# Ss

# MySQL执行计划

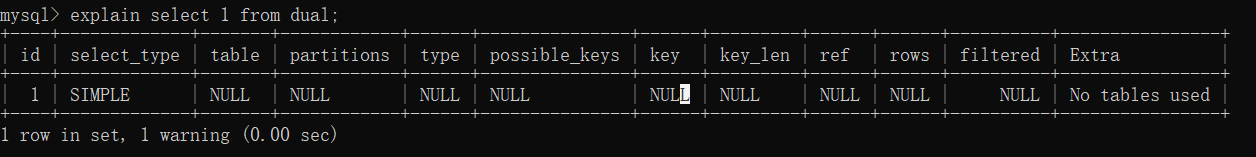
SQL性能问题：

a.分析SQL的执行计划：explain+select语句

b.MySQL查询优化器会干扰我们的优化

MySQL优化（官网）：https://dev.mysql.com/doc/refman/8.0/en/optimization.html

## 查看执行计划(explain+SQL语句)



参数说明：

id 编号

select\_type 查询类型

table 表

type 索引类型

possible\_key 可能会用的索引

key 实际使用的索引（为null说明不使用索引）

key\_len 实际使用索引的长度(针对联合索引)

ref 表之间的引用

rows 查询到的数据量

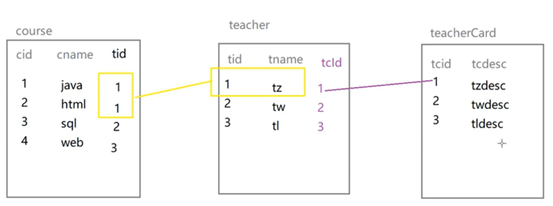
extra 额外的信息

## 编号（id）

select查询的序列号，包含一组数字，表示查询中执行select子句或操作表的顺序。

**结论：若id值相同，从上往下顺序执行；若id值不同，值越大越优先查询。**

测试实例：



准备表：

create table course (cid int(3), cname varchar(20), tid int(3));

create table teacher(tid int(3), tname varchar(20), tcid int(3));

create table teacherCard(tcid int(3), tcdesc varchar(200));

准备数据：略

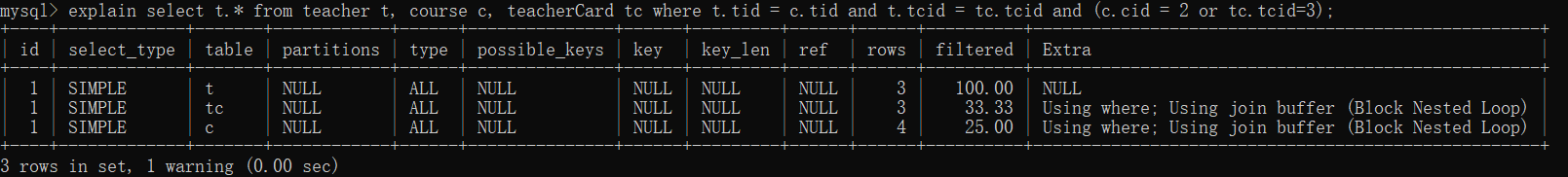
**###测试1：多表查询**

描述：查询课程编号cid为2、或教师证编号tcid为3 的老师信息

explain select t.\* from teacher t, course c, teacherCard tc where t.tid = c.tid and t.tcid = tc.tcid and (c.cid = 2 or tc.tcid=3);

执行顺序：t->tc->c 数据量分别是：3->3->4。

结论1：id值相同，从上往下，顺序执行。



另：

当t表数据为6条时的执行顺序为：tc->c->t 数据量：3->4->6

结论：表的执行顺序：因数量个数改变而改变，原因是：笛卡儿积。

2\*3 = 6 \* 4 = 24

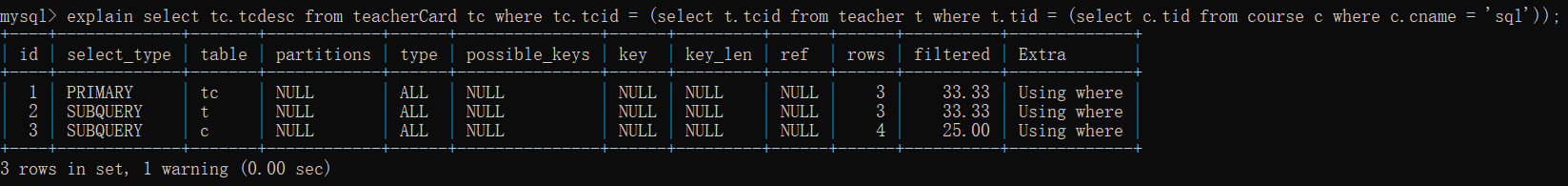
3\*4 = 12 \* 2 = 24

结论2：数据量最小的表优先查询。

**###测试2：子查询**

描述：查询教SQL课程的老师描述：（能看到id值不一样）

explain select tc.tcdesc from teacherCard tc where tc.tcid = (select t.tcid from teacher t where t.tid = (select c.tid from course c where c.cname = 'sql'));



结论3：id值越大越优先查询。

**###测试3：子查询+多表**

能看到id既有一样，也有不一样的。

## 索引类型（type）

**取值：system > const > eq\_ref > ref > range > index > all**

优化前提：有索引

结论：值越往左边性能越高。其中system和const只是理想情况，实际能达到ref>range

system：忽略，只有一条数据的系统表；或 衍生表只有一条数据的主查询。

const：仅能查到一条数据SQL，用于primary key或unique索引，即查询类型与索引类型有关。

如：explain select id from t\_user id = 2;

eq\_ref：唯一性索引，对于每个索引键的查询，仅匹配唯一一行数据（仅有1个，不能多，不能0），常见于唯一索引或主键索引。结果多条，但每条数据是唯一的。

如：explain select..from t\_user where username = ‘aaa’

alter table 表名 add constaint 主键名 primary key(列名); --创建主键

alter table 表名 add constaint 索引名 unique index(列名); --创建唯一性索引

示例：

explain select t.tcid from teacher t, teacherCard tc where t.tcid = tc.tcid

说明：用到的索引是teacher表中的tcid字段；若teacher表的数据个数和teacherCard的数据个数一致（都是3条数据），则有可能满足eq\_ref级别，否则无法满足。

ref：非唯一索引，对于每个索引键的查询，返回匹配的所有行（0/多条）。

range：检索指定范围的行，where后面是一个范围查询(between,>,>=,<,<=等)

index：查询全部索引中的数据

如：explain select tid from t\_teacher; --tid是索引

all：查询全部数据，无索引

如：explain select birthday from t\_user; --birthday不是索引，需要全表扫描

## 实际用到的索引（key）

若possible\_keys/key为null，则说明没用上索引

## 索引使用的长度（key\_len）

作用：用于判断联合索引是否被完全使用。

示例1：

create table test01( name char(20) not null default '' );

alter table test01 add index idx\_name(name);

explain select \* from test01 where name='';

结果：key\_len为60，在utf8中1个字符占3个字节

示例2：

alter table test01 add column name1 char(20); --name1可以为null

alter table test01 add index idx\_name1(name1);

explain select \* from test01 where name1='';

结果：key\_len为61，若索引字段可以为null，则会使用1个字节用于标识

varchar(20)对应key\_len=20\*3 + 1(null) + 2(用2个字符标识可变长度) = 63

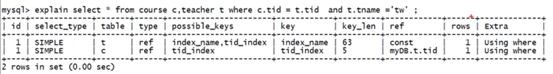
在utf8中，1个字符对应3个字节。

## 指明当前表所参照的字段（ref）

select ... where a.x = b.y; --其中b.y可以是常量，const

示例：

explain select \* from course c,teacher t where c.tid=t.tid and t.tname='tw';



## 查询出来的数据量（rows）

被索引优化查询的数据个数（实际通过索引而查询到的数据个数）

## 额外的信息（extra）

### using filesort

性能消耗大，需要“额外”的一次排序

**###示例1：单索引排序**

create table test02(

a1 char(3),

a2 char(3),

a3 char(3),

index idx\_a1(a1),

index idx\_a2(a2),

index idx\_a3(a3)

)

explain select \* from test02 where a1 = '' order by a1;

explain select \* from test02 where a1 = '' order by a2; --出现using filesort

结论1：对于单索引，若排序和查找是同一字段，则不会出现using filesort;若不是则会出现using filesort。

建议：where的字段和order by的字段相同

**###示例2：联合索引排序（不能跨列，遵循最左前缀原则）**

drop index idx\_a1 on test02;

drop index idx\_a2 on test02;

drop index idx\_a3 on test02;

alter table test02 add index idx\_a1\_a2\_a3(a1, a2, a3);

explain select \* from test02 where a1 = '' order by a3; --出现using filesort

explain select \* from test02 where a2 = '' order by a3; --出现using filesort

explain select \* from test02 where a1 = '' order by a2;

结论2：where和order by按照联合索引的顺序使用，不要跨列或无序使用

### using temporary

性能损耗大，用到了临时表。一般出现在group by或union语句中。

group by的字段不在查询字段中，因此需要多一张表将group by的字段查出来再group by。

explain select a1 from test02 where a1 in ('1', '2', '3') group by a1;

explain select a1 from test02 where a1 in ('1', '2', '3') group by a2; --出现using temporary

建议：查询字段和group by字段相同

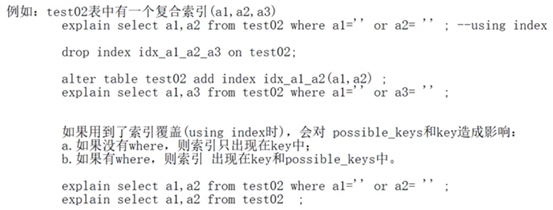
SQL解析过程：from -> on -> join -> where -> group by -> having -> select -> order by -> limit

### using index

性能提升，索引覆盖。

原因：不读取源文件，只从索引文件中获取数据，即不需要回表查询。

条件：只要使用到的字段全部都在索引中，就是索引覆盖using index。



### using where

需要回表查询

如：explain select name, age from age = XX; --若age是索引列，此语句必须回原表查出name，因此会显示using name

示例：联合索引

create table test03(

a1 int(4) not null,

a2 int(4) not null,

a3 int(4) not null,

a4 int(4) not null

);

alter table test03 add index idx\_a1\_a2\_a3\_a4(a1, a2, a3, a4);

explain select a1,a2,a3,a4 from test03 where a1=1 and a2=2 and a3=3 and a4=4; --推荐

explain select a1,a2,a3,a4 from test03 where a4=4 and a2=2 and a3=3 and a1=1; --不推荐，但MySQL优化器会对该SQL优化

结论：

a.若联合索引(a,b,c,d)和使用的顺序全一致（且不跨列使用），则联合索引全部使用。若部分一致则使用部分索引。

b.where子句和group by不要跨列使用

## explain优化案例-单表优化

示例：

create table book (

bid int(4) primary key,

name varchar(20) not null,

authorId int4) not null,

publicId int(4) not null,

typeId int(4) not null

);

# 查询authorId=1，且typeId为2或3的bid字段

explain select bid from book where typeId in (2, 3) and authorId=1 order by typeId desc;

结果：全表扫描，出现using where和using filesort

优化1：添加复合索引

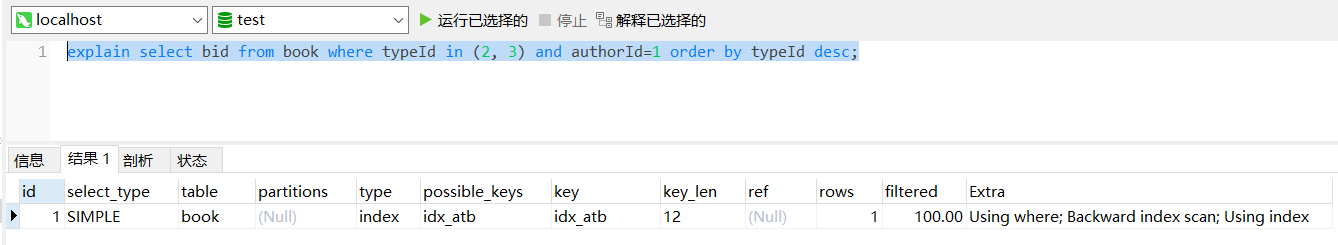
alter table book add index idx\_bta (typeId, authorId, bid);

分析：虽然可以回表查询bid，但是将bid放到索引中，可以提升使用using index。

优化2（之前是index级别）：in可能导致索引失效

drop index idx\_bta on book;

alter table book add index idx\_atb (authorId, typeId, bid);



结果：

本例中同时出现了using where(需要回原表)；using index(不需要回原表)。

原因：where authorId=1 and typeId in(2,3)中authorId在索引(authorId, typeId, bid)中，因此不需要回原表（直接在索引列表中能查到），而typeId虽然也在该索引中，但是含in的范围查询已经使该typeId索引失效，因此相当于没有typeId这个索引，所以需要回原表(using where)。

小结：

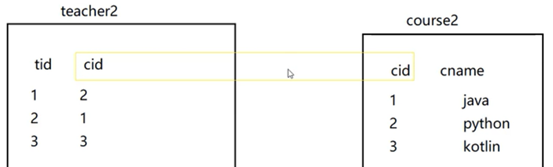
a.索引不能跨列使用（最左前缀），保持索引的定义和使用的顺序一致性。

b.索引需要逐步优化

c.将含in的范围查询放到where条件的最后，防止索引失效

## explain优化案例-双表优化

示例：



左连接：select \* from teacher2 t left outer join course2 c on t.cid = c.cid where c.cname='java';

小表驱动大表。where 小表.x = 大表.x;

索引建立在经常使用的字段上。（本题 on t.cid = c.cid可知，t.cid字段使用频繁，因此给该字段加索引）

结论：一般情况下对于左外连接，给左表加索引；对于右外连接，给右表加索引。

当编写..on t.cid = c.cid时，将数据量小的表放左边（假设此时t表数据量小）

**若小表10条数据，大表300条数据，分析如下：**

分析1：select .. where 小表.x = 大表.y;

for (int i = 0; i < 小表.length; i++) {

for (int j = 0; j < 大表.length; j++) {

...

}

}

分析2：select .. where 大表.x = 小表.x;

for (int i = 0; i < 大表.length; i++) {

for (int j = 0; j < 小表.length; j++) {

...

}

}

结论：两个for循环的循环次数都相同；但是对于双层循环来说，数据小的循环放外层，数据大的循环放内层。

## explain优化案例-三表优化

优化原则：

a.小表驱动大表

b.索引建立在经常查询的字段上

# MySQL二进制日志(binlog)

## binlog概述

binlog是用于记录表结构变更(create，alter table)和表数据变更(insert，update，delete)，并不会记录select和show语句这类操作，以二进制的形式保存到磁盘中。

binlog是mysql的逻辑日志，并且由Server层进行记录，使用任何存储引擎的mysql数据库都会记录binlog日志。

binlog是通过追加的方式进行写入的，可以通过max\_binlog\_size参数设置每个binlog文件的大小，当文件大小达到给定值之后，会生成新的文件来保存日志。

**binlog用于归档，主要用于主从复制和数据恢复；redo log用于崩溃恢复；undo log用于实现MVCC和事务回滚支持。**

附：

逻辑日志：可以简单理解为记录的就是sql语句。

物理日志：因为mysql数据最终是保存在数据页中的，物理日志记录的就是数据页变更。

二进制日志启动开关：log-bin [=file\_name]

查看是否开启binlog：show variables like 'log\_%';

查看二进制日志的工具：mysqlbinlog.exe

Binlog包括两种文件：

-1）索引文件：binlog.index文件，用于记录所有的二进制文件

-2）日志文件：binlog.00000\*文件，用于记录数据库所有的DDL和DML语句，分两部分，一部分在缓存，一部分在硬盘，缓冲达到一定次数，会刷到磁盘里。

## binlog日志格式

binlog日志有三种格式，分别为STATMENT、ROW和MIXED。

在 MySQL 5.7.7之前，默认的格式是STATEMENT，MySQL 5.7.7之后，默认值是ROW。日志格式通过binlog-format指定。

**STATMENT 基于SQL语句的复制**(statement-based replication, SBR)，每一条会修改数据的sql语句会记录到binlog中。优点：不需要记录每一行的变化，减少了binlog日志量，节约了IO, 从而提高了性能；缺点：在某些情况下会导致主从数据不一致，比如执行sysdate()、slepp()等。

**ROW 基于行的复制**(row-based replication, RBR)，不记录每条sql语句的上下文信息，仅需记录哪条数据被修改了。优点：不会出现某些特定情况下的存储过程、或function、或trigger的调用和触发无法被正确复制的问题；缺点：会产生大量的日志，尤其是alter table的时候会让日志暴涨

**MIXED 基于STATMENT和ROW两种模式的混合复制**(mixed-based replication, MBR)，一般的复制使用STATEMENT模式保存binlog，对于STATEMENT模式无法复制的操作使用ROW模式保存binlog

## binlog刷盘机制

对于InnoDB存储引擎而言，只有在事务提交时才会记录biglog，此时记录还在内存中，那么biglog是什么时候刷到磁盘中的呢？mysql通过sync\_binlog参数控制biglog的刷盘时机，取值范围是0-N：

0：不去强制要求，由系统自行判断何时写入磁盘；

1：每次commit的时候都要将binlog写入磁盘；

N：每N个事务，才会将binlog写入磁盘。

从上面可以看出，sync\_binlog最安全的是设置是1，这也是MySQL 5.7.7之后版本的默认值。但是设置一个大一些的值可以提升数据库性能，因此实际情况下也可以将值适当调大，牺牲一定的一致性来获取更好的性能。

## binlog应用场景

**binlog常见的三个应用场景：**

**1）数据恢复：**删除库后，根据mysqlbinlog工具恢复数据

教程：<https://www.iteblog.com/mysql-binlog_basic_usage/>

如：mysqlbinlog  data\mysql-bin.000002 --stop-pos=653 | mysql -uroot -p123456

**2）主从复制：**

主库有一个log dump线程，将binlog传给从库；

从库有两个线程，一个I/O线程，一个SQL线程，I/O线程读取主库传过来的binlog内容并写入到relay log，SQL线程从relay log里面读取内容，写入从库的数据库。

**3）审计：**用户可以通过二进制日志中的信息来进行审计，判断是否有对数据库进行注入攻击。

# MySQL 事务日志（redo&undo）

## 概述

InnoDB事务日志包括redo log和undo log。undo log指事务开始之前先将需要操作的数据备份；redo log指事务中操作的任何数据，将最新的数据备份到一个地方。

作用：实例或介质失败，事务日志文件就能派上用场。

1、redo log不是随着事务的提交才写入的，而是在事务的执行过程中，便开始写入redo中。具体的刷盘策略可以进行配置。防止在发生故障的时间点，尚有脏页未写入磁盘，在重启mysql服务的时候，根据redo log进行重做，从而达到事务的未入磁盘数据进行持久化这一特性。Redo Log是为了实现事务的持久性而出现的产物。

2、undo log用来回滚记录到某个版本。事务未提交之前，undo保存了未提交之前的版本数据，undo中的数据可作为数据旧版本快照供其他并发事务进行快照读。是为了实现事务的原子性而出现的产物,在Mysql innodb存储引擎中用来实现多版本并发控制。

示例：

假设有A、B两条数据，值分别为1、2，开启事务（把1改为3，2改为4），那么实际的记录如下：

A.事务开始

B.记录A=1到undo log

C.修改A=3

D.记录A=3到redo log

E.记录B=2到undo log

F.修改B=4

G.记录B=4到redo log

H.将redo log写入磁盘（刷盘策略）

I.事务提交

说明：

undo+redo的设计主要考虑的是提升IO性能，增大数据库吞吐量。可看出B D E G H均是新增操作，但是B D E G是缓冲到buffer区，只有G是增加了IO操作。

## redo log

### 概述

## undo-log

undo-log是为了回滚而用的，具体内容就是copy事务前的数据库内容到undo buffer，在适合的时间把undo buffer中的内容刷新到磁盘。undo buffer与redo buffer一样，也是环形缓冲，但当缓冲满时，undo buffer中的内容也会被刷新到磁盘；与redo log不同的是，磁盘上不存在单独的undo log文件，所有的undo log均存放在主ibd数据文件中（表空间），即使客户端设置了每表一个数据文件也是如此。

# MySQL MVCC介绍

参考：https://www.bilibili.com/read/cv6580973

## 概述

MVCC（Mutil-Version Concurrency Control，基于多版本的并发控制协议），**MySQL InnoDB存储引擎**的实现，其基本思想是为每次事务生成一个新版本的数据，在读取数据时选择不同版本的数据即可实现对事务结果的完整性读取。

最大特点是读不加锁，读写不冲突。

作用范围：MVCC只在 READ COMMITTED和REPEATABLE READ两个隔离级别下工作，其它两个隔离级别够和MVCC不兼容（READ UNCOMMITTED 总是读取最新的数据行，而不是符合当前事务版本的数据行， SERIALIZABLE 则会对所有读取的行都加锁）。

MVCC是一种规范，不仅是MySQL，包括Oracle，PostgreSQL等其它数据系统都实现了MVCC，但各自的实现机制不尽相同，因为MVCC没有一个统一的实现标准。

MVCC为了解决什么问题？

锁机制可以控制并发操作，但其系统开销较大，而MVCC在大多数情况下代替行级锁，能降低其系统开销。

使用版本来控制并发情况下的数据问题，在事务B开始修改某一行数据后，事务A需要读取这行数据，此时会读取到事务B修改之前的副本数据，但如果事务A也要修改这行数据时则必须要**等待事务B提交事务**，若事务B长时间未提交事务，则事务A会回滚修改操作。

## MVCC实现

MVCC的实现通过**版本链**和**Read View**来实现。通过read-view机制与undo版本链比对机制，使得不同的事务会根据数据版本链对比规则读取同一条数据在版本链上的不同版本数据。

每条记录后面都保存了两个隐藏列：事务ID（DB\_TRX\_ID）和回滚指针（DB\_ROLL\_PTR）。每开启一个新的事务，都会产生一个自动递增的事务id。每次更新操作都会拷贝原数据到undo log日志中，然后修改该行记录，并把该行记录的回滚指针指向刚才undo log中，这样形成版本链。查询的时候会查询Read View，通过与版本链上的事务ID比较，找到对应记录。

步骤：

1、事务的排它锁形式修改数据

2、修改之前先把数据放到undo log日志，通过回滚指针关联，形成版本链

3、成功了什么都不做，失败了则从undo log回滚数据

### 数据行结构和undo版本链

**行结构**：InnoDB在每行数据后面都增加了一些隐藏字段

记录ID (DB\_ROW\_ID)：6字节，若表设置了主键，则改值就是主键；若无主键，则会使用定义的第一个唯一索引；若无唯一性索引，则会默认生成一个值。

**事务ID(DB\_TRX\_ID)**：6字节，表示这条记录的最新事务id。

**回滚指针(DB\_ROLL\_PTR)**：7字节，指向这条记录的undo log记录，通过该字段能找到修改前的数据。

**删除标记(DELETE BIT)**：用于标记该记录是否被删除。

说明：

DB\_ROW\_ID跟事务无关系，DB\_TRX\_ID和DB\_ROLL\_PTR才是跟事务有关系的。

示例：

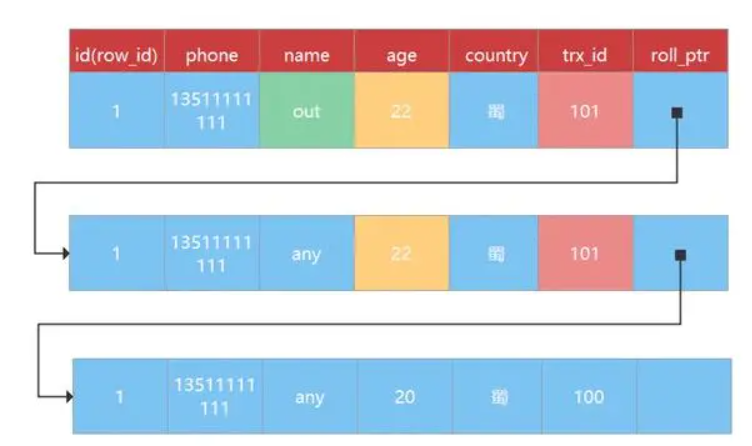
start transaction;

update person set age = 22 where id = 1;

update person set name = 'out' where id = 1;

commit;

当执行完上面两条语句之后，但还没有提交事务之前，它的版本链如下：



### Read View

Read View用来判断每一个读取语句有资格读取版本链中的哪条记录。所以在读取之前，都会生成一个Read View，然后根据生成的Read View再去读取版本链上的记录。

在事务中，只有执行插入、更新、删除操作时才会分配到一个事务 id。如果事务只是一个单纯的读取事务，那么它的事务 id 就是默认的 0。

Read View保证MySQL事务的隔离性，只针对读已提交和可重复读这两种隔离级别。可重复读返回的read-view是第一条记录的，读已提交每次查询都返回新的read-view。

问：开启一个事务会自动创建read-view，那最大事务id不是一直都是当前事务id吗？

开启一个事务，并不会立即创建read-view，在等有数据查询操作时才会创建。因此在这段时间内可能会有新事务开启，从而read-view的最大事务id就不一定是当前事务id了。

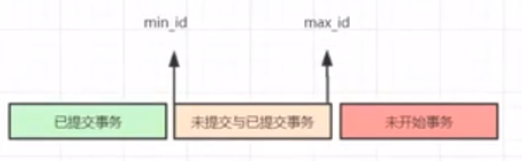
**ReadView结构：**

rw\_trx\_ids：当前未提交事务id数组

min\_trx\_id：未提交最小事务id。生成ReadView时，赋值为当前已提交的事务id+1，即rw\_trs\_id的最小值

max\_trx\_id：未提交最大事务id

curr\_trx\_id：当前事务id



MySQL查询时会读取ReadView，并根据ReadView从版本链中最新记录依次往下找：

1. trx\_id < min\_trx\_id：表示这个版本是已提交的事务生成，这个数据是可见的。
2. 若min\_id <= trx\_id <= max\_id，这里包括两种情况：

a.若row的trx\_id在数组中，表示这个版本由还没提交的事务生成的，不可见（当前自己的事务是可见的）；

b.若row的trx\_id不在数组中，表示这个版本是已经提交了的事务生成的，可见。

1. trx\_id > max\_trx\_id：表示这个版本是由将来启动的事务生成的，是肯定不可见的。

trx\_id in (rw\_trx\_ids)情况（min\_id <= trx\_id <= max\_id另外说法）：

表示该版本由未提交的事务生成，可不见

trx\_id = curr\_trx\_id：该记录在当前事务可见，因为修改该版本记录的事务就是当前事务。

查询操作：

select \* from t --创建了查询快照，记录执行sql这一刻已提交的事务id

当执行查询sql时会生成一致性视图read-view，它由执行查询时所有未提交事务id数组（数组里最小的id为min\_id）和已创建的最大事务id（max\_id）组成，查询的数据结果需要跟read-view做对比从而得到快照结果。

删除操作：

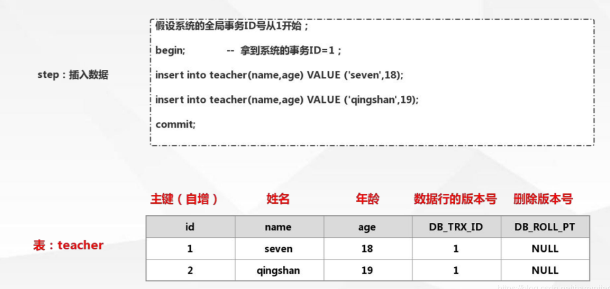
可以认为是update的特殊情况，会将版本链上最新的数据复制一份，然后将trx\_id修改成删除操作的trx\_id，同时在该条记录的头信息(record header)里的(delete flag)标记位写上true，表示该记录已被删除，在查询时按照上面的规则查询到对应的记录如果delete\_flag为true将不返回。

## 案例1：增删改查操作

MySQL的每次操作都会生成全局事务ID。

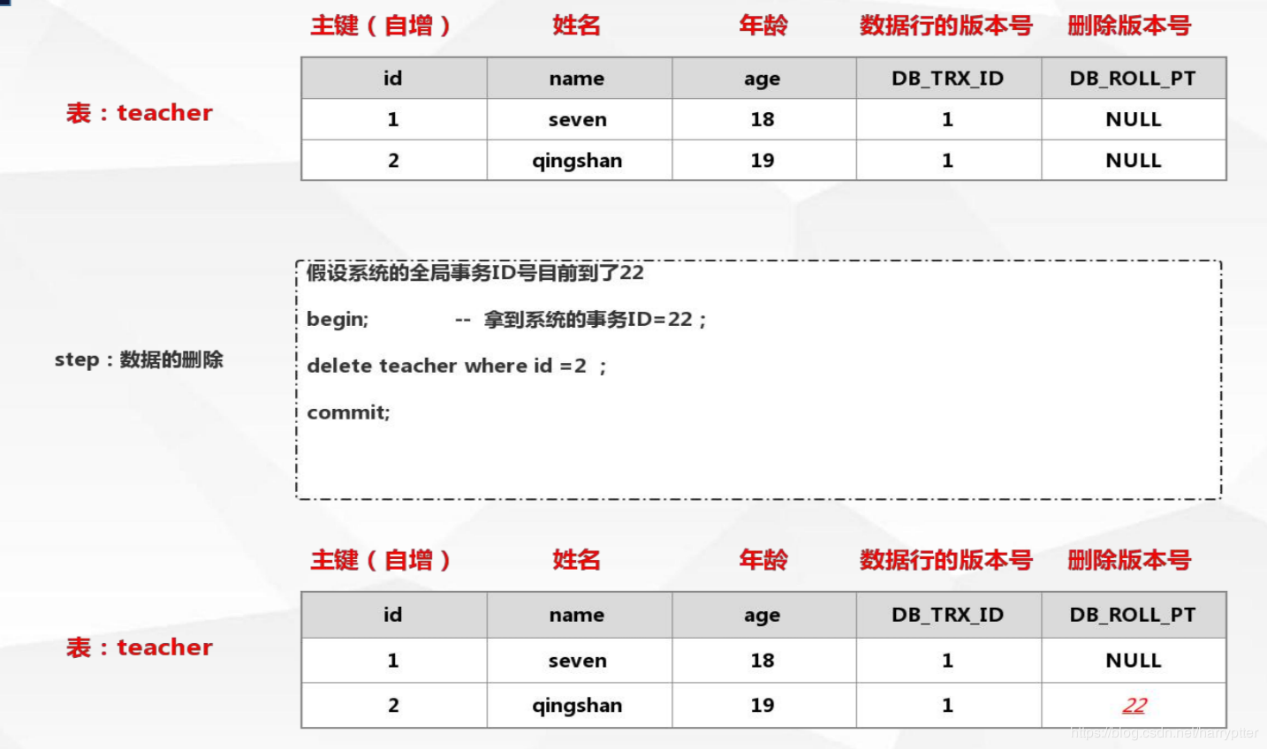
### 插入数据

插入规则：DB\_TRX\_ID设置为当前全局事务ID，DB\_ROLL\_PT设置为NULL。



### 删除数据

删除规则：DB\_ROLL\_PT设置为当前全局事务ID。



如图，在事务中删除id为2的数据，那么在数据的DB\_ROLL\_PT为当前事务ID。

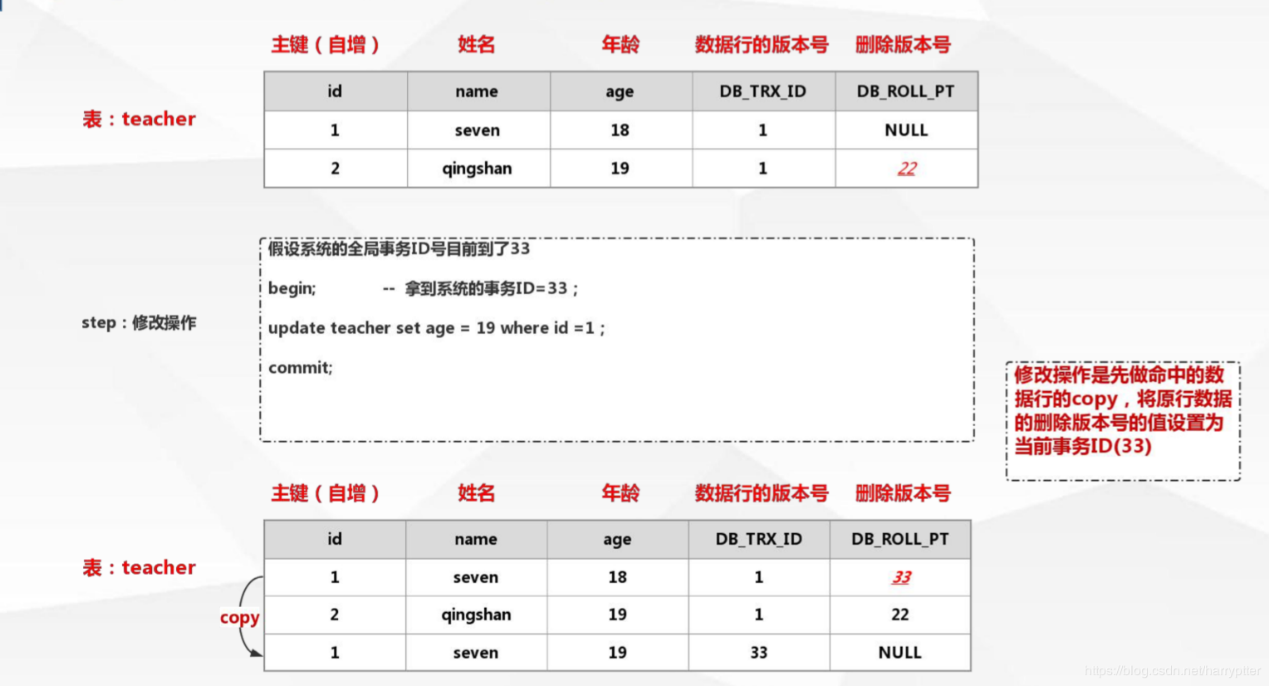
### 修改数据

修改规则：（跟案例3的反了）

1）先做命中的数据行的复制；

2）将旧的行数据的删除版本号设置为当前全局事务ID；

3）将新的行数据的数据行版本号也设置为当前全局事务ID，删除版本号为NULL。



如图，在事务中更改id为1的数据信息， 先复制元数据，然后将元数据删除版本号改为当前事务全局版本号，再插入新的数据，将数据行版本号设置为当前版本号。

### 查找数据

查找规则：

1）查找数据行版本号小于或者等于当前全局事务ID（即确认事务读取的行在开始前已经存在，或者是事务自身插入或者修改的）；

2）查找删除版本号为NULL、或者大于当前事务版本号的ID（即确保读取出来的行在此事务开启之前没有被删除）

公式：

1) CURRENT\_DB\_TRX\_ID <= DB\_TRX\_ID

2) DB\_ROLL\_PTR = NULL || DB\_ROLL\_PTR > DB\_TRX\_ID



## 案例2：不同事务操作同一数据

描述：在事务B开始修改某一行数据后，事务A需要读取这行数据，此时会读取到事务B修改之前的副本数据，但如果事务A也要修改这行数据时则必须要**等待事务B提交事务**，若事务B长时间未提交事务，则事务A会回滚修改操作。

**示例1：开启两个客户端A和B，分别作为两个事务**

步骤1：开启一个客户端A，开启事务，并增加一条数据

mysql> start transaction;

mysql> insert into tb\_user(username, age) values('aaa', 35);

步骤2：开启一个客户端B，开启事务，并查询user表

mysql> start transaction;

mysql> select \* from tb\_user; #并没有查到事务A刚插入的数据

步骤3：客户端A提交事务，客户端B再次查询

mysql> commit; #客户端A执行语句

mysql> select \* from tb\_user; #客户端B执行语句，还是没有查询到事务A刚插入的数据

步骤4：事务B提交事务

mysql> commit;

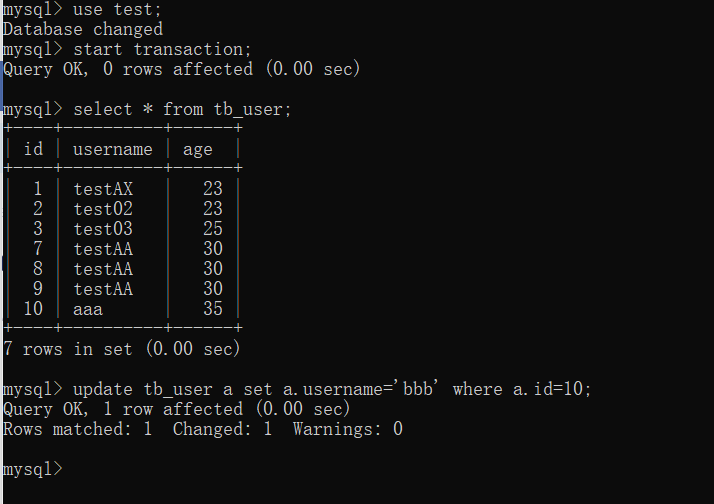
步骤5：客户端B再次执行查询能看到数据（此时是新的事务，事务A和事务B都已提交，因此能看到它们的数据）

mysql> select \* from tb\_user;

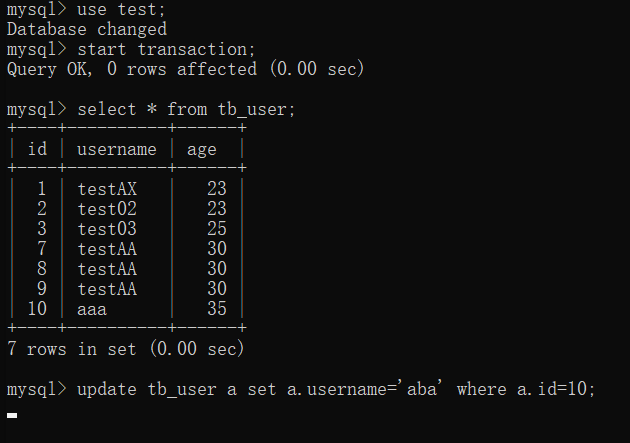
**示例2：同时开启两个客户端模拟开启多个事务，然后同时对同一条记录更新**

结论：若事务A对id=1的数据进行修改，但还没有提交。这时候事务B也对id=1的数据进行修改，这时候事务B会被挂起，直到事务A完成了（回滚或提交）。

步骤1：打开客户端A，开启事务，执行更新操作



步骤2：打开客户端B，开启事务，执行更新操作（能看到这里会阻塞）



阻塞过了一段时间后会报错：

mysql> update tb\_user a set a.username='aba' where a.id=10;

ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction

步骤3：客户端B再次进行数据修改，然后客户端A提交事务（在A提交后，B立马能写入成功）

mysql> update tb\_user a set a.username='aba' where a.id=10; #事务B

mysql> commit; #事务A

步骤4：客户端A再次进行查询

mysql> select \* from tb\_user; #能看到刚才事务A已修改的数据，但不能看到事务B修改的事务

步骤5：客户端B再次进行查询

mysql> select \* from tb\_user; #能看到事务B修改的数据

步骤6：客户端B提交事务，客户端A进行查询

mysql> commit; #客户端B

mysql> select \* from tb\_user; #客户端A，能够看到B提交的数据

## 案例3：多个事务修改操作同一记录？（这个为准）

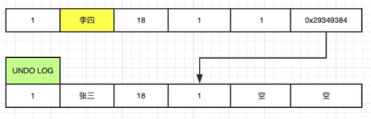
新增规则：DB\_TRX\_ID设置为当前全局事务ID，DB\_ROLL\_PT设置为NULL。

1）事务A进行insert操作（感觉图画错）：



修改规则：a.用排它锁锁定这条记录；b.拷贝这条记录到undo log日志中，并把这条记录的DB\_ROLL\_PTR指向undo log中；c.修改DB\_TRX\_ID为当前事务ID。

2）事务A进行update操作：

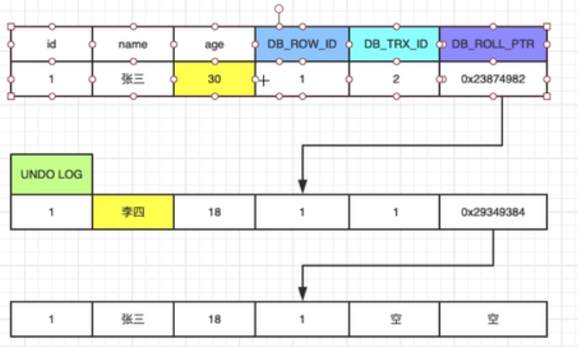


3）事务A对数据修改后并未提交，若此时事务B也对这条数据进行修改（会进行等待，直到事务A提交，或事务B执行超时出现报错）。

4）事务A对数据修改并提交后，若此时事务C对这条数据进行修改。

描述：事务A修改了数据并提交（name从张三改成李四），这时事务C将age从18改成30。

流程：a.用排它锁锁定这条记录；b.拷贝这条记录到undo log日志中，并把这条记录的DB\_ROLL\_PTR指向undo log中。c.修改DB\_TRX\_ID为当前事务ID。



说明：

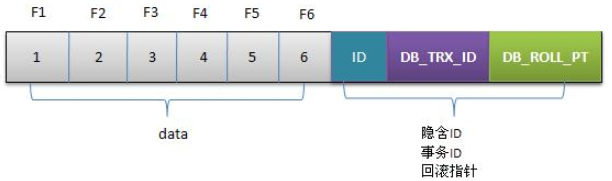
与事务A相同的操作，此时undo log中有两行记录，并且通过回滚指针连在一起。

因此，如果undo log一直不删除，则会通过当前记录的回滚指针回溯到该行创建时的初始内容，所幸的时在Innodb中存在purge线程，它会查询那些比现在最老的活动事务还早的undo log，并删除它们，从而保证undo log文件不至于无限增长。

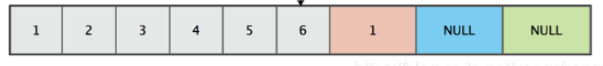
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*另外一个示例\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**示例：**

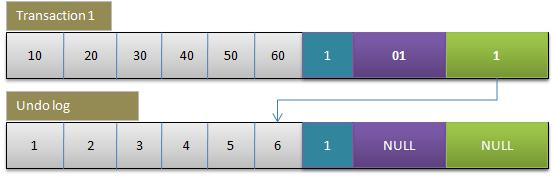
**1.初始数据行(insert)**



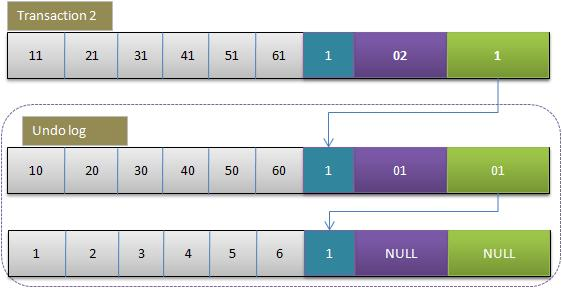
**事务1插入数据：假设这条数据是刚insert的，可以认为ID为1，其它两个字段为空。**



**2.事务1修改该行的值。**



**3.事务2修改该行的值**

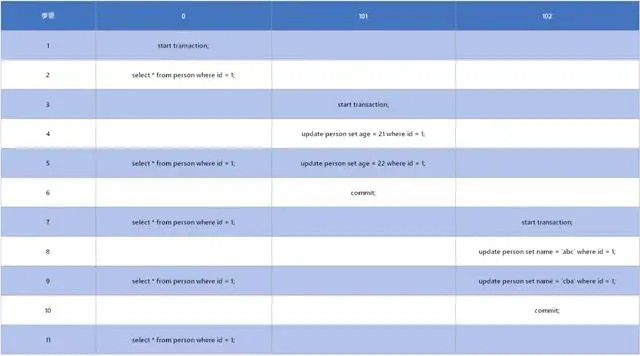


说明：

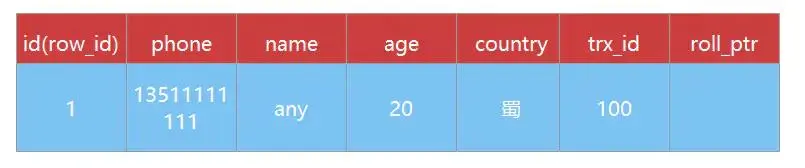
与事务2相同，此时undo log中有两行记录，并且通过回滚指针连在一起。

## 案例4（比较全面）

示例：



步骤1：开启一个读取事务，因为只读事务，所以它的事务id为0。紧接着我们查询id为1的记录。此时版本链如下：



注：跟红色表头连接在一起的记录都是在 B+ 树中的，而通过 roll\_ptr 指针连接的记录都是存在于 undo log 中的。以下的所有版本链都是这种形式。

### Read Committed

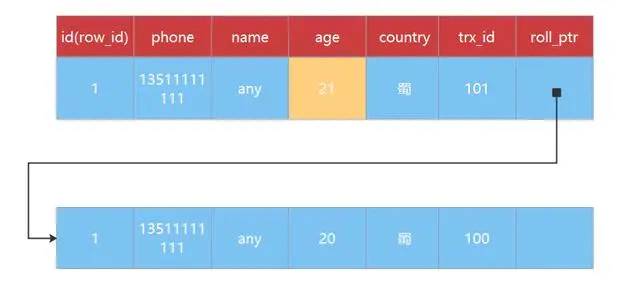
在 READ COMMITTED 隔离级别下，会使用 MVCC。在开启一个读取事务之后，它会在每一个 select 操作之前都生成一个 Read View。

步骤1：开启一个事务0。

步骤2：select读取时，没有活跃的事务，表明所有的事务都是已经提交了，所以它能读取到第一条记录。

步骤3：开启新的事务101

步骤4：事务101修改了id=1的记录，此时版本链如下：



步骤5：事务0执行select操作，事务0会重新生成ReadView。

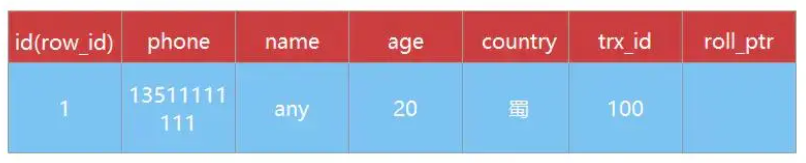


我们根据上面版本链规则：

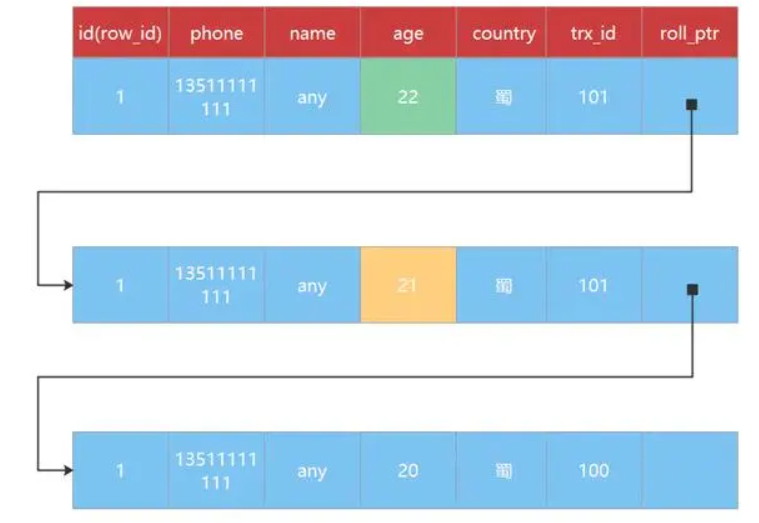
1.版本链中的第一条记录，它的trx\_id 不小于ReadView的min\_trx\_id（不满足条件1），所以该记录不可见。

2.通过roll\_ptr找到版本链的第二条记录，它的trx\_id小于min\_trx\_id，所以该记录可见。

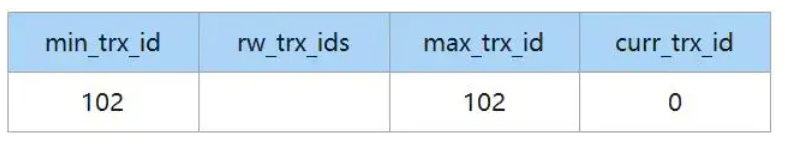
所以对于此次的查询结果是：



事务101在步骤5中执行更新操作，执行步骤6，提交该事务之后，版本链如下：



步骤7：事务0执行select操作，因为是Read Committed隔离界别，会重新生成一个Read View。

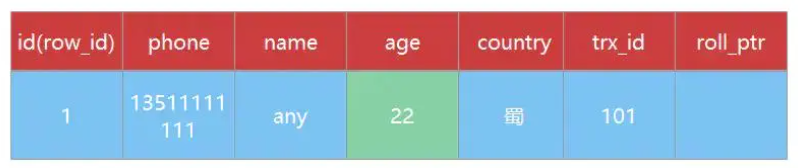


根据版本链规则：

因为没有活跃的事务，可知所有事务都已经提交，所以rw\_trx\_ids为空。

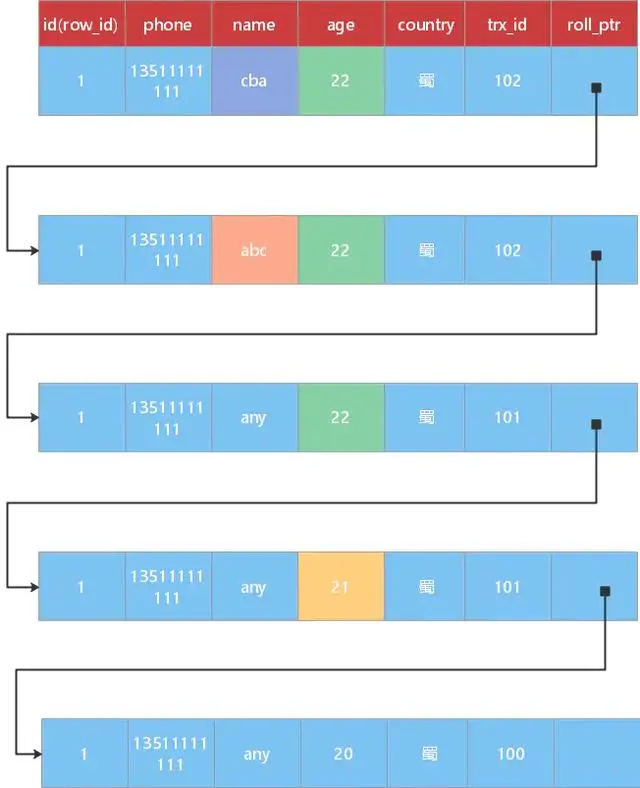
版本链第一条记录，它的trx\_id小于min\_trx\_id，所以此记录可见。

所以对于此次的查询结果是：

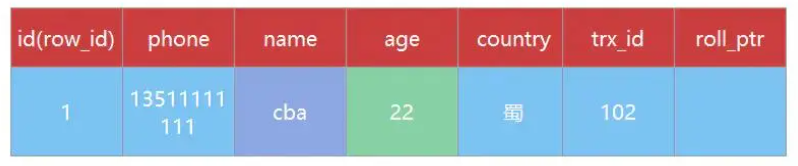


步骤7：事务0执行查询完之后，又开启一个新的事务102。

步骤8和步骤9执行update操作后，最终版本链如下：



步骤11：根据版本链规则，它能获取到的记录：

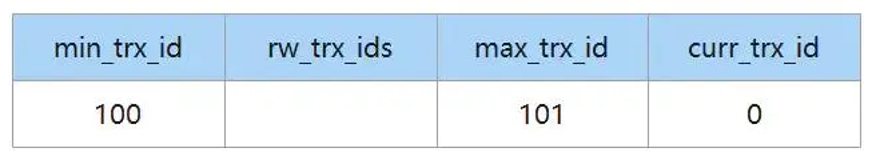


### Repeatable Read

实际上，REPEATABLE READ 与 READ COMMITTED 的区别只有在生成 Read View 的时机上。

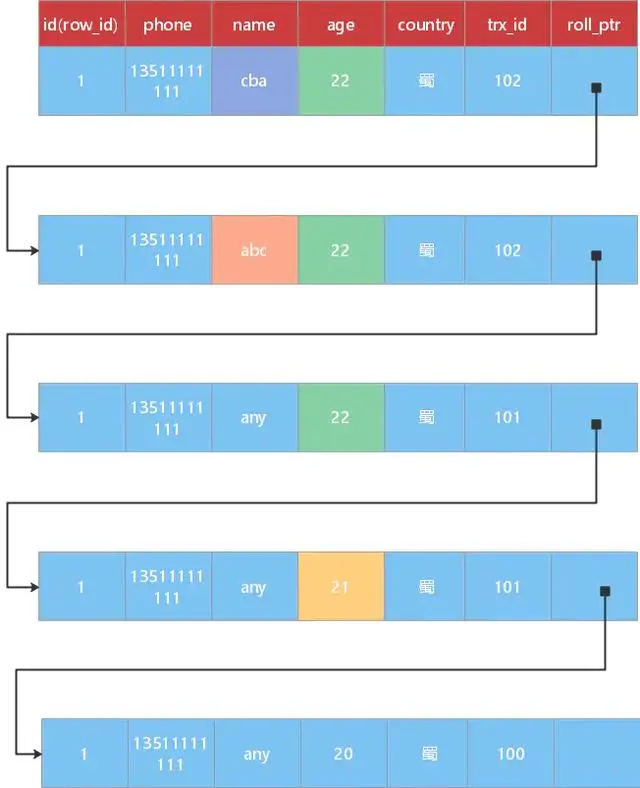
READ COMMITTED 是在每次执行 select 操作时，都会生成一个新的 Read View。而 REPEATABLE READ 只会在第一次执行 select 操作时生成一个 Read View，直到该事务提交之前，所有的 select 操作都是使用第一次生成的 Read View。

首先，执行到步骤 2，事务 0 开启了事务之后，并执行一次 select 查询。此时会生成一个 Read View。该 Read View 的结构如下：



生成的 Read View 将会一直使用，直到事务 0 提交。

所以，尽管后面的开启了两个事务，并且对记录进行修改，使得最终的版本链变为如下所示：



但是事务 0 依然只能读取到最开始的那条记录，也就是：



不管事务 0 在任何时候执行 select \* from person where id = 1; 读取记录，那么它都只会使用第一次生成的 Read View 在版本链中选择可以读取的记录。

# 相关问题

## int类型主键到最大值后出现什么问题？

MySQL的int，会默认采用有符号类型，数值范围为-2的32次方到2的31次方-1。

答案是一直都是2147483647（int类型最大值），所以就会导致主键冲突！

## 索引优缺点

优点：查的快，支持range

缺点：大部分查询实际需要回表，索引建立会额外消耗内存和磁盘，对开发者的sql也有要求。

建议字段：区分度大的字段

## 时间类型DATE、DATETIME和TIMESTAMP

DATE、DATETIME和TIMESTAMP 表达的时间范围

|  |  |  |
| --- | --- | --- |
| **Type** | **Range** | **Remark** |
| DATE | '1000-01-01' to '9999-12-31' | 只有日期部分，没有时间部分 |
| DATETIME | ‘1000-01-01 00:00:00’ to '9999-12-31 23:59:59' | 时间格式为YYYY-MM-DD hh:mm:ss，默认精确到秒 |
| TIMESTAMP | '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07'UTC | 默认精确到秒 |

**DATETIME和TIMESTAMP区别：**

1）时间范围不一样，TIMESTAMP要小很多 ，且最大范围为2038-01-19 03:14:07.999999，到期也不远了。

2）对于TIMESTAMP，它把客户端插入的时间从当前时区转化为UTC（世界标准时间）进行存储。查询时，将其又转化为客户端当前时区进行返回。而对于DATETIME，不做任何改变，基本上是原样输入和输出。

**java.util.Date、java.sql.Date、java.sql.Time、java.sql.Timestamp区别和总结：**

java.util.Date这个类是java.sql.Date, java.sql.Time, java.slq.Timestamp这三个类的父类。这三个类对java.util.Date类进行了包装。

## 压测工具MySQLSlap

1.自动生成sql测试

mysqlslap -a-to-generate sql -root -root #自动生成sql并执行测试性能

mysqlslap --auto-generate-sql --concurrency= 100 -uroot -proot #模拟100个客户端执行sql

mysqlslap --auto-generate-sql --concurrency= 100 --interations=3 -uroot -proot #模拟100个客户端执行sql.执行3轮

2.存储引擎测试

# （针对InnoDB）模拟100个客户端执行sql.执行3轮，innodb的处理性能

mysqlslap --auto-generate-sql --concurrency=100 --interations=3 -- engine-innodb -uroot -proot

# （针对MyISam）模拟100个客户端执行sql.执行3轮，myisam的处理性能

mysqlslap --auto-generate-sql --concurrency=100 --interations=3 --engine-innodb -uroot -proot