

# MySQL 记录锁+间隙锁可以防止删除操作而导致的幻读吗？

大家好，我是小林。

昨天有位读者在美团二面的时候，被问到关于幻读的问题：

---

林哥，向你请教一个问题。昨天晚上美团二面问到关于 Mysql 一些问题。聊到可重复读情况下解决幻读的方案，我回答的是在当前读的情况下加记录锁与间隙锁解决幻读，然后说到对间隙加锁的目的是防止数据的插入预防幻读。面试官反问我如果这个时候执行删除指令会不会导致幻读

当时面了有一个小时了，就说会出现幻读，但今天回忆起来觉得不确定

不知道这种情况应该怎么判断

面试官反问的大概意思是，MySQL 记录锁+间隙锁可以防止删除操作而导致的幻读吗？

答案是可以的。

接下来，通过几个小实验来证明这个结论吧，顺便再帮大家复习一下记录锁+间隙锁。

# 什么是幻读？

---

首先来看看 MySQL 文档是怎么定义幻读（Phantom Read）的：

*The so-called phantom problem occurs within a transaction when the same query produces different sets of rows at different times. For example, if a SELECT is executed twice, but returns a row the second time that was not returned the first time, the row is a “phantom” row.*

翻译：当同一个查询在不同的时间产生不同的结果集时，事务中就会出现所谓的幻象问题。例如，如果 SELECT 执行了两次，但第二次返回了第一次没有返回的行，则该行是“幻像”行。

举个例子，假设一个事务在 T1 时刻和 T2 时刻分别执行了下面查询语句，途中没有执行其他任何语句：

```
SELECT * FROM t_test WHERE id > 100;
```

只要 T1 和 T2 时刻执行产生的结果集是不相同的，那就发生了幻读的问题，比如：

- T1 时间执行的结果是有 5 条行记录，而 T2 时间执行的结果是有 6 条行记录，那就发生了幻读的问题。
- T1 时间执行的结果是有 5 条行记录，而 T2 时间执行的结果是有 4 条行记录，也是发生了幻读的问题。

## MySQL 是怎么解决幻读的？

MySQL InnoDB 引擎的默认隔离级别虽然是「可重复读」，但是它很大程度上避免幻读现象（并不是完全解决了，详见这篇[文章](#)），解决的方案有两种：

- 针对**快照读**（普通 select 语句），是**通过 MVCC 方式解决了幻读**，因为可重复读隔离级别下，事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，即使中途有其他事务插入了一条数据，是查询不出来这条数据的，所以就很好了避免幻读问题。
- 针对**当前读**（select ... for update 等语句），是**通过 next-key lock（记录锁+间隙锁）方式解决了幻读**，因为当执行 select ... for update 语句的时候，会加上 next-key lock，如果

有其他事务在 next-key lock 锁范围内插入了一条记录，那么这个插入语句就会被阻塞，无法成功插入，所以就很好了避免幻读问题。

## 实验验证

接下来，来验证「MySQL 记录锁+间隙锁[可以防止](#)删除操作而导致的幻读问题」的结论。

实验环境：MySQL 8.0 版本，可重复读隔离级。

现在有一张用户表（t\_user），表里[只有一个主键索引](#)，表里有以下行数据：

id	name	age	reward
1	路飞	19	3000000000
2	索隆	21	11100000000
3	山治	21	1000000000
4	乌索普	19	500000000
5	香克斯	39	4000000000
6	鹰眼	43	3500000000
7	罗	23	3000000000
8	基德	23	3000000000
9	乔巴	17	1000

现在有一个 A 事务执行了一条查询语句，查询到年龄大于 20 岁的用户共有 6 条行记录。

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t_user where age > 20 for update;
```

id	name	age	reward
2	索隆	21	11100000000
3	山治	21	1000000000
7	罗	23	3000000000
8	基德	23	3000000000
5	香克斯	39	4000000000
6	鹰眼	43	3500000000

```
6 rows in set (0.00 sec)
```

事务 A

然后，B 事务执行了一条删除 id = 2 的语句：

```
mysql> delete from t_user where id = 2;
```

事务 B，阻塞。。。

此时，B 事务的删除语句就陷入了等待状态，说明是无法进行删除的。

因此，MySQL 记录锁+间隙锁可以防止删除操作而导致的幻读问题。

## 加锁分析

问题来了，A 事务在执行 select ... for update 语句时，具体加了什么锁呢？

我们可以通过 select \* from performance\_schema.data\_locks\G; 这条语句，查看事务执行 SQL 过程中加了什么锁。

输出的内容很多，共有 11 行信息，我删减了一些不重要的信息：

```
***** 1. row *****
      /// 表级锁: X 型的意向锁      ///
      ENGINE: INNODB
      LOCK_TYPE: TABLE
      LOCK_MODE: IX
      LOCK_DATA: NULL
***** 2. row *****
      /// 行级锁: X 型的 next-key 锁, 范围: (-∞, 1]      ///
      ENGINE: INNODB
      INDEX_NAME: PRIMARY
      LOCK_TYPE: RECORD
      LOCK_MODE: X
      LOCK_DATA: 1
***** 3. row *****
      /// 行级锁: X 型的 next-key 锁, 范围: (1, 2]      ///
      ENGINE: INNODB
      INDEX_NAME: PRIMARY
      LOCK_TYPE: RECORD
      LOCK_MODE: X
      LOCK_DATA: 2
***** 4. row *****
      /// 行级锁: X 型的 next-key 锁, 范围: (2, 3]      ///
      ENGINE: INNODB
      INDEX_NAME: PRIMARY
      LOCK_TYPE: RECORD
      LOCK_MODE: X
      LOCK_DATA: 3
***** 5. row *****
      /// 行级锁: X 型的 next-key 锁, 范围: (3, 4]      ///
      ENGINE: INNODB
      INDEX_NAME: PRIMARY
      LOCK_TYPE: RECORD
      LOCK_MODE: X
      LOCK_DATA: 4
***** 6. row *****
      /// 行级锁: X 型的 next-key 锁, 范围: (4, 5]      ///
      ENGINE: INNODB
      INDEX_NAME: PRIMARY
      LOCK_TYPE: RECORD
      LOCK_MODE: X
      LOCK_DATA: 5
***** 7. row *****
```

```

***** 7. row *****
从上面输出的信息可以看到，共加上了两种不同粒度的锁，分别是：
///// 行级锁：X 型的 next-key 锁，范围：(5, 6] /////
ENGINE: INNODB
INDEX_NAME: PRIMARY
LOCK_TYPE: RECORD
LOCK_MODE: X
LOCK_DATA: 5
这里我们重点关注「行锁」，图中 LOCK_TYPE 中的 RECORD 表示行级锁，而不是记录锁的意思；
***** 8. row *****
///// 行级锁：X 型的 next-key 锁，范围：(6, 7] /////
ENGINE: INNODB
INDEX_NAME: PRIMARY
LOCK_TYPE: RECORD
LOCK_MODE: X
LOCK_DATA: 6
然后通过 LOCK_DATA 信息，可以确认 next-key 锁的范围，具体怎么确定呢？
***** 9. row *****
ENGINE: INNODB
INDEX_NAME: PRIMARY
LOCK_TYPE: RECORD
LOCK_MODE: X
LOCK_DATA: 7
根据我的经验，行级锁：X 型的 next-key 锁，范围：(7, 8] 那么 LOCK_DATA 就表示锁的范围最右值，而锁范围的最左值为 LOCK_DATA 的上一条记录的值。
因此，此时事务 A 在主键索引（INDEX_NAME : PRIMARY）上加了 10 个 next-key 锁，如下：
***** 10. row *****
ENGINE: INNODB
INDEX_NAME: PRIMARY
LOCK_TYPE: RECORD
LOCK_MODE: X
LOCK_DATA: 8
X 型的 next-key 锁，范围：(-∞, 1]
///// 行级锁：X 型的 next-key 锁，范围：(8, 9] /////
ENGINE: INNODB
INDEX_NAME: PRIMARY
LOCK_TYPE: RECORD
LOCK_MODE: X
LOCK_DATA: 9
***** 11. row *****
ENGINE: INNODB
INDEX_NAME: PRIMARY
LOCK_TYPE: RECORD
LOCK_MODE: X
LOCK_DATA: supremum pseudo record
这相当于把整个表给锁住了，其他事务在对表进行增、删、改操作的时候都会被阻塞。
11 rows in set (0.00 sec)

```

只有在事务 A 提交了事务，事务 A 执行过程中产生的锁才会被释放。

为什么只是查询年龄 20 岁以上行记录，而把整个表给锁住了呢？

这是因为事务 A 的这条查询语句是**全表扫描**，锁是在遍历索引的时候加上的，并不是针对输出的结果加锁。



```
1 EXPLAIN select * from t_user where age > 20 for update;
```

全表扫描											
Message Result 1 Profile Status											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	9	33.33	Using where

因此，**在线上在执行 update、delete、select ... for update 等具有加锁性质的语句，一定要检查语句是否走了索引，如果是全表扫描的话，会对每一个索引加 next-key 锁，相当于把整个表锁住了**，这是挺严重的问题。

如果对 age 建立索引，事务 A 这条查询会加什么锁呢？

接下来，我对 **age 字段建立索引**，然后再执行这条查询语句：

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t_user where age > 20 for update;
```

id	name	age	reward
2	索隆	21	11100000000
3	山治	21	10000000000
7	罗	23	30000000000
8	基德	23	30000000000
5	香克斯	39	40000000000
6	鹰眼	43	35000000000

```
6 rows in set (0.00 sec)
```

事务 A

接下来，继续通过 `select * from performance_schema.data_locks\G;` 这条语句，查看事务执行 SQL 过程中加了什么锁。

具体的信息，我就不打印了，我直接说结论吧。

**因为表中有两个索引，分别是主键索引和 age 索引，所以会分别对这两个索引加锁。**

主键索引会加如下的锁：

- X 型的记录锁，锁住 id = 2 的记录；
- X 型的记录锁，锁住 id = 3 的记录；



- X 型的记录锁，锁住 id = 5 的记录；
- X 型的记录锁，锁住 id = 6 的记录；
- X 型的记录锁，锁住 id = 7 的记录；
- X 型的记录锁，锁住 id = 8 的记录；

分析 age 索引加锁的范围时，要先对 age 字段进行排序。

id	name	age ^	reward
9	乔巴	17	1000
1	路飞	19	3000000000
4	乌索普	19	5000000000
2	索隆	21	11100000000
3	山治	21	10000000000
7	罗	23	3000000000
8	基德	23	3000000000
5	香克斯	39	4000000000
6	鹰眼	43	3500000000

范围: (19, 21]

范围: (21, 21]

范围: (21, 23]

范围: (23, 23]

范围: (23, 39]

范围: (39, 43]

范围: (43, +∞]

age 索引加的锁：

- X 型的 next-key lock，锁住 age 范围 (19, 21] 的记录；
- X 型的 next-key lock，锁住 age 范围 (21, 21] 的记录；
- X 型的 next-key lock，锁住 age 范围 (21, 23] 的记录；
- X 型的 next-key lock，锁住 age 范围 (23, 23] 的记录；
- X 型的 next-key lock，锁住 age 范围 (23, 39] 的记录；
- X 型的 next-key lock，锁住 age 范围 (39, 43] 的记录；
- X 型的 next-key lock，锁住 age 范围 (43, +∞] 的记录；

化简一下，age 索引 next-key 锁的范围是 (19, +∞]。

可以看到，对 age 字段建立了索引后，查询语句是索引查询，并不会全表扫描，因此不会把整张表给锁住。

```
1 EXPLAIN select * from t_user where age > 20 for update;
```

Message Result 1 Profile Status											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user	(NULL)	range	index_age	index_age	4	(NULL)	6	100.00	Using index condition

总结一下，在对 age 字段建立索引后，事务 A 在执行下面这条查询语句后，主键索引和 age 索引会加下图中的锁。

	id	name	age	reward
	9	乔巴	17	1000
	1	路飞	19	3000000000
	4	乌索普	19	5000000000
主键索引: id 为 2 的记录锁	2	索隆	21	11100000000
主键索引: id 为 3 的记录锁	3	山治	21	10000000000
主键索引: id 为 7 的记录锁	7	罗	23	30000000000
主键索引: id 为 8 的记录锁	8	基德	23	30000000000
主键索引: id 为 5 的记录锁	5	香克斯	39	40000000000
主键索引: id 为 6 的记录锁	6	鹰眼	43	35000000000

age 索引: next-key 范围 (19, +∞]

事务 A 加上锁后，事务 B、C、D、E 在执行以下语句都会被阻塞。

事务 A	事务 B	事务 C	事务 D	事务 E
begin	begin	begin	begin	begin
select * from t_user where age > 20 for update;				
	update t_user set age = 20 where id = 1; // 阻塞			
		delete from t_user where id = 2; // 阻塞		
			delete from t_user where age = 23; // 阻塞	
				insert into t_user (age) values (100); // 阻塞

## 总结

在 MySQL 的可重复读隔离级别下，针对当前读的语句会对索引加记录锁+间隙锁，这样可以避免其他事务执行增、删、改时导致幻读的问题。

有一点要注意的是，在执行 update、delete、select ... for update 等具有加锁性质的语句，一定要检查语句是否走了索引，如果是全表扫描的话，会对每一个索引加 next-key 锁，相当于把整个表锁住了，这是挺严重的问题。

完！