HUBBLE 产品手册 V5.4.1

Hubble 团队

2024-12-13

目录

1 用户手册

1.1 基础知识

1.1.1 对象

在 Hubble 中，您可以使用常见的数据库对象，按照 SQL 标准，分为了 catalog,schema,table。我们可以把它们理解为一个容器或者数据库对象命名空间中的一个层次，主要用来解决命名冲突问题。从概念上说，一个 Hubble 数据库包含多个 catalog，每个 catalog 又包含多个 schema，而每个 schema 又包含多个数据库对象（表、视图、字段等），反过来讲一个数据库对象必然属于一个 schema，而该 schema 又必然属于一个 catalog，这样我们就可以得到该数据库对象的完全限定名称从而解决命名冲突问题了。例如一个数据库表的完全限定名称就可以表示为：**Catalog 名称.Schema 名称. 表名称**。

举个例子，比如想查 Hubble 存储下 tpcds 库下的 customer 表，可以使用：

```
select * from hubble.tpcds.customer;
```

1.1.1.1 目录

在 Hubble 中，目录 `catalog` 相当于给数据源一个命名，如 hubble、system。

```
show catalogs;
```

```
+---------+
| Catalog |
+---------+
| hubble  |
| system  |
+---------+
```

> 这里 system 是只读的系统内置源，存放一些系统数据库。如元信息库，运行状态库等。

1.1.1.2 模式

模式 `schema` 也可理解为我们通常意义上的数据库 `database`，用于管理一组表的集合。模式指代数据库的结构或布局，描述了数据库对象（如表、列、数据类型）之间的关系和属性。

可以使用 USE 指令切换到目标数据库目录下。在所在数据库内可以直接使用对象名指代对象。

```
use hubble.tpcds;

select * from customer;
```

如果在当前目录要使用其他数据库中的对象，需要使用全限定名 `<catalog_name>.<database_name>.<object_name>` 来指代，例如:

```
select * from hubble.db_test.student;
```

创建库时需要指定连接源，例如：

```
create schema hubble.my_db;
```

1.1.1.3 表

Hubble 的表和其他数据库类型中的表一样，主要用于存储数据。表按关系型数据库的形式组织数据，即按行和列来组织存储数据。

Hubble 的表支持多种数据类型，包括基础的布尔，整型，浮点型，精确数值，字符串，日期时间等类型，也支持复杂的结构类型，如数组和 Map。

Hubble 支持按不同的粒度对表进行切分如分区和分桶，也支持事务表的创建。存储在 Hubble 中的表与 hive 兼容，详情可参考 SQL 语法章节详细内容。

1.1.1.4 视图

普通视图

创建语法:

```
CREATE [ OR REPLACE ] VIEW as <Query_SQL>
```

普通视图使用查询语句创建一个逻辑上的虚拟表，它不存储数据，而是实时从源表中获取。视图不能 insert 或 update 数据，每次对视图进行查询时，视图所用的查询语句会被再次执行一次，我们也可以将视图理解为查询的封装。视图不能和表重名。视图的作用很多，比如简化查询、对表内容进行只读权限控制等。

物化视图

创建语法:

```
CREATE [ OR REPLACE ] MATERIALIZED VIEW as <Query_SQL>
```

物化视图是一个具有实际物理存储的表。它将复杂的关联数据转化为一张实际的表，使得数据查询可以直接从物化视图中进行，无需再执行复杂的关联操作。物化视图的数据是预先计算和存储的，它需要通过刷新来更新数据。

```
CREATE OR REPLACE MATERIALIZED VIEW hubble.v_test.v_course2
WITH (
  refresh_interval = '5m',
  grace_period = '5m',
  max_import_duration = '1m'
)
as select * from hubble.db_ai.course ;
```

刷新可以是手动的，也可以是自动的。创建物化视图时，可以通过设置`refresh_interval`属性的值来定义刷新时间。当源表数据更新或删除后，物化视图定时刷新查询结果，可以提供更好的查询性能。

1.1.1.5 函数

您可以使用函数对 Hubble 中的数据进行各种计算和变换。Hubble 拥有大量的内置函数，这些函数可进行字符串处理、数值类型处理、科学计算、日期时间处理、敏感信息加解密、复杂数据类型处理等，具体函数及其用法请参考函数章节。

1.1.2 数据类型

1.1.2.1 BOOLEAN

此类型获取布尔值 `true` 和 `false`

1.1.2.2 TINYINT

8 位有符号整数，最小值为-2^{7}，最大值为 2^{7} - 1

1.1.2.3 SMALLINT

16 位有符号整数，最小值为-2^{15}，最大值为 2^{15} - 1

1.1.2.4 INTEGER

32 位有符号整数，最小值为-2^{31}，最大值为 2^{31} - 1，可使用`INT`代替

1.1.2.5 BIGINT

64 位有符号整数，最小值为 -2^{63} ，最大值为 2^{63} - 1

1.1.2.6 REAL

real 是 32 位不精确，可变精度，基于 IEEE 标准 754 的二进制浮点算法的实现

1.1.2.7 DOUBLE

double 是 64 位不精确，可变精度，基于 IEEE 标准 754 的二进制浮点算法的实现

1.1.2.8 DECIMAL

固定精度的十进制数，支持 38 位精度，但 18 位时性能最好，有两个参数，总位数`precision`, 小数部分的位数`scale`

类型定义示例：

```
DECIMAL(18,3)
```

1.1.2.9 VARCHAR

可变长度字符数据，可设置最大长度。例：`varchar`,`varchar(12)`

1.1.2.10 CHAR

定长字符数据。无参时默认长度为 1，例：`char`,`char(12)`

1.1.2.11 VARBINARY

可变长度二进制数据。二进制数据必须使用前缀为`X`或`x`的十六进制格式。

1.1.2.12 DATE

日期，年月日。

```
DATE '2018-10-15'
```

1.1.2.13 TIME(P)

不带时区的时间，P 为精度，默认为 3。精度的可选值为 3|6|9|12，分别代表毫秒，微秒，纳秒和皮秒。

```
TIME '03:04:05.321'
```

1.1.2.14 TIME WITH TIME ZONE

带时区的时间，秒的精度为毫秒，时区使用 UTC 时区的偏移值表示。

```
TIME '03:04:05.321 +08:00'
```

1.1.2.15 TIMESTAMP(P)

不带时区的时间戳，是 DATE 和 TIME(P) 类型的组合，P 为精度，默认为 3。精度的可选值为 3|6|9|12，分别代表毫秒，微秒，纳秒和皮秒。

```
TIMESTAMP '2001-08-22 03:04:05.321'
```

1.1.2.16 TIMESTAMP(P) WITH TIME ZONE

带时区的时间戳，P 为精度，默认为 3。精度的可选值为 3|6|9|12，分别代表毫秒，微秒，纳秒和皮秒。时区使用 UTC 时区的偏移值表示。

```
TIMESTAMP '2001-08-22 03:04:05.321 +08:00'
```

1.1.2.17 INTERVAL

时间间隔，支持的间隔单位包括，YEAR，QUARTER，MONTH，WEEK，DAY，HOUR，MINUTE，SECOND，MILLISECOND。

```
INTERVAL '3' MONTH
```

1.1.2.18 JSON

JSON 值类型，包括 JSON 对象、JSON 数组、JSON 字符串。

1.1.2.19 ARRAY

数组。

```
ARRAY['a', 'b', 'c']
```

1.1.2.20 MAP

键值对。

```
MAP(ARRAY['a', 'b', 'c'], ARRAY[1, 2, 3])
```

1.1.2.21 ROW

由混合类型的字段组成的行结构，其中的字段可以是任何 SQL 类型。

```
CAST(ROW(1, 2.0) AS ROW(x BIGINT, y DOUBLE))
```

1.2 参考

1.2.1 sql 语句

1.2.1.1 DDL

数据库模式定义语言 DDL(Data Definition Language)，是用于描述数据库中要存储的现实世界实体的语言。主要由 create（添加）、alter（修改）、drop（删除）关键字完成。

1.2.1.1.1 CREATE SCHEMA

规则

```
CREATE SCHEMA [ IF NOT EXISTS ] schema_name
[WITH (property_name = expression [, ...])]
```

描述

创建一个新的，空的库，使用关键字SCHEMA，在 Hubble 中SCHEMA和DATABASE是同义的，库是用来保存表、视图和其他数据对象的命名空间。如果库已经存在，可使用IF NOT EXISTS来进行判断，防止出错。

可选的WITH语句用于设置新创建库的属性，要列出所有可用的库属性，可使用语句进行查询:

```
SELECT * FROM system.metadata.schema_properties
```

例子

创建一个库user在当前的 catalog 下:

```
CREATE SCHEMA user;
```

在 catalog hubble下创建一个tpcds库:

```
CREATE SCHEMA hubble.tpcds;
```

判断 catalog hubble下如果不存在tpcds库则创建:

```
CREATE SCHEMA IF NOT EXISTS hubble.tpcds;
```

创建一个新的 schema 并设置所属于 test 用户:

```
CREATE SCHEMA tpcds AUTHORIZATION test;
```

1.2.1.1.2 CREATE TABLE

规则

```
CREATE TABLE [IF NOT EXISTS]
table_name (
  {column_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

创建一个空表，如果是创建有数据的表，请使用CREATE TABLE AS

防止表已存在导致的报错使用IF NOT EXISTS。

可选WITH子句可用于在新创建的表或单列上设置属性。要列出所有可用的表属性，请运行以下查询:

```
SELECT * FROM system.metadata.table_properties;
```

可选列属性请运行以下查询:

```
SELECT * FROM system.metadata.column_properties;
```

LIKE子句可用于将现有表中的所有列定义包含在新表中。可以指定多个LIKE子句，允许从多个表中复制列。

```
create table
  hubble.db_test.tab_l (
    idx int,
    name varchar,
    like hubble.db_test.account,
    like hubble.db_test.page
  );
```

如果指定了INCLUDING PROPERTIES，则所有表属性都将复制到新表中。如果WITH子句指定的属性名与复制的属性之一相同，则将使用WITH子句中的值。默认排除属性。最多可以为一个表指定INCLUDING PROPERTIES选项。

例子

在 catalog hubble中创建ORC格式的itemline表:

```
CREATE TABLE
  hubble.tpch.lineitem (
    l_orderkey integer,
    l_partkey integer,
    l_suppkey integer,
    l_linenumber integer,
    l_quantity double,
    l_extendedprice double,
    l_discount double,
    l_tax double,
    l_returnflag varchar,
    l_linestatus varchar,
    l_shipdate varchar,
    l_commitdate varchar,
    l_receiptdate varchar,
    l_shipinstruct varchar,
    l_shipmode varchar,
    l_comment varchar
  )
WITH
  (format = 'ORC');
```

如果已经指定了 hubble 和 tpch，则可以省略:

```
CREATE TABLE
  lineitem (
    l_orderkey integer,
```

```
    l_partkey integer,
    l_suppkey integer,
    l_linenumber integer,
    l_quantity double,
    l_extendedprice double,
    l_discount double,
    l_tax double,
    l_returnflag varchar,
    l_linestatus varchar,
    l_shipdate varchar,
    l_commitdate varchar,
    l_receiptdate varchar,
    l_shipinstruct varchar,
    l_shipmode varchar,
    l_comment varchar
  )
WITH
  (format = 'ORC');
```

分区表

```
CREATE TABLE
  p2_t2 (
    account varchar (32),
    date_time varchar (32),
    date_p varchar (32),
    biz_date integer
  )
WITH
  (
    format = 'MULTIDELIMIT',
    partitioned_by = ARRAY['date_p', 'biz_date'],
    textfile_field_separator = '|+|'
  );

-- 加载带分区表数据

从本地文件系统加载数据到表中的指定分区列

load data local inpath '/opt/zl/d_data/v3.txt' into table p2_t2 PARTITION(date_p
    ↪ ='20230320',biz_date='3');

hubble方式导入数据的，执行call语句

call system.sync_partition_metadata (
  schema_name => 'zl',
  table_name => 'p2_t2',
```

```
  mode => 'FULL',
  case_sensitive => true
);
```

创建表带有多个分隔符:

```
CREATE TABLE
  t2 (id varchar, name varchar, age varchar)
WITH
  (
    format = 'MULTIDELIMIT', -- 多分隔符设置
    textfile_field_separator = '|+|' -- 分隔符定义
  );

-- 添加注解
comment on table t2 is '测试2表';
comment on column t2.id is '表id';
comment on column t2.name is '名称';
comment on column t2.age is '年龄';
```

format='MULTIDELIMIT' 多分隔符设置

textfile_field_separator 分隔符

textfile_field_separator_escape 分割转义

分隔符: \t

```
create table
  customer_address (
    ca_address_sk int,
    ca_gmt_offset decimal (5, 2),
    ca_location_type char (20)
  )
WITH
  (
    format = 'MULTIDELIMIT',
    textfile_field_separator = U&'\0009'
  );
```

分隔符: \001

```
create table
  t3 (a varchar, b varchar)
WITH
  (
    format = 'MULTIDELIMIT',
    textfile_field_separator = U&'\0001'
  );
```

创建事务表:

```
-- 事务表 用于合并更新数据 格式类型必须ORC格式
CREATE TABLE
  t_p2_t1 (
    account varchar (32),
    date_time varchar (32),
    date_p varchar (32),
    biz_date integer
  )
WITH
  (format = 'ORC', transactional = true);
```

> transactional=true 事务
>
> 事务表用于合并更新数据格式类型必须 ORC 格式

创建表lineitem如果表不存在，并且可以增加comment:

```
CREATE TABLE
  lineitem (
    l_orderkey integer,
    l_partkey integer,
    l_suppkey integer,
    l_linenumber integer,
    l_quantity double,
    l_extendedprice double,
    l_discount double,
    l_tax double,
    l_returnflag varchar,
    l_linestatus varchar,
    l_shipdate varchar COMMENT 'ship date in lineitem.',
    l_commitdate varchar,
    l_receiptdate varchar,
    l_shipinstruct varchar,
    l_shipmode varchar,
    l_comment varchar
  ) COMMENT 'A table belong to tpch.'
```

创建一个从lineitem复制的表，并增加一些字段:

```
CREATE TABLE
  new_lineitem (
    n_l_orderkey integer,
    like lineitem,
    n_l_comment varchar
  );
```

1.2.1.1.3 CREATE TABLE AS

规则

```
CREATE TABLE [IF NOT EXISTS] table_name [(column_alias, ...)]
[COMMENT table_comment]
[WITH (property_name = expression [, ...])]
AS query
[WITH [NO] DATA]
```

描述

创建包含 SELECT 查询结果的新表。

防止表已存在导致的报错使用IF NOT EXISTS。

可选WITH子句可用于设置新创建的表的属性。要列出所有可用的表属性，请运行以下查询：

```
SELECT * FROM system.metadata.table_properties;
```

例子

创建一个新表lineitem_column_aliased，复制lineitem表的指定列的字段类型和数据内容：

```
CREATE TABLE
  lineitem_column_aliased (orderkey, quantity) AS
SELECT
  l_orderkey,
  l_quantity
FROM
  lineitem;
```

创建一个lineitem表，按l_quantity表的l_orderkey字段分组统计：

```
CREATE TABLE
  lineitem_by_orderkey COMMENT 'Summary of lineitem by orderkey'
WITH
  (format = 'ORC') AS
SELECT
  l_orderkey,
  sum(l_quantity) AS price
FROM
  lineitem
GROUP BY
  l_orderkey;
```

创建一个空的empty_lineitem表：

```
CREATE TABLE
  empty_lineitem AS
SELECT
  *
FROM
```

```
  lineitem
WITH
  NO DATA;
```

1.2.1.1.4 CREATE VIEW

规则

```
CREATE [OR REPLACE] VIEW view_name
[SECURITY {DEFINER | INVOKER}]
AS query
```

描述

使用 SELECT 语句创建视图。该视图是一个逻辑表，可被查询使用。视图不包含任何数据，每次查询视图时，都会再次执行创建该视图的查询语句。

可使用CREATE OR REPLACE语句避免视图已存在导致的视图创建错误。

例子

创建一个简单的test视图来访问order表:

```
CREATE VIEW
  test AS
SELECT
  orderkey,
  orderstatus,
  totalprice / 2 AS half
FROM
  orders;
```

创建一个orders_by_date视图来访问order表的聚合数据:

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

创建一个覆盖视图:

```
CREATE OR REPLACE VIEW
  test AS
SELECT
  orderkey,
  orderstatus,
  totalprice / 4 AS quarter
```

```
FROM
  orders;
```

1.2.1.1.5 CREATE MATERIALIZED VIEW

规则

```
CREATE [OR REPLACE] MATERIALIZED VIEW view_name AS query
```

描述

使用 SELECT 语句创建物化视图。物化视图是一个具有实际物理存储的表。它将复杂的关联数据转化为一张实际的表，使得数据查询可以直接从物化视图中进行，无需再执行复杂的关联操作。物化视图的数据是预先计算和存储的，它需要通过刷新来更新数据。

可使用CREATE OR REPLACE语句避免视图已存在导致的视图创建错误。

> 使用物化视图时要保证缓存服务处于开启状态。

例子

基于统计分析结果创建一个物化视图：

```
CREATE MATERIALIZED VIEW hubble.v_test.mv_cust_return2
WITH (
  refresh_interval = '5m',
  grace_period = '10m',
  max_import_duration = '3m'
) AS
    SELECT
      sr_customer_sk ctr_customer_sk,
      sr_store_sk ctr_store_sk,
      sum(sr_return_amt) ctr_total_return
    FROM
    hubble.tpcds_external.store_returns,
    hubble.tpcds_external.date_dim
    WHERE ( (sr_returned_date_sk = d_date_sk) AND (d_year > 2000) )
    GROUP BY sr_customer_sk, sr_store_sk
;
```

创建一个单表映射物化视图：

```
CREATE OR REPLACE MATERIALIZED VIEW
  hubble.v_test.v_course2
WITH
  (
    refresh_interval = '5m',
    grace_period = '5m',
    max_import_duration = '1m'
  ) as
```

```
select
  *
from
  mysql.db_ai.course;
```

参数说明： refresh_interval, 刷新频率。配置的最小时间为 5 分钟; grace_period, 查询可以接受的过期快照时长。要大于等于 refresh_interval; max_import_duration, 刷新运行的最长时间, 要小于等于 refresh_interval;

创建一个定时刷新的物化视图:

```
CREATE MATERIALIZED VIEW hubble.v_test.mv_cust_return5
WITH (
  cron = '30 2 * * *',
  grace_period = '10m',
  max_import_duration = '3m',
  incremental_column = 'ctr_store_sk'
) AS
    SELECT
      sr_customer_sk ctr_customer_sk,
      sr_store_sk ctr_store_sk,
      sum(sr_return_amt) ctr_total_return
    FROM
      hubble.tpcds_external.store_returns,
      hubble.tpcds_external.date_dim
    WHERE ( (sr_returned_date_sk = d_date_sk) AND (d_year > 2000) )
    GROUP BY sr_customer_sk, sr_store_sk
;
```

> cron 属性开启定时刷新;
>
> incremental_column 属性用于标识增量刷新判断的列。

1.2.1.1.6 ALTER TABLE

规则

```
ALTER TABLE name ADD COLUMN column_name data_type [ COMMENT comment ] [ WITH (
    ↪ property_name = expression [, ...] ) ]
ALTER TABLE name DROP COLUMN column_name
ALTER TABLE name RENAME COLUMN column_name TO new_column_name
```

描述

修改一个已存在的表的相关属性。

例子

为 sales 增加一个列:

```
ALTER TABLE sales ADD COLUMN address varchar;
```

删除address列:

```
ALTER TABLE sales DROP COLUMN address;
```

重命名列:

```
ALTER TABLE sales RENAME COLUMN id TO sale_id;
```

1.2.1.1.7 ALTER TABLE EXECUTE

增加了表存储优化功能，便于用户简单直接的对表完成小文件合并，建议此功能可以在非业务时段进行。需要注意，考虑到分桶表不存在小文件问题，此功能不对分桶表生效。

```
-- 必须在执行表存储优化前，设置non_transactional_optimize_enabled参数
set session hubble.non_transactional_optimize_enabled=true;
-- 默认的file_size_threshold阈值为100MB，即小于此阈值的存储文件将会被compaction
ALTER TABLE hubble.tpcds.item EXECUTE optimize;
-- 用户可根据实际情况，指定需要做compaction文件的存储大小阈值
ALTER TABLE hubble.tpcds.item EXECUTE optimize(file_size_threshold => '10MB');
-- 指定表的具体分区，进行compaction优化
ALTER TABLE hubble.tpcds.item EXECUTE optimize where batch_date = '20220610';
```

注意事项:

- 如果对当前正在优化的表运行查询，则可能会读取重复的行。
- 在优化操作期间发生异常的极端情况下，需要手动清理表目录。在这种情况下，请参考日志并查询失败消息，以查看需要删除哪些文件。

1.2.1.1.8 ALTER VIEW

规则

```
ALTER VIEW name RENAME TO new_name
ALTER VIEW name SET AUTHORIZATION ( user | USER user | ROLE role )
```

描述

更改现有视图的属性。

例子

```
ALTER VIEW teacher RENAME TO people;
```

1.2.1.1.9 DROP SCHEMA

规则

```
DROP SCHEMA [ IF EXISTS ] schema_name
```

描述

删除一个已存在的库，库必须是空的。

例子

删除 web 库

```
DROP SCHEMA web;
```

1.2.1.1.10 DROP TABLE

规则

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

描述

删除已存在的表。

例子

删除表lineitem_by_date；如果存在表orders_by_date则进行删除。

```
DROP TABLE lineitem_by_date;
DROP TABLE IF EXISTS orders_by_date;
```

1.2.1.1.11 DROP VIEW

规则

```
DROP [MATERIALIZED] VIEW [ IF EXISTS ] view_name
```

描述

删除一个已存在的视图。

如果删除的是一个物化视图，需要带关键字 MATERIALIZED 。

例子

删除视图view_order

```
DROP VIEW view_order;
```

删除物化视图mv_customer;

```
DROP MATERIALIZED VIEW mv_customer;
```

删除视图如果已存在view_orders

```
DROP VIEW IF EXISTS view_orders;
```

1.2.1.1.12 USE

规则

```
USE catalog.schema;
USE schema;
```

描述

更新 session 指定当前的catalog和schema。

例子

指定 catalog 为 hubble，库为 tpcds：

```
use hubble.tpcds;
```

1.2.1.1.13 SHOW CATALOGS

规则

```
SHOW CATALOGS [ LIKE pattern ]
```

描述

查看所有可用的 catalog，可选LIKE用来限制查看的 catalog 名。

1.2.1.1.14 SHOW SCHEMAS

规则

```
SHOW SCHEMAS [ FROM catalog ] [ LIKE pattern ]
```

描述

查看指定 catalog 或当前 catalog 下所有库，可选LIKE用来限制查看的 database 名。

1.2.1.1.15 SHOW TABLES

规则

```
SHOW TABLES [ FROM schema ] [ LIKE pattern ]
```

描述

查看指定库或当前库下所有表，可选LIKE用来限制查看的表名。

1.2.1.1.16 SHOW TABLE LOCATION

规则

```
SHOW TABLE LOCATION tableName
```

描述

查看指定表的文件系统存储路径。

1.2.1.1.17 SHOW PARTITIONS

规则

```
SHOW PARTITIONS tableName
```

描述

查看指定表的分区信息列表。

1.2.1.1.18 SHOW COLUMNS

规则

```
SHOW COLUMNS [ FROM table ] [ LIKE pattern ]
```

描述

查看指定表的字段，可选LIKE用来限制查看的表名。

示例：

```
SHOW COLUMNS FROM call_center LIKE '%name';
```

1.2.1.1.19 SHOW CREATE SCHEMA

规则

```
SHOW CREATE SCHEMA schema_name
```

描述

查看指定的库的创建语句。

1.2.1.1.20 SHOW CREATE TABLE

规则

```
SHOW CREATE TABLE table_name
```

描述

查看指定的表的创建语句。

1.2.1.1.21 SHOW CREATE VIEW

规则

```
SHOW CREATE [MATERIALIZED] VIEW view_name
```

描述

查看指定的视图的创建语句。

1.2.1.1.22 DESCRIBE

规则

```
DESCRIBE table_name
```

描述

查看表的列，包括列名，类型等。

同时也可以使用SHOW COLUMNS FROM table语法，二者同义。

1.2.1.1.23 COMMENT ON

规则

```
COMMENT ON COLUMN table_name.column_name IS 'comment';
```

描述

对字段添加注释。

1.2.1.1.24 START TRANSACTION

规则

```
START TRANSACTION [ mode [, ...] ]
```

> 当模式是其中之一：
>
> ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE } READ { ONLY | WRITE }

描述

为当前会话启动一个新事务

例子

```
START TRANSACTION;

START TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION READ WRITE;

START TRANSACTION ISOLATION LEVEL READ COMMITTED, READ ONLY;

START TRANSACTION READ WRITE, ISOLATION LEVEL SERIALIZABLE;
```

1.2.1.1.25 COMMIT

规则

```
 COMMIT tablename
```

描述

提交当前事物

例子

```
COMMIT;

COMMIT WORK;
```

1.2.1.1.26 ROLLBACK

规则

```
ROLLBACK tablename;
```

描述回滚当前事物

例子

```
ROLLBACK;

ROLLBACK WORK;
```

1.2.1.1.27 EXPLAIN

规则

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

描述

显示语句的逻辑或分布式执行计划，或验证语句。默认情况下显示分布式计划。分布式计划的每个计划片段由单个或多个节点执行。

例子

```
EXPLAIN SELECT a, b FROM (
  SELECT a, MAX(b) AS b FROM t GROUP BY a
) AS x;
```

1.2.1.1.28 CALL

规则

```
CALL procedure_name ( [ name => ] expression [, ...] )
```

描述

调用系统表。

例子

结束某个耗资源的查询

```
CALL system.runtime.kill_query(query_id => '20190614084824001 60NjkwT', message
    ↪ => 'Using too many resources');
```

在catalog hubble中创建新分区

```
CALL system.create_empty_partition(
    schema_name => 'web',
    table_name => 'page_views',
    partition_columns => ARRAY['ds', 'country'],
    partition_values => ARRAY['2019-06-09', 'CN']);
```

刷新缓存及表的统计信息：

```
-- flush所有的缓存数据
CALL hubble.system.flush_metadata_cache();
-- flush指定表的分区元信息缓存
CALL hubble.system.flush_metadata_cache(
    schema_name => 'tpcds',
    table_name => 'catalog_sales',
    partition_column => ARRAY['cs_sold_date_sk'],
    partition_value => ARRAY['2452634']
);
```

1.2.1.2 DML

数据操纵语言，SQL 中处理数据等操作统称为数据操纵语言，包括 INSERT,UPDATE,DELETE 等等。

1.2.1.2.1 INSERT

规则

```
INSERT INTO table_name [ ( column [, ... ] ) ] query
```

描述

在 table 中插入新行。

如果指定了列名列表，则它们必须与查询生成的列完全匹配。表中未出现在列名列表中的每一列都将用空值填充。如果未指定列名，则查询生成的列必须与要插入的表中的列完全匹配。

例子

从已有表new_customer加载数据到customer

```
INSERT INTO customer SELECT * FROM new_customer
```

插入单行到表customer

```
INSERT INTO customer VALUES(1,'joe')
```

向表customer中插入多行

```
INSERT INTO customer VALUES(2,'bob'),(3,'john')
```

通过指定列插入表

```
INSERT INTO
  nation (nationkey, name, regionkey, comment)
VALUES
  (1, 'CHINA', 1, 'no comment');
```

插入不包含列名的值，将对列comment补充NULL

```
INSERT INTO
  nation (nationkey, name, regionkey)
```

```
VALUES
  (1, 'CHINA', 1);
```

1.2.1.2.2 UPDATE

规则

```
UPDATE table_name SET [ ( column [, ... ] ) ] [ WHERE condition ]
```

描述

更新表中指定字段数据。使用WHERE语句时进行条件更新，否则进行全表更新。

例子

更新表customer中customer_id为 2 的customer_name为 peter。

```
update customer
set
  customer_name = 'peter'
where
  customer_id = 2
```

> 目前只有事务表支持UPDATE语法。

1.2.1.2.3 DELETE

规则

```
DELETE FROM table_name [ WHERE condition ]
```

描述

删除表中数据。使用WHERE时会根据条件匹配进行删除，不使用WHERE条件时会删除表中所有数据。

例子

删除lineitem表中shipmode等于AIR的数据。

```
DELETE FROM lineitem WHERE shipmode = 'AIR';
```

两表关联删除

```
DELETE FROM lineitem
WHERE
  orderkey IN (
    SELECT
      orderkey
    FROM
      orders
    WHERE
      priority = 'LOW'
  );
```

删除表中所有数据

```
DELETE FROM lineitem
```

> 目前只有事务表才支持DELETE语法。

> 对于更新和删除尽量使用条件进行，否则可能占用集群资源，并且效率不高。

1.2.1.2.4 MERGE

规则

```
MERGE INTO target_table [ [ AS ]  target_alias ]
USING { source_table | query } [ [ AS ] source_alias ]
ON search_condition
when_clause [...]
where when_clause is one of

WHEN MATCHED [ AND condition ]
    THEN DELETE
WHEN MATCHED [ AND condition ]
    THEN UPDATE SET ( column = expression [, ...] )
WHEN NOT MATCHED [ AND condition ]
    THEN INSERT [ column_list ] VALUES (expression, ...)
```

描述

有条件地更新和/或删除表中的行和/或在表中插入新行。

MERGE 支持任意数量的具有不同 MATCH 条件的 WHEN 子句，在由 MATCHED 状态和匹配条件选择的第一个 WHEN 子句中执行 DELETE、UPDATE 或 INSERT 操作。

对于每个源行，将按顺序处理 WHEN 子句。仅执行第一个匹配的 WHEN 子句，并忽略后续子句。当单个目标表行与多个源行匹配时，将引发MERGE_TARGET_ROW_MULTIPLE_MATCHES异常。

如果源行与任何 WHEN 子句不匹配，并且没有“不匹配时”子句，则忽略源行。

在具有 UPDATE 操作 WHEN 子句中，列值表达式可以依赖于目标或源的任何字段。在不匹配的情况下，插入表达式可以依赖于源的任何字段。

例子

注意：合并更新的表必须是事务表和主键约束的

```
MERGE INTO t_p2_t1 t USING tmp_01 s ON (
  t.account = s.account
  and t.date_p = s.date_p
) WHEN MATCHED THEN
UPDATE
SET
  date_time = s.date_time;
```

```
MERGE INTO t_p2_t1 t USING (
  select
    *
  from
    tmp_01 tt
) s ON (
  t.account = s.account
  and t.date_p = s.date_p
) WHEN MATCHED THEN
UPDATE
SET
  date_time = s.date_time
```

1.2.1.3 DQL

数据查询语言

1.2.1.3.1 SELECT

规则

```
[ WITH with_query [, ...] ]
SELECT [ ALL | DISTINCT ] select_expr [, ...]
[ 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 } ]
```

from_item是以下的一种:

```
table_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
```

或

```
from_item join_type from_item [ ON join_condition | USING ( join_column [, ...]
    ↪ ) ]
```

join_type包括:

```
[ INNER ] JOIN
LEFT [ OUTER ] JOIN
RIGHT [ OUTER ] JOIN
FULL [ OUTER ] JOIN
CROSS JOIN
```

grouping_element包括:

```
()
expression
GROUPING SETS ( ( column [, ...] ) [, ...] )
CUBE ( column [, ...] )
ROLLUP ( column [, ...] )
```

描述

查询语法，由一个表或多个表查询数据。

详细解释

WITH 子句

用于定义在查询时使用的命名关系，它允许扁平化的嵌套查询或简化子查询。例如，以下查询是等效的:

```
SELECT a, b
FROM (
  SELECT a, MAX(b) AS b FROM t GROUP BY a
) AS x;

WITH x AS (SELECT a, MAX(b) AS b FROM t GROUP BY a)
SELECT a, b FROM x;
```

也可以用于多子查询语句:

```
WITH
  t1 AS (SELECT a, MAX(b) AS b FROM x GROUP BY a),
  t2 AS (SELECT a, AVG(d) AS d FROM y GROUP BY a)
SELECT t1.*, t2.*
FROM t1
JOIN t2 ON t1.a = t2.a;
```

并且可以是一个链式:

```
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;
```

GROUP BY 子句

GROUP BY将 SELECT 语句的输出分成包含匹配值的行组，简单的GROUP BY包含由列组成的表达式，也可以按位置选择输出列的序号（从 1 开始）。

以下查询是等效的，都是按nationkey对查询进行分组，第一个使用序号，第二个使用列名:

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

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

GROUP BY语句也可以按不在SELECT内的字段进行分组：

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

所有的输出都必须是 select 中的字段或者是聚合函数。

复杂的GROUP BY语句，支持使用GROUPING SETS,CUBE，ROLLUP语法，允许对需要的多组列进行聚合分析，复杂分组操作不支持对输入列组成的表达式进行分组，只允许使用列名和序号。

复杂分组操作通常等价于一个联合的所有简单分组的表达式，当然，由于连接源的不同，有些等价表达式可能并不适用。

- GROUPING SETS

允许指定分组的列的多个列表，不属于分组列的指定子列表的列将被设置为 NULL。

```
SELECT * FROM shipping;
```

```
  origin_city | origin_zip | destination_city | destination_zip | package_weight
--------------+------------+------------------+-----------------+----------------
   ↪
  beijing     | 100000     | tianjin          | 300000          |             12
  tianjin     | 300010     | shanghai         | 200000          |             60
  beijing     | 100100     | chongqing        | 400000          |             38
  guangzhou   | 510000     | tianjin          | 300100          |             14
  beijing     | 100012     | shanghai         | 200000          |             78
  beijing     | 100000     | tianjin          | 300010          |             40

(6 rows in set)
```

分组例子：

```
SELECT origin_city, origin_zip, destination_city, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS (
    (origin_city),
    (origin_city, origin_zip),
    (destination_city));
```

```
  origin_city | origin_zip | destination_city | _col3
+-------------+------------+------------------+-------+
  beijing     | NULL       | NULL             |   168
  beijing     | 100100     | NULL             |    38
  tianjin     | 300010     | NULL             |    60
  guangzhou   | NULL       | NULL             |    14
  beijing     | 100012     | NULL             |    78
  beijing     | 100000     | NULL             |    52
  tianjin     | NULL       | NULL             |    60
  guangzhou   | 510000     | NULL             |    14
  NULL        | NULL       | shanghai         |   138
  NULL        | NULL       | chongqing        |    38
```

```
  NULL          | NULL          | tianjin           |     66

(11 rows in set)
```

逻辑上等同于下面的语法:

```
SELECT origin_city, NULL, NULL, sum(package_weight)
FROM shipping GROUP BY origin_city
UNION ALL
SELECT origin_city, origin_zip, NULL, sum(package_weight)
FROM shipping GROUP BY origin_city, origin_zip
UNION ALL
SELECT NULL, NULL, destination_city, sum(package_weight)
FROM shipping GROUP BY destination_city;
```

使用 UNION ALL 将从基础表中读取三次数据，而使用GROUPING SETS只需要读取一次。

- CUBE

CUBE为给定的列集生成所有可能的分组集（即幂集）。例如，查询:

```
SELECT origin_city, destination_city, sum(package_weight)
FROM shipping
GROUP BY CUBE (origin_city, destination_city);
```

等同于:

```
SELECT origin_city, destination_city, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS (
    (origin_city, destination_city),
    (origin_city),
    (destination_city),
    ());
```

得到结果:

```
  origin_city | destination_city | _col2
+-------------+------------------+-------+
  tianjin     | NULL             |    60
  NULL        | chongqing        |    38
  NULL        | NULL             |   242
  guangzhou   | NULL             |    14
  tianjin     | shanghai         |    60
  beijing     | chongqing        |    38
  guangzhou   | tianjin          |    14
  NULL        | shanghai         |   138
  NULL        | tianjin          |    66
  beijing     | shanghai         |    78
  beijing     | tianjin          |    52
  beijing     | NULL             |   168
```

```
(12 rows in set)
```

- ROLLUP

为给定的列集生成所有可能的分类汇总。例如，查询：

```
SELECT origin_city, origin_zip, sum(package_weight)
FROM shipping
GROUP BY ROLLUP (origin_city, origin_zip);
```

等同于

```
SELECT origin_city, origin_zip, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS ((origin_city, origin_zip), (origin_city), ());
```

得到结果

```
 origin_city | origin_zip | _col2
+------------+------------+------+
 beijing     | 100000     |   52
 tianjin     | NULL       |   60
 beijing     | 100100     |   38
 guangzhou   | NULL       |   14
 beijing     | NULL       |  168
 beijing     | 100012     |   78
 NULL        | NULL       |  242
 guangzhou   | 510000     |   14
 tianjin     | 300010     |   60

(9 rows in set)
```


- 组合多个 grouping 语法

```
SELECT origin_city, destination_city, origin_zip, sum(package_weight)
FROM shipping
GROUP BY
    GROUPING SETS ((origin_city, destination_city)),
    ROLLUP (origin_zip);
```

可以重写为：

```
SELECT origin_city, destination_city, origin_zip, sum(package_weight)
FROM shipping
GROUP BY
    GROUPING SETS ((origin_city, destination_city)),
    GROUPING SETS ((origin_zip), ());
```

等同于：

```
SELECT origin_city, destination_city, origin_zip, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS (
    (origin_city, destination_city, origin_zip),
    (origin_city, destination_city));
```

输出结果:

```
 origin_city | destination_city | origin_zip | _col3
+------------+------------------+------------+------+
 beijing     | tianjin          | 100000     |    52
 beijing     | tianjin          | NULL       |    52
 guangzhou   | tianjin          | NULL       |    14
 beijing     | chongqing        | 100100     |    38
 guangzhou   | tianjin          | 510000     |    14
 beijing     | chongqing        | NULL       |    38
 tianjin     | shanghai         | NULL       |    60
 tianjin     | shanghai         | 300010     |    60
 beijing     | shanghai         | NULL       |    78
 beijing     | shanghai         | 100012     |    78

(10 rows in set)
```

ALL和DISTINCT确定重复分组集是否每个都产生不同的输出行。当在同一查询中组合多个复杂分组集时，这尤其有用。例如，以下查询:

```
SELECT origin_city, destination_city, origin_zip, sum(package_weight)
FROM shipping
GROUP BY ALL
    CUBE (origin_city, destination_city),
    ROLLUP (origin_city, origin_zip);
```

等同于:

```
SELECT origin_city, destination_city, origin_zip, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS (
    (origin_city, destination_city, origin_zip),
    (origin_city, origin_zip),
    (origin_city, destination_city, origin_zip),
    (origin_city, origin_zip),
    (origin_city, destination_city),
    (origin_city),
    (origin_city, destination_city),
    (origin_city),
    (origin_city, destination_city),
    (origin_city),
    (destination_city),
```

```
    ());
```

而如果使用DISTINCT:

```
SELECT origin_city, destination_city, origin_zip, sum(package_weight)
FROM shipping
GROUP BY DISTINCT
    CUBE (origin_city, destination_city),
    ROLLUP (origin_city, origin_zip);
```

则等同于:

```
SELECT origin_city, destination_city, origin_zip, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS (
    (origin_city, destination_city, origin_zip),
    (origin_city, origin_zip),
    (origin_city, destination_city),
    (origin_city),
    (destination_city),
    ());
```

- GROUPING 操作

分组操作返回转换为十进制的位集，指示分组中存在哪些列。它必须与GROUPING SETS, ROLLUP, CUBE或 GROUP BY一起使用，其参数必须与相应GROUPING SETS, ROLLUP, CUBE或 GROUP BY子句中引用的列完全匹配。

为了计算特定行的结果位集，位被分配给参数列，最右边的列是最低有效位。对于给定的分组，如果分组中包含相应的列，则位设置为 0，否则设置为 1。例如，考虑下面的查询:

```
SELECT origin_city, origin_zip, destination_city, sum(package_weight),
       grouping(origin_city, origin_zip, destination_city)
FROM shipping
GROUP BY GROUPING SETS (
        (origin_city),
        (origin_city, origin_zip),
        (destination_city));
```

结果:

```
 origin_city | origin_zip | destination_city | _col3 | _col4
+------------+------------+------------------+-------+------+
 beijing     | NULL       | NULL             |   168 |     3
 beijing     | 100000     | NULL             |    52 |     1
 guangzhou   | NULL       | NULL             |    14 |     3
 guangzhou   | 510000     | NULL             |    14 |     1
 tianjin     | NULL       | NULL             |    60 |     3
 beijing     | 100012     | NULL             |    78 |     1
 NULL        | NULL       | tianjin          |    66 |     6
 NULL        | NULL       | chongqing        |    38 |     6
 NULL        | NULL       | shanghai         |   138 |     6
```

```
 tianjin      | 300010      | NULL                |     60 |      1
 beijing      | 100100      | NULL                |     38 |      1

(11 rows in set)
```

HAVING 子句

HAVING 子句与聚合函数和 GROUP BY 子句一起使用，以控制选择哪些组。HAVING 子句消除不满足给定条件的组。在计算组和聚合之后有过滤器组。

以下示例查询 customer 表并选择帐户余额大于指定值的组：

```
SELECT count(*), c_mktsegment, c_nationkey,
       CAST(sum(c_acctbal) AS bigint) AS totalbal
FROM customer
GROUP BY c_mktsegment, c_nationkey
HAVING sum(c_acctbal) > 542666087
ORDER BY totalbal DESC;
```

结果：

```
 _col0  | c_mktsegment | c_nationkey | totalbal
+-------+--------------+-------------+-----------+
 120838 | AUTOMOBILE   |          20 | 543268712
 120339 | BUILDING     |          10 | 543056590
 120760 | HOUSEHOLD    |           6 | 543017181

(3 rows in set)
```

UNION | INTERSECT | EXCEPT 子句

UNION INTERSECT EXCEPT 都是 set 操作. 这些子句用于将多个 select 语句的结果组合到单个结果集中：

```
query UNION [ALL | DISTINCT] query
query INTERSECT [DISTINCT] query
query EXCEPT [DISTINCT] query
```

参数ALL或DISTINCT控制最终结果集中包含哪些行。如果指定了参数ALL，则即使行相同，也包括所有行。如果指定了参数DISTINCT，则组合结果集中只包含唯一的行。如果两者都未指定，则行为默认为DISTINCT。INTERSECT ↪ 或EXCEPT不支持ALL参数。

除非通过括号明确指定顺序，否则将从左到右处理多个集合操作。此外，INTERSECT比EXCEPT和UNION优先级更高。也就是说，A UNION B INTERSECT C EXCEPT D，等同于A UNION (B INTERSECT C)EXCEPT D。

- UNION

连接操作，返回两个结果合并的集合。

```
SELECT 13
UNION
SELECT 42;
```

```
 _col0
+-------+
    13
    42

(2 rows in set)
```

```
SELECT 13
UNION
SELECT * FROM (VALUES 42, 13);
```

```
 _col0
+-------+
    42
    13
(2 rows in set)
```

```
SELECT 13
UNION ALL
SELECT * FROM (VALUES 42, 13);
```

```
 _col0
+-------+
    42
    13
    13

(3 rows in set)
```


由上面的三个例子可以看出，如果不带ALL实际结果进行了合并去重。

- INTERSECT

返回两个结果集的交集。也就是说返回两个结果集中都有的结果。

```
SELECT * FROM (VALUES 13, 42)
INTERSECT
SELECT 13;
```

```
 _col0
+-------+
   13

(1 row in set)
```

- EXCEPT

返回第一个集合的相对补集。即在第一个集合中存在，在第二个集合中不存在的值。

```
SELECT * FROM (VALUES 13, 42)
EXCEPT
SELECT * FROM (VALUES 13, 45);
```

```
 _col0
+-------+
    42

(1 row in set)
```

ORDER BY 子句

ORDER BY子句用于给结果集进行排序。

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

ORDER BY子句在GROUP BY或HAVING之后，在OFFSET,LIMIT或FETCH FIRST之前计算。默认 NULL 排在后边，与升降序无关。

OFFSET 子句

OFFSET用于跳过结果集的多个行。

如果存在ORDER BY，则将对排序后的结果集计算，并跳过指定行，结果仍然是排序的。

```
SELECT n_name FROM nation ORDER BY n_name OFFSET 22;
```

```
     n_name
+----------------+
 UNITED KINGDOM
 UNITED STATES
 VIETNAM

(3 rows in set)
```

当跳过的行数大于结果集的大小，将返回空。

LIMIT或FETCH FIRST子句

LIMIT或FETCH FIRST子句限制结果集返回的行数。

```
LIMIT { count | ALL }
```

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

下面的示例查询一个大表，但是LIMIT子句限制输出只有五行（因为查询缺少 order by，所以返回的行是任意的）：

```
SELECT o_orderdate FROM orders LIMIT 5;
```

```
 o_orderdate
+-------------+
 1994-02-12
 1997-07-11
 1994-06-18
 1992-01-19
 1996-09-03
(5 rows in set)
```

LIMIT ALL等同于LIMIT。

FETCH FIRST子句支持FIRST或NEXT关键字以及ROW或ROWS关键字。这些关键字是等效的，关键字的选择对查询执行没有影响。

如果在FETCH FIRST子句中未指定计数，则默认为 1：

```
SELECT o_orderdate FROM orders FETCH FIRST ROW ONLY;
```

```
 o_orderdate
+-------------+
 1994-02-12

(1 row in set)
```

如果存在OFFSET子句，则LIMIT或者FETCH FIRST在其后进行计算。

```
SELECT * FROM (VALUES 5, 2, 4, 1, 3) t(x) ORDER BY x OFFSET 2 LIMIT 2;
```

```
 x
+---+
 3
 4
(2 rows in set)
```

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

如果只指定了参数，则结果集将限制为由计数确定的行数。

如果指定了WITH TIES的参数，则需要存在ORDER BY子句。结果集由同一组的所有行组成，这些行与ORDER BY子句中的排序所建立的最后一个行（“ties”）相同。结果集排序：

```
SELECT n_name, n_regionkey FROM nation ORDER BY n_regionkey FETCH FIRST ROW WITH
    ↪ TIES;
```

```
  n_name   | n_regionkey
+----------+-------------+
 ETHIOPIA  |           0
 MOROCCO   |           0
 KENYA     |           0
```

```
 ALGERIA    |           0
 MOZAMBIQUE|           0

(5 rows in set)
```

TABLESAMPLE

包含的采样方法有：

- BERNOULLI

选择每一行作为表样本，其概率为样本百分比。当使用 BERNOULLI 方法对一个表进行采样时，将扫描该表的所有物理块，并跳过某些行（基于采样百分比与运行时计算的随机值之间的比较）。

结果中包含的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，可能会影响总查询时间。

- SYSTEM

此采样方法将表划分为数据的逻辑段，并以此粒度对表进行采样。此采样方法要么选择特定数据段中的所有行，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）。

系统采样中选择的行将取决于使用的 catalog。例如，当与 Hubble 一起使用时，它取决于数据上的布局方式。这种方法不能保证独立的抽样概率。

> 这两个方法都不允许返回的行数有确定的界限。

例子：

```
SELECT *
FROM nation TABLESAMPLE BERNOULLI(25);

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

与 join 结合使用：

```
SELECT o.*, i.*
FROM orders o TABLESAMPLE SYSTEM (10)
JOIN lineitem i TABLESAMPLE BERNOULLI (40)
  ON o.o_orderkey = i.l_orderkey;
```

UNNEST

UNNEST可用于将ARRAY或MAP展开为关系。ARRAY 展开为单列，MAP 展开为两列（键、值）。UNNEST还可以与多个参数一起使用，在这种情况下，它们将展开为多个列，行数与最高基数参数相同（其他列用空值填充）。UNNEST可以有WITH ORDINALITY子句，在这种情况下，会在末尾添加一个附加的有序列。UNNEST通常与JOIN一起使用，可以引用JOIN左侧关系中的列。

```
SELECT numbers, animals, n, a
FROM (
  VALUES
    (ARRAY[2, 5], ARRAY['dog', 'cat', 'bird']),
```

```
    (ARRAY[7, 8, 9], ARRAY['cow', 'pig'])
) AS x (numbers, animals)
CROSS JOIN UNNEST(numbers, animals) AS t (n, a);
```

```
+-----------+-----------------+------+------+
|  numbers  |     animals     |  n   |  a   |
+-----------+-----------------+------+------+
| [2, 5]    | [dog, cat, bird] |    2 | dog  |
| [2, 5]    | [dog, cat, bird] |    5 | cat  |
| [2, 5]    | [dog, cat, bird] | NULL | bird |
| [7, 8, 9] | [cow, pig]       |    7 | cow  |
| [7, 8, 9] | [cow, pig]       |    8 | pig  |
| [7, 8, 9] | [cow, pig]       |    9 | NULL |
+-----------+-----------------+------+------+
(6 rows in set)
```

有序列:

```
SELECT numbers, n, a
FROM (
  VALUES
    (ARRAY[2, 5]),
    (ARRAY[7, 8, 9])
) AS x (numbers)
CROSS JOIN UNNEST(numbers) WITH ORDINALITY AS t (n, a);
```

```
+-----------+---+---+
|  numbers  | n | a |
+-----------+---+---+
| [2, 5]    | 2 | 1 |
| [2, 5]    | 5 | 2 |
| [7, 8, 9] | 7 | 1 |
| [7, 8, 9] | 8 | 2 |
| [7, 8, 9] | 9 | 3 |
+-----------+---+---+
(5 rows in set)
```


JOIN

JOIN 允许组合来自多个关系的数据。

- CROSS JOIN

CROSS JOIN返回两个关系的笛卡尔积（所有组合）。

下面两个查询等价:

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

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

如果一个表有 25 行，另一个表有 5 行，使用CROSS JOIN将返回 125 行结果。

- LATERAL

FROM子句中出现的子查询前面可以有关键字 lateral。这允许它们引用前面FROM提供的列。

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

- 别名

如果两个关系有同名列需要使用别名。

```
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;
```

- 子查询

子查询是由查询组成的表达式。当子查询引用子查询之外的列时，它是相关的。从逻辑上讲，将为周围查询中的每一行计算子查询。因此，在子查询的任何单个计算期间，引用的列都将是常量。

> 并非所有标准的子查询都支持。

- EXISTS子查询

返回子查询存在的行。

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

- IN子查询

返回子查询中存在的行，子查询必须是单列。

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

- 标准子查询

```
SELECT name
FROM nation
WHERE regionkey = (SELECT max(regionkey) FROM region)
```

PREPARE

支持对 limit，OFFSET，FETCH FIRST 中使用变量。

```
-- offset limit 样例
PREPARE my_sql FROM
SELECT * FROM item offset ? limit ?;
EXECUTE my_sql USING 1, 3;
-- offset FETCH 样例
PREPARE my_sql FROM
SELECT * FROM item OFFSET ? ROWS FETCH NEXT ? ROWS ONLY;
EXECUTE my_sql USING 1, 3;
```

支持在create schema中绑定变量。

```
PREPARE my_createschema FROM
CREATE SCHEMA hubbleap.tpcds WITH ( LOCATION = ? );
EXECUTE my_createschema USING '/hubble/data/tpcds';
```

查看表的分区：

```
SELECT * FROM hubble.default."testfq01$partitions"
```

1.2.1.4 其他

1.2.1.4.1 ANALYZE

规则

```
ANALYZE table_name
```

描述

对表进行预分析，统计数据表的记录数，最大值、最小值等统计信息。

示例：

```
analyze order;
```

1.2.1.4.2 SHOW STATS

规则

```
SHOW STATS FOR (SQL_STATEMENT)
```

描述

支持对任意查询语句的统计信息支持。

示例：

```
SHOW STATS FOR (
  SELECT i_current_price,i_brand_id,i_brand
  FROM ITEM
);

SHOW STATS FOR (
  SELECT * FROM
    web_sales,
    date_dim
  WHERE
    ws_ship_date_sk = d_date_sk
);
```

1.2.1.4.3 SET TIME ZONE

规则

```
SET TIME ZONE [ZONE]
```

描述

可在会话属性中设置timezone。需要注意，如果全局配置了sql.forced-session-time-zone属性，则在会话属性中自定义tiemzone将无法生效。

示例：

```
SET TIME ZONE LOCAL;
SET TIME ZONE '-06:00';
SET TIME ZONE 'Asia/Shanghai';
-- 查看当前timezone
select current_timezone();
```

1.2.1.4.4 SET SESSION

规则

```
SET SESSION name = expression;
SET SESSION catalog.name = expression;
```

描述

设置会话属性值或 catalog 会话属性。

示例：

```
SET SESSION task_concurrency=16
```

1.2.1.4.5 SET PATH

规则

```
SET PATH catalog.schema
```

描述

设置会话路径，指定路径必须存在。

示例:

```
SET PATH hubble.Demo1;
```

1.2.1.4.6 RESET SESSION

规则

```
RESET SESSION name
```

描述

将会话属性值重置为默认值。

示例:

```
RESET SESSION task_concurrency
```

1.2.1.4.7 DESCRIBE INPUT

规则

```
DESCRIBE INPUT name
```

描述

列出预准备语句的输入参数以及每个参数的位置和类型。不确定的类型会显示unknown。

示例:

准备带有参数的查询

```
PREPARE mytestdesc1
FROM
SELECT name FROM tab_1 where id=? and pid=?
```

列出信息

```
DESCRIBE INPUT mytestdesc1

 Position | Type
----------+------
 0        | unknown
 1        | int
 2        | varchar
```

准备不带有参数的查询

```
PREPARE mytestdesc2 FROM SELECT * FROM tab_1
```

列出信息

```
DESCRIBE INPUT mytestdesc2

 Position | Type
----------+------
```

1.2.1.4.8 DESCRIBE OUTPUT

规则

```
DESCRIBE INPUT name
```

描述

列出预准备语句的输出参数列包括列名、数据源、库名、表、字段类型、类型大小以及是否具有布尔值。

示例:

准备带有输出列的查询

```
PREPARE mytestdesc3 FROM SELECT name FROM tab_1
```

列出信息

```
DESCRIBE OUTPUT mytestdesc3

 Column Name | Catalog | Schema | Table |    Type      | Type Size | Aliased
-------------+---------+--------+-------+--------------+-----------+---------
 id          | hubble  | zl     | p1    | varchar(32)  |        10 | false
 name        | hubble  | zl     | p1    | varchar(30)  |        18 | false
 pid         | hubble  | zl     | p1    | varchar(32)  |        10 | false
```

1.2.1.4.9 DEALLOCATE PREPARE

规则

```
DEALLOCATE PREPARE name
```

描述

从准备列表里删除具有该名称的语句。

示例:

准备名为mytestdesc4的查询

```
PREPARE mytestdesc4 FROM SELECT name FROM tab_1
```

删除mytestdesc4

```
DEALLOCATE PREPARE mytestdesc4
```

1.2.2 安全

1.2.2.1 审计日志

传统的审计日志，通常只记录下用户的执行语句、时间点、具体用户、访问源。而考虑到持续优化的运维需求，Hubble 的审计日志除了传统内容外，还增加了许多对资源及实际任务执行状态的信息，当然这些信息都是查询任务完成后信息。用户需要考虑到审计日志功能，都会对 master 节点的 cpu 与内存增加消耗，但考虑到审计日志的信息对于用户的持续优化是长期有益的，并且分析场景并不需要对毫秒级的性能损耗过渡顾虑，建议性能损耗可控的前提下将其开启。

默认的审计日志配置文件为 conf 目录下的 event-listener.properties，也可以在 conf/hubble-site.xml 中配置指定审计日志配置文件。

```
<property>
    <name>event-listener.config-files</name>
    <value>/data/hubble-5.4.1/conf/audit-log.properties</value>
</property>
```

具体配置内容如下：

```
event-listener.name=audit-log

# 审计日志文件输出的绝对路径（请不要使用相对路径）
audit-log.path=/data/hubble-5.4.1/logs/audit

# 审计日志限制最大文件size，默认为100MB，当日志大小超过限制后，将会自动将其gz压
    ↪ 缩保留在同路径下
audit-log.max-size=64MB

# 审计最多保留文件个数，默认为30
audit-log.max-history=60

# 审计日志监听策略文件，支持对用户，SQL操作类型和表名进行规则配置
file.audit-file=/data/hubble-server-5.4.1/conf/audit_rule.properties

# 审计日志策略的动态刷新时间
file.refresh-period=5s
```

审计策略文件配置

```
# 监听的用户，多个用户使用逗号隔开
audit-log.users=abc,sky

# 监听的全限定表名，多个表名使用逗号隔开
audit-log.tables=hubble.db_test.student,hubble.db_test.account

# 监听的SQL操作类型，多个操作类型使用逗号隔开
```

```
# 操作类型: SELECT|EXPLAIN|DESCRIBE|INSERT|UPDATE|DELETE|ANALYZE|DATA_DEFINITION
    ↪ |ALTER_TABLE_EXECUTE|MERGE
audit-log.optType=insert,select
```

1.3 连接使用操作

1.3.1 交互方式

Hubble 的交互方式有命令行方式和 jdbc 方式两种。此外，还可以在 web 端使用 Hubble 数据库管理系统的 SQL 编辑器对 hubble 数据库进行操作。

1.3.1.1 命令行方式

在hubble的bin目录下，脚本程序hubble-sql.sh启动 cli。

有用的参数有:

```
--catalog <catalog>              进入hubble命令行的默认catalog
--schema <database>             进入hubble命令行的默认database
--execute <SQL>                      使用-execute后边跟SQL进行查询，用引号将需要执
    ↪ 行的SQL包起来
-f <file>                        使用-f后边跟文件进行查询
--server <server>                hubble提供服务的地址
--user <user>                    hubble的用户名
--password <password>            hubble的密码
--hubvar<name>                   外部输入参数名称
--output-format <output-format> 执行查询输出的格式，包括ALIGNED，VERTICAL，JSON，
    ↪  CSV，TSV,CSV_HEADER，CSV_UNQUOTED,CSV_HEADER_UNQUOTED等格式，默认CSV格式
    ↪ 输出
```

例如

1. 在命令行下执行查询:

```
bin/hubble-sql.sh --server=hubble01:30008 --user=hubble
hubble-cli> show catalogs;
```

```
+---------+
| Catalog |
+---------+
| hubble  |
| system  |
+---------+
(4 rows in set)

2.47 sec
```

2. 使用--execute 参数查询:

```
bin/hubble-sql.sh --server=hubble01:30008 --user=hubble --execute "show catalogs
    ↪ "
```

```
"hubble"
"system"
```

3. 使用-f 参数查询:

新建文本 showcata.sql

内容show catalogs

```
bin/hubble-sql.sh --server=hubble01:30008 --user=hubble -f showcata.sql
"hubble"
"system"
```

4. 使用-hubvar 参数查询:

使用语法:

```
bin/hubble-sql.sh --server= ip+端口 --user=用户 --catalog=catalog数据源 --
    ↪ password
--execute"slect * from tablename where column='\${hubvar:参数名称}' "
--hubvar "参数名称=xx"
```

数据样例

```
 f1 | f2 | f3
----+----+----
 l  | 18 | 男
 c  | 20 | 男
 w  | 22 | 女
```

样例: 查询表 t12 的内容 f1 字段判断条件通过 hubvar 传递

```
[hubble@hubble01 bin]$./hubble-sql.sh --server=https://hubble01.hubbledb.cn:7077
    ↪  --user=hubble  --catalog=hubble --password
--execute "select * from hubble.test.t12 where f1 ='\${hubvar:a}'" --hubvar "a=l
    ↪ ";

--返回查询结果
"l","18","男 "
```

> **注意:** 使用命令行传参时需要加转义符号 并且 ''单引号不是 ${hubvar:name} 中参数具体看字段类型

样例: 使用多个传递参数

```
[hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
    ↪ :7077 --user=hubble  --
catalog=hubble --password  --execute "select * from hubble.test.t12 where \${
    ↪ hubvar:a} ='\${hubvar:b}'
" --hubvar "a=f1" --hubvar "b=l" ;
--返回查询结果
```

```
"1","18","男 "
```

-f 命令执行 sql 文件并使用 hubvar 传递参数

使用语法:

```
#client参数配置
bin/hubble-sql.sh --server= ip+端口 --user=用户 --catalog=catalog数据源 --
    ↪ password
-f 调用sql文件名称
--hubvar "参数设置名称=xx"
#sql文件中配置
select * from tablename where cloumn ='${hubvar:xxx}'
```

样例: 使用-f 命令查询 a.sql 并使用 hubvar 传递文件中参数

创建 a.sql 文件

```
select * from hubble.test.t12 where f1 ='${hubvar:a}' and f2=${hubvar:b};
```

> **注意:** sql 文件调用时不需要加转义符

执行命令调用 a.sql

```
[hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
    ↪ :7077 --user=hubble  --catalog=hubble --password  -f a.sql --hubvar "a=c"
    ↪  --hubvar "b=20";

--返回查询结果
"c","20","男 "
```

5. 用户密码认证

```
[hubble@hubble01 hubble-5.2.1]$ export HUBBLE_PASSWORD=hubble; bin/hubble-sql.sh
    ↪  --server=https://hubble01.hubbledb.cn:33484    --user=hubble --password
Password:
hubble> show schemas in hubble;
       Schema
--------------------
 default
 information_schema
(2 rows)

Query 20230329_060232_00000_rnitj, FINISHED, 5 nodes
Splits: 87 total, 87 done (100.00%)
4.01 [2 rows, 35B] [0 rows/s, 9B/s]

hubble> create schema hubble.zl;
```

注意：

用户密码认证使用 https 方式访问时，server 中要指定 master 域名

1.3.1.2 JDBC 方式-无认证

使用 Hubble JDBC 驱动提供 JDBC 方式访问 Hubble。

包括两种：写 JDBC 程序，导入 JDBC 驱动来访问。使用dbeaver等通用数据库端工具的方式进行访问。

1.3.1.2.1 代码方式

java 方式

url 可以是下列中的任一种：

```
jdbc:hubble://host:port
jdbc:hubble://host:port/<catalog>
jdbc:hubble://host:port/<catalog>/<database>
```

catalog及shema为可选项。可以限定 jdbc 程序默认连接的catalog和database。

在项目中导入 jdbc 驱动包，并创建下面的类：

```
package com.beagledata.hubble.hubblejdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JdbcTest {

    private static final String JDBC_URL = "jdbc:hubble://master:30008/hubble/
        ↪ tpcds";
    private static final String username = "hadoop";

    public static void main(String[] args) {
        try {
            Class.forName("com.beagledata.hubble.jdbc.HubbleDriver");

            try(Connection conn = DriverManager.getConnection(JDBC_URL, username
                ↪ , null);
                Statement stmt = conn.createStatement()){
                ResultSet res = stmt.executeQuery("select * from tpcds.lineitem
                    ↪  limit 20");
                int col = res.getMetaData().getColumnCount();
                while (res.next()) {
                    for (int i = 1; i <= col; i++) {
```

```
                    System.out.print(res.getString(i) + "\t");
                    if ((i == 2) && (res.getString(i).length() < 8)) {
                        System.out.print("\t");
                    }
                }
                System.out.println("");
            }
        }
    } catch (ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    }
  }
}
```

运行，即可查询出 tpcds 库下的 lineitem 表中的数据。

使用 hubvar 调用参数

```
package com.zendesk.maxwell.util;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

import com.beagledata.hubble.jdbc.HubbleDriver;



public class HubbleDemo {
    public static void main(String[] args) {
        String sql="select * from hubble.default.t7 where nmae='${hubvar
            ↪ :nmae}'";
        try {
            testconnection1(sql);
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    public static void testconnection1(String sql) throws
        ↪ ClassNotFoundException, SQLException {
    Class.forName("com.beagledata.hubble.jdbc.HubbleDriver");
```

```
                Properties pro = new Properties();
                String url= "jdbc:hubble://192.168.100.123:7077";
                HubbleDriver hubbleDriver =new HubbleDriver();
                pro.put("user", "hubble");
                pro.put("hubvar:nmae", "a3");
                Connection conn=hubbleDriver.connect(url, pro);

                Statement stmt = conn.createStatement();


                ResultSet re=stmt.executeQuery(sql);
                while(re.next()) {
                        System.out.println(re.getString(1));
                }

                stmt.close();
                conn.close();
        }

}
```

python 方式

导入的 Hubble 包需要找相关工作人员获取

```
# coding:utf8
import hubble

conn = hubble.dbapi.connect(
    host='192.168.100.134',#主机IP
    port=20088,#数据库端口为启动端口
    user='hubble',#用户
    catalog='hubble',#指定数据源
    schema='test'# 库名
)


cur = conn.cursor()
# -----------------------  sql 查询 --------------------------------
sl_sql="""select * from hubble.test.t_test2 limit 1 """
cur.execute(sl_sql)
#获取每一行,可选其他参数:fetchone获取一行
rows = cur.fetchall()

# 显示每列的详细信息
des = cur.description
```

```
print("表的描述:", des)
# 获取表头
print("表头:", ",".join([item[0] for item in des]))
print(rows)
conn.close()
cur.close()
```

插入数据

```
import hubble
from  hubble import transaction
with hubble.dbapi.connect(
    host='192.168.100.134',#主机ip
    port=20088,#数据库端口
    user='hubble',#用户名
    catalog='hubble',#数据源
    schema='test',#库名
    isolation_level=transaction.IsolationLevel.REPEATABLE_READ,#开启事物
) as conn:
    cur = conn.cursor()
#-------------------------------------insertsql
    ↪ ------------------------------------
    ins_sql="""INSERT INTO hubble.test.t_test2 VALUES (1, '2', '3')"""
    cur.execute(ins_sql)
    cur.fetchall()
#释放资源
    caonn.close()
    cur.close()
```

1.3.1.2.2 通用客户端方式

1. 使用DBeaver进行连接。

下载地址：DBeaver

打开 DBeaver 后，在导航栏找到数据库-驱动管理器。

选择新建

填写驱动名称，类型选择 Generic, 类名填写com.beagledata.hubble.jdbc.HubbleDriver。

添加文件添加 Hubble 的 JDBC 驱动程序。

点击OK。

接下来进行配置连接，文件-新建。

选择数据库连接，并点击Next按钮。

选择上面配的驱动，点击Next按钮。

填写上正确的信息后，点击finish。

连接后就可以进行使用了。

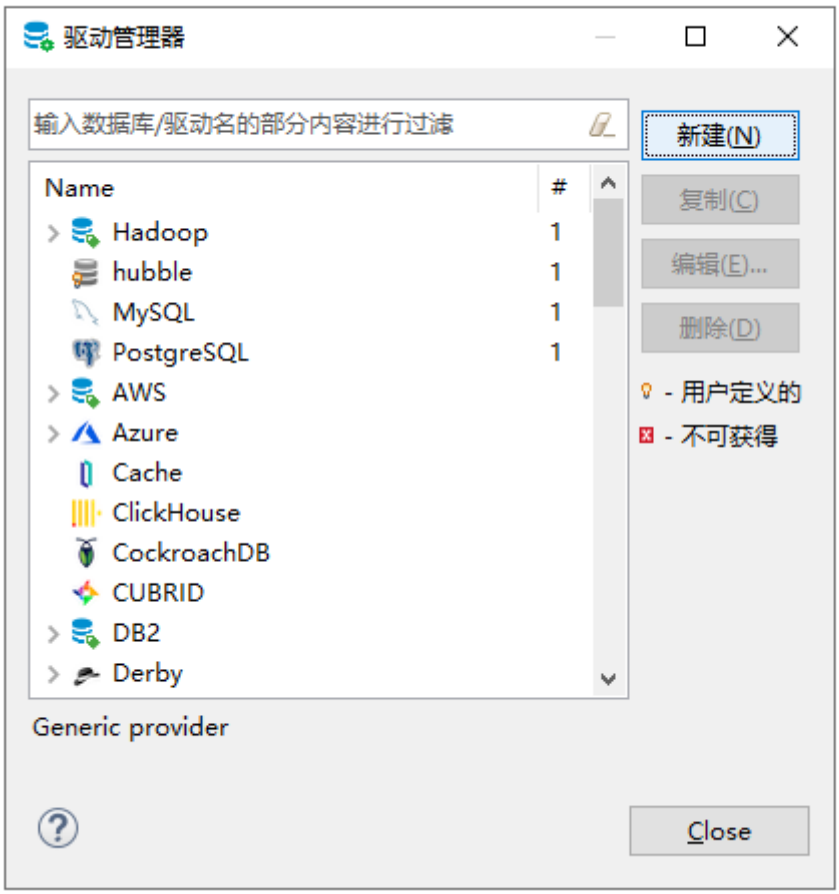
驱动管理器
输入数据库/驱动名的部分内容进行过滤
新建(N)
复制(C)
编辑(E)...
删除(D)
Name
#
Hadoop 1
hubble 1
MySQL 1
PostgreSQL 1
AWS
Azure
Cache
ClickHouse
CockroachDB
CUBRID
DB2
Derby
- 用户定义的
- 不可获得
Generic provider
Close

图 1: driver

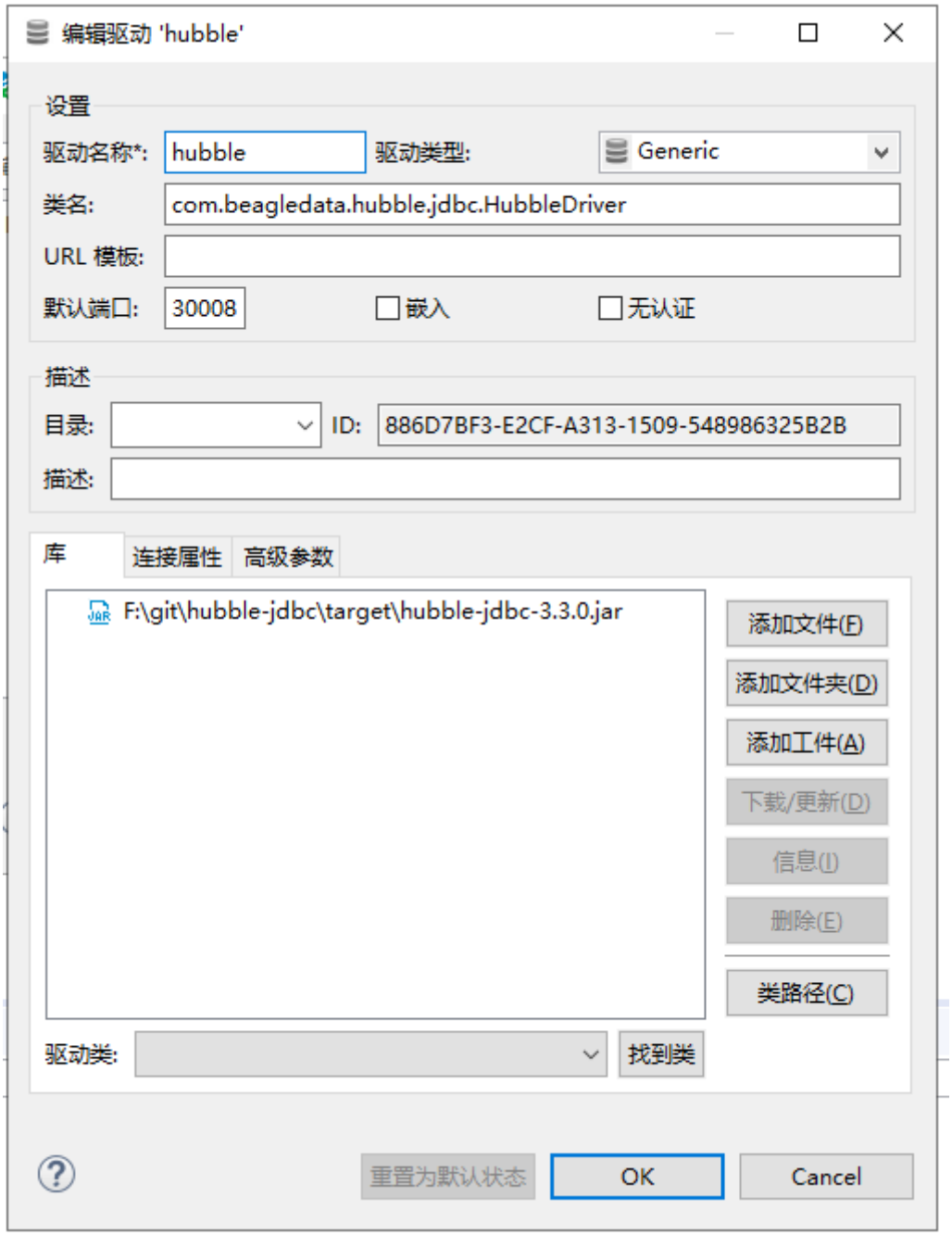
编辑驱动 'hubble'
设置
驱动名称*: hubble
驱动类型: Generic
类名: com.beagledata.hubble.jdbc.HubbleDriver
URL 模板:
默认端口: 30008
嵌入
无认证
描述
目录:
ID: 886D7BF3-E2CF-A313-1509-548986325B2B
描述:
库
连接属性
高级参数
F:\git\hubble-jdbc\target\hubble-jdbc-3.3.0.jar
添加文件(F)
添加文件夹(D)
添加工件(A)
下载/更新(D)
信息(I)
删除(E)
类路径(C)
驱动类:
找到类
重置为默认状态
OK
Cancel

图 2: jdbc

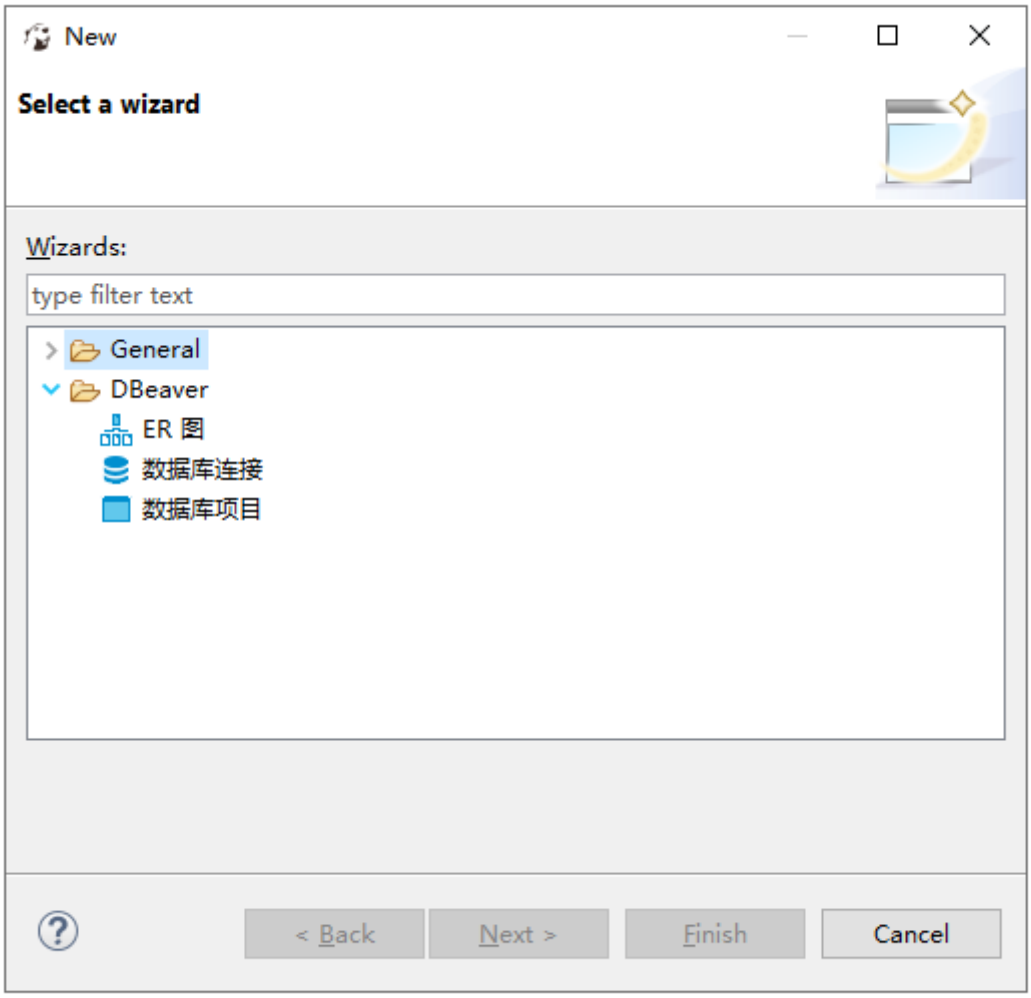
New
Select a wizard
Wizards:
type filter text
General
DBeaver
ER 图
数据库连接
数据库项目
< Back
Next >
Finish
Cancel

图 3: new

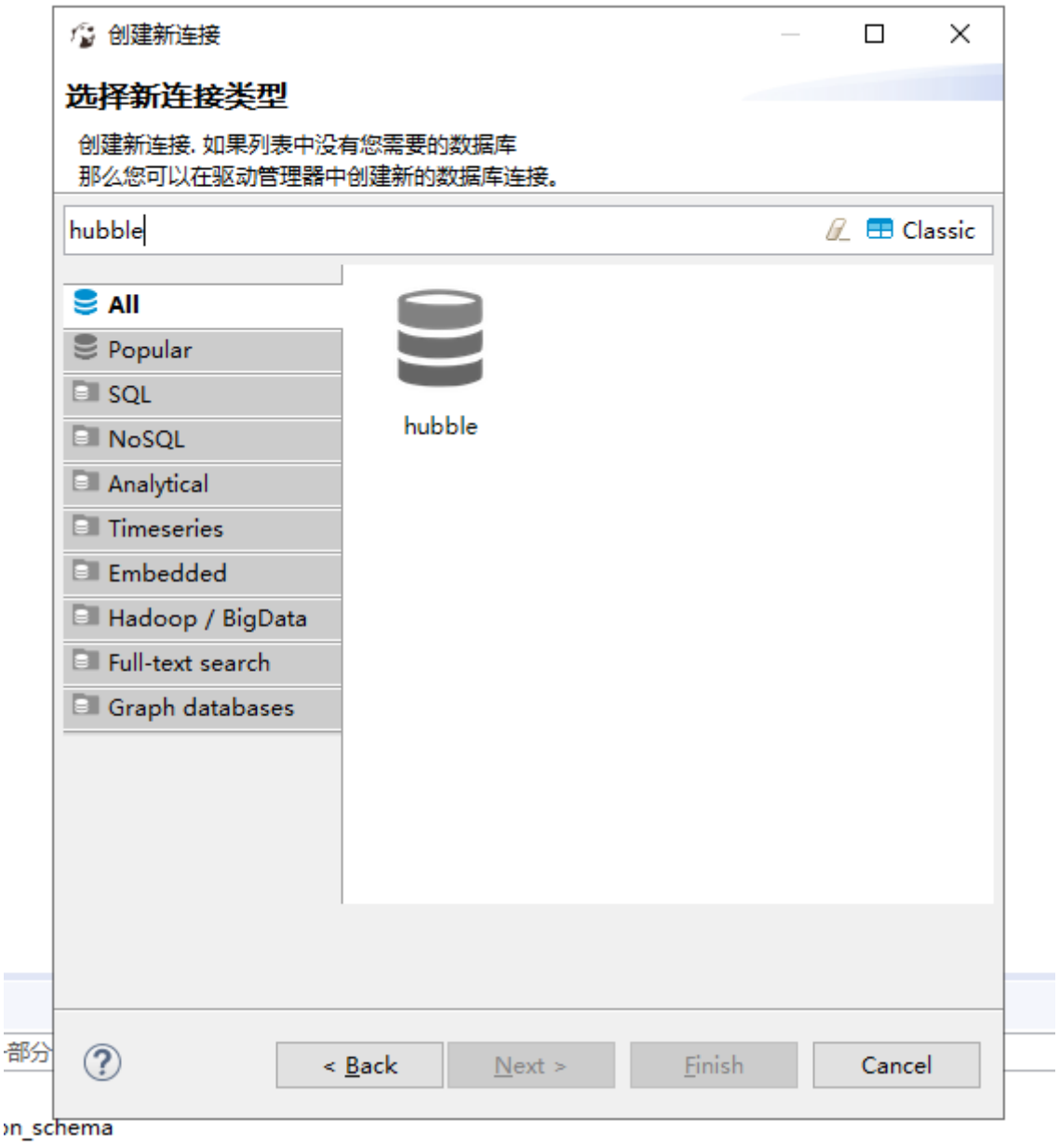

图 4: newhubble

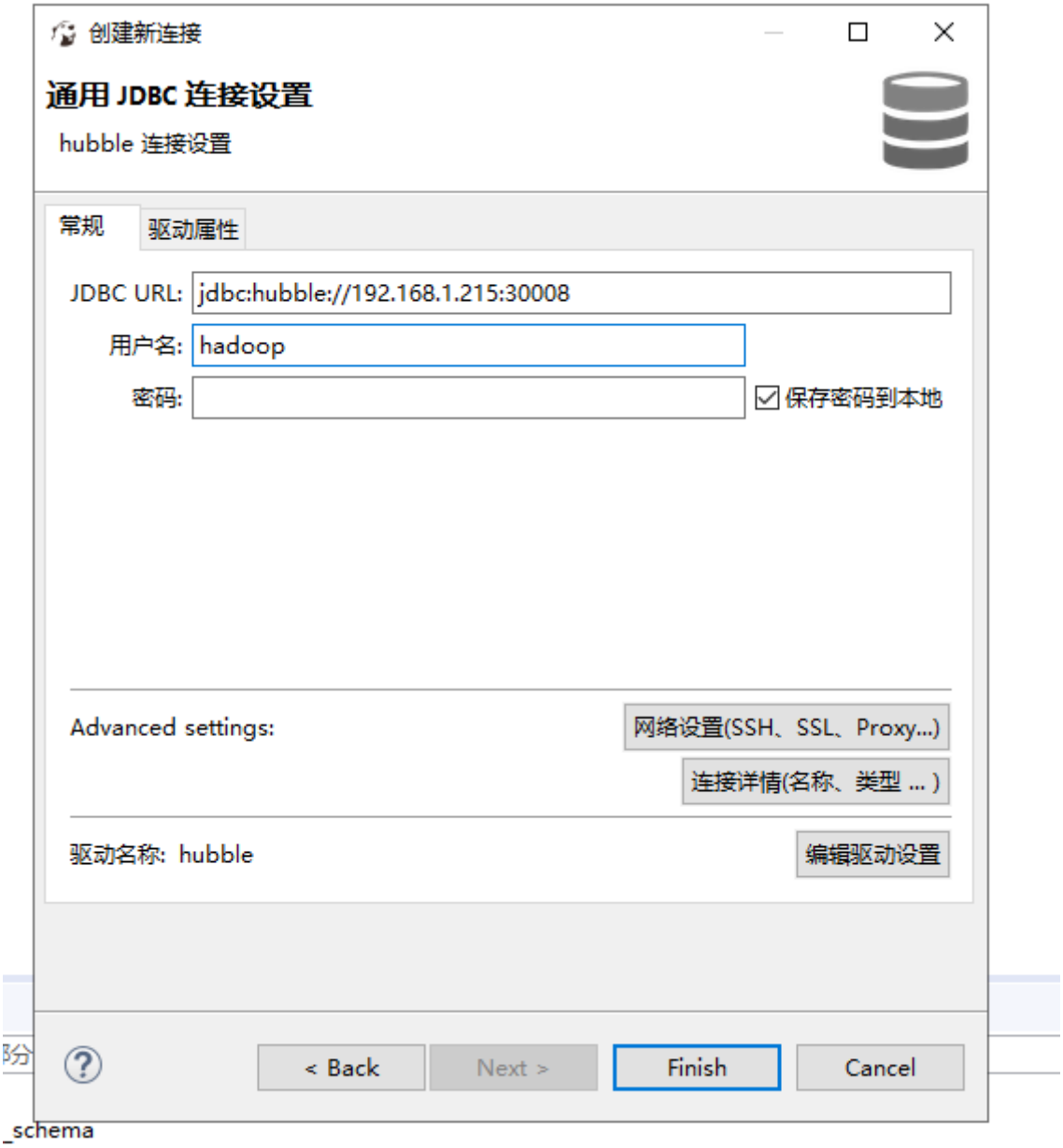
创建新连接
通用 JDBC 连接设置
hubble 连接设置
常规
驱动属性
JDBC URL:
jdbc:hubble://192.168.1.215:30008
用户名:
hadoop
密码:
保存密码到本地
Advanced settings:
网络设置(SSH、SSL、Proxy...)
连接详情(名称、类型 ...)
驱动名称: hubble
编辑驱动设置
< Back
Next >
Finish
Cancel
_schema

图 5: config

1.3.1.3 JDBC 方式-有认证

1.3.1.3.1 代码方式

java 代码

```
package com.beagledata.hubble.jdbc;


import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JdbcTest {

    private static final String JDBC_URL = "jdbc:hubble://master:30008/hubble/
        ↪ tpcds?SSL=true&SSLVerification=NONE";
    private static final String username = "hubble";
    private static final String password = "hubble";
    public static void main(String[] args) {
        try {
            Class.forName("com.beagledata.hubble.jdbc.HubbleDriver");

            try(Connection conn = DriverManager.getConnection(JDBC_URL, username
                ↪ , password);
                Statement stmt = conn.createStatement()){
                ResultSet res = stmt.executeQuery("select * from tpcds.lineitem
                    ↪ limit 20");
                int col = res.getMetaData().getColumnCount();
                while (res.next()) {
                    for (int i = 1; i <= col; i++) {
                        System.out.print(res.getString(i) + "\t");
                        if ((i == 2) && (res.getString(i).length() < 8)) {
                            System.out.print("\t");
                        }
                    }
                    System.out.println("");
                }
            }
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

python 代码

```
from hubble.dbapi import connect
from hubble.auth import BasicAuthentication
import urllib3

# 用户名和密码
username = "hubble"
password = "hubble"

# 禁用 urllib3 警告
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

# 连接到 hubble
conn = connect(
    host="192.168.100.123",
    port="7077",
    user=username,
    auth=BasicAuthentication(username, password),
    http_scheme="https",
    verify=None
)
cur = conn.cursor()
#查询sql
sl_sql="""SELECT * FROM hubble.zh_wh_odsfff.t12"""
# 执行查询
result = cur.execute(sl_sql)

# 打印查询结果
for row in result:
    print(row)
```


> **注意:**
>
> 必须添加 SSL=true&SSLVerification=NONE

1.3.1.3.2 通用客户端

在 jdbc 参数后添加证书认证

```
jdbc:hubble://192.168.100.123:33484/hubble/zl?SSL=true&SSLVerification=NONE
```

> **注意:**
>
> 必须添加 SSL=true&SSLVerification=NONE

1.3.1.4 web 端登录

Hubble 数据库管理系统集成了 SQL 编辑器。

使用帐号密码登录:

图 6: db_manage

1.3.1.5 SQL 编辑器

SQL 编辑器可以方便数据库开发人员在 WEB 网页端使用 SQL 语言便捷的创建和编辑 SQL 文本，运行已选择的查询。

在左侧集群预览模块，可以查看当前用户有权限操作的库表、视图。当库表较多时，可以在库 `schema` 和表 `table` 两个级别进行过滤筛选，以快速的找到目标表。

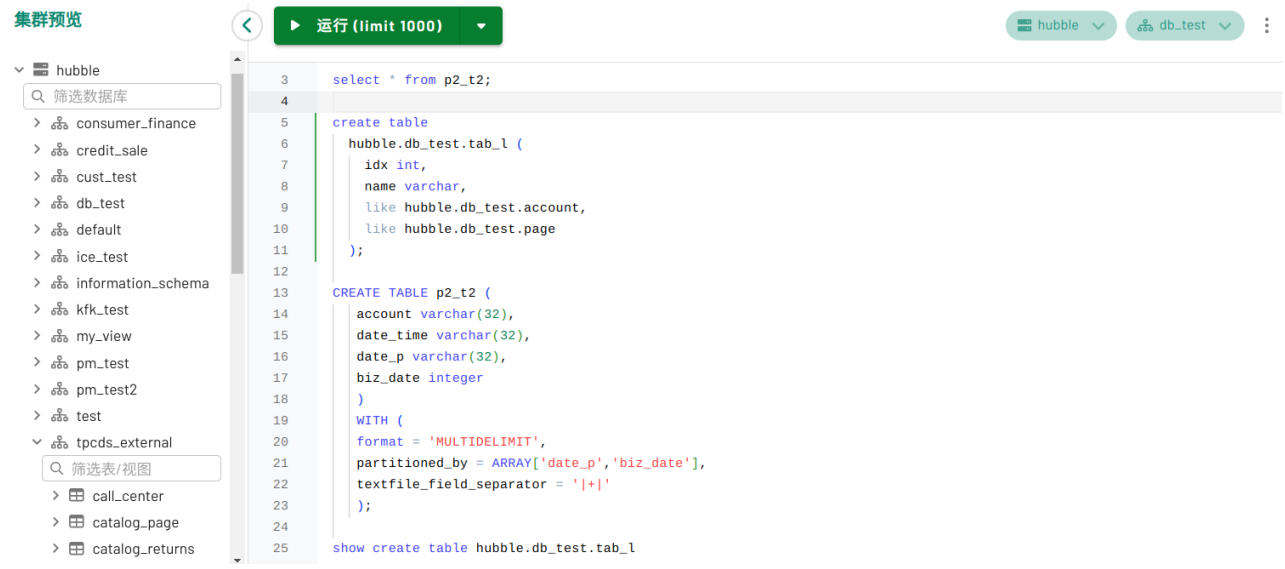

图 7: sql_editer

在右侧的 SQL 面板中，可以选中多个 SQL 一起执行。不同 SQL 的执行结果可以在下面的窗口 TAB 页中切换查看。窗口中还记录了 SQL 的执行时间和读写速度。

在 SQL 面板的右上角可以选择默认的目录和库，还可以对选定区域进行 SQL 格式化。

1.4 权限设置

Hubble 数据库的用户权限支持两种配置策略，一种是使用配置文件在后台进行用户和权限的配置，另一种是在 Hubble 数据库管理系统上使用基于 RBAC 的角色权限控制。

1.4.1 基于文件的用户权限

用户权限

Hubble 用户密码采用 htpasswd 用于创建、更新储存用户名和密码文件。

1.4.1.1 用户添加

进入 conf/目录下执行

```
htpasswd -B -C 10 hubble_password user2
```

```
[hubble@hubble01 conf]$ htpasswd -B -C 10 hubble_password user2
New password:
Re-type new password:
Adding password for user user2
[hubble@hubble01 conf]$ cat hubble_password
hubble:$2y$10$MTQ6cSV.yw83/0G1XyTRHOWTB7Fw9EbTb3oeIlMqb2RKNJ.X9a9q6
zhaol:$2y$10$vg26N3hypG9S2Qmq.n6TK.gLb8LGeIZT2QZAht5BhosaxoAXUvJpq
user2:$2y$10$YpVD3wiuUZXxJVX8PqHJ9ehn.J114/0bz9o50Me33t9bUpLV0cSZ6
[hubble@hubble01 conf]$
```

> **注意：**
>
> 修改密码执行上面语句，则可以直接更新密码
>
> 删除用户，执行删除 hubble_password 中对应的的记录

1.4.1.2 给用户授权

授权统一使用 conf/hubble_access.json 进行权限管控，默认管控权限由 hubble 账户来执行。

```
{
  "schemas": [
    {
      "user": "hubble",
      "catalog":".*",
      "schema": ".*",
      "owner": true
    },
    {
      "user": "zhaol",
      "catalog":"(hubble|system)",
      "schema": "zl",
      "owner": true
    },
```

```
    {
      "user": "(user1|user2|user3)",
      "catalog":"(hubble|system)",
      "schema": "zl2",
      "owner": true
    }
  ],
  "tables": [
    {
      "user": "hubble",
      "catalog":".*",
      "schema": ".*",
      "table": ".*",
      "privileges": ["SELECT","INSERT"," DELETE", "UPDATE", "OWNERSHIP","
         ↪ GRANT_SELECT"]
    },
    {
      "user": "(user1|user2|user3)",
      "catalog":"hubble",
      "schema": "zl2",
      "table": ".*",
      "privileges": ["SELECT", "INSERT", "DELETE","UPDATE", "OWNERSHIP","
         ↪ GRANT_SELECT"]
    },
    {
      "user": "zhaol",
       "catalog":"hubble",
      "schema": "(zl|zl3)",
      "table": ".*",
      "privileges": ["SELECT","INSERT"," DELETE", "UPDATE", "OWNERSHIP","
         ↪ GRANT_SELECT"]
    },
     {
      "user": "(zhaol|user1|user2|user3)",
       "catalog":"system",
      "schema": ".*",
      "table": ".*",
      "privileges": ["SELECT"]
    }
  ]
}
```

授权参考 user 中的配置 user1 或者 zhaol

1.4.2 基于 RBAC 的用户权限

Hubble 支持基于 RBAC 的角色权限控制。

使用管理员用户可以在平台上创建用户和角色，为角色分配库表权限。

1.4.2.1 创建用户

图 8: user_create.png

创建用户时，密码需要符合规则，如包含大小写字母和数字等。

管理员可以在用户详情页面为用户修改密码，也可以在用户列表重置密码，解除锁定状态。

普通用户在登录后，可以在个人中心修改自己的密码。

个人中心，这里可以查看当前用户的所属角色和所属组。

注意，用户的密码是有时效性的，有效期内可以正常登录，过了有效期则会被锁定。在个人中心可以查看当前密码的有效期，当临近有效期时，还会有临期提醒。用户可以在这里更新自己的密码。

1.4.2.2 创建角色

管理员可以创建角色，输入角色名称和描述，即可添加角色到系统中。

> 使用 hubble 的角色权限访问控制功能，需要在 hubble 的配置文件中，配置属性 `hubble.access-control.enabled` 为 `true` 。

为角色分配用户

为角色分配一个用户：

可以在下拉列表中选择一个已有用户，也可输入一个用户名。当用户名不存在时会默认创建一个新用户。


个人中心
基本信息
hubble 您的密码过期日是2024-05-02 08:00。
归属角色
public
sysadmin
归属组
develop
修改密码
当前密码 *
新密码 *
必须包含小写字母、数字
确认新密码 *
更改密码

图 9: user_set

图 10: role_create.png

图 11: role_add_user.png

1.4.2.3 角色赋权

在角色详情页面可以为角色添加权限。

图 12: role_grant_table.png

可以为角色分配目录 catalog 、数据库 schema 、表 table 和字段 column 级别的不同权限，可分配权限包括：修改 Alter | 创建 Create | 删除 Delete | 销毁 Drop | 插入 Insert | 刷新 Refresh | 查询 Select | 显示 Show | 更新 Update等。

> 详细的使用说明请参考 《Hubble 数据库管理系统操作手册》

1.4.3 资源管理

```
{
  "rootGroups": [
    {
      "name": "hubble_g",
      "softMemoryLimit": "80%",
      "maxQueued": 100,
      "hardConcurrencyLimit": 20,
      "schedulingPolicy": "query_priority",
```

```
      "jmxExport": true
    },
    {
      "name": "zhaol_g",
      "softMemoryLimit": "30%",
      "maxQueued": 5,
      "hardConcurrencyLimit": 2,
      "schedulingPolicy": "query_priority",
      "jmxExport": true
    },
    {
      "name": "zzfx",
      "softMemoryLimit": "30%",
      "maxQueued": 1000,
      "hardConcurrencyLimit": 50,
      "schedulingPolicy": "query_priority",
      "jmxExport": true
    },
    {
      "name": "any",
      "softMemoryLimit": "20%",
      "maxQueued": 1000,
      "hardConcurrencyLimit": 50,
      "schedulingPolicy": "query_priority",
      "jmxExport": true
    }
  ],
  "selectors": [
    {
      "user": "hubble",
      "group": "hubble_g"
    },
    {
      "user": "zhaol",
      "group": "zhaol_g"
    },
    {
      "user": "user1",
      "group": "zzfx"
    },
    {
      "group": "any"
    }
  ]
}
```

rootGroups 为资源控制，资源池可用的内存多少，以及任务并行度和队列

```
"name": "hubble_g",  --资源组名
"softMemoryLimit": "20%", --资源组可使用的最大分配内存
"maxQueued": 100,  --最大队列数
"hardConcurrencyLimit": 20,      --最多任务并行数
"schedulingPolicy": "query_priority", --优先级 保持默认
"jmxExport": true
```

selectors 为选择器规则指定用户可以访问什么资源

```
{
  "user": "hubble",  --用户
 "group": "hubble_g" --资源组名
}
```

{ ”group”: ”any” }

该选择器，用于未指定的用户使用

1.5 数据导出导入

规则

```
bin/hubble-sql.sh --server=https:hostname:port --user=username --password --
    ↪ catalog=catalogname --schema=schemaname --output-format <otput-format> --
    ↪ execute='query sql'  > filename
```

```
--server     <server>                hubble提供服务的地址

--user       <user>                  hubble的用户名

--password   <password>              hubble的密码

--catalog    <catalog>               指定数据源

--schema     <schema(database)>      指定数据源下的库

--session    <session>           指定seesion参数 (完整参数可通过show session查看)

--f,--file   <flie>          指定要执行的sql文件

--execute <execute>              执行sql进行查询 (需要执行的sql需要用''引号包起来)

--output-format <output-format>    执行查询输出的格式, 包括ALIGNED, VERTICAL,
    ↪ JSON, CSV, TSV等格式, (默认CSV格式输出)
--delimiter    <|+|>                 指定数据导出分隔符 支持多字符分隔符
```

```
--output-charset utf-8          指定数据编码方式 默认utf-8

--hubvar<name>                  外部输入参数名称
```

1.5.1 文件导出

例子

```
export HUBBLE_PASSWORD=hubble;export HUBBLE_PASSWORD=hubble; bin/hubble-sql.sh
    ↪ --server=https://poc-hubble01.hubbledb.cn:18080 --user=hubble --password
    ↪ --output-format CSV_UNQUOTED --execute='select * from hubble.zl.f2'  > f2.
    ↪ txt
```

```
export HUBBLE_PASSWORD=hubble; bin/hubble-sql.sh --server=https://poc-hubble01.
    ↪ hubbledb.cn:18080 --user=hubble --password --output-format TSV --execute='
    ↪ select * from hubble.zl.f2'  > /home/hubble/f3.txt
```

1.5.2 文件导入

规则

```
LOAD DATA [PATH]/LOCAL INPATH [pathlocation] IN TO TABLE table_name [PARTITTION]
```

描述

导入数据文件

例子

1.5.2.1 指定分区

```
load data local inpath '/opt/zl/d_data/v3.txt'  into table p2_t2 PARTITION(
    ↪ date_p='20230320',biz_date='3');
```

加载分区表需要手动刷新数据

```
call system.sync_partition_metadata(
schema_name => 'zl',
table_name => 'p2_t2',
mode => 'FULL',
case_sensitive => true
);
```

从本地导入无分区数据

```
load data local inpath '/data/lsc/test01.txt' into table hubble.default.
    ↪ lsctest01;
```

从 hubblestore 中导入无分区数据

```
load data inpath '/hubbledata/dt/test02.txt' into table hubble.default.lsctest01
    ↪ ;
```

默认从配置文件读取 hubblestore 链接地址

1.5.3 包含参数的文件导出示例

使用语法:

```
[hubble@hubble01 hubbleap-5.2.1]$ bin/hubble-sql.sh --server=hubbleip地址+端口
    ↪ --us
er=用户名  --catalog=数据源 --password --execute"执行sql" --output-format=输出格
    ↪ 式  --delimiter=分隔符 --output-charset=字符集编码  >输出文件名
```

--output-format

1. DELIMITER: 用于指定数据文件中字段之间的分隔符。当导出数据时，数据文件中的字段将使用该分隔符进行分割。例如，如果设置 DELIMITER 为逗号 (,)，则数据文件中的字段将使用逗号进行分隔。
2. DELIMITER_HEADER: 用于指定数据文件中包含列名的行在导出过程中的处理方式。加入这个参数时，表示数据文件的第一行是列名，并且不会被导入或导出。如果不设置这个参数，表示数据文件的第一行是数据，会被导入或导出。
3. DELIMITER_UNQUOTED: 将分隔符视为常规字符，即使它用引号括起来。这对于包含带引号的字符串的文件很有用，其中分隔符可以用作字符串的一部分。
4. DELIMITER_HEADER_UNQUOTED: 将标题行中的分隔符视为常规字符，即使它用引号引起来也是如此。这对于 CSV 文件很有用，其中分隔符通常用于分隔标题行中的列。

--delimiter ,

```
[hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
    ↪ :7077 --user=hubble  --
catalog=hubble --password  --output-format='DELIMITER_HEADER'  --delimiter=','
    ↪ --execute "select
 * from hubble.zh_wh_odsfff.t12"
"f1","f2","f3"
"l","18","男 "
"c","20","男 "
"w","22","女 "
```

不指定 --delimiter= 参数时默认分隔符为,

```
[hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
    ↪ :7077 --user=hubble  --
catalog=hubble --password  --output-format='DELIMITER_HEADER'   --execute "
    ↪ select
 * from hubble.zh_wh_odsfff.t12"
"f1","f2","f3"
```

```
"l","18","男 "
"c","20","男 "
"w","22","女 "
```

不指定--output-format=参数时默认为不带表头

```
 [hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
    ↪ :7077 --user=hubble --c
atalog=hubble --password    --execute "select * from hubble.zh_wh_odsfff.t12"  >
   ↪  n.csv
[hubble@hubble01 bin]$ cat n.csv
"l","18","男 "
"c","20","男 "
"w","22","女 "
```

特殊分隔符' |+| '

```
[hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
   ↪ :7077 --user=hubble --c
atalog=hubble --password --output-format=DELIMITER_HEADER  --delimiter='|+|'
   ↪ --execute "select *
from hubble.zh_wh_odsfff.t12"
"f1"|+|"f2"|+|"f3"
"l"|+|"18"|+|"男 "
"c"|+|"20"|+|"男 "
"w"|+|"22"|+|"女 "
```

特殊分隔符' | '

```
[hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
   ↪ :7077 --user=hubble  --
catalog=hubble --password  --output-format='DELIMITER_HEADER'  --delimiter=' | '
   ↪   --execute "select
 * from hubble.zh_wh_odsfff.t12"
"f1" | "f2" | "f3"
"l" | "18" | "男 "
"c" | "20" | "男 "
"w" | "22" | "女 "
```

特殊分隔符 '\x01|\01'

```
 [hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
    ↪ :7077 --user=hubble  --
catalog=hubble --password  --output-format='DELIMITER_HEADER'  --delimiter='\x01
   ↪ |\01'  --execute "s
elect * from hubble.zh_wh_odsfff.t12"
"f1"\x01|\01"f2"\x01|\01"f3"
"l"\x01|\01"18"\x01|\01"男 "
"c"\x01|\01"20"\x01|\01"男 "
```

```
"w"\x01|\01"22"\x01|\01"女 "
```

--output-charset=utf-8 or gbk 等等

写入文件

指定保留表头指定分隔符\x01|\01 输出结果到 b.csv 文件

```
[hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
    ↪ :7077 --user=hubble  --
catalog=hubble --password    --output-format=DELIMITER_HEADER  --delimiter='\x01
    ↪ |\01'  --execute "s
elect * from hubble.zh_wh_odsfff.t12"  > b.csv
[hubble@hubble01 bin]$ cat b.csv
"f1"\x01|\01"f2"\x01|\01"f3"
"l"\x01|\01"18"\x01|\01"男 "
"c"\x01|\01"20"\x01|\01"男 "
"w"\x01|\01"22"\x01|\01"女 "
```

output-format=DELIMITER_UNQUOTED，为不指定表头，并分隔符视为常规字符

```
[hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
    ↪ :7077 --user=hubble  --
catalog=hubble --password    --output-format=DELIMITER_UNQUOTED  --delimiter='\
    ↪ x01|\01'  --execute
"select * from hubble.zh_wh_odsfff.t12"  > d.csv
[hubble@hubble01 bin]$ cat d.csv
l\x01|\0118\x01|\01男
c\x01|\0120\x01|\01男
w\x01|\0122\x01|\01女
```

output-format=DELIMITER_HEADER_UNQUOTED，为指定表头，并分隔符视为常规字符

```
[hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn
    ↪ :7077 --user=hubble  --
catalog=hubble --password    --output-format=DELIMITER_HEADER_UNQUOTED  --
    ↪ delimiter='\x01|\01'  --e
xecute "select * from hubble.zh_wh_odsfff.t12"  > b.csv
[hubble@hubble01 bin]$ cat b.csv
f1\x01|\01f2\x01|\01f3
l\x01|\0118\x01|\01男
c\x01|\0120\x01|\01男
w\x01|\0122\x01|\01女
```

output-charset 字符集 [hubble@hubble01 bin]$ ./hubble-sql.sh --server=https://hubble01.hubbledb.cn:7077 --user=hubble -- catalog=hubble --password --output-format=DELIMITER_HEADER_UNQUOTED --delimiter='\x01|\01' --output-charset gbk --e xecute "select * from hubble.zh_wh_odsfff.t12" > b.csv

1.6 函数

1.6.1 逻辑运算符

运算符	描述	示例
AND	逻辑与	a AND b
OR	逻辑或	a OR b
NOT	逻辑非	NOT a

1.6.1.1 NULL

如果 AND 表达式中有一边或者两边都是 null，那么整个 AND 表达式的结果将会是 null。如果 AND 表达式中至少有一边的值是 false，那么整个 AND 表达式的值都是 false。

例如：

```
SELECT CAST(null AS boolean) AND true; -- null

SELECT CAST(null AS boolean) AND false; -- false

SELECT CAST(null AS boolean) AND CAST(null AS boolean); -- null
```

如果 OR 表达式的一边或者两边都是 null，那么整个 OR 表达式的值就是 null。如果 OR 表达式中只要有一边的值为 true，那么整个 OR 表达式的值就是 true。

例如：

```
SELECT CAST(null AS boolean) OR true; -- true

SELECT CAST(null AS boolean) OR false; -- null

SELECT CAST(null AS boolean) OR CAST(null AS boolean); -- null
```

下表说明了 AND 和 OR 表达式的计算规则：

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

NULL 的 NOT 表达式的结果还是 NULL，如下所示：

```
SELECT NOT CAST(null AS boolean); -- null
```

下表说明了 NOT 表达式的计算规则：

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

1.6.2 比较函数和运算符

运算符

运算符	描述
<	小于
>	大于
<=	小于等于
>=	大于等于
=	等于
<>	不等于
!=	不等于

范围运算符

- BETWEEN运算符检测值是否在指定范围内。它使用语法value BETWEEN min AND max:

```
SELECT 3 BETWEEN 2 AND 6; -- true
```

上面的语句等效于下面的语句:

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

- NOT BETWEEN运算符检测某个值不在指定范围内。它使用语法value NOT BETWEEN min AND max:

```
SELECT 3 NOT BETWEEN 2 AND 6; -- false
```

上面的语句等效于下面的语句:

```
  SELECT 3 < 2 OR 3 > 6; -- false
```

NULL在BETWEEN或者NOT BETWEEN语句中使用标准进行计算，应用于等效表达式的计算规则:

例如:

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

SELECT 2 BETWEEN NULL AND 6; -- null

SELECT 2 BETWEEN 1 AND NULL; -- false

SELECT 8 BETWEEN NULL AND 6; -- false
```

BETWEEN和NOT BETWEEN运算符也可用于评估任何可排序类型。

例如：

```
SELECT 'Paul' BETWEEN 'John' AND 'Ringo'; -- true
```

> 请注意，值、最小值和最大值参数BETWEEN和NOT BETWEEN必须是同一类型。例如，如果您询问 John 是否在 2.3 和 35.2 之间，它将产生错误。

1.6.2.1 NULL 和 NOT NULL

IS NULL和IS NOT NULL运算符检测值是否为空。适用于所有数据类型。

实时逻辑运算

- IS DISTINCT FROM

```
A IS DISTINCT FROM B
```

> 如果 A 和 B 的值不完全相同则返回 TRUE。如果 A 和 B 的值相同返回 FALSE。在这里 NULL 视为已知值，返回 TRUE。

- IS NOT DISTINCT FROM

```
A IS NOT DISTINCT FROM B
```

> 如果 A 和 B 的值不完全相同则返回 FALSE。如果 A 和 B 的值相同返回 TRUE。在这里 NULL 视为已知值，返回 FALSE。

直观的给出一个表格：

a	b	a 等于 b	a 不等于 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

最大和最小

非 SQL 标准函数，但属于常用扩展。

greatest(value1, value2, ..., valueN) 返回提供值中最大的。

least(value1, value2, ..., valueN) 返回提供值中最小的。

> 与大多数其他函数一样，如果任何一个参数为 null，它们将返回 null。

1.6.2.2 ALL、ANY 和 SOME 比较

ALL、ANY和SOME量词可以按以下方式与比较运算符一起使用：

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

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
```

量词和比较运算符组合的含义：

表示	意义
A = ALL(...)	结果为true当... 的时候A等于所有值。
A <> ALL(...)	结果为true当... 的时候A不匹配任何值。
A < ALL(...)	结果为true当... 的时候A小于最小值。
A = ANY(...)	结果为true当... 的时候A等于任何一个值。这种形式相当于A IN (...).
A <> ANY(...)	结果为true当... 的时候A与一个或多个值不匹配。
A < ANY(...)	结果为true当... 的时候A小于最大值。

ANY 任意，ALL 全部，SOME 其中的一些。

ANY和SOME意思相同，可以互换使用。

1.6.2.3 LIKE 比较

LIKE运算符可用于将值与模板进行比较：

```
column [NOT] LIKE 'pattern' ESCAPE 'character';
```

匹配字符区分大小写，该模板支持两种匹配符号：

- _匹配任何单个字符
- %匹配零个或多个字符

通常，它经常在中用作一个条件WHERE声明。例子是查找所有大洲的查询E，它返回：Europe

```
SELECT * FROM (VALUES 'America', 'Asia', 'Africa', 'Europe', 'Australia', '
    ↪ Antarctica') AS t (continent)
WHERE continent LIKE 'E%';
```

可以通过添加以下内容来否定结果NOT，并得到所有其他大陆，所有不开始E：

```
SELECT * FROM (VALUES 'America', 'Asia', 'Africa', 'Europe', 'Australia', '
    ↪ Antarctica') AS t (continent)
WHERE continent NOT LIKE 'E%';
```

如果只有一个特定的字符要匹配，可以使用_每个字符的符号。下面的查询使用了两个下划线，生成结果是：Asia

```
SELECT * FROM (VALUES 'America', 'Asia', 'Africa', 'Europe', 'Australia', '
    ↪ Antarctica') AS t (continent)
WHERE continent LIKE 'A__a';
```

通配符_和%必须进行转义，以允许将它们作为文本进行匹配。这可以通过指定ESCAPE要使用的字符：

```
SELECT 'South_America' LIKE 'South\_America' ESCAPE '\';
```

上述查询返回true因为转义的下划线符号匹配。

1.6.3 条件表达式

1.6.3.1 CASE

标准的 SQL 中CASE表达式有两种形式。简单从左向右查找表达式的每个值直到找出相等的表达式：

```
CASE expression
    WHEN value THEN result
    [ WHEN ... ]
    [ ELSE result ]
END
```

返回匹配值的结果。如果没有匹配到任何值，则返回 ELSE 子句的结果；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
```

查找从左向右判断每个condition的布尔值，直到判断为真，返回匹配结果：

```
CASE
    WHEN condition THEN result
    [ WHEN ... ]
    [ ELSE result ]
END
```

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

```
SELECT a, b,
       CASE
           WHEN a = 1 THEN 'aaa'
           WHEN b = 2 THEN 'bbb'
           ELSE 'ccc'
       END
```

1.6.3.2 IF

IF表达式有两种形式一种仅提供true_value，另一种同时提供true_value和false_value

- **if**(condition, true_value)

如果condition为真，则计算并返回true_value，否则返回 null，且不计算true_value。

- **if**(condition, true_value, false_value)

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

```
SELECT IF(1 > 0, 'true') AS result; --true

SELECT IF(1 > 0, 'true', 'false') AS result; --true
```

1.6.3.3 COALESCE

```
coalesce(value[, ...])
```

返回参数列表中的第一个非空值。与CASE表达式相似，仅在必要时计算参数。

```
SELECT COALESCE(NULL, 0);  -- 0

SELECT COALESCE(NULL, NULL); -- NULL
```

> COALESCE 函数需要许多参数，并返回第一个非 NULL 参数。如果所有参数都为 NULL，则 COALESCE 函数返回 NULL。

1.6.3.4 NULLIF

```
nullif(value1, value2)
```

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

案例 1

```
select nullif (0,0);  -- 0
```

案例 2

```
select nullif (1,2); -- null
```

案例 3

```
select nullif ('hubble','hubble'); -- null
```

案例 4

```
select nullif ('hubble','hubble1'); --hubble
```

案例 5

```
select nullif (0, NULL);  -- 0
```

案例 6

```
select nullif (null,0); --null
```

NULIF(0,0) 返回 NULL，因为 0 等于 0。
NULLIF(1,2) 返回 1，这是第一个参数，因为 1 不等于 2。
NULLIF('hubble'，'hubble') 返回 NULL，因为两个参数是相同的字符串。
NULLIF('hubble'，'hubble1') 返回 hubble，因为两个字符串不相等。
NULLIF(1，NULL) 返回 0，因为 0 不等于 NULL。
NULLIF(NULL，0) 返回第一个参数，即 NULL，因为 NULL 不等于 0。

1.6.3.5 TRY

```
try(expression)
```

计算表达式，如果发生错误返回 null。

如果希望查询生成NULL或者默认值，而不是在遇到损坏或无效数据时失败，该TRY函数可能很有用。若要指定默认值，请将TRY函数与COALESCE函数结合使用。

下列错误使用TRY处理：

- 被零除
- 无效的转换或函数参数
- 数值超出范围

例如，包含一些无效数据的源表：

```
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)
```

使用TRY返回 NULL 值：

```
SELECT TRY(CAST(origin_zip AS BIGINT)) FROM shipping;
```

```
 origin_zip
------------
      94131
      NULL
      94025
      08544
(4 rows)
```

TRY和COALESCE结合使用指定默认值：

```
SELECT COALESCE(TRY(total_cost / packages), 0) AS per_package FROM shipping;
```

```
 per_package
-------------
           4
          14
           0
          19
(4 rows)
```

1.6.4 Lambda 表达式

Lambda 表达式使用->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(x AS JSON)
x -> x + TRY(1 / 0)
```

大多数 SQL 表达式都可以用在 lambda 体中，但有一些例外:

- 不支持子查询: `x -> 2 + (SELECT 3)`
- 不支持聚合: `x -> max(y)`

例子:

获取数组列的平方元素transform():

```
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)
```

该函数transform()还可用于将数组元素安全地转换为字符串:

```
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)
```

除了正在操作的数组列之外，其他列也可以在 lambda 表达式中捕获。以下语句展示了此功能，使用transform
↪ ()用于计算线性函数f(x) = ax + b的值：

```
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)
```

使用any_match()查找包含至少一个大于100的值的数组元素：

```
SELECT numbers
FROM (
    VALUES
        (ARRAY[1,NULL,3]),
        (ARRAY[10,20,30]),
        (ARRAY[100,200,300])
) AS t(numbers)
WHERE any_match(numbers, n ->  COALESCE(n, 0) > 100);
-- [100, 200, 300]
```

通过以下方式将字符串中的第一个单词大写regexp_replace()：

```
SELECT regexp_replace('once upon a time ...', '^(\w)(\w*)(\s+.*)$',x -> upper(x
    ↪ [1]) || x[2] || x[3]);
```

```
-- Once upon a time ...
```

Lambda 表达式也可以应用于聚合函数。以下语句是一个示例，该示例使用reduce_agg()对列的所有元素的总和进行过于复杂的计算：

```
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);
-- 15
```

1.6.5 转换函数

1.6.5.1 CAST

- cast(value AS type)→ type

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

```
SELECT CAST(100 AS VARCHAR);  --100

SELECT CAST('100' AS INT); --100

SELECT CAST("10:00:10" AS TIME);   -- 10:00:10
```

当字符串 cast 为 char(n)，不足的字符用空格填充，多的字符被截断

```
select length(cast('hello world' as char(100))); -- 100
```

- try_cast(value AS type)→ type

与 cast() 相似，区别是转换失败返回 null。

```
SELECT TRY_CAST('test' AS int) ; -- null

SELECT TRY_CAST('123' AS int) ;   -- 123
```

```
  SELECT
    CASE WHEN TRY_CAST('test' AS int) IS NULL
     THEN 'Cast failed'
     ELSE 'Cast succeeded'
    END AS Result;
  --Cast failed

   SELECT
    CASE WHEN TRY_CAST('123' AS int) IS NULL
     THEN 'Cast failed'
     ELSE 'Cast succeeded'
    END AS Result;
  -- Cast succeeded
```

1.6.5.2 FORMAT

- format(format, args...)→ varchar

使用指定的格式字符串和参数返回格式化字符串。

例如:

```
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

值转化为存储容量。

例如:

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

1.6.5.3 PARSE_DATA_SIZE

- parse_data_size(string)

将容量转换成纯数字值。

例如:

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

parse_data_size支持以下单位:

Unit	Description	Value
B	Bytes	1
kB	Kilobytes	1024
MB	Megabytes	10242
GB	Gigabytes	10243
TB	Terabytes	10244
PB	Petabytes	10245
EB	Exabytes	10246
ZB	Zettabytes	10247
YB	Yottabytes	10248

1.6.5.4 TYPEOF

- typeof(expr)→ varchar

返回所提供表达式的类型名称。

例如:

```
SELECT typeof(123); -- integer

SELECT typeof('cat'); -- varchar(3)

SELECT typeof(cos(2) + 1.5); -- double
```

1.6.6 数学函数和运算符

数学运算符

运算符	描述
+	加法
-	减法
*	乘法
/	除法（整数除法进行截断）
%	模数（余数）

1.6.6.1 数学函数

- abs(x)→ 与输入类型相同

返回x的绝对值。

```
select abs(-11.19); -- 11.19
```

- cbrt(x)→ double

返回x的立方根。

```
select cbrt(27.0); -- 3.0
```

- ceil(x)→ 与输入类型相同

是ceiling()的同名方法。

- ceiling(x)→ 与输入类型相同

返回x的向上取整的数值。

```
select ceil(23.7); -- 24
```

- degrees(x)→ double

将弧度转为角度。

```
select degrees(0.5); -- 28.64788975654116
```

- e() → double

返回 Euler 数的常数

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

- exp(x)→ double

返回 Euler 数提升到x的幂。

```
select exp(1.0); -- 2.718281828459045
```

- floor(x)→ 与输入类型相同

返回x向下舍入到最接近的整数。

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

- ln(x)→ double

返回x的自然对数。

```
select ln(2.0); -- 0.6931471805599453
```

- log(b, x)→ double

返回以b为底的对数x。

```
select log(2, 64); -- 6.0
```

- log2(x)→ double

返回以 2 为底的对数x。

```
select log2(8); -- 3.8
```

- log10(x)→ double

返回以 10 为底的对数x。

```
select log10(100); -- 2.0
```

- mod(n, m)→ 与输入类型相同

返回n除以m的模数 (余数)。

```
select mod(8,3); -- 2
```

- pi()→ double

返回圆周率。

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

- pow(x, p)→ double

是power()的同名方法。

- power(x, p)→ double

返回x的p次方。

```
select power(2, 16); -- 65536.0
```

- radians(x)→ double

将角度转换为弧度。

```
select radians(90); -- 1.5707963267948966
```

- round(x)→ 与输入类型相同

返回x四舍五入到最接近的整数。

```
select round(125.315); -- 125
```

- round(x, d)→ 与输入类型相同

返回x四舍五入到d小数位数。

```
select round(125.315, 2); -- 125.320
```

- sign(x) → 与输入类型相同

函数返回x的正负号，即：

- 如果参数为 0，则为 0，
- 如果参数大于 0，则为 1，
- 如果参数小于 0，则为-1。

对于浮点参数，该函数还会返回：

- 如果参数是 NaN，则为 NaN，
- 如果参数是 +Infinity，则为 1，
- 如果参数是 -Infinity，则为-1。

```
select sign(-2.6); -- -1
```

- sqrt(x)→ double

返回x的平方根。

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

- truncate(x)→ double

截取函数，舍去小数点后的数字，返回x中整数部分。

```
select truncate(124.344); -- 124
```

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

返回一个桶，这个桶是在一个有n个桶，上界为bound1，下界为bound2的柱图中x将被赋予的那个桶。若输入在范围外部，则返回 0 或者 n+1。

```
SELECT width_bucket(5.3, 0.2, 10.6, 5); -- 3
```

- width_bucket(x, bins)→ bigint

返回一个桶，它是在给定数组列出所有的桶中x将被赋予的那个桶，bins 数组必须 double 数组，并且是升序排序的。若输入在范围外部，则返回 0 或者 n+1。

```
select width_bucket(5.3, array [0.2,1.2,4.2,6.2,8.2,10.2]); -- 3
```

1.6.6.2 随机函数

- rand()→ double

是random()的同名方法。

- random()→ double

返回 0.0 <= x < 1.0 范围内的伪随机值。

```
select random(); -- 0.10773738703313251
```

- random(n)→ 与输入类型相同

返回 0 到 n（不包括）之间的伪随机数。

```
select random(10); -- 6
```

- random(integer,integer)→ integer

返回指定两个值之间的伪随机数。

```
select random(10, 20); -- 15
```

- random(smallint,smallint)→ smallint

  返回指定两个值之间的伪随机数。

- random(tinyint,tinyint)→ tinyint

  返回指定两个值之间的伪随机数。

1.6.6.3 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

返回x的反余弦值。

```
select acos(0.87); -- 0.5155940062460905
```

- asin(x)→ double

返回x的反正弦值。

```
select asin(1); -- 1.5707963267948966
```

- atan(x)→ double

返回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(2.11); -- -0.5134528123039594
```

- cosh(x)→ double

返回x的双曲余弦值。

```
select cosh(1.3); -- 1.9709142303266285
```

- sin(x)→ double

返回x的正弦值。

```
select sin(1); -- 0.8414709848078965
```

- tan(x)→ double

返回x的正切值。

```
select tan(1); -- 1.5574077246549023
```

- tanh(x)→ double

返回x的双曲正切值。

```
select tanh(1); -- 0.7615941559557649
```

1.6.6.4 浮点函数

- infinity()→ double

返回表示正无穷大的常量 Infinity。

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

- is_finite(x)→ boolean

返回x是否是有限的。

```
select is_finite(3); -- true
```

- is_infinite(x)→ boolean

返回x是否是无限的。

```
select is_infinite(3); -- false
```

- is_nan(x)→ boolean

返回x是否不是数字。

```
select is_nan(nan()); -- true
```

- nan()→ double

返回表示非数字的常量 NaN。

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

1.6.6.5 转换函数

- from_base(string, radix)→ bigint

返回被解释为基数的字符串的值。

```
select from_base('100', 2); -- 4
```

- to_base(x, radix)→ varchar

将数字转换为具备给定基数的文本表示。

```
select to_base(4, 2); -- 100
```

1.6.6.6 统计函数

- 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
```

- wilson_interval_lower(successes, trials, z)→ double

返回 Bernoulli 方程计算的 Wilson 评分的下限。

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

- wilson_interval_upper(successes, trials, z)→ double

返回 Bernoulli 方程计算的 Wilson 评分的上限。

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

1.6.6.7 累计分布函数

- beta_cdf(a, b, v)→ double

贝塔累计分布函数概率。a、b 参数必须是正实数，值 v 必须是实数。值 v 必须位于区间 [0, 1] 内。

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

- inverse_beta_cdf(a, b, p)→ double

贝塔累计分布函数逆概率。a、b 参数必须是正实数值。概率 p 必须位于区间 [0, 1] 内。

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

- inverse_normal_cdf(mean, sd, p)→ double

计算一个普通累计分布函数逆概率通过平均值和标准差，平均值必须是实数，标准差必须是实数，标准差必须是正实数。概率 p 必须位于区间 [0, 1] 内。

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

- normal_cdf(mean, sd, v)→ double

使用平均值和标准差计算普通累计分布函数概率。平均值和值 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
```

1.6.7 位运算函数

- bit_count(x, bits)→ bigint

计算x的二进制补码中的位数（视为有符号整数）。

```
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

按位与，返回x和y二进制补码的按位与。

例如：19(二进制：10011) 和25(二进制：11001) 的按位与的运算结果为17(二进制：10001)

```
SELECT bitwise_and(19,25); -- 17
```

- bitwise_not(x)→ bigint

按位非，返回x和y二进制补码的按位非。NOT x = -x - 1

例如：

```
SELECT bitwise_not(-12); --  11
SELECT bitwise_not(19);  -- -20
SELECT bitwise_not(25);  -- -26
```

- bitwise_or(x, y)→ bigint

按位或，返回x和y二进制补码的按位或。

例如：19(二进制：10011) 和25(二进制：11001) 的按位或的运算结果为27(二进制：11011)

```
SELECT bitwise_or(19,25); -- 27
```

- bitwise_xor(x, y)→ bigint

按位异或，返回x和y二进制补码的按位异或。

例如：19(二进制：10011) 和25(二进制11001) 的按位或的运算结果为10(二进制：01010)

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

- bitwise_left_shift(value, shift)→ 与输入类型相同

对指定数值，逻辑左移位数。

例如：

将1(二进制：001) 向左移动两位结果为4(二进制：00100) 。

```
SELECT bitwise_left_shift(1, 2); -- 4
```

将5(二进制：0101) 向左移动两位结果为20(二进制：010100) 。

```
SELECT bitwise_left_shift(5, 2); -- 20
```

将value向左移动0位结果总为原来的值。

```
SELECT bitwise_left_shift(20, 0); -- 20
SELECT bitwise_left_shift(42, 0); -- 42
```

将0向左移动shift位结果都为 0。

```
SELECT bitwise_left_shift(0, 1); -- 0
SELECT bitwise_left_shift(0, 2); -- 0
```

- bitwise_right_shift(value, shift)→ 与输入类型相同

对指定数值，逻辑右移位数。

例如：

将8(二进制：1000) 向右移动三位结果为1(二进制：001) 。

```
SELECT bitwise_right_shift(8, 3); -- 1
```

将9(二进制：1001) 向右移动一位结果为4(二进制：100) 。

```
SELECT bitwise_right_shift(9, 1); -- 4
```

将value向右移动0位结果总为原来的值。

```
SELECT bitwise_right_shift(20, 0); -- 20
SELECT bitwise_right_shift(42, 0); -- 42
```

将value向右移动64或更多位结果都为 0。

```
SELECT bitwise_right_shift( 12, 64); -- 0
SELECT bitwise_right_shift(-45, 64); -- 0
```

将0向右移动shift位结果都为 0。

```
SELECT bitwise_right_shift(0, 1); -- 0
SELECT bitwise_right_shift(0, 2); -- 0
```

- `bitwise_right_shift_arithmetic(value, shift)→` 与输入类型相同

对指定数值，算数右移位数。

当向右移动小于 64 位时，结果与`bitwise_right_shift()`相同。

向右移动64或更多位时，当`value`为正数时结果为 0，负数时结果为-1。

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

1.6.8 数值常量与计算

1.6.8.1 数值常量

使用`DECIMAL 'aaaaaaa.bbbbbbbb'`语法定义数值类型。

数值例子	数据类型
DECIMAL '0'	DECIMAL(1)
DECIMAL '12345'	DECIMAL(5)
DECIMAL '0000012345.1234500000'	DECIMAL(20, 10)

1.6.8.2 数值计算精度

支持标准数学运算符。下表说明了结果的精度和宽度计算规则。假设 x 是`DECIMAL(xp, xs)`类型，y 是`DECIMAL (yp, ys)` 类型。

操作	结果的精度	结果的宽度
`x + y`或`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不能精确地用结果数据类型的精度和比例表示，将会引发异常`Value is out of range`。

> 在具有不同精度的数值计算时，最好先将值类型的精度和范围统一，对于一个接近最大精度 38 的类型，可能会导致错误，如一个值类型 DECIMAL(38,0)，一个值 DECIMAL(38,1)，计算结果可能超出最大精度引发错误。

1.6.9 字符串函数和运算符

1.6.9.1 字符串运算符

使用运算符 `||` 完成字符串连接。

使用运算符 `LIKE` 用于模板匹配。

1.6.9.2 字符串函数

注意

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。

具体而言，对于立陶宛语、土耳其语和阿塞拜疆语，这将返回不正确的结果。

- chr(n)→ varchar

以单个字符串的形式返回 Unicode 代码点n。

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

- codepoint(string)→ integer

返回string的唯一字符的 Unicode 编码点。

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

- concat(string1, ..., stringN)→ varchar

返回string1、string2、...、stringN的连接结果。该函数提供与 SQL 标准连接运算符 (||) 相同的功能。

```
select concat('hello', 'hubble'); -- hellohubble
```

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

连接字符串string1, string2, ..., stringN使用string0作为分隔符，将多个字符串进行拼接。如果string0为空，则返回值为空。连接时跳过分隔符后面的参数中提供的任何空值。

```
select concat_ws(',', 'hello', 'hubble'); -- hello,hubble
```

- hamming_distance(string1, string2)→ bigint

返回string1和string2的汉明距离，即对应字符不同的位置数。请注意，这两个字符串的长度必须相同。

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

- length(string)→ bigint

返回string字符串的长度。

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

- levenshtein_distance(string1, string2)→ bigint

返回String1和String2的Levenshtein编辑距离，即将String1更改为String2所需的最小单字符编辑数（插入、删除或替换）。

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

- lower(string)→ varchar

转换string为小写。

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

- lpad(string, size, padstring)→ varchar

从左边对字符串string使用指定的字符padstring进行拼接，直到长度达到size。如果size小于string的长度，结果被截断为size个字符。size不得为负，且padstring不能为空。

```
select lpad('kmy', 5, 'zda'); -- zdkmy
```

- ltrim(string)→ varchar

删除字符串string所有前导空格。

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

- position(substring IN string)→ bigint

返回字符串string中子字符串substring第一次出现的位置，位置从1开始，如果未找到则返回0。

```
select position('l' in 'hello'); -- 3
```

- replace(string, search)→ varchar

将string中所有的search内容移除。

```
select replace('hello', 'l'); -- heo
```

- replace(string, search, replace)→ varchar

将string中的search内容替换为replace。

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

- reverse(string)→ varchar

返回string逆序后的字符串。

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

- rpad(string, size, padstring)→ varchar

从右边对字符串string使用指定的字符padstring进行拼接，直到长度达到size，并返回拼接后的字符串。如果size小于string的长度，结果被截断为size个字符。size不得为负，且padstring不能为空。

```
select rpad('kmy',5, 'zda'); -- kmyzd
```

- rtrim(string)→ varchar

删除字符串string所有后置空格。

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

- soundex(char)→ varchar

将字符串编码为 SOUNDEX 值。以评估两个字符串在发音时的相似性。

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'); -- M460
```

- split(string, delimiter)-> array(varchar)

将字符串string按分隔符delimiter进行分隔，并返回数组。

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

- split(string, delimiter, limit)-> array(varchar)

将字符串string按分隔符delimiter进行分隔，并返回按limit大小限制的数组。数组中的最后一个元素包含字符串中的所有剩余内容。limit必须是正数。

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

- split_part(string, delimiter, index)→ varchar

将字符串string按分隔符delimiter进行分隔，并返回分隔后数组下标为index的子串。index从 1 开始，超出范围返回 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)

将string按entryDelimiter和keyValueDelimiter拆分并返回map。entryDelimiter将字符串分解为key-value对，keyValueDelimiter将每对key-value分隔成key和value。

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

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

将string按entryDelimiter和keyValueDelimiter拆分并返回map，其中包含每个唯一key的value数组。entryDelimiter将字符串分解为key-value对，keyValueDelimiter将每对key-value分隔成key和value。每个key的value的顺序与它们在string中出现的顺序相同。

```
select split_to_multimap('zhang:18,li:17,zhang:20,wang:19', ',', ':');
  --  {zhang=[18, 20], wang=[19], li=[17]}
```

- strpos(string, substring)→ bigint

返回字符串string中子字符串substring第一次出现的位置。位置以 1 开始，如果未找到则返回 0。

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

- strpos(string, substring, instance)→ bigint

返回字符串string中子字符串substring第instance次出现的位置。当instance为负数时，将从string的末尾开始搜索。

位置以 1 开始，如果未找到则返回 0。

```
select strpos('hello!', 'l', -2); -- 3
```

- starts_with(string, substring)→ boolean

判断子字符串substring是否是字符串string的前缀。

```
select starts_with('hello!', 'hel'); -- true
```

- substr(string, start)→ varchar

是substring()的同名方法。

- substr(string, start, length)→ varchar

是substring()的同名方法。

- substring(string, start)→ varchar

返回字符串string从start位置开始到结束的子串。位置以 1 开始，如果start<0，则start位置从字符串的末尾开始倒数。

```
select substring('abcde', 3); -- cde
```

- substring(string, start, length)→ varchar

返回字符串string从start位置开始长度为length的子串。位置以 1 开始，如果start<0，则start位置从字符串的末尾开始倒数。

```
select substring('abcde', 3, 2); -- cd
```

- translate(source, from, to)→ varchar

将source字符串中，符合from的字符，替换为to，并返回。如果from 字符串包含重复项，则仅使用第一个。如果source字符串中不存在该字符from，source字符串将被复制而无需替换。如果字符串中匹配字符的索引from超出字符串to的长度，则该字符将从结果字符串source中省略。

```
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', 'b', U&'\+01F600'); -- a<U+1F600>cd
SELECT translate('abcd', 'a', ''); -- 'bcd'
SELECT translate('abcd', 'a', 'zy'); -- 'zbcd'
SELECT translate('abcd', 'ac', 'z'); -- 'zbd'
SELECT translate('abcd', 'aac', 'zq'); -- 'zbd'
```

- trim(string)→ varchar

删除字符串string所有的前后空格。

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

- trim([[specification] [string] FROM] source)→ varchar

删除指定的任何前导和/或尾随字符。

```
SELECT trim('!' FROM '!foo!'); -- 'foo'
SELECT trim(LEADING FROM '  abcd');  -- 'abcd'
SELECT trim(BOTH '$' FROM '$var$'); -- 'var'
SELECT trim(TRAILING 'ER' FROM upper('worker')); -- 'WORK'
```

- upper(string)→ varchar

转换string为大写。

```
select upper('hello hubble!'); -- HELLO HUBBLE!
```

- word_stem(word)→ varchar

返回英语中word的词干。

```
select word_stem('greating'); -- great
```

- word_stem(word, lang)→ varchar

返回lang语言中word的词干。

```
select word_stem('ultramoderne', 'fr'); -- ultramodern
```

1.6.9.3 Unicode 函数

- normalize(string)→ varchar

用 UNICODE NFC 标准化形式转换字符串。

```
select normalize('e'); -- e
```

- normalize(string, form)→ varchar

用指定的标准化形式转换字符串。form必须是以下关键字之一:

form	描述
NFD	正则分解
NFC	正则分解，随后跟正则分解
NFKD	兼容性分解
NFKC	兼容性分解，后跟正则分解

```
select normalize('é', NFC); -- é
```

- to_utf8(string)→ varbinary

字符串转 UTF-8 表示形式。

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

- from_utf8(binary)→ varchar

解码 UTF-8 编码的字符串binary。无效的 UTF-8 序列被替换为 Unicode 替换字符U+FFFD.

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

- from_utf8(binary, replace)→ varchar

解码 UTF-8 编码的字符串binary。无效的 UTF-8 序列被替换为replace。替换字符串 replace 必须是单个字符或为空（在这种情况下，无效字符将被删除）。

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

1.6.10 正则表达式函数

注意

所有正则表达式函数都使用 Java 模式语法，但有一些值得注意的例外：

- 当使用多行模式（通过(?m)标志启用）时，仅将\n识别为行终止符。此外，不支持(?d)标志，不得使用该标志。
- 不区分大小写的匹配（通过(?i)标志启用）始终以支持 Unicode 的方式执行。不过，不支持上下文相关和局部相关的匹配。此外，不支持(?u)标志，不得使用该标志。
- 不支持代理项对。例如，\uD800\uDC00不被视为U+10000，必须将其指定为\x{10000}。对于没有基字符的不占位标记，会错误地处理边界 (\b)。
- 在字符类（如[A-Z123]）中不支持\Q和\E，而应将其视为字面量。
- 支持 Unicode 字符类 (\p{prop})，但有如下差异：
- 必须删除名称中的所有下划线。例如，使用OldItalic代替Old_Italic。
- 必须直接指定脚本，而不使用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，而不使用Is、general_category=或gc=前缀。示例：\p{L}
- 必须直接指定二进制属性，而不使用Is。示例：\p{NoncharacterCodePoint}

- regexp_count(string, pattern)-> bigint

返回正则表达式pattern在string中出现次数。

例如：

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

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

返回正则表达式pattern在string中匹配的子字符串。

例如：

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

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

查找string中出现的所有正则表达式pattern实例，并返回捕获组编号group。

例如:

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

- regexp_extract(string, pattern)→ varchar

返回正则表达式pattern在string中匹配的第一个子字符串。

例如:

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

- regexp_extract(string, pattern, group)→ varchar

查找string中出现的第一个正则表达式pattern实例，并返回捕获组编号group。

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

- regexp_like(string, pattern)→ boolean

计算正则表达式pattern并确定它是否包含在string中。

该函数与LIKE运算符类似，不过只需在string中包含模板，而无需匹配整个string。换句话说，该函数执行的是**包含**运算，而不是**匹配**运算。可以通过使用^和$定位模板来匹配整个字符串。

例如:

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

- regexp_position(string, pattern)→ integer

返回正则表达式pattern在string中第一个匹配项的索引（从 1 开始计数）。如果未找到，则返回 -1。

例如:

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

- regexp_position(string, pattern, start)→ integer

返回正则表达式pattern在string中第一个匹配项的索引，从start开始（包括start）。如果未找到，则返回 -1。

例如:

```
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

返回正则表达式pattern在string中第 n 个匹配项的索引occurrence，从start开始（包括start）。如果未找到，则返回 -1。

例如：

```
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

从string中删除由正则表达式pattern匹配的子字符串的每个实例。

例如：

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

- regexp_replace(string, pattern, replacement)→ varchar

将string中由正则表达式pattern匹配的子字符串的每个实例替换为replacement。捕获组可以使用$g（对于编号的组）或${name}（对于命名的组）在replacement中引用。替换时，可以通过使用反斜杠 (\$) 进行转义来包含符号 ($)。

例如：

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

- regexp_replace(string, pattern, function)→ varchar

使用function替换string中由正则表达式pattern匹配的子字符串的每个实例。会针对每个匹配项调用lambda
↪ expression <lambda> function，其中以数组形式传入捕获组。捕获组编号从 1 开始；整个匹配没有组（如果需要，可以使用圆括号将整个表达式括起来）。

例如：

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

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

使用正则表达式pattern拆分string并返回一个数组。保留尾随空字符串。

例如：

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

1.6.11 二进制函数和运算符

二进制运算符

|| 运算符执行连接。

1.6.11.1 二进制函数

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

返回 binary1、binary2、...、binaryN 的连接结果。该函数提供与 SQL 标准连接运算符 (||) 相同的功能。

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

- length(binary)→ bigint

返回 binary 的长度，以字节为单位。

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

- lpad(binary, size, padbinary)→ varbinary

使用 padbinary 将 binary 左填充至 size 个字节。如果 size 小于 binary 的长度，结果将被截断至 size 个字符。size 不得为负数，并且 padbinary 必须为非空值。

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

- rpad(binary, size, padbinary)→ varbinary

使用 padbinary 将 binary 右填充至 size 个字节。如果 size 小于 binary 的长度，结果将被截断至 size 个字符。size 不得为负数，并且 padbinary 必须为非空值。

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

- substr(binary, start)→ varbinary

从起始位置 start 开始返回 binary 的其余部分，以字节为单位。位置从 1 开始。负起始位置表示相对于字符串的末尾。

```
select substr(x'15245F', 2); -- 24 5f
```

- substr(binary, start, length)→ varbinary

从起始位置 start 开始从 binary 返回长度为 length 的子字符串，以字节为单位。位置从 1 开始。负起始位置表示相对于字符串的末尾。

```
select substr(x'15245F', 1, 3); -- 15 24 5f
```

- reverse(binary)→ varbinary

返回binary字节的逆序。

```
select reverse(x'15245F'); -- 5f 24 15
```

1.6.11.2 Base64 编码函数

> Base64 函数实现中指定的编码 RFC 4648。

- from_base64(string)→ varbinary

从以 base64 编码的 string 解码二进制数据。

```
select from_base64('hellohubble'); --  85 e9 65 a2 1b 9b 6e 57
```

- to_base64(binary)→ varchar

将 binary 编码为 base64 字符串表示形式。

```
select to_base64(x'85e965a21b9b6e57'); -- hellohubblc=
```

- from_base64url(string)→ varbinary

使用 URL 安全字母表从以 base64 编码的 string 解码二进制数据。

```
select from_base64url('ABCD'); -- 00 10 83
```

- to_base64url(binary)→ varchar

使用 URL 安全字母表将 binary 编码为 base64 字符串表示形式。

```
select to_base64url(x'001083');  -- ABCD
```

- from_base32(string)→ varbinary

从以 base32 编码的 string解码二进制数据。

```
select from_base32('ABCD'); -- 00 44
```

- to_base32(binary)→ varchar

将 binary 编码为 base32 字符串表示形式。

```
select to_base32(x'0044'); -- ABCA====
```

1.6.11.3 十六进制编码函数

- from_hex(string)→ varbinary

从以十六进制编码的 string 解码二进制数据。

```
select from_hex('FFFF'); --  ff ff
```

- to_hex(binary)→ varchar

将 binary 编码为十六进制字符串表示形式。

```
select to_hex(x'ffff');  --  FFFF
```

1.6.11.4 整数编码函数

- from_big_endian_32(binary)→ integer

解码 32 位二进制补码 big-endian binary。输入必须正好是 4 个字节。

```
select  from_big_endian_32(to_big_endian_32(10));  --  10
```

- to_big_endian_32(integer)→ varbinary

以 32 位二进制补码 big-endian 格式对 integer 进行编码。

```
select  to_big_endian_32(10);  --  00 00 00 0a
```

- from_big_endian_64(binary)→ bigint

解码 64 位二进制补码 big-endian binary。输入必须正好是 8 个字节。

```
select  from_big_endian_64(to_big_endian_64(10));  --  10
```

- to_big_endian_64(bigint)→ varbinary

以 64 位二进制补码 big-endian 格式对 bigint 进行编码。

```
select  to_big_endian_64(10);  --  00 00 00 00 00 00 00 0a
```

1.6.11.5 浮点编码函数

- from_ieee754_32(binary)→ real

对采用 IEEE 754 单精度浮点格式的 32 位 big-endian binary 进行解码。输入必须正好是 4 个字节。

```
select from_ieee754_32(to_ieee754_32(10));  --  10
```

- to_ieee754_32(real)→ varbinary

根据 IEEE 754 单精度浮点格式将 real 编码为 32 位 big-endian 二进制数。

```
select to_ieee754_32(10);   --  41 20 00 00
```

- from_ieee754_64(binary)→ double

对采用 IEEE 754 双精度浮点格式的 64 位 big-endian binary 进行解码。输入必须正好是 8 个字节。

```
select from_ieee754_64(to_ieee754_64(10));  --  10
```

- to_ieee754_64(double)→ varbinary

根据 IEEE 754 双精度浮点格式将 double 编码为 64 位 big-endian 二进制数。

```
select to_ieee754_64(10);    --  40 24 00 00 00 00 00 00
```

1.6.11.6 哈希函数

- crc32(binary)→ bigint

  计算 binary 的 CRC-32 值。对于通用哈希，请使用 xxhash64，因为它速度更快并且能生成质量更好的哈希值。

```
select crc32(from_base64('aaaaaa'));  --  90176811
```

- md5(binary)→ varbinary

  计算 binary 的 MD5 哈希值。

```
select md5(from_base64('aaaaaa'));
  --  1d 2d 3a be d3 6f 04 52 bf 93 fd 57 51 60 c6 38
```

- sha1(binary)→ varbinary

  计算 binary 的 SHA1 哈希值。

```
select sha1(from_base64('aaaaaa'));
  -- 23 f0 b5 a9 bc c1 8e a7 a3 6e 09 b2 27 6a df 22 9d 91 4c c5
```

- sha256(binary)→ varbinary

  计算 binary 的 SHA256 哈希值。

```
select sha256(from_base64('aaaaaa'));
  --  9c ee 7d aa e7 f8 d5 73 78 9b ef a3 35 b5 5d 5a
      53 6d 64 45 8c 0d 29 ec fa 1d 99 94 8c 16 fd 00
```

- sha512(binary)→ varbinary

  计算 binary 的 SHA512 哈希值。

```
select sha512(from_base64('aaaaaa'));
  --  81 d1 a5 77 1b f1 c8 af f0 f0 1c cd d0 e2 78 6c
      6e 8a 23 cd 08 cb 69 d4 b3 9a e7 af 9f af 1f 5e
      a2 18 be dd 9e 8d 50 6a 14 5c 28 84 ad db 56 53
      dd 73 7a 1c bf d6 dd a3 a7 75 d3 10 6c fe 55 f3
```

- spooky_hash_v2_32(binary)→ varbinary

  计算 binary 的 32 位 SpookyHashV2 哈希值。

```
select spooky_hash_v2_32(from_base64('aaaaaa')) --  a1 6e 2c 23
```

- spooky_hash_v2_64(binary)→ varbinary

  计算 binary 的 64 位 SpookyHashV2 哈希值。

```
  select spooky_hash_v2_64(from_base64('aaaaaa')) ;
  --  bf b3 00 b2 a1 6e 2c 23
```

- xxhash64(binary)→ varbinary

  计算 binary 的 xxhash64 哈希值。

```
  select xxhash64(from_base64('aaaaaa'));
  --  c2 d4 6c 79 1a 94 80 b7
```

- murmur3(binary)→ varbinary

  计算binary的 128 位 MurmurHash3 哈希值。

```
  SELECT murmur3(from_base64('aaaaaa'));
  -- ba 58 55 63 55 69 b4 2f 49 20 37 2c a0 e3 96 ef
```

1.6.11.7 HMAC 函数

- hmac_md5(binary, key)→ varbinary

  使用给定的 key 计算 binary 的 HMAC 值（采用 md5）。

```
select hmac_md5(x'555555', x'aa'); -- 81 a8 79 3f 89 f9 33 fa c8 1e 08 64 c4 d1
    ↪ 0b 80
```

- hmac_sha1(binary, key)→ varbinary

  使用给定的 key 计算 binary 的 HMAC 值（采用 sha1）。

```
select hmac_sha1(x'555555', x'aa');
  -- ef 7c 81 0e 2e 72 dd 70 f8 68 a1 62 92 04 2a e0 7c 62 2b dc
```

- hmac_sha256(binary, key)→ varbinary

  使用给定的 key 计算 binary 的 HMAC 值（采用 sha256）。

```
select hmac_sha256(x'555555', x'aa');
  -- ef 7c 81 0e 2e 72 dd 70 f8 68 a1 62 92 04 2a e0 7c 62 2b dc
```

- hmac_sha512(binary, key)→ varbinary

  使用给定的 key 计算 binary 的 HMAC 值（采用 sha512）。

```
select hmac_sha512(x'555555', x'aa');
  -- eb 4b 9c bc 9f 57 67 2f a7 de 56 dc f1 8b 61 b8
     36 e9 ab 5a 4d 5f b9 27 ab c0 b3 21 43 19 67 7a
     fa 87 46 a8 12 4c 3e f4 ed 64 0d 06 35 82 f2 e5
     d7 e8 1a 41 56 31 8f 6b 1b 93 a8 58 73 c9 af a5
```

1.6.12 JSON 函数和运算符

1.6.12.1 转换为 JSON

以下类型可以转换为 JSON：

- BOOLEAN
- TINYINT
- SMALLINT
- INTEGER
- BIGINT
- REAL
- DOUBLE
- VARCHAR

此外，满足以下要求时，可以将 ARRAY、MAP或ROW类型转换为JSON：

- ARRAY当数组的元素类型是支持的类型之一时，可以转换类型。
- MAP当映射的键类型是VARCHAR并且映射的值类型是受支持的类型时，可以转换类型，
- ROW当行的每个字段类型都是受支持的类型时，可以转换类型。

注意

具有支持的字符串类型的转换操作将输入视为字符串，而不是作为 JSON 进行验证。这意味着使用无效 JSON 的字符串类型输入的转换操作会导致成功转换为无效 JSON。

相反，可以考虑使用json_parse()函数从字符串创建经过验证的 JSON。

下面通过示例展示了使用这些类型转换为 JSON 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 '{"v1
               ↪ ":123,"v2":"abc","v3":true}'
```

注意

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。

1.6.12.2 从 JSON 进行转换

支持转换为BOOLEAN、TINYINT、SMALLINT、INTEGER、BIGINT、REAL、DOUBLE或VARCHAR。当数组的元素类型为支持的类型之一或 Map 的键类型为VARCHAR且 Map 的值类型为支持的类型之一时，支持转换为ARRAY和MAP。下面通过示例展示了转换的行为：

```
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)); -- {v1=123, v2=abc, v3=true
                ↪ }
```

注意

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，Hubble 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']
```

> **注意**
>
> 在从JSON转换为ROW时，支持 JSON 数组和 JSON 对象。

1.6.12.3 JSON 函数

- `is_json_scalar(json)→ boolean`

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

```
SELECT is_json_scalar('1');          -- true
SELECT is_json_scalar('[1, 2, 3]'); -- false
```

- json_array_contains(json, value)→ boolean

确定json（包含 JSON 数组的字符串）中是否存在value：

```
SELECT json_array_contains('[1, 2, 3]', 2); -- true
```

- json_array_get(json_array, index)→ json

> **警告**
>
> 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）。
>
> 我们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，并且可能会在将来的版本中删除该函数。

将指定索引处的元素返回到json_array中。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]'
```

该函数还支持负索引，以便获取从数组的末尾开始索引的元素：

```
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]'
```

如果指定索引处的元素不存在，该函数将返回 NULL：

```
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（包含 JSON 数组的字符串）的数组长度：

```
SELECT json_array_length('[1, 2, 3]'); -- 3
```

- json_extract(json, json_path)→ json

计算json（包含 JSON 的字符串）上的类似于 JSONPath 表达式json_path并将结果作为 JSON 字符串返回：

```
SELECT json_extract(json, '$.store.book');
SELECT json_extract(json, '$.store[book]');
SELECT json_extract(json, '$.store["book name"]');
```

- json_extract_scalar(json, json_path)→ varchar

与json_extract()类似，但将结果值作为字符串返回（而不是编码为 JSON）。json_path引用的值必须是常量（布尔值、数字或字符串）：

```
SELECT json_extract_scalar('[1, 2, 3]', '$[2]');
SELECT json_extract_scalar(json, '$.store.book[0].author');
```

- json_format(json)→ varchar

返回从输入 JSON 值序列化的 JSON 文本。这是json_parse()的反函数:

```
SELECT json_format(JSON '[1, 2, 3]'); -- '[1,2,3]'
SELECT json_format(JSON '"a"');       -- '"a"'
```

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 RFC 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

返回从输入 JSON 文本反序列化的 JSON 值。这是json_format()的反函数:

```
SELECT json_parse('[1, 2, 3]');    -- JSON '[1,2,3]'
SELECT json_parse('"abc"');        -- JSON '"abc"'
```

注意

json_parse()和CAST(string AS JSON)具有完全不同的语义。

json_parse()期望输入符合 RFC 7159 标准的 JSON 文本，并返回从该 JSON 文本反序列化

的 JSON 值。JSON 值可以是 JSON 对象、JSON 数组、JSON 字符串、JSON 数字、true、false或null：

SELECT json_parse('not_json'); -- ERROR!

SELECT json_parse('{"a": 1, "b": 2}'); -- JSON '{"a": 1, "b": 2}'

SELECT json_parse('[1, 2, 3]'); -- JSON '[1,2,3]'

SELECT json_parse('"abc"'); -- JSON '"abc"'

SELECT json_parse('42'); -- JSON '42'

SELECT json_parse('true'); -- JSON 'true'

SELECT json_parse('null'); -- JSON 'null'

CAST(string AS JSON)接受任何 VARCHAR 值作为输入，并返回其值为输入字符串的 JSON 串：

SELECT CAST(not_json AS JSON); -- JSON '"not_json"'

SELECT CAST('["a": 1, "b": 2]' AS JSON); -- JSON '"["a": 1, "b": 2]"'

SELECT CAST('[1, 2, 3]' AS JSON); -- JSON '"[1, 2, 3]"'

SELECT CAST('"abc"' AS JSON); -- JSON '""abc""'

SELECT CAST('42' AS JSON); -- JSON '"42"'

SELECT CAST('true' AS JSON); -- JSON '"true"'

SELECT CAST('null' AS JSON); -- JSON '"null"'

- json_size(json, json_path)→ bigint

与json_extract()类似，但返回值的大小。对于对象或数组，该大小为成员数量，标量值的大小为 0：

```
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
```

1.6.13 日期和时间函数和运算符

日期和时间运算符

运算符	示例	结果
+	date '2021-08-08' + interval '2' day	2021-08-10
+	time '01:00' + interval '3' hour	04:00:00.000
+	timestamp '2021-08-08 01:00' + interval '29' hour	2021-08-09 06:00:00.000
+	timestamp '2021-10-31 01:00' + interval '1' month	2021-11-30 01:00:00.000
+	interval '2' day + interval '3' hour	2 03:00:00.000
+	interval '3' year + interval '5' month	3-5
-	date '2021-08-08' - interval '2' day	2021-08-06
-	time '01:00' - interval '3' hour	22:00:00.000
-	timestamp '2021-08-08 01:00' - interval '29' hour	2021-08-06 20:00:00.000
-	timestamp '2021-10-31 01:00' - interval '1' month	2021-09-30 01:00:00.000

运算符	示例	结果
-	interval '2' day - interval '3' hour	1 21:00:00.000
-	interval '3' year - interval '5' month	2-7

1.6.13.1 时区转换

运算符：AT TIME ZONE，用于设置时间戳的时区：

```
SELECT timestamp '2021-10-31 01:00 UTC';
-- 2021-10-31 01:00:00.000 UTC

SELECT timestamp '2021-10-31 01:00 UTC' AT TIME ZONE 'Asia/Shanghai';
-- 2021-10-30 18:00:00.000 Asia/Shanghai
```

1.6.13.2 日期时间函数

- current_date -> date

返回查询开始时的当前日期。

```
select current_date; -- 2021-10-30
```

- current_time -> time with time zone

返回查询开始时带时区的当前时间。

```
select current_time; -- 20:38:35.205-07:00
```

- current_timestamp -> timestamp with time zone

返回查询开始时带时区的当前时间戳。具有3亚秒精度的数字。

```
select current_timestamp; -- 2021-10-30 20:39:49.572 Asia/Shanghai
```

- current_timestamp(p)-> timestamp with time zone

返回查询开始时带时区的当前时间戳。具有p亚秒精度的数字。

```
select current_timestamp; -- 2021-10-30 20:39:49.34 Asia/Shanghai
```

- current_timezone()-> varchar

以 IANA 定义的格式（如Asia/Shanghai）或相对于 UTC 的固定偏移量（如+08:35）返回当前时区。

```
select current_timezone(); -- Asia/Shanghai
```

- date(x)-> date

这是CAST(x AS date)的别名。

- last_day_of_month(x)-> date

返回该月的最后一天。

```
select last_day_of_month(date '2021-10-30'); -- 2021-10-31
```

- from_iso8601_timestamp(string)-> timestamp(3)with time zone

将以 ISO 8601 格式表示的string（可选时间和时区）解析为timestamp(3)with time zone。时间默认为00:00:00.000，时区默认为会话时区：

```
SELECT from_iso8601_timestamp('2020-05-11');
  -- 2020-05-11 00:00:00.000 Asia/Shanghai

SELECT from_iso8601_timestamp('2020-05-11T11:15:05');
  -- 2020-05-11 11:15:05.000 Asia/Shanghai
```

- from_iso8601_timestamp_nanos(string)-> timestamp(9)with time zone

解析 ISO 8601 格式的日期和时间string。时区默认为会话时区：

```
SELECT from_iso8601_timestamp_nanos('2020-05-11T11:15:05');
  -- 2020-05-11 11:15:05.000000000 Asia/Shanghai
```

- from_iso8601_date(string)-> date

解析 ISO 8601 格式的日期string为date。该日期可以是日历日期、使用 ISO 周编号的周日期，或者是年份和年份中的某一天的组合：

```
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
```

- at_timezone(timestamp, zone)-> timestamp(p)with time zone

将时区分量timestamp精确地更改p为 zone同时保留时间戳。

```
select at_timezone(timestamp '2020-07-22 15:00:15', '+8'); -- 2020-07-21
    ↪ 23:00:15 -08:00
```

- with_timezone(timestamp, zone)-> timestamp(p)with time zone

返回带有时区的时间戳timestamp精确到p和zone。

```
select at_timezone(timestamp '2020-07-22 15:00:15', '+8'); -- 2020-07-22
    ↪ 15:00:15 +08:00
```

- from_unixtime(unixtime)-> timestamp(3)with time zone

将 UNIX 时间戳unixtime作为带时区的时间戳返回。unixtime是从1970-01-01 00:00:00开始经历的秒数。

```
select from_unixtime(1635523200); -- 2021-10-30 00:00:00.000 Asia/Shanghai
```

- from_unixtime(unixtime, zone)-> timestamp(3)with time zone

将 UNIX 时间戳unixtime作为带时区的时间戳返回，其中使用zone作为时区。unixtime是从1970-01-01
↪ 00:00:00开始经历的秒数。

```
select from_unixtime(1635523200, '-8'); -- 2021-10-29 08:00:00.000 -08:00
```

- from_unixtime(unixtime, hours, minutes)-> timestamp(3)with time zone

将 UNIX 时间戳unixtime作为带时区的时间戳返回，其中使用hours和minutes作为时区偏移量。unixtime是从1970-01-01 00:00:00开始经历的秒数。

```
select from_unixtime(1635523200, +8, +1); -- 2021-10-30 00:01:00.000 +08:01
```

- from_unixtime_nanos(unixtime)-> timestamp(9)with time zone

将 UNIX 时间戳unixtime作为带时区的时间戳返回。unixtime是从1970-01-01 00:00:00开始经历的秒数:

```
SELECT from_unixtime_nanos(1635523200); --  1970-01-01 08:00:01.635523200 Asia/
    ↪ Shanghai

SELECT from_unixtime_nanos(DECIMAL '1234'); -- 1970-01-01 08:00:00.000001234
    ↪ Asia/Shanghai

SELECT from_unixtime_nanos(DECIMAL '1234.499'); -- 1970-01-01 00:00:00.000001234
    ↪  Asia/Shanghai

SELECT from_unixtime_nanos(DECIMAL '-1234'); -- 1969-12-31 23:59:59.999998766
    ↪ Asia/Shanghai
```

- localtime -> time

返回查询开始时的当前时间。

```
select localtime; -- 14:11:05.225
```

- localtimestamp -> timestamp

返回查询开始时的当前时间戳。具有3亚秒精度的数字。

```
select localtimestamp; -- 2021-10-30 14:11:34.981
```

- localtimestamp(p)-> timestamp

返回查询开始时的当前时间戳。具有p亚秒精度的数字:

```
SELECT localtimestamp(6); -- 2020-10-30 14:14:28.235548
```

- now()→ timestamp(3)with time zone

这是current_timestamp()的别名。

- to_iso8601(x)→ varchar

将x格式化为 ISO 8601 字符串。x可以是 date、timestamp 或 timestamp with time zone。

```
select to_iso8601(date '2020-07-25'); -- 2020-07-25
select to_iso8601(timestamp '2020-07-25 15:22:15.214'); -- 2020-07-25T15
    ↪ :22:15.214+08:00
```

- to_milliseconds(interval)→ bigint

以毫秒为单位返回以天和秒为单位的间隔interval。

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

- to_unixtime(timestamp)→ double

将timestamp作为 UNIX 时间戳返回。

```
select to_unixtime(cast('2020-10-30 14:32:15.147' as timestamp)); --
    ↪ 1.604039535147E9
```

> **注意**
>
> 以下 SQL 标准函数不使用括号:
>
> - current_date
> - current_time
> - current_timestamp
> - localtime
> - localtimestamp

1.6.13.3 date_trunc 截断函数

date_trunc函数支持以下单位:

单位	截断值示例
second	2021-08-22 03:04:05.000
minute	2021-08-22 03:04:00.000
hour	2021-08-22 03:00:00.000
day	2021-08-22 00:00:00.000
week	2021-08-20 00:00:00.000
month	2021-08-01 00:00:00.000
quarter	2021-07-01 00:00:00.000
year	2021-01-01 00:00:00.000

上面的示例使用时间戳2021-08-22 03:04:05.321作为输入。

- date_trunc(unit, x)-> 与输入类型相同

返回x截断至unit后的值:

```
SELECT date_trunc('day' , TIMESTAMP '2022-10-20 05:10:00'); -- 2022-10-20
    ↪ 00:00:00.000

SELECT date_trunc('month' , TIMESTAMP '2022-10-20 05:10:00'); -- 2022-10-01
    ↪ 00:00:00.000

SELECT date_trunc('year', TIMESTAMP '2022-10-20 05:10:00'); -- 2022-01-01
    ↪ 00:00:00.000
```

1.6.13.4 间隔函数

本节中的函数支持以下间隔单位：

单位	说明
millisecond	毫秒
second	秒
minute	分钟
hour	小时
day	天
week	周
month	月
quarter	季度
year	年

- date_add(unit, value, timestamp) -> 与输入类型相同

向timestamp添加类型为unit的间隔value。可以使用负值做减法。

```
SELECT date_add('second', 86, TIMESTAMP '2020-03-01 00:00:00'); -- 2020-03-01
    ↪ 00:01:26.000

SELECT date_add('hour', 9, TIMESTAMP '2020-03-01 00:00:00'); -- 2020-03-01
    ↪ 09:00:00.000

SELECT date_add('day', -1, TIMESTAMP '2020-03-01 00:00:00 UTC'); -- 2020-02-29
    ↪ 00:00:00.000 UTC
```

- date_diff(unit, timestamp1, timestamp2)-> 与输入类型相同

以unit为单位返回timestamp2 - timestamp1的值。

```
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 12:30:45.000000000', TIMESTAMP
    ↪ '2020-06-02 12:30:45.123456789');
  -- 86400

SELECT date_diff('millisecond', TIMESTAMP '2020-06-01 12:30:45.000000000',
    ↪ TIMESTAMP '2020-06-02 12:30:45.123456789');
  -- 86400123
```

1.6.13.5 parse_duration 持续时间函数

parse_duration函数支持以下单位：

单位	说明
ns	纳秒
us	微秒
ms	毫秒
s	秒
m	分钟
h	小时
d	天

- parse_duration(string)→ interval

将格式为value unit的string解析为一个区间，其中value是以unit为单位的小数值：

```
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
```

- human_readable_seconds(double)→ varchar

将双精度值seconds格式化为可读的字符串，其中包含weeks, days, hours, minutes，及seconds：

```
SELECT human_readable_seconds(96); -- 1 minute, 36 seconds

SELECT human_readable_seconds(3762); -- 1 hour, 2 minutes, 42 seconds

SELECT human_readable_seconds(56363463); -- 93 weeks, 1 day, 8 hours, 31 minutes
    ↪ , 3 seconds
```

1.6.13.6 MySQL 日期函数

该部分中的函数使用与 MySQL date_parse和str_to_date函数兼容的格式字符串。下表根据 MySQL 手册说明了格式说明符：

说明符	说明
%a	工作日简称 (Sun …Sat)
%b	月份简称 (Jan …Dec)
%c	以数字表示的月份 (1 …12) [4]
%D	带英文后缀的一个月中的第几日（0th、1st、2nd、3rd、…）
%d	以数字表示的一个月中的第几日 (01 …31) [4]
%e	以数字表示的一个月中的第几日 (1 …31) [4]
%f	微秒（打印 6 位：000000 …999000；解析 1–9 位：0 …999999999）[1]
%H	小时 (00 …23)
%h	小时 (01 …12)
%I	小时 (01 …12)

说明符	说明
%i	以数字表示的分钟 (00 …59)
%j	一年中的某日 (001 …366)
%k	小时 (0 …23)
%l	小时 (1 …12)
%M	月份名称 (January …December)
%m	以数字表示的月份 (01 …12) [4]
%p	AM或PM
%r	12 小时制时间（hh:mm:ss，后跟AM或PM）
%S	秒 (00 …59)
%s	秒 (00 …59)
%T	24 小时制时间 (hh:mm:ss)
%U	周 (00 …53)，其中星期日为一周中的第一天
%u	周 (00 …53)，其中星期一为一周中的第一天
%V	周 (01 …53)，其中星期日为一周中的第一天；与%X配合使用
%v	周 (01 …53)，其中星期一为一周中的第一天；与%x配合使用
%W	周日名称 (Sunday …Saturday)
%w	星期几 (0 …6)，其中星期日是一周中的第一天 [3]
%X	周所在的年份，其中星期日是一周中的第一天，以四位数字表示，与%V配合使用
%x	周所在的年份，其中星期一是一周中的第一天，以四位数字表示，与%v配合使用
%Y	年份，以四位数字表示
%y	年份，以两位数字表示 [2]
%%	%字符
%x	x，用于上面未列出的任何x

1. 时间戳被截断至毫秒。
2. 在进行解析时，两位数的年份格式采用的范围为1970至2069，因此“70”将产生年份1970，但“69”将产生2069。
3. 尚不支持此说明符。考虑使用day_of_week()（该函数使用1-7，而不使用0-6）。
4. ([1], [2], [3], [4]) 该说明符不支持使用0作为月份或日。

> **警告**
>
> 当前不支持以下说明符：%D %U %u %V %w %X

- date_format(timestamp, format)→ varchar

使用format将timestamp格式化为字符串：

```
SELECT date_format(TIMESTAMP '2022-10-20 05:10:00', '%m-%d-%Y %H'); --
    ↪ 10-20-2022 05
```

- date_parse(string, format)→ timestamp

使用format将string解析为时间戳：

```
SELECT date_parse('2022/10/20/05', '%Y/%m/%d/%H'); -- 2022-10-20 05:00:00.000
SELECT date(date_parse('20221021', '%Y%m%d'));-- 2022-10-21
```

1.6.13.7 extract 提取函数

extract函数支持以下字段:

字段	说明
YEAR	year()
QUARTER	quarter()
MONTH	month()
WEEK	week()
DAY	day()
DAY_OF_MONTH	day()
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()

extract函数支持的类型因要提取的字段而异。大多数字段都支持所有日期和时间类型。

- extract(field FROM x)→ bigint

返回field自x:

```
SELECT extract(YEAR FROM TIMESTAMP '2022-10-20 05:10:00'); -- 2022
```

> **注意**
>
> 该 SQL 标准函数使用特殊的语法来指定参数。

1.6.13.8 便捷提取函数

- day(x)→ bigint

从x返回一个月中的第几日。

```
select day(date '2022-10-20'); -- 20
```

- day_of_month(x)→ bigint

这是day(x)的别名。

- day_of_week(x)→ bigint

从x返回星期几 (ISO)。值的范围为1（星期一）至7（星期日）。

```
select day_of_week(date '2022-10-20'); -- 4
```

- day_of_year(x)→ bigint

从x返回一年中的第几日。值的范围为1至366。

```
select day_of_year(date '2022-10-20'); -- 293
```

- dow(x)→ bigint

从x返回星期几 (ISO)。值的范围为1（星期一）至7（星期日）。

这是day_of_week()的别名。

```
select dow(date '2022-10-20'); -- 4
```

- doy(x)→ bigint

从x返回一年中的第几日。值的范围为1至366。

这是day_of_year()的别名。

```
select doy(date '2022-10-20'); -- 293
```

- hour(x)→ bigint

从x返回一天中的第几个小时。值的范围为0至23。

```
select hour(timestamp '2022-10-20 05:10:15.100'); -- 5
```

- millisecond(x)→ bigint

从x返回一秒中的第几个毫秒。

```
select millisecond(timestamp '2022-10-20 05:10:15.100'); -- 100
```

- minute(x)→ bigint

从x返回一小时中的第几分钟。

```
select minute(timestamp '2022-10-20 05:10:15.100'); -- 10
```

- month(x)→ bigint

从x返回一年中的某个月份。

```
select month(timestamp '2022-10-20 05:10:15.100'); -- 10
```

- quarter(x)→ bigint

从x返回一年中的某个季度。值的范围为1至4。

```
select quarter(timestamp '2022-10-20 05:10:15.100'); -- 4
```

- second(x)→ bigint

从x返回一分中的第几秒。

```
select second(timestamp '2022-10-20 05:10:15.100'); -- 15
```

- timezone_hour(timestamp)→ bigint

从timestamp返回时区偏移量的小时数。

```
select timezone_hour(timestamp '2022-10-20 05:10:15.100'); -- 8
```

- timezone_minute(timestamp)→ bigint

从timestamp返回时区偏移量的分钟数。

```
select timezone_minute(timestamp '2021-10-30 00:01:00.000 +08:01'); -- 1
```

- week(x)→ bigint

从x返回一年中的第几周 (ISO)。值的范围为1至53。

```
select week(timestamp '2022-10-20 05:10:15.100'); -- 42
```

- week_of_year(x)→ bigint

这是week()的别名。

```
select week_of_year(timestamp'2022-10-20 05:10:15.100');  --  42
```

- year(x)→ bigint

从x返回 ISO 周的年份。

```
select year(timestamp '2022-10-20 05:10:15.100'); -- 2022
```

1.6.14 聚合函数

聚合函数对一组值进行操作以计算得出单个结果。

除了 count()、count_if()、max_by()、min_by() 和 approx_distinct() 之外，所有这些聚合函数都忽略空值，并且对于没有输入行或所有值都是 NULL 的情况都返回 NULL。例如，sum() 返回 NULL 而不是 0，avg() 在计数中不包括 NULL 值。可以使用 coalesce 函数将 NULL 转换为 0。

1.6.14.1 聚合期间排序

某些聚合函数array_agg()会根据输入值的顺序产生不同的结果。可以通过在聚合函数中编写 ORDER BY 子句来指定此顺序：

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

1.6.14.2 聚合期间过滤

关键字FILTER可用于从聚合处理中删除具有使用WHERE子句表达的条件的行。在聚合中之前会对每一行进行评估，并且所有聚合函数都支持它。

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

示例数据

```
--建表
create table fruit (num integer, name varchar, price integer);
--插入数据
insert into fruit values (1,'peach',5), (2,'apple',2), (3,'orange',6), (4,'
    ↪ watermelon',10);
```

```
--建表
create table sample_collection(id integer, time_cost integer, weight decimal
    ↪ (5,2));
--插入数据
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);
```

1.6.14.3 一般聚合函数

- arbitrary(x)→ [same as input]

返回 x 的任意非 NULL 值（如果存在）。

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

- array_agg(x)→ array<[same as input]>

返回通过输入 x 元素创建的数组。

```
select array_agg(price) from fruit; -- [5, 2, 6, 10]
```

- avg(x)→ double

返回所有输入值的平均值（算术平均值）。

```
select avg(price) from fruit; -- 5.75
```

- avg(time interval type)→ time interval type

返回所有输入值的平均间隔长度。

```
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)→ boolean

如果所有输入值都为 TRUE 返回 TRUE，否则返回 FALSE。

```
select bool_and(isorno) from (values ('01',true), ('02',false)) as items(num,
    ↪ isorno); -- false
select bool_and(isorno) from (values ('01',true), ('02',true)) as items(num,
    ↪ isorno); -- true
```

- bool_or(boolean)→ boolean

如果任一输入值为 TRUE，则返回 TRUE，否则返回 FALSE。

```
select bool_or(isorno) from (values ('01',false), ('02',false)) as items(num,
    ↪ isorno); -- false
select bool_or(isorno) from (values ('01',true), ('02',false)) as items(num,
    ↪ isorno); -- true
```

- checksum(x)→ varbinary

返回给定值的不区分顺序的校验和。

```
select checksum(price) from fruit; -- 7e 16 c0 1c 24 1c d6 4a
```

- count(*)→ bigint

返回输入行的数量。

```
select count(*) from fruit; -- 4
```

- count(x)→ bigint

返回非 NULL 输入值的数量。

```
select count(name) from fruit; -- 4
```

- count_if(x)→ bigint

返回 TRUE 输入值的数量。该函数等价于 count(CASE WHEN x THEN 1 END)。

```
select count_if(price > 7) from fruit; -- 1
```

- every(boolean)→ boolean

  bool_and() 的别名。

- geometric_mean(x)→ double

返回所有输入值的几何平均值。

```
select geometric_mean(price) from fruit; -- 4.949232003839765
```

- listagg(x, separator)→ varchar

返回串联的输入值，由separator字符串分隔。

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

如果separator未指定，则空字符串将用作separator。

该函数的最简单形式如下所示：

```
SELECT listagg (name, ',') WITHIN GROUP (ORDER BY name) from fruit;
  -- apple,orange,peach,watermelon
```

如果函数输出的长度超过1048576字节，溢出行为默认会抛出错误：

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

在函数输出的长度超过1048576字节的情况下，还存在截断输出的可能性WITH COUNT或者WITHOUT COUNT：

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

如果未指定，则截断填充字符串默认为'...'。

这个聚合函数也可以用在涉及到分组的场景中：

```
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
```

当前LISTAGG函数的实现不支持窗口框架。

- max(x) → [与输入相同]

返回所有输入值中的最大值。

```
select max(price) from fruit; -- 10
```

- max(x, n)→ array<[与 x 相同]>

返回 x 的所有输入值中的前 n 个最大值。

```
select max(price, 2) from fruit; -- [10, 6]
```

- max_by(x, y)→ [与 x 相同]

返回与所有输入值中 y 的最大值相关联的 x 值。

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

- max_by(x, y, n)→ array<[与 x 相同]>

返回与 y 降序排列时 y 的所有输入值中前 n 个最大值相关联的 n 个 x 值。

```
select max_by(name, price, 2) from fruit; -- [watermelon, orange]
```

- min(x)→ [与输入相同]

返回所有输入值中的最小值。

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

- min(x, n)→ array<[与 x 相同]>

返回 x 的所有输入值中的前 n 个最小值。

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

- min_by(x, y)→ [与 x 相同]

返回与所有输入值中 y 的最小值相关联的 x 值。

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

- min_by(x, y, n)→ array<[与 x 相同]>

返回与 y 升序排列时 y 的所有输入值中前 n 个最小值相关联的 n 个 x 值。

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

- sum(x)→ [与输入相同]

返回所有输入值的总和。

```
select sum(price) from fruit; -- 23
```

1.6.14.4 按位聚合函数

- bitwise_and_agg(x)-> bigint

返回以二进制补码表示的所有输入值的按位与运算结果。

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

- bitwise_or_agg (x)-> bigint

返回以二进制补码表示的所有输入值的按位或运算结果。

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

1.6.14.5 映射聚合函数

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

返回一个映射，其中包含每个输入值出现的次数。

```
select histogram(x) from (values (15),(10),(21),(15),(15),(8),(7),(21)) as t(x);
  -- {8=1, 10=1, 21=2, 7=1, 15=3}
```

- map_agg(key, value)-> map(K, V)

返回通过输入 key/value 对创建的映射。

```
select map_agg(name,price) from fruit; -- {orange=6, peach=5, apple=2,
    ↪ watermelon=10}
```

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

返回所有输入映射的并集。如果在多个输入映射中找到某个键，则生成的映射中该键的值来自任意输入映射。

```
select map_union(x) from (values (map(array['banana'],array[10.0])), (map(array[
    ↪ 'banana'],array[2.0])), (map(array['apple'],array[7.0]))) as t(x); -- {
    ↪ banana=10.0, apple=7.0}
```

- multimap_agg(key, value)-> map(K, array(V))

返回通过输入 key/value 对创建的多重映射。每个键可以关联多个值。

```
select multimap_agg(key, value) from (values ('banana',10),('banana',8),('banana
    ↪ ',8),('apple',5) ) as t(key,value); -- {banana=[10, 8, 8], apple=[5]}
```

1.6.14.6 近似聚合函数

- approx_distinct(x)→ bigint

返回不重复输入值的近似数量。本函数给出 count(DISTINCT x) 的近似值。如果所有输入值都为空则返回 0。

本函数会产生 2.3% 的误差，（近似正态）误差分布的标准偏差会覆盖全部数据集。对于任意指定的输入，不保证误差上限。

```
select approx_distinct(price) from fruit; -- 4
```

- approx_distinct(x, e)→ bigint

返回不重复输入值的近似数量。本函数给出 count(DISTINCT x) 的近似值。如果所有输入值都为空则返回 0。

本函数会产生不超过 e 的误差，（近似正态）误差分布的标准偏差会覆盖全部数据集。对于任意指定的输入，不保证误差上限。目前的函数实现要求 e 在 [0.01150, 0.26000] 范围之间。

```
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([与 value 相同], bigint)

近似计算最多元素buckets的最高频繁值。函数的近似估计使我们能够以较少的内存获取频繁的值。更大的capacity在牺牲内存容量的情况下提高了底层算法的准确性。返回值是一个映射，其中包含具有相应估计频率的顶部元素。

函数的误差取决于值的排列及其基数。我们可以将容量设置为与底层数据的基数相同，以实现最少的错误。

- buckets和capacity必须是bigint类型。value可以是数字或字符串类型。

该函数使用 A. Metwalley、D. Agrawl 和 A. Abbadi 在论文 Efficient Computation of Frequent and Top-k Elements in Data Streams 中提出的流汇总数据结构。

- approx_percentile(x, percentage)→ [与 x 相同]

返回在给定 percentage 时 x 的所有输入值的近似百分位数。percentage 值必须介于 0 和 1 之间，并且对于所有输入行是一个常量。

```
select approx_percentile(x, 0.5) from (values (2), (3), (7), (8), (9)) as t(x);
    ↪ --7
```

- approx_percentile(x, percentages)→ array([与 x 相同])

返回 x 的所有输入值在每个指定百分比处的近似百分位数。percentages 数组的每个元素必须介于 0 和 1 之间，并且该数组对于所有输入行必须是一个常量。

```
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)→ [与 x 相同]

返回在百分比 p 处 x 的所有输入值（使用每项权重 w）的近似加权百分位数。权重必须是一个整数值，最小为 1。它实际上是百分位集中值 x 的重复计数。p 值必须介于 0 和 1 之间，并且对于所有输入行是一个常量。

```
select approx_percentile(x, 5, 0.5) from (values (2), (3), (7), (8), (9)) as t(x
    ↪ ); -- 7
```

- approx_percentile(x, w, percentages)→ array<[与 x 相同]>

返回在数组中指定的每个给定百分比处 x 的所有输入值（使用每项权重 w）的近似加权百分位数。权重必须是一个整数值，最小为 1。它实际上是百分位集中值 x 的重复计数。数组的每个元素必须介于 0 和 1 之间，并且该数组对于所有输入行必须是一个常量。

```
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]
```

- numeric_histogram(buckets, value)→ map<double, double>

  计算所有 value 的近似直方图（存储桶的数量达 buckets 个）。该函数相当于 numeric_histogram() 的变体，接受 weight，每项权重为 1。

```
select numeric_histogram(20, x) from ( values (2), (3), (7), (8), (9)) as t(x);
  -- {2.0=1.0, 3.0=1.0, 7.0=1.0, 8.0=1.0, 9.0=1.0}
```

- numeric_histogram(buckets, value, weight)→ map<double, double>

  计算所有 value（每项权重为 weight）的近似直方图（存储桶的数量达 buckets 个）。算法大致基于：

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

```
select numeric_histogram(20, x, 4) from (values (2), (3), (7), (8), (9)) as t(x)
    ↪ ;
  -- {2.0=4.0, 3.0=4.0, 7.0=4.0, 8.0=4.0, 9.0=4.0}
```

buckets 必须是 bigint。value 和weight 必须是数字。

1.6.14.7 统计聚合函数

- corr(y, x)→ double

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)→ double

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)→ double

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)→ double

返回所有输入值的超值峰度。使用以下表达式的无偏估计:

```
kurtosis(x) = n(n+1)/((n-1)(n-2)(n-3))sum[(x_i-mean)^4]/stddev(x)^4-3(n-1)^2/((n
    ↪ -2)(n-3))
```

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

- regr_intercept(y, x)→ double

返回输入值的线性回归截距。y 为非独立值。x 为独立值。

```
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)→ double

返回输入值的线性回归斜率。y 为非独立值。x 为独立值。

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

- skewness(x)→ double

返回所有输入值的偏度。

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

- stddev(x)→ double

这是 stddev_samp() 的别名。

- stddev_pop(x)→ double

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)→ double

返回所有输入值的样本标准差。

```
select stddev_samp(x) from (values (1), (2), (3), (4)) as t(x); --
    ↪ 1.2909944487358056
```

- variance(x)→ double

这是 var_samp() 的别名。

- var_pop(x)→ double

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)→ double

返回输入值的样本方差。

```
select var_samp(x) from (values (1), (2), (3), (4)) as t(x); --
    ↪ 1.6666666666666667
```

1.6.14.8 lambda 聚合函数

- reduce_agg(inputValue T , initialState S , inputFunction(S , T , S), combineFunction(S
  ↪ , S , S))→ S

将所有输入值缩减为单个值。会为每个非 NULL 输入值调用 inputFunction。除了接受输入值之外，inputFunction 还接受当前状态（最初为 initialState）并返回新状态。会调用 combineFunction 将两个状态合并成一个新的状态。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)
```

状态类型必须是布尔型、整型、浮点型或日期/时间/间隔型。

1.6.15 窗口函数

窗口函数对查询结果中的行执行计算。窗口函数在HAVING子句之后、ORDER BY子句之前运行。调用窗口函数需要使用特殊的语法（使用OVER子句来指定窗口）。

例如，以下查询按价格对每个售货员的订单进行排序：

```
SELECT orderkey, clerk, totalprice,
       rank() OVER (PARTITION BY clerk
                    ORDER BY totalprice DESC) AS rnk
FROM orders
ORDER BY clerk, rnk
```

1.6.15.1 聚合函数

通过添加OVER子句，可以将所有aggregate用作窗口函数。会为当前行的窗口框架中的每个行计算聚合函数。

例如，以下查询为每个售货员生成按天滚动的订单价格总和：

```
SELECT clerk, orderdate, orderkey, totalprice,
       sum(totalprice) OVER (PARTITION BY clerk
                             ORDER BY orderdate) AS rolling_sum
FROM orders
ORDER BY clerk, orderdate, orderkey
```

示例数据

为便于理解各函数的使用方法，本文为您提供源数据，基于源数据提供函数相关示例。创建表 emp，并添加数据，命令示例如下：

```
create table if not exists emp
   (empno integer,
```

```
    ename varchar,
    job varchar,
    mgr integer,
    hiredate TIMESTAMP,
    sal integer,
    comm integer,
    deptno integer)
    WITH (
        format = 'MULTIDELIMIT',
        textfile_field_separator = ','
    );
load data local inpath 'emp.txt' into table emp;
```

emp.txt 中的数据如下：

```
7369,SMITH,CLERK,7902,1980-12-17 00:00:00,800,,20
7499,ALLEN,SALESMAN,7698,1981-02-20 00:00:00,1600,300,30
7521,WARD,SALESMAN,7698,1981-02-22 00:00:00,1250,500,30
7566,JONES,MANAGER,7839,1981-04-02 00:00:00,2975,,20
7654,MARTIN,SALESMAN,7698,1981-09-28 00:00:00,1250,1400,30
7698,BLAKE,MANAGER,7839,1981-05-01 00:00:00,2850,,30
7782,CLARK,MANAGER,7839,1981-06-09 00:00:00,2450,,10
7788,SCOTT,ANALYST,7566,1987-04-19 00:00:00,3000,,20
7839,KING,PRESIDENT,,1981-11-17 00:00:00,5000,,10
7844,TURNER,SALESMAN,7698,1981-09-08 00:00:00,1500,0,30
7876,ADAMS,CLERK,7788,1987-05-23 00:00:00,1100,,20
7900,JAMES,CLERK,7698,1981-12-03 00:00:00,950,,30
7902,FORD,ANALYST,7566,1981-12-03 00:00:00,3000,,20
7934,MILLER,CLERK,7782,1982-01-23 00:00:00,1300,,10
7948,JACCKA,CLERK,7782,1981-04-12 00:00:00,5000,,10
7956,WELAN,CLERK,7649,1982-07-20 00:00:00,2450,,10
7956,TEBAGE,CLERK,7748,1982-12-30 00:00:00,1300,,10
```

1.6.15.2 排序函数

- cume_dist()→ bigint

返回一组值中某个值的累积分布。结果是窗口分区的窗口排序中的行前面或与之对等的行数量除以窗口分区中的总行数。因此，排序中的任何绑定值都将计算为相同的分布值。

例如：将所有职工根据部门（deptno）分组（作为开窗列），计算薪水（sal）在同一组内的前百分之几。

```
select
      deptno,
      ename,
      sal,
      round(cume_dist() over (partition by deptno order by sal desc)* 100, 2) as
          ↪ cume_dist
  from emp order by deptno;
```

返回结果如下：

```
 deptno | ename  | sal  | cume_dist
--------+--------+------+-----------
     10 | KING   | 5000 |     33.33
     10 | JACCKA | 5000 |     33.33
     10 | CLARK  | 2450 |     66.67
     10 | WELAN  | 2450 |     66.67
     10 | MILLER | 1300 |     100.0
     10 | TEBAGE | 1300 |     100.0
     20 | SCOTT  | 3000 |      40.0
     20 | FORD   | 3000 |      40.0
     20 | JONES  | 2975 |      60.0
     20 | ADAMS  | 1100 |      80.0
     20 | SMITH  |  800 |     100.0
     30 | BLAKE  | 2850 |     16.67
     30 | ALLEN  | 1600 |     33.33
     30 | TURNER | 1500 |      50.0
     30 | WARD   | 1250 |     83.33
     30 | MARTIN | 1250 |     83.33
     30 | JAMES  |  950 |     100.0
```

- dense_rank()→ bigint

返回某个值在一组值中的排名。这与rank()类似，只是绑定值不在序列中产生间隙。例如：将所有职工根据部门（deptno）分组（作为开窗列），每个组内根据薪水（sal）做降序排序，获得职工自己组内的序号。

```
select
      deptno,
      ename,
      sal,
      dense_rank() over (partition by deptno order by sal desc) as nums
  from emp order by deptno;
```

返回结果如下：

```
 deptno | ename  | sal  | nums
--------+--------+------+------
     10 | KING   | 5000 |    1
     10 | JACCKA | 5000 |    1
     10 | CLARK  | 2450 |    2
     10 | WELAN  | 2450 |    2
     10 | MILLER | 1300 |    3
     10 | TEBAGE | 1300 |    3
     20 | SCOTT  | 3000 |    1
     20 | FORD   | 3000 |    1
     20 | JONES  | 2975 |    2
     20 | ADAMS  | 1100 |    3
     20 | SMITH  |  800 |    4
```

```
      30 | BLAKE  | 2850 |    1
      30 | ALLEN  | 1600 |    2
      30 | TURNER | 1500 |    3
      30 | WARD   | 1250 |    4
      30 | MARTIN | 1250 |    4
      30 | JAMES  |  950 |    5
```

- ntile(n)→ bigint

  将每个窗口分区的行分到n个桶中，范围为1至n（最大）。桶值最多相差1。如果分区中的行数未平均分成桶数，则从第一个桶开始每个桶分配一个剩余值。

  例如，对于6行和4个桶，桶值如下：1 1 2 2 3 4 。

  例如：将所有职工根据部门按薪水（sal）从高到低切分为 3 组，并获得职工自己所在组的序号。

```
select
      deptno,
      ename,
      sal,
      ntile(3) over (partition by deptno order by sal desc) as nt3
  from emp order by deptno;
```

返回结果如下：

```
  deptno | ename  | sal  | nt3
  -------+--------+------+-----
      10 | KING   | 5000 |   1
      10 | JACCKA | 5000 |   1
      10 | CLARK  | 2450 |   2
      10 | WELAN  | 2450 |   2
      10 | MILLER | 1300 |   3
      10 | TEBAGE | 1300 |   3
      20 | SCOTT  | 3000 |   1
      20 | FORD   | 3000 |   1
      20 | JONES  | 2975 |   2
      20 | ADAMS  | 1100 |   2
      20 | SMITH  |  800 |   3
      30 | BLAKE  | 2850 |   1
      30 | ALLEN  | 1600 |   1
      30 | TURNER | 1500 |   2
      30 | WARD   | 1250 |   2
      30 | MARTIN | 1250 |   3
      30 | JAMES  |  950 |   3
```

- percent_rank()→ double

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

  例如：计算员工薪水在组内的百分比排名。

```
select
      deptno,
      ename,
      sal,
      percent_rank() over (partition by deptno order by sal desc) as sal_new
  from emp order by deptno;
```

返回结果如下：

```
   deptno | ename  | sal  | sal_new
  --------+--------+------+---------
       10 | KING   | 5000 |     0.0
       10 | JACCKA | 5000 |     0.0
       10 | CLARK  | 2450 |     0.4
       10 | WELAN  | 2450 |     0.4
       10 | MILLER | 1300 |     0.8
       10 | TEBAGE | 1300 |     0.8
       20 | SCOTT  | 3000 |     0.0
       20 | FORD   | 3000 |     0.0
       20 | JONES  | 2975 |     0.5
       20 | ADAMS  | 1100 |    0.75
       20 | SMITH  |  800 |     1.0
       30 | BLAKE  | 2850 |     0.0
       30 | ALLEN  | 1600 |     0.2
       30 | TURNER | 1500 |     0.4
       30 | WARD   | 1250 |     0.6
       30 | MARTIN | 1250 |     0.6
       30 | JAMES  |  950 |     1.0
```

- rank()→ bigint

  返回某个值在一组值中的排序。排序是该行之前与该行不对等的行数加一。因此，排序中的绑定值将在序列中产生间隙。会对每个窗口分区进行排序。

  例如：将所有职工根据部门（deptno）分组（作为开窗列），每个组内根据薪水（sal）做降序排序，获得职工自己组内的序号。

```
select
      deptno,
      ename,
      sal,
      rank() over (partition by deptno order by sal desc) as nums
  from emp order by deptno;
```

返回结果如下：

```
   deptno | ename  | sal  | nums
  --------+--------+------+------
       10 | KING   | 5000 |    1
```

```
      10 | JACCKA | 5000 |    1
      10 | CLARK  | 2450 |    3
      10 | WELAN  | 2450 |    3
      10 | MILLER | 1300 |    5
      10 | TEBAGE | 1300 |    5
      20 | SCOTT  | 3000 |    1
      20 | FORD   | 3000 |    1
      20 | JONES  | 2975 |    3
      20 | ADAMS  | 1100 |    4
      20 | SMITH  |  800 |    5
      30 | BLAKE  | 2850 |    1
      30 | ALLEN  | 1600 |    2
      30 | TURNER | 1500 |    3
      30 | WARD   | 1250 |    4
      30 | MARTIN | 1250 |    4
      30 | JAMES  |  950 |    6
```

- row_number()→ bigint

  根据窗口分区内行的排序，从 1 开始返回每行的唯一序列号。

  例如：将所有职工根据部门（deptno）分组（作为开窗列），每个组内根据薪水（sal）做降序排序，获得职工在自己组内的序号。

```
select
      deptno,
      ename,
      sal,
      row_number() over (partition by deptno order by sal desc) as nums
  from emp order by deptno;
```

返回结果如下：

```
   deptno | ename  | sal  | nums
  --------+--------+------+------
      10 | KING   | 5000 |    1
      10 | JACCKA | 5000 |    2
      10 | CLARK  | 2450 |    3
      10 | WELAN  | 2450 |    4
      10 | MILLER | 1300 |    5
      10 | TEBAGE | 1300 |    6
      20 | SCOTT  | 3000 |    1
      20 | FORD   | 3000 |    2
      20 | JONES  | 2975 |    3
      20 | ADAMS  | 1100 |    4
      20 | SMITH  |  800 |    5
      30 | BLAKE  | 2850 |    1
      30 | ALLEN  | 1600 |    2
      30 | TURNER | 1500 |    3
```

```
       30 | WARD   | 1250 |    4
       30 | MARTIN | 1250 |    5
       30 | JAMES  |  950 |    6
```

1.6.15.3 值函数

- first_value(x)→ [与输入相同]

返回窗口的第一个值。

例如：将所有职工根据部门分组，返回每组中的第一行数据。

- 不指定order by:

```
select
        deptno,
        ename,
        sal,
        first_value(sal) over (partition by deptno) as first_value
    from emp order by deptno;
```

返回结果如下:

```
     deptno | ename  | sal  | first_value
    --------+--------+------+-------------
         10 | CLARK  | 2450 |        2450
         10 | KING   | 5000 |        2450
         10 | MILLER | 1300 |        2450
         10 | JACCKA | 5000 |        2450
         10 | WELAN  | 2450 |        2450
         10 | TEBAGE | 1300 |        2450
         20 | SMITH  |  800 |         800
         20 | JONES  | 2975 |         800
         20 | SCOTT  | 3000 |         800
         20 | ADAMS  | 1100 |         800
         20 | FORD   | 3000 |         800
         30 | ALLEN  | 1600 |        1600
         30 | WARD   | 1250 |        1600
         30 | MARTIN | 1250 |        1600
         30 | BLAKE  | 2850 |        1600
         30 | TURNER | 1500 |        1600
         30 | JAMES  |  950 |        1600
```

- 指定order by:

```
select
        deptno,
        ename,
        sal,
        first_value(sal) over (partition by deptno order by sal desc) as
            ↪ first_value
```

```
    from emp order by deptno;
```

返回结果如下：

```
     deptno | ename  | sal  | first_value
    --------+--------+------+-------------
         10 | KING   | 5000 |        5000
         10 | JACCKA | 5000 |        5000
         10 | CLARK  | 2450 |        5000
         10 | WELAN  | 2450 |        5000
         10 | MILLER | 1300 |        5000
         10 | TEBAGE | 1300 |        5000
         20 | SCOTT  | 3000 |        3000
         20 | FORD   | 3000 |        3000
         20 | JONES  | 2975 |        3000
         20 | ADAMS  | 1100 |        3000
         20 | SMITH  |  800 |        3000
         30 | BLAKE  | 2850 |        2850
         30 | ALLEN  | 1600 |        2850
         30 | TURNER | 1500 |        2850
         30 | WARD   | 1250 |        2850
         30 | MARTIN | 1250 |        2850
         30 | JAMES  |  950 |        2850
```

- last_value(x)→ [与输入相同]

  返回窗口的最后一个值。

  例如：将所有职工根据部门分组，返回每组中的最后一行数据。

- 不指定order by，当前窗口为第一行到最后一行的范围，返回当前窗口的最后一行的值。：

```
select
        deptno,
        ename,
        sal,
        last_value(sal) over (partition by deptno) as last_value
    from emp order by deptno;
```

返回结果如下：

```
     deptno | ename  | sal  | last_value
    --------+--------+------+------------
         10 | CLARK  | 2450 |       1300
         10 | KING   | 5000 |       1300
         10 | MILLER | 1300 |       1300
         10 | JACCKA | 5000 |       1300
         10 | WELAN  | 2450 |       1300
         10 | TEBAGE | 1300 |       1300
         20 | SMITH  |  800 |       3000
         20 | JONES  | 2975 |       3000
```

```
        20 | SCOTT  | 3000 |        3000
        20 | ADAMS  | 1100 |        3000
        20 | FORD   | 3000 |        3000
        30 | ALLEN  | 1600 |         950
        30 | WARD   | 1250 |         950
        30 | MARTIN | 1250 |         950
        30 | BLAKE  | 2850 |         950
        30 | TURNER | 1500 |         950
        30 | JAMES  |  950 |         950
```

- 指定order by，当前窗口为第一行到当前行的范围。返回当前窗口的当前行的值：

```
select
        deptno,
        ename,
        sal,
        last_value(sal) over (partition by deptno order by sal desc) as
           ↪ last_value
    from emp order by deptno;
```

返回结果如下：

```
     deptno | ename  | sal  | last_value
    --------+--------+------+------------
         10 | KING   | 5000 |        5000
         10 | JACCKA | 5000 |        5000
         10 | CLARK  | 2450 |        2450
         10 | WELAN  | 2450 |        2450
         10 | MILLER | 1300 |        1300
         10 | TEBAGE | 1300 |        1300
         20 | SCOTT  | 3000 |        3000
         20 | FORD   | 3000 |        3000
         20 | JONES  | 2975 |        2975
         20 | ADAMS  | 1100 |        1100
         20 | SMITH  |  800 |         800
         30 | BLAKE  | 2850 |        2850
         30 | ALLEN  | 1600 |        1600
         30 | TURNER | 1500 |        1500
         30 | WARD   | 1250 |        1250
         30 | MARTIN | 1250 |        1250
         30 | JAMES  |  950 |         950
```

- nth_value(x, offset) → [与输入相同]

返回相对于窗口开头的指定偏移处的值。偏移从1开始。偏移可以是任何标量表达式。如果偏移为 NULL 或大于窗口中的值的数量，则返回 NULL。如果偏移为零或负数，则产生错误。

例如：将所有职工根据部门分组，返回每组中的第 6 行数据。

- 不指定order by，当前窗口为第一行到最后一行的范围，返回当前窗口第 6 行的值。

```
select
        deptno,
        ename,
        sal,
        nth_value(sal, 6) over (partition by deptno) as nth_value
    from emp order by deptno;
```

返回结果如下（当前窗口没有第 6 行的，返回 NULL）：

```
 deptno | ename  | sal  | nth_value
--------+--------+------+-----------
     10 | CLARK  | 2450 |       1300
     10 | KING   | 5000 |       1300
     10 | MILLER | 1300 |       1300
     10 | JACCKA | 5000 |       1300
     10 | WELAN  | 2450 |       1300
     10 | TEBAGE | 1300 |       1300
     20 | SMITH  |  800 |       NULL
     20 | JONES  | 2975 |       NULL
     20 | SCOTT  | 3000 |       NULL
     20 | ADAMS  | 1100 |       NULL
     20 | FORD   | 3000 |       NULL
     30 | ALLEN  | 1600 |        950
     30 | WARD   | 1250 |        950
     30 | MARTIN | 1250 |        950
     30 | BLAKE  | 2850 |        950
     30 | TURNER | 1500 |        950
     30 | JAMES  |  950 |        950
```

- 指定order by，当前窗口为第一行到当前行的范围，返回当前窗口第 6 行的值。

```
select
        deptno,
        ename,
        sal,
        nth_value(sal, 6) over (partition by deptno order by sal) as nth_value
    from emp order by deptno;
```

返回结果如下（当前窗口没有第 6 行的，返回 NULL）：

```
 deptno | ename  | sal  | nth_value
--------+--------+------+-----------
     10 | MILLER | 1300 |       NULL
     10 | TEBAGE | 1300 |       NULL
     10 | CLARK  | 2450 |       NULL
     10 | WELAN  | 2450 |       NULL
     10 | KING   | 5000 |       5000
     10 | JACCKA | 5000 |       5000
```

```
        20 | SMITH  |  800 |      NULL
        20 | ADAMS  | 1100 |      NULL
        20 | JONES  | 2975 |      NULL
        20 | SCOTT  | 3000 |      NULL
        20 | FORD   | 3000 |      NULL
        30 | JAMES  |  950 |      NULL
        30 | WARD   | 1250 |      NULL
        30 | MARTIN | 1250 |      NULL
        30 | TURNER | 1500 |      NULL
        30 | ALLEN  | 1600 |      NULL
        30 | BLAKE  | 2850 |      2850
```

- `lead(x[, offset[, default_value]])→ [与输入相同]`

返回窗口中当前行之后`offset`行处的值。偏移从0（当前行）开始。偏移可以是任何标量表达式。默认`offset`为`1`。如果偏移为 NULL 或大于窗口，则返回`default_value`，如果未指定该值，则返回`null`。该`lead()`函数要求指定窗口顺序。不得指定窗口框架。

例如：将所有职工根据部门（deptno）分组（作为开窗列），每位员工的薪水（sal）做偏移。

```
select
      deptno,
      ename,
      sal,
      lead(sal,
      1) over (partition by deptno order by sal) as sal_new
  from emp order by deptno;
```

返回结果如下：

```
 deptno | ename  | sal  | sal_new
--------+--------+------+---------
     10 | MILLER | 1300 |    1300
     10 | TEBAGE | 1300 |    2450
     10 | CLARK  | 2450 |    2450
     10 | WELAN  | 2450 |    5000
     10 | KING   | 5000 |    5000
     10 | JACCKA | 5000 |    NULL
     20 | SMITH  |  800 |    1100
     20 | ADAMS  | 1100 |    2975
     20 | JONES  | 2975 |    3000
     20 | SCOTT  | 3000 |    3000
     20 | FORD   | 3000 |    NULL
     30 | JAMES  |  950 |    1250
     30 | WARD   | 1250 |    1250
     30 | MARTIN | 1250 |    1500
     30 | TURNER | 1500 |    1600
     30 | ALLEN  | 1600 |    2850
     30 | BLAKE  | 2850 |    NULL
```

- lag(x[, offset[, default_value]])→ [与输入相同]

返回窗口中当前行之前offset行处的值。偏移从0（当前行）开始。偏移可以是任何标量表达式。默认offset为1。如果偏移为 NULL 或大于窗口，则返回default_value，如果未指定该值，则返回null。该lag()函数要求指定窗口顺序。不得指定窗口框架。

例如：将所有职工根据部门（deptno）分组（作为开窗列），每位员工的薪水（sal）做偏移。

```
select
      deptno,
      ename,
      sal,
      lag(sal, 1) over (partition by deptno order by sal) as sal_new
  from emp order by deptno;
```

返回结果如下：

```
   deptno | ename  | sal  | sal_new
  --------+--------+------+---------
       10 | MILLER | 1300 |    NULL
       10 | TEBAGE | 1300 |    1300
       10 | CLARK  | 2450 |    1300
       10 | WELAN  | 2450 |    2450
       10 | KING   | 5000 |    2450
       10 | JACCKA | 5000 |    5000
       20 | SMITH  |  800 |    NULL
       20 | ADAMS  | 1100 |     800
       20 | JONES  | 2975 |    1100
       20 | SCOTT  | 3000 |    2975
       20 | FORD   | 3000 |    3000
       30 | JAMES  |  950 |    NULL
       30 | WARD   | 1250 |     950
       30 | MARTIN | 1250 |    1250
       30 | TURNER | 1500 |    1250
       30 | ALLEN  | 1600 |    1500
       30 | BLAKE  | 2850 |    1600
```


1.6.16 数组函数和运算符

1.6.16.1 下标运算符：

[] 运算符用于访问数组中的一个元素，其索引从 1 开始：

```
  select my_array[1] AS first_element
```

1.6.16.2 连接运算符：||

|| 运算符用于将数组与一个相同类型的数组或元素进行连接：

```
SELECT ARRAY [1] || ARRAY [2]; -- [1, 2]
SELECT ARRAY [1] || 2; -- [1, 2]
SELECT 2 || ARRAY [1]; -- [2, 1]
```

1.6.16.3 数组函数

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

返回数组的所有元素是否都匹配给定的谓词。true如果所有元素都匹配谓词则返回（特殊情况是数组为空时）；false如果一个或多个元素不匹配；NULL如果谓词函数返回NULL一个或多个元素以及true所有其他元素。

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

返回数组的任何元素是否匹配给定的谓词。true如果一个或多个元素与谓词匹配则返回；false如果没有元素匹配（特殊情况是数组为空）；NULL如果谓词函数返回NULL一个或多个元素以及false 所有其他元素。

- array_distinct(x)→ array

删除数组 x 中的重复值。

```
select  array_distinct(array [1,2,5,6,8,4,5,1,2,6]); --  [1, 2, 5, 6, 8, 4]
```

- array_intersect(x, y)→ array

返回 x 与 y 的交集中的元素构成的数组，不含重复元素。

```
select array_intersect(array [1,2,3,4,8], array [6,5,2,3,1]); -- [1, 2, 3]
```

- array_union(x, y)→ array

返回 x 与 y 的并集中的元素构成的数组，不含重复元素。

```
select array_union(array [1,2,3,4,8], array [6,5,2,3,1]); -- [1, 2, 3, 4, 8, 6,
    ↪ 5]
```

- array_except(x, y)→ array

返回位于 x 但不位于 y 中的元素构成的数组，不含重复元素。

```
select array_except(array [1,2,3,4,8], array [6,5,2,3,1]); -- [4, 8]
```

- array_join(x, delimiter, null_replacement)→ varchar

使用分隔符和一个用于替换 NULL 的可选字符串连接给定数组的元素。

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

- array_max(x)→ x

返回输入数组中的最大值。

```
select array_max(array [1,2,3,4,8]); -- 8
```

- array_min(x)→ x

返回输入数组中的最小值。

```
select array_min(array [1,2,3,4,8]); -- 1
```

- array_position(x, element)→ bigint

返回数组 x 中 element 第一次出现的位置（如果没有找到，则返回 0）。

```
select array_position(array [1,2,3,4,8,3,2,1], 2); -- 2
```

- array_remove(x, element)→ array

删除数组 x 中所有等于 element 的元素。

```
select array_remove(array [1,2,3,4,8,3,2,1], 2); -- [1, 3, 4, 8, 3, 1]
```

- array_sort(x)→ array

对数组 x 进行排序并返回该数组。x 的元素必须是可排序的。NULL 元素将被放置在返回的数组的末尾。

```
select array_sort(array [1,2,3,4,8,3,null,2,1]); -- [1, 1, 2, 2, 3, 3, 4, 8,
    ↪ NULL]
```

- array_sort(array(T), function(T, T, int))→ array(T)

基于给定的比较函数 function 对 array 进行排序并将其返回。比较函数将接受两个可以为 NULL 的参数来表示 array 中两个可以为 NULL 的元素。当第一个可以为 NULL 的元素小于、等于或大于第二个可以为 NULL 的元素时，该函数返回 -1、0 或 1。如果比较函数返回其他值（包括 NULL），查询将失败并产生一个错误：

```
SELECT array_sort(ARRAY[3, 2, 5, 1, 2],
                  (x, y) -> IF(x < y, 1, IF(x = y, 0, -1)));
  -- [5, 3, 2, 2, 1]

SELECT array_sort(ARRAY['bc', 'ab', 'dc'],
                  (x, y) -> IF(x < y, 1, IF(x = y, 0, -1)));
  -- ['dc', 'bc', 'ab']


SELECT array_sort(ARRAY[3, 2, null, 5, null, 1, 2],
                  -- sort null first with descending order
                  (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);
  -- [null, null, 5, 3, 2, 2, 1]

SELECT array_sort(ARRAY[3, 2, null, 5, null, 1, 2],
                  -- sort null last with descending order
                  (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);
  -- [5, 3, 2, 2, 1, null, null]

SELECT array_sort(ARRAY['a', 'abcd', 'abc'],
                  -- sort by string length
                  (x, y) -> IF(length(x) < length(y), -1,
                               IF(length(x) = length(y), 0, 1)));
  -- ['a', 'abc', 'abcd']
```

```
SELECT array_sort(ARRAY[ARRAY[2, 3, 1], ARRAY[4, 2, 1, 4], ARRAY[1, 2]],
                  -- sort by array length
                  (x, y) -> IF(cardinality(x) < cardinality(y), -1,
                              IF(cardinality(x) = cardinality(y), 0, 1)));
  -- [[1, 2], [2, 3, 1], [4, 2, 1, 4]]
```

- arrays_overlap(x, y)→ boolean

测试数组 x 和 y 是否有任何共同的非 NULL 元素。如果没有共同的非 NULL 元素，但任一数组包含 NULL，则返回 NULL。

```
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)→ bigint

返回数组 x 的基数（大小）。

```
select cardinality(array [1,2,3,4,8,3,2,1]); -- 8
```

- concat(array1, array2, ..., arrayN)→ array 连接数组 array1、array2、... arrayN。该函数提供与 SQL 标准连接运算符 (||) 相同的功能。

```
select concat(array [1,2,3], array [0], array [7,8]); -- [1, 2, 3, 0, 7, 8]
```

- combinations(array(T), n)→ array(array(T))

返回输入数组的 n 元素子组。如果输入数组没有重复项，则 combinations 返回 n 元素子集：

```
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]]
```

子组的顺序是确定的，但未经指定。子组中元素的顺序是确定的，但未经指定。n 不得大于 5，生成的子组的总大小必须小于 100000。

- contains(x, element)→ boolean

如果数组 x 包含 element，则返回 true。

```
select contains(array [1,2,3,4,8,3,2,1], 2); -- true
```

- contains_sequence(x, seq)→ boolean

如果数组x包含所有数组seq作为子序列（所有值都以相同的连续顺序），则返回 true。

```
select contains_sequence(array [1,2,3], array [1,2]); -- true
```

- element_at(array(E), index)→ E

返回 array 在给定 index 处的元素。如果 index > 0，则该函数提供与 SQL 标准下标运算符 ([]) 相同的功能。如果 index < 0，则 element_at 按照从最后一个到第一个的顺序访问元素。

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

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

通过 function 针对其返回 true 的 array 的元素构造一个数组。

```
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)→ array

以串联的方式将 array(array(T)) 展开为 array(T)。

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

返回 array 的 n-gram（包含 n 个相邻元素的子序列）。结果中 n-gram 的顺序未经指定。

```
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))→ boolean

返回数组中是否没有元素与给定谓词匹配。true如果没有元素与谓词匹配则返回（特殊情况是数组为空时）；false如果一个或多个元素匹配；NULL如果谓词函数返回NULL一个或多个元素以及false所有其他元素。

```
SELECT none_match(ARRAY[1, 2, 3, 4], x -> x < 0); -- true
```

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

返回从 array 简化得到的单个值。会按顺序为 array 中的每个元素调用 inputFunction。除了接受元素之外，inputFunction 还接受当前状态（最初为 initialState）并返回新状态。会调用 outputFunction 以将最终状态转换为结果值。该函数可能是恒等函数 (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], BIGINT '0',
                (s, x) -> s + x,
                s -> s);
  -- 2147483648

  -- calculates arithmetic average
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)→ array

将 element 重复 count 次。

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

- reverse(x)→ array

返回一个数组，该数组中元素的顺序与数组 x 相反。

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

- sequence(start, stop)-> array(bigint)

生成一个从 start 到 stop 的整数序列，如果 start 小于等于 stop，则以 1 为单位递增，否则以 -1 为单位递增。

```
select sequence(5, 10); -- [5, 6, 7, 8, 9, 10]
```

- sequence(start, stop, step)-> array(bigint)

生成一个从 start 到 stop 的整数序列，以 step 为单位递增。

```
select sequence(1, 10, 2); -- [1, 3, 5, 7, 9]
```

- sequence(start, stop)-> array(date)

生成一个从 start 日期到 stop 日期的日期序列，如果 start 日期小于等于 stop 日期，则以 1 天为单位递增，否则以 -1 天为单位递增。

```
select sequence(date'2022-10-25', date'2022-10-30');
  -- [2022-10-25, 2022-10-26, 2022-10-27, 2022-10-28, 2022-10-29, 2022-10-30]
```

- sequence(start, stop, step)-> array(date)

生成一个从 start 到 stop 的序列，以 step 为单位递增。step 的类型可以是 INTERVAL DAY TO SECOND 或 INTERVAL YEAR TO MONTH。

```
select sequence(date'2022-10-25', date'2022-10-30', INTERVAL '2' DAY);
  -- [2022-10-25, 2022-10-27, 2022-10-29]
```

- sequence(start, stop, step)-> array(timestamp)

生成一个从 start 到 stop的时间戳序列，以 step 为单位递增。step 的类型可以是 INTERVAL DAY TO SECOND 或 INTERVAL YEAR TO MONTH。

```
select sequence(timestamp'2022-10-25 15:00:00', timestamp'2022-10-25 15:30:00',
    ↪ INTERVAL '10' MINUTE);
  -- [2022-10-25 15:00:00, 2022-10-25 15:10:00, 2022-10-25 15:20:00, 2022-10-25
     ↪ 15:30:00]
```

- shuffle(x)→ array

生成给定数组 x 的随机排列。

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

- slice(x, start, length)→ array

从索引 start 开始（如果 start 为负数，则从末尾开始）生成数组 x 的子集，其长度为 length。

```
select slice(array['a','b','c','d','e'], 2, 3); -- [b, c, d]
```

- trim_array(x, n)→ array

从数组末尾移除指定数量的元素。

```
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)

返回一个数组，该数组是对 array 的每个元素应用 function 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)

将给定的数组按元素合并到单个行数组中。第 N 个参数的第 M 个元素将是第 M 个输出元素的第 N 个字段。如果参数的长度不一致，则使用 NULL 填充缺少的值：

```
SELECT zip(ARRAY[1, 2], ARRAY['1b', null, '3b']); -- [ROW(1, '1b'), ROW(2, null)
    ↪ , ROW(null, '3b')]
```

- zip_with(array(T), array(U), function(T, U, R))-> array(R)

使用 function 将两个给定的数组按元素合并到单个数组中。如果一个数组较短，在应用 function 之前在其末尾添加 NULL 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));
  -- [ROW('a', 1), ROW('b', 3), ROW('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']
```

1.6.17 Map 函数和运算符

1.6.17.1 下标运算符：

[]运算符用于从 map 中检索给定键对应的值：

```
SELECT name_to_age_map['Bob'] AS bob_age;
```

1.6.17.2 map 函数

- cardinality(x)→ bigint

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)→ V

返回给定的key的值，如果键不包含在 map 中，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()→ map<unknown, unknown>

返回一个空 map：

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

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

返回使用给定的键/值数组创建的 map：

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

另请参见map_agg和multimap_agg，以了解如何创建作为聚合的 map。

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

返回从给定的项数组创建的 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))

返回从给定的项数组创建的多重映射。每个键可以关联多个值：

```
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))

返回一个包含给定的 map 中所有项的数组：

```
SELECT map_entries(MAP(ARRAY[1, 2], ARRAY['x', 'y'])); -- [ROW(1, 'x'), ROW(2, '
    ↪ y')]
```

- map_concat(map1(K, V), map2(K, V), ..., mapN(K, V))-> map(K, V)

返回所有给定的 map 的并集。如果在多个给定的 map 中找到某个键，则在生成的 map 中该键的值来自这些 map 中的最后一个 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);
```

```
  -- {k1 -> 20, k3 -> 15}
```

- map_filter(map(K, V), function(K, V, boolean))-> map(K, V)

通过function针对其返回 true 的map的项构造一个 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);
  -- {k1 -> 20, k3 -> 15}
```

- map_keys(x(K, V))-> array(K)

返回 map x中的所有键。

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

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

返回 map x中的所有值。

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

- map_zip_with(map(K, V1), map(K, V2), function(K, V1, V2, V3))-> map(K, V3)

通过向具有相同键的一对值应用function，将两个给定的 map 合并为单个 map。对于仅出现在一个 map 中的键，会传入 NULL 以用作缺失的键的值：

```
SELECT map_zip_with(MAP(ARRAY[1, 2, 3], ARRAY['a', 'b', 'c']),
                    MAP(ARRAY[1, 2, 3], ARRAY['d', 'e', 'f']),
                    (k, v1, v2) -> concat(v1, v2));
  -- {1 -> ad, 2 -> be, 3 -> cf}

SELECT map_zip_with(MAP(ARRAY['k1', 'k2'], ARRAY[1, 2]),
                    MAP(ARRAY['k2', 'k3'], ARRAY[4, 9]),
                    (k, v1, v2) -> (v1, v2));
  -- {k1 -> ROW(1, null), k2 -> ROW(2, 4), k3 -> ROW(null, 9)}

SELECT map_zip_with(MAP(ARRAY['a', 'b', 'c'], ARRAY[1, 8, 27]),
                    MAP(ARRAY['a', 'b', 'c'], ARRAY[1, 2, 3]),
                    (k, v1, v2) -> k || CAST(v1 / v2 AS VARCHAR));
  -- {a -> a1, b -> b4, c -> c9}
```

- transform_keys(map(K1, V), function(K1, V, K2))-> map(K2, V)

返回一个向map的每个项应用function并转换键的 map：

```
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, 4 -> 2, 9 -> 3}

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]);
  -- {one -> 1.0, two -> 1.4}
```

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

返回一个向map的每个项应用function并转换值的 map:

```
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}
```

1.6.18 URL 函数

1.6.18.1 提取函数

URL 提取函数从 HTTP URL（或任何符合 RFC 2396 标准的有效 URI）中提取组成部分。支持以下语法:

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

提取的组成部分不包含:或?等 URI 语法分隔符。

- url_extract_fragment(url)→ varchar

  从url返回片断标识符。

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

- url_extract_host(url)→ varchar

从url返回主机。

```
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

返回 URL 中的参数, 即?query,query 中参数name对应的值。

从url返回第一个名为name的查询字符串参数的值。按照 1866#section-8.2.1 中指定的典型方式来处理参数提取。

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

- url_extract_path(url)→ varchar

从url返回路径。

```
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

从url返回端口号。

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

- url_extract_protocol(url)→ varchar

从url返回协议。

```
SELECT url_extract_protocol('http://localhost:8080/req_path'); -- http

SELECT url_extract_protocol('https://127.0.0.1:8080/req_path'); -- https
```

```
SELECT url_extract_protocol('ftp://path/file'); -- ftp
```

- url_extract_query(url)→ varchar

从url返回查询字符串。

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

1.6.18.2 编码函数

- url_encode(value)→ varchar

通过对value进行编码来对其进行转义，以便可以安全地将其包含在 URL 查询参数名称和值中：

- 不对字母数字字符进行编码。
- 不对字符.、-、*和_进行编码。
- 将 ASCII 空格字符编码为+。
- 将所有其他字符都转换为 UTF-8，将字节编码为字符串%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%3
     ↪ D25%23teacher
```

- url_decode(value)→ varchar

对 URL 编码value进行反转义。该函数是url_encode的反函数。

```
select url_decode('http%3A%2F%2Fwww.example.com%3A80%2Fstu%2Findex.html%3Fname%3
    ↪ Dxxx%26age%3D25%23teacher');
  -- http://www.example.com:80/stu/index.html?name=xxx&age=25#teacher
```

1.6.19 UUID 函数

- uuid()→ uuid

返回伪随机生成的 UUID 类型（类型 4）。

```
select uuid(); -- 88db06ad-31d6-4bb1-9794-85ea69533d63
```

1.6.20 颜色函数

- bar(x, width)→ varchar

使用默认的low_color 红色和 high_color 绿色呈现 ANSI 条形图中的单个条形。例如，将 x 值 25% 和 width 值 40 传递给该函数，则呈现一个 10 字符红色条形图，后跟 30 个空格，从而创建一个 40 字符条形图。

- bar(x, width, low_color, high_color)→ varchar

呈现 ANSI 条形图中具有指定 width 的单个行。参数 x 是处于 [0,1] 范围之内的 double 值。处于范围 [0,1] 之外的 x 值将被截断为值 0 或 1。low_color 和 high_color 捕获颜色以用于水平条形图的两端。例如，如果 x 为 0.5，width 为 80，low_color 为 0xFF0000，high_color 为 0x00FF00，则该函数返回一个 40 字符条形，其颜色从红色 (0xFF0000) 变为黄色 (0xFFFF00)，使用空格对 80 字符条形的其余部分进行填充。

```
hubble> select bar(0.75, 80, rgb(255, 0, 0), rgb(0, 255, 0)) as bar;
                                    bar
-------------------------------------------------------------------------------

(1 row)

Query 20230330_020233_00007_as8md, FINISHED, 1 node
Splits: 1 total, 1 done (100.00%)
0.13 [0 rows, 0B] [0 rows/s, 0B/s]
```

- `color(string)→ color`

从格式为“#000”的 4 字符字符串捕获解码的 RGB 值，返回相应的颜色。输入字符串应该为 varchar，其中包含 CSS 样式的短 RGB 字符串，或者为 `black`、`red`、`green`、`yellow`、`blue`、`magenta`、`cyan` 和 `white` 之一。

- `color(x, low, high, low_color, high_color)→ color`

返回一个介于 `low_color` 和 `high_color` 之间的颜色，使用 double 参数 `x`、`low` 和 `high` 计算得出一个小数，然后将该小数传给下面显示的 `color(fraction, low_color, high_color)` 函数。如果 `x` 处于 `low` 和 `high` 定义的范围之外，则对其值进行截断，以使其处于该范围之内。

- `color(x, low_color, high_color)→ color`

根据介于 0 和 1.0 之间的 double 参数 `x` 返回一个介于 `low_color` 和 `high_color` 之间的颜色。参数 `x` 是处于 [0,1] 范围之内的 double 值。处于范围 [0,1] 之外的 `x` 值将被截断为值 0 或 1。

- `render(x, color)→ varchar`

使用特定的颜色（使用 ANSI 颜色代码）呈现值 `x`。`x` 可以为 double、bigint 或 varchar 类型。

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

- `render(b)→ varchar`

接受 boolean 值 `b` 并使用 ANSI 颜色代码将绿色呈现为 true 或将红色呈现为 false。

- `rgb(red, green, blue)→ color`

返回一个颜色值，捕获三个作为 int 参数（范围为 0 至 255）提供的分量颜色值的 RGB 值：`red`、`green`、`blue`。

1.6.21 会话信息

- `current_user → varchar`

返回当前运行查询的用户。

```
select current_user; -- hubble
```

- `current_groups()→ varchar`

返回运行查询的当前用户的组列表。

```
select current_groups(); -- []
```

- `current_catalog → varchar`

返回表示当前目录名称的字符串。

```
select current_catalog; -- hubble
```

- current_schema → varchar

返回表示当前非限定模式名称的字符串。

```
select current_schema; -- examp
```

> **注意**
>
> 这是 SQL 标准的一部分，不使用圆括号。

地址：北京市朝阳区金田公园内 8 号-22 栋

网址：http://www.beagledata.com

邮箱：websales@beagledata.com

电话：18610182713