20 | 幻读是什么，幻读有什么问题？

2018-12-28 林晓斌

在上一篇文章最后，我给你留了一个关于加锁规则的问题。今天，我们就从这个问题说起吧。

为了便于说明问题，这一篇文章，我们就先使用一个小一点儿的表。建表和初始化语句如下（为了便于本期的例子说明，我把上篇文章中用到的表结构做了点儿修改）：

```
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);
```

这个表除了主键id外，还有一个索引c，初始化语句在表中插入了6行数据。

上期我留给你的问题是，下面的语句序列，是怎么加锁的，加的锁又是什么时候释放的呢？

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

比较好理解的是，这个语句会命中d=5的这一行，对应的主键id=5，因此在select 语句执行完成后，id=5这一行会加一个写锁，而且由于两阶段锁协议，这个写锁会在执行commit语句的时候释放。

由于字段d上没有索引，因此这条查询语句会做全表扫描。那么，其他被扫描到的，但是不满足条件的5行记录上，会不会被加锁呢？

我们知道，InnoDB的默认事务隔离级别是可重复读，所以本文接下来没有特殊说明的部分，都是设定在可重复读隔离级别下。

幻读是什么？

现在，我们就来分析一下，如果只在id=5这一行加锁，而其他行的不加锁的话，会怎么样。

下面先来看一下这个场景（注意：这是我假设的一个场景）：

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/ result: (5,5,5)		
T2		update t set d=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/ result: (0,0,5),(5,5,5)		
T4			insert into t values(1,1,5);
T5	select * from t where d=5 for update; /*Q3*/ result: (0,0,5),(1,1,5),(5,5,5)		
T6	commit;		

图 1 假设只在id=5这一行加行锁

可以看到，session A里执行了三次查询，分别是Q1、Q2和Q3。它们的SQL语句相同，都是select * from t where d=5 for update。这个语句的意思你应该很清楚了，查所有d=5的行，而且使用的是当前读，并且加上写锁。现在，我们来看一下这三条SQL语句，分别会返回什么结果。

1. Q1只返回id=5这一行；

2. 在T2时刻，session B把id=0这一行的d值改成了5，因此T3时刻Q2查出来的是id=0和id=5这两行；

3. 在T4时刻，session C又插入一行（1,1,5），因此T5时刻Q3查出来的是id=0、id=1和id=5的这三行。

其中，Q3读到id=1这一行的现象，被称为“幻读”。也就是说，幻读指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行。

这里，我需要对“幻读”做一个说明：

1. 在可重复读隔离级别下，普通的查询是快照读，是不会看到别的事务插入的数据的。因此，幻读在“当前读”下才会出现。

2. 上面session B的修改结果，被session A之后的select语句用“当前读”看到，不能称为幻读。幻读仅专指“新插入的行”。

如果只从第8篇文章《事务到底是隔离的还是不隔离的？》我们学到的事务可见性规则来分析的话，上面这三条SQL语句的返回结果都没有问题。

因为这三个查询都是加了for update，都是当前读。而当前读的规则，就是要能读到所有已经提交的记录的最新值。并且，session B和sessionC的两条语句，执行后就会提交，所以Q2和Q3就是应该看到这两个事务的操作效果，而且也看到了，这跟事务的可见性规则并不矛盾。

但是，这是不是真的没问题呢？

不，这里还真就有问题。

幻读有什么问题？

首先是语义上的。session A在T1时刻就声明了，“我要把所有d=5的行锁住，不准别的事务进行读写操作”。而实际上，这个语义被破坏了。

如果现在这样看感觉还不明显的话，我再往session B和session C里面分别加一条SQL语句，你再看看会出现什么现象。

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/		
T2		update t set d=5 where id=0; update t set c=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/		

T4			insert into t
			values(1,1,5);
			update t set c=5
			where id=1;

T5	select * from t where d=5 for update; /*Q3*/		
T6	commit;		

图 2 假设只在id=5这一行加行锁--语义被破坏

session B的第二条语句update t set c=5 where id=0，语义是“我把id=0、d=5这一行的c值，改成了5”。

由于在T1时刻，session A 还只是给id=5这一行加了行锁， 并没有给id=0这行加上锁。因此，session B在T2时刻，是可以执行这两条update语句的。这样，就破坏了 session A 里Q1语句要锁住所有d=5的行的加锁声明。

session C也是一样的道理，对id=1这一行的修改，也是破坏了Q1的加锁声明。

其次，是数据一致性的问题。

我们知道，锁的设计是为了保证数据的一致性。而这个一致性，不止是数据库内部数据状态在此刻的一致性，还包含了数据和日志在逻辑上的一致性。

为了说明这个问题，我给session A在T1时刻再加一个更新语句，即：update t set d=100 where d=5。

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/ update t set d=100 where d=5;		
T2		update t set d=5 where id=0; update t set c=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/		

T4			insert into t
			values(1,1,5);
			update t set c=5
			where id=1;

T5	select * from t where d=5 for update; /*Q3*/		
T6	commit;		

图 3 假设只在id=5这一行加行锁--数据一致性问题

update的加锁语义和select ..for update 是一致的，所以这时候加上这条update语句也很合理。session A声明说“要给d=5的语句加上锁”，就是为了要更新数据，新加的这条update语句就是把它认为加上了锁的这一行的d值修改成了100。

现在，我们来分析一下图3执行完成后，数据库里会是什么结果。

1. 经过T1时刻，id=5这一行变成 (5,5,100)，当然这个结果最终是在T6时刻正式提交的;
2. 经过T2时刻，id=0这一行变成(0,5,5);
3. 经过T4时刻，表里面多了一行(1,5,5);
4. 其他行跟这个执行序列无关，保持不变。

这样看，这些数据也没啥问题，但是我们再来看看这时候binlog里面的内容。

1. T2时刻，session B事务提交，写入了两条语句；
2. T4时刻，session C事务提交，写入了两条语句；
3. T6时刻，session A事务提交，写入了update t set d=100 where d=5 这条语句。

我统一放到一起的话，就是这样的：

```
update t set d=5 where id=0; /*(0,0,5)*/
update t set c=5 where id=0; /*(0,5,5)*/

insert into t values(1,1,5); /*(1,1,5)*/
update t set c=5 where id=1; /*(1,5,5)*/

update t set d=100 where d=5;/*所有d=5的行，d改成100*/
```

好，你应该看出问题了。这个语句序列，不论是拿到备库去执行，还是以后用binlog来克隆一个库，这三行的结果，都变成了 (0,5,100)、(1,5,100)和(5,5,100)。

也就是说，id=0和id=1这两行，发生了数据不一致。这个问题很严重，是不行的。

到这里，我们再回顾一下，**这个数据不一致到底是怎么引入的？**

我们分析一下可以知道，这是我们假设“select * from t where d=5 for update这条语句只给d=5这一行，也就是id=5的这一行加锁”导致的。

所以我们认为，上面的设定不合理，要改。

那怎么改呢？我们把扫描过程中碰到的行，也都加上写锁，再来看看执行效果。

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/ update t set d=100 where d=5;		
T2		update t set d=5 where id=0; (blocked) update t set c=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/		
T4			insert into t values(1,1,5); update t set c=5 where id=1;
T5	select * from t where d=5 for update; /*Q3*/		
T6	commit;		

图 4 假设扫描到的行都被加上了行锁

由于session A把所有的行都加了写锁，所以session B在执行第一个update语句的时候就被锁住了。需要等到T6时刻session A提交以后，session B才能继续执行。

这样对于id=0这一行，在数据库里的最终结果还是 (0,5,5)。在binlog里面，执行序列是这样的：

```
insert into t values(1,1,5); /*(1,1,5)*/
update t set c=5 where id=1; /*(1,5,5)*/

update t set d=100 where d=5;/*所有d=5的行，d改成100*/

update t set d=5 where id=0; /*(0,0,5)*/
update t set c=5 where id=0; /*(0,5,5)*/
```

可以看到，按照日志顺序执行，id=0这一行的最终结果也是(0,5,5)。所以，id=0这一行的问题解决了。

但同时你也可以看到，id=1这一行，在数据库里面的结果是(1,5,5)，而根据binlog的执行结果是(1,5,100)，也就是说幻读的问题还是没有解决。为什么我们已经这么“凶残”地，把所有的记录都上了锁，还是阻止不了id=1这一行的插入和更新呢？

原因很简单。在T3时刻，我们给所有行加锁的时候，id=1这一行还不存在，不存在也就加不上锁。

也就是说，即使把所有的记录都加上锁，还是阻止不了新插入的记录，这也是为什么“幻读”会被单独拿出来解决的原因。

到这里，其实我们刚说明完文章的标题 ：幻读的定义和幻读有什么问题。

接下来，我们再看看InnoDB怎么解决幻读的问题。

如何解决幻读？

现在你知道了，产生幻读的原因是，行锁只能锁住行，但是新插入记录这个动作，要更新的是记录之间的“间隙”。因此，为了解决幻读问题，InnoDB只好引入新的锁，也就是间隙锁(Gap Lock)。

顾名思义，间隙锁，锁的就是两个值之间的空隙。比如文章开头的表t，初始化插入了6个记录，这就产生了7个间隙。

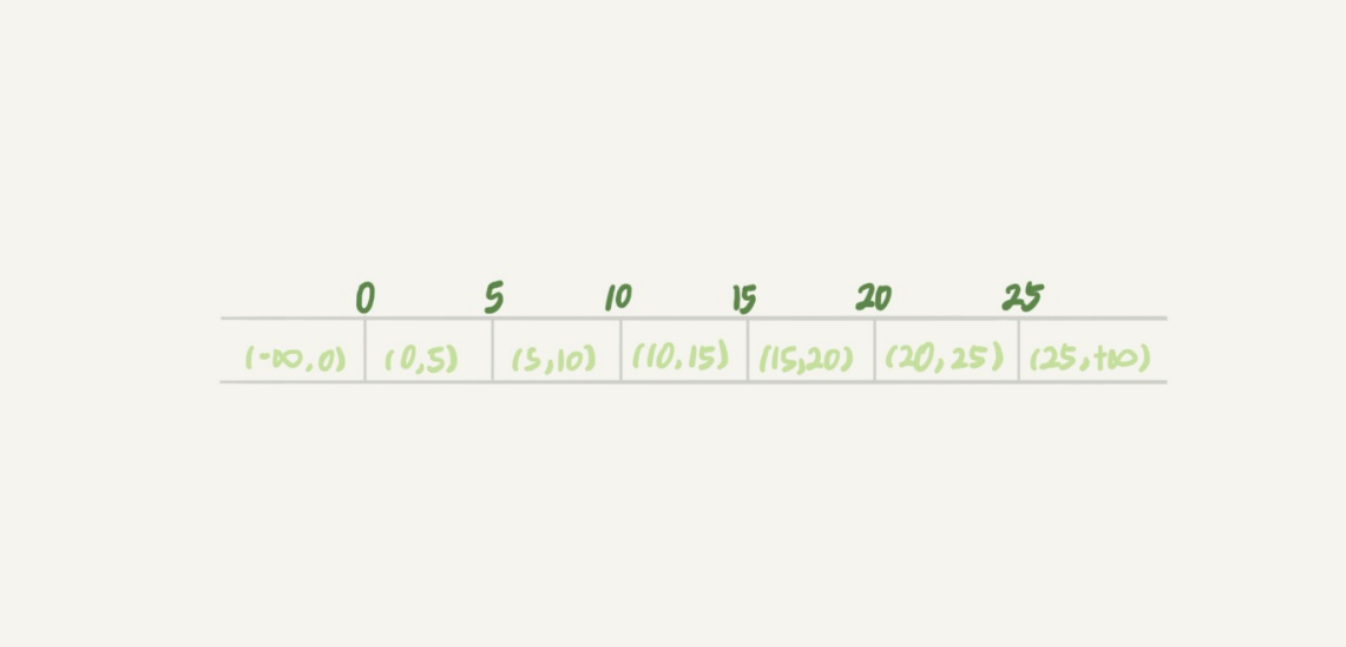

图 5 表t主键索引上的行锁和间隙锁

这样，当你执行 select * from t where d=5 for update的时候，就不止是给数据库中已有的6个记录加上了行锁，还同时加了7个间隙锁。这样就确保了无法再插入新的记录。

也就是说这时候，在一行行扫描的过程中，不仅将给行加上了行锁，还给行两边的空隙，也加上了间隙锁。

现在你知道了，数据行是可以加上锁的实体，数据行之间的间隙，也是可以加上锁的实体。但是间隙锁跟我们之前碰到过的锁都不太一样。

比如行锁，分成读锁和写锁。下图就是这两种类型行锁的冲突关系。

	读锁	写锁
读锁	兼容	冲突
写锁	冲突	冲突

图6 两种行锁间的冲突关系

也就是说，跟行锁有冲突关系的是“另外一个行锁”。

但是间隙锁不一样，**跟间隙锁存在冲突关系的，是“往这个间隙中插入一个记录”这个操作。**间隙锁之间都不存在冲突关系。

这句话不太好理解，我给你举个例子：

session A	session B
begin; select * from t where c=7 lock in share mode;	
	begin; select * from t where c=7 for update;

图7 间隙锁之间不互锁

这里session B并不会被堵住。因为表t里并没有c=7这个记录，因此session A加的是间隙锁(5,10)。而session B也是在这个间隙加的间隙锁。它们有共同的目标，即：保护这个间隙，不允许插入值。但，它们之间是不冲突的。

间隙锁和行锁合称next-key lock，每个next-key lock是前开后闭区间。也就是说，我们的表t初始化以后，如果用select * from t for update要把整个表所有记录锁起来，就形成了7个next-key lock，分别是 (-∞,0]、(0,5]、(5,10]、(10,15]、(15,20]、(20, 25]、(25, +supremum]。

> 备注：这篇文章中，如果没有特别说明，我们把间隙锁记为开区间，把next-key lock记为前开后闭区间。

你可能会问说，这个supremum从哪儿来的呢？

这是因为+∞是开区间。实现上，InnoDB给每个索引加了一个不存在的最大值supremum，这样才符合我们前面说的“都是前开后闭区间”。

间隙锁和next-key lock的引入，帮我们解决了幻读的问题，但同时也带来了一些“困扰”。

在前面的文章中，就有同学提到了这个问题。我把他的问题转述一下，对应到我们这个例子的表来说，业务逻辑这样的：任意锁住一行，如果这一行不存在的话就插入，如果存在这一行就更新它的数据，代码如下：

```
begin;
select * from t where id=N for update;

/*如果行不存在*/
insert into t values(N,N,N);
/*如果行存在*/
update t set d=N set id=N;

commit;
```

可能你会说，这个不是insert ...on duplicate key update 就能解决吗？但其实在有多个唯一键的时候，这个方法是不能满足这位提问同学的需求的。至于为什么，我会在后面的文章中再展开说明。

现在，我们就只讨论这个逻辑。

这个同学碰到的现象是，这个逻辑一旦有并发，就会碰到死锁。你一定也觉得奇怪，这个逻辑每次操作前用for update锁起来，已经是最严格的模式了，怎么还会有死锁呢？

这里，我用两个session来模拟并发，并假设N=9。

session A	session B
begin; select * from t where id=9 for update;	
	begin; select * from t where id=9 for update;
	insert into t values(9,9,9); (blocked)
insert into t values(9,9,9); (ERROR 1213 (40001): Deadlock found)	

图8 间隙锁导致的死锁

你看到了，其实都不需要用到后面的update语句，就已经形成死锁了。我们按语句执行顺序来分析一下：

1. session A 执行select ...for update语句，由于id=9这一行并不存在，因此会加上间隙锁(5,10);
2. session B 执行select ...for update语句，同样会加上间隙锁(5,10)，间隙锁之间不会冲突，因此这个语句可以执行成功；
3. session B 试图插入一行(9,9,9)，被session A的间隙锁挡住了，只好进入等待；
4. session A试图插入一行(9,9,9)，被session B的间隙锁挡住了。

至此，两个session进入互相等待状态，形成死锁。当然，InnoDB的死锁检测马上就发现了这对死锁关系，让session A的insert语句报错返回了。

你现在知道了，**间隙锁的引入，可能会导致同样的语句锁住更大的范围，这其实是影响了并发度的**。其实，这还只是一个简单的例子，在下一篇文章中我们还会碰到更多、更复杂的例子。

你可能会说，为了解决幻读的问题，我们引入了这么一大串内容，有没有更简单一点的处理方法呢。

我在文章一开始就说过，如果没有特别说明，今天和你分析的问题都是在可重复读隔离级别下的，间隙锁是在可重复读隔离级别下才会生效的。所以，你如果把隔离级别设置为读提交的话，就没有间隙锁了。但同时，你要解决可能出现的数据和日志不一致问题，需要把binlog格式设置为row。这，也是现在不少公司使用的配置组合。

前面文章的评论区有同学留言说，他们公司就使用的是读提交隔离级别加binlog_format=row的组合。他曾问他们公司的DBA说，你为什么要这么配置。DBA直接答复说，因为大家都这么用呀。

所以，这个同学在评论区就问说，这个配置到底合不合理。

关于这个问题本身的答案是，如果读提交隔离级别够用，也就是说，业务不需要可重复读的保证，这样考虑到读提交下操作数据的锁范围更小（没有间隙锁），这个选择是合理的。

但其实我想说的是，配置是否合理，跟业务场景有关，需要具体问题具体分析。

但是，如果DBA认为之所以这么用的原因是“大家都这么用”，那就有问题了，或者说，迟早会出问题。

比如说，大家都用读提交，可是逻辑备份的时候，mysqldump为什么要把备份线程设置成可重复读呢？（这个我在前面的文章中已经解释过了，你可以再回顾下第6篇文章《全局锁和表锁：给表加个字段怎么有这么多阻碍？》的内容）

然后，在备份期间，备份线程用的是可重复读，而业务线程用的是读提交。同时存在两种事务隔离级别，会不会有问题？

进一步地，这两个不同的隔离级别现象有什么不一样的，关于我们的业务，“用读提交就够了”这个结论是怎么得到的？

如果业务开发和运维团队这些问题都没有弄清楚，那么“没问题”这个结论，本身就是有问题的。

小结

今天我们从上一篇文章的课后问题说起，提到了全表扫描的加锁方式。我们发现即使给所有的行都加上行锁，仍然无法解决幻读问题，因此引入了间隙锁的概念。

我碰到过很多对数据库有一定了解的业务开发人员，他们在设计数据表结构和业务SQL语句的时候，对行锁有很准确的认识，但却很少考虑到间隙锁。最后的结果，就是生产库上会经常出现由于间隙锁导致的死锁现象。

行锁确实比较直观，判断规则也相对简单，间隙锁的引入会影响系统的并发度，也增加了锁分析的复杂度，但也有章可循。下一篇文章，我就会为你讲解InnoDB的加锁规则，帮你理顺这其中的“章法”。

作为对下一篇文章的预习，我给你留下一个思考题。

session A	session B	session C
begin; select * from t where c>=15 and c<=20 order by c desc for update;		
	insert into t values(11,11,11);	
		insert into t values(6,6,6);

图9 事务进入锁等待状态

如果你之前没有了解过本篇文章的相关内容，一定觉得这三个语句简直是风马牛不相及。但实际上，这里session B和session C的insert 语句都会进入锁等待状态。

你可以试着分析一下，出现这种情况的原因是什么？

这里需要说明的是，这其实是我在下一篇文章介绍加锁规则后才能回答的问题，是留给你作为预习的，其中session C被锁住这个分析是有点难度的。如果你没有分析出来，也不要气馁，我会在下一篇文章和你详细说明。

你也可以说说，你的线上MySQL配置的是什么隔离级别，为什么会这么配置？你有没有碰到什么场景，是必须使用可重复读隔离级别的呢？

你可以把你的碰到的场景和分析写在留言区里，我会在下一篇文章选取有趣的评论跟大家一起分享和分析。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

我们在本文的开头回答了上期问题。有同学的回答中还说明了读提交隔离级别下，在语句执行完成后，是只有行锁的。而且语句执行完成后，InnoDB就会把不满足条件的行行锁去掉。

当然了，c=5这一行的行锁，还是会等到commit的时候才释放的。

评论区留言点赞板：

> @薛畅 、@张永志同学给出了正确答案。而且提到了在读提交隔离级别下，是只有行锁的。@帆帆帆帆帆帆帆帆、@欧阳成 对上期的例子做了验证，需要说明一下，需要在启动配置里

面增加performance_schema=on，才能用上这个功能，performance_schema库里的表才有数据。

精选留言

令狐少侠 15

老师，今天的文章对我影响很大，发现之前掌握的知识有些错误的地方，课后我用你的表结构根据以前不清楚的地方实践了一遍，现在有两个问题，麻烦您解答下

1.我在事务1中执行 begin;select * from t where c=5 for update;事务未提交，然后事务2中begin;update t set c=5 where id=0;执行阻塞，替换成update t set c=11 where id=0;执行不阻塞，我觉得原因是事务1执行时产生next-key lock范围是(0,5].(5,10]。我想问下update set操作c=xxx是会加锁吗？以及加锁的原理。

2.一直以为gap只会在二级索引上，看了你的死锁案例，发现主键索引上也会有gap锁？

2018-12-28

作者回复

1. 好问题。你可以理解为要在索引c上插入一个(c=5,id=0)这一行，是落在(0,5],(5,10]里面的，11可以对吧

2. 嗯，主键索引的间隙上也要有Gap lock保护的

2018-12-28

xuery 3

老师之前的留言说错了，重新梳理下：

图8：间隙锁导致的死锁；我把innodb_locks_unsafe_for_binlog设置为1之后，session B并不会blocked，session A insert会阻塞住，但是不会提示死锁；然后session B提交执行成功，session A提示主键冲突

这个是因为将innodb_locks_unsafe_for_binlog设置为1之后，什么原因造成的？

2019-01-28

作者回复

对， innodb_locks_unsafe_for_binlog 这个参数就是这个意思 “不加gap lock”，

这个已经要被废弃了（8.0就没有了），所以不建议设置哈，容易造成误会。

如果真的要去掉gap lock，可以考虑改用RC隔离级别+binlog_format=row

2019-02-01

AI杜嘉嘉 33

说真的，这一系列文章实用性真的很强，老师非常负责，想必牵扯到老师大量精力，希望老师再出好文章，谢谢您了，辛苦了

2018-12-28

作者回复

精力花了没事，睡一觉醒来还是一条好汉

主要还是得大家有收获，我就值了

2018-12-28

薛畅 26

可重复读隔离级别下，经试验：
SELECT * FROM t where c>=15 and c<=20 for update; 会加如下锁：
next-key lock:(10, 15], (15, 20]
gap lock:(20, 25)

SELECT * FROM t where c>=15 and c<=20 order by c desc for update; 会加如下锁：
next-key lock:(5, 10], (10, 15], (15, 20]
gap lock:(20, 25)

session C 被锁住的原因就是根据索引 c 逆序排序后多出的 next-key lock:(5, 10]

同时我有个疑问：加不加 next-key lock:(5, 10] 好像都不会影响到 session A 可重复读的语义，那么为什么要加这个锁呢？

2018-12-29

作者回复

是的，这个其实就是为啥总结规则有点麻烦，有时候只是因为代码是这么写的

2018-12-29

沉浮

16

通过打印锁日志帮助理解问题
锁信息见括号里的说明。

TABLE LOCK table `guo_test`.`t` trx id 105275 lock mode IX
RECORD LOCKS space id 31 page no 4 n bits 80 index c of table `guo_test`.`t` trx id 105275 lock_mode X
Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0 ----(Next-Key Lock，索引锁c（5，10])
0: len 4; hex 8000000a; asc ;;
1: len 4; hex 8000000a; asc ;;

Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 0 ----(Next-Key Lock，索引锁c (10,15])
0: len 4; hex 8000000f; asc ;;
1: len 4; hex 8000000f; asc ;;

Record lock, heap no 6 PHYSICAL RECORD: n_fields 2; compact format; info bits 0 ----(Next-Key Lock，索引锁c (15,20])
0: len 4; hex 80000014; asc ;;
1: len 4; hex 80000014; asc ;;

Record lock, heap no 7 PHYSICAL RECORD: n_fields 2; compact format; info bits 0 ----(Next-Key Lock，索引锁c (20,25])
0: len 4; hex 80000019; asc ;;
1: len 4; hex 80000019; asc ;;

RECORD LOCKS space id 31 page no 3 n bits 80 index PRIMARY of table `guo_test`.`t` trx id 105275 lock_mode X locks rec but not gap
Record lock, heap no 5 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
----(记录锁 锁c=15对应的主键）
0: len 4; hex 8000000f; asc ;;
1: len 6; hex 0000000199e3; asc ;;
2: len 7; hex ca000001470134; asc G 4;;
3: len 4; hex 8000000f; asc ;;
4: len 4; hex 8000000f; asc ;;

Record lock, heap no 6 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
0: len 4; hex 80000014; asc ;;
----(记录锁 锁c=20对应的主键）
1: len 6; hex 0000000199e3; asc ;;
2: len 7; hex ca000001470140; asc G @;;
3: len 4; hex 80000014; asc ;;

4: len 4; hex 80000014; asc ;;
由于字数限制，正序及无排序的日志无法帖出，倒序日志比这两者，多了范围(Next-Key Lock，索引锁c（5，10])，个人理解是，加锁分两次，第一次，即正序的锁，第二次为倒序的锁，即多出的(5,10],在RR隔离级别，
innodb在加锁的过程中会默认向后锁一个记录，加上Next-Key Lock,第一次加锁的时候10已经在范围，由于倒序，向后，即向5再加Next-key Lock,即多出的(5,10]范围

2018-12-28

作者回复

优秀

2018-12-28

郭江伟

14

insert into t values(0,0,0),(5,5,5),
(10,10,10),(15,15,15),(20,20,20),(25,25,25);
运行mysql> begin;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from t where c>=15 and c<=20 order by c desc for update;
c 索引会在最右侧包含主键值，c索引的值为(0,0) (5,5) (10,10) (15,15) (20,20) (25,25)
此时c索引上锁的范围其实还要匹配主键值 。
思考题答案是，上限会扫到c索引(20,20) 上一个键，为了防止c为20 主键值小于25 的行插入，需要锁定(20,20) (25,25) 两者的间隙；开启另一会话(26,25,25)可以插入，而(24,25,25)会被堵塞。
下限会扫描到(15,15)的下一个键也就是(10,10),测试语句会继续扫描一个键就是(5,5)，此时会锁定，(5,5) 到(15,15)的间隙，由于id是主键不可重复所以下限也是闭区间；
在本例的测试数据中添加(21,25,25)后就可以正常插入(24,25,25)

2018-12-28

作者回复

感觉你下一篇看起来会很轻松了哈

2018-12-28

慧鑫coming

8

这篇需要多读几遍，again

2018-12-28

en

5

老师您好，我mysql的隔离级别是可重复读，数据是(0,0,0),(5,5,5),(10,10,10),(15,15,15),(20,20,20),(25,25,25)，使用了begin;select * from t where c>=15 and c<=20 order by c desc for update;然后sessionB的11阻塞了，但是(6,6,6)的插入成功了这是什么原因呢？

2018-12-31

简海青

4

加锁过程的分析，这篇文章也是很棒的；供同学们参考
http://hedengcheng.com/?p=771

2019-05-04

kabuka

4

这样，当你执行 select * from t where d=5 for update 的时候，就不止是给数据库中已有的 6 个记录加上了行锁，还同时加了 还同时加了 7 个间隙锁

老師這句話沒看太明白，數據庫只有一條d=5的記錄，為什麼會給6個記錄加上行鎖呢？

2019-03-11

作者回复

因为d上没有索引，这个语句要走全表扫描

2019-03-23

郭健

4

老师，想请教您几个问题。1.在第六章MDL锁的时候，您说给大表增加字段和增加索引的时候要小心，之前做过测试，给一个一千万的数据增加索引有时需要40分钟，但是增加索引不会对表增加MDL锁吧。除了增加索引慢，还会对数据库有什么影响吗，我问我们dba，他说就开始和结束的时候上一下锁，没什么影响，我个人是持怀疑态度的。2，老师讲到表锁除了MDL锁，还有显示命令lock table的命令的表锁，老师我可以认为，在mysql中如果不显示使用lock table表锁的话，那么mysql是永远不会使用表锁的，如果锁的条件没有索引，使用的是锁住行锁+间隙控制并发。

2018-12-30

作者回复

1. 在锁方面你们dba说的基本是对的。一开始和结束有写锁，执行中间40分钟只有读锁
但是1000万的表要做40分钟，可能意味着系统压力大（或者配置偏小），这样可能不是没影响对，比较这个操作还是要吃IO和CPU的

2. 嗯，innodb引擎是这样的。

2018-12-30

滔滔

3

老师，听了您的课收获满满～～感谢您的付出！您可不可以在分析死锁的时候讲一下如何分析死锁日志，期待～～

2018-12-29

作者回复

谢谢你的肯定。

嗯死锁分析会有一篇专门说。

不过你可以提前说一下碰到的疑问

2018-12-29

yan华建

2

什么是幻读?
幻读指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行。（幻读在当前读下才会出现；幻读仅专指新插入的行）
如何解决幻读?
间隙锁（Gap lock）:（两个值之间的锁）。
间隙锁和行锁合称 next-key lock，每个 next-key lock 是前开后闭区间。
间隙锁为开区间。
next-key-lock为前开后闭区间。
间隙锁引入什么问题？
可能会导致同样的语句锁住更大的范围，这其实是影响了并发度的。
间隙锁在RR级别下才有效，RC级别下无间隙锁。
不使用间隙锁方法:
使用读提交隔离级别+ binlog_format=row组合。

2019-06-16

Cv

2

gap锁是否只会在可重复读的情况下才有?
在提交读和有唯一索引的情况下, 我也有遇到过因为gap死锁的情况
大致是这种sql
session1
delete from t where id in (1,3,5);
insert into t id(1,3,5);
session2
delete from t where id in (2,4,6);
insert into t id(2,4,6);

2019-03-07

作者回复

读提交隔离级别一般没有gap lock，不过也有例外情况， 比如insert 出现主键冲突的时候，也可能加间隙锁

2019-03-09

南友力max先森

2

丁老师，想问下，innodb的行锁是怎么实现的，有单独的数据结构存放哪些数据块记录是被锁的么？还是在聚簇索引上对该行数据进行锁定标记？或者是其他？

2019-02-27

作者回复

看下08篇哦，
里面有介绍到行锁
还有问题再在那个文章下面发哈

2019-02-27

杰哥长得帅

2

想问一下老师哪一张会讲意向锁。后面会不会对mysql所有的锁种类做一个总结

2019-01-16

作者回复

由于有metadata lock，意向锁其实没什么作用了，所以不会专门介绍哦，可能会在讨论其他问题的时候顺便带一下

2019-01-16

Geek_89bbab

2

表结构

CREATE TABLE `t2` (

`id` int(11) DEFAULT NULL,

`v` int(11) DEFAULT NULL

) ENGINE=InnoDB;

两个session，

session1, | session2

step1： set session transaction isolation level repeatable read;(session1) | set session transaction isolation level repeatable read;(session2)

step2： begin;(session1)

step3： begin; (session2)

step4： insert into t2 (id,v) values(1,1); （session1)

step5： insert into t2 (id,v) select 2,2 from dual where not exists(select * from t2 where id=2); (session2) // 这里为什么会阻塞，直到session1提交呢？

step6： commit; (session1) 该句执行完 session2不再阻塞

step7：commit;(session2)

我的疑惑就是为什么step5 那一步会阻塞？select * from t2 where id=2 不是快照读吗？也没有用for update，share lock 之类的语句，而且insert into 也没有什么唯一键约束，主键约束，怎么用数据库锁和隔离级别的知识来解释这个现象呢？请老师指点

2019-01-07

作者回复

好问题

Insert...select 是会给select部分加读锁的

这个也是为了保证一致性

2019-01-07

信信

2

老师你好，如果图1的字段d有索引，按前面说的T1时刻后，只有id等于5这一行加了写锁。那么session B 操作的是id等于0这一行，应该不会被阻断吧？如果没阻断的话，仍然会产生语义问题及数据不一致的情况啊。想不明白。。。

2018-12-29

作者回复

如果d有索引，而且写法是d=5，那么其他语句要把其他行的d改成5，也是不行的哦

2018-12-29

某、人

2

按照我的理解select * from t where c>=15 and c<=20 order by c desc for update;
这条语句的加锁顺序的以及范围应该是[25,20),[20,15],(15,10],但是通过实验得出来多了(10,5)gap锁
而且不管是用二级索引还是用主键索引,都会加这段gap锁.
有点不太清楚为什么倒序扫描就需要加上了这段gap锁,目的又是为了什么?
不会气磊,期待老师下一期的答案。

2018-12-28

作者回复

嗯嗯下周一见

2018-12-28

往事随风，顺其自然

2

总结：for update 是锁住所有行还有间隙锁，但是间隙之间互不冲突，但是互不冲突，为什么插入9这一行会被间隙锁等待，原来没有这一行，这和查询9这一行不是一样？

2018-12-28