A FAMILY OF MULTI-VERSION LOCKING PROTOCOLS WITH NO ROLLBACKS

Abraham Silberschatz
Gael N. Buckley

Department of Computer Sciences
University of Texas at Austin
Austin, Texas

TR-218 March 1983

A FAMILY OF MULTI-VERSION LOCKING PROTOCOLS WITH NO ROLLBACKS[1]

Abraham Silberschatz
Gael N. Buckley

Department of Computer Science
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 previous database systems using this concept preserved consistency by use of transaction rollbacks. These rollbacks require a considerable amount of overhead, which degrades performance of the system. In this paper we develop a new multi-version locking protocol that is based upon the non-two-phase guard locking protocol of Silberschatz and Kedem. The new scheme ensures consistency and deadlock-freedom without the use of rollbacks.

[1]This research was supported in part by the Office of Naval Research under Contract N00014-80-K-0987, and by the National Science Foundation under Grant MCS 81-04017.

1. Introduction

Designers of large database systems have long realized that the response time of queries and updates can be decreased by concurrent processing of multiple user transactions. If it is assumed that each transaction when executed alone maintains the consistency of the database, then the database system must ensure that a set of transactions when executed concurrently will also maintain consistency of the database. A system that guarantees this property is said to ensure serializability [1].

In order to ensure serializability, the system must restrict the interactions of the transactions executing in the database. This can be done using locking protocols [2], atomic actions [3], optimistic concurrency control schemes [4], or time-stamp ordering [5]. All these schemes were devised to increase the level of concurrency allowed in the system.

One newly popular method for increasing concurrency is the multi-version data item concept. This concept allows each data item to have a sequence of versions of different values, and a set of rules that specify which version a particular transaction is to read. 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 both Bayer et al. [8,9] and Stearns and Rosenkrantz [10] have presented various rules for ensuring consistency for systems using the multi-version concept. Necessary and sufficient conditions for serializability of multi-version database protocols, and hierarchies of multi-version protocols, are discussed by Papadimitriou and Kanellakis [11] and Bernstein and Goodman [12].

Most previous systems which use the multi-version concept use transaction rollback as a means to ensure consistency and deadlock freedom. Transaction rollback is used when the database system has allowed a set of transactions to enter a deadlocked or potentially non-serializable state. Serializability or deadlock freedom is reestablished by removing a transaction from the system, which entails removing all versions created by this transaction and all system information concerning this transaction. These rollbacks require a considerable amount of overhead, and therefore degrade performance of the system. This performance cost has been acceptable in existing database systems, since at most only a few transactions are concurrently active. However, the number of concurrent transactions can be expected to rise dramatically, due to new hardware technologies and the evolution of computing environments consisting of networks of local machines accessing global databases. In such an environment, the overhead due to rollbacks can be expected to increase significantly.

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 guard locking protocol previously proposed by Silberschatz and Kedem [13,14].

2. Locking Protocols

Many database systems ensure serializability by dividing the database into entities, and restricting access to an entity by use of a concurrency control scheme. 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 after all accesses are complete. A locking protocol may thus be viewed as a set of rules defining the allowable sequences of lock and unlock instructions 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. For all protocols discussed in this section, we allow a data item locked in S mode to have several simultaneous S locks on it, but restrict a data item locked in X mode to only have one lock on it. The multi-version protocol introduced in the next section relaxes this restriction on the X lock.

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 without restrictions on the order in which entities may be locked, it is necessary and sufficient that all transactions be two-phase to ensure serializability. The two-phase protocol has two drawbacks. First, it restricts the amount of concurrency allowed in a system; second, it is not deadlock free.

Silberschatz and Kedem [15] have shown that if one has a priori knowledge as to how the entities of the database are organized (logically or physically), one may design non-two-phase locking protocols which assure serializability and deadlock freedom. Since then a number of new non-two-phase locking protocols were developed which potentially allow more concurrency than two-phase protocols [16-20].

One of the more general non-two-phase locking protocols is the guard protocol [13], used for databases modelled as directed acyclic graphs. By proper choice of sets of vertices in the guards, all previously proposed non-two-phase protocols (e.g. tree [15], Majority [17], and DAG [18]) can be obtained as special cases of that protocol. The basic protocol restricted a transaction to employ only X mode locks. In [14], a general result concerning the extension of all protocols that employ X locks only to also employ S locks was presented. Below, we apply this general result to the guard protocol to produce a new protocol, called heterogeous guard protocol. This new protocol assures serializability and deadlock-freedom, and is the basis of the multiversion protocol presented in this paper.

Definition 1: The database graph is a rooted acyclic graph G=(V,E), with V the set of data items and E the access path between data items.

Definition 2: A database graph G is a guarded database graph if and only if for each vertex $v \in V$ (except the root) we associate a non-empty set of pairs (subguards):

$$\text{guard}(v) = \{<A_1^v, B_1^v>, \ldots, <A_M^v B_M^v>\}$$

satisfying the conditions:

1. $\emptyset \neq B_i^v \subseteq A_i^v \subseteq V$,
2. If $u \in A_i^v$, then u is a father of v, and
3. $A_i^v \cap B_j^v \neq \emptyset$ for every i and j.

Definition 3: We shall say that a subguard (A_i^v, B_i^v) is satisfied by transaction T, if and only if T currently holds a lock on all vertices in B_i^v, and it had locked (and possibly unlocked) all the vertices in $A_i^v - B_i^v$.

There are two types of transactions in the heterogeneous guard protocol, read-only transactions and update transactions, denoted R_i and U_i, respectively. An update transaction may only issue X locks, while a read-only transaction may only issue S locks. These transactions operate on a guarded graph G by the following rules:

1. Update transactions must start by locking the root of the DAG first. Read-only transactions may lock any vertex first.
2. A transaction T may lock any subsequent vertex v only if T has not previously locked v and there exists a subguard (A_i^v, B_i^v) satisfied by T.
3. A vertex may be unlocked at any point in time.

Theorem 1: The heterogeneous guard protocol assures serializability and deadlock freedom.

Proof: The serializability proof follows from the fact that the guard protocol with X locks only assures serializability [13], and by applying the general result given in [14] to that protocol. □

In order to prove freedom from deadlock, we must first establish several definitions and lemmas.

Definition 4: Let T_i and T_j be two transactions. We shall write $T_i \cap T_j \neq 0$ if both T_i and T_j access a common data item and one of them updated it.

Definition 5: We say vertex v_i is above v_j if there exists a path from v_i to v_j in the graph. An acyclic graph admits a partial ordering of vertices, and thus the above relation is not reflexive.

Definition 6: Let T_i be either an update or read-only transaction. We will denote the first vertex locked by T_i by F(i).

Lemma 1: Let T_i and T_j be two transactions such that $T_i \cap T_j \neq \emptyset$. If F(i) is above F(j), then F(j) is the first vertex locked in $T_i \cap T_j$ (by either transaction).

Proof: Assume by contradiction that some other vertex $v \neq F(j)$ is the first vertex locked in

$T_i \cap T_j$. If $v = F(i)$, there must be a path from F(j) to F(i) (implied by the guard protocol), which contradicts the assumption that F(i) is above F(j). If v is neither F(i) nor F(j), there exists an A_j^v and B_i^v locked by T_j and T_i respectively. But since the guard protocol stipulates $A_j^v \cap B_i^v \neq \emptyset$, this contradicts the assumption that v was the first vertex locked in $T_i \cap T_j$. □

Lemma 2: Let T_i and T_j be two transactions such that $T_i \cap T_j \neq \emptyset$, F(i) is above F(j), and at least one transaction is an update transaction. If T_i locks some $v \in T_i \cap T_j$ first, T_i locks all $v \in T_i \cap T_j$ first.

Proof: By induction on the longest path from F(j) to a vertex v.

1. p=0. Trivial.
2. p>0. By the heterogeneous guard protocol, $A_j^v \cap B_i^v \neq \emptyset$, where all vertices in B_i^v and A_j^v are fathers of v. Since any path from F(j) to a father of v is at least one arc shorter than the longest path from F(j) to v, the induction hypothesis states that T_i locked all vertices in $B_i^v \cap A_j^v$ first. Since a transaction can only lock a data item once, and X and S locks are mutually exclusive, T_i must lock v first. □

Proof of theorem (deadlock freedom part): Assume by contradiction that a deadlock exists and there is a cycle $T_1, T_2, \ldots, T_N, T_1$, where $T_{i+1 \bmod N}$ waits for T_i to release a data item. We will show that this reduces to the case where the cycle consists of update transactions only, each of which claims to lock the root before the next transaction in the cycle. Indeed, for all sequences of the form U_i, U_{i+1}, U_i must have locked some vertex before U_{i+1}, since U_{i+1} waits on U_i to unlock a data item. All other sequences must be of the form U_{i-1}, R_i, U_{i+1}, since read-only transactions need not wait on each other. Each of the three transactions in this sequence must lock F(i), by Lemma 1 and the fact that the root of G is above all other vertices in the graph. Since U_{i-1} locks F(i) before R_i and R_i locks F(i) before U_{i+1}, U_{i-1} locks F(i) before U_{i+1}. We now construct a new cycle by removing all R_i from the earlier cycle, so that each update transaction U_i in this new cycle has locked some vertex before U_{i+1}, and by Lemma 2, locked all vertices in $U_i \cap U_{i+1}$ before U_{i+1}. The existence of such a cycle (ordered by the time at which transactions lock the root of G) contradicts the restriction of the guard protocol that each transaction can only lock a vertex once. □

We now extend the heterogeneous guard protocol to database systems using the multi-version concept. We enhance concurrency by allowing the possibility of read-only and update transactions to access the same data item simultaneously.

3. The Multi-version Guard Protocol

The multi-version guard protocol (MVG) uses the heterogeneous guard protocol precisely as described above, and adds concurrency by not requiring shared and exclusive locks to be mutually exclusive. The protocol must still guarantee that each transaction reads the proper version of the data item in question. This is accomplished by the use of a timestamp ordering on all entities of the database system.

We affix a timestamp to each transaction R_i and U_i, and also to each data item E. The timestamps are assigned as follows:

1. U_i is assigned timestamp TS_i when U_i locks the root of G. It may be generated in a variety of ways, but must fulfill the conditions that $TS_i > TS_j$ if U_j has unlocked the root.
2. Data item E has a new version created by U_i and appended to E immediately after U_i unlocks E. This version has timestamp $k = TS_i$, and is denoted e_k. The sequence of versions of E are ordered by timestamp and are denoted $<e_1,...,e_k>$.
3. R_i is assigned timestamp MAX_i when R_i successfully locks F(i) with sequence $<f_1,...,f_k>$. $MAX_i = k$, the timestamp of f_k.

We note that by defining Max_i in this manner we may have the undesirable possibility of having a read-only transaction R_i and an update transaction U_j such that U_j has already terminated, R_i is still executing in the system, and R_i is before U_j in the serializability order. This is done in order to simplify the presentation of our basic algorithm. In Section 5 we modify the way the value of Max_i is assigned to eliminate the above deficiency.

The availability of multi-version data items allows the possibility for transactions to have both S and X locks on a data item simultaneously. We say X and S locks are compatible if they can be held concurrently on a data item. More formally, given a set of locking modes, we can define compatibility relations among them as follows. Suppose transaction T_j currently holds a lock of mode B on data item E, and T_i requests a lock of mode A on item E. 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 COMP(A,B) = true. Traditionally, the compatibility relation among the S and X modes of locking is defined to be COMP(A,B) = true if and only if A = B = S. This is the relation for the heterogeneous guard protocol in section two. For the MVG protocol, we 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 ⇔ 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 guard protocol with the multi-version data item scheme. The new protocol can be summarized as follows:

1. Update transactions can only request X locks. Read-only transactions can only request S locks.
2. Update transactions must start by locking the root of the DAG first. Read-only transactions may lock any vertex first.
3. A transaction T may lock any subsequent vertex v only if T has not previously locked v and there exists a subguard (A_i^v, B_i^v) satisfied by T.
4. A Read-only transaction R_i that has successfully locked entity E, must read version e_g such that g is the largest time-stamp $\leq Max_i$.
5. An Update transaction U_i that has successfully locked entity E, must read version e_g such that g is the largest time-stamp $\leq TS_i$.
6. A vertex may be unlocked at any point in time.

Note that rule five allows an update transaction to read its own version of entity E, even though this version has not yet been committed and become part of the sequence of entity E.

4. Proof of Serializability and Deadlock Freedom

We now prove that MVG assures serializability and deadlock freedom. In order to do so we must first introduce some standard definitions to be used in the remainder of this section.

<u>Definition 7</u>: A history H is the trace, in chronological order, of the concurrent execution of a set of transactions $T = \{T_0,...,T_{n-1}\}$.

<u>Definition 8</u>: 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_j and T_i read or created version e_m, $m < j$.

b. T_j read version e_n and T_i created version e_i, $i \leq n$.

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

<u>Lemma 3</u>: The MVG protocol assures serializability if and only if all allowable concurrent executions of transactions produce an acyclic $\rightarrow$ relation.

<u>Proof</u>: 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. □

<u>Lemma 4</u>: 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 guard protocol, the result follows. $\square$

Lemma 5: 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 longest path from F(j) to E.

q=0: Let $<f_1,f_2...,f_k>$ be the sequence when R_j successfully locked F(j). By our scheme $MAX_j = k$. Clearly $k \geq TS_i$. If $k = TS_i$ then the result follows from the fact that f_k is inserted into the sequence only after U_i unlocked F(j). If $k > TS_i$ then let U_m be the transaction that created f_k. Since $TS_m > TS_i$, by Lemma 4 it follows that U_m locked F(j) only after U_i unlocked F(j), and the result follows.

q>0: By the heterogeneous guard protocol, $B_i^E \cap A_j^E \neq \emptyset$, where all vertices in B_i^E and A_j^E are fathers of E. Since any path from F(j) to a father of E is at least one arc shorter that the longest path from F(j) to E, the induction hypothesis states that U_i locked all vertices in $B_i^E \cap A_j^E$ first. Since a transaction can only lock a vertex once, and COMP(S,X) = false if $MAX_j \ngeq TS_i$, the result follows. $\square$

Lemma 6: Let U_i and U_j be update transactions that interact in the system. Then $U_i \rightarrow U_j \Leftrightarrow TS_i < TS_j$.

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 4, U_j successfully locked E only after U_i unlocked it. If U_i 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 the result follows. $\square$

Lemma 7: Let U_i and R_j be update and read-only transactions respectively that interact in the system. Then $U_i \rightarrow R_j \Leftrightarrow TS_i \leq Max_j$.

Proof:

$\Rightarrow$: By the definition of MVG, 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 5, R_j locked some item E only after U_i unlocked E. Since $MAX_j \geq TS_i$, R_j must have read version e_m, $m > TS_i$ and the result follows. $\square$

Lemma 8: Let R_i and U_j be read-only and update transaction respectively that interact in the system. Then $R_i \rightarrow U_j \Leftrightarrow Max_i < TS_j$.

Proof:

$\Rightarrow$: Assume by contradiction that $Max_i \geq TS_j$. By Lemma 5, 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. To establish a precedence relation between R_i and U_j, U_j created version e_g with $g = TS_j$. By the rules of MVG, R_i must read version e_k, $k \geq TS_j$ and hence $U_i \rightarrow R_j$, which is a contradiction.

$\Leftarrow$: By the definition of MVG, R_i can only read a version e_k of data item E if $k \leq Max_i$. $\square$.

Theorem 2: The MVG protocol assures serializability.

Proof: The proof follows directly from Lemmas 3, 7, 8, and 9. 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. $\square$

Theorem 3: The MVG protocol assures deadlock freedom.

Proof: The proof follows from two facts. First, each transaction must follow the heterogeneous guard protocol, which was proven deadlock free in Theorem 1. Second, the compatibility relation between the S and X mode of locking for MVG has fewer delay conditions (i.e., an update transaction can never be delayed by a Read-only transaction) than the one defined for the guard protocol. $\square$

5. Discarding Old Versions

Since no database system can keep an indefinite number of versions per data item on mass storage (e.g., disk), there must be some method to discard older versions (i.e., delete them, or move them to tape) when they will no longer be accessed by active transactions. In this section we present such a method and show how this can be used in eliminating the problem we discussed in Section 4 concerning the value of Max_i.

All update transactions access only the most current version of a data item E, since all update transactions lock all items in the order they lock the root of G, which is in ascending order of timestamp. However, a read-only transaction R_j can access a version earlier than the most current version of E, since no recent update transaction may have created a version for F(j). If we can construct a number M such that all active R_j have $MAX_j \geq M$, then the system can discard e_1 through e_{g-1} in the sequence $<e_1,...,e_{g-1},e_g,e_{g+1},...,e_k>$ of data item E, where the timestamp of $e_g \leq M$ and the timestamp of $e_{g+1} > M$.

To find such a number, the database system can scan the entire database to determine the data item whose most current version has the smallest k. Since each $MAX_i \geq k$, this fulfills the requirement given above. However, this procedure is prohibitively expensive for any database of reasonable size. Hence we will use the timestamps of the update transactions to force MAX_j of any read-only transaction past some precalculated value. In other words, we modify the way MAX_j is computed.

We can artificially assign M as the lower bound for the timestamp of any read-only transaction entering the database, provided there exists no active update transaction which can create a new version with timestamp $\leq$ M. Intuitively, this follows from the fact that any update transaction which later locks F(j) can only create a version with a larger timestamp than M, and hence will preserve the ordering of transactions.

Definition 9: If U_j has the least timestamp of any update transaction presently active in the system, then $M = TS_{j-1}$.

Definition 10: We redefine the way MAX_j is computed to be the maximum(M,k), where M is defined above, and k is the timestamp of f_k in the sequence $<f_1,...,f_k>$ of F(j) at the time R_j successfully locked F(j).

This redefinition of MAX_j requires the following alteration of each of the proofs of Lemmas 6, 8, and 9. Since $M < TS_i$ for any active transaction U_i and each proof assumes $MAX_j \geq TS_i$, it remains that $MAX_j = k$. Therefore all the lemmas hold, and the redefined protocol is still serializable.

We now present the algorithm to determine M using the timestamps of the update transactions. The function to generate a timestamp for update transaction U_i is denoted $succ(U_{i-1})$, where U_{i-1} is the update transaction which locked the root of G before U_i. There is an ordered list A containing all active update transactions in ascending order of timestamp, and is of size equal to the number of active update transactions. The algorithm to determine the value of M functions as follows:

1. Initialize. Set A to empty and M:=0.
2. When U_i enters the system, append U_i to the end of A.
3. When U_i leaves the system, delete U_i from A. If TS_i = succ(M), M must be updated. If A is empty, M is set to the timestamp of the update transaction that last locked the root of G, otherwise M is set to TS_{c-1}, where U_c is the first element of A.

We present a short example to illustrate the operation of the algorithm. For simplicity, we assume $TS_1 = 1$ and $succ(TS_i) = TS_i+1$. Transactions U_1, U_2, and U_3 enter the system, and are appended to A in ascending order of timestamp. The value of M=0. Transaction U_2 completes and is deleted from A. Since $TS_2 \neq succ(M)$, no action is taken to recompute M. At a later time transaction U_1 completes, is deleted from A, and finds TS_1 = succ(M) = 1. U_3 is the first element of A, so M is set to 2, the timestamp of U_2.

We finally must guarantee that all read-only transactions active in the database have $MAX_j \geq$ k, for some number k. The scheme just presented divides the set of active read-only transactions into classes, where each class is composed of transactions with $MAX_j \geq M_j$, the value of M when the transaction entered the database. In order to distinguish which class a read-only transaction

R_j belongs in, each R_j remembers both MAX_j and M_j. These individual classes can be counted separately, or grouped into two major classes around a particular value of M called CHANGE, where the first class has $M_j \geq$ CHANGE, and the second class has $M_j <$ CHANGE. We present the latter algorithm, where the first and second classes are counted in RC_1 and RC_2 respectively. The algorithm functions as follows:

1. Initialize. Set RC_1:=0, RC_2:=0, and CHANGE:=0.
2. When the system decides to discard additional versions, set CHANGE:=M, RC_2:=RC_1+RC_2, and RC_1:=0. This step must be uninterrupted by the arrival or departure of any read-only transaction.
3. When R_j enters the database, set RC_1:=RC_1+1, M_j:=M, and MAX_j:=max(M,k).
4. When R_j leaves the database, if $M_j \geq$ CHANGE, set RC_1:=RC_1-1, else set RC_2:=RC_2-1.
5. When RC_2=0, the database may discard all versions $<e_1,...e_{g-1}>$ of data item E, given that the timestamp of $e_g \leq$ CHANGE and the timestamp of $e_{g+1} >$ CHANGE.

We illustrate the algorithm by extending the same example given above. The system begins with M, CHANGE, RC_1, and RC_2 set to 0. R_1 and R_2 enter, are assigned M_j=0, and set RC_1 to 2. RC_1 completes, finds $M_1 \geq 0$, and sets RC_1 to 1. Then update transactions U_2 and U_1 finish and set M to 2. The system decides to discard versions, sets CHANGE to 2, RC_2 to 1, and RC_1 to 0. R_3 enters, is assigned M_3=2, and sets RC_1 to 1. R_2 completes, finds $M_2 < 2$, and sets RC_2 to 0. The system finds RC_2=0, and discards any appropriate versions.

6. Comparison

It is quite difficult to compare our locking protocol with other previously published schemes using the multi-version concept. 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.

To simplify the comparisons, we first present the relevant portions of each of the three schemes. All three schemes commit versions created by a transaction only at the time the transaction completes and leaves the database. To test for serializability, both A1 and A3 assign a timestamp to each transaction in the database. A1 uses the timestamp to restrict the sequence of operations on a data item to conform to both the $\rightarrow$ relation (as defined earlier) and the increasing sequence of timestamps, while A3 uses timestamps as a priority ordering to detect possible inconsistency and deadlock, and to decide which transaction should be rolled back. A2 uses a global dependency graph to detect both deadlocks and possible nonserializable sequences, where an arc in the graph is the precedence relation between a pair of conflicting transactions on an individual data item. We now compare the four schemes in each of four areas.

a. Inconsistency and deadlock -- Both A1 and A2 allow a set of transactions to reach potentially nonserializable states. If A1 cannot order the sequence of operations on a data item to conform to $\rightarrow$ and still increase in timestamp number, the update transactions with versions violating this criterion must be rolled back. In A2, a cycle detection algorithm must be frequently performed to decide if a cycle exists in the graph. Existence of a cycle implies deadlock or potentially nonserializable results, and some transaction in the cycle is then removed. A3 decides if the possibility of deadlock or inconsistency exists by using both the state of the conflicting transactions and their respective timestamps. If such a possibility exists, the system rolls back one of the conflicting transactions. Our scheme prevents deadlock and ensures serializability by a partial order on the locking sequence of data items, and the compatibility relations between lock modes. This precedence relation implies that additional data items may need to be locked.

b. Reading a data item -- If a read request is issued on a data item with an uncommitted version, then A1, A3, and our scheme may need to delay the request until the appropriate version has been committed. As mentioned before, commitment in A1 and A3 occurs only when the transaction has completed. In addition, A3 can grant a read request of higher priority, which implies transactions with previously granted write requests of lower priority may need to be rolled back. In our scheme, commitment occurs at the moment when a transaction unlocks a data item, which can occur well before the transaction completes.

   In A2, a data item may always be read immediately. However, a cycle detection algorithm must be performed for each read request to determine which version should be read. Cycle detection algorithms are quite expensive, especially in a distributed environment.

c. Writing a data item -- In each of A1-A3, a transaction which issues a write request may need to be rolled back to preserve consistency of the database. In A1, the write request is granted if it preserves the ordered sequence of operations to the data item, otherwise it is rolled back. A2 grants the write request immediately if there is no conflicting active transaction, otherwise the cycle detection algorithm is invoked to determine if the request should be granted or the transaction rolled back. In A3, the write request is immediately granted if there is no conflicting active transaction. If there is conflict, the priority numbers and states of the transactions are compared, and either the request is delayed until the previous transaction has completed, or one conflicting transaction is immediately rolled back.

   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.

d. Termination -- Of the four schemes, only A2 requires an additional termination procedure. It checks consistency at termination by invoking one final cycle detection, and rolls back the transaction if a cycle is found.

Since the MVG protocol requires no special handling for termination, deadlock, or inconsistent database states, it has a simple and efficient implementation without relying on rollback. The main disadvantage is that additional data items may have to be locked because of the required access ordering. 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 for a transaction.

7. Conclusion

We have presented a new multi-version concurrency control scheme which guarantees serializability and deadlock freedom without the use of rollbacks. Rollbacks degrade performance both in the time required by the database coordinator to reset the database to an earlier state and the computation time wasted by the individual transaction. This disadvantage becomes greater as new computing environments increase the number of concurrently executing transactions, where the increased probability of conflict increases the frequency of rollbacks. It should be noted, however, that rollbacks may still occur in the system due to such circumstances as hardware failure. Thus, in order to ensure atomicity, a commit protocol must also be used. In a previous paper [22], we have shown how a simple commit protocol can be effectively constructed without interfering with the protocol that is used for ensuring serializability. Arguments in support of protocols that do not require rollbacks as a means for ensuring serializability were also presented in that paper.

A possible disadvantage of our scheme is that additional data items may have to be locked. However, since a non-two-phase protocol is used, 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 the MVG protocol can be effectively 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. The access paths between and within the databases at each site must be able to be modelled as a rooted DAG. In each site a transaction accesses the data item residing there. In such an environment the implementation of the MVG protocol is identical to the centralized system. Most importantly the MVG 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]). Also note that although it appears that the root node will create a bottleneck, this is not really the case. This is due to the fact that only update transactions must lock the root node first.

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). The writing to all copies of the data item must be carried out before the item is unlocked.

Hence the MVG protocol can be implemented in both a centralized and distributed environment.

References

1. Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. The notions of consistency and predicate locks in a database system. CACM 10, 11 (Nov. 1976), 624-723.

2. Gray, J., Notes on database operating systems. Research Report, IBM Research Lab, San Jose, Feb. 1978.

3. Reed, D.P. Naming and synchronization in a decentralized computer system. Ph.D. Thesis, M.I.T. Dept. of Electrical Engineering and Computer Science, Sept. 1978.

4. Kung, H.T., and Robinson, J.T. On optimistic methods for concurrency control. ACM Transaction of Database Systems 6, 2 (June 1981) 213-226.

5. Bernstein, P.A. and Goodman, N. Time-stamp-based algorithms for concurrency control in distributed database systems. Proceedings Sixth International Conference on Very Large Data Bases (Oct. 1980), 285-300.

6. Honeywell File Management Supervisor, Order Number DB54, Honeywell Information Systems Inc., 1973.

7. Stearns, R.E., Lewis, P.M. and Rosenkrantz, D.J. Concurrency control for database systems. Proceedings IEEE Symposium on Foundations of Computer Science (Oct. 1976), 19-32.

8. Bayer, R., Heller, H. and Reiser, A. Parallelism and recovery in database systems. ACM Transactions on Database Systems 5, 2 (June 1980), 139-156.

9. Bayer, R., Elhardt, E., Heller, H. and Reiser, A. Distributed concurrency control in database systems. Proceedings Sixth International Conference on Very Large Data Bases (Oct. 1980), 275-284.

10. Stearns, R.E. and Rosenkrantz, D. Distributed database concurrency control using before values. Proceedings ACM-SIGMOD International Conference on Management of Data (April 1981).

11. Papadimitriou, C. and Kanellakis, P. On Concurrency Control by Multiple Versions, Proceedings ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (March 1982), 76-82.

12. Bernstein, P. and Goodman, M. Concurrency Control Algorithms for Multiversion Database Systems, Proceedings ACM SIGACT-SIGOPS Symposium on Distributed Computing (August 1982), 209-215.

13. Silberschatz, A., and Kedem, Z. A family of locking protocols for database systems that are modeled by directed graphs. IEEE Transactions on Software Engineering (to appear).

14. Kedem, Z., and Silberschatz, A. Locking protocols: from exclusive to shared locks. University of Texas at Austin, Technical Report, 1980.

15. Silberschatz, A., and Kedem, Z. Consistency in hierarchical database system. JACM 27, 1 (Jan. 1980), 72-80.

16. Kedem, Z., and Silberschatz, A. Non-two-phase locking protocols with shared and exclusive locks. Proceedings International Conference on Very Large Data Bases (Oct. 1980).

17. Kedem, Z., and Silberschatz, A. Controlling concurrency using locking protocols.

Proceedings 20th IEEE Symposium on Foundations of Computer Science (Oct. 1979), 274-285.

18. Yannakakis, M., Papadimitriou, C.H., and Kung, H.T., Locking policy: safety and freedom from deadlock. Proceedings 20th IEEE Symposium on Foundation of Computer Science (Oct. 1979), 286-297.

19. Fussell, D., Kedem, Z., and Silberschatz, A. A theory of correct protocols for database systems, Proceedings Seventh International Conference of Very Large Data Bases (Sept. 1981), 112-124.

20. Mohan, C., Fussell, D., and Silberschatz, A. Compatibility and commutativity in non-two-phase locking protocols, Proceedings ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (March 1982), 283-292.

21. Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M. System level concurrency control for distributed database systems. ACM Transactions on Database Systems 3, 2 (June 1978), 178-198.

22. Silberschatz, A. A case for non-two-phase locking. IEEE Transactions on Software Engineering (to appear).