**PostgreSQL针对ACID的实现机制（**<http://www.jasongj.com/sql/mvcc/>**）**

**数据库ACID**

数据库事务包含如下四个特性

* **原子性（Atomicity）** 指一个事务要么全部执行，要么不执行。也即一个事务不可能只执行一半就停止（哪怕是因为意外也不行）。比如从取款机取钱，这个事务可以分成两个步骤：1)划卡；2)出钱。不可能划了卡，而钱却没出来。这两步必须同时完成，或者同时不完成。
* **一致性（Consistency）** 事务的运行不可改变数据库中数据的一致性，事务必须将数据库中的数据从一个正确的状态带到另一个正确的状态。事务在开始时，完全可以假定数据库中的数据是处于正确（一致）状态的，而不必作过多验证（从而提升效率），同时也必须保证事务结束时数据库数据处于正确（一致）状态。例如，完整性约束了a+b=10，一个事务改变了a，那么b也应该随之改变。
* **隔离性（Isolation）** 在并发数据操作时，不同的事务拥有各自的数据空间，其操作不会对对方产生干扰。隔离性允许事务行为独立或隔离于其它事务并发运行。
* 持久性（Durability）事务执行成功以后，该事务对数据库所作的更改是持久的保存在数据库之中，不会无缘无故的回滚。

**ACID在PostgreSQL中的实现原理**

事务的实现原理可以解读为RDBMS采取何种技术确保事务的ACID特性,PostgreSQL针对ACID的实现技术如下表所示。

| **ACID** | **实现技术** |
| --- | --- |
| 原子性（Atomicity） | MVCC |
| 一致性（Consistency） | 约束（主键、外键等） |
| 隔离性 | MVCC |
| 持久性 | WAL |

从上表可以看到，PostgreSQL主要使用MVCC和WAL两项技术实现ACID特性。实际上，MVCC和WAL这两项技术都比较成熟，主流关系型数据库中都有相应的实现，但每个数据库中具体的实现方式往往存在较大的差异。本文将介绍PostgreSQL中的MVCC实现原理。

**PostgreSQL中的MVCC原理**

**事务ID**

在PostgreSQL中，每个事务都有一个唯一的事务ID，被称为XID。注意：除了被BEGIN - COMMIT/ROLLBACK包裹的一组语句会被当作一个事务对待外，不显示指定BEGIN - COMMIT/ROLLBACK的单条语句也是一个事务。

数据库中的事务ID递增。可通过txid\_current()函数获取当前事务的ID。

**隐藏多版本标记字段**

PostgreSQL中，对于每一行数据（称为一个tuple），包含有4个隐藏字段。这四个字段是隐藏的，但可直接访问。

* xmin 在创建（insert）记录（tuple）时，记录此值为插入tuple的事务ID
* xmax 默认值为0.在删除tuple时，记录此值
* cmin和cmax 标识在同一个事务中多个语句命令的序列值，从0开始，用于同一个事务中实现版本可见性判断

下面通过实验具体看看这些标记如何工作。在此之前，先创建测试表

|  |  |
| --- | --- |
| 1 2 3 4 5 | CREATE TABLE test  (  id INTEGER,  value TEXT ); |

开启一个事务，查询当前事务ID（值为3277），并插入一条数据，xmin为3277，与当前事务ID相等。符合上文所述——插入tuple时记录xmin，记录未被删除时xmax为0

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | postgres=> BEGIN; BEGIN postgres=> SELECT TXID\_CURRENT();  txid\_current  --------------  3277 (1 row)  postgres=> INSERT INTO test VALUES(1, 'a'); INSERT 0 1 postgres=> SELECT \*, xmin, xmax, cmin, cmax FROM test;  id | value | xmin | xmax | cmin | cmax  ----+-------+------+------+------+------  1 | a | 3277 | 0 | 0 | 0 (1 row) |

继续通过一条语句插入2条记录，xmin仍然为当前事务ID，即3277，xmax仍然为0，同时cmin和cmax为1，符合上文所述cmin/cmax在事务内随着所执行的语句递增。虽然此步骤插入了两条数据，但因为是在同一条语句中插入，故其cmin/cmax都为1，在上一条语句的基础上加一。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | INSERT INTO test VALUES(2, 'b'), (3, 'c'); INSERT 0 2 postgres=> SELECT \*, xmin, xmax, cmin, cmax FROM test;  id | value | xmin | xmax | cmin | cmax  ----+-------+------+------+------+------  1 | a | 3277 | 0 | 0 | 0  2 | b | 3277 | 0 | 1 | 1  3 | c | 3277 | 0 | 1 | 1 (3 rows) |

将id为1的记录的value字段更新为’d’，其xmin和xmax均未变，而cmin和cmax变为2，在上一条语句的基础之上增加一。此时提交事务。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | UPDATE test SET value = 'd' WHERE id = 1; UPDATE 1 postgres=> SELECT \*, xmin, xmax, cmin, cmax FROM test;  id | value | xmin | xmax | cmin | cmax  ----+-------+------+------+------+------  2 | b | 3277 | 0 | 1 | 1  3 | c | 3277 | 0 | 1 | 1  1 | d | 3277 | 0 | 2 | 2 (3 rows)  postgres=> COMMIT; COMMIT |

开启一个新事务，通过2条语句分别插入2条id为4和5的tuple。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | BEGIN; BEGIN postgres=> INSERT INTO test VALUES (4, 'x'); INSERT 0 1 postgres=> INSERT INTO test VALUES (5, 'y');  INSERT 0 1 postgres=> SELECT \*, xmin, xmax, cmin, cmax FROM test;  id | value | xmin | xmax | cmin | cmax  ----+-------+------+------+------+------  2 | b | 3277 | 0 | 1 | 1  3 | c | 3277 | 0 | 1 | 1  1 | d | 3277 | 0 | 2 | 2  4 | x | 3278 | 0 | 0 | 0  5 | y | 3278 | 0 | 1 | 1 (5 rows) |

此时，将id为2的tuple的value更新为’e’，其对应的cmin/cmax被设置为2，且其xmin被设置为当前事务ID，即3278

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | UPDATE test SET value = 'e' WHERE id = 2; UPDATE 1 postgres=> SELECT \*, xmin, xmax, cmin, cmax FROM test;  id | value | xmin | xmax | cmin | cmax  ----+-------+------+------+------+------  3 | c | 3277 | 0 | 1 | 1  1 | d | 3277 | 0 | 2 | 2  4 | x | 3278 | 0 | 0 | 0  5 | y | 3278 | 0 | 1 | 1  2 | e | 3278 | 0 | 2 | 2 |

在另外一个窗口中开启一个事务，可以发现id为2的tuple，xin仍然为3277，但其xmax被设置为3278，而cmin和cmax均为2。符合上文所述——若tuple被删除，则xmax被设置为删除tuple的事务的ID。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | BEGIN; BEGIN postgres=> SELECT \*, xmin, xmax, cmin, cmax FROM test;  id | value | xmin | xmax | cmin | cmax  ----+-------+------+------+------+------  2 | b | 3277 | 3278 | 2 | 2  3 | c | 3277 | 0 | 1 | 1  1 | d | 3277 | 0 | 2 | 2 (3 rows) |

这里有几点要注意

* 新旧窗口中id为2的tuple对应的value和xmin、xmax、cmin/cmax均不相同，实际上它们是该tuple的2个不同版本
* 在旧窗口中，更新之前，数据的顺序是2，3，1，4，5，更新后变为3，1，4，5，2。因为在PostgreSQL中更新实际上是将旧tuple标记为删除，并插入更新后的新数据，所以更新后id为2的tuple从原来最前面变成了最后面
* 在新窗口中，id为2的tuple仍然如旧窗口中更新之前一样，排在最前面。这是因为旧窗口中的事务未提交，更新对新窗口不可见，新窗口看到的仍然是旧版本的数据

提交旧窗口中的事务后，新旧窗口中看到数据完全一致——id为2的tuple排在了最后，xmin变为3278，xmax为0，cmin/cmax为2。前文定义中，xmin是tuple创建时的事务ID，并没有提及更新的事务ID，但因为PostgreSQL的更新操作并非真正更新数据，而是将旧数据标记为删除，并插入新数据，所以“更新的事务ID”也就是“创建记录的事务ID”。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | SELECT \*, xmin, xmax, cmin, cmax FROM test;  id | value | xmin | xmax | cmin | cmax  ----+-------+------+------+------+------  3 | c | 3277 | 0 | 1 | 1  1 | d | 3277 | 0 | 2 | 2  4 | x | 3278 | 0 | 0 | 0  5 | y | 3278 | 0 | 1 | 1  2 | e | 3278 | 0 | 2 | 2 (5 rows) |

**MVCC保证原子性**

原子性（Atomicity）指得是一个事务是一个不可分割的工作单位，事务中包括的所有操作要么都做，要么都不做。

对于插入操作，PostgreSQL会将当前事务ID存于xmin中。对于删除操作，其事务ID会存于xmax中。对于更新操作，PostgreSQL会将当前事务ID存于旧数据的xmax中，并存于新数据的xin中。换句话说，事务对增、删和改所操作的数据上都留有其事务ID，可以很方便的提交该批操作或者完全撤销操作，从而实现了事务的原子性。

**MVCC保证事物的隔离性**

隔离性（Isolation）指一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

标准SQL的事务隔离级别分为如下四个级别

| **隔离级别** | **脏读** | **不可重复读** | **幻读** |
| --- | --- | --- | --- |
| 未提交读（read uncommitted） | 可能 | 可能 | 可能 |
| 提交读（read committed） | 不可能 | 可能 | 可能 |
| 可重复读（repeatable read） | 不可能 | 不可能 | 可能 |
| 串行读（serializable） | 不可能 | 不可能 | 不可能 |

从上表中可以看出，从未提交读到串行读，要求越来越严格。

注意，SQL标准规定，具体数据库实现时，对于标准规定不允许发生的，绝不可发生；对于可能发生的，并不要求一定能发生。换句话说，具体数据库实现时，对应的隔离级别只可更严格，不可更宽松。

事实中，PostgreSQL实现了三种隔离级别——未提交读和提交读实际上都被实现为提交读。

下面将讨论提交读和可重复读的实现方式

**MVCC提交读**

提交读只可读取其它已提交事务的结果。PostgreSQL中通过pg\_clog来记录哪些事务已经被提交，哪些未被提交。具体实现方式将在下一篇文章《SQL优化（七） WAL PostgreSQL实现事务和高并发的重要技术》中讲述。

**MVCC可重复读**

相对于提交读，重复读要求在同一事务中，前后两次带条件查询所得到的结果集相同。实际中，PostgreSQL的实现更严格，不紧要求可重复读，还不允许出现幻读。它是通过只读取在当前事务开启之前已经提交的数据实现的。结合上文的四个隐藏系统字段来讲，PostgreSQL的可重复读是通过只读取xmin小于当前事务ID且已提交的事务的结果来实现的。

**PostgreSQL中的MVCC优势**

* 使用MVCC，读操作不会阻塞写，写操作也不会阻塞读，提高了并发访问下的性能
* 事务的回滚可立即完成，无论事务进行了多少操作
* 数据可以进行大量更新，不像MySQL和Innodb引擎和Oracle那样需要保证回滚段不会被耗尽

**PostgreSQL中的MVCC缺点**

**事务ID个数有限制**

事务ID由32位数保存，而事务ID递增，当事务ID用完时，会出现wraparound问题。

PostgreSQL通过VACUUM机制来解决该问题。对于事务ID，PostgreSQL有三个事务ID有特殊意义：

* 0代表invalid事务号
* 1代表bootstrap事务号
* 2代表frozon事务。frozon transaction id比任何事务都要老

可用的有效最小事务ID为3。VACUUM时将所有已提交的事务ID均设置为2，即frozon。之后所有的事务都比frozon事务新，因此VACUUM之前的所有已提交的数据都对之后的事务可见。PostgreSQL通过这种方式实现了事务ID的循环利用。

**大量过期数据占用磁盘并降低查询性能**

由于上文提到的，PostgreSQL更新数据并非真正更改记录值，而是通过将旧数据标记为删除，再插入新的数据来实现。对于更新或删除频繁的表，会累积大量过期数据，占用大量磁盘，并且由于需要扫描更多数据，使得查询性能降低。

PostgreSQL解决该问题的方式也是VACUUM机制。从释放磁盘的角度，VACUUM分为两种

* VACUUM 该操作并不要求获得排它锁，因此它可以和其它的读写表操作并行进行。同时它只是简单的将dead tuple对应的磁盘空间标记为可用状态，新的数据可以重用这部分磁盘空间。但是这部分磁盘并不会被真正释放，也即不会被交还给操作系统，因此不能被系统中其它程序所使用，并且可能会产生磁盘碎片。
* VACUUM FULL 需要获得排它锁，它通过“标记-复制”的方式将所有有效数据（非dead tuple）复制到新的磁盘文件中，并将原数据文件全部删除，并将未使用的磁盘空间还给操作系统，因此系统中其它进程可使用该空间，并且不会因此产生磁盘碎片。