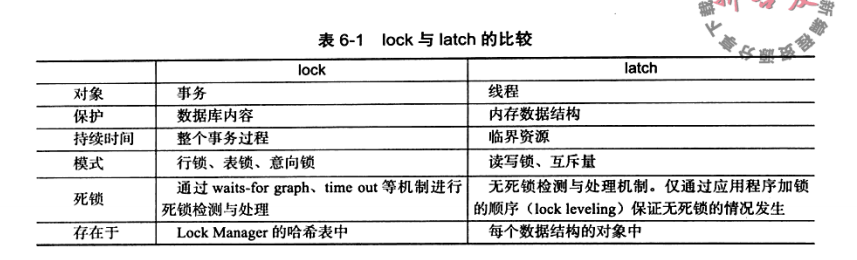
1. **Lock和latch介绍**

Latch锁的对象是数据库线程中的资源，分为mutex和readwriteLock，用来保证并发过程中临近资源安全性，没有死锁检测。

Lock锁的对象是事务，用来锁定数据库中的对象，如表、页、行。Lock对象在事务commit或rollback后释放，有死锁机制。



对于innodb中的latch，我们可以通过show engine innodb mutex来查询。Debug模式下，可以看到更多信息。

至于lock信息，在可以通过show engine innodb status查询。具体死锁排查过程参见《mysql-5-事务-死锁-binlog》

1. **Lock算法**

行锁主要有3种方式

* 行锁record lock：只锁定这一行
* 间隙锁gap lock：锁定一个左开右闭区间范围
* next key lock：record lock + gap lock

这里详细描述下next key lock。例如一个索引有10，11，13，20这4个值，那么gap lock区间为：

(-INF, 10]

(10, 11]

(11, 13]

(13, 20]

(20, +INF)

这里的区间必须是左开右闭区间。

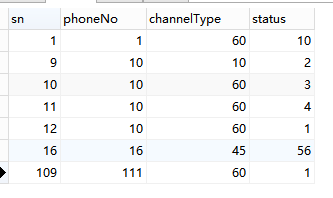
**需要注意的是，只有RR支持next key lock和gap lock，RC隔离等级的情况下，不符合where 条件的行记录的间隙锁会被释放**。

下面以smstest表为例来解释RR隔离级别下各个情况下next key lock情况。

|  |
| --- |
| CREATE TABLE `smstest` (  `sn` int(11) NOT NULL AUTO\_INCREMENT COMMENT '自增编号' ,  `phoneNo` int(16) NOT NULL ,  `channelType` int(11) NULL DEFAULT NULL COMMENT '通道识别' ,  `status` tinyint(4) NOT NULL COMMENT '短信转态,1.发送成功,2.发送失败,3.发送异常' ,  PRIMARY KEY (`sn`),  INDEX `in\_p\_index` (`phoneNo`) USING BTREE  )  ENGINE=InnoDB  DEFAULT CHARACTER SET=utf8 COLLATE=utf8\_general\_ci  COMMENT='短信发送成功记录表'  AUTO\_INCREMENT=114  ROW\_FORMAT=DYNAMIC; |

Sn为主键，phoneNo为辅助索引。

初始数据如下：



1. **RR唯一索引的next key lock——行锁（错！）**

如果where条件是唯一键（主键），则next key lock会降级成record lock，只锁定where条件范围内的行。错！

和辅助索引一致，都是锁索引!

例如：

|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
| select \* from smstest t where t.`sn` BETWEEN 1 and  9 for UPDATE; |  |  |
|  | start TRANSACTION;  insert into SmsTest values(2,110,1,1);  commit; | 阻塞，不可以直接插入。 |
| commit; |  |  |

1. **RR辅助索引的next key lock**
2. **Where 条件=**

如果where条件中是等于，此时会有左右两个gap lock以及record lock本身。例如：

select \* from smstest t where t.`phoneNo`=16 for UPDATE;

其next key lock所在索引phoneNo的区间为：(10, 16), 16,(16, 111)。这与辅助索引本身结构有关，此时的next key lock如下：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PhoneNo  辅助索引 | 1 | 10 | | | | GAP  LOCK | 16  (RECORD  LOCK) | GAP  LOCK | 111 |
| Sn(主键) | 1 | 9 | 10 | 11 | 12 | 16 | 109 |

注意10和111本身来说都不在间隙锁范围内，但是由于辅助索引的数据页中聚簇索引是顺序排列的，如果我们插入聚簇索引在间隙锁范围内的新数据，例如：

insert into SmsTest values(13,10,1,1); 第一列是sn，第二列是phoneNo

显然位于间隙锁范围内，此时任然会阻塞。反之，如果我们插入间隙锁外的数据，如：

insert into SmsTest values(8,10,1,1);

则可以直接插入。

下面来看例子：

间隙锁区间范围内

|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
| select \* from smstest t where t.`phoneNo`=16 for UPDATE; |  |  |
|  | start TRANSACTION;  insert into SmsTest values(8,10,1,1);  commit; | 间隙锁范围内，阻塞，等待超时 |
| commit; |  |  |

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围内。

|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
| select \* from smstest t where t.`phoneNo`=16 for UPDATE; |  |  |
|  | start TRANSACTION;  insert into SmsTest values(108,111,1,1);  commit; | 虽然111是间隙锁区间外，但实际数据插入位置在间隙锁范围内，阻塞，等待超时 |
| commit; |  |  |

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围外。

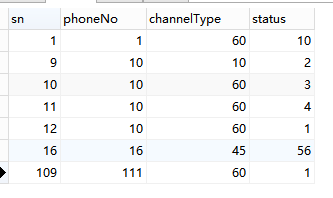
|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
| select \* from smstest t where t.`phoneNo`=16 for UPDATE; |  |  |
|  | start TRANSACTION;  insert into SmsTest values(110,111,1,1);  commit; | 实际数据插入位置在间隙锁范围外，直接插入 |
| commit; |  |  |

1. **Where 条件 <**

如果字段在where条件中是小于x，那么首先会在索引中从x向右寻找最近的那个索引值y，where col < x 等价于where col < y。

例如下表中，对于select \* from newtable t where t.`phoneNo`<15 for UPDATE;

就相当于select \* from newtable t where t.`phoneNo`<16 for UPDATE;



此时next key lock加锁范围如下，注意最左侧的gap lock位于负无穷。显然我们可以插入(17,16,1,1)，但是不能插入(15,16,1,1)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PhoneNo  辅助索引 | GAP  LOCK | 1 | 10 | | | | GAP  LOCK | 16 | 111 |
| Sn(主键) | 1 | 9 | 10 | 11 | 12 | 16 | 109 |

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围内。

|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
| select \* from smstest t where t.`phoneNo`<16 for UPDATE; |  |  |
|  | start TRANSACTION;  insert into SmsTest values(15,16,1,1);  commit; | 虽然16是间隙锁区间外，但实际数据插入位置在间隙锁范围内，阻塞，等待超时 |
| commit; |  |  |

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围外。

|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
| select \* from smstest t where t.`phoneNo<16 for UPDATE; |  |  |
|  | start TRANSACTION;  insert into SmsTest values(17,16,1,1);  commit; | 实际数据插入位置在间隙锁范围外，直接插入 |
| commit; |  |  |

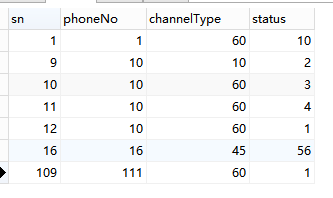
如果有小于等于，则间隙锁范围需要与1)中的锁范围求并集合

1. **Where 条件 >**

如果字段在where条件中是大于x，那么首先会在索引中从x向左寻找最近的那个索引值。

例如下表中，对于select \* from newtable t where t.`phoneNo`>14 for UPDATE;

就相当于select \* from newtable t where t.`phoneNo`>10 for UPDATE;



此时next key lock加锁范围是如下，，注意最右侧的gap lock位于正无穷。显然我们可以插入(8,10,1,1)，但是不能插入(13,10,1,1)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PhoneNo  辅助索引 | 1 | 10 | | | | GAP  LOCK | 16 | 111 | GAP  LOCK |
| Sn(主键) | 1 | 9 | 10 | 11 | 12 | 16 | 109 |

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围内。

|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
| select \* from smstest t where t.`phoneNo`>14 for UPDATE; |  |  |
|  | start TRANSACTION;  insert into SmsTest values(13,10,1,1);  commit; | 虽然10是间隙锁区间外，但实际数据插入位置在间隙锁范围内，阻塞，等待超时 |
| commit; |  |  |

间隙锁区间范围边缘，但是事实插入位置在间隙锁范围外。

|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
| select \* from smstest t where t.`phoneNo`>14 for UPDATE; |  |  |
|  | start TRANSACTION;  insert into SmsTest values(8,10,1,1);  commit; | 实际数据插入位置在间隙锁范围外，直接插入 |
| commit |  |  |

如果有小于等于，则间隙锁范围需要与1)中的锁范围求并集合

1. **无索引情况下的next key lock——锁表**

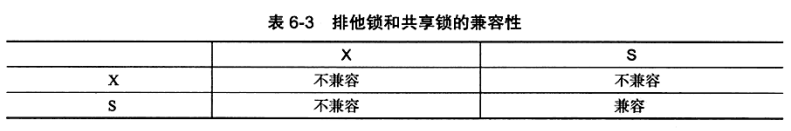
where条件中字段无索引情况下，间隙锁会扩大到整个表，即锁表！

|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
| update SmsTest set channelType=10 where `status` = 2 |  |  |
|  | start TRANSACTION;  update SmsTest set channelType=11 where `status` = 5  commit; | Status无索引，直接锁表 |
| commit; |  |  |

1. **Lock分类**
2. **行锁与表锁**

行锁和表锁主要有两种：

* 排他锁（X-lock）：写锁
* 共享锁（S-LOCK）：读锁、



由于行锁和表锁的加锁对象不同，所以二者如何同步也就成了问题。SessionA我们对一行数据加了S锁，sessionB试图对该表整体加X锁，显然互斥。表X锁如何才能获取呢[1]？

step1：判断表是否已被其他事务用表锁锁表

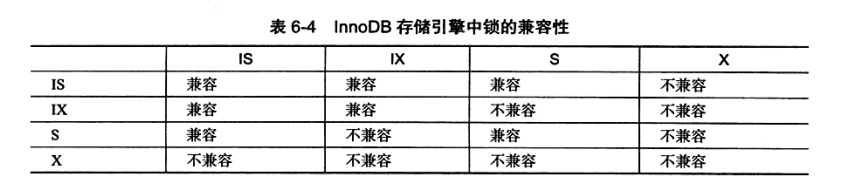
step2：判断表中的每一行是否已被行锁锁住。

step2中需要遍历全表，显然效率极低。所以我们在加行锁之前对表对象加入意向锁，这样就可以表征行锁加入对表锁的影响。在意向锁存在的情况下，事务A必须先申请表的意向共享锁，成功后再申请一行的行锁。

在意向锁存在的情况下，上面的判断可以改成：

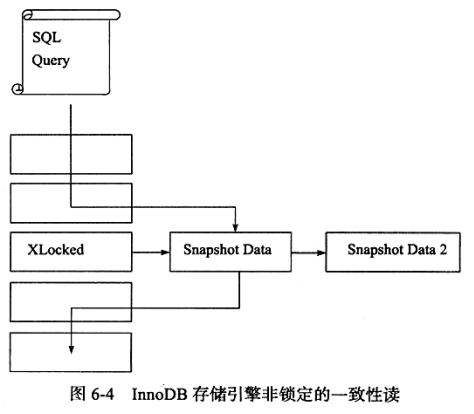
step1：不变

step2：发现表上有意向共享锁，说明表中有些行被共享行锁锁住了，因此，事务B申请表的写锁会被阻塞。



1. **一致性非锁定读**

一致性非锁定读是指innodb通过多版本并发控制（MVCC）来读取当前时间数据库中的行数据。如果读取过程中正执行DML操作，读操作不会等待行锁释放，而是读取行的一个快照。



显然，一致性非锁定读是对普通读写锁的极大优化，可以提高数据库查询效率。其中，snapshot是通过**undo段**来完成的，undo用于事务中回滚数据，因此快照本身没有额外开销。

1. **Read Committed——当前读**

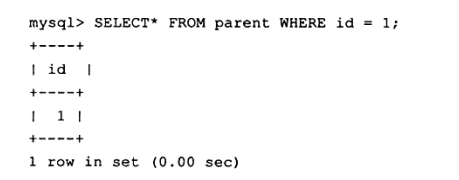
Read Committed级别下，总是进行当前读（最近一份快照），即读取最近一份commited数据，所以会有不可重复读的现象.

例子如下：

* sessionA：

Start transaction;

Select \* from parent where id = 1;



* sessionB：

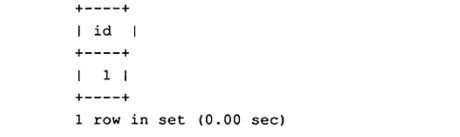
Start transaction;

udpate parent set id = 3 where id = 1;

此时sessionB尚未commit。

* sessionA：

Select \* from parent where id = 1;



* sessionB：

commit;

* sessionA：

Select \* from parent where id = 1;

此时结果为空。说明RC级别下是读commit最新版本，即当前读。

1. **Repeatable Read——快照读**

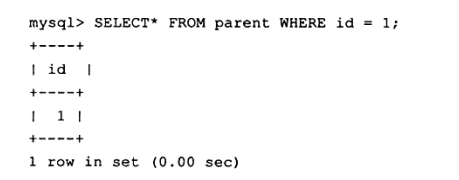
Repeatable Read级别下，总是进行快照度（第一次读的快照），所以不会有不可重复读的问题，且可以避免一般意义上的幻读。

例子如下：

* sessionA：

Start transaction;

Select \* from parent where id = 1;



* sessionB：

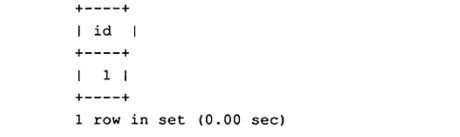
Start transaction;

udpate parent set id = 3 where id = 1;

此时sessionB尚未commit。

* sessionA：

Select \* from parent where id = 1;

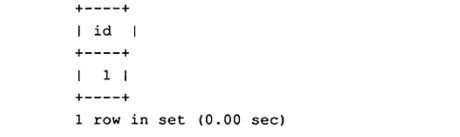


* sessionB：

commit;

* sessionA：

Select \* from parent where id = 1;



此时结果任然为事务快事的快照。



1. **一致性锁定读**

* Select … for update——X锁
* Select … lock in share mode——S锁

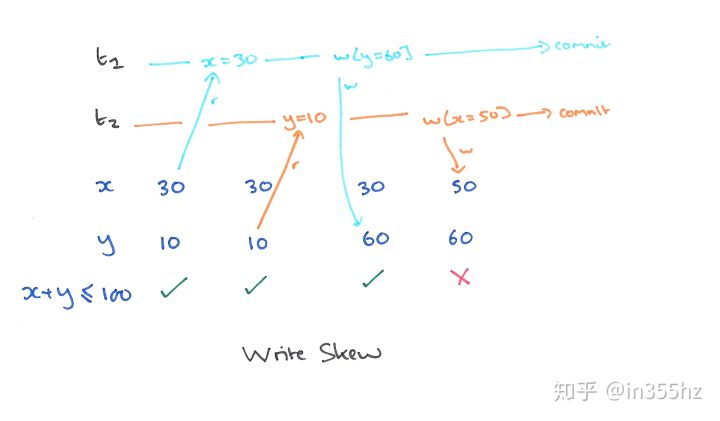
1. **RR级别下一致性非锁定读与幻读**

RR级别下既然有了MVCC，是不是就可以避免幻读了呢？

1. **普通幻读消除**

MySQL（innodb）的 RR 隔离级别实际上是 snapshot isolation，可以避免通常意义的幻读。snapshot isolation 的问题是无法处理如下的 read-write conflict（write skew）。

1. **write skew解决**



我们约定，x + y <= 100. 如果都是快照读/当前读，t1先读取x=30，然后根据快照/当前版本判断，进行y=60更新满足 x + y <= 100。

t2先读取y = 10，在t1更新y=60之后，由于t1没有提交，此时t2依然可以满足x+y<=100的情况下修改x=50。

之后t1和t2都可以成功commit。但显然违背了我们之前的约束条件。

这种现象称为write skew。由于 UPDATE 本身也是一种 read-write，如果执行 UPDATE 也会有 write skew 问题, 那对实际应用来说就太糟糕了。

Innodb为了解决RR情况下的这个问题，强行把read分为快照读和当前读（Locking read），快照读就是普通的read，当前读就是select … for update。在update或select … for update时，Innodb执行当前读，在过程中加入record lock和gap lock（next key lock），相当于变相提升到了Serializable级别，从而消除write skew。

1. **next key lock的必要性**

上文中，update或select … for update时，Innodb执行当前读，在过程中加入record lock和gap lock（next key lock）。

**Record lock就可以通过悲观锁实现目标资源的串行化，为啥recordLock还不够，还得使用next key lock呢？**

在update或select … for update之后，会更新快照进行当前读，如果只有record lock没有next key lock，在这个过程中可能会有gap内新数据插入，从而出现了原有幻读的现象。所以必须有next key lock来在更新快照后避免传统意义上的幻读出现。

1. **新的不一致读**

虽然解决了write skew，但是当前读（Locking read）和快照本身是矛盾，会出现**新的不一致读**。由于并没有Serializable的悲观锁，在两个不同的事务中仍然可以并发的执行update语句，update/select for update执行当前读之后，会更新快照，导致与之前的快照读不一致的情况。

只有Serializable才能避免这种情况。

例如：mysql正确处理的结果如下[4]：

CREATE TABLE char\_encode (

glyph CHAR(1) NOT NULL,

codepoint TINYINT(3) NOT NULL

) ENGINE=InnoDB

INSERT INTO char\_encode VALUES ('a', 97), ('b', 98);

* SESSION-1

|  |  |  |
| --- | --- | --- |
| 序号 | 执行语句 | 结果 |
| 1 | SHOW SESSION VARIABLE LIKE 'tx\_isolation' | REPEATABLE-READ |
| 2 | START TRANSACTION; |  |
| 3 | SELECT \* FROM char\_encode; | a | 97  b | 98 |

* SESSION-2

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 执行语句 | 结果 | 解释 |
|  | SHOW SESSION VARIABLE LIKE 'tx\_isolation' | REPEATABLE-READ |  |
|  | START TRANSACTION; |  |  |
|  | SELECT \* FROM char\_encode; | a | 97  b | 98 |  |
|  | UPDATE char\_encode SET codepoint = 100 WHERE glyph = 'a'; | 1 Rows affected; |  |
|  | SELECT \* FROM char\_encode; | a | 100  b | 98 | 如预期，一个事务能看到自己的本地改变 |
|  | COMMIT |  |  |

* SESSION-1

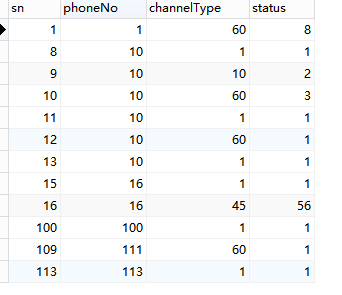
|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 执行语句 | 结果 | 解释 |
|  | SELECT \* FROM char\_encode WHERE glyph = 'a'; | a | 97 | 完美，如预期，尽管session-2提交了，但是一致性非锁定读就是应该读自己事务中第一次读取时的快照 |
|  | UPDATE char\_encode SET codepoint = codepoint + 1 WHERE glyph = 'a'; | 1 Rows affected; |  |
|  | SELECT \* FROM char\_encode WHERE glyph = 'a'; | a | 101 | 刚才读才是97，更新后应该是97+1=98，为什么不是快照读了？因为update语句或select for update 会更新快照，和之前的快照不一致 |

根据[4]中的官方解释，mysql允许覆盖更新这种行为（不遵守first commit win rule)，这让它产生了上面的这种幻像（但又不是幻读）。

1. **隔离级别与binlog**

一般来说，如果是RC隔离级别，我们必须将binlog设置为Row。这不仅是数据记录更为详细，更重要的是binlog是在事务commit后才生产，由于RC没有next-key lock只有行锁，只能锁定具体的行而不是对满足条件的索引进行锁定，使用statement格式会生成和当前主库中不一致的binlog，在执行master-slave复制时出错。而RR隔离级别利用nextkey-lock则可以避免这个情况。具体参见后文。

例如数据如下：



1. **RR——安全使用binlog-Statement**

对于RR隔离级别，由于在tx1已经对索引phoneNo=16前后加入了next keylock且未提交，tx2此时试图将sn=1（sn也是辅助索引）的phoneNo修改为16，对索引phoneNo而言是执行了插入phoneNo=16的操作，显然在nextkey-lock锁范围内，此时tx2阻塞等待tx1释放next key lock。此时事务提交顺序tx1, tx2。对应statement类型的binlog为：

*Tx1: update smstest t set t.status = 9 where t.phoneNo='16'*

*Tx2: update smstest t set t.phoneNo='16' where t.sn='1*';

|  |  |  |
| --- | --- | --- |
| tx1 | tx2 | 解释 |
| set tx\_isolation='repeatable-read'; |  | 设置当前session事务隔离级别为RR |
| BEGIN;  update smstest t set t.status = 9 where t.phoneNo='16' |  | 对索引phoneNo=16前后加入了next keylock |
|  | set tx\_isolation='read- read';  BEGIN; |  |
| update smstest t set t.phoneNo='16' where t.sn='1'; | tx2此时试图将sn=1（sn也是辅助索引）的phoneNo修改为16，对索引phoneNo是执行了插入phoneNo=16的操作，显然在tx1的next keylock锁范围内，tx2阻塞等待tx1释放锁才能执行commit |
| commit; | 阻塞 |
| commit; |  |  |
|  |  | 此时tx2 commit才成功 |

1. **RC——数据库状态与binlog-Statement不一致**

对于RC隔离级别，情况如下：

|  |  |  |
| --- | --- | --- |
| tx1 | tx2 | 解释 |
| set tx\_isolation='read-committed'; |  | 设置当前session事务隔离级别为RC |
| BEGIN;  update smstest t set t.status = 9 where t.phoneNo='16' |  | 对索引phoneNo=16只加入了行锁 |
|  | set tx\_isolation='read-committed';  BEGIN;  update smstest t set t.phoneNo='16' where t.sn='1'; | tx2此时试图将sn=1（sn也是辅助索引）的phoneNo修改为16，对索引phoneNo是执行了插入phoneNo=16的操作，由于只有行锁直接成功 |
| commit; | 成功 |
| commit; |  |  |

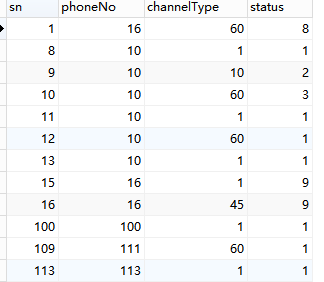
由于是RC隔离级别，只有行锁，tx2会直接执行成功，此时产生的statement 的binlog如下：

*Tx2: update smstest t set t.phoneNo='16' where t.sn='1*';

*Tx1: update smstest t set t.status = 9 where t.phoneNo='16'*

显然RC隔离级别下，数据库中数据和binlog不是等价的。

master数据库中实际数据如下：



而按照上面的binlog，在slave中所有*phoneNo=16*的status都等于9.

所以RC情况下binlog-format只能是row，不能是statement。

1. **生产环境的隔离性**

一般来说，普通电商业务并没有RR实际使用场景，所以一般生产环境使用RC隔离级别+row binlog，使用row 类型Binlog还方便使用canal等中间件监听数据变化情况。当然，一般来说RR和RC在速度上差别不大，RR+row类型binlog效率会慢很多。

**参考**

1. InnoDB 的意向锁有什么作用？<https://www.zhihu.com/question/51513268>
2. MySQL 中隔离级别 RC 与 RR 的区别<https://www.cnblogs.com/digdeep/archive/2015/11/16/4968453.html>
3. mysql的innodb通过nextkey lock解决了幻读，为什么还说默认隔离级别是可重复读<https://www.zhihu.com/question/350352149/answer/865763594>
4. 既然MySQL中InnoDB使用MVCC，为什么REPEATABLE-READ不能消除幻读？<https://www.zhihu.com/question/334408495/answer/745098902>
5. mysql解决RR下的write skew <https://bugs.mysql.com/bug.php?id=63870>
6. MySQL隔离级别为读提交的时候为什么会出现的数据和日志不一致，还必须把binlog格式设置为row？https://www.zhihu.com/question/344037151