# 幻读和间隙锁

### 一．1个例子说明什么是幻读

CREATE TABLE `t` (

`id` int(11) NOT NULL,

`c` int(11) DEFAULT NULL,

`d` int(11) DEFAULT NULL,

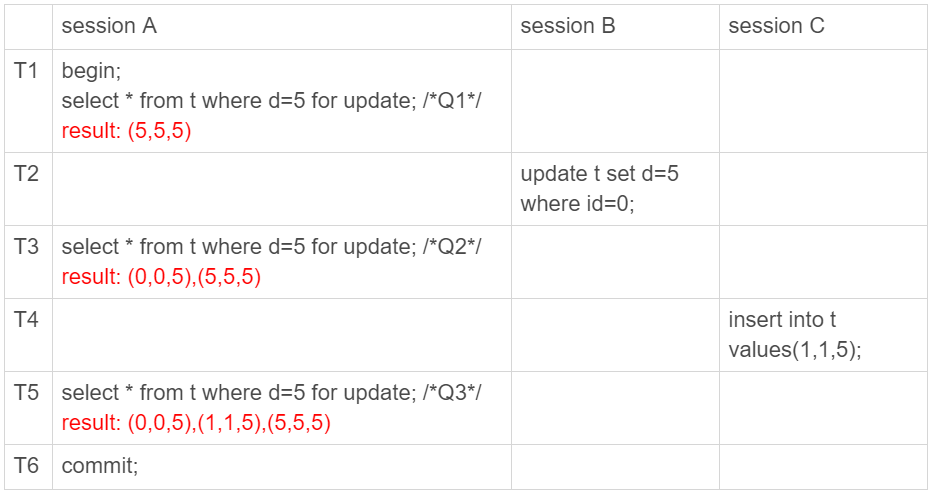
PRIMARY KEY (`id`),

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);



假设默认为RR（可重复读）隔离级别。

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。

Q3 读到 id=1 这一行的现象，被称为“幻读”。

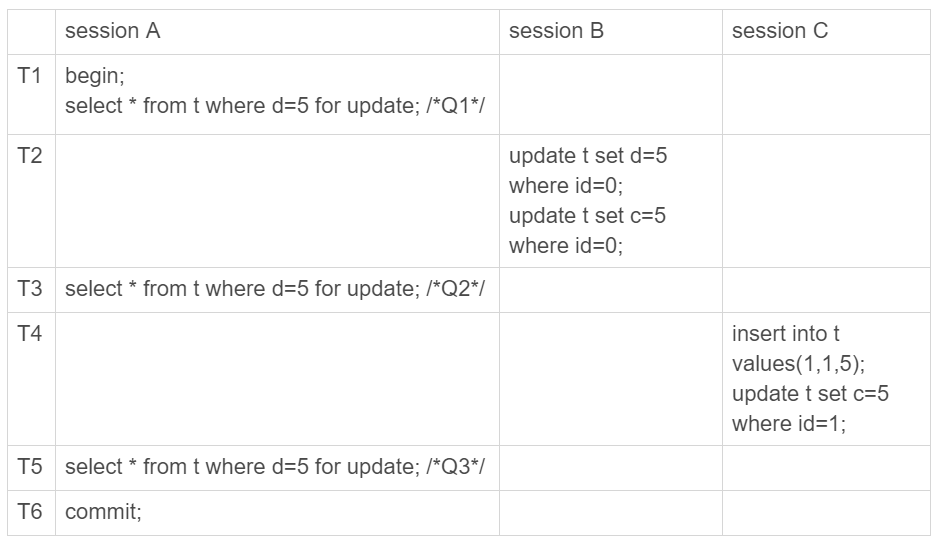
也就是说，幻读指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行。

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；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”。

因为这三个查询都是加了 for update排他锁，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。

### 幻读存在的问题

1. **破坏语义**

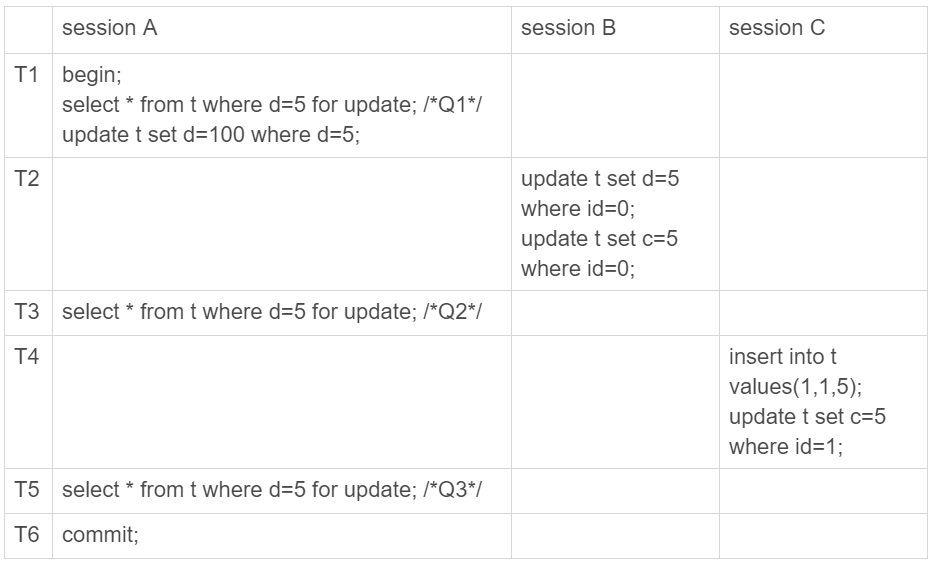


由于在 T1 时刻，session A 还只是给 id=5 这一行加了行锁， 并没有给 id=0 这行加上锁。因此，session B 在 T2 时刻，是可以执行这两条 update 语句的。这样，就破坏了 session A 里 Q1 语句要锁住所有 d=5 的行的加锁声明。

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。

1. **破坏数据和日志逻辑上的一致性**

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。



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 ;经过 T2 时刻，id=0 这一行变成 (0,5,5);经过 T4 时刻，表里面多了一行 (1,5,5);其他行跟这个执行序列无关，保持不变。

但是我们再来看看这时候 binlog 里面的内容：T2 时刻，session B 事务提交，写入了两条语句；T4 时刻，session C 事务提交，写入了两条语句；T6 时刻，session A 事务提交，写入了 update t set d=100 where d=5 这条语句。

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)：

// T2时刻

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

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

// T4时刻

insert into t values(1,1,5); /\*(1,1,5)\*/

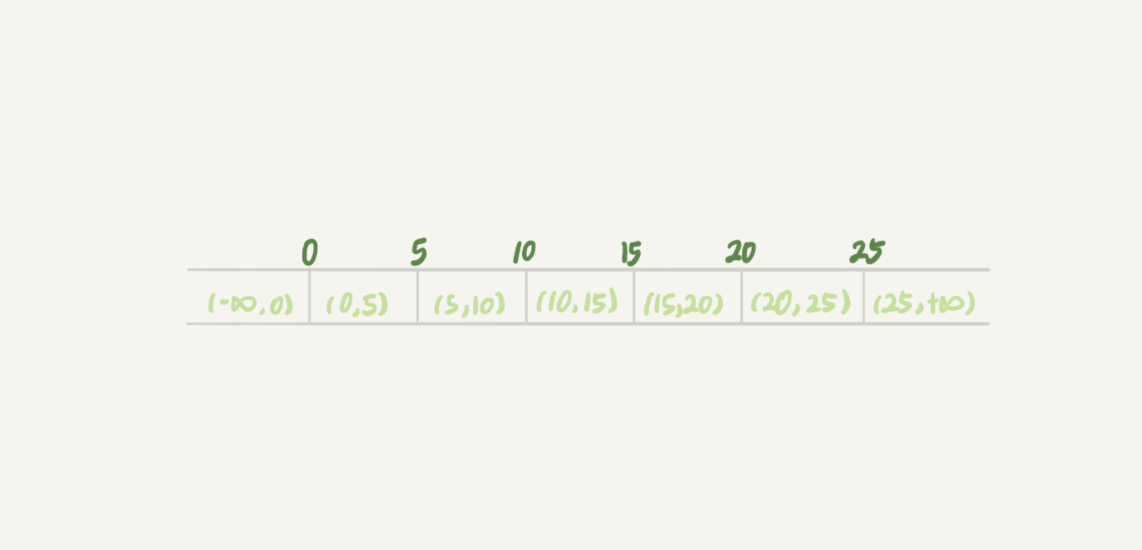
update t set c=5 where id=1; /\*(1,5,5)\*/

// T6时刻

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

1. 如何解决幻读

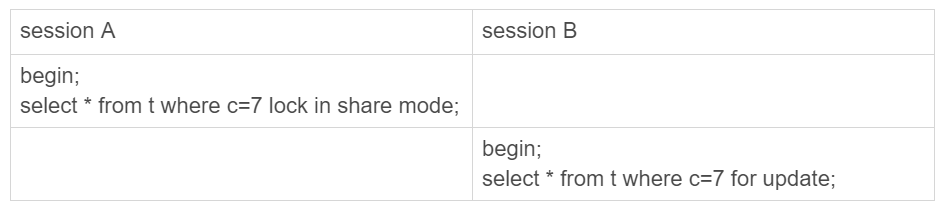
现在你知道了，产生幻读的原因是，行锁只能锁住行，但是新插入记录这个动作，要更新的是记录之间的“间隙”。因此，为了解决幻读问题，InnoDB 只好引入新的锁，也就是间隙锁 (Gap Lock)。



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。

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。

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。

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。

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”。

begin;

select \* from t where id=N for update;

/\*如果行不存在\*/

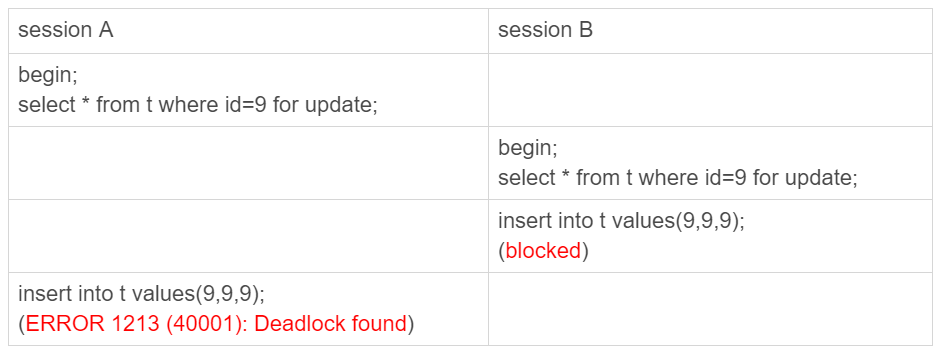
insert into t values(N,N,N);

/\*如果行存在\*/

update t set d=N set id=N;

commit;

上面的例子，并发会导致死锁。



为什么并发会死锁？

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);

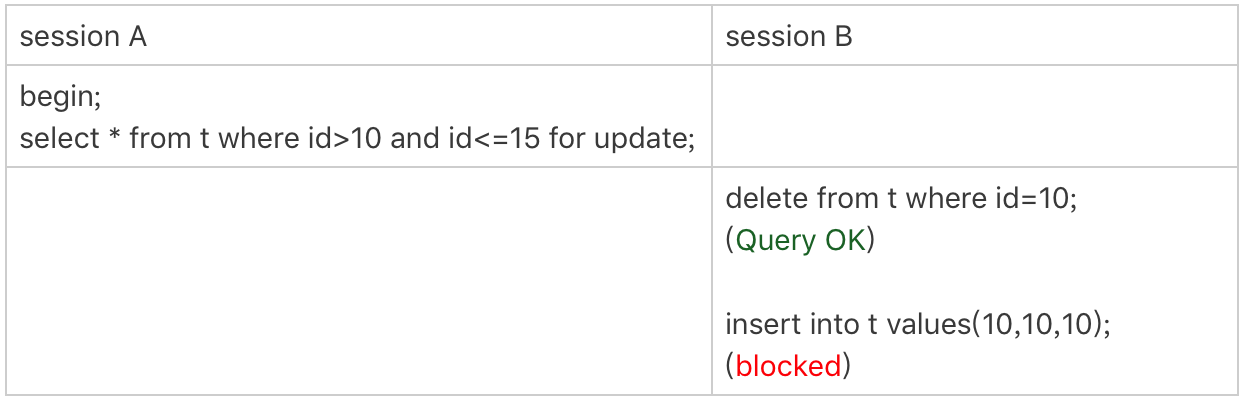
2）session B 执行 select … for update 语句，同样会加上间隙锁 (5,10)，间隙锁之间不会冲突，因此这个语句可以执行成功；

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。

5）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。

例子：

基于文章开头的数据，可重复读隔离级别下，经试验：  
SELECT \* FROM t where c>10 and c<=15 for update;   


由于 session A 并没有锁住 c=10 这个记录，所以 session B 删除 id=10 这一行是可以的。但是之后，session B 再想 insert id=10 这一行回去就不行了.

这是因为，由于 delete 操作把 id=10 这一行删掉了，原来的两个间隙 (5,10)、(10,15）变成了一个 (5,15)。

当我们执行 select \* from t where c>=15 and c<=20 order by c desc lock in share mode;

先定位索引c上最右边c=20的行，所以第一个等值查询会扫描到c=25，然后通过优化2，next-key lock退化为间隙锁，则会加上间隙锁（20，25），紧接着再向左遍历，会加 next-key lock (15, 20], (10, 15], 因为要扫描到c=10才停下来，所以也会加next-key lock (5,10]；

向左扫描到 c=10 的时候，要把 (5, 10]锁起来。

也就是说，所谓“间隙”，其实根本就是由“这个间隙右边的那个记录”定义的。

加锁规则里面，包含了两个“原则”、两个“优化”和一个“bug”。

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。

原则 2：查找过程中访问到的对象才会加锁。

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。

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。

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。