数据湖探索

HetuEngine SQL 语法参考

文档版本 01

发布日期 2025-05-09

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

安全声明

漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：
https://www.huawei.com/cn/psirt/vul-response-process
如企业客户须获取漏洞信息，请参见如下网址：
https://securitybulletin.huawei.com/enterprise/cn/security-advisory

目 录

1 HetuEngine SQL 语法

1.1 使用前必读

使用须知

- DLI HetuEngine功能为白名单功能，如需使用，请在管理控制台右上角，选择“工单 > 新建工单”，提交申请。
- 使用前您需要先创建一个HetuEngine类型的SQL队列，具体操作请参考**创建弹性资源池并添加队列**。
- HetuEngine SQL需搭配lakeformation使用。详细内容请参考**DLI对接LakeFormation**。

HetuEngine 简介

HetuEngine是华为推出的高性能交互式SQL分析及数据虚拟化引擎，能够与大数据生态无缝融合，实现海量数据的秒级交互式查询。

DLI+HetuEngine能够快速处理大规模数据集的查询请求，迅速和高效从大数据中提取信息，极大地简化了数据的管理和分析流程，提升大数据环境下的索引和查询性能。

- TB级数据秒级响应：

  HetuEngine通过自动优化资源与负载的配比，能够对TB级数据实现秒级响应，极大提升了数据查询的效率。
- Serverless资源开箱即用：

  Serverless服务模式无需关注底层配置、软件更新和故障问题，资源易维护，易扩展。
- 多种资源类型满足不同场景业务需求：

  共享资源池：按量计费，提供更具性价比的计算资源。

  独享资源池：提供独享资源池，满足高性能资源需求。

1.2 数据类型

1.2.1 数据类型介绍

目前使用Hetu引擎建表时支持的数据类型有：tinyint，smallint，bigint，int，boolean，real，decimal，double，varchar，string，binary，varbinary，timestamp，date，char，array，row，map，struct。其余的类型在数据查询和运算时支持。

通常情况下，大部分非复合数据类型都可以通过字面量加字符串的方式来输入，示例为添加了一个json格式的字符串：

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

1.2.2 布尔类型

“真”值的有效文本值是：TRUE、't'、'true'、'1'。

“假”值的有效文本值是：FALSE、'f'、'false'、'0'。

使用TRUE和FALSE是比较规范的用法（也是SQL兼容的用法）。

示例：

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

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

select BOOLEAN 't';
 _col0
-------
 true
(1 row)
```

1.2.3 整数类型

表 1-1 整数类型

名称	描述	存储空间	取值范围	字面量
TINYINT	微整数	8位	-128~127	TINYINT
SMALLINT	小整数	16位	-32,768 ~ +32,767	SMALLINT
INTEGER	整数	32位	-2,147,483,648 ~ +2,147,483,647	INT

名称	描述	存储空间	取值范围	字面量
BIGINT	大整数	64位	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807	BIGINT

示例：

```
--创建具有TINYINT类型数据的表。
CREATE TABLE int_type_t1  (IT_COL1 TINYINT) ;
--插入TINYINT类型数据
insert into int_type_t1  values (TINYINT'10');
--查看数据。
SELECT * FROM int_type_t1;
 it_col1
---------
 10
(1 row)
--删除表。
DROP TABLE int_type_t1;
```

1.2.4 固定精度型

名称	描述	存储空间	取值范围	字面量
DECIMAL	固定精度的十进制数。精度最高支持到38位，但精度小于18位能保障性能最优。 Decimal有两个输入参数： ● precision：总位数，默认38 ● scale：小数部分的位数，默认0 **说明** 如果小数位为零，即十进制（38,0），则支持最高19位精度。	64位	$-10^{38}+1 \sim 10^{38}-1$	DECIMAL
NUMERIC	同DECIMAL	128位	$-10^{38}+1 \sim 10^{38}-1$	NUMERIC

表 1-2 字面量示例

字面量示例	数据类型
DECIMAL '0'	DECIMAL(1)
DECIMAL '12345'	DECIMAL(5)
DECIMAL '0000012345.1234500000'	DECIMAL(20, 10)


```
--创建具有DECIMAL类型数据的表
CREATE TABLE decimal_t1 (dec_col1 DECIMAL(10,3)) ;

--插入具有DECIMAL类型数据
insert into decimal_t1 values (DECIMAL '5.325');

--查看数据
SELECT * FROM decimal_t1;
 dec_col1
---------
 5.325
(1 row)

--删除表
DROP TABLE decimal_t1;

--创建NUMERIC 类型表
CREATE TABLE tb_numberic_hetu(col1 NUMERIC(9,7));

--插入数据
INSERT INTO tb_numberic_hetu values(9.12);


--查看数据
SELECT * FROM tb_numberic_hetu;
col1
------------
  9.1200000
(1 row)
```

1.2.5 浮点型

名称	描述	存储空间	取值范围	字面量
REAL	实数	32位	1.40129846432481707e-45 ~3.40282346638528860e+38，正或负	REAL
DOUBLE	双精度浮点数，15到17个有效位，具体取决于使用场景，有效位位数并不取决于小数点位置	64位	4.94065645841246544e-324 ~1.79769313486231570e+308，正或负	DOUBLE
FLOAT	单精度浮点数，6到9个有效位，具体取决于使用场景，有效位位数并不取决于小数点位置	32位	1.40129846432481707e-45 ~3.40282346638528860e+38，正或负	FLOAT

用法说明：

- 分布式查询使用高性能硬件指令进行单精度或者双精度运算时，由于每次执行的顺序不一样，在调用聚合函数，比如SUM()，AVG()，特别是当数据规模非常大时，达到数千万甚至数十亿，其运算结果可能会略有不同。这种情况下，建议使用DECIMAL数据类型来运算。

- 可以使用别名来指定数据类型。

  示例：

  ```
  --创建具有float类型数据的表
  CREATE TABLE float_t1 (float_col1 FLOAT) ;
  --插入具有float类型数据
  insert into float_t1 values (float '3.50282346638528862e+38');
  --查看数据
  SELECT * FROM float_t1;
  float_col1
  ------------
  Infinity
  (1 row)
  --删除表
  DROP TABLE float_t1;
  ```

- 当小数部分为0时，可以通过cast()转为对应范围的整数处理，小数部分会四舍五入。

  示例：

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

- 使用指数表达式时，可以将字符串转为对应类型。

  示例：

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

1.2.6 字符类型

名称	描述
VARCHAR(n)	变长字符串，n指字节长度。
CHAR(n)	定长字符串，不足补空格。n是指字节长度，如不带精度n，默认为1。
VARBINARY	变长二进制数据。需要带上前缀X，如：X'65683F'，暂不支持指定长度的二进制字符串。
JSON	取值可以是a JSON object、a JSON array、a JSON number、a JSON string、true、false or null。
STRING	兼容impala的String，底层是varchar。
BINARY	兼容hive的Binary，底层实现为varbinary。

- SQL表达式中，支持简单的字符表达式，也支持Unicode方式，一个Unicode字符串是以U&为固定前缀，以4位数值表示的Unicode前需要加转义符。

  ```
  -- 字符表达式
  select 'hello,winter!';
  ```



```
     _col0
------------------
 hello,winter!
(1 row)
-- Unicode 表达式
select U&'Hello winter \2603 !';
     _col0
------------------
 Hello winter !
(1 row)
-- 自定义转义符
select U&'Hello winter #2603 !' UESCAPE '#';
     _col0
------------------
 Hello winter !
(1 row)
```

- VARBINARY与BINARY。

```
-- 创建VARBINARY类型或BINARY类型的表
 create table binary_tb(col1 BINARY);

-- 插入数据
 INSERT INTO binary_tb values (X'63683F');

--查询数据
 select * from binary_tb ;  -- 63 68 3f
```

- 在做CHAR 数值比较的时候，在对两个仅尾部空格数不同的CHAR进行比较时，会认为它们是相等的。

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

1.2.7 时间和日期类型

限制

时间和日期类型目前精确到毫秒。

表 1-3 时间和日期类型

名称	描述	存储空间
DATE	日期和时间。仅支持ISO 8601格式：'2020-01-01'	32位
TIME	不带时区的时间（时、分、秒、毫秒） 例如：TIME '01:02:03.456'	64位
TIME WITH TIMEZONE	带时区的时间（时、分、秒、毫秒），时区用UTC值表示 例如：TIME '01:02:03.456 -08:00'	96位
TIMESTAMP	时间戳	64位
TIMESTAMP WITH TIMEZONE	带时区的时间戳	64位

名称	描述	存储空间
INTERVAL YEAR TO MONTH	时间间隔字面量，年，月，格式： SY-M S：可选符号（ +/- ） Y：年数 M：月数	128位
INTERVAL DAY TO SECOND	时间间隔字面量，日，小时，分钟，秒，精确到毫秒，格式：SD H:M:S.nnn S：可选符号（ +/- ） D：天数 M：分钟数 S：秒数 nnn：毫秒数	128位

示例：

```
-- 查询日期
SELECT DATE '2020-07-08';
   _col0
------------
 2020-07-08
(1 row)

-- 查询时间
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)

-- 时间间隔用法
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 复杂类型

ARRAY

数组。

示例：ARRAY[1, 2, 3]。

```
--创建ARRAY类型表
create table array_tb(col1 ARRAY<STRING>);

--插入一条ARRAY类型数据
insert into array_tb values(ARRAY['HetuEngine','Hive','Mppdb']);

--查询数据
select * from array_tb; --     [HetuEngine, Hive, Mppdb]
```

MAP

键值对数据类型。

示例：MAP(ARRAY['foo', 'bar']、ARRAY[1, 2])。

```
--创建Map类型表
create table map_tb(col1 MAP<STRING,INT>);

--插入一条Map类型数据
insert into map_tb values(MAP(ARRAY['foo','bar'],ARRAY[1,2]));

--查询数据
select * from map_tb; --    {bar=2, foo=1}
```

ROW

ROW的字段可是任意所支持的数据类型，也支持各字段数据类型不同的混合方式。

说明

当建表使用ROW类型或STRUCT类型的字段时，如果表仅包含单列，则无法插入数据。

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

--插入ROW类型数据
insert into row_tb values (1,ROW(1,'HetuEngine'));

--查询数据
select * from row_tb;
 id |     col1
----|--------------z
  1 | {a=1, b=HetuEngine}

--字段是支持命名的，默认情况下，Row的字段是未命名的
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)

--命名后的字段，可以通过域操作符"."访问
select col1.b from row_tb; -- HetuEngine
```



```
--命名和未命名的字段，都可以通过位置索引来访问，位置索引从1开始，且必须是一个常量
select col1[1] from row_tb;   -- 1
```

STRUCT

底层用ROW实现，参照**ROW**。

说明

当建表使用ROW类型或STRUCT类型的字段时，如果表仅包含单列，则无法插入数据。

示例：

```
-- 创建struct 表
create table struct_tab (id int,col1 struct<col2: integer, col3: string>);

--插入 struct 类型数据
 insert into struct_tab VALUES(1, struct<2, 'HetuEngine'>);

--查询数据
select * from struct_tab;
 id |        col1
----|---------------------
  1 | {col2=2, col3=HetuEngine}
```

IPADDRESS

IP地址，可以表征IPv4或者IPv6地址。但在系统内，该类型是一个统一的IPv6地址。

对于IPv4的支持，是通过将IPv4映射到IPv6的取值范围（RFC 4291#section-2.5.5.2）来实现的。当创建一个IPv4时，会被映射到IPv6。当格式化时，如果数据是IPv4又会被重新映射为IPv4。 其他的地址则会按照RFC 5952所定义的规范格式来进行格式化。

示例：

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

UUID

标准UUID (Universally Unique IDentifier)，也被称为GUID (Globally Unique IDentifier)。

遵从RFC 4122标准所定义的格式。

示例：

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

QDIGEST

分位数（Quantile），亦称分位点，是指将一个随机变量的概率分布范围分为几个等份的数值点，常用的有中位数（即二分位数）、四分位数、百分位数等。quantile digest是一个分位数的集合，当需要查询的数据落在某个分位数附近时，就可以用这个

分位数作为要查询数据的近似值。它的精度可以调节，但更高精度的结果会带来空间的昂贵开销。

1.3 DDL 语法

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,...)]

描述

创建一个空的schema。schema是表、视图以及其他数据库对象的容器。当指定可选参数IF NOT EXISTS时，如果系统已经存在同名的schema，将不会报错。

示例

- 创建一个名为web的schema：
```
CREATE SCHEMA web;
```
- 在指定路径创建schema，路径必须是obs的并行桶，路径末尾不能加/，指定路径兼容写法示例：
```
CREATE SCHEMA test_schema_5 LOCATION 'obs://${bucket}/user/hive';
```
- 在名为Hive的CATALOG下创建一个名为sales的schema：
```
CREATE SCHEMA hive.sales;
```
- 如果当前catalogs下名为traffic的schema不存在时，则创建一个名为traffic的schema：
```
CREATE SCHEMA IF NOT EXISTS traffic;
```
- 创建一个带属性的schema：
```
CREATE DATABASE createtestwithlocation COMMENT 'Holds all values' LOCATION '/user/hive/warehouse/create_new' WITH dbproperties('name'='akku', 'id' ='9');

--通过describe schema|database 语句来查看刚创建的schema
describe schema createtestwithlocation;
```

1.3.2 CREATE TABLE

语法

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 [, ...])]

限制

- 创建分区表时，如果bucket_count为-1且建表语句中未设置buckets，则使用默认值16。
- 默认外部表存储位置{lakeformation_catalog_url}/{*schema_name*}.db/{*table_name*}，其中{lakeformation_catalog_url}对接的lakeformation catalog配置的location，{schema_name}为建表时使用的schema，{table_name}为表名。
- 不允许向托管表（表属性external = true）插入数据。

描述

使用CREATE TABLE创建一个具有指定列的、新的空表。使用CREATE TABLE AS创建带数据的表。

- 使用可选参数IF NOT EXISTS，如果表已经存在则不会报错。
- WITH子句可用于在新创建的表或单列上设置属性，如表的存储位置（location）、是不是外表（external）等。
- LIKE子句用于在新表中包含来自现有表的所有列定义。可以指定多个LIKE子句，从而允许从多个表中复制列。如果指定了INCLUDING PROPERTIES，则将所有表属性复制到新表中。如果WITH子句指定的属性名称与复制的属性名称相同，则将使用WITH子句中的值。默认是EXCLUDING PROPERTIES属性，而且最多只能为一个表指定INCLUDING PROPERTIES属性。
- PARTITIONED BY能够用于指定分区的列；CLUSTERED BY能够被用于指定分桶的列；SORT BY和 SORTED BY能够用于给指定的分桶列进行排序；BUCKETS能够被用于指定分桶数；EXTERNAL可用于指定创建外部表；STORED AS能被用于指定文件存储的格式；LOCATION能被用于指定在OBS上存储的路径。

示例

- 创建一个新表orders，使用子句with指定创建表的存储格式、存储位置、以及是否为外表。

  通过“auto.purge”参数可以指定涉及到数据移除操作（如DROP、DELETE、INSERT OVERWRITE、TRUNCATE TABLE）时是否清除相关数据：

  - "auto.purge"='true'时，清除元数据和数据文件。
  - "auto.purge"='false'时，仅清除元数据，数据文件会移入OBS回收站。默认值为“false”，且不建议用户修改此属性，避免数据删除后无法恢复。

```
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);

-- 通过DESC FORMATTED 语句，可以查看建表的详细信息
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)
```

- 创建一个新表，指定Row format：

```
--建表时，指定表的字段分隔符为‘,’号（如果创建外表，要求数据文件中的每条记录的字段是以逗号进行分隔）
CREATE TABLE student(
id string,birthday string,
grade int,
memo string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```



```
--建表时，指定字段分隔符为'\t'，换行符为'\n'
CREATE TABLE test(
id int,
name string ,
tel string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

- 如果表orders不存在，则创建表orders，并且增加表注释和列注释：

```
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');
```

- 使用表orders的列定义创建表bigger_orders：

```
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)
```

- 标号① 建表示例：

```
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');
```

- 标号② 建表示例：

```
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');
```

- 标号③ 建表示例：

```
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');
```

- 查看表的建表语句：

```
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)
```

创建分区表

```
--创建schema
CREATE SCHEMA hive.web WITH (location = 'obs://bucket/user');
--创建分区表
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
);

--查看分区
SELECT * FROM hive.web."page_views$partitions";
     ds     | country
------------|---------
 2020-07-18 | US
 2020-07-17 | US
--插入数据
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' );

--查询数据
 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

语法

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

限制

① 和 ②的语法不能组合使用。

当使用了avro_schema_url属性时，以下操作是不支持的：

- 不支持CREATE TABLE AS操作
- 使用CREATE TABLE时不支持partitioned_by 和 bucketed_by
- 不支持使用alter table修改column

描述

创建包含SELECT查询结果的新表。

使用CREATE TABLE创建空表。

使用IF NOT EXISTS子句时，如果表已经存在则不会报错。

可选WITH子句可用于设置新创建的表的属性，如表的存储位置（location）、是不是外表（external）等。

示例

- 用指定列的查询结果创建新表orders_column_aliased：

```
CREATE TABLE orders_column_aliased (order_date, total_price)
AS
SELECT orderdate, totalprice FROM orders;
```

- 用表orders的汇总结果新建一个表orders_by_data：

```
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;
```


- 如果表orders_by_date不存在，则创建表orders_by_date：
```
CREATE TABLE IF NOT EXISTS orders_by_date AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate;
```
- 用和表orders具有相同schema创建新表empty_orders table，但是没数据：
```
CREATE TABLE empty_orders AS
SELECT *
FROM orders
WITH NO DATA;
```
- 使用VALUES 创建表，参考 **VALUES**。
- 分区表示例:
```
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

语法

```
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 [,… ] ) ]
```

描述

使用LIKE子句可以在一个新表中包含一个已存在的表所有的列定义。可以使用多个LIKE来复制多个表的列。

如果使用了INCLUDING PROPERTIES，表的所有属性也会被复制到新表，该选项最多只能对一个表生效。

对于从表中复制过来的属性，可以使用WITH子句指定属性名进行修改。

默认使用EXCLUDING PROPERTIES属性。

对于带分区的表，如果用括号包裹like子句，复制的列定义不会包含分区键的信息。

示例

- 创建基础表order01和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);
```

- 创建表orders_like01，它将包含表order01定义的列及表属性

```
CREATE TABLE orders_like01 like order01 INCLUDING PROPERTIES;
```

- 创建表orders_like02，它将包含表order02定义的列，并将表的存储格式设置为‘TEXTFILE’

```
CREATE TABLE orders_like02 like order02 STORED AS TEXTFILE;
```

- 创建表orders_like03，它将包含表order01定义的列及表属性，order02定义的列，以及额外的列c1和c2

```
CREATE TABLE orders_like03 (c1 int,c2 float,LIKE order01 INCLUDING PROPERTIES,LIKE order02);
```

- 创建表orders_like04和orders_like05，它们都会包含同一个表order_partition的定义，但orders_like04不会包含分区键信息，而orders_like05会包含分区键的信息

```
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

语法

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

限制

仅Hive数据源的Catalog支持视图的列描述。

在HetuEngine中创建的视图，视图的定义以编码方式存储在数据源里。在数据源可以查询到该视图，但无法对该视图执行操作。

视图是只读的，不可对它执行LOAD、INSERT操作。

视图可以包含ORDER BY和LIMIT子句，如果关联了该视图的查询语句也包含了这些子句，那么查询语句中的ORDER BY和LIMIT子句将以视图的结果为基础进行运算。

描述

使用SELECT查询结果创建新视图。视图是一个逻辑表，可以被将来的查询所引用，视图中没有数据。该视图对应的查询在每次被其他查询引用该视图时都会被执行。

如果视图已经存在，则可选ORREPLACE子句将导致视图被替换，而不会报错。

示例

- 通过表orders创建一个视图test：
```
CREATE VIEW test (oderkey comment 'orderId',orderstatus comment 'status',half comment 'half') AS
SELECT orderkey, orderstatus, totalprice / 2 AS half FROM orders;
```
- 通过表orders的汇总结果创建视图orders_by_date：
```
CREATE VIEW orders_by_date AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate;
```
- 创建一个新视图来替换已经存在的视图：
```
CREATE OR REPLACE VIEW test AS
SELECT orderkey, orderstatus, totalprice / 4 AS quarter
FROM orders
```
- 创建一个视图的同时设置表属性：
```
create or replace view  view1 comment 'the first view' TBLPROPERTIES('format'='orc') as select * from fruit;
```

注意事项

当使用alter修改创建视图所依赖的表时，需要重新创建视图，否则再次查询视图会报错。

1.3.6 ALTER TABLE

语法

说明

name，new_name，column_name，new_column_name，table_name_*为用户自定义参数。

1. 重命名一个表。

   ALTER TABLE name RENAME TO new_name

2. 修改表的列名，为列添加注释（可选项）和属性（可选项），可参考**描述**查看支持的列属性。

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

3. 删除表中名为column_name的列。

   ALTER TABLE name DROP COLUMN column_name

   须知

   不支持删除分区列或者分桶列。

4. 将表中列名为column_name的列重命名为new_column_name。

   ALTER TABLE name RENAME COLUMN column_name TO new_column_name

须知

不支持重命名分区列或者分桶列。

5. 分区表添加分区。

   ALTER TABLE name ADD [IF NOT EXISTS] PARTITION partition_spec [LOCATION 'location'][PARTITION partition_spec [LOCATION 'location'], ...];

6. 分区表删除分区。这个操作会从分区移除数据和元数据。无论表是internal table还是external table，如果ADD PARTITION时指定了分区保存路径，那么在DROP PARTITION执行后，分区所在文件夹和数据不会被删除。如果ADD PARTITION时未指定分区保存路径，分区目录将从OBS上删除，数据会移到.Trash/Current文件夹。

   ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec[, PARTITION partition_spec, ...];

7. 重命名分区。

   ALTER TABLE table_name PARTITION(partition_key = partition_value1) rename to partition(partition_key = partition_value2)

8. 新增/修改表属性。

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

   说明

   TBLPROPERTIES允许用户通过键值对的方式（属性名和属性都必须是单引号或双引号包裹的字符串），添加或修改连接器支持的表属性，以Hive连接器为例：

   TBLPROPERTIES ("auto.purge"="true")，可能的取值为[true,false]

9. 修改表的列属性。

   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]

须知

- 对一个已经存在的表，修改列名、数据类型、注释、位置（ [FIRST|AFTER column_name] 用于指定列被修改后出现的位置）或者以上任意组合。如果语法中包含了分区子句，那么相应分区的元数据也会一起变动。CASCADE模式会让语法对表和表分区的元数据产生作用，而默认的模式为RESTRICT，对列的修改，仅对表的元数据产生作用。
- 列修改命令只能修改表/分区的元数据，而不会修改数据本身。用户应确保表/分区的实际数据布局符合元数据定义。
- 不支持更改表的分区列/桶列，也不支持更改ORC表。

10. 修改表或分区的存储位置。

    ALTER TABLE table_name [PARTITION partition_spec] SET LOCATION location;

说明

- 可以使用ALTER TABLE [PARTITION] SET位置设置表的表或分区位置。
- 在Set location命令之后，表/分区数据可能不会显示。
- Set location在创建表/分区目录时会使用给定目录路径，而不是hive在创建表/分区时创建的默认路径。
- 该语句不会对表或分区原有数据产生影响，也不会修改原有的表或分区目录，但是新增的数据，都会保存到新指定的目录下。

限制

- ALTER TABLE table_name ADD | DROP col_name命令仅对于ORC/PARQUET存储格式的非分区表可用。

示例

- 将表名从users 修改为 people：
  ALTER TABLE *users* **RENAME TO** *people*;
- 在表users中增加名为zip的列：
  ALTER TABLE *users* **ADD COLUMN** *zip* **varchar;**
- 从表users中删除名为zip的列：
  ALTER TABLE *users* **DROP COLUMN** *zip*;
- 将表users中列名id更改为user_id：
  ALTER TABLE *users* **RENAME COLUMN id TO** *user_id*;
- 修改分区操作：

```
--创建两个分区表
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;

--添加分区
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) ;

--查看分区
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)



--重命名分区
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)

--修改分区表
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;

--修改分区表的存储位置（需要先在obs上创建目录，执行语句后，无法查到之前插入的那条数据）
alter table altercolumn4 partition (c=500) set Location 'obs://bucket/user/hive/warehouse/c500';


--修改列 b 改名为name，同时类型从integer转为string
create table altercolumn1(a integer, b integer) stored as textfile;

alter table altercolumn1 change column b name string;


--查看altercolumn1的属性
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

语法

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

描述

“ALTER VIEW view_name AS select_statement;”用于改变已存在的视图的定义，语法效果与CREATE OR REPLACE VIEW类似。

“ALTER VIEW view_name SET TBLPROPERTIES table_properties;”中table_properties格式为 (property_name = property_value, property_name = property_value, ...)。

视图可以包含Limit和ORDER BY子句，如果关联视图的查询语句也包含了这类子句，则最后执行结果将根据视图的子句运算后得到。例如视图V指定了返回5条数据，而关联查询为select * from V limit 10，则最终只有5条数据返回。

限制

以上两种语法不可混用。

当视图包含分区，那么将无法通过这个语法来改变定义。

示例

```
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

语法

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, ...);

描述

这条命令并不会将SCHEMA当前的内容移动到修改后的路径下，也不会修改与指定schema关联的表或分区，它只会修改新添加进数据库的表的上级目录。

示例

```
Create schema foo;
--修改schema 存储路径
ALTER SCHEMA foo SET LOCATION 'obs://bucket/newlocation';
--修改schema 的所有者
ALTER SCHEMA foo SET OWNER user admin;
```

1.3.9 DROP SCHEMA

语法

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

描述

从Catalog中删除指定的数据库，如果数据库中包含表，则必须在执行DROP DATABASE之前删除这些表，或者使用CASCADE模式。

DATABASE和SCHEMA在概念上是等价可互换的。

[IF EXISTS]

如果目标数据库不存在，将抛出错误提示，但如果使用了IF EXISTS子句则不会抛出错误提示。

[RESTRICT|CASCADE]

可选参数RESTRICT|CASCADE用于指定删除的模式默认是RESTRICT模式，在这种模式下，数据库必须为空，不包含任何表才能删除，如果是CASCADE模式，表示级联删除，会先删除数据库下面的表 ，再删除数据库，该模式请谨慎使用。

示例

- 删除schema web：
  ```
  DROP SCHEMA web;
  ```


- 如果schema sales存在，删除该schema：
```
DROP SCHEMA IF EXISTS sales;
```
- 级联删除schema test_drop，schema test_drop中存在表tb_web，会先删除tb_web，再删除test_drop：
```
CREATE SCHEMA test_drop;

CREATE TABLE tb_web(col1 int);

DROP DATABASE test_drop CASCADE;
```

1.3.10 DROP TABLE

语法

DROP TABLE [IF EXISTS] table_name

描述

删除存在的表，可选参数IF EXISTS指定时，如果删除的表不存在，则不会报错。被删除的数据行将被移动到OBS的回收站。

示例

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

1.3.11 DROP VIEW

语法

DROP VIEW [IF EXISTS] view_name

描述

删除存在的视图，可选参数IF EXISTS指定时，如果删除的视图不存在，则不会报错。

示例

- 创建视图
```
create view orders_by_date as select * from orders;
```
- 删除视图orders_by_date，如果视图不存在则会报错
```
DROP VIEW orders_by_date;
```
- 删除视图orders_by_date，使用参数IF EXISTS，如果视图存在则删除视图，如果视图不存在，也不会报错
```
DROP VIEW IF EXISTS orders_by_date;
```

1.3.12 TRUNCATE TABLE

语法

TRUNCATE TABLE table_name

描述

从表或分区中移除所有行。当表属性“auto.purge”采用默认值“false”时，被删除的数据行将保存到文件系统的回收站，否则，当“auto.purge”设置为“true”时，数据行将被直接删除。

限制

目标表必须是管控表（表属性external=false），否则执行语句将报错。

示例

```
-- 删除原生/管控表
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

语法

COMMENT ON TABLE name IS 'comments'

描述

设置表的注释信息，可以通过设置注释信息为NULL来删除注释。

示例

修改表users的注释为“master table”，表的注释语句可以通过show create table tablename语句查看：

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

1.3.14 VALUES

语法

VALUES row [, ...]

where row is a single expression or

(column_expression [, ...])

描述

VALUES用于查询可以使用的任何地方（例如SELECT、INSERT的FROM子句）。VALUES用于创建了一个没有列名的匿名表，但是表和列可以使用具有列别名的AS子句命名。

示例

- 返回一个1列3行的表：
```
VALUES 1, 2, 3
```
- 返回一个2列3行的表：
```
VALUES
(1, 'a'),
(2, 'b'),
(3, 'c')
```
- 返回具有列名id、name的表：
```
SELECT * FROM (values (1, 'a'), (2, 'b'),(3, 'c')) AS t (id, name);
```
- 创建一个具有列名id、name的新表：
```
CREATE TABLE example AS
 SELECT * FROM (VALUES (1, 'a'), (2, 'b'), (3, 'c')) AS t (id, name);
```

1.3.15 SHOW 语法使用概要

SHOW语法主要用来查看数据库对象的相关信息，其中LIKE子句用来对数据库对象过滤，匹配规则如下，具体示例可参看SHOW TABLES：

规则1：_可以用来匹配单个任意字符。

规则2：%可以用来匹配0个或者任意个任意字符。

规则3：* 可以用来匹配0个或者任意个任意字符。

规则4：|可以用来配置多种规则，规则之间用“|”分隔。

规则5：当想将“_”作为匹配条件时，可以使用ESCAPE 指定一个转义字符，对“_”进行转义，以免按照规则1对“_”进行解析。

1.3.16 SHOW SCHEMAS（DATABASES）

语法

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

描述

该语法中DATABASES和SCHEMAS在概念上是等价的，是可互换的，该语法用于例举所有metastore中定义的schemas。可选子句LIKE可以使用规则运算来过滤结果，它支持的通配符为“*”（匹配任意字符）和“|”(匹配可选项)。

示例

列出当前catalog所有的schemas：
```
SHOW SCHEMAS;
```

列出指定catalog下的schema_name前缀为 " t " 的所有schemas：


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

--等价写法:
SHOW SCHEMAS IN hive LIKE 't%';
```

如果匹配字符串中有字符与通配符冲突，可以指定转义字符来标识，示例为查询hive这个catalog下，schema_name前缀为“pm_”的所有schema，转义字符为“/”：

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

1.3.17 SHOW TABLES

语法

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

描述

这个表达式用于列出指定schema下的所有表。如果没有指定schema，则默认使用当前所在的schema。

可选参数like被用于基于关键字来进行匹配。

示例

```
--创建测试表
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);

--匹配单字符'_'
show tables in  default like 'show_table_';
  Table
-------------
 show_table1
 show_table2
(2 rows)

--匹配多字符'*'，'%'
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)

--转义字符使用,第二个示例将'_'作为过滤条件，结果集不包含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)

--同时满足多个条件，查询default中'show_'开头或者'in'开头的表
show tables in default like 'show$_%|in%' ESCAPE '$';
   Table
-------------
 intable
 show_table1
 show_table2
(3 rows)
```

1.3.18 SHOW TBLPROPERTIES TABLE|VIEW

语法

SHOW TBLPROPERTIES table_name|view_name[(property_name)]

描述

如果不指定属性的关键词，该语句将返回所有的表属性，否则返回给定关键词的属性值。

示例

```
--查看show_table1的所有表属性
                          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)

--查看show_table1的压缩算法
SHOW TBLPROPERTIES show_table1('orc.compression.codec');
SHOW TBLPROPERTIES
---------------------
 GZIP
(1 row)
```

1.3.19 SHOW TABLE/PARTITION EXTENDED

语法

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

描述

用于展示表或分区的详细信息。

可以使用规则运算表达式来同时匹配多个表，但不可用于匹配分区。

展示的信息将包括表的基本信息和相关的文件系统信息，其中文件系统信息包括总文件数、总文件大小、最大文件长度、最小文件长度、最后访问时间以及最后更新时间。如果指定了分区，将给出指定分区的文件系统信息，而不是分区所在表的文件系统信息。

参数说明

- IN | FROM schema_name

  指定schema名称，未指定时默认使用当前的schema。

- LIKE 'identifier_with_wildcards'

  identifier_with_wildcards只支持包含“*”和“|”的规则匹配表达式。

  其中“*”可以匹配单个或多个字符，“|”适用于匹配多种规则匹配表达式中的任意一种的情况，它用于分隔这些规则匹配表达式。

  规则匹配表达式首尾的空格，不会参与匹配计算。

- partition_spec

  一个可选参数，使用键值对来指定分区列表，键值对之间通过逗号分隔。需要注意，指定分区时，表名不支持模糊匹配。

示例

```
-- 演示数据准备
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);

--查询表名以"show"开始的表的详细信息
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)

-- 查询表名以"from"或者"show"开头的表的详细信息
 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)
-- 查询web schema下的page_views表扩展信息
 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 STATS

语法

SHOW STATS FOR table_name;

SHOW STATS FOR (SELECT * FROM table [WHERE condition]);

限制

SHOW STATS 首先会ANALYZE表，参考**ANALYZE**。在ANALYZE之前对目标表进行show stats会显示所有的值都是空值。

描述

返回表的近似统计信息。

返回每一列的统计信息。

列	描述
column_name	列名（汇总行为NULL）
data_size	列中所有值的总大小（以字节为单位）
distinct_values_count	列中不同值的数量
nulls_fraction	列中值为NULL的部分
row_count	行数（仅针对摘要行返回）
low_value	在此列中找到的最小值（仅对于某些类型）
high_value	在此列中找到的最大值（仅适用于某些类型）

示例

```
SHOW STATS FOR orders;
SHOW STATS FOR (SELECT * FROM orders);
```

- 在 Analyze nation表之前：

```
SHOW STATS FOR nation;
column_name | data_size | distinct_values_count | nulls_fraction | row_count | low_value | high_value
-------------|-----------|-----------------------|----------------|-----------|-----------|------------
 name        |      NULL |                  NULL |           NULL |      NULL | NULL      | NULL
 regionkey   |      NULL |                  NULL |           NULL |      NULL | NULL      | NULL
 NULL        |      NULL |                  NULL |           NULL |       6.0 | NULL      | NULL
(3 rows)
```

- 在 Analyze nation表之后：

```
Analyze nation;
ANALYZE: 6 rows

--查询分析后的结果
SHOW STATS FOR nation;
column_name | data_size | distinct_values_count | nulls_fraction | row_count | low_value | high_value
-------------|-----------|-----------------------|----------------|-----------|-----------|------------
 name        |      45.0 |                   5.0 |            0.0 |      NULL | NULL      | NULL
 regionkey   |      NULL |                   2.0 |            0.0 |      NULL | 0         | 2
 NULL        |      NULL |                  NULL |           NULL |       6.0 | NULL      | NULL
(3 rows)
```

1.3.21 SHOW FUNCTIONS

语法

```
SHOW FUNCTIONS [LIKE pattern [ESCAPE escapeChar]];
```


描述

显示所有内置函数的定义信息。

示例

```
SHOW functions;

--使用LIKE子句
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)

--如果匹配字符串中有字符与通配符冲突，可以指定转义字符来标识，示例为查询default这个schema下，table_name前缀为 " t_ " 的所有table，转义字符为 " \ " :
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
```

1.3.22 SHOW PARTITIONS

语法

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

描述

这个表达式用于列出指定的的所有分区。

示例

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

1.3.23 SHOW COLUMNS

语法

SHOW COLUMNS [FROM | IN] table

描述

这个表达式用于列出指定表的列信息。

示例

列出fruit表的列信息：

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

1.3.24 SHOW CREATE TABLE

语法

SHOW CREATE TABLE table_name

描述

显示指定数据表的SQL创建语句。

示例

显示能够创建orders表的SQL 语句：

```
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)
```

1.3.25 SHOW VIEWS

语法

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

描述

列举指定Schema中所有满足条件的视图。

默认使用当前Schema，也可以通过in/from子句来指定Schema。

通过可选子句“LIKE”，筛选视图名满足规则运算表达式的视图，如果不使用这个子句，会列举所有视图。匹配的视图会按字母顺序排列。

目前规则运算表达式只支持“*”（匹配任意字符）。

示例

创建示例所需视图：

```
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)
```

1.3.26 SHOW CREATE VIEW

语法

SHOW CREATE VIEW view_name

描述

显示指定数据视图的SQL创建语句。

示例

显示能够创建order_view视图的SQL语句：

```
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 语法

1.4.1 INSERT

语法

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

限制

- 不支持向同一个表的相同新分区执行并发的插入操作。

- 如果数据表中只有一个字段，且字段类型为row、struct，那么插入数据时需要用row对类型进行包裹。

```
-- 单字段表插入复杂类型需要用row()包裹
CREATE TABLE test_row (id row(c1 int, c2 string));

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

--多字段表复杂类型可以直接插入
CREATE TABLE test_multy_value(id int, col row(c1 int, c2 string));

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

描述

- 向表中插入新的数据行。
- 如果指定了列名列表，那么这些列名列表必须与query语句产生列列表名完全匹配。表中不在列名列表中的每一列，其值会设置为null。
- 如果没有指定列名列表，则query语句产生的列必须与将要插入的列完全匹配。
- 使用insert into时，会往表中追加数据，而使用insert overwrite时，如果表属性“auto.purge”被设置为“true”，直接删除原表数据，再写入新的数据。
- 如果对象表是分区表时，insert overwrite会删除对应分区的数据而非所有数据。
- insert into后面的table关键字为可选，以兼容hive语法。

示例

- 创建fruit和fruit_copy表：

```
create table fruit (name varchar,price double);
create table fruit_copy (name varchar,price double);
```

- 向fruit表中插入一行数据：

```
insert into fruit values('LIchee',32);
-- 兼容写法示例,带上table关键字
insert into table fruit values('Cherry',88);
```

- 向fruit表中插入多行数据：

```
insert into fruit values('banana',10),('peach',6),('lemon',12),('apple',7);
```

- 将fruit表中的数据行加载到fruit_copy表中，执行后表中有5条记录：

```
insert into fruit_copy select * from fruit;
```

- 先清空fruit_copy表，再将fruit中的数据加载到表中，执行之后表中有2条记录：

```
insert overwrite fruit_copy select *  from fruit limit 2;
```

- 对于varchar类型，仅当目标表定义的列字段长度大于源表的实际字段长度时，才可以使用INSERT... SELECT...的形式从源表中查数据并且插入到目标表：

```
create table varchar50(c1 varchar(50));
insert into varchar50 values('hetuEngine');
create table varchar100(c1 varchar(100));
insert into varchar100 select * from varchar50;
```

- 分区表使用insert overwrite语句时，只会清理插入值所在分区的数据，而不是整个表：

```
--创建表
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)
```



```
--清理分区partition(dept_id=25, status='overwrite')，并插入一条数据
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

--清理分区partition(dept_id=10, status='good')，并插入一条数据
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)

--分区表插入数据
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);

-- 填充数据到test_p_1
 insert into test_p_1 partition (provice = 'hebei', city= 'baoding') values ('xiaobei',15),( 'xiaoming',22);
-- 根据test_p_1 插入数据到test_p_2

-- 方式一
from test_p_1 insert into table test_p_2 partition (provice = 'hebei', city= 'baoding') select name,age;

-- 方式二
insert into test_p_2 partition(provice = 'hebei', city= 'baoding') select name,age from test_p_1;
```

1.5 DQL 语法

1.5.1 SELECT

语法

```
[/*+ 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 } ]
```


说明

- from_item 可以是以下形式：
  - 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 可以是以下形式：
  - [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 可以是以下形式：
  - ()
  - expression
  - GROUPING SETS ((column [, ...]) [, ...])
  - CUBE (column [, ...])
  - ROLLUP (column [, ...])

描述

从零个或多个表中检索行数据。

查询stu表的内容。

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

1.5.2 WITH

WITH子句定义查询子句的命名关系，可以展平嵌套查询或简化子查询语句。例如下面的查询语句是等价的：

```
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;
```

多个子查询

```
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 的链式形式

```
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将SELECT语句的输出行划分成包含匹配值的分组。简单的GROUP BY可以包含由输入列组成的任何表达式，也可以是按位置选择输出列的序号。

以下查询是等效的：

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

GROUP BY可以按未出现在SELECT语句输出中的输入列名对输出进行分组。

例如：

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

可以指定多个列进行分组，结果列中不属于分组列的将被设置为NUll。具有复杂分组语法（GROUPING SETS、CUBE或ROLLUP）的查询只从基础数据源读取一次，而使用UNION ALL的查询将读取基础数据三次。这就是当数据源不具有确定性时，使用UNION ALL的查询可能会产生不一致的结果的原因。

```
--创建一个航运表
create table shipping(origin_state varchar(25),origin_zip integer,destination_state
varchar(25) ,destination_zip integer,package_weight integer);

--插入数据
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);

--执行查询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 ));
--这个的查询在逻辑上等同于多个分组查询的union all
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;
--结果
 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

为给定的列生成所有可能的分组，比如 (origin_state, destination_state) 的可能分组为：(origin_state, destination_state)，(origin_state)，(destination_state)，()。

```
SELECT
   origin_state,
   destination_state,
   sum( package_weight )
FROM
   shipping
GROUP BY
   CUBE ( origin_state, destination_state );
--等同于
SELECT
origin_state,
destination_state,
sum( package_weight )
FROM
   shipping
GROUP BY
   GROUPING SETS (
      ( origin_state, destination_state ),
      ( origin_state ),
   ( destination_state ),
   ());
```

ROLLUP

为给定的列集生成部分可能的分类汇总：

```
SELECT
   origin_state,
   origin_zip,
   sum( package_weight )
FROM
   shipping
GROUP BY
   ROLLUP ( origin_state, origin_zip );
--等同于
SELECT
origin_state,
origin_zip,
sum( package_weight )
FROM
   shipping
GROUP BY
   GROUPING SETS ((origin_state,origin_zip ),( origin_state ),());
```

说明

Group by 子句目前不支持使用列的别名，例如：

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

报错如下：

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

1.5.4 HAVING

HAVING

HAVING与聚合函数和GROUP BY一起使用，来控制选在哪些组。HAVING能够在分组和聚合计算之后，过滤掉不满足给定条件的组。

例如：

```
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和EXCEPT都是集合操作。都用来将多个SELECT语句的结果集合并成单个结果集。

UNION

UNION将第一个查询的结果集中的所有行与第二个查询的结果集中的行合并。

query UNION [ALL | DISTINCT] query

ALL和DISTINCT表示是否返回包含重复的行。ALL返回所有的行；DISTINCT返回只包含唯一的行。如果未设置，默认为DISTINCT。

INTERSECT

query INTERSECT [DISTINCT] query

INTERSECT仅返回第一个和第二个查询的结果相交的行。以下是最简单的INTERSECT子句之一的示例。它选择值13和42，并将此结果集与选择值13的第二个查询合并。由于42仅在第一个查询的结果集中，因此不包含在最终结果中。

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

EXCEPT

query EXCEPT [DISTINCT] query

EXCEPT返回在第一个查询结果而不在第二个查询结果中的行。

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


说明

Having子句目前不支持使用列的别名，例如：

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

报错如下：

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

1.5.6 ORDER BY

ORDER BY

ORDER BY子句用于按一个或多个输出表达式对结果集排序。

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

每个expression可以由输出列组成，也可以是按位置选择输出列的序号。

ORDER BY子句在GROUP BY或HAVING子句之后，在OFFSET、LIMIT或FETCH FIRST子句之前进行计算。

须知

按照SQL规范，ORDER BY子句只影响包含该子句的查询结果的行顺序。HetuEngine遵循该规范，并删除该子句的冗余用法，以避免对性能造成负面影响。

例如在执行INSERT语句时，ORDER BY子句不会对插入的数据产生影响，是个冗余的操作，会对整个INSERT语句的整体性能产生负面影响，因此HetuEngine会跳过ORDER BY操作。

- ORDER BY只作用于SELECT子句：
```
INSERT INTO some_table
SELECT * FROM another_table
ORDER BY field;
```
- ORDER BY冗余的例子是嵌套查询，不影响整个语句的结果：
```
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的作用是丢弃结果集中的前若干行数据。

OFFSET count [ROW | ROWS]

如果有ORDER BY，则OFFSET将会作用于排序后的结果集，OFFSET丢弃前若干行数据后保留的数据集，仍然是排序的：

```
SELECT name FROM fruit ORDER BY name OFFSET 3;
name
------------
peach
pear
```



```
watermelon
(3 rows)
```

否则，如果没有使用ORDER BY，被丢弃的行可能是任意的行。如果OFFSET指定的行数等于或超过了结果集的大小，则最终返回的结果为空。

1.5.8 LIMIT | FETCH FIRST

LIMIT和FETCH FIRST都可以限制结果集中的行数。Limit和offset可以配合使用进行分页查询。

LIMIT

LIMIT { count | ALL }

下面的查询限制返回的行数为5：

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

LIMIT ALL 与省略LIMIT的作用一样。

FETCH FIRST

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

FETCH FIRST支持FIRST或NEXT关键字以及ROW或ROWS关键字。这些关键字等效，不影响query执行。

- 如果FETCH FIRST未指定数量，默认为1：

```
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)
```

- 如果使用了OFFSET，则LIMIT或FETCH FIRST会在OFFSET之后应用于结果集：

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

- 对于FETCH FIRST子句，参数ONLY或WITH TIES控制结果集中包含哪些行。

  如果指定了ONLY参数，则结果集将限制为包含参数数量的前若干行。

  如果指定了WITH TIES参数，则要求必须带ORDER BY子句。其结果集中包含符合条件的前若干行基本结果集以及额外的行。这些额外的返回行与基本结果集中最后一行的ORDER BY的参数一样：

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

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

--返回regionkey与第一条相同的所有记录。
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

有BERNOULLI和SYSTEM两种采样方法。

这两种采样方法都不允许限制结果集返回的行数。

BERNOULLI

每一行都将基于指定的采样率选择到采样表中。当使用Bernoulli方法对表进行采样时，将扫描表的所有物理块并跳过某些行（基于采样百分比和运行时计算的随机值之间的比较）。结果中包含一行的概率与任何其他行无关。这不会减少从磁盘读取采样表所需的时间。如果进一步处理采样输出，则可能会影响总查询时间。

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

SYSTEM

此采样方法将表划分为数据的逻辑段，并按此粒度对表进行采样。此采样方法要么从特定数据段中选择所有行，要么跳过它（基于采样百分比与运行时计算的随机值之间的比较）。系统采样中行的选择依赖于使用的connector。例如，如果使用Hive数据源，这将取决于数据在OBS上的布局。这种采样方法不能保证独立的抽样概率。

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

1.5.10 UNNEST

UNNEST可以将ARRAY或MAP展开成relation。ARRAYS展开为单独一列，MAP展开为两列（key，value）。UNNEST还可以与多个参数一起使用，将被展开成多列，行数与最高基数参数相同（其他列用空填充）。UNNEST可以选择使用WITH ORDINALITY子句，在这种情况下，会在末尾添加一个额外的ORDINALITY列。UNNEST通常与JOIN一起使用，可以引用JOIN左侧关系中的列。

使用单独一列

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

使用多个列

```
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

允许合并多个relation的数据。

HetuEngine支持JOIN类型为：CROSS JOIN、INNER JOIN、OUTER JOIN（LEFT JOIN、RIGHT JOIN、FULL JOIN）、SEMIN JOIN和ANTI JOIN。

CROSS JOIN

CROSS JOIN返回两个关系的笛卡尔积。可以使用CROSS JOIN语法指定，也可以在FROM子句中指定多个relation。

以下的query是等价的：

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

INNER JOIN

两个表中至少存在一个相匹配的数据时才返回行，等价于JOIN。也可以转换为等价的WHERE语句，转换方式如下：

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

OUTER JOIN

OUTER JOIN返回符合查询条件的行的同时也返回不符合的行，分为以下三类：

- 左外连接：LEFT JOIN或LEFT OUTER JOIN，表示以左表（nation）为基础返回左表所有的行及右表（region）中相匹配行的数据，若右表中没有匹配，则该行对应的右表的值为空。
- 右外连接：RIGHT JOIN或RIGHT OUTER JOIN，表示以右表（region）为基础返回右表所有的行及左表（nation）中相匹配行的数据，若左表中没有匹配，则该行对应的左表的值为空。
- 全外连接：FULL JOIN或FULL OUTER JOIN，表示只要其中某个表存在匹配，则返回相匹配的行，相当于LEFT JOIN和RIGHT JOIN结合。

```
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

FROM中的子查询可以加上LATERAL关键字，允许引用前面FROM项提供的列：

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

SEMI JOIN、ANTI JOIN

当一张表在另一张表找到匹配的记录之后，半连接（semi-join）返回第一张表中的记录。与条件连接相反，即使在右节点中找到几条匹配的记录，左节点的表也只会返回一条记录。另外，右节点的表一条记录也不会返回。半连接通常使用IN或EXISTS作为连接条件。

而anti-join则与semi-join相反，即当在第二张表没有发现匹配记录时，才会返回第一张表里的记录；当使用not exists/not in的时候会用到。

其他支持的条件包括如下内容：

- where子句中的多个条件
- 别名关系
- 下标表达式

- 解引用表达式
- 强制转换表达式
- 特定函数调用

须知

目前，只在如下情况下支持多个semi/anti join表达式：第一个表中的列在其直接后续的join表达式中被查询，且不与其它join表达式有关系。

示例如下：

```
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

当JOIN的两个relation有相同的列名时，列引用必须使用relation别名（如果relation有别名）或relation名称进行限定：

```
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

EXISTS谓词确定是否返回任意行：

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

IN

确定子查询生成的任意值是否等于给定的表达式。IN的结果遵循null的标准规则。子查询必须只生成一列：

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

1.5.13 SELECT VIEW CONTENT

语法

SELECT column_name FROM view_name

描述

查询视图内容

```
SELECT * FROM test_view;
```

1.6 辅助命令语法

1.6.1 DESCRIBE

语法

DESCRIBE [EXTENDED| FORMATTED] table_name

DESCRIBE [EXTENDED| FORMATTED] table_name PARTITION (partition_spec)

描述

查看指定表的元数据信息。该语法目前只能显示列的元数据信息，等效于语法SHOW COLUMNS。

添加EXTENDED关键字会将表的所有元数据信息以“Thrift”序列化的格式显示出来。

添加FORMATTED关键字会将表的元数据信息以表格的形式展示。

示例

显示fruit数据表的列信息：

```
DESCRIBE fruit;
```

显示fruit 元数据信息：

```
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

语法

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

描述

描述表或分区的列信息，将包含指定表或分区的列的统计数据。

示例

```
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

语法

DESCRIBE DATABASE|SCHEMA [EXTENDED] schema_name

描述

DATABASE和SCHEMA在此处是等价的，可互换的，它们有这相同的含义。

该语法用于显示SCHEMA的名称、注释、还有它在文件系统上的根路径。

可选项EXTENDED可以用来显示SCHEMA的数据库属性。

示例

```
CREATE SCHEMA web;

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

1.6.4 EXPLAIN

语法

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

其中选项可以是以下选项之一：

FORMAT { TEXT | GRAPHVIZ | JSON }

TYPE { LOGICAL | DISTRIBUTED | VALIDATE | IO }

描述

显示一条语句的逻辑的或者分布式的执行计划，也可以用于校验一条SQL语句，或者是分析IO。

参数TYPE DISTRIBUTED用于显示分片后的计划（fragmented plan）。每一个fragment都会被一个或者多个节点执行。Fragments separation表示数据在两个节点之间进行交换。Fragment type表示一个fragment如何被执行以及数据在不同fragment之间怎样分布。

- SINGLE

  Fragment会在单个节点上执行。

- HASH

  Fragment会在固定数量的节点上执行，输入数据通过哈希函数进行分布。

- ROUND_ROBIN

  Fragment会在固定数量的节点上执行，片段在固定数量的节点上执行，输入数据以轮循方式进行分布。

- BROADCAST

  Fragment会在固定数量的节点上执行，输入数据被广播到所有的节点。

- SOURCE

  Fragment在访问输入分段的节点上执行。

示例

- 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
```

- IO：

```
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" : [ ]
  } ]
}
```

1.6.5 ANALYZE

语法

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

描述

收集给定表的表和列统计信息。

可选WITH子句可用于指定connector的属性。使用下面命令可列出所有可用的属性：SELECT * FROM system.metadata.analyze_properties。

示例

- 收集表fruit的统计信息：

```
ANALYZE fruit;
```

- 统计catalog hive、schema default下的表存储：

```
ANALYZE hive.default.orders;
```

- 从hive分区表中统计分区'2020-07-17' , '2020-07-18'信息：

```
ANALYZE hive.web.page_views WITH (partitions = ARRAY[ARRAY['2020-07-17','US'],
ARRAY['2020-07-18','US']]);
```

1.7 预留关键字

表1-4罗列了系统预留的关键字，以及它们在其他SQL标准中是否为预留关键字。如果需要使用这些关键字作为标识符，请加注双引号。

表 1-4 关键字

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

Keyword	SQL：2016	SQL-92
FALSE	reserved	reserved
FOR	reserved	reserved
FROM	reserved	reserved
FULL	reserved	reserved
GROUP	reserved	reserved
GROUPING	reserved	reserved
HAVING	reserved	reserved
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

Keyword	SQL：2016	SQL-92
SELECT	reserved	reserved
TABLE	reserved	reserved
THEN	reserved	reserved
TRUE	reserved	reserved
UESCAPE	reserved	reserved
UNION	reserved	reserved
UNNEST	reserved	reserved
USING	reserved	reserved
VALUES	reserved	reserved
WHEN	reserved	reserved
WHERE	reserved	reserved
WITH	reserved	reserved

1.8 SQL 函数和操作符

1.8.1 逻辑运算符

逻辑运算符

操作	描述	例子
AND	两个值都为true，则为true	a AND b
OR	两个值其中一个为true，则为true	a OR b
NOT	值为false，结果则为true	NOT a

以下真值表反映了AND和OR如何处理NULL值：

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

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

以下真值表反映了NOT如何处理NULL值：

value	NOT value
TRUE	FALSE
FALSE	TRUE
NULL	NULL

1.8.2 比较函数和运算符

比较操作

操作	描述
<	小于
>	大于
<=	小于等于
>=	大于等于
=	等于
<>	不等于
!=	不等于

- 范围比较：between
  between适用于值在一个特定的范围内，如：value BETWEEN min AND max
  Not between适用于值不在某个特定范围内。
  null值不能出现在between操作中，如下两种执行结果都是Null：

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


HetuEngine中，value，min和max 三个参数在between和not between中必须是同一数据类型。

错误示例：'John' between 2.3 and 35.2

BETWEEN等价写法示例：

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

NOT BETWEEN等价写法示例：

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

- IS NULL和IS NOT NULL

  用于判断值是否为空，所有数据类型都可以用于此判断。

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

- IS DISTINCT FROM和IS NOT DISTINCT FROM

  特有用法。在HetuEngine的SQL中，NULL代表未知值，所有与NULL有关的比较，产生的结果也是NULL。IS DISTINCT FROM和IS NOT DISTINCT FROM可以把null值当成某个已知值，从而使结果返回true或者false（即使表达式中有Null值）。

  示例：

  ```
  --建表
  create table dis_tab(col int);
  --插入数据
  insert into dis_tab values (2),(3),(5),(null);
  --查询
  select col from dis_tab where col is distinct from null;
   col
  -----
   2
   3
   5
  (3 rows)
  ```

  如以下真值表，演示了IS DISTINCT FROM和IS NOT DISTINCT FROM对正常数据和NULL值的处理结果：

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和LEAST

  这两个函数不是SQL标准函数，是常用的扩展。参数中不能有Null值。

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

    返回提供的最大值。

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

    返回提供的最小值。

- 批量比较判断：ALL，ANY和SOME

  量词ALL，ANY和SOME可以参考以下方式，结合比较操作符一起使用：

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

  以下是一些量词和比较运算符组合的含义，ANY和SOME具有相同的含义，表中的ANY换为SOME也同样：

表达式	含义
A = ALL (...)	当A与所有值相等时返回true
A <> ALL (...)	当A与任何值都不相等时返回true。
A < ALL (...)	当A小于最小值时返回true。
A = ANY (...)	当A与任意一个值相同时返回true。等价于A IN (...)。
A <> ANY (...)	当A与任意一个值都不相同时返回true。
A < ANY (...)	当A小于最大值时返回true。

  例如：

  ```
  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 条件表达式

CASE

标准的SQL CASE表达式有两种模式。

- “简单模式”从左向右查找表达式的每个value，直到找出相等的expression：

  CASE expression

  WHEN value THEN result

  [WHEN ...]

  [ELSE result]

  END

  返回匹配value的result。如果没有匹配到任何值，则返回ELSE子句的result；如果没有ELSE子句，则返回空。示例：

  ```
  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)
  ```

- “查找模式”从左向右判断每个condition的布尔值，直到判断为真，返回匹配result：

  CASE

  WHEN condition THEN result

  [WHEN ...]

  [ELSE result] END

  如果判断条件都不成立，则返回ELSE子句的result；如果没有ELSE子句，则返回空。示例：

```
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

IF函数是语言结构，它与下面的CASE表达式功能相同：

CASE

WHEN condition THEN true_value

[ELSE false_value] END

- if(condition, true_value)

  如果condition为真，返回true_value；否则返回NULL，true_value不进行计算。

```
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)

  如果condition为真，返回true_value；否则计算并返回false_value 。

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

COALESCE

coalesce(value[, ...])

返回参数列表中的第一个非空value。与CASE表达式相似，仅在必要时计算参数。

可类比MySQL的nvl功能，经常用于转空值为0或者' '（空字符）。

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


NULLIF

- nullif(value1, value2)

  如果value1与value2相等，返回NULL；否则返回value1 。

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

- ZEROIFNULL(value)

  如果value为null，返回0，否则返回原值。目前支持数值类型还有varchar类型。

```
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)

  如果value1为NULL，返回value2，否则，返回value1。

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

- ISNULL(value)

  如果value1为NULL，返回true，否则返回false。

```
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)

  如果value1为NULL，返回false，否则返回true。

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

TRY

评估一个表达式，如果出错，则返回Null。类似于编程语言中的try catch。try函数一般结合COALESCE使用，COALESCE可以将异常的空值转为0或者空，以下情况会被try捕获：

- 分母为0
- 错误的cast操作或者函数入参
- 数字超过了定义长度

不推荐使用，应该明确以上异常，做数据预处理

示例：

假设有以下表，字段origin_zip中包含了一些无效数据：


```
-- 创建表
create  table shipping (origin_state varchar,origin_zip varchar,packages int ,total_cost int);

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

-- 查询数据
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)
```

不使用Try查询失败：

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

使用Try返回NULL：

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

不使用try查询失败：

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

使用TRY和COALESCE返回默认值：

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

1.8.4 Lambda 表达式

Lambda表达式可以用->来表示：

```
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)
```

大部分SQL表达式都可以在Lambda函数体内使用，除了以下场景：

- 不支持子查询
  ```
  x -> 2 + (SELECT 3)
  ```


- 不支持聚合函数
```
x -> max(y)
```

示例

- 通过transform()函数获取数组元素的平方：
```
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)
```
- 利用transform()函数将数组元素转为字符串，无法转换则转为NULL输出，避免报错产生：
```
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$]
(2 rows)
```
- 在对数组元素进行运算时，也能获取其它列来参与运算。例如使用transform()来计算线性方程f(x) =ax + b：
```
SELECT xvalues, a, b, transform(xvalues, x -> a * x + b) as linear_function_values FROM (VALUES (ARRAY[1, 2], 10, 5), (ARRAY[3, 4], 4, 2)) AS t(xvalues, a, b);
 xvalues | a  | b | linear_function_values
---------|----|---|------------------------
 [1, 2]  | 10 | 5 | [15, 25]
 [3, 4]  |  4 | 2 | [14, 18]
(2 rows)
```
- 通过any_match()过滤出至少有一个元素值大于100的数组：
```
SELECT numbers FROM (VALUES (ARRAY[1,NULL,3]), (ARRAY[10,200,30]), (ARRAY[100,20,300])) AS t(numbers) WHERE any_match(numbers, n ->  COALESCE(n, 0) > 100);
    numbers
----------------
 [10, 200, 30]
 [100, 20, 300]
(2 rows)
```
- 使用regexp_replace()将首字母大写：
```
SELECT regexp_replace('once upon a time ...', '^(\w)(\w*)(\s+.*)$',x -> upper(x[1]) || x[2] || x[3]); -- Once upon a time ...
```
- 在聚合函数中应用Lambda表达式。如使用reduce_agg()计算一个较为复杂的按列求元素和：
```
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 转换函数

cast 转换函数

HetuEngine会将数字和字符值隐式转换成正确的类型。HetuEngine不会把字符和数字类型相互转换。例如，一个查询期望得到一个varchar类型的值，HetuEngine不会自动将bigint类型的值转换为varchar类型。

如果有必要，可以将值显式转换为指定类型。

- cast(value AS type) → type

  显式转换一个值的类型。可以将varchar类型的值转为数字类型，反过来转换也可以。

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

- try_cast(value AS type) → type

  与cast()相似，区别是转换失败返回null。

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

说明

当出现数字溢出，null值转换等情况，会返回NULL，但无法转换的情况，还是会报错。

例如：select try_cast(186 as date);

Cannot cast integer to date

Format

- format(format, args...) → varchar

  描述：对一个字符串，按照格式字符串指定的方式进行格式化，并返回。

```
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

  描述：返回按照单位符号格式化的字符串

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

Data Size

parse_presto_data_size函数支持以下单位：

单位	描述	值
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

单位	描述	值
ZB	Zettabytes	1024^7
YB	Yottabytes	1024^8

parse_presto_data_size(string) → decimal(38)

将带单位的格式化的值转为数字，值可以是小数，如下所示：

```
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
```

其它

typeof(expr) → varchar

返回表达式的数据类型名称。

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

1.8.6 数学函数和运算符

数学运算符

运算符	描述
+	加
-	减
*	乘
/	除
%	取余

数学函数

- abs(x) → [same as input]
  返回x的绝对值
  ```
  SELECT abs(-17.4);-- 17.4
  ```
- bin(bigint x) -> string
  返回x的二进制格式
  ```
  select bin(5); --101
  ```
- bround(double x) -> double
  银行家舍入法：

  - 1~4：舍
  - 6~9：进
  - 5的前位数是偶数：舍
  - 5的前位数是奇数：进

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

- bround(double x, int y) -> double
  保留y位小数的银行家舍入法

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

- ceil(x) → [same as input]
  同ceiling()

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

  ceiling(x) → [same as input]
  返回x的向上取整的数值

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

- conv(bigint num, int from_base, int to_base)
- conv(string num, int from_base, int to_base)
  对num做进制转换操作，示例为从10进制转为2进制

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

- rand() → double
  返回0到1之间的随机小数

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

- cbrt(x) → double
  返回x的立方根

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

- e() → double
  返回欧拉常数

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

- exp(x) → double
  返回e的x次方

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

- factorial(int x) -> bigint
  返回x的阶乘，x的有效值范围[0,20]

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

- floor(x) → [same as input]
  返回x舍入最接近的整数

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

- from_base(string, radix) → bigint
  将一个指定进制数转为bigint，如将3进制数'200' 转为十进制数

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

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

如果x为int或二进制形式，则十六进制格式数字以string类型返回。否则，如果x为string，则会将字符串的每个字符转换为十六进制表示形式，并返回结果string

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

- to_base(*x*, *radix*) → varchar

  将一个整数转成radix进制数的字符表示，如将十进制的18转为3进制的表示法

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

- ln(x) → double

  返回x的自然对数

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

- log2(x) → double

  返回x的以2为底的对数

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

- log10(x) → double

  返回x的以10为底的对数

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

- log(b, x) → double

  返回x的以b为底的对数

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

- mod(n, m) → [same as input]

  返回n除以m的模数

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

- pi() → double

  返回圆周率

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

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

  返回x除y的余数的绝对值

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

- pow(x, p) → double

  同power()

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

- power(x,p)

  返回x的p次方

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

- radians(x) → double

  将角度x转为弧度

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

- degrees(x) → double

  将角度x（以弧度表示）转为角度

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


- round(*x*) → [same as input]
  返回x舍入到最近的整数
  ```
  select round(8.57);-- 9
  ```
- round(*x*, *d*) → [same as input]
  x四舍五入到保留d位小数
  ```
  select round(8.57,1);-- 8.60
  ```
- shiftleft(tinyint|smallint|int x, int y) -> int
- shiftleft(bigint x, int y) -> bigint
  返回x左移y个位置的值
  ```
  select shiftleft(8,2);--32
  ```
- shiftright(tinyint|smallint|int a, int b) -> int
- shiftright(bigint a, int b) -> bigint
  返回x右移y个位置的值
  ```
  select shiftright(8,2);--2
  ```
- shiftrightunsigned(tinyint|smallint|int x, int y) -> int
- shiftrightunsigned(bigint x, int y) -> bigint
  按位无符号右移，返回x右移y个位置的值。当x为tinyint、smallint、int时，返回int类型；当x为bigint时，返回bigint类型
  ```
  select shiftrightunsigned(8,3); -- 1
  ```
- sign(*x*) → [same as input]
  返回x的符号函数
  - 如果x=0，返回0
  - x<0，返回-1
  - x>0，返回1

  ```
  select sign(-32.133);-- -1
  select sign(32.133); -- 1
  select sign(0);--0
  ```
  对于double类型的参数
  - 参数是NaN，返回NaN
  - 参数是+∞，返回1
  - 参数是-∞，返回-1

  ```
  select sign(NaN());--NaN
  select sign(Infinity());-- 1.0
  select  sign(-infinity());-- -1.0
  ```
- sqrt(*x*) → double
  返回x的平方根
  ```
  select sqrt(100); -- 10.0
  ```
- truncate(*number,num_digits*)
  - Number需要截尾取整的数字，Num_digits用于指定取整精度的数字
  - Num_digits的默认值为 0
  - truncate ()函数截取时不进行四舍五入

  ```
  select truncate(10.526); -- 10
  select truncate(10.526,2); --  10.520
  ```
- trunc(*number,num_digits*) 参考 truncate(*number,num_digits*)

- unhex(string x) -> binary

  返回十六进制的倒数

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

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

  在具有指定bound1和bound2边界以及n个存储桶的等宽直方图中返回x的容器数量

```
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

  根据数组bin指定的bin返回x的bin数量。bins参数必须是双精度数组，并假定为升序排列

```
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

  描述：计算左边数字除于右边数字的值，会抛弃部分小数部分的值

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

随机数

- rand() → double

  同random()

- random() → double

  返回范围为0.0 <= x <1.0的伪随机值

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

- random(*n*) → [same as input]

  返回介于0和n（不包括n）之间的伪随机数

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

须知

random(n)包含数据类型tinyint，bigint，smallint，integer。

统计学函数

二项分布的置信区间有多种计算公式，最常见的是["正态区间"]，但是，它只适用于样本较多的情况（ np > 5 且 n(1 − p) > 5 ），对于小样本，它的准确性很差。于是采用威尔逊区间：

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

z —— 正态分布，均值 + z * 标准差 置信度。 z = 1.96，置信度为95%

以好评率统计为例，pos是好评数，n是评论总数，phat是好评率

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)

下界值(z1-z2)/m，上界值 (z1+z2)/m

- wilson_interval_lower(successes, trials, z) → double
  返回伯努利试验过程的威尔逊分数区间的下界，置信值由z分数z指定。
  ```
  select wilson_interval_lower(1, 5, 1.96);-- 0.036223160969787456
  ```
- wilson_interval_upper(successes, trials, z) → double
  返回伯努利试验过程的威尔逊分数区间的上界，置信值由z分数z指定。
  ```
  select wilson_interval_upper(1, 5, 1.96);--  0.6244717358814612
  ```
- cosine_similarity(*x*, *y*) → double
  返回稀疏向量x和y之间的余弦相似度。
  ```
  SELECT cosine_similarity (MAP(ARRAY['a'],ARRAY[1.0]),MAP(ARRAY['a'],ARRAY[2.0]));-- 1.0
  ```

累计分布函数

- beta_cdf(*a*, *b*, *v*) → double

用给定的a，b参数计算贝塔分布的累计分布函数：P（ N <v; a，b ）。参数a，b必须为正实数，而值v必须为实数。值v必须位于间隔[0，1]上。

beta分布的累积分布函数公式也称为不完全beta函数比(常用Ix表示)，对应公式：

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

```
select beta_cdf(3,4,0.0004); --  1.278848368599041E-9
```

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

贝塔累计分布函数的逆运算，通过给定累计概率p的a和b参数：P（N <n）。参数a，b必须为正实数，p在区间[0,1]上。

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

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

  给定累积概率（p）：P（N <n）相关的均值和标准偏差，计算正态累计分布函数的逆。平均值必须是实数值，标准偏差必须是正实数值。概率p必须位于间隔（0，1）上。

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

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

  给定平均值和标准差，计算正态分布函数值。P(N<v;mean,sd)，平均值和v必须是实数值，标准差必须是正实数值。

```
select normal_cdf(2, 5, 0.95);--  0.4168338365175577
```

三角函数

所有三角函数的参数都是以弧度表示。参考单位转换函数degrees()和radians()。

- acos(x) → double

  求反余弦值。

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

- asin(x) → double

  求反正弦值。

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

- atan(x) → double

  求x的反正切值。

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

- atan2(y, x) → double

  返回y/x的反正切值。

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

- cos(x) → double

  返回x的余弦值。

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

- cosh(x) → double

  返回x的双曲余弦值。

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

- sin(x) → double

  求x的正弦值。

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

- tan(x) → double

  求x的正切值。

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

- tanh(x) → double

  求x双曲正切值。

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


浮点函数

- infinity() → double
  返回表示正无穷大的常数。
  ```
  select infinity();-- Infinity
  ```
- is_finite(x) → boolean
  判断x是否有限值。
  ```
  select is_finite(infinity());-- false
  select is_finite(50000);--true
  ```
- is_infinite(x) → boolean
  判断x是否无穷大。
  ```
  select is_infinite(infinity());-- true
  select is_infinite(50000);--false
  ```
- is_nan(x) → boolean
  判断x是否非数字。
  ```
  --输入的值必须为double类型
  select is_nan(null); -- NULL
  select is_nan(nan()); -- true
  select is_nan(45);-- false
  ```
- nan() → double
  返回表示非数字的常数。
  ```
  select nan(); -- NaN
  ```

1.8.7 Bitwise 函数

- bit_count(x, bits) → bigint
  计算2的补码表示法中x中设置的位数（视为有符号位的整数）。
  ```
  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
  以二进制补码形式返回x和y按位与的结果。
  ```
  select bitwise_and(8, 7); -- 0
  ```
- bitwise_not(x) → bigint
  以二进制补码形式返回x按位非的结果。
  ```
  select bitwise_not(8);-- -9
  ```
- bitwise_or(x, y) → bigint
  以二进制补码形式返回x和y按位或的结果。
  ```
  select bitwise_or(8,7);-- 15
  ```
- bitwise_xor(x, y) → bigint
  以二进制补码形式返回x和y按位异或的结果。
  ```
  SELECT bitwise_xor(19,25); -- 10
  ```
- bitwise_left_shift(*value*, *shift*) → [same as value]
  描述：返回value左移shift位后的值。
  ```
  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]

  描述：返回value右移shift位后的值。

```
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
-- 右移超过64位，返回0
SELECT bitwise_right_shift( 12, 64); -- 0
```

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

  描述：返回value的算术右移值，当shift小于64位时，返回结果与bitwise_right_shift一样，当移动位数达到或者超过64位时，value是正数时返回0，负数时返回-1：

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

1.8.8 十进制函数和操作符

DECIMAL 字面量

可以使用 DECIMAL 'xxxxxxx.yyyyyyy' 语法来定义 DECIMAL 类型的字面量。

DECIMAL 类型的字面量精度将等于字面量（包括尾随零和前导零）的位数。范围将等于小数部分（包括尾随零）的位数。

示例字面量	数据类型
DECIMAL '0'	DECIMAL(1)
DECIMAL '12345'	DECIMAL(5)
DECIMAL '0000012345.1234500000'	DECIMAL(20, 10)

二进制算术 decimal 运算符

支持标准数学运算符。下表说明了结果的精度和范围计算规则。假设x的类型为DECIMAL(xp, xs)，y的类型为DECIMAL(yp, ys)。

运算	结果类型精度	结果类型范围
x + y 和 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)

如果运算的数学结果无法通过结果数据类型的精度和范围精确地表示，则发生异常情况：Value is out of range。

当对具有不同范围和精度的decimal类型进行运算时，值首先被强制转换为公共超类型。对于接近于最大可表示精度 (38) 的类型，当一个操作数不符合公共超类型时，这可能会导致“值超出范围”错误。例如：decimal(38, 0) 和decimal(38, 1) 的公共超类型是decimal(38, 1)，但某些符合decimal(38, 0) 的值无法表示为decimal(38, 1)。

比较运算符

所有标准比较运算符和BETWEEN运算符都适用于DECIMAL类型。

一元 decimal 运算符

运算符“-”执行取负运算，结果的类型与参数的类型相同。

1.8.9 字符串函数和运算符

字符串运算符

||表示字符连接

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

字符串函数

这些函数假定输入字符串包含有效的UTF-8编码的Unicode代码点。不会显式检查UTF-8数据是否有效，对于无效的UTF-8数据，函数可能会返回错误的结果。可以使用from_utf8来更正无效的UTF-8数据。

此外，这些函数对Unicode代码点进行运算，而不是对用户可见的字符（或字形群集）进行运算。某些语言将多个代码点组合成单个用户感观字符（这是语言书写系统的基本单位），但是函数会将每个代码点视为单独的单位。

lower和upper函数不执行某些语言所需的区域设置相关、上下文相关或一对多映射。

- chr(n) → varchar
  描述：返回Unicode编码值为n的字符值。
  ```
  select chr(100); --d
  ```
- char_length(string) → bigint
  参考length(string)
- character_length(string) → bigint
  参考length(string)
- codepoint(*string*) → integer
  描述：返回单个字符对应的Unicode编码。
  ```
  select codepoint('d'); --100
  ```
- concat(string1, string2) → varchar
  描述：字符串连接。
  ```
  select concat('hello','world'); -- helloworld
  ```
- concat_ws(string0, string1, ..., stringN) → varchar
  描述：将string1、string2、...,stringN，以string0作为分隔符串联成一个字符串。如果string0为null，则返回值为null。分隔符后的参数如果是NULL值，将会被跳过。


```
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

  描述：将数组中的元素以string0为分隔符进行串联。如果string0为null，则返回值为null。数组中的任何null值都将被跳过。

```
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

  描述：根据给定的字符集将第一个参数编码为字符串，支持的字符集包括（'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'），当第一个参数为null，将返回null。

```
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

  描述：字符串按照给定的字符集进行编码。

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

- find_in_set (string str, string strList) →int

  描述：返回str在逗号分隔的strList中第一次出现的位置。当有参数为null时，返回值也为null。

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

- format_number(number x, int d) →string

  描述：将数字x格式化为'#,###,###.##'，保留d位小数，以字符串的形式返回结果。

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

- format(format,args...) → varchar

  描述：参见**Format**。

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

  描述：返回子串在字符串的第pos位后第一次出现的位置。没有满足条件的返回0。

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

- length(string) → bigint

  描述：返回字符串的长度。

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

- levenshtein_distance(string1, string2) → bigint

描述：计算string1和string2的Levenshtein距离，即将string转为string2所需要的单字符编辑（包括插入、删除或替换）最少次数。

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

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

  描述：返回字符串1和字符串2的汉明距离，即对应位置字符不同的数量。 请注意，两个字符串的长度必须相同。

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

- instr(string,substring) → bigint

  描述：查找substring 在string中首次出现的位置。

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

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

  描述：返回字符串1和字符串2的Levenshtein编辑距离，即将字符串1更改为字符串2所需的最小单字符编辑（插入，删除或替换）次数。

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

- lower(string) → varchar

  描述：将字符转换为小写。

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

- lcase(string A) → varchar

  描述：同lower(string)。

- ltrim(string) → varchar

  描述：去掉字符串开头的空格。

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

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

  描述：右填充字符串以使用padstring调整字符大小。如果size小于字符串的长度，则结果将被截断为size个字符。大小不能为负，并且填充字符串必须为非空。

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

- luhn_check(string) → boolean

  描述：根据Luhn算法测试数字字符串是否有效。

  这种校验和函数，也称为模10，广泛应用于信用卡号码和政府身份证号码，以区分有效号码和键入错误、错误的号码。

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

- octet_length(string str) → int

  描述：返回用于保存UTF-8编码的字符串str的字节数。

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

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

  描述：返回URL的指定部分。partToExtract参数有效值包括：HOST、PATH、QUERY、REF、PROTOCOL、AUTHORITY、FILE和USERINFO。keyToExtract为可选参数，用于选取QUERY中的key对应的值。

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



```
-- 查询URL中QUERY部分service对应的值
select parse_url('https://www.example.com/query/index.html?name=panda','QUERY','name');
 _col0
-------
 panda
(1 row)
```

- position(*substring IN string*) →bigint

  描述：返回子串在父串中第一次出现的位置

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

- quote(String text) → string

  描述：返回单引号包裹的字符串。不支持含单引号的字符串。

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

- repeat2(string str, int n) → string

  描述：返回str重复n次获得的字符串。

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

- replace(string, ‘a’) → varchar

  描述：去掉字符串中的a字符。

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

- replace(string, ‘a’, ‘b’) → varchar

  描述：把字符串中所有的a字符 替换为b。

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

- reverse(string) → varchar

  描述：字符串倒序。

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

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

  描述：右填充字符串以使用padstring调整字符大小。如果size小于字符串的长度，则结果将被截断为size个字符。大小不能为负，并且填充字符串必须为非空。

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

- rtrim(string) → varchar

  描述：去掉字符串尾部的空格。

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

- space(int n) → varchar

  描述：返回n个空格。

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

  (1 row)

  select length(space(4));
   _col0
  -------
      4
  (1 row)
  ```

- split(string, delimiter) → array

描述：将字符串按限定符（delimiter）分隔为一个array。

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

- split(string, delimiter, limit) → array

  描述：将字符串按delimiter分割为一个array，元素个数为limit。最后一个元素包含了最后一个字符串后面所有的字符。Limit 必须是个数字。

  ```
  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

  描述：将字符串按delimiter分隔为一个array，并取出索引值为index的元素。index从1开始，如果index超过了数组长度，则返回null。

  ```
  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>

  描述：将字符串按entryDelimiter分割为Map的键值对，而每个键值对又按照keyValueDelimiter来区分Key和Value。

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

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

  描述：将字符串按照entryDelimiter和keyValueDelimiter分割，返回一个map，每个key对应一个类型为array的value。其中，entryDelimiter将字符串分割为键值对，keyValueDelimiter将键值对分割为Key和Value。

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

- strpos(string, substring) → bigint

  描述：返回字符串中第一次出现substring的位置。从1开始，如果未找到，返回0。举例：

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

- str_to_map() 参考split_to_map()
- substr(string, start) → varchar

  描述：从start位置开始截取字符串。

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

- substr(string, start, length) → varchar

  描述：从start位置开始截取字符串，截取的长度为length。

  一般用于截取时间戳格式。

  ```
  Select substr('2019-03-10 10:00:00',1,10); --截取到日 2019-03-10
  Select substr('2019-03-10 10:00:00',1,7); --截取到月 2019-03
  ```

- substring(string, start) → varchar

  参考substr(string, start)

- substring_index(string A, string delim, int count) → varchar

  描述：当count为正数时，返回从左边开始计数的第count个分隔符delim左边的所有内容。当count为负数时，返回从右边开始计数的第count个分隔符delim右侧的所有内容。

  ```
  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

  描述：SOUNDEX返回由四个字符组成的代码（SOUNDEX）以评估两个字符串在发音时的相似性。规则如下：

  表 1-5 字符对应规则

字符	对应数字
a、e、h、i、o、u、w、y	0
b、f、p、v	1
c、g、j、k、q、s、x、z	2
d、t	3
l	4
m、n	5
r	6

  - 提取字符串的首字母作为soundex的第一个值。
  - 按照上面的字母对应规则，将后面的字母逐个替换为数字。如果有连续的相等的数字，只保留一个，其余的都删除掉，并去除所有的0。
  - 如果结果超过4位，取前四位。如果结果不足4位向后补0。

    ```
    select soundex('Miller');
     _col0
    -------
     M460
    (1 row)
    ```

- translate(string|char|varchar input, string|char|varchar from, string|char|varchar to) →varchar

  描述：对于input字符串，将其中的参数from指代字符串替换为参数to指代的字符串。三个参数有一个为NULL，则结果返回NULL。

  ```
  select translate('aabbcc','bb','BB');
   _col0
  --------
   aaBBcc
  (1 row)
  ```

- trim(string) → varchar

  描述：去掉字符串首尾的空格。

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


- btrim(String str1,String str2) → varchar
  描述：从str1首尾去掉str2中包含的所有字符。
  ```
  select btrim('hello','hlo');-- e
  ```
- upper(string) → varchar
  描述：将字符串转为大写。
  ```
  select upper('heLLo');-- HELLO
  ```
- ucase(string A) → varchar
  描述：同upper(string)。
- base64decode(STRING str)
  描述：对字符串进行base64反编码。
  ```
  SELECT to_base64(CAST('hello world' as varbinary));-- aGVsbG8gd29ybGQ=
  select base64decode('aGVsbG8gd29ybGQ=');-- hello world
  ```
- jaro_distance(STRING str1, STRING str2)
  描述：比较两个字符串的相似度。
  ```
  select JARO_DISTANCE('hello', 'hell');-- 0.9333333333333332
  ```
- FNV_HASH(type v)
  描述：计算字符串的hash值。
  ```
  select FNV_HASH('hello');-- -6615550055289275125
  ```
- word_stem(word) → varchar
  描述：返回英语单词的词干。
  ```
  select word_stem('greating');-- great
  ```
- word_stem(*word*, *lang*) → varchar
  描述：返回指定语种单词中的词干。
  ```
  select word_stem('ultramoderne','fr');-- ultramodern
  ```
- translate(*source*, *from*, *to*) → varchar
  描述：通过将源字符串中找到的字符替换为目标字符串中的相应字符来返回翻译后的源字符串。如果from字符串包含重复项，则仅使用第一个。如果源字符在from字符串中不存在，则将复制源字符而不进行翻译。如果在from字符串中匹配字符的索引超出了to字符串的长度，则将从结果字符串中省略源字符。
  ```
  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函数

- normalize(string) → varchar
  描述：返回NFC形式的标准字符串。
  ```
  select normalize('e');
   _col0
  -------
   e
  (1 row)
  ```
- normalize(string, form) → varchar
  描述：Unicode允许你用不同的字节来写相同的字符，例如é和é，第一个是由0xC3 0xA9这两个字节组成的，第二个是由0x65 0xCC 0x81这三个字节组成的。

normalize()将根据参数form给定的Unicode规范化形式（包括NFC、NFD、NFKC、NFKD）返回标准字符串，如未指定参数，默认使用NFC。

```
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

  将字符串编码为utf8格式字符串。

```
select to_utf8('panda');
     _col0
----------------
 70 61 6e 64 61
(1 row)
```

- from_utf8(binary) → varchar

  描述：将一个二进制串编码为UTF-8格式字符串。无效的UTF-8序列将被Unicode字符U+FFFD替换。

```
select from_utf8(X'70 61 6e 64 61');
 _col0
-------
 panda
(1 row)
```

- from_utf8(binary, replace) → varchar

  描述：将一个二进制串编码为UTF-8格式字符串。无效的UTF-8序列将被参数replace替换。参数replace必须为单个字符或空（以免无效字符被移除）。

```
select from_utf8(X'70 61 6e 64 61 b1','!');
 _col0
--------
 panda!
(1 row)
```

1.8.10 正则表达式函数

概述

所有的正则表达式函数都使用Java样式的语法。但以下情况除外：

- 使用多行模式（通过（？ m）标志启用）时，只有\ n被识别为行终止符。 此外，不支持（？ d）标志，因此不能使用。
- 大小写区分模式（通过（？ i）标志启用）时，总是以unicode的模式去实现。同时，不支持上下文敏感匹配和局部敏感匹配。此外，不支持（？ u）标志。
- 不支持Surrogate Pair编码方式。例如，\ uD800 \ uDC00不被视为U + 10000，必须将其指定为\ x {10000}。
- 边界字符（\b）无法被正确处理，因为它一个不带基字符的非间距标记。

- \Q和\E在字符类（如[A-Z123]）中不受支持，而是作为文本处理。
- 支持Unicode字符类（\ p {prop}），但有以下差异：
  - 名称中的所有下划线都必须删除。例如，使用OldItalic而不是Old_Italic
  - 必须直接指定脚本，不能带Is，script =或sc =前缀。示例：\ p {Hiragana}
  - 必须使用In前缀指定块。不支持block =和blk =前缀。示例：\p{Mongolian}
  - 必须直接指定类别，而不能带Is，general_category =或gc =前缀。示例：\p{L}
  - 二进制属性必须直接指定，而不是Is。示例：\p{NoncharacterCodePoint}

函数

- regexp_count(*string*, *pattern*) → bigint

  描述：返回字符串中pattern匹配的次数。

  ```
  SELECT regexp_count('1a 2b 14m', '\s*[a-z]+\s*'); -- 3
  ```

- regexp_extract_all(string, pattern) -> array(varchar)

  描述：以数组格式返回匹配的所有子串。

  ```
  SELECT regexp_extract_all('1a 2b 14m','\d+');-- [1, 2, 14]
  ```

- regexp_extract_all(string, pattern, group) -> array(varchar)

  描述：当pattern包含多个分组时，用group指定返回满足**被捕获分组**的所有子串。

  ```
  SELECT regexp_extract_all('1a 2b 14m','(\d+)([a-z]+)',2);-- [a, b, m]
  ```

- regexp_extract(string, pattern) → varchar

  描述：返回与字符串中的正则表达式模式匹配的第一个子字符串。

  ```
  SELECT regexp_extract('1a 2b 14m','\d+');-- 1
  ```

- regexp_extract(string, pattern, group) → varchar

  描述：当pattern包含多个分组时，用group指定返回满足**被捕获分组**的第一个子字符串。

  ```
  SELECT regexp_extract('1a 2b 14m','(\d+)([a-z]+)',2);-- 'a'
  ```

- regexp_like(string, pattern) → boolean

  描述：验证字符串是否包含满足正则表达式的子串，如果有，返回true。

  ```
  SELECT regexp_like('1a 2b 14m','\d+b');-- true
  ```

- regexp_position(string, pattern) → integer

  描述：返回字符串中pattern第一次匹配到的索引。没有匹配的项则返回-1。

  ```
  SELECT regexp_position('I have 23 apples, 5 pears and 13 oranges', '\b\d+\b'); -- 8
  ```

- regexp_position(string, pattern, start) → integer

  描述：返回字符串从start（含start）开始pattern第一次匹配到的项的索引。没有匹配的项则返回-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

  描述：返回字符串中从索引start（含start）开始，pattern第occurrence次匹配到的项的索引。没有匹配的项则返回-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

  描述：从目标字符串中移除满足正则表达式的子串。

  ```
  SELECT regexp_replace('1a 2b 14m','\d+[ab] ');-- '14m'
  ```

- regexp_replace(string, pattern, replacement) → varchar

  描述：使用replacement替换目标字符串中满足正则表达式的子串。如果replacement中包含'$'字符，使用'\$' 进行转义。在替换中，可以对编号组使用$g引用捕获组，对命名组使用${name}引用捕获组。

  ```
  SELECT regexp_replace('1a 2b 14m','(\d+)([ab]) ','3c$2 ');-- '3ca 3cb 14m'
  ```

- regexp_replace(string, pattern, function) → varchar

  描述：使用function替换与字符串中的正则表达式模式匹配的子字符串的每个实例。对于每个匹配，以数组形式传递的**捕获组**都会调用lambda表达式函数。捕获组号从1开始；整个匹配没有分组（如果需要，请用括号将整个表达式括起来）。

  ```
  SELECT regexp_replace('new york','(\w)(\w*)',x->upper(x[1])||lower(x[2]));--'New York'
  ```

- regexp_split(string, pattern) -> array(varchar)

  描述：使用正则表达式模式拆分字符串并返回一个数组。尾随的空字符串被保留。

  ```
  SELECT regexp_split('1a 2b 14m','\s*[a-z]+\s*');-- [1, 2, 14, ]
  ```

1.8.11 二进制函数和运算符

二进制运算符

|| 运算符执行连接。

二进制函数

- length(binary) → bigint

  返回binary的字节长度。

  ```
  select length(x'00141f');-- 3
  ```

- concat(binary1, ..., binaryN) → varbinary

  将binary1，binary2，binaryN串联起来。这个函数返回与SQL标准连接符||相同的功能。

  ```
  select concat(X'32335F',x'00141f'); -- 32 33 5f 00 14 1f
  ```

- to_base64(binary) → varchar

  将binary编码为base64字符串表示。

  ```
  select to_base64(CAST('hello world' as binary)); -- aGVsbG8gd29ybGQ=
  ```

- from_base64(string) → varbinary

  将base64编码的string解码为varbinary。

  ```
  select from_base64('helloworld'); --  85 e9 65 a3 0a 2b 95
  ```

- unbase64(string) → varbinary

  将base64编码的string解码为varbinary。

  ```
  SELECT from_base64('helloworld'); --  85 e9 65 a3 0a 2b 95
  ```

- to_base64url(binary) → varchar

  使用URL安全字符，将binary编码为base64字符串表示。

  ```
  select to_base64url(x'555555');  -- VVVV
  ```


- from_base64url(string) → varbinary
  使用URL安全字符，将base64编码的string解码为二进制数据。
  ```
  select from_base64url('helloworld'); --  85 e9 65 a3 0a 2b 95
  ```
- to_hex(binary) → varchar
  将binary编码为16进制字符串表示。
  ```
  select to_hex(x'15245F');  --  15245F
  ```
- from_hex(string) → varbinary
  将16进制编码的string解码为二进制数据。
  ```
  select from_hex('FFFF'); --  ff ff
  ```
- to_big_endian_64(*bigint*) → varbinary
  将bigint类型的数字编码为64位大端补码格式。
  ```
  select to_big_endian_64(1234);
           _col0
  ------------------------
   00 00 00 00 00 00 04 d2
  (1 row)
  ```
- from_big_endian_64(*binary*) → bigint
  64位大端补码格式的二进制解码为bigint类型的数字。
  ```
  select from_big_endian_64(x'00 00 00 00 00 00 04 d2');
   _col0
  -------
    1234
  (1 row)
  ```
- to_big_endian_32(*integer*) → varbinary
  将bigint类型的数字编码为32位大端补码格式。
  ```
  select to_big_endian_32(1999);
     _col0
  -------------
   00 00 07 cf
  (1 row)
  ```
- from_big_endian_32(*binary*) → integer
  32位大端补码格式的二进制解码为bigint类型的数字。
  ```
  select from_big_endian_32(x'00 00 07 cf');
   _col0
  -------
    1999
  (1 row)
  ```
- to_ieee754_32(*real*) → varbinary
  根据IEEE 754算法，将单精度浮点数编码为一个32位大端字节序的二进制块。
  ```
  select to_ieee754_32(3.14);
     _col0
  -------------
   40 48 f5 c3
  (1 row)
  ```
- from_ieee754_32(*binary*) → real
  对采用IEEE 754单精度浮点格式的32位大端字节序binary进行解码。
  ```
  select from_ieee754_32(x'40 48 f5 c3');
   _col0
  -------
    3.14
  (1 row)
  ```
- to_ieee754_64(*double*) → varbinary

根据IEEE 754算法，将双精度浮点数编码为一个64位大端字节序的二进制块。

```
select to_ieee754_64(3.14);
 _col0
-------------------------
 40 09 1e b8 51 eb 85 1f
(1 row)
```

- from_ieee754_64(*binary*) → double

  对采用IEEE 754单精度浮点格式的64位大端字节序binary进行解码。

```
 select from_ieee754_64(X'40 09 1e b8 51 eb 85 1f');
 _col0
--------
  3.14
(1 row)
```

- lpad(*binary*, *size*, *padbinary*) → varbinary

  左填充二进制以使用padbinary调整字节大小。如果size小于二进制文件的长度，则结果将被截断为size个字符。size不能为负，并且padbinary不能为空。

```
select lpad(x'15245F', 11,x'15487F') ; -- 15 48 7f 15 48 7f 15 48 15 24 5f
```

- rpad(*binary*, *size*, *padbinary*) → varbinary

  右填充二进制以使用padbinary调整字节大小。如果size小于二进制文件的长度，则结果将被截断为size个字符。size不能为负，并且padbinary不能为空。

```
SELECT rpad(x'15245F', 11,x'15487F'); -- 15 24 5f 15 48 7f 15 48 7f 15 48
```

- crc32(*binary*) → bigint

  计算二进制块的CRC 32值。

- md5(*binary*) → varbinary

  计算二进制块的MD 5哈希值。

- sha1(*binary*) → varbinary

  计算二进制块的SHA 1哈希值。

- sha2(*string, integer*) → string

  安全散列算法2， 是一种密码散列函数算法标准，其输出长度可以取224位，256位， 384位、512位，分别对应SHA-224、SHA-256、SHA-384、SHA512

- sha256(*binary*) → varbinary

  计算二进制块的SHA 256哈希值。

- sha512(*binary*) → varbinary

  计算二进制块的SHA 512哈希值。

- xxhash64(*binary*) → varbinary

  计算二进制块的XXHASH 64哈希值。

- spooky_hash_v2_32(*binary*) → varbinary

  计算二进制块的32位SpookyHashV2哈希值。

- spooky_hash_v2_64(*binary*) → varbinary

  计算二进制块的64位SpookyHashV2哈希值。

- hmac_md5(*binary*, *key*) → varbinary

  使用给定的key计算二进制块的HMAC值（采用 md5）。

- hmac_sha1(*binary*, *key*) → varbinary

  使用给定的key计算二进制块的HMAC值（采用 sha1）。

- hmac_sha256(*binary*, *key*) → varbinary
  使用给定的key计算二进制块的HMAC值（采用 sha256 ）。
- hmac_sha512(*binary*, *key*) → varbinary
  使用给定的key计算二进制块的HMAC值（采用 sha512 ）。

须知

CRC32、MD5、SHA1算法在密码学场景已被攻击者破解，不建议应用于密码学安全场景。

1.8.12 Json 函数和运算符

- 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 函数

说明

NULL到JSON的转换并不能简单地实现。从独立的NULL进行转换将产生一个SQLNULL，而不是JSON 'null'。不过，在从包含NULL的数组或Map进行转换时，生成的JSON将包含NULL。

在从ROW转换为JSON时，结果是一个JSON数组，而不是一个JSON对象。这是因为对于SQL中的行，位置比名称更重要。

支持从BOOLEAN、TINYINT、SMALLINT、INTEGER、BIGINT、REAL、DOUBLE或VARCHAR进行转换。当数组的元素类型为支持的类型之一、Map的键类型是VARCHAR且Map的值类型是支持的类型之一或行的每个字段类型是支持的类型之一时支持从ARRAY、MAP或ROW进行转换。下面通过示例展示了转换的行为：

```
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 转其它类型

```
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']
```

说明

在从JSON转换为ROW时，支持JSON数组和JSON对象。

JSON数组可以具有混合元素类型，JSON Map可以有混合值类型。这使得在某些情况下无法将其转换为SQL数组和Map。为了解决该问题，HetuEngine支持对数组和Map进行部分转换：

```
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

  判断json是否为标量（即JSON数字、JSON字符串、true、false或null）：

  ```
  select is_json_scalar(json'[1,22]'); -- false
  ```

- json_array_contains(json, value) → boolean

  判断json中是否包含某value

  ```
  select json_array_contains(json '[1,23,44]',23); -- true
  ```

- json_array_get(json_array, index) → json

  须知

  该函数的语义已被破坏。如果提取的元素是字符串，它将被转换为未正确使用引号括起来的无效JSON值（值不会被括在引号中，任何内部引号不会被转义）。建议不要使用该函数。无法在不影响现有用法的情况下修正该函数，可能会在将来的版本中删除该函数。

  返回指定索引位置的json元素，索引从0开始

  ```
  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]'
  ```

  索引页支持负数，表示从最后开始，-1表示最后一个元素，索引超过实际长度会返回null

  ```
  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]'
  ```

  如果指定索引位置的json元素不存在，将返回NULL值

  ```
  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

  返回json的长度

  ```
  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)；

  按照json_path格式抓取json中的信息

  ```
  SELECT get_json_object('{"id": 1, "value":"xxx"}', '$.value');  -- "xxx"
  ```


- json_extract(json, json_path) → json

  按照json_path格式抓取json中的信息

  ```
  SELECT json_extract(json '{"id": 1, "value":"xxx"}', '$.value');-- JSON "xxx"
  ```

- json_extract_scalar(json, json_path) → varchar

  和json_extract功能相同，返回值是varchar

  ```
  SELECT json_extract_scalar(json '{"id": 1, "value": "xxx"}', '$.value'); -- xxx
  ```

- json_format(json) → varchar

  把json值转为序列化的json文本，这是json_parse的反函数：

  ```
  SELECT JSON_format(json '{"id": 1, "value":"xxx"}'); -- {"id":1, "value":"xxx"}
  ```

  注意：

  json_format和CAST(json AS VARCHAR)具有完全不同的语义。

  json_format将输入JSON值序列化为遵守7159标准的JSON文本。JSON值可以是JSON对象、JSON数组、JSON字符串、JSON数字、true、false或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)将JSON值转换为对应的SQL VARCHAR值。对于JSON字符串、JSON数字、true、false或null，转换行为与对应的SQL类型相同。JSON对象和JSON数组无法转换为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

  和json_format(json)功能相反，将json格式的字符串转换为json

  Json_parse和json_extract通常结合使用，用于解析数据表中的json字符串

  ```
  select JSON_parse('{"id": 1, "value":"xxx"}'); -- json {"id":1, "value":"xxx"}
  ```

- json_size(json, json_path) → bigint

  和json_extract类似，但是返回的是json里的对象个数

  ```
  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 日期、时间函数及运算符

日期时间运算符

运算符	示例	结果
+	date '2012-08-08' + interval '2' day	2012-08-10
+	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

运算符	示例	结果
+	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

时区转换

运算符：AT TIME ZONE，用于设置一个时间戳的时区。

```
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
```

日期时间函数

- current_date -> date
  返回当前日期(utc时区)
  ```
  select current_date; -- 2020-07-25
  ```
- current_time -> time with time zone
  返回当前时间(utc时区)
  ```
  select current_time;-- 16:58:48.601+08:00
  ```
- current_timestamp -> timestamp with time zone
  返回当前时间戳(当前时区)
  ```
  select current_timestamp; -- 2020-07-25 11:50:27.350 Asia/Singapore
  ```
- current_timezone() → varchar
  返回当前时区
  ```
  select current_timezone();-- Asia/Singapore
  ```
- date(x) → date
  将日期字面量转换成日期类型的变量
  ```
  select date('2020-07-25');-- 2020-07-25
  ```
- from_iso8601_timestamp(string) → timestamp with time zone
  将ISO 8601格式的时戳字面量转换成带时区的时戳变量


```
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

  将ISO 8601格式的日期字面量转换成日期类型的变量

  ```
  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

  将UNIX时戳转换为时间戳变量（当前时区）

  ```
  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

  将UNIX时戳转换成时戳变量，可以带时区选项

  ```
  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

  将UNIX时戳转换成带时区的时戳变量，hours和minutes表示时区偏移量

  ```
  select from_unixtime(1.595658735E9, 8, 30);-- 2020-07-25 14:32:15.000 +08:30
  ```

- localtime -> time

  获取当前时间

  ```
  select localtime;-- 14:16:13.096
  ```

- localtimestamp -> timestamp

  获取当前时间戳

  ```
  select localtimestamp;-- 2020-07-25 14:17:00.567
  ```

- months_between(date1, date2) -> double

  返回date1和date2之间的月数，如果date1比date2迟，结果就是正数，那么结果就是负数；如果两个日期的日数相同，那么结果就是整数，否则按照每月31天以及时分秒的差异来计算小数部分。date1和date2的类型可以是date，timestamp，也可以是“yyyy-MM-dd”或“yyyy-MM-dd HH:mm:ss”格式的字符串

  ```
  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

  获取当前时间，current_timestamp的别名

  ```
  select now();-- 2020-07-25 14:39:39.842 Asia/Singapore
  ```

- unix_timestamp()

  获取当前unix时间戳

  ```
  select unix_timestamp(); -- 1600930503
  ```

- to_iso8601(x) → varchar

  将x转换成ISO8601格式的字符串。这里x可以是DATE、TIMESTAMP [with time zone]这几个类型

  ```
  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

  获取当前距当天零时已经过去的毫秒数

  ```
  select to_milliseconds(interval '8' day to second);-- 691200000
  ```


- to_unixtime(timestamp) → double
  将时间戳转换成UNIX时间
  ```
  select to_unixtime(cast('2020-07-25 14:32:15.147' as timestamp));-- 1.595658735147E9
  ```
- trunc(string date, string format) →string
  按照format格式去截取日期值，支持的格式有：MONTH/MON/MM，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
  ```

  说明

  使用下列 SQL 标准函数时，兼容使用圆括号的方式：
  - current_date
  - current_time
  - current_timestamp
  - localtime
  - Localtimestamp

  如：select current_date();

截取函数

类似于保留几位小数的操作，函数 date_trunc 支持如下单位：

单位	截取后的值
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

上面的例子使用时间戳2001-08-22 03:04:05.321作为输入。

date_trunc(unit, x) → [same as input]

返回x截取到单位unit之后的值。

```
select date_trunc('hour', timestamp '2001-08-22 03:04:05.321'); -- 2001-08-22 03:00:00.000
```

间隔函数

本章中的函数支持如下所列的间隔单位：

单位	描述
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]

  在timestamp的基础上加上value个unit。如果想要执行相减的操作，可以通过将value赋值为负数来完成。

  ```
  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

  返回timestamp2 - timestamp1之后的值，该值的表示单位是unit。

  unit的值是字符串。例如：‘day’、‘week’、‘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]

  描述：日期加法。输入的类型可以是date或timestamp，表示对日期做加减，当做减法时，bigint对应值为负。

  ```
  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
  ```

持续时间函数

持续时间可以使用以下单位：

单位	描述
ns	纳秒
us	微秒
ms	毫秒

单位	描述
s	秒
m	分钟
h	小时
d	天

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 日期函数

在这一章节使用与MySQL date_parse和str_to_date方法兼容的格式化字符串。

- date_format(timestamp, format) → varchar
  使用format格式化timestamp。
  ```
  select date_format(timestamp '2020-07-22 15:00:15', '%Y/%m/%d');-- 2020/07/22
  ```
- date_parse(string, format) → timestamp
  按format格式解析日期字面量。
  ```
  select date_parse('2020/07/20', '%Y/%m/%d');-- 2020-07-20 00:00:00.000
  ```

下面的表格是基于MySQL手册列出的，描述了各种格式化描述符：

格式化描述符	描述
%a	对应的星期几（Sun .. Sat）
%b	对应的月份（Jan .. Dec）
%c	对应的月份（1 .. 12）
%D	对应该月的第几天（0th, 1st, 2nd, 3rd, ...）
%d	对应该月的第几天，数字（01 .. 31）（两位，前面会补0）
%e	对应该月的第几天，数字（1 .. 31）
%f	小数以下的秒（6 digits for printing: 000000 .. 999000; 1 - 9 digits for parsing: 0 .. 999999999）
%H	小时（00 .. 23）
%h	小时（01 .. 12）
%I	小时（01 .. 12）
%i	分钟，数字（00 .. 59）
%j	一年的第几天（001 .. 366）

格式化描述符	描述
%k	小时（0 .. 23）
%l	小时（1 .. 12）
%M	月份名称（January .. December）
%m	月份，数字（01 .. 12）
%p	AM or PM
%r	时间，12小时制（hh:mm:ss followed by AM or PM）
%S	秒（00 .. 59）
%s	秒（00 .. 59）
%T	时间，24小时制（hh:mm:ss）
%U	周（00 .. 53）， 星期天是一周的第一天
%u	周（00 .. 53）， 星期一是一周的第一天
%V	周（01 .. 53），星期天是一周的第一天，与%X配合使用
%v	星期（01 .. 53）， 第一条为星期一，与%X配合使用
%W	周几（Sunday .. Saturday）
%w	本周的第几天（0 .. 6），星期天是一周的第一天
%X	年份，数字，4位，第一天为星期日
%x	年份，数字，4位，第一天为星期一
%Y	年份，数字，4位
%y	年份，数字，2位，表示年份范围为[1970, 2069]
%%	表示字符'%'

示例：

```
select date_format(timestamp '2020-07-25 15:04:00.124','一年的第%j天，%m月的第%d天，%p %T %W');
                    _col0
-----------------------------------------------------
 一年的第207天，07月的第25天，PM 15:04:00 Saturday
(1 row)
```

说明

这些格式化描述符现在还不支持：%D、%U、%u、%V、%w、%X。

- date_format(timestamp, format) → varchar
  使用format格式化timestamp
- date_parse(string, format) → timestamp
  解析时间戳字符串

  ```
  select date_parse('2020/07/20', '%Y/%m/%d');-- 2020-07-20 00:00:00.000
  ```


Java 日期函数

在这一章节中使用的格式化字符串都是与Java的SimpleDateFormat样式兼容的。

- format_datetime(timestamp, format) → varchar
  使用format格式化timestamp
- parse_datetime(string, format) → timestamp with time zone
  使用指定的格式，将字符串格式化为timestamp with time zone

```
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)
```

常用提取函数

域	描述
YEAR	year()
QUARTER	quarter()
MONTH	month()
WEEK	week()
DAY	day()
DAY_OF_MONTH	day_of_month()
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()

例如：

```
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)

  描述：获取月份名称。

  ```
  SELECT monthname(timestamp '2019-09-09 12:12:12.000');-- SEPTEMBER
  SELECT monthname(date '2019-07-09');--JULY
  ```

- extract(field FROM x) → bigint

  描述：从x中返回域，对应的域字段，参照本篇的表格。

  ```
  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
  ```

函数	示例	描述
SECONDS_ADD(TIMESTAMP date, INT seconds)	SELECT seconds_add(timestamp '2019-09-09 12:12:12.000', 10);	给时间以秒为单位进行加法
SECONDS_SUB(TIMESTAMP date, INT seconds)	SELECT seconds_sub(timestamp '2019-09-09 12:12:12.000', 10);	给时间以秒为单位进行减法
MINUTES_ADD(TIMESTAMP date, INT minutes)	SELECT MINUTES_ADD(timestamp '2019-09-09 12:12:12.000', 10);	给时间以分钟为单位进行加法
MINUTES_SUB(TIMESTAMP date, INT minutes)	SELECT MINUTES_SUB(timestamp '2019-09-09 12:12:12.000', 10);	给时间以分钟为单位进行减法
HOURS_ADD(TIMESTAMP date, INT hours)	SELECT HOURS_ADD(timestamp '2019-09-09 12:12:12.000', 1);	给时间以小时为单位进行加法
HOURS_SUB(TIMESTAMP date, INT hours)	SELECT HOURS_SUB(timestamp '2019-09-09 12:12:12.000', 1);	给时间以小时为单位进行减法

- last_day(timestamp) -> date

  描述：根据指定的时间戳返回每个月的最后一天。

  ```
  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]

  描述：通过将指定的月份增加指定的日期来返回正确的日期。

  ```
  SELECT add_months(timestamp'2019-09-09 00:00:00.000', 11);-- 2020-08-09 00:00:00.000
  ```

- next_day() (timestamp, string) -> date

  描述：根据指定日期返回指定周几的下一天日期。

  ```
  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

  描述：将传递的整数值转换为day类型，例如BIGINT类型。

  ```
  SELECT numtoday(2);-- 2
  ```

1.8.14 聚合函数

聚合函数对一组值进行运算，最终获得一个单值。

除count()、count_if()、max_by()、min_by()和approx_distinct()外，其它聚合函数都忽略空值，并在没有输入行或所有值都为空时返回空值。例如sum()返回null而不是零，并且avg()在统计时不会包含null值。coalesce函数可用于将null转换为零。

聚合函数的子句

- 排序order by

  有些聚合函数可能会因为输入值的顺序不同而导致产生不同的结果，可以通过在聚合函数中使用order by子句来指定此顺序。

  ```
  array_agg(x ORDER BY y DESC);
  array_agg(x ORDER BYx,y,z);
  ```

- 过滤filter

  使用filter关键字可以在聚合的过程中，通过使用where的条件表达式来过滤掉不需要的行。所有的聚合函数都支持这个功能。

  ```
  aggregate_function(...) FILTER (WHERE <condition>)
  ```

示例：

```
--建表
create table fruit (name varchar, price int);
--插入数据
insert into fruit values ('peach',5),('apple',2);
--排序
select array_agg (name order by price) from fruit;-- [apple, peach]
--过滤
select array_agg(name) filter (where price<10) from fruit;-- [peach, apple]
```

常用聚合函数

聚合函数通常作用于数据集（表或视图）的某个具体字段，以下的参数x，均用于代指该字段。

- arbitrary(x)

  描述：返回类型和X 一样，返回X的任意一个非null值。

  ```
  select arbitrary(price) from fruit;-- 5
  ```

- array_agg(x)

  描述：返回由输入的x字段构成的数组，元素类型和输入字段一样。

  ```
  select array_agg(price) from fruit;-- [5,2]
  ```

- avg(x)

  描述：以double类型返回所有输入值的平均值。

  ```
  select avg(price) from fruit;-- 3.5
  ```

- avg(time interval type)

  描述： 返回所有输入时间间隔的平均长度，返回类型为 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  假设有日志表记录用户距离上次登录的时间，那么这个结果表明平均登录时间间隔为0天7小时15分钟30秒
  ```

- bool_and(boolean value)

  描述：当每个输入值都是true，返回true，否则返回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)

  描述：只要输入值中有为true的，返回true，否则返回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)

  描述：返回输入值的检查和，其值不受输入顺序影响，结果类型为varbinary。

  ```
  select checksum(price) from fruit; -- fb 28 f3 9a 9a fb bf 86
  ```

- count(*)

  描述：返回输入记录的条数，结果类型为bigint。

  ```
  select count(*) from fruit; -- 2
  ```

- count(x)

  描述：返回输入字段非null值的记录条数，结果类型为bigint。

  ```
  select count(name) from fruit;-- 2
  ```

- count_if(x)

  描述：类似于count(CASE WHEN x THEN 1 END)，返回输入值为true的记录数，bigint类型。

  ```
  select count_if(price>7) from fruit;-- 0
  ```

- every(boolean)

  描述：是bool_and()的一个别名。

- geometric_mean(x)

  描述：返回输入字段值的几何平均数，double类型。

  ```
  select geometric_mean(price) from fruit; -- 3.162277660168379
  ```

- listagg(x, separator) → varchar

  描述：返回由输入值连接的字符串，输入值之间由指定分隔符隔开

  语法：

  ```
  LISTAGG( expression [, separator] [ON OVERFLOW overflow_behaviour])  WITHIN GROUP (ORDER BY sort_item, [...])
  ```

  如果separator未指定， 将默认使用空字符作为分隔符。

  ```
  SELECT listagg(value, ',') WITHIN GROUP (ORDER BY value) csv_value FROM (VALUES 'a', 'c', 'b') t(value);
  csv_value
  -----------
  'a,b,c'
  ```

  当该函数的输出值超过了1048576字节时，overflow_behaviour 可以指定这种情况下的行为，默认是抛出一个Error：

  ```
  SELECT listagg(value, ',' ON OVERFLOW ERROR) WITHIN GROUP (ORDER BY value) csv_value FROM (VALUES 'a', 'b', 'c') t(value);
  ```

  也可以是当函数输出长度超出1048576字节，截断超出非空字符串，并用 TRUNCATE 指定的字符串替代，WITH COUNT和WITHOUT COUNT,表示输出结果是否包含计数：

  ```
  SELECT LISTAGG(value, ',' ON OVERFLOW TRUNCATE '.....' WITH COUNT) WITHIN GROUP (ORDER BY value)FROM (VALUES 'a', 'b', 'c') t(value);
  ```

  listagg函数也可以用于分组相关的场景，例如：

  ```
  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)

  描述：返回与所有输入值中y字段的最大值相关联的x的值。

  ```
  select max_by(name,price) from fruit; -- peach
  ```

- max_by(x, y, n)

  描述：返回按y降序排列的对应n个x值。

  ```
  select max_by(name,price,2) from fruit;-- [peach, apple]
  ```

- min_by(x,y)

  描述：返回与所有输入值中y字段的最小值相关联的x的值。

  ```
  select min_by(name,price) from fruit;-- apple
  ```

- min_by(x, y, n)

  描述：返回按y升序排列的对应n个x值。

  ```
  select min_by(name,price,2) from fruit;-- [apple, peach]
  ```

- max(x)

  描述：返回输入字段x的最大值。

  ```
  select max(price) from fruit;-- 5
  ```

- max(x, n)

  描述：返回输入字段x降序排列的前n个值。

  ```
  select max(price,2) from fruit; -- [5, 2]
  ```

- min(x)

  描述：返回输入字段x的最小值。

  ```
  select min(price) from fruit;-- 2
  ```

- min(x, n)

  描述：返回输入字段x升序排列的前n个值。

  ```
  select min(price,2) from fruit;-- [2, 5]
  ```

- sum(T, x)

  描述：对输入字段x求和，T为数值类型，如int，double，interval day to second等。

  ```
  select sum(price) from fruit;-- 7
  ```

- regr_avgx(T independent, T dependent) → double

  描述：计算回归线的自变量（ expr2 ）的平均值，去掉了空对（ expr1, expr2 ）后，等于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

  描述：计算回归线的因变量（ expr1 ）的平均值，去掉了空对（ expr1, expr2 ）后，等于AVG(expr1)。

```
select regr_avgy(time_cost,weight) from sample_collection;
 _col0
-------
  27.5
(1 row)
```

- regr_count(T independent, T dependent) → double

  描述：返回用于拟合线性回归线的非空对数。

```
select regr_count(time_cost,weight) from sample_collection;
 _col0
-------
    10
(1 row)
```

- regr_r2(T independent, T dependent) → double

  描述：返回回归的确定系数。

```
select regr_r2(time_cost,weight) from sample_collection;
      _col0
--------------------
 0.1446739237728169
(1 row)
```

- regr_sxx(T independent, T dependent) → double

  描述：返回值等于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

  描述：返回值等于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

  描述：返回值等于REGR_COUNT(expr1, expr2) * VAR_POP(expr1)。

```
select regr_syy(time_cost,weight) from sample_collection;
 _col0
--------
 2062.5
(1 row)
```

Bitwise 聚合函数

- bitwise_and_agg(x)

  描述：用补码表示输入字段x的按位与，返回类型为bigint。

```
select bitwise_and_agg(x) from (values (31),(32)) as t(x);-- 0
```


- bitwise_or_agg(x)

  描述：用补码表示输入字段x的按位或，返回类型为bigint。

  ```
  select bitwise_or_agg(x) from (values (31),(32)) as t(x);-- 63
  ```

Map 聚合函数

- histogram(x) -> map(K, bigint)

  描述：返回一个map，包含了所有输入字段x出现的次数。

  ```
  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)

  描述：返回一个由输入字段key和输入字段value为键值对的map。

  ```
  select map_agg(name,price) from fruit;-- {apple=2, peach=5}
  ```

- map_union(x(K, V)) -> map(K, V)

  描述：返回所有输入map的并集。如果一个key值在输入集中出现多次，对应的value取输入集中的key对应的任意值。

  ```
  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))

  描述：返回一个由输入key、value键值对组成的多重映射map。每个key可以对应多个value。

  ```
  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]}
  ```

近似值聚合函数

在实际情况下，对大量数据进行统计时，有时只关心一个近似值，而非具体值，比如统计某产品的销量，这种时候，近似值聚合函数就很有用，它使用较少的内存和CPU资源，以便可以获取数据结果而不会出现任何问题，例如溢出到磁盘或CPU峰值。这对于数十亿行数据运算的需求很有用。

- approx_median(x) → bigint

  描述：该函数返回一个值，该值近似为输入值集的中位数。

  ```
  select approx_median(price) from fruit; -- 10.0
  ```

- approx_distinct(x) → bigint

  描述：该函数返回类型为bigint，它提供了count(distinct x)的近似计数。如果所有输入都是null值，则返回0。

  此函数所有可能的值相对于正确的值的误差服从近似正态分布，其标准差为2.3%。它不保证任何特定输入集误差的上限。

  ```
  select approx_distinct(price) from fruit; -- 2
  ```

- approx_distinct(x, e)→ bigint

  描述：该函数返回类型为bigint，它提供了count(distinct x)的近似计数。如果所有输入都是null值，则返回0。

  此函数所有可能的值相对于正确的值的误差服从近似正态分布，其标准差应小于e。它不保证任何特定输入集的误差的上限。

  当前该函数的实现中，e的取值范围为[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>

  描述：近似统计出前buckets个最频繁出现的元素。函数统计高频值时，采用近似估算的方式使用的内存更少。capacity值越大，结果越精确，但消耗的内存也更多。该函数的返回结果是一个map，map的键值对为高频值及对应的频次。

  ```
  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}
  ```

  说明

  分位数，常用的有二分位数，四分位数，十分位数，百分位数等，意味将输入集合均分为对应等份，然后找到大约位于该位置的数值。比如approx_percentile(x, 0.5)就是找到大约位于x值排序后大约50%位置的值，也就是二分位数。

- approx_percentile(x, percentage) → [same as x]

  描述：根据给定的百分比，返回对应的近似百分位数。这个百分比的值对于所有输入的行来说必须是0到1之间的一个常量。

  ```
  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]>

  描述：以给定的百分比数组中的每个百分比，返回所有输入字段x值的近似百分位数。这个百分比数组中的每个值对于所有输入的行来说必须是0到1之间的一个常量。

  ```
  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]>

  描述：按照百分比percentage，返回所有x输入值的近似百分位数。每一项的权重值为w且必须为正数。x设置有效的百分位。percentage的值必须在0到1之间，并且所有输入行必须为常量。

  ```
  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]

  描述：按照百分比percentage，返回所有x输入值的近似百分位数。每一项的权重值为w且必须为正数。x设置有效的百分位。percentage的值必须在0到1之间，并且所有输入行必须为常量。其中，近似值的最大进度误差由accuracy指定。

  ```
  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]

  描述：按照百分比数组中的每个百分比值，返回所有 x 输入值的近似百分位数。每一项的权重值为w且必须为正数。x设置有效的百分位。百分比数组中每个元素值必须在0到1之间，并且所有输入行必须为常量。

  ```
  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]
  ```

  说明

  以上approx_percentile函数也支持同参数集的percentile_approx函数。

- numeric_histogram(buckets, value, weight)

  描述：按照buckets桶的数量，为所有的value计算近似直方图，每一项的宽度使用weight。本算法大体上基于。

  ```
  Yael Ben-Haim and Elad Tom-Tov, "A streaming parallel decision tree algorithm", J. Machine Learning Research 11 (2010), pp. 849--872.
  ```

  buckets必须是bigint。value和weight必须是数值类型。


```
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)

  描述：与numeric_histogram(buckets, value,weight)相比，相当于将weight设为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)
```

统计聚合函数

- corr(y,x)

  描述：返回输入值的相关系数。

```
select corr(y,x) from (values (1,5),(2,6),(3,7),(4,8)) as t(x,y);-- 1.0
```

- covar_pop(y, x)

  描述：返回输入值的总体协方差。

```
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)*

  描述：返回输入值的样本协方差。

```
select covar_samp(y,x) from (values (1,5),(2,6),(3,7),(4,8)) as t(x,y);-- 1.6666666
```

- kurtosis(*x*)

  描述：峰度又称峰态系数，表征概率密度分布曲线在平均值处峰值高低的特征数，即是描述总体中所有取值分布形态陡缓程度的统计量。直观看来，峰度反映了峰部的尖度。这个统计量需要与正态分布相比较。

  定义上峰度是样本的标准四阶中心矩（standardized 4th central moment)。

  随机变量的峰度计算方法为随机变量的四阶中心矩与方差平方的比值。

  具体计算公式为：

  $$\text{Kurtosis} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^4 / \text{SD}^4 - 3$$

```
select kurtosis(x) from (values (1),(2),(3),(4)) as t(x); -- -1.1999999999999993
```

- regr_intercept(y, x)

  描述：返回输入值的线性回归截距。y是从属值。x是独立值。

```
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)

  描述：返回输入值的线性回归斜率。y是从属值。x是独立值。

```
select regr_slope(y,x) from (values (1,5),(2,6),(3,7),(4,8)) as t(x,y);--  1.0
```

- skewness(x)

  描述：返回所有输入值的偏斜度。

```
select skewness(x) from (values (1),(2),(3),(4)) as t(x); -- 0.0
```

- stddev(x)

  描述：stedev_samp()的别名。

- stddev_pop(x)
  描述：返回所有输入值的总体标准差。
  ```
  select stddev_pop(x) from (values (1),(2),(3),(4)) as t(x);--  1.118033988749895
  ```
- stddev_samp(x)
  描述：返回所有输入值的样本标准偏差。
  ```
  select stddev_samp(x) from (values (1),(2),(3),(4)) as t(x);--  1.2909944487358056
  ```
- variance(x)
  描述：var_samp()的别名。
- var_pop(x)
  描述：返回所有输入值的总体方差。
  ```
  select var_pop(x) from (values (1),(2),(3),(4)) as t(x);-- 1.25
  ```
- var_samp(x)
  描述：返回所有输入值的样本方差。
  ```
  select var_samp(x) from (values (1),(2),(3),(4)) as t(x);--  1.6666666666666667
  ```

Lambda 聚合函数

reduce_agg(*inputValue T*, *initialState S*, *inputFunction(S*, *T*, *S)*, *combineFunction(S*, *S*, *S)*)

每个非空输入值将调用inputFunction。除了获取输入值之外，inputFunction还获取当前状态，最初为initialState，然后返回新状态。将调用CombineFunction将两个状态合并为一个新状态。 返回最终状态。

```
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)
```

说明

状态值必须是 boolean、integer、 floating-point或date、time、interval。

1.8.15 窗口函数

窗口函数跨查询结果的行执行计算。它们在HAVING子句之后但在ORDER BY子句之前运行。调用窗口函数需要使用OVER子句来指定窗口的特殊语法。窗口具有三个组成部分：

- 分区规范，它将输入行分为不同的分区。这类似于GROUP BY子句如何将行分为聚合函数的不同组。
- 排序规范，它确定窗口函数将处理输入行的顺序。
- 窗口框架，指定给定行该功能要处理的行的滑动窗口。如果未指定帧，则默认为“RANGE UNBOUNDED PRECEDING”，与“UNBOUNDEEN PREBODING AND CURRENT ROWGE”相同。该帧包含从分区的开始到当前行的最后一个对等方的所有行。在没有ORDER BY的情况下，所有行都被视为对等行，因此未绑定的前导和当前行之间的范围等于未绑定的前导和未绑定的后续之间的范围。

  例如：下面的查询将salary表中的信息按照每个部门员工工资的大小进行排序。

```
 --创建数据表并插入数据
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);

--数据查询
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

所有的聚合函数都能通过添加over子句来当做窗口函数使用。聚合函数将在当前窗口框架下的每行记录进行运算。

下面的查询生成每个职员按天计算的订单价格的滚动总和。

```
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

  描述：小于等于当前值的行数/分组内总行数–比如，统计小于等于当前薪水的人数，所占总人数的比例。

```
 --查询示例
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

  描述：返回值在一组值中的排名。这与rank（ ）相似，不同的是tie值不会在序列中产生间隙。

- ntile(*n*)→ bigint

  描述：用于将分组数据按照顺序切分成n片，返回当前切片值。NTILE不支持ROWS BETWEEN，比如NTILE(2) OVER(PARTITION BY cookieid ORDER BY createtime ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)如果切片不均匀，默认增加第一个切片的分布。

```
--创建表并插入数据
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);
-- 查询结果
SELECT cookieid,createtime,pv,
NTILE(2) OVER(PARTITION BY cookieid ORDER BY createtime) AS rn1,    --分组内将数据分成2片
NTILE(3) OVER(PARTITION BY cookieid ORDER BY createtime) AS rn2,  --分组内将数据分成3片
NTILE(4) OVER(ORDER BY createtime) AS rn3   --将所有数据分成4片
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

  描述：返回值在一组值中的百分比排名。 结果为（ r-1 ）/（ n-1 ），其中r是该行的rank（ ），n是窗口分区中的总行数。

```
SELECT dept,userid,sal,
PERCENT_RANK() OVER(ORDER BY sal) AS rn1,   --分组内
RANK() OVER(ORDER BY sal) AS rn11,          --分组内RANK值
SUM(1) OVER(PARTITION BY NULL) AS rn12,     --分组内总行数
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

  描述：返回值在一组值中的排名。等级为1加上该行之前与该行不对等的行数。因此，排序中的平局值将在序列中产生缺口。对每个窗口分区执行排名。

```
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

  描述：从1开始，按照顺序，生成分组内记录的序列–比如，按照pv降序排列，生成分组内每天的pv名次ROW_NUMBER() 的应用场景非常多，再比如，获取分组内排序第一的记录。获取一个session中的第一条refer等。

```
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

通常情况下，要重视null值。如果指定了IGNORE NULLS，那么计算中所有包含x为null值的行都会被排除掉，如果所有行的x字段值都是null值，将会返回默认值，否则返回null值。

```
-- 数据准备
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');
```

- first_value(*x*) → [same as input]

  描述：返回窗口的第一个值。

```
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 | urll3 |  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]

  描述：返回窗口的最后一个值。

```
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 | urll3 |  3 | urll3
 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]

描述：返回距窗口开头指定偏移量的值。偏移量从1开始。偏移量可以是任何标量表达式。如果偏移量为null或大于窗口中的值数，则返回null。偏移量不允许为0或者负数。

```
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]

  描述：返回窗口分区中当前行之后的偏移行处的值。偏移量从0开始，即当前行。偏移量可以是任何标量表达式。默认偏移量为1。如果偏移量为null，则返回null。如果偏移量指向不在分区内的行，则返回default_value，或者如果未指定，则返回null。lead（ ）函数要求指定窗口顺序。不得指定窗框。

```
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]

  描述：返回窗口分区中当前行之前的偏移行的值，偏移量从0开始，即当前行，偏移量可以是任何标量表达式，默认偏移量为1。如果偏移量为null，则返回null。如果偏移量指向不在分区内的行，则返回default_value。如果未指定，则返回null。lag（ ）函数要求指定窗口顺序，不得指定窗框。

```
 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 数组函数和运算符

下标操作符：[]

描述：下标操作符用于访问数组中的元素，并从1开始建立索引。

```
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 : ||

|| 操作符用于将相同类型的数组或数值串联起来。

```
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 函数

下标运算符: []

下标运算符 [] 用于获取数组中对应位置的值。

```
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

  描述：如果数组x中包含element，则返回true。

  ```
  select array_contains (array[1,2,3,34,4],4); -- true
  ```

- all_match(array(T), function(T, boolean)) → boolean

  描述：返回是否数组的所有元素满足给定的断言函数。如果都满足断言函数或者数组为空时，返回true, 如果有一个或者多个元素不满足断言函数，则返回false。当断言函数对于一个或者多个元素的结果是NULL时，返回结果也是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

  描述：返回数组是否存在满足断言的元素。当一个或多个元素满足断言时，返回true。都不满足断言或者数组为空时，返回false。当断言函数对于一个或者多个元素的结果是NULL时，返回结果也是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

  描述：输出去重后的数组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

  描述：返回两个数组去重后的交集。

  ```
  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

  描述：返回两个数组的并集。

  ```
  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

  描述：返回去重后的在x中但不在y中的元素数组。

  ```
  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

  描述：使用分隔符来连接给定数组x的元素，并用可选字符替换x中的null值。

  ```
  select array_join(array[1,2,3,null,5,6],'|','0');--  1|2|3|0|5|6
  ```

- array_max(*x*) → x

  描述：返回数组x的最大值。


```
select array_max(array[2,54,67,132,45]);  -- 132
```

- array_min(*x*) → x

  描述：返回数组x的最小值。

```
select array_min(array[2,54,67,132,45]);  -- 2
```

- array_position(x,element) → bigint

  描述：返回数组x中element第一次出现的位置，没找到则返回0。

```
select array_position(array[2,3,4,5,1,2,3],3); -- 2
```

- array_remove(*x*, *element*) → array

  描述：移除数组x中的值为element的元素并返回。

```
select array_remove(array[2,3,4,5,1,2,3],3); --  [2, 4, 5, 1, 2]
```

- array_sort(*x*) → array

  排序并返回数组x。 x的元素必须是可排序的。 空元素将放置在返回数组的末尾。

```
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))*

  描述：根据给定的比较器函数对数组进行排序并返回。比较器将使用两个可为空的参数，表示数组的两个可为空的元素。当第一个可为空的元素小于，等于或大于第二个可为空的元素时，它将返回-1、0或1。如果比较器函数返回其他值（包括NULL），则查询将失败并引发错误。

```
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)

 -- null值排在前，其余值按降序排列
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)

-- null值在后的降序排列
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)

-- 按字符串长度排序
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)

-- 按数组长度排序
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*)

  描述：如果两个数组有共同的非null元素，则返回true。否则返回false。

  ```
  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*)

  描述:返回数组x的容量。

  ```
  select cardinality(array[1,2,3,4,5,6]); --6
  ```

- concat(*array1*, *array2*, ..., *arrayN*)

  描述：此函数提供与sql标准连接运算符(||)相同的功能。

- combinations(*array(T)*, *n) -> array(array(T)*)

  描述：返回输入数组的n个元素子组。 如果输入数组没有重复项，则组合将返回n个元素的子集。

  ```
  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]]
  ```

  子组以及子组中的元素，虽未指明，都是有序的。参数n必须不大于5，且产生的子组个数最大不超过100000。

- contains(*x*, *element*)

  描述：如果数组x中包含element，则返回true。

  ```
  select contains(array[1,2,3,34,4],4); -- true
  ```

- element_at(*array(E)*, *index*)

  描述：返回给定索引处数组的元素。 如果index> 0，则此函数提供与SQL标准下标运算符（ [] ）相同的功能，但在访问大于数组长度的索引时该函数返回NULL，且下标运算符在这种情况下将失败。 如果index <0，则element_at从最后到第一个访问元素。

  ```
  select element_at(array['a','b','c','d','e'],3); -- c
  ```

- filter(*array(T)*, *function(T*, *boolean)) -> array(T*)

  描述：删选出按函数运算结果为true的元素构成的数组。

  ```
  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*)

  描述：以串联的方式将array(array(T)) 展开为array(T)。

- ngrams(*array(T)*, *n) -> array(array(T)*)

描述：返回数组的n元语法（相邻n个元素的子序列）。结果中n元语法的顺序未指定。

```
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)*)

  描述：返回数组是否没有元素与给定谓词匹配。如果没有元素与谓词匹配，则返回true（特殊情况是当数组为空时）。如果一个或多个元素匹配，则为false；如果谓词函数对一个或多个元素返回NULL，而对所有其他元素返回false，则为NULL。

- reduce(*array(T)*, *initialState S*, *inputFunction(S*, *T*, *S)*, *outputFunction(S*, *R)*)

  返回从数组减少的单个值。将按顺序为数组中的每个元素调用inputFunction。除了获取元素之外，inputFunction还获取当前状态，最初为initialState，然后返回新状态。将调用outputFunction将最终状态转换为结果值。它可能是恒等函数（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*)

  描述：将element重复输出count次，填充到数组中。

```
select repeat(4,5);-- [4, 4, 4, 4, 4]
```

- reverse(*x*)

  描述：以相反顺序将数组元素填充到返回的新数组中。

```
select reverse(array[1,2,3,4,5]); --[5, 4, 3, 2, 1]
```

- sequence(start, *stop)*

  描述：输出一个从start开始，到*stop*结束的数组。start不大于*stop*时，每次递增1，否则，每次递减1。

  start和stop的数据类型还可以是date或者timstamp类型，按1天递增或递减。

```
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

  描述：以步长step从start输出到stop。

```
select sequence(1,30,5);--  [1, 6, 11, 16, 21, 26]
```


- shuffle(*x*) → array

  描述：根据给的数组随机排列获得一个新的数组。

  ```
  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

  描述：返回arrayx的容量。

  ```
  select size(array[1,2,3,4,5,6]); --6
  ```

- slice(*x*, *start*, *length*) → array

  描述：子集数组x从索引开头开始（如果start为负，则从结尾开始），长度为length。

  ```
  select slice(array[1,2,3,4,5,6],2,3);-- [2, 3, 4]
  ```

- sort_array(x)

  参考array_sort(x)。

- trim_array(*x*, *n*) → array

  描述：移除数组末尾n个元素。

  ```
  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*)

  描述：返回一个数组，该数组是将函数应用于数组的每个元素的结果。

  ```
  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*)

  描述：将给定数组按元素合并到单个行数组中。第N个自变量的第M个元素将是第M个输出元素的第N个字段。如果参数长度不均匀，则缺少的值将填充为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*)

  描述：使用函数将两个给定的数组逐个元素合并到单个数组中。如果一个数组较短，则在应用函数之前，将在末尾添加空值以匹配较长数组的长度。

  ```
  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 函数和运算符

下表操作符: []

描述：[]运算符用于从映射中检索与给定键对应的值。

```
select age_map['li'] from (values (map(array['li','wang'],array[15,27]))) as table_age(age_map);-- 15
```

Map 函数

- cardinality(*x*)

描述：返回map x的基数大小。

```
select cardinality(map(array['num1','num2'],array[11,12]));-- 2
```

- element_at(*map(K, V)*, *key*)

  描述：返回map中key对应值，如果map中不包含这个key，则返回NULL。

  ```
  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()

  描述：返回一个空的map。

  ```
  select map();-- {}
  ```

- map(*array(K), array(V)) -> map(K, V)*

  描述：根据给定的键值对数组，返回map。聚合函数中的map_agg()和multimap_agg()也同样能用于生成map。

  ```
  SELECT map(ARRAY[1,3],ARRAY[2,4]);-- {1=2, 3=4}
  ```

- map_from_entries(*array(row(K, V))) -> map(K, V)*

  描述：使用给定数组生成map。

  ```
  SELECT map_from_entries(ARRAY[(1, 'x'), (2, 'y')]); --  {1=x, 2=y}
  ```

- multimap_from_entries(*array(row(K, V))) -> map(K, array(V)*)

  描述：根据给定的row数组返回复合map，每个键可以对应多个值。

  ```
  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)*)

  描述：使用给定map生成一个row数组。

  ```
  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))*

  描述：合并多个map，当key值一样时，取最后一个map的value来构造键值对。如下示例中，a就使用了最后一个map的value值10。

  ```
  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)*

  描述：使用map中仅给定函数映射为true的entry去构造一个新的map。

  ```
  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*)

  描述：返回map中所有的key构造的数组。

  ```
  select map_keys(map(array['num1','num2'],array[11,12])); -- [num1, num2]
  ```

- map_values(*x(K, V)) -> array(V*)

  描述：返回map中所有的value构造的数组。

  ```
  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))*

  描述：通过将函数应用于具有相同键的一对值，将两个给定的map合并为一个map。对于仅在一个map中显示的键，将传递NULL作为缺少键的值。

  ```
  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)*

  描述：对map中的每个entry，将key值K1映射为新的key值K2，保持对应的value不变。

```
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

  描述：返回Map(x) 的容量。

```
select size(map(array['num1','num2'],array[11,12])); --2
```

- transform_values(*map(K, V1)*, *function(K, V2, V2)) -> map(K, V2)*

  描述：对map中的每个entry，将value值V1映射为新的value值V2，保持对应的key不变。

```
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 函数

提取函数

描述：提取函数用于从HTTP URL（或任何符合RFC 2396标准的URL）中提取内容。

```
[protocol:][//host[:port]][path][?query][#fragment]
```

提取的内容不会包含URI的语法分割符，比如“：”或“？”。

- url_extract_fragment(*url*) → varchar

  描述：返回url的片段标识符，即#后面的字符串。

  ```
  select url_extract_fragment('http://www.example.com:80/stu/index.html?
  name=xxx&age=25#teacher');--teacher
  ```

- url_extract_host(url) → varchar

  描述：返回url中的主机域名。

  ```
  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

  描述：返回url中参数名为name的参数。

  ```
  select url_extract_parameter('http://www.example.com:80/stu/index.html?
  name=xxx&age=25#teacher','age');-- 25
  ```

- url_extract_path(*url*) → varchar

  描述：提取url中的路径。

  ```
  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

  描述：提取url中的端口。

  ```
  select url_extract_port('http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher');-- 80
  ```

- url_extract_protocol(*url*) → varchar

  描述：提取url中的协议。

  ```
  select url_extract_protocol('http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher');
  --  http
  ```

- url_extract_query(*url*) → varchar

  描述：提取url中的查询字符串。

  ```
  select url_extract_query('http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher'); --
  name=xxx&age=25
  ```

编码函数

- url_encode(*value*) → varchar

  描述：对value进行转义处理，以便可以安全地将其包含在URL查询参数名和值中：

  - 字母字符不会被编码。
  - 字符 ., -, * 和 _不会被编码。
  - ASCII 空格字符会被编码为+ 。
  - 所有其他字符都将转换为UTF-8，并且字节被编码为字符串%XX，其中XX是UTF-8字节的大写十六进制值。

  ```
  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

  描述：对value编码后的URL进行解码操作。

  ```
  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 函数

UUID函数用于生成一个伪随机的唯一通用标识符（Universally Unique Identifier，UUID）。

UUID 是一种 128 位的标识符，通常以 32 个十六进制字符表示，分为 5 组，格式为 8-4-4-4-12。

```
select uuid();
```

返回示例：

```
6b4c7d5a-2f3e-4a19-8e9f-5d2e8e9f4a1b
```

1.8.20 Color 函数

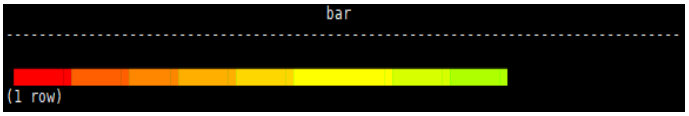
- bar(*x*, *width*)

  描述：使用默认的低频红色和高频绿色渲染ANSI条形图中的单个条形。例如，如果将25%的x和40的宽度传递给此函数。将绘制一个10个字符的红色条形，后跟30个空格，以创建一个40个字符的条形。

- bar(x, width, low_color, high_color)

  描述：在ANSI条形图中以指定宽度绘制一条直线。参数x是0到1之间的一个双精度值。x的值超出[0，1]范围将被截断为0或1值。low_color和high_color捕获用于水平条形图任一端的颜色。例如，如果x为0.5，宽度为80，low_color为0xFF0000，high_color为0x00FF00，则此函数将返回一个40个字符的条形，该条形由红色（0xFF0000）和黄色（0xFFFF00）组成，其余80个字符条为用空格填充。

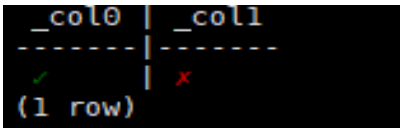
  ```
  select bar(0.75,80,rgb(255,0,0),rgb(0,255,0));
  ```

- render(b)

  描述：根据布尔值返回对错符号。

  ```
  select render(true),render(false);
  ```

1.8.21 Teradata 函数

以下函数提供Teradata SQL的能力。

字符串函数

- char2hexint(*string*)

  描述：返回字符串的UTF-16BE编码的十六进制表示形式。

- index(*string*, *substring*)

  描述：同strpos() 函数。

日期函数

本节中的函数使用与Teradata datetime函数兼容的格式字符串。下表基于Teradata参考手册，描述了受支持的格式说明符。

说明符	说明
- / , . ; :	忽略标点符号
dd	一个月中的第几日（1-31）
hh	一天中的第几个小时（1-12）
hh24	一天中的第几个小时（0-23）
mi	分钟（0-59）
mm	月份（01-12）
ss	秒（0-59）
yyyy	四位年份
yy	两位年份

说明

当前不支持不区分大小写。 所有说明符必须小写。

- to_char(*timestamp*, *format*)

  描述：将时间戳按指定格式输出为字符串。

  ```
  select to_char(timestamp '2020-12-18 15:20:05','yyyy/mmdd hh24:mi:ss');-- 2020/1218 15:20:05
  ```

- to_timestamp(*string*, *format*)

  描述：将字符串按规定格式解析为timestamp。

  ```
  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*)

  描述：将字符串按格式转换为日期。

  ```
  select to_date('2020/12/04','yyyy/mm/dd'); -- 2020-12-04
  ```

1.8.22 Data masking 函数

数据脱敏(Data masking) 指对某些敏感信息通过脱敏规则进行数据的变形，实现敏感隐私数据的可靠保护。

- mask_first_n(string str[, int n]) →varchar

  描述：返回str的屏蔽版本，前n个值被屏蔽。大写字母被转为 " X " ，小写字母被转为 " x " ，数字被转为 " 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

描述：返回str的屏蔽版本，后n个值被屏蔽。大写字母被转为 " X " ，小写字母被转为 " x " ，数字被转为 " 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

  描述：返回str的屏蔽版本，只显示前n个字符。大写字母被转为 " X " ，小写字母被转为 " x " ，数字被转为 " 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

  描述：返回str的屏蔽版本，只显示后n个值。大写字母被转为 " X " ，小写字母被转为 " x " ，数字被转为 " 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

  描述：返回基于str的散列值。散列是一致的，可以用于跨表连接被屏蔽的值。对于非字符串类型，返回NULL。

```
select mask_hash('panda');
                            _col0
------------------------------------------------------------------
 a7cdf5d0586b392473dd0cd08c9ba833240006a8a7310bf9bc8bf1aefdfaeadb
(1 row)
```

1.8.23 IP Address 函数

contains(network, address) → boolean

当CIDR网络中包含address时返回true。

```
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 函数

概述

Quantile digest（分位数摘要）是存储近似百分位信息的数据草图。HetuEngine中用qdigest表示这种数据结构。

函数

- merge(qdigest) → qdigest

  描述：将所有输入的qdigest数据合并成一个qdigest。

- value_at_quantile(qdigest(T), quantile) → T

描述：给定0到1之间的数字分位数，返回分位数摘要中的近似百分位值。

- values_at_quantiles(qdigest(T), quantiles) -> array(T)

  描述：给定一组0到1之间的数字分位数，从分位数摘要中返回对应的近似百分位值组成的数组。

- qdigest_agg(x) -> qdigest([same as x])

  描述：返回由x的所有输入值组成的qdigest。

- qdigest_agg(x, w) -> qdigest([same as x])

  描述：返回由x的所有输入值（使用每项权重w）组成的qdigest。

- qdigest_agg(x, w, accuracy) -> qdigest([same as x])

  描述：返回由x的所有输入值（使用每项权重w和最大误差accuracy）组成的qdigest。accuracy必须是一个大于0且小于1的值，并且对于所有输入行是一个常量。

1.8.25 T-Digest 函数

概述

T-digest是存储近似百分位信息的数据草图。HetuEngine中用tdigest表示这种数据结构。T-digest可以合并，在存储时可以强转为VARBINARY，检索时再由VARBINARY转换为T-digest

函数

- merge(tdigest)→tdigest

  描述：将所有输入的tdigest数据合并成一个tdigest。

- value_at_quantile(tdigest,quantile)→double

  描述：给定0到1之间的数字分位数，返回T-digest中的近似百分位值。

- values_at_quantiles(tdigest,quantiles)->array(double)

  描述：给定一组0到1之间的数字分位数，从T-digest中返回对应的分位数组成的数组。

- tdigest_agg(x)->tdigest

  描述：返回由x的所有输入值组成的tdigest。x可以是任何数值类型。

- tdigest_agg(x,w)->tdigest

  描述：返回由x的所有输入值（使用每项权重w）组成的tdigest。w必须大于或等于1。x和w可以是任何数值类型。

2 数据类型隐式转换

2.1 简介

数据类型隐式转换是指在用户通过客户端访问HetuEngine资源时，当查询中使用的数据类型与表中定义的数据类型不匹配时，HetuEngine会自动进行数据类型转换。这一功能减少了因强数据类型校验导致的错误，提升了用户体验。

目前，HetuEngine支持在以下场景中进行数据类型隐式转换：

- 插入数据（Insert）：在插入数据时，自动将输入数据转换为目标表的对应数据类型。
- 条件判断（Where）：在条件表达式中，自动将不同类型的数据转换为可比较的类型。
- 运算操作（+、-、*、/）：在进行算术运算时，自动将操作数转换为兼容的类型。
- 函数调用（连接操作 ||）：在调用函数或进行字符串连接时，自动将不同数据类型转换为函数或操作符所需的类型。

通过这些隐式转换功能，用户无需手动处理数据类型不匹配的问题，从而简化了操作流程并提高了查询效率。

2.2 隐式转换对照表

当数据类型不匹配时会隐式转换，但并是不是所有的数据类型都支持隐式转换。以下为当前隐式转换功能支持的数据类型转换表：

表 2-1 隐式转换对照表

-	BOOLEAN	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	DOUBLE	DECIMAL	VARCHAR
BOOLEAN	\	Y(1)	Y	Y	Y	Y	Y	Y	Y(2)

-	BOOLEAN	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	DOUBLE	DECIMAL	VARCHAR
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
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

表 2-2 隐式转换对照表（续）

-	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
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	\

说明

- BOOLEAN->NUMBER 结果只会是0/1。
- BOOLEAN->VARCHAR 字符结果只会是‘TRUE’/‘FALSE’。
- NUMBER -> BOOLEAN 0就是false，非0就是true。
- BIG PRECISION -> SMALL不能大于目标类型的取值范围，否则会报错。
- REAL/FLOAT ->DECIMAL目标类型的整数位必须大于或等于REAL/FLOAT整数位，否则转换报错，小数位不足会截断。
- DECIMAL->DECIMAL目标类型整数位的范围必须大于等于源类型，否则转换失败，小数位不足会截断。
- VARCHAR->BOOLEAN字符只有 '0'，'1'，'TRUE'，'FALSE' 可转换。
- VARCHAR->DECIMAL如果小数位大于目标decimal的小数位，则会发生截断，如果整数位超过目标decimal的范围则报错。
- VARCHAR->CHAR 如果VARCHAR长度超过目标长度，则会截断。
- VARCHAR->DATE仅支持按照“-”分割的日期，例如2000-01-01。
- VARCHAR->TIME仅支持严格的日期格式：HH:MM:SS.XXX。
- VARCHAR->TIME ZONE仅支持严格的格式：例如01:02:03.456 America/Los_Angeles。
- VARCHAR->TIMESTAMP仅支持严格的格式YYYY-MM-DD HH:MM:SS.XXX。
- DATE->TIMESTAMP自动补齐时间，补零 '2010-01-01' -> 2010-01-01 00:00:00.000。
- TIME->TIME WITH TIME ZONE自动补齐时区。
- TIME->TIMESTAMP自动补齐日期，取默认值1970-01-01。

3 附录

3.1 本文样例表数据准备

```
--创建具有TINYINT类型数据的表。
 CREATE TABLE int_type_t1  (IT_COL1 TINYINT) ;
--插入TINYINT类型数据
 insert into int_type_t1  values (TINYINT'10');
--创建具有DECIMAL类型数据的表。
CREATE TABLE decimal_t1        (dec_col1 DECIMAL(10,3)) ;
--插入具有DECIMAL类型数据
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));
```

```
--创建Map类型表
create table map_tb(col1 MAP<STRING,INT>);
--插入一条Map类型数据
insert into map_tb values(MAP(ARRAY['foo','bar'],ARRAY[1,2]));
--查询数据
select * from map_tb; --   {bar=2, foo=1}
```

```
--创建ROW表
create table row_tb (id int,col1 row(a int,b varchar));
--插入ROW类型数据
insert into row_tb values (1,ROW(1,'SSS'));
--查询数据
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
```

```
-- 创建struct 表
create table struct_tab (id int,col1 struct<col2: integer, col3: string>);
--插入 struct 类型数据
 insert into struct_tab VALUES(1, struct<2, 'test'>);
--查询数据
select * from struct_tab; --
 id |         col1
----|---------------------
  1 | {col2=2, col3=test}
```

```
--创建一个名为web的schema：
CREATE SCHEMA web;
```



```
--在hive 数据源下创建一个名为sales的schema：
CREATE SCHEMA hive.sales;
--创建一个名为traffic，如果不存在的话：
CREATE SCHEMA IF NOT EXISTS traffic;
```

```
--创建一个新表orders，使用子句with指定创建表的存储格式、存储位置、以及是否为外表:
CREATE TABLE orders (
orderkey bigint,
orderstatus varchar,
totalprice double,
orderdate date
)
WITH (format = 'ORC', location='/user',external=true);
--如果表orders不存在，则创建表orders，并且增加表注释和列注释：
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.';
--使用表orders的列定义创建表bigger_orders：
CREATE TABLE bigger_orders (
another_orderkey bigint,
LIKE orders,
another_orderdate date
);
```

```
CREATE SCHEMA hive.web WITH (location = 'obs://bucket/user');
--创建分区表
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
);
--插入数据
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' );
```

```
-- 删除分区表数据（删除where子句指定的分区所有数据）
delete from hive.web.page_views where ds=date '2020-07-17' and country='US';
```

```
--用指定列的查询结果创建新表orders_column_aliased：
CREATE TABLE orders_column_aliased (order_date, total_price)
AS
SELECT orderdate, totalprice FROM orders;
--用表orders的汇总结果新建一个表orders_by_data：
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;
--如果表orders_by_date不存在，则创建表orders_by_date：
```



```
CREATE TABLE IF NOT EXISTS orders_by_date AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate;
--用和表orders具有相同schema创建新表empty_orders table，但是没数据：
CREATE TABLE empty_orders AS
SELECT *
FROM orders
WITH NO DATA;
```

```
--通过表orders创建一个视图test_view:
CREATE VIEW test_view (oderkey comment 'orderId',orderstatus comment 'status',half comment 'half') AS
SELECT orderkey, orderstatus, totalprice / 2 AS half FROM orders;
--通过表orders的汇总结果创建视图orders_by_date_view:
CREATE VIEW orders_by_date_view AS
SELECT orderdate, sum(totalprice) AS price
FROM orders
GROUP BY orderdate;
--创建一个新视图来替换已经存在的视图:
CREATE OR REPLACE VIEW test_view AS
SELECT orderkey, orderstatus, totalprice / 4 AS quarter
FROM orders;
```

```
--更改已存在表的定义。
--数据准备
create table users (id int,name varchar);
--将表名从users 修改为 people：
ALTER TABLE users RENAME TO people;
--在表people中增加名为zip的列：
ALTER TABLE people ADD COLUMN zip varchar;
--从表people中删除名为zip的列：
ALTER TABLE people DROP COLUMN zip;
--将表people中列名id更改为user_id：
ALTER TABLE people RENAME COLUMN id TO user_id;
```

```
create table testfordrop(name varchar);
```

```
--创建视图
create view orders_by_date as select * from orders;
--设置表的注释信息，可以通过设置注释信息为NULL来删除注释
COMMENT ON TABLE people IS 'master table';
```

```
--创建一个具有列名id、name的新表：
CREATE TABLE example AS
SELECT * FROM (
VALUES
(1, 'a'),
(2, 'b'),
(3, 'c')
) AS t (id, name);
```

```
--创建fruit 和 fruit_copy表
create table fruit (name varchar,price double);
create table fruit_copy (name varchar,price double);
--向 fruit 表中插入一行数据
insert into fruit values('LIchee',32);
--向fruit 表中插入多行数据
insert into fruit values('banana',10),('peach',6),('lemon',12),('apple',7);
--将fruit表中的数据行加载到fruit_copy 表中，执行后表中有5条记录
insert into fruit_copy select * from fruit;
--先清空fruit_copy表，再将fruit 中的数据加载到表中，执行之后表中有2条记录。
insert overwrite fruit_copy select *  from fruit limit 2;
```

```
--创建一个航运表
create table shipping(origin_state varchar(25),origin_zip integer,destination_state
varchar(25) ,destination_zip integer,package_weight integer);
```



```
--插入数据
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);
```

```
--创建表并插入数据
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);
```

```
-- 创建表
create  table new_shipping (origin_state varchar,origin_zip varchar,packages int ,total_cost int);
```

```
-- 插入数据
insert into new_shipping
values
('California','94131',25,100),
('California','P332a',5,72),
('California','94025',0,155),
('New Jersey','08544',225,490);
```

```
--创建数据表并插入数据
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);
```

```
-- 数据准备
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 常用数据源语法兼容性

语法	Hive	Hudi
数据库的show schemas	Y	Y
数据库的create schema	Y	Y
数据库的alter schema	Y	N
数据库的drop schema	Y	Y
表的show tables/show create table/show functions/show session	Y	Y
表的create	Y	N
表的create table TABLENAME as	Y	N
表的insert into TABLENAME values	Y	N
表的insert into TABLENAME select	Y	N
表的insert overwrite TABLENAME values	Y	N
表的insert overwrite TABLENAME select	Y	N
表的alter	Y	N
表的select	Y	Y
表的delete	Y	N
表的drop	Y	N
表的desc/describe TABLENAME	Y	Y
表的comment	Y	N

语法	Hive	Hudi
表的explain	Y	Y
表的show columns	Y	Y
表的select column	Y	Y
视图的create view	Y	N
视图的create or replace view	Y	N
视图的alter	Y	N
视图的drop	Y	N
视图的select	Y	Y
视图的desc/describe VIEWNAME	Y	Y
视图的show views/show create view	Y	Y
视图的show columns	Y	Y
视图的select column	Y	Y

