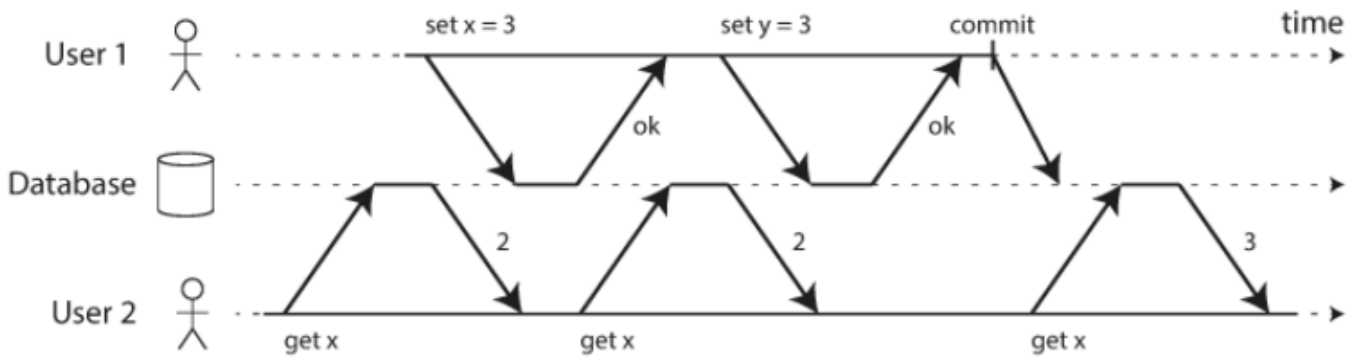
事务的弱隔离级别之Read Committed



上一篇文章介绍了事务的ACID属性。

程序员阿sir：事务之ACID2 赞同 · 0 评论文章

其中提到的的**隔离性**要求并发执行的事务之间彼此相互隔离，不应该相互影响。

要知道，如果两个事务访问的不是相同的一些数据，那他们怎么并行运行都不会相互影响。

并发问题（或者称为竞争条件）只会出现在以下两种情况中：

- 当一个事务读一个数据，而这个数据刚好在被另一个事务修改。
- 两个事务尝试同时修改同一个数据。

然而，并发问题的bug是很难复现并debug的，出现的概率很低。

因此，数据库尝试通过**隔离性**来简化问题

> *因为隔离性要保证多个事务运行之间不相互影响，让我们可以假设程序是被线性执行的。*

但是实际上隔离是非常复杂的，尤其是**可序列化隔离 (Serializable Isolation) 。**

可序列化隔离的意思是：数据库保证事务之间运行结果与他们**串行执行**的结果完全一样。

然而，这种隔离级别会**大大降低性能**。

> *比如一种实现方式就是通过**加锁**的方法，把读写数据变成串行。 但是服务器在**同一时刻**只能允许**一个**请求访问这条数据，这就导致服务器性能大打折扣。*

所以，很多常见的数据库都实现了弱一点的隔离级别（简称**弱隔离级别**），并将其作为默认的数据库隔离级别设置。

也就是说这些**弱隔离级别**都能解决一些并发问题，但是不保证所有的并发问题都能解决，也就是看起来并不是真的和串行执行完全一致。

虽然他们可能会导致bug，但是还是有很多数据库都为了性能而支持这些弱隔离级别，并且被用户广泛使用。

> 注意！这些弱隔离级别真的会由于并发问题导致数据出错。因此**银行系统**这种一定要使用完全隔离，完全满足 ACID 的数据库，因为如果出错都是真金白银的损失。

下面我们看一些**弱隔离级别**，也叫**不可序列化隔离**。以及可能存在的问题。

1. 读已提交 (Read Committed)

1.1. 概念

Read Commited是最基础的事务隔离级别，它保证了以下两点：

> 读数据时只能读到已经提交了的数据 （没有脏读）
> 写数据时只能覆盖已经提交了的数据 （没有脏写）

下面具体介绍这两点：

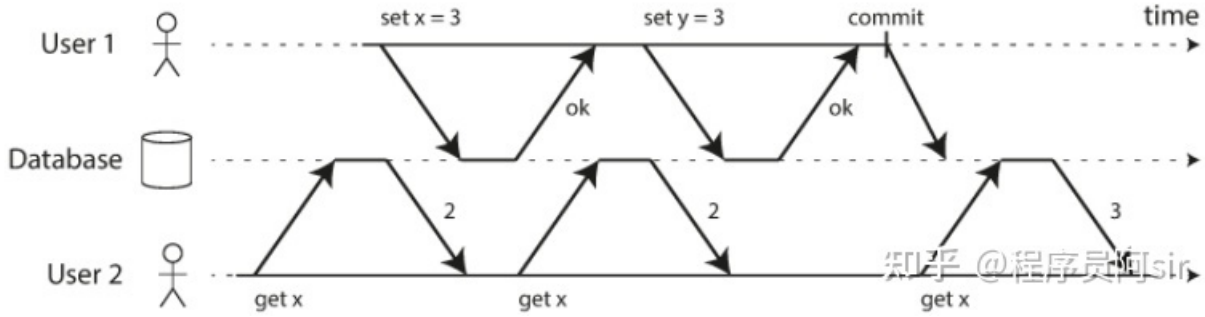
（1）没有脏读 (No Dirty Reads)

> **脏读**：从一个事务中可以读到另一个事务中**未提交**或**回滚**了的数据。

Read Committed 隔离级别保证**没有脏读**。

也就是说一个事务中的多次写操作只会在提交 (Commit) 的一刻才会被外界看到。

比如下面的例子：



> 用户 2 先读到x的值是2。
> 在用户 2 下次读之前，用户 1 把x的值改成了3。
> 但是用户 2 第二次读 x值还是2。
> *这是因为用户 1 此时还没有 commit 这些写操作。*
> 当用户 1 commit 之后，用户 2 才能读到 x 最新的值，也就是3。
> 所以这个例子就是 Read Committed 可以保证的。

那么"没有脏读"到底有什么用呢？

看上去，上面的例子中，用户 1 读到 x 的值是 2 或 3 没有影响。

然而，脏读往往会产生以下两个弊端：

- 【弊端一】：如果一个事务需要更新多个表，脏读可能导致其他事务只能读到其中的部分更新。

  比如对邮件系统。
  一封邮件到达时，邮件系统会依次进行以下操作：
  操作1：把接收方的**未读邮件数**加 1
  操作2：把邮件信息加到未读邮箱中
  假如恰好当接收者打开邮箱查看时，邮件到达的事务刚好**执行了一半**
  （即刚把未读邮件个数加 1，还未执行操作2）
  那用户就看到邮箱未读邮件数是 1，但是却没看到未读邮件。
  这就让用户十分困惑。

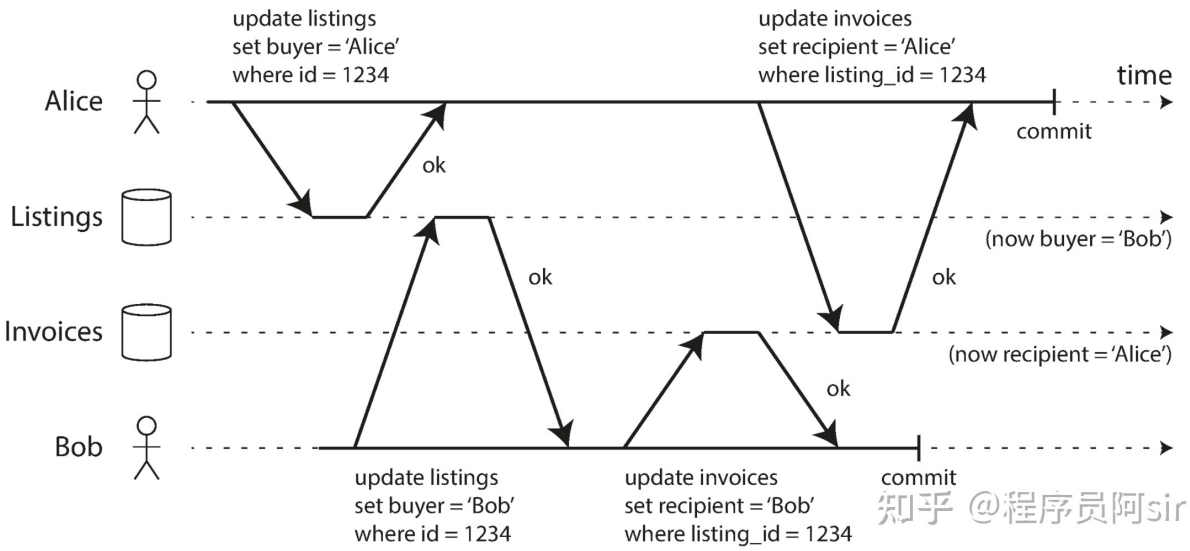
- 【弊端二】：如果一个事务回滚，那所有事务中的写操作都会回滚。假设在回滚之前，另一个事务已经脏读到了一部分数据，这就意味着该事务读到了从未 commit 的值，这种情况非常不合常理。

（2）没有脏写 (No Dirty Writes)

脏写：一个事务写到数据库里的值**覆盖**了另一个事务中**还未提交**的值。

为了避免脏写，隔离级别需要做到的是避免一些**并发问题**。

比如下图中的例子：



图中表示两个人要去买汽车，但是是**同时下单**的。
下单之后的事务依次进行以下两项操作：
操作一：更新这辆车的买主是谁。
操作二：更新这辆车的账单给谁。

如果存在**脏写**
系统会记录的这辆车卖给了 Bob。
（因为 Bob 的更新购买者列表操作是后执行的，把 Alice 的未提交记录覆盖了。）
但是账单却发给了 Alice。
（因为 Alice 的账单是后更新的。）
这明显是非常不合理的！

1.2. 如何实现 Read Committed

目前，很多数据库都实现了**Read Committed**这个隔离级别。

它甚至成为了很多数据库的**默认隔离级别**设置。

比如 Oracle 11g、*PostgreSQL*、*SQL Server 2012*、 *MemSQL 等等。*

最常见的实现**避免脏写**的方式是**行级锁 (Row-level Locks)。**

当一个事务想要写一些对象（比如：行或者 NoSQL 中的 document）时，它必须要满足两个条件：

首先，获得一个这个对象的锁
并且，持有这个锁直到这个事务 committed 或者 回滚 (Aborted)。
同时要求！**只有一个**事务可以持有这个锁
如果其他事务也想持有同一个锁，就必须要等到这个锁被**释放**才可以。
（这里的锁是在read committed或者更强的隔离级别下由数据库自动控制的。）

如何避免脏读？

我们可以通过加**行级锁**的方式进行。

但是，现实的使用情况中，读的请求要比写的请求多很多。

如果一个事务读了数据，并长时间占用当前的锁，就会**阻塞**其他事务读这一条数据，进而导致**性能**严重受到影响。

因此在实际应用中，**更常用**的方式是：

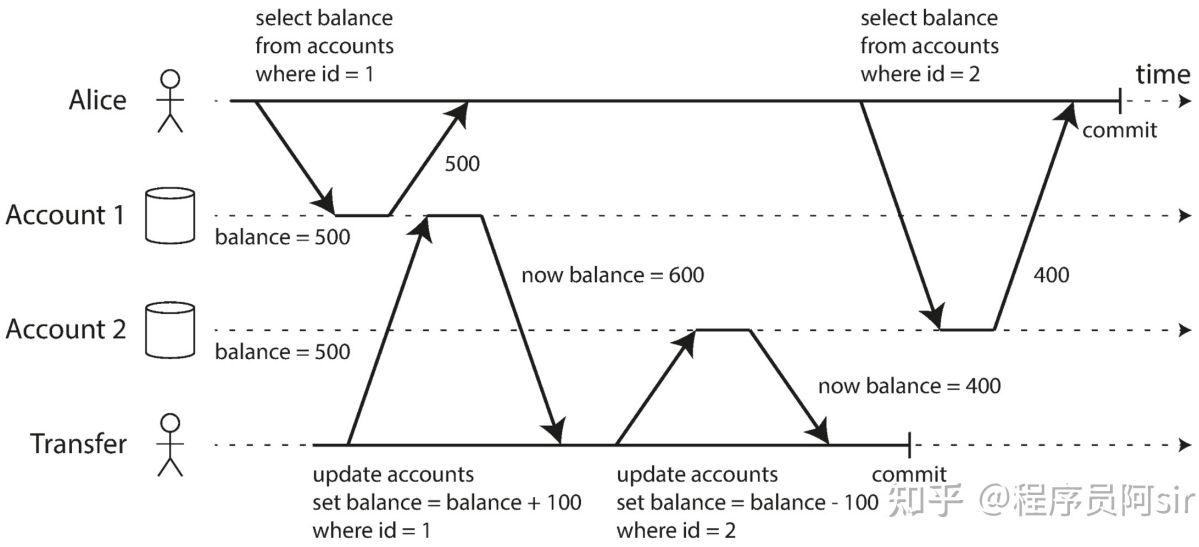
数据库记下来旧的已经commited的值以及持有写锁 (Write Lock) 的事务的新值。
当事务**没结束**时，其他任何事务都只能读到**旧值**；
当事务**commited**时，再用当前事务中存的新值**替换**旧值。

1.3. Read Committed 问题

看上去好像 read committed可以解决所有并发问题。但是实际其中有很多bug：

比如不可重复读问题 (Nonrepeatable Read)，也称为读时序异常（Read Skew)。

如下图所示：



在这个例子中，假设 Alice 有两个账号：账号 1 和账号 2。
他从账号 2 中给账号 1 转账 100 元。
现在图中是两个事务：转账的事务、查询两账号余额的事务。
这里面查询账号 1 余额的操作发生在更新账号 1 余额的事务之前。
因此读到账户 1 的余额是500。
然后，转账事务开始，成功把两个账号余额更新后，结束。
这时，查询账号 2 余额的操作被执行，读到余额是400。
这样 Alice 总共看到两账户总余额是900元，其中100元凭空消失了。
但是，如果 Alice 在查询事务结束之前，又一次查询了一次账号 1 中的余额，会发现读到值更新为 600 元。
这时候余额总和又正常了！

在一次事务中读同一个数据却读到了两个不同的值。这就是不可重复读 (Nonrepeatalbe Read)

这也是满足Read Committed的要求的

（即没有脏读脏写）。

其实， Alice 的例子只是一个暂时的问题。

因为最终他的账户余额肯定是一致的。

虽然，不可重复读虽然给 Alice 带来了一点困惑，但是没什么实际的损失。

但是！存在着某些问题是**不能容忍**这种临时不一致的状态的！

比如**备份 (Backups)**

> 此处的**备份**指对整个数据库的备份

实际应用中，一般要花费几个小时才能完成大数据库的备份。

在备份过程中，数据库接受着各种**写操作**，因此会存在旧的值已经被写入备份 (比如上面例子中账户 1 的余额 500 元已被写入)

假设此时未经备份的账户2，出现了上述例子中的转账行为（账户2给账户1转账100元，此时账户2的余额变成了 400 元）

账户2此时余额将会以的新值400元被写入备份。

这样备份的数据就发生**数据不一致**的情况。

> ***备份数据库：****账户1余额500元，账户2余额400元*
> ***原数据库：****账户1余额500元，账户2余额400元*

当需要从备份恢复数据库，将会出现不可预料的问题。

为了解决这个不可重复读的问题，需要一种更强一些的隔离级别--快照隔离 (Snapshot Isolation)

我们将在下一篇文章中具体介绍它。

（未完待续）

参考文献

[1] Kleppmann, Martin. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. " O'Reilly Media, Inc.", 2017.