说明：无论是操作系统还是数据库系统，并发与资源竞争始终是影响系统性能的核心要素，而死锁却是影响系统性能的一个特殊又普遍的现象。在前面的压测实验，我们见到其实只要并发线程竞争足够激烈，死锁的出现几乎是不可避免的。系统也许会自动检测死锁并通过回滚某一事务解决问题。但细致地回顾和分析死锁记录则是系统设计者的责任。

这次实训共有三个案例(第34-36章)，分析了数据库死锁问题的现象和成因。在此之前不妨回顾一下死锁发生的四个必要条件：

1. 互斥条件
2. 部分分配条件
3. 不剥夺条件
4. 环路条件

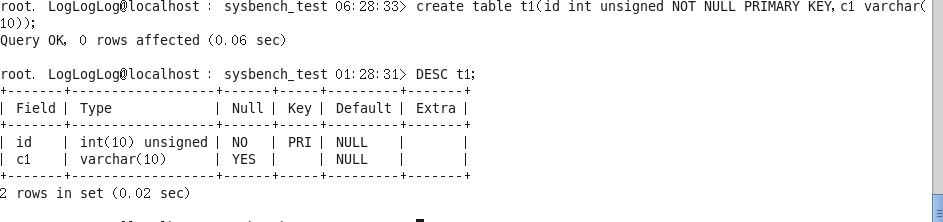
练习时建议结合这四个必要条件，以死锁日志分析为线索，充分理解死锁模拟情境中的加锁过程和死锁发生的具体原因。

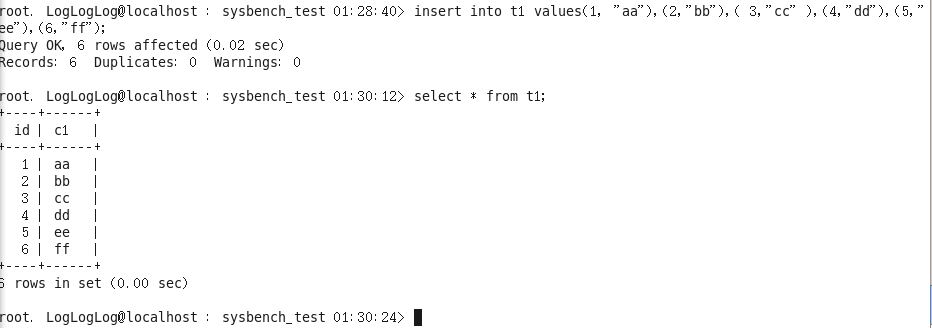
## 第34章 并发删除数据造成死锁

## 34.1 问题现象

本章的案例是客户在夜间批量执行数据处理时发生了死锁，这是由不同的会话并发删除数据引起的。这个问题的原因比较简单，但是想通过这个案例让大家熟悉**如何排查死锁问题、如何阅读死锁日志**才是目的。

准备工作：





insert into t1 values(1, "aa"),(2,"bb"),( 3,"cc" ),(4,"dd"),(5,"ee"),(6,"ff");

模拟死锁发生的环境：

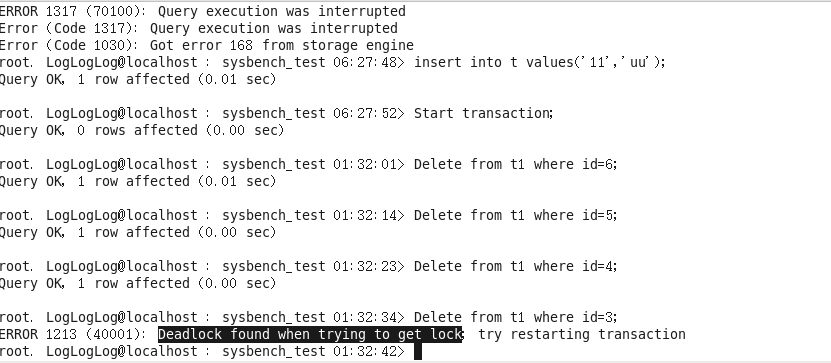
操作步骤：

Start transaction;

Delete from t1 where id=1;

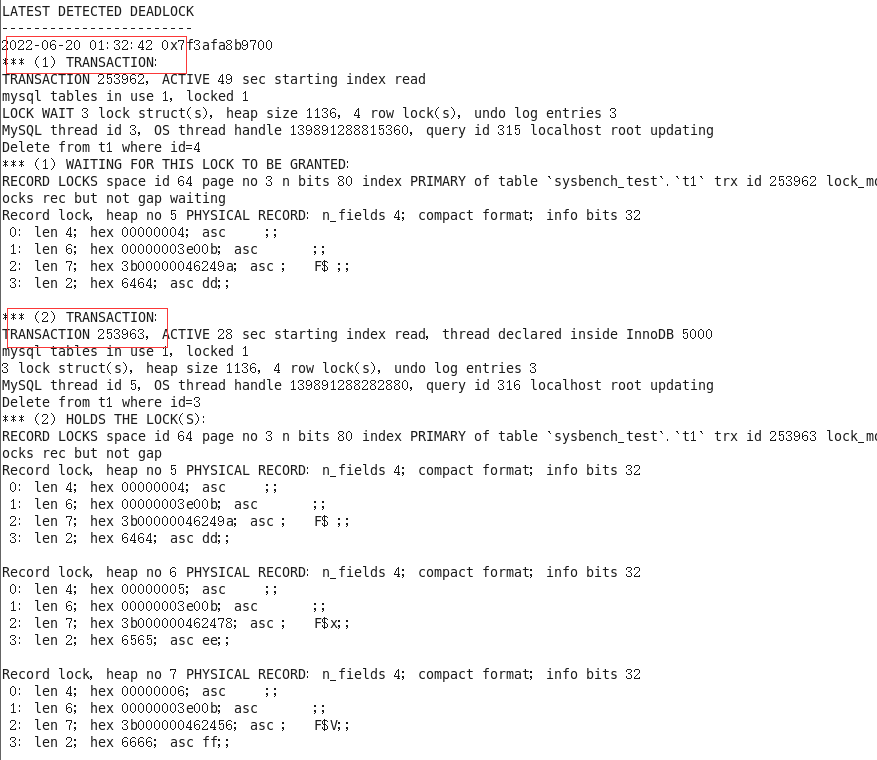
表格

描述已自动生成



注：留意死锁发生的时间。

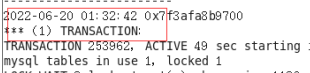
通过show engine innodb status查询死锁日志，历史上发生过死锁会一直留着直到最新死锁出现，所以要确认检测时间：



## 34.2 如何阅读死锁日志

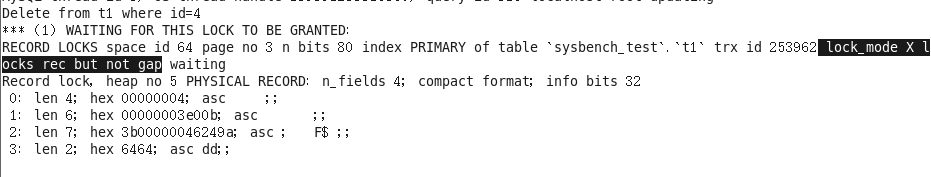
MySQL死锁日志看起来并不是很直观，需要我们一步一步地**耐心分析**。

1. 两个事务的事务ID，简单起见约定只用编号1、2表示

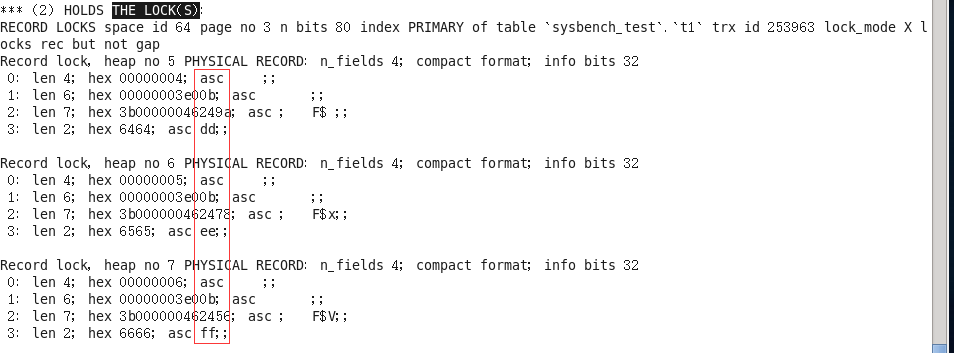




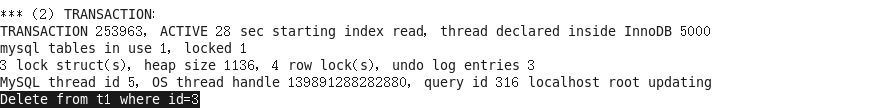
1. 事务1在执行DELETE语句时发生了锁等待。事务1在执行DELETE语句时，在申请id=4(对应dd)这条记录上的X锁时发生锁等待：lock\_mode X locks rec but not gap waiting。



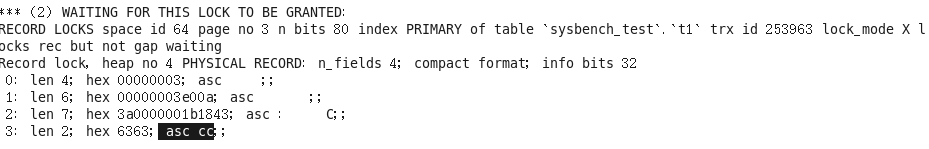
1. 事务2持有锁的信息：持有id=4,5,6(对应dd、ee、ff)记录上的X锁。注意事务2持有的**id=4(对应dd)的记录锁，正好就是事务1等待的锁。**



1. 事务2执行DELETE语句，在申请id=3记录上的X锁时发生了锁等待，执行的语句是“DELETE FROM t1 WHERE id = 3”

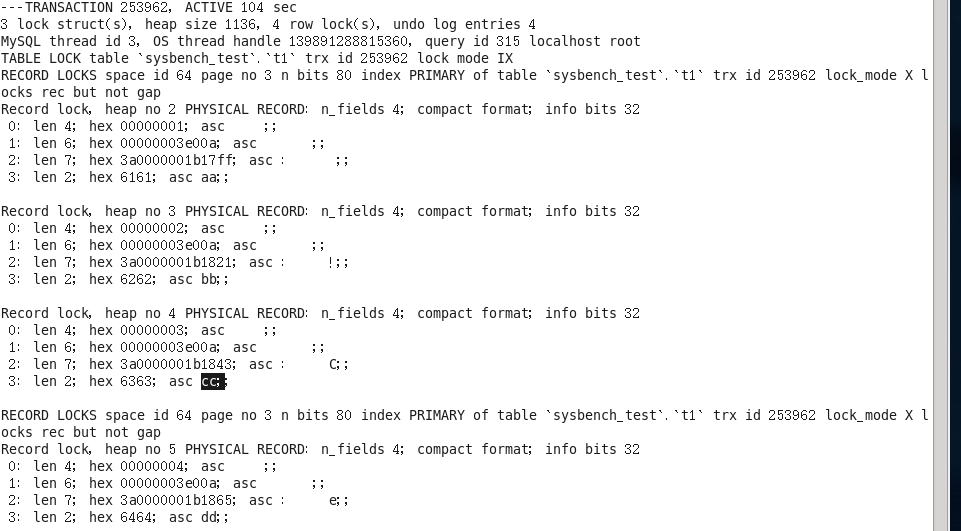


同样发生了锁等待：



由此可以得知事务1在id=3记录上持有X锁，但是在死锁日志中并没有显示出事务1持有锁的信息。

注：其实我们现在已经可以推知事务2等待id=3(对应cc)的记录锁，正好是事务1持有的锁，因为这个是两个事务的死锁记录。而事务1持有锁的信息只是没有在“LATEST DETECTED DEADLOCK”段中出现，但在“TRANSACTION”段可查询：



两个事务加锁的顺序应该是：

1. 事务1持有id=3记录上的X锁。
2. 事务2持有id=4记录上的X锁。
3. 事务1在申请id=4记录上的X锁时发生了锁等待，执行的语句是“DELETE FROM t1 WHERE id = 4”。
4. 事务2在申请id=3记录上的X锁时触发了死锁，因为此时双方都在申请对方持有的锁不能继续执行。
5. 事务2被回滚



1. 事务2**持有**id = 4,5,6记录上的X锁是由哪条语句引起的，无法直观地从死锁日志中看出。可以通过打开普通日志、binlog或者业务代码来查看整个事务的逻辑。

## 34.4 本章小结

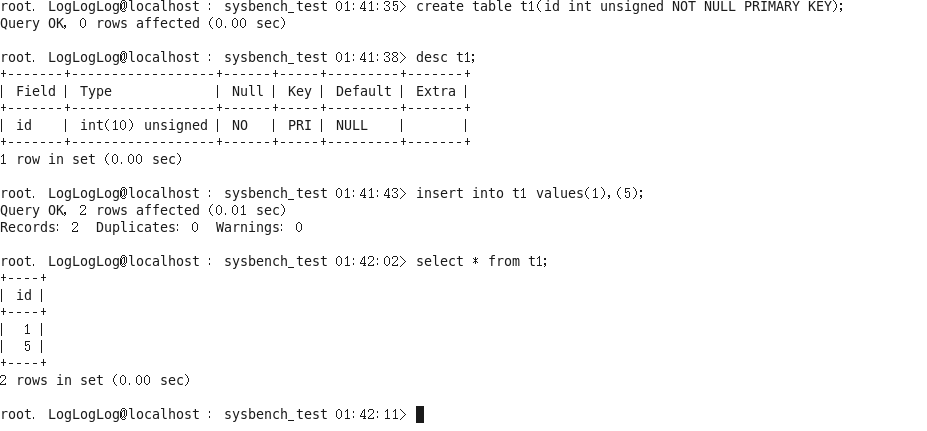
这个案例是两个会话在同时删除数据时，**没有控制好删除的顺序造成了死锁**。这就需要我们在做应用开发时，对数据库操作一定要**注意操作数据的前后关系、是否有数据依赖，以及会话之间是否会操作相同的数据等**。通过这个案例介绍，相信大家都会觉得死锁日志也不是很难阅读的，只要按着本章所讲的方法，将日志拆分开阅读，肯定就能清晰地分析出死锁原因，希望大家在工作中遇到死锁时能够通过阅读死锁日志来分析问题。

## 第35章 删除不存在的数据造成死锁

## 35.1 问题现象

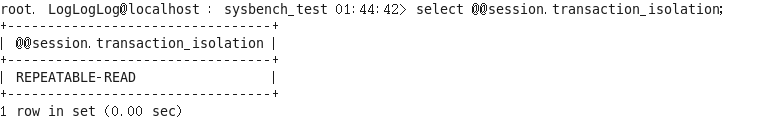
在MySQL中RR（REPEATABLE-READ）隔离级别下的GAP锁确实是一个很容易“踩坑”的地方，**如果不了解GAP锁的机制，就很容易造成锁等待或者死锁**，本章我们就来分析一个删除不存在的数据造成死锁的案例，通过这个案例让大家更深入地了解在RR隔离级别下为何会容易造成锁等待或者死锁。

准备工作，注意这个表虽然简单但有设置主键：



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

要检查并确保处于RR隔离级别，否则后面的死锁不会发生：



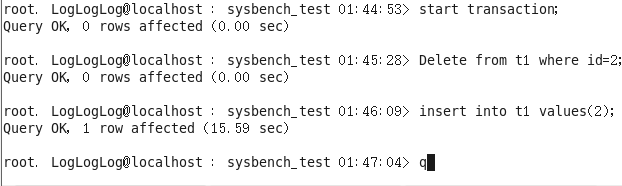
模拟死锁发生的环境，注意id=2,4的数据行不存在：

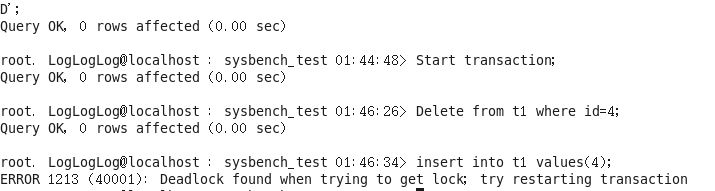
Start transaction;

Delete from t1 where id=4;

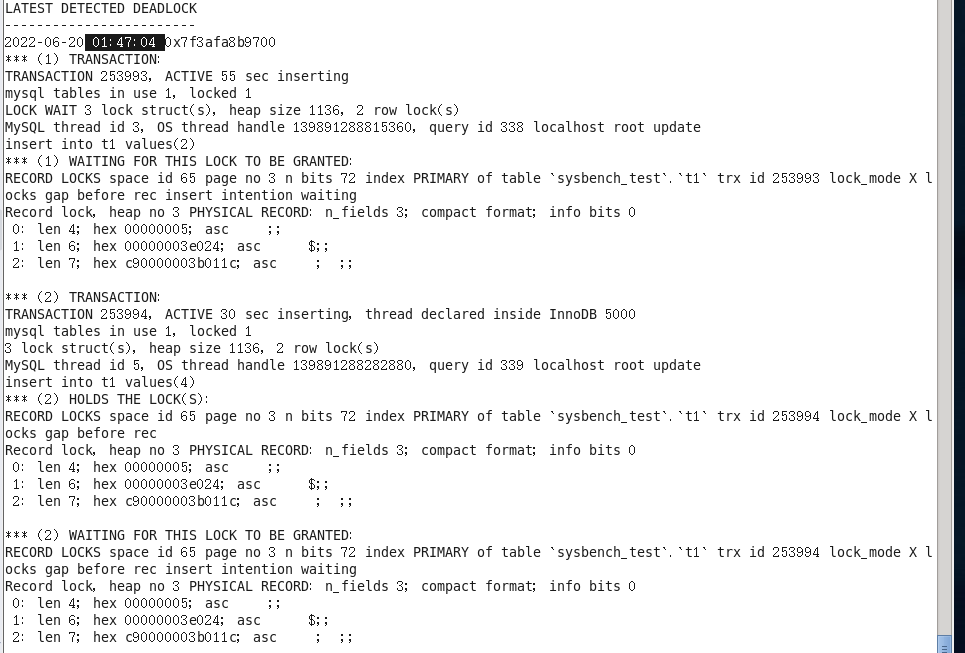
表格

描述已自动生成





大家可能很难理解删除不存在的数据是如何造成死锁的，下面我们一起来分析，先看一下复现后的死锁日志(查到的结果要核对是不是模拟时的时间)：



## 35.2 问题分析

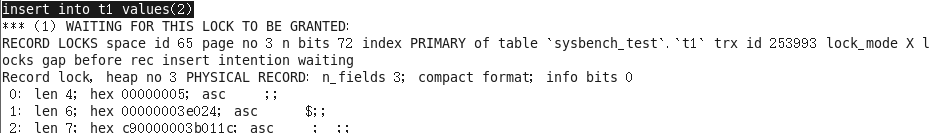
类似于第34章给出的死锁日志阅读方法：

1. 两个事务(1、2)的事务ID

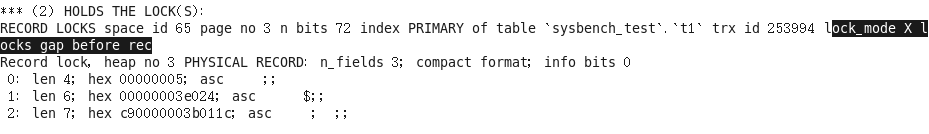




1. 事务1在执行INSERT语句申请插入意向锁时发生了锁等待，插入的语句是“INSERT INTO t1VALUES（2）”:

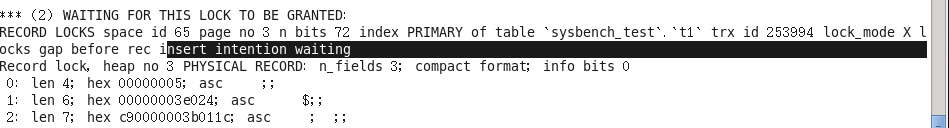


1. 事务2在主键值为5的这条记录上持有GAP锁，**GAP锁的范围应是（x,5），x代表未知。**

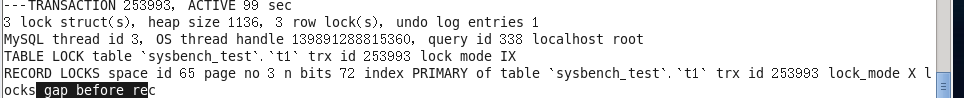


这里已经构成了事务1被事务2阻塞的事实：事务1等待的插入意向锁(“INSERT INTO t1VALUES（2）”)，与事务2持有的GAP锁冲突。

1. 同时事务2在执行“INSERT INTO t1 VALUES（4）”语句申请插入意向锁时发生了锁等待，且可以看到锁等待记录的主键值是5：



查“TRANSACTION”段可发现，事务1同时也**在主键值为5的这条记录上持有相同的GAP锁（可对比第(3)点）：**

两个事务都在互相申请对方的锁而不能释放造成了死锁：

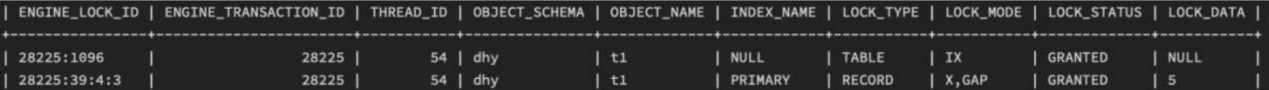
问题的关键在于两个事务可以同时持有间隙锁(x,5)，**它们皆因施加插入意向锁而等待对方释放持有的间隙锁而阻塞。**

## 35.3 问题扩展

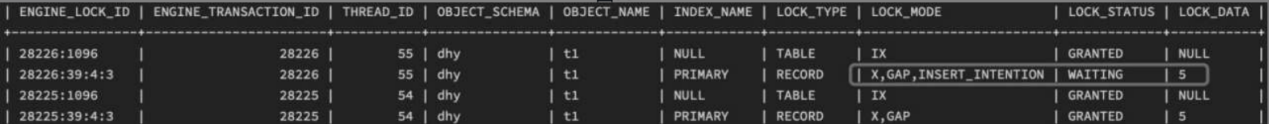
注意：这一节注意需要阅读书中内容然后再操作。因为书中不少句子被拆开分解来逐句解释了。但脱离书本读起来会比较费解。

“我们通过死锁日志分析出死锁的原因，但还需要继续分析为何删除不存在的数据也会造成死锁。前面我们分析**GAP锁的范围是（x,5），这个范围是如何产生的**？”

注：我认为书中相关操作所显示结果并未真正回答以上问题，最起码并不能从图35-1



和图35-2



获知问题中x应为1。不过，书中指出：“锁定的范围应是（1,5），这时插入的值在这个范围内都会被阻塞，原因是防止在这个范围内再有新的数据插入造成幻读。”这个理由是合理的。

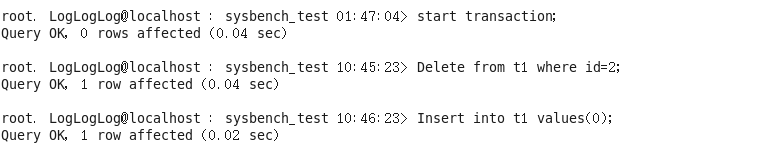
Start transaction;

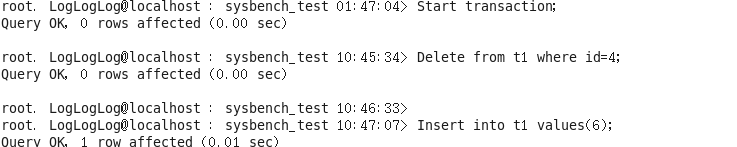
Delete from t1 where id=2;

Insert into t1 values(0);

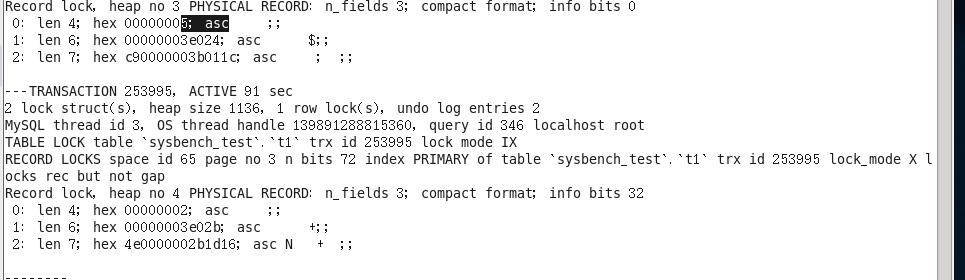
“但是当插入的记录不在（1,5）范围内时，则不会发生锁等待。”

注：书里这么说，我们再试一次，的确如此：



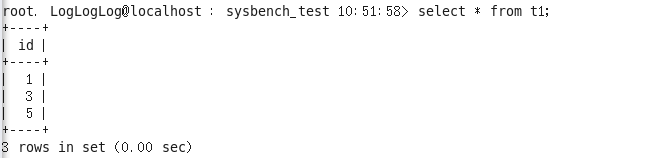


再查一下锁信息，还是两个事务同时持有相同的间隙锁：

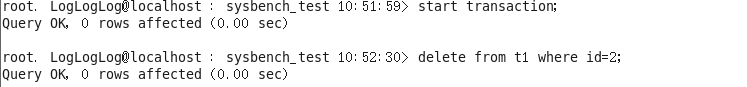


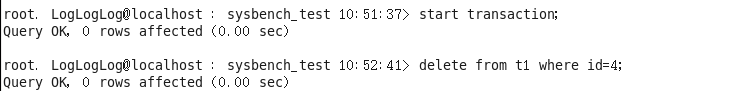
“可以看出，当删除的记录不存在时，GAP锁的范围会比较大，很容易造成锁等待。如果表中在id=1与id=5之间还存在值，则会将锁定的范围减小。”

举例解释上面的话。现在表t1如下：

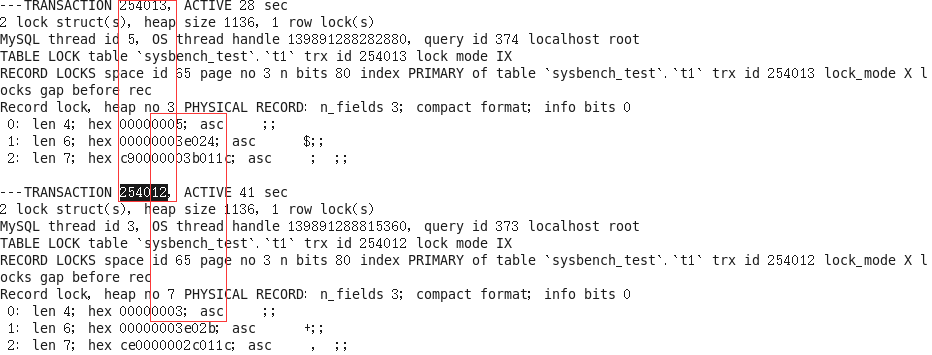


然后开启两个事务分别删除id=2和id=4两行：

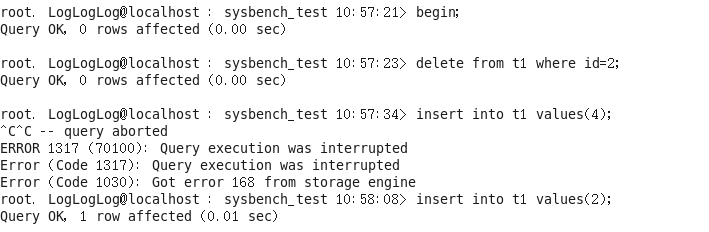
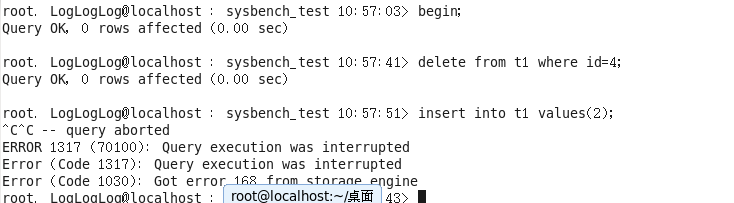




然后查看锁信息，对比前面的结果可知锁定范围的确减小了：

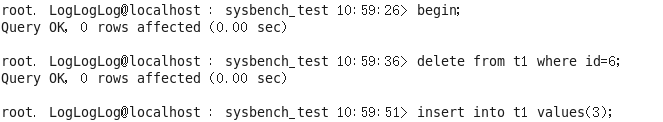


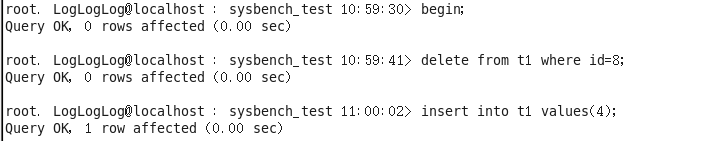
按照前面的分析思路，事务1(ID号较小的事务)应该锁定了间隙(1,3)，而事务2锁定了间隙(3,5)。这意味着，事务1应该可以插入间隙(1,3)，因为它持有间隙锁(1,3)，但不能插入间隙(3,5)，因为它被事务2锁定。换言之，事务1可以插入id=2的数据行，但不能插入id=4的数据行。对称地，事务2可以插入id=4的数据行，但不能插入id=2的数据行。下面操作验证这个猜测：



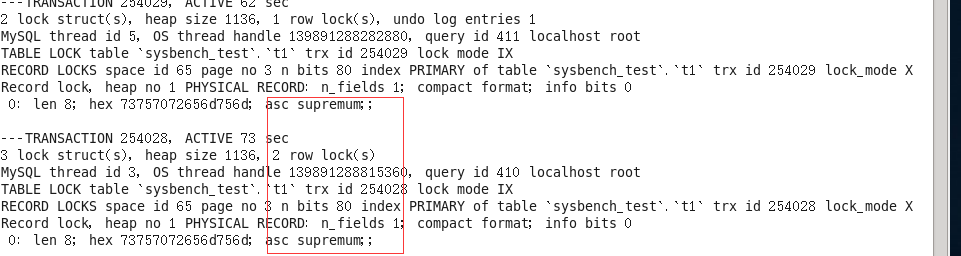
“但是如果删除的记录比id=5这条记录大，则锁定的范围将是（5, +∞）。”

举例解释上面的话。建立两个事务操作如下：





由此可见，避开这些同时锁定的范围就不会发生死锁。而查看锁信息，锁定的范围的确是（5, +∞）：



插一句话：我们有疑问的地方，都可以通过操作和假设去验证。

## 35.5 本章小结

通过这个案例介绍，相信你对不建议使用RR隔离级别这个规范有了一定的了解。**在RR隔离级别下我们要深刻理解GAP锁的范围**，有时候发生死锁就是由于我们一时的疏忽。此案例中对于删除不存在的数据这种情况，如果我们对GAP锁没有深刻的认识，则在开发应用程序时很容易发生死锁这种问题，大家可以结合第20章中介绍的各种加锁实验，逐渐对GAP锁加深印象。

## 第36章 插入意向锁死锁

本章的案例是客户在生产系统中遇到一个死锁现象，但是不能理解在这种情况下为何会发生死锁。**通常死锁发生的情况：**

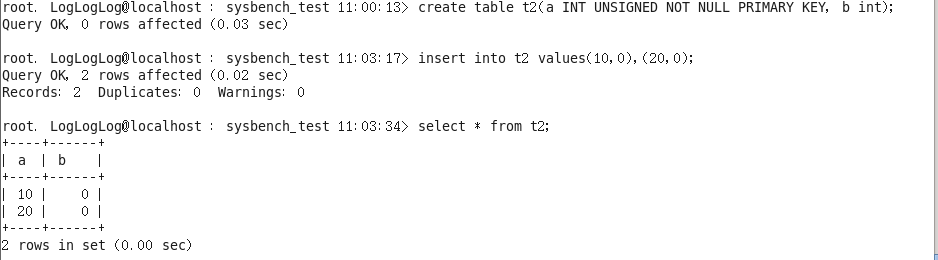
图形用户界面, 文本, 应用程序, 聊天或短信

描述已自动生成

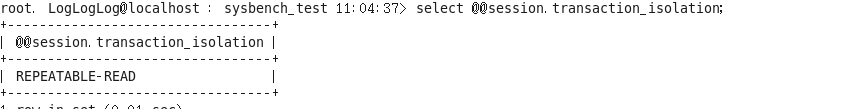
当两个事务都试图获取另一个事务已经拥有的锁时，就会发生死锁。

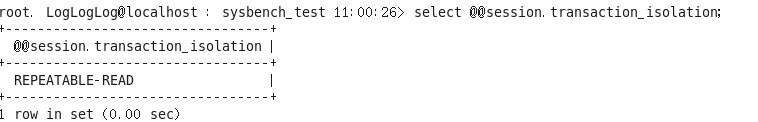
这个案例中死锁的产生和上述所讲的情况有些不一样：

准备工作：



确认两个事务处于RR隔离级别：





模拟死锁发生的环境，这里用的是表t2:

表格

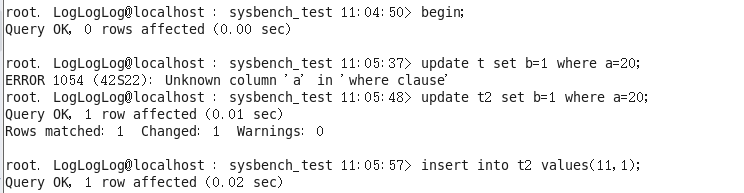
描述已自动生成

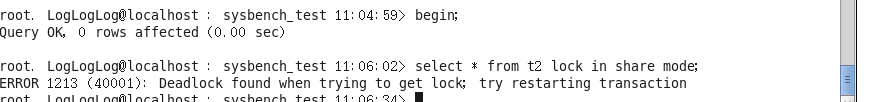
首先令事务2试图加S锁，被事务1阻塞：

图片包含 图形用户界面

描述已自动生成

然后事务1插入一行，同时事务2报死锁被回滚：





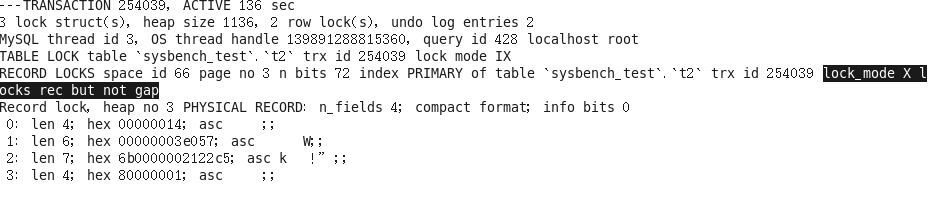
## 36.2 问题分析

查看死锁日志分析问题(书中使用data\_locks表，mysql8.0可用)。我们需要对死锁环境再模拟一次(之前的操作注意回滚，还原到表原来的状态)。不过，mysql5.7不能够查询到“LOCK IN SHARE MODE”的锁信息(详细讨论可参见：

http://blog.itpub.net/7728585/viewspace-2638875/)。所以我们这里用“for update”

● 两个事务之间是如何加锁的？

当Session 1执行完UPDATE语句后，加锁情况：



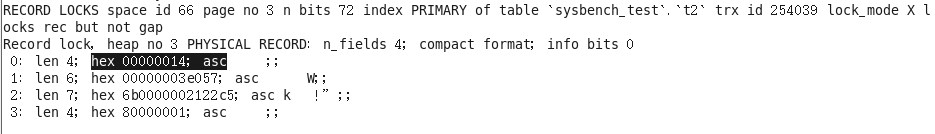
在这里可以看到对表添加了IX锁，同时对记录20(“hex 00000014”)添加了X锁。

Session 2执行SELECT\* FROM t2 for update语句后的加锁情况。Session 2执行后总共会申请三个锁：

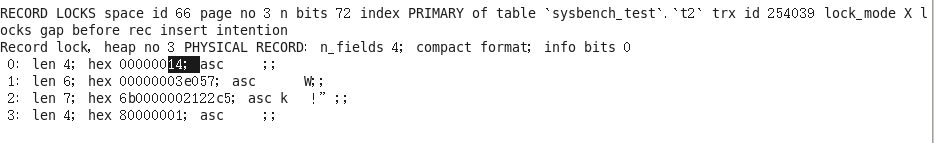
①对表添加IX锁。



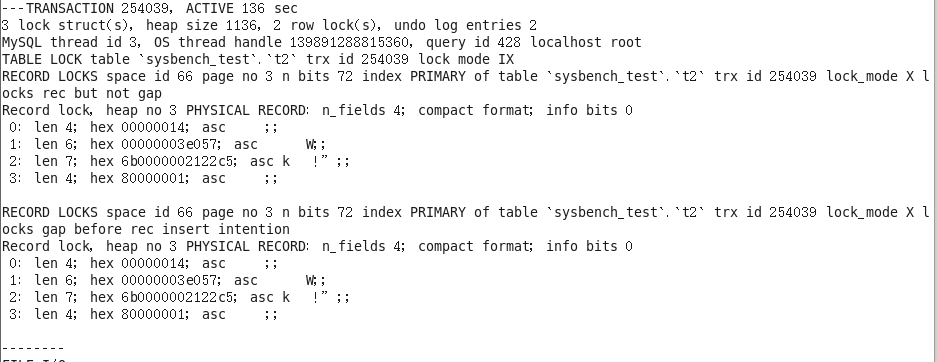
②对a=10(hex 00000014)这条记录添加Next-Key Lock（X）锁。



③对a=20这条记录添加Next-Key Lock（X）锁。由于在申请a=20记录上的X锁时发生了阻塞，我们看不到“supremum pseudo-record”，



如果(用另外的事务)**单独执行**这条语句，则加锁情况：



需要注意Session 2执行完上锁的类型应是Next-Key Lock（S）锁，这条语句锁定的范围是**（-∞,10],（10,20],（20, +∞）**。

● 为何产生了死锁？

主要原因：Next-Key Lock与插入意向锁的不兼容

表格

描述已自动生成

结合死锁日志分析(这里又再模拟了一次)：

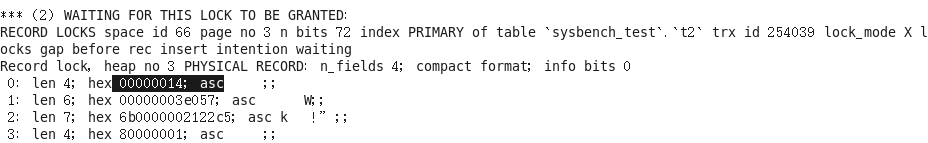
Session 2(事务ID更大者)等待Session 1中X锁的释放：

注：事务2等待的锁：

图片包含 文本

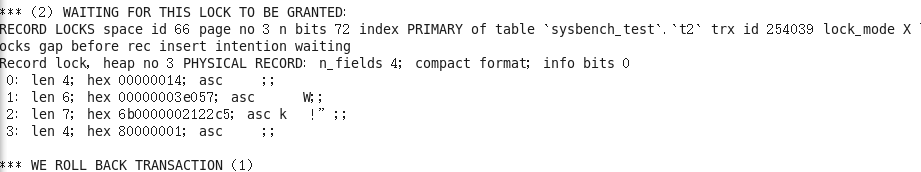
描述已自动生成

事务1持有的锁：



随后的插入意向锁与Session 2中的Next-Key Lock（S）锁不兼容：

事务1等待的锁：



跟前面一样，事务2持有的锁死锁日志里不会显示，这意味着我们需要像前面分析“两个事务之间是如何加锁的？”那样分析，便能知道到底事务2加了些什么锁导致事务1阻塞。

● 发生死锁后为什么是Session 2中的事务被回滚了？

当发生死锁时InnoDB选择回滚占用资源最少的事务，通过innodb\_trx表中的trx\_weight来判断所占用资源的多少。

表格

描述已自动生成

## 36.3 本章小结

重点是分析插入意向锁与Next-Key Lock锁是否兼容。同时我们也知道了发生回滚时InnoDB是如何选择的。

Goodbye teacher.Chen