重要：我们先回顾书中前言的一段话：

“从某种程度来说，MySQL数据库性能优化的问题是一个并行的问题，归根结底是锁和资源争用的问题。”

我们在实训5以及其他一些练习中曾讨论锁问题排查的基本方法。问题是如果我们不理解Mysql中锁的种类及其工作原理，就无法真正理清事务之间到底在竞争些什么，等待些什么，自然也无法给出有效的优化方案。就好比为了备考你准备去图书馆占个位置(行锁)，如果连进图书馆需要什么条件，什么时候开关门(表锁)都不清楚，你也没法提出所谓更优的备考时间安排。

因此，本章主要讨论在并发环境中处于某种隔离级别的各事务之间要争夺什么样的锁，夺下这些锁后会如何阻塞其他事务，而本次实训主要介绍InnoDB的行锁和表锁。

目录

[20.1 InnoDB锁概述 1](#_Toc18110)

[20.1.1 InnoDB锁分类 1](#_Toc14191)

[20.1.2 行锁 1](#_Toc32750)

[20.1.3 表锁 6](#_Toc22572)

[20.1.4 锁模式对应的含义 9](#_Toc30192)

## 20.1 InnoDB锁概述

## 20.1.1 InnoDB锁分类

#对InnoDB锁按颗粒度进行划分，可分为表锁和行锁，其中行锁有以下几种：

● 共享锁(S)与排他锁(X)（Shared and Exclusive Lock）。

● 记录锁（Record Lock）

● 间隙锁（Gap Lock）

● 记录锁与间隙锁的组合（Next-Key Lock）

● 插入意向锁（Insert Intention Lock）

表锁有以下几种：

● 意向锁（Intention Lock）

● 自增锁（AUTO-INC Lock）

## 20.1.2 行锁

**1. 共享锁(S)与排他锁(X): S锁与X锁互相冲突，**

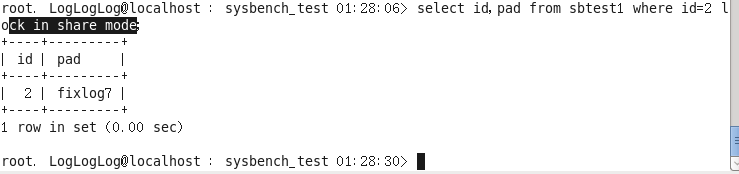
● 当读取一行记录时，为了防止别人修改，则需要添加S锁。

注：我们在讨论事务隔离时已知普通查询无法阻止数据被另一事务读写，MVCC技术保证了数据读取的一致性。这里是通过额外加锁防止别人修改。

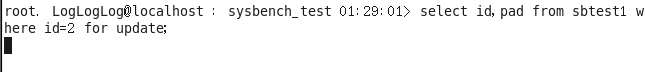
**●** 当修改一行记录时，为了防止别人同时进行修改，则需要添加X锁。

实例：添加S锁其他事务可以读但修改会被阻塞

事务1，加S锁：



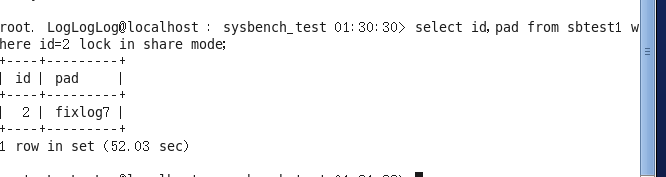
事务2，尝试加X锁



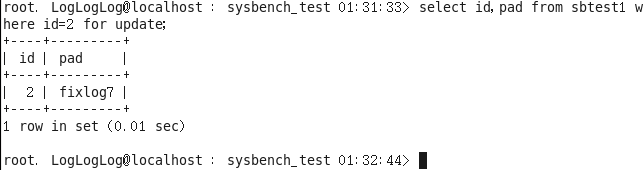
超时失败：



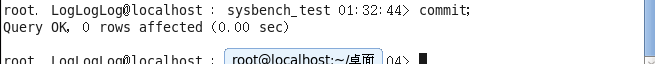
事务3，尝试加S锁



当事务1和事务3都提交后，以上两个S锁均会解除，事务2就能加X锁：

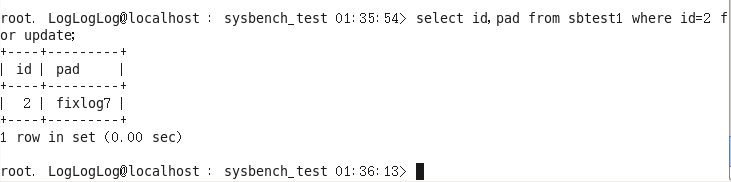


结束事务2：



实例：添加X锁，其他事务修改或者执行SELECT … FOR SHARE都会被阻塞。

事务1，加X锁：

q

事务2，尝试加S锁，被阻塞：



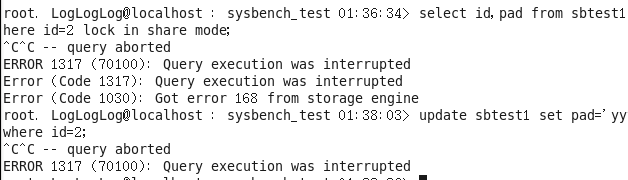
等待其超时结束：



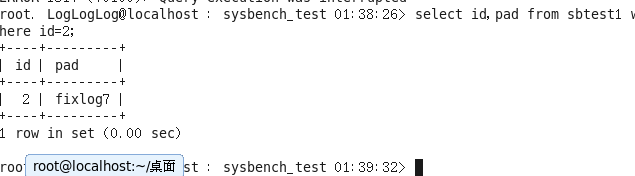
然后尝试直接更新，被阻塞：



按ctrl+c中断更新：



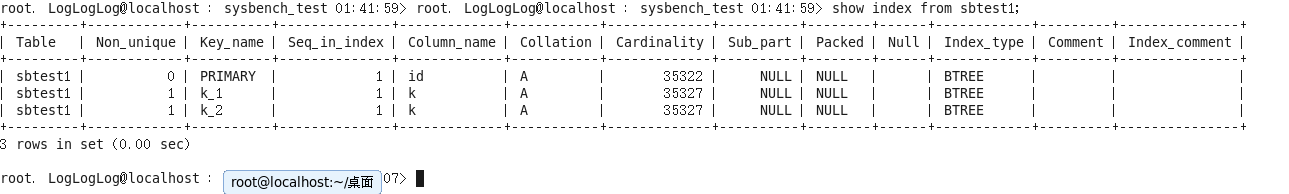
继续查询但不加锁，被允许：



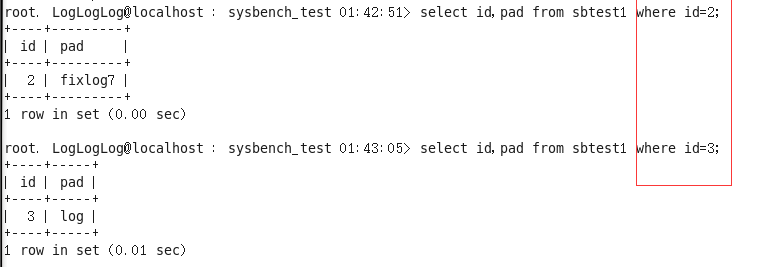
注意：以上是对行锁按照共享与排他的二分法进行讨论，下面开始讨论在实际中innodb的行锁类型。

**2．记录锁：记录锁都是添加在索引上的**

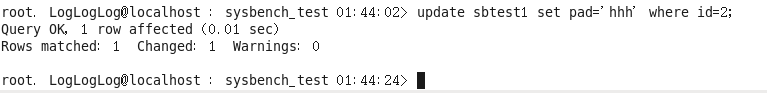
实例：首先查看表sbtest1的索引：



假设查到如下两个数据行

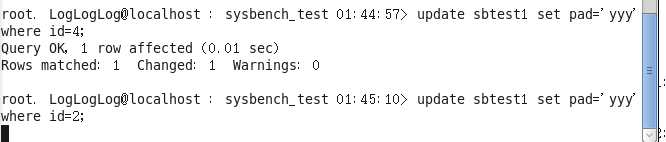


会话1更新一行数据：

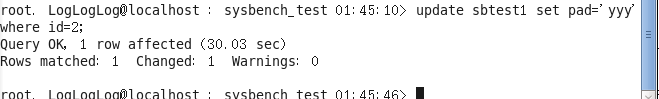


会话2能够更新另一行数据，说明其实只是锁定对应行，不受会话1影响：

但是，注意会话2不允许更新会话1锁定的那一行：



除非对方事务已提交，就可发现更新被允许



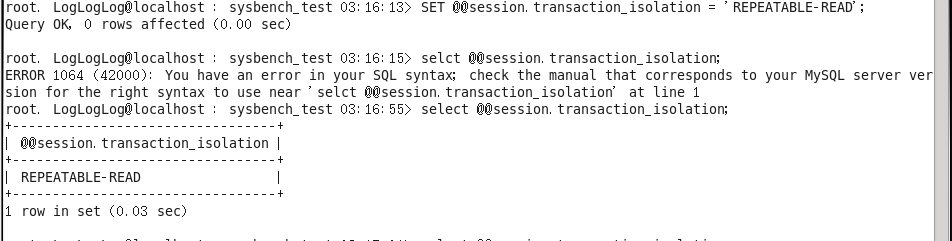
因此，记录锁能保证两个会话最大限度互不影响同时写。

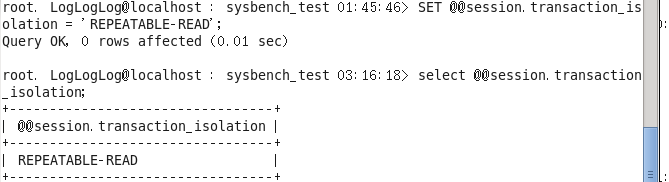
**3. 间隙锁：**间隙锁的锁定范围是索引记录之间的间隙(比如对应索引id=2,4有数据，那么id=3位于区间[2,4]的“间隙”)，间隙锁是针对事务隔离级别为可重复读或以上级别的。

实例：设置两个会话的事务隔离级别(实验室环境默认是READ-COMMITTED)：

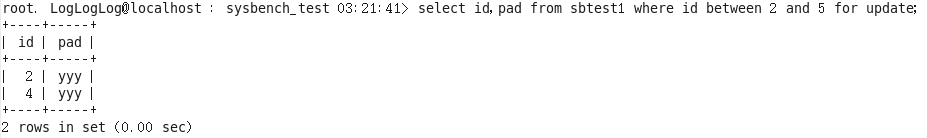
SET @@session.transaction\_isolation = 'REPEATABLE-READ';

select @@session.transaction\_isolation;



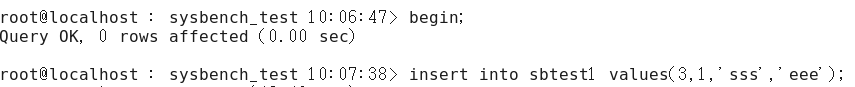


会话1，加间隙锁锁定2到5之间的间隙，当前结果如下：

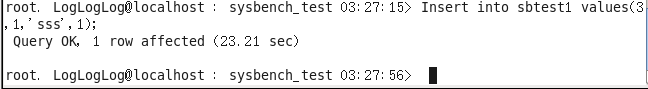


会话2，想在上述锁定的间隙中插入一条记录，可发现被阻塞：

Insert into sbtest1 values(3,1,'sss','sss');



回到会话1，提交事务意味着解锁，然后回到会话2可发现插入操作被执行：



前面指出过，间隙锁保证在REPEATABLE-READ级别下能防止幻读，就在于通过间隙锁在一个事务更新一个数据行区间时，不允许另一事务在其中插入数据行。

**4. Next-Key Lock**：记录锁与间隙锁的功能组合，实际锁定了一个记录以及它前面的间隙。

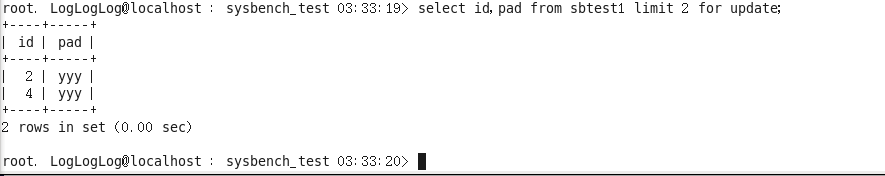
实例：设置两个会话的事务隔离级别为可重复读（过程略）。

准备数据行如下(自行设置好，不需要完全相同)：

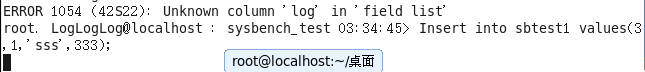
文本

描述已自动生成

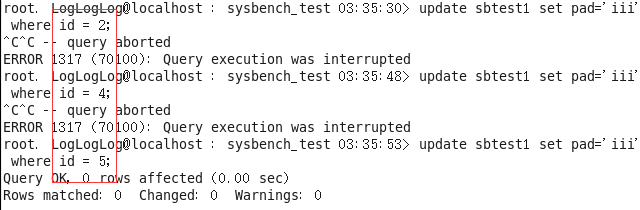
会话1，注意这里不使用where条件查询：



会话2，插入id=2之前和id为2至8之间的间隙(比如id=3)不允许：



写入到id=2，8行自然也不允许(“锁定了一个记录以及它前面的间隙”)，但id=10行未被锁定：



实际上Next-Key Lock的设置非常复杂，需要结合索引实际情况分析，这里只通过如上实例介绍概念。

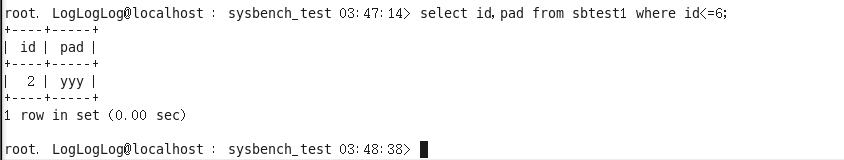
**5. 插入意向锁**(注意这里讲的是行锁-插入意向锁，要与下面讲的表锁-意向锁区分)：插入具有相同索引间隙的多个事务，如果插入的值不同，则不需要互相等待。

注：插入意向锁实质锁定的是插入行所在的间隙。所指的“意向”限定为插入数据行，insert语句表达了这种意向。因此只要彼此插入的数据行不同，但在同一个间隙，则允许并发插入。

实例：设置两个会话的事务隔离级别为可重复读（过程略）。

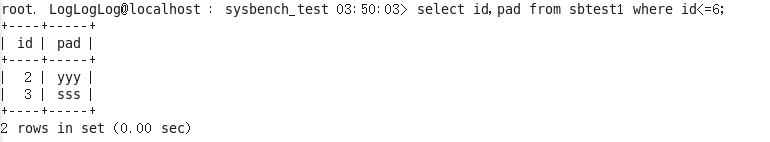
SET @@session.transaction\_isolation = 'REPEATABLE-READ';

准备数据行如下(自行设置好，id=1-6只有id=2有数据)：



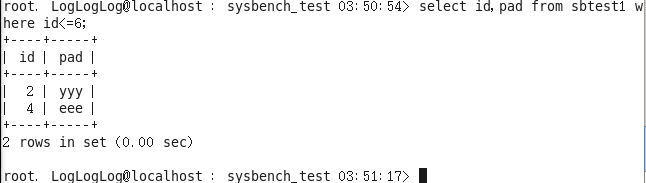
会话1插入数据：

Insert into sbtest1 values(3,3,'sss','sss');



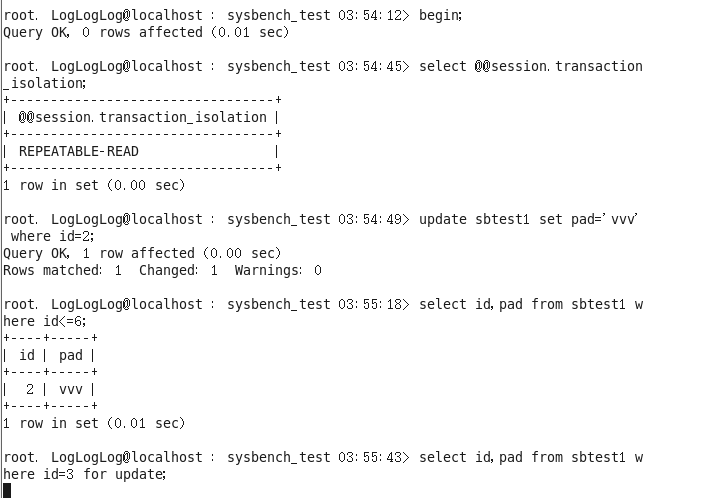
会话2也插入数据：

Insert into sbtest1 values(4,4,'sss','eee');



由此说明以上两个会话能并发插入数据行。也就是说，只要以插入数据行为目的的加锁都是兼容的。

但是，对其他目的的加锁却不一定不兼容的，会话3:



以上对id=2更新操作只加了排他的记录锁，可被允许。但是如果加入针对id=3的Next-Key 锁，不被允许。

## 20.1.3 表锁

**1. 意向锁：**

● SELECT … FOR SHARE，添加意向共享锁（IS, intention shared lock）。在获取表中**某行的共享锁(S)之前**，首先必须获取**表的IS锁**(以表明意图)。

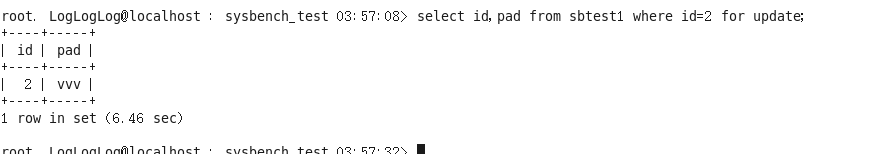
● SELECT … FOR UPDATE，添加意向排他锁（IX, intention exclusive lock）。在获取表中**某行的独占锁(X)之前**，首先必须获取**表的IX锁**(以表明意图)。

注：这意味我们前面的实例其实都已经加过表的意向锁。书中总结了为什么要对表加意向锁：

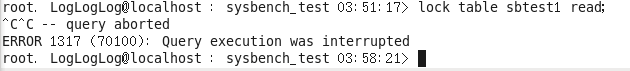
在申请表锁（执行LOCK table语句，锁定全表）时表记录不能存在锁，**在没有意向锁的情况下，就需要扫描表中每一条记录**(关键点)，查看记录上是否存在锁；但是有了意向锁之后，只要判断表上是否存在意向锁即可，如果有意向锁存在，则说明表中某行记录已被锁定或者将要被锁定，表锁的申请语句（LOCK table）会等待，意向锁设计提高了效率。

实例：

会话1，对表加意向排他锁，

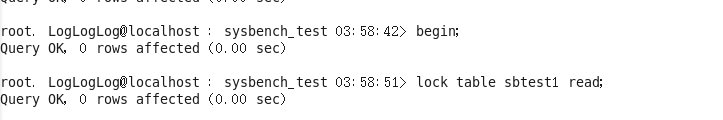


会话2尝试对表加读锁(写锁更不用说)，但发现有一行(不必知道是哪一行)被锁定，阻塞。自行ctrl+c中断：



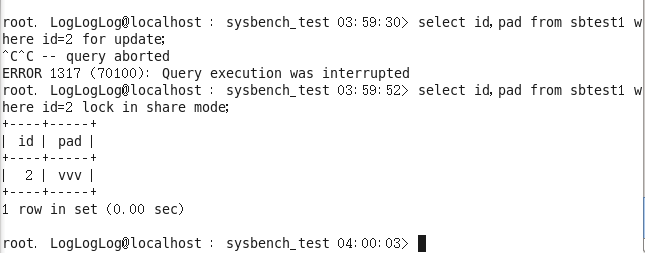
自行解锁，略。

反之，如果会话1对表加读锁：



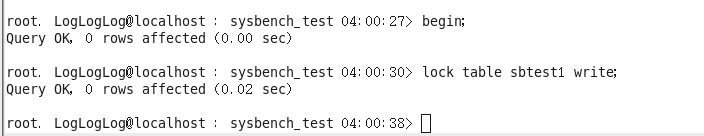
然后会话2加意向排他锁，自然会被阻塞(锁定读取的内容不允许再更改)

但是会话2加意向共享锁没问题(可以一起共享读)：

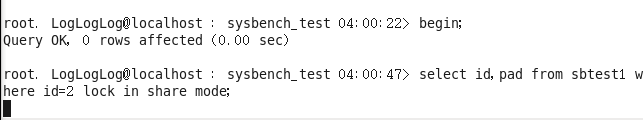


自行解锁，略。

如果会话1对表加写锁，



那么连意向共享锁都不能加：



意向锁和**行锁**之间的冲突及兼容列表

表格

描述已自动生成

**2. 自增锁：**

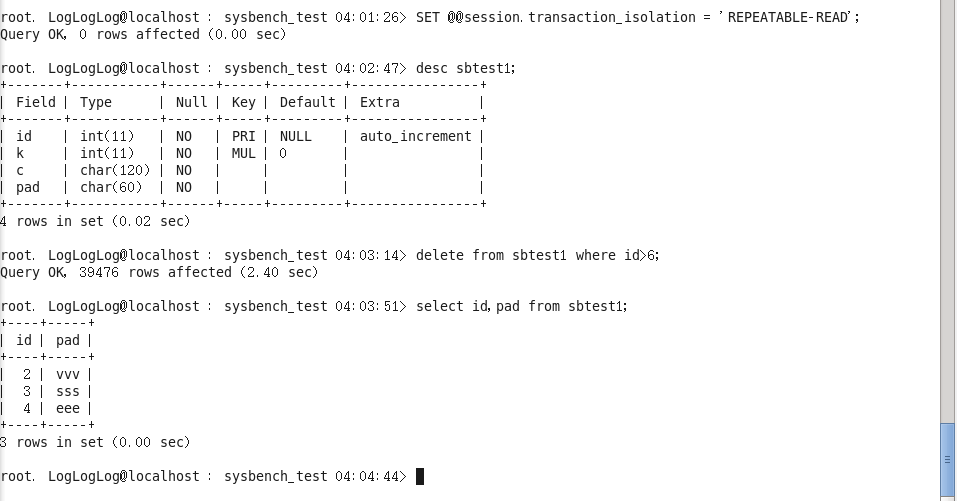
插入到具有AUTO\_INCREMENT字段的表中的事务所采用的特殊表级锁。如果一个事务正在向表中插入值，则其他任何事务都必须等待插入语句执行完成。

注：当插入语句执行完就会解锁，不用等到事务提交。目的是为了保证插入语句按顺序执行，从而保证主键值连续自增。

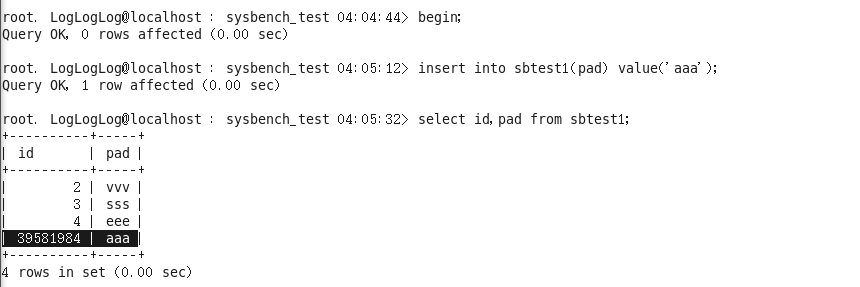
实例：设置两个会话的事务隔离级别为可重复读（过程略）

SET @@session.transaction\_isolation = 'REPEATABLE-READ';

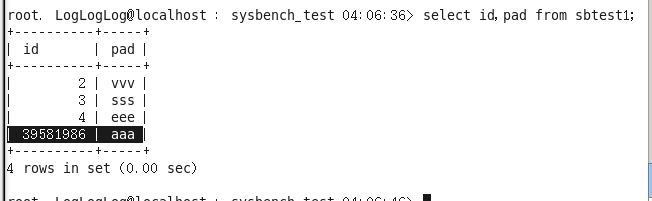
。把多余的数据行从表sbtest1中删除



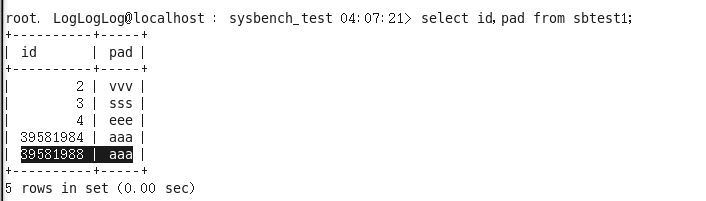
会话1，按自动编号插入新一行：



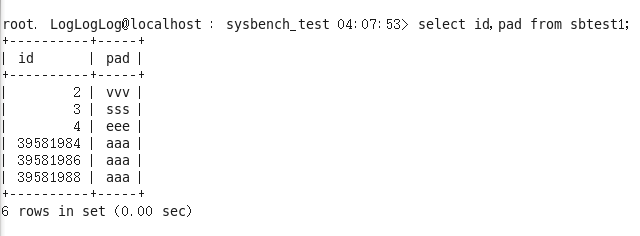
会话2按自动编号插入另一行，



注意会话2没有被阻塞，由此可见自增锁作为一个表锁并没有等到事务结束才解锁，而是申请到id值就解锁。否则会话2不能插入新数据。

会话1同样能够再插入新的一行：

会话1和会话2都提交后，就能够查到所有新增的数据行：



此外，可重复读的事务隔离级别保证不会出现幻读。

## 20.1.4 锁模式对应的含义

当我们使用show engine innodb语句查看锁信息时，经常会看到LOCK\_MODE字段，也就是锁模式，只有知道各种模式都分别代表什么意思才能更好地去分析锁等待和死锁问题。这是一个**非常重要的知识点**。

● IX：代表意向排他锁。

● X：代表Next-Key Lock锁定记录本身和记录之前的间隙（X）。

● S：代表Next-Key Lock锁定记录本身和记录之前的间隙（S）。

● X, REC\_NOT\_GAP：代表只锁定记录本身（X）。

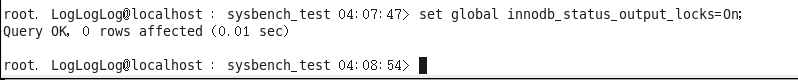
● S, REC\_NOT\_GAP：代表只锁定记录本身（S）。

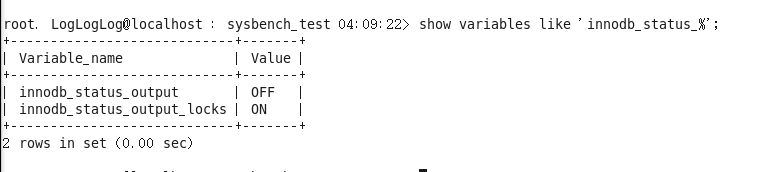
● X, GAP：代表间隙锁，不锁定记录本身（X）。

● S, GAP：代表间隙锁，不锁定记录本身（S）。

● X, GAP, INSERT\_INTENTION：代表插入意向锁。

实例：在没有经过一定训练之前，以上这些标志其实非常费解，但是对于后面实验却又比较重要。因此我们需要通过具体操作熟悉以上列表以及理解前面介绍的各种行锁和表锁。首先打开如下开关以获取更多的锁信息：

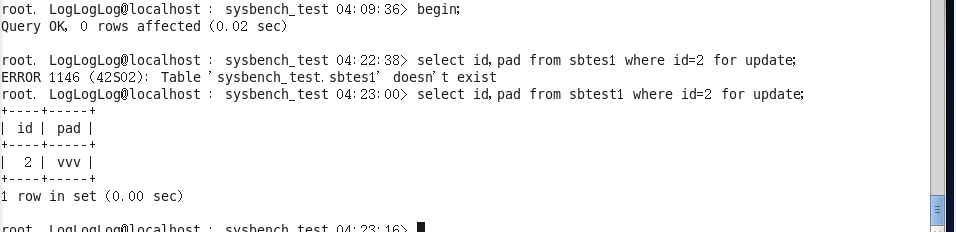




由于操作时会输出大量的状态信息，所以设置(菜单“编辑”)字符终端不限定回滚行数：

此外，这里innodb\_status\_output开关这里也开了，那么系统会每15秒输出innodb记录到error.log，这个可以考虑但由于输出内容太多有时未必有用且增加系统负荷。

作为尝试，在一个会话开启如下事务：



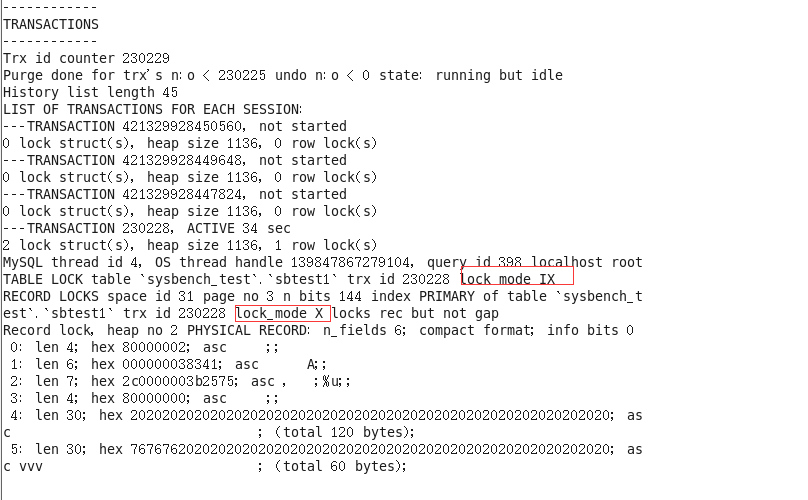
然后在另一个会话查看innodb状态：

文本, 信件

描述已自动生成

然后在执行过程中，查看innodb引擎的状态，对照上面锁模式列表，在“TRANSACTION”段就能发现对应事务持有该锁的信息。

root@localhost : sysbench\_test 10:12:19> show engine innodb status;



其中lock mode字段能够看到首先对表sbtest1加了意向排他锁(IX)，然后再对应id=2这一行加了行锁X，但不是间隙锁(X, REC\_NOT\_GAP)。总之：先锁表，再锁行。

为了能观察到更多锁类型信息，下面执行sysbench\_test压测：

文本

描述已自动生成

在测试过程中调用上述show语句，获得返回结果示例如下。可以看到，为更新表sbtest4中的某行，该事务当前有两个锁结构，先加意向锁(IX)锁表sbtest4，然后加间隙锁锁定该行：

