自修复数据库系统日志机制研究

谢美意 朱 虹 冯玉才
(华中科技大学计算机科学与技术学院 武汉 430074)

摘 要 选择性恢复使得一个自修复数据库系统在受到恶意攻击后,只需撤销历史中受到恶意事务感染的那部分操作,无需回滚整段历史,但要求日志机制支持对事务间依赖关系的追踪及前像数据的长期保存。通过分析传统日志机制的不足以及现有原型系统实现方法存在的问题,提出了一种新的日志结构。该日志包含事务依赖信息,并以前像表代替传统日志机制中的回滚段。给出了基于该日志结构的数据库恢复方法,并在时间和空间开销方面对本方法与其它方法进行了分析和比较。

关键词 自修复,选择性恢复,事务依赖,前像表
中图法分类号 TP311.13 **文献标识码** A

Research on Logging Mechanism of Self-healing Database System

XIE Mei-yi ZHU Hong FENG Yu-cai
(College of Computer Science & Technology, Huazhong University of Science and Technology, Wuhan 430074, China)

Abstract Selective recovery allows a self-healing database system to undo only those operations affected by malicious transactions instead of rollbacking the whole history after malicious attacks, but this function relies on the logging mechanism's support on inter-transaction dependency tracking and longtime preservation of before image data. The inadequacy of traditional logging mechanisms and the problems of the methods used in current prototypes were analyzed. Following this, a new logging structure was proposed which contains inter-transaction dependency information and replaces the rollback segments in traditional logging mechanisms with the before image tables. Database recovery approaches based on the new mechanism were also presented. At last, we analyzed the performance and space overhead of our method, and compared it with other methods.

Keywords Self-healing, Selective recovery, Inter-transaction dependency, Before image table

1 引言

自修复数据库[1]要求系统在发现入侵事务之后,能够自动清除入侵事务对数据库造成的影响。由于入侵事务被发现的时候通常已经提交,它对数据库的错误修改可能已被后续的事务读取并由此导致了新的错误操作,因此仅仅撤销入侵事务显然是不够的。而如果将整个数据库回滚到入侵发生之前的状态,则意味着从入侵发生到入侵发现之间的所有工作都不得不重做,这将带来巨大的工作量和难以避免的服务中断。因此,自修复数据库研究要解决的一个重要问题就是,如何在确保数据库得到正确修复的前提下尽可能保留已完成的工作,即只撤销受到入侵感染的不可信事务而不撤销未受影响的正常事务,这种恢复方式被称为选择性恢复[2]。

自修复数据库系统的实现原理如图1所示。用户通过执行数据库事务对数据库中的数据进行操作,与这些操作相关的日志及审计信息也同时保存在数据库中;系统运行过程中,数据库中的审计信息被不断送往入侵检测模块进行分析,若发现其中有恶意事务,则向受损评估模块发出入侵警报;受损评估模块接到报警后,根据日志中隐含或显式记录的操作依赖关系查找被恶意事务感染的事务,得到需撤销的事务集合(Undo Transaction Set,简称UTS);受损修复模块负责构造并执行补偿事务,该补偿事务等价于UTS中所有事务操作按其执行时序颠倒的逆操作,从而消除恶意事务对数据库所造成的影响。

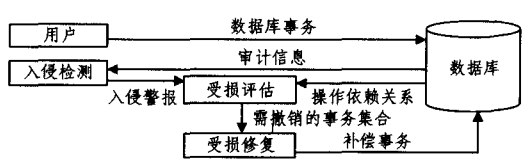

图1 自修复数据库系统体系结构图

自修复数据库系统对日志有以下要求:

(1)日志中应包含能反映事务依赖关系的信息。

(2)日志中应包含前像数据,即Update或Delete命令所修改的数据项的旧值。

(3)允许事务依赖关系信息及数据前像在日志中有较长的保留期限。

到稿日期:2009-05-12 返修日期:2009-07-22 本文受863高技术研究发展计划基金项目(2006AA01Z430)资助。
谢美意 女,讲师,主要研究方向为数据库及自管理技术等,E-mail:xiemeiyi@sina.com;朱 虹 女,博士,博士生导师,主要研究方向为数据库理论与安全等;冯玉才 男,教授,博士生导师,主要研究方向为数据库、信息技术等。

(4)日志所采用的结构应便于自修复算法利用。

2 自修复数据库日志研究现状分析

2.1 传统的日志机制的不足

传统日志机制[3]主要服务于事务故障、系统故障及介质故障的恢复,因此对于支持自修复数据库系统中特有的选择性恢复难免存在着一些不足。

首先,传统日志不包含任何与读操作有关的信息。虽然读操作导致了事务间的逻辑关联,但传统故障恢复要么恢复单个事务,要么恢复故障发生时尚未提交或未及写盘的所有事务,不需要考虑事务间的逻辑关系。这一点为选择性恢复中事务依赖关系的确定带来了困难。

其次,传统故障恢复的原则是保证事务的 ACID 特性,更明确地说,即事务一旦夭折,则必须撤销其所有操作;事务一旦提交,则必须保证其对数据库的更新永不丢失。这个特点使得传统日志中不一定需要保留事务修改操作对象的前像,即使保留了,在事务提交后也没有必要永久地保留下去,只需要保存最近一次备份以来的后像数据就足够了。

最后,流行 DBMS 的日志采用的结构与现有自修复算法中设计的日志结构存在相当大的差异,不利于实现高效可靠的自修复。

2.2 现有原型系统采用的方法及问题

ITDB[4,8]是一个较具代表性的自修复数据库原型系统,它的设计思想是通过一个工具来提供自修复功能。该工具借助通用编程接口在 DBMS 的上层实现,自修复过程使用的日志实际上是该工具在数据库中维护的两张表,称为读日志表和写日志表。这种实现方式的优点是无需重新设计 DBMS 日志且具有较好的可移植性,而缺点在于 DBMS 本身的日志完全没有得到利用,因此系统中实际上保留了两份内容大致重复的日志,导致较大的性能开销。

针对读日志表的维护开销过大的问题,文献[5]中提出了一种替代的方法。该方法事先静态分析应用中各类事务的轮廓(profile),并从中提取每类事务的读集合模板。需要得到某个具体事务的读信息时,将该事务的输入参数代入相应的模板即可求出该事务的读集合。这种方法的代价比前一种小,但得到的结果不一定准确,因此可能会有无辜事务被误当成可疑事务,导致不必要的撤销和重做。

文献[6]介绍的 Phoenix 是在开源数据库管理系统 PostgreSQL 的基础上实现的一个自修复 DBMS 原型系统。PostgreSQL 采用了多版本并发控制机制,元组的所有旧版本都保留在数据表中。Phoenix 巧妙地利用这些旧版本信息进行修复,无需额外维护一个前像日志,使得其性能表现明显优于其它原型系统。Phoenix 的问题在于它所依赖的多版本并发控制机制并不流行,目前主流的 DBMS 一般采用基于封锁的并发控制机制,因此该方法不易推广。

Phoenix 的研究者在文献[7]中介绍了他们新设计的另一个原型系统 Blastema。Blastema 采用了与 ITDB 类似的体系结构,但其研究重点在于如何准确高效地追踪事务依赖关系,并未涉及数据库修复过程,因此其日志机制的设计尚不完整。

3 一种新的日志机制

传统日志缺乏自修复功能所需的一些必要信息,而为自修复功能额外维护一份日志的开销过高。考虑到这两份日志的内容存在很大程度的重复,我们的解决思路是将传统日志与自修复原型系统中使用的日志融合起来,形成一份能二者兼顾的日志,避免不必要的重复开销。为此我们设计了一组新的日志结构,由事务依赖日志、前像表日志和重做日志 3 部分构成。

3.1 日志结构

3.1.1 事务依赖日志

事务依赖的相关定义如下[3,9]:

定义 1 一个事务 T_i 是具有偏序关系 $<_i$ 的偏序集,满足:

(1)$T_i \subseteq \{(r_i[x], w_i[x] \mid x \in DB) \cup \{a_i, c_i\}$;

(2)当且仅当 $c_i \notin T_i$ 时,$a_i \in T_i$;

(3)如果 t 为 c_i 或 a_i,则对于 T_i 中的任意其它操作 p,有 $t <_i p$;

(4)如果 $r_i[x], w_i[x] \in T_i$,则有 $r_i[x] <_i w_i[x]$ 或者 $w_i[x] <_i r_i[x]$。

其中 r, w, a, c 分别代表读、写、中断和提交操作。

定义 2 令 $T=\{T_1, T_2, \cdots, T_n\}$ 表示一组事务集合,一个事务执行历史 H 是 T 上具有偏序关系 $<_H$ 的偏序集,满足:

(1)$H = \bigcup_{i=1}^{n} T_i$;

(2)$\bigcup_{i=1}^{n} <_i \subseteq <_H$;

(3)如果有两个冲突操作 $p, q \in H$,则有 $p <_H q$ 或者 $q <_H p$。

定义 3 在事务执行历史 H 中,如果存在数据项 x 使得事务 T_i 与事务 T_j 满足下列条件,则称 T_j 依赖于 T_i:

(1) $w_i[x] <_H r_j[x]$;

(2) $c_i \in H$;

(3) 在事务 T_i 与 T_j 之间不存在任何提交事务 T_k,使得 $w_i[x] <_H w_k[x]$ 且 $w_k[x] <_H r_j[x]$。

为方便讨论,给出以下假设:

(1)事务 T 中所有显式或隐式的查询请求已经被分解为若干对单个数据项的读操作;

(2)读写操作的对象,即数据项的粒度为元组级;

(3)写操作只包括插入操作和删除操作两种类型(修改操作被视为一个删除操作后面紧跟着一个插入操作);

(4)系统采用不低于读提交级别的事务隔离级。

记录事务依赖关系有两种方式:一种是在日志中记录事务的读操作信息,当数据库需要修复时,通过扫描及分析日志间接地得到读写依赖关系;另一种是在系统处理读操作的过程中即时确定事务依赖关系,并将依赖关系信息显式地保存在日志中。本文采取第二种方式,以一个事务依赖日志记录事务间的依赖关系。

为了能在运行时追踪事务依赖关系,扩充了元组数据结构,如图 2 所示。

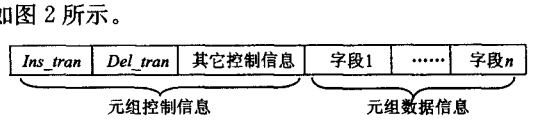

图 2 扩充的元组数据结构

在元组的控制信息中增加了两个属性 *Ins_tran* 和 *Del_tran*,分别表示插入该元组的事务 ID 和删除该元组的事务 ID。对每个活动事务 T_i,系统维护一个初值为空的集合

DTS_i。对于事务 T_i 执行的每个读操作 $r_i[x]$，令 $DTS_i = DTS_i \cup \{x.Ins\_tran\}$。

引理 1 在对同一个数据项的插入操作和读操作之间不存在已提交的写操作。

证明（反证法）：假设事务执行历史 H 上存在偏序 $w_l[x] <_H w_k[x]$ 和 $w_k[x] <_H r_j[x]$，其中 $w_l[x]$ 是一个插入操作，$w_k[x]$ 是一个已提交的写操作。由假设(3)可知，对于系统中的任意一个数据项 x 的写操作只可能有一个插入操作和一个删除操作，因此 $w_k[x]$ 只可能是对 x 的删除操作。但是，如果 $w_k[x]$ 已经提交，则数据项 x 已不在数据库中，因而 $r_j[x]$ 不可能存在。这与假设矛盾，故命题得证。

定理 1 当事务 T_i 提交时，DTS_i 是 T_i 所依赖的全部事务的 ID 的集合。

证明：首先，对于 DTS_i 中的任一元素 d，d 表示 T_i 读的某元组 x 的插入事务的 ID。由假设(4)可保证事务 T_d 不是一个夭折事务，由引理 1 可知 $w_d[x]$ 与 $r_j[x]$ 之间不可能存在其它已提交的写操作，因此根据事务依赖的定义可知事务 T_i 依赖于 T_d（正确性）；其次，对于 T_i 中所有的读操作 $r_i[x]$，都有 $x.Ins\_tran \in DTS_i$（完全性）。综上所述，可知 DTS_i 是 T_i 所依赖的全部事务的 ID 的集合。（证毕）

事务依赖日志由事务开始记录、事务依赖记录和事务回滚记录构成。事务开始时，系统在日志中写入一条开始记录；如果事务夭折，则在日志中写入一条回滚记录；如果事务正常提交，则将 Dep_tran_set 的内容构造为一条事务依赖记录写入日志。事务依赖记录的结构如下：

$(Rec\_type, TranID, Dep\_tran\_num, Dep\_tranID_1, \cdots, Dep\_tranID_n)$

其中，Rec_type 表示记录类型，$TranID$ 表示某事务的 ID，Dep_tran_num 表示该事务所依赖的事务数，Dep_tranID_k 表示被依赖事务的 ID。

3.1.2 前像表日志

事务的恢复可分为撤销（Undo）和重做（Redo）两种情况。Undo 时需要使用前像数据，Redo 时需要使用后像数据。由于前像数据在传统恢复机制中不需要长期保存，因此在日志实现上，前像数据与后像数据一般是分开存放的。常见的前像日志有回滚段[10]等形式。在本方法中，为了方便采用统一的方式进行恢复处理，我们使用了一种新颖的结构来保存前像数据，称为前像表日志。

前像表日志由若干个前像表构成，前像表与基表间存在一一对应关系，即对于每个基表 R，在前像表日志中都存在且仅存在一个与其对应的前像表 R^{bi}。R^{bi} 用于保存从 R 中删除的数据项，其元组结构与 R 完全相同，以保证从 R 中删除的元组无需经过任何格式变更就可以直接插入到 R^{bi} 中。R 与 R^{bi} 的不同之处在于：

(1) R^{bi} 上没有完整性约束条件。对于前像表中的数据，无需考虑其值的正确性，并且基表上的某些完整性约束在前像表中是无法满足的（如主码约束，前像表中可能保存拥有同一元组的多个前像值）；

(2) R^{bi} 的存储结构与 R 不同。前像表的用途在于恢复，而恢复是以事务为单位的，因此对前像表的查询多以 Del_tran 属性为关键字。在设计前像表的存储结构时需要重点考虑的是如何提高此类查询的性能。

当事务 T 对基表 R 执行 *Delete* 或 *Update* 命令时，对于 R 中将要被删除或修改的每条元组 r，都有一个对应的删除操作。操作流程如图 3 所示，其中实线框中为新增的步骤。图 3 中的条件判断是基于这样的理由：如果 r 是由 T 插入的，则恢复时 r 原本就应该被丢弃，不需要将 r 重新插入 R，因此也就没有保留 r 的必要。

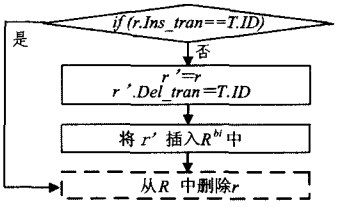

图 3 删除操作流程图

3.1.3 重做日志

重做日志用于记录对数据库的修改的后像数据，这里所指的对数据库的修改包括对①数据文件、②事务依赖日志文件以及③前像表日志文件的修改。重做日志的作用是在系统崩溃后将数据库恢复到故障前一刻的状态，因此写重做日志必须遵循 WAL（write-ahead logging）原则，即对上述 3 类文件的修改在写盘之前，必须先将相应修改的日志记录写入重做日志，并且对于同一个事务，系统保证其日志记录写入的顺序与其修改操作的顺序一致。

本方法中的重做日志采用物理日志形式，日志中记录的修改以数据页为单位，记录结构如下：

$(PageID, offset, len, data)$

其中，$PageID$ 为数据页的标识信息，$offset$ 为被修改数据在页内的偏移，len 为被修改数据的长度，$data$ 为数据被修改后的内容。

为提高 I/O 效率，事务的重做日志信息先放在日志缓冲区中，只有在下列时机才会被真正写入磁盘：

(1)事务提交；

(2)数据页刷盘；

(3)检查点；

(4)日志缓冲区写满。

3.2 基于本日志的恢复机制原理

3.2.1 事务故障的恢复

发生事务故障时，要求撤销该事务对数据库的所有操作，包括删除该事务插入的数据和重新插入该事务删除的数据。由于元组结构扩充后，每条元组中已经包含了插入事务和删除事务的信息，且被删除的数据都已保存在前像表中，因此恢复过程可简述如下。

输入：发生故障的事务 T

输出：撤销了 T 的所有操作后的数据库

步骤：

```
for（事务 T 修改过的每个基表 R）{
    /* 删除事务 T 插入的所有元组 */
    删除 R 中所有 Ins_tran=T.ID 的元组；
    /* 恢复事务 T 删除的所有元组 */
    将 R 的前像表 R^bi 中所有 Del_tran=T.ID 的元组重新插入到 R 中；
}
```

3.2.2 系统故障及介质故障的恢复

发生系统故障时，内存中的数据全部丢失。系统重启后，

首先确定恢复的起点，一般是故障前的最后一个检查点。根据重做日志，重写数据库自最后一个检查点以来对数据库所做的所有修改，将数据库恢复到故障时刻的状态，但并不一定是一个一致性状态，还需要将该时刻尚未提交的事务全部回滚，方法如下。

输入：无

输出：撤销了所有未提交事务后的数据库

步骤：

```
初始化回滚事务集合 S 为空；
for (事务依赖日志中的每条记录 t) {
    if ( t.Rec_type==开始记录类型)
        S=S∪{t.TranID};
    if (t.Rec_type==回滚记录类型 or
      t.Rec_type==事务依赖记录类型)
        S=S-{t.TranID};
}
for ( 数据库中的每个基表 R ) {
    /* 删除 S 中所有事务插入的元组 */
    删除 R 中所有 r.Ins_tran∈S 的元组 r;
    /* 恢复 S 中所有事务删除的元组 */
    将 R 的前像表 R^bi 中所有 r.Del_tran∈S 且 r.Ins_tran∉S
    的元组 r 重新插入 R;
}
```

需要说明的是，对于 S 中的两个事务 T_1 和 T_2，如果在 R^{bi} 中存在元组 r 是由 T_1 插入且又被 T_2 删除，则恢复时不应再将 r 插入到 R 中。

发生介质故障时，系统先利用备份数据将数据库还原到最后一次备份时的状态，然后再利用此后备份的日志进行恢复。利用日志恢复的原理与系统故障恢复的原理相同，在此不再赘述。

3.2.3 选择性恢复

当系统检测到恶意事务或发现错误操作时，选择性恢复可以以最小的代价清除其对数据库带来的影响。对于入侵检测组件报告的恶意事务 B，根据事务依赖日志确定所有依赖于该事务的不可信事务集合并进行撤销，方法如下。

输入：恶意事务 B

输出：撤销了 B 及依赖于 B 的所有事务后的数据库

步骤：

```
初始化回滚事务集合 S={B};
    在事务依赖日志中定位事务 B 的开始记录 t_b;
    for (t_b 后的每条记录 r) {
    if (t.Rec_type==事务依赖记录类型 ) {
        for (i=1;i<=t.Dep_tran_num;i++)
            if (t.Dep_tranID_i∈S) {
                S=S∪{t.TranID};
                break;
            }
        }
    }
    for (数据库中的每个基表 R) {
        删除 R 中所有 r.Ins_tran∈S 的元组 r;
        将 R 的前像表 R^bi 中所有 r.Del_tran∈S 且 r.Ins_tran∉S
        的元组 r 重新插入 R;
    }
```

4 性能分析和比较

4.1 空间开销

与传统日志机制相比，本方法在空间上增加的开销主要在两个方面。

(1) 元组中新增 *Ins_tran* 和 *Del_tran* 属性的空间开销。以 TPCC 测试为例，在传统模式下，装载 50 个仓库后数据库的大小约为 3438Mb，元组数约为 25,000,000 条。按每个事务 ID 占用 4 个字节来计算，则在本方法下，每个元组长度将增加 8 个字节，数据库的大小将增加 200Mb，空间开销增加约 5.8%。

(2) 事务依赖日志的空间开销。假设 DBMS 平均每秒钟处理 200 个事务，每个事务平均依赖于 10 个事务，则事务依赖记录的大小平均为 46 个字节，每日的空间开销为 $200\times3600\times24\times46\approx758$Mb。假设应用系统设置的事务依赖信息的保存期限为 2 周，则系统需要 10.4Gb 的固定额外空间来存放事务依赖日志，这对一般系统来说都是可以承受的。

前像表日志与一般系统中回滚段等机制的功能类似，都是用于记录前像数据。对于同一个事务，前像表与回滚段的空间开销大致相当，因此对于相同的前像保存期限要求，本方法与传统方法的空间开销是大致相当的。

与 ITDB 原型系统所采用的方法相比，本方法不需要保存读操作信息，也不在传统机制之外另外保存一个写操作日志，因此空间开销明显减小。

4.2 时间开销

从两个方面考虑本方法的时间性能：一个是正常运行时的性能，一个是恢复时的性能。

系统正常运行时，与传统日志机制相比，本方法需要：①维护 *Ins_tran* 和 *Del_tran* 字段；②在读操作处理过程中分析并保存事务依赖关系。与两个典型原型系统 Phoenix 和 ITDB 相比，Phoenix 的方法由于利用了 PostgreSQL 特有的元组结构，因此只有②的开销。考虑到①的开销很小(只有两个简单的赋值命令)，本方法的性能开销只略高于 Phoenix 方法。而 ITDB 方法需要对所有查询进行动态重写，并额外维护一个读写日志，且这些功能由于实现在应用层而导致了大量信息交互，因此其性能显然低于本方法。

系统恢复时，对于传统故障类型，现有方法与传统方法并无区别，而本方法则做了较大改变。从恢复算法来看，恢复性能主要受到前像表查询性能的影响。当前像表规模较大时，查询效率降低，恢复性能也会随之下降，二者呈线性关系。但由于恢复算法对前像表的查询条件非常简单，通过设计恰当的存储结构及内存缓冲机制，完全可以将查询性能控制在合理范围之内。对于选择性恢复，与 ITDB 方法相比，本方法在恢复过程中不需要分析读写操作间依赖关系，可以一次回滚多个事务在同一张表上的操作并减少其间不必要的重复操作，因此具有较大的性能优势。

结束语 选择性恢复是自修复数据库研究中提出的一种新的数据库恢复技术。不同于传统的面向故障的恢复方式，选择性恢复针对的是恶意攻击及内部误用等人为因素导致的数据库损坏，该技术的研究对于提高信息系统在日益复杂的应用环境中的可生存性具有重要的意义。

(下转第 166 页)

同隐含层规模的 BP 神经网络算法分别进行了 130 次独立实验，并对实验结果进行了统计分析，结果如图 4 所示。

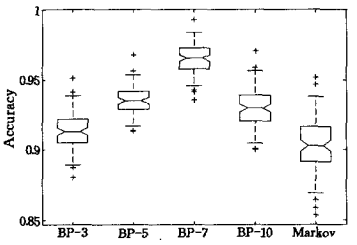

图 4　基于测试数据的仿真试验结果

从图 4 中可以看出，不论采用何种隐含层规模的 BP 神经网络，其最终预测精度均高于单纯的马尔科夫方法，其中 4 种神经网络所达到的平均预测精度分别为 91.4%，93.5%，96.3%以及 92.9%，而马尔科夫方法的平均预测精度为 90.3%。以上结果充分证明了神经网络方法在软件抗衰中的性能优势。

此外，我们可以看到在 4 种不同隐含层规模的神经网络中，规模为 7 的神经网络达到了最优性能。这是因为如果隐含层神经元过少，网络的学习性能不足，从而网络权值将无法达到最优；而如果隐含层神经元过多，网络的学习性能将达到"过饱和"状态，这将导致网络过早地陷入局部最优状态。以上的实验结果为我们选取最优的隐含层规模提供了参考依据。

结束语　从本文可以看出，运行中的软件系统状态的马尔科夫模型可由 MBP 神经网络来表示并求解。利用本文提出的 MBP 神经网络对马尔科夫预测模型进行不断训练以达到期望值，利用此方法进行软件系统衰退预测，可以克服单纯使用马尔科夫模型时预测不够准确的问题，解决以往方法所面临的一些困难，它是一种值得深入研究的软件系统衰退预测方法。

参 考 文 献

[1] Avritzer A, Weyuker E J. Monitoring smoothly degrading systems for increased dependability [J]. Empirical Software Eng, 1997,12(1):55-77

[2] Garg S, Puliafito A, Telek M, et al. Analysis of preventive maintenance[J]. IEEE Trans on Computers, 1998,47(1):96-107

[3] Huang Y, Kintala C, Kolettis N, et al. Software rejuvenation: analysis module and applications[C]// IEEE Intl. Symposium on Fault Tolerant Computing. 1995:381-390

[4] 王田. SMSC 负荷状态检测方法研究[J]. 电子科技大学学报, 2003,32(2)

[5] 徐建，张坤，刘玉凤. 软件抗衰研究综述[J]. 小型微型计算机系统，2007,28(11)

[6] 阎平凡，张长水. 人工神经网络与模拟进化计算[M]. 北京：清华大学出版社，2005:26-30

[7] 戴葵，仇广煜，胡守仁. 一种基于离散马尔科夫模型的神经网络可靠性设计方法[J]. 计算机工程与科学，1999,21(3)

[8] 周成容. BP 神经网络的模糊改进及应用[J]. 重庆工学院学报：自然科学版，2008,22(6):153-156,158

(上接第 142 页)

选择性恢复功能根据事务间的依赖关系来决定需要撤销的事务，以便最大限度地保留无辜事务所作的工作，这对传统日志机制提出了新的需求。现有自修复数据库原型系统的实现方法在性能及通用性等方面存在着各种问题，本文在分析了现有方法不足的基础上，提出了一种新的日志机制，该机制中增加了事务依赖日志，并以前像表日志代替传统的回滚段，使系统能以统一的方式兼顾传统恢复功能和选择性恢复功能。性能分析表明，本方法在时间性能和空间性能上都优于现有的 ITDB 方法。与基于 MVCC 机制的 Phoenix 方法相比，本方法适用于主流的基于封锁机制的 DBMS，因而具有更好的通用性。

参 考 文 献

[1] Liu P, Jing J. Architectures for Self-healing Databases under Cyber Attacks[J]. International Journal of Computer Science and Network Security, 2006,6(1B):204-216

[2] Smirnov A, Chiueh T. A Portable Implementation Framework for Intrusion-Resilient Database Management Systems[C]// Proc. of the 2004 International Conference on Dependable Systems and Networks (DSN'04). 2004:443-452

[3] Bernstein P A, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems[M]. Reading, MA: Addison-Wesley, 1987

[4] Luenam P, Liu P. ODAM: An On-the-fly Damage Assessment and Repair system for Commercial Database Applications[C]// Proc. 15th IFIP WG 11.3 Working Conference on Database and Application Security. July 2003

[5] Amman P, Jajodia S, Liu P. Recovery from Malicious Transactions[J]. IEEE Transactions on Knowledge and Data Engineering, 2002,14(5):1167-1185

[6] Chiueh T, Pilania D. Design, Implementation, and Evaluation of A Repairable Database Management System[C]// Proc. of the 21st International Conference on Data Engineering (ACSAC 04). 2005:1024-1035

[7] Chiueh T, Bajpai S. Accurate and Efficient Inter-transaction Dependency Tracking[C]// Proc. of the 2008 IEEE 24th International Conference on Data Engineering (ICDE 08). 2008:1209-1218

[8] Wang H, Liu P, Li L. Evaluating the survivability of Intrusion Tolerant Database systems and the impact of intrusion detection deficiencies[J]. International Journal of Information and Computer Security, 2007,1(3):315-340

[9] Liu P, Amman P, Jajodia S. Rewriting Histories: Recovering from Malicious Transactions[J]. Distributed and Parallel Databases, 2000,8(1):7-40

[10] Oracle9i LogMiner[EB/OL]. http://otn.oracle.com/products/oracle9i/daily/oct25.html