MySQL的并发控制与加锁分析

本文主要是针对MySQL/InnoDB的并发控制和加锁技术做一个比较深入的剖析，并且对其中涉及到的重要的概念，如多版本并发控制（MVCC），脏读（dirty read），幻读（phantom read），四种隔离级别（isolation level）等作详细的阐述，并且基于一个简单的例子，对MySQL的加锁进行了一个详细的分析。本文的总结参考了何登成前辈的博客，并且在前辈总结的基础上，进行了一些基础性的说明，希望对刚入门的同学产生些许帮助，如有错误，请不吝赐教。按照我的写作习惯，还是通过几个关键问题来组织行文逻辑，如下：

- 什么是MVCC（多版本并发控制）？如何理解快照读（snapshot read）和当前读（current read）？
- 什么是隔离级别？脏读？幻读？InnoDB的四种隔离级别的含义是什么？
- 什么是死锁？
- InnoDB是如何实现MVCC的？
- 一个简单的sql在不同场景下的加锁分析
- 一个复杂的sql的加锁分析

接下来，我将按照这几个关键问题的顺序，对以上问题作一一解答，并且在解答的过程中，争取将加锁技术的细节，阐述的更加清楚。

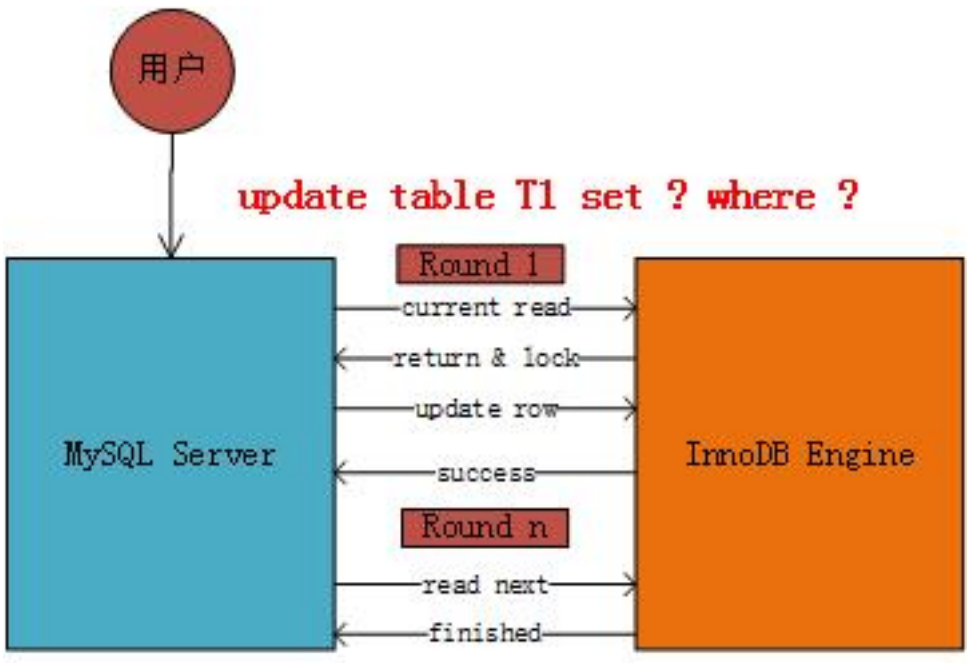
1.1 MVCC：Multi-Version Concurrent Control 多版本并发控制

MVCC是为了实现数据库的并发控制而设计的一种协议。从我们的直观理解上来看，要实现数据库的并发访问控制，最简单的做法就是**加锁访问**，即读的时候不能写（允许多个西线程同时读，即**共享锁**，S锁），写的时候不能读（一次最多只能有一个线程对同一份数据进行写操作，即**排它锁**，X锁）。这样的加锁访问，其实并不算是真正的并发，或者说它只能实现并发的读，因为它最终实现的是**读写串行化**，这样就大大降低了数据库的读写性能。加锁访问其实就是和MVCC相对的LBCC，即基于锁的并发控制（Lock-Based Concurrent Control），是四种隔离级别中级别最高的Serialize隔离级别。为了提出比LBCC更优越的并发性能方法，MVCC便应运而生。

几乎所有的RDBMS都支持MVCC。它的最大好处便是，读不加锁，读写不冲突。在MVCC中，读操作可以分成两类，快照读（Snapshot read）和当前读（current read）。快照读，读取的是记录的可见版本（可能是历史版本，即最新的数据可能正在被当前执行的事务并发修改），不会对返回的记录加锁；而当前读，读取的是记录的最新版本，并且会对返回的记录加锁，保证其他事务不会并发修改这条记录。在MySQL InnoDB中，简单的select操作，如 select * from table where ? 都属于快照读；属于当前读的包含以下操作：

1. select * from table where ? lock in share mode; （加S锁）
2. select * from table where ? for update; （加X锁，下同）
3. insert, update, delete操作

针对一条当前读的SQL语句，InnoDB与MySQL Server的交互，是一条一条进行的，因此，加锁也是一条一条进行的。先对一条满足条件的记录加锁，返回给MySQL Server，做一些DML操作；然后再读取下一条加锁，直至读取完毕。需要注意的是，以上需要加X锁的都是当前读，而普通的select（除了for update）都是快照读，每次insert、update、delete之前都是会进行一次当前读的，这个时候会上锁，防止其他事务对某些行数据的修改，从而造成数据的不一致性。我们广义上说的幻读现象是通过MVCC解决的，意思是通过MVCC的快照读可以使得事务返回相同的数据集。如下图所示：



注意，我们一般说在MyISAM中使用表锁，因为MyISAM在修改数据记录的时候会将整个表锁起来；而InnoDB使用的是行锁，即我们以上所谈的MVCC的加锁问题。但是，并不是InnoDB引擎不会使用表锁，比如在alter table的

时候，Innodb就会将该表用表锁锁起来。

1.2 隔离级别

在SQL的标准中，定义了四种隔离级别。每一种级别都规定了，在一个事务中所做的修改，哪些在事务内和事务间是可见的，哪些是不可见的。低级别的隔离可以执行更高级别的并发，性能好，但是会出现脏读和幻读的现象。首先，我们从两个基础的概念说起：

脏读（dirty read）：两个事务，一个事务读取到了另一个事务未提交的数据，这便是脏读。

幻读（phantom read）：两个事务，事务A与事务B，事务A在自己执行的过程中，执行了两次相同查询，第一次查询事务B未提交，第二次查询事务B已提交，从而造成两次查询结果不一样，这个其实被称为**不可重复读**；如果事务B是一个会影响查询结果的insert操作，则好像新多出来的行像幻觉一样，因此被称为幻读。**其他事务的提交会影响在同一个事务中的重复查询结果**。

下面简单描述一下SQL中定义的四种标准隔离级别：

1. **READ UNCOMMITTED (未提交读)**：隔离级别：0. 可以读取未提交的记录。会出现脏读。
2. **READ COMMITTED (提交读)**：隔离级别：1. 事务中只能看到已提交的修改。不可重复读，会出现幻读。（在InnoDB中，会加行所，但是不会加间隙锁）该隔离级别是大多数数据库系统的默认隔离级别，但是MySQL的则是RR。
3. **REPEATABLE READ (可重复读)**：隔离级别：2. **在InnoDB中是这样的**：RR隔离级别保证对读取到的记录加锁 (记录锁)，同时保证对读取的范围加锁，新的满足查询条件的记录不能够插入 (间隙锁)，因此不存在幻读现象。但是标准的RR只能保证在同一事务中多次读取同样记录的结果是一致的，而无法解决幻读问题。InnoDB的幻读解决是依靠MVCC的实现机制做到的。
4. **SERIALIZABLE （可串行化）**：隔离级别：3. 该隔离级别会在读取的每一行数据上都加上锁，退化为基于锁的并发控制，即LBCC。

需要注意的是，MVCC只在RC和RR两个隔离级别下工作，其他两个隔

离级别都和MVCC不兼容。

1.3 死锁

死锁是指两个或者多个事务在同一资源上相互作用，并请求锁定对方占用的资源，从而导致恶性循环的现象。**当多个事务试图以不同的顺序锁定资源时，就可能产生死锁。多个事务同时锁定同一个资源时，也会产生死锁。**且看下面的两个产生死锁的例子：

死锁情况：两个事务的两条SQL产生

```
Table T1(id primary key, name)
```

```
Session 1:
Begin;

Select * from t1 where id=1 for update;

Update t1 set name= 'deadlock' where id=5;
```

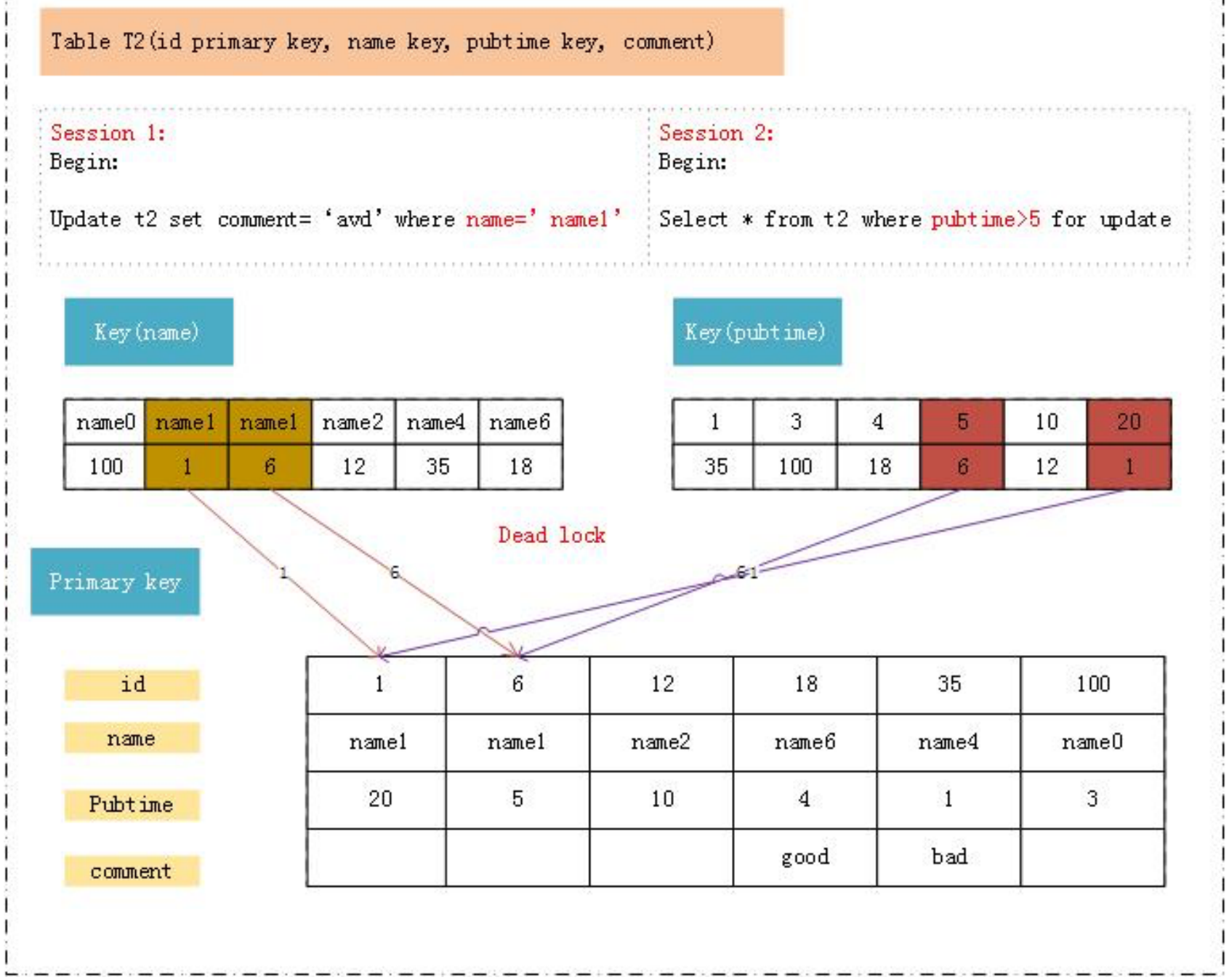
```
Session 2:
Begin;

Delete from t1 where id=5;

Delete from t1 where id=1;
```

死锁发生

1	2	3	4	5	6
Aaa	bbb	ccc	ddd	eee	ff



第一个死锁很好理解，而第二个死锁，由于在主索引（聚簇索引表）上仍旧是对两条记录进行了不同顺序的加锁，因此仍旧会造成死锁。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**。因此，我们通过分析加锁细节，可以判断所写的sql是否会发生死锁，同时发生死锁的时候，我们应该如何处理。

1.4 InnoDB的MVCC实现机制

MVCC可以认为是行级锁的一个变种，它可以在很多情况下避免加锁操作，因此开销更低。MVCC的实现大都都实现了非阻塞的读操作，写操作也只锁定必要的行。InnoDB的MVCC实现，是通过保存数据在某个时间点的快照来实现的。**一个事务，不管其执行多长时间，其内部看到的数据是一致的**。也就是事务在执行的过程中不会相互影响。下面我们简述一下MVCC在InnoDB中的实现。

InnoDB的MVCC，通过在每行记录后面保存两个隐藏的列来实现：一个保存了行的创建时间，一个保存行的过期时间（删除时间），当然，这里的时间并不是时间戳，而是系统版本号，每开始一个新的事务，系统版本号就会递增。在RR隔离级别下，MVCC的操作如下：

1. select操作。a. InnoDB只查找版本早于（包含等于）当前事务版本的数据行。可以确保事务读取的行，要么是事务开始前就已存在，或者事务自身插入或修改的记录。b. 行的删除版本要么未定义，要么大于当前事务版本号。可以确保事务读取的行，在事务开始之前未删除。
2. insert操作。将新插入的行保存当前版本号为行版本号。
3. delete操作。将删除的行保存当前版本号为删除标识。
4. update操作。变为insert和delete操作的组合，insert的行保存当前版本号为行版本号，delete则保存当前版本号到原来的行作为删除标识。

由于旧数据并不真正的删除，所以必须对这些数据进行清理，innodb会开启一个后台线程执行清理工作，具体的规则是将删除版本号小于当前系统版本的行删除，这个过程叫做purge。

1.5 一个简单SQL的加锁分析

在MySQL的InnoDB中，都是基于聚簇索引表的。而且普通的select操作都是基于快照读，是不需要加锁的。那么我们在分析其他的sql语句的时候，如何分析加锁细节？下面我们以一个简单的delete操作的SQL为例，进行一个详细的阐述。且看下面的SQL：

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

如果对这条SQL进行加锁分析，那么MySQL是如何加锁的呢？一般情况下，我们直观的感受是：会在id=10的记录上加锁。但是，这样轻率的下结论是片面的，要想确定MySQL的加锁情况，我们还需要知道更多的条件。还需要知道哪些条件呢？比如：

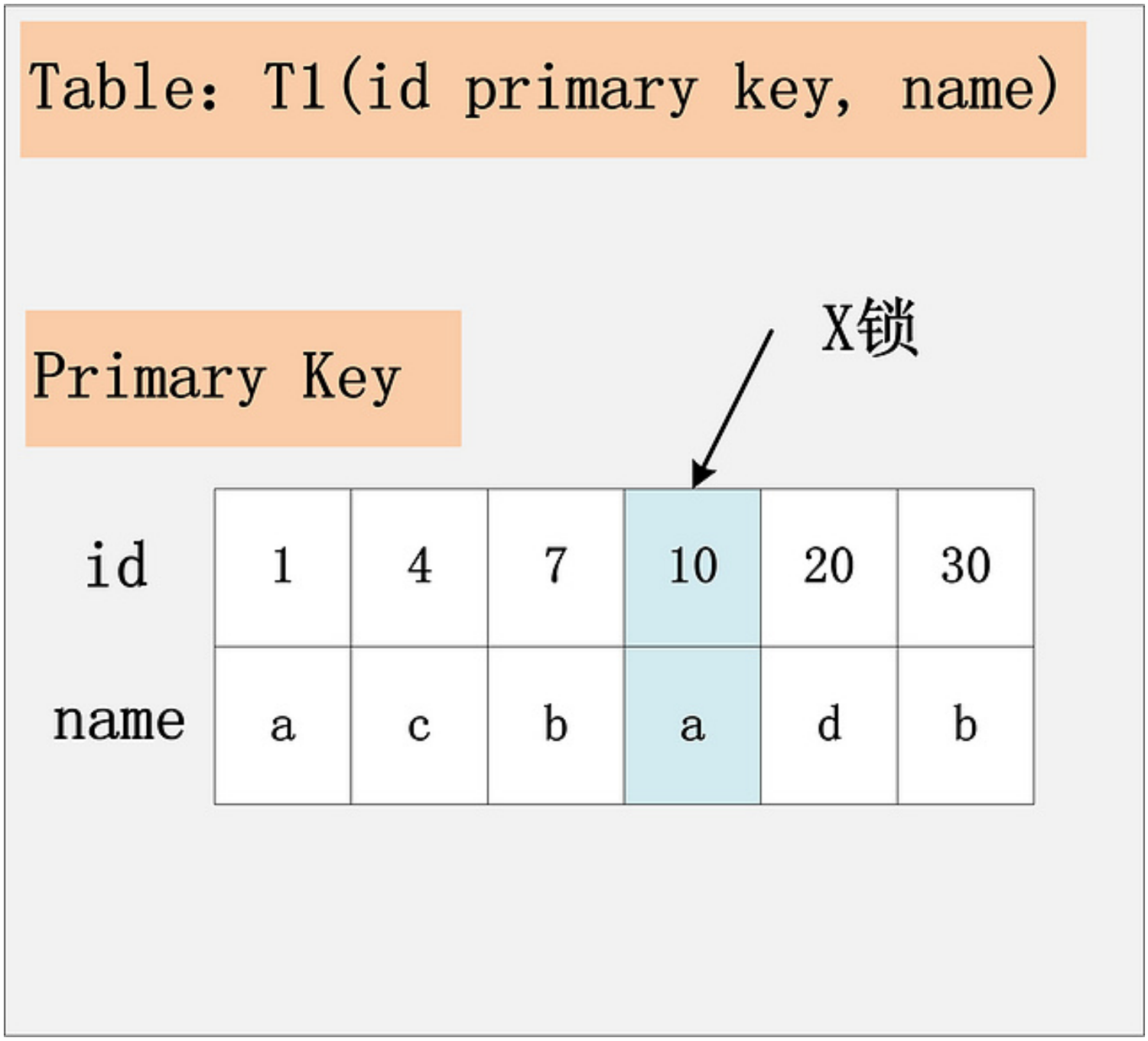
1. id列是不是主键？
2. 系统的隔离级别是什么？
3. id非主键的话，其上有建立索引吗？
4. 建立的索引是唯一索引吗？
5. 该SQL的执行计划是什么？索引扫描？全表扫描？

接下来，我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。

- 组合1：id列是主键，RC隔离级别
- 组合2：id列是二级唯一索引，RC隔离级别
- 组合3：id列是二级非唯一索引，RC隔离级别
- 组合4：id列上没有索引，RC隔离级别
- 组合5：id列是主键，RR隔离级别
- 组合6：id列是二级唯一索引，RR隔离级别
- 组合7：id列是二级非唯一索引，RR隔离级别
- 组合8：id列上没有索引，RR隔离级别
- 组合9：Serializable隔离级别

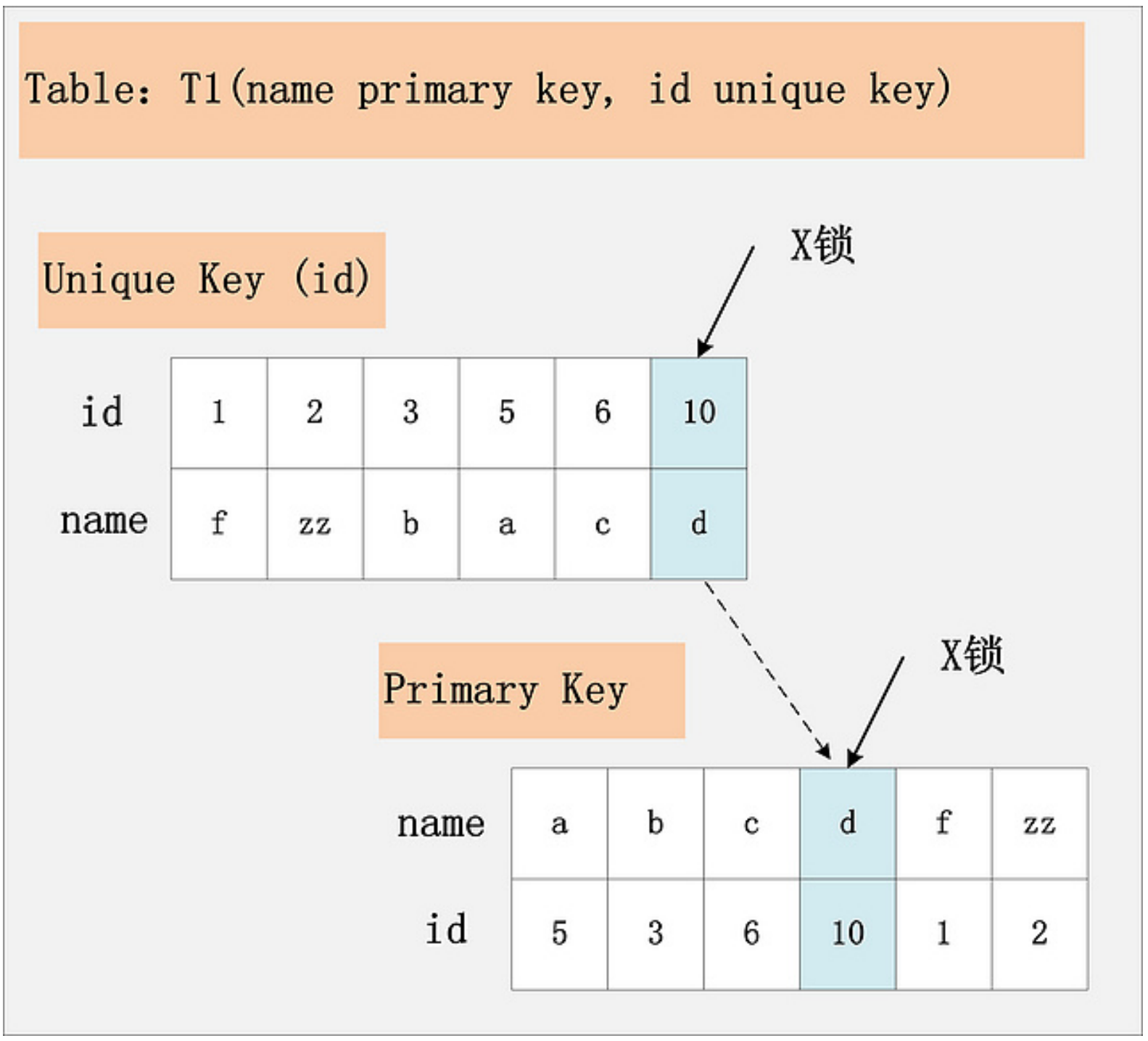
组合1：id列是主键，RC隔离级别

当id是主键的时候，我们只需要在该id=10的记录上加上x锁即可。如下图所示：



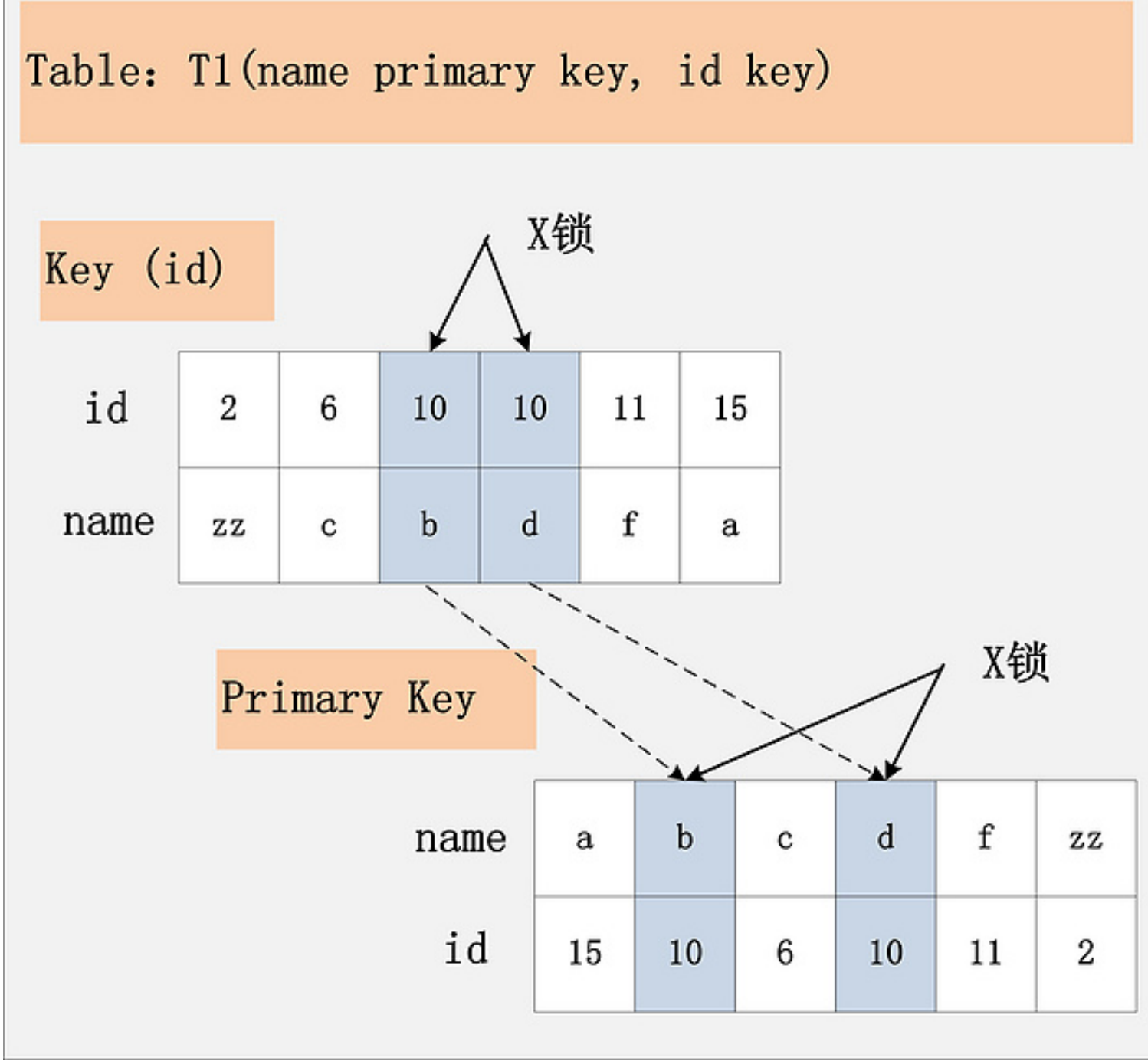
组合2：id列是二级唯一索引，RC隔离级别

在这里我先解释一下聚簇索引和普通索引的区别。在InnoDB中，主键可以被理解为聚簇索引，聚簇索引中的叶子结点就是相应的数据行，具有聚簇索引的表也被称为聚簇索引表，数据在存储的时候，是按照主键进行排序存储的。我们都知道，数据库在select的时候，会选择索引列进行查找，索引列都是按照B+树（多叉搜索树）数据结构进行存储，找到主键之后，再回到聚簇索引表中进行查询，这叫回表查询。那我们自然会问，**当使用索引进行查询的时候，与索引相对应的记录会被上锁吗？会的。**如果id是唯一索引，那么只给该唯一索引所对应的索引记录上x锁；如果id是非唯一索引，那么所对应的所有的索引记录上都会上x锁。如下图所示：



组合3：id列是二级非唯一索引，RC隔离级别

解释同上，如下图：

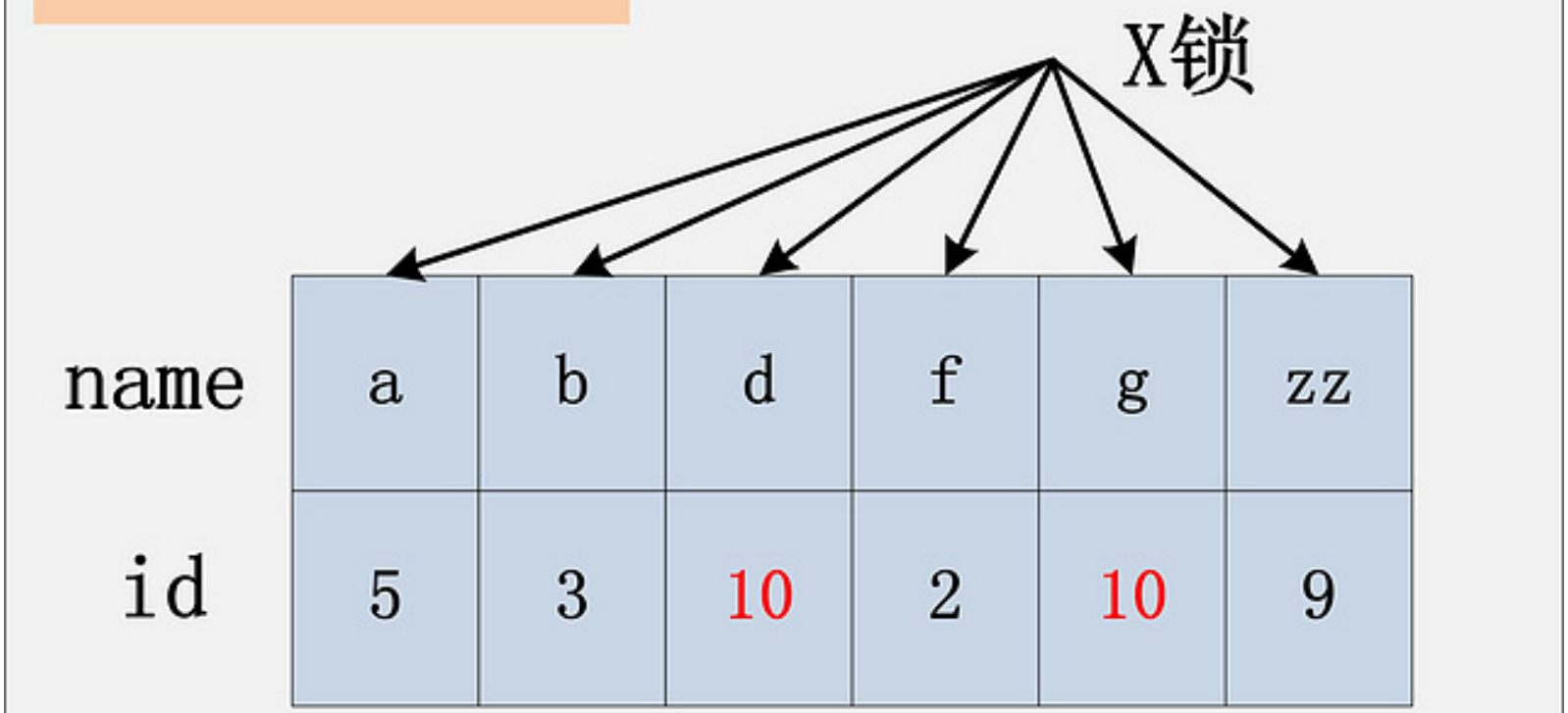


组合4：id列上没有索引，RC隔离级别

由于id列上没有索引，因此只能走聚簇索引，进行全部扫描。有人说会在表上加X锁；有人说会在聚簇索引上，选择出来的id = 10 的记录加上X锁。真实情况如下图：

Table: T1(name primary key, id)

Primary Key



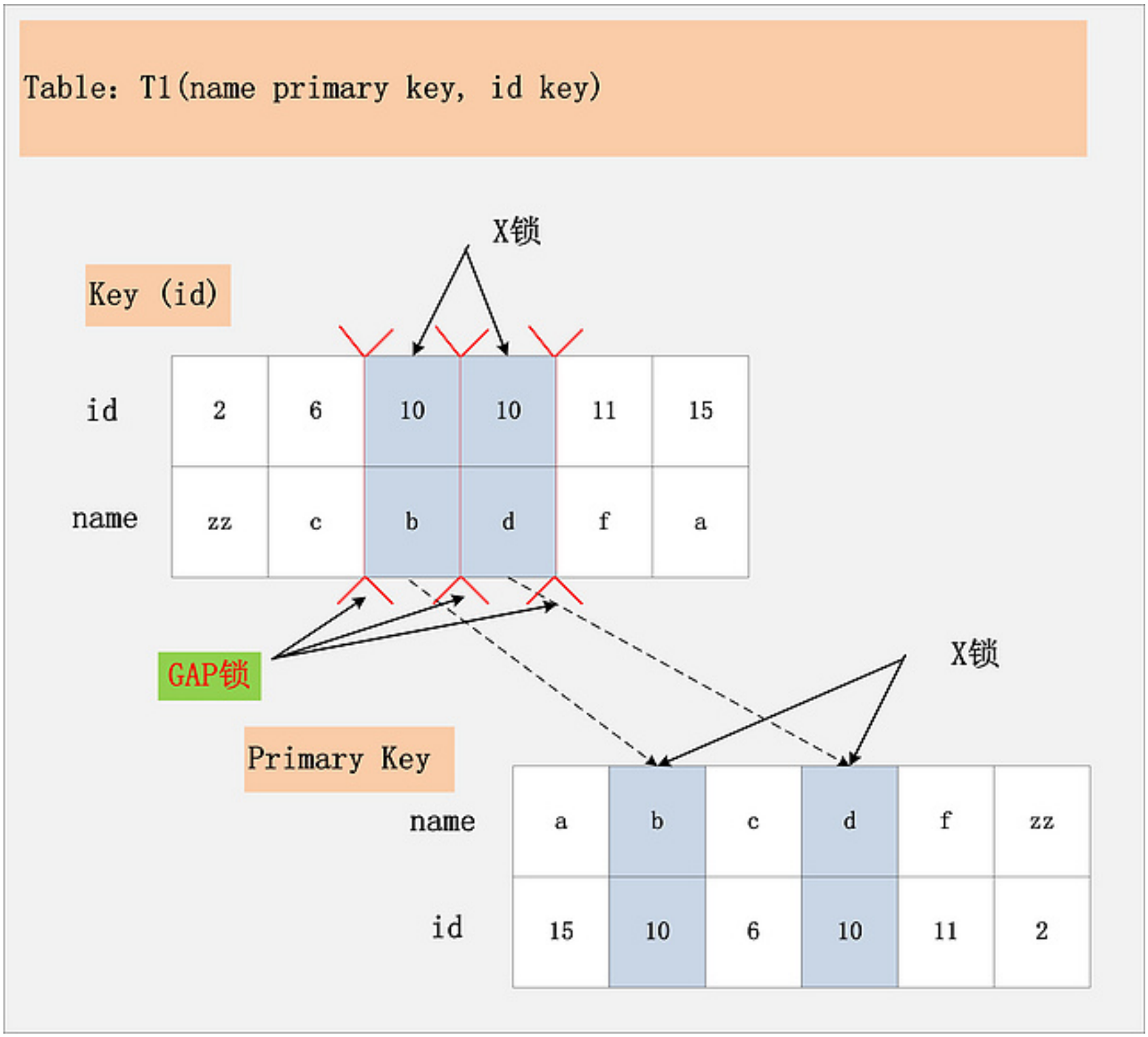
若id列上没有索引，SQL会走聚簇索引的全扫描进行过滤，由于过滤是由MySQL Server层面进行的。因此每条记录，无论是否满足条件，都会被加上X锁。但是，为了效率考量，MySQL做了优化，对于不满足条件的记录，会在判断后放锁，最终持有的，是满足条件的记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略。同时，优化也违背了2PL的约束（同时加锁同时放锁）。

组合5，6同以上（因为只有一条结果记录，只能在上面加锁）

组合7：id列是二级非唯一索引，RR隔离级别

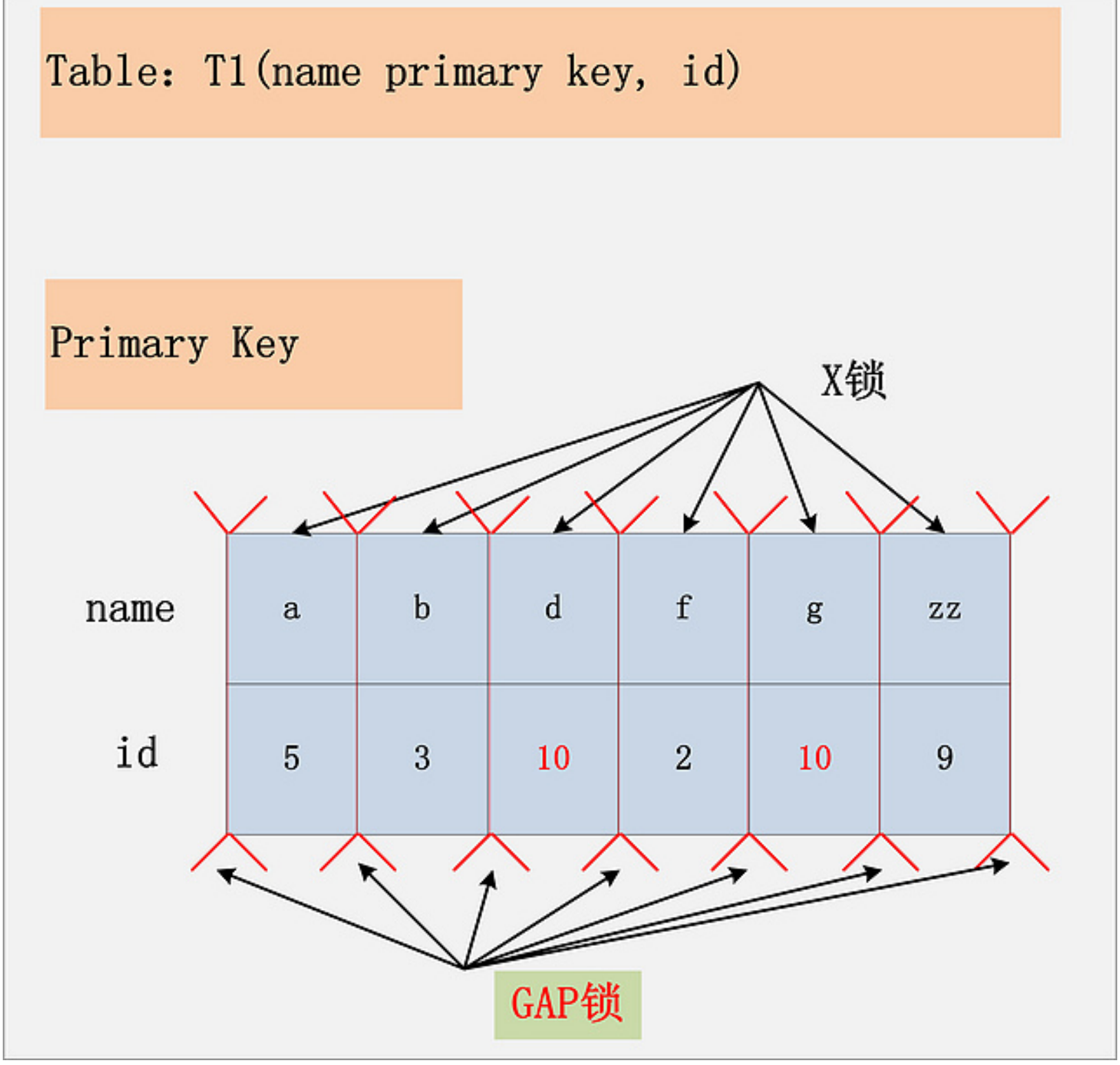
在RR隔离级别下，为了防止幻读的发生，会使用Gap锁。这里，你可以把Gap锁理解为，不允许在数据记录前面插入数据。首先，通过id索引定位到第一条满足查询条件的记录，加记录上的X锁，加GAP上的GAP锁，然后加主键聚簇索引上的记录X锁，然后返回；然后读取下一条，重复进行。

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。如下图所示：



组合8：id列无索引，RR隔离级别

在这种情况下，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。如下图：



但是，MySQL是做了相关的优化的，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，同时也不会添加Gap锁。

组合9：Serializable隔离级别

和RR隔离级别一样。

1.6 一个复杂的SQL的加锁分析

这里我们只是列出一个结论，因为要涉及到MySQL的where查询条件的分析，因此这里先不做详细介绍，我会在之后的博客中详细说明。如下图：

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

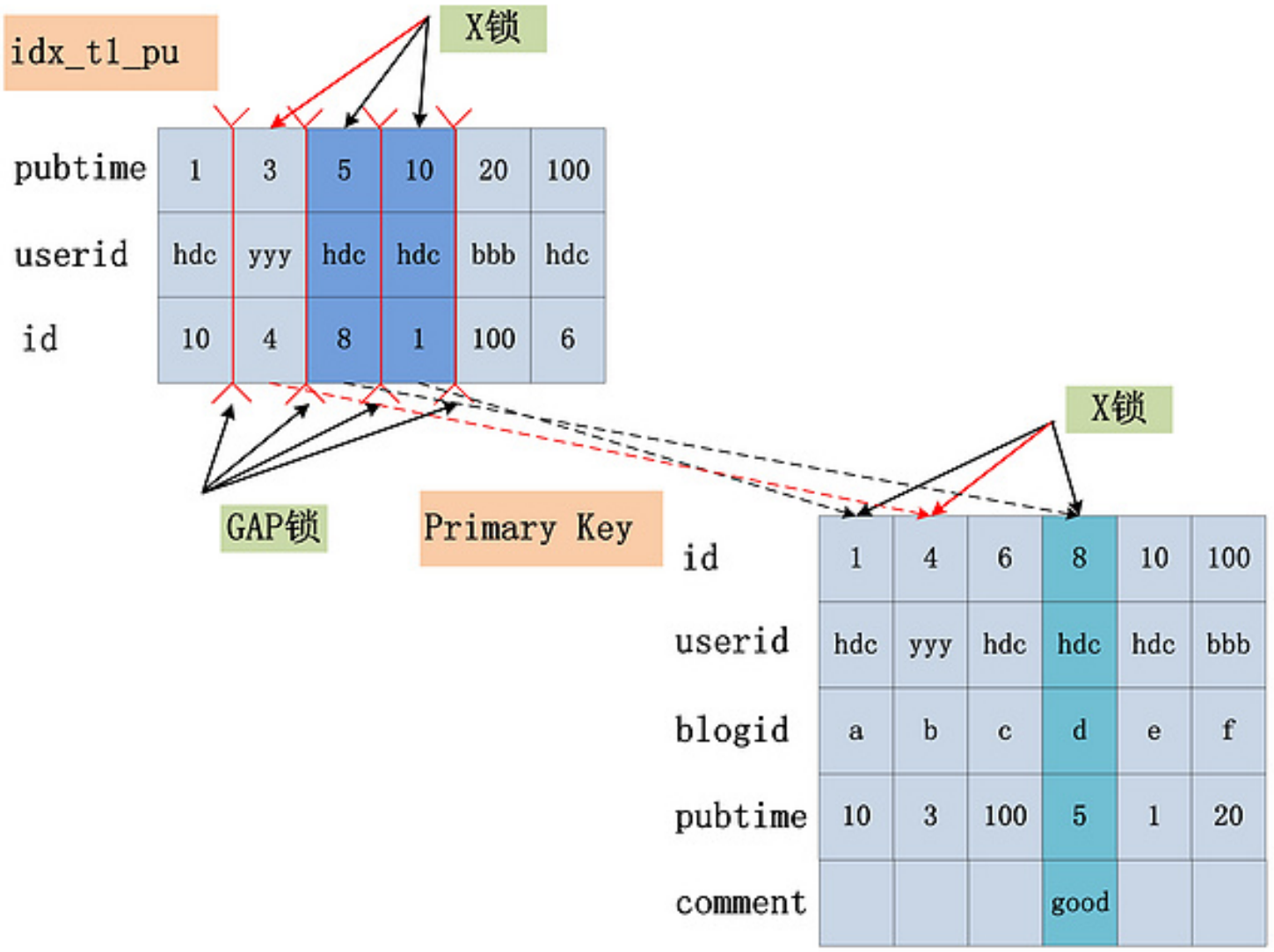
Primary Key

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;
```

结论：在RR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GAP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X锁，否则需要X锁；Table Filter过滤条件，无论是否满足，都需要加X锁。加锁的结果如下所示：

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




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

总结

本文只是对MVCC的一些基础性的知识点进行了详细的总结，参考了网上和书上比较多的资料和实例。希望能对各位的学习有所帮助。