超全面MySQL语句加锁分析（上篇）（求转）

原创 小孩子4919 我们都是小青蛙 2019-05-07

说在前面的话

本文是用来系统阐述在MySQL中，不同语句在各种条件下的加锁情况，并不是解释各种锁是什么（或者说加锁的本质是什么），大家如果不理解什么是 MVCC 、 ReadView 、 正经记录锁 、 gap 锁 、 next-key锁 、 插入意向锁 这些概念的，可以参考 MySQL 的官方文档，或者直接参照《MySQL是怎样运行的：从根儿上理解MySQL》这本小册（里边有比官方文档更贴心，更详细的解释，文章中涉及到的所有概念均在小册中有详细解释，有疑惑，并且有兴趣的同学可以扫描下边二维码看看）：

建议：

1. 本篇文章不适合碎片化时间阅读，最好使用电脑观看，或者将字体跳到最小效果好一些
2. 可能一下子看不完，关注 + 收藏 + 好看 + 转发一波
3. 不要跳着看

事前准备

建立一个存储三国英雄的 hero 表：

```
CREATE TABLE hero (
    number INT,
```

```
    name VARCHAR(100),
    country varchar(100),
    PRIMARY KEY (number),
    KEY idx_name (name)
) Engine=InnoDB CHARSET=utf8;
```

然后向这个表里插入几条记录：

```
INSERT INTO hero VALUES
    (1, 'l刘备', '蜀'),
    (3, 'z诸葛亮', '蜀'),
    (8, 'c曹操', '魏'),
    (15, 'x荀彧', '魏'),
    (20, 's孙权', '吴');
```

然后现在 hero 表就有了两个索引（一个二级索引，一个聚簇索引），示意图如下：

语句加锁分析

其实啊，“XXX语句该加什么锁”本身就是个伪命题，一条语句需要加的锁受到很多条件制约，比方说：

- 事务的隔离级别
- 语句执行时使用的索引（比如聚簇索引、唯一二级索引、普通二级索引）
- 查询条件（比方说 = 、 =< 、 >= 等等）
- 具体执行的语句类型

在继续详细分析语句的加锁过程前，大家一定要有一个全局概念： 加锁 只是解决并发事务执行过程中引起的 脏写 、 脏读 、 不可重复读 、 幻读 这些问题的一种解决方案（ MVCC 算是一种解决 脏读 、 不可重复读 、 幻读 这些问题的一种解决方案），一定要意识到 加锁 的出发点是为了解决这些问题，不同情景下要解决的问题不一样，才导致加的锁不一样，千万不要为了加锁而加锁，容易把自己绕进

去。当然，有时候因为 MySQL 具体的实现而导致一些情景下的加锁有些不太好理解，这就得我们死记硬背了～

我们这里把语句分为3种大类：普通的 SELECT 语句、锁定读的语句、 INSERT 语句，我们分别看一下。

普通的SELECT语句

普通的 SELECT 语句在：

- READ UNCOMMITTED 隔离级别下，不加锁，直接读取记录的最新版本，可能发生 脏读 、 不可重复读 和 幻读 问题。
- READ COMMITTED 隔离级别下，不加锁，在每次执行普通的 SELECT 语句时都会生成一个 ReadView ，这样解决了 脏读 问题，但没有解决 不可重复读 和 幻读 问题。
- REPEATABLE READ 隔离级别下，不加锁，只在第一次执行普通的 SELECT 语句时生成一个 ReadView ，这样把 脏读 、 不可重复读 和 幻读 问题都解决了。

  不过这里有一个小插曲：

```
# 事务T1，REPEATABLE READ隔离级别下
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM hero WHERE number = 30;
Empty set (0.01 sec)

# 此时事务T2执行了：INSERT INTO hero VALUES(30, 'g关羽', '魏'); 并提交

mysql> UPDATE hero SET country = '蜀' WHERE number = 30;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM hero WHERE number = 30;
+--------+---------+---------+
| number | name    | country |
+--------+---------+---------+
|     30 | g关羽   | 蜀      |
```

```
+--------+---------+---------+
1 row in set (0.01 sec)
```

在 REPEATABLE READ 隔离级别下， T1 第一次执行普通的 SELECT 语句时生成了一个 ReadView ，之后 T2 向 hero 表中新插入了一条记录便提交了， ReadView 并不能阻止 T1 执行 UPDATE 或者 DELETE 语句来对改动这个新插入的记录（因为 T2 已经提交，改动该记录并不会造成阻塞），但是这样一来这条新记录的 trx_id 隐藏列就变成了 T1 的 事务id ，之后 T1 中再使用普通的 SELECT 语句去查询这条记录时就可以看到这条记录了，也就把这条记录返回给客户端了。因为这个特殊现象的存在，你也可以认为 InnoDB 中的 MVCC 并不能完完全全的禁止幻读。

- SERIALIZABLE 隔离级别下，需要分为两种情况讨论：
  - 在系统变量 autocommit=0 时，也就是禁用自动提交时，普通的 SELECT 语句会被转为 SELECT ... LOCK IN SHARE MODE 这样的语句，也就是在读取记录前需要先获得记录的 S锁 ，具体的加锁情况和 REPEATABLE READ 隔离级别下一样，我们后边再分析。
  - 在系统变量 autocommit=1 时，也就是启用自动提交时，普通的 SELECT 语句并不加锁，只是利用 MVCC 来生成一个 ReadView 去读取记录。

    为啥不加锁呢？因为启用自动提交意味着一个事务中只包含一条语句，一条语句也就没有啥 不可重复读 、 幻读 这样的问题了。

锁定读的语句

我们把下边四种语句放到一起讨论：

- 语句一： SELECT ... LOCK IN SHARE MODE;
- 语句二： SELECT ... FOR UPDATE;
- 语句三： UPDATE ...
- 语句四： DELETE ...

我们说 语句一 和 语句二 是 MySQL 中规定的两种 锁定读 的语法格式，而 语句三 和 语句四 由于在执行过程需要首先定位到被改动的记录并给记录加锁，也可以被认为是一种 锁定读 。

READ UNCOMMITTED/READ COMMITTED隔离级别下

在 READ UNCOMMITTED 下语句的加锁方式和 READ COMMITTED 隔离级别下语句的加锁方式基本一致，所以就放到一块儿说了。值得注意的是，采用 加锁 方式解决并发事务带来的问题时，其实 脏读 和 不可重复读 在任何一个隔离级别下都不会发生（因为 读-写 操作需要排队进行）。

对于使用主键进行等值查询的情况

- 使用 SELECT ... LOCK IN SHARE MODE 来为记录加锁，比方说：

```
SELECT * FROM hero WHERE number = 8 LOCK IN SHARE MODE;
```

这个语句执行时只需要访问一下聚簇索引中 number 值为 8 的记录，所以只需要给它加一个 S型正经记录锁 就好了，如图所示：

- 使用 SELECT ... FOR UPDATE 来为记录加锁，比方说：

```
SELECT * FROM hero WHERE number = 8 FOR UPDATE;
```

这个语句执行时只需要访问一下聚簇索引中 number 值为 8 的记录，所以只需要给它加一个 X型正经记录锁 就好了，如图所示：

> 小贴士： 为了区分S锁和X锁，我们之后在示意图中就把加了S锁的记录染成蓝色，把加了X锁的记录染成紫色。

- 使用 UPDATE ... 来为记录加锁，比方说：

```
UPDATE hero SET country = '汉' WHERE number = 8;
```

这条 UPDATE 语句并没有更新二级索引列，加锁方式和上边所说的 SELECT ... FOR UPDATE 语句一致。

如果 UPDATE 语句中更新了二级索引列，比方说：

```
UPDATE hero SET name = 'cao曹操' WHERE number = 8;
```

该语句的实际执行步骤是首先更新对应的 number 值为 8 的聚簇索引记录，再更新对应的二级索引记录，所以加锁的步骤就是：

1. 为 number 值为 8 的聚簇索引记录加上 X型正经记录锁 （该记录对应的）。
2. 为该聚簇索引记录对应的 idx_name 二级索引记录（也就是 name 值为 'c曹操'， number 值为 8 的那条二级索引记录）加上 X型正经记录锁 。

画个图就是这样：

> 小贴士： 我们用带圆圈的数字来表示为各条记录加锁的顺序。

- 使用 DELETE ... 来为记录加锁，比方说：

```
DELETE FROM hero WHERE number = 8;
```

我们平时所说的“DELETE表中的一条记录”其实意味着对聚簇索引和所有的二级索引中对应的记录做 DELETE 操作，本例子中就是要先把 number 值为 8 的聚簇索引记录执行 DELETE 操作，然后把对应的 idx_name 二级索引记录删除，所以加锁的步骤和上边更新带有二级索引列的 UPDATE 语句一致，就不画图了。

对于使用主键进行范围查询的情况

- 使用 SELECT ... LOCK IN SHARE MODE 来为记录加锁，比方说：

```
SELECT * FROM hero WHERE number <= 8 LOCK IN SHARE MODE;
```

这个语句看起来十分简单，但它的执行过程还是有一丢丢小复杂的：

1. 先到聚簇索引中定位到满足 number <= 8 的第一条记录，也就是 number 值为 1 的记录，然后为其加锁。

2. 判断一下该记录是否符合 索引条件下推 中的条件。

   我们前边介绍过一个称之为 索引条件下推 （ Index Condition Pushdown，简称 ICP ）的功能，也就是把查询中与被使用索引有关的查询条件下推到存储引擎中判断，而不是返回到 server 层再判断。不过需要注意的是， 索引条件下推 只是为了减少回表次数，也就是减少读取完整的聚簇索引记录的次数，从而减少 IO 操作。而对于 聚簇索引 而言不需要回表，它本身就包含着全部的列，也起不到减少 IO 操作的作用，所以设计 InnoDB 的大叔们规定这个 索引条件下推 特性只适用于 二级索引 。也就是说在本例中与被使用索引有关的条件是： number <= 8 ，而 number 列又是聚簇索引列，所以本例中并没有符合 索引条件下推 的查询条件，自然也就不需要判断该记录是否符合 索引条件下推 中的条件。

3. 判断一下该记录是否符合范围查询的边界条件

   因为在本例中是利用主键 number 进行范围查询，设计 InnoDB 的大叔规定每从聚簇索引中取出一条记录时都要判断一下该记录是否符合范围查询的边界条件，也就是 number <= 8 这个条件。如果符合的话将其返回给 server层 继续处理，否则的话需要释放掉在该记录上加的锁，并给 server层 返回一个查询完毕的信息。

   对于 number 值为 1 的记录是符合这个条件的，所以会将其返回到 server层 继续处理。

4. 将该记录返回到 server层 继续判断。

   server层 如果收到存储引擎层提供的查询完毕的信息，就结束查询，否则继续判断那些没有进行 索引条件下推 的条件，在本例中就是继续判断 number <= 8 这个条件是否成立。噫，不是在第3步中已经判断过了么，怎么在这又判断一回？是的，设计 InnoDB 的大叔采用的策略就是这么简单粗暴，把凡是没有经过 索引条件下推 的条件都需要放到 server 层再判断一遍。如果该记录符合剩余的条件（没有进行 索引条件下推 的条件），那么就把它发送给客户端，不然的话需要释放掉在该记录上加的锁。

5. 然后刚刚查询得到的这条记录（也就是 number 值为 1 的记录）组成的单向链表继续向后查找，得到了 number 值为 3 的记录，然后重复第 2 ， 3 ， 4 、 5 这几个步骤。

> 小贴士： 上述步骤是在MySQL 5.7.21这个版本中验证的，不保证其他版本有无出入。

但是这个过程有个问题，就是当找到 number 值为 8 的那条记录的时候，还得向后找一条记录（也就是 number 值为 15 的记录），在存储引擎读取这条记录的时候，也就是上述的第 1 步中，就得为这条记录加锁，然后在第3步时，判断该记录不符合 number <= 8 这个条件，又要释放掉这条记录的锁，这个过程导致 number 值为 15 的记录先被加锁，然后把锁释放掉，过程就是这样：

这个过程有意思的一点就是，如果你先在事务 T1 中执行：

```
# 事务T1
BEGIN;
SELECT * FROM hero WHERE number <= 8 LOCK IN SHARE MODE;
```

然后再到事务 T2 中执行：

```
# 事务T2
BEGIN;
SELECT * FROM hero WHERE number = 15 FOR UPDATE;
```

是没有问题的，因为在 T2 执行时，事务 T1 已经释放掉了 number 值为 15 的记录的锁，但是如果你先执行 T2 ，再执行 T1 ，由于 T2 已经持有了 number 值为 15 的记录的锁，事务 T1 将因为获取不到这个锁而等待。

我们再看一个使用主键进行范围查询的例子：

```
SELECT * FROM hero WHERE number >= 8 LOCK IN SHARE MODE;
```

这个语句的执行过程其实和我们举的上一个例子类似。也是先到聚簇索引中定位到满足 number >= 8 这个条件的第一条记录，也就是 number 值为 8 的记录，然后就可以沿着由记录组成的单

向链表一路向后找，每找到一条记录，就会为其加上锁，然后判断该记录符不符合范围查询的边界条件，不过这里的边界条件比较特殊： number >= 8 ，只要记录不小于8就算符合边界条件，所以判断和没判断是一样一样的。最后把这条记录返回给 server层 ， server层 再判断 number >= 8 这个条件是否成立，如果成立的话就发送给客户端，否则的话就结束查询。不过 InnoDB 存储引擎找到索引中的最后一条记录，也就是 Supremum 伪记录之后，在存储引擎内部就可以立即判断这是一条伪记录，不必要返回给 server层 处理，也没必要给它也加上锁（也就是说在第1步中就压根儿没给这条记录加锁）。整个过程会给 number 值为 8 、 15 、 20 这三条记录加上 S型正经记录锁 ，画个图表示一下就是这样：

- 使用 SELECT ... FOR UPDATE 语句来为记录加锁：

  和 SELECT ... FOR UPDATE 语句类似，只不过加的是 X型正经记录锁 。

- 使用 UPDATE ... 来为记录加锁，比方说：

```
UPDATE hero SET country = '汉' WHERE number >= 8;
```

这条 UPDATE 语句并没有更新二级索引列，加锁方式和上边所说的 SELECT ... FOR UPDATE 语句一致。

如果 UPDATE 语句中更新了二级索引列，比方说：

```
UPDATE hero SET name = 'cao曹操' WHERE number >= 8;
```

这时候会首先更新聚簇索引记录，再更新对应的二级索引记录，所以加锁的步骤就是：

1. 为 number 值为 8 的聚簇索引记录加上 X型正经记录锁 。
2. 然后为上一步中的记录索引记录对应的 idx_name 二级索引记录加上 X型正经记录锁 。
3. 为 number 值为 15 的聚簇索引记录加上 X型正经记录锁 。
4. 然后为上一步中的记录索引记录对应的 idx_name 二级索引记录加上 X型正经记录锁 。
5. 为 number 值为 20 的聚簇索引记录加上 X型正经记录锁 。

6. 然后为上一步中的记录索引记录对应的 idx_name 二级索引记录加上 X型正经记录锁 。

画个图就是这样：

如果是下边这个语句：

```
UPDATE hero SET namey = '汉' WHERE number <= 8;
```

则会对 number 值为 1 、 3 、 8 聚簇索引记录以及它们对应的二级索引记录加 X型正经记录锁 ，加锁顺序和上边语句中的加锁顺序类似，都是先对一条聚簇索引记录加锁后，再给对应的二级索引记录加锁。之后会继续对 number 值为 15 的聚簇索引记录加锁，但是随后 InnoDB 存储引擎判断它不符合边界条件，随即会释放掉该聚簇索引记录上的锁（注意这个过程中没有对 number 值为 15 的聚簇索引记录对应的二级索引记录加锁）。具体示意图就不画了。

- 使用 DELETE ... 来为记录加锁，比方说：

```
DELETE FROM hero WHERE number >= 8;
```

和

```
DELETE FROM hero WHERE number <= 8;
```

这两个语句的加锁情况和更新带有二级索引列的 UPDATE 语句一致，就不画图了。

对于使用二级索引进行等值查询的情况

> 小贴士： 在READ UNCOMMITTED和READ COMMITTED隔离级别下，使用普通的二级索引和唯一二级索引进行加锁的过程是一样的，所以我们也就不分开讨论了。

- 使用 SELECT ... LOCK IN SHARE MODE 来为记录加锁，比方说：

```
SELECT * FROM hero WHERE name = 'c曹操' LOCK IN SHARE MODE;
```

这个语句的执行过程是先通过二级索引 idx_name 定位到满足 name = 'c曹操' 条件的二级索引记录，然后进行回表操作。所以先要对二级索引记录加 S型正经记录锁 ，然后再给对应的聚簇索引记录加 S型正经记录锁 ，示意图如下：

这里需要再次强调一下这个语句的加锁顺序：

1. 先对 name 列为 'c曹操' 二级索引记录进行加锁。
2. 再对相应的聚簇索引记录进行加锁

> 小贴士： 我们知道idx_name是一个普通的二级索引，到idx_name索引中定位到满足name= 'c曹操'这个条件的第一条记录后，就可以沿着这条记录一路向后找。可是从我们上边的描述中可以看出来，并没有对下一条二级索引记录进行加锁，这是为什么呢？这是因为设计InnoDB的大叔对等值匹配的条件有特殊处理，他们规定在InnoDB存储引擎层查找到当前记录的下一条记录时，在对其加锁前就直接判断该记录是否满足等值匹配的条件，如果不满足直接返回（也就是不加锁了），否则的话需要将其加锁后再返回给server层。所以这里也就不需要对下一条二级索引记录进行加锁了。

现在要介绍一个非常有趣的事情，我们假设上边这个语句在事务 T1 中运行，然后事务 T2 中运行下边一个我们之前介绍过的语句：

```
UPDATE hero SET name = '曹操' WHERE number = 8;
```

这两个语句都是要对 number 值为 8 的聚簇索引记录和对应的二级索引记录加锁，但是不同点是加锁的顺序不一样。这个 UPDATE 语句是先对聚簇索引记录进行加锁，后对二级索引记录进行加锁，如果在不同事务中运行上述两个语句，可能发生一种贼奇妙的事情——

- 事务 T2 持有了聚簇索引记录的锁，事务 T1 持有了二级索引记录的锁。
- 事务 T2 在等待获取二级索引记录上的锁，事务 T1 在等待获取聚簇索引记录上的锁。

两个事务都分别持有一个锁，而且都在等待对方已经持有的那个锁，这种情况就是所谓的 死锁 ，两个事务都无法运行下去，必须选择一个进行回滚，对性能影响比较大。

- 使用 SELECT ... FOR UPDATE 语句时，比如：

```
SELECT * FROM hero WHERE name = 'c曹操' FOR UPDATE;
```

这种情况下与 SELECT ... LOCK IN SHARE MODE 语句的加锁情况类似，都是给访问到的二级索引记录和对应的聚簇索引记录加锁，只不过加的是 X型正经记录锁 罢了。

- 使用 UPDATE ... 来为记录加锁，比方说：

与更新二级索引记录的 SELECT ... FOR UPDATE 的加锁情况类似，不过如果被更新的列中还有别的二级索引列的话，对应的二级索引记录也会被加锁。

- 使用 DELETE ... 来为记录加锁，比方说：

与 SELECT ... FOR UPDATE 的加锁情况类似，不过如果表中还有别的二级索引列的话，对应的二级索引记录也会被加锁。

对于使用二级索引进行范围查询的情况

- 使用 SELECT ... LOCK IN SHARE MODE 来为记录加锁，比方说：

```
SELECT * FROM hero FORCE INDEX(idx_name)  WHERE name >= 'c曹操' LOCK IN SHARE MODE;
```

> 小贴士： 因为优化器会计算使用二级索引进行查询的成本，在成本较大时可能选择以全表扫描的方式来执行查询，所以我们这里使用FORCE INDEX(idx_name)来强制使用二级索引idx_name来执行查询。

这个语句的执行过程其实是先到二级索引中定位到满足 name >= 'c曹操' 的第一条记录，也就是 name 值为 c曹操 的记录，然后就可以沿着这条记录的链表一路向后找，从二级索引 idx_name 的示意图中可以看出，所有的用户记录都满足 name >= 'c曹操' 的这个条件，所以所有的二级索引记录都会被加 S型正经记录锁 ，它们对应的聚簇索引记录也会被加 S型正经记录锁 。不过需要注意一下加锁顺序，对一条二级索引记录加锁完后，会接着对它相应的聚簇索引记录加锁，完后才会对下一条二级索引记录进行加锁，以此类推～ 画个图表示一下就是这样：

再来看下边这个语句：

```
SELECT * FROM hero FORCE INDEX(idx_name) WHERE name <= 'c曹操' LOCK IN SHARE MODE;
```

这个语句的加锁情况就有点儿有趣了。前边说在使用 number <= 8 这个条件的语句中，需要把 number 值为 15 的记录也加一个锁，之后又判断它不符合边界条件而把锁释放掉。而对于查询条件 name <= 'c曹操' 的语句来说，执行该语句需要使用到二级索引，而与二级索引相关的条件是可以使用 索引条件下推 这个特性的。设计 InnoDB 的大叔规定，如果一条记录不符合 索引条件下推 中的条件的话，直接跳到下一条记录（这个过程根本不将其返回到 server层 ），如果这已经是最后一条记录，那么直接向 server层 报告查询完毕。但是这里头有个问题呀：先对一条记录加了锁，然后再判断该记录是不是符合索引条件下推的条件，如果不符合直接跳到下一条记录或者直接向server层报告查询完毕，这个过程中并没有把那条被加锁的记录上的锁释放掉呀！！！。本例中使用的查询条件是 name <= 'c曹操' ，在为 name 值为 'c曹操' 的二级索引记录以及它对应的聚簇索引加锁之后，会接着二级索引中的下一条记录，也就是 name 值为 'l刘备' 的那条二级索引记录，由于该记录不符合 索引条件下推 的条件，而且是范围查询的最后一条记录，会直接向 server层 报告查询完毕，重点是这个过程中并不会释放 name 值为 'l刘备' 的二级索引记录上的锁，也就导致了语句执行完毕时的加锁情况如下所示：

这样子会造成一个尴尬情况，假如 T1 执行了上述语句并且尚未提交， T2 再执行这个语句：

```
SELECT * FROM hero WHERE name = 'l刘备' FOR UPDATE;
```

T2 中的语句需要获取 name 值为 l刘备 的二级索引记录上的 X型正经记录锁 ， 而 T1 中仍然持有 name 值为 l刘备 的二级索引记录上的 S型正经记录锁 ， 这就造成了 T2 获取不到锁而进入等待状态。

> 小贴士： 为啥不能释放不符合索引条件下推中的条件的二级索引记录上的锁呢？这个问题我也没想明白，人家就是这么规定的，如果有明白的小伙伴可以加我微信 xiaohaizi4919 来讨论一下哈～ 再强调一下，我使用的MySQL版本是5.7.21，不保证其他版本中的加锁情景是否完全一致。

- 使用 SELECT ... FOR UPDATE 语句时：

  和 SELECT ... FOR UPDATE 语句类似，只不过加的是 X型正经记录锁 。

- 使用 UPDATE ... 来为记录加锁，比方说：

```
UPDATE hero SET country = '汉' WHERE name >= 'c曹操';
```

> 小贴士： FORCE INDEX只对SELECT语句起作用，UPDATE语句虽然支持该语法，但实质上不起作用，DELETE语句压根儿不支持该语法。

假设该语句执行时使用了 idx_name 二级索引来进行 锁定读 ，那么它的加锁方式和上边所说的 SELECT ... FOR UPDATE 语句一致。如果有其他二级索引列也被更新，那么也会为对应的二级索引记录进行加锁，就不赘述了。不过还有一个有趣的情况，比方说：

```
UPDATE hero SET country = '汉' WHERE name <= 'c曹操';
```

我们前边说的 索引条件下推 这个特性只适用于 SELECT 语句，也就是说 UPDATE 语句中无法使用，那么这个语句就会为 name 值为 'c曹操' 和 'l刘备' 的二级索引记录以及它们对应的聚簇索引进行加锁，之后在判断边界条件时发现 name 值为 'l刘备' 的二级索引记录不符合 name <= 'c曹操' 条件，再把该二级索引记录和对应的聚簇索引记录上的锁释放掉。这个过程如下图所示：

- 使用 DELETE ... 来为记录加锁，比方说：

```
DELETE FROM hero WHERE name >= 'c曹操';
```

和

```
DELETE FROM hero WHERE name <= 'c曹操';
```

如果这两个语句采用二级索引来进行 锁定读 ，那么它们的加锁情况和更新带有二级索引列的 UPDATE 语句一致，就不画图了。

全表扫描的情况

比方说：

```
SELECT * FROM hero WHERE country  = '魏' LOCK IN SHARE MODE;
```

由于 country 列上未建索引，所以只能采用全表扫描的方式来执行这条查询语句，存储引擎每读取一条聚簇索引记录，就会为这条记录加锁一个 S型正常记录锁 ，然后返回给 server层 ，如果 server层 判断 country = '魏' 这个条件是否成立，如果成立则将其发送给客户端，否则会释放掉该记录上的锁，画个图就像这样：

使用 SELECT ... FOR UPDATE 进行加锁的情况与上边类似，只不过加的是 X型正经记录锁 ，就不赘述了。

对于 UPDATE ... 和 DELETE ... 的语句来说，在遍历聚簇索引中的记录，都会为该聚簇索引记录加上 X型正经记录锁 ，然后：

- 如果该聚簇索引记录不满足条件，直接把该记录上的锁释放掉。

- 如果该聚簇索引记录满足条件，则会对相应的二级索引记录加上 X型正经记录锁（ DELETE 语句会对所有二级索引列加锁， UPDATE 语句只会为更新的二级索引列对应的二级索引记录加锁）。

未完待续

下一章节继续唠叨在 REPEATABLE READ隔离级别下，各种语句的加锁情况，以及INSERT语句的加锁情况，敬请期待。如果文中有任何问题，请联系作者：xiaohaizi4919（正经微信，扯犊子的请勿添加）。

关注小青蛙，全都是技术干货哈：

原文链接

大家可以点击原文链接，查看《MySQL是怎样运行的：从根儿上理解MySQL》的完整内容，通俗到爆炸💥的MySQL进阶读物，原来学习可以这么有趣～

文章已于2019-05-08修改

阅读原文