

21 | 为什么我只改一行的语句，锁这么多？

2018-12-31 林晓斌



在上一篇文章中，我和你介绍了间隙锁和next-key lock的概念，但是并没有说明加锁规则。间隙锁的概念理解起来确实有点儿难，尤其在配合上行锁以后，很容易在判断是否会出现锁等待的问题上犯错。

所以今天，我们就先从这个加锁规则开始吧。

首先说明一下，这些加锁规则我没在别的地方看到过有类似的总结，以前我自己判断的时候都是想着代码里面的实现来脑补的。这次为了总结成不看代码的同学也能理解的规则，是我又重新刷了代码临时总结出来的。所以，这个规则有以下两条前提说明：

1. MySQL后面的版本可能会改变加锁策略，所以这个规则只限于截止到现在的最新版本，即5.x系列 $\leq 5.7.24$ ，8.0系列 $\leq 8.0.13$ 。
2. 如果大家在验证中有发现bad case的话，请提出来，我会再补充进这篇文章，使得一起学习本专栏的所有同学都能受益。

因为间隙锁在可重复读隔离级别下才有效，所以本篇文章接下来的描述，若没有特殊说明，默认是可重复读隔离级别。

我总结的加锁规则里面，包含了两个“原则”、两个“优化”和一个“bug”。

1. 原则1：加锁的基本单位是next-key lock。希望你还记得，next-key lock是前开后闭区间。

- 2. 原则2: 查找过程中访问到的对象才会加锁。
- 3. 优化1: 索引上的等值查询，给唯一索引加锁的时候，next-key lock退化为行锁。
- 4. 优化2: 索引上的等值查询，向右遍历时且最后一个值不满足等值条件的时候，next-key lock退化为间隙锁。
- 5. 一个bug: 唯一索引上的范围查询会访问到不满足条件的第一个值为止。

我还是以上篇文章的表t为例，和你解释一下这些规则。表t的建表语句和初始化语句如下。

```
CREATE TABLE `t` (  
  `id` int(11) NOT NULL,  
  `c` int(11) DEFAULT NULL,  
  `d` int(11) DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `c` (`c`)  
) ENGINE=InnoDB;  
  
insert into t values(0,0,0),(5,5,5),  
(10,10,10),(15,15,15),(20,20,20),(25,25,25);
```

接下来的例子基本都是配合着图片说明的，所以我建议你对照着文稿看，有些例子可能会“毁三观”，也建议你读完文章后亲手实践一下。

案例一：等值查询间隙锁

第一个例子是关于等值条件操作间隙：

session A	session B	session C
begin; update t set d=d+1 where id=7;		
	insert into t values(8,8,8); (blocked)	
		update t set d=d+1 where id=10; (Query OK)

图1 等值查询的间隙锁

由于表t中没有id=7的记录，所以用我们上面提到的加锁规则判断一下的话：

1. 根据原则1，加锁单位是next-key lock，session A加锁范围就是(5,10];
2. 同时根据优化2，这是一个等值查询(id=7)，而id=10不满足查询条件，next-key lock退化成间隙锁，因此最终加锁的范围是(5,10)。

所以，session B要往这个间隙里面插入id=8的记录会被锁住，但是session C修改id=10这行是可以的。

案例二：非唯一索引等值锁

第二个例子是关于覆盖索引上的锁：

session A	session B	session C
begin; select id from t where c=5 lock in share mode;		
	update t set d=d+1 where id=5; (Query OK)	
		insert into t values(7,7,7); (blocked)

图2 只加在非唯一索引上的锁

看到这个例子，你是不是有一种“该锁的不锁，不该锁的乱锁”的感觉？我们来分析一下吧。

这里session A要给索引c上c=5的这一行加上读锁。

1. 根据原则1，加锁单位是next-key lock，因此会给(0,5]加上next-key lock。
2. 要注意c是普通索引，因此仅访问c=5这一条记录是不能马上停下来的，需要向右遍历，查到c=10才放弃。根据原则2，访问到的都要加锁，因此要给(5,10]加next-key lock。
3. 但是同时这个符合优化2：等值判断，向右遍历，最后一个值不满足c=5这个等值条件，因此退化成间隙锁(5,10)。
4. 根据原则2，只有访问到的对象才会加锁，这个查询使用覆盖索引，并不需要访问主键索引，所以主键索引上没有加任何锁，这就是为什么session B的update语句可以执行完成。

但session C要插入一个(7,7,7)的记录，就会被session A的间隙锁(5,10)锁住。

需要注意，在这个例子中，lock in share mode只锁覆盖索引，但是如果是for update就不一样了。执行 for update时，系统会认为你接下来要更新数据，因此会顺便给主键索引上满足条件的

行加上行锁。

这个例子说明，锁是加在索引上的；同时，它给我们的指导是，如果你要用lock in share mode来给行加读锁避免数据被更新的话，就必须得绕过覆盖索引的优化，在查询字段中加入索引中不存在的字段。比如，将session A的查询语句改成select d from t where c=5 lock in share mode。你可以自己验证一下效果。

案例三：主键索引范围锁

第三个例子是关于范围查询的。

举例之前，你可以先思考一下这个问题：对于我们这个表t，下面这两条查询语句，加锁范围相同吗？

```
mysql> select * from t where id=10 for update;
mysql> select * from t where id>=10 and id<11 for update;
```

你可能会想，id定义为int类型，这两个语句就是等价的吧？其实，它们并不完全等价。

在逻辑上，这两条查语句肯定是等价的，但是它们的加锁规则不太一样。现在，我们就让session A执行第二个查询语句，来看看加锁效果。

session A	session B	session C
begin; select * from t where id>=10 and id<11 for update;		
	insert into t values(8,8,8); (Query OK) insert into t values(13,13,13); (blocked)	
		update t set d=d+1 where id=15; (blocked)

图3 主键索引上范围查询的锁

现在我们就用前面提到的加锁规则，来分析一下session A 会加什么锁呢？

1. 开始执行的时候，要找到第一个id=10的行，因此本该是next-key lock(5,10]。根据优化1，主键id上的等值条件，退化成行锁，只加了id=10这一行的行锁。
2. 范围查找就往后继续找，找到id=15这一行停下来，因此需要加next-key lock(10,15]。

所以，session A这时候锁的范围就是主键索引上，行锁id=10和next-key lock(10,15]。这样，session B和session C的结果你就能理解了。

这里你需要注意一点，首次session A定位查找id=10的行的时候，是当做等值查询来判断的，而向右扫描到id=15的时候，用的是范围查询判断。

案例四：非唯一索引范围锁

接下来，我们再看两个范围查询加锁的例子，你可以对照着案例三来看。

需要注意的是，与案例三不同的是，案例四中查询语句的where部分用的是字段c。

session A	session B	session C
begin; select * from t where c>=10 and c<11 for update;		
	insert into t values(8,8,8); (blocked)	
		update t set d=d+1 where c=15; (blocked)

图4 非唯一索引范围锁

这次session A用字段c来判断，加锁规则跟案例三唯一的不同的是：在第一次用c=10定位记录的时候，索引c上加了(5,10]这个next-key lock后，由于索引c是非唯一索引，没有优化规则，也就是说不会蜕变为行锁，因此最终session A加的锁是，索引c上的(5,10]和(10,15]这两个next-key lock。

所以从结果上来看，session B要插入（8,8,8)的这个insert语句时就被堵住了。

这里需要扫描到c=15才停止扫描，是合理的，因为InnoDB要扫到c=15，才知道不需要继续往后找了。

案例五：唯一索引范围锁bug

前面的四个案例，我们已经用到了加锁规则中的两个原则和两个优化，接下来再看一个关于加锁

规则中bug的案例。

session A	session B	session C
begin; select * from t where id>10 and id<=15 for update;		
	update t set d=d+1 where id=20; (blocked)	
		insert into t values(16,16,16); (blocked)

图5 唯一索引范围锁的bug

session A是一个范围查询，按照原则1的话，应该是索引id上只加(10,15]这个next-key lock，并且因为id是唯一键，所以循环判断到id=15这一行就应该停止了。

但是实现上，InnoDB会往前扫描到第一个不满足条件的行为止，也就是id=20。而且由于这是个范围扫描，因此索引id上的(15,20]这个next-key lock也会被锁上。

所以你看到了，session B要更新id=20这一行，是会被锁住的。同样地，session C要插入id=16的一行，也会被锁住。

照理说，这里锁住id=20这一行的行为，其实是没有必要的。因为扫描到id=15，就可以确定不用往后再找了。但实现上还是这么做了，因此我认为这是个bug。

我也曾找社区的专家讨论过，官方bug系统上也有提到，但是并未被verified。所以，认为这是bug这个事儿，也只能算我的一家之言，如果你有其他见解的话，也欢迎你提出来。

案例六：非唯一索引上存在"等值"的例子

接下来的例子，是为了更好地说明“间隙”这个概念。这里，我给表t插入一条新记录。

```
mysql> insert into t values(30,10,30);
```

新插入的这一行c=10，也就是说现在表里有两个c=10的行。那么，这时候索引c上的间隙是什么状态了呢？你要知道，由于非唯一索引上包含主键的值，所以是不可能存在“相同”的两行的。

索引c

0	5	10	10	15	20	25
0	5	10	30	15	20	25

图6 非唯一索引等值的例子

可以看到，虽然有两个`c=10`，但是它们的主键值`id`是不同的（分别是10和30），因此这两个`c=10`的记录之间，也是有间隙的。

图中我画出了索引`c`上的主键`id`。为了跟间隙锁的开区间形式进行区别，我用`(c=10,id=30)`这样的形式，来表示索引上的一行。

现在，我们来看一下案例六。

这次我们用`delete`语句来验证。注意，`delete`语句加锁的逻辑，其实跟`select ... for update`是类似的，也就是我在文章开始总结的两个“原则”、两个“优化”和一个“bug”。

session A	session B	session C
begin; delete from t where c=10;		
	insert into t values(12,12,12); (blocked)	
		update t set d=d+1 where c=15; (Query OK)

图7 delete 示例

这时，**session A**在遍历的时候，先访问第一个**c=10**的记录。同样地，根据原则1，这里加的是**(c=5,id=5)**到**(c=10,id=10)**这个**next-key lock**。

然后，**session A**向右查找，直到碰到**(c=15,id=15)**这一行，循环才结束。根据优化2，这是一个等值查询，向右查找到了不满足条件的行，所以会退化成**(c=10,id=10)** 到 **(c=15,id=15)**的间隙锁。

也就是说，这个**delete**语句在索引**c**上的加锁范围，就是下图中蓝色区域覆盖的部分。

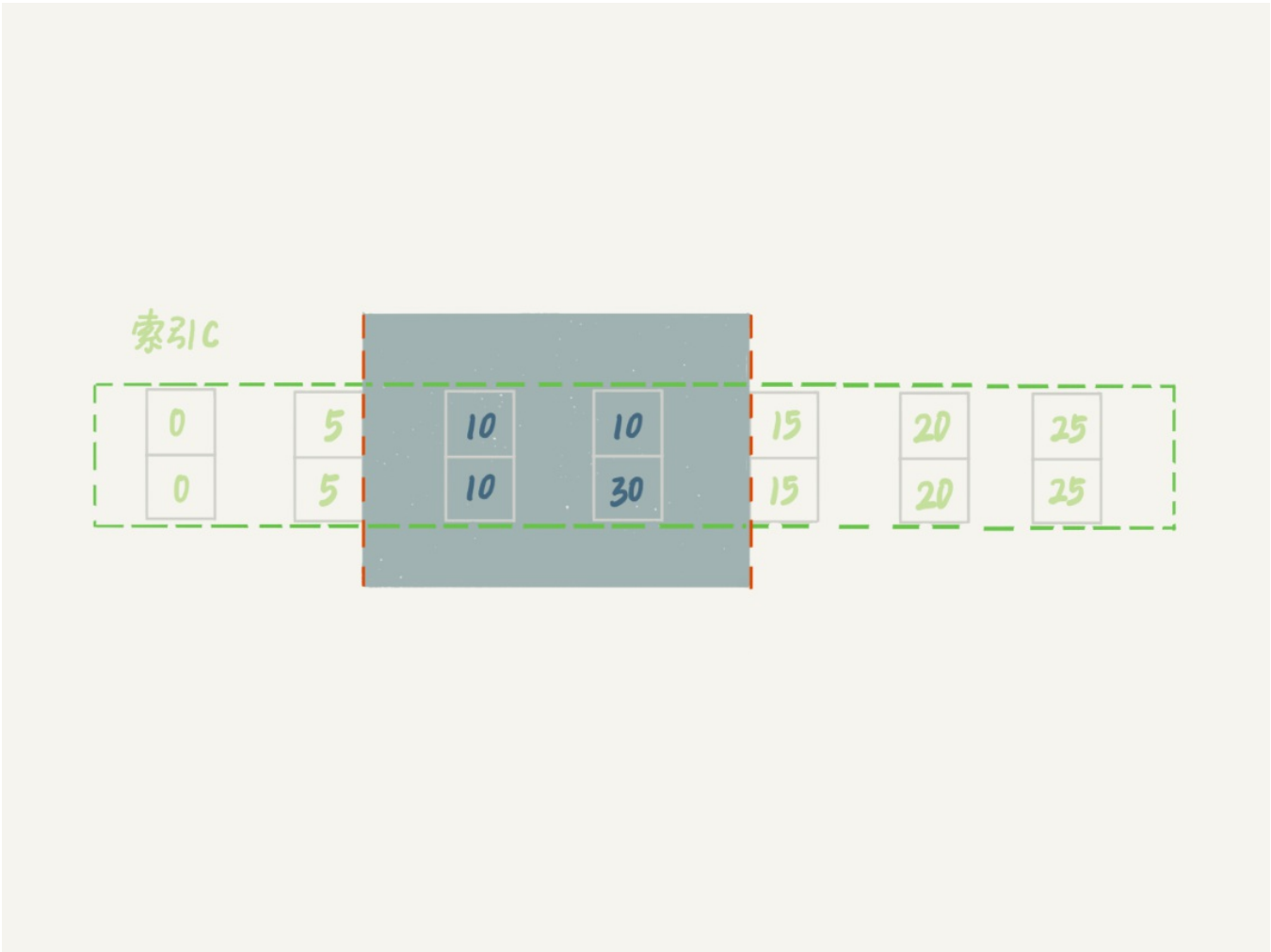


图8 delete加锁效果示例

这个蓝色区域左右两边都是虚线，表示开区间，即(c=5,id=5)和(c=15,id=15)这两行上都没有锁。

案例七：limit 语句加锁

例子6也有一个对照案例，场景如下所示：

session A	session B
begin; delete from t where c=10 limit 2;	
	insert into t values(12,12,12); (Query OK)

图9 limit 语句加锁

这个例子里，session A的delete语句加了 limit 2。你知道表t里c=10的记录其实只有两条，因此加不加limit 2，删除的效果都是一样的，但是加锁的效果却不同。可以看到，session B的insert

语句执行通过了，跟案例六的结果不同。

这是因为，案例七里的`delete`语句明确加了`limit 2`的限制，因此在遍历到`(c=10, id=30)`这一行之后，满足条件的语句已经有两条，循环就结束了。

因此，索引`c`上的加锁范围就变成了从`(c=5, id=5)`到`(c=10, id=30)`这个前开后闭区间，如下图所示：

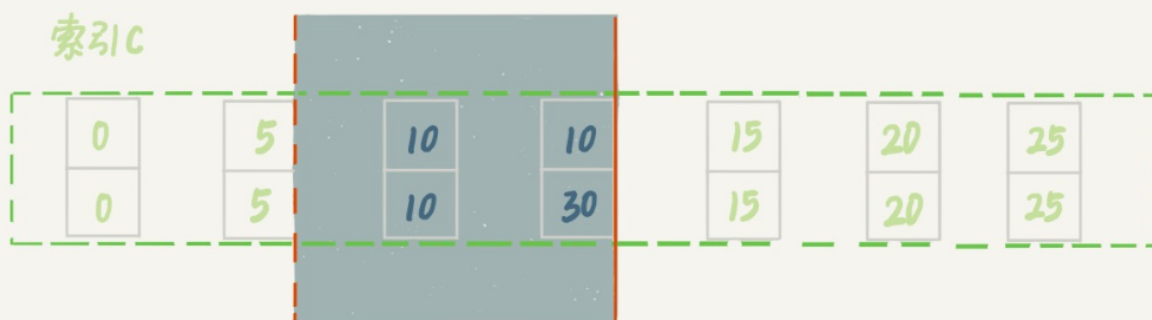


图10 带`limit 2`的加锁效果

可以看到，`(c=10, id=30)`之后的这个间隙并没有在加锁范围里，因此`insert`语句插入`c=12`是可以执行成功的。

这个例子对我们实践的指导意义就是，在删除数据的时候尽量加`limit`。这样不仅可以控制删除数据的条数，让操作更安全，还可以减小加锁的范围。

案例八：一个死锁的例子

前面的例子中，我们在分析的时候，是按照`next-key lock`的逻辑来分析的，因为这样分析比较方便。最后我们再看一个案例，目的是说明：`next-key lock`实际上是间隙锁和行锁加起来的結果。

你一定会疑惑，这个概念不是一开始就说了吗？不要着急，我们先来看下面这个例子：

session A	session B
begin; select id from t where c=10 lock in share mode;	
	update t set d=d+1 where c=10; (blocked)
insert into t values(8,8,8);	
	ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

图11 案例八的操作序列

现在，我们按时间顺序来分析一下为什么是这样的结果。

1. session A 启动事务后执行查询语句加lock in share mode，在索引c上加了next-key lock(5,10] 和间隙锁(10,15);
2. session B 的update语句也要在索引c上加next-key lock(5,10]，进入锁等待；
3. 然后session A要再插入(8,8,8)这一行，被session B的间隙锁锁住。由于出现了死锁，InnoDB让session B回滚。

你可能会问，session B的next-key lock不是还没申请成功吗？

其实是这样的，session B的“加next-key lock(5,10]”操作，实际上分成了两步，先是加(5,10)的间隙锁，加锁成功；然后加c=10的行锁，这时候才被锁住的。

也就是说，我们在分析加锁规则的时候可以用next-key lock来分析。但是要知道，具体执行的时候，是要分成间隙锁和行锁两段来执行的。

小结

这里我再次说明一下，我们上面的所有案例都是在可重复读隔离级别(repeatable-read)下验证的。同时，可重复读隔离级别遵守两阶段锁协议，所有加锁的资源，都是在事务提交或者回滚的时候才释放的。

在最后的案例中，你可以清楚地知道next-key lock实际上是由间隙锁加行锁实现的。如果切换到

读提交隔离级别(read-committed)的话，就好理解了，过程中去掉间隙锁的部分，也就是只剩下行锁的部分。

其实读提交隔离级别在外键场景下还是有间隙锁，相对比较复杂，我们今天先不展开。

另外，在读提交隔离级别下还有一个优化，即：语句执行过程中加上的行锁，在语句执行完成后，就要把“不满足条件的行”上的行锁直接释放了，不需要等到事务提交。

也就是说，读提交隔离级别下，锁的范围更小，锁的时间更短，这也是不少业务都默认使用读提交隔离级别的原因。

不过，我希望你学过今天的课程以后，可以对next-key lock的概念有更清晰的认识，并且会用加锁规则去判断语句的加锁范围。

在业务需要使用可重复读隔离级别的时候，能够更细致地设计操作数据库的语句，解决幻读问题的同时，最大限度地提升系统并行处理事务的能力。

经过这篇文章的介绍，你再看一下上一篇文章最后的思考题，再来尝试分析一次。

我把题目重新描述和简化一下：还是我们在文章开头初始化的表t，里面有6条记录，图12的语句序列中，为什么session B的insert操作，会被锁住呢？

session A	session B
<pre>begin; select * from t where c>=15 and c<=20 order by c desc lock in share mode;</pre>	
	<pre>insert into t values(6,6,6); (blocked)</pre>

图12 锁分析思考题

另外，如果你有兴趣多做一些实验的话，可以设计好语句序列，在执行之前先自己分析一下，然后实际地验证结果是否跟你的分析一致。

对于那些你自己无法解释的结果，可以发到评论区里，后面我争取挑一些有趣的案例在文章中分析。

你可以把你关于思考题的分析写在留言区，也可以分享你自己设计的锁验证方案，我会在下一篇文章的末尾选取有趣的评论跟大家分享。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题，我在本期继续作为了课后思考题，所以会在下篇文章再一起公布“答案”。

这里，我展开回答一下评论区几位同学的问题。

- @令狐少侠 说，以前一直认为间隙锁只在二级索引上有。现在你知道了，有间隙的地方就可能有关锁。
- @浪里白条 同学问，如果是varchar类型，加锁规则是什么样的。
回答：实际上在判断间隙的时候，varchar和int是一样的，排好序以后，相邻两个值之间就有间隙。
- 有几位同学提到说，上一篇文章自己验证的结果跟案例一不同，就是在session A执行完这两个语句：


```
begin;  
select * from t where d=5 for update; /*Q1*/
```

以后，session B 的update 和session C的insert 都会被堵住。这是不是跟文章的结论矛盾？

其实不是的，这个例子用的是反证假设，就是假设不堵住，会出现问题；然后，推导出session A需要锁整个表所有的行和所有间隙。

评论区留言点赞板：

| @某、人、@郭江伟 两位同学尝试分析了上期问题，并给了有启发性的解答。

 极客时间

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。



堕落天使

13

老师，您好。假期的没跟上，今天补到了这节课，看了之后有几点不是太明白。望能解答一下。

1. 索引c上的锁算不算是行锁。假如索引c上的next-key lock为(0,5] (5,10]，那么5算不算是c上的行锁？
2. 在案例六中，执行“delete from t where c=10;”语句，索引c上的next-key lock是(5,10],(10,10],(10,15)。那么主键索引上的锁是什么呢？是只有行锁，锁住的是 (10,10,10) 和 (30,10,30) 两行吗？
3. 也是在案例六中，session A不变，在session B中执行“update t_20 set d=50 where c=5;”、“update t_20 set d=50 where c=15;”、“insert into t_20 values(40,15,40);”均执行成功，但执行“insert into t_20 values(50,5,50);”时，却被阻塞。为什么呢？具体执行语句如下

session A

```
mysql> begin;
```

```
mysql> explain delete from t_20 where c=10;
```

```
id select_type table partitions type possible_keys key key_len ref rows filtered Extra
```

```
1 DELETE t_20 range c c 5 const 2 100 Using where
```

```
mysql> delete from t_20 where c=10;
```

session B

```
mysql> update t_20 set d=50 where c=5;
```

```
Query OK, 1 row affected (0.01 sec)
```

```
Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> update t_20 set d=50 where c=15;
```

```
Query OK, 1 row affected (0.00 sec)
```

```
Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> insert into t_20 values(40,15,40);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> explain insert into t_20 values(50,5,50);
```

```
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
```

```
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | INSERT | t_20 | NULL | ALL | c | NULL | NULL | NULL | NULL | NULL |
```

```
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> insert into t_20 values(50,5,50);
```

```
(block)
```

我使用的mysql版本是：5.7.23-0ubuntu0.16.04.1

show variables的结果太多，我截取了一部分，或许对您分析有帮助：

```
innodb_version 5.7.23
protocol_version 10
slave_type_conversions
tls_version TLSv1,TLSv1.1
version 5.7.23-0ubuntu0.16.04.1
version_comment (Ubuntu)
version_compile_machine x86_64
version_compile_os Linux
```

2019-01-03

作者回复

1. Next-key lock 就是间隙锁+行锁，所以包含=5这一行
2. 对
3. (c=5,id=50)是在这个gap里哦，你试试插入(1,5,50)对比一下。好问题

2019-01-03



张三

👍 63

Happy New Year !这个专栏绝对是极客时间最好我买过最值的专栏。

2018-12-31



undifined

👍 18

遇到一个有趣的问题，在老师的解答下终于弄明白了：

```
CREATE TABLE z (
  id INT PRIMARY KEY AUTO_INCREMENT,
  b INT,
  KEY b(b)
)
ENGINE = InnoDB
DEFAULT CHARSET = utf8;
```

```
INSERT INTO z (id, b)
VALUES (1, 2),
(3, 4),
(5, 6),
(7, 8),
(9, 10);
```

session A

BEGIN;


```
SELECT *  
FROM z  
WHERE b = 6 FOR UPDATE;
```

session B

```
INSERT INTO z VALUES (0, 4);
```

这里为什么会被锁住

答案比较长，写在我自己的笔记里了，地址是 <https://helloworlde.github.io/blog/blog/MySQL/MySQL-%E4%B8%AD%E5%85%B3%E4%BA%8Egap-lock-next-key-lock-%E7%9A%84%E4%B8%80%E4%B8%AA%E9%97%AE%E9%A2%98.html>

大家可以看看

2019-01-07

作者回复

好问题，质量很高的笔记

2019-01-10



约书亚

👍 18

早晨睡不着打开极客时间一看，竟然更新了。今天是周日而且在假期中哎...

2018-12-31

作者回复

风雨无阻 节假日不休，包括元旦和春节👍

2018-12-31



HuaMax

👍 10

首先老师新年快乐，学习专栏受益良多！

上期问过老师的问题已了解答案，锁是加在索引上的。再尝试回答问题。**c**上是普通索引，根据原则2，访问到的都要加锁，在查询**c>=15**这个条件时，在查找到**15**后加锁（**10, 15]**，继续往右查找，按理说不会锁住**6**这个索引值，但查询语句中加了**order by c desc**，我猜想会优化为使用**c<=20**这条语句，查找到**20**后往左查找，这样会访问到**15**左边的值**10**，从而加锁（**5, 10]**，不知我理解对否？

2019-01-01

作者回复

新年好

对的👍

2019-01-01



Leon👍

👍 7

老师，案例八session B的操作语句**update t set d = d + 1 where c = 10**; 由于**c**是非唯一键索引，锁（**5, 10]**可以理解，为什么不锁（**10,15]**呢，不是应该继续向后扫描直到第一个不满足条件的值为止吗

2019-01-29

作者回复

好问题，新年快乐

会锁的，只是因为(5,10]就被锁住了，所以后面的锁加不上去了

2019-02-01



乾坤

7

您好，关于"优化 2: 索引上的等值查询，向右遍历时且最后一个值不满足等值条件的时候，next-key lock 退化为间隙锁。"，我觉得改为"从第一个满足等值条件的索引记录开始向右遍历到第一个不满足等值条件记录，并将第一个不满足等值条件记录上的next-key lock 退化为间隙锁"更明确些

2019-01-01

作者回复

感觉没大差别，嗯嗯，理解就好

2019-01-02



时隐时现

6

不好意思，这次又来晚了，看这种连载技术文章，和看小说一样，养肥了集中看~~
这次的问题如下，希望丁老师有空解答一下。

版本: mysql 5.6.39

```
CREATE TABLE `t` (  
  `a` int(11) NOT NULL,  
  `b` int(11) DEFAULT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
insert into t values(1,1),(2,2),(3,3),(4,4),(5,5);
```

采用READ-COMMITTED隔离级别

案例1、

session A:

```
begin;
```

```
update t set a=6 where b=1;
```

session B:

```
begin;
```

```
update t set a=7 where b=2;
```

A和B均能执行成功

问题1: 官档上说对于RC且全表扫描的update，先逐行添加行锁然后释放掉不符合where条件的，那么session A成功对(1,1)加锁，理论上session B在扫描(1,1)并尝试加锁时会被阻塞，为何还能执行成功？官档链接：<https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html>

案例2:

session A:

```
begin;
```

```
update t set a=6 where b=1;
```

session B:

```
begin;
```

```
delete from t where b=2; -- 被阻塞
```

问题2: 为何案例1 中的session B不会被阻塞, 而案例2的却被session A的行数阻塞, update和delete都是全部扫描, 难道加锁机制不一样?

2019-01-30

作者回复

好问题, 在read-commited隔离级别下, update语句

有一个“semi-consistent” read优化,

意思是, 如果update语句碰到一个已经被锁了的行, 会读入最新的版本, 然后判断一下是不是满足查询条件,

a) 如果不满足, 就直接跳过;

b) 如果满足, 才进入锁等待

你的第二个问题: 这个策略, 只对update有效, delete无效

新春快乐~

2019-02-04



Geek_9ca34e

6

老师, 你好:

我练习实例的时候发现一个问题: 如 案例五: 唯一索引范围锁 bug

```
begin;
```

```
select * from t where id>10 and id<=15 for update;
```

1、执行如上语句加锁范围(10,15]和(15,20];

2、因为10未加锁, 所以我单独再开一个连接, 执行delete from t where id=10;不会锁等待, 能正常删除;

3、但是我再执行insert into t values(10,10,10); 语句会等待, 无法正常执行;

4、经过分析我发现第一个连接执行的语句的加锁范围已经变成(5,15]和(15,20], 代表锁蔓延了; 这是为什么呢?

2019-01-09

作者回复

好问题, 我会加到答疑文章中,

Gap是一个动态的概念

2019-01-09



往事随风, 顺其自然

6

这和分两步有什么关系?

(5,10]已经是被锁住, 分不分两步来加锁, 这个间隙和行锁都被锁住了, session b应该是拿不

到锁才对。

2019-01-01



郭江伟

👍 6

老师这次的留下的问题，语句跟上次不一样，上期问题语句是`select id from t where c>=15 and c<=20 order by c desc for update;`；这次缺少了 `order by c desc`，不加`desc`的话`insert into t values(6,6,6);`不会被堵塞；

根据优化3：索引上的等值查询，在向右遍历时且最后一个值不满足等值条件的时候`next-key lock`退化为间隙锁；

问题中的`sql`语句加了`desc`，是向左扫描，该优化用不上，所以下限10是闭区间，为了防止`c`为10的行加入，需要锁定到索引`c`键（5,5）

此例中`insert into t values(6,5,6)` 会堵塞，`insert into t values(4,5,6)` 不会堵塞，

2018-12-31

作者回复

嗯你说的对

不过是我少打一个词了，加上去了，要`desc`哦

重新分析下👍

2018-12-31



Geek_maxwell

👍 2

老师有个疑问，就是既然`repeat read`已经解决了幻读（利用`next key lock`），那下一个事务级别（序列化）主要是解决什么问题呢？因为原本我以为`repeat read` 其实没有解决幻读

2019-04-14



null

👍 2

@唯她命 的问题，案例一为啥是间隙锁，而不是行锁？

我自己的理解：因为 `id=7` 的记录不存在，没有上锁的实体，所以无法使用优化 1。

优化 1 是说索引上的等值查询，如果是唯一索引，且记录存在，就退化成行锁。

优化 2 是说在索引上的等值查询，不管是唯一索引，还是非唯一索引，只有向右遍历时，最后一个值不等于等值查询的条件，就会退化成间隙锁。如 `where id=7`，扫描完 `id=5`，接着扫描 `id=10`，但是 7 不等于 10，所以会退化成间隙锁。

如果理解有误，还望老师指正👍

2019-04-13



陈

👍 2

老师在案例一中`update t set d=d+1 where id=7` 中`id`是主键也是唯一索引，按优化1应该退化成行锁才对，为什么`insert into t values(8,8,8)`会被锁住，我是那儿理解错了？

2019-01-11

作者回复

这一行存在的时候是行锁，这一行不存在，那就是间隙锁啦。

`insert into t values(8,8,8)`是被主键上`(5,10)`的间隙锁锁住的

2019-01-11



鸿翱

👍 2

老师我今天在回顾的时候又想到一个新问题，为什么RR级别下要以next-key lock作为加锁的基本单位呢？

比如说案例5中的那个例子，我将其修改成`select * from t where id > 10 and id < 15 for update`，按照加锁规则，需要对`(10,15]`上锁，那么按照逻辑来看，15的那个行锁其实没必要的（难道是有必要的嘛？），既然next-key lock本质也是先加间隙锁再加行锁，为什么非要把那个行锁也加上，就是因为next-key lock是加锁基本单位嘛？

2019-01-10

作者回复

代码实现上是这样子的。。

2019-01-11



木子

👍 1

老师，你好，回头看第二遍的时候，突然发现一个地方没明白：在案例1中 `id=7`等值查询时，`id`是主键，也满足唯一索引，为什么没有根据优化1 next-key lock 退化为行锁而是加了`(5,10)`的间隙锁？而案例三中 主键索引范围锁案例中 第一步 找到`id=10`的记录 这时候是根据优化2 主键索引上的等值查询 退化为行锁？同样是主键索引的等值查询为什么一个加了间隙锁一个退化成了行锁？麻烦老师看到之后回复一下，非常感谢

2019-05-14



null

👍 1

@坠落天使 的问题，案例六的 session A 不变，session B `insert (c=5, id=50)` 的记录被 block，可以借用“线段”的概念来帮助理解，下面是我自己的理解：

session A 我们只关注 `a (c=5, id=5)` 和 `b (c=10, id=10)` 这两个点 (为了方便说明，其他的锁我们选择忽略)。

在平面上，`a` 和 `b` 两点连成的线段，就是我们的加锁范围。

现在有一个点 `f(c=5, id=6)` 需要插入线段中，因为 `a` 和 `f` 的 `c` 值都是 5，但是 `f` 的 `id` 比 `a` 的 `id` 要大，所以在线段上表示，`f` 在 `a` 的右边，落入了线段内，即落入了锁区间，所以被 block 了。

有一个点 `e(c=5, id=4)` 需要在平面展示，同理 `e` 点是在 `a` 点的左边，没落到锁区间，所以 `insert` 成功。

2019-04-14



王伯轩

👍 1

这两章今天来来回回啃了好几遍，终于感觉有关锁的知识成体系了。

尤其是总结的加锁原则，简直太精辟了，感谢丁老师。

原则 2：查找过程中访问到的对象才会加锁。

对于原则2 我有个疑问，访问到的对象应该还应该包含其影响到的索引。

老师的例子中某些隐含了这些东西，其他同学的留言中也表明了这点，望老师指点

比如 一个表t (id , c,d,e)

id是主键 ， 其他列都有非唯一索引。

执行insert 需要获取所有索引上的锁；

执行删除（即使根据id删除）也需要获取其他索引的锁；

执行update（即使where条件使用id）如果更新的有索引列，也需要获取上面的锁

因为更新操作肯定会导致索引树的修改，如果不更改，会导致索引搜索时返回的数据和实际不一样；如果更改的话，肯定需要防止并发就需要加锁

不知道理解的是否正确

2019-04-06

作者回复

对的，要update和delete的时候，要“先读后写”，这个读就开始加锁了。

insert的时候要有插入意向锁（就是会跟gap lock冲突的那个）

2019-04-07



唯她命

1

老师，唯一索引的等值查询 退换成 行锁，案例一的等值查询为啥是间隙锁？

2019-03-23



Barnett

1

老师，您好，关于精选留言中堕落天使的第三个问题，next-key lock是(5,10],[10,10],[10,15)，您的回答是(c=5,id=50)也是在这个gap里的。但是5不是开区间的吗？

2019-03-10

作者回复

(5,10]是个简写

其实就是左边界是 (c=5,id=5) , 右边界是(c=10, id=10),

左开右闭

这样你再看下这个结论

2019-03-10