全局锁、表级锁以及行锁（上）

MySQL 为什么需要锁?

- 数据库的设置主要解决两个问题：高效的存储和查询数据，以及处理用户的并发请求
- 当出现并发访问时，数据库需要对访问的资源进行合理的控制，以及制定不同的访问规则，锁就是做这件事情的
- 另外需要注意的是，MySQL 中的锁和操作系统的互斥量、信号量是有区别的，比如 MySQL 中的锁带有死锁检测，并且有读锁和写锁

全局锁

- 顾名思义，全局锁就是对整个数据库实例加锁。MySQL 提供了添加全局读锁的方法，即 FTWRL，如右侧所示

```
flush tables with read lock;
// 业务逻辑
unlock tables;
```

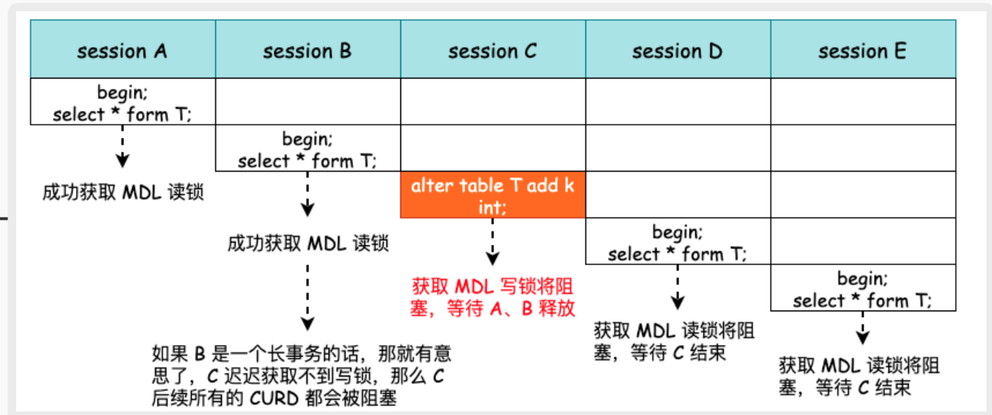
- 当我们为整个数据库实例添加了全局读锁以后，此时数据库将处于只读的状态，对数据库的任何修改，例如修改、新增数据，创建或更新表结构，都会失败。全局锁的一个最主要的应用就是做全库的逻辑备份
- 当我们在主库上使用 FTWRL 做全库备份时，主库无法进行更新，业务可能就直接跪在那里了。当我们对从库使用同样的方式进行整库备份时，那么从库的 SQL Thread 无法将 binlog 写入至从库，将会导致主从产生延时
- 既然全局锁这么危险，为什么还需要有全局锁?
  - 全局锁主要是提供给不支持事务的存储引擎进行逻辑一致的数据备份的，例如 MyISAM
  - MyISAM 并不支持事务，也就没有 MVCC 之说，所以没有办法提供非锁定的一致性读，只能通过添加全局锁的方式来进行一致性的整库备份
- 对于 InnoDB 而言，在使用 mysqldump 的时候可以添加 –single-transaction 参数，使其主动开启一个事务，然后就可以进行非锁定的一致性读了

表级锁

- 表级锁分为两种，一种是表锁，另一种则是我们经常碰到的元数据锁（Meta Data Lock, MDL）。MDL 在我们对表进行结构变更时添加
- 表锁：表锁的语法为 lock tables T read/write，比较有意思的地方 lock tables 语法除了会限制别的线程的读写外，也限定了本线程接下来的操作对象。也就是说，假设 thread A 在 t1 添加 read lock，那么 A 只能读取数据；若添加了 write lock，那么是可以进行读写操作的

元数据锁

- MDL 不需要显示添加，当我们对数据进行 CURD 操作时，就会首先去获取 MDL 读锁，保证此时没有其它线程在对表结构进行更改
- 当我们对表结构进行更改，例如执行 alter table T ... 的时候，就会尝试为 T 添加 MDL 写锁，表示开始对 T 进行元数据的更改
- 如果我们在表 T 繁忙时进行 DDL 的话，可能可能会导致整个表无法处理任何请求，尽管表 T 的数据量可能非常少



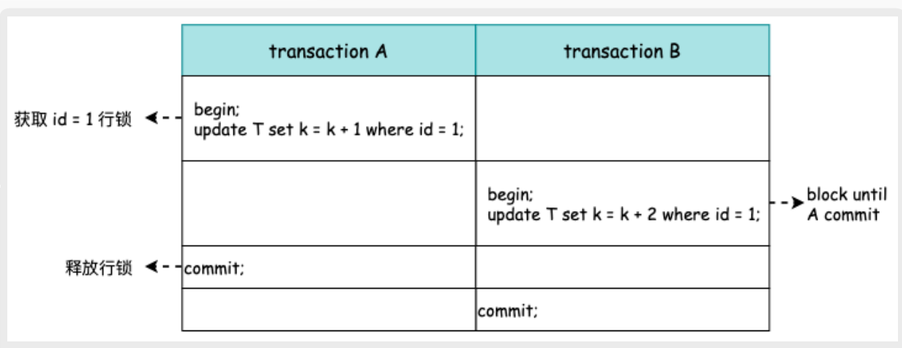
- 如上图所示，当我们开始执行 alter table T 的时候，由于前面有事务进行，所以 session C 获取 MDL 写锁将阻塞。但是由于 B 是长事务，session C 可能阻塞非常长的时间，同时，如果后续仍有查询、修改、删除等请求时，将获取不到 MDL 读锁，同样会阻塞。这样一来，这个库的线程数可能很快就爆满了
- 在 MySQL 5.7 中，online DDL 已经比较完善了，只有在初始准备阶段以及收尾阶段才需要获取 MDL 写锁

获取 MDL 写锁	阻塞其它 CURD
降级 MDL 为读锁	降级之前会做一些 prepare 工作
进行实际的 DDL	此时 table 的 CURD 不会被阻塞
升级 MDL 为写锁	阻塞其它 CURD
释放 MDL 写锁	DDL 结束

- 因此，当我们执行 DDL 的时候，有两个要素必须要考虑：当前 table 是否有长事务的执行，以及当前 table 的请求是否繁忙
- Question
  - 如果我们在从库上使用 mysqldump –single-transaction 进行备份时，主库的某一个小表执行了 alter table T ... 动作，从库会发生什么?
  - 前面我们提到过，mysqldump –single-transaction 会主动开启一个事务，使备份走一致性非锁定读，那么既然是读取数据，必然需要获取到该表的 MDL 读锁。当备份进行时，SQL Thread 执行主库发来的 alter table 时，获取 MDL 写锁将阻塞，主从将出现延迟

行锁

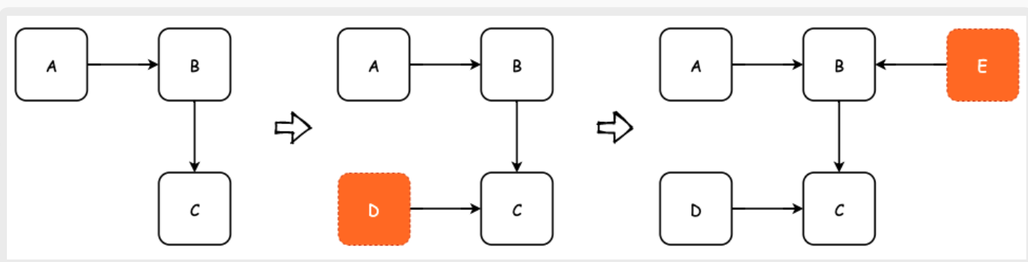
- 在 MySQL 中，行锁由各个存储引擎自行实现，并不由 MySQL Server 实现。因此，MyISAM 就不支持行锁，而 InnoDB 则支持行锁
- 行锁，通常而言是在 UPDATE、INSERT 语句执行时开始获取，而不是事务开始时获取，并且在事务执行完毕后释放，而不是语句执行完毕后释放，这一点非常关键



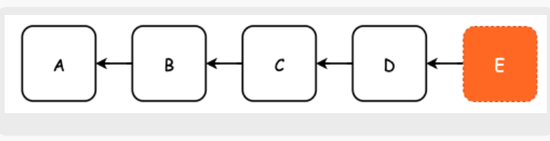
- 也就是说，当事务 B 开始执行时，若事务 A 未执行完毕，将会一直阻塞
- 因此，当我们的业务需要在一个事务中更新多条数据时，要把最可能产生冲突的更新语句放到事务的后面儿，以减少锁的持有实现，从而提高事务的并发度

死锁与死锁检测

- 行锁由于非常贴近业务，所以也是死锁的高发地带。死锁简单的来说就是循环依赖，依赖和依赖之间所组成的有向图存在环
- 死锁处理方式
  1. 什么都不做，直接等待。可以通过 innodb_lock_wait_timeout 这个参数来设置锁的等待时间，默认为 50s
     - 锁等待时间不能设置的太小，因为有时候长事务执行时后续更新也需要等待，而这时候并没有死锁发生
     - 直接等待的方式对于业务端而言，50s 是很难接受的
  2. 开启死锁检测，通过 innodb_deadlock_detect = on 进行设置，那么当 InnoDB 发现有死锁发生时，直接回滚其中一个事务
     - 无需等待，但是开启死锁检测是有代价的
- 这里我们着重的对死锁检测的时间复杂度进行一个基本的分析



- 如上图所示，假设每一个该 DAG 中每一个节点都表示一个事务，方向则表示依赖，也就是等待该事务 commit。当我们每新增一个事务时，都需要去该图中判断是否因为自身的加入而形成的环，有向图环检测的时间复杂度为 O(V+E)，即必须遍历所有节点才能得知是否有环。因此，死锁检测的整体时间复杂度为 O(E*V*V)
- 当然，并不是所有新加入的事务都需要进行死锁检测，而是只在当前事务被阻塞时才会进行 deadlock detect，并且也并不需要扫描全部的事务，只需要在一个链路上进行扫描即可
- 但是存在一个极端情况，就是所有的事务都去更新同一行数据，那么此时 DAG 就可能变成了一个链表



- 此时环检测的平均时间复杂度就变成了 O(n^2)，n 为并发执行的事务数量
- 也就是说，假如有 100 个事务同时更新同一行数据，死锁检测需要遍历 1 + 2 + 3 + 4 + ... + 100 = 5050 次才能结束

减少死锁检测带来的代价

1. 降低并发
   - 最实用的方式还是降低修改同一条语句的事务的并发执行度，也就是原本有 100 个事务并发执行，通过业务层的处理，将其降低成 5 个或者 10 个事务并发执行
   - 此时只能通过数据库中间件来完成，也就是对更新同一行数据的事务进行排队处理
2. 拆分逻辑
   - 对热点数据进行拆分，比如将一行拆成多行数据，业务在更新数据时随机地选择一条数据进行更新，读取数据时再进行汇总
   - 这种方式其实也是减少对同一条数据的并发更新数量

- 最后，乐观锁也可以避免死锁的发生，只不过乐观锁适用于读多写少、并且事务执行比较简短的情况。同时，也需要对原有的业务代码进行修改

关于意向锁、排它锁以及共享锁的内容放在下篇，欲知后事如何，且听下回分解