IMPROVEMENTS IN DATABASE CONCURRENCY CONTROL WITH LOCKING

by

Albert Croker

Information Systems Department
Graduate School of Business Administration
New York University
90 Trinity Place
New York, N.Y. 10006

November 1986

Center for Research on Information Systems
Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #134
GBA #86-99

ABSTRACT

Various techniques have been proposed to ensure the safe, concurrent execution of a set of database transactions. Locking protocols are the most prominent and widely used of these techniques, with two-phase locking and tree-locking being but two examples of these protocols. A locking protocol defines a general set of restrictions on the placement of lock and unlock steps within transactions. In this paper we show that it is possible to further increase the potential level of concurrency of a set of transactions, within the context of a specific locking protocol, by further restricting the placement of lock and unlock steps within each transaction. We also discuss a variation of the tree-locking protocol that allows transaction to be locked with respect to a dynamically changing set of tree structures. In addition we define and discuss the concept of a concurrency cost function for a locked transaction. This cost function measures the potential for conflict of a transaction with other transactions.

1. Introduction

A *locking protocol* is a set of rules governing the placement of lock and unlock steps among the access steps of a transaction. Various locking protocols have been proposed [BG,EGLT,G,KS1,KS2,SK1,SK2,Y] to ensure the correct concurrent execution of a set of database transactions. These protocols ensure correctness by allowing transactions, through the use of the lock and unlock primitives to control the access to database objects by other transaction. The locking of a database object by one transaction will cause any other transaction that attempts to lock that object to be delayed until the first transaction unlocks the object. (Throughout this paper we will assume that a lock step provides the locking transaction with exclusive access to the object referenced by that step.)

Lock steps are useful for ensuring serializability because of their ability to block transactions that attempt to lock and access a currently locked database object. However this ability to block the execution of transaction can limit the level of concurrent execution of a set transactions in database system. The longer a transaction keeps a database object locked, the greater is the likelihood that it will conflict with another transaction that will attempt to lock and possibly access the same object. This conflict will lead to the second transaction becoming blocked.

In this paper we define techniques that can be used in conjunction with previously defined locking protocols (the *two-phase locking protocol* (2PL) of Eswaran et al. [EGLT], and the *tree locking protocol* (TL) of Silberschatz and Kedem [SK1] to increase the overall level of concurrency in database transacton systems. In addition, we define a transaction metric that provides a relative measure of the length of time that a transaction maintains locks on database objects.

In Section 2 we provide a set of definitions and notation that we will make use of. In Section 3 we define a *concurrency conflict potential*, a transaction metric that reflects the likelihood that a given transaction will conflict with other transactions. In Sections 4 and 5 we discuss techniques for increasing the level of concurrency in the context of

two-phase locking and tree locking, respectively. These techniques involve the manipulation of lock, unlock, and access steps. Additionally, in Section 5 we discuss a variation of the tree locking protocol that allows trees to be defined dynamically based on a currently executing set of transactions. We conclude in Section 6.

2. Definitions

We define a *transaction* **t** to be a linear sequence of access steps

$$\mathbf{t} = \mathbf{a_1.x_1}, \mathbf{a_2.x_2}, \mathbf{a_3.x_3}, \ldots, \mathbf{a_n.x_n}$$

where each step $\mathbf{a_i.x_i}$ represents a *read* (**r**) or a *write* (**w**) of a single database object **x**. (Two different steps may access the same database object.) A *locked transaction* **t** is a transaction, among whose steps are interspersed a sequence of *lock* and *unlock* steps. That is,

$$\mathbf{t} = \mathbf{s_1.x_1}, \mathbf{s_2.x_2}, \mathbf{s_3.x_3}, \ldots, \mathbf{s_n.x_n}$$

is a locked transaction where **s** represents an access (**r** or **w**), lock (**l**) or unlock (**u**) step. We refer to a locked transaction **t** as a *locked version* of its underlying transaction.

A *schedule* **S** of a set of (locked) transactions **T** is an interleaving of the steps of those (locked) transactions, and restricted so that no step belonging to one transaction and referencing a database object **x** can be placed between a lock step and its *corresponding* unlock step of a second transaction. (An unlock step corresponds to a lock step if they are contained in the same locked transaction, and if the unlock step is the next following unlock step referencing the same database object as the lock step.) That is, in no schedule involving the two transactions

$$\mathbf{t_1} = \ldots, \mathbf{l_{1,i}.x}, \ldots, \mathbf{u_{1,j}.x}, \ldots$$

and

$t_2 = \ldots, r_{2,k}.x, \ldots$

can the step $r_{2,k}.x$ be placed between $l_{1,i}.x$ and $u_{1,j}.x$. (We assume that t_1 contains no other lock or unlock steps that also reference **x** between these two steps.)

As is typical of locking protocols, the two-phase locking and tree locking protocols can be defined in terms of the set of restrictions that each imposes on the placement of lock and unlock steps with respect to each other and the access steps of a locked transaction. Each set of restrictions guarantees that any schedule definable over a set of locked transactions, each of which satisfies the restrictions, is *serializable* (i.e., equivalent to a serial schedule). We use serializability as the criterion for schedule correctness.

The restrictions imposed by the two-phase locking protocol are:

1. each access step **a.x** must be preceded by the lock step **l.x**, and followed by the unlock step **u.x**, and

2. no lock step may follow an unlock step.

By these two restrictions, all two-phase locked transactions are characterized by an initial sequence of lock and access steps that is followed by a sequence access and unlock steps. The first sequence of steps is called the *growing phase* of the transaction, and the second sequence is called the *shrinking phase*. The *phase-shift point* of the transaction separates the the growing phase from the shrinking phase.

The tree locking protocol assumes that the set of database objects that are to be accessed by a transaction are organized hierarchically. The restrictions on the placement of lock and unlock steps within a transaction are defined in terms of this hierarchical organization. The restrictions, with respect to a given database hierarchy, imposed by the tree locking protocol are:

1. each access step **a.x** must be preceded by the lock step **l.x**, and followed by the unlock step **u.x**,

2. with the exception of the first lock step, each lock step **l.y** must be preceded by the lock step **l.x**, and followed by the unlock step **u.x**, where the

database object **x** is the parent of **y** in the database hierarchy, and

3. an unlock step **u.x** may occur no more than once.

3. Transaction Cost

In this section we define a function that provides a measure of the overall length of time that a transaction maintains locks on the database objects in its lock set. We assume that a transaction may execute concurrently with any arbitrary set of other transactions. Because of this assumption we define this function in terms of only the transaction to which it is applied, and thus it provides only a "relative" measure of the length of time that a (locked) transaction maintains locks on database objects.

By their design locking protocols such as 2PL and TL restrict the set of schedules that would otherwise be definable over a set of transactions. Any schedule is definable over a set of transactions since without locks there can be no blocking. The lock primitive allows a locked transaction to block another transaction that attempts to lock a currently locked database object. This blocking action lasts from the time the successful lock is executed until the transaction holding the lock releases it with an unlock step.

The longer a transaction holds a lock on a database object the greater the probability that it will conflict with (block) other transactions that also attempt to lock that object; in turn the greater the level of conflict between transaction the lower will be the level of concurrency of any resulting schedule.

When a transaction locks a database object that object becomes unavailable to other transactions until it is later unlocked. Under a static analysis of the transaction, the duration of a lock will depend on the number and duration of the steps that occur between that lock and its corresponding unlock. For example, in the transaction

$$\mathbf{t} = \ldots, \mathbf{l.x}, \mathbf{a_i.x_i}, \mathbf{a_{i+1}.x_{i+1}}, \ldots, \mathbf{a_{i+n}.x_{i+n}}, \mathbf{u.x}, \ldots$$

there are n+1 steps $\mathbf{a_j}$ between the lock step **l.x** and its corresponding unlock step **u.x**.

In our model of a locked transaction, each step $\mathbf{s_j}$ can be one of four types: read, write, lock or unlock. In general the duration of a step can depend on many factors: the type of step, the types of storage devices being used, whether the referenced database object is currently locked by another transaction, etc. However, because we seek a metric that is defined only in terms of the transaction to which it is applied, we consider only the type of the step.

Read and write steps are similar types of operations in that they both cause a transfer of data between a database on some secondary storage medium and a transaction's workspace. (They differ in the direction of the data flow.) Given this similarity, we assume that the duration of each of these two types of steps to be the same.

The execution of a lock or unlock step generally requires some communication between a locked transaction and the lock manager of a database management system. In carrying out the granting or releasing of locks, the lock manager requires little or no time consuming transfer of data between primary and secondary memory. Thus, in comparison to that of a read or write step, the duration of their execution is negligible. We assume the duration of the execution of a lock step to be 0 time units, and that of read and write steps to be 1 time unit.

Consistent with the above assumptions We define the *duration of the lock step* **l.x** in the locked transaction **t** (denoted $\lambda(\mathbf{t,x})$) to be the number of access steps occurring between **l.x** and its corresponding unlock step. The *concurrency conflict potential* of a (locked) transaction **t**, and denoted $\mathbf{C(t)}$, is defined as

$$\mathbf{C(t)} = \Sigma_{\mathbf{l.x} \,\epsilon\, \mathbf{t}}\, \lambda(\mathbf{t,x}).$$

Although it is based on several simplifying assumptions, the concurrency conflict potential is adequate for our needs. First, it is defined solely in terms of the transaction

to which it is applied. Second, it allows two different transaction to be compared to each other in terms of the relative expected duration of the length of time that they maintain locks.

The concurrency conflict potential of any transaction is zero since by our definition a transaction contains no lock or unlock steps. The *concurrency conflict potential with respect to a locking protocol* **P** of a transaction **t** is the minimum concurrency conflict potential associated with any **P**-*locked* version of **t**.

In the next two sections we will use the concurrency conflict potential function as the basis for defining optimal placements of lock and unlock steps, and the reordering of access steps within locked transactions.

4. Optimal Two Phase Locking

Typically, there are many two-phase locked versions of a given transaction. Each of these versions are characterized by a unique placement of lock and unlock steps. In this section we define an algorithm that when applied to a transaction produces an *optimally two-phase locked version* of that transaction.

Let transaction **t** be defined as

$$\mathbf{t} = \mathbf{a_1.x_1, a_2.x_2, a_3.x_3, \ldots, a_n.x_n}$$

(To simplify our presentation we will assume here that each database object $\mathbf{x_i}$ is unique.) The locked transaction $\mathbf{t_1}$ (shown below) represents one possible placement of lock and unlock steps allowed under the two-phase locking protocol.

$$\mathbf{t_1 = l.x_1, l.x_2, \ldots, l.x_n, a_1.x_1, a_2.x_2, \ldots, a_n.x_n, u.x_1, u.x_2, \ldots, u.x_n}$$

(This locked transaction could result if it was required that all lock steps precede all access steps and all locks are released (unlocked) when the locked transaction terminates. The concurrency conflict potential of this locked transaction is

$$C(t_1) = \Sigma_{l.x_i \in t_1} \lambda(t_1, x_i) = n^2$$

Shifting each lock step **l.x** rightward so that it immediately precedes the first access step **a.x**, so that an object is not locked until it is to be accessed, results in the locked transaction

$$t_2 = l.x_1, a_1.x_1, l.x_2, a_2.x_2, \ldots, l.x_n, a_n.x_n, u.x_1, u.x_2, \ldots, u.x_n$$

Applying the cost function to this transaction we get

$$C(t_2) = \Sigma_{l.x_i \in t_2} \lambda(t_2, x_i) = (n^2 + n) / 2$$

For $n = 5$, $C(t_1) = 25$ and $C(t_2) = 15$; for $n = 10$, $C(t_1) = 100$ and $C(t_2) = 55$. (As n increases, the ratio of $C(t_2)$ to $C(t_1)$ approaches 2.) The locked transactions t_1 and t_2 demonstrate that the placement of lock (and unlock) steps can have a significant impact on the concurrency conflict potential of a transaction.

Given the restrictions on the placement of locks by the two-phase locking protocol, it is not possible to further reduce the concurrency potential associated with t_2 by shifting lock steps further to the right. However it can be reduced by shifting both lock and unlock steps leftward as we show with the following locked transaction.

$$t_3 = l.x_1, a_1.x_1, \ldots, l.x_{n/2}, l.x_{n/2+1}, l.x_{n/2+2}, \ldots, l.x_n, u.x_1, u.x_2, \ldots, u.x_{n/2}, a_{n/2+1}.x_{n/2+1}, u.x_{n/2+1}, \ldots, a_n.x_n, u.x_n$$

The value of the cost function applied to this locked transaction is

$$C(t_3) = \Sigma_{l.x \in t_3} \lambda(t_3, x) = n^2/4 + n/2$$

For $n=5$, $C(t_3) = 8.75$; for $n=10$, $C(t_3) = 30$.

The phase-shift point occurs in a different position in the locked transactions t_1, t_2, and t_3. The phase shift point occurs after the access step $a_n.x_n$ in the locked

transactions t_1 and t_2, and between access steps $a_{n/2}.x_{n/2}$ and $a_{n/2+1}.x_{n/2+1}$ in the locked transaction t_3. In general the phase-shift point may be defined as occurring between any two transaction steps.

In the following lemma, we define a pattern for the placement of lock and unlock steps relative to a given position of the phase-shift point such that the resulting two-phase locked transaction has the lowest concurrency conflict potential of any other two-phase locked version of the same transaction with a similarly positioned phase-shift point. (Unlike our earlier example, this lemma permits a transaction to have multiple steps that access the same database object.)

Lemma:

Let **t'** be a two-phase locked transaction that is derived from the transaction

$$t = a_1, a_2, \ldots, a_j, a_{j+1}, \ldots, a_n$$

and has the structure:

- For each database object **x**,
  - if a_i, $i \leq j$, is the first step accessing **x**, then a_i is immediately preceded by the step **l.x**
  - if a_k, $k \geq j+1$, is the last step accessing **x**, then a_k is immediately followed in **t'** by the unlock step **u.x**.
- All other lock and unlock steps--and thus the phase shift point--in **t'**, occur between the access steps a_j and a_{j+1} (with unlock steps following lock steps).

If **t''** is any other two-phase locked transaction that is derived from the transaction **t** and has a phase-shift point between the access steps a_j, and a_{j+1}, then

$$C(t') \leq C(t'')$$

Proof:

By the definition of the locked transaction **t'**, shifting a lock step, **l.x**, rightward (or an unlock step, **u.x**, leftward) over any access steps will either violate the two-phase locking protocol or change the position of the phase-shift point.

Shifting a lock step, **l.x**, leftward, or an unlock step **u.x** rightward in the locked transaction **t'** will increase the value of λ**(t', x)** by an amount equal to the number of access steps that were shifted over. Thus for each database object **x** accessed by **t'**, λ**(t', x)** is minimal with respect to the specified position of the phase shift point, and

$$\mathbf{C(t')} = (\Sigma_{\mathbf{l(x)}\epsilon\mathbf{t'}}\ \lambda\mathbf{(t', x)}) \leq \mathbf{C(t'')}.$$

The following algorithm transforms a transaction into an optimally two-phase locked version of that transaction.

Algorithm I.

input: transaction $t = a_1, a_2, \ldots, a_n$

output: optimally two-phase locked version of t

1. for each x in A(t), insert a lock of x, l(x), to the left of a_1

2. for each x in A(t), insert an unlock of x, u(x), immediately after the last step that accesses x.

   (The result of Steps 1 and 2 is a two-phase locked version of t where the phase-shift point precedes the access step a_1, and all of the lock steps are adjacent to (not separated by an access step from) the phase-shift point, and no unlock steps are adjacent to the phase-shift point.)

3. while the number of lock steps adjacent to the phase-shift point is greater than the number of unlock steps adjacent to the phase-shift point

   - shift the phase-shift point to the immediate right of the next access step

   - shift those lock steps which do not lock objects accessed by any step on the left of the phase-shift point rightward so that they are once again adjacent to the phase-shift point

   - shift those unlock steps which were previously adjacent to the phase-shift point rightward so that they once again become adjacent to and

follow the phase-shift point

Example 1:

Let transaction

t = r.a, w.b, r.c, r.d, w.c, w.d

be an input transaction to Algorithm I. Execution of the first two steps of the algorithm results in the two-phase locked transaction

t_1 = l.a, l.b, l.c, l.d, | r.a, u.a, w.b, u.b, r.c, r.d, w.c, u.c, w.d, u.d

with the indicated phase-shift point.

Executing Step 3 of the algorithm results in the sequence of two-phase locked transactions:

t_2 = l.a, r.a, l.b, l.c, l.d, | u.a, w.b, u.b, r.c, r.c, w.c, u.c, w.d, u.d

t_3 = l.a, r.a, l.b, w.b, l.c, l.d, | u.a, u.b, r.c, r.d, w.c, u.c, w.d, u.d

At this point Algorithm I terminates and outputs the two-phase locked transaction $\mathbf{t_3}$. The concurrency cost associated with this locked transaction is

$$\mathbf{C}(t_3) = 2 + 1 + 3 + 4 = 10$$

Theorem 1.

If **t'** is the locked transaction that results from applying Algorithm I to transaction **t**, then **t'** is an optimally two-phase locked version of t.

Proof:

After the initial insertion of lock and unlock steps by Steps 1 and 2 of the algorithm, each iteration of Step 3 causes the phase-shift point to be shifted rightward over one access step. Let $\mathbf{t_1}$ be an intermediate locked transaction that exists before an iteration of Step 3 with $\mathbf{C}(t_1) = c$, and $\mathbf{t_2}$ be the locked transaction that results after one

additional iteration. Also let **m** and **n** be the number of lock and unlock steps respectively, that are adjacent to the phase-shift point. One iteration of Step 3 will cause either **m** or **m-1** lock steps to be shifted rightward over one access step and all **n** unlock steps to be shifted over the same access step. Thus the concurrency conflict potential of the locked transaction t_2 is

$$C(t_2) = c + n - m \text{ or } C(t_2) = c + n - m + 1$$

With each iteration of Step 3 the number of lock steps that are adjacent to the phase-shift point monotonically decreases, while the number of unlock steps that are adjacent to the phase-shift point monotomically increases. By Lemma 1, each iteration of Step 3 results in a two-phase locked version of the input transaction having a minimum concurrency cost for the resulting position of its phase-shift point. Since **n**, the number of lock steps adjacent to the phase-shift point is initially equal to zero, each iteration of Step 3 causes the concurrency cost associated with the resulting locked transaction to monotonically decrease while $m > n$, and then to increase. Since the iteration of Step 3 stops when **m** becomes less than or equal to **n**, the final locked transaction **t'** that is output from this step has the minimum concurrency cost of any two-phase locked version of the input transaction **t**.

4.1. Transaction Transformation with Two Phase Locking

In the preceding we have defined rules for the placement of lock (and unlock) steps in a transaction so that the resulting locked transaction is an optimally two-phase locked version of the original transaction. Since the minimum set of database objects that must be locked by a two-phase locked transaction **t** is the access set of **t**, **A(t)**, these rules can be viewed as specifying how this "well-defined" set of lock and unlock steps should be arranged within a transaction. Consistent with this view we now look at how the access (read and write) steps of a transaction can be manipulated to further decrease the concurrency conflict potential of the resulting transaction.

Manipulating the lock steps of a locked transaction does not affect the semantics of

(i.e., what is computed by) the underlying transaction. Rearranging the read and write steps of the underlying transaction results in a syntactically "different" transaction. However, if this new transaction can be guaranteed to have the same semantics as the original, and at the same time have a smaller concurrency conflict potential with respect to 2PL, then it would be preferable to the original.

In our model, a transaction

$$\mathbf{t} = a_1.x_1, a_2.x_2, \ldots, a_n.x_n \text{ where } a_i \in \{\mathbf{r}, \mathbf{w}\}$$

is a finite sequence of read and write steps. The interpretation of a read step r.x in a (locked) transaction **t** is that the current value of **x** is retrieved for manipulation by **t**. We assume that the value read is not dependent on any of the preceding steps in **t**.

The execution of a write step **w.x** by the (locked) transaction **t** results in a new value for **x**. Unlike a read step, the value written may be dependent on previous steps in transaction t. In particular, the value written by step **w.x** in **t** may be defined as a function of some or all of the values read by previous steps of **t**. (We assume that each data item accessed by a transaction is not read or written more than once and a read to a data item must precede any write to that data item.) In the absence of additional semantic information in our model we must assume that the value written by each write step of a transaction is dependent on each of the values read by preceding steps.

The implication of the above analysis is that in any transaction a read step can be shifted to the left (performed earlier) in the transaction without affecting what the transaction computes. A read step can be shifted to the right as long as it continues to precede each of the write steps that it originally preceded. Similarly a write step may be shifted to the right without affecting the value written, and shifted to the left as long as it continues to follow each of the read steps that it originally followed.

Although in our model it is not possible to eliminate read and write steps without affecting the semantics of a transaction, it is often possible and desirable to reorder

transaction steps. The reordering of transaction steps is possible when, the order, with respect to each other, of each pair of read and write steps is preserved, and thus the resulting transaction can be presumed to have the same meaning. A particular reordering is desirable if it has a smaller concurrency conflict potential with respect to 2PL.

We have defined an optimal placement of lock and unlock steps, and thus an optimal positioning of the phase-shift point, in a two-phase locked transaction. From this definition it follows that the closer the phase-shift point can be brought to the beginning of a transaction, the sooner the unlocking of database objects can begin.

Analysing Algorithm I we see that the optimal position of the phase-shift point occurs between the two transaction steps where the number of objects that have been accessed for the last time by the transaction is most equal to the number of objects that have not as yet been accessed. Thus the sooner a transaction can make a set of final accesses to objects in its access set, the sooner can occur the phase-shift point.

Algorithm I determines the the optimal phase-shift point of a transaction by initially defining it at the beginning of the transaction, and then repeatedly shifting it leftward. This shifting continues for as long as the concurrency conflict potential associated with the resulting locked transactions decreases. The faster the number of locks and unlocks that are adjacent to the phase-shift point can be reduced and increased, respectively, the closer the phase-shift point will be to the beginning of the resulting optimally two-phase locked transaction. This observation, along with the earlier stated restrictions on the reordering of transaction steps suggests the following strategy for reordering a transaction's access steps.

1. Place at (shift forward to) the beginning of the transaction those steps that read data items that are not later written transaction.

   *the rationale for this step is that as soon as the phase-shift point is reached, each of the objects read by these steps can be unlocked. Additionally, shifting the phase-shift point rightward over this sequence of read steps will cause the number of lock steps that are adjacent to it

to decrease by one, while the number of unlock steps that are adjacent to it will increase by one. (Also, this strategy may increase the likelihood that if a transaction has to be aborted it will have read from, but not written to, the database, and thus allow for an easier recovery.)

2. Next, follow the sequence **S** of read steps produced by Step 1 above with those write steps that in the original transaction were preceded by only those read steps in **S**

   *again shifting the phase-shift point over these steps will cause the number of lock steps adjacent to it to decrease while the number of unlock steps adjacent to it increases.

3. The remaining steps of the original transaction should be placed following the sequence of read and write steps resulting from Step 2 above. The ordering of these remaining steps should be the same as it was in the original transaction.

   *the rationale for Step 3 is that each read step that precedes a write step in the original transaction must also do so in any resulting transaction. (Two consecutive write steps may be placed in any order with respect to each other.)

The inclusion of additional semantic information in our model, for example, the specific read steps on which a write step is dependent, would provide additional flexibility for reordering transaction steps.

5. Optimal Tree Locking

The tree locking protocol is an example of a locking protocol that assumes that some structure is imposed over the set of database objects. This protocol assumes that the set of database objects is hierarchically structured. While this locking protocol will often require that a locked transaction lock database objects that will not later access, it also allows transactions to lock and unlock objects in a non two-phase manner. That is, unlock steps can follow lock steps in transactions.

In this section we define an algorithm that when applied to a transaction and a hierarchically structured set of database objects, results in an optimally tree locked

version of the transaction.

The hierarchical structure imposed over a set of database objects defines a partial ordering of those objects. The tree locking protocol requires that any tree locked transaction accessing these database objects lock them in an order consistent with this partial order. For this reason we refer to this hierarchy as a data access tree (DAT).

By definition, the tree locking protocol requires that a transaction lock a subtree of the DAT. This subtree is defined as the smallest subtree that contains the access set of the transaction. For a DAT Δ and a transaction t, we call the minimal subtree of Δ that must be locked by any tree-locked version of transaction t the *t-induced subtree* of Δ and denote it as Δ_t.

Given a database tree Δ, and a transaction **t**, the following algorithm will generate a tree locked version of **t** that is optimally tree locked with respect to Δ.

Algorithm II.

input: transaction $t = a_1, a_2, ..., a_n$, DAT Δ

output: optimally tree locked (with respect to Δ) version of t

1. for each $x \epsilon A(t)$
   - insert a lock of x, l.x, immediately before the first step, a_i, accessing x
   - insert an unlock of x, u.x, immediately after the last step, a_j, accessing x
2. for each x in Δ_t but not in A(t)
   - insert a lock of x, l.x, at the end of the transaction
   - insert an unlock of x, u.x, at the beginning of the transaction

   (let $S = s_1, s_2, ..., s_m$ be the resulting sequence)
3. for $s := s_1$ to s_m
   - if s is a lock step l.x and there exists a lock step l.y to the left of l.x in S, and the database object x is an ancestor of y in Δ, move l.x so that

it immediately precedes the leftmost such l.y

(let S' = $s'_1, s'_2, \ldots, s'_m$ be the resulting sequence)

4. for s := s'_m to s'_1

   - if s is the unlock step u.x and there exists a lock step l.y to the right of u.x in S and (x,y) is an edge in Δ_t, move u.x to the immediate right of the rightmost such step l.y.

Example 2

Let transaction

t = r.a, w.b, r.c, r.d, w.c, w.d

and the DAT shown in Figure 1 be the input to Algorithm 2.

Executing Steps 1 and 2 of Algorithm II results in the sequence

S = u.e, u.h, l.a, r.a, u.a, l.b, w.b, u.b, l.c, r.c, l.d, r.d, w.c, u.c, w.d, u.d, l.e, l.h

Step 3 shifts the lock steps l.d, l.e, l.h leftward, resulting in the sequence

S' = u.e, u.h, l.e, l.a, r.a, u.a, l.b, w.b, u.b, l.d, l.h, l.c, r.c, r.d, w.c, u.c, w.d, u.d

Finally, Step 4 shifts unlock steps u.e and u.h rightward resulting in the tree locked with respect to Δ transaction

t' = l.e, l.a, r.a, u.a, l.b, w.b, u.b, l.d, u.e, l.h, l.c, u.h, r.c, r.d, w.c, u.c, w.d, u.d

The concurrency cost associated with t' is **C(t') = 11.**

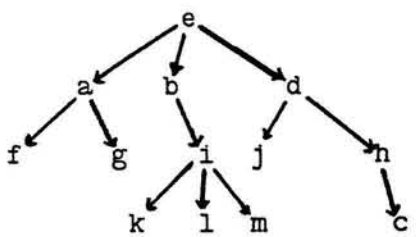

Figure 1

Definition: A lock step, l.x, in a tree locked with respect to Δ transaction t is *rightmost* if for the next access step, a(y), either y $\doteq$ x (the lock and access steps reference the same database object) or y is a descendent of x in the DAT Δ.

Similarly,

Definition: An unlock step, u(x), in a tree locked with respect to Δ transaction t is *leftmost-R* if it is immediately preceded by either the access step a(x) or a rightmost lock step l(y), where y is an immediate descendent (child) of x. (The "*-R*" is used to emphasize the lack of symmetry between this and the previous definition.)

Theorem 2

Let locked transaction **t** be tree-locked with respect to Δ. If each lock step in **t** is rightmost, and each unlock step is leftmost-R, then **t** is optimally tree-locked with respect to Δ.

Theorem 3

If **t'** is the locked transaction that results from applying Algorithm 2 to a transaction t and the DAT Δ, then **t'** is optimally tree-locked with respect to Δ.

5.1. Transaction Transformation with Optimal Tree Locking

In this section we have defined an optimal tree-locking strategy for transactions. Next we show that it is possible to reorder the steps of some transactions so that the resulting transaction has a lower concurrency conflict potential (with respect to tree-locking with respect to a specified database access tree) than the original transaction.

The optimal ordering of steps in a transaction that is to be tree-locked is dependent on the particular database tree with respect to which the transaction is to be tree-locked. Let Δ be a database access tree. The order in which objects are locked by a locked transaction **t** that is tree-locked with respect to Δ must be consistent with the partial order of database objects defined by the tree Δ.

How soon a database object can be unlocked by a transaction is restricted both by the requirements of the tree-locking protocol (except for the first object it locks, a tree-locked transaction can only lock a database object if it is currently holding a lock on the parent of that object), and the position of the last access step referencing that object in the transaction.

The restrictions on the placement of lock and unlock steps in a transaction suggest the following approach to the ordering of access steps in a transaction that is to be tree-locked with respect to a DAT Δ.

> To the extent possible, the order in which a transaction accesses database objects should be consistent with the partial order defined over them by the DAT Δ. It will generally not be possible to achieve complete consistency because, as with two-phase locking it is required that each read step that precedes a write step in the original transaction must continue to do so in the derived transaction.

Although we have suggested a reordering of access step we have not developed what we feel to be an efficient algorithm for performing the reordering of transaction steps.

5.2. Dynamic Database Trees

Even in the presence of optimally tree locked transactions two properties of the tree-locking protocol constrain the degree of concurrency obtainable. First, by definition, a tree locked transaction is required to lock a subtree of the database tree. Thus, a tree locked transaction generally must lock database objects that are not accessed. Second, a tree locked transaction, with the exception of its first lock, can lock a database object only if it is currently holding a lock on that objects's parent (with respect to the database tree). If the access set of a locked transaction is dispersed throughout the database tree, then the locked transaction will either have to maintain a lock on a database object near the root of the tree or maintain locks on much of its access set. In both cases these locks will have to be maintained for longer periods of time than would otherwise be necessary.

Together, these two properties contribute to an increased probability that locked

transactions, tree locked with respect to the same database tree, will have overlapping lock sets. The extent of this overlap will depend on the extent of the overlap of the access sets of the locked transaction, and how dispersed throughout the database tree each access set is. For example, if the locked transactions $\mathbf{t_1}$, accessing the database objects **a** and **g**, and $\mathbf{t_2}$, accessing the database objects **i** and **d**, are tree locked with respect to the database tree in Figure 1, then, in addition to the objects in their access sets, they will also have to lock the database objects **e** and **b**.

If the access set of each tree locked transaction could be localized within the database tree so as to minimize the difference between the locked transaction's lock set and access set, then the extent to which lock sets overlap would be attributable largely to the extent to which access sets overlap.

Locked transactions having no overlap of their access sets would be able to run concurrently without conflicting with each other. However, unlike the situation that exists with two-phase locking, deadlock would not be a possibility.

Unfortunately it is not always possible to structure a database tree so that the access set of each locked transaction is localized. Localizing the access set of one transaction may cause the access set of another transaction to be more dispersed. Additionally, in order to define the appropriate database tree, we would need to know in advance the set of transactions that are to access the database.

Since DATs are not dependent on the logical organization of a database (DATs may be defined over relational and network databases), different DATs can be defined over the same set of database objects. Thus, it is possible to define a DAT for each transaction such that its access set is localized within that database tree. However, if transactions are tree locked with respect to different database trees, a non-serializable or deadlocked schedule may result.

The following variation of tree locking allows database trees to be customized to a currently executing set of transactions.

Definition: *The Dynamic Tree Locking Protocol* [C,CM]

1. When a transaction t is to be executed, its access set, A(t), is determined.

2. All currently existing DATs defined over database objects in A(t) are joined together (by defining additional edges) to form a single database tree Δ.

3. Each of the database objects in A(t) that are not currently in Δ are added to Δ. (The tree structure of Δ must be maintained.)

4. Transaction t is then tree locked with respects to Δ resulting in the locked transaction t'. (Algorithm II can be used for this purpose.)

5. Locked transaction t' is then allowed to execute.

(We assume that transactions arrive one at a time to execute, and initially no database trees exist.)

The dynamic nature of this protocol is attributable to Steps 1 and 2. Unlike the tree locking protocol defined by Silberschatz and Kedem, the set of database objects to be locked and the order in which they will be locked by each locked transaction will depend on the set of locked transactions currently executing. Example 2 shows how an implementation of dynamic tree locking would work.

Example 2: Let the three database trees Δ_1, Δ_2, and Δ_3 shown in Figure 2.a exist at the time the transaction

t = r.d, r.b, w.a

enters the system to be executed. In Step 1 of the protocol, the access set of transaction t is determined to be

A(t) = {a, b, d}

Using A(t), Step 2 merges the database trees Δ_1 and Δ_2 to form the database tree Δ_4 shown in Figure 2.b. Since the database tree Δ_4 does not contain all of the objects accessed by t, it is extended into the database tree Δ_5 (Figure 2.c) by Step 3. Transaction t is then tree locked with respect to Δ_5 using Algorithm II. The locked transaction t' is one possible result of this tree locking.

t' = l.a, l.d, r.d, l.g, u.d, l.b, u.g, r.b, u.b, w.a, u.a

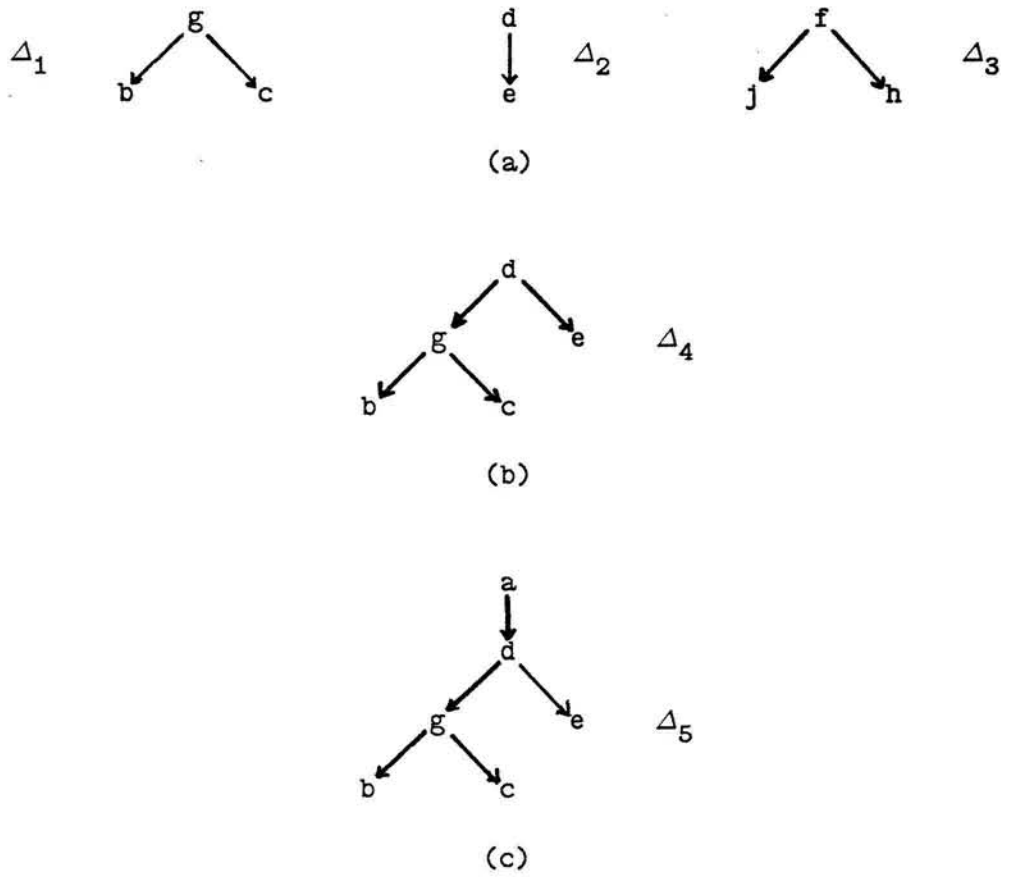

Figure 2

Theorem 4: If π is a schedule of a set of dynamically tree locked transactions **T**, then π is serializable and free from deadlock.

6. Conclusion

In this paper we have presented various techniques for reducing the length of time that locked transactions maintain locks on database objects. These techniques were presented for the tree locking and two-phase locking protocols. In order to provide a measure for this time, we have defined a concurrency conflict potential function.

This work represents an initial effort at determining the feasibility of developing an automated transaction compiler/processor based on the concept of minimizing the

concurrency conflict potential of the resulting locked transactions. The degree of improvement that we have been able to achieve was limited by the model that we used to represent a transaction. As a continuation of our efforts we are investigating more semantically descriptive transaction models in order to determine other improvements that may be made in the degree of concurrency that can be achieved for transaction processing in a database management system.

References

BG — Berstein, P.A., Goodman, N.
Fundamental Algorithms for Concurrency Control in Distributed Database Systems
Tech. Rep. CCA-80-05, Feb. 1980,
Computer Corporation of America, Cambridge, MA

C — Croker, A.
Increasing Database Concurrency through Locking
Ph.D. Dissertation, S.U.N.Y. at Stony Brook, 1984
Stony Brook, NY

CM — Croker, A., Maier, D.
A Dynamic Tree-Locking Protocol
Proceedings of IEEE Conference on Data Engineering
pp. 49-56, Los Angeles, CA, 1986

EGLT — Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.
The Notion of Consistency and Predicate Locks in a Database System
CACM, v19:11, pp. 624-633, 1976

G — Gray, J.N.
Notes on Database Operating Systems
IBM Research Lab. RP RJ2188, 1978
San Jose, CA

KS1 — Kedem, Z., Silberschatz, A.
Controlling Concurrency using Locking Protocols
Proc. 20th IEEE Symp. Foundations of Computer Science,
pp. 274-285, 1979

KS2 — Kedem, Z., Silberschatz, A.
Non-two-phase Locking Protocols with Shared and Exclusive Locks
Proceedings International Conference on VLDB, 1980

SK1 — Silberschatz, A., Kedem, Z.
Consistency in Hierarchical Database Systems
Journal of the ACM, v27:1, pp. 72-80, 1980

SK2 Silberschatz, A., Kedem, Z.
A Family of Locking Protocols for Database Systems that are Modelled by Directed Graphs
IEEE Transactions on Software Eng
SE-8, pp. 558-562, 1982

Y Yannakis, M.
A Theory of Safe Locking Policies in Database Systems
Journal of the ACM, v29.3, pp. 718-740, 1982