USE OF TRANSACTION STRUCTURE FOR IMPROVING CONCURRENCY

Wei Zhao
Krithivasan Ramamritham
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 86-2

March 1985

Use of Transaction Structure For Improving Concurrency*

Wei Zhao† *and* Krithi Ramamritham††

ABSTRACT

Information about the manner in which transactions access objects in a database has been ignored by the majority of the literature on database consistency theory. Consequently, the potential concurrency of the system is not fully exploited. In this paper, we propose a locking scheme in which the underlying structure of the database, as well as the locking behaviour of transactions, are both considered, thus achieving higher concurrency than is made possible by earlier schemes. This is done while ensuring both serializability and deadlock freedom of the system.

Intuitively, the consistency theory and the deadlock theory should be closely related because both are theories on resource allocation. In this paper we formalize this intuition and prove that the dependency relation developed in consistency theory is equivalent to the wait-for relation used in deadlock theory if a locking scheme can guarantee both serializability and deadlock freedom.

Key Words and Phrases: Database System, Concurrency Control, Deadlock, Transactions.
CR Categories: D.4.1, H.2.4

1. Introduction

A database is said to be *consistent* if it meets a set of pre-specified constraints. A *transaction* is a sequence of operations performed on the database such that complete execution of the sequence will leave the database in a consistent state if the database was consistent before the transaction.

When a set of transactions execute concurrently, the database may be left in an inconsistent state or some transactions may receive an inconsistent view of the database if the operations of the transactions are not properly interleaved. To prevent this, the execution ordering of transaction operations has to be controlled. This ordering of execution of operations for a set of transactions is known as a *schedule* of the operations in the transactions.

Given that every transaction leaves the database in a consistent state if executed by itself, then if a set of transactions are executed *serially*, then they will leave the database in a consistent state and each of the transactions will receive a consistent view of the database. Such schedules are called *serial schedules*. A schedule is said to be *serializable* if it leaves the database in a consistent state and lets each transaction receive the same view of the database as some serial schedule.

---

* This work was partly supported by the Office of Naval Research of United States under contract N00014-85-K-0398, by the National Science Foundation of United States under grant DCR-8500332, and by the Australian Research Council under grant A48830314.

† Department of Computer Science, University of Adelaide, Adelaide, SA 5001, Australia.

†† Department of Computer and Information Science,University of Massachusetts, Amherst, MA 01003, United States of America.

Although, besides serializable schedules, there could be other (types of) schedules which could leave the database in a consistent state and let each transaction receive a consistent view of the database, the majority of the literature in this area concerns sufficient conditions for serializable schedules. This paper is no exception. The reason for looking for different sufficient conditions for the serializable schedules is twofold: some conditions may be easier to implement than others; and some may achieve higher concurrency than others.

Eswaran *et al.* [Eswaran *et al.* 76] first gave a sufficient condition, two phase locking, for serializable schedules. Stated simply, the two phase locking condition says that if each transaction starts unlocking entities only after it locks all the entities it needs, then the schedules are serializable. The two phase locking scheme guarantees the serializable schedules. However, it could result in reduced concurrency. For example, consider the "Accounts" database in a bank. A transaction, observing the two phase locking scheme, calculates the interest for each account and summaries them. Using the two phase locking scheme, this transaction will have to lock all the accounts, then release them. Meanwhile it is impossible for other transactions to run on those accounts, even if an another transaction is willing to just follow the interests calculation transaction (i.e., only locking an account after the interests calculation transaction has finished using it). This suggests that if we utilize the behaviour of transactions, it should be possible to improve concurrency. In this paper, we address this suggestion and present a locking scheme based on this suggestion.

[Eswaran *et al.* 76] formally gave a general condition for the serializable schedules: If for a given schedule of a set of actions the dependency relation is acyclic, then the schedule is serializable. Yannakakis [Yannakakis 84] further proved that the acyclicity of the dependency relation under a locking scheme is a necessary and sufficient condition for the sub-schedules (i.e., schedules involving a subset of the transactions) to be serializable. Although this general condition has been discovered for a database with arbitrary structure, a general locking scheme which can achieve higher concurrency than the two phase scheme has not been reported.

However, research efforts in this area continued with the consideration of the underlying structures of database systems. Silberschatz and others [Silberschatz and Kedem 80], [Silberschatz and Kedem 82], and [Kedem and Silberschatz 83] suggest a family of non-two-phase locking schemes for databases structured as directed acyclic graphs. In [Mohan *et al.* 85], the issue of lock conversion for that family of locking schemes is discussed. Korth [Korth 82a], [Korth 82b], and [Korth 83] proposes non-two-phase locking schemes for database systems with acyclic structures [Gray 78]. The difference between the structures of Silberschatz *et al.* and Korth is that in Korth's work, data are not associated with internal (i.e., non-terminal) nodes, but in the former, they are. All schemes proposed in [Silberschatz and Kedem 80], [Silberschatz and Kedem 82], [Korth 82a], and [Korth 82b] generate serializable schedules and guarantee deadlock freedom.

On the other hand, we not only consider the underlying structure of the database, but also the locking and unlocking behaviour of transactions. As a result we are able to achieve higher concurrency than the above schemes.

It is interesting to note that several locking schemes such as [Silberschatz and Kedem 80], [Silberschatz and Kedem 82], [Korth 82a], [Korth 82b] and ours can guarantee serializability and deadlock freedom at the same time. Intuitively, deadlock theory and consistency theory should have some tight relationship because both are based on the ordering of the resource allocations. We prove in this paper that the dependency relation developed in consistency theory and the wait-for relation used in deadlock theory are equivalent if a locking scheme guarantees both serializability and deadlock freedom. This results in simpler proofs of the correctness of our scheme. Further, it is hoped that this result may lead an effective combination of the two theories.

The rest of this paper is organized as follows: Section 2 defines our database and transaction model. Section 3 deals with the locking scheme. Section 4 shows the equivalence

between the dependency and wait-for relations and proves the correctness of our locking scheme. Section 5 compares our study and the previous work, suggests the possible extensions and discusses the meaning of the equivalency between dependency relation and wait-for relation.

2. A Formal Transaction Model

2.1. The Structure of the Database

The database is structured as a tree. Data are associated only with the leaves of the tree and the internal nodes represent components of the database at different granularities. For example, a national bank's "accounts" database may consist of one sub-database for each city; each city's database having sub-databases, one per branch in that city; each branch having one sub-databases per account type, etc. Leaves are numbered sequentially, left to right, starting at 1 and ending at the "maximum leaf". The identity of a leaf is the number associated with it.

2.2. Ordered and Disordered Transactions

We recognize two classes of transactions, ordered and disordered. Intuitively, an ordered transaction locks a portion of the database, that is, leaves, in the order in which they are numbered. A disordered transaction, on the other hand, may lock data without this limitation. A disordered transaction in this paper is the same as the traditional "transaction", as defined in [Korth 82a] and [Korth 82b]. Transactions in each class observe locking rules applicable to that class. These rules are introduced in the next section. First, we formally define the transactions and related terms.

A transaction is *ordered*, if

a) it locks leaves in the order they are numbered,
b) it unlocks leaves in the same order it locks them, and
c) in locking internal nodes, it observes a top-down locking scheme, described in Section 3.

The leaves locked by an ordered transaction form its *range*: [L, U], where L is the leaf with minimum identity locked by the ordered transaction and U is the leaf with the maximum identity. Thus, the range of an ordered transaction identifies the contiguous set of leaves that will be locked by the transaction.

Associated with each sub-tree accessed by an ordered transaction are two pairs of bounds: The *dynamic lower/upper bounds*, denoted as [DL, DU] and the *static lower/upper bounds*, [SL, SU]. They are defined as follows:

a) SL, the static lower bound for an ordered transaction, is the leaf with the smallest dentity locked by the transaction, in the sub-tree;
b) SU, the static upper bound for an ordered transaction, is the leaf with the largest identity locked by the transaction, in the sub-tree;
c) DL, the dynamic lower bound for an ordered transaction, is the leaf with the smallest identity *currently* locked by the transaction, in the sub-tree;
d) DU, the dynamic upper bound for an ordered transaction, is the leaf with the largest identity *currently* locked by the transaction, in the sub-tree.

Let SL(T) denote SL for transaction T. Similarly for SU, DL, and DU.

Notice that for a given sub-tree, the leaves in [DL, DU] are contained in [SL, SU]. Also, the range [L, U] defined above is equivalent to [SL, SU] for the whole tree. [SL, SU] for a given sub-tree is equal to the intersection of [L, U] and all the leaves of the sub-tree.

Example 1 Consider the tree in Figure 1. If an ordered transaction T locks leaves 4 to 7 in that order, then we have the following:

L = 4, U = 7;
SL = 4 and SU = 7, for the sub-tree rooted at c;
SL = 4 and SU = 5, for the sub-tree rooted at f;
SL = 6 and SU = 7, for the sub-tree rooted at g.

Suppose, T first places locks on leaves 4 and 5.
Then DL = 4 and DU = 5 for the sub-tree rooted at f;
If after accessing leaf 4, T unlocks it, then
DL = 5 and DU = 5 for the sub-tree rooted at f. []

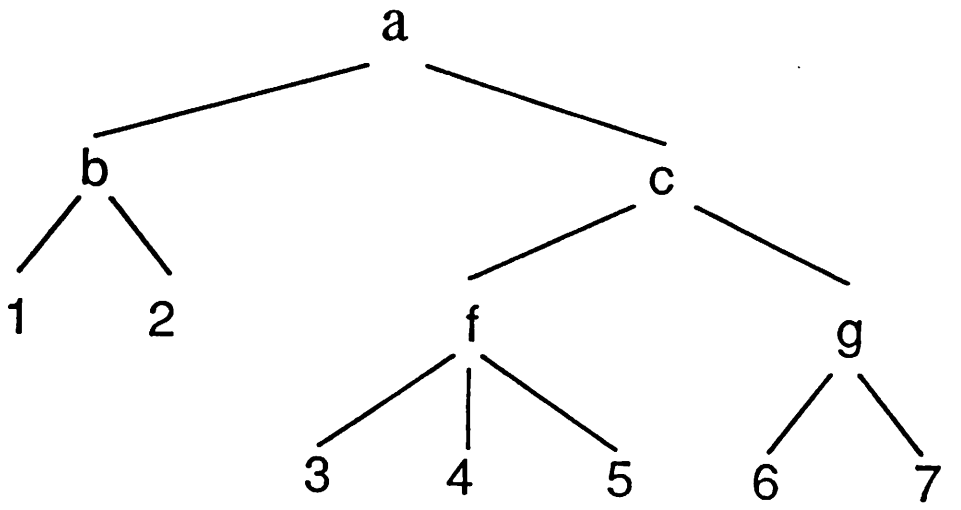

Figure 1: Tree Representing a Database

For disordered transactions, we assume that information about the static lower and upper bounds of the transaction with respect to a sub-tree are available when the transaction begins accessing the sub-tree. Because a disordered transaction can lock the nodes in [SL, SU] in any order, no [DL, DU] is associated with such a transaction.

2.3. Follower Relation for Transactions

A transaction is said to be *active* in a sub-tree if its [SL, SU] covers more than one sub-tree of that sub-tree. That is, it will lock (perhaps with intention locks) at least two nodes at the level right below the root of the sub-tree. For example, transaction T in Example 1 may be active in the sub-trees rooted at c, f, and g, but cannot be active in the tree rooted at a.

For the purpose of controlling the execution of concurrent transactions, we define a relationship, called the *Follower Relation*, denoted as FR, among the transactions in a sub-tree. This relation holds between two transactions only if at least one of the transactions is ordered and at least one of the transactions is active in that sub-tree.

The purpose of the FR relation is to force an order among conflicting actions. Since the locking rules for disordered transactions are sufficient to impose an ordering between two disordered transactions, there is no FR relation among any two disordered transactions. Also, in

a given sub-tree, two transactions are not related by the FR relation unless at least one is active in that sub-tree. This ensures that ordering is imposed only when needed to avoid conflicts.

When a transaction arrives at ((intention) locks) the root of a sub-tree, an FR relation will be established between the newly arrived transaction and those which are already active in the sub-tree: Let T_O be an old active transaction and T_n a newly arriving transaction. We define that

$$\underline{\text{if } (SL(T_n) \leq SU(T_O)) \text{ and } (SU(T_n) \geq SL(T_O)) \text{ then } T_n \text{ FR } T_O.}$$

Thus, the two transactions are related through FR if and only if the leaves accessed by them intersect. In addition, all transactions observe the following rule: In a sub-tree, if T_i FR T_j, then T_i can never lock a leaf which has an identity larger than or equal to DL of T_j (or SL if T_j is disordered) until T_j leaves the sub-tree.

Note that the FR relationship propagates down the database, i.e., the tree. If in a sub-tree T_i FR T_j, then in any sub-tree of the sub-tree, either T_i FR T_j, or T_i and T_j have no FR relation. The latter is possible only if either T_i or T_j (or neither) does not access the sub-tree. This propagation makes it much less expensive to implement this relation in a database.

3. Locking Schemes

Here we assume that an ordered transaction typically places intention locks on non-leaf nodes and exclusive locks on the leaf nodes that it needs to access. A disordered transaction can however place an exclusive lock on the root node of a sub-tree if it needs to access all the leaves of that sub-tree. We believe that extensions to our scheme to handle shared locks are straightforward.

The disordered transactions observe the locking rules specified in [Korth 82a] and [Korth 82b]:

a) Each disordered transaction always halts in finite time if run serially;
b) Each disordered transaction locks the nodes in the root-to-leaf order;
c) Each disordered transaction unlocks the nodes in the leaf-to-root order;
d) Each disordered transaction locks a non-root node n only if it holds a lock on the edge (m, n) where m is the parent of n;
e) Each disordered transaction requests locks on a node n and its out-going-edges (n, p_1), ..., (n, p_k) simultaneously, where p_1, ..., p_k are the children of n and are needed by the disordered transaction.

For ordered transactions, the locking rules are as follows:

a) Each ordered transaction always halts in a finite time if run serially;
b) Until it becomes active, each ordered transaction intention locks the root of sub-tree and locks the out-going-edge simultaneously in a top-down manner;
c) As soon as an ordered transaction becomes active in a sub-tree, it does not lock edges any more. It locks the needed nodes of the sub-tree in the depth first order. An ordered transaction will always intention lock an internal node, never exclusive lock it.

In addition, both classes of transactions will observe the rule specified by FR relationship, that is (as stated in the previous section): in a sub-tree, if T_1 FR T_2, then T_1 is not allowed to lock a leaf with an identity larger than or equal to that of $DL(T_2)$ ($SL(T_2)$ if T_2 is disordered) until T_2 leaves the sub-tree.

Example 2 This example shows that the scheme developed in [Korth 82a] and [Korth 82b] restricts the potential concurrency of transactions. The database consists of a two level tree of four leaves (Figure 2).

Assume that T_1 wants to use data nodes d_3 and d_4 and that T_2 and T_3 want to use data nodes d_1, d_2, d_3 and d_4. T_1 comes into the system (intention locks root O) before T_2. T_2 comes before T_3. Here is one scenario:

t_1: T_1 intention locks root O and edges (O, d_3) and (O, d_4).
t_2: T_1 locks d_3, releases edge (O, d_3).
t_3: T_2 tries to intention lock root O and edges (O, d_1), (O, d_2), (O, d_3), and (O, d_4). But fails, because (O, d_4) is held by T_1. T_2 must wait until the lock on edge (O, d_4) is released.
t_4: T_1 locks d_4, releases edge (O, d_4).
t_5: T_2 intention locks root O and all out-going-edges (O, d_1), (O, d_2), (O, d_3), and (O, d_4), and so on.

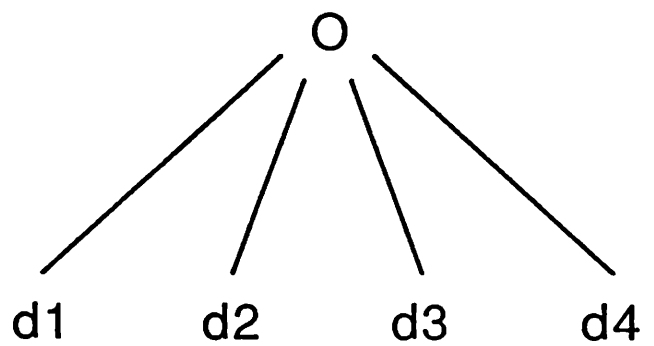

Figure 2: An Example Database

Note that until T_1 has released edge (O, d_4), T_2 cannot proceed, even if T_2 first needs to access nodes d_1 and d_2 which are not being used by T_1. Similarly, T_3 cannot come in until T_2 has locked all the data nodes it needs and releases all the edges, even if T_3 is designed to follow T_2 to lock all the data nodes.

Example 3 We now show how our scheme exploits the available concurrency in a set of transactions.

The same transactions working on the same database as in the previous example. In addition, we assume that T_2 and T_3 are ordered transactions. Our locking scheme will allow T_2 to intention lock root O and then lock data nodes d_1 and d_2 immediately after T_2 comes into the system. T_3 is allowed to lock each data item, d_1 to d_4, as soon as T_2 releases it. Thus, the addition of the notion of ordered transactions and locking rules appropriate for them allows us to execute more transactions in parallel than the scheme proposed in [Korth 82a] and [Korth 82b].

4. Proofs of Deadlock Freedom and Serializability

In this section, we shall prove that the locking scheme proposed in the last section ensures deadlock freedom and serializability. We first define the dependency relation and the wait-for relation and show that they are equivalent if a locking scheme can guarantee serializability and deadlock freedom at the same time. Then we prove that the locking scheme

proposed in Section 3 ensures deadlock freedom. Since the dependency relation and wait-for relation are equivalent in our locking scheme, this implies that our locking scheme also ensures serializability.

4.1. Dependency and Wait-for Relations

Let NODES(T) be the set of nodes accessed by T. If the intersection of NODES(T_i) and NODES(T_j) is not empty and for each node in the intersection, T_j accesses the node only after T_i has accessed it for the last time, then we say that T_j *depends on* T_i. This is denoted by $T_i > T_j$ or $T_j < T_i$.

If it is possible that a transaction T_i is waiting to access a node locked by another transaction T_j, we say that T_i *possibly waits* for T_j, and denote it by T_i PW T_j.

Both the dependency and wait-for relations are fundamental to consistency theory and deadlock theory respectively. Both theories are used for the synchronization of transactions and determine how resources are allocated to transactions. The two theories are closely tied as indicated by the following theorem.

Theorem 1 If a locking scheme ensures serializability and deadlock freedom, then the dependency relation and the wait-for relation are equivalent, i.e., for all i and j, $T_i < T_j$ if and only if T_i PW T_j.

Proof If a locking scheme ensures serializability and deadlock freedom, and $T_i < T_j$, then T_i PW T_j since T_i depends on the changes made to the data by T_j and hence may have to wait until T_j finishes using the shared resources.

On the other hand, if T_i PW T_j, because of the ensured serializability and deadlock freedom, T_j has to acquire all the resources it shares with T_i before T_i. This implies that $T_i < T_j$. □

The converse of the theorem does not always hold. In fact, we have the following theorem.

Theorem 2 If under a locking scheme, dependency relation and wait-for relation are equivalent, then the locking scheme ensures either both serializability and deadlock freedom, or neither of them.

Proof Serializability requires an acyclic dependency relation; and deadlock freedom requires an acyclic wait-for relation. If both relations are equivalent, then either both are acyclic, or neither one is. □

Theorems 1 and 2 are establish the relationship between consistency theory and deadlock theory. An immediate implication is that if under a locking scheme, then two relations are proved to be equivalent and one of the serializability and deadlock freedom is ensured, then another is automatically guaranteed. This will simplify the proof of correctness of our locking scheme.

4.2. Deadlock Freedom

Under our locking scheme, the following lemmas can be easily proved. The proofs are not provided here due to the space limit.

Lemma 1 In a sub-tree, if T_i PW T_j and at least one of T_i and T_j is an ordered transaction, then

T_i FR T_j.

Lemma 2 If T_i and T_j are disordered transactions, and T_i acquires some nodes in the intersection of NODES(T_i) and NODES(T_j) before T_j, then T_i acquires the root before T_j.

This lemma is actually Lemma 4.4 of [Korth 82b].

Lemma 3 If T_i FR T_j, then T_i arrives at every sub-tree, in which T_i and T_j share leaves, later than T_j.

Lemma 4 If there is a waiting chain in a sub-tree

T_1 PW T_2 PW T_3 PW ..... PW T_n,

then T_1 arrives at the sub-tree later than T_n.

Lemma 4 implies that the wait-for relation under our locking scheme is acyclic, hence

Theorem 3 Our locking scheme is deadlock free.

4.3. Serializability

Lemma 5 With our locking scheme, for any two transactions T_i and T_j, $T_i < T_j$ if and only if T_i PW T_j.

Proof The proof for the "only if" part is trivial. We just prove the "if" part.

First T_i PW T_j means that the intersection of NODES(T_i) and NODES(T_j) is not empty. Assume that T_i PW T_j, and let n be the node locked by T_j which T_i is waiting for.

If both T_i and T_j are disordered, by Lemma 2, T_j must acquire the root of the whole tree before T_i. Because of this, for any other node in the intersection of NODES(T_i) and NODES(T_j), T_j must be able to acquire it before T_i. Therefore $T_i < T_j$.

If at least one of T_i and T_j is ordered, by Lemma 1, the fact that T_i PW T_j means that T_i FR T_j. From our locking scheme, because T_i can never lock beyond T_j's lower bound, it is guaranteed that any node in the intersection of NODES(T_i) and NODES(T_j) is acquired by T_j first. []

Theorem 4 Our locking scheme ensures serializability.

Proof Lemma 5 establishes equivalence between the dependency relation and the wait-for relation under our locking scheme. This theorem, then, follows from Theorems 2 and 3. []

5. Conclusions

In this paper, we have presented a locking scheme for ordered and disordered transactions. In the development of the locking scheme, we have considered the possible locking behaviours of the ordered transactions as well as the underlying structures of the database system. Our scheme can achieve higher concurrency than the two phase locking scheme [Eswaran *et al.* 76] and the non-two-phase locking schemes proposed in [Korth 82a] and [Korth 82b]. In fact, these locking schemes require a transaction such as our ordered transaction to lock all edges from a node to all the children nodes accessed by the transaction even if it is not going to access them immediately. Our scheme, on the other hand, requires locking only the nodes

immediately needed by an ordered transaction. Our scheme achieves this while being serializable and deadlock free.

This paper describes work currently in progress. There are many possible extensions. For example, we have assumed only exclusive access mode to data nodes. It should not be too difficult to extend the scheme to handle more than one access mode. Also it should be possible to extend our scheme to deal with databases which have general acyclic graph structures as do in [Gray et al. 75], [Korth 82a], and [Korth 82b]. One of the issues we are currently addressing is the implementation of this algorithm in a tree structured distributed database system. Efficient implementation of the FR relation is an important consideration here.

In this paper, we have also proved the equivalence between the dependency and wait-for relations. We have showed how this equivalence simplifies the proof of serializability for our locking scheme. More importantly, it reveals why locking schemes such as ours and those of [Silberschatz and Kedem 80], [Silberschatz and Kedem 82], [Korth 82a], and [Korth 82b] can guarantee serializability while ensuring deadlock freedom, and vice versa. A common characteristic inherent in these locking schemes is that the transactions are strictly "ordered"[1], therefore, it is impossible for a transaction to "jump ahead" to another transaction if they share some resources and the latter has acquired some. As far as we know, our result is the first to formally establish a relationship between serializability and deadlock freedom. The establishment of the relationship between the two theories may prove helpful in applying results from one area to another, or in developing a uniform theory for both.

Acknowledgment

The authors thank the referees for their helpful comments.

6. References

[Badrinath and Ramamritham 88] Badrinath, B. R. and K. Ramamritham, "Synchronizing Transactions on Objects", *IEEE Transactions on Computers*, Vol. 37, No. 5, May 1988.

[Eswaran *et al.* 76] Eswaran, K. P., J. N., Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *CACM*, Vol. 19, Number 11, November 1976.

[Gray *et al.* 75] Gray, J. N., R. A. Lorie, G. R. Putzolu, I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database", *IBM Research Report 1654 (24264)*, September 1975.

[Gray 78] Gray, J. N., "Notes on Database Operating Systems", in *Operating Systems: An Advanced Course* R. Bayer, R. M. Graham, and G. Segmuller, (eds), Lecture Notes in Computer Science, vol 60, Spring-Verlag, Berlin and New York, 1978.

[Kedem and Silberschatz 83] Kedem, Z. I., A. Silberschatz, "Locking Protocol: from Exclusive to Shared Locks", *Journal of ACM*, Vol. 30, No. 4, October 1983.

[Korth 82a] Korth, H. T., "Edge Locks and Deadlock Avoidance in Distributed Systems", *ACM SIGACT-SIGOPS Symp. on Principle of Distributed Computing*, August 1982.

[Korth 82b] Korth, H. T., "Deadlock Freedom Using Edge Locks", *ACM Transactions on Database Systems*, Vol 7, No. 4, December 1982.

---

[1] This is the order among the transactions, not the order in which a transaction locks resources as do ordered transactions in our scheme.

[Korth 83] Korth, H. T., "Locking Primitives in a Database System", *Journal of ACM*, Vol. 30, No. 1, January 1983.

[Leu and Bhaargaava 87] Leu, P. and B. Bhargava, "Multidimensional Timestamp Protocols for Control", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 12, December 1987.

[Mohan *et al.* 85] Mohan, C., D. Fussell, Z. I. Kedem, A. Silberschatz, "Locking Conversion in Non-Two-Phase Locking Protocols", *IEEE Transactions on Software Engineering*, Vol SE-11, No. 1, January 1985.

[Silberschatz and Kedem 80] Silberschatz, A., Z. Kedem, "Consistency in Hierarchical Database Systems", *Journal of ACM*, Vol. 27, No. 1, January 1980.

[Silberschatz and Kedem 82] Silberschatz, A., Z. Kedem, "A Family of Locking Protocols for database Systems that are Modeled by Directed Graphs", *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 11, November 1982.

[Yannakakis 84] Yannakakis, M., "Serializability by Locking", *Journal of ACM*, Vol. 31, No. 2, April 84.