THE VIRTUES OF LOCKING BY SYMBOLIC NAMES

by

Ouri Wolfson

Technical Report #416
(Revised Version)

June 1986
THE VIRTUES OF LOCKING BY SYMBOLIC NAMES

Ouri Wolfson

The Technion - Israel Institute of Technology,
Computer Science Dept., Haifa 32000, Israel.

ABSTRACT

We propose an algorithm for improving the concurrency of two phase locked transaction systems, which use symbolic-name locking. The algorithm determines by preanalysis which entities can be unlocked before all locks have been obtained, without compromising serializability. This extends the work we published in [12], in three ways. First, the transactions are not restricted to exclusive locks, and may use shared locks as well. Second, a method is proposed to prevent the potential problem of cascading restarts, which results from unlocking of entities before commitment. Third, the transactions may be designed for a distributed database.
1. INTRODUCTION

Database management systems which control concurrency by locking usually adopt the two phase protocol ([5]). It requires that a transaction obtains all locks before issuing any unlock. The protocol is used whether locking is performed by symbolic names, available at compile time, or by values determined at run time. The difference between the two types of locking has generally been overlooked in the literature, and has simply been treated as a locking granularity issue. In this paper we present a method of using the additional information provided by symbolic-name locking to improve concurrency. Specifically, we propose an algorithm that analyzes a given set of two-phase-locked (2PL) transactions which are to be executed concurrently. The analysis is performed only once, after the transaction system is defined and before it is run on-line. The purpose is to reduce mutual interference, while ensuring that the locks still suffice to guarantee serializability. Reduced mutual interference is obtained by unlocking of some entities before all locks have been requested. The following example exhibits a sample input to our algorithm.

Example 1: Assume that $x, y, z$ are database entities (e.g., relations) and $SL, EL, U, R$ and $W$ represent shared-lock, exclusive-lock, unlock, read and write operations respectively. Then, $t_1$ and $t_2$ below are a possible input.

\[
\begin{align*}
& t_1 = ELx, Rx, SLy, Ry, Wx, ELz, Ux, Uy, Wz, Uz. \\
& t_2 = ELy, Ry, SLx, Rx, Wy, ELz, Wz, Ux, Uy, Uz.
\end{align*}
\]

Assuming that the transactions are known in advance, such representation is possible by examining the programs, provided that locking is done by symbolic-names available at compile time. Locking by symbolic names is common for large locking granularities, and has the advantage of low overhead in managing the locks, and preventing the "phantom" problem ([5]).

The algorithm we propose will determine that $t_1$ can unlock $y$ immediately after $Ry$, and $t_2$ can unlock $x$ immediately after $Rx$, and still ensure serializability of every possible interleaved execution. It is clear that there are interleavings of the transactions steps in which the modified $t_1$ and $t_2$ will allow better concurrency. (For example, when $t_2$ waits for $t_1$ on $y$.) Considering that $t_1$ and $t_2$ are run repeatedly, as is the case with predefined transaction systems, performance can improve significantly. Note that in our example, we do not even risk hav-
ing to postpone commitment of \( t_i \) until \( t_j \) commits, because early unlocking was performed on read-only entities (more about it in the section 5). Therefore, for many transaction systems there is absolutely no reason to postpone some unlocks until all locks have been obtained. Our algorithm provides a general scheme for identifying and exploiting these situations.

This paper involves two methods which have been studied, mostly separately, in the past: non-two-phase locking and preanalysis. We used them both in [12], however that work was restricted to exclusive locks in a centralized database, and did not address the problem of cascading restarts. Preanalysis was originally introduced in the SDD1 concurrency control mechanism ([2]), where it was combined with time-stamping. Recently, Lausen et al. ([9]) introduced an algorithm which preanalyzes transactions in a centralized database to establish lock-unlock positions. Their algorithm uses syntactic exclusive locks, unrelated to entities, and as a result, a large number of locks may have to be managed. Concurrency of the produced set of transactions may not be as good as allowed by 2PL, however the advantage compared to 2PL is that deadlock cannot occur.

Non-two-phase locking has been studied in the context of structured locking protocols, such as tree ([10]), dag ([8,11,13]), and hypergraph ([3,13]). These works assume that the database is logically or physically structured (e.g., as a tree), and exploit the structure to improve concurrency by advance unlocking. The drawback of the approach is that a transaction cannot reference an arbitrary subset of entities, and as a result, transactions may have to lock entities they do not read or write. (For example, in the tree protocol, transactions must "walk along" paths in the tree.) Preanalysis enables us to overcome this difficulty. Unlocks are advanced, but transactions still lock exactly the same entities they do in their 2PL format. The proposed algorithm builds a most general structure (i.e., a hypergraph), and uses a related result proved by Yannakakis in [13]. The result provides a sufficient condition for serializability assurance (which can be tested in polynomial time), and the algorithm advances unlocks while avoiding violation of the condition. Why not use a necessary and sufficient condition? In [13] it is proven that the problem of determining serializability assurance of a set of locked transactions is coNP-complete. Therefore, basing an efficient algorithm on such a test is probably impossible.

Advance unlocking may introduce the problem of cascading restarts. It arises when a transaction reads uncommitted data, i.e. data which had been written and unlocked by an uncommitted transaction. If the writing transaction aborts, then the reading transaction must do so as well. It may have, in turn, written uncommitted
data, etc. In other words, an aborting transaction can trigger a chain reaction of aborts. The hypergraph structure built enables the proposed algorithm to be adapted to eliminate this undesirable phenomenon, in case it is initiated by deadlock. Indeed, deadlock is the most common cause of cascading restarts (see [3,15] for arguments supporting this claim).

Before outlining the rest of the paper we would like to emphasize an important point. A large number of papers devising concurrency control algorithms for distributed databases have appeared in the literature. In most cases the main problem of the work is that convincing performance comparisons with other algorithms is not carried out. Therefore, it is often not clear when one should use the proposed algorithm, rather than others. By contrast, the algorithm presented here compares well with the most simple and popular concurrency control algorithm, namely 2PL. It cannot be worse, and in many (transaction system) cases it is better.

The rest of the paper is organized as follows. In section 2 we describe the model. In section 3 we provide preliminary definitions, and present the supporting condition proved in [13]. Section 4 describes the algorithm. In section 5 we address the problem of cascading restarts, and propose a solution. The supporting condition and the algorithm are extended to distributed databases in section 6. In section 7 we conclude and discuss the problem of ad hoc queries.

2. TRANSACTIONS AND TRANSACTION SYSTEMS

A transaction is a sequence of steps \( s_1, \ldots, s_n \). Each \( s_i \) is the read of an entity \( x \), denoted \( Rx \), or the write of \( x \), denoted \( Wx \), or the shared-mode lock of \( x \) (\( SLx \)), or the exclusive-mode lock of \( x \) (\( ELx \)), or the unlock of \( x \) (\( Ux \)). We denote by \( Lx \) the lock of \( x \) when the mode is irrelevant, or the reference is to both types of lock. Each entity is read at most once, and written at most once in a transaction. We assume that an entity is locked if and only if it is read or written by the transaction. For an entity \( x \), there is at most one lock in a transaction (no upgrading of shared locks or downgrading of exclusive locks will be allowed). \( Lx \) precedes the read-write steps of \( x \) and has a unique corresponding \( Ux \) which succeeds them. If \( x \) is not written by the transaction, then the lock may be shared; otherwise it must be exclusive. Two different locks on an entity conflict if at least one of them is exclusive.
A transaction system is a finite set of transactions. A schedule of a transaction system, \( \tau \), is an interleaving (or merging) of the steps of some transactions in \( \tau \) in which between every pair of conflicting locks of the same entity there exists an unlock of the entity. It represents a possible execution of the transactions in \( \tau \), which respects the locks issued by the transactions; namely a lock for an entity which is already locked by another transaction in a conflicting mode, is delayed until the entity is unlocked. A transaction of \( \tau \) may appear in a schedule more than once. A schedule of \( \tau \) is serializable if it is equivalent to a serial schedule ([5]). \( \tau \) ensures serializability if each one of its schedules is serializable [5].

3. A SUFFICIENT CONDITION FOR SERIALIZABILITY ASSURANCE

Our algorithm is based on a condition proved in [13] as sufficient for a transaction system to ensure serializability. In order to state the condition we need the following preliminaries. Given a transaction system \( \tau \), we define a corresponding hypergraph ([1]), \( H(\tau) \); its nodes are the entities referenced in \( \tau \), and its hyperedges are defined as follows. Assume that some transaction \( t \in \tau \) locks \( n \) entities, and that the first entity it locks (regardless of mode) is \( x_1 \), the second is \( x_2 \), etc. Then \( H \) has \( n \) hyperedges "corresponding" to \( t \)

\[
\{x_1\}, \{x_1,x_2\}, \{x_1,x_2,x_3\}, \ldots, \{x_1,\ldots,x_{n-1}\}, \{x_1,\ldots,x_n\}.
\]

Each hyperedge \( h \) is partitioned into two sets, \( h(e) \) and \( h(s) \), representing the entities locked by \( t \) in exclusive and shared mode, respectively. We will denote \( h \) as a set, whose members are separated by a vertical bar, \( ; \). The members of \( h(e) \) appear to the left of the vertical bar, and the members of \( h(s) \) to the right. Therefore, if for example in \( t \) all entities except \( x_1 \) are shared locked, then its corresponding hyperedges are denoted

\[
\{x_1\}, \{x_1|x_2\}, \{x_1|x_2,x_3\}, \ldots, \{x_1|x_2,\ldots,x_{n-1}\}, \{x_1|x_2,\ldots,x_n\}.
\]

Example 1 (continued): The hyperedges corresponding to \( t_1 \) are \( h_1 = \{x_1\} \), \( h_2 = \{x_1|x_2\} \), \( h_3 = \{x_1|x_2|x_3\} \), \( \ldots \), and \( h_{10} = \{x_1|x_2,\ldots,x_{n-1}\} \), \( \{x_1|x_2,\ldots,x_n\} \).

The hyperedges corresponding to \( t_2 \) are \( h_4 = \{y_1\} \), \( h_5 = \{y_1|y\} \), \( h_6 = \{y_1|y\} \), and \( h_7 = \{y_1|y\} \). The corresponding hypergraph is illustrated in Fig. 1.

For two arbitrary hyperedges \( h \) and \( f \), the potentially-conflicting pairs are \( \{h(e),f(e)\}, \{h(s),f(e)\} \) and \( \{h(e),f(s)\} \). They represent conflicting locks. Two hyperedges conflict if the intersection of one of their potentially-conflicting pairs is nonempty. Also, we say that an entity \( x \) in mode \( m \) (exclusive or shared)
conflicts with a hyperedge $h$, if $x$ belongs to $h$, and if $m$ is shared, then $x$ is in $h(e)$.

A path in the hypergraph corresponding to a set of transactions is from an "entity in a mode" to another "entity in a mode". Formally, a path from entity $x$-in-mode-$m$ to entity $y$-in-mode-$l$ is sequence of hyperedges $h_1,h_2,...,h_k$ such that: $h_i$ conflicts with $h_{i-1}$ for $i=2,...,k$, and $x$-in-mode-$m$ conflicts with $h_1$, and $y$-in-mode-$l$ conflicts with $h_k$. The path is denoted $xh_1h_2...h_ky$. For example, in the hypergraph of figure 1, corresponding to the set of transactions in example 1, $xh_3h_5y$ is a path from $x$-in-shared-mode to $y$-in-shared-mode. It can be determined in $O(N^2)$ steps whether there is a path between two entities in a hypergraph corresponding to a set of input transactions $\tau$; $N$ is the total length of the input-set $\tau$. This time complexity can be easily achieved by using depth-first-search on the graph representing $H(\tau)$. This is a bipartite undirected graph having one node for each entity and one node for each hyperedge, and an edge between a hyperedge and an entity it contains. For each edge of the representing graph, an indication whether the entity is contained in the shared or exclusive part is required. Note that the existence of a path can actually be determined in $O(NM)$ steps, where $M$ is the maximal length of a transaction. However, in discussing time-complexity in this paper we will disregard functions of the input other than its length.

Assume that a set of entities $K$ is partitioned into $s$ and $e$ subsets. $K$ separates in $H(\tau)$ two entities, $x$-in-mode-$m$ and $y$-in-mode-$l$, if: after eliminating from $H(\tau)$ all hyperedges which conflict with $K$, there remains no path from $x$-in-mode-$m$ to $y$-in-mode-$l$. For a transaction $t \in \tau$ and entity $x$ locked in $t$, denote by $b_x$ the...
set of entities locked by \( t \) before \( Lx \), and unlocked after \( Lx \). \( b_x \) is partitioned into \( s \) and \( e \) subsets \( b_x(s) \) and \( b_x(e) \). \( x \) is *safely locked* in \( t \) if \( b_x \) separates any entity \( y \) unlocked before \( Lx \), from \( x \) (in the modes \( t \) locks them respectively). \( t \) is *safe* if every entity that \( t \) accesses is safely locked.

Theorem 1 [13, Section 8]: A transaction system \( \tau \) ensures serializability if every \( t \in \tau \) is safe.

4. THE ALGORITHM

Assume that the input set of 2PL transactions is \( \tau \). Our algorithm first constructs \( H(\tau) \), and then outputs new unlock positions for each transaction \( t \in \tau \), while preserving the safety of \( t \). Determining new unlock positions for \( t \) proceeds in four stages. Stage I uses Theorem 1 to identify for each lock of \( t \) entities that can be unlocked before it. \( H(\tau) \) is the only global information (about the other transactions) which is necessary for this purpose. Stage II uses the output of Stage I to build a family of sets. The members of each set are possible new unlock positions. From each set at least one new unlock position must be selected in the output transaction, in order to ensure safety of \( t \). This still leaves a certain amount of freedom in selecting the new unlock positions. Therefore, in Stage III, we assign a weight to each unlock position. It represents how "bad" is the position as far as restricting concurrency in \( \tau \). Finally, in Stage IV, we select from the possible new unlock positions the ones which restrict concurrency the least, according to a concurrency measure we propose.

Stage I: First note intuitively how Theorem 1 is used to identify entities which \( t \) can unlock before it has issued all lock requests. Consider \( t_1 \) in example 1. If \( x \) is unlocked after \( ELx \), then even if \( y \) is unlocked before \( ELx \), \( x \) remains safely locked in \( t_1 \). Generally, in this stage we construct for each entity \( x \) of \( t \) a family \( C(x) \) of *path breaking sets* (pbs's). Each pbs is a subset of the set of entities which \( t \) locks before \( x \). This set is partitioned into \( s \) and \( e \) subsets. If one entity of a pbs is unlocked after \( Lx \), then paths to \( x \) in \( H(\tau) \) are broken. \( C(x) \) consists of the minimal set of pbs's, which satisfy the following condition: if at least one entity from each pbs of \( C(x) \) is unlocked after \( Lx \), then the rest of the entities can be unlocked before \( C(x) \), and \( x \) remains safely locked. \( C(x) \) is built by the following procedure for each entity \( x \) accessed in \( t \).
1. let $H' = H(\tau)$
2. for each entity $y$ locked before $Lx$ do;
3. while there is a path $p = yh_1, \ldots, h_mx$ in $H'$ do;
4. denote by $h_j$ the last hyperedge on the path $p$, which conflicts with
   some entity locked before $Lx$ in $t$. Add to $C(x)$ a pbs $A$ consisting of
   all entities locked before $Lx$, which conflict with $h_j$;
5. eliminate $h_j$ from $H'$;
6. end;
7. end;
8. eliminate from $C(x)$ any set which contains another set of $C(x)$;

This procedure can be executed in $O(N^2)$ steps by performing a depth-first-search starting from $x$. Each time a
hyperedge which conflicts with the set of entities accessed before $x$ is encountered, all conflicting entities are
added as a set to $C(x)$, and the hyperedge is eliminated.

**Stage II:** We assume now that $t$ locks entities $x_1, \ldots, x_n$, in this order, and will build a set $S_i$ of possible new
unlock positions for each $x_i$. $S_i$ consists of $\{(x_i, 0)\} \cup \{(x_i, k)\}$ where $k$ is the index of an entity locked after the last
action on $x_i$. $(x_i, 0)$ represents positioning of $x_i$ immediately after the last action on $x_i$, and $(x_i, k)$ represents
the positioning of $x_i$ between $Lx_k$ and the first action after it. Denote $S = \bigcup_{i=1}^{n} S_i$. Let $Q$ be a set of some $C(x_j)$.
Based on it we build a new set, $D(Q)$, defined as $\{(x_i, k) | x_i \in Q$ and $k \geq j$ and $(x_i, k) \in S \}$. If $x_i \in Q$,
and the last action on $x_i$ succeeds $Lx_j$ in $t$, then $(x_i, 0)$ is added to $D(Q)$.
$D(Q)$ is build for each $Q$, in each $C(x_j)$. Denote by $C$ the family of sets so obtained. At this point the algorith-
then can move unlocks to the "left" in $t$, namely closer to the last action on the corresponding entity. As long
as the new unlock positions "hit" every set of $C$, the transaction $t$ remains safely locked; by "hitting" we mean
that the intersection of the set of final unlock positions with every set in $C$, is nonempty. However, there can
be many sets of final unlock positions with this property, all of them representing an improvement over $2^{\Pi L}$.
The next stages identify one such set.

**Stage III:** In this stage we assign a weight, $w(x_i, k)$, to each unlock position $(x_i, k)$. It enables determining a set
of final unlock positions in the next stage. \( w(x_i,k) \) represents the total number of time units that other transactions of \( \tau \) may wait for \( t \) on \( x_i \) if \( t \) uses the unlock position. \( w(x_i,k) \) is obtained by multiplying the number of time units \( x_i \) is kept locked, denoted \( q \), by the number \( r \) of other transactions in \( \tau \) which may wait for \( t \) on \( x_i \). We assume that each read, write, or lock step takes one time unit; unlock steps are instantaneous and take zero time units. Thus \( q \) is the number of read, write, or lock steps (on any entity), between \( Lx_i \) and the position \((x_i,k)\). \( r \) is the number of transactions in \( \tau-\{t\} \) which lock \( x_i \) in a conflicting mode.

**Stage IV:** In this stage we want to select from \( S \) (built in Stage II), a set of final unlock positions, \( F \), such that:

1. \( F \) "hits" every \( Q \in C \), namely \( F \cap Q \neq \emptyset \) and
2. \( \sum_{(x_i,k) \in F} w(x_i,k) \), denoted \( M(t) \), is minimal among all sets which satisfy the first condition. \( M(t) \) represents the extent to which \( t \) may block other transactions of \( \tau \) during execution.) However, this is the Weighted Hitting Set problem, which is known to be NP-complete ([6,7]). The previous stages can be performed in polynomial time. In order to preserve this time complexity we approximate the optimal solution using the following Greedy-type procedure. It starts with the set of earliest unlock positions, and at each iteration postpones an unlock until safety of \( t \) is assured.

1. let \( F \) consist of the positions \((x_i,0)\), for \( 1 \leq i \leq n \);
2. remove from \( C \) any set which contains another set, or contains an entity of \( F \);
3. do while \( C \) is not empty;
4. move from \( S \) to \( F \) the position \((x_i,k)\) with the minimal relative weight;
   (the relative weight is the weight, divided by the number of sets of \( C \) the position appears in)
5. remove from \( C \) the sets which the member \((x_i,k)\) hits;
6. remove from \( F \) the earlier unlock positions for \( x_i \) (i.e. \((x_i,j)\) where \( j < k \));
7. end;

It can be shown based on an analysis in [4] (about the set cover problem) that the value we obtain for \( M(t) \) is worse than the optimal one by a factor of at most \( \sum_{i=1}^{d} \frac{1}{l} \); \( d \) is the maximal number of sets of \( C \) that an entity is contained in. In terms of our problem, \( d \) is the maximal number of transactions conflicting with \( t \) on an entity, times \( n \).
This concludes the description of the algorithm. A straightforward analysis shows that its running time is not more than $O(N^4)$.

Example 1 (continued): We will demonstrate how our algorithm derives new unlock positions for $t_1$. $C(y)$ consists of $Q_1 = \{x\}$. $C(z)$ consists of two pbs's $Q_2 = \{x\}$ and $Q_3 = \{x, y\}$. The key to advancing the unlock of $y$ is $Q_3$, which basically indicates that to ensure serializability $E L z$ has to precede $U x$ or $U y$, but not necessarily both.

$S$ consists of (weights in square brackets) $(x, 0)[4], (x, 3)[5], (y, 0)[1], (y, 3)[3], (z, 0)[1]$.

$D(Q_1) = \{(x, 0), (x, 3)\}, D(Q_2) = \{(x, 3)\}, D(Q_3) = \{(x, 3)(y, 3)\}$.

The set of final unlock positions selected by our algorithm consists of $(x, 3), (y, 0), (z, 0)$ representing $t_1$ with $U y$ issued between $R y$ and $W x$. Similarly, we can show that $t_2$ can unlock $x$ immediately after $R x$.

5. CASCADING RESTARTS

Assume that during execution a transaction $t$ reads entity $x$, which was modified and unlocked by a transaction $r$ before $r$ obtained all locks and committed. Since $t$ reads an uncommitted value of $x$, it cannot commit until $r$ does. In the most common implementation of 2PL, i.e. the one in which entities are unlocked only after commitment, this complication does not arise; but in this 2PL case $t$ waits for commitment of $r$ at the $L x$ request, rather than $t$’s commit point, which may occur much later. Therefore, even when considering the wait-for-commit, advance unlocking is advantageous. However, the advance unlocking algorithm raises the problem of cascading restarts. Restarting a transaction $t$ may cause the transactions which read entities written by $t$ to be restarted, which in turn may cause other restarts, etc. The most common cause of the first restart for the transaction systems we discuss is deadlock. Thus, to drastically reduce the risk of cascading restarts we would like to ensure that a transaction cannot participate in a deadlock cycle after it has unlocked an entity it held locked in exclusive mode. An easy way of doing so is to assume a fictitious action on each such entity after the last lock that the transaction issues. This will force our algorithm to unlock the exclusively locked entities after the last lock has been obtained, and will prevent cascading restarts.

However, we can do better. Assume that a transaction $t$ in a set $T$, can be deadlocked waiting on $L x$, while holding a lock on which another deadlocked transaction is waiting. Then we say that $L x$ is a dangerous lock of
In many cases, the last dangerous lock of a transaction is issued before its last lock. Then our algorithm has more freedom. It can be shown (based on [14, Theorem 6]), that in a transaction output by our algorithm, a lock is dangerous if and only if it is dangerous in the input transaction. Thus, in order to ensure freedom from cascading restarts as a result of deadlock, we can proceed as follows. First, we determine the last dangerous lock of the input transaction \( t \). Then, we insert in \( t \) after the last dangerous lock, \( L_y \), a fictitious action on every entity exclusively locked before \( L_y \). Then we run our algorithm as outlined in section 4.

Example 2: Assume that \( t_1 \) and \( t_2 \) are as in Example 1, except that the shared locks of \( t_1 \) and \( t_2 \) are exclusive, and the reads which \( t_1 \) and \( t_2 \) issue for \( y \) and \( x \) respectively, are writes. Then, naively applying the rule that exclusively locked entities can only be unlocked after the last lock has been obtained will prevent our algorithm from improving concurrency over 2PL. However, the scheme we discuss in this section will enable us to determine that \( E_L z \) is not dangerous for \( t_1 \) (or \( t_2 \)), thus enabling unlocking of \( y \) immediately after \( W_y \).

Note that a transaction \( t \) can be deadlocked waiting on a lock \( L_y \), which is not dangerous. This is the case if \( y \) is held locked by a deadlocked transaction, but \( t \) does not hold any lock that another deadlocked transaction is waiting on. However, in this case the deadlock cannot be resolved by restarting \( t \). Therefore, although \( t \) is deadlocked, it does not participate in a deadlock cycle. Thus, it may have unlocked an exclusive lock before \( L_y \), without risking cascading restarts.

How can we identify the last dangerous lock of a transaction? The problem is NP-hard for the following reason. Determining whether a set of two phase locked transactions is deadlock free, is coNP-complete (see [14]). If we could determine for a transaction what is the last "dangerous" lock, in polynomial time, we could determine in polynomial time whether the set of transactions is deadlock-free. Thus, in order to retain the polynomial time complexity of our algorithm, we define the following condition, denoted A1. It is necessary but not sufficient for a lock to be dangerous.

**Definition:** Condition A1 holds for lock \( L_y \), in a transaction \( t \), of a two-phase-locked system \( \tau \), if and only if: there is an entity \( x \), locked before \( L_y \) in \( t \), such that \( b_x - \{ x \} \) does not separate \( x \) from \( y \) in \( H(\tau) \).

For example, assume that \( \tau \) consists of two transactions, \( t_1 \) and \( t_2 \). Assume further that the locks of \( t_1 \) are \( E_L w, E_L z, E_L y \), in this order, and the locks of \( t_2 \) are \( E_L x, E_L y, E_L z \) in this order. Then condition A1 is satisfied for \( E_L y \) in \( t_1 \), because \( w \) does not separate \( y \) from \( z \) in \( H(\tau) \).
Condition A1 can be tested in $O(N^2)$ steps. A lock that satisfies it is possibly dangerous.

Theorem 2: Let $\tau$ be a set of two phase locked transactions. Assume that $L_y_1$ is a dangerous lock of $t_1 \in \tau$. Then condition A1 holds for $L_y_1$ in $t_1$.

Proof: Assume that $S$ is a partial schedule of $\tau$ resulting in a deadlock. Assume further, that in the deadlock situation $t_1$ is waiting to lock $y_1$, and holding a lock on $y_2$, on which another transaction $t_2 \in \tau$ is waiting. Then there are transactions $t_2, ..., t_k$ executed in $S$, and entities $y_2, ..., y_k$, such that for $1 \leq i \leq k$:

1) The step $t_i$ waits on in the deadlock situation is $L_y_i$; and
2) $t_i$ holds $y_{i+1}$ (mod $k$) locked in $S$; and
3) The lock that $t_i$ issues for $y_i$ conflicts with the lock issued by $t_{i-1}$ (mod $k$) for $y_i$.

Assume now that $t_i$ locks the set of entities $A_i$ in $S$. Then, by the way $H(\tau)$ was built, $h_i = A_i \cup \{y_i\}$ is a hyperedge of $H(\tau)$ for each $1 \leq i \leq k$. Note that $y_2 \in b_1$, and $y_2$ is the only entity of $b_1$, which conflicts with some $h_i$ for $2 \leq i \leq k$; otherwise $S$ cannot be a legal partial schedule. Note also, that $y_2h_2, ..., h_ky_1$ is a path in $H(\tau)$. Therefore, $b_1\{y_2\}$ does not separate $y_2$ from $y_1$ in $H(\tau)$. [ ]

Based on theorem 2, by ensuring that our algorithm does not unlock an exclusively locked entity before the last "possibly dangerous" lock, we prevent cascading restarts as a result of deadlock. As the following example shows, the last "possibly dangerous" lock need not be the last lock in a transaction.

Example 2 (continued): The hypergraph corresponding to $t_1$ and $t_2$ consists of hyperedges $\{x\}$, $\{y_1\}$, $\{x, y_1\}$ and $\{x, y, z\}$. $z$ is not possibly-dangerous in $t_1$ because $y$ separates $x$ from $z$, and $x$ separates $y$ from $z$. [ ]

6. EXTENSION TO DISTRIBUTED TRANSACTIONS

6.1 Preliminaries

In a distributed database, the entities referenced by the read, write, and lock steps are partitioned among sites (or computers). A transaction usually consists of several processes which execute simultaneously at different sites. The main difference between a centralized and a distributed transaction, is that in the latter case, a time precedence between two steps executing at different sites may not be established. For example, a transaction may have two steps $A$ and $B$, executed at different sites, such that in one run of the transaction $A$
occurs before $B$, and in another, $B$ occurs before $A$. Therefore, we cannot assume any more that the steps of a transaction are totally ordered at system definition.

A (distributed) transaction is modeled by a directed acyclic graph (dag), whose nodes are labeled by the transaction steps (see [16]). The arcs represent time precedence of step execution. As in the centralized case, we assume that each step is a $Rx$, $Wx$, $SLx$, $ELx$, or $Ux$ operation. Again, we assume that an entity $x$ is read (written) if and only if it is preceded by a $SLx$ ($ELx$) step and succeeded by a $Ux$. We assume that all the steps which reference entities residing at the same site are totally ordered. Practically, it reflects the fact that they are (sequentially) executed by one computer; theoretically, it is a convenient assumption because if the database resides at one site only (is centralized), then the distributed transaction definition reduces to our definition of a centralized transaction. Two steps referencing entities residing at different sites (in other words, executed at different sites) may or may not precede one another in the transaction dag. The graph of Fig. 2 represents a transaction.

We can think of a partial order $T'$ as the set of its linear extensions (total orders compatible with it); if $t$ is a linear extension of $T$ we will write $t \in T$. Let $\tau = \{T_1, ..., T_n\}$ be a transaction system. Denote $\tau' = \{t \mid t \in T_i \text{ for some } T_i \in \tau\}$. A schedule of $\tau$ is a schedule of $\tau'$, when each member of $\tau'$ is regarded as a centralized transaction. Serializability (assurance) is defined as in the centralized case.

The main problem encountered in extending the algorithm to a distributed transaction system is that there is no general condition which is sufficient to ensure serializability, and can be performed efficiently, such as

---

Figure 2: A transaction accessing entities $x$, $y$, $z$ each residing at a different site. The transaction is two-phase-locked.
Theorem 1 provides for the centralized case. By the definitions given for the distributed case, it is clear that a set of transactions ensures serializability if and only if its set of linear extensions does so. However, the size of the set of linear extensions can be exponential in the size of the input set of transactions. The solution to the problem will be given by Theorem 3 formulated in subsection 6.2. It establishes that a condition, denoted C1 and defined in the next subsection, is equivalent to all linear extensions being safe. It can be determined whether C1 holds for an input transaction system in a number of steps not higher than \(O(N^6)\).

We would like to mention that Theorem 3 has implications beyond the algorithm discussed in this paper. In [13] Yannakakis introduces a general class of locking policies, called L-policies, which subsume the existing locking policies that ensure serializability (e.g., tree policy [10], dag.policy [11]). A L-policy gives for every entity \(x\) of the database, a set of lock, unlock sequences, denoted \(W(x)\). A transaction obeys the policy, if before locking an arbitrary entity \(y\), it executes a sequence of lock-unlock steps which is in \(W(y)\). Yannakakis proves that a L-policy is "safe" (namely, if every transaction in a set obeys it, then the set ensures serializability) if and only if each prefix in \(\bigcup_{x} W(x)\) is safe in the sense of Theorem 1. Theorem 3 enables testing efficiently whether a L-policy is safe or not, even if the set \(W(x)\) does not consist of sequences, but of partial orders. Thus, the definition of a L-policy, and its efficient test for safety, can be extended to distributed transactions. Also, even for centralized transactions, the definition of a L-policy can be made more concise (even exponentially so), while preserving the polynomial time complexity of the test for safety. The analysis in the next subsection also disproves a conjecture made in [16, Section 6]. The conjecture is that Theorem 1 can be extended to the distributed case by simply interpreting "before" in the safety definition of a centralized transaction, as "preceding" in a partial order.

6.2 A Sufficient Condition for Serializability Assurance in the Distributed-Transaction Case

For defining and proving condition C1 we need the following definitions. Let \(T\) be a (distributed) transaction, and \(C\) a subset of the steps of \(T\). Then the pruning of \(T\) at \(C\), denoted \(T(C)\), consists of the subgraph of \(T\) induced by the subset of steps \(C'\); \(C'\) is the set of steps of \(T\) which are not in \(C\), and do not succeed (in \(T\)) a step of \(C\). Intuitively, it represents all the steps that \(T\) can execute before executing a step of \(C\). For example, if \(T\) is the transaction of Fig. 2, then \(T(\{Rx,Ry\})\) is the graph \((\{ELx,SLy\}, \emptyset)\).
Next, we define based on $T$ a digraph, denoted by $D(T)$. Its nodes are the entities referenced in $T$, and $x \rightarrow y$ is an arc in $D(T)$ if and only if $Lx$ precedes $Uy$ in $T$. Assume that $H$ is some hypergraph whose nodes are the database entities, and each hyperedge is partitioned into an exclusive part and a shared part. For a hyperedge $h$ of $H$, denote by $D(T,h)$ the subgraph of $D(T)$ induced by the entities on which $h$ and $T$ conflict.

For a hyperedge $h$ of $H$, let $h_T = \{Lx | Lx \text{ is issued by } T, h \text{ contains } x \text{ in a conflicting mode, and there is no } Ly \text{ which precedes } Lx \text{ such that } h \text{ contains } y \text{ in a conflicting mode} \}$.

We say that condition C1 holds for transaction $T$ with respect to hypergraph $H$, if the two following subconditions hold:

C1(a) For each hyperedge $h$ of $H$ the graph $D(T,h)$ is strongly connected.

C1(b) For each hyperedge $h$ of $H$ if:

1) all hyperedges which conflict with an entity locked in $h_T$, except $h$, are eliminated from $H$; and

2) all hyperedges which conflict with an entity locked-but-not-unlocked in $T(h_T)$ are eliminated.

then there remains no path in $H$ from an entity unlocked in $T(h_T)$ to an entity locked in $h_T$ (see Fig.3).

Lemma 1: Assume that subcondition C1(a) holds for a transaction $T$ with respect to a hypergraph $H$. Let $t$ be a linear extension of $T$, and $Lx$ a lock in $t$. Assume that the set $b_x$ does not conflict with some hyperedge $h$, but $x$ does. Then $x$ is the first entity which conflicts with $h$, that $t$ locks.

---

Figure 3: Transaction $T$ and hyperedge $h$ conflict on entities $x,y,z$. Partitioning of $T$ into $T(h_T)$ (above the broken line), and the other steps, is illustrated. Condition C1(b) states that if $wh_1 \cdots h_{m-1}hx$ is a path in $H$, then $r$ in shared mode conflicts with some $h_i$ for $1 \leq i \leq m-1$. 

---
Proof: Assume that the step \(Ly\) precedes \(Lx\) in \(t\), and \(y\) conflicts with \(h\). Let \(y=y_1\to\ldots\to y_k=x\) be a path from \(y\) to \(x\) in \(D(T,h)\). Based on the fact that \(b_x\) does not conflict with \(h\), it is easy to show by induction on \(i\), that \(Uy_i\) precedes \(Lx\) in \(t\), for \(i=1,\ldots,k\). But this implies that \(Ux (=Uy_k)\) precedes \(Lx\). Contradiction. \(\square\)

The implication of Lemma 1 is that under the stated conditions, \(t\) cannot have a lock and an unlock of an entity which conflicts with \(h\), before the \(Lx\) step.

We say that condition C2 holds for transaction \(T\) w.r.t. hypergraph \(H\) if for every \(Lx\) in every extension \(t\) of \(T\), the following is true. The set of entities \(b_x\) separates every entity unlocked before \(Lx\) in \(t\), from \(x\), (in the respective modes).

Lemma 2: Condition C1 holds for a transaction \(T\) with respect to a hypergraph \(H\), if and only if condition C2 holds for \(T\) with respect to \(H\).

Proof: \((\Leftarrow)\) Proof of C1(a): Assume that for the hyperedge \(h_0\) the following is true. In \(D(T,h_0)\) there is a pair of entities, denoted \(x\) and \(y\), with no path from \(x\) to \(y\) (namely; the subgraph is not strongly connected). Then we can build an extension \(i_0\) of \(T\), to show that C2 does not hold. In \(i_0\), the set of steps in \(V\) (defined below) precedes the rest of the steps of \(T\). \(V\) is built as follows:

1. Initialize \(V\) to \(Uy\) and its predecessors in \(T\).
2. While \(V\) contains some \(Lz\) but not \(Uz\), for some \(z\) which conflicts with \(h_0\), add \(Uz\) and its predecessors (in \(T\)) to \(V\).

\(T\) can be linearly extended into \(i_0\) because the predecessors of every node in \(V\), are also in \(V\). It can easily be proved, by induction on the number of executions of step 2 of the above procedure, that: if \(w\) is locked in a mode which conflicts with \(h_0\), and \(Lw\in V\), then there is a path in \(D(T,h_0)\) from \(w\) to \(y\). Thus, \(Lx\) is not in \(V\). Therefore, there exists an entity of \(h_0\) locked in a conflicting mode by \(i_0\); the lock is issued after all steps of \(V\) are completed. Consider \(z\), the first such entity locked by \(i_0\). It is clear that \(yh_0z\) constitutes a path in \(H\), and \(h_0\) does not conflict with any entities of \(b_z\); contradiction to C2.

Proof of C1(b): Assume that C1(b) does not hold, and consider some hyperedge \(h\) which violates it. Consider an extension \(t\) of \(T\), in which all steps of \(h_T\) and their successors, succeed every step of \(T(h_T)\). Let \(Lx\) be the first of the steps in \(h_T\) issued by \(t\). It is clear that \(b_x\) does not separate \(x\) from \(y\) in \(H\); contradiction to C2.
Assume that \( C_2 \) is false. Namely, there exists a extension \( t \) of \( T \), which violates the condition. Let \( Lx \) be the last lock of \( t \) which has a \( Uy \) preceding it, but \( b_x \) does not separate \( x \) from \( y \) in \( H \). Consider a path \( y h_1, \ldots, h_m x \) in \( H \) such that: no \( h_i \) conflicts with \( b_x \), and \( x \) does not conflict with a hyperedge \( h_i \) for \( 1 \leq i < m \) (otherwise we take the shorter path). Denote \( h_m \) by \( h \). By Lemma 1, among all entities on which \( t \) and \( h \) conflict, \( x \) is the first one that \( t \) locks. Thus, \( Lx \in h_T \), and \( m > 1 \). This also implies that \( Uy \) is in \( T(h_T) \) (since \( Uy \) precedes \( t \) the first lock of a conflicting entity, it cannot succeed in \( T \) any lock of \( h_T \)). By condition \( C1(b) \), the set of entities locked but not unlocked in \( T(h_T) \) conflicts with some hyperedge \( h_i \), for \( 1 \leq i < m \).

Denote by \( R_i \) the set of entities on which \( h_i \) and \( T(h_T) \) conflict, for \( 1 \leq i < m \). Let \( R = \bigcup_{i=1}^{m-1} R_i \). No entity of \( R \) can be unlocked in \( t \) before \( Lx \) (every such unlock succeeds some lock in \( h_T \)). Thus, all the entities of \( R \) must be locked after \( Lx \) in \( t \). Consider the first entity of \( R \) locked in \( t \). Denote it \( z \). Assume that \( z \) conflicts with \( h_q \), for \( q < m \). It is easy to see that \( y h_1, \ldots, h_q z \) is a path in which no hyperedge conflicts with \( b_x \). But this is a contradiction to the fact that \( Lx \) is the last lock of \( t \) which is not safely locked. 

Next, we define a hypergraph equivalent to \( H(\tau') \) (the hypergraph corresponding to the set of linear extensions of \( \tau \)). The equivalence is w.r.t. separation by a set of entities, and is precisely stated in the next lemma. As opposed to \( H(\tau') \), it has only a polynomial number of hyperedges. Assume that \( \tau = \{ T_1, \ldots, T_k \} \). For each \( T_i \in \tau \) we define a family \( F_i \) of (shared-exclusive partitioned) sets of entities, as follows. For a pair of entities \( x \) and \( y \) locked in \( T_i \), their corresponding set (or hyperedge), denoted \( h_{xy} \), consists of: \( x \) and \( y \), and each entity whose lock precedes \( Lx \) or \( Ly \). \( F_i \) consists of the sets \( h_{xy} \) for each pair of entities locked in \( T_i \). Define \( F = \bigcup_{i=1}^{k} F_i \). By the way each \( h_{xy} \) was built, and by the definition of hyperedges corresponding to a centralized transaction, it is clear that \( F \) is a subset of the hyperedges of \( H(\tau') \). Let \( N \) be the set of database entities, and define the hypergraph \( Q = (N, F) \).

**Lemma 3:** Let \( \tau \) be a set of distributed transactions. A set of nodes \( D \) (partitioned into \( s \) and \( e \) subsets) separates two entities in \( H(\tau') = (N, F') \), if and only if \( D \) separates them in \( Q(\tau) = (N, F) \).

**Proof:** 

\( \Rightarrow \) Trivial since \( F \subseteq F' \).

\( \Leftarrow \) Assume \( p = x h_1', \ldots, h_n' y \) is a path in \( H(\tau') \), and \( D \) does not conflict with any \( h_i' \). We will show that there
is a path \( xh_1, \ldots, h_n y \) such that \( h_i \) is a hyperedge of \( F \) and \( h_i \subset h'_i \) (namely, \( h_i(e) \subset h'_i(e) \) and \( h_i(s) \subset h'_i(s) \)). Therefore, \( D \) does not conflict with any \( h_i \), and does not separate \( x \) from \( y \) in \( \hat{Q}(t) \). Consider some \( h'_i \) in \( p \), and assume that it conflicts on \( v \) with \( h'_{i-1} \) (or \( v=x \)), and on \( w \) with \( h'_{i+1} \) (or \( w=y \)). Assume that \( h'_i \) corresponds to a linear extension \( t \) in \( \tau \) of some transaction \( T \in \tau \). Before executing \( Lv \) (\( Lw \)), \( t \) must execute all the steps preceding \( Lv \) (\( Lw \)) in \( T \). Therefore, \( h'_i \) includes \( h_{vw} \) built for \( T \).

**Theorem 3:** Let \( \tau \) be a set of distributed transactions, and \( \tau' \) the set of linear extensions of transactions in \( \tau \). Then every member of \( \tau' \) (regarded as a centralized transaction) is safe, if and only if condition C1 holds for every \( T \in \tau \), w.r.t. the hypergraph \( Q(\tau) \).

**Proof:** Immediate from Lemmas 2,3. []

**Corollary 1:** A set of distributed transactions \( \tau \) ensures serializability if condition C1 holds for every \( T \in \tau \), w.r.t. \( Q(\tau) \). []

### 6.3 The Algorithm for Distributed Transactions

Based on Corollary 1, the algorithm of Section 4 can be extended to a set of two-phase-locked distributed transactions, \( \tau \). The main ideas of the extension are as follows. For each transaction \( T \) of \( \tau \) unlocks are advanced, without changing any precedences between the remaining steps. In stage I a set, \( C \), of pbs's is built. Each pbs consists of ordered pairs of entities. The ordered pair \((x,y)\) indicates that \( Lx \) precedes \( Uy \) in the output transaction, built from \( T \). \( C \) is built incrementally, by successively examining each hyperedge in \( Q(\tau) \). Hyperedge \( h \) should be examined before any \( h' \) which properly includes it. \( C \) is not unique, and different scanning orders of the hyperedges may produce different \( C \)'s. For each hyperedge \( h \) two sets of pbs's \( C_a(h) \) and \( C_b(h) \) are added to \( C \). \( C_a(h) \) ensures that subcondition C1(a) holds for \( h \), and \( C_b(h) \) ensures that subcondition C1(b) holds. Initially, \( D(T) \) has an arc \((x,y)\) for each pair of entities for which \( Lx \) precedes \( Ly \) in \( T \). Each time a singleton set is added to \( C \), the respective arc is added to \( D(T) \).

\( C_a(h) \) consists of a minimal number of pbs's, such that if at least one member from each pbs is added as an arc to \( D(T,h) \), then \( D(T,h) \) becomes strongly connected. It is built based on a linear-time algorithm given in [19], to solve the unweighted strong-connectivity-augmentation problem (in our case weights are assigned only in
stage III). This is the problem of finding a minimal number of arcs to be added to a digraph, to make it strongly connected. The output of that algorithm consists of a minimal number of pairs of strongly connected components (of $D(T,h)$), such that: if an arc is added to $D(T,h)$ from a node in $a$ to a node in $b$, for each scc pair $(a,b)$, then $D(T,h)$ becomes strongly connected. In our case, we add a pbs to $C$ for each scc pair $(a,b)$ output by the algorithm. The pbs consists of all possible pairs of nodes, the first from $a$, and the second from $b$.

$C_b(h)$ is established by the procedure given below, which is similar to the one for centralized transactions. The main difference is, that instead of considering the sequence of steps up to $Lx$, we consider $T(h_T)$.

1. let $H'=Q(t)$;
2. eliminate from $H'$ all hyperedges which conflict with an entity locked in $h_T$, except $h$;
3. for each entity $y$ referenced in $T(h_T)$ do;
4. while there is a path $p=yh_1,...,h_nhx$ in $H'$, where $x$ is in $h_T$, do;
5. denote by $h_j$ the last hyperedge on $p$, which conflicts with some entity locked in $T(h_T)$. Add to $C_b(h)$ a pbs consisting of all pairs $(w,x)$, such that $z$ is locked in $T(h_T)$ and conflicts with $h_j$, and $w$ is locked in $h_T$;
6. eliminate $h_j$ from $H'$;
7. end;
8. end;

In stage II, each member $(x,y)$ in a pbs is replaced by the feasible positions of $Uy$ which succeed $Lx$. A feasible position for $Uy$ is a position between two consecutive steps executed at the residence site of $y$, and succeeding the last action on $y$.

In stage III, the weight of an unlock position is established as in the centralized case, with the following exception. The number of time units $x$ is kept locked is the maximal number of read, write, or lock steps, on a path from $Lx$ to $Ux$ in that position. Stage IV is carried out as in the centralized case.

The modification to prevent cascading restarts (Section 5) is also extendible to distributed transactions. Assume that $T$ is an input 2PL transaction. $Ly$ is "possibly dangerous" in $T$, if there is a $Lx$ which does not succeed $Ly$, for which the set \{\{z | Lz precedes $Lx$ or $Ly$\} \{x\} \} does not separate $x$ from $y$ in $Q(t)$. If in our algorithm we ensure that all exclusively locked entities of $T$, are unlocked after all possibly dangerous locks,
then deadlock cannot cause cascading restarts.

**Example 3:** Assume that \( \tau \) consists of the transaction \( T_1 \) of Fig. 4(a), and the transaction of \( T_2 \) of Fig. 2 (both two phase locked). Then the algorithm will determine that in \( T_1 \), \( y \) can be safely unlocked immediately after \( \text{Wy} \), obtaining the transaction of Fig. 4(b). The hyperedges of \( O(\tau) \) are: \( h_1 = \{ x \} \), \( h_2 = \{ x, y \} \), \( h_3 = \{ y \} \), \( h_4 = \{ y, z \} \), \( h_5 = \{ x, y \} \), \( h_6 = \{ x, z, w \} \) \), \( h_7 = \{ y, x \} \), \( h_8 = \{ y, x, z \} \), \( h_9 = \{ y, x, z, w \} \).

Stage I determines \( C \) as follows:

\[
C_a(h_1) = \emptyset \quad \text{and} \quad C_b(h_1) = \{(x, y)\}
\]
\[
C_a(h_2) = \emptyset \quad \text{and} \quad C_b(h_2) = \emptyset
\]
\[
C_a(h_3) = \{(x, y)\} \quad \text{and} \quad C_b(h_3) = \emptyset
\]
\[
C_a(h_4) = \emptyset \quad \text{and} \quad C_b(h_4) = \{(y, x)\}
\]

Overall, \( C \) consists of three sets \( \{(x, y)\} \), \( \{(y, x)\} \), \( \{(x, x), (y, y)\} \). Stages II and III obtain the following instance of the weighted hitting set problem (weights in square brackets):

\[
S_1 = \{ \text{Uy-immediately-after-Wy} \ [2], \text{Uy-immediately-after-} \text{SLw} \ [3], \text{Uy-immediately-after-Rw} \ [4] \}
\]
\[
S_2 = \{ \text{Ux-after-Rx} \ [3] \}
\]
\[
S_3 = \{ \text{Ux-after-Rx} \ [3], \text{Uy-immediately-after-} \text{SLw} \ [3], \text{Uy-immediately-after-Rw} \ [4] \}
\]

---

![Diagram](image_url)
with the solution $U_x$-after-$Rx$, and $U_y$-immediately-after-$Wy$.

$SLz$ is not "possibly dangerous", because by eliminating the hyperedges which conflict with $\{z,x\}$ there remains no path from $y$ to $z$, and by eliminating the hyperedges which conflict with $\{y,z\}$ there remains no path from $x$ to $z$. Thus, $T_1$ cannot deadlock after advancing $Uy$. []

7. DISCUSSION

In this paper we described an efficient algorithm that advances unlocks of a given set of 2PL transactions, using symbolic name locking. The early unlocking does not compromise serializability. The algorithm is based on the fact that preanalysis enables discovery of a locking order, in a seemingly unstructured database. It can be used in a centralized or distributed database, employing shared and exclusive locks. Time complexity of the algorithm is within practical limits, especially considering that it has to be run only once, at transaction-system definition. The attractiveness of the algorithm stems from the fact that following this initial run, its benefits in the form of increased concurrency are reaped continuously during on-line execution. Since unlocking entities before a transaction commits may cause cascading restarts, we suggest a modification to the algorithm that almost eliminates the problem.

The proposed method relies on preanalysis of expected transactions, and therefore cannot handle ad hoc queries. We will make a few comments about about this difficulty. First, note that any query reading only one entity can be run even though it did not participate in the preanalysis. The reason is that these queries do not affect serializability; also, since they cannot hold a lock and then request one, they do not participate in a deadlock cycle. For the large locking granularities we discussed, these type of queries will be quite common (e.g., selection from a relation). Note that these queries stand to benefit significantly from advance unlocking in the long updating transactions. What about ad-hoc queries which access more than one database entity? Assume that we want to be able to safely submit any such query. Then we have to add an edge for each pair of database entities to the hypergraph corresponding to a set of transactions. The edge will have an empty exclusive part. Note that this will suffice to allow ad-hoc queries of any length, and may still allow an improvement of concurrency over 2PL. However, no transaction will be able to unlock an exclusively locked entity before it obtains all exclusive locks, and the ad-hoc queries will have to be two-phase-locked. If we choose not to add the edges, and still run ad-hoc queries which read more than one database entity, then the database view such query
receives may be inconsistent. However, consistency of the database will not be compromised, because any schedule restricted to updating transactions will be serializable. In other words, for the purpose of database consistency queries need not be considered. Since all updating transactions participate in the preanalysis, any one of their schedules is serializable.

ACKNOWLEDGEMENT
I wish to thank the referee for helping improve the presentation in this paper.
REFERENCES


