19.MySQL-Innodb锁机制√

1.锁机制的介绍

> 存储引擎层角度：保证了写的隔离
> 整个数据库角度：保护并发访问的资源

2.锁的类型（整个数据库角度）

一 从保护的资源角度分类：

> server层（连接层+SQL层）：
> 1. latch(闩锁)分为两类,主要是为了保证内存资源不会被抢占。（A事务使用指定数据页加载到内存，B事务就不能使用这个数据页）
> - rwlock 写
> - mutex 读
> 2. MDL（metadata_lock）元数据锁 （阻塞DDL与DDL之间，DDL与DML之间，不阻塞DML与DML之间）
> 3. table_lock 表锁
> - 场景一：mysqldump备份非innodb数据时，触发FTWRL全局锁表
> - 场景二：行锁升级为表锁
> engine层：
> row lock 行锁（默认锁粒度） 加锁方式都是基于索引
> 1. record lock 记录锁 RC级别下
> 2. gap lock 间隙锁 RR级别下，防止幻读现象
> 3. next key lock 下一键锁 RR级别下，防止幻读现象
> record lock+gap lock = next key lock

二 从功能角度分类

属于表级别

> 1. IS锁：意向共享锁 （可显示手动添加，或隐式自动触发） 手动添加：`select * from t1 lock in shared mode;`
> 2. IX锁：意向排他锁（添加在表上） 手动添加：`select * from t1 for update;`

意向锁介绍

> 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。
> 意向共享锁（IS锁）：事务在请求S锁前，要先获得IS锁
> 意向共享锁（IS锁）：事务在请求X锁前，要先获得IX锁
> 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。

意向锁问题

q1：为什么意向锁是表级锁呢?

> （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数据行被锁定，提升性能。

q2：意向锁怎么支持表锁和行锁并存?

> 1）首先明确并存的概念是指数据库同时支持表、行锁，而不是任何情况都支持一个表中同时有一个事务A持有行锁、又有一个事务B持有表锁，因为表一旦被上了一个表级的写锁，肯定不能再上一个行级的锁。
> （2）如果事务A对某一行上锁，其他事务就不可能修改这一行。这与“事务B锁住整个表就能修改表中的任意一行”形成了冲突。所以，没有意向锁的时候，让行锁与表锁共存，就会带来很多问题。于是有了意向锁的出现，如q1的答案中，数据库不需要在检查每一行数据是否有锁，而是直接判断一次意向锁是否存在即可，能提升很多性能。

属于行记录级别

> 3. S锁： 共享锁（读锁） 手动添加：`lock table t1 read ;` **多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改**
> 4. X锁 ： 排他锁（写锁） **一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的其他锁，包括共享锁和排他锁，但是获取排他锁的事务是可以对数据就行读取和修改。**

补充：

> DML语句（update,delete,insert）都会自动给涉及到的数据加上排他锁，select语句默认不会加任何锁类型，
> 如果加排他锁可以使用select ...for update语句，加共享锁可以使用select ... lock in share mode语句。
> 所以加过排他锁的数据行在其他事务中是不能修改数据的，也不能通过for update和lock in share mode锁的方式查询数据，但可以直接通过select ...from...查询数据，因为普通查询没有任何锁机制。

3.锁的兼容性表

为什么会出现阻塞，是因为锁类型出现冲突。
compatible 兼容 conflict冲突

> 1.当事务A对某个数据范围（行或表）上了“某锁”后，另一个事务B是否能在这个数据范围上“某锁”。
> 2.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。
> 3.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。
> 4.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，所以即使是行级排他锁，因为表级锁定的行肯定包括行级锁定的行，所以表级X和IX、X都不兼容。
>
> 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。

	行锁		表锁	
	s锁(共享锁)	x锁（排他锁）	IS锁（意向共享锁）	IX锁
s锁(共享锁)	兼容	冲突	兼容	冲突

x锁（排他锁）	冲突	冲突	冲突	冲突
IS锁（意向共享锁）	兼容	冲突	兼容	兼容
IX锁（意向排他锁）	冲突	冲突	兼容	兼容

4.sever层的表锁–MDL锁

4.0 涉及到的英文单词

> intention n.计划
> shared n.共享
> exclusive adj.专用的
> upgradable adj.可升级的

4.1 MDL细分根据锁住的对象分类

我们可以`show full processlist`获取锁等待信息

属性	含义	范围/对象
GLOBAL	全局锁	范围
COMMIT	提交保护锁	范围
SCHEMA	库锁	对象
TABLE	表锁	对象
FUNCTION	函数锁	对象
PROCEDURE	存储过程锁	对象
TRIGGER	触发器锁	对象
EVENT	事件锁	对象

4.2 MDL细分根据锁的持有时间分类

属性	含义
MDL_STATEMENT	从语句开始执行时获取，到语句执行结束后释放
MDL_TRANSACTION	在一个事务中涉及所有表获取DML,一直到事务commit或rollback才释放

MDL_EXPLICIT	需要MDL_context::release_lock()显示释放。语句或者事务结束，也仍然持有。

4.3 MDL细分根据操作类型和对象分类

英文单词

（shared n.共享 intention n.计划）

属性	含义	阻塞实列
MDL_INTENTION_EXCLUSIVE(IX)	意向排他锁用于global和commit的加锁	truncate table t1;(会加 insert into t1 values（ 会加如下锁 global级别：MDL_STA schema级别：MDL_TF
MDL_SHARED(S)	只访问元数据，比如表结构，不涉及数据	设置数据库只读，阻塞 set global_read_only= 会加如下的锁 global（全局锁），MD
MDL_SHARED_HIGH_PRIO(SH)	用于访问information_schema表，不涉及数据	select * from informat show create table xx; 会加如下的锁 table(表锁)，MDL_TRA
MDL_SHARED_READ(SR)	访问表结构并且读表数据	select * from t1; lock table t1 read; 会加如下的锁 table(表锁)，MDL_TRA
MDL_SHARED_WRITE(SW)	访问表结构并且读表数据	insert/update/delete/ 会加如下的锁 table(表锁)，MDL_TRA
MDL_SHARED_UPGRADABLE(SU)	是mysql5.6版本引入新的元数据锁， 在alter table/create index/ drop index 会加该锁，为了ONline ddl 引入特点是是允许DML ,防止DDL操作。	会加如下的锁 table(表锁)，MDL_TRA
MDL_SHARED_NO_WRITE(SNW)	可升级锁，访问表结构并且读写表数据，并且禁止其他事务写	alter table t1 modify c 会加如下的锁 table(表锁)，MDL_TRA

MDL_SHARED_NO_READ_WRITE(SNRW)	可升级锁，访问表结构并且读写表数据，并且禁止其他事务读写	lock table t1 wirite; 会加如下的锁 table(表锁)，MDL_TRA MDL_SHARED_NO_RE
MDL_EXCLUSIVE(X)	级别最高的排他锁，防止其他线程读写元数据	online-DDL copy算法的 锁 会加如下的锁 table(表锁)，MDL_TRA

4.4 几种经典语句加（释放锁的流程）

select语句操作MDL锁流程（执行比较快）

Bash | Copy

```
第一阶段:
1.opening tables阶段（打开表），会加共享锁（不阻塞其他共享锁，阻塞排他锁和意向排他锁）
1.1 加意向排他锁（MDL_INTENTION_EXCLUSIVE） 阻塞共享锁和排他锁
1.2 加共享读锁（MDL_SHARED_READ）
第二阶段
2.事务提交阶段，释放MDL锁
2.1 释放意向排他锁（MDL_INTENTION_EXCLUSIVE）
2.2 释放共享读锁（MDL_SHARED_READ）
```

DML语句操作MDL锁流程

Bash | Copy

```
第一阶段:
1.opening tables阶段（打开表），会加共享锁（不阻塞其他共享锁，阻塞排他锁和意向排他锁）
1.1 加意向排他锁（MDL_INTENTION_EXCLUSIVE） 阻塞共享锁和排他锁
1.2 加共享写锁（MDL_SHARED_WRITE）
第二阶段
2.事务提交阶段，释放MDL锁
2.1 释放意向排他锁（MDL_INTENTION_EXCLUSIVE）
2.2 释放共享写锁（MDL_SHARED_WRITE）
```

alter语句操作MDL锁流程（copy方式）

```
第一阶段
1.opening tables阶段（打开表），会加共享锁（不阻塞其他共享锁，阻塞排他锁和意向排他锁）
1.1 加意向排他锁（MDL_INTENTION_EXCLUSIVE） 阻塞共享锁和排他锁
1.2 加MDL_SHARED_UPGRADABLE锁，升级到MDL_SHARED_NO_WRITE锁
第二阶段
2.操作数据（copy data），不加锁
2.1 创建临时表，重新定义临时表修改后的表结构
2.2从原表读取数据插入到临时表中
第三阶段
3.升级锁操作
3.1 将MDL_SHARED_WRITE升级为MDL_EXCLUSIVE，因为要删除原表，将临时表重命名为原表
4.提交阶段，释放MDL锁
4.1 释放意向排他锁（MDL_INTENTION_EXCLUSIVE）
4.2 释放排他锁（MDL_EXCLUSIVE）
```

4.5 监控分析MDL锁的方法

查看performance_schema系统库下的metadata_locks表

4.6（重点）如何查看分析锁等待

1.查看默认锁等待实际

```
mysql> show variables like '%wait%'; 默认锁等待50秒
+----------------------------------------------------+-----------+
| Variable_name                                      | Value     |
+----------------------------------------------------+-----------+
| innodb_lock_wait_timeout                           | 50        |
```

2.有关于锁的表在 sys系统库下innodb_lock_waits

```
mysql> select * from innodb_lock_waits\G;
*************************** 1. row ***************************
                wait_started: 2021-04-17 21:17:05
                    wait_age: 00:00:48
               wait_age_secs: 48
                locked_table: `world`.`cry`                          查看被锁的表
          locked_table_schema: world
            locked_table_name: cry
       locked_table_partition: NULL
    locked_table_subpartition: NULL
                locked_index: PRIMARY                               innodb加锁是基于索引加锁
                 locked_type: RECORD                                加锁的类型：行锁
              waiting_trx_id: 12832
         waiting_trx_started: 2021-04-17 21:16:30
             waiting_trx_age: 00:01:23
     waiting_trx_rows_locked: 1                                     加锁限定的行数个数
   waiting_trx_rows_modified: 0
                 waiting_pid: 12                                    谁在等待锁操作，谁被锁住了
               waiting_query: update cry set name='jj' where id=1   锁等待的要执行的语句
             waiting_lock_id: 139751497162104:39:4:2:139751390782624
           waiting_lock_mode: X,REC_NOT_GAP
             blocking_trx_id: 12831
                blocking_pid: 8                                     阻塞的人是谁
              blocking_query: NULL
            blocking_lock_id: 139751497162952:39:4:2:139751390788784
          blocking_lock_mode: X,REC_NOT_GAP
        blocking_trx_started: 2021-04-17 21:15:10
            blocking_trx_age: 00:02:43
    blocking_trx_rows_locked: 1
  blocking_trx_rows_modified: 1
     sql_kill_blocking_query: KILL QUERY 8                          处理锁等待方法（谨慎）
sql_kill_blocking_connection: KILL 8                                处理锁等待方法（谨慎）
1 row in set (0.00 sec)
```

3.分析锁等待（连接线程----> sql线程）

Bash | Copy

```
连接线程----> sql线程
1.首先sql> select * from innodb_lock_waits\G;
找出被阻塞和阻塞者的连接线程（show processlist）是谁?
*************************** 1. row ***************************
                wait_started: 2021-04-17 21:17:05
                    wait_age: 00:00:48
               wait_age_secs: 48
                locked_table: `world`.`cry`                          查看被锁的表
         locked_table_schema: world
           locked_table_name: cry
      locked_table_partition: NULL
   locked_table_subpartition: NULL
               locked_index: PRIMARY                                 innodb加锁是基于索引加锁
                locked_type: RECORD                                  加锁的类型：行锁
             waiting_trx_id: 12832
        waiting_trx_started: 2021-04-17 21:16:30
            waiting_trx_age: 00:01:23
    waiting_trx_rows_locked: 1                                       加锁限定的行数个数
   waiting_trx_rows_modified: 0
                 waiting_pid: 12                                     谁在等待锁操作，谁被锁住了
               waiting_query: update cry set name='jj' where id=1    锁等待的要执行的语句
             waiting_lock_id: 139751497162104:39:4:2:139751390782624
           waiting_lock_mode: X,REC_NOT_GAP
             blocking_trx_id: 12831
                blocking_pid: 8                                      阻塞的人是谁
              blocking_query: NULL
            blocking_lock_id: 139751497162952:39:4:2:139751390788784
          blocking_lock_mode: X,REC_NOT_GAP
        blocking_trx_started: 2021-04-17 21:15:10
            blocking_trx_age: 00:02:43
    blocking_trx_rows_locked: 1
   blocking_trx_rows_modified: 1
     sql_kill_blocking_query: KILL QUERY 8                           处理锁等待方法（谨慎）
sql_kill_blocking_connection: KILL 8                                 处理锁等待方法（谨慎）

提取信息被阻塞的是12（连接线程id号），发起阻塞的是8（连接线程id号）

2.通过performance_schema下的threads找到连接线程和sql线程的对应关系
select * from performance.threads\G;

       THREAD_ID: 52
                 NAME: thread/sql/one_connection
                 TYPE: FOREGROUND
      PROCESSLIST_ID: 12
    PROCESSLIST_USER: root
    PROCESSLIST_HOST: localhost
      PROCESSLIST_DB: world
PROCESSLIST_COMMAND: Sleep
    PROCESSLIST_TIME: 585
   PROCESSLIST_STATE: NULL
```

```
    PROCESSLIST_INFO: update cry set name='jj' where id=1        显示会话做的最后一条sql语句
    PARENT_THREAD_ID: NULL
                ROLE: NULL
        INSTRUMENTED: YES
             HISTORY: YES
     CONNECTION_TYPE: Socket
        THREAD_OS_ID: 3863
      RESOURCE_GROUP: USR_default

提取信息：找到THREAD_ID: 52（sql线程id号）

3.通过查询 performance_schema .events_statements_history 中对应的线程id号，得到执行sql的历史记录进行分析
select * from performance_schema.events_statements_history  where thread_id=52\G;

```

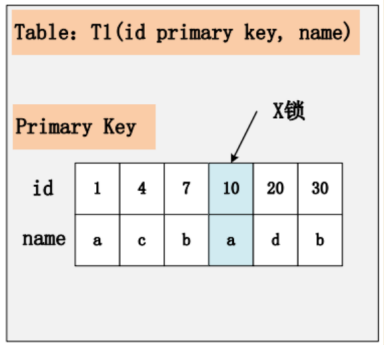
5.engine层的行锁

不同隔离级别下的锁处理

隔离级	脏读可能性	不可重复读可能性	幻读可能性	加锁读
READ UNCOMMITTED	是	是	是	否
READ COMMITTED	否	是	是	否
REPEATABLE READ	否	否	是	否
SERIALIZABLE	否	否	否	是

RC级别

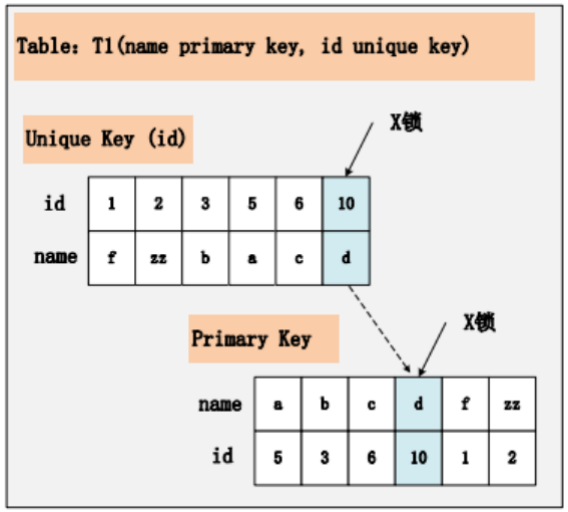
```
这个组合，是最简单，最容易分析的组合。id 是主键，Read Committed 隔离级别，
给定 SQL： delete from t1 where id = 10; 只需要将主键上，id = 10 的记录加上 X 锁即可。如下图所示：
```

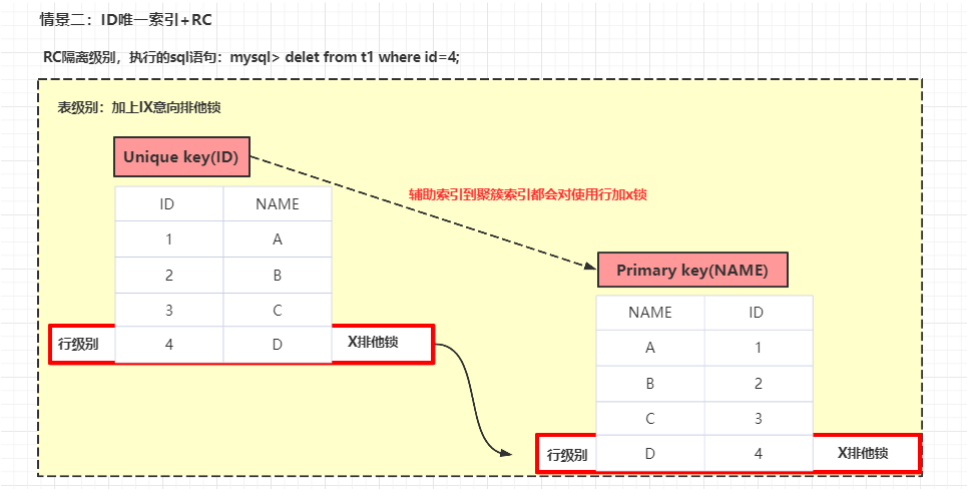


情景二：id 唯一索引+RC

Bash | Copy

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：
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。

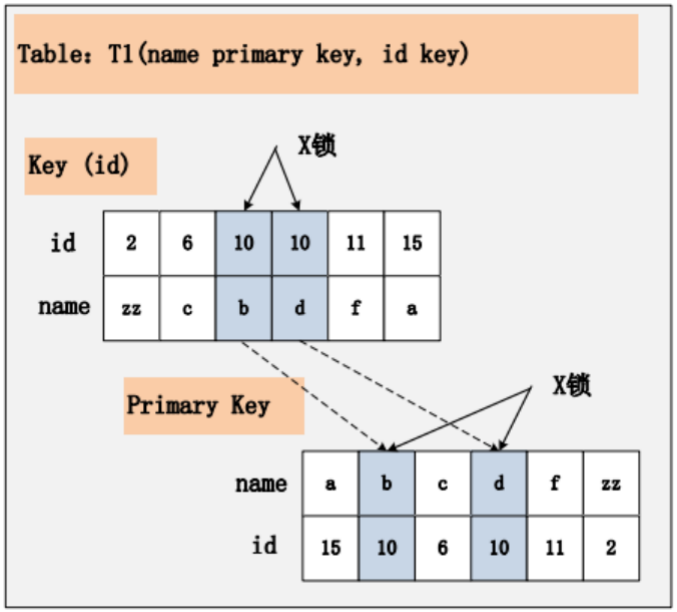




情景三：id 非唯一索引+RC

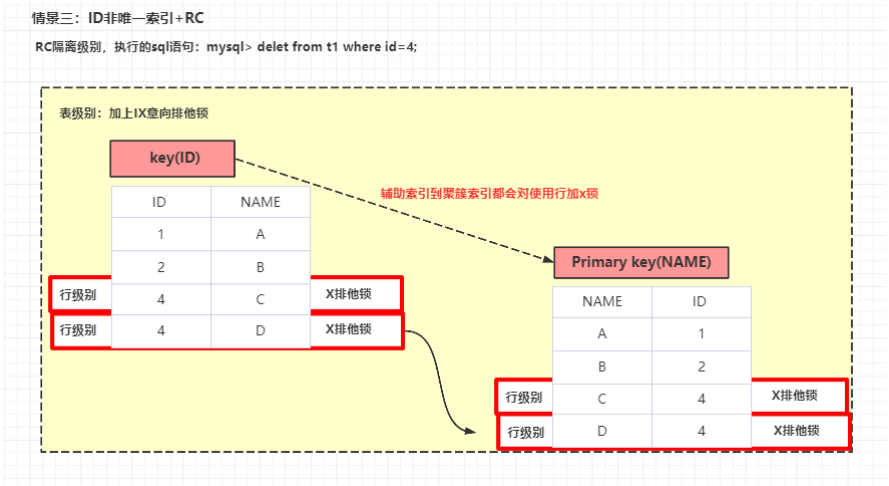
Bash | Copy

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：



Bash | Copy

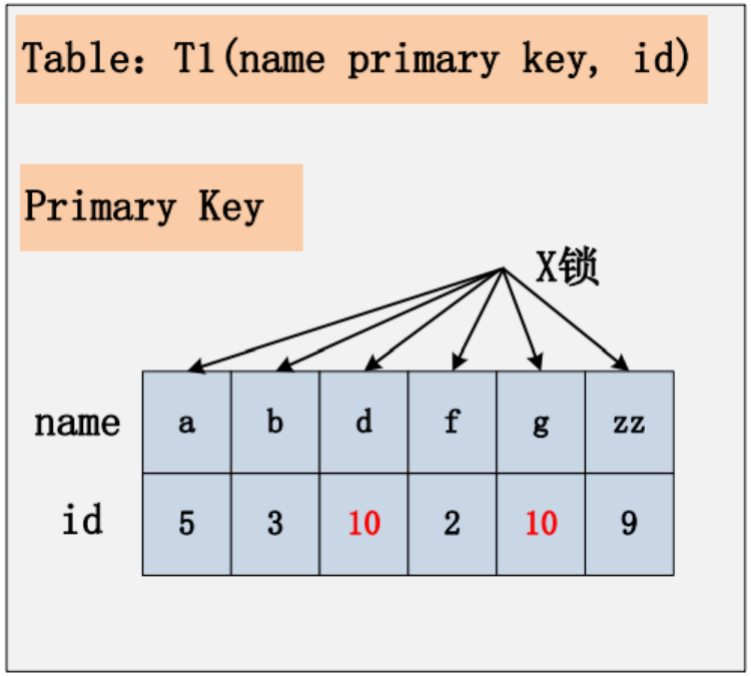
若 id 列上有非唯一索引，那么对应的所有满足 SQL 查询条件的记录，都会被加锁。 同时，这些记录在主键索引上的记录，也会被加锁。



情景四：id 无索引+RC

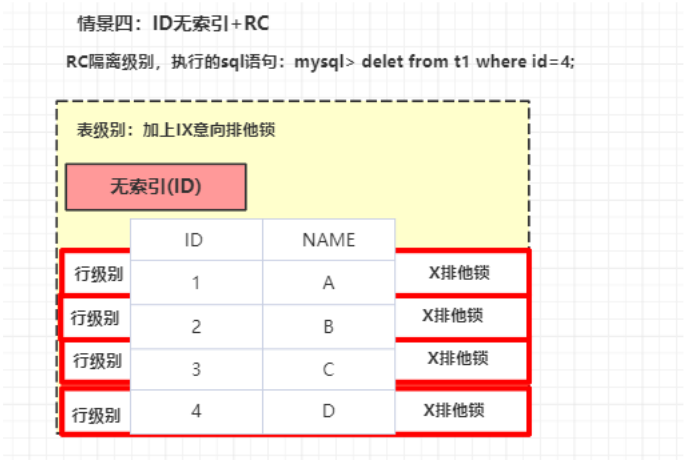
Bash | Copy

```
1  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：
```



Bash | Copy

```
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
```



RR级别

RR级别加锁原则

Bash | Copy

```
原则一：查找过程中访问到的索引对象才会加锁
原则二：加锁的基本单位是Next-key lock ,net-key lock 的阻塞空间区域是左开右闭 （3，8】 3不会被阻塞，4，5，6，7，8会被阻塞
原则三：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会退化为间隙锁
原则四：索引上的等值查询（where条件），给唯一索引加锁的时候，next-key lock退化为行锁

8.0.19 之前的bug ：  唯一索引上的范围查询会访问到不满足条件的第一个值为止。
```

RR级别下加锁原则图

情景五：id 主键+RR

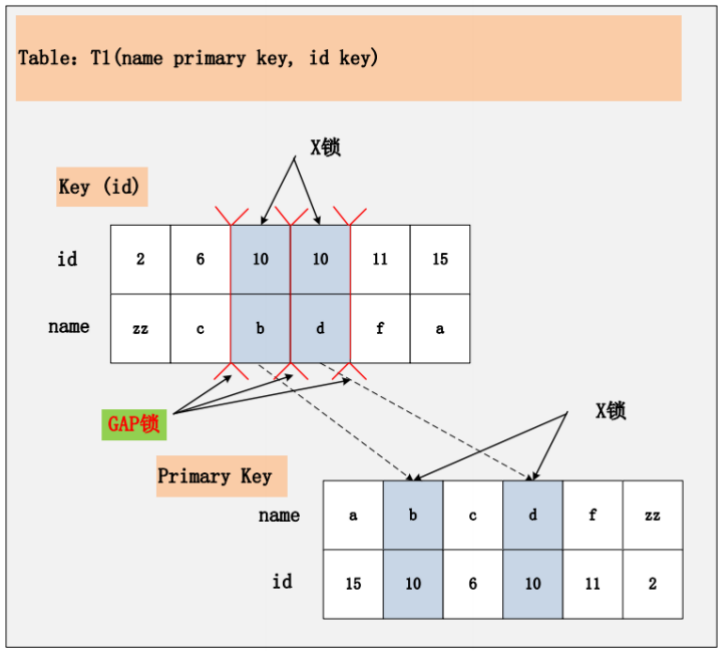
```
id 列是主键列，Repeatable Read 隔离级别，针对 delete from t1 where id = 6;
这条 SQL，加锁与组合一：[id 主键，Read Committed]一致。
```

情景六：id 唯一索引+RR

```
1 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。 注：id 为唯一索引，针对 id 的并发等值删除操作，有可能会产生死锁。（优化章节案例讲解）
```

情景七：id 非唯一索引+RR

```
1 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：
```

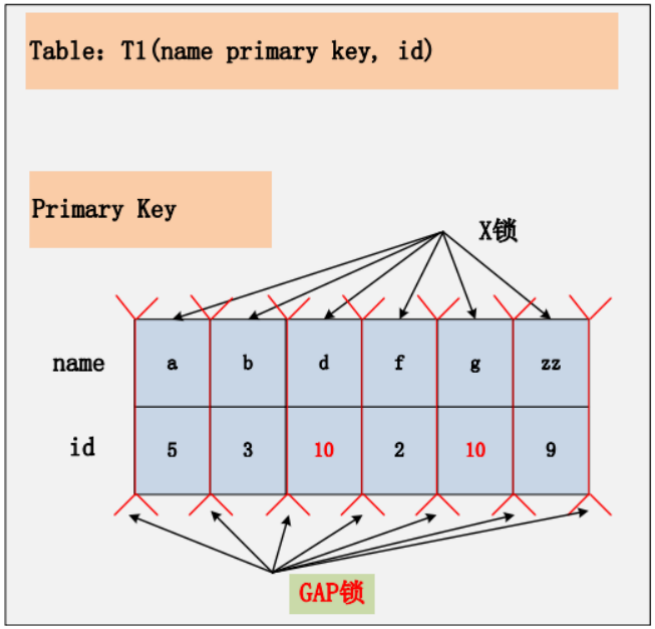



```
1 Repeatable Read 隔离级别下，id 列上有一个非唯一索引，对应 SQL：delete from t1 where id = 10; 首先，通过 id 索引定位到第一条满足查询条件的记录，加记录上的 X 锁，加 GAP 上的 GAP 锁，然后加主键聚簇索引上的记录 X 锁，然后返回；然后读取下一条，重复进行。直至进行到第一条不满足条件的记录[11,f]，此时，不需要加记录 X 锁，但是仍旧需要加 GAP 锁，最后返回结束。
```

情景八：id 无索引+RR

Bash | Copy

```
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：
```



Bash | Copy

```
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。
```

Serializable级别

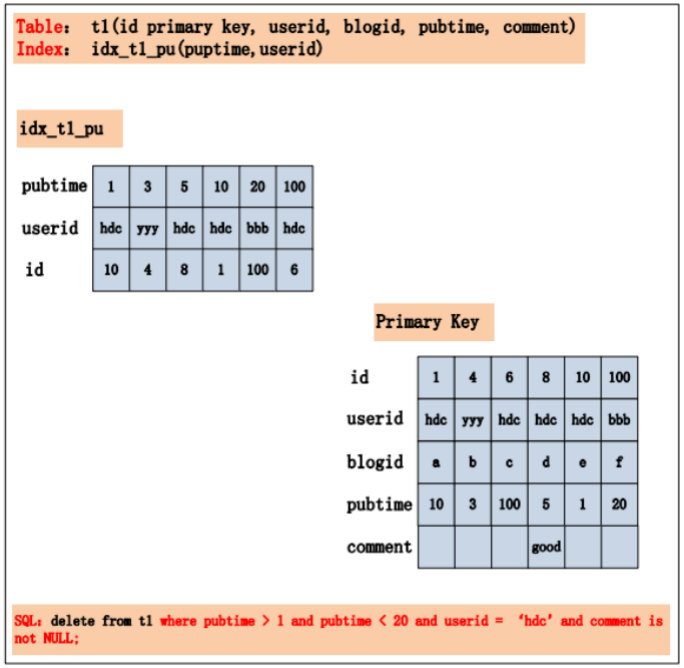
情景九：

Bash | Copy

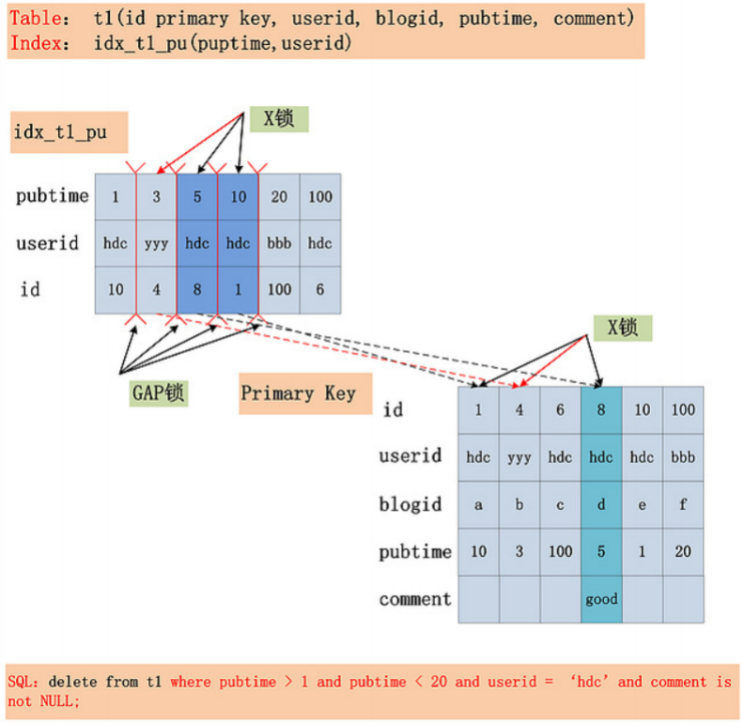
```
针对前面提到的简单的 SQL，最后一个情况：Serializable 隔离级别。对于 SQL2：delete from t1 where id = 10; 来说，Serializable 隔离级别与 Repeatable Read 隔离级别完全一致， 因此不 做介绍。 Serializable 隔离级别，影响的是 SQL1：select * from t1 where id = 10; 这条 SQL，在 RC，RR 隔离级别下，都是快照读，不加锁。但是在 Serializable 隔离级别，SQL1 会加读锁，也就是 说快照读不复存在，MVCC 并发控制降级为 Lock-Based CC。 结论：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。
```

扩展

```
1  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。
```




```
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。 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。
```



案列（RR级别）

案列一：

0.创建一个模拟表

Bash | Copy

```
创建表
create table t1(id int(11) primary key auto_increment not null,cname int(11) not null ,snum int(11) not null
插入数据
insert into t1 values(0,0,0),(5,5,5),(10,10,10),(20,20,20),(25,25,25);
查看表
mysql> select * from t1;
+----+-------+------+
| id | cname | snum |
+----+-------+------+
|  1 |     0 |    0 |
|  5 |     5 |    5 |
| 10 |    10 |   10 |
| 20 |    20 |   20 |
| 25 |    25 |   25 |
+----+-------+------+
```

1.模拟操作，记录操作结果

打开三个窗口，在同一事务内我们用表格的方式记录每个窗口进行了什么操作

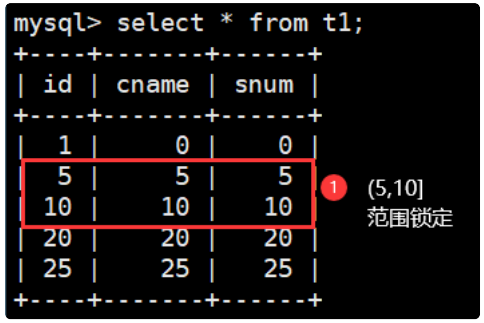
窗口一	窗口二	窗口三
进行对t1表，id=7的行进行修改数据 update t1 set d=d+1 where id=7	无操作	无操作
无操作	进行对t1表，插入id=8的行数据 insert into t1 values(8,8,8,); 结果被阻塞	无操作
无操作	无操作	进行对t1表，插入id=10的行 insert into t1 values(10,10, 结果不被阻塞

2.分析加锁原理

1. 窗口一 `update t1 set d=d+1 where id=7`

RR级别下默认加next-key lock

原则二 加上了（5,10] 的next-key lock ，但是更新选择id=7 7在索引列中没有不是等值查询所以不会退化为行锁，不满足原则四



2.窗口二 `insert into t1 values(8,8,8,);`

因为上面加上了（5,10] 的next-key lock 所以进行了阻塞

3. 窗口三 `insert into t1 values(10,10,10);`

原则三 id=10 是索引上的等值查询 所以next-key lock 退化为间隙锁， （5,10） 所以不会被阻塞

Innodb%E9%94%81%E6%9C%BA%E5%88%B6%E2%88%9A%20%7C%201.%E9%94%81%E6%9C%BA%E5%88%B6%E7%9A%84%E4%BB%8B%E7%BB%8D%E5%AD%98%E5%82%A8%E5%BC%95%E6%9