A Multi-Version Concurrency Scheme With No Rollbacks

Abraham Silberschatz
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

The multi-version data item concept is a method for increasing concurrency in a database system. All previously proposed schemes utilizing this concept relied on transaction rollback as a means for preserving consistency. These rollbacks require a considerable amount of overhead which degrades performance. In this paper we develop a new scheme that utilizes the multi-version data item concept. Our proposed scheme ensures consistency without the use of rollbacks. It is based upon the non-two phase tree locking protocol previously proposed by Silberschatz and Kedem.

1.0 INTRODUCTION

Designers of large database systems have long realized that concurrent processing of multiple user transactions can decrease the response time of queries and updates. If one assumes that each of these transactions maintains the consistency of the database when run alone, then any database system which supports concurrent processing must guarantee that any allowable execution scenario involving a number of transactions will also maintain the consistency of the data. A system that guarantees this property is said to ensure serializability [1].

In order to ensure serializability, the system must control the interactions of the transactions executing in the database. This control may be achieved through such methods as locking protocols [2], atomic actions [3], optimistic concurrency control schemes [4], and time-stamp ordering [5]. All these schemes have a goal in common -- to increase the level of concurrency allowed in the system.

One method for increasing concurrency that has recently gained popularity, is the use of the multi-version data item concept. This concept has been used as early as 1973, in a version of the Honeywell FMS system [6]. It was formalized by Stearns et al. [7] in 1976, and was the nucleus of Reed's atomic action scheme [3]. Recently Bayer et al. [8,9] and Stearns and Rosenkrantz [10] have presented various rules for ensuring consistency for systems using the multi-version concept.

All previous work uses transaction rollback as a means for preserving consistency and deadlock freedom. These rollbacks require a considerable amount of overhead, and therefore degrade performance of the system. This performance cost has been considered acceptable since existing concurrent database systems have typically had at most only a few transactions concurrently active. With the advent of new hardware technologies and the evolution of computing environments consisting of networks of local machines accessing global databases and other resources, the amount of concurrency in a typical database system can be...
expected to rise dramatically. In such an environment, the use of rollbacks as a means for preserving consistency will become more burdensome.

It is our aim here to develop a scheme that will take advantage of the availability of multi-version data items, and at the same time will ensure consistency without the use of rollbacks. Our proposed scheme is based upon the non-two phase Tree protocol previously proposed by Silberschatz and Kedem [11,12].

2.0 LOCKING PROTOCOLS

Solutions to the problem of ensuring serializability have been based almost exclusively on strategy of dividing the database into units or entities, access to which may be controlled by a database concurrency control. The most common model for such a system involves the notion of a locking protocol. Each transaction which executes in the system must lock an entity before it wishes to access that entity, and unlock the entity when it no longer needs to access it. A locking protocol may thus be viewed as a set of rules defining the allowable sequences of locks and unlocks which may appear in a transaction. A transaction may hold either an exclusive (X) or a shared (S) lock on a data item. An X mode lock on a data item permits the transaction holding that lock to read and modify the item, while an S mode lock permits only the reading of the item.

Given a set of locking modes, we can define the compatibility relation among them as follows. Suppose that transaction \( T_i \) requests a lock of mode \( A \) on data item \( E \) on which transaction \( T_j \) currently holds a lock of mode \( B \). If \( T_i \)'s request can be immediately granted in spite of the presence of the mode \( B \) lock, then we say mode \( A \) is compatible with mode \( B \), denoted by \( \text{COMP}(A,B) = \text{true} \). Traditionally, the compatibility relation among the S and X modes of locking is defined to be \( \text{COMP}(A,B) = \text{true} \) if and only if \( A = B = S \).

The first useful locking protocol developed was the two-phase locking protocol [1], which states that a transaction is not allowed to lock a data item after it has unlocked any other item. Eswaran et al. [1] have shown that for systems which have no restrictions on the order in which entities may be locked, it is necessary and sufficient that all transactions be two-phase in order for all concurrent scenarios to be serializable. The two-phase protocol has two drawbacks. First, it severely restricts the amount of concurrency allowed in a system; second, it is not deadlock free.

Silberschatz and Kedem [11] have shown that if one has a priori knowledge as to how the entities of the database are organized, one may be able to design locking protocols that ensure serializability and freedom from deadlock and which are not two-phase. Since then a number of new non-two-phase locking protocols were developed which potentially allow more concurrency than do two-phase protocols [12-17]. In this paper we restrict our attention to databases that are organized (logically or physically) as rooted trees. We note that our results can be extended to rooted DAGs as well.

We begin by describing the basic non-two-phase locking protocols for rooted trees, called the tree protocol. When a transaction enters the database it arbitrarily selects the first entity in the tree to be locked. Subsequently, the transaction may lock an entity only if its father is currently locked by that transaction, and that entity has not yet been locked. A transaction may unlock an entity at any point in time.

The tree protocol described in [11] was defined for systems employing only X mode locks. In a subsequent paper [12] Kedem and Silberschatz have extended that protocol to support both X and S modes of locking. The new protocol can be summarized as follows:

- **a)** The transactions accessing the database are classified into two types:
  - **Update** -- Those that can issue only X mode lock requests.
  - **Read-only** -- Those that can issue only S mode lock requests.

- **b)** The rules of the protocol are:
1. Update transactions must start by locking the root of the tree first. Read-only transactions may lock any vertex first.

2. A transaction may lock any subsequent vertex $v$ only if it is holding a lock on the father of $v$.

3. A vertex may be unlocked at any point in time.

In the following we extend the tree protocol so that it can be applied to database systems in which every data item has a multi-version associated with it.

3.0 THE MULTI-VERSION TREE PROTOCOL

Let the database be organized in the form of a rooted tree where each vertex in the tree is a data item. As in the extended tree protocol, the transactions accessing the database are classified into Read-only and Update transactions. The Read-only transactions may only issue S mode lock requests, and the Update transactions may only issue X mode lock requests.

The compatibility among the Read-only and Update transactions depends upon which version of a data item a transaction needs to read, and this is determined by a time stamp scheme. Each Update transaction $U_i$ is given a unique time stamp, denoted by $TS_i$. The time stamp is assigned to $U_i$ immediately after $U_i$ has successfully locked the root. The time stamps may be generated in a variety of ways, as long as they satisfy the condition that $TS_i$ is greater than the time stamps of all other update transactions that have succeeded in locking the root.

When an Update transaction $U_i$ writes onto data item $E$, a new version of $E$ is created; the version has time stamp $TS_i$ associated with it. Thus with each data item $E$ a sequence of versions are associated, ordered by their time stamp. A version of $E$ created by transaction $U_i$ is added to the sequence only and immediately after $U_i$ has unlocked $E$. In the sequel we denote a version of entity $E$ with time stamp $k$, by $e_k$.

With each Read-only transaction $R_i$ we associate a time-stamp value, denoted by $Max_i$. The value of $Max_i$ is obtained when $R_i$ successfully locks its first entity, say $E$, with the sequence $<e_1, ..., e_k>$, and is computed to be $k$.

The availability of multi-version data items allows us to relax the restrictions on the compatibility relation among the S and X modes of locking. This in turn will allow us to gain greater levels of concurrency in the system. Let us now extend the compatibility relation as follows:

Let $T_i$ be a transaction requesting lock mode $A$ on data item $E$ that is currently being locked in mode $B$ by transaction $T_j$, then $COMP(A, B)$ is defined as follows:

- $COMP(X, X) = false$
- $COMP(S, S) = true$
- $COMP(X, S) = true$
- $COMP(S, X) = true <\Rightarrow Max_i < TS_j$

Note that the delay time may be decreased because of the new, more lenient definition of $COMP(X, S)$ and $COMP(S, X)$.

Finally, we note that when a Read-only transaction $R_i$ tries to lock its first data item $E$, the value of $Max_i$ is undefined. Thus if $E$ is currently being held in X mode, $R_i$ can either immediately lock $E$ or wait until $E$ is unlocked. Either option can be used in the protocol defined below; the only difference will be in the value $Max_i$ will be assigned.

We can now extend the tree protocol with the multi-version data item scheme. The new protocol, called Multi-Version tree (MVT) protocol, can be summarized as follows:

1. Each transaction must follow the rules of the Tree protocol.

2. A Read-only transaction $R_i$ that has successfully locked entity $E$, must read version $e_q$ such that $q$ is the largest time-stamp $\leq Max_i$. 

218
3. An Update transaction $U_i$ that has successfully locked entity $E$, must read version $e_q$ such that $q$ is the largest time-stamp $\leq TS_i$.

We will prove that MVT assures serializability and deadlock freedom. In order to do so we must first introduce some standard definitions which will be required in the subsequent technical discussions.

**Definition 1**: A history $H$ is the trace, in chronological order, of the concurrent execution of a set of transactions $T = \{T_0, \ldots, T_{n-1}\}$.

**Definition 2**: We define a precedence relation $\rightarrow$ on a history $H$ by writing $T_i \rightarrow T_j$ if and only if there exists an entity $E$, accessed (i.e., read or written) by $T_i$ and $T_j$, such that either one of the following holds:

a) $T_j$ created version $e_n$ and $T_i$ read or created version $e_m$, $m < n$.

b) $T_j$ read version $e_n$ and $T_i$ created version $e_m$, $m \leq n$.

We shall say that $T_i$ and $T_j$ interact in the system if they are related via the $\rightarrow$ relation.

**Lemma 1**: The MVT protocol assures serializability if and only if all allowable concurrent executions of transactions produce an acyclic $\rightarrow$ relation.

**Proof**: We only note that a relation is acyclic if and only if it admits a consistent enumeration (topological sort), namely can be embedded in a linear order.

**Lemma 2**: Let $E$ be a data-item locked by the Update transactions $U_i$ and $U_j$. If $TS_i < TS_j$, then $U_j$ successfully locked $E$ only after $U_i$ unlocked $E$.

**Proof**: Since $TS_i < TS_j$ it follows that $U_i$ must have locked the root before $T_j$ did. Since $COMP(X, X) = false$, and since the transactions follow the original tree protocol, the result follows.

**Lemma 3**: Let $E$ be a data item locked by the Read-only transaction $R_j$, and the Update transaction $U_i$. If $TS_i \leq MAX_j$, then $R_j$ successfully locked $E$ only after $U_i$ unlocked $E$.

**Proof**: By induction on $q$, the distance of $E$ from first node locked by transaction $R_j$.

$q = 0$: Let $\langle e_1, e_2, \ldots, e_k \rangle$ be the sequence when $R_j$ successfully locked $E$. By our scheme $MAX_j = k$. Clearly $k \geq TS_i$. If $k = TS_i$ then the result follows from the fact that $e_k$ is inserted into the sequence only after $U_i$ unlocked $E$. If $k > TS_i$ then let $U_m$ be the transaction that created $e_k$. Since $TS_m > TS_i$, by Lemma 2 it follows that $U_m$ locked $E$ only after $U_i$ unlocked $E$, and the result follows.

$q > 0$: Let $F(E)$ denote the father of $E$. From the definition of the tree protocol, $U_i$ can lock $E$ only when holding a lock on $F(E)$. By the induction hypothesis, $U_i$ successfully locked $F(E)$ only after $U_i$ unlocked it, at which time $U_i$ has a lock on $E$, or has unlocked it. Since $COMP(S, X) = false$ if $MAX_j < TS_i$, the result follows.

**Lemma 4**: Let $U_i$ and $U_j$ be update transactions that interact in the system. Then $U_i \rightarrow U_j \iff TS_i < TS_j$.

**Proof**: $\Rightarrow$: From the definition of the $\rightarrow$ relation.

$\Leftarrow$: Let $E$ be any data item locked by $U_i$ and $U_j$. From Lemma 2, $U_j$ successfully locked $E$ only after $U_i$ unlocked it. If $U_j$ created a version of $E$, the result immediately follows from the definition of $\rightarrow$. If $U_j$ created a version of $E$, it was only after $U_i$ read a version of $E$ and hence result follows.

**Lemma 5**: Let $U_i$ and $R_j$ be update and read-only transactions respectively that interact in the system. Then $U_i \rightarrow R_j \iff TS_i < MAX_j$.

**Proof**: $\Rightarrow$: By the definition of MVT, $R_j$ can only read a version $e_k$ of data item $E$ if $k \leq MAX_j$.

$\Leftarrow$: Let $E$ be any data item written by $U_i$ and read by $R_j$. By Lemma 3, $R_j$ locked only after $U_i$ unlocked. Since $MAX_j > TS_i$, $R_j$ must have read version $e_m$, $m > TS_i$ and the result follows.
Lemma 6: Let \( R_i \) and \( U_j \) be read-only and update transaction respectively that interact in the system. Then \( R_i \rightarrow U_j \iff \text{Max}_i < \text{TS}_j \).

Proof:

\( \Rightarrow \): Assume by contradiction that \( \text{Max}_i > \text{TS}_j \). By Lemma 3, any data-item read by \( R_i \) and written by \( U_j \) must have been successfully locked by \( R_i \) only after \( U_j \) unlocked \( E \). By the rules of MVT, \( R_i \) must read version \( \text{ek} \), \( k > \text{TS}_j \) and hence \( U_i \rightarrow R_j \) which is a contradiction.

\( \Leftarrow \): By the definition of MVT, \( R_i \) can only read a version \( \text{ek} \) of data item \( E \) if \( k \leq \text{Max}_i \).

Theorem 1: The MVT protocol assures serializability.

Proof: The proof follows directly from Lemmas 1, 4, 5 and 6. The serializability order corresponds to the time-stamp ordering of the various transactions in the system, where the Read-only transactions are after the write transactions of the same number.

Theorem 2: The MVT protocol assures deadlock freedom.

Proof: The proof follows from the fact that each transaction must follow the Tree protocol, which assures deadlock freedom, and the fact that the compatibility relation between the S and X mode of locking for MVT has fewer delay conditions (i.e., an update transaction can never be delayed by a Read-only transaction) than the one defined for the Tree protocol.

4.0 DISCUSSION

In this section we compare our scheme with other previously proposed schemes and briefly discuss the question of implementation in a distributed environment.

4.1 Comparison

How does our locking scheme compare with other previously published schemes utilizing the multi-version data item concept? As the reader might have already guessed, there is no simple answer. This is due to the fact that we are dealing with a diverse set of mechanisms which do not seem to have a common ground for comparison. Nevertheless, let us try to argue in favor of our scheme. In the sequel we refer to Reed's scheme [3] as A1, Bayer's et al. scheme [8, 9] as A2, and Stearns and Rosenkrantz scheme [10] as A3.

a) Reading a data item -- All the schemes except A2 may require a transaction that issues a read request to wait. In A1 and A3 a transaction must wait until the needed version is committed; this occurs when the transaction that has created this version has terminated. In contrast, in our scheme a transaction must only wait until the data item has been unlocked. Since we use a non-two-phase protocol this can occur before the transaction has terminated. In A3 the reading of a data item may also result in the rollback of one of the transactions in the system.

In A2, reading of a data item always succeeds. However, in order to determine which version is to be read, a cycle detection algorithm must be performed (this is done every time a read request is made). Such a detection algorithm is required in order to determine whether serializability has been violated, or a deadlock has occurred. Such a detection algorithm is quite expensive, especially in a distributed environment. If a cycle is found rollback is required.

b) Writing a data item -- In all the schemes except ours, creating a new version may result in the rollback of one of the transactions. In A1, a write request is either immediately granted or the transaction is rolled back. In A2, a write is either immediately granted (if no conflict occurs), or a cycle detection algorithm is invoked; if a cycle is found then the transaction must be rolled back. In A3, three possibilities exist: (1) the
request is immediately granted; (2) the
transaction is delayed until the time the
transaction with which it has conflicted
terminates; (3) rollback of one of the
conflicting transactions is carried out.

In our scheme, a write is either
immediately carried out, or the transaction
is delayed until the data item is unlocked.
No rollbacks are necessary.

c) Termination -- In all the schemes except A2
termination does not require any special
handling. In A2 however, a cycle detection
algorithm must be performed when the
transaction terminates. Again if a cycle
is found rollback is required.

d) Inconsistency and deadlock -- In order to
detect inconsistency of data and possible
deadlocks, each of the above schemes uses a
different strategy. In A1 time stamp
ordering is used. In A2 a dependency graph
must be maintained, and a cycle detection
algorithm must be frequently invoked. In
A3 time stamps are used and additional
control messages must be sent in order to
inform transactions about various states.
In our scheme, the time stamps and tree
ordering of data items is used. Use of
precedence in locking implies that
additional data items may have to be
locked.

To summarize, our scheme has the main advantage
that no rollbacks are required in order to
guarantee serializability and deadlock freedom, and
it has a simple and efficient implementation. The
main disadvantage is that additional data items may
have to be locked. However, the fact that this is
a non-two phase locking protocol reduces the amount
of time each data item is actually locked.
Moreover, the new compatibility relation between
the S and X lock modes in our model are quite
lenient, resulting in less delay time.

4.2 Implementation

The MVT protocol can be utilized in either a
centralized system or a distributed system. The
implementation in a centralized system is
straightforward, simple and efficient. This is due
to the fact that locking decisions are made locally
and deadlock problems do not arise. In a
distributed environment two types of systems need
to be considered.

a) Non-replicated databases -- in this types
of system each data item resides in one and
only one site. A transaction in such an
environment may be viewed as locus of
control, migrating from one site to
another, carrying with it the relevant
information needed to perform its
designated task. In each site a
transaction accesses the data item residing
there. In such an environment the
implementation of the MVT protocol is
identical to the centralized system. Most
importantly the MVT protocol does not
require the presence of deadlock detection
algorithms that are either expensive
(searching for cycles in a wait-for graph),
or require transaction rollback even if no
deadlocks occur (e.g., the kill/die scheme
of Rosenkrantz et al. [18]).

b) Replicated databases -- in this type of
system a data item may be replicated in
several sites. A transaction thus must
access a data item in its site, or if this
data item is not available at that site,
the transaction must request a copy of that
item from any of the sites in which it is
replicated. In such an environment, our
locking scheme must be modified. The
simplest way of doing so is for each data
item to designate one of the replicated
sites as the owner of that item [9]. Thus
each data item has one and only one owner.
With this scheme our locking protocol
simply requires that locking should be done
on the owner's copy, resulting in a minimal
amount of overhead in terms of locking.
However the reading and writing of a data item is carried out at the local site (if it is replicated there).

Hence the MVT protocol can be efficiently implemented in both a centralized and distributed environment. An implementation issue that we have not discussed thus far is the question of when a particular version can be discarded. To answer this we need to define the following terms:

1. Let \( k \) denote the largest time stamp in the multi-version sequence associated with data item \( E \).
2. Let \( \text{MIN} \) denote the time stamp such that \( \text{MIN} < k \) for all data items \( E \) in the system.

It can be easily shown that any version with a time stamp less than \( \text{MIN} \) can be safely discarded provided that every currently active Read-only transaction has \( \text{MAX}_i > \text{MIN} \).

There are a variety of ways to implement the above scheme. One simple method is to simply discard a version only when no Read-only transactions are currently active in the system. A more effective method is for the system to maintain two counters \( C_0 \) and \( C_1 \) and an index \( J \) initialized to 0. When a Read-only transaction \( R_i \) enters the system, counter \( C_j \) is incremented by one, and the value of \( J \) (0 or 1) is retained by \( R_i \) in location \( J' \). When \( R_i \) leaves the system it decrements \( C_j \) by one. When the system decides on discarding the old versions, the following steps are taken:

1. determine the value of \( \text{MIN} \),
2. \( T := J \)
3. \( J := (J + 1) \mod 2 \),
4. when \( C_j = 0 \), then discard all the versions with time stamp \( < \text{MIN} \).

Note that this algorithm can be reexecuted only after step 4 has been completed; that is, after \( C_j = 0 \) and some of the old versions have been possibly discarded.

5.0 CONCLUSION

We have presented a new multi-version concurrency control scheme called MVT. Our scheme has the main advantage that no rollbacks are required in order to guarantee serializability and deadlock freedom in both non-replicated and replicated distributed systems. The main disadvantage of our scheme is that additional data items may have to be locked. We note however that since we use a non-two-phase protocol, the amount of time each data item is actually being locked is reduced. Moreover, the new compatibility relation among the new \( S \) and \( X \) modes of locking is quite lenient, which decreases delay time. Finally we note that our results can be extended to locking protocols for databases that are organized as rooted DAGs. This however is beyond the scope of this paper.

Acknowledgment

I wish to thank Gael Buckley for her helpful comments concerning the material in this paper.

References


