EXTENDING GRAPH BASED LOCKING PROTOCOLS WITH EDGE LOCKS

Gael N. Buckley
and
Avi Silberschatz

TR-226 June 1983
TR83-226

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

EXTENDING GRAPH BASED LOCKING PROTOCOLS WITH EDGE LOCKS[1]

Gael N. Buckley

Avi Silberschatz

Department of Computer Science

University of Texas at Austin

Austin, Texas 78712

Abstract

A large number of locking protocols use precedence relations among data items to assure the serializability of the database system. These protocols have extended the semantics of the exclusive lock from prohibiting access to a data item to prohibiting access to an entire subgraph. In this paper we argue that combining the use of exclusive locks for these different purposes is ill conceived. We show how these two distinct functions can be separated into the traditional locks operating on the individual data items, and a corresponding set operating on the edges of graph. This separation is illustrated by a general transformation between a given graph protocol and the new type of protocol that retains the same properties as the original protocol. This separation clearly illustrates the relation between shared and exclusive locks in the various protocols, and allows greater concurrency for the database system.

---

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

1. Introduction

A database system has a set of user programs P which can operate on the set of data items D. If it is assumed that each user program (or transaction) when executed alone preserves the consistency of the database, then it is often necessary to ensure that any concurrent execution of several user programs also preserves consistency of the database. A database system that guarantees this property is said to be *serializable* [1].

Most systems ensure serializability by controlling access to each item in the set D via the use of a concurrency control scheme. A very common model for such systems is the use of a *locking protocol*. A transaction must lock a data item before the first access to that item, and unlock it when all accesses to the item are complete. Locking protocols commonly use two types of locks, *exclusive* locks and *shared* locks. Exclusive locks preclude any other lock to be held on a data item, while shared locks permit other shared locks on the same item. Thus a locking protocol is a restriction on when a transaction may lock and unlock each of the data items in the database.

Eswaran et al [1], were the first to discuss the importance of serializability. They also introduced the first useful locking protocol - the two phase protocol. This protocol specifies that no data item can be unlocked until all data items to be accessed have been locked. Eswaran et al have also demonstrated that if no additional information concerning the structure of the database is available, the two phase criterion is necessary and sufficient to ensure serializability. To overcome this problem, several authors have used a precedence order on the database structure to develop locking protocols that are not two phase [6, 4, 3, 8]. We term these *graph protocols*.

The earliest of the graph protocols is the tree protocol [6], used in databases organized (logically or physically) as trees. In this protocol, a transaction can lock any data item in exclusive mode first, and lock a subsequent item in exclusive mode only if its father is locked, and the item has not yet been locked. A transaction can unlock a data item at any time. In this protocol exclusive locks are used for three different purposes. These are:

a) to prohibit access of other transactions from its unlocked data items to the subtree where it will yet lock additional data items, and

b) to prohibit access to an individual data item that it intends to update

c) to prevent deadlock.

Hence, the tree protocol extended the semantics of the exclusive lock.

The idea of restricting access to parts of the database graph was further developed by more general non-two-phase graph protocols, applicable to both acyclic and arbitrary directed graphs. (see, for example [8, 4, 3]). These protocols operate on a database modelled as a directed graph or undirected hypergraph. In general, the restrictions for

these protocols state that an item cannot be locked unless the transaction is holding locks on some set of ancestors (usually some number of fathers of the item), and that this set has some functional relation to other sets for the item to maintain serializability (and perhaps deadlock freedom).

The intent of this paper is to show that the idea of using exclusive locks to prohibit access to the rest of the graph is ill conceived. We will elaborate on some of the problems that may arise in connection with the use of such methods and develop new mechanisms for eliminating these problems. These problems may be eliminated by the use of an *edge* lock, which operates on the precedence relations between data items rather than the data items themselves. In this manner one may distinctly see the utilization of subgraph exclusion of the previous protocols, and use this separation to enhance the behavior of the original protocols. We will give a simple transformation from existing graph protocols to the corresponding edge protocols. While these transformations do not utilize the full power and semantic separation of edge locks, they do increase concurrency for the corresponding protocols.

The remainder of the paper is organized as follows. Section 2 illustrates the motivation for edge locks by presenting various examples of the restricted concurrency inherent in the family of tree protocols. Section 3 introduces edge locks, and shows how the corresponding edge tree protocols resolve these difficulties. Section 4 presents a general transformation between a graph protocol and its corresponding edge protocol, and shows that the properties of the original protocol is preserved. Section 5 concludes the paper with a discussion on the adaptability and extensibility of the edge lock concept.

2. Motivation for Edge Locks

In this section we describe three variations of the tree protocol, and show how the use of the extended semantics of exclusive locks limits the amount of permissible concurrency of each database system. The first variation requires a transaction to issue only exclusive locks, while the other two protocols employ both shared and exclusive locks. We begin by describing the system model.

2.1. The System Model

For the first three protocols and their analogs, the database is organized in the form of a tree, where the vertices of the graph correspond to the data items in the database. One major example of a tree structured database is the IMS system developed by IBM. We represent data items by single capital letters. An arc e_k from item A to item B implies that a transaction may access A before accessing B, and is denoted by <A,B>. The terms father, path, and ancestor are used in their usual graph theoretic sense. The term "the closest ancestor with property p" is defined as the ancestor M of V such that no vertex in the path from M to V has property p except M.

A transaction T_i must access data items according to the precedence relation of the graph as defined by the locking protocol; it must lock an item before the first access to that item, and unlock the item once all accesses are complete. A transaction may only lock a data item once. If T_i will only read access an item, it may request a *shared* or *exclusive* lock (denoted S or X, respectively), otherwise it must request an *exclusive* lock. There may be several S locks on a data item, but one X lock on an item precludes other locks on the item. When the system issues T_i's request for a S lock, an X lock, or an unlock on some data item A, we denote this by LS(A), LX(A), or UN(A), respectively.

2.2. The Tree[X] Protocol

We now describe the first of the tree protocols, which requires that a transaction can issue only exclusive locks [6]. The tree[X] protocol allows a transaction T_i to request an X lock on vertex V if both of the following conditions are satisfied:

1. a. V is the first vertex to be locked by T_i, **or**
   b. the father of V is locked in X mode by T_i

2. V has not yet been locked by T_i.

A vertex can be unlocked at any time.

We now present a simple example which shows that updating a data item unnecessarily restricts access to the subgraph reachable from that data item. Consider the database of figure 2.1 with two transactions T_1 and T_2. T_1 locks data items A and C for 10 time units each, and T_2 locks item B for 100 units. Consider the following history, where the vertical axis denotes when the transactions obtain and release locks.

T_1	T_2
LX(A)	
	LX(B)
	UN(B)
LX(B)	
LX(C)	
UN(B)	
UN(C)	
UN(A)	

In this example, T_1's completion time (or response time) must be 120 units, even though the two transactions lock no data items in common. Indeed, this same behavior is true for all graph protocols using only exclusive locks. Note that T_1's completion time for the 2PL protocol is only 20 units, a large performance difference, especially to a database user.

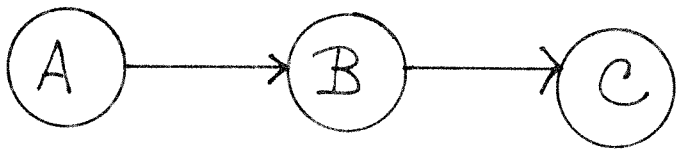

Figure 2.1

2.3. The Tree[XS] Protocol

If a transaction reads many data items and only updates a few, it would be more efficient to allow a transaction to request both shared and exclusive locks. One protocol which allows this is the tree[XS] protocol (this denotes that there is one class of transactions which can issue both X and S locks), presented in [4]. To simplify the presentation of the protocol, we define the function **top**(V,T_i)=W as follows:

a) If there exists an ancestor of V locked in X mode by T_i, let W be the son of the closest such ancestor; otherwise

b) If the first vertex locked by T_i is locked in S mode by T_i, let W be that vertex; otherwise

c) let W = nil.

The Tree[XS] protocol allows a transaction T_i to request a lock on vertex V if both of the following conditions hold:

1. a. V is the first vertex locked by T_i **or**

   b. i. **top**(V,T_i) $\neq$ nil **and**

      ii. all vertices except V in the path(**top**(V,T_i),V) are locked in S mode by T_i, **and**

      iii. all vertices R with **top**(R,T_i)=**top**(V,T_i) have not yet been unlocked.

2. V has not yet been locked by T_i.

An example of these conditions are shown in figure 2.2, where T_i has locked A,B,C, D, and it wishes to lock E. Top(E,T_i) is B, and hence D must be locked in S mode, and B, C, D must still be locked by T_i.

We now illustrate the case where a transaction using the tree[XS] tree protocol and only requiring shared locks must either not unlock any item, or must issue an exclusive lock to prohibit access to the subgraph. Hence, it must restrict access to the data field of

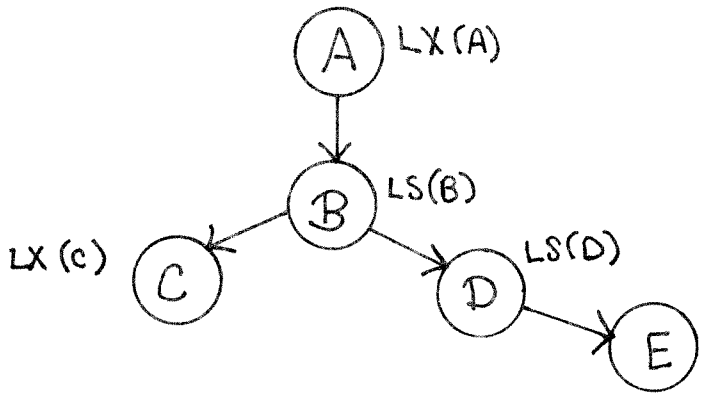

Figure 2.2

the root of the subgraph, even though the function is not intended. Consider again the database of figure 2.1, where transaction T_1 reads A, B, then C for 10 units each, T_2 writes A and reads B for 1 unit each, and T_3 writes C for 50 units. If T_1 locks A, B, and C in S mode, then it cannot unlock A until it locks C. Consider the following history:

T_1	T_2	T_3
LS(A)		LX(C)
LS(B)		
		UN(C)
LS(C)		
UN(A)		
UN(B)		
	LX(A)	
	LS(B)	

In this case, T_2 must wait 40 additional units until T_1 can lock C. Notice this lock history can also be generated by the 2PL protocol. In order to allow T_1 to unlock A earlier, T_1 can lock B in X mode, shown by the following history:

T_1	T_2	T_3
LS(A)		LX(C)
LX(B)		
UN(A)		
	LX(A)	
		UN(C)
LS(C)		
UN(B)		
	LS(B)	

This history has T_2 wait to access B until T_1 removes the X lock from B. Either option unnecessarily reduces concurrency.

Notice that the two protocols described use an X lock on a data item as the blocking factor to prohibit access to the subgraph rooted at the item. Both of these problems can

be resolved by separating the semantics of graph access from data item access, as we show in the next section.

2.4. The Tree[X,S] Protocol

Finally, we show that these problems remain even when transactions can be separated into sets of read-only transactions and update transactions. For this we use the tree[X,S] protocol given in [4]. Upon entry into the system, a transaction which only reads data items is classified as read-only, and only issues S locks. All other transactions are classified as update transactions, and can issue only X locks. The tree[X,S] protocol allows a transaction to request a lock on vertex V in mode M iff:

1. V is the first vertex locked by T_i, **or**
   father(V) is locked in mode M;
2. V has not been locked by T_i;
3. all update transactions must lock the root of the tree first.

As before, a vertex can be unlocked at any time.

Consider figure 2.1 with update transactions T_1 and T_3, and one read-only transaction T_2. T_1 writes B for 40 units, T_2 reads A and B for 5 units each, and T_3 writes A for 5 units. Using the tree[X,S] protocol, T_1 must begin by locking A, the root of the tree. Consider the following history:

T_1	T_2	T_3
LX(A)		
LX(B)		
UN(A)		
	LS(A)	
UN(B)		
	LS(B)	
	UN(A)	
		LX(A)
		UN(A)

Notice that T_2 must retain the lock on A until T_1 releases B, even though A is no longer being read. This again shows how the restrictions on the graph decrease concurrency by preventing access to a data item. We now show how these concepts can be separated, and how they increase the concurrency of the systems illustrated above.

3. The New Edge Tree Protocols

In this section we define a new type of lock, called *edge locks*, and show how the problems presented above can be resolved by the use of these new type of locks. Edge locks have also been used by Korth [5] to prevent deadlocks for protocols operating on multiple granularity DAGS [2]. The edge locks we introduce also operate on edges, but are used for a completely different purpose. As the tree protocols enforce serializability by holding locks on a set of data items, the edge tree protocols enforce serializability by holding locks on a set of edges. We present a simple transformation from the original protocols to the new protocols, and generalize the transformation in the next section. We begin by defining terminology for the use of edge locks.

All terms have direct analogies to the original graph protocol. We denote an edge from vertex A to vertex B as <A,B>. An edge <A,B> may be locked in *exclusive* or *shared* mode (denoted EX or ES, respectively). An edge may be locked with several ES locks simultaneously, but may only have one lock if the lock mode is EX. The lock and unlock instructions issued by the system on edge <A,B> are represented by LEX(<A,B>), LES(<A,B>), and UNE(<A,B>). An edge may only be locked once for the transformations of the original protocols.

Using terms in the same manner as the previous section, we define the father of edge <A,B> as the edge <M,A>, where vertex M is the traditional father of vertex A. The use of the terms ancestor, path, and closest ancestor with property p are constructed analogously. Finally, any vertex V with no incoming edge has a dummy edge <0,V> from a dummy vertex to V, simply to avoid special cases for the protocols.

We now present the edge protocols corresponding to the various tree protocols presented in section 2, and demonstrate how edge locks resolve the blocking problems encountered in each of the earlier examples. The three new protocols look nearly identical to the old protocols, the only difference being that vertices are now replaced by their corresponding edges.

We do not present in this section the proofs that the new edge protocols ensure serializability. This is done in section 4, where we present general transformation rules and correctness rules which apply to the edge tree protocols presented below.

All the following edge tree protocols allow a transaction to request a lock on vertex V in mode M only when the transaction can request a lock on edge <V,R> in mode EM.

3.1. The Edge Tree[X] Protocol

We give the conditions necessary to lock an edge for the new protocol, and show how the blocking problem presented in section 2.2 is eliminated by use of the corresponding edge protocol.
The condition to request a lock on edge <M,V> under the edge tree[X] protocol is:

1. <M,V> is the first edge to be locked by T_i **or**
   <L,M> is locked in EX mode by T_i
2. <M,V> has not been locked by T_i.

We now present the corresponding edge lock history for the problem given in section 2.2. We first redraw figure 2.1 to construct the dummy edge and label edges in the figure to correspond to the histories.

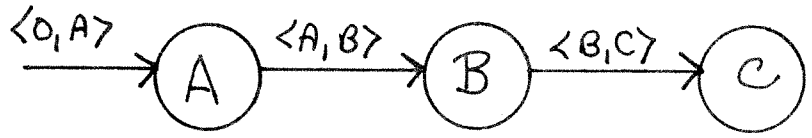

Figure 3.1

The corresponding history for the tree[X] protocol is:

T_1	T_2
LEX(<0,A>)	
	LEX(<A,B>)
LX(A)	
	LX(B)
	UNE(<A,B>)
LEX(<A,B>)	
LEX(<B,C>)	
LX(C)	
	UN(B)

This example illustrates how the use of edge locks allows T_1 to bypass T_2, and dramatically decrease its completion (or response) time.

3.2. The Edge Tree[XS] Protocol

The original tree[XS] protocol was defined in terms of a function on the ancestors of a data item. Similarly, the edge protocol is defined in terms of the edge ancestors of an edge, where edge <Q,M> is the father of edge <M,V>. Ancestors are constructed in the same manner. We define **etop**(<M,V>,T_i) = W in the same way as before:

a) If there exists an ancestor of <M,V> locked in EX mode by T_i, let W be the son of closest such ancestor, otherwise

b) If the first edge locked by T_i is locked in S mode by T_i, let W be that edge, otherwise

c) let W = nil.

We now present the conditions which must be satisfied for T_i to request a lock on edge <M,V>:

1. a. <M,V> is the first edge to be locked by T_i **or**

   b. i. **etop**(<M,V>,T_i) $\neq$ nil **and**

      ii. all edges except <M,V> in the path(**etop**(<M,V>,T_i)) are locked in ES mode **and**

      iii. all edges <R,S> with **etop**(<R,S>,T_i)=**etop**(<M,V>,T_i) are locked by T_i

      iiii. If T_i has locked the vertex S for any edge <R,S> locked in ES mode fulfilling conditions ii or iii above, T_i still has a lock on S.

2. <M,V> has not been locked by T_i.

We now give the corresponding edge history for the problem given in section 2.3.

T_1	T_2	T_3
LES(<0,A>)		
LS(A)		
		LEX(<B,C>)
		LX(C)
		UNE(<B,C>)
LES(<A,B>)		
LS(B)		
LEX(<B,C>)		
UNE(<0,A>)		
	LEX(<0,A>)	
	LES(<A,B>)	
UN(A)		
UN(B)		
	LX(A)	
	LS(B)	
		U(C)
	UN(A)	
	UN(B)	

Notice that T_2 no longer waits on the use of C, and yet the EX lock on <B,C> establishes the necessary access restriction that T_1 requires, without delaying T_2. One can easily see from this example that if transactions with intersecting sets of data items split into disjoint subgraphs, the completion time of each transaction will decrease. Only when the yet to be locked data items reside in the same subgraph will the transactions be delayed, as is necessary for serializability.

3.3. The Edge Tree[X,S] Protocol

Finally, we give the conditions necessary to lock an edge in the tree[X,S] protocol. To be consistent with the original protocol, an update transaction issues only EX edge locks, and a read-only transaction issues only ES locks. To request a lock on edge <M,V> in EX mode:

1. <M,V> is the dummy edge for the root V **or**
   <Q,M> is locked in EX mode by T_i
2. <M,V> has not been locked by T_i

To lock <M,V> in ES mode:

1. <M,V> is the first edge to be locked by T_i **or** <Q,M> is locked in ES mode by T_i
2. <M,V> has not been locked by T_i

An edge or vertex may be unlocked at any time.

We now present the new history for the problem given for the tree[X,S] tree protocol.

T_1	T_2	T_3
LEX(<O,A>)		
LEX(<A,B>)		
UNE(<O,A>)		
	LES(<O,A>)	
LX(B)		
UNE(<A,B>)		
	LS(A)	
	LES(<A,B>)	
	UNE(<O,A>)	
		LEX(<O,A>)
	UN(A)	
		LX(A)
UN(B)		
	LS(B)	
	UNE(<A,B>)	
	UN(B)	
		UN(A)

This new history allows T_3 to write A as soon as T_2 has finished reading it, without waiting for T_1 to release B.

Through these three examples, we have shown that separation of function both increases clarity and extracts additional concurrency from this set of graph protocols. In the next section we demonstrate how it is possible to achieve this with any graph protocol.

4. Deriving Edge Protocols from Graph Protocols

We now present the general transformation from any graph protocol to its corresponding edge protocol. We demonstrate that if the original protocol ensures serializability and possibly deadlock freedom, then the derived edge protocol ensures the same properties. This transformation is done by replacing vertices in the original protocol by edges. Since all graph protocols use information about the graph, we first define the graph relations used, and their corresponding edge relations.

4.1. Transformation Rules

In a general directed graph, vertex M is a father of V if there exists an edge from M to V. There is a path from M to V if there is a path from M to Q and Q is a father of V. The length of a path is the number of edges in the sequence. A vertex M is an ancestor of V if there is a path from M to V. The definitions for edge relation are identical. Edge <Q,M> is a father of edge <M,V> if <Q,M> is an incoming edge to vertex M and <M,V> is an outgoing edge from vertex M. There is a path from edge <Q,R> to <M,V> if there is a path from edge <Q,R> to <L,M> and <L,M> is a father of <M,V>. The length of an edge path is the number of edges in the path minus 1. An edge <Q,R> is an ancestor of <M,V> if there is a path from <Q,R> to <M,V>.

The database systems used in graph protocols operate upon database structured as trees, directed acyclic graphs, or general directed graphs. Every protocol operates on the concept of some form of separation between unlocked items and items yet to be locked, using some combination of the graph relations presented above. Additionally, some protocols proscribe special status for particular vertices, such as the first vertex locked in the tree protocols presented earlier. To derive an edge protocol from its corresponding graph protocol, observe the following steps:

1. Replace every reference to a vertex with some edge, such that any graph relation between vertices in the original protocol are maintained in the transformed edge protocol. The first object locked by T_i must be an edge. Thus the derived protocol specifies when an edge may be locked or unlocked. The two steps below define when a vertex may be locked or unlocked.

2. A vertex V can be locked in mode M by transaction T_i if V has not been locked by T_i, V is not presently locked in an incompatible mode by some other transaction, and the edge protocol would allow T_i to request a lock on some outgoing edge <V,Q> in the corresponding edge mode EM. This edge <V,Q> need not actually be locked by T_i.

3. A vertex V presently locked can be unlocked only when unlocking a dummy edge $<V,0_V>$ locked in the same edge mode as V does not prohibit locking any future edge using the edge protocol.

For the graph protocols described in this paper, this last condition is important only in the tree[XS] protocol. This occurs when the incoming edge is locked in ES mode, and the vertex itself is locked in X mode. Hence, the dummy edge is considered locked in EX mode, and would have an **etop** identical to the outgoing edges from the vertex.

The transformation has been used to derive the three edge tree protocols given in section 3 from the graph protocols given in section 2. We now present one additional graph protocol, the guard protocol of [7].

With each vertex V, associate a set of pairs of sets of vertices, denoted by $\{[A_{V1},B_{V1}],...[A_{VM},B_{VM}]\}$, where $A_{Vi} \subset B_{Vi}$ and $A_{Vi} \cap B_{Vj} \neq 0$. If $M \in B_{Vi}$, then M is a father of V.
Transaction T_i can request a lock on V in X mode iff:

1. V is the first vertex to be locked by T_i **or**
   A_{Vi} is locked in X mode by T_i, and B_{Vi} has at some time been locked in X mode by T_i.
2. V has not been locked by T_i.

The corresponding edge guard protocol is defined as follows:

With each edge $<P,V>$ associate a set of pairs of sets of edges, denoted by $\{[A_{<P,V>1},B_{<P,V>1}],...[A_{<P,V>M},B_{<P,V>M}]\}$, where $A_{<P,V>i} \subset B_{<P,V>i}$ and $A_{<P,V>i} \cap B_{<P,V>j} \neq 0$. All edges in $A_{<P,V>i}$ are fathers of $<P,V>$.
A transaction T_i can request a lock on edge $<P,V>$ in EX mode iff:

1. $<P,V>$ is the first edge to be locked by T_i **or**
   $A_{<P,V>i}$ is locked in EX mode and $B_{<P,V>i}$ has at some time been locked in EX mode by T_i.
2. $<P,V>$ has not been locked by T_i.

4.2. Proofs of Serializability and Deadlock Freedom

We now present the proof of serializability for this general transformation. We do so by showing that whatever properties the original graph protocol had are maintained by the new protocol. To do so, we need to perform two transformations on the graph that the graph protocol operates on, and show that these maintain the structure of the graph. As in section 3, we assume that the original graph already has an incoming edge for every vertex to enable locking of that vertex.

There are three types of graphs that protocols operate on: trees, acyclic graphs (rooted or otherwise), and arbitrary directed graphs. To a graph $\mathcal{G}$ we add a new vertex 0_V for every vertex V, with an edge $<V,0_V>$ from V to 0_V. We term this graph $\mathcal{G}'$. Since there are no outgoing arcs from 0_V, this addition does not change the graph structure,

nor introduce any new paths between vertices in $\mathcal{G}$. We then construct the dual of $\mathcal{G}$', denoted by $\mathcal{G}$'', by transforming each edge <A,B> in $\mathcal{G}$' into a vertex (<A,B>), with an edge from one vertex to another if the edges they represent are incoming and outgoing edges of the same vertex in $\mathcal{G}$'.

We now prove that if there is a path from one vertex to another in $\mathcal{G}$'', then there was a path of the same length from one edge to another in $\mathcal{G}$'.

Lemma 1: There is a path from vertex (<A,B>) to (<C,D>) in $\mathcal{G}$'' if and only if there is a path of the same length from edge <A,B> to <C,D> in $\mathcal{G}$'.

Proof: Proof by induction on the length of the path.

i=1 There is an arc from (<A,B>) to (<C,D>) if and only if edges <A,B> and <C,D> are incoming and outgoing edges of the same vertex in $\mathcal{G}$', by construction of $\mathcal{G}$''. Both of these paths are of length 1.

i>k From the inductive assumption, there is a path of length k from (<R,Q>) to (<C,D>) if and only if there is a path of length k from <R,Q> to <C,D> in $\mathcal{G}$'. There is an arc from (<A,B>) to (<R,Q>) in $\mathcal{G}$'' if and only if <A,B> and <R,Q> are incoming and outgoing edges of the same vertex in $\mathcal{G}$', by construction of $\mathcal{G}$''. Both of these paths are of length 1, and hence the paths from (<A,B>) to (<C,D>) in $\mathcal{G}$'' and <A,B> to <C,D> in $\mathcal{G}$' are of length k+1.

We now present two definitions which are necessary in proving serializability and deadlock freedom.

Definition 1: A *history* is the time ordered sequence of the lock and unlock operations of a set of transactions $\mathcal{T} = \{T_1,...,T_N\}$ performed on the database. There is a *dependency* between transaction T_i and T_j if both transactions lock vertex V, and at least one transaction locks V in X mode. If T_i locks V first, we say $T_i < T_j$. A history is serializable if and only if the $<$ relation on the set $\mathcal{T}$ is acyclic.

Definition 2: If T_i holds a lock on vertex V, and T_j requests a lock on V, where the lock request and lock are not both S mode, the lock request must be delayed until T_i unlocks V; in this case we say that T_j *waits-for* T_i. A protocol is deadlock free if all wait-for relations produced are acyclic.

Theorem 1: If the original graph protocol is serializable, then the transformed edge protocol is serializable.

Proof: Consider a history consisting of edge and vertex locks obtained by executing the edge protocol on some graph $\mathcal{G}$. Construct the graph $\mathcal{G}'$ in the manner given above. Execute the given history on this graph, with the addition that edge $<V,0_V>$ is locked in mode EM by T_i immediately after vertex V is locked in mode M by T_i, and edge $<V,0_V>$ is unlocked immediately after vertex V is unlocked. This locking is allowed by the edge protocol since a vertex can be locked only when an outgoing edge can be locked. The unlocking is allowed by restriction 3 of the edge transformation.

Since the dependencies and wait-for relations for the vertices in the original history are duplicated by the edge locks on the edges $<V,0_V>$, we can remove the vertex locks from the new history. Apply this sequence to $\mathcal{G}''$. The edge locks in the sequence are once again vertex locks when applied to $\mathcal{G}''$. By Lemma 1, the transformation from $\mathcal{G}'$ to $\mathcal{G}''$ maintains exactly the same graph relations between vertices as the relations between the edges in $\mathcal{G}'$. The transformation from locking a vertex in the original graph protocol to locking an edge in the edge protocol maintains the same graph relations between edges as there were between vertices. Hence, we have obtained the original graph protocol executing on a set of vertices on $\mathcal{G}''$, a graph of the same structure as $\mathcal{G}$. Hence, if the original protocol ensures serializability, the corresponding edge protocol ensures serializability on the graph $\mathcal{G}''$. Since any $<$ relations between the dummy edges $<V,0_V>$ are acyclic, and these duplicate the $<$ relations between vertices locked in the graph $\mathcal{G}'$, the entire set of dependencies of the edge protocol on the graph $\mathcal{G}'$ are acyclic, and hence serializable.

Theorem 2: If the original graph protocol is deadlock free, then the transformed edge protocol is deadlock free.

Proof: Construct the graph $\mathcal{G}''$ and the same three sequences of locks and unlocks as

given in the proof of Theorem 2. Again we obtain the original graph protocol executing on a set of vertices of $\mathcal{G}$'', a graph of the same structure as $\mathcal{G}$. If the original protocol produces an acyclic wait-for relation, the corresponding edge protocol has an acyclic wait-for relation on the graph $\mathcal{G}$''. Since any wait-for relations between edges $<V,0_V>$ duplicate the wait-for relations of the vertex locks in the edge history of $\mathcal{G}$, the entire wait-for relation remains acyclic.

To illustrate the transformations given above, we use this graph and history given for the edge tree[X] protocol in section 3.1. The graph $\mathcal{G}$ is figure 2.1, and graphs $\mathcal{G}$' and $\mathcal{G}$'' are presented below, in figure 4.1.

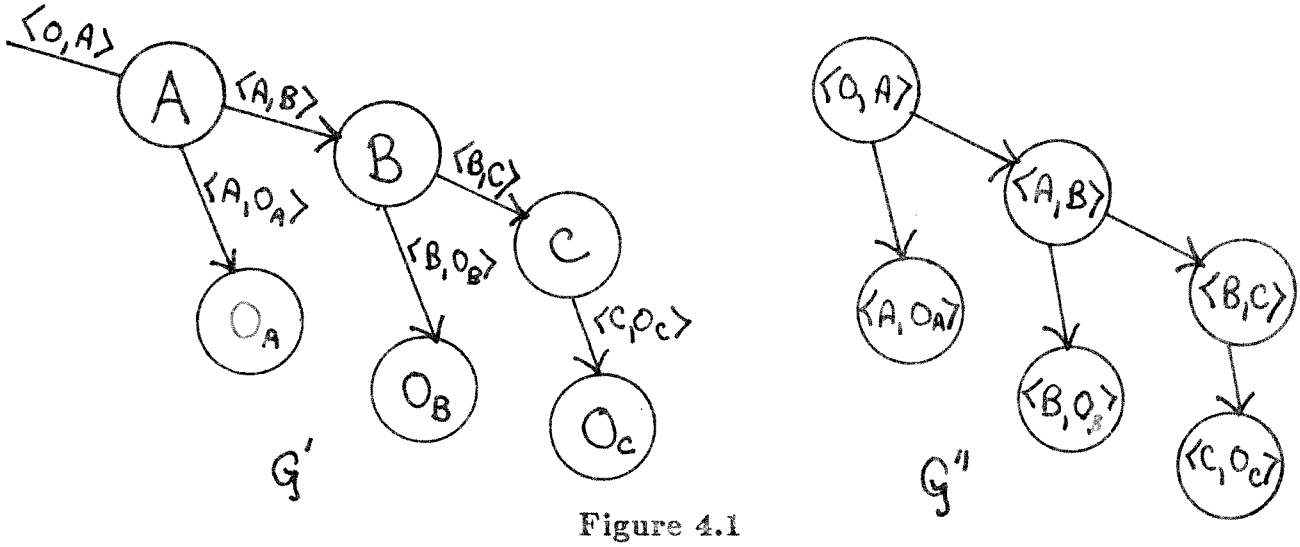

Figure 4.1

The histories are arranged in the same order as the proof: the original edge history on the graph $\mathcal{G}$, the extended edge history on the graph $\mathcal{G}$', and the history with the vertex locks and unlocks removed, which is to be applied to the dual of $\mathcal{G}$'.

T_1	T_2
LEX(<0,A>)	
	LEX(<A,B>)
LX(A)	
	LX(B)
UN(A)	
	UNE(<A,B>)
LEX(<A,B>)	
UNE(<0,A>)	
LEX(<B,C>)	
LX(C)	
UNE(<A,B>)	
	UN(B)
UN(C)	
UNE(<B,C>)	

T_1	T_2
LEX(<0,A>)	
	LEX(<A,B>)
LX(A)	
LEX(<A,0_A>)	
	LX(B)
LEX(<B,0_B>)	
UN(A)	
UNE(<A,0_A>)	
	UNE(<A,B>)
LEX(<A,B>)	
UNE(<0,A>)	
LEX(<B,C>)	
LX(C)	
LEX(<C,0_C>)	
UNE(<A,B>)	
	UN(B)
	UNE(<B,0_B>)
UN(C)	
UNE(<C,0_C>)	
UNE(<B,C>)	

T_1	T_2
LEX(<O,A>)	
	LEX(<A,B>)
LEX(<A,O_A>)	
LEX(<B,O_B>)	
UNE(<A,O_A>)	
	UNE(<A,B>)
LEX(<A,B>)	
UNE(<O,A>)	
LEX(<B,C>)	
LEX(<C,O_C>)	
UNE(<A,B>)	
	UNE(<B,O_B>)
UNE(<C,O_C>)	
UNE(<B,C>)	

This example shows that the resulting sequence of edge locks is once again the original tree[X] protocol, operating on a tree extended by the addition of the vertices corresponding to the $<V,0_V>$ edges.

5. Conclusion

We have introduced a new method of separating the several functions that exclusive and shared locks represent in non-two-phase graph protocols. This separation admits more concurrency in the database system. We have shown that there exists simple transformations from existing graph protocols to their corresponding edge protocols which admits the same behavior as the original graph protocol.

References

1. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System." *Communications of the ACM 10*, 11 (Nov. 1976), 624-723.

2. Gray, J., Lorie, R.A., Putzolu, G.R., Traiger, I.L. Granularity of Locks and Degrees of Consistency in a Shared Data Base. Tech. Rept. RJ1654, IBM Research - San Jose, 1975.

3. Kedem, Z., Silberschatz, A. Non-Two-Phase Locking Protocols with Shared and Exclusive Locks. Proc. Sixth International Conference on Very Large Data Bases, October, 1980.

4. Kedem, Z. and Silberschatz, A. "Locking protocols: from Exclusive to Shared Locks." *Journal of the ACM* (to appear).

5. Korth, H. "Deadlock Freedom Using Edge Locks." *ACM Transactions on Database Systems 7*, 4 (December 1982), 632-652.

6. Silberschatz, A., Kedem, Z. "Consistency in Hierarchical Database Systems." *Journal of the ACM 27*, 1 (Jan. 1980), 72-80.

7. Silberschatz, A., Kedem, Z. "A Family of Locking Protocols for Database Systems that are Modeled by Directed Graphs." *IEEE Transactions on Software Engineering 8*, 6 (November 1982), 558-562.

8. Yannakakis, M. "A Theory of Safe Locking Policies in Database Systems." *Journal of the ACM 29*, 4 (July 1982), 221-244.