Deadlock Removal Using Partial Rollback in Database Systems*

Donald Fussell**
Zvi M. Kedem***
Abraham Silberschatz**

ABSTRACT

The problem of removing deadlocks from concurrent database systems using the two-phase locking protocol is considered. In particular, for systems which use no a priori information about transaction behavior in order to avoid deadlocks, it has generally been assumed necessary to totally remove and restart some transaction involved in a deadlock in order to relieve the situation. In this paper, a new approach to deadlock removal in such systems based on partial rollbacks is introduced. This approach does not in general require the total removal of a transaction to eliminate a deadlock. The task of optimizing deadlock removal using this method is discussed for systems allowing both exclusive and shared locking. A method is given for implementing this approach with no more storage overhead than that required for total removal and restart.

1.0 INTRODUCTION

Designers of large database systems have recognized for some time that the concurrent processing of multiple user transactions can have significant advantages in terms of response time to queries and updates. If one assumes that these transactions are correct in the sense that each of them maintains the integrity of the database when run alone, then the basic problem in the design of database systems which support concurrent processing is to guarantee that any allowable execution scenario involving a number of transactions will also maintain the integrity of the data [4,7,10].

Solutions to this problem to date have been almost exclusively on a strategy of dividing the database into units or entities, access to which may be controlled by a database concurrency control. The most common model for such a system involves the notion of a locking protocol. Each transaction which is permitted to execute in the system is responsible for locking an entity when it wishes to access that entity, and for unlocking the entity when it no longer needs to access it. (Note that the system may equivalently release any entities which a transaction has failed to unlock at the time the transaction terminates.) A transaction may hold either an exclusive or a shared lock on an entity. Exclusive locks are used by transactions which may read and update the entity locked, whereas those which will only read the entity. A locking protocol is then a set of rules defining allowable sequences of lock and unlock operations which may appear in a transaction. The purpose of these rules is to guarantee that any possible concurrent execution of a set of transactions which obey the rules has an effect on the database equivalent to that of some serial execution of the transactions in the set. In such case the protocol is said to guarantee serializability and any execution of a set of such transactions is said to be serializable.

Eswaran et al. [4] have shown that for systems in which there are no restrictions on the order in which entities may be locked, it is necessary and sufficient that all transactions be two-phase in order for all concurrent scenarios to be serializable. A two-phase transaction is one in which all lock operations are performed, after which all unlock operations occur. Similar systems, in which all the responsibility for obtaining and releasing access rights to entities rests with the database management system, known as a concurrency control, have been proposed in [7,10].

Although the two-phase protocol ensures serializability, it does not ensure freedom from deadlock [2]. Various strategies for resolving 40
this difficulty have been proposed in the literature. For example, it is possible to avoid the occurrence of deadlocks in systems which make use of sufficient a priori information about the accessing behavior of the transactions. The simple locking protocols of Silberschatz and Kedem [6,9] for databases in which all transactions access entities in a common hierarchical order provide both deadlock freedom and serializability. The method of Dijkstra's banker's algorithm [3], in which each transaction must declare the entities it intends to access before beginning execution, provides another example of a means by which deadlocks may be avoided in concurrent systems.

In this paper, we will be concerned with systems in which no such a priori information about transaction behavior is available. In such situations, it is not possible to avoid deadlocks. Instead, the system must have the ability to detect deadlocks after they occur and must then intervene to alleviate the situation. Such a scheme was proposed in [7,10]. In these systems, deadlock intervention is achieved by removing some transaction involved in the deadlock from the executing environment, restoring all changes made by the transaction to the database, and restarting it from the beginning. Such a procedure has a very adverse effect on the performance of the transaction operated on.

This has been considered an acceptable strategy for handling deadlocks since existing concurrent database systems have typically had at most only a few transactions concurrently active. With the advent of new hardware technologies and the evolution of computing environments consisting of networks of local machines accessing global databases and other resources, the amount of concurrency in a typical database system can be expected to rise dramatically. Deadlocks will then become a more common occurrence, and the use of such expensive means of handling the problem will become more burdensome.

Henceforth, no less drastic procedure for eliminating deadlocks after they have occurred has been considered. Our contribution is to introduce a more general procedure for deadlock intervention in systems using a two-phase protocol or its equivalent which generally has a less deleterious effect on a transaction's progress than previous methods. This will involve the rollback of a transaction involved in a deadlock to a previous state in its execution at which it is no longer involved in deadlock. Clearly, the total removal and restart procedure mentioned above is the extreme case of our operation in which the transaction is rolled back to its initial state. We explore the possibilities of rollback optimization in two-phase systems using only exclusive locks and in those using both exclusive and shared locks. We then examine in detail the implementation of limited rollback, concentrating on the tradeoff between the extra storage overhead and system monitoring requirements versus the advantage of eliminating the adverse effects of deadlock rollback on a transaction's progress. Finally, we will examine some aspects of the structure of a transaction which affect its performance in systems using limited rollback.

2.0 SYSTEM MODEL

A database system consists of a set of global data entities, a set of programs which operate on these entities, and a database concurrency control, whose function is to monitor the concurrent execution of the programs. A transaction is an execution instance of one of these programs. For each global entity in the database, there is associated with it a range of values which it may assume. Likewise, each local variable of any transaction may also assume values from some range. A database state is an assignment to each global entity in the database of an element in its range. We assume that for any given database, there exists by a set of constraints defining the set of consistent states, a subset of the set of all possible states of the database. It is also assumed that every transaction allowed to execute alone in the system with a consistent initial state will terminate, leaving the database in a consistent state.

Let $T$ be a set of concurrently executing transactions in a database system. Each transaction $T_i$ in $T$ can be viewed alternatively as either a sequence of atomic operations, each performed on a single global entity or on a variable local to $T_i$, or as a sequence of transaction states, where each atomic operation maps one state into the next.

As will be seen later, we shall need to be able to identify and reproduce certain of these states in order to rollback a transaction. It may appear cumbersome to require the system to maintain sufficient information to reproduce every single state every transaction ever achieved in order to make rollbacks possible. In a later section, however, we shall see that only relatively few states of a transaction ever need to be reproduced and that only a reasonable amount of information on the transaction's history must be maintained by the system.

With each state of a transaction we associate an index whose value is equal to the number of states preceding the given one in the transaction. Each state of a transaction will consist of:

1. an assignment to each local variable of the transaction of a value in its range,
2. an assignment of a value in its range to each of the global entities on which the transaction currently holds a lock when it reaches that state, and
3. the index of the state.

A transaction can either read or write a global entity with each atomic operation. Transactions which intend to execute only read operations on an entity must first perform either a shared-lock request (LS) or an exclusive-lock request (UX) for it. Transactions wishing to both read and write an entity must first perform an exclusive-lock request for it. A transaction which no longer requires access to an entity may release it by performing an unlock request for it, subject only to the two-phase requirement that no further
lock requests be executed after the unlock. The database management system responds to these operations according to the following rules.

1. It may grant the lock if no other transaction currently holds a lock on the entity requested.
2. Otherwise it must make the requesting transaction wait until the entity is available to be locked.
3. If response 2 results in a deadlock, the system must then roll back one of the involved transactions until the deadlock is broken.

A rollback operation is a response to a lock request by some transaction. A rollback of transaction T from state S2 to state S1 where S2 is the most recent state reached by T involves the following operations.

1. All entities on which T holds exclusive locks at S2 but not at S1 are restored to the values they had immediately before they were locked by T.
2. Locks held by T on all entities mentioned in 1 are released, as are all shared locks on entities held at S2 but not at S1.
3. All local variables are restored to their values at S1.
4. The state index (program counter) is reset to the value it had at S1.

We shall see how S1 may be chosen in succeeding sections.

Since we concern ourselves with rollback only as a means of deadlock removal, and since a transaction which has successfully performed all its lock operations cannot subsequently become involved in deadlocks, we assume that transactions are never rolled back after they have unlocked an entity in two-phase systems. It is clear that rollbacks do not interfere with the serializability of the two-phase protocol. We need not address the issue of serializability further.

3.0 DEADLOCKS AND ROLLBACKS

We now discuss the use of rollbacks to eliminate deadlocks in database systems with concurrent processing. We begin by defining the relation $\rightarrow$ on a concurrent set $T$ of transactions. If at a given time $t$, a transaction $T_4$ in $T$ is waiting to lock an entity $A$ which is locked by another transaction $T_j$, then we say $T_j \rightarrow T_4$. Define $T_j \rightarrow T_4$ if $A(T_j \rightarrow T_4)$. We define a concurrency graph $G_L(T)$ of $T$ at time $t$ to be a set $V$ of vertices in one-to-one correspondence with the transactions in $T$ such that a vertex corresponding to $T_j \in T$ has label $i$, and a set $A$ of arcs, such that each arc $\langle T_j, T_4 \rangle$ is included iff $T_j \rightarrow T_4$ at time $t$. A labeled concurrency graph $G_L(T) = \langle V, A \rangle$ has sets $V$ and $A$ defined as for $G_L(T)$ with a labeling function $f$ on a such that $f(a) = A$ if $a = \langle T_j, T_4 \rangle$ and $T_j A \rightarrow T_4$.

A deadlock is defined without loss of generality to be a subset $T^d = \{T_1, T_2, \ldots, T_k\}$ of $T$ such that at some time $t$, the transactions of $T^d$ form a cycle in $G_L(T)$.  

3.1 Exclusive Locks

In this section we consider systems which allow exclusive locks only. We have the following simple theorem.

**Theorem 1.** For a set $T$ of concurrent transactions, there is no deadlock in $T$ at time $t$ iff $G_L(T)$ is a forest.

Let a set $T^d = \{T_1, T_2, \ldots, T_k\}$ of transactions be deadlocked at time $t$. Then without loss of generality there exists in $G_L(T)$ a minimal cycle $T_1 A \rightarrow T_2 B \rightarrow \ldots \rightarrow T_k C \rightarrow T_1$ for some global entities $A, B, C, \ldots$. The deadlock may be broken by any operation which causes deletion of an arc or vertex from this cycle. This can be accomplished by rolling back any transaction in the cycle to a state in which it no longer holds a lock on an entity being waited for by another transaction in the cycle. We say that two transactions are in conflict over an entity whenever one of them has a lock on the entity and the other requests a lock on it. We say that the latter transaction causes the conflict by its request. Now, given that the system maintains $G_L(T)$ for the set $T$ of all transactions executing at any time $t$, then it is certainly possible to determine whether a wait response to a given conflict will result in deadlock. Indeed, given that $G_L(T)$ is acyclic at any time when deadlocks are not present, the system needs merely to check if the entity requested by the transaction $T_j$ which causes the conflict is already locked by a descendant of $T_j$ in $G_L(T)$. If so, then a wait response to $T_j$'s request must result in a cycle, and thus a deadlock. The system may choose to rollback any of the transactions involved in the deadlock in order to remove it. Given a definition of the cost of a rollback, this degree of freedom in choosing a transaction to remove from a deadlock introduces the possibility of choosing the optimal transaction for rollback.

Let us define the cost of a rollback of transaction $T_j$ involved in a deadlock. If $T_j$ is waiting in its own state $S_1$, and the state of highest index in $T_j$ in which $T_j$ does not hold a lock on an entity which would involve $T_j$ in a conflict with another transaction in the deadlock is $S_m$, then the cost of a rollback of $T_j$ is $S_m - S_1$. If the system maintains for each locked entity $A$ the index of the last state of every transaction $T_j$ which locks it in which $T_j$ does not hold a lock on $A$, then the system can easily compute this cost function for the rollback of any transaction in a deadlock. We may choose to rollback any of the transactions involved in the deadlock in order to remove it. Given a definition of the cost of a rollback, this degree of freedom in choosing a transaction to remove from a deadlock introduces the possibility of choosing the optimal transaction for rollback.

Let us define the cost of a rollback of transaction $T_j$ involved in a deadlock. If $T_j$ is waiting in its own state $S_1$, and the state of highest index in $T_j$ in which $T_j$ does not hold a lock on an entity which would involve $T_j$ in a conflict with another transaction in the deadlock is $S_m$, then the cost of a rollback of $T_j$ is $S_m - S_1$. If the system maintains for each locked entity $A$ the index of the last state of every transaction $T_j$ which locks it in which $T_j$ does not hold a lock on $A$, then the system can easily compute this cost function for the rollback of any transaction in a deadlock. We may choose to rollback any of the transactions involved in the deadlock in order to remove it. Given a definition of the cost of a rollback, this degree of freedom in choosing a transaction to remove from a deadlock introduces the possibility of choosing the optimal transaction for rollback.
lower index than assumed here.) The forest structure of $G_1(T)$ at any time when no deadlocks exist ensures that a wait response to a particular conflict can result in at most a single deadlock. It is then a simple matter for the system to traverse the resulting cycle in the graph in the event of deadlock, find all involved transactions and the cost of rolling each back, and make an optimal decision.

An example concurrency graph with the state indices of each transaction at the time it requested an entity added at the head and tail of that entity's arc is given in Figure 1 (a). Thus $T_2$ requested $b$ on the transition from its 8th state and requested $e$ from state 12, etc. Rollback of $T_2$ until it no longer holds a lock on $b$ will remove the deadlock, as will rollback of $T_3$ until it releases $c$ or $T_4$ until it releases $e$. The cost of a rollback of $T_2$ is $12-8=4$, of $T_3$ is $11-5=6$ and of $T_4$ is $15-10=5$, so $T_2$ is chosen for rollback. The result is shown in Figure 1 (b). Note that $T_1$ no longer waits for $T_2$ after the rollback.

Injudicious use of rollback can, while removing deadlocks, result in a set of transactions becoming involved in a situation in which each transaction in turn causes another transaction to be rolled back. Consider a system which uses the above optimization technique in the scenario represented in Figure 1 (b). If $T_1$, $T_5$, and $T_6$ subsequently execute to completion, and $T_3$ requests entity $f$ from its 14th state, while $T_2$ holds a lock on $f$ requested from its state 4 , we arrive at the scenario of Figure 2 (a). The system chooses $T_1$ for rollback, resulting in the situation of Figure 2 (b). This is the configuration of $T_2$, $T_3$, and $T_4$ which led to the deadlock of Figure 1 (a). Of course, $T_1$ could in the meantime have changed the value of $a$ after locking it, and $T_2$ may as a result not request a lock on $e$ as in Figure 1 (a). Thus it is by no means assured that a scenario which repeats some number of times will continue to do so in the presence of such outside interference, although it does have the potential to continue to occur indefinitely. We shall accordingly call this phenomenon potentially infinite mutual preemption.

A system clearly cannot exercise the full freedom of rollback optimization without risking potentially infinite mutual preemption. A general way of avoiding this is to place an ordering on the set of transactions concurrently in the system and use the ordering to determine which transactions to rollback. We may, for example, order transactions according to their time of entry into the executing environment as done in [7,10]. If a deadlock results from a conflict caused by $T_j$, we then find each transaction $T_i$ involved in the deadlock such that $T_j$ entered the system later than $T_i$ and rollback $T_i$ for which the operation will be least costly. More generally, we have the following theorem.

**Theorem 2:** Given a set of concurrent transactions $T$, if there exists a (time-invariant) partial order $\omega$ on $T$ such that $T_i$ is rolled back as a result of a conflict caused by $T_j$ iff $T_i \omega T_j$, then there will be no subset of $T$ involved in potentially infinite mutual preemption.

**Proof:** The result is obtained by a simple induction on $|T|$, which is omitted for the sake of brevity.

It should be pointed out here that the removal of deadlocks and potentially infinite mutual preemptions from a system does not suffice to guarantee that all concurrently executing transactions will terminate. For example, an unfair scheduling algorithm could result in some transactions never being allowed to lock entities they have requested. There are also some pathological situations such as a transaction which never halts because it must look at each entity in the system, and new entities are constantly being added by other transactions before it finishes. A treatment of such problems can be found in [7] and will not concern us here.

### 3.2 Shared And Exclusive Locks

We have seen that it is not generally necessary to roll a transaction back to its initial state in order to remove deadlocks from concurrent database systems allowing only exclusive locks. For systems which also allow shared locks, deadlocks may be eliminated in essentially the same way without always rolling transactions back to their initial state. The main difference in systems with both exclusive and shared locks is in the difficulty of optimization.

A conflict in such a system arises either

1. when a transaction requests a shared lock on an entity on which some other transaction holds an exclusive lock (Type 1), or
2. when a transaction requests an exclusive lock on an entity on which another transaction holds any lock (Type 2).

In the latter case, a wait response may cause a transaction to wait for more than one other transaction to unlock an entity. Thus $C(G(T))$ for a set of transactions in such a system will not generally be a forest in the absence of deadlock but rather a more general acyclic digraph. A single wait response to a lock request cannot then be guaranteed to result in at most a single cycle in the graph, but may in general result in the formation of an arbitrary number of cycles. Clearly, all of the cycles thus formed will include the vertex corresponding to the transaction which caused the conflict resulting in the cycles. Thus rollback of either the transaction which caused the conflict which led to deadlock, or of the transaction(s) conflicting with it, will suffice to remove the deadlock. Figure 3 shows examples of concurrency graphs for a system with exclusive and shared locks. In Figure 3 (a), we see a situation in which $T_3$ has requested an exclusive lock on entity $c$ on which $T_1$ and $T_2$ hold shared locks. $T_3$ has made a similar request on entity $d$. $T_2$ has either requested a shared lock on entity $a$ on which $T_1$ has an exclusive lock or else $T_2$ has requested an exclusive lock on $a$, and $T_3$ has a similar conflict with $T_2$ at entity $b$. Figure 3 (b) is the graph resulting either from $T_1$ requesting a shared lock on $e$ on which $T_2$ has an exclusive lock or from $T_2$, requesting an exclusive lock on $e$ when $T_3$ is the only other transaction holding any type of lock on $e$. Figure 3 (c) shows the result of an exclusive lock request by $T_1$ on entity $f$ on which $T_2$ and $T_3$ both hold shared locks. In the situations of Figure 3 (b) and 3 (c), multiple deadlocks result. All of which involve $T_1$, the transaction which created the deadlocks. In both cases, rollback of $T_1$ will remove all deadlocks. In Figure 3 (b), rollback of $T_2$ will also remove all deadlocks, while in 3 (c) both $T_2$ and $T_3$ would need to be rolled back if $T_1$ is not.

Optimization of deadlock removal in a system with shared and exclusive locks involves finding a set of transactions whose rollback will remove all cycles from the graph and the sum of whose rollback costs is minimal. This is equivalent to the graph theoretic problem of finding a minimum cost vertex cut set in a directed graph on whose vertices a cost function is defined and in which all cycles share at least one common vertex. A vertex cut set is a set of vertices whose removal from the graph breaks all cycles. Unfortunately, the problem appears to be NP-complete, as is the closely-related feedback vertex set problem. Gray [5] discusses a similar problem, with the same conclusion.

3.3 Distributed Systems

The methods of deadlock removal discussed here all require global information about the interactions among all the transactions in the system (namely the information contained in the concurrency graph) in order for the system to detect deadlocks. For centralized database systems, the maintenance of such global information is not especially difficult. For distributed systems, in which transactions process data at a number of different sites, the communications among sites required for the maintenance of such global data may make it impractical to do so. In such cases, deadlocks involving only a single site may be treated using the above means. However, the occurrence of deadlocks involving a number of sites cannot be detected by the system. Various methods, such as using timestamps or an a priori ordering of the sites to determine whether wait or rollback is used as a response to a given conflict, have been proposed to solve this problem [1,7,10]. These mechanisms in no way invalidate the advantages of rolling a transaction back to the latest possible state in which the conflict necessitating the rollback no longer exists. Thus partial rollback may be of use in distributed systems as well as centralized ones. It should be noted, however, that the implementations described in the following section will require the communication of database information between sites at any time that a transaction moves from one site to another than do methods using only total rollback.

4.0 IMPLEMENTATION OF ROLLBACK

Obviously, returning a transaction to a previous point in its execution requires that the system provide a way of erasing the effects on the database of all operations performed by the transaction between its current point of execution and the point to which it is rolled back. On the assumption that the system always knows the current state of a transaction, total rollback requires the system to effectively halt a transaction, restore it to its initial state, and restore all entities locked by the transaction to their values at the time they were locked before releasing them. Systems of two-phase transactions using total rollback have managed this by keeping local copies for each transaction of all entities on which the transaction holds exclusive locks at any given time. Any changes made to an entity are made to the local copy, the final value of which is then copied to the database when the transaction unlocks it. Thus, total rollback of a two-phase transaction involves simply releasing the locks it holds on any global entities and re-running it.
Maintaining the ability to roll a transaction back to an arbitrary preceding state generally requires more information than this. The system must have the ability to reproduce any of the states the transaction passed through in achieving its current state. If each operation (except READ or WRITE) performed by a transaction has a well-defined inverse, it may be possible for the system to actually "run a portion of the transaction backwards" as it were, erasing its effects as it goes. Some advantages of a similar approach are discussed in Schlageter [8]. Such methods require a system knowledge of transaction semantics however, and thus fall outside the purview of this paper. Without semantic information, we are forced to employ methods involving multiple copies of updated entities.

We have seen in the previous section some examples of ways in which rollback can be used in deadlock avoidance. In these situations, it was required that a transaction be rolled back to a point in its execution in which it has not referenced particular entity. In order to minimize the loss of progress in the execution of the rolled-back transaction, it is desirable that the transaction be rolled back to the latest state in its execution in which this is true. In general, this will mean rolling a transaction back to the state immediately preceding its execution of a lock request. We shall call such a state a lock state, and we shall assume henceforth that transactions are only rolled back to lock states.

It will be convenient from now on to consider a transaction to be a sequence of lock states, since rollback will always be initiated from a lock state to another lock state. Now, when a transaction T_i locks an entity A, it knows its value at that time. Let us call this value the global value of A. We assume that the global value of an entity does not change until the transaction unfolds it. While T_i has A locked, it makes a number of reads and writes to one or more local copies of A. The final value of the latest such copy becomes the new global value when T_i unlocks A. Recall that by our earlier assumptions, T_i is two-phase and is never rolled back after it has unlocked an entity.

There is a one-to-one correspondence between lock states and entities locked by a transaction. We can envision a system which associates a stack with the lock state which immediately precedes the execution of an exclusive lock request on each entity A by a transaction T_i. The stack will hold the sequence of values assigned to A by T_i during its execution. Each stack element has two fields, a value field and an index field. When A is locked by T_i, its global value is pushed onto the stack along with an associated index.

Note that only the value produced by the last write operation performed on a global entity or an element of L_i before a given lock state will occur as part of that lock state. Thus, if transactions are always rolled back to lock states, we need only keep separate copies of the final values of each entity for each lock state. Let us define the lock index of an entity or an operation to be equal to the number of lock states preceding it in the transaction. Let the system place the current lock index in the value field of each new element pushed onto a stack. The system then pushes a new element onto the stack for a given lock state iff the lock index of the write operation producing the new value of the entity is greater than the lock index of the stack. Otherwise, the two indices must be equal, in which case the value field of the current top element in the stack is updated. When an entity A is unlocked by T_i, the top of the stack is copied as the new global value of A and the stack is returned to free storage.

A similar stack is created for each element L_i of L_i, the set of local variables of T_i. Each such local stack is initialized to contain an element whose value is the initial value of L_i and whose index is 0. We also assign fixed indices to the stacks themselves when they are created. Each stack for a global entity is assigned an index equal to the lock index of the state it is associated with. Each stack for a local variable is assigned index 0. Now we can implement a rollback of transaction T_i from the current lock state S_q with lock index j, to previous lock state S_p with lock index k, k<j, as follows.

1. Wait the transaction.
2. Delete each stack with lock index \(j \leq k\).
3. For each stack with lock index \(j < k\), pop each element with lock index \(j\).
4. For each lock request of lock index \(m\) such that \(k < j\), unlock the entity requested, but do not update its global value if it was exclusive-locked.
5. Set the current state index to \(q\) (and change the program counter accordingly).

We shall refer to this implementation of rollback as the multi-lock copy strategy (MCS). The main drawback to this simple approach is that the space required in the worst case is proportional to the square of the number of entities locked.

**Theorem 2:** Let \(n\) be the number of global entities on which a transaction T_i holds locks at a given time. There can be at most \(n(n+1)/2\) local copies of global entities and \(nL_i\) copies of local variables associated with T_i using MCS.

**Proof:** Let j be the lock index of a stack for L_i. The stack can hold at most \(n\) elements since at most one element per subsequent lock request can have been placed on the stack since its creation. Since local variables have 0 for their stack index, the maximum number of copies of each local variable is \(n\) for a total of \(nL_i\) copies of all local variables. The maximum number of copies of global entities occurs if all such entities are held with exclusive locks. In this case the maximum possible total of all copies of all global entities is \(\sum_{j=0}^{n-1}(n-j)^2 = n(n+1)/2\).
Limitations in this space overhead can be achieved at the expense of the ability to limit transaction rollbacks to the minimum loss of progress consistent with deadlock avoidance. This is, of course, clearly evident in the extreme case of systems using only total rollback and needing only one local copy of each entity. We present a less extreme approach which also requires only one local copy of each entity.

We have already made the assumption that the global value of a global entity is maintained in the database unmodified while a transaction operates on local copies of the entity. Suppose the first write operation by transaction T to a global entity E occurs immediately after state S_j, and suppose that state S_j is the lock state for E. At all states between S_j and S_i, the local value of E is the same as its global value, since it has not yet been updated. If a rollback is performed on T to a state S_k such that j < k < i, then the system can restore the value of E at S_k, regardless of how many times the value of E has subsequently been modified since it knows the global value of E. We define a state to be the last state before the first write operation in entity E a restorable state for E. All other states are non-restorable. Likewise, we define a transaction to be well-defined if it has a last lock state preceding the index of restorability of E or L. We assume for convenience that no write operations occur before the first lock request in a transaction.

We can now define a state-dependency graph for a transaction T_i at lock state S_p in its execution, which we shall call G_p = <V,E>. V is the set of vertices corresponding to the lock states of T_i of index less than or equal to p, with each vertex labeled with the lock index of its corresponding lock state, and E is a set of (undirected) edges {v_1,v_2} such that either

1. the labels of v_1 and v_2 differ by 1, or
2. a write operation following a state with lock index equal to the label of v_2 occurs to an entity whose index of restorability is equal to the label of v_1.

We will let A(v) denote the label of a vertex v. We say that state S_j of a transaction T is undefined at state S_i if when T has reached S_j, S_i is non-restorable for some entity E. Otherwise, S_j is said to be well-defined at S_i. We now show a necessary and sufficient condition for a vertex of a state dependency graph to represent a well-defined state of a transaction.

**Theorem 4:** A lock state S_n with lock index q is undefined at lock state S_p of transaction T_j iff in G_p there exists an edge {v_r,v_s} such that A(v_r) = q < A(v_s).

**Proof:**

If:

Such an edge must have endpoints which differ by more than 1, and thus have been formed by condition (2) in the definition of G_p. In such case, (v_r) is the index of restorability of some entity E, or local variable L, and a write operation has occurred to E or L since S_n. Thus the value of E or L at S_n has been destroyed by the time state S_n was reached, so S_p is not restorable for E and is thus undefined.

Only if:

Assume S_p is undefined. Then there must exist entity E for which S_p is not restorable. By definition, if u is the index of restorability of E, then u < q. Also by definition, a write operation to E immediately preceded by a state of lock index u or less must have occurred to make S_p unrestorable for E. Thus there will exist in G_p vertices v_u and v_q such that A(v_u) = u and A(v_q) = q, along with edge {v_u,v_q}.

**Corollary 1:** A lock state S_q such that 0 < q < p is well-defined at state S_p in the execution of transaction T_i iff vertex v_q in G_p has a label such that A(v_q) = q < A(v_s) for all vertices v_s in G_p. Also, the vertex v_q is an articulation point in G_p when and only when there is no path in G_p from v_q to any vertex v_s such that A(v_s) = q.

**Corollary 2:** Assume that, in G_p, there exist edges {(v_1,v_2),...,(v_n,v_m)} such that

A(v_1) < A(v_2) < ... < A(v_n)

Construct G_p by deleting all edges listed above except (v_x,v_y). A vertex v is an articulation point of G_p if for some transaction T_i with the corresponding states, lock states, and lock indices indicated above the transaction steps, Figure 4(b) contains the state-dependency graph G_p for T_i. Figure 4(c) shows the corresponding C_20. Note that there are no articulation points in either graph, so the only well-defined states are the trivial ones with lock index 0 or lock index 6. If we delete the operation C_i < K from T_i, the resulting transaction T_i has C_i at state S_j corresponding to S_20 of T_i as shown in Figure 4(d). Thus lock state S_13, with lock index 4, is well-defined. We could rollback T_i from S_19 to S_13 by simply releasing the locks held by T_i on entities E and F and resetting the state (and program counter) of T_i.

Thus we see that the problem of determining whether we can restore a transaction to a particular previous lock state of index q when it
has reached lock state $S_8$ can be reduced to that of finding the articulation points of a graph with vertices of degree $\leq 6$ and checking whether one of them corresponds to state $S_8$, since obviously only well-defined states can be recreated. Upon the addition of any edge to a state dependency graph, the system need merely delete all articulation points of higher index than either endpoint of the edge in order to maintain the graph. Articulation points are added for each new lock-state. The overhead in maintaining a state dependency graph is clearly very low.

![Diagram](image.png)

**Figure 4**

In cases where allowing a transaction $T_i$ to wait for an entity $E$ would result in deadlock, in this implementation it is only possible to roll a transaction back to its lock state for $E$ if that state is well-defined. Otherwise, we must find the well-defined lock state of largest index less than that of the lock state for $E$, and roll the transaction back to that state. Thus, we find that rollback can be implemented in the absence of lock states and without the tremendous overhead required by the previous systems, but we lose some ability to minimize the extent of the rollback. On the other hand, we would expect that generally this method would cause substantially less loss of progress to transactions than would the exclusive use of total rollback, but with the added overhead of requiring system monitoring of all write operations to both local variables and global entities.

### 5.0 Conclusions

We have seen that in a database system allowing only two-phase transactions, the ability to rollback a transaction to a point where a particular global entity is not locked by that transaction is required if the system is to guarantee deadlock-freedom. The usual system strategy employed for this purpose is to totally remove a transaction from the system and restart it. This is the extreme special case of a more general operation which we have defined here.

We have demonstrated implementations which allow us to minimize the extent to which we need to rollback a transaction to a greater or lesser degree, but which extract penalties of varying severity in terms of system overhead. Of particular interest is our final implementation based on state-dependency graphs, which allows much less drastic rollbacks than the total removal method in many cases, yet requires no great increase in system overhead.

![Diagram](image.png)

**Figure 5**

It is clear that the performance of such a system depends heavily on the structure of the two-phase transactions running in it. For example, consider transaction $T_4$ of Figure 5 (a), in which the same operations are performed as in $T_4$ of Figure 4 (a) but in a different order, along with the state-dependency graph of $T_4$ in Figure 5 (b). We notice that the number of well-defined states is much higher for $T_4$ than for $T_5$, which indicates that rollbacks need not proceed as often beyond the minimum extent necessary to avoid a deadlock.
Systems in which transactions similar to $T_2$ execute concurrently will be much more efficient in removing deadlocks than will systems with transactions similar to $T_1$.

The property of $T_1$ that makes it more efficient is the clustering of the write operations for each entity. That is, there are as few lock states as possible between successive write operations to a given entity, thus minimizing the number of undefined states caused by these writes. Transactions intended to run efficiently in a system using implementations similar to those illustrated here should incorporate this principle as much as possible. Note that this strategy is also efficient for the MCS implementation as it minimizes the number of copies that must be kept.

A second method for increasing system efficiency by properly structuring transactions becomes available if we assume that all two-phase transactions declare in some way to the system the execution of their last lock request. Since we assume rollback is performed only in response to lock requests which lead to deadlock, the system knows upon receiving such a declaration that the declaring transaction will not be rolled back henceforth, and may cease monitoring it. If a transaction waits to perform write operations to any entity until after it performs its last lock request, then these operations will have no adverse effects on system performance. Transactions in which this rule is strictly enforced will have three distinct phases: an acquisition phase where entities are only locked, an update phase, and a release phase.

These relationships between the structure of transactions and their efficiency in a database system using our rollback implementations raise interesting possibilities for the optimization of transactions intended to run in such systems, perhaps at the time of their compilation. Such possibilities are outside the scope of this paper and must await further research.

Let us note finally that the state-dependency graph implementation of partial rollback can easily be extended to allow more than one local copy to be kept for entities. The problem of determining how to allocate a bounded amount of extra storage to the entities in order to maximize the number of well-defined states in such systems remains another interesting question for further study.

6.0 REFERENCES


