MySQL 加锁处理分析

网易杭研　　何登成

1 背景

MySQL/InnoDB 的加锁分析，一直是一个比较困难的话题。我在工作过程中，经常会有同事咨询这方面的问题。同时，微博上也经常会收到 MySQL 锁相关的私信，让我帮助解决一些死锁的问题。本文，准备就 MySQL/InnoDB 的加锁问题，展开较为深入的分析与讨论，主要是介绍一种思路，运用此思路，拿到任何一条 SQL 语句，就能完整的分析出这条语句会加什么锁？会有什么样的使用风险？甚至是分析线上的一个死锁场景，了解死锁产生的原因。

注：MySQL 是一个支持插件式存储引擎的数据库系统。本文下面的所有介绍，都是基于 InnoDB 存储引擎，其他引擎的表现，会有较大的区别。

1.1 MVCC：Snapshot Read vs Current Read

MySQL InnoDB 存储引擎，实现的是基于多版本的并发控制协议——MVCC (Multi-Version Concurrency Control) (注：与 MVCC 相对的，是基于锁的并发控制，Lock-Based Concurrency Control)。MVCC 最大的好处，相信也是耳熟能详：读不加锁，读写不冲突。在读多些少的 OLTP 应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的 RDBMS，都支持了 MVCC。

在 MVCC 并发控制中，读操作可以分成两类：快照读 (snapshot read)与当前读 (current read)。快照读，读取的是记录的可见版本 (有可能是历史版本)，不用加锁。当前读，读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

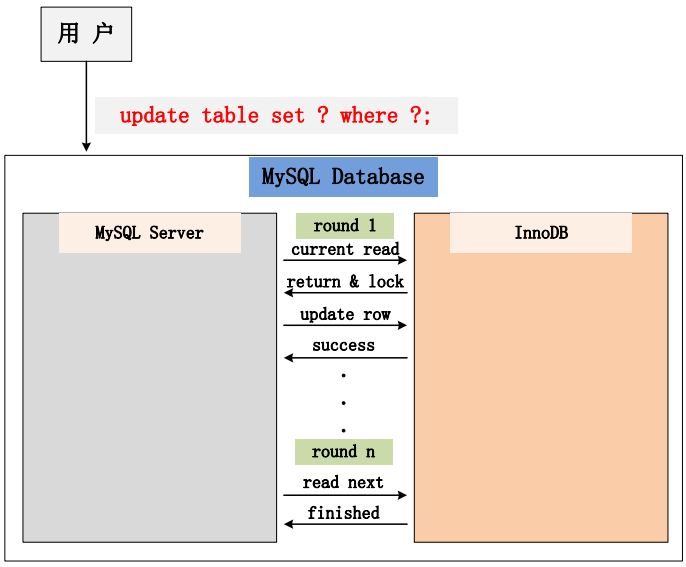
在一个支持 MVCC 并发控制的系统中，哪些读操作是快照读？哪些操作又是当前读呢？以 MySQL InnoDB 为例：

- 快照读：简单的 select 操作，属于快照读，不加锁。(当然，也有例外，下面会分析)
  - select * from table where ?;

- 当前读：特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。
  - select * from table where ? lock in share mode;
  - select * from table where ? for update;
  - insert into table values (…);
  - update table set ? where ?;
  - delete from table where ?;

  所有以上的语句，都属于当前读，读取记录的最新版本。并且，读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。其中，除了第一条语句，对读取记录加 S 锁 (共享锁)外，其他的操作，都加的是 X 锁 (排它锁)。

为什么将 插入/更新/删除 操作，都归为当前读？可以看看下面这个 更新 操作，在数据库中的执行流程：



从图中，可以看到，一个 Update 操作的具体流程。当 Update SQL 被发给 MySQL 后，MySQL Server 会根据 where 条件，读取第一条满足条件的记录，然后 InnoDB 引擎会将第一条记录返回，并加锁 (current read)。待 MySQL Server 收到这条加锁的记录之后，会再发起一个 Update 请求，更新这条记录。一条记录操作完成，再读取下一条记录，直至没有满足条件的记录为止。因此，Update 操作内部，就包含了一个当前读。同理，Delete 操作也一样。Insert 操作会稍微有些不同，简单来说，就是 Insert 操作可能会触发 Unique Key 的冲突检查，也会进行一个当前读。

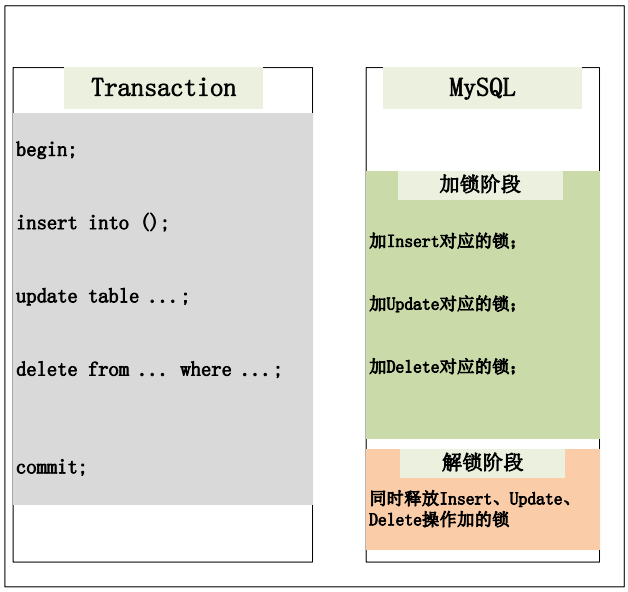
注：根据上图的交互，针对一条当前读的 SQL 语句，InnoDB 与 MySQL Server 的交互，是一条一条进行的，因此，加锁也是一条一条进行的。先对一条满足条件的记录加锁，返回给 MySQL Server，做一些 DML 操作；然后在读取下一条加锁，直至读取完毕。

1.2 Cluster Index：聚簇索引

InnoDB 存储引擎的数据组织方式，是聚簇索引表：完整的记录，存储在主键索引中，通过主键索引，就可以获取记录所有的列。关于聚簇索引表的组织方式，可以参考 MySQL 的官方文档：Clustered and Secondary Indexes 。本文假设读者对这个，已经有了一定的认识，就不再做具体的介绍。接下来的部分，主键索引/聚簇索引 两个名称，会有一些混用，望读者知晓。

1.3 2PL：Two-Phase Locking

传统 RDBMS 加锁的一个原则，就是 2PL (二阶段锁)：Two-Phase Locking。相对而言，2PL 比较容易理解，说的是锁操作分为两个阶段：加锁阶段与解锁阶段，并且保证加锁阶段与解锁阶段不相交。下面，仍旧以 MySQL 为例，来简单看看 2PL 在 MySQL 中的实现。



从上图可以看出，2PL 就是将加锁/解锁分为两个完全不相交的阶段。加锁阶段：只加锁，不放锁。解锁阶段：只放锁，不加锁。

1.4 Isolation Level

隔离级别：Isolation Level，也是 RDBMS 的一个关键特性。相信对数据库有所了解的朋友，对于 4 种隔离级别：Read Uncommited，Read Committed，Repeatable Read，Serializable，都有了深入的认识。本文不打算讨论数据库理论中，是如何定义这 4 种隔离级别的含义的，而是跟大家介绍一下 MySQL/InnoDB 是如何定义这 4 种隔离级别的。

MySQL/InnoDB 定义的 4 种隔离级别：

- **Read Uncommited**
  可以读取未提交记录。此隔离级别，不会使用，忽略。
- **Read Committed (RC)**
  快照读忽略，本文不考虑。
  针对当前读，**RC 隔离级别保证对读取到的记录加锁 (记录锁)**，存在幻读现象。
- **Repeatable Read (RR)**
  快照读忽略，本文不考虑。
  针对当前读，**RR 隔离级别保证对读取到的记录加锁 (记录锁)，同时保证对读取的范围加锁，新的满足查询条件的记录不能够插入 (间隙锁)**，不存在幻读现象。
- **Serializable**
  从 MVCC 并发控制退化为基于锁的并发控制。部分快照读与当前读，所有的读操作均为当前读，读加读锁 (S 锁)，写加写锁 (X 锁)。
  Serializable 隔离级别下，读写冲突，因此并发度急剧下降，在 MySQL/InnoDB 下不建议使用。

2 一条简单 SQL 的加锁实现分析

在介绍完一些背景知识之后，本文接下来将选择几个有代表性的例子，来详细分析 MySQL 的加锁处理。当然，还是从最简单的例子说起。经常有朋友发给我一个 SQL，然后问我，这个 SQL 加什么锁？就如同下面两条简单的 SQL，他们加什么锁？

- **SQL1：**select * from t1 where id = 10;
- **SQL2：**delete from t1 where id = 10;

针对这个问题，该怎么回答？我能想象到的一个答案是：

- **SQL1：**不加锁。因为 MySQL 是使用多版本并发控制的，读不加锁。
- **SQL2：**对 id = 10 的记录加写锁 (走主键索引)。

这个答案对吗？说不上来。即可能是正确的，也有可能是错误的，已知条件不足，这个问题没有答案。如果让我来回答这个问题，我必须还要知道以下的一些前提，前提不同，我能给出的答案也就不同。要回答这个问题，还缺少哪些前提条件？

- **前提一：**id 列是不是主键？
- **前提二：**当前系统的隔离级别是什么？
- **前提三：**id 列如果不是主键，那么 id 列上有索引吗？
- **前提四：**id 列上如果有二级索引，那么这个索引是唯一索引吗？
- **前提五：**两个 SQL 的执行计划是什么？索引扫描？全表扫描？

没有这些前提，直接就给定一条 SQL，然后问这个 SQL 会加什么锁，都是很业余的表现。而当这些问题有了明确的答案之后，给定的 SQL 会加什么锁，也就一目了然。下面，我将这些问题的答案进行组合，然后按照从易到难的顺序，逐个分析每种组合下，对应的 SQL 会加哪些锁？

注：下面的这些组合，我做了一个前提假设，也就是有索引时，执行计划一定会选择使用索引进行过滤 (索引扫描)。但实际情况会复杂很多，真正的执行计划，还是需要根据 MySQL 输出的为准。

- **组合一：**id 列是主键，RC 隔离级别
- **组合二：**id 列是二级唯一索引，RC 隔离级别
- **组合三：**id 列是二级非唯一索引，RC 隔离级别
- **组合四：**id 列上没有索引，RC 隔离级别
- **组合五：**id 列是主键，RR 隔离级别

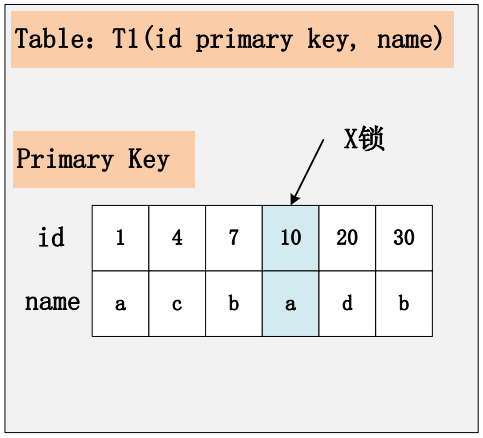
- **组合六**：id 列是二级唯一索引，RR 隔离级别
- **组合七**：id 列是二级非唯一索引，RR 隔离级别
- **组合八**：id 列上没有索引，RR 隔离级别
- **组合九**：Serializable 隔离级别

排列组合还没有列举完全，但是看起来，已经很多了。真的有必要这么复杂吗？事实上，要分析加锁，就是需要这么复杂。但是从另一个角度来说，只要你选定了一种组合，SQL 需要加哪些锁，其实也就确定了。接下来，就让我们来逐个分析这 9 种组合下的 SQL 加锁策略。

注：在前面八种组合下，也就是 RC，RR 隔离级别下，SQL1：select 操作均不加锁，采用的是快照读，因此在下面的讨论中就忽略了，主要讨论 SQL2：delete 操作的加锁。

2.1 组合一：id 主键+RC

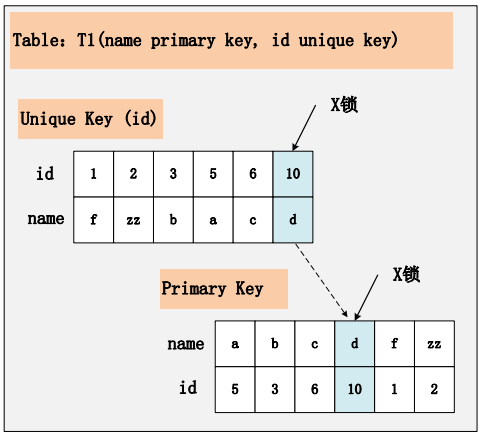
这个组合，是最简单，最容易分析的组合。id 是主键，Read Committed 隔离级别，给定 SQL：delete from t1 where id = 10; 只需要将主键上，id = 10 的记录加上 X 锁即可。如下图所示：



结论：id 是主键时，此 SQL 只需要在 id=10 这条记录上加 X 锁即可。

2.2 组合二：id 唯一索引+RC

这个组合，id 不是主键，而是一个 Unique 的二级索引键值。那么在 RC 隔离级别下，delete from t1 where id = 10; 需要加什么锁呢？见下图：

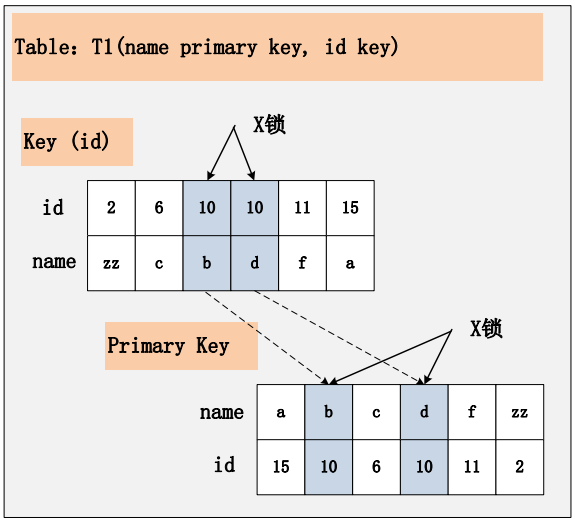


此组合中，id 是 unique 索引，而主键是 name 列。此时，加锁的情况由于组合一有所不同。由于 id 是 unique 索引，因此 delete 语句会选择走 id 列的索引进行 where 条件的过滤，在找到 id=10 的记录后，首先会将 unique 索引上的 id=10 索引记录加上 X 锁，同时，会根据读取到的 name 列，回主键索引(聚簇索引)，然后将聚簇索引上的 name = 'd' 对应的主键索引项加 X 锁。为什么聚簇索引上的记录也要加锁？试想一下，如果并发的一个 SQL，是通过主键索引来更新：update t1 set id = 100 where name = 'd'; 此时，如果 delete 语句没有将主键索引上的记录加锁，那么并发的 update 就会感知不到 delete 语句的存在，违背了同一记录上的更新/删除需要串行执行的约束。

结论：若 id 列是 unique 列，其上有 unique 索引。那么 SQL 需要加两个 X 锁，一个对应于 id unique 索引上的 id = 10 的记录，另一把锁对应于聚簇索引上的[name='d',id=10]的记录。

2.3 组合三：id 非唯一索引+RC

相对于组合一、二，组合三又发生了变化，隔离级别仍旧是 RC 不变，但是 id 列上的约束又降低了，id 列不再唯一，只有一个普通的索引。假设 delete from t1 where id = 10; 语句，仍旧选择 id 列上的索引进行过滤 where 条件，那么此时会持有哪些锁？同样见下图：



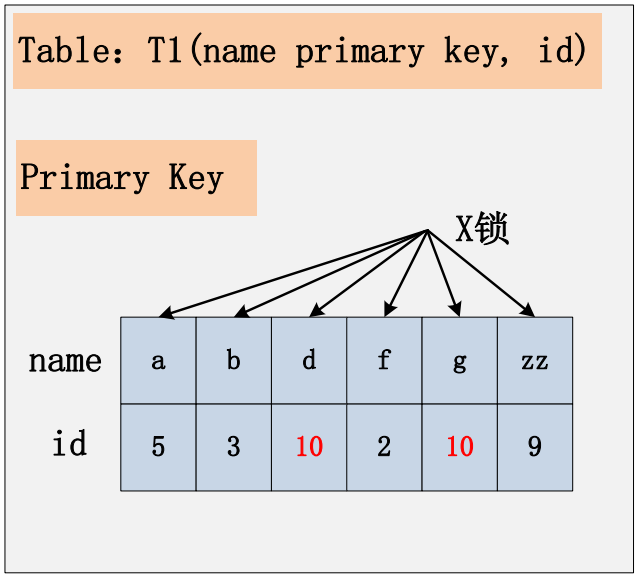
根据此图，可以看到，首先，id 列索引上，满足 id = 10 查询条件的记录，均已加锁。同时，

这些记录对应的主键索引上的记录也都加上了锁。与组合二唯一的区别在于，组合二最多只有一个满足等值查询的记录，而组合三会将所有满足查询条件的记录都加锁。

结论：若 id 列上有非唯一索引，那么对应的所有满足 SQL 查询条件的记录，都会被加锁。同时，这些记录在主键索引上的记录，也会被加锁。

2.4 组合四：id 无索引+RC

相对于前面三个组合，这是一个比较特殊的情况。id 列上没有索引，where id = 10;这个过滤条件，没法通过索引进行过滤，那么只能走全表扫描做过滤。对应于这个组合，SQL 会加什么锁？或者是换句话说，全表扫描时，会加什么锁？这个答案也有很多：有人说会在表上加 X 锁；有人说会将聚簇索引上，选择出来的 id = 10;的记录加上 X 锁。那么实际情况呢？请看下图：



由于 id 列上没有索引，因此只能走聚簇索引，进行全部扫描。从图中可以看到，满足删除条件的记录有两条，但是，聚簇索引上所有的记录，都被加上了 X 锁。无论记录是否满足条件，全部被加上 X 锁。既不是加表锁，也不是在满足条件的记录上加行锁。

有人可能会问？为什么不是只在满足条件的记录上加锁呢？这是由于 MySQL 的实现决定的。如果一个条件无法通过索引快速过滤，那么存储引擎层面就会将所有记录加锁后返回，然后由 MySQL Server 层进行过滤。因此也就把所有的记录，都锁上了。

注：在实际的实现中，MySQL 有一些改进，在 MySQL Server 过滤条件，发现不满足后，会调用 unlock_row 方法，把不满足条件的记录放锁 (违背了 2PL 的约束)。这样做，保证了最后只会持有满足条件记录上的锁，但是每条记录的加锁操作还是不能省略的。

结论：若 id 列上没有索引，SQL 会走聚簇索引的全扫描进行过滤，由于过滤是由 MySQL Server 层面进行的。因此每条记录，无论是否满足条件，都会被加上 X 锁。但是，为了效率考量，MySQL 做了优化，对于不满足条件的记录，会在判断后放锁，最终持有的，是满足条件的

记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略。同时，优化也违背了 2PL 的约束。

2.5 组合五：id 主键+RR

上面的四个组合，都是在 Read Committed 隔离级别下的加锁行为，接下来的四个组合，是在 Repeatable Read 隔离级别下的加锁行为。

组合五，id 列是主键列，Repeatable Read 隔离级别，针对 delete from t1 where id = 10; 这条 SQL，加锁与组合一：[id 主键，Read Committed]一致。

2.6 组合六：id 唯一索引+RR

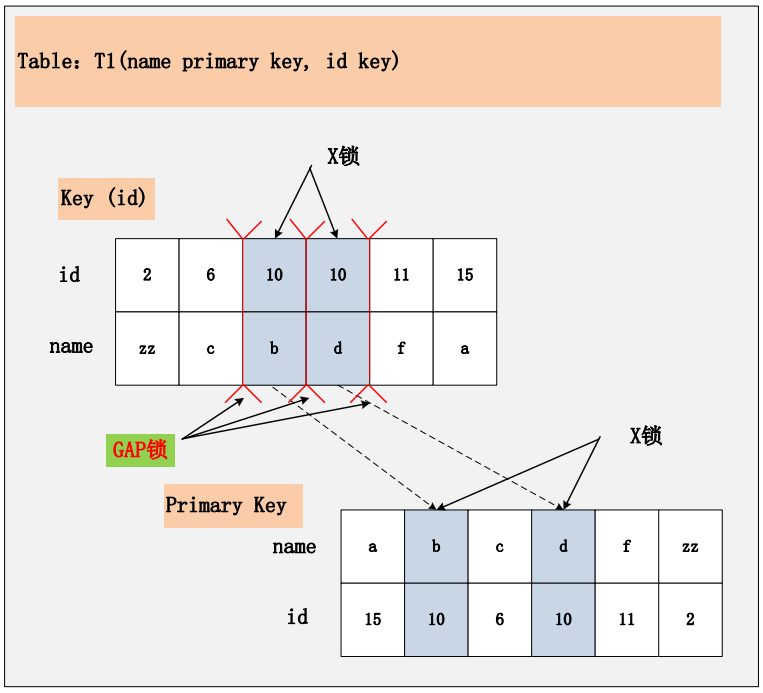
与组合五类似，组合六的加锁，与组合二：[id 唯一索引，Read Committed]一致。两个 X 锁，id 唯一索引满足条件的记录上一个，对应的聚簇索引上的记录一个。

注：根据博文《MySQL 加锁处理分析》下面的评论，id 为唯一索引，针对 id 的并发等值删除操作，有可能会产生死锁。具体死锁的场景与分析，可参考本人的另一篇文章：《一个最不可思议的 MySQL 死锁分析》。

2.7 组合七：id 非唯一索引+RR

还记得前面提到的 MySQL 的四种隔离级别的区别吗？RC 隔离级别允许幻读，而 RR 隔离级别，不允许存在幻读。但是在组合五、组合六中，加锁行为又是与 RC 下的加锁行为完全一致。那么 RR 隔离级别下，如何防止幻读呢？问题的答案，就在组合七中揭晓。

组合七，Repeatable Read 隔离级别，id 上有一个非唯一索引，执行 delete from t1 where id = 10; 假设选择 id 列上的索引进行条件过滤，最后的加锁行为，是怎么样的呢？同样看下面这幅图：



此图，相对于组合三：[id 列上非唯一锁，Read Committed]看似相同，其实却有很大的区别。最大的区别在于，这幅图中多了一个 GAP 锁，而且 GAP 锁看起来也不是加在记录上的，倒像是加载两条记录之间的位置，GAP 锁有何用？

其实这个多出来的 GAP 锁，就是 RR 隔离级别，相对于 RC 隔离级别，不会出现幻读的关键。确实，GAP 锁锁住的位置，也不是记录本身，而是两条记录之间的 GAP。所谓幻读，就是同一个事务，连续做两次当前读（例如：select * from t1 where id = 10 for update;），那么这两次当前读返回的是完全相同的记录（记录数量一致，记录本身也一致），第二次的当前读，不会比第一次返回更多的记录（幻象）。

如何保证两次当前读返回一致的记录，那就需要在第一次当前读与第二次当前读之间，其他的事务不会插入新的满足条件的记录并提交。为了实现这个功能，GAP 锁应运而生。

如图中所示，有哪些位置可以插入新的满足条件的项（id = 10），考虑到 B+树索引的有序性，满足条件的项一定是连续存放的。记录[6,c]之前，不会插入 id=10 的记录；[6,c]与[10,b]间可以插入[10, aa]；[10,b]与[10,d]间，可以插入新的[10,bb],[10,c]等；[10,d]与[11,f]间可以插入满足条件的[10,e],[10,z]等；而[11,f]之后也不会插入满足条件的记录。因此，为了保证[6,c]与[10,b]间，[10,b]与[10,d]间，[10,d]与[11,f]不会插入新的满足条件的记录，MySQL 选择了用 GAP 锁，将这三个 GAP 给锁起来。

Insert 操作，如 insert [10,aa]，首先会定位到[6,c]与[10,b]间，然后在插入前，会检查这个 GAP 是否已经被锁上，如果被锁上，则 Insert 不能插入记录。因此，通过第一遍的当前读，不仅将满足条件的记录锁上（X 锁），与组合三类似。同时还是增加 3 把 GAP 锁，将可能插入满足条件记录的 3 个 GAP 给锁上，保证后续的 Insert 不能插入新的 id=10 的记录，也就杜绝了同一事务的第二次当前读，出现幻象的情况。

有心的朋友看到这儿，可以会问：既然防止幻读，需要靠 GAP 锁的保护，为什么组合五、

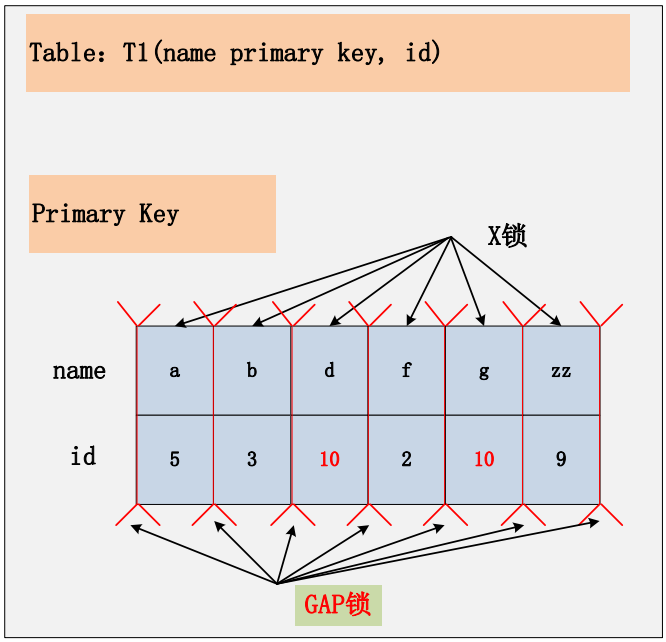
组合六，也是 RR 隔离级别，去不需要加 GAP 锁呢？

首先，这是一个好问题。其次，回答这个问题，也很简单。GAP 锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况。而组合五，id 是主键；组合六，id 是 unique 键，都能够保证唯一性。一个等值查询，最多只能返回一条记录，而且新的相同取值的记录，一定不会在新插入进来，因此也就避免了 GAP 锁的使用。其实，针对此问题，还有一个更深入的问题：如果组合五、组合六下，针对 SQL：select * from t1 where id = 10 for update; 第一次查询，没有找到满足查询条件的记录，那么 GAP 锁是否还能够省略？此问题留给大家思考。

结论：Repeatable Read 隔离级别下，id 列上有一个非唯一索引，对应 SQL：delete from t1 where id = 10; 首先，通过 id 索引定位到第一条满足查询条件的记录，加记录上的 X 锁，加 GAP 上的 GAP 锁，然后加主键聚簇索引上的记录 X 锁，然后返回；然后读取下一条，重复进行。直至进行到第一条不满足条件的记录[11,f]，此时，不需要加记录 X 锁，但是仍旧需要加 GAP 锁，最后返回结束。

2.8 组合八：id 无索引+RR

组合八，Repeatable Read 隔离级别下的最后一种情况，id 列上没有索引。此时 SQL：delete from t1 where id = 10; 没有其他的路径可以选择，只能进行全表扫描。最终的加锁情况，如下图所示：



如图，这是一个很恐怖的现象。首先，聚簇索引上的所有记录，都被加上了 X 锁。其次，聚簇索引每条记录间的间隙(GAP)，也同时被加上了 GAP 锁。这个示例表，只有 6 条记录，一共需要 6 个记录锁，7 个 GAP 锁。试想，如果表上有 1000 万条记录呢？

在这种情况下，这个表上，除了不加锁的快照度，其他任何加锁的并发 SQL，均不能执行，

不能更新，不能删除，不能插入，全表被锁死。

当然，跟组合四：[id 无索引, Read Committed]类似，这个情况下，MySQL 也做了一些优化，就是所谓的 semi-consistent read。semi-consistent read 开启的情况下，对于不满足查询条件的记录，MySQL 会提前放锁。针对上面的这个用例，就是除了记录[d,10]，[g,10]之外，所有的记录锁都会被释放，同时不加 GAP 锁。semi-consistent read 如何触发：要么是 read committed 隔离级别；要么是 Repeatable Read 隔离级别，但是设置了 innodb_locks_unsafe_for_binlog 参数。更详细的关于 semi-consistent read 的介绍，可参考我之前的一篇博客：MySQL+InnoDB semi-consitent read 原理及实现分析 。

结论：在 Repeatable Read 隔离级别下，如果进行全表扫描的当前读，那么会锁上表中的所有记录，同时会锁上聚簇索引内的所有 GAP，杜绝所有的并发 更新/删除/插入 操作。当然，也可以通过触发 semi-consistent read，来缓解加锁开销与并发影响，但是 semi-consistent read 本身也会带来其他问题，不建议使用。

2.9 组合九：Serializable

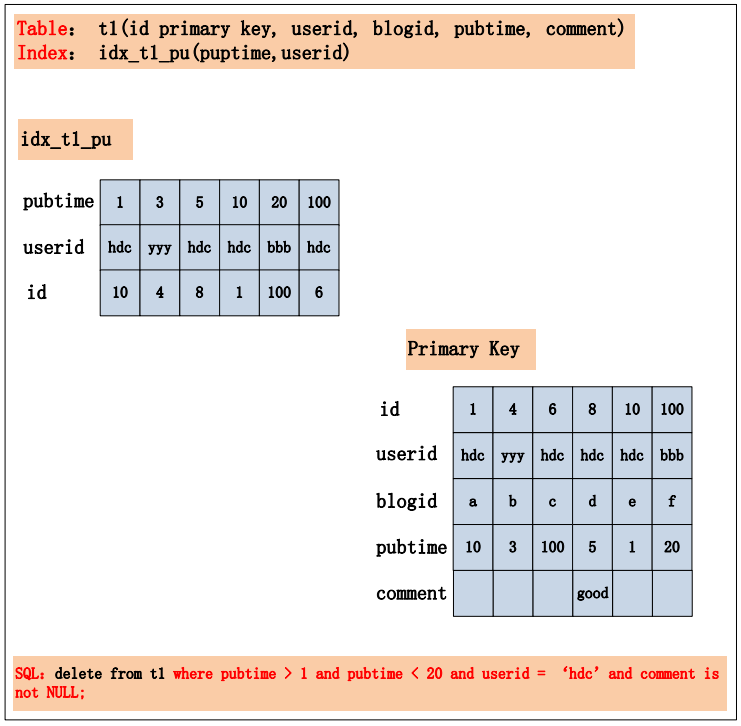
针对前面提到的简单的 SQL，最后一个情况：Serializable 隔离级别。对于 SQL2：delete from t1 where id = 10; 来说，Serializable 隔离级别与 Repeatable Read 隔离级别完全一致，因此不做介绍。

Serializable 隔离级别，影响的是 SQL1：select * from t1 where id = 10; 这条 SQL，在 RC，RR 隔离级别下，都是快照读，不加锁。但是在 Serializable 隔离级别，SQL1 会加读锁，也就是说快照读不复存在，MVCC 并发控制降级为 Lock-Based CC。

结论：在 MySQL/InnoDB 中，所谓的读不加锁，并不适用于所有的情况，而是隔离级别相关的。Serializable 隔离级别，读不加锁就不再成立，所有的读操作，都是当前读。

3 一条复杂的 SQL

写到这里，其实 MySQL 的加锁实现也已经介绍的八八九九。只要将本文上面的分析思路，大部分的 SQL，都能分析出其会加哪些锁。而这里，再来看一个稍微复杂点的 SQL，用于说明 MySQL 加锁的另外一个逻辑。SQL 用例如下：



如图中的 SQL，会加什么锁？假定在 Repeatable Read 隔离级别下 (Read Committed 隔离级别下的加锁情况，留给读者分析。)，同时，假设 SQL 走的是 idx_t1_pu 索引。

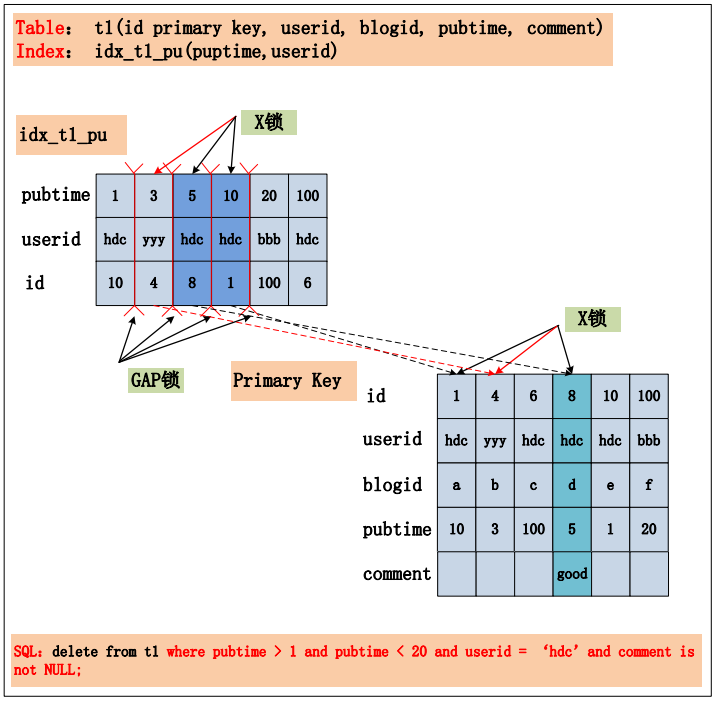
在详细分析这条 SQL 的加锁情况前，还需要有一个知识储备，那就是一个 SQL 中的 where 条件如何拆分？具体的介绍，建议阅读我之前的一篇文章：SQL 中的 where 条件，在数据库中提取与应用浅析 。在这里，我直接给出分析后的结果：

- **Index key：** pubtime > 1 and puptime < 20。此条件，用于确定 SQL 在 idx_t1_pu 索引上的查询范围。

- **Index Filter：** userid = 'hdc' 。此条件，可以在 idx_t1_pu 索引上进行过滤，但不属于 Index Key。

- **Table Filter：** comment is not NULL。此条件，在 idx_t1_pu 索引上无法过滤，只能在聚簇索引上过滤。

在分析出 SQL where 条件的构成之后，再来看看这条 SQL 的加锁情况 (RR 隔离级别)，如下图所示：



从图中可以看出，在 Repeatable Read 隔离级别下，由 Index Key 所确定的范围，被加上了 GAP 锁；Index Filter 锁给定的条件 (userid = 'hdc')何时过滤，视 MySQL 的版本而定，在 MySQL 5.6 版本之前，不支持 Index Condition Pushdown(ICP)，因此 Index Filter 在 MySQL Server 层过滤，在 5.6 后支持了 Index Condition Pushdown，则在 index 上过滤。若不支持 ICP，不满足 Index Filter 的记录，也需要加上记录 X 锁，若支持 ICP，则不满足 Index Filter 的记录，无需加记录 X 锁 (图中，用红色箭头标出的 X 锁，是否要加，视是否支持 ICP 而定)；而 Table Filter 对应的过滤条件，则在聚簇索引中读取后，在 MySQL Server 层面过滤，因此聚簇索引上也需要 X 锁。最后，选取出了一条满足条件的记录[8,hdc,d,5,good]，但是加锁的数量，要远远大于满足条件的记录数量。

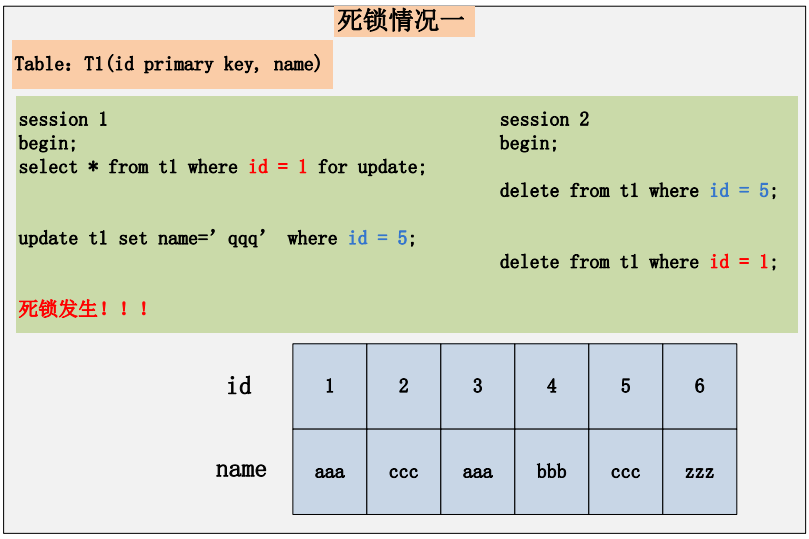
结论：在 Repeatable Read 隔离级别下，针对一个复杂的 SQL，首先需要提取其 where 条件。Index Key 确定的范围，需要加上 GAP 锁；Index Filter 过滤条件，视 MySQL 版本是否支持 ICP，若支持 ICP，则不满足 Index Filter 的记录，不加 X 锁，否则需要 X 锁；Table Filter 过滤条件，无论是否满足，都需要加 X 锁。

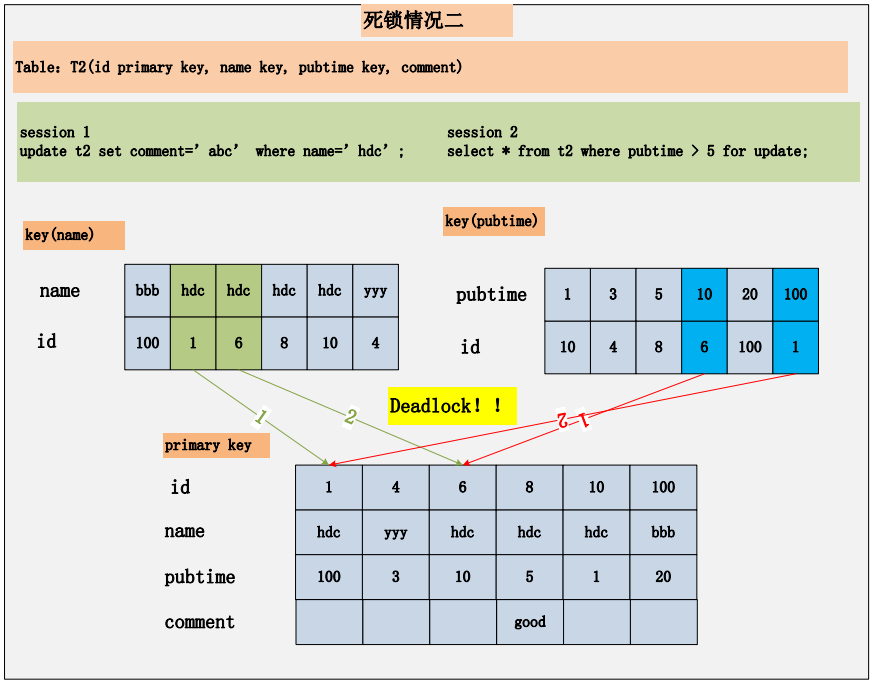
4 死锁原理与分析

本文前面的部分，基本上已经涵盖了 MySQL/InnoDB 所有的加锁规则。深入理解 MySQL 如何加锁，有两个比较重要的作用：

- 可以根据 MySQL 的加锁规则，写出不会发生死锁的 SQL；
- 可以根据 MySQL 的加锁规则，定位出线上产生死锁的原因；

下面，来看看两个死锁的例子 (一个是两个 Session 的两条 SQL 产生死锁；另一个是两个 Session 的一条 SQL，产生死锁)：





上面的两个死锁用例。第一个非常好理解，也是最常见的死锁，每个事务执行两条 SQL，分别持有了一把锁，然后加另一把锁，产生死锁。

第二个用例，虽然每个 Session 都只有一条语句，仍旧会产生死锁。要分析这个死锁，首先必须用到本文前面提到的 MySQL 加锁的规则。针对 Session 1，从 name 索引出发，读到的[hdc, 1]，[hdc, 6]均满足条件，不仅会加 name 索引上的记录 X 锁，而且会加聚簇索引上的记录 X 锁，加锁顺序为先[1,hdc,100]，后[6,hdc,10]。而 Session 2，从 pubtime 索引出发，[10,6],[100,1]均满足过滤条件，同样也会加聚簇索引上的记录 X 锁，加锁顺序为[6,hdc,10]，后[1,hdc,100]。发现没有，跟 Session 1 的加锁顺序正好相反，如果两个 Session 恰好都持有了第一把锁，请求加第二把锁，死锁就发生了。

结论：死锁的发生与否，并不在于事务中有多少条 SQL 语句，**死锁的关键在于：**两个(或以

上)的 Session 加锁的顺序不一致。而使用本文上面提到的，分析 MySQL 每条 SQL 语句的加锁规则，分析出每条语句的加锁顺序，然后检查多个并发 SQL 间是否存在以相反的顺序加锁的情况，就可以分析出各种潜在的死锁情况，也可以分析出线上死锁发生的原因。

5 总结

写到这儿，本文也告一段落，做一个简单的总结，要做的完全掌握 MySQL/InnoDB 的加锁规则，甚至是其他任何数据库的加锁规则，需要具备以下的一些知识点：

- 了解数据库的一些基本理论知识：数据的存储格式 (堆组织表 vs 聚簇索引表)；并发控制协议 (MVCC vs Lock-Based CC)；Two-Phase Locking；数据库的隔离级别定义 (Isolation Level)；
- 了解 SQL 本身的执行计划 (主键扫描 vs 唯一键扫描 vs 范围扫描 vs 全表扫描)；
- 了解数据库本身的一些实现细节 (过滤条件提取；Index Condition Pushdown；Semi-Consistent Read)；
- 了解死锁产生的原因及分析的方法 (加锁顺序不一致；分析每个 SQL 的加锁顺序)

有了这些知识点，再加上适当的实战经验，全面掌控 MySQL/InnoDB 的加锁规则，当不在话下。