Deadlock Detection is Cheap

Rakesh Agrawal [*]
Michael J. Carey [+]
David J. DeWitt [*]

[*] Computer Sciences Department, University of Wisconsin, Madison
[+] Computer Sciences Department, University of California, Berkeley

This research was partially supported by the National Science Foundation under grant MCS82-01870, the Department of Energy under contract #DE-AC02-81ER10920, the Air Force Office of Scientific Research under grant AFOSR-78-3596, and Naval Electronic Systems Command under contract NESC-N00039-81-C-0569.

Abstract. *Deadlock detection is usually considered to be expensive, and timeouts or deadlock prevention techniques are usually resorted to as a result, which many times causes unnecessary transaction restarts. In this paper, we show that under certain reasonable assumptions, deadlocks can be detected very cheaply.*

1. Introduction

Deadlocks have been a topic of active interest during the past several years (see [9] for an annotated bibliography). They are typically characterized in terms of a waits-for graph [6, 7], a directed graph that represents which transactions are waiting for which other transactions. In this paper, we explore some of the special properties of the waits-for-graph in the context of database systems, and present very efficient *linear* deadlock detection algorithms.

It seems to us that whenever people talk about deadlock detection, it is dismissed as being an expensive operation. Recently, while evaluating alternative concurrency control and recovery techniques [1], we needed a handle on the cost of doing deadlock detection. With very little effort, we developed an inexpensive algorithm. This left us wondering whether we had just rediscovered what everybody already knew or whether no one had ever really looked at the problem. With some trepidation, we finally decided that SIGMOD Record would be a good place to present our ideas. We hope that people wishing to comment on or refute our results will use the SIGMOD Record as forum for their comments.

The organization of the paper is as follows. In Section 2, we describe our assumptions regarding the locking protocol used for concurrency control. In particular, we assume all locks to be exclusive. We outline some important properties of a waits-for-graph in Section 3. Our *continuous* deadlock detection algorithm is presented in Section 4. The theoretical basis for the algorithm is presented in Appendix A. In Section 5, we relax our assumption about the locks being exclusive to allow shared read-locks and present our modified deadlock detection algorithm. The proof of correctness of the modified algorithm is given in Appendix B. In Section 6, we extend our algorithm to perform *periodic* deadlock detection instead of deadlock detection every time a transaction blocks. In Section 7, we present our conclusions.

2. Assumptions

Deadlocks arise in database systems in the context of concurrency control algorithms based on locking [4]. In locking, access to database objects is mediated by a lock manager, and a transaction must set a lock on an object that it wishes to access before being allowed to access the object. In this paper, we assume such a locking protocol is employed, and we make the following assumptions about this protocol:

1. The locking protocol is the *strict two-phase* protocol, that is, a transaction holds all its locks till its completion[1].
2. A transaction requests one lock at a time and is blocked if a lock cannot be granted.
3. All locks are exclusive.

3. Waits-for Graph in Database Systems

Deadlocks have been expressed in terms of waits-for graphs. It has been shown [6,7] that *there exists a deadlock if and only if there is a cycle in the waits-for graph.* A waits-for graph G is a directed graph whose vertices represent transactions and an edge $(T_i,T_j) \in G$ if the transaction T_i is waiting for a lock owned by T_j. We will say that T_i is **waiting* on T_j if there is a path from T_i to T_j in the waits-for graph.

Management of the Waits-for Graph

The waits-for graph is maintained by the lock manager. For each locked object, the lock manager keeps the transaction number of the owner of the lock and a queue of the transactions that are waiting for the object to become free. We will assume that the queue discipline is 'first in first out (FIFO)'[2]. Before allowing a transaction T_i to wait for a

[1] Gray [4] has shown that to avoid a cascade of transaction aborts, a transaction must hold all the locks until it executes the commit action and then release them together.

[2] Other queue disciplines can be implemented with straight forward modifications to the algorithm presented in this paper.

transaction T_j[3], the lock manager checks that the addition of the edge (T_i,T_j) to the waits-for graph will not result in a cycle in the graph. The edge (T_i,T_j) is added to the graph and T_i is blocked, only if this test succeeds. When a lock is released and a blocked transaction is activated, or when a transaction completes, the waits-for graph is appropriately modified.

Properties of a waits-for graph

1. A cycle-free waits-for graph is a forest of trees (Theorem 2 in Appendix A).

2. If the transaction T_i waits for T_j, then deadlock can occur if and only if T_i is an ancestor of T_j, that is, T_j is *waiting for T_i. (Theorem 4 in Appendix A).

3. Only the transactions corresponding to the roots in a cycle-free waits-for graph are active. All descendants of each of the roots are blocked *waiting for the root (Theorem 3 in Appendix A). Thus, a cycle is created only when the transaction corresponding to a root waits for one of its descendants.

4. Any connected subgraph of a waits-for graph can have at most one cycle (Theorem 5 in Appendix A).

4. Continuous Deadlock Detection Algorithm

The basic idea of the algorithm is that whenever a transaction T_i requests a lock owned by T_j, test if T_j is *waiting for T_i. This test is performed by taking a directed walk starting from T_j to the root of the tree. A deadlock occurs, only if the root corresponds to T_i.

Data Structures

Assume that each transaction is assigned a unique transaction number. Define the following data structure:

[3] T_j is the transaction immediately preceding T_i in the FIFO queue.

```
Tran : Array[0..N-1] of {
        Waiting-for : transaction#;
        SomeOne-waiting : boolean}.
```

N is a prime number that is used to map a transaction number (by taking mod) to an array index[4]. If the transaction t_i is blocked for a lock held by t_j, then Tran[t_i].Waiting-for = t_j. Tran[t_i].SomeOne-waiting is true only if at least one transaction is waiting for t_i to complete.

Deadlock Management Module

```
Chk-cycle($t_i$,$t_j$ : transaction#) {  -- $t_i$ requests a lock held by $t_j$

    If (Tran[$t_i$].SomeOne-waiting is false) then { -- deadlock not possible (Theorem 4)
        Add-edge($t_i$,$t_j$);
        return(o.k.)}
    else  { -- take a directed walk from $t_j$
        ancestor := Tran[$t_j$].Waiting-for;
        loop {
            If (ancestor = $t_i$) then   -- $t_j$ *waiting for $t_i$
                return(deadlock)
            else if (ancestor = Null) then { -- $t_j$ not a descendant of $t_i$
                Add-edge($t_i$,$t_j$);
                return(o.k.)};
            ancestor := Tran[ancestor].Waiting-for;
            } -- end loop
        }
    }
```

[4]Collisions may be handled using standard techniques [8].

```
Activate($t_i$ : transaction#) {
    Tran[$t_i$].Waiting-for := Null;
    }


Terminate($t_i$ : transaction#) {
    Tran[$t_i$].SomeOne-waiting := false;
    }


Add-edge($t_i$,$t_j$ : transaction#) {
    Tran[$t_i$].Waiting-for := $t_j$;
    Tran[$t_j$].SomeOne-waiting := true;
    }


Initialize {
    for i:=0 to N-1 do {
        Tran[i].Waiting-for := Null;
        Tran[i].SomeOne-waiting := false};
    }
```

Observations

Note that the loop in the function, Chk-cycle, always terminates because of Theorem 6 in Appendix A. The loop is executed as many times as the path length, PL, from t_j to the root of the tree[5]. In the worst case, PL = number of blocked transactions in the connected subgraph of the waits-for graph that contains the vertex at which the function Chk-cycle begins the search. Gray et al. [3] have observed that the probability of a transaction deadlocked in a cycle of length more than two is very rare and all

[5]The tree that contains the vertex corresponding to t_j.

deadlock cycles are essentially of length two. Hence, in practice, PL = 2, and the deadlock detection will be very inexpensive.

An optimization has been built into the algorithm by keeping track for each transaction whether any transaction is waiting for it. Theorem 7 in Appendix A provides the basis for maintaining this information. The field, SomeOne-waiting, will avoid the execution of the loop in all the cases where there is no transaction waiting for t_i. Thus, the deadlock detection will be still more efficient.

The space complexity of the algorithm is O(N).

5. Shared Read Locks - An Embellishment

The deadlock detection scheme presented in the previous section is based on the assumption that all locks are exclusive. However, many real systems allow read-locks to be shared and only write-locks are required to be exclusive. In such an environment, the number of outgoing edges from a vertex in the waits-for graph is not bounded by one (Lemma 1 in Appendix A) and the deadlock detection scheme described in the previous section is not directly applicable.

Modified Deadlock Detection Scheme

We will present a modification in the way the waits-for graph is managed that will guarantee that there is at most one outgoing edge from each of the vertices of the waits-for graph. With this modification, the deadlock detection scheme presented in the previous section can be used.

When a writer T_i wishes to wait on a read-lock and there are more than one readers, the lock manager selects *one* of the current readers[6], T_j, ensures that the addition of the edge T_i->T_j would not create a cycle, and adds T_i->T_j to the waits-for graph, Later, when T_j commits, the lock manager checks if there are still readers. If not,

[6]A good heuristic to minimize the additional overhead might be to pick the reader that started reading last, based on the assumption that it might be the one to more likely finish reading last.

T_i is granted the lock and is allowed to proceed. If yes, however, T_i->T_j is changed to T_i->T_k for some ongoing reader T_k, if it does not introduce a cycle.

This is analogous to what might happen in a "typical" system in the following situation: T_1 gets a read-lock on X, T_2 requests a write-lock on X and is blocked (T_2->T_1), and T_3 requests a read-lock on X. If the system allows new readers when a writer is waiting (a policy decision that each system must make, based on a fairness versus throughput tradeoff), it would probably not add T_2->T_3. Rather, it would probably let T_2 wake up when T_1 completes, re-blocking T_1 if T_3 is still reading at that time.

Note that this scheme still allows deadlocks to be detected in linear time, although not always right as they arise. In the above example, suppose that T_3 ends up waiting on T_2. Deadlock won't be detected until T_1 completes, at which time the edge T_2->T_3 is added, and the cycle is found.

The proof of correctness is presented in Appendix B.

6. Periodic Deadlock Detection

In this section, we present an outline of the extension to the continuous deadlock detection scheme that enables periodic deadlock detection in linear time. With periodic detection, instead of checking for a cycle before adding an edge to the waits-for graph, edges are added to the graph without any test and the graph is periodically examined for cycles.

Define a function Detect-cycle(v) analogous to the Chk-cycle function defined previously that causes a directed walk in the waits-for graph starting from the vertex v. The walk will either terminate at a root or will again reach v, in which case, a cycle has been detected. Detect-cycle marks every vertex that it touches in the process of searching for a cycle as *visited*.

We will now present the periodic deadlock detection algorithm.

```
Periodic-detection {
    for index := 0 to N-1 do Tran[index].Visited := false;   -- initialize
    index := 0;
    while (index < N) {
        Detect-cycle(index);
        while ((Tran[index].Visited is True) And (index < N)) index := index +1;
        }}
```

The algorithm simply runs the function Detect-cycle on the first vertex, advances to the next unvisited vertex, runs Detect-cycle there, etc. In other words, it runs our linear deadlock detector at every connected subgraph of the waits-for graph. The time complexity of the algorithm is O(N), that is, it is linear in the total number of blocked transactions.

7. Conclusions

In this paper, we have shown that deadlock handling does not have to be expensive in the context of database systems. Given certain reasonable assumptions about the nature of the locking protocol employed, deadlock detection can be accomplished very inexpensively, and we have presented an implementation of such a scheme. This scheme is directly applicable to uniprocessor database systems, and it may easily be extended for use in distributed database systems where a central deadlock detection mechanism is employed [10].

8. Acknowledgements

Dan Ries pointed out that an earlier version of the deadlock detection scheme presented here was applicable only in the environment where all locks were exclusive. Michael Stonebraker helped us muster enough courage to submit this article. Toni Guttman, Margie Murphy and Clark Thompson provided helpful discussions and comments on various aspects of our ideas.

APPENDIX A

Definitions

We will first introduce some graph-theoretic definitions adapted from [2,5].

A *directed graph* (or a *digraph* for short) G consists of a set of *vertices* $V = \{v_1, v_2, \ldots\}$, a set of *edges* $E = \{e_1, e_2, \ldots\}$, and a mapping that maps every edge onto some *ordered* pair of vertices (v_i, v_j). A vertex is represented by a point and an edge by a line segment between v_i and v_j with an arrow directed from v_i to v_j. The vertex v_i is called the *initial vertex* and v_j the *terminal vertex* of the edge.

The number of edges incident out of a vertex v_i is called the *out-degree* of v_i and is written $d^o(v_i)$. The number of edges incident into v_i is called the *in-degree* of v_i and is written $d^i(v_i)$. A *sink* is a vertex v_i with $d^o(v_i) = 0$.

A *(directed) walk* in a digraph is an alternating sequence of vertices and edges, $\{v_0, e_1, v_1, \ldots e_n, v_n\}$ in which each edge e_i is (v_{i-1}, v_i). A *closed walk* has $v_n = v_0$. A *path* is a walk in which all vertices are distinct; a *cycle* is a nontrivial closed walk with all vertices distinct (except the first and the last). An edge having the same vertex as both its initial and terminal vertices is called a *self-loop*. If there is a path from v_i to v_j, then v_j is said to be *reachable* from v_i. The *length* of a path is the number of vertices involved in the path.

Each walk is directed from the first vertex v_0 to the last vertex v_n. We need a concept that does not have this directional property. A *semiwalk* is again an alternating sequence $\{v_0, e_1, v_1, \ldots e_n, v_n\}$ of vertices and edges but each edge e_i may be either (v_{i-1}, v_i) or (v_i, v_{i-1}). A *semipath* and a *semicycle* is analogously defined.

A digraph is said to be *connected* if there is at least one semipath between every pair of its vertices; otherwise, it is *disconnected*. It is easy to see that a disconnected graph consists of two or more connected subgraphs. Each of these connected subgraphs is called a *component.*

An *in-tree* is a digraph G such that 1) G contains neither a cycle nor a semicycle, 2) G has precisely one sink. This sink is called the *root* of the in-tree.

Characteristics of a waits-for graph[7]

THEOREM 1. *A waits-for graph does not have any self-loop.*

Proof. A transaction does not wait for a lock that it owns itself.

LEMMA 1. *Assuming all locks to be exclusive, for all vertices v_i in a waits-for graph, $d^o(v_i) \leq 1$.*

Proof. A transaction cannot wait for more than one transaction at a time.

LEMMA 2. *G is a digraph with n vertices. If there is a unique semipath between every two vertices of G, then the number of edges in G = $n-1$.*

Proof. Theorem 4.1 in [5].

LEMMA 3. *In a digraph, the sum of the out-degrees of all vertices is equal to the number of edges in the digraph.*

Proof. Each edge contributes exactly one out-degree.

LEMMA 4. *Any component of a waits-for graph cannot have more than one sink.*

Proof. Suppose a component G has two sinks v_0 and v_n. Since G is connected, we can find a semipath between v_0 and v_n. Extract the subgraph G' that has only the vertices and the edges comprising this semipath. Let there be p vertices in G'. By Lemma 2, number of edges in G' = p-1 and by Lemma 3, the sum of out-degrees of all vertices in G' = p-1. However, since $d^o(v_0) = d^o(v_n) = 0$, there must be some vertex v in G', and hence in G, that has $d^o(v) > 1$. But, this contradicts Lemma 1.

LEMMA 5. *An acyclic digraph has at least one vertex of out-degree zero.*

[7]We will assume through out that the graph is non-empty

Proof. Theorem 16.2 in [5].

LEMMA 6. *A connected digraph G is an in-tree if and only if exactly one vertex of G has out-degree 0 and* all others have out-degree 1.

Proof. Theorem 16.4' in [5].

THEOREM 2. *A cycle-free waits-for graph is a forest of in-trees.*

Proof. Follows from Lemma 1, Lemma 4, Lemma 5 and Lemma 6.

LEMMA 7. *In an in-tree, there is a unique path from every vertex to the root.*

Proof. Theorem 9.3 in [2]

THEOREM 3. *The root v of each of the in-trees in a cycle-free waits-for graph corresponds to an active transaction for which all other transactions in the tree are *waiting.*

Proof. If v corresponds to a waiting transaction, then $d^{o}(v) = 1$, a contradiction. The second part of the theorem follows from Lemma 7.

LEMMA 8. *Blocking of a transaction that has no other transaction waiting for it cannot create a cycle in the waits-for graph.*

Proof. The vertex corresponding to a transaction that has no transaction waiting for it has no incoming edge.

LEMMA 9. *Waiting for a lock owned by an active transaction cannot result in a cycle in the waits-for graph.*

Proof. The vertex corresponding to an active transaction has no outgoing edge.

THEOREM 4. *Wait by a transaction T_i for a lock held by T_j will result in a cycle in the waits-for graph if and only if T_j is *waiting for T_i.*

Proof. First, if T_j is *waiting for T_i, then there is a path from T_j to T_i, and the addition of the edge (T_i,T_j) will create a cycle in the waits-for graph.

Suppose now that T_j is not *waiting for T_i and the addition of the edge (T_i,T_j) creates a cycle in the waits-for graph. We will show a contradiction. If T_j is active, waiting for T_j cannot create a cycle by Lemma 9. If T_j is *waiting for a transaction T ($\neq T_i$), let us traverse the cycle created by the addition of the edge (T_i,T_j) starting from T_i. Since, there is a unique path between T_j and T (Lemma 1), the cycle should have a path between T and T_i. But that would imply that T_j is *waiting for T_i.

LEMMA 10. *A vertex can be on at most one distinct cycle in a waits-for graph.*

Proof. Let a vertex v be on more than one distinct cycles. Starting from v and moving along the cycles, a vertex v' (v' may be v) will be reached such that there are two out-going edges from v'. But, then $d^o(v') > 1$, and that contradicts Lemma 1.

THEOREM 5. *Any component of a waits-for graph can have at most one cycle.*

Proof. The transaction T corresponding to the root of a cycle-free component of a waits-for graph is the only active transaction amongst the transactions involved in the component. Hence, by Theorem 4, only a wait by T can cause a cycle in the component, and by Lemma 10, T can cause at most one cycle.

THEOREM 6. *A directed walk from any vertex in a component of a waits-for graph would result either in detection of the cycle or termination at the root.*

Proof. Follows from Lemma 7 and Theorem 5.

THEOREM 7. *The in-degree of a vertex in the waits-for graph of a database system increases monotonically until the vertex is removed from the graph with the completion of the corresponding transaction.*

Proof. A transaction holds all the locks until its completion [4].

APPENDIX B

Model

We postulate a waits-for graph, as before, whose vertices are transactions and edges (called *explicit* edges from here on) of the form T_i->T_j indicate that T_i is explicitly waiting on T_j. For proof purposes, there is also a collection of *implicit* edges of the form T_i->T_k, indicating that T_k is one of the *other* readers of some object X that T_i wishes to write. These implicit edges would be present in a waits-for graph, if we were not trying to bound the out-degree of vertices by one. Let the graph with just explicit edges be called the *E-graph*, and the graph with both explicit and implicit edges be called the *EI-graph*.

Proof of Correctness

Our algorithm will be deemed correct if it can be shown that no cycle (implicit or explicit) in the EI-graph can persist forever.

LEMMA 1. *If a vertex in the EI-graph has an outgoing implicit edge, there must be at least one outgoing explicit edge from it.*

Proof. A transaction T_i can implicitly wait for a transaction T_k if T_i requests a write-lock for some object X that has been read-locked by T_k and at least one other transaction T_j ($j \neq k$), and T_i chooses T_j to explicitly wait for. The other scenario where T_i can implicitly wait for T_k is when T_i is already explicitly waiting for some transaction T_j that holds a read-lock on X, and T_k arrives later on and is also granted a read-lock on X. In both situations, the implicit edge T_i->T_k cannot be present without the explicit edge T_i->T_j being present as well.

Now, when transaction T_j commits, the explicit edge T_i->T_j is removed and an implicit edge T_i->T_m is made explicit. If $k = m$, the implicit edge T_i->T_k is now explicit; otherwise, the implicit edge T_i->T_k remains implicit, but still has a related outgoing explicit edge.

LEMMA 2. *Sinks in the E-graph are also sinks in the EI-graph.*

Proof. Suppose there is a vertex v that is a sink in the E-graph but not in the EI-graph. The vertex v cannot have an outgoing explicit edge in the EI-graph because, by definition of the E-graph, the same edge would be outgoing from v in the E-graph and v is a sink in the E-graph. If v has an outgoing implicit edge, then by Lemma 1, there must be an outgoing explicit edge from v as well, and that is not possible. Hence, v is a sink in the EI-graph as well.

LEMMA 3. *If our deadlock detection algorithm is applied to the E-graph, then, at any time, the E-graph will have at least one sink.*

Proof. By Theorem 2 in Appendix A, a cycle-free E-graph is a forest of in-trees and our deadlock detection algorithm never allows any cycle to be formed in the E-graph.

THEOREM 1. *If our deadlock detection algorithm is applied to the E-graph, we cannot reach a state in which no transaction can proceed.*

Proof. Either no transaction is waiting, or by Lemmas 2 and 3, at any time, there is at least one sink in the EI-graph that corresponds to a runnable transaction.

COROLLARY 1. *Cycles in the EI-graph cannot persist forever.*

Proof. Assume that we quiesce the system, in the sense that no new transactions are allowed to enter. By Theorem 1, at any time, there is at least one runnable transaction in a non-empty system. Assuming that transactions are finite in length, all transactions will eventually either commit or abort. Either way, all vertices are eventually removed from the graph, and hence, all cycles eventually go away.

REFERENCES

[1] R. Agrawal and D.J. DeWitt, Performance of Integrated Recovery and Concurrency Control Mechanisms, in preparation (1982).

[2] N. Deo, "Graph Theory with Applications to Engineering and Computer Science," Prentice-Hall, Englewood Cliffs, N.J. (1974).

[3] J.N. Gray, P. Homan, H. Korth, and R. Obermarck, "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System," Rep. RJ3066, IBM Research Lab., San Jose, California (Feb. 1981).

[4] J.N. Gray, "Notes on Database Operating Systems," in *Lecture Notes in Computer Science 60*, *Advanced Course on Operating Systems*, ed. G. Seegmuller,Springer Verlag, New York (1978).

[5] F. Harary, "Graph Theory," Addison-Wesley, Reading, Massachusetts (1972).

[6] R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys 4*, 3, pp. 179-196 (Sept. 1972).

[7] P.F. King and A.J. Collmeyer, "Database Sharing - An Efficient Method for Supporting Concurrent Processes," *Proc. AFIPS 1973 Natl. Computer Conf.*, pp. 271-275 (1973).

[8] Knuth, D.E., "The Art of Computer Programming: Fundamental Algorithms," Vol. 1, 2nd edition, Addison-Wesley, Reading, Massachusetts (1973).

[9] G. Newton, "Deadlock Prevention, Detection and Resolution," *ACM-SIGOPS Operating Systems Review 13*, 4, pp. 33-44 (April 1979).

[10] M.R. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. Software Eng. SE-5*, 3, pp. 188-194 (May 1979).