ORC</b> , <b>SEQUENCEFILE</b> , <b>RCFILE</b> , or <b>PARQUET</b> .
table_comment	No	Table description, which can only be string constants.

Parameter	Mandatory	Description
obs_path	Yes	OBS storage path where data files are stored. You are advised to use an OBS parallel file system for storage. Format: <b>obs://bucketName/tblPath</b> <i>bucketName</i> : bucket name <i>tblPath</i> : directory name. You do not need to specify the file name following the directory. If there is a folder and a file with the same name in the OBS directory, the path pointed to by the OBS table will prioritize the file over the folder.
key = value	No	Set table properties and values. For example, if you want to enable multiversion, you can set " <b>dli.multi.version.enable</b> "=" <b>true</b> ".
select_statement	No	Used in the <b>CREATE TABLE AS</b> statement to insert the <b>SELECT</b> query results of the source table or a data record to a table newly created in the OBS bucket.

## Example 1: Creating an OBS Non-Partitioned Table

Example description: Create an OBS non-partitioned table named **table1** and use the **STORED AS** keyword to set the storage format of the table to **orc**.

In practice, you can store OBS tables in **textfile**, **avro**, **orc**, **sequencefile**, **rcfile**, or **parquet** format.

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

## Example 2: Creating an OBS Partitioned Table

Example description: Create a partitioned table named **student**, which is partitioned using **facultyNo** and **classNo**.

In practice, you can select a proper partitioning field and add it to the end of the **PARTITIONED BY** keyword.

```
CREATE TABLE IF NOT EXISTS student(  
  id INT,  
  name STRING  
)  
STORED AS avro  
LOCATION 'obs://bucketName/filePath'  
PARTITIONED BY (  
  facultyNo INT,  
  classNo INT  
);
```

### Example 3: Using CTAS to Create an OBS Table Using All or Part of the Data in the Source Table

Example description: Based on the OBS table **table1** created in [Example 1: Creating an OBS Non-Partitioned Table](#), use the CTAS syntax to copy data from **table1** to **table1\_ctas**.

When using CTAS to create a table, you can ignore the syntax used to create the table being copied. This means that regardless of the syntax used to create **table1**, you can use the DataSource syntax to create **table1\_ctas**.

In addition, in this example, the storage format of **table1** is **orc**, and the storage format of **table1\_ctas** may be **sequencefile** or **parquet**. This means that the storage format of the table created by CTAS may be different from that of the original table.

Use the **SELECT** statement following the **AS** keyword to select required data and insert the data to **table1\_ctas**.

The **SELECT** syntax is as follows: **SELECT** <Column name> **FROM** <Table name> **WHERE** <Related filter criteria>.

- In this example, **SELECT \* FROM table1** is used. \* indicates that all columns are selected from **table1** and all data in **table1** is inserted into **table1\_ctas**.
- To filter and insert data into **table1\_ctas** in a customized way, you can use the following **SELECT** statement: **SELECT col\_1 FROM table1 WHERE col\_1 = 'Ann'**. This will allow you to select only **col\_1** from **table1** and insert data into **table1\_ctas** where the value equals 'Ann'.

```
CREATE TABLE IF NOT EXISTS table1_ctas
STORED AS sequencefile
LOCATION 'obs://bucketName/filePath'
AS
SELECT *
FROM table1;
```

```
CREATE TABLE IF NOT EXISTS table1_ctas
STORED AS parquet
LOCATION 'obs:// bucketName/filePath'
AS
SELECT col_1
FROM table1
WHERE col_1 = 'Ann';
```

### Example 4: Creating an OBS Non-Partitioned Table and Customizing the Data Type of a Column Field

Example description: Create an OBS non-partitioned table named **table2**. You can customize the native data types of column fields based on service requirements.

- **STRING**, **CHAR**, or **VARCHAR** can be used for text characters.
- **TIMESTAMP** or **DATE** can be used for time characters.
- **INT**, **SMALLINT/SHORT**, **BIGINT/LONG**, or **TINYINT** can be used for integer characters.
- **FLOAT**, **DOUBLE**, or **DECIMAL** can be used for decimal calculation.
- **BOOLEAN** can be used if only logical switches are involved.

For details, see "Data Types" > "Primitive Data Types".

For details, see [Primitive Data Types](#).

```
CREATE TABLE IF NOT EXISTS table2 (  
  col_01 STRING,  
  col_02 CHAR (2),  
  col_03 VARCHAR (32),  
  col_04 TIMESTAMP,  
  col_05 DATE,  
  col_06 INT,  
  col_07 SMALLINT,  
  col_08 BIGINT,  
  col_09 TINYINT,  
  col_10 FLOAT,  
  col_11 DOUBLE,  
  col_12 DECIMAL (10, 3),  
  col_13 BOOLEAN  
)  
STORED AS parquet  
LOCATION 'obs://bucketName/filePath';
```

## Example 5: Creating an OBS Partitioned Table and Customizing TBLPROPERTIES Parameters

Example description: Create an OBS partitioned table named **table3** and partition the table based on **col\_3**. Set **dli.multi.version.enable**, **comment**, **orc.compress**, and **auto.purge** in **TBLPROPERTIES**.

- **dli.multi.version.enable**: In this example, set this parameter to **true**, indicating that the DLI data versioning function is enabled for table data backup and restoration.
- **comment**: table description, which can be modified later.
- **orc.compress**: compression mode of the **orc** format, which is **ZLIB** in this example.
- **auto.purge**: In this example, set this parameter to **true**, indicating that data that is deleted or overwritten will bypass the recycle bin and be permanently deleted.

```
CREATE TABLE IF NOT EXISTS table3 (  
  col_1 STRING,  
  col_2 STRING  
)  
PARTITIONED BY (col_3 DATE)  
STORED AS rcfile  
LOCATION 'obs://bucketName/filePath'  
TBLPROPERTIES (  
  dli.multi.version.enable = true,  
  comment                  = 'Created by dli',  
  orc.compress              = 'ZLIB',  
  auto.purge                = true  
);
```

## Example 6: Creating a Non-Partitioned Table in textfile Format and Setting ROW FORMAT

Example description: Create a non-partitioned table named **table4** in the **textfile** format and set **ROW FORMAT** (the ROW FORMAT function is available only for textfile tables).

- **FIELDS**: columns in a table. Each field has a name and data type. Fields in a table are separated by slashes (/).
- **COLLECTION ITEMS**: A collection item refers to an element in a group of data, which can be an array, a list, or a collection. Collection items in a table are separated by \$.

- **MAP KEYS:** A map key is a data structure of key-value pairs and is used to store a group of associated data. Map keys in a table are separated by number signs (#).
- **LINES:** rows in a table. Each row contains a group of field values. Rows in a table end with `\n`. (Note that only `\n` can be used as the row separator.)
- **NULL:** a special value that represents a missing or unknown value. In a table, **NULL** indicates that the field has no value or the value is unknown. When there is a null value in the data, it is represented by the string **null**.

```
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 7: Creating a Table and Setting a Multi-Character Delimiter

Example description: A Hive table named **table5** is created. The serialization and deserialization class **ROW FORMAT SERDE** is specified for the table, with a field delimiter set to **/#** and data stored in a text file format.

- The field delimiter can contain multiple characters only when **ROW FORMAT SERDE** is set to **org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe**.
- You can only specify multi-character delimiters when creating Hive OBS tables.
- Tables with multi-character delimiters specified do not support data writing statements such as **INSERT** and **IMPORT**. To insert data into these tables, save the data file in the OBS path where the tables are located. For example, in this example, store the data file in **obs://bucketName/filePath**.
- In this example, **field.delim** is set to **/#**.
- The **ROW FORMAT** function is available only for textfile tables.

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

## 5.2 Creating a DLI Table

## 5.2.1 Creating a DLI Table Using the DataSource Syntax

### Function

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

### Precautions

- Table properties cannot be specified using CTAS table creation statements.
- If no separator is specified, a comma (,) is used by default.
- **Instructions on using partitioned tables:**
  - When a partitioned table is created, the column specified in PARTITIONED BY must be a column in the table, and the partition type must be specified. Partition columns can only be in the **string, boolean, tinyint, smallint, short, int, bigint, long, decimal, float, double, date, or timestamp** format.
  - When a partitioned table is created, the partition field must be the last one or several fields of the table field, and the sequence of the partition fields must be the same. Otherwise, an error occurs.
  - A maximum of 200,000 partitions can be created in a single table.
  - Starting from January 2024, new users who register for DLI and use Spark 3.3 or later will be able to use CTAS statements to create partitioned tables when creating tables using the DataSource syntax.

### 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:** Prevents system errors when the created table exists.
- **USING:** Storage format.
- **OPTIONS:** Property name and property value when a table is created.
- **COMMENT:** Field or table description.
- **PARTITIONED BY:** Partition field.
- **AS:** Run the **CREATE TABLE AS** statement to create a table.

## Parameters

**Table 5-6** Parameters

Parameter	Mandatory	Description
db_name	No	Database name The value can contain letters, numbers, and underscores (_), but it cannot contain only numbers or start with a number or underscore (_).
table_name	Yes	Table name in the database The value can contain letters, numbers, and underscores (_), but it cannot contain only numbers or start with a number or underscore (_). The matching rule is <code>^(?!_)(?![0-9]+\$)[A-Za-z0-9_]*\$</code> . Special characters must be enclosed in single quotation marks ("). The table name is case insensitive.
col_name	Yes	Column names with data types separated by commas (,) The column name can contain letters, numbers, and underscores (_), but it cannot contain only numbers and must contain at least one letter. The column name is case insensitive.
col_type	Yes	Data type of a column field, which is primitive. For details, see <a href="#">Primitive Data Types</a> .
col_comment	No	Column field description, which can only be string constants.
file_format	Yes	Data storage format of DLI tables. The value can be <b>parquet</b> or <b>orc</b> .
table_comment	No	Table description, which can only be string constants.
select_statement	No	The <b>CREATE TABLE AS</b> statement is used to insert the <b>SELECT</b> query result of the source table or a data record to a newly created DLI table.

**Table 5-7** OPTIONS parameters

Parameter	Mandatory	Description	Default Value
multiLevelDir Enable	No	Whether to iteratively query data in subdirectories. When this parameter is set to <b>true</b> , all files in the table path, including files in subdirectories, are iteratively read when a table is queried.	false
compression	No	Compression format. Generally, you need to set this parameter to <b>zstd</b> for parquet files.	-

### Example 1: Creating a DLI Non-Partitioned Table

Example description: Create a DLI non-partitioned table named **table1** and use the **USING** keyword to set the storage format of the table to **orc**.

You can save DLI tables in the **parquet** format.

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

### Example 2: Creating a DLI Partitioned Table

Example description: Create a partitioned table named **student**, which is partitioned using **facultyNo** and **classNo**.

In practice, you can select a proper partitioning field and add it to the end of the **PARTITIONED BY** keyword.

```
CREATE TABLE IF NOT EXISTS student (  
  Name      STRING,  
  facultyNo INT,  
  classNo   INT  
)  
USING orc  
PARTITIONED BY (facultyNo, classNo);
```

### Example 3: Using CTAS to Create a DLI Table Using All or Part of the Data in the Source Table

Example description: Based on the DLI table **table1** created in [Example 1: Creating a DLI Non-Partitioned Table](#), use the CTAS syntax to copy data from **table1** to **table1\_ctas**.

When using CTAS to create a table, you can ignore the syntax used to create the table being copied. This means that regardless of the syntax used to create **table1**, you can use the DataSource syntax to create **table1\_ctas**.

In addition, in this example, the storage format of **table1** is **orc**, and the storage format of **table1\_ctas** may be **orc** or **parquet**. This means that the storage format of the table created by CTAS may be different from that of the original table.

Use the **SELECT** statement following the **AS** keyword to select required data and insert the data to **table1\_ctas**.

The **SELECT** syntax is as follows: **SELECT** <Column name> **FROM** <Table name> **WHERE** <Related filter criteria>.

- In this example, **SELECT \* FROM table1** is used. **\*** indicates that all columns are selected from **table1** and all data in **table1** is inserted into **table1\_ctas**.

```
CREATE TABLE IF NOT EXISTS table1_ctas
USING parquet
AS
SELECT *
FROM table1;
```

- To filter and insert data into **table1\_ctas** in a customized way, you can use the following **SELECT** statement: **SELECT col\_1 FROM table1 WHERE col\_1 = 'Ann'**. This will allow you to select only **col\_1** from **table1** and insert data into **table1\_ctas** where the value equals **'Ann'**.

```
CREATE TABLE IF NOT EXISTS table1_ctas
USING parquet
AS
SELECT col_1
FROM table1
WHERE col_1 = 'Ann';
```

## Example 4: Creating a DLI Non-Partitioned Table and Customizing the Data Type of a Column Field

Example description: Create a DLI non-partitioned table named **table2**. You can customize the native data types of column fields based on service requirements.

- **STRING**, **CHAR**, or **VARCHAR** can be used for text characters.
- **TIMESTAMP** or **DATE** can be used for time characters.
- **INT**, **SMALLINT/SHORT**, **BIGINT/LONG**, or **TINYINT** can be used for integer characters.
- **FLOAT**, **DOUBLE**, or **DECIMAL** can be used for decimal calculation.
- **BOOLEAN** can be used if only logical switches are involved.

For details, see "Data Types" > "Primitive Data Types".

For details, see [Primitive Data Types](#).

```
CREATE TABLE IF NOT EXISTS table2 (
  col_01 STRING,
  col_02 CHAR (2),
  col_03 VARCHAR (32),
  col_04 TIMESTAMP,
  col_05 DATE,
  col_06 INT,
  col_07 SMALLINT,
  col_08 BIGINT,
  col_09 TINYINT,
  col_10 FLOAT,
  col_11 DOUBLE,
  col_12 DECIMAL (10, 3),
  col_13 BOOLEAN
)
USING parquet;
```

## Example 5: Creating a DLI Partitioned Table and Customizing OPTIONS Parameters

Example description: When creating a DLI table, you can customize property names and values. For details about OPTIONS parameters, see [Table 5-7](#).

In this example, a DLI partitioned table named **table3** is created and partitioned based on **col\_2**. Set **multiLevelDirEnable** and **compression** in **OPTIONS**.

- **multiLevelDirEnable**: In this example, this parameter is set to **true**, indicating that all files and subdirectories in the table path are read iteratively when the table is queried. If this parameter is not required, set it to **false** or leave it blank (the default value is **false**).
- **compression**: If the created OBS table needs to be compressed, you can use the keyword **compression** to configure the compression format. In this example, the **zstd** compression format is used.

```
CREATE TABLE IF NOT EXISTS table3 (  
  col_1 STRING,  
  col_2 int  
)  
USING parquet  
PARTITIONED BY (col_2)  
OPTIONS (  
  multiLevelDirEnable = true,  
  compression         = 'zstd'  
);
```

## FAQ

- **What should I do if the error message "xxx dli datasource v2 tables is only supported in spark3.3 or later version." appears when I create a DataSource table using the default queue?**

Ensure that you use Spark 3.3.1 or a later version when creating such a table. If the error message appears, use the Hive syntax to create the table. For details, see [Creating a DLI Table Using the Hive Syntax](#).

- **What should I do if the error message "xxx don't support dli v1 table." appears when I use Spark 3.3.1 to run a Jar job?**

This error message indicates that table operations cannot be performed when Spark 3.3.1 is used to execute the Jar job. Use the Hive syntax to recreate the tables' data structure. For example, you can use **[STORED AS file\_format] CTAS** to recreate the table and then run the job. For how to create a table, see [Creating a DLI Table Using the Hive Syntax](#).

## 5.2.2 Creating a DLI Table Using the Hive Syntax

### Function

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

### Precautions

- Table properties cannot be specified using CTAS table creation statements.

- You cannot specify multi-character delimiters when creating Hive DLI tables.
- **Instructions on using partitioned tables:**
  - When you create a partitioned table, ensure that the specified column in **PARTITIONED BY** is not a column in the table and the data type is specified. The partition column supports only the open-source Hive table types including **string**, **boolean**, **tinyint**, **smallint**, **short**, **int**, **bigint**, **long**, **decimal**, **float**, **double**, **date**, and **timestamp**.
  - Multiple partition fields can be specified. The partition fields need to be specified after the **PARTITIONED BY** keyword, instead of the table name. Otherwise, an error occurs.
  - A maximum of 200,000 partitions can be created in a single table.
  - Spark 3.3 or later supports creating partitioned tables using CTAS statements of the Hive syntax.

## 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 (key = value)]
[AS select_statement];
```

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

## Keywords

- **IF NOT EXISTS:** Prevents system errors when the created table exists.
- **COMMENT:** Field or table description.
- **PARTITIONED BY:** Partition field.
- **ROW FORMAT:** Row data format.
- **STORED AS:** Specifies the format of the file to be stored. Currently, only the **TEXTFILE**, **AVRO**, **ORC**, **SEQUENCEFILE**, **RCFILE**, and **PARQUET** format are supported. This keyword is mandatory when you create DLI tables.
- **TBLPROPERTIES:** This keyword is used to add a **key/value** property to a table.
  - If the table storage format is Parquet, you can use **TBLPROPERTIES(parquet.compression = 'zstd')** to set the table compression format to **zstd**.
- **AS:** Run the **CREATE TABLE AS** statement to create a table.

## Parameters

**Table 5-8** Parameters

Parameter	Mandatory	Description
db_name	No	Database name The value can contain letters, numbers, and underscores (_), but it cannot contain only numbers or start with a number or underscore (_).
table_name	Yes	Table name in the database The value can contain letters, numbers, and underscores (_), but it cannot contain only numbers or start with a number or underscore (_). The matching rule is <code>^(?!_)(?![0-9]+\$)[A-Za-z0-9_\$]*\$</code> . If special characters are required, use single quotation marks (") to enclose them.
col_name	Yes	Column name The column field can contain letters, numbers, and underscores (_), but it cannot contain only numbers and must contain at least one letter. The column name is case insensitive.
col_type	Yes	Data type of a column field, which is primitive. For details, see <a href="#">Primitive Data Types</a> .
col_comment	No	Column field description, which can only be string constants.
row_format	Yes	Row data format The ROW FORMAT function is available only for textfile tables.
file_format	Yes	Data storage format of DLI tables. The options include <b>textfile</b> , <b>avro</b> , <b>orc</b> , <b>sequencefile</b> , <b>rcfile</b> , and <b>parquet</b> .
table_comment	No	Table description, which can only be string constants.
key = value	No	Set table properties and values. If the table storage format is Parquet, you can use <b>TBLPROPERTIES(parquet.compression = 'zstd')</b> to set the table compression format to <b>zstd</b> .
select_statement	No	The <b>CREATE TABLE AS</b> statement is used to insert the <b>SELECT</b> query result of the source table or a data record to a newly created DLI table.

## Example 1: Creating a DLI Non-Partitioned Table

Example description: Create a DLI non-partitioned table named **table1** and use the **STORED AS** keyword to set the storage format of the table to **orc**.

You can save DLI tables in the **textfile**, **avro**, **orc**, **sequencefile**, **rcfile**, or **parquet** format.

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

## Example 2: Creating a DLI Partitioned Table

Example description: Create a partitioned table named **student**, which is partitioned using **facultyNo** and **classNo**.

In practice, you can select a proper partitioning field and add it to the end of the **PARTITIONED BY** keyword.

```
CREATE TABLE IF NOT EXISTS student(  
  id int,  
  name STRING  
)  
STORED AS avro  
PARTITIONED BY (  
  facultyNo INT,  
  classNo INT  
);
```

## Example 3: Using CTAS to Create a DLI Table Using All or Part of the Data in the Source Table

Example description: Based on the DLI table **table1** created in [Example 1: Creating a DLI Non-Partitioned Table](#), use the CTAS syntax to copy data from **table1** to **table1\_ctas**.

When using CTAS to create a table, you can ignore the syntax used to create the table being copied. This means that regardless of the syntax used to create **table1**, you can use the DataSource syntax to create **table1\_ctas**.

In this example, the storage format of **table1** is **orc**, and the storage format of **table1\_ctas** may be **parquet**. This means that the storage format of the table created by CTAS may be different from that of the original table.

Use the **SELECT** statement following the **AS** keyword to select required data and insert the data to **table1\_ctas**.

The **SELECT** syntax is as follows: **SELECT** <Column name> **FROM** <Table name> **WHERE** <Related filter criteria>.

- In the example, **select \* from table1** indicates that all statements are selected from **table1** and copied to **table1\_ctas**.

```
CREATE TABLE IF NOT EXISTS table1_ctas  
STORED AS sequencefile  
AS  
SELECT *  
FROM table1;
```

- If you do not need all data in **table1**, change **AS SELECT \* FROM table1** to **AS SELECT col\_1 FROM table1 WHERE col\_1 = Ann**. In this way, you can run the **SELECT** statement to insert all rows whose **col\_1** column is **Ann** from **table1** to **table1\_ctas**.

```
CREATE TABLE IF NOT EXISTS table1_ctas
USING parquet
AS
SELECT col_1
FROM table1
WHERE col_1 = 'Ann';
```

## Example 4: Creating a DLI Non-Partitioned Table and Customizing the Data Type of a Column Field

Example description: Create a DLI non-partitioned table named **table2**. You can customize the native data types of column fields based on service requirements.

- **STRING**, **CHAR**, or **VARCHAR** can be used for text characters.
- **TIMESTAMP** or **DATE** can be used for time characters.
- **INT**, **SMALLINT/SHORT**, **BIGINT/LONG**, or **TINYINT** can be used for integer characters.
- **FLOAT**, **DOUBLE**, or **DECIMAL** can be used for decimal calculation.
- **BOOLEAN** can be used if only logical switches are involved.

For details, see "Data Types" > "Primitive Data Types".

For details, see [Primitive Data Types](#).

```
CREATE TABLE IF NOT EXISTS table2 (
  col_01 STRING,
  col_02 CHAR (2),
  col_03 VARCHAR (32),
  col_04 TIMESTAMP,
  col_05 DATE,
  col_06 INT,
  col_07 SMALLINT,
  col_08 BIGINT,
  col_09 TINYINT,
  col_10 FLOAT,
  col_11 DOUBLE,
  col_12 DECIMAL (10, 3),
  col_13 BOOLEAN
)
STORED AS parquet;
```

## Example 5: Creating a DLI Partitioned Table and Customizing TBLPROPERTIES Parameters

Example description: Create a DLI partitioned table named **table3** and partition the table based on **col\_3**. Set **dli.multi.version.enable**, **comment**, **orc.compress**, and **auto.purge** in **TBLPROPERTIES**.

- **dli.multi.version.enable**: In this example, set this parameter to **true**, indicating that the DLI data versioning function is enabled for table data backup and restoration.
- **comment**: table description, which can be modified later.
- **orc.compress**: compression mode of the **orc** format, which is **ZLIB** in this example.

- **auto.purge:** In this example, set this parameter to **true**, indicating that data that is deleted or overwritten will bypass the recycle bin and be permanently deleted.

```
CREATE TABLE IF NOT EXISTS table3 (  
  col_1 STRING,  
  col_2 STRING  
)  
PARTITIONED BY (col_3 DATE)  
STORED AS rcfile  
TBLPROPERTIES (  
  dli.multi.version.enable = true,  
  comment                  = 'Created by dli',  
  orc.compress              = 'ZLIB',  
  auto.purge                = true  
);
```

## Example 6: Creating a Non-Partitioned Table in Textfile Format and Setting ROW FORMAT

Example description: In this example, create a non-partitioned table named **table4** in the **textfile** format and set **ROW FORMAT** (the ROW FORMAT function is available only for textfile tables).

- **Fields:** columns in a table. Each field has a name and data type. Fields in a table are separated by slashes (/).
- **COLLECTION ITEMS:** A collection item refers to an element in a group of data, which can be an array, a list, or a collection. Collection items in **table4** are separated by \$.
- **MAP KEYS:** A map key is a data structure of key-value pairs and is used to store a group of associated data. Map keys in a table are separated by number signs (#).
- **Rows:** rows in a table. Each row contains a group of field values. Rows in a table end with \n. (Note that only \n can be used as the row separator.)
- **NULL:** a special value that represents a missing or unknown value. In a table, **NULL** indicates that the field has no value or the value is unknown. When there is a null value in the data, it is represented by the string **null**.

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

## 5.3 Deleting a Table

### Function

This statement is used to delete tables.

## Syntax

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

## Keywords

- If the table is stored in OBS, only the metadata is deleted. The data stored on OBS is not deleted.
- If the table is stored in DLI, the data and the corresponding metadata are all deleted.

## Parameters

Table 5-9 Parameters

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

## Precautions

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

## Example

1. Create a table. For details, see [Creating an OBS Table](#) or [Creating a DLI Table](#).
2. Run the following statement to delete table **test** from the current database:  

```
DROP TABLE IF EXISTS test;
```

## 5.4 Viewing a Table

### 5.4.1 Viewing All Tables

#### Function

This statement is used to check all tables and views in the current database.

#### Syntax

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

## Keywords

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

## Parameters

**Table 5-10** Parameters

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

## Precautions

None

## Example

1. Create a table. For details, see [Creating an OBS Table](#) or [Creating a DLI Table](#).
2. To show all tables and views in the current database, run the following statement:  

```
SHOW TABLES;
```
3. To show all tables started with **test** in the **testdb** database, run the following statement:  

```
SHOW TABLES IN testdb LIKE "test*";
```

## 5.4.2 Viewing Table Creation Statements

### Function

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

### Syntax

```
SHOW CREATE TABLE table_name;
```

Use the following syntax to view table creation statements when using Spark 3.3.1 (this applies only to querying table creation statements for Hive tables):

```
SHOW CREATE TABLE table_name AS SERDE;
```

### Keywords

CREATE TABLE: statement for creating a table

## Parameters

**Table 5-11** Parameter

Parameter	Description
table_name	Table name

## Precautions

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

## Example

### Example of Spark 2.4.5:

- Run the following command to return the statement for creating the **testDB01.testTable5** table:

#### **SHOW CREATE TABLE testDB01.testTable5**

- Return the statement for creating the **test** table.

```
createtab_stmt
CREATE TABLE `testDB01`.`testTable5`(`id` INT, `age` INT, `money` DOUBLE)
COMMENT 'test'
ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
WITH SERDEPROPERTIES (
  'serialization.format' = '1'
)
STORED AS
INPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
TBLPROPERTIES (
  'hive.serialization.extend.nesting.levels' = 'true',
  'ddlUpdateTime' = '1707202585460'
)
```

### Example of Spark 3.3.1:

- Run the following command to return the statement for creating the **testDB02.testTable5** table:

#### **SHOW CREATE TABLE testDB02.testTable5 AS SERDE**

- Return the statement for creating the **test** table.

```
createtab_stmt
CREATE TABLE testDB02.testTable5 (
  id INT,
  age INT,
  money DOUBLE)
ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
WITH SERDEPROPERTIES (
  'serialization.format' = '1')
STORED AS
INPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
TBLPROPERTIES (
  'hive.serialization.extend.nesting.levels' = 'true',
  'transient_lastDdlTime' = '1707201874')
```

## 5.4.3 Viewing Table Properties

### Function

Check the properties of a table.

### Syntax

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

### Keywords

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

### Parameters

Table 5-12 Parameters

Parameter	Description
table_name	Table name
property_name	<ul style="list-style-type: none"><li>• If this parameter is not specified, all properties and their values are returned.</li><li>• If a property name is specified, only the specified property and its value are returned.</li></ul>

### Precautions

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

### Example

To return the value of **property\_key1** in the test table, run the following statement:

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

## 5.4.4 Viewing All Columns in a Specified Table

### Function

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

### Syntax

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

## Keywords

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

## Parameters

**Table 5-13** Parameters

Parameter	Description
table_name	Table name
db_name	Database name

## Precautions

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

## Example

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

```
SHOW COLUMNS IN student;
```

## 5.4.5 Viewing All Partitions in a Specified Table

### Function

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

### Syntax

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

### Keywords

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

## Parameters

**Table 5-14** Parameters

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

## Precautions

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

## Example

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

## 5.4.6 Viewing Table Statistics

### Function

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

### Syntax

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

### Keywords

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

## Parameters

**Table 5-15** Parameters

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

## Precautions

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

## Example

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

```
DESCRIBE student;
```

# 5.5 Modifying a Table

## 5.5.1 Adding a Column

### Function

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

### Syntax

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

### Keywords

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

## Parameters

**Table 5-16** Parameters

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

## Precautions

Do not run this SQL statement concurrently. Otherwise, columns may be overwritten.

## Example

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

## 5.5.2 Modifying Column Comments

### Function

You can modify the column comments of non-partitioned or partitioned tables.

### Syntax

```
ALTER TABLE [db_name.]table_name CHANGE COLUMN col_name col_name col_type COMMENT  
'col_comment';
```

### Keywords

- CHANGE COLUMN: Modify a column.
- COMMENT: column description

## Parameters

**Table 5-17** Parameters

Parameter	Mandatory	Description
db_name	No	Database name. Only letters, digits, and underscores (_) are allowed. The name cannot contain only digits or start with an underscore (_).
table_name	Yes	Table name
col_name	Yes	Column name. The value must be the name of an existing column.
col_type	Yes	Column data type specified when the table is created, which cannot be modified.
col_comment	Yes	Column comment after modification. The comment can contain a maximum of 1024 bytes.

## Example

Change the comment of the **c1** column in the **t1** table to **the new comment**.

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

## 5.5.3 Enabling or Disabling Data Multi-Versioning (Deprecated, Not Recommended)

### Function

The multi-versioning feature is to be deprecated and is not recommended. Recommended functions: [Hudi multi-versioning cleanup](#) and [Hudi archive](#).

DLI controls multiple versions of backup data for restoration. After the multiversion function is enabled, the system automatically backs up table data when you delete or modify the data using **insert overwrite** or **truncate**, and retains the data for a certain period. You can quickly restore data within the retention period. For details about the syntax related to the multiversion function, see [Backing Up and Restoring Multi-Versioning Data \(Deprecated, Not Recommended\)](#).

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

### Syntax

- Enable the multiversion function.  

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

- Disable the multiversion function.

```
ALTER TABLE [db_name.]table_name  
UNSET TBLPROPERTIES ("dli.multi.version.enable");
```

After multiversion is enabled, data of different versions is automatically stored in the OBS storage directory when **insert overwrite** or **truncate** is executed.

After multiversion is disabled, run the following statement to restore the multiversion backup data directory:

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

## Keywords

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

## Parameters

Table 5-18 Parameters

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

## Precautions

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

## Example

- Modify the **test\_table** table to enable multiversion.

```
ALTER TABLE test_table  
SET TBLPROPERTIES ("dli.multi.version.enable"="true");
```

- Modify the **test\_table** table to disable multiversion.

```
ALTER TABLE test_table  
UNSET TBLPROPERTIES ("dli.multi.version.enable");
```

Restore the multiversion backup data directory.

```
RESTORE TABLE test_table TO initial layout;
```

## 5.6 Partition-related Syntax

## 5.6.1 Adding Partition Data (Only OBS Tables Supported)

### Function

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

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

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

### Syntax

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

### Keywords

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

### Parameters

Table 5-19 Parameters

Parameter	Description
table_name	Table name
partition_specs	Partition fields
obs_path	OBS path

### Precautions

- When you add a partition to a table, the table and the partition column (specified by **PARTITIONED BY** during table creation) must exist, and the partition to be added cannot be added repeatedly. Otherwise, an error is reported. You can use **IF NOT EXISTS** to avoid errors if the partition does not exist.
- If tables are partitioned by multiple fields, you need to specify all partitioning fields in any sequence when adding partitions.

- By default, parameters in **partition\_specs** contain parentheses (). For example: **PARTITION (dt='2009-09-09',city='xxx')**.
- If you need to specify an OBS path when adding a partition, the OBS path must exist. Otherwise, an error occurs.
- To add multiple partitions, you need to use spaces to separate each set of **LOCATION 'obs\_path'** in the **PARTITION partition\_specs**. The following is an example:

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

- If the path specified in the new partition contains subdirectories (or nested subdirectories), all file types and content in the subdirectories are considered partition records.

Ensure that all file types and file content in the partition directory are the same as those in the table. Otherwise, an error is reported.

You can set **multiLevelDirEnable** to **true** in the **OPTIONS** statement to query the content in the subdirectory. The default value is **false** (Note that this configuration item is a table attribute, exercise caution when performing this operation. Hive tables do not support this configuration item.)

## Example

- The following example shows you how to add partition data when the OBS table is partitioned by a single column.
  - a. Use the DataSource syntax to create an OBS table, and partition the table by column **external\_data**. The partition data is stored in **obs://bucketName/datapath**.

```
create table testobstable(id varchar(128), external_data varchar(16)) using JSON OPTIONS  
(path 'obs://bucketName/datapath') PARTITIONED by (external_data);
```
  - b. Copy the partition directory to **obs://bucketName/datapath**. In this example, copy all files in the partition column **external\_data=22** to **obs://bucketName/datapath**.
  - c. Run the following command to add partition data:

```
ALTER TABLE testobstable ADD  
PARTITION (external_data='22')  
LOCATION 'obs://bucketName/datapath/external_data=22';
```
  - d. After the partition data is added successfully, you can perform operations such as data query based on the partition column.

```
select * from testobstable where external_data='22';
```
- The following example shows you how to add partition data when the OBS table is partitioned by multiple columns.
  - a. Use the DataSource syntax to create an OBS table, and partition the table by columns **external\_data** and **dt**. The partition data is stored in **obs://bucketName/datapath**.

```
create table testobstable(  
id varchar(128),  
external_data varchar(16),  
dt varchar(16)  
) using JSON OPTIONS (path 'obs://bucketName/datapath') PARTITIONED by (external_data,  
dt);
```
  - b. Copy the partition directories to **obs://bucketName/datapath**. In this example, copy files in **external\_data=22** and its subdirectory **dt=2021-07-27** to **obs://bucketName/datapath**.

- c. Run the following command to add partition data:

```
ALTER TABLE
  testobstable
ADD
  PARTITION (external_data = '22', dt = '2021-07-27') LOCATION 'obs://bucketName/datapath/
external_data=22/dt=2021-07-27';
```

- d. After the partition data is added successfully, you can perform operations such as data query based on the partition columns.

```
select * from testobstable where external_data = '22';
select * from testobstable where external_data = '22' and dt='2021-07-27';
```

## 5.6.2 Renaming a Partition (Only OBS Tables Supported)

### Function

This statement is used to rename partitions.

### Syntax

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

### Keywords

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

### Parameters

Table 5-20 Parameters

Parameter	Description
table_name	Table name
partition_specs	Partition fields

### Precautions

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

### Example

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

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

## 5.6.3 Deleting a Partition

### Function

This statement is used to delete one or more partitions from a partitioned table.

Partitioned tables are classified into OBS tables and DLI tables. You can delete one or more partitions from a DLI or OBS partitioned table based on specified conditions. OBS tables also support deleting partitions by specifying filter criteria. For details, see [Deleting Partitions by Specifying Filter Criteria \(Only Supported on OBS Tables\)](#).

### Precautions

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

### Syntax

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

### Keywords

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

### Parameters

**Table 5-21** Parameters

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

Parameter	Description
partition_specs	Partition information, in the format of "key=value", where <b>key</b> indicates the partition field and <b>value</b> indicates the partition value. In a table partitioned using multiple fields, if you specify all the fields of a partition name, only the partition is deleted; if you specify only some fields of a partition name, all matching partitions will be deleted. By default, parameters in <b>partition_specs</b> contain parentheses (), for example, <b>PARTITION (facultyNo=20, classNo=103);</b>

## Example

To help you understand how to use this statement, this section provides an example of deleting a partition from the source data.

**Step 1** Use the DataSource syntax to create an OBS partitioned table.

An OBS partitioned table named **student** is created, which contains the student ID (**id**), student name (**name**), student faculty number (**facultyNo**), and student class number (**classNo**) and uses **facultyNo** and **classNo** for partitioning.

```
create table if not exists student (  
id int,  
name STRING,  
facultyNo int,  
classNo INT)  
using csv  
options (path 'obs://bucketName/filePath')  
partitioned by (facultyNo, classNo);
```

**Step 2** Insert partition data into the table.

You can insert the following data:

```
INSERT into student  
partition (facultyNo = 10, classNo = 101)  
values (1010101, "student01"), (1010102, "student02");  
  
INSERT into student  
partition (facultyNo = 10, classNo = 102)  
values (1010203, "student03"), (1010204, "student04");  
  
INSERT into student  
partition (facultyNo = 20, classNo = 101)  
values (2010105, "student05"), (2010106, "student06");  
  
INSERT into student  
partition (facultyNo = 20, classNo = 102)  
values (2010207, "student07"), (2010208, "student08");  
  
INSERT into student  
partition (facultyNo = 20, classNo = 103)  
values (2010309, "student09"), (2010310, "student10");  
  
INSERT into student  
partition (facultyNo = 30, classNo = 101)  
values (3010111, "student11"), (3010112, "student12");  
  
INSERT into student  
partition (facultyNo = 30, classNo = 102)  
values (3010213, "student13"), (3010214, "student14");
```

**Step 3** View the partitions.

You can view all partitions in the table.

The example code is as follows:

**SHOW partitions student;**

**Table 5-22** Example table data

facultyNo	classNo
facultyNo=10	classNo=101
facultyNo=10	classNo=102
facultyNo=20	classNo=101
facultyNo=20	classNo=102
facultyNo=20	classNo=103
facultyNo=30	classNo=101
facultyNo=30	classNo=102

**Step 4** Delete a partition.

- **Example 1: deleting a partition by specifying multiple filter criteria**

In this example, the partition whose **facultyNo** is **20** and **classNo** is **103** is deleted.

 **NOTE**

For details about how to delete a partition by specifying filter criteria, see [Deleting Partitions by Specifying Filter Criteria \(Only Supported on OBS Tables\)](#).

The example code is as follows:

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

Use the method described in step 3 to check the partitions in the table. You can see that the partition has been deleted.

```
SHOW partitions student;
```

- **Example 2: deleting a partition by specifying a single filter criterion**

In this example, the partitions whose **facultyNo** is **30** is deleted. During data insertion, there are two partitions whose **facultyNo** is **30**.

 **NOTE**

For details about how to delete a partition by specifying filter criteria, see [Deleting Partitions by Specifying Filter Criteria \(Only Supported on OBS Tables\)](#).

The example code is as follows:

```
ALTER TABLE student  
DROP IF EXISTS  
PARTITION (facultyNo = 30);
```

Execution result:

**Table 5-23** Example table data

facultyNo	classNo
facultyNo=10	classNo=101
facultyNo=10	classNo=102
facultyNo=20	classNo=101
facultyNo=20	classNo=102
facultyNo=20	classNo=103

----End

## 5.6.4 Deleting Partitions by Specifying Filter Criteria (Only Supported on OBS Tables)

### Function

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

### Precautions

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

### Syntax

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

### Keywords

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

## Parameters

**Table 5-24** Parameters

Parameter	Description
db_name	Database name that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_).
table_name	Table name of a database that contains letters, digits, and underscores (_). It cannot contain only digits or start with an underscore (_). The matching rule is <code>^(?!_)(?![0-9]+\$)[A-Za-z0-9_\$]*\$</code> . If special characters are required, use single quotation marks (') to enclose them. <b>This statement is used for OBS table operations.</b>
partition_filter condition	Condition used to search partitions to be deleted. The format is as follows: <i>Partition column name</i> <b>Operator</b> <i>Value to compare</i> Example: <code>start_date &lt; '201911'</code> <ul style="list-style-type: none"><li>• Example 1: <code>&lt;partition_filtercondition1&gt; AND OR &lt;partition_filtercondition2&gt;</code> Example: <code>start_date &lt; '201911' OR start_date &gt;= '202006'</code></li><li>• Example 2: <code>(&lt;partition_filtercondition1&gt;)[,partitions (&lt;partition_filtercondition2&gt;), ...]</code> Example: <code>(start_date &lt;&gt; '202007'), partitions(start_date &lt; '201912')</code></li></ul>

## Example

To help you understand how to use this statement, this section provides an example of deleting a partition from the source data.

**Step 1** Use the DataSource syntax to create an OBS partitioned table.

An OBS partitioned table named **student** is created, which contains the student ID (**id**), student name (**name**), student faculty number (**facultyNo**), and student class number (**classNo**) and uses **facultyNo** and **classNo** for partitioning.

```
create table if not exists student (  
id int,  
name STRING,  
facultyNo int,  
classNo INT)  
using csv  
options (path 'path 'obs://bucketName/filePath')  
partitioned by (facultyNo, classNo);
```

**Step 2** Insert partition data into the table.

You can insert the following data:

```
INSERT into student  
partition (facultyNo = 10, classNo = 101)
```

```

values (1010101, "student01"), (1010102, "student02");

INSERT into student
partition (facultyNo = 10, classNo = 102)
values (1010203, "student03"), (1010204, "student04");

INSERT into student
partition (facultyNo = 20, classNo = 101)
values (2010105, "student05"), (2010106, "student06");

INSERT into student
partition (facultyNo = 20, classNo = 102)
values (2010207, "student07"), (2010208, "student08");

INSERT into student
partition (facultyNo = 20, classNo = 103)
values (2010309, "student09"), (2010310, "student10");

INSERT into student
partition (facultyNo = 30, classNo = 101)
values (3010111, "student11"), (3010112, "student12");

INSERT into student
partition (facultyNo = 30, classNo = 102)
values (3010213, "student13"), (3010214, "student14");

```

**Step 3** View the partitions.

You can view all partitions in the table.

The example code is as follows:

```
SHOW partitions student;
```

**Table 5-25** Example table data

facultyNo	classNo
facultyNo=10	classNo=101
facultyNo=10	classNo=102
facultyNo=20	classNo=101
facultyNo=20	classNo=102
facultyNo=20	classNo=103
facultyNo=30	classNo=101
facultyNo=30	classNo=102

**Step 4** Delete a partition.

 **NOTE**

This step describes how to delete a partition by specifying filter criteria. If you want to delete a partition without specifying filter criteria, see [Deleting a Partition](#).

This example cannot be used together with that in [Deleting a Partition](#). Distinguish the keyword **partitions** in this example from the keyword **partition** in the example in [Deleting a Partition](#).

- **Example 1: deleting partitions by specifying filter criteria (only supported on OBS tables), and using the AND statement to delete partitions**

**Table 5-26** Data before execution

facultyNo	classNo
facultyNo=10	classNo=101
facultyNo=10	classNo=102
facultyNo=20	classNo=101
facultyNo=20	classNo=102

Run the following statements to delete the partitions whose **facultyNo** is **20** and **classNo** is **102**:

```
ALTER TABLE student  
DROP IF EXISTS  
PARTITIONS (facultyNo = 20 AND classNo = 102);
```

You can see that the statement deletes the partitions that meet both the criteria.

**Table 5-27** Data after execution

facultyNo	classNo
facultyNo=10	classNo=101
facultyNo=10	classNo=102
facultyNo=20	classNo=101

- **Example 2: deleting partitions by specifying filter criteria (only supported on OBS tables), and using the OR statement to delete partitions**

**Table 5-28** Data before execution

facultyNo	classNo
facultyNo=10	classNo=101
facultyNo=10	classNo=102
facultyNo=20	classNo=101
facultyNo=20	classNo=102

Run the following statements to delete the partitions whose **facultyNo** is **10** or **classNo** is **101**:

```
ALTER TABLE student  
DROP IF EXISTS  
PARTITIONS (facultyNo = 10),  
PARTITIONS (classNo = 101);
```

Execution result:

**Table 5-29** Data after execution

facultyNo	classNo
facultyNo=20	classNo=102

Under the selected deletion criteria, the first record in the partition meets both **facultyNo** and **classNo**, the second record meets **facultyNo**, and the third record meets **classNo**.

As a result, only one partition row remains after executing the partition deletion statement.

According to method 1, the foregoing execution statement may also be written as:

```
ALTER TABLE student
DROP IF EXISTS
PARTITIONS (facultyNo = 10 OR classNo = 101);
```

- **Example 3: deleting partitions by specifying filter criteria (only supported on OBS tables), and using relational operator statements to delete specified partitions**

**Table 5-30** Data before execution

facultyNo	classNo
facultyNo=10	classNo=101
facultyNo=10	classNo=102
facultyNo=20	classNo=101
facultyNo=20	classNo=102
facultyNo=20	classNo=103

Run the following statements to delete partitions whose **classNo** is greater than 100 and less than 102:

```
ALTER TABLE student
DROP IF EXISTS
PARTITIONS (classNo BETWEEN 100 AND 102);
```

Execution result:

**Table 5-31** Data before execution

facultyNo	classNo
facultyNo=20	classNo=103

----End

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

### Function

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

### Syntax

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

### Keywords

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

### Parameters

Table 5-32 Parameters

Parameter	Description
table_name	Table name
partition_specs	Partition fields
obs_path	OBS path

### Precautions

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

### Example

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

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

## 5.6.6 Updating Partitioned Table Data (Only OBS Tables Supported)

### Function

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

### Syntax

```
MSCK REPAIR TABLE table_name;
```

Or

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

### Keywords

- PARTITIONS: partition information
- SERDEPROPERTIES: Serde attribute

### Parameters

Table 5-33 Parameters

Parameter	Description
table_name	Table name
partition_specs	Partition fields
obs_path	OBS path

### Precautions

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

### Example

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

```
MSCK REPAIR TABLE ptable;
```

Or

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

## 5.6.7 Updating Table Metadata with REFRESH TABLE

### Function

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

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

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

### Syntax

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

### Keywords

None

### Parameters

**Table 5-34** Parameters

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

### Precautions

None

### Example

Update metadata of the **test** table.

```
REFRESH TABLE test;
```

## 5.7 Backing Up and Restoring Multi-Versioning Data (Deprecated, Not Recommended)

### 5.7.1 Setting the Retention Period of Multi-Versioning Backup Data (Deprecated, Not Recommended)

#### Function

The multi-versioning feature is to be deprecated and is not recommended. Recommended functions: [Hudi multi-versioning cleanup](#) and [Hudi archive](#).

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

For details about the syntax for enabling or disabling the multiversion function, see [Enabling or Disabling Data Multi-Versioning \(Deprecated, Not Recommended\)](#).

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

#### Syntax

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

#### Keywords

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

#### Parameters

Table 5-35 Parameters

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

## Precautions

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

## Example

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

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

## 5.7.2 Viewing Multi-Versioning Backup Data (Deprecated, Not Recommended)

### Function

The multi-versioning feature is to be deprecated and is not recommended. Recommended functions: [Hudi multi-versioning cleanup](#) and [Hudi archive](#).

After the multiversion function is enabled, you can run the **SHOW HISTORY** command to view the backup data of a table. For details about the syntax for enabling or disabling the multiversion function, see [Enabling or Disabling Data Multi-Versioning \(Deprecated, Not Recommended\)](#).

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

### Syntax

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

### Keywords

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

### Parameters

Table 5-36 Parameters

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

Parameter	Description
value	Value corresponding to the partition column name

## Precautions

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

## Example

- View multiversion backup data of the **test\_table** table.  
`SHOW HISTORY FOR TABLE test_table;`
- View multiversion backup data of the **dt** partition in the **test\_table** partitioned table.  
`SHOW HISTORY FOR TABLE test_table PARTITION (dt='2021-07-27');`

## 5.7.3 Restoring Multi-Versioning Backup Data (Deprecated, Not Recommended)

### Function

The multi-versioning feature is to be deprecated and is not recommended. Recommended functions: [Hudi multi-versioning cleanup](#) and [Hudi archive](#).

After the multiversion function is enabled, you can run the **RESTORE TABLE** statement to restore a table or partition of a specified version. For details about the syntax for enabling or disabling the multiversion function, see [Enabling or Disabling Data Multi-Versioning \(Deprecated, Not Recommended\)](#).

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

### Syntax

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

### Keywords

- **RESTORE TABLE**: Used to restore backup data
- **PARTITION**: Used to specify the partition column
- **TO VERSION**: Used to specify the version number You can run the **SHOW HISTORY** command to obtain the version number. For details, see [Viewing Multi-Versioning Backup Data \(Deprecated, Not Recommended\)](#).

## Parameters

**Table 5-37** Parameters

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
table_name	Table name
column	Partition column name
value	Value corresponding to the partition column name
version_id	Target version of the backup data to be restored You can run the <b>SHOW HISTORY</b> command to obtain the version number. For details, see <a href="#">Viewing Multi-Versioning Backup Data (Deprecated, Not Recommended)</a> .

## Precautions

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

## Example

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

## 5.7.4 Configuring the Recycle Bin for Expired Multi-Versioning Data (Deprecated, Not Recommended)

### Function

The multi-versioning feature is to be deprecated and is not recommended. Recommended functions: [Hudi multi-versioning cleanup](#) and [Hudi archive](#).

After the multiversion function is enabled, expired backup data will be directly deleted by the system when the **insert overwrite** or **truncate** statement is executed. You can configure the trash bin of the OBS parallel file system to accelerate the deletion of expired backup data. To enable the trash bin, add **dli.multi.version.trash.dir** to the table properties. For details about the syntax for enabling or disabling the multiversion function, see [Enabling or Disabling Data Multi-Versioning \(Deprecated, Not Recommended\)](#).

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

## Syntax

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

## Keywords

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

## Parameters

Table 5-38 Parameters

Parameter	Description
db_name	Database name, which consists of letters, digits, and underscores (_). The value cannot contain only digits or start with a digit or underscore (_).
table_name	Table name
OBS bucket for expired multiversion backup data	A directory in the bucket where the current OBS table locates. You can change the directory path as needed. For example, if the current OBS table directory is <b>obs://bucketName/filePath</b> and a <b>Trash</b> directory has been created in the OBS table directory, you can set the trash bin directory to <b>obs://bucketName/filePath/Trash</b> .

## Precautions

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

**Figure 5-1** Creating a lifecycle rule

**Create Lifecycle Rule** [Learn more](#)

Basic Information

Status  Enable  Disable

Rule Name

Applies To  Object name prefix  Entire file system

Prefix  ?

Current Version

Transition to Infrequent Access  Do not configure  Configure now

Transition to Archive  Do not configure  Configure now

Delete Files  Do not configure  Configure now

**OK** Cancel

## Example

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

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

## 5.7.5 Clearing Multi-Versioning Data (Deprecated, Not Recommended)

### Function

The multi-versioning feature is to be deprecated and is not recommended. Recommended functions: [Hudi multi-versioning cleanup](#) and [Hudi archive](#).

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

### Syntax

Delete multiversion backup data out of the retention period.  
`clear history for table` [db\_name.]table\_name **older\_than** 'timestamp';

### Keywords

- `clear history for table`: Used to delete multiversion backup data

- `older_than`: Used to specify the time range for deleting multiversion backup data

## Parameters

**Table 5-39** Parameters

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

## Precautions

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

## Example

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

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

## 5.8 Table Lifecycle Management

### 5.8.1 Specifying the Lifecycle of a Table When Creating the Table

#### Function

DLI provides table lifecycle management to allow you to specify the lifecycle of a table when creating the table. DLI determines whether to reclaim a table based on the table's last modification time and its lifecycle. By setting the lifecycle of a table, you can better manage a large number of tables, automatically delete data tables that are no longer used for a long time, and simplify the process of reclaiming data tables. Additionally, data restoration settings are supported to prevent data loss caused by misoperations.

## Table Reclamation Rules

- When creating a table, use **TBLPROPERTIES** to specify the lifecycle of the table.
  - **Non-partitioned table**  
If the table is not a partitioned table, the system determines whether to reclaim the table after the lifecycle time based on the last modification time of the table.
  - **Partitioned table**  
If the table is a partitioned table, the system determines whether the partition needs to be reclaimed based on the last modification time (**LAST\_ACCESS\_TIME**) of the partition. After the last partition of a partitioned table is reclaimed, the table is not deleted.  
Only table-level lifecycle management is supported for partitioned tables.
- Lifecycle reclamation starts at a specified time every day to scan all partitions. Lifecycle reclamation starts at a specified time every day. Reclamation only occurs if the last modification time of the table data (**LAST\_ACCESS\_TIME**) detected when scanning complete partitions exceeds the time specified by the lifecycle.  
Assume that the lifecycle of a partitioned table is one day and the last modification time of the partitioned data is 15:00 on May 20, 2023. If the table is scanned before 15:00 on May 20, 2023 (less than one day), the partitions in the table will not be reclaimed. If the last data modification time (**LAST\_ACCESS\_TIME**) of a table partition exceeds the time specified by the lifecycle during reclamation scan on May 20, 2023, the partition will be reclaimed.
- The lifecycle function periodically reclaims tables or partitions, which are reclaimed irregularly every day depending on the level of busyness of the service. It cannot ensure that a table or partition will be reclaimed immediately after its lifecycle expires.
- After a table is deleted, all properties of the table, including the lifecycle, will be deleted. After a table with the same name is created again, the lifecycle of the table will be determined by the new property.

## Notes and Constraints

- The table lifecycle function is currently in the open beta test (OBT) phase. If necessary, contact customer service to whitelist it.
- You are advised to configure the **qli\_data\_clean\_agency** agency before using the lifecycle feature.  
For details about agency permission policies, refer to [Agency Permission Policies in Common DLI Scenarios](#).  
If the **qli\_data\_clean\_agency** agency is not configured, the system reads the previous-generation DLI system agency **qli\_admin\_agency** by default. However, if **qli\_admin\_agency** is not configured for your account, the current table lifecycle feature cannot be used.  
You can check whether you have the **qli\_admin\_agency** agency in IAM agencies.
- The table lifecycle function currently only supports creating tables and versioning tables using Hive and Datasource syntax.

- The unit of the lifecycle is in days. The value should be a positive integer.
- The lifecycle can be set only at the table level. The lifecycle specified for a partitioned table applies to all partitions of the table.
- After the lifecycle is set, DLI and OBS tables will support data backup. The backup directory for OBS tables needs to be set manually. The backup directory must be in the parallel file system and in the same bucket as the original table directory. It cannot have the same directory or subdirectory name as the original table.

## Syntax

- **Creating a DLI table using the Datasource syntax**

```
CREATE TABLE table_name(name string, id int)
USING parquet
TBLPROPERTIES( "dli.lifecycle.days"=1 );
```

- **Creating a DLI table using the Hive syntax**

```
CREATE TABLE table_name(name string, id int)
stored as parquet
TBLPROPERTIES( "dli.lifecycle.days"=1 );
```

- **Creating an OBS table using the Datasource syntax**

```
CREATE TABLE table_name(name string, id int)
USING parquet
OPTIONS (path "obs://dli-test/table_name")
TBLPROPERTIES( "dli.lifecycle.days"=1, "external.table.purge"='true', "dli.lifecycle.trash.dir"='obs://dli-test/Lifecycle-Trash' );
```

- **Creating an OBS table using the Hive syntax**

```
CREATE TABLE table_name(name string, id int)
STORED AS parquet
LOCATION 'obs://dli-test/table_name'
TBLPROPERTIES( "dli.lifecycle.days"=1, "external.table.purge"='true', "dli.lifecycle.trash.dir"='obs://dli-test/Lifecycle-Trash' );
```

## Keywords

- **TBLPROPERTIES:** Table properties, which can be used to extend the lifecycle of a table.
- **OPTIONS:** path of the new table, which is applicable to OBS tables created using the Datasource syntax.
- **LOCATION:** path of the new table, which is applicable to OBS tables created using the Hive syntax.

## Parameters

**Table 5-40** Parameters

Parameter	Mandato ry	Description
table_name	Yes	Name of the table whose lifecycle needs to be set
dli.lifecycle.days	Yes	Lifecycle duration. The value must be a positive integer, in days.

Parameter	Mandatory	Description
external.table.purge	No	<p>This parameter is available only for OBS tables.</p> <p>Whether to clear data in the path when deleting a table or partition. The data is not cleared by default.</p> <p>When this parameter is set to <b>true</b>:</p> <ul style="list-style-type: none"> <li>After a file is deleted from a non-partitioned OBS table, the table directory is also deleted.</li> <li>The custom partition data in the partitioned OBS table is also deleted.</li> </ul>
dli.lifecycle.trash.dir	No	<p>This parameter is available only for OBS tables.</p> <p>When <b>external.table.purge</b> is set to <b>true</b>, the backup directory will be deleted. By default, backup data is deleted seven days later.</p>

## Example

- **Create the test\_datasource\_lifecycle table using the Datasource syntax. The lifecycle is set to 100 days.**

```
CREATE TABLE test_datasource_lifecycle(id int)
USING parquet
TBLPROPERTIES( "dli.lifecycle.days"=100);
```
- **Create the test\_hive\_lifecycle table using the Hive syntax. The lifecycle is set to 100 days.**

```
CREATE TABLE test_hive_lifecycle(id int)
stored as parquet
TBLPROPERTIES( "dli.lifecycle.days"=100);
```
- **Create the test\_datasource\_lifecycle\_obs table using the Datasource syntax. The lifecycle is set to 100 days. When the lifecycle expires, data is deleted by default and backed up to the obs://dli-test/ directory.**

```
CREATE TABLE test_datasource_lifecycle_obs(name string, id int)
USING parquet
OPTIONS (path "obs://dli-test/xxx")
TBLPROPERTIES( "dli.lifecycle.days"=100, "external.table.purge"='true', "dli.lifecycle.trash.dir"='obs://dli-test/Lifecycle-Trash' );
```
- **Create the test\_hive\_lifecycle\_obs table using the Hive syntax. The lifecycle is set to 100 days. When the lifecycle expires, data is deleted by default and backed up to the obs://dli-test/ directory.**

```
CREATE TABLE test_hive_lifecycle_obs(name string, id int)
STORED AS parquet
LOCATION 'obs://dli-test/xxx'
TBLPROPERTIES( "dli.lifecycle.days"=100, "external.table.purge"='true', "dli.lifecycle.trash.dir"='obs://dli-test/Lifecycle-Trash' );
```

## 5.8.2 Modifying the Lifecycle of a Table

### Function

This section describes how to modify the lifecycle of an existing partitioned or non-partitioned table.

When the lifecycle function is enabled for the first time, the system scans tables or partitions, scans table data files in the path, and updates **LAST\_ACCESS\_TIME** of tables or partitions. The time required depends on the number of partitions and files.

### Notes and Constraints

- The table lifecycle function is currently in the OBT phase. If necessary, contact customer service to whitelist it.
- The table lifecycle function currently only supports creating tables and versioning tables using Hive and Datasource syntax.
- The unit of the lifecycle is in days. The value should be a positive integer.
- The lifecycle can be set only at the table level. The lifecycle specified for a partitioned table applies to all partitions of the table.

### Syntax

```
ALTER TABLE table_name  
SET TBLPROPERTIES("dli.lifecycle.days"='N')
```

### Keywords

**TBLPROPERTIES**: Table properties, which can be used to extend the lifecycle of a table.

### Parameters

Table 5-41 Parameters

Parameter	Mandatory	Description
table_name	Yes	Name of the table whose lifecycle needs to be modified
dli.lifecycle.days	Yes	Lifecycle duration after the modification. The value must be a positive integer, in days.

### Example

- Example 1: Enable the lifecycle function for the **test\_lifecycle\_exists** table and set the lifecycle to 50 days.

```
alter table test_lifecycle_exists  
SET TBLPROPERTIES("dli.lifecycle.days"='50');
```

- Example 2: Enable the lifecycle function for an existing partitioned or non-partitioned table for which lifecycle is not set, for example, for the **test\_lifecycle\_exists** table, and set the lifecycle to 50 days.

```
alter table test_lifecycle_exists
SET TBLPROPERTIES(
  "dli.lifecycle.days"='50',
  "dli.table.lifecycle.status"='enable'
);
```

## 5.8.3 Disabling or Restoring the Lifecycle of a Table

### Function

This section describes how to disable or restore the lifecycle of a specified table or partition.

You can disable or restore the lifecycle of a table in either of the following scenarios:

1. If the lifecycle function has been enabled for a table or partitioned table, the system allows you to disable or restore the lifecycle of the table by changing the value of **dli.table.lifecycle.status**.
2. If the lifecycle function is not enabled for a table or partitioned table, the system will add the **dli.table.lifecycle.status** property to allow you to disable or restore the lifecycle function of the table.

### Notes and Constraints

- The table lifecycle function is currently in the open beta test (OBT) phase. If necessary, contact customer service to whitelist it.
- The table lifecycle function currently only supports creating tables and versioning tables using Hive and Datasource syntax.
- The unit of the lifecycle is in days. The value should be a positive integer.
- The lifecycle can be set only at the table level. The lifecycle specified for a partitioned table applies to all partitions of the table.

### Syntax

- This syntax can be used to disable or restore the lifecycle of a table at the table level.  
`ALTER TABLE table_name SET TBLPROPERTIES("dli.table.lifecycle.status"={enable|disable});`
- This syntax can be used to disable or restore the lifecycle of a specified table at the table or partition table level.  
`ALTER TABLE table_name [pt_spec] LIFECYCLE {enable|disable};`

### Keywords

**TBLPROPERTIES:** Table properties, which can be used to extend the lifecycle of a table.

## Parameters

**Table 5-42** Parameters

Parameter	Mandatory	Description
table_name	Yes	Name of the table whose lifecycle is to be disabled or restored
pt_spec	No	Partition information of the table whose lifecycle is to be disabled or restored. The format is <b>partition_col1=col1_value1, partition_col2=col2_value1....</b> For a table with multi-level partitions, all partition values must be specified.
enable	No	Restores the lifecycle function of a table or a specified partition. <ul style="list-style-type: none"> <li>• The table and its partitions participate in lifecycle reclamation again. By default, the lifecycle configuration of the current table and its partitions is used.</li> <li>• Before enabling the table lifecycle function, it is recommended to modify the lifecycle configuration of the table and its partitions. This will help prevent any accidental data reclamation caused by the previous configuration once the table lifecycle function is enabled.</li> </ul>
disable	No	Disables the lifecycle function of a table or a specified partition. <ul style="list-style-type: none"> <li>• Prevents a table and all its partitions from being reclaimed by the lifecycle. It takes priority over restoring the lifecycle of a table and its partitions. That is, when the lifecycle function of a table or a specified partition is disabled, the partition information of the table whose lifecycle is to be disabled or restored is invalid.</li> <li>• After the lifecycle function of a table is disabled, the lifecycle configuration of the table and the enable and disable flags of its partitions are retained.</li> <li>• After the lifecycle function of a table is disabled, the lifecycle configuration of the table and partitioned table can still be modified.</li> </ul>

## Example

- Example 1: Disable the lifecycle function of the **test\_lifecycle** table.  
`alter table test_lifecycle SET TBLPROPERTIES("dli.table.lifecycle.status"='disable');`
- Example 2: Disable the lifecycle function for the partition whose time is **20230520** in the **test\_lifecycle** table.  
`alter table test_lifecycle partition (dt='20230520') LIFECYCLE 'disable';`

### NOTE

- After the lifecycle function of a partitioned table is disabled, the lifecycle function of all partitions within the table will also be disabled.

# 6 Data

## 6.1 Importing Data

### Function

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

### Syntax

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

### Keywords

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

### Parameters

**Table 6-1** Parameters

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

The following configuration options can be used during data import:

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

The configuration item is **OPTIONS** ('DATA\_TYPE' = 'CSV').

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

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

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

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

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

For CSV data, the following delimiters are supported:

- Tab character, for example, '**DELIMITER**'='\t'.
- Any binary character, for example, '**DELIMITER**'='\u0001(^A)'.
- Single quotation mark ('). A single quotation mark must be enclosed in double quotation marks (" "). For example, '**DELIMITER**'= ""'.
- **\001(^A)** and **\017(^Q)** are also supported, for example, '**DELIMITER**'='\001(^A)' and '**DELIMITER**'='\017(^Q)'.

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

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

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

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

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

The configuration item is **OPTIONS('HEADER'='true')**.

- **FILEHEADER**: If the source file does not contain any header, add a header to the **LOAD DATA** statement.

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

- **ESCAPECHAR**: Is used to perform strict verification of the escape character on CSV files. The default value is a slash (\\).

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

#### NOTE

Enter **ESCAPECHAR** in the CSV data. **ESCAPECHAR** must be enclosed in double quotation marks (" "). For example, "a\b".

- **MAXCOLUMNS:** This parameter is optional and specifies the maximum number of columns parsed by a CSV parser in a line.  
The configuration item is `OPTIONS('MAXCOLUMNS'='400')`.

**Table 6-2** MAXCOLUMNS

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

 **NOTE**

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

- **DATEFORMAT:** Specifies the date format of a column.  
`OPTIONS('DATEFORMAT'='dateFormat')`

 **NOTE**

- The default value is yyyy-MM-dd.
- The date format is specified by the date mode string of **Java**. For the Java strings describing date and time pattern, characters **A** to **Z** and **a** to **z** without single quotation marks (') are interpreted as pattern characters, which are used to represent date or time string elements. If the pattern character is quoted by single quotation marks ('), text matching rather than parsing is performed. For the definition of pattern characters in Java, see [Table 6-3](#).

**Table 6-3** Definition of characters involved in the date and time patterns

Character	Date or Time Element	Example
G	Epoch ID	AD
y	Year	1996; 96
M	Month	July; Jul; 07
w	Number of the week in a year	27 (the twenty-seventh week of the year)
W	Number of the week in a month	2 (the second week of the month)
D	Number of the day in a year	189 (the 189th day of the year)
d	Number of the day in a month	10 (the tenth day of the month)
u	Number of the day in a week	1 (Monday), ..., 7 (Sunday)

Character	Date or Time Element	Example
a	am/pm flag	pm (12:00-24:00)
H	Hour time (0-23)	2
h	Hour time (1-12)	12
m	Number of minutes	30
s	Number of seconds	55
S	Number of milliseconds	978
z	Time zone	Pacific Standard Time; PST; GMT-08:00

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

*OPTIONS('TIMESTAMPFORMAT'='timestampFormat')*

 **NOTE**

- Default value: yyyy-MM-dd HH:mm:ss.
- The timestamp format is specified by the Java time pattern string. For details, see [Table 6-3](#).
- **Mode:** Specifies the processing mode of error records while importing. The options are as follows: **PERMISSIVE**, **DROPMALFORMED**, and **FAILFAST**.

*OPTIONS('MODE'='permissive')*

 **NOTE**

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

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

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

 **NOTE**

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

## Precautions

- When importing or creating an OBS table, you must specify a folder as the directory. If a file is specified, data import may be failed.
- Only the raw data stored in the OBS path can be imported.

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

## Example

### NOTE

Before importing data, you must create a table. For details, see [Creating an OBS Table](#) or [Creating a DLI Table](#).

- To import a CSV file to a DLI table named **t**, run the following statement:

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

- To import a JSON file to a DLI table named **jsontb**, run the following statement:

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

## 6.2 Inserting Data

### Function

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

### Notes and Constraints

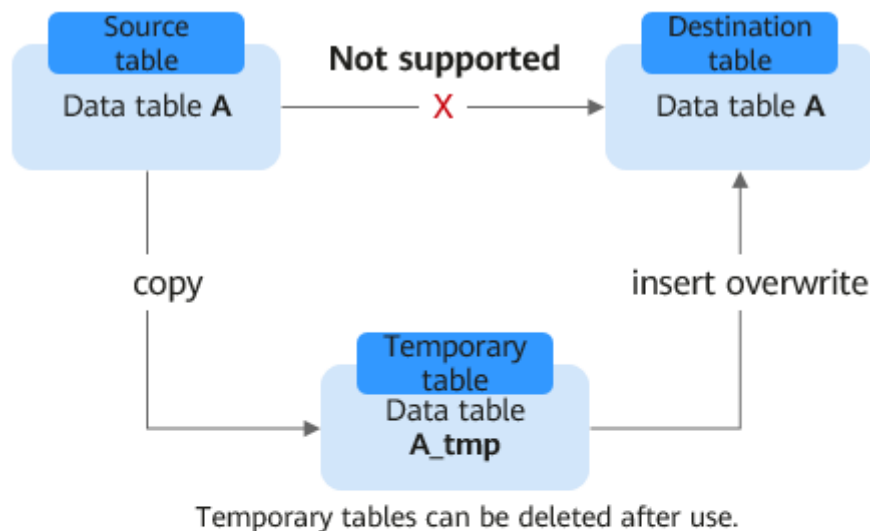
- The **insert overwrite** syntax does not apply to self-read and self-write scenarios within the same table (including both partitioned and non-partitioned tables). Directly executing **insert overwrite** on the original table may lead to risks of data loss or inconsistency.

To implement data operations in self-read and self-write scenarios, you are advised to use a temporary table to handle the data. See [Figure 6-1](#).

Self-read and self-write means that the destination table and the data source table are the same table. For example, suppose you want to extract information of students with **class\_no = 1** from the **student** table and overwrite the original table, the following statements represent typical operations in self-read and self-write scenarios:

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

**Figure 6-1** Alternative solution for self-read and self-write scenarios by running insert overwrite



- When using Hive and Datasource tables (excluding Hudi), executing data modification commands (such as **insert into** and **load data**) may result in data duplication or inconsistency if the data source does not support transactions and there is a system failure or queue restart.

To avoid this situation, you are advised to prioritize data sources that support transactions, such as Hudi data sources. This type of data source has Atomicity, Consistency, Isolation, Durability (ACID) capabilities, which helps ensure data consistency and accuracy.

To learn more, refer to [How Do I Handle Duplicate Records After Executing the INSERT INTO Statement?](#)

## Syntax

- Insert the SELECT query result into a table.

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

- Insert a data record into a table.

```
INSERT INTO [TABLE] [db_name.]table_name
[PARTITION part_spec] VALUES values_row [, values_row ...];
```

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

## Keywords

**Table 6-4** INSERT keywords

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

## Precautions

- The target DLI table must exist.
- If no partition needs to be specified for dynamic partitioning, place **part\_spec** in the SELECT statement as a common field.
- During creation of the target OBS table, only the folder path can be specified.
- The source table and the target table must have the same data types and column field quantity. Otherwise, data insertion fails.
- You are not advised to insert data concurrently into the same table as it may result in abnormal data insertion due to concurrency conflicts.
- The **INSERT INTO** statement is used to add the query result to the target table.
- The **INSERT OVERWRITE** statement is used to overwrite existing data in the source table.
- The **INSERT INTO** statement can be batch executed, but the **INSERT OVERWRITE** statement can be batch executed only when data of different partitioned tables is inserted to different static partitions.
- The **INSERT INTO** and **INSERT OVERWRITE** statements can be executed at the same time. However, the result is unknown.
- When you insert data of the source table to the target table, you cannot import or update data of the source table.
- The dynamic INSERT OVERWRITE statement of Hive partitioned tables can overwrite the involved partition data but cannot overwrite the entire table data.

- To overwrite data in a specified partition of the datasource table, set **dli.sql.dynamicPartitionOverwrite.enabled** to **true** and run the **insert overwrite** statement. The default value of **dli.sql.dynamicPartitionOverwrite.enabled** is **false**, indicating that data in the entire table is overwritten. The following is an example:

```
insert overwrite table tb1 partition(part1='v1', part2='v2') select * from ...
```

#### NOTE

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

- You can configure the **spark.sql.shuffle.partitions** parameter to set the number of files to be inserted into the OBS bucket in the non-DLI table. In addition, to avoid data skew, you can add **distribute by rand()** to the end of the INSERT statement to increase the number of concurrent jobs. The following is an example:

```
insert into table table_target select * from table_source distribute by cast(rand() * N as int);
```

## Example

#### NOTE

Before importing data, you must create a table. For details, see [Creating an OBS Table](#) or [Creating a DLI Table](#).

- Example 1: Insert the SELECT query result into a table.
  - Use the DataSource syntax to create a parquet partitioned table.

```
CREATE TABLE data_source_tab1 (col1 INT, p1 INT, p2 INT)
  USING PARQUET PARTITIONED BY (p1, p2);
```
  - Insert the query result to the partition (p1 = 3, p2 = 4).

```
INSERT INTO data_source_tab1 PARTITION (p1 = 3, p2 = 4)
  SELECT id FROM RANGE(1, 3);
```
  - Insert the new query result to the partition (p1 = 3, p2 = 4).

```
INSERT OVERWRITE TABLE data_source_tab1 PARTITION (p1 = 3, p2 = 4)
  SELECT id FROM RANGE(3, 5);
```
- Example 2: Insert a piece of data into a table.
  - Create a Parquet partitioned table with Hive format

```
CREATE TABLE hive_serde_tab1 (col1 INT, p1 INT, p2 INT)
  USING HIVE OPTIONS(fileFormat 'PARQUET') PARTITIONED BY (p1, p2);
```
  - Insert two data records into the partition (p1 = 3, p2 = 4).

```
INSERT INTO hive_serde_tab1 PARTITION (p1 = 3, p2 = 4)
  VALUES (1), (2);
```
  - Insert new data to the partition (p1 = 3, p2 = 4).

```
INSERT OVERWRITE TABLE hive_serde_tab1 PARTITION (p1 = 3, p2 = 4)
  VALUES (3), (4);
```

## How Do I Handle Duplicate Records After Executing the INSERT INTO Statement?

- **Symptom**

When using Hive and Datasource tables (excluding Hudi), executing data modification commands (such as **insert into** and **load data**) may result in data duplication or inconsistency if the data source does not support transactions and there is a system failure or queue restart.
- **Possible causes**

If queue resources are restarted in the data commit phase, data may have been restored to a formal directory. If an **insert into** statement is executed and a retry is triggered after a resource restart, there is a possibility that data will be repeatedly written.

- **Solution**
  - a. Hudi data sources that support ACID properties are recommended.
  - b. Use idempotent syntax such as **insert overwrite** instead of non-idempotent syntax such as **insert into** to insert data.
  - c. If it is strictly required that data cannot be duplicated, you are advised to perform deduplication on the table data after executing the **insert into** statement to prevent duplicate data.

## 6.3 Reusing Results of Subqueries

### Function

To improve query performance, caching the results of a subquery and reusing them in different parts of the query can be done to avoid redundant calculations of the same subquery during the execution of a SELECT statement.

### Syntax

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

### Keywords

**Table 6-5** Keywords

Parameter	Description
cte_name	Custom subquery name
table_name	Name of the table where subquery commands are executed

### Example

- Example 1: Define the **sales\_data** subquery based on the **sales** table, query all items with sales amounts greater than 100, and retrieve all data from the subquery.

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

- Example 2: Define two subqueries, **sales\_data1** and **sales\_data2**, based on the **sales** table, where **sales\_data1** retrieves all items with sales amounts greater than 100 and **sales\_data2** retrieves all items with sales amounts less than 20. Finally, merge the data from both subqueries.

```
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;
```

## 6.4 Clearing Data

### Function

This statement is used to delete data from the DLI or OBS table.

### Syntax

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

### Keywords

**Table 6-6** Keywords

Parameter	Description
tablename	Name of the target DLI or OBS table that runs the <b>Truncate</b> statement.
partcol1	Partition name of the DLI or OBS table to be deleted.

### Precautions

Only data in the DLI or OBS table can be deleted.

### Example

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

# 7 Exporting Query Results

## Function

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

## Syntax

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

## Keywords

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

## Parameter

**Table 7-1** INSERT OVERWRITE DIRECTORY parameters

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

### NOTE

If **file\_format** is set to **csv**, see [Table 5-3](#) for the OPTIONS parameters.

## Precautions

- You can configure the **spark.sql.shuffle.partitions** parameter to set the number of files to be inserted into the OBS bucket in the non-DLI table. In

addition, to avoid data skew, you can add **distribute by rand()** to the end of the INSERT statement to increase the number of concurrent jobs. The following is an example:

```
insert into table table_target select * from table_source distribute by cast(rand() * N as int);
```

- When the configuration item is **OPTIONS('DELIMITER=',')**, you can specify a separator. The default value is `,`.

For CSV data, the following delimiters are supported:

- Tab character, for example, **'DELIMITER='\t'**.
- You can specify a delimiter using Unicode encoding, for example: **'DELIMITER='\u0001'**.
- Single quotation mark (`'`). A single quotation mark must be enclosed in double quotation marks (`" "`). For example, **'DELIMITER'= ""**.

## Example

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

# 8 Datasource Connections

---

## 8.1 Creating a Datasource Connection with an HBase Table

### 8.1.1 Creating a DLI Table and Associating It with HBase

#### Function

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

#### NOTE

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

For details about datasource authentication, see [Introduction to Datasource Authentication](#).

#### Prerequisites

- Before creating a DLI table and associating it with HBase, you need to create a datasource connection. For operations on the management console, see [Enhanced Datasource Connections](#).
- Ensure that the `/etc/hosts` information of the master node in the MRS cluster is added to the host file of the DLI queue.

For details about how to add an IP-domain mapping, see [Enhanced Datasource Connection](#) in the *Data Lake Insight User Guide*.

- The syntax is not supported for security clusters.

#### Syntax

- Single row key  

```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME (  
  ATTR1 TYPE,
```

```
ATTR2 TYPE,
ATTR3 TYPE)
USING [CLOUDTABLE | HBASE] OPTIONS (
'ZKHost'='xx',
'TableName'='TABLE_IN_HBASE',
'RowKey'='ATTR1',
'Cols'='ATTR2:CF1.C1, ATTR3:CF1.C2');
```

- Combined row key

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

## Keywords

**Table 8-1** CREATE TABLE keywords

Parameter	Description
USING [CLOUDTABLE   HBASE]	Specify the HBase datasource to CLOUDTABLE or HBASE. The value is case insensitive.
ZKHost	<p>ZooKeeper IP address of the HBase cluster.</p> <p>Create a datasource connection before you can obtain this IP address. For operations on the management console, see <a href="#">Enhanced Datasource Connections</a>.</p> <ul style="list-style-type: none"> <li>• Access the CloudTable cluster and enter the ZooKeeper IP address (internal network).</li> <li>• To access the MRS cluster, enter the IP address of the node where the ZooKeeper is located and the external port number of the ZooKeeper. The format is <b>ZK_IP1:ZK_PORT1,ZK_IP2:ZK_PORT2</b>.</li> </ul> <p><b>NOTE</b> To access the MRS cluster, you can only create the enhanced datasource connections and configure the host information. For details about operations on the management console, see <a href="#">Enhanced Datasource Connections</a>. For details about APIs, see <a href="#">Creating an Enhanced Datasource Connection</a>.</p>
TableName	Specifies the name of a table that has been created in the HBase cluster.
RowKey	Specifies the row key field of the table connected to DLI. The single and composite row keys are supported. A single row key can be of the numeric or string type. The length does not need to be specified. The composite row key supports only fixed-length data of the string type. The format is <b>attribute name 1:Length, attribute name 2:length</b> .

Parameter	Description
Cols	Provides mappings between fields in the DLI table and columns in the HBase table. The mappings are separated by commas (,). In a mapping, the field in the DLI table is located before the colon (:), and information about the HBase table follows the colon (:). In the HBase table information, the column family and column name are separated using a dot (.).

## Precautions

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

## Example

```
CREATE TABLE test_hbase(  
ATTR1 int,  
ATTR2 int,  
ATTR3 string)  
using hbase OPTIONS (  
'ZKHost'='to-hbase-1174405101-CE1bDm5B.datasources.com:2181',  
'TableName'='HBASE_TABLE',
```

```
'RowKey'='ATTR1',  
'Cols'='ATTR2:CF1.C1, ATTR3:CF1.C2');
```

## 8.1.2 Inserting Data to an HBase Table

### Function

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

### Syntax

- Insert the SELECT query result into a table.

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

- Insert a data record into a table.

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

### Keywords

For details about the SELECT keywords, see [Basic Statements](#).

### Parameters

**Table 8-2** Parameters

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

### Precautions

- A DLI table is available.

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

## Example

- Query data in the user table and insert the data into the test table.

```
INSERT INTO test
SELECT ATTR_EXPR
FROM user
WHERE user_name='cyz'
LIMIT 3
GROUP BY user_age
```
- Insert data 1 into the test table.

```
INSERT INTO test
VALUES (1);
```

## 8.1.3 Querying an HBase Table

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

### Syntax

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

### Keywords

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

### Precautions

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

### Example

Query data in the table.

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

### Query Pushdown

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

- Scenarios where query pushdown is supported

- Query pushdown can be performed on data of the following types:
  - Int
  - boolean
  - short
  - long
  - double
  - string

 **NOTE**

Data of the float type does not support query pushdown.

- Query pushdown is not supported for the following filter criteria:

- >, <, >=, <=, =, !=, and, or

The following is an example:

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

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

The following is an example:

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

- IsNotNull()

The following is an example:

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

- in and not in

The following is an example:

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

- between \_ and \_

The following is an example:

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

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

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

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

- Scenarios where query pushdown is not supported

- Query pushdown can be performed on data of the following types:

Except for the preceding data types where query pushdown is supported, data of other types does not support query pushdown.

- Query pushdown is not supported for the following filter criteria:

- Length, count, max, min, join, groupby, orderby, limit, and avg
- Column comparison

The following is an example:

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

## 8.2 Creating a Datasource Connection with an OpenTSDB Table

### 8.2.1 Creating a DLI Table and Associating It with OpenTSDB

#### Function

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

#### Prerequisites

Before creating a DLI table and associating it with OpenTSDB, you need to create a datasource connection. For operations on the management console, see [Enhanced Datasource Connections](#).

#### Syntax

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

#### Keywords

Table 8-3 CREATE TABLE keywords

Parameter	Description
host	<p>OpenTSDB IP address.</p> <p>Create a datasource connection before you can obtain this IP address. For operations on the management console, see <a href="#">Enhanced Datasource Connections</a>.</p> <ul style="list-style-type: none"><li>• After successfully created a connection, you can access the CloudTable OpenTSDB by entering the IP address of the OpenTSDB.</li><li>• You can also access the MRS OpenTSDB. If you have created an enhanced datasource connection, enter the IP address and port number of the node where the OpenTSDB is located. The format is <b>IP:PORT</b>. If the OpenTSDB has multiple nodes, enter one of the node IP addresses.</li></ul>

Parameter	Description
metric	Name of the metric in OpenTSDB corresponding to the DLI table to be created.
tags	Tags corresponding to the metric. The tags are used for classification, filtering, and quick retrieval. You can set 1 to 8 tags, which are separated by commas (.). The parameter value includes values of all tagKs in the corresponding metric.

## Precautions

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

- TAG1 String
- TAG2 String
- timestamp Timestamp
- value double

## Example

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

## 8.2.2 Inserting Data to the OpenTSDB Table

### Function

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

#### NOTE

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

### Syntax

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

## Keywords

**Table 8-4** INSERT INTO keywords

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

## Precautions

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

## Example

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

## 8.2.3 Querying an OpenTSDB Table

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

### NOTE

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

## Syntax

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

## Keywords

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

## Precautions

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

## Example

Query data in the **opentsdb\_table** table.

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

## 8.3 Creating a Datasource Connection with a DWS Table

### 8.3.1 Creating a DLI Table and Associating It with DWS

#### Function

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

#### NOTE

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

For details about datasource authentication, see [Introduction to Datasource Authentication](#).

#### Prerequisites

Before creating a DLI table and associating it with DWS, you need to create a datasource connection. For operations on the management console, see [Enhanced Datasource Connections](#).

#### Syntax

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

## Keywords

**Table 8-5** CREATE TABLE keywords

Parameter	Description
url	<p>Create a datasource connection before you can obtain this GaussDB(DWS) connection URL. For operations on the management console, see <a href="#">Enhanced Datasource Connections</a>.</p> <p>If you have created an enhanced datasource connection, you can use the <b>JDBC Connection String (intranet)</b> provided by DWS or the intranet address and port number to access DWS. The format is <b><i>protocol header://Internal IP address.Internal network port/Database name</i></b>, for example: <b><code>jdbc:postgresql://192.168.0.77:8000/postgres</code></b>.</p> <p><b>NOTE</b> The DWS IP address is in the following format: <b><i>protocol header://IP address.port number/database name</i></b></p> <p>The following is an example: <code>jdbc:postgresql://to-dws-1174405119-ihlUr78j.datasource.com:8000/postgres</code></p> <p>If you want to connect to a database created in DWS, change <b>postgres</b> to the corresponding database name in this connection.</p>
dbtable	Specifies the name or <b>Schema name.Table name</b> of the table that is associated with the DWS. For example: <b><code>public.table_name</code></b> .
user	(Discarded) DWS username.
password	User password of the DWS cluster.
passwdauth	Datasource password authentication name. For details about how to create datasource authentication, see <a href="#">Datasource Authentication</a> in the <i>Data Lake Insight User Guide</i> .
encryption	Set this parameter to <b>true</b> when datasource password authentication is used.
partitionColumn	<p>This parameter is used to set the numeric field used concurrently when data is read.</p> <p><b>NOTE</b></p> <ul style="list-style-type: none"> <li>The <b>partitionColumn</b>, <b>lowerBound</b>, <b>upperBound</b>, and <b>numPartitions</b> parameters must be set at the same time.</li> <li>To improve the concurrent read performance, you are advised to use auto-increment columns.</li> </ul>
lowerBound	Minimum value of a column specified by <b>partitionColumn</b> . The value is contained in the returned result.
upperBound	Maximum value of a column specified by <b>partitionColumn</b> . The value is not contained in the returned result.

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

## Precautions

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

## Example

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

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

## 8.3.2 Inserting Data to the DWS Table

### Function

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

### Syntax

- Insert the SELECT query result into a table.

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

- Insert a data record into a table.

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

### Keywords

For details about the SELECT keywords, see [Basic Statements](#).

### Parameters

Table 8-6 Parameters

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

### Precautions

- A DLI table is available.

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

## Example

- Query data in the user table and insert the data into the test table.

```
INSERT INTO test
SELECT ATTR_EXPR
FROM user
WHERE user_name='cyz'
LIMIT 3
GROUP BY user_age
```

- Insert data 1 into the test table.

```
INSERT INTO test
VALUES (1);
```

## 8.3.3 Querying the DWS Table

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

### Syntax

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

### Keywords

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

### Precautions

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

### Example

To query data in the **dli\_to\_dws** table, enter the following statement:

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

## 8.4 Creating a Datasource Connection with an RDS Table

### 8.4.1 Creating a DLI Table and Associating It with RDS

#### Function

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

 NOTE

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

For details about datasource authentication, see [Introduction to Datasource Authentication](#).

## Prerequisites

Before creating a DLI table and associating it with RDS, you need to create a datasource connection. For operations on the management console, see [Enhanced Datasource Connections](#).

## Syntax

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

## Keywords

**Table 8-7** CREATE TABLE keywords

Parameter	Description
url	<p>Create a datasource connection before you can obtain this RDS connection URL. For operations on the management console, see <a href="#">Enhanced Datasource Connections</a>.</p> <p>After an enhanced datasource connection is created, use the internal network domain name or internal network address and database port number provided by RDS to connect to DLI. If MySQL is used, the format is <b><i>protocol header://internal IP address.internal network port number</i></b>. If PostgreSQL is used, the format is <b><i>protocol header://internal IP address.internal network port number/database name</i></b>.</p> <p>For example: <b><code>jdbc:mysql://192.168.0.193:3306</code></b> or <b><code>jdbc:postgresql://192.168.0.193:5432/postgres</code></b>.</p>
driver	<p>JDBC driver class name. To connect to a MySQL cluster, enter <b><code>com.mysql.jdbc.Driver</code></b>. To connect to a PostgreSQL cluster, enter <b><code>org.postgresql.Driver</code></b>.</p>

Parameter	Description
dbtable	<ul style="list-style-type: none"> <li>To access the MySQL cluster, enter <b><i>Database name.Table name</i></b>.</li> </ul> <p><b>CAUTION</b> The name of the RDS database cannot contain hyphens (-) or ^. Otherwise, the table fails to be created.</p> <ul style="list-style-type: none"> <li>To access the PostGre cluster, enter <b><i>Schema name.Table name</i></b></li> </ul> <p><b>NOTE</b> The schema name is the name of the database schema. A schema is a collection of database objects, including tables and views.</p>
user	(Discarded) Specifies the RDS username.
password	(Discarded) Specifies the RDS username and password.
passwdauth	Datasource password authentication name. For how to create datasource authentication, see <a href="#">Datasource Authentication</a> "Datasource Authentication" in <i>Data Lake Insight User Guide</i> .
encryption	Set this parameter to <b>true</b> when datasource password authentication is used.
partitionColumn	<p>This parameter is used to set the numeric field used concurrently when data is read.</p> <p><b>NOTE</b></p> <ul style="list-style-type: none"> <li>The <b>partitionColumn</b>, <b>lowerBound</b>, <b>upperBound</b>, and <b>numPartitions</b> parameters must be set at the same time.</li> <li>To improve the concurrent read performance, you are advised to use auto-increment columns.</li> </ul>
lowerBound	Minimum value of a column specified by <b>partitionColumn</b> . The value is contained in the returned result.
upperBound	Maximum value of a column specified by <b>partitionColumn</b> . The value is not contained in the returned result.
numPartitions	<p>Number of concurrent read operations.</p> <p><b>NOTE</b> When data is read, the number of concurrent operations are evenly allocated to each task according to the <b>lowerBound</b> and <b>upperBound</b> to obtain data. The following is an example: 'partitionColumn'='id', 'lowerBound'='0', 'upperBound'='100', 'numPartitions'='2'</p> <p>Two concurrent tasks are started in DLI. The execution ID of one task is greater than or equal to <b>0</b> and the ID is less than <b>50</b>, and the execution ID of the other task is greater than or equal to <b>50</b> and the ID is less than <b>100</b>.</p>

Parameter	Description
fetchsize	Number of data records obtained in each batch during data reading. The default value is <b>1000</b> . If this parameter is set to a large value, the performance is good but more memory is occupied. If this parameter is set to a large value, memory overflow may occur.
batchsize	Number of data records written in each batch. The default value is <b>1000</b> . If this parameter is set to a large value, the performance is good but more memory is occupied. If this parameter is set to a large value, memory overflow may occur.
truncate	Indicates whether to clear the table without deleting the original table when <b>overwrite</b> is executed. The options are as follows: <ul style="list-style-type: none"><li>• true</li><li>• false</li></ul> The default value is <b>false</b> , indicating that the original table is deleted and then a new table is created when the <b>overwrite</b> operation is performed.
isolationLevel	Transaction isolation level. The options are as follows: <ul style="list-style-type: none"><li>• NONE</li><li>• READ_UNCOMMITTED</li><li>• READ_COMMITTED</li><li>• REPEATABLE_READ</li><li>• SERIALIZABLE</li></ul> The default value is <b>READ_UNCOMMITTED</b> .

## Precautions

When creating up an RDS association table for the first time, there is no need to define the table's schema. DLI will automatically retrieve the schema from the RDS parameter **dbtable** to create the association table.

If you make changes to the fields in the RDS table, the associated table will not update automatically. In such cases, you need to recreate the association table to ensure its schema matches the modified RDS table.

## Example

### Accessing MySQL

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

### Accessing PostgreSQL

```
CREATE TABLE IF NOT EXISTS dli_to_rds
USING JDBC OPTIONS (
'url='jdbc:postgresql://to-rds-1174405119-oLRHAGE7.datasources.com:5432/postgreDB',
'driver'='org.postgresql.Driver',
'dbtable'='pg_schema.test1',
'password'='xxx',
'encryption'='true');
```

## 8.4.2 Inserting Data to the RDS Table

### Function

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

### Syntax

- Insert the SELECT query result into a table.

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

- Insert a data record into a table.

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

### Keywords

For details about the SELECT keywords, see [Basic Statements](#).

### Parameters

**Table 8-8** Parameters

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

## Precautions

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

## Example

- Query data in the user table and insert the data into the test table.

```
INSERT INTO test
SELECT ATTR_EXPR
FROM user
WHERE user_name='cyz'
LIMIT 3
GROUP BY user_age
```

- Insert data 1 into the test table.

```
INSERT INTO test
VALUES (1);
```

## 8.4.3 Querying the RDS Table

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

### Syntax

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

### Keywords

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

### Precautions

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

### Example

Query data in the **test\_ct** table.

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

## 8.5 Creating a Datasource Connection with a CSS Table

## 8.5.1 Creating a DLI Table and Associating It with CSS

### Function

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

#### NOTE

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

For details about datasource authentication, see [Introduction to Datasource Authentication](#).

### Prerequisites

Before creating a DLI table and associating it with CSS, you need to create a datasource connection. For operations on the management console, see [Enhanced Datasource Connections](#).

### Syntax

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

### Keywords

**Table 8-9** CREATE TABLE keywords

Parameter	Description
es.nodes	<p>Create a datasource connection before you can obtain this CSS connection address. For operations on the management console, see <a href="#">Enhanced Datasource Connections</a>.</p> <p>If you have created an enhanced datasource connection, you can use the internal IP address provided by CSS. The format is <b><i>IP1:PORT1,IP2:PORT2</i></b>.</p>

Parameter	Description
resource	<p>The <b>resource</b> is used to specify the CSS datasource connection name. You can use <b>/index/type</b> to specify the resource location (for easier understanding, the <b>index</b> can be seen as <b>database</b> and <b>type</b> as <b>table</b>).</p> <p><b>NOTE</b></p> <ul style="list-style-type: none"><li>• In ES 6.X, a single index supports only one type, and the type name can be customized.</li><li>• In ES 7.X, a single index uses <b>_doc</b> as the type name and cannot be customized. To access ES 7.X, set this parameter to <b>index</b>.</li></ul>
pushdown	Indicates whether the press function of CSS is enabled. The default value is set to <b>true</b> . If there are a large number of I/O transfer tables, the <b>pushdown</b> can be enabled to reduce I/Os when the <b>where</b> filtering conditions are met.
strict	Indicates whether the CSS <b>pushdown</b> is strict. The default value is set to <b>false</b> . In exact match scenarios, more I/Os are reduced than <b>pushdown</b> .
batch.size.entries	Maximum number of entries that can be inserted to a batch processing. The default value is <b>1000</b> . If the size of a single data record is so large that the number of data records in the bulk storage reaches the upper limit of the data amount of a single batch processing, the system stops storing data and submits the data based on the <b>batch.size.bytes</b> .
batch.size.bytes	Maximum amount of data in a single batch processing. The default value is 1 MB. If the size of a single data record is so small that the number of data records in the bulk storage reaches the upper limit of the data amount of a single batch processing, the system stops storing data and submits the data based on the <b>batch.size.entries</b> .
es.nodes.wan.only	Indicates whether to access the Elasticsearch node using only the domain name. The default value is <b>false</b> . If the original internal IP address provided by CSS is used as the <b>es.nodes</b> , you do not need to set this parameter or set it to <b>false</b> .
es.mapping.id	Specifies a field whose value is used as the document ID in the Elasticsearch node. <b>NOTE</b> <ul style="list-style-type: none"><li>• The document ID in the same <b>/index/type</b> is unique. If a field that functions as a document ID has duplicate values, the document with the duplicate ID will be overwritten when the ES is inserted.</li><li>• This feature can be used as a fault tolerance solution. When data is being inserted, the DLI job fails and some data has been inserted into Elasticsearch. The data is redundant. If Document id is set, the last redundant data will be overwritten when the DLI job is executed again.</li></ul>
es.net.ssl	Whether to connect to the secure CSS cluster. The default value is <b>false</b> .

Parameter	Description
es.certificat e.name	Name of the datasource authentication used to connect to the secure CSS cluster. For details about how to create datasource authentication, see <a href="#">Datasource Authentication</a> in the <i>Data Lake Insight User Guide</i> .

#### NOTE

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

## Example

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

## 8.5.2 Inserting Data to the CSS Table

### Function

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

### Syntax

- Insert the SELECT query result into a table.

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

- Insert a data record into a table.

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

### Keywords

For details about the SELECT keywords, see [Basic Statements](#).

## Parameters

**Table 8-10** Parameters

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

## Precautions

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

## Example

- Query data in the user table and insert the data into the test table.

```
INSERT INTO test
SELECT ATTR_EXPR
FROM user
WHERE user_name='cyz'
LIMIT 3
GROUP BY user_age
```

- Insert data 1 into the test table.

```
INSERT INTO test
VALUES (1);
```

### 8.5.3 Querying the CSS Table

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

## Syntax

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

## Keywords

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

## Precautions

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

## Example

To query data in the **dli\_to\_css** table, enter the following statement:

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

# 8.6 Creating a Datasource Connection with a DCS Table

## 8.6.1 Creating a DLI Table and Associating It with DCS

### Function

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

#### NOTE

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

For details about datasource authentication, see [Introduction to Datasource Authentication](#).

### Prerequisites

Before creating a DLI table and associating it with DCS, you need to create a datasource connection and bind it to a queue. For details about operations on the management console, see [Enhanced Datasource Connection](#)

### Syntax

- Specified key

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

- Wildcard key  

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

## Keywords

**Table 8-11** CREATE TABLE keywords

Parameter	Description
host	To connect to DCS, you need to create a datasource connection first. For details about operations on the management console, see <a href="#">Enhanced Datasource Connections</a> .  After creating an enhanced datasource connection, use the connection address provided by DCS. If there are multiple connection addresses, select one of them.  <b>NOTE</b> Currently, only enhanced datasource is supported.
port	DCS connection port, for example, 6379.
password	Password entered during DCS cluster creation. You do not need to set this parameter when accessing a non-secure Redis cluster.
passwdauth	Datasource password authentication name. For details about how to create datasource authentication, see <a href="#">Datasource Authentication</a> in the <i>Data Lake Insight User Guide</i> .
encryption	Set this parameter to <b>true</b> when datasource password authentication is used.
table	The key or hash key in Redis. <ul style="list-style-type: none"> <li>• This parameter is mandatory when Redis data is inserted.</li> <li>• Either this parameter or the <b>keys.pattern</b> parameter when Redis data is queried.</li> </ul>
keys.pattern	Use a regular expression to match multiple keys or hash keys. This parameter is used only for query. Either this parameter or <b>table</b> is used to query Redis data.
key.column	(Optional) Specifies a field in the schema as the key ID in Redis. This parameter is used together with the <b>table</b> parameter when data is inserted.
partitions.number	Number of concurrent tasks during data reading.

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

#### NOTE

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

The following methods can be used to process complex data:

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

## Example

- Specifying a table

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

- Wildcarding the table name

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

## 8.6.2 Inserting Data to a DCS Table

### Function

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

### Syntax

- Insert the SELECT query result into a table.

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

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

- Insert a data record into a table.

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

## Keywords

For details about the SELECT keywords, see [Basic Statements](#).

## Parameters

**Table 8-12** Parameters

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

## Precautions

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

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

The Redis database contains two tables, naming **test\_with\_key\_column:James** and **test\_with\_key\_column:Michael** respectively.

```
192.168.7.238:6379> keys test_with_key_column:*
1) "test_with_key_column:Michael"
2) "test_with_key_column:James"
192.168.7.238:6379>
```

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

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

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

In Redis, there are two tables named **test\_without\_key\_column:uuid**.

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

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

## Example

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

## 8.6.3 Querying the DCS Table

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

## Syntax

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

## Keywords

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

## Example

Query data in the **test\_redis** table.

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

## 8.7 Creating a Datasource Connection with a DDS Table

### 8.7.1 Creating a DLI Table and Associating It with DDS

#### Function

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

#### NOTE

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

For details about datasource authentication, see [Introduction to Datasource Authentication](#).

#### Prerequisites

Before creating a DLI table and associating it with DDS, you need to create a datasource connection and bind it to a queue. For details about operations on the management console, see [Enhanced Datasource Connection](#)

#### Syntax

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

#### NOTE

Document Database Service (DDS) is fully compatible with the MongoDB protocol. Therefore, the syntax used is **using mongo options**.

## Keywords

**Table 8-13** CREATE TABLE keywords

Parameter	Description
url	Before obtaining the DDS IP address, you need to create a datasource connection first. For details about operations on the management console, see <a href="#">Enhanced Datasource Connections</a> . After creating an enhanced datasource connection, use the random connection address provided by DDS. The format is as follows: "IP:PORT[,IP:PORT]/[DATABASE][.COLLECTION] [AUTH_PROPERTIES]" Example: "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
database	DDS database name. If the database name is specified in the URL, the database name in the URL does not take effect.
collection	Collection name in the DDS. If the collection is specified in the URL, the collection in the URL does not take effect.
user	(Discarded) Username for accessing the DDS cluster.
password	(Discarded) Password for accessing the DDS cluster.
passwdauth	Datasource password authentication name. For details about how to create datasource authentication, see <a href="#">Datasource Authentication</a> in the <i>Data Lake Insight User Guide</i> .
encryption	Set this parameter to <b>true</b> when datasource password authentication is used.

### NOTE

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

## Example

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

## 8.7.2 Inserting Data to the DDS Table

### Function

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

## Syntax

- Insert the SELECT query result into a table.

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

- Insert a data record into a table.

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

- Overwriting the inserted data

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

## Keywords

For details about the SELECT keywords, see [Basic Statements](#).

## Parameters

**Table 8-14** Parameters

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

## Precautions

A DLI table is available.

## Example

- Query data in the user table and insert the data into the test table.

```
INSERT INTO test
SELECT ATTR_EXPR
FROM user
WHERE user_name='cyz'
LIMIT 3
GROUP BY user_age
```

- Insert data 1 into the test table.

```
INSERT INTO test
VALUES (1);
```

### 8.7.3 Querying the DDS Table

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

#### Syntax

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

#### Keywords

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

#### Precautions

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

#### Example

Query data in the **test\_table1** table.

```
SELECT * FROM test_table1 limit 100;
```

## 8.8 Creating a Datasource Connection with an Oracle Table

### 8.8.1 Creating a DLI Table and Associating It with Oracle

#### Function

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

#### Prerequisites

- Before creating a DLI table and associating it with Oracle, you need to create an enhanced datasource connection.  
For details about operations on the management console, see [Enhanced Datasource Connections](#).
- Only enhanced datasource connections can be used to connect to Oracle, and only pay-per-use and yearly/monthly queues support enhanced datasource

connections. So, only SQL jobs on pay-per-use and yearly/monthly queues can be connected to Oracle databases.

## Syntax

```
CREATE TABLE [IF NOT EXISTS] TABLE_NAME
USING ORACLE OPTIONS (
  'url'='xx',
  'driver'='DRIVER_NAME',
  'dbtable'='db_in_oracle.table_in_oracle',
  'user' = 'xxx',
  'password' = 'xxx',
  'resource' = 'obs://rest-authinfo/tools/oracle/driver/ojdbc6.jar'
);
```

## Keywords

Table 8-15 CREATE TABLE keywords

Parameter	Description
url	URL of the Oracle database. The URL can be in either of the following format: <ul style="list-style-type: none"><li>• Format 1: <b>jdbc:oracle:thin:@host:port:SID</b>, in which <i>SID</i> is the unique identifier of the Oracle database.</li><li>• Format 2: <b>jdbc:oracle:thin:@//host:port/service_name</b>. This format is recommended by Oracle. For a cluster, the SID of each node may differ, but their service name is the same.</li></ul>
driver	Oracle driver class name: <b>oracle.jdbc.driver.OracleDriver</b>
dbtable	Name of the table associated with the Oracle database or <i>Username.Table name</i> , for example, <b>public.table_name</b> .
user	Oracle username.
password	Oracle password.
resource	OBS path of the Oracle driver package. Example: <b>obs://rest-authinfo/tools/oracle/driver/ojdbc6.jar</b> If the driver JAR file defined in this parameter is updated, you need to restart the queue for the update to take effect.

## Example

### Creating an Oracle datasource table

```
CREATE TABLE IF NOT EXISTS oracleTest
USING ORACLE OPTIONS (
  'url'='jdbc:oracle:thin:@//192.168.168.40:1521/helowin',
  'driver'='oracle.jdbc.driver.OracleDriver',
  'dbtable'='test.Student',
  'user' = 'test',
  'password' = 'test',
  'resource' = 'obs://rest-authinfo/tools/oracle/driver/ojdbc6.jar'
);
```

## 8.8.2 Inserting Data to an Oracle Table

### Function

This statement is used to insert data into an associated Oracle table.

### Syntax

- Insert the SELECT query result into a table.

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

- Insert a data record into a table.

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

- Overwriting the inserted data

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

### Keywords

For details about the SELECT keywords, see [Basic Statements](#).

### Parameters

**Table 8-16** Parameters

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

## Precautions

A DLI table is available.

## Example

- Query data in the user table and insert the data into the test table.

```
INSERT INTO test
SELECT ATTR_EXPR
FROM user
WHERE user_name='cyz'
LIMIT 3
GROUP BY user_age
```

- Insert data 1 into the test table.

```
INSERT INTO test
VALUES (1);
```

## 8.8.3 Querying an Oracle Table

### Function

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

### Syntax

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

### Keywords

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

### Precautions

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

### Example

Querying data in the **test\_oracle** table

```
SELECT * FROM test_oracle limit 100;
```

# 9 Views

---

## 9.1 Creating a View

### Function

This statement is used to create views.

### Syntax

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

### Keywords

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

### Precautions

- The view to be created must not exist in the current database. Otherwise, an error will be reported. When the view exists, you can add keyword **OR REPLACE** to avoid the error message.
- The table or view information contained in the view cannot be modified. If the table or view information is modified, the query may fail.
- If the compute engines used for creating tables and views are different, the view query may fail due to incompatible varchar types.  
For example, if a table is created using Spark 3.x, you are advised to use Spark 2.x to create a view.

### Example

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

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

## 9.2 Deleting a View

### Function

This statement is used to delete views.

### Syntax

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

### Keywords

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

### Precautions

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

### Example

To delete a view named **student\_view**, run the following statement:

```
DROP VIEW student_view;
```

# 10 Viewing the Execution Plan

---

## Function

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

## Syntax

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

## Keywords

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

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

## Precautions

None

## Example

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

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

# 11 Data Permissions

## 11.1 Data Permissions List

**Table 11-1** describes the SQL statement permission matrix in DLI in terms of permissions on databases, tables, and roles.

**Table 11-1** Permission matrix

Category	SQL statement	Permission	Description
Database	DROP DATABASE db1	The <b>DROP_DATABASE</b> permission of <b>database.db1</b>	-
	CREATE TABLE tb1(...)	The <b>CREATE_TABLE</b> permission of <b>database.db1</b>	-
	CREATE VIEW v1	The <b>CREATE_VIEW</b> permission of <b>database.db1</b>	-
	EXPLAIN query	The <b>EXPLAIN</b> permission of <b>database.db1</b>	Depending on the permissions required by <b>query</b> statements
Table	SHOW CREATE TABLE tb1	The <b>SHOW_CREATE_TABLE</b> permission of <b>database.db1.tables.tb1</b>	-
	DESCRIBE [EXTENDED] [FORMATTED] tb1	The <b>DESCRIBE_TABLE</b> permission of <b>databases.db1.tables.tb1</b>	-
	DROP TABLE [IF EXISTS] tb1	The <b>DROP_TABLE</b> permission of <b>database.db1.tables.tb1</b>	-

Category	SQL statement	Permission	Description
	SELECT * FROM tb1	The <b>SELECT</b> permission of <b>database.db1.tables.tb1</b>	-
	SELECT count(*) FROM tb1	The <b>SELECT</b> permission of <b>database.db1.tables.tb1</b>	-
	SELECT * FROM view1	The <b>SELECT</b> permission of <b>database.db1.tables.view1</b>	-
	SELECT count(*) FROM view1	The <b>SELECT</b> permission of <b>database.db1.tables.view1</b>	-
	LOAD DLI TABLE	The <b>INSERT_INTO_TABLE</b> permission of <b>database.db1.tables.tb1</b>	-
	INSERT INTO TABLE	The <b>INSERT_INTO_TABLE</b> permission of <b>database.db1.tables.tb1</b>	-
	INSERT OVERWRITE TABLE	The <b>INSERT_OVERWRITE_TABLE</b> permission of <b>database.db1.tables.tb1</b>	-
	ALTER TABLE ADD COLUMNS	The <b>ALTER_TABLE_ADD_COLUMNS</b> permission of <b>database.db1.tables.tb1</b>	-
	ALTER TABLE RENAME	The <b>ALTER_TABLE_RENAME</b> permission of <b>database.db1.tables.tb1</b>	-
ROLE&PRIVILEGE	CREATE ROLE	The <b>CREATE_ROLE</b> permission of <b>db</b>	-
	DROP ROLE	The <b>DROP_ROLE</b> permission of <b>db</b>	-
	SHOW ROLES	The <b>SHOW_ROLES</b> permission of <b>db</b>	-
	GRANT ROLES	The <b>GRANT_ROLE</b> permission of <b>db</b>	-
	REVOKE ROLES	The <b>REVOKE_ROLE</b> permission of <b>db</b>	-
	GRANT PRIVILEGE	The <b>GRANT_PRIVILEGE</b> permission of <b>db</b> or <b>table</b>	-
	REVOKE PRIVILEGE	The <b>REVOKE_PRIVILEGE</b> permission of <b>db</b> or <b>table</b>	-

Category	SQL statement	Permission	Description
	SHOW GRANT	The <b>SHOW_GRANT</b> permission of <b>db</b> or <b>table</b>	-

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

- Permissions that can be assigned or revoked on databases are as follows:
  - DROP\_DATABASE (Deleting a database)
  - CREATE\_TABLE (Creating a table)
  - CREATE\_VIEW (Creating a view)
  - EXPLAIN (Explaining a SQL statement as an execution plan)
  - CREATE\_ROLE (Creating a role)
  - DROP\_ROLE (Deleting a role)
  - SHOW\_ROLES (Displaying a role)
  - GRANT\_ROLE (Bounding a role)
  - REVOKE\_ROLE (Unbinding a role)
  - DESCRIBE\_TABLE (Describing a table)
  - DROP\_TABLE (Deleting a table)
  - Select (Querying a table)
  - INSERT\_INTO\_TABLE (Inserting)
  - INSERT\_OVERWRITE\_TABLE (Overwriting)
  - GRANT\_PRIVILEGE (Granting permissions to a database)
  - REVOKE\_PRIVILEGE (Revoking permissions from a database)
  - SHOW\_PRIVILEGES (Checking the database permissions of other users)
  - ALTER\_TABLE\_ADD\_PARTITION (Adding partitions to a partitioned table)
  - ALTER\_TABLE\_DROP\_PARTITION (Deleting partitions from a partitioned table)
  - ALTER\_TABLE\_RENAME\_PARTITION (Renaming table partitions)
  - ALTER\_TABLE\_RECOVER\_PARTITION (Restoring table partitions)
  - ALTER\_TABLE\_SET\_LOCATION (Setting the path of a partition)
  - SHOW\_PARTITIONS (Displaying all partitions)
  - SHOW\_CREATE\_TABLE (Checking table creation statements)
- Permissions that can be assigned or revoked on tables are as follows:
  - DESCRIBE\_TABLE (Describing a table)
  - DROP\_TABLE (Deleting a table)
  - Select (Querying a table)
  - INSERT\_INTO\_TABLE (Inserting)
  - INSERT\_OVERWRITE\_TABLE (Overwriting)

- GRANT\_PRIVILEGE (Granting permissions to a table)
- REVOKE\_PRIVILEGE (Revoking permissions from a table)
- SHOW\_PRIVILEGES (Checking the table permissions of other users)
- ALTER\_TABLE\_ADD\_COLUMNS (Adding a column)
- ALTER\_TABLE\_RENAME (Renaming a table)
- ALTER\_TABLE\_ADD\_PARTITION (Adding partitions to a partitioned table)
- ALTER\_TABLE\_DROP\_PARTITION (Deleting partitions from a partitioned table)
- ALTER\_TABLE\_RENAME\_PARTITION (Renaming table partitions)
- ALTER\_TABLE\_RECOVER\_PARTITION (Restoring table partitions)
- ALTER\_TABLE\_SET\_LOCATION (Setting the path of a partition)
- SHOW\_PARTITIONS (Displaying all partitions)
- SHOW\_CREATE\_TABLE (Checking table creation statements)

## 11.2 Creating a Role

### Function

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

### Syntax

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

### Keywords

None

### Precautions

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

### Example

```
CREATE ROLE role1;
```

## 11.3 Deleting a Role

### Function

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

### Syntax

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

### Keywords

None

### Precautions

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

### Example

```
DROP ROLE role1;
```

## 11.4 Binding a Role

### Function

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

### Syntax

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

### Keywords

None

### Precautions

The **role\_name** and **username** must exist. Otherwise, an error will be reported.

### Example

```
GRANT role1 TO user_name1;
```

## 11.5 Unbinding a Role

### Function

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

### Syntax

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

### Keywords

None

### Precautions

role\_name and user\_name must exist and user\_name has been bound to role\_name.

### Example

To unbind the user\_name1 from role1, run the following statement:

```
REVOKE role1 FROM user_name1;
```

## 11.6 Displaying a Role

### Function

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

### Syntax

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

### Keywords

ALL: Displays all roles.

### Precautions

Keywords ALL and user\_name cannot coexist.

### Example

- To display all roles bound to the user, run the following statement:  

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

```
SHOW ALL ROLES;
```

 **NOTE**

- Only the administrator has the permission to run the **show all roles** statement.
- To display all roles bound to the user named **user\_name1**, run the following statement:  
SHOW ROLES user\_name1;

## 11.7 Granting a Permission

### Function

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

### Syntax

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

### Keywords

ROLE: The subsequent **role\_name** must be a role.

USER: The subsequent **user\_name** must be a user.

### Precautions

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

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

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

- Database format: databases.db\_name  
For details about the permission types supported by a database, see [Data Permissions List](#).
- Table format: databases.db\_name.tables.table\_name  
For details about the permission types supported by a table, see [Data Permissions List](#).
- View format: databases.db\_name.tables.view\_name  
Permission types supported by a view are the same as those supported by a table. For details, see table permissions in [Data Permissions List](#).
- Column format:  
databases.db\_name.tables.table\_name.columns.column\_name  
Columns support only the SELECT permission.

## Example

Run the following statement to grant user\_name1 the permission to delete the **db1** database:

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

Run the following statement to grant user\_name1 the SELECT permission of data table **tb1** in the **db1** database:

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

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

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

## 11.8 Revoking a Permission

### Function

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

### Syntax

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

### Keywords

ROLE: The subsequent **role\_name** must be a role.

USER: The subsequent **user\_name** must be a user.

### Precautions

- The privilege must be the granted permissions of the authorized object in the resource. Otherwise, the permission fails to be revoked. For details about the permission types supported by the privilege, see [Data Permissions List](#).
- The resource can be a queue, database, table, view, or column. The formats are as follows:

- Queue format: queues.queue\_name
- Database format: databases.db\_name
- Table format: databases.db\_name.tables.table\_name
- View format: databases.db\_name.tables.view\_name
- Column format:  
databases.db\_name.tables.table\_name.columns.column\_name

## Example

To revoke the permission of user **user\_name1** to delete database **db1**, run the following statement:

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

To revoke the SELECT permission of user **user\_name1** on table **tb1** in database **db1**, run the following statement:

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

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

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

## 11.9 Displaying the Granted Permissions

### Function

This statement is used to show the permissions granted to a user on a resource.

### Syntax

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

### Keywords

USER: The subsequent **user\_name** must be a user.

### Precautions

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

- Queue format: queues.queue\_name
- Database format: databases.db\_name
- Table format: databases.db\_name.tables.table\_name
- Column format: databases.db\_name.tables.table\_name.columns.column\_name
- View format: databases.db\_name.tables.view\_name

## Example

Run the following statement to show permissions of **user\_name1** in the **db1** database:

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

## 11.10 Displaying the Binding Relationship Between All Roles and Users

### Function

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

### Syntax

```
SHOW PRINCIPALS ROLE;
```

### Keywords

None

### Precautions

The ROLE variable must exist.

### Example

```
SHOW PRINCIPALS role1;
```

# 12 Data Types

## 12.1 Overview

Data type is a basic attribute of data and used to distinguish different types of data. Different data types occupy different storage space and support different operations.

Data is stored in data tables in the database. Each column of a table defines the data type. During storage, data must be stored according to data types.

DLI only supports primitive data types.

## 12.2 Primitive Data Types

**Table 12-1** lists the primitive data types supported by DLI.

**Table 12-1** Primitive data types

Data Type	Description	Storage Space	Value Range	Support by OBS Table	Support by DLI Table
INT	Signed integer	4 bytes	-2147483648 to 2147483647	Yes	Yes
STRING	String	-	-	Yes	Yes
FLOAT	Single-precision floating point	4 bytes	-	Yes	Yes
DOUBLE	Double-precision floating-point	8 bytes	-	Yes	Yes

Data Type	Description	Storage Space	Value Range	Support by OBS Table	Support by DLI Table
DECIMAL(precision,scale)	<p>Decimal number. Data type of valid fixed places and decimal places, for example, 3.5.</p> <ul style="list-style-type: none"> <li><b>precision:</b> indicates the maximum number of digits that can be displayed.</li> <li><b>scale:</b> indicates the number of decimal places.</li> </ul>	-	<p>1&lt;=precision&lt;=38 0&lt;=scale&lt;=38</p> <p>If <b>precision</b> and <b>scale</b> are not specified, <b>DECIMAL (38,38)</b> is used by default.</p>	Yes	Yes
BOOLEAN	Boolean	1 byte	TRUE/ FALSE	Yes	Yes
SMALLINT/SHORT	Signed integer	2 bytes	-32768~32767	Yes	Yes
TINYINT	Signed integer	1 byte	-128~127	Yes	No
BIGINT/ LONG	Signed integer	8 bytes	- 9223372036854775808 to 9223372036854775807	Yes	Yes
TIMESTAMP	<p>Timestamp in raw data format, indicating the date and time</p> <p>Example: 1621434131222</p>	-	-	Yes	Yes
CHAR	Fixed-length string	-	-	Yes	Yes
VARCHAR	Variable-length string	-	-	Yes	Yes

Data Type	Description	Storage Space	Value Range	Support by OBS Table	Support by DLI Table
DATE	Date type in the format of <i>yyyy-mm-dd</i> , for example, <b>2014-05-29</b>	-	<b>DATE</b> does not contain time information. Its value ranges from <b>0000-01-01</b> to <b>9999-12-31</b> .	Yes	Yes

 **NOTE**

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

## INT

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

## STRING

String.

## FLOAT

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

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

## DOUBLE

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

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

## DECIMAL

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

## BOOLEAN

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

## SMALLINT/SHORT

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

## TINYINT

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

## BIGINT/LONG

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

## TIMESTAMP

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

## CHAR

String with a fixed length. In DLI, the **STRING** type is used.

## VARCHAR

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

## DATE

**DATE** supports only explicit conversion (cast) with **DATE**, **TIMESTAMP**, and **STRING**. For details, see [Table 12-2](#).

**Table 12-2** cast function conversion

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

## 12.3 Complex Data Types

Spark SQL supports complex data types, as shown in [Table 12-3](#).

**Table 12-3** 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 <a href="#">Example of ARRAY</a> .
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 <a href="#">Example of Map</a> .

Data Type	Description	Syntax
STRUCT	Indicates a group of named fields. The data types of the fields can be different.	struct(<value1>,<value2>[, ..]) For details, see <a href="#">Example of STRUCT</a> .

## Restrictions

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

## Example of ARRAY

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

1. Create a table.

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

2. Run the following statements to insert test data:

```
INSERT INTO array_test VALUES ('test',array(1,2,3,4));
INSERT INTO array_test VALUES ('test2',array(4,5,6,7))
INSERT INTO array_test VALUES ('test3',array(7,8,9,0));
```

3. Query the result.

To query all data in the **array\_test** table, run the following statement:

```
SELECT * FROM array_test;
```

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

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

```
SELECT id[0] FROM array_test;
```

```
7  
4  
1
```

## Example of Map

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

1. Create a table.

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

2. Run the following statements to insert test data:

```
INSERT INTO map_test VALUES ('test4',map('math',70,'chemistry',84));  
INSERT INTO map_test VALUES ('test5',map('math',85,'chemistry',97));  
INSERT INTO map_test VALUES ('test6',map('math',88,'chemistry',80));
```

3. Query the result.

To query all data in the **map\_test** table, run the following statement:

```
SELECT * FROM map_test;
```

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

To query the math score in the **map\_test** table, run the following statement:

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

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

## Example of STRUCT

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

1. Create a table.

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

2. Run the following statements to insert test data:

```
INSERT INTO struct_test VALUES (8, struct('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":"user2","age":26}  
9{"name":"user3","age":25}
```

Query **name** and **age** in the **struct\_test** table.

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

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

# 13 User-Defined Functions

---

## 13.1 Creating a Function

### Function

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

For details about the custom functions, see [Calling UDFs in Spark SQL Jobs](#) and [Calling UDTFs in Spark SQL Jobs](#).

### 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 exists in the database, the system reports an error.
- Only the Hive syntax can be used to create functions.
- If you specify the same class name for two UDFs, the functions conflict though the package names are different. Avoid this problem because it causes failure of job execution.
- To enable UDF hot loading, submit a service ticket to request whitelisting of your account.

## Keywords

- USING <resources>: resources to be loaded. It can be a list of JARs, files, or URIs.
- OR REPLACE: Whether to reload function resources
  - The following table describes scenarios where you do not need OR REPLACE.

**Table 13-1** Scenarios where OR REPLACE is not specified

No.	Description	Examples	Take Effect When	Operation Impact
1	You modify the implementation of the original class and specify the original JAR package name and class name for a new function.	<ol style="list-style-type: none"> <li>UDF F1 created for a Spark SQL queue is contained in C1 in the JAR package J1.</li> <li>You modify the implementation of function in the J1 package, save the function as UDF F2, and specify its class as C1 and the JAR package as J1.</li> </ol> <p><b>NOTE</b> Note that if UDF names conflict, the function will fail to be created. In this case, you can use <b>OR REPLACE</b> to replace F1 in all jobs with F2.</p>	The Spark SQL queue is restarted.	<ol style="list-style-type: none"> <li>Running jobs are interrupted because the Spark SQL queue must be restarted.</li> <li>After the queue is restarted, original UDF F1 becomes the same as F2.</li> </ol>

No.	Description	Examples	Take Effect When	Operation Impact
2	You create a new class in the original package, specify the new class to the UDF you created, and retain the original package name.	<ol style="list-style-type: none"><li>1. UDF F1 created for a Spark SQL queue is contained in C1 in the JAR package J1.</li><li>2. You add class C2 to the JAR package J1. C1 remains unchanged. You create UDF F2 and specify its class name to C2 and its JAR package to J1.</li></ol>	The Spark SQL queue is restarted.	<ol style="list-style-type: none"><li>1. Running jobs are interrupted because the Spark SQL queue must be restarted.</li><li>2. After the queue is restarted, F1 remains unchanged.</li></ol>

No.	Description	Examples	Take Effect When	Operation Impact
3	You keep the implementation of the original functions unchanged, and repack the program. You specify a new JAR package for the new function and retain the original class name.	<ol style="list-style-type: none"> <li>1. UDF F1 created for a Spark SQL queue is contained in C1 in the JAR package J1.</li> <li>2. You repack the JAR package as J2. The function logic remains unchanged. You specify the class of the created UDF to C1, and the JAR package name to J2.</li> </ol>	UDF F2 is created.	None

- If OR REPLACE is used when you create a UDF, the existing function will be replaced and the replacement takes effect immediately.

---

 **CAUTION**

- **To use the OR REPLACE keyword, you need to submit a service ticket.**
  - To make the replacement take effect immediately on all SQL queues, run the statement for each SQL queue. The replacement may take effect 0 to 12 hours later on the queues where the statement is not executed.
  - If you modify the J1 package (for example, CREATE OR REPLACE FUNCTION F1) in the middle of job execution, jobs being executed run the original UDF F1 (of the C1 class in the J1 package) and jobs that starts after the modification run the new F1 logic.
-

**Table 13-2** Scenarios where OR REPLACE is used

No.	Description	Examples	Take Effect When	Operation Impact
1	You modify the implementation of the original class and specify the original JAR package name and class name for a new function.	<ol style="list-style-type: none"><li data-bbox="767 407 962 703">1. UDF F1 created for a Spark SQL queue is contained in C1 in the JAR package J1.</li><li data-bbox="767 710 962 1247">2. You modify the implementation of a function in the J1 package, save the function as UDF F1, and specify its class as C1 and the JAR package as J1.</li></ol>	Immediately	F1 is changed.

No.	Description	Examples	Take Effect When	Operation Impact
2	You create a new class in the original package, specify the new class to the UDF you created, and retain the original package name.	<ol style="list-style-type: none"> <li>1. UDF F1 created for a Spark SQL queue is contained in C1 in the JAR package J1.</li> <li>2. You add class C2 to the JAR package J1. C1 remains unchanged. You create UDF F2 and specify its class name to C2 and its JAR package to J1. F2 does not take effect immediately because it is the first function specified to the C2 class.</li> </ol>	<p>Either of the following happens:</p> <ul style="list-style-type: none"> <li>• CREATE OR REPLACE FUNCTION is executed again.</li> <li>• The Spark SQL queue is restarted.</li> </ul>	If you restart the Spark SQL queue, running jobs will be affected.

No.	Description	Examples	Take Effect When	Operation Impact
3	You change the class name and specify the new class name to the new UDF. The package name remains unchanged.	<ol style="list-style-type: none"> <li>1. UDF F1 created for a Spark SQL queue is contained in C1 in the JAR package J1.</li> <li>2. You change C1 in the JAR package of J1 to C2, create the UDF F2, specify its class name to C2, and its JAR package name to J1.</li> </ol>	Immediately	F1 is replaced by F2 immediately.
4	You keep the implementation of the original functions unchanged, and repack the program. You specify a new JAR package for the new function and retain the original class name.	<ol style="list-style-type: none"> <li>1. UDF F1 created for a Spark SQL queue is contained in C1 in the JAR package J1.</li> <li>2. You repack the JAR package as J2. The function logic remains unchanged. You specify the class of the created UDF to C1, and the JAR package name to J2.</li> </ol>	Immediately	None

## Example

Create the mergeBill function.

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

## 13.2 Deleting a Function

### Function

This statement is used to delete functions.

### Syntax

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

### Keywords

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

### Precautions

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

## Example

The mergeBill function is deleted.

```
DROP FUNCTION mergeBill;
```

## 13.3 Displaying Function Details

### Function

Displays information about a specified function.

### Syntax

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

### Keywords

**EXTENDED**: displays extended usage information.

## Precautions

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

## Example

Displays information about the mergeBill function.

```
DESCRIBE FUNCTION mergeBill;
```

# 13.4 Displaying All Functions

## Function

View all functions in the current project.

## Syntax

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

In the preceding statement, regex is a regular expression. For details about its parameters, see [Table 13-3](#).

**Table 13-3** regex parameters

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

For details about other expressions, see the official website.

## Keywords

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

## Precautions

The function that matches the given regular expression or function name are displayed. If no regular expression or name is provided, all functions are displayed.

If `USER` or `SYSTEM` is specified, user-defined Spark SQL functions and system-defined Spark SQL functions are displayed, respectively.

## Example

This statement is used to view all functions.

```
SHOW FUNCTIONS;
```

# 14 Built-In Functions

## 14.1 Date Functions

### 14.1.1 Overview

**Table 14-1** lists the date functions supported by DLI.

**Table 14-1** Date/time functions

Function	Syntax	Value Type	Description
<b>add_months</b>	add_months(string start_date, int num_months)	STRING	Returns the date that is <b>num_months</b> after <b>start_date</b> .
<b>current_date</b>	current_date()	DATE	Returns the current date, for example, <b>2016-07-04</b> .
<b>current_timestamp</b>	current_timestamp()	TIMESTAMP	Returns a timestamp of the TIMESTAMP type.
<b>date_add</b>	date_add(string startdate, int days)	STRING or DATE	Adds a number of days to a date.

Function	Syntax	Value Type	Description
<b>dateadd</b>	dateadd(string date, bigint delta, string datepart)	STRING or DATE	Changes a date based on <b>datepart</b> and <b>delta</b> . <b>date</b> : This parameter is mandatory. Date value, which is of the STRING type. The time format is <b>yyyy-mm-dd hh:mi:ss</b> , for example, <b>2021-08-28 00:00:00</b> . <b>delta</b> : This parameter is mandatory. Adjustment amplitude, which is of the BIGINT type. <b>datepart</b> : Adjustment unit, which is a constant of the STRING type. This parameter is mandatory.
<b>date_sub</b>	date_sub(string startdate, int days)	STRING	Subtracts a number of days from a date.
<b>date_format</b>	date_format(string date, string format)	STRING	Converts a date into a string based on the format specified by <b>format</b> .
<b>datediff</b>	datediff(string date1, string date2)	BIGINT	Calculates the difference between <b>date1</b> and <b>date2</b> .
<b>datediff1</b>	datediff1(string date1, string date2, string datepart)	BIGINT	Calculates the difference between <b>date1</b> and <b>date2</b> and returns the difference in a specified datepart.
<b>datepart</b>	datepart (string date, string datepart)	BIGINT	Returns the value of the field that meets a specified time unit in the date.
<b>datetrunc</b>	datetrunc (string date, string datepart)	STRING	Calculates the date obtained through the truncation of a specified date based on a specified datepart. <b>date</b> : The value is in the <b>yyyy-mm-dd</b> or <b>yyyy-mm-dd hh:mi:ss</b> format. This parameter is mandatory. <b>datepart</b> : Supported date format, which is a STRING constant. This parameter is mandatory.
<b>day/dayofmonth</b>	day(string date), dayofmonth(string date)	INT	Returns the date of a specified time.

Function	Syntax	Value Type	Description
<a href="#">from_unixtime</a>	from_unixtime(bigint unixtime)	STRING	Converts a timestamp to a time, in <b>yyyy-MM-dd HH:mm:ss</b> or <b>yyyyMMddHHmmss.uuuuuu</b> format. For example, <b>select FROM_UNIXTIME(1608135036,'yyyy-MM-dd HH:mm:ss')</b> .
<a href="#">from_utc_timestamp</a>	from_utc_timestamp(string timestamp, string timezone)	TIMESTAMP	Converts a UTC timestamp to the corresponding timestamp in a specific time zone.
<a href="#">getdate</a>	getdate()	STRING	Obtains the current system time.
<a href="#">hour</a>	hour(string date)	INT	Returns the hour (from 0 to 23) of a specified time.
<a href="#">isdate</a>	isdate(string date , string format)	BOOLEAN	<b>date:</b> Date, which is implicitly converted to the STRING type and then used for calculation. This parameter is mandatory. <b>format:</b> Unsupported date extension format, which is a STRING constant. This parameter is mandatory. If there are redundant format strings in <b>format</b> , only the date value corresponding to the first format string is used. Other format strings are used as separators. For example, <b>isdate("1234-yyyy", "yyyy-yyyy")</b> returns <b>True</b> .
<a href="#">last_day</a>	last_day(string date)	DATE	Returns the last day of the month a date belongs to, in the format of <b>yyyy-MM-dd</b> , for example, <b>2015-08-31</b> .
<a href="#">lastday</a>	lastday(string date)	STRING	Returns the last day of the month a date belongs to. The value is of the STRING type and is in the <b>yyyy-mm-dd hh:mi:ss</b> format.
<a href="#">minute</a>	minute(string date)	INT	Returns the minute (from 0 to 59) of a specified time.
<a href="#">month</a>	month(string date)	INT	Returns the month (from January to December) of a specified time.

Function	Syntax	Value Type	Description
<b>months_between</b>	months_between(string date1, string date2)	DOUBLE	Returns the month difference between <b>date1</b> and <b>date2</b> .
<b>next_day</b>	next_day(string start_date, string day_of_week)	DATE	Returns the date closest to <b>day_of_week</b> after <b>start_date</b> , in the <b>yyyy-MM-dd</b> format. <b>day_of_week</b> indicates a day in a week, for example, Monday or Friday.
<b>quarter</b>	quarter(string date)	INT	Returns the quarter of the date, timestamp, or string, for example, <b>quarter('2015-04-01')=2</b> .
<b>second</b>	second(string date)	INT	Returns the second (from 0 to 59) of a specified time.
<b>to_char</b>	to_char(string date, string format)	STRING	Converts a date into a string in a specified format.
<b>to_date</b>	to_date(string timestamp)	DATE	Returns the date part of a time string, for example, <b>to_date("1970-01-01 00:00:00") = "1970-01-01"</b> .
<b>to_date1</b>	to_date1(string date, string format)	STRING	The value is of the STRING type, in the <b>yyyy-mm-dd hh:mi:ss</b> format. If the value of <b>date</b> or <b>format</b> is <b>NULL</b> , <b>NULL</b> is returned.  Converts a string in a specified format to a date value.
<b>to_utc_timestamp</b>	to_utc_timestamp(string timestamp, string timezone)	TIMESTAMP	Converts a timestamp in a given time zone to a UTC timestamp.
<b>trunc</b>	trunc(string date, string format)	DATE	Resets the date in a specified format. Supported formats are <b>MONTH/MON/MM</b> and <b>YEAR/YYYY/YY</b> , for example, <b>trunc('2015-03-17', 'MM') = 2015-03-01</b> .
<b>unix_timestamp</b>	unix_timestamp(string timestamp, string pattern)	BIGINT	Returns a Unix timestamp (the number of seconds since <b>1970-01-01 00:00:00</b> ) as an unsigned integer if the function is called without parameters.

Function	Syntax	Value Type	Description
<a href="#">weekday</a>	weekday (string date)	INT	Returns the day of the current week.
<a href="#">weekofyear</a>	weekofyear(string date)	INT	Returns the week number (from 0 to 53) of a specified date.
<a href="#">year</a>	year(string date)	INT	Returns the year of a specified date.

## 14.1.2 add\_months

This function is used to calculate the date after a date value is increased by a specified number of months. That is, it calculates the data that is **num\_months** after **start\_date**.

### Syntax

```
add_months(string start_date, int num_months)
```

### Parameters

Table 14-2 Parameters

Parameter	Mandatory	Type	Description
start_date	Yes	DATE or STRING	Start date The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
num_months	Yes	INT	Number of months to be added

### Return Values

The date that is **num\_months** after **start\_date** is returned, in the **yyyy-mm-dd** format.

The return value is of the DATE type.

 NOTE

- If the value of **start\_date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **start\_date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **start\_date** is **NULL**, an error is reported.
- If the value of **num\_months** is **NULL**, **NULL** is returned.

## Example Code

The value **2023-05-26** is returned.

```
select add_months('2023-02-26',3);
```

The value **2023-05-14** is returned.

```
select add_months('2023-02-14 21:30:00',3);
```

The value **NULL** is returned.

```
select add_months('20230815',3);
```

The value **NULL** is returned.

```
select add_months('2023-08-15 20:00:00',null);
```

## 14.1.3 current\_date

This function is used to return the current date, in the **yyyy-mm-dd** format.

Similar function: [getdate](#). The getdate function is used to return the current system time, in the **yyyy-mm-dd hh:mi:ss** format.

### Syntax

```
current_date()
```

### Parameters

None

### Return Values

The return value is of the DATE type, in the **yyyy-mm-dd** format.

## Example Code

The value **2023-08-16** is returned.

```
select current_date();
```

## 14.1.4 current\_timestamp

This function is used to return the current timestamp.

### Syntax

```
current_timestamp()
```

## Parameters

None

## Return Values

The return value is of the `TIMESTAMP` type.

## Example Code

The value **1692002816300** is returned.

```
select current_timestamp();
```

## 14.1.5 date\_add

This function is used to calculate the number of days in which **start\_date** is increased by **days**.

To obtain the date with a specified change range based on the current date, use this function together with the [current\\_date](#) or [getdate](#) function.

Note that the logic of this function is opposite to that of the [date\\_sub](#) function.

## Syntax

```
date_add(string startdate, int days)
```

## Parameters

**Table 14-3** Parameters

Parameter	Mandatory	Type	Description
start_date	Yes	DATE or STRING	Start date The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
days	Yes	BIGINT	Number of days to be added <ul style="list-style-type: none"><li>• If the value is greater than 0, the number of days is increased.</li><li>• If the value is less than 0, the number of days is subtracted.</li><li>• If the value is 0, the date does not change.</li><li>• If the value is <b>NULL</b>, <b>NULL</b> is returned.</li></ul>

## Return Values

The return value is of the DATE type, in the **yyyy-mm-dd** format.

### NOTE

- If the value of **start\_date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **start\_date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **start\_date** is **NULL**, **NULL** is returned.
- If the value of **days** is **NULL**, **NULL** is returned.

## Example Code

The value **2023-03-01** is returned after one day is added.

```
select date_add('2023-02-28 00:00:00', 1);
```

The value **2023-02-27** is returned after one day is subtracted.

```
select date_add(date '2023-02-28', -1);
```

The value **2023-03-20** is returned.

```
select date_add('2023-02-28 00:00:00', 20);
```

If the current time is **2023-08-14 16:00:00**, **2023-08-13** is returned.

```
select date_add(getdate(),-1);
```

The value **NULL** is returned.

```
select date_add('2023-02-28 00:00:00', null);
```

## 14.1.6 dateadd

This function is used to change a date based on **datepart** and **delta**.

To obtain the date with a specified change range based on the current date, use this function together with the **current\_date** or **getdate** function.

## Syntax

```
dateadd(string date, bigint delta, string datepart)
```

## Parameters

**Table 14-4** Parameters

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Start date The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
delta	Yes	BIGINT	Amplitude, based on which the date is modified
datepart	Yes	STRING	Unit, based on which the date is modified This parameter supports the following extended date formats: year, month or mon, day, and hour. <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>MM</b> indicates the month.</li><li>• <b>dd</b> indicates the day.</li><li>• <b>hh</b> indicates the hour.</li><li>• <b>mi</b> indicates the minute.</li><li>• <b>ss</b> indicates the second.</li></ul>

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.
- If the value of **delta** or **datepart** is **NULL**, **NULL** is returned.

## Example Code

The value **2023-08-15 17:00:00** is returned after one day is added.

```
select dateadd( '2023-08-14 17:00:00', 1, 'dd');
```

The value **2025-04-14 17:00:00** is returned after 20 months are added.

```
select dateadd('2023-08-14 17:00:00', 20, 'mm');
```

The value **2023-09-14 17:00:00** is returned.

```
select dateadd('2023-08-14 17:00:00', 1, 'mm');
```

The value **2023-09-14** is returned.

```
select dateadd('2023-08-14', 1, 'mm');
```

If the current time is **2023-08-14 17:00:00**, **2023-08-13 17:00:00** is returned.

```
select dateadd(getdate(),-1,'dd');
```

The value **NULL** is returned.

```
select dateadd(date '2023-08-14', 1, null);
```

## 14.1.7 date\_sub

This function is used to calculate the number of days in which **start\_date** is subtracted by **days**.

To obtain the date with a specified change range based on the current date, use this function together with the [current\\_date](#) or [getdate](#) function.

Note that the logic of this function is opposite to that of the [date\\_add](#) function.

### Syntax

```
date_sub(string startdate, int days)
```

### Parameters

**Table 14-5** Parameters

Parameter	Mandatory	Type	Description
start_date	Yes	DATE or STRING	Start date The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

Parameter	Mandatory	Type	Description
days	Yes	BIGINT	Number of days to be subtracted <ul style="list-style-type: none"><li>When <b>days</b> is greater than 0, the specified number of days is subtracted from <b>startdate</b>.</li><li>When <b>days</b> is less than 0, the specified number of days is added to <b>startdate</b>.</li><li>When <b>days</b> equals 0, that means adding 0 days, so the date does not change.</li><li>When the value is <b>NULL</b>, <b>NULL</b> is returned.</li></ul>

## Return Values

The return value is of the DATE type.

### NOTE

- If the value of **start\_date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **start\_date** is of the DATE or STRING type but is not in one of the supported formats, NULL is returned.
- If the value of **date** is **NULL**, **NULL** is returned.
- If the value of **format** is **NULL**, **NULL** is returned.

## Example Code

The value **2023-08-12** is returned after two days are subtracted.

```
select date_sub('2023-08-14 17:00:00', 2);
```

The value **2023-08-15** is returned after one day is added.

```
select date_sub(date'2023-08-14', -1);
```

If the current time is **2023-08-14 17:00:00**, **2023-08-13** is returned.

```
select date_sub(getdate(),1);
```

The value **NULL** is returned.

```
select date_sub('2023-08-14 17:00:00', null);
```

## 14.1.8 date\_format

This function is used to convert a date into a string based on the format specified by **format**.

## Syntax

```
date_format(string date, string format)
```

## Parameters

**Table 14-6** Parameters

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date to be converted The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
format	Yes	STRING	Format, based on which the date is converted The value is a combination of the time unit (year, month, day, hour, minute, and second) and any character. <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>MM</b> indicates the month.</li><li>• <b>dd</b> indicates the day.</li><li>• <b>HH</b> indicates the 24-hour clock.</li><li>• <b>hh</b> indicates the 12-hour clock.</li><li>• <b>mm</b> indicates the minute.</li><li>• <b>ss</b> indicates the second.</li></ul>

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.
- If the value of **format** is **NULL**, **NULL** is returned.

## Example Code

The value **2023-08-14** is returned.

```
select date_format('2023-08-14','yyyy-MM-dd');
```

The value **2023-08** is returned.

```
select date_format('2023-08-14','yyyy-MM')
```

The value **20230814** is returned.

```
select date_format('2023-08-14','yyyyMMdd')
```

## 14.1.9 datediff

This function is used to calculate the difference between **date1** and **date2**.

Similar function: [datediff1](#). The **datediff1** function is used to calculate the difference between **date1** and **date2** and return the difference in a specified datepart.

### Syntax

```
datediff(string date1, string date2)
```

### Parameters

Table 14-7 Parameters

Parameter	Mandatory	Type	Description
date1	Yes	DATE or STRING	Minuend of the date difference between <b>date1</b> and <b>date2</b> . The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
date2	Yes	DATE or STRING	Subtrahend of the date difference between <b>date1</b> and <b>date2</b> . The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

### Return Values

The return value is of the BIGINT type.

 NOTE

- If the values of **date1** and **date2** are not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the values of **date1** and **date2** are of the DATE or STRING type but are not in one of the supported formats, **NULL** is returned.
- If the value of **date1** is smaller than that of **date2**, the return value is a negative number.
- If the value of **date1** or **date2** is **NULL**, **NULL** is returned.

## Example Code

The value **10** is returned.

```
select datediff('2023-06-30 00:00:00', '2023-06-20 00:00:00');
```

The value **11** is returned.

```
select datediff(date '2023-05-21', date '2023-05-10');
```

The value **NULL** is returned.

```
select datediff(date '2023-05-21', null);
```

### 14.1.10 datediff1

This function is used to calculate the difference between **date1** and **date2** and return the difference in a specified datepart.

Similar function: [datediff](#). The **datediff** function is used to calculate the difference between **date1** and **date2** but does not return the difference in a specified datepart.

## Syntax

```
datediff1 (string date1, string date2, string datepart)
```

## Parameters

Table 14-8 Parameters

Parameter	Mandatory	Type	Description
date1	Yes	DATE or STRING	Minuend of the date difference between <b>date1</b> and <b>date2</b> . The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

Parameter	Mandatory	Type	Description
date2	Yes	DATE or STRING	Subtrahend of the date difference between <b>date1</b> and <b>date2</b> . The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
datepart	Yes	STRING	Unit of the time to be returned This parameter supports the following extended date formats: year, month or mon, day, and hour. <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>MM</b> indicates the month.</li><li>• <b>dd</b> indicates the day.</li><li>• <b>hh</b> indicates the hour.</li><li>• <b>mi</b> indicates the minute.</li><li>• <b>ss</b> indicates the second.</li></ul>

## Return Values

The return value is of the BIGINT type.

### NOTE

- If the values of **date1** and **date2** are not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the values of **date1** and **date2** are of the DATE or STRING type but are not in one of the supported formats, **NULL** is returned.
- If the value of **date1** is smaller than that of **date2**, the return value is a negative number.
- If the value of **date1** or **date2** is **NULL**, **NULL** is returned.
- If the value of **datepart** is **NULL**, **NULL** is returned.

## Example Code

The value **14400** is returned.

```
select datediff1('2023-06-30 00:00:00', '2023-06-20 00:00:00', 'mi');
```

The value **10** is returned.

```
select datediff1(date '2023-06-21', date '2023-06-11', 'dd');
```

The value **NULL** is returned.

```
select datediff1(date '2023-05-21', date '2023-05-10', null);
```

The value **NULL** is returned.

```
select datediff1(date '2023-05-21', null, 'dd');
```

## 14.1.11 datepart

This function is used to calculate the value that meets the specified **datepart** in **date**.

### Syntax

```
datepart (string date, string datepart)
```

### Parameters

Table 14-9 Parameters

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Start date The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
datepart	Yes	STRING	Time unit of the value to be returned This parameter supports the following extended date formats: year, month or mon, day, and hour. <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>MM</b> indicates the month.</li><li>• <b>dd</b> indicates the day.</li><li>• <b>hh</b> indicates the hour.</li><li>• <b>mi</b> indicates the minute.</li><li>• <b>ss</b> indicates the second.</li></ul>

### Return Values

The return value is of the BIGINT type.

 NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **datepart** is **NULL**, **NULL** is returned.
- If the value of **datepart** is **NULL**, **NULL** is returned.

## Example Code

The value **2023** is returned.

```
select datepart(date '2023-08-14 17:00:00', 'yyyy');
```

The value **2023** is returned.

```
select datepart('2023-08-14 17:00:00', 'yyyy');
```

The value **59** is returned.

```
select datepart('2023-08-14 17:59:59', 'mi')
```

The value **NULL** is returned.

```
select datepart(date '2023-08-14', null);
```

## 14.1.12 datetrunc

This function is used to calculate the date obtained through the truncation of a specified date based on a specified datepart.

It truncates the date before the specified datepart and automatically fills the remaining part with the default value. For details, see [Example Code](#).

## Syntax

```
datetrunc (string date, string datepart)
```

## Parameters

**Table 14-10** Parameters

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Start date The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

Parameter	Mandatory	Type	Description
datepart	Yes	STRING	Time unit of the value to be returned This parameter supports the following extended date formats: year, month or mon, day, and hour. <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>MM</b> indicates the month.</li><li>• <b>dd</b> indicates the day.</li><li>• <b>hh</b> indicates the hour.</li><li>• <b>mi</b> indicates the minute.</li><li>• <b>ss</b> indicates the second.</li></ul>

## Return Values

The return value is of the DATE or STRING type.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **datepart** is **NULL**, **NULL** is returned.
- If the value of **datepart** is hour, minute, or second, the date is truncated to the day and returned.

## Example Code

Example static data

The value **2023-01-01 00:00:00** is returned.

```
select datetrunc('2023-08-14 17:00:00', 'yyyy');
```

The value **2023-08-01 00:00:00** is returned.

```
select datetrunc('2023-08-14 17:00:00', 'month');
```

The value **2023-08-14** is returned.

```
select datetrunc('2023-08-14 17:00:00', 'DD');
```

The value **2023-01-01** is returned.

```
select datetrunc('2023-08-14', 'yyyy');
```

The value **2023-08-14 17:00:00** is returned.

```
select datetrunc('2023-08-14 17:11:11', 'hh');
```

The value **NULL** is returned.

```
select datetrunc('2023-08-14', null);
```

### 14.1.13 day/dayofmonth

This function is used to return the day of a specified date.

#### Syntax

```
day(string date), dayofmonth(string date)
```

#### Parameters

**Table 14-11** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

#### Return Values

The return value is of the INT type.

##### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

#### Example Code

The value **1** is returned.

```
select day('2023-08-01');
```

The value **NULL** is returned.

```
select day('20230816');
```

The value **NULL** is returned.

```
select day(null);
```

### 14.1.14 from\_unixtime

This function is used to convert a timestamp represented by a numeric UNIX value to a date value.

## Syntax

```
from_unixtime(bigint unixtime)
```

## Parameters

**Table 14-12** Parameter

Parameter	Mandatory	Type	Description
unixtime	Yes	BIGINT	Timestamp to be converted, in UNIX format Set this parameter to the first 10 digits of the timestamp in UNIX format.

## Return Values

The return value is of the STRING type, in the **yyyy-mm-dd hh:mi:ss** format.

### NOTE

If the value of **unixtime** is **NULL**, **NULL** is returned.

## Example Code

The value **2023-08-16 09:39:57** is returned.

```
select from_unixtime(1692149997);
```

The value **NULL** is returned.

```
select from_unixtime(NULL);
```

### Example table data

```
select unixdate, from_unixtime(unixdate) as timestamp_from_unixtime from database_t;
```

Output:

```
+-----+-----+
| unixdate          | timestamp_from_unixtime |
+-----+-----+
| 1690944759224    | 2023-08-02 10:52:39     |
| 1690944999811    | 2023-08-02 10:56:39     |
| 1690945005458    | 2023-08-02 10:56:45     |
| 1690945011542    | 2023-08-02 10:56:51     |
| 1690945023151    | 2023-08-02 10:57:03     |
+-----+-----+
```

## 14.1.15 from\_utc\_timestamp

This function is used to convert a UTC timestamp to a UNIX timestamp in a given time zone.

## Syntax

```
from_utc_timestamp(string timestamp, string timezone)
```

## Parameters

**Table 14-13** Parameters

Parameter	Mandatory	Type	Description
timestamp	Yes	DATE STRING TINYINT SMALLINT INT BIGINT	Time to be converted Date value of the DATE or STRING type, or timestamp of the TINYINT, SMALLINT, INT, or BIGINT type. The following formats are supported: yyyy-mm-dd yyyy-mm-dd hh:mi:ss yyyy-mm-dd hh:mi:ss.ff3
timezone	Yes	STRING	Time zone where the time to be converted belongs

## Return Values

The return value is of the **TIMESTAMP** type.

### NOTE

- If the value of **timestamp** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **timestamp** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **timestamp** is **NULL**, **NULL** is returned.
- If the value of **timezone** is **NULL**, **NULL** is returned.

## Example Code

The value **1691978400000** is returned, indicating 2023-08-14 10:00:00.

```
select from_utc_timestamp('2023-08-14 17:00:00','PST');
```

The value **1691917200000** is returned, indicating 2023-08-13 17:00:00.

```
select from_utc_timestamp(date '2023-08-14 00:00:00','PST');
```

The value **NULL** is returned.

```
select from_utc_timestamp('2023-08-13',null);
```

### 14.1.16 getdate

This function is used to return the current system time, in the **yyyy-mm-dd hh:mi:ss** format.

Similar function: [current\\_date](#). The **current\_date** function is used to return the current date, in the **yyyy-mm-dd** format.

## Syntax

```
getdate()
```

## Parameters

None

## Return Values

The return value is of the STRING type, in the **yyyy-mm-dd hh:mi:ss** format.

## Example Code

If the current time is **2023-08-10 10:54:00**, **2023-08-10 10:54:00** is returned.

```
select getdate();
```

## 14.1.17 hour

This function is used to return the hour (from 0 to 23) of a specified time.

## Syntax

```
hour(string date)
```

## Parameters

**Table 14-14** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the INT type.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

## Example Code

The value **10** is returned.

```
select hour('2023-08-10 10:54:00');
```

The value **12** is returned.

```
select hour('12:00:00');
```

The value **NULL** is returned.

```
select hour('20230810105600');
```

The value **NULL** is returned.

```
select hour(null);
```

## 14.1.18 isdate

This function is used to determine whether a date string can be converted into a date value based on a specified format.

### Syntax

```
isdate(string date , string format)
```

### Parameters

**Table 14-15** Parameters

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	String to be checked If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation. The value can be any string.

Parameter	Mandatory	Type	Description
format	Yes	STRING	<p>Format of the date to be converted Constant of the STRING type. Extended date formats are not supported.</p> <p>The value is a combination of the time unit (year, month, day, hour, minute, and second) and any character.</p> <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>mm</b> indicates the month.</li><li>• <b>dd</b> indicates the day.</li><li>• <b>hh</b> indicates the hour.</li><li>• <b>mi</b> indicates the minute.</li><li>• <b>ss</b> indicates the second.</li></ul>

## Return Values

The return value is of the BOOLEAN type.

### NOTE

If the value of **date** or **format** is **NULL**, **NULL** is returned.

## Example Code

The value **true** is returned.

```
select isdate('2023-08-10','yyyy-mm-dd');
```

The value **false** is returned.

```
select isdate(123456789,'yyyy-mm-dd');
```

## 14.1.19 last\_day

This function is used to return the last day of the month a date belongs to.

Similar function: [lastday](#). The **lastday** function is used to return the last day of the month a date belongs to. The hour, minute, and second part is 00:00:00.

## Syntax

```
last_day(string date)
```

## Parameters

**Table 14-16** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the DATE type, in the **yyyy-mm-dd** format.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

## Example Code

The value **2023-08-31** is returned.

```
select last_day('2023-08-15');
```

The value **2023-08-31** is returned.

```
select last_day('2023-08-10 10:54:00');
```

The value **NULL** is returned.

```
select last_day('20230810');
```

### 14.1.20 lastday

This function is used to return the last day of the month a date belongs to. The hour, minute, and second part is 00:00:00.

Similar function: [last\\_day](#). The **last\_day** function is used to return the last day of the month a date belongs to. The return value is in the **yyyy-mm-dd** format.

## Syntax

```
lastday(string date)
```

## Parameters

**Table 14-17** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the STRING type, in the **yyyy-mm-dd hh:mi:ss** format.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

## Example Code

The value **2023-08-31** is returned.

```
select lastday('2023-08-10');
```

The value **2023-08-31 00:00:00** is returned.

```
select lastday ('2023-08-10 10:54:00');
```

The value **NULL** is returned.

```
select lastday (null);
```

## 14.1.21 minute

This function is used to return the minute (from 0 to 59) of a specified time.

## Syntax

```
minute(string date)
```

## Parameters

**Table 14-18** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the INT type.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

## Example Code

The value **54** is returned.

```
select minute('2023-08-10 10:54:00');
```

The value **54** is returned.

```
select minute('10:54:00');
```

The value **NULL** is returned.

```
select minute('20230810105400');
```

The value **NULL** is returned.

```
select minute(null);
```

## 14.1.22 month

This function is used to return the month (from January to December) of a specified time.

## Syntax

```
month(string date)
```

## Parameters

**Table 14-19** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed If the value is of the STRING type, the value must contain at least yyyy-mm-dd and cannot contain redundant strings. The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the INT type.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

## Example Code

The value **8** is returned.

```
select month('2023-08-10 10:54:00');
```

The value **NULL** is returned.

```
select month('20230810');
```

The value **NULL** is returned.

```
select month(null);
```

### 14.1.23 months\_between

This function returns the month difference between **date1** and **date2**.

## Syntax

```
months_between(string date1, string date2)
```

## Parameters

**Table 14-20** Parameters

Parameter	Mandatory	Type	Description
date1	Yes	DATE or STRING	Minuend The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
date2	Yes	DATE or STRING	Subtrahend The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the DOUBLE type.

### NOTE

- If the values of **date1** and **date2** are not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the values of **date1** and **date2** are of the DATE or STRING type but are not in one of the supported formats, **NULL** is returned.
- If **date1** is later than **date2**, the return value is positive. If **date2** is later than **date1**, the return value is negative.
- If **date1** and **date2** correspond to the last day of two different months, an integer month is returned. Otherwise, the calculation is based on the number of days between **date1** and **date2** divided by 31.
- If the value of **date1** or **date2** is **NULL**, **NULL** is returned.

## Example Code

The value **0.0563172** is returned.

```
select months_between('2023-08-16 10:54:00', '2023-08-14 17:00:00');
```

The value **0.06451613** is returned.

```
select months_between('2023-08-16','2023-08-14');
```

The value **NULL** is returned.

```
select months_between('2023-08-16',null);
```

## 14.1.24 next\_day

This function is used to return the date closest to **day\_of\_week** after **start\_date**.

### Syntax

```
next_day(string start_date, string day_of_week)
```

### Parameters

**Table 14-21** Parameters

Parameter	Mandatory	Type	Description
start_date	Yes	DATE or STRING	Date that needs to be processed If the value is of the STRING type, the value must contain at least yyyy-mm-dd and cannot contain redundant strings. The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
day_of_week	Yes	STRING	One day of a week, either the first two or three letters of the word, or the full name of the word for that day. For example, <b>TU</b> indicates Tuesday.

### Return Values

The return value is of the DATE type, in the **yyyy-mm-dd** format.

#### NOTE

- If the value of **start\_date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **start\_date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **start\_date** is **NULL**, **NULL** is returned.
- If the value of **day\_of\_week** is **NULL**, **NULL** is returned.

### Example Code

The value **2023-08-22** is returned.

```
select next_day('2023-08-16','TU');
```

The value **2023-08-22** is returned.

```
select next_day('2023-08-16 10:54:00','TU');
```

The value **2023-08-23** is returned.

```
select next_day('2023-08-16 10:54:00','WE');
```

The value **NULL** is returned.

```
select next_day('20230816','TU');
```

The value **NULL** is returned.

```
select next_day('20230816 20:00:00',null);
```

## 14.1.25 quarter

This function is used to return the quarter of a date. The value ranges from 1 to 4.

### Syntax

```
quarter(string date)
```

### Parameters

**Table 14-22** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

### Return Values

The return value is of the INT type.

#### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

### Example Code

The value **3** is returned.

```
select quarter('2023-08-16 10:54:00');
```

The value **3** is returned.

```
select quarter('2023-08-16');
```

The value **NULL** is returned.

```
select quarter(null);
```

## 14.1.26 second

This function is used to return the second (from 0 to 59) of a specified time.

### Syntax

```
second(string date)
```

### Parameters

Table 14-23 Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

### Return Values

The return value is of the INT type.

#### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

### Example Code

The value **36** is returned.

```
select second('2023-08-16 10:54:36');
```

The value **36** is returned.

```
select second('10:54:36');
```

The value **NULL** is returned.

```
select second('20230816105436');
```

The value **NULL** is returned.

```
select second(null);
```

## 14.1.27 to\_char

This function is used to convert a date into a string in a specified format.

### Syntax

```
to_char(string date, string format)
```

### Parameters

**Table 14-24** Parameters

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• <code>yyyy-mm-dd</code></li><li>• <code>yyyy-mm-dd hh:mi:ss</code></li><li>• <code>yyyy-mm-dd hh:mi:ss.ff3</code></li></ul>
format	Yes	STRING	Format of the date to be converted Constant of the STRING type. Extended date formats are not supported. The value is a combination of the time unit (year, month, day, hour, minute, and second) and any character. <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>mm</b> indicates the month.</li><li>• <b>dd</b> indicates the day.</li><li>• <b>hh</b> indicates the hour.</li><li>• <b>mi</b> indicates the minute.</li><li>• <b>ss</b> indicates the second.</li></ul>

### Return Values

The return value is of the STRING type.

 NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **format** is **NULL**, **NULL** is returned.

## Example Code

The static data **2023-08\*16** is returned.

```
select to_char('2023-08-16 10:54:36','Example static data yyyy-mm*dd');
```

The value **20230816** is returned.

```
select to_char('2023-08-16 10:54:36', 'yyyymmdd');
```

The value **NULL** is returned.

```
select to_char('Example static data 2023-08-16','Example static data yyyy-mm*dd');
```

The value **NULL** is returned.

```
select to_char('20230816', 'yyyy');
```

The value **NULL** is returned.

```
select to_char('2023-08-16 10:54:36', null);
```

## 14.1.28 to\_date

This function is used to return the year, month, and day in a time.

Similar function: [to\\_date1](#). The **to\_date1** function is used to convert a string in a specified format to a date value. The date format can be specified.

## Syntax

```
to_date(string timestamp)
```

## Parameters

**Table 14-25** Parameter

Parameter	Mandatory	Type	Description
timestamp	Yes	DATE STRING	Time to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the DATE type, in the **yyyy-mm-dd** format.

### NOTE

- If the value of **timestamp** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **timestamp** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.

## Example Code

The value **2023-08-16** is returned.

```
select to_date('2023-08-16 10:54:36');
```

The value **NULL** is returned.

```
select to_date(null);
```

## 14.1.29 to\_date1

This function is used to convert a string in a specified format to a date value.

Similar function: [to\\_date](#). The **to\_date** function is used to return the year, month, and day in a time. The date format cannot be specified.

## Syntax

```
to_date1(string date, string format)
```

## Parameters

Table 14-26 Parameters

Parameter	Mandatory	Type	Description
date	Yes	STRING	String to be converted The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

Parameter	Mandatory	Type	Description
format	Yes	STRING	<p>Format of the date to be converted Constant of the STRING type. Extended date formats are not supported.</p> <p>The value is a combination of the time unit (year, month, day, hour, minute, and second) and any character.</p> <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>mm</b> indicates the month.</li><li>• <b>dd</b> indicates the day.</li><li>• <b>hh</b> indicates the hour.</li><li>• <b>mi</b> indicates the minute.</li><li>• <b>ss</b> indicates the second.</li></ul>

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.
- If the value of **format** is **NULL**, a date in the **yyyy-mm-dd** format is returned.

## Example Code

The value **2023-08-16 10:54:36** is returned.

```
select to_date1('2023-08-16 10:54:36','yyyy-mm-dd hh:miss');
```

The value **2023-08-16 00:00:00** is returned.

```
select to_date1('2023-08-16','yyyy-mm-dd');
```

The value **NULL** is returned.

```
select to_date1(null);
```

The value **2023-08-16** is returned.

```
select to_date1('2023-08-16 10:54:36');
```

## 14.1.30 to\_utc\_timestamp

This function is used to convert a timestamp in a given time zone to a UTC timestamp.

## Syntax

```
to_utc_timestamp(string timestamp, string timezone)
```

## Parameters

**Table 14-27** Parameters

Parameter	Mandatory	Type	Description
timestamp	Yes	DATE STRING TINYINT SMALLINT INT BIGINT	Time to be processed Date value of the DATE or STRING type, or timestamp of the TINYINT, SMALLINT, INT, or BIGINT type. The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
timezone	Yes	STRING	Time zone where the time to be converted belongs

## Return Values

The return value is of the BIGINT type.

### NOTE

- If the value of **timestamp** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **timestamp** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **timestamp** is **NULL**, **NULL** is returned.
- If the value of **timezone** is **NULL**, **NULL** is returned.

## Example Code

The value **1692028800000** is returned.

```
select to_utc_timestamp('2023-08-14 17:00:00','PST');
```

The value **NULL** is returned.

```
select to_utc_timestamp(null);
```

## 14.1.31 trunc

This function is used to reset a date to a specific format.

Resetting means returning to default values, where the default values for year, month, and day are **01**, and the default values for hour, minute, second, and millisecond are **00**.

## Syntax

```
trunc(string date, string format)
```

## Parameters

**Table 14-28** Parameters

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
format	Yes	STRING	Format of the date to be converted The value is a combination of the time unit (year, month, day, hour, minute, and second) and any character. <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>MM</b> indicates the month.</li></ul>

## Return Values

The return value is of the DATE type, in the **yyyy-mm-dd** format.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.
- If the value of **format** is **NULL**, **NULL** is returned.

## Example Code

The value **2023-08-01** is returned.

```
select trunc('2023-08-16', 'MM');
```

The value **2023-08-01** is returned.

```
select trunc('2023-08-16 10:54:36', 'MM');
```

The value **NULL** is returned.

```
select trunc(null, 'MM');
```

## 14.1.32 unix\_timestamp

This function is used to convert a date value to a numeric date value in UNIX format.

The function returns the first ten digits of the timestamp in normal UNIX format.

### Syntax

```
unix_timestamp(string timestamp, string pattern)
```

### Parameters

**Table 14-29** Parameters

Parameter	Mandatory	Type	Description
timestamp	No	DATE or STRING	Date to be converted The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>
pattern	No	STRING	Format to be converted If this parameter is left blank, the default format <b>yyyy-MM-dd hh:mm:ss</b> is used. The value is a combination of the time unit (year, month, day, hour, minute, and second) and any character. <ul style="list-style-type: none"><li>• <b>yyyy</b> indicates the year.</li><li>• <b>MM</b> indicates the month.</li><li>• <b>dd</b> indicates the day.</li><li>• <b>hh</b> indicates the hour.</li><li>• <b>mi</b> indicates the minute.</li><li>• <b>ss</b> indicates the second.</li></ul>

### Return Values

The return value is of the BIGINT type.

 NOTE

- If the value of **timestamp** is **NULL**, **NULL** is returned.
- If both **timestamp** and **pattern** are left blank, the timestamp represented by the number of seconds since 1970-01-01 00:00:00 is returned.

## Example Code

The value **1692149997** is returned.

```
select unix_timestamp('2023-08-16 09:39:57')
```

If the current system time is **2023-08-16 10:23:16**, **1692152596** is returned.

```
select unix_timestamp();
```

The value **1692115200** (2023-08-16 00:00:00) is returned.

```
select unix_timestamp("2023-08-16 10:56:45", "yyyy-MM-dd");
```

### Example table data

```
select timestamp1, unix_timestamp(timestamp1) as date1_unix_timestamp, timestamp2,
unix_timestamp(datetime1) as date2_unix_timestamp, timestamp3, unix_timestamp(timestamp1) as
date3_unix_timestamp from database_t; output:
```

timestamp1	date1_unix_timestamp	timestamp2	date2_unix_timestamp	timestamp3	date3_unix_timestamp
2023-08-02 00:00:00.123456789	1690905600000	2023-08-02 11:09:14	1690945754793	2023-01-11	
2023-08-03 00:00:00.123456789	1690992000000	2023-08-02 11:09:31	1690945771994	2023-02-11	
2023-08-04 00:00:00.123456789	1691078400000	2023-08-02 11:09:41	1690945781270	2023-03-11	
2023-08-05 00:00:00.123456789	1691164800000	2023-08-02 11:09:48	1690945788874	2023-04-11	
2023-08-06 00:00:00.123456789	1691251200000	2023-08-02 11:09:59	1690945799099	2023-05-11	
2023-08-06 00:00:00.123456789	1683734400000				

## 14.1.33 weekday

This function is used to return the day of the current week.

### Syntax

```
weekday (string date)
```

## Parameters

**Table 14-30** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the INT type.

### NOTE

- If Monday is used as the first day of a week, the value **0** is returned. For other weekdays, the return value increases in ascending order. For Sunday, the value **6** is returned.
- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

## Example Code

The value **2** is returned.

```
select weekday ('2023-08-16 10:54:36');
```

The value **NULL** is returned.

```
select weekday (null);
```

## 14.1.34 weekofyear

This function is used to return the week number (from 0 to 53) of a specified date.

## Syntax

```
weekofyear(string date)
```

## Parameters

**Table 14-31** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the INT type.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

## Example Code

The value **33** is returned.

```
select weekofyear('2023-08-16 10:54:36');
```

The value **NULL** is returned.

```
select weekofyear('20230816');
```

The value **NULL** is returned.

```
select weekofyear(null);
```

## 14.1.35 year

This function is used to return the year of a specified date.

## Syntax

```
year(string date)
```

## Parameters

**Table 14-32** Parameter

Parameter	Mandatory	Type	Description
date	Yes	DATE or STRING	Date that needs to be processed The following formats are supported: <ul style="list-style-type: none"><li>• yyyy-mm-dd</li><li>• yyyy-mm-dd hh:mi:ss</li><li>• yyyy-mm-dd hh:mi:ss.ff3</li></ul>

## Return Values

The return value is of the INT type.

### NOTE

- If the value of **date** is not of the DATE or STRING type, the error message "data type mismatch" is displayed.
- If the value of **date** is of the DATE or STRING type but is not in one of the supported formats, **NULL** is returned.
- If the value of **date** is **NULL**, **NULL** is returned.

## Example Code

The value **2023** is returned.

```
select year('2023-08-16 10:54:36');
```

The value **NULL** is returned.

```
select year('23-01-01');
```

The value **NULL** is returned.

```
select year('2023/08/16');
```

The value **NULL** is returned.

```
select year(null);
```

## 14.2 String Functions

### 14.2.1 Overview

[Table 14-33](#) lists the string functions supported by DLI.

Table 14-33 String functions

Function	Syntax	Value Type	Description
<b>ascii</b>	ascii(string <str>)	BIGINT	Returns the numeric value of the first character in a string.
<b>concat</b>	concat(array<T> <a>, array<T> <b>[,...]), concat(string <str1>, string <str2>[,...])	ARRAY or STRING	Returns a string concatenated from multiple input strings. This function can take any number of input strings.
<b>concat_ws</b>	concat_ws(string <separator>, string <str1>, string <str2>[,...]), concat_ws(string <separator>, array<string> <a>)	ARRAY or STRUCT	Returns a string concatenated from multiple input strings that are separated by specified separators.
<b>char_matchcount</b>	char_matchcount(string <str1>, string <str2>)	BIGINT	Returns the number of characters in str1 that appear in str2.
<b>encode</b>	encode(string <str>, string <charset>)	BINARY	Returns str encoded in charset format.
<b>find_in_set</b>	find_in_set(string <str1>, string <str2>)	BIGINT	Returns the position (starting from 1) of str1 in str2 separated by commas (,).
<b>get_json_object</b>	get_json_object(string <json>, string <path>)	STRING	Parses the JSON object in a specified JSON path. The function will return <b>NULL</b> if the JSON object is invalid.
<b>instr</b>	instr(string <str>, string <substr>)	INT	Returns the index of substr that appears earliest in str. It returns <b>NULL</b> if either of the arguments are <b>NULL</b> and returns <b>0</b> if substr does not exist in str. Note that the first character in str has index 1.
<b>instr1</b>	instr1(string <str1>, string <str2>[, bigint <start_position>[, bigint <nth_appearance>]])	BIGINT	Returns the position of str2 in str1.

Function	Syntax	Value Type	Description
<b>initcap</b>	initcap(string A)	STRING	Converts the first letter of each word of a string to upper case and all other letters to lower case.
<b>keyvalue</b>	keyvalue(string <str>,[string <split1>,string <split2>],) string <key>)	STRING	Splits str by split1, converts each group into a key-value pair by split2, and returns the value corresponding to the key.
<b>length</b>	length(string <str>)	BIGINT	Returns the length of a string.
<b>lengthb</b>	lengthb(string <str>)	STRING	Returns the length of a specified string in bytes.
<b>levenshtein</b>	levenshtein(string A, string B)	INT	Returns the Levenshtein distance between two strings, for example, <b>levenshtein('kitten','sitting') = 3.</b>
<b>locate</b>	locate(string <substr>, string <str>[, bigint <start_pos>])	BIGINT	Returns the position of substr in str.
<b>lower/lcase</b>	lower(string A) , lcase(string A)	STRING	Converts all characters of a string to the lower case.
<b>lpad</b>	lpad(string <str1>, int <length>, string <str2>)	STRING	Returns a string of a specified length. If the length of the given string (str1) is shorter than the specified length (length), the given string is left-padded with str2 to the specified length.
<b>ltrim</b>	ltrim([<trimChars>, ,] string <str>)	STRING	Trims spaces from the left hand side of a string.

Function	Syntax	Value Type	Description
<a href="#">parse_url</a>	<code>parse_url(string urlString, string partToExtract [, string keyToExtract])</code>	STRING	<p>Returns the specified part of a given URL. Valid values of <b>partToExtract</b> include <b>HOST</b>, <b>PATH</b>, <b>QUERY</b>, <b>REF</b>, <b>PROTOCOL</b>, <b>AUTHORITY</b>, <b>FILE</b>, and <b>USERINFO</b>.</p> <p>For example, <code>parse_url('http://facebook.com/path1/p.php?k1=v1&amp;k2=v2#Ref1', 'HOST')</code> returns <code>'facebook.com'</code>.</p> <p>When the second parameter is set to <b>QUERY</b>, the third parameter can be used to extract the value of a specific parameter. For example, <code>parse_url('http://facebook.com/path1/p.php?k1=v1&amp;k2=v2#Ref1', 'QUERY', 'k1')</code> returns <code>'v1'</code>.</p>
<a href="#">printf</a>	<code>printf(String format, Obj... args)</code>	STRING	Prints the input in a specific format.
<a href="#">regexp_count</a>	<code>regexp_count(string &lt;source&gt;, string &lt;pattern&gt;[, bigint &lt;start_position&gt;])</code>	BIGINT	Returns the number of substrings that match a specified pattern in the source, starting from the <b>start_position</b> position.
<a href="#">regexp_extract</a>	<code>regexp_extract(string &lt;source&gt;, string &lt;pattern&gt;[, bigint &lt;groupid&gt;])</code>	STRING	Matches the string <b>source</b> based on the <b>pattern</b> grouping rule and returns the string content that matches <b>groupid</b> .
<a href="#">replace</a>	<code>replace(string &lt;str&gt;, string &lt;old&gt;, string &lt;new&gt;)</code>	STRING	Replaces the substring that matches a specified string in a string with another string.

Function	Syntax	Value Type	Description
<a href="#">regexp_replace</a>	<ul style="list-style-type: none"> <li>For Spark 2.4.5: <code>regexp_replace(string &lt;source&gt;, string &lt;pattern&gt;, string &lt;replace_string&gt;)</code></li> <li>For Spark 3.3.1: <code>regexp_replace(string &lt;source&gt;, string &lt;pattern&gt;, string &lt;replace_string&gt;[, bigint &lt;occurrence&gt;])</code></li> </ul>	STRING	<ul style="list-style-type: none"> <li>For Spark 2.4.5: Replaces the substring that matches the pattern for the occurrence time in the source string and the substring that matches the pattern later with the specified string <b>replace_string</b> and returns the result string.</li> <li>For Spark 3.3.1: Replaces the substring that matches the pattern for the occurrence time in the source string and the substring that matches the pattern later with the specified string <b>replace_string</b> and returns the result string.</li> </ul>
<a href="#">regexp_replace1</a>	<code>regexp_replace1(string &lt;source&gt;, string &lt;pattern&gt;, string &lt;replace_string&gt;[, bigint &lt;occurrence&gt;])</code>	STRING	Replaces the substring that matches pattern for the occurrence time in the source string with the specified string <b>replace_string</b> and returns the result string.
<a href="#">regexp_instr</a>	<code>regexp_instr(string &lt;source&gt;, string &lt;pattern&gt;[, bigint &lt;start_position&gt;[, bigint &lt;occurrence&gt;[, bigint &lt;return_option&gt;]]])</code>	BIGINT	Returns the start or end position of the substring that matches a specified pattern for the occurrence time, starting from <b>start_position</b> in the source string.
<a href="#">regexp_substr</a>	<code>regexp_substr(string &lt;source&gt;, string &lt;pattern&gt;[, bigint &lt;start_position&gt;[, bigint &lt;occurrence&gt;]]])</code>	STRING	Returns the substring that matches a specified pattern for the occurrence time, starting from <b>start_position</b> in the source string.
<a href="#">repeat</a>	<code>repeat(string &lt;str&gt;, bigint &lt;n&gt;)</code>	STRING	Repeats a string for <i>N</i> times.
<a href="#">reverse</a>	<code>reverse(string &lt;str&gt;)</code>	STRING	Returns a string in reverse order.

Function	Syntax	Value Type	Description
<b>rpadd</b>	rpadd(string <str1>, int <length>, string <str2>)	STRING	Right-pads str1 with str2 to the specified length.
<b>rtrim</b>	rtrim([<trimChars>, ]string <str>), rtrim(trailing [<trimChars>] from <str>)	STRING	Trims spaces from the right hand side of a string.
<b>soundex</b>	soundex(string <str>)	STRING	Returns the soundex string from str, for example, <b>soundex('Miller') = M460</b> .
<b>space</b>	space(bigint <n>)	STRING	Returns a specified number of spaces.
<b>substr/ substring</b>	substr(string <str>, bigint <start_position>[, bigint <length>]), substring(string <str>, bigint <start_position>[, bigint <length>])	STRING	Returns the substring of str, starting from <b>start_position</b> and with a length of <b>length</b> .
<b>substring_index</b>	substring_index(string <str>, string <separator>, int <count>)	STRING	Truncates the string before the <b>count</b> separator of str. If the value of <b>count</b> is positive, the string is truncated from the left. If the value of <b>count</b> is negative, the string is truncated from the right.
<b>split_part</b>	split_part(string <str>, string <separator>, bigint <start>[, bigint <end>])	STRING	Splits a specified string based on a specified separator and returns a substring from the start to end position.
<b>translate</b>	translate(string char varchar input, string char varchar from, string char varchar to)	STRING	Translates the input string by replacing the characters or string specified by <b>from</b> with the characters or string specified by <b>to</b> . For example, replaces <b>bcd</b> in <b>abcde</b> with <b>BCD</b> using <b>translate("abcde", "bcd", "BCD")</b> .

Function	Syntax	Value Type	Description
<b>trim</b>	trim([<trimChars>, ]string <str>), trim([BOTH] [<trimChars>] from <str>)	STRING	Trims spaces from both ends of a string.
<b>upper/ucase</b>	upper(string A), ucase(string A)	STRING	Converts all characters of a string to the upper case.

## 14.2.2 ascii

This function is used to return the ASCII code of the first character in str.

### Syntax

```
ascii(string <str>)
```

### Parameters

Table 14-34 Parameter

Parameter	Mandatory	Type	Description
str	Yes	STRING	If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is automatically converted to the STRING type for calculation. Example: <b>ABC</b>

### Return Values

The return value is of the BIGINT type.

#### NOTE

- If the value of **str** is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of **str** is **NULL**, **NULL** is returned.

### Example Code

- Returns the ASCII code of the first character in string ABC. An example command is as follows:  
The value **97** is returned.

```
select ascii('ABC');
```

- The value of the input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select ascii(null);
```

## 14.2.3 concat

This function is used to concatenate arrays or strings.

### Syntax

If multiple arrays are used as the input, all elements in the arrays are connected to generate a new array.

```
concat(array<T> <a>, array<T> <b>[,...])
```

If multiple strings are used as the input, the strings are connected to generate a new string.

```
concat(string <str1>, string <str2>[,...])
```

### Parameters

- Using arrays as the input

**Table 14-35** Parameters

Parameter	Mandatory	Type	Description
a, b	Yes	STRING	Array In array<T>, <b>T</b> indicates the data type of the elements in the array. The elements in the array can be of any type. The data types of elements in arrays a and b must be the same. If the values of the elements in an array are <b>NULL</b> , the elements are involved in the operation.

- Using strings as the input

**Table 14-36** Parameters

Parameter	Mandatory	Type	Description
str1, str2	Yes	STRING	String If the value of the input parameter is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is automatically converted to the STRING type for calculation. For other types of values, an error is reported.

## Return Values

The return value is of the ARRAY or STRING type.

### NOTE

- If the return value is of the ARRAY type and any input array is **NULL**, **NULL** is returned.
- If the return value is of the STRING type and there is no parameter or any parameter is **NULL**, **NULL** is returned.

## Example Code

- Connect the array (1, 2) and array (2, -2). An example command is as follows:  
The value **[1, 2, 2, -2]** is returned.  

```
select concat(array(1, 2), array(2, -2));
```
- Any array is **NULL**. An example command is as follows:  
The value **NULL** is returned.  

```
select concat(array(10, 20), null);
```
- Connect strings ABC and DEF. An example command is as follows:  
The value **ABCDEF** is returned.  

```
select concat('ABC','DEF');
```
- The input is empty. An example command is as follows:  
The value **NULL** is returned.  

```
select concat();
```
- The value of any string is **NULL**. An example command is as follows:  
The value **NULL** is returned.  

```
select concat('abc', 'def', null);
```

## 14.2.4 concat\_ws

This function is used to return a string concatenated from multiple input strings that are separated by specified separators.

### Syntax

```
concat_ws(string <separator>, string <str1>, string <str2>[,...])
```

or

```
concat_ws(string <separator>, array<string> <a>)
```

Returns the result of joining all the strings in the parameters or the elements in an array using a specified separator.

## Parameters

**Table 14-37** Parameters

Parameter	Mandatory	Type	Description
separator	Yes	STRING	Separator of the STRING type
str1, str2	Yes	STRING	At least two strings must be specified. The value is of the STRING type. If the value is of the BIGINT, DECIMAL, DOUBLE, or DATETIME type, the value is implicitly converted to the STRING type for calculation.
a	Yes	ARRAY	The elements in an array are of the STRING type.

## Return Values

The return value is of the STRING or STRUCT type.

### NOTE

- If the value of **str1** or **str2** is not of the STRING, BIGINT, DECIMAL, DOUBLE, or DATETIME type, an error is reported.
- If the value of the parameter (string to be concatenated) is **NULL**, the parameter is ignored.
- If there is no input parameter (string to be concatenated), **NULL** is returned.

## Example Code

- Use a colon (:) to connect strings ABC and DEF. An example command is as follows:

The value **ABC:DEF** is returned.

```
select concat_ws(':', 'ABC', 'DEF');
```

- The value of any input parameter is **NULL**. An example command is as follows:

The value **avg:18** is returned.

```
select concat_ws(':', 'avg', null, '18');
```

- Use colons (:) to connect elements in the array ('name', 'lilei'). An example command is as follows:

The value **name:lilei** is returned.

```
select concat_ws(':', array('name', 'lilei'));
```

## 14.2.5 char\_matchcount

This parameter is used to return the number of characters in str1 that appear in str2.

### Syntax

```
char_matchcount(string <str1>, string <str2>)
```

### Parameters

**Table 14-38** Parameter

Parameter	Mandatory	Type	Description
str1, str2	Yes	STRING	str1 and str2 to be calculated

### Return Values

The return value is of the BIGINT type.

#### NOTE

If the value of **str1** or **str2** is **NULL**, **NULL** is returned.

### Example Code

The value **3** is returned.

```
select char_matchcount('abcz','abcde');
```

The value **NULL** is returned.

```
select char_matchcount(null,'abcde');
```

## 14.2.6 encode

This function is used to encode str in charset format.

### Syntax

```
encode(string <str>, string <charset>)
```

## Parameters

**Table 14-39** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	At least two strings must be specified. The value is of the STRING type. If the value is of the BIGINT, DECIMAL, DOUBLE, or DATETIME type, the value is implicitly converted to the STRING type for calculation.
charset	Yes	STRING	Encoding format. The options are <b>UTF-8</b> , <b>UTF-16</b> , <b>UTF-16LE</b> , <b>UTF-16BE</b> , <b>ISO-8859-1</b> , and <b>US-ASCII</b> .

## Return Values

The return value is of the BINARY type.

 **NOTE**

If the value of **str** or **charset** is **NULL**, **NULL** is returned.

## Example Code

- Encodes string abc in UTF-8 format. An example command is as follows:  
The value **abc** is returned.  

```
select encode("abc", "UTF-8");
```
- The value of any input parameter is **NULL**. An example command is as follows:  
The return value is **NULL**.  

```
select encode("abc", null);
```

### 14.2.7 find\_in\_set

This function is used to return the position (starting from 1) of str1 in str2 separated by commas (,).

## Syntax

```
find_in_set(string <str1>, string <str2>)
```

## Parameters

**Table 14-40** Parameters

Parameter	Mandatory	Type	Description
str1	Yes	STRING	String to be searched for
str2	Yes	STRING	String separated by a comma (,)

## Return Values

The return value is of the BIGINT type.

### NOTE

- If str1 cannot be matched in str2 or str1 contains commas (,), **0** is returned.
- If the value of **str1** or **str2** is **NULL**, **NULL** is returned.

## Example Code

- Search for the position of string **ab** in string **abc,123,ab,c**. An example command is as follows:

The value **3** is returned.

```
select find_in_set('ab', 'abc,123,ab,c');
```

- Search for the position of string **hi** in string **abc,123,ab,c**. An example command is as follows:

The value **0** is returned.

```
select find_in_set('hi', 'abc,123,ab,c');
```

- The value of any input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select find_in_set(null, 'abc,123,ab,c');
```

## 14.2.8 get\_json\_object

This function is used to parse the JSON object in a specified JSON path. The function will return **NULL** if the JSON object is invalid.

## Syntax

```
get_json_object(string <json>, string <path>)
```

## Parameters

**Table 14-41** Parameters

Parameter	Mandatory	Type	Description
json	Yes	STRING	Standard JSON object, in the <b>{Key:Value, Key:Value,...}</b> format.
path	Yes	STRING	Path of the object in JSON format, which starts with \$. The meanings of characters are as follows: <ul style="list-style-type: none"> <li>• \$ indicates the root node.</li> <li>• . indicates a subnode.</li> <li>• [] indicates the index of an array, which starts from 0.</li> <li>• * indicates the wildcard for []. The entire array is returned. * does not support escape.</li> </ul>

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **json** is empty or in invalid JSON format, **NULL** is returned.
- If the value of **json** is valid and **path** is specified, the corresponding string is returned.

## Example Code

- Extracts information from the JSON object **src\_json.json**. An example command 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" }
```

Extracts the information of the **owner** field and returns **Tony**.  

```
select get_json_object(jsonString, '$.owner');
```

Extracts the first array information of the **store.fruit** field and returns **{"weight":8,"type":"apple"}**.  

```
select get_json_object(jsonString, '$.store.fruit[0]');
```

Extracts information about a field that does not exist and returns **NULL**.  

```
select get_json_object(jsonString, '$.non_exist_key');
```
- Extracts information about an array JSON object. An example command is as follows:  
The value **22** is returned.  

```
select get_json_object({'array':[['a",11],[ "b",22],[ "c",33]]}, '$.array[1][1]');
```

The value **["h00", "h11", "h22"]** is returned.  

```
select get_json_object({'a':"b", "c":{"d":"e", "f":"g", "h":["h00", "h11", "h22"]}, "i":"j"}, '$.c.h[*]');
```

The value `["h00","h11","h22"]` is returned.

```
select get_json_object({'a':"b","c":{"d":"e","f":"g","h":["h00","h11","h22"],"i":"j"},".c.h'});
```

The value `h11` is returned.

```
select get_json_object({'a':"b","c":{"d":"e","f":"g","h":["h00","h11","h22"],"i":"j"},".c.h[1]});
```

- Extracts information from a JSON object with a period (.). An example command is as follows:

Create a table.

```
create table json_table (id string, json string);
```

Insert data into the table. The key contains a period (.).

```
insert into table json_table (id, json) values ("1", '{"city1":{"region":{"rid":6}}});
```

Insert data into the table. The key does not contain a period (.).

```
insert into table json_table (id, json) values ("2", '{"city1":{"region":{"rid":7}}});
```

Obtain the value of `rid`. If the key is `city1`, `6` is returned. Only `[ ]` can be used for parsing because a period (.) is included.

```
select get_json_object(json, "$['city1'].region['id']") from json_table where id =1;
```

Obtain the value of `rid`. If the key is `city1`, `7` is returned. You can use either of the following methods:

```
select get_json_object(json, "$['city1'].region['id']") from json_table where id =2;
```

```
select get_json_object(json, "$.city1.region['id']") from json_table where id =2;
```

- The `json` parameter is either empty or has an invalid format. An example command is as follows:

The value `NULL` is returned.

```
select get_json_object('',$.array[2]);
```

The value `NULL` is returned.

```
select get_json_object("'array':['a',1],'b':['c',3]',$.array[1][1]');
```

- A JSON string involves escape. An example command is as follows:

The value `3` is returned.

```
select get_json_object({'a':"\\3\\","b":"6"}, '$.a');
```

The value `3` is returned.

```
select get_json_object({'a':"\\3\\","b":"6"}, '$.a');
```

- A JSON object can contain the same key and can be parsed successfully.

The value `1` is returned.

```
select get_json_object({'b':"1","b":"2"}, '$.b');
```

- The result is output in the original sorting mode of the JSON string.

The value `{"b":"3","a":"4"}` is returned.

```
select get_json_object({'b':"3","a":"4"}, '$');
```

## 14.2.9 instr

This function is used to return the index of substr that appears earliest in str.

It returns `NULL` if either of the arguments are `NULL` and returns `0` if substr does not exist in str. Note that the first character in str has index 1.

Similar function: [instr1](#). The `instr1` function is used to calculate the position of the substring str2 in the string str1. The `instr1` function allows you to specify the start search position and the number of matching times.

## Syntax

```
instr(string <str>, string <substr>)
```

## Parameters

Table 14-42 Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	Target string to be searched for If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation. For other types of values, an error is reported.
substr	Yes	STRING	Substring to be matched If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation. For other types of values, an error is reported.

## Return Values

The return value is of the BIGINT type.

### NOTE

- If **str2** is not found in **str1**, **0** is returned.
- If **str2** is an empty string, the matching is always successful. For example, **select instr('abc','');** returns 1.
- If the value of **str1** or **str2** is **NULL**, **NULL** is returned.

## Example Code

- Returns the position of character b in string abc. An example command is as follows:

The value **2** is returned.

```
select instr('abc', 'b');
```

- The value of any input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select instr('abc', null)
```

## 14.2.10 instr1

This function is used to return the position of substring str2 in string str1.

Similar function: [instr](#). The **instr** function is used to return the index of substr that appears earliest in **str**. However, **instr** does not support specifying the start search position and matching times.

## Syntax

```
instr1(string <str1>, string <str2>[, bigint <start_position>[, bigint <nth_appearance>]])
```

## Parameters

**Table 14-43** Parameters

Parameter	Mandatory	Type	Description
str1	Yes	STRING	Target string to be searched for If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation. For other types of values, an error is reported.
str2	Yes	STRING	Substring to be matched If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation. For other types of values, an error is reported.
start_position	No	BIGINT	Sequence number of the character in str1 the search starts from. The default start position is position 1 (position of the first character). Negative numbers are not supported.
nth_appearance	No	BIGINT	Position of str2 that is matched for the nth_appearance time in str1. If the value is of another type or is less than or equal to 0, an error is reported.

## Return Values

The return value is of the BIGINT type.

### NOTE

- If **str2** is not found in **str1**, **0** is returned.
- If **str2** is an empty string, the matching is always successful.
- If the value of **str1**, **str2**, **start\_position**, or **nth\_appearance** is **NULL**, **NULL** is returned.

## Example Code

The value **10** is returned.

```
select instr('Tech on the net', 'h', 5, 1);
```

The value **2** is returned.

```
select instr('abc', 'b');
```

The value **NULL** is returned.

```
select instr('abc', null);
```

## 14.2.11 initcap

This function is used to convert the first letter of each word of a string to upper case and all other letters to lower case.

### Syntax

```
initcap(string A)
```

### Parameters

**Table 14-44** Parameter

Parameter	Mandatory	Type	Description
A	Yes	STRING	Text string to be converted

### Return Values

The return value is of the STRING type. In the string, the first letter of each word is capitalized, and the other letters are lowercased.

## Example Code

The value **Dli Sql** is returned.

```
SELECT initcap("dLI sql");
```

## 14.2.12 keyvalue

This function is used to split str by split1, convert each group into a key-value pair by split2, and return the value corresponding to the key.

### Syntax

```
keyvalue(string <str>,[string <split1>],[string <split2>],] string <key>)
```

## Parameters

**Table 14-45** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	String to be split
split1, split2	No	STRING	Strings used as separators to split the source string. If these two separators are not specified in the expression, the default values are ; for <b>split1</b> and : for <b>split2</b> . If there are multiple occurrences of split2 within a string that has been split by split1, the result is undefined.
key	No	BIGINT	The corresponding value when the string is split using <b>split1</b> and <b>split2</b> .

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **split1** or **split2** is **NULL**, **NULL** is returned.
- If the value of **str** or **key** is **NULL** or no matching key is found, **NULL** is returned.
- If multiple key-value pairs are matched, the value corresponding to the first matched key is returned.

## Example Code

The value **2** is returned.

```
select keyvalue('a:1;b:2', 'b');
```

The value **2** is returned.

```
select keyvalue("\;abc:1\;def:2", "\;";";"def");
```

## 14.2.13 length

This function is used to return the length of a string.

Similar function: **lengthb**. The **lengthb** function is used to return the length of string **str** in bytes and return a value of the STRING type.

## Syntax

```
length(string <str>)
```

## Parameters

**Table 14-46** Parameter

Parameter	Mandatory	Type	Description
str	Yes	STRING	Target string to be searched for If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation. For other types of values, an error is reported.

## Return Values

The return value is of the BIGINT type.

### NOTE

- If the value of **str** is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of **str** is **NULL**, **NULL** is returned.

## Example Code

- Calculate the length of string abc. An example command is as follows:  
The value **3** is returned.

```
select length('abc');
```

- The value of the input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select length(null);
```

## 14.2.14 lengthb

This function is used to return the length of a specified string in bytes.

Similar function: [length](#). The **length** function is used to return the length of a string and return a value of the BIGINT type.

## Syntax

```
lengthb(string <str>)
```

## Parameters

**Table 14-47** Parameter

Parameter	Mandatory	Type	Description
str	Yes	STRING	Input string

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **str** is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of **str** is **NULL**, **NULL** is returned.

## Example Code

The value **5** is returned.

```
select lengthb('hello');
```

The value **NULL** is returned.

```
select lengthb(null);
```

## 14.2.15 levenshtein

This function is used to returns the Levenshtein distance between two strings, for example, **levenshtein('kitten','sitting') = 3**.

### NOTE

Levenshtein distance is a type of edit distance. It indicates the minimum number of edit operations required to convert one string into another.

## Syntax

```
levenshtein(string A, string B)
```

## Parameters

**Table 14-48** Parameter

Parameter	Mandatory	Type	Description
A, B	Yes	STRING	String to be entered for calculating the Levenshtein distance

## Return Values

The return value is of the INT type.

## Example Code

The value **3** is returned.

```
SELECT levenshtein('kitten','sitting');
```

## 14.2.16 locate

This function is used to return the position of substr in str. You can specify the starting position of your search using "start\_pos," which starts from 1.

## Syntax

```
locate(string <substr>, string <str>[, bigint <start_pos>])
```

## Parameters

**Table 14-49** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	Target string to be searched for If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation. For other types of values, an error is reported.
substr	Yes	STRING	Substring to be matched If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation. For other types of values, an error is reported.
start_pos	No	BIGINT	Start position for the search

## Return Values

The return value is of the BIGINT type.

### NOTE

- If substr cannot be matched in str, **0** is returned.
- If the value of **str** or **substr** is **NULL**, **NULL** is returned.
- If the value of **start\_pos** is **NULL**, **0** is returned.

## Example Code

- Search for the position of string **ab** in string **abhiab**. An example command is as follows:

The value **1** is returned.

```
select locate('ab', 'abhiab');
```

The value **5** is returned.

```
select locate('ab', 'abhiab', 2);
```

The value **0** is returned.

```
select locate('ab', 'abhiab', null);
```

- Search for the position of string **hi** in string **hanmeimei and lilei**. An example command is as follows:

The value **0** is returned.

```
select locate('hi', 'hanmeimei and lilei');
```

## 14.2.17 lower/lcase

This function is used to convert all characters of a string to the lower case.

### Syntax

```
lower(string A) / lcase(string A)
```

### Parameters

**Table 14-50** Parameter

Parameter	Mandatory	Type	Description
A	Yes	STRING	Text string to be converted

### Return Values

The return value is of the STRING type.

#### NOTE

- If the value of the input parameter is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of the input parameter is **NULL**, **NULL** is returned.

## Example Code

Converts uppercase characters in a string to the lower case. An example command is as follows:

The value **abc** is returned.

```
select lower('ABC');
```

The value of the input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select lower(null);
```

## 14.2.18 lpad

This function is used to return a string of a specified length. If the length of the given string (**str1**) is shorter than the specified length (**length**), the given string is left-padded with **str2** to the specified length.

### Syntax

```
lpad(string <str1>, int <length>, string <str2>)
```

### Parameters

**Table 14-51** Parameters

Parameter	Mandatory	Type	Description
str1	Yes	STRING	String to be left-padded
length	Yes	STRING	Number of digits to be padded to the left
str2	No	STRING	String used for padding

### Return Values

The return value is of the STRING type.

#### NOTE

- If the value of **length** is smaller than the number of digits in **str1**, the string whose length is truncated from the left of **str1** is returned.
- If the value of **length** is **0**, an empty string is returned.
- If there is no input parameter or the value of any input parameter is **NULL**, **NULL** is returned.

### Example Code

- Uses string **ZZ** to left pad string **abcdefgh** to 10 digits. An example command is as follows:

The value **ZZabcdefgh** is returned.

```
select lpad('abcdefgh', 10, 'ZZ');
```

- Uses string **ZZ** to left pad string **abcdefgh** to 5 digits. An example command is as follows:

The value **abcde** is returned.

```
select lpad('abcdefgh', 5, 'ZZ');
```

- The value of **length** is **0**. An example command is as follows:  
An empty string is returned.

```
select lpad('abcdefgh', 0, 'ZZ');
```

- The value of any input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select lpad(null,0, 'ZZ');
```

## 14.2.19 ltrim

This function is used to remove characters from the left of **str**.

- If **trimChars** is not specified, spaces are removed by default.
- If **trimChars** is specified, the function removes the longest possible substring from the left end of **str** that consists of characters in the **trimChars** set.

Similar functions:

- **rtrim**. This function is used to remove characters from the right of **str**.
- **trim**. This function is used to remove characters from the left and right of **str**.

## Syntax

```
ltrim([<trimChars>] string <str>)
```

## Parameters

Table 14-52 Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	String from which characters on the left are to be removed. If the value is of the BIGINT, DECIMAL, DOUBLE, or DATETIME type, the value is implicitly converted to the STRING type for calculation.
trimChars	Yes	STRING	Characters to be removed

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **str** is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of **str** or **trimChars** is **NULL**, **NULL** is returned.

## Example Code

- Removes spaces on the left of string **abc**. An example command is as follows:  
The value **stringabc** is returned.

```
select ltrim('  abc');
```

It is equivalent to the following statement:

```
select trim(leading from '  abc');
```

**leading** indicates that the leading spaces in a string are removed.

- The value of the input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select ltrim(null);
select ltrim('xy', null);
select ltrim(null, 'xy');
```

- Removes all substrings from the left end of the string **yxlycyxx** that consist of characters in the set **xy**.

The function returns **lycyxx**, as any substring starting with **x** or **y** from the left end is removed.

```
select ltrim('xy', 'yxlycyxx');
```

It is equivalent to the following statement:

```
select trim(leading 'xy' from 'yxlycyxx');
```

## 14.2.20 parse\_url

This character is used to return the specified part of a given URL. Valid values of **partToExtract** include **HOST**, **PATH**, **QUERY**, **REF**, **PROTOCOL**, **AUTHORITY**, **FILE**, and **USERINFO**.

For example, **parse\_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST')** returns **'facebook.com'**.

When the second parameter is set to **QUERY**, the third parameter can be used to extract the value of a specific parameter. For example, **parse\_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1')** returns **'v1'**.

### Syntax

```
parse_url(string urlString, string partToExtract [, string keyToExtract])
```

### Parameters

Table 14-53 Parameters

Parameter	Mandatory	Type	Description
urlString	Yes	STRING	URL. If the URL is invalid, an error is reported.
partToExtract	Yes	STRING	The value is case-insensitive and can be <b>HOST</b> , <b>PATH</b> , <b>QUERY</b> , <b>REF</b> , <b>PROTOCOL</b> , <b>AUTHORITY</b> , <b>FILE</b> , or <b>USERINFO</b> .
keyToExtract	No	STRING	If the value of <b>partToExtract</b> is <b>QUERY</b> , the value is obtained based on the key.

## Return Values

The return value is of the STRING type. The return rules are as follows:

### NOTE

- If the value of **urlString**, **partToExtract**, or **keyToExtract** is **NULL**, **NULL** is returned.
- If the value of **partToExtract** does not meet requirements, an error is reported.

## Example Code

The value **example.com** is returned.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'HOST');
```

The value **/over/there/index.dtb** is returned.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'PATH');
```

The value **animal** is returned.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'QUERY', 'type');
```

The value **nose** is returned.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'REF');
```

The value **file** is returned.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'PROTOCOL');
```

The value **username@example.com:8042** is returned.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'AUTHORITY');
```

The value **username** is returned.

```
select parse_url('file://username@example.com:666/over/there/index.dtb?
type=animal&name=narwhal#nose', 'USERINFO');
```

## 14.2.21 printf

This function is used to print the input in a specific format.

### Syntax

```
printf(String format, Obj... args)
```

## Parameters

**Table 14-54** Parameters

Parameter	Mandatory	Type	Description
format	Yes	STRING	Output format
Obj	No	STRING	Other input parameters

## Return Values

The return value is of the STRING type.

The value is returned after the parameters that filled in **Obj** are specified for **format**.

## Example Code

The string **name: user1, age: 20, gender: female, place of origin: city 1** is returned.

```
SELECT printf('Name: %s, Age: %d, Gender: %s, Place of origin: %s', "user1", 20, "Female", "City 1");
```

## 14.2.22 regexp\_count

This function is used to return the number of substrings that match a specified pattern in the source, starting from the **start\_position** position.

## Syntax

```
regexp_count(string <source>, string <pattern>[, bigint <start_position>])
```

## Parameters

**Table 14-55** Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	String to be searched for. For other types, an error is reported.
pattern	Yes	STRING	Constant or regular expression of the STRING type. Pattern to be matched. If the value of this parameter is an empty string or of other types, an error is reported.

Parameter	Mandatory	Type	Description
start_position	No	BIGINT	Constant of the BIGINT type. The value must be greater than 0. If the value is of another type or is less than or equal to 0, an error is reported. If this parameter is not specified, the default value <b>1</b> is used, indicating that the matching starts from the first character of <b>source</b> .

## Return Values

The return value is of the BIGINT type.

### NOTE

- If no match is found, **0** is returned.
- If the value of **source** or **pattern** is **NULL**, **NULL** is returned.

## Example Code

The value **4** is returned.

```
select regexp_count('ab0a1a2b3c', '[0-9]');
```

The value **3** is returned.

```
select regexp_count('ab0a1a2b3c', '[0-9]', 4);
```

The value **null** is returned.

```
select regexp_count('ab0a1a2b3c', null);
```

## 14.2.23 regexp\_extract

This function is used to match the string **source** based on the **pattern** grouping rule and return the string content that matches **groupid**.

## Syntax

```
regexp_extract(string <source>, string <pattern>[, bigint <groupid>])
```

## Parameters

**Table 14-56** Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	String to be split

Parameter	Mandatory	Type	Description
pattern	Yes	STRING	Constant or regular expression of the STRING type. Pattern to be matched.
groupid	No	BIGINT	Constant of the BIGINT type. The value must be greater than or equal to 0.

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **pattern** is an empty string or there is no group in **pattern**, an error is reported.
- If the value of **groupid** is not of the BIGINT type or is less than 0, an error is reported.
- If this parameter is not specified, the default value **1** is used, indicating that the first group is returned.
- If the value of **groupid** is **0**, the substring that meets the entire pattern is returned.
- If the value of **source**, **pattern**, or **groupid** is **NULL**, **NULL** is returned.

## Example Code

Splits **basketball** by **bas(.\*)(ball)**. The value **ket** is returned.

```
select regexp_extract('basketball', 'bas(.*)(ball)');
```

The value **basketball** is returned.

```
select regexp_extract('basketball', 'bas(.*)(ball)',0);
```

The value **99** is returned. When submitting SQL statements for regular expression calculation on DLI, two backslashes (\) are used as escape characters.

```
select regexp_extract('8d99d8', '8d(\\d+)d8');
```

The value **[Hello]** is returned.

```
select regexp_extract('[Hello] hello', '([^\x{00}-\x{ff}]+)');
```

The value **Hello** is returned.

```
select regexp_extract('[Hello] hello', '([\x{4e00}-\x{9fa5}]+)');
```

## 14.2.24 replace

This function is used to replace the part in a specified string that is the same as the string **old** with the string **new** and return the result.

If the string has no same characters as the string **old**, **str** is returned.

## Syntax

```
replace(string <str>, string <old>, string <new>)
```

## Parameters

**Table 14-57** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	String to be replaced
old	Yes	STRING	String to be compared
new	Yes	STRING	String after replacement

## Return Values

The return value is of the STRING type.

 **NOTE**

If the value of any input parameter is **NULL**, **NULL** is returned.

## Example Code

The value **AA123AA** is returned.

```
select replace('abc123abc','abc','AA');
```

The value **NULL** is returned.

```
select replace('abc123abc',null,'AA');
```

### 14.2.25 regexp\_replace

This function has slight variations in its functionality depending on the version of Spark being used.

- For Spark 2.4.5 or earlier: Replaces the substring that matches **pattern** in the string **source** with the specified string **replace\_string** and returns the result string.
- For Spark 3.1.1: Replaces the substring that matches the pattern for the occurrence time in the source string and the substring that matches the pattern later with the specified string **replace\_string** and returns the result string.

Similar function: [regexp\\_replace1](#). The **regexp\_replace1** function is used to replace the substring that matches pattern for the occurrence time in the source string with the specified string **replace\_string** and return the result string. However, the **egexp\_replace1** function applies only to Spark 2.4.5 or earlier.

The **regexp\_replace1** function is applicable to Spark 2.4.5. It supports specifying the **occurrence** parameter, whereas the **regexp\_replace** function does not support it.

## Syntax

- Spark 2.4.5 or earlier  
`regexp_replace(string <source>, string <pattern>, string <replace_string>)`
- Spark 3.1.1  
`regexp_replace(string <source>, string <pattern>, string <replace_string>[, bigint <occurrence>])`

## Parameters

Table 14-58 Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	String to be replaced
pattern	Yes	STRING	Constant or regular expression of the STRING type. Pattern to be matched. For more guidelines on writing regular expressions, refer to the regular expression specifications. If the value of this parameter is an empty string, an error is reported.
replace_string	Yes	STRING	String that replaces the one matching the <b>pattern</b> parameter
occurrence	No	BIGINT	The value must be greater than or equal to 1, indicating that the string that is matched for the occurrence time is replaced with <b>replace_string</b> . When the value is <b>1</b> , all matched substrings are replaced. If the value is of another type or is less than 1, an error is reported. The default value is <b>1</b> . <b>NOTE</b> This parameter is available only when Spark 3.1.1 is used.

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **pattern** is an empty string or there is no group in **pattern**, an error is reported.
- If a group that does not exist is referenced, the group is not replaced.
- If the value of **replace\_string** is **NULL** and the pattern is matched, **NULL** is returned.
- If the value of **replace\_string** is **NULL** but the pattern is not matched, **NULL** is returned.
- If the value of **source**, **pattern**, or **occurrence** is **NULL**, **NULL** is returned.

## Example Code

- For Spark 2.4.5 or earlier  
The value **num-num** is returned.  

```
SELECT regexp_replace('100-200', '(\d+)', 'num');
```
- For Spark 3.1.1  
The value **2222** is returned.  

```
select regexp_replace('abcd', '[a-z]', '2');
```

  
The value **2222** is returned.  

```
select regexp_replace('abcd', '[a-z]', '2', 1);
```

  
The value **a222** is returned.  

```
select regexp_replace('abcd', '[a-z]', '2', 2);
```

  
The value **ab22** is returned.  

```
select regexp_replace('abcd', '[a-z]', '2', 3);
```

  
The value **abc2** is returned.  

```
select regexp_replace('abcd', '[a-z]', '2', 4);
```

### 14.2.26 regexp\_replace1

This function is used to replace the substring that matches pattern for the occurrence time in the source string with the specified string **replace\_string** and return the result string.

#### NOTE

This function applies only to Spark 2.4.5 or earlier.

Similar function: [regexp\\_replace](#). The **regexp\_replace** function has slightly different functionalities for different versions of Spark. For details, see [regexp\\_replace](#).

## Syntax

```
regexp_replace1(string <source>, string <pattern>, string <replace_string>[, bigint <occurrence>])
```

## Parameters

**Table 14-59** Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	String to be replaced
pattern	Yes	STRING	Constant or regular expression of the STRING type. Pattern to be matched. For more guidelines on writing regular expressions, refer to the regular expression specifications. If the value of this parameter is an empty string, an error is reported.

Parameter	Mandatory	Type	Description
replace_string	Yes	STRING	String that replaces the one matching the <b>pattern</b> parameter
occurrence	No	BIGINT	The value must be greater than or equal to 1, indicating that the string that is matched for the occurrence time is replaced with <b>replace_string</b> . When the value is <b>1</b> , all matched substrings are replaced. If the value is of another type or is less than 1, an error is reported. The default value is <b>1</b> .

## Return Values

The return value is of the STRING type.

### NOTE

- If a group that does not exist is referenced, the group is not replaced.
- If the value of **replace\_string** is **NULL** and the pattern is matched, **NULL** is returned.
- If the value of **replace\_string** is **NULL** but the pattern is not matched, **NULL** is returned.
- If the value of **source**, **pattern**, or **occurrence** is **NULL**, **NULL** is returned.

## Example Code

The value **2222** is returned.

```
select regexp_replace1('abcd', '[a-z]', '2');
```

The value **2bcd** is returned.

```
select regexp_replace1('abcd', '[a-z]', '2', 1);
```

The value **a2cd** is returned.

```
select regexp_replace1('abcd', '[a-z]', '2', 2);
```

The value **ab2d** is returned.

```
select regexp_replace1('abcd', '[a-z]', '2', 3);
```

The value **abc2** is returned.

```
select regexp_replace1('abcd', '[a-z]', '2', 4);
```

## 14.2.27 regexp\_instr

This function is used to return the start or end position of the substring that matches a specified pattern for the occurrence time, starting from **start\_position** in the string **source**.

## Syntax

```
regexp_instr(string <source>, string <pattern>[,bigint <start_position>[, bigint <occurrence>[, bigint <return_option>]]])
```

## Parameters

**Table 14-60** Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	Source string
pattern	Yes	STRING	Constant or regular expression of the STRING type. Pattern to be matched. If the value of this parameter is an empty string, an error is reported.
start_position	No	BIGINT	Constant of the BIGINT type. Start position of the search. If it is not specified, the default value <b>1</b> is used.
occurrence	No	BIGINT	Constant of the BIGINT type. It indicates the specified number of matching times. If this parameter is not specified, the default value <b>1</b> is used, indicating that the first occurrence position is searched.
return_option	No	BIGINT	Constant of the BIGINT type. This parameter indicates the location to be returned. The value can be <b>0</b> or <b>1</b> . If this parameter is not specified, the default value <b>0</b> is used. If this parameter is set to a value of another type or a value that is not allowed, an error message is returned. The value <b>0</b> indicates that the start position of the match is returned, and the value <b>1</b> indicates that the end position of the match is returned.

## Return Values

The return value is of the BIGINT type. **return\_option** indicates the start or end position of the matched substring in **source**.

### NOTE

- If the value of **pattern** is an empty string, an error is reported.
- If the value of **start\_position** or **occurrence** is not of the BIGINT type or is less than or equal to 0, an error is reported.
- If the value of **source**, **pattern**, **start\_position**, **occurrence**, or **return\_option** is **NULL**, **NULL** is returned.

## Example Code

The value **6** is returned.

```
select regexp_instr('a1b2c3d4', '[0-9]', 3, 2);
```

The value **NULL** is returned.

```
select regexp_instr('a1b2c3d4', null, 3, 2);
```

## 14.2.28 regexp\_substr

This function is used to return the substring that matches a specified pattern for the occurrence time, starting from **start\_position** in the string **source**.

### Syntax

```
regexp_substr(string <source>, string <pattern>[, bigint <start_position>[, bigint <occurrence>]])
```

### Parameters

**Table 14-61** Parameters

Parameter	Mandatory	Type	Description
source	Yes	STRING	String to be searched for
pattern	Yes	STRING	Constant or regular expression of the STRING type. Pattern to be matched.
start_position	No	BIGINT	Start position. The value must be greater than 0. If this parameter is not specified, the default value <b>1</b> is used, indicating that the matching starts from the first character of <b>source</b> .
occurrence	No	BIGINT	The value is a constant of the BIGINT type, which must be greater than 0. If it is not specified, the default value <b>1</b> is used, indicating that the substring matched for the first time is returned.

### Return Values

The return value is of the STRING type.

#### NOTE

- If the value of **pattern** is an empty string, an error is reported.
- If no match is found, **NULL** is returned.
- If the value of **start\_position** or **occurrence** is not of the BIGINT type or is less than or equal to 0, an error is reported.
- If the value of **source**, **pattern**, **start\_position**, **occurrence**, or **return\_option** is **NULL**, **NULL** is returned.

## Example Code

The value **a** is returned.

```
select regexp_substr('a1b2c3', '[a-z]');
```

The value **b** is returned.

```
select regexp_substr('a1b2c3', '[a-z]', 2, 1);
```

The value **c** is returned.

```
select regexp_substr('a1b2c3', '[a-z]', 2, 2);
```

The value **NULL** is returned.

```
select regexp_substr('a1b2c3', null);
```

## 14.2.29 repeat

This function is used to return the string after **str** is repeated for **n** times.

### Syntax

```
repeat(string <str>, bigint <n>)
```

### Parameters

**Table 14-62** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation.
n	Yes	BIGINT	Number used for repetition

### Return Values

The return value is of the STRING type.

#### NOTE

- If the value of **str** is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of **n** is empty, an error is reported.
- If the value of **str** or **n** is **NULL**, **NULL** is returned.

### Example Code

The value **123123** is returned after the string **123** is repeated twice.

```
SELECT repeat('123', 2);
```

## 14.2.30 reverse

This function is used to return a string in reverse order.

### Syntax

```
reverse(string <str>)
```

### Parameters

**Table 14-63** Parameter

Parameter	Mandatory	Type	Description
str	Yes	STRING	If the value is of the BIGINT, DOUBLE, DECIMAL, or DATETIME type, the value is implicitly converted to the STRING type for calculation.

### Return Values

The return value is of the STRING type.

#### NOTE

- If the value of **str** is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of **str** is **NULL**, **NULL** is returned.

### Example Code

The value **LQS krapS** is returned.

```
SELECT reverse('Spark SQL');
```

The value **[3,4,1,2]** is returned.

```
SELECT reverse(array(2, 1, 4, 3));
```

## 14.2.31 rpad

This function is used to right pad **str1** with **str2** to the specified length.

### Syntax

```
rpad(string <str1>, int <length>, string <str2>)
```

## Parameters

**Table 14-64** Parameters

Parameter	Mandatory	Type	Description
str1	Yes	STRING	String to be right-padded
length	Yes	INT	Number of digits to be padded to the right
str2	Yes	STRING	String used for padding

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **length** is smaller than the number of digits in **str1**, the string whose length is truncated from the left of **str1** is returned.
- If the value of **length** is **0**, an empty string is returned.
- If there is no input parameter or the value of any input parameter is **NULL**, **NULL** is returned.

## Example Code

The value **hi???** is returned.

```
SELECT rpad('hi', 5, '?');
```

The value **h** is returned.

```
SELECT rpad('hi', 1, '?');
```

## 14.2.32 rtrim

This function is used to remove characters from the right of **str**.

- If **trimChars** is not specified, spaces are removed by default.
- If **trimChars** is specified, the function removes the longest possible substring from the right end of **str** that consists of characters in the **trimChars** set.

Similar functions:

- **ltrim**. This function is used to remove characters from the left of **str**.
- **trim**. This function is used to remove characters from the left and right of **str**.

## Syntax

```
rtrim([<trimChars>, ]string <str>)
```

or

```
rtrim(trailing [<trimChars>] from <str>)
```

## Parameters

**Table 14-65** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	String from which characters on the right are to be removed. If the value is of the BIGINT, DECIMAL, DOUBLE, or DATETIME type, the value is implicitly converted to the STRING type for calculation.
trimChars	Yes	STRING	Characters to be removed

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **str** is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of **str** or **trimChars** is **NULL**, **NULL** is returned.

## Example Code

- Removes spaces on the right of the string **yxabcxx**. An example command is as follows:

The value **yxabcxx** is returned.

```
select rtrim('yxabcxx ');
```

It is equivalent to the following statement:

```
select trim(trailing from ' yxabcxx ');
```

- Removes all substrings from the right end of the string **yxabcxx** that consist of characters in the set **xy**.

The function returns **yxabc**, as any substring starting with **x** or **y** from the right end is removed.

```
select rtrim('xy', 'yxabcxx');
```

It is equivalent to the following statement:

```
select trim(trailing 'xy' from 'yxabcxx');
```

- The value of the input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select rtrim(null);  
select rtrim('yxabcxx', 'null');
```

## 14.2.33 soundex

This function is used to return the soundex string from **str**, for example, **soundex('Miller') = M460**.

### Syntax

```
soundex(string <str>)
```

### Parameters

**Table 14-66** Parameter

Parameter	Mandatory	Type	Description
str	Yes	STRING	String to be converted

### Return Values

The return value is of the STRING type.

 **NOTE**

If the value of **str** is **NULL**, **NULL** is returned.

### Example Code

The value **M460** is returned.

```
SELECT soundex('Miller');
```

## 14.2.34 space

This function is used to return a specified number of spaces.

### Syntax

```
space(bigint <n>)
```

### Parameters

**Table 14-67** Parameter

Parameter	Mandatory	Type	Description
n	Yes	BIGINT	Number of spaces

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **n** is empty, an error is reported.
- If the value of **n** is **NULL**, **NULL** is returned.

## Example Code

The value **6** is returned.

```
select length(space(6));
```

## 14.2.35 substr/substring

This function is used to return the substring of **str**, starting from **start\_position** and with a length of **length**.

## Syntax

```
substr(string <str>, bigint <start_position>[, bigint <length>])
```

or

```
substring(string <str>, bigint <start_position>[, bigint <length>])
```

## Parameters

**Table 14-68** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	If the value is of the BIGINT, DECIMAL, DOUBLE, or DATETIME type, the value is implicitly converted to the STRING type for calculation.
start_position	Yes	BIGINT	Start position. The default value is <b>1</b> . If the value is positive, the substring is returned from the left. If the value is negative, the substring is returned from the right.
length	No	BIGINT	Length of the substring. The value must be greater than 0.

## Return Values

The return value is of the STRING type.

 NOTE

- If the value of **str** is not of the STRING, BIGINT, DECIMAL, DOUBLE, or DATETIME type, an error is reported.
- If the value of **length** is not of the BIGINT type or is less than or equal to 0, an error is reported.
- When the **length** parameter is omitted, a substring that ends with **str** is returned.
- If the value of **str**, **start\_position**, or **length** is NULL, NULL is returned.

## Example Code

The value **k SQL** is returned.

```
SELECT substr('Spark SQL', 5);
```

The value **SQL** is returned.

```
SELECT substr('Spark SQL', -3);
```

The value **k** is returned.

```
SELECT substr('Spark SQL', 5, 1);
```

## 14.2.36 substring\_index

This function is used to truncate the string before the **count** separator of **str**. If the value of **count** is positive, the string is truncated from the left. If the value of **count** is negative, the string is truncated from the right.

## Syntax

```
substring_index(string <str>, string <separator>, int <count>)
```

## Parameters

Table 14-69 Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	String to be truncated
separator	Yes	STRING	Separator of the STRING type
count	No	INT	Position of the delimiter

## Return Values

The return value is of the STRING type.

 NOTE

If the value of any input parameter is NULL, NULL is returned.

## Example Code

The value **hello.world** is returned.

```
SELECT substring_index('hello.world.people', '.', 2);
```

The value **world.people** is returned.

```
select substring_index('hello.world.people', '.', -2);
```

## 14.2.37 split\_part

This function is used to split a specified string based on a specified separator and return a substring from the start to end position.

### Syntax

```
split_part(string <str>, string <separator>, bigint <start>[, bigint <end>])
```

### Parameters

**Table 14-70** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	String to be split
separator	Yes	STRING	Constant of the STRING type. Separator used for splitting. The value can be a character or a string.
start	Yes	STRING	Constant of the BIGINT type. The value must be greater than 0. Start number of the returned part, starting from 1.
end	No	BIGINT	Constant of the BIGINT type. The value must be greater than or equal to the value of <b>start</b> . End number of the returned part and can be omitted. The default value indicates that the value is the same as that of <b>start</b> , and the part specified by <b>start</b> is returned.

### Return Values

The return value is of the STRING type.

 NOTE

- If the value of **start** is greater than the actual number of segments after splitting, for example, if there are four segments after string splitting and the value of **start** is greater than 4, an empty string is returned.
- If **separator** does not exist in **str** and **start** is set to 1, the entire **str** is returned. If the value of **str** is an empty string, an empty string is output.
- If the value of **separator** is an empty string, the original string **str** is returned.
- If the value of **end** is greater than the number of segments, the substring starting from **start** is returned.
- If the value of **str** is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of **start** or **end** is not a constant of the BIGINT type, an error is reported.
- If the value of any parameter except **separator** is **NULL**, **NULL** is returned.

## Example Code

The value **aa** is returned.

```
select split_part('aa,bb,cc,dd', ',', 1);
```

The value **aa,bb** is returned.

```
select split_part('aa,bb,cc,dd', ',', 1, 2);
```

An empty string is returned.

```
select split_part('aa,bb,cc,dd', ',', 10);
```

The value **aa,bb,cc,dd** is returned.

```
select split_part('aa,bb,cc,dd', ':', 1);
```

An empty string is returned.

```
select split_part('aa,bb,cc,dd', ':', 2);
```

The value **aa,bb,cc,dd** is returned.

```
select split_part('aa,bb,cc,dd', ',', 1);
```

The value **bb,cc,dd** is returned.

```
select split_part('aa,bb,cc,dd', ',', 2, 6);
```

The value **NULL** is returned.

```
select split_part('aa,bb,cc,dd', ',', null);
```

## 14.2.38 translate

This function is used to translate the input string by replacing the characters or string specified by **from** with the characters or string specified by **to**.

For example, it replaces **bcd** in **abcde** with **BCD**.

```
translate("abcde", "bcd", "BCD")
```

## Syntax

```
translate(string|char|varchar input, string|char|varchar from, string|char|varchar to)
```

## Parameters

**Table 14-71** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	String to be truncated
separator	Yes	STRING	Separator of the STRING type
count	No	INT	Position of the delimiter

## Return Values

The return value is of the STRING type.

 **NOTE**

If the value of any input parameter is **NULL**, **NULL** is returned.

## Example Code

The value **A1B2C3** is returned.

```
SELECT translate('AaBbCc', 'abc', '123');
```

## 14.2.39 trim

This function is used to remove characters from the left and right of **str**.

- If **trimChars** is not specified, spaces are removed by default.
- If **trimChars** is specified, the function removes the longest possible substring from both the left and right ends of **str** that consists of characters in the **trimChars** set.

Similar functions:

- **ltrim**. This function is used to remove characters from the left of **str**.
- **rtrim**. This function is used to remove characters from the right of **str**.

## Syntax

```
trim([<trimChars>],string <str>)
```

or

```
trim([BOTH] [<trimChars>] from <str>)
```

## Parameters

**Table 14-72** Parameters

Parameter	Mandatory	Type	Description
str	Yes	STRING	String from which characters on both left and right are to be removed If the value is of the BIGINT, DECIMAL, DOUBLE, or DATETIME type, the value is implicitly converted to the STRING type for calculation.
trimChars	No	STRING	Characters to be removed

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **str** is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of **str** or **trimChars** is **NULL**, **NULL** is returned.

## Example Code

- Removes spaces on both left and right of the string **yxabcxx**. An example command is as follows:

The value **yxabcxx** is returned.

```
select trim(' yxabcxx ');
```

It is equivalent to the following statement:

```
select trim(both from ' yxabcxx ');  
select trim(from ' yxabcxx ');
```

- Removes all substrings from both the left and right ends of the string **yxabcxx** that consist of characters in the set **xy**.

The function returns **abc**, as any substring starting with **x** or **y** from both the left and right ends is removed.

```
select trim('xy', 'yxabcxx');
```

It is equivalent to the following statement:

```
select trim(both 'xy' from 'yxabcxx');
```

- The value of the input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select trim(null);  
select trim(null, 'yxabcxx');
```

## 14.2.40 upper/ucase

This function is used to convert all characters of a string to the upper case.

### Syntax

```
upper(string A)
```

or

```
ucase(string A)
```

### Parameters

**Table 14-73** Parameter

Parameter	Mandatory	Type	Description
A	Yes	STRING	Text string to be converted

### Return Values

The return value is of the STRING type.

#### NOTE

- If the value of the input parameter is not of the STRING, BIGINT, DOUBLE, DECIMAL, or DATETIME type, an error is reported.
- If the value of the input parameter is **NULL**, **NULL** is returned.

### Example Code

Converts lowercase characters in a string to the upper case. An example command is as follows:

The value **ABC** is returned.

```
select upper('abc');
```

The value of the input parameter is **NULL**. An example command is as follows:

The value **NULL** is returned.

```
select upper(null);
```

## 14.3 Mathematical Functions

### 14.3.1 Overview

[Table 14-74](#) lists the mathematical functions supported by DLI.

**Table 14-74** Mathematical functions

Function	Syntax	Value Type	Description
<b>abs</b>	abs(DOUBLE a)	DOUBLE or INT	Returns the absolute value.
<b>acos</b>	acos(DOUBLE a)	DOUBLE	Returns the arc cosine value of <b>a</b> .
<b>asin</b>	asin(DOUBLE a)	DOUBLE	Returns the arc sine value of <b>a</b> .
<b>atan</b>	atan(DOUBLE a)	DOUBLE	Returns the arc tangent value of <b>a</b> .
<b>bin</b>	bin(BIGINT a)	STRING	Returns a number in binary format.
<b>bround</b>	bround(DOUBLE a)	DOUBLE	In HALF_EVEN rounding, the digit 5 is rounded up if the digit before 5 is an odd number and rounded down if the digit before 5 is an even number. For example, bround(7.5) = 8.0, bround(6.5) = 6.0.
<b>bround</b>	bround(DOUBLE a, INT d)	DOUBLE	The value is rounded off to d decimal places in HALF_EVEN mode. The digit 5 is rounded up if the digit before 5 is an odd number and rounded down if the digit before 5 is an even number. For example, bround(8.25, 1) = 8.2, bround(8.35, 1) = 8.4.
<b>cbrt</b>	cbrt(DOUBLE a)	DOUBLE	Returns the cube root of <b>a</b> .
<b>ceil</b>	ceil(DOUBLE a)	DECIMAL	Returns the smallest integer that is greater than or equal to <b>a</b> . For example, ceil(21.2) = 22.
<b>conv</b>	conv(BIGINT num, INT from_base, INT to_base), conv(STRING num, INT from_base, INT to_base)	STRING	Converts a number from <b>from_base</b> to <b>to_base</b> . For example, convert 5 from decimal to quaternary using conv(5,10,4) = 11.
<b>cos</b>	cos(DOUBLE a)	DOUBLE	Returns the cosine value of <b>a</b> .
<b>cot1</b>	cot1(DOUBLE a)	DOUBLE or DECIMAL	Returns the cotangent of a specified radian value.

Function	Syntax	Value Type	Description
<b>degrees</b>	degrees(DOUBLE a)	DOUBLE	Returns the angle corresponding to the radian.
<b>e</b>	e()	DOUBLE	Returns the value of <b>e</b> .
<b>exp</b>	exp(DOUBLE a)	DOUBLE	Returns the value of <b>e</b> raised to the power of <b>a</b> .
<b>factorial</b>	factorial(INT a)	BIGINT	Returns the factorial of <b>a</b> .
<b>floor</b>	floor(DOUBLE a)	BIGINT	Returns the largest integer that is less than or equal to A. For example, floor(21.2) = 21.
<b>greatest</b>	greatest(T v1, T v2, ...)	DOUBLE	Returns the greatest value of a list of values.
<b>hex</b>	hex(BIGINT a) hex(String a)	STRING	Converts an integer or character into its hexadecimal representation.
<b>least</b>	least(T v1, T v2, ...)	DOUBLE	Returns the least value of a list of values.
<b>ln</b>	ln(DOUBLE a)	DOUBLE	Returns the natural logarithm of a given value.
<b>log</b>	log(DOUBLE base, DOUBLE a)	DOUBLE	Returns the natural logarithm of a given base and exponent.
<b>log10</b>	log10(DOUBLE a)	DOUBLE	Returns the base-10 logarithm of a given value.
<b>log2</b>	log2(DOUBLE a)	DOUBLE	Returns the base-2 logarithm of a given value.
<b>median</b>	median(colname)	DOUBLE or DECIMAL	Returns the median.
<b>negative</b>	negative(INT a)	DECIMAL or INT	Returns the opposite number of <b>a</b> . For example, if negative(2) is given, -2 is returned.

Function	Syntax	Value Type	Description
percentile	percentile(colname,DOUBLE p)	DOUBLE or ARRAY	Returns the exact percentile, which is applicable to a small amount of data. Sorts a specified column in ascending order, and then obtains the exact pth percentage. The value of <b>p</b> must be between 0 and 1.
percentile_approx	percentile_approx (colname,DOUBLE p)	DOUBLE or ARRAY	Returns the approximate percentile, which is applicable to a large amount of data. Sorts a specified column in ascending order, and then obtains the value corresponding to the pth percentile.
pi	pi()	DOUBLE	Returns the value of <b>pi</b> .
pmod	pmod(INT a, INT b)	DECIMAL or INT	Returns the positive value of the remainder after division of <b>x</b> by <b>y</b> .
positive	positive(INT a)	DECIMAL, DOUBLE, or INT	Returns the value of <b>a</b> , for example, <b>positive(2) = 2</b> .
pow	pow(DOUBLE a, DOUBLE p), power(DOUBLE a, DOUBLE p)	DOUBLE	Returns the value of <b>a</b> raised to the power of <b>p</b> .
radians	radians(DOUBLE a)	DOUBLE	Returns the radian corresponding to the angle.
rand	rand(INT seed)	DOUBLE	Returns an evenly distributed random number that is greater than or equal to 0 and less than 1. If the seed is specified, a stable random number sequence is displayed.
round	round(DOUBLE a)	DOUBLE	Round off
round	round(DOUBLE a, INT d)	DOUBLE	Rounds <b>a</b> to <b>d</b> decimal places, for example, <b>round(21.263,2) = 21.26</b> .
shiftleft	shiftleft(BIGINT a, INT b)	INT	Bitwise signed left shift. Interprets <b>a</b> as a binary number and shifts the binary number <b>b</b> positions to the left.
shiftright	shiftright(BIGINT a, INT b)	INT	Bitwise signed right shift. Interprets <b>a</b> as a binary number and shifts the binary number <b>b</b> positions to the right.

Function	Syntax	Value Type	Description
<b>shiftrightunsigned</b>	shiftrightunsigned(BIGINT a, INT b)	INT	Bitwise unsigned right shift. Interprets <b>a</b> as a binary number and shifts the binary number <b>b</b> positions to the right.
<b>sign</b>	sign(DOUBLE a)	DOUBLE	Returns the sign of <b>a</b> . <b>1.0</b> is returned if <b>a</b> is positive. <b>-1.0</b> is returned if <b>a</b> is negative. Otherwise, <b>0.0</b> is returned.
<b>sin</b>	sin(DOUBLE a)	DOUBLE	Returns the sine value of the given angle <b>a</b> .
<b>sqrt</b>	sqrt(DOUBLE a)	DOUBLE	Returns the square root of <b>a</b> .
<b>tan</b>	tan(DOUBLE a)	DOUBLE	Returns the tangent value of the given angle <b>a</b> .

### 14.3.2 abs

This function is used to calculate the absolute value of an input parameter.

#### Syntax

```
abs(DOUBLE a)
```

#### Parameters

Table 14-75 Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

#### Return Values

The return value is of the DOUBLE or INT type.

 NOTE

If the value of **a** is **NULL**, **NULL** is returned.

### Example Code

The value **NULL** is returned.

```
select abs(null);
```

The value **1** is returned.

```
select abs(-1);
```

The value **3.1415926** is returned.

```
select abs(-3.1415926);
```

## 14.3.3 acos

This function is used to return the arc cosine value of a given angle **a**.

### Syntax

```
acos(DOUBLE a)
```

### Parameters

**Table 14-76** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value range is [-1, 1]. The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

### Return Values

The return value is of the DOUBLE type. The value ranges from 0 to  $\pi$ .

 NOTE

- If the value of **a** is not within the range [-1, 1], **NaN** is returned.
- If the value of **a** is **NULL**, **NULL** is returned.

### Example Code

The value **3.141592653589793** is returned.

```
select acos(-1);
```

The value **0** is returned.

```
select acos(1);
```

The value **NULL** is returned.

```
select acos(null);
```

The value **NAN** is returned.

```
select acos(10);
```

## 14.3.4 asin

This function is used to return the arc sine value of a given angle **a**.

### Syntax

```
asin(DOUBLE a)
```

### Parameters

**Table 14-77** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value range is [-1, 1]. The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

### Return Values

The return value is of the DOUBLE type. The value ranges from  $-\pi/2$  to  $\pi/2$ .

#### NOTE

- If the value of **a** is not within the range [-1, 1], **NaN** is returned.
- If the value of **a** is **NULL**, **NULL** is returned.

### Example Code

The value **1.5707963267948966** is returned.

```
select asin(1);
```

The value **0.6435011087932844** is returned.

```
select asin(0.6);
```

The value **NULL** is returned.

```
select asin(null);
```

The value **NAN** is returned.

```
select asin(10);
```

## 14.3.5 atan

This function is used to return the arc tangent value of a given angle **a**.

### Syntax

```
atan(DOUBLE a)
```

### Parameters

**Table 14-78** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

### Return Values

The return value is of the DOUBLE type. The value ranges from  $-\pi/2$  to  $\pi/2$ .

#### NOTE

- If the value of **a** is not within the range  $[-1, 1]$ , **NaN** is returned.
- If the value of **a** is **NULL**, **NULL** is returned.

### Example Code

The value **0.7853981633974483** is returned.

```
select atan(1);
```

The value **0.5404195002705842** is returned.

```
select atan(0.6);
```

The value **NULL** is returned.

```
select atan(null);
```

## 14.3.6 bin

This function is used to return the binary format of **a**.

### Syntax

```
bin(BIGINT a)
```

## Parameters

**Table 14-79** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value is an integer. If the value is not of the BIGINT type, the system will implicitly convert it to the BIGINT type for calculation.

## Return Values

The return value is of the STRING type.

 **NOTE**

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **1** is returned.

```
select bin(1);
```

The value **NULL** is returned.

```
select bin(null);
```

The value **1000** is returned.

```
select bin(8);
```

The value **1000** is returned.

```
select bin(8.123456);
```

## 14.3.7 bround

This function is used to return a value that is rounded off to **d** decimal places.

## Syntax

```
bround(DOUBLE a, INT d)
```

## Parameters

**Table 14-80** Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. It indicates the value that needs to be rounded. The digit 5 is rounded up if the digit before 5 is an odd number and rounded down if the digit before 5 is an even number. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.
d	No	DOUBLE, BIGINT, DECIMAL, or STRING	It indicates the number of decimal places to which the value needs to be rounded. If the value is not of the INT type, the system will implicitly convert it to the INT type for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **a** or **d** is **NULL**, **NULL** is returned.

## Example Code

The value **123.4** is returned.

```
select bround(123.45,1);
```

The value **123.6** is returned.

```
select bround(123.55,1);
```

The value **NULL** is returned.

```
select bround(null);
```

The value **123.457** is returned.

```
select bround(123.456789,3.123456);
```

## 14.3.8 cbrt

This function is used to return the cube root of **a**.

## Syntax

```
cbrt(DOUBLE a)
```

## Parameters

**Table 14-81** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **3** is returned.

```
select cbrt(27);
```

The value **3.3019272488946267** is returned.

```
select cbrt(36);
```

The value **NULL** is returned.

```
select cbrt(null);
```

## 14.3.9 ceil

This function is used to round up **a** to the nearest integer.

## Syntax

```
ceil(DOUBLE a)
```

## Parameters

Table 14-82 Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DECIMAL type.

 **NOTE**

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **2** is returned.

```
select ceil(1.3);
```

The value **-1** is returned.

```
select ceil(-1.3);
```

The value **NULL** is returned.

```
select ceil(null);
```

### 14.3.10 conv

This function is used to convert a number from **from\_base** to **to\_base**.

## Syntax

```
conv(BIGINT num, INT from_base, INT to_base)
```

## Parameters

**Table 14-83** Parameters

Parameter	Mandatory	Type	Description
num	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	Number whose base needs to be converted The value can be a float, integer, or string.
from_base	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	It represents the base from which the number is converted. The value can be a float, integer, or string.
to_base	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	It represents the base to which the number is converted. The value can be a float, integer, or string.

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **num**, **from\_base**, or **to\_base** is **NULL**, **NULL** is returned.
- The conversion process works with 64-bit precision and returns **NULL** when there is overflow.
- If the value of **num** is a decimal, it will be converted to an integer before the base conversion, and the decimal part will be discarded.

## Example Code

The value **8** is returned.

```
select conv('1000', 2, 10);
```

The value **B** is returned.

```
select conv('1011', 2, 16);
```

The value **703710** is returned.

```
select conv('ABCDE', 16, 10);
```

The value **27** is returned.

```
select conv(1000.123456, 3.123456, 10.123456);
```

The value **18446744073709551589** is returned.

```
select conv(-1000.123456, 3.123456, 10.123456);
```

The value **NULL** is returned.

```
select conv('1100', null, 10);
```

### 14.3.11 cos

This function is used to calculate the cosine value of **a**, with input in radians.

#### Syntax

```
cos(DOUBLE a)
```

#### Parameters

**Table 14-84** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

#### Return Values

The return value is of the DOUBLE type.

#### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

#### Example Code

The value **-0.9999999999999986** is returned.

```
select cos(3.1415926);
```

The value **NULL** is returned.

```
select cos(null);
```

### 14.3.12 cot1

This function is used to calculate the cotangent value of **a**, with input in radians.

#### Syntax

```
cot1(DOUBLE a)
```

## Parameters

**Table 14-85** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE or DECIMAL type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **1.0000000000000002** is returned.

```
select cot1(pi()/4);
```

The value **NULL** is returned.

```
select cot1(null);
```

## 14.3.13 degrees

This function is used to calculate the angle corresponding to the returned radian.

## Syntax

```
degrees(DOUBLE a)
```

## Parameters

**Table 14-86** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **90.0** is returned.

```
select degrees(1.5707963267948966);
```

The value **0** is returned.

```
select degrees(0);
```

The value **NULL** is returned.

```
select degrees(null);
```

## 14.3.14 e

This function is used to return the value of **e**.

## Syntax

```
e()
```

## Return Values

The return value is of the DOUBLE type.

## Example Code

The value **2.718281828459045** is returned.

```
select e();
```

## 14.3.15 exp

This function is used to return the value of **e** raised to the power of **a**.

## Syntax

```
exp(DOUBLE a)
```

## Parameters

**Table 14-87** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

 **NOTE**

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **7.38905609893065** is returned.

```
select exp(2);
```

The value **20.085536923187668** is returned.

```
select exp(3);
```

The value **NULL** is returned.

```
select exp(null);
```

## 14.3.16 factorial

This function is used to return the factorial of **a**.

## Syntax

```
factorial(INT a)
```

## Parameters

**Table 14-88** Parameter

Parameter	Mandatory	Type	Description
a	Yes	BIGINT, INT, SMALLINT, or TINYINT	The value is an integer. If the value is not of the INT type, the system will implicitly convert it to the INT type for calculation. The string is converted to its corresponding ASCII code.

## Return Values

The return value is of the BIGINT type.

### NOTE

- If the value of **a** is **0**, **1** is returned.
- If the value of **a** is **NULL** or outside the range of [0, 20], **NULL** is returned.

## Example Code

The value **720** is returned.

```
select factorial(6);
```

The value **1** is returned.

```
select factorial(1);
```

The value **120** is returned.

```
select factorial(5.123456);
```

The value **NULL** is returned.

```
select factorial(null);
```

The value **NULL** is returned.

```
select factorial(21);
```

## 14.3.17 floor

This function is used to round down **a** to the nearest integer.

## Syntax

```
floor(DOUBLE a)
```

## Parameters

**Table 14-89** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the BIGINT type.

 **NOTE**

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **1** is returned.

```
select floor(1.2);
```

The value **-2** is returned.

```
select floor(-1.2);
```

The value **NULL** is returned.

```
select floor(null);
```

## 14.3.18 greatest

This function is used to return the greatest value in a list of values.

## Syntax

```
greatest(T v1, T v2, ...)
```

## Parameters

**Table 14-90** Parameters

Parameter	Mandatory	Type	Description
v1	Yes	DOUBLE, BIGINT, or DECIMAL	The value can be a float or integer.

Parameter	Mandatory	Type	Description
v2	Yes	DOUBLE, BIGINT, or DECIMAL	The value can be a float or integer.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **4.0** is returned.

```
select greatest(1,2,0,3,4.0);
```

The value **NULL** is returned.

```
select greatest(null);
```

## 14.3.19 hex

This function is used to convert an integer or character into its hexadecimal representation.

## Syntax

```
hex(BIGINT a)
```

## Parameters

**Table 14-91** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the BIGINT type, the system will implicitly convert it to the BIGINT type for calculation. The string is converted to its corresponding ASCII code.

## Return Values

The return value is of the STRING type.

### NOTE

- If the value of **a** is **0**, **0** is returned.
- If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **0** is returned.

```
select hex(0);
```

The value **61** is returned.

```
select hex('a');
```

The value **10** is returned.

```
select hex(16);
```

The value **31** is returned.

```
select hex('1');
```

The value **3136** is returned.

```
select hex('16');
```

The value **NULL** is returned.

```
select hex(null);
```

## 14.3.20 least

This function is used to return the smallest value in a list of values.

### Syntax

```
least(T v1, T v2, ...)
```

### Parameters

**Table 14-92** Parameters

Parameter	Mandatory	Type	Description
v1	Yes	DOUBLE, BIGINT, or DECIMAL	The value can be a float or integer.
v2	Yes	DOUBLE, BIGINT, or DECIMAL	The value can be a float or integer.

## Return Values

The return value is of the DOUBLE type.

### NOTE

- If the value of **v1** or **v2** is of the STRING type, an error is reported.
- If the values of all parameters are **NULL**, **NULL** is returned.

## Example Code

The value **1.0** is returned.

```
select least(1,2.0,3,4.0);
```

The value **NULL** is returned.

```
select least(null);
```

## 14.3.21 ln

This function is used to return the natural logarithm of a given value.

## Syntax

```
ln(DOUBLE a)
```

## Parameters

**Table 14-93** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

- If the value of **a** is negative or **0**, **NULL** is returned.
- If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **1.144729868791239** is returned.

```
select ln(3.1415926);
```

The value **1** is returned.

```
select ln(2.718281828459045);
```

The value **NULL** is returned.

```
select ln(null);
```

## 14.3.22 log

This function is used to return the natural logarithm of a given base and exponent.

### Syntax

```
log(DOUBLE base, DOUBLE a)
```

### Parameters

**Table 14-94** Parameters

Parameter	Mandatory	Type	Description
base	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

### Return Values

The return value is of the DOUBLE type.

#### NOTE

- If the value of **base** or **a** is **NULL**, **NULL** is returned.
- If the value of **base** or **a** is negative or **0**, **NULL** is returned.
- If the value of **base** is **1** (would cause a division by zero), **NULL** is returned.

### Example Code

The value **2** is returned.

```
select log(2, 4);
```

The value **NULL** is returned.

```
select log(2, null);
```

The value **NULL** is returned.

```
select log(null, 4);
```

### 14.3.23 log10

This function is used to return the natural logarithm of a given value with a base of 10.

#### Syntax

```
log10(DOUBLE a)
```

#### Parameters

**Table 14-95** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

#### Return Values

The return value is of the DOUBLE type.

#### NOTE

If the value of **a** is negative, **0**, or **NULL**, **NULL** is returned.

#### Example Code

The value **NULL** is returned.

```
select log10(null);
```

The value **NULL** is returned.

```
select log10(0);
```

The value **0.9542425094393249** is returned.

```
select log10(9);
```

The value **1** is returned.

```
select log10(10);
```

### 14.3.24 log2

This function is used to return the natural logarithm of a given value with a base of 2.

## Syntax

```
log2(DOUBLE a)
```

## Parameters

**Table 14-96** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **a** is negative, **0**, or **NULL**, **NULL** is returned.

## Example Code

The value **NULL** is returned.

```
select log2(null);
```

The value **NULL** is returned.

```
select log2(0);
```

The value **3.1699250014423126** is returned.

```
select log2(9);
```

The value **4** is returned.

```
select log2(16);
```

## 14.3.25 median

This function is used to calculate the median of input parameters.

## Syntax

```
median(colname)
```

## Parameters

**Table 14-97** Parameter

Parameter	Mandatory	Type	Description
colname	Yes	DOUBLE, DECIMAL, STRING, or BIGINT	Name of the column to be sorted The elements in a column are of the DOUBLE type. If an element in a column is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE or DECIMAL type.

 **NOTE**

If the column name does not exist, an error is reported.

## Example Code

Assume that the elements in the **int\_test** column are 1, 2, 3, and 4 and they are of the INT type.

The value **2.5** is returned.

```
select median(int_test) FROM int_test;
```

### 14.3.26 negative

This function is used to return the additive inverse of **a**.

## Syntax

```
negative(INT a)
```

## Parameters

**Table 14-98** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string.

## Return Values

The return value is of the DECIMAL or INT type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **-1** is returned.

```
SELECT negative(1);
```

The value **3** is returned.

```
SELECT negative(-3);
```

## 14.3.27 percentile

This function is used to return the exact percentile, which is applicable to a small amount of data. It sorts a specified column in ascending order, and then obtains the exact value of the pth percentile.

## Syntax

```
percentile(colname,DOUBLE p)
```

## Parameters

**Table 14-99** Parameters

Parameter	Mandatory	Type	Description
colname	Yes	STRING	Name of the column to be sorted The elements in the column must be integers.
p	Yes	DOUBLE	The value ranges from 0 to 1. The value is a float.

## Return Values

The return value is of the DOUBLE or ARRAY type.

### NOTE

- If the column name does not exist, an error is reported.
- If the value of **p** is **NULL** or outside the range of [0, 1], an error is reported.

## Example Code

Assume that the elements in the `int_test` column are 1, 2, 3, and 4 and they are of the INT type.

The value **3.0999999999999996** is returned.

```
select percentile(int_test,0.7) FROM int_test;
```

The value **3.997** is returned.

```
select percentile(int_test,0.999) FROM int_test;
```

The value **2.5** is returned.

```
select percentile(int_test,0.5) FROM int_test;
```

The value **[1.3, 1.9, 2.5, 2.8, 3.7]** is returned.

```
select percentile (int_test,ARRAY(0.1,0.3,0.5,0.6,0.9)) FROM int_test;
```

## 14.3.28 percentile\_approx

This function is used to return the approximate percentile, which is applicable to a large amount of data. It sorts a specified column in ascending order, and then obtains the value closest to the pth percentile.

### Syntax

```
percentile_approx (colname,DOUBLE p)
```

### Parameters

**Table 14-100** Parameters

Parameter	Mandatory	Type	Description
colname	Yes	STRING	Name of the column to be sorted The elements in a column are of the DOUBLE type. If an element in a column is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.
p	Yes	DOUBLE	The value can be a float, integer, or string. The value ranges from 0 to 1. The value is a float.

### Return Values

The return value is of the DOUBLE or ARRAY type.

 NOTE

- If the column name does not exist, an error is reported.
- If the value of **p** is **NULL** or outside the range of [0, 1], an error is reported.

## Example Code

Assume that the elements in the **int\_test** column are 1, 2, 3, and 4 and they are of the INT type.

The value **3** is returned.

```
select percentile_approx(int_test,0.7) FROM int_test;
```

The value **3** is returned.

```
select percentile_approx(int_test,0.75) FROM int_test;
```

The value **2** is returned.

```
select percentile_approx(int_test,0.5) FROM int_test;
```

The value **[1,2,2,3,4]** is returned.

```
select percentile_approx (int_test,ARRAY(0.1,0.3,0.5,0.6,0.9)) FROM int_test;
```

## 14.3.29 pi

This function is used to return the value of  $\pi$ .

### Syntax

```
pi()
```

### Return Values

The return value is of the DOUBLE type.

## Example Code

The value **3.141592653589793** is returned.

```
select pi();
```

## 14.3.30 pmod

This function is used to return the positive value of the remainder after division of **x** by **y**.

### Syntax

```
pmod(INT a, INT b)
```

## Parameters

**Table 14-101** Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string.
b	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string.

## Return Values

The return value is of the DECIMAL or INT type.

### NOTE

- If the value of **a** or **b** is **NULL**, **NULL** is returned.
- If the value of **b** is **0**, **NULL** is returned.

## Example Code

The value **2** is returned.

```
select pmod(2,5);
```

The value **3** is returned.

```
select pmod (-2,5) (parse: -2=5* (-1)...3);
```

The value **NULL** is returned.

```
select pmod(5,0);
```

The value **1** is returned.

```
select pmod(5,2);
```

The value **0.877** is returned.

```
select pmod(5.123,2.123);
```

### 14.3.31 positive

This function is used to return the value of **a**.

## Syntax

```
positive(INT a)
```

## Parameters

**Table 14-102** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string.

## Return Values

The return value is of the DECIMAL, DOUBLE, or INT type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **3** is returned.

```
SELECT positive(3);
```

The value **-3** is returned.

```
SELECT positive(-3);
```

The value **123** is returned.

```
SELECT positive('123');
```

## 14.3.32 pow

This function is used to calculate and return the pth power of **a**.

## Syntax

```
pow(DOUBLE a, DOUBLE p),  
power(DOUBLE a, DOUBLE p)
```

## Parameters

**Table 14-103** Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.
p	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

 **NOTE**

If the value of **a** or **p** is **NULL**, **NULL** is returned.

## Example Code

The value **16** returned.

```
select pow(2, 4);
```

The value **NULL** is returned.

```
select pow(2, null);
```

The value **17.429460393524256** is returned.

```
select pow(2, 4.123456);
```

## 14.3.33 radians

This function is used to return the radian corresponding to an angle.

## Syntax

```
radians(DOUBLE a)
```

## Parameters

**Table 14-104** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

 **NOTE**

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **1.0471975511965976** is returned.

```
select radians(60);
```

The value **0** is returned.

```
select radians(0);
```

The value **NULL** is returned.

```
select radians(null);
```

## 14.3.34 rand

This function is used to return an evenly distributed random number that is greater than or equal to 0 and less than 1.

## Syntax

```
rand(INT seed)
```

## Parameters

**Table 14-105** Parameter

Parameter	Mandatory	Type	Description
seed	No	INT	The value can be a float, integer, or string. If this parameter is specified, a stable random number sequence is obtained within the same running environment.

## Return Values

The return value is of the DOUBLE type.

## Example Code

The value **0.3668915240363728** is returned.

```
select rand();
```

The value **0.25738143505962285** is returned.

```
select rand(3);
```

## 14.3.35 round

This function is used to calculate the rounded value of **a** up to **d** decimal places.

## Syntax

```
round(DOUBLE a, INT d)
```

## Parameters

**Table 14-106** Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	It indicates the value to be rounded off. The value can be a float, integer, or string.

Parameter	Mandatory	Type	Description
d	No	INT	The default value is <b>0</b> . It indicates the number of decimal places to which the value needs to be rounded. If the value is not of the INT type, the system will implicitly convert it to the INT type for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

- If the value of **d** is negative, an error is reported.
- If the value of **a** or **d** is **NULL**, **NULL** is returned.

## Example Code

The value **123.0** is returned.

```
select round(123.321);
```

The value **123.4** is returned.

```
select round(123.396, 1);
```

The value **NULL** is returned.

```
select round(null);
```

The value **123.321** is returned.

```
select round(123.321, 4);
```

The value **123.3** is returned.

```
select round(123.321,1.33333);
```

The value **123.3** is returned.

```
select round(123.321,1.33333);
```

## 14.3.36 shiftleft

This function is used to perform a signed bitwise left shift. It takes the binary number **a** and shifts it **b** positions to the left.

## Syntax

```
shiftleft(BIGINT a, BIGINT b)
```

## Parameters

**Table 14-107** Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the BIGINT type, the system will implicitly convert it to the BIGINT type for calculation.
b	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the BIGINT type, the system will implicitly convert it to the BIGINT type for calculation.

## Return Values

The return value is of the INT type.

### NOTE

If the value of **a** or **b** is **NULL**, **NULL** is returned.

## Example Code

The value **8** is returned.

```
select shiftright(1,3);
```

The value **48** is returned.

```
select shiftright(6,3);
```

The value **48** is returned.

```
select shiftright(6.123456,3.123456);
```

The value **NULL** is returned.

```
select shiftright(null,3);
```

### 14.3.37 shiftright

This function is used to perform a signed bitwise right shift. It takes the binary number **a** and shifts it **b** positions to the right.

## Syntax

```
shiftright(BIGINT a, BIGINT b)
```

## Parameters

**Table 14-108** Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the BIGINT type, the system will implicitly convert it to the BIGINT type for calculation.
b	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the BIGINT type, the system will implicitly convert it to the BIGINT type for calculation.

## Return Values

The return value is of the INT type.

 **NOTE**

If the value of **a** or **b** is **NULL**, **NULL** is returned.

## Example Code

The value **2** is returned.

```
select shiftright(16,3);
```

The value **4** is returned.

```
select shiftright(36,3);
```

The value **4** is returned.

```
select shiftright(36.123456,3.123456);
```

The value **NULL** is returned.

```
select shiftright(null,3);
```

### 14.3.38 shiftrightunsigned

This function is used to perform an unsigned bitwise right shift. It takes the binary number **a** and shifts it **b** positions to the right.

## Syntax

```
shiftrightunsigned(BIGINT a, BIGINT b)
```

## Parameters

**Table 14-109** Parameters

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the BIGINT type, the system will implicitly convert it to the BIGINT type for calculation.
b	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the BIGINT type, the system will implicitly convert it to the BIGINT type for calculation.

## Return Values

The return value is of the INT type.

 **NOTE**

If the value of **a** or **b** is **NULL**, **NULL** is returned.

## Example Code

The value **2** is returned.

```
select shiftrightunsigned(16,3);
```

The value **536870910** is returned.

```
select shiftrightunsigned(-16,3);
```

The value **2** is returned.

```
select shiftrightunsigned(16.123456,3.123456);
```

The value **NULL** is returned.

```
select shiftrightunsigned(null,3);
```

### 14.3.39 sign

This function is used to return the positive and negative signs corresponding to **a**.

## Syntax

```
sign(DOUBLE a)
```

## Parameters

**Table 14-110** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string.

## Return Values

The return value is of the DOUBLE type.

### NOTE

- If the value of **a** is a positive number, **1** is returned.
- If the value of **a** is a negative number, **-1** is returned.
- If the value of **a** is **0**, **0** is returned.
- If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **-1** is returned.

```
select sign(-3);
```

The value **1** is returned.

```
select sign(3);
```

The value **0** is returned.

```
select sign(0);
```

The value **1** is returned.

```
select sign(3.1415926);
```

The value **NULL** is returned.

```
select sign(null);
```

## 14.3.40 sin

This function is used to return the sine value of **a**, with input in radians.

## Syntax

```
sin(DOUBLE a)
```

## Parameters

**Table 14-111** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

 **NOTE**

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **1** is returned.

```
select sin(pi()/2);
```

The value **NULL** is returned.

```
select sin(null);
```

## 14.3.41 sqrt

This function is used to return the square root of a value.

## Syntax

```
sqrt(DOUBLE a)
```

## Parameters

**Table 14-112** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **2.8284271247461903** is returned.

```
select sqrt(8);
```

The value **4** is returned.

```
select sqrt(16);
```

The value **NULL** is returned.

```
select sqrt(null);
```

## 14.3.42 tan

This function is used to return the tangent value of **a**, with input in radians.

## Syntax

```
tan(DOUBLE a)
```

## Parameters

**Table 14-113** Parameter

Parameter	Mandatory	Type	Description
a	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	The value can be a float, integer, or string. If the value is not of the DOUBLE type, the system will implicitly convert it to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

The value **0.9999999999999999** is returned.

```
select tan(pi()/4);
```

The value **NULL** is returned.

```
select tan(null);
```

## 14.4 Aggregate Functions

### 14.4.1 Overview

**Table 14-114** lists the aggregate functions supported by DLI.

**Table 14-114** Aggregate functions

Function	Syntax	Value Type	Description
<b>avg</b>	avg(col), avg(DISTINCT col)	DOUBLE	Returns the average value.
<b>corr</b>	corr(col1, col2)	DOUBLE	Returns the coefficient of correlation of a pair of numeric columns.
<b>count</b>	count([distinct all] <colname>)	BIGINT	Returns the number of records.
<b>covar_pop</b>	covar_pop(col1, col2)	DOUBLE	Returns the covariance of a pair of numeric columns.
<b>covar_samp</b>	covar_samp(col1, col2)	DOUBLE	Returns the sample covariance of a pair of numeric columns.
<b>max</b>	max(col)	DOUBLE	Returns the maximum value.
<b>min</b>	min(col)	DOUBLE	Returns the minimum value.
<b>percentile</b>	percentile(BIGINT col, p)	DOUBLE	Returns the percentage value point of the value area. The value of <b>p</b> must be between 0 and 1. Otherwise, <b>NULL</b> is returned. The value cannot be a float.

Function	Syntax	Value Type	Description
<a href="#">percentile_approx</a>	percentile_approx(DOUBLE col, p [, B])	DOUBLE	Returns the approximate pth percentile of a numerical column within the group, including floating-point numbers. The value of p should be between 0 and 1. The parameter B controls the accuracy of the approximation, with a higher value of B resulting in a higher level of approximation. The default value is <b>10000</b> . If the number of non-repeating values in the column is less than B, an exact percentile is returned.
<a href="#">stddev_pop</a>	stddev_pop(col)	DOUBLE	Returns the deviation of a specified column.
<a href="#">stddev_samp</a>	stddev_samp(col)	DOUBLE	Returns the sample deviation of a specified column.
<a href="#">sum</a>	sum(col), sum(DISTINCT col)	DOUBLE	Returns the sum of the values in a column.
<a href="#">variance/var_pop</a>	variance(col), var_pop(col)	DOUBLE	Returns the variance of a column.
<a href="#">var_samp</a>	var_samp(col)	DOUBLE	Returns the sample variance of a specified column.

## 14.4.2 avg

This function is used to return the average value.

### Syntax

```
avg(col), avg(DISTINCT col)
```

### Parameters

**Table 14-115** Parameter

Parameter	Mandatory	Type	Description
col	Yes	All data types	The value can be of any data type and can be converted to the DOUBLE type for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of `col` is **NULL**, the column is not involved in calculation.

## Example Code

- Calculates the average number of items across all warehouses. An example command is as follows:

```
select avg(items) from warehouse;
```

The command output is as follows:

```
_c0  
100.0
```

- Calculates the average inventory of all items in each warehouse when used with **group by**. An example command is as follows:

```
select warehouseid, avg(items) from warehouse group by warehouseid;
```

The command output is as follows:

```
warehouseid _c1  
city1 155  
city2 101  
city3 194
```

## 14.4.3 corr

This function is used to return the correlation coefficient between two columns of numerical values.

## Syntax

```
corr(col1, col2)
```

## Parameters

**Table 14-116** Parameters

Parameter	Mandatory	Type	Description
col1	Yes	DOUBLE, BIGINT, INT, SMALLINT, TINYINT, FLOAT, or DECIMAL	Columns with a data type of numeric. If the value is of any other type, <b>NULL</b> is returned.
col2	Yes	DOUBLE, BIGINT, INT, SMALLINT, TINYINT, FLOAT, or DECIMAL	Columns with a data type of numeric. If the values are of any other type, <b>NULL</b> is returned.

## Return Values

The return value is of the DOUBLE type.

## Example Code

- Calculates the correlation coefficient between the inventory (items) and the price of all products. An example command is as follows:

```
select corr(items,price) from warehouse;
```

The command output is as follows:

```
_c0  
1.242355
```

- When used with **group by**, it groups all offerings by warehouse (warehouseid) and returns the correlation coefficient between the inventory (items) and the price of offerings within each group. An example command is as follows:

```
select warehouseid, corr(items,price) from warehouse group by warehouseid;
```

The command output is as follows:

```
warehouseid _c1  
city1 0.43124  
city2 0.53344  
city3 0.73425
```

## 14.4.4 count

This function is used to return the number of records.

## Syntax

```
count([distinct|all] <colname>)
```

## Parameters

Table 14-117 Parameters

Parameter	Mandator y	Description
distinct or all	No	Determines whether duplicate records should be excluded during counting. <b>all</b> is used by default, indicating that all records will be included in the count. If <b>distinct</b> is specified, only the number of unique values is counted.
colname	Yes	The column value can be of any type. The value can be *, that is, <b>count(*)</b> , indicating that the number of all rows is returned.

## Return Values

The return value is of the BIGINT type.

 NOTE

If the value of **colname** is **NULL**, the row is not involved in calculation.

## Example Code

- Calculates the total number of records in the warehouse table. An example command is as follows:

```
select count(*) from warehouse;
```

The command output is as follows:

```
_c0  
10
```

- When used with **group by**, it groups all offerings by warehouse (warehouseid) and calculates the number of offerings in each warehouse (warehouseid). An example command is as follows:

```
select warehouseid, count(*) from warehouse group by warehouseid;
```

The command output is as follows:

```
warehouseid _c1  
city1 6  
city2 5  
city3 6
```

Example 3: Calculates the number of warehouses through distinct deduplication. An example command is as follows:

```
select count(distinct warehouseid) from warehouse;
```

The command output is as follows:

```
_c0  
3
```

## 14.4.5 covar\_pop

This function is used to return the covariance between two columns of numerical values.

### Syntax

```
covar_pop(col1, col2)
```

### Parameters

**Table 14-118** Parameters

Parameter	Mandatory	Description
col1	Yes	Columns with a data type of numeric. If the values are of any other type, <b>NULL</b> is returned.
col2	Yes	Columns with a data type of numeric. If the values are of any other type, <b>NULL</b> is returned.

### Return Values

The return value is of the DOUBLE type.

## Example Code

- Calculates the covariance between the inventory (items) and the price of all offerings. An example command is as follows:

```
select covar_pop(items, price) from warehouse;
```

The command output is as follows:

```
_c0  
1.242355
```

- When used with **group by**, it groups all offerings by warehouse (warehouseid) and returns the covariance between the inventory (items) and the price of offerings within each group. An example command is as follows:

```
select warehouseid, covar_pop(items, price) from warehouse group by warehouseid;
```

The command output is as follows:

```
warehouseid _c1  
city1 1.13124  
city2 1.13344  
city3 1.53425
```

## 14.4.6 covar\_samp

This function is used to return the sample covariance between two columns of numerical values.

### Syntax

```
covar_samp(col1, col2)
```

### Parameters

Table 14-119 Parameters

Parameter	Mandatory	Description
col1	Yes	Columns with a data type of numeric. If the values are of any other type, <b>NULL</b> is returned.
col2	Yes	Columns with a data type of numeric. If the values are of any other type, <b>NULL</b> is returned.

### Return Values

The return value is of the DOUBLE type.

## Example Code

- Calculates the sample covariance between the inventory (items) and the price of all offerings. An example command is as follows:

```
select covar_samp(items, price) from warehouse;
```

The command output is as follows:

```
_c0  
1.242355
```

- When used with **group by**, it groups all offerings by warehouse (warehouseId) and returns the sample covariance between the inventory (items) and the price of offerings within each group. An example command is as follows:  

```
select warehouseId, covar_samp(items,price) from warehouse group by warehouseId;
```

The command output is as follows:

```
warehouseId_c1  
city1 1.03124  
city2 1.03344  
city3 1.33425
```

## 14.4.7 max

This function is used to return the maximum value.

### Syntax

```
max(col)
```

### Parameters

**Table 14-120** Parameter

Parameter	Mandatory	Type	Description
col	Yes	Any type except BOOLEAN	The value can be of any type except BOOLEAN.

### Return Values

The return value is of the DOUBLE type.

#### NOTE

The return type is the same as the type of **col**. The return rules are as follows:

- If the value of **col** is **NULL**, the row is not involved in calculation.
- If the value of **col** is of the BOOLEAN type, it cannot be used for calculation.

### Example Code

- Calculates the maximum inventory (items) of all offerings. An example command is as follows:

```
select max(items) from warehouse;
```

The command output is as follows:

```
_c0  
900
```

- When used with **group by**, it returns the maximum inventory of each warehouse. An example command is as follows:

```
select warehouseId, max(items) from warehouse group by warehouseId;
```

The command output is as follows:

```
warehouseId_c1  
city1 200
```

```
city2 300  
city3 400
```

## 14.4.8 min

This function is used to return the minimum value.

### Syntax

```
min(col)
```

### Parameters

Table 14-121 Parameter

Parameter	Mandatory	Type	Description
col	Yes	Any type except BOOLEAN	The value can be of any type except BOOLEAN.

### Return Values

The return value is of the DOUBLE type.

#### NOTE

The return type is the same as the type of **col**. The return rules are as follows:

- If the value of **col** is **NULL**, the row is not involved in calculation.
- If the value of **col** is of the BOOLEAN type, it cannot be used for calculation.

### Example Code

- Calculates the minimum inventory (items) of all offerings. An example command is as follows:

```
select min(items) from warehouse;
```

The command output is as follows:

```
_c0  
600
```

- When used with **group by**, it returns the minimum inventory of each warehouse. An example command is as follows:

```
select warehouseid, min(items) from warehouse group by warehouseid;
```

The command output is as follows:

```
warehouseid _c1  
city1 15  
city2 10  
city3 19
```

## 14.4.9 percentile

This function is used to return the numerical value at a certain percentage point within a range of values.

## Syntax

```
percentile(BIGINT col, p)
```

## Parameters

**Table 14-122** Parameters

Parameter	Mandatory	Description
col	Yes	Columns with a data type of numeric. If the values are of any other type, <b>NULL</b> is returned.
p	Yes	The value should be between 0 and 1. Otherwise, <b>NULL</b> is returned.

## Return Values

The return value is of the DOUBLE type.

### NOTE

The value should be between 0 and 1. Otherwise, **NULL** is returned.

## Example Code

- Calculates the 0.5 percentile of all offering inventories (items). An example command is as follows:

```
select percentile(items,0.5) from warehouse;
```

The command output is as follows:

```
+-----+
|_c0   |
+-----+
| 500.6 |
+-----+
```

- When used with **group by**, it groups all offerings by warehouse (warehouseId) and returns the 0.5 percentile of the offering inventory (items) in the same group. An example command is as follows:

```
select warehouseId, percentile(items, 0.5) from warehouse group by warehouseId;
```

The command output is as follows:

```
+-----+-----+
|warehouseId|_c1   |
+-----+-----+
| city1     | 499.6 |
| city2     | 354.8 |
| city3     | 565.7 |
+-----+-----+
```

### 14.4.10 percentile\_approx

This function is used to approximate the pth percentile (including floating-point numbers) of a numeric column within a group.

## Syntax

```
percentile_approx(DOUBLE col, p [, B])
```

## Parameters

**Table 14-123** Parameters

Parameter	Mandatory	Description
col	Yes	Columns with a data type of numeric. If the values are of any other type, <b>NULL</b> is returned.
p	Yes	The value should be between 0 and 1. Otherwise, <b>NULL</b> is returned.
B	Yes	The parameter B controls the accuracy of the approximation, with a higher value of B resulting in a higher level of approximation. The default value is <b>10000</b> . If the number of non-repeating values in the column is less than B, an exact percentile is returned.

## Return Values

The return value is of the DOUBLE type.

## Example Code

- Calculates the 0.5 percentile of all offering inventories (items), with an accuracy of 100. An example command is as follows:

```
select PERCENTILE_APPROX(items,0.5,100) from warehouse;
```

The command output is as follows:

```
+-----+
|_c0   |
+-----+
| 521  |
+-----+
```

- When used with **group by**, it groups all offerings by warehouse (warehouseId) and returns the 0.5 percentile of the offering inventory (items) in the same group, with an accuracy of 100. An example command is as follows:

```
select warehouseId, PERCENTILE_APPROX(items, 0.5, 100) from warehouse group by warehouseId;
```

The command output is as follows:

```
+-----+-----+
|warehouseId|_c1   |
+-----+-----+
| city1    | 499  |
| city2    | 354  |
| city3    | 565  |
+-----+-----+
```

## 14.4.11 stddev\_pop

This function is used to return the deviation of a specified column.

### Syntax

```
stddev_pop(col)
```

### Parameters

**Table 14-124** Parameter

Parameter	Mandatory	Description
col	Yes	Columns with a data type of numeric. If the value is of any other type, <b>NULL</b> is returned.

### Return Values

The return value is of the DOUBLE type.

### Example Code

- Calculates the deviation of all offering inventories (items). An example command is as follows:

```
select stddev_pop(items) from warehouse;
```

The command output is as follows:

```
_c0  
1.342355
```

- When used with **group by**, it groups all offerings by warehouse (warehouseId) and returns the deviation of the offering inventory (items) in the same group. An example command is as follows:

```
select warehouseId, stddev_pop(items) from warehouse group by warehouseId;
```

The command output is as follows:

```
warehouseId_c1  
city1 1.23124  
city2 1.23344  
city3 1.43425
```

## 14.4.12 stddev\_samp

This function is used to return the sample deviation of a specified column.

### Syntax

```
stddev_samp(col)
```

## Parameters

**Table 14-125** Parameter

Parameter	Mandatory	Description
col	Yes	Columns with a data type of numeric. If the values are of any other type, <b>NULL</b> is returned.

## Return Values

The return value is of the DOUBLE type.

## Example Code

- Calculates the sample deviation of all offering inventories (items). An example command is as follows:

```
select stddev_samp(items) from warehouse;
```

The command output is as follows:

```
+-----+
|_c0    |
+-----+
| 1.342355 |
+-----+
```

- When used with **group by**, it groups all offerings by warehouse (warehouseid) and returns the sample deviation of the offering inventory (items) in the same group. An example command is as follows:

```
select warehouseid, stddev_samp(items) from warehouse group by warehouseid;
```

The command output is as follows:

```
+-----+-----+
|warehouseid|_c1    |
+-----+-----+
|city1     | 1.23124 |
|city2     | 1.23344 |
|city3     | 1.43425 |
+-----+-----+
```

## 14.4.13 sum

This function is used to calculate the total sum.

## Syntax

```
sum(col),
sum(DISTINCT col)
```

## Parameters

**Table 14-126** Parameter

Parameter	Mandatory	Description
col	Yes	The value can be of any data type and can be converted to the DOUBLE type for calculation. The value can be of the DOUBLE, DECIMAL, or BIGINT type. If the value is of the STRING type, the system implicitly converts it to DOUBLE for calculation.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **col** is **NULL**, the row is not involved in calculation.

## Example Code

- Calculates the total number of offerings (items) in all warehouses. An example command is as follows:

```
select sum(items) from warehouse;
```

The command output is as follows:

```
_c0  
55357
```
- When used with **group by**, it groups all offerings by warehouse (warehouseId) and calculates the total number of offerings in all warehouses. An example command is as follows:

```
select warehouseId, sum(items) from warehouse group by warehouseId;
```

The command output is as follows:

```
warehouseId|_c1  
city1      15500  
city2      10175  
city3      19400
```

### 14.4.14 variance/var\_pop

This function is used to return the variance of a column.

## Syntax

```
variance(col),  
var_pop(col)
```

## Parameters

Table 14-127 Parameter

Parameter	Mandatory	Description
col	Yes	Columns with a data type of numeric. If the value is of any other type, <b>NULL</b> is returned.

## Return Values

The return value is of the DOUBLE type.

## Example Code

- Calculates the variance of all offering inventories (items). An example command is as follows:

```
select variance(items) from warehouse;  
-- It is equivalent to the following statement:  
select var_pop(items) from warehouse;
```

The command output is as follows:

```
_c0  
203.42352
```

- When used with **group by**, it groups all offerings by warehouse (warehouseId) and returns the variance of the offering inventory (items) in the same group. An example command is as follows:

```
select warehouseId, variance(items) from warehouse group by warehouseId;  
-- It is equivalent to the following statement:  
select warehouseId, var_pop(items) from warehouse group by warehouseId;
```

The command output is as follows:

```
warehouseId_c1  
city1 19.23124  
city2 17.23344  
city3 12.43425
```

### 14.4.15 var\_samp

This function is used to return the sample variance of a specified column.

## Syntax

```
var_samp(col)
```

## Parameters

**Table 14-128** Parameter

Parameter	Mandatory	Description
col	Yes	Columns with a data type of numeric. If the values are of any other type, <b>NULL</b> is returned.

## Return Values

The return value is of the DOUBLE type.

## Example Code

- Calculates the sample variance of all offering inventories (items). An example command is as follows:

```
select var_samp(items) from warehouse;
```

The command output is as follows:

```
_c0  
294.342355
```

- When used with **group by**, it groups all offerings by warehouse (warehouseId) and returns the sample variance of the offering inventory (items) in the same group. An example command is as follows:

```
select warehouseId, var_samp(items) from warehouse group by warehouseId;
```

The command output is as follows:

```
warehouseId _c1  
city1 18.23124  
city2 16.23344  
city3 11.43425
```

# 14.5 Window Functions

## 14.5.1 Overview

[Table 14-129](#) lists the window functions supported by DLI.

**Table 14-129** Window functions

Function	Syntax	Value Type	Description
<a href="#">cume_dist</a>	cume_dist()	DOUBLE	Returns the cumulative distribution, which is equivalent to calculating the proportion of data in the partition that is greater than or equal to, or less than or equal to, the current row.
<a href="#">first_value</a>	first_value(col)	Data type of the argument	Returns the value of the first data record in a column in a result set.
<a href="#">last_value</a>	last_value(col)	Data type of the argument	Returns the value of the last data record from a column.
<a href="#">lag</a>	lag (col,n,DEFAULT)	Data type of the argument	Returns the value from the <i>n</i> th row preceding the current row. The first argument specifies the column name. The second argument specifies the <i>n</i> th row preceding the current row. The configuration of the second argument is optional, and the default argument value is <b>1</b> if the argument is not specified. The third argument is set to a default value. If the <i>n</i> th row preceding the current row is <b>null</b> , the default value is used. The default value of the third argument is <b>NULL</b> if the argument is not specified.

Function	Syntax	Value Type	Description
<a href="#">lead</a>	lead (col,n,DEFAULT)	Data type of the argument	Returns the value from the <i>n</i> th row following the current row. The first argument specifies the column name. The second argument specifies the <i>n</i> th row following the current row. The configuration of the second argument is optional, and the default argument value is <b>1</b> if the argument is not specified. The third argument is set to a default value. If the <i>n</i> th row following the current row is <b>null</b> , the default value is used. The default value of the third argument is <b>NULL</b> if the argument is not specified.
<a href="#">percent_rank</a>	percent_rank()	DOUBLE	Returns the rank of a value from the column specified by the ORDER BY clause of the window. The return value is a decimal between 0 and 1, which is calculated using $(RANK - 1)/(- 1)$ .
<a href="#">rank</a>	rank()	INT	Returns the rank of a value in a set of values. When multiple values share the same rank, the next rank in the sequence is not consecutive.
<a href="#">row_number</a>	row_number() over (order by col_1[,col_2 ...])	INT	Assigns a unique number to each row.

## 14.5.2 cume\_dist

This function is used to return the cumulative distribution, which is equivalent to calculating the proportion of data in the partition that is greater than or equal to, or less than or equal to, the current row.

### Restrictions

The restrictions on using window functions are as follows:

- Window functions can be used only in select statements.
- Window functions and aggregate functions cannot be nested in window functions.

- Window functions cannot be used together with aggregate functions of the same level.

## Syntax

```
cume_dist() over([partition_clause] [orderby_clause])
```

## Parameters

**Table 14-130** Parameters

Parameter	Mandatory	Description
partition_clause	No	Partition. Rows with the same value in partition columns are considered to be in the same window.
orderby_clause	No	How data is sorted in a window

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the salary table and add data:

```
CREATE EXTERNAL TABLE salary (  
  dept STRING, -- Department name  
  userid string, -- Employee ID  
  sal INT -- Salary  
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
stored as textfile;
```

Adds the following data:

```
d1,user1,1000  
d1,user2,2000  
d1,user3,3000  
d2,user4,4000  
d2,user5,5000
```

- Calculates the proportion of employees whose salary is 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
```

- Calculates the proportion of employees whose salary is less than or equal to the current salary by 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
d2 user4 4000 0.5
d2 user5 5000 1.0
```

- After sorting by **sal** in descending order, the result is the ratio of employees whose salary is 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
```

### 14.5.3 first\_value

This function is used to obtain the value of the first data record in the window corresponding to the current row.

#### Restrictions

The restrictions on using window functions are as follows:

- Window functions can be used only in select statements.
- Window functions and aggregate functions cannot be nested in window functions.
- Window functions cannot be used together with aggregate functions of the same level.

#### Syntax

```
first_value(<expr>[, <ignore_nulls>]) over ([partition_clause] [orderby_clause] [frame_clause])
```

## Parameters

**Table 14-131** Parameters

Parameter	Mandatory	Description
expr	Yes	Expression whose return result is to be calculated
ignore_nulls	No	The value is of the BOOLEAN type, indicating whether to ignore NULL values. The default value is <b>False</b> . If the value is <b>True</b> , the first non-null value in the window is returned.
partition_clause	No	Partition. Rows with the same value in partition columns are considered to be in the same window.
orderby_clause	No	It is used to specify how data is sorted in a window.
frame_clause	No	It is used to determine the data boundary.

## Return Values

The return value is of the data type of the parameter.

## Example Code

### Example data

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the **logs** table and add data:

```
create table logs(  
  cookieid string,  
  createtime string,  
  url string  
)  
STORED AS parquet;
```

Adds the following data:

```
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: Groups all records by **cookieid**, sorts the records by **createtime** in ascending order, and returns the first row of data in each group. An example command is as follows:

```
SELECT cookieid, createtime, url,  
       FIRST_VALUE(url) OVER (PARTITION BY cookieid ORDER BY createtime) AS first  
FROM logs;
```

The command output is as follows:

```
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  
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
```

## 14.5.4 last\_value

This function is used to obtain the value of the last data record in the window corresponding to the current row.

### Restrictions

The restrictions on using window functions are as follows:

- Window functions can be used only in select statements.
- Window functions and aggregate functions cannot be nested in window functions.
- Window functions cannot be used together with aggregate functions of the same level.

### Syntax

```
last_value(<expr>[, <ignore_nulls>]) over ([partition_clause] [orderby_clause] [frame_clause])
```

### Parameters

**Table 14-132** Parameters

Parameter	Mandatory	Description
expr	Yes	Expression whose return result is to be calculated

Parameter	Mandator y	Description
ignore_nulls	No	The value is of the BOOLEAN type, indicating whether to ignore NULL values. The default value is <b>False</b> . If the value is <b>True</b> , the first non-null value in the window is returned.
partition_clause	No	Partition. Rows with the same value in partition columns are considered to be in the same window.
orderby_clause	No	It is used to specify how data is sorted in a window.
frame_clause	No	It is used to determine the data boundary.

## Return Values

The return value is of the data type of the parameter.

## Example Code

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the **logs** table and add data:

```
create table logs(  
  cookieid string,  
  createtime string,  
  url string  
)  
STORED AS parquet;
```

Adds the following data:

```
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: Groups all records by **cookieid**, sorts the records by **createtime** in ascending order, and returns the last row of data in each group. An example command is as follows:

```
SELECT cookieid, createtime, url,  
  LAST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS last  
FROM logs;
```

```
-- Returned 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 in the current row is actually just itself.

## 14.5.5 lag

This function is used to return the value of the  $n$ th row upwards within a specified window.

### Restrictions

The restrictions on using window functions are as follows:

- Window functions can be used only in select statements.
- Window functions and aggregate functions cannot be nested in window functions.
- Window functions cannot be used together with aggregate functions of the same level.

### Syntax

```
lag(<expr>[, bigint <offset>[, <default>]]) over([partition_clause] orderby_clause)
```

### Parameters

**Table 14-133** Parameters

Parameter	Mandatory	Description
expr	Yes	Expression whose return result is to be calculated
offset	No	Offset. It is a constant of the BIGINT type and its value is greater than or equal to 0. The value <b>0</b> indicates the current row, the value <b>1</b> indicates the previous row, and so on. The default value is <b>1</b> . If the input value is of the STRING or DOUBLE type, it is implicitly converted to the BIGINT type before calculation.

Parameter	Mandatory	Description
default	Yes	Constant. The default value is <b>NULL</b> . Default value when the range specified by <b>offset</b> is out of range. The value must be the same as the data type corresponding to <b>expr</b> . If <b>expr</b> is non-constant, the evaluation is performed based on the current row.
partition_clause	No	Partition. Rows with the same value in partition columns are considered to be in the same window.
orderby_clause	No	It is used to specify how data is sorted in a window.

## Return Values

The return value is of the data type of the parameter.

## Example Code

### Example data

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the **logs** table and add data:

```
create table logs(  
  cookieid string,  
  createtime string,  
  url string  
)  
STORED AS parquet;
```

Adds the following data:

```
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
```

Groups all records by **cookieid**, sorts the records by **createtime** in ascending order, and returns the value of the second row above the window. An example command is as follows:

Example 1:

```
SELECT cookieid, createtime, url,  
  LAG(createtime, 2) OVER (PARTITION BY cookieid ORDER BY createtime) AS last_2_time
```

```
FROM logs;
-- Returned 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

Note: Because no default value is set, **NULL** is returned when the preceding two rows do not exist.

### 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 cookie4;
-- Result:
cookieid createtime      url last_1_time
cookie1 2015-04-10 10:00:00 url1 1970-01-01 00:00:00 (The default value is 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 (The default value is 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
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
```

## 14.5.6 lead

This function is used to return the value of the  $n$ th row downwards within a specified window.

### Restrictions

The restrictions on using window functions are as follows:

- Window functions can be used only in select statements.
- Window functions and aggregate functions cannot be nested in window functions.
- Window functions cannot be used together with aggregate functions of the same level.

### Syntax

```
lead(<expr>[, bigint <offset>[, <default>]]) over([partition_clause] orderby_clause)
```

## Parameters

**Table 14-134** Parameters

Parameter	Mandatory	Description
expr	Yes	Expression whose return result is to be calculated
offset	No	Offset. It is a constant of the BIGINT type and its value is greater than or equal to 0. The value <b>0</b> indicates the current row, the value <b>1</b> indicates the previous row, and so on. The default value is <b>1</b> . If the input value is of the STRING or DOUBLE type, it is implicitly converted to the BIGINT type before calculation.
default	Yes	Constant. The default value is <b>NULL</b> . Default value when the range specified by <b>offset</b> is out of range. The value must be the same as the data type corresponding to <b>expr</b> . If <b>expr</b> is non-constant, the evaluation is performed based on the current row.
partition_clause	No	Partition. Rows with the same value in partition columns are considered to be in the same window.
orderby_clause	No	It is used to specify how data is sorted in a window.

## Return Values

The return value is of the data type of the parameter.

## Example Code

### Example data

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the **logs** table and add data:

```
create table logs(
  cookieid string,
  createtime string,
  url string
)
STORED AS parquet;
```

Adds the following data:

```
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
```

Groups all records by **cookieid**, sorts them by **createtime** in ascending order, and returns the values of the second and first rows downwards within the specified window. An example command is as follows:

```
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;
```

-- Returned 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

## 14.5.7 percent\_rank

This function is used to return the value of the column specified in the ORDER BY clause of a window, expressed as a decimal between 0 and 1. It is calculated as (the rank value of the current row within the group - 1) divided by (the total number of rows in the group - 1).

### Restrictions

The restrictions on using window functions are as follows:

- Window functions can be used only in select statements.
- Window functions and aggregate functions cannot be nested in window functions.
- Window functions cannot be used together with aggregate functions of the same level.

### Syntax

```
percent_rank() over([partition_clause] [orderby_clause])
```

## Parameters

**Table 14-135** Parameters

Parameter	Mandatory	Description
partition_clause	No	Partition. Rows with the same value in partition columns are considered to be in the same window.
orderby_clause	No	It is used to specify how data is sorted in a window.

## Return Values

The return value is of the DOUBLE type.

## Example Code

### Example data

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the salary table and add data:

```
CREATE EXTERNAL TABLE salary (  
  dept STRING, -- Department name  
  userid string, -- Employee ID  
  sal INT -- Salary  
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ';'   
stored as textfile;
```

Adds the following data:

```
d1,user1,1000  
d1,user2,2000  
d1,user3,3000  
d2,user4,4000  
d2,user5,5000
```

Example: Calculates the percentage ranking of employees' salaries in a department.

```
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
```

## 14.5.8 rank

This function is used to return the rank of a value in a set of values. When multiple values share the same rank, the next rank in the sequence is not consecutive.

## Restrictions

The restrictions on using window functions are as follows:

- Window functions can be used only in select statements.
- Window functions and aggregate functions cannot be nested in window functions.
- Window functions cannot be used together with aggregate functions of the same level.

## Syntax

```
rank() over ([partition_clause] [orderby_clause])
```

## Parameters

Table 14-136 Parameters

Parameter	Mandator y	Description
partition_clause	No	Partition. Rows with the same value in partition columns are considered to be in the same window.
orderby_clause	No	It is used to specify how data is sorted in a window.

## Return Values

The return value is of the INT type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the **logs** table and add data:

```
CREATE TABLE logs (  
  cookieid string,  
  createtime string,  
  pv INT  
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
stored as textfile;
```

Adds the following data:

```
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: Groups all records by **cookieid**, sorts them by **pv** in descending order, and returns the sequence number of each row in the group. An example command is as follows:

```
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 (third in parallel)
cookie1 2015-04-15 4 3
cookie1 2015-04-13 3 5 (skip 4 and start from 5)
cookie1 2015-04-14 2 6
cookie1 2015-04-10 1 7
```

## 14.5.9 row\_number

This function is used to return the row number, starting from 1 and increasing incrementally.

### Restrictions

The restrictions on using window functions are as follows:

- Window functions can be used only in select statements.
- Window functions and aggregate functions cannot be nested in window functions.
- Window functions cannot be used together with aggregate functions of the same level.

### Syntax

```
row_number() over([partition_clause] [orderby_clause])
```

### Parameters

**Table 14-137** Parameters

Parameter	Mandatory	Description
partition_clause	No	Partition. Rows with the same value in partition columns are considered to be in the same window.
orderby_clause	No	It is used to specify how data is sorted in a window.

## Return Values

The return value is of the DOUBLE type.

### NOTE

If the value of **a** is **NULL**, **NULL** is returned.

## Example Code

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the **logs** table and add data:

```
CREATE TABLE logs (  
  cookieid string,  
  createtime string,  
  pv INT  
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
stored as textfile;
```

Adds the following data:

```
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: Groups all records by **cookieid**, sorts them by **pv** in descending order, and returns the sequence number of each row in the group. An example command is as follows:

```
select cookieid, createtime, pv,  
       row_number() over (partition by cookieid order by pv desc) as index  
from logs;
```

```
-- Returned 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
```

## 14.6 Other Functions

## 14.6.1 Overview

The following table lists the functions provided by DLI, such as **decode1**, **javahash**, and **max\_pt**.

**Table 14-138** Added functions

Function	Syntax	Value Type	Description
<b>decode1</b>	decode1(<expression>, <search>, <result>[, <search>, <result>]...[, <default>])	Data type of the argument	Implements if-then-else branch selection.
<b>javahash</b>	javahash(string a)	STRING	Returns a hash value.
<b>max_pt</b>	max_pt(<table_full_name>)	STRING	Returns the name of the largest level-1 partition that contains data in a partitioned table and reads the data of this partition.
<b>ordinal</b>	ordinal(bigint <nth>, <var1>, <var2>[,...])	DOUBLE or DATETIME	Sorts input variables in ascending order and returns the value at the position specified by nth.
<b>trans_array</b>	trans_array (<num_keys>, <separator>, <key1>,<key2>, ...,<col1>,<col2>, <col3>) as (<key1>,<key2>,... ,<col1>, <col2>)	Data type of the argument	Converts an array split by a fixed separator in a column into multiple rows.
<b>trunc_numeric</b>	trunc_numeric(<number>[, bigint<decimal_places>])	DOUBLE or DECIMAL	Truncates the <b>number</b> value to a specified decimal place.
<b>url_decode</b>	url_decode(string <input>[, string <encoding>])	STRING	Converts a string from the <b>application/x-www-form-urlencoded MIME</b> format to regular characters.
<b>url_encode</b>	url_encode(string <input>[, string <encoding>])	STRING	Encodes a string in the <b>application/x-www-form-urlencoded MIME</b> format.

## 14.6.2 decode1

This function is used to implement if-then-else branch selection.

### Syntax

```
decode1(<expression>, <search>, <result>[, <search>, <result>]...[, <default>])
```

### Parameters

Table 14-139 Parameters

Parameter	Man dato ry	Type	Description
expression	Yes	All data types	Expression to be compared
search	Yes	Same as that of <b>expression</b>	Search item to be compared with <b>expression</b>
result	Yes	All data types	Return value when the values of <b>search</b> and <b>expression</b> match
default	No	Same as that of <b>result</b>	If all search items do not match, the value of this parameter is returned. If no search item is specified, <b>NULL</b> is returned.

### Return Values

**result** and **default** are return values. These values can be of any data type.

#### NOTE

- If they match, the value of **result** is returned.
- If no match is found, the value of **default** is returned.
- If **default** is not specified, **NULL** is returned.
- If the search options are duplicate and matched, the first value is returned.

### Example Code

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the salary table and add data:

```
CREATE EXTERNAL TABLE salary (  
  dept_id STRING, -- Department  
  userid string, -- Employee ID  
  sal INT
```

```
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
stored as textfile;
```

Adds the following data:

```
d1,user1,1000  
d1,user2,2000  
d1,user3,3000  
d2,user4,4000  
d2,user5,5000
```

### Example

Returns the name of each department.

If **dept\_id** is set to **d1**, **DLI** is returned. If it is set to **d2**, **MRS** is returned. In other scenarios, **Others** is returned.

```
select dept, decode1(dept, 'd1', 'DLI', 'd2', 'MRS', 'Others') as result from sale_detail;
```

Returned result:

```
d1 DLI  
d2 MRS  
d3 Others  
d4 Others  
d5 Others
```

## 14.6.3 javahash

This function is used to return the hash value of **a**.

### Syntax

```
javahash(string a)
```

### Parameters

Table 14-140 Parameter

Parameter	Mandatory	Type	Description
a	Yes	STRING	Data whose hash value needs to be returned

### Return Values

The return value is of the STRING type.

#### NOTE

The hash value is returned. If the value of **a** is **null**, an error is reported.

### Example Code

The value **48690** is returned.

```
select javahash("123");
```

The value **123** is returned.

```
select javahash(123);
```

## 14.6.4 max\_pt

This function is used to return the name of the largest level-1 partition that contains data in a partitioned table and read the data of this partition.

### Syntax

```
max_pt(<table_full_name>)
```

### Parameters

Table 14-141 Parameter

Parameter	Mandatory	Type	Description
table_full_name	Yes	STRING	Specified table name. You must have the read permission on the table.

### Return Values

The return value is of the STRING type.

#### NOTE

- The value of the largest level-1 partition is returned.
- If a partition is added to a table using the **ALTER TABLE** command, but it does not contain any data, it will not be included in the returned values.

### Example Code

For example, table1 is a partitioned table with partitions of **20120801** and **20120802**, both of which contain data, and the **max\_pt** value in the following statement will be **20120802**. The DLI SQL statement will read data from the partition with pt = 20120802.

An example command is as follows:

```
select * from table1 where pt = max_pt('dbname.table1');
```

It is equivalent to the following statement:

```
select * from table1 where pt = (select max(pt) from dbname.table1);
```

## 14.6.5 ordinal

This function is used to sort input variables in ascending order and return the value at the position specified by **nth**.

## Syntax

```
ordinal(bigint <nth>, <var1>, <var2>[,...])
```

## Parameters

Table 14-142 Parameters

Parameter	Mandatory	Type	Description
nth	Yes	BIGINT	Position value to be returned
var	Yes	BIGINT, DOUBLE, DATETIME , or STRING	Value to be sorted

## Return Values

The return value is of the DOUBLE or DECIMAL type.

### NOTE

- Value of the nth bit. If there are no implicit conversions, the return value has the same data type as the input parameter.
- If there are type conversions, **DOUBLE** is returned for the conversion between **DOUBLE**, **BIGINT**, and **STRING**, and **DATETIME** is returned for the conversion between **STRING** and **DATETIME**. Other implicit conversions are not allowed.
- **NULL** indicates the minimum value.

## Example Code

The value 2 is returned.

```
select ordinal(3, 1, 3, 2, 5, 2, 4, 9);
```

## 14.6.6 trans\_array

This function is used to convert an array split by a fixed separator in a column into multiple rows.

## Restrictions

- All columns used as keys must be placed before the columns to be transposed.
- Only one UDTF is allowed in a select statement.
- This function cannot be used together with **group by**, **cluster by**, **distribute by**, or **sort by**.

## Syntax

```
trans_array (<num_keys>, <separator>, <key1>,<key2>,...,<col1>,<col2>,<col3>) as (<key1>,<key2>,...,<col1>,<col2>)
```

## Parameters

Table 14-143 Parameters

Parameter	Mandatory	Type	Description
num_keys	Yes	BIGINT	The value is a constant of the BIGINT type and must be greater than or equal to 0. This parameter indicates the number of columns that are used as transposed keys when being converted to multiple rows.
separator	Yes	STRING	The value is a constant of the STRING type, which is used to split a string into multiple elements. If this parameter is left blank, an error is reported.
keys	Yes	STRING	Columns used as keys during transpose. The number of columns is specified by <b>num_keys</b> . If <b>num_keys</b> specifies that all columns are used as keys (that is, <b>num_keys</b> is equal to the number of all columns), only one row is returned.
cols	Yes	STRING	Array to be converted to rows. All columns following <b>keys</b> are regarded as arrays to be transposed and must be of the STRING type.

## Return Values

The return value is of the data type of the parameter.

### NOTE

- Transposed rows are returned. The new column name is specified by **as**.
- The type of the key column does not change, and the type of other columns is **STRING**.
- The number of rows after the split is subject to the array with more rows. If there are not enough rows, **NULL** is added.

## Example Code

To help you understand how to use functions, this example provides source data and function examples based on the source data. Run the following command to create the salary table and add data:

```
CREATE EXTERNAL TABLE salary (  
dept_id STRING, -- Department
```

```
userid string, -- Employee ID
sal INT -- Salary
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
stored as textfile;
```

Adds the following data:

```
d1,user1/user4,1000/6000
d1,user2/user5,2000/7000
d1,user3/user6,3000
d2,user4/user7,4000
d2,user5/user8,5000/8000
```

Executes the SQL statement

```
select trans_array(1, "/", dept_id, user_id, sal) as (dept_id, user_id, sal) from salary;
```

The command output is as follows:

```
d1,user1,1000
d1,user4,6000
d1,user2,2000
d1,user5,7000
d1,user3,3000
d1,user6,NULL
d2,user4,4000
d2,user7,NULL
d2,user5,5000
d2,user8,8000
```

## 14.6.7 trunc\_numeric

This function is used to truncate the **number** value to a specified decimal place.

### Syntax

```
trunc_numeric(<number>[, bigint<decimal_places>])
```

### Parameters

**Table 14-144** Parameters

Parameter	Mandatory	Type	Description
number	Yes	DOUBLE, BIGINT, DECIMAL, or STRING	Data to be truncated
decimal_places	No	BIGINT	The default value is <b>0</b> , indicating that the decimal place at which the number is truncated.

### Return Values

The return value is of the DOUBLE or DECIMAL type.

 NOTE

The return rules are as follows:

- If the **number** value is of the DOUBLE or DECIMAL type, the corresponding type is returned.
- If the **number** value is of the STRING or BIGINT type, **DOUBLE** is returned.
- If the **decimal\_places** value is not of the BIGINT type, an error is reported.
- If the value of **number** is **NULL**, **NULL** is returned.

## Example Code

The value **3.141** is returned.

```
select trunc_numeric(3.1415926, 3);
```

The value **3** is returned.

```
select trunc_numeric(3.1415926);
```

An error is reported.

```
select trunc_numeric(3.1415926, 3.1);
```

## 14.6.8 url\_decode

This function is used to convert a string from the **application/x-www-form-urlencoded MIME** format to regular characters.

### Syntax

```
url_decode(string <input>[, string <encoding>])
```

### Parameters

Table 14-145 Parameters

Parameter	Mandatory	Type	Description
input	Yes	STRING	String to be entered
encoding	No	STRING	Encoding format. Standard encoding formats such as GBK and UTF-8 are supported. If this parameter is not specified, <b>UTF-8</b> is used by default.

### Return Values

The return value is of the STRING type.

 NOTE

UTF-8-encoded string of the STRING type

## Example Code

The value **Example for URL\_DECODE:// dsf (fasfs)** is returned.

```
select url_decode('Example+for+url_decode+%3A%2F%2F+dsf%28fasfs%29', 'GBK');
```

## 14.6.9 url\_encode

This function is used to encode a string into the **application/x-www-form-urlencoded MIME** format.

### Syntax

```
url_encode(string <input>[, string <encoding>])
```

### Parameters

**Table 14-146** Parameters

Parameter	Mandatory	Type	Description
input	Yes	STRING	String to be entered
encoding	No	STRING	Encoding format. Standard encoding formats such as GBK and UTF-8 are supported. If this parameter is not specified, <b>UTF-8</b> is used by default.

### Return Values

The return value is of the STRING type.

#### NOTE

If the value of **input** or **encoding** is **NULL**, **NULL** is returned.

## Example Code

The value **Example+for+url\_encode+%3A%2F%2F+dsf%28fasfs%29** is returned.

```
select url_encode('Example for url_encode:// dsf(fasfs)', 'GBK');
```

# 15 SELECT

## 15.1 Basic Statements

### Function

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

### Syntax

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

### Keywords

Table 15-1 SELECT keywords

Parameter	Description
ALL	The keyword <b>ALL</b> is used to return all matching rows from a database, including duplicates. It can only be followed by an asterisk (*); otherwise, an error will occur.  This is the default behavior in SQL statements and is typically not written explicitly. If neither <b>ALL</b> nor <b>DISTINCT</b> is specified, the query result will include all rows, even if they are duplicates.
DISTINCT	When the <b>DISTINCT</b> keyword is used in a SELECT statement, the system removes duplicate data from the query results to ensure uniqueness.

Parameter	Description
WHERE	Specifies the filter criteria for a query. Arithmetic operators, relational operators, and logical operators are supported.
where_condition	Filter criteria.
GROUP BY	Specifies the grouping field. Single-field grouping and multi-field grouping are supported.
col_name_list	Field list
ORDER BY	Sort the query results.
ASC/DESC	ASC sorts from the lowest value to the highest value. DESC sorts from the highest value to the lowest value. ASC is the default sort order.
CLUSTER BY	CLUSTER BY is used to bucket the table according to the bucketing fields and then sort within the bucketed table. If the field of DISTRIBUTE BY is the same as the field of SORT BY and the sorting is in descending order, the combination of DISTRIBUTE BY and SORT BY achieves the same function as CLUSTER BY.
DISTRIBUTE BY	Specifies the bucketing fields without sorting the table.
SORT BY	The objects will be sorted in the bucket.
LIMIT	LIMIT is used to limit the query results. Only INT type is supported by the <b>number</b> parameter.

## Precautions

- The table to be queried must already exist, or an error message will be displayed.
- When submitting SQL statements to display binary data on the DLI management console, the binary data is converted to Base64.

## Example

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

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

## 15.2 Sort

## 15.2.1 ORDER BY

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

### Example

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

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

## 15.2.2 SORT BY

### Function

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

### Syntax

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

### Keyword

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

## Precautions

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

## Example

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

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

## 15.2.3 CLUSTER BY

### Function

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

### Syntax

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

### Keyword

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

## Precautions

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

## Example

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

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

## 15.2.4 DISTRIBUTE BY

### Function

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

### Syntax

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

## Keyword

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

## Precautions

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

## Example Value

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

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

# 15.3 Grouping

## 15.3.1 Column-Based GROUP BY

### Function

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

### Syntax

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

### Keywords

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

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

### Precautions

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

## Example

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

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

## 15.3.2 Expression-Based GROUP BY

### Function

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

### Syntax

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

### Keywords

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

### Precautions

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

### Example

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

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

## 15.3.3 Using HAVING in GROUP BY

### Function

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

### Syntax

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

### Keyword

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

## Precautions

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

## Example

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

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

## 15.3.4 ROLLUP

### Function

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

### Syntax

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

### Keyword

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

- Counting the (a, b, c) combinations  

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

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

```
SELECT a, NULL, NULL, sum(expression) FROM table
GROUP BY a;
```
- Total  

```
SELECT NULL, NULL, NULL, sum(expression) FROM table;
```

## Precautions

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

## Example

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

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

## 15.3.5 GROUPING SETS

### Function

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

### Syntax

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

### Keyword

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

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

It can be converted to the following query:

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

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

It can be converted to the following two queries:

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

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

It can be converted to the following two queries:

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

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

It can be converted to the following four queries:

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

## Precautions

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

## Example

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

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

# 15.4 Joins

## 15.4.1 INNER JOIN

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

### Example

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

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

## 15.4.2 LEFT OUTER JOIN

### Function

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

## Syntax

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

## Keyword

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

## Precautions

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

## Example

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

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

## 15.4.3 RIGHT OUTER JOIN

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

### Example

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

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

## 15.4.4 FULL OUTER JOIN

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

### Example

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

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

## 15.4.5 IMPLICIT JOIN

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

## Example

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

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

## 15.4.6 Cartesian JOIN

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

### Example

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

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

## 15.4.7 LEFT SEMI JOIN

### Function

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

### Syntax

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

### Keyword

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

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

## Precautions

- The to-be-joined table must exist. Otherwise, an error is reported.
- The fields in attr\_expr\_list must be the fields in the left table. Otherwise, an error is reported.

## Example

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

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

## 15.4.8 NON-EQUIJOIN

### Function

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

### Syntax

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

### Keyword

The **non\_equi\_join\_condition** is similar to **join\_condition**. The only difference is that the JOIN condition is inequation.

### Precautions

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

### Example

To return all the pairs of different student names from the **student\_info\_1** and **student\_info\_2** tables, run the following statement:

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

## 15.5 Clauses

## 15.5.1 FROM

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

### Example

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

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

## 15.5.2 OVER

### Function

This statement is used together with the window function. The OVER statement is used to group data and sort the data within the group. The window function is used to generate serial numbers for values within the group.

### Syntax

```
SELECT window_func(args) OVER  
([PARTITION BY col_name, col_name, ...]  
[ORDER BY col_name, col_name, ...]  
[ROWS | RANGE BETWEEN (CURRENT ROW | (UNBOUNDED [[num]) PRECEDING)  
AND (CURRENT ROW | ( UNBOUNDED | [num]) FOLLOWING)]);
```

### Keywords

- PARTITION BY: used to partition a table with one or multiple fields. Similar to GROUP BY, PARTITION BY is used to partition table by fields and each

partition is a window. The window function can apply to the entire table or specific partitions. A maximum of 7,000 partitions can be created in a single table.

- **ORDER BY:** used to specify the order for the window function to obtain the value. ORDER BY can be used to sort table with one or multiple fields. The sorting order can be ascending (specified by **ASC**) or descending (specified by **DESC**). The window is specified by WINDOW. If the window is not specified, the default window is ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. In other words, the window starts from the head of the table or partition (if PARTITION BY is used in the OVER clause) to the current row.
- **WINDOW:** used to define the window by specifying a range of rows.
- **CURRENT ROW:** indicates the current row.
- **num PRECEDING:** used to specify the start of the defined window. The window starts from the num row precedes the current row.
- **UNBOUNDED PRECEDING:** used to indicate that there is no start of the window.
- **num FOLLOWING:** used to specify the end of the defined window. The window ends from the num row following the current row.
- **UNBOUNDED FOLLOWING:** used to indicate that there is no end of the window.
- The differences between ROWS BETWEEN... and RANGE BETWEEN... are as follows:
  - ROWS refers to the physical window. After the data is sorted, the physical window starts at the  $n$ th row in front of the current row and ends at the  $m$ th row following the current row.
  - RANGE refers to the logic window. The column of the logic window is determined by the values rather than the location of rows.
- The scenarios of the window are as follows:
  - The window only contains the current row.  
ROWS BETWEEN CURRENT ROW AND CURRENT ROW
  - The window starts from three rows precede the current row and ends at the fifth row follows the current row.  
ROWS BETWEEN 3 PRECEDING AND 5 FOLLOWING
  - The window starts from the beginning of the table or partition and ends at the current row.  
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  - The window starts from the current window and ends at the end of the table or partition.  
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
  - The window starts from the beginning of the table or partition and ends at the end of the table or partition.  
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

## Precautions

The three options of the OVER clause are PARTITION BY, ORDER BY, and WINDOW. They are optional and can be used together. If the OVER clause is empty, the window is the entire table.

## Example

To start the window from the beginning of the table or partition and end the window at the current row, sort the `over_test` table according to the `id` field, and return the sorted `id` fields and corresponding serial numbers, run the following statement:

```
SELECT id, count(id) OVER (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM over_test;
```

## 15.5.3 WHERE

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

## Example

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

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

## 15.5.4 HAVING

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

### Example

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

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

## 15.5.5 Multi-Layer Nested Subquery

### Function

This statement is used to nest queries in the subquery.

### Syntax

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

### Keyword

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

- DISTINCT is used to remove the repeated line from the result.

## Precautions

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

## Example

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

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

# 15.6 Alias

## 15.6.1 Table Alias

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

### Example

- To specify alias **n** for table **simple\_table** and visit the name field in table **simple\_table** by using n.name, run the following statement:  

```
SELECT n.score FROM simple_table n WHERE n.name = "leilei";
```
- To specify alias **m** for the subquery result and return all the query results using **SELECT \* FROM m**, run the following statement:  

```
SELECT * FROM (SELECT * FROM simple_table WHERE score > 90) AS m;
```

## 15.6.2 Column Alias

### Function

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

### Syntax

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

### Keyword

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

### Precautions

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

### Example

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

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

## 15.7 Set Operations

### 15.7.1 UNION

#### Function

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

#### Syntax

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

#### Keyword

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

#### Precautions

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

## Example

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

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

## 15.7.2 INTERSECT

### Function

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

### Syntax

```
select_statement INTERSECT select_statement;
```

### Keyword

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

### Precautions

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

## Example

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

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

## 15.7.3 EXCEPT

### Function

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

### Syntax

```
select_statement EXCEPT select_statement;
```

### Keywords

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

## Precautions

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

## Example

To remove the records that exist in both **SELECT \* FROM student\_1** and **SELECT \* FROM student\_2** from **SELECT \* FROM student\_1** and return the results, run the following statement:

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

# 15.8 WITH...AS

## Function

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

## Syntax

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

## Keyword

- **cte\_name**: Name of a public expression. The name must be unique.
- **select\_statement**: complete SELECT clause.
- **sql\_containing\_cte\_name**: SQL statement containing the defined common expression.

## Precautions

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

## Example

Define **SELECT courseId FROM course\_info WHERE courseName = 'Biology'** as CTE **nv** and use **nv** as the SELECT statement in future queries.

```
WITH nv AS (SELECT courseId FROM course_info WHERE courseName = 'Biology') SELECT DISTINCT courseId FROM nv;
```

# 15.9 CASE...WHEN

## 15.9.1 Basic CASE Statement

### Function

This statement is used to display **result\_expression** according to the joined results of **input\_expression** and **when\_expression**.

### Syntax

```
CASE input_expression WHEN when_expression THEN result_expression [...] [ELSE else_result_expression] END;
```

### Keyword

CASE: Subquery is supported in basic CASE statement. However, **input\_expression** and **when\_expression** must be joinable.

### Precautions

If there is no **input\_expression = when\_expression** with the TRUE value, **else\_result\_expression** will be returned when the ELSE clause is specified. If the ELSE clause is not specified, NULL will be returned.

### Example

To return the name field and the character that is matched to id from the student table with the following matching rules, run the following statement:

- If id is 1, 'a' is returned.
- If id is 2, 'b' is returned.
- If id is 3, 'c' is returned.
- Otherwise, **NULL** is returned.

```
SELECT name, CASE id WHEN 1 THEN 'a' WHEN 2 THEN 'b' WHEN 3 THEN 'c' ELSE NULL END FROM student;
```

## 15.9.2 CASE Query Statement

### Function

This statement is used to obtain the value of **boolean\_expression** for each WHEN statement in a specified order. Then return the first **result\_expression** with the value **TRUE** of **boolean\_expression**.

### Syntax

```
CASE WHEN boolean_expression THEN result_expression [...] [ELSE else_result_expression] END;
```

### Keyword

**boolean\_expression**: can include subquery. However, the return value of **boolean\_expression** can only be of Boolean type.

## Precautions

If there is no Boolean\_expression with the TRUE value, else\_result\_expression will be returned when the ELSE clause is specified. If the ELSE clause is not specified, NULL will be returned.

## Example

To query the student table and return the related results for the name and score fields: EXCELLENT if the score is higher than 90, GOOD if the score ranges from 80 to 90, and BAD if the score is lower than 80, run the following statement:

```
SELECT name, CASE WHEN score >= 90 THEN 'EXCELLENT' WHEN 80 < score AND score < 90 THEN 'GOOD'  
ELSE 'BAD' END AS level FROM student;
```

# 16 Identifiers

---

## 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

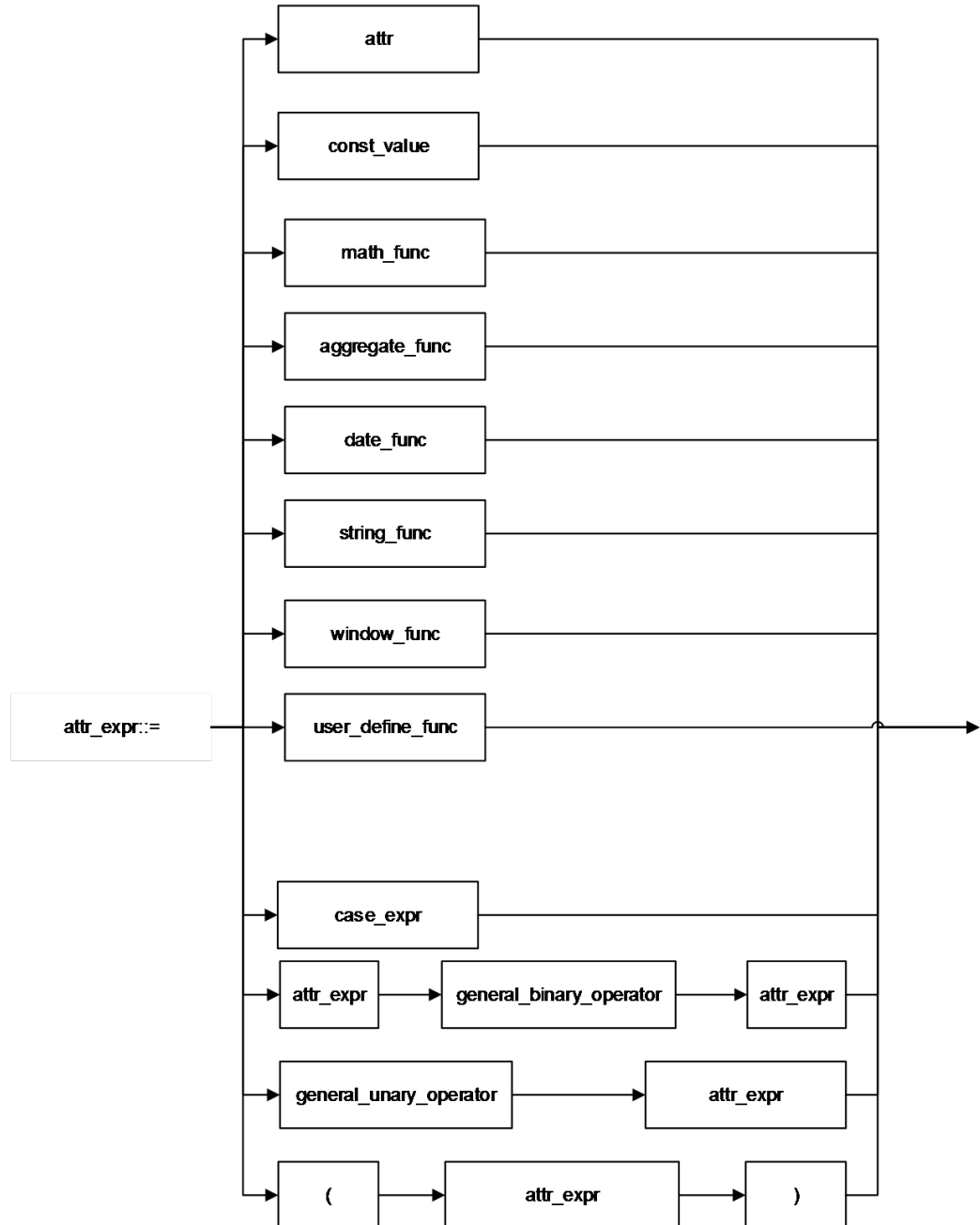
None.

### Description

Alias, which must be STRING type. It can be assigned to a field, table, view, or subquery.

## 16.3 attr\_expr

### Syntax



### Description

Syntax	Description
attr_expr	Attribute expression.

Syntax	Description
attr	Table field, which is the same as col_name.
const_value	Constant value.
case_expr	Case expression.
math_func	Mathematical function.
date_func	Date function.
string_func	String function.
aggregate_func	Aggregate function.
window_func	Analysis window function.
user_define_func	User-defined function.
general_binary_operator	General binary operator.
general_unary_operator	General unary operator.
(	Start of the specified subattribute expression.
)	End of the specified subattribute expression.

## 16.4 attr\_expr\_list

### Syntax

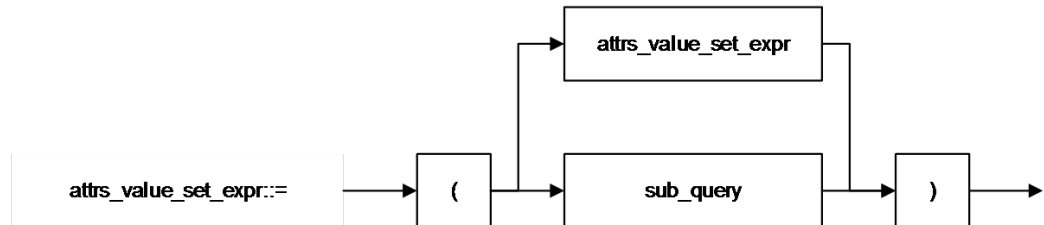
None.

### Description

List of attr\_expr, which is separated by commas (,).

## 16.5 attrs\_value\_set\_expr

### Syntax



### Description

Syntax	Description
attrs_value_set_expr	Collection of attribute values.
sub_query	Subquery clause.
(	Start of the specified subquery expression.
)	End of the specified subquery expression.

## 16.6 boolean\_expression

### Syntax

None.

### Description

Return a boolean expression.

## 16.7 class\_name

### Syntax

None.

### Description

Name of the dependent class of a function. A class name must contain the full path to the package where the class resides.

## 16.8 col

### Syntax

None.

### Description

Formal parameter for function call. It is usually a field name, which is the same as **col\_name**.

## 16.9 col\_comment

### Syntax

None.

### Description

Column (field) description, which must be STRING type and cannot exceed 256 bytes.

## 16.10 col\_name

### Syntax

None.

### Description

Column name, which must be STRING type and cannot exceed 128 bytes.

## 16.11 col\_name\_list

### Syntax

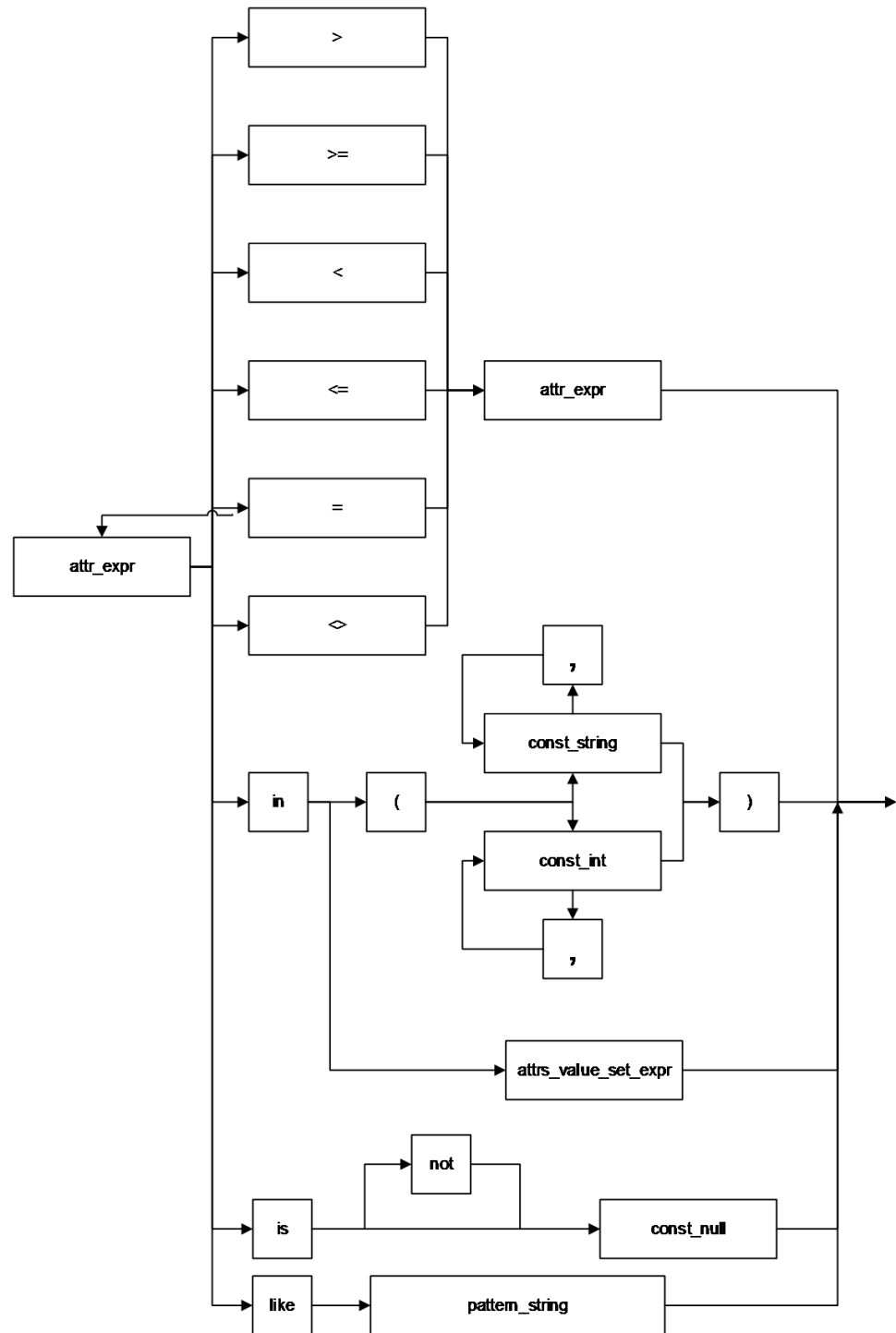
None.

### Description

Field list, which consists of one **col\_name** or more. If there is more than one **col\_name**, separate them by using a comma (,).

## 16.12 condition

### Syntax

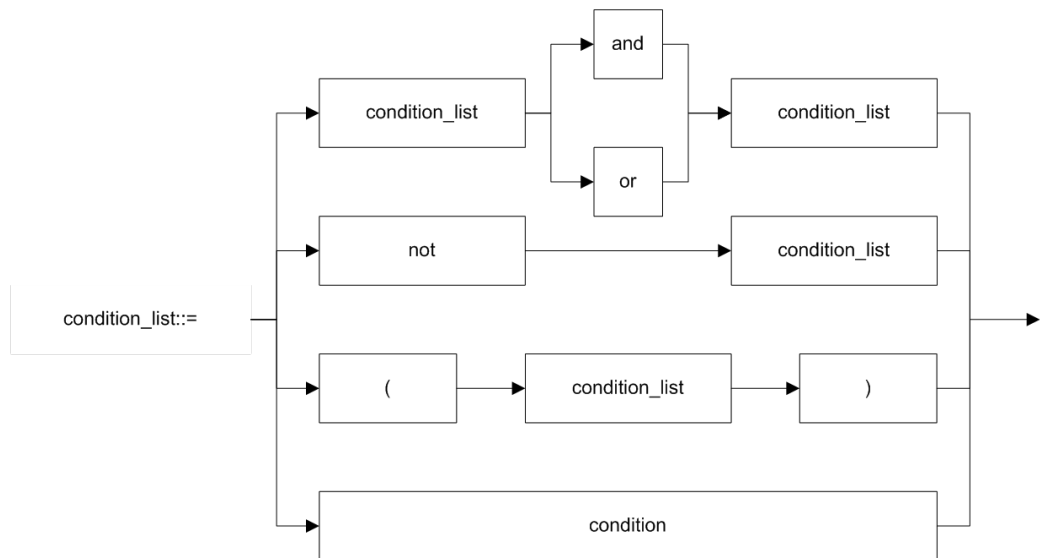


## Description

Syntax	Description
condition	Judgment condition.
>	Relational operator: >
>=	Relational operator: ≥
<	Relational operator: <
<=	Relational operator: ≤
=	Relational operator: =
<>	Relational operator: <>
is	Relational operator: is
is not	Relational operator: is not
const_null	Constant value: null
like	Relational operator: used for wildcard matching.
pattern_string	Pattern matching string, which supports wildcard matching. In WHERE LIKE, SQL wildcard characters "%" and "_" are supported. "%" represents one or more characters. "_" represents only one character.
attr_expr	Attribute expression.
attrs_value_set_expr	Collection of attribute values.
in	Keyword used to determine whether attributes are in the same collection.
const_string	String constant.
const_int	Integer constant.
(	Start of the specified constant collection.
)	End of the specified constant collection.
,	Separator comma (,)

## 16.13 condition\_list

### Syntax



### Description

Syntax	Description
condition_list	List of judgment conditions.
and	Logical operator: AND
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjgment condition.
)	End of the subjgment condition.
condition	Judgment condition.

## 16.14 cte\_name

### Syntax

None.

### Description

Common expression name.

## 16.15 data\_type

### Syntax

None.

### Description

Data type. Currently, only the primitive data types are supported.

## 16.16 db\_comment

### Syntax

None.

### Description

Database description, which must be STRING type and cannot exceed 256 characters.

## 16.17 db\_name

### Syntax

None.

### Description

Database name, which must be STRING type and cannot exceed 128 bytes.

## 16.18 else\_result\_expression

### Syntax

None.

### Description

Returned result for the **ELSE** clause of the **CASE WHEN** statement.

## 16.19 file\_format

### Format

| AVRO

| CSV  
| JSON  
| ORC  
| PARQUET

## Description

- Currently, the preceding formats are supported.
- Both **USING** and **STORED AS** can be used for specifying the data format. You can specify the preceding data formats by **USING**, but only the **ORC** and **PARQUET** formats by **STORED AS**.
- **ORC** has optimized **RFile** to provide an efficient method to store **Hive** data.
- **PARQUET** is an analytical service-oriented and column-based storage format.

## 16.20 file\_path

### Syntax

None.

### Description

File path, which is the OBS path

## 16.21 function\_name

### Syntax

None.

### Description

Function name, which must be STRING type.

## 16.22 groupby\_expression

### Syntax

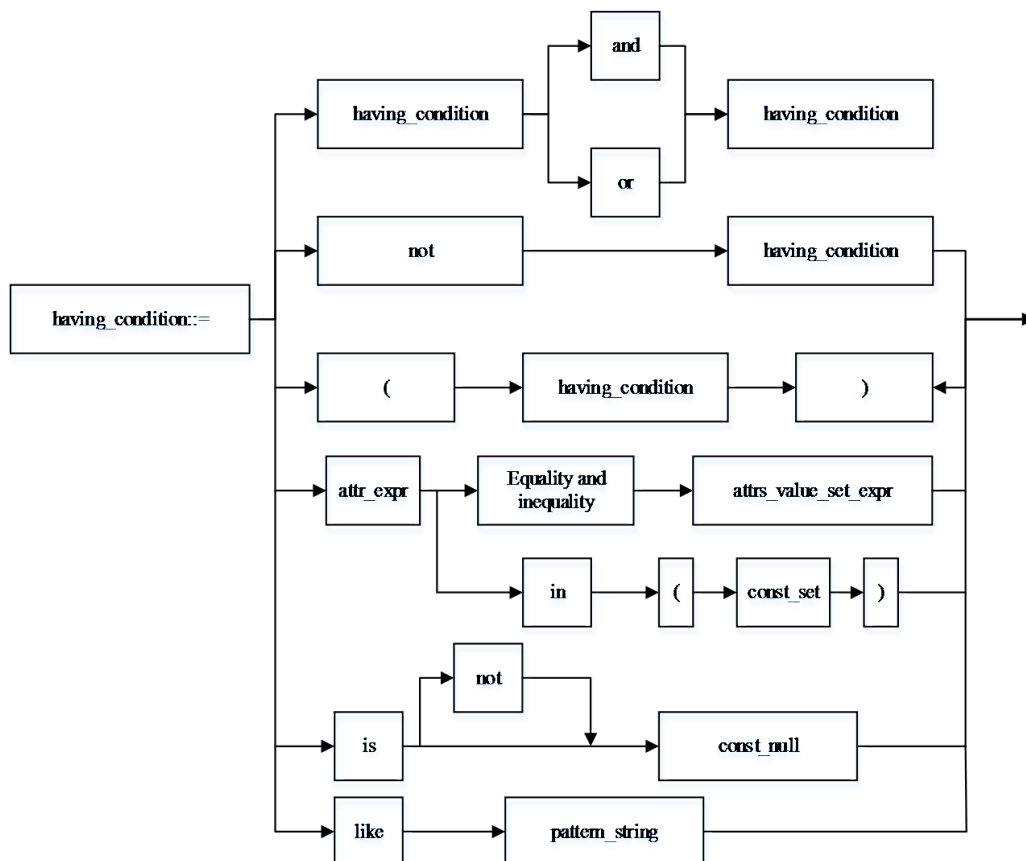
None.

### Description

Expression that includes GROUP BY.

## 16.23 having\_condition

### Syntax



### Description

Syntax	Description
having_condition	Judgment condition of <b>having</b> .
and	Logical operator: AND
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjgment condition.
)	End of the subjgment condition.
condition	Judgment condition.
const_set	Collection of constants, which are separated by using comma (,).

Syntax	Description
in	Keyword used to determine whether attributes are in the same collection.
attrs_value_set_expr	Collection of attribute values.
attr_expr	Attribute expression.
Equality and inequality	Equation and inequality. For details, see <a href="#">Relational Operators</a> .
pattern_string	Pattern matching string, which supports wildcard matching. In WHERE LIKE, SQL wildcard characters "%" and "_" are supported. "%" represents one or more characters. "_" represents only one character.
like	Relational operator: used for wildcard matching.

## 16.24 hdfs\_path

### Syntax

None.

### Description

HDFS path, for example, **hdfs:///tmp**.

## 16.25 input\_expression

### Syntax

None.

### Description

Input expression of the **CASE WHEN** statement.

## 16.26 input\_format\_classname

### Syntax

None.

### Description

The name of a class, for example, ***org.apache.hadoop.mapred.TextInputFormat***, that specifies the input format.

## 16.27 jar\_path

### Syntax

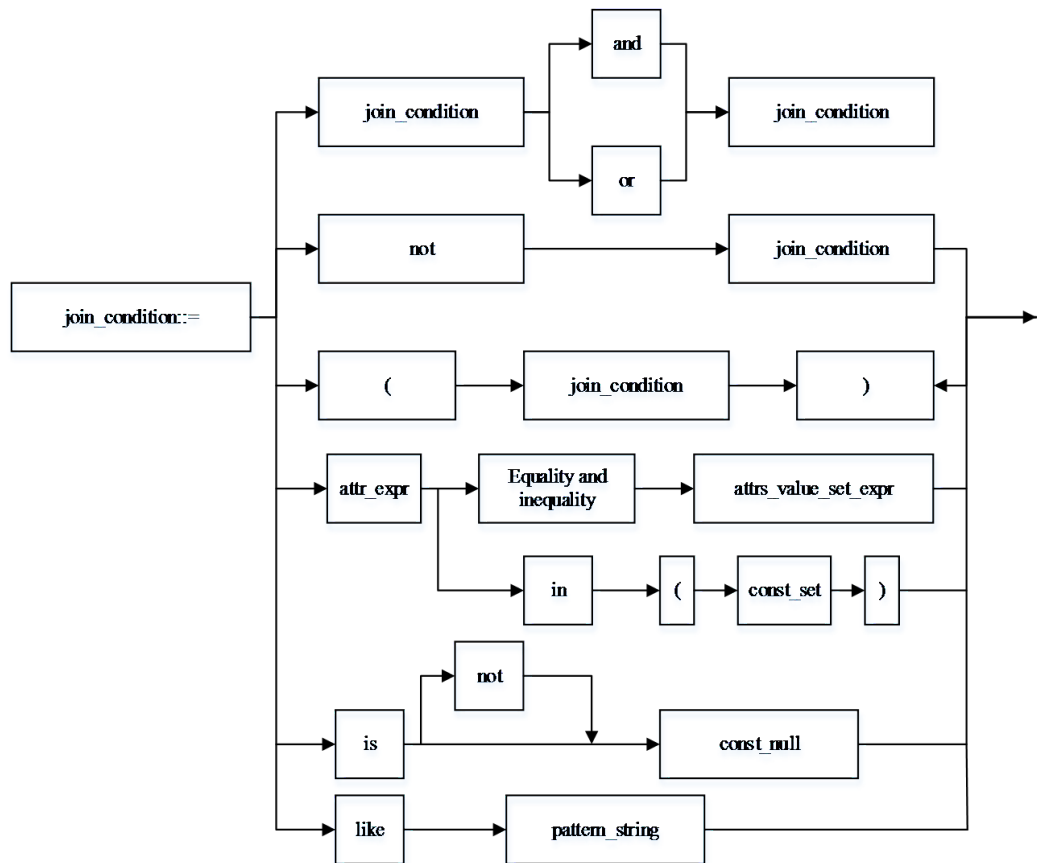
None.

### Description

Path to a JAR package, which can be a local path or HDFS path.

## 16.28 join\_condition

### Syntax



### Description

Syntax	Description
join_condition	Judgment condition of <b>join</b> .
and	Logical operator: AND

Syntax	Description
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjgment condition.
)	End of the subjgment 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.
atrrs_value_set_expr	Collection of attribute values.
attr_expr	Attribute expression.
Equality and inequality	Equation and inequality. For details, see <a href="#">Relational Operators</a> .
pattern_string	Pattern matching string, which supports wildcard matching. In WHERE LIKE, SQL wildcard characters "%" and "_" are supported. "%" represents one or more characters. "_" represents only one character.

## 16.29 non\_equi\_join\_condition

### Syntax

None.

### Description

The condition of an inequality join.

## 16.30 number

### Syntax

None.

### Description

Maximum number of output lines specified by **LIMIT**. Which must be INT type.

## 16.31 num\_buckets

### Syntax

None.

### Description

Buckets number. Which must be INT type.

## 16.32 output\_format\_classname

### Syntax

None.

### Description

The name of a class, for example, *org.apache.hadoop.hive.qi.io.HiveIgnoreKeyTextOutputFormat*, that specifies the output format.

## 16.33 partition\_col\_name

### Syntax

None.

### Description

Partition column name, that is, partition field name, which must be STRING type.

## 16.34 partition\_col\_value

### Syntax

None.

### Description

Partition column value, that is, partition field value.

## 16.35 partition\_specs

### Syntax

```
partition_specs : (partition_col_name = partition_col_value, partition_col_name = partition_col_value, ...);
```

### Description

Table partition list, which is expressed by using key=value pairs, in which **key** represents **partition\_col\_name**, and **value** represents **partition\_col\_value**. If there is more than one partition field, separate every two key=value pairs by using a comma (,).

## 16.36 property\_name

### Syntax

None.

### Description

Property name, which must be STRING type.

## 16.37 property\_value

### Syntax

None.

### Description

Property value, which must be STRING type.

## 16.38 regex\_expression

### Syntax

None.

### Description

Pattern matching string, which supports wildcard matching.

## 16.39 result\_expression

### Syntax

None.

### Description

Returned result for the **THEN** clause of the **CASE WHEN** statement.

## 16.40 row\_format

### 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** indicates the separator or substitute character in the syntax. Only the **CHAR** type is supported.
- **FIELDS TERMINATED BY** specifies the delimiter of field levels in a table. Only the **CHAR** type is supported.
- **COLLECTION ITEMS TERMINATED BY** specifies the delimiter of the set level. Only the **CHAR** type is supported.
- **MAP KEY TERMINATED BY** specifies the separator between the key and value in the **MAP** type. Only the **CHAR** type is supported.
- **LINES TERMINATED BY** indicates the separator between lines. Only **\n** is currently supported.
- 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' = '')

## 16.41 select\_statement

### Syntax

None.

### Description

Query clause for the basic **SELECT** statement.

## 16.42 separator

### Syntax

None.

### Description

Separator, which can be customized by users, for example, comma (,), semicolon (;), and colon (:). Which must be CHAR type.

## 16.43 serde\_name

### Syntax

None.

### Description

Name of serde.

## 16.44 sql\_containing\_cte\_name

### Syntax

None.

### Description

SQL statement containing the common expression defined by cte\_name.

## 16.45 sub\_query

### Syntax

None.

## Description

Subquery.

## 16.46 table\_comment

### Syntax

None.

### Description

Table description, which must be STRING type and 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.

## 16.48 table\_properties

### Syntax

None.

### Description

Table property list, which is expressed by using key=value pairs. key represents **property\_name**, and value represents **property\_value**. If there is more than one key=value pair, separate every two key=value pairs by using a comma (,).

## 16.49 table\_reference

### Syntax

None.

### Description

Table or view name, which must be STRING type. It can also be a subquery. If it is subquery, an alias must also be provided.

## 16.50 view\_name

### Syntax

None.

### Description

View name, which must be STRING type and cannot exceed 128 bytes.

## 16.51 view\_properties

### Syntax

None.

### Description

View property list, which is expressed by using key=value pairs. key represents **property\_name**, and value represents **property\_value**. If there is more than one key=value pair, separate every two key=value pairs by using a comma (,).

## 16.52 when\_expression

### Syntax

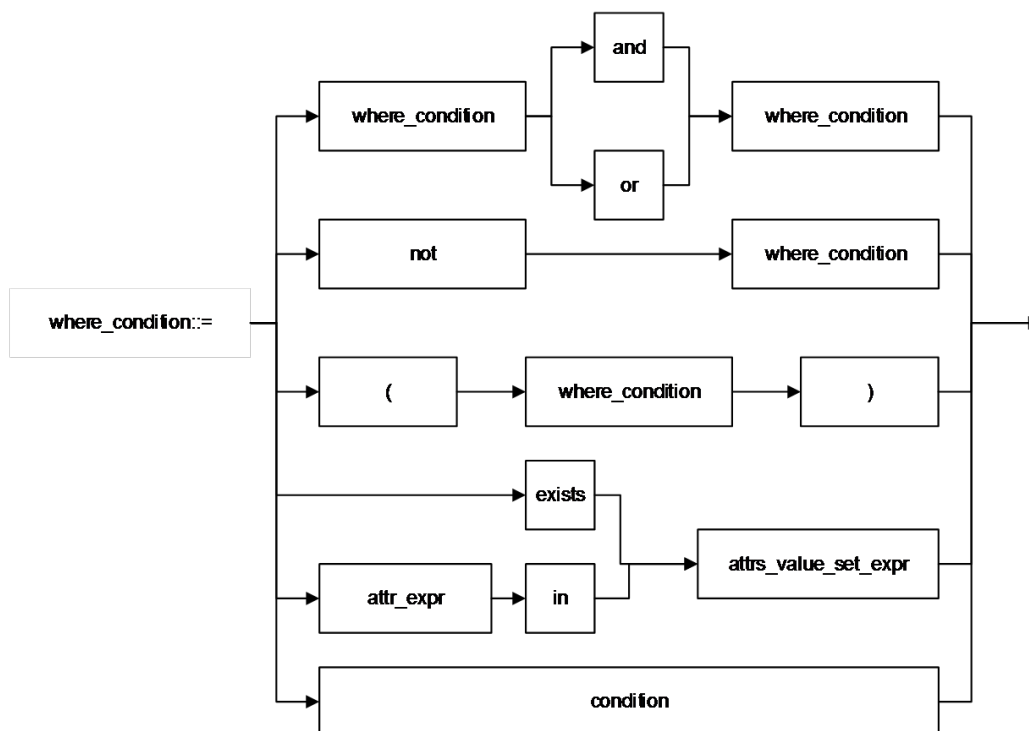
None.

### Description

When expression of the **CASE WHEN** statement. It is used for matching with the input expression.

## 16.53 where\_condition

### Syntax



### Description

Syntax	Description
where_condition	Judgment condition of <b>where</b> .
and	Logical operator: AND
or	Logical operator: OR
not	Logical operator: NOT
(	Start of the subjgment condition.
)	End of the subjgment condition.
condition	Judgment condition.
exists	Keyword used to determine whether a non-empty collection exists. If exists is followed by a subquery, then the subquery must contain a judgment condition.
in	Keyword used to determine whether attributes are in the same collection.
attrs_value_set_expr	Collection of attribute values.

Syntax	Description
attr_expr	Attribute expression.

## 16.54 window\_function

### Syntax

None

### Description

Analysis window function.

# 17 Operators

## 17.1 Relational Operators

All data types can be compared by using relational operators and the result is returned as a BOOLEAN value.

Relationship operators are binary operators. Two compared data types must be of the same type or they must support implicit conversion.

[Table 17-1](#) lists the relational operators provided by DLI.

**Table 17-1** Relational operators

Operator	Result Type	Description
A = B	BOOLEAN	If A is equal to B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. This operator is used for value assignment.
A == B	BOOLEAN	If A is equal to B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. This operator cannot be used for value assignment.
A <=> B	BOOLEAN	If A is equal to B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A and B are <b>NULL</b> , then <b>TRUE</b> is returned. If A or B is <b>NULL</b> , then <b>FALSE</b> is returned.
A <> B	BOOLEAN	If A is not equal to B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned. This operator follows the standard SQL syntax.
A != B	BOOLEAN	This operator is the same as the <> logical operator. It follows the SQL Server syntax.

Operator	Result Type	Description
A < B	BOOLEAN	If A is less than B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned.
A <= B	BOOLEAN	If A is less than or equal to B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned.
A > B	BOOLEAN	If A is greater than B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned.
A >= B	BOOLEAN	If A is greater than or equal to B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned.
A BETWEEN B AND C	BOOLEAN	If A is greater than or equal to B and less than or equal to C, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A, B, or C is <b>NULL</b> , then <b>NULL</b> is returned.
A NOT BETWEEN B AND C	BOOLEAN	If A is less than B or greater than C, <b>TRUE</b> is returned; otherwise, <b>FALSE</b> is returned. If A, B, or C is <b>NULL</b> , then <b>NULL</b> is returned.
A IS NULL	BOOLEAN	If A is <b>NULL</b> , then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned.
A IS NOT NULL	BOOLEAN	If A is not <b>NULL</b> , then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned.
A LIKE B	BOOLEAN	If A matches B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned.
A NOT LIKE B	BOOLEAN	If A does not match B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned.
A RLIKE B	BOOLEAN	This operator is used for the LIKE operation of JAVA. If A or its substring matches B, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned.
A REGEXP B	BOOLEAN	The result is the same as A RLIKE B.

## 17.2 Arithmetic Operators

Arithmetic operators include binary operators and unary operators. For both types of operators, the returned results are numbers. [Table 17-2](#) lists the arithmetic operators supported by DLI.

**Table 17-2** Arithmetic operators

Operator	Result Type	Description
A + B	All numeric types	A plus B. The result type is associated with the operation data type. For example, if floating-point number is added to an integer, the result will be a floating-point number.
A - B	All numeric types	A minus B. The result type is associated with the operation data type.
A * B	All numeric types	Multiply A and B. The result type is associated with the operation data type.
A / B	All numeric types	Divide A by B. The result is a number of the double type (double-precision number).
A % B	All numeric types	A on the B Modulo. The result type is associated with the operation data type.
A & B	All numeric types	Check the value of the two parameters in binary expressions and perform the AND operation by bit. If the same bit of both expressions are 1, then the bit is set to 1. Otherwise, the bit is 0.
A   B	All numeric types	Check the value of the two parameters in binary expressions and perform the OR operation by bit. If one bit of either expression is 1, then the bit is set to 1. Otherwise, the bit is set to 0.
A ^ B	All numeric types	Check the value of the two parameters in binary expressions and perform the XOR operation by bit. Only when one bit of either expression is 1, the bit is 1. Otherwise, the bit is 0.
~A	All numeric types	Perform the NOT operation on one expression by bit.

## 17.3 Logical Operators

Common logical operators include AND, OR, and NOT. The operation result can be TRUE, FALSE, or NULL (which means unknown). The priorities of the operators are as follows: NOT > AND > OR.

**Table 17-3** lists the calculation rules, where A and B represent logical expressions.

**Table 17-3** Logical operators

Operator	Result Type	Description
A AND B	BOOLEAN	If A and B are <b>TRUE</b> , then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned.
A OR B	BOOLEAN	If A or B is <b>TRUE</b> , then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned. If A or B is <b>NULL</b> , then <b>NULL</b> is returned. If one is <b>TRUE</b> and the other is <b>NULL</b> , then <b>TRUE</b> is returned.
NOT A	BOOLEAN	If A is <b>FALSE</b> , then <b>TRUE</b> is returned. If A is <b>NULL</b> , then <b>NULL</b> is returned. Otherwise, <b>FALSE</b> is returned.
! A	BOOLEAN	Same as NOT A.
A IN (val1, val2, ...)	BOOLEAN	If A is equal to any value in (val1, val2, ...), then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned.
A NOT IN (val1, val2, ...)	BOOLEAN	If A is not equal to any value in (val1, val2, ...), then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned.
EXISTS (subquery)	BOOLEAN	If the result of any subquery contains at least one line, then <b>TRUE</b> is returned. Otherwise, <b>FALSE</b> is returned.
NOT EXISTS (subquery)	BOOLEAN	If the subquery output does not contain any row, <b>TRUE</b> is returned; otherwise, <b>FALSE</b> is returned.