从一道数据库面试题彻谈MySQL加锁机制

编者荐语：
之前我在腾讯云开发者的一篇MySQL加锁分析的文章，分享给大家。

👉导读

有一道关于「数据库锁」的面试题。我们发现其实很多 DBA （数据库管理员，Database administrator）包括工作好几年的 DBA 都答得不太好。这说明 MySQL 锁的机制其实还是比较复杂，值得深入研究。本文对3条简单的查询语句加锁情况进行分析，以期帮助各位开发者彻底搞清楚加锁细节。欢迎阅读~

👉看目录，点收藏

首先来看这个面试题：

已知表t是 innodb 引擎，有主键：id（int类型），下面3条语句是否加锁？加锁的话，是什么锁？

1. select * from t where id=X;

2. begin;select * from t where id=X;

3. begin;select * from t where id=X for update;

这里其实有很多依赖条件，并不能一开始就给出一个很确定的答复。我们一层层展开来分析。

MySQL 有哪些锁?

锁类型	范围	是否innodb锁	锁的信息表
Flush tables with read lock (FTWRL)	全局	否	
flush tables … with read lock;	表级	否	performance_schema.table_handles
lock tables … read/write	表级	否	performance_schema.table_handles
metadata lock	表级	否	performance_schema.metadata_locks
Intention Locks	表级	是	performance_schema.data_locks
Record Locks	行级	是	performance_schema.data_locks
Gap Locks	行级	是	performance_schema.data_locks
Next-Key Locks	行级	是	performance_schema.data_locks
Insert Intention Locks	行级	是	performance_schema.data_locks
Shared and Exclusive Locks	行级	是	performance_schema.data_locks
AUTO-INC Locks	表级	是	
Predicate Locks for Spatial Indexes	行级	是	

首先要知道 MySQL 有哪些锁。如上图所示，至少有12类锁。

其中，自增锁是事务向包含了 AUTO_INCREMENT 列的表中新增数据时会持有， predicate locks for spatial index 为空间索引专用，本文不讨论这2类锁。

锁按粒度可分为全局、表级、行级，共3类。

1.1 全局锁

对整个数据库实例加锁。

加锁表现：数据库处于只读状态，阻塞对数据的所有 DML/DDL
加锁方式： Flush tables with read lock
释放锁：unlock tables(发生异常时会自动释放)
作用场景：全局锁主要用于做数据库实例的逻辑备份，与设置数据库只读命令 set global readonly=true 相比，全局锁在发生异常时会自动释放

1.2 表锁

对操作的整张表加锁，锁定颗粒度大，资源消耗少。不会出现死锁，但会导致写入并发度低。具体又分为3类：

1）显式表锁

分为共享锁（S）和排他锁（X）
显式加锁方式：lock tables ... read/write
释放锁：unlock tables(连接中断也会自动释放)

2） Metadata-Lock （元数据锁）

MySQL5.5版本开始引入，主要功能是并发条件下，防止session1的查询事务未结束的情况下，session2对表结构进行修改，保护元数据的一致性。

在 session1 持有 metadata-lock 的情况下， session2 处于等待状态：show proces slist 可见 Waiting for table metadata lock 。其具体加锁机制如下：

- DML->先加MDL 读锁（ SHARED_READ，SHARED_WRITE ）
- DDL->先加MDL 写锁（ EXCLUSIVE ）
- 读锁之间兼容
- 读写锁之间、写锁之间互斥

3）Intention Locks（意向锁）

意向锁为表锁（表示为IS或者IX），由存储引擎自己维护，用户无法干预。

下面举一个例子说明其功能。假设有2个事务：T1和T2
T1: 锁住表中的一行，只能读不能写（行级读锁）。
T2: 申请整个表地写锁（表级写锁）。
如T2申请成功，则能任意修改表中的一行，但这与T1持有的行锁是冲突的。故数据库应识别这种冲突，让T2的锁申请被阻塞，直到T1释放行锁。

有2种方法可以实现冲突检测：
1、判断表是否已被其他事务用表锁锁住；
2、判断表中的每一行是否已被行锁锁住。

其中2需要遍历整个表，效率太低。因此 innodb 使用意向锁来解决这个问题：T1需要先申请表的意向共享锁（IS），成功后再申请某一行的记录锁S。

在意向锁存在的情况下，上面的判断可以改为：T2发现表上有意向共享锁IS，因此申请表地写锁被阻塞。

1.3 行锁

InnoDB 引擎支持行级别锁，行锁粒度小，并发度高，但加锁开销大，也可能会出现死锁。

加锁机制：innodb 行锁锁住的是索引页，回表时，主键地聚簇索引也会加上锁。

- X：代表Next-Key Lock锁定记录本身和记录之前的间隙（X）。
- S：代表Next-Key Lock锁定记录本身和记录之前的间隙（S）。
- X, REC_NOT_GAP：代表只锁定记录本身（X）。
- S, REC_NOT_GAP：代表只锁定记录本身（S）。
- X, GAP：代表间隙锁，不锁定记录本身（X）。
- S, GAP：代表间隙锁，不锁定记录本身（S）。
- X, GAP,INSERT_INTENTION：代表插入意向锁。

行锁具体类别如上图所示，包括： Record lock/Gap Locks/Next-Key Locks ，每类又可分为共享锁（S）或者排它锁（X），一共2*3=6类，最后还有1类插入意向锁：

Record lock （记录锁）：最简单的行锁，仅仅锁住一行。记录锁永远都是加在索引上的，即使一个表没有索引， InnoDB 也会隐式地创建一个索引，并使用这个索引实施记录锁。

Gap Locks （间隙锁）：加在两个索引值之间的锁，或者加在第一个索引值之前，或最后一个索引值之后的间隙。使用间隙锁锁住的是一个区间，而不仅仅是这个区间中的每一条数据。间隙锁只阻止其他事务插入到间隙中，不阻止其他事务在同一个间隙上获得间隙锁，所以 gap x lock 和 gap s lock 有相同的作用。它是一个左开右开区间：如（1，3）。

Next-Key Locks： 记录锁和间隙锁的组合，它指的是加在某条记录以及这条记录前面间隙上的锁。它是一个左开右闭区间：如（1，3】。

Insert Intention （插入意向锁）： 该锁只会出现在 insert 操作执行前（并不是所有insert操作都会出现），目的是提高并发插入能力。它在插入一行记录操作之前设置一种特殊的间隙锁，多个事务在相同的索引间隙插入时，如果不是插入间隙中相同的位置就不需要互相等待。

TIPS:
1.不存在 unlock tables ... read/write， 只有 unlock tables
2. If a session begins a transaction, an implicit UNLOCK TABLES is performed

锁的兼容情况

引入意向锁后，表锁之间的兼容性情况如下表：

	X	IX	S	IS
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	Compatible	Conflict	Compatible
S	Conflict	Conflict	Compatible	Compatible
IS	Conflict	Compatible	Compatible	Compatible

总结：

意向锁之间都兼容。

X,IX和其它都不兼容（除了1）。

S,IS和其它都兼容（除了1,2）。

锁信息查看方式

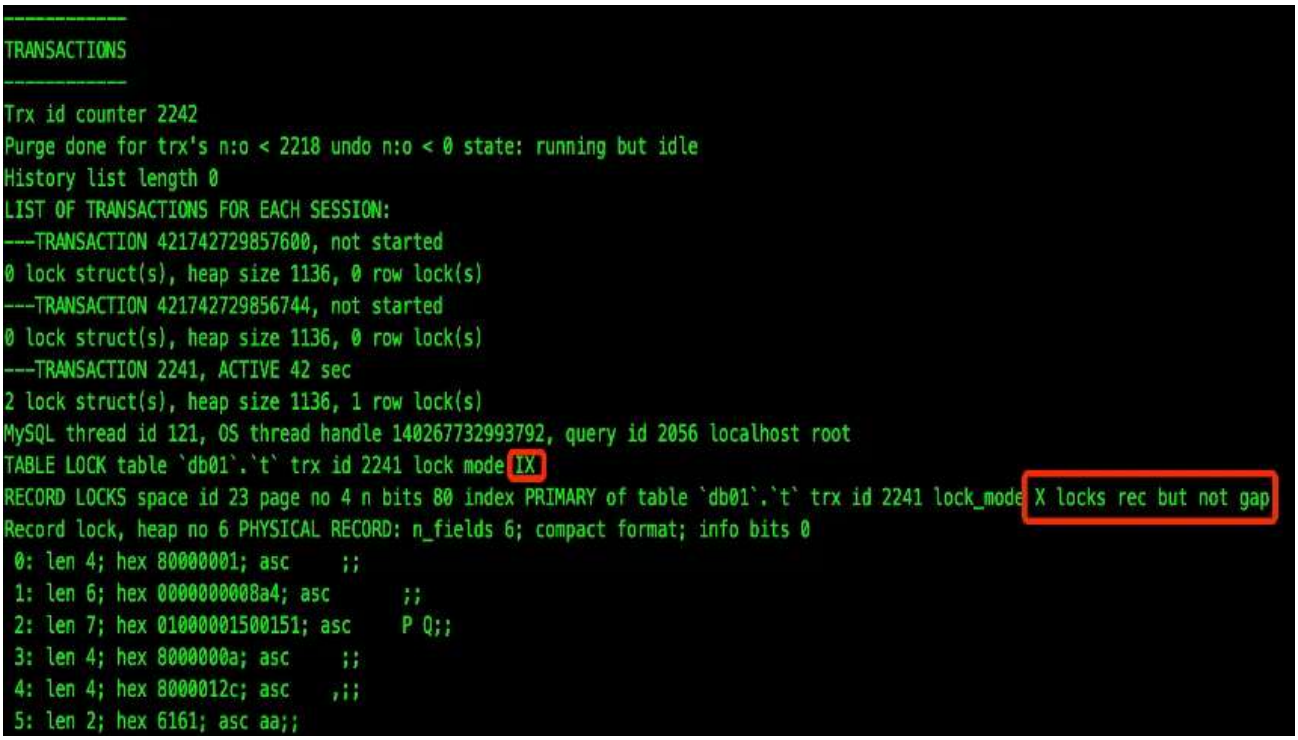
MySQL 5.6.16版本之前，需要建立一张特殊表 innodb_lock_monitor ，然后使用 show engine innodb status 查看。

```
CREATE TABLE innodb_lock_monitor (a INT) ENGINE=INNODB;
DROP TABLE innodb_lock_monitor;
```

MySQL 5.6.16版本之后，修改系统参数 innodb_status_output 后，使用show engine innodb status 查看。

```
set GLOBAL innodb_status_output=ON;
set GLOBAL innodb_status_output_locks=ON;
```

每15秒输出一次INNODB运行状态信息到错误日志。



MySQL 5.7 版本之后
可以通过 information_schema.innodb_locks 查看事务的锁情况，但只能看到阻塞事务的锁；如果事务并未被阻塞，则在该表中看不到该事务的锁情况。

MySQL 8.0
删除 information_schema.innodb_locks ，添加 performance_schema.data_locks ，即使事务并未被阻塞，依然可以看到事务所持有的锁，同时通过 performance_schema.table_handles 、 performance_schema.metadata_locks 可以非常方便地看到元数据锁等表锁。

测试环境搭建

4.1 建立测试表

该表包含一个主键，一个唯一键和一个非唯一键：

```
CREATE TABLE `t` (
`id` int(11) NOT NULL,
```

```
`a` int(11) DEFAULT NULL,
`b` int(11) DEFAULT NULL,
`c` varchar(10),
PRIMARY KEY (`id`),
unique KEY `a` (`a`),
key b(b))
ENGINE=InnoDB;
```

4.2 写入测试数据

```
insert into t values(1,10,100,'a'),(3,30,300,'c'),(5,50,500,'e');
```

记录存在时的加锁

对于innodb引擎来说，加锁的2个决定因素：

一、当前的事务隔离级别。
二、当前记录是否存在。

假设 id 为3的记录存在，则在不同的4个隔离级别下3个语句的加锁情况汇总如下表(select 3表示 select * from t where id =3)：

隔离级别	select 2	begin;select 2	begin;select 2 for update
RU	无	SHARED_READ	SHARED_WRITE IX X,REC_NOT_GAP：3
RC	无	SHARED_READ	SHARED_WRITE IX X,REC_NOT_GAP：3

RR	无	SHARED_READ	SHARED_WRITE IX X,REC_NOT_GAP：3
Serial	无	SHARED_READ IS S,REC_NOT_GAP：3	SHARED_WRITE IX X,REC_NOT_GAP：3

分析：

使用以下语句在4种隔离级别之间切换：

```
set global transaction_isolation='READ-UNCOMMITTED';
set global transaction_isolation='READ-COMMITTED';
set global transaction_isolation='REPEATABLE-READ';
set global transaction_isolation='Serializable';
```

对于 auto commit=true， select 没有显式开启事务（ begin ）的语句，元数据锁和行锁都不加，是真的“读不加锁”。

对于 begin ; select ... where id =3这种只读事务，会加元数据锁SHARED_READ，防止事务执行期间表结构变化，查询performance_schema.metadata_locks 表可见此锁：

```
+-------------+--------------------+----------------+-------------+-----------------------+-------------+---------------+-------------+-----------------+-----------------+
| OBJECT_TYPE | OBJECT_SCHEMA      | OBJECT_NAME    | COLUMN_NAME | OBJECT_INSTANCE_BEGIN | LOCK_TYPE   | LOCK_DURATION | LOCK_STATUS | SOURCE          | OWNER_THREAD_ID | OWNER_EVENT_ID |
+-------------+--------------------+----------------+-------------+-----------------------+-------------+---------------+-------------+-----------------+-----------------+----------------+
| TABLE       | db01               | t              | NULL        |      140265852664400 | SHARED_READ | TRANSACTION   | GRANTED     | sql_parse.cc:6161 |             174 |             27 |
| TABLE       | performance_schema | metadata_locks | NULL        |      140265714618288 | SHARED_READ | TRANSACTION   | GRANTED     | sql_parse.cc:6161 |             175 |              9 |
+-------------+--------------------+----------------+-------------+-----------------------+-------------+---------------+-------------+-----------------+-----------------+----------------+
```

对于 begin; select ... where id =3这种只读事务，MySQL在RC和RR隔离级别下，使用 MVCC 快照读，不加行锁，但在Serial隔离级别下，读写互斥，会加意向共享锁（表锁）和共享记录锁（行锁）。

对于begin; select ... where id=3 for update，会加元数据锁SHARED_WRITE。

对于begin; select ... where id=3 or update，4种隔离级别都会加意向排它锁（表锁）和排它记录锁（行锁），查询 performance_schema.data_locks 可见此2类锁。

```
+------------+-----------+---------------+-----------+
| INDEX_NAME | LOCK_TYPE | LOCK_MODE     | LOCK_DATA |
+------------+-----------+---------------+-----------+
| NULL       | TABLE     | IX            | NULL      |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | 3         |
+------------+-----------+---------------+-----------+
```

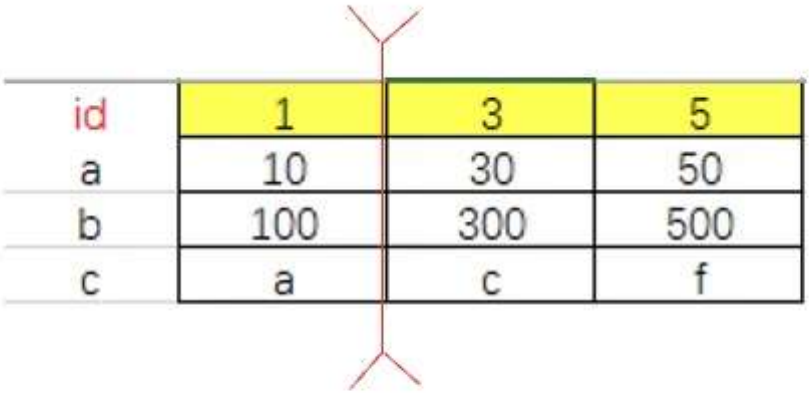
记录不存在时的加锁

隔离级别	select 2	begin;select 2	begin;select 2 for update
RU	无	SHARED_READ	SHARED_WRITE IX
RC	无	SHARED_READ	SHARED_WRITE IX
RR	无	SHARED_READ	SHARED_WRITE IX X,GAP：3
Serial	无	SHARED_READ IS S,GAP：3	SHARED_WRITE IX X,GAP：3

分析：
当记录不存在的时候，RU和RC隔离级别只有意向锁，没有行锁了。

RR，Serial 隔离级别下，记录锁变成了 Gap Locks（间隙锁），可以防止幻读，lock_data 为3的 GAP lock 锁住区间（1，3），此时ID=2的记录插入会被阻塞。

```
+------------+-----------+-----------+-----------+-------------+
| INDEX_NAME | lock_type | lock_mode | lock_data | LOCK_STATUS |
+------------+-----------+-----------+-----------+-------------+
| NULL       | TABLE     | IX        | NULL      | GRANTED     |
| PRIMARY    | RECORD    | X,GAP     | 3         | GRANTED     |
+------------+-----------+-----------+-----------+-------------+
```



那么对于主键范围查询，唯一键查询，非唯一键查询，在不同隔离级别下又是如何加锁的呢？

构造测试环境

该表包含一个主键，一个唯一键和一个非唯一键，有3条测试记录：

```
CREATE TABLE `t` (
`id` int(11) NOT NULL,
`a` int(11) DEFAULT NULL,
`b` int(11) DEFAULT NULL,
`c` varchar(10),
```

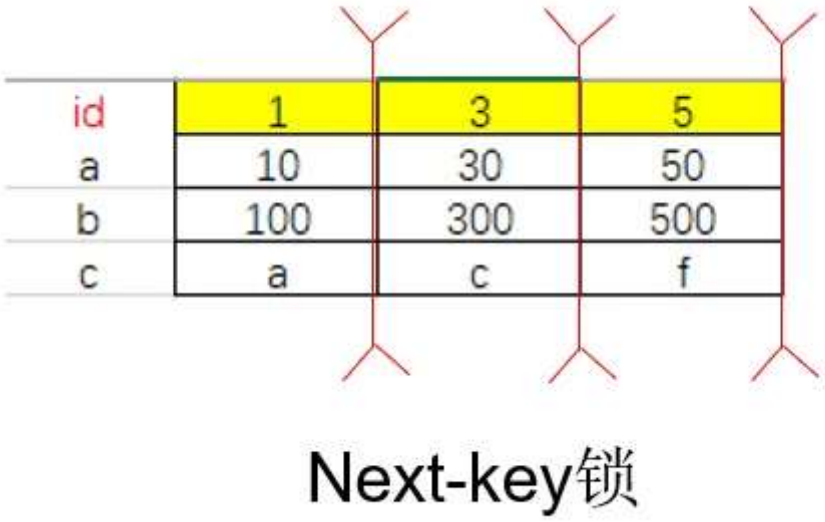
```
PRIMARY KEY (`id`),
unique KEY `a` (`a`),
key b(b))
ENGINE=InnoDB;
insert into t values(1,10,100,'a'),(3,30,300,'c'),(5,50,500,'e');
```

主键范围读取

8.1 RR隔离级别

begin;
select * from t where id>1 and id<7 for update;

```
+-----------+------------+-----------+-----------+-------------+-------------------------+
| THREAD_ID | INDEX_NAME | lock_type | lock_mode | lock_status | lock_data               |
+-----------+------------+-----------+-----------+-------------+-------------------------+
|       228 | NULL       | TABLE     | IX        | GRANTED     | NULL                    |
|       228 | PRIMARY    | RECORD    | X         | GRANTED     | supremum pseudo-record  |
|       228 | PRIMARY    | RECORD    | X         | GRANTED     | 3                       |
|       228 | PRIMARY    | RECORD    | X         | GRANTED     | 5                       |
+-----------+------------+-----------+-----------+-------------+-------------------------+
```



原则1：innodb 行锁锁住的是索引页。

原则2：索引查找过程中访问到的对象会加锁。

原则3：RR 隔离级别为了防止幻读，存在间隙锁（GAP LOCK）。

原则4：加锁的基本单位是 next-key lock，next-key lock 是前开后闭区间。

所以加了3个 X 锁（锁定记录本身和之前的区间，等于间隙锁+行锁），分别锁定(1,3】，(3,5】，(5,+∞】区间。

说明：
1. InnoDB 给每个索引加了一个不存在的最大值 supremum，代表+∞
2. 幻读：当某个事务在读取某个范围内的记录时，另一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻读。

唯一索引等值查询

9.1 RR隔离级别

```
begin;
select * from t where a=30 for update;
```

原则1：innodb 行锁锁住的是索引页，回表时，主键地聚簇索引也会加上锁。

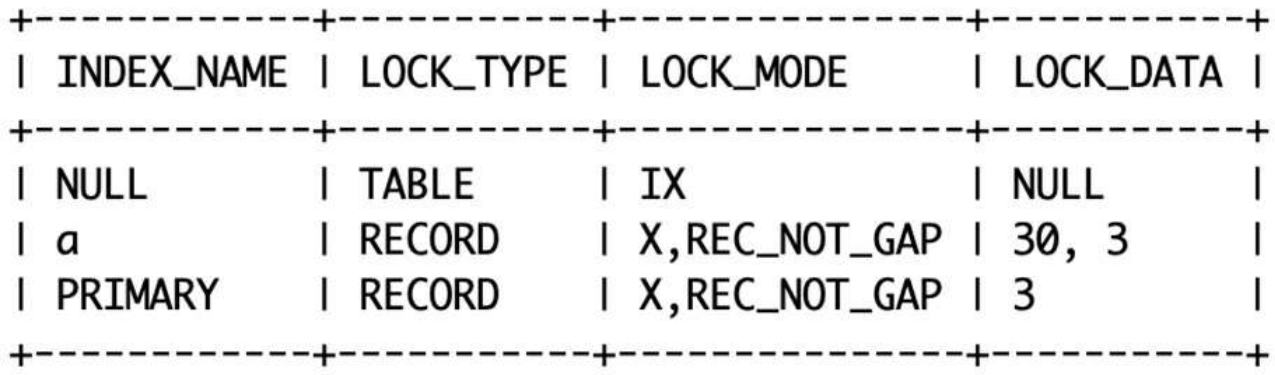
原则2：二级索引（非聚簇索引）的叶子节点包含了引用行的主键值。

原则3：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。

所以加了2个记录锁，记录锁30，3代表锁定唯一索引 a 上的（id=3,a=30）这条记录，记录锁3代表锁定了主键上的 id=3 这条记录。

9.2 RC隔离级别

begin;
select * from t where a=30 for update;



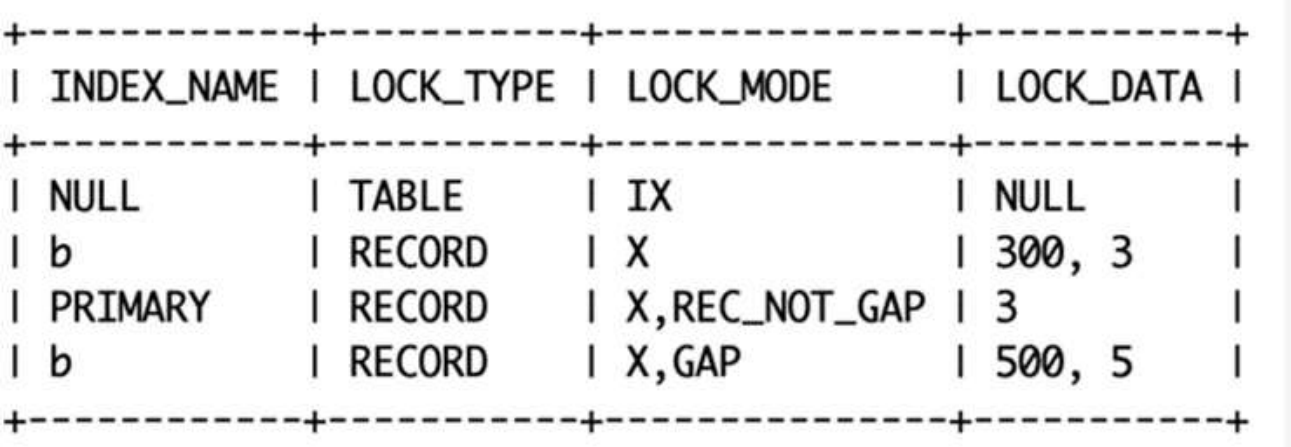
INDEX_NAME	LOCK_TYPE	LOCK_MODE	LOCK_DATA
NULL	TABLE	IX	NULL
a	RECORD	X,REC_NOT_GAP	30, 3
PRIMARY	RECORD	X,REC_NOT_GAP	3

对于该条语句，RC 隔离级别下加锁完全一样。

非唯一索引等值查询

10.1 RR隔离级别

begin;
select * from t where b=300 for update;



INDEX_NAME	LOCK_TYPE	LOCK_MODE	LOCK_DATA
NULL	TABLE	IX	NULL
b	RECORD	X	300, 3
PRIMARY	RECORD	X,REC_NOT_GAP	3
b	RECORD	X,GAP	500, 5

原则：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。

所以对于非唯一索引 b，锁定了((b=100,id=1),(b=300,id=3)】区间和((b=300,id=3),(b=500,id=5))区间和主键上的 id=3。

```
begin;
select * from t where b=400 for update;
```

可以看到，查询得值 b=400不存在，但加锁情况和 b=300值存在的时候是一样的。

10.2 RC隔离级别

```
begin;
select * from t where b=300 for update;
```

```
+------------+-----------+---------------+-----------+
| INDEX_NAME | LOCK_TYPE | LOCK_MODE     | LOCK_DATA |
+------------+-----------+---------------+-----------+
| NULL       | TABLE     | IX            | NULL      |
| b          | RECORD    | X,REC_NOT_GAP | 300, 3    |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | 3         |
+------------+-----------+---------------+-----------+
```

原则：读提交隔离级别 (read-committed) 只有行锁，没有间隙锁。

所以只锁定了锁引 b 上的 (b=300,id=3) 和主键上的 id=3。

```
begin;
select * from t where b=400 for update;
```

INDEX_NAME	LOCK_TYPE	LOCK_MODE	LOCK_DATA
NULL	TABLE	IX	NULL

因为 RC 隔离级别没有间隙锁，所以 b=400 值不存在的时候，只有IX意向排它锁。

非唯一索引加覆盖索引

11.1 RR隔离级别

select id from t where b=300 lock in share mode;

INDEX_NAME	lock_type	lock_mode	lock_data	LOCK_STATUS
NULL	TABLE	IS	NULL	GRANTED
b	RECORD	S	300, 3	GRANTED
b	RECORD	S,GAP	500, 5	GRANTED

原则：如果一个索引包含所有需要查询的字段的值，就是覆盖索引，对于二级索引来说，可以避免对主键索引的查询（回表）。

因为二级索引 b 包括 (b,id)，所以主键索引上无锁。

因为是 lock in share mode 所以加的是共享锁（S）和共享意向锁（IS）。

无索引

```
begin;
select * from t where c='aa' for update;
```

```
+------------+-----------+-----------+------------------------+
| INDEX_NAME | LOCK_TYPE | LOCK_MODE | LOCK_DATA              |
+------------+-----------+-----------+------------------------+
| NULL       | TABLE     | IX        | NULL                   |
| PRIMARY    | RECORD    | X         | supremum pseudo-record |
| PRIMARY    | RECORD    | X         | 1                      |
| PRIMARY    | RECORD    | X         | 3                      |
| PRIMARY    | RECORD    | X         | 5                      |
+------------+-----------+-----------+------------------------+
```

没有索引的时候，要全表扫描，有主键就扫主键。

所以锁定范围：(-∞,1]、(1,3]、(3,5]、(5,+supremum]，可以看出来把整张表都锁住了，所以对于实时业务一定要避免非索引查询。

总结

以上就是MySQL 加锁机制的详细分析，希望对你有帮助。

公众号"数据库之巅"分享这十几年来我在数据库特别是互联网金融数据库运维走过的路和踩过的坑，欢迎大家关注。