

AN EFFICIENT DEADLOCK REMOVAL SCHEME FOR
NON-TWO-PHASE LOCKING PROTOCOLS*

Z. KEDEM¹
C. MOHAN²
A. SILBERSCHATZ³

ABSTRACT

Over the years several locking protocols have been proposed for coordinating the concurrent use of a data base by multiple transactions. Of these, the non-two-phase locking (non-2PL) protocols form a large class. The Pitfall Protocol (PP) is one of the non-2PL protocols. While the rules of PP assure serializability, they do not prevent deadlocks from occurring. Resolution of a deadlock by partially/fully undoing (i.e. rolling back) the actions of one of the transactions involved in the deadlock may result in two undesirable consequences: a) cascading rollbacks -- more than one transaction may have to be rolled back, b) rollback of completed transaction -- a transaction that has terminated could be required to be rolled back. Thus a complex commit protocol may be necessary to determine whether a transaction may be allowed to commit.

It is the goal of this paper to introduce a simple additional condition to the rules of PP that will allow very simple handling of deadlocks by partial rollbacks, without causing the above undesirable effects.

1. INTRODUCTION

When a data base is accessed and updated concurrently by a number of asynchronously running transactions it could lead to inconsistencies in the stored/retrieved data if such operations are not regulated. A common

*This research was partially supported by NSF Grant Numbers MCS 80-25376, 81-04017, 81-04886, 81-10097, by ONR Contract N00014-80-K-0987, and by DARPA Contract N00039-82-C-0427.

¹Department of Computer Science, Columbia University, New York, NY, 10027, on leave from Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794.

²IBM Research Laboratory, K55/61F, San Jose, CA 95193.

³Department of Computer Sciences, The University of Texas, Austin, TX 78712.

approach to dealing with this problem is to define a transaction as a unit that preserves consistency (e.g., it is assumed that each transaction, when executed alone, transforms a consistent state into a new consistent state), and require that the outcome of processing a set of transactions concurrently be the same as the one produced by running these transactions one-at-a-time (i.e. serially) in some order. A system that ensures this property is said to be serializable [2]. Associated with concurrent access to data is the problem of deadlocks. Deadlocks arise as a result of circular wait conditions involving two or more transactions. A system which does not allow deadlocks to occur is said to be deadlock-free [1].

Serializability can be ensured via a number of concurrency control mechanisms, the most common one being a locking protocol. Such a protocol can be simply viewed as a restriction on when a transaction may lock and unlock each of the data base items. Locking a data item inhibits certain types of concurrent activity on that item until the lock is released. The first useful locking protocol developed was the two-phase locking (2PL) protocol [2], which is characterized by the fact that a transaction is not allowed to lock a data base item after it has unlocked any other item.

Recently a number of non-2PL protocols have been defined for data bases in which a pattern of allowable accesses by a transaction is described by a directed acyclic graph structure superimposed on the data items [3, 6, 7, 8, 10, 11, 13]. One of the most general non-2PL protocols is the pitfall protocol (PP) presented in [7]. It is a versatile protocol in the sense that several previously developed protocols (e.g. the tree protocol [10], the majority protocol [6], and the DAG protocol [13]) are special cases of it. While PP assures serializability, it does not assure freedom from deadlocks.

One approach to dealing with deadlocks is to prevent them by, for instance, enumerating the vertices of the graph (a topological sort) and requiring the transactions to acquire locks on the vertices in the increasing order of their enumeration. Another approach is to allow deadlocks to occur and then take corrective

action. Each approach has its own advantages and disadvantages [1]. Here we are concerned with only the latter approach.

A number of schemes have been proposed for recovering from deadlocks in general [4, 5, 9, 12]. These schemes usually require one or more transactions to be rolled back (i.e., the effects of the transactions to be "undone"). Once a transaction has been chosen for being rolled back it could be rolled back completely (all the actions of the transaction are "undone"), or partially (the transaction is rolled back only to the point in its execution where the deadlock cycle is broken). The notion of partial rollback was introduced in [4], in the context of the 2PL protocol.

Rolling back one transaction may force some other transactions which were dependent on the former also to be rolled back. This phenomenon is called cascading rollback. Clearly cascading rollbacks are expensive to deal with and hence should be avoided whenever possible. The advantages of avoiding cascading rollbacks are:

1. Only one transaction needs to be rolled back to resolve any deadlock. This leads to a decrease in the amount of partial transaction executions that are repeated.
2. More importantly, transaction executions need not be monitored to keep track of transaction dependencies (information like which transaction read which transactions' output). In systems where cascading rollbacks are inevitable, such information is necessary to determine, when a decision is made to rollback a particular transaction, what other transactions must also be rolled back.

If cascading rollbacks are not avoided then:

1. An increase in the delay between the time at which a deadlock occurs and the time at which it is detected could potentially lead to an increase in the number of transactions that need to be rolled back.
2. Even terminated transactions may have to be rolled back, if cascading rollbacks were to be necessary. This may introduce considerable complexity into the commit stage as a transaction must not be allowed to commit, if it has to be rolled back later.

In general, transaction failures/abortions may also necessitate rollbacks. In this paper however, we restrict our attention only to the

problem of cascading rollbacks due to the occurrence of deadlocks. It is interesting to note that while the 2PL protocol does not assure deadlock-freedom it does however guard against cascading rollbacks. Unfortunately this is not the case with the Pitfall Protocol. We thus propose a condition (called the Super Pitfall Condition) which, when adhered to by all transactions following PP, will guarantee that no cascading rollbacks may result.

2. THE PITFALL PROTOCOL

Let the data base be organized in the form of a directed acyclic graph (DAG) with each vertex (in the set of vertices V) being a data item. PP supports the X and S modes of locking. An X mode lock on a data item allows read and write access to the item, while an S mode lock permits only read access to the item. A request to lock an entry in X (S) mode is accomplished through the LX (LS) instruction. Before presenting the protocol a few definitions are in order.

Definition 2-1: A history H is the trace, in chronological order, of the concurrent execution of a set of transactions $\bar{T} = \{T_0, \dots, T_{N-1}\}$.

Definition 2-2: We define the $<_e$ and $<$ relation on a history H of a set $\bar{T}$ of transactions as follows:

$$T_i <_e T_j \iff T_i \text{ held an } M_i \text{ mode lock on } e \text{ and } T_j \text{ held an } M_j \text{ mode lock on } e \text{ later, where } X \in \{M_i, M_j\}.$$

$$T_i < T_j \iff \nexists e [T_i <_e T_j]$$

If T_i and T_j are related via the $<$ relation then we shall say that T_i conflicts with T_j .

Lemma 2-1: A protocol assures serializability if for all concurrent executions of transactions following it the associated relation $<$ on $\bar{T}$ is acyclic.

Definition 2-3: We define the $\rightarrow_e$ and $\rightarrow$ relations (the "wait-for" relations) as follows:

$$T_i \rightarrow_e T_j \iff T_j \text{ is currently holding an } M_j \text{ mode lock on } e \text{ and } T_i \text{ has requested an } M_i \text{ mode lock on } e, \text{ where } X \in \{M_i, M_j\}.$$

$$T_i \rightarrow T_j \iff \nexists e [T_i \rightarrow_e T_j]$$

Note that the relation $\rightarrow$ is time dependent.

Lemma 2-2: A protocol assures deadlock-freedom if for all concurrent executions of transactions following it the associated relation $\rightarrow$ on T is acyclic, at any instance of time during the execution.

Definition 2-4: Let $L(T_i)$ be the set of all vertices for which T_i issued a locking instruction, $LX(T_i)$ be the set of vertices for which T_i issued the instruction to lock in X mode, and $LS(T_i)$ be the set of vertices for which T_i issued the instruction to lock in S mode.

Definition 2-5: For each transaction T_i , certain subsets of $L(T_i)$, called pitfalls, are defined as follows. Consider the subgraph spanned by the set $LS(T_i)$. Generally it splits into a number of connected components, say $A_1, A_2, \dots, A_k$. A pitfall of T_i is defined as a set of the form $A_j \cup \{v \in LX(T_i) \mid v \text{ is a neighbor of some } w \in A_j\}$.

Note that the pitfalls are not necessarily disjoint and that the pitfalls of a given transaction in no way depend upon the locking activities of other transactions.

Definition 2-6: A transaction T_i is said to be two-phase on a set of entities $K \subseteq L(T_i)$ if and only if the subsequence of T_i obtained by deleting all instructions referring to entities (vertices) in $L(T_i) - K$ is two-phase.

Definition 2-7: We shall say that a graph is a guarded graph if and only if with each $v \in V$ (except the roots of the DAG) we associate a non-empty set of pairs:

$$\text{guard}(v) = \{\langle A_1^V, B_1^V \rangle, \dots, \langle A_M^V, B_M^V \rangle\}$$

satisfying the conditions:

1. $\emptyset \neq B_1^V \subseteq A_1^V \subseteq V$,
2. $\forall u \in A_1^V [u \text{ is a parent of } v]$,

3. if $A_i^V \cap B_j^V = \emptyset$, then there exists no biconnected component** of the DAG that includes vertices from both A_i^V and B_j^V .

The rules of the pitfall protocol for each transaction are:

1. The first lock may be acquired on any vertex.
2. Subsequently, a vertex v can be locked only if locks are held on all vertices in some B_1^V (thus $M_V > 0$) and if all vertices in the corresponding $A_1^V - B_1^V$ have been locked (and possibly unlocked).
3. The transaction is two-phase on each of its pitfalls (it does not require a transaction be two-phase on the union of its pitfalls).

Theorem 2-1: The pitfall protocol assures serializability.

Proof: See [7].

We show now that not only is the protocol deadlock-prone, but that to handle deadlocks is quite a complex task. Consider the database graph of Figure 2-1,

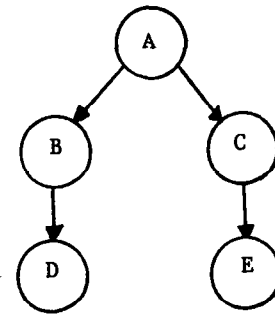


Figure 2-1

where the guards are defined by:

**We remind the reader that a biconnected component of a graph consists of a maximal collection of vertices $v_1, v_2, \dots, v_p$, for $p \geq 3$, such that for any pair $\{v_i, v_j\}$, there exist two (undirected) chains between v_i and v_j which are vertex-disjoint other than at the ends.

V	guard (u)
A	$\emptyset$
B	$\{\langle A \rangle, \{A\}\}$
C	$\{\langle A \rangle, \{A\}\}$
D	$\{\langle B \rangle, \{B\}\}$
E	$\{\langle C \rangle, \{C\}\}$

Note that the guard simply define the tree protocol as described in [10] with S locks allowed.

Consider the (partial) history defined by the table:

<u>T1</u>	<u>T2</u>	<u>T3</u>	<u>T4</u>
LS A			
LX B			
LX D			
UN D			
	LX D		
	UN D		
		LS A	
		LX C	
		LX E	
		UN E	
			LX E
			UN E
LX C		LX B	

All the transactions followed the PP, but at this point in time we have the following cycle:

$T_1 \rightarrow_C T_3 \rightarrow_B T_1$

which implies the existence of a deadlock.

To resolve this deadlock, either T_1 or T_3 must be rolled back. Since the situation is symmetric, assume T_1 is rolled back so that T_3 can proceed. T_1 must be rolled back to a point in time where the only locking instruction executed by it was LS A. Since transaction T_2 depends on T_1 (T_2 might have read the item D which was possibly modified by T_1), it must also be rolled back. At this point in time however, T_2 might have already terminated. Hence not only do we have to worry about cascading rollbacks but we must make sure that transactions do not terminate (commit) too early.

3. THE SUPERPITFALL PROTOCOL

Let p be a pitfall of some transaction T . Then r_p denotes the set of root vertices of the subgraph spanned by the vertices of p and nr_p denotes the set of the other (non-root) vertices of the subgraph.

The instructions of T contain two subsequences defined by the pitfall p :

1. Initial Phase (IP): This phase begins at the point at which the first of the vertices in r_p is locked and ends at the point at which the last of the vertices in r_p is locked.
2. Final Phase (FP): This phase begins at the point at which the first of the vertices in nr_p is locked and ends at the point at which the last of the vertices in nr_p is locked.

Note that the instructions in the two phases may refer to vertices outside the pitfall, and furthermore, the two phases may overlap. To illustrate the latter point, consider the database graph of Figure 3-1, and the following segment of a transaction T following the majority protocol*** defined in [8]:

LX R
LS B
LX C
LS E
LX D
LS F
UN X
:
:

For transaction T we have the following pitfalls:

$p1 = \{R, B, C, D\}$

$p2 = \{C, D, E, F\}$

For $p2$ the IP has to include the locking of the vertices C and D, and the FP has to include the locking of the vertices E and F. The two phases overlap due to the acquisition of the lock on E preceding the acquisition of the lock on D.

Definition 3-1: We shall say that a transaction follows the superpitfall protocol if it follows the pitfall protocol and in addition,

*** A majority protocol is a special case of the PP for which the $\text{guard}(v) = \{\langle A, A \rangle \mid A \text{ is a majority of fathers of } v \text{ in some biconnected component.}\}$

4. No unlocking instructions are issued in the IP and the FP stages of any pitfall.

Note that unlocks may occur between IP and FP (if they do not overlap).

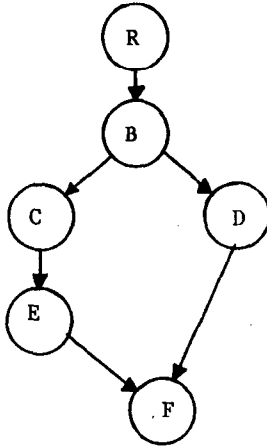


Figure 3-1

Theorem 3-1: Let there be a set of transactions following the Superpitfall protocol and assume that a deadlock (cycle) of the following form has occurred:

$$T_0 \rightarrow u_1 T_1 \rightarrow u_2 \dots \rightarrow u_{n-1} T_{n-1} \rightarrow u_0 T_0$$

Then for any i ,

1. T_i can be rolled back to the point at which it issued the instruction to lock u_{i-1} ,
2. No other transaction T_j , ($j \neq i$) needs to be rolled back,
3. T_j will be able later to continue its execution.

Proof: In order to aid in intuitive understanding, we will prove the theorem only for the case where the database graph is a (directed) tree (i.e., for each vertex v , $\text{guard}(v) = \{\langle w \rangle, \{w\} \mid w \text{ is a parent of } v\}$). We emphasize however that our results apply to DAGs as well. We will also break the proof into a number of steps.

Before proceeding with the proof let us briefly elaborate on statements (1) - (3) above.

Statements (1) and (2) will allow us to pick any transaction to be rolled back, thus allowing the system to make "progress". Indeed, if the "youngest" transactions are chosen for rollback, some older transactions will progress. (For related issues see also [9, 12].)

Statement (3) is not trivial. One can imagine the case that a transaction is rolled back so far that it no longer holds enough elements in the appropriate guards so it can lock the vertices it needs to access. Fortunately, this will not be the case.

We will assume that the cycle is minimal under the relation $\rightarrow$ as any cycle can be broken by successive breaking of minimal cycles.

Lemma 3-1: If a set of transactions is involved in a cycle of the following form:

$$T_0 \rightarrow u_1 T_1 \rightarrow u_2 \dots \rightarrow u_{n-1} T_{n-1} \rightarrow u_0 T_0,$$

then in the partial history, obtained by considering the instruction up to the point when the deadlock occurred,

$$\forall i \forall j [T_i \prec T_j \text{ and } T_j \not\prec T_i].$$

Proof: Similar to a proof of Theorem 11 in [13].

Definition 3-2: We will define the following two sets:

$$Q(T_i) = \{v \mid v \text{ was locked (and possibly unlocked) by } T_i \text{ before it issued the instruction to lock } u_{i+1}\}$$

$$R(T_i) = Q(T_i) \cup \{u_{i+1}\}.$$

Lemma 3-2: For any i , either u_i and u_{i+1} are in the same pitfall, or u_i is a parent of u_{i+1} .

Proof: Note that $\{u_i, u_{i+1}\} \subseteq Q(T_i) \cup \{u_{i+1}\}$ and that $\{u_i, u_{i+1}\} \subseteq \{u_i\} \cup (\bigcup_{j \neq i} Q(T_j))$. As we assume that the underlying graph is a tree, it follows that there is a unique chain, say $u_j = x_0, x_1, \dots, x_q = u_{j+1}$, $q \geq 1$, between u_j and u_{j+1} in the graph. As both $Q(T_i)$ and $\bigcup_{j \neq i} Q(T_j)$ are connected, it follows that

$$\{x_1, \dots, x_{q-1}\} \subseteq Q(T_i) \cap (\bigcup_{j \neq i} Q(T_j)).$$

If $q=1$ (and $\{x_1, \dots, x_{q-1}\} = \emptyset$), then by the rules of the protocol u_i is a parent of u_{i+1} .

If $q>1$, it follows that each vertex between u_i and u_{i+1} in the chain was locked by both T_i and some transaction T_j , $j \in \{0, 1, \dots, i-1, i+1, \dots, n-1\}$. By Lemma 3-1, it

follows that it was locked by T_i only in the S mode, and thus if $q > 1$, $\{u_i, u_{i+1}\}$ was a subset of a single pitfall of T_i .

Any pitfall is a rooted directed tree. For a pitfall p , we shall denote by $r(p)$ its root.

Lemma 3-3: For any transaction T , and any pitfall of p , (at least) one of the following holds:

1. $r(p)$ is the first element locked by T ,
2. $r(p) \in LX(T)$.

Proof: Assume that (1) does not hold. Then it is a non-trivial descendant of the first vertex locked by T , and thus the parent of $r(p)$, say w , is in $L(T)$. Now as $w \notin p$, and some child of $r(p)$ is in $LS(T)$, it follows that $p \in LX(T)$.

Lemma 3-4: $\{u_i, u_{i+1}\}$ are a subset of some single pitfall p of T_i , and one of the two cases hold:

1. $u_i \neq r(p_i)$,
2. $r(p)$ is the first vertex locked by T_i .

Proof: The following are the possible cases to be considered.

- (a) $\{u_i, u_{i+1}\}$ is a subset of some pitfall p and $r(p)$ is not the first vertex locked by T_i .
- (b) $\{u_i, u_{i+1}\}$ is a subset of some pitfall p and $r(p)$ is the first vertex locked by T_i .
- (c) $\{u_i, u_{i+1}\}$ is not a subset of any pitfall of T_i .

Assume that case (c) holds. Then u_i is a parent of u_{i+1} and was locked by T_i in X mode. As T_{i-1} is attempting to lock u_i , it follows that no vertex locked so far by T_{i-1} (including u_{i-1}) lies in the partial tree rooted at u_i and consisting of the descendants of u_i . It easily follows (see Figure 3-2) that any chain between

u_{i-1} and u_i must include u_{i+1} . As $\{u_{i-1}, u_{i+1}\}$ is a subset of the connected set $\bigcup_{j \neq i} Q(T_j)$, it follows by Lemma 3-1 that $u_i \notin LX(T_i)$, showing that case (c) cannot hold.

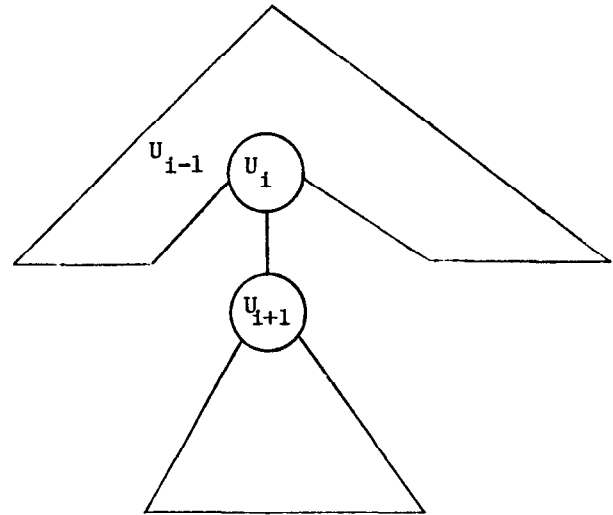


Figure 3.2

We have thus shown that $\{u_i, u_{i+1}\}$ is a subset of some pitfall p . Assume that case (b) does not hold. Then by Lemma 3-3, $r(p) \in LX(T_i)$. Assume by contradiction that $u_i = r(p)$. We see that u_{i+1} is a descendant of u_i and that any chain between u_{i-1} and u_{i+1} must include u_i . This is shown to be impossible similar to case (c).

Proof of the Theorem: We now show that T_i can be rolled back to the point before the instruction to lock u_i was issued (the rollback point). We consider the two cases of Lemma 3-4.

- (1) Since $u_i \neq r(p)$, it follows that no unlock instructions were issued by T_i between the rollback point and the deadlock point. Furthermore T_i was holding a lock on the parent of u_i at the rollback point. Thus it will be able to restore correct data to all the vertices whose value depends on the values in the vertices locked between the rollback and the deadlock points, and then proceed following the protocol, when allowed by the concurrency control.

- (2) In this case no unlock instructions were issued at all by T_1 . The original values in $Q(T_1)$ can be restored and the transaction removed from the system. It can then be restarted at the convenience of the concurrency control.

We complete our discussion concerning the superpitfall protocol by noting the following:

1. Rollbacks can be done in such a way that progress is made. For example, we can always let the oldest transaction (in the overall priority) proceed whenever a deadlock occurs.
2. Returning to the example of Figure 2-1, let us see when the two transactions T_1 and T_2 violated condition (4) of the Superpitfall Protocol. For T_1 , the vertices R, C and A form a pitfall, say p_1 . T_1 violated condition (4) when it unlocked B during the FP of p_1 . It should be easy to see how T_2 violated condition (4).

4. CONCLUSION

After pointing out that the problem of cascading rollbacks due to deadlocks could arise only in non-2PL protocols, we discussed the advantages of avoiding cascading rollbacks. We described a non-2PL protocol (namely the Pitfall Protocol (PP)) and illustrated a deadlock situation requiring cascading rollbacks. Next we presented the Superpitfall protocol and proved that when all transactions follow the protocol the need for cascading rollbacks will not arise. While we have shown how to avoid cascading rollbacks and thereby accrue some advantages it should be recognized that there is a cost associated with the condition. There is potential for a decrease in the level of concurrency supported when the condition is adopted since the transactions may be forced to hold locks for a longer time than was necessary before.

5. REFERENCES

1. Coffman, E., Elphich, M., Shoshani, A., "System Deadlocks," Computing Surveys 2, 3 (June 1971), 67-78.
2. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L., "The Notions of Consistency and Predicate Locks in a Database System," CACM 10, 11 (November 1976), 624-723.

3. Fussell, D.S., Kedem Z., Silberschatz, A., "A Theory of Correct Protocols for Database Systems," Proc. Seventh International Conference on Very Large Data Bases, September 1981.
4. Fussell, D., Kedem, Z., Silberschatz, A., "Deadlock Removal Using Partial Rollback in Database Systems," Proc. ACM-SIGMOD International Conference on Management of Data, May 1981.
5. Gray, J., "Notes on Data Base Operating Systems," Operating Systems, Lecture Notes in Computer Science: Volume 60, Springer-Verlag, 1978.
6. Kedem, Z., Silberschatz, A., "Controlling Concurrency Using Locking Protocols," Proc. Twentieth IEEE Symposium on Foundations of Computer Science, October 1979.
7. Kedem, Z. and Silberschatz, A., Locking protocols: from exclusive to shared locks, University of Texas, Technical Report, 1980.
8. Kedem, Z., Silberschatz, A., "Non-Two-Phase Locking Protocols with Shared and Exclusive Locks," Proc. Sixth International Conference on Very Large Data Bases, October 1980.
9. Rosenkrantz, D.J., Stearns, R., Lewis, P.M., "System Level Concurrency Control for Distributed Database Systems," ACM Transactions on Data Base Systems 3, 2 (June 1978), 178-198.
10. Silberschatz, A., Kedem, Z., "Consistency in Hierarchical Database Systems," JACM 27, 1 (January 1980), 72-80.
11. Silberschatz, A., Kedem, Z., "A Family of Locking Protocols for Database Systems that are Modeled by Directed Graphs," IEEE Transactions on Software Engineering (to appear).
12. Stearns, R., Lewis, P.M., Rosenkrantz, D.J., "Concurrency Control for Database Systems," Proc. Seventeenth IEEE Symposium on Foundations of Computer Science, October 1976.
13. Yannakakis, M., Papadimitriou, C., Kung, H.T., "Locking Protocols: Safety and Freedom from Deadlocks," Proc. Twentieth IEEE Symposium on Foundations of Computer Science, October 1979.