Lock Conversion in Non-Two-Phase Locking Protocols

C. MOHAN, DONALD FUSSELL, ZVI M. KEDEM, MEMBER, IEEE, AND ABRAHAM SILBERSCHATZ

Abstract—A locking protocol is a set of rules governing the manner in which the database entities may be accessed. Such a protocol usually employs several kinds of locks. Most of the previous work in this area has assumed that once a transaction acquires a particular kind of lock on a data item it is not allowed to convert this lock to another kind. In this paper we perform a systematic study of the consequences of allowing lock conversions in non-two-phase locking protocols, and show how this leads to increased concurrency and affects deadlock-freedom. The non-two-phase protocols that we study are the very general guard protocols defined for databases in which a directed acyclic graph structure can be superimposed on the data items. We present very natural generalizations of these protocols, including correctness proofs, and develop deadlock removal methods.

Index Terms—Concurrency, consistency, database systems, deadlocks, locking protocols, rollbacks, serializability, transactions.

I. INTRODUCTION

ENSURING the consistency of the stored/retrieved data is an important issue in database management when the database is accessed and updated concurrently by a number of transactions. A common approach to this problem is to define a transaction as a unit that preserves consistency (e.g., it is assumed that each transaction, when executed alone, transforms a consistent state into a new consistent state), and require that the outcome of processing a set of transactions concurrently be the same as one produced by running these transactions serially in some order. A system that ensures this property is said to be serializable [3]. Associated with concurrent access to data is the problem of deadlocks. Deadlocks arise as a result of circular wait conditions involving two or more transactions. A system which does not give rise to deadlocks is said to assure deadlock-freedom.

While many kinds of concurrency control mechanisms have been proposed over the years to ensure serializability, the most common one is a locking protocol. Such a protocol can be simply viewed as a restriction on when a transaction may lock and unlock each of the database items. Locking a data item inhibits certain types of concurrent activity on that item until the lock is released. The pioneering effort in this area was the proposal of the two-phase locking (2PL) protocol [3]. 2PL prevents a transaction from acquiring a lock on any item after it has unlocked some item. In other words, 2PL constrains a transaction’s locking activities to consist of two phases: a phase in which only lock acquisitions are allowed, followed by a phase in which only lock releases are permitted. The two-phase protocol has the major drawback that it severely restricts the amount of concurrency allowed in a system.

The problem of insufficient amount of concurrency can be partially resolved by providing more interpretation concerning the precise functions performed by a transaction [10]. One such interpretation is to organize the items of the database in the form of a directed acyclic graph (DAG), with each vertex of the graph being a data item, and forcing the transactions to acquire locks on those items in some constrained ways. The edges of the graph could model logical or physical relationships. Adoption of this graph interpretation has led to the development of a number of non-two-phase (non-2PL) protocols which potentially allow more concurrency than two-phase protocols [4], [6], [7], [14], [17]–[19].

Throughout our work we adopt serializability as the correctness criterion for locking protocols. Here we will only be concerned with two modes of locking—exclusive (X) and shared (S). 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 the reading of the item. It should be clear that when an X mode lock is being held on an item, no other lock requests on that item can be granted. On the other hand, when an S mode lock is held, other S mode requests on that item can be granted.

As for concurrency, some protocols may have the following undesirable features, which may potentially reduce the level of allowable concurrent access to data.

1) A transaction may be required to hold locks longer than needed; that is, when a transaction wants access to a data item only at a certain point in its execution, it may be forced to acquire the lock on that item at an earlier point in time.

2) A transaction may be forced to acquire unnecessary access privileges; that is, when an S mode lock would suffice, a transaction being forced to acquire an X mode lock.

3) A transaction may be forced to acquire unnecessary locks; that is, locks on data items the transaction does not need access to.

As for deadlocks, we examine each protocol to determine
whether deadlocks are possible. If they are, we make sure that deadlock detection and/or recovery is simple and less costly. In many cases, we propose additional conditions which guarantee freedom from deadlocks.

In this paper, we examine the mechanism of lock conversion as a means of increasing concurrency. We consider protocols which allow transactions to convert a lock held on an item from one mode to another without unlocking it. In order to acquire and release a lock on a data item, a transaction must issue one of the following instructions.

- **rising:** LS—lock S;
- LX—lock X;
- UP—upgrade from S to X.

- **falling:** UN—unlock either S or X;
- DN—downgrade from X to S.

All the protocols described in this paper require that a transaction be rising-falling on each data item. That is, if a transaction issues a falling instruction on a data item then it may no longer issue a rising instruction on that data item.

Most of the previous work on locking protocols implicitly assumed that once a transaction obtains a particular mode of lock on a data item, it would hold only that mode of lock on that item until it is unlocked. Our primary motivation for investigating the subject of lock conversions is our desire to provide a means for the user to exploit the potential for additional concurrency. When a transaction needs to modify a data item, it may first need to read that item and some other items, and to do a long computation in order to produce the new value for the former. If the transaction were to obtain an X mode lock on that item right at the beginning, then it would not be available to any other transaction for a long time. On the other hand, if the update transaction were to initially obtain an S mode lock, do the computation, and just before installing the new value obtain an X mode lock, then some transactions which wanted only to read the item might have been able to gain access to that item in the meantime (if conversions were allowed).

Another typical situation might arise when a transaction does not know until it actually reads several data items whether it needs to modify a certain item. In such cases the transaction can avoid “locking out” (by obtaining an X mode lock on the latter) other transactions unnecessarily by first obtaining only an S mode lock on the latter. Further, if the granularity of the data items is very coarse, then a transaction might, after updating a portion of a data item, still want to read other portions of the data item (without any more updating of that data item). In such a case, once the transaction has finished updating, it should be able to let other transactions read the item in parallel.

Finally, it is worthwhile to point out that lock conversion has been found to be of practical interest, as is evidenced by the fact that a commercial database management system (IBM's IMS [12]) provides one form of lock conversion (upgrading or conversion from S mode to X mode). Thus, the study of lock conversions is not only of theoretical interest but also of practical interest. There are few papers which consider lock conversions [1], [2], [5], [11], [12], [16]. Not much emphasis has been given in the above papers to the effects of lock conversions on locking protocols which assure serializability.

Recently, a generalization of 2PL to allow lock conversion was proposed [9], [13]. This new protocol requires that a transaction’s locking activities constitute two phases: a growing phase in which in addition to acquiring new locks, S mode locks may be converted to X mode locks, followed by a shrinking phase in which in addition to releasing locks, X mode locks may be converted to S mode locks. In this paper, we continue with this line of work by studying the impact of lock conversions on two types of non-two-phase locking protocols. We present very natural generalizations of those protocols (including correctness proofs) and develop deadlock-removal methods.

## II. Review of the Non-Two-Phase Guard Protocol

We have pointed out in the Introduction that by imposing a certain ordering relationship on the data items and by forcing the transactions to acquire locks on those items in some constrained ways, non-two-phase (non-2PL) protocols can be designed that provide more concurrency than the 2PL protocol. One of the most general non-two-phase locking protocols, is the Guard locking protocol (GLP) presented in [18]. Let us first describe the basic protocol when transactions are restricted to X mode locks. To simplify our discussion we restrict our attention to the version of GLP for rooted DAG’s. We denote the root as R and the set of vertices as V.

### A. The Basic Guard Protocol

We start our discussion by presenting several definitions.

**Definition 2-1:** We shall say that a graph is a guarded graph if and only if with each vertex v ∈ V we associate a nonempty set of pairs (subguards)

\[
guard(v) = \{< A^v_1, B^v_1 >, \cdots, < A^v_{M_0}, B^v_{M_0} > \}
\]

satisfying the conditions

1) \( \emptyset \neq B^v_i \subset A^v_i \subset V \)
2) \( \forall u \in A^v_i \; [u \text{ is a parent of } v] \)
3) \( A^v_i \cap B^v_j \neq \emptyset \; \text{for every } i \; \text{and } j \).

**Definition 2-2:** We shall say that a subguard \((A^v_i, B^v_i)\) is satisfied in mode m where m is X or S by transaction T if and only if T is currently holding a mode m lock on all the vertices in \(B^v_i\), and it had locked (and possibly unlocked) all the vertices in \(A^v_i - B^v_i\) in mode m.

**The rules of the guard protocol** (for transaction T) are as follows.

1) Any vertex may be locked first.
2) Subsequently, a vertex v can be locked only if there exists a subguard \((A^v_i, B^v_i)\) satisfied in mode X by T.
3) Vertices may be unlocked any time.

**Theorem 2-1:** The guard protocol assures serializability and deadlock-freedom.

**Proof:** See [18].

GLP is one of the most general non-2PL protocols. By proper choice of the sets of vertices in the guards, we can get the previously proposed protocols (like the tree protocol [17], the strong/majority protocol [6], and the DAG protocol [19]) as
special cases (see [18] for many examples) of GLP. For example, the DAG protocol of [19] allows a transaction to lock any vertex at first and to subsequently lock a vertex $v$ only if all immediate predecessors of $v$ [i.e., $F(v)$] had earlier been locked (and possibly unlocked) and at least one parent of $v$ is still locked. The DAG protocol can be obtained by defining the guards as follows:

$$\text{guard}(v) = \{<F(v), [u]> | u \in F(v)\}.$$ 

Note that this definition obeys the conditions of Definition 2-1.

In this paper we do not address the issues relating to the advantages/disadvantages of the choice of one way of defining the guards in preference to other ways of defining the guards.

The Guard locking protocol was extended in [8] to support both the $X$ and $S$ modes of locking. The protocols can be broadly classified into two types: heterogeneous and homogeneous. The heterogeneous protocols distinguish between read-only transactions (i.e., those that acquire only $S$ locks) and update transactions (i.e., those that acquire $X$ locks also), and give slightly different set of rules to be obeyed by them. The homogeneous protocols, on the other hand, do not make any such distinctions among the transactions. In the following we briefly review these two types of protocols.

### B. Heterogeneous Protocol

The heterogeneous guard protocol (called GLP') is characterized by distinction being made between update and read-only transactions. Update transactions are allowed to acquire only $X$ mode locks, while read-only transactions are allowed to acquire only $S$ mode locks. Further, the update transactions are required to start by locking $R$ (the root of the DAG) first. But for this, the two types of transactions follow the same guard protocol that we have described above. As a consequence of the requirements of GLP', the update transactions are serialized in the order in which they obtain a lock on $R$. Note that if the update transactions were also to be allowed to start by locking any vertex first then serializability cannot be guaranteed. To illustrate this, consider the database graph of Fig. 1. Suppose that the tree locking protocol is employed which requires transactions requesting a lock on any vertex $v$, other than the first, to be holding a lock on the parent of the vertex $v$. Consider now the following execution history.

$T_1 \text{ LS} R; T_1 \text{ LS A}; T_1 \text{ UN R}; T_2 \text{ LX R}; T_2 \text{ UN R};$
$T_3 \text{ LS R}; T_3 \text{ LS A}; T_3 \text{ LS B}; T_3 \text{ UN R}; T_3 \text{ UN A};$
$T_3 \text{ UN B}; T_4 \text{ LX B}; T_4 \text{ UN B}; T_1 \text{ LS B}; T_1 \text{ UN A};$
$T_1 \text{ UN B}$

This execution is not serializable since it requires $T_1$ to both precede and succeed $T_4$ in the equivalent serial history.

In the above example, the update transaction $T_4$ did not start by locking the root first. It locked only the vertex $B$. It is this action which causes the nonserializability of the concurrent execution of the four transactions. If $T_4$ had started by locking $R$ first, $A$ next, and then $B$, it would have been forced to wait for $T_1$ to finish and thereby be made to follow $T_1$ in the equivalent serial execution of the four transactions.

### C. Homogeneous Protocol

The homogeneous guard protocol (called the pitfall protocol) does not distinguish between read-only and update transactions. Instead, any transaction may acquire both $S$ and $X$ mode locks (on different items, of course), and start locking any vertex first. However, the protocol requires that the transactions be two-phase during certain segments of their locking activities. Before we present the rules of the pitfall protocol we need to present a few definitions.

**Definition 2-3:** Let $L(T_i)$ be the set of all data items locked (in mode $X$ or $S$) by $T_i$. $LX(T_i)$ is the set of data items locked in mode $X$ by $T_i$ and $LS(T_i)$ is the set of data items locked in mode $S$ by $T_i$. A lock is held on each member of these sets by $T_i$ some time during its execution.

**Definition 2-4:** We define certain subsets of $L(T_i)$, called pitfalls, as follows. Consider the subgraph of the tree spanned by the set $LS(T_i)$. Generally it splits into a number of connected components, say $A_1, A_2, \ldots, A_k$. A pitfall of $T_i$ is defined as a set of the form $A_1 \cup \{v \in LX(T_i) | v$ is a neighbor of some $w \in A_1\}$.

Note that the pitfalls are not necessarily disjoint and that the pitfalls of a given transaction in no way depend upon the locking activities of other transactions. Only vertices of $LX(T_i)$ could possibly be members of more than one pitfall of $T_i$. In Fig. 2 we present an example in which the locks acquired by a particular transaction form two pitfalls. The modes of the locks acquired are indicated next to each vertex.

**Definition 2-5:** A transaction $T_i$ is said to be two-phase on a set of items $K \subseteq L(T_i)$ if and only if $T_i^K$, the set of instructions of $T_i$ referring to only the vertices in $K$, is two-phase.

The rules of the pitfall protocol (for transaction $T$) are as follows.

1) The first lock can be acquired on any vertex.
2) Subsequently, a vertex $v$ can be locked only if there exists a subguard $(A^v_1, B^v_2)$ satisfied by $T$ (each vertex in the subguard could have been locked in any mode).

3) The transaction must be two-phase on each pitfall (but not on the union of the pitfalls).

**Theorem 2.2:** The pitfall protocol assures serializability.

**Proof:** See [8].

III. HETEROGENEOUS PROTOCOLS

Note that under GLP' an update transaction is forced to lock in $X$ mode all the items that it needs to access, even though it may be modifying only one of them. In some cases, an update transaction may, after examining some values, decide that it does not need to modify any values at all. In such instances, some potential concurrency which had existed would not have been exploited. For these reasons and others mentioned in the Introduction, we modify GLP in a natural way so that update transactions are allowed to obtain $S$ mode locks and to perform lock conversions. Both of these changes could potentially lead to increased concurrency. The new protocol is called MGLP' (modified GLP') and is defined below.

**The rules of MGLP' (for $T$) are as follows.**

1) If the first item is locked in $T$ mode, then that item (vertex) must be the root of the DAG.

2) Subsequently, a vertex $v$ can be upgraded or locked in $X$ (respectively $S$) mode only if there exists a subguard of $v$ satisfied in $X$ (respectively $S$ or $X$) mode by $T$. The exception to this rule is in the case of the root $R$—an $S$ mode lock on $R$ can be at any time upgraded to an $X$ mode lock.

3) A read-only transaction can obtain its first lock on any vertex. Vertices can be unlocked or downgraded any time.

**Theorem 3.1:** MGLP' assures serializability.

**Proof:** See Appendix.

A. **Observations**

- If a transaction starts out by locking any vertex other than $R$, it can never obtain an $X$ mode lock on any vertex.
- If a transaction starts out by locking $R$ in mode $S$ and continues by locking a number of its successors in mode $S$, then if it needs to convert an $S$ mode lock on one of the latter vertices to an $X$ mode lock, it must proceed by converting its $S$ mode lock on $R$ to an $X$ mode lock and continue to do conversions along (not necessarily all) the paths which lead to the desired vertex. If the $S$ mode lock on $R$ had been released before the transaction found a need to upgrade some other $S$ mode lock, then that transaction will not be able to achieve that conversion (since a vertex can be locked only once, a new lock on $R$ cannot be acquired).
- Even if upgrading and downgrading are not allowed in MGLP', it is still an extension of GLP, since it permits update transactions to acquire $S$ mode locks.

The main characteristics of the protocol can be best illustrated via a simple example. Fig. 3 presents a trace of the locking actions of a transaction $T$ obeying MGLP'. After the state indicated in Fig. 3(a) has been reached, $T$ decides that it wants to modify $B$. To set the stage for converting the $S$ lock on $B$ to an $X$ lock, $T$ first upgrades the lock on $R$ [see Fig. 3(b)]. Next it upgrades the lock on $A$ [see Fig. 3(c)]. Then it downgrades the lock on $R$ [see Fig. 3(d)]. (By doing this operation, $T$ loses its ability to upgrade the locks on $C$ and $D$, which was possible after state (c) was reached.) Next $T$ releases the lock on $R$ [see Fig. 3(e)] and finally upgrades the lock on $B$ [see Fig. 3(f)]. Note that the upgrading of $B$ could have been done soon after state (c) was reached.

While GLP' assures deadlock-freedom, MGLP' as presented so far does not. It can be easily shown that a deadlock can arise only when two or more transactions holding $S$ mode locks on $R$ try to upgrade those $S$ mode locks and that a dead-
lock cycle can be of length two only (see [13] for an outline of the proof). There are two methods in which the deadlock problem can be dealt with. One method is to let deadlocks occur and then take corrective action. Another method is to modify MGLP so that deadlocks are not possible. We consider both methods below.

B. Deadlock Detection

Notice that deadlock detection is not a costly operation. All that needs to be done is to check at the time a conversion request on R is made whether there is already such a pending conversion request. As soon as a second conversion request on R is made, the lock manager will detect the deadlock. To resolve the deadlock, one of the transactions can be aborted by releasing the locks (which could be only S locks) it holds. Note that this abortion will not require any rollback actions since, until that time, the transaction could not have locked any vertex in X mode. Delaying resubmission of aborted transactions will, ultimately, let one transaction obtain the X mode lock on R. The distinguishing features of MGLP compared to other deadlock-prone protocols (like 2PL) are that in the case of the former deadlocks are easily detectable and they cause no rollbacks that require restoration of the values of any database items.

C. Deadlock Prevention

One approach to deadlock prevention (without introducing new lock modes) is to disallow transactions from performing that operation which may cause a deadlock. Adopting this approach, the previously stated rules can be easily modified to disallow the upgrading of the S mode lock on the root R.

As an alternative approach to handling the deadlock problem, we introduce the U lock mode. The U mode is the same as the update mode of IMS [12]. As far as access privileges are concerned, both the U and S modes are the same in the sense that both permit the transaction to read the data item. But the U mode lock can be upgraded to an X mode lock, whereas now the S mode lock cannot be.

The lock compatibility matrix COMP is shown below. COMP(m, n) = Y means that the modes m and n are compatible (i.e., when an m mode lock is currently being held on an item by one transaction, an n mode lock request on that item by another transaction can be immediately granted). Notice that while multiple S mode locks can be held simultaneously (by different transactions) on a vertex, only one U mode lock can be held on that vertex along with the S mode locks. Note that COMP relates only to locks of two different transactions. It does not relate to upgrading of a lock by a single transaction.

The rules of the deadlock-free version of MGLP are as follows.
1) An update transaction must lock R first in U or X mode.
2) Subsequently, a vertex v can be locked in or upgraded to X mode only if the guard of v is satisfied in X mode at that time (this rule does not apply to the upgrading of the U mode lock on R).
3) A vertex v can be locked in U mode only if the guard of v is satisfied in U or X mode.
4) A vertex v can be locked in S mode only if the guard of v is satisfied in X, U, or S mode.
5) Downward conversions can be done any time.

Theorem 3-2: The extended MGLP protocol assures serializability and deadlock-freedom.

Proof: The proof is similar to the proof of Theorem 3-1.

Note that if an update transaction already holds U mode locks on the items it needs to modify (and S mode locks on the other items it needs to access), then it need not wait until it gets the X mode locks on those items to compute the new values. The new values can be computed in the transaction's private workspace and installed in the database after obtaining the X mode locks. This way the transaction would hold the X mode locks for a shorter period of time than otherwise possible. This optimization is applicable in the case of the unrestricted version of MGLP also, but there if the update transaction becomes the victim in the event of a deadlock, then the update computation would turn out to be wasteful.

IV. Homogeneous Protocols

In this section we extend the pitfall protocol by permitting lock conversion. Before we present the rules of the new protocol, a few definitions are in order.

Definition 4-1: Let LS(T) be the set of all items which at step t either were held by transaction T in S mode or were held in S mode just before being unlocked. The set LX(T) can be similarly defined.

An M-pitfall of T at time t is defined to be the union of the set of items in a maximal connected component of LS(T) and the set of its neighbors in LX(T).

Note that the M-pitfalls are not necessarily disjoint and that the M-pitfalls of a given transaction in no way depend upon the locking activities of other transactions.

Theorem 4-1: The M-pitfall protocol (for T) are as follows.
1) Any vertex can be locked first.
2) Subsequently, a vertex v can be locked only if there exists a subguard (A^v, B^v) satisfied in either X or S mode by T.
3) T must be generalized two-phase on its M-pitfalls. That is, T can lock or upgrade a lock on an entity e in some M-pitfall Q only if it did not unlock or downgrade a lock on any of entities in Q.

Theorem 4-1: The M-pitfall protocol assures serializability.

Proof: The proof is similar to the serializability proof of the pitfall protocol presented in [8]. For sake of brevity it is not presented in this paper.

Fig. 4 illustrates the locking activities of a transaction following the M-pitfall protocol. The following observations can be made.
1) As new locks are acquired an $M$-pitfall may both grow and shrink. Shrinkage may occur as a result of an upgrade lock conversion which splits an $M$-pitfall into several smaller $M$-pitfalls [see Fig. 4(b)].

2) Several $M$-pitfalls may merge together due to a downgrade lock conversion [see Fig. 4(f)].

3) The protocol does not require that transactions be generalized two-phase on the union of the pitfalls. Otherwise, the transition, for example, from the state in Fig. 4(c) to that in Fig. 4(d) will not be allowed.

As we have shown, it is very easy to cope with deadlocks in the case of MGLP', but the deadlock problem turns out to be a difficult one in the case of the $M$-pitfall protocol. While the deadlocks in MGLP' are found not to cause rollbacks, the deadlocks in the $M$-pitfall protocol may result in rollbacks being performed. Deadlocks are avoidable in the pitfall protocol by enumerating the vertices of the tree and requiring the transactions to lock the vertices in the increasing order of enumeration. Such an approach does not work with the $M$-pitfall protocol as stated, but it works if we exclude from the $M$-pitfall protocol the ability to do upgrading.

V. CONCLUSION

In this paper, we have argued that the study of lock conversions is of both theoretical and practical interest. In particular, we have examined the integration of lock conversion into existing non-two-phase protocols. We have presented natural generalizations of these protocols, including correctness proofs. Methods for alleviating the deadlock problem were also proposed. We have shown through examples that our results could be effectively applied to existing locking protocols to produce new protocols which potentially increase the level of concurrency supported by the database management system.

Now we briefly compare the various protocols. GLP supports only the $X$ mode of locking. The heterogeneous protocol GLP' supports both the $X$ and $S$ modes, but it requires that each transaction acquire all its locks in only one mode. The homogeneous Pitfall protocol places no such restriction on the transactions. $M$-Pitfall is an extension of the Pitfall protocol which allows lock conversions, while MGLP' is such an extension of GLP'.

We have not addressed the issue of which class of non-two-phase protocol is better, the heterogeneous or the homogeneous. The difficulty in doing this is due to the fact that while the former may reduce concurrency by forcing update transactions to start from the root of the DAG (thus potentially increasing the number of "excess vertices" that an update transaction has to lock)—by "excess vertices" we are referring to those vertices which the transaction does not need to read/write, but which all the same have to be locked since they are
"on the way" to the needed vertices), the latter may also reduce concurrency due to its two-phase requirement in pitfalls (which may result in some vertices being held in a locked state for a longer time than otherwise necessary). Another factor that needs to be considered is the deadlock problem—whether deadlocks are possible and if so how they are dealt with. The deadlocks of the heterogeneous protocols are less costly to handle (in terms of the amount of work to be done) than those of the homogeneous protocols. These tradeoffs deserve further research work.

The lock conversion approach to increasing the level of concurrency supported by protocols is not restricted to the guard protocols family of non-2PL protocols alone. A new nonguard protocol proposed in [15] has also been extended with lock conversion. We have also extended the directed hypergraph model of locking [4, 19] to include lock conversions [13].

**APPENDIX**

In this Appendix we present the proof of Theorem 3-1. Before doing so let us present some basic definitions and lemmas, the proofs of which are for the sake of brevity.

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

**Definition 7-2:** We define the "precedence" $<$ relation on a history $H$ of a set $T$ of transactions by writing $T_i < T_j$ if and only if either $T_j$ successfully locked $e$ in $X$ mode after $T_i$ unlocked it, or $T_i$ successfully locked $e$ in $X$ mode before $T_j$ locked it.

We extend $<$ by defining

$$T_i < T_j \iff \exists e [T_i < e T_j]$$

**Lemma 7-1:** A protocol assures serializability if and only if for all concurrent executions of transactions following it the associated relation $<_{\text{on}}$ on $T$ is acyclic.

**Lemma 7-2:** Let $H$ be a history consisting of the set of transactions $T = \{T_0, T_1, \cdots, T_{n-1}\}$. If each transaction $T_i$ can issue a rising instruction only if it did not issue a falling instruction, then the associated relation $<_{\text{on}}$ on $T$ is acyclic.

**Lemma 7-3:** A protocol $P$ is said to be closed under truncation if for every transaction $T$ following the protocol, the instructions referring to the last item locked by $T$ can be deleted from $T$ to form a new transaction $T'$ which also follows $P$. Note that all known protocols have this very natural property.

**Definition 7-4:** Let # denote a vertex-enumeration function $\#: V \to \{0, 1, \cdots\}$ where $V$ is the set of vertices of the rooted DAG, such that if vertex $e$ is a successor of vertex $f$, then $\#(f) < \#(e)$.

**Definition 7-5:** Let $\text{LCA}(T_i, T_j) = e$ be the lowest common ancestor of $T_i$ and $T_j$, when $L(T_i) \cap L(T_j) \neq \emptyset$. That is, the vertex $e$ satisfies the following:

$$\#(e) = \min \{\#(a) | a \in \{L(T_i) \cap L(T_j)\}\}$$

Also, let $FV(T_j)$ be the first vertex to be locked by $T_j$. Note that for update transactions $FV$ is required to be $R$.

**Lemma 7-3:** If $L(T_i) \cap L(T_j) \neq \emptyset$ then $\text{LCA}(T_i, T_j)$ is the vertex $v$ such that $\#(v) = \max \{\#(FV(T_i)), \#(FV(T_j))\}$. 

**Proof:** Simple.

**Lemma 7-4:** If $\text{LCA}(T_i, T_j)$ is the ancestor of vertex $u \in L(T_i) \cap L(T_j)$ then $T_i$ and $T_j$ traverse at least one common path from $\text{LCA}(T_i, T_j)$ to $u$.

**Proof:** Simple.

**Lemma 7-5:** Let $u$ be a descendant of $\text{LCA}(T_i, T_j)$ as above. For any history involving $T_i$ and $T_j$, if $T_i < u T_j$ then $T_i < \text{LCA}(T_i, T_j) T_j$.

**Proof:** Suppose that $u$ is a son of $\text{LCA}(T_i, T_j)$. Since $T_i < u T_j$, then either $T_i$ or $T_j$ or both obtained $X$ mode locks on $u$ in the history. By the rules of the protocol, any transaction acquiring an $X$ lock on $u$ must have previously obtained an $X$ lock on $\text{LCA}(T_i, T_j)$. Thus either $T_i < \text{LCA}(T_i, T_j) T_j$ or $T_j < \text{LCA}(T_i, T_j) T_i$. By the rules of the protocol, $T_i$ must request a lock on $u$ before releasing $\text{LCA}(T_i, T_j)$. But $T_i$ locks $u$ before $T_j$ does, and must do so after locking $\text{LCA}(T_i, T_j)$. It can only have done this before $T_j$ locked it due to the incompatibility of the locking modes involved, and thus it must be the case that $T_i < \text{LCA}(T_i, T_j) T_j$.

This argument forms the basis for a proof of the general result by induction on the length of a path from $\text{LCA}(T_i, T_j)$ to $u$ traversed in common by $T_i$ and $T_j$ as in Lemma 7-4. Assume the result holds for all such paths of length $k$. If $u$ is the $k + 1$st vertex on such a path, and $w$ is $u$'s father on the path, we can see that $T_i < u T_j$ implies $T_i < w T_j$ by the argument used for the basis. By the induction hypothesis $T_i < w T_j$ implies $T_i < \text{LCA}(T_i, T_j) T_j$ and the result is obtained.

**Proof of Theorem 3-1:** By Lemma 7-1, a nonserial history contains a cycle in $<$. Without loss of generality, let this cycle be minimal and of the form

$$T_0 < u_1 T_1 < u_2 \cdots < u_{m-1} T_{m-1} < u_0 T_0$$

We can further choose each $u_i$ to be the first vertex locked in common by $T_{i-1}$ and $T_i$ on which this precedence is established. (All arithmetic on all subscripts is mod $m$.) By Lemma 7-5, then, $u_i = \text{LCA}(T_{i-1}, T_i)$. Let $k$ be the number of read-only transactions in the cycle. We show that the cycle cannot exist by induction on $k$.

If $k = 0$, then no read-only transactions are involved. Thus, for each $T_i$, $FV(T_i) = R$. By Lemma 7-3, each $u_i$ is therefore $R$. Thus we have

$$T_0 < R T_1 < R \cdots < R T_{m-1} < R T_0$$

Since the protocol is closed under truncation, we can replace each $T_i$ with $T'_i$, a truncated transaction referencing only $R$ and find a history in which

$$T'_0 < R T'_1 < R \cdots < R T'_{m-1} < R T'_0$$

This is impossible due to Lemma 7-2. Hence the original cycle cannot exist.

Assume that no cycle containing $k$ read-only transactions exists, and without loss of generality let $T_1$ be one of the read-only transactions in a cycle containing $k + 1$ of them. Then we have

$$T_0 < FV(T_0) T_1 < FV(T_1) T_2 < \cdots < T_{m-1} < T_0$$

where $T_0$ and $T_2$ must be update transactions which both lock $FV(T_1)$ in $X$ mode. But then $T_0$ must have locked $FV(T_1)$ before $T_2$ did. If $T_0 = T_2$, then this transaction would have had to lock $FV(T_1)$ once before and once following $T_1$, which is clearly impossible. If there are at least three transactions in
the cycle, then $T_0 < F V (T_1) T_2$ and we have the cycle

$$T_0 < T_2 < \cdots < T_{m-1} < T_0$$

with only $k$ read-only transactions, contradicting the induction hypothesis and establishing the result.

**References**


C. Mohan received the B.Tech. degree from the Indian Institute of Technology, Madras, India, in 1977, and the Ph.D. degree in computer science from the University of Texas at Austin in 1981.

He is a Research Staff Member at the IBM San Jose Research Laboratory, where he is conducting research in the areas of concurrency control, deadlock management, fault tolerance, and data distribution. Currently, he is a member of the design and implementation team of the R* distributed database management system. He has been a Visiting Scientist at two European laboratories, INRIA and Hahn-Meitner-Institut. He has lectured and published papers in the areas of distributed data management, distributed programming languages, design methodologies, operating systems, and distributed control. He has coauthored chapters in books on distributed and database systems. He is the editor of the tutorial text *Recent Advances in Distributed Data Base Management* to be published by the IEEE Computer Society Press. He has refereed papers for numerous journals and conferences, and has served on the Program Committees of COMPCON.

Dr. Mohan is a member of the IEEE Computer Society and the Association for Computing Machinery.

Donald Fussell received the B.A. degree from Dartmouth College in 1973, and the M.S. and Ph.D. degrees in computer science from the University of Texas at Austin in 1977 and 1980, respectively.

Since 1980 he has an Assistant Professor of Computer Science at the University of Texas at Austin. His research interests include database concurrency control, VLSI systems design, and computer graphics. He has published over 20 journal and conference articles on graph-based database locking protocols, modeling and techniques for realistic image generation, fault-tolerant and special-purpose VLSI architectures, and wafer-scale integrated circuits.

Dr. Fussell is a member of the Association for Computing Machinery and the IEEE Computer Society.

Zvi M. Kedem (M'82) received the B.Sc. degree in 1967, the M.Sc. degree in 1969, and the D.Sc. degree in 1974, all from the Technion—Israel Institute of Technology, Haifa.

From 1975 to 1980 he was on the faculty of the Programs in Mathematical Sciences at the University of Texas at Dallas. Since 1980 he has been on the faculty of the Department of Computer Science at the State University of New York at Stony Brook, currently on leave to the faculty of the Department of Computer Science at New York University—Courant Institute. His current research interests include database systems. VLSI complexity, and parallel computations.

Abraham Silberschatz received the Ph.D. degree in computer science from the State University of New York, Stony Brook, in 1976.

He is currently a Professor of Computer Science at the University of Texas at Austin, specializing in the area of concurrent processing. His research interests include operating systems, database systems, distributed systems, and programming languages. He has published over 60 journal and conference papers on such topics as language-based access control schemes, interprocess communication facilities, and graph-based locking protocols. He is also the coauthor of a recently published textbook entitled *Operating System Concepts*.

Dr. Silberschatz received the 1978 IEEE Computer Society Outstanding Paper Award for the paper “Capability Manager,” which appeared in *IEEE Transactions on Software Engineering*. 