基于 MVCC 的保存点设计与实现

陶能锋　叶晓俊
（清华大学软件学院　北京 100084）

摘　要　通过保存点实现部分回滚是事务理论中的一个重要方法，它能够极大地提高数据库管理系统的事务管理的灵活性和系统的性能。基于保存点的原理，本文提出了基于多版本并发控制下 DBMS 的事务保存点实现方法，并通过改进 PostgreSQL 进行了验证。最后给出保存点在工作流事务中的应用。

关键词　事务，保存点，MVCC

Design and Implementation of Savepoint Based on MVCC

TAO Neng-Feng　YE Xiao-Jun
(School of Software, Tsinghua University, Beijing 100084)

Abstract　Implementating partial rollback by savepoint is a fundament method in transaction theory, this method can improve flexibility of transaction management and system performance. This paper describes the way to realize savepoint based on multi-version concurrency control and presents a detailed description of how it is combined with PostgreSQL's concurrency control component. Furthermore, we discuss the application of savepoint in the work-flow transaction field.

Keywords　Transaction, Savepoint, MVCC

1　引言

传统数据库管理系统（DBMS）的事务是扁平事务[1]，它是一个操作序列，包含了一组数据库操作命令，组成一个不可分割的工作逻辑单元。事务的原子性要求“要么全做，要么全不做”。事务回滚是保证事务原子性的一种重要机制，它处理事务在执行过程中的失败语义（如事务执行过程中的程序异常、死锁处理[2]、用户干预等），撤销自从事务开始以来所产生的所有影响，实现失败语义下保证事务的“全不做”的性质。

事务的这种“要么全做，要么全不做”的特性既是优点也是缺点，一方面它大大简化了事务失败语义的处理；另一方面，“全不做”的处理逻辑缺乏灵活性，使得自事务开始以来所有的工作完全丢弃，造成资源浪费，特别是在时间运行比较长或资源占用率高的事务中，这种浪费尤为显著。

随着应用的发展，单纯的事务回滚（完全回滚）已经不能满足现实的需要，在这种情况下，人们便考虑在 DBMS 中提出部分回滚功能。部分回滚允许恢复到事务的某个执行时刻而不必全部回滚整个事务，极大地提高了数据处理的灵活性。利用保存点实现部分回滚是事务理论终的一种重要方法，已经写入了 SQL99 标准[3]。商业数据库管理系统（如 Oracle、Microsoft SQL server 等）已经提供了保存点和部分回滚的功能。

PostgreSQL 是目前功能最强大、特性最丰富和最复杂的开源对象关系数据库管理系统（ORDBMS）。2005 年发布的 PostgreSQL 8.0 也开始引入了保存点的概念[5]，但是它采用封闭嵌套事务[1]来实现，需要额外增加一个类似事务状态日志（Clog）[6]管理的模块，增加了内存占用量。此外，以嵌套事务来实现保存点的方式使得一个事务可以拥有多个事务 ID（XID），增加了访问事务状态日志（Clog）[6]以及获取父事务[1]信息而引起的磁盘 I/O 次数，增加了程序的响应时间。针对这一情况，本文提出了一种新的保存点实现方法，以妥善地解决上述所提到的问题。

本文的组织如下：第 2 节介绍保存点的原理；第 3 节简单介绍 PostgreSQL 的多版本并发控制机制；第 4 节详细讨论 MVCC 下保存点的设计与实现；最后讨论保存点的应用。

2　保存点原理

在扁平事务[1]模型下，事务以一种“单入口，单出口”的组织形式，顺序地执行所包含的读、写操作。事务只有在 Commit 或者 Abort 时才能确切地知道事务的最终执行结果。保存点则是在事务执行过程中增加一些“锚点”，或者是称之为标记（Mark），将事务分为多个“区间”，以实现在处理事务失败语义时不必强制回滚整个事务，而是根据需要回滚到事务执行过程中的某个特定位置，即实现事务部分回滚的功能。

一个保存点代表了事务执行过程中某个时刻的处理状态。它涉及事务处理中资源管理和数据处理两个方面。部分回滚既要恢复事务的执行状态，也要恢复数据库的状态[3]。其中数据库的状态指的是数据库系统中各个数据元素的状态，而事务执行状态是指主存数据结构，如锁，游标以及文件句柄等。对于数据库状态的恢复，目前普遍采用的是基于日志的方法，而对事务执行状态的恢复，则与具体的 DBMS 的事务管理系统密切相关。

3　PostgresSQL 的并发控制机制

PostgreSQL 的多版本并发控制使得读、写不相互排斥，提高了 PostgreSQL 的并发度，极大地提高了系统性能。在

陶能锋　硕士，研究方向：数据库事务处理；**叶晓俊**　副教授，主要研究领域：信息建模、数据仓库、企业信息移动、数据库理论等。

PostgreSQL 中产生写写冲突等 MVCC 无法解决的情况时，又结合了两阶段(2PL)封锁协议[4]。

PostgreSQL 采用的是一种非覆盖写的技术，物理上存储多个版本信息。文[6,7]成功地改造了 PostgreSQL 原有非覆盖写技术为覆盖写技术。对数据库数据进行修改时，不但记录用于保证事务持久性和原子性的事务日志，而且还另外记录一种专门记录数据前像(Before Image)的 Undo 数据条目，用于支持一致性读和多版本选择，并将这种 Undo 数据条目组织在一起，实现了基于回滚段[6,7]的多版本并发控制。基于文[6,7]工作，本文提出在 PostgreSQL 中实现保存点的实现方案。

PostgreSQL 中每个元组都有一个元组头，记录了一些必需的控制信息。多版本的选择以及元组的更新等都依赖于这个元组头结构。元组头结构记录修改(包括创建，更新和删除)该元组的事务、命令，以及在回滚段的位置等信息。增加回滚段[6,7]后，出于多版本并发控制的需要，修改了原有的元组头结构信息。如下：

```
typedef struct HeapTupleHeaderData
{
  union{
    TransactionId t_xid;      /* 事务 id */
    CommandId   t_cid;        /* 命令编号 */
    Roll_Ptr      t_rollptr;  /* 回滚段中位置 */
    }t_choice;
    ItemPointerData t_ctid;
    int16           t_natts;
    uint16          t_infomask;
.....
}HeapTupleHeaderData;
```

其中，t_infomask 记录了元组的相应状态。t_cid 记录了修改该元组的命令编号，即修改该元组的 SQL 命令在事务中的编号。根据这个编号，可以推算出修改该元组时事务所处的“区间”。

4 保存点的设计及实现

保存点仅在事务活动时有效(不考虑持久保存点[1]的情况)，与保存点相关的控制结构信息应当尽量保存在内存中而不是保存在固定存储中。基于这一点，可以利用 PostgreSQL 原有元组头结构中存储的 CommandID 字段来间接表示事务所处的“区间”，即修改该元组时，事务所处的保存点位置(指该保存点所管理的区间)。

保存点需要记录事务的处理状态，既包括事务的执行状态信息，也包括数据库的状态信息[3]。对数据库的状态信息是以元组的形式存放在固定存储(物理磁盘)上的，而事务的执行状态(锁、文件句柄等信息)则是存放在内存中的。在多版本并发控制模型下，对锁的管理大大简化，严格遵循 2PL 协议[4]的锁仅仅用于 MVCC 无法解决的情况下，如避免写写冲突等。

在事务处理过程中，如果需要对数据元素进行修改，则不但要记录一条用于保证事务的原子性和持久性的事务日志，而且还要记录用于读一致性的 Undo 数据条目。Undo 数据条目记录了被修改的数据元素的前像(Before Image)信息。属于同一事务的各个 Undo 数据条目都有一个编号(Undo 编号)来唯一表示该条目。所有 Undo 数据条目都存放在回滚段[7]中。

在没有保存点的情况下，整个事务处理过程中所申请的锁，文件句柄等资源，以及对数据元组的修改都同属一个事务，没有什么差别。当事务具有保存点后，事务所拥有的资源以及所做的修改虽然仍然属于同一个事务，但是需要体现出各保存点之间的区别，体现所处的事务“区间”，以实现事务的部分回滚。

每个事务都有唯一的一个事务状态结构，用于控制事务执行过程中的状态转换。此外，每个保存点都有对应的一个事务结构，用于将事务的执行过程分成多个不同的“区间”，如下：

```
typedef struct TransactionData
{
    SavepointNo savepointno;  /* 保存点编号 */
    uint64      roll_limit;   /* 回滚编号 */
    bool        iscommit;     /* 是否提交 */
    bool        isabort;      /* 是否回滚 */
    char *      name;         /* 保存点名称 */
    MemoryContext curTransactionContext;
    ResourceOwner curTransactionOwner;
    struct TransactionStateData * previous;
.....
} TransactionData;
```

在该结构中记录了保存点所属的事务，保存点名称，保存点编号，保存点开始时下一个修改操作记录 Undo 日志记录条目时所用的 Undo 编号等信息。事务结构中的 Undo 编号用于记录事务部分回滚时的截至点。

当事务进行部分回滚时，首先先前事务处理过程中所产生的 Undo 日志信息取消指定保存点后事务对数据库状态的改变，恢复数据库状态，然后释放该保存点以及该保存点之后的所有保存点所表示的事务“区间”所获取的所有资源，恢复事务的执行状态。

当显式释放保存点(对应 Release 命令)时，从当前保存点开始，依次将当前保存点后所申请的资源等交由其前驱保存点管理，直到目标保存点被处理。

当隐式释放保存点(事务 Commit)时，则最后一个保存点开始，依次释放在该保存点所表示的事务“区间”所获得的所有资源。

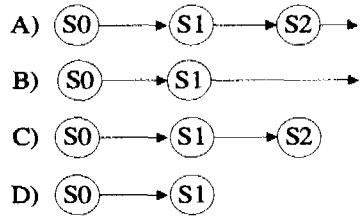

图 1 保存点的建立、释放以及回滚

当事务开始时，隐式地开始一个保存点。图 1 中 a)表示事务处理过程中先后显式定义了两个保存点 S1，S2；b)表示释放保存点 S2；c)表示回滚到保存点 S2；d)表示回滚到保存点 S1。回滚到指定保存点，该保存点之后事务对数据库所产生的影响都将消除，该保存点开始之后所申请的所有资源都将释放，但是对应该保存点的事务结构并不删除，以实现对同一保存点的多次回滚。

实现部分回滚的另一个重要功能是为了尽快地解决事务之间的阻塞。在 PostgreSQL 原有版本中，每个事务开始时都向锁表中插入一个事务锁单元，该单元由事务的事务 ID 唯一表示。增加事务的保存点之后，事务锁单元由事务 ID 以及保存点编号唯一表示。保存点编号是一个单调递增的整数，从 0 开始。即使在事务处理过程中释放了某个保存点，该保存点所使用的编号在当前事务中将不会再被使用。

在 PostgreSQL 中，每个 Backend 后台进程(或称 Postgres 进程)代表一个事务，具有一个进程结构。事务之间的交互，如事务的等待和事务阻塞后的重新唤醒等，都是通过事务结构进行的。在进程结构中，记录了当前事务的一个保存点链表(在进程私有空间中)。保存点结构如下：

```
struct Savepoint
{
SavepointNo          savepointno;  /* 保存点编号 */
CommandId            commandid;    /* 命令编号 */
struct Savepoint *   previous;     /* 前驱保存点结构 */
};
```

其中 savepointno 是一个单调递增的整数编号，commandid 表示定义该保存点时的命令编号。

当定义一个保存点时(事务开始时隐式产生的保存点除外)，就在链表尾添加一个节点；释放保存点时，则将该保存点节点及其之后的所有节点都删除；当事务 Commit 或者 Abort 时，则释放所有保存点节点。由保存点编号和事务的命令编号来决定事务所处的阶段。

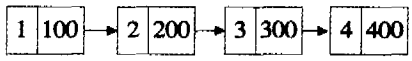

图 2　进程结构中的保存点链表

事务读取元组时，首先根据元组头中记录的事务 ID 取到对应的进程结构，然后利用记录的命令 ID，从保存点链表的链尾开始扫描，找到第一个小于或等于指定命令 ID 的节点。从该节点记录中得到保存点编号，从而确定所要等待的事务锁单元。如图 2 所示，假设某个事务取得了特定事务所修改的元组，其元组头中记录的命令 ID 为 256。从保存点链表的链尾开始扫描，找到第一个小于该命令 ID 的节点，从该节点得到保存点编号，从而可以得知该元组是在事务建立编号为 2 的保存点之后，建立编号为 3 的保存点之前修改的，根据事务 ID 和保存点编号确定锁单元。如果发生冲突而导致事务阻塞，则在等待的事务回滚到编号为 2 或者更前的保存点时，或者事务 Abort 或提交时，该冲突就可以解除。

5　保存点应用

本文提出的保存点方案已经在 PostgreSQL 上得到实现并得到应用验证。保存点的使用主要有 3 个方面的好处：①支持长事务，避免回滚整个事务，从而减少损失；②降低死锁开销；③提高系统处理的灵活性，为终端用户提供更友好的界面。针对第三点应用，我们结合国家自然科学基金项目基于过程度量和挖掘的适应性工作流管理系统关键技术研究，对事务工作流部分进行了实现。

事务工作流(或称工作流事务)[8]由为完成某个商业目的而预先定义的多个活动组成，活动之间存在先后和优先关系[9]。事务工作流比传统的扁平事务提供了更加丰富的语义。事务工作流兼有事务和工作流的特点，其涉及到多个自治的、分布的、异构的数据源。

在处理工作流事务的失败语义时，保存点的思想同样也可以值得借鉴，只不过需要考虑到工作流事务访问多个数据源的问题。为恢复到同一事务工作流里面的较早一个状态，可以首先使用补偿事务[9]来取消已经执行成功的活动的影响，恢复数据源状态；然后使用保存点来恢复事务的执行状态，从而达到恢复的目的。至于怎样表示补偿事务以及协调事务工作流中各个活动，还有待进一步研究。

结束语　通过保存点实现部分回滚能够极大地提高数据库管理系统的事务管理的灵活性和系统的性能。基于保存点的原理，本文提出了基于多版本并发控制下 DBMS 的事务保存点实现方法，通过改进 PostgreSQL 进行了验证。相比 PostgreSQL8.0 中的保存点实现方法，本文提出的保存点实现方法无论是从要求的内存数量，还是由多版本并发控制而引起的磁盘 I/O 次数，或是具体实现所需要的代码量都得到了很大的改进。

保存点用于恢复到同一事务里面的较早的一个状态，主要考虑恢复数据源的状态以及恢复事务的执行状态，这种思想可以应用到所有的“单层”事务中。论文最后还讨论了应用保存点所带来的好处以及保存点在工作流事务中的应用。

参 考 文 献

1　Gray J, Reuter A. Transaction Processing: Concepts and Techniques. Corrected Second Printing. Morgan Kanfmann, 1993

2　Fussell D, Kedem Z M, Silberschatz A. Deadlock removal using partial rollback in database systems. In: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, 1981

3　Kim S H, Jung M S, Park J H, et al. A Design and Implementation of Savepoints and Partial Rollbacks Considering Transaction Isolation Levels of SQL2. In: Proceedings of the Sixth International Conference on Database Systems for Advanced Applications, Hsinchu, Taiwan, 1999

4　Eswaran K P, Gray J N, Lorie R A, et al. The Notions of Consistency and Predicate Locks in a Database System. Communications of the ACM, 1976, 19(11): 624～633

5　PostgreSQL8.0. 2005

6　萧美阳. 基于回滚段的多版本并发控制研究与实现:[硕士论文]. 清华大学, 2005

7　陈足先. 基于回滚段的日志管理器研究与实现:[硕士论文]. 清华大学, 2005

8　Alonso G, Agrawal D, El Abbadi A, et al. Advanced Transaction Models in Workflow Contexts. In: Proceedings of the Twelfth International Conference, New Orleans, Louisiana, 1996

9　Schuldt H, Alonso G, Beeri C, et al. In: Atomicity and Isolation for Transactional Processes. ACM Transactions on Database Systems, 2002, 27(1): 63～116