# Mdrill项目在lucene的改进上的10点心得

<https://github.com/alibaba/mdrill>

## 修改创建索引逻辑，让索引能够基于hdfs中进行创建

### 目的：

这样就不在需要使用本地硬盘，可以通过mapreduce并发的在hadoop中创建索引，从而解决离线创建索引的速度，而且也同时解决创建索引过程中本地必须要有大硬盘的囧况。

### 原理：

之所以原先的lucene不能再hdfs中创建索引，是因为lucene中存在随机写，而hdfs 不支持随机写导致，仔细阅读lucene源码发现，lucene使用随机写的场景只有两种，

一种是在文件的头部预留出一个int长度的空间，等待索引创建完毕后，更新这个预留的int位置，标记上该索引一共有多少条记录。

另外一种是存在文件校验crc32，前面预留出一个long类型的空间，在后续写入数据后，得到其crc32的值后，重新写入。

综上所述，这些随机写是可以避免的，我们的处理办法是不在预留这些空间，而是将其值顺序的写到另外一个文件中去。

## addIndexesNoOptimize的优化

### 目的：

该方法了解lucene的人应该知道，是向当前索引中添加一个新的索引，通常来说我们在mapreduce的第一个阶段会通过大并发创建小索引，在第二个阶段会通过addIndexesNoOptimize的方法将这些小的索引合并成一个完整的最终的索引。

目前lucene在这个地方的实现并不是特别好，addIndexesNoOptimize的处理逻辑是先将外部的索引copy到当前索引所在的目录，然后在进行合并，所以这个就多了一个copy的过程

这样做目前有3个缺点

1. 当数据量特别大的时候，因为有了一次额外的copy，这种copy带来的开销是很大的，而且也是没必要的。
2. 因为这这种copy将索引都copy到同一个目录上了，也就意味着在同一个磁盘上，那么在合并索引的时候还需要将这些文件重新读取一遍，单个磁盘的读取速度是有限的，不能利用多个磁盘进行合并会影响合并速度。
3. 很多时候我希望当前索引下的不同的sigments能够分布到不同的硬盘上，这样检索的时候，同一个索引不同的sigments能够使用不同的硬盘进行检索。

### 原理：

针对上述问题，我们对lucene进行了一次比较小的改进，大家可以将其理解为linux下的文件的软连接，实际的addIndexesNoOptimize方法并不会真正的发生copy，而是仅仅在当前的索引中做了一个标记，标记出他们附加的外部索引存储在什么位置，而不是真的去copy他们。

之前我们创建索引使用60个shards，也就是60个reduce，但是我们的map数量为8700个，故reduce数量偏小，并行数偏低，发挥不了集群的性能。  
分析后，之所以使用60个shards，是我们最终要生成20个索引，而生成这60个索引除了optimize这个步骤将所有的小索引统一合并外，其他的生 成小索引完全是可以增加并行数的。故我们启动了两次mapreduce，第一次mr就是专门负责生成小索引，reduce数量为600个,第二次就是将这 些索引通过addIndexesNoOptimize合并成60个最终的索引,reduce=60。

## 关于lucene倒排表 tii文件的改进

### 目的：

常规lucene,并不适合频繁的打开于关闭索引，主要原因是打开索引于在加载 tii文件上的开销比较大，而mdrill因为本身会接入很多表，而且每张表都会存在很多分区，因数据量巨大（千亿级别）在mdrill中会存在大量的索引文件，因内存等系统资源限制，这些索引文件不可能都同时打开，而是查询那个索引去打开那个索引，查看哪天的数据在打开哪天的数据，不同的索引是处于关闭的状态的。

故我们对lucene的tii文件做了改动，目的是解决如下2个问题

1. 减少频繁打开与关闭索引带来的加载tii文件的开销，包括时间开销。
2. 减少tii文件整体load到内存中带来的内存开销。

### 原理：

在lucene中tii文件与tis文件一般组合使用，tii文件是tis文件的跳跃表，当我们对一个关键词进行检索的时候，首先去tii文件中区查找某个关键词在tis文件中可能落在那个区间范围，这个范围定位后再去 tis 文件中去查找某个关键词出现在那些文档里，从而达到检索的作用。

目前lucene的实现tii文件整体式加载到内存里的，在内存里构建一个有序的数组，查询比对的时候采用二分查找法。可以想象tii文件时tis文件的1/128，文档数量比较少，重复值比较大的列还可以接受，但是像userid ，sessionid等列，因为记录数很多，重复值很少，所以倒排表tis本身就很大，对应的tii文件也特别大，这样打开索引的时候构建时间就很长（虽然可能其中的绝大部分列我根本不会去查），另外一个是放到内存中的有序的数组特别的占用内存，对于内存资源严重缺乏的mdrill来说太浪费了。

我们目前的做法是，稍微小改动了一些tii文件，将其实现由首次打开load到内存中进行二分查找，改为基于文件内的二分查找，这个如果大家理解纯真IP库的原理的话，那么基于文件的二分查找应该不会特别难理解，我们采用2个文件，第一个文件存储了定长的偏移量，第二个文件存储了改偏移量对应的term的内容。

## 关于solr的fieldvaluecache的改进，以及我们对solr内存的优化

### 目的：

Solr的fieldvaluecache故名思议，就是提供一种方法，快速的获取某一个列的值,一般在对数据进行分组和统计的时候会经常用到此类，对统计性能影响至关重要，相信很多类似系统，比如说mdrill和garuda都会在这个上面花费了很多的功夫，而且系统的性能好坏与这个地方的设计好坏存在最为直接的关系。

这里说下solr的实现思路

Solr在这里并没有使用lucene的正排的方式直接获取某一个docid对应的某一个列的值，原因为：lucene的正向存储，并没有采用列存储，如果想要读取一个列的值，需要先根据docid去fdx中区查询其在fdt中的偏移量offset，然后根据偏移量offset去fdt文件中对应的区域去查找该列的值，因为并没有采用列存储，故一次docid的查找会存在很多次的seek操作（列数越多，seek的次数越多），因此存在大量的随机读。我们的统计场景一次的统计往往是数亿个docid,如果每个docid又有上百个列，那么seek的次数将会非常恐怖，目前的普通磁盘时绝对受不了这个的。

Solr的实现目前有两种不同的实现

第一：对于 sum,max,min,avg等统计场景，solr3.6使用lucene内置的实现

即便利一次倒排表，在内存中构建docid->对应值得一个数组，如果是字符串类型的列，会对相同的字符串进行优化，仅仅在内存中保存一次。

不言而喻，这种lucene的实现是很坑爹的（貌似lucene4中有较大的改进），数据量比较小的时候，性能还可以，但是数据量很大的时候，占用的内存非常高，使用solr的人在数据量大的时候经常朋友oom有没有

第二：稍微做了一些改进，针对仅仅facet求count的场景，是采用遍历一次倒排表，因为倒排表本身安列存储，并且倒排表里列的值得存储时有序的，故solr在遍历倒排表的同时会对该列的每一个值进行一个编号，比如说用1代替张三，用2代替李四，然后在内存中构建一个docid到编号对应的素组，在进行facet计算的时候仅仅使用这个编号进行group by 进行统计，在展示给用户的时候在去索引中查找将编号还原成原始的值，这种方法的优点是，比较节省内存，缺点是只能进行count，以及单列的group by.

上述两种方法除了上面说的问题外还有如下的问题

1. 首次查询的时候因为fieldvaluecache还没有构建，都需要遍历利一次倒排表去实现，首次查询用户的等待时间过长，哪怕是我统计的记录仅仅是整体的1/10，甚至是1/100，也要完整的扫描一遍倒排表，你说坑爹不坑爹。
2. 对于实时场景，因为经常有新记录的添加和修改很频繁，话费了很大力气去构建的这个fieldvaluecache往往因为只添加了一条记录，导致这个cache失效，还得重新去生成一遍，频繁的生成和失效，solr频繁的full gc 有没有。

针对这种现状，我们要解决如下的几个问题

1. 减少对内存的使用，如果能降低到原来的十分之一，甚至是百分之一，那么mdrill整体能够存储的数据量就会多很多。
2. 尽量缩短首次构建这个cache的时间。
3. 实时模式的添加和修改，尽量减少cache失效的频率。
4. 频繁full gc的问题。

### **原理:**内存的问题：

1. 借鉴solr进行termNum to termText构建优化

Lucene那种将全部的值都load到一个数组中肯定是非常不可取的，那样的话mdrill就会沦落为一个基于内存的系统了，只能是那种高富帅的产品才能用得起，故想要屌丝产品能玩，内存必须得优化。

故我们借鉴solr的一些做法

termNum to termText 是solr为了压缩fieldValue cache而实现，在一些长文本的上能极大的减少fieldValue的内存使用量。原理就是简单的通过将field的值进行编号，比方说 如果某一列值是 aaaaaa，bbbbbb，cccccc ，编号后，1就代表aaaaa，2代表bbbbbb，3代表ccccc ，如果字符串比较长，这种编号就能节省很多内存

termNum to termText采用类似128跳跃表的方式存储，故对内存消耗比较小。

针对这个思路，我们将lucene默认的将整个列数据全部都load到内存中的方式修改为load 每个列的值的编码代号，操作的时候也仅仅操作这些代号，真正展现的时候再将这些编号转换成真实的值，编号的数据类型根据某个列的值的重复程度可以为byte,short,int

具体的内存的数据结构为2个素组，一个数组存储 docid对应的编号，另外一个数组存储编号对应的真实值（对于字符串类型，是存储在文件中，内存中存储偏移量）

在group by和排序的时候，仅使用数值代号

呈现给用户的时候，将代号转换回原始值

这样做

一是节省了内存，

二是对于字符串类型的长文本是有好处的，首先解决了字符串类型的计算相比数值型慢的问题，也解决了IO问题，长文本占用空间较大，完整扫描一遍IO时间太长……

1. 上面那个仅仅是通过变长的方法节省了一部分内存，但是如果数据量特别大内存问题依然没有解决，故我们改变了一种思路，不在一味的追求巨大的索引，比如说对于我们的一些表，每天大概1亿的数据量，但是因为要存储1年，意味着要存储365亿的数据，如果生成一个完整的大索引365亿个byte的内存也是非常大的，我们对索引进行了拆分，比如说我们拆分成了365分的话，计算整体的话，我们是依次的计算第1,2,3….365个索引，然后将每个索引的计算结果进行合并，这样整体的内存消耗就降低到了原先的1/365,默认solr是不支持这样干的，在solr core的代码上我们处理了一下，

用到的数据会加载到分区中，不用的分区会从内存中踢出，采用LRU的方式管理，如果同时需要检索大量的分区，则进行排队处理，一个分区一个分区的处理.针对如何进行分区，我们提供了接口出来，交给用户处理

1. 在solr3.6版本中，如果存在多个表，需要为每张表单独分配内存，随着接入的表越来越多，需要的内存也越来越多，但是往往很多表根本就差的很少，却要为他们分配内存，资源浪费特别严重，故我们修改了下solr，让多个表之间可以共享内存，不经常使用的表，腾出内存给频繁使用的表来使用。
2. 原先的solr中有一个merger server，是跟真实的索引（shard）放在一起的，用来合并多个shard的结果，每次查询的时候随机使用其中一个shard作为merger server,如果每次merger server使用1G内存，那么所有的shard都要为之单独分配1G内存,shard数量特别多，太浪费了，所以我们将merger server单独分离出来，避免内存的浪费。
3. 按照内存大小进行LRU，而不是按照field的个数，不同列因为重复读不同对内存的消耗也不一样，按照个数lru不合理，修改为按照列使用的总内存使用LRU。

### **原理:**首次查询构建时间太长的问题：

Solr采用的遍历倒排表构建fieldvalue的cache的方式是特别亏的

首先：倒排表存储了term的真是值，对应doclist的各种偏移量，词频率，位置以及由很多种压缩算法需要消耗计算资源，IO和cpu使用都比较大。

其次：很多时候的查询并非对全部记录进行扫描，可能仅仅扫描其中的百分之一，甚至千分之一、万分之一，但是却要扫描整个倒排表去构建

针对这种情况我们在倒排表的基础上使用了多种方式去优化，比如[说对doclist进行各种压缩](https://github.com/alibaba/mdrill/issues/14)，使用[doclist的跳跃功能跳跃过不必要的doc](https://github.com/alibaba/mdrill/issues/69)，但效果都不是特别好，最终我们采用了一个比较简单干净的方法，就是在创建索引的时候，直接构建好docid到对应的值代号的关系 的文件，这个文件的特点是只存储值代号，并且值代号是定长，可以进行跳跃的读取（跳跃掉不需要的记录），而且这个值代号的存储也不一定是4字节，根绝列的重复成都可以为1~4字节，值代号对应的值得真实值也是采取这种类似的方式，定长存储。而且在真正查询的时候，读取的时候我们采用顺序的向后跳跃读取，不会反复的在文件前后来回跳跃，只会向前跳跃，而且采用列式存储，列和列之前分开。对于很多场景值需要读出docid对应的值代号，而不真的需要读出代号的真实的值。

### **原理:**减少cache失效的频率

之前solr的cache的粒度仅仅到了索引 index级别，因为频繁的添加和修改索引，导致实时模式的cache无效，故我们更改了cache的粒度，由index级别到index下的sigment级别，这样添加和修改索引，改变的仅仅是索引index下的一个sigment，而绝大部分sigment并没有频繁的改变,这样cache利用率提升很高

### **原理:频繁full gc的问题**

采用block cache来实现，不会的频繁的申请和释放内存，当某个内存对象不使用的时候不是交给gc 去释放，而是简单做了个标记，下次申请内存的时候直接再次拿出来使用。

## solr 的FQ Cache的不足以及在TOP N 全文检索上的改进

举个倒排表的例子

性别：男 =>1,2,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20

手机：1340100xxxx =>11

可以看出上述两个列的值有很大区别，性别列，因为值得重复程度特别多会有大量的docid对应性别是男的用户

而对于手机这个列为，一般一个手机号只对应一个docid

第一个场景

那么如果我去查找 性别是男的 前10条记录 而不考虑任何的排序的话，我仅仅从头读出10个docid 就可以了，但实际上solr和lucene本身并没有这样干，solr是为了生成一个完整的bitset作为缓存，将全部的值都会读出来，之后作为缓存放在内存里，对于lucene来说它的默认的collect实现也是收集全部的docid，而不是收集到10个就停止了（它这样做的目的是为了全文检索里面的余弦排序，但很多场景并不需要排序），如果对应几千万条记录的话，IO浪费很多，是很亏的，很有必要自己单独写一个collect.

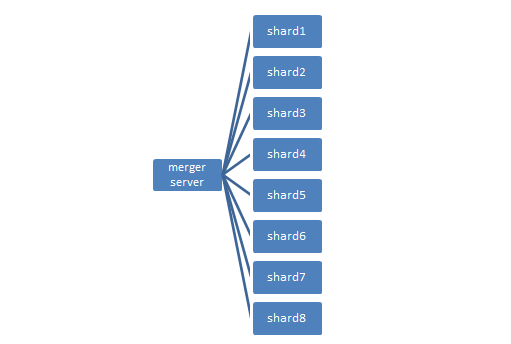
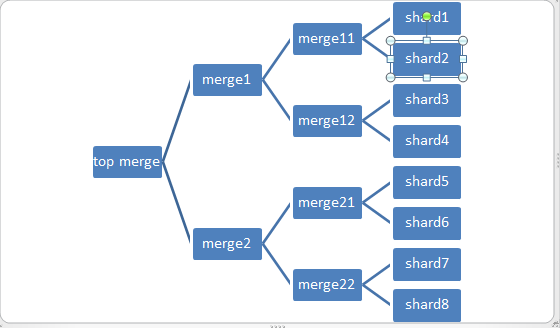
第二个场景

我们查找性别是男的 并且手机号是 1340100xxxx的用户，很明显，结果就是docid=11的这个用户，这个处理的时候 如若大家的过滤条件是通过solr的两个不同的fq参数传递进去的时候，就还会存在第一个场景的问题，性别是男的那个列浪费了很多的IO，所以这个地方要注意改为 让他们在同一个FQ里面，使用lucene的booleanQuery去查询，这样因为doclist本身具有跳跃的性质，性别的那个列的相当一部分的docid都会跳跃过去，而节省了IO，所以自某些场景要做适当的优化。

## 多级merger server

参考地址：<http://user.qzone.qq.com/165162897/blog/1355157666>

目的为解决集群的可拓展性，解决单个merger server瓶颈的问题

原先是这样的  
  
  
现在是  
  


merger server的调度，多层次的merger server会优先合并本机shards的结果，减少网络IO

## 分区

之前的solr只能使用单个索引，每次搜索的时候，也只能使用这个索引，如果数据量太大，那么solr分配的内存肯定不够(参考后面fieldvalue对内存的阐述)。  
但是实际情况是，大部分用户的查询仅仅局限在最近的几个月的数据，其他很久前的数据很少查询，如果也load到solr中特别浪费资源。  
故考虑到将sorl进行分区处理，值加载经常使用的索引，不经常使用的索引，查询完毕后资源会释放。

## 废弃正向索引fdt,fdx等不在使用，减少2/3的存储体积

在mdrill模式下，正向索引已经没有用了，全部替换后面的fieldvaluecache

## 修改solr facet，可以进行多列group by与统计

之前solr facet只能进行单列的count，现在修改为可以多列group by，以及sum,max,min,avg等统计，并且可以按照列或者sum,avg,max,min等排序

**这里重点介绍下 ：对于某些维度的组合结果特别多的TOP N的处理**

假设 每个shard里的数据 分布规律 跟整体是一致的(在海量数据下 数据量越多 每个shard的分布规律与整体越像 )

所谓的top N是近似计算

是对每个shard内取的top N后进行汇总

而不是先汇总后在取TOP N，这其中的排序会有可能有误差

一共分为两次查询，第一次查询的结果仅仅是为了排序，真正的结果是第二次查询出来的。

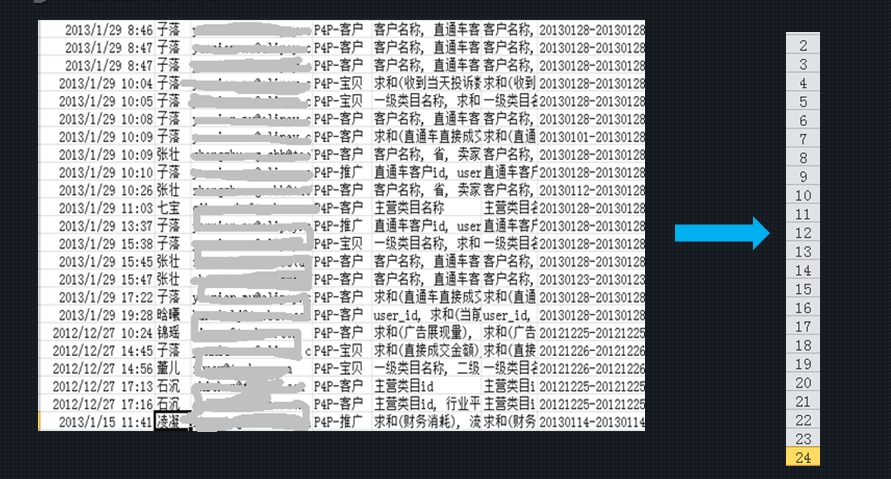
以userid为例，我想查询这一年，购买商品次数最多的用户.  
这些用户的交易，在mdrill里存储是分布在不同的机器里上的。  
由于排序是在每个shard里进行的排序，  
第一shard里排序结果是1,2,3,4,5,6,7 取三个用户为1,2,3，  
第二个shard里 可能是 3,4,5,6,7,8,9,2,3 取top3为3,4,5.

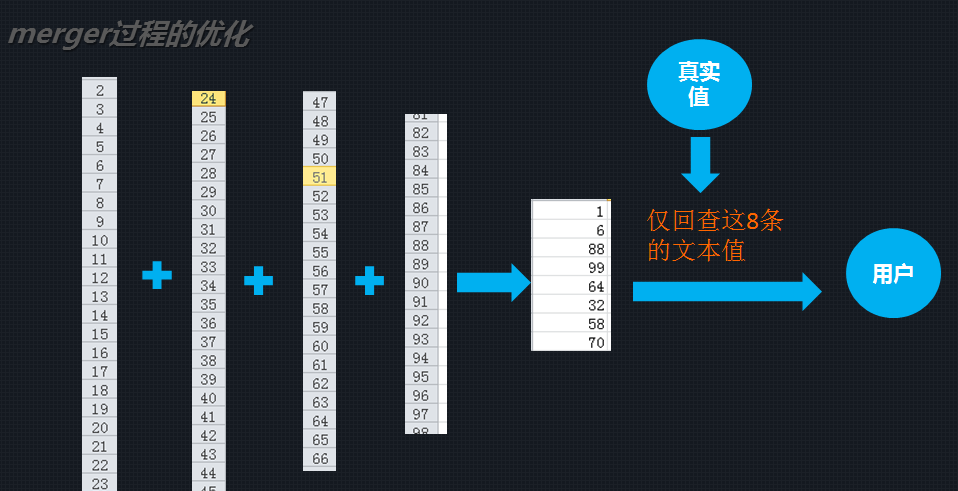
汇总后结果为2,3,4  
那么对于2和4这个用户来说 信息是不完整的，仅仅提现了近似的排序关系。  
故需要进行二次查询，重新计算一次2,3,4的真实汇总结果，第二次查询，屏蔽了2,3,4以外的组，所以数据迁移成本很小

## merger过程的优化

之前每个shard返回给merger server的原始真实的值，这个数据可能比较大，特别是返回的数据行数比较多的查询，跨机器的IO比较多，故我们做了一个处理，除了排序字段外，对返回的每条记录进行一个crc32计算，在每个shard内保存crc32对应的真实值，然后返回给merger server的仅仅是crc32的值，当merger server合并完成后，真实的应用可能仅仅需要其中的几条记录，那么在根据这几条记录的crc32，去shard里回查出来即可，如下图所示

看不到图可以去mdrill官方的ppt里查找





By 子落 20140612