

MySQL锁篇

一、一条update语句

我们的故事继续发展，我们还是使用t这个表：

```
1 CREATE TABLE t (  
2     id INT PRIMARY KEY,  
3     c VARCHAR(100)  
4 ) Engine=InnoDB CHARSET=utf8;
```

现在表里的数据就是这样的：

```
1 mysql> SELECT * FROM t;  
2 +----+-----+  
3 | id | c      |  
4 +----+-----+  
5 | 1  | 刘备   |  
6 +----+-----+  
7 1 row in set (0.01 sec)
```

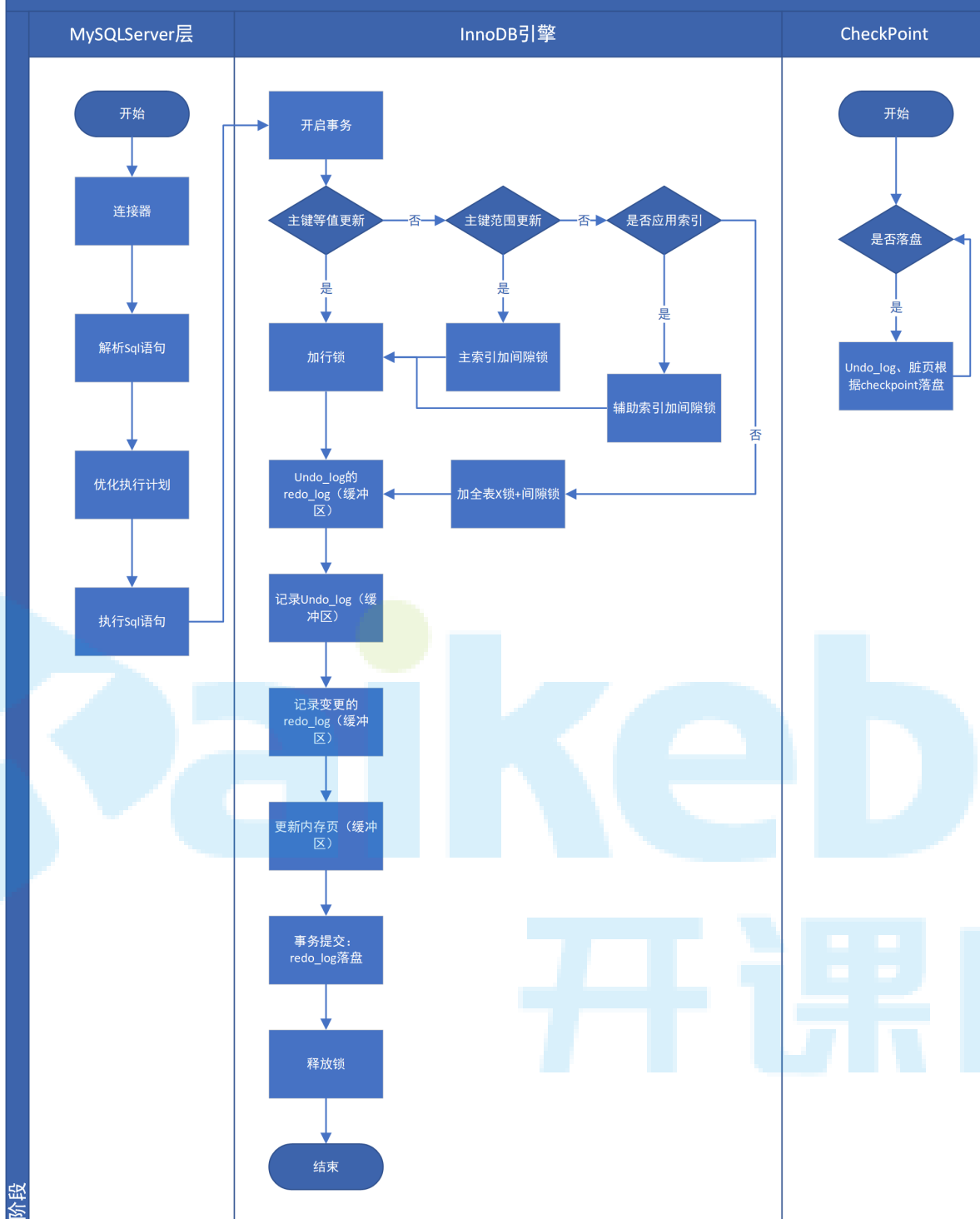
然后更新表里的一条数据：

```
1 update t set c='曹操' where id = 1;
```

执行流程：

开课吧

Update语句执行流程



二、MySQL锁介绍

- 按照锁的粒度来说，MySQL主要包含三种类型（级别）的锁定机制：
 - 全局锁：锁的是整个database。由MySQL的SQL layer层实现的
 - 表级锁：锁的是某个table。由MySQL的SQL layer层实现的
 - 行级锁：锁的是某行数据，也可能锁定行之间的间隙。由某些存储引擎实现，比如InnoDB。
- 按照锁的功能来说分为：**共享锁**和**排他锁**。

共享锁Shared Locks (S锁)：

- 兼容性：加了S锁的记录，允许其他事务再加S锁，不允许其他事务再加X锁
- 加锁方式：select...lock in share mode

排他锁Exclusive Locks (X锁)：

- 1、兼容性：加了X锁的记录，不允许其他事务再加S锁或者X锁
- 2、加锁方式：select...for update

三、全局锁

全局锁就对整个数据库实例加锁，加锁后整个实例就处于只读状态，后续的MDL的写语句，DDL语句，已经更新操作的事务提交语句都将被阻塞。其典型的使用场景是做全库的逻辑备份，对所有的表进行锁定，从而获取一致性视图，保证数据的完整性。

加全局锁的命令为：

```
1 | mysql> flush tables with read lock;
```

释放全局锁的命令为：

```
1 | mysql> unlock tables;
```

或者断开加锁session的连接，自动释放全局锁。

说到全局锁用于备份这个事情，还是很危险的。因为如果在主库上加全局锁，则整个数据库将不能写入，备份期间影响业务运行，如果在从库上加全局锁，则会导致不能执行主库同步过来的操作，造成主从延迟。

对于innodb这种支持事务的引擎，使用mysqldump备份时可以使用--single-transaction参数，利用mvcc提供一致性视图，而不使用全局锁，不会影响业务的正常运行。而对于有MyISAM这种不支持事务的表，就只能通过全局锁获得一致性视图，对应的mysqldump参数为--lock-all-tables。

四、MySQL表级锁

1、表级锁介绍

MySQL的表级锁有四种：

- 1、表读、写锁。
- 2、元数据锁（meta data lock，MDL）。
- 3、意向锁 Intention Locks（InnoDB）
- 4、自增锁（AUTO-INC Locks）

2、表读S、写锁X

1) 表锁相关命令

- MySQL 实现的表级锁定的争用状态变量：

```
1 | mysql> show status like 'table%';
```

```
mysql> show status like 'table%';
```

Variable_name	Value
Table_locks_immediate	113
Table_locks_waited	0
Table_open_cache_hits	5
Table_open_cache_misses	1
Table_open_cache_overflows	0

- 1 - table_locks_immediate: 产生表级锁定的次数;
- 2 - table_locks_waited: 出现表级锁定争用而发生等待的次数;

- 表锁有两种表现形式:

- 1 表共享读锁 (Table Read Lock)
- 2 表独占写锁 (Table Write Lock)

- 手动增加表锁:

```
1 lock table 表名称 read(write),表名称2 read(write), 其他;
```

- 查看表锁情况:

```
1 show open tables;
```

- 删除表锁:

```
1 unlock tables;
```

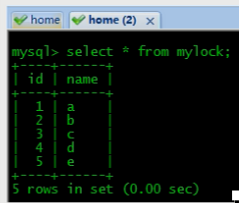
2) 表锁演示

1.环境准备

```
1 CREATE TABLE mylock (
2     id int(11) NOT NULL AUTO_INCREMENT,
3     NAME varchar(20) DEFAULT NULL,
4     PRIMARY KEY (id)
5 );
6
7 INSERT INTO mylock (id,NAME) VALUES (1, 'a');
8 INSERT INTO mylock (id,NAME) VALUES (2, 'b');
9 INSERT INTO mylock (id,NAME) VALUES (3, 'c');
10 INSERT INTO mylock (id,NAME) VALUES (4, 'd');
```

2.读锁演示

我们为mylock表加read锁(读阻塞写例子)

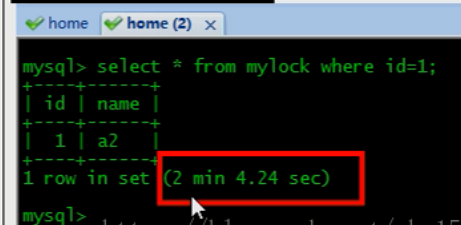
session_1	session_2
<p>获得表mylock的READ锁定</p> <pre>mysql> lock table mylock read; Query OK, 0 rows affected (0.00 sec) mysql></pre>	<p>连接终端</p>
<p>当前session可以查询该表记录</p> <pre>mysql> select * from mylock; +----+-----+ id name +----+-----+ 1 a 2 b 3 c 4 d 5 e +----+-----+ 5 rows in set (0.00 sec)</pre>	<p>其他session也可以查询该表的记录</p> 

<p>当前session不能查询其它没有锁定的表</p> <pre>mysql> select * from book; ERROR 1100 (HY000): Table 'book' was not locked with LOCK TABLES mysql></pre>	<p>其他session可以查询或者更新未锁定的表</p> <pre>mysql> update staffs set name='z2' where id=1; Query OK, 1 row affected (0.02 sec) Rows matched: 1 Changed: 1 Warnings: 0 mysql> select * from staffs; +----+-----+-----+-----+-----+ id NAME age pos add_time +----+-----+-----+-----+-----+ 1 z2 21 manager 2016-02-14 23:01:33 2 z4 22 manager 2016-02-14 23:01:34 +----+-----+-----+-----+-----+</pre>
<p>当前session中插入或者更新锁定的表都会提示错误：</p> <pre>mysql> select * from mylock; +----+-----+ id name +----+-----+ 1 a 2 b 3 c 4 d 5 e +----+-----+ 5 rows in set (0.00 sec) mysql> insert into mylock(name) values('e'); ERROR 1099 (HY000): Table 'mylock' was locked with a READ lock and can't be updated mysql> update mylock set name='k' where id=1; ERROR 1099 (HY000): Table 'mylock' was locked with a READ lock and can't be updated mysql></pre>	<p>其他session插入或者更新锁定表会一直等待获得锁：</p> <pre>mysql> insert into mylock(name) values('e');</pre>

3.写锁演示

我们为mylock表加write锁(MyISAM存储引擎的写阻塞读例子)

session_1	session_2
<p>获得表mylock的WRITE锁定</p> <pre>mysql> lock tables mylock write; Query OK, 0 rows affected (0.00 sec) mysql></pre>	<p>待Session1开启写锁后，session2再连接终端</p>
<p>当前session对锁定表的查询+更新+插入操作都可以执行：</p> <pre>mysql> select * from mylock where id=1; +----+-----+ id name +----+-----+ 1 a +----+-----+ 1 row in set (0.00 sec) mysql> update mylock set name='a2' where id=1; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0 mysql> insert into mylock(name) values('f'); Query OK, 1 row affected (0.00 sec) mysql> select * from mylock; +----+-----+ id name +----+-----+ 1 a2 +----+-----+</pre>	<p>其他session对锁定表的查询被阻塞，需要等待锁被释放：</p>  <p>备注：如果可以，请换成不同的id来进行测试，因为mysql聪明有缓存，第2次的条件会从缓存取得，影响锁效果演示</p>

<p>释放锁</p> <pre>mysql> unlock tables; Query OK, 0 rows affected (0.00 sec) mysql></pre>	<p>Session2获得锁，查询返回：</p> 
---	---

3、元数据锁

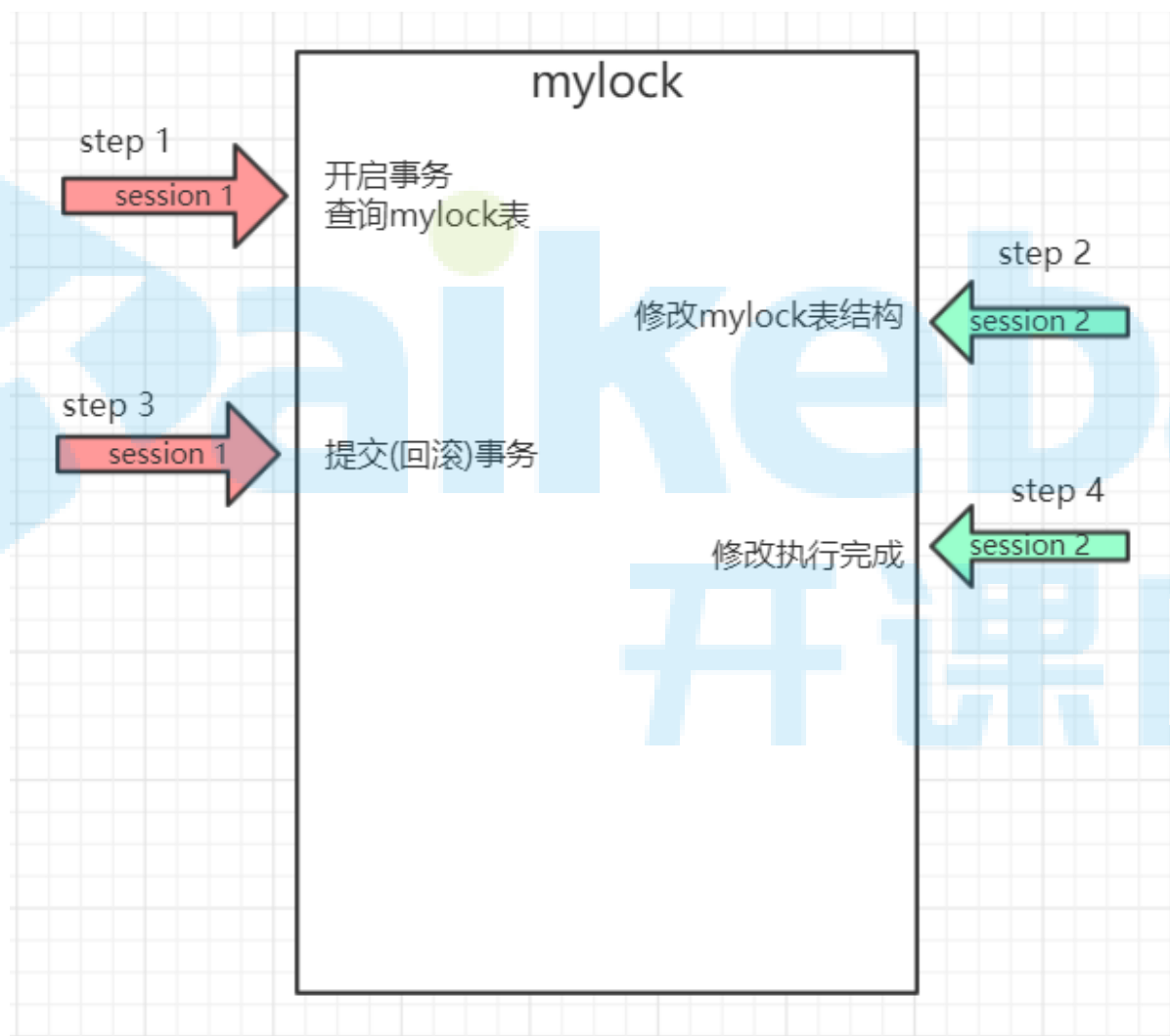
1) 元数据锁介绍

MDL不需要显式使用，在访问一个表的时候会被自动加上。MDL的作用是，保证读写的正确性。你可以想象一下，如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个表结构做变更，删了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。

因此，在MySQL 5.5版本中引入了MDL，当对一个表做增删改查操作的时候，加MDL读锁；当要对表做结构变更操作的时候，加MDL写锁。

- 读锁之间不互斥，因此你可以有多个线程同时对一张表增删改查。
- 读写锁之间、写锁之间是互斥的，用来保证变更表结构操作的安全性。因此，如果有两个线程要同时给一个表加字段，其中一个要等另一个执行完才能开始执行。

2) 元数据锁演示



session1 (Navicat) 、 session2 (mysql)

```
1 1、session1: begin;--开启事务
2       select * from mylock;--加MDL读锁
3 2、session2: alter table mylock add f int; -- 修改阻塞
4 3、session1: commit; --提交事务 或者 rollback 释放读锁
5 4、session2: Query OK, 0 rows affected (38.67 sec) --修改完成
6       Records: 0 Duplicates: 0 Warnings: 0
7
8
```

4、自增锁(AUTO-INC Locks)

AUTO-INC锁是一种特殊的表级锁，发生涉及AUTO_INCREMENT列的事务性插入操作时产生。

五、MySQL行级锁

1、行级锁介绍

MySQL的**行级锁**，是由**存储引擎**来实现的，这里我们主要讲解**InnoDB**的行级锁。

InnoDB行锁是通过给索引上的**索引项加锁来实现的**，因此InnoDB这种行锁实现特点意味着：**只有通过索引条件检索的数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！**

- InnoDB的行级锁，按照**锁定范围**来说，分为四种：

- 1 记录锁（Record Locks）：锁定索引中一条记录。
- 2 间隙锁（Gap Locks）：要么锁住索引记录中间的值，要么锁住第一个索引记录前面的值或者最后一个索引记录后面的值。
- 3 临键锁（Next-Key Locks）：是索引记录上的记录锁和在索引记录之前的间隙锁的组合（间隙锁+记录锁）。
- 4 插入意向锁（Insert Intention Locks）：做insert操作时添加的对记录id的锁。

- InnoDB的行级锁，按照功能来说，分为两种：

- 1 共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。
- 2 排他锁（X）：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。

对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X）；对于普通SELECT语句，InnoDB不会加任何锁，事务可以通过以下语句显示给记录集加共享锁或排他锁。

手动添加共享锁（S）：

```
1 SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE
```

手动添加排他锁（x）：

```
1 SELECT * FROM table_name WHERE ... FOR UPDATE
```

案例：

```
1 CREATE TABLE `t1_simple` (  
2   `id` int(11) NOT NULL,  
3   `pubtime` int(11) NULL DEFAULT NULL,  
4   PRIMARY KEY (`id`) USING BTREE,  
5   INDEX `idx_pu` (`pubtime`) USING BTREE  
6 ) ENGINE = InnoDB;  
7 INSERT INTO `t1_simple` VALUES (1, 10);  
8 INSERT INTO `t1_simple` VALUES (4, 3);  
9 INSERT INTO `t1_simple` VALUES (6, 100);  
10 INSERT INTO `t1_simple` VALUES (8, 5);  
11 INSERT INTO `t1_simple` VALUES (10, 1);  
12 INSERT INTO `t1_simple` VALUES (100, 20);
```

2、意向锁 Intention Locks

1) 意向锁介绍

InnoDB也实现了表级锁，也就是意向锁，意向锁是mysql内部使用的，[不需要用户干预](#)。意向锁和行锁可以共存，意向锁的主要作用是为了【全表更新数据】时的性能提升。否则在全表更新数据时，需要先检索该范是否某些记录上面有行锁。

1. 表明“某个事务正在某些行持有了锁、或该事务准备去持有锁”
2. 意向锁的存在是为了协调行锁和表锁的关系，支持多粒度（表锁与行锁）的锁并存，。
3. 例子：事务A修改user表的记录r，会给记录r上一把行级的排他锁（X），同时会给user表上一把意向排他锁（IX），这时事务B要给user表上一个表级的排他锁就会被阻塞。意向锁通过这种方式实现了行锁和表锁共存且满足事务隔离性的要求。
4. 1) 意向共享锁（IS锁）：事务在请求S锁前，要先获得IS锁
2) 意向排他锁（IX锁）：事务在请求X锁前，要先获得IX锁

2) 意向锁的作用

当我们需要加一个排他锁时，需要根据意向锁去判断表中有没有数据行被锁定（行锁）；

- (1) 如果意向锁是行锁，则需要遍历每一行数据去确认；
- (2) 如果意向锁是表锁，则只需要判断一次即可知道有没有数据行被锁定，提升性能。

3) 意向锁和共享锁、排他锁的兼容关系

下图表示意向锁和共享锁、排他锁的兼容关系

是否兼容	当事务A上了：IS	IX	S	X
事务B能否上：IS	是	是	是	否
IX	是	是	否	否
S	是	否	是	否
X	否	否	否	否

<https://blog.csdn.net/u010841296>

意向锁相互兼容，因为IX、IS只是表明申请更低层次级别元素（比如 page、记录）的X、S操作。

因为上了表级S锁后，不允许其他事务再加X锁，所以表级S锁和X、IX锁不兼容

上了表级X锁后，会修改数据，所以表级X锁和 IS、IX、S、X（即使是行排他锁，因为表级锁定的行肯定包括行级速订的行，所以表级X和IX、行级X）不兼容。

注意：上了行级X锁后，行级X锁不会因为有别的事务上了IX而堵塞，一个mysql是允许多个行级X锁同时存在的，只要他们不是针对相同的数据行。

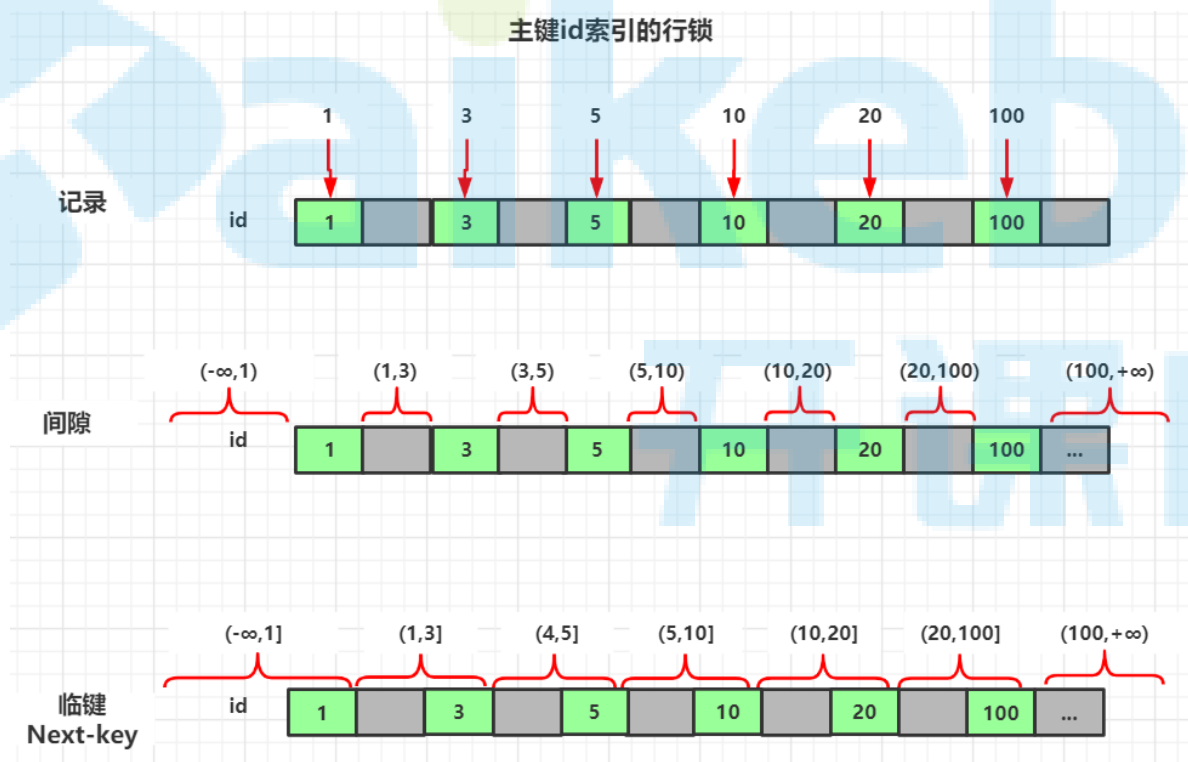
3、记录锁(Record Locks)

(1) 记录锁, 仅仅锁住索引记录的一行, 在单条索引记录上加锁。
(2) record lock锁住的永远是索引, 而非记录本身, 即使该表上没有任何索引, 那么innodb会在后台创建一个隐藏的聚集主键索引, 那么锁住的就是这个隐藏的聚集主键索引。
所以说当一条sql没有走任何索引时, 那么将会在每一条聚合索引后面加X锁, 这个类似于表锁, 但原理上和表锁应该是完全不同的。

```
1  -- 加记录共享锁
2  select * from t1_simple where id = 1 lock in share mode;
3  -- 加记录排它锁
4  select * from t1_simple where id = 1 for update;
```

4、间隙锁(Gap Locks)

(1) 区间锁, 仅仅锁住一个索引区间 (开区间, 不包括双端端点)。
(2) 在索引记录之间的间隙中加锁, 或者是在某一条索引记录之前或者之后加锁, 并不包括该索引记录本身。
(3) 间隙锁可用于防止幻读, 保证索引间的不会被插入数据



```
1  session1:
2  begin;
3  select * from t1_simple where id > 4 for update;
4  -----
5  session2:
6  insert into t1_simple values (7,100); --阻塞
7  insert into t1_simple values (3,100); --成功
```

5、临键锁(Next-Key Locks)

- (1) record lock + gap lock, **左开右闭区间**, 例如 (5,8]。
- (2) 默认情况下, innodb使用next-key locks来锁定记录。select ... for update
- (3) 但当查询的索引含有唯一属性的时候, Next-Key Lock 会进行优化, 将其降级为Record Lock, 即仅锁住索引本身, 不是范围。
- (4) Next-Key Lock在不同的场景中会退化:

场景	退化成的锁类型
使用unique index精确匹配 (=), 且记录存在	Record Lock
使用unique index精确匹配 (=), 且记录不存在	Gap Lock
使用unique index范围匹配 (<和>)	Record Lock + Gap Lock

当前数据库中的记录信息:

```
1  mysql> select * from t1_simple;
2  +-----+-----+
3  | id | pubtime |
4  +-----+-----+
5  | 10 | 1 |
6  | 13 | 1 |
7  | 4 | 3 |
8  | 11 | 4 |
9  | 8 | 5 |
10 | 12 | 9 |
11 | 1 | 10 |
12 | 100 | 20 |
13 | 6 | 100 |
14 +-----+-----+
```

session1执行:

```
1  begin;
2  select * from t1_simple where pubtime = 20 for update;
3  -- 临键锁区间(10,20],(20,100]
```

session2执行:

```
1  insert into t1_simple values (16, 19); --阻塞
2  select * from t1_simple where pubtime = 20 for update; --阻塞
3  insert into t1_simple values (16, 50); --阻塞
4  insert into t1_simple values (16, 101); --成功
```

6、行锁加锁规则

1) 主键索引

1. 等值查询

- (1) 命中记录, 加记录锁。
- (2) 未命中记录, 加间隙锁。

2. 范围查询

- (1) 没有命中任何一条记录时，加间隙锁。
- (2) 命中1条或者多条，包含where条件的临键区间，加临键锁

2) 辅助索引

1. 等值查询

- (1) 命中记录，命中记录的辅助索引项+主键索引项加记录锁，辅助索引项两侧加间隙锁。
- (2) 未命中记录，加间隙锁

2. 范围查询

- (1) 没有命中任何一条记录时，加间隙锁。
- (2) 命中1条或者多条，包含where条件的临键区间加临键锁。命中记录的id索引项加记录锁。

7、插入意向锁(Insert Intention Locks)

- (1) 插入意向锁是一种Gap锁，不是意向锁，在insert操作时产生。
- (2) 在多事务同时写入不同数据至同一索引间隙的时候，并不需要等待其他事务完成，不会发生锁等待。
- (3) 假设有一个记录索引包含键值4和7，不同的事务分别插入5和6，每个事务都会产生一个加在4-7之间的插入意向锁，获取在插入行上的排它锁，但是不会被互相锁住，因为数据行并不冲突。
- (4) 插入意向锁不会阻止任何锁，对于插入的记录会持有一个记录锁。

8、锁相关参数

Innodb所使用的行级锁定争用状态查看：

```
1 mysql> show status like 'innodb_row_lock%';
```

```
mysql> show status like 'innodb_row_lock%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Innodb_row_lock_current_waits | 0 |
| Innodb_row_lock_time | 0 |
| Innodb_row_lock_time_avg | 0 |
| Innodb_row_lock_time_max | 0 |
| Innodb_row_lock_waits | 0 |
+-----+-----+
5 rows in set (0.00 sec)
```

- 1 - Innodb_row_lock_current_waits: 当前正在等待锁定的数量;
- 2
- 3 - Innodb_row_lock_time: 从系统启动到现在锁定总时间长度;
- 4
- 5 - Innodb_row_lock_time_avg: 每次等待所花平均时间;
- 6
- 7 - Innodb_row_lock_time_max: 从系统启动到现在等待最常的一次所花的时间;
- 8
- 9 - Innodb_row_lock_waits: 系统启动后到现在总共等待的次数;

对于这5个状态变量，比较重要的主要是：

```
1 - Innodb_row_lock_time_avg（等待平均时长）
2
3 - Innodb_row_lock_waits（等待总次数）
4
5 - Innodb_row_lock_time（等待总时长）这三项。
```

尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手指定优化计划。

查看事务、锁的sql:

```
1 select * from information_schema.innodb_locks;
2 select * from information_schema.innodb_lock_waits;
3 select * from information_schema.innodb_trx;
```

六、行锁分析实战

在介绍完一些背景知识之后，接下来将选择几个有代表性的例子，来详细分析MySQL的加锁处理。当然，还是从最简单的例子说起。经常有朋友发给我一个SQL，然后问我，这个SQL加什么锁？就如同下面两条简单的SQL，他们加什么锁？

- SQL1:

```
1 select * from t1 where id = 10;
```

- SQL2:

```
1 delete from t1 where id = 10;
```

针对这个问题，该怎么回答？能想象到的一个答案是：

- SQL1：不加锁。因为MySQL是使用多版本并发控制的，读不加锁。
- SQL2：对id = 10的记录加写锁（走主键索引）。

这个答案对吗？说不上来。即可能是正确的，也有可能是错误的，已知条件不足，这个问题没有答案。必须还要知道以下的一些前提，前提不同，能给出的答案也就不同。要回答这个问题，还缺少哪些前提条件？

- 前提一：id列是不是主键？
- 前提二：当前系统的隔离级别是什么？
- 前提三：id列如果不是主键，那么id列上有索引吗？
- 前提四：id列上如果有二级索引，那么这个索引是唯一索引吗？
- 前提五：两个SQL的执行计划是什么？索引扫描？全表扫描？

没有这些前提，直接就给定一条SQL，然后问这个SQL会加什么锁，都是很业余的表现。而当这些问题有了明确的答案之后，给定的SQL会加什么锁，也就一目了然。下面，我们将这些问题的答案进行组合，然后按照从易到难的顺序，逐个分析每种组合下，对应的SQL会加哪些锁？

注：下面的这些组合，需要做一个前提假设，也就是有索引时，执行计划一定会选择使用索引进行过滤（索引扫描）。但实际情况会复杂很多，真正的执行计划，还是需要根据MySQL输出的为准。

```
1 组合一：id列是主键，RC隔离级别
2
```

3 组合二: id列是二级唯一索引, RC隔离级别
4
5 组合三: id列是二级非唯一索引, RC隔离级别
6
7 组合四: id列上没有索引, RC隔离级别
8
9 组合五: id列是主键, RR隔离级别
10
11 组合六: id列是二级唯一索引, RR隔离级别
12
13 组合七: id列是二级非唯一索引, RR隔离级别
14
15 组合八: id列上没有索引, RR隔离级别
16
17 组合九: Serializable隔离级别

排列组合还没有列举完全, 但是看起来, 已经很多了。真的有必要这么复杂吗? 事实上, 要分析加锁, 就是需要这么复杂。但是从另一个角度来说, 只要你选定了一种组合, SQL需要加哪些锁, 其实也就确定了。接下来, 就让我们来逐个分析这9种组合下的SQL加锁策略。

注: 在前面八种组合下, 也就是RC, RR隔离级别下SQL1: [select操作均不加锁, 采用的是快照读](#), 因此在下面的讨论中就忽略了, [主要讨论SQL2: delete操作的加锁](#)。

1) 组合一: id主键+RC

这个组合, 是最简单, 最容易分析的组合。id是主键, Read Committed隔离级别, 给定SQL: [delete from t1 where id = 10](#); 只需要将主键上id = 10的记录加上X锁即可。如下图所示:

Table: T1(id primary key, name)

Primary Key

X锁

id	1	4	7	10	20	30
name	a	c	b	a	d	b

结论:

1 | id是主键时，此SQL只需要在id=10这条记录上加X锁即可。

2) 组合二: id唯一索引+RC

这个组合, id不是主键, 而是一个Unique的二级索引键值。那么在RC隔离级别下, delete from t1 where id = 10; 需要加什么锁呢? 见下图:

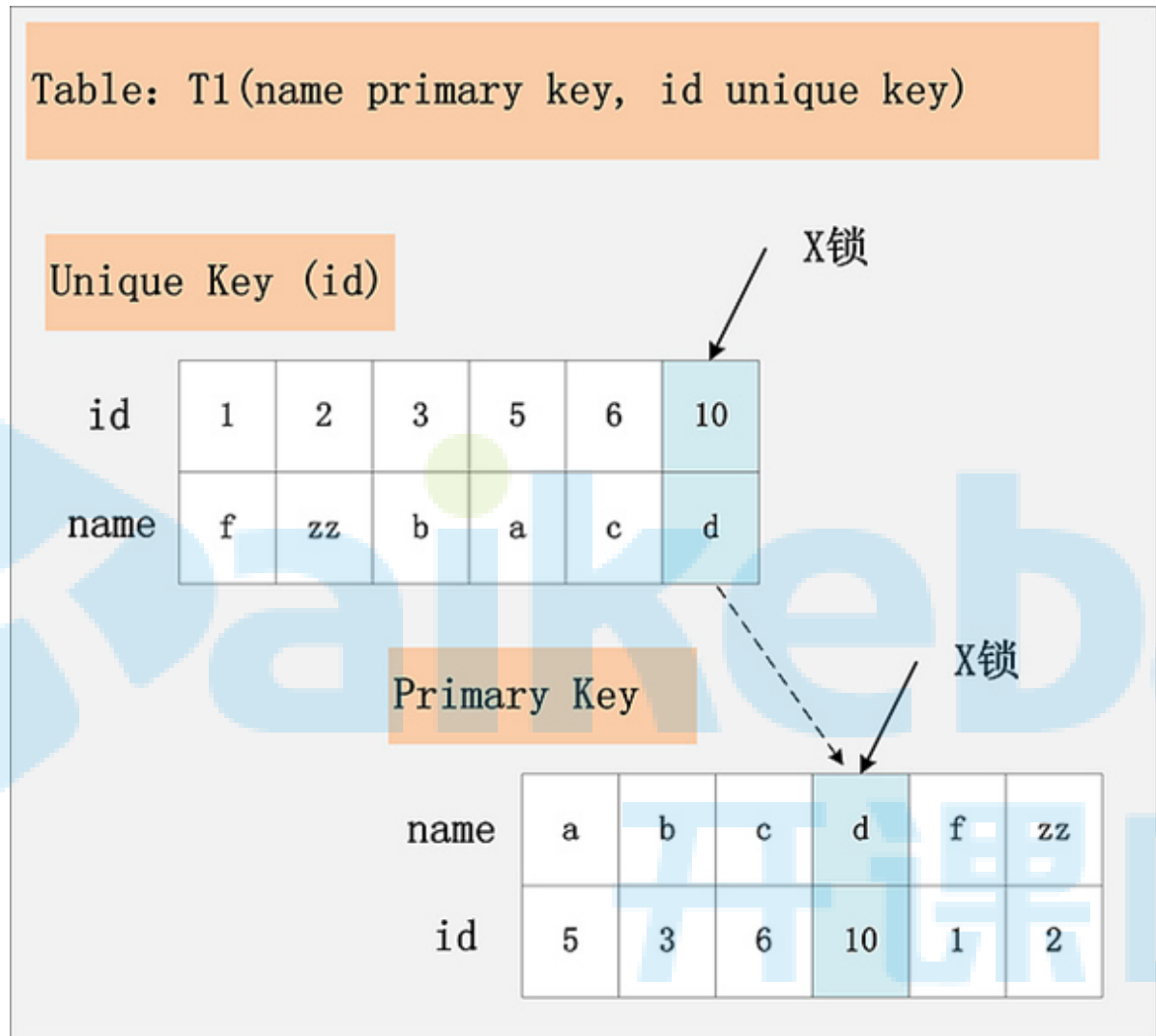


Table: T1(name primary key, id unique key)

Unique Key (id)

id	1	2	3	5	6	10
name	f	zz	b	a	c	d

X锁

首先在次要索引中，找到符合条件的记录，加X锁

Primary Key

在次要索引中找到符合条件的记录之后，接着取出主键，然后去主键索引中查找记录，找到也加X锁。

name	a	b	c	d	f	zz
id	5	3	6	10	1	2

X锁

此组合中，id是unique索引，而主键是name列。此时，加锁的情况由于组合一有所不同。由于id是unique索引，因此delete语句会选择走id列的索引进行where条件的过滤，在找到id=10的记录后，首先会将unique索引上的id=10索引记录加上X锁，同时，会根据读取到的name列，回主键索引(聚簇索引)，然后将聚簇索引上的name = 'd' 对应的主键索引项加X锁。为什么聚簇索引上的记录也要加锁？试想一下，如果并发的一个SQL，是通过主键索引来更新：update t1 set id = 100 where name = 'd'; 此时，如果delete语句没有将主键索引上的记录加锁，那么并发的update就会感知不到delete语句的存在，违背了同一记录上的更新/删除需要串行执行的约束。

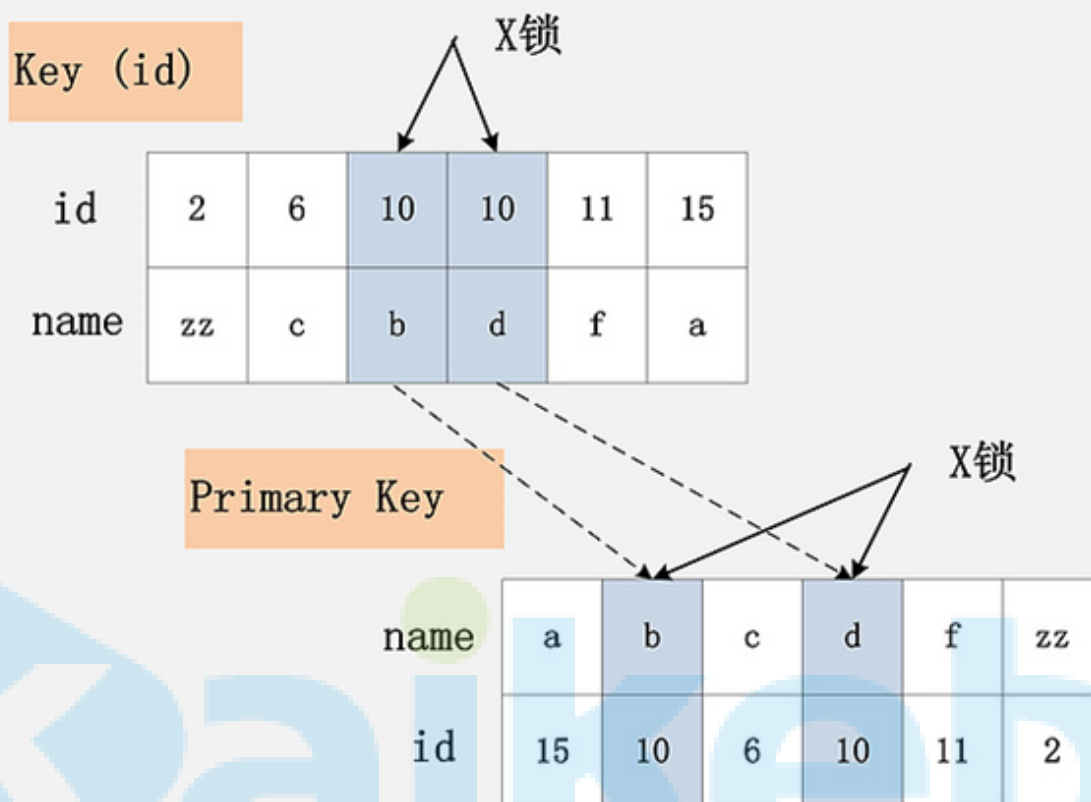
结论：

- 1 若id列是unique列，其上有unique索引。那么SQL需要加两个X锁，一个对应于id unique索引上的id = 10的记录，另一把锁对应于聚簇索引上的【name='d', id=10】的记录。

3) 组合三：id非唯一索引+RC

相对于组合一、二，组合三又发生了变化，隔离级别仍旧是RC不变，但是id列上的约束又降低了，id列不再唯一，只有一个普通的索引。假设delete from t1 where id = 10; 语句，仍旧选择id列上的索引进行过滤where条件，那么此时会持有哪些锁？同样见下图：

Table: T1(name primary key, id key)



根据此图，可以看到，首先，id列索引上，满足id = 10查询条件的记录，均已加锁。同时，这些记录对应的主键索引上的记录也都加上了锁。与组合二唯一的区别在于，组合二最多只有一个满足等值查询的记录，而组合三会将所有满足查询条件的记录都加锁。

结论：

- 1 若id列上有非唯一索引，那么对应的所有满足SQL查询条件的记录，都会被加锁。同时，这些记录在主键索引上的记录，也会被加锁。

4) 组合四：id无索引+RC

相对于前面三个组合，这是一个比较特殊的情况。id列上没有索引，`where id = 10`这个过滤条件，没法通过索引进行过滤，那么只能走全表扫描做过滤。

对应于这个组合，SQL会加什么锁？或者是换句话说，全表扫描时，会加什么锁？这个答案也有很多：有人说会在表上加X锁；有人说会将聚簇索引上，选择出来的id = 10;的记录加上X锁。那么实际情况呢？请看下图：

Table: T1(name primary key, id)

Primary Key

X锁

name	a	b	d	f	g	zz
id	5	3	10	2	10	9

由于id列上没有索引，因此只能走聚簇索引，进行全部扫描。从图中可以看到，满足删除条件的记录有两条，但是，聚簇索引上所有的记录，都被加上了X锁。无论记录是否满足条件，全部被加上X锁。既不是加表锁，也不是在满足条件的记录上加行锁。

有人可能会问？为什么不是只在满足条件的记录上加锁呢？这是由于MySQL的实现决定的。如果一个条件无法通过索引快速过滤，那么存储引擎层面就会将所有记录加锁后返回，然后由MySQL Server层进行过滤。因此也就把所有的记录，都锁上了。

注：在实际的实现中，MySQL有一些改进，在MySQL Server过滤条件，发现不满足后，会调用unlock_row方法，把不满足条件的记录放锁（违背了2PL的约束）。这样做，保证了最后只会持有满足条件记录上的锁，但是每条记录的加锁操作还是不能省略的。

结论：

- 1 若id列上没有索引，SQL会走聚簇索引的全扫描进行过滤，由于过滤是由MySQL Server层面进行的。因此每条记录，无论是否满足条件，都会被加上X锁。但是，为了效率考量，MySQL做了优化，对于不满足条件的记录，会在判断后放锁，最终持有的，是满足条件的记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略。同时，优化也违背了2PL的约束。

5) 组合五：id主键+RR

上面的四个组合，都是在Read Committed隔离级别下的加锁行为，接下来的四个组合，是在Repeatable Read隔离级别下的加锁行为。

组合五, [id列是主键列](#), [Repeatable Read](#)隔离级别, 针对`delete from t1 where id = 10;`这条SQL, 加锁与组合一: [\[id主键, Read Committed\]](#)一致。

6) 组合六: id唯一索引+RR

与组合五类似, 组合六的加锁, 与组合二: [\[id唯一索引, Read Committed\]](#)一致。两个X锁, id唯一索引满足条件的记录上一个, 对应的聚簇索引上的记录一个。

7) 组合七: id非唯一索引+RR

还记得前面提到的MySQL的四种隔离级别的区别吗? [RC隔离级别允许幻读, 而RR隔离级别, 不允许存在幻读](#)。但是在组合五、组合六中, 加锁行为又是与RC下的加锁行为完全一致。那么RR隔离级别下, 如何防止幻读呢? 问题的答案, 就在组合七中揭晓。

组合七, [Repeatable Read](#)隔离级别, [id上有一个非唯一索引](#), 执行`delete from t1 where id = 10;`假设选择id列上的索引进行条件过滤, 最后的加锁行为, 是怎样的呢? 同样看下面这幅图:

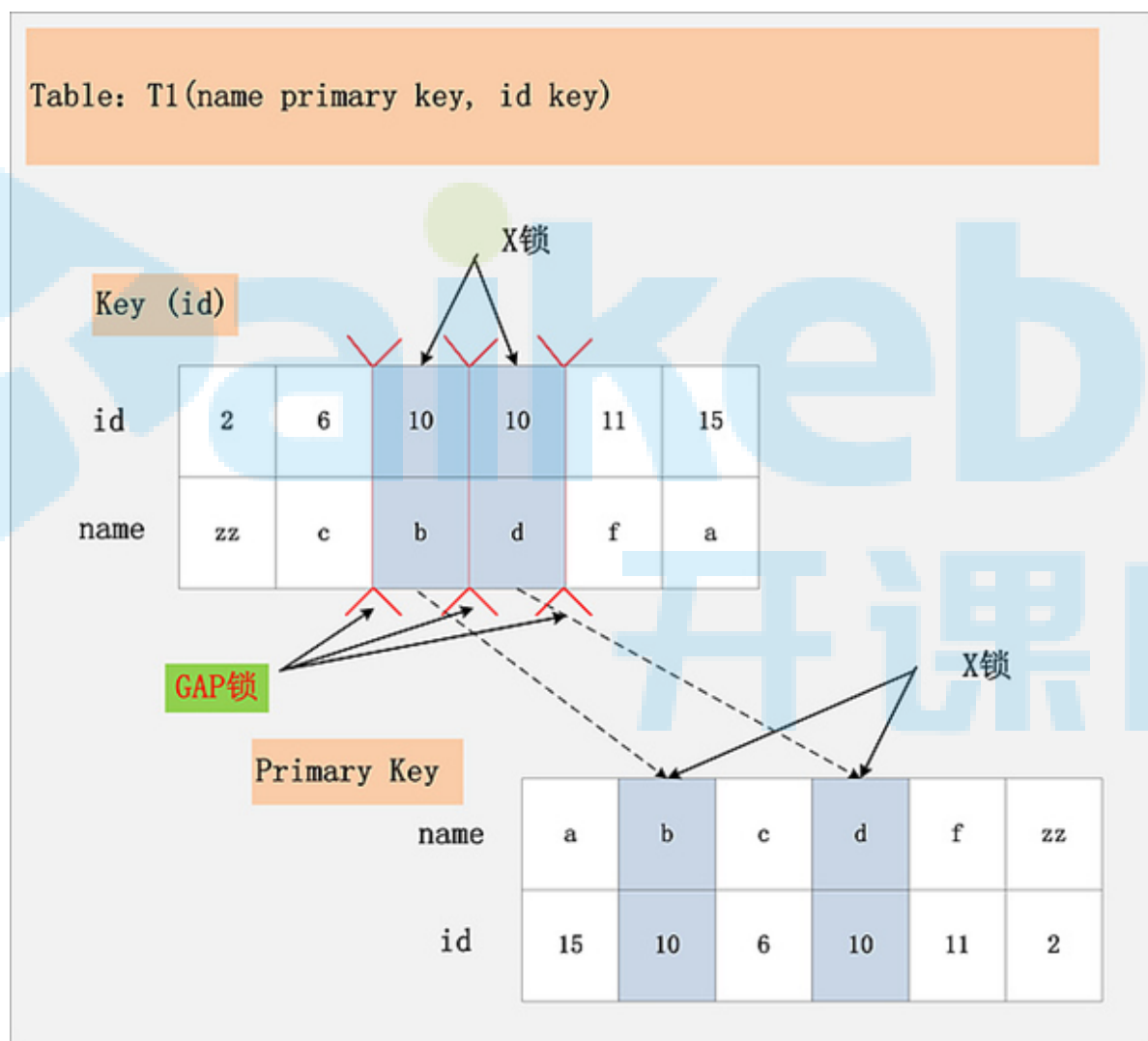
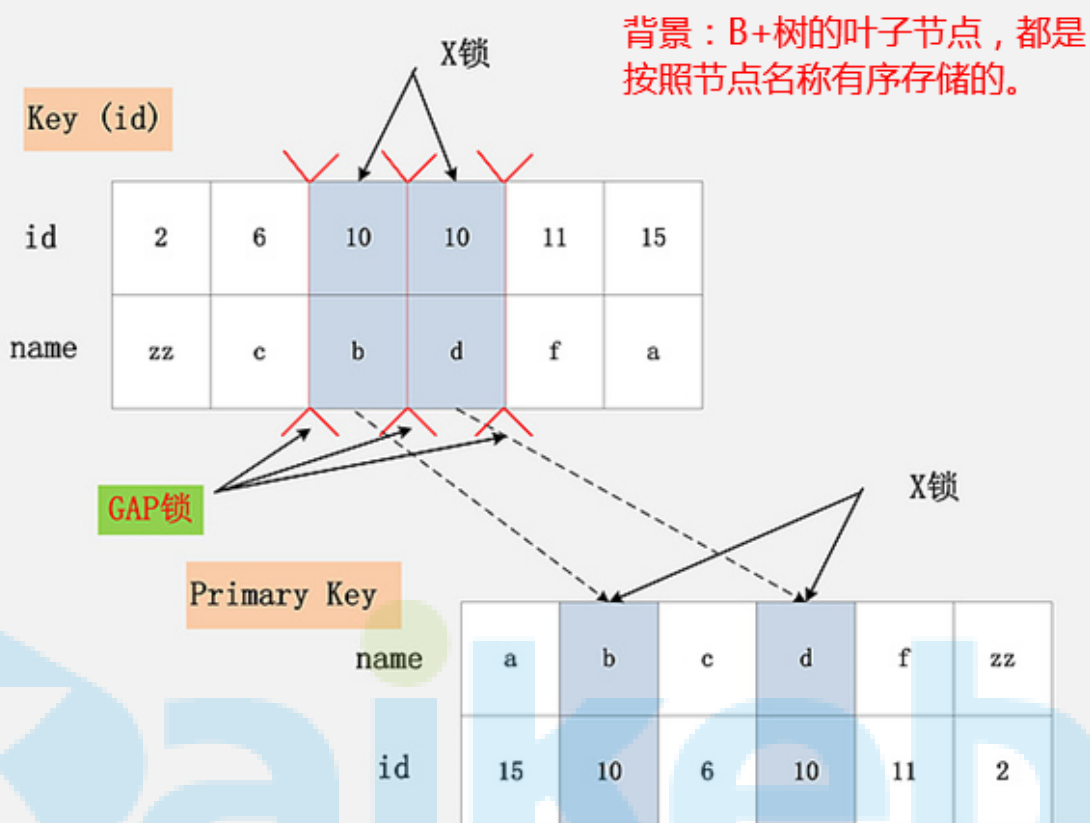


Table: T1(name primary key, id key)



此图，相对于组合三：[\[id列上非唯一锁, Read Committed\]](#)看似相同，其实却有很大的区别。最大的区别在于，这幅图中多了一个GAP锁，而且GAP锁看起来也不是加在记录上的，倒像是加载两条记录之间的位置，GAP锁有何用？

[其实这个多出来的GAP锁，就是RR隔离级别，相对于RC隔离级别，不会出现幻读的关键。](#)确实，GAP锁锁住的位置，也不是记录本身，而是两条记录之间的GAP。所谓幻读，就是同一个事务，连续做两次当前读（例如：`select * from t1 where id = 10 for update;`），那么这两次当前读返回的是完全相同的记录（记录数量一致，记录本身也一致），第二次的当前读，不会比第一次返回更多的记录（幻象）。

如何保证两次当前读返回一致的记录，那就需要在第一次当前读与第二次当前读之间，其他的事务不会插入新的满足条件的记录并提交。为了实现这个功能，GAP锁应运而生。

如图中所示，有哪些位置可以插入新的满足条件的项（`id = 10`），考虑到B+树索引的有序性，满足条件的项一定是连续存放的。记录[6,c]之前，不会插入`id=10`的记录；[6,c]与[10,b]间可以插入[10,aa]；[10,b]与[10,d]间，可以插入新的[10,bb],[10,c]等；[10,d]与[11,f]间可以插入满足条件的[10,e],[10,z]等；而[11,f]之后也不会插入满足条件的记录。因此，为了保证[6,c]与[10,b]间，[10,b]与[10,d]间，[10,d]与[11,f]不会插入新的满足条件的记录，MySQL选择了用GAP锁，将这三个GAP给锁起来。

Insert操作，如`insert [10,aa]`，首先会定位到[6,c]与[10,b]间，然后在插入前，会检查这个GAP是否已经被锁上，如果被锁上，则Insert不能插入记录。因此，通过第一遍的当前读，不仅将满足条件的记录锁上（X锁），与组合三类似。同时还是增加3把GAP锁，将可能插入满足条件记录的3个GAP给锁上，保证后续的Insert不能插入新的`id=10`的记录，也就杜绝了同一事务的第二次当前读，出现幻象的情况。

有心的朋友看到这儿，可以会问：[既然防止幻读，需要靠GAP锁的保护，为什么组合五、组合六，也是RR隔离级别，却不需要加GAP锁呢？](#)

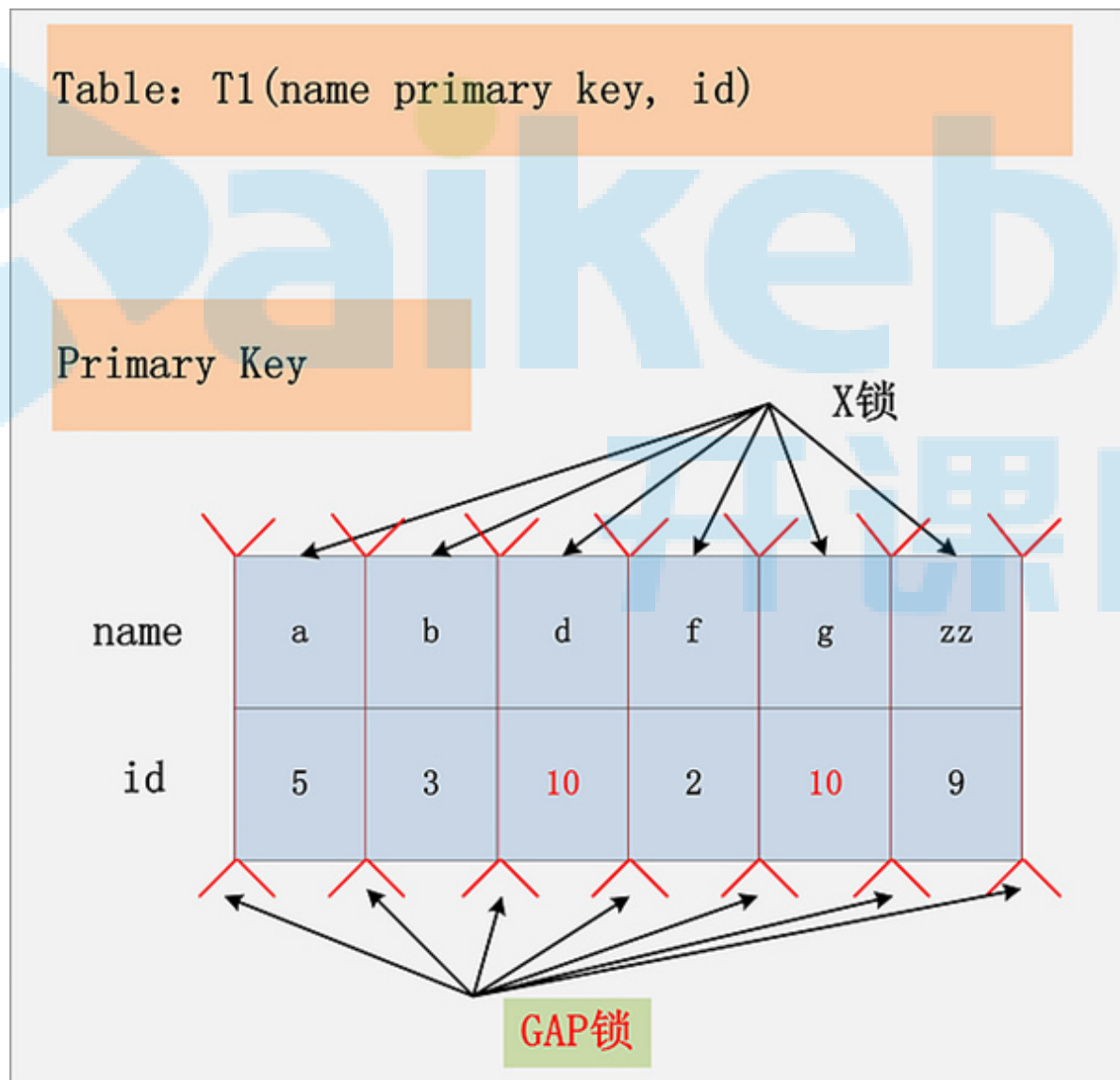
首先，这是一个好问题。其次，回答这个问题，也很简单。GAP锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况。而组合五，id是主键；组合六，id是unique键，都能够保证唯一性。一个等值查询，最多只能返回一条记录，而且新的相同取值的记录，一定不会在新插入进来，因此也就避免了GAP锁的使用。其实，针对此问题，还有一个更深入的问题：如果组合五、组合六下，针对SQL: [select * from t1 where id = 10 for update](#); 第一次查询，没有找到满足查询条件的记录，那么GAP锁是否还能够省略？此问题留给大家思考。

结论：

- 1 Repeatable Read隔离级别下，id列上有一个非唯一索引，对应SQL: `delete from t1 where id = 10`；首先，通过id索引定位到第一条满足查询条件的记录，加记录上的X锁，加GAP上的GAP锁，然后加主键聚簇索引上的记录X锁，然后返回；然后读取下一条，重复进行。直至进行到第一条不满足条件的记录[11, f]，此时，不需要加记录X锁，但是仍旧需要加GAP锁，最后返回结束。

8) 组合八：id无索引+RR

组合八，Repeatable Read隔离级别下的最后一种情况，[id列上没有索引](#)。此时SQL: [delete from t1 where id = 10](#)；没有其他的路径可以选择，只能进行全表扫描。最终的加锁情况，如下图所示：



如图，这是一个很恐怖的现象。首先，聚簇索引上的所有记录，都被加上了X锁。其次，聚簇索引每条记录间的间隙(GAP)，也同时被加上了GAP锁。这个示例表，只有6条记录，一共需要6个记录锁，7个GAP锁。试想，如果表上有1000万条记录呢？

在这种情况下，这个表上，除了不加锁的快照读，其他任何加锁的并发SQL，均不能执行，不能更新，不能删除，不能插入，全表被锁死。

当然，跟组合四：[\[id无索引, Read Committed\]](#)类似，这个情况下，MySQL也做了一些优化，就是所谓的semi-consistent read。semi-consistent read开启的情况下，对于不满足查询条件的记录，MySQL会提前放锁。针对上面的这个用例，就是除了记录[d,10]，[g,10]之外，所有的记录锁都会被释放，同时不加GAP锁。semi-consistent read如何触发：要么是read committed隔离级别；要么是Repeatable Read隔离级别，同时设置了 [innodb locks unsafe for binlog](#) 参数。

结论：

- 1 在Repeatable Read隔离级别下，如果进行全表扫描的当前读，那么会锁上表中的所有记录，同时会锁上聚簇索引内的所有GAP，杜绝所有的并发 更新/删除/插入 操作。当然，也可以通过触发semi-consistent read，来缓解加锁开销与并发影响，但是semi-consistent read本身也会带来其他问题，不建议使用。

9) 组合九：Serializable

针对前面提到的简单的SQL，最后一个情况：Serializable隔离级别。对于SQL2来说，Serializable隔离级别与Repeatable Read隔离级别完全一致，因此不做介绍。

```
1 delete from t1 where id = 10
```

Serializable隔离级别，影响的是SQL1这条SQL：

```
1 select * from t1 where id = 10
```

在RC，RR隔离级别下，都是快照读，不加锁。但是在Serializable隔离级别，SQL1会加读锁，也就是说快照读不复存在，[MVCC并发控制降级为Lock-Based CC](#)。

结论：

- 1 在MySQL/InnoDB中，所谓的读不加锁，并不适用于所有的情况，而是隔离级别相关的。Serializable隔离级别，读不加锁就不再成立，所有的读操作，都是当前读。

七、一条复杂SQL的加锁分析

写到这里，其实MySQL的加锁实现也已经介绍的八八九九。只要将本文上面的分析思路，大部分的SQL，都能分析出其会加哪些锁。而这里，再来看一个稍微复杂点的SQL，用于说明MySQL加锁的另外一个逻辑。SQL用例如下：

Table: t1(id primary key, userid, blogid, pubtime, comment)
Index: idx_t1_pu(pubtime, userid)

idx_t1_pu

pubtime	1	3	5	10	20	100
userid	hdc	yyy	hdc	hdc	bbb	hdc
id	10	4	8	1	100	6

Primary Key

id	1	4	6	8	10	100
userid	hdc	yyy	hdc	hdc	hdc	bbb
blogid	a	b	c	d	e	f
pubtime	10	3	100	5	1	20
comment				good		

SQL: delete from t1 where pubtime > 1 and pubtime < 20 and userid = 'hdc' and comment is not NULL;

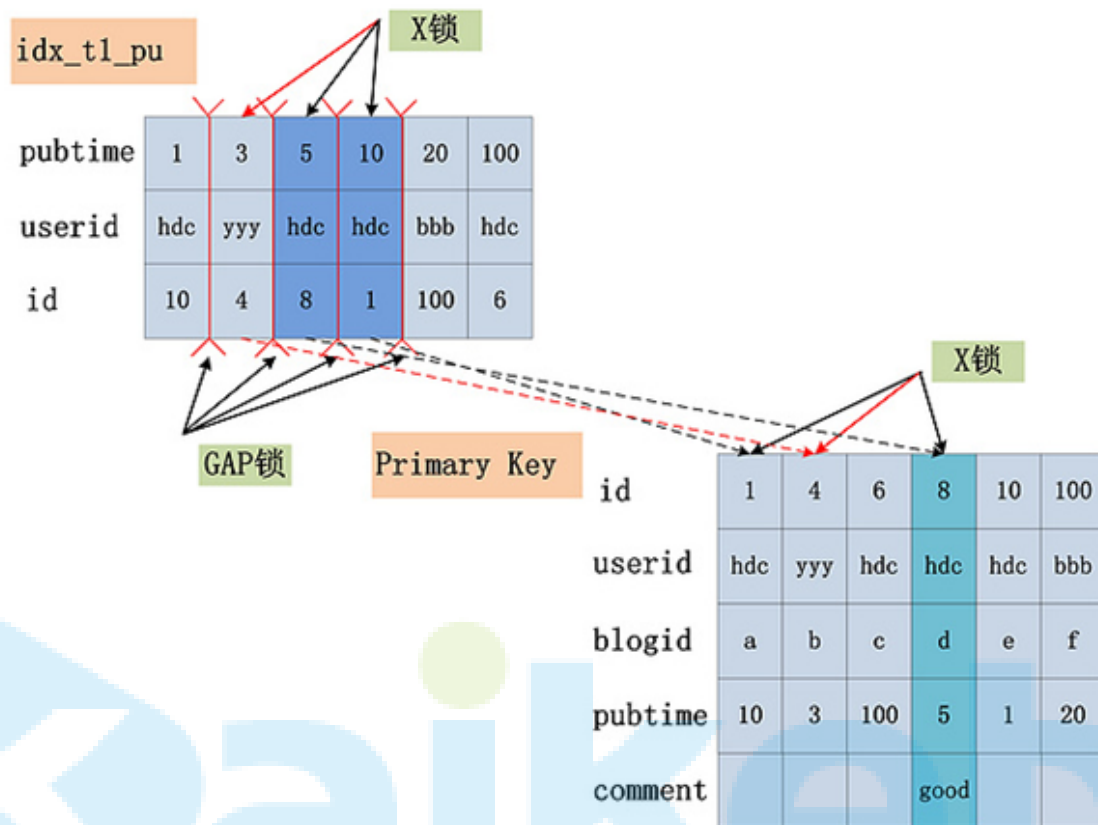
如图中的SQL，会加什么锁？假定在Repeatable Read隔离级别下 (Read Committed隔离级别下的加锁情况，留给学生们分析)，同时，假设SQL走的是idx_t1_pu索引。

在详细分析这条SQL的加锁情况前，还需要有一个知识储备，那就是一个SQL中的where条件如何拆分？在这里，我直接给出分析后的结果：

- **Index key:** pubtime > 1 and pubtime < 20。此条件，用于确定SQL在idx_t1_pu索引上的查询范围。
- **Index Filter:** userid = 'hdc'。此条件，可以在idx_t1_pu索引上进行过滤，但不属于Index Key。
- **Table Filter:** comment is not NULL。此条件，在idx_t1_pu索引上无法过滤，只能在聚簇索引上过滤。

在分析出SQL where条件的构成之后，再看看这条SQL的加锁情况 (RR隔离级别)，如下图所示：

Table: t1(id primary key, userid, blogid, pubtime, comment)
Index: idx_t1_pu(pubtime, userid)



SQL: delete from t1 where pubtime > 1 and pubtime < 20 and userid = 'hdc' and comment is not NULL;

从图中可以看出，在Repeatable Read隔离级别下，由Index Key所确定的范围，被加上了GAP锁；Index Filter锁给定的条件 (userid = 'hdc')何时过滤，视MySQL的版本而定，在MySQL 5.6版本之前，不支持Index Condition Pushdown(ICP)，因此Index Filter在MySQL Server层过滤，在5.6后支持了Index Condition Pushdown，则在index上过滤。若不支持ICP，不满足Index Filter的记录，也需要加上记录X锁；若支持ICP，则不满足Index Filter的记录，无需加记录X锁 (图中，用红色箭头标出的X锁，是否要加，视是否支持ICP而定)；而Table Filter对应的过滤条件，则在聚簇索引中读取后，在MySQL Server层面过滤，因此聚簇索引上也需要X锁。最后，选取出了一条满足条件的记录[8,hdc,d,5,good]，但是加锁的数量，要远远大于满足条件的记录数量。

结论：

- 1 在Repeatable Read隔离级别下，针对一个复杂的SQL，首先需要提取其where条件。
- 2 - Index Key确定的范围，需要加上GAP锁；
- 3 - Index Filter过滤条件，视MySQL版本是否支持ICP，若支持ICP，则不满足Index Filter的记录，不加X锁，否则需要X锁；
- 4 - Table Filter过滤条件，无论是否满足，都需要加X锁。

八、死锁原理与分析

本文前面的部分，基本上已经涵盖了MySQL/InnoDB所有的加锁规则。深入理解MySQL如何加锁，有两个比较重要的作用：

- 可以根据MySQL的加锁规则，写出不会发生死锁的SQL；

- 可以根据MySQL的加锁规则，定位出线上产生死锁的原因；

下面，来看看两个死锁的例子 ([一个是两个Session的两条SQL产生死锁](#)；[另一个是两个Session的一条SQL，产生死锁](#))：

死锁情况一

Table: T1(id primary key, name)

```

session 1
begin;
select * from t1 where id = 1 for update;

update t1 set name=' qq' where id = 5;

```

```

session 2
begin;

delete from t1 where id = 5;

delete from t1 where id = 1;

```

死锁发生!!!

id

1	2	3	4	5	6
aaa	ccc	aaa	bbb	ccc	zzz

死锁情况二

Table: T2(id primary key, name key, pubtime key, comment)

```

session 1
update t2 set comment=' abc' where name=' hdc' ;

```

```

session 2
select * from t2 where pubtime > 5 for update;

```

key(name)

name	bbb	hdc	hdc	hdc	hdc	yyy
id	100	1	6	8	10	4

key(pubtime)

pubtime	1	3	5	10	20	100
id	10	4	8	6	100	1

primary key

id	1	4	6	8	10	100
name	hdc	yyy	hdc	hdc	hdc	bbb
pubtime	100	3	10	5	1	20
comment				good		

Deadlock!!

上面的两个死锁用例。第一个非常好理解，也是最常见的死锁，每个事务执行两条SQL，分别持有了一把锁，然后加另一把锁，产生死锁。

第二个用例，虽然每个Session都只有一条语句，仍旧会产生死锁。要分析这个死锁，首先必须用到本文前面提到的MySQL加锁的规则。[针对Session 1](#)，从name索引出发，读到的[hdc, 1], [hdc, 6]均满足条件，不仅会加name索引上的记录X锁，而且会加聚簇索引上的记录X锁，[加锁顺序为先\[1,hdc,100\]，后\[6,hdc,10\]](#)。而[Session 2](#)，从pubtime索引出发，[10,6],[100,1]均满足过滤条件，同样也会加聚簇索引上的记录X锁，[加锁顺序为\[6,hdc,10\]，后\[1,hdc,100\]](#)。发现没有，跟Session 1的加锁顺序正好相反，如果两个Session恰好都持有了第一把锁，请求加第二把锁，死锁就发生了。

结论：

- 1 死锁的发生与否，并不在于事务中有多少条SQL语句，【死锁的关键在于】：两个(或以上)的Session【加锁的顺序】不一致。而使用本文上面提到的，分析MySQL每条SQL语句的加锁规则，分析出每条语句的加锁顺序，然后检查多个并发SQL间是否存在以相反的顺序加锁的情况，就可以分析出各种潜在的死锁情况，也可以分析出线上死锁发生的原因。

如何避免死锁呢？

MySQL默认会主动探知死锁，并回滚某一个影响最小的事务。等另一事务执行完成之后，再重新执行该事务。

如何避免死锁

1、注意程序的逻辑

根本的原因是程序逻辑的顺序，最常见的是交差更新

Transaction 1: 更新表A -> 更新表B

Transaction 2: 更新表B -> 更新表A

Transaction获得两个资源

2、保持事务的轻量

越是轻量的事务，占有越少的锁资源，这样发生死锁的几率就越小

3、提高运行的速度

避免使用子查询，尽量使用主键等等

4、尽量快提交事务，减少持有锁的时间

越早提交事务，锁就越早释放