优化时，把 hive sql 当做 map reduce 程序来读，会有意想不到的惊喜。
理解 hadoop 的核心能力，是 hive 优化的根本。这是这一年来，项目组所有成员宝贵的经验总结。

长期观察 hadoop 处理数据的过程，有几个显著的特征：
1.不怕数据多，就怕数据倾斜。
2．对 jobs 数比较多的作业运行效率相对比较低，比如即使有几百行的表，如果多次关联多次汇总，产生十几个 jobs，没半小时是跑不完的。map reduce 作业初始化的时间是比较长的。
3.对 sum，count 来说，不存在数据倾斜问题。
4.对 count(distinct),效率较低，数据量一多，准出问题，如果是多 count(distinct)效率更低。

优化可以从几个方面着手：

1. 好的模型设计事半功倍。
2. 解决数据倾斜问题。
3. 减少 job 数。
4. 设置合理的 map reduce 的 task 数，能有效提升性能。(比如，10w+级别的计算，用 160 个 reduce，那是相当的浪费，1 个足够)。
5. 自己动手写 sql 解决数据倾斜问题是个不错的选择。set hive.groupby.skewindata=true;这是通用的算法优化，但算法优化总是漠视业务，习惯性提供通用的解决方法。 Etl 开发人员更了解业务，更了解数据，所以通过业务逻辑解决倾斜的方法往往更精确，更有效。
6. 对 count(distinct)采取漠视的方法，尤其数据大的时候很容易产生倾斜问题，不抱侥幸心理。自己动手，丰衣足食。
7. 对小文件进行合并，是行至有效的提高调度效率的方法，假如我们的作业设置合理的文件数，对云梯的整体调度效率也会产生积极的影响。
8. 优化时把握整体，单个作业最优不如整体最优。

迁移和优化过程中的案例：

问题 1：如日志中，常会有信息丢失的问题，比如全网日志中的 user_id，如果取其中的 user_id 和 bmw_users 关联，就会碰到数据倾斜的问题。
方法：解决数据倾斜问题
解决方法 1. User_id 为空的不参与关联，例如：

```
Select *
From log a
Join    bmw_users b
On a.user_id is not null
And a.user_id = b.user_id
Union all
Select *
from log a
where a.user_id is null.
```

解决方法 2 ：

```
Select *
from log a
```

left outer join bmw_users b
on case when a.user_id is null then concat('dp_hive',rand()) else a.user_id end = b.user_id;

总结：2 比 1 效率更好，不但 io 少了，而且作业数也少了。1 方法 log 读取两次，jobs 是 2。2 方法 job 数是 1 。**这个优化适合无效 id(比如-99,'',null 等)产生的倾斜问题。**把空值的 key 变成一个字符串加上随机数，就能把倾斜的数据分到不同的 reduce 上 ,解决数据倾斜问题。因为空值不参与关联，即使分到不同的 reduce 上，也不影响最终的结果。附上 hadoop 通用关联的实现方法（关联通过二次排序实现的，关联的列为 parition key,关联的列 c1 和表的 tag 组成排序的 group key,根据 parition key 分配 reduce。同一 reduce 内根据 group key 排序）。

问题 2：不同数据类型 id 的关联会产生数据倾斜问题。
一张表 s8 的日志，每个商品一条记录，要和商品表关联。但关联却碰到倾斜的问题。s8 的日志中有字符串商品 id,也有数字的商品 id,类型是 string 的，但商品中的数字 id 是 bigint 的。猜测问题的原因是把 s8 的商品 id 转成数字 id 做 hash 来分配 reduce，所以字符串 id 的 s8 日志，都到一个 reduce 上了，解决的方法验证了这个猜测。
方法：把数字类型转换成字符串类型

```
Select * from s8_log a
Left outer join r_auction_auctions b
On a.auction_id = cast(b.auction_id as string);
```

问题 3：利用 hive 对 UNION ALL 的优化的特性
hive 对 union all 优化只局限于非嵌套查询。
比如以下的例子：

```
select * from
      (select * from t1
        Group by c1,c2,c3
        Union all
        Select * from t2
        Group by c1,c2,c3) t3
    Group by c1,c2,c3;
```

从业务逻辑上说，子查询内的 group by 怎么都看显得多余（功能上的多余,除非有 count(distinct)），如果不是因为 hive bug 或者性能上的考量(曾经出现如果不子查询 group by ，数据得不到正确的结果的 hive bug)。所以这个 hive 按经验转换成

```
select * from
      (select * from t1
        Group by c1,c2,c3
        Union all
        Select * from t2
        Group by c1,c2,c3) t3
    Group by c1,c2,c3;
```

经过测试，并未出现 union all 的 hive bug,数据是一致的。mr 的作业数有 3 减少到 1。
t1 相当于一个目录，t2 相当于一个目录，那么对 map reduce 程序来说，t1,t2 可以做为 map reduce 作业的 mutli inputs。那么，这可以通过一个 map reduce 来解决这个问题。Hadoop 的计算框架，不怕数据多，就怕作业数多。

但如果换成是其他计算平台如 oracle，那就不一定了，因为把大的输入拆成两个输入，分别排序汇总后 merge(假如两个子排序是并行的话)，是有可能性能更优的（比如希尔排序比冒泡排序的性能更优）。

问题 4：比如推广效果表要和商品表关联，效果表中的 auction id 列既有商品 id,也有数字 id,和商品表关联得到商品的信息。那么以下的 hive sql 性能会比较好

```
Select * from effect a
Join (select auction_id as auction_id from auctions
    Union all
    Select auction_string_id as auction_id from auctions
    ) b
On a.auction_id = b.auction_id。
```

比分别过滤数字 id,字符串 id 然后分别和商品表关联性能要好。

这样写的好处,1 个 MR 作业,商品表只读取一次，推广效果表只读取一次。把这个 sql 换成 MR 代码的话，map 的时候，把 a 表的记录打上标签 a,商品表记录每读取一条，打上标签 b，变成两个<key ,value>对，<b,数字 id>，<b,字符串 id>。所以商品表的 hdfs 读只会是一次。

问题 5：先 join 生成临时表，在 union all 还是写嵌套查询，这是个问题。比如以下例子：

```
Select *
From (select *
        From t1
        Uion all
        select *
        From t4
        Select *
        From t2
        Join t3
        On t2.id = t3.id
        )
Group by c1,c2;
```

这个会有 4 个 jobs。假如先 join 生成临时表的话 t5,然后 union all，会变成 2 个 jobs。

```
Insert overwrite table t5
Select *
        From t2
        Join t3
        On t2.id = t3.id
;
Select * from (t1 union t4 union all t5) ;
```

hive 在 union all 优化上可以做得更智能（把子查询当做临时表），这样可以减少开发人员的负担。出现这个问题的原因应该是 union all 目前的优化只局限于非嵌套查询。如果写 MR 程序这一点也不是问题，就是 muti inputs。

问题 6：使用 map join 解决数据倾斜的常景下小表关联大表的问题，但如果小表很大，怎么解决。这个使用的频率非常高，但如果小表很大，大到 map join 会出现 bug 或异常，这时

就需要特别的处理。云瑞和玉玑提供了非常给力的解决方案。以下例子：

```
Select * from log a
Left outer join members b
On a.memberid = b.memberid.
```

Members 有 600w+的记录，把 members 分发到所有的 map 上也是个不小的开销，而且 map join 不支持这么大的小表。如果用普通的 join，又会碰到数据倾斜的问题。

解决方法：

```
Select /*+mapjoin(x)*/* from log a
Left outer join (select    /*+mapjoin(c)*/d.*
                    From (select    distinct memberid from log ) c
                    Join members d
                    On c.memberid = d.memberid

)x
On a.memberid = b.memberid。
```

先根据 log 取所有的 memberid，然后 mapjoin 关联 members 取今天有日志的 members 的信息，然后在和 log 做 mapjoin。

假如，log 里 memberid 有上百万个，这就又回到原来 map join 问题。所幸，每日的会员 uv 不会太多，有交易的会员不会太多，有点击的会员不会太多，有佣金的会员不会太多等等。所以这个方法能解决很多场景下的数据倾斜问题。

问题 7：HIVE 下通用的数据倾斜解决方法,double 被关联的相对较小的表，这个方法在 mr 的程序里常用。还是刚才的那个问题：

```
Select    * from log a
Left outer join (select    /*+mapjoin(e)*/
                    memberid, number
                    From members d
                    Join num e
                    ) b
On a.memberid=    b.memberid
And mod(a.pvtime,30)+1=b.number。
```

Num 表只有一列 number，有 30 行，是 1,30 的自然数序列。就是把 member 表膨胀成 30 份，然后把 log 数据根据 memberid 和 pvtime 分到不同的 reduce 里去，这样可以保证每个 reduce 分配到的数据可以相对均匀。就目前测试来看，使用 mapjoin 的方案性能稍好。后面的方案适合在 map join 无法解决问题的情况下。

长远设想，把如下的优化方案做成通用的 hive 优化方法

1. **采样 log 表，哪些 memberid 比较倾斜，得到一个结果表 tmp1。由于对计算框架来说，所有的数据过来，他都是不知道数据分布情况的，所以采样是并不可少的。Stage1**
2. **数据的分布符合社会学统计规则，贫富不均。倾斜的 key 不会太多，就像一个社会的富人不多，奇特的人不多一样。所以 tmp1 记录数会很少。把 tmp1 和 members 做 map join 生成 tmp2,把 tmp2 读到 distribute file cache。这是一个 map 过程。Stage2**
3. **map 读入 members 和 log，假如记录来自 log,则检查 memberid 是否在 tmp2 里，如果是，输出到本地文件 a,否则生成<memberid,value>的 key,value 对，假如记录来自 member,**

生成<memberid,value>的 key,value 对，进入 reduce 阶段。Stage3.

4. 最终把 a 文件，把 Stage3 reduce 阶段输出的文件合并起写到 hdfs。

这个方法在 hadoop 里应该是能实现的。Stage2 是一个 map 过程，可以和 stage3 的 map 过程可以合并成一个 map 过程。

这个方案目标就是：倾斜的数据用 mapjoin,不倾斜的数据用普通的 join，最终合并得到完整的结果。用 hive sql 写的话，sql 会变得很多段，而且 log 表会有多次读。倾斜的 key 始终是很少的，这个在绝大部分的业务背景下适用。那是否可以作为 hive 针对数据倾斜 join 时候的通用算法呢？

问题 8：多粒度(平级的)uv 的计算优化，比如要计算店铺的 uv。还有要计算页面的 uv,pvip.

方案 1:

```
Select shopid,count(distinct uid)
From log group by shopid;
Select pageid, count(distinct uid),
From log group by pageid;
```

由于存在数据倾斜问题，这个结果的运行时间是非常长的。

方案二：

```
From log
Insert overwrite table t1 (type='1')
Select shopid
Group by shopid ,acookie
Insert overwrite table t1 (type='2')
Group by pageid,acookie;
```

店铺 uv:

```
Select shopid,sum(1)
From t1
Where type ='1'
Group by shopid ;
```

页面 uv:

```
Select pageid,sum(1)
From t1
Where type ='1'
Group by pageid ;
```

这里使用了 multi insert 的方法，有效减少了 hdfs 读，但 multi insert 会增加 hdfs 写，多一次额外的 map 阶段的 hdfs 写。使用这个方法，可以顺利的产出结果。

方案三：

```
Insert into t1
Select type,type_name,'' as uid
From (
Select   'page' as type,
          Pageid as type_name,
          Uid
From log
Union all
```

```
Select    'shop' as type,
          Shopid as type_name,
          Uid
From log ) y
Group by type,type_name,uid;
Insert into t2
Select type,type_name,sum(1)
From t1
Group by type,type_name;
From t2
Insert into t3
Select type,type_name,uv
Where type='page'
Select type,type_name,uv
Where type='shop' ;
```

最终得到两个结果表 t3,页面 uv 表，t4,店铺结果表。从 io 上来说，log 一次读。但比方案 2 少次 hdfs 写（multi insert 有时会增加额外的 map 阶段 hdfs 写）。作业数减少 1 个到 3，有 reduce 的作业数由 4 减少到 2，第三步是一个小表的 map 过程，分下表，计算资源消耗少。但方案 2 每个都是大规模的去重汇总计算。

这个优化的主要思路是，map reduce 作业初始化话的时间是比较长，既然起来了，让他多干点活，顺便把页面按 uid 去重的活也干了，省下 log 的一次读和作业的初始化时间，省下网络 shuffle 的 io，但增加了本地磁盘读写。效率提升较多。

这个方案适合平级的不需要逐级向上汇总的多粒度 uv 计算，粒度越多，节省资源越多，比较通用。

问题 9：多粒度，逐层向上汇总的 uv 结算。比如 4 个维度，a,b,c,d，分别计算 a,b,c,d,uv；a,b,c,uv;a,b,uv;a;uv,total uv4 个结果表。这可以用问题 8 的方案二，这里由于 uv 场景的特殊性，多粒度，逐层向上汇总，就可以使用一次排序，所有 uv 计算受益的计算方法。

案例：目前 mm_log 日志一天有 25 亿+的 pv 数，要从 mm 日志中计算 uv，与 ipuv,一共计算三个粒度的结果表

（memberid,siteid,adzoneid,province,uv,ipuv） R_TABLE_4

（memberid,siteid,adzoneid,uv,ipuv） R_TABLE_3

(memberid,siteid,uv,ipuv) R_TABLE_2

第一步：按 memberid,siteid,adzoneid,province,使用 group 去重,产生临时表，对 cookie,ip 打上标签放一起，一起去重，临时表叫 T_4;

```
Select memberid,siteid,adzoneid,province,type,user
From(
Select memberid,siteid,adzoneid,province, 'a' type ,cookie as user from mm_log where ds=20101205
Union all
Select memberid,siteid,adzoneid,province,'i' type ,ip as user from mm_log where ds=20101205
) x group by memberid,siteid,adzoneid,province,type,user ;
```

第二步：排名,产生表 T_4_NUM.Hadoop 最强大和核心能力就是 parition 和 sort.按 type，acookie 分组，

Type，acookie，memberid,siteid,adzoneid,province 排名。

```
Select * ,
row_number(type,user,memberid,siteid,adzoneid ) as adzone_num ,
row_number(type,user,memberid,siteid ) as site_num,
row_number(type,user,memberid ) as member_num,
row_number(type,user ) as total_num
from (select  * from T_4 distribute by type,user sort by type,user, memberid,siteid,adzoneid ) x;
```

这样就可以得到不同层次粒度上 user 的排名，相同的 user id 在不同的粒度层次上，排名等于 1 的记录只有 1 条。取排名等于 1 的做 sum，效果相当于 Group by user 去重后做 sum 操作。

第三步：不同粒度 uv 统计，先从最细粒度的开始统计，产生结果表 R_TABLE_4,这时，结果集只有 10w 的级别。

如统计 memberid,siteid,adzoneid,provinceid 粒度的 uv 使用的方法就是

```
Select memberid,siteid,adzoneid, provinceid,
sum(case when  type =’a’ then cast(1) as bigint end ) as province_uv ,
sum(case when  type =’i’ then cast(1) as bigint end ) as province_ip ,
sum(case when adzone_num =1 and type =’a’ then cast(1) as bigint end ) as adzone_uv ,
sum(case when adzone_num =1 and type =’i’ then cast(1) as bigint end ) as adzone_ip ,
sum(case when site_num =1 and type =’a’ then cast(1) as bigint end ) as site_uv ,
sum(case when site_num =1 and type =’i’ then cast(1) as bigint end ) as site_ip ,
sum(case when member_num =1 and type =’a’ then cast(1) as bigint end ) as member_uv ,
sum(case when member_num =1 and type =’i’ then cast(1) as bigint end ) as member_ip ,
sum(case when total_num =1 and type =’a’ then cast(1) as bigint end ) as total_uv ,
sum(case when total_num =1 and type =’i’ then cast(1) as bigint end ) as total_ip ,
from T_4_NUM
group by memberid,siteid,adzoneid, provinceid ;
```

广告位粒度的 uv 的话，从 R_TABLE_4 统计，这是源表做 10w 级别的统计

```
Select memberid,siteid,adzoneid,sum(adzone_uv),sum(adzone_ip)
From R_TABLE_4
Group by memberid,siteid,adzoneid；
```

memberid,siteid 的 uv 计算 ，

memberid 的 uv 计算,

total uv 的计算也都从 R_TABLE_4 汇总。