

数据库，主要讲述的是一个章节的前半部分，关于事务和并发控制。在本章中，将介绍事务的基本概念、事务的语义、事务的隔离级别以及如何保证事务的一致性。同时，还将讨论并发控制的基本原理，包括锁机制、MVCC等，并分析它们在 PostgreSQL 中的具体实现。

第 7 章

Chapter 7

事务与并发控制

在本章中，将首先介绍事务的基本概念、语义和隔离级别，然后探讨如何通过锁机制来保证事务的一致性和并发控制。接着，将分析 PostgreSQL 中的 MVCC 机制，并讨论如何通过事务的提交点（commit point）来保证一致性。最后，将介绍如何通过锁机制来避免死锁，并分析 PostgreSQL 中的死锁检测和回滚策略。

事务是关系型数据库最重要的概念，而并发通常能带来更大的吞吐量、资源利用率和更好的性能。但是当多个事务并发执行时，即使每个单独的事务都正确执行，数据库的一致性也可能被破坏。为了控制并发事务之间的相互影响，解决并发可能带来的资源争用及数据不一致性问题，数据库的并发控制系统引入了基于锁的并发控制机制（Lock-Based Concurrency Control）和基于多版本的并发控制机制 MVCC（Multi-Version Concurrency Control）。本章将简单介绍 PostgreSQL 事务的概念，并重点讨论 PostgreSQL MVCC 的基本知识、工作细节和原理，数据库管理员和应用开发者都应该熟悉它。

7.1 事务和并发控制的概念

7.1.1 事务的基本概念和性质

事务是数据库系统执行过程中最小的逻辑单位。当事务被提交时，数据库管理系统要确保一个事务中的所有操作都成功完成，并且在数据库中永久保存操作结果。如果一个事务中的一部分操作没有成功完成，则数据库管理系统会把数据库回滚到操作执行之前的状态。在 PostgreSQL 中，显式地指定 BEGIN...END/COMMIT/ROLLBACK 包括的语句块或一组语句为一个事务，未指定 BEGIN...END/COMMIT/ROLLBACK 的单条语句也称为一个事务。事务有四个重要的特性：原子性、一致性、隔离性、持久性。

- □ 原子性（Atomicity）：一个事务的所有操作，要么全部执行，要么全部不执行。
- □ 一致性（Consistency）：执行事务时保持数据库从一个一致的状态变更到另一个一致的状态。
- 隔离性（Isolation）：即使每个事务都能确保一致性和原子性，如果并发执行时，由



于它们的操作以人们不希望的方式交叉运行，就会导致不一致的情况发生。确保事务与事务并发执行时，每个事务都感觉不到有其他事务在并发地执行。

- 持久性 (Durability)：一个事务完成之后，即使数据库发生故障，它对数据库的改变应该永久保存在数据库中；

这四个特性分别取它们名称的首字母，通常习惯称之为 ACID。其中，事务一致性由主键、外键这类约束保证，持久性由预写日志 (WAL) 和数据库管理系统的恢复子系统保证，原子性、隔离性则由事务管理器和 MVCC 来控制。

7.1.2 并发引发的现象

如果所有的事务都按照顺序执行，所有事务的执行时间没有重叠交错就不会存在事务并发性。如果以不受控制的方式允许具有交织操作的并发事务，则可能发生不期望的结果。这些不期望的结果可能被并发地写入和并发地读取而得到非预期的数据。PostgreSQL 中可以把这些非预期的现象总结为：脏读 (Dirty read)、不可重复读 (Non-repeatable read)、幻读 (Phantom Read) 和序列化异常 (Serialization Anomaly)。下面我们会分别举例演示这几种非预期的读现象。

 **注意** 由于 PostgreSQL 内部将 READ UNCOMMITTED 设计为和 READ COMMITTED 一样，在 PostgreSQL 数据库中无论如何都无法产生脏读，所以读者可以使用能够出现脏读现象的数据库观察脏读现象。由于 PostgreSQL 在 READ COMMITTED 隔离级别可能出现不可重复读、幻读的现象，所以例子中都使用 PostgreSQL 默认的事务隔离级别 READ COMMITTED 来演示。关于隔离级别的概念后文会有解释。

1. 脏读

当第一个事务读取了第二个事务中已经修改但还未提交的数据，包括 INSERT、UPDATE、DELETE，当第二个事务不提交并执行 ROLLBACK 后，第一个事务所读取到的数据是不正确的，这种读现象称作脏读。下面演示一个脏读的例子。

首先创建一张测试表并插入测试数据，如下所示：

```
CREATE TABLE tbl_mvcc (
    id INT NOT NULL AUTO_INCREMENT,
    ival INT,
    PRIMARY KEY (id)
);
-- 插入一条测试数据
INSERT INTO tbl_mvcc (ival) VALUES (1);
```

按照从上到下的顺序分别执行 T1 和 T2 如下所示：

T1	T2
<pre>set session transaction isolation level read uncommitted;start transaction; select * from tbl_mvcc where id = 1; /* id=1,ival=1 */</pre>	

(续)

T1	T2
<pre>start transaction; update tbl_mvcc set ival = 10 where id = 1;</pre> <pre>select * from tbl_mvcc where id = 1; /* id=1, ival=10 */</pre>	<pre>rollback;</pre>

上面的例子中，事务 T1 在 `tbl_mvcc` 表中查询数据，得到 `id=1, ival=1` 的行，这时事务 T2 更新表中 `id=1` 的行的 `ival` 值为 10，此时事务 T1 查询 `tbl_mvcc` 表，而事务 T2 此时并未提交，`ival` 预期的值理应等于 1，但是事务 T1 却得到了 `ival` 等于 10 的值。事务 T2 最终进行了 `ROLLBACK` 操作，很显然，事务 T1 将得到错误的值，引发了脏读现象。

2. 不可重复读

当一个事务第一次读取数据之后，被读取的数据被另一个已提交的事务进行了修改，事务再次读取这些数据时发现数据已经被另一个事务修改，两次查询的结果不一致，这种读现象称为不可重复读。下面演示一个不可重复读的例子。

首先创建一张测试表并插入测试数据，如下所示：

```
CREATE TABLE tbl_mvcc (
    id SERIAL PRIMARY KEY,
    ival INT
);
-- 插入一条测试数据
INSERT INTO tbl_mvcc (ival) VALUES (1);
```

按照从上到下的顺序分别执行 T1 和 T2，如下所示：

T1	T2
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;	
SELECT id, ival FROM tbl_mvcc WHERE id = 1;	
<pre>id ival ---+--- 1 1 (1 row)</pre>	
	<pre>BEGIN; UPDATE tbl_mvcc SET ival = 10 WHERE id = 1; COMMIT;</pre>
SELECT id, ival FROM tbl_mvcc WHERE id = 1;	
<pre>id ival ---+--- 1 10 (1 row)</pre>	
END;	

在上面的例子中，事务 T1 在 tbl_mvcc 表中第一次查询数据，得到 id=1, ival=1 的行，这时事务 T2 更新表中 id=1 的行的 ival 值为 10，并且事务 T2 成功地进行了 COMMIT 操作。此时事务 T1 查询 tbl_mvcc 表，得到 ival 的值等于 10，我们的预期是数据库在第二次 SELECT 请求的时候，应该返回事务 T2 更新之前的值，但实际查询到的结果与第一次查询得到的结果不同，由于事务 T2 的并发操作，导致事务 T1 不能重复读取到预期的值，这就是不可重复读的现象。

3. 幻读

指一个事务的两次查询的结果集记录数不一致。例如一个事务第一次根据范围条件查询了一些数据，而另一个事务却在此时插入或删除了这个事务的查询结果集中的部分数据，这个事务在接下来的查询中，会发现有一些数据在它先前的查询结果中不存在，或者第一次查询结果中的一些数据不存在了，两次查询结果不相同，这种读现象称为幻读。幻读可以认为是受 INSERT 和 DELETE 影响的不可重复读的一种特殊场景。

下面演示一个幻读的例子，使用前面创建的测试表 tbl_mvcc 和表中的测试数据，如下所示：

T1	T2
<pre>BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED; SELECT id,ival FROM tbl_mvcc WHERE id > 3 AND id < 10; id ival ----+--- 4 4 5 5 (2 rows)</pre>	
	<pre>BEGIN; INSERT INTO tbl_mvcc (id , ival) VALUES (6 , 6); END;</pre>
<pre>SELECT id,ival FROM tbl_mvcc WHERE id > 3 AND id < 10; id ival ----+--- 4 4 5 5 6 6 (3 rows)</pre>	
END;	

在上面的例子中，事务 T1 在 tbl_mvcc 表中第一次查询 id 大于 3 并且小于 10 的数据，得到两行数据，这时事务 T2 在表中插入了一条 id 等于 6 的数据，这条数据正好满足事务 T1 的 WHERE 条件中，id 大于 3 并且小于 10 的查询条件，事务 T1 再次查询时，查询结

果会多一条非预期的数据，像产生了幻觉。不可重复读和幻读很相似，它们之间的区别主要在于不可重复读主要受到其他事务对数据的 UPDATE 操作，而幻读主要受到其他事务 INSERT 和 DELETE 操作的影响。

7.1.3 ANSI SQL 标准的事务隔离级别

为了避免事务与事务之间并发执行引发的副作用，最简单的方法是串行化地逐个执行事务，但是串行化地逐个执行事务会严重降低系统吞吐量，降低硬件和系统的资源利用率。为此，ANSI SQL 标准定义了四类隔离级别，每一个隔离级别都包括了一些具体规则，用来限定事务内外的哪些改变对其他事务是可见的，哪些是不可见的，也就是允许或不允许出现脏读、不可重复读，幻读的现象。通过这些事务隔离级别规定了一个事务必须与其他事务所进行的资源或数据更改相隔离的程度。这四类事务隔离级别包括：

- Read Uncommitted（读未提交）：在该隔离级别，所有事务都可以看到其他未提交事务的执行结果，在多用户数据库中，脏读是非常危险的，在并发情况下，查询结果非常不可控，即使不考虑结果的严谨性只追求性能，它的性能也并不比其他事务隔离级别好多少，可以说脏读没有任何好处。所以未提交读这一事务隔离级别很少用于实际应用。
- Read Committed（读已提交）：这是 PostgreSQL 的默认隔离级别，它满足了一个事务只能看见已经提交事务对关联数据所做的改变的隔离需求。
- Repeatable Read（可重复读）：确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。
- Serializable（可序列化）：这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

下面的表格是 ANSI SQL 标准定义的事务隔离级别与读现象的关系：

隔 离 级 别	脏 读	不 可 重 复 读	幻 读
Read Uncommitted	可能	可能	可能
Read Committed	不可能	可能	可能
Repeatable Read	不可能	不可能	可能
Serializable	不可能	不可能	不可能

对于同一个事务来说，不同的事务隔离级别执行结果可能会不同。隔离级别越高，越能保证数据的完整性和一致性，但是需要更多的系统资源，增加了事务阻塞其他事务的概率，对并发性能的影响也越大，吞吐量也会更低；低级别的隔离级别一般支持更高的并发处理，并拥有更低的系统开销，但增加了并发引发的副作用的影响。对于多数应用程序，优先考虑 Read Committed 隔离级别。它能够避免脏读，而且具有较好的并发性能。尽管它

会导致不可重复读、幻读和丢失更新这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制。

7.2 PostgreSQL 的事务隔离级别

尽管不同的数据库系统中事务隔离的实现不同，但都会遵循 SQL 标准中“不同事务隔离级别必须避免哪一种读现象发生”的约定。SQL 标准中 Read Uncommitted 的事务隔离级别是允许脏读的，其本意应该是数据库管理系统在这一事务隔离级别能够支持非阻塞的读，即常说的写不阻塞读，但 PostgreSQL 默认就提供了非阻塞读，因此在 PostgreSQL 内部只实现了三种不同的隔离级别，PostgreSQL 的 Read Uncommitted 模式的行为和 Read Committed 相同，并且 PostgreSQL 的 Repeatable Read 实现不允许幻读。而 SQL 标准定义的四种隔离级别只定义了哪种现象不能发生，描述了每种隔离级别必须提供的最小保护，但是没有定义哪种现象必须发生，这是 SQL 标准特别允许的。在 PostgreSQL 中，依然可以使用四种标准事务隔离级别的任意一种，但是要理解 PostgreSQL 的事务隔离级别有别于其他数据库隔离级别的定义。除此以外，这里还引入了一个新的数据冲突问题：序列化异常。序列化异常是指成功提交的一组事务的执行结果与这些事务按照串行执行方式的执行结果不一致。下面演示一个序列化异常的例子，按照从上到下的顺序分别执行 T1 和 T2，如下所示：

T1	T2
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ ; SELECT id,ival FROM tbl_mvcc WHERE id = 1; id ival ---+--- 1 1 (1 rows)	
	UPDATE tbl_mvcc SET ival = ival * 10 WHERE id = 1;
UPDATE tbl_mvcc SET ival = ival + 1 WHERE id = 1; ERROR: could not serialize access due to concurrent update	
ROLLBACK	

在上面的例子中，事务 T1 开始时查询出 id=1 的数据，事务 T2 在事务 T1 提交之前对数据做了更新操作，并且在事务 T1 提交之前提交成功；当事务 T1 提交时，如果按照先执行 T2 再执行 T1 的顺序执行，事务 T1 在事务开始时查询到的数据应该是事务 T2 提交之后的结果：ival=10，但由于事务 T1 是可重复读的，当它进行 UPDATE 时，事务 T1 读到的数据却是它开始时读到的数据：ival=1；这时就发生了序列化异常的现象。Serializable 与 Repeatable Read 在 PostgreSQL 里是基本一样的，除了 Serializable 不允许序列化异常。

下面的表格是 PostgreSQL 中不同的事务隔离级别与读现象的关系：

隔离级别	脏读	不可重复读	幻读	序列化异常
Read Uncommitted	不可能	可能	可能	可能
Read Committed	不可能	可能	可能	可能
Repeatable Read	不可能	不可能	不可能	可能
Serializable	不可能	不可能	不可能	不可能

下面我们通过一个例子验证在 Repeatable Read 事务隔离级别不可能出现幻读，如下所示：

T1	T2
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ; SELECT id, ival FROM tbl_mvcc WHERE id > 3 AND id < 10; id ival ----+--- 4 4 5 5 (2 rows)	
	BEGIN; INSERT INTO tbl_mvcc (id , ival) VALUES (6 , 6); END;
SELECT id, ival FROM tbl_mvcc WHERE id > 3 AND id < 10; id ival ----+--- 4 4 5 5 (2 rows)	
END;	

在上面的例子中，事务 T1 在 `tbl_mvcc` 表中第一次查询 `id` 大于 3 并且小于 10 的数据，得到两行数据，这时事务 T2 在表中插入了一条 `id` 等于 6 的数据，这条数据正好满足事务 T1 的 `WHERE` 条件中，`id` 大于 3 并且小于 10 的查询条件，事务 T1 再次查询时，与第一次查询的结果相同，说明没有出现幻读现象。其他事务隔离级别对读现象的影响这里不再演示，有兴趣的读者可以自行实验。

7.2.1 查看和设置数据库的事务隔离级别

PostgreSQL 默认的事务隔离级别是 `Read Committed`。查看全局事务隔离级别的代码如下所示：

```
mydb=# SELECT name, setting FROM pg_settings WHERE name = 'default_transaction_isolation';
          name           |      setting
-----+-----

```



```
default_transaction_isolation | repeatable read  
(1 row)
```

或：

```
mydb=# SELECT current_setting('default_transaction_isolation');  
current_setting  
-----  
repeatable read  
(1 row)
```

7.2.2 修改全局的事务隔离级别

方法 1：通过修改 postgresql.conf 文件中的 default_transaction_isolation 参数修改全局事务隔离级别，修改之后 reload 实例使之生效；

方法 2：通过 ALTER SYSTEM 命令修改全局事务隔离级别：

```
mydb=# ALTER SYSTEM SET default_transaction_isolation TO 'REPEATABLE READ';  
ALTER SYSTEM  
mydb=# SELECT pg_reload_conf();  
pg_reload_conf  
-----  
t  
(1 row)  
mydb=# SELECT current_setting('transaction_isolation');  
current_setting  
-----  
repeatable read  
(1 row)
```

7.2.3 查看当前会话的事务隔离级别

查看当前会话的事务隔离级别的代码如下所示：

```
mydb=# SHOW transaction_isolation ;  
transaction_isolation  
-----  
read committed  
(1 row)
```

或：

```
mydb=# SELECT current_setting('transaction_isolation');  
current_setting  
-----  
read committed  
(1 row)
```

7.2.4 设置当前会话的事务隔离级别

设置当前会话的事务隔离级别的代码如下所示：



```
mydb=# SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET
mydb=# SHOW transaction_isolation ;
transaction_isolation
-----
read uncommitted
(1 row)
```

7.2.5 设置当前事务的事务隔离级别

在启动事务的同时设置事务隔离级别，如下所示：

```
mydb=# START TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

```
START TRANSACTION
```

```
...
```

```
...
```

```
mydb=# END;
```

```
COMMIT
```

或：

```
mydb=# BEGIN ISOLATION LEVEL READ UNCOMMITTED READ WRITE;
```

```
...
```

```
...
```

```
...
```

```
mydb=# END/COMMIT/ROLLBACK;
```

START TRANSACTION 和 BEGIN 都是开启一个事务，具有相同的功能。

7.3 PostgreSQL 的并发控制

在多用户环境中，允许多人同时访问和修改数据，为了保持事务的隔离性，系统必须对并发事务之间的相互作用加以控制，在这种情况下既要确保用户以一致的方式读取和修改数据，还要争取尽量多的并发数，这是数据库管理系统的并发控制器需要做的事情。当多个事务同时执行时，即使每个单独的事务都正确执行，数据的一致性也可能被破坏。为了控制并发事务之间的相互影响，应把事务与事务在逻辑上隔离开，以保证数据库的一致性。数据库管理系统中并发控制的任务便是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性、数据的一致性以及数据库的一致性，也就是解决丢失更新、脏读、不可重复读、幻读、序列化异常的问题。并发控制模型有基于锁的并发控制 (Lock-Based Concurrency Control) 和基于多版本的并发控制 (Multi-Version Concurrency Control)。封锁、时间戳、乐观并发控制（又名“乐观锁”，Optimistic Concurrency Control，缩写为“OCC”）和悲观并发控制（又名“悲观锁”，Pessimistic Concurrency Control，缩写为“PCC”）是并发控制采用的主要技术手段。



7.3.1 基于锁的并发控制

为了解决并发问题，数据库引入了“锁”的概念。基本的封锁类型有两种：排它锁（Exclusive locks，X 锁）和共享锁（Share locks，S 锁）。

排它锁：被加锁的对象只能被持有锁的事务读取和修改，其他事务无法在该对象上加其他锁，也不能读取和修改该对象。

共享锁：被加锁的对象可以被持锁事务读取，但是不能被修改，其他事务也可以在上面再加共享锁。

封锁对象的大小称为封锁粒度（Granularity）。封锁的对象可以是逻辑单元，也可以是物理单元。以关系数据库为例，封锁对象可以是这样一些逻辑单元：属性值、属性值的集合、元组、关系、索引项、整个索引项甚至整个数据库；也可以是这样的一些物理单元：页（数据页或索引页）、物理记录等。封锁的策略是一组规则，这些规则阐明了事务何时对数据项进行加锁和解锁，通常称为封锁协议（Locking Protocol）。由于采用了封锁策略，一次只能执行一个事务，所以只会产生串行调度，迫使事务只能等待前面的事务结束之后才可以开始，所以基于锁的并发控制机制导致性能低下，并发程度低。

关于锁以及 PostgreSQL 特有的 Advisory Lock（咨询锁）的相关书籍和文档都比较丰富，本书不做赘述。

7.3.2 基于多版本的并发控制

基于锁的并发控制机制要么延迟一项操作，要么中止发出该操作的事务来保证可串行性。如果每一数据项的旧值副本保存在系统中，这些问题就可以避免。这种基于多个旧值版本的并发控制即 MVCC。一般把基于锁的并发控制机制称成为悲观机制，而把 MVCC 机制称为乐观机制。这是因为锁机制是一种预防性的机制，读会阻塞写，写也会阻塞读，当封锁粒度较大，时间较长时并发性能就不会太好；而 MVCC 是一种后验性的机制，读不阻塞写，写也不阻塞读，等到提交的时候才检验是否有冲突，由于没有锁，所以读写不会相互阻塞，避免了大粒度和长时间的锁定，能更好地适应对读的响应速度和并发性要求高的场景，大大提升了并发性能，常见的数据库如 Oracle、PostgreSQL、MySQL（Innodb）都使用 MVCC 并发控制机制。在 MVCC 中，每一个写操作创建一个新的版本。当事务发出一个读操作时，并发控制管理器选择一个版本进行读取。也就是为数据增加一个关于版本的标识，在读取数据时，连同版本号一起读出，在更新时对此版本号加一。

MVCC 通过保存数据在某个时间点的快照，并控制元组的可见性来实现。快照记录 READ COMMITTED 事务隔离级别的事务中的每条 SQL 语句的开头和 SERIALIZABLE 事务隔离级别的事务开始时的元组的可见性。一个事务无论运行多长时间，在同一个事务里都能够看到一致的数据。根据事务开始的时间不同，在同一个时刻不同事务看到的相同表里的数据可能是不同的。



PostgreSQL 为每一个事务分配一个递增的、类型为 int32 的整型数作为唯一的事务 ID，称为 xid。创建一个新的快照时，将收集当前正在执行的事务 id 和已提交的最大事务 id。根据快照提供的信息，PostgreSQL 可以确定事务的操作是否对执行语句是可见的。PostgreSQL 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。在 PostgreSQL 的内部数据结构中，每个元组（行记录）有 4 个与事务可见性相关的隐藏列，分别是 xmin、xmax、cmin、cmax，其中 cmin 和 cmax 分别是插入和删除该元组的命令在事务中的命令序列标识，xmin、xmax 与事务对其他事务的可见性相关，用于同一个事务中的可见性判断。可以通过 SQL 直接查询到它们的值，如下所示：

```
mydb=# SELECT xmin,xmax,cmin,cmax,id,ival FROM tbl_mvcc WHERE id = 1;
      xmin | xmax | cmin | cmax | id | ival
-----+-----+-----+-----+
    1930 |     0 |     0 |     0 |  1 |    1
(1 row)
```

其中 xmin 保存了创建该行数据的事务的 xid，xmax 保存的是删除该行的 xid，PostgreSQL 在不同事务时间使用 xmin 和 xmax 控制事务对其他事务的可见性。

1. 通过 xmin 决定事务的可见性

当插入一行数据时，PostgreSQL 会将插入这行数据的事务的 xid 存储在 xmin 中。通过 xmin 值判断事务中插入的行记录对其他事务的可见性有两种情况：

1) 由回滚的事务或未提交的事务创建的行对于任何其他事务都是不可见的。例如我们开启一个新的事务，如下所示：

```
mydb=# BEGIN;
mydb=# SELECT txid_current();
      txid_current
-----
      1937
(1 row)
mydb=# INSERT INTO tbl_mvcc(id,ival) VALUES(7,7);
INSERT 0 1
mydb=# SELECT xmin,xmax,cmin,cmax,id,ival FROM tbl_mvcc WHERE id = 7;
      xmin | xmax | cmin | cmax | id | ival
-----+-----+-----+-----+
    1937 |     0 |     0 |     0 |  7 |    7
(1 row)
```

通过 SELECT txid_current() 语句我们查询到当前的事务的 xid 是 1937，插入一条 id 等于 7 的数据，查询这条新数据的隐藏列可以看到 xmin 的值等于 1937，也就是插入这行数据的事务的 xid。

开启另外一个事务，如下所示：

```
mydb=# BEGIN;
BEGIN
```



```
mydb=# SELECT txid_current();
txid_current
-----
1938
(1 row)

mydb=# SELECT * FROM tbl_mvcc WHERE id = 7;
id | ival
-----+
(0 rows)

mydb=# END;
COMMIT
```

可以看到由于第一个事务并没有提交，所以第一个事务对第二个事务是不可见的。

2) 无论提交成功或回滚的事务，xid 都会递增。对于 Repeatable Read 和 Serializable 隔离级别的事务，如果它的 xid 小于另外一个事务的 xid，也就是元组的 xmin 小于另外一个事务的 xmin，那么另外一个事务对这个事务是不可见的。下面举例说明。

```
mydb=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
mydb=# SELECT txid_current();
txid_current
-----
1939
(1 row)
```

以上语句开启了一个事务，这个事务的 xid 是 1939。再开始另外一个事务如下所示：

```
mydb=# BEGIN;
BEGIN
mydb=# SELECT txid_current();
txid_current
-----
1940
(1 row)

mydb=# INSERT INTO tbl_mvcc (id,ival) VALUES (7,7);
INSERT 0 1
mydb=# SELECT xmin,xmax,cmin,cmax,id,ival FROM tbl_mvcc WHERE id = 7;
xmin | xmax | cmin | cmax | id | ival
-----+-----+-----+-----+-----+
1940 |     0 |     0 |     0 |    7 |    7
(1 row)
mydb=# COMMIT;
COMMIT
```

第二个事务的 xid 是 1940，并在这个事务中在表中插入一条新的数据，xmin 记录了第二个事务的 xid，第二个事务提交成功。在第一个事务中查询第二个事务提交的数据，如下所示：





```
mydb=# SELECT xmin,xmax,cmin,cmax,id,ival FROM tbl_mvcc WHERE id = 7;
      xmin | xmax | cmin | cmax | id | ival
-----+-----+-----+-----+-----+
(0 rows)
mydb=# END;
COMMIT
```

从上面的例子可以看到，尽管第二个事务提交成功，但在第一个事务中并未能查询到第二个事务插入的数据，因为第一个事务的 XID 是 1939，第二个事务插入的数据的 xmin 值是 1940，小于第二个事务的 xmin，所以插入的 id 等于 7 的数据对第一个事务是不可见的。

2. 通过 xmax 决定事务的可见性

通过 xmax 值判断事务的更新操作和删除操作对其他事务的可见性有这几种情况：1) 如果没有设置 xmax 值，该行对其他事务总是可见的；2) 如果它被设置为回滚事务的 xid，该行对其他事务也是可见的；3) 如果它被设置为一个正在运行，没有 COMMIT 和 ROLLBACK 的事务的 xid，该行对其他事务是可见的；4) 如果它被设置为一个已提交的事务的 xid，该行对在这个已提交事务之后发起的所有事务都是不可见的。

7.3.3 通过 pageinspect 观察 MVCC

通过 PostgreSQL 文件系统的存储格式可以理解得更清晰。在 PostgreSQL 中，可以使用 pageinspect 这个外部扩展来观察数据库页面的内容。pageinspect 提供了一些函数可以得到数据库的文件系统中页面的详细内容，使用它之前先在数据库中创建扩展，如下所示：

```
mydb=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
mydb=# \dx+ pageinspect
      Objects in extension "pageinspect"
      Object description
-----
...                                          
function get_raw_page(text,integer)
...
function heap_page_items(bytea)
...
(19 rows)
```

下面介绍两个会用到的函数。get_raw_page get_raw_page(relname text, fork text, blkno int) 和它的一个重载 get_raw_page(relname text,blkno int)，用于读取 relation 中指定的块的值，其中 relname 是 relation name，参数 fork 可以有 main、vm、fsm、init 这几个值，fork 默认值是 main，main 表示数据文件的主文件，vm 是可见性映射的块文件，fsm 为 free space map 的块文件，init 是初始化的块。get_raw_page 以一个 bytea 值的形式返回一个拷贝。heap_page_items heap_page_items 显示一个堆页面上所有的行指针。对那些使用中的





行指针，元组头部和元组原始数据也会被显示。不管元组对于拷贝原始页面时的 MVCC 快照是否可见，它们都会被显示。一般使用 `get_raw_page` 函数获得堆页面映像作为参数传递给 `heap_page_items`。

我们创建如下的视图以便更清晰地观察 PostgreSQL 的 MVCC 是如何控制并发时的多版本。这个视图来自 BRUCE MOMJIAN 的一篇博客文章，他的博客地址：<http://momjian.us/>。

```
DROP VIEW IF EXISTS v_pageinspect;
CREATE VIEW v_pageinspect AS
SELECT  '(0,' || lp || ')' AS ctid,
        CASE lp_flags
          WHEN 0 THEN 'Unused'
          WHEN 1 THEN 'Normal'
          WHEN 2 THEN 'Redirect to ' || lp_off
          WHEN 3 THEN 'Dead'
        END,
        t_xmin::text::int8 AS xmin,
        t_xmax::text::int8 AS xmax,
        t_ctid
   FROM heap_page_items(get_raw_page('tbl_mvcc', 0))
  ORDER BY lp;
```

先看不考虑并发的情况：当 INSERT 数据时，事务会将 INSERT 的数据的 xmin 的值设置为当前事务的 xid，xmax 设置为 NULL，如下所示：

```
mydb=# BEGIN;
mydb=# SELECT txid_current();
txid_current
-----
      565
(1 row)
-- 当前的事务id为565
mydb=# INSERT INTO tbl_mvcc (ival) VALUES (1);
mydb=# SELECT * FROM v_pageinspect;
   ctid | case  | xmin  | xmax | t_ctid
-----+-----+-----+-----+
 (0,1) | Normal | 565  |    0 | (0,1)
(1 row)
mydb=# END;
-- 上一条INSERT语句插入的数据xmin的值设置为当前事务的xid, 565, xmax设置为NULL。
```

当 DELETE 数据时，将 xmax 的值设置为当前事务的 xid，如下所示：

```
mydb=# BEGIN;
BEGIN
mydb=# SELECT txid_current();
txid_current
-----
      561
(1 row)
```





```
mydb=# DELETE FROM tbl_mvcc WHERE id = 1;
DELETE 1
mydb=# SELECT * FROM v_pageinspect;
  ctid | case   | xmin  | xmax  | t_ctid
-----+-----+-----+-----+
 (0,1) | Normal | 565   | 566   | (0,1)
(1 row)

mydb=# END;
COMMIT
```

当 UPDATE 数据时，对于每个更新的行，首先 DELETE 原先的行，再执行 INSERT，如下所示：

```
mydb=# INSERT INTO tbl_mvcc (ival) VALUES (2); -- 先插入一条准备用来测试UPDATE的数据
mydb=# BEGIN;
mydb=# SELECT txid_current();
      txid_current
-----
       639
(1 row)
-- 当前的事务id为639
mydb=# SELECT * FROM tbl_mvcc;
     id | ival
-----+-----
      2 |    2
(1 row)
-- 现在tbl_mvcc中只有一行记录
mydb=# SELECT * FROM v_pageinspect;
  ctid | case   | xmin  | xmax  | t_ctid
-----+-----+-----+-----+
 (0,1) | Normal | 567   |    0   | (0,1)
(1 row)
-- 通过pageinspect查看page的内部，这条记录的xmin为当前事务的xid，也就是插入它的那个事务的
  xid，567。
mydb=# UPDATE tbl_mvcc SET ival = 20 WHERE id = 2;
-- 更新这条记录
mydb=# SELECT * FROM v_pageinspect;
  ctid | case   | xmin  | xmax  | t_ctid
-----+-----+-----+-----+
 (0,1) | Normal | 567   | 639   | (0,2)
 (0,2) | Normal | 639   |    0   | (0,2)
(2 rows)
mydb=# END;
```

通过 pageinspect 查看 page 的内部，可以看到 UPDATE 实际上是先 DELETE 前的数据，再 INSERT 一行新的数据，前面验证过插入这条数据的事务的 xid 为 567，可以看到 ctid 为 (0,1) 的这条记录的 xmin 为 567，xmax 等于当前事务的 xid：639，另外在这个 page 中多了一条 ctid 为 (0,2) 的记录，它的 xmin 等于当前事务的 xid：639。这时候数据库中就





存在两个版本了，一个是被 UPDATE 之前的那条数据，另外一个是 UPDATE 之后被重新插入的那条数据。

7.3.4 使用 pg_repack 解决表膨胀问题

尽管 PostgreSQL 的 MVCC 读不阻塞写，写不阻塞读，实现了高性能和高吞吐量，但也有它不足的地方。通过观察数据块的内部结构，我们已经了解到在 PostgreSQL 中数据采用堆表保存，并且 MVCC 的旧版本和新版本存储在同一个地方，如果更新大量数据，将会导致数据表的膨胀。例如一张一万条数据的表，如果对它进行一次全量的更新，根据 PostgreSQL 的 MVCC 的实现方式，在数据文件中每条数据实际会有两个版本存在，一个版本是更新之前的旧版本，一个版本是更新之后的新版本，这两个版本并存必然导致磁盘的使用率是实际数据的一倍，对性能也略有影响。

使用 VACUUM 命令或者 autovacuum 进程将旧版本的磁盘空间标记为可用，尽管 VACUUM 已经被实现得非常高效，但是没有办法把已经利用的磁盘空间释放给操作系统，VACUUM FULL 命令可以回收可用的磁盘空间，但它会阻塞所有其他的操作。

pg_repack 是一个可以在线重建表和索引的扩展。它会在数据库中建立一个和需要清理的目标表一样的临时表，将目标表中的数据 COPY 到临时表，并在临时表上建立与目标表一样的索引，然后通过重命名的方式用临时表替换目标表。

这个小工具使用非常简单，可以下载源码编译安装，也可以通过 yum 源安装，这里以通过 yum 源安装为例。首先安装 pg_repack，如下所示：

```
[root@pghost1 ~]# yum install -y pg_repack10
```

然后在数据库中创建 pg_repack 扩展，如下所示：

```
mydb=# CREATE EXTENSION pg_repack;
```

在命令行中，使用 pg_repack 对 tbl_mvcc 表进行重建，如下所示：

```
[postgres@pghost1 ~]# /usr/pgsql-10/bin/pg_repack -t tbl_mvcc -j 2 -D -k -h pghost1  
-U postgres -d mydb
```

可以使用定时任务的方式，定期对超过一定阈值的表和索引进行重建，达到给数据库瘦身的目的。PostgreSQL 全球开发组在接下来的一两个版本中，将对 MVCC 的实现方式作较大的改进，我们拭目以待。

7.3.5 支持事务的 DDL

PostgreSQL 事务的一个高级功能就是它能够通过预写日志设计来执行事务性的 DDL。也就是把 DDL 语句放在一个事务中，比如创建表、TRUNCATE 表等。举个创建表的例子，如下所示：



```

mydb=# DROP TABLE IF EXISTS tbl_test;
NOTICE: table "tbl_test" does not exist, skipping
DROP TABLE
mydb=# BEGIN;
BEGIN
mydb=# CREATE TABLE tbl_test (ival int);
CREATE TABLE
mydb=# INSERT INTO tbl_test VALUES (1);
INSERT 0 1
mydb=# ROLLBACK;
ROLLBACK
mydb=# SELECT * FROM tbl_test;
ERROR: relation "tbl_test" does not exist

```

再举个 TRUNCATE 的例子，如下所示：

```

mydb=# SELECT COUNT(*) FROM tbl_mvcc;
 count
-----
 9
(1 row)
mydb=# BEGIN;
BEGIN
mydb=# TRUNCATE tbl_mvcc ;
TRUNCATE TABLE
mydb=# ROLLBACK;
ROLLBACK
mydb=# SELECT COUNT(*) FROM tbl_mvcc;
 count
-----
 9
(1 row)

```

在上面的例子中，TRUNCATE 命令放在了一个事务中，但最后这个事务回滚了，表中的数据都完好无损。

7.4 本章小结

事务和多版本并发控制是数据库的两个非常重要的概念，本篇提到的内容只是冰山一角，在本章中我们讨论了数据库并发情况下可能发生的脏读、不可重复读和幻读现象以及事务的概念，通过几个例子演示了 PostgreSQL 中这几种读的现象。了解了如何开始一个事务，如何设置数据库、会话、单个事务的事务隔离级别，简单介绍了 PostgreSQL 事务隔离级别的特点，以及 PostgreSQL 如何通过隐藏列控制事务的可见性，并通过 pageinspect 扩展和一个视图观察了 MVCC 的内部信息，最后还介绍了 PostgreSQL 强大的支持事务的 DDL。

