最强 HiveSQL 开发指南

本文档来自公众号：**五分钟学大数据**

微信扫码关注

目录

第一部分：.....3
Hive 查询语句.....5
第二部分.....9
1. 对数据库的操作.....9
2. 对数据表的操作.....9
hive 的 DQL 查询语法.....17
单表查询.....17
Hive 函数.....20
聚合函数.....20
关系运算.....21
数学运算.....21
逻辑运算.....22
数值运算.....22
条件函数.....24
日期函数.....25
字符串函数.....28
复合类型构建操作.....33
复杂类型访问操作.....34
复杂类型长度统计函数.....35
hive 当中的 lateral view 与 explode 以及 reflect.....36
使用 explode 函数将 hive 表中的 Map 和 Array 字段数据进行拆分.....36
使用 explode 拆分 json 字符串.....37
配合 LATERAL VIEW 使用.....39
行转列.....40
列转行.....42
reflect 函数.....43
Hive 窗口函数.....45

本文整体分为两部分，第一部分是简写，如果能看懂会用，就直接从此部分查，方便快捷，如果不是很理解此 SQL 的用法，则查看第二部分，是详细说明，当然第二部分语句也会更全一些！

第一部分：

1. hive 模糊搜索表：`show tables like '*name*';`
2. 查看表结构信息：`desc table_name;`
3. 查看分区信息：`show partitions table_name;`
4. 加载本地文件：`load data local inpath '/xxx/test.txt' overwrite into table dm.table_name;`
5. 从查询语句给 table 插入数据：`insert overwrite table table_name partition(dt) select * from table_name;`
6. 导出数据到本地系统：`insert overwrite local directory '/tmp/text' select a.* from table_name a order by 1;`
7. 创建表时指定的一些属性：

- 字段分隔符：`row format delimited fields terminated by '\t'`
- 行分隔符：`row format delimited lines terminated by '\n'`
- 文件格式为文本型存储：`stored as textfile`

8. 命令行操作：`hive -e 'select table_cloum from table'` 执行一个查询，在终端上显示 mapreduce 的进度，执行完毕后，最后把查询结果输出到终端上，接着 hive 进程退出，不会进入交互模式

`hive -S -e 'select table_cloum from table'` -S，终端上的输出不会有 mapreduce 的进度，执行完毕，只会把查询结果输出到终端上。

9. hive 修改表名：`alter table old_table_name rename to new_table_name;`
10. hive 复制表结构：`create table new_table_name like table_name;`
11. hive 添加字段：`alter table table_name add columns(columns_values bigint comment 'comm_text');`
12. hive 修改字段：`alter table table_name change old_column new_column string comment 'comm_text';`
13. 删除分区：`alter table table_name drop partition(dt='2021-11-30');`
14. 添加分区：`alter table table_name add partition (dt='2021-11-30');`
15. 删除空数据库：`drop database myhive2;`

16. 强制删除数据库：`drop database myhive2 cascade;`
17. 删除表：`drop table score5;`
18. 清空表：`truncate table score6;`
19. 向 hive 表中加载数据

- 直接向分区表中插入数据：`insert into table score partition(month ='202107') values ('001','002','100');`
- 通过 load 方式加载数据：`load data local inpath '/export/servers/hivedatas/score.csv' overwrite into table score partition(month='201806');`
- 通过查询方式加载数据：`insert overwrite table score2 partition(month = '202106') select s_id,c_id,s_score from score1;`
- 查询语句中创建表并加载数据：`create table score2 as select * from score1;`
- 在创建表是通过 location 指定加载数据的路径：`create external table score6 (s_id string,c_id string,s_score int) row format delimited fields terminated by ',' location '/myscore';`
- export 导出与 import 导入 hive 表数据（内部表操作）：

```
create table techer2 like techer; --依据已有表结构创建表
export table techer to '/export/techer';
import table techer2 from '/export/techer';
```

20. hive 表中数据导出

- insert 导出

将查询的结果导出到本地：`insert overwrite local directory '/export/servers/exporthive' select * from score;`

将查询的结果格式化导出到本地：`insert overwrite local directory '/export/servers/exporthive' row format delimited fields terminated by '\t' collection items terminated by '#' select * from student;`

将查询的结果导出到 HDFS 上(没有 local)：`insert overwrite directory '/export/servers/exporthive' row format delimited fields terminated by '\t' collection items terminated by '#' select * from score;`

- Hadoop 命令导出到本地：`dfs -get /export/servers/exporthive/000000_0 /export/servers/exporthive/local.txt;`

- hive shell 命令导出

基本语法：（hive -f/-e 执行语句或者脚本 > file） `hive -e "select * from myhive.score;" > /export/servers/exporthive/score.txt`

`hive -f export.sh > /export/servers/exporthive/score.txt`

- export 导出到 HDFS 上： `export table score to '/export/exporthive/score';`

Hive 查询语句

1. GROUP BY 分组：`select s_id ,avg(s_score) avgscore from score group by s_id having avgscore > 85;` 对分组后的数据进行筛选，使用 having
2. join 连接：inner join 内连接；left join 左连接；right join 右链接；full join 全外链接。
3. order by 排序：ASC（ascend）：升序（默认） DESC（descend）：降序
4. sort by 局部排序：每个 MapReduce 内部进行排序，对全局结果集来说不是排序。
5. distribute by 分区排序：类似 MR 中 partition，进行分区，结合 sort by 使用

Hive 函数

1. 聚合函数

1. 指定列值的数目：count()
2. 指定列值求和：sum()
3. 指定列的最大值：max()
4. 指定列的最小值：min()
5. 指定列的平均值：avg()
6. 非空集合总体变量函数：var_pop(col)
7. 非空集合样本变量函数：var_samp (col)
8. 总体标准偏离函数：stddev_pop(col)
9. 分位数函数：percentile(BIGINT col, p)
10. 中位数函数：percentile(BIGINT col, 0.5)

2. 关系运算

1. A LIKE B： LIKE 比较，如果字符串 A 符合表达式 B 的正则语法，则为 TRUE
2. A RLIKE B：JAVA 的 LIKE 操作，如果字符串 A 符合 JAVA 正则表达式 B 的正则语法，则为 TRUE
3. A REGEXP B：功能与 RLIKE 相同

3. 数学运算

支持所有数值类型：加(+)、减(-)、乘(*)、除(/)、取余(%)、位与(&)、位或(|)、位异或(^)、位取反(~)

4. 逻辑运算

支持：逻辑与(and)、逻辑或(or)、逻辑非(not)

5. 数值运算

1. 取整函数：round(double a)
2. 指定精度取整函数：round(double a, int d)
3. 向下取整函数：floor(double a)
4. 向上取整函数：ceil(double a)
5. 取随机数函数：rand(),rand(int seed)
6. 自然指数函数：exp(double a)
7. 以 10 为底对数函数：log10(double a)
8. 以 2 为底对数函数：log2()
9. 对数函数：log()
10. 幂运算函数：pow(double a, double p)
11. 开平方函数：sqrt(double a)
12. 二进制函数：bin(BIGINT a)
13. 十六进制函数：hex()
14. 绝对值函数：abs()
15. 正取余函数：pmod()

6. 条件函数

1. if
2. case when
3. coalesce(c1,c2,c3)
4. nvl(c1, c2)

7. 日期函数

1. 获得当前时区的 UNIX 时间戳: unix_timestamp()
2. 时间戳转日期函数：from_unixtime()
3. 日期转时间戳：unix_timestamp(string date)
4. 日期时间转日期函数：to_date(string timestamp)
5. 日期转年函数：year(string date)
6. 日期转月函数：month (string date)
7. 日期转天函数: day (string date)
8. 日期转小时函数: hour (string date)
9. 日期转分钟函数：minute (string date)
10. 日期转秒函数: second (string date)
11. 日期转周函数: weekofyear (string date)
12. 日期比较函数: datediff(string enddate, string startdate)
13. 日期增加函数: date_add(string startdate, int days)
14. 日期减少函数：date_sub (string startdate, int days)

8. 字符串函数

1. 字符串长度函数：length(string A)
2. 字符串反转函数：reverse(string A)
3. 字符串连接函数: concat(string A, string B…)
4. 带分隔符字符串连接函数：concat_ws(string SEP, string A, string B…)
5. 字符串截取函数: substr(string A, int start, int len)
6. 字符串转大写函数: upper(string A)
7. 字符串转小写函数：lower(string A)
8. 去空格函数：trim(string A)

9. 左边去空格函数：ltrim(string A)
10. 右边去空格函数：rtrim(string A)
11. 正则表达式替换函数： regexp_replace(string A, string B, string C)
12. 正则表达式解析函数: regexp_extract(string subject, string pattern, int index)
13. URL 解析函数：parse_url(string urlString, string partToExtract [, string keyToExtract]) 返回值: string
14. json 解析函数：get_json_object(string json_string, string path)
15. 空格字符串函数：space(int n)
16. 重复字符串函数：repeat(string str, int n)
17. 首字符 ascii 函数：ascii(string str)
18. 左补足函数：lpad(string str, int len, string pad)
19. 右补足函数：rpad(string str, int len, string pad)
20. 分割字符串函数: split(string str, string pat)
21. 集合查找函数: find_in_set(string str, string strList)

9. 窗口函数

1. 分组求和函数：sum(pv) over(partition by cookieid order by createtime)
   有坑，加不加 order by 差别很大，具体详情在下面第二部分。
2. 分组内排序，从 1 开始顺序排：ROW_NUMBER() 如：1234567
3. 分组内排序，排名相等会在名次中留下空位：RANK() 如：1233567
4. 分组内排序,排名相等不会在名次中留下空位:DENSE_RANK() 如:1233456
5. 有序的数据集合平均分配到指定的数量（num）个桶中：NTILE()
6. 统计窗口内往上第 n 行值：LAG(col,n,DEFAULT)
7. 统计窗口内往下第 n 行值：LEAD(col,n,DEFAULT)
8. 分组内排序后，截止到当前行，第一个值：FIRST_VALUE(col)
9. 分组内排序后，截止到当前行，最后一个值: LAST_VALUE(col)
10. 小于等于当前值的行数/分组内总行数：CUME_DIST()

以下函数建议看第二部分详细理解下，此处仅简写，！

11. 将多个 group by 逻辑写在一个 sql 语句中: GROUPING SETS
12. 根据 GROUP BY 的维度的所有组合进行聚合：CUBE
13. CUBE 的子集，以最左侧的维度为主，从该维度进行层级聚合：ROLLUP

第二部分

1. 对数据库的操作

- 创建数据库:

```
create database if not exists myhive;
说明：hive 的表存放位置模式是由 hive-site.xml 当中的一个属性指定
的 :hive.metastore.warehouse.dir

创建数据库并指定 hdfs 存储位置 :
create database myhive2 location '/myhive2';
```

- 修改数据库:

```
alter  database  myhive2  set  dbproperties('createtime'='20210329');
```

说明：可以使用 alter database 命令来修改数据库的一些属性。但是数据库的元数据信息是不可更改的，包括数据库的名称以及数据库所在的位置

- 查看数据库详细信息

```
查看数据库基本信息
hive (myhive)> desc  database  myhive2;

查看数据库更多详细信息
hive (myhive)> desc database extended  myhive2;
```

- 删除数据库

```
删除一个空数据库，如果数据库下面有数据表，那么就会报错
drop  database  myhive2;

强制删除数据库，包含数据库下面的表一起删除
drop  database  myhive  cascade;
```

2. 对数据表的操作

对管理表(内部表)的操作:

- 建内部表:

```
hive (myhive)> use myhive; -- 使用myhive数据库
hive (myhive)> create table stu(id int,name string);
hive (myhive)> insert into stu values (1,"zhangsan");
hive (myhive)> insert into stu values (1,"zhangsan"),(2,"lisi");  -- 一次插入多条数据
hive (myhive)> select * from stu;
```

- hive 建表时候的字段类型:

分类	类型	描述	字面量示例
原始类型	BOOLEAN	true/false	TRUE
	TINYINT	1 字节的有符号整数 -128~127	1Y
	SMALLINT	2 个字节的有符号整数，-32768~32767	1S
	INT	4 个字节的带符号整数	1
	BIGINT	8 字节带符号整数	1L
	FLOAT	4 字节单精度浮点数 1.0	
	DOUBLE	8 字节双精度浮点数	1.0
	DEICIMAL	任意精度的带符号小数	1.0
	STRING	字符串，变长	“a”,’b’
	VARCHAR	变长字符串	“a”,’b’
	CHAR	固定长度字符串	“a”,’b’

分类	类型	描述	字面量示例
	BINARY	字节数组	无法表示
	TIMESTAMP	时间戳，毫秒值精度	122327493795
	DATE	日期	‘2016-03-29’
	INTERVAL	时间频率间隔	
复杂类型	ARRAY	有序的的同类型的集合	array(1,2)
	MAP	key-value,key 必须为原始类型，value 可以任意类型	map(‘a’ ,1,’ b’ ,2)
	STRUCT	字段集合,类型可以不同	struct(‘1’ ,1,1.0), named_stract(‘col1’ ,’ 1’ ,’ col2’ ,1,’ clo3’ ,1.0)
	UNION	在有限取值范围内的一个值	create_union(1,’ a’ ,63)

对 decimal 类型简单解释下：

用法：decimal(11,2) 代表最多有 11 位数字，其中后 2 位是小数，整数部分是 9 位；如果整数部分超过 9 位，则这个字段就会变成 null；如果小数部分不足 2 位，则后面用 0 补齐两位，如果小数部分超过两位，则超出部分四舍五入

也可直接写 decimal，后面不指定位数，默认是 decimal(10,0) 整数 10 位，没有小数

- 创建表并指定字段之间的分隔符

```
create  table if not exists stu2(id int ,name string) row format delimited fields terminated by '\t' stored as textfile location '/user/stu2';
```

row format delimited fields terminated by ’\t’ 指定字段分隔符，默认分隔符为 ’\001’

```
stored as 指定存储格式
location 指定存储位置
```

- 根据查询结果创建表

```
create table stu3 as select * from stu2;
```

- 根据已经存在的表结构创建表

```
create table stu4 like stu2;
```

- 查询表的结构

```
只查询表内字段及属性
desc stu2;

详细查询
desc formatted  stu2;
```

- 查询创建表的语句

```
show create table stu2;
```

对外部表操作

外部表因为是指定其他的 hdfs 路径的数据加载到表当中来，所以 hive 表会认为自己不完全独占这份数据，所以删除 hive 表的时候，数据仍然存放在 hdfs 当中，不会删掉，只会删除表的元数据

- 构建外部表

```
create external table student (s_id string,s_name string) row format delimited fiel
ds terminated by '\t';
```

- 从本地文件系统向表中加载数据

```
追加操作
load data local inpath '/export/servers/hivedatas/student.csv' into table student;

覆盖操作
load data local inpath '/export/servers/hivedatas/student.csv' overwrite  into tabl
e student;
```

- 从 hdfs 文件系统向表中加载数据

```
load data inpath '/hivedatas/techer.csv' into table techer;
```

加载数据到指定分区

```
load data inpath '/hivedatas/techer.csv' into table techer partition(cur_date=20201
210);
```

- 注意：
  1. 使用 load data local 表示从本地文件系统加载，文件会拷贝到 hdfs 上
  2. 使用 load data 表示从 hdfs 文件系统加载，文件会直接移动到 hive 相关目录下，注意不是拷贝过去，因为 hive 认为 hdfs 文件已经有 3 副本了，没必要再次拷贝了
  3. 如果表是分区表，load 时不指定分区会报错
  4. 如果加载相同文件名的文件，会被自动重命名

对分区表的操作

- 创建分区表的语法

```
create table score(s_id string, s_score int) partitioned by (month string);
```

- 创建一个表带多个分区

```
create table score2 (s_id string, s_score int) partitioned by (year string,month st
ring,day string);
```

注意：
hive 表创建的时候可以用 location 指定一个文件或者文件夹，当指定文件夹时，hive 会加载文件夹下的所有文件，当表中无分区时，这个文件夹下不能再有文件夹，否则报错
当表是分区表时，比如 partitioned by (day string)， 则这个文件夹下的每一个文件夹就是一个分区，且文件夹名为 day=20201123 这种格式，然后使用：msck repair table score; 修复表结构，成功之后即可看到数据已经全部加载到表当中去了

- 加载数据到一个分区的表中

```
load data local inpath '/export/servers/hivedatas/score.csv' into table score partition (month='201806');
```

- 加载数据到一个多分区的表中去

```
load data local inpath '/export/servers/hivedatas/score.csv' into table score2 partition(year='2018',month='06',day='01');
```

- 查看分区

```
show  partitions  score;
```

- 添加一个分区

```
alter table score add partition(month='201805');
```

- 同时添加多个分区

```
 alter table score add partition(month='201804') partition(month = '201803');
```

注意：添加分区之后就可以在 hdfs 文件系统当中看到表下面多了一个文件夹

- 删除分区

```
 alter table score drop partition(month = '201806');
```

对分桶表操作

将数据按照指定的字段进行分成多个桶中去，就是按照分桶字段进行哈希划分到多个文件当中去
分区就是分文件夹，分桶就是分文件
分桶优点：
1. 提高 join 查询效率
2. 提高抽样效率

- 开启 hive 的捅表功能

```
set hive.enforce.bucketing=true;
```

- 设置 reduce 的个数

```
set mapreduce.job.reduces=3;
```

- 创建桶表

```
create table course (c_id string,c_name string) clustered by(c_id) into 3 buckets;
```

桶表的数据加载：由于桶表的数据加载通过 hdfs dfs -put 文件或者通过 load data 均不可以，只能通过 insert overwrite 进行加载

所以把文件加载到桶表中，需要先创建普通表，并通过 insert overwrite 的方式将普通表的数据通过查询的方式加载到桶表当中去

- 通过 insert overwrite 给桶表中加载数据

```
insert overwrite table course select * from course_common cluster by(c_id);  -- 最后指定桶字段
```

修改表和删除表

- 修改表名称

```
alter  table  old_table_name  rename  to  new_table_name;
```

- 增加/修改列信息

```
查询表结构
desc score5;

添加列
alter table score5 add columns (mycol string, mysco string);

更新列
alter table score5 change column mysco mysconew int;
```

- 删除表操作

```
drop table score5;
```

- 清空表操作

```
truncate table score6;
```

```
说明：只能清空管理表，也就是内部表；清空外部表，会产生错误
```

注意：truncate 和 drop：

如果 hdfs 开启了回收站，drop 删除的表数据是可以从回收站恢复的，表结构恢复不了，需要自己重新创建；truncate 清空的表是不进回收站的，所以无法恢复

truncate 清空的表
所以 truncate 一定慎用，一旦清空将无力回天

向 hive 表中加载数据

- 直接向分区表中插入数据

```
insert into table score partition(month ='201807') values ('001','002','100');
```

- 通过 load 方式加载数据

```
 load data local inpath '/export/servers/hivedatas/score.csv' overwrite into table score partition(month='201806');
```

- 通过查询方式加载数据

```
insert overwrite table score2 partition(month = '201806') select s_id,c_id,s_score from score1;
```

- 查询语句中创建表并加载数据

```
create table score2 as select * from score1;
```

- 在创建表是通过 location 指定加载数据的路径

```
create external table score6 (s_id string,c_id string,s_score int) row format delimited fields terminated by ',' location '/myscore';
```

- export 导出与 import 导入 hive 表数据（内部表操作）

```
create table techer2 like techer; --依据已有表结构创建表

export table techer to  '/export/techer';

import table techer2 from '/export/techer';
```

hive 表中数据导出

- insert 导出

```
将查询的结果导出到本地
insert overwrite local directory '/export/servers/exporthive' select * from score;
```

```
将查询的结果格式化导出到本地
insert overwrite local directory '/export/servers/exporthive' row format delimited
fields terminated by '\t' collection items terminated by '#' select * from student;

将查询的结果导出到 HDFS 上(没有 local)
insert overwrite directory '/export/servers/exporthive' row format delimited fields
 terminated by '\t' collection items terminated by '#' select * from score;
```

- Hadoop 命令导出到本地

```
dfs -get /export/servers/exporthive/000000_0 /export/servers/exporthive/local.txt;
```

- hive shell 命令导出

```
基本语法：（hive -f/-e 执行语句或者脚本 > file）

hive -e "select * from myhive.score;" > /export/servers/exporthive/score.txt

hive -f export.sh > /export/servers/exporthive/score.txt
```

- export 导出到 HDFS 上

```
export table score to '/export/exporthive/score';
```

hive 的 DQL 查询语法

单表查询

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list [HAVING condition]]
[CLUSTER BY col_list
  | [DISTRIBUTE BY col_list] [SORT BY| ORDER BY col_list]
]
[LIMIT number]
```

注意：

1、order by 会对输入做全局排序，因此只有一个 reducer，会导致当输入规模较大时，需要较长的计算时间。

2、sort by 不是全局排序，其在数据进入 reducer 前完成排序。因此，如果用 sort by 进行排序，并且设置 mapred.reduce.tasks>1，则 sort by 只保证每个 reducer 的输出有序，不保证全局有序。

3、distribute by(字段)根据指定的字段将数据分到不同的 reducer，且分发算法是 hash 散列。
4、Cluster by(字段) 除了具有 Distribute by 的功能外，还会对该字段进行排序。因此，如果分桶和 sort 字段是同一个时，此时，cluster by = distribute by + sort by

- WHERE 语句

```
select * from score where s_score < 60;
```

注意：
小于某个值是不包含 null 的，如上查询结果是把 s_score 为 null 的行剔除的

- GROUP BY 分组

```
select s_id ,avg(s_score) from score group by s_id;
```

```
分组后对数据进行筛选，使用 having
 select s_id ,avg(s_score) avgscore from score group by s_id having avgscore > 85;
```

注意：
如果使用 group by 分组，则 select 后面只能写分组的字段或者聚合函数
where 和 having 区别：
1 having 是在 group by 分完组之后再对数据进行筛选，所以 having 要筛选的字段只能是分组字段或者聚合函数
2 where 是从数据表中的字段直接进行的筛选的，所以不能跟在 gruop by 后面，也不能使用聚合函数

- join 连接

```
INNER JOIN 内连接：只有进行连接的两个表中都存在与连接条件相匹配的数据才会被保留下来
select * from techer t [inner] join course c on t.t_id = c.t_id; -- inner 可省略

LEFT OUTER JOIN 左外连接：左边所有数据会被返回，右边符合条件的被返回
select * from techer t left join course c on t.t_id = c.t_id; -- outer 可省略

RIGHT OUTER JOIN 右外连接：右边所有数据会被返回，左边符合条件的被返回、
select * from techer t right join course c on t.t_id = c.t_id;

FULL OUTER JOIN 满外(全外)连接：将会返回所有表中符合条件的所有记录。如果任一表的指定字段没有符合条件的值的话，那么就使用 NULL 值替代。
SELECT * FROM techer t FULL JOIN course c ON t.t_id = c.t_id ;
```

注：1. hive2 版本已经支持不等值连接，就是 join on 条件后面可以使用大于小于符号了;并且也支持 join on 条件后跟 or（早前版本 on 后只支持 = 和 and，不支持 >< 和 or）
2. 如 hive 执行引擎使用 MapReduce，一个 join 就会启动一个 job，一条 sql 语句中如有多个 join，则会启动多个 job
注意：表之间用逗号(,)连接和 inner join 是一样的

```
select * from table_a,table_b where table_a.id=table_b.id;
```

它们的执行效率没有区别，只是书写方式不同，用逗号是 sql 89 标准，join 是 sql 92 标准。用逗号连接后面过滤条件用 where ，用 join 连接后面过滤条件是 on。

- order by 排序

```
全局排序，只会有一个 reduce
ASC（ascend）：升序（默认） DESC（descend）：降序
SELECT * FROM student s LEFT JOIN score sco ON s.s_id = sco.s_id ORDER BY sco.s_sco
re DESC;
```

注意：order by 是全局排序，所以最后只有一个 reduce，也就是在一个节点执行，如果数据量太大，就会耗费较长时间

- sort by 局部排序

```
每个 MapReduce 内部进行排序，对全局结果集来说不是排序。

设置 reduce 个数
set mapreduce.job.reduces=3;

查看设置 reduce 个数
set mapreduce.job.reduces;

查询成绩按照成绩降序排列
select * from score sort by s_score;

将查询结果导入到文件中（按照成绩降序排列）
insert overwrite local directory '/export/servers/hivedatas/sort' select * from sco
re sort by s_score;
```

- distribute by 分区排序

```
distribute by：类似 MR 中 partition，进行分区，结合 sort by 使用

设置 reduce 的个数，将我们对应的 s_id 划分到对应的 reduce 当中去
set mapreduce.job.reduces=7;
```

通过 distribute by 进行数据的分区

```
select * from score distribute by s_id sort by s_score;
```

注意：Hive 要求 distribute by 语句要写在 sort by 语句之前

- cluster by

当 distribute by 和 sort by 字段相同时，可以使用 cluster by 方式.

cluster by 除了具有 distribute by 的功能外还兼具 sort by 的功能。但是排序只能是正序排序，不能指定排序规则为 ASC 或者 DESC。

以下两种写法等价

```
select * from score cluster by s_id;
select * from score distribute by s_id sort by s_id;
```

Hive 函数

聚合函数

hive 支持 count(),max(),min(),sum(),avg() 等常用的聚合函数

注意：

聚合操作时要注意 null 值

count(*) 包含 null 值，统计所有行数

count(id) 不包含 null 值

min 求最小值是不包含 null，除非所有值都是 null

avg 求平均值也是不包含 null

- 非空集合总体变量函数：var_pop

语法：var_pop(col)

返回值：double

说明：统计结果集中 col 非空集合的总体变量（忽略 null）

- 非空集合样本变量函数：var_samp

语法：var_samp (col)

返回值：double

说明：统计结果集中 col 非空集合的样本变量（忽略 null）

- 总体标准偏离函数：stddev_pop

语法：stddev_pop(col)
返回值：double
说明：该函数计算总体标准偏离，并返回总体变量的平方根，其返回值与 VAR_POP 函数的平方根相同

- 中位数函数：percentile

语法：percentile(BIGINT col, p)
返回值：double
说明：求准确的第 pth 个百分位数，p 必须介于 0 和 1 之间，但是 col 字段目前只支持整数，不支持浮点数类型

关系运算

支持：等值(=)、不等值(!= 或 <>)、小于(<)、小于等于(<=)、大于(>)、大于等于(>=)

空值判断(is null)、非空判断(is not null)

- LIKE 比较：LIKE

语法：A LIKE B
操作类型：strings
描述：如果字符串 A 或者字符串 B 为 NULL，则返回 NULL；如果字符串 A 符合表达式 B 的正则语法，则为 TRUE；否则为 FALSE。B 中字符”_”表示任意单个字符，而字符”%”表示任意数量的字符。

- JAVA 的 LIKE 操作：RLIKE

语法：A RLIKE B
操作类型：strings
描述：如果字符串 A 或者字符串 B 为 NULL，则返回 NULL；如果字符串 A 符合 JAVA 正则表达式 B 的正则语法，则为 TRUE；否则为 FALSE。

- REGEXP 操作：REGEXP

语法：A REGEXP B
操作类型：strings
描述：功能与 RLIKE 相同
示例：select 1 from tableName where 'footbar' REGEXP '^f.*r$';
结果：1

数学运算

支持所有数值类型：加(+)、减(-)、乘(*)、除(/)、取余(%)、位与(&)、位或(|)、位异或(^)、位取反(~)

逻辑运算

支持：逻辑与(and)、逻辑或(or)、逻辑非(not)

数值运算

- 取整函数：round

```
语法：round(double a)
返回值：BIGINT
说明：返回 double 类型的整数值部分 （遵循四舍五入）
示例：select round(3.1415926) from tableName;
结果：3
```

- 指定精度取整函数：round

```
语法：round(double a, int d)
返回值：DOUBLE
说明：返回指定精度 d 的 double 类型
hive> select round(3.1415926,4) from tableName;
3.1416
```

- 向下取整函数：floor

```
语法：floor(double a)
返回值：BIGINT
说明：返回等于或者小于该 double 变量的最大的整数
hive> select floor(3.641) from tableName;
3
```

- 向上取整函数：ceil

```
语法：ceil(double a)
返回值：BIGINT
说明：返回等于或者大于该 double 变量的最小的整数
hive> select ceil(3.1415926) from tableName;
4
```

- 取随机数函数：rand

```
语法：rand(),rand(int seed)
返回值：double
说明：返回一个 0 到 1 范围内的随机数。如果指定种子 seed，则会等到一个稳定的随机数序列
hive> select rand() from tableName; -- 每次执行此语句得到的结果都不同
```

```
0.5577432776034763

hive> select rand(100) ;  -- 只要指定种子，每次执行此语句得到的结果一样的
0.7220096548596434
```

- 自然指数函数：exp

```
语法: exp(double a)
返回值: double
说明: 返回自然对数 e 的 a 次方
hive> select exp(2) ;
7.38905609893065
```

- 以 10 为底对数函数：log10

```
语法: log10(double a)
返回值: double
说明: 返回以 10 为底的 a 的对数
hive> select log10(100) ;
2.0
```

此外还有：以 2 为底对数函数：log2()、对数函数：log()

- 幂运算函数：pow

```
语法: pow(double a, double p)
返回值: double
说明: 返回 a 的 p 次幂
hive> select pow(2,4) ;
16.0
```

- 开平方函数：sqrt

```
语法: sqrt(double a)
返回值: double
说明: 返回 a 的平方根
hive> select sqrt(16) ;
4.0
```

- 二进制函数：bin

```
语法: bin(BIGINT a)
返回值: string
说明: 返回 a 的二进制代码表示
hive> select bin(7) ;
111
```

十六进制函数：hex()、将十六进制转化为字符串函数：unhex()
进制转换函数：conv(bigint num, int from_base, int to_base) 说明：将数值 num 从 from_base 进制转化到 to_base 进制
此外还有很多数学函数： 绝对值函数：abs()、正取余函数：pmod()、正弦函数：sin()、反正弦函数：asin()、余弦函数：cos()、反余弦函数：acos()、positive 函数：positive()、negative 函数：negative()

条件函数

- If 函数：if

```
语法：if(boolean testCondition, T valueTrue, T valueFalseOrNull)
返回值：T
说明：当条件 testCondition 为 TRUE 时，返回 valueTrue；否则返回 valueFalseOrNull
hive> select if(1=2,100,200) ;
200
hive> select if(1=1,100,200) ;
100
```

- 非空查找函数：coalesce

```
语法：coalesce(T v1, T v2, …)
返回值：T
说明：返回参数中的第一个非空值；如果所有值都为 NULL，那么返回 NULL
hive> select coalesce(null,'100','50') ;
100
```

- 条件判断函数：case when（两种写法，其一）

```
语法：case when a then b [when c then d]* [else e] end
返回值：T
说明：如果 a 为 TRUE,则返回 b；如果 c 为 TRUE，则返回 d；否则返回 e
hive> select case when 1=2 then 'tom' when 2=2 then 'mary' else 'tim' end from tabl
eName;
mary
```

- 条件判断函数：case when（两种写法，其二）

```
语法：case a when b then c [when d then e]* [else f] end
返回值：T
说明：如果 a 等于 b，那么返回 c；如果 a 等于 d，那么返回 e；否则返回 f
hive> Select case 100 when 50 then 'tom' when 100 then 'mary' else 'tim' end from t
```

```
ableName;
mary
```

日期函数

注：以下 SQL 语句中的 from tableName 可去掉，不影响查询结果

- 1. 获取当前 UNIX 时间戳函数：unix_timestamp

```
语法: unix_timestamp()
返回值: bigint
说明: 获得当前时区的 UNIX 时间戳
hive> select unix_timestamp() from tableName;
1616906976
```

- 2. UNIX 时间戳转日期函数：from_unixtime

```
语法: from_unixtime(bigint unixtime[, string format])
返回值: string
说明: 转化 UNIX 时间戳（从 1970-01-01 00:00:00 UTC 到指定时间的秒数）到当前时区的时间格式
hive> select from_unixtime(1616906976,'yyyyMMdd') from tableName;
20210328
```

- 3. 日期转 UNIX 时间戳函数：unix_timestamp

```
语法: unix_timestamp(string date)
返回值: bigint
说明: 转换格式为"yyyy-MM-dd HH:mm:ss"的日期到 UNIX 时间戳。如果转化失败，则返回 0。
hive>  select unix_timestamp('2021-03-08 14:21:15') from tableName;
1615184475
```

- 4. 指定格式日期转 UNIX 时间戳函数：unix_timestamp

```
语法: unix_timestamp(string date, string pattern)
返回值: bigint
说明: 转换 pattern 格式的日期到 UNIX 时间戳。如果转化失败，则返回 0。
hive>  select unix_timestamp('2021-03-08 14:21:15','yyyyMMdd HH:mm:ss') from tableN
ame;
1615184475
```

-

5. 日期时间转日期函数：to_date

```
语法：to_date(string timestamp)
返回值：string
说明：返回日期时间字段中的日期部分。
hive> select to_date('2021-03-28 14:03:01') from tableName;
2021-03-28
```

- 6. 日期转年函数：year

```
语法：year(string date)
返回值：int
说明：返回日期中的年。
hive> select year('2021-03-28 10:03:01') from tableName;
2021
hive> select year('2021-03-28') from tableName;
2021
```

- 7. 日期转月函数：month

```
语法：month (string date)
返回值：int
说明：返回日期中的月份。
hive> select month('2020-12-28 12:03:01') from tableName;
12
hive> select month('2021-03-08') from tableName;
8
```

- 8. 日期转天函数：day

```
语法：day (string date)
返回值：int
说明：返回日期中的天。
hive> select day('2020-12-08 10:03:01') from tableName;
8
hive> select day('2020-12-24') from tableName;
24
```

- 9. 日期转小时函数：hour

```
语法: hour (string date)
返回值: int
说明: 返回日期中的小时。
hive> select hour('2020-12-08 10:03:01') from tableName;
10
```

- 10.日期转分钟函数: minute

```
语法: minute (string date)
返回值: int
说明: 返回日期中的分钟。
hive> select minute('2020-12-08 10:03:01') from tableName;
3
```

- 11.日期转秒函数: second

```
语法: second (string date)
返回值: int
说明: 返回日期中的秒。
hive> select second('2020-12-08 10:03:01') from tableName;
1
```

- 12.日期转周函数: weekofyear

```
语法: weekofyear (string date)
返回值: int
说明: 返回日期在当前的周数。
hive> select weekofyear('2020-12-08 10:03:01') from tableName;
49
```

- 13.日期比较函数: datediff

```
语法: datediff(string enddate, string startdate)
返回值: int
说明: 返回结束日期减去开始日期的天数。
hive> select datediff('2020-12-08','2012-05-09') from tableName;
213
```

- 14.日期增加函数: date_add

```
语法: date_add(string startdate, int days)
返回值: string
说明: 返回开始日期 startdate 增加 days 天后的日期。
hive> select date_add('2020-12-08',10) from tableName;
2020-12-18
```

- 15.日期减少函数：date_sub

```
语法: date_sub (string startdate, int days)
返回值: string
说明: 返回开始日期 startdate 减少 days 天后的日期。
hive> select date_sub('2020-12-08',10) from tableName;
2020-11-28
```

字符串函数

- 1. 字符串长度函数：length

```
语法: length(string A)
返回值: int
说明：返回字符串 A 的长度
hive> select length('abcedfg') from tableName;
7
```

- 2. 字符串反转函数：reverse

```
语法: reverse(string A)
返回值: string
说明：返回字符串 A 的反转结果
hive> select reverse('abcedfg') from tableName;
gfdecba
```

- 3. 字符串连接函数：concat

```
语法: concat(string A, string B…)
返回值: string
说明：返回输入字符串连接后的结果，支持任意个输入字符串
hive> select concat('abc','def’,'gh')from tableName;
abcdefgh
```

- 4. 带分隔符字符串连接函数：concat_ws

```
语法: concat_ws(string SEP, string A, string B…)
返回值: string
说明：返回输入字符串连接后的结果，SEP 表示各个字符串间的分隔符
hive> select concat_ws(',','abc','def','gh')from tableName;
abc,def,gh
```

- 5. 字符串截取函数：substr,substring

```
语法: substr(string A, int start),substring(string A, int start)
返回值: string
说明：返回字符串 A 从 start 位置到结尾的字符串
hive> select substr('abcde',3) from tableName;
cde
hive> select substring('abcde',3) from tableName;
cde
hive> select substr('abcde',-1) from tableName; （和 ORACLE 相同）
e
```

- 6. 字符串截取函数：substr,substring

```
语法: substr(string A, int start, int len),substring(string A, int start, int len)
返回值: string
说明：返回字符串 A 从 start 位置开始，长度为 len 的字符串
hive> select substr('abcde',3,2) from tableName;
cd
hive> select substring('abcde',3,2) from tableName;
cd
hive>select substring('abcde',-2,2) from tableName;
de
```

- 7. 字符串转大写函数：upper,ucase

```
语法: upper(string A) ucase(string A)
返回值: string
说明：返回字符串 A 的大写格式
hive> select upper('abSEd') from tableName;
ABSED
hive> select ucase('abSEd') from tableName;
ABSED
```

- 8. 字符串转小写函数：lower,lcase

```
语法: lower(string A) lcase(string A)
返回值: string
说明：返回字符串 A 的小写格式
hive> select lower('abSEd') from tableName;
absed
hive> select lcase('abSEd') from tableName;
absed
```

- 9. 去空格函数：trim

```
语法: trim(string A)
返回值: string
说明：去除字符串两边的空格
hive> select trim(' abc ') from tableName;
abc
```

- 10.左边去空格函数：ltrim

```
语法: ltrim(string A)
返回值: string
说明：去除字符串左边的空格
hive> select ltrim(' abc ') from tableName;
abc
```

- 11.右边去空格函数：rtrim

```
语法: rtrim(string A)
返回值: string
说明：去除字符串右边的空格
hive> select rtrim(' abc ') from tableName;
abc
```

- 12.正则表达式替换函数：regexp_replace

```
语法: regexp_replace(string A, string B, string C)
返回值: string
说明：将字符串 A 中的符合 java 正则表达式 B 的部分替换为 C。注意，在有些情况下要使用转义字符,类似 oracle 中的 regexp_replace 函数。
```

```
hive> select regexp_replace('foobar', 'oo|ar', '') from tableName;
fb
```

- 13. 正则表达式解析函数：regexp_extract

```
语法: regexp_extract(string subject, string pattern, int index)
返回值: string
说明：将字符串 subject 按照 pattern 正则表达式的规则拆分，返回 index 指定的字符。
hive> select regexp_extract('foothebar', 'foo(.*?)(bar)', 1) from tableName;
the
hive> select regexp_extract('foothebar', 'foo(.*?)(bar)', 2) from tableName;
bar
hive> select regexp_extract('foothebar', 'foo(.*?)(bar)', 0) from tableName;
foothebar
strong>注意，在有些情况下要使用转义字符，下面的等号要用双竖线转义，这是 java 正则表达式的规则。
select data_field,
regexp_extract(data_field,'.*?bgStart\\=([^&]+)',1) as aaa,
regexp_extract(data_field,'.*?contentLoaded_headStart\\=([^&]+)',1) as bbb,
regexp_extract(data_field,'.*?AppLoad2Req\\=([^&]+)',1) as ccc
from pt_nginx_loginlog_st
where pt = '2021-03-28' limit 2;
```

- 14. URL 解析函数：parse_url

```
语法: parse_url(string urlString, string partToExtract [, string keyToExtract])
返回值: string
说明：返回 URL 中指定的部分。partToExtract 的有效值为：
HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, and USERINFO.
hive> select parse_url
('https://www.tableName.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST')
from tableName;
www.tableName.com
hive> select parse_url
('https://www.tableName.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1')
 from tableName;
v1
```

- 15. json 解析函数：get_json_object

```
语法: get_json_object(string json_string, string path)
返回值: string
说明：解析 json 的字符串 json_string,返回 path 指定的内容。如果输入的 json 字符串无效，那么返
```

回 NULL。

```
hive> select  get_json_object('{"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"}','$.owner') from tableName;
```

- 16. 空格字符串函数：space

```
语法: space(int n)
返回值: string
说明：返回长度为 n 的字符串
hive> select space(10) from tableName;
hive> select length(space(10)) from tableName;
10
```

- 17. 重复字符串函数：repeat

```
语法: repeat(string str, int n)
返回值: string
说明：返回重复 n 次后的 str 字符串
hive> select repeat('abc',5) from tableName;
abcabcabcabcabc
```

- 18. 首字符 ascii 函数：ascii

```
语法: ascii(string str)
返回值: int
说明：返回字符串 str 第一个字符的 ascii 码
hive> select ascii('abcde') from tableName;
97
```

- 19. 左补足函数：lpad

```
语法: lpad(string str, int len, string pad)
返回值: string
说明：将 str 进行用 pad 进行左补足到 len 位
hive> select lpad('abc',10,'td') from tableName;
tdtdtdtabc
注意：与 GP，ORACLE 不同，pad 不能默认
```

- 20. 右补足函数：rpad

```
语法: rpad(string str, int len, string pad)
返回值: string
说明：将 str 进行用 pad 进行右补足到 len 位
hive> select rpad('abc',10,'td') from tableName;
abctdtdtdt
```

- 21.分割字符串函数: split

```
语法: split(string str, string pat)
返回值: array
说明: 按照 pat 字符串分割 str，会返回分割后的字符串数组
hive> select split('abtcdtef','t') from tableName;
["ab","cd","ef"]
```

- 22.集合查找函数: find_in_set

```
语法: find_in_set(string str, string strList)
返回值: int
说明: 返回 str 在 strlist 第一次出现的位置，strlist 是用逗号分割的字符串。如果没有找该 str 字符，则返回 0
hive> select find_in_set('ab','ef,ab,de') from tableName;
2
hive> select find_in_set('at','ef,ab,de') from tableName;
0
```

复合类型构建操作

- Map 类型构建: map

```
语法: map (key1, value1, key2, value2, …)
说明：根据输入的 key 和 value 对构建 map 类型
hive> Create table mapTable as select map('100','tom','200','mary') as t from table
Name;
hive> describe mapTable;
t       map<string ,string>
hive> select t from tableName;
{"100":"tom","200":"mary"}
```

- 2. Struct 类型构建: struct

```
语法: struct(val1, val2, val3, …)
说明：根据输入的参数构建结构体 struct 类型
hive> create table struct_table as select struct('tom','mary','tim') as t from tabl
eName;
hive> describe struct_table;
t       struct<col1:string ,col2:string,col3:string>
hive> select t from tableName;
{"col1":"tom","col2":"mary","col3":"tim"}
```

- 3. array 类型构建：array

```
语法: array(val1, val2, …)
说明：根据输入的参数构建数组 array 类型
hive> create table arr_table as select array("tom","mary","tim") as t from tableNam
e;
hive> describe tableName;
t       array<string>
hive> select t from tableName;
["tom","mary","tim"]
```

复杂类型访问操作

- 1. array 类型访问：A[n]

```
语法: A[n]
操作类型：A 为 array 类型，n 为 int 类型
说明：返回数组 A 中的第 n 个变量值。数组的起始下标为 0。比如，A 是个值为['foo', 'bar']的数组类
型，那么 A[0]将返回'foo',而 A[1]将返回'bar'
hive> create table arr_table2 as select array("tom","mary","tim") as t
 from tableName;
hive> select t[0],t[1] from arr_table2;
tom     mary    tim
```

- 2. map 类型访问：M[key]

```
语法: M[key]
操作类型：M 为 map 类型，key 为 map 中的 key 值
说明：返回 map 类型 M 中，key 值为指定值的 value 值。比如，M 是值为
{'f' -> 'foo', 'b' -> 'bar', 'all' -> 'foobar'}的 map 类型，那么 M['all']将会返回
'foobar'
hive> Create table map_table2 as select map('100','tom','200','mary') as t from tab
```

```
leName;
hive> select t['200'],t['100'] from map_table2;
mary    tom
```

- 3. struct 类型访问：S.x

```
语法：S.x
操作类型：S 为 struct 类型
说明：返回结构体 S 中的 x 字段。比如，对于结构体 struct foobar {int foo, int bar}，foobar.foo
返回结构体中的 foo 字段
hive> create table str_table2 as select struct('tom','mary','tim') as t from tableN
ame;
hive> describe tableName;
t       struct<col1:string ,col2:string,col3:string>
hive> select t.col1,t.col3 from str_table2;
tom     tim
```

复杂类型长度统计函数

- 1. Map 类型长度函数：size(Map<k .V>)

```
语法：size(Map<k .V>)
返回值：int
说明：返回 map 类型的长度
hive> select size(t) from map_table2;
2
```

- 2. array 类型长度函数：size(Array)

```
语法：size(Array<T>)
返回值：int
说明：返回 array 类型的长度
hive> select size(t) from arr_table2;
4
```

- 3. 类型转换函数 ***

```
类型转换函数：cast
语法：cast(expr as <type>)
返回值：Expected "=" to follow "type"
```

```
说明: 返回转换后的数据类型
hive> select cast('1' as bigint) from tableName;
1
```

hive 当中的 lateral view 与 explode 以及 reflect

使用 explode 函数将 hive 表中的 Map 和 Array 字段数据进行拆分

lateral view 用于和 split、explode 等 UDTF 一起使用的，能将一行数据拆分成多行数据，在此基础上可以对拆分的数据进行聚合，lateral view 首先为原始表的每行调用 UDTF，UDTF 会把一行拆分成一行或者多行，lateral view 在把结果组合，产生一个支持别名表的虚拟表。

其中 explode 还可以用于将 hive 一列中复杂的 array 或者 map 结构拆分成多行

需求：现在有数据格式如下

```
zhangsan child1,child2,child3,child4 k1:v1,k2:v2

lisi child5,child6,child7,child8 k3:v3,k4:v4
```

字段之间使用\t 分割，需求将所有的 child 进行拆开成为一列

```
+----------+--+
| mychild  |
+----------+--+
| child1   |
| child2   |
| child3   |
| child4   |
| child5   |
| child6   |
| child7   |
| child8   |
+----------+--+
```

将 map 的 key 和 value 也进行拆开，成为如下结果

```
+-----------+-------------+--+
| mymapkey  | mymapvalue  |
+-----------+-------------+--+
| k1        | v1          |
| k2        | v2          |
| k3        | v3          |
| k4        | v4          |
+-----------+-------------+--+
```

- 1. 创建 hive 数据库

```
创建 hive 数据库
hive (default)> create database hive_explode;
hive (default)> use hive_explode;
```

- 2. 创建 hive 表，然后使用 explode 拆分 map 和 array

```
hive (hive_explode)> create  table t3(name string,children array<string>,address Ma
p<string,string>) row format delimited fields terminated by '\t'  collection items
terminated by ',' map keys terminated by ':' stored as textFile;
```

- 3. 加载数据

```
node03 执行以下命令创建表数据文件
 mkdir -p /export/servers/hivedatas/
 cd /export/servers/hivedatas/
 vim maparray
内容如下:
zhangsan child1,child2,child3,child4 k1:v1,k2:v2
lisi child5,child6,child7,child8 k3:v3,k4:v4

hive 表当中加载数据
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/maparray' in
to table t3;
```

- 4. 使用 explode 将 hive 当中数据拆开

```
将 array 当中的数据拆分开
hive (hive_explode)> SELECT explode(children) AS myChild FROM t3;

将 map 当中的数据拆分开

hive (hive_explode)> SELECT explode(address) AS (myMapKey, myMapValue) FROM t3;
```

使用 explode 拆分 json 字符串

需求：需求：现在有一些数据格式如下：

```
a:shandong,b:beijing,c:hebei|1,2,3,4,5,6,7,8,9|[{"source":"7fresh","monthSales":490
0,"userCount":1900,"score":"9.9"},{"source":"jd","monthSales":2090,"userCount":7898
```

```
1,"score":"9.8"},{"source":"jdmart","monthSales":6987,"userCount":1600,"score":"9.0
"}]
```

其中字段与字段之间的分隔符是 |

我们要解析得到所有的 monthSales 对应的值为以下这一列（行转列）

4900

2090

6987

- 1. 创建 hive 表

```
hive (hive_explode)> create table explode_lateral_view
                   > (`area` string,
                   > `goods_id` string,
                   > `sale_info` string)
                   > ROW FORMAT DELIMITED
                   > FIELDS TERMINATED BY '|'
                   > STORED AS textfile;
```

- 2. 准备数据并加载数据

```
准备数据如下
cd /export/servers/hivedatas
vim explode_json
```

```
a:shandong,b:beijing,c:hebei|1,2,3,4,5,6,7,8,9|[{"source":"7fresh","monthSales":490
0,"userCount":1900,"score":"9.9"},{"source":"jd","monthSales":2090,"userCount":7898
1,"score":"9.8"},{"source":"jdmart","monthSales":6987,"userCount":1600,"score":"9.0
"}]
```

```
加载数据到 hive 表当中去
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/explode_json
' overwrite into table explode_lateral_view;
```

- 3. 使用 explode 拆分 Array

```
hive (hive_explode)> select explode(split(goods_id,',')) as goods_id from explode_l
ateral_view;
```

- 4. 使用 explode 拆解 Map

```
hive (hive_explode)> select explode(split(area,',')) as area from explode_lateral_v
iew;
```

- 5. 拆解 json 字段

```
hive (hive_explode)> select explode(split(regexp_replace(regexp_replace(sale_info,'
\\[\\{',''),'}]',''),'},\\{')) as  sale_info from explode_lateral_view;
```

然后我们想用 get_json_object 来获取 key 为 monthSales 的数据：

```
hive (hive_explode)> select get_json_object(explode(split(regexp_replace(regexp_rep
lace(sale_info,'\\[\\{',''),'}]',''),'},\\{')),'$.monthSales') as  sale_info from e
xplode_lateral_view;
```

然后挂了

```
FAILED: SemanticException [Error 10081]: UDTF's are not supported outside the SELEC
T clause, nor nested in expressions
```

UDTF explode 不能写在别的函数内

如果你这么写，想查两个字段，

```
select explode(split(area,',')) as area,good_id from explode_lateral_view;
```

会报错

```
FAILED: SemanticException 1:40 Only a single expression in the SELECT clause is sup
ported with UDTF's. Error encountered near token 'good_id'
```

使用 UDTF 的时候，只支持一个字段，这时候就需要 LATERAL VIEW 出场了

配合 LATERAL VIEW 使用

配合 lateral view 查询多个字段

```
hive (hive_explode)> select goods_id2,sale_info from explode_lateral_view LATERAL V
IEW explode(split(goods_id,','))goods as goods_id2;
```

其中 LATERAL VIEW explode(split(goods_id,','))goods 相当于一个虚拟表，与原表 explode_lateral_view 笛卡尔积关联

也可以多重使用

```
hive (hive_explode)> select goods_id2,sale_info,area2
                   from explode_lateral_view
                   LATERAL VIEW explode(split(goods_id,','))goods as goods_id2
                   LATERAL VIEW explode(split(area,','))area as area2;也是三个表笛卡
尔积的结果
```

最终，我们可以通过下面的句子，把这个 json 格式的一行数据，完全转换成二维表的方式展现

```
hive (hive_explode)> select get_json_object(concat('{',sale_info_1,'}'),'$.source') as source,get_json_object(concat('{',sale_info_1,'}'),'$.monthSales') as monthSales,get_json_object(concat('{',sale_info_1,'}'),'$.userCount') as monthSales,get_json_object(concat('{',sale_info_1,'}'),'$.score') as monthSales from explode_lateral_view LATERAL VIEW explode(split(regexp_replace(regexp_replace(sale_info,'\\[\\{',''),'}]',''),'},\\{'))sale_info as sale_info_1;
```

总结：

Lateral View 通常和 UDTF 一起出现，为了解决 UDTF 不允许在 select 字段的问题。 Multiple Lateral View 可以实现类似笛卡尔乘积。 Outer 关键字可以把不输出的 UDTF 的空结果，输出成 NULL，防止丢失数据。

行转列

相关参数说明：

CONCAT(string A/col, string B/col…)：返回输入字符串连接后的结果，支持任意个输入字符串；

CONCAT_WS(separator, str1, str2,...)：它是一个特殊形式的 CONCAT()。第一个参数剩余参数间的分隔符。分隔符可以是与剩余参数一样的字符串。如果分隔符是 NULL，返回值也将为 NULL。这个函数会跳过分隔符参数后的任何 NULL 和空字符串。分隔符将被加到被连接的字符串之间；

COLLECT_SET(col)：函数只接受基本数据类型，它的主要作用是将某字段的值进行去重汇总，产生 array 类型字段。

数据准备：

name	constellation	blood_type
孙悟空	白羊座	A
老王	射手座	A
宋宋	白羊座	B
猪八戒	白羊座	A
凤姐	射手座	A

需求：把星座和血型一样的人归类到一起。结果如下：

```
射手座,A            老王|凤姐
白羊座,A            孙悟空|猪八戒
白羊座,B            宋宋
```

实现步骤：

- 1. 创建本地 constellation.txt，导入数据

```
node03 服务器执行以下命令创建文件，注意数据使用\t 进行分割
cd /export/servers/hivedatas
vim constellation.txt

数据如下：
孙悟空 白羊座 A
老王 射手座 A
宋宋 白羊座 B
猪八戒 白羊座 A
凤姐 射手座 A
```

- 2. 创建 hive 表并导入数据

```
创建 hive 表并加载数据
hive (hive_explode)> create table person_info(
                    name string,
                    constellation string,
                    blood_type string)
                    row format delimited fields terminated by "\t";

加载数据
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/constellatio
n.txt' into table person_info;
```

- 3. 按需求查询数据

```
hive (hive_explode)> select
                        t1.base,
                        concat_ws('|', collect_set(t1.name)) name
                    from
                        (select
                            name,
                            concat(constellation, "," , blood_type) base
                        from
                            person_info) t1
```

```
                    group by
                        t1.base;
```

列转行

所需函数：

EXPLODE(col)：将 hive 一列中复杂的 array 或者 map 结构拆分成多行。

LATERAL VIEW

用法：LATERAL VIEW udtf(expression) tableAlias AS columnAlias

解释：用于和 split, explode 等 UDTF 一起使用，它能够将一列数据拆成多行数据，在此基础上可以对拆分后的数据进行聚合。

数据准备：

```
cd /export/servers/hivedatas
vim movie.txt
文件内容如下： 数据字段之间使用\t 进行分割
《疑犯追踪》 悬疑,动作,科幻,剧情
《Lie to me》 悬疑,警匪,动作,心理,剧情
《战狼 2》 战争,动作,灾难
```

需求：将电影分类中的数组数据展开。结果如下：

```
《疑犯追踪》 悬疑
《疑犯追踪》 动作
《疑犯追踪》 科幻
《疑犯追踪》 剧情
《Lie to me》 悬疑
《Lie to me》 警匪
《Lie to me》 动作
《Lie to me》 心理
《Lie to me》 剧情
《战狼 2》 战争
《战狼 2》 动作
《战狼 2》 灾难
```

实现步骤：

- 1. 创建 hive 表

```
create table movie_info(
    movie string,
    category array<string>)
```

```
row format delimited fields terminated by "\t"
collection items terminated by ",";
```

- 2. 加载数据

```
load data local inpath "/export/servers/hivedatas/movie.txt" into table movie_info;
```

- 3. 按需求查询数据

```
select
    movie,
    category_name
from
    movie_info lateral view explode(category) table_tmp as category_name;
```

reflect 函数

reflect 函数可以支持在 sql 中调用 java 中的自带函数，秒杀一切 udf 函数。

需求 1：使用 java.lang.Math 当中的 Max 求两列中最大值

实现步骤：

- 1. 创建 hive 表

```
create table test_udf(col1 int,col2 int) row format delimited fields terminated by
',';
```

- 2. 准备数据并加载数据

```
cd /export/servers/hivedatas
vim test_udf
```

```
文件内容如下：
1,2
4,3
6,4
7,5
5,6
```

- 3. 加载数据

```
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/test_udf' ov
erwrite into table test_udf;
```

- 4. 使用 java.lang.Math 当中的 Max 求两列当中的最大值

```
hive (hive_explode)> select reflect("java.lang.Math","max",col1,col2) from test_udf
;
```

需求 2：文件中不同的记录来执行不同的 java 的内置函数

实现步骤：

- 1. 创建 hive 表

```
hive (hive_explode)> create table test_udf2(class_name string,method_name string,co
l1 int , col2 int) row format delimited fields terminated by ',';
```

- 2. 准备数据

```
cd /export/servers/hivedatas
vim test_udf2
```

```
文件内容如下：
java.lang.Math,min,1,2
java.lang.Math,max,2,3
```

- 3. 加载数据

```
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/test_udf2' o
verwrite into table test_udf2;
```

- 4. 执行查询

```
hive (hive_explode)> select reflect(class_name,method_name,col1,col2) from test_udf
2;
```

需求 3：判断是否为数字

实现方式：

使用 apache commons 中的函数，commons 下的 jar 已经包含在 hadoop 的 classpath 中，所以可以直接使用。

```
select reflect("org.apache.commons.lang.math.NumberUtils","isNumber","123")
```

Hive 窗口函数

窗口函数最重要的关键字是 partition by 和 order by

具体语法如下：XXX over (partition by xxx order by xxx)

特别注意：over()里面的 partition by 和 order by 都不是必选的，over()里面可以只有 partition by，也可以只有 order by，也可以两个都没有，大家需根据需求灵活运用。

窗口函数我划分了几个大类，我们一类一类的讲解。

1. SUM、AVG、MIN、MAX

讲解这几个窗口函数前，先创建一个表，以实际例子讲解大家更容易理解。

首先创建用户访问页面表：user_pv

```
create table user_pv(
cookieid string,  -- 用户登录的cookie，即用户标识
createtime string, -- 日期
pv int -- 页面访问量
);
```

给上面这个表加上如下数据：

```
cookie1,2021-05-10,1
cookie1,2021-05-11,5
cookie1,2021-05-12,7
cookie1,2021-05-13,3
cookie1,2021-05-14,2
cookie1,2021-05-15,4
cookie1,2021-05-16,4
```

- SUM()使用

执行如下查询语句：

```
select cookieid,createtime,pv,
sum(pv) over(partition by cookieid order by createtime) as pv1
from user_pv;
```

结果如下：（因命令行原因，下图字段名和值是错位的，请注意辨别！）

```
cookieid        createtime      pv      pv1
cookie1 2021-05-10      1       1
cookie1 2021-05-11      5       6
cookie1 2021-05-12      7       13
cookie1 2021-05-13      3       16
cookie1 2021-05-14      2       18
cookie1 2021-05-15      4       22
cookie1 2021-05-16      4       26
```

执行如下查询语句：

```
select cookieid,createtime,pv,
sum(pv) over(partition by cookieid ) as pv1
from user_pv;
```

结果如下：

```
cookieid        createtime      pv      pv1
cookie1 2021-05-16      4       26
cookie1 2021-05-15      4       26
cookie1 2021-05-14      2       26
cookie1 2021-05-13      3       26
cookie1 2021-05-12      7       26
cookie1 2021-05-11      5       26
cookie1 2021-05-10      1       26
```

第一条 SQL 的 over()里面加 order by ，第二条 SQL 没加 order by ，结果差别很大

所以要注意了：

- over()里面加 order by 表示：分组内从起点到当前行的 pv 累积，如，11 号的 pv1=10 号的 pv+11 号的 pv，12 号=10 号+11 号+12 号；
- over()里面不加 order by 表示：将分组内所有值累加。

AVG，MIN，MAX，和 SUM 用法一样，这里就不展开讲了，但是要注意 AVG，MIN，MAX 的 over()里面加不加 order by 也和 SUM 一样，如 AVG 求平均值，如果加上 order by，表示分组内从起点到当前行的平局值，不是全部的平局值。MIN，MAX 同理。

2. ROW_NUMBER、RANK、DENSE_RANK、NTILE

还是用上述的用户登录日志表：user_pv，里面的数据换成如下所示：

```
cookie1,2021-05-10,1
cookie1,2021-05-11,5
cookie1,2021-05-12,7
cookie1,2021-05-13,3
cookie1,2021-05-14,2
cookie1,2021-05-15,4
```

```
cookie1,2021-05-16,4
cookie2,2021-05-10,2
cookie2,2021-05-11,3
cookie2,2021-05-12,5
cookie2,2021-05-13,6
cookie2,2021-05-14,3
cookie2,2021-05-15,9
cookie2,2021-05-16,7
```

- ROW_NUMBER()使用：

ROW_NUMBER()从 1 开始，按照顺序，生成分组内记录的序列。

```
SELECT
cookieid,
createtime,
pv,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn
FROM user_pv;
```

结果如下：

cookieid	createtime	pv	rn
cookie1	2021-05-12	7	1
cookie1	2021-05-11	5	2
cookie1	2021-05-16	4	3
cookie1	2021-05-15	4	4
cookie1	2021-05-13	3	5
cookie1	2021-05-14	2	6
cookie1	2021-05-10	1	7
cookie2	2021-05-15	9	1
cookie2	2021-05-16	7	2
cookie2	2021-05-13	6	3
cookie2	2021-05-12	5	4
cookie2	2021-05-11	3	5
cookie2	2021-05-14	3	6
cookie2	2021-05-10	2	7

- RANK 和 DENSE_RANK 使用：

RANK() 生成数据项在分组中的排名，排名相等会在名次中留下空位。
DENSE_RANK()生成数据项在分组中的排名，排名相等会在名次中不会留下空位。

```
SELECT
cookieid,
createtime,
pv,
RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn1,
```

```
DENSE_RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn2,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv DESC) AS rn3
FROM user_pv
WHERE cookieid = 'cookie1';
```

结果如下：

```
cookieid        createtime      pv      rn1     rn2     rn3
cookie1 2021-05-12      7       1       1       1
cookie1 2021-05-11      5       2       2       2
cookie1 2021-05-16      4       3       3       3
cookie1 2021-05-15      4       3       3       4
cookie1 2021-05-13      3       5       4       5
cookie1 2021-05-14      2       6       5       6
cookie1 2021-05-10      1       7       6       7
```

- NTILE 的使用：

有时会有这样的需求:如果数据排序后分为三部分，业务人员只关心其中的一部分，如何将这中间的三分之一数据拿出来呢?NTILE 函数即可以满足。

ntile 可以看成是：把有序的数据集合平均分配到指定的数量（num）个桶中，将桶号分配给每一行。如果不能平均分配，则优先分配较小编号的桶，并且各个桶中能放的行数最多相差 1。

然后可以根据桶号，选取前或后 n 分之几的数据。数据会完整展示出来，只是给相应的数据打标签;具体要取几分之几的数据,需要再嵌套一层根据标签取出。

```
SELECT
cookieid,
createtime,
pv,
NTILE(2) OVER(PARTITION BY cookieid ORDER BY createtime) AS rn1,
NTILE(3) OVER(PARTITION BY cookieid ORDER BY createtime) AS rn2,
NTILE(4) OVER(ORDER BY createtime) AS rn3
FROM user_pv
ORDER BY cookieid,createtime;
```

结果如下：

```
cookieid        createtime      pv      rn1     rn2     rn3
cookie1 2021-05-10      1       1       1       1
cookie1 2021-05-11      5       1       1       1
cookie1 2021-05-12      7       1       1       2
cookie1 2021-05-13      3       1       2       2
cookie1 2021-05-14      2       2       2       3
cookie1 2021-05-15      4       2       3       4
cookie1 2021-05-16      4       2       3       4
cookie2 2021-05-10      2       1       1       1
cookie2 2021-05-11      3       1       1       1
cookie2 2021-05-12      5       1       1       2
cookie2 2021-05-13      6       1       2       2
cookie2 2021-05-14      3       2       2       3
cookie2 2021-05-15      9       2       3       3
cookie2 2021-05-16      7       2       3       4
```

3. LAG、LEAD、FIRST_VALUE、LAST_VALUE

讲解这几个窗口函数时还是以实例讲解，首先创建用户访问页面表：user_url

```
CREATE TABLE user_url (
cookieid string,
createtime string,   --页面访问时间
url string        --被访问页面
);
```

表中加入如下数据：

```
cookie1,2021-06-10 10:00:02,url2
cookie1,2021-06-10 10:00:00,url1
cookie1,2021-06-10 10:03:04,1url3
cookie1,2021-06-10 10:50:05,url6
cookie1,2021-06-10 11:00:00,url7
cookie1,2021-06-10 10:10:00,url4
cookie1,2021-06-10 10:50:01,url5
cookie2,2021-06-10 10:00:02,url22
cookie2,2021-06-10 10:00:00,url11
cookie2,2021-06-10 10:03:04,1url33
cookie2,2021-06-10 10:50:05,url66
cookie2,2021-06-10 11:00:00,url77
cookie2,2021-06-10 10:10:00,url44
cookie2,2021-06-10 10:50:01,url55
```

- LAG 的使用：

LAG(col,n,DEFAULT) 用于统计窗口内往上第 n 行值。

第一个参数为列名，第二个参数为往上第 n 行（可选，默认为 1），第三个参数为默认值（当往上第 n 行为 NULL 时候，取默认值，如不指定，则为 NULL）

```
SELECT cookieid,
createtime,
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LAG(createtime,1,'1970-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY createt
ime) AS last_1_time,
LAG(createtime,2) OVER(PARTITION BY cookieid ORDER BY createtime) AS last_2_time
FROM user_url;
```

结果如下：

```
cookieid        createtime      url     rn      last_1_time     last_2_time
cookie1 2021-06-10 10:00:00     url1    1       1970-01-01 00:00:00     NULL
cookie1 2021-06-10 10:00:02     url2    2       2021-06-10 10:00:00     NULL
cookie1 2021-06-10 10:03:04     1url3   3       2021-06-10 10:00:02     2021-06-10 10:00:00
cookie1 2021-06-10 10:10:00     url4    4       2021-06-10 10:03:04     2021-06-10 10:00:02
cookie1 2021-06-10 10:50:01     url5    5       2021-06-10 10:10:00     2021-06-10 10:03:04
cookie1 2021-06-10 10:50:05     url6    6       2021-06-10 10:50:01     2021-06-10 10:10:00
cookie1 2021-06-10 11:00:00     url7    7       2021-06-10 10:50:05     2021-06-10 10:50:01
cookie2 2021-06-10 10:00:00     url11   1       1970-01-01 00:00:00     NULL
cookie2 2021-06-10 10:00:02     url22   2       2021-06-10 10:00:00     NULL
cookie2 2021-06-10 10:03:04     1url33  3       2021-06-10 10:00:02     2021-06-10 10:00:00
cookie2 2021-06-10 10:10:00     url44   4       2021-06-10 10:03:04     2021-06-10 10:00:02
cookie2 2021-06-10 10:50:01     url55   5       2021-06-10 10:10:00     2021-06-10 10:03:04
cookie2 2021-06-10 10:50:05     url66   6       2021-06-10 10:50:01     2021-06-10 10:10:00
cookie2 2021-06-10 11:00:00     url77   7       2021-06-10 10:50:05     2021-06-10 10:50:01
```

解释：

```
last_1_time: 指定了往上第 1 行的值，default 为'1970-01-01 00:00:00'
                cookie1 第一行，往上 1 行为 NULL,因此取默认值 1970-01-01 00:00:00
                cookie1 第三行，往上 1 行值为第二行值，2021-06-10 10:00:02
                cookie1 第六行，往上 1 行值为第五行值，2021-06-10 10:50:01
last_2_time: 指定了往上第 2 行的值，为指定默认值
       cookie1 第一行，往上 2 行为 NULL
       cookie1 第二行，往上 2 行为 NULL
       cookie1 第四行，往上 2 行为第二行值，2021-06-10 10:00:02
       cookie1 第七行，往上 2 行为第五行值，2021-06-10 10:50:01
```

- LEAD 的使用：

与 LAG 相反

LEAD(col,n,DEFAULT) 用于统计窗口内往下第 n 行值。

第一个参数为列名，第二个参数为往下第 n 行（可选，默认为 1），第三个参数为默认值（当往下第 n 行为 NULL 时候，取默认值，如不指定，则为 NULL）

```
SELECT cookieid,
createtime,
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LEAD(createtime,1,'1970-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY create
time) AS next_1_time,
LEAD(createtime,2) OVER(PARTITION BY cookieid ORDER BY createtime) AS next_2_time
FROM user_url;
```

结果如下：

```
cookieid        createtime      url     rn      next_1_time     next_2_time
cookie1 2021-06-10 10:00:00     url1    1       2021-06-10 10:00:02     2021-06-10 10:03:04
cookie1 2021-06-10 10:00:02     url2    2       2021-06-10 10:03:04     2021-06-10 10:10:00
cookie1 2021-06-10 10:03:04     1url3   3       2021-06-10 10:10:00     2021-06-10 10:50:01
cookie1 2021-06-10 10:10:00     url4    4       2021-06-10 10:50:01     2021-06-10 10:50:05
cookie1 2021-06-10 10:50:01     url5    5       2021-06-10 10:50:05     2021-06-10 11:00:00
cookie1 2021-06-10 10:50:05     url6    6       2021-06-10 11:00:00     NULL
cookie1 2021-06-10 11:00:00     url7    7       1970-01-01 00:00:00     NULL
cookie2 2021-06-10 10:00:00     url11   1       2021-06-10 10:00:02     2021-06-10 10:03:04
cookie2 2021-06-10 10:00:02     url22   2       2021-06-10 10:03:04     2021-06-10 10:10:00
cookie2 2021-06-10 10:03:04     1url33  3       2021-06-10 10:10:00     2021-06-10 10:50:01
cookie2 2021-06-10 10:10:00     url44   4       2021-06-10 10:50:01     2021-06-10 10:50:05
cookie2 2021-06-10 10:50:01     url55   5       2021-06-10 10:50:05     2021-06-10 11:00:00
cookie2 2021-06-10 10:50:05     url66   6       2021-06-10 11:00:00     NULL
cookie2 2021-06-10 11:00:00     url77   7       1970-01-01 00:00:00     NULL
```

- FIRST_VALUE 的使用：

取分组内排序后，截止到当前行，第一个值。

```
SELECT cookieid,
createtime,
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
FIRST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS first1
FROM user_url;
```

结果如下：

```
cookieid        createtime      url     rn      first1
cookie1 2021-06-10 10:00:00     url1    1       url1
cookie1 2021-06-10 10:00:02     url2    2       url1
cookie1 2021-06-10 10:03:04     1url3   3       url1
cookie1 2021-06-10 10:10:00     url4    4       url1
cookie1 2021-06-10 10:50:01     url5    5       url1
cookie1 2021-06-10 10:50:05     url6    6       url1
cookie1 2021-06-10 11:00:00     url7    7       url1
cookie2 2021-06-10 10:00:00     url11   1       url11
cookie2 2021-06-10 10:00:02     url22   2       url11
cookie2 2021-06-10 10:03:04     1url33  3       url11
cookie2 2021-06-10 10:10:00     url44   4       url11
cookie2 2021-06-10 10:50:01     url55   5       url11
cookie2 2021-06-10 10:50:05     url66   6       url11
cookie2 2021-06-10 11:00:00     url77   7       url11
```

- LAST_VALUE 的使用：

取分组内排序后，截止到当前行，最后一个值。

```
SELECT cookieid,
createtime,
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LAST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS last1
FROM user_url;
```

结果如下：

```
cookieid        createtime      url     rn      last1
cookie1 2021-06-10 10:00:00     url1    1       url1
cookie1 2021-06-10 10:00:02     url2    2       url2
cookie1 2021-06-10 10:03:04     1url3   3       1url3
cookie1 2021-06-10 10:10:00     url4    4       url4
cookie1 2021-06-10 10:50:01     url5    5       url5
cookie1 2021-06-10 10:50:05     url6    6       url6
cookie1 2021-06-10 11:00:00     url7    7       url7
cookie2 2021-06-10 10:00:00     url11   1       url11
cookie2 2021-06-10 10:00:02     url22   2       url22
cookie2 2021-06-10 10:03:04     1url33  3       1url33
cookie2 2021-06-10 10:10:00     url44   4       url44
cookie2 2021-06-10 10:50:01     url55   5       url55
cookie2 2021-06-10 10:50:05     url66   6       url66
cookie2 2021-06-10 11:00:00     url77   7       url77
```

如果想要取分组内排序后最后一个值，则需要变通一下：

```
SELECT cookieid,
createtime,
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LAST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS last1,
FIRST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime DESC) AS last2
FROM user_url
ORDER BY cookieid,createtime;
```

注意上述 SQL，使用的是 FIRST_VALUE 的倒序取出分组内排序最后一个值！

结果如下：

```
cookieid        createtime      url     rn      last1   last2
cookie1 2021-06-10 10:00:00     url1    1       url1    url7
cookie1 2021-06-10 10:00:02     url2    2       url2    url7
cookie1 2021-06-10 10:03:04     1url3   3       1url3   url7
cookie1 2021-06-10 10:10:00     url4    4       url4    url7
cookie1 2021-06-10 10:50:01     url5    5       url5    url7
cookie1 2021-06-10 10:50:05     url6    6       url6    url7
cookie1 2021-06-10 11:00:00     url7    7       url7    url7
cookie2 2021-06-10 10:00:00     url11   1       url11   url77
cookie2 2021-06-10 10:00:02     url22   2       url22   url77
cookie2 2021-06-10 10:03:04     1url33  3       1url33  url77
cookie2 2021-06-10 10:10:00     url44   4       url44   url77
cookie2 2021-06-10 10:50:01     url55   5       url55   url77
cookie2 2021-06-10 10:50:05     url66   6       url66   url77
cookie2 2021-06-10 11:00:00     url77   7       url77   url77
```

此处要特别注意 order by

如果不指定 ORDER BY，则进行排序混乱，会出现错误的结果

```
SELECT cookieid,
createtime,
url,
FIRST_VALUE(url) OVER(PARTITION BY cookieid) AS first2
FROM user_url;
```

结果如下：

```
cookieid        createtime      url     first2
cookie1 2021-06-10 10:00:02     url2    url2
cookie1 2021-06-10 10:50:01     url5    url2
cookie1 2021-06-10 10:10:00     url4    url2
cookie1 2021-06-10 11:00:00     url7    url2
cookie1 2021-06-10 10:50:05     url6    url2
cookie1 2021-06-10 10:03:04     1url3   url2
cookie1 2021-06-10 10:00:00     url1    url2
cookie2 2021-06-10 10:50:01     url55   url55
cookie2 2021-06-10 10:10:00     url44   url55
cookie2 2021-06-10 11:00:00     url77   url55
cookie2 2021-06-10 10:50:05     url66   url55
cookie2 2021-06-10 10:03:04     1url33  url55
cookie2 2021-06-10 10:00:00     url11   url55
cookie2 2021-06-10 10:00:02     url22   url55
```

上述 url2 和 url55 的 createtime 即不属于最靠前的时间也不属于最靠后的时间，所以结果是混乱的。

4. CUME_DIST

先创建一张员工薪水表：staff_salary

```
CREATE EXTERNAL TABLE staff_salary (
dept string,
userid string,
sal int
);
```

表中加入如下数据：

```
d1,user1,1000
d1,user2,2000
d1,user3,3000
d2,user4,4000
d2,user5,5000
```

- CUME_DIST 的使用：

此函数的结果和 order by 的排序顺序有关系。

CUME_DIST：小于等于当前值的行数/分组内总行数。 order 默认顺序 ： 正序

比如，统计小于等于当前薪水的人数，所占总人数的比例。

```
SELECT
dept,
userid,
sal,
CUME_DIST() OVER(ORDER BY sal) AS rn1,
CUME_DIST() OVER(PARTITION BY dept ORDER BY sal) AS rn2
FROM staff_salary;
```

结果如下：

dept	userid	sal	rn1	rn2
d1	user1	1000	0.2	0.3333333333333333
d1	user2	2000	0.4	0.6666666666666666
d1	user3	3000	0.6	1.0
d2	user4	4000	0.8	0.5
d2	user5	5000	1.0	1.0

解释：

```
rn1: 没有 partition,所有数据均为 1 组，总行数为 5，
     第一行：小于等于 1000 的行数为 1，因此，1/5=0.2
     第三行：小于等于 3000 的行数为 3，因此，3/5=0.6
rn2: 按照部门分组，dpet=d1 的行数为 3,
     第二行：小于等于 2000 的行数为 2，因此，2/3=0.6666666666666666
```

5. GROUPING SETS、GROUPING__ID、CUBE、ROLLUP

这几个分析函数通常用于 OLAP 中，不能累加，而且需要根据不同维度上钻和下钻的指标统计，比如，分小时、天、月的 UV 数。

还是先创建一个用户访问表：user_date

```
CREATE TABLE user_date (
month STRING,
day STRING,
cookieid STRING
);
```

表中加入如下数据：

```
2021-03,2021-03-10,cookie1
2021-03,2021-03-10,cookie5
2021-03,2021-03-12,cookie7
2021-04,2021-04-12,cookie3
```

```
2021-04,2021-04-13,cookie2
2021-04,2021-04-13,cookie4
2021-04,2021-04-16,cookie4
2021-03,2021-03-10,cookie2
2021-03,2021-03-10,cookie3
2021-04,2021-04-12,cookie5
2021-04,2021-04-13,cookie6
2021-04,2021-04-15,cookie3
2021-04,2021-04-15,cookie2
2021-04,2021-04-16,cookie1
```

- GROUPING SETS 的使用：

grouping sets 是一种将多个 group by 逻辑写在一个 sql 语句中的便利写法。等价于将不同维度的 GROUP BY 结果集进行 UNION ALL。

```
SELECT
month,
day,
COUNT(DISTINCT cookieid) AS uv,
GROUPING__ID
FROM user_date
GROUP BY month,day
GROUPING SETS (month,day)
ORDER BY GROUPING__ID;
```

注：上述 SQL 中的 GROUPING__ID，是个关键字，表示结果属于哪一个分组集合，根据 grouping sets 中的分组条件 month，day，1 是代表 month，2 是代表 day。结果如下：

```
month   day     uv      grouping__id
2021-04 NULL    6       1
2021-03 NULL    5       1
NULL    2021-04-16      2       2
NULL    2021-04-15      2       2
NULL    2021-04-13      3       2
NULL    2021-04-12      2       2
NULL    2021-03-12      1       2
NULL    2021-03-10      4       2
```

上述 SQL 等价于：

```
SELECT month,
NULL as day,
COUNT(DISTINCT cookieid) AS uv,
1 AS GROUPING__ID
FROM user_date
```

```
GROUP BY month

UNION ALL

SELECT NULL as month,
day,
COUNT(DISTINCT cookieid) AS uv,
2 AS GROUPING__ID
FROM user_date
GROUP BY day;
```

- CUBE 的使用：

根据 GROUP BY 的维度的所有组合进行聚合。

```
SELECT
month,
day,
COUNT(DISTINCT cookieid) AS uv,
GROUPING__ID
FROM user_date
GROUP BY month,day
WITH CUBE
ORDER BY GROUPING__ID;
```

结果如下：

```
month   day         uv      grouping__id
NULL    NULL        7       0
2021-03 NULL        5       1
2021-04 NULL        6       1
NULL    2021-04-16  2       2
NULL    2021-04-15  2       2
NULL    2021-04-13  3       2
NULL    2021-04-12  2       2
NULL    2021-03-12  1       2
NULL    2021-03-10  4       2
2021-04 2021-04-12  2       3
2021-04 2021-04-16  2       3
2021-03 2021-03-12  1       3
2021-03 2021-03-10  4       3
2021-04 2021-04-15  2       3
2021-04 2021-04-13  3       3
```

上述 SQL 等价于：

```
SELECT NULL,NULL,COUNT(DISTINCT cookieid) AS uv,0 AS GROUPING__ID FROM user_date

UNION ALL
```

```
SELECT month,NULL,COUNT(DISTINCT cookieid) AS uv,1 AS GROUPING__ID FROM user_date G
ROUP BY month

UNION ALL

SELECT NULL,day,COUNT(DISTINCT cookieid) AS uv,2 AS GROUPING__ID FROM user_date GRO
UP BY day

UNION ALL

SELECT month,day,COUNT(DISTINCT cookieid) AS uv,3 AS GROUPING__ID FROM user_date GR
OUP BY month,day;
```

- ROLLUP 的使用：

是 CUBE 的子集，以最左侧的维度为主，从该维度进行层级聚合。

比如，以 month 维度进行层级聚合：

```
SELECT
month,
day,
COUNT(DISTINCT cookieid) AS uv,
GROUPING__ID
FROM user_date
GROUP BY month,day
WITH ROLLUP
ORDER BY GROUPING__ID;
```

结果如下：

```
month   day     uv      grouping__id
NULL    NULL    7       0
2021-04 NULL    6       1
2021-03 NULL    5       1
2021-04 2021-04-16      2       3
2021-04 2021-04-15      2       3
2021-04 2021-04-13      3       3
2021-04 2021-04-12      2       3
2021-03 2021-03-12      1       3
2021-03 2021-03-10      4       3
```

把 month 和 day 调换顺序，则以 day 维度进行层级聚合：

```
SELECT
day,
month,
COUNT(DISTINCT cookieid) AS uv,
GROUPING__ID
```

```
FROM user_date
GROUP BY day,month
WITH ROLLUP
ORDER BY GROUPING__ID;
```

结果如下：

```
day         month    uv         grouping__id
NULL        NULL     7          0
2021-04-12  NULL     2          1
2021-04-15  NULL     2          1
2021-03-12  NULL     1          1
2021-04-16  NULL     2          1
2021-03-10  NULL     4          1
2021-04-13  NULL     3          1
2021-04-16  2021-04  2          3
2021-04-15  2021-04  2          3
2021-04-13  2021-04  3          3
2021-03-12  2021-03  1          3
2021-03-10  2021-03  4          3
2021-04-12  2021-04  2          3
```

这里，根据日和月进行聚合，和根据日聚合结果一样，因为有父子关系，如果是其他维度组合的话，就会不一样。