LBCC + MVCC【已总结】

由于Mysql 默认的隔离级别是Repeatable Read、所以本篇文章的部分结论是基于RR隔离级别得出

事务具有四大特性 ACID：

原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）

关于原子性、一致性、和持久性是如何保证的，在前面文章中已经介绍：**传送门**

本文只介绍事务隔离级别相关的特性

首先需要了解一下违反事物隔离级别的几个特征

1. 脏读

   当前事务中读取到了其它没有提交的事务修改的数据———读到了未提交的数据

2. 不可重复读（针对修改或删除的情况）

   同一个事务中分别两次对同一条数据进行了查询，发现数据被更改（修改/删除）了，且这个更改是别的事务更改的———同一个事务两次读取结果不一致

3. 幻读（针对新增的情况）

   幻读和不可重复读很像，同样是同一个事务中两次查找结果不一致现象，但是幻读只是针对于新增数据的情况，如：第一次查找和第二次查找中间别的事务可能新增了数据，导致两次结果不一致的情况

为了避免出现上的情况，SQL规范中定义了四种事务隔离级别，并且Mysql都支持

1. Read Uncommitted (读未提交)

   脏读、不可重复读、幻读 都没有解决、一般用不到

2. Read Committed (读已提交)

   只解决了脏读的问题、没有解决不可重复读和幻读问题，是Oracle的默认隔离级别

3. Repeatable Read (可重复读)

   解决了脏读和不可重复读问题、但没有解决幻读问题，是Mysql的默认隔离级别

   但是Mysql是具体怎么解决幻读问题的（锁控制），后面会介绍。

4. Serializable (串行化)

   脏读、不可重复读、幻读问题都有解决，但是很影响性能，一般也不用

事务隔离级别的实现方式（LBCC+MVCC）

LBCC：Lock-Based Concurrent Control　基于锁并发的控制实现

MVCC：Multi-Version Concurrent Control 基于多版本快照的实现

LBCC+MVCC 在满足隔离级别的基础上，同样也保证了数据库的性能

其中普通select语句均是snapshot read（MVCC）

而delete/update/select for update等语句是加锁实现的current read（LBCC）

LBCC：基于锁并发的控制实现

首先看一下锁的分类：

按照模式划分，可分为共享锁、排它锁、意向锁（意向共享锁、意向排它锁）、自增锁

- 共享锁（行锁）—— Shared Locks（简称S锁）：

  说明：InnoDB引擎的行锁机制锁的是索引，不是行记录

  又叫读锁，多个事务可以共用同一把共享锁读取数据，但是无法修改，想要修改，必须所有的共享锁释放完成之后才可以。

  共享锁加锁方式： select * from table where id = 1 lock in share mode

  释放锁的方式：Commit 提交、Rollback 回滚

- 排它锁（行锁）——Exclusive Locks （简称X锁）

  一个事务对某一个资源上了排它锁后，只允许当前事务对其进行增删改查，其它人无法对当前资源进行任何操作，包括读操作。**即排它锁和其他锁匙互斥关系**

  排它锁加锁方式：

自动：DML语句默认会自动加锁

手动：select * from student where id = 1 for update

释放锁的方式：Commit 提交、Rollback 回滚

- 意向锁（表锁）—— Intention Locks：

  意向锁不是用来锁定数据的，而是用来告诉这个表中是否加了排他锁、共享锁，这样以后再创建表锁的时候不用去扫描表中排它锁、共享锁的状态，直接根据意向锁状态就可以知道是否可以创建，可以理解成一个标记。目的是为了提升加表锁的效率。

- 自增锁（表锁） Auto-inc Locks：

  当向使用含有AUTO_INCREMENT列的表中插入数据时需要获取的一种特殊的表级锁

  自增锁并不是事务锁，为了保证效率，自增锁的释放可以通过一个参数innodb_autoinc_lock_mode控制

  该参数可以设置三个值 0 | 1 | 2，三个值分别代表的意思如下：

  0：语句执行结束后才释放锁

  1：普通insert语句，申请完锁后就释放、insert ... select 语句执行结束后释放，保证顺序

  2：申请后就释放锁

  具体细节可以参考：
  https://blog.csdn.net/qq_40378034/article/details/90736544

按照粒度划分，可以划分为表锁和行锁

如果操作的数据 where 条件不是索引的话，是不可以添加行锁的，会进化成表锁（相对）。

当不是索引的时候，Mysql会为表中的所有行加锁，上锁之后会过滤掉不满足条件的记录，最终留下的是满足条件的记录。即便是有一个过滤操作，但是加锁/释放锁的过程对性能消耗也是很大的。

即使抛开上锁和释放锁的这个过程，假设一个表中满足where条件的记录很多的时候，别的事务在操作这个范围的事务的时候也是不可以操作的。

如： select * from table where name = '张三' for update;

如果name字段不是索引，加锁范围会扩大，性能会降低，所以在实际应用场景中应该合理的使用索引或给常用字段添加索引。即加锁的时候一定要考虑是否是索引列，避免进化成表级锁

锁的算法：

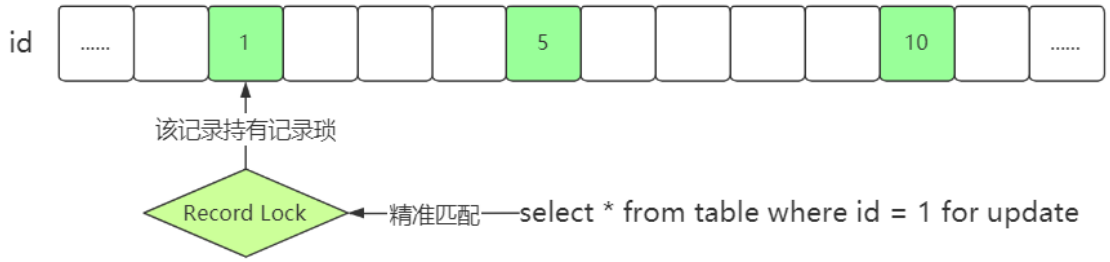
锁的算法分为 记录锁（Record Locks）、临键锁（Next-Key Locks）、间隙锁（Gap Locks）

说明：以下内容均是where 条件中的字段为索引的情况

- 记录锁（Record Locks）：

  当使用条件是等值查询记录的时候的场景的时候会出现记录锁

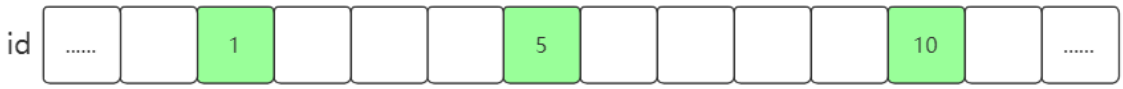
  如：当查询某一个已存在的id 为1 的记录时：



- 间隙锁（Gap Locks）：

  范围查询的时候一般会出现它，他是innodb独有的　只存在于可重复读的隔离级别

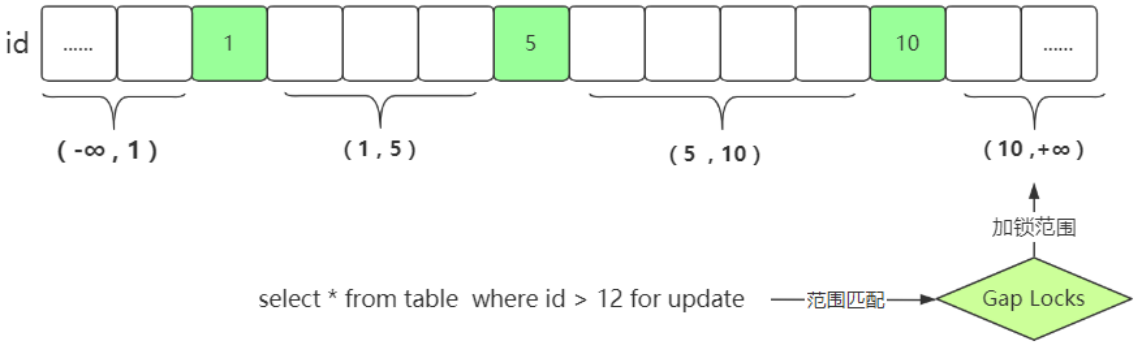
  注：间隙是根据数据库的记录划分的，不是根据条件划分的



如有一个SessionA：

select * from table where id > 12 for update

上面语句实际锁住的区间是 大于12 的区间，即：



假设SessionA 还没有Commit 或者Rollback，这时候有一个 Session B：

update table set name = 'goudan' where id = 20

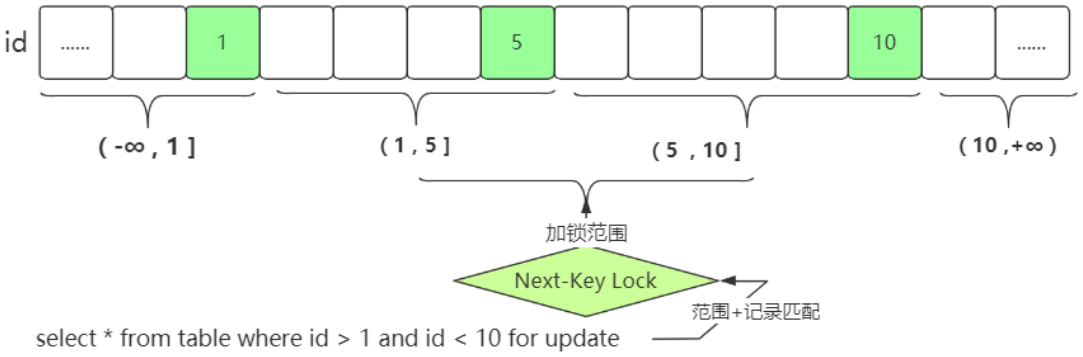
这个情况下 SessionB 这条语句是执行不成功的，只可以修改 ≤ 10 这个范围的记录

- 临键锁（Next-Key Locks）：

临键锁，锁的是左开右闭的区间，相当于间隙 + 记录

如：select * from table where id > 1 and id < 10 for update

执行上面语句时，锁住的区间为 (1, 5] 和 （5，10]，即：



MVCC：基于多版本快照的实现

多版本并发控制，主要为了提升并发性能的考虑，通过行级锁的变种，避免了很多情况下加锁的控制而增加的开销，其次MVCC只适用于读已提交和可重复读两个隔离级别下工作，其它的两个隔离级别是不兼容的。

在数据库中每一个表最后面会隐藏三个列，分别是TRX_ID和ROLL_PTR，它们分别是记录事务ID和版本链的引用。如果表中没有主键，也会隐藏一个主键列。

ID	NAME	.....	TRX_ID	ROLL_PTR
1	zhangsan	.....	10	#xxxx

TRX_ID：当插入/更新/删除一条记录的时候会把当前事务的id写到这条sql语句上去，每次+1

ROOL_PTR：当对记录进行修改操作的时候，会插入一条更新后的数据，然后将旧的记录放到undo log版本链中。

> 如下灰色部分标识历史版本数据，MVCC 就是通过当前事务环境下生成的一致性视图来实现数据查找的。在进行普通select查询的时候，会将当前环境下的所有活跃事务和最大事务做一个快照（readview），然后通过事务ID进行数据查找

在读未提交隔离级别下，通过读取最新的数据即可，对于串行化隔离级别来说本身就是通过互斥锁来访问数据的，压根用不倒MVCC。这也就是前面提到的MVCC只有在读已提交合可重复读两个隔离级别下工作的原因。

所以只需要关注不可重复读和读已提交两种隔离级别即可

这两种隔离级别下主要的问题是怎么保证历史版本数据对当前事务的可见性，所以MVCC的实现是select操作的时候做了一个活跃事务列表的快照，mvcc 就是通过这个事务列表到 undo log 版本链中查找满足条件的数据的。

在当前事务中具体满足什么样的条件才是合理的数据，才可以保证可见性呢？这个问题在这两种隔离级别下是有差别的，以下内容只针对Repeatable Read 隔离级别说明。

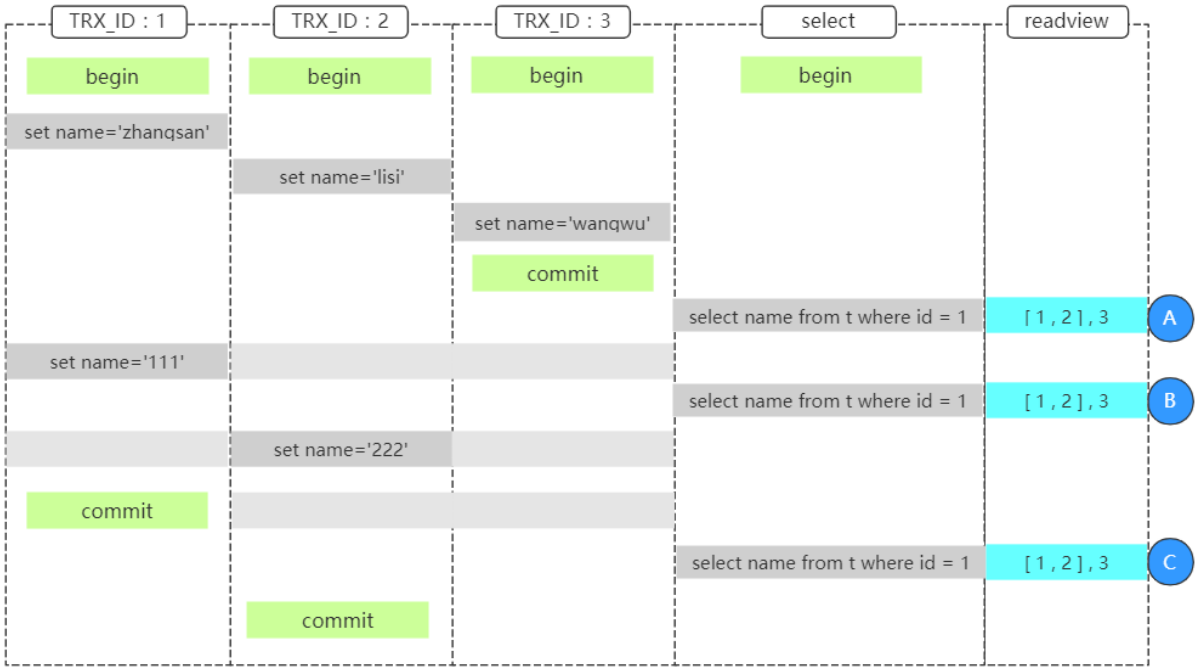
在RR隔离级别下当mysql 在当前事务操作中进行第一次查询操作的时候，会生成一致性视图readview。是通过未提交事务和最大事务ID形成的快照列表，在进行数据查找的时候有一个版本链对比规则：

1. 当前trx id < 最小事务id，说明当前事务时最早进来的，说明当前数据是可见的
2. 当前trx id > 最大事务id，说明是其它事务生成的时候，后来生成的事务id，肯定是不可见的
3. 如果当前在最大事务和最小事务中间，则分两种情况

   a、row 的 trx id 在最大事务和最小事务中间，表示当前版本是在还没有提交的事务的基础上提交的，所以不可见

   b、row 的trx id 不在活跃事务id 范围内，说明当前版本是已经提交了事务id 的，说明可见

可以参考下图进行理解：



在 A 语句执行的时候会生成当前会话的 readview，由于 TRX 3 已经commit，所以最大事务为 3，而 TRX 1 和 TRX 2 都没有提交，所以活跃事务列表为 1 2 ，所以最大

事务ID 为 3 。

根据版本连对比规则，可以不难的出 A 、 B、 的执行结果 均是 wangwu，又因为在同一个Session 中 一致性事务 readview 是一致的，所以C 的执行结果和上面的一样也是 wangwu。

对于删除的情况，可以认为是update的修改操作：

在进行删除操作的，会复制一份最新的数据到版本链，并修改事务ID为delete操作的事务id，同时在当前记录的 header 中添加一个标记为 （deleted flag= true），用来标识当前记录被删除，在进行查询的时候如果检查到 deleted flag 为true的情况下，表示当前记录已经被删除，则不返回数据

LBCC

MVCC