# SQL语句执行得很慢

## 分类讨论

一条SQL语句执行的很慢，那是每次执行都很慢呢？还是大多数情况下是正常的，偶尔出现很慢呢？所以我觉得，我们还得分以下两种情况来讨论。

1、大多数情况是正常的，只是偶尔会出现很慢的情况。

2、在数据量不变的情况下，这条SQL语句一直以来都执行的很慢。

针对这两种情况，我们来分析下可能是哪些原因导致的。

## 思路

一般应急调优的思路：针对突然的业务办理卡顿，无法进行正常的业务处理，需要马上解决的场景。

1、show processlist

2、explain select id ,name from stu where name='clsn'; # ALL id name age sex

select id,name from stu where id=2-1 函数 结果集>30;

　 show index from table;

3、通过执行计划判断，索引问题（有没有、合不合理）或者语句本身问题

4、show status like '%lock%'; #查询锁状态

　 kill SESSION\_ID; # 杀掉有问题的session

常规调优思路：针对业务周期性的卡顿，例如在每天 10-11 点业务特别慢，但是还能够使用，过了这段时间就好了。

1、查看slowlog，分析slowlog，分析出查询慢的语句；

2、按照一定优先级，一个一个排查所有慢语句；

3、分析top SQL，进行explain调试，查看语句执行时间；

4、调整索引或语句本身。

## 针对偶尔很慢的情况

一条SQL大多数情况正常，偶尔才能出现很慢的情况，针对这种情况，我觉得这条SQL语句的书写本身是没什么问题的，而是其他原因导致的，那会是什么原因呢？

### 数据库在刷新脏页

当我们要往数据库插入一条数据、或者要更新一条数据的时候，我们知道数据库会在内存中把对应字段的数据更新了，但是更新之后，这些更新的字段并不会马上同步持久化到磁盘中去，而是把这些更新的记录写入到redo log日志中去，等到空闲的时候，在通过redo log里的日志把最新的数据同步到磁盘中去。

当内存数据页跟磁盘数据页内容不一致的时候，我们称这个内存页为“脏页”。内存数据写入到磁盘后，内存和磁盘上的数据页的内容就一致了，称为“干净页”。

刷脏页有下面4种场景（后两种不用太关注“性能”问题）：

**redo log写满了**：redo log里的容量是有限的，如果数据库一直很忙，更新又很频繁，这个时候redo log很快就会被写满了，这个时候就没办法等到空闲的时候再把数据同步到磁盘的，只能暂停其他操作，全身心来把数据同步到磁盘中去的，而这个时候，就会导致我们平时正常的SQL语句突然执行的很慢，所以说，数据库在在同步数据到磁盘的时候，就有可能导致我们的SQL语句执行的很慢了。

**内存不够用了**：如果一次查询较多的数据，恰好碰到所查数据页不在内存中时，需要申请内存，而此时恰好内存不足的时候就需要淘汰一部分内存数据页，如果是干净页，就直接释放，如果恰好是脏页就需要刷脏页。

**MySQL认为系统“空闲”的时候**：这时系统没什么压力。

**MySQL正常关闭的时候**：这时候，MySQL会把内存的脏页都flush到磁盘上，这样下次 MySQL 启动的时候，就可以直接从磁盘上读数据，启动速度会很快。

### 获取锁失败

我们要执行的这条语句，刚好这条语句涉及到的表，别人在用，并且加锁了，我们拿不到锁，只能慢慢等待别人释放锁了。或者，表没有加锁，但要使用到的某个一行被加锁了，这个时候，也只能等待。

如果要判断是否真的在等待锁，我们可以用 show processlist这个命令来查看当前的状态。

## 针对一直都这么慢的情况

如果在数据量一样大的情况下，这条SQL语句每次都执行的这么慢，那就就要考虑下SQL书写，下面我们来分析下哪些原因会导致我们的SQL语句执行的很不理想。

我们先来假设我们有一个表，表里有下面两个字段，分别是主键id和两个普通字段c和d。

mysql> CREATE TABLE `t` (

`id` int(11) NOT NULL,

`c` int(11) DEFAULT NULL,

`d` int(11) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB;

### 未使用索引

没有用上索引，这个原因是很多人都能想到的，例如你要查询这条语句

select \* from t where 100 <c and c < 100000;

（1）、字段没有索引

刚好c字段上没有索引，那么只能走全表扫描了，所以，这会导致这条查询语句很慢。

（2）、字段有索引，但却没有用索引

给c这个字段加上了索引，然后又查询了一条语句

select \* from t where c - 1 = 1000;

这样子在查询的时候会用索引查询吗？

不会，如果我们在字段的左边做了运算，那么在查询的时候，就不会用上索引了，所以，大家要注意这种字段上有索引，但由于自己的疏忽，导致系统没有使用索引的情况了。

正确的查询应该如下

select \* from t where c = 1000 + 1;

有人可能会说，右边有运算就能用上索引？难道数据库就不会自动帮我们优化一下，自动把c - 1=1000自动转换为c = 1000+1。

确实不会帮你，所以，要注意。

（3）、函数操作导致没有用上索引

如果我们在查询的时候，对字段进行了函数操作，也是会导致没有用上索引的，例如

select \* from t where pow(c,2) = 1000;

这里我只是做一个例子，假设函数pow是求c的n次方，实际上可能并没有 pow(c,2)这个函数。其实这个和上面在左边做运算也是很类似的。

所以呢，一条语句执行都很慢的时候，可能是该语句没有用上索引了，不过具体是啥原因导致没有用上索引的呢，你就要会分析了，我上面列举的三个原因，应该是出现的比较多的吧。

### 数据库选错索引

我们在进行查询操作的时候，例如

select \* from t where 100 < c and c < 100000;

我们知道，主键索引和非主键索引是有区别的，主键索引存放的值是整行字段的数据，而非主键索引上存放的值不是整行字段的数据，而且存放主键字段的值。

也就是说，我们如果走c这个字段的索引的话，最后会查询到对应主键的值，然后，再根据主键的值走主键索引，查询到整行数据返回。

就算你在c字段上有索引，系统也并不一定会走c这个字段上的索引，而是有可能会直接扫描扫描全表，找出所有符合100 < c and c < 100000的数据。

**为什么会这样呢？**

其实是这样的，系统在执行这条语句的时候，会进行预测：究竟是走c索引扫描的行数少，还是直接扫描全表扫描的行数少呢？显然，扫描行数越少当然越好了，因为扫描行数越少，意味着I/O操作的次数越少。

如果是扫描全表的话，那么扫描的次数就是这个表的总行数了，假设为n；而如果走索引c的话，我们通过索引c找到主键之后，还得再通过主键索引来找我们整行的数据，也就是说，需要走两次索引。而且，我们也不知道符合100 c < and c < 10000这个条件的数据有多少行，万一这个表是全部数据都符合呢？这个时候意味着，走c索引不仅扫描的行数是n，同时还得每行数据走两次索引。

所以呢，**系统是有可能走全表扫描而不走索引的**。

**那系统是怎么判断呢？**

判断来源于系统的预测，也就是说，如果要走c字段索引的话，系统会预测走c字段索引大概需要扫描多少行。如果预测到要扫描的行数很多，它可能就不走索引而直接扫描全表了。

**系统是怎么预测判断的呢？**

系统是通过索引的区分度来判断的，一个索引上不同的值越多，意味着出现相同数值的索引越少，意味着索引的区分度越高。我们也把区分度称之为基数，即区分度越高，基数越大。所以呢，基数越大，意味着符合100 < c and c < 10000这个条件的行数越少。

所以呢，一个索引的基数越大，意味着走索引查询越有优势。

**那么问题来了，怎么知道这个索引的基数呢？**

系统当然是不会遍历全部来获得一个索引的基数的，代价太大了，索引系统是通过遍历部分数据，也就是通过采样的方式，来预测索引的基数的。

居然是采样，那就有可能出现失误的情况，也就是说，c这个索引的基数实际上是很大的，但是采样的时候，却很不幸，把这个索引的基数预测成很小。例如你采样的那一部分数据刚好基数很小，然后就误以为索引的基数很小。然后系统就不走c索引了，直接走全部扫描了。

所以，得出结论：由于统计的失误，导致系统没有走索引，而是走了全表扫描，而这，也是导致我们SQL语句执行的很慢的原因。

注意：**系统判断是否走索引，扫描行数的预测其实只是原因之一，这条查询语句是否需要使用使用临时表、是否需要排序等也是会影响系统的选择的**。

我们有时候也可以通过强制走索引的方式来查询，例如

select \* from t force index(a) where c < 100 and c < 100000;

我们也可以通过

show index from t;

来查询索引的基数和实际是否符合，如果和实际很不符合的话，我们可以重新来统计索引的基数，可以用这条命令

analyze table t;

来重新统计分析。

既然会预测错索引的基数，这也意味着，当我们的查询语句有多个索引的时候，系统有可能也会选错索引，这也可能是 SQL 执行的很慢的一个原因。

## 总结

一个 SQL 执行的很慢，我们要分两种情况讨论：

1、大多数情况下很正常，偶尔很慢，则有如下原因

(1)、数据库在刷新脏页，例如 redo log 写满了需要同步到磁盘。

(2)、执行的时候，遇到锁，如表锁、行锁。

2、这条 SQL 语句一直执行的很慢，则有如下原因。

(1)、没有用上索引：例如该字段没有索引；由于对字段进行运算、函数操作导致无法用索引。

(2)、数据库选错了索引。

# 分页查询很慢

当需要从数据库查询的表有上万条记录的时候，一次性查询所有结果会变得很慢，特别是随着数据量的增加特别明显，这时需要使用分页查询。对于数据库分页查询，也有很多种方法和优化的点。

## 背景

为了对下面列举的一些优化进行测试，下面针对已有的一张表进行说明。

表名：order\_history

描述：某个业务的订单历史表

主要字段：unsigned int id，tinyint(4) int type

字段情况：该表一共37个字段，不包含text等大型数据，最大为varchar(500)，id字段为索引，且为递增。

数据量：5709294

MySQL版本：5.7.16 线下找一张百万级的测试表可不容易，如果需要自己测试的话，可以写shell脚本什么的插入数据进行测试。以下的 sql 所有语句执行的环境没有发生改变，下面是基本测试结果：

select count(\*) from orders\_history;

返回结果：5709294

三次查询时间分别为：

8903 ms

8323 ms

8401 ms

## 一般分页查询

一般的分页查询使用简单的 limit 子句就可以实现。limit子句声明如下：

SELECT \* FROM table LIMIT [offset,] rows | rows OFFSET offset

LIMIT子句可以被用于指定SELECT语句返回的记录数。需注意以下几点：

第一个参数指定第一个返回记录行的偏移量，注意从 0开始

第二个参数指定返回记录行的最大数目

如果只给定一个参数：它表示返回最大的记录行数目

第二个参数为-1表示检索从某一个偏移量到记录集的结束所有的记录行

初始记录行的偏移量是0(而不是1)

下面是一个应用实例：

Select \* from orders\_history where type=8 limit 1000,10;

该条语句将会从表 orders\_history 中查询 offset:1000开始之后的10条数据，也就是第1001条到第1010条数据（ 1001<=id<=1010）。

数据表中的记录默认使用主键（一般为id）排序，上面的结果相当于：

Select \*from orders\_history where type=8 order by id limit 10000,10;

**三次查询时间分别为：**

3040 ms

3063 ms

3018 ms

针对这种查询方式，下面测试查询记录量对时间的影响：

select \*from orders\_history where type=8 limit 10000,1;

Select \* from orders\_history where type=8 limit 10000,10;

select \* from orders\_history where type=8 limit 10000,100;

select \* from orders\_history where type=8 limit 10000,1000;

select \* from orders\_history where type=8 limit 10000,10000;

**三次查询时间如下：**

查询1条记录：3072ms 3092ms 3002ms

查询10条记录：3081ms 3077ms 3032ms

查询100条记录：3118ms 3200ms 3128ms

查询1000条记录：3412ms 3468ms 3394ms

查询10000条记录：3749ms 3802ms 3696ms

另外还做了十来次查询，从查询时间来看，基本可以确定，在查询记录量低于100时，查询时间基本没有差距，随着查询记录量越来越大，所花费的时间也会越来越多。

针对查询偏移量的测试：

**三次查询时间如下：**

查询100偏移：25ms 24ms 24ms

查询1000偏移：78ms 76ms 77ms

查询10000偏移：3092ms 3212ms 3128ms

查询100000偏移：3878ms 3812ms 3798ms

查询1000000偏移：14608ms 14062ms 14700ms

随着查询偏移的增大，尤其查询偏移大于10万以后，查询时间急剧增加。

这种分页查询方式会从数据库第一条记录开始扫描，所以越往后，查询速度越慢，而且查询的数据越多，也会拖慢总查询速度。

## 使用子查询优化

这种方式先定位偏移位置的id，然后往后查询，这种方式适用于id递增的情况。

select \* from orders\_history where type=8 limit 100000,1;

select id from orders\_history where type=8 limit 100000,1;

select \* from orders\_history where type=8 and id>=(select id from orders\_history where type=8 limit 100000,1) limit 100;

select \* from orders\_history where type=8 limit 100000,100;

**4条语句的查询时间如下：**

第1条语句：3674ms

第2条语句：1315ms

第3条语句：1327ms

第4条语句：3710ms

**针对上面的查询需要注意：**

比较第1条语句和第2条语句：使用 select id 代替 select \* 速度增加了3倍

比较第2条语句和第3条语句：速度相差几十毫秒

比较第3条语句和第4条语句：得益于 select id 速度增加，第3条语句查询速度增加了3倍

这种方式相较于原始一般的查询方法，将会增快数倍。

## 使用 id 限定优化

这种方式假设数据表的id是连续递增的，则我们根据查询的页数和查询的记录数可以算出查询的id的范围，可以使用 id between and 来查询：

select \* from orders\_history where type=2 and id between 1000000 and 1000100 limit 100;

查询时间：15ms 12ms 9ms

这种查询方式能够极大地优化查询速度，基本能够在几十毫秒之内完成。限制是只能使用于明确知道id的情况，不过一般建立表的时候，都会添加基本的id字段，这为分页查询带来很多便利。

还可以有另外一种写法：

select \* from orders\_history where id >= 1000001 limit 100;

当然还可以使用 in 的方式来进行查询，这种方式经常用在多表关联的时候进行查询，使用其他表查询的id集合，来进行查询：

select \* from orders\_history where id in (select order\_id from trade\_2 where goods = 'pen') limit 100;

这种 in 查询的方式要注意：某些 mysql 版本不支持在 in 子句中使用 limit。

## 使用临时表优化

这种方式已经不属于查询优化，这儿附带提一下。

对于使用 id 限定优化中的问题，需要 id 是连续递增的，但是在一些场景下，比如使用历史表的时候，或者出现过数据缺失问题时，可以考虑使用临时存储的表来记录分页的id，使用分页的id来进行 in 查询。这样能够极大的提高传统的分页查询速度，尤其是数据量上千万的时候。

关于数据表的id说明

一般情况下，在数据库中建立表的时候，强制为每一张表添加 id 递增字段，这样方便查询。

如果像是订单库等数据量非常庞大，一般会进行分库分表。这个时候不建议使用数据库的 id 作为唯一标识，而应该使用分布式的高并发唯一 id 生成器来生成，并在数据表中使用另外的字段来存储这个唯一标识。

使用先使用范围查询定位 id （或者索引），然后再使用索引进行定位数据，能够提高好几倍查询速度。即先 select id，然后再 select \*；