MySQL锁



**开销、加锁速度、死锁、粒度、并发性能**

--表锁：开销小，加锁快，不会出现死锁，锁定粒度大，发生锁冲突概率高，并发度低

--行锁：开销大，加锁慢，会出现死锁，锁定粒度小，发生锁冲突概率低，并发度高

--页锁：介于行锁和表锁之间，会出现死锁，并发度一般

表锁更适用于**以查询为主**，只有**少量按索引条件更新数据**的应用；行锁更适用于有**大量按索引条件并发更新少量不同数据**，同时**又有并发查询**的应用

**MyISAM表锁**

MyISAM存储引擎只支持表锁，随着应用对事务完整性和并发性要求的不断提高，MySQL才开始开发基于事务的存储引擎，后来慢慢出现了支持页锁的BDB存储引擎和支持行锁的InnoDB存储引擎。但是MyISAM的表锁依然是使用最为广泛的锁类型。

查询表级锁争用情况，可以通过检查table\_locks\_waited和table\_locks\_immediate状态变量来分析系统上的表锁定争夺。

如果table\_locks\_waited的值比较高，则说明存在较严重的表级锁竞争；table\_locks\_immediate表明立即获得锁

**MySQL表级锁的锁模式**

MySQL的表级锁有两种模式：**表共享**读锁（Table Read Lock）和**表独占**写锁（Table Write Lock）。



可以多个线程读，只能有一个线程写，一旦开始写，其他线程不可读写

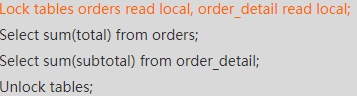
如何加表锁？

MyISAM在执行查询语句前，会自动给相关的表加读锁，在执行更新操作的前，会自动给相关的表加写锁；自动加锁时，MyISAM总是一次获得sql语句相关表的全部锁，这也是MyISAM表不会出现死锁的原因。

有一个订单表orders，其中记录有各订单的总金额total，同时还有一个订单明细表order\_detail，其中记录有各订单每一产品的金额小计 subtotal，假设我们需要检查这两个表的金额合计是否相符，可能就需要执行如下两条SQL：



此时需要给查询语句加读锁，避免在查询的时候其他线程对其数据进行了修改



Local后面章节会介绍，显式加表锁时，需要同时取得所有相关的表的锁，对于其他未加锁的表，不能访问。

当使用lock tables时，不仅需要一次锁定用到的所有表，而且同一个表在sql中出现多少次，不管是用别名还是原来名字，都得在lock tables语句后面一一写出来，否则就会出错

例如：mysql> lock table actor read;只对actor进行了加锁

然后mysql> select a.first\_name,a.last\_name,b.first\_name,b.last\_name from **actor a,actor b** where就会出错，说table a没有被lock tables锁定；因此需要对别名进行lock tables，即下面的lock table actor as a read,actor as b read;，这样就能解决。

**并发插入**

Concurrent\_insert用于控制并发插入的行为：

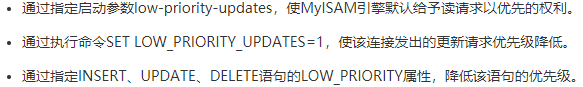
当Concurrent\_insert = 0，不允许并发插入，当Concurrent\_insert = 1，如果MyISAM表中没有空洞（表中间的数据没有是需要删除的），MyISAM允许在一个线程读表时，另一个线程从表尾插入数据。当Concurrent\_insert = 2时，无论是否有空洞，都允许在表尾并发插入记录。

如果session1获得了read local锁，该线程可以对表film-test进行查询操作，但不能对表进行更新插入操作，session2不能对film-test进行删除更新操作，但可以对表进行并发插入操作。当session2执行插入和更新的操作的时候，插入操作执行结束，更新操作就进入等待，但是即使session2插入结束了，session1依然无法访问session2插入的数据，直到释放锁，session1可以访问插入的数据，并且session2进行更新操作。

**MyISAM的锁调度**

Mysql认为写请求一般比读请求重要，即使读请求先到等待队列，写请求后到，也是写请求优先执行。因此，**MyISAM表不适合于有大量更新操作和查询操作的应用，因为大量更新操作会造成查询操作长时间阻塞。**

通过以下操作调节调度行为：



**InnoDB锁问题**

InnoDB与MyISAM的最大不同有两点：一是**支持事务**（TRANSACTION）；二是采用了**行级锁**。

**InnoDB的行锁模式及加锁方法**

共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排它锁。

排他锁（X）：允许获得排它锁的事务更新数据，阻止其他事务取得相同数据的共享读锁和排他写锁。

内部使用的意向锁，都是表锁：

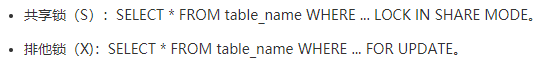
意向共享锁（IS）：给数据行加行共享锁，加锁前必须取得该表的IS锁

意向排他锁（IX）：给数据行加排它锁，家所欠必须先取得该表的IX锁



意向锁是自动添加的，不需要用户干预。对于增删改，innodb会自动给涉及的数据集加排它锁，对于普通的select语句，innodb不会加任何锁。

显式加锁：



In share mode获得共享锁，主要用于确认某行数据是否存在，并确保没有人对这个记录进行更新或者删除操作，但是如果当前事务也需要对该记录进行更新操作，就有可能造成死锁；因此对于锁定行记录之后还需要进行更新操作，应该是用for update获得排它锁

**共享锁操作**：session1对记录行a进行加共享锁，session2也对记录行a进行加共享锁；然后session1对记录行进行更新操作，即加排它锁，此时如果没有其他线程有共享锁，就可以进行更新操作，但是session2对a加了共享锁，因此，session1的更新操作等待session2对共享锁的释放，接着session2也对a记录进行更新操作，就会出现session2等待session1释放排他锁，session1等待session2释放排它锁，产生死锁，session2退出，session1获得锁，然后进行更新。

**排它锁**：session1对记录b添加排它锁，其他session能够查询该记录，但是不能对该记录加共享锁和排他锁，session2查询记录，并添加排它锁，等待session1释放锁，session1进行更新操作，然后commit操作释放锁，session2获得锁之后，查询到新记录，并且对新记录加了排它锁。

**InnoDB行锁实现方式**

InnoDB行锁实现是通过索引上的索引项加锁实现的，意味着：只有通过索引条件检索数据，InnoDB才会使用行锁，否则使用表锁。

例子：

情况1：未使用索引列，导致不同记录阻塞

session1查询id=1的行，session2查询id=2的行，此时id不是索引列，当session1对id=1的记录行添加排它锁的，此时还没有commit，也就是排它锁还在，session2对id=2的记录行也添加排它锁，此时session2等待，因为此时id不是索引列，行锁是加在索引上的，因此InnoDB只能使用表锁，把整张表锁住。因此想使用行锁的时候，需要使用索引列作为条件。

情况2：使用相同索引键值导致的阻塞例子—id为主键索引

Session1对id=1，name=1的记录行进行加排它锁，session2对id=1，name=4的记录行进行加排它锁，此时虽然他们不是同一行记录，但是他们同一个主键索引，因此会阻塞。

情况3：即使你使用了索引列，但是优化器认为全盘扫描效率更高，或者查询过程中有需要进行全盘扫描的列，都会将行锁升级为表锁。

**间隙锁（gap-key）**

使用范围条件查询的时候，对于在条件范围内，但不存在的数据，叫做间隙，InnoDB也会对这个间隙进行加锁。

搜索id>100的数据，但是数据只有101,102；此时会对102以后的不存在的数据进行加锁。间隙锁是为了防止幻读，如果查找id>100的记录，第一次查找之后，其中有线程进行了同范围内的插入数据，那么在此查找的时候就会引起幻读。

**间隙锁阻塞例子**

Session1对不存在记录行id=102进行加间隙锁，session2对其进行插入id=102的数据，此时会出现间隙锁阻塞，session1由于没有当前记录，执行rollback，释放间隙锁，此时session2获得锁，进行插入数据。

**什么时候使用表锁**

①事务需要更新大部分或者全部数据，使用行锁会让执行效率低下，并且冲突和等待增多。

②事务涉及多个表，容易引起死锁，造成事务回滚