A Family of Locking Protocols for Database Systems that Are Modeled by Directed Graphs

ABRAHAM SILBERSCHATZ and ZVI M. KEDEM, MEMBER, IEEE

Abstract—This paper is concerned with the problem of ensuring the integrity of database systems that are accessed concurrently by a number of independent asynchronously running transactions. It is assumed that the database system is partitioned into small units that are referred to as the database entities. The relation between the entities is represented by a directed acyclic graph in which the vertices correspond to the database entities and the arcs correspond to certain access rights. We develop a family of non-two-phase locking protocols for such systems that will be shown to ensure serializability and deadlock-freedom. This family is sufficiently general to encompass all the previously developed non-two-phase locking protocols as well as a number of new protocols. One of these new protocols that seems to be particularly useful is also presented in this paper.

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

INTRODUCTION

A DATABASE system in a general sense may be viewed as a pair $DS = \langle D, P \rangle$, where $D$ is the set of the database entities and $P$ is the set of all the programs that may access $D$. A transaction in such a system is the execution of one of the programs in the set $P$. An important issue which arises in the design of a database system is the problem of ensuring the consistency of the database when it is accessed concurrently by several asynchronously running transactions. A common approach to 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 into a consistent state), and require that the outcome of processing a set of transactions concurrently will be the same as the one produced by running these transactions serially in some order. A system that ensures this property is said to be serializable [1], [2].

In order to ensure serializability some form of supervision must be present to influence the manner in which the transactions executing in the database interact with each other. If no such supervision exists, consistency in general is not assured. To illustrate this point, consider the two transactions $T_1$ and $T_2$ defined as follows (this example is due to Rosenkrantz et al. [3]):

$T_1$: if $A = 0$ then $B \leftarrow B + 1$  
$T_2$: if $B = 0$ then $A \leftarrow A + 1$.

Let the consistency requirement be $A = 0 \lor B = 0$, with $A = B = 0$ the initial values. Consider the following sequence of execution:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$: if $A = 0$</td>
<td>true</td>
</tr>
<tr>
<td>$T_2$: if $B = 0$</td>
<td>true</td>
</tr>
<tr>
<td>$T_3$: $A \leftarrow A + 1$</td>
<td>$A = 1$</td>
</tr>
<tr>
<td>$T_4$: $B \leftarrow B + 1$</td>
<td>$B = 1$</td>
</tr>
</tbody>
</table>

In this case we have $\neg (A = 0 \lor B = 0)$ after the execution of both $T_1$ and $T_2$ and thus the state is inconsistent.

Only relatively recently have researchers on concurrency in database systems identified and studied the serializability [1], [2] issues. To see how poorly understood this problem has been, note that some proposals published in the past for design of concurrent database systems using the network model permit interactions of concurrent transactions which produce incorrect results [4].

Since the realization of the importance of serializability, several researchers have investigated the problem and produced initial promising results [5]–[9]. All these results require the use of locking protocols to influence the manner in which the database transactions may interact with each other. (For a tutorial on this subject, see [10].) It is assumed that a transaction must lock an entity before accessing it. Furthermore, the transaction must unlock that entity at some point in time in the future (unless a deadlock occurs). Thus, a locking protocol may be viewed as a restriction on when a transaction may lock and unlock each of the entities in the set $D$.

Eswaran et al. [1] have proposed a two-phase locking protocol to ensure serializability. Such a locking protocol is characterized by the fact that a transaction cannot request a new lock after releasing a lock. Thus, the restrictions on locking are that a transaction has not previously issued an unlock instruction. Eswaran et al. have shown that in the absence of information concerning the manner in which the database entities are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability. One drawback of the two-phase protocol is that is does not ensure freedom from deadlocks [11]. This deficiency can be remedied through the use of various recovery schemes which require one or more transactions to be rolled back [2], [3].

Although the use of a two-phase protocol is necessary in general for ensuring serializability, Silberschatz and Kedem [7]
have shown that if one has a priori knowledge as to how the entities of the database are organized, one may be able to design locking protocols that ensure serializability (and freedom from deadlock) and which are not two-phase. In particular, they proposed a locking protocol for databases that are organized as rooted trees. That protocol (the tree protocol) allows a transaction to select any entity in the database tree as the first to be locked. Subsequently, the transaction may lock an entity only if its father (vertex) is currently locked by that transaction. A transaction may unlock an entity at any point in time. However, once an entity has been unlocked, it cannot be locked again by the same transaction. One nice property of this protocol is that in addition to ensuring serializability and freedom from deadlocks it is very simple and efficient. It does not require the presence of a concurrency control, and no rollbacks are necessary.

These results were extended by Kedem and Silberschatz [8], who developed a non-two-phase locking protocol for database systems that are organized as directed acyclic graphs. Another protocol for such database systems was developed by Yannakakis et al. [9]. They also studied conditions on general sets of transactions which assure serializability. The various interactions between transactions were expressed in terms of hypergraphs. The derived conditions are useful for proving serializability but seem to require extension for proving deadlock-freedom.

In this paper a family of general non-two-phase locking protocols is developed. This family includes all the previously developed protocols on directed acyclic graphs. A new protocol is also presented. All these protocols require a priori knowledge of the structure of the database. Thus, the protocols are restricted since the structure of the database and the accesses to the data, are strongly controlled. Discussions concerning the advantages of the non-two-phase protocols as well as their limitations are also presented.

II. System Model

Following the notation presented in [1], we consider a transaction $T_i$ as a sequence:

$$<T_i, a_1, e_1>; <T_i, a_2, e_2>; \cdots; <T_i, a_n, e_n>$$

where $a_j$ is the instruction executed at step $j$, and $e_j$ is the entity acted upon by that instruction.

A transaction $T_i$ may access a database entity $e$ only if it has successfully locked that entity with an EXCLUSIVE lock (in this paper we do not consider SHARED locks; for results concerning this subject, see [12]). A request to lock $e$ is accomplished via the LOCK EXCLUSIVE instruction $(T_i, LX, e)$, while a request to unlock $e$ is accomplished via the UNLOCK instruction $(T_i, UN, e)$. Each transaction must follow a locking protocol.

We will consider only uninterpreted transactions, namely transactions from which only the $LX$ and $UN$ were extracted. (Each transaction will, of course, satisfy certain obvious syntactic restrictions, such as: an item that is not locked cannot be unlocked, etc. For a more thorough discussion on this issue see [1] or [9] we wish to examine traces of some concurrent executions between two quiescent states. (The discussion could easily be extended to an infinite set of transactions for which the second quiescent state may not exist.) Such a trace will be referred to a schedule (history). In order to distinguish between the event of a transaction requesting a lock and the one of acquiring a lock, we introduce the pseudoinstruction $LX$ which indicates acquisition of a lock. Unless $LX$ is followed immediately in the schedule by $LX$, the transaction is suspended between. To clarify this notation we present a very simple example of a schedule consisting of three transactions accessing a database of two entities $a, b$:

$$<T_0, LX, a>; <T_0, L\bar{X}, a>; <T_0, LX, b>;$$
$$<T_0, L\bar{X}, b>; <T_1, LX, a>; <T_2, LX, a>;$$
$$<T_0, UN, a>; <T_2, L\bar{X}, a>; <T_2, LX, b>;$$
$$<T_0, UN, b>; <T_2, L\bar{X}, b>; <T_2, UN, a>;$$
$$<T_1, L\bar{X}, a>; <T_1, UN, a>; <T_2, UN, b>.$$ Generaly, a schedule may not be “complete,” as the set of transactions participating in it may reach quiescent state not only by finishing the execution of the transactions, but also by entering a deadlock.

We will consider two issues arising in the context of concurrent execution, serializability and deadlock-freedom. (Intuitively, they parallel loosely partial correctness and termination for serial programs.) As has been shown in the Introduction, not all schedules correspond to “correct” execution, even if each transaction is correct. We are sure though that serial schedules, namely schedules in which each transaction executes alone in the database, are correct. We want to impose restrictions on the various transactions so that each possible schedule will be equivalent to some serial schedule, namely be serializable. Our second concern will be prevention of deadlocks.

Let the database consist of the items in the set $V = \{v_0, v_1, \cdots, v_{n-1}\}$. Consider a schedule of a finite set of transactions $T = \{T_0, T_1, \cdots, T_{m-1}\}$:

$$a_1; a_2; \cdots; a_p; (T_i, LX, e); b_1; b_2; \cdots; b_q;$$

We shall say that $T_i < T_j$ if one of the following holds:

1) one of the $b_j$'s is of the form $(T_j, LX, e)$;
2) one of the $a_k$'s or $b_k$'s is of the form $(T_j, LX, e)$ but no subsequent instruction is of the form $(T_j, LX, e)$.

Define now

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

We say that a locking protocol is reliable if and only if every possible schedule of a set of transactions following the locking protocol is serializable and deadlock-free.

Proposition 1: A locking protocol is reliable if it ensures that the relation $<$(for any possible schedule) is acyclic.

Proof: Note that a relation is acyclic if and only if it can be embedded in a linear order.

III. Basic Result

It is natural to consider a database organized as a directed graph $G = \langle V, A \rangle$ whose vertices correspond to the lockable entities and whose arcs correspond to some locking rights. A
classical example would be the IMS system [6]; other examples can be found in [7]–[9]. We will thus consider the database to be modeled by a directed acyclic graph whose vertices are the items in \( V \).

We shall say that a directed acyclic graph \( G \) is a guarded graph if and only if with each \( v \in V \) there is associated a (possibly empty) set of pairs:

\[
guard(v) = \{ (A_i^v, B_i^v) \mid i \in \{1, 2, \ldots, n\} \}
\]

satisfying the conditions:

1. \( \phi \not= B_i^v \subseteq A_i^v \subseteq V \).
2. the vertices of \( A_i^v \) are fathers of \( v \).
3. If \( A_i^v \cap B_j^v = \phi \), then there is no biconnected component \(^1\) of \( G \) including vertices from both \( A_i \) and \( B_j \).

Note that we do not specify how the sets \( A_i^v \), \( B_i^v \) are chosen.

More on this subject later.

We define now a new locking protocol, called the guard protocol. Each transaction following this protocol (on a guarded graph) must obey the following.

1. A transaction may lock any vertex first. To lock any other vertex \( v \) it must be holding a lock on the vertices in some \( B_i^v \) and must have locked (and possibly unlocked) the vertices of the corresponding \( A_i^v - B_i^v \).
2. A transaction may lock a vertex at most once.
3. A transaction may unlock or lock one at any time.

Clearly, condition 3) is not a restriction; we list it to stress the differences with the two-phase protocol.

Theorem 1: The guard protocol is reliable; that is, it ensures serializability and deadlock-freedom.

Proof: Since the proof is quite lengthy, and since the reader need not understand it in order to proceed, the details of the proofs are deferred to the Appendix.

The guard protocol is a family of non-two-phase locking protocols. Each member of the family relies on a priori information as to how the entities in the database system are organized and accessed. Clearly, the protocol has certain limitations since the structure of the database and the access to the data are strongly controlled. For example, the number of locks required may be larger than the case of the two-phase protocol. On the other hand, deadlock cannot occur no rollbacks are necessary. Moreover, since locking time is potentially reduced, concurrency may increase.

Contrary to common belief, locking is relatively cheap. What is expensive is "what is being locked." Thus, one criterion in selecting a locking protocol may be to choose one that will allow unlocking to occur as soon as possible. Clearly, the two-phase protocol does not satisfy this criterion. The guard protocol, on the other hand, does.

The two-phase approach seems viable in systems in which the number of concurrently running transactions is relatively small. However, given current technology trends, we can expect in the near future systems in which a large number of transactions run concurrently. We believe that in such an environment our approach would be preferable. Further work, however, needs to be done on exploring the tradeoff between the various approaches (e.g., locking time, number of rollbacks necessary to accomplish a specific task, access restrictions). This, however, is beyond the scope of this paper.

IV. The Generality of the Guard Protocol

In this section we demonstrate the versatility of the guard locking protocol. We will show that the tree protocol [7], strong locking protocol [8], and the DAG protocol [9] are special cases of our guard protocol. In addition, a new protocol will also be developed. All the protocols are required to observe points 2 and 3 of our locking protocol; they will differ in the implementation of point 1. In this section we consider only connected directed acyclic graphs. In the following we denote by \( F(w) \) the set of all predecessors (fathers) of vertex \( w \).

A. Rooted Tree Protocol [7]

The database is organized as a rooted tree. A transaction \( T_i \) can lock an entity \( w \not= E(T_i) \) only if it is currently holding a lock on the father of \( w \).

Using the guard protocol, simply define:

\[
guard(w) = \{ (F(w), F(w)) \}.
\]

B. Strong Locking Protocol [8]

The database is organized as a DAG \( G = (V, A) \). Pick \( w \in V \). Define the following binary relation on \( F(w) \):

\[
u_1 \sim u_2 \iff [w, u_1, \text{ and } u_2 \text{ lie in a single biconnected component of } G].
\]

We note that \( \sim \) is an equivalence relation. Let \( C_1, C_2, \ldots, C_m \) be the equivalence classes under this relation. Then a transaction \( T_i \) can lock \( w \not= E(T_i) \) if and only if it is currently holding a lock on (at least) a majority of fathers of \( w \) within a single \( C_i \).

Formally,

\[
guard(w) = \{ (F(w), \{ w \}) \mid \exists C \subseteq C_i \wedge |C| = \lfloor |C_i|/2 \rfloor + 1 \}
\]

where \( |C_i| \) denotes the cardinality of \( C_i \).

C. Rooted DAG Protocol [9]

The database is organized as a rooted DAG. A transaction \( T_i \) can lock an entity \( w \not= E(T_i) \) only if it traversed all the vertices in \( F(w) \) and is currently holding a lock on at least one vertex in \( F(w) \).

Define:

\[
guard(w) = \{ (F(w), \{ w \}) \mid u \in F(w) \}.
\]

D. Dominating Sets Protocol

The database is organized as a DAG, which is broken into a number of biconnected components. To simplify the discussion we consider only the case where \( G \) is a single biconnected component.

We say that \( D^w = \{ d_1, d_2, \ldots, d_q \} \) is a dominating set of the vertex \( w \). If and only if every path in \( G^T \) (\( G^T \) is the graph obtained by reversing the direction of the arcs in \( G \)) from \( w \) to one of the \( G^T \) sinks passes through one of the \( d_i \)’s.
Assume now that with each \( w \in V \) there is associated a pair \(<E(w), D(w)>\) such that:

1. \( E(w) \subseteq V \);
2. if \( D^w \in D(w) \), then \( D^w \) is a dominating set of the vertex \( w \);
3. if \( e \in E(w) \), then every path from \( e \) to \( w \) includes a vertex in each \( D^w \in D(w) \) (this means intuitively that \( E(w) \) is "above" \( D(w) \); although some vertex of some \( D^w \in D(w) \) could be some \( e \in E(w) \)).

Each \( T_i \), which locks \( w \), follows the following protocol.

1. It enters at some weak ancestor (the vertex or its ancestor) of some \( e \in E(w) \).
2. To lock \( w \) it
   a) locked (and possibly unlocked) a predecessor of \( w \),
   b) is holding locks on some \( D^w \in D(w) \).

Formally, let \( X^w \) denote the set of vertices which form some elementary path from some \( e \in E(w) \) to \( w \), and let \( P(w) = 2^V \) denote the set of all such \( X^w \)'s.

Define

\[
guard(w) = \{<D^w \cup X^w, D^w>|D^w \in D(w), X^w \in P(w)>\}.
\]

Then the protocol is a special case of the guard protocol.

V. DISCUSSIONS

We have presented a family of locking protocols that assure serializability and freedom from deadlocks. It was sufficiently general to encompass all the previously developed non-two-phase protocols.

The condition of the guard protocol can provide guidelines as to how one may construct a "custom made" locking protocol for his own database system. By examining the structure of the data and transactions in a particular database system, one may construct the sets \( A^*_i \) and \( B^*_i \) according to some criteria that seem to be of a particular importance for that database system. We are in the process of examining the question of how such design decisions can be automated.

As the number of different "reliable" locking protocols increases, it becomes apparent that more work needs to be done on exploring the tradeoff between the various protocols. One approach is to try and apply the various protocols to some benchmark database systems, to determine the applicability of each of these protocols. This issue, however, is beyond the scope of this paper.

APPENDIX

**Lemma 1:** For each \( T_i \) following the guard protocol the set \( L(T_i) \) consisting of all vertices \( v \) for which \( T_i \) issued the instruction \((T_i, L^X, v)\) during its execution spans a rooted connected subgraph of \( G \).

**Proof:** Simple.

**Lemma 2:** Let \( H = <W, B> \) be a biconnected component of a graph and let \( w_1, w_2, \ldots, w_m \) be an elementary chain such that \( \{w_1, w_m\} \subseteq W \). Then \( \{w_1, w_2, \ldots, w_m\} \subseteq W \).

**Proof:** Simple.

In the sequel we assume that \( V = \{v_1, v_2, \ldots, v_m\} \) is consistently enumerated (topologically sorted), namely if \( u_i \) is an ancestor of \( u_j \) then \( i < j \). For ease of notation we shall also sometimes write \( i \) for \( u_i \).

**Lemma 3:** The relation \(<\) on \( T \) is antisymmetric.
"connecting" $L_{m-2}'$ with $L_0'$. Clearly, $L' = L_0' \cup L_1' \cup \cdots \cup L_{m-1}'$ can be traversed by a simple circuit. Let $H$ be the biconnected component whose vertices form a superset of $L'$. This $H$ satisfies the conditions of the lemma.

**Lemma 5**: Let $H$ be a biconnected component spanned by $W$ and let $T_0$ and $T_1$ be two transactions such that

$$L(T_0) \cap L(T_1) \cap W \neq \phi.$$ 

If $F(T_i) = \min \{k | k \in L(T_i) \cap W\}$, and

$$F(T_0, T_1) = \min \{k | k \in L(T_0) \cap L(T_1) \cap W\},$$

then

$$F(T_0, T_1) \in \{F(T_0), F(T_1)\}.$$  

**Proof**: Clearly, $F(T_0, T_1) < F(T_0)$. $F(T_0, T_1) < F(T_1)$. Assume by contradiction that $F(T_0) < F(T_0, T_1)$ and $F(T_1) < F(T_0, T_1)$. By the locking protocol it follows that $T_1$ "used" some $<A_i, B_i>$ to lock $F(T_0, T_1)$ for $i = 0, 1$. We show that $A_0 \subseteq W$ (and then, of course, also $A_1 \subseteq W$). Indeed assume that $u \in A_0 - W$. As $L(T_0)$ spans a rooted subgraph of $G$ it follows that there exists a chain between $F(T_0)$ and $u$ in $G - \{F(T_0, T_1)\}$ (see Fig. 2). Pick a shortest such chain and extend it by one arc so it "reaches" $F(T_0, T_1)$. By Lemma 2 all the vertices in this chain are in $H$ and thus $u \in W$. It thus follows that $A_0$ and $A_1$ lie in a single biconnected component of $G$ and thus by the locking protocol $A_0 \cap A_1 \neq \phi$. If $w \in A_0 \cap A_1$, then $w < F(T_0, T_1)$ contradicting the definition of $F(T_0, T_1)$.

**Proof of Theorem 1**: (The guard protocol is reliable.) We show, without loss of generality, by induction on $m \geq 2$ that there are no minimal cycles of the form

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

$m = 2$: Proved in Lemma 3.

$m > 2$: Let $H = <W, B>$ be a biconnected component as in Lemma 4. We first show that all $F(T_i, T_{i+1})'$s (subscripts modulo $m$) are distinct (where $F(T_i, T_{i+1})' = \min \{k | k \in L(T_i) \cap L(T_{i+1}) \cap W\}$). Indeed, otherwise we have

$$F(T_i, T_{i+1}) = F(T_j, T_{j+1})$$

for some $i \neq j$ and (at least) 3 distinct transactions share a vertex, implying the existence of a shorter cycle.

Pick now $i_0$ that maximizes $F(T_i, T_{i+1})$. We now have

$$F(T_{i_0}) = F(T_{i_0-1}, T_{i_0}) < F(T_{i_0}, T_{i_0+1})$$

$$F(T_{i_0+1}) = F(T_{i_0+1}, T_{i_0+2}) < F(T_{i_0}, T_{i_0+1}).$$

Thus,

$$F(T_{i_0}, T_{i_0+1}) \notin \{F(T_{i_0}), F(T_{i_0+1})\},$$

contradicting Lemma 5. □

**References**


**Abraham Silberschatz** received the Ph.D. degree in computer science from the State University of New York, Stony Brook, in 1976. He is currently an Associate Professor of Computer Science at the University of Texas, Austin, specializing in the area of concurrent processing. His research interests include operating systems, database systems, distributed systems, and programming languages.

**Zvi M. Kedem** (M’82) received the B.Sc., M.Sc., and D.Sc. degrees in mathematics from the Technion—Israel Institute of Technology, Haifa, Israel.

He is currently an Associate Professor of Computer Sciences at the State University of New York, Stony Brook. During 1982 he is a Visiting Associate Professor of Computer Science at Columbia University. Previously, he was a Postdoctoral Instructor at the Massachusetts Institute of Technology and on the faculty of the University of Texas, Dallas. His research interests include computer graphics, design and analysis of algorithms, database systems, and VLSI.