Day14_Hive窗口函数及优化

大数据-张军锋 Day14 Hive 优化 窗口函数 日志分析

Hive窗口函数

简介

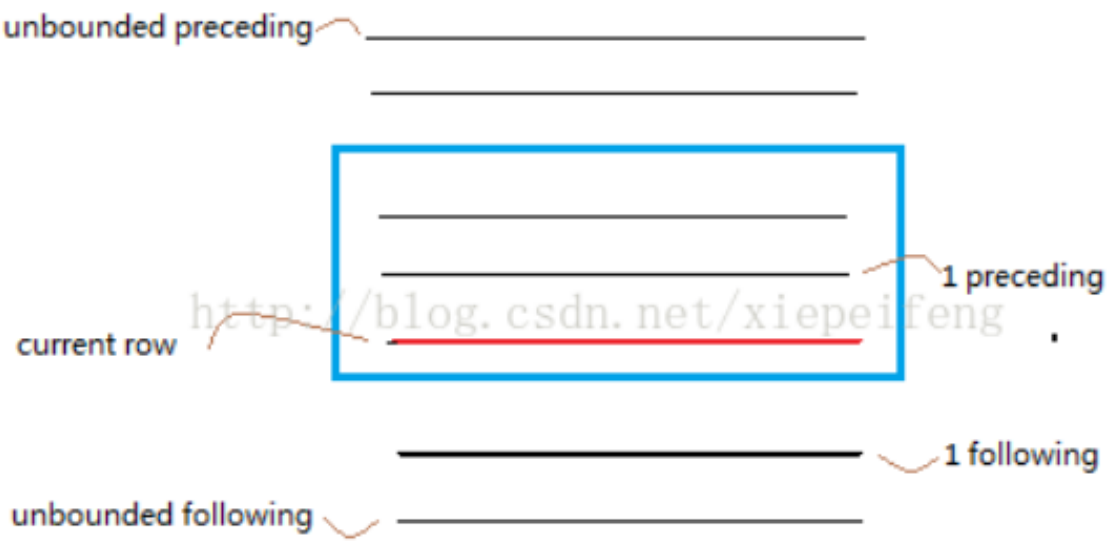
本文主要介绍hive中的窗口函数.hive中的窗口函数和sql中的窗口函数相类似,都是用来做一些数据分析类的工作,一般用于olap分析

概念

我们都知道在sql中有一类函数叫做聚合函数,例如sum()、avg()、max()等等,这类函数可以将多行数据按照规则聚集为一行,一般来讲聚集后的行数是要少于聚集前的行数的.但是有时我们想要既显示聚集前的数据,又要显示聚集后的数据,这时我们便引入了窗口函数.

> 在深入研究Over字句之前，一定要注意：在SQL处理中，窗口函数都是最后一步执行，而且仅位于Order by字句之前.

首先，**我们要知道什么是窗口子句**：
需要指定一个窗口的边界，**语法**是这样的：
ROWS betweenCURRENT ROW | UNBOUNDED PRECEDING | [num] PRECEDING AND UNBOUNDED FOLLOWING | [num] FOLLOWING| CURRENT ROW
或
RANGE between [num] PRECEDING AND [num]FOLLOWING
如下图：



ROWS是物理窗口，从行数上控制窗口的尺寸的；
RANGE是逻辑窗口，从列值上控制窗口的尺寸。这个比较难理解，但说白了就简单了，具体解释如下面例子.

> 结合order by子句使用，如果在order by子句后面没有指定窗口子句，则默认为：range between unbounded preceding and current row

数据准备

我们准备一张**order**表,字段分别为name,orderdate,cost.数据内容如下:

```
jack,2015-01-01,10
tony,2015-01-02,15
jack,2015-02-03,23
tony,2015-01-04,29
jack,2015-01-05,46
jack,2015-04-06,42
tony,2015-01-07,50
jack,2015-01-08,55
mart,2015-04-08,62
mart,2015-04-09,68
neil,2015-05-10,12
mart,2015-04-11,75
neil,2015-06-12,80
mart,2015-04-13,94
```

在hive中建立一张表t_window,将数据插入进去.

聚合函数+over

> 假如说我们想要查询在2015年4月份购买过的顾客及总人数,我们便可以使用窗口函数去去实现

```
select name,count(*) over()
from t_window
where substring(orderdate,1,7) = '2015-04'
```

得到的结果如下:

```
name    count_window_0
mart    5
mart    5
mart    5
mart    5
jack    5
```

> 可见其实在2015年4月一共有5次购买记录,mart购买了4次,jack购买了1次.事实上,大多数情况下,我们是只看去重后的结果的.针对于这种情况,我们有两种实现方式

- 第一种：distinct

```
select distinct name,count(*) over()
from t_window
where substring(orderdate,1,7) = '2015-04'
```

> 第二种:group by

```
select name,count(*) over()
from t_window
where substring(orderdate,1,7) = '2015-04'
group by name
```

执行后的结果如下:

```
name count_window_0
mart 2
jack 2
```

partition by子句

Over子句之后第一个提到的就是Partition By.Partition By子句也可以称为**查询分区子句**，非常类似于Group By，都是将数据按照边界值分组，而**Over之前的函数在每一个分组之内进行，如果超出了分组，则函数会重新计算**.

> 我们想要去看顾客的购买明细及月购买总额,可以执行如下的sql

```
select name,orderdate,cost,sum(cost)
over(partition by month(orderdate))
from t_window
```

执行结果如下:

```
name    orderdate   cost    sum_window_0
jack    2015-01-01   10         205
jack    2015-01-08   55         205
tony    2015-01-07   50         205
jack    2015-01-05   46         205
tony    2015-01-04   29         205
tony    2015-01-02   15         205
jack    2015-02-03   23         23
mart    2015-04-13   94         341
jack    2015-04-06   42         341
mart    2015-04-11   75         341
mart    2015-04-09   68         341
mart    2015-04-08   62         341
neil    2015-05-10   12         12
neil    2015-06-12   80         80
```

可以看出数据已经按照月进行汇总了.

order by子句

上述的场景,假如我们想要**将cost按照月进行累加**.这时我们引入**order by**子句.

> order by子句会让输入的数据强制排序（文章前面提到过，窗口函数是SQL语句最后执行的函数，因此可以把SQL结果集想象成输入数据）。Order By子句对于诸如Row_Number()，Lead()，LAG()等函数是必须的，因为如果数据无序，这些函数的结果就没有任何意义。因此如果有了Order By子句，则Count()，Min()等计算出来的结果就没有任何意义。

我们在上面的代码中加入order by

```
select name,orderdate,cost,sum(cost)
over(partition by month(orderdate) order by orderdate )
from t_window
```

得到的结果如下：(order by默认情况下聚合从起始行当当前行的数据)

```
name    orderdate   cost    sum_window_0
jack    2015-01-01   10          10
tony    2015-01-02   15          25
tony    2015-01-04   29          54
jack    2015-01-05   46          100
tony    2015-01-07   50          150
jack    2015-01-08   55          205
jack    2015-02-03   23          23
jack    2015-04-06   42          42
mart    2015-04-08   62          104
mart    2015-04-09   68          172
mart    2015-04-11   75          247
mart    2015-04-13   94          341
neil    2015-05-10   12          12
neil    2015-06-12   80          80
```

window子句

我们在上面已经通过使用partition by子句将数据进行了分组的处理.如果我们想要**更细粒度的划分**，我们就要引入window子句了.

> 我们首先要理解两个概念:
>
> - 如果只使用partition by子句,未指定order by的话,我们的聚合是分组内的聚合.
> - 使用了order by子句,未使用window子句的情况下,默认从起点到当前行.

当同一个select查询中存在多个窗口函数时,他们相互之间是没有影响的.每个窗口函数应用自己的规则.

window子句：

- **PRECEDING：**往前
- **FOLLOWING：**往后
- **CURRENT ROW：**当前行
- **UNBOUNDED：**起点，**UNBOUNDED PRECEDING** 表示从前面的起点，**UNBOUNDED FOLLOWING：**表示到后面的终点

> 我们按照name进行分区,按照购物时间进行排序,做cost的累加.
> 如下我们结合使用window子句进行查询

```
select name,orderdate,cost,
--所有行相加
sum(cost) over() as sample1,
--按name分组，组内数据相加
sum(cost) over(partition by name) as sample2,
--按name分组，组内数据累加
sum(cost) over(partition by name order by orderdate) as sample3,
--和sample3一样,由起点到当前行的聚合
sum(cost) over(partition by name order by orderdate rows between UN
BOUNDED PRECEDING and current row )  as sample4 ,
--当前行和前面一行做聚合
sum(cost) over(partition by name order by orderdate rows between 1
PRECEDING   and current row) as sample5,
--当前行和前边一行及后面一行
sum(cost) over(partition by name order by orderdate rows between 1
PRECEDING   AND 1 FOLLOWING  ) as sample6,
--当前行及后面所有行
sum(cost) over(partition by name order by orderdate rows between cu
rrent row and UNBOUNDED FOLLOWING ) as sample7
from t_window;
```

得到查询结果如下：

```
name    orderdate   cost    sample1 sample2 sample3 sample4 sample5
sample6 sample7
jack    2015-01-01  10      661     176     10      10      10
56      176
jack    2015-01-05  46      661     176     56      56      56
111     166
jack    2015-01-08  55      661     176     111     111     101
124     120
jack    2015-02-03  23      661     176     134     134     78
120     65
jack    2015-04-06  42      661     176     176     176     65
65      42
mart    2015-04-08  62      661     299     62      62      62
130     299
mart    2015-04-09  68      661     299     130     130     130
205     237
mart    2015-04-11  75      661     299     205     205     143
237     169
mart    2015-04-13  94      661     299     299     299     169
169     94
neil    2015-05-10  12      661     92      12      12      12
92      92
neil    2015-06-12  80      661     92      92      92      92
92      80
tony    2015-01-02  15      661     94      15      15      15
44      94
tony    2015-01-04  29      661     94      44      44      44
94      79
tony    2015-01-07  50      661     94      94      94      79
79      50
```

窗口函数中的序列函数

主要序列函数是不支持window子句的.

hive中常用的序列函数有下面几个:

NTILE

NTILE(n)，用于将分组数据按照顺序切分成n片，返回当前切片值

> NTILE不支持ROWS BETWEEN，
> 比如 NTILE(2) OVER(PARTITION BY cookieid ORDER BY createtime ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)
>
> - 如果切片不均匀，默认增加第一个切片的分布
> - 这个函数用什么应用场景呢?假如我们想要每位顾客购买金额前1/3的交易记录,我们便可以使用这个函数.

```
select name,orderdate,cost,
    --全局数据切片
    ntile(3) over() as sample1 ,
    -- 按照name进行分组,在分组内将数据切成3份
    ntile(3) over(partition by name),
    --全局按照cost升序排列,数据切成3份
    ntile(3) over(order by cost),
    --按照name分组，在分组内按照cost升序排列,数据切成3份
    ntile(3) over(partition by name order by cost)
from t_window
```

得到的数据如下：

```
name    orderdate   cost    sample1 sample2 sample3 sample4
jack    2015-01-01   10        3       1       1       1
jack    2015-02-03   23        3       1       1       1
jack    2015-04-06   42        2       2       2       2
jack    2015-01-05   46        2       2       2       2
jack    2015-01-08   55        2       3       2       3
mart    2015-04-08   62        2       1       2       1
mart    2015-04-09   68        1       2       3       1
mart    2015-04-11   75        1       3       3       2
mart    2015-04-13   94        1       1       3       3
neil    2015-05-10   12        1       2       1       1
neil    2015-06-12   80        1       1       3       2
tony    2015-01-02   15        3       2       1       1
tony    2015-01-04   29        3       3       1       2
tony    2015-01-07   50        2       1       2       3
```

如上述数据，我们去sample4 = 1的那部分数据就是我们要的结果

```
row_number、rank、dense_rank
```

这三个窗口函数的使用场景非常多

- row_number()从1开始，按照顺序，生成分组内记录的序列,row_number()的值不会存在重复,当排序的值相同时,按照表中记录的顺序进行排列
- 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

> **注意：** rank和dense_rank的区别在于排名相等时会不会留下空位.**

举例如下:

```
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 lxw1234
WHERE cookieid = 'cookie1';
```

得到的数据如下：

```
cookieid day           pv       rn1     rn2     rn3
cookie1 2015-04-12      7        1       1       1
cookie1 2015-04-11      5        2       2       2
cookie1 2015-04-15      4        3       3       3
cookie1 2015-04-16      4        3       3       4
cookie1 2015-04-13      3        5       4       5
cookie1 2015-04-14      2        6       5       6
cookie1 2015-04-10      1        7       6       7
```

```
rn1: 15号和16号并列第3, 13号排第5
rn2: 15号和16号并列第3, 13号排第4
rn3: 如果相等，则按记录值排序，生成唯一的次序，如果所有记录值都相等，或许会随机排吧。
```

LAG和LEAD函数

这两个函数为常用的窗口函数,可以返回上下数据行的数据.

> 以我们的订单表为例,假如我们想要查看顾客上次的购买时间可以这样去查询

```
select name,orderdate,cost,
lag(orderdate,1,'1900-01-01') over(partition by name order by order
date ) as time1,
lag(orderdate,2) over (partition by name order by orderdate) as tim
e2
from t_window;
```

查询后的数据为:

```
name    orderdate   cost    time1   time2
jack    2015-01-01   10  1900-01-01  NULL
jack    2015-01-05   46  2015-01-01  NULL
jack    2015-01-08   55  2015-01-05  2015-01-01
jack    2015-02-03   23  2015-01-08  2015-01-05
jack    2015-04-06   42  2015-02-03  2015-01-08
mart    2015-04-08   62  1900-01-01  NULL
mart    2015-04-09   68  2015-04-08  NULL
mart    2015-04-11   75  2015-04-09  2015-04-08
mart    2015-04-13   94  2015-04-11  2015-04-09
neil    2015-05-10   12  1900-01-01  NULL
neil    2015-06-12   80  2015-05-10  NULL
tony    2015-01-02   15  1900-01-01  NULL
tony    2015-01-04   29  2015-01-02  NULL
tony    2015-01-07   50  2015-01-04  2015-01-02
```

time1取的为按照name进行分组,分组内升序排列,取上一行数据的值.

time2取的为按照name进行分组，分组内升序排列,取上面2行的数据的值,注意当lag函数为设置行数值时,默认为1行.未设定取不到时的默认值时,取null值.

> lead函数与lag函数方向相反,取向下的数据.

first_value和last_value

- **first_value**取分组内排序后，截止到当前行，第一个值
- **last_value**取分组内排序后，截止到当前行，最后一个值

```
select name,orderdate,cost,
first_value(orderdate) over(partition by name order by orderdate) a
s time1,
last_value(orderdate) over(partition by name order by orderdate) as
time2
from t_window
```

查询结果如下:

```
name    orderdate   cost    time1       time2
jack    2015-01-01   10  2015-01-01  2015-01-01
jack    2015-01-05   46  2015-01-01  2015-01-05
jack    2015-01-08   55  2015-01-01  2015-01-08
jack    2015-02-03   23  2015-01-01  2015-02-03
jack    2015-04-06   42  2015-01-01  2015-04-06
mart    2015-04-08   62  2015-04-08  2015-04-08
mart    2015-04-09   68  2015-04-08  2015-04-09
mart    2015-04-11   75  2015-04-08  2015-04-11
mart    2015-04-13   94  2015-04-08  2015-04-13
neil    2015-05-10   12  2015-05-10  2015-05-10
neil    2015-06-12   80  2015-05-10  2015-06-12
tony    2015-01-02   15  2015-01-02  2015-01-02
tony    2015-01-04   29  2015-01-02  2015-01-04
tony    2015-01-07   50  2015-01-02  2015-01-07
```

课堂练习

数据准备

```
create table order_items(
    order_item_id int,
    order_id int,
    product_id int,
    quantity tinyint,
    subtotal float,
    product_price float
)
row format delimited
fields terminated by '|'

load data inpath '/orderdata/order_items'
overwrite into table order_items
```

ntile函数的应用

> 高价商品的平均价格，低价产品的平均价格

```
create table ntile_order_item
stored as orc
as
select order_item_id,
       order_id,
       product_id,
       quantity,
       subtotal,
       product_price,
       ntile(2) over(order by product_price) splitno
from order_items

select splitno,
        avg(quantity)
from ntile_order_item
group by splitno
```

rank()，dense_rank()，row_number()函数的应用

> 计算每个产品，每个订单的销量排名

```
create table rank_order_item
stored as orc
as
select order_item_id,
       product_id,
       quantity,
       rank() over(partition by product_id order by quantity desc),
       dense_rank() over(partition by product_id order by quantity
desc),
       row_number() over(partition by product_id order by quantity
desc)
from order_items

create table month_finish(
    date_month string,
    dep_name string,
    finish_amount string,
    task_amount string
)
row format delimited
fields terminated by '\t'
lines terminated by '\n'
stored as textfile;
load data local inpath '/opt/Software/Test/monthfinish.txt'
overwrite into table month_finish
```

数学函数+over函数

> 每个部门按照时间计算每个月的年累计完成销量

```
select date_month,
       dep_name,
       finish_amount,
       task_amount,
       -- 按照排序向上累计计算finish_amount值
       -- 等同于
       -- sum(finish_amount) over(partition by dep_name order by t
o_date(date_month) rows between unbounded preceding and current ro
w)
       sum(finish_amount) over(partition by dep_name order by to_da
te(date_month)),
       -- 计算部门一年总的完成量
       sum(finish_amount) over(partition by dep_name),
       -- 计算部门三个月的完成量
       sum(finish_amount) over(partition by dep_name order by to_da
te(date_month) rows between 2 preceding and current row),
       -- 计算部门当前往前两个月往后一个月的完成量
       sum(finish_amount) over(partition by dep_name order by to_da
te(date_month) rows between 2 preceding and 1 following),
       -- 计算部门累计完成量，往下累计
       sum(finish_amount) over(partition by dep_name order by to_da
te(date_month) rows between current row and unbounded following)
from month_finish
```

> 统计每个部门的年任务累计完成率

```
select  date_month,
        dep_name,
        finish_amount,
        task_amount,
        sum(finish_amount) over(partition by dep_name order by to_d
ate(date_month)),
        (sum(finish_amount) over(partition by dep_name order by t
o_date(date_month)))/(sum(task_amount) over(partition by dep_name))
from month_finish
```

日志分析

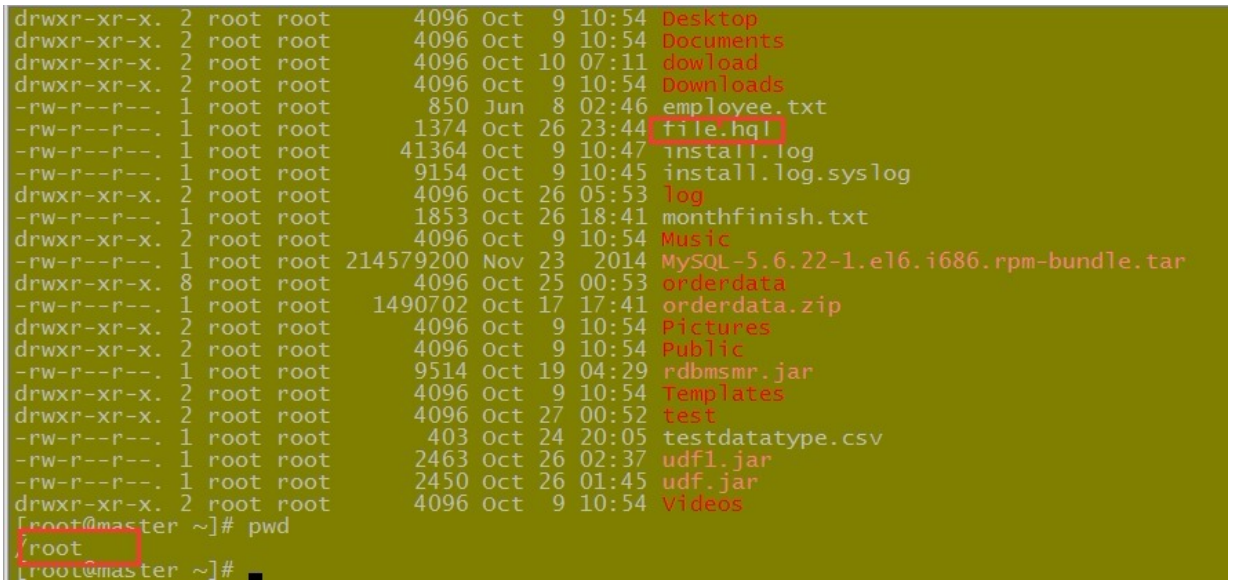
> 原来我们将分析日志，是直接在SQuirrel工具上执行的，没有考虑到sql的固化操作，等等。下面以日志分析为例子，阐述一下sql固化的问题

1. **编写hql**
   dateday是一个变量名，将日期给抽取出来了，提高代码的复用率

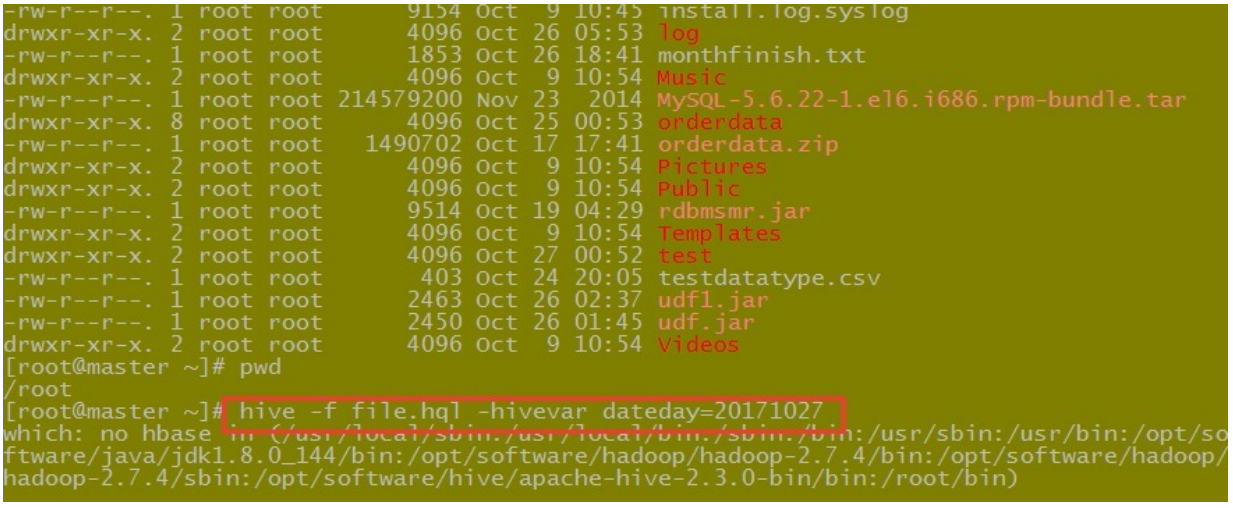
```
use db14;

-- create table
CREATE external TABLE if not exists apache_log (
  host STRING,
  identity STRING,
  username STRING,
  time STRING,
  request STRING,
  status STRING,
  size STRING,
  referer STRING,
  agent STRING)
  partitioned by (date_day string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" = "([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\]) ([^
\"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)(?: ([^ \"]*|\"[^\"]*\") ([^
\"]*|\"[^\"]*\"))?"
  ,"output.format.string"="%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s
%9$s"
)
STORED AS TEXTFILE;

alter table apache_log drop partition(date_day='${dateday}');
alter table apache_log add partition(date_day='${dateday}') locatio
n '/apachlog/${dateday}';
set hive.support.concurrency=true;
set hive.exec.dynamic.partition.mode=nonstrict;
set hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManage
r;
set hive.compactor.initiator.on=true;
set hive.compactor.worker.threads=1;


create table if not exists day_pv_uv(
    date_day int
    ,pv int
    ,uv int
)
clustered by(date_day) into 2 buckets
stored as orc
tblproperties("transactional"="true");

delete from day_pv_uv where date_day = '${dateday}';
insert into day_pv_uv
select  ${dateday}
    ,count(1) pv
```

```
      ,count(distinct host) uv
  from apache_log
  where date_day = '${dateday}';
```

2. **将hql上传到装hive的机器上**



3. **执行 hive -f aa.hql –hivevar dateday=20171027**



Hive优化

Group by 优化

Group By 很容易导致数据倾斜问题，因为实际业务中，通常是数据集中在某些点上，这也符合常见的2/8 原则，这样会造成对数据分组后，某一些分组上数据量非常大，而其他的分组上数据量很小，而在mapreduce 程序中，同一个分组的数据会分配到同一个reduce 操作上去，导致某一些reduce 压力很大，其他的reduce 压力很小，这就是数据倾斜，整个job 执行时间取决于那个执行最慢的那个reduce。
解决这个问题的方法是配置一个参数：set hive.groupby.skewindata=true。
当选项设定为 true，生成的查询计划会有两个 MR Job。第一个 MR Job 中， Map 的输出结果会随机分布到 Reduce 中，每个 Reduce 做部分聚合操作，并输出结果，这样处理的结果是相同的Group By Key 有可能被分发到不同的 Reduce 中，从而达到负载均衡的目的；第二个 MR Job再根据预处理的数据结果按照 Group By Key 分布到 Reduce 中（这个过程可以保证相同的GroupBy Key 被分布到同一个 Reduce 中），最后完成最终的聚合操作。

order by 优化

因为order by 只能是在一个reduce 进程中进行的，所以如果对一个大数据集进行order by,会导致一个reduce 进程中处理的数据相当大，造成查询执行超级缓慢。在要有进行order by 全局排序的需求时，用以下几个措施优化：

1. 在最终结果上进行order by，不要在中间的大数据集上进行排序。如果最终结果较少，可以
   在一个reduce 上进行排序时，那么就在最后的结果集上进行order by。
2. 如果需求是取排序后前N 条数据，那么可以使用distribute by 和sort by 在各个reduce 上进行排
   序后取前N 条，然后再对各个reduce 的结果集合并后在一个reduce 中全局排序，再取前N 条，因为参与
   全局排序的Order By 的数据量最多有reduce 个数*N，所以速度很快。

```
select a.leads_id,a.user_name from
(
    select leads_id,user_name from dealer_leads
    distribute by length(user_name) sort by length(user_name) desc
limit 10
) a order by length(a.user_name) desc limit 10;
```

sql语句优化

1. 尽量在select后面不要用*，需要哪些字段，使用字段名称来获取
2. 尽量不要使用distinct，用group by的特性来对数据进行排重

3. 使用exists和not exists代替in和not in
4. 有些时候or和可以使用union方式代替

join 优化

- 优先过滤后再join,最大限度地减少参与Join 的数据量。

```
select *
from employee a
inner join department b
on a.belong_dep_code=b.dep_code
where a.gender='女' and b.dep_address like '%北京%'

select *
from (
    select * from employee where gender='女'
)a
inner join （
    select * from department where dep_address like '%北京%'
）b
on a.belong_dep_code=b.dep_code
```

- 表 join 大表原则

> 应该遵守小表join 大表原则，原因是Join 操作的reduce 阶段，位于join 左边的表内容会被加载进内存，将条目少的表放在左边，可以有效减少发生内存溢出的几率。join 中执行顺序是从做到右生成Job，应该保证连续查询中的表的大小从左到右是依次增加的。

- join on 条件相同的放入一个job

> hive 中，当多个表进行join 时，如果join on 的条件相同，那么他们会合并为一个MapReduce Job，所以利用这个特性，可以将相同的join on 的放入一个job 来节省执行时间。

```sql
select pt.page_id,count(t.url) PV
from rpt_page_type pt
join
(
    select url_page_id,url from trackinfo where ds='2016-10-11'
) t on pt.page_id=t.url_page_id
join
(
    select page_id from rpt_page_kpi_new where ds='2016-10-11'
) r on t.url_page_id=r.page_id
group by pt.page_id;
```