Data Lake Insight

HetuEngine SQL Syntax Reference

Issue 01

Date 2024-12-31

HUAWEI TECHNOLOGIES CO., LTD.

Security Declaration

Vulnerability

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

Contents

1 HetuEngine SQL Syntax

1.1 Before You Start

Notes

- To use the DLI HetuEngine function, which is currently in the whitelist, submit a request by choosing **Service Tickets** > **Create Service Ticket** in the upper right corner of the management console.
- Before using this function, you need to create a HetuEngine SQL queue. For details, see **Creating an Elastic Resource Pool and Creating Queues Within It**.
- HetuEngine SQL must be used together with LakeFormation. For details, see **Connecting DLI to LakeFormation**.

Introduction to HetuEngine

HetuEngine is a high-performance interactive SQL analysis and data virtualization engine launched by Huawei, seamlessly integrating with the big data ecosystem to achieve second-level interactive queries on massive datasets.

DLI + HetuEngine can quickly handle query requests on large-scale datasets, swiftly and efficiently extracting information from big data, significantly simplifying data management and analysis processes, and enhancing indexing and query performance in a big data environment.

- Responses to terabytes of data within mere seconds:

  Through automatic optimization of resource and load balancing, HetuEngine can respond to TB-level data within seconds, significantly improving the efficiency of data queries.
- Serverless resources ready to use:

  The serverless service model requires no attention to underlying configurations, software updates, or failure issues, making resources easy to maintain and expand.
- Multiple resource types to meet various service requirements:

  Shared resource pool: pay-per-use billing, providing more cost-effective compute resources.

  Dedicated resource pool: offers a dedicated resource pool to meet high-performance resource requirements.

1.2 Data Type

1.2.1 Data Types

Currently, HetuEngine supports the following data types when creating tables: TINYINT, SMALLINT, BIGINT, INT, BOOLEAN, REAL, DECIMAL, DOUBLE, VARCHAR, STRING, BINARY, VARBINARY, TIMESTAMP, DATE, CHAR, ARRAY, ROW, MAP, and STRUCT. Other types are supported during data query and calculation.

Generally, most non-composite data types can be entered by literals and character strings. In the example, a string in the JSON format is added.

```
select json '{"name": "aa", "sex": "man"}';
            _col0
---------------------------------
 {"name":"aa","sex":"man"}
(1 row)
```

1.2.2 Boolean

The valid text values for "true" are **TRUE**, **t**, **true**, and **1**.

The valid text values for the "false" value are **FALSE**, **f**, **false**, and **0**.

TRUE and **FALSE** are standard usages (SQL-compatible).

Example:

```
select BOOLEAN '0';
 _col0
-------
 false
(1 row)

select BOOLEAN 'TRUE';
 _col0
-------
 true
(1 row)

select BOOLEAN 't';
 _col0
```

```
-------
 true
(1 row)
```

1.2.3 Integer

Table 1-1 Integer

Name	Description	Storage Space	Value Range	Literal
TINYINT	Tiny integer	8 bits	-128~127	TINYINT
SMALLINT	Small integer	16 bits	-32,768 to +32,767	SMALLINT
INTEGER	Integer	32 bits	-2,147,483,648 to +2,147,483,647	INT
BIGINT	Big integer	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	BIGINT

Example:

```
--Create a table containing TINYINT data:
CREATE TABLE int_type_t1  (IT_COL1 TINYINT) ;
--Insert data of the TINYINT type:
insert into int_type_t1  values (TINYINT'10');
--View data:
SELECT * FROM int_type_t1;
 it_col1
---------
 10
(1 row)
--Drop a table:
DROP TABLE int_type_t1;
```

1.2.4 Fixed Precision

Name	Description	Storage Space	Value Range	Literal
DECIMAL	Decimal number with fixed precision. The precision can be as high as 38 bits, but for optimal performance, a precision of less than 18 bits is recommended. Decimal has two input parameters: ● **precision**: total number of digits. The default value is **38**. ● **scale**: number of decimal places. The default value is **0**. **NOTE** If **scale** is **0** and **precision** is **38**, the maximum precision is 19 bits.	64 bits	$-10^{38}+1$ ~ $10^{38}-1$	DECIMAL
NUMERIC	Same as DECIMAL	128 characters	$-10^{38}+1$ ~ $10^{38}-1$	NUMERIC

Table 1-2 Examples of literals

Literal	Data Type
DECIMAL '0'	DECIMAL(1)
DECIMAL '12345'	DECIMAL(5)
DECIMAL '0000012345.1234500000'	DECIMAL(20, 10)

```
--Create a table containing DECIMAL data.
CREATE TABLE decimal_t1 (dec_col1 DECIMAL(10,3)) ;

– Insert data of the DECIMAL type.
insert into decimal_t1 values (DECIMAL '5.325');

--View data.
SELECT * FROM decimal_t1;
 dec_col1
---------
 5.325
(1 row)

--Drop a table.
```

```
DROP TABLE decimal_t1;

--Create a NUMERIC type table.
CREATE TABLE tb_numberic_hetu(col1 NUMERIC(9,7));

--Insert data.
INSERT INTO tb_numberic_hetu values(9.12);


--View data.
SELECT * FROM tb_numberic_hetu;
col1
------------
  9.1200000
(1 row)
```

1.2.5 Float

Name	Description	Storage Space	Value Range	Literal
REAL	Real number	32 bits	1.40129846432481707e-45 to 3.40282346638528860e+38, positive or negative	REAL
DOUBLE	Double-precision floating point number with 15 to 17 valid digits, depending on the application scenario. The number of valid digits does not depend on the decimal point.	64 bits	4.94065645841246544e-324 to 1.79769313486231570e+308, positive or negative	DOUBLE
FLOAT	Single-precision floating point number with 6 to 9 valid digits, depending on the application scenario. The number of valid digits does not depend on the decimal point.	32 bits	1.40129846432481707e-45 to 3.40282346638528860e+38, positive or negative	FLOAT

Usage:

- When a distributed query uses high-performance hardware instructions to perform single-precision or double-precision computing, the computing result may be slightly different because the execution sequence is different each time when an aggregate function, such as **SUM()** or **AVG()**, is invoked. Especially when the data volume is large (tens of millions or even billions of records), the computing result may be slightly different. In this case, you are advised to use the DECIMAL data type.

- An alias can be used to specify the data type.

  Example:

```
--Create a table that contains float data.
CREATE TABLE float_t1 (float_col1 FLOAT) ;
--Insert data of the float type.
insert into float_t1 values (float '3.50282346638528862e+38');
--View data.
SELECT * FROM float_t1;
float_col1
------------
Infinity
(1 row)
--Drop the table.
DROP TABLE float_t1;
```

- When the decimal part is 0, you can use **cast()** to convert the decimal part to an integer of the corresponding range. The decimal part is rounded off.

  Example:

```
select CAST(1000.0001 as INT);
_col0
-------
1000
(1 row)
select CAST(122.5001 as TINYINT);
_col0
-------
123
(1 row)
```

- When an exponential expression is used, the string can be converted to the corresponding type.

  Example:

```
select CAST(152e-3 as double);
_col0
-------
0.152
(1 row)
```

1.2.6 Character

Name	Description
VARCHAR(n)	Variable-length character string. *n* indicates the byte length.
CHAR(n)	Fixed-length character string. If the length is insufficient, spaces are added. *n* indicates the byte length. If the precision *n* is not specified, the default value **1** is used.
VARBINARY	Variable-length binary data. The value must be prefixed with **X**, for example, **X'65683F'**. Currently, a binary character string of a specified length is not supported.
JSON	The value can be **a JSON object**, **a JSON array**, **a JSON number**, **a JSON string**, **true**, **false** or **null**.
STRING	String compatible with impala. The bottom layer is varchar.

Name	Description
BINARY	Compatible with Hive BINARY. The underlying implementation is VARBINARY.

- In SQL expressions, simple character expressions and unicodes are supported. A unicode character string uses **U&** as the fixed prefix. An escape character must be added before Unicode that is represented by a 4-digit value.

```
-- Character expression
select 'hello,winter!';
      _col0
------------------
 hello,winter!
(1 row)
-- Unicode expression
select U&'Hello winter \2603 !';
      _col0
------------------
 Hello winter !
(1 row)
-- User-defined escape character
select U&'Hello winter #2603 !' UESCAPE '#';
      _col0
------------------
 Hello winter !
(1 row)
```

- VARBINARY and BINARY

```
-- Creating a VARBINARY or BINARY Table
 create table binary_tb(col1 BINARY);

--Insert data.
 INSERT INTO binary_tb values (X'63683F');

--Query data.
 select * from binary_tb ;  -- 63 68 3f
```

- When two CHARs with different numbers of spaces at the end are compared, the two CHARs are considered equal.

```
SELECT CAST('FO' AS CHAR(4)) = CAST('FO     ' AS CHAR(5));
 _col0
-------
 true
(1 row)
```

1.2.7 Time and Date Type

Remarks

The time and date are accurate to milliseconds.

Table 1-3 Time and Date Type

Name	Description	Storage Space
DATE	Date and time. Only the ISO 8601 format is supported, for example **2020-01-01**.	32 bits

Name	Description	Storage Space
TIME	Time (hour, minute, second, millisecond) without a time zone Example: TIME '01:02:03.456'	64 bits
TIME WITH TIMEZONE	Time with a time zone (hour, minute, second, millisecond). Time zones are expressed as the numeric UTC offset value. Example: TIME '01:02:03.456 -08:00'	96 bits
TIMESTAMP	Timestamp	64 bits
TIMESTAMP WITH TIMEZONE	Timestamp with time zone	64 bits
INTERVAL YEAR TO MONTH	Time interval literal year and month, in the format of SY-M. S: optional symbols (+/-) Y: years M: months	128 characters
INTERVAL DAY TO SECOND	The time interval literally indicates the day, hour, minute, and second, and is accurate to millisecond. The format is SD H:M:S.nnn. S: optional symbols (+/-) D: days M: minutes S: seconds nnn: milliseconds	128 characters

Example:

```
-- Query the date:
SELECT DATE '2020-07-08';
   _col0
------------
 2020-07-08
(1 row)

-- Query time:
SELECT TIME '23:10:15';
 _col0
--------------
 23:10:15
(1 row)

SELECT TIME '01:02:03.456 -08:00';
 _col0
--------------
 01:02:03.456-08:00
(1 row)
```

```
-- Time interval usage
SELECT TIMESTAMP '2015-10-18 23:00:15' + INTERVAL '3 12:15:4.111' DAY TO SECOND;
 _col0
-------------------------
 2015-10-22 11:15:19.111
(1 row)

SELECT TIMESTAMP '2015-10-18 23:00:15' + INTERVAL '3-1' YEAR TO MONTH;
          _col0
-------------------------
 2018-11-18 23:00:15
(1 row)

select INTERVAL '3' YEAR + INTERVAL '2' MONTH ;
 _col0
-------
 3-2
(1 row)

select INTERVAL '1' DAY+INTERVAL '2' HOUR +INTERVAL '3' MINUTE +INTERVAL '4' SECOND ;
     _col0
----------------
 1 02:03:04.000
(1 row)
```

1.2.8 Complex Type

ARRAY

Array

Example: **ARRAY[1, 2, 3]**.

```
-- Create an ARRAY type table.
create table array_tb(col1 ARRAY<STRING>);

-- Insert a record of the ARRAY type.
insert into array_tb values(ARRAY['HetuEngine','Hive','Mppdb']);

-- Query data.
select * from array_tb; --    [HetuEngine, Hive, Mppdb]
```

MAP

Data type of a key-value pair.

Example: **MAP(ARRAY['foo', 'bar'], ARRAY[1, 2])**.

```
-- Create a Map table.
create table map_tb(col1 MAP<STRING,INT>);

-- Insert a piece of data of the Map type.
insert into map_tb values(MAP(ARRAY['foo','bar'],ARRAY[1,2]));

-- Query data.
select * from map_tb; --   {bar=2, foo=1}
```

ROW

ROW fields can be any supported data type or a combination of different field data types.

```
-- Create a ROW table.
create table row_tb (id int,col1 row(a int,b varchar));
```

```
-- Insert data of the ROW type.
insert into row_tb values (1,ROW(1,'HetuEngine'));

-- Query data.
select * from row_tb;
 id |     col1
----|--------------z
  1 | {a=1, b=HetuEngine}

--Fields can be named. By default, a Row field is not named.
select  row(1,2e0),CAST(ROW(1, 2e0) AS ROW(x BIGINT, y DOUBLE));
           _col0           |    _col1
------------------------|--------------
 {1, 2.0} | {x=1, y=2.0}
(1 row)

--Named fields can be accessed using the domain operator ".".
select col1.b from row_tb; -- HetuEngine

--Both named and unnamed fields can be accessed through location indexes, which start from 1 and must be a constant.
select col1[1] from row_tb;   -- 1
```

IPADDRESS

IP address, which can be an IPv4 or IPv6 address. In the system, however, this type of address is a unified IPv6 address.

IPv4 is supported by mapping IPv4 addresses to the IPv6 address range (RFC 4291#section-2.5.5.2). When an IPv4 address is created, it is mapped to an IPv6 address. During formatting, if the data is IPv4, the data will be mapped to IPv4 again. Other addresses are formatted according to the format defined in RFC 5952.

Example:

```
select IPADDRESS '10.0.0.1', IPADDRESS '2001:db8::1';
  _col0   |    _col1
----------|-------------
 10.0.0.1 | 2001:db8::1
(1 row)
```

UUID

A standard universally unique identifier (UUID) is also called a globally unique identifier (GUID).

It complies with the format defined in RFC 4122.

Example:

```
select UUID '12151fd2-7586-11e9-8f9e-2a86e4085a59';
                _col0
--------------------------------------
 12151fd2-7586-11e9-8f9e-2a86e4085a59
(1 row)
```

QDIGEST

A quantile, also referred to as a quantile point, refers to that a probability distribution range of a random variable is divided into several equal numerical points. Commonly used quantile points include a median (that is, a 2-quantile), a 4-quartile, and a 100-percentile. A quantile digest is a set of quantiles. When the data being queried is in proximity to a specific quantile, that quantile can be used as an approximate value for the queried data. Its precision can be adjusted, but higher precision results in expensive space overhead.

STRUCT

The bottom layer is implemented using ROW. For details, see **ROW**.

Example:

```
-- Create a STRUCT table.
create table struct_tab (id int,col1 struct<col2: integer, col3: string>);

-- Insert data of the STRUCT type.
 insert into struct_tab VALUES(1, struct<2, 'HetuEngine'>);

-- Query data.
select * from struct_tab;
 id |        col1
----|----------------------
  1 | {col2=2, col3=HetuEngine}
```

1.3 DDL Syntax

1.3.1 CREATE SCHEMA

CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name

[COMMENT database_comment]

[LOCATION obs_path]

[WITH DBPROPERTIES (property_name=property_value,...)];

CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name

[WITH (property_name=property_value,...)]

Description

This statement is used to create an empty schema. A schema is a container for tables, views, and other database objects. If the optional parameter **IF NOT EXISTS** is specified and a schema with the same name already exists in the system, no error is reported.

Example

- Creating a schema named "web":
  ```
  CREATE SCHEMA web;
  ```
- Create a schema in a specified path. The path must be an OBS parallel bucket and cannot end with a slash (/). The following is an example of a valid path:
  ```
  CREATE SCHEMA test_schema_5 LOCATION 'obs://${bucket}/user/hive';
  ```
- Run the following command to create a schema named **sales** in the catalog named **Hive**:
  ```
  CREATE SCHEMA hive.sales;
  ```

- If the schema named **traffic** does not exist in the current catalog, run the following command to create a schema named **traffic**:
  ```
  CREATE SCHEMA IF NOT EXISTS traffic;
  ```
- Create a schema with attributes:
  ```
  CREATE DATABASE createtestwithlocation COMMENT 'Holds all values' LOCATION '/user/hive/
  warehouse/create_new' WITH dbproperties('name'='akku', 'id' ='9');

  --Use the describe schema|database statement to query the schema created:
  describe schema createtestwithlocation;
  ```

1.3.2 CREATE TABLE

Syntax

①

CREATE TABLE [IF NOT EXISTS]

[catalog_name.][db_name.]table_name (

{ column_name data_type [NOT NULL]

[COMMENT col_comment]

[WITH (property_name = expression [, ...])]

| LIKE existing_table_name

[{ INCLUDING | EXCLUDING } PROPERTIES]

}

[, ...]

)

[COMMENT table_comment]

[WITH (property_name = expression [, ...])]

②

CREATE [EXTERNAL] TABLE [IF NOT EXISTS]

[catalog_name.][db_name.]table_name (

{ column_name data_type [NOT NULL]

[COMMENT comment]

[WITH (property_name = expression [, ...])]

| LIKE existing_table_name

[{ INCLUDING | EXCLUDING } PROPERTIES]

}

[, ...]

)

[COMMENT 'table_comment']

```
[PARTITIONED BY(col_name data_type, ....)]

[CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name, col_name, ...)]
INTO num_buckets BUCKETS] ]

[ROW FORMAT row_format]

[STORED AS file_format]

[LOCATION 'obs_path']

[TBLPROPERTIES (orc_table_property = value [, ...] ) ]
```

③

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS]

[catalog_name.][db_name.]table_name (

{ column_name data_type [ NOT NULL ]

[ COMMENT comment ]

[ WITH ( property_name = expression [, ...] ) ]

| LIKE existing_table_name

[ { INCLUDING | EXCLUDING } PROPERTIES ]

}

[, ...]

)

[PARTITIONED BY(col_name data_type, ....)]

[SORT BY ([column [, column ...]])]

[COMMENT 'table_comment']

[ROW FORMAT row_format]

[STORED AS file_format]

[LOCATION 'obs_path']

[TBLPROPERTIES (orc_table_property = value [, ...] ) ]
```

Remarks

- During partitioned table creation, if **bucket_count** is set to **-1** and **buckets** is not set in the table creation statement, the default value **16** is used.
- By default, external tables are stored in *{lakeformation_catalog_url}***/** *{schema_name}***.db/** *{table_name}*. *{lakeformation_catalog_url}* indicates the location configured for the LakeFormation catalog, *{schema_name}* indicates the schema used for table creation, and *{table_name}* indicates the table name.
- Data insertion is not allowed for hosted tables with the **external** attribute set to **true**.

Description

Creates a new empty table with specified columns by using CREATE TABLE. Use the **CREATE TABLE AS** statement to create a table with data.

- When the optional parameter **IF NOT EXISTS** is used, no error is reported if the table already exists.
- The WITH clause can be used to set properties for a newly created table or a single column, such as the storage location of the table and whether the table is an external table.
- The LIKE clause is used to include all column definitions from an existing table in a new table. You can specify multiple LIKE clauses to allow columns to be copied from multiple tables. If **INCLUDING PROPERTIES** is specified, all table properties are copied to the new table. If the attribute name specified in the WITH clause is the same as the copied attribute name, the value in the WITH clause is used. By default, the EXCLUDING PROPERTIES attribute is used. You can specify the INCLUDING PROPERTIES attribute for only one table.
- **PARTITIONED BY** can be used to specify partition columns. **CLUSTERED BY** can be used to specify bucket columns. **SORT BY** and **SORTED BY** can be used to sort specified bucket columns. **BUCKETS** can be used to specify the number of buckets. **EXTERNAL** can be used to create foreign tables. **STORED AS** can be used to specify the file storage format. **LOCATION** can be used to specify the storage path on OBS.

Example

- Create a new table **orders** and use the WITH clause to specify the storage format, storage location, and whether the table is a foreign table.

  The **auto.purge** parameter can be used to specify whether to clear related data when data removal operations (such as DROP, DELETE, INSERT OVERWRITE, and TRUNCATE TABLE) are performed.
  - If it is set to **true**, metadata and data files are cleared.
  - If it is set to **false**, only metadata is cleared and data files are moved to the OBS recycle bin. The default value is **false**. You are advised not to change the value. Otherwise, deleted data cannot be restored.

```
CREATE TABLE orders (
orderkey bigint,
orderstatus varchar,
totalprice double,
orderdate date
)
WITH (format = 'ORC', location='obs://bucket/user',orc_compress='ZLIB',external=true,
"auto.purge"=false);

-- You can run the DESC FORMATTED statement to view details about table creation.
desc formatted  orders ;
                         Describe Formatted Table
------------------------------------------------------------------------------
 # col_name      data_type      comment
 orderkey      bigint
 orderstatus      varchar
 totalprice      double
 orderdate      date

 # Detailed Table Information
 Database:                  default
 Owner:                     admintest
 LastAccessTime:            0
 Location:                  obs://bucket/user
 Table Type:                EXTERNAL_TABLE

 # Table Parameters:
        EXTERNAL                TRUE
        auto.purge              false
        orc.compress.size       262144
        orc.compression.codec   ZLIB
        orc.row.index.stride    10000
        orc.stripe.size         67108864
        presto_query_id         20220812_084110_00050_srknk@default@HetuEngine
        presto_version          1.2.0-h0.cbu.mrs.320.r1-SNAPSHOT
        transient_lastDdlTime   1660293670

 # Storage Information
 SerDe Library:             org.apache.hadoop.hive.ql.io.orc.OrcSerde
 InputFormat:               org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
 OutputFormat:              org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat
 Compressed:                No
 Num Buckets:               -1
 Bucket Columns:            []
 Sort Columns:              []
 Storage Desc Params:
        serialization.format    1
(1 row)
```

- Create a table with the specified row format.

```
-- When creating a table, use commas (,) to separate fields in each record in the data file.
CREATE TABLE student(
id string,birthday string,
grade int,
memo string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

-- When creating a table, specify the field delimiter and newline character as '\t' and '\n', respectively.
CREATE TABLE test(
id int,
name string ,
tel string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

- If the **orders** table does not exist, create the **orders** table and add table comments and column comments:

```
CREATE TABLE IF NOT EXISTS orders (
orderkey bigint,
orderstatus varchar,
totalprice double COMMENT 'Price in cents.',
orderdate date
)
COMMENT 'A table to keep track of orders.';
insert into orders values
(202011181113,'online',9527,date '2020-11-11'),
(202011181114,'online',666,date '2020-11-11'),
(202011181115,'online',443,date '2020-11-11'),
(202011181115,'offline',2896,date '2020-11-11');
```

- Create the **bigger_orders** table using the column definition of the **orders** table:

```
CREATE TABLE bigger_orders (
another_orderkey bigint,
LIKE orders,
another_orderdate date
);

SHOW CREATE TABLE bigger_orders ;
                              Create Table
```

```
----------------------------------------------------------------------
 CREATE TABLE hive.default.bigger_orders (
    another_orderkey bigint,
    orderkey bigint,
    orderstatus varchar,
    totalprice double,
    ordersdate date,
    another_orderdate date
 )
 WITH (
    external = false,
    format = 'ORC',
    location = 'obs://bucket/user/hive/warehouse/bigger_orders',
    orc_compress = 'GZIP',
    orc_compress_size = 262144,
    orc_row_index_stride = 10000,
    orc_stripe_size = 67108864
 )
(1 row)
```

- **①Example of creating a table:**
```
CREATE EXTERNAL TABLE hetu_test (orderkey bigint, orderstatus varchar, totalprice double, orderdate date) PARTITIONED BY(ds int) SORT BY (orderkey, orderstatus) COMMENT 'test' STORED AS ORC LOCATION '/user' TBLPROPERTIES (orc_compress = 'SNAPPY', orc_compress_size = 6710422, orc_bloom_filter_columns = 'orderstatus,totalprice');
```
- **②Example of creating a table:**
```
CREATE EXTERNAL TABLE hetu_test1 (orderkey bigint, orderstatus varchar, totalprice double, orderdate date) COMMENT 'test' PARTITIONED BY(ds int) CLUSTERED BY (orderkey, orderstatus) SORTED BY (orderkey, orderstatus) INTO 16 BUCKETS STORED AS ORC LOCATION '/user' TBLPROPERTIES (orc_compress = 'SNAPPY', orc_compress_size = 6710422, orc_bloom_filter_columns = 'orderstatus,totalprice');
```
- **③Example of creating a table:**
```
CREATE TABLE hetu_test2 (orderkey bigint, orderstatus varchar, totalprice double, orderdate date, ds int) COMMENT 'This table is in Hetu syntax' WITH (partitioned_by = ARRAY['ds'], bucketed_by = ARRAY['orderkey', 'orderstatus'], sorted_by = ARRAY['orderkey', 'orderstatus'], bucket_count = 16, orc_compress = 'SNAPPY', orc_compress_size = 6710422, orc_bloom_filter_columns = ARRAY['orderstatus', 'totalprice'], external = true, format = 'orc', location = '/user');
```
- Run the following statements to view the table:
```
show create table hetu_test1;
                          Create Table
-------------------------------------------------------------------
 CREATE TABLE hive.default.hetu_test1 (
    orderkey bigint,
    orderstatus varchar,
    totalprice double,
    orderdate date,
    ds integer
 )
 COMMENT 'test'
 WITH (
    bucket_count = 16,
    bucketed_by = ARRAY['orderkey','orderstatus'],
    bucketing_version = 1,
    external_location = 'obs://bucket/user',
    format = 'ORC',
    orc_bloom_filter_columns = ARRAY['orderstatus','totalprice'],
    orc_bloom_filter_fpp = 5E-2,
    orc_compress = 'SNAPPY',
    orc_compress_size = 6710422,
    orc_row_index_stride = 10000,
    orc_stripe_size = 67108864,
    partitioned_by = ARRAY['ds'],
    sorted_by = ARRAY['orderkey','orderstatus']
 )
(1 row)
```

Create a partitioned table.

```
--Create a schema.
CREATE SCHEMA hive.web WITH (location = 'obs://bucket/user');
--Create a partitioned table.
CREATE TABLE hive.web.page_views (
  view_time timestamp,
  user_id bigint,
  page_url varchar,
  ds date,
  country varchar
)
WITH (
  format = 'ORC',
  partitioned_by = ARRAY['ds', 'country'],
  bucketed_by = ARRAY['user_id'],
  bucket_count = 50
);

--View the partition.
SELECT * FROM hive.web."page_views$partitions";
     ds     | country
------------|---------
 2020-07-18 | US
 2020-07-17 | US
--Insert data.
insert into hive.web.page_views values(timestamp '2020-07-17 23:00:15',bigint '15141','www.local.com',date
'2020-07-17','US' );
insert into hive.web.page_views values(timestamp '2020-07-18 23:00:15',bigint '18148','www.local.com',date
'2020-07-18','US' );

--Query data.
 select * from hive.web.page_views;
        view_time        | user_id |   page_url    |     ds     | country
-------------------------|---------|---------------|------------|---------
 2020-07-17 23:00:15.000 |   15141 | www.local.com | 2020-07-17 | US
2020-07-18 23:00:15.000 |   18148 | www.local.com | 2020-07-18 | US
```

1.3.3 CREATE TABLE AS

Syntax

CREATE [EXTERNAL]① TABLE [IF NOT EXISTS] [catalog_name.][db_name.]table_name [(column_alias, ...)]

[[PARTITIONED BY ①(col_name, ...)] [SORT BY① ([*column* [, *column* ...]])]]①

[COMMENT 'table_comment']

[WITH (property_name = expression [, ...])]②

[[STORED AS file_format]①

[LOCATION 'obs_path']①

[TBLPROPERTIES (orc_table_property = value [, ...])]]①

AS query

[WITH [NO] DATA]②

Remarks

The syntax of ① and ② cannot be used together.

When the **avro_schema_url** attribute is used:

- **CREATE TABLE AS** is not supported.
- When **CREATE TABLE** is used, **partitioned_by** and **bucketed_by** are not supported.
- Columns cannot be modified by using **alter table**.

Description

It is used to create a table that contains the **SELECT** query result.

Use the **CREATE TABLE** statement to create an empty table.

When the **IF NOT EXISTS** clause is used, no error is reported if the table already exists.

The **WITH** clause can be used to set the properties of a newly created table, such as the storage location of the table and whether the table is an external table.

Example

- Run the following statement to create the **orders_column_aliased** table based on the query result of a specified column:
  ```
  CREATE TABLE orders_column_aliased (order_date, total_price)
  AS
  SELECT orderdate, totalprice FROM orders;
  ```
- Create the **orders_by_data** table based on the summary result of the **orders** table.
  ```
  CREATE TABLE orders_by_date
  COMMENT 'Summary of orders by date'
  WITH (format = 'ORC')
  AS
  SELECT orderdate, sum(totalprice) AS price
  FROM orders
  GROUP BY orderdate;
  ```
- If the **orders_by_date** table does not exist, create the **orders_by_date** table:
  ```
  CREATE TABLE IF NOT EXISTS orders_by_date AS
  SELECT orderdate, sum(totalprice) AS price
  FROM orders
  GROUP BY orderdate;
  ```
- Create the **empty_orders table** using the schema that is the same as that of the **orders** table but does not contain data:
  ```
  CREATE TABLE empty_orders AS
  SELECT *
  FROM orders
  WITH NO DATA;
  ```
- Use **VALUES** to create a table. For details, see **VALUES**.
- Partitioned table example:
  ```
  CREATE EXTERNAL TABLE hetu_copy(corderkey, corderstatus, ctotalprice, corderdate, cds)
   PARTITIONED BY(cds)
   SORT BY (corderkey, corderstatus)
   COMMENT 'test'
   STORED AS orc
   LOCATION 'obs://{bucket}/user/hetuserver/tmp'
   TBLPROPERTIES (orc_bloom_filter_fpp = 0.3, orc_compress = 'SNAPPY', orc_compress_size = 6710422,
  orc_bloom_filter_columns = 'corderstatus,ctotalprice')
   as select * from hetu_test;

   CREATE TABLE hetu_copy1(corderkey, corderstatus, ctotalprice, corderdate, cds)
   WITH (partitioned_by = ARRAY['cds'], bucketed_by = ARRAY['corderkey', 'corderstatus'],
  ```

```
sorted_by = ARRAY['corderkey', 'corderstatus'],
bucket_count = 16,
orc_compress = 'SNAPPY',
orc_compress_size = 6710422,
orc_bloom_filter_columns = ARRAY['corderstatus', 'ctotalprice'],
external = true,
format = 'orc',
location = 'obs://{bucket}/user/hetuserver/tmp')
 as select * from hetu_test;
```

1.3.4 CREATE TABLE LIKE

Syntax

CREATE TABLE [IF NOT EXISTS] table_name ({coulumn_name data_type [COMMENT comment] [WITH (property_name = expression [, ...])] | LIKE existing_table_name [{INCLUDING| EXCLUDING} PROPERTIES]}) [, ...] [COMMENT table_comment] [WITH (property_name = expression [, ...])]

Description

The **LIKE** clause allows you to include all column definitions of an existing table in a new table. You can use multiple **LIKE** statements to copy the columns of multiple tables.

If the **INCLUDING PROPERTIES** option is used, all attributes of the table are copied to the new table. This option takes effect for only one table.

You can use the **WITH** clause to modify the name of the attribute copied from the table.

The **EXCLUDING PROPERTIES** attribute is used by default.

For a table with partitions, if the **LIKE** clause is enclosed in parentheses, the copied column definition does not contain information about the partition key.

Example

- Create basic tables order01 and order02.
  ```
  CREATE TABLE order01(id int,name string,tel string) ROW FORMAT DELIMITED FIELDS TERMINATED
  BY '\t' LINES TERMINATED BY '\n'STORED AS TEXTFILE;
  CREATE TABLE order02(sku int, sku_name string, sku_describe string);
  ```
- Create the orders_like01 table that contains the columns defined by the order01 table and the table properties.
  ```
  CREATE TABLE orders_like01 like order01 INCLUDING PROPERTIES;
  ```
- Create the orders_like02 table that contains the columns defined by the order02 table, and set the storage format of the table to **TEXTFILE**.
  ```
  CREATE TABLE orders_like02 like order02 STORED AS TEXTFILE;
  ```
- Create the orders_like03 table that contains the columns defined by the order01 table and the table properties, the columns defined by the order02 table, and additional columns c1 and c2.
  ```
  CREATE TABLE orders_like03 (c1 int,c2 float,LIKE order01 INCLUDING PROPERTIES,LIKE order02);
  ```
- Create the orders_like04 and orders_like05 tables. Both tables contain the definition of the same order_partition. The orders_like04 table does not contain but the orders_like05 table contains the partition key information.
  ```
  CREATE TABLE order_partition(id int,name string,tel string) PARTITIONED BY (sku int);
  CREATE TABLE orders_like04 (like order_partition);
  ```

```
CREATE TABLE orders_like05 like order_partition;
DESC orders_like04;
 Column |  Type   | Extra | Comment 
--------|---------|-------|---------
 id     | integer |       |         
 name   | varchar |       |         
 tel    | varchar |       |         
 sku    | integer |       |         
(4 rows)

DESC orders_like05;

 Column |  Type   |     Extra     | Comment 
--------|---------|---------------|---------
 id     | integer |               |         
 name   | varchar |               |         
 tel    | varchar |               |         
 sku    | integer | partition key |         
(4 rows)
```

1.3.5 CREATE VIEW

Syntax

CREATE [OR REPLACE] VIEW view_name [(column_name [COMMENT 'column_comment'][, ...])] [COMMENT 'view_comment'] [TBLPROPERTIES (property_name = property_value)] AS query

Remarks

Only Catalog of the Hive data source supports the column description of the view.

For the views created in HetuEngine, the definition of the views is stored in the data source in encoding mode. You can query the view in the data source but cannot perform operations on the view.

Views are read-only. You cannot perform the LOAD or INSERT operation on views.

A view can contain the **ORDER BY** and **LIMIT** clauses. If a query statement associated with the view also contains these clauses, the **ORDER BY** and **LIMIT** clauses in the query statement are calculated based on the result of the view.

Description

It is used to create a view using the SELECT query result. A view is a logical table that can be referenced by future queries. A view has no data. The query corresponding to the view is executed each time the view is referenced by another query.

If the view already exists, the optional ORREPLACE clause causes the view to be replaced without reporting an error.

Example

- To create a view named **test** in the **orders** table:
```
CREATE VIEW test (oderkey comment 'orderId',orderstatus comment 'status',half comment 'half') AS
SELECT orderkey, orderstatus, totalprice / 2 AS half FROM orders;
```
- Create the **orders_by_date** view based on the summary result of the **orders** table.

```
CREATE VIEW orders_by_date AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate;
```

- Create a view to replace the existing view:
  ```
  CREATE OR REPLACE VIEW test AS
  SELECT orderkey, orderstatus, totalprice / 4 AS quarter
  FROM orders
  ```
- Create a view and set table attributes:
  ```
  create or replace view  view1 comment 'the first view' TBLPROPERTIES('format'='orc') as select * from fruit;
  ```

Precautions

When **alter** is used to modify the table on which the created view depends, you need to create the view again. Otherwise, an error will be reported when you query the view again.

1.3.6 ALTER TABLE

Syntax

NOTE

name, **new_name**, **column_name**, **new_column_name**, and **table_name_*** are user-defined parameters.

1. The following statement is used to rename a table.

   ALTER TABLE name RENAME TO new_name

2. Change the column name of the table and add comments (optional) and properties (optional) to the column. For details about supported column properties, see **Description**.

   ALTER TABLE name ADD COLUMN column_name data_type [COMMENT comment] [WITH (property_name = expression [, ...])]

3. The following statement is used to delete the **column_name** column from the table.

   ALTER TABLE name DROP COLUMN column_name

NOTICE

- Partition or bucket columns cannot be deleted.
- DROP COLUMN does not support tables stored in RCTEXT, RCBINARY, or RCFILE format. Connector accesses columns in different file formats in different modes. The query may fail after the **DROP COLUMN** operation is performed. For example:
  - For a non-partitioned table stored in ORC format, if the query fails after the **DROP COLUMN** operation is performed, run the following command to set the Session property:

    set session hive.orc_use_column_names=true;
  - For a non-partitioned table stored in Parquet format, if the query fails after the **DROP COLUMN** operation is performed, run the following command to set the Session property:

    set session hive.parquet_use_column_names=true;
  - For partitioned or transaction tables in ORC or Parquet format, session properties cannot be configured to ensure query success after the **DROP COLUMN** operation is performed.

4. The following statement is used to rename the **column_name** column to **new_column_name**.

   ALTER TABLE name RENAME COLUMN column_name TO new_column_name

   NOTICE

   Partition or bucket columns cannot be renamed.

5. The following statement is used to add partitions to a partitioned table.

   ALTER TABLE name ADD [IF NOT EXISTS] PARTITION partition_spec [LOCATION 'location'][PARTITION partition_spec [LOCATION 'location'], ...];
6. The following statement is used to delete a partition from a partitioned table. This operation deletes data and metadata from the partition. If the directory for storing partition is specified when **ADD PARTITION** is run, the folder where the partition is located and data will not be deleted after **DROP PARTITION** is run, regardless of whether the table is an internal table or external table. If no partition storage path is specified when running **ADD PARTITION**, the partition directory will be deleted from OBS, and the data will be moved to the **.Trash/Current** folder.

   ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec[, PARTITION partition_spec, ...];
7. The following statement is used to rename a partition.

   ALTER TABLE table_name PARTITION(partition_key = partition_value1) rename to partition(partition_key = partition_value2)
8. The following statements are used to add or modify table properties.

ALTER TABLE table_name SET TBLPROPERTIES (property_name = property_value[, property_name = property_value, ...]);

NOTE

TBLPROPERTIES allows you to add or modify table attributes supported by a connector in key-value pair mode (attribute names and attributes must be strings enclosed in single or double quotation marks). The following uses the Hive connector as an example:

- TBLPROPERTIES ("transactional"="true"). The value can be **true** or **false**.
- TBLPROPERTIES ("auto.purge"="true"). The value can be **true** or **false**.

9. The following statement is used to modify column properties.

   ALTER TABLE table_name [PARTITION partition_spec] CHANGE [COLUMN] col_old_name col_new_name column_type [COMMENT col_comment] [FIRST|AFTER column_name] [CASCADE|RESTRICT]

NOTICE

- For an existing table, modify the column name, data type, comment, location (**[FIRST|AFTER column_name]** is used to specify the location of the column after modification), or any combination of the preceding items. If a partition clause is included in the syntax, the metadata of the corresponding partition also changes. In **CASCADE** mode, the syntax will take effect on the metadata of the table and table partition. In the default **RESTRICT** mode, the modification to a column takes effect only on the metadata of the table.
- The column modification statement can modify only the metadata of a table or partition, but cannot modify the data itself. Ensure that the actual data layout of the table or partition complies with the metadata definition.
- The partition column or bucket column of a table and Optimized Row Columnar (ORC) tables cannot be modified.

10. The following statement is used to change the storage location of the table or partition.

    ALTER TABLE table_name [PARTITION partition_spec] SET LOCATION location;

NOTE

- You can run the **ALTER TABLE [PARTITION] SET** statement to set the table or partition location.
- After the **SET LOCATION** statement is run, table or partition data may not be displayed.
- When a table or partition directory is created, **SET LOCATION** uses the specified directory instead of the default directory created on Hive during the creation of the table or partition.
- This statement does not affect the original data in the table or partition, or modify the original table or partition directory. New data is saved to the new directory.

Remarks

- The **ALTER TABLE table_name ADD | DROP col_name** statement is available only for non-partitioned tables in ORC or PARQUET format.

Example

- To change the table name from **users** to **people**:
  ALTER TABLE *users* **RENAME TO** *people***;**
- To add the **zip** column to the **users** table:
  ALTER TABLE *users* **ADD COLUMN** *zip* **varchar;**
- To delete the **zip** column from the **users** table:
  ALTER TABLE *users* **DROP COLUMN** *zip***;**
- To change the column name **id** in the **users** table to **user_id**:
  ALTER TABLE *users* **RENAME COLUMN id TO** *user_id***;**
- To modify a partition:

```
--Create two partitioned tables.
CREATE TABLE IF NOT EXISTS hetu_int_table5 (eid int, name String, salary String, destination String,
dept String, yoj int) COMMENT 'Employee Names' partitioned by (dt timestamp,country String, year
int, bonus decimal(10,3)) STORED AS TEXTFILE;

CREATE TABLE IF NOT EXISTS hetu_int_table6 (eid int, name String, salary String, destination String,
dept String, yoj int) COMMENT 'Employee Names' partitioned by (dt timestamp,country String, year
int, bonus decimal(10,3)) STORED AS TEXTFILE;

--Add partitions.
ALTER TABLE hetu_int_table5 ADD IF NOT EXISTS PARTITION (dt='2008-08-08 10:20:30.0',
country='IN', year=2001, bonus=500.23) PARTITION (dt='2008-08-09 10:20:30.0', country='IN',
year=2001, bonus=100.50) ;

--View the partition.
show partitions hetu_int_table5;
           dt            | country | year |  bonus
-------------------------|---------|------|---------
 2008-08-09 10:20:30.000 | IN      | 2001 | 100.500
 2008-08-08 10:20:30.000 | IN      | 2001 | 500.230
(2 rows)



--Rename a partition.
CREATE TABLE IF NOT EXISTS hetu_rename_table ( eid int, name String, salary String, destination
String, dept String, yoj int)
COMMENT 'Employee details'
partitioned by (year int)
STORED AS TEXTFILE;

ALTER TABLE hetu_rename_table ADD IF NOT EXISTS PARTITION (year=2001);

SHOW PARTITIONS hetu_rename_table;
year
------
 2001
(1 row)

ALTER TABLE hetu_rename_table PARTITION (year=2001) rename to partition (year=2020);

SHOW PARTITIONS hetu_rename_table;
year
------
 2020
(1 row)

--Modify a partitioned table.
create table altercolumn4(a integer, b string) partitioned by (c integer);


insert into altercolumn4 values (100, 'Daya', 500);
```

```
alter table altercolumn4 partition (c=500) change column b empname string comment 'changed
column name to empname' first;

--Change the storage location of a partitioned table. Specifically, you need to first create a directory
on OBS. After executing the statement, you will not be able to retrieve the previously inserted data.
alter table altercolumn4 partition (c=500) set Location 'obs://bucket/user/hive/warehouse/c500';


--Change the name of column b to name and the data type from integer to string.
create table altercolumn1(a integer, b integer) stored as textfile;

alter table altercolumn1 change column b name string;


--View the properties of altercolumn1.
describe formatted altercolumn1;
                              Describe Formatted Table
------------------------------------------------------------------------------------------
 # col_name      data_type      comment
 a      integer
 name      varchar

 # Detailed Table Information
 Database:                  default
 Owner:                     admintest
 LastAccessTime:            0
 Location:                  obs://bucket/user/hive/warehouse/altercolumn1
 Table Type:                MANAGED_TABLE

 # Table Parameters:
        STATS_GENERATED_VIA_STATS_TASK  workaround for potential lack of HIVE-12730
        numFiles                0
        numRows                 0
        orc.compress.size       262144
        orc.compression.codec   GZIP
        orc.row.index.stride    10000
        orc.stripe.size         67108864
        presto_query_id         20210325_025238_00034_f63xj@default@HetuEngine
        presto_version
        rawDataSize             0
        totalSize               0
        transient_lastDdlTime   1616640758

 # Storage Information
 SerDe Library:             org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
 InputFormat:               org.apache.hadoop.mapred.TextInputFormat
 OutputFormat:              org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
 Compressed:                No
 Num Buckets:               25
 Bucket Columns:            [a, name]
 Sort Columns:              [SortingColumn{columnName=name, order=ASCENDING}]
 Storage Desc Params:
        serialization.format    1
(1 row)

Query 20210325_090522_00091_f63xj@default@HetuEngine, FINISHED, 1 node
Splits: 1 total, 1 done (100.00%)
0:00 [0 rows, 0B] [0 rows/s, 0B/s]
```

1.3.7 ALTER VIEW

Syntax

- ALTER VIEW view_name AS select_statement;
- ALTER VIEW view_name SET TBLPROPERTIES table_properties;

Description

ALTER VIEW view_name AS select_statement; is used to change the definition of an existing view. Its syntax effect is similar to that of **CREATE OR REPLACE VIEW**.

The format of **table_properties** in **ALTER VIEW view_name SET TBLPROPERTIES table_properties;** is *(property_name = property_value, property_name = property_value, ...)*.

A view can contain the **Limit** and **ORDER BY** clauses. If the query statement of the associated view also contains these clauses, the final execution result is obtained from the calculation based on the clauses of the view. For example, if view V is specified to return five pieces of data and the associated query result is **select * from V limit 10**, only five pieces of data are returned.

Remarks

The preceding two syntax statements cannot be used together.

When a view contains partitions, the definition cannot be changed by using this syntax.

Example

```
CREATE OR REPLACE VIEW tv_view as SELECT id,name from (values (1, 'HetuEngine')) as x(id,name);

SELECT * FROM tv_view;
 id | name
----|------
  1 | HetuEngine
(1 row)

ALTER VIEW tv_view as SELECT id, brand FROM (VALUES (1, 'brand_1', 100), (2, 'brand_2', 300) ) AS x (id,
brand, price);

SELECT * FROM tv_view;
 id |  brand
----|---------
  1 | brand_1
  2 | brand_2
(2 rows)

ALTER VIEW tv_view SET TBLPROPERTIES ('comment' = 'This is a new comment');

show tblproperties tv_view;
                         SHOW TBLPROPERTIES
---------------------------------------------------------------------
        comment                 'This is a new comment'
        presto_query_id          '20210325_034712_00040_f63xj@default@HetuEngine'
        presto_version
        presto_view             'true'
        transient_lastDdlTime
'1616644032'
(1 row)
```

1.3.8 ALTER SCHEMA

Syntax

ALTER (DATABASE|SCHEMA) schema_name SET LOCATION obs_location

ALTER (DATABASE|SCHEMA) database_name SET OWNER USER username

ALTER (DATABASE|SCHEMA) database_name SET DBPROPERTIES (property_name=property_value, ...);

Description

This statement does not move the current content of SCHEMA to the new path or modify the table or partition associated with the specified schema. It only modifies the upper-level directory of the table that is newly added to the database.

Example

```
Create schema foo;
-- Change the schema storage path.
ALTER SCHEMA foo SET LOCATION 'obs://bucket/newlocation';
-- Change the owner of the schema.
ALTER SCHEMA foo SET OWNER user admin;
```

1.3.9 DROP SCHEMA

Syntax

DROP (DATABASE|SCHEMA) [IF EXISTS] databasename [RESTRICT|CASCADE]

Description

Delete the specified database from the catalog. If the database contains tables, you must delete these tables before running **DROP DATABASE**, or use the **CASCADE** mode.

DATABASE and **SCHEMA** are conceptually equivalent and interchangeable.

[IF EXISTS]

If the target database does not exist, an error message is displayed. However, if the **IF EXISTS** clause is used, no error message is displayed.

[RESTRICT|CASCADE]

The optional parameter **RESTRICT|CASCADE** is used to specify the deletion mode. The default mode is **RESTRICT**. In this mode, the database can be deleted only when it is empty without any table. In **CASCADE** mode, tables in the database are deleted first, and then the database. Exercise caution when using this mode.

Example

- To drop the schema **web**.
  ```
  DROP SCHEMA web;
  ```
- If the schema **sales** exists, drop the schema:
  ```
  DROP SCHEMA IF EXISTS sales;
  ```
- Delete **schema test_drop** in cascading mode. The **tb_web** table that exists in **schema test_drop** is deleted before **schema test_drop** is deleted.
  ```
  CREATE SCHEMA test_drop;

  CREATE TABLE tb_web(col1 int);

  DROP DATABASE test_drop CASCADE;
  ```

1.3.10 DROP TABLE

Syntax

DROP TABLE [IF EXISTS] table_name

Description

This statement is used to drop an existing table. If the optional parameter IF EXISTS is specified and the table to be deleted does not exist, no error is reported. The deleted data rows will be moved to the recycle bin on OBS.

Example

```
create table testfordrop(name varchar);
drop table if exists testfordrop;
```

1.3.11 DROP VIEW

Syntax

DROP VIEW [IF EXISTS] view_name

Description

This statement is used to delete an existing view. If the optional parameter IF EXISTS is specified and the view to be deleted does not exist, no error is reported.

Example

- Creating a View
  ```
  create view orders_by_date as select * from orders;
  ```
- Delete the **orders_by_date** view. If the view does not exist, an error is reported.
  ```
  DROP VIEW orders_by_date;
  ```
- Delete the **orders_by_date** view. Use the **IF EXISTS** parameter. If the view exists, it will be deleted. If the view does not exist, no error will be reported.
  ```
  DROP VIEW IF EXISTS orders_by_date;
  ```

1.3.12 TRUNCATE TABLE

Syntax

TRUNCATE TABLE table_name

Description

This statement is used to remove all rows from a table or partition. When the table property **auto.purge** is set to the default value **false**, the deleted data rows will be saved in the file system's recycle bin. However, if **auto.purge** is set to **true**, the data rows will be directly deleted.

Remarks

The target table must be a control table, which means table propery **external** is set to **false**. Otherwise, an error will be reported during statement execution.

Example

```
-- Delete a native or control table
Create table simple(id int, name string);

Insert into simple values(1,'abc'),(2,'def');

select * from simple;
 id | name 
----|------
  1 | abc  
  2 | def  
(2 rows)

Truncate table simple;

select * from simple;
 id | name 
----|------
(0 rows)
```

1.3.13 COMMENT

Syntax

COMMENT ON TABLE name IS 'comments'

Description

This statement is used to set the comment information of a table. You can delete the comment by setting the comment information to **NULL**.

Example

Change the comment of the **users** table to "master table". You can run the **show create table tablename** statement to view the comment.

```
COMMENT ON TABLE users IS 'master table';
```

1.3.14 VALUES

Syntax

VALUES row [, ...]

where row is a single expression or

(column_expression [, ...])

Description

VALUES is used to query any place that can be used (for example, the FROM clause of SELECT and INSERT). VALUES is used to create an anonymous table without column names, but tables and columns can be named using AS clauses with column aliases.

Example

- To return a table with one column and three rows:
  ```
  VALUES 1, 2, 3
  ```
- To return a table with two columns and three rows:
  ```
  VALUES
  (1, 'a'),
  (2, 'b'),
  (3, 'c')
  ```
- To return the table with the column name **id** and **name**:
  ```
  SELECT * FROM (values (1, 'a'), (2, 'b'),(3, 'c')) AS t (id, name);
  ```
- Create a table with the column **id** and **name**:
  ```
  CREATE TABLE example AS
   SELECT * FROM (VALUES (1, 'a'), (2, 'b'), (3, 'c')) AS t (id, name);
  ```

1.3.15 SHOW Syntax Overview

The **SHOW** syntax is used to view information about database objects. The **LIKE** clause is used to filter database objects. The following lists the matching rules. For details, see **SHOW TABLES**.

Rule 1: Underscores (_) can be used to match any single character.

Rule 2: Percent signs (%) can be used to match zero or any number of characters.

Rule 3: Asterisks (*) can be used to match zero or any number of characters.

Rule 4: Vertical bars (|) can be used to separate multiple rules.

Rule 5: To use an underscore (_) as a matching condition, use **ESCAPE** to specify an escape character to escape the underscore (_) so that it will not be parsed according to rule 1.

1.3.16 SHOW SCHEMAS (DATABASES)

Syntax

SHOW SCHEMAS|DATABASES [(FROM| IN) catalog] [LIKE pattern [ESCAPE escapeChar]]

Description

In this statement, **DATABASES** and **SCHEMAS** are conceptually equivalent and interchangeable. This statement is used to list all schemas defined in MetaStore. The optional clause **LIKE** can use rule operations to filter results. It supports the wildcards '*' (matching any character) and '|' (matching optional items).

Example

List all schemas of the current catalog:
```
SHOW SCHEMAS;
```

List all schemas whose schema name prefix is **t** in a specified catalog:

```
SHOW SCHEMAS FROM hive LIKE 't%';

--Equivalent writing:
SHOW SCHEMAS IN hive LIKE 't%';
```

If a character in the matching character string conflicts with the wildcard, you can specify an escape character to identify the character. For example, to query all schemas whose schema name prefix is **pm_** in the **hive** catalog, set the escape character to **/**.

```
SHOW SCHEMAS IN hive LIKE 'pm/_%' ESCAPE '/';
```

1.3.17 SHOW TABLES

Syntax

SHOW TABLES [(FROM | IN) schema] [LIKE pattern [ESCAPE escapeChar]]

Description

This expression is used to list all tables in a specified schema. If no schema is specified, the current schema is used by default.

The optional parameter **like** is used for keyword-based matching.

Example

```
--Create a test table.
Create table show_table1(a int);
Create table show_table2(a int);
Create table showtable5(a int);
Create table intable(a int);
Create table fromtable(a int);

-- Match a single character '_'.
show tables in  default like 'show_table_';
  Table
-------------
 show_table1
 show_table2
(2 rows)

-- Match multiple characters '*' and '%'.
show tables in default like 'show%';
Table
-------------
 show_table1
 show_table2
 showtable5
(3 rows)

show tables in default like 'show*';
Table
-------------
 show_table1
 show_table2
 showtable5
(3 rows)

-- The escape character is used. In the second example, '_' is used as the filter condition. The result set does
not contain showtable5.
show tables in default like 'show_%';
    Table
-------------
```

```
 show_table1
 show_table2
 showtable5
(3 rows)

show tables in default like 'show$_%' ESCAPE '$';
    Table
-------------
 show_table1
 show_table2
(2 rows)

-- If multiple conditions are met, query the tables whose names start with show_ or in in default.
show tables in default like 'show$_%|in%' ESCAPE '$';
    Table
-------------
 intable
 show_table1
 show_table2
(3 rows)
```

1.3.18 SHOW TBLPROPERTIES TABLE|VIEW

Syntax

SHOW TBLPROPERTIES table_name|view_name[(property_name)]

Description

If you do not specify an attribute keyword, this statement returns all table attributes. Otherwise, this statement returns the attribute that matches the specified keyword.

Example

```
-- View all table attributes of show_table1.
                            SHOW TBLPROPERTIES
---------------------------------------------------------------------------
        STATS_GENERATED_VIA_STATS_TASK  'workaround for potential lack of HIVE-12730'
        auto.purge              'false'
        numFiles                '0'
        numRows                 '0'
        orc.compress.size       '262144'
        orc.compression.codec   'GZIP'
        orc.row.index.stride    '10000'
        orc.stripe.size         '67108864'
        presto_query_id         '20230909_095107_00042_2hwbg@default@HetuEngine'
        presto_version          '399'
        rawDataSize             '0'
        totalSize               '0'
        transient_lastDdlTime   '1694253067'

(1 row)

-- View the compression algorithm of show_table1.
SHOW TBLPROPERTIES show_table1('orc.compression.codec');
SHOW TBLPROPERTIES
---------------------
 GZIP
(1 row)
```

1.3.19 SHOW TABLE/PARTITION EXTENDED

Syntax

SHOW TABLE EXTENDED [IN | FROM schema_name] LIKE 'identifier_with_wildcards' [PARTITION (partition_spec)]

Description

This statement is used to display details about a table or partition.

Regular expressions can be used to match multiple tables at the same time, but cannot be used to match partitions.

The displayed information includes basic table information and file system information. The file system information includes the total number of files, total file size, maximum file length, minimum file length, last access time, and last update time. For a specified partition, its file system information is provided, instead of the file system information of the table where the partition is located.

Parameters

- IN | FROM schema_name

  This parameter specifies the schema name. If this parameter is not specified, the current schema is used by default.
- LIKE 'identifier_with_wildcards'

  identifier_with_wildcards supports only rule matching expressions that contain asterisks (*) and vertical bars (|).

  The asterisk (*) can match one or more characters. The vertical bar (|) separates multiple matching rules.

  Spaces at the beginning and end of a rule matching expression are not used for matching.
- partition_spec

  This parameter is optional. It specifies the partition list using key pairs. Multiple key pairs are separated by commas. Table names do not support fuzzy match when partitions are specified.

Example

```
-- Data preparation
create schema show_schema;

create table show_table1(a int,b string);
create table show_table2(a int,b string);
create table from_table1(a int,b string);
create table in_table1(a int,b string);

-- Query the detailed information about tables whose names start with "show".
show table extended  like 'show*';
                              tab_name
--------------------------------------------------------------------------
 tableName:show_table1
 owner:admintest
 location:obs://bucket/user/hive/warehouse/show_schema.db/show_table1
 InputFormat:org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
 OutputFormat:org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat
 columns:struct columns {int a,string b}
 partitioned:false
 partitionColumns:
 totalNumberFiles:0
 totalFileSize:0

 tableName:show_table2
 owner:admintest
 location:obs://bucket/user/hive/warehouse/show_schema.db/show_table2
 InputFormat:org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
 OutputFormat:org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat
 columns:struct columns {int a,string b}
 partitioned:false
 partitionColumns:
 totalNumberFiles:0
 totalFileSize:0

(1 row)

-- Query the detailed information about tables whose names start with "from" or "show".
 show table extended  like 'from*|show*';
                              tab_name
----------------------------------------------------------------------
 tableName           show_table1
 owner               admintest
 location            obs://bucket/user/hive/warehouse/show_table1
 InputFormat         org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
 OutputFormat        org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat
 columns             struct columns {int a,string b}
 partitioned         false
 partitionColumns
 totalNumberFiles    0
 totalFileSize       null

 tableName           from_table1
 owner               admintest
 location            obs://bucket/user/hive/warehouse/from_table1
 InputFormat         org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
 OutputFormat        org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat
 columns             struct columns {int a,string b}
 partitioned         false
 partitionColumns
 totalNumberFiles    0
 totalFileSize       null

 tableName           show_table2
 owner               admintest
 location            obs://bucket/user/hive/warehouse/show_table2
 InputFormat         org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
 OutputFormat        org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat
 columns             struct columns {int a,string b}
 partitioned         false
 partitionColumns
 totalNumberFiles    0
 totalFileSize       null

(1 row)
-- Query the detailed information about the page_views table in the web schema.
 show table extended from web like 'page*';
                                tab_name
---------------------------------------------------------------------------
 tableName:page_views
 owner:admintest
 location:obs://bucket/user/web.db/page_views
 InputFormat:org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
 OutputFormat:org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat
 columns:struct columns {timestamp view_time,bigint user_id,string page_url}
 partitioned:true
 partitionColumns: struct partition_columns {date ds,string country}
```

```
totalNumberFiles:0
totalFileSize:0

(1 row)
```

1.3.20 SHOW FUNCTIONS

Syntax

SHOW FUNCTIONS [LIKE pattern [ESCAPE escapeChar]];

SHOW EXTERNAL FUNCTIONS;

SHOW EXTERNAL FUNCTION qualified_function_name;

Description

Displays the definitions of all built-in functions.

Displays the description of all Java functions.

Displays the definition of a specified function.

Example

```
SHOW functions;

-- Use the LIKE clause.
show functions like 'boo_%';
 Function | Return Type | Argument Types | Function Type | Deterministic | Description
----------|-------------|----------------|---------------|---------------|-------------
 bool_and | boolean     | boolean        | aggregate     | true          |
 bool_or  | boolean     | boolean        | aggregate     | true          |
(2 rows)

-- If a character in the matching character string conflicts with the wildcard, you can specify an escape
character to identify the character. For example, to query all tables whose table name prefix is t_ in the
default schema, set the escape character to \.
SHOW FUNCTIONS LIKE 'array\_%' escape '\';
  Function      | Return Type |         Argument Types          | Function Type | Deterministic
|                                          Description                                          | Variable Arity | Built In
-----------------|-------------|---------------------------------|---------------|---------------
|-----------------------------------------------------------------------------------------------|----------------|----
------
 array_agg       | array(T)    | T                               | aggregate     | true
| return an array of values                                                                     | false          | true
 array_contains  | boolean     | array(T), T                     | scalar        | true
| Determines whether given value exists in the array                                            | false          |
true
 array_distinct  | array(E)    | array(E)                        | scalar        | true
| Remove duplicate values from the given array                                                  | false          |
true
 array_except    | array(E)    | array(E), array(E)              | scalar        | true
| Returns an array of elements that are in the first array but not the second, without duplicates.       |
false          | true
 array_intersect | array(E)    | array(E), array(E)              | scalar        | true
| Intersects elements of the two given arrays                                                   | false          | true
 array_join      | varchar     | array(T), varchar               | scalar        | true
| Concatenates the elements of the given array using a delimiter and an optional string to replace nulls |
false          | true
 array_join      | varchar     | array(T), varchar, varchar      | scalar        | true
| Concatenates the elements of the given array using a delimiter and an optional string to replace nulls |
false          | true
 array_max       | T           | array(T)                        | scalar        | true
| Get maximum value of array                                                                    | false          | true
```

```
 array_min       | T           | array(T)                      | scalar        | true
| Get minimum value of array                                                          | false          | true
 array_position  | bigint      | array(T), T                   | scalar        | true
| Returns the position of the first occurrence of the given value in array (or 0 if not found)          |
false          | true
 array_remove    | array(E)    | array(E), E                   | scalar        | true
| Remove specified values from the given array                                        | false          | true
 array_sort      | array(E)    | array(E)                      | scalar        | true
| Sorts the given array in ascending order according to the natural ordering of its elements.           |
false          | true
 array_sort      | array(T)    | array(T), function(T,T,integer) | scalar        | true
| Sorts the given array with a lambda comparator.                                     | false          |
true
 array_union     | array(E)    | array(E), array(E)            | scalar        | true
| Union elements of the two given arrays                                              | false          | true

-- View all Java functions.
SHOW external functions;
  Function                   |   Owner
----------------------------|-----------
 example.namespace02.repeat | admintest
 hetu.default.add_two       | admintest
(2 rows)

-- View the definition of a specified function.
SHOW external function example.namespace02.repeat;
         External Function
-------------------------------------
External FUNCTION example.namespace02.repeat (
    s varchar,
    n integer
 )
 RETURNS varchar

 COMMENT 'repeat'
 LANGUAGE JAVA
 DETERMINISTIC
 CALLED ON NULL INPUT
 SYMBOL com.test.udf.hetuengine.functions.repeat
 URI obs://bucket/user/hetuserver/udf/data/hetu_udf/udf-test-0.0.1-SNAPSHOT.jar

 FUNCPROPERTIES (
    owner = 'admintest'
 )
```

1.3.21 SHOW PARTITIONS

Syntax

SHOW PARTITIONS [catalog_name.][db_name.]table_name [PARTITION (partitionSpecs)];

Description

This statement is used to list all specified partitions.

Example

```
SHOW PARTITIONS test PARTITION(hr = '12', ds = 12);
SHOW PARTITIONS test PARTITION(ds > 12);
```

1.3.22 SHOW COLUMNS

Syntax

SHOW COLUMNS [FROM | IN] table

Description

This expression is used to list the column information of a specified table.

Example

List the column information of the **fruit** table:

```
SHOW COLUMNS FROM fruit;
SHOW COLUMNS IN fruit;
```

1.3.23 SHOW CREATE TABLE

Syntax

SHOW CREATE TABLE table_name

Description

This statement is used to display the SQL creating statement for a specified data table.

Example

To display the SQL statements that can be used to create the table **orders**:

```
CREATE TABLE orders (
 orderkey bigint,
 orderstatus varchar,
 totalprice double,
 orderdate date
 )
 WITH (format = 'ORC', location='obs://bucket/user',orc_compress='ZLIB',external=true, "auto.purge"=false);

show create table orders;

                  Create Table
-------------------------------------------------
 CREATE TABLE hive.default.orders (
    orderkey bigint,
    orderstatus varchar,
    totalprice double,
    orderdate date
 )
 WITH (
    external_location = 'obs://bucket/user',
    format = 'ORC',
    orc_compress = 'ZLIB',
    orc_compress_size = 262144,
    orc_row_index_stride = 10000,
    orc_stripe_size = 67108864
 )
(1 row)
```

1.3.24 SHOW VIEWS

Syntax

SHOW VIEWS [IN/FROM database_name] [LIKE pattern [ESCAPE escapeChar]]

Description

This statement is used to list all views that meet the conditions in a specified schema.

By default, the current schema is used. You can also use the **in/from** clause to specify a schema.

The **LIKE** clause is used to filter views whose names meet the regular expression. If this clause is not used, all views are listed. The matched views are sorted in alphabetic order.

Currently, the regular expression supports only the asterisk (*) (matching any character).

Example

Create views:

```
Create schema test1;

Create table t1(id int, name string);
Create view v1 as select * from t1;
Create view v2 as select * from t1;
Create view t1view as select * from t1;
Create view t2view as select * from t1;

Show views;
Table
--------
 t1view
 t2view
 v1
 v2
(4 rows)

Show views like 'v1';
 Table
-------
 v1
(1 row)

Show views 'v_';
Table
-------
 v1
 v2
(2 rows)
show views like 't*';
 Table
--------
 t1view
 t2view

Show views in test1;
 Table
--------
```

```
 t1view
 t2view
 v1
 v2
(4 rows)
```

1.3.25 SHOW CREATE VIEW

Syntax

SHOW CREATE VIEW view_name

Description

This statement is used to display the SQL creation statement of a specified data view.

Example

To display the SQL statement that can be used to create the **order_view** view:

```
SHOW CREATE VIEW test_view;
              Create View
---------------------------------------
 CREATE VIEW hive.default.test_view AS
 SELECT
   orderkey
 , orderstatus
 , (totalprice / 4) quarter
 FROM
   orders
(1 row)
```

1.4 DML Syntax

1.4.1 INSERT

Syntax

INSERT { INTO | OVERWRITE } [TABLE] table_name [(column_list)] [PARTITION (partition_clause)] {select_statement | VALUES (value [, value ...]) [, (value [, value ...]) ...] }

FROM from_statement INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement

FROM from_statement INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...) select_statement

Remarks

If there is only one field in the data table and the field type is **row** or **struct**, use **row** to encapsulate the type when inserting data.

```
-- When a complex type is inserted into a single field table, the complex type must be wrapped by row().
CREATE TABLE test_row (id row(c1 int, c2 string));
```

```
INSERT INTO test_row values row(row(1, 'test'));

--The complex type of a multi-field table can be directly inserted.
CREATE TABLE test_multy_value(id int, col row(c1 int, c2 string));

INSERT INTO test_multy_value values (1,row(1,'test'));
```

Description

- This statement is used to insert a new data row into a table.
- If a list of column names is specified, the list of column names must exactly match the name of the column list generated by the **query** statement. The value of each column that is not in the column name list is set to **null**.
- If no column name list is specified, the column generated by the query statement must exactly match the column to be inserted.
- When **INSERT INTO** is used, data is added to the table. When **INSERT OVERWRITE** is used, if table property **auto.purge** is set to **true**, data in the original table data is directly deleted and new data is written to the table.
- If the object table is a partitioned table, **insert overwrite** deletes the data in the corresponding partition instead of all data.
- The **table** keyword following INSERT INTO is optional to be compatible with the Hive syntax.

Example

- Create the **fruit** and **fruit_copy** tables.
  ```
  create table fruit (name varchar,price double);
  create table fruit_copy (name varchar,price double);
  ```
- Inserts a row of data into the **fruit** table.
  ```
  insert into fruit values('LIchee',32);
  --The following is an example of the compatible format with the table keyword:
  insert into table fruit values('Cherry',88);
  ```
- Insert multiple lines of data into the **fruit** table.
  ```
  insert into fruit values('banana',10),('peach',6),('lemon',12),('apple',7);
  ```
- Load the data lines in the **fruit** table to the **fruit_copy** table. After the execution, there are five records in the table.
  ```
  insert into fruit_copy select * from fruit;
  ```
- Clear the **fruit_copy** table, and then load the data in the **fruit** table to the table. After the execution, there are two records in the **fruit_copy** table.
  ```
  insert overwrite fruit_copy select *  from fruit limit 2;
  ```
- For the VARCHAR type, INSERT can be used only when the column length defined in the target table is greater than the actual column length of the source table. SELECT queries data from the source table and inserts the data to the target table.
  ```
  create table varchar50(c1 varchar(50));
  insert into varchar50 values('hetuEngine');
  create table varchar100(c1 varchar(100));
  insert into varchar100 select * from varchar50;
  ```
- When the **insert overwrite** statement is used for a partitioned table, only the data in the partition where the inserted value is located is cleared, not the entire table.
  ```
  --Create a table.
  create table test_part (id int, alias varchar) partitioned by (dept_id int, status varchar);

  insert into test_part  partition(dept_id=10, status='good') values (1, 'xyz'), (2, 'abc');
  ```

```
select * from test_part order by id;
 id | alias | dept_id | status
----|-------|---------|--------
  1 | xyz   |      10 | good
  2 | abc   |      10 | good
(2 rows)

--Clear the partition(dept_id=25, status='overwrite') partition and insert a data record.
insert overwrite test_part (id, alias, dept_id, status) values (3, 'uvw', 25, 'overwrite');
 select * from test_part ;
 id | alias | dept_id |  status
----|-------|---------|-----------
  1 | xyz   |      10 | good
  2 | abc   |      10 | good
  3 | uvw   |      25 | overwrite

--Clear the partition(dept_id=10, status='good') partition and insert a data record.
insert overwrite test_part (id, alias, dept_id, status) values (4, 'new', 10, 'good');
select * from test_part order by id;
 id | alias | dept_id |  status
----|-------|---------|-----------
  3 | uvw   |      25 | overwrite
  4 | new   |      10 | good
(2 rows)

-- Insert data to a partitioned table.
create table test_p_1(name string, age int) partitioned by (provice string, city string);

 create table test_p_2(name string, age int) partitioned by (provice string, city string);

-- Add data to test_p_1.
 insert into test_p_1 partition (provice = 'hebei', city= 'baoding') values ('xiaobei',15),( 'xiaoming',22);
-- Insert data into test_p_2 based on test_p_1.

-- Method 1
from test_p_1 insert into table test_p_2 partition (provice = 'hebei', city= 'baoding') select name,age;

-- Method 2
insert into test_p_2 partition(provice = 'hebei', city= 'baoding') select name,age from test_p_1;
```

Precautions

Data cannot be inserted into foreign tables.

1.5 DQL Syntax

1.5.1 SELECT

Syntax

[/*+ query_rewrite_hint*/]

[WITH [RECURSIVE] with_query [, ...]]

SELECT [ALL | DISTINCT] select_expression [, ...]

[FROM from_item [, ...]]

[WHERE condition]

[GROUP BY [ALL | DISTINCT] grouping_element [, ...]]

[HAVING condition]

[{ UNION | INTERSECT | EXCEPT } [ALL | DISTINCT] select]

[ORDER BY expression [ASC | DESC] [, ...]]

[OFFSET count [ROW | ROWS]]

[LIMIT { count | ALL }]

[FETCH { FIRST | NEXT } [count] { ROW | ROWS } { ONLY | WITH TIES }]

NOTE

- **from_item** can be used in the following formats:
  - table_name [[AS] alias [(column_alias [, ...])]]
  - from_item join_type from_item [ON join_condition | USING (join_column [, ...])]
  - table_name [[AS] alias [(column_alias [, ...])]]
    MATCH_RECOGNIZE pattern_recognition_specification
    [[AS] alias [(column_alias [, ...])]]
- **join_type** can be used in the following formats:
  - [INNER] JOIN
  - LEFT [OUTER] JOIN
  - RIGHT [OUTER] JOIN
  - FULL [OUTER] JOIN
  - LEFT [SEMI] JOIN
  - RIGHT [SEMI] JOIN
  - LEFT [ANTI] JOIN
  - RIGHT [ANTI] JOIN
  - CROSS JOIN
- **grouping_element** can be:
  - ()
  - expression
  - GROUPING SETS ((column [, ...]) [, ...])
  - CUBE (column [, ...])
  - ROLLUP (column [, ...])

Description

This statement is used to retrieve row data from zero or more tables.

Query the content of the **stu** table.

```
SELECT id,name FROM stu;
```

1.5.2 WITH

The WITH clause defines the naming relationship of query clauses, which can flatten nested queries or simplify subquery statements. For example, the following query statements are equivalent:

```
SELECT name, maxprice FROM (SELECT name, MAX(price) AS maxprice FROM fruit GROUP BY name) AS x;
```

```
WITH x AS (SELECT name, MAX(price) AS maxprice FROM fruit GROUP BY name) SELECT name, maxprice
FROM x;
```

Multiple Subqueries

```
with
t1 as(select name,max(price) as maxprice from fruit group by name),
t2 as(select name,avg(price) as avgprice from fruit group by name)
select t1.*,t2.* from t1 join t2 on t1.name = t2.name;
```

WITH Chain Form

```
WITH
x AS (SELECT a FROM t),
y AS (SELECT a AS b FROM x),
z AS (SELECT b AS c FROM y)
SELECT c FROM z;
```

1.5.3 GROUP BY

GROUP BY

GROUP BY groups the output rows of a SELECT statement into groups that contain matching values. A simple GROUP BY can contain any expression consisting of input columns, or select the sequence number of the output column by position.

The following queries are equivalent:

```
SELECT count(*), nationkey FROM customer GROUP BY 2;
SELECT count(*), nationkey FROM customer GROUP BY nationkey;
```

GROUP BY can group the output by the input column names that do not appear in the output of the SELECT statement.

Example:

```
SELECT count(*) FROM customer GROUP BY mktsegment;
GROUPING SETS
```

You can specify multiple columns for grouping. The result column that does not belong to the grouping column is set to **NULL**. Queries with complex grouping syntax (GROUPING SETS, CUBE, or ROLLUP) read the underlying data source only once, while queries using UNION ALL read the underlying data three times. This is why queries that use UNION ALL can produce inconsistent results when the data source is not deterministic.

```
-- Create a shipping table:
create table shipping(origin_state varchar(25),origin_zip integer,destination_state
varchar(25) ,destination_zip integer,package_weight integer);

--Insert data.
insert into shipping values ('California',94131,'New Jersey',8648,13),
('California',94131,'New Jersey',8540,42),
('California',90210,'Connecticut',6927,1337),
('California',94131,'Colorado',80302,5),
('New York',10002,'New Jersey',8540,3),
('New Jersey',7081,'Connecticut',6708,225);

-- Query the grouping sets.
SELECT
    origin_state,
    origin_zip,
    destination_state,
    sum( package_weight )
FROM shipping
GROUP BY GROUPING SETS (
        ( origin_state ),
    ( origin_state, origin_zip ),
    ( destination_state ));
--Logically, this query is equivalent to the union all of multiple group queries.
SELECT origin_state, NULL,NULL,sum( package_weight ) FROM shipping GROUP BY origin_state UNION
ALL  SELECT origin_state,origin_zip,NULL,sum( package_weight ) FROM shipping GROUP BY
origin_state,origin_zip UNION ALL  SELECT NULL,NULL,destination_state,sum( package_weight ) FROM
shipping GROUP BY  destination_state;
--Result
 origin_state | origin_zip | destination_state | _col3
--------------|------------|-------------------|-------
 New Jersey   |       NULL | NULL              |   225
 California   |      94131 | NULL              |    60
 California   |       NULL | NULL              |  1397
 New York     |      10002 | NULL              |     3
 NULL         |       NULL | New Jersey        |    58
 NULL         |       NULL | Connecticut       |  1562
 California   |      90210 | NULL              |  1337
 New York     |       NULL | NULL              |     3
 NULL         |       NULL | Colorado          |     5
 New Jersey   |       7081 | NULL              |   225
(10 rows)
```

CUBE

Generate all possible groups for given columns. For example, the possible groups of (**origin_state**, **destination_state**) are **(origin_state, destination_state)**, **(origin_state)**, **(destination_state)**, and **()**.

```
SELECT
    origin_state,
    destination_state,
    sum( package_weight )
FROM
    shipping
GROUP BY
    CUBE ( origin_state, destination_state );
-- Equivalent to:
SELECT
origin_state,
destination_state,
sum( package_weight )
FROM
    shipping
GROUP BY
    GROUPING SETS (
        ( origin_state, destination_state ),
        ( origin_state ),
    ( destination_state ),
    ());
```

ROLLUP

Generates partial possible subtotals for a given set of columns.

```
SELECT
    origin_state,
    origin_zip,
    sum( package_weight )
FROM
    shipping
GROUP BY
    ROLLUP ( origin_state, origin_zip );
-- Equivalent to:
SELECT
origin_state,
origin_zip,
sum( package_weight )
FROM
   shipping
GROUP BY
   GROUPING SETS ((origin_state,origin_zip ),( origin_state ),());
```

NOTE

Currently, GROUP BY does not support column aliases. For example:

select count(userid) as num ,dept as aaa from salary group by aaa having sum(sal)>2000;

The following error is reported:

Query 20210630_084610_00018_wc8n9@default@HetuEngine failed: line 1:63: Column 'aaa' cannot be resolved

1.5.4 HAVING

HAVING

HAVING is used with aggregate functions and GROUP BY to control which groups are selected. HAVING filters out groups that do not meet specified conditions after grouping and aggregation calculation.

Example:

```
SELECT count(*), mktsegment, nationkey,
CAST(sum(acctbal) AS bigint) AS totalbal
FROM customer
GROUP BY mktsegment, nationkey
HAVING sum(acctbal) > 5700000
ORDER BY totalbal DESC;
```

1.5.5 UNION | INTERSECT | EXCEPT

UNION, INTERSECT, and EXCEPT are collection operations. All of them are used to merge the result sets of multiple SELECT statements into a single result set.

UNION

UNION merges all rows in the result set of the first query with rows in the result set of the second query.

query UNION [ALL | DISTINCT] query

ALL and DISTINCT indicate whether duplicate rows are returned. ALL returns all rows. DISTINCT returns only one row. If this parameter is not specified, the default value DISTINCT is used.

INTERSECT

query INTERSECT [DISTINCT] query

INTERSECT returns only the rows that intersect the results of the first and second queries. The following is an example of one of the simplest INTERSECT clauses. It selects values **13** and **42** and merges this result set with the second query that selects value 13. Because **42** is only in the result set of the first query, it is not included in the final result:

```
SELECT * FROM (VALUES 13,42) INTERSECT SELECT 13;
 _col0 -------
    13
 (1 row)
```

EXCEPT

query EXCEPT [DISTINCT] query

EXCEPT returns rows that are in the first query result and not in the second query result.

```
SELECT * FROM (VALUES 13, 42) EXCEPT SELECT 13;
 _col0
-------
   42
(1 row)
```

NOTE

Currently, the having clause does not support column aliases. For example:

select count(userid) as num ,dept as aaa from salary group by dept having aaa='d1';

The following error is reported:

Query 20210630_085136_00024_wc8n9@default@HetuEngine failed: line 1:75: Column 'aaa' cannot be resolved

1.5.6 ORDER BY

ORDER BY

The ORDER BY clause is used to sort the result set by one or more output expressions.

ORDER BY expression [ASC | DESC] [NULLS { FIRST | LAST }] [, ...]

Each expression can consist of output columns or you can select the sequence number of an output column by position.

The ORDER BY clause is executed after the GROUP BY or HAVING clause and before the OFFSET, LIMIT, or FETCH FIRST clause.

NOTICE

According to the SQL specification, the ORDER BY clause affects only the row order of the query result that contains this clause. HetuEngine complies with this specification and deletes redundant usage of the clause to avoid negative impact on performance.

For example, when an INSERT statement is executed, the ORDER BY clause does not affect the inserted data. It is a redundant operation and adversely affects the overall performance of the INSERT statement. Therefore, HetuEngine skips this ORDER BY clause.

- ORDER BY applies only to the SELECT clause.
  ```
  INSERT INTO some_table
  SELECT * FROM another_table
  ORDER BY field;
  ```

- The example of ORDER BY redundancy is nested query, which does not affect the result of the entire statement.
  ```
  SELECT *
  FROM some_table
  JOIN (SELECT * FROM another_table ORDER BY field) u
  ON some_table.key = u.key;
  ```

1.5.7 OFFSET

OFFSET

OFFSET is used to discard the first several rows of data in the result set.

OFFSET count [ROW | ROWS]

If ORDER BY exists, OFFSET applies to the sorted result set. OFFSET discards the first several rows of data and retains the sorted data set.

```
SELECT name FROM fruit ORDER BY name OFFSET 3;
name
------------
peach
pear
watermelon
(3 rows)
```

Otherwise, if ORDER BY is not used, the discarded row may be any row. If the number of rows specified by OFFSET is greater than or equal to the size of the result set, the returned result is null.

1.5.8 LIMIT | FETCH FIRST

Both LIMIT and FETCH FIRST can limit the number of rows in the result set. **LIMIT** and **OFFSET** can be used together for pagination query.

LIMIT

LIMIT { count | ALL }

The following query limits the number of returned rows to 5:

```
SELECT * FROM fruit LIMIT 5;
```

LIMIT ALL has the same function as omitting LIMIT.

FETCH FIRST

FETCH { FIRST | NEXT } [count] { ROW | ROWS } { ONLY | WITH TIES

FETCH FIRST supports the FIRST or NEXT keywords and the ROW or ROWS keywords. These keywords are equivalent and do not affect the execution of queries.

- If FETCH FIRST is not specified, the default value **1** is used.
  ```
  SELECT orderdate FROM orders FETCH FIRST ROW ONLY;
    orderdate
  --------------
   2020-11-11

  SELECT *  FROM new_orders FETCH FIRST 2 ROW ONLY;
     orderkey   | orderstatus | totalprice |  orderdate
  ```

```
--------------|-------------|------------|--------------
 202011181113 | online      |     9527.0 | 2020-11-11
 202011181114 | online      |      666.0 | 2020-11-11
(2 rows)
```

- If OFFSET is used, LIMIT or FETCH FIRST applies to the result set after OFFSET:

```
SELECT * FROM (VALUES 5, 2, 4, 1, 3) t(x) ORDER BY x OFFSET 2  FETCH FIRST ROW ONLY;
 x
---
 3
(1 row)
```

- For the FETCH FIRST clause, the ONLY or WITH TIES parameter controls which rows are contained in the result set.

  If the ONLY parameter is specified, the result set is limited to the first several rows that contain the number of parameters.

  If WITH TIES is specified, the ORDER BY clause must be used. The result set contains the basic result set of the first several rows that meet the conditions and additional rows. These extra return rows are the same as the parameters of ORDER BY in the last row of the basic result set:

```
CREATE TABLE nation (name varchar,  regionkey integer);

insert into nation values ('ETHIOPIA',0),('MOROCCO',0),('ETHIOPIA',2),('KENYA',2),('ALGERIA',0),
('MOZAMBIQUE',0);

--Return all records whose regionkey is the same as the first record.
SELECT name, regionkey FROM nation ORDER BY regionkey FETCH FIRST ROW WITH TIES;
    name    | regionkey
------------|-----------
 ALGERIA    |         0
 ETHIOPIA   |         0
 MOZAMBIQUE |         0
 MOROCCO    |         0
(4 rows)
```

1.5.9 TABLESAMPLE

There are two sampling methods: BERNOULLI and SYSTEM.

Neither of these sampling methods allows you to limit the number of rows returned by the result set.

BERNOULLI

Each row is selected to the sample table based on the specified sampling rate. When the Bernoulli method is used to sample a table, all physical blocks of the table are scanned and some rows are skipped (based on the comparison between the sampling percentage and the random value calculated at run time). The probability that the result contains one row is irrelevant to any other rows. This does not reduce the time required to read the sample table from disk. If the sampling output is further processed, the total query time may be affected.

```
SELECT * FROM users TABLESAMPLE BERNOULLI (50);
```

SYSTEM

This sampling method divides a table into logical segments of data and samples the table based on the granularity. This sampling method either selects all rows from a particular data segment or skips it (based on a comparison between the sampling percentage and a random value calculated at run time). The selection of rows in the system sampling depends on the connector used. For example, if a Hive data source is used, this depends on the layout of the data on OBS. This sampling method cannot ensure an independent sampling probability.

```
SELECT * FROM users TABLESAMPLE SYSTEM (75);
```

1.5.10 UNNEST

UNNEST can expand ARRAY or MAP to form a relation. ARRAYS is expanded into a single column, and MAP is expanded into two columns (**key**, **value**). UNNEST can also be used together with multiple parameters, which will be expanded into multiple columns with the same number of rows as the maximum base parameter (other columns are filled with nulls). UNNEST can choose to use the WITH ORDINALITY clause, in which case an additional ORDINALITY column is added at the end. UNNEST is usually used together with JOIN, which can reference columns in the left relationship of JOIN.

Using a Separate Column

```
SELECT student, score FROM tests CROSS JOIN UNNEST(scores) AS t (score);
```

Using Multiple Columns

```
SELECT numbers, animals, n, a
FROM (
VALUES
(ARRAY[2, 5], ARRAY['fishg', 'cat', 'bird']),
(ARRAY[7, 8, 9], ARRAY['cow', 'rabbit'])
) AS x (numbers, animals)
CROSS JOIN UNNEST(numbers, animals) AS t (n, a);
```

1.5.11 JOINS

Data of multiple relations can be combined.

HetuEngine supports the following types of JOIN: CROSS JOIN, INNER JOIN, OUTER JOIN (LEFT JOIN, RIGHT JOIN, FULL JOIN), SEMIN JOIN, and ANTI JOIN.

CROSS JOIN

CROSS JOIN returns the Cartesian product of two relationships. You can specify multiple relations using the CROSS JOIN syntax or the FROM subclause.

The following queries are equivalent:

```
SELECT * FROM nation CROSS JOIN region;
SELECT * FROM nation, region;
```

INNER JOIN

Rows can be returned only when two tables contain at least one piece of matched data, which is equivalent to JOIN. The clause can also be converted to an equivalent WHERE statement as follows:

```
SELECT * FROM nation (INNER) JOIN region ON nation.name=region.name;
SELECT * FROM nation ,region WHERE nation.name=region.name;
```

OUTER JOIN

OUTER JOIN returns all rows, both matched and unmatched, in both tables. It subdivides further into:

- Left outer join: LEFT JOIN or LEFT OUTER JOIN. This clause returns all rows from the left table (nation) and matched rows from the right table (region) based on the left table. If a row in the left table is not matched in the right table, the value of the row in the right table is NULL.
- Right outer join: RIGHT JOIN or RIGHT OUTER JOIN. This clause returns all rows from the right table (region) and matched rows from the left table (nation) based on the right table. If a row in the right table is not matched in the left table, the value of the row in the left table is NULL.
- Full outer join: FULL JOIN or FULL OUTER JOIN. This clause returns matched rows as long as there are matches in either of the tables. It is equivalent to the combination of LEFT JOIN and RIGHT JOIN.

```
SELECT * FROM nation LEFT (OUTER) JOIN region ON nation.name=region.name;
SELECT * FROM nation RIGHT (OUTER) JOIN region ON nation.name=region.name;
SELECT * FROM nation FULL (OUTER) JOIN region ON nation.name=region.name;
```

LATERAL

The LATERAL keyword can be added to the FROM subquery to allow referencing of the columns provided by the FROM item.

```
SELECT name, x, y FROM nation CROSS JOIN LATERAL (SELECT name || ' :-' AS x) CROSS JOIN LATERAL
(SELECT x || ')' AS y);
```

SEMI JOIN and ANTI JOIN

When a table finds a matched record in another table, **semi-join** returns the record in the first table. Contrary to conditional join, the table on the left node returns only one record even if several matching records are found on the right node. In addition, no record in the table on the right node is returned. The semi-join usually uses IN or EXISTS as the connection condition.

anti-join is opposite to **semi-join**. That is, the records in the first table are returned only when no matching record is found in the second table. It is used when **not exists** or **not in** is used.

Other supported conditions are as follows:

- Multiple conditions in the WHERE clause
- Alias relationships
- Subscript expressions
- Dereference expressions
- Forcible conversion expressions
- Specific functions

NOTICE

Currently, multiple semi or anti join expressions are supported only when the columns in the first table are queried in the subsequent join expressions and are not related to other join expressions.

Example:

```
CREATE SCHEMA testing ;

CREATE TABLE table1(id int, name varchar,rank int);

INSERT INTO table1 VALUES(10,'sachin',1),(45,'rohit',2),(46,'rohit',3),(18,'virat',4),(25,'dhawan',5);

CREATE TABLE table2(serial int,name varchar);

INSERT INTO table2 VALUES(1,'sachin'),(2,'sachin'),(3,'rohit'),(4,'virat');

CREATE TABLE table3(serial int, name varchar,country varchar);

INSERT INTO table3 VALUES(1,'sachin','india'),(20,'bhuvi','india'),(45,'boult','newzealand'),
(3,'maxwell','australia'),(45,'rohit','india'),(4,'pant','india'),(10,'KL','india'),(445,'rohit','india');

CREATE TABLE table4(id int, name varchar,rank int);

INSERT INTO table4 VALUES(10,'sachin',1),(45,'rohit',2),(46,'rohit',3),(18,'virat',4),(25,'dhawan',5);

select * from table1 left semi join table2 on table1.name=table2.name where table1.name='rohit' and
table2.serial=3;
 id | name  | rank
----|-------|------
 45 | rohit |    2
 46 | rohit |    3
(2 rows)


select * from table1 left anti join table2 on table1.name=table2.name where table1.name='rohit' and
table2.serial=3;
 id |  name  | rank
----|--------|------
 10 | sachin |    1
 18 | virat  |    4
 25 | dhawan |    5
(3 rows)

select * from table1 right semi join table2 on table1.name=table2.name where table1.name='rohit' and
table2.serial=3;
 serial | name
--------|-------
      3 | rohit
(1 row)

select * from table1 right anti join table2 on table1.name=table2.name where table1.name='rohit' and
table2.serial=3;
 serial |  name
--------|--------
      1 | sachin
      2 | sachin
      4 | virat
(3 rows)

SELECT * FROM table1 t1 LEFT SEMI JOIN table2 t2 on t1.name=t2.name left semi join table3 t3 on
t1.name = t3.name left semi join table4 t4 on t1.name=t4.name;
id | name  | rank
----|-------|------
 10 | sachin |   1
 45 | rohit |    2
 46 | rohit |    3
(3 rows)
```

Qualifying Column Names

When two relations of JOIN have the same column name, the relation alias (if any) or relation name must be used for column reference.

```
SELECT nation.name, region.name FROM nation CROSS JOIN region;
SELECT n.name, r.name FROM nation AS n CROSS JOIN region AS r;
SELECT n.name, r.name FROM nation n CROSS JOIN region r;
```

1.5.12 Subqueries

EXISTS

The EXISTS predicate determines whether to return any row.

```
SELECT name FROM nation WHERE EXISTS (SELECT * FROM region WHERE region.regionkey =
nation.regionkey)
```

IN

It determines whether any value generated by a subquery is equal to a given expression. The IN result complies with the standard null rule. Only one column must be generated for a subquery.

```
SELECT name FROM nation WHERE regionkey IN (SELECT regionkey FROM region)
```

1.5.13 SELECT VIEW CONTENT

Syntax

SELECT column_name FROM view_name

Description

This statement is used to query view content.

```
SELECT * FROM test_view;
```

1.6 Auxiliary Command Syntax

1.6.1 DESCRIBE

Syntax

DESCRIBE [EXTENDED| FORMATTED] table_name

DESCRIBE [EXTENDED| FORMATTED] table_name PARTITION (partition_spec)

Description

This statement is used to view the metadata information of a specified table. Currently, this syntax can display only the metadata of columns, which is equivalent to the SHOW COLUMNS syntax.

After the **EXTENDED** keyword is added, all metadata of the table is displayed in the Thrift serialization format.

If the **FORMATTED** keyword is added, the metadata of the table is displayed in a table.

Example

Display the column information of the **fruit** data table:

```
DESCRIBE fruit;
```

Display the Fruit metadata:

```
DESCRIBE FORMATTED fruit;
                         Describe Formatted Table
------------------------------------------------------------------------------
 # col_name      data_type      comment
 name      varchar
 price      integer

 # Detailed Table Information
 Database:                  default
 Owner:                     admintest
 LastAccessTime:            0
 Location:                  obs://bucket/user/hive/warehouse/fruit
 Table Type:                MANAGED_TABLE

 # Table Parameters:
       Owner                  ggg
       STATS_GENERATED_VIA_STATS_TASK  workaround for potential lack of HIVE-12730
       numFiles               0
       numRows                0
       orc.compress.size      262144
       orc.compression.codec  GZIP
       orc.row.index.stride   10000
       orc.stripe.size        67108864
       presto_query_id        20210308_072339_00075_5ck2k@default@HetuEngine
       presto_version
       rawDataSize            0
       totalSize              0
       transient_lastDdlTime  1615188219

 # Storage Information
 SerDe Library:             org.apache.hadoop.hive.ql.io.orc.OrcSerde
 InputFormat:               org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
 OutputFormat:              org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat
 Compressed:                No
 Num Buckets:               -1
 Bucket Columns:            []
 Sort Columns:              []
 serialization.format:      1
(1 row)

Query 20210309_022835_00007_2i9yy@default@HetuEngine, FINISHED, 1 node
Splits: 1 total, 1 done (100.00%)
0:01 [0 rows, 0B] [0 rows/s, 0B/s];
```

1.6.2 DESCRIBE FORMATTED COLUMNS

Syntax

DESCRIBE FORMATTED [db_name.]table_name [PARTITION partition_spec] col_name

Description

This statement is used to describe the column information of a table or partition and collect statistics on the columns of specified tables or partitions.

Example

```
describe formatted show_table1 a;
  Describe Formatted Column
------------------------------
 col_name             a
 data_type           integer
 min
 max
 num_nulls
 distinct_count       0
 avg_col_len
 max_col_len
 num_trues
 num_falses
 comment
(1 row)
```

1.6.3 DESCRIBE DATABASE| SCHEMA

Syntax

DESCRIBE DATABASE|SCHEMA [EXTENDED] schema_name

Description

DATABASE and **SCHEMA** are equivalent and interchangeable. They have the same meaning.

This statement is used to display the name, comment, and root path of a schema on the file system.

The option **EXTENDED** can be used to display the database attributes of the schema.

Example

```
CREATE SCHEMA web;

DESCRIBE SCHEMA web;
                              Describe Schema
------------------------------------------------------------------------
 web      obs://bucket/user/hive/warehouse/web.db    dli    USER
(1 row)
```

1.6.4 EXPLAIN

Syntax

EXPLAIN [(option [, ...])] statement

The option can be the following:

FORMAT { TEXT | GRAPHVIZ | JSON }

TYPE { LOGICAL | DISTRIBUTED | VALIDATE | IO }

Description

This statement is used to display the logical or distributed execution plan of a statement. It can also be used to verify an SQL statement or analyze I/Os.

The TYPE DISTRIBUTED parameter is used to display the fragmented plan. Each fragment is executed by one or more nodes. **Fragments separation** indicates that data is exchanged between two nodes. **Fragment type** indicates how a fragment is executed and how data is distributed among different fragments.

- SINGLE

  Fragments are executed on a single node.
- HASH

  Fragments are executed on a fixed number of nodes, and the input data is distributed using the hash function.
- ROUND_ROBIN

  Fragments are executed on a fixed number of nodes, and input data is distributed in round-robin mode.
- BROADCAST

  Fragments are executed on a fixed number of nodes, and the input data is broadcast to all nodes.
- SOURCE

  Fragments are executed on the node that accesses the input fragments.

Example

- LOGICAL:

```
CREATE TABLE testTable (regionkey int, name varchar);
EXPLAIN SELECT regionkey, count(*) FROM testTable GROUP BY 1;
                                                  Query Plan
-----------------------------------------------------------------------------------------------------------------------
--------------
 Output[regionkey, _col1]
 │   Layout: [regionkey:integer,
count:bigint]
 │   Estimates: {rows: ? (?), cpu: ?, memory: ?,
network: ?}
 │   _col1 := count
 └─
RemoteExchange[GATHER]
    │   Layout: [regionkey:integer,
count:bigint]
    │   Estimates: {rows: ? (?), cpu: ?, memory: ?,
network: ?}
```

```
  └─ Project[]
     │   Layout: [regionkey:integer,
count:bigint]
     │   Estimates: {rows: ? (?), cpu: ?, memory: ?,
network: ?}
     └─ Aggregate(FINAL)[regionkey]
[$hashvalue]
        │   Layout: [regionkey:integer, $hashvalue:bigint,
count:bigint]
        │   Estimates: {rows: ? (?), cpu: ?, memory: ?,
network: ?}
        │   count := count("count_8")
        └─ LocalExchange[HASH][$hashvalue]
("regionkey")
           │   Layout: [regionkey:integer, count_8:bigint,
$hashvalue:bigint]
           │   Estimates: {rows: ? (?), cpu: ?, memory: ?,
network: ?}
           └─ RemoteExchange[REPARTITION]
[$hashvalue_9]
              │   Layout: [regionkey:integer, count_8:bigint,
$hashvalue_9:bigint]
              │   Estimates: {rows: ? (?), cpu: ?, memory: ?,
network: ?}
              └─ Aggregate(PARTIAL)[regionkey]
[$hashvalue_10]
                 │   Layout: [regionkey:integer, $hashvalue_10:bigint,
count_8:bigint]
                 │   count_8 := count(*)
                 └─ ScanProject[table =
hive:default:testtable]
                        Layout: [regionkey:integer,
$hashvalue_10:bigint]
                        Estimates: {rows: 0 (0B), cpu: 0, memory: 0B, network: 0B}/{rows: 0 (0B), cpu: 0,
memory: 0B, network: 0B}
                        $hashvalue_10 := "combine_hash"(bigint '0', COALESCE("$operator
$hash_code"("regionkey"), 0))
                        regionkey := regionkey:int:0:REGULAR
```

- DISTRIBUTED:

```
EXPLAIN (type DISTRIBUTED) SELECT regionkey, count(*) FROM testTable GROUP BY 1;
                                    Query Plan
------------------------------------------------------------------------------------------------------------------
 Fragment 0 [SINGLE]
     Output layout: [regionkey, count]
     Output partitioning: SINGLE []
     Stage Execution Strategy:
UNGROUPED_EXECUTION
     Output[regionkey, _col1]
     │   Layout: [regionkey:integer, count:bigint]
     │   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
     │   _col1 := count
     └─ RemoteSource[1]
            Layout: [regionkey:integer, count:bigint]

 Fragment 1 [HASH]
     Output layout: [regionkey, count]
     Output partitioning: SINGLE []
     Stage Execution Strategy:
UNGROUPED_EXECUTION
     Project[]
     │   Layout: [regionkey:integer, count:bigint]
     │   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
     └─ Aggregate(FINAL)[regionkey][$hashvalue]
        │   Layout: [regionkey:integer, $hashvalue:bigint, count:bigint]
        │   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
        │   count := count("count_8")
        └─ LocalExchange[HASH][$hashvalue] ("regionkey")
           │   Layout: [regionkey:integer, count_8:bigint, $hashvalue:bigint]
           │   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
```

```
            └─ RemoteSource[2]
                   Layout: [regionkey:integer, count_8:bigint, $hashvalue_9:bigint]

 Fragment 2 [SOURCE]
     Output layout: [regionkey, count_8, $hashvalue_10]
     Output partitioning: HASH [regionkey][$hashvalue_10]
     Stage Execution Strategy: 
UNGROUPED_EXECUTION
     Aggregate(PARTIAL)[regionkey][$hashvalue_10]
     │   Layout: [regionkey:integer, $hashvalue_10:bigint, count_8:bigint]
     │   count_8 := count(*)
     └─ ScanProject[table = hive:default:testtable, grouped = false]
             Layout: [regionkey:integer, $hashvalue_10:bigint]
             Estimates: {rows: 0 (0B), cpu: 0, memory: 0B, network: 0B}/{rows: 0 (0B), cpu: 0, memory: 0B, 
network: 0B}
             $hashvalue_10 := "combine_hash"(bigint '0', COALESCE("$operator$hash_code"("regionkey"), 
0))
             regionkey := regionkey:int:0:REGULAR
```

- VALIDATE:

```
EXPLAIN (TYPE VALIDATE) SELECT regionkey, count(*) FROM testTable GROUP BY 1;
 Valid 
-------
 true
```

- I/O:

```
EXPLAIN (TYPE IO, FORMAT JSON) SELECT regionkey , count(*) FROM testTable GROUP BY 1;

            Query Plan
---------------------------------
 {
   "inputTableColumnInfos" : [ {
     "table" : {
       "catalog" : "hive",
       "schemaTable" : {
         "schema" : "default",
         "table" : "testtable"
       }
     },
     "columnConstraints" : [ ]
   } ]
 }
```

1.7 Reserved Keywords

Table 1-4 lists the keywords reserved by the system and whether they are reserved in other SQL standards. If you need to use these keywords as identifiers, add double quotation marks.

Table 1-4 Keywords

Keyword	SQL:2016	SQL-92
ALTER	reserved	reserved
AND	reserved	reserved
AS	reserved	reserved
BETWEEN	reserved	reserved
BY	reserved	reserved
CASE	reserved	reserved

Keyword	SQL:2016	SQL-92
CAST	reserved	reserved
CONSTRAINT	reserved	reserved
CREATE	reserved	reserved
CROSS	reserved	reserved
CUBE	reserved	reserved
CURRENT_DATE	reserved	reserved
CURRENT_PATH	reserved	reserved
CURRENT_ROLE	reserved	reserved
CURRENT_TIME	reserved	reserved
CURRENT_TIMESTAMP	reserved	reserved
CURRENT_USER	reserved	reserved
DEALLOCATE	reserved	reserved
DELETE	reserved	reserved
DESCRIBE	reserved	reserved
DISTINCT	reserved	reserved
DROP	reserved	reserved
ELSE	reserved	reserved
END	reserved	reserved
ESCAPE	reserved	reserved
EXCEPT	reserved	reserved
EXECUTE	reserved	reserved
EXISTS	reserved	reserved
EXTRACT	reserved	reserved
FALSE	reserved	reserved
FOR	reserved	reserved
FROM	reserved	reserved
FULL	reserved	reserved
GROUP	reserved	reserved
GROUPING	reserved	reserved
HAVING	reserved	reserved

Keyword	SQL:2016	SQL-92
IN	reserved	reserved
INNER	reserved	reserved
INSERT	reserved	reserved
INTERSECT	reserved	reserved
INTO	reserved	reserved
IS	reserved	reserved
JOIN	reserved	reserved
LEFT	reserved	reserved
LIKE	reserved	reserved
LOCALTIME	reserved	reserved
LOCALTIMESTAMP	reserved	reserved
NATURAL	reserved	reserved
NORMALIZE	reserved	reserved
NOT	reserved	reserved
NULL	reserved	reserved
ON	reserved	reserved
OR	reserved	reserved
ORDER	reserved	reserved
OUTER	reserved	reserved
PREPARE	reserved	reserved
RECURSIVE	reserved	reserved
RIGHT	reserved	reserved
ROLLUP	reserved	reserved
SELECT	reserved	reserved
TABLE	reserved	reserved
THEN	reserved	reserved
TRUE	reserved	reserved
UESCAPE	reserved	reserved
UNION	reserved	reserved
UNNEST	reserved	reserved

Keyword	SQL:2016	SQL-92
USING	reserved	reserved
VALUES	reserved	reserved
WHEN	reserved	reserved
WHERE	reserved	reserved
WITH	reserved	reserved

1.8 SQL Functions and Operators

1.8.1 Logical Operators

Logical Operators

Operation	Description	Example
AND	If both values are **true**, the value is **true**.	a AND b
OR	If one of the two values is **true**, the value is **true**.	a OR b
NOT	If the value is **false**, the result is **true**.	NOT a

The following truth table reflects how AND and OR handle NULL values.

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	TRUE	NULL	TRUE
NULL	FALSE	FALSE	NULL
NULL	NULL	NULL	NULL

The following truth table reflects how NOT handle NULL values.

value	NOT value
TRUE	FALSE
FALSE	TRUE
NULL	NULL

1.8.2 Comparison Functions and Operators

Comparison

Operation	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal to
<>	Not equal to
!=	Not equal to

- Scope comparison: **between**

  between is applicable to values in a specified range, for example, *value* **BETWEEN** *min* **AND** *max*.

  Not between is used when the value is not in a specified range.

  The **null** value cannot be used in the **between** operation. The execution results of the following two operations are **Null**:

  ```
  SELECT NULL BETWEEN 2 AND 4; -- null
  SELECT 2 BETWEEN NULL AND 6; -- null
  ```

  In HetuEngine, the *value*, *min*, and *max* parameters must be of the same data type in **BETWEEN** and **NOT BETWEEN**.

  Incorrect usage: **'John' between 2.3 and 35.2**

  Example expression equivalent to BETWEEN:

  ```
  SELECT 3 BETWEEN 2 AND 6; -- true
  SELECT 3 >= 2 AND 3 <= 6; -- true
  ```

  Example expression equivalent to NOT BETWEEN:

  ```
  SELECT 3 NOT BETWEEN 2 AND 6; -- false
  SELECT 3 < 2 OR 3 > 6; -- false
  ```

- **IS NULL** and **IS NOT NULL**

  They are used to determine whether a value is empty. All data types can be used for this determination.

```
SELECT 3.0 IS NULL; -- false
```

- **IS DISTINCT FROM** and **IS NOT DISTINCT FROM**

  This is a special usage. In HetuEngine SQL statements, NULL indicates an unknown value. All comparisons related to NULL also produce NULL results. **IS DISTINCT FROM** and **IS NOT DISTINCT FROM** can take a null value as a known value and return **true** or **false** (even if the expression contains a null value).

  Example:

```
--Create a table.
create table dis_tab(col int);
--Insert data.
insert into dis_tab values (2),(3),(5),(null);
--Query:
select col from dis_tab where col is distinct from null;
 col
----
 2
 3
 5
(3 rows)
```

  The following truth table demonstrates how IS DISTINCT FROM and IS NOT DISTINCT FROM process common data and NULL values.

a	b	a = b	a <> b	a DISTINCT b	a NOT DISTINCT b
1	1	TRUE	FALSE	FALSE	TRUE
1	2	FALSE	TRUE	TRUE	FALSE
1	NULL	NULL	NULL	TRUE	FALSE
NULL	NULL	NULL	NULL	FALSE	TRUE

- **GREATEST** and **LEAST**

  The two functions are not standard SQL functions. They are typical extensions. The parameter cannot contain **null** values.

  – greatest(value1, value2, ..., valueN)

    Returns the provided maximum value.

  – least(value1, value2, ..., valueN) → [same as input]

    Returns the provided minimum value.

- Batch comparison: **ALL**, **ANY**, and **SOME**

  Quantifiers ALL, ANY, and SOME can be used together with comparison operators in the following ways:

```
expression operator quantifier ( subquery )
```

  The meanings of some combinations of quantifiers and comparison operators are as follows. ANY and SOME have the same meaning. You can replace ANY with SOME when using the expressions in the following table.

Expression	Definition
A = ALL (...)	Returns **true** when A is equal to all values.
A <> ALL (...)	Returns **true** when A is not equal to any value.
A < ALL (...)	Returns **true** when A is less than the minimum value.
A = ANY (...)	Returns **true** when A is the same as any value, which is equivalent to A IN (...).
A <> ANY (...)	Returns **true** when A is different from any value.
A < ANY (...)	Returns **true** when A is less than the maximum value.

Example:

```
SELECT 'hello' = ANY (VALUES 'hello', 'world'); -- true
SELECT 21 < ALL (VALUES 19, 20, 21); -- false
SELECT 42 >= SOME (SELECT 41 UNION ALL SELECT 42 UNION ALL SELECT 43);-- true
```

1.8.3 Condition Expression

CASE

Standard SQL CASE expressions have two modes.

- In simple mode, search for each value of the expression from left to right until the same expression is found.

  CASE expression

  WHEN value THEN result

  [WHEN ...]

  [ELSE result]

  END

  Returns the result that matches the value. If no value is matched, the result of the **ELSE** clause is returned. If there is no **ELSE** clause, **null** is returned.
  Example:

```
select a,
case a
 when 1 then 'one'
 when 2 then 'two'
 else 'many' end from
 (values (1),(2),(3),(4)) as t(a);
 a | _col1
---|-------
 1 | one
 2 | two
 3 | many
 4 | many
(4 rows)
```

- In search mode, the system checks the Boolean value of each condition from left to right until the value is true and returns the matching result.

  CASE

  WHEN condition THEN result

  [WHEN ...]

  [ELSE result] END

  If none of the conditions is met, the result of the **ELSE** clause is returned. If there is no **ELSE** clause, null is returned. Example:

```
select a,b,
case
when a=1 then 'one'
when b=2 then 'tow'
else 'many' end from (values (1,2),(3,4),(1,3),(4,2)) as t(a,b);
 a | b | _col2
---|---|-------
 1 | 2 | one
 3 | 4 | many
 1 | 3 | one
 4 | 2 | tow
(4 rows)
```

IF

The IF function is a language structure. It has the same function as the following CASE expression:

CASE

WHEN condition THEN true_value

[ELSE false_value] END

- if(condition, true_value)

  If *condition* is **true**, *true_value* is returned. Otherwise, **null** is returned and *true_value* is not calculated.

```
select if(a=1,8) from (values (1),(1),(2)) as t(a); -- 8 8 NULL
select if(a=1,'value') from (values (1),(1),(2)) as t(a); -- value value NULL
```

- if(condition, true_value, false_value)

  If *condition* is **true**, *true_value* is returned. Otherwise, *false_value* is returned.

```
select if(a=1,'on','off') from (values (1),(1),(2)) as t(a);
_col0
-------
 on
 on
 off
(3 rows)
```

COALESCE

coalesce(value[, ...])

Returns the first non-null value in the parameter list. Similar to CASE expressions, parameters are calculated only when necessary.

It is similar to the nvl function of MySQL and is often used to convert a null value to 0 or ' ' (null character).

```
select coalesce(a,0) from (values (2),(3),(null)) as t(a); -- 2 3 0
```

NULLIF

- nullif(value1, value2)

  If *value1* is equal to *value2*, **null** is returned. Otherwise, *value1* is returned.

  ```
  select nullif(a,b) from (values (1,1),(1,2)) as t(a,b); --
   _col0
  -------
    NULL
      1
  (2 rows)
  ```

- ZEROIFNULL(value)

  If the value is **null**, **0** is returned. Otherwise, the original value is returned. Currently, the varchar type is also supported.

  ```
  select zeroifnull(a),zeroifnull(b),zeroifnull(c) from (values (null,13.11,bigint '157'),(88,null,bigint '188'),
  (55,14.11,null)) as t(a,b,c);
   _col0 | _col1 | _col2
  -------|-------|-------
       0 | 13.11 |   157
      88 |  0.00 |   188
      55 | 14.11 |     0
  (3 rows)
  ```

- NVL(value1,value2)

  If *value1* is **null**, *value2* is returned. Otherwise, *value1* is returned.

  ```
  select nvl(NULL,3);  -- 3
  select nvl(2,3);    --2
  ```

- ISNULL(value)

  If *value1* is **null**, **true** is returned. Otherwise, **false** is returned.

  ```
  Create table nulltest(col1 int,col2 int);
  insert into nulltest values(null,3);
  select isnull(col1),isnull(col2) from nulltest;
   _col0 | _col1
  -------|-------
   true  | false
  (1 row)
  ```

- ISNOTNULL(value)

  If *value1* is **null**, **false** is returned. Otherwise, **true** is returned.

  ```
  select isnotnull(col1),isnotnull(col2) from nulltest;
   _col0 | _col1
  -------|-------
   false | true
  (1 row)
  ```

TRY

Evaluates an expression. If an error occurs, **Null** is returned. It is similar to **try catch** in the programming language. The **TRY** function is generally used together with **COALESCE**. **COALESCE** can convert an abnormal null value to **0** or **null**. The following situations will be captured by the **TRY** function:

- The denominator is 0.
- The cast operation or function input parameter is incorrect.
- The number exceeds the defined length.

This method is not recommended. Specify the preceding exceptions and preprocess data.

Example:

Assume that the **origin_zip** field in the following table contains invalid data:

```
--Create a table.
create  table shipping (origin_state varchar,origin_zip varchar,packages int ,total_cost int);

--Insert data.
insert into shipping
values
('California','94131',25,100),
('California','P332a',5,72),
('California','94025',0,155),
('New Jersey','08544',225,490);

--Query data.
SELECT * FROM shipping;
 origin_state | origin_zip | packages | total_cost
 --------------+------------+----------+------------
 California   |      94131 |       25 |        100
 California   |      P332a |        5 |         72
 California   |      94025 |        0 |        155
 New Jersey   |      08544 |      225 |        490
(4 rows)
```

The query fails when **TRY** is not used:

```
SELECT CAST(origin_zip AS BIGINT) FROM shipping;
Query failed: Cannot cast 'P332a' to BIGINT
```

When **TRY** is used, **null** is returned:

```
SELECT TRY(CAST(origin_zip AS BIGINT)) FROM shipping;
 origin_zip
 ------------
     94131
     NULL
     94025
     08544
 (4 rows)
```

The query fails when **TRY** is not used:

```
SELECT total_cost/packages AS per_package FROM shipping;
Query failed: Division by zero
```

The default values are returned when **TRY** and **COALESCE** are used.

```
SELECT COALESCE(TRY(total_cost/packages),0) AS per_package FROM shipping;
 per_package
 -------------
    4
    14
    0
    19
 (4 rows)
```

1.8.4 Lambda Expression

The Lambda expression can be represented by ->.

```
x->x+1
(x,y)->x+y
x->regexp_like(x,'a+')
x->x[1]/x[2]
x->IF(x>0,x,-x)
x->COALESCE(x,0)
x->CAST(xASJSON)
x->x+TRY(1/0)
```

Most SQL expressions can be used in the Lambda function body except in the following scenarios:

- Subqueries are not supported.
```
x -> 2 + (SELECT 3)
```
- Aggregate functions are not supported.
```
x -> max(y)
```

Examples

- Use the **transform()** function to obtain the square of array elements.
```
SELECT numbers, transform(numbers, n -> n * n) as squared_numbers FROM (VALUES (ARRAY[1, 2]),
(ARRAY[3, 4]),(ARRAY[5, 6, 7])) AS t(numbers);
  numbers  | squared_numbers
-----------|-----------------
 [1, 2]    | [1, 4]
 [3, 4]    | [9, 16]
 [5, 6, 7] | [25, 36, 49]
(3 rows)
```
- Use the **transform()** function to convert array elements into strings. If the conversion fails, the array elements will be converted into NULL to avoid errors.
```
SELECT transform(prices, n -> TRY_CAST(n AS VARCHAR) || '$') as price_tags FROM (VALUES
(ARRAY[100, 200]),(ARRAY[30, 4])) AS t(prices);
  price_tags
--------------
 [100$, 200$]
 [30$, 4$]
(2 rows)
```
- When an operation is performed on an array element, other columns can also be used in the operation. For example, use **transform()** to calculate the linear equation **f(x) =ax + b**.
```
SELECT xvalues, a, b, transform(xvalues, x -> a * x + b) as linear_function_values FROM (VALUES
(ARRAY[1, 2], 10, 5), (ARRAY[3, 4], 4, 2)) AS t(xvalues, a, b);
 xvalues | a  | b | linear_function_values
---------|----|---|------------------------
 [1, 2]  | 10 | 5 | [15, 25]
 [3, 4]  |  4 | 2 | [14, 18]
(2 rows)
```
- Use **any_match()** to search for an array where at least one element is greater than 100.
```
SELECT numbers FROM (VALUES (ARRAY[1,NULL,3]), (ARRAY[10,200,30]), (ARRAY[100,20,300])) AS
t(numbers) WHERE any_match(numbers, n ->  COALESCE(n, 0) > 100);
    numbers
----------------
 [10, 200, 30]
 [100, 20, 300]
(2 rows)
```
- Use **regexp_replace()** to capitalize the first letter.
```
SELECT regexp_replace('once upon a time ...', '^(\w)(\w*)(\s+.*)$',x -> upper(x[1]) || x[2] || x[3]); --
Once upon a time ...
```
- Use the Lambda expression in the aggregate function. For example, use **reduce_agg()** to calculate the sum of elements by column.
```
SELECT reduce_agg(value, 0, (a, b) -> a + b, (a, b) -> a + b) sum_values FROM (VALUES (1), (2), (3),
(4), (5)) AS t(value);
 sum_values
------------
         15
(1 row)
```

1.8.5 Conversion Functions

cast Conversion Function

HetuEngine implicitly converts numeric and character values to the correct type. HetuEngine does not convert between character and numeric types. For example, if a query expects a value of the varchar type, HetuEngine does not automatically convert a value of the bigint type to a value of the varchar type.

Values can be explicitly converted to the specified type, if necessary.

- cast(value AS type) → type

  Explicitly converts the type of a value. You can convert a value of the varchar type to the numeric type, or vice versa.

  ```
  select cast('186' as int );
  select cast(186 as varchar);
  ```

- try_cast(value AS type) → type

  It is similar to **cast()**. The difference is that null is returned if the conversion fails.

  ```
  select try_cast(1860 as tinyint);
   _col0
  -------
    NULL
  (1 row)
  ```

NOTE

When a number overflows or a null value is converted, **null** is returned. However, when the conversion fails, an error is reported.

Example: select try_cast(186 as date);

Cannot cast integer to date

Format

- format(format, args...) → varchar

  Description: Formats a string in the format specified by the format string and returns the formatted string.

  ```
  SELECT format('%s%%',123);-- '123%'
  SELECT format('%.5f',pi());-- '3.14159'
  SELECT format('%03d',8);-- '008'
  SELECT format('%,.2f',1234567.89);-- '1,234,567.89'
  SELECT format('%-7s,%7s','hello','world');-- 'hello  ,  world'
  SELECT format('%2$s %3$s %1$s','a','b','c');-- 'b c a'
  SELECT format('%1$tA, %1$tB %1$te, %1$tY',date'2006-07-04');-- 'Tuesday, July 4, 2006
  ```

- format_number(number) → varchar

  Description: Returns a string formatted by unit symbol.

  ```
  SELECT format_number(123456); -- '123K'
  SELECT format_number(1000000); -- '1M'
  ```

Data Size

The **parse_presto_data_size** function supports the following units:

Unit	Description	Value
B	Bytes	1
kB	Kilobytes	1024
MB	Megabytes	1024^2
GB	Gigabytes	1024^3
TB	Terabytes	1024^4
PB	Petabytes	1024^5
EB	Exabytes	1024^6
ZB	Zettabytes	1024^7
YB	Yottabytes	1024^8

parse_presto_data_size(string) → decimal(38)

Convert formatted values with a unit to a number. The value can be a decimal.

```
SELECT parse_presto_data_size('1B'); -- 1
SELECT parse_presto_data_size('1kB'); -- 1024
SELECT parse_presto_data_size('1MB'); -- 1048576
SELECT parse_presto_data_size('2.3MB'); -- 2411724
```

Others

typeof(expr) → varchar

Returns the data type name of an expression.

```
SELECT typeof(123);-- integer
SELECT typeof('cat');-- varchar(3)
SELECT typeof(cos(2)+1.5);-- double
```

1.8.6 Mathematical Functions and Operators

Mathematical Operator

Operator	Description
+	Add
-	Deduct
*	Multiple
/	Divide
%	Remainder

Mathematical Functions

- abs(x) → [same as input]

  Returns the absolute value of *x*.

  ```
  SELECT abs(-17.4);-- 17.4
  ```

- bin(bigint x) -> string

  Returns *x* in binary format.

  ```
  select bin(5); --101
  ```

- bround(double x) -> double

  Banker's rounding:

  – 1 to 4: rounding down
  – 6 to 9: rounding up
  – The number before 5 is even: rounding down
  – The number before 5 is odd: rounding up

  ```
  select bround(3.5); -- 4.0
  select bround(2.5); -- 2.0
  select bround(3.4); -- 3.0
  ```

- bround(double x, int y) -> double

  Banker's rounding with *y* decimal places reserved.

  ```
  select bround(8.35,1); --8.4
  select bround(8.355,2); --8.36
  ```

- ceil(x) → [same as input]

  Same as **ceiling()**

  ```
  SELECT ceil(-42.8); -- -42
  ```

  ceiling(x) → [same as input]

  Returns the rounded-up value of *x*.

  ```
  SELECT ceiling(-42.8); -- -42
  ```

- conv(bigint num, int from_base, int to_base)
- conv(string num, int from_base, int to_base)

  Converts **num**, for example, from decimal to binary.

  ```
  select conv('123',10,2); -- 1111011
  ```

- rand() → double

  Returns a random decimal number between 0 and 1.

  ```
  select rand();--  0.049510824616263105
  ```

- cbrt(x) → double

  Returns the cube root of *x*.

  ```
  SELECT cbrt(27.0); -- 3
  ```

- e() → double

  Returns the Euler constant.

  ```
  select e();-- 2.718281828459045
  ```

- exp(x) → double

  Returns the value of *e* raised to the power of *x*.

  ```
  select exp(1);--2.718281828459045
  ```

- factorial(int x) -> bigint

  Returns the factorial of *x*. The value range of *x* is [0, 20].

```
select factorial(4); --24
```

- floor(x) → [same as input]

  Returns the nearest integer rounded off from *x*.

  ```
  SELECT floor(-42.8);-- -43
  ```

- from_base(string, radix) → bigint

  Converts a specified number system to bigint. For example, converts the ternary number 200 to a decimal number.

  ```
  select from_base('200',3);--18
  ```

- hex(bigint|string|binary x) -> string

  Returns a hexadecimal number as a string if *x* is of the int or binary type. If *x* is a string, converts each character of the string to a hexadecimal representation and returns a string.

  ```
  select hex(68); -- 44
  select hex('AE'); -- 4145
  ```

- to_base(*x*, *radix*) → varchar

  Converts an integer into a character string in the radix system. For example, converts the decimal number 18 to a ternary number.

  ```
  select to_base(18,3);-- 200
  ```

- ln(x) → double

  Returns the natural logarithm of *x*.

  ```
  select ln(10);--2.302585092994046
  select ln(e());--1.0
  ```

- log2(x) → double

  Returns the logarithm of *x* to base 2.

  ```
  select log2(4);-- 2.0
  ```

- log10(x) → double

  Returns the logarithm of *x* to base 10.

  ```
  select log10(1000);-- 3.0
  ```

- log(b, x) → double

  Returns the logarithm of *x* to base *b*.

  ```
  select log(3,81); -- 4.0
  ```

- mod(n, m) → [same as input]

  Returns the modulus of *n* divided by *m*.

  ```
  select mod(40,7) ;-- 5
  select mod(-40,7);  -- -5
  ```

- pi() → double

  Returns pi.

  ```
  select pi();--3.141592653589793
  ```

- pmod(int x,int y) -> int
- pmod(double x,double y) -> double

  Returns the positive value of the remainder after division of **x** by **y**.

  ```
  select pmod(8,3); --2
  Select pmod(8.35,2.0); --0.35
  ```

- pow(x, p) → double

  Same as **power()**.

  ```
  select pow(3.2,3);-- 32.76800000000001
  ```

- power(x,p)

  Returns the value of *x* raised to the power of *p*.

  ```
  select power(3.2,3);-- 32.76800000000001
  ```

- radians(x) → double

  Converts the angle *x* to a radian.

  ```
   select radians(57.29577951308232);-- 1.0
  ```

- degrees(x) → double

  Converts an angle *x* (represented by a radian) into an angle.

  ```
  select degrees(1);-- 57.29577951308232
  ```

- round(*x*) → [same as input]

  Return the integer that is rounded to the nearest integer of *x*.

  ```
  select round(8.57);-- 9
  ```

- round(*x*, *d*) → [same as input]

  x is rounded off to *d* decimal places.

  ```
  select round(8.57,1);-- 8.60
  ```

- shiftleft(tinyint|smallint|int x, int y) -> int
- shiftleft(bigint x, int y) -> bigint

  Returns the value of *x* shifted leftwards by *y* positions.

  ```
  select shiftleft(8,2);--32
  ```

- shiftright(tinyint|smallint|int a, int b) -> int
- shiftright(bigint a, int b) -> bigint

  Returns the value of *x* shifted rightwards by *y* positions.

  ```
  select shiftright(8,2);--2
  ```

- shiftrightunsigned(tinyint|smallint|int x, int y) -> int
- shiftrightunsigned(bigint x, int y) -> bigint

  Shifts to the right by bit without symbols, and returns the value of *x* shifted rightwards by *y* positions. Returns an int if *x* is tinyint, smallint, or int. Returns a bigint if *x* is bigint.

  ```
  select shiftrightunsigned(8,3); -- 1
  ```

- sign(*x*) → [same as input]

  Returns the symbol function of x.

  – If *x* is equal to **0**, **0** is returned.

  – If *x* is less than **0**, the value **–1** is returned.

  – If *x* is greater than **0**, **1** is returned.

  ```
  select sign(-32.133);-- -1
  select sign(32.133); -- 1
  select sign(0);--0
  ```

  For parameters of the double type:

  – If the parameter is NaN, **NaN** is returned.

  – If the parameter is +∞, **1** is returned.

  – If the parameter is -∞, **-1** is returned.

  ```
  select sign(NaN());--NaN
  select sign(Infinity());-- 1.0
  select  sign(-infinity());-- -1.0
  ```

- sqrt(*x*) → double

  Returns the square root of *x*.

  ```
  select sqrt(100); -- 10.0
  ```

- truncate(*number,num_digits*)
  - *Number* indicates the number to be truncated, and *Num_digits* indicates the decimal places retained.
  - The default value of *Num_digits* is **0**.
  - The **truncate()** function does not round off the result.

  ```
  select truncate(10.526); -- 10
  select truncate(10.526,2); --  10.520
  ```

- trunc(*number,num_digits*)

  See truncate(*number,num_digits*).

- unhex(string x) -> binary

  Returns the reciprocal of a hexadecimal number.

  ```
  select unhex('123'); --^A#
  ```

- width_bucket(*x*, *bound1*, *bound2*, *n*) → bigint

  Returns the number of containers x in the equi-width histogram with the specified **bound1** and **bound2** boundaries and *n* buckets.

  ```
  select value,width_bucket(value,1,5000,10) from (values (1),(100),(500),(1000),(2000),(2500),(3000),
  (4000),(4500),(5000),(8000)) as t(value);
  value | _col1
  -------|-------
      1 |     1
    100 |     1
    500 |     1
   1000 |     2
   2000 |     4
   2500 |     5
   3000 |     6
   4000 |     8
   4500 |     9
   5000 |    11
   8000 |    11
  (11 rows)
  ```

- width_bucket(*x*, *bins*) → bigint

  Returns the number of bins of x based on the bin specified by the array **bin**. The **bins** parameter must be a double-precision array and is assumed to be in ascending order.

  ```
  select width_bucket(x,array [1.00,2.89,3.33,4.56,5.87,15.44,20.78,30.77]) from (values (3),(4)) as t(x);
   _col0
  -------
      2
      3
  (2 rows)
  ```

- quotient(BIGINT numerator, BIGINT denominator)→bigint

  Returns the value of the left number divided by the right number. Part of the decimal part is discarded.

  ```
  select quotient(25,4);-- 6
  ```

Random

- rand() → double

  Same as **random()**

- random() → double

  Returns a pseudo-random value in the range of 0.0 <= x < 1.0.

  ```
  select random();-- 0.021847965885988363
  select random();-- 0.5894438037549372
  ```

- random(*n*) → [same as input]

  Returns a pseudo-random number between 0 and n (excluding n).

  ```
  select random(5);-- 2
  ```

NOTICE

random(n) contains the following data types: tinyint, bigint, smallint and integer.

Statistical Function

The binomial distribution confidence interval has multiple calculation formulas, and the most common one is ["normal interval"]. However, it is applicable only to a case in which there are a relatively large quantity of samples (np > 5 and n(1 p) > 5). For a small sample, the accuracy is poor. To solve this problem, the Wilson Score Interval is used.

$$\left(\hat{p}+\frac{z_{\alpha/2}^{2}}{2n} \pm z_{\alpha/2}\sqrt{\left[\hat{p}(1-\hat{p})+z_{\alpha/2}^{2}/4n\right]/n}\right)/(1+z_{\alpha/2}^{2}/n).$$

z —— normal distribution, average value + z x standard deviation confidence. z = 1.96, confidence level: 95%

Take, for example, the collecting of positive rate. **pos** indicates the number of positive reviews; **n** indicates the total number of reviews; and **phat** indicates the positive review rate.

z = 1.96

phat= 1.0* pos/n

z1=phat + z * z/(2 * n)

z2 = $z * \sqrt{\text{paht}(1-\text{phat})/n + z^{2}/(4 * n^{2})}$

m = (1 + z * z/n)

Lower limit **(z1-z2)/m**, upper limit **(z1+z2)/m**

- wilson_interval_lower(successes, trials, z) → double

  Returns the lower bound of the Wilson score interval for the Bernoulli test process. The confidence value is specified by the z-score **z**.

  ```
  select wilson_interval_lower(1, 5, 1.96);-- 0.036223160969787456
  ```

- wilson_interval_upper(successes, trials, z) → double

  Returns the upper bound of the Wilson score interval for the Bernoulli test process. The confidence value is specified by the z-score **z**.

  ```
   select wilson_interval_upper(1, 5, 1.96);--  0.6244717358814612
  ```

- cosine_similarity(*x*, *y*) → double

  Returns the cosine similarity between sparse vectors *x* and *y*.

  ```
  SELECT cosine_similarity (MAP(ARRAY['a'],ARRAY[1.0]),MAP(ARRAY['a'],ARRAY[2.0]));-- 1.0
  ```

Cumulative Distribution Function

- beta_cdf(*a*, *b*, *v*) → double

Use the given **a** and **b** parameters to calculate the cumulative distribution function (P (N <v; a, b)) of the beta distribution. Parameters **a** and **b** must be positive real numbers, and the value **v** must be a real number. The value **v** must be within the interval [0, 1].

A cumulative distribution function formula of beta distribution is also referred to as an incomplete beta function ratio (which is usually represented by **Ix**), and corresponds to the following formula:

$$F(x) = I_x(p,q) = \frac{\int_0^x t^{p-1}(1-t)^{q-1}dt}{B(p,q)} \quad 0 \leq x \leq 1; p,q > 0$$

```
select beta_cdf(3,4,0.0004); --  1.2788483685990 41E-9
```

- inverse_beta_cdf(*a*, *b*, *p*) → double

  The inverse operation of the beta cumulative distribution function, given the **a** and **b** parameters of the cumulative probability p: P (N < n). Parameters **a** and **b** must be positive real numbers, and **p** must be within the range of [0,1].

  ```
  select inverse_beta_cdf(2, 5, 0.95) ;--0.5818034093775719
  ```

- inverse_normal_cdf(*mean*, *sd*, *p*) → double

  Given the cumulative probability (p): P (N < n) related mean and standard deviation, calculate the inverse of the normal cumulative distribution function. The average value must be a real value, and the standard deviation must be a positive real value. The probability **p** must be in the interval (0, 1).

  ```
  select inverse_normal_cdf(2, 5, 0.95);-- 10.224268134757361
  ```

- normal_cdf(*mean*, *sd*, *v*) → double

  Calculate the value of the normal distribution function based on the average value and standard deviation. P(N<v; mean,sd). The average value and **v** must be real values, and the standard deviation must be positive real values.

  ```
   select normal_cdf(2, 5, 0.95);--  0.41683383651 75577
  ```

Trigonometric Function

The parameters of all trigonometric functions are expressed in radians. Refer to the unit conversion functions **degrees()** and **radians()**.

- acos(x) → double

  Calculates the arc cosine value.

  ```
  SELECT acos(-1);-- 3.14159265358979
  ```

- asin(x) → double

  Calculates the arc sine value.

  ```
  SELECT asin(0.5);-- 0.5235987755982989
  ```

- atan(x) → double

  Returns the arc tangent value of *x*.

  ```
  SELECT atan(1);-- 0.7853981633974483
  ```

- atan2(y, x) → double

  Return the arc tangent value of *y/x*.

  ```
  SELECT atan2(2,1);-- 1.1071487177940904
  ```

- cos(x) → double

  Returns the cosine value of *x*.

  ```
  SELECT cos(-3.1415927);-- -0.9999999999999989
  ```

- cosh(x) → double

  Returns the hyperbolic cosine value of *x*.

  ```
  SELECT cosh(3.1415967);-- 11.592000006553231
  ```

- sin(x) → double

  Returns the sine value of *x*.

  ```
  SELECT sin(1.57079);--  0.9999999999799858
  ```

- tan(x) → double

  Returns the tangent value of *x*.

  ```
  SELECT tan(20);-- 2.23716094422474
  ```

- tanh(x) → double

  Returns the hyperbolic tangent value of *x*.

  ```
  select tanh(3.1415927);-- 0.9962720765661324
  ```

Floating-Point Function

- infinity() → double

  Returns a constant representing positive infinity.

  ```
  select infinity();-- Infinity
  ```

- is_finite(x) → boolean

  Checks whether *x* is a finite value.

  ```
  select is_finite(infinity());-- false
  select is_finite(50000);--true
  ```

- is_infinite(x) → boolean

  Determines whether *x* is infinite.

  ```
  select is_infinite(infinity());-- true
  select is_infinite(50000);--false
  ```

- is_nan(x) → boolean

  Checks whether *x* is a non-digit character.

  ```
  -- The input value must be of the double type.
  select is_nan(null); -- NULL
  select is_nan(nan()); -- true
  select is_nan(45);-- false
  ```

- nan() → double

  Returns a constant representing a non-numeric number.

  ```
  select nan(); -- NaN
  ```

1.8.7 Bitwise Functions

- bit_count(x, bits) → bigint

  Calculate the number of bits set in x (regarded as an integer with a signed bit) in the complementary code representation of 2.

  ```
  SELECT bit_count(9, 64); -- 2
  SELECT bit_count(9, 8); -- 2
  SELECT bit_count(-7, 64); -- 62
  SELECT bit_count(-7, 8); -- 6
  ```

- bitwise_and(x, y) → bigint

  Returns the bitwise AND result of x and y in binary complement.

  ```
  select bitwise_and(8, 7); -- 0
  ```

- bitwise_not(x) → bigint

  Returns the result of x bitwise NOT in binary complement.

  ```
  select bitwise_not(8);-- -9
  ```

- bitwise_or(x, y) → bigint

  Returns the bitwise OR result of x and y in binary complement.

  ```
  select bitwise_or(8,7);-- 15
  ```

- bitwise_xor(x, y) → bigint

  Returns the bitwise XOR result of x and y in binary complementary code format.

  ```
  SELECT bitwise_xor(19,25); -- 10
  ```

- bitwise_left_shift(*value*, *shift*) → [same as value]

  Description: Returns the value that is shifted leftwards by **shift** bits.

  ```
  SELECT bitwise_left_shift(1, 2); -- 4
  SELECT bitwise_left_shift(5, 2); -- 20
  SELECT bitwise_left_shift(0, 1); -- 0
  SELECT bitwise_left_shift(20, 0); -- 20
  ```

- bitwise_right_shift(value, shift) → [same as value]

  Description: Returns the value that is shifted rightwards by **shift** bits.

  ```
  SELECT bitwise_right_shift(8, 3); -- 1
  SELECT bitwise_right_shift(9, 1); -- 4
  SELECT bitwise_right_shift(20, 0); -- 20
  SELECT bitwise_right_shift(0, 1); -- 0
  -- If the value is shifted rightwards by more than 64 bits, return 0.
  SELECT bitwise_right_shift( 12, 64); -- 0
  ```

- bitwise_right_shift_arithmetic(*value*, *shift*) → [same as value]

  Description: Returns the value that is arithmetically right shifted. When shift is less than 64 bits, the return value is the same as the value returned by **bitwise_right_shift**. When **shift** reaches or exceeds 64 bits, this function returns **0** if the value is a positive number, and returns **-1** if the value is a negative number.

  ```
  SELECT bitwise_right_shift_arithmetic( 12, 64); --  0
  SELECT bitwise_right_shift_arithmetic(-45, 64); -- -1
  ```

1.8.8 Decimal Functions and Operators

DECIMAL Literal

You can use the DECIMAL'xxxxxxx.yyyyyyy' syntax to define literals of the DECIMAL type.

The literal precision of the DECIMAL type will be equal to the number of bits of the literal (including trailing zeros and leading zeros). The range will be equal to the number of digits in the decimal part (including trailing zeros).

Example Literal	Data Type
DECIMAL '0'	DECIMAL(1)
DECIMAL '12345'	DECIMAL(5)
DECIMAL '0000012345.1234500000'	DECIMAL(20, 10)

Binary Arithmetic Decimal Operator

Standard mathematical operators are supported. The following table describes the rules for calculating the precision and range of the results. Assume that the type of x is DECIMAL(xp, xs) and the type of y is DECIMAL(yp, ys).

Calculation	Result Type Precision	Result Type Range
x + y and x - y	min(38, 1 + min(xs, ys) + min(xp - xs, yp - ys))	max(xs, ys)
x * y	min(38, xp + yp)	xs + ys
x / y	min(38, xp + ys + max(0, ys-xs))	max(xs, ys)
x % y	min(xp - xs, yp - ys) + max(xs, bs)	max(xs, ys)

If the mathematical result of the operation cannot be accurately represented by the precision and range of the result data type, the exception occurs: Value is out of range.

When an operation is performed on a decimal type with different ranges and precisions, the value is first cast to a common supertype. For types close to the maximum representable precision (38), this may cause a "value out of range" error when an operand does not conform to the public supertype. For example, the common supertype of decimal (38, 0) and decimal (38, 1) is decimal (38, 1), but some values that comply with decimal (38, 0) cannot be represented as decimal (38, 1).

Comparison Operators

All standard comparison operators and BETWEEN operators apply to DECIMAL types.

Unary Decimal Operators

The operator "-" performs a negative operation. The type of the result is the same as that of the parameter.

1.8.9 String Functions and Operators

String Operators

|| Indicates the character connection.

```
SELECT 'he'||'llo'; --hello
```

String Functions

These functions assume that the input string contains valid UTF-8 encoded Unicode code points. They do not explicitly check whether UTF-8 data is valid. For invalid UTF-8 data, the function may return an incorrect result. You can use from_utf8 to correct invalid UTF-8 data.

In addition, these functions operate on Unicode code points, not on characters (or font clusters) visible to users. Some languages combine multiple code points into a single user-perceived character (which is the basic unit of the language writing system), but functions treat each code point as a separate unit.

The lower and upper functions do not perform locale-related, context-related, or one-to-many mappings required by certain languages.

- chr(n) → varchar

  Description: Returns the value of a character whose Unicode encoding value is **n**.

  ```
  select chr(100); --d
  ```

- char_length(string) → bigint

  For details, see **length(string)**.

- character_length(string) → bigint

  For details, see **length(string)**.

- codepoint(*string*) → integer

  Description: Returns the Unicode encoding of a single character.

  ```
  select codepoint('d'); --100
  ```

- concat(string1, string2) → varchar

  Description: Concatenates strings.

  ```
  select concat('hello','world'); -- helloworld
  ```

- concat_ws(string0, string1, ..., stringN) → varchar

  Description: Concatenates string1, string2, ..., and stringN into a string using string0 as the separator. If string0 is null, the return value is null. If the parameter following the separator is null, the parameter will be skipped during concatenation.

  ```
  select concat_ws(',','hello','world'); -- hello,world
  select concat_ws(NULL,'def'); --NULL
  select concat_ws(',','hello',NULL,'world'); -- hello,world
  select concat_ws(',','hello','','world'); -- hello,,world
  ```

- concat_ws(string0, array(varchar)) → varchar

  Description: Concatenates elements in an array using string0 as the separator. If string0 is null, the return value is null. Any null value in the array will be skipped.

```
select concat_ws(NULL,ARRAY['abc']);--NULL
select concat_ws(',',ARRAY['abc',NULL,NULL,'xyz']); -- abc,xyz
select concat_ws(',',ARRAY['hello','world']); -- hello,world
```

- decode(binary bin, string charset) →varchar

  Description: Encodes the first parameter into a string based on the specified character set. The supported character sets include UTF-8, UTF-16BE, UTF-16LE, and UTF-16. If the first parameter is null, null is returned.

```
select decode(X'70 61 6e 64 61','UTF-8');
 _col0
-------
 panda
(1 row)

select decode(X'00 70 00 61 00 6e 00 64 00 61','UTF-16BE');
 _col0
-------
 panda
(1 row)
```

- encode(string str, string charset) →binary

  Description: Encodes a string based on the specified character set.

```
select encode('panda','UTF-8');
     _col0
----------------
 70 61 6e 64 61
(1 row)
```

- find_in_set (string str, string strList) →int

  Description: Returns the position of the first occurrence of the string in the comma-separated strList. If a parameter is null, null is returned.

```
select find_in_set('ab', 'abc,b,ab,c,def'); -- 3
```

- format_number(number x, int d) →string

  Description: Formats the number *x* to **#,###,###.##**, reserves *d* decimal places, and returns the result as a string.

```
select format_number(541211.212,2); -- 541,211.21
```

- format(format,args...) → varchar

  Description: For details, see **Format**.

- locate(string substr, string str, int pos]) →int

  Description: Returns the position of the first occurrence of the substring after the *pos* position in the string. If the condition is not met, **0** is returned.

```
select locate('aaa','bbaaaaa',6);-- 0
select locate('aaa','bbaaaaa',1);-- 3
select locate('aaa','bbaaaaa',4);-- 4
```

- length(string) → bigint

  Description: Returns the length of the string.

```
select length('hello');-- 5
```

- levenshtein_distance(string1, string2) → bigint

  Description: Calculates the Levenshtein distance between string1 and string2, that is, the minimum number of single-character edits (insertions, deletions, or substitutions) required to convert string1 to string2.

```
select levenshtein_distance('helo word','hello,world'); -- 3
```

- hamming_distance(*string1*, *string2*) → bigint

Description: Returns the Hamming distance between character strings 1 and 2, that is, the number of different characters in the corresponding positions. Note that the lengths of the two strings must be the same.

```
select hamming_distance('abcde','edcba');-- 4
```

- instr(string,substring) → bigint

  Description: Locates the first occurrence of a substring in a string.

  ```
  select instr('abcde', 'cd');--3
  ```

- levenshtein(*string1*, *string2*) → bigint

  For details, see **levenshtein_distance(*string1*, *string2*)**.

- levenshtein_distance(*string1*, *string2*) → bigint

  Description: Returns the Levenshtein edit distance between string 1 and string 2, that is, the minimum number of single-character edits (insertion, deletion, or replacement) required to change string 1 to string 2.

  ```
  select levenshtein_distance('apple','epplea');-- 2
  ```

- lower(string) → varchar

  Description: Converts characters into lowercase letters.

  ```
  select lower('HELLo!');-- hello!
  ```

- lcase(string A) → varchar

  Description: Same as **lower(string)**.

- ltrim(string) → varchar

  Description: Removes spaces at the beginning of a character string.

  ```
  select ltrim('   hello');-- hello
  ```

- lpad(*string*, *size*, *padstring*) → varchar

  Description: Pads the string to the right to resize it using **padstring**. If *size* is less than the length of the string, the result is truncated to *size* characters. The size cannot be negative, and the padding string must not be empty.

  ```
  select lpad('myk',5,'fish'); -- domyk
  ```

- luhn_check(string) → boolean

  Description: Tests whether a numeric string is valid based on the Luhn algorithm.

  This checksum function, also known as mod 10, is widely used to validate a variety of identification numbers, such as credit card numbers and ID card numbers.

  ```
  select luhn_check('79927398713'); -- true
  select luhn_check('79927398714'); -- false
  ```

- octet_length(string str) → int

  Description: Returns the number of bytes for saving the string encoded using UTF-8.

  ```
  select octet_length('query');--5
  ```

- parse_url(string urlString, string partToExtract [, string keyToExtract]) →string

  Description: Returns the specified part of a URL. The valid value of the **partToExtract** parameter is **HOST**, **PATH**, **QUERY**, **REF**, **PROTOCOL**, **AUTHORITY**, **FILE**, and **USERINFO**. **keyToExtract** is an optional parameter, which is used to select the value corresponding to the key in **QUERY**.

  ```
  select parse_url('https://www.example.com/index.html','HOST');
    _col0
  ----------
  ```

```
 www.example.com
(1 row)

-- Query the value of service in QUERY of the URL.
select parse_url('https://www.example.com/query/index.html?name=panda','QUERY','name');
 _col0
-------
 panda
(1 row)
```

- position(*substring IN string*) →bigint

  Description: Returns the position of the first occurrence of a substring in the parent string.

  ```
  select position('ab' in 'sssababa');-- 4
  ```

- quote(String text) → string

  Description: Returns a string enclosed in single quotation marks. Strings containing single quotation marks are not supported.

  ```
  select quote('DONT');-- 'DONT'
  select quote(NULL);-- NULL
  ```

- repeat2(string str, int n) → string

  Description: Returns a string obtained by repeating the **str** string for *n* times.

  ```
  select repeat2('abc',4);
      _col0
  --------------
   abcabcabcabc
  (1 row)
  ```

- replace(string, 'a')→ varchar

  Description: Removes the character **a** from the character string.

  ```
  select replace('hello','e');-- hllo
  ```

- replace(string, 'a', 'b')→ varchar

  Description: Replaces all **a** characters in a string with **b**.

  ```
  select replace('hello','l','m');-- hemmo
  ```

- reverse(string) → varchar

  Description: reverses the string.

  ```
  select reverse('hello');-- olleh
  ```

- rpad(*string*, *size*, *padstring*) → varchar

  Description: Pads the string to the right to resize it using *padstring*. If *size* is less than the length of the string, the result is truncated to *size* characters. The size cannot be negative, and the padding string must not be empty.

  ```
  select rpad('myk',5,'fish'); -- mykdo
  ```

- rtrim(string) → varchar

  Description: Removes spaces at the end of a character string.

  ```
  select rtrim('hello world!   ');-- hello world!
  ```

- space(int n) → varchar

  Description: Returns *n* spaces.

  ```
  select space(4);
   _col0
  -------

  (1 row)

  select length(space(4));
   _col0
  ```

```
-------
    4
(1 row)
```

- split(string, delimiter) → array

  Description: Splits the string by delimiters into an array.

  ```
  select split('a:b:c:d',':');-- [a, b, c, d]
  ```

- split(string, delimiter, limit) → array

  Description: Splits a string into an array by delimiter. **limit** indicates the number of elements. The last element contains all the characters of the last string. **limit** must be a number.

  ```
  select split('a:b:c:d',':',2);-- [a, b:c:d]
  select split('a:b:c:d',':',4);-- [a, b, c, d]
  ```

- split_part(string, delimiter, index) → varchar

  Description: Splits a string into an array by delimiter and extracts the elements whose index value is index. The index starts from 1. If the index exceeds the array length, **NULL** is returned.

  ```
  select split_part('a:b:c:d',':',2); -- b
  select split_part('a:b:c:d',':',5); -- NULL
  ```

- split_to_map (string, entryDelimiter, keyValueDelimiter) → map<varchar, varchar>

  Description: Splits a string into mapped key-value pairs by **entryDelimiter**, and each key-value pair differentiates keys and values by **keyValueDelimiter**.

  ```
  select split_to_map('li:18,wang:17',',',':');--{wang=17, li=18}
  ```

- split_to_multimap(string, entryDelimiter, keyValueDelimiter) -> map(varchar, array(varchar)

  Description: Splits a string by **entryDelimiter** and **keyValueDelimiter** and returns a map. Each key corresponds to a value of the array type. **entryDelimiter** splits a string into key-value pairs, and **keyValueDelimiter** splits a key-value pair into a key and a value.

  ```
  select split_to_multimap('li:18,wang:17,li:19,wang:18',',',':');--{wang=[17, 18], li=[18, 19]}
  ```

- strpos(string, substring) → bigint

  Description: Returns the position of the first occurrence of **substring** in a string. The value starts from 1. If the value is not found, the value **0** is returned. Example:

  ```
  select strpos('hello world!','l'); --3
  select strpos('hello world!','da'); --0
  ```

- str_to_map() For details, see **split_to_map()**.
- substr(string, start) → varchar

  Description: Truncates a character string from the **start** position.

  ```
  select substr('hello world',3);-- llo world
  ```

- substr(string, start, length) → varchar

  Description: Truncates a character string from the **start** position. The truncated length is **length**.

  Generally, it is used to truncate the timestamp format.

  ```
  Select substr('2019-03-10 10:00:00',1,10); --Truncate to March 10, 2019.
  Select substr('2019-03-10 10:00:00',1,7); --Truncate to March 2019.
  ```

- substring(string, start) → varchar

  For details, see **substr(string, start)**.

- substring_index(string A, string delim, int count) → varchar

  Description: If *count* is a positive number, all content before the *count* delimiter from the left is returned. If *count* is a negative number, all content after the *count* delimiter from the right is returned.

```
select substring_index('one.two.three','.',2);
      _col0
-----------------
 one.two
(1 row)

select substring_index('one.two.three','.',-2);
      _col0
-----------------
 two.three
(1 row)

select substring_index('one.two.three','.',0);
 _col0
-------
 NULL
(1 row)
```

- soundex(string A) →varchar

  Description: Returns code (**soundex**) consisting of four characters to evaluate the similarity of two strings in pronunciation. The rules are as follows:

Table 1-5 Character mapping rule

Character	Digit
a, e, h, i, o, u, w, and y	0
b. f, p, and v	1
c. g, j, k, q, s, x, and z	2
d and t	3
l	4
m and n	5
r	6

  - Extracts the first letter of a string as the first value of **soundex**.
  - Replaces the latter letters with digits one by one based on the preceding letter mapping rules. If there are consecutive equal numbers, retain only one number and delete.
  - If the result contains more than four digits, the first four digits are used. If the result contains less than four digits, pad 0s to the end.

```
select soundex('Miller');
 _col0
-------
 M460
(1 row)
```

- translate(string|char|varchar input, string|char|varchar from, string|char|varchar to) →varchar

Description: Replaces the string specified by the **from** parameter with the string specified by the **to** parameter for an input string. If one of the three parameters is null, **NULL** is returned.

```
select translate('aabbcc','bb','BB');
 _col0
--------
 aaBBcc
(1 row)
```

- trim(string) → varchar

  Description: Removes spaces at the beginning and end of a character string.

  ```
  select trim('  hello world!  ');-- hello world!
  ```

- btrim(String str1,String str2) → varchar

  Description: Removes all characters contained in **str2** from the beginning and end of **str1**.

  ```
  select btrim('hello','hlo');-- e
  ```

- upper(string) → varchar

  Description: Converts character strings to uppercase letters.

  ```
  select upper('heLLo');-- HELLO
  ```

- ucase(string A) → varchar

  Description: Same as **upper(string)**.

- base64decode(STRING str)

  Description: Performs Base64 reverse encoding on the character string.

  ```
  SELECT to_base64(CAST('hello world' as varbinary));-- aGVsbG8gd29ybGQ=
  select base64decode('aGVsbG8gd29ybGQ=');-- hello world
  ```

- jaro_distance(STRING str1, STRING str2)

  Description: Compares the similarity between two character strings.

  ```
  select JARO_DISTANCE('hello', 'hell');-- 0.9333333333333332
  ```

- FNV_HASH(type v)

  Description: Calculates the hash value of a character string.

  ```
  select FNV_HASH('hello');-- -6615550055289275125
  ```

- word_stem(word) → varchar

  Description: Returns the stem of an English word.

  ```
  select word_stem('greating');-- great
  ```

- word_stem(*word*, *lang*) → varchar

  Description: Returns the stem of a word in a specified language.

  ```
  select word_stem('ultramoderne','fr');-- ultramodern
  ```

- translate(*source*, *from*, *to*) → varchar

  Description: Returns the translated source string by replacing the characters found in the source string with the corresponding characters in the target string. If the **from** string contains duplicate items, only the first one is used. If the source character does not exist in the **from** string, the source character is copied without translation. If the index of the matching character in the **from** string exceeds the length of the **to** string, the source character is omitted from the result string.

  ```
  SELECT translate('abcd', '', ''); -- 'abcd'
  SELECT translate('abcd', 'a', 'z'); -- 'zbcd'
  SELECT translate('abcda', 'a', 'z'); -- 'zbcdz'
  SELECT translate('Palhoça', 'ç','c'); -- 'Palhoca'
  ```

```
SELECT translate('abcd', 'a', ''); -- 'bcd'
SELECT translate('abcd', 'a', 'zy'); -- 'zbcd'
SELECT translate('abcd', 'ac', 'z'); -- 'zbd'
SELECT translate('abcd', 'aac', 'zq'); -- 'zbd'
```

Unicode functions

- normalize(string) → varchar

  Description: Returns a standard string in NFC format.

  ```
  select normalize('e');
   _col0
  -------
   e
  (1 row)
  ```

- normalize(string, form) → varchar

  Description: Unicode allows you to write the same character in different bytes. For example, **é** consists of **0xC3** and **0xA9**, and **é** consists of **0x65**, **0xCC**, and **0x81**.

  normalize() returns a standard string based on the Unicode standard formats (including NFC, NFD, NFKC, and NFKD) specified by the parameter format. If no parameter format is specified, NFC is used by default.

  ```
  select to_utf8('é');
   _col0
  -------
   c3 a9
  (1 row)

  select to_utf8('é');
    _col0
  ----------
   65 cc 81
  (1 row)

  select normalize('é',NFC)=normalize('é',NFC);
   _col0
  -------
   true
  (1 row)
  ```

- to_utf8(string) → varbinary

  Description: Encodes a string into a UTF-8 string.

  ```
  select to_utf8('panda');
       _col0
  ----------------
   70 61 6e 64 61
  (1 row)
  ```

- from_utf8(binary) → varchar

  Description: Encodes a binary string into a UTF-8 string. An invalid UTF-8 sequence will be replaced by the Unicode character U+FFFD.

  ```
  select from_utf8(X'70 61 6e 64 61');
   _col0
  -------
   panda
  (1 row)
  ```

- from_utf8(binary, replace) → varchar

  Description: Encodes a binary string into a UTF-8 string. An invalid UTF-8 sequence will be replaced by the **replace** parameter. The value of the **replace** parameter must be a single character or empty to prevent invalid characters from being removed.

```
select from_utf8(X'70 61 6e 64 61 b1','!');
 _col0
--------
 panda!
(1 row)
```

1.8.10 Regular Expressions

Overview

All regular expression functions use Java-style syntax, except in the following cases:

- Use the multi-line mode (through (? m) flag enabling), only **\n** is identified as a line terminator. In addition, it does not support (? d) Flag. Therefore, it cannot be used.
- In case-sensitive mode (through (? i) flag enabling), the unicode mode is always used. In addition, context-sensitive matching and local sensitive matching are not supported. In addition, it does not support (? u) flag.
- The Surrogate Pair encoding mode is not supported. For example, **\ uD800 \ uDC00** is not considered as U + 10000 and must be specified as **\ x {10000}**.
- The boundary character (**\b**) cannot be handled correctly because it is a non-spaced marker without a base character.
- \Q and \E are not supported in character classes (such as [A-Z123]). They are processed as text.
- Unicode characters (\ p {prop}) are supported. The differences are as follows:
  - All underscores in the name must be deleted. For example, use **OldItalic** instead of **Old_Italic**.
  - You must specify a script without the prefix **Is**, **script =**, or **sc =**. Example: **\p {Hiragana}**
  - The **In** prefix must be used to specify a block. The prefix **block =** or **blk =** is not supported. Example: **\p{Mongolian}**
  - You must specify a category without the prefix **Is**, **general_category =**, or **gc =**. Example: **\p{L}**
  - The binary attribute must be specified directly, not **Is**. Example: **\p{NoncharacterCodePoint}**

Function

- regexp_count(*string*, *pattern*) → bigint

  Description: Returns the number of pattern matches in a string.

  ```
  SELECT regexp_count('1a 2b 14m', '\s*[a-z]+\s*'); -- 3
  ```

- regexp_extract_all(string, pattern) -> array(varchar)

  Description: Returns all matched substrings in array format.

  ```
  SELECT regexp_extract_all('1a 2b 14m','\d+');-- [1, 2, 14]
  ```

- regexp_extract_all(string, pattern, group) -> array(varchar)

  Description: When the pattern contains multiple groups, group is used to return all substrings that meet the **captured group** conditions.

  ```
  SELECT regexp_extract_all('1a 2b 14m','(\d+)([a-z]+)',2);-- [a, b, m]
  ```

- regexp_extract(string, pattern) → varchar

  Description: Returns the first substring that matches the regular expression pattern in a string.

  ```
  SELECT regexp_extract('1a 2b 14m','\d+');-- 1
  ```

- regexp_extract(string, pattern, group) → varchar

  Description: When the pattern contains multiple groups, **group** is used to specify the first substring that meets **captured group**.

  ```
  SELECT regexp_extract('1a 2b 14m','(\d+)([a-z]+)',2);-- 'a'
  ```

- regexp_like(string, pattern) → boolean

  Description: Checks whether a string contains substrings that meet the regular expression. If yes, **true** is returned.

  ```
  SELECT regexp_like('1a 2b 14m','\d+b');-- true
  ```

- regexp_position(string, pattern) → integer

  Description: Returns the index that matches the pattern for the first time in a string. If no index is matched, returns **-1**.

  ```
  SELECT regexp_position('I have 23 apples, 5 pears and 13 oranges', '\b\d+\b'); -- 8
  ```

- regexp_position(string, pattern, start) → integer

  Description: Returns the index of the item that matches the pattern for the first time starting from the **start** index (included). If no index is matched, returns **-1**.

  ```
  SELECT regexp_position('I have 23 apples, 5 pears and 13 oranges', '\b\d+\b', 5); -- 8
  SELECT regexp_position('I have 23 apples, 5 pears and 13 oranges', '\b\d+\b', 12); -- 19
  ```

- regexp_position(string, pattern, start, occurrence) → integer

  Description: Returns the index of the item that matches the pattern for the **occurrence** time starting from the **start** index (included). If no index is matched, returns **-1**.

  ```
  SELECT regexp_position('I have 23 apples, 5 pears and 13 oranges','\b\d+\b',12,1);-- 19
  SELECT regexp_position('I have 23 apples, 5 pears and 13 oranges','\b\d+\b',12,2);-- 31
  SELECT regexp_position('I have 23 apples, 5 pears and 13 oranges','\b\d+\b',12,3);-- -1
  ```

- regexp_replace(string, pattern) → varchar

  Description: Removes substrings that meet the regular expression from the target string.

  ```
  SELECT regexp_replace('1a 2b 14m','\d+[ab] ');-- '14m'
  ```

- regexp_replace(string, pattern, replacement) → varchar

  Description: Replaces the substring that meets the regular expression in the target string with *replacement*. If the *replacement* contains the character **$**, use **\$** to escape the character. During replacement, you can use **$g** to reference a capture group for a numbered group and **${name}** to reference a capture group for a named group.

  ```
  SELECT regexp_replace('1a 2b 14m','(\d+)([ab]) ','3c$2 ');-- '3ca 3cb 14m'
  ```

- regexp_replace(string, pattern, function) → varchar

  Description: Replaces each instance of the substring that matches the regular expression pattern in the string with function. For each match, the **captured group** passed as an array calls the **lambda** expression function. The capture group ID starts from 1. The entire match is not grouped (brackets enclose the entire expression if necessary).

  ```
  SELECT regexp_replace('new york','(\w)(\w*)',x->upper(x[1])||lower(x[2]));--'New York'
  ```

- regexp_split(string, pattern) -> array(varchar)

  Description: Splits a string using the regular expression pattern and returns an array. The following empty character string is reserved:

  ```
  SELECT regexp_split('1a 2b 14m','\s*[a-z]+\s*');-- [1, 2, 14, ]
  ```

1.8.11 Binary Functions and Operators

Binary Operators

|| The operator performs the join.

Binary Functions

- length(binary) → bigint

  Return the byte length of **binary**.

  ```
  select length(x'00141f');-- 3
  ```

- concat(binary1, ..., binaryN) → varbinary

  Concatenates *binary1*, *binary2*, and *binaryN*. This function returns the same function as the SQL standard connector ||.

  ```
  select concat(X'32335F',x'00141f'); -- 32 33 5f 00 14 1f
  ```

- to_base64(binary) → varchar

  Encodes *binary* to a Base64 character string.

  ```
  select to_base64(CAST('hello world' as binary)); -- aGVsbG8gd29ybGQ=
  ```

- from_base64(string) → varbinary

  Decode the Base64-encoded string as varbinary.

  ```
  select from_base64('helloworld'); --  85 e9 65 a3 0a 2b 95
  ```

- unbase64(string) → varbinary

  Decode the Base64-encoded string as varbinary.

  ```
  SELECT from_base64('helloworld'); --  85 e9 65 a3 0a 2b 95
  ```

- to_base64url(binary) → varchar

  Use URL security characters to encode **binary** to a base64 character string.

  ```
  select to_base64url(x'555555');  -- VVVV
  ```

- from_base64url(string) → varbinary

  Use the URL security character to decode the Base64-encoded **string** into binary data.

  ```
  select from_base64url('helloworld'); --  85 e9 65 a3 0a 2b 95
  ```

- to_hex(binary) → varchar

  Encode the **binary** to a hexadecimal string.

  ```
  select to_hex(x'15245F');  --  15245F
  ```

- from_hex(string) → varbinary

  Decodes a hexadecimal string into binary data.

  ```
  select from_hex('FFFF'); --  ff ff
  ```

- to_big_endian_64(*bigint*) → varbinary

  Encodes a number of the bigint type into a 64-bit big-endian complement.

  ```
  select to_big_endian_64(1234);
          _col0
  ```

```
-------------------------
 00 00 00 00 00 00 04 d2
(1 row)
```

- from_big_endian_64(*binary*) → bigint

  The binary code in 64-bit big-endian complement format is decoded as a number of the bigint type.

```
select from_big_endian_64(x'00 00 00 00 00 00 04 d2');
 _col0
-------
  1234
(1 row)
```

- to_big_endian_32(*integer*) → varbinary

  Encodes a number of the bigint type into a 32-bit big-endian complement.

```
select to_big_endian_32(1999);
    _col0
-------------
 00 00 07 cf
(1 row)
```

- from_big_endian_32(*binary*) → integer

  The 32-bit big-endian two's complement format is decoded into a number of the bigint type.

```
select from_big_endian_32(x'00 00 07 cf');
 _col0
-------
  1999
(1 row)
```

- to_ieee754_32(*real*) → varbinary

  According to the IEEE 754 algorithm, a single-precision floating-point number is encoded into a 32-bit big-endian binary block.

```
select to_ieee754_32(3.14);
    _col0
-------------
 40 48 f5 c3
(1 row)
```

- from_ieee754_32(*binary*) → real

  Decodes the 32-bit big-endian binary in IEEE 754 single-precision floating-point format.

```
select from_ieee754_32(x'40 48 f5 c3');
 _col0
-------
  3.14
(1 row)
```

- to_ieee754_64(*double*) → varbinary

  According to the IEEE 754 algorithm, a double-precision floating point number is encoded into a 64-bit big-endian binary block.

```
select to_ieee754_64(3.14);
 _col0
-------------------------
 40 09 1e b8 51 eb 85 1f
(1 row)
```

- from_ieee754_64(*binary*) → double

  Decodes 64-bit big-endian binary in IEEE 754 single-precision floating-point format.

```
 select from_ieee754_64(X'40 09 1e b8 51 eb 85 1f');
 _col0
```

```
-------
  3.14
(1 row)
```

- lpad(*binary*, *size*, *padbinary*) → varbinary

  Left-padded binary to adjust byte size using padbinary. If *size* is less than the length of the binary file, the result will be truncated to *size* characters. The value of *size* cannot be negative, and the value of *padbinary* cannot be empty.

  ```
  select lpad(x'15245F', 11,x'15487F') ; -- 15 48 7f 15 48 7f 15 48 15 24 5f
  ```

- rpad(*binary*, *size*, *padbinary*) → varbinary

  Right-padded binary to use padbinary to resize bytes. If *size* is less than the length of the binary file, the result will be truncated to *size* characters. The value of *size* cannot be negative, and the value of *padbinary* cannot be empty.

  ```
  SELECT rpad(x'15245F', 11,x'15487F'); -- 15 24 5f 15 48 7f 15 48 7f 15 48
  ```

- crc32(*binary*) → bigint

  Calculate the CRC 32 value of the binary block.

- md5(*binary*) → varbinary

  Calculates the MD 5 hash value of a binary block.

- sha1(*binary*) → varbinary

  Calculates the SHA 1 hash value of a binary block.

- sha2(*string, integer*) → string

  SHA2 is a standard cryptographic hash algorithm used to convert a variable-length string into a string. Its output can be 224 bits, 256 bits, 384 bits, or 512 bits, corresponding to SHA-224, SHA-256, SHA-384 and SHA512, respectively.

- sha256(*binary*) → varbinary

  Calculates the SHA 256 hash value of a binary block.

- sha512(*binary*) → varbinary

  Calculates the SHA 512 hash value of a binary block.

- xxhash64(*binary*) → varbinary

  Calculates the XXHASH 64 hash value of a binary block.

- spooky_hash_v2_32(*binary*) → varbinary

  Calculates the 32-bit SpookyHashV2 hash value of a binary block.

- spooky_hash_v2_64(*binary*) → varbinary

  Calculates the 64-bit SpookyHashV2 hash value of a binary block.

- hmac_md5(*binary*, *key*) → varbinary

  Use the given key to calculate the HMAC value of the binary block (using MD5).

- hmac_sha1(*binary*, *key*) → varbinary

  Use the given key to calculate the HMAC value of the binary block (using SHA1).

- hmac_sha256(*binary*, *key*) → varbinary

  Use the given key to calculate the HMAC value of the binary block (using SHA256).

- hmac_sha512(*binary*, *key*) → varbinary

  Use the given key to calculate the HMAC value of the binary block (using SHA512).

NOTICE

The CRC32, MD5, and SHA1 algorithms have been found to have vulnerabilities. Do not use them for cryptography.

1.8.12 JSON Functions and Operators

- Cast to JSON
```
SELECT CAST(9223372036854775807 AS JSON); -- JSON '9223372036854775807'
```
- Cast from JSON
```
SELECT CAST(JSON '[1,23,456]' AS ARRAY(INTEGER)); -- [1, 23, 456]
```

JSON Function

NOTE

The conversion from NULL to JSON cannot be simply implemented. Converting from a separate NULL produces a SQLNULL instead of JSON'null'. However, when converted from an array or Map containing NULL, the generated JSON will contain NULL.

When converted from ROW to JSON, the result is a JSON array, not a JSON object. This is because for rows in SQL, the location is more important than the name.

The value can be converted from BOOLEAN, TINYINT, SMALLINT, INTEGER, BIGINT, REAL, DOUBLE, or VARCHAR. If the element type of an array is one of the supported types, the key type of a map is VARCHAR, and the value type of a map is one of the supported types, or the type of each field in a row is one of the supported types, the array can be converted from ARRAY, MAP, or ROW. The following example shows the behavior of the transformation:

```
SELECT CAST(NULL AS JSON);-- NULL
SELECT CAST(1 AS JSON);-- JSON '1'
SELECT CAST(9223372036854775807 AS JSON);-- JSON '9223372036854775807'
SELECT CAST('abc' AS JSON);-- JSON '"abc"'
SELECT CAST(true AS JSON);-- JSON 'true'
SELECT CAST(1.234 AS JSON);-- JSON '1.234'
SELECT CAST(ARRAY[1, 23, 456] AS JSON);-- JSON '[1,23,456]'
SELECT CAST(ARRAY[1, NULL, 456] AS JSON);-- JSON '[1,null,456]'
SELECT CAST(ARRAY[ARRAY[1, 23], ARRAY[456]] AS JSON);-- JSON '[[1,23],[456]]'
SELECT CAST(MAP(ARRAY['k1', 'k2', 'k3'], ARRAY[1, 23, 456]) AS JSON);-- JSON '{"k1":1,"k2":23,"k3":456}'
SELECT CAST(CAST(ROW(123, 'abc', true) AS ROW(v1 BIGINT, v2 VARCHAR, v3 BOOLEAN)) AS JSON);--
JSON '[123,"abc",true]'
```

JSON-to-other Types

```
SELECT CAST(JSON 'null' AS VARCHAR);-- NULL
SELECT CAST(JSON '1' AS INTEGER);-- 1
SELECT CAST(JSON '9223372036854775807' AS BIGINT);-- 9223372036854775807
SELECT CAST(JSON '"abc"' AS VARCHAR);-- abc
SELECT CAST(JSON 'true' AS BOOLEAN);-- true
SELECT CAST(JSON '1.234' AS DOUBLE);-- 1.234
SELECT CAST(JSON '[1,23,456]' AS ARRAY(INTEGER));-- [1, 23, 456]
SELECT CAST(JSON '[1,null,456]' AS ARRAY(INTEGER));-- [1, NULL, 456]
SELECT CAST(JSON '[[1,23],[456]]' AS ARRAY(ARRAY(INTEGER)));-- [[1, 23], [456]]
SELECT CAST(JSON '{"k1":1, "k2":23, "k3":456}' AS MAP(VARCHAR, INTEGER));-- {k1=1, k2=23, k3=456}
SELECT CAST(JSON '{"v1":123, "v2":"abc","v3":true}' AS ROW(v1 BIGINT, v2 VARCHAR, v3 BOOLEAN));--
{v1=123, v2=abc, v3=true}
SELECT CAST(JSON '[123, "abc",true]' AS ROW(v1 BIGINT, v2 VARCHAR, v3 BOOLEAN));-- {value1=123,
value2=abc, value3=true}
SELECT CAST(JSON'[[1, 23], 456]'AS ARRAY(JSON));-- [JSON '[1,23]', JSON '456']
SELECT CAST(JSON'{"k1": [1, 23], "k2": 456}'AS MAP(VARCHAR,JSON));-- {k1 = JSON '[1,23]', k2 = JSON
'456'}
SELECT CAST(JSON'[null]'AS ARRAY(JSON));-- [JSON 'null']
```

NOTE

JSON arrays and JSON objects are supported during conversion from JSON to ROW.

JSON arrays can have mixed element types, and JSON Maps can have mixed value types. This makes it impossible to convert it to SQL arrays and maps in some cases. To solve this problem, HetuEngine supports partial conversion of arrays and maps.

```
SELECT CAST(JSON'[[1, 23], 456]'AS ARRAY(JSON));-- [JSON '[1,23]', JSON '456']
SELECT CAST(JSON'{"k1": [1, 23], "k2": 456}'AS MAP(VARCHAR,JSON));-- {k1 = JSON '[1,23]', k2 =
JSON '456'}
SELECT CAST(JSON'[null]'AS ARRAY(JSON));-- [JSON 'null']
```

- is_json_scalar(json) → boolean

  Check whether JSON is a scalar (that is, JSON number, JSON character string, true, false, or null).

  ```
  select is_json_scalar(json'[1,22]'); -- false
  ```

- json_array_contains(json, value) → boolean

  Checks whether a value is contained in **json**.

  ```
  select json_array_contains(json '[1,23,44]',23); -- true
  ```

- json_array_get(json_array, index) → json

  NOTICE

  The semantics of the function has been broken. If the extracted element is a string, it will be converted to an invalid JSON value that is not correctly enclosed in quotation marks (the value will not be enclosed in quotation marks, and any internal quotation marks will not be escaped). You are not advised to use this function. The function cannot be corrected without affecting existing usage and may be deleted in future versions.

  Returns the JSON element at the specified index position. The index starts from 0.

  ```
  SELECT json_array_get('["a", [3, 9], "c"]', 0); -- JSON 'a' (invalid JSON)
  SELECT json_array_get('["a", [3, 9], "c"]', 1); -- JSON '[3,9]'
  ```

  The index page supports negative numbers, indicating that the index page starts from the last element. The value **-1** indicates the last element. If the index length exceeds the actual length, **null** is returned.

  ```
  SELECT json_array_get('["c", [3, 9], "a"]', -1); -- JSON 'a' (invalid JSON)
  SELECT json_array_get('["c", [3, 9], "a"]', -2); -- JSON '[3,9]'
  ```

  If the JSON element at the specified index does not exist, NULL is returned.

  ```
  SELECT json_array_get('[]', 0);                -- NULL
  SELECT json_array_get('["a", "b", "c"]', 10);  -- NULL
  SELECT json_array_get('["c", "b", "a"]', -10); -- NULL
  ```

- json_array_length(json) → bigint

  Returns the length of **json**.

  ```
  SELECT json_array_length(json '[1,2,3,4]'); -- 4
  SELECT json_array_length('[1, 2, 3]'); -- 3
  ```

- get_json_object(string json,string json_path);

  Captures information in **json** based on the **json_path** format.

  ```
  SELECT get_json_object('{"id": 1, "value":"xxx"}', '$.value');  -- "xxx"
  ```

- json_extract(json, json_path) → json

  Captures information in **json** based on the **json_path** format.

  ```
  SELECT json_extract(json '{"id": 1, "value":"xxx"}', '$.value');-- JSON "xxx"
  ```

- json_extract_scalar(json, json_path) → varchar

  The function is the same as that of **json_extract**. The return value is varchar.

  ```
  SELECT json_extract_scalar(json '{"id": 1, "value": "xxx"}', '$.value'); -- xxx
  ```

- json_format(json) → varchar

  Converts a JSON value to a serialized JSON text. This is the inverse function of json_parse.

  ```
  SELECT JSON_format(json '{"id": 1, "value":"xxx"}'); -- {"id":1, "value":"xxx"}
  ```

  Notes:

  json_format and CAST(json AS VARCHAR) have completely different semantics.

  json_format serializes the input JSON value into JSON text that complies with the 7159 standard. The JSON value can be a JSON object, JSON array, JSON string, JSON number, true, false, or null:

  ```
  SELECT json_format(JSON '{"a": 1, "b": 2}'); -- '{"a":1,"b":2}'
  SELECT json_format(JSON '[1, 2, 3]'); -- '[1,2,3]'
  SELECT json_format(JSON '"abc"'); -- '"abc"'
  SELECT json_format(JSON '42'); -- '42'
  SELECT json_format(JSON 'true'); -- 'true'
  SELECT json_format(JSON 'null'); -- 'null'
  ```

  CAST(json AS VARCHAR) converts the JSON value to the corresponding SQL VARCHAR value. For JSON strings, JSON numbers, true, false, or null, the conversion behavior is the same as that of the corresponding SQL type. JSON objects and JSON arrays cannot be converted to VARCHAR.

  ```
  SELECT CAST(JSON '{"a": 1, "b": 2}' AS VARCHAR); -- ERROR!
  SELECT CAST(JSON '[1, 2, 3]' AS VARCHAR);         -- ERROR!
  SELECT CAST(JSON '"abc"' AS VARCHAR);             -- 'abc' (the double quote is gone)
  SELECT CAST(JSON '42' AS VARCHAR);               -- '42'
  SELECT CAST(JSON 'true' AS VARCHAR);             -- 'true'
  SELECT CAST(JSON 'null' AS VARCHAR);             -- NULL
  ```

- json_parse(string) → json

  Contrary to **json_format(json)**, convert a JSON character string to JSON.

  json_parse and **json_extract** are used together to parse JSON character strings in data tables.

  ```
  select JSON_parse('{"id": 1, "value":"xxx"}'); -- json {"id":1, "value":"xxx"}
  ```

- json_size(json, json_path) → bigint

  It is similar to **json_extract**, but the number of objects in JSON is returned.

  ```
  SELECT json_size('{ "x": {"a": 1, "b": 2} }', '$.x'); => 2
  SELECT json_size('{ "x": [1, 2, 3] }', '$.x'); =>3
  SELECT json_size('{ "x": {"a": 1, "b": 2} }', '$.x.a'); => 0
  ```

1.8.13 Date and Time Functions and Operators

Date and Time Operators

Operator	Example	Result
+	date '2012-08-08' + interval '2' day	2012-08-10

Operator	Example	Result
+	time '01:00' + interval '3' hour	04:00:00.000
+	timestamp '2012-08-08 01:00' + interval '29' hour	2012-08-09 06:00:00.000
+	timestamp '2012-10-31 01:00' + interval '1' month	2012-11-30 01:00:00.000
+	interval '2' day + interval '3' hour	2 03:00:00.000
+	interval '3' year + interval '5' month	3-5
-	date '2012-08-08' - interval '2' day	2012-08-06
-	time '01:00' - interval '3' hour	22:00:00.000
-	timestamp '2012-08-08 01:00' - interval '29' hour	2012-08-06 20:00:00.000
-	timestamp '2012-10-31 01:00' - interval '1' month	2012-09-30 01:00:00.000
-	interval '2' day - interval '3' hour	1 21:00:00.000
-	interval '3' year - interval '5' month	2-7

Time Zone Conversion

Operator: **AT TIME ZONE** sets the time zone of a timestamp.

```
SELECT timestamp '2012-10-31 01:00 UTC';-- 2012-10-31 01:00:00.000 UTC
SELECT timestamp '2012-10-31 01:00 UTC' AT TIME ZONE 'Asia/Singapore'; -- 2012-10-30 09:00:00.000
Asia/Singapore
```

Date/Time Functions

- current_date -> date

  Returns the current date (UTC time zone).

  ```
  select current_date; -- 2020-07-25
  ```

- current_time -> time with time zone

  Returns the current time (UTC time zone).

  ```
  select current_time;-- 16:58:48.601+08:00
  ```

- current_timestamp -> timestamp with time zone

  Returns the current timestamp (current time zone).

  ```
  select current_timestamp; -- 2020-07-25 11:50:27.350 Asia/Singapore
  ```

- current_timezone() → varchar

  Returns the current time zone.

  ```
  select current_timezone();-- Asia/Singapore
  ```

- date(x) → date

  Converts a date literal to a variable of the date type.

```
select date('2020-07-25');-- 2020-07-25
```

- from_iso8601_timestamp(string) → timestamp with time zone

  Converts a timestamp literal in ISO 8601 format into a timestamp variable with a time zone.

```
SELECT from_iso8601_timestamp('2020-05-11');-- 2020-05-11 00:00:00.000 Asia/Singapore
SELECT from_iso8601_timestamp('2020-05-11T11:15:05'); -- 2020-05-11 11:15:05.000 Asia/Singapore
SELECT from_iso8601_timestamp('2020-05-11T11:15:05.055+01:00');-- 2020-05-11 11:15:05.055 +01:00
```

- from_iso8601_date(string) → date

  Converts a date literal in ISO 8601 format into a variable of the date type.

```
SELECT from_iso8601_date('2020-05-11');-- 2020-05-11
SELECT from_iso8601_date('2020-W10');-- 2020-03-02
SELECT from_iso8601_date('2020-123');-- 2020-05-02
```

- from_unixtime(unixtime) → timestamp with time zone

  Converts a UNIX timestamp to a timestamp variable (current time zone).

```
Select FROM_UNIXTIME(1.595658735E9); -- 2020-07-25 14:32:15.000 Asia/Singapore
Select FROM_UNIXTIME(875996580); --1997-10-05 04:23:00.000 Asia/Singapore
```

- from_unixtime(unixtime, string) → timestamp with time zone

  Converts a UNIX timestamp into a timestamp variable. The time zone option can be contained.

```
select from_unixtime(1.595658735E9, 'Asia/Singapore');-- 2020-07-25 14:32:15.000 Asia/Singapore
```

- from_unixtime(unixtime, hours, minutes) → timestamp with time zone

  Converts a UNIX timestamp to a timestamp variable with a time zone. **hours** and **minutes** indicate the time zone offsets.

```
select from_unixtime(1.595658735E9, 8, 30);-- 2020-07-25 14:32:15.000 +08:30
```

- localtime -> time

  Obtains the current time

```
select localtime;-- 14:16:13.096
```

- localtimestamp -> timestamp

  Obtains the current timestamps.

```
select localtimestamp;-- 2020-07-25 14:17:00.567
```

- months_between(date1, date2) -> double

  Return the number of months between *date1* and *date2*. If *date1* is later than *date2*, the result is a positive number. Otherwise, the result is a negative number. If the days of the two dates are the same, the result is an integer. Otherwise, the decimal part is calculated based on the difference between the hour, minute, and second (31 days of each month). The type of *date1* and *date2* can be date, timestamp, or a string in the yyyy-MM-dd or yyyy-MM-dd HH:mm:ss format.

```
select months_between('2020-02-28 10:30:00', '2021-10-30');-- -20.05040323
select months_between('2021-01-30', '2020-10-30'); -- 3.0
```

- now() → timestamp with time zone

  Obtains the current time, which is the alias of **current_timestamp**.

```
select now();-- 2020-07-25 14:39:39.842 Asia/Singapore
```

- unix_timestamp()

  Obtains the current **unix** timestamp.

```
select unix_timestamp(); -- 1600930503
```

- to_iso8601(x) → varchar

Converts *x* into a character string in the ISO 8601 format. x can be DATE or TIMESTAMP [with time zone].

```
select to_iso8601(date '2020-07-25'); -- 2020-07-25
select to_iso8601(timestamp '2020-07-25 15:22:15.214'); -- 2020-07-25T15:22:15.214
```

- to_milliseconds(interval) → bigint

  Obtains the number of milliseconds since 00:00 on the current day.

  ```
  select to_milliseconds(interval '8' day to second);-- 691200000
  ```

- to_unixtime(timestamp) → double

  Converts the timestamp to the UNIX time.

  ```
  select to_unixtime(cast('2020-07-25 14:32:15.147' as timestamp));-- 1.595658735147E9
  ```

- trunc(string date, string format) →string

  Truncates a date value based on the format. The supported format is MONTH/MON/MM or YEAR/YYYY/YY, QUARTER/Q

  ```
  select trunc(date '2020-07-08','yy');-- 2020-01-01
  select trunc(date '2020-07-08','MM');-- 2020-07-01
  ```

NOTE

You can use the parentheses () when using the following SQL standard functions:

- current_date
- current_time
- current_timestamp
- localtime
- Localtimestamp

For example: **select current_date ();**

Truncation Function

Similar to the operation of reserving decimal places, the **date_trunc** function supports the following units:

Unit	Value After Truncation
second	2001-08-22 03:04:05.000
minute	2001-08-22 03:04:00.000
hour	2001-08-22 03:00:00.000
day	2001-08-22 00:00:00.000
week	2001-08-20 00:00:00.000
month	2001-08-01 00:00:00.000
quarter	2001-07-01 00:00:00.000
year	2001-01-01 00:00:00.000

In the preceding example, the timestamp **2001-08-22 03:04:05.321** is used as the input.

date_trunc(unit, x) → [same as input]

Returns the value of *x* after the **unit**.

```
select date_trunc('hour', timestamp '2001-08-22 03:04:05.321'); -- 2001-08-22 03:00:00.000
```

Interval Functions

The functions in this chapter support the following interval units:

Unit	Description
second	Seconds
minute	Minutes
hour	Hours
day	Days
week	Weeks
month	Months
quarter	Quarters of a year
year	Years

- date_add(unit, value, timestamp) → [same as input]

  Add *value* units to timestamp. If you want to perform the subtraction operation, you can assign a negative value to the *value*.

  ```
  SELECT date_add('second', 86, TIMESTAMP '2020-03-01 00:00:00');-- 2020-03-01 00:01:26
  SELECT date_add('hour', 9, TIMESTAMP '2020-03-01 00:00:00');-- 2020-03-01 09:00:00
  SELECT date_add('day', -1, TIMESTAMP '2020-03-01 00:00:00 UTC');-- 2020-02-29 00:00:00 UTC
  ```

- date_diff(unit, timestamp1, timestamp2) → bigint

  Returns the value of timestamp2 minus timestamp1. The unit of the value is *unit*.

  The value of *unit* is a character string. For example, **day**, **week**, and **year**.

  ```
  SELECT date_diff('second', TIMESTAMP '2020-03-01 00:00:00', TIMESTAMP '2020-03-02 00:00:00');--
  86400
  SELECT date_diff('hour', TIMESTAMP '2020-03-01 00:00:00 UTC', TIMESTAMP '2020-03-02 00:00:00
  UTC');-- 24
  SELECT date_diff('day', DATE '2020-03-01', DATE '2020-03-02');-- 1
  SELECT date_diff('second', TIMESTAMP '2020-06-01 12:30:45.000', TIMESTAMP '2020-06-02
  12:30:45.123');-- 86400
  SELECT date_diff('millisecond', TIMESTAMP '2020-06-01 12:30:45.000', TIMESTAMP '2020-06-02
  12:30:45.123');-- 86400123
  ```

- adddate(date, bigint)→ [same as input]

  Description: Date addition. The input type can be date or timestamp, indicating that the date is added or deducted. If the input type is subtraction, the value of bigint is negative.

  ```
  select ADDDATE(timestamp '2020-07-04 15:22:15.124',-5);-- 2020-06-29 15:22:15.124
  select ADDDATE(date '2020-07-24',5); -- 2020-07-29
  ```

Duration Function

The duration can use the following units:

Unit	Description
ns	nanosecond
us	microsecond
ms	millisecond
s	second
m	minute
h	hour
d	day

parse_duration(string) → interval

```
SELECT parse_duration('42.8ms'); -- 0 00:00:00.043
SELECT parse_duration('3.81 d'); -- 3 19:26:24.000
SELECT parse_duration('5m'); -- 0 00:05:00.000
```

MySQL Date Functions

The formatted strings that are compatible with the MySQL **date_parse** and **str_to_date** methods in this section.

- date_format(timestamp, format) → varchar

  Uses **format** to format **timestamp**.

  ```
  select date_format(timestamp '2020-07-22 15:00:15', '%Y/%m/%d');-- 2020/07/22
  ```

- date_parse(string, format) → timestamp

  Parses the date literals using **format**.

  ```
  select date_parse('2020/07/20', '%Y/%m/%d');-- 2020-07-20 00:00:00.000
  ```

The following table is based on the MySQL manual and describes various format descriptors.

Format Descriptor	Description
%a	Day in a week (Sun .. Sat)
%b	Month (Jan .. Dec)
%c	Month (1 .. 12)
%D	Day of the month (0th, 1st, 2nd, 3rd, ...)
%d	Day in the month (01.. 31) (Two digits. Zeros are added before the single digits.)

Format Descriptor	Description
%e	Day in the month (1 .. 31)
%f	Seconds after the decimal point (6 digits for printing: 000000 .. 999000; 1 - 9 digits for parsing: 0 .. 999999999)
%H	Hour (00 .. 23)
%h	Hour (01.. 12)
%I	Hour (01.. 12)
%i	Minute, number (00 .. (59)
%j	Day of the year (001 .. 366)
%k	Hour (0 .. 23)
%l	Hour (1.. 12)
%M	Month name (January .. December)
%m	Month, number (01 .. 12)
%p	AM or PM
%r	Time in the 12-hour format (hh:mm:ss followed by AM or PM)
%S	Second (00 .. 59)
%s	Second (00 .. 59)
%T	Time in the 24-hour format (hh:mm:ss)
%U	Week (00 .. 53) Sunday is the first day of a week.
%u	Week (00 .. 53) Monday is the first day of a week.
%V	Week (01.. 53) Sunday is the first day of a week. Used together with %X.
%v	Week (01 .. 53) Monday is the first day of a week. Used together with %X.
%W	Day of the week (Sunday .. Saturday)
%w	Day of the week (0.. 6) Sunday is the first day of a week.
%X	Year, a four-digit number. The first day is Sunday.
%x	Year, a four-digit number. The first day is Monday.
%Y	Year, a four-digit number.
%y	Year. The value is a two-digit number ranging from 1970 to 2069.

Format Descriptor	Description
%%	Indicates the character '%'.

Example:

```
select date_format(timestamp '2020-07-25 15:04:00.124','%j day of a year with English suffix (0th, 1st, 2nd, 3rd...),%m month %d day,%p %T %W');
                        _col0
---------------------------------------------------
 207th day of the year, 25th day of July, PM 15:04:00 Saturday
(1 row)
```

NOTE

These format descriptors are not supported: %D, %U, %u, %V, %w, %X.

- date_format(timestamp, format) → varchar

  Uses **format** to format **timestamp**.
- date_parse(string, format) → timestamp

  Parses the timestamp character string.

  ```
  select date_parse('2020/07/20', '%Y/%m/%d');-- 2020-07-20 00:00:00.000
  ```

Java Date Functions

The formatting strings used in this section are compatible with the Java **SimpleDateFormat** style.

- format_datetime(timestamp, format) → varchar

  Uses **format** to format **timestamp**.
- parse_datetime(string, format) → timestamp with time zone

  Format a string to timestamp with time zone in a specified format.

  ```
  select parse_datetime('1960/01/22 03:04', 'yyyy/MM/dd HH:mm');
                   _col0
  ---------------------------------------
   1960-01-22 03:04:00.000 Asia/Shanghai
  (1 row)
  ```

Common Extraction Functions

Domain	Description
YEAR	year()
QUARTER	quarter()
MONTH	month()
WEEK	week()
DAY	day()
DAY_OF_MONTH	day_of_month()

Domain	Description
DAY_OF_WEEK	day_of_week()
DOW	day_of_week()
DAY_OF_YEAR	day_of_year()
DOY	day_of_year()
YEAR_OF_WEEK	year_of_week()
YOW	year_of_week()
HOUR	hour()
MINUTE	minute()
SECOND	second()
TIMEZONE_HOUR	timezone_hour()
TIMEZONE_MINUTE	timezone_minute()

Example:

```
select second(timestamp '2020-02-12 15:32:33.215');-- 33
select timezone_hour(timestamp '2020-02-12 15:32:33.215');-- 8
```

- MONTHNAME(date)

  Description: Obtains the month name.

  ```
  SELECT monthname(timestamp '2019-09-09 12:12:12.000');-- SEPTEMBER
  SELECT monthname(date '2019-07-09');--JULY
  ```

- extract(field FROM x) → bigint

  Description: Returns the field from x. For details about the corresponding field, see the table in this document.

  ```
  select extract(YOW FROM timestamp '2020-02-12 15:32:33.215');-- 2020
  select extract(SECOND FROM timestamp '2020-02-12 15:32:33.215');-- 33
  select extract(DOY FROM timestamp '2020-02-12 15:32:33.215');--43
  ```

Function	Example	Description
SECONDS_ADD(TIMESTAMP date, INT seconds)	SELECT seconds_add(timestamp '2019-09-09 12:12:12.000', 10);	The time is added in seconds.
SECONDS_SUB(TIMESTAMP date, INT seconds)	SELECT seconds_sub(timestamp '2019-09-09 12:12:12.000', 10);	Subtracts time in seconds.
MINUTES_ADD(TIMESTAMP date, INT minutes)	SELECT MINUTES_ADD(timestamp '2019-09-09 12:12:12.000', 10);	Add the time in the unit of minute.

Function	Example	Description
MINUTES_SUB(TIMESTAMP date, INT minutes)	SELECT MINUTES_SUB(timestamp '2019-09-09 12:12:12.000', 10);	The time is subtracted in the unit of minute.
HOURS_ADD(TIMESTAMP date, INT hours)	SELECT HOURS_ADD(timestamp '2019-09-09 12:12:12.000', 1);	Add the time in the unit of hour.
HOURS_SUB(TIMESTAMP date, INT hours)	SELECT HOURS_SUB(timestamp '2019-09-09 12:12:12.000', 1);	The time is subtracted in hours.

- last_day(timestamp) -> date

  Description: Returns the last day of each month based on the specified timestamp.

  ```
  SELECT last_day(timestamp '2019-09-09 12:12:12.000');--  2019-09-30
  SELECT last_day(date '2019-07-09');--2019-07-31
  ```

- add_months(timestamp) -> [same as input]

  Description: Returns the correct date by adding the specified date to the specified month.

  ```
  SELECT add_months(timestamp'2019-09-09 00:00:00.000', 11);-- 2020-08-09 00:00:00.000
  ```

- next_day() (timestamp, string) -> date

  Description: Returns the next day of the specified weekday based on the specified date.

  ```
  SELECT next_day(timestamp'2019-09-09 00:00:00.000', 'monday');-- 2019-09-16 00:00:00.000
  SELECT next_day(date'2019-09-09', 'monday');-- 2019-09-16
  ```

- numtoday(integer) -> BIGINT

  Description: Converts transferred integer values to values of the day type, for example, BIGINT.

  ```
  SELECT numtoday(2);-- 2
  ```

1.8.14 Aggregate Functions

The aggregate function operates on a set of values to obtain a single value.

Except **count()**, **count_if()**, **max_by()**, **min_by()**, and **approx_distinct()**, other aggregate functions ignore null values and return null values when there is no input row or all values are null. For example, **sum()** returns null instead of 0, and **avg()** does not take null values during statistics collection. The **coalesce** function converts null to 0.

Clause of an Aggregate Function

- Order by

  Some aggregate functions may generate different results due to different sequences of input values. You can use the **order by** clause in the aggregate function to specify the sequence.

```
array_agg(x ORDER BY y DESC);
array_agg(x ORDER BYx,y,z);
```

- Filter

  You can use the **filter** keyword to filter out unnecessary rows by using the **where** condition expression during aggregation. All aggregate functions support this function.

  ```
  aggregate_function(...) FILTER (WHERE <condition>)
  ```

Example:

```
--Create a table.
create table fruit (name varchar, price int);
--Insert data.
insert into fruit values ('peach',5),('apple',2);
--Sorting:
select array_agg (name order by price) from fruit;-- [apple, peach]
--Filtering:
select array_agg(name) filter (where price<10) from fruit;-- [peach, apple]
```

Common Aggregate Functions

An aggregate function usually applies to a specific field in a data set (table or view). The following parameter *x* is used to refer to the field.

- arbitrary(x)

  Description: Returns a non-null value of X. The return type is the same as X.

  ```
  select arbitrary(price) from fruit;-- 5
  ```

- array_agg(x)

  Description: Returns an array of input x fields of the same type as the input field.

  ```
  select array_agg(price) from fruit;-- [5,2]
  ```

- avg(x)

  Description: Returns the average of all input values in **double** type.

  ```
  select avg(price) from fruit;-- 3.5
  ```

- avg(time interval type)

  Description: Returns the average length of all input intervals. The return type is **interval**.

  ```
  select avg(last_login) from (values ('admin',interval '0 06:15:30' day to second),('user1',interval '0
  07:15:30' day to second),('user2',interval '0 08:15:30' day to second)) as login_log(user,last_login);
  -- 0 07:15:30.000  Assume that a log table records the time since the last login. The result indicates
  that the average login interval is 0 days, 7 hours, 15 minutes, and 30 seconds.
  ```

- bool_and(boolean value)

  Description: Returns true if each input value is **true**, otherwise returns **false**.

  ```
  select bool_and(isfruit) from (values ('item1',true), ('item2',false),('item3',true)) as
  items(item,isfruit);--false
  select bool_and(isfruit) from (values ('item1',true), ('item2',true),('item3',true)) as
  items(item,isfruit);-- true
  ```

- bool_or(boolean value)

  Description: Returns **true** if any of the input values is **true**, otherwise returns **false**.

  ```
  select bool_or(isfruit) from (values ('item1',false), ('item2',false),('item3',false)) as
  items(item,isfruit);-- false
  select bool_or(isfruit) from (values ('item1',true), ('item2',false),('item3',false)) as items(item,isfruit); --
  true
  ```

- checksum(x)

  Description: Returns the checksum of the input value, which is not affected by the input sequence. The result type is **varbinary**.

  ```
  select checksum(price) from fruit; -- fb 28 f3 9a 9a fb bf 86
  ```

- count(*)

  Description: Returns the number of input records. The result type is **bigint**.

  ```
  select count(*) from fruit; -- 2
  ```

- count(x)

  Description: Returns the number of records whose input field is not null. The result type is **bigint**.

  ```
  select count(name) from fruit;-- 2
  ```

- count_if(x)

  Description: Returns the number of records whose input value is **true**. This function is similar to count**(CASE WHEN x THEN 1 END)** and is of the **bigint** type.

  ```
  select count_if(price>7) from fruit;-- 0
  ```

- every(boolean)

  Description: Is an alias for **bool_and()**.

- geometric_mean(x)

  Description: Returns the geometric mean of the input field value. The value is of the **double** type.

  ```
  select geometric_mean(price) from fruit; -- 3.162277660168379
  ```

- listagg(x, separator) → varchar

  Description: Returns a string concatenated by input values separated by specified separators.

  Syntax:

  ```
  LISTAGG( expression [, separator] [ON OVERFLOW overflow_behaviour])    WITHIN GROUP (ORDER
  BY sort_item, [...])
  ```

  If separator is not specified, the null character is used as the separator by default.

  ```
  SELECT listagg(value, ',') WITHIN GROUP (ORDER BY value) csv_value FROM (VALUES 'a', 'c', 'b')
  t(value);
  csv_value
  -----------
  'a,b,c'
  ```

  When the return value of this function exceeds 1048576 bytes, you can use **overflow_behaviour** to specify the action to take in this case. By default, an error will be thrown.

  ```
  SELECT listagg(value, ',' ON OVERFLOW ERROR) WITHIN GROUP (ORDER BY value) csv_value FROM
  (VALUES 'a', 'b', 'c') t(value);
  ```

  If the return value exceeds 1048576 bytes, truncate the extra non-null string and replace it with the string specified by **TRUNCATE**. **WITH COUNT** and **WITHOUT COUNT** indicate whether the return value contains the count.

  ```
  SELECT LISTAGG(value, ',' ON OVERFLOW TRUNCATE '.....' WITH COUNT) WITHIN GROUP (ORDER BY
  value)FROM (VALUES 'a', 'b', 'c') t(value);
  ```

  The **listagg** function can also be used for grouping strings. The following is an example:

  ```
  SELECT id, LISTAGG(value, ',') WITHIN GROUP (ORDER BY o) csv_value FROM (VALUES
      (100, 1, 'a'),
  ```

```
    (200, 3, 'c'),
    (200, 2, 'b') ) t(id, o, value)
GROUP BY id
ORDER BY id;
id   | csv_value
-----+-------------
100  | a  200 | b,c
```

- max_by(x, y)

  Description: Returns the value of **x** associated with the maximum value of the **y** field in all input values.

  ```
  select max_by(name,price) from fruit; -- peach
  ```

- max_by(x, y, n)

  Description: Returns *n* **x** values sorted by **y** in descending order.

  ```
  select max_by(name,price,2) from fruit;-- [peach, apple]
  ```

- min_by(x,y)

  Description: Returns the value of **x** associated with the minimum value of the **y** field in all input values.

  ```
  select min_by(name,price) from fruit;-- apple
  ```

- min_by(x, y, n)

  Description: Returns *n* **x** values sorted by **y** in ascending order.

  ```
  select min_by(name,price,2) from fruit;-- [apple, peach]
  ```

- max(x)

  Description: Returns the maximum value of the input field **x**.

  ```
  select max(price) from fruit;-- 5
  ```

- max(x, n)

  Description: Returns the first *n* values of the input field **x** in descending order.

  ```
  select max(price,2) from fruit; -- [5, 2]
  ```

- min(x)

  Description: Returns the minimum value of the input field **x**.

  ```
  select min(price) from fruit;-- 2
  ```

- min(x, n)

  Description: Returns the first *n* values of the input field **x** in ascending order.

  ```
  select min(price,2) from fruit;-- [2, 5]
  ```

- sum(T, x)

  Description: Sums up the input field *x*. *T* is of the numeric type, for example, **int**, **double**, or **interval day to second**.

  ```
  select sum(price) from fruit;-- 7
  ```

- regr_avgx(T independent, T dependent) → double

  Description: Calculates the average value of the independent variable (*expr2*) of the regression line. After the empty pair (*expr1, expr2*) is removed, the value is *AVG(expr2)*.

  ```
  create table sample_collection(id int,time_cost int,weight decimal(5,2));

  insert into sample_collection values
  (1,5,86.38),
  (2,10,281.17),
  (3,15,89.91),
  (4,20,17.5),
  (5,25,88.76),
  ```

```
(6,30,83.94),
(7,35,44.26),
(8,40,17.4),
(9,45,5.6),
(10,50,145.68);

select regr_avgx(time_cost,weight) from sample_collection;
       _col0
-------------------
 86.06000000000002
(1 row)
```

- regr_avgy(T independent, T dependent) → double

  Description: Calculates the average value of the dependent variable (*expr1*) of the regression line. After the empty pair (*expr1, expr2*) is removed, the value is *AVG(expr1)*.

```
select regr_avgy(time_cost,weight) from sample_collection;
 _col0
-------
  27.5
(1 row)
```

- regr_count(T independent, T dependent) → double

  Description: Returns the non-null logarithm used to fit a linear regression line.

```
select regr_count(time_cost,weight) from sample_collection;
 _col0
-------
    10
(1 row)
```

- regr_r2(T independent, T dependent) → double

  Description: Returns the coefficient of determination for regression.

```
select regr_r2(time_cost,weight) from sample_collection;
       _col0
--------------------
 0.1446739237728169
(1 row)
```

- regr_sxx(T independent, T dependent) → double

  Description: Returns the value of REGR_COUNT(expr1, expr2) * VAR_POP(expr2).

```
select regr_sxx(time_cost,weight) from sample_collection;
       _col0
--------------------
 59284.886600000005
(1 row)
```

- regr_sxy(T independent, T dependent) → double

  Description: Returns the value of REGR_COUNT(expr1, expr2) * COVAR_POP(expr1, expr2).

```
select regr_sxy(time_cost,weight) from sample_collection;
  _col0
----------
 -4205.95
(1 row)
```

- regr_syy(T independent, T dependent) → double

  Description: Returns the value of REGR_COUNT(expr1, expr2) * VAR_POP(expr1).

```
select regr_syy(time_cost,weight) from sample_collection;
 _col0
--------
 2062.5
(1 row)
```

Bitwise Aggregate Function

- bitwise_and_agg(x)

  Description: Uses two's complement to represent the bitwise AND operation on the input field **x**. The return type is **bigint**.

  ```
  select bitwise_and_agg(x) from (values (31),(32)) as t(x);-- 0
  ```

- bitwise_or_agg(x)

  Description: Uses two's complement to represent the bitwise OR operation on the input field **x**. The return type is **bigint**.

  ```
  select bitwise_or_agg(x) from (values (31),(32)) as t(x);-- 63
  ```

Map Aggregate Function

- histogram(x) -> map(K, bigint)

  Description: Returns a map containing the number of occurrences of all input field **x**.

  ```
  select histogram(x),histogram(y) from (values (15,17),(15,18),(15,19),(15,20)) as t(x,y);-- {15=4},
  {17=1, 18=1, 19=1, 20=1}
  ```

- map_agg(key, value) -> map(K, V)

  Description: Returns a map whose input field key and input field value are key-value pairs.

  ```
  select map_agg(name,price) from fruit;-- {apple=2, peach=5}
  ```

- map_union(x(K, V)) -> map(K, V)

  Description: Returns the union of all input maps. If a key appears multiple times in the input set, the corresponding value is any value corresponding to the key in the input set.

  ```
  select map_union(x) from (values (map(array['banana'],array[10.0])),
  (map(array['apple'],array[7.0]))) as t(x);-- {banana=10.0, apple=7.0}
  select map_union(x) from (values (map(array['banana'],array[10.0])),
  (map(array['banana'],array[7.0]))) as t(x);-- {banana=10.0}
  ```

- multimap_agg(key, value) -> map(K, array(V))

  Description: Returns a map consisting of input key-value pairs. Each key can correspond to multiple values.

  ```
  select multimap_agg(key, value) from (values ('apple',7),('apple',8),('apple',8),('lemon',5) ) as
  t(key,value); - {apple=[7, 8, 8], lemon=[5]}
  ```

Approximation Aggregate Function

In actual situations, when we collect statistics on a large amount of data, sometimes we only care about an approximate value instead of a specific value. For example, when we collect statistics on the sales volume of a product, the approximation aggregate function is useful because it uses less memory and CPU resources, in this way, you can obtain data results without any problems, such as overflowing to disks or CPU peaks. This is useful for billions of rows of data calculations.

- approx_median(x) → bigint

  Description: Calculates the median value of a distribution of values.

  ```
  select approx_median(price) from fruit; -- 10.0
  ```

- approx_distinct(x) → bigint

  Description: The return type of this function is **bigint**, which provides an approximate count of **count(distinct x)**. If all inputs are null, **0** is returned.

The errors of all possible values of this function relative to the correct values follow an approximate normal distribution with a standard deviation of 2.3%. It does not guarantee the upper limit of any specific input set error.

```
select approx_distinct(price) from fruit; -- 2
```

- approx_distinct(x, e)→ bigint

  Description: The return type of this function is **bigint**, which provides an approximate count of **count(distinct x)**. If all inputs are null, **0** is returned.

  The errors of all possible values of this function relative to the correct values follow an approximate normal distribution with a standard deviation of less than *e*. It does not guarantee the upper limit of any specific input set error.

  In the current implementation of the function, the value range of e is [0.0040625, 0.26000].

```
select approx_distinct(weight,0.0040625) from sample_collection; -- 10
select approx_distinct(weight,0.26) from sample_collection; -- 8
```

- approx_most_frequent(*buckets*, *value*, *capacity*) → map<[same as value], bigint>

  Description: Approximately collects statistics on the top **buckets** elements that appear most frequently. Less memory will be used for approximate estimation of high-frequency values. A larger **capacity** value indicates a more accurate result, which consumes more memory. The return value of this function is a map that consists of key-value pairs of high-frequency values and their frequencies.

```
SELECT approx_most_frequent(3, x, 15) FROM (values 'A', 'B', 'A', 'C', 'A', 'B', 'C', 'D', 'E') t(x); --  {A=3, B=2, C=2}
SELECT approx_most_frequent(3, x, 100) FROM (values 1, 2, 1, 3, 1, 2, 3, 4, 5) t(x); -- {1=3, 2=2, 3=2}
```

NOTE

The commonly used quantiles are binary, quaternary, decimal, and percentile. This means that the input set is evenly divided into equal parts, and then the value at the corresponding position is found. For example, **approx_percentile(x, 0.5)** is used to find a value that is about 50% of the x value, that is, a 2-quantile value.

- approx_percentile(x, percentage)→[same as x]

  Description: Returns the approximate percentile based on the given percentage. The percentage value must be a constant between 0 and 1 for all input rows.

```
select approx_percentile(x, 0.5) from ( values (2),(3),(7),(8),(9)) as t(x); --7
```

- approx_percentile(x, percentages)→ array<[same as x]>

  Description: Returns an approximate percentile of the x value of all input fields in a given percentage array. Each value in this percentage array must be a constant between 0 and 1 for all input rows.

```
select approx_percentile(x, array[0.1,0.2,0.3,0.5]) from ( values (2),(3),(7),(8),(9)) as t(x); --[2, 3, 3, 7]
```

- approx_percentile(x, w, percentage)→array<[same as x]>

  Description: Returns the approximate percentile of all **x** input values by **percentage**. The weight of each item is **w** and must be a positive number. Set a valid percentile for **x**. The value of **percentage** must range from 0 to 1, and all input rows must be constants.

```
select approx_percentile(x, 5,array[0.1,0.2,0.3,0.5]) from ( values (2),(3),(7),(8),(9)) as t(x); --[2, 3, 3, 7]
```

- approx_percentile(x, w, percentage, accuracy) →[same as x]

Description: Returns the approximate percentile of all **x** input values by **percentage**. The weight of each item is **w** and must be a positive number. Set a valid percentile for **x**. The value of **percentage** must range from 0 to 1, and all input rows must be constants. The maximum progress error of the approximate value is specified by **accuracy**.

```
select approx_percentile(x, 5,0.5,0.97) from ( values (2),(3),(7),(8),(9)) as t(x); --7
```

- approx_percentile(x, w, percentages)→[same as x]

  Description: Returns an approximate percentile of all **x** input values based on each percentage value in the percentage array. The weight of each item is **w** and must be a positive number. Set a valid percentile for **x**. Each element value in the percentage array must range from 0 to 1, and all input rows must be constants.

```
select approx_percentile(x,5, array[0.1,0.2,0.3,0.5]) from ( values (2),(3),(7),(8),(9)) as t(x);  -- [2, 3, 3, 7]
```

NOTE

The preceding **approx_percentile** function also supports the **percentile_approx** function of the same parameter set.

- numeric_histogram(buckets, value, weight)

  Description: Calculates the approximate histogram for all values based on the number of buckets. The width of each item uses **weight**. This algorithm is based on:

```
Yael Ben-Haim and Elad Tom-Tov, "A streaming parallel decision tree algorithm", J. Machine Learning Research 11 (2010), pp. 849--872.
```

  The value of **buckets** must be **bigint**. The values of **value** and **weight** must be numbers.

```
select numeric_histogram(20,x,4) from ( values (2),(3),(7),(8),(9)) as t(x);
_col0
-----------------------------------------------
 {2.0=4.0, 3.0=4.0, 7.0=4.0, 8.0=4.0, 9.0=4.0}
(1 row)
```

- numeric_histogram(buckets, value)

  Description: Compared with **numeric_histogram(buckets, value,weight)**, this function sets **weight** to **1**.

```
select numeric_histogram(20,x) from ( values (2),(3),(7),(8),(9)) as t(x);
_col0
-----------------------------------------------
 {2.0=1.0, 3.0=1.0, 7.0=1.0, 8.0=1.0, 9.0=1.0}
(1 row)
```

Statistical Aggregate Function

- corr(y,x)

  Description: Returns the correlation coefficient of the input value.

```
select corr(y,x) from (values (1,5),(2,6),(3,7),(4,8)) as t(x,y);-- 1.0
```

- covar_pop(y, x)

  Description: Returns the population covariance of the input value.

```
select covar_pop(y,x) from (values (1,5),(2,6),(3,7),(4,8)) as t(x,y); --1.25
```

- *covar_samp(y, x*)

  Description: Returns the sample covariance of the input value.

```
select covar_samp(y,x) from (values (1,5),(2,6),(3,7),(4,8)) as t(x,y);-- 1.6666666
```

- kurtosis(*x*)

  Description: Kurtosis, also called peak-state coefficient, indicates the number of peak values at the average value of the probability density distribution curve, that is, the statistical value that describes the steepness of all value distribution forms in the entire system. Intuitively, kurtosis reflects the sharpness of the peak. This statistic needs to be compared with the normal distribution.

  The kurtosis is defined as the **4th standardized central moment** of the sample.

  The kurtosis of a random variable is the ratio of the fourth-order central moment of the random variable to the square of the variance.

  The calculation formula is as follows:

$$\mathrm{Kurtosis} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^4 \,/\, \mathrm{SD}^4 - 3$$

```
select kurtosis(x) from (values (1),(2),(3),(4)) as t(x); -- -1.1999999999999993
```

- regr_intercept(y, x)

  Description: Returns the linear regression intercept of the input value. **y** is a subordinate value. **x** is an independent value.

```
select regr_intercept(y,x) from (values (1,5),(2,6),(3,7),(4,8)) as t(x,y);-- 4.0
```

- regr_slope(y, x)

  Description: Returns the linear regression slope of the input value. **y** is a subordinate value. **x** is an independent value.

```
select regr_slope(y,x) from (values (1,5),(2,6),(3,7),(4,8)) as t(x,y);--  1.0
```

- skewness(x)

  Description: Returns the skew degree of all input values.

```
select skewness(x) from (values (1),(2),(3),(4)) as t(x); -- 0.0
```

- stddev(x)

  Description: Alias of **stedev_samp()**

- stddev_pop(x)

  Description: Returns the population standard deviation of all input values.

```
select stddev_pop(x) from (values (1),(2),(3),(4)) as t(x);--  1.118033988749895
```

- stddev_samp(x)

  Description: Returns the sample standard deviation of all input values.

```
select stddev_samp(x) from (values (1),(2),(3),(4)) as t(x);--  1.2909944487358056
```

- variance(x)

  Description: Alias of **var_samp()**

- var_pop(x)

  Description: Returns the population variance of all input values.

```
select var_pop(x) from (values (1),(2),(3),(4)) as t(x);-- 1.25
```

- var_samp(x)

  Description: Returns the sample variance of all input values.

```
select var_samp(x) from (values (1),(2),(3),(4)) as t(x);--  1.6666666666666667
```

Lambda Aggregation Function

reduce_agg(*inputValue T*, *initialState S*, *inputFunction(S, T, S)*, *combineFunction(S, S, S)*)

inputFunction is called for each non-null input value. In addition to obtaining the input value, **inputFunction** also obtains the current status, which is **initialState** initially, and then returns the new status. **CombineFunction** is invoked to combine the two states into a new state. Return the final status:

```
SELECT id, reduce_agg(value, 0, (a, b) -> a + b, (a, b) -> a + b)
FROM (
    VALUES
        (1, 3),
        (1, 4),
        (1, 5),
        (2, 6),
        (2, 7)
) AS t(id, value)
GROUP BY id;
-- (1, 12)
-- (2, 13)

SELECT id, reduce_agg(value, 1, (a, b) -> a * b, (a, b) -> a * b)
FROM (
    VALUES
        (1, 3),
        (1, 4),
        (1, 5),
        (2, 6),
        (2, 7)
) AS t(id, value)
GROUP BY id;
-- (1, 60)
-- (2, 42)
```

NOTE

The status value must be Boolean, integer, floating-point, date, time, or interval.

1.8.15 Window Functions

Window functions perform calculations across rows of query results. They are run after the **HAVING** clause but before the **ORDER BY** clause. To call a window function, you need to use the **OVER** clause to specify the special syntax of the window. The window consists of three parts:

- Partition specification, which divides the input rows into different partitions. This is similar to how the **GROUP BY** clause divides rows into different groups in an aggregate function.
- Sorting specification, which determines the order in which the window function will process the input rows
- Window frame, which specifies the sliding window of the row to be processed by the specified function. If no frame is specified, the default value is **RANGE UNBOUNDED PRECEDING**, which is the same as the value of **UNBOUNDEEN PREBODING AND CURRENT ROWGE**. The frame contains all rows from the beginning of the partition to the last peer of the current row. In the absence of **ORDER BY**, all rows are treated as equivalent, so the range between the unbound preamble and the current row is equal to the range between the unbound preamble and the unbound subsequent rows.

For example, in the following query, the information in the **salary** table is sorted according to the salary of employees in each department.

```
-- Create a data table and insert data.
create table salary (dept varchar, userid varchar, sal double);
 insert into salary values ('d1','user1',1000),('d1','user2',2000),('d1','user3',3000),('d2','user4',4000),
('d2','user5',5000);

--Query data.
select dept,userid,sal,rank() over (partition by dept order by sal desc) as rnk from salary order by
dept,rnk;
dept | userid |  sal   | rnk
------|--------|--------|-----
 d1   | user3  | 3000.0 |   1
 d1   | user2  | 2000.0 |   2
 d1   | user1  | 1000.0 |   3
 d2   | user5  | 5000.0 |   1
 d2   | user4  | 4000.0 |   2
```

Aggregate Functions

All aggregate functions can be used as window functions by adding **over** clauses. The aggregate function operates on each row of records in the current window framework.

The following query generates the rolling sum of the order price calculated by each employee by day:

```
select dept,userid,sal,sum(sal) over (partition by dept order by sal desc) as rolling_sum from salary order by
dept,userid,sal;
dept | userid |  sal   | rolling_sum
------|--------|--------|-------------
 d1   | user1  | 1000.0 |      6000.0
 d1   | user2  | 2000.0 |      5000.0
 d1   | user3  | 3000.0 |      3000.0
 d2   | user4  | 4000.0 |      9000.0
 d2   | user5  | 5000.0 |      5000.0
(5 rows)
```

Ranking Functions

- cume_dist()→ bigint

  Description: Number of lines less than or equal to the current value/Total number of lines in the group – For example, calculate the proportion of the number of employees whose salary is less than or equal to the current salary to the total number of employees.

```
 --Query Example
SELECT dept, userid, sal, CUME_DIST() OVER(ORDER BY sal) AS rn1, CUME_DIST() OVER(PARTITION
BY dept ORDER BY sal) AS rn2 FROM salary;
dept | userid |  sal   | rn1 |         rn2
------|--------|--------|-----|--------------------
 d2   | user4  | 4000.0 | 0.8 |                0.5
 d2   | user5  | 5000.0 | 1.0 |                1.0
 d1   | user1  | 1000.0 | 0.2 | 0.3333333333333333
 d1   | user2  | 2000.0 | 0.4 | 0.6666666666666666
 d1   | user3  | 3000.0 | 0.6 |                1.0
(5 rows)
```

- dense_rank()→ bigint

  Description: Ranking of the returned value in a group of values. This is similar to **rank ()**. The difference is that the tie value does not generate gaps in the sequence.

- ntile(*n*)→ bigint

  Description: Divides packet data into n fragments in sequence and returns the current fragment value. NTILE does not support ROWS BETWEEN. For example, if **NTILE(2) OVER(PARTITION BY cookieid ORDER BY createtime ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)** fragments are not evenly distributed, the distribution of the first fragment is added by default.

```
--Create a table and insert data into the table.
create table cookies_log (cookieid varchar,createtime  date,pv int);
insert into cookies_log values
    ('cookie1',date '2020-07-10',1),
    ('cookie1',date '2020-07-11',5),
    ('cookie1',date '2020-07-12',7),
    ('cookie1',date '2020-07-13',3),
    ('cookie1',date '2020-07-14',2),
    ('cookie1',date '2020-07-15',4),
    ('cookie1',date '2020-07-16',4),
    ('cookie2',date '2020-07-10',2),
    ('cookie2',date '2020-07-11',3),
    ('cookie2',date '2020-07-12',5),
        ('cookie2',date '2020-07-13',6),
        ('cookie2',date '2020-07-14',3),
        ('cookie2',date '2020-07-15',9),
    ('cookie2',date '2020-07-16',7);
-- Query results.
SELECT cookieid,createtime,pv,
NTILE(2) OVER(PARTITION BY cookieid ORDER BY createtime) AS rn1,-- divides data into two
fragments in a group.
NTILE(3) OVER(PARTITION BY cookieid ORDER BY createtime) AS rn2,  -- divides data into three
fragments in a group.
NTILE(4) OVER(ORDER BY createtime) AS rn3   --divides all data into four fragments.
FROM cookies_log
ORDER BY cookieid,createtime;
cookieid  | createtime | pv | rn1 | rn2 | rn3
----------|------------|----|-----|-----|-----
 cookie1  | 2020-07-10 |  1 |   1 |   1 |   1
 cookie1  | 2020-07-11 |  5 |   1 |   1 |   1
 cookie1  | 2020-07-12 |  7 |   1 |   1 |   2
 cookie1  | 2020-07-13 |  3 |   1 |   2 |   2
 cookie1  | 2020-07-14 |  2 |   2 |   2 |   3
 cookie1  | 2020-07-15 |  4 |   2 |   3 |   4
 cookie1  | 2020-07-16 |  4 |   2 |   3 |   4
 cookie2  | 2020-07-10 |  2 |   1 |   1 |   1
 cookie2  | 2020-07-11 |  3 |   1 |   1 |   1
 cookie2  | 2020-07-12 |  5 |   1 |   1 |   2
 cookie2  | 2020-07-13 |  6 |   1 |   2 |   2
 cookie2  | 2020-07-14 |  3 |   2 |   2 |   3
 cookie2  | 2020-07-15 |  9 |   2 |   3 |   3
 cookie2  | 2020-07-16 |  7 |   2 |   3 |   4
(14 rows)
```

- percent_rank()→ double

  Description: Rankings of return values in percentage within a set of values. The result is *(r-1)/(n-1)*, where r is the **rank ()** of the row and *n* is the total number of rows in the window partition.

```
SELECT dept,userid,sal,
PERCENT_RANK() OVER(ORDER BY sal) AS rn1, -- in a group
RANK() OVER(ORDER BY sal) AS rn11,          -- RANK value in a group
SUM(1) OVER(PARTITION BY NULL) AS rn12,     --Total number of lines in the group
PERCENT_RANK() OVER(PARTITION BY dept ORDER BY sal) AS rn2
from salary;
 dept | userid |  sal   | rn1  | rn11 | rn12 | rn2
------|--------|--------|------|------|------|-----
 d2   | user4  | 4000.0 | 0.75 |    4 |    5 | 0.0
 d2   | user5  | 5000.0 |  1.0 |    5 |    5 | 1.0
 d1   | user1  | 1000.0 |  0.0 |    1 |    5 | 0.0
 d1   | user2  | 2000.0 | 0.25 |    2 |    5 | 0.5
 d1   | user3  | 3000.0 |  0.5 |    3 |    5 | 1.0
(5 rows)
```

- rank() → bigint

  Description: Ranking of the returned value in a group of values. The level is 1 plus the number of rows that are not equal to the current row. Therefore, the average value in the sort will create a gap in the sequence. Ranks each window partition.

```
SELECT
cookieid,
createtime,
pv,
RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn1,
DENSE_RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn2,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv DESC) AS rn3
FROM cookies_log
WHERE cookieid = 'cookie1';
 cookieid | createtime | pv | rn1 | rn2 | rn3
----------|------------|----|-----|-----|-----
 cookie1  | 2020-07-12 |  7 |   1 |   1 |   1
 cookie1  | 2020-07-11 |  5 |   2 |   2 |   2
 cookie1  | 2020-07-15 |  4 |   3 |   3 |   3
 cookie1  | 2020-07-16 |  4 |   3 |   3 |   4
 cookie1  | 2020-07-13 |  3 |   5 |   4 |   5
 cookie1  | 2020-07-14 |  2 |   6 |   5 |   6
 cookie1  | 2020-07-10 |  1 |   7 |   6 |   7
(7 rows)
```

- row_number() → bigint

  Description: Starting from 1, the sequence of records in a group is generated in sequence. For example, generate the daily **pv** ranking in descending order. There are many application scenarios for **ROW_NUMBER()**. For another example, obtain the record that ranks first in a group, or obtain the first **refer** in a session.

```
SELECT cookieid, createtime, pv, ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv desc)
AS rn from cookies_log;
  cookieid | createtime | pv | rn
 ----------|------------- |----|----
 cookie2  | 2020-07-15 |  9 |  1
 cookie2  | 2020-07-16 |  7 |  2
 cookie2  | 2020-07-13 |  6 |  3
 cookie2  | 2020-07-12 |  5 |  4
 cookie2  | 2020-07-14 |  3 |  5
 cookie2  | 2020-07-11 |  3 |  6
 cookie2  | 2020-07-10 |  2 |  7
 cookie1  | 2020-07-12 |  7 |  1
 cookie1  | 2020-07-11 |  5 |  2
 cookie1  | 2020-07-15 |  4 |  3
 cookie1  | 2020-07-16 |  4 |  4
 cookie1  | 2020-07-13 |  3 |  5
 cookie1  | 2020-07-14 |  2 |  6
 cookie1  | 2020-07-10 |  1 |  7
(14 rows)
```

Value Functions

Generally, the null value must be considered. If **IGNORE NULLS** is specified, all rows containing x whose value is null will be excluded. If the value of **x** in all rows is null, the default value is returned. Otherwise, null is returned.

```
Data Preparation
create table cookie_views( cookieid varchar,createtime timestamp,url varchar);
insert into cookie_views values
('cookie1',timestamp '2020-07-10 10:00:02','url20'),
('cookie1',timestamp '2020-07-10 10:00:00','url10'),
('cookie1',timestamp '2020-07-10 10:03:04','url13'),
('cookie1',timestamp '2020-07-10 10:50:05','url60'),
('cookie1',timestamp '2020-07-10 11:00:00','url70'),
('cookie1',timestamp '2020-07-10 10:10:00','url40'),
('cookie1',timestamp '2020-07-10 10:50:01','url50'),
('cookie2',timestamp '2020-07-10 10:00:02','url23'),
('cookie2',timestamp '2020-07-10 10:00:00','url11'),
('cookie2',timestamp '2020-07-10 10:03:04','url33'),
('cookie2',timestamp '2020-07-10 10:50:05','url66'),
('cookie2',timestamp '2020-07-10 11:00:00','url77'),
('cookie2',timestamp '2020-07-10 10:10:00','url47'),
('cookie2',timestamp '2020-07-10 10:50:01','url55');
```

- first_value(*x*)→ [same as input]

  Description: Returns the first value of the window.

```
SELECT cookieid,
createtime,
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
FIRST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS first1
FROM cookie_views;
cookieid |       createtime        |  url  | rn | first1
----------|-------------------------|-------|----|--------
 cookie1  | 2020-07-10 10:00:00.000 | url10 |  1 | url10
 cookie1  | 2020-07-10 10:00:02.000 | url20 |  2 | url10
 cookie1  | 2020-07-10 10:03:04.000 | url13 |  3 | url10
 cookie1  | 2020-07-10 10:10:00.000 | url40 |  4 | url10
 cookie1  | 2020-07-10 10:50:01.000 | url50 |  5 | url10
 cookie1  | 2020-07-10 10:50:05.000 | url60 |  6 | url10
 cookie1  | 2020-07-10 11:00:00.000 | url70 |  7 | url10
 cookie2  | 2020-07-10 10:00:00.000 | url11 |  1 | url11
 cookie2  | 2020-07-10 10:00:02.000 | url23 |  2 | url11
 cookie2  | 2020-07-10 10:03:04.000 | url33 |  3 | url11
 cookie2  | 2020-07-10 10:10:00.000 | url47 |  4 | url11
 cookie2  | 2020-07-10 10:50:01.000 | url55 |  5 | url11
 cookie2  | 2020-07-10 10:50:05.000 | url66 |  6 | url11
 cookie2  | 2020-07-10 11:00:00.000 | url77 |  7 | url11
(14 rows)
```

- last_value(*x*)→ [same as input]

  Description: Returns the last value of the window.

```
SELECT cookieid,createtime,url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LAST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS last1
FROM cookie_views;
 cookieid |       createtime        |  url  | rn | last1
----------|-------------------------|-------|----|-------
 cookie2  | 2020-07-10 10:00:00.000 | url11 |  1 | url11
 cookie2  | 2020-07-10 10:00:02.000 | url23 |  2 | url23
 cookie2  | 2020-07-10 10:03:04.000 | url33 |  3 | url33
 cookie2  | 2020-07-10 10:10:00.000 | url47 |  4 | url47
 cookie2  | 2020-07-10 10:50:01.000 | url55 |  5 | url55
 cookie2  | 2020-07-10 10:50:05.000 | url66 |  6 | url66
 cookie2  | 2020-07-10 11:00:00.000 | url77 |  7 | url77
 cookie1  | 2020-07-10 10:00:00.000 | url10 |  1 | url10
 cookie1  | 2020-07-10 10:00:02.000 | url20 |  2 | url20
 cookie1  | 2020-07-10 10:03:04.000 | url13 |  3 | url13
 cookie1  | 2020-07-10 10:10:00.000 | url40 |  4 | url40
 cookie1  | 2020-07-10 10:50:01.000 | url50 |  5 | url50
 cookie1  | 2020-07-10 10:50:05.000 | url60 |  6 | url60
 cookie1  | 2020-07-10 11:00:00.000 | url70 |  7 | url70
(14 rows)
```

- nth_value(*x*, *offset*)→ [same as input]

  Description: Returns the value of the specified offset from the beginning of the window. The offset starts from 1. The offset can be any scalar expression.

If the offset is null or greater than the number of values in the window, null is returned. The offset cannot be 0 or a negative number.

```
SELECT cookieid,createtime,url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
NTH_VALUE(url,3) OVER(PARTITION BY cookieid ORDER BY createtime) AS last1
FROM cookie_views;
cookieid |       createtime        |  url  | rn | last1
----------|-------------------------|-------|----|-------
 cookie1  | 2020-07-10 10:00:00.000 | url10 |  1 | NULL
 cookie1  | 2020-07-10 10:00:02.000 | url20 |  2 | NULL
 cookie1  | 2020-07-10 10:03:04.000 | urll3 |  3 | urll3
 cookie1  | 2020-07-10 10:10:00.000 | url40 |  4 | urll3
 cookie1  | 2020-07-10 10:50:01.000 | url50 |  5 | urll3
 cookie1  | 2020-07-10 10:50:05.000 | url60 |  6 | urll3
 cookie1  | 2020-07-10 11:00:00.000 | url70 |  7 | urll3
 cookie2  | 2020-07-10 10:00:00.000 | url11 |  1 | NULL
 cookie2  | 2020-07-10 10:00:02.000 | url23 |  2 | NULL
 cookie2  | 2020-07-10 10:03:04.000 | url33 |  3 | url33
 cookie2  | 2020-07-10 10:10:00.000 | url47 |  4 | url33
 cookie2  | 2020-07-10 10:50:01.000 | url55 |  5 | url33
 cookie2  | 2020-07-10 10:50:05.000 | url66 |  6 | url33
 cookie2  | 2020-07-10 11:00:00.000 | url77 |  7 | url33
(14 rows)
```

- lead(*x*[, *offset*[, *default_value*]]) → [same as input]

  Description: Returns the value of the offset row after the current row in the window partition. The offset starts from 0, that is, the current line. The offset can be any scalar expression. The default offset is **1**. If the offset is null, null is returned. If the offset points to a row that is not in the partition, **default_value** is returned. If it is not specified, null is returned. The **lead()** function requires that the window sequence be specified. Do not specify a window frame.

```
SELECT cookieid,createtime,url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LEAD(createtime,1,timestamp '2020-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY
createtime) AS next_1_time,
LEAD(createtime,2) OVER(PARTITION BY cookieid ORDER BY createtime) AS next_2_time
FROM cookie_views;
 cookieid |       createtime        |  url  | rn |       next_1_time       |       next_2_time
----------|-------------------------|-------|----|-------------------------|-------------------------
 cookie2  | 2020-07-10 10:00:00.000 | url11 |  1 | 2020-07-10 10:00:02.000 | 2020-07-10 10:03:04.000
 cookie2  | 2020-07-10 10:00:02.000 | url23 |  2 | 2020-07-10 10:03:04.000 | 2020-07-10 10:10:00.000
 cookie2  | 2020-07-10 10:03:04.000 | url33 |  3 | 2020-07-10 10:10:00.000 | 2020-07-10 10:50:01.000
 cookie2  | 2020-07-10 10:10:00.000 | url47 |  4 | 2020-07-10 10:50:01.000 | 2020-07-10 10:50:05.000
 cookie2  | 2020-07-10 10:50:01.000 | url55 |  5 | 2020-07-10 10:50:05.000 | 2020-07-10 11:00:00.000
 cookie2  | 2020-07-10 10:50:05.000 | url66 |  6 | 2020-07-10 11:00:00.000 | NULL
 cookie2  | 2020-07-10 11:00:00.000 | url77 |  7 | 2020-01-01 00:00:00.000 | NULL
 cookie1  | 2020-07-10 10:00:00.000 | url10 |  1 | 2020-07-10 10:00:02.000 | 2020-07-10 10:03:04.000
 cookie1  | 2020-07-10 10:00:02.000 | url20 |  2 | 2020-07-10 10:03:04.000 | 2020-07-10 10:10:00.000
 cookie1  | 2020-07-10 10:03:04.000 | urll3 |  3 | 2020-07-10 10:10:00.000 | 2020-07-10 10:50:01.000
 cookie1  | 2020-07-10 10:10:00.000 | url40 |  4 | 2020-07-10 10:50:01.000 | 2020-07-10 10:50:05.000
 cookie1  | 2020-07-10 10:50:01.000 | url50 |  5 | 2020-07-10 10:50:05.000 | 2020-07-10 11:00:00.000
 cookie1  | 2020-07-10 10:50:05.000 | url60 |  6 | 2020-07-10 11:00:00.000 | NULL
 cookie1  | 2020-07-10 11:00:00.000 | url70 |  7 | 2020-01-01 00:00:00.000 | NULL
(14 rows)
```

- lag(*x*[, *offset*[, *default_value*]]) → [same as input]

  Description: Returns the value of the offset row before the current row in the window partition. The offset starts from **0**, that is, the current row. The offset can be any scalar expression. The default offset is **1**. If the offset is null, null is returned. If the offset points to a row that is not in the partition, **default_value** is returned. If this parameter is not specified, **null** is returned. The **lag()** function requires that the window sequence be specified and the window frame cannot be specified.

```
 SELECT cookieid, createtime, url, ROW_NUMBER() OVER(PARTITION BY cookieid  ORDER BY
createtime) AS rn,
   LAG(createtime,1, timestamp '2020-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY
createtime)  AS last_1_time,
   LAG(createtime,2) OVER(PARTITION BY cookieid ORDER BY createtime)  AS last_2_time
    FROM cookie_views;

cookieid |       createtime         |  url  | rn |       last_1_time       |       last_2_time
----------|-------------------------|-------|----|-------------------------|-----------------------
 cookie2  | 2020-07-10 10:00:00.000 | url11 |  1 | 2020-01-01 00:00:00.000 | NULL
 cookie2  | 2020-07-10 10:00:02.000 | url23 |  2 | 2020-07-10 10:00:00.000 | NULL
 cookie2  | 2020-07-10 10:03:04.000 | url33 |  3 | 2020-07-10 10:00:02.000 | 2020-07-10 10:00:00.000
 cookie2  | 2020-07-10 10:10:00.000 | url47 |  4 | 2020-07-10 10:03:04.000 | 2020-07-10 10:00:02.000
 cookie2  | 2020-07-10 10:50:01.000 | url55 |  5 | 2020-07-10 10:10:00.000 | 2020-07-10 10:03:04.000
 cookie2  | 2020-07-10 10:50:05.000 | url66 |  6 | 2020-07-10 10:50:01.000 | 2020-07-10 10:10:00.000
 cookie2  | 2020-07-10 11:00:00.000 | url77 |  7 | 2020-07-10 10:50:05.000 | 2020-07-10 10:50:01.000
 cookie1  | 2020-07-10 10:00:00.000 | url10 |  1 | 2020-01-01 00:00:00.000 | NULL
 cookie1  | 2020-07-10 10:00:02.000 | url20 |  2 | 2020-07-10 10:00:00.000 | NULL
 cookie1  | 2020-07-10 10:03:04.000 | urll3 |  3 | 2020-07-10 10:00:02.000 | 2020-07-10 10:00:00.000
 cookie1  | 2020-07-10 10:10:00.000 | url40 |  4 | 2020-07-10 10:03:04.000 | 2020-07-10 10:00:02.000
 cookie1  | 2020-07-10 10:50:01.000 | url50 |  5 | 2020-07-10 10:10:00.000 | 2020-07-10 10:03:04.000
 cookie1  | 2020-07-10 10:50:05.000 | url60 |  6 | 2020-07-10 10:50:01.000 | 2020-07-10 10:10:00.000
 cookie1  | 2020-07-10 11:00:00.000 | url70 |  7 | 2020-07-10 10:50:05.000 | 2020-07-10 10:50:01.000
(14 rows)
```

1.8.16 Array Functions and Operators

Subscript Operator: []

Description: Subscript operators are used to access elements in an array and create indexes starting from **1**.

```
select myarr[5] from (values array [1,4,6,78,8,9],array[2,4,6,8,10,12]) as t(myarr);
 _col0
-------
     8
    10
```

Concatenation Operator: ||

|| Operators are used to concatenate arrays or values of the same type.

```
SELECT ARRAY[1] || ARRAY[2];
 _col0
--------
 [1, 2]
(1 row)

SELECT ARRAY[1] || 2;
 _col0
--------
 [1, 2]
(1 row)

SELECT 2 || ARRAY[1];
 _col0
--------
 [2, 1]
(1 row)
```

Array Function

Subscript operator: []

The subscript operator **[]** is used to obtain the value of the corresponding position in the array.

```
SELECT ARRAY[5,3,41,6][1] AS first_element; -- 5
```

Concatenation Operator: ||

The || operator is used to concatenate an array with an array or an element of the same type:

```
SELECT ARRAY[1]||ARRAY[2];-- [1, 2]
SELECT ARRAY[1]||2; -- [1, 2]
SELECT 2||ARRAY[1];-- [2, 1]
```

- array_contains(*x*, *element*) → boolean

  Description: Returns **true** if the array **x** contains elements.

  ```
  select array_contains (array[1,2,3,34,4],4); -- true
  ```

- all_match(array(T), function(T, boolean)) → boolean

  Description: Returns whether all elements of an array satisfy the given assertion function. If all elements satisfy the assertion function or the array is empty, **true** is returned. If one or more elements do not satisfy the assertion function, **false** is returned. If the assertion function is not satisfied by one or more elements, it returns null, and the return value of the **all_match** function is also null.

  ```
  select all_match(a, x-> true) from (values array[]) t(a); -- true
  select all_match(a, x-> x>2) from (values array[4,5,7]) t(a); -- true
  select all_match(a, x-> x>2) from (values array[1,5,7]) t(a); -- false
  select all_match(a, x-> x>1) from (values array[NULL, NULL ,NULL]) t(a); --NULL
  ```

- any_match(array(T), function(T, boolean)) → boolean

  Description: Returns whether an array has elements that satisfy the assertion function. If one or more elements satisfy the assertion function, **true** is returned. If all elements cannot satisfy the assertion function or the array is empty, **false** is returned. If the assertion function is not satisfied by one or more elements, it returns null, and the return value of the **all_match** function is also null.

  ```
  select any_match(a, x-> true) from (values array[]) t(a); -- false
  select any_match(a, x-> x>2) from (values array[0,1,2]) t(a); -- false
  select any_match(a, x-> x>2) from (values array[1,5,7]) t(a); -- true
  select any_match(a, x-> x>1) from (values array[NULL, NULL ,NULL]) t(a); -- NULL
  ```

- array_distinct(*x*) → array

  Description: Outputs the deduplicated array **x**.

  ```
  select  array_distinct(array [1,1,1,1,1,1,3,3,3,33,4,5,6,6,6,6]);--  [1, 3, 33, 4, 5, 6]
  ```

- array_intersect(*x*, *y*) → array

  Description: Returns the intersection of two arrays after deduplication.

  ```
  select array_intersect(array [1,3,5,7,9],array [1,2,3,4,5]);
    _col0
  -----------
   [1, 3, 5]
  (1 row)
  ```

- array_union(*x*, *y*) → array

  Description: Returns the union of two arrays.

  ```
  select array_union(array [1,3,5,7,9],array [1,2,3,4,5]);
          _col0
  -----------------------
   [1, 3, 5, 7, 9, 2, 4]
  (1 row)
  ```

- array_except(*x*, *y*) → array

  Description: Returns an array of deduplicated elements that are in **x** but not in **y**.

  ```
  select array_except(array [1,3,5,7,9],array [1,2,3,4,5]);
   _col0
  --------
   [7, 9]
  (1 row)
  ```

- array_join(*x*, *delimiter*, *null_replacement*) → varchar

  Description: Concatenates elements of a given array **x** with delimiters and replaces the null value in **x** with optional characters.

  ```
  select array_join(array[1,2,3,null,5,6],'|','0');--  1|2|3|0|5|6
  ```

- array_max(*x*) → x

  Description: Returns the maximum value of array **x**.

  ```
  select array_max(array[2,54,67,132,45]);  -- 132
  ```

- array_min(*x*) → x

  Description: Returns the minimum value of array **x**.

  ```
  select array_min(array[2,54,67,132,45]);  -- 2
  ```

- array_position(x,element) → bigint

  Description: Returns the first occurrence position of an element in the array **x**. If the element is not found, **0** is returned.

  ```
  select array_position(array[2,3,4,5,1,2,3],3); -- 2
  ```

- array_remove(*x*, *element*) → array

  Description: Removes elements whose values are elements from array **x** and returns the elements.

  ```
  select array_remove(array[2,3,4,5,1,2,3],3); --  [2, 4, 5, 1, 2]
  ```

- array_sort(*x*) → array

  Sorts and returns the array **x**. The elements of **x** must be sortable. The empty element is placed at the end of the returned array.

  ```
  select array_sort(array[2,3,4,5,1,2,3]); -- [1, 2, 2, 3, 3, 4, 5]
  ```

- array_sort(*array(T)*, *function(T*, *T*, *int))*

  Description: Sorts and returns the array based on the given comparator function. The comparator will use two parameters that can be null, representing the two elements of the array that can be null. When the first element that can be empty is less than, equal to, or greater than the second element that can be empty, **-1**, **0**, or **1** is returned. If the comparator function returns other values, including **null**, the query fails and an error is thrown.

  ```
  SELECT array_sort(ARRAY [3, 2, 5, 1, 2], (x, y) -> IF(x < y, 1, IF(x = y, 0, -1)));
   _col0
  --------
  [5, 3, 2, 2, 1]
  (1 row)

  SELECT array_sort(ARRAY ['bc', 'ab', 'dc'], (x, y) -> IF(x < y, 1, IF(x = y, 0, -1)));
   _col0
  --------
  [dc, bc, ab]
  (1 row)

  -- The null value is placed in the front, and other values are sorted in descending order.
  SELECT array_sort(ARRAY [3, 2, null, 5, null, 1, 2],
                  (x, y) -> CASE WHEN x IS NULL THEN -1
  ```

```
                              WHEN y IS NULL THEN 1
                              WHEN x < y THEN 1
                              WHEN x = y THEN 0
                              ELSE -1 END);
 _col0
--------
[null, null, 5, 3, 2, 2, 1]
(1 row)

-- The null value is placed in the back, and other values are sorted in descending order.
SELECT array_sort(ARRAY [3, 2, null, 5, null, 1, 2],
                (x, y) -> CASE WHEN x IS NULL THEN 1
                              WHEN y IS NULL THEN -1
                              WHEN x < y THEN 1
                              WHEN x = y THEN 0
                              ELSE -1 END);
 _col0
--------
[5, 3, 2, 2, 1, null, null]
(1 row)

-- Sorting by character string length:
SELECT array_sort(ARRAY ['a', 'abcd', 'abc'],
                 (x, y) -> IF(length(x) < length(y), -1,
                            IF(length(x) = length(y), 0, 1)));
 _col0
--------
[a, abc, abcd]
(1 row)

-- Sorting by array length:
SELECT array_sort(ARRAY [ARRAY[2, 3, 1], ARRAY[4, 2, 1, 4], ARRAY[1, 2]],
                 (x, y) -> IF(cardinality(x) < cardinality(y),
                            -1,
                            IF(cardinality(x) = cardinality(y), 0, 1)));
 _col0
--------
[[1, 2], [2, 3, 1], [4, 2, 1, 4]]
(1 row)
```

- arrays_overlap(*x*, *y*)

  Description: Returns **true** if two arrays have common non-null elements. Otherwise, **false** is returned.

  ```
  select arrays_overlap(array[1,2,3],array[3,4,5]);-- true
  select arrays_overlap(array[1,2,3],array[4,5,6]);-- false
  ```

- cardinality(*x*)

  Description: Returns the capacity of array **x**.

  ```
  select cardinality(array[1,2,3,4,5,6]); --6
  ```

- concat(*array1*, *array2*, ..., *arrayN*)

  Description: Provides the same function as the SQL standard connection operator (**||**).

- combinations(*array(T)*, *n) -> array(array(T)*)

  Description: Returns *n* element subgroups of the input array. If the input array has no duplicate items, the combination returns a subset of *n* elements:

  ```
  SELECT combinations(ARRAY['foo', 'bar', 'baz'], 2); -- [[foo, bar], [foo, baz], [bar, baz]]

  SELECT combinations(ARRAY[1, 2, 3], 2); -- [[1, 2], [1, 3], [2, 3]]

  SELECT combinations(ARRAY[1, 2, 2], 2); -- [[1, 2], [1, 2], [2, 2]]
  ```

  The subgroups and the elements in the subgroups, though not specified, are ordered. The value of **n** cannot be greater than 5, and the maximum number of generated subgroups cannot exceed 100000.

- contains(*x*, *element*)

  Description: Returns **true** if the array **x** contains elements.

  ```
  select contains(array[1,2,3,34,4],4); -- true
  ```

- element_at(*array(E)*, *index*)

  Description: Returns the elements of an array at the given index. If the value of *index* is greater than **0**, this function provides the same function as the SQL standard subscript operator (**[]**). However, when this function accesses an index whose length is greater than the array length, **null** is returned, and the subscript operator fails in this case. If the *index* is less than **0**, **element_at** accesses elements from the last to the first.

  ```
  select element_at(array['a','b','c','d','e'],3); -- c
  ```

- filter(*array(T)*, *function(T*, *boolean)) -> array(T*)

  Description: Deletes an array consisting of elements whose operation result is **true**.

  ```
  SELECT filter(ARRAY [], x -> true); -- []

  SELECT filter(ARRAY [5, -6, NULL, 7], x -> x > 0); -- [5, 7]

  SELECT filter(ARRAY [5, NULL, 7, NULL], x -> x IS NOT NULL); -- [5, 7]
  ```

- flatten(*x*)

  Description: Expands **array(array(T))** to **array(T)** in series.

- ngrams(*array(T)*, *n) -> array(array(T)*)

  Description: Returns the n-gram (subsequence of *n* adjacent elements) of an array. The sequence of n-grams in the result is not specified.

  ```
  SELECT ngrams(ARRAY['foo', 'bar', 'baz', 'foo'], 2); -- [[foo, bar], [bar, baz], [baz, foo]]

  SELECT ngrams(ARRAY['foo', 'bar', 'baz', 'foo'], 3); -- [[foo, bar, baz], [bar, baz, foo]]

  SELECT ngrams(ARRAY['foo', 'bar', 'baz', 'foo'], 4); -- [[foo, bar, baz, foo]]

  SELECT ngrams(ARRAY['foo', 'bar', 'baz', 'foo'], 5); -- [[foo, bar, baz, foo]]

  SELECT ngrams(ARRAY[1, 2, 3, 4], 2); -- [[1, 2], [2, 3], [3, 4]]
  ```

- none_match(*array(T)*, *function(T*, *boolean)*)

  Description: Returns whether an array has no elements matching the given predicate. If no element matches the predicate, **true** is returned (a special case is when the array is empty). The value is **false** if one or more elements match. The value is **null** if the predicate function returns **null** for one or more elements and **false** for all other elements.

- reduce(*array(T)*, *initialState S*, *inputFunction(S*, *T*, *S)*, *outputFunction(S*, *R)*)

  Returns a single value reduced from an array. **inputFunction** is called for each element in the array in sequence. In addition to getting the element, **inputFunction** also gets the current state, initially **initialState**, and then returns the new state. **outputFunction** will be called to convert the final state to a result value. It may be a constant function (**i-> i**).

  ```
  SELECT reduce(ARRAY [], 0, (s, x) -> s + x, s -> s); -- 0
  SELECT reduce(ARRAY [5, 20, 50], 0, (s, x) -> s + x, s -> s); -- 75
  SELECT reduce(ARRAY [5, 20, NULL, 50], 0, (s, x) -> s + x, s -> s); -- NULL
  SELECT reduce(ARRAY [5, 20, NULL, 50], 0, (s, x) -> s + COALESCE(x, 0), s -> s); -- 75
  SELECT reduce(ARRAY [5, 20, NULL, 50], 0, (s, x) -> IF(x IS NULL, s, s + x), s -> s); -- 75
  SELECT reduce(ARRAY [2147483647, 1], CAST (0 AS BIGINT), (s, x) -> s + x, s -> s); -- 2147483648
  SELECT reduce(ARRAY [5, 6, 10, 20], CAST(ROW(0.0, 0) AS ROW(sum DOUBLE, count INTEGER)),(s,
  x) -> CAST(ROW(x + s.sum, s.count + 1) AS ROW(sum DOUBLE, count INTEGER)), s -> IF(s.count = 0,
  NULL, s.sum / s.count)); -- 10.25
  ```

- repeat(*element*, *count*)

  Description: Number of times that the element is repeatedly output. The value is filled in the array.

  ```
  select repeat(4,5);-- [4, 4, 4, 4, 4]
  ```

- reverse(*x*)

  Description: Populates the returned array with array elements in reverse order.

  ```
  select reverse(array[1,2,3,4,5]); --[5, 4, 3, 2, 1]
  ```

- sequence(start, *stop)*

  Description: Outputs an array from *start* to *stop*. If the value of *start* is not greater than that of *stop*, the value is incremented by 1 each time. Otherwise, the value is decremented by 1 each time.

  The data type of *start* and *stop* can also be date or timestamp, which increases or decreases by one day.

  ```
  select sequence(5,1);
        _col0
  -----------------
   [5, 4, 3, 2, 1]
  (1 row)

  select sequence(5,10);
          _col0
  ---------------------
   [5, 6, 7, 8, 9, 10]
  (1 row)
  ```

- sequence(*start*, *stop*, *step)*→ array

  Description: Outputs from *start* to *stop* with a *step* length.

  ```
  select sequence(1,30,5);--  [1, 6, 11, 16, 21, 26]
  ```

- shuffle(*x*)→ array

  Description: Gets a new array based on the given array randomization.

  ```
  select shuffle(array[1,2,3,4,5]);-- [1, 5, 4, 2, 3]
  select shuffle(array[1,2,3,4,5]);--  [2, 1, 3, 5, 4]
  ```

- size(*x*) → bigint

  Description: Returns the capacity of array **x**.

  ```
  select size(array[1,2,3,4,5,6]); --6
  ```

- slice(*x*, *start*, *length*) → array

  Description: Subset array x starts from the beginning of the index (if the *start* is negative, the array starts from the end) and its length is *length*.

  ```
  select slice(array[1,2,3,4,5,6],2,3);-- [2, 3, 4]
  ```

- sort_array(x)

  For details, see **array_sort(x)**.

- trim_array(*x*, *n*) → array

  Description: Deletes the last **n** elements in an array.

  ```
  SELECT trim_array(ARRAY[1, 2, 3, 4], 1); -- [1, 2, 3]
  SELECT trim_array(ARRAY[1, 2, 3, 4], 2); -- [1, 2]
  ```

- transform(*array(T)*, *function(T*, *U)) -> array(U*)

  Description: Returns an array that is the result of applying the function to each element of the array.

  ```
  SELECT transform(ARRAY [], x -> x + 1); -- []
  SELECT transform(ARRAY [5, 6], x -> x + 1); -- [6, 7]
  ```

```
SELECT transform(ARRAY [5, NULL, 6], x -> COALESCE(x, 0) + 1); -- [6, 1, 7]
SELECT transform(ARRAY ['x', 'abc', 'z'], x -> x || '0'); -- [x0, abc0, z0]
SELECT transform(ARRAY [ARRAY [1, NULL, 2], ARRAY[3, NULL]], a -> filter(a, x -> x IS NOT NULL));
-- [[1, 2], [3]]
```

- zip(*array1, array2[, ...]) -> array(row*)

  Description: Merges a given array into a single row array by element. The *M*th element of the *N*th independent variable is the *N*th field of the *M*th output element. If the parameter length is uneven, the missing value is filled with **null**.

  ```
  SELECT zip(ARRAY[1, 2], ARRAY['1b', null, '3b']); --  [{1, 1b}, {2, NULL}, {NULL, 3b}]
  ```

- zip_with(*array(T)*, *array(U)*, *function(T*, *U*, *R)) -> array(R*)

  Description: Combines two given arrays one by one into a single array using functions. If an array is short, a null value is added to the end of the function to match the length of the longer array.

  ```
  SELECT zip_with(ARRAY[1, 3, 5], ARRAY['a', 'b', 'c'], (x, y) -> (y, x)); -- [{a, 1}, {b, 3}, {c, 5}]

  SELECT zip_with(ARRAY[1, 2], ARRAY[3, 4], (x, y) -> x + y); -- [4, 6]

  SELECT zip_with(ARRAY['a', 'b', 'c'], ARRAY['d', 'e', 'f'], (x, y) -> concat(x, y)); -- [ad, be, cf]

  SELECT zip_with(ARRAY['a'], ARRAY['d', null, 'f'], (x, y) -> coalesce(x, y)); -- [a, null, f]
  ```

1.8.17 Map Functions and Operators

Subscript Operator: []

Description: The **[]** operator is used to retrieve a value corresponding to a given key from a mapping.

```
select age_map['li'] from (values (map(array['li','wang'],array[15,27]))) as table_age(age_map);-- 15
```

Map Functions

- cardinality(*x*)

  Description: Returns the cardinality of map *x*.

  ```
  select cardinality(map(array['num1','num2'],array[11,12]));-- 2
  ```

- element_at(*map(K*, *V)*, *key*)

  Description: Returns the value of *key* in a map. If the map does not contain the *key*, **null** is returned.

  ```
  select element_at(map(array['num1','num2'],array[11,12]),'num1'); --11
  select element_at(map(array['num1','num2'],array[11,12]),'num3');-- NULL
  ```

- map()

  Description: Returns an empty map.

  ```
  select map();-- {}
  ```

- map(*array(K)*, *array(V)) -> map(K*, *V*)

  Description: Returns a map based on the given key-value pair array. The **map_agg()** and **multimap_agg()** functions in the aggregate function can also be used to generate maps.

  ```
  SELECT map(ARRAY[1,3],ARRAY[2,4]);-- {1=2, 3=4}
  ```

- map_from_entries(*array(row(K*, *V))) -> map(K*, *V*)

  Description: Generates a map using a given array.

  ```
  SELECT map_from_entries(ARRAY[(1, 'x'), (2, 'y')]); --  {1=x, 2=y}
  ```

- multimap_from_entries(*array(row(K, V))) -> map(K, array(V)*)

  Description: Returns a composite map based on a given row array. Each key can correspond to multiple values.

  ```
  SELECT multimap_from_entries(ARRAY[(1, 'x'), (2, 'y'), (1, 'z')]); -- {1=[x, z], 2=[y]}
  ```

- map_entries(*map(K, V)) -> array(row(K, V)*)

  Description: Generates a row array using the given map.

  ```
  SELECT map_entries(MAP(ARRAY[1, 2], ARRAY['x', 'y'])); --  [{1, x}, {2, y}]
  ```

- map_concat(*map1(K, V)*, *map2(K, V)*, *...*, *mapN(K, V))*

  Description: Returns the union of all the given maps. If a key is found in multiple given maps, that key's value in the resulting map comes from the last one of those maps. In the following example, key a uses the value 10 of the last map.

  ```
  select map_concat(map(ARRAY['a','b'],ARRAY[1,2]),map(ARRAY['a', 'c'], ARRAY[10, 20]));
         _col0
  -------------------
   {a=10, b=2, c=20}
  (1 row)
  ```

- map_filter(*map(K, V)*, *function(K, V, boolean)) -> map(K, V*)

  Description: Constructs a new map using only the entry that maps the given function to **true**.

  ```
  SELECT map_filter(MAP(ARRAY[], ARRAY[]), (k, v) -> true); -- {}
  SELECT map_filter(MAP(ARRAY[10, 20, 30], ARRAY['a', NULL, 'c']), (k, v) -> v IS NOT NULL); -- {10=a, 30=c}
  SELECT map_filter(MAP(ARRAY['k1', 'k2', 'k3'], ARRAY[20, 3, 15]), (k, v) -> v > 10); -- {k3=15, k1=20}
  ```

- map_keys(*x(K, V)) -> array(K*)

  Description: Returns all arrays constructed by keys in a map.

  ```
  select map_keys(map(array['num1','num2'],array[11,12])); -- [num1, num2]
  ```

- map_values(*x(K, V)) -> array(V*)

  Description: Returns all value-constructed arrays in a map.

  ```
  select map_values(map(array['num1','num2'],array[11,12]));-- [11, 12]
  ```

- map_zip_with(*map(K, V1)*, *map(K, V2)*, *function(K, V1, V2, V3))*

  Description: Merges two given maps into a map by applying the function to a pair of values with the same key. For keys that appear only in one map, **null** is passed as the value for the missing key.

  ```
  SELECT map_zip_with(MAP(ARRAY[1, 2, 3], ARRAY['a', 'b', 'c']), -- {1 -> ad, 2 -> be, 3 -> cf}
                      MAP(ARRAY[1, 2, 3], ARRAY['d', 'e', 'f']),
                      (k, v1, v2) -> concat(v1, v2));
         _col0
  --------------------
   {1=ad, 2=be, 3=cf}
  (1 row)

  SELECT map_zip_with(MAP(ARRAY['k1','k2'],ARRAY[1,2]),Map(ARRAY['K2','k3'],ARRAY[4,9]),(k,v1,v2)->(v1,v2));  -- {k3={NULL, 9}, k1={1, NULL}, k2={2, NULL}, K2={NULL, 4}}
                                    _col0
  -------------------------------------------------------------------------------
  {k3={NULL, 9}, k1={1, NULL}, k2={2, NULL}, K2={NULL, 4}}
  (1 row)

   SELECT map_zip_with(MAP(ARRAY['a', 'b', 'c'], ARRAY[1, 8, 27]), -- {a -> a1, b -> b4, c -> c9}
                      MAP(ARRAY['a', 'b', 'c'], ARRAY[1, 2, 3]),
                      (k, v1, v2) -> k || CAST(v1/v2 AS VARCHAR));
         _col0
  --------------------
   {a=a1, b=b4, c=c9}
  (1 row)
  ```

- transform_keys(*map(K1*, *V)*, *function(K1*, *V*, *K2)) -> map(K2*, *V*)

  Description: For each entry in the map, map the key value K1 to the new key value K2 and keep the corresponding value unchanged.

  ```
  SELECT transform_keys(MAP(ARRAY[], ARRAY[]), (k, v) -> k + 1); -- {}

  SELECT transform_keys(MAP(ARRAY [1, 2, 3], ARRAY ['a', 'b', 'c']), (k, v) -> k + 1); -- {2=a, 3=b, 4=c}

  SELECT transform_keys(MAP(ARRAY ['a', 'b', 'c'], ARRAY [1, 2, 3]), (k, v) -> v * v); -- {1=1, 9=3, 4=2}

  SELECT transform_keys(MAP(ARRAY ['a', 'b'], ARRAY [1, 2]), (k, v) -> k || CAST(v as VARCHAR)); --
  {a1=1, b2=2}

  SELECT transform_keys(MAP(ARRAY [1, 2], ARRAY [1.0, 1.4]), (k, v) -> MAP(ARRAY[1, 2],
  ARRAY['one', 'two'])[k]); -- {two=1.4, one=1.0}
  ```

- size(*x*) → bigint

  Description: Returns the capacity of Map(x) **x**.

  ```
  select size(map(array['num1','num2'],array[11,12])); --2
  ```

- transform_values(*map(K, V1)*, *function(K*, *V2*, *V2)) -> map(K*, *V2*)

  Description: Maps value **V1** to value **V2** for each entry in the map and keeps the corresponding key unchanged.

  ```
  SELECT transform_values(MAP(ARRAY[], ARRAY[]), (k, v) -> v + 1); -- {}

  SELECT transform_values(MAP(ARRAY [1, 2, 3], ARRAY [10, 20, 30]), (k, v) -> v + k); -- {1=11, 2=22,
  3=33}

  SELECT transform_values(MAP(ARRAY [1, 2, 3], ARRAY ['a', 'b', 'c']), (k, v) -> k * k); -- {1=1, 2=4, 3=9}

  SELECT transform_values(MAP(ARRAY ['a', 'b'], ARRAY [1, 2]), (k, v) -> k || CAST(v as VARCHAR)); --
  {a=a1, b=b2}

  SELECT transform_values(MAP(ARRAY [1, 2], ARRAY [1.0, 1.4]),(k, v) -> MAP(ARRAY[1, 2],
  ARRAY['one', 'two'])[k] || '_' || CAST(v AS VARCHAR)); -- {1=one_1.0, 2=two_1.4}
  ```

1.8.18 URL Function

Extraction Function

Description: Extracts content from an HTTP URL (or any URL that complies with the RFC 2396 standard).

```
[protocol:][//host[:port]][path][?query][#fragment]
```

The extracted content does not contain URI syntax separators, such as **:** or **?**.

- url_extract_fragment(*url*) → varchar

  Description: Returns the segment identifier of the URL, that is, the character string following #.

  ```
  select url_extract_fragment('http://www.example.com:80/stu/index.html?
  name=xxx&age=25#teacher');--teacher
  ```

- url_extract_host(url)→ varchar

  Description: Returns the host domain name in *url*.

  ```
   select url_extract_host('http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher');--
  www.example.com
  ```

- url_extract_parameter(*url*, *name*)→ varchar

  Description: Returns the **name** parameter in *url*.

  ```
  select url_extract_parameter('http://www.example.com:80/stu/index.html?
  name=xxx&age=25#teacher','age');-- 25
  ```

- url_extract_path(*url*)→ varchar

  Description: Extracts the path from *url*.

  ```
   select url_extract_path('http://www.example.com:80/stu/index.html?
  name=xxx&age=25#teacher');-- /stu/index.html
  ```

- url_extract_port(*url*)→ bigint

  Description: Extracts the port number from *url*.

  ```
  select url_extract_port('http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher');-- 80
  ```

- url_extract_protocol(*url*)→ varchar

  Description: Extracts the protocol from *url*.

  ```
  select url_extract_protocol('http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher');
  --  http
  ```

- url_extract_query(*url*)→ varchar

  Description: Extracts the query character string from *url*.

  ```
  select url_extract_query('http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher'); --
  name=xxx&age=25
  ```

Encoding Function

- url_encode(*value*) → varchar

  Description: Escapes *value* so that it can be securely contained in the URL query parameter name and value.

  – Letter characters are not encoded.

  – Characters **.**, **-**, *****, and _ are not encoded.

  – ASCII space characters are encoded as **+**.

  – All other characters are converted to UTF-8, and the byte is encoded as a string **%***XX*, where *XX* is an uppercase hexadecimal value of UTF-8 bytes.

  ```
  select url_encode('http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher');
  -- http%3A%2F%2Fwww.example.com%3A80%2Fstu%2Findex.html%3Fname%3Dxxx%26age
  %3D25%23teacher
  ```

- url_decode(*value*) → varchar

  Description: Decodes the URL after value encoding.

  ```
  select url_decode('http%3A%2F%2Fwww.example.com%3A80%2Fstu%2Findex.html%3Fname%3Dxxx
  %26age%3D25%23teacher');
  -- http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher
  ```

1.8.19 UUID Function

This function is used to generate a pseudo-random unique universal identifier.

```
select uuid();
```

1.8.20 Color Function

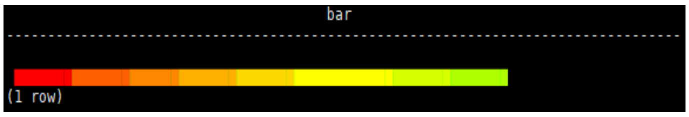
- bar(*x*, *width*)

  Description: Renders a single bar in the ANSI bar chart using the default low-frequency red and high-frequency green. For example, if you pass 25% of *x* and 40 widths to this function. A 10-character red bar followed by 30 spaces will be drawn to create a 40-character bar.

- bar(x, width, low_color, high_color)

  Description: Draws a straight line with the specified width in an ANSI bar chart. The x parameter is a double-precision value ranging from 0 to 1. If the value of *x* is out of the range [0,1], the value is truncated to 0 or 1. **low_color** and **high_color** capture the color at either end of the horizontal bar chart. For example, if *x* is **0.5**, *width* is **80**, **low_color** is **0xFF0000**, and **high_color** is **0x00FF00**, this function returns a 40-character bar. The bar consists of red (0xFF0000) and yellow (0xFFFF00), the remaining 80 characters are padded with spaces.

```
select bar(0.75,80,rgb(255,0,0),rgb(0,255,0));
```



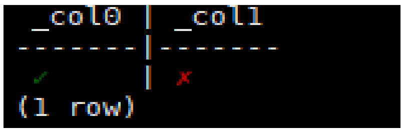
```
                                      bar
--------------------------------------------------------------------------------

(1 row)
```

- render(b)

  Description: Returns right and wrong symbols based on Boolean values.

```
select render(true),render(false);
```



```
 _col0 | _col1
-------|-------
 ✓     | ✗
(1 row)
```

1.8.21 Teradata Function

The following functions provide the Teradata SQL capability.

String Functions

- char2hexint(*string*)

  Description: Returns the hexadecimal representation of the UTF-16BE encoding of a string.

- index(*string*, *substring*)

  Description: Same as strpos().

Date Functions

The functions in this section use format strings that are compatible with the Teradata datetime function. The following table describes the supported format specifiers based on the Teradata reference manual:

Specifier	Description
- / , . ; :	Ignore punctuation
dd	Day (1 to 31) in a month
hh	Hour (1 to 12) in a day
hh24	Hour (0 to 23) in a day
mi	Minute (0 to 59)
mm	Month (01 to 12)
ss	Second (0 to 59)

Specifier	Description
yyyy	Four-digit year
yy	Two-digit year

NOTE

Case-insensitive is not supported. All specifiers must be in lower case.

- to_char(*timestamp*, *format*)

  Description: Outputs a timestamp as a string in a specified format.

  ```
  select to_char(timestamp '2020-12-18 15:20:05','yyyy/mmdd hh24:mi:ss');-- 2020/1218 15:20:05
  ```

- to_timestamp(*string*, *format*)

  Description: Parses a string to a timestamp in a specified format.

  ```
  select to_timestamp('2020-12-18 15:20:05','yyyy-mm-dd hh24:mi:ss'); -- 2020-12-18 15:20:05.000
  ```

- to_date(*string*, *format*)

  Description: Converts a string to a date in the specified format.

  ```
  select to_date('2020/12/04','yyyy/mm/dd'); -- 2020-12-04
  ```

1.8.22 Data Masking Functions

Data masking refers to the process of distorting sensitive information based on masking rules to protect sensitive privacy data.

- mask_first_n(string str[, int n]) →varchar

  Description: Returns the masked version of a string. The first *n* values are masked. Uppercase letters are converted to **X**, lowercase letters to **x**, and digits to **n**.

  ```
  select mask_first_n('Aa12-5678-8765-4321', 4);
          _col0
  ---------------------
   Xxnn-5678-8765-4321
  (1 row)
  ```

- mask_last_n(string str[, int n]) →varchar

  Description: Returns the masked version of a string. The last *n* values are masked. Uppercase letters are converted to **X**, lowercase letters to **x**, and digits to **n**.

  ```
  select mask_last_n('1234-5678-8765-Hh21', 4);
          _col0
  ---------------------
   1234-5678-8765-Xxnn
  (1 row)
  ```

- mask_show_first_n(string str[, int n]) →varchar

  Description: Returns the masked version of a string. Only the first *n* characters are displayed. Uppercase letters are converted to **X**, lowercase letters to **x**, and digits to **n**.

  ```
  select mask_show_first_n('1234-5678-8765-4321',4);
          _col0
  ---------------------
   1234-nnnn-nnnn-nnnn
  (1 row)
  ```

- mask_show_flairst_n(string str[, int n]) →varchar

  Description: Returns the masked version of a string. Only the last *n* characters are displayed. Uppercase letters are converted to **X**, lowercase letters to **x**, and digits to **n**.

  ```
  select mask_show_last_n('1234-5678-8765-4321',4);
          _col0
  ---------------------
   nnnn-nnnn-nnnn-4321
  (1 row))
  ```

- mask_hash(string|char|varchar str) →varchar

  Description: Returns a hash value of a string. Hash values are consistent and can be used to join masked values across tables. For a non-string, **NULL** is returned.

  ```
  select mask_hash('panda');
                                _col0
  ------------------------------------------------------------------
   a7cdf5d0586b392473dd0cd08c9ba833240006a8a7310bf9bc8bf1aefdfaeadb
  (1 row)
  ```

1.8.23 IP Address Functions

contains(network, address) → boolean

If the CIDR network contains the **address** value, **true** is returned.

```
SELECT contains('10.0.0.0/8', IPADDRESS '10.255.255.255'); -- true
SELECT contains('10.0.0.0/8', IPADDRESS '11.255.255.255'); -- false
SELECT contains('2001:0db8:0:0:0:ff00:0042:8329/128', IPADDRESS '2001:0db8:0:0:0:ff00:0042:8329'); -- true
SELECT contains('2001:0db8:0:0:0:ff00:0042:8329/128', IPADDRESS '2001:0db8:0:0:0:ff00:0042:8328'); -- false
```

1.8.24 Quantile Digest Functions

Overview

Quantile digest is a data sketch that stores approximate percentile information. The HetuEngine type for this data structure is called qdigest.

Function

- merge(qdigest) → qdigest

  Description: Merges all input qdigest data into one qdigest.

- value_at_quantile(qdigest(T), quantile) → T

  Description: Returns the approximate percentile values from the quantile digest given the number **quantile** between 0 and 1.

- values_at_quantiles(qdigest(T), quantiles) -> array(T)

  Description: Returns the approximate percentile values as an array given the input quantile digest and array of values between 0 and 1 which represent the quantiles to return.

- qdigest_agg(x) -> qdigest([same as x])

  Description: Returns the **qdigest** which is composed of all input values of x.

- qdigest_agg(x, w) -> qdigest([same as x])

  Description: Returns the **qdigest** which is composed of all input values of **x** using the per-item weight **w**.

- qdigest_agg(x, w, accuracy) -> qdigest([same as x])

  Description: Returns the **qdigest** which is composed of all input values of **x** using the per-item weight **w** and maximum error of accuracy. **accuracy** must be a value greater than zero and less than one, and it must be constant for all input rows.

1.8.25 T-Digest Functions

Overview

A T-digest is a data sketch that stores approximate percentile information. The HetuEngine type for this data structure is called tdigest. Tdigests may be merged without losing precision, and for storage and retrieval they may be cast to/from VARBINARY.

Function

- merge(tdigest)→tdigest

  Description: Merges all input tdigests into a single tdigest.
- value_at_quantile(tdigest,quantile)→double

  Description: Returns the approximate percentile values from the T-digest given the number **quantile** between 0 and 1.
- values_at_quantiles(tdigest,quantiles)->array(double)

  Description: Returns the approximate percentile values as an array given the input T-digest and array of values between 0 and 1 which represent the quantiles to return.
- tdigest_agg(x)->tdigest

  Description: Returns the **tdigest** which is composed of all input values of x. **x** can be of any numeric type.
- tdigest_agg(x,w)->tdigest

  Description: Returns the **tdigest** which is composed of all input values of **x** using the per-item weight **w**. **w** must be no less than 1. **x** and **w** can be of any numeric type.

2 Implicit Data Type Conversion

2.1 Introduction

Implicit data type conversion enables HetuEngine to automatically convert the data type when the queried data type does not match the data type of the table when users access the HetuEngine resources through the client. This avoids inconvenience caused by strong data type verification. Currently, the data type can be implicitly converted for INSERT statement, WHERE conditions, operations (+, -, *, and /), and function calls (connection operations ||).

2.2 Implicit Conversion Table

Data types will be implicitly converted when they do not match, but not all data types support implicit conversion. The following table lists the data type conversion tables supported by the implicit conversion function.

Table 2-1 Implicit conversion table

-	BOOLEAN	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	DOUBLE	DECIMAL	VARCHAR
BOOLEAN	\	Y(1)	Y	Y	Y	Y	Y	Y	Y(2)
TINYINT	Y(3)	\	Y	Y	Y	Y	Y	Y	Y
SMALLINT	Y	Y(4)	\	Y	Y	Y	Y	Y	Y
INTEGER	Y	Y	Y	\	Y	Y	Y	Y	Y

-	BOOLEAN	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	DOUBLE	DECIMAL	VARCHAR
BIGINT	Y	Y	Y	Y	\	Y	Y	Y	Y
REAL	Y	Y	Y	Y	Y	\	Y	Y(5)	Y
DOUBLE	Y	Y	Y	Y	Y	Y	\	Y	Y
DECIMAL	Y	Y	Y	Y	Y	Y	Y	\(6)	Y
VARCHAR	Y(7)	Y	Y	Y	Y	Y	Y	Y(8)	\
CHAR	N	N	N	N	N	N	N	N	Y
VARBINARY	N	N	N	N	N	N	N	N	N
JSON	N	N	N	N	N	N	N	N	Y
DATE	N	N	N	N	N	N	N	N	Y
TIME	N	N	N	N	N	N	N	N	Y
TIME WITH TIME ZONE	N	N	N	N	N	N	N	N	Y
TIMESTAMP	N	N	N	N	N	N	N	N	Y
TIMESTAMP WITH TIME ZONE	N	N	N	N	N	N	N	N	Y

Table 2-2 Implicit conversion table (continued)

-	CHAR	VARBINARY	JSON	DATE	TIME	TIME WITH TIME ZONE	TIMESTAMP	TIMESTAMP WITH TIME ZONE
BOOLEAN	N	N	Y	N	N	N	N	N
TINYINT	N	N	Y	N	N	N	N	N

-	CHAR	VARBINARY	JSON	DATE	TIME	TIME WITH TIME ZONE	TIMESTAMP	TIMESTAMP WITH TIME ZONE
SMALLINT	N	N	Y	N	N	N	N	N
INTEGER	N	N	Y	N	N	N	N	N
BIGINT	N	N	Y	N	N	N	N	N
REAL	N	N	Y	N	N	N	N	N
DOUBLE	N	N	Y	N	N	N	N	N
DECIMAL	N	N	Y	N	N	N	N	N
VARCHAR	Y(9)	Y	Y	Y(10)	Y(11)	Y(12)	Y(13)	Y
CHAR	\	N	N	N	N	N	N	N
VARBINARY	N	\	N	N	N	N	N	N
JSON	N	N	\	N	N	N	N	N
DATE	N	N	Y	\	N	N	Y(14)	Y
TIME	N	N	N	N	\	Y(15)	Y(16)	Y
TIME WITH TIME ZONE	N	N	N	N	Y	\	Y	Y
TIMESTAMP	N	N	N	Y	Y	Y	\	Y
TIMESTAMP WITH TIME ZONE	N	N	N	Y	Y	Y	Y	\

NOTE

- The result of BOOLEAN->NUMBER can only be **0** or **1**.
- The result of BOOLEAN->VARCHAR can only be **TRUE** or **FALSE**.
- For NUMBER -> BOOLEAN, **0** is **false**, and other values are **true**.
- The value of BIG PRECISION -> SMALL cannot be greater than the value range of the target type. Otherwise, an error is reported.
- The integer part of REAL/FLOAT ->DECIMAL must be greater than or equal to the integer part of REAL/FLOAT. Otherwise, an error is reported during conversion. If the decimal part is insufficient, the data is truncated.
- The integer part of the DECIMAL->DECIMAL target type must be greater than or equal to that of the source type. Otherwise, the conversion fails and the decimal part will be truncated if it is insufficient.
- For VARCHAR->BOOLEAN, only **0**, **1**, **TRUE**, and **FALSE** can be converted.
- For VARCHAR->DECIMAL, if the number of decimal places is greater than the number of decimal places of the target decimal, truncation occurs. If the number of integer places is greater than the number of decimal places of the target decimal, an error is reported.
- For VARCHAR->CHAR, if the length of VARCHAR exceeds the target length, truncation occurs.
- For VARCHAR->DATE, dates can only be separated by hyphens (-), for example, 2000-01-01.
- For VARCHAR->TIME, only the strict date format HH:MM:SS.XXX is supported.
- For VARCHAR->TIME ZONE, only the strict time format is supported, for example, 01:02:03.456 America/Los_Angeles.
- For VARCHAR->TIMESTAMP, only the strict format YYYY-MM-DD HH:MM:SS.XXX is supported.
- DATE->TIMESTAMP automatically supplements the time by adding 0s to the end., for example, 2010-01-01 to 2010-01-01 00:00:00.000.
- TIME->TIME WITH TIME ZONE automatically supplements the time zone.
- TIME->TIMESTAMP automatically supplements the date. The default value **1970-01-01** is used.

3 Appendix

3.1 Data preparation for the sample table in this document

```
-- Create a table containing TINYINT data:
 CREATE TABLE int_type_t1  (IT_COL1 TINYINT) ;
--Insert data of the TINYINT type.
 insert into int_type_t1  values (TINYINT'10');
-- Create a table containing DECIMAL data:
CREATE TABLE decimal_t1       (dec_col1 DECIMAL(10,3)) ;
– Insert data of the DECIMAL type:
insert into decimal_t1  values (DECIMAL '5.325' );
create table array_tb(col1 array<int>,col2 array<array<int>>);
create table row_tb(col1 row(a int,b varchar));

-- Create a Map table.
create table map_tb(col1 MAP<STRING,INT>);
-- Insert a piece of data of the Map type.
insert into map_tb values(MAP(ARRAY['foo','bar'],ARRAY[1,2]));
--Query data.
select * from map_tb; --   {bar=2, foo=1}

-- Create a ROW table.
create table row_tb (id int,col1 row(a int,b varchar));
-- Insert data of the ROW type.
insert into row_tb values (1,ROW(1,'SSS'));
--Query data.
select * from row_tb; --
 id |     col1
----|--------------
  1 | {a=1, b=SSS}
select col1.b from row_tb; -- SSS
select col1[1] from row_tb; -- 1

-- Creating a STRUCT table.
create table struct_tab (id int,col1 struct<col2: integer, col3: string>);
-- Insert data of the STRUCT type.
 insert into struct_tab VALUES(1, struct<2, 'test'>);
--Query data.
select * from struct_tab; --
 id |        col1
----|---------------------
```

```
  1 | {col2=2, col3=test}


-- Creating a schema named "web":
CREATE SCHEMA web;
--Create a schema named sales in the Hive data source:
CREATE SCHEMA hive.sales;
-- Creating a schema named traffic, if it does not exist:
CREATE SCHEMA IF NOT EXISTS traffic;

-- Create a new table orders and use the WITH clause to specify the storage format, storage location, and whether the table is a foreign table.
CREATE TABLE orders (
orderkey bigint,
orderstatus varchar,
totalprice double,
orderdate date
)
WITH (format = 'ORC', location='/user',external=true);
-- If the orders table does not exist, create the orders table and add table comments and column comments:
CREATE TABLE IF NOT EXISTS new_orders (
orderkey bigint,
orderstatus varchar,
totalprice double COMMENT 'Price in cents.',
orderdate date
)
COMMENT 'A table to keep track of orders.';
-- Create the bigger_orders table using the column definition of the orders table:
CREATE TABLE bigger_orders (
another_orderkey bigint,
LIKE orders,
another_orderdate date
);

CREATE SCHEMA hive.web WITH (location = 'obs://bucket/user');
--Create a partitioned table.
CREATE TABLE hive.web.page_views (
  view_time timestamp,
  user_id bigint,
  page_url varchar,
  ds date,
  country varchar
)
WITH (
  format = 'ORC',
  partitioned_by = ARRAY['ds', 'country'],
  bucketed_by = ARRAY['user_id'],
  bucket_count = 50
);
--Insert data.
insert into hive.web.page_views values(timestamp '2020-07-17 23:00:15',bigint '15141','www.local.com',date '2020-07-17','US' );
insert into hive.web.page_views values(timestamp '2020-07-17 23:00:16',bigint '15142','www.abc.com',date '2020-07-17','US' );
insert into hive.web.page_views values(timestamp '2020-07-18 23:00:18',bigint '18148','www.local.com',date '2020-07-18','US' );

-- Delete all data in the partition specified by the WHERE clause from the partitioned table.
delete from hive.web.page_views where ds=date '2020-07-17' and country='US';

--Run the following statement to create the orders_column_aliased table based on the query result of a specified column:
CREATE TABLE orders_column_aliased (order_date, total_price)
AS
SELECT orderdate, totalprice FROM orders;
--Create the orders_by_data table based on the summary result of the orders table.
CREATE TABLE orders_by_date
COMMENT 'Summary of orders by date'
```

```
WITH (format = 'ORC')
AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate;
--If the orders_by_date table does not exist, create the orders_by_date table:
CREATE TABLE IF NOT EXISTS orders_by_date AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate;
--Create the empty_orders table using the schema that is the same as that of the orders table but does
not contain data.
CREATE TABLE empty_orders AS
SELECT *
FROM orders
WITH NO DATA;

--Create a view named test_view in the orders table:
CREATE VIEW test_view (oderkey comment 'orderId',orderstatus comment 'status',half comment 'half') AS
SELECT orderkey, orderstatus, totalprice / 2 AS half FROM orders;
--Create the orders_by_date_view view based on the summary result of the orders table.
CREATE VIEW orders_by_date_view AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate;
--Create a new view to replace the existing view:
CREATE OR REPLACE VIEW test_view AS
SELECT orderkey, orderstatus, totalprice / 4 AS quarter
FROM orders;

--Modify the definition of an existing table.
--Data preparation.
create table users (id int,name varchar);
--Change the table name from users to people:
ALTER TABLE users RENAME TO people;
--Add the zip column to the people table:
ALTER TABLE people ADD COLUMN zip varchar;
--Delete the zip column from the people table:
ALTER TABLE people DROP COLUMN zip;
--change the column name id in the people table to user_id:
ALTER TABLE people RENAME COLUMN id TO user_id;

create table testfordrop(name varchar);

--CreatE a view.
create view orders_by_date as select * from orders;
--Set the comment information of a table. You can delete the comment by setting the comment
information to NULL.
COMMENT ON TABLE people IS 'master table';

--create a table with the column id and name:
CREATE TABLE example AS
SELECT * FROM (
VALUES
(1, 'a'),
(2, 'b'),
(3, 'c')
) AS t (id, name);

--Create the fruit and fruit_copy tables:
create table fruit (name varchar,price double);
create table fruit_copy (name varchar,price double);
--Insert a row of data into the fruit table:
insert into fruit values('LIchee',32);
--Insert multiple lines of data into the fruit table:
insert into fruit values('banana',10),('peach',6),('lemon',12),('apple',7);
--Load the data lines in the fruit table to the fruit_copy table. After the execution, there are five records in
the table.
insert into fruit_copy select * from fruit;
```

```
--Clear the fruit_copy table, and then load the data in the fruit table to the table. After the execution,
there are two records in the fruit_copy table.
insert overwrite fruit_copy select *  from fruit limit 2;

--Create a shipping table:
create table shipping(origin_state varchar(25),origin_zip integer,destination_state
varchar(25) ,destination_zip integer,package_weight integer);

--Insert data.
insert into shipping values ('California',94131,'New Jersey',8648,13),
('California',94131,'New Jersey',8540,42),
('California',90210,'Connecticut',6927,1337),
('California',94131,'Colorado',80302,5),
('New York',10002,'New Jersey',8540,3),
('New Jersey',7081,'Connecticut',6708,225);

--Create a table and insert data into the table.
create table cookies_log (cookieid varchar,createtime  date,pv int);
insert into cookies_log values
    ('cookie1',date '2020-07-10',1),
    ('cookie1',date '2020-07-11',5),
    ('cookie1',date '2020-07-12',7),
    ('cookie1',date '2020-07-13',3),
    ('cookie1',date '2020-07-14',2),
    ('cookie1',date '2020-07-15',4),
    ('cookie1',date '2020-07-16',4),
    ('cookie2',date '2020-07-10',2),
    ('cookie2',date '2020-07-11',3),
    ('cookie2',date '2020-07-12',5),
     ('cookie2',date '2020-07-13',6),
      ('cookie2',date '2020-07-14',3),
      ('cookie2',date '2020-07-15',9),
    ('cookie2',date '2020-07-16',7);

--Create a table.
create  table new_shipping (origin_state varchar,origin_zip varchar,packages int ,total_cost int);

--Insert data.
insert into new_shipping
values
('California','94131',25,100),
('California','P332a',5,72),
('California','94025',0,155),
('New Jersey','08544',225,490);

--Create a data table and insert data.
create table salary (dept varchar, userid varchar, sal double);
 insert into salary values ('d1','user1',1000),('d1','user2',2000),('d1','user3',3000),('d2','user4',4000),
('d2','user5',5000);

-- Prepare data.
create table cookie_views( cookieid varchar,createtime timestamp,url varchar);
insert into cookie_views values
('cookie1',timestamp '2020-07-10 10:00:02','url20'),
('cookie1',timestamp '2020-07-10 10:00:00','url10'),
('cookie1',timestamp '2020-07-10 10:03:04','urll3'),
('cookie1',timestamp '2020-07-10 10:50:05','url60'),
('cookie1',timestamp '2020-07-10 11:00:00','url70'),
('cookie1',timestamp '2020-07-10 10:10:00','url40'),
('cookie1',timestamp '2020-07-10 10:50:01','url50'),
('cookie2',timestamp '2020-07-10 10:00:02','url23'),
('cookie2',timestamp '2020-07-10 10:00:00','url11'),
('cookie2',timestamp '2020-07-10 10:03:04','url33'),
('cookie2',timestamp '2020-07-10 10:50:05','url66'),
('cookie2',timestamp '2020-07-10 11:00:00','url77'),
('cookie2',timestamp '2020-07-10 10:10:00','url47'),
('cookie2',timestamp '2020-07-10 10:50:01','url55');

CREATE TABLE visit_summaries (  visit_date date,  hll varbinary);
```

```
insert into visit_summaries select createtime,cast(approx_set(cookieid) as varbinary) from cookies_log
group by createtime;

CREATE TABLE nation (name varchar,  regionkey integer);
insert into nation values ('ETHIOPIA',0),
('MOROCCO',0),
('ETHIOPIA',0),
('KENYA',0),
('ALGERIA',0),
('MOZAMBIQUE',0);

CREATE TABLE region (  name varchar,  regionkey integer);
insert into region values ('ETHIOPIA',0),
('MOROCCO',0),
('ETHIOPIA',0),
('KENYA',0),
('ALGERIA',0),
('MOZAMBIQUE',0);
```

3.2 Syntax Compatibility of Common Data Sources

Syntax	Hive	Hudi
show schemas of databases	Y	Y
create schema of databases	Y	Y
alter schema of databases	Y	N
drop schema of databases	Y	Y
show tables/show create table/ show functions/show session of tables	Y	Y
create of tables	Y	N
create table TABLENAME as of tables	Y	N
insert into TABLENAME values of tables	Y	N
insert into TABLENAME select of tables	Y	N
insert overwrite TABLENAME values of tables	Y	N
insert overwrite TABLENAME select of tables	Y	N
alter of tables	Y	N
select of tables	Y	Y
delete of tables	Y	N
drop of tables	Y	N

Syntax	Hive	Hudi
desc/describe TABLENAME of tables	Y	Y
comment of tables	Y	N
explain of tables	Y	Y
show columns of tables	Y	Y
select column of tables	Y	Y
create view of views	Y	N
create or replace view of views	Y	N
alter of views	Y	N
drop of views	Y	N
select of views	Y	Y
desc/describe VIEWNAME of views	Y	Y
show views/show create view of views	Y	Y
show columns of views	Y	Y
select column of views	Y	Y