**SELECT a.PlateNo,**

**a.DeviceID,**

**a.AppearTime AS A,**

**b.PlateNo,**

**b.AppearTime AS B,**

**b.DeviceID**

**FROM (SELECT AppearTime,**

**PlateNo,**

**DeviceID**

**FROM motorvehicle**

**WHERE PlateNo='川AG331E') a**

**JOIN (SELECT AppearTime,**

**PlateNo,**

**DeviceID**

**FROM motorvehicle**

**WHERE PlateNo<>'川AG331E') b**

**ON a.DeviceID = b.DeviceID**

**WHERE**

**CASE** *#SQL条件语句CASE 条件 WHEN 分支1 WHEN 分支2...ELSE 默认分支 END;CASE WHEN 分支1 WHEN 分支2..ELSE 默认分支 END*

**WHEN a.AppearTime < b.AppearTime**

**THEN TIMESTAMPDIFF(SECOND,a.AppearTime,b.AppearTime)<60**

**WHEN a.AppearTime > b.AppearTime**

**THEN TIMESTAMPDIFF(SECOND,b.AppearTime,a.AppearTime)<60***#TIMESTAMPDIFF(unit,datetime\_expr1,datetime\_expr2)对于比较的两个时间，时间小的放在前面，时间大的放在后面。*

**END**

**ORDER BY b.PlateNo;**

**SELECT b.PlateNo,**

**COUNT(b.PlateNo)**

**FROM (SELECT AppearTime,**

**PlateNo,**

**DeviceID**

**FROM motorvehicle**

**WHERE PlateNo='川AG331E') a**

**JOIN (SELECT AppearTime,**

**PlateNo,**

**DeviceID**

**FROM motorvehicle**

**WHERE PlateNo<>'川AG331E') b**

**ON a.DeviceID = b.DeviceID**

**WHERE**

**CASE** *#SQL条件语句CASE 条件 WHEN 分支1 WHEN 分支2...ELSE 默认分支 END;CASE WHEN 分支1 WHEN 分支2..ELSE 默认分支 END*

**WHEN a.AppearTime < b.AppearTime**

**THEN TIMESTAMPDIFF(SECOND,a.AppearTime,b.AppearTime)<60**

**WHEN a.AppearTime > b.AppearTime**

**THEN TIMESTAMPDIFF(SECOND,b.AppearTime,a.AppearTime)<60***#TIMESTAMPDIFF(unit,datetime\_expr1,datetime\_expr2)对于比较的两个时间，时间小的放在前面，时间大的放在后面。*

**END**

**GROUP BY b.PlateNo**

**ORDER BY b.PlateNo;**

**SELECT b.PlateNo,**

**COUNT(b.PlateNo) AS count**

**FROM (SELECT AppearTime,**

**PlateNo,**

**DeviceID**

**FROM motorvehicle**

**WHERE PlateNo='川AG331E') a**

**JOIN (SELECT AppearTime,**

**PlateNo,**

**DeviceID**

**FROM motorvehicle**

**WHERE PlateNo<>'川AG331E') b**

**ON a.DeviceID = b.DeviceID**

**WHERE**

**CASE** *#SQL条件语句CASE 条件 WHEN 分支1 WHEN 分支2...ELSE 默认分支 END;CASE WHEN 分支1 WHEN 分支2..ELSE 默认分支 END*

**WHEN a.AppearTime < b.AppearTime**

**THEN TIMESTAMPDIFF(SECOND,a.AppearTime,b.AppearTime)<60**

**WHEN a.AppearTime > b.AppearTime**

**THEN TIMESTAMPDIFF(SECOND,b.AppearTime,a.AppearTime)<60***#TIMESTAMPDIFF(unit,datetime\_expr1,datetime\_expr2)对于比较的两个时间，时间小的放在前面，时间大的放在后面。*

**END**

**GROUP BY b.PlateNo**

**HAVING count>=6**

**ORDER BY count DESC;**

[**mysql having的用法**](https://www.cnblogs.com/lmaster/p/6373045.html)

having的用法

having字句可以让我们筛选成组后的各种数据，where字句在聚合前先筛选记录，也就是说作用在group by和having字句前。而 having子句在聚合后对组记录进行筛选。我的理解就是真实表中没有此数据，这些数据是通过一些函数生存。

SQL实例：

一、显示每个地区的总人口数和总面积．  
SELECT region, SUM(population), SUM(area) FROM bbc GROUP BY region

先以region把返回记录分成多个组，这就是GROUP BY的字面含义。分完组后，然后用聚合函数对每组中  
的不同字段（一或多条记录）作运算。

二、 显示每个地区的总人口数和总面积．仅显示那些面积超过1000000的地区。

SELECT region, SUM(population), SUM(area)  
FROM bbc  
GROUP BY region  
HAVING SUM(area)>1000000

在这里，我们不能用where来筛选超过1000000的地区，因为表中不存在这样一条记录。  
相反，having子句可以让我们筛选成组后的各组数据

[MySQL](http://lib.csdn.net/base/mysql)判断某个字段的长度：

select home\_page from aaa表 where char\_length(trim(home\_page))<10 and char\_length(trim(home\_page))>1;

**mysql中的where和having子句的区别**

mysql中的where和having子句都可以实现过滤记录的功能,但他们的用法还是有一些区别的,看一例子:  
用group by和having子句联合来查出不重复的记录,sql如下:  
select uid,email,count(\*) as ct from `edm\_user081217` GROUP BY email  
然后看这个，就容易理解了  
select uid,email,count(\*) as ct from `edm\_user081217` GROUP BY email HAVING ct > 1  
先用group by 对email进行分组,在用having来过滤大于1的,这样查找出来的就是重复的记录了.  
  
以下是having和where的区别：  
Select city FROM weather WHERE temp\_lo = (SELECT max(temp\_lo) FROM weather);  
作用的对象不同。WHERE 子句作用于表和视图，HAVING 子句作用于组。  
WHERE 在分组和聚集计算之前选取输入行（因此，它控制哪些行进入聚集计算）， 而 HAVING 在分组和聚集之后选取分组的行。因此，WHERE 子句不能包含聚集函数； 因为试图用聚集函数判断那些行输入给聚集运算是没有意义的。 相反，HAVING 子句总是包含聚集函数。（严格说来，你可以写不使用聚集的 HAVING 子句， 但这样做只是白费劲。同样的条件可以更有效地用于 WHERE 阶段。）  
在前面的例子里，我们可以在 WHERE 里应用城市名称限制，因为它不需要聚集。 这样比在 HAVING 里增加限制更加高效，因为我们避免了为那些未通过 WHERE 检查的行进行分组和聚集计算  
综上所述：  
having一般跟在group by之后，执行记录组选择的一部分来工作的。  
where则是执行所有数据来工作的。  
再者having可以用聚合函数，如having sum(qty)>1000

2017-07-05 12:03:48更新：

我们来对比一下

SELECT ip,ip1,ip2,ip3,ip4,count(\*) ct,mobile\_info FROM `lmaster\_log`

GROUP BY ip1,ip2,ip3 ORDER BY ip1,ip2,ip3,ip4 HAVING ct > **2**;

SELECT ip,ip1,ip2,ip3,ip4,count(\*) ct,mobile\_info FROM `lmaster\_log`

GROUP BY ip1,ip2,ip3 HAVING ct > **2** ORDER BY ip1,ip2,ip3,ip4;

一个语句

[Err] 1064 - You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'HAVING ct > 2' at line 2

语句2是出现结果，所以我们在使用order by时要注意，他的位置

MYSQL中如何强制终止一条语句的执行

KILL命令的语法格式如下：KILL [CONNECTION | QUERY] thread\_id

步骤如下：

1、KILL允许自选的CONNECTION或QUERY修改符：KILL CONNECTION与不含修改符的KILL一样：它会终止与给定的thread\_id有关的连接。

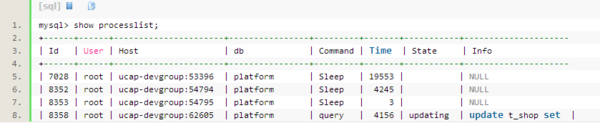
2、KILL QUERY会终止连接当前正在执行的语句，但是会保持连接的原状。

3、如果您拥有PROCESS权限，则您可以查看所有线程。

4、如果您拥有超级管理员权限，您可以终止所有线程和语句。否则，您只能查看和终止您自己的线程和语句。

5、您也可以使用mysqladmin processlist和mysqladmin kill命令来检查和终止线程。

首先登录mysql，然后使用： show processlist; 查看当前mysql中各个线程状态。

[](https://gss0.baidu.com/-4o3dSag_xI4khGko9WTAnF6hhy/zhidao/pic/item/c83d70cf3bc79f3d6483cbdfb1a1cd11738b29fa.jpg)

以上显示出当前正在执行的sql语句列表，找到消耗资源最大的那条语句对应的id.

然后运行kill命令，命令格式如下：

[sql] view plain copy

kill id;

- 示例:

kill 8358

杀掉即可。

Mysql进行操作以后没有commit会造成死锁.

**mysql进程SHOW PROCESSLIST详解Command命令状态**

SHOW PROCESSLIST显示哪些线程正在运行。您也可以使用**mysqladmin processlist**语句得到此信息。如果您有SUPER权限，您可以看到所有线程。否则，您只能看到您自己的线程（也就是，与您正在使用的[**MySQL**](http://lib.csdn.net/base/mysql)账户相关的线程）。请参见[13.5.5.3节，“KILL语法”](http://blog.csdn.net/dbadream/article/details/9427423)。如果您不使用FULL关键词，则只显示每个查询的前100个字符。

本语句报告TCP/IP连接的主机名称（采用host\_name:client\_port格式），以方便地判定哪个客户端正在做什么。

如果您得到“too many connections”错误信息，并且想要了解正在发生的情况，本语句是非常有用的。MySQL保留一个额外的连接，让拥有SUPER权限的 账户使用，以确保管理员能够随时连接和检查系统（假设您没有把此权限给予所有的用户）。

这个命令中最关键的就是state列，mysql列出的状态主要有以下几种：

**Checking table**　正在检查数据表（这是自动的）。  
**Closing tables**  
　正在将表中修改的数据刷新到磁盘中，同时正在关闭已经用完的表。这是一个很快的操作，如果不是这样的话，就应该确认磁盘空间是否已经满了或者磁盘是否正处于重负中。  
**Connect Out**  
　复制从服务器正在连接主服务器。  
**Copying to tmp table on disk**  
　由于临时结果集大于tmp\_table\_size，正在将临时表从内存存储转为磁盘存储以此节省内存。  
**Creating tmp table**  
　正在创建临时表以存放部分查询结果。  
**deleting from main table**  
　服务器正在执行多表删除中的第一部分，刚删除第一个表。  
**deleting from reference tables**　服务器正在执行多表删除中的第二部分，正在删除其他表的记录。  
**Flushing tables**　正在执行FLUSH TABLES，等待其他线程关闭数据表。  
**Killed**　发送了一个kill请求给某线程，那么这个线程将会检查kill标志位，同时会放弃下一个kill请求。MySQL会在每次的主循环中检查kill标志位，不过有些情况下该线程可能会过一小段才能死掉。如果该线程程被其他线程锁住了，那么kill请求会在锁释放时马上生效。  
**Locked**

被其他查询锁住了。  
**Sending data**  
　正在处理SELECT查询的记录，同时正在把结果发送给客户端。  
**Sorting for group**　正在为GROUP BY做排序。  
　Sorting for order  
　正在为ORDER BY做排序。  
**Opening tables**  
　这个过程应该会很快，除非受到其他因素的干扰。例如，在执ALTER TABLE或LOCK TABLE语句行完以前，数据表无法被其他线程打开。正尝试打开一个表。  
**Removing duplicates**  
　正在执行一个SELECT DISTINCT方式的查询，但是MySQL无法在前一个阶段优化掉那些重复的记录。因此，MySQL需要再次去掉重复的记录，然后再把结果发送给客户端。  
**Reopen table**  
　获得了对一个表的锁，但是必须在表结构修改之后才能获得这个锁。已经释放锁，关闭数据表，正尝试重新打开数据表。  
**Repair by sorting**  
　修复指令正在排序以创建索引。  
**Repair with keycache**  
　修复指令正在利用索引缓存一个一个地创建新索引。它会比Repair by sorting慢些。  
**Searching rows for update**　正在讲符合条件的记录找出来以备更新。它必须在UPDATE要修改相关的记录之前就完成了。  
**Sleeping**  
　正在等待客户端发送新请求.  
**System lock**  
　正在等待取得一个外部的系统锁。如果当前没有运行多个mysqld服务器同时请求同一个表，那么可以通过增加--skip-external-locking参数来禁止外部系统锁。  
**Upgrading lock**  
　INSERT DELAYED正在尝试取得一个锁表以插入新记录。  
**Updating**  
　正在搜索匹配的记录，并且修改它们。

**User Lock**  
　正在等待GET\_LOCK()。  
**Waiting for tables**　该线程得到通知，数据表结构已经被修改了，需要重新打开数据表以取得新的结构。然后，为了能的重新打开数据表，必须等到所有其他线程关闭这个表。以下几种情况下会产生这个通知：FLUSH TABLES tbl\_name, ALTER TABLE, RENAME TABLE, REPAIR TABLE, ANALYZE TABLE,或OPTIMIZE TABLE。  
**waiting for handler insert**  
　INSERT DELAYED已经处理完了所有待处理的插入操作，正在等待新的请求。  
　大部分状态对应很快的操作，只要有一个线程保持同一个状态好几秒钟，那么可能是有问题发生了，需要检查一下。  
　还有其他的状态没在上面中列出来，不过它们大部分只是在查看服务器是否有存在错误是才用得着。

**mysql 查看当前连接数**

命令： show processlist;   
如果是root帐号，你能看到所有用户的当前连接。如果是其它普通帐号，只能看到自己占用的连接。   
show processlist;只列出前100条，如果想全列出请使用show full processlist;   
mysql> show processlist;

命令： show status;

Aborted\_clients 由于客户没有正确关闭连接已经死掉，已经放弃的连接数量。   
Aborted\_connects 尝试已经失败的MySQL服务器的连接的次数。   
Connections 试图连接MySQL服务器的次数。   
Created\_tmp\_tables 当执行语句时，已经被创造了的隐含临时表的数量。   
Delayed\_insert\_threads 正在使用的延迟插入处理器线程的数量。   
Delayed\_writes 用INSERT DELAYED写入的行数。   
Delayed\_errors 用INSERT DELAYED写入的发生某些错误(可能重复键值)的行数。   
Flush\_commands 执行FLUSH命令的次数。   
Handler\_delete 请求从一张表中删除行的次数。   
Handler\_read\_first 请求读入表中第一行的次数。   
Handler\_read\_key 请求数字基于键读行。   
Handler\_read\_next 请求读入基于一个键的一行的次数。   
Handler\_read\_rnd 请求读入基于一个固定位置的一行的次数。   
Handler\_update 请求更新表中一行的次数。   
Handler\_write 请求向表中插入一行的次数。   
Key\_blocks\_used 用于关键字缓存的块的数量。   
Key\_read\_requests 请求从缓存读入一个键值的次数。   
Key\_reads 从磁盘物理读入一个键值的次数。   
Key\_write\_requests 请求将一个关键字块写入缓存次数。   
Key\_writes 将一个键值块物理写入磁盘的次数。

Max\_used\_connections 同时使用的连接的最大数目。   
Not\_flushed\_key\_blocks 在键缓存中已经改变但是还没被清空到磁盘上的键块。   
Not\_flushed\_delayed\_rows 在INSERT DELAY队列中等待写入的行的数量。   
Open\_tables 打开表的数量。   
Open\_files 打开文件的数量。   
Open\_streams 打开流的数量(主要用于日志记载）   
Opened\_tables 已经打开的表的数量。   
Questions 发往服务器的查询的数量。   
Slow\_queries 要花超过long\_query\_time时间的查询数量。   
Threads\_connected 当前打开的连接的数量。   
Threads\_running 不在睡眠的线程数量。   
Uptime 服务器工作了多少秒。

**java下的mysql数据库插入越插越慢的问题解决（百万数据量级别）**

1.分析是否是由主码，外码，索引造成的插入效率降低

主码：由于主码是每张表必须有的，不能删除。而mysql会对主码自动建立一个索引，这个索引默认是Btree索引，因此每次插入数据要额外的对Btree进行一次插入。这个额外的插入时间复杂度约为log(n)。这个索引无法删除，因此无法优化。但是每次插入的时候，由于主码约束需要检查主码是否出现，这又需要log(n)，能否减少这个开销呢？答案是肯定的。我们可以设置主码为自增id

AUTO\_INCREMENT ,这样数据库里会自动记录当前的自增值，保证不会插入重复的主码，也就避免了主码的重复性检查。

外码：由于我的项目的插入表中存在外码，因此每次插入时需要在另一张表检测外码存在性。这个约束是与业务逻辑相关的，不能随便删除。并且这个时间开销应当是与另一张表大小成正比的常数，不应当越插入越慢才对。所以排除。

索引：为了减少Btree插入的时间损耗，我们可以在建表时先不建索引，先将所有的数据插入。之后我们再向表里添加索引。该方法确实也降低了时间的开销。

经过以上的折腾，再进行测试，发现速度快了一点，但是到了50w条后又开始慢了。看来问题的关键不在这里。于是继续查资料，又发现了个关键问题：

2.将单条插入改为批量插入（参考：点击打开链接）

由于java中的executeUpdate(sql)方法只是执行一条sql操作，就需要调用sql里的各种资源，如果使用for循环不停的执行这个方法来插入，无疑是开销很大的。因此，在mysql提供了一种解决方案：批量插入。 也就是每次的一条sql不直接提交，而是先存在批任务集中，当任务集的大小到了指定阈值后，再将这些sql一起发送至mysql端。在100w的数据规模中，我将阈值设置为10000，即一次提交10000条sql。最后的结果挺好，插入的速度比之前快了20倍左右。批量插入代码如下：

public static void insertRelease() { Long begin = new Date().getTime(); String sql = "INSERT INTO tb\_big\_data (count, create\_time, random) VALUES (?, SYSDATE(), ?)"; try { conn.setAutoCommit(false); PreparedStatement pst = conn.prepareStatement(sql); for (int i = 1; i <= 100; i++) { for (int k = 1; k <= 10000; k++) { pst.setLong(1, k \* i); pst.setLong(2, k \* i); pst.addBatch(); } pst.executeBatch(); conn.commit(); } pst.close(); conn.close(); } catch (SQLException e) { e.printStackTrace(); } Long end = new Date().getTime(); System.out.println("cast : " + (end - begin) / 1000 + " ms"); }

3.一条UPDATE语句的VALUES后面跟上多条的(?,?,?,?)

这个方法一开始我觉得和上面的差不多，但是在看了别人做的实验后，发现利用这个方法改进上面的批量插入，速度能快5倍。后来发现，mysql的导出sql文件中，那些插入语句也是这样写的。。即UPDATE table\_name (a1,a2) VALUES (xx,xx),(xx,xx),(xx,xx)... 。也就是我们需要在后台自己进行一个字符串的拼接，注意由于字符串只是不停的往末尾插入，用StringBuffer能够更快的插入。下面是代码：

public static void insert() { // 开时时间 Long begin = new Date().getTime(); // sql前缀 String prefix = "INSERT INTO tb\_big\_data (count, create\_time, random) VALUES "; try { // 保存sql后缀 StringBuffer suffix = new StringBuffer(); // 设置事务为非自动提交 conn.setAutoCommit(false); // Statement st = conn.createStatement(); // 比起st，pst会更好些 PreparedStatement pst = conn.prepareStatement(""); // 外层循环，总提交事务次数 for (int i = 1; i <= 100; i++) { // 第次提交步长 for (int j = 1; j <= 10000; j++) { // 构建sql后缀 suffix.append("(" + j \* i + ", SYSDATE(), " + i \* j \* Math.random() + "),"); } // 构建完整sql String sql = prefix + suffix.substring(0, suffix.length() - 1); // 添加执行sql pst.addBatch(sql); // 执行操作 pst.executeBatch(); // 提交事务 conn.commit(); // 清空上一次添加的数据 suffix = new StringBuffer(); } // 头等连接 pst.close(); conn.close(); } catch (SQLException e) { e.printStackTrace(); } // 结束时间 Long end = new Date().getTime(); // 耗时 System.out.println("cast : " + (end - begin) / 1000 + " ms"); } 做了以上的优化后，我发现了一个很蛋疼的问题。虽然一开始的插入速度的确快了几十倍，但是插入了50w条数据后，插入速度总是会一下突然变的非常慢。这种插入变慢是断崖式的突变，于是我冥思苦想，无意中打开了系统的资源管理器，一看发现：java占用的内存在不断飙升。 突然脑海中想到：是不是内存溢出了？

4.及时释放查询结果

在我的数据库查询语句中，使用到了pres=con.prepareStatement(sql)来保存一个sql执行状态，使用了resultSet=pres.executeQuery来保存查询结果集。而在边查边插的过程中，我的代码一直没有把查询的结果给释放，导致其不断的占用内存空间。当我的插入执行到50w条左右时，我的内存空间占满了，于是数据库的插入开始不以内存而以磁盘为介质了，因此插入的速度就开始变得十分的低下。因此，我在每次使用完pres和resultSet后，加入了释放其空间的语句：resultSet.close();

pres.close(); 。重新进行测试，果然，内存不飙升了，插入数据到50w后速度也不降低了。原来问题的本质在这里！

---------------------

作者：shjyoudp

来源：CSDN

原文：https://blog.csdn.net/qq547276542/article/details/75097602

版权声明：本文为博主原创文章，转载请附上博文链接！