Open Nested Transactions*

4 R − 3

Shinji Yasuzawa and Makoto Takizawa †
Dept. of Computer Science and Systems Engineering
Tokyo Denki University ‡

1 Introduction

In order to develop new applications like *groupware* and *CAD*, transactions are nested, i.e. composed of modules. In these applications, transactions manipulate more objects for longer time than the conventional transactions. Strict *2PL* scheme is used in the conventional system. Although it implies serializability and no cascading abort, transactions hold objects for longer time. In this paper, we present new synchronization schemes named *open nested* ones by which objects obtained can be released during the transaction execution.

In section 2, we present our system model. In section 3, synchronization mechanism is presented. In section 4, we discuss how to abort transactions by using compensate transactions.

2 System Model

A system M is composed of multiple *objects*. Each object o provides two kinds of operations, *primitive* and *public* operations to manipulate o. Users issue the public operations to o in order to manipulate o. Each public operation is realized by a sequence of public operations on another objects and primitive operations on o. The primitive operations manipulate directly o but do not invoke another operations. A transaction is an atomic sequence of operation invocations. Since each operation may invoke another operations, transactions are represented in a tree, i.e. *nested*. In the transaction tree, the leaves, non-leaves, the root denote primitive operations, public operations, and the transaction, respectively.

Suppose that an operation op on an object o invokes n (≥ 1) operations op_1, ..., op_n in this order [Fig.1]. The execution of op is represented as a sequence $\langle$ [op, op_1, ..., op_n op] $\rangle$, where [op and op] denote the *begin* and *end* of op, respectively.

For two operations op_1 and op_2 in M, two transitve relations $\rightarrow$ and $\Rightarrow$ are defined as follows. $op_1 \rightarrow op_2$ if op_1 invokes op_2. $op_1 \Rightarrow op_2$ if (1)$op_1 \rightarrow op_2$, (2) $op_1 \rightarrow op_3$, and op_2 and op_3 belong to the same object, or (3) $op_3 \rightarrow op_2$, and op_1 and op_3 belong to the same object. M is *hierarchical* iff for every public operation op of every object o, all the operations which invoke op belong to the same object and not $op \Rightarrow op$.

For each object o, a compatibility relation among the public operations is defined on the basis of the semantics of o. Let op_1 and op_2 be operations of o, and op_i invoke op_{i1}, ..., op_{ik_i} (i=1, 2). If op_1 and op_2 are compatible, any interleaving of op_{1j} and op_{2k} are allowed. This is semantic serializability [6].

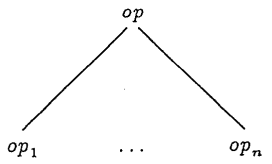

Figure 1: Nested transaction

3 Synchronization

Let op be an operation on an object o, and invoke op_1, ..., op_n where each op_i is an operation on an object o_i (if op_i is public). Before executing op on o, o has to be locked in a mode of op as follows.

[Locking scheme] (1) o is locked by op. [op is a lock operation on o.
(2) op_1, ..., op_n are executed, i.e. before executing each op_i, o_i is tried to be locked if op_i is public. □

One problem is how to release locks obtained in op. One way to release objects is a famous *two-phase locking* (*2PL*) scheme. In the *strict 2PL*, when all the operations in T complete, i.e. T commits, all the locks obtained in T are released as presented in (3-1). The execution scheme is named a *closed* nested transaction [4]. It implies serializability and no cascading abort.

(3-1) When op_1, ..., op_n commit, if op is the root, all the locks obtaind in op are released.

Another way is to release objects before T commits. There are two ways to realize op], i.e. when op_1, ..., op_n commit,

(3-2) all the locks obtained by op_i ..., op_k are released.
(3-3) o is released.

In (3-2), it is noted that o is not released even if o_1, ..., o_n obtained by op_1, ..., op_n are released. On the other hand, all the locks obtained in op are released when op commits in (3-3). If op is an operation like *print*, the printer can be released after printing out. Thus, (3-3) can be used to execute operations which cannot be recovered.

*Open Nested Transactions
†Shinji Yasuzawa and Makoto Takizawa
‡Tokyo Denki University

Transactions obeying (3-2) and (3-3) are *partially* and *totally open*, respectively. In this paper, the partially open nested transaction are considered.

[Theorem] Every schedule obtained from partially open transactions is *semantically serializable* in a hierarchical system.
[Proof] Suppose that op_i on o invokes op_{ij} on o_{ij} (i=1,2, j=1, ..., k_i). If op_1 and op_2 are compatible, any interleaving sequence of op_{ij} is allowed. Hence, let us consider a case that op_1 and op_2 are incompatible. Suppose that op_1 is first executed and then op_2. While op_1 holds o, op_2 cannot use o. Hence, op_{11}, ..., op_{1k_1} are executed before op_{21}, ..., op_{2k_2}. □

4 Compensation

In order to abort an operation op on o, a compensate operation $\bar{op}$ is invoked. $\bar{op}$ is an operation on o such that for every system state s, $\bar{op}(op(s)) = op(\bar{op}(s))$. Let p = $\langle op_1, \ldots, op_n \rangle$ be an execution sequence. p is compensated by $\langle \bar{op}_n, \ldots, \bar{op}_1 \rangle$. In the partially open transactions, when an operation op commits, objects obtained by op_1, ..., op_n are released. Hence, the transaction T is represented as a sequence $\langle$ [T, op_1, ..., op_{m-1}, [op_m, op_{m1}, ..., [op_{mk},..., [$op_{mk...h}$) where T invokes op_1, ..., op_m, op_m invokes op_{m1}, ... op_{mk}, op_{mk} invokes ... $op_{mk...h}$, and $op_{mk...h}$ is the current operation. The operations without [like op_1 means ones which commit. In order to abort T, the compensate sequence $\langle$ [$\bar{op}_{m...h}$, ..., $\bar{op}_{m1}$, [$\bar{op}_m$, $\bar{op}_{m-1}$, ..., $\bar{op}_1$, [$\bar{T}$ $\rangle$ is executed.

5 Deadlock

Since compensate operations might require objects which are not held by the transaction, deadlock might occur by executing them. *Uncompensatable deadlock* in the close nested transactions is discussed in [4]. A system state is represented in a well-known *wait-for* graph. The wait-for graph is extended to include the precedence relation among operations in each transaction. An operation op_1 depends on op_2 ($op_1 \mapsto op_2$) *iff* (1) op_1 waits for op_2, (2) op_1 precedes op_2 in the same transaction, or (3) for some op_3, $op_1 \mapsto op_3 \mapsto op_2$. An *extended wait-for* (*EWF*) graph is a directed graph whose nodes represent operations, and whose directed edge from op_1 to op_2 denotes $op_1 \mapsto op_2$. op is *deadlocked iff* op is included in a directed cycle, i.e. $op \mapsto op$.

Suppose that a transaction T is compensated by executing $\bar{c}$ in Fig.2. Suppose that $\bar{c}$ invokes f and f waits for w. Further, U is compensated by $\bar{w}$, $\bar{w}$ invokes y, and y waits for b. This is deadlock. Suppose that U is selected and $\bar{w}$ is compensated. Then, w is compensated again by $\bar{w}$. The same deadlock state is obtained. This is *uncompensatable deadlock*.

[Theorem] No uncompensatble deadlock occurs in the hierarchical system.

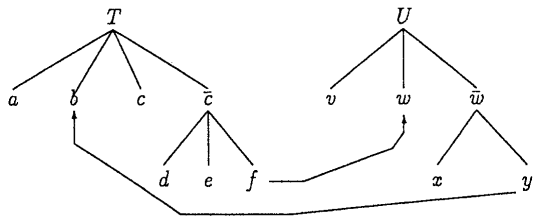

Figure 2: Uncompensatable deadlock

[Proof] In Fig.2, f never waits for operations v and w. If so, f and w are at the same level, y and b are at the same level, y is lower than w, and f is lower than b. It is contradiction. □

6 Concluding Remarks

In this paper, we have discussed new execution schemes of nested transactions on multiple objects, i.e. *partially* and *totally open* transactions. Since the objects can be released before the transaction commits, more concurrency can be obtained.
Further problem is how to resolve the cascading abort[1, 2].

References

[1] Garcia-Molina, H. and Salem, K., "Sagas," *Proc. of the ACM SIGMOD*, 1987, pp.249-259.

[2] Korth, H. F., Levy,E., and Silberschalz, A., "A Formal Approach to Recovery by Compensating transactions," *Proc. of the VLDB*, 1990, pp.95-106.

[3] Moss, J. E., "Nested Transactions: An Approach to Reliable Distributed Computing," *The MIT Press Series in Information Systems*, 1985.

[4] Takizawa, M. and Deen, S, M., "Lock Mode Based Resolution of Uncompensatable Deadlock in Compensating Nested Transaction," *Proc. of the 2nd Far-East Workshop on Future Database Systems*, 1992, pp.168-175.

[5] Traiger, I. L., "Trends in System Aspects of Database Management," *Proc. of the 2nd International Conf. on Database (ICOD-2)*, 1983, pp.1-21.

[6] Weihl, W. E., "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types," *ACM Trans. on Programming Language and Systems*, Vol.11, No.2, 1989, pp.249-283.