大数据计算服务

SQL

为了无法计算的价值 | 阿里云

SQL

概要介绍

MaxCompute SQL适用于海量数据(TB级别)，实时性要求不高的场合，它的每个作业的准备，提交等阶段要花费较长时间，因此要求每秒处理几千至数万笔事务的业务是不能用 MaxCompute 完成的。

MaxCompute SQL采用的是类似于SQL的语法，可以看作是标准SQL的子集，但不能因此简单的把 MaxCompute 等价成一个数据库，它在很多方面并不具备数据库的特征，如事务、主键约束、索引等。目前在 MaxCompute 中允许的最大SQL长度是2MB。

关键字

MaxCompute 将SQL语句的关键字作为保留字。在对表、列或是分区命名时请不要使用，否则会报错。保留字不区分大小写。下面只给出常用的保留字列表，完整的保留字列表请参阅 MaxCompute SQL保留字。

```
  %   &   &&   (   )   *   +
- . / ; < <= <>
= > >= ? ADD ALL ALTER
AND AS ASC BETWEEN BIGINT BOOLEAN BY
CASE CAST COLUMN COMMENT CREATE DESC DISTINCT
DISTRIBUTE DOUBLE DROP ELSE FALSE FROM FULL
GROUP IF IN INSERT INTO IS JOIN
LEFT LIFECYCLE LIKE LIMIT MAPJOIN NOT NULL
ON OR ORDER OUTER OVERWRITE PARTITION RENAME
REPLACE RIGHT RLIKE SELECT SORT STRING TABLE
THEN TOUCH TRUE UNION VIEW WHEN WHERE
```

MaxCompute SQL允许数据类型之间的转换，类型转换方式包括：显式类型转换及隐式类型转换。

类型转换说明

显式类型转换

显式类型转换是用cast将一种数据类型的值转换为另一种类型的值的行为，在MaxCompute SQL中支持的显式类型转换如下：

From/To	Bigint	Double	String	Datetime	Boolean	Decimal

Bigint	–	Y	Y	N	N	Y
Double	Y	–	Y	N	N	Y
String	Y	Y	–	Y	N	Y
Datetime	N	N	Y	–	N	N
Boolean	N	N	N	N	–	N
Decimal	Y	Y	Y	N	N	-

其中，’ Y’ 表示可以转换，’ N’ 表示不可以转换，’ –’ 表示不需要转换。

比如：

```
select cast(user_id as double) as new_id from user;

select cast('2015-10-01 00:00:00' as datetime) as new_date from user;
```

备注:

- 将double类型转为bigint类型时，小数部分会被截断，例如：cast(1.6 as bigint) = 1；
- 满足double格式的string类型转换为bigint时，会先将string转换为double，再将double转换为bigint，因此，小数部分会被截断，例如cast(“1.6” as bigint) = 1；
- 满足bigint格式的string类型可以被转换为double类型，小数点后保留一位，例如：cast(“1” as double) = 1.0；
- 不支持的显式类型转换会导致异常；
- 如果在执行时转换失败，报错退出；
- 日期类型转换时采用默认格式yyyy-mm-dd hh:mi:ss，详细说明信息请参考String类型与Datetime类型之间的转换；
- 部分类型之间不可以通过显式的类型转换，但可以通过SQL内建函数进行转换，例如：从boolean类型转换到string类型，可使用函数to_char，详细介绍请参考 TO_CHAR ，而to_date函数同样支持从string类型到datetime类型的转换，详细介绍请参考 TO_DATE ；
- 关于cast的介绍请参阅 CAST ；
- DECIMAL超出值域，CAST STRING TO DECIMAL可能会出现最高位溢出报错，最低位溢出截断等情况。

隐式类型转换及其作用域

隐式类型转换是指在运行时，由 MaxCompute 依据上下文使用环境及类型转换规则自动进行的类型转换。MaxCompute 支持的隐式类型转换规则与显式转换相同：

From/To	Bigint	Double	String	Datetime	Boolean	Decimal
Bigint	–	Y	Y	N	N	Y
Double	Y	–	Y	N	N	Y

String	Y	Y	–	Y	N	Y
Datetime	N	N	Y	–	N	N
Boolean	N	N	N	N	–	N
Decimal	Y	Y	Y	N	N	-

其中，’ Y’ 表示可以转换，’ N’ 表示不可以转换，’ –’ 表示不需要转换。

常见用法如下：

```
select user_id+age+'12345',
concat(user_name,user_id,age)
from user;
```

备注:

- 不支持的隐式类型转换会导致异常；
- 如果在执行时转换失败，也会导致异常；
- 由于隐式类型转换是 MaxCompute 依据上下文使用环境自动进行的类型转换，因此，我们推荐在类型不匹配时显式的用cast进行转换;
- 隐式类型转换规则是有发生作用域的。在某些作用域中，只有一部分规则可以生效。详细信息请参考隐式类型转换的作用域；

关系运算符

关系运算符包括：=, <>, <, <=, >, >=, IS NULL, IS NOT NULL, LIKE, RLIKE和IN。由于LIKE, RLIKE和IN的隐式类型转换规则不同于其他关系运算符，将单独拿出章节对这三种关系运算符做出说明。本小节的说明不包含这三种特殊的关系运算符。当不同类型的数据共同参与关系运算时，按照下述原则进行隐式类型转换。

From/To	Bigint	Double	String	Datetime	Boolean	Decimal
Bigint	–	Double	Double	N	N	Decimal
Double	Double	–	Double	N	N	Decimal
String	Double	Double	–	Datetime	N	Decimal
Datetime	N	N	Datetime	–	N	N
Boolean	N	N	N	N	–	N
Decimal	Decimal	Decimal	Decimal	N	N	-

备注:

- 如果待比较的两个类型间不能进行隐式类型转换，则该关系运算不能完成，报错退出；
- 关系运算符介绍，请参阅 关系操作符 ；

特殊的关系运算符(LIKE, RLIKE, IN)

LIKE及RLIKE的使用方式形如：

```
    source like pattern;
source rlike pattern;
```

此二者在隐式类型转换中的注意事项：

- LIKE和RLIKE的source和pattern参数均仅接受string类型；
- 其他类型不允许参与运算，也不能进行到string类型的隐式类型转换；IN的使用方式形如：

```
key in (value1, value2, ...)
```

In的隐式转换规则：

- In右侧的value值列表中的数据类型必须一致；
- 当key与values之间比较时，若bigint, double, string之间比较，统一转double，若datetime和string之间比较，统一转datetime。除此之外不允许其它类型之间的转换。

算术运算符

算术运算符包括：+, -, * , /, %, +, - ，其隐式转换规则：

只有string、bigint、double和Decimal才能参与算术运算。String在参与运算前会进行隐式类型转换到double。Bigint和double共同参与计算时，会将bigint隐式转换为double。日期型和布尔型不允许参与算数运算。

备注：

- 算术运算符的相关章节 **算术操作符**。

逻辑运算符

逻辑运算符包括：and, or和not，其隐式转换规则:

- 只有boolean才能参与逻辑运算。
- 其他类型不允许参与逻辑运算，也不允许其他类型的隐式类型转换。

备注：

- 逻辑运算符的相关章节 **逻辑操作符**。

MaxCompute SQL内建函数

MaxCompute SQL提供了大量的系统函数，方便用户对任意行的一列或多列进行计算，输出任意种的数据类型。其隐式转换规则:

- 在调用函数时，如果输入参数的数据类型与函数定义的参数数据类型不一致，把输入参数的数据类型转换为函数定义的数据类型。
- 每个ODPS SQL内建函数的参数对于允许的隐式类型转换的要求不同，详见 内建函数 部分的说明。

CASE WHEN

Case when的隐式转换规则:

- 如果返回类型只有bigint,double，统一转double；
- 如果返回类型中有string类型，统一转string，如果不能转则报错(如boolean类型)；
- 除此之外不允许其它类型之间的转换；

备注：

- Case when的详细介绍请参阅 [CASE WHEN表达式]

分区表

MaxCompute SQL支持分区表。指定分区表会对用户带来诸多便利，例如：提高SQL运行效率，减少计费等。在如下场景下使用分区表将会带来较大的收益：

- 在Select语句的Where条件过滤中使用分区列作为过滤条件；

```
create table src (key string, value bigint) partitioned by (pt string);  -- 目前，MaxCompute 仅承诺String类型分区
select * from src where pt='20151201'; -- 正确使用方式。MaxCompute 在生成查询计划时只会将'20151201'分区的数据纳入输入中
select * from src where pt = 20151201; -- 错误的使用方式。在这样的使用方式下，MaxCompute并不能保障分区过滤机制的有效性。pt是String类型，当String类型与Bigint(20151201)比较时，MaxCompute会将二者转换为Double类型，此时有可能会有精度损失。
```

与此同时，部分对分区操作的SQL的运行效率则较低，给您带来较高的计费，例如：

- 使用动态分区

对于部分 MaxCompute 操作命令，处理分区表和非分区表时的语法有差别，详细情况请参考DDL语句 及 DML语句 部分的说明 。目前，ODPS分区仅支持string类型，不承诺其他分区类型的正确性，不支持其他任意类型的隐式类型转换。

UNION ALL

参与 UNION ALL 运算的所有列的数据类型、列个数、列名称必须完全一致，否则抛异常。

String类型与Datetime类型之间的转换

MaxCompute 支持string类型和datetime类型之间的相互转换。转换时使用的格式为yyyy-mm-dd hh:mi:ss，其中：

单位	字符串(忽略大小写)	有效值域
年	yyyy	0001 ~ 9999
月	mm	01 ~ 12
日	dd	01 ~ 28,29,30,31
时	hh	00 ~ 23
分	mi	00 ~ 59
秒	ss	00 ~ 59

备注：

- 各个单位的值域中，如果首位为0，不可省略，例如：”2014-1-9 12:12:12”就是非法的datetime格式，无法从这个string类型数据转换为datetime类型，必须写为”2014-01-09 12:12:12”。
- 只有符合上述格式描述的string类型才能够转换为datetime类型，例如：cast(“2013-12-31 02:34:34” as datetime)，将会把string类型”2013-12-31 02:34:34”转换为datetime类型。同理，datetime转换为string时，默认转换为yyyy-mm-dd hh:mi:ss的格式。

类似于下面的转换尝试，将会失败导致异常，例如：

```
cast("2013/12/31 02/34/34" as datetime)
cast("20131231023434" as datetime)
cast("2013-12-31 2:34:34" as datetime)
```

值得注意的是，” dd”部分的阈值上限取决于月份实际拥有的天数，如果超出对应月份实际拥有的天数，将会导致异常退出，例如：

```
cast("2013-02-29 12:12:12" as datetime)     -- 异常返回，2013年2月没有29日
cast("2013-11-31 12:12:12" as datetime) -- 异常返回，2013年11月没有31日
```

MaxCompute 提供了to_date函数，用以将不满足日期格式的string类型数据转换为datetime类型。详细信息请参阅 TO_DATE 。

关系操作符

操作符	说明
A=B	如果A或B为NULL，返回NULL；如果A等于B，返回TRUE，否则返回FALSE
A<>B	如果A或B为NULL，返回NULL；如果A不等于B，返回TRUE，否则返回FALSE
A<B	如果A或B为NULL，返回NULL；如果A小于B，返回TRUE，否则返回FALSE
A<=B	如果A或B为NULL，返回NULL；如果A小于等于B，返回TRUE，否则返回FALSE
A>B	如果A或B为NULL，返回NULL；如果A大于B，返回TRUE，否则返回FALSE
A>=B	如果A或B为NULL，返回NULL；如果A大于等于B，返回TRUE，否则返回FALSE
A IS NULL	如果A为NULL，返回TRUE，否则返回FALSE
A IS NOT NULL	如果A不为NULL，返回TRUE，否则返回FALSE
A LIKE B	如果A或B为NULL，返回NULL，A为字符串，B为要匹配的模式， 如果匹配，返回TRUE，否则返回FALSE。'%'匹配任意多个字符，'_'匹配单个字符。要匹配'%'或’ _’ 需要用转义符表示’ \%’ ，’ _’ 。 - ‘aaa’ like ‘a__’ = TRUE ‘aaa’ like ‘a%’ = TRUE ‘aaa’ like ‘aab’ = FALSE ‘a%b’ like ‘a\%b’ = TRUE ‘axb’ like ‘a\%b’ = FALSE
A RLIKE B	A是字符串，B是字符串常量正则表达式； 如果匹配成功，返回TRUE，否则返回FALSE； 如果B为空串会报错退出；如果A或B为NULL，返回NULL；
A IN B	B是一个集合，如果A为NULL，返回NULL，如A在B中则返回TRUE，否则返回FALSE 若B仅有一个元素NULL，即A IN (NULL)，则返回NULL。 若B含有NULL元素，将NULL视为B集合中其他元素的类型。 B必须是常数并且至少有一项，所有类型要一致

常见用法如下：

```
select * from user where user_id = '0001';
select * from user where user_name <> 'maggie';
select * from user where age > ‘50’;
select * from user where birth_day >= '1980-01-01 00:00:00';
select * from user where is_female is null;
select * from user where is_female is not null;
select * from user where user_id in (0001,0010);
select * from user where user_name like 'M%';
```

由于double值存在一定的精度差，因此，我们不建议直接使用等号=对两个double类型数据进行比较。用户可以使用两个double类型相减，而后取绝对值的方式判断。当绝对值足够小时，认为两个double数值相等，例如:

```
abs(0.9999999999 - 1.0000000000) < 0.000000001
-- 0.9999999999和1.0000000000为10位精度，而0.000000001为9位精度。
-- 此时可以认为0.9999999999和1.0000000000相等。
```

备注：

- Abs是ODPS提供的内建函数，意为取绝对值，详细可参考 ABS 。
- 通常情况下，ODPS的double类型能够保障14位有效数字。

算术操作符

操作符	说明
A + B	如果A或B为NULL，返回NULL；否则返回A + B的结果。
A – B	如果A或B为NULL，返回NULL；否则返回A – B的结果。
A * B	如果A或B为NULL，返回NULL；否则返回A * B的结果。
A / B	如果A或B为NULL，返回NULL；否则返回A / B的结果。注：如果A和B为bigint类型，返回结果为double类型。
A % B	如果A或B为NULL，返回NULL；否则返回A模B的结果。
+A	仍然返回A。
-A	如果A为NULL，返回NULL，否则返回-A。

常见用法如下：

```
select age+10, age-10, age%10, -age, age*age, age/10
```

```
from user;
```

备注

- 只有string，bigint，double才能参与算术运算，日期型和布尔型不允许参与运算。
- String类型在参与运算前会进行隐式类型转换到double类型。
- bigint和double共同参与计算时，会将bigint隐式转换为double再进行计算，返回结果为double类型。
- A和B都是bigint类型，进行A/B运算,返回结果为double类型；进行上述其他运算仍然返回bigint类型。

位操作符

操作符	说明
A & B	返回A与B进行按位与的结果。例如：1&2返回0，1&3返回1，NULL与任何值按位与都为NULL。 A和B必须为Bigint类型。
A | B	返回A与B进行按位或的结果。例如：1 |2返回3，1 |3返回3，NULL与任何值按位或都为NULL。A和B 必须为Bigint类型。

备注：

- 位运算符不支持隐式转换，只允许bigint类型。

逻辑操作符

```
操作符          说明
A and B       TRUE and TRUE=TRUE
              TRUE and FALSE=FALSE
              FALSE and TRUE=FALSE
              FALSE and NULL=FALSE
              NULL and FALSE=FALSE
              TRUE and NULL=NULL
              NULL and TRUE=NULL
              NULL and NULL=NULL

A or B        TRUE or TRUE=TRUE
              TRUE or FALSE=TRUE
              FALSE or TRUE=TRUE
              FALSE or NULL=NULL
              NULL or FALSE=NULL
              TRUE or NULL=TRUE
```

```
            NULL or TRUE=TRUE
            NULL or NULL=NULL

NOT A       如果A是NULL，返回NULL
            如果A是TRUE，返回FALSE
            如果A是FALSE，返回TRUE
```

备注

- 逻辑操作符只允许boolean类型参与运算，不支持隐式类型转换。

表操作

创建表

语法格式

```
   CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[LIFECYCLE days]
[AS select_statement]
[STORED BY StorageHandler] -- 仅限外部表
[WITH SERDEPROPERTIES (Options)] -- 仅限外部表
[LOCATION OSSLocation];-- 仅限外部表


CREATE TABLE [IF NOT EXISTS] table_name
LIKE existing_table_name
```

说明：

- 表名与列名均对大小写不敏感。
- 在创建表时，如果不指定if not exists选项而存在同名表，则返回出错；若指定此选项，则无论是否存在同名表，即使原表结构与要创建的目标表结构不一致，均返回成功。已存在的同名表的元信息不会被改动。
- 数据类型只能是：bigint，double，boolean，datetime及string。
- 表名，列名中不能有特殊字符，只能用英文的a-z，A-Z及数字和下划线_，且以字母开头，名称的长度不超过128字节。
- Partitioned by 指定表的分区字段，目前仅支持string类型。分区值不允许有双字节字符(如中文)，必须是以英文字母a-z，A-Z开始后可跟字母数字，名称的长度不超过128字节。 允许的字符包括：空格 ‘ ’，冒号’:’ ，下划线’_’ ，美元符’$’ ，井号’#’ ，点’.’ ，感叹号’!’和’@’ ，出现其他字符行为未定义，例如：”\t” ，”\n” ，” /”等。当利用分区字段对表进行分区时，新增分区

、更新分区内数据和读取分区数据均不需要做全表扫描，可以提高处理效率。
- 注释内容是长度不超过1024字节的有效字符串。
- lifecycle指明此表的生命周期，create table like语句不会复制源表的生命周期属性。
- 目前，在表中建的分区层次不能超过6级。
- 更多有关外部表的说明请参考云栖社区文章之MaxCompute上如何处理非结构化数据。

在下面的例子中，创建表sale_detail保存销售记录，该表使用销售时间(sale_date)和销售区域(region)作为分区列：

```
    create table if not exists sale_detail(
shop_name string,
customer_id string,
total_price double)
partitioned by (sale_date string,region string);
-- 创建一张分区表sale_detail
```

也可以通过create table ... as select ..语句创建表，并在建表的同时将数据复制到新表中，如

```
    create table sale_detail_ctas1 as
select * from sale_detail;
```

此时，如果sale_detail中存在数据，上面的示例会将sale_detail的数据全部复制到sale_detail_ctas1表中。但请注意，此处sale_detail是一张分区表，而通过create table ... as select ... 语句创建的表不会复制分区属性，只会把源表的分区列作为目标表的一般列处理，即sale_detail_ctas1是一个含有5列的非分区表。

在create table ... as select ...语句中，如果在select子句中使用常量作为列的值，建议指定列的名字，例如：

```
    create table sale_detail_ctas2 as
select shop_name,
customer_id,
total_price,
'2013' as sale_date,
'China' as region
from sale_detail;
```

如果不加列的别名，如：

```
    create table sale_detail_ctas3 as
select shop_name,
customer_id,
total_price,
'2013',
'China'
from sale_detail;
```

则创建的表sale_detail_ctas3的第四、五列会是类似_c5，_c6。 如果希望源表和目标表具有相同的表结构，可以尝试使用create table ... like操作，如：

```
create table sale_detail_like like sale_detail;
```

此时，sale_detail_like的表结构与sale_detail完全相同。除生命周期属性外，列名、列注释以及表注释等均相同。但sale_detail中的数据不会被复制到sale_detail_like表中。

删除表

语法格式

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

说明：

- 如果不指定if exists选项而表不存在，则返回异常；若指定此选项，无论表是否存在，皆返回成功。
- 删除外部表时，OSS上的数据不会被删除；

示例：

```
create table sale_detail_drop like sale_detail;
drop table sale_detail_drop;
--若表存在，成功返回；若不存在，异常返回；
drop table if exists sale_detail_drop2;
--无论是否存在sale_detail_drop2表，均成功返回；
```

重命名表

语法格式

```
ALTER TABLE table_name RENAME TO new_table_name;
```

说明：

- rename操作仅修改表的名字，不改动表中的数据。
- 如果已存在与new_table_name同名表，报错。
- 如果table_name不存在，报错：

示例：

```
create table sale_detail_rename1 like sale_detail;
alter table sale_detail_rename1 rename to sale_detail_rename2;
```

修改表的注释

命令格式：

```
ALTER TABLE table_name SET COMMENT 'tbl comment';
```

说明：

- table_name必须是已存在的表；comment最长1024字节；

示例：

```
alter table sale_detail set comment 'new coments for table sale_detail';
```

通过 MaxCompute 的desc命令可以查看表中comment的修改，请参阅 查看表信息 。

修改表的生命周期属性

MaxCompute 提供数据生命周期管理功能，方便用户释放存储空间，简化回收数据的流程。

语法格式：

```
ALTER TABLE table_name SET lifecycle days;
```

说明：

- days 参数为生命周期时间，只接受正整数。单位：天；

如果表table_name是非分区表，自最后一次数据被修改开始计算，经过days天后数据仍未被改动，则此表无需用户干预，

将会被 MaxCompute 自动回收(类似drop table操作)。在 MaxCompute 中，每当表的数据被修改后，表的LastDataModifiedTime将会被更新，因此，

MaxCompute 会根据每张表的LastDataModifiedTime以及lifecycle的设置来判断是否要回收此表。如果table_name是分区表，

则根据各分区的LastDataModifiedTime判断该分区是否该被回收。关于LastDataModifiedTime的介绍请参考 查看表信息 。

不同于非分区表，分区表的最后一个分区被回收后，该表不会被删除。生命周期只能设定到表级别，不能再分区级设置生命周期。创建表时即可指定生命周期。

示例：

```
create table test_lifecycle(key string) lifecycle 100;
-- 新建test_lifecycle表，生命周期为100天。

alter table test_lifecycle set lifecycle 50;
-- 修改test_lifecycle表，将生命周期设为50天。
```

修改表的修改时间

MaxCompute SQL提供touch操作用来修改表的LastDataModifiedTime。效果会将表的LastDataModifiedTime修改为当前时间。

语法格式：

```
ALTER TABLE table_name TOUCH;
```

说明：

- table_name不存在，则报错返回；；
- 此操作会改变表的”LastDataModifiedTime”的值，此时，MaxCompute 会认为表的数据有变动，生命周期的计算会重新开始。

清空非分区表里的数据

将指定的非分区表中的数据清空， 该命令不支持分区表。对于分区表，可以用ALTER TABLE table_name DROP PARTITION的方式将分区里的数据清除。

语法格式:

```
TRUNCATE TABLE table_name;
```

视图操作

创建视图

语法格式

```
CREATE [OR REPLACE] VIEW [IF NOT EXISTS] view_name
[(col_name [COMMENT col_comment], ...)]
[COMMENT view_comment]
[AS select_statement]
```

说明：

- 创建视图时，必须有对视图所引用表的读权限。
- 视图只能包含一个有效的select语句。
- 视图可以引用其它视图，但不能引用自己，也不能循环引用。
- 不允许向视图写入数据，例如使用insert into 或者insert overwrite操作视图。
- 当视图建好以后，如果视图的引用表发生了变更，有可能导致视图无法访问，例如：删除被引用表。

用户需要自己维护引用表及视图之间的对应关系。
- 如果没有指定if not exists，在视图已经存在时用create view会导致异常。这种情况可以用create or replace view来重建视图，重建后视图本身的权限保持不变。

示例：

```
create view if not exists sale_detail_view
(store_name, customer_id, price, sale_date, region)
comment 'a view for table sale_detail'
as select * from sale_detail;
```

删除视图

语法格式

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

说明：

- 如果视图不存在且没有指定if exists，报错。

示例：

```
DROP VIEW IF EXISTS sale_detail_view;
```

重命名视图

语法格式

```
ALTER VIEW view_name RENAME TO new_view_name;
```

说明：

- 如果已存在同名视图，报错。

示例：

```
create view if not exists sale_detail_view
(store_name, customer_id, price, sale_date, region)
comment 'a view for table sale_detail'
as select * from sale_detail;

alter view sale_detail_view rename to market;
```

分区/列操作

添加分区

语法格式:

```
ALTER TABLE TABLE_NAME ADD [IF NOT EXISTS] PARTITION partition_spec

partition_spec:
: (partition_col1 = partition_col_value1, partition_col2 = partiton_col_value2, ...)
```

说明：

- 如果未指定if not exists而同名的分区已存在，则出错返回。
- 目前 MaxCompute 支持的分区数量上限为6万。
- 对于多级分区的表，如果想添加新的分区，必须指明全部的分区值。

示例：为sale_detail表添加一个新的分区，

```
alter table sale_detail add if not exists partition (sale_date='201312', region='hangzhou');
-- 成功添加分区，用来存储2013年12月杭州地区的销售记录。

alter table sale_detail add if not exists partition (sale_date='201312', region='shanghai');
-- 成功添加分区，用来存储2013年12月上海地区的销售记录。

alter table sale_detail add if not exists partition(sale_date='20111011');
-- 仅指定一个分区sale_date，出错返回

alter table sale_detail add if not exists partition(region='shanghai');
-- 仅指定一个分区region，出错返回
```

删除分区

```
ALTER TABLE TABLE_NAME DROP [IF EXISTS] PARTITION partition_spec;

partition_spec:
: (partition_col1 = partition_col_value1, partition_col2 = partiton_col_value2, ...)
```

说明：

- 如果分区不存在且未指定if exists，则出错返回。

示例：从表sale_detail中删除一个分区，

```
alter table sale_detail drop if exists partition(sale_date='201312',region='hangzhou');
```

```
-- 成功删除2013年12月杭州分区的销售。
```

添加列

命令格式：

```
ALTER TABLE table_name ADD COLUMNS (col_name1 type1, col_name2 type2...)
```

说明：

- 列的数据类型只能是：bigint，double，boolean，datetime及string类型。

修改列名

命令格式：

```
ALTER TABLE table_name CHANGE COLUMN old_col_name RENAME TO new_col_name;
```

说明：

- old_col_name必须是已存在的列；
- 表中不能有名为new_col_name的列；

修改列、分区注释

命令格式：

```
ALTER TABLE table_name CHANGE COLUMN col_name COMMENT comment_string;
```

说明：

- COMMENT内容最长1024字节；

同时修改列名及列注释

命令格式：

```
ALTER TABLE table_name CHANGE COLUMN old_col_name new_col_name column_type COMMENT
column_comment;
```

说明：

- old_col_name必须是已存在的列；

- 表中不能有名为new_col_name的列；
- COMMENT内容最长1024字节；

修改表、分区的修改时间

MaxCompute SQL提供touch操作用来修改分区的LastDataModifiedTime。效果会将分区的LastDataModifiedTime修改为当前时间。

语法格式：

```
ALTER TABLE table_name TOUCH PARTITION(partition_col='partition_col_value', ...);
```

说明：

- table_name或partition_col不存在，则报错返回；
- 指定的partition_col_value不存在，则报错返回；
- 此操作会改变表的”LastDataModifiedTime”的值，此时， MaxCompute 会认为表或分区的数据有变动，生命周期的计算会重新开始。

修改分区值

MaxCompute SQL支持通过rename操作更改对应表的分区值。

语法格式：

```
ALTER TABLE table_name PARTITION (partition_col1 = partition_col_value1, partition_col2 = partiton_col_value2, ...)
RENAME TO PARTITION (partition_col1 = partition_col_newvalue1, partition_col2 = partiton_col_newvalue2, ...);
```

说明：

- 不支持修改分区列列名,只能修改分区列对应的值。
- 修改多级分区的一个或者多个分区值，多级分区的每一级的分区值都必须写上。

更新表中的数据(INSERT OVERWRITE/INTO)

语法格式：

```
INSERT OVERWRITE|INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]
select_statement
FROM from_statement;
```

备注:

- MaxCompute 的Insert语法与通常使用的MySQL或Oracle的Insert语法有差别，在insert overwrite|into 后需要加入table关键字，后不是直接使用tablename。

在MaxCompute SQL处理数据的过程中，insert overwrite/into用于将计算的结果保存目标表中。insert into与insert overwrite的区别是，insert into会向表或表的分区中追加数据，而insert overwrite则会在向表或分区中插入数据前清空表中的原有数据。在使用MaxCompute 处理数据的过程中，insert overwrite/into是最常用到的语句，它们会将计算的结果保存一个表中，可以供下一步计算使用。如，可以用如下操作计算sale_detail表中不同地区的销售额

```
create table sale_detail_insert like sale_detail;
alter table sale_detail_insert add partition(sale_date='2013', region='china');
insert overwrite table sale_detail_insert partition (sale_date='2013', region='china')
select shop_name, customer_id, total_price from sale_detail;
```

需要注意的是，在进行insert更新数据操作时，源表与目标表的对应关系依赖于在select子句中列的顺序，而不是表与表之间列名的对应关系，下面的SQL语句仍然是合法的：

```
insert overwrite table sale_detail_insert partition (sale_date='2013', region='china')
select customer_id, shop_name, total_price from sale_detail;
-- 在创建sale_detail_insert表时，列的顺序为：
-- shop_name string, customer_id string, total_price bigint
-- 而从sale_detail向sale_detail_insert插入数据是，sale_detail的插入顺序为：
-- customer_id, shop_name, total_price
-- 此时，会将sale_detail.customer_id的数据插入sale_detail_insert.shop_name
-- 将sale_detail.shop_name的数据插入sale_detail_insert.customer_id
```

向某个分区插入数据时，分区列不允许出现在select列表中：

```
insert overwrite table sale_detail_insert partition (sale_date='2013', region='china')
select shop_name, customer_id, total_price, sale_date, region from sale_detail;
-- 报错返回，sale_date, region为分区列，不允许出现在静态分区的 insert 语句中。
```

同时，partition的值只能是常量，不可以出现表达式。以下用法是非法的：

```
insert overwrite table sale_detail_insert partition (sale_date=datepart('2016-09-18 01:10:00', 'yyyy') ,
region='china')
select shop_name, customer_id, total_price from sale_detail;
```

多路输出(MULTI INSERT)

MaxCompute SQL支持在一个语句中插入不同的结果表或者分区

语法格式：

```
    FROM from_statement
INSERT OVERWRITE | INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)]
select_statement1
[INSERT OVERWRITE | INTO TABLE tablename2 [PARTITION (partcol1=val3, partcol2=val4 ...)]
select_statement2]
```

说明：

- 一般情况下，单个SQL里最多可以写128路输出，超过128路报语法错误。
- 在一个multi insert中，对于分区表，同一个目标分区不允许出现多次; 对于未分区表，该表不能出现多次。
- 对于同一张分区表的不同分区，不能同时有insert overwrite和insert into操作，否则报错返回。

如，

```
    create table sale_detail_multi like sale_detail;

from sale_detail
insert overwrite table sale_detail_multi partition (sale_date='2010', region='china' )
select shop_name, customer_id, total_price
insert overwrite table sale_detail_multi partition (sale_date='2011', region='china' )
select shop_name, customer_id, total_price;
-- 成功返回，将sale_detail的数据插入到sales里的2010年及2011年中国大区的销售记录中

from sale_detail
insert overwrite table sale_detail_multi partition (sale_date='2010', region='china' )
select shop_name, customer_id, total_price
insert overwrite table sale_detail_multi partition (sale_date='2010', region='china' )
select shop_name, customer_id, total_price;
-- 出错返回，同一分区出现多次

from sale_detail
insert overwrite table sale_detail_multi partition (sale_date='2010', region='china' )
select shop_name, customer_id, total_price
insert into table sale_detail_multi partition (sale_date='2011', region='china' )
select shop_name, customer_id, total_price;
-- 出错返回，同一张表的不同分区，不能同时有insert overwrite和insert into操作
```

输出到动态分区(DYNAMIC PARTITION)

在insert overwrite到一张分区表时，可以在语句中指定分区的值。也可以用另外一种更加灵活的方式，在分区中指定一个分区列名，但不给出值。相应的，在select子句中的对应列来提供分区的值。

语法格式：

```
    insert overwrite table tablename partition (partcol1, partcol2 ...) select_statement from from_statement;
```

说明：

- 目前，在使用动态分区功能的SQL中，在分布式环境下，单个进程最多只能输出512个动态分区，否则引发运行时异常；
- 在现阶段，任意动态分区SQL不允许生成超过2000个动态分区，否则引发运行时异常；
- 动态生成的分区值不允许以为NULL，否则会引发异常；
- 如果目标表有多级分区，在运行insert语句时允许指定部分分区为静态，但是静态分区必须是高级分区；

下面，我们使用一个简单的例子来说明动态分区：

```
    create table total_revenues (revenue bigint) partitioned by (region string);
insert overwrite table total_revenues partition(region)
select total_price as revenue, region
from sale_detail;
```

按照这种写法，在SQL运行之前，是不知道会产生哪些分区的，只有在select运行结束后，才能由region字段产生的值确定会产生哪些分区，这也是为什么叫做”动态分区”的原因。

其他示例：

```
    create table sale_detail_dypart like sale_detail;
insert overwrite table sale_detail_dypart partition (sale_date, region)
select * from sale_detail;
-- 成功返回

insert overwrite table sale_detail_dypart partition (sale_date='2013', region)
select shop_name,customer_id,total_price,region from sale_detail;
-- 成功返回，多级分区，指定一级分区

insert overwrite table sale_detail_dypart partition (sale_date='2013', region)
select shop_name,customer_id,total_price from sale_detail;
-- 失败返回，动态分区插入时，动态分区列必须在select列表中

insert overwrite table sales partition (region='china', sale_date)
select shop_name,customer_id,total_price,region from sale_detail;
-- 失败返回，不能仅指定低级子分区，而动态插入高级分区
```

SELECT操作

语法格式：

```
    SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[ORDER BY order_condition]
[DISTRIBUTE BY distribute_condition [SORT BY sort_condition] ]
[LIMIT number]
```

在使用select语句时需要注意如下几点：

- select操作从表中读取数据，要读的列可以用列名指定，或者用*代表所有的列，一个简单的select如下:

```
select * from sale_detail;
```

或者只读取sale_detail的一列shop_name

```
select shop_name from sale_detail;
```

在where中可以指定过滤的条件，如

```
select * from sale_detail where shop_name like 'hang%';
```

备注:

- 请注意，当使用Select语句屏显时，目前最多只能显示1000行结果。当select作为子句时，无此限制，select子句会将全部结果返回给上层查询。

- where子句：支持的过滤条件包括：

过滤条件	描述
>, < , =, >=, <=, <>	
like, rlike	
in, not in	如果在in/not in条件后加子查询，子查询只能返回一列值，且返回值的数量不能超过1000。

在select语句的where子句中可以指定分区范围，这样可以仅仅扫描表的指定部分，避免全表扫描。如下所示：

```
select sale_detail.*
from sale_detail
where sale_detail.sale_date >= '2008' and sale_detail.sale_date <= '2014';
```

MaxCompute SQL的where子句不支持between条件查询。

- 在table_reference中支持使用嵌套子查询，如：

```
select * from (select region from sale_detail) t where region = 'shanghai';
```

- distinct：如果有重复数据行时，在字段前使用distinct，会将重复字段去重，只返回一个值，而使用all将返回字段中所有重复的值，不指定此选项时默认效果和all相同。使用distinct只返回一行记录

，如

```
    select distinct region from sale_detail;
select distinct region, sale_date from sale_detail;
-- distinct多列，distinct的作用域是select的列集合，不是单个列。
```

- group by：分组查询, 一般group by是和聚合函数配合使用。在select中包含聚合函数时：

1. 用group by的key可以是输入表的列名;
2. 也可以是由输入表的列构成的表达式，不允许是select语句的输出列的别名;
3. 规则1的优先级高于规则2。当规则1和规则2发生冲突时，即group by的key即是输入表的列或表达式，又是select的输出列，以规则1为准。

如果如

```
    select region from sale_detail group by region;
-- 直接使用输入表列名作为group by的列，可以运行

select sum(total_price) from sale_detail group by region;
-- 以region值分组，返回每一组的销售额总量，可以运行

select region, sum(total_price) from sale_detail group by region;
-- 以region值分组，返回每一组的region值(组内唯一)及销售额总量，可以运行



select region as r from sale_detail group by r;
-- 使用select列的别名运行，报错返回

select 'China-' + region as r from sale_detail group by 'China-' + region;
-- 必须使用列的完整表达式



select region, total_price from sale_detail group by region;
-- 报错返回，select的所有列中，没有使用聚合函数的列，必须出现在group by中

select region, total_price from sale_detail group by region, total_price;
-- 可以运行
```

有这样的限制是因为，在SQL解析中，group by操作通常是先于select操作的，因此group by只能接受输入表的列或表达式为key。

备注:

- 关于聚合函数的介绍请参考 **聚合函数**

- order by：对所有数据按照某几列进行全局排序。如果您希望按照降序对记录进行排序，可以使用DESC关键字。由于是全局排序，order by必须与limit共同使用。对在使用order by排序时，NULL会被认为比任何值都小，这个行为与Mysql一致，但是与Oracle不一致。与group by不同，order by后

面必须加select列的别名，当select某列时，如果没有指定列的别名，将列名作为列的别名。

```
    select * from sale_detail order by region;
-- 报错返回，order by没有与limit共同使用

select * from sale_detail order by region limit 100;

select region as r from sale_detail order by region;
-- 报错返回，order by后面必须加列的别名。

select region as r from sale_detail order by r;
```

[limit number]的number是常数，限制输出行数。当使用无limit的select语句直接从屏幕输出查看结果时，最多只输出5000行。每个项目空间的这个屏显最大限制限制可能不同，可以通过控制台面板控制。

distribute by：对数据按照某几列的值做hash分片，必须使用select的输出列别名。

```
    select region from sale_detail distribute by region;
-- 列名即是别名，可以运行

select region as r from sale_detail distribute by region;
-- 报错返回，后面必须加列的别名。

select region as r from sale_detail distribute by r;
```

- sort by：局部排序，语句前必须加distribute by。实际上sort by是对distribute by的结果进行局部排序。必须使用select的输出列别名。

```
    select region from sale_detail distribute by region sort by region;
select region as r from sale_detail sort by region;
-- 没有distribute by，报错退出。
```

- order by不和distribute by/sort by共用，同时group by也不和distribute by/sort by共用，必须使用select的输出列别名。

备注:

- order by/sort by/distribute by的key必须是select语句的输出列，即列的别名。在MaxCompute SQL解析中， order by/sort by/distribute by是后于select操作的，因此它们只能接受select语句的输出列为key。

子查询

普通的select是从几张表中读数据，如select column_1, column_2 ... from table_name，但查询的对象也可以是另外一个select操作，如：

```
select * from (select shop_name from sale_detail) a;
```

备注：

- 子查询必须要有别名。

在from子句中，子查询可以当作一张表来使用，与其它的表或子查询进行join操作，如

```
create table shop as select * from sale_detail;

select a.shop_name, a.customer_id, a.total_price from
(select * from shop) a join sale_detail on a.shop_name = sale_detail.shop_name;
```

UNION ALL

语法格式：

```
select_statement UNION ALL select_statement
```

将两个或多个select操作返回的数据集联合成一个数据集，如果结果有重复行时，会返回所有符合条件的行，不进行重复行的去重处理。需要注意的是：MaxCompute SQL不支持顶级的两个查询结果合并，要改写为一个子查询的形式，如

```
select * from sale_detail where region = 'hangzhou'
union all
select * from sale_detail where region = 'shanghai';
```

需要改成：

```
select * from (
select * from sale_detail where region = 'hangzhou'
union all
select * from sale_detail where region = 'shanghai')
t;
```

备注:

- union all操作对应的各个子查询的列个数、名称和类型必须一致。如果列名不一致时，可以使用列的别名加以解决。

- 一般情况下，MaxCompute 最多允许128路union all，超过此限制报语法错误。

JOIN操作

MaxCompute 的JOIN支持多路间接，但不支持笛卡尔积，即无on条件的链接。语法定义：

```
    join_table:
table_reference join table_factor [join_condition]
| table_reference {left outer|right outer|full outer|inner} join table_reference join_condition

table_reference:
table_factor
| join_table

table_factor:
tbl_name [alias]
| table_subquery alias
| ( table_references )

join_condition:
on equality_expression ( and equality_expression )*
```

备注:

- equality_expression是一个等式表达式

left join 会从左表(shop)那里返回所有的记录，即使在右表(sale_detail)中没有匹配的行。

```
    select a.shop_name as ashop, b.shop_name as bshop from shop a
left outer join sale_detail b on a.shop_name=b.shop_name;
-- 由于表shop及sale_detail中都有shop_name列，因此需要在select子句中使用别名进行区分。
```

right outer join 右连接，返回右表中的所有记录，即使在左表中没有记录与它匹配，例如：

```
    select a.shop_name as ashop, b.shop_name as bshop from shop a
right outer join sale_detail b on a.shop_name=b.shop_name;
```

full outer join 全连接，返回左右表中的所有记录，例如：

```
    select a.shop_name as ashop, b.shop_name as bshop from shop a
full outer join sale_detail b on a.shop_name=b.shop_name;
```

在表中存在至少一个匹配时，inner join 返回行。 关键字inner可省略。

```
    select a.shop_name from shop a inner join sale_detail b on a.shop_name=b.shop_name;
```

```
select a.shop_name from shop a join sale_detail b on a.shop_name=b.shop_name;
```

连接条件，只允许and连接的等值条件，并且最多支持16路join操作。只有在MAPJOIN中，可以使用不等值连接或者使用or连接多个条件。

```
    select a.* from shop a full outer join sale_detail b on a.shop_name=b.shop_name
full outer join sale_detail c on a.shop_name=c.shop_name;
-- 支持多路join链接示例，最多支持16路join

select a.* from shop a join sale_detail b on a.shop_name != b.shop_name;
-- 不支持不等值Join链接条件，报错返回。
```

MAPJOIN HINT

当一个大表和一个或多个小表做join时，可以使用mapjoin，性能比普通的join要快很多。mapjoin的基本原理是：在小数据量情况下，SQL会将用户指定的小表全部加载到执行join操作的程序的内存中，从而加快join的执行速度。需要注意，使用mapjoin时：

- left outer join的左表必须是大表；
- right outer join的右表必须是大表；
- inner join左表或右表均可以作为大表；
- full outer join不能使用mapjoin；
- mapjoin支持小表为子查询；
- 使用mapjoin时需要引用小表或是子查询时，需要引用别名；
- 在mapjoin中，可以使用不等值连接或者使用or连接多个条件；
- 目前MaxCompute 在mapjoin中最多支持指定6张小表，否则报语法错误；
- 如果使用mapjoin，则所有小表占用的内存总和不得超过512MB。请注意由于MaxCompute 是压缩存储，因此小表在被加载到内存后，数据大小会急剧膨胀。此处的512MB限制是加载到内存后的空间大小；
- 多个表join时，最左边的两个表不能同时是mapjoin的表。

下面是一个简单的示例：

```
    select /* + mapjoin(a) */
a.shop_name,
b.customer_id,
b.total_price
from shop a join sale_detail b
on a.shop_name = b.shop_name;
```

MaxCompute SQL不支持支持在普通join的on条件中使用不等值表达式、or 逻辑等复杂的join条件，但是在mapjoin中可以进行如上操作，例如：

```
    select /*+ mapjoin(a) */
```

```
a.total_price,
b.total_price
from shop a join sale_detail b
on a.total_price < b.total_price or a.total_price + b.total_price < 500;
```

HAVING子句

由于MaxCompute SQL的WHERE关键字无法与合计函数一起使用，可以采用having字句。

语法格式：

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

使用场景举例：比如有一张订单表Orders，包括客户名称（Customer,），订单金额（OrderPrice），订单日期（Order_date），订单号（Order_id）四个字段。现在希望查找订单总额少于2000的客户。现在我们可以写如下语句：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING SUM(OrderPrice)<2000
```

数学函数

ABS

函数定义：

```
    double abs(double number)
bigint abs(bigint number)
```

用途：返回绝对值。

参数说明：

- number：Double或bigint类型，输入为bigint时返回bigint，输入为double时返回double类型。若输入为string类型会隐式转换到double类型后参与运算，其它类型抛异常。

返回值：Double或者bigint类型，取决于输入参数的类型。若输入为null，返回null。

备注：

- 当输入bigint类型的值超过bigint的最大表示范围时，会返回double类型，这种情况下可能会损失精度。

示例：

```
   abs(null) =  null
abs(-1) = 1
abs(-1.2) = 1.2
abs("-2") = 2.0
abs(12232083745629837659238745692374 8) = 1.2232083745629837e32
```

下面是一个完整的abs函数在SQL中使用的例子，其他内建函数(除窗口函数、聚合函数外)的使用方式与其类似，不再一一举例：

```
   select abs(id) from tbl1;
-- 取tbl1表内id字段的绝对值
```

ACOS

函数定义：

```
   double acos(double number)
```

用途：计算number的反余弦函数。

参数说明：

- number：Double类型，-1≤number≤1。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Double类型，值域在0 ~ π 之间。若number为NULL，返回NULL。

示例：

```
   acos("0.87") = 0.5155940062460905
acos(0) = 1.5707963267948966
```

ASIN

函数定义：

```
   double asin(double number)
```

用途：反正弦函数。

参数说明：

- number：Double类型，-1≤number≤1。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Double类型，值域在-π/2 ~π/2之间。若number为NULL，返回NULL。

示例：

```
    asin(1) = 1.5707963267948966
asin(-1) = -1.5707963267948966
```

ATAN

函数定义：

```
double atan(double number)
```

用途：反正切函数。

参数说明：

- number：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Double类型，值域在-π/2 ~π/2之间。若number为NULL，返回NULL。

示例：

```
    atan(1) = 0.7853981633974483
atan(-1) = -0.7853981633974483
```

CEIL

函数定义：

```
bigint ceil(double value)
```

用途：返回不小于输入值value的最小整数

参数说明：

- value：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Bigint类型。任一输入为NULL，返回NULL。

示例：

```
    ceil(1.1) = 2
ceil(-1.1) = -1
```

CONV

函数定义：

```
    string conv(string input, bigint from_base, bigint to_base)
```

用途：进制转换函数

参数说明：

- input：以string表示的要转换的整数值，接受bigint，double的隐式转换。
- from_base，to_base：以十进制表示的进制的值，可接受的的值为2，8，10，16。接受string及double的隐式转换。

返回值：String类型。任一输入为NULL，返回NULL。转换过程以64位精度工作，溢出时报异常。输入如果是负值，即以”-“开头，报异常。如果输入的是小数，则会转为整数值后进行进制转换，小数部分会被舍弃。

示例

```
    conv('1100', 2, 10) = '12'
conv('1100', 2, 16) = 'c'
conv('ab', 16, 10) = '171'
conv('ab', 16, 16) = 'ab'
```

COS

函数定义：

```
    double cos(double number)
```

用途：余弦函数，输入为弧度值。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。

返回值：Double类型。若number为NULL，返回NULL。

示例：

```
    cos(3.1415926/2)=2.6794896585028633e-8
cos(3.1415926)=0.9999999999999986
```

COSH

函数定义：

```
    double cosh(double number)
```

用途：双曲余弦函数。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型。若number为NULL，返回NULL。

COT

函数定义：

```
    double cot(double number)
```

用途：余切函数，输入为弧度值。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型。若number为NULL，返回NULL。

EXP

函数定义：

```
    double exp(double number)
```

用途：指数函数。返回number的指数值。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其

他类型抛异常。返回值：Double类型。若number为NULL，返回NULL。

FLOOR

函数定义：

```
bigint floor(double number)
```

用途：向下取整，返回比number小的整数值。

参数说明：

- number：Double类型，若输入为string类型或bigint型会隐式转换到double类型后参与运算，其他类型抛异常返回值：返回Bigint类型。若number为NULL，返回NULL。

示例

```
floor(1.2)=1
floor(1.9)=1
floor(0.1)=0
floor(-1.2)=-2
floor(-0.1)=-1
floor(0.0)=0
floor(-0.0)=0
```

LN

函数定义：

```
double ln(double number)
```

用途：返回number的自然对数。

参数说明：

- number：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。若number为NULL返回NULL,若number为负数或零，则抛异常。返回值：Double类型。

LOG

函数定义：

```
double log(double base, double x)
```

用途：返回以base为底的x的对数。

参数说明：

- base：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。
- x：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型的对数值，若base和x中存在NULL，则返回NULL；若base和x中某一个值为负数或0，会引发异常；若base为1(会引发一个除零行为)也会引发异常。

POW

函数定义：

```
double pow(double x, double y)
```

用途：返回x的y次方，即x^y。

参数说明：

- X：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。
- Y：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型。若x或y为NULL，则返回NULL

RAND

函数定义：

```
double rand(bigint seed)
```

用途：以seed为种子返回double类型或Decimal类型的随机数，返回值区间是的0～1。

参数说明：

- seed：可选参数，Bigint类型，随机数种子，决定随机数序列的起始值。

返回值：Double类型或Decimal类型。

示例：

```
select rand() from dual;
select rand(1) from dual;
```

ROUND

函数定义：

```
double round(double number, [bigint decimal_places])
```

用途：四舍五入到指定小数点位置。

参数说明：

- number：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。
- decimal_place：Bigint类型常量，四舍五入计算到小数点后的位置，其他类型参数会引发异常. 如果省略表示四舍五入到个位数。默认值为0。返回值：返回Double类型。若number或decimal_places为NULL，返回NULL。

备注：

- decimal_places可以是负数。负数会从小数点向左开始计数，并且不保留小数部分；如果decimal_places超过了整数部分长度，返回0.示例：

```
round(125.315) = 125.0
round(125.315, 0) = 125.0
round(125.315, 1) = 125.3
round(125.315, 2) = 125.32
round(125.315, 3) = 125.315
round(-125.315, 2) = -125.32
round(123.345, -2) = 100.0
round(null) = null
round(123.345, 4) = 123.345
round(123.345, -4) = 0.0
```

SIGN

函数声明：

```
sign(x)
```

用途：判断x是否为正值或者是否为负值。参数说明：x:Double类型或者Decimal类型，可以为常量、函数或者表达式。返回值：

- 当x为正值时，返回1.0。
- 当x为负值时，返回-1.0。
- 当x为0时，返回0.0。
- 当x为空时，抛异常。

示例：

```
select sign(5-13) from dual;
返回：
+------------+
| _c0 |
+------------+
| -1.0 |
+------------+
```

SIN

函数定义：

```
double sin(double number)
```

用途：正弦函数，输入为弧度值。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型。若number为NULL，返回NULL。

SINH

函数定义：

```
double sinh(double number)
```

用途：双曲正弦函数。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型。若number为NULL，返回NULL。

SQRT

函数定义：

```
double sqrt(double number)
```

用途：计算平方根。

参数说明：

- number：Double类型，必须大于0。小于0时引发异常。若输入为string类型或bigint类型会隐式转

换到double类型后参与运算，其他类型抛异常。返回值：返回double类型。若number为NULL，返回NULL。

TAN

函数声明：

```
double tan(double number)
```

用途：正切函数，输入为弧度值。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型。若number为NULL，返回NULL。

TANH

函数声明：

```
double tanh(double number)
```

用途：双曲正切函数。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。返回值：Double类型。若number为NULL，返回NULL。

TRUNC

函数声明：

```
double trunc(double number[, bigint decimal_places])
```

用途：将输入值number截取到指定小数点位置。

参数说明：

- number：Double类型，若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。
- decimal_places：Bigint类型常量，要截取到的小数点位置，其他类型参数会隐式转为bigint，省略此参数时默认到截取到个位数。返回值：返回值类型为Double。若number或decimal_places为NULL，返回NULL。

备注:

- 截取掉的部分补0。
- decimal_places可以是负数，负数会从小数点向左开始截取，并且不保留小数部分；如果decimal_places超过了整数部分长度，返回0。

示例：

```
    trunc(125.815) = 125.0
trunc(125.815, 0) =125.0
trunc(125.815, 1) = 125.8
trunc(125.815, 2) = 125.81
trunc(125.815, 3) = 125.815
trunc(-125.815, 2) = -125.81
trunc(125.815, -1) = 120.0
trunc(125.815, -2) = 100.0
trunc(125.815, -3) = 0.0
trunc(123.345, 4) = 123.345
trunc(123.345, -4) = 0.0
```

字符串函数

CHAR_MATCHCOUNT

函数声明：

```
    bigint char_matchcount(string str1, string str2)
```

用途：用于计算str1中有多少个字符出现在str2中。

参数说明：

- str1，str2：String类型，必须为有效的UTF-8字符串，如果对比中发现有无效字符则函数返回负值。

返回值：Bigint类型。任一输入为NULL返回NULL。

示例：

```
    char_matchcount('abd', 'aabc') = 2
-- str1中得两个字符串'a', 'b'在str2中出现过
```

CHR

函数声明：

```
string chr(bigint ascii)
```

用途：将给定ASCII码ascii转换成字符。

参数说明：

- ascii：Bigint类型ASCII值，若输入为string类型或double类型会隐式转换到bigint类型后参与运算，其它类型抛异常。

返回值：String类型。参数范围是0~255，超过此范围会引发异常。输入值为NULL返回NULL。

CONCAT

函数声明：

```
string concat(string a, string b...)
```

用途：返回值是将参数中的所有字符串连接在一起的结果。

参数说明：

- a，b等为String类型，若输入为bigint，double或datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。如果没有参数或者某个参数为NULL，结果均返回NULL。

示例：

```
concat('ab', 'c') = 'abc'
concat() = NULL
concat('a', null, 'b') = NULL
```

GET_JSON_OBJECT

函数声明:

```
STRING GET_JSON_OBJECT(STRING json,STRING path)
```

用途：在一个标准json字符串中，按照path抽取指定的字符串.

参数说明：

- json:String类型，标准的json格式字符串。
- path: **String类型，用于描述在json中的path，以$开头。** 关于新实现中json path的说明：参考 http://goessner.net/articles/JsonPath/index.html#e2***$表示根节点*** “.”表示child

"[number]"表示数组下标 对于数组，格式为 key[sub1][sub2][sub3]……***[]返回整个数组*** 不支持转义

返回值：String类型

注解：

- 如果json为空或者非法的json格式，返回NULL
- 如果path为空或者不合法（json中不存在）返回NULL
- 如果json合法，path也存在则返回对应字符串

示例1

```
+-----+
json
+-----+
{"store":
{"fruit":\[{"weight":8,"type":"apple"},{"weight":9,"type":"pear"}],
"bicycle":{"price":19.95,"color":"red"}
},
"email":"amy@only_for_json_udf_test.net",
"owner":"amy"
}
```

通过以下查询，可以提取json对象中的信息：

```
 odps> SELECT get_json_object(src_json.json, '$.owner') FROM src_json;
amy
odps> SELECT get_json_object(src_json.json, '$.store.fruit\[0]') FROM src_json;
{"weight":8,"type":"apple"}
odps> SELECT get_json_object(src_json.json, '$.non_exist_key') FROM src_json;
NULL
```

示例2

```
get_json_object('{"array":[[aaaa,1111],[bbbb,2222],[cccc,3333]]}','$.array[1].[1]') = "2222"

get_json_object('{"aaa":"bbb","ccc":{"ddd":"eee","fff":"ggg","hhh":["h0","h1","h2"]},"iii":"jjj"}','$.ccc.hhh[*]') =
"["h0","h1","h2"]"

get_json_object('{"aaa":"bbb","ccc":{"ddd":"eee","fff":"ggg","hhh":["h0","h1","h2"]},"iii":"jjj"}','$.ccc.hhh[1]') = "h1"
```

INSTR

函数声明：

```
bigint instr(string str1, string str2[, bigint start_position[, bigint nth_appearance]])
```

用途：计算一个子串str2在字符串str1中的位置。

参数说明：

- str1：String类型，搜索的字符串，若输入为bigint，decimal，double或datetime类型会隐式转换为string后参与运算，其它类型报异常。
- str2：String类型，要搜索的子串，若输入为bigint，decimal，double或datetime类型会隐式转换为string后参与运算，其它类型报异常。
- start_position：Bigint类型，其它类型会抛异常，表示从str1的第几个字符开始搜索，默认起始位置是第一个字符位置1。当start_position为负数时表示开始位置是从字符串的结尾往前倒数，最后一个字符是-1，往前数依次就是-2，-3…。
- nth_appearance：Bigint类型，大于0，表示子串在字符串中的第nth_appearance次匹配的位置，如果nth_appearance为其它类型或小于等于0会抛异常。

返回值：Bigint类型。

备注:

- 如果在str1中未找到str2，返回0。
- 任一输入参数为NULL返回NULL
- 如果str2为空串时总是能匹配成功，因此instr（‘abc’，‘’）会返回1。

示例：

```
instr('Tech on the net', 'e') = 2
instr('Tech on the net', 'e', 1, 1) = 2
instr('Tech on the net', 'e', 1, 2) = 11
instr('Tech on the net', 'e', 1, 3) = 14
instr('Tech on the net', 'e', -1, 1) = 14
instr('Tech on the net', 'e', -3, 2) = 2
instr('Tech on the net', 'o', -1, 2) = 0
```

IS_ENCODING

函数声明：

```
boolean is_encoding(string str, string from_encoding, string to_encoding)
```

用途：判断输入字符串str是否可以从指定的一个字符集from_encoding转为另一个字符集to_encoding。可用于判断输入是否为”乱码”，通常的用法是将from_encoding设为”utf-8”，to_encoding设为”gbk”。

参数说明：

- str：String类型，输入为NULL返回NULL。空字符串则可以被认为属于任何字符集。

- from_encoding，to_encoding：String类型，源及目标字符集。输入为NULL返回NULL。

返回值：Boolean类型，如果str能够成功转换，则返回true，否则返回false

示例：

```
    is_encoding('测试', 'utf-8', 'gbk') = true
is_encoding('測試', 'utf-8', 'gbk') = true
-- gbk字库中有这两个繁体字
is_encoding('測試', 'utf-8', 'gb2312') = false
-- gb2312库中不包括这两个字
```

KEYVALUE

函数声明:

```
KEYVALUE(STRING srcStr，STRING split1，STRING split2， STRING key)
KEYVALUE(STRING srcStr， STRING key) //split1 = ";"，split2 = ":"
```

用途: 将srcStr（源字符串）按split1分成“key-value”对，按split2将key-value对分开，返回“key”所对应的value。

参数说明：

- srcStr 输入待拆分的字符串。
- key：string类型。源字符串按照split1和split2拆分后，根据该key值的指定，返回其对应的value。
- split1, split2：用来作为分隔符的字符串，按照指定的这两个分隔符拆分源字符串。如果表达式中没有指定这两项，默认split1为’;’，split2为’:’。当某个被split1拆分后的字符串中有多个split2时，返回结果未定义；

返回值:

- String类型。
- Split1或split2为NULL时，返回NULL。
- srcStr,key为NULL或者没有匹配的key时，返回NULL。
- 如果有多个key-value匹配，返回第一个匹配上的key对应的value。

示例：

```
keyvalue('0:1\;1:2', 1) = '2'
-源字符串为“0:1\;1:2”，因为没有指定split1和split2，默认split1为";"，split2为“:”。经过split1拆分后，key-value对为：
0:1\,1:2
经过split2拆分后变成：
0 1/
1 2
返回key为1所对应的value值，为2。

keyvalue("\;decreaseStore:1\;xcard:1\;isB2C:1\;tf:21910\;cart:1\;shipping:2\;pf:0\;market:shoes\;instPayAmount:0\;",
```

```
"\;",":","tf") = "21910"
-源字符串为
 “\;decreaseStore:1\;xcard:1\;isB2C:1\;tf:21910\;cart:1\;shipping:2\;pf:0\;market:shoes\;instPayAmount:0\;” ，按照split1“\;”拆分后，得出的key-value对为：
decreaseStore:1，xcard:1，isB2C:1，tf:21910，cart:1，shipping:2，pf:0，market:shoes，instPayAmount:0
按照split2":"拆分后变成：
decreaseStore 1
xcard 1
isB2C 1
tf 21910
cart 1
shipping 2
pf 0
market shoes
instPayAmount 0
key值为“tf”,返回其对应的value:21910。

keyvalue("阿里云=飞天=2;飞天=数据平台", ";","=", "阿里云") 返回NULL，请用户避免这种用法。
```

LENGTH

命令格式：

```
bigint length(string str)
```

用途：返回字符串str的长度。

参数说明：

- str：String类型，若输入为bigint，double或datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：Bigint类型。若str是NULL返回NULL。如果str非UTF-8编码格式，返回-1。

示例

```
length('hi! 中国') = 6
```

LENGTHB

函数声明：

```
bigint lengthb(string str)
```

用途：返回字符串str的以字节为单位的长度。

参数说明：

- str：String类型，若输入为bigint，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：Bigint类型。若str是NULL返回NULL。

示例

```
lengthb('hi! 中国') = 10
```

MD5

函数声明：

```
string md5(string value)
```

用途：计算输入字符串value的md5值

参数说明：

- value：String类型，如果输入类型是bigint，double或者datetime会隐式转换成string类型参与运算，其它类型报异常。输入为NULL，返回NULL。

返回值：String类型。

REGEXP_EXTRACT

函数声明：

```
string regexp_extract(string source, string pattern[, bigint occurrence])
```

用途：将字符串source按照pattern正则表达式的规则拆分，返回第occurrence个group的字符。

参数说明：

- source：String类型，待搜索的字符串。
- pattern：String类型常量，pattern为空串时抛异常，pattern中如果没有指定group，抛异常。
- occurrence：Bigint类型常量，必须>=0，其它类型或小于0时抛异常，不指定时默认为1，表示返回第一个group。若occurrence = 0，返回满足整个pattern的子串。

返回值：String类型，任一输入为NULL返回NULL。

示例：

```
regexp_extract('foothebar', 'foo(.*?)(bar)', 1) = the
regexp_extract('foothebar', 'foo(.*?)(bar)', 2) = bar
regexp_extract('foothebar', 'foo(.*?)(bar)', 0) = foothebar
```

```
regext_extract('8d99d8', '8d(\\d+)d8') = 99
-- 如果是在MaxCompute客户端上提交正则计算的SQL，需要使用两个"\"作为转移字符

regexp_extract('foothebar', 'foothebar')
-- 异常返回，pattern中没有指定group
```

REGEXP_INSTR

命令格式：

```
    bigint regexp_instr(string source, string pattern[,
bigint start_position[, bigint nth_occurrence[, bigint return_option]])
```

用途：返回字符串source从start_position开始，和pattern第n次(nth_occurrence)匹配的子串的起始/结束位置。任一输入参数为NULL时返回NULL。

参数说明：

- source：String类型，待搜索的字符串。
- pattern：String类型常量，pattern为空串时抛异常。
- start_position：Bigint类型常量，搜索的开始位置。不指定时默认值为1，其它类型或小于等于0的值会抛异常。
- nth_occurrence：Bigint类型常量，不指定时默认值为1，表示搜索第一次出现的位置。小于等于0或者其它类型抛异常。
- return_option：Bigint类型常量，值为0或1，其它类型或不允许的值会抛异常。0表示返回匹配的开始位置，1表示返回匹配的结束位置。

返回值：Bigint类型。视return_option指定的类型返回匹配的子串在source中的开始或结束位置。

示例：

```
    regexp_instr("i love www.taobao.com", "o[[:alpha:]]{1}", 3, 2) = 14
```

REGEXP_REPLACE

函数声明：

```
    string regexp_replace(string source, string pattern, string replace_string[, bigint occurrence])
```

用途：将source字符串中第occurrence次匹配pattern的子串替换成指定字符串replace_string后返回。

参数说明：

- source：String类型，要替换的字符串。
- pattern：String类型常量，要匹配的模式，pattern为空串时抛异常。

- replace_string：String类型，将匹配的pattern替换成的字符串。
- occurrence：Bigint类型常量，必须大于等于0，表示将第几次匹配替换成replace_string，为0时表示替换掉所有的匹配子串。其它类型或小于0抛异常。可缺省，默认值为0。

返回值：String类型，当引用不存在的组时，不进行替换。当输入source，pattern，occurrence参数为NULL时返回NULL，若replace_string为NULL且pattern有匹配，返回NULL，replace_string为NULL但pattern不匹配，则返回原串 。

备注：

- 当引用不存在的组时，行为未定义。

示例：

```
    regexp_replace("123.456.7890", "([[:digit:]]{3})\\.([[:digit:]]{3})\\.([[:digit:]]{4})",
"(\\1)\\2-\\3", 0) = "(123)456-7890"
regexp_replace("abcd", "(.)", "\\1 ", 0) = "a b c d "
regexp_replace("abcd", "(.)", "\\1 ", 1) = "a bcd"
regexp_replace("abcd", "(.)", "\\2", 1) = "abcd"
-- 因为pattern中只定义了一个组，引用的第二个组不存在，
-- 请避免这样使用，引用不存在的组的结果未定义。
regexp_replace("abcd", "(.*)(.)$", "\\2", 0) = "d"
regexp_replace("abcd", "a", "\\1", 0) = "bcd"
-- 因为在pattern中没有组的定义，所以\1引用了不存在的组，
-- 请避免这样使用，引用不存在的组的结果未定义。
```

REGEXP_SUBSTR

函数声明：

```
    string regexp_substr(string source, string pattern[, bigint start_position[, bigint nth_occurrence]])
```

用途：从start_position位置开始，source中第nth_occurrence次匹配指定模式pattern的子串。

参数说明：

- source：String类型，搜索的字符串。
- pattern：String类型常量，要匹配的模型，pattern为空串时抛异常。
- start_position：Bigint常量，必须大于0。其它类型或小于等于0时抛异常，不指定时默认为1，表示从source的第一个字符开始匹配。不指定时默认为1，表示从source的第一个字符开始匹配。
- nth_occurrence：Bigint常量，必须大于0，其它类型或小于等于0时抛异常。不指定时默认为1，表示返回第一次匹配的子串。不指定时默认为1，表示返回第一次匹配的子串。

返回值：String类型。任一输入参数为NULL返回NULL。没有匹配时返回NULL。

示例：

```
    regexp_substr ("I love aliyun very much", "a[[:alpha:]]{5}") = "aliyun"
regexp_substr('I have 2 apples and 100 bucks!', '[[:blank:]][[:alnum:]]*', 1, 1) = " have"
regexp_substr('I have 2 apples and 100 bucks!', '[[:blank:]][[:alnum:]]*', 1, 2) = " 2"
```

REGEXP_COUNT

函数声明：

```
    bigint regexp_count(string source, string pattern[, bigint start_position])
```

用途：计算source中从start_position开始，匹配指定模式pattern的子串的次数。

参数说明：

- source：String类型，搜索的字符串，其它类型报异常。
- pattern：String类型常量，要匹配的模型，pattern为空串时抛异常，其它类型报异常。
- start_position：Bigint类型常量，必须大于0。其它类型或小于等于0时抛异常，不指定时默认为1，表示从source的第一个字符开始匹配。

返回值：Bigint类型。没有匹配时返回0。任一输入参数为NULL返回NULL。

示例：

```
    regexp_count('abababc', 'a.c') = 1
regexp_count('abcde', '[[:alpha:]]{2}', 3) = 1
```

SPLIT_PART

函数声明：

```
    string split_part(string str, string separator, bigint start[, bigint end])
```

用途：依照分隔符separator拆分字符串str，返回从第start部分到第end部分的子串(闭区间)。

参数说明：

- Str：String类型，要拆分的字符串。如果是bigint，double或者datetime类型会隐式转换到string类型后参加运算，其它类型报异常。
- Separator：String类型常量，拆分用的分隔符，可以是一个字符，也可以是一个字符串，其它类型会引发异常。
- start：Bigint类型常量，必须大于0。非常量或其它类型抛异常。返回段的开始编号(从1开始)，如果没有指定end，则返回start指定的段。
- end：Bigint类型常量，大于等于start，否则抛异常。返回段的截止编号，非常量或其他类型会引发异常。可省略，缺省时表示最后一部分。

返回值：String类型。若任意参数为NULL，返回NULL；若separator为空串，返回原字符串str。

备注:

- 如果separator不存在于str中，且start指定为1，返回整个str。若输入为空串，输出为空串。
- 如果start的值大于切分后实际的分段数，例如：字符串拆分完有6个片段，但start大于6，返回空串”“ 。
- 若end大于片段个数，按片段个数处理。

示例：

```
    split_part('a,b,c,d', ',', 1) = 'a'
split_part('a,b,c,d', ',', 1, 2) = 'a,b'
split_part('a,b,c,d', ',', 10) = ''
```

SUBSTR

函数声明：

```
    string substr(string str, bigint start_position[, bigint length])
```

用途：返回字符串str从start_position开始往后数，长度为length的子串。

参数说明：

- str：String类型，若输入为bigint，decimal，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。
- start_position：Bigint类型，起始位置为1。当start_position为负数时表示开始位置是从字符串的结尾往前倒数，最后一个字符是-1，往前数依次就是-2，-3...，其它类型抛异常。
- length：Bigint类型，大于0，其它类型或小于等于0抛异常。子串的长度。

返回值：String类型。若任一输入为NULL，返回NULL。

备注：

- 当length被省略时，返回到str结尾的子串。

示例

```
    substr("abc", 2) = "bc"
substr("abc", 2, 1) = "b"
substr("abc",-2,2")="bc"
substr("abc",-3)="abc"
```

TOLOWER

函数声明：

```
string tolower(string source)
```

用途：输出英文字符串source对应的小写字符串。

参数说明：

- source：String类型，若输入为bigint，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。输入为NULL时返回NULL。

示例：

```
tolower("aBcd") = "abcd"
tolower("哈哈Cd") = "哈哈cd"
```

TOUPPER

函数声明：

```
string toupper(string source)
```

用途：输出英文字符source串对应的大写字符串。

参数说明：

- source：String类型，若输入为bigint，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。输入为NULL时返回NULL。

示例：

```
toupper("aBcd") = "ABCD"
toupper("哈哈Cd") = "哈哈CD"
```

TO_CHAR

函数声明：

```
```

```
    string to_char(boolean value)
string to_char(bigint value)
string to_char(double value)
```

用途：将Boolean类型、bigint类型或者double类型转为对应的string类型表示

参数说明：

- value：可以接受boolean类型、bigint类型或者double类型输入，其它类型抛异常。对datetime类型的格式化输出请参考另一同名函数 TO_CHAR 。

返回值：String类型。如果输入为NULL，返回NULL。

示例：

```
    to_char(123) = '123'
to_char(true) = 'TRUE'
to_char(1.23) = '1.23'
to_char(null) = NULL
```

TRIM

函数声明：

```
    string trim(string str)
```

用途：将输入字符串str去除左右空格。

参数说明：

- str：String类型，若输入为bigint，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。输入为NULL时返回NULL。

LTRIM

函数声明：

```
 string ltrim(string str)
```

用途：将输入的字符串str去除左边空格。

参数说明：

- str：String类型，若输入为bigint，decimal，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。返回值：String类型。输入为NULL时返回NULL。

示例:

```
select ltrim(' abc ') from dual;
返回：
+-----+
| _c0 |
+-----+
| abc |
+-----+
```

RTRIM

函数声明：

```
 string rtrim(string str)
```

用途：将输入的字符串str去除右边空格。

参数说明：

- str：String类型，若输入为bigint，decimal，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。返回值：String类型。输入为NULL时返回NULL。

示例:

```
select rtrim('a abc ') from dual;
返回：
+-----+
| _c0 |
+-----+
| a abc |
+-----+
```

REVERSE

函数申明：

```
 STRING REVERSE(string str)
```

用途：返回倒序字符串。参数说明：

- str：String类型，若输入为bigint，double，decimal或datetime类型会隐式转换为string后参与运算，其它类型报异常。返回值：String类型。输入为NULL时返回NULL。

示例：

```
select reverse('abcedfg') from dual;

返回：
+-----+
| _c0 |
+-----+
| gfdecba |
+-----+
```

SPACE

函数声明：

```
STRING SPACE(bigint n)
```

用途：空格字符串函数，返回长度为n的字符串。参数说明：

- n: bigint类型。长度不超过2M。如果为空，则抛异常。返回值：String类型。

示例：

```
select length(space(10)) from dual;    ----返回10。
select space(400000000000) from dual；  ----报错，长度超过2M。
```

REPEAT

函数声明：

```
STRING REPEAT(string str, bigint n)
```

用途：返回重复n次后的str字符串。参数说明：

- str：String类型，若输入为bigint，double，decimal或datetime类型会隐式转换为string后参与运算，其它类型报异常。
- n: Bigint类型。长度不超过2M。如果为空，则抛异常。返回值：String类型。

示例：

```
 select repeat('abc',5) from lxw_dual;
返回：abcabcabcabcabc
```

ASCII

函数声明：

```
Bigint ASCII(string str)
```

用途：返回字符串str第一个字符的ascii码。参数说明：

- str：String类型，若输入为bigint，double，decimal或datetime类型会隐式转换为string后参与运算，其它类型报异常。返回值：Bigint类型。

示例：

```
select ascii('abcde') from dual;
返回值：97
```

日期函数

MaxCompute SQL提供了针对datetime类型的操作函数。

DATEADD

函数声明：

```
datetime dateadd(datetime date, bigint delta, string datepart)
```

用途：按照指定的单位datepart和幅度delta修改date的值。

参数说明：

- date：Datetime类型，日期值。若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。。
- delta：Bigint类型，修改幅度。若输入为string类型或double型会隐式转换到bigint类型后参与运算，其他类型会引发异常。若delta大于0，加；否则减。
- datepart：String类型常量。此字段的取值遵循string与datetime类型转换的约定，即”yyyy”表示年，” mm”表示月....关于类型转换的规则请参考 String类型与Datetime类型之间的转换 。此外也支持扩展的日期格式：年-“year” ，月-“month”或”mon” ，日-“day” ，小时-“hour”。非常量、不支持的格式会或其它类型抛异常。

返回值：Datetime类型。若任一输入参数为NULL，返回NULL。

备注:

- 按照指定的单位增减delta时导致的对更高单位的进位或退位，年、月、时、分、秒分别按照10进制、12进制、24进制、60进制、60进制计算。当delta的单位是月时，计算规则如下：若

datetime的月部分在增加delta值之后不造成day溢出，则保持day值不变，否则把day值设置为结果月份的最后一天。
- datepart的取值遵循string与datetime类型转换的约定，即”yyyy”表示年，” mm”表示月.... datetime相关的内建函数如无特殊说明均遵守此约定。同时如果没有特殊说明，所有datetime相关的内建函数的part部分也同样支持扩展的日期格式：年-“year” ，月-“month”或”mon” ，日-“day” ，小时-“hour”。

示例：

```
   若trans_date = 2005-02-28 00:00:00：
dateadd(trans_date, 1, 'dd') = 2005-03-01 00:00:00
-- 加一天，结果超出当年2月份的最后一天，实际值为下个月的第一天
dateadd(trans_date, -1, 'dd') = 2005-02-27 00:00:00
-- 减一天
dateadd(trans_date, 20, 'mm') = 2006-10-28 00:00:00
-- 加20个月，月份溢出，年份加1

若trans_date = 2005-02-28 00:00:00, dateadd(transdate, 1, 'mm') = 2005-03-28 00:00:00
若trans_date = 2005-01-29 00:00:00, dateadd(transdate, 1, 'mm') = 2005-02-28 00:00:00
-- 2005年2月没有29日，日期截取至当月最后一天
若trans_date = 2005-03-30 00:00:00, dateadd(transdate, -1, 'mm') = 2005-02-28 00:00:00
```

此处对trans_date的数值表示仅作示例使用，在文档中有关datetime介绍会经常使用到这种简易的表达方式。在MaxCompute SQL中，datetime类型没有直接的常数表示方式，如下使用方式是错误的：

```
   select dateadd(2005-03-30 00:00:00, -1, 'mm') from tbl1;
```

如果一定要描述datetime类型常量，请尝试如下方法：

```
   select dateadd(cast("2005-03-30 00:00:00" as datetime), -1, 'mm') from tbl1;
-- 将String类型常量显式转换为Datetime类型
```

DATEDIFF

命令格式：

```
   bigint datediff(datetime date1, datetime date2, string datepart)
```

用途：计算两个时间date1，date2在指定时间单位datepart的差值。

参数说明：

- datet1，date2：Datetime类型，被减数和减数，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。
- datepart：String类型常量。支持扩展的日期格式。若datepart不符合指定格式或者其它类型则会发

生异常。

返回值：Bigint类型。任一输入参数是NULL，返回NULL。如果date1小于date2，返回值可以为负数。

备注：

- 计算时会按照datepart切掉低单位部分，然后再计算结果。

示例：

```
若start = 2005-12-31 23:59:59，end = 2006-01-01 00:00:00:
datediff(end, start, 'dd') = 1
datediff(end, start, 'mm') = 1
datediff(end, start, 'yyyy') = 1
datediff(end, start, 'hh') = 1
datediff(end, start, 'mi') = 1
datediff(end, start, 'ss') = 1

datediff(2013-05-31 13:00:00, 2013-05-31 12:30:00, 'ss') = 1800
datediff(2013-05-31 13:00:00, 2013-05-31 12:30:00, 'mi') = 30
```

DATEPART

函数声明：

```
bigint datepart(datetime date, string datepart)
```

用途：提取日期date中指定的时间单位datepart的值。

参数说明：

- date：Datetime类型，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。
- datepart：String类型常量。支持扩展的日期格式。若datepart不符合指定格式或者其它类型则会发生异常。

返回值：Bigint类型。若任一输入参数为NULL，返回NULL。

示例：

```
datepart('2013-06-08 01:10:00', 'yyyy') = 2013
datepart('2013-06-08 01:10:00', 'mm') = 6
```

DATETRUNC

函数声明：

```
datetime datetrunc (datetime date, string datepart)
```

用途：返回日期date被截取指定时间单位datepart后的日期值。

参数说明：

- date：Datetime类型，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。
- datepart：String类型常量。支持扩展的日期格式。若datepartt不符合指定格式或者其它类型则会发生异常。

返回值：Datetime类型。任意一个参数为NULL的时候返回NULL。

示例：

```
datetrunc(2011-12-07 16:28:46, 'yyyy') = 2011-01-01 00:00:00
datetrunc(2011-12-07 16:28:46, 'month') = 2011-12-01 00:00:00
datetrunc(2011-12-07 16:28:46, 'DD') = 2011-12-07 00:00:00
```

FROM_UNIXTIME

函数声明：

```
datetime from_unixtime(bigint unixtime)
```

用途：将数字型的unix时间日期值unixtime转为日期值。

参数说明：

- unixtime：Bigint类型，秒数，unix格式的日期时间值，若输入为string，double类型会隐式转换为bigint后参与运算。

返回值：Datetime类型的日期值，unixtime为NULL时返回NULL。

示例：

```
from_unixtime(123456789) = 2009-01-20 21:06:29
```

GETDATE

函数声明：

```
datetime getdate()
```

用途：获取当前系统时间。使用东八区时间作为MaxCompute标准时间。

返回值：返回当前日期和时间，datetime类型。

备注：

- 在一个MaxCompute SQL任务中(以分布式方式执行)，getdate总是返回一个固定的值。返回结果会是MaxCompute SQL执行期间的任意时间，时间精度精确到秒。

ISDATE

函数声明：

```
boolean isdate(string date, string format)
```

用途：判断一个日期字符串能否根据对应的格式串转换为一个日期值，如果转换成功返回TRUE，否则返回FALSE。

参数说明：

- date：String格式的日期值，若输入为bigint，double或者datetime类型会隐式转换为string类型后参与运算，其它类型报异常。
- format：String类型常量，不支持日期扩展格式。其它类型或不支持的格式会抛异常。如果format中出现多余的格式串，则只取第一个格式串对应的日期数值，其余的会被视为分隔符。如isdate(“1234-yyyy ”, “yyyy-yyyy ”)，会返回TRUE。

返回值：Boolean类型，如任意参数为NULL，返回NULL。

LASTDAY

函数声明：

```
datetime lastday(datetime date)
```

用途：取date当月的最后一天，截取到天，时分秒部分为00:00:00。

参数说明：

- date：Datetime类型，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型报异常。

返回值：Datetime类型，如输入为NULL，返回NULL

TO_DATE

函数声明：

```
datetime to_date(string date, string format)
```

用途：将一个字符串date按照format指定的格式转成日期值。

参数说明：

- date：String类型，要转换的字符串格式的日期值，若输入为bigint，double或者datetime类型会隐式转换为String类型后参与运算，为其它类型抛异常，为空串时抛异常。
- format：String类型常量，日期格式。非常量或其他类型会引发异常。format不支持日期扩展格式，其他字符作为无用字符在解析时忽略。format参数至少包含”yyyy”，否则引发异常，如果format中出现多余的格式串，则只取第一个格式串对应的日期数值，其余的会被视为分隔符。如to_date(“1234-2234 “, “yyyy-yyyy “)会返回1234-01-01 00:00:00。

返回值：Datetime类型。若任一输入为NULL，返回NULL值。

示例：

```
    to_date('阿里巴巴2010-12*03', '阿里巴巴yyyy-mm*dd') = 2010-12-03 00:00:00
to_date('20080718', 'yyyymmdd') = 2008-07-18 00:00:00

to_date('2008718', 'yyyymmdd')
-- 格式不符合，引发异常
to_date('阿里巴巴2010-12*3', '阿里巴巴yyyy-mm*dd')
-- 格式不符合，引发异常
to_date('2010-24-01', 'yyyy')
-- 格式不符合，引发异常
```

TO_CHAR

函数声明：

```
string to_char(datetime date, string format)
```

用途：将日期类型date按照format指定的格式转成字符串

参数类型：

- date：Datetime类型，要转换的日期值，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。
- format：String类型常量。非常量或其他类型会引发异常。format中的日期格式部分会被替换成相应的数据，其它字符直接输出。

返回值：String类型。任一输入参数为NULL，返回NULL。

示例：

```
to_char('2010-12-03 00:00:00', '阿里金融yyyy-mm*dd') = '阿里金融2010-12*03'
to_char('2008-07-18 00:00:00', 'yyyymmdd') = '20080718'
to_char('阿里巴巴2010-12*3', '阿里巴巴yyyy-mm*dd') -- 引发异常
to_char('2010-24-01', 'yyyy') -- 会引发异常
to_char('2008718', 'yyyymmdd') -- 会引发异常
```

备注:

- 关于其他类型向string类型转换请参考 字符串函数 TO_CHAR 。

UNIX_TIMESTAMP

函数声明：

```
bigint unix_timestamp(datetime date)
```

用途：将日期date转化为整型的unix格式的日期时间值。

参数说明：

- date：Datetime类型日期值，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。

返回值：Bigint类型，表示unix格式日期值，date为NULL时返回NULL。

WEEKDAY

函数声明：

```
bigint weekday (datetime date)
```

用途：返回date日期当前周的第几天。

参数说明：

- date：Datetime类型，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。

返回值：Bigint类型，若输入参数为NULL，返回NULL。周一作为一周的第一天，返回值为0。其他日期依次递增，周日返回6。

WEEKOFYEAR

函数声明：

```
bigint weekofyear(datetime date)
```

用途：返回日期date位于那一年的第几周。周一作为一周的第一天。需要注意的是，关于这一周算上一年， 还是下一年，主要是看这一周大多数日期（4天以上）在哪一年多。 算在前一年，就是前一年的最后一周。算在后一年就是后一年的第一周。

参数说明:

- date：Datetime类型日期值，若输入为string类型会隐式转换为datetime类型后参与运算，其它类型抛异常。

返回值：Bigint类型。若输入为NULL，返回NULL。

示例说明：

```
select weekofyear(to_date("20141229", "yyyymmdd")) from dual;
返回结果：
+------------+
| _c0 |
+------------+
| 1 |
+------------+
-虽然20141229属于2014年，但是这一周的大多数日期是在2015年，因此返回结果为1，表示是2015年的第一周。

select weekofyear(to_date("20141231", "yyyymmdd")) from dual；--返回结果为1。
select weekofyear(to_date("20151229", "yyyymmdd")) from dual；--返回结果为53。
```

窗口函数

MaxCompute SQL中可以使用窗口函数进行灵活的分析处理工作，窗口函数只能出现在select子句中，窗口函数中不要嵌套使用窗口函数和聚合函数，窗口函数不可以和同级别的聚合函数一起使用。目前在一个MaxCompute SQL语句中，可以使用至多5个窗口函数。

窗口函数的语法

```
    window_func() over (partition by col1, [col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] windowing_clause)
```

- partition by部分用来指定开窗的列。分区列的值相同的行被视为在同一个窗口内。现阶段，同一窗口内最多包含1亿行数据，否则运行时报错。
- order by用来指定数据在一个窗口内如何排序
- windowing_clause部分可以用rows指定开窗方式，有两种方式： **rows between x**

preceding|following and y preceding|following表示窗口范围是从前或后x行到前或后y行。 rows x preceding|following窗口范围是从前或后第x行到当前行。** x，y必须为大于等于0的整数常量，限定范围0 ~ 10000，值为0时表示当前行。必须指定order by才可以用rows方式指定窗口范围。

备注:

- 并非所有的窗口函数都可以用rows指定开窗方式，支持这种用法的窗口函数有avg、count、max、min、stddev和sum。

COUNT

函数声明：

```
bigint count([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
```

用途：计算计数值。

参数说明：

- expr：任意类型，当值为NULL时，该行不参与计算。当指定distinct关键字时表示取唯一值的计数值。
- partition by col1[, col2...]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内expr的计数值，指定order by时返回结果以指定的顺序排序，并且值为当前窗口内从开始行到当前行的累计计数值。

返回值：Bigint类型。

备注:

- 当指定distinct关键字时不能写order by。

示例：假设存在表test_src，表中存在bigint类型的列user_id，

```
select user_id,
count(user_id) over (partition by user_id) as count
from test_src;

+---------+------------+
| user_id | count |
+---------+------------+
| 1 | 3 |
| 1 | 3 |
| 1 | 3 |
| 2 | 1 |
| 3 | 1 |
```

```
+---------+------------+

-- 不指定order by时，返回当前窗口内user_id的计数值

select user_id,
count(user_id) over (partition by user_id order by user_id) as count
from test_src;

+---------+------------+
| user_id | count |
+---------+------------+
| 1 | 1 | -- 窗口起始
| 1 | 2 | -- 到当前行共计两条记录，返回2
| 1 | 3 |
| 2 | 1 |
| 3 | 1 |
+---------+------------+

-- 指定order by时，返回当前窗口内从开始行到当前行的累计计数值。
```

AVG

函数声明：

```
    avg([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc] [, col2[asc|desc]...]] [windowing_clause])
```

用途：计算平均值。

参数说明：

- distinct：当指定distinct关键字时表示取唯一值的平均值。
- expr：Double类型，若输入为string，bigint会隐式转换到double类型后参与运算，其它类型抛异常。当值为NULL时，该行不参与计算。Boolean类型不允许参与计算。
- partition by col1[, col2]...：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时返回当前窗口内所有值的平均值，指定order by时返回结果以指定的方式排序，并且返回窗口内从开始行到当前行的累计平均值。

返回值：Double类型。

备注:

- 指明distinct关键字时不能写order by。

MAX

函数声明：

```
max([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
```

用途：计算最大值。

参数说明：

- expr：除Boolean以外的任意类型，当值为NULL时，该行不参与计算。当指定distinct关键字时表示取唯一值的最大值(指定该参数与否对结果没有影响)。
- partition by col1[, col2...]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内的最大值。指定order by时，返回结果以指定的方式排序，并且值为当前窗口内从开始行到当前行的最大值。

返回值：同expr类型。

备注：

- 当指定distinct关键字时不能写order by。

MIN

函数声明：

```
min([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
```

用途：计算最小值。

参数说明：

- expr：除boolean以外的任意类型，当值为NULL时，该行不参与计算。当指定distinct关键字时表示取唯一值的最小值(指定该参数与否对结果没有影响)。
- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内的最小值。指定order by时，返回结果以指定的方式排序，并且值为当前窗口内从开始行到当前行的最小值。

返回值：同expr类型。

备注：

- 当指定distinct关键字时不能写order by。

MEDIAN

函数声明：

```
double median(double number1,number2...) over(partition by col1[, col2...])

decimal median(decimal number1,number2...) over(partition by col1[,col2...])
```

用途：计算中位数。

参数说明：

number1,number1...：Double类型或decimal类型的1到255个数字。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为null时忽略。如果传入的参数是一个Double类型的数，会默认转成一个double 的array。

partition by col1[, col2...]：指定开窗口的列。

返回值：Double类型。

STDDEV

函数声明：

```
double stddev([distinct] expr) over(partition by col1[, col2...]

[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])

decimal stddev([distinct] expr) over(partition by col1[,col2...] [order by col1 [asc|desc][, col2[asc|desc]...]]
[windowing_clause])
```

用途：总体标准差。

参数说明：

expr：Double类型或decimal类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略该行。当指定distinct关键字时表示计算唯一值的总体标准差。

partition by col1[, col2..]：指定开窗口的列。

order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内的总体标准差。指定order by时，返回结果以指定的方式排序，并且值为当前窗口内从开始行到当前行的总体标准差。

返回值：输入为Decimal类型时返回Decimal类型，否则返回Double类型。

示例：

```
select window, seq, stddev_pop('1\01') over (partition by window order by seq) from dual;
```

备注：

- 当指定distinct关键字时不能写order by。
- stddev还有一个别名函数stddev_pop，用法跟stddev一样。

STDDEV_SAMP

函数声明：

```
double stddev_samp([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
```

用途：样本标准差。

参数说明：

- expr：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略该行。当指定distinct关键字时表示计算唯一值的样本标准差。
- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内的样本标准差。指定order by时，返回结果以指定的方式排序，并且值为当前窗口内从开始行到当前行的样本标准差。

返回值：Double类型。

备注：

- 当指定distinct关键字时不能写order by。

SUM

函数声明：

```
sum([distinct] expr) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]] [windowing_clause])
```

用途：计算汇总值。

参数说明：

- expr：Double类型或Bigint类型，当输入为string时隐式转换为double参与运算，其它类型报异常。当值为NULL时，该行不参与计算。指定distinct关键字时表示计算唯一值的汇总值。
- partition by col1[, col2..]：指定开窗口的列。

- order by col1 [asc|desc], col2[asc|desc]：不指定order by时，返回当前窗口内expr的汇总值。指定order by时，返回结果以指定的方式排序，并且返回当前窗口从首行至当前行的累计汇总值。返回值：输入参数是bigint返回bigint， 输入参数为double或string时，返回double类型。

返回值：输入参数是bigint返回bigint，输入参数为double或string时，返回double类型。

备注：

- 当指定distinct时不能用order by。

DENSE_RANK

命令格式：

```
bigint dense_rank() over(partition by col1[, col2...]
order by col1 [asc|desc][, col2[asc|desc]...])
```

用途：计算连续排名。col2相同的行数据获得的排名相同。

参数说明：

- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定排名依据的值。

返回值：Bigint类型。

RANK

命令格式：

```
bigint rank() over(partition by col1[, col2...]
order by col1 [asc|desc][, col2[asc|desc]...])
```

用途：计算排名。col2相同的行数据获得排名顺序下降。

参数说明：

- partition by col2[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定排名依据的值。

返回值：Bigint类型。

LAG

函数声明：

```
lag(expr，bigint offset, default) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]])
```

用途：按偏移量取当前行之前第几行的值，如当前行号为rn，则取行号为rn-offset的值。

参数说明：

- expr：任意类型。
- offset：Bigint类型常量，输入为string，double到bigint的隐式转换，offset > 0。
- default：当offset指定的范围越界时的缺省值，常量，默认值为NULL。
- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定返回结果的排序方式。

返回值：同expr类型。

LEAD

函数声明：

```
lead(expr, bigint offset, default) over(partition by col1[, col2...]
[order by col1 [asc|desc][, col2[asc|desc]...]])
```

用途：按偏移量取当前行之后第几行的值，如当前行号为rn则取行号为rn+offset的值。

参数说明：

expr：任意类型。

offset：可选，Bigint类型常量，输入为string，decimal，double到bigint的隐式转换，offset > 0。

default：可选，当offset指定的范围越界时的缺省值，常量。

partition by col1[, col2..]：指定开窗口的列。

order by col1 [asc|desc], col2[asc|desc]：指定返回结果的排序方式。

返回值：同expr类型。

示例：

```
select c_double_a,c_string_b,c_int_a,lead(c_int_a,1) over(partition by c_double_a order by c_string_b) from dual;

select c_string_a,c_time_b,c_double_a,lead(c_double_a,1) over(partition by c_string_a order by c_time_b) from dual;

select c_string_in_fact_num,c_string_a,c_int_a,lead(c_int_a) over(partition by c_string_in_fact_num order by c_string_a)
from dual;
```

PERCENT_RANK

函数声明：

```
    percent_rank() over(partition by col1[, col2...]
order by col1 [asc|desc][, col2[asc|desc]...])
```

用途：计算一组数据中某行的相对排名。

参数说明：

- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定排名依据的值。

返回值：Double类型，值域为[0, 1]，相对排名的计算方式为为：(rank-1)/(number of rows -1)。

备注:

- 目前限制单个窗口内的行数不超过10,000,000条。

ROW_NUMBER

函数声明：

```
    row_number() over(partition by col1[, col2...]
order by col1 [asc|desc][, col2[asc|desc]...])
```

用途：计算行号，从1开始。

参数说明：

- partition by col1[, col2..]：指定开窗口的列。
- order by col1 [asc|desc], col2[asc|desc]：指定结果返回时的排序的值。

返回值：Bigint类型。

CLUSTER_SAMPLE

函数声明：

```
    boolean cluster_sample(bigint x[, bigint y])
over(partition by col1[, col2..])
```

用途：分组抽样。

参数说明：

- x：Bigint类型常量，x>=1。若指定参数y，x表示将一个窗口分为x份；否则，x表示在一个窗口中抽取x行记录(即有x行返回值为true)。x为NULL时，返回值为NULL。
- y：Bigint类型常量，y>=1，y<=x。表示从一个窗口分的x份中抽取y份记录(即y份记录返回值为true)。y为NULL时，返回值为NULL。
- partition by col1[, col2]：指定开窗口的列。

返回值：Boolean类型。

示例，如表test_tbl中有key，value两列，key为分组字段，值有groupa，groupb两组，value为值，如下

```
    +------------+--------------------+
| key | value |
+------------+--------------------+
| groupa | -1.34764165478145 |
| groupa | 0.740212609046718 |
| groupa | 0.167537127858695 |
| groupa | 0.630314566185241 |
| groupa | 0.0112401388646925 |
| groupa | 0.199165745875297 |
| groupa | -0.320543343353587 |
| groupa | -0.273930924365012 |
| groupa | 0.386177958942063 |
| groupa | -1.09209976687047 |
| groupb | -1.10847690938643 |
| groupb | -0.725703978381499 |
| groupb | 1.05064697475759 |
| groupb | 0.135751224393789 |
| groupb | 2.13313102040396 |
| groupb | -1.11828960785008 |
| groupb | -0.849235511508911 |
| groupb | 1.27913806620453 |
| groupb | -0.330817716670401 |
| groupb | -0.300156896191195 |
| groupb | 2.4704244205196 |
| groupb | -1.28051882084434 |
+------------+--------------------+
```

想要从每组中抽取约10%的值，可以用以下MaxCompute SQL完成：

```
    select key, value
from (
select key, value, cluster_sample(10, 1) over(partition by key) as flag
from tbl
) sub
```

```
where flag = true;

+--------+--------------------+
| key | value |
+--------+--------------------+
| groupa | -1.34764165478145 |
| groupb | -0.725703978381499 |
| groupb | 2.4704244205196 |
+-----+----------------------+
```

聚合函数

聚合函数，其输入与输出是多对一的关系，即将多条输入记录聚合成一条输出值。可以与SQL中的group by语句联用。

COUNT

函数声明：

```
    bigint count([distinct|all] value)
```

用途：计算记录数。

参数说明：

- distinct|all：指明在计数时是否去除重复记录，默认是all，即计算全部记录，如果指定distinct，则可以只计算唯一值数量。
- value：可以为任意类型，当value值为NULL时，该行不参与计算，value可以为，*当count()*时，返回所有行数。

返回值：Bigint类型。

示例：

```
    -- 如表tbla有列col1类型为bigint
+------+
| COL1 |
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+

select count(*) from tbla; -- 值为3,
select count(col1) from tbla; -- 值为2
```

聚合函数可以和group by一同使用，例如：假设存在表test_src，存在如下两列：key string类型，value double类型，

```
-- test_src的数据为

+-----+-------+
| key | value |
+-----+-------+
| a | 2.0 |
+-----+-------+
| a | 4.0 |
+-----+-------+
| b | 1.0 |
+-----+-------+
| b | 3.0 |
+-----+-------+

-- 此时执行如下语句，结果为：

select key, count(value) as count from test_src group by key;


+-----+-------+
| key | count |
+-----+-------+
| a | 2 |
+-----+-------+
| b | 2 |
+-----+-------+

-- 聚合函数将对相同key值得value值做聚合计算。下面介绍的其他聚合函数使用方法均与此例相同，不一一举例。
```

AVG

函数声明：

```
double avg(double value)
```

用途：计算平均值。

参数说明：

- value：Double类型，若输入为string或bigint会隐式转换到double类型后参与运算，其它类型抛异常。当value值为NULL时，该行不参与计算。Boolean类型不允许参与计算。

返回值：Double类型。

示例：

```
-- 如表tbla有一列value，类型为bigint
```

```
+--------+
| value |
+--------+
| 1 |
| 2 |
| NULL |
+--------+

-- 则对该列计算avg结果为(1+2)/2=1.5

select avg(value) as avg from tbla;

+------+
| avg |
+------+
| 1.5 |
+------+
```

MAX

函数声明：

```
    max(value)
```

用途：计算最大值。

参数说明：

- value：可以为任意类型，当列中的值为NULL时，该行不参与计算。Boolean类型不允许参与运算。

返回值：与value类型相同。

示例：

```
    -- 如表tbla有一列col1，类型为bigint
+------+
| col1 |
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+

select max(value) from tbla; -- 返回值为2
```

MIN

命令格式：

```
MIN(value)
```

用途：计算最小值。

参数说明：

- value：可以为任意类型，当列中的值为NULL时，该行不参与计算。Boolean类型不允许参与计算。

示例：

```
-- 如表tbla有一列value，类型为bigint
+------+
| value|
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+
select min(value) from tbla; -- 返回值为1
```

MEDIAN

函数声明：

```
double median(double number)
```

用途：计算中位数。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略。

返回值：Double类型。

STDDEV

函数声明：

```
double stddev(double number)
```

用途：计算总体标准差。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略。

返回值：Double类型。

STDDEV_SAMP

命令格式：

```
double stddev_samp(double number)
```

用途：计算样本标准差。

参数说明：

- number：Double类型。若输入为string类型或bigint类型会隐式转换到double类型后参与运算，其他类型抛异常。当输入值为NULL时忽略。

返回值：Double类型。

SUM

函数声明：

```
sum(value)
```

用途：计算汇总值。

参数说明：

- value：Double或bigint类型，若输入为string会隐式转换到double类型后参与运算，当列中的值为NULL时，该行不参与计算。Boolean类型不允许参与计算。

返回值：输入为bigint时返回bigint，输入为double或string时返回double类型。

示例：

```
-- 如表tbla有一列value，类型为bigint
+------+
| value|
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+
```

```
select sum(value) from tbla; -- 返回值为3
```

WM_CONCAT

函数声明：

```
string wm_concat(string separator, string str)
```

用途：用指定的spearator做分隔符，链接str中的值。

参数说明：

- separator：String类型常量，分隔符。其他类型或非常量将引发异常。
- str：String类型，若输入为bigint，double或者datetime类型会隐式转换为string后参与运算，其它类型报异常。

返回值：String类型。

备注:

- 对语句”select wm_concat(‘,’ , name) from test_src;” ，若test_src为空集合，这MaxCompute SQL条语句返回NULL值。聚合函数，其输入与输出是多对一的关系，即将多条输入记录聚合成一条输出值。可以与SQL中的group by语句联用。

其他函数

CAST

函数声明：

```
cast(expr as <type>)
```

用途：将表达式的结果转换成目标类型，如cast(‘1’ as bigint)将字符串”1”转为整数类型的1，如果转换不成功或不支持的类型转换会引发异常。

备注:

- cast(double as bigint)，将double值转换成bigint。
- cast(string as bigint) 在将字符串转为bigint时，如果字符串中是以整型表达的数字，会直接转为bigint类型。如果字符串中是以浮点数或指数形式表达的数字，则会先转为double类型，再转为bigint类型。
- cast(string as datetime) 或 cast(datetime as string)时，会采用默认的日期格式yyyy-mm-dd

hh:mi:ss。

COALESCE

函数声明：

```
coalesce(expr1, expr2, ...)
```

用途：返回列表中第一个非NULL的值，如果列表中所有的值都是NULL则返回NULL。

参数说明：

- expri是要测试的值。所有这些值类型必须相同或为NULL，否则会引发异常。

返回值：返回值类型和参数类型相同。

备注:

- 参数至少要有一个，否则引发异常。

DECODE

函数声明：

```
decode(expression, search, result[, search, result]...[, default])
```

用途：实现if-then-else分支选择的功能。

参数说明：

- expression：要比较的表达式。
- search：和expression进行比较的搜索项。
- result：search和expression的值匹配时的返回值。
- default：可选项，如果所有的搜索项都不匹配，则返回此default值，如果未指定，则会返回NULL。

返回值：返回匹配的search；如果没有匹配，返回default；如果没有指定default，返回NULL。

备注:

- 至少要指定三个参数。
- 所有的result类型必须一致，或为NULL。不一致的数据类型会引发异常。所有的search和expression类型必须一致，否则报异常。
- 如果decode中的search选项有重复时且匹配时，会返回第一个值。

示例：

```
    select
decode(customer_id,
1, 'Taobao',
2, 'Alipay',
3, 'Aliyun',
NULL, 'N/A',
'Others') as result
from sale_detail;
```

上面的decode函数实现了下面if-then-else语句中的功能：

```
    if customer_id = 1 then
result := 'Taobao';
elsif customer_id = 2 then
result := 'Alipay';
elsif customer_id = 3 then
result := 'Aliyun';
...
else
result := 'Others';
end if;
```

但需要用户注意的是，通常情况下MaxCompute SQL在计算NULL = NULL时返回NULL，但在decode函数中，NULL与NULL的值是相等的。在上述事例中，当customer_id的值为NULL时，decode函数返回”N/A”。

GREATEST

函数声明：

```
    greatest(var1, var2, ...)
```

用途：返回输入参数中最大的一个。

参数说明：

- var1，var2可以为bigint，double，datetime或者string。若所有值都为NULL则返回NULL。

返回值:

- 输入参数中的最大值，当不存在隐式转换时返回同输入参数类型。
- NULL为最小值。
- 当输入参数类型不同时，double，bigint，string之间的比较转为double；string，datetime的比较转为datetime。不允许其它的隐式转换。

ORDINAL

函数声明：

```
ordinal(bigint nth, var1, var2, ...)
```

用途：将输入变量按从小到大排序后，返回nth指定的位置的值。

参数说明：

- nth：Bigint类型，指定要返回的位置，为NULL时返回NULL。
- var1，var2：类型可以为bigint，double，datetime或者string。

返回值：

- 排在第nth位的值，当不存在隐式转换时返回同输入参数类型。
- 有类型转换时，double，bigint，string之间的转换返回double。string，datetime之间的转换返回datetime。不允许其它的隐式转换。
- NULL为最小。

示例：

```
ordinal(3, 1, 3, 2, 5, 2, 4, 6) = 2
```

LEAST

函数声明：

```
least(var1, var2, ...)
```

用途：返回输入参数中最小的一个。

参数说明：

- var1，var2可以为bigint，double，datetime或者string。若所有值都为NULL则返回NULL。

返回值：

- 输入参数中的最小值，当不存在隐式转换时返回同输入参数类型。
- NULL为最小。
- 有类型转换时，double，bigint，string之间的转换返回double。string，datetime之间的转换返回datetime。不允许其它的隐式类型转换。

UUID

命令格式：

```
```

```
string uuid()
```

用途：返回一个随机ID，形式示例：”29347a88-1e57-41ae-bb68-a9edbdd94212”。

SAMPLE

函数声明：

```
boolean sample(x, y, column_name)
```

用途：对所有读入的column_name的值，sample根据x，y的设置做采样，并过滤掉不满足采样条件的行。

参数说明：

- x，y：Bigint类型，表示哈希为x份，取第y份。y可省略，省略时取第一份，如果省略参数中的y，则必须同时省略column_name。x，y为整型常量，大于0，其它类型或小于等于0时抛异常，若y>x也抛异常。x，y任一输入为NULL时返回NULL。
- column_name是采样的目标列。column_name可以省略，省略时根据x，y的值随机采样。任意类型，列的值可以为NULL。不做隐式类型转换。如果column_name为常量NULL会报异常。

返回值：Boolean类型。

备注:

- 为了避免NULL值带来的数据倾斜，因此对于column_name中为NULL的值，会在x份中进行均匀哈希。如果不加column_name，则数据量比较少时输出不一定均匀，在这种情况下建议加上column_name，以获得比较好的输出结果。

示例：假定存在表tbla，表内有列名为cola的列，

```
select * from tbla where sample (4, 1 , cola) = true;
-- 表示数值会根据cola hash为4份，取第1份
select * from tbla where sample (4, 2) = true;
-- 表示数值会对每行数据做随机哈希分配为4份，取第2份
```

CASE WHEN表达式

MaxCompute提供两种case when语法格式，如下所述：

```
case value
when (_condition1) then result1
when (_condition2) then result2
...
else resultn
end
```

```
case
when (_condition1) then result1
when (_condition2) then result2
when (_condition3) then result3
...
else resultn
end
```

case when表达式可以根据表达式value的计算结果灵活返回不同的值， 如以下语句根据shop_name的不同情况得出所属区域

```
    select
case
when shop_name is null then 'default_region'
when shop_name like 'hang%' then 'zj_region'
end as region
from sale_detail;
```

说明：

- 如果result类型只有bigint，double，统一转double再返回；
- 如果result类型中有string类型，统一转string再返回，如果不能转则报错(如boolean型)；
- 除此之外不允许其它类型之间的转换；

IF

函数声明：

```
if(testCondition, valueTrue, valueFalseOrNull)
```

用途：判断testCondition是否为真，如果为真,返回valueTrue，如果不满足则返回另一个值(valueFalse或者Null)。

参数说明：

- testCondition:要判断的表达式， boolean类型；
- valueTrue: 表达式testCondition为True的时候，返回的值。
- valueFalseOrNull：不满足表达式testCondition时2，返回的值，可以设为Null.返回值：返回值类型和参数valueTrue或者valueFalseOrNul的类型一致。示例：

```
select if(1=2,100,200) from dual;
返回值：
+------------+
| _c0 |
+------------+
| 200 |
+------------+
```

UDF

UDF全称User Defined Function，即用户自定义函数。ODPS提供了很多内建函数来满足用户的计算需求，同时用户还可以通过创建自定义函数来满足 不同的计算需求。UDF在使用上与普通的 内建函数 类似。

使用Maven的用户可以从Maven库中搜索"odps-sdk-udf"获取不同版本的Java SDK，相关配置信息：

```
<dependency>
  <groupId>com.aliyun.odps</groupId>
  <artifactId>odps-sdk-udf</artifactId>
  <version>0.20.7-public</version>
</dependency>
```

在ODPS中，用户可以扩展的UDF有两种：

UDF 分类	描述
User Defined Scalar Function，通常也称之为UDF	用户自定义标量值函数(User Defined Scalar Function)通常也称之为UDF。其输入与输出是一对一的关系，即读入一行数据，写出一条输出值。
UDTF(User Defined Table Valued Function)	自定义表值函数，是用来解决一次函数调用输出多行数据场景的，也是唯一能返回多个字段的自定义函数。而UDF只能一次计算输出一条返回值。
UDAF(User Defined Aggregation Function)	自定义聚合函数，其输入与输出是多对一的关系，即将多条输入记录聚合成一条输出值。可以与SQL中的Group By语句联用。具体语法请参考 聚合函数 。

备注:

- UDF广义的说法代表了自定义标量函数，自定义聚合函数及自定义表函数三种类型的自定义函数的集合。狭义来说，仅代表用户自定义标量函数。 文档会经常使用这一名词，请读者根据文档上下文判断具体含义。

参数与返回值类型

UDF支持ODPS SQL的数据类型有：bigint, string, double, boolean以及datetime类型。ODPS数据类型与Java类型的对应关系如下：

ODPS SQL Type	Bigint	String	Double	Boolean	Decimal
Java Type	Long	String	Double	Boolean	Decimal

备注:

- SQL中的NULL值通过Java中的NULL引用表示，因此Java primitive type是不允许使用的，因为无法表示SQL中的NULL值。

UDF示例

请参考快速入门分册UDF示例。

UDF

实现UDF需要继承com.aliyun.odps.udf.UDF类，并实现evaluate方法。evaluate方法必须是非static的public方法。Evaluate方法的参数和返回值类型将作为SQL中UDF的函数签名。这意味着用户可以在UDF中实现多个evaluate方法，在调用UDF时，框架会依据UDF调用的参数类型匹配正确的evaluate方法。

下面是一个UDF的例子。

```
    package org.alidata.odps.udf.examples;
import com.aliyun.odps.udf.UDF;

public final class Lower extends UDF {
public String evaluate(String s) {
if (s == null) { return null; }
return s.toLowerCase();
}
}
```

可以通过实现void setup(ExecutionContext ctx)和void close()来分别实现UDF的初始化和结束代码。

UDF的使用方式于ODPS SQL中普通的内建函数相同，详情请参考内建函数。

UDAF

实现Java UDAF类需要继承com.aliyun.odps.udf.Aggregator，并实现如下几个接口：

```
public abstract class Aggregator implements ContextFunction {

@Override
public void setup(ExecutionContext ctx) throws UDFException {
}

@Override
```

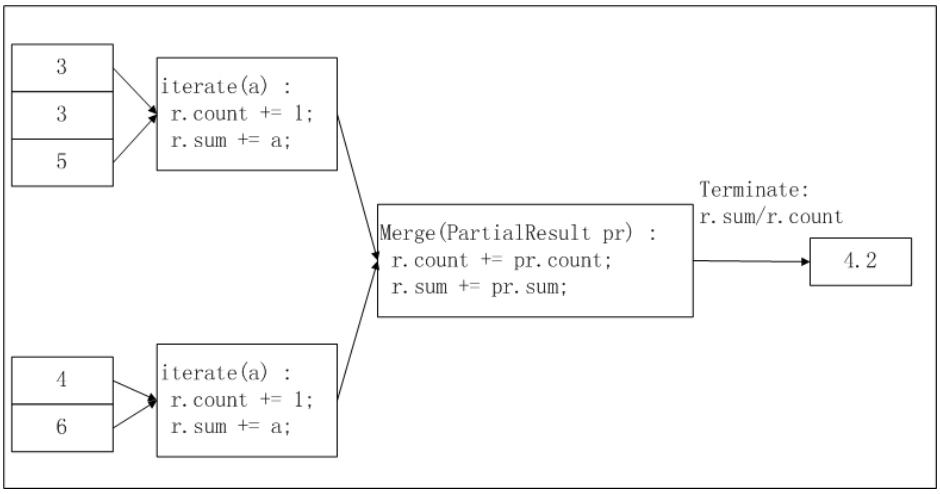
```
public void close() throws UDFException {
}

/**
* 创建聚合Buffer
* @return Writable 聚合buffer
*/
abstract public Writable newBuffer();
/**
* @param buffer 聚合buffer
* @param args SQL中调用UDAF时指定的参数
* @throws UDFException
*/
abstract public void iterate(Writable buffer, Writable[] args) throws UDFException;

/**
* 生成最终结果
* @param buffer
* @return Object UDAF的最终结果
* @throws UDFException
*/
abstract public Writable terminate(Writable buffer) throws UDFException;

abstract public void merge(Writable buffer, Writable partial) throws UDFException;
}
```

其中最重要的是iterate，merge和terminate三个接口，UDAF的主要逻辑依赖于这三个接口的实现。此外，还需要用户实现自定义的Writable buffer。以实现求平均值avg为例，下图简要说明了在ODPS UDAF中这一函数的实现逻辑及计算流程：



在上图中，输入数据被按照一定的大小进行分片(有关分片的描述可参考 MapReduce)，每片的大小适合一个worker在适当的时间内完成。这个分片大小的设置需要用户手动配置完成。UDAF的计算过程分为两阶段：

- 在第一阶段，每个worker统计分片内数据的个数及汇总值，我们可以将每个分片内的数据个数及汇总值视为一个中间结果；
- 在第二阶段，worker汇总上一个阶段中每个分片内的信息。在最终输出时，r.sum / r.count即是所有输入数据的平均值；

下面是一个计算平均值的UDAF的代码示例:

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import com.aliyun.odps.io.DoubleWritable;
import com.aliyun.odps.io.Writable;
import com.aliyun.odps.udf.Aggregator;
import com.aliyun.odps.udf.UDFException;
import com.aliyun.odps.udf.annotation.Resolve;

@Resolve({"double->double"})
public class AggrAvg extends Aggregator {

private static class AvgBuffer implements Writable {

private double sum = 0;
private long count = 0;

@Override
public void write(DataOutput out) throws IOException {
out.writeDouble(sum);
out.writeLong(count);
}
@Override
public void readFields(DataInput in) throws IOException {
sum = in.readDouble();
count = in.readLong();
}
}

private DoubleWritable ret = new DoubleWritable();

@Override
public Writable newBuffer() {
return new AvgBuffer();
}

@Override
public void iterate(Writable buffer, Writable[] args) throws UDFException {
DoubleWritable arg = (DoubleWritable) args[0];
AvgBuffer buf = (AvgBuffer) buffer;
if (arg != null) {
buf.count += 1;
buf.sum += arg.get();
}
}

@Override
public Writable terminate(Writable buffer) throws UDFException {
AvgBuffer buf = (AvgBuffer) buffer;
if (buf.count == 0) {
ret.set(0);
```

```
} else {
ret.set(buf.sum / buf.count);
}
return ret;
}

@Override
public void merge(Writable buffer, Writable partial) throws UDFException {
AvgBuffer buf = (AvgBuffer) buffer;
AvgBuffer p = (AvgBuffer) partial;
buf.sum += p.sum;
buf.count += p.count;
}

}
```

注意：

- UDAF在SQL中的使用语法与普通的内建聚合函数相同，详情请参考 聚合函数 。
- 关于如何运行UDTF,方法与UDF类似，具体步骤请参考运行UDF。

UDTF

Java UDTF需要继承com.aliyun.odps.udf.UDTF类。这个类需要实现4个接口。

接口定义	描述
public void setup(ExecutionContext ctx) throws UDFException	初始化方法，在UDTF处理输入数据前，调用用户自定义的初始化行为。在每个Worker内setup会被先调用一次。
public void process(Object[] args) throws UDFException	这个方法由框架调用，SQL中每一条记录都会对应调用一次process，process的参数为SQL语句中指定的UDTF输入参数。输入参数以Object[]的形式传入，输出结果通过forward函数输出。用户需要在process函数内自行调用forward，以决定输出数据。
public void close() throws UDFException	UDTF的结束方法，此方法由框架调用，并且只会被调用一次，即在处理完最后一条记录之后。
public void forward(Object ...o) throws UDFException	用户调用forward方法输出数据，每次forward代表输出一条记录。对应SQL语句UDTF的as子句指定的列。

下面将给出一个UDTF程序示例：

```
package org.alidata.odps.udtf.examples;

import com.aliyun.odps.udf.UDTF;
import com.aliyun.odps.udf.UDTFCollector;
```

```
import com.aliyun.odps.udf.annotation.Resolve;
import com.aliyun.odps.udf.UDFException;

// TODO define input and output types, e.g., "string,string->string,bigint".
@Resolve({"string,bigint->string,bigint"})
public class MyUDTF extends UDTF {

@Override
public void process(Object[] args) throws UDFException {
String a = (String) args[0];
Long b = (Long) args[1];

for (String t: a.split("\\s+")) {
forward(t, b);
}
}
}
```

注：

- 以上只是程序示例，关于如何在ODPS中运行UDTF,方法与UDF类似，具体实现步骤请参考运行UDF。

在SQL中可以这样使用这个UDTF，假设在ODPS上创建UDTF时注册函数名为user_udtf：

```
select user_udtf(col0, col1) as (c0, c1) from my_table;
```

假设my_table的col0， col1的值为：

```
+------+------+
| col0 | col1 |
+------+------+
| A B | 1 |
| C D | 2 |
+------+------+
```

则select出的结果为：

```
+----+----+
| c0 | c1 |
+----+----+
| A | 1 |
| B | 1 |
| C | 2 |
| D | 2 |
+----+----+
```

使用说明

UDTF在SQL中的常用方式如下：

```
select user_udtf(col0, col1, col2) as (c0, c1) from my_table;
select user_udtf(col0, col1, col2) as (c0, c1) from
(select * from my_table distribute by key sort by key) t;
select reduce_udtf(col0, col1, col2) as (c0, c1) from
(select col0, col1, col2 from
(select map_udtf(a0, a1, a2, a3) as (col0, col1, col2) from my_table) t1
distribute by col0 sort by col0, col1) t2;
```

但使用UDTF有如下使用限制：

- 同一个SELECT子句中不允许有其他表达式

```
select value, user_udtf(key) as mycol ...
```

- UDTF不能嵌套使用

```
select user_udtf1(user_udtf2(key)) as mycol...
```

- 不支持在同一个select子句中与 group by / distribute by / sort by 联用

```
select user_udtf(key) as mycol ... group by mycol
```

其他UDTF示例

在UDTF中，用户可以读取ODPS的 资源 。下面将介绍利用udtf读取ODPS资源的示例。

编写udtf程序，编译成功后导出jar包(udtfexample1.jar)。

```
package com.aliyun.odps.examples.udf;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Iterator;

import com.aliyun.odps.udf.ExecutionContext;
import com.aliyun.odps.udf.UDFException;
import com.aliyun.odps.udf.UDTF;
import com.aliyun.odps.udf.annotation.Resolve;

/**
* project: example_project
* table: wc_in2
```

```
* partitions: p2=1,p1=2
* columns: colc,colb
*/
@Resolve({ "string,string->string,bigint,string" })
public class UDTFResource extends UDTF {
ExecutionContext ctx;
long fileResourceLineCount;
long tableResource1RecordCount;
long tableResource2RecordCount;

@Override
public void setup(ExecutionContext ctx) throws UDFException {
this.ctx = ctx;
try {
InputStream in = ctx.readResourceFileAsStream("file_resource.txt");
BufferedReader br = new BufferedReader(new InputStreamReader(in));
String line;
fileResourceLineCount = 0;
while ((line = br.readLine()) != null) {
fileResourceLineCount++;
}
br.close();

Iterator<Object[]> iterator = ctx.readResourceTable("table_resource1").iterator();
tableResource1RecordCount = 0;
while (iterator.hasNext()) {
tableResource1RecordCount++;
iterator.next();
}

iterator = ctx.readResourceTable("table_resource2").iterator();
tableResource2RecordCount = 0;
while (iterator.hasNext()) {
tableResource2RecordCount++;
iterator.next();
}

} catch (IOException e) {
throw new UDFException(e);
}
}

@Override
public void process(Object[] args) throws UDFException {
String a = (String) args[0];
long b = args[1] == null ? 0 : ((String) args[1]).length();

forward(a, b, "fileResourceLineCount=" + fileResourceLineCount + "|tableResource1RecordCount="
+ tableResource1RecordCount + "|tableResource2RecordCount=" + tableResource2RecordCount);

}
}
```

添加资源到ODPS:

```
Add file file_resource.txt;
Add jar udtfexample1.jar;
Add table table_resource1 as table_resource1;
Add table table_resource2 as table_resource2;
```

在ODPS中创建UDTF函数(my_udtf)：

```
create function mp_udtf as com.aliyun.odps.examples.udf.UDTFResource using 'udtfexample1.jar,
file_resource.txt, table_resource1, table_resource2';
```

在odps创建资源表table_resource1、table_resource2, 物理表tmp1。并插入相应的数据。

运行该udtf：

```
select mp_udtf("10","20") as (a, b, fileResourceLineCount) from table_resource1;

返回：
+-------+------------+-------+
| a | b | fileResourceLineCount |
+-------+------------+-------+
| 10 | 2 | fileResourceLineCount=3|tableResource1RecordCount=0|tableResource2RecordCount=0 |
| 10 | 2 | fileResourceLineCount=3|tableResource1RecordCount=0|tableResource2RecordCount=0 |
+-------+------------+-------+
```

附录

在ODPS SQL中的字符串常量可以用单引号或双引号表示，可以在单引号括起的字符串中包含双引号，或在双引号括起的字符串中包含单引号， 否则要用转义符来表达，如以下表达方式都是可以的

```
"I'm a happy manong!"
'I\'m a happy manong!'
```

在ODPS SQL中反斜线”\”是转义符，用来表达字符串中的特殊字符，或将其后跟的字符解释为其本身。当读入字符串常量时， 如果反斜线后跟三位有效的8进制数字，范围在001 ~177之间，系统会根据ASCII值转为相应的字符。对于以下情况，则会将其解释为特殊字符：

转义	字符
\b	backspace
\t	tab
\n	newline

\r	carriage-return
\’	单引号
\”	双引号
\ \	反斜线
\;	分号
\Z	control-Z
\0或\00	结束符

```
select length('a\tb') from dual;
```

结果是3，表示字符串里实际有三个字符,”\t”被视为一个字符。在转义符后的其它字符被解释为其本身。

```
select 'a\ab',length('a\ab') from dual;
```

结果是’ aab’，3。” \a”被解释成了普通的”a”。

在LIKE匹配时，” %”表示匹配任意多个字符，” _”表示匹配单个字符，如果要匹配”%”或”_”本身，则要对其进行转义, “\%”匹配字符”%” ，” _”匹配字符”_”。

```
'abcd' like 'ab%' -- true
'abcd' like 'ab\%' -- false
'ab%cd' like 'ab\\%%' -- true
```

备注：

- 关于字符串的字符集，目前ODPS SQL支持UTF-8的字符集，如果数据是以其它格式编码，可能计算出的结果不正确。

ODPS SQL中的正则表达式采用的是PCRE的规范，匹配时是按字符进行，支持的元字符如下：

```
元字符        说明
^             行首
$             行尾
.             任意字符
*             匹配零次或多次
+             匹配1次或多次
?             匹配零次或1次
?             匹配修饰符，当该字符紧跟在任何一个其他限制符 (* ,+, ?, {n}, {n,}, {n,m})后面时，匹配模式是非贪 婪的。
             非贪婪模式尽可能少的匹配所搜索的字符串， 而默认的贪婪模式则尽可能多的匹配所搜索的字符串。
A|B             A或B
(abc)*          匹配abc序列零次或多次
```

```
{n}或{m,n}       匹配的次数
[ab]           匹配括号中的任一字符,例中模式匹配a或b
[a-d]          匹配a,b,c,d任一字符
[^ab]          ^表示非，匹配任一非a非 b的字符
[::]           见下表POSIX字符组
\              转义符
\n             n为数字1-9，后向引用
\d             数字
\D             非数字
```

POSIX字符组

```
POSIX字符组       说明              范围
[[:alnum:]]      字母字符和数字字符    [a-zA-Z0-9]
[[:alpha:]]      字母              [a-zA-Z]
[[:ascii:]]      ASCII字符          [\x00-\x7F]
[[:blank:]]      空格字符和制表符    [ \t]
[[:cntrl:]]      控制字符           [\x00-\x1F\x7F]
[[:digit:]]      数字字符           [0-9]
[[:graph:]]      空白字符之外的字符    [\x21-\x7E]
[[:lower:]]      小写字母字符        [a-z]
[[:print:]]      [:graph:]和空白字符    [\x20-\x7E]
[[:punct:]]      标点符号           [][!"#$%&'()*+,./:;<=>? @\^_`{|}~-]
[[:space:]]      空白字符           [ \t\r\n\v\f]
[[:upper:]]      大写字母字符        [A-Z]
[[:xdigit:]]    十六进制字符        [A-Fa-f0-9]
```

由于系统采用反斜线"\"作为转义符，因此正则表达式的模式中出现的"\"都要进行二次转义。如正则表达式要匹配字符串"a+b"，其中"+"是正则中的一个特殊字符，因此要用转义的方式表达，在正则引擎中的表达方式是"a\+b"，由于系统还要解释一层转义，因此能够匹配该字符串的表达式是"a\\+b"。例如，假设存在表test_dual，

```
select 'a+b' rlike 'a\\\+b' from test_dual;

+------+
| _c1  |
+------+
| true |
+------+
```

极端的情况，如果在要匹配字符"\"，由于在正则引擎中"\"是一个特殊字符，因此要表示为"\"，而系统还要对表达式进行一次转义，因此写成"\\"

```
select 'a\\b', 'a\\b' rlike 'a\\\\b' from test_dual;

+-----+------+
| _c0 | _c1  |
+-----+------+
| a\b | true |
+-----+------+
```

备注:

- 在ODPS SQL中写”a\b” ， 而在输出结果中显示’ a\b’ ，同样是因为ODPS会对表达式进行转义。

如果字符串中有制表符TAB, 系统在读入’ \t’ 这两个字符的时候，即已经将其存为一个字符，因此在正则的模式中也是一个普通的字符。

```
select 'a\tb', 'a\tb' rlike 'a\tb' from test_dual;

+---------+------+
| _c0     | _c1 |
+---------+------+
| a     b | true |
+---------+------+
```

以下ODPS SQL的全部保留字，在对表、列或是分区命名时请不要使用，否则会报错。保留字不区分大小写。

```
  %   &   &&   (   )   *   +
  -   .   /   ;   <   <=   <>
  =   >   >=   ?   ADD   AFTER   ALL
  ALTER   ANALYZE   AND   ARCHIVE   ARRAY   AS   ASC
  BEFORE   BETWEEN   BIGINT   BINARY   BLOB   BOOLEAN   BOTH
  BUCKET   BUCKETS   BY   CASCADE   CASE   CAST   CFILE
  CHANGE   CLUSTER   CLUSTERED   CLUSTERSTATUS   COLLECTION   COLUMN   COLUMNS
  COMMENT   COMPUTE   CONCATENATE   CONTINUE   CREATE   CROSS   CURRENT
  CURSOR   DATA   DATABASE   DATABASES   DATE   DATETIME   DBPROPERTIES
  DEFERRED   DELETE   DELIMITED   DESC   DESCRIBE   DIRECTORY   DISABLE
  DISTINCT   DISTRIBUTE   DOUBLE   DROP   ELSE   ENABLE   END
  ESCAPED   EXCLUSIVE   EXISTS   EXPLAIN   EXPORT   EXTENDED   EXTERNAL
  FALSE   FETCH   FIELDS   FILEFORMAT   FIRST   FLOAT   FOLLOWING
  FORMAT   FORMATTED   FROM   FULL   FUNCTION   FUNCTIONS   GRANT
  GROUP   HAVING   HOLD_DDLTIME   IDXPROPERTIES   IF   IMPORT   IN
  INDEX   INDEXES   INPATH   INPUTDRIVER   INPUTFORMAT   INSERT   INT
  INTERSECT   INTO   IS   ITEMS   JOIN   KEYS   LATERAL
  LEFT   LIFECYCLE   LIKE   LIMIT   LINES   LOAD   LOCAL
  LOCATION   LOCK   LOCKS   LONG   MAP   MAPJOIN   MATERIALIZED
  MINUS   MSCK   NOT   NO_DROP   NULL   OF   OFFLINE
  ON   OPTION   OR   ORDER   OUT   OUTER   OUTPUTDRIVER
  OUTPUTFORMAT   OVER   OVERWRITE   PARTITION   PARTITIONED   PARTITIONPROPERTIES   PARTITIONS
  PERCENT   PLUS   PRECEDING   PRESERVE   PROCEDURE   PURGE   RANGE
  RCFILE   READ   READONLY   READS   REBUILD   RECORDREADER   RECORDWRITER
  REDUCE   REGEXP   RENAME   REPAIR   REPLACE   RESTRICT   REVOKE
  RIGHT   RLIKE   ROW   ROWS   SCHEMA   SCHEMAS   SELECT
  SEMI   SEQUENCEFILE   SERDE   SERDEPROPERTIES   SET   SHARED   SHOW
  SHOW_DATABASE   SMALLINT   SORT   SORTED   SSL   STATISTICS   STORED
  STREAMTABLE   STRING   STRUCT   TABLE   TABLES   TABLESAMPLE   TBLPROPERTIES
  TEMPORARY   TERMINATED   TEXTFILE   THEN   TIMESTAMP   TINYINT   TO
  TOUCH   TRANSFORM   TRIGGER   TRUE   UNARCHIVE   UNBOUNDED   UNDO
  UNION   UNIONTYPE   UNIQUEJOIN   UNLOCK   UNSIGNED   UPDATE   USE
  USING   UTC   UTC_TMESTAMP   VIEW   WHEN   WHERE   WHILE
```