# 概述

在MySQL中的ORDER BY有两种排序实现方式：

1、利用有序索引获取有序数据；

2、文件排序。

在使用explain分析查询的时候，利用有序索引获取有序数据显示Using index。如果MySQL在排序的时候没有使用到索引那么就会输出using filesort，即使用文件排序。

Using filesort经常出现在order by、group by、distinct、join等情况下。

文件排序是通过相应的排序算法，将取得的数据在内存中进行排序：MySQL需要将数据在内存中进行排序，所使用的内存区域也就是我们通过sort\_buffer\_size系统变量所设置的sort buffer（排序区）。**这个sort buffer是每个Thread独享的**，所以说可能在同一时刻在MySQL中可能存在多个sort buffer内存区域。

参考：

<https://blog.csdn.net/lijingkuan/article/details/70341176>

<https://www.cnblogs.com/aeolian/p/10212892.html>

# 排序模式

select \* from film where Producer like '东京热%' and prod\_time>'2015-12-01' order by actor\_age;

我们想查询'东京热'出品的，从去年12月1号以来，并且按照演员的年龄排序的电影信息。

这种情况下，使用索引已经无法避免排序了，那MySQL排序到底会怎么做列。笼统的来说，它会按照：

1. 依据“Producer like‘东京热%’and prod\_time>’2015-12-01’”过滤数据，查找需要的数据；
2. 对查找到的数据按照“order by actor\_age”进行排序，并按照“select \*”将必要的数据按照actor\_age依序返回给客户端。

空口无凭，我们可以利用MySQL的optimize trace来查看是否如上所述。 如果通过optimize trace看到更详细的MySQL优化器trace信息，可以查看阿里印风的博客初识5.6的optimizer trace trace结果如下：

依据“Producer like ‘东京热%’and prod\_time>’2015-12-01’”过滤数据，查找需要的数据

"attaching\_conditions\_to\_tables": {

"original\_condition": "((`film`.`Producer` like '东京热%') and (`film`.`prod\_time` > '2015-12-01'))",

"attached\_conditions\_computation": [

],

"attached\_conditions\_summary": [

{

"table": "`film`",

"attached": "((`film`.`Producer` like '东京热%') and (`film`.`prod\_time` > '2015-12-01'))"

}

]

}

对查找到的数据按照“order by actor\_age”进行排序，并按照“select \*”将必要的数据按照actor\_age依序返回给客户端

"join\_execution": {

"select#": 1,

"steps": [

{

"filesort\_information": [

{

"direction": "asc",

"table": "`film`",

"field": "actor\_age"

}

],

"filesort\_priority\_queue\_optimization": {

"usable": false,

"cause": "not applicable (no LIMIT)"

},

"filesort\_execution": [

],

"filesort\_summary": {

"rows": 1,

"examined\_rows": 5,

"number\_of\_tmp\_files": 0,

"sort\_buffer\_size": 261872,

"sort\_mode": "<sort\_key, packed\_additional\_fields>"

}

}

]

}

这里，我们可以明显看到，MySQL在执行这个select的时候执行了针对film表.actor\_age字段的asc排序操作。

"filesort\_information": [

{

"direction": "asc",

"table": "`film`",

"field": "actor\_age"

}

我们这里主要关心MySQL到底是怎么排序的，采用了什么排序算法。

请关注这里：

"sort\_mode": "<sort\_key, packed\_additional\_fields>"

MySQL的sort\_mode有三种。摘录5.7.13中sql/filesort.cc源码如下：

Opt\_trace\_object(trace, "filesort\_summary")

.add("rows", num\_rows)

.add("examined\_rows", param.examined\_rows)

.add("number\_of\_tmp\_files", num\_chunks)

.add("sort\_buffer\_size", table\_sort.sort\_buffer\_size())

.add\_alnum("sort\_mode",

param.using\_packed\_addons() ?

"<sort\_key, packed\_additional\_fields>" :

param.using\_addon\_fields() ?

"<sort\_key, additional\_fields>" : "<sort\_key, rowid>");

“< sort\_key, rowid >”和“< sort\_key, additional\_fields >” 看过其他介绍介绍MySQL排序文章的同学应该比较清楚，“< sort\_key, packed\_additional\_fields >” 相对较新。

< sort\_key, rowid >对应的是MySQL 4.1之前的“原始排序模式”。

< sort\_key, additional\_fields >对应的是MySQL 4.1以后引入的“修改后排序模式”。

< sort\_key, packed\_additional\_fields >是MySQL 5.7.3以后引入的进一步优化的"打包数据排序模式”。

## 回表排序模式

根据索引或者全表扫描，按照过滤条件获得需要查询的排序字段值和row ID；

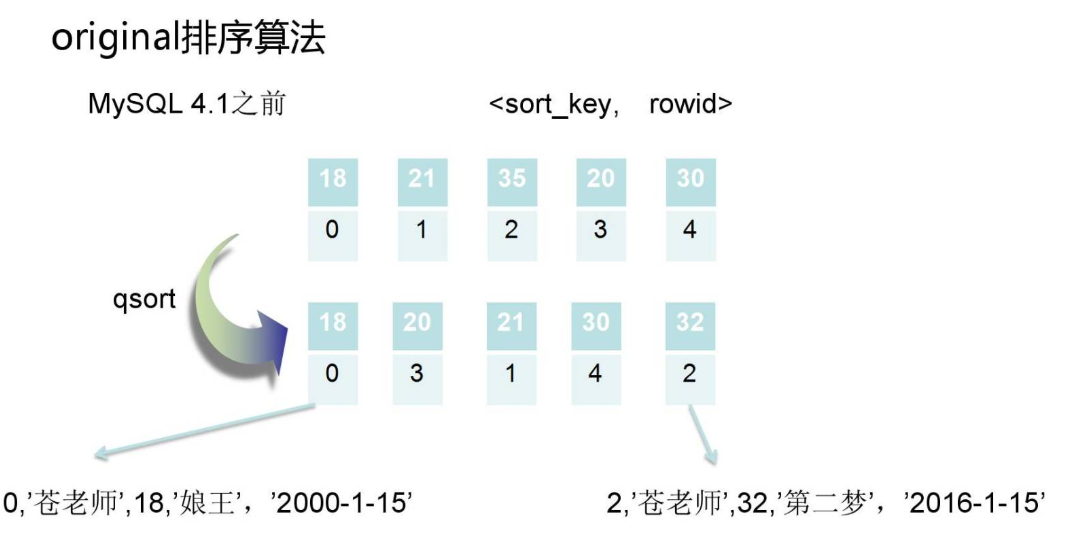
将要排序字段值和row ID组成键值对，存入sort buffer中；

如果sort buffer内存大于这些键值对的内存，就不需要创建临时文件了。否则，每次sort buffer填满以后，需要直接用qsort(快速排序算法)在内存中排好序，并写到临时文件中；

重复上述步骤，直到所有的行数据都正常读取了完成；

用到了临时文件的，需要利用磁盘外部排序，将row id写入到结果文件中；

根据结果文件中的row ID按序读取用户需要返回的数据。由于row ID不是顺序的，导致回表时是随机IO，为了进一步优化性能（变成顺序IO），MySQL会读一批row ID，并将读到的数据按排序字段顺序插入缓存区中(内存大小read\_rnd\_buffer\_size)。



## 不回表排序模式

根据索引或者全表扫描，按照过滤条件获得需要查询的数据；

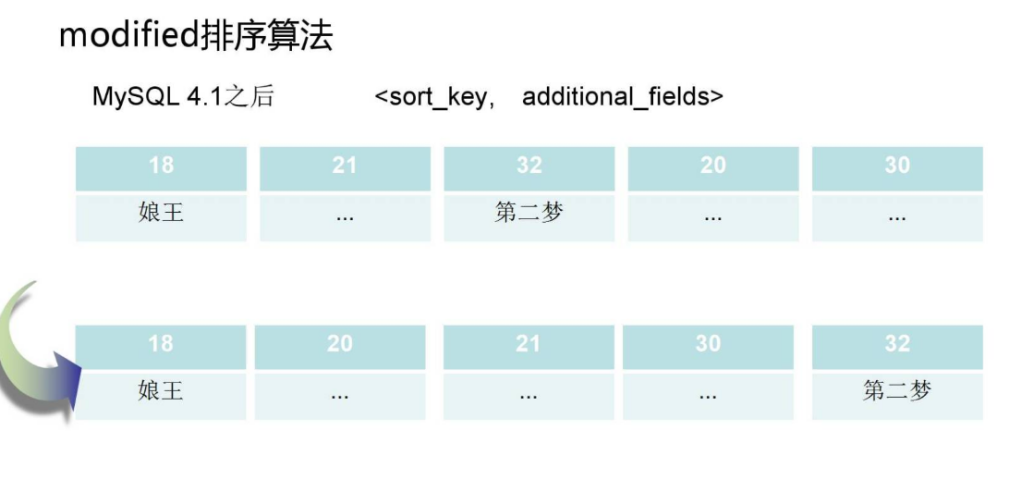
将要排序的列值和**用户需要返回的字段**组成键值对，存入sort buffer中；

如果sort buffer内存大于这些键值对的内存，就不需要创建临时文件了。否则，每次sort buffer填满以后，需要直接用qsort(快速排序算法)在内存中排好序，并写到临时文件中；

重复上述步骤，直到所有的行数据都正常读取了完成；

用到了临时文件的，需要利用磁盘外部排序，将排序后的数据写入到结果文件中；

直接从结果文件中返回用户需要的字段数据，而不是根据row ID再次回表查询。



## 打包数据排序模式

第三种排序模式的改进仅仅在于将char和varchar字段存到sort buffer中时，更加紧缩。

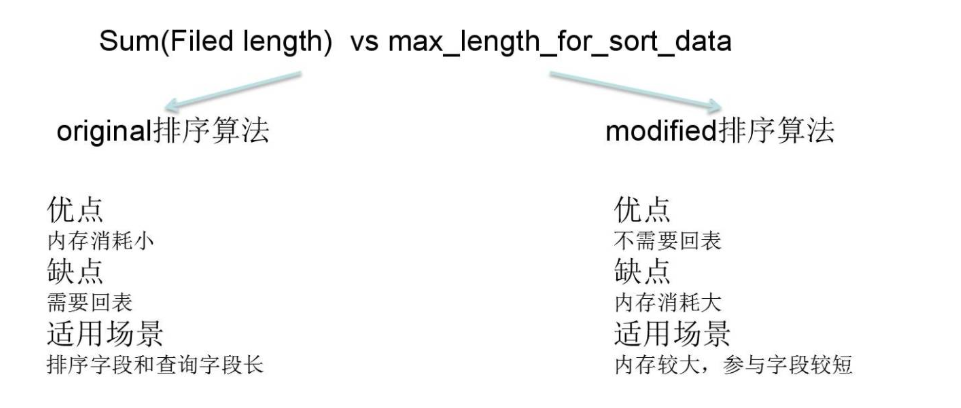
在之前的两种模式中，存储了”yes”3个字符的定义为VARCHAR(255)的列会在内存中申请255个字符内存空间，但是5.7.3改进后，只需要存储2个字节的字段长度和3个字符内存空间（用于保存”yes”这三个字符）就够了，内存空间整整压缩了50多倍,可以让更多的键值对保存在sort buffer中。

## 三种模式比较

第二种模式是第一种模式的改进，避免了二次回表，采用的是用空间换时间的方法。

但是由于sort buffer就那么大，如果用户要查询的数据非常大的话，很多时间浪费在多次磁盘外部排序，导致更多的IO操作，效率可能还不如第一种方式。

所以，MySQL给用户提供了一个max\_length\_for\_sort\_data的参数。当“排序的键值对大小”> max\_length\_for\_sort\_data时，MySQL认为磁盘外部排序的IO效率不如回表的效率，会选择第一种排序模式；反之，会选择第二种不回表的模式。



第三种模式主要是解决变长字符数据存储空间浪费的问题，对于实际数据不多，字段定义较长的改进效果会更加明显。

很多文章写到这里可能就差不多了，但是大家忘记关注一个问题了：“如果排序的数据不能完全放在sort buffer内存里面，是怎么通过外部排序完成整个排序过程的呢？”要解决这个问题，我们首先需要简单查看一下外部排序到底是怎么做的。

# 外部排序

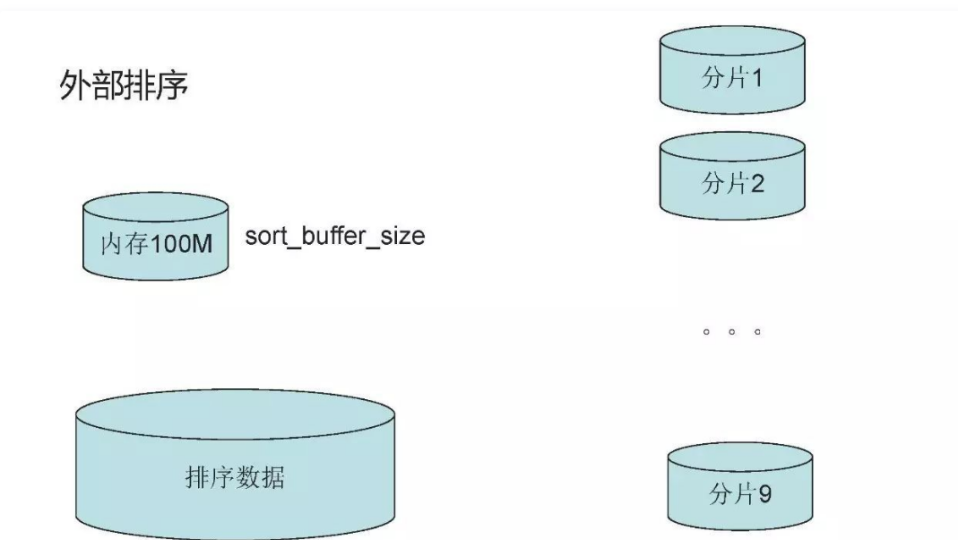
## 普通外部排序

### 两路外部排序

我们先来看一下最简单，最普遍的两路外部排序算法。

假设内存只有100M，但是排序的数据有900M，那么对应的外部排序算法如下：

1. 从要排序的900M数据中读取100MB数据到内存中，并按照传统的内部排序算法（快速排序）进行排序；
2. 将排序好的数据写入磁盘；
3. 重复1，2两步，直到每个100MB chunk大小排序好的数据都被写入磁盘；
4. 每次读取排序好的chunk中前10MB（= 100MB / (9 chunks + 1)）数据，一共9个chunk需要90MB，剩下的10MB作为输出缓存；
5. 对这些数据进行一个“9路归并”，并将结果写入输出缓存。如果输出缓存满了，则直接写入最终排序结果文件并清空输出缓存；如果9个10MB的输入缓存空了，从对应的文件再读10MB的数据，直到读完整个文件。最终输出的排序结果文件就是900MB排好序的数据了。

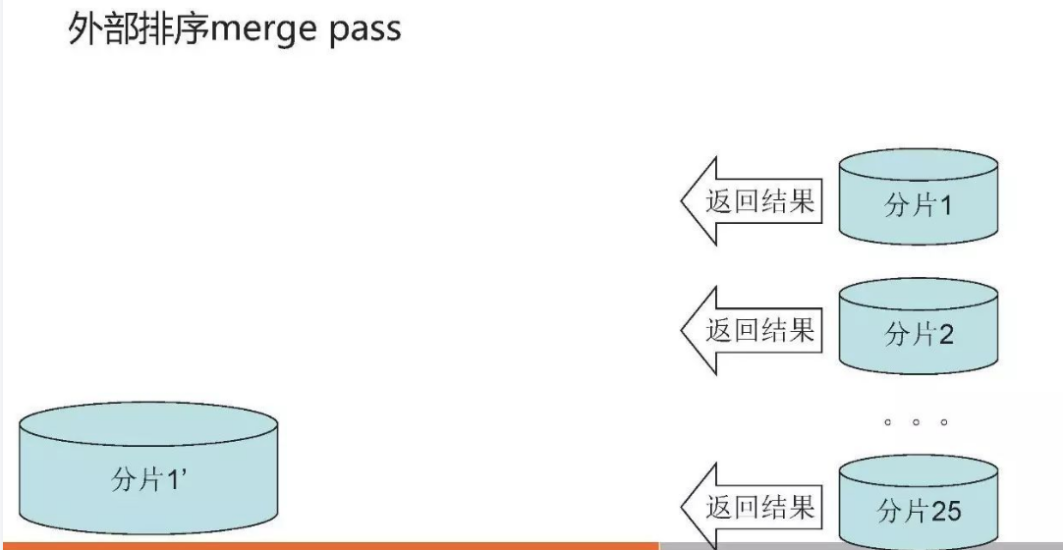


### 多路外部排序

上述排序算法是一个两路排序算法（先排序，后归并）。但是这种算法有一个问题，假设要排序的数据是50GB而内存只有100MB，那么每次从500个排序好的分片中取200KB（100MB / 501 约等于200KB）就是很多个随机IO。效率非常慢，对应可以这样来改进：

1. 从要排序的50GB数据中读取100MB数据到内存中，并按照传统的内部排序算法（快速排序）进行排序；
2. 将排序好的数据写入磁盘；
3. 重复1，2两步，直到每个100MB chunk大小排序好的数据都被写入磁盘；
4. 每次取25个分片进行归并排序，这样就形成了20个（500/25=20）更大的2.5GB有序的文件；
5. 对这20个2.5GB的有序文件进行归并排序，形成最终排序结果文件。

对应的数据量更大的情况可以进行更多次归并。



## MySQL外部排序

### MySQL外部排序算法

那MySQL使用的外部排序是怎么样的列，我们以回表排序模式为例：

1. 根据索引或者全表扫描，按照过滤条件获得需要查询的数据；
2. 将要排序的列值和row ID组成键值对，存入sort buffer中；
3. 如果sort buffer内存大于这些键值对的内存，就不需要创建临时文件了。否则，每次sort buffer填满以后，需要直接用qsort(快速排序模式)在内存中排好序，作为一个block写到临时文件中。跟正常的外部排序写到多个文件中不一样，MySQL只会写到一个临时文件中，并通过保存文件偏移量的方式来模拟多个文件归并排序；
4. 重复上述步骤，直到所有的行数据都正常读取了完成；
5. 每MERGEBUFF (7) 个block抽取一批数据进行排序，归并排序到另外一个临时文件中，直到所有的数据都排序好到新的临时文件中；
6. 重复以上归并排序过程，直到剩下不到MERGEBUFF2 (15)个block。

通俗一点解释：

第一次循环中，一个block对应一个sort buffer（大小为sort\_buffer\_size）排序好的数据；每7个做一个归并。

第二次循环中，一个block对应MERGEBUFF (7) 个sort buffer的数据，每7个做一个归并。

…

直到所有的block数量小于MERGEBUFF2 (15)。

1. 最后一轮循环，仅将row ID写入到结果文件中；
2. 根据结果文件中的row ID按序读取用户需要返回的数据。为了进一步优化性能，MySQL会读一批row ID，并将读到的数据按排序字段要求插入缓存区中(内存大小read\_rnd\_buffer\_size)。

这里我们需要注意的是：

MySQL把外部排序好的分片写入同一个文件中，通过保存文件偏移量的方式来区别各个分片位置；

MySQL每MERGEBUFF (7)个分片做一个归并，最终分片数达到MERGEBUFF2 (15)时，做最后一次归并。

### sort\_merge\_passes

MySQL手册中对Sort\_merge\_passes的描述只有一句话

Sort\_merge\_passes

The number of merge passes that the sort algorithm has had to do. If this value is large, you should consider increasing the value of the sort\_buffer\_size system variable.

这段话并没有把sort\_merge\_passes到底是什么，该值比较大时说明了什么，通过什么方式可以缓解这个问题。 我们把上面MySQL的外部排序算法搞清楚了，这个问题就清楚了。

其实sort\_merge\_passes对应的就是MySQL做归并排序的次数，也就是说，如果sort\_merge\_passes值比较大，说明sort\_buffer和要排序的数据差距越大，我们可以通过增大sort\_buffer\_size或者让填入sort\_buffer\_size的键值对更小来缓解sort\_merge\_passes归并排序的次数。

对应的，我们可以在源码中看到证据。

上述MySQL外部排序的算法中第5到第7步，是通过sql/filesort.cc文件中merge\_many\_buff()函数来实现，第5步单次归并使用merge\_buffers()实现，源码摘录如下：

int merge\_many\_buff(Sort\_param \*param, Sort\_buffer sort\_buffer,

Merge\_chunk\_array chunk\_array,

size\_t \*p\_num\_chunks, IO\_CACHE \*t\_file){...

for (i=0 ; i < num\_chunks - MERGEBUFF \* 3 / 2 ; i+= MERGEBUFF)

{

if (merge\_buffers(param, // param

from\_file, // from\_file

to\_file, // to\_file

sort\_buffer, // sort\_buffer

last\_chunk++, // last\_chunk [out]

Merge\_chunk\_array(&chunk\_array[i], MERGEBUFF),

0)) // flag

goto cleanup;

}

if (merge\_buffers(param,

from\_file,

to\_file,

sort\_buffer,

last\_chunk++,

Merge\_chunk\_array(&chunk\_array[i], num\_chunks - i),

0))

break; /\* purecov: inspected \*/...}

截取部分merge\_buffers()的代码如下，

int merge\_buffers(Sort\_param \*param, IO\_CACHE \*from\_file,

IO\_CACHE \*to\_file, Sort\_buffer sort\_buffer,

Merge\_chunk \*last\_chunk,

Merge\_chunk\_array chunk\_array,

int flag){...

current\_thd->inc\_status\_sort\_merge\_passes();...}

可以看到：每个merge\_buffers()都会增加sort\_merge\_passes，也就是说每一次对MERGEBUFF (7) 个block归并排序都会让sort\_merge\_passes加一，sort\_merge\_passes越多表示排序的数据太多，需要多次merge pass。解决的方案无非就是缩减要排序数据的大小或者增加sort\_buffer\_size。

## trace结果解释

说明白了三种排序模式和外部排序的方法，我们回过头来看一下trace的结果。

### 是否存在磁盘外部排序

"number\_of\_tmp\_files": 0,

number\_of\_tmp\_files表示有多少个分片，如果number\_of\_tmp\_files不等于0，表示一个sort\_buffer\_size大小的内存无法保存所有的键值对，也就是说，MySQL在排序中使用到了磁盘来排序。

### 是否存在优先队列优化排序

由于我们的这个SQL里面没有对数据进行分页限制，所以filesort\_priority\_queue\_optimization并没有启用

"filesort\_priority\_queue\_optimization": {

"usable": false,

"cause": "not applicable (no LIMIT)"

},

而正常情况下，使用了Limit会启用优先队列的优化。优先队列类似于FIFO先进先出队列。

算法稍微有点改变，以回表排序模式为例。

**sort\_buffer\_size足够大**

如果Limit限制返回N条数据，并且N条数据比sort\_buffer\_size小，那么MySQL会把sort buffer作为priority queue，在第二步插入priority queue时会按序插入队列；在第三步，队列满了以后，并不会写入外部磁盘文件，而是直接淘汰最尾端的一条数据，直到所有的数据都正常读取完成。

**算法如下：**

1. 根据索引或者全表扫描，按照过滤条件获得需要查询的数据
2. 将要排序的列值和row ID组成键值对，按序存入中priority queue中
3. 如果priority queue满了，直接淘汰最尾端记录
4. 重复上述步骤，直到所有的行数据都正常读取了完成
5. 最后一轮循环，仅将row ID写入到结果文件中
6. 根据结果文件中的row ID按序读取用户需要返回的数据。为了进一步优化性能，MySQL会读一批row ID，并将读到的数据按排序字段要求插入缓存区中(内存大小read\_rnd\_buffer\_size)。

**sort\_buffer\_size不够大**

否则，N条数据比sort\_buffer\_size大的情况下，MySQL无法直接利用sort buffer作为priority queue，正常的文件外部排序还是一样的，只是在最后返回结果时，只根据N个row ID将数据返回出来。具体的算法我们就不列举了。

这里MySQL到底是否选择priority queue是在sql/filesort.cc的check\_if\_pq\_applicable()函数中确定的，具体的代码细节这里就不展开了。

另外，我们也没有讨论limit m,n的情况，如果是Limit m,n， 上面对应的“N个row ID”就是“M+N个row ID”了，MySQL的limit m,n 其实是取m+n行数据，最后把M条数据丢掉。

从上面我们也可以看到sort\_buffer\_size足够大对limit数据比较小的情况，优化效果是很明显的。

# 原理

MySQL对排序有两种实现：

## 双路排序

### 原理

第一遍扫描出需要排序的字段，然后进行排序后，根据排序结果，第二遍再扫描一下需要select的列数据。这样会引起大量的**随机IO**，效率不高，但是节约内存。排序使用quick sort。但是如果内存不够则会按照block进行排序，将排序结果写入磁盘文件，然后再将结果合并。

注：查询中随机IO会降低效率，在OceanBase等NewSQL架构中采用LSM将随机IO转换为顺序IO，提高访问效率。

**具体过程：**

1、读取所有满足条件的记录。

2、对于每一行，存储一对值到缓冲区**（排序列，行记录指针）**，一个是排序的索引列的值，即order by用到的列值，和指向该行数据的行指针（缓冲区的大小为sort\_buffer\_size大小）。

3、当缓冲区满后，运行一个快速排序（qsort）来将缓冲区中数据排序，并将排序完的数据存储到一个临时文件，并保存一个存储块的指针，当然如果缓冲区不满，则不会重建临时文件了。

4、重复以上步骤，直到将所有行读完，并建立相应的有序的临时文件。

5、对块级进行排序，这个类似于归并排序算法，只通过两个临时文件的指针来不断交换数据，最终达到两个文件，都是有序的。

6、重复5直到所有的数据都排序完毕。

7、采取顺序读的方式，将每行数据读入内存，并取出数据传到客户端，这里读取数据时并不是一行一行读，读如缓存大小由read\_rnd\_buffer\_size来指定。

### 特点

采取的方法为：快速排序 + 归并排序。

但有一个问题，就是，一行数据会被读两次，第一次是where条件过滤时，第二个是排完序后还得用行指针去读一次，一个优化的方法是，直接读入数据，排序的时候也根据这个排序，排序完成后，就直接发送到客户端了。

## 单路排序

在MySQL4.1版本之前只有第一种排序算法双路排序，第二种算法是从MySQL4.1开始的改进算法，主要目的是为了减少第一次算法中需要两次访问表数据的IO操作，将两次变成了一次，但相应也会耗用更多的sortbuffer空间。当然，MySQL4.1开始的以后所有版本同时也支持第一种算法。

### 原理

即**一遍扫描数据后将select需要的列数据以及排序的列数据都取出来，然后在sort buffer中排序**，这样就不需要进行第二遍扫描了，当然内存不足时也会使用磁盘临时文件进行外排。

过程如下：

1、读取满足条件的记录

2、对于每一行，记录排序的key和数据行指针，并且把要查询的列也读出来

3、根据索引key排序

4、读取排序完成的文件，并直接根据数据位置读取数据返回客户端，而不是去访问表

### 特点

单路排序一次性将结果读取出来，然后在sort buffer中排序，避免了双路排序的两次读的随机IO。

这也有一个问题：当获取的列很多的时候，排序起来就很占空间，因此，max\_length\_for\_sort\_data变量就决定了是否能使用这个排序算法。

MySQL根据sort\_buffer\_size来判断是否使用磁盘临时文件，如果需要排序的数据能放入sort\_buffer\_size则无需使用磁盘临时文件，此时explain只会输出using filesort否则需要使用磁盘临时文件explain会输出using temporary;using filesort。

# 参数

## max\_sort\_length

这里需要区别max\_sort\_length和max\_length\_for\_sort\_data。

max\_length\_for\_sort\_data是为了让MySQL选择”< sort\_key, rowid >”还是”< sort\_key, additional\_fields >”的模式。

而max\_sort\_length是键值对的大小无法确定时（比如用户要查询的数据包含了SUBSTRING\_INDEX(col1, ‘.’,2)）MySQL会对每个键值对分配max\_sort\_length个字节的内存，这样导致内存空间浪费，磁盘外部排序次数过多。

## innodb\_disable\_sort\_file\_cache

innodb\_disable\_sort\_file\_cache设置为ON的话，表示在排序中生成的临时文件不会用到文件系统的缓存，类似于O\_DIRECT打开文件。

## innodb\_sort\_buffer\_size

这个参数其实跟我们这里讨论的SQL排序没有什么关系。innodb\_sort\_buffer\_size设置的是在创建InnoDB索引时，使用到的sort buffer的大小。

以前写死为1M，现在开放出来，允许用户自定义设置这个参数了。

# 选择

MySQL主要通过比较我们所设定的系统参数max\_length\_for\_sort\_data的大小和Query语句所取出的字段类型大小总和来判定需要使用哪一种排序算法。

如果需要的列数据一行可以放入max\_length\_for\_sort\_data则使用一遍扫描否则使用两遍扫描（如果max\_length\_for\_sort\_data更大，则使用第二种优化后的算法，反之使用第一种算法）。所以如果希望ORDER BY操作的效率尽可能的高，一定要注意max\_length\_for\_sort\_data参数的设置。

如果数据库出现大量的排序等待，造成系统负载很高，而且响应时间变得很长，可以考虑是否为MySQL 使用了传统的第一种排序算法而导致，在加大了max\_length\_for\_sort\_data参数值之后，系统负载是否马上得到了大的缓解，响应是否快很多。

# 优化

对于文件排序的优化，应该让MySQL避免使用第一种双路排序，尽量选择使用第二种单路算法来进行排序。这样可以减少大量的随机IO操作，很大幅度地提高排序工作的效率。

1、加大max\_length\_for\_sort\_data参数的设置

在MySQL中，决定使用老的双路排序算法还是改进版单路排序算法是通过参数max\_length\_for\_ sort\_data来决定的。当所有返回字段的最大长度小于这个参数值时，MySQL就会选择改进后的单路排序算法，反之，则选择老式的双路排序算法。所以，如果有充足的内存让MySQL存放需要返回的非排序字段，就可以加大这个参数的值来让MySQL选择使用改进版的排序算法。

2、去掉不必要的返回字段或列长度尽量小一些

对于内存不是非常充裕的情况，不能强行增大配置项max\_length\_for\_sort\_data，否则可能会造成MySQL不得不将数据分成很多段，然后进行排序，这样可能会得不偿失。此时可以选择去掉不必要的返回字段或者将列长度尽可能设置小一些，让返回结果长度适应max\_length\_for\_sort\_data参数的限制。

3、增大sort\_buffer\_size参数设置

增大sort\_buffer\_size并不是为了让MySQL选择改进版的单路排序算法，而是为了让MySQL尽量**减少在排序过程中对需要排序的数据进行分段**，因为分段会造成MySQL不得不使用临时表来进行交换排序。

如果大量的查询较小的话，这个很好，就缓存中就搞定了。

4、增加read\_rnd\_buffer\_size大小，可以一次性多读到内存中

该变量可以被任何存储引擎使用，当从一个已经排序的键值表中读取行时，会先从该缓冲区中获取而不再从磁盘上获取。默认为256K。

5、改变tmpdir，使其指向多个物理盘（不是分区）的目录，这将机会循环使用做为临时文件区。

tmpdir建议独立存放，放在高速存储设备上。

# 总结

当看到MySQL的explain输出using filesort时，说明排序时没有使用索引。如果输出using temporary;using filesort则说明使用文件排序和磁盘临时表，这种情况需要引起注意，效率会比较低。

总结来说，尽量避免出现文件排序，如果出现using filesort需要考虑优化。