PostgreSQL的MVCC并发处理

06-06 18:41
1.59k

原文:	https://devcenter.heroku.com/articles/postgresql-concurrency
翻译:	piglei

Postgre数据库的很大的卖点之一就是它处理并发的方式。我们的期望很简单：读永远不阻塞写，反之亦然。Postgres通过一个叫做 多版本并发控制*(MVCC)* 的机制做到了这一点。这个技术并不是Postgres所特有的：还有好几种数据库都实现了不同形式的MVCC，包括 Oracle、Berkeley DB、CouchDB 等等 。当你使用PostgreSQL来设计高并发的应用时，理解它的MVCC是怎么实现的很重要。它事实上是复杂问题的一种非常优雅和简单的解法。

MVCC如何工作

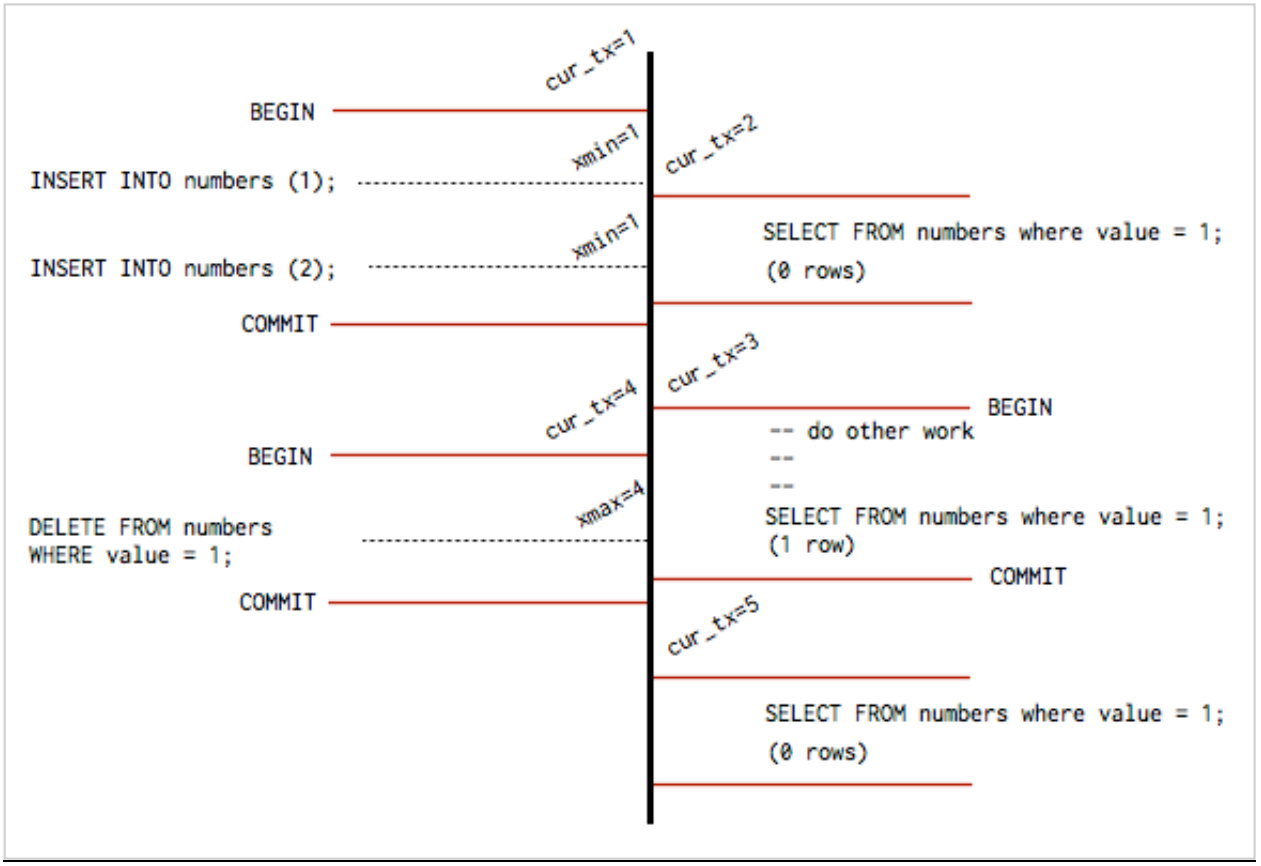
在Postgres中，每一个事务都会得到一个被称作为 **XID** 的事务ID。这里说的事务不仅仅是被 `BEGIN - COMMIT` 包裹的一组语句，还包括单条的insert、update或者delete语句。当一个事务开始时，Postgrel递增XID，然后把它赋给这个事务。Postgres还在系统里的每一行记录上都存储了事务相关的信息，这被用来判断某一行记录对于当前事务是否可见。

举个例子，当你插入一行记录时，Postgre会把当前事务的XID存储在这一行中并称之为 `xmin` 。只有那些*已提交的而且 `xmin`` 比当前事务的XID小的记录对当前事务才是可见的。这意味着，你可以开始一个新事务然后插入一行记录，直到你提交（ `COMMIT` ）之前，你插入的这行记录对其他事务永远都是不可见的。等到提交以后，其他后创建的新事务就可以看到这行新记录了，因为他们满足了 `xmin < XID` 条件，而且创建哪一行记录的事务也已经完成。

对于 `DELETE` 和 `UPDATE` 来说，机制也是类似的，但不同的是对于它们Postgres使用叫做 `xmax` 的值来判断数据的可见性。这幅图展示了在两个并发的插入/读取数据的事务中，MVCC在事务隔离方面是怎么起作用的。

在下面的图中，假设我们先执行了这个建表语句：

```
CREATE TABLE numbers (value int);
```



虽然 `xmin` 和 `xmax` 的值在日常使用中都是被隐藏的，但是你可以直接请求他们，Postgres会高兴的把值给你：

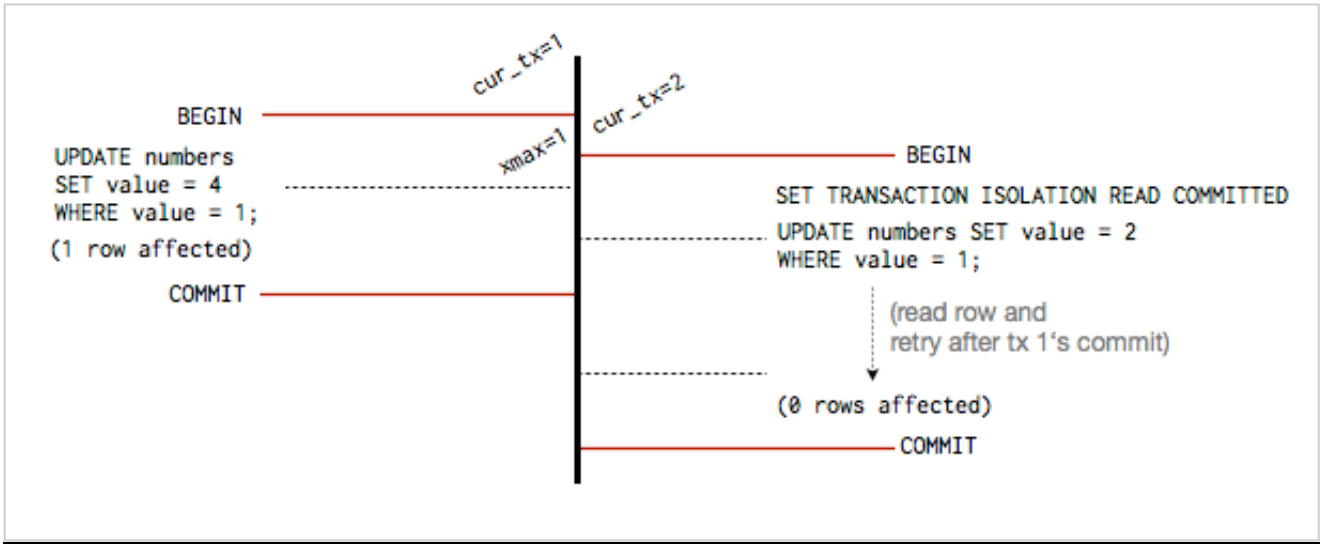
```
SELECT *, xmin, xmax FROM numbers;
```

获取当前事务的XID也很简单：

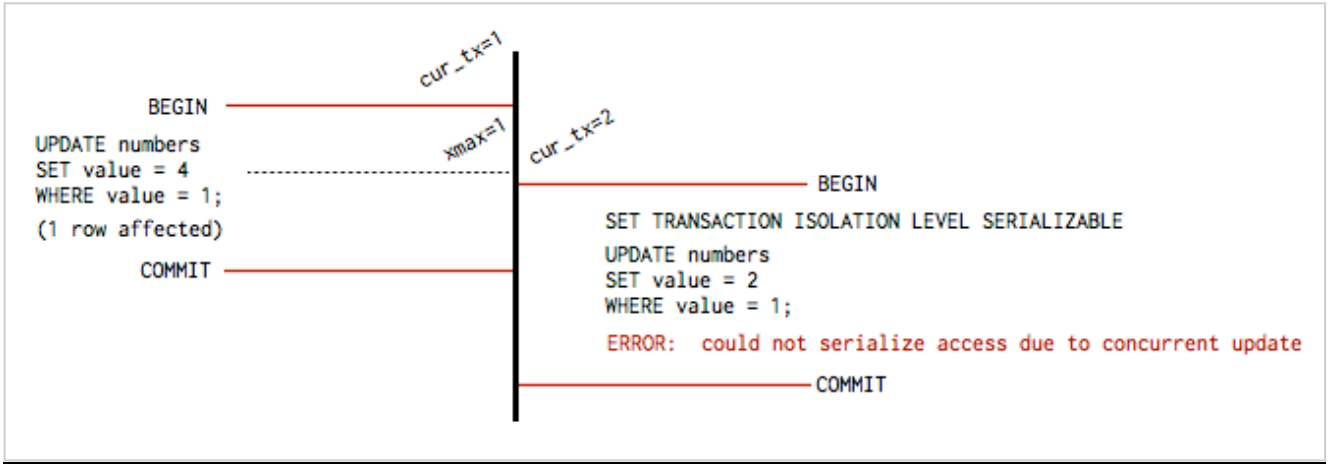
```
SELECT txid_current();
```

干净利落！

我知道你现在在想：要是同时有两个事务修改同一行数据会怎么样？这就是事务隔离级别（transaction isolation levels）登场的时候了。Postgres支持两个基本的模型来让你控制应该怎么处理这样的情况。默认情况下使用 `读已提交（READ COMMITTED）` ，等待初始的事务完成后再读取行记录然后执行语句。如果在等待的过程中记录被修改了，它就从头再来一遍。举一个例子，当你执行一条带有 `WHERE` 子句的 `UPDATE` 时， `WHERE` 子句会在最初的事务被提交后返回命中的记录结果，如果这时 `WHERE` 子句的条件任然能得到满足的话， `UPDATE` 才会被执行。在下面这个例子中，两个事务同时修改同一行记录，最初的 `UPDATE` 语句导致第二个事务的 `WHERE` 不会返回任何记录，因此第二个事务根本没有修改到任何记录：



如果你需要更好的控制这种行为，你可以把事务隔离级别设置为 `可串行化（SERIALIZABLE）` 。在这个策略下，上面的场景会直接失败，因为它遵循这样的规则：“如果我正在修改的行被其他事务修改过的话，就不再尝试”，同时 Postgres会返回这样的错误信息：`由于并发修改导致无法进行串行访问` 。捕获这个错误然后重试就是你的应用需要去做的事情了，或者不重试直接放弃也行，如果那样合理的话。



MVCC的缺点

现在你已经知道MVCC和事务隔离是怎么工作了吧，你获得了又一个工具用来解决这类问题：`可串行化事务隔离级别` 迟早会派上用场。然而MVCC的优点虽然很明显但它也存在着一些缺点。

因为不同的事务会看到不同状态的记录，Postgres连那些可能过期的数据也需要保留着。这就是为什么 `UPDATE` 实际上是创建一行新纪录而 `DELETE` 并不真正的删除记录（它只是简单的把记录标记成已删除然后设置XID的值）的原因。当事务完成后，数据库里会存在一些对以后的事务永远不可见的记录。它们被称作dead rows。MVCC带来的另外一个问题是，事务的ID只能不断的增加 - 它是32个bits，只能”支持大约四十亿个事务。当XID达到最大值后，它会变回零重新开始。突然间所有的记录都变成了发生在将来的事务所产生的，所有的新事务都没有办法访问到这些旧记录了。

上面说到的dead row和事务XID循环问题都是通过执行VACUUM命令（Postgres用来执行清理操作的命令）来解决的。这应该成为一个例行的维护，所以Postgre自带了auto_vacuum守护进程会在一个可配置的周期内自动执行清理。留意点auto_vacuum很重要，因为在不同的部署环境中需要执行清理的周期也会不同。你可以在Postgres的文档里找到关于VACUUM的更多说明。