参考：

MySQL实战45讲

前言

在学Java时就会涉及到一个头等大的问题——并发，那个时候我们听到最多的一个字就是锁。在MySQL或者说大部分数据库中，肯定也是会遇到并发的问题的，所以一款优秀的数据库软件必然会有锁的支持，该问就来谈谈MySQL所涉及到的锁。

全局锁

见名知意，这个锁是全局的，整个数据库都会被锁住，MySQL就提供了一个加全局读锁的方法，命令是

Flush tables with read lock

简称就是FTWRL操作，使用后，数据更新语句，数据定义语句以及更新类事务的提交语句都将被阻塞。

相当于全库被锁住只有读的功能了，有啥用呢，有个典型场景就是全库的逻辑备份（这个备份过程中显然不应该有额外的更新操作，容易造成数据的紊乱）。

怎么个紊乱法呢，举个例子，某电商app的用户余额表和用户已购商品表。

全库在备份了，小明却在这时购买商品了，我们模拟一个过程。

1.全局备份，余额表备份完毕；

2.商品表，余额表因小明的购买相应调整；

3.全局备份，已购商品表备份完毕；

4.用备份文件将数据库还原至备份的状态。

这个流程下来有啥后果呢，小明的已购商品表多了商品，余额却没有减少，所以做全局备份全局锁是有必要的。

但仔细一想FTWRL操作也有点可怕，往往全局逻辑备份肯定不是秒级就能完成了，是需要一段时间的。

1.主库备份的话，业务将基本处于停滞状态；

2从库备份的话，期间b不能执行主库同步过来的binlog，会导致主从延迟。

那么，其实FTWRL操作还真不适合像电商这种24小时无间断都可能高频操作的系统。就没有其它办法了吗？当然有，我们之前整理过的事务隔离就提过可重复读这个隔离级别，这样我在这个事务中备份，从头至尾都是一致性的视图，并且由于MVCC的支持，我们是可以在这个过程中做其它数据更新的。官方提供了mysqldump逻辑备份工具，使用参数-single-transaction时，导数据前就会启动一个事务。

有人要说，早说嘛，扯那么多干嘛。细心的同学可能还想起我们提到的是事务来解决，那么可不是所有的数据库引擎都支持事务的，比如MyISAM。当然不出意外现在的MySQL版本大家都用的InnoDB引擎，所以针对FTWRL这个操作大家也可以权当了解下。

那可能有人了解的挺多的，还知道有个set global readonly=true，这样也会让全库进入只读状态，但在不支持事务可重复读的情况下，还是依旧建议使用FTWRL，为啥呢：

1.有些系统中，readonly被用来做其它逻辑处理，比如判断一个库是主库还是备库，因此修改global的方式影响面更大；

2.设值readonly后，若客户端发生异常，数据库还是会一直保持readonly状态，可能会长时间导致数据库不可写，而FTWRL本质还是个命令，客户端异常断连，MySQL会自动释放全局锁。

总得来说，全局非特殊情况不建议使用，有人要说不是有可重复读的事务吗，这个嘛，上文提及这个是来替代全局锁的不灵活性，它又不是全局锁，全局锁本身FTWRL还是不建议使用哈（经常有这种屌丝系统，夜间11点到隔天5点之类的系统维护，不就是要维护数据嘛，却垃圾的让用户停止使用，你说气人不，也就是这种系统一般是什么公交充值之类，你体验不好能咋样，还是得坐公交）。

表级锁

这就是要小一级别的锁，一种是表锁，还有一种是元数据锁。

先将表锁，语法一般这样lock tables ... read/write。与FTWRL类似，可以用unlock tables主动释放锁，也可以在客户端断开的时候自动释放。

不过表锁的影响很大，不仅限制别的线程的读写，也限制了本线程的操作。举个例子，线程A执行lock tables t1 read,t2 write，这样其它线程写t1，读写t2都会阻塞，同时线程A也只能读t1，读写t2，写t1以及访问其它表都不行。

虽然影响面大，在没有更小粒度的锁之前，这已经是处理并发的最有效手段了。

不过还有种表级的锁MDL（元数据锁）还是用的挺多的，是在MySQL5.5之后引入的。

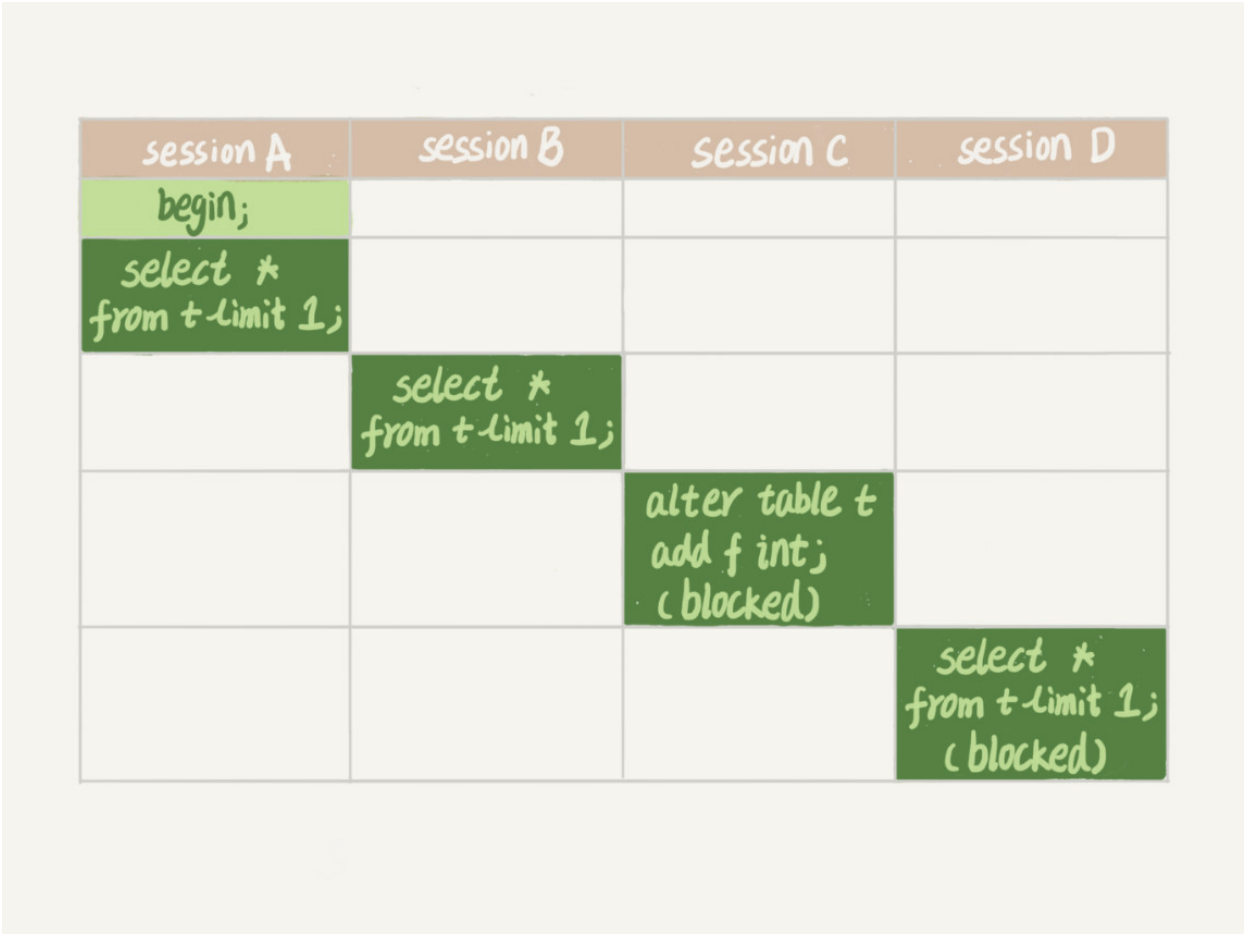
MDL不需要显示调用，访问一张表的时候会自动加上，它是用来保证读写的正确性的。举个例子，正在查询一张表中的数据，执行期间另一个线程对这张表做了DDL操作删了一列，这样结果就不是一行的问题了，是所有数据都有问题了。

所有MySQL5.5之后引入了MDL锁的概念，当做增删改查时加MDL读锁，当做表结构变更时加MDL写锁。

1.读锁不互斥，所以多个线程可以同时做增删改查；

2.写锁之间包括读写锁之间都是互斥的，保证表结构变更的安全性，有两个线程要对一个表加字段，一个必须等另一个做完。

正是因为这个特性，也是有陷阱的，举个例子。



如图，A给t加了MDL读锁，B紧接着也来加了MDL读锁，读锁不互斥，所以不影响，这时候C又来加了MDL写锁，如上文所说，读写锁互斥，由于AB的读锁都没有释放，C只能等着，这时候D又来了加MDL读锁，就出事了，由于C加了MDL写锁，D也只能等着。

这样，只要这个事务不提交，这张表等于完全不能读写了。针对这个问题，有两个考虑，一评估事务长度以及重要程度，就是能不能kill掉，二就是我们的DDL操作暂时就不做。

真实场合，第一种思路很难实现，热点操作，可能你刚kill掉事务，它又重新建立了，所以还是可以考虑第二种思路。

行锁

介绍完上两种锁，无疑还是觉得范围太广，我们操作数据库要能精确到行那才是最理想，所以后续很多引擎自己实现了行锁这个概念，比如我们先MySQL的主流引擎InnoDB。说到底，锁的发展就是在保证并发数据安全性的前提下又能提高业务处理效率，我们就围绕这个概念来继续了解下InnoDB中的行锁吧。

先引入一个例子。

执行B的update语句会有什么现象呢，那么我们先看事务A，它先执行完两条update语句，但还未提交，所以这时A是持有id为1和2的行锁的，而B的updae就是针对id为1这一行的，InnoDB中行锁是需要时才加

上的，但不会立即释放得等到事务提交（这个就是两阶段锁协议），所以B会被阻塞，直到A提交事务后。

明白了InnoDB中行锁的这一特性，我们可能很容易想到一个问题。举个例子。

如图，我把事务A和B的操作调整了下，这样会咋样呢，A先持有id=1的行锁，然后B又持有了id=2的行锁，这时紧接着，A想要id=2的行锁，B想要id=1的行锁，但都被对方占用，并且事务都没结束，这样就陷入无限互相等待的死胡同了，我们称之为死锁。

针对死锁，我们有两种策略可以考虑：

1.就让它们等待，但是我们设置等待时间（参数innodb_lock_wait_timeout，默认是50s），这样就不至于无限时间了；

2.发起死锁检测，发现死锁，主动回滚死锁链条中的某个事务，释放占用锁，这样牺牲个别就能让其它事务跑下去了。参数innodb_deadlock_detect设置为on。

当然一般情况我们肯定是选择第二种策略去优化的，锁等待的情况比较复杂，又不是只有死锁，你把等待时间调小那不就是宁可错杀一千也不放过一个吗，那不调小呢，等待时间过长，那业务还做不做了。

那么我们浅显的用第二种策略是否又合理呢，死锁检测那显然不是做善事，那是要开销性能的，假设1000个并发线程同时更新一行，那么死锁检测就能达到100万这个量级的（每个线程都要循环去判断和另外999个有没有发生死锁，粗略的就是1000*1000）。要检测出来了那也算不枉费消耗，有时候偏偏还没有死锁，大费周章就执行了几个事务还没检测到死锁。

针对这个，我们要讨论的就是怎么解决热点行更新导致的性能问题。

1.关闭死锁检测，不是说我们就用上面的第一种策略了，二手我们要确保这个业务一定不会出现死锁，这个风险相当的大，有种死马当活马医的感觉；

2.控制并发度，我们上面假设的是并发1000个线程，比方我们控制同一行同时最多10个线程更新，这个性能的开销就相当低了。那么怎么控制呢，建议两处，一数据库服务端，二中间件（不建议客户端，集群一做，量瞬间又大上去了）。