An Optimal Deadlock Resolution Algorithm in Multidatabase Systems

Xuemin Lin
Department of Computer Science
The University of Western Australia
Nedlands, Perth, WA 6907, Australia
lxue@cs.uwa.oz.au

Jian Chen
Department of Software Development
Monash University
Melbourne, Australia
jianchen@insect.sd.monash.edu.au

Abstract

In this paper, we propose a novel deadlock resolution algorithm. In the presence of global deadlocks in a multidatabase system, this algorithm always selects an optimal set of victims for removing deadlocks. It makes the use of network flow techniques, and runs in time $O(n^3)$, where n is the number of the deadlocked global transactions. Furthermore, the proposed deadlock resolution algorithm dose not have livelock and transaction processing starvation problems.

Keywords: Deadlock, Multidatabase, Concurrency Control, Optimization.

1 Introduction

A *multidatabase system* (MDBS) is a federation of independently developed component database systems through a communication network. These component database systems are also called *local database systems* (LDBSs) in contrast to an MDBS. There are two types of transactions in an MDBS:

- *Local transactions* - that execute in a single LDB-S.
- *Global transactions* - that execute on several LDBSs.

One major issue in transaction management in an MDBS is concurrency control. The current developments in concurrency control techniques in multidatabase systems can be classified into two families: one is deadlock free [3,9], and another requires deadlock detection [5,10]. Discussions about advantages and disadvantages of these two approaches are outside the coverage of this paper. The interested reader may refer to [12] for details. Usually, a concurrency control approach without forcing deadlock free can potentially provide a greater concurrency degree. However, this should co-operate with an efficient and effective deadlock resolution.

In this paper, we assume that the currency control technique used in a multidatabase management system may cause the existence of deadlocks. Particularly, we employ the multidatabase management system and the concurrency control mechanism from [5,6]:

1. **At the global level**: the global transaction manager (GTM) has no access to local DBMS; each global transaction can have at most one subtransaction at each LDBS; each subtransaction may consist of several operations; the GTM submits the operations of a global transaction one by one; and the GTM submits an operation of a transaction T to an LDBS only if the previous submitted operation of T has been completed.
2. **At the local level**: each local database management system uses a *two-phase locking* protocol [2] for local serializability, and has a mechanism for ensuring freedom from local deadlocks; no changes can be made to local database systems in order to preserve local autonomy; a local database management system is not able to distinguish between local global transactions which are active at the LDBS, that is, it treats a subtransaction decomposed from a global transaction as a local transaction; and a local database management system is not able to communicate directly with other local DBMSs to synchronize the execution of a global transaction active at several LDBSs.

Consequently, a global transaction may wait, at most, at one LDBS each time; and we can assume that each local schedule is serializable and that local deadlocks can be resolved through a local concurrency control approach. However, global deadlocks (that is, the deadlocks among global transactions) may still exist due to possible conflicts among global transactions. These conflicts may even be *indirect conflicts* [5,10].

Two deadlocks resolution algorithms have been recently reported [6,7]. Due to indirect conflicts unknown to a GTM, these two algorithms cannot overcome all of the following drawbacks: 1) unnecessarily increase transaction abortion costs, 2) create *livelock* problem, and 3) create transaction processing *starvation* problem.

In this paper, we will present a novel deadlock resolution algorithm, which can avoid all of the above drawbacks. The algorithm presented in this paper uses the network flow techniques. Particularly, we apply the algorithm for solving the maximum flow problem. As the outcome, the proposed algorithm can always

0-8186-7267-6/96 $05.00 © 1996 IEEE

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:22 from IEEE Xplore. Restrictions apply.

choose a set of victims with the minimum overall abortion costs to remove deadlocks in combining with a time-out technique.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the algorithms in [6,7], and the problems inherent in them. Section 3 presents our deadlock resolution algorithm, together with its performance analysis. Section 4 concludes the paper.

2 An Overview of Related Works

To detect global deadlocks, the information about the conflicts among global transactions is required. However, indirect conflicts cannot be detected due to site autonomy. To resolve this situation, a *potential conflict graph* has been introduced in [5] to give an approximation description of conflicts.

A global transaction T_i has a *server* at a site S_j (LDBS) if the transaction contains a subtransaction to be processed at S_j. The sever of T_i at S_j is responsible for the communication between S_j and the GTM for processing T_i. A transaction T_i is *active* at site S_j if it has a server at S_j, and if the server is performing the operation of T_i at the site or has completed the current operation of T_i and is ready to receive the next operation of T_i. A transaction that is not active at site S_j is said to be *waiting* at site S_j, provided that it has a server at the site and at least one operation of the transaction was submitted to the site. A potential conflict graph (PCG) is described as a directed graph $G = (V, A)$ whose vertex set V consists of the global transactions. An arc $T_j \to T_i$ is in A if there is a site at which T_j is waiting and T_i is active.

A time-out mechanism BLS has been proposed by Breibart, Litwin and Silberschatz in [6] which cooperates with the potential conflict graph to remove global deadlocks. The algorithm BLS initially issues a timestamp and sets up a timer to each global transaction. Once the time-out expires on a waiting transaction T, BLS implements the following two steps:

BLS1: If there is a cycle containing T in the potential conflict graph at that time, determine the set of all transactions which are active at the waiting site of T and involved in a cycle through T. If T is older (with respect to timestamp) than all transactions in the set, T continues waiting; otherwise, T aborts.

BLS2: If there is no cycle in the potential graph, T continues waiting.

Another time-out mechanism PPCG is outlined in [7]. Once the time-out expires on T, PPCG carries out the following two steps:

PPCG1: If there is at least one cycle in PCG containing T, find the set g of the transactions involved in a number of cycles equal to or greater than the number of cycles in which T is involved, such that each transaction in g is involved in a cycle through T. If T is the youngest in g, then abort T. Otherwise choose the least expensive transaction $\tilde{T}$ from g; if the expense of T is the same as that of $\tilde{T}$, then abort T.

PPCG2: Otherwise T continues to wait with a reinitiated time-out.

Note that in the presence of expensive transactions (long-time running transactions), it is believed that some abortions are more expensive than waiting, and unnecessary abortions result in waste of system resources.

BLS is efficient, and simple to be implemented. One problem of BLS has been reported in [7] from their implementations: BLS may abort expensive transactions.

PPCG can avoid this problem, but may cause the other problems. The first problem is the computational efficiency issue. The computation of g at PPCG1 involves the computation of the number of cycles in a directed graph. There is no polynomial time bounded algorithm, so far, which may compute the number of cycles in a directed graph. So, PPCG can only apply to applications where small number of deadlocks are involved. Another problem is that if it is decided not to abort T, it is possible that there is a cycle in the potential conflict graph through T which is a real deadlock and may not be broken at that time. At the next time-out, the potential conflict graph may be changed (extended), and this remaining deadlock may again fail to be broken. Thus, this deadlock may exist forever (that is, in case that the new global transactions are continuously issued, the transactions in the deadlock may have to wait forever). Thus, the *livelock* occurs, that is, a transaction cannot proceed for an indefinite period of time. Further, PPCG may also cause transaction processing *starvation* problem, that is, the same transaction is repeatedly selected as a victim, thus causing it to abort and never finish execution. One can easily construct examples to demonstrate these livelock and starvation problems in PPCG for a cheap global transaction.

Consider that the potential conflict graph only approximately provides the deadlock information. A cycle in a potential conflict graph (PCG) could be either a false or a real deadlock. This results in uncertainty about the information of global deadlocks. To solve this uncertainty problem, a time-out mechanism is useful.

Meanwhile, we would like to have a quick response for processing each transaction, that is, the avoidance of livelock and starvation problems. Furthermore, we should avoid unnecessary abortion of expensive transactions. In the next section, we will present our deadlock resolution algorithm, which can solve the problems in BLS and PPCG.

3 A New Deadlock Resolution Algorithm

Since the combination of a PCG and time-out can confirm deadlock situation among global transactions at some level, we should abort all cycles through a transaction T in the potential conflict graph once the time-out expires on T. There are two ways to break those cycles through T: one is to abort T, another is to abort a set of other transactions which are through all these cycles. To avoid the three problems listed

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:22 from IEEE Xplore. Restrictions apply.

in Section 1, we should treat them as an integrated whole. To do this, we suggest that

> once the time-out expires on a transaction T and if there is at least one cycle in the PCG through T, instead of aborting T we may find a set of other transactions such that the abortion cost is minimized and the abortion will break all cycles through T.

After aborting a transaction T, all those submitted operations of T should be re-submitted for computation. Obviously, the system resources are wasted for the abortion of operations, which include the communication cost. So, we use the number $N(T)$ of currently submitted operations in the execution of transaction T as the abortion cost. If we use only this to measure the abortion cost, it may cause the livelock and starvation problems for cheap transactions. Thus, we also record the time $t(T)$ from the first issuing time of T up to now as another measure. The abortion cost of transaction T, denoted by $ac(T)$ is measure by

$$\alpha N(T) + \beta t(T), \tag{1}$$

where α and β give respectively the means of $N(T)$ and $t(T)$, and $\alpha + \beta = 1$. The *abortion cost* of a set M of transactions, denoted by $ac(M)$, is the sum of the abortion cost of each transaction in M. We use the following example to illustrate the necessity of our consideration.

Example 1. Suppose that when time-out expires on transaction T, the execution status of the global transactions and the potential conflict graph are illustrated by Figure 1. The abortion costs are listed as follows: $ac(T) = 8$, $ac(T_1) = 2$, $ac(T_2) = 2$, $ac(T_3) = 2$, $ac(T_4) = 3$, and $ac(T_5) = 2$. Further, suppose that T is not older than either T_1 or T_2. Then according to the algorithm BLS, T will be aborted. Clearly, the abortion cost of $\{T_1, T_2, T_4\}$ is smaller than $ac(T)$, and this abortion will also break all cycles through T. The smallest abortion cost for breaking all cycles through T is 2 which is required to abort T_3. □

3.1 An Outline of the New Algorithm

Below, we outline our global deadlock removal algorithm OVS. Once the time-out expires on a transaction T, our algorithm OVS consists of the following two steps:

Step1: Determine whether or not there are cycles containing T in the current PCG. If there are no such cycles, T continues to wait with a re-initiated time-out. Otherwise, find the set X of the transactions which are in such cycles, and go to Step 2.

Step2: Find a subset M of other transactions from X which have the smallest abortion costs such that the abortion of the transactions in M will break all cycles through T. If $ac(T)$ is smaller than $ac(M)$, then abort T. Otherwise abort M, starts to process T, and re-initiates a time-out on T.

In next subsection, we present an efficient implementation of this algorithm.

3.2 Efficiently Implementing Step 1

Step 1 corresponds to find the "strongly connected component" X of PCG containing T. This can be done by a standard algorithm in time $O(m)$ [14] where m is the number of the arcs in the PCG.

A *strongly connected component* in a directed graph G is a subgraph $\tilde{G}$ such that:

1. For each pair of vertices u and v in $\tilde{G}$, there are at least two directed paths - one is from u to v and another is from v to u.
2. For each pair of vertices u and v with u in $\tilde{G}$ and v not in $\tilde{G}$, there are no such two paths.

Note that any potential conflict graph has no arc that connects the same vertex. Consequently, the potential conflict graph has at least one cycle through T if and only if the strongly connected component X containing T has at least two vertices. Furthermore, any transaction are in a cycle containing T must be in X. So, if X contains only T, then there is no cycle in the PCG through T. Otherwise, output X and goto Step 2. Thus, Step 1 can be implemented in $O(m)$.

In the next subsection, we show that by using network flow techniques, Step 2 can be implemented in $O(n^3)$, where n is the size of the vertex set of the strongly connected component X.

3.3 Efficiently Implementing Step 2

Before presenting the implementation, we first overview the basic knowledge in network flows.

3.3.1 Networks

An $s-t$ *network* is an *arc weighted directed graph* $N = (V, A, c)$ with two distinguished vertices s and t such that $c : A \rightarrow I$ where I is the positive integer set, and all the arcs attached to s must be the going-out arcs from s and all the arcs attached to t are the incoming arcs to t. The vertex s is the *source* of N, and t is the *sink* of N. The function c is the *capacity function* of N and its value on an arc a is the *capacity* of a.

A *flow* in an $s-t$ network N is a mapping $f : A \rightarrow I$ such that:

- for each $a \in A$, $0 \leq f(a) \leq c(a)$, and
- for each vertex u other than s and t, $\sum_{a \in A_u^+} f(a) = \sum_{a \in A_u^-} f(a)$, where A_u^+ is the set of the arcs going-out from u, and A_u^- the set of arcs coming to u.

The *maximum flow* problem of an $s-t$ network N is to find a flow f in N such that

$$\sum_{a \text{ attaches to } s} f(a)$$

is maximized.

A *cut* (V_s, V_t) in an $s-t$ network $N = (V, A, c)$ is a partition on V, that is, $V = V_s \cup V_t$, $V_s \cap V_t = \emptyset$,

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:22 from IEEE Xplore. Restrictions apply.

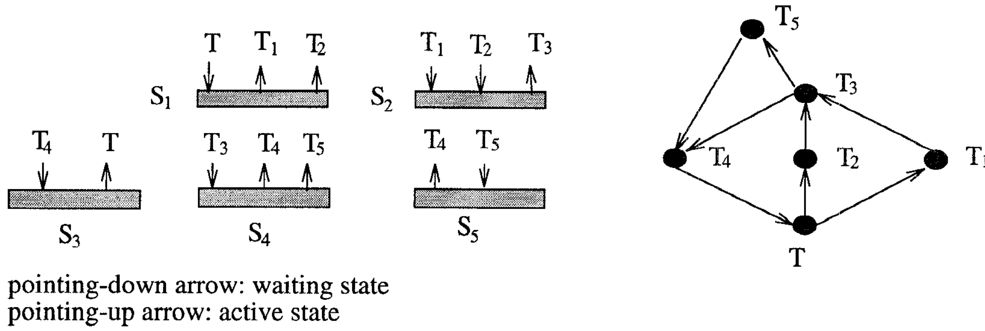

Figure 1: Example 2

and $t \in V_t$ and $s \in V_s$. The *capacity* of a cut (V_s, V_t), denoted as $c((V_s, V_t))$, is defined as the sum of the capacities of the arcs from V_s to V_t. The *minimum cut* of an $s-t$ network N is a cut (V_s, V_t) such that $c((V_s, V_t))$ is minimized.

We use $V(G)$ to denote the vertex set of a graph G, and $A(G)$ the arc set of G.

3.3.2 A Transformation

Step 2 corresponds to solving the following problem.
Minimum Vertex Cut Problem (MVCP)
Instance: Given a vertex weighted directed graph $G = (V, A, ac)$ which is strongly connected and where ac is a mapping from V to the positive integer set I, a vertex $v \in V$.
Question: How can we find a subset M of V such that the deletion of M removes all cycles through T in G, and $\sum_{u \in M} ac(u)$ is minimized?

Thanks to the developments in the maximum flow problem [14,13], MVCP can be solved in polynomial time. Below, we translate MVCP to the maximum flow problem.

For a given strongly connected and vertex weighted directed graph $G = (V, A, ac)$ and a given vertex v, we may first modify G into a $s-t$ network G_v, named by the *auxiliary network* of G with respect to v, as follows.

- For each arc a in G, assign the capacity $c(a) = \sum_{u \in V} ac(u) + 1$.
- Split v into two vertices s and t^1. All incoming arcs to v are moved to attach t^1 with the same capacity as that in G, and all outgoing arcs from v are moved to attach s with the same capacity. Add one vertex t and an arc $t^1 \to t$ with the capacity $c(t^1 \to t) = ac(v)$.
- For each other vertex $u \in V$, split it into two vertices u^1 and u^2. All incoming arcs to u are moved to attach u^1 with the same capacity as that in G, and all outgoing arcs from u are moved to attach u^2 with the same capacity. Add one arc $u^1 \to u^2$ with the capacity $c(u^1 \to u^2) = ac(u)$.

Based on Example 2, the auxiliary network of the potential conflict graph with respect to T is illustrated in Figure 2.

Obviously, the auxiliary network G_v of G with respect to v has the vertex set size $2|V|+1$, and the arc set size $|A|+|V|$, and G_v is an $s-t$ network. Below, we prove the fundamental Theorem in this paper. Clearly, a vertex set M, which breaks all the cycles through v and has the minimum overall weight, either contains only vertex v or does not contain v.

Theorem 1 *Suppose that a vertex weighted directed graph $G = (V, A, ac)$ is strongly connected, and v is a vertex. Then there is a subset $M \subseteq V$ such that the removal of M breaks all the cycles through v and $\sum_{u \in M} ac(u)$ achieves the minimum value if and only if in the auxiliary network G_v of G with respect v, any minimum cut (V_s, V_t) has the properties:*

- *the capacity $c((V_s, V_t))$ of the cut is equal to $\sum_{u \in M} ac(u)$, and*
- *the set of arcs from V_s to V_t is either $\{u^1 \to u^2 : u \in M \text{ where } v \notin M\}$ or $\{t^1 \to t\}$ in case that M contains only v.*

Proof: We first prove the "if" part by the approach of a reduction to absurdity. Suppose that there is a subset $\tilde{M}$ of V such that the removal of $\tilde{M}$ breaks all the cycles through v and $\sum_{u \in \tilde{M}} ac(u) < \sum_{u \in M} ac(u)$. Assume that $\tilde{M}$ does not contain v. (If $\tilde{M}$ contains v, then the cut $(\tilde{V}_s = V(G_v) - \{t\}, \tilde{V}_t = \{t\})$ is against the minimal property of M; thus "if" part holds.) Further, let $\tilde{A} = \{u^1 \to u^2 : u \in \tilde{M}\}$. Now we construct $\tilde{V}_s$ and $\tilde{V}_t$ in G_v as follows:

- $\tilde{V}_t = V(G_v) - \tilde{V}_s$, and
- $\tilde{V}_s = \{u^1 : u \in \tilde{M}\} \cup \{s\} \cup V_1$, where V_1 consists of vertices which are on a path, including none arc from $\tilde{A}$, in G_v from s to a vertex in $\{u^1 : u \in \tilde{M}\}$.

From the constructions of G_v, $\tilde{V}_s$ and $\tilde{V}_t$, it immediately follows that for each pair of vertices u^1 and

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:22 from IEEE Xplore. Restrictions apply.

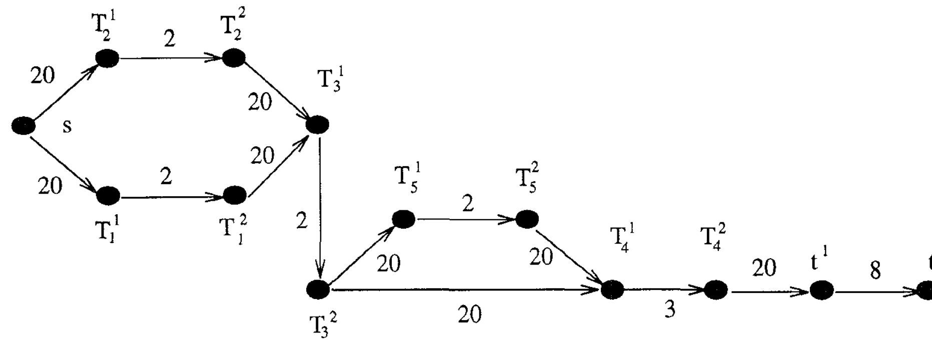

Figure 2: the auxiliary network - an s-t network

u^2 such that $u \notin \tilde{M}$, either they are all in $\tilde{V}_s$ or none of them is in $\tilde{V}_s$ (because, there is only one arc from u^1 to u^2). Clearly, $t \in \tilde{V}_t$, since the removal of $\tilde{M}$ breaks all the cycles, in G, through v. Also, each u^2 for $u \in \tilde{M}$ is in $\tilde{V}_t$. From the above facts and the constructions of $\tilde{V}_s$ and $\tilde{V}_t$, it follows that $(\tilde{V}_s, \tilde{V}_t)$ is a cut, and the set of the arcs from $\tilde{V}_s$ to $\tilde{V}$ is $\tilde{A}$. Thus $c((\tilde{V}_s, \tilde{V}_t)) = \sum_{u \in \tilde{M}} ac(u)$. It follows that $c((\tilde{V}_s, \tilde{V}_t)) < c((V_s, V_t))$. This contradicts the fact that $c((V_s, V_t))$ is the minimum cut of G_v.

Again, we prove the "only if" part through the approach of a reduction to absurdity. We may assume that M dose not contain v (If M contains v, a counter example can be immediately constructed). First, it is clear, from the above proof, that from M we may construct a cut (V_s, V_t) of G_v such that $c((V_s, V_t)) = \sum_{u \in M} ac(u)$. Suppose that there is a cut (V_s^1, V_t^1) of G_v such that $c((V_t^1, V_s^1)) < c((V_t, V_s))$, and the set π of arcs from V_s^1 to V_t^1 either contains only $t^1 \rightarrow t$ or does not contain $t^1 \rightarrow t$. Without loss of generality, we may assume that π does not contain $t^1 \rightarrow t$.

From constructions of G_v and the cut (V_s, V_t), it follows that π is a subset of $\{u^1 \rightarrow u^2 : u \in V(G)\}$ (noting that the weight of each other arc is larger than $c((V_s, V_t))$). Clearly, the removal of π means that there is no path left from s to t. From these two facts, it can be seen that the removal of M_1 will break all the cycles in G through v, where $M_1 = \{u : \text{ for a } u^1 \rightarrow u^2 \in \pi\}$. From the construction of G_v, it follows that $\sum_{u \in M_1} ac(u) = c((V_t^1, V_s^1))$. Thus, $\sum_{u \in M_1} ac(u) < \sum_{u \in M} ac(u)$. This contradicts the minimum property of M. □

3.3.3 Detailed Implementation of Step 2

According to Theorem 1, the implementation of Step 2 corresponds to finding a minimum cut from an $s-t$ network. The following Lemma from [14] says that we may apply the algorithm for solving the maximum flow problem to find a minimum cut.

Lemma 1 *For any $s-t$ network $N = (V, A, c)$, the value of the maximum flow equals the capacity of the minimum cut, and a flow f and cut (V_s, V_t) are jointly optimal if and only if*

1. *$f(u \leftarrow v) = 0$ for $u \leftarrow v \in A$ and $u \in V_s$, $v \in V_t$; and*
2. *$f(u \rightarrow v) = c(u \rightarrow v)$ for $u \rightarrow v \in A$ and $u \in V_s$, $v \in V_t$.*

Based on Theorem 1 and Lemma 1, we may carry out Step 2 in the following 4 stages.

s1: With respect to T, obtain the auxiliary network G_T of the strongly connected graph X where each vertex has been assigned a weight corresponding to the abortion cost. Go to s2.

s2: Find a maximum flow f in G_T. Go to s3.

s3: In G_T, let $V_1 = t$ and $V_0 = V(G_T) - V_1$. Then we apply breadth-first search to iteratively extend V_1 and reduce V_0 (that is, we iteratively carry out the following two operations until no changes on V_1 and V_0):

- for an arc $u \rightarrow v \in A(G_T)$, if $u \in V_0$, $v \in V_1$ and $f(u \rightarrow v) < c(u \rightarrow v)$, then remove u from V_0 to V_1; and
- for an arc $v \rightarrow u \in A(G_T)$, if $u \in V_0$, $v \in V_1$ and $f(v \rightarrow u) > 0$, then remove u from V_0 to V_1.

Go to s4.

s4: From Theorem 1 and Lemma 1, it follows that (V_0, V_1) is a minimum cut of G_T, and the set of arcs from V_0 to V_1 is in the form of either:

$\{u^1 \rightarrow u^2 : u \in M \text{ for some subset } M \text{ of } V(X)\}$ or $\{t^1 \rightarrow t\}$.

(In the later case, let $M = \{T\}$.) Abort the transactions in M.

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:22 from IEEE Xplore. Restrictions apply.

Because all the cycles through T in the potential conflict graph must be included in the strongly connected component X containing T, by combining this with Theorem 1 and Lemma 1, it follows that

the implementation of the above four steps s1 - s4 can find a subset M of the global transactions such that the abortion of M will break all cycles through T, and $ac(M)$ is minimized.

Clearly, s1 can be implemented in linear time with respect to $|A(X)|$. The most efficient algorithm for solving maximum flow problem takes time $O(|V(G_T)||A(G_T)|log\frac{|V(G_T)|^2}{|A(G_T)|})$ [13], which is slightly better than the algorithm $O(|V|^3)$ in [14] for sparse graphs. But the algorithm in [14] is much easier to implement. We suggest applying this algorithm. Thus, s2 can run in time $O(|V(G_T)|^3) = O(|V(X)|^3)$. Note that s3 and s4 can be implemented together, and take $O(|A(G_T)|) = O(|A(X)|)$ (time for breadth-first search). So, we have the following Theorem.

Theorem 2 *Step 2 in the algorithm OVG can be implemented in $O(n^3)$ where n is the number of vertices in X.*

3.4 Properties of the New Deadlock Detection Algorithms

From the preceding discussion, it follows that our deadlock detection algorithm OVS runs in time $O(m+n^3)$ where m is the number of arcs in a current PCG, and n is the number of the transactions in the deadlocks containing a given global transaction T.

The algorithm OVS will always select less expensive transactions to be aborted among the transactions whose first issuing times are the same. Furthermore, the choice of abortion cost in (1) will avoid the occurrence of livelock and starvation problems, because:

- If a transaction T is uncommitted for a very long period due to repeated abortions of T, then the abortion cost of T will be increased every time. Eventually, the abortion of T will never occur, and T will be exclusively executed.
- Every time after the other transactions deadlocking T are aborted, we can at least execute one more operation of T before new deadlocks occur on T.

4 Conclusions

In this paper, we provided a novel deadlock resolution algorithm in multidatabase systems. It always chooses less expensive victims to be aborted, and does not have livelock and transaction processing starvation problems. By using the network flow techniques, the algorithm runs in cubic time.

References

[1] R. Argrawal, M. Carey and L. McVoy, The Performance Alternative Strategies for Dealing with Deadlocks in Database Management Systems, *IEEE Transactions on Software Engineering*, 12, Sep. 1987.

[2] P. Bernstein, V. Hadzilacos and G. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[3] R. K. Batra, M. Runsinkiewicz and D. Georgakopoulos, A Decentralized Deadlock-free Concurrency Control Method for Multidatabase Transactions, *12th International Conference on Distributed Computing Systems*, 1992.

[4] Y. Breibart, W. Litwin and A. Silberschatz, Multidatabase Concurrency Control Systems, *Technical Report*, 154-89, Department of Computer Science, University of Kentucky, 1989.

[5] Y. Breibart, A. Silberschatz, and G. Thompo- son, Reliable Transaction Management in a Multidatabase System, *SIGMOD Record*, 1990.

[6] Y. Breibart, W. Litwin and A. Silberschatz, Deadlock Problems in a Multidatabase Environment, IEEE Data Engineering, January 1991.

[7] O. Bukhres, J. Alm and N. Boudriga, A Priority-Based PCG Algorithm for Global Deadlock Detection and Resolution in Multidatabase Systems, *First International Workshop on Interoperability in Multidatabase Systems*, IEEE CS press, 1993.

[8] S. Ceri and G. Pelagatti, *Distributed Database Principles and Systems*, McGraw Hill, 1984.

[9] A. K. Elmagarmid and W. Du, A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems, *IEEE Proceedings of the 6th International Conference on Data Engineering*, 1990.

[10] D. Georgakopoulos, M. Runsinkiewicz and A. Sheth, On Serializability of Multidatabase Transactions Through Forced Local Conflicts, *IEEE Proceedings of the 7th International Conference on Data Engineering*, 1991.

[11] V. Gligor and R. Popescu-Zeletin, *Concurrency Control Issues in Distributed Heterogeneous Database Management Systems*, Tutorial: Distributed Database Management, 1985.

[12] S. Mehrotra, R. Rastogi, Y. Breibart, H. F. Korth and A. Silberschatz, The Concurrency Control Problem in Multidatabases: Characteristics and Solutions, *ACM SIGMOD*, 1992.

[13] B. M. E. Moret and H. D. Shapiro, *Algorithms from P to NP, Volume 1: Design and Efficiency*, Benjamin/Cummings, 1990.

[14] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall Publish.

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:22 from IEEE Xplore. Restrictions apply.