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：

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

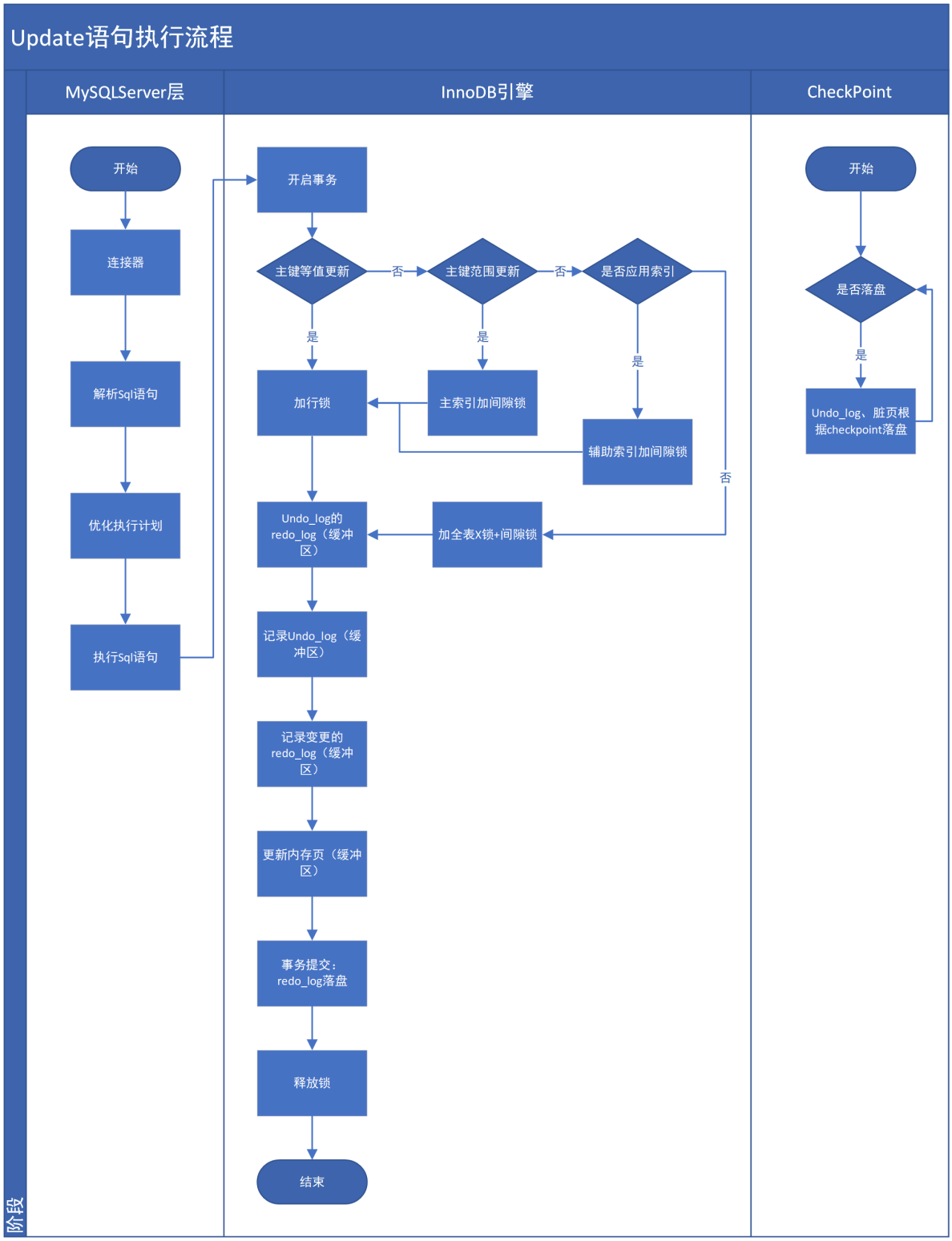
现在表里的数据就是这样的：

```
mysql> SELECT * FROM t;
+----+--------+
| id | c      |
+----+--------+
|  1 |  刘备   |
+----+--------+
1 row in set (0.01 sec)
```

然后更新表里的一条数据：

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

执行流程：



二、MySQL锁介绍

在实际的数据库系统中，每时每刻都在发生锁定，当某个用户在修改一部分数据时，MySQL会通过锁定防止其他用户读取同一数据。

在处理并发读或者写时，通过实现一个由两种类型的锁组成的锁系统来解决问题。两种锁通常被称为**共享锁(shared lock)**和**排他锁(exclusive lock)**，也叫**读锁(read lock)**和**写锁(write lock)**。

读锁是共享的，是互相不阻塞的。多个客户端在同一时刻可以同时读取同一个资源，而不互相干扰。写锁则是排他的，也就是说一个写锁会阻塞其他的写锁和读锁，这是出于安全策略的考虑，只有这样才能确保在给定的时间里，只有一个用户能执行写入，并防止其他用户读取正在写入的同一资源。

2.1 锁分类

按锁粒度分：

- 全局锁：锁整database，由MySQL的SQL layer层实现的。
- 表级锁：锁某table，由MySQL的SQL layer层实现的。
- 行级锁：锁某行数据，也可锁定行之间的间隙，由某些存储引擎实现【InnoDB】

按锁功能分：

- **共享锁Shared Locks（S锁，也叫读锁）：** 为了方便理解，下文我们全部使用**读锁**来称呼
  - 加了读锁的记录，允许其他事务再加读锁
  - 加锁方式：select...lock in share mode
- **排他锁Exclusive Locks（X锁，也叫写锁）：**为了方便理解，下文我们全部使用**写锁**来称呼
  - 加了写锁的记录，不允许其他事务再加读锁或者写锁
  - 加锁方式：select...for update

2.2 什么是全局锁?

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：

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

释放全局锁的命令为：

```
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。

三、表级锁

3.1 什么是表级锁?

MySQL的表级锁有四种：

- 表读
- 写锁。
- 元数据锁（meta data lock，MDL)。
- 自增锁(AUTO-INC Locks)

3.2 表读锁、写锁

1）表锁相关命令

MySQL 实现的表级锁定的争用状态变量：

```
# 查看表锁定状态
mysql> show status like 'table%';
```

```
mysql> show status like 'table%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Table_locks_immediate      | 99    |
| Table_locks_waited         | 0     |
| Table_open_cache_hits      | 0     |
| Table_open_cache_misses    | 0     |
| Table_open_cache_overflows | 0     |
+----------------------------+-------+
5 rows in set (0.01 sec)
```

- table_locks_immediate：产生表级锁定的次数；
- 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：

- 表读锁（Table Read Lock）
- 表写锁（Table Write Lock）

手动增加表锁：

```
lock table 表名称 read(write),表名称2 read(write)，其他;
# 举例：
lock table t read; #为表t加读锁
lock table t write; #为表t加写锁
```

查看表锁情况：

```
show open tables;
```

删除表锁：

```
unlock tables;
```

2）表锁演示

1. 环境准备

```
CREATE TABLE mylock (
    id int(11) NOT NULL AUTO_INCREMENT,
    NAME varchar(20) DEFAULT NULL,
    PRIMARY KEY (id)
);

INSERT INTO mylock (id,NAME) VALUES (1, 'a');
INSERT INTO mylock (id,NAME) VALUES (2, 'b');
INSERT INTO mylock (id,NAME) VALUES (3, 'c');
INSERT INTO mylock (id,NAME) VALUES (4, 'd');
```

2. 读锁演示：mylock表加read锁【读阻塞写】

时间	session01	session02
T1	连接MySQL	
T2	获得表mylock的Read Lock锁定：`lock table mylock read;`	连接MySQL
T3	当前Session可以查询该表记录：`select * from mylock;`	其他Session也可以查询该表的记录：`select * from mylock;`
T4	当前Session不能查询其他没有锁定的表：`select * from t;`	其他Session可以查询或更新未锁定的表：`update t set c='张飞' where id=1`
T5	当前Session插入或更新**锁定的表**会提示错误：`insert into mylock (name) values('e');`	其他Session插入或更新锁定表**会一直等待**获取锁：`insert into mylock (name) values('e');`
T6	释放锁：`unlock tables;`	插入成功：

3. 写锁演示：mylock表加write锁【写阻塞读】

时间	session01	session02
T1	连接MySQL	待session1开启锁后，session2再获取连接
T2	获得表mylock的write锁：`lock table mylock write;`	
T3	当前session对锁定表的查询+更新+插入操作都可以执行：`select * from mylock where id=1;`	连接MySQL
T4		其他session对锁定表的查询被阻塞，需要等待锁被释放 `select * from mylock where id=1;`
T5	释放锁：`unlock tables;`	获得锁，返回查询结果：

> 注意：mysql有缓存，如果在查询过程中没有被阻塞说明查询到的是缓存中的数据。

3.3 元数据锁

1）元数据锁介绍

元数据锁不需要显式使用，在访问一个表的时候会被自动加上。锁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 元数据锁**，当对一个表做增删改查操作的时候，加 元数据 读锁；当要对表做结构变更操作的时候，加 元数据 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加读锁，保证数据在读取的过程中不会被其他线程修改。
- 写锁之间，写锁与读锁之间是互斥的，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

时间	session01	session02
T1	开启事务：begin	
T2	加元数据读锁：`select * from mylock;`	修改表结构：`alter table mylock add f int;`
T3	提交/回滚事务：`commit/rollback` 释放锁	
T4		获取锁，修改完成

3.4 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。

四、行级锁

4.1 什么是行级锁?

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：
  - 记录锁（Record Locks）:锁定索引中一条记录。
  - 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。
  - 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）。
  - 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：
  - 读锁：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。
  - 写锁：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。

如何加行级锁?

- **对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加写锁**;
- **对于普通SELECT语句，InnoDB不会加任何锁**

事务可以通过以下语句手动给记录集加共享锁或排他锁。

添加读锁:

```
SELECT * FROM t1_simple WHERE id=4 LOCK IN SHARE MODE;
```

添加写锁:

```
SELECT * FROM t1_simple WHERE id=4 FOR UPDATE;
```

案例:

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

4.2 行锁四兄弟：意向、记录、间隙和临键锁

4.2.1 意向锁

1）什么是意向锁?

InnoDB也实现了表级锁，也就是意向锁【Intention Locks】。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**全表更新数据时的提升性能。**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。

举个栗子：

事务A修改user表的记录r，会给记录r上一把行级的**写锁**，同时会给user表上一把**意向写锁（IX）**，这时事务B要给user表上一个表级的**写锁**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。

2）作用

- 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"
- 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

举个栗子：

当我们需要加一个写锁时，需要根据意向锁去判断表中有没有数据行被锁定；

（1）如果行锁，则需要遍历每一行数据去确认；

（2）如果表锁，则只需要判断一次即可知道有没数据行被锁定，提升性能。

3）意向锁和读锁【S锁】、写锁【X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

- 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。
- 表级S锁和X、IX锁不兼容：因为上了表级S锁后，不允许其他事务再加X锁。
- 表级X锁和 IS、IX、S、X不兼容：因为上了表级X锁后，会修改数据。

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

4.2.2 记录锁

记录锁(Record Locks)，仅仅锁住索引记录的一行，在单条索引记录上加锁。记录锁锁住的永远是索引，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写锁。

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

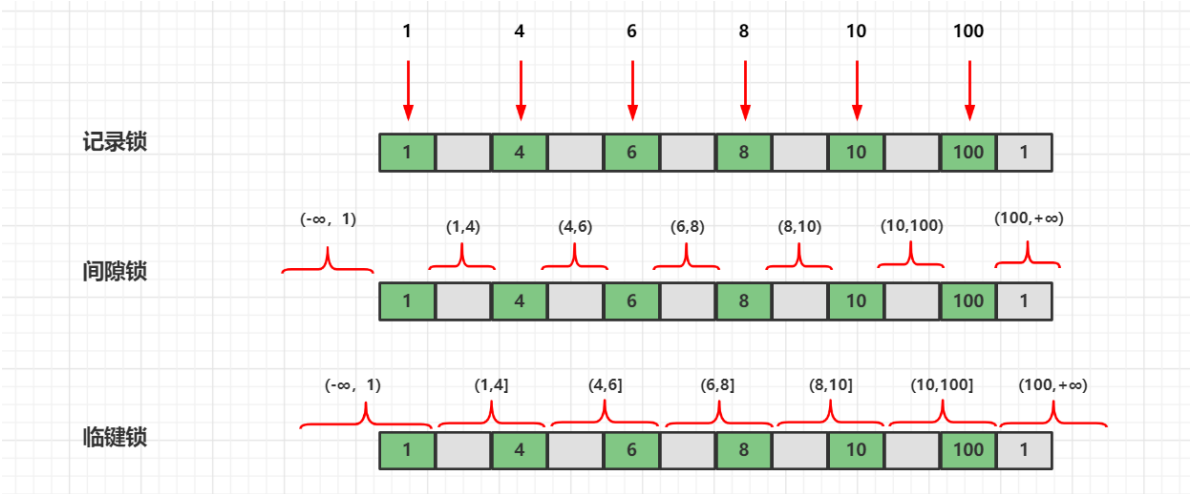
4.2.3 间隙锁

（1）间隙锁(Gap Locks)，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，保证索引间不会被插入数据

主键id索引的行锁区间划分图：



session1执行：

```
begin;
select * from t1_simple where id > 4 for update;
```

session2执行：

```
insert into t1_simple values (7,100); --阻塞
insert into t1_simple values (7,100); --成功
```

4.2.4 临键锁

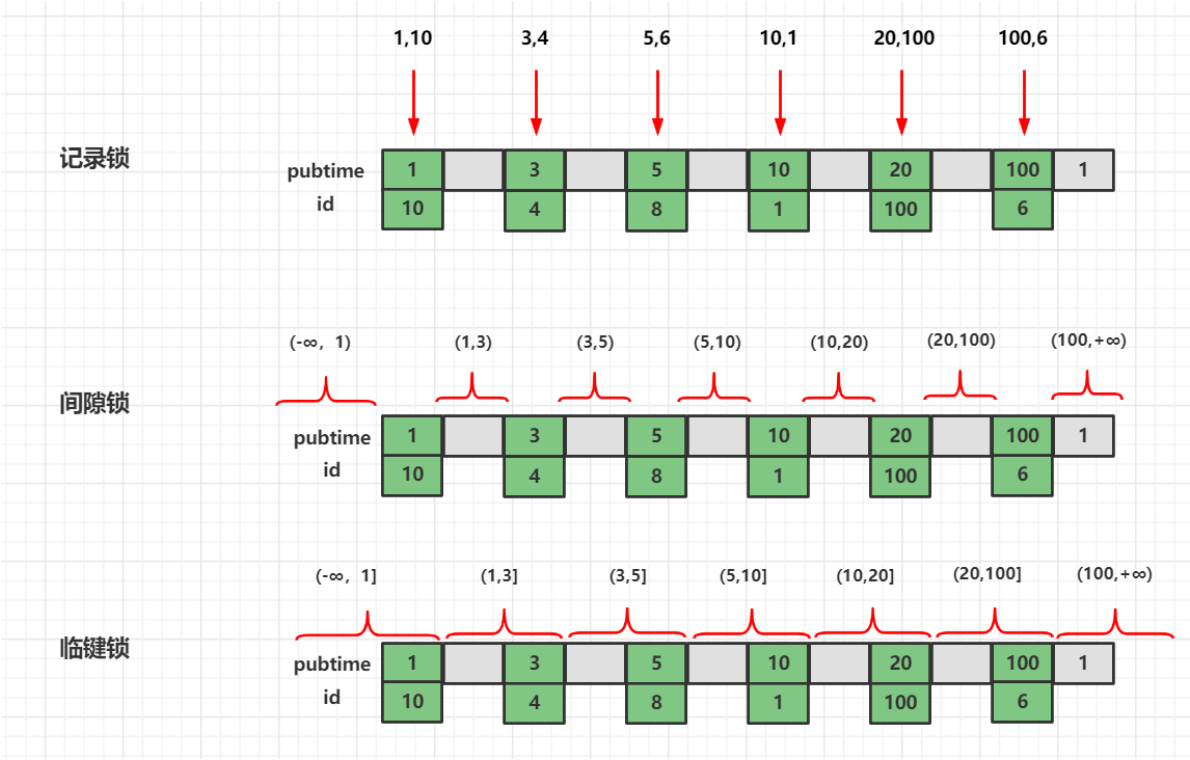
（1）临键锁(Next-Key Locks)相当于记录锁 + 间隙锁**【左开右闭区间】**，例如（5,8]

（2）默认情况下，innodb使用**临键锁**来锁定记录。

（3）但当查询的索引含有唯一属性的时候，**临键锁**会进行优化，将其降级为**记录锁**，即仅锁住索引本身，不是范围。

（4）临键锁在不同的场景中会退化:

普通索引index(pubtime)行锁的区间划分图：



场景	退化成的锁类型
使用Unique index 精确匹配【=】，且记录存在	记录锁
使用Unique index 精确匹配【=】，且记录不存在	间隙锁
使用Unique index 范围匹配【<和>】	临键锁

当前数据库中的记录信息：

```
mysql> select * from t1_simple;
```

```
mysql> select * from t1_simple;
+-----+---------+
| id  | pubtime |
+-----+---------+
|  10 |       1 |
|   4 |       3 |
|   8 |       5 |
|   1 |      10 |
| 100 |      20 |
|   3 |     100 |
|   6 |     100 |
+-----+---------+
7 rows in set (0.00 sec)
```

session1执行：

```
begin;
select * from t1_simple where pubtime = 20 for update;
-- 间隙锁(10,20],(20,100]
```

session2执行：

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

4.3 加锁规则

主键索引

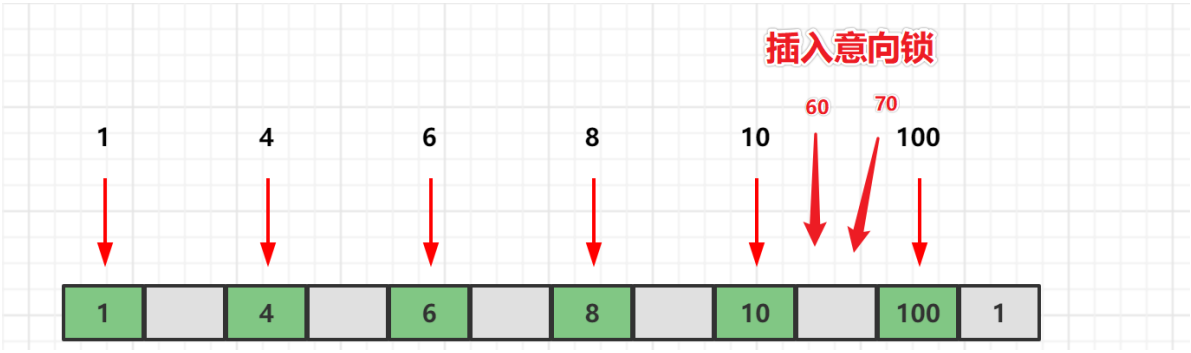
- 等值查询，命中，加记录锁
- 等值查询，未命中，加间隙锁
- 范围查询，命中，包含where条件的临键区间，加临键锁
- 范围查询，没有命中，加间隙锁

辅助索引

- 等值查询，命中，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
- 等值查询，未命中，加间隙锁
- 范围查询，命中，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
- 范围查询，没有命中，加间隙锁

4.4 插入意向锁

- （1）插入意向锁(Insert Intention Locks)是一种间隙锁，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10和100，不同的事务分别插入60和70，每个事务都会产生一个加在10-100之间的插入意向锁，获取在插入行上的写锁，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。



4.5 锁相关参数

Innodb所使用的**行级锁定**争用状态查看：

```
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)
```

- Innodb_row_lock_current_waits：当前正在等待锁定的数量；
- Innodb_row_lock_time：从系统启动到现在锁定总时间长度；
- Innodb_row_lock_time_avg：每次等待所花平均时间；
- 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；
- 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：

- Innodb_row_lock_time_avg（等待平均时长）
- Innodb_row_lock_waits（等待总次数）
- 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：

```sql
# 查看锁的SQL
select * from information_schema.innodb_locks;
select * from information_schema.innodb_lock_waits;
# 查看事务SQL
select * from information_schema.innodb_trx;
# 查看未关闭的事务详情
SELECT
   a.trx_id,a.trx_state,a.trx_started,a.trx_query,
   b.ID,b.USER,b.DB,b.COMMAND,b.TIME,b.STATE,b.INFO,
   c.PROCESSLIST_USER,c.PROCESSLIST_HOST,c.PROCESSLIST_DB,d.SQL_TEXT
FROM
   information_schema.INNODB_TRX a
LEFT JOIN information_schema.PROCESSLIST b ON a.trx_mysql_thread_id = b.id
AND b.COMMAND = 'Sleep'
LEFT JOIN PERFORMANCE_SCHEMA.threads c ON b.id = c.PROCESSLIST_ID
LEFT JOIN PERFORMANCE_SCHEMA.events_statements_current d ON d.THREAD_ID =
c.THREAD_ID;
```

五、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。从最简单的例子说起，下面两条简单的SQL，他们加的什么锁？

- **SQL1：**

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

- **SQL2：**

```
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！！！

- 读已提交【RC】隔离级别
  - 组合一：id列是主键，
  - 组合二：id列是二级唯一索引
  - 组合三：id列是二级非唯一索引
  - 组合四：id列上没有索引
- 可重复读【RC】隔离级别
  - 组合五：id列是主键
  - 组合六：id列是二级唯一索引
  - 组合七：id列是二级非唯一索引
  - 组合八：id列上没有索引
- 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？

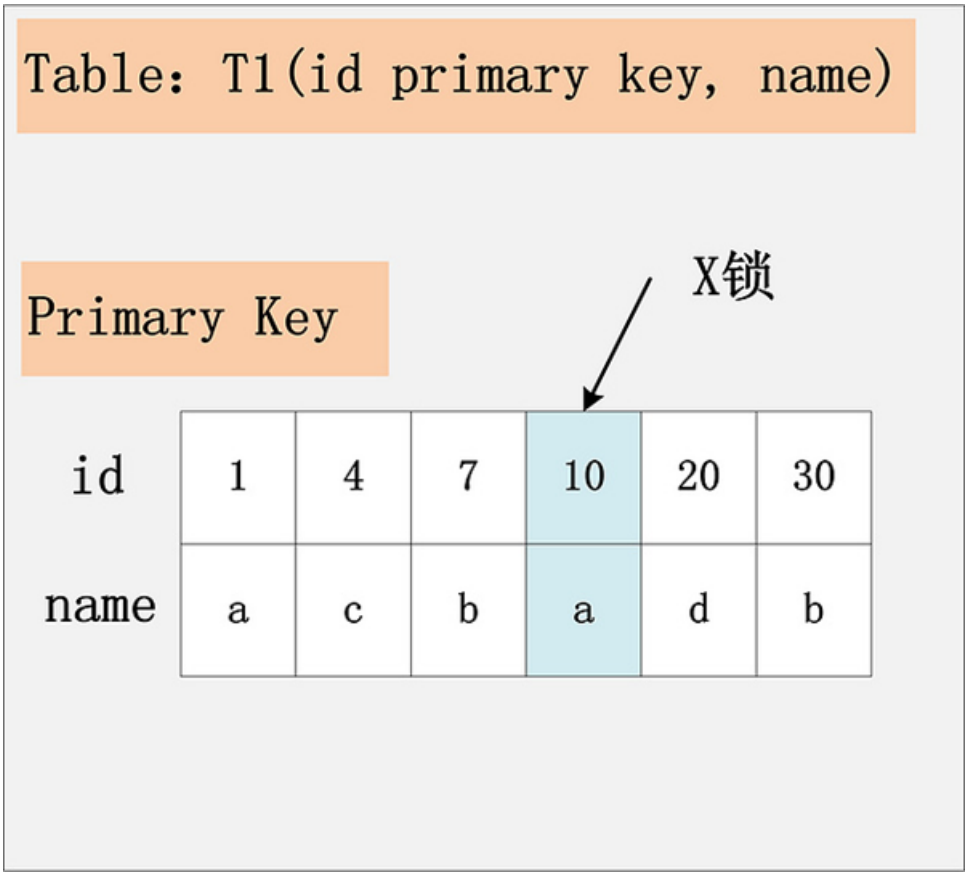
事实上，要分析加锁，就是需要这么复杂。但是从另一个角度来说，只要你选定了一种组合，SQL需要加哪些锁，其实也就确定了。接下来，就让我们来逐个分析这9种组合下的SQL加锁策略。

> 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。**

5.1 读已提交RC

1）组合一：id主键

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的记录加上写锁即可。如下图所示：

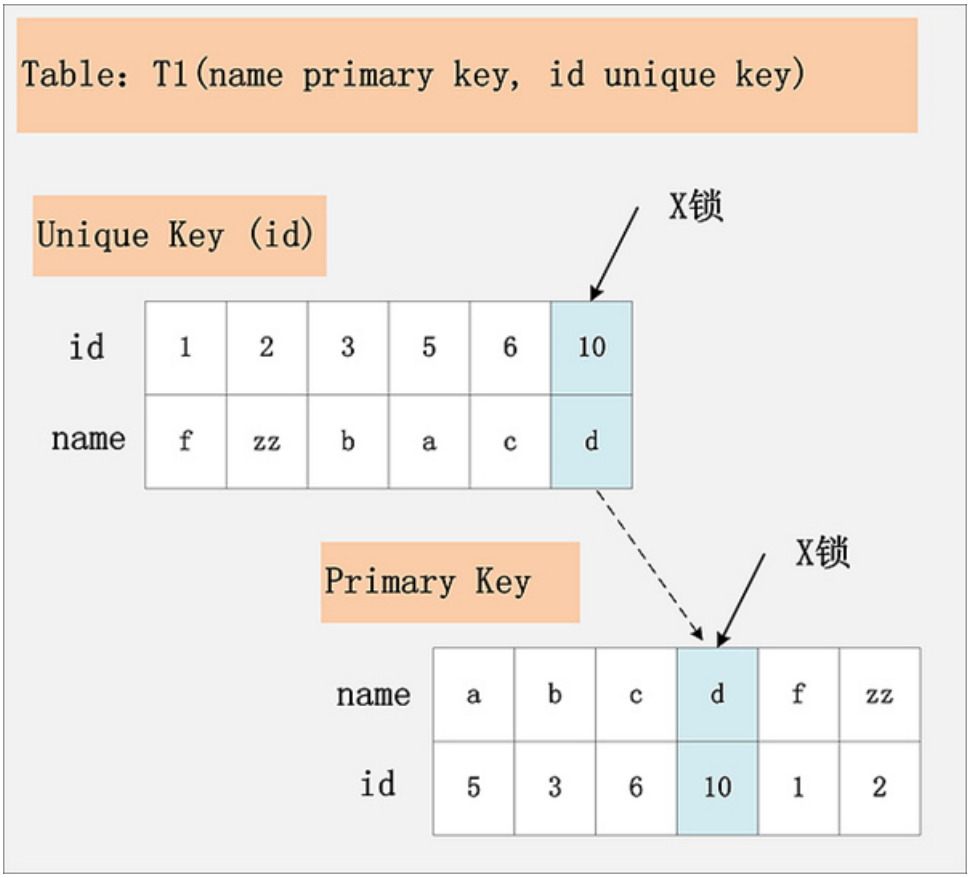


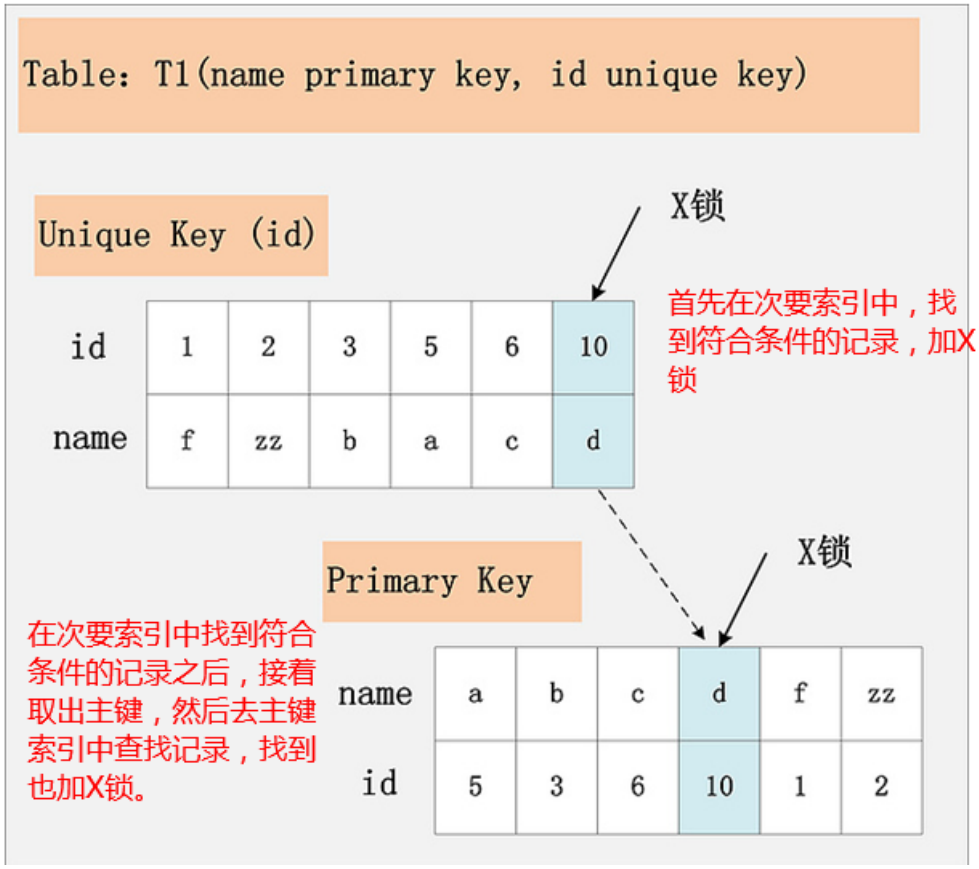
结论：id是主键时，此SQL只需要在id=10这条记录上加写锁即可。

2）组合二：id唯一索引

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：





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**写锁**，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**写锁**。

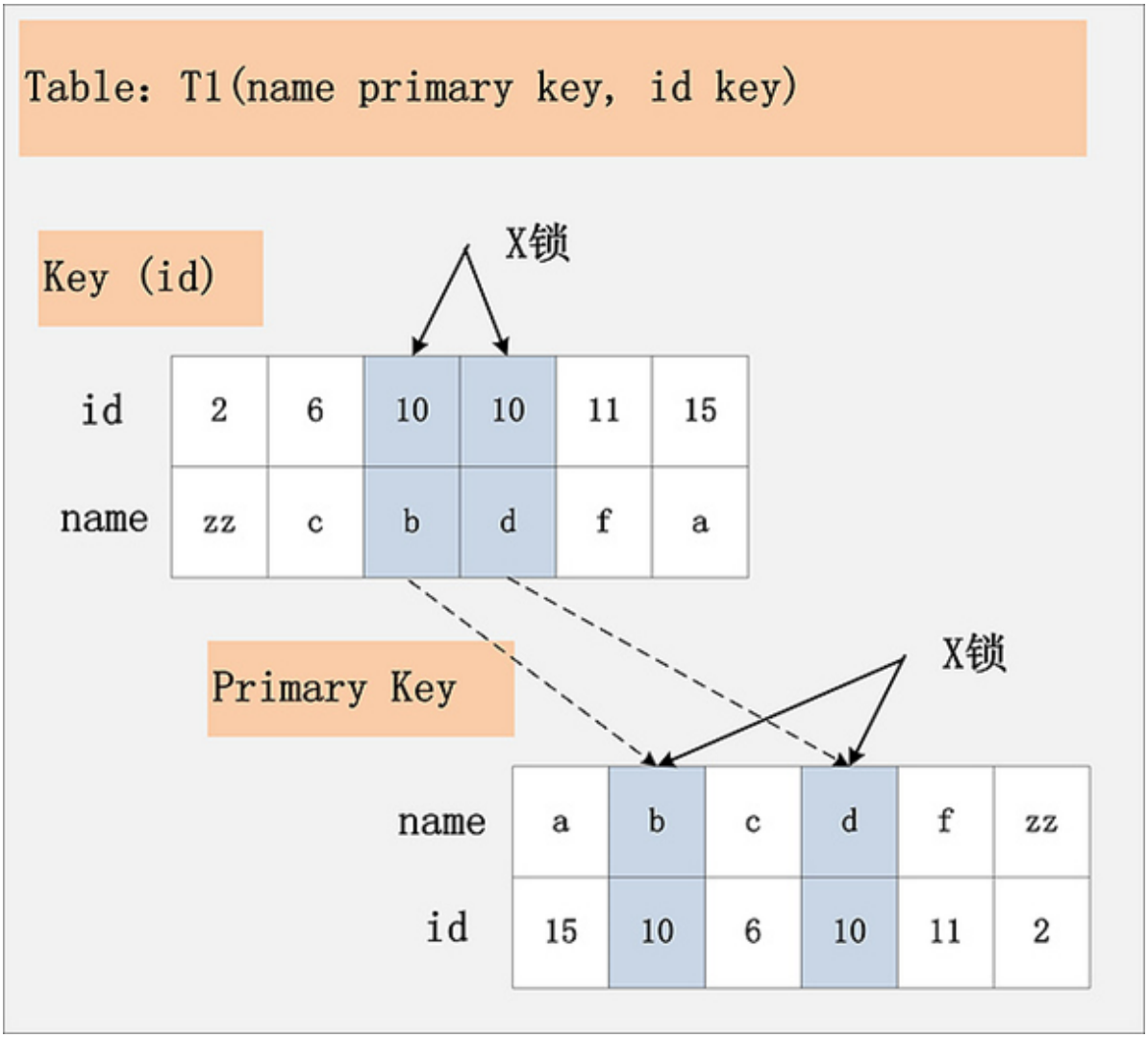
为什么聚簇索引上的记录也要加锁?

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：若id列是unique列，其上有unique索引。那么SQL需要加两个**写锁**，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

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：



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：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。

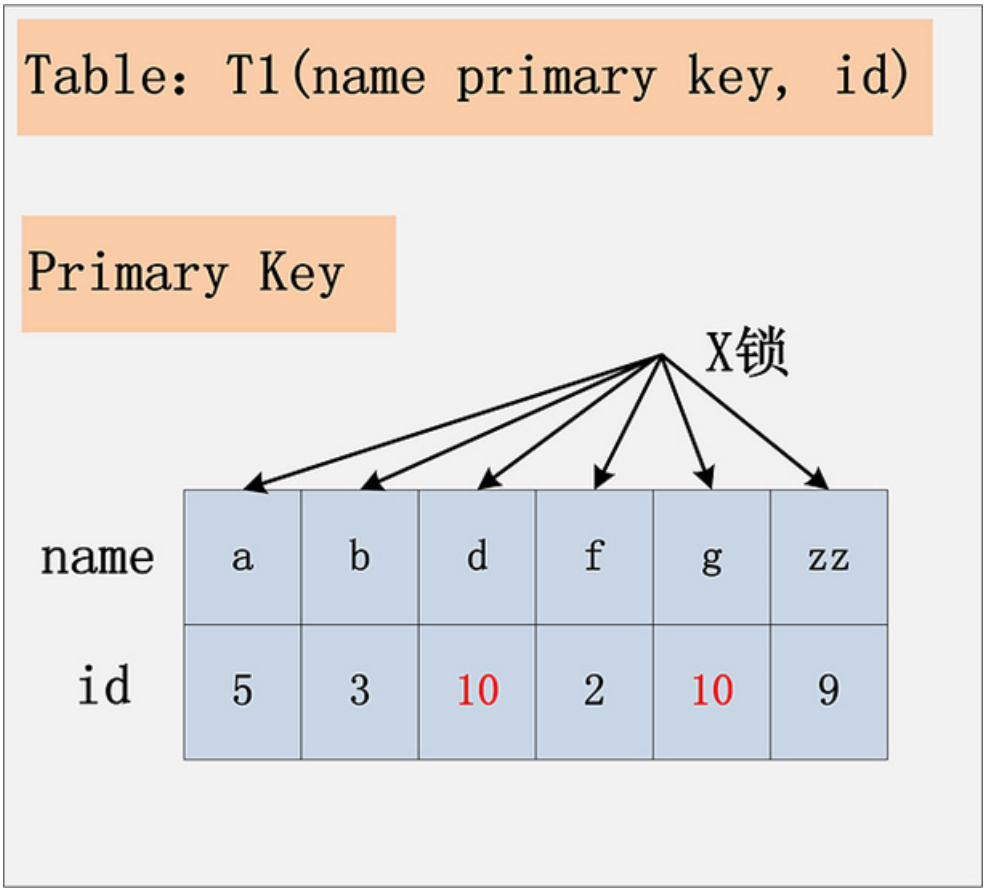
4）组合四：id无索引

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**写锁**；有人说会将聚簇索引上，选择出来的id = 10;的记录加上**写锁**。那么实际情况呢？

请看下图：



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**写锁**。无论记录是否满足条件，全部被加上**写锁**。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。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：

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**写锁**。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。

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。

5.2 可重复RR

1）组合五：id主键

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。

2）组合六：id唯一索引

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。**两个写锁，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。

3）组合七：id非唯一索引

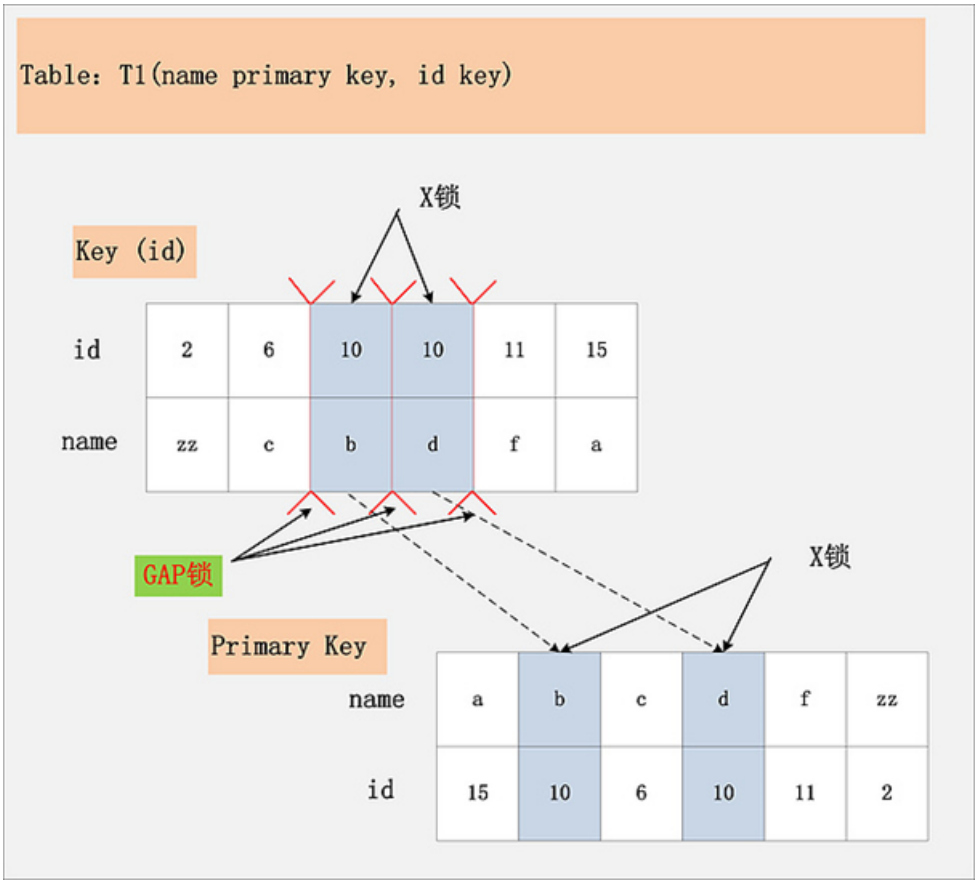
还记得MySQL的四种隔离级别的区别吗?

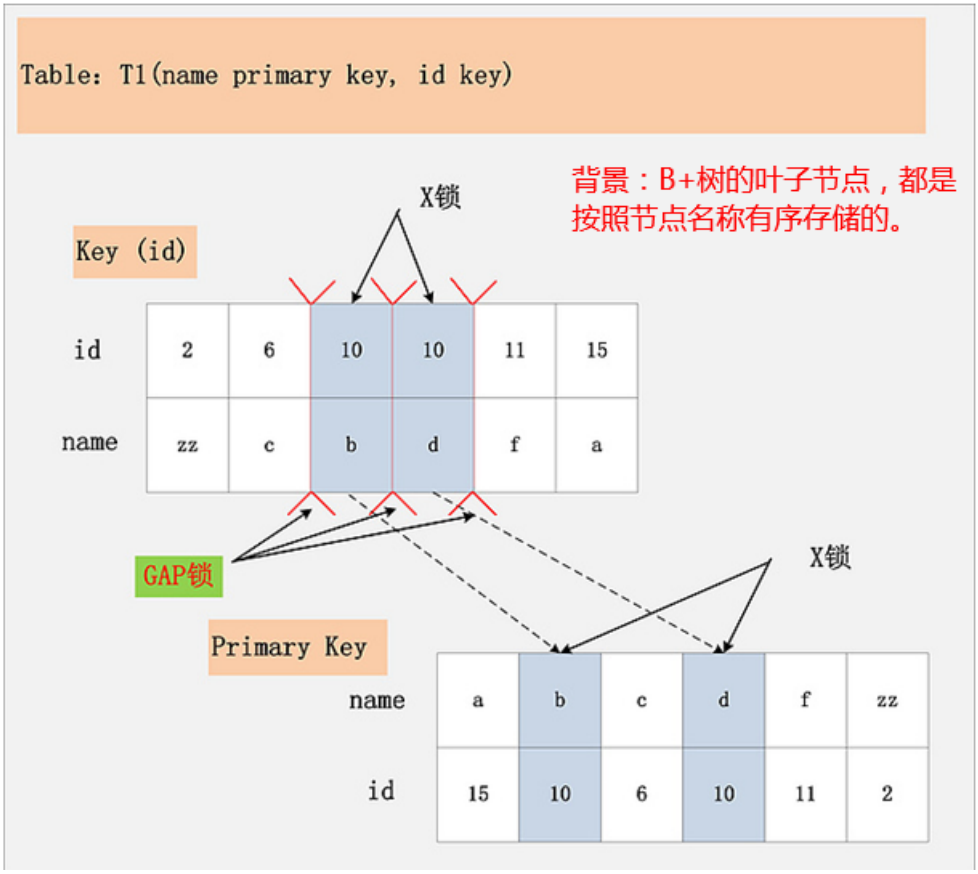
- **RC隔离级别允许幻读，而RR隔离级别，不允许存在幻读。**
- 在组合五、组合六中，加锁行为又是与RC下的加锁行为完全一致。

那么RR隔离级别下，如何防止幻读呢?

- 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，是怎么样的呢?

同样看下面这幅图：





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，这幅图中多了一个间隙锁，而且间隙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，**间隙锁有何用?**

其实这个多出来的间隙锁，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，间隙锁锁住的位置，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)。

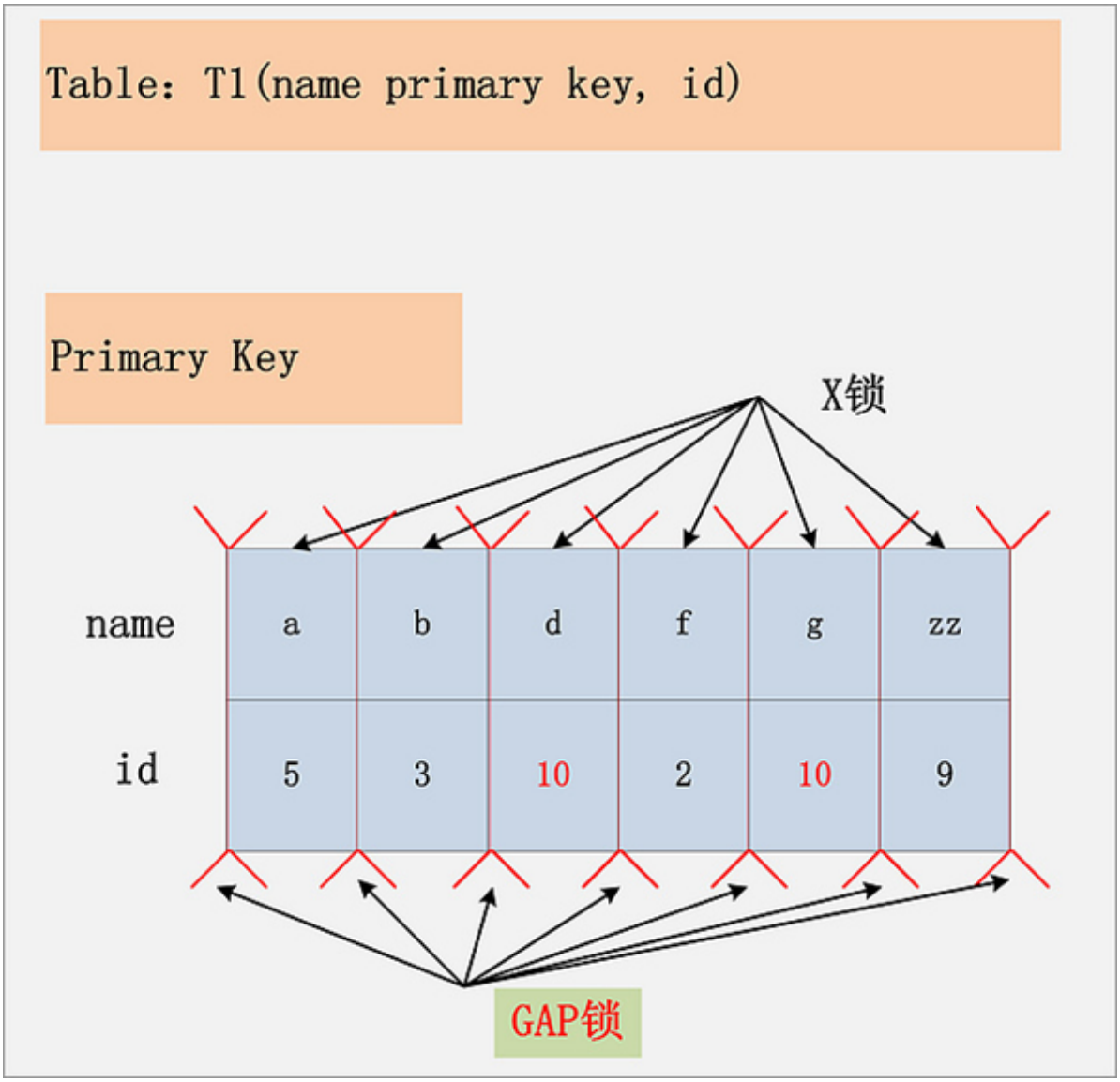
如何保证两次当前读返回一致的记录? 那就需要在第一次当前读与第二次当前读之间，其他的事务不会插入新的满足条件的记录并提交。为了实现这个功能，间隙锁应运而生。

结论：

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，加记录上的写锁，加GAP上的间隙锁，然后加主键聚簇索引上的记录**写锁**，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**写锁**，但是仍旧需要加间隙锁，最后返回结束。

4）组合八：id无索引

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**写锁。**其次，聚簇索引每条记录间的间隙，也同时被加上了间隙锁。这个示例表，只有6条记录，一共需要6个记录锁，7个间隙锁。试想，如果表上有1000万条记录呢?

在这种情况下，这个表上，除了不加锁的快照度，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，同时不加间隙锁。

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:

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间隙，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。

5.3 串行化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隔离级别组合八情况完全一致，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LBCC。**

结论：

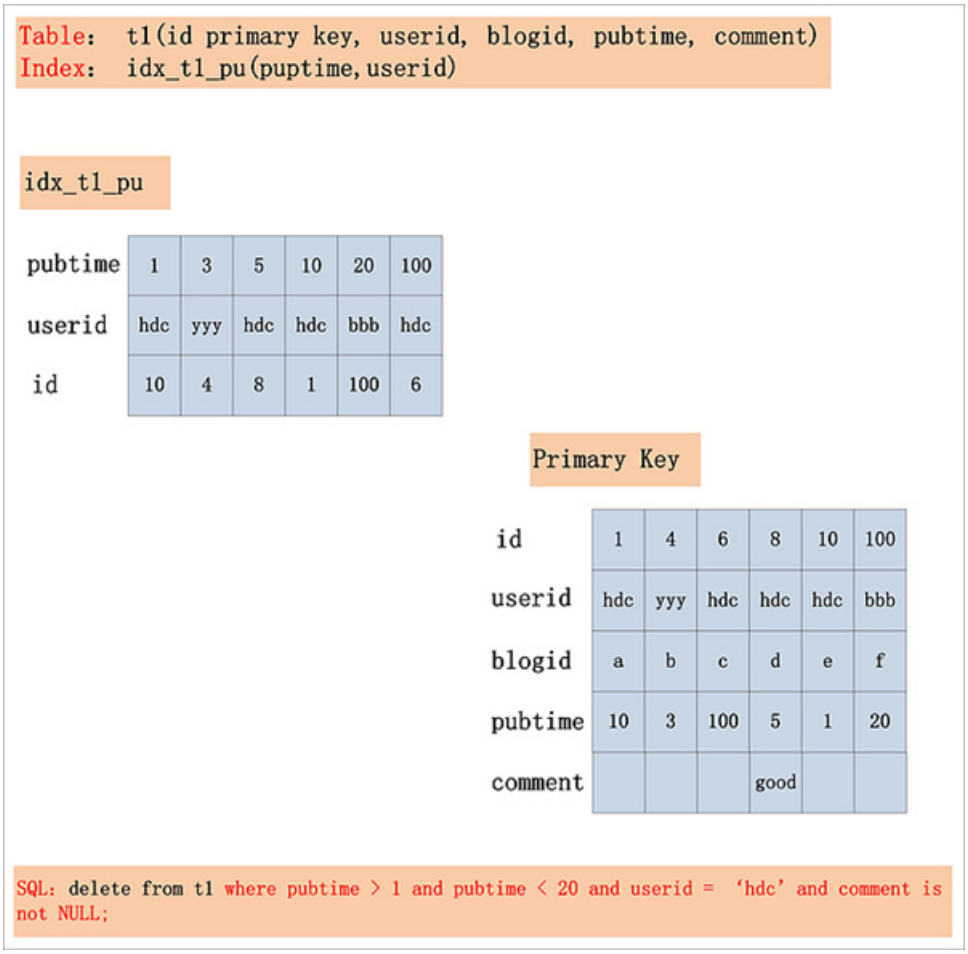
在MySQL/InnoDB中，所谓的读不加锁，并不适用于所有的情况，而是隔离级别相关的。Serializable隔离级别，读不加锁就不再成立，所有的读操作，都是当前读。

5.4 复杂SQL加锁分析

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：

```
1 delete from t1 where pubtime > 1 and pubtime < 20 and userid='hdc' and commit
  is not null;
```



如图中的SQL，会加什么锁?

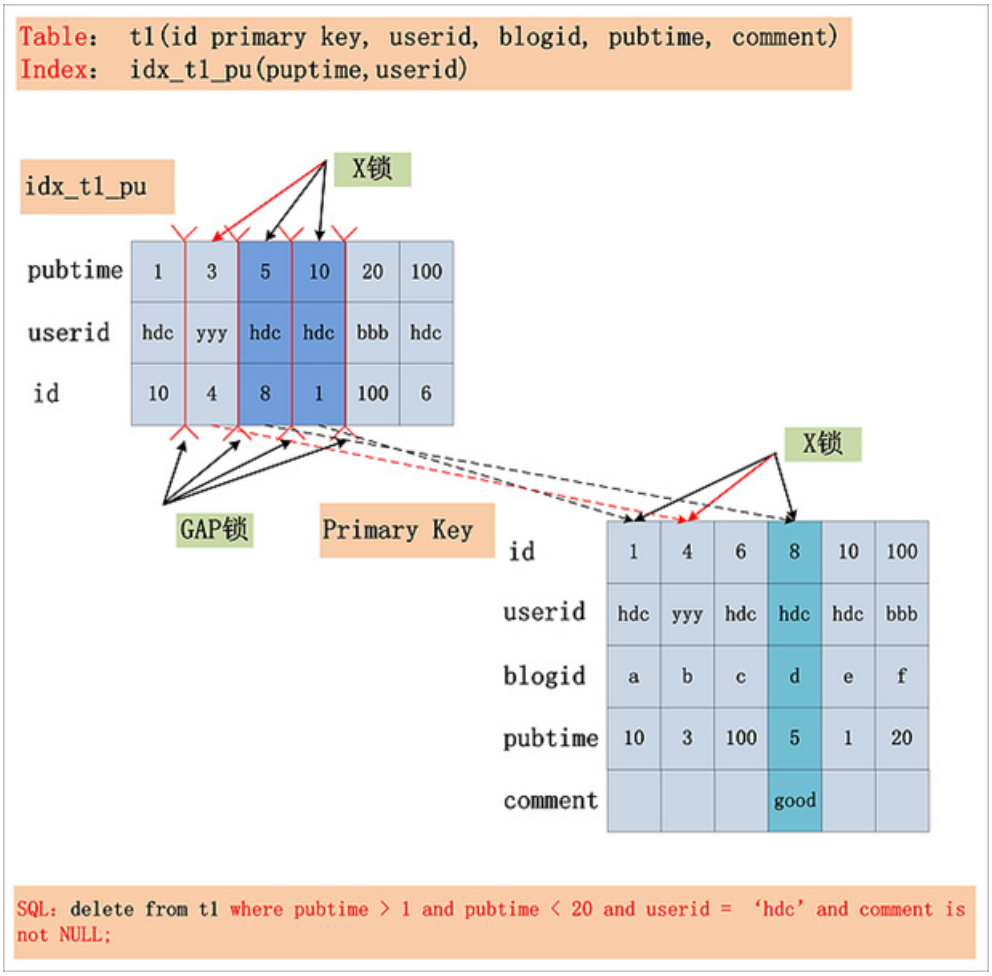
假定在Repeatable Read隔离级别下，同时，假设SQL走的是idx_t1_pu索引。

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 puptime < 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，再来看看这条SQL的加锁情况 (RR隔离级别)，如下图所示:



从图中可以看出，在Repeatable Read隔离级别下，由Index Key所确定的范围，被加上了间隙锁；Index Filter锁给定的条件 (userid = 'hdc')何时过滤，视MySQL的版本而定【 图中，用红色箭头标出的写锁是否要加，与ICP有关】

- 在MySQL 5.6版本之前，不支持Index Condition Pushdown，因此Index Filter在MySQL Server层过滤。
    - 若不支持ICP，不满足Index Filter的记录，也需要加上记录写锁；
- 在MySQL 5.6版本之后，支持了Index Condition Pushdown，则在index上过滤。
    - 若支持ICP，则不满足Index Filter的记录，无需加记录写锁；

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写锁。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：

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。

- Index Key确定的范围，需要加上间隙锁；
- 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，不加写锁，否则需要写锁；
- Table Filter过滤条件，无论是否满足，都需要加写锁。

六、死锁原理

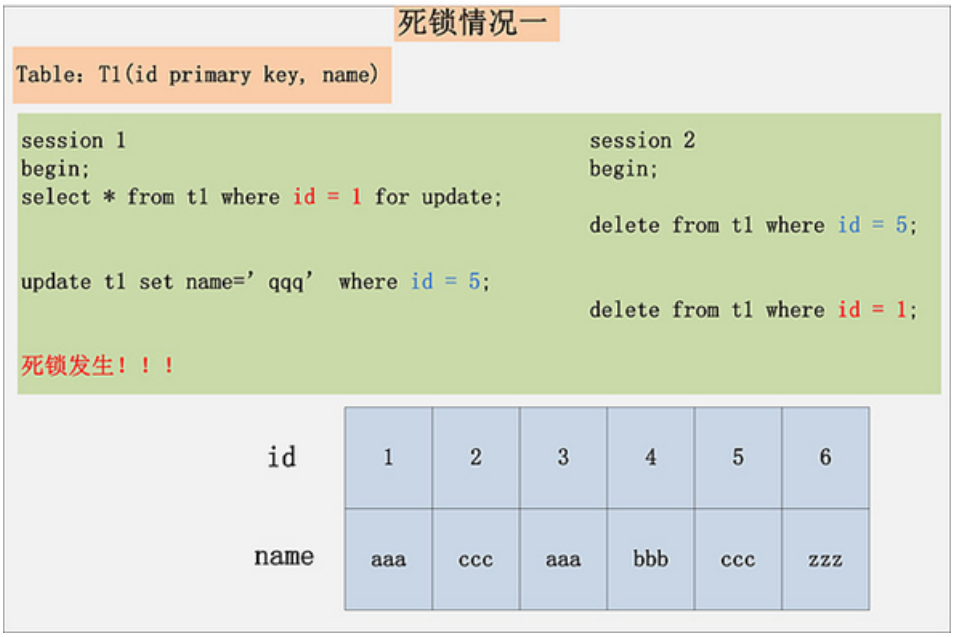
深入理解MySQL如何加锁，有两个比较重要的作用：

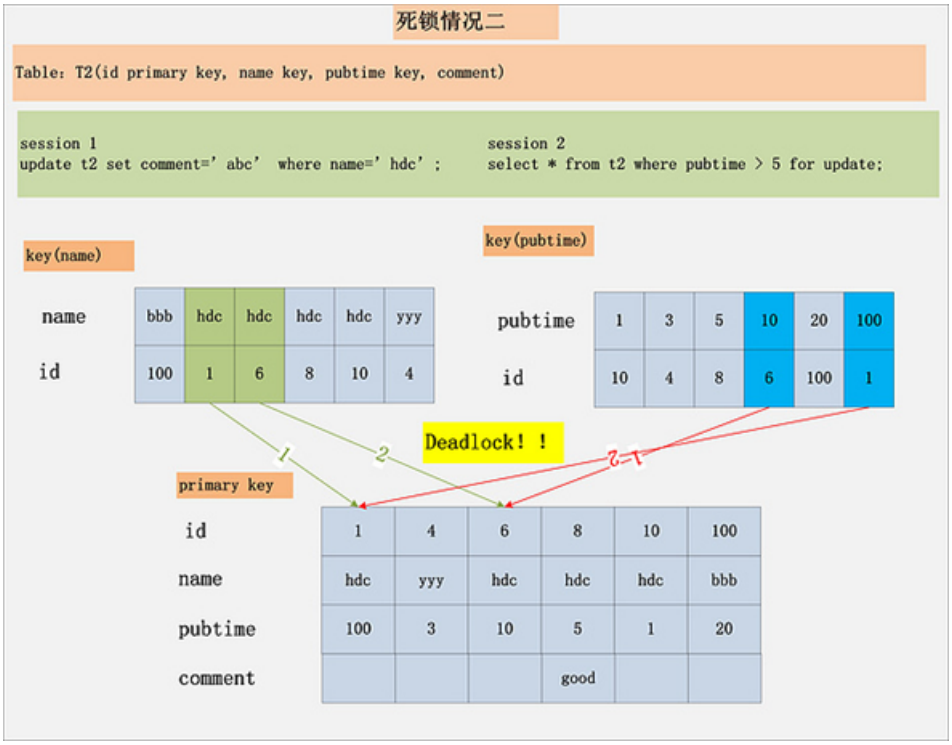
- 可以根据MySQL的加锁规则，写出不会发生死锁的SQL；

- 可以根据MySQL的加锁规则，定位出线上产生死锁的原因；

6.1 什么是死锁?

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**：





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。

- 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索引上的记录写锁，而且会加聚簇索引上的记录写锁，加锁顺序为先[1,hdc,100]，后[6,hdc,10]。
- 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写锁，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：

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。

6.2 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。

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