Compatibility and Commutativity of Lock Modes*

C. Mohan, D. Fussell, and A. Silberschatz

Department of Computer Sciences, University of Texas at Austin,
Austin, Texas 78712

An important characteristic of concurrency control mechanisms is the level of concurrency that they support. In this paper, we study this problem in the context of non-two-phase locking protocols which are defined for databases in which a directed acyclic graph structure is superimposed on the data items. A new lock mode is introduced called INV, with properties fundamentally different from lock modes previously studied and show how this allows increased concurrency. Through the introduction of the INV mode of locking, a new principle of the theory of database concurrency control is enunciated. This principle involves the separation of the effects of commutativity (which relates to serializability) and compatibility (which relates to deadlock-freedom of data manipulation operations. It is shown how the level of concurrency in an existing very general protocol could be increased. Then how the extension affects the occurrence of deadlocks is examined. Certain conditions under which deadlock-freedom is maintained are identified, and simple methods for removing deadlocks in other situations are presented. © 1984 Academic Press, Inc.

1. INTRODUCTION

Data base systems usually interleave the operations (read and write) of different transactions due to performance considerations. Safeguarding the consistency of the stored/retrieved data is of great significance under such circumstances. The widely accepted approach to dealing with this problem is to define a transaction as a unit that preserves consistency (i.e., it is assumed that each transaction, when executed alone, transforms a consistent state of the database into a new consistent state), and require that the outcome of processing a set of transactions concurrently be the same as the one produced by running these transactions one at a time (i.e., serially) in some order. A system that ensures this property is said to be serializable (Eswaran et al., 1976). Another important issue in data base management is the problem of deadlocks. Deadlocks arise as a result of circular wait conditions involving two or more transactions. A system which does not allow deadlocks to occur is said to assure deadlock-freedom.

* This research was partially supported by ONR Contract N00014-80-K-0987 and NSF Grant MCS81-04017. C. Mohan was also supported by an IBM Pre-Doctoral Fellowship.

† Current address: IBM Research Laboratory, K55/281, San Jose, CA 95193.
Serializability can be ensured via a number of concurrency control mechanisms, the most common one being a locking protocol. Such a protocol can be simply viewed as a restriction on when a transaction may lock and unlock each of the data base items. Locking a data item in a certain mode inhibits certain kinds of concurrent activity on that item until the item is unlocked. The first useful locking protocol developed was the 2-phase locking (2PL) protocol (Eswaran et al., 1976) which is characterized by the fact that a transaction is not allowed to lock a data base item after it has unlocked any other item. One drawback of the 2-phase protocol is that it severely restricts the amount of concurrency allowed in a system.

This deficiency has led to the development of a number of non-two-phase locking protocols (Kedem & Silberschatz, 1979, 1980, 1983; Silberschatz & Kedem, 1980, 1982; Yannakakis, Papadimitriou, & Kung, 1979). One of the most general non-2PL protocols is the guard protocol (GLP), presented in (Silberschatz & Kedem, 1982). This protocol assures serializability and deadlock-freedom. It is a versatile protocol in the sense that several previously developed protocols (e.g., the tree protocol, Silberschatz & Kedem, 1980; majority protocol, Kedem & Silberschatz, 1979; and the DAG protocol, Yannakakis, Papadimitriou, & Kung, 1979) are special cases of it.

Throughout our work we adopt serializability as the correctness criterion for locking protocols. As for concurrency, some protocols may have the following undesirable features, which may potentially reduce the level of allowable concurrent access to data:

1. Transactions required to hold locks longer than they need them (e.g., 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. Transactions forced to acquire unnecessary access privileges (e.g., when a shared (S) mode lock would suffice, a transaction being forced to acquire an exclusive (X) mode lock).
3. Transactions forced to acquire unnecessary locks, i.e., locks on items the transactions do not need access to.

In our work, we try to reduce the frequency of occurrence of these undesirable situations in a variety of protocols.

As for deadlocks, we examine each protocol to check if deadlocks are possible. If they are, we make sure that deadlock detection and/or recovery is easy and is not too costly. In many cases, we propose additional conditions which guarantee freedom from deadlocks.

1 The metric we use in this paper for amount of potential concurrency is the one introduced in (Kung & Papadimitriou, 1979).
We introduce a new lock mode, called INV, in an extended version of GLP and show how this leads to increased concurrency. Through the introduction of the INV mode of locking, which does not grant any access privileges (read/write) to the holder of the lock on the associated data item, we have enunciated a new principle of the theory of data base concurrency control. This principle involves the separation of the effects of commutativity (which relates to serializability) and compatibility (which relates to deadlock-freedom) of data manipulation operations. Thus we depart from the traditional approaches to concurrency control which do not make this separation. We also see how the introduction of such a locking mode affects the problem of maintaining deadlock-freedom in a system and show how this problem can be handled.

It should be emphasized that none of the earlier work on non-two-phase protocols has concerned itself with the problems caused by transaction and system failures. This paper is no exception. It should be clear that allowing transactions to release locks before the end of the transaction executions could cause serious problems (by way of cascading rollbacks) if failures are considered.

The rest of this paper is organized as follows. In Section 2 we introduce the invisible (INV) lock mode. In Section 3 we discuss some existing protocols and motivate the need for the INV mode. Then in Section 4 we present the super guard protocol (SGP) which allows transactions to lock data items in INV, S, and X modes. SGP is shown to assure serializability. Section 5 treats the issue of deadlocks in SGP which is shown to be deadlock-free for rooted trees but not for more general DAG's. Simple and effective methods for handling deadlocks when they occur are given.

2. Lock Modes and Their Properties

Let a data base consist of a set of data items $V$. We ignore the exact nature of the granularity of the items, but it should be noted that we do not consider in our work protocols supporting a variable granularity of locking (Gray, 1978; Korth, 1981) and those supporting lock conversions (Korth, 1981; Mohan et al., in press). Associated with the data base is a set of consistency constraints, the exact nature of which is not of concern to us. A state of the data base is an assignment of values to the elements of $V$. A given data base state is said to be consistent if that state satisfies the consistency constraints.

One of the components of the data base system is a lock manager which receives and processes the lock requests of the transactions. This means that once a transaction issues a lock request, it cannot proceed until that lock has been granted. In all the protocols to be presented here, locks can only be
obtained one at a time. Further, once a data item is unlocked by a transaction it cannot be relocked by that transaction. Each item can be locked only once.

We assume that the transactions are well-formed; that is, every action that a transaction performs is permitted by the locks that it holds at the time the action is performed and all locks held by a transaction are released by the transaction before it terminates. We also assume that every transaction when run alone on a consistent state of the data base transforms the latter into a consistent state.

The protocols that we present in this paper allow three modes of locking: X (exclusive), S (shared), and INV (invisible). These modes can be obtained via the LX, LS, and LINV instructions, respectively. Locks can be released using the UN instruction. If a transaction holds an X mode lock on a data item, it can read and modify that data item, and if it holds an S mode lock, it can only read the data item. When holding an INV mode lock, the transaction can neither read nor modify the data item (i.e., the data item is invisible to the transaction). While the INV mode is a kind of intention mode in the sense that it conveys some information about what kind of locks a transaction may acquire in the future, it bears no further resemblance to the intention modes of Gray (1978) and Korth (1981), which are used to support variable granularity of locking. The motivation for the introduction of the INV mode will become apparent to the reader when we present an example in Section 3.

Since we assume that transactions are well formed we need to consider only the locking activities of transactions to study the serializability and deadlock-freedom properties. Hence we define a transaction $T$ by listing the finite sequence of lock and unlock instructions issued by $T$. We consider only protocols which assure the serializability property. A protocol is said to assure serializability only if the effect of concurrent executions of transactions permitted by the protocol is equivalent to some serial execution of those transactions.

Given a set of locking modes, and the compatibility relation among them, the lock manager behaves as follows. Suppose that a transaction $T_i$ requests a lock of mode $A$ on item $e$ on which transaction $T_j$ currently holds a lock of mode $B$. The lock manager is allowed to grant $T_i$'s request in spite of the presence of the mode $B$ lock, if mode $A$ is compatible with mode $B$. Such a relation, particularly on a relatively few locking modes, can conveniently be represented by a matrix. The compatibility relation among the 3 modes of locking used in this paper is given by the matrix, COMP (Fig. 2.1). An element, say $COMP(I, J)$, of the matrix has the value $T$ if and only if mode $I$ is compatible with mode $J$.

Note that INV mode is compatible with X mode, but not with S mode. At any time one X mode lock and zero or more INV mode locks can be
simultaneously held (by different transactions) on a particular data item. A subsequent X mode lock request has to wait until the currently held X mode lock is released.

Another important characteristic of a set of locking modes is the commutativity relation among them. We say locking modes \( A \) and \( B \) are commutative if all operations which a transaction is allowed to perform while holding a mode \( A \) lock on an item and all operations which are allowed under a mode \( B \) lock are commutative. Thus, the results produced on an item by two transactions, one holding a mode \( A \) lock and the other a mode \( B \) lock, will be identical regardless of the order in which the transactions acquire locks on the item. As with the compatibility relation, the commutativity relation is conveniently represented by a matrix. The commutativity matrix \( \text{COMM} \) for the locking modes used here is given in Fig. 2.2. \( \text{COMM}(I, J) = T \) if modes \( I \) and \( J \) are commutative. For example, mode \( S \) is not commutative with mode \( X \). If one operation reads data item \( v \) and a subsequent operation modifies \( v \), then changing the order of these two operations must be assumed to result in a different value being read by the read operation. On the other hand, mode \( S \) is commutative with mode \( S \) since changing the order of reads will not change the value read. Notice that even though the \( S \) and \( \text{INV} \) modes are incompatible, they are still commutative.

In previous research on locking, the \( \text{COMP} \) and \( \text{COMM} \) matrices were assumed to be identical, as are the matrices presented in this section if we

\[
\begin{array}{ccc}
X & S & \text{INV} \\
X & T & \\
S & T & T \\
\text{INV} & T & T \\
\end{array}
\]

**Figure 2.1.** Lock compatibility matrix \( \text{COMP} \)

\[
\begin{array}{ccc}
X & S & \text{INV} \\
X & T & \\
S & T & T \\
\text{INV} & T & T \\
\end{array}
\]

**Figure 2.2.** Lock commutativity matrix \( \text{COMM} \)

\(^2\)This commutativity definition can be stated formally in terms of the Herbrand interpretation of the changes in values of data items (Manna, 1974).
ignore the rows and columns corresponding to the INV mode. Because existing techniques for proving the serializability and deadlock-freedom of locking protocols depend on this fact, we have been forced to augment them with some new methodologies.

We now introduce some standard definitions which will be required in the subsequent technical discussion.

**Definition 2.1.** A history $H$ is the trace in chronological order of the concurrent execution of a set of transactions $\mathcal{T} = \{T_0, \ldots, T_{n-1}\}$.

**Definition 2.2.** We define the $<$ and $\preceq$ relations (the "precedes" relations) on a history $H$ of a set $\mathcal{T}$ of transactions as:

$T_i < T_j \iff T_i$ has held an $M_i$ mode lock on $e$ initially and $T_j$ has held an $M_j$ mode lock on $e$ later, and $\text{COMM}(M_i, M_j) \neq T$.

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

We note that the $<$ relation (the "precedes" relation) pertains to the serializability of the history $H$. In the above case, it means that in an equivalent serial history $T_i$ must appear before $T_j$. Note the close relationship between the commutativity relation on lock modes and the precedence relation on histories.

**Proposition 2.1.** A protocol assures serializability if and only if for all concurrent executions of transactions following it the associated relation $<$ is acyclic.

**Definition 2.3.** We define the $\rightarrow_e$ and $\rightarrow$ relations (the "wait-for" relations) as:

$T_i \rightarrow_e T_j \iff T_j$ is currently holding a lock on $e$ in $M_j$ which is incompatible with the mode $M_i$ in which $T_i$ has requested a lock on $e$ (i.e., $\text{COMP}(M_i, M_j) \neq T$).

$T_i \rightarrow T_j \iff \exists e \ [T_i \rightarrow_e T_j]$.

Note that the relationship between the compatibility relation among locking modes and the waits-for relation on histories is similar to that noted above for commutativity and precedence.

**Proposition 2.2.** A protocol assures deadlock-freedom if and only if for all concurrent executions of transactions following it the associated relation $\rightarrow$ is acyclic, at any instance of time during the execution.

The proofs of Propositions 2.1 and 2.2 can be found in (Ullman, 1980).
3. Existing Protocols

The guard locking protocol (GLP) for data bases in which the data items are organized as directed acyclic graphs (DAGs) was presented in (Silberschatz & Kedem, 1982). It allows transactions to obtain only X mode locks. Here we restrict our attention to the version of GLP for rooted DAGs.

**Definition 3.1.** Let \( V \) be the set of vertices of the rooted DAG with the vertex \( R \) being the root and each vertex a data item. We shall say that this is a guarded graph if and only if with each \( v \) (except \( R \)) \( \in V \) we associate a non-empty set of pairs (subguards),

\[
\text{guard}(v) = \{ (A^v_i, B^v_i), \ldots, (A^v_{M_r}, B^v_{M_r}) \}
\]

satisfying the conditions:

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

**Definition 3.2.** We shall say that a subguard \( (A^v_i, B^v_i) \) is satisfied in mode \( m \) (where \( m \) is X, S, or INV) 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 \( m \). A guard is said to be satisfied if at least one of its subguards is satisfied in mode \( m \).

**The rules of the guard protocol** (for transaction \( T \) are:

1. Any vertex may be locked at 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 3.1.** The guard protocol assures serializability and deadlock-freedom.

**Proof.** See (Silberschatz & Kedem, 1982).

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 as special cases (see Silberschatz & Kedem, 1982, for many examples) of GLP. For example, the DAG protocol of (Yannakakis, Papadimitriou, & Kung, 1979) 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. DP can be obtained by defining the guards as follows:

\[
guard(v) = \{ \langle F(v), \{ u \} \rangle | u \in F(v) \}.
\]

The non-2PL protocols which support both X and S modes of locking can be broadly classified into two types: **heterogeneous** and **homogeneous**. The heterogeneous protocols are characterized by distinctions being made between read-only transactions (those that need to only read the data base and hence need to acquire only S mode locks) and update transactions (those that need to modify the data base and hence need to acquire X mode locks). The homogeneous protocols do not make such distinctions. The latter protocols require transactions that acquire S mode locks to follow the 2-phase protocol during certain periods of their locking activities. In the rest of this paper, we are concerned with only the heterogeneous protocols.

The reader can find some interesting discussions and results concerning protocols that support more than one mode of locking in (Fussell, Kedem, & Silberschatz, 1981b; Yannakakis, 1982). Sufficient and necessary conditions for protocols to assure serializability and deadlock-freedom are presented there.

In (Kedem & Silberschatz, 1983), a set of conditions were described which can be applied to GLP to produce a new protocol, called GLP', which allows both the X and S modes of locking. In this protocol, the update (respectively, read-only) transactions are allowed to acquire only X (S) mode locks. Further, the update transactions are required to start by locking the root of the DAG first. In this case, 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 to be allowed to start by locking any vertex first then serializability cannot be granted. To illustrate this point, consider the database graph of Fig. 3.1, with the set of guards:

\[
\begin{align*}
guard(r) &= \{ \} \\
guard(a) &= \{ \langle \{ r \}, \{ r \} \rangle \} \\
guard(b) &= \{ \langle \{ a \}, \{ a \} \rangle \}.
\end{align*}
\]

Consider the following history consisting of four transactions:

\[
\begin{align*}
T_1 &\text{ LS } r; T_1 \text{ LS } a; T_1 \text{ UN } r; T_2 \text{ LX } r; T_2 \text{ UN } r; T_3 \text{ LS } r; \\
T_3 &\text{ LS } a; T_3 \text{ LS } b; T_3 \text{ UN } r; T_3 \text{ UN } a; T_3 \text{ UN } b; T_4 \text{ LX } b; \\
T_4 &\text{ UN } b; T_1 \text{ LS } b; T_1 \text{ UN } a; T_1 \text{ UN } b;
\end{align*}
\]
This history is nonserializable since the cycle $T_1 <_r T_2 <_r T_3 <_b T_4 <_b T_1$ exists.

In this example $T_3$ starts by locking the nonroot vertex $b$. Although $T_2$ locks only $r$ and $T_4$ only $b$ (and thereby do not lock any common vertices) still a precedence ordering is forced between them ($T_3$ has to precede $T_4$). This happens because of $T_1$ which locks both $r$ and $b$. $T_1$ “waits-around” by unlocking $r$ and holding the lock on $a$. After $T_4$ locks and unlocks $b$, $T_1$ locks $b$. If only GLP’ were to have been followed then $T_4$ would have been forced to start by locking $r$ and it would have had to wait for $T_1$ to release the locks on $a$ and $b$ before it ($T_4$) could lock $b$. So $T_4$ would have preceded $T_4$ and the history would have been serializable. Intuitively stated, $T_4$ by acquiring X mode locks starting with $r$ “would have pushed forward” $T_1$ and would have prevented the latter from “waiting around.”

Let us illustrate how some potential concurrency is lost due to the requirements of GLP’. Suppose at a certain point in time, transaction $T_1$, which needs to update only the root $r$, obtains an X mode lock on $r$. Immediately after that, transaction $T_2$ starts, and it needs to update only $a$, where $a$ is an immediate successor of $r$. $T_2$ is required to acquire an X mode lock on $r$ before it can lock $a$. As a consequence of this requirement, $T_2$ is forced to wait until $T_1$ finishes its updating of $r$. Thus some potential concurrency is lost. What we would like to see happen is $T_2$ being allowed to “overtake” (or “step-over”) $T_1$ and lock $a$. In this way, both $T_1$ and $T_2$ will be able to perform their updates in parallel. It is only to provide this “overtaking” ability to update transactions, which could potentially lead to an increased level of concurrency, we have introduced the INV mode of locking. In the next section we present our results.

To provide some intuition concerning the compatibility and commutativity
properties of the INV mode we need to reconsider the above examples. Since the INV mode (which was designed to provide the "overtaking ability") does not allow reading and writing of the associated data item it should be obvious as to why the INV mode is commutative with the X, INV, and S modes. The idea of the INV mode came about when we closely examined what caused the nonserializability in the first example. As pointed out before we had to "push-forward" \( T_1 \). We felt that the solution embodied in GLP' was too severe (it was sufficient but not necessary). We had to make \( T_3 \) acquire a lock on \( A \) which was incompatible with the S mode lock hold on \( A \) by \( T_1 \). To provide the "overtaking" ability (in the case of the second example), we wanted \( T_2 \) to acquire a lock on \( R \) which was compatible with the X mode lock held on \( R \) by \( T_1 \). Hence we defined INV to be compatible with X and INV, but not with S.

4. The Super Guard Protocol

The super guard protocol (SGP) is intended for data bases organized as rooted DAGs. We classify the transactions that access the data base into two types:

1. **Update transactions**—Those transactions that can issue INV, X, and S mode lock requests (at least one X mode lock request must be issued).

2. **Read-only transactions**—Those transactions that can issue only S mode lock requests.

The rules of the SGP protocol are:

(a) **Read-only** transactions may start anywhere. They should obey the guard protocol rules stated in the last section.

(b) **Update** transactions:

(1) Each transaction must start at the root; it may lock the root in either INV or X mode.

(2) Subsequently, a vertex \( v \) can be locked in mode

- **INV**—only if the guard of \( v \) is satisfied in INV mode and the first X mode lock has not been acquired so far.
- **X**—only if (a) the guard of \( v \) is satisfied in X mode or (b) this is the first X mode lock request, all the vertices so far locked (in INV mode) are predecessors of \( v \) and the guard of \( v \) is satisfied in INV mode.
- **S**—only if the guard of \( v \) is satisfied in X or S mode.

(3) Vertices may be unlocked any time.
We now make some observations which should provide some insight into the workings of the protocol and help in understanding the proof of the protocol.

(1) All the vertices locked in INV mode, if any, will span a rooted DAG, with R as the root. Furthermore, if we take any update transactions acquiring INV mode locks, remove all X and S mode lock and unlock instructions in it and convert the INV mode lock requests to S mode lock requests then the resulting read-only transaction will obey the SGP protocol.

(2) All the vertices locked in X mode will span a rooted DAG, with the first vertex to be locked in X mode as the root. Furthermore, if we take any update transaction, remove all INV and S mode lock and unlock instructions in it and convert the X mode lock requests to S mode lock requests then the resulting read-only transaction will be one that obeys the SGP protocol.

(3) Once the first X mode lock request has been issued no more INV mode lock requests can be issued. While this condition is not required for assuring serializability it guarantees deadlock-freedom when the rooted DAG is a tree, as will be shown later on.

(4) In contrast to the vertices locked in INV and X modes, all the vertices, if any, locked by an update transaction in S mode need not necessarily span only a single connected component. If we remove from an update transaction all instructions except those referring to vertices which form a single connected component of S mode locks then the resulting read-only transaction may not obey the SGP protocol. On the other hand, if we remove only the INV mode lock and unlock instructions and convert the X mode locks to S mode locks then the resulting read-only transaction will be one that obeys the SGP protocol.

(5) Every vertex locked in INV mode must be a predecessor of some other vertex locked in S or X mode. This observation follows from the restrictions on when INV and X mode locks can be acquired and the requirement that at least one X mode lock must be acquired by an update transaction. Thus no "useless" INV mode locking is allowed. While this condition is not necessary for assuring serializability it guarantees deadlock freedom in the case of rooted trees, as will be shown later on. The INV mode locks are "helpful" only to get the first X mode lock.

(6) All vertices locked in S mode by an update transaction must be successors of at least the first vertex to be locked in X mode.

(7) As mentioned before, GLP' does not allow S mode locks to be acquired by update transactions. Even if we place such a restriction on update transactions following SGP, SGP will still be an extension of GLP'.
that potentially supports more concurrency (by providing the INV mode of locking) than GLP'.

Figure 4.1 illustrates some of these observations. To give some feeling for why SGP is superior to GLP' let us consider a rooted tree (with a root R) and a transaction T which needs to modify the set of vertices M. Let N be the least common ancestor of the vertices in M. Let us assume N is not the root R. If T were to follow GLP' then the vertices on the path from R to N would have to be locked in X mode. On the other hand if SGP were to be used then all the vertices along that path (except N) need to be locked only in INV mode, thus permitting other transactions to acquire X mode locks on those vertices; T would have to acquire its first X mode lock only on N.

We claim that SGP supports more concurrency than GLP' for the following reasons: Any transaction which was designed to follow GLP' (on a particular rooted DAG) can be run under SGP without any change (on the same DAG). In addition to those transactions, SGP permits other transactions also (update transactions with INV and/or S mode locks). Compared to GLP', SGP permits an update transaction to "overtake" other update transactions. Unlike the former, it does not force the update transactions to be serialized according to the order in which they acquired a lock on the root R.

**Theorem 4.1.** The super guard protocol assures serializability.

**Proof:** Before proceeding with the proof we need to state some basic definitions and lemmas, the proofs of which are omitted or outlined for the
sake of brevity. In the sequel, all the transactions that we discuss obey the SGP protocol.

**Definition 4.1.** Let \( \# \) denote a vertex-enumeration function \( \#: V \rightarrow \{0, 1, \ldots\} \), 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 4.2.** Let \( L(T_i) \) be the set of all vertices locked by \( T_i \), \( LINV(T_i) \) be the set of vertices locked in INV mode by \( T_i \), \( LX(T_i) \) be the set of vertices locked in X mode by \( T_i \) and \( LS(T_i) \) be the set of vertices locked in S mode by \( T_i \). A lock is held on each member of these sets by \( T_i \) during its execution.

**Definition 4.3.** Let \( LCA(T_i, T_j) = e \) be the lowest common ancestor of \( L(T_i) \) and \( L(T_j) \), when \( L(T_i) \cap L(T_j) \neq \emptyset \). That is, the vertex \( e \) satisfies

\[
\#(e) = \min \{ \#(a) \mid a \in \{L(T_i) \cap L(T_j)\} \}.
\]

**Definition 4.4.** When \( T \) is an update transaction let \( FX(T) \) be the first vertex to be locked in X mode by transaction \( T \) (If this vertex is \( e \) then \( \#(e) = \min \{ \#(v) \mid v \in LX(T) \cup LS(T) \} \)).

When \( T \) is a read-only transaction let \( FS(T) \) be the first vertex to be locked in S mode (If this vertex is \( f \) then \( \#(f) = \min \{ \#(v) \mid v \in LS(T) \} \)).

Let \( FV(T) \) be the first vertex to be locked by transaction \( T \) in a non-INV mode. \( FV(T) = FX(T) \), if \( T \) is an update transaction. Otherwise, \( FV(T) = FS(T) \).

**Lemma 4.1.** If \( u \) and \( w \) are distinct vertices in \( L(T_1) \cap L(T_2) \) then there exists a (undirected) chain in the underlying graph between \( u \) and \( w \) which lies entirely in \( L(T_1) \cap L(T_2) \).

This lemma follows from condition (3) of the guard definition (3.1) when we consider all chains connecting \( u \) and \( w \) and the subguards satisfied by \( T_1 \) and \( T_2 \). The next 4 lemmas follow from condition (3) and the observations by considering the common vertices in the subguards satisfied in locking a vertex that is common to more than one transaction.

**Lemma 4.2.** If \( LX(T_1) \cap LS(T_2) \neq \emptyset \) then

1. if \( T_2 \) is a read-only transaction then \( FS(T_2) \in LX(T_1) \cup LINV(T_1) \),
2. if \( T_2 \) is an update transaction then

\[
FX(T_2) \in LX(T_1) \cup LINV(T_1) \text{ and } FX(T_1) \in L(T_2).
\]
LEMMA 4.3. If $\text{LX}(T_1) \cap \text{LX}(T_2) \neq \emptyset$ then

\[
\text{FX}(T_2) \in \text{LX}(T_1) \cup \text{LIN}(T_1)
\]
\[
\text{FX}(T_1) \in \text{LX}(T_2) \cup \text{LIN}(T_2)
\]

and either

\[
\text{FX}(T_2) \in \text{LX}(T_1) \text{ or } \text{FX}(T_1) \in \text{LX}(T_2).
\]

LEMMA 4.4. If $T_1$ and $T_2$ are update transactions such that $\text{LS}(T_1) \cap \text{LS}(T_2) \neq \emptyset$ then either $\text{FX}(T_1) \in \text{LX}(T_2) \cup \text{LS}(T_2)$ or $\text{FX}(T_2) \in \text{LX}(T_1) \cup \text{LS}(T_1)$.

LEMMA 4.5. If $T_2$ is a read-only transaction with $\text{FV}(T_2) = v$ and if $T_1$ is an update transaction such that $v \in \text{LX}(T_1)$ and no successor of $v$ is in $\text{L}(T_1)$, then there exists a transaction $T_3$ following SGP such that $\text{L}(T_3) = \text{L}(T_1) \cup \text{L}(T_2)$, $\text{LIN}(T_3) = \text{LIN}(T_1)$, $\text{LX}(T_3) = \text{LX}(T_1)$, and

\[
\text{LS}(T_3) = \text{LS}(T_1) \cup \text{LS}(T_2) - \{v\}.
\]

Proof of Theorem 4.1. Assume by contradiction the existence of a minimal cycle (of length $n$) of the form

\[
T_0 < T_1 < \cdots < T_{n-1} < T_0.
\]

Truncate each transaction $T_i$ to $T'_i$ so that $\text{L}(T'_i)$ has the minimal number of elements for the cycle to still exist. Note that these transactions also obey the SGP protocol. The history consists of only these transactions. This gives us

\[
T'_0 < u_0 T'_1 < u_1 \cdots < u_{n-2} T'_{n-1} < u_{n-1} T'_0.
\]

Remember that $T'_i < u_i T'_{i+1}$ means that at least either $T'_{i+1}$ or $T'_i$ had locked $u_i$ in X mode. The $u_i$s in the cycle are chosen in such a way that (when $n > 2$) for no predecessor $v$ of $u_i$, $T'_i < u_v T'_{i+1}$.

Non-Distinct Vertices

It is possible for $u_i$ and $u_{i-1}$ to be the same vertex $u$, giving us

\[
T'_{i-1} < u T'_i < u T'_{i+1}.
\]

This can happen only when $T'_{i-1}$ and $T'_{i+1}$ had locked $u$ in S mode and $T'_i$ had locked $u$ in X mode. Note that no other vertex $u_j$ can be equal to $u$ since in that case either $T'_j$ or $T'_{j+1}$ must have locked $u (= u_j)$ in X mode, thus resulting in a shorter cycle with $T'_i < T'_j$ or $T'_j < T'_i$, thus contradicting the assumption about the minimality of the cycle.
Note that in the above case $T_i'$ must be an update transaction, while either $T_{i-1}'$ or $T_{i+1}'$ must be a read-only transaction. The latter condition follows from Lemma 4.4 and the assumption about the minimality of the cycle.

We proceed with the proof by inducting on the length $(n)$ of the minimal cycle. In the sequel, for brevity we drop the superscript (') of the transactions and write $T_i$ for each transaction $T_i'$. Thus the minimal cycle becomes

$$T_0 < u_0 T_1 < u_1 \cdots < u_{n-2} T_{n-1} < u_{n-1} T_0.$$

**Basis Step**

As the basis of the induction we make $n = 2$ and assume the existence of the cycle

$$T_0 < u T_1 < u T_0.$$  

Now consider the vertex $w$ such that $\#(w) = \max\{\#(FV(T_0)), \#(FV(T_1))\}$. From Lemma 4.1 there must be paths, common to $T_0$ and $T_1$, from $u$ and $v$ to $w$. From the rules of the protocol it should be clear that if a transaction had locked $u$ or $v$ in $X$ mode then it must have locked $w$ also in $X$ mode. The locks acquired by $T_0$ and $T_1$ on the nodes in the common paths must be in noncommutative modes. By "pushing" the conflict $T_0 < u T_1$ along the common path to $w$ we will get $T_0 < w T_1$. Similarly by "pushing" the conflict $T_1 < w T_0$ we will get $T_1 < w T_0$. Thus we will get a contradiction and hence the impossibility of a cycle of length 2. Note that in Section 2 we pointed out that all the protocols that we consider do not allow an item to be relocked by a transaction after the transaction had unlocked that item.

**Induction Step**

Notice that each $u_i$ has to be either $FV(T_i)$ or $FV(T_{i+1})$ and that for each $T_i$, $FV(T_i)$ is $u_i$ (only if $u_i = u_{i-1}$ or if $u_{i-1}$ is a successor of $u_i$), $u_{i-1}$ (only if $u_{i-1} = u_i$ or if $u_i$ is a successor of $u_{i-1}$), or a predecessor of $u_i$ due to the way each $u_i$ is chosen. The proof of the induction step can be divided into two major cases.

**Case 1.** Cycles involving only distinct $u_i$s. Pick that $u_j$ for which $\#(u_j) = \max\{\#(u_i), k = 0 \cdots n - 1\}$. It must be that $u_j \in \{FV(T_j), FV(T_{j+1})\}$. From the way $u_j$ has been chosen it must be that $\#(u_{j-1}) < \#(u_j)$ and that $\#(u_{j+1}) < \#(u_j)$. This means that $FV(T_j)$ must be $u_{j-1}$ or a predecessor of $u_{j-1}$ and $u_j$, and $FV(T_{j+1})$ must be $u_{j+1}$ or a predecessor of $u_j$ and $u_{j+1}$. It is clear that neither $FV(T_j)$ nor $FV(T_{j+1})$ equals $u_j$. Thus we get a contradiction. Hence the impossibility of the cycle.

**Case 2.** Cycles involving one or more pairs of nondistinct vertices. It is this interesting case that we consider below. The strategy that we adopt is,
given a minimal cycle of length \( n \), to show that by merging two adjacent transactions in the cycle we can produce a history with a minimal cycle of length \( n - 1 \), thus contradicting the induction hypothesis. Our contributions to the area of serializability proof methodologies are the techniques of modification of a transaction by changing the modes in which locks are requested and merging of transactions.

Let us consider a pair of nondistinct vertices as described above. We have

\[
T_{i-1} <_u T_i <_u T_{i+1}
\]

and the history \(^3\)

... \( T_{i-1} \) LS \( u \);... \( T_{i-1} \) UN \( u \);... \( T_i \) LX \( u \);..., 
\( T_i \) UN \( u \); \((-I1.-)\) \( T_{i+1} \) LS \( u \);... \( T_{i+1} \) UN \( u \);...

From the minimization (truncation) performed above remember that the last lock acquired by \( T_i \) is the one (in X mode) on \( u \). All subsequent instructions of \( T_i \) will be unlock instructions. Notice that \( u \) could not have been locked in a non-INV mode by any transaction other than the above three transactions (otherwise the cycle would not be minimal).

Now we need to consider 2 subcases.

**Subcase 1.** \( u = \text{FV}(T_{i+1}) \). This immediately implies that \( T_{i+1} \) is a read-only transaction. \( u \) may or may not be \( \text{FV}(T_i) \). Similarly \( u \) may or may not be \( \text{FV}(T_{i-1}) \) (note that if \( u \) is not equal to \( \text{FV}(T_i) \) then \( u \) must be \( \text{FV}(T_{i-1}) \), as pointed out before). Note that \( u \) may be the root of the DAG.

From Lemma 4.5 we know that we can merge \( T_i \) and \( T_{i+1} \), and get, say, \( T'_i \) which follows the SGP. Now the above history, after the removal of one unlock of \( T_i \) and one lock of \( T_{i+1} \), and appropriate renaming of the other instructions of the two transactions, will look like

... \( T_{i-1} \) LS \( u \);... \( T_{i-1} \) UN \( u \);... \( T'_i \) LX \( u \);... \( T'_i \) UN \( u \);...

This is still a legal history since during the interval \( I1 \) other transactions (if any) could have held \( u \) in only the INV mode and those transactions would still be able to hold those locks simultaneously with \( T'_i \) since the INV and X modes are compatible. By merging the two transactions we have obtained the relation

\[
T_{i-1} <_u T'_i <_u T_{i+1} \]

and hence a cycle of length \( n - 1 \).

\(^3\) In this history \( I1 \) indicates an interval between two actions.
Subcase 2. \( u \neq FV(T_{i+1}) \). We know immediately that \( u \) must be \( FV(T_i) \). Otherwise, there would be a parent of \( u \) which was locked in noncommutative modes by \( T_{i+1} \) and \( T_i \), contradicting the assumption about the choice of \( u \)'s. Note that any predecessors of \( u \) that were locked by \( T_i \) must have been locked only in INV mode. Note further that \( u \) cannot be the root of the DAG (that possibility is covered by Subcase 1).

Let \( v \) be a vertex such that \( \#(v) = \max\{\#(FV(T_{i-1})), \#(FV(T_{i+1}))\} \). Assuming that \( v \neq u \), it can be shown that the history must be of the form

\[
\langle -J6.-\rangle T_{i-1} \text{ LS } v; \ldots T_{i-1} \text{ UN } v; \langle -J3.-\rangle T_i \text{ LINV } v;
\]

\[
\langle -J5.-\rangle T_i \text{ UN } v; \langle -J4.-\rangle T_{i+1} \text{ LS } v; \ldots T_{i+1} \text{ UN } v; \ldots
\]

Briefly, the reasons as to why this should be the case are: We know that \( T_{i-1} \) and \( T_{i+1} \) locked \( v \) in S mode and that \( T_i \) locked it in INV mode. Without loss of generality assume that \( T_{i+1} \) is a read-only transaction and that \( T_{i-1} \) is an update transaction (i.e., \( v = FV(T_{i+1}) \)). By using condition (3) of the guard definition, and knowing that INV and S modes are incompatible, and by inducting on the length of the path common to \( T_i \) and \( T_{i+1} \), from \( v \) to \( u \), it can be easily shown that \( T_i \) must have locked \( v \) before \( T_{i+1} \) locked it. Now we need to show that \( T_{i-1} \) must have locked \( v \) before \( T_i \) locked it. Consider the path \( v = w_0 \rightarrow w_1 \rightarrow \cdots \rightarrow w_{k-1} \rightarrow w_k = u \), common to \( T_{i-1} \) and \( T_i \). Due to the fact that the \( T_j \)'s were defined so that the \( L(T_j) \)'s had the minimal number of elements, it must be that the subguard of \( w_j \) \((j = 1, \ldots, k)\) satisfied by \( T_{i-1} \) has \( w_{j-1} \). From the rules of SGP, we can deduce that since \( T_{i-1} \) had locked \( v \) (i.e., \( w_0 \)) in S mode it could have locked \( w_1 \) only in S mode. By repeating this argument we can show that all vertices on the common path must have been locked by \( T_{i-1} \) in S mode. Once this is known it can be easily shown that \( T_{i-1} \) must have locked \( v \) before \( T_i \) locked it.

Notice that during the time interval from the point of unlocking of \( v \) by \( T_{i-1} \) to the point of unlocking of \( v \) by \( T_{i+1} \) no transaction could have locked \( v \) in X mode. During intervals \( I3 \) and \( I4 \) some transactions could have locked \( u \) in S mode.

Within subcase 2 we need to consider two cases.

Subcase 2a. \( v = FV(T_{i+1}) \). This means that \( u \neq v \). We modify the history by truncating \( T_i \) so that it acquires locks only on the predecessors of \( v \) in INV mode, and a lock on \( v \) in X mode (instead of INV mode) and subsequently no other locks (i.e., after the acquisition of the X mode lock on \( v \), \( T_i \) will consist of only unlock instructions). By doing this the history becomes

\[ Such a path exists due to Lemma 4.1. \]
... $T_{i-1} \text{ LS } v;... T_{i-1} \text{ UN } v;... T_i \text{ LX } v;$

$\langle - i5, r \rangle T_i \text{ UN } v;... T_{i+1} \text{ LS } v;... T_{i+1} \text{ UN } v;$

we still maintain $T_{i-1} < T_i < T_{i+1}$.

This history is legal since we know that no transaction could have acquired an X or S mode lock on $v$ during the interval $i5$. If any transaction $T_j$ other than $T_{i-1}$ or $T_{i+1}$ had locked $v$ in S or X mode then we will have a shorter cycle with $T_i < T_j$ or $T_j < T_i$. This would have contradicted the induction hypothesis. If there is no such transaction then we have an instance of Subcase 1 and we can take the steps outlined there to show the impossibility of a minimal cycle of length $n$.

Subcase 2b. $v \neq FV(T_{i+1})$. This means that $v$ must be $FV(T_{i-1})$ and hence $T_{i-1}$ must be a read-only transaction. If $u \neq v$, just as in Subcase 2a modify $T_i$ so that it acquires an X mode lock on $v$. If after that the cycle becomes shorter then the proof is finished. Otherwise (i.e., if the cycle does not become shorter or to start with $u = v$) we need to do something more (unlike in the case of Subcase 2a we do not now have an instance of Subcase 1). Notice that this means that the vertex $u_{i-2}$ must be a successor of $v$ and that $v$ had not been locked in X mode by any transaction and that $v$ had been locked in S mode by only $T_{i-1}$ and $T_{i+1}$.

We merge transactions $T_{i-1}$ and $T_i$ to get the transaction $T'_i$. $T'_i$ is made to acquire an X mode lock on $v$. We modify the history by shifting all the lock and unlock instructions of $T_i$ except the unlock $v$ instruction to the very beginning of the history, removing $T_{i-1} \text{ LS } v$ and replacing $T_{i-1} \text{ UN } v$ with $T'_i \text{ LX } v$.

These changes leave the new history in a legal state since during interval $i6$ some other transactions (if any) could have locked $v$ only in INV mode and by letting $T'_i$ hold an X mode lock on $v$ during that period we are not disallowing the former.

By merging the two transactions we have obtained the relation

$T_{i-2} < u_{i-2} T'_i < v T_{i+1}$

and a cycle of length $n - 1$, thus contradicting the induction hypothesis. Thus we have proved the serializability of the SGP.

5. Deadlocks in SGP

One of the consequences of the use of a locking protocol to assure serializability is the possibility of creating deadlocks. In this section we will show that on a DAG which is a rooted tree, SGP assures deadlock-freedom, while on an unrestricted DAG, deadlocks are possible. To illustrate that a
deadlock may occur in an unrestricted DAG, consider the database graph depicted in Fig. 5.1, with the following set of guards:

\[
\begin{align*}
guard(r) &= \{\{r\}, \{r\}\} \\
guard(b) &= \{\{b\}, \{b\}\} \\
guard(c) &= \{\{b\}, \{b\}\} \\
guard(d) &= \{\{b\}, \{b\}\} \\
guard(f) &= \{\{r\}, \{r\}\} \\
guard(e) &= \{\{c\}, \{c\}\} \\
guard(g) &= \{\{c\}, \{c\}\}
\end{align*}
\]

The following sequence of execution results in a deadlock:

\[
\begin{align*}
T_2 &\text{ LINV } r; T_2 \text{ LINV } b; T_2 \text{ LINV } c; T_2 \text{ LINV } e; \\
T_3 &\text{ UN } b; T_2 \text{ UN } c; T_1 \text{ LS } b; T_1 \text{ LS } d; T_1 \text{ LS } c; \\
T_1 &\text{ UN } b; T_3 \text{ LX } r; T_3 \text{ LSF } f; T_3 \text{ LX } b; T_3 \text{ LX } d; \\
T_1 &\text{ LS } e; T_2 \text{ LINVF } f;
\end{align*}
\]

before we present the proof of the former result, we present a general result concerning deadlocks in any protocol (not restricted to SGP) supporting the INV, X, and S modes of locking.
DEFINITION 5.1. When we view a protocol as a set of allowed trans-
actions, a protocol $P$ is said to be closed under truncation if for every trans-
action $T$ of $P$, the instructions referring to the last item locked by $T$ can be
deleted from $T$ to form a new transaction $T'$ of $P$.

A locking protocol is called a $T$-protocol if it is closed under truncation.
Most locking protocols that have been proposed in the literature are
$T$-protocols.

One of the results of (Yannakakis, 1982) is that if a set of transactions
following a $T$-protocol $P$ which assures serializability are involved in a
deadlock of the form

$$T_0 \rightarrow u_0 T_1 \rightarrow u_1 \cdots \rightarrow u_{n-2} T_{n-1} \rightarrow u_{n-1} T_0$$

then for any pair of transactions $T_i$ and $T_j$ involved in the deadlock, $T_i \prec T_j$
and $T_j \not{\prec} T_i$.

While (Yannakakis, 1982) considered only the X mode, the above result is
ture even when we consider $T$-protocols which allow the X and S modes (as
shown in Fussell et al., 1981b). The result is not true in general if we
consider any $T$-protocol which supports the INV, X, and S modes. As a
counterexample, consider SGP and the deadlock situation presented in
Fig. 5.1, where we have the cycle

$$T_1 \rightarrow_e T_2 \rightarrow_f T_3 \rightarrow_p T_1.$$ 

In this case the relation $T_1 \prec_n T_3$ is true at the time of the deadlock.

However, if we impose a few additional restrictions, then the above result
will again hold even when INV mode locks are allowed.

THEOREM 5.1. If a set of transactions following a $T$-protocol $P$ which
assures serializability and which supports the X, S, and INV modes of
locking get involved in a deadlock like

$$T_0 \rightarrow u_0 T_1 \rightarrow u_1 \cdots \rightarrow u_{i-1} T_i \rightarrow u_i T_{i+1} \rightarrow u_{i+1} \cdots \rightarrow u_{n-2} T_{n-1} \rightarrow u_{n-1} T_0$$

and if $\forall i$ ($u_i$ is neither held by $T_{i+1}$ in INV mode nor waited for by $T_i$ in
INV mode) then $\forall j \forall k$ ($T_j \not{\prec} T_k$ and $T_k \not{\prec} T_j$ for $j \neq k$) at the time of
deadlock.

Proof. The proof is very similar to the ones given in (Yannakakis, 1982;
Fussell et al., 1981b) for the results quoted from those papers. We briefly
sketch the proof.

The proof is by contradiction. Let us assume that when the hypothesis of
the theorem is true that there exist transactions $T_l$ and $T_m$ in the deadlock cycle

$$T_0 \rightarrow u_0 T_1 \rightarrow u_1 \cdots \rightarrow u_{l-1} T_l \rightarrow u_l \cdots \rightarrow u_{m-1} T_m \rightarrow u_m \cdots \rightarrow u_{n-2} T_{n-1} \rightarrow u_{n-1} T_0$$

such that $T_l < T_m$ at the time of the deadlock.

Now we modify every transaction $T_i$ so that it releases all locks soon after requesting (i.e., before requesting any other locks) the lock on $u_i$ and then terminates without requesting any more locks. In the case of $T_m$ we modify it further by making it not request the lock on $u_m$ (note that since it is a $T$-protocol all the modified transactions are in the protocol). Consequently, the deadlock will not arise and all the transactions will get their last lock (none of which was held or requested in INV mode) giving us the precedence relation

$$T_m <_u u_{m-1} T_m-1 <_u u_{m-2} T_m-2 \cdots <_u u_{l+1} T_l+1 <_u u_l T_l.$$

Combining this with the $T_l < T_m$ relation assumed above we get the nonserialized relation

$$T_l < T_m < T_{m-1} < T_{m-2} \cdots < T_{l+1} < T_l,$$

contradicting the hypothesis which said that the protocol assured serializability. $\blacksquare$

**Theorem 5.2.** The super guard protocol assures deadlock-freedom for rooted trees.

**Proof:** The technique that we use for proving deadlock-freedom is the following: We assume the existence of a minimal deadlock cycle involving $n$ transactions. We allow for the possibility of some $u_i$s being held in INV mode or being waited for in INV mode. Let $m$ be the number of $u_i$s at which an INV mode lock is involved. We induct on $m$. We consider every such $u_i$ and modify the transactions involved and get a new deadlock cycle in which every $u_i$ is neither waited for in INV mode nor held in INV mode.

**Basis Step.** $m = 0$. From Theorem 5.1 we know that no pair of transactions in such a cycle can be related by the $<$ relation. This means that each $u_i$ must be $\text{FV}(T_{l+1})$ and that $u_{l+1}$ must be a successor of $u_l$. Clearly, we cannot have a cycle in this predecessor–successor relationship since we are dealing with a tree. Hence the impossibility of this cycle.

**Induction Step.** From the minimality of the cycle we know that every vertex $u_i$ is being held by only one transaction (namely $T_{l+1}$) at the time of
the deadlock and that all the \( u_i \)'s are distinct. Let us consider the vertex \( u_i \) such that

\[
\rightarrow_{u_{i-1}} T_i \rightarrow_{u_i} T_{i+1} \rightarrow_{u_{i+1}}.
\]

We need to consider two cases.

**Case 1.** \( u_i \) is held in INV mode by \( T_{i+1} \). This means that \( u_i \) is being waited for by \( T_i \) in S mode. It also means that \( T_{i+1} \) may be waiting for an INV, X, or S mode lock on \( u_{i+1} \). Thus, when we consider the pair of vertices \( u_i \) and \( u_{i+1} \) it must be the case that \( u_{i+1} \) is a successor of \( u_i \). Now we take the following steps:

(i) Find the last transaction (call it \( T_j \)), if any, in the history which had locked the vertex \( u_i \) in a non-INV mode. If \( T_j \) had locked \( u_i \) in X mode remove the \( T_j \) UN \( u_i \) instruction so that \( T_j \) continues to hold \( u_i \). By doing this we get a history in which

\[
T_0 \rightarrow_{u_0} T_1 \rightarrow_{u_1} \cdots T_i \rightarrow_{u_i} T_j \rightarrow_{u_j} \cdots \rightarrow_{u_{n-2}} T_{n-1} \rightarrow_{u_{n-1}} T_0.
\]

At least transaction \( T_{i+1} \) would have been eliminated from the cycle, thus giving us a cycle in which \( m \) is less than \( k \), thus contradicting the induction hypothesis.

(ii) If no \( T_j \) as specified in the previous step exists, then modify \( T_{i+1} \) so that it acquires an X mode lock on \( u_i \), instead of an INV mode lock. Then move to that point in the history, where all transactions other than \( T_{i+1} \) are waiting for their requests to be granted, all instructions of \( T_{i+1} \) which request locks for the vertices on the path from \( u_i \) to \( u_{i+1} \). Replace instructions requesting INV mode locks with those requesting X mode locks. It may be the case that \( T_{i+1} \) gets “stuck” even before it could request a lock on \( u_{i+1} \). This will happen if a vertex, say \( v \), which is predecessor of \( u_{i+1} \) is currently being held in X or S mode by a transaction, say \( T_j \), in the deadlock cycle and \( T_{j+1} \) requests an X mode lock on \( v \). If this happens it is fine, since we then will have the relation

\[
\rightarrow_{u_i} T_{i+1} \rightarrow_v T_j \rightarrow_{u_j}.
\]

This may result in the cycle length becoming smaller than \( n \) (we say “may” because, it is possible that \( T_j \) and \( T_{i+2} \) are the same transaction). In the worst case, \( T_{i+1} \) will get “stuck” requesting a lock on \( u_{i+1} \). In any case, we will have reduced the number of \( u_i \)'s on which an INV mode lock is needed or being held by at least 1. There will be a reduction of 2 if \( T_{i+1} \) had requested an INV mode lock on \( u_{i+1} \).

**Case 2.** \( u_i \) is being held by \( T_{i+1} \) in S mode. This means that \( T_i \) must be needing \( u_i \) in INV mode. It also means that \( u_{i-1} \) is being held by \( T_i \) in INV
mode. This is because a transaction requesting an INV mode lock could not have already acquired a lock in any mode other than in INV. Now if we consider the pair of vertices \( u_{i-1} \) and \( u_i \) then we have an instance of Case 1. Hence we can take the steps outlined under Case 1 to handle this case also.

Having shown that SGP assures deadlock-freedom for rooted trees, we shift our attention now to the deadlock problem of unrestricted DAGs. The simplest way to avoid the possibility of deadlocks is to force the transactions to lock the vertices in the ascending order of their enumeration (see Definition 4.1). Forcing the transactions to follow this rule could potentially lead to reduced concurrency since some vertices may remain locked for a time span longer than the time span during which they otherwise would remain locked.

In general, when deadlocks are allowed to occur they can be resolved using a number of recovery schemes (Fussell et al., 1981a; Rosenkrantz et al., 1978; Yannakakis et al., 1979), which usually require one or more transactions to be rolled back (i.e., the effects of the transactions to be “undone”). Once a transaction has been chosen for being rolled back it could be rolled back completely (all the actions of the transaction are “undone”) or partially. The notion of partial rollback was introduced in (Fussell et al., 1981a), in the context of the 2PL protocol. Rolling back one transaction may force some other transactions which were dependent on the former also to be rolled back. This phenomenon is called cascading rollback. Cascading rollbacks due to deadlocks are unique to non-2PL protocols (Kedem et al., 1982).

Once we know that deadlocks will occur and that they must be dealt with, it is interesting to see if cascading rollbacks could be avoided. The advantages of avoiding cascading rollbacks are:

1. Only one transaction needs to be rolled back to resolve any deadlock. This leads to a decrease in the amount of partial transaction executions that are repeated.

2. More importantly, transaction executions need not be monitored to keep track of transaction dependencies (information like which transaction read which transactions’ output). In systems where cascading rollbacks are inevitable, such information is necessary to determine when a decision is made to rollback a particular transaction, what other transactions must also be rolled back.

\[5\] In a partial rollback the transaction is rolled back only to the point in its execution where it is about to acquire a lock on that item which is held by this transaction at the time of the deadlock and which is being needed by the neighboring transaction in the deadlock cycle; by doing this, the neighboring transaction can be granted that lock and the deadlock cycle broken.
(3) If cascading rollbacks are not avoided then an increase in the delay between the time at which a deadlock occurs and the time at which it is detected could potentially lead to an increase in the number of transactions that need to be rolled back.

It turns out that with a very simple condition (which does not restrict the transactions' ability to lock anything that the unrestricted version of the protocol allows) such rollbacks can be avoided. Even the rollback that needs to be done for a single transaction turns out to be a very simple one. It does not require the restoration of the value of any item. Before we discuss that simple condition we need to state a property that holds for any possible deadlock cycle under SGP.

**THEOREM 5.3.** If the concurrent execution of a set of transactions following SGP results in a deadlock cycle of the form,

\[ T_0 \rightarrow_{u_0} T_1 \rightarrow_{u_1} \cdots \rightarrow_{u_{i-1}} T_i \rightarrow_{u_i} \cdots \rightarrow_{u_{n-1}} T_{n-1} \rightarrow_{u_{n-1}} T_0 \]

then there exists a \( u_j \) that is either held in INV mode by \( T_{j+1} \) or needed in INV mode by \( T_j \).

**Proof.** Let us assume that there is no such \( u_j \). Then from Theorem 5.1 we know that at the time of deadlock

\[ \forall j \forall k \, [T_j \bowtie T_k \text{ and } T_k \bowtie T_j]. \]

This would mean that \( u_i \in \{FV(T_{i+1}), FV(T_i)\} \) and that \( FV(T_i) \) is a predecessor of \( u_i \) and \( u_{i-1} \) or that \( FV(T_i) \) is equal to \( u_i \) or \( u_{i-1} \), if one of those two vertices (\( u_i \) and \( u_{i-1} \)) is a predecessor of the other vertex (note that since we have a deadlock cycle all \( u_i \)'s will be distinct). With this information in hand, we can follow the steps given in the proof of Theorem 4.1 (Basis step and Case 1 of the Induction step) to show that such a cycle is not possible. Hence a contradiction and the truth of the theorem.

Next we state the simple condition which guarantees freedom from cascading rollbacks.

**DEFINITION 5.2.** We say that a transaction satisfies The INV lock release (ILR) condition if it releases all INV mode locks immediately after the acquisition of the first X mode lock (before any reading/modification of the item locked in X mode takes place and before any other locks are acquired).

**THEOREM 5.4.** If all update transactions obey the ILR condition, in addition to the rules of SGP, then when a deadlock occurs there will exist at
least one transaction that could be rolled back without causing any cascading
effect and without having to restore the value of any data base item.

Proof. Let there be a deadlock cycle. Then from Theorem 5.3 we know
that \( \exists u_j \) (\( u_j \) is either needed in INV mode by \( T_j \) or held in INV mode by
\( T_{j+1} \)).

Case 1. \( u_j \) is needed in INV mode by \( T_j \). From the rules of SGP we
know that \( T_j \) could have acquired only INV mode locks so far. So \( u_j \) can be
rolled back by merely unlocking the INV mode locks it holds.

Case 2. \( u_j \) is held in INV mode by \( T_{j+1} \). If \( T_{j+1} \) is requesting an INV
mode lock on \( u_{j+1} \) then we have an instance of Case 1. Otherwise, note that
due to the ILR condition \( T_{j+1} \) could only be requesting an X mode lock on
\( u_{j+1} \) and that, too, it must be its first X mode lock request. Since so far \( T_{j+1} \)
would have acquired only INV mode locks, it can be rolled back by
releasing those locks. We call transactions like \( T_{j+1} \), INV transactions.

The interesting thing to notice is that cascading rollbacks are avoided even
if we choose to rollback an INV transaction completely (as opposed to
partially). This is in contrast to what happens in the case of the pitfall
protocol, where only a partial rollback of one of the transactions in the
deadlock cycle will avoid a cascading rollback (see Kedem et al., 1982).

Injudicious use of rollbacks can, while removing deadlocks, result in a set
of transactions becoming involved in a situation in which each transaction in
the set in turn causes another transaction in the set to be rolled back. Such a
state of affairs has the potential to continue to occur indefinitely in the
absence of outside interference, resulting in a sort of dynamic analogue of
deadlock which we call potentially infinite mutual preemption.

In particular, if every deadlock is resolved only by rolling back the INV
transactions, which is what prevents cascading rollbacks, then it is possible
for such transactions to suffer from infinite preemption; but once such a
transaction manages to get its first X mode lock then it is guaranteed to
finish (assuming that the scheduling algorithm is fair in granting pending
lock requests), although it might get involved in deadlocks after that point in
time. Due to the way deadlocks are resolved here the transaction will not get
rolled back as a consequence of the latter deadlocks.

6. Conclusion

In this paper, we have presented an example of how concurrency can be
increased in data base locking protocols by separating the effects of the
commutativity and the compatibility of multiple concurrent accesses of data
items. In particular, we have introduced a new locking mode INV which is
used solely for concurrency control and which does not grant any access privileges (read/write) to the holder of the lock on the associated data item. Thus the "operations" performed by a transaction on an item which is locked in INV mode are commutative with respect to any other transactions' operations on that item, even though the S and INV modes are incompatible. As a result, acquisition of an INV mode lock by a transaction on an item does not imply that the transaction be ordered in an equivalent serial schedule with respect to other concurrently active transactions which have locked (in whatever mode at whatever time) the same item.

We have presented a very general protocol that supports the INV mode of locking. We have examined the effects of this locking mode on the occurrence of deadlocks and have presented an effective and efficient scheme for handling them. We have shown that in the case of rooted trees deadlock-freedom is still assured.

These results indicate that it is not always necessary to equate the commutativity and compatibility of locking modes as has always been done in previous models of concurrency control. The use of this principle in conjunction with other means of increasing concurrency such as allowing conversion of locking modes (Mohan et al., in press) is being examined. We have obtained some results concerning the extension of the directed hypergraph model of locking protocols (Fussell et al., 1981b; Yannakakis et al., 1979; Yannakakis, 1982) to include the INV mode and lock conversions (Mohan, 1981). It is important to recognize that the utility of the INV mode is not restricted to only the super guard protocol. In (Mohan et al., 1982), we have increased the level of concurrency supported by a nonguard protocol, namely the biased protocol, by introducing the INV mode in it.

Received: April 5, 1983; Accepted: April 23, 1984

References


