MySQL性能调优

上次咱掰扯了一个非常牛逼的内存型数据库，叫做Redis的东西。但是，我们还是没办法完全摆脱传统的关系型数据库。至于为啥，主要就是这几个原因：

1，Redis还是不够安全，断电就完犊子了（当然，如果你们有Redis集群的话，这话当我没说）

2，对于一个主键不止一个数据的情况下，Redis就会变得非常难管理，比如说订单，一个用户的订单肯定不止一个，所以到底生成这个Key的时候应该拿什么字段去生成呢？这个就会很麻烦了（企业级开发里面会自己定义一个KeyGenerator，但是这样非常考验这个生成器的生成算法，而且大部分情况下，生成出来的Key已经失去可读性了，这更加加大了维护的难度）

3，范围查询直接扑街，哈希表本来就是无序集合，我要查昨天到今天产生的交易记录简直够呛，除非全表扫描，别无他法。

能从哪些方面优化？

当然，所谓的性能优化，都是指电脑在还有性能空间的情况下的，如果一台电脑CPU性能稀烂，储存用的甚至还是机械硬盘，那么其实做不做优化都无所谓，反正也快不起来。

1，表太大了，或者里面存储的数据太多了。这个是硬伤，关系型数据库逃不掉的命运，基本上数据量到达几百万之后，无论再牛逼的索引和算法，都得慢。那么这个时候我们可以做一个切片的操作，比如说，原来有1000万条数据，那么现在把0-300万条放进去1号，300-600万条放进去2号，剩下的放进去3号。这样子把查询的压力分担开来，确实可以做到性能优化。

2，频繁读写导致频繁加锁，导致等待时间过长。之前我有文章讲过，为了保证数据不会被两个线程同时修改，数据库在读写的时候会给表上锁，比如说共享锁和互斥锁等等。当数据库锁住的时候，所有的请求必须等待，等到锁释放之后才能继续下去。那么对于这个，我们有一个叫做读写分离的手段。

3，索引建立不当。比如说某个索引经常发生范围查询，比如说日期之类的，结果你用哈希表建了个索引，这个不就是撞枪口上了吗？

4，频繁，重复查询的热点数据。就像Redis一样，MySQL自己带有一个非常简易的缓存功能，基本上用起来就是0配置的那种，当然它能实现的功能也没有Redis那么强大。

5，大量外键的引用。没错，外键确实是维护数据库统一性的一个很好的工具，然而，外键的使用弊端也是很大的。

6，无效的查询浪费大量的资源。服务器老是去查一个不存在的值，这会导致很多资源被浪费了。

优化的手段有哪些？

当然，你要说最终极的手段当然就是买一台强大的电脑啦，i9 CPU配上巨大的内存和raid0的固态硬盘，想慢都难。但是，不是所有人都那么朴实无华且枯燥，我们这里只着重说几个常见的普适的办法：MySQL主从复制读写分离，索引，乐观锁机制，减少外键，布隆过滤器。

1，主从复制，读写分离。

一般而言，在数据库插入（或者是修改、删除）数据消耗的时间比读取是小得多的。那么，我们可以做一下手脚，我们准备一个MySQL集群，这个集群里面有3台实例，两台slave，一台主机，两台slave通过一定的机制保持和主机的数据一致。那么，只要是写入（或者是修改、删除）就操作主机，读取就操作两台slave，因为有两台slave，就可以分担一下查询的压力。并且这两台slave都是只做读取任务，理论上不会发生悲观锁，所以效率可以大大提高。

这个方法确实是一个妙招，基本上很多的服务端后台也都是使用的这种办法。但是呢，建立MySQL集群需要money，而且也需要非常稳定的机房，不然……炸了一台，就很容易扑街了。

2，索引

索引相当于是数据库的一个外挂，可以加快一些热点数据（字段）的查询效率。但是肯定有人要问了，MySQL底层不已经是B树了吗，这个索引是建了来干嘛的？我在这里多说几句MySQL索引建立的规则：以常用的InnoDB引擎为例（MYISAM已经淘汰了，这里就不说这个了），InnoDB引擎虽然不要求你建表的时候一定要写主键，但是，它会自动帮你维护一个主键（只不过你看不见而已，背后偷偷有的）这个隐式的主键会在什么时候产生呢，如下：1，没有指定主键。2，主键不是一个自动增长的int类型的字段。3，联合主键。4，BLOB/TEXT类型做了主键（这个已经被禁止了，现在已经不允许这个做主键了）。其实这个也就意味着，只要你指定了一个自动增长的int类型的字段做主键，它就不会生成这个隐式的主键。

所以我们可以知道，MySQL底层的那棵树建立的依据是你的主键（或者是那个隐式的主键），那么除了用主键进行查找之外，所有字段的查找，都会涉及全表扫描。打个比方，我有一个学生表，里面放着一个id（自增长int），姓名（TEXT）和班级（TEXT）

那么，如果我查找高一1班有哪些人的时候，就会发生全表扫描，因为学生id是一个毫无意义的整数，靠这个无法检索班上的学生。所以问题也就来了，我们如果想让按照班级找人找的快，我们就应该按照班级来建立一个索引，对吧？这个时候，MySQL会建立一棵树，这个树依据的就是班级，而不是那个主键了，那么我们再按照班级进行查找的时候，就不会触发全表扫描，效率也自然而然就高了。

欸等等，我是不是前面不小心暴露了点什么东西，索引也是一棵树？这个倒不一定哈，只不过树可以更好的完成范围查询的任务。那么常见的建立索引的数据结构是什么？常见的索引树有两种，一个是B树，一个是哈希表。我们来详细聊聊这两个东西的区别。

B树

B树的结构和原理我这里不重复了，要是忘记了，就去这个网站看看：

```
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
```

不过，事实告诉我们，红黑树的查找效率会比B树更加的高，但是，为什么MySQL的工程师会选择B树这个结构呢？明明有好的用，为啥不弄啊？我们先来铺垫一点知识，磁盘的读取速度是极其慢的，如果是顺序读取倒还有100多MB一秒，还算过得去，但是随机读取简直就不忍直视了，可能只有0.0几MB一秒。然而对于内存来说，内存读写速度可以轻轻松松上去40GB/s，这个显然是存在极大的速度差的。如果说把所有的数据都读进去内存里，然后在分别使用红黑树和B树进行索引，也许红黑树会胜出。但是，如果要考虑上从磁盘加载的时间，情况也许就不一样了。

当然也许有人会杠，你把所有数据都读进来内存里不就好了吗？是的，如果你这么想的话，你的电脑干嘛还要硬盘，买个1TB的内存不好吗？不过就是每次开机都要新装个系统嘛。

对于红黑树，每走到一个节点，就需要读取一次磁盘，然后把读出来的值和查询的值比对，再走向下一个节点。然而在B树中，每走到一个节点，就把这个节点内所有的数据一次性加载到内存里，然后进行横向比较时，就是在内存中调取数据来进行了，速度自然就快起来了。相比起红黑树需要多次从硬盘读取数据进来，B树显著减少了磁盘IO的次数，因此，即便在自身性能有些微劣势的情况下，B树依然可以比红黑树快，而且还快不少。

哈希表

哈希表这个东西前面说Redis的时候说的多了。大家对哈希表的印象就是，非常快，对吧？但是别忘了我之前说过的一个问题，哈希表本身就无序，一旦涉及范围查询，直接扑街。

所以其实现在也很明确了，在建立索引的时候，我们可以有B树和哈希表两个选项，B树基本上是一个万金油级别的索引，如果你的业务关系比较多，或者说范围查询密集的情况下，B树显然是更好的。如果你的业务中大量涉及的是点对点的查询，比如说登陆的时候匹配用户名和密码这种，用哈希表也许就会有奇效。

3，乐观锁

乐观锁其实已经在数据库层面实现了，一般而言我们无需考虑在业务代码中考虑这件事情。以MySQL的InnoDB引擎为例，其实所谓的乐观锁就是读锁。由于InnoDB中一般使用行锁，所以我们只说行锁。行锁只会锁住某一行数据，因而不会导致整个表锁住而引起极大的性能损失。当读锁发生的时候，其他用户是可以继续读取的，不受影响，只有其他用户在写入时，会被阻塞等待。所以，如果在一个业务中，读取发生得比较多得时候，我们可以使用乐观锁的机制，这样就不会导致大量读取请求被阻塞了。

4，外键

关系型数据库最最重要的一个特性就是外键。外键相当于是两张表间关系的一个桥梁，它们起到了维护数据一致性的重要作用。但是在现代的数据库系统中，这个东西已经越来越成为累赘了。很简单的一个问题，数据一致性其实完全可以在业务代码中完成，何必使用外键带来额外的负担呢？而且，大部分情况下，外键的不当使用，往往是灾难性的后果，甚至不仅起不到维护数据一致性的作用，反而在帮倒忙。

我举一个非常简单的例子。老师给我们上课的时候讲过一个外键的维护策略叫做CASCADE，这个意味着，当父表更新的时候，子表会跟着更新，父表删除的时候，子表会跟着删除。这看着很正常很符合我们正常的思维对不对？然而，假设我们有一个教师表，然后还有一个课程表，通过一个教师id的外键添加关系。如果我们使用CASCADE策略的时候，就会发生如下滑稽的事情：

蔡昊兴老师曾经教过我们概率论与数理统计，然而他现在不在华工工作了。我们把他从教师表里删除掉，结果完了，2018级计创班的同学们全都没上过概率论与数理统计课了。所以其实这个也就意味着，CASCADE是生产环境下最最最最最不该用的东西，很容易引起麻烦的。

> 我这里插播一下外键的四个策略，一个叫CASCADE，一个叫SET NULL，一个叫RESTRICT，最后一个是NO ACTION
> CASCADE上面讲过了，就是说父表变化，子表跟着变
> SET NULL是一个择中的策略，打个比方，蔡昊兴老师离开华工了，那么2018级计创的学生们其实还是有上过概率论与数理统计课的，只不过老师是谁，就不知道了。
> RESTRICT 中文意思就是限制，它更加绝，只要发现子表有对父表的引用，父表就不允许更新和删除。打个比方，就是蔡昊兴老师即便离职了，也不可以把他从数据库里面删掉。事实上，这个才是生产环境下用的最多的东西，删除不是真正的删除，而是把它标记为不可见而已。这样就避免了我们上面说的一切问题，当然，缺点就是占空间嘛。
> NO ACTION在MySQL中和RESTRICT基本上类似，意思就是，不允许动作，无动作。

再说到一个问题，就是外键导致的性能低下的问题。既然把两张表要连在一起，那么我父表要修改或者删除的时候，必须要检查一下子表，子表新增东西的时候，还得判断到底新增的东西到底有没有在父表中存在。这个性能差距有多大呢？我有一个大概存了一百万行的数据库，如果我没有使用外键，我运行一次SQL脚本只需要半小时左右，但是如果用上了外键，就要花费4个多小时。况且100万行数据在生产环境下并不算多的，如果外键多了，或者数据库东西再多一点，运行一次花个几天不是什么稀奇事。说到这，同志们你们还敢随便用外键了吗？

5，布隆过滤器

其实在Redis那里我们提到过布隆过滤器，但是我当时没有细讲，只是扔了几个链接。其实布隆过滤器就是一个用来过滤无效查询的工具。如果说明知道一个东西不存在的情况下，我们干脆就不进去数据库找了，这样不就可以省下来很多资源嘛。首先咱们又来铺垫一点东西：布隆过滤器是啥，能有啥用？举个栗子，我有一个数据库，存了20亿条黄色网站，我现在要禁止人们访问这些网站，所以我的浏览器里要设置一个拦截器，如果用户请求了一个20亿条黄网中的某个网站，就不予以跳转。

看到这么大的数据量，肯定很多人会第一时间想到Redis。但是，我们来做一个计算吧。假设那些网址全部都是15个字符吧（其实绝对不止，50个都可以有），那么，假设用UTF-8进行编码之后，每个字符需要3个字节（英文是1个，中文要3个，emoji要4个），然后我们算一算这里有多大哈：90GB。谁家电脑内存这么大？？？我开个浏览器你就要我90GB的内存？搞笑呢？

那既然Redis不行了，我们得想一个更加简单的办法，于是乎人们想到了用bit数组来解决问题。既然20亿条URL我存不下，但是20亿个bit还是可以的哇，这个才250MB，是不是很赞嘛？所以这个时候我们又得请出我们的哈希表了。我们先建立一个哈希表，里面存的全都是0，那么，对20亿条url分别计算哈希，把对应的位置标成1。那么现在就相当于我们有一个哈希表了。当用户进行请求的时候，我们只需要计算一下url的哈希值，如果对应的bit是0，就表明这个网站可以访问，如果是1，代表这个网站！！有可能！！不能访问。

欸等等，为啥变成可能了？1不是代表我这里标定过嘛？因为哈希碰撞会产生这样的误会，假设我的哈希函数是n%10，那么我11和21的计算结果是一样的。所以，也正因为哈希碰撞，所以布隆过滤器是有几率犯错误的，但是，有错误不代表就不好，如果这个错误率非常低，那么其实带来的影响也就微乎其微了。就像我们前面提到的Redis的缓存穿透，虽然使用布隆过滤器不可以完全避免穿透，但是，我们能拦截大部分的缓存穿透，就已经是一个胜利了。

但是，布隆过滤器还有一个非常棒的性质，它的反向判断是一定正确的。如果我告诉你这个网站不在20条之中，那它就一定不存在！严谨的证明我还没有本事做，不妨这么理解，如果数字A模10和数字B模10相等，我并不可以判断AB是不是相等的，它们可能相等，也可能不相等。但是如果数字A模10和数字B模10不相等，AB一定不相等！

但是，布隆过滤器依然还是存在一个缺陷，就是删除数据的问题。如果我从20亿条里面移除一个URL，我可以把过滤器里面哈希值相应的位置从1改成0吗？那当然是不行的，其实还是因为哈希碰撞。

现在已经说完大部分的MySQL优化了，日后如果有空的话，我们可以谈谈ORM框架的运用以及跟数据库解耦的问题。目前数据库系列暂时停更啦。