Deadlock Detection is Really Cheap

Bin Jiang

Department of Computer Science
Technical University Darmstadt
Alexanderstr. 24
D-6100 Darmstadt, West-Germany

Abstract

The treatment of deadlock has previously been considered as expensive, and implementations of the involved algorithm have appeared rather complex. In this paper a new algorithm for the treatment of deadlock is described. It is not only very efficient (in the worst case $O(e)$ where e is the number of the edges in the "wait-for" graph) but also fairly simple. With this algorithm one can detect all deadlocks as soon as possible and list all participators in the corresponding cycle of the "wait-for" graph. The algorithm is described for the use of database transaction management.

0. Introduction

A multi-user database system allows several users to operate on shared data. Usually, the synchronisation of the data acceses is provided by the transaction management component which typically uses the method of locking objects before granting access. But it is well known that locking can lead to situations like the following: While user A (e.g. transaction A) waits for object Y which has been locked by user B, user B waits for object Z which has been locked by user C, while user C, in turn, waits for object X which has been locked by user A. Such a situation is known as *deadlock*, and the task we are embarking on in this paper is to detect and eliminate deadlock situations. For a long time, this problem has been considered as rather costly, and in fact timeout is often used as an alternative to deadlock detection. It seems that Agrawal et al. [ACD 83] have been the first which claimed the opposite, i.e. that deadlock detection is cheap. They presented an efficient algorithm (linear w.r.t. the number of nodes in the "wait-for" graph) where only "exclusive" locks are considered, and applied that algorithm repeatedly to cope with "shared locks" as well. However, in the later case their method does not detect deadlocks right as they arise, and so transactions may hold resources and block concurrent users unnecessarily long, before they eventually become deadlock victims anyway. In contrast to this, the algorithm proposed here detects deadlocks as soon as possible at still low costs.

The organization of the paper is as follows. The underlying model of database transaction management is described in section 1 as the basis of the following discussion. In section 2, our algorithm for the treatment of deadlock is introduced. An

analysis of the complexity and completeness of the algorithm is presented in section 3.

1. Transaction Management

Our model of a database system assumes that several transactions run concurrently. Each transaction consists of a sequence of read or write operations on common objects, e.g. records or pages.

To avoid interferences of transactions, an object is locked, before access is granted and according to the well-known strict two-phase locking protocol [Gr 78], kept locked until the corresponding transaction is completed. Two lock modes are introduced in our model, viz. "*shared*" for concurrent readers and "*exclusive*" for writers.

If an object is locked by a transaction in "shared" mode, other transactions are permitted to lock the object only in "shared" mode. However, if an object is locked by a transaction in "exclusive" mode, other transactions are not allowed to lock the object at all.

Only read operations are allowed in "shared" mode, write operations require "exclusive" mode. A transaction which tries to lock an object that has already been locked by other "conflicting" transactions must wait until all locks on the object are released. This results in a "wait-for" relationship where a transaction could not only block several other transactions (e.g. a transaction which locks an object in "exclusive" mode blocks all other transactions requesting a lock on the same object), but could also wait for several other transactions (e.g. a transaction that tries to lock an object in "exclusive" mode waits for all transactions concurrently holding a "shared" lock on the object).

If each transaction in the system is represented as a vertex and the "wait-for" relationship - transaction A waits for transaction B - as a directed edge from vertex A to vertex B, the "wait-for" relationship of all transactions in the system can be represented as a directed graph, usually known as "*wait-for*" *graph*. Note that it is possible that a vertex is a start point and end point of several edges simultaneously.

In database transaction management, the objective is not only to prevent the illegal interferences of concurrent users, but also to detect and eliminate the situations which have been described in the introduction, i.e. deadlocks. In our graphical representation a deadlock is just a cycle in the graph. This means that the problem of deadlock detection can be transformed into another problem, i.e. detection of cycles in a directed graph.

For the elimination of a deadlock one of the transactions participating in the

deadlock has to be rolled back. There are various criteria for the selection of this victim, e.g. the youngest transaction, the laziest transaction and so on. But it is still a question which transactions participate in the deadlock at all. The corresponding question for the graph representation is which vertices lies on the cyclic path.

Our system runs as following: As soon as a transaction tries to lock an object, it is tested whether the request can be satisfied. If it is the case, the object is locked by the transaction in the desired mode and the system proceeds. If the request can not be granted at the moment, i.e. a lock (conflict), the transaction must wait, and it is tested whether this can cause a deadlock. If not, the system goes on, otherwise one of the transactions participating in the deadlock is rolled back. Note that there is no deadlock in the system before a new lock request is suspended.

2. Algorithms for the Treatment of Deadlock

From the above sketch one could see that the problem of treatment of deadlock can be partitioned into two subproblems:

(1) detection of deadlock in the system;
(2) elimination of the deadlock from the system.

In our graph representation these correspond to:

(1) detection of cycle in the "wait-for" graph;
(2) listing all vertices which lay on the cycle.

Many efforts have been made, to solve the problem ([Ne 79], [Zõ 83] and [El 86]). But until now, none of the proposed algorithms is actully satisfactory for the use of database transaction managment: They are either expensive, or complex, or incomplete, or even not correct. One of the few algorithms[ACD 83] that can cope with "shared" locks has the disadvantage that deadlocks may exist for quite a while before they are detected and eliminated. For this reason, we have recently attempted once more to work out a better algorithm. The basic idea of the algorithm is: *From a certain vertex one visits breadth-first all reachable vertices exactly once.* If the directed graph is represented with a $n \times n$-matrix A where n is the number of the vertices in the graph and for all i and j with $1 \leq i,j \leq n$: $A[i,j] = 1$, if there is an edge from vertex i to vertex j, else $A[i,j] = 0$, our algorithm looks as follows:

```
1 procedure Cycle ( k: integer; var A: array[1..n, 1..n] of integer );
2           var i, j, l: integer;
            begin
3               for l:= 1 to n - 1 do
4                   for i:= 1 to n do
5                       if A[k, i] = l then
6                           for j:= 1 to n do
7                               if A[i,j] = 1 and A[k, j] = 0 then
8                                   A[k,j]:= l + 1;
            end.
```

Figure 1: Algorithm "Cycle"

Explanations:

Line 1: Algorithm "Cycle" has two parameters. One is the vertex number k (for the transaction number) which is tested whether it is involved in a cycle; the other is an $n \times n$-matrix A which represents the "wait-for" graph.

Line 2: Three working variables are used, viz. i, j, l.

Line 3: The following computation is carried out n-1-times. Each time, it is regarded with a certain length l of path (the length of a path is the number of edges which lead from one end to another):

Line 4: For all n vertices the following will be computed:

Line 5: If there is a shortest path of length l which leads from vertex k to vertex i, the following will be done:

Line 6: It is tested for each vertex j

Line 7: whether there is a vertex that leads from vertex i to vertex j, and no path is known yet between vertex k and vertex j. If this is the case, it is evident that there is a shortest path of length $l + 1$ which leads from vertex k to vertex j.

Line 8: This fact is noted down.

If after the computation of the algorithm, $A[k,k]$ is not equal to 0, there exists a cycle in the graph. The following figure shows an example:

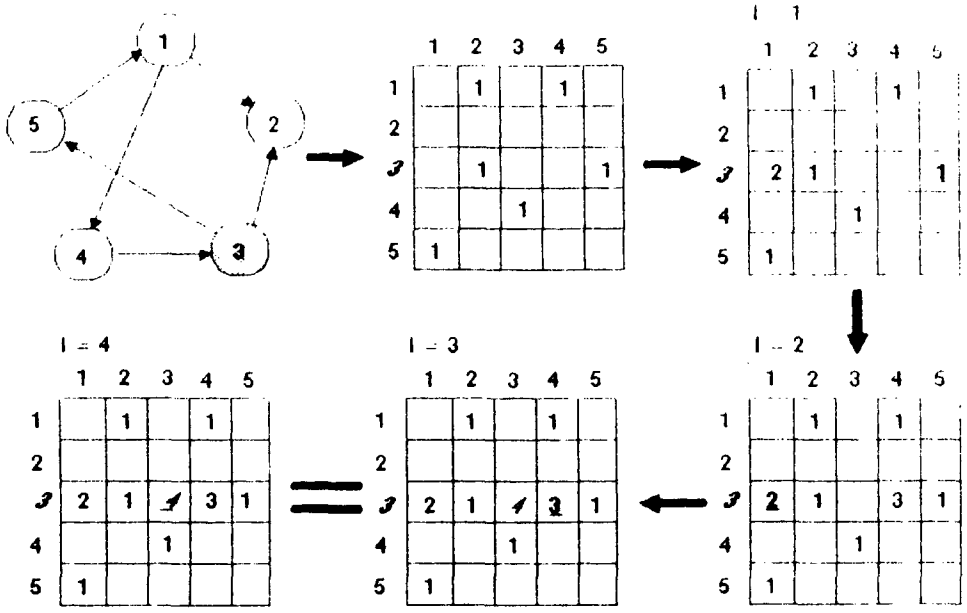

Figure 2: Example 1.

If we modify the algorithm as follows, the computation will stop as soon as a cycle has been found:

```
0 procedure Cycle*( k: integer; var A: array[1..n, 1..n] of integer );
            var i, j, l, flag: integer;
            begin
1               flag:= 1;
                l:= 1;
2               while flag = 1 and A[k,k] = 0 and l < n do
                begin
3                   flag := 0;
                    i := 0;
4                   while A[k,k] = 0 and i < n do
                    begin
                        if A[k,j] = l then
                        begin
                            j := 0;
                            while j < n do
                            begin
                                if A[i,j] = 1 and A[k,j] = 0 then
                                begin
                                    A[k,j] := l + 1;
5                                   flag := 1;
                                end;
                                j := j + 1;
                            end;
                        end;
                        i := i + 1;
                    end;
                    l := l + 1;
                end;
            end.
```

Figure 3: Algorithm "Cycle*"

Explanations:
Algorithm "Cycle*" is a variant of "Cycle". The differences are:

1. All for-loops in "Cycle" have been replaced by while-loops.
2. The stop conditions of the loops in "Cycle*" are stricter than those in "Cycle".

Line 1: Assuming that from vertex k there are paths to be traversed,
Line 2: it will be tested at most n-1-times whether a path in the graph has been found which leads back from vertex k to itself, or whether there could still

be a path from vertex k which has not been completely traversed yet.

Line 3: At the beginning of each computation it is not known whether there exists a path which has not been completely traversed in the previous iterations.

Line 4: Provided that no path has been found that leads back from vertex k to itself, the following computation is carried out for all other n - 1 vertices.

Line 5: It is possible that there is still a part of this path that has not yet been traversed. It will be tested in the next computation if this is the case.

If the outcome of algorithm "Cycle*" determines that vertex k is involved in a cycle, then let the following algorithm "Participators" run on the transformed matrix A: Participators(k, l-2, k, 1, A, B). This algorithm gathers all vertices participating in the cycle (except vertex k)into array B.

```
1  procedure Participators( k, l, j, m: integer;
                 var A: array[1..n, 1..n] of integer;
                 var B: array[1..n] of integer );
2                var i: integer;
                 begin
3                     i:= 0;
4                     while i < n and not( A[k,i] = l and A[i,j] = 1 ) do
5                           i := i + 1;
6                     B[m]:= i;
7                     m:= m + 1;
8                     if not( l = 1 )then
                      begin
9                           l := l - 1;
10                          Participators( k, l, i, m, A, B);
                      end;
                 end.
```

Figure 4: Algorithm "Participators"

Explanations:

Line 1: Algorithm "Participators" has 6 Parameters: k is the number of the vertex that lies on the cycle; l is the length of the path which is currently considered; j is the number of a certain vertex; m is the current tail of the queue of vertices which form the cycle; A is the result matrix of algorithm "Cycle*"; B is the queue of participators in the cycle.

Line 2: i is a working variable.

Line 3-5: A vertex (i) is searched such that there is a path of length l which leads from vertex k to this vertex and there is an edge which leads from this vertex to a certain vertex (j).

Line 6-7: The found vertex (i) is put at the end of the queue of participators.

Line 9-10: If the algorithm has been performed for all possible path lengths, the list of participators is complete, otherwise proceeds with length l - 1.

The following is an example:

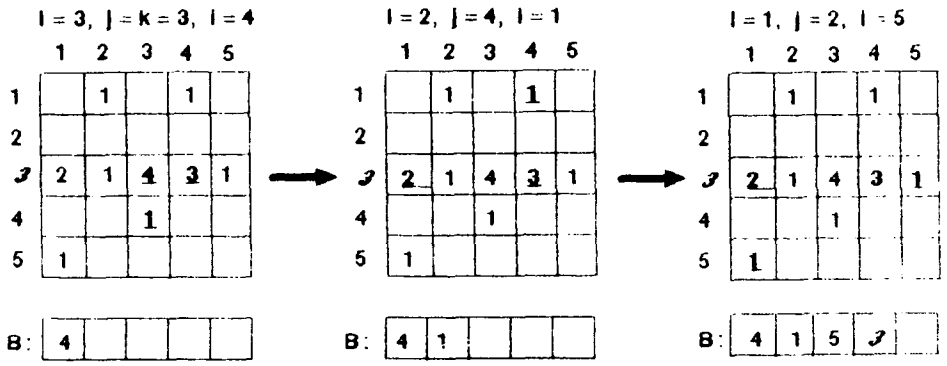

Figure 5: Example 2.

Now, by puting both algorithms together, we get a complete algorithm for deadlock treatment.

3. Complexity and Completeness of the Algorithm

At first sight one could get the impression that algorithm "Cycle" has complexity $O(n^3)$ since it has three for-loops (lines 3, 4 and 6). But in fact its complexity is $O(n^2)$.

The comprising part - lines 6 - 8 of algorithm "Cycle" - is $O(n)$. But note that the execution of this piece is restricted by the condition in line 5, i.e. if there is a shortest path of length l which leads from vertex k to vertex i ($A[k,i] = l$).

There could be several paths from vertex k to another vertex. But note that the minimum length of these paths is unique for each vertex. All such minimums regarding each vertex are gradually computed yielding l as the result and noted down in row k for each vertex. The fact that it is a minimum that has been noted down, can be noticed from lines 5, 7 and 8 of "Cycle". With these minimums we can determine which vertices have already been visited. Since there are at most n such minimums, and for each minimum lines 6 - 8 of algorithm "Cycle" are executed only once, a computational upper bound is n × (lines 6 - 8) for the search for all vertices which can be reached from vertex k. Ihis means that the whole computation is $O(n^2)$.

After the computation of "Cycle*", "Participators" is called which apparently is of $O(n)$ (lines 4 - 5). However, it is a recursive algorithm (line 10) and will be called at most $(n - 1)$ times altogether. This means that listing all participators in the deadlock has complexity $n \times O(n) = O(n^2)$. Combining both parts, we get the result that the entire algorithm has complexity $O(n^2)$.

If instead of matrix A we use a set of chains, each of which represents a row of the matrix where each element of the chain stands for an element of the matrix that has been marked with *1* and contains the vertex number and a pointer to the next element of the chain (the chain representing row k consists of the elements which are considered at that point, i.e. they are marked with *1*, and has a similar structure as the following chain-matrix), we can make algorithm "Cycle*" even more efficient, viz. $O(e)$ where e is the number of the elements in the chains or the number of the vertices in the "wait-for" graph. In our case a general upper bound for e is: $e < \frac{n^2}{2}$.

But this data structure is not suitable for the algorithm "Participators", for it would lead to $O(n^3)$. To avoid this we introduce a new data structure *chain-matrix*: an $(n+1) \times n$ -matrix. Each element of the first column is the index of the first marked element in the same row. Each marked element is the index of the next marked element in the same row. The last marked element in each row is a negative number. With such a data structure the complexity of "Participators" remains unchanged. The following is an example of the data structure:

1	2	3	4	5	6
3		5		- 1	
- 1					
3		6			- 1
4			- 1		
2	- 1				

Figure 6: Chain-matrix.

If vertex k lies on a cycle, this fact is detected by "Cycle" in any case. If vertex k is at the same time in several cycles which are of the same size (w.r.t. the number of vertices), they can be determined from the computed matrix A. If we use the following "Participator*" instead of "Participators", all cycles will be listed:

```
0 procedure Participators*( k, l, j, m: integer;
            var A: array[1..n, 1..n] of integer;
            var B: array[1..n] of integer );
            var i: integer;
             begin
                  for i:= 1 to n do
                      if( A[k,i] = l and A[i,j] = 1 ) then
                      begin
                            B[m]:= i;
                            m:= m + 1;
                            if not( l = 1 )then
                            begin
                                  l := l - 1;
                                  Participators( k, l, i, m, A, B);
                            end;
                            else
                                  writeln(B);
                       end;
            end.
```

Figure 7: Algorithm "Participators*".

With a minor change to the treatment of element $A[k,k]$ in "Cycle", all cycles containing vertex k can even be determined from the computed matrix A. The algorithm for the listing is similar to "Participators*". The costs of this are dependent on the number of the cycles. In the worst case it is an exponential function of the number of the vertices: $O(3^{\frac{n}{3}})^*$. But in practice it is certainly far less costly, since most deadlocks seem to consist of only two transactions [GHKO 81].

---

* The general problem is the searching for the maximum of $i^j \times l^m \times ... \times p^q$ where $i \times j + l \times m + ... + p \times q = n$.

4. Conclusion

A new algorithm for the treatment of deadlock has been described. It is partitioned into two parts, one for deadlock detection and the other for listing the participators of deadlocks.

The algorithm is

- . very efficient: O(e);
- . very simple: see Figure 1 and Figure 4;
- . correct: evident; and
- . complete.

With appropriate modifications it could also be used on "*AND/OR- wait-for*" graph for cycle detection [Bi 78], [BO 81]. The algorithm has been implemented as part of the transaction manager in the database kernels *DASDBS*[SW 86], [PSSWD 87] and *DBKSI* [PSSW 87].

Acknowledgments

I would like to thank Gerhard Weikum, Uwe Deppich and Peiling Zhu for the critical reading, references and discussions, Prof. H.-J. Schek, Prof. A. Söder and all colleagues in DVSI and Insotec-T7 for the support and encouragement.

References

[Ne 79] : Newton, *Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography*, ACM Operating Systems Review, p33-44, April 1979.

[ACD 83] : Agrawal et al., *Deadlock Detection is Cheap*, ACM SIGMOD Record, Vol.13, No.2, p19-34, Jan. 1983.

[Zö 83] : Zöbel, *The Deadlock Problem: A Classifying Bibliography*, ACM Operating Systems Review, p6-15, Oct. 1983.

[El 86] : Elmagarmid, *A Survey of Distributed Deadlock Detection Algorithms*, ACM SIGMOD Record, p37-45, Sept. 1986.

[GHKO 81] : Gray et al., *A Straw Man Analysis of the Probability of Waiting and Deadlock in DB System*, RJ3066 IBM Research Laboratory, San Jose, California, Feb. 1981.

[Bi·79] : Bittman et al., *Models and Algorithms for Deadlock Detection*, in Operating systems: Theory and Practice. North Holland Publishing Company, p101-111, 1979.

[BO 81] : Beeri et al., *A Resource Class Independent Deadlock Detection Algorithm*, VLDB, p166-178,1981.

[SW 86] : Schek, Weikum, *DASDBS: Concepts and Architecture of a Database System for Advanced Applications*, Techn. Rep. DVSI-1986-T1, TU Darmstadt, 1986.

[Gr 78] : Gray, *Notes on Data Base Operating Systems*, In Operating Systems - An Advanced Course, Lecture Notes in Computer Science 60, Springer, Berlin-Heidelberg-Nowyork, 1978.

[PSSW 87] : H.-B. Paul, A. Söder, H.-J. Schek, G. Weikum, *Supporting the Office Filing Service by a Database Kernel System* (in German), Proc. of the GI Conference on Database Systems for Office Automation, Engineering, and Scientific Applications, Darmstadt, 1987.

[PSSWD 87] : H.-B. Paul, H.-J. Schek, M.H. Scholl, G. Weikum, U. Deppisch, *Architecture and Implementation of the Darmstadt Database Kernel System*, ACM SIGMOD Conference, San Francisco, 1987.