用友云
yonyou cloud

浅谈MySql的事务与锁

日期：2020年06月18日

用友云
yonyou cloud

目录

CONTENS

1 事务

事务--概念和四要素

用友云
yonyou cloud

事务

事务是一个工作单位，它必须使用一个持久性资源(诸如数据库或消息队列)来完成所有的工作。一般来说，单个事务只可能在一个连接中完成。

四要素（简称ACID属性）：

原子性（Atomicity）：事务是一个原子操作单元，其对数据的修改，要么全部执行，要么全都不执行；

一致性（Consistent）：在事务开始和完成时，数据都必须保持一致状态；

隔离性（Isolation）：数据库提供一定的隔离机制（锁机制）,保证事务在不受外部并发操作影响的“独立”环境执行；

持久性（Durable）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

事务—并发问题

用友云
yonyou cloud

示例表：

构造一张表：

主键id

姓名name

年龄age

其中name为普通索引（二级索引）。

看一下在事务并发处理数据时，可能存在哪些问题（先不考虑事务隔离级别）。

id	name	age
1	zg	36
2	zt	34
3	qq	8
4	mm	3

事务—并发问题—脏读（dirty read）

用友云
yonyou cloud

事务 1 没有提交的数据，被事务 2 读到了，属于脏读。

事务 1	事务 2
`update user set age =age+1 where name='qq';`	
	`select age from user where name='qq'; --9`
rollback;	
	`select age from user where name='qq'; --8`

事务一并发问题一不可重复读（unrepeatable read）

用友云
yonyou cloud

事务 1	事务 2
	`select age from user where name='qq'; --8`
`update user set age =age+1 where name='qq';`	
commit;	
	`select age from user where name='qq'; --9`

事务2对同一条记录读取两边，两次读出来的结果不一样。不可重复读与脏读的区别在于，脏读是读取了另外一个事务未提交的修改，而不可重复读是读取了另外一个事务提交之后的修改，这就是不可重复读。

事务一并发问题一幻读（phantom read）

用友云
yonyou cloud

事务 1	事务 2
	`select age from user where id > 2'; -- id=3,4 的两条记录`
`insert into user values(5,'dd',0);`	
commit;	
	`select age from user where id > 2'; -- id=3,4,5 的三条记录`

同样的条件，第一次和第二次读出来的记录数不一样。幻读和不可重复读的区别在于，后者是两次读取同一条记录，得到不一样的结果；而前者是两次读取同一个范围内的记录，得到不一样的记录数（这种说法其实只是便于理解，但并不准确，因为可能存在另一个事务先插入一条记录然后再删除一条记录的情况，这个时候两次查询得到的记录数也是一样的，但这也是幻读，所以严格点的说法应该是：两次读取得到的结果集不一样）。很显然，不可重复读是因为其他事务进行了 UPDATE 操作，幻读是因为其他事务进行了 INSERT 或者 DELETE 操作。

事务—并发问题—丢失更新（lost update）

用友云
yonyou cloud

上面的场景，都是一个事务写，一个事务读，由于一个事务的写，导致了另一个事务读到了不应该读到的数据。如果两个事务都是写，会有什么问题?

两个事务同时对name=’qq’的年龄做修改，第一次查出是8，然后事务修改为9并提交。这时age=9，这时事务1不知道name=’qq’的age已变动，而是继续在之前8的基础进行-1修改并提交事务。这样事务2的提交被覆盖了，最后name=’qq’的用户，age=7。也就是说事务1的提交覆盖了事务2的提交。事务2的update操作完全丢失了。

事务 1	事务 2
`select age from user where name='qq'; --8`	`select age from user where name='qq'; --8`
`insert into user values(5,'dd',0);`	
	`update user set age =8+1 where name='qq'; --age=9`
	`commit;`
`update user set age =8-1 where name='qq'; --age=7`	
`commit;`	

事务—并发问题—丢失更新（lost update）

用友云
yonyou cloud

典型场景：

秒杀的超卖问题。2件商品秒杀，下单后减1，流程如下：

1. 线程A查询库存为2.
2. 线程B查询库存为2.
3. 线程A下单，库存>0，开始下单.
4. 线程B下单，库存>0，开启下单.
5. 线程A下单完成，更新库存2- 1=1
6. 线程B下单完成，更新库存2- 1=1
7. 线程C查询库存=1，下单成功，更新库存1- 1=0.

实际2件库存，但却成交3个订单。由于最后一步是提交操作，这种场一般被称为**提交覆盖**

事务—并发问题—丢失更新（lost update）

和提交覆盖相对的，还有另外一种场景，叫丢失更新，也叫回滚覆盖。流程如下：

事务 1	事务 2
`select age from user where name='qq'; --8`	`select age from user where name='qq'; --8`
	`update user set age =8+1 where name='qq'; --age=9`
	`commit;`
`rollback; --回滚，age=8`	

操作和提交覆盖情景基本上一样，只是最后一步事务 1 的提交变成了回滚，这样 name='qq'的age恢复成原始值 8，事务2的 UPDATE 操作完全没有生效。一个事务的回滚操作竟然影响了另一个正常提交的事务，就应该是bug了。因此几乎所有的数据库都不允许回滚覆盖。

有时候我们把回滚覆盖称之为 **第一类丢失更新** 问题，提交覆盖称为 **第二类丢失更新** 问题。

2

隔离级别

隔离级别

用友云
yonyou cloud

隔离级别作用：

数据库为了解决并发问题，就要使用锁，比如一个事务在读写数据库加锁，不允许第二个事务读写，这样就不会导致数据的异常。

如隔离级别中的序列化级别，就能保证数据安全运行，数据一致性得到保证。但这样会大大的降低事务的并发能力，性能最低。

为平衡数据安全性和数据库的并发性能，就有了四种不同的隔离级别。

read uncommited--读未提交

该隔离级别指即使一个事务的更新语句没有提交，
但是别的事务可以读到这个改变，几种异常情况都可能出现。极易出错，没有安全性可言，基本不会使用。

read committed --读已提交

该隔离级别指一个事务只能看到其他事务的已经提交的更新，看不到未提交的更新，消除了脏读和第一类丢失更新，这是大多数数据库的默认隔离级别，如Oracle,Sqlserver。

repeatable read --可重复读

该隔离级别
指一个事务中进行两次或多次同样的对于数据内容的查询，得到的结果是一样的，但不保证对于数据条数的查询是一样的（幻读），只要存在读该行数据就禁止写，消除了不可重复读和第二类更新丢失，这是Mysql数据库的默认隔离级别。

serializable --序列化读

意思是说这个事务执行的时候不允许别的事务并发写操作的执行.完全串行化的读，只要存在读就禁止写,但可以同时读，消除了幻读。这是事务隔离的最高级别，虽然最安全最省心，但是效率太低，一般不会用。

隔离级别

用友云
yonyou cloud

事务 4 种隔离级别比较：

隔离级别/读数据一致性及允许的并发副作用	读数据一致性	脏读	不可重复读	幻读	第一类丢失更新（回滚覆盖）	第二类丢失更新（提交覆盖）
未提交读（Read uncommitted）	最低级别，读数据不加锁	是	是	是	否	是
已提交度（Read committed）	语句级	否	是	是	否	是
可重复读（Repeatable read）	事务级	否	否	是	否	否
可序列化（Serializable）	最高级别，事务级	否	否	否	否	否

基于锁的并发控制（传统）：

实现了并发的读读，但并发读写还有冲突。并发性能不好。

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

InnoDb 通过 MVCC 实现了读写并行，但在不同的隔离级别下，读的方式也是有所区别。在 read uncommit 隔离级别下，每次都是读取最新版本的数据行，所以不能用 MVCC 的多版本，而 serializable 隔离级别每次读取操作都会为记录加上读锁，也和 MVCC 不兼容，所以只有 RC 和 RR 这两个隔离级别才有 MVCC 。

隔离级别—RR与RC对比—RR模式

用友云 yonyou cloud

示例1：隔离级别设置为 RR。

1.会话 1 执行：

```
mysql> select * from user;
+----+------+------+
| id | name | age  |
+----+------+------+
|  1 | zg   |   36 |
|  2 | zt   |   34 |
|  3 | qq   |    8 |
|  4 | mm   |    3 |
+----+------+------+
4 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from user where id =3;
+----+------+------+
| id | name | age  |
+----+------+------+
|  3 | qq   |    8 |
+----+------+------+
1 row in set (0.00 sec)

mysql> update user set age = age +1 where id =3;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from user where id =3;
+----+------+------+
| id | name | age  |
+----+------+------+
|  3 | qq   |    9 |
+----+------+------+
1 row in set (0.00 sec)

mysql>
```

2.会话 2 执行：(开启了事务：begin)

```
mysql> select * from user where id =3;
+----+------+------+
| id | name | age  |
+----+------+------+
|  3 | qq   |    8 |
+----+------+------+
1 row in set (0.00 sec)
```

3.会话 2 的 age 为 name='qq'的原始信息 8。这时会话 1 使用 commit 提交：

```
mysql> commit;
Query OK, 0 rows affected (0.01 sec)
```

4.使用会话 2 再查询一次，查出的结果不变，还是原始值 age=8。

```
mysql> select * from user where id =3;
+----+------+------+
| id | name | age  |
+----+------+------+
|  3 | qq   |    8 |
+----+------+------+
1 row in set (0.00 sec)
```

隔离级别—RR与RC对比—RC模式

用友云 yonyou cloud

示例 2：隔离级别设置为 RC。（`show global variables like '%isolation%';` `set global transaction_isolation ='read-committed';`
`-- set global transaction_isolation ='repeatable-read';`）

```
mysql> select * from user;
+----+------+------+
| id | name | age  |
+----+------+------+
|  1 | zg   |   36 |
|  2 | zt   |   34 |
|  3 | qq   |    9 |
|  4 | mm   |    3 |
+----+------+------+
4 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update user set age = age +1 where id =3;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

```
1 row in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from user where id = 3;
+----+------+------+
| id | name | age  |
+----+------+------+
|  3 | qq   |    9 |
+----+------+------+
1 row in set (0.00 sec)

mysql> select * from user where id = 3;
+----+------+------+
| id | name | age  |
+----+------+------+
|  3 | qq   |   10 |
+----+------+------+
1 row in set (0.00 sec)

mysql>
```

会话 1 在 Commit 之前，会话 2 查询的值为原始值 9。会话 1 提交后，会话 2 再查，变成新值 10。

隔离级别

用友云
yonyou cloud

RR和RC模式下MVCC下读区别：

读的实现不同：

1. RC总是读取记录的最新版本，如果记录被锁，则读取记录最新的一次快照，因此不能保证可重复读。
2. RR则是读取该记录事务开始时的那个版本。

重要概念：

快照读：

不论如何读，最后的结果都能读到（读的快照数据），并不会被写阻塞，这种读叫快照读（Snapshot Read）。

当前读：

Current Read，这种读操作读的不再是数据的快照版本，而是数据的最新版本，并会对数据加锁。

SELECT ... LOCK IN SHARE MODE：加 S 锁（锁定读）

SELECT ... FOR UPDATE：加 X 锁（锁定读）

INSERT / UPDATE / DELETE：加 X 锁

3 锁

锁–分类

用友云
yonyou cloud

锁的作用：

锁是实现事务的关键。通过对锁的类型（读或写锁），锁的粒度（行锁或表锁），持有锁的时间（临时锁或持续锁）进行组合，就可以实现四种不同的隔离级别。

锁的分类：

全局锁、表级锁、行级锁。（BDB引擎使用页级锁，是一种粒度介于表锁和行锁的锁，锁在page上）。

锁–表锁

用友云
yonyou cloud

表锁：

语法是:lock tables ... read/ write。即两种表共享锁： （Table Read Lock）和表独占写锁（Table Write Lock）。可以用unlock tables主动释放锁，也可以在客户端断开的时候自动释放。

元数据锁：

用于解决DDL操作与DML操作之间的一致性。不需要显式使用，在访问一个表的时候会被自动加上。在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。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。MDL会直到事务提交才会释放，在做表结构变更的时候，一定要小心不要导致锁住线上查询和更新。

表锁兼容性：

当前锁模式/是否兼容/请求锁模式	读锁	写锁
读锁	是	否
写锁	否	否

会话 1	会话 2
BEGIN;	
SELECT * FROM t2	
	DROP TABLE t2
SELECT * FROM t2	

Drop table t2 在等待 Share_read 锁：

锁–行锁

用友云
yonyou cloud

表锁的问题：

表锁使用简单并且占用资源少，如MYISAM存储引擎使用表锁机制，查询自动加表级读锁，更新自动添加表级写锁，以此来解决可能的并发问题。但表锁粒度太粗，导致数据库的并发性能降低。
因此能提供行锁的Innodb目前成为Mysql的默认引擎，行锁是加在索引上的。

表锁行锁对比：

表锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低；
行锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

锁–锁模式

用友云
yonyou cloud

读写锁：

将锁分为读锁和写锁主要是为了提高读的并发，如果不区分读写锁，那么数据库将没办法并发读，并发性将大大降低。而IS（读意向）和IX（写意向）只会应用在表锁上，方便表锁和行锁之间的冲突检测。

共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同记录的排他锁（X)。

例：Select * from table_name where .....lock in share mode

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。

例：select * from table_name where.....for update或UPDATE、DELETE和INSERT语句。

读写意向锁（表锁）：

为了允许行锁和表锁共存，实现多粒度锁机制。同时还有两种内部使用的意向锁（加在表上），分别为意向共享锁和意向排他锁。

意向共享锁（IS）事务打算给数据行加共享锁前，先给表加一个IS锁。

意向排他锁（IX）：事务打算给数据行加排他锁，先给表加一个IX锁。

锁–行锁分类

用友云
yonyou cloud

误区：

对于Innodb行锁的一个误区是：行锁就是将锁锁在行上，这一行记录不能被其他人修改。其实也有可能并不是锁在行上而是行与行之间的间隙上。

行锁分类：

1.LOCK_ORDINARY：也称Next-Key Lock，同时锁住行记录和记录之间的间隙，（RR）

2.LOCK_GAP：GAP间隙锁，锁两个记录之间的 GAP，防止记录插入。

例如：select * from t where id between 10 and 30 for update;会在id=10和id=30之间加上gap锁，所以其他会话如果要插入id=15记录时，就会被阻塞。是为了防止同一事务的2次当前读读出的数据，出现其他事务新插入的数据（幻读）。

3.LOCK_REC_NOT_GAP：Recode锁，只锁记录。

4.LOCK_INSERT_INTENSION：插入意向 GAP 锁，插入记录时使用，是 LOCK_GAP 的一种特例。

锁–行锁分类—记录锁（Record Locks）

用友云
yonyou cloud

记录锁

示例：update user set age =8+1 where id=4; - - Id为主键

1. 在 id = 4 这条记录上加上记录锁，防止其他事务对 id = 4 这条记录进行修改或删除。
2. 记录锁永远都是加在索引上的，就算一个表没有建索引，数据库也会隐式的创建一个索引。
3. 如果 WHERE 条件中指定的列是个二级索引，那么记录锁不仅会加在这个二级索引上，还会加在这个二级索引所对应的聚簇索引上。
4. 如果 SQL 语句无法使用索引时会走主索引实现全表扫描，这个时候 MySQL 会给整张表的所有数据行加记录锁。如果一个 WHERE 条件无法通过索引快速过滤，存储引擎层面就会将所有记录加锁后返回，再由 MySQL Server 层进行过滤。
5. 实际使用过程中，MySQL（5.7） 做了一些改进，在 MySQL Server 层进行过滤的时候，如果发现不满足，会调用 unlock_row 方法，把不满足条件的记录释放锁（显然违背了二段锁协议，设置了 innodb_locks_unsafe_for_binlog 开启 semi- consistent read 的话，对于不满足查询条件的记录，MySQL 会提前放锁， 不过加锁的过程是不可避免的 。Mysql8.0后RR模块不支持了）。这样做，保证了最后只会持有满足条件记录上的锁。(强调：直接使用RC模式)
6. 可见在没有索引时，不仅会消耗大量的锁资源，增加数据库的开销，而且极大的降低了数据库的并发性能，所以说，更新操作一定要记得走索引。

锁-行锁分类—间隙锁（Gap Locks）

用友云
yonyou cloud

间隙锁

示例：update user set age =8+1 where id=4;

如果id=4的记录不存在，这个 SQL 语句在 RC 隔离级别不会加任何锁，在 RR 隔离级别会在 id = 4 前后两个索引之间加上间隙锁。间隙锁和间隙锁之间是互不冲突的，间隙锁唯一的作用就是为了防止其他事务的插入，所以加间隙 S 锁和加间隙 X 锁没有任何区别。

锁-行锁分类—Next-Key Locks

用友云
yonyou cloud

Next Key锁

示例：update user set age =8+1 where id=4;

加在某条记录以及这条记录前面间隙上的锁。假设一个索引包含10、13 和 20 这几个值，可能的 Next- key 锁如下：(- ∞ , 10] 、(10, 13]、(13, 20]、(20, +∞)

通常我们都用这种左开右闭区间来表示 Next- key 锁，其中，圆括号表示不包含该记录，方括号表示包含该记录。前面3个都是 Next- key 锁，最后一个为间隙锁。和间隙锁一样，在 RC 隔离级别下没有 Next- key 锁，只有 RR 隔离级别才有。

对于上面SQL，如果 id 不是主键，而是二级索引，且不是唯一索引，那么这个 SQL 在 RR 隔离级别下会加Next- key 锁，如下：(a, 4]、(4, b)。
其中，a 和 b 是 id = 4 前后两个索引值，我们假设 a = 1、b = 10，那么此时如果插入一条 id = 3 的记录将会阻塞住。之所以要把 id = 4 前后的间隙都锁住，仍然是为了解决幻读问题，因为 id 是非唯一索引，所以 id = 4 可能会有多条记录，为了防止再插入一条 id = 4 的记录，必须将下面标记 ^ 的位置都锁住，因为这些位置都可能再插入一条 id = 4 的记录：1 ^ 4 ^ 4 ^ 4 ^ 10 13 15

Next- key 锁确实可以避免幻读，但是带来的副作用是连插入 id = 3 这样的记录也被阻塞了，这根本就不会引起幻读问题的。

锁–查看行锁

用友云
yonyou cloud

一：通过下面的 SQL 语句：

```
select * from  information_schema.innodb_locks;
select * from performance_schema.data_locks;（my8.0以后查询这张表）
```

这个命令会打印出 InnoDb 的所有锁信息，包括锁 ID、事务 ID、以及每个锁的类型和模式等其他信息。

二：使用下面的 SQL 语句：

```
show engine innodb status\ G;
```

锁-查看行锁—综合案例

用友云
yonyou cloud

```
--rr级别:
create table t(a varchar(10),b int,c int,d varchar(20),primary key(a),unique iux_b(b),index idx_c(c))
insert into t values ('a',11,22,'dataa'),('b',111,222,'datab'),('c',1111,2222,'datac'),('d',11111,22222,'datad'),('e',111111,222222,'datae')

t1: begin;update t set c=224 where c=222;
t2: begin;insert into t values ('f',116,226,'dataf')

18  select * from performance_schema.data_locks
19
```

	ENGINE VarChar(32)	ENGI... VarCha...	ENGINE_TRANS... UInt64	THREAD_ID UInt64	EVENT... UInt64	OBJECT_SCHE... VarChar(64)	O... V	PAR... VarC...	S... Va...	INDEX_NAME VarChar(64)	OBJECT_INSTAN... UInt64	LOCK_TYPE VarChar(32)	LOCK_MODE VarChar(32)	LOCK_STATUS VarChar(32)	LOCK_DATA VarChar(8192)
1	INNODB	21627...	335812	85	68	zg	t...	NULL	N...	NULL	2162699273544	TABLE	IX	GRANTED	NULL
2	INNODB	21627...	335812	85	68	zg	t...	NULL	N...	idx_c	2162699270760	RECORD	X,GAP,INSERT_INTENTION	WAITING	2222, 'c'
3	INNODB	21627...	335810	87	28	zg	t...	NULL	N...	NULL	2162699287608	TABLE	IX	GRANTED	NULL
4	INNODB	21627...	335810	87	28	zg	t...	NULL	N...	idx_c	2162699284824	RECORD	X	GRANTED	222, 'b'
5	INNODB	21627...	335810	87	28	zg	t...	NULL	N...	PRIMARY	2162699285168	RECORD	X,REC_NOT_GAP	GRANTED	'b'
6	INNODB	21627...	335810	87	28	zg	t...	NULL	N...	idx_c	2162699285512	RECORD	X,GAP	GRANTED	2222, 'c'
7	INNODB	21627...	335810	87	28	zg	t...	NULL	N...	idx_c	2162699285512	RECORD	X,GAP	GRANTED	224, 'b'

锁-加锁流程—单条记录

用友云
yonyou cloud

索引

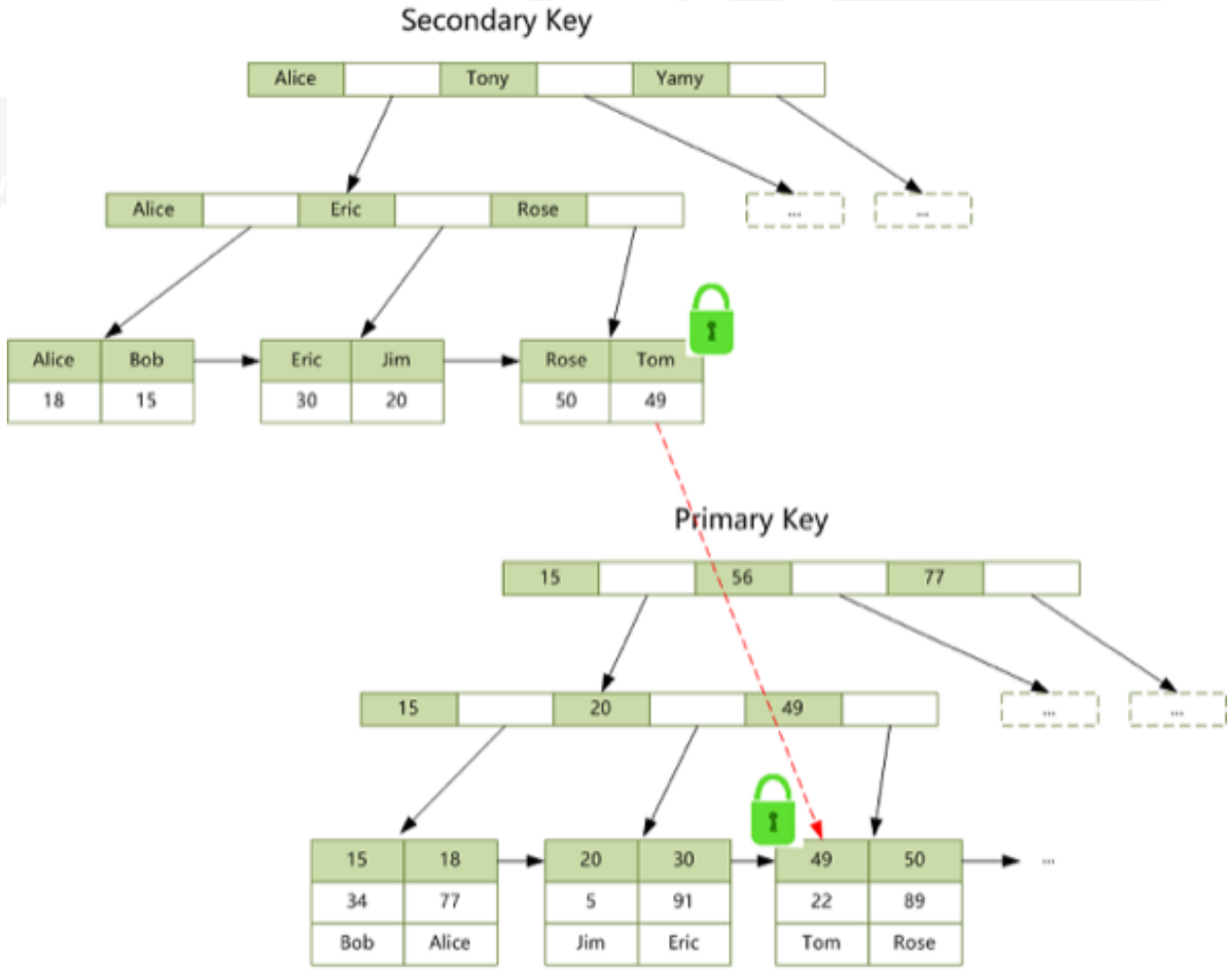
由于行锁是加在索引上：主键索引（Primary Index）和非主键索引（Secondary Index，又称为二级索引、辅助索引）

简单sql：

1. update students set score = 100 where id = 49;
2. id = 49 这个主键索引上加一把 X 锁。

3. update studentsset score = 100 where name = 'Tom’;

在 name = 'Tom' 这个索引上加一把 X 锁，同时会通过 name = 'Tom' 这个二级索引定位到 id = 49 这个主键索引，并在 id = 49 这个主键索引上加一把 X 锁。

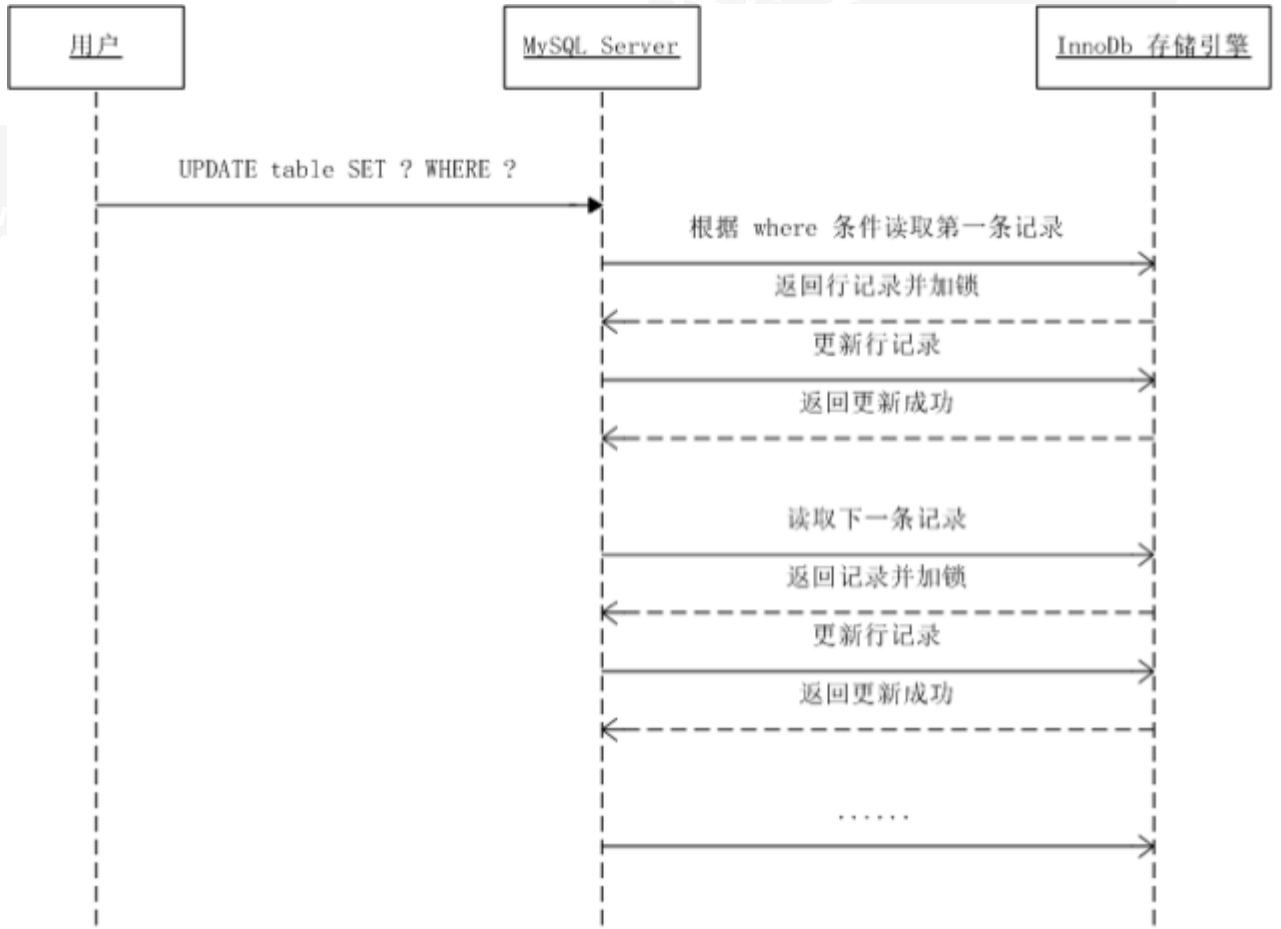


锁-加锁流程—多条记录

update students set level = 3 where score >= 60;

当 UPDATE 语句被发给 MySQL 后，MySQL Server 会根据 WHERE 条件读取第一条满足条件的记录，然后 InnoDB 引擎会将第一条记录返回并加锁（current read），待 MySQL Server 收到这条加锁的记录之后，会再发起一个 UPDATE 请求，更新这条记录。一条记录操作完成，再读取下一条记录，直至没有满足条件的记录为止。

因此，MySQL 在操作多条记录时 InnoDB 与 MySQL Server 的交互是一条一条进行的，加锁也是一条一条依次进行的，先对一条满足条件的记录加锁，返回给 MySQL Server，做一些 DML 操作，然后在读取下一条加锁，直至读取完毕。



锁–加锁详解—加锁规则

加锁规则：

原则1：加锁的基本单位是next- key lock，next- key lock是前开后闭区间。

原则2：查找过程中访问到的对象才会加锁。

优化1：索引上的等值查询，给唯一索引加锁的时候，next- key lock退化为行锁。

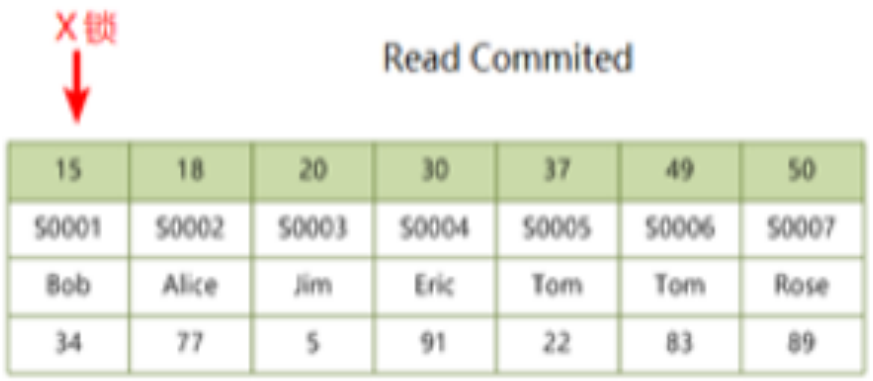
优化2：索引上的等值查询，向右遍历时且最后一个值不满足等值条件的时候，next- key lock退化为间隙锁。

锁–加锁案例1

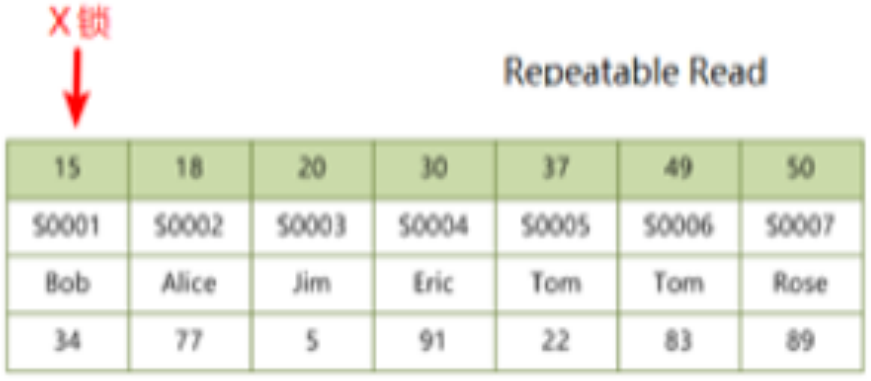
用友云
yonyou cloud

5.4.1.1. 聚簇索引，查询命中

语句 UPDATE students SET score = 100 WHERE id = 15 在 RC 和 RR 隔离级别下加锁情况一样，都是对 id 这个聚簇索引加 X 锁，如下：



15	18	20	30	37	49	50
S0001	S0002	S0003	S0004	S0005	S0006	S0007
Bob	Alice	Jim	Eric	Tom	Tom	Rose
34	77	5	91	22	83	89



15	18	20	30	37	49	50
S0001	S0002	S0003	S0004	S0005	S0006	S0007
Bob	Alice	Jim	Eric	Tom	Tom	Rose
34	77	5	91	22	83	89

锁-加锁案例2

用友云
yonyou cloud

5.4.1.2. 聚簇索引，查询未命中

如果查询未命中纪录，在 RC 和 RR 隔离级别下加锁是不一样的，因为 RR 有 GAP 锁。语句 UPDATE students SET score = 100 WHERE id = 16 在 RC 和 RR 隔离级别下的加锁情况如下（RC 不加锁）：

Read Commited

15	18	20	30	37	49	50
S0001	S0002	S0003	S0004	S0005	S0006	S0007
Bob	Alice	Jim	Eric	Tom	Tom	Rose
34	77	5	91	22	83	89

GAP 锁

Repeatable Read

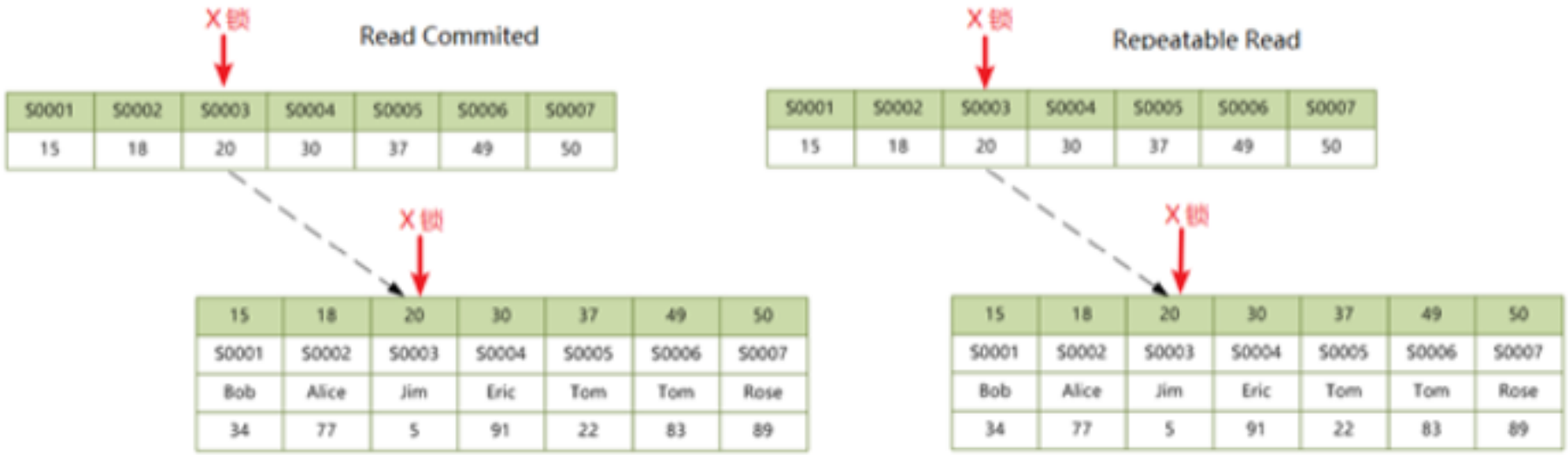
15	18	20	30	37	49	50
S0001	S0002	S0003	S0004	S0005	S0006	S0007
Bob	Alice	Jim	Eric	Tom	Tom	Rose
34	77	5	91	22	83	89

锁–加锁案例3

用友云 yonyou cloud

5.4.1.3. 二级唯一索引，查询命中

语句 UPDATE students SET score = 100 WHERE no = 'S0003' 命中二级唯一索引，我们知道二级索引的叶子节点中保存了主键索引的位置，在给二级索引加锁的时候，主键索引也会一并加锁。这个在 RC 和 RR 两种隔离级别下没有区别：



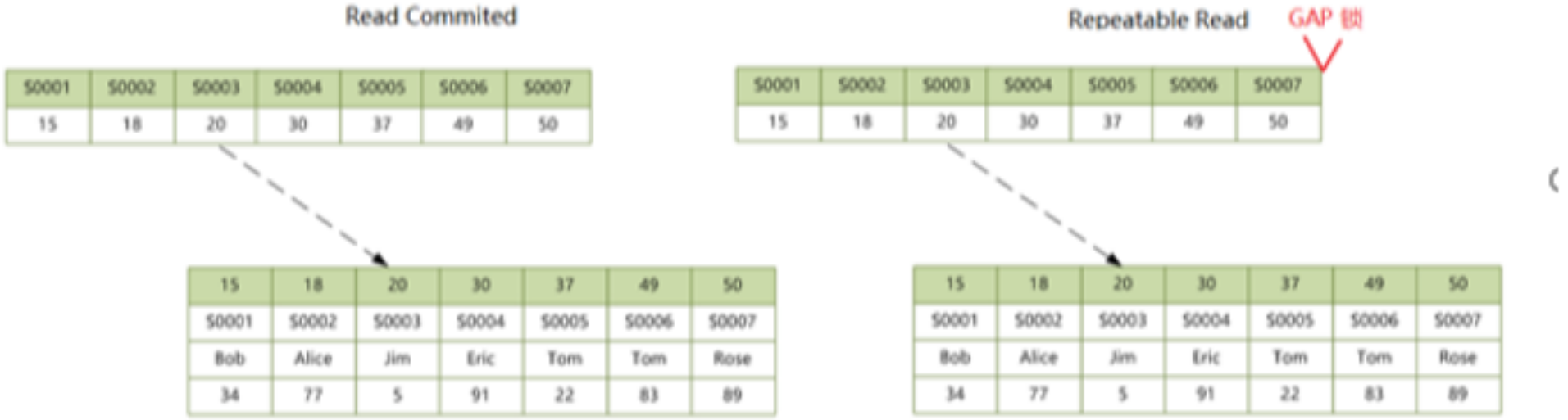
那么，为什么主键索引上的记录也要加锁呢？因为有可能其他事务会根据主键对 students 表进行更新，如：UPDATE students SET score = 100 WHERE id = 20，试想一下，如果主键索引没有加锁，那么显然会存在并发问题。

锁–加锁案例4

用友云 yonyou cloud

5.4.1.4. 二级唯一索引，查询未命中

如果查询未命中纪录，RR 隔离级别会加 GAP 锁，RC 无锁。语句 UPDATE students SET score = 100 WHERE no = 'S0008' 加锁情况如下：



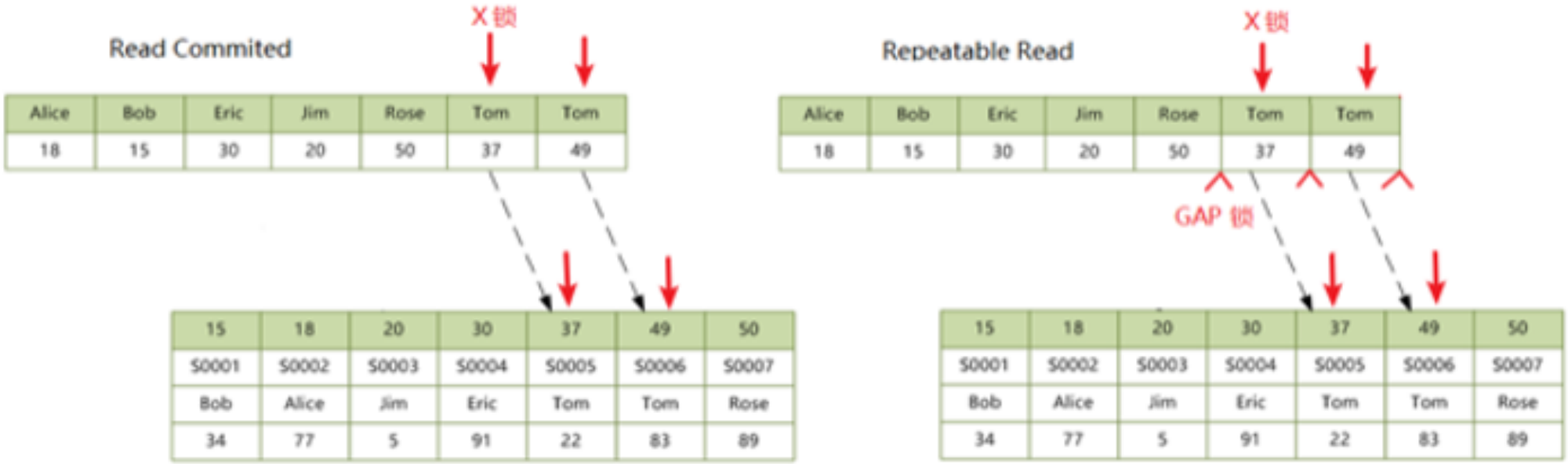
这种情况下只会在二级索引加锁，不会在聚簇索引上加锁。

锁–加锁案例5

用友云 yonyou cloud

5.4.1.5. 二级非唯一索引，查询命中

如果查询命中的是二级非唯一索引，在 RR 隔离级别下，还会加 GAP 锁。语句 UPDATE students SET score = 100 WHERE name = 'Tom' 加锁如下：



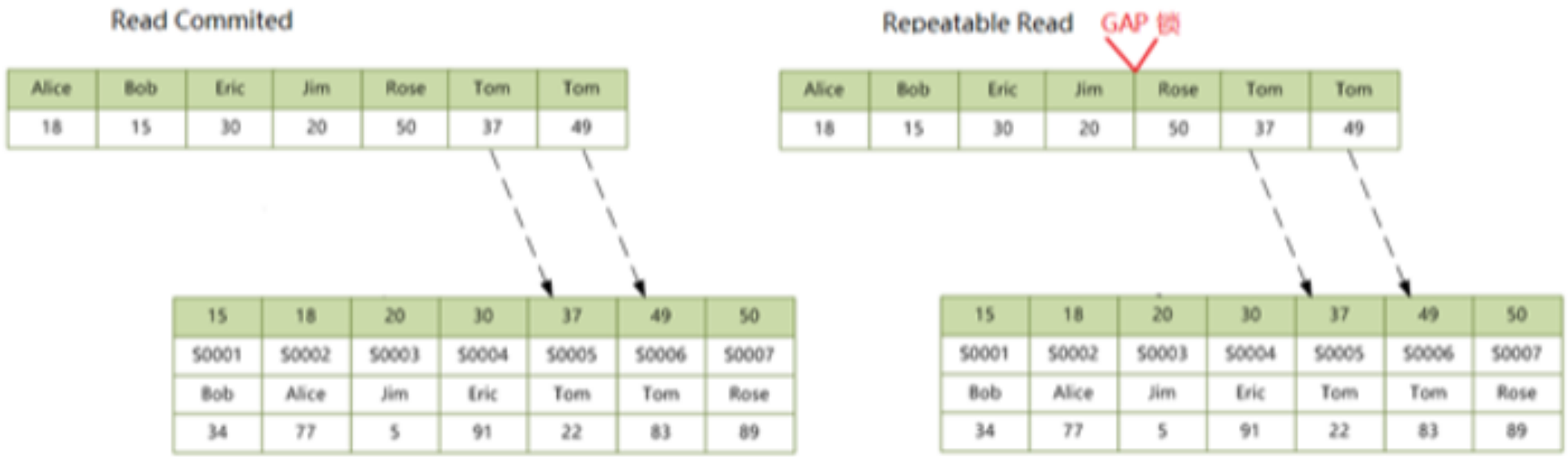
为什么非唯一索引会加 GAP 锁，而唯一索引不用加 GAP 锁呢？原因很简单，GAP 锁的作用是为了解决幻读，防止其他事务插入相同索引值的记录，而唯一索引和主键约束都已经保证了该索引值肯定只有一条记录，所以无需加 GAP 锁。

锁–加锁案例6

用友云 yonyou cloud

5.4.1.6. 二级非唯一索引，查询未命中

如果查询未命中纪录，RR 隔离级别会加 GAP 锁，RC 无锁。语句 UPDATE students SET score = 100 WHERE name = 'John' 加锁情况如下：

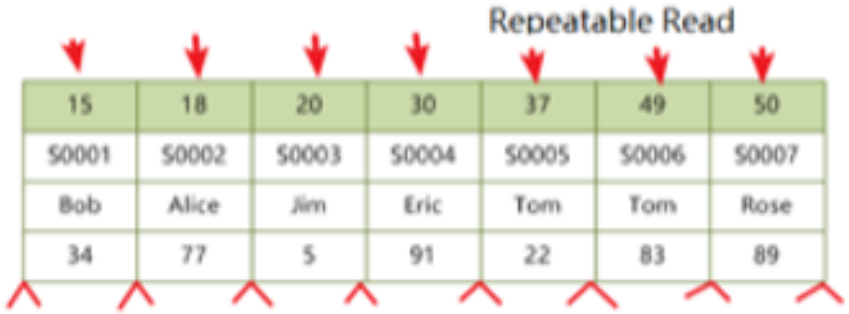


锁–加锁案例7

用友云 yonyou cloud

5.4.1.7. 无索引

如果 WHERE 条件不能走索引，MySQL 会如何加锁呢？有的人说会在表上加 X 锁，也有人说会根据 WHERE 条件将筛选出来的记录在聚簇索引上加上 X 锁，看下图：



在没有索引的时候，只能走聚簇索引，对表中的记录进行全表扫描。在 RC 隔离级别下会给所有记录加行锁，在 RR 隔离级别下，不仅会给所有记录加行锁，所有聚簇索引和聚簇索引之间还会加上 GAP 锁。

语句 UPDATE students SET score = 100 WHERE score = 22 满足条件的虽然只有 1 条记录，但是聚簇索引上所有的记录，都被加上了 X 锁。那么，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。

锁–加锁案例8

用友云
yonyou cloud

5.4.1.8. 聚簇索引，范围查询

Read Commited

15	18	20	30	37	49	50
S0001	S0002	S0003	S0004	S0005	S0006	S0007
Bob	Alice	Jim	Eric	Tom	Tom	Rose
34	77	5	91	22	83	89

Repeatable Read

15	18	20	30	37	49	50
S0001	S0002	S0003	S0004	S0005	S0006	S0007
Bob	Alice	Jim	Eric	Tom	Tom	Rose
34	77	5	91	22	83	89

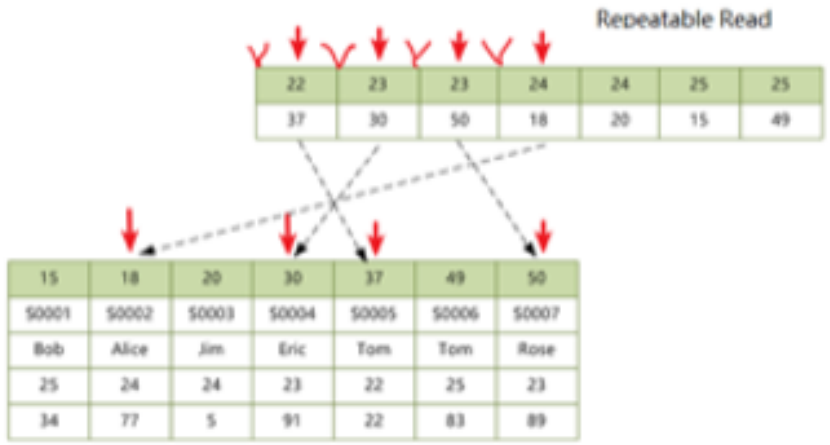
对于范围查询，如果 WHERE 条件是 id <= N，那么 N 后一条记录也会被加上 Next-key 锁；如果条件是 id < N，那么 N 这条记录会被加上 Next-key 锁。另外，如果 WHERE 条件是 id >= N，只会给 N 加上记录锁，以及给比 N 大的记录加锁，不会给 N 前一条记录加锁；如果条件是 id > N，也不会锁前一条记录，连 N 这条记录都不会锁。

锁–加锁案例9

用友云 yonyou cloud

5.4.1.9. 二级索引，范围查询

把范围查询应用到二级非唯一索引上来，SQL 语句为：UPDATE students SET score = 100 WHERE age <= 23，加锁情况如下图所示：



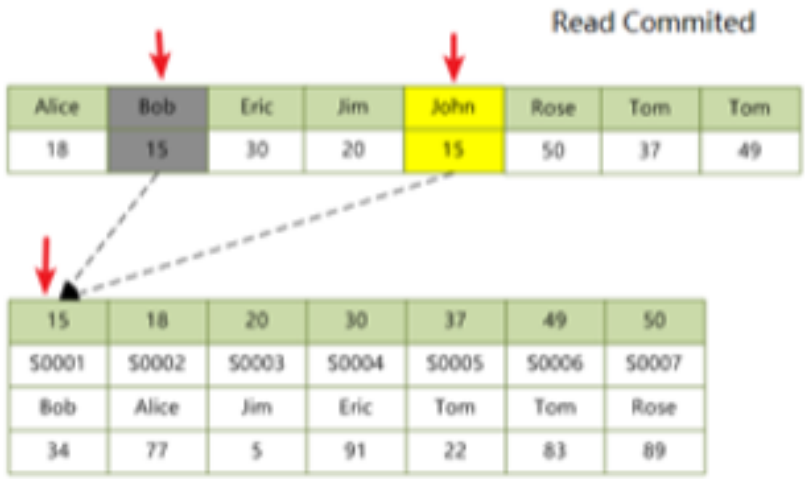
可以看出和聚簇索引的范围查询一样，除了 WHERE 条件范围内的记录加锁之外，后面一条记录也会加上 Next-key 锁，这里有意思的一点是，尽管满足 age＝24 的记录有两条，但只有第一条被加锁，第二条没有加锁，并且第一条和第二条之间也没有加锁。

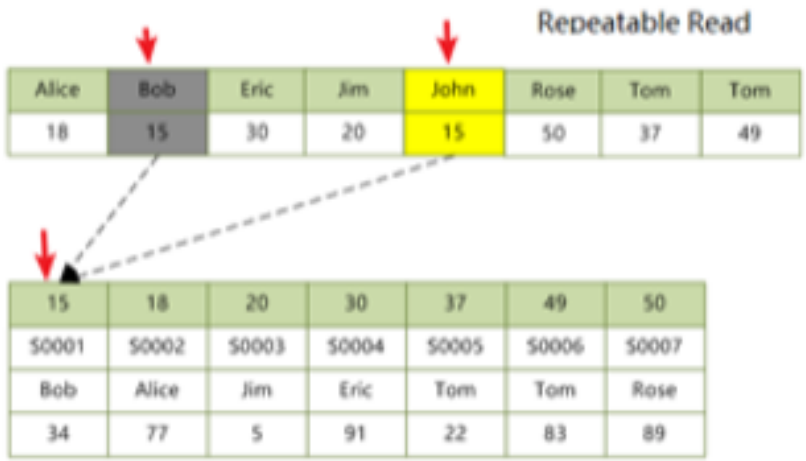
锁–加锁案例10

用友云
yonyou cloud

5.4.1.10. 修改索引值

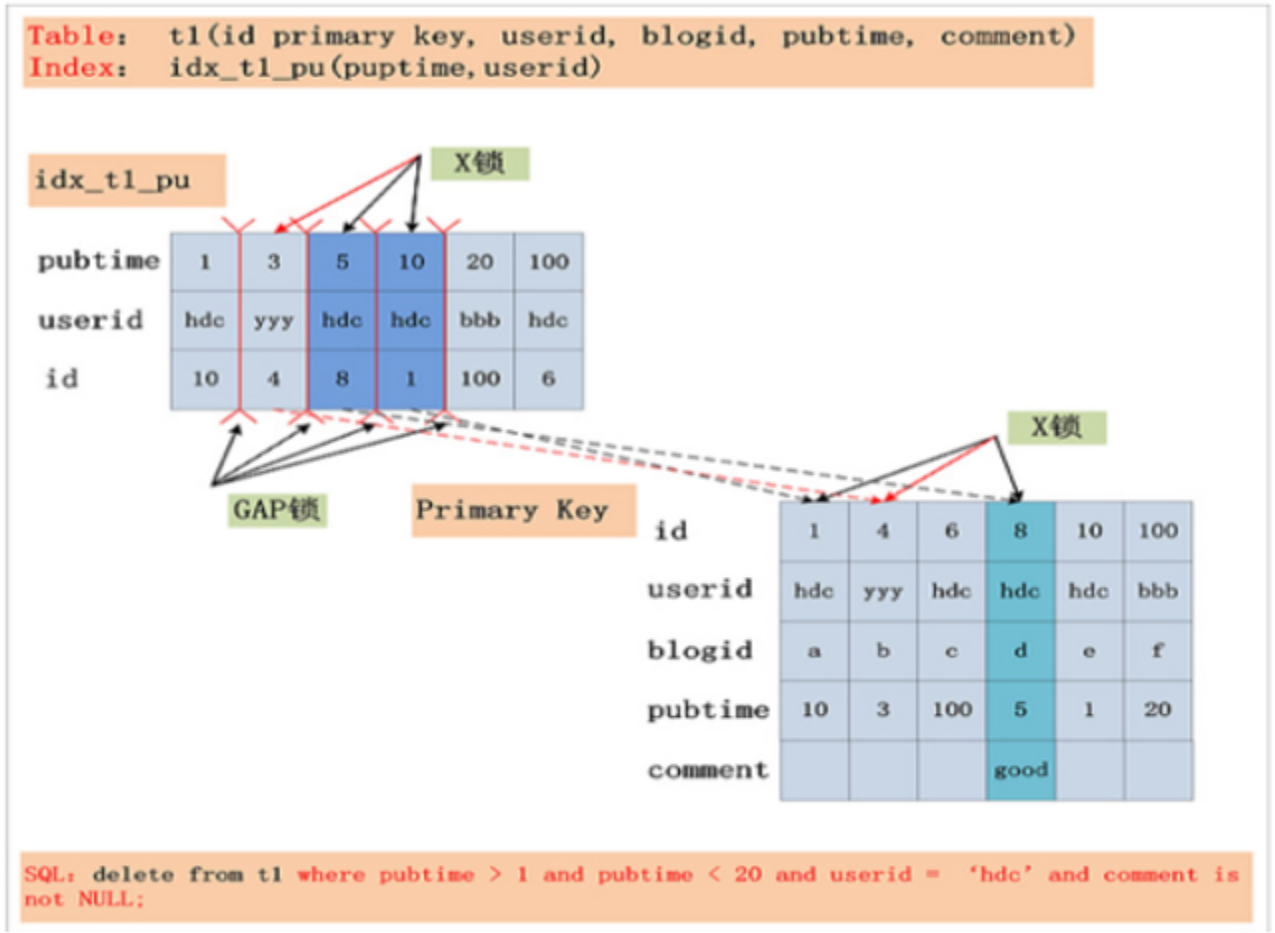
WHERE 部分的索引加锁原则和上面介绍的一样，多的是 SET 部分的加锁。譬如 UPDATE students SET name = 'John' WHERE id = 15 不仅在 id = 15 记录上加锁之外，还会在 name = 'Bob'（原值）和 name = 'John'（新值） 上加锁。示意图如下：





RC 和 RR 没有区别。

锁–加锁案例11–复杂条件加锁



可以看到 pubtime > 1 and pubtime < 20 为 Index First Key 和 Index Last Key，MySQL 会在这个范围内加上记录锁和间隙锁；userid = 'hdc' 为 Index Filter，这个过滤条件可以在索引层面就可以过滤掉一条记录，因此如果数据库支持 ICP 的话，(4, yyy, 3) 这条记录就不会加锁；comment is not NULL 为 Table Filter，虽然这个条件也可以过滤一条记录，但是它不能在索引层面过滤，而是在根据索引读取了整条记录之后才过滤的，因此加锁并不能省略。

4

锁–死锁分析

用友云
yonyou cloud

死锁如何产生的：

在数据库中，如果两个不同的事务在执行时，互相持有了对方所需的锁，此时由于它们都在等待某个资源，永远不会释放自己获得的锁，因此就会产生死锁（deadlock）。表级锁不会产生死锁.所以解决死锁主要还是针对于最常用的InnoDB引擎。

```
create table zg (id int primary key ,name int);
insert into zg values(1,1),(2,2),(3,3),(4,4),(5,5);
```

会话 1	会话 2
begin; update zg set name =33 where id =3 --(X)	
	begin; update zg set name =555 where id =5 --（X）
update zg set name =55 where id =5 --（卡住）	
	update zg set name =333 where id =3
(1 row affected)	[MySql.Data] ErrorCode: -2147467259, Number: 1213 ErrorMessage: Deadlock found when trying to get lock; try restarting transaction

锁–死锁日志

以下是通过 SHOW ENGINE INNODB STATUS 结果中的 LATEST DETECTED DEADLOCK 节中关于死锁的详细信息：

```
LATEST DETECTED DEADLOCK
2020-05-30 13:30:16 0x73f0
*** (1) TRANSACTION:
TRANSACTION 335387, ACTIVE 12 sec starting index read    --执行了 12 秒，正在使用索引读取数据行
mysql tables in use 1, locked 1    --正在使用 1 个表，涉及锁表的 1 个
LOCK WAIT 3 lock struct(s), heap size 1136, 2 row lock(s), undo log entries 1 –等待 3 个锁，涉及 2 行记录
MySQL thread id 8, OS thread handle 8616, query id 119 localhost 127.0.0.1 root updating
update zg set name =55 where id =5
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 272 page no 4 n bits 72 index PRIMARY of table `testzg`.`zg` trx id 335387 lock_mode X locks rec but not gap waiting
Record lock, heap no 6 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 4; hex 80000005; asc     ;;
*** (2) TRANSACTION:
TRANSACTION 335388, ACTIVE 10 sec starting index read
mysql tables in use 1, locked 1
3 lock struct(s), heap size 1136, 2 row lock(s), undo log entries 1
MySQL thread id 9, OS thread handle 29680, query id 120 localhost 127.0.0.1 root updating
update zg set name =333 where id =3
*** (2) HOLDS THE LOCK(S): --持有哪些锁
RECORD LOCKS space id 272 page no 4 n bits 72 index PRIMARY of table `testzg`.`zg` trx id 335388 lock_mode X locks rec but not gap
Record lock, heap no 6 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 4; hex 80000005; asc     ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 272 page no 4 n bits 72 index PRIMARY of table `testzg`.`zg` trx id 335388 lock_mode X locks rec but not gap waiting
Record lock, heap no 4 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 4; hex 80000003; asc     ;;
*** WE ROLL BACK TRANSACTION (2)
```

用友云 yonyou cloud | 数字企业智能服务