Consistency in Hierarchical Database Systems

ABRAHAM SILBERSCHATZ AND ZVI KEDEM

The University of Texas at Dallas, Richardson, Texas

ABSTRACT The problems of locking and consistency in database systems are examined. It is assumed that each transaction, when executed alone, transforms a consistent state into a consistent state. A set of conditions is derived to guarantee that when transactions are processed concurrently, the results are the same as would be obtained by processing the transactions serially. These conditions are used to establish a locking protocol in hierarchical database systems. The locking protocol allows transactions to request new locks after releasing a lock. However, a data item may be locked at most once as a result of each transaction. It is shown that the protocol ensures consistency and that it is deadlock free.

KEY WORDS AND PHRASES: database, consistency, lock, concurrency, transaction, integrity, deadlock, serializability

CR CATEGORIES 4.33, 4.34

1. Introduction

In recent years there has been a growing awareness of the need to ensure the integrity of database systems that are subjected to multiple concurrent update activity [4, 7, 13]. A common approach to the integrity problem is to group the actions of a process into sequences called transactions which are units of consistency [4]. It is assumed that each transaction when executed alone transforms a consistent state into a new consistent state; that is, transactions preserve consistency. Given that each transaction when executed alone cannot violate database consistency, consistency is assured if the total set of transactions executed in the system is serializable [4, 7]. Serializability dictates that the outcome of processing all those transactions in the set concurrently must be the same as that produced by running the transactions serially in some order.

When no control is induced on a system which allows concurrent execution of transactions, consistency is generally not assured. For example, consider the two transactions $T_1$ and $T_2$ described below (this example is due to Esowaran et al. [4]):

$T_1$: $A \leftarrow A + 1,$

$B \leftarrow B + 1,$

$T_2$: $B \leftarrow 2 \cdot B,$

$A \leftarrow 2 \cdot A.$

Suppose that the consistency constraint on the system state is that $A = B$. Consider the following sequence of execution:

$T_1$: $A \leftarrow A + 1,$

$T_2$: $B \leftarrow 2 \cdot B,$

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This work was supported in part by the National Science Foundation under Grants MCS 7702463 and MCS 7703905.

Authors' address Mathematical Sciences Program, The University of Texas at Dallas, Richardson, TX 75080

© 1980 ACM 0004-5411/80/0100-0072 $00.75

Consistency in Hierarchical Database Systems

\[ T_1: \quad B \leftarrow B + 1, \]
\[ T_2: \quad A \leftarrow 2 \times A. \]

In this case \( A \neq B \) after the execution of both \( T_1 \) and \( T_2 \), and thus the state is inconsistent.

In order to ensure consistency, transactions must interact in such a way that the effect of their execution must be equivalent to the effect of some serial noninterleaved sequence of those same transactions. In other words, consistency is ensured if

1. each transaction sees a consistent database,
2. each transaction eventually terminates,
3. the final database after all transactions terminate is consistent.

There are two main techniques to handle the question of consistency. First, partition the database into small disjoint units which are individually locked. A locking algorithm is employed to ensure consistency \([4, 6, 7, 11]\). Second, provide a concurrency control and several protocols to be executed by that control to resolve conflicts that can occur during execution \([12, 13]\). In Section 2 a brief survey of these two techniques is given. We point out the difficulties we have encountered, and propose alternatives which we feel may provide better solutions. In particular, we examine the problem of consistency in database systems that are hierarchically organized \([14]\). The reason we have chosen hierarchical systems is twofold. First, there are a number of successful database management systems which are hierarchically organized, which indicates that hierarchical structures might be a right tool for organizing database systems. Second, the hierarchy imposed on the database allows us to obtain better results concerning consistency.

2. Previous Work

A considerable amount of work has been performed in areas related to the problem discussed in this paper. Eswaran et al. \([4]\) proposed a locking protocol which allows a transaction to lock as many data items as it wants. However, as soon as it unlocks one of these items no more locking is allowed. This scheme ensures consistency but does not ensure freedom from deadlock. For example, consider the transactions \( T_1 \) and \( T_2 \) previously described. Suppose that we associate a lock with the variable \( A \) and a lock with the variable \( B \).

\[ T_1: \quad \text{LOCK} A, \]
\[ T_1: \quad A \leftarrow A + 1, \]
\[ T_2: \quad \text{LOCK} B, \]
\[ T_2: \quad B \leftarrow 2 \times B, \]
\[ T_1: \quad \text{LOCK} B, \]
\[ T_2: \quad \text{LOCK} A. \]

\( T_1 \) now is waiting for \( T_2 \) to UNLOCK \( B \) and \( T_2 \) is waiting for \( T_1 \) to UNLOCK \( A \). Thus a deadlock occurs, although Eswaran's locking protocol is observed.

Various issues in deadlock detection were discussed in Coffman et al. \([2]\) and in Chamberlin et al. \([1]\). Deadlock detection in database concurrency controls were discussed in King and Collmeyer \([8]\) and in Gray et al. \([6]\). Deadlock detection and recovery in general is an expensive task and should be avoided whenever it is possible. We therefore seek a locking protocol that will ensure freedom from deadlocks.

The question of consistency in database systems was approached by Stearns et al. \([13]\) and Rosenkrantz et al. \([12]\) from a different point of view. Their starting point was the requirement that transactions do not lock and unlock the database entities they access. Instead they provided a concurrency control which decided upon actions to be taken in response to requests by each transaction to read and write into the database. In their approach several versions of each entity were kept in the system. A transaction that conflicts with another transaction either waits or is required to rollback and restart its execution. Since restarting a transaction and keeping several versions of each entity is an expensive proposition, one should look at alternative approaches.
This paper examines the problems of locking and consistency in more detail. In Section 3 we define a “less than” relation between pairs of transactions that access some common variables. Given this relation, which is irreflexive, antisymmetric, and transitive, we prove that if this relation holds between every pair of interacting transactions, then the system is serializable. This “less than” relation provides a general method for deciding what kind of locking protocols ensure consistency. (For example, Eswaran’s result stating that his locking protocol ensures consistency follows from our Theorem 1.) In Section 4 we apply our result to hierarchical database systems. We take advantage of the structure of the database and define an efficient locking protocol which allows a transaction to lock a data item after releasing another data item. However, a data item may not be locked twice by the same transaction. We demonstrate that our locking protocol ensures serializability and freedom from deadlocks.

3. General Result

We proceed to investigate what conditions will guarantee that a system is serializable. In order to do so it is necessary to define several terms and state several assumptions.

The set of all transactions executed between two consecutive quiescent states will be denoted by $T$. A system is said to be in a quiescent state if no transactions are being processed at that time. Each transaction $T_i$ in $T$ accesses (retrieves or/and updates) some of the database variables as a result of its execution, the set of all these variables is denoted by $A(T_i)$. We define a relation $\alpha \subseteq T \times T$, where $T_i \alpha T_j$ (reads $T_i$ precedes $T_j$) iff

(a) $A(T_i) \cap A(T_j) \neq \emptyset$, and
(b) for every $V \in A(T_i) \cap A(T_j)$, $T_i$ accessed $V$ for the first time only after $T_j$ had accessed $V$ for the last time.

**Lemma 1.** $\alpha$ is irreflexive and antisymmetric (but not necessarily transitive).

**Theorem 1.** Let the set $T = \{T_i | i = 0, 1, \ldots, n - 1\}$ of the transactions executed between two quiescent states satisfy the following conditions:

(a) if $i \neq j$ and $A(T_i) \cap A(T_j) \neq \emptyset$, then $T_i \alpha T_j$ or $T_j \alpha T_i$, and
(b) the relation $\alpha$ has no cycles.

Then $T$ is serializable.

**Proof.** Let $\alpha^*$ be the reflexive-transitive closure of $\alpha$. By assumption (b) $\alpha^*$ is antisymmetric and therefore a partial order. Thus, $T$ admits a consistent enumeration under $\alpha^*$ [10]. Without loss of generality let it be $T_0, T_1, \ldots, T_{n-1}$. Thus, because of assumption (a), and by the definition of $\alpha$, the outcome of the concurrent execution is the same as the serial execution: $T_0, T_1, \ldots, T_{n-1}$. 

The relation $\alpha^*$, which is the transitive closure of $\alpha$, is in fact the “less than” relation referred to in Section 2.

The assumptions of Theorem 1 provide a framework for deciding upon locking rules in an environment where concurrent execution is permitted. These assumptions do not specify how locking should be done, but rather provide an insight into this question. For example, Eswaran et al. [4] described a locking protocol which requires that a transaction cannot request new locks after releasing a lock. It can be shown that their protocol satisfies the assumptions of Theorem 1. In Section 4 a locking protocol for hierarchical database systems is described. The proof that the protocol ensures serializability follows from our theorem.

4. Locking Protocol for Hierarchical Database Systems

In Section 3 we have defined a set of conditions to ensure serializability. We proceed now to use these conditions to define a locking protocol for hierarchical database systems [14]. We prove that the protocol ensures that a system is serializable and that it is deadlock free.
We assume that the database is partitioned into small disjoint units which we refer to as records. The relationship between records is represented by a tree [9]. As usual we write \( X \geq Y \) for two nodes \( X \) and \( Y \) in the tree if and only if \( X \) is a weak ancestor of \( Y \), namely, \( Y \) or its ancestor. An example of a small database tree is presented in Figure 1, where each node corresponds to a record.

A locking algorithm for a hierarchical organized system was discussed by Gray [5]. A lock is associated with each record in the tree (restricted hierarchy). A transaction may request to lock a particular record in the tree. Once a request is granted, the entire subtree rooted at the requested locked record is considered to be locked by the transaction requesting the lock. The problem is to find a technique for implicitly locking an entire subtree. In order to lock a subtree rooted at record \( R \) it is necessary to prevent a transaction from successfully locking the ancestors of \( R \) while \( R \) is still being locked. Gray has proposed a new access mode, called intention mode, to "tag" (lock) all ancestors of a node to be locked. These tags signal the fact that locking is being done at a lower level and thereby prevent implicit or explicit locks on the ancestors. Thus in order to lock a subtree rooted at node \( R \) one must first lock all the ancestors of \( R \) in intention mode and only then lock node \( R \).

In Gray's model locks are requested in root to leaf order. Once a node is locked, no further explicit locking is required at lower levels. Locks are released either at the end of the transaction in any order (or in leaf to root order). This scheme has the advantage that for certain hierarchies the number of locks that need to be set may be far fewer than the number of nodes accessed by the locking transaction. This is due to the fact that locking is done on a subtree basis rather than on a record basis. One drawback of this scheme is that in some cases it may result in the loss of potential concurrency. This is because of the requirement that all the ancestors of an accessed node must be locked by the accessing transaction. This requirement may also result in unnecessary overhead for intention mode locking if a transaction need not start accessing the tree at its root. In the following we propose an alternative to Gray's locking algorithm in which locking is done on a record basis. This will eliminate the drawbacks we have pointed out in Gray's scheme. However, for certain hierarchies, a greater number of locks (to be set) may be required.

Under our scheme, in order to access a record, a transaction must first lock it. If the record is already locked (by another transaction), the transaction will be suspended. The suspended transaction may resume its execution when another transaction unlocks this record. Only one transaction resumes its execution as a result of a single unlock instruction.
We denote the fact that a transaction $T$, has issued a LOCK instruction for a record $X$ and has successfully locked $X$ by writing $T$, locked $X$. Each transaction $T$, must select a first record in the database tree to be locked; this first record is denoted by $E(T)$. The records in the subtree with root node $E(T)$ are the only records that may be locked by transaction $T$. Moreover, a transaction may lock a particular record only if its father is currently being locked by the transaction. For example, suppose that $E(T)$ is the node $B$ in the database tree of Figure 1. $T$, can only access records $B, C, D, E, F, G, H,$ and $I$. In order to lock $D$, $T,$ must first lock the record $C$; similarly to lock $E$ it must first lock $G$.

A transaction may unlock a particular record after having finished accessing that record. We assume that all locked records are unlocked within a finite amount of time. (We will show that our locking protocol is deadlock free.) We also require that a record which has been unlocked by $T,$ is not locked again by the same transaction. These requirements can be shown to guarantee serializability. In addition these rules ensure that a system is deadlock free.

Let us illustrate how these locking rules apply to the database tree of Figure 1. Suppose $E(T) = B$. A particular sequence of LOCK and UNLOCK instructions which follows our locking rules is

```
LOCK B,
LOCK C,
LOCK D,
UNLOCK D,
LOCK G,
UNLOCK B,
UNLOCK C,
LOCK E,
LOCK F,
UNLOCK G,
UNLOCK F,
UNLOCK E
```

The following sequence of LOCKs and UNLOCKs violates our rules since $C$ is locked twice by the same transaction:

```
LOCK B,
LOCK C,
UNLOCK C,
LOCK F,
LOCK C,
UNLOCK F,
UNLOCK C
```

We now formally present our locking protocol and the results following from it. We denote by $E(T)$ the first node (record) locked by transaction $T$. We also say that $T,$ is holding a lock on record $R$ if and only if the node $R$ was locked by $T,$ previously and has not yet been unlocked.

Locking Protocol

(a) $T,$ may lock $R \neq E(T)$ if and only if $T,$ is holding a lock on $R$'s father in the database tree.
(b) After unlocking $R,$ $T,$ may not lock it again.
(c) $T,$ may access only those nodes on which it is holding a lock.

It can be easily seen that our locking protocol does not require transactions locking some set of nodes to lock these nodes in the same order. Moreover, the subgraph of the tree spanned by the nodes of the tree for which a transaction is holding a lock (at some
point in time) is not necessarily connected. Also it is evident that the set of all the nodes locked by a transaction throughout the execution is connected. Furthermore, for any connected subgraph of the tree there exists a transaction obeying our protocol which locks exactly those nodes which span the subgraph. (For example, a transaction may lock the nodes in the order imposed by a depth-first search of the connected subset; then it may unlock all the locked nodes.)

We now examine in greater detail the interactions possible among the transactions executed concurrently. For a transaction $T_i$ we denote by $L(T_i)$ the set of all nodes of the tree for which $T_i$ issued a LOCK instruction.

We define a relation $\omega \subseteq T \times T$ where $T_i \omega T_j$ iff

(a) $L(T_i) \cap L(T_j) \neq \emptyset$, and

(b) there exists a node $V \in L(T_i) \cap L(T_j)$ such that $T_i$ locked $V$ and $T_j$ either never locked it or locked it only after $T_i$ unlocked it. (Note that it is irrelevant when $T_j$ issued a LOCK $V$ instruction.)

The relation $\omega$ describes a very weak constraint on the relative interaction between $T_i$ and $T_j$. We proceed to show that it implies a much stronger "ordering" relation

**Lemma 2.** The relation $\omega$ is antisymmetric

**Proof.** Without loss of generality, assume by contradiction that $T_0 \omega T_1$, and $T_1 \omega T_0$. We first note that either $E(T_0) \geq E(T_1)$ or $E(T_1) \geq E(T_0)$. Indeed, otherwise the subtrees rooted at $E(T_0)$ and $E(T_1)$ are disjoint and $L(T_0) \cap L(T_1) = \emptyset$. Let $V = \min (E(T_0), E(T_1))$. Then it follows that $V$ must be the greatest node in the set $L(T_0) \cap L(T_1)$. Without loss of generality, assume that $V = E(T_0)$. Define now:

$$i \triangleq \begin{cases} 0, & \text{if } T_0 \text{ locked } V \text{ before } T_1 \text{ locked } V; \\ 0, & \text{if } T_0 \text{ locked } V \text{ and } T_1 \text{ never locked } V; \\ 1, & \text{otherwise;} \end{cases}$$

and $j \triangleq 1 - i$.

To reiterate: $T_i$ locked $V$; and if transaction $T_j$ ever locked $V$, it did so after $T_i$ unlocked it.

Let

$$X_1 = E(T_i), X_2, \ldots, X_p = V, \ldots, X_{10}$$

be the sequence of all the nodes for which $T_i$ issued a LOCK instruction. The nodes are listed in the order in which the LOCK instructions were issued. (If $i = 0$, then $p = 1$.) Note that $T_i$ locked at least the nodes $X_1, X_2, \ldots, X_{10}$. As $T_j \omega T_i$, it follows that if

$$M \triangleq \{m|T_j \text{ locked } X_m \text{ before } T_i \text{ locked it, or } T_i \text{ never locked } X_m\}$$

then $M \neq \emptyset$. Let $r = \min(M)$. (Note that $X_r$ is the first element in the sequence which is also in $M$, and not a minimal element in the tree which is also in $M$.) By the definition of the protocol, for some $q, p \leq q < r, X_q$ is the father of $X_r$ in the tree. As $X_q \leq V \leq E(T_i)$ in the tree, it follows that $X_q$ must have been locked by $T_j$ when it locked $X_r$. But, by the definition of $r$, it also follows that $T_i$ locked (and unlocked) $X_q$ before $T_j$ locked it. There are two cases to be considered:

(a) $T_i$ locked $X_r$. In this case it follows from $X_q \leq V \leq E(T_i)$ that $T_i$ could not lock $X_r$ without holding a lock on $X_q$ at the same time. As $T_j$ locked $X_q$ only after $T_i$ unlocked it, it follows that $T_i$ could not lock $X_r$ before $T_i$ unlocked it—a contradiction to the fact that $r \in M$.

(b) $T_i$ issued a LOCK $X_r$ instruction but never locked $X_r$. In this case it follows that $T_i$ never unlocked $X_q$ and thus $T_j$ never locked $X_q$, and thus never locked $X_r$—a contradiction.
We have thus shown that we cannot have both $T_i \omega T_j$ and $T_j \omega T_i$, and Lemma 2 has been proved.

**Corollary 1.** If $T_i \omega T_j$, then every node in $L(T_i) \cap L(T_j)$ which has been locked by both $T_i$ and $T_j$ was locked by $T_i$ first.

**Proof.** Indeed, otherwise $T_i \omega T_j$ and $T_j \omega T_i$, contradicting Lemma 2.

We can now summarize some constraints on the relative interactions between transactions. For any two transactions $T_i, T_j$ the set $L(T_i) \cap L(T_j)$ is a tree; furthermore if $T_i$ locked the root of that tree before $T_j$ did, then its every node that was locked by $T_i$ and $T_j$ was locked by $T_i$ first. An algorithm developed independently by Ellis [3] for concurrent operations on 2-3 trees has a similar feature. Oversimplifying, and using some of her terminology, a writer, in its restructuring phase, moves along a path from a leaf to the root of the tree. For the segment of such a path common to two writers, if the writer $T_i$ locked the lowest node before $T_j$ did, then every node in that segment was locked by $T_i$ before $T_j$ locked it.

**Theorem 2.** $\omega^*$, the reflexive-transitive closure of $\omega$, is a partial order.

**Proof.** To prove Theorem 2, it will suffice to show that there are no cycles of length two or more in the directed graph $(T, \omega)$. Let $T = \{T_0, T_1, \ldots, T_{n-1}\}$. We shall prove by induction on $m$ that there are no cycles of length $m \geq 2$.

$m = 2$. Follows immediately from Lemma 2.

$m = 3$. Assume by contradiction, without loss of generality, that $T_0 \omega T_1$, $T_1 \omega T_2$, $T_2 \omega T_0$. Without loss of generality, let $V = E(T_i)$ be minimal in the set $\{E(T_0), E(T_1), E(T_2)\}$. It follows that $E(T_0) \geq E(T_1)$ and $E(T_2) \geq E(T_1)$. As $T_0 \omega T_1$ and $E(T_0) \geq V$, it follows by Corollary 1 that $T_0$ locked $V$ before $T_1$ locked it. As $T_1 \omega T_2$ and $E(T_2) \geq V$, it follows that $T_1$ locked $V$ before $T_2$ could lock it. Thus $T_0$ locked $V$ before $T_2$ could. (We have not shown that $T_2$ even locks $V$; we know though that it has issued a LOCK instruction for it.) Thus by the definition of $\omega$, $T_0 \omega T_2$. We now have $T_0 \omega T_2$, $T_2 \omega T_0$, a cycle of length 2, contradicting the induction hypothesis.

$m > 3$. Assume by contradiction, without loss of generality, that $T_0 \omega T_1$, $T_1 \omega T_2$, $\ldots$, $T_{m-1} \omega T_0$. Let $V$ be a minimal node in the set $\{E(T_0), E(T_1), \ldots, E(T_{m-1})\}$. As $T_{i-1} \omega T_i$ and $T_i \omega T_{i+1}$, we again note that $E(T_{i-1}) \geq E(T_i)$ and $E(T_{i+1}) \geq E(T_i)$. (All indices are modulo $m$.) We also know that both $T_{i-1}$ and $T_{i+1}$ issued a LOCK $V$ instruction. There are two cases to be considered.

(a) Neither $T_{i-1}$ nor $T_{i+1}$ ever locks $V$. We first note that as $T_{i-1} \omega T_i$ and $T_{i+1} \omega T_{i+2}$, it follows by the definition of $\omega$ that each of them locked at least one node in the tree. It follows thus that while both of them were waiting to lock $V$ after they had issued a LOCK for it, they were holding a lock on $V$'s father in the tree, which is impossible. Thus a contradiction is reached.

(b) At least one of $T_{i-1}$, $T_{i+1}$ locked $V$ (and possibly unlocked it). Then as both of them issued a LOCK for it, it follows by the definition of $\omega$ that either $T_{i-1} \omega T_{i+1}$ or $T_{i+1} \omega T_{i-1}$. In the first case,
Consistency in Hierarchical Database Systems

\[ T_0 \omega T_1, \ldots, T_{i-1} \omega T_{i+1}, \ldots, T_{m-1} \omega T_0, \]

a cycle of length \( m - 1 < m \) is obtained; and in the second case,

\[ T_{i-1} \omega T_i, \quad T_i \omega T_{i+1}, \quad T_{i+1} \omega T_{i-1}, \]

a cycle of length \( 3 < m \) is obtained. This contradicts the induction hypothesis. \( \square \)

**Lemma 3.** \( \alpha \subseteq \omega. \) (This means that \( T_i \alpha T_j \) implies \( T_i \omega T_j. \))

**Proof.** Let \( i \) and \( j \) be such that \( T_i \alpha T_j. \) This implies that for at least one variable \( v \) located at some node \( V, \) \( T_i \) and \( T_j \) both accessed \( v, \) but the first access by \( T_j \) occurs after the last access by \( T_i. \) But then of course \( T_i \) locked and unlocked \( V \) before \( T_j \) locked it. Thus by the definition of \( \omega, \) \( T_i \omega T_j. \) \( \square \)

**Corollary 2.** \( T \) is serializable.

**Proof.** It is sufficient to show that the two assumptions (a) and (b) of Theorem 1 are satisfied.

(a) Let \( i \) and \( j \) be such that \( i \neq j \) and \( A(T_i) \cap A(T_j) \neq \emptyset. \) Let \( v \in A(T_i) \cap A(T_j), \) and let \( v \) be located in some node \( V. \) Then \( V \in L(T_i) \cap L(T_j), \) and either \( T_i \) locked and unlocked \( V \) before \( T_j \) locked it, or \( T_j \) locked and unlocked \( V \) before \( T_i \) locked it. Without loss of generality, assume that the first case holds. Then by the definition of \( \omega, \) \( T_i \omega T_j; \) and by Corollary 1 it immediately follows that for every variable \( v \in A(T_i) \cap A(T_j), \) the first access of \( v \) by \( T_j \) occurs after the last access by \( T_i. \) Thus by the definition of \( \alpha, T_i \alpha T_j. \)

(b) Assume by contradiction that there exists a cycle in the graph \( \langle T, \alpha \rangle. \) As, by Lemma 3, \( \langle T, \alpha \rangle \) is a partial subgraph of \( \langle T, \omega \rangle, \) a cycle exists in the graph \( \langle T, \omega \rangle, \) contradicting Theorem 2. Thus, there are no cycles under the relation \( \alpha. \) \( \square \)

**Corollary 3.** A system operating under our locking protocol is deadlock free.

**Proof.** For a point \( t \) in time define a relation \( \delta_t \subseteq T \times T, \) where \( T, \delta_t T_j \) iff at time \( t, \) \( T_i \) is holding a lock on \( X, \) and \( T_j \) has issued \( \text{LOCK} X \) but did not lock it by time \( t. \) (At time \( t, T_j \) is waiting to lock a node on which \( T_i \) is holding a lock.) Assume by contradiction that the system is deadlocked at time \( s. \) Then, without loss of generality, for some \( m > 1 \) there exists a cycle

\[ T_0 \delta_s T_1, T_1 \delta_s T_2, \ldots, T_{m-1} \delta_s T_0. \]

But, by the definitions of \( \delta_s \) and \( \omega, \) it follows that \( \delta_s \subseteq \omega \) and thus

\[ T_0 \omega T_1, T_1 \omega T_2, \ldots, T_{m-1} \omega T_0, \]

contradicting Theorem 2. \( \square \)

5. Conclusion

We have presented results concerning consistency in database systems. It was assumed that each transaction, when executed alone, transforms a consistent state into a consistent state. A set of conditions was derived to guarantee that when transactions are processed concurrently, the results are the same as would be obtained by processing the transactions serially. These conditions were used to establish a locking protocol in hierarchical database systems. It was shown that the protocol ensures consistency and that it is deadlock free.

The proposed locking protocol can be interpreted in two different ways. First, the tree structure imposed on records is a logical view of the record structure. That is, it provides a systematic way for deciding how a transaction should \text{LOCK} and \text{UNLOCK} records, so that consistency is preserved and no deadlocks may occur. Second, the tree structure is a physical structure in which pointers (explicit or implicit) from one record to another exist. In this case our locking protocol provides a guideline as to how one should traverse a tree so that consistency is ensured.
Since the relationship between records is represented by a tree, one may hope that high-level mechanisms can be provided for automatic LOCKing and UNLOCKing of records. That is, a programmer writing the definition of a transaction need not specify how the setting and resetting of locks should be done. Instead, a compiler might generate for each transaction definition the appropriate sequence of LOCKs and UNLOCKs. This question, however, is beyond the scope of this paper and will be examined separately.

ACKNOWLEDGMENT. We acknowledge the suggestions and helpful criticism given by Jeffrey Ullman.

REFERENCES