Deadlock Problems in a Multidatabase Environment

Yuri Breitbart†
Computer Science Dept.
University of Kentucky
Lexington, KY 40506

Witold Litwin††
Universite Paris Dauphine
Pl du Mal de Lattre de Tassigny
Paris, Cedex 16, France

Avi Silberschatz†††
Computer Science Dept.
University of Texas
Austin, TX 78712

Abstract

A deadlock detection algorithm and a deadlock prevention algorithm in a multidatabase environment are introduced. The deadlock detection algorithm is based on the potential conflict graph (PCG) introduced in [BRE190]. The deadlock prevention algorithm is based on the value date protocol discussed here. We prove correctness of both algorithms and briefly discuss their performance.

1. Introduction

A transaction management scheme in a multidatabase environment must ensure the global consistency and freedom from deadlocks of the multidatabase system in the presence of local transactions (i.e., transactions executed outside of the multidatabase system control) and in the face of the inability of local DBMSs to coordinate execution of multidatabase transactions (called global transactions), under the assumption that no design changes are allowed in local DBMSs.

Designing such a scheme appears to be a difficult problem. The difficulties stem from a requirement that each local DBMS operate autonomously and that local transactions are permitted to execute outside of the multidatabase system control. The autonomy of local DBMS assumes that a local DBMS may abort a global transaction at local site at any time.

The multidatabase transaction management problem was first discussed by Gligor and Popescu-Zeletin [GLIG85]. They outlined basic requirements for a transaction management system that assures consistent data update in a heterogeneous database environment and pointed out the inherent difficulties of data update in such systems. Since then the problem was extensively studied in two basic directions: restricted autonomy of the local DBMS's [PU87] and complete preservation of local DBMS autonomy ([ALON87], [DU89] and [BRE188]), [BRE190]).

In spite of many attempts, however, there is still no satisfactory transaction management scheme for multidatabase systems with no restrictions on local DBMSs. One way to devise such a scheme is to impose restrictions on the structure of the local concurrency control mechanisms. In [BRE190], a multidatabase system was discussed where each local DBMS uses the strict 2PL protocol (i.e., a transaction may not receive any lock after it releases at least one of its locks, and a transaction releases its locks only after it commits or aborts). A reliable multidatabase transaction management scheme was proposed that ensures both global serializability and freedom from global deadlocks. However, the algorithm proposed in [BRE190] imposes significant restrictions on the way global and local transactions can access the multidatabase data items.

In this paper we continue our study of transaction management in multidatabases systems. We assume that such a system consists of local DBMSs each of which is using the strict 2PL protocol. For the purposes of our study we disallow failures. The use of the strict 2PL protocol by each local DBMS ensures global database consistency [BRE188]). Unfortunately, the second major issue of multidatabase transaction management - global deadlocks - still remains to be resolved.

Generally, there are two techniques to deal with deadlock situations: deadlock detection and deadlock prevention. In a multidatabase environment one cannot be certain whether a global deadlock has occurred, or if it just takes a local DBMS a very long time to execute global transaction operations (we assume that local deadlocks are detected and broken by a local DBMS). Thus, global deadlock detection and global deadlock prevention terminology needs to be refined in a multidatabase environment. To clarify this, we will assume that if in a multidatabase system, a global deadlock may occur, then the DBMS must employ an algorithm for detecting this situation and recovering from it. Such an algorithm is referred to as a deadlock detection algorithm.

On the other hand, if a multidatabase system ensures that no global deadlocks may occur, then the algorithm will be called a deadlock prevention algorithm.

In this paper we present both a deadlock detection algorithm and a deadlock prevention algorithm. The deadlock detection algorithm is based on the potential conflict graph (PCG) introduced in [BRE190]. The deadlock prevention algorithm is based on the value date protocol discussed here. We prove correctness of both algorithms and briefly discuss their performance.

The remainder of the paper is organized as follows. In Section 2 we define the multidatabase model used in
this paper. Section 3 contains our deadlock detection algorithm and the proof of its correctness. Section 4 contains discussion of the value-date protocol and the proof that a multidatabase system with each local DBMS using the value-date protocol retains global database consistency. Section 5 contains the deadlock prevention algorithm that uses value date protocol. Section 6 concludes the paper.

2. The MDBS Model

The MDBS used in this paper consists of global and local components, as depicted in Figure 1. Interactions with the database are conducted by user programs called transactions. A transaction is simply a sequence of read (denoted by r) and write (denoted by w) operations defined on the various data items associated with the database. We call two operations conflicting if they are from different transactions, defined on the same data item and at least one of them is a write operation. We introduce the notion of conflict equivalent schedules in the usual manner [BERN87] and define a serializable schedule as one that is equivalent to a serial schedule. We use serializability as a correctness criterion for the MDBS and local DBMS concurrency control mechanisms.

A global transaction is a transaction that is submitted to MDBS and is executed under its control. A local transaction, on the other hand, is a transaction submitted to a local DBMS, outside of the MDBS control.

The global transaction manager (GTM) controls the execution order of global transaction’s operations. The GTM assigns a timestamp to each global transaction that enters the system. The timestamp of a global transaction T is denoted by ts(T). For each operation, the GTM selects a local site (or a set of sites) where the operation should be executed. For each site that was selected, the GTM allocates a server, one per transaction per site. Each server is responsible for translating the global read and write operations into the appropriate query language operations of the local DBMS, and for submitting them for execution to the local DBMS. Each time a global transaction submits the next operation, the GTM transmits that operation to the server. Each server eventually reports to the GTM the results of the execution. The local DBMS treats each server as a local transaction. It merges operations of these transactions with other local transactions that are submitted outside of MDBS control.

The MDBS model discussed in this paper is based on the following assumptions:

1. No changes can be made to the local DBMS software. The global transaction manager is aware of the fact that local transactions may be executed at local sites. However, the GTM is not aware of any specifics of the local transactions, nor which data items they access.

2. A local DBMS is not able to distinguish between local and global transactions which are active at the local site. A local DBMS at one site is not able to communicate directly with another local DBMS located at some other site in order to synchronize the execution of a global transaction that is active at both sites.

3. Each local DBMS uses the two-phase locking (2PL) protocol which ensures local serializability [ESWA76]. Locks are released either explicitly by the transaction using an unlock operation, or by the local DBMS as a result of executing the end of transaction (eot) operation. In addition, each local DBMS ensures freedom from local deadlocks.

4. For each global transaction, the end of transaction (eot) operation is broadcasted by the GTM to each local site at which the transaction was active only after each transaction’s server has reported to the GTM the completion of all the transaction’s operations, and the GTM has submitted all transaction’s read/write operations to the appropriate servers.

As a result of the first two assumptions, the GTM is the only mechanism that is capable of coordinating global transaction execution at different local sites. However, any such coordination should be conducted in the absence of any local DBMS control information. Thus, the GTM should make the most pessimistic assumptions about the behavior of local transaction managers in order to ensure global database consistency and freedom from global deadlocks.

There are several alternatives for submitting the global read/write operations to servers. One alternative is to submit all the operations of a single global transaction to be executed at the same local site in one single message. This alternative is frequently used for global transactions whose operations at one local site do not depend on data values obtained from another local site. Another alternative is to submit one global operation at a time. The next transaction’s operation is not sent until the current operation of the same transaction has been completed at each local site. In either case, under Assumption (4), we require that the end of transaction operation will not be sent to all the local sites at which the transaction was executing until each local site reports the completion of the transaction’s read/write operations.
3. Multidatabase Deadlock Detection Algorithm

In [BREI88a] we proved that any MDBS that satisfies Assumptions (1) - (4) always retains global database consistency for any combination of local transactions. Since any local DBMS detects and recovers from local deadlocks, any local schedule is deadlock-free. This means that any local wait-for-graph [KORT86] does not contain a cycle. In this environment, however, there is the possibility of global deadlocks.

Consider a set of global and local transactions that contains at least two or more global transactions. If each transaction in this set waits for a local lock allocated to another transaction in the set, then each transaction in the set is waiting. Therefore, no transaction in the set can release its local locks which are needed by other transactions in the set. We will call such situation a *global deadlock*. The MDBS must be able to detect and break global deadlocks. Since the set of local transactions is not known to the MDBS, detecting global deadlock situations is not simple. To illustrate this, consider the following example.

**Example 1:** Consider a multidatabase that consists of data items a and b at site S1, and c and d at site S2. Let us further assume that the following global transactions are submitted to the MDBS:

\[
T_1; \pi_1(a) \pi_1(d) \text{ eot} \backslash
T_2; \pi_2(c) \pi_2(d) \text{ eot}
\]

In addition to the global transactions, at each of the local sites the following local transactions are submitted:

\[
L_3; w_3(b) w_3(a) \text{ eot} \backslash
L_4; w_4(d) w_4(c) \text{ eot}
\]

The GTM receives \( \pi_1(a) \) and sends it to \( S_1 \). It then receives \( \pi_2(c) \) and sends it to \( S_2 \). The GTM receives responses from sites \( S_1 \) and \( S_2 \), respectively, that the read locks have been obtained for a and c by \( T_1 \) and \( T_2 \), respectively.

Transaction \( L_3 \) now requests a write lock for b and obtains it, and then requests a write lock for a and must wait for \( T_1 \) to release that lock. Transaction \( L_4 \) requests a write lock for d and obtains it and then requests a write lock for c and must wait for \( T_2 \) to release that lock.

Since the GTM is not aware of \( L_3 \) and \( L_4 \) the operations \( \pi_1(d) \) and \( \pi_2(b) \) are submitted to local sites. This situation results in a global deadlock state, since \( T_1 \) is waiting for \( L_3 \) which, in turn is waiting for \( T_2 \) at site \( S_2 \), and \( T_2 \) is waiting for \( L_4 \) which in turn is waiting for \( T_1 \) at site \( S_1 \).

We now present our global deadlock detection algorithm and prove its correctness. In order to detect global deadlocks we will further restrict our model. In addition to the assumptions of Section 2, we also assume that the GTM does not send a transaction operation to a server until the server responds that it has obtained the required local locks for the previous transaction's operation.

Deadlocks in centralized and distributed homogeneous database environments have been extensively studied and various schemes based on the *wait-for-graph* concept were introduced to detect deadlocks among transactions. Here we use a similar technique.

**Lemma 1:** Let \( GL \) be a set of global and local transactions that contain at least two global transactions. Let \( \gamma \) be the union of all local *wait-for-graphs*. A *global deadlock* exists if and only if \( \gamma \) contains a cycle.

In order to detect global deadlocks, it is necessary for the MDBS to have access to the various local wait-for-graphs. Since this information is not available to the MDBS, it is necessary to devise a different method for approximating the union of the local wait-for-graphs. This approximation may result in the detection of false deadlocks, but must ensure that no global deadlock will be missed. To achieve this, we introduce a new type of graph called a *potential-conflict-graph* (PCG). Before defining the graph, we must first introduce the notion of a global transaction being *active* or *waiting* at a site.

A transaction \( T_i \) is *active at site* \( S_j \) if it has a server at \( S_j \) and the server is either 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 transaction \( T_i \). 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 transaction that is either *active* or *waiting* at a local site is said to be *executing* at the site.

We assume that each global transaction can be in the waiting status at most at one site. This restriction obviously holds for nonreplicated global databases. It can also hold for a replicated global database as well. In the latter case, if the transaction should execute a write operation on a replicated data item, then local write lock requests should be submitted in sequence. The next site's write lock request is not sent until the local write lock request from the previous site is satisfied.

A *Potential Conflict Graph* (PCG) is a directed graph with a set of vertices \( V \) consisting of all global transactions executing in the system, and a set of edges \( E \) such that edge \( T_i \rightarrow T_j \) is in \( E \) if and only if there is a site at which \( T_i \) is waiting and \( T_j \) is active.

A PCG changes whenever a transaction at some site changes its status from active to waiting or vice versa. If a transaction is waiting at site \( S_j \), then it waits for the local DBMS to allocate local locks required to perform the transaction operation. After the transaction has received the requested local locks, the transaction's status at the site is changed to active. Thus, when a transaction has completed all its operations at all local sites at which it was executing, and it receives the eot command, its status at all such sites is active and remains active until the transaction either commits or aborts. After the transaction completes its execution, it is removed from the PCG along with all edges incident to it.

**Example 2:** Consider a multidatabase consisting of sites \( S_1, S_2, S_3, S_4, S_5, \) and \( S_6 \). Let \( T_1, T_2, T_3, T_4, T_5, T_6, T_7, \) and \( T_8 \), be global transactions that are either in the waiting or active state.

![Figure 2](image-url)
Figure 3

at some local site as shown in Figure 2. In Figure 2, if a transaction is waiting at some site, this is indicated by having an arrow directed from the transaction to the site. If a transaction is active at some site, this is indicated by having the arrow from the site to the transaction. Thus we can see, for instance, that at site $S_2$ transaction $T_6$ is waiting, and transaction $T_7$ is active.

For the transaction configuration shown in Figure 2, the PCG is shown in Figure 3. Assume now that at site $S_2$ transaction $T_1$ enters the waiting state. Then the PCG is changed accordingly as shown in Figure 4.

Figure 4

Lemma 2: If the PCG does not contain a cycle, then there is no possibility of a global deadlock, provided that any local wait-for-graph is acyclic.

We are in a position now to present our global deadlock detection algorithm. The algorithm is invoked by the GTM whenever some global transaction $T_i$ is in the waiting state at some local site $S_j$ for more than some prespecified timeout period. The algorithm consists of the following two steps.

1. If there is a cycle in the PCG, then:
   1.1 Let $(T_{i1}, ..., T_{im})$ be the set of all global transactions that are active at $S_j$ and appear in at least one cycle with $T_i$.
   1.2 If $ts(T_i) < \min(ts(T_{i1}), ..., ts(T_{im}))$ then $T_i$ continues to wait at $S_j$ with a reinitiated timeout, otherwise $T_i$ is aborted at all sites.

2. If there is no cycle in the PCG, then $T_i$ continues to wait at $S_j$ with a reinitiated timeout.

After the server has responded that the local lock has been obtained, the status of $T_i$ at $S_j$ is changed to active and the PCG is modified accordingly.

Let us elaborate on the structure of the algorithm above. The GTM uses a timeout strategy to decide whether a global deadlock has occurred. If the GTM does not receive a message indicating the operation result, then either a global deadlock has occurred, or it takes the local DBMS longer than the prespecified timeout period to allocate a lock for the transaction. We are using the wait-die scheme [ROSE78] to decide whether $T_i$ should continue to wait for a local lock after the timeout period has expired, or if it should abort (see Step 1.2 of the algorithm). We assume that the older the global transaction is, the more of its operations were executed. Therefore, we let the older transaction wait for a local lock in order to avoid costly aborts. If the system is in a global deadlock, the GTM will eventually detect it when one of the younger transactions in the cycle requests a local lock. Then transaction $T_i$ will obtain the requested local lock in the subsequent timeout periods.

The global transaction that is aborted is restarted later on with the same timestamp. This ensures that no livelock situation will occur, as will be proven in Theorem 1. To avoid repeated aborts for younger transactions, we do not restart the aborted transaction until after the oldest transaction from the site has completed.

Example 3: Consider again the situation depicted in Example 2. The PCG is shown in Figure 5. Suppose that $T_1$ is older than $T_2$ (i.e., $ts(G_1) < ts(G_2)$), and that the GTM does not receive a response from the server of $T_1$ within the prespecified timeout period. The global deadlock algorithm is thus invoked for transaction $T_1$ waiting at $S_2$. According to step 1.2 of the algorithm, $T_1$ continues to wait, since $ts(G_1) < ts(G_2)$. Afterwards, the timeout period expires for $T_2$ and the algorithm is invoked for transaction $T_2$ waiting at site $S_1$. This time $T_2$ is aborted, since it is younger than $T_1$. After $L_4$ obtains the lock on data item $c$ (as a result of the abort of $T_2$), it completes its execution and releases its local locks. Finally, $T_1$ obtains the local lock on $d$, successfully completes its execution at both sites, and exits the system. The GTM restarts $T_2$ and this time the transaction successfully completes.

Figure 5

Theorem 1: Our deadlock detection algorithm ensures freedom from global deadlocks. Every global transaction in the system eventually either commits or aborts.

Preliminary results of the performance study that is being currently conducted indicate that the deadlock detection algorithm presented here does not require a significant overhead. In fact, it is not called that often,
provided that a timeout is carefully chosen. One cannot choose a timeout too large, since it significantly impacts the transaction throughput. On the other hand, the timeout should not be chosen too small either, since in that case it may cause unnecessary aborts of global transactions. Our results also indicate that our algorithm does reduces a number of global transactions aborts compared with a simple timeout deadlock detection algorithm without using potential conflict graph. Since global transactions aborts are rather costly in a multibase environment, our findings provide good arguments for using non trivial deadlock detection algorithms as oppose to use just a simple timeout mechanism.

4. The Concept of Value Date

In [LITW89] a new concurrency control protocol was proposed based on the concept of value date. Here we propose a modified version of the protocol that allows shared reads that were not allowed in the original protocol.

We assume that each data item a is appended (virtually or physically) with a read value date attribute - rvd(a) and a write value date attribute - wvd(a). Read and write value dates are kept by the scheduler. These values may be updated by the scheduler with each new operation performed on the data item. A write value date has the semantics of declaring the moment in time (the date) after which the value of data item a can be accessed by any transaction. If, however, the scheduler assigns a write value date on behalf of transaction T, then T and only T can access a at any time. If a transaction issues a read or write operation on a data item that has a write value date larger than the transaction's value date, then the DBMS assumes that the operation arrived too late and, consequently, the transaction is aborted. Similarly, the semantic of a read value date is to indicate a moment at which the data item can be safely changed by other transactions. If a transaction issues a read operation on a data item that has a read value date larger than the transaction's value date, then the DBMS assumes that the write operation arrived too late and consequently the transaction is aborted.

The concept of value date has been used in real life for many years. An example is the way banks deal with checks. The account is credited instantly when the cashier or an ATM that processed the check finishes typitng it in. The account value, however, is considered "safe" usually only after a certain number of days have passed. In the meantime, the account value is "unsafe", as the check may return unpaid, implying that there are certain restrictions to apply on the type of operations the customer may invoke on the account. For example, if a customer deposits a $1,000 check to an account with the current balance of $100, and later asks to withdraw $900 before the value date, a typical bank will refuse this transaction.

Each transaction upon submission to the system is assigned value date - V_d(T) - by the scheduler. It is assigned in such a way that actual date - A_d - is always less than V_d(T). Intuitive meaning of the transaction value date is to indicate a moment at which transaction should be committed. If a transaction is not completed by its value date, then it is aborted. In any event, if A_d > V_d(T), then T is either committed or aborted.

We assume that no two transactions can have the same value date assigned by a scheduler. Thus, V_d(T) can serve as a transaction identifier. In the sequel, if T_1 is a transaction, then i is its value date. We assume that initial values for A_d as well as for any read and write value dates of any data item are zeros and the value date for any transaction is more than zero.

We are in a position now to present the value date concurrency control algorithm, referred to as VDA. The algorithm works as follows:

1. If V_d(T) of the transaction that submitted an operation for execution is smaller than or equal to A_d, then T is aborted (it should have been completed before V_d(T)).

Otherwise:

1.1. If an operation submitted for execution is r(a).

If A_d ≥ wvd(a), then
execute r(a);
rvd(a) = max(rvd(a), V_d(T))
If A_d < wvd(a), then
If V_d(T) ≥ wvd(a) then
r(a) waits until after A_d = wvd(a),
else abort T

1.2. If an operation submitted for execution is w(a).

If A_d ≥ rvd(a) and A_d ≥ wvd(a), then
execute w(a);
wvd(a) = max(wvd(a), V_d(T))
If A_d ≥ rvd(a) and A_d < wvd(a), then
If V_d(T) ≥ wvd(a), then w(a) waits until after A_d = wvd(a) else abort T

If A_d < rvd(a) and A_d ≥ wvd(a), then
If rvd(a) = V_d(T) then execute w(a);
wvd(a) = max(wvd(a), V_d(T))
else If V_d(T) ≥ rvd(a) then w(a) waits until after A_d = rvd(a) else abort T

If A_d < rvd(a) and A_d < wvd(a), then
If rvd(a) = V_d(T) then execute w(a);
wvd(a) = max(wvd(a), V_d(T))
else If V_d(T) ≥ rvd(a) then w(a) waits until after max(rvd(a), wvd(a)) else abort T

Let us consider several examples.

Example 4: Consider two transactions:

T_1: r_1(a)r_2(c)
T_2: r_2(b)r_3(g)w_2(a)

Let transaction T_2 start first and execute r_2(a). Thus, rvd(a) = 2 and wvd(a) = 0. When T_1 submits to the scheduler its first operation, it also will be executed, since the wvd(a) did not change after T_2 reads a. After that we may assume that T_1 continues its execution and executes r_1(c). After T_1 completes its execution, T_2 submits r_2(b) and executes it and finally it submits w_2(a). At this time wvd(a) = 0 and rvd(a) = 2. Since V_d(T_2) = rvd(a), we execute w_2(a). □

Example 5: Consider two transactions:

T_1: r_1(b)r_1(a)
T_2: r_2(a)w_2(a)r_3(b);

Let us assume that the following schedule has been generated so far by the scheduler:

S: r_2(a)w_2(a)r_1(b)
At this time the $rvd(a) = wvd(a) = 2$, $rvd(b) = 1$, $wvd(b) = 0$, and $A_d = 0$. $T_j$ submits $r_j(b)$ to the scheduler and according to the algorithm the scheduler lets it proceed. After that $T_i$ submits $r_i(a)$. Since at this time $wvd(a) = 2$ and $V_d (T_i) = 1$, transaction $T_i$ is aborted.

On the other hand, the following schedule can be produced by the value date algorithm:

$$ S: r_j(b)r_i(b)r_i(a)\text{w}_j(a)\text{w}_j(b) $$

Thus, we were able to execute these transactions concurrently without aborting any of them.

A schedule $S$ is called rigorous [BRE90a] if and only if for any two conflicting operations $op_k(x)$ and $op_l(x)$ from $T_i$ and $T_j$, respectively, if $op_k(x)$ precedes $op_l(x)$ in $S$, then $T_i$ completes in $S$ before $op_l(x)$. In [BRE90a] we proved that each rigorous schedule is serializable.

**Theorem 2:** The algorithm described above always generates rigorous schedules that are deadlock and livelock free.

If we replace Assumption (3) with the assumption that each local DBMS uses the VDA concurrency control method, then combining Theorem 2 with results from [BRE90a], we obtain that any such MDBS retains global database consistency under any combination of local transactions.

Let us denote a class of schedules generated by the VDA scheduler as VDA and a class of schedules generated by the strict 2PL scheduler as S2PL. The following theorem holds.

**Theorem 3:** S2PL = VDA.

Thus, the VDA scheduler generates the same class of schedules as the strict 2PL scheduler but the VDA does not use data locks.

5. Value Date Based Deadlock Prevention Algorithm

We assume that the GTM uses the VDA protocol but we do not assume that each local DBMS uses the VDA protocol. We assume, however, that the model satisfies Assumptions (1)-(4) of Section 2. The value of $V_d(T_i)$ for each global transaction $T_i$, is sent to each site at which the transaction will execute. (As no two global transactions have the same value date, this value is also the transaction identification - TID.) The GTM applies the VDA protocol to detect and avoid conflicts between local transactions competing for the same data item. This requires that GTM maintain its own concurrency control on data items in the multidatabase. The GTM’s concurrency control mechanism uses a table that contains for each data item its read and write value dates. An operation of a global transaction is sent to a server at the local site iff it is allowed according to the VDA policy. If the GTM decides to abort $T_i$, a request to abort is broadcast to all the sites at which $T_i$ was executing. If a site decides to abort $T_i$, the request is forwarded to the GTM that broadcasts the corresponding actions to all the sites at which $T_i$ was executing.

If $T_i$ comes up to its end at all sites, then each server may report the success to the GTM which, in turn, may send the $eot$. The $eot$ may be $V_d(T_i)$ itself. Global database consistency is assured [BRE88a]. A new possibility is that the transaction’s value date may also serve as an implicit $eot$. Unless a server issued a request to abort $T_i$, broadcasted by the GTM before $A_d$.

the local server should then issue the $eot$ itself for the local DBMS. If we have $n$ sites involved in the $T_i$’s execution, this option saves about $2n$ messages.

It is interesting to note that if the multidatabase model satisfied only assumptions (1), (2) and (4) of Section 2, the VDA protocol alone would not assure global database consistency, as Example 6 below demonstrates.

**Example 6:** Consider a multidatabase with the VDA protocol, but each local DBMS does not use the strict 2PL protocol. Let the multidatabase consist of data items $a$ and $b$ at site $S_1$, and $c$ at site $S_2$. Let us further assume that the following global transactions are submitted to the MDBS: $T_1: r_i(a) r_i(c)\quad T_2: r_j(d) r_j(b)$

In addition to the global transactions, at each of the local sites the following local transactions are submitted: $L_3: w_i(b) w_i(a)\quad L_4: w_d(c) w_d(a)$

Since global transactions do not have any data items in common, the VDA protocol trivially applies. Thus, the following situation may occur. The GTM receives $r_i(a)$ and sends it to $S_1$. It then receives $r_j(d)$ and sends it to $S_2$. The GTM receives responses from sites $S_1$ and $S_2$, respectively, that the read operations have been completed.

Since the strict 2PL protocol is not used by the local DBMS, transactions $L_3$ and $L_4$ can start and successfully complete at $S_1$ and $S_2$, respectively.

Since the GTM is not aware of $L_3$ and $L_4$ the operations $r_j(d)$ and $r_j(b)$ are submitted to local sites. The operations are successfully completed and as a result we obtained two local schedules at site $S_1$ and $S_2$, respectively that violate a global database consistency. At site $S_j$, $T_j$ precedes in a serializable order $T_2$, while at $S_2$, $T_2$ precedes in a serializable order $T_1$. Thus, a global serialization graph does contain a loop.

Unlike the situation in the centralized environment, it may happen, however, that a global deadlock occurs. The reason is that global transactions may interact with local transactions without value dates and the GTM is not aware of these transactions. These transactions may either dynamically relate global transactions, according to the VDA policy. Other transactions may then proceed. The GTM may be the only one allowed to abort a global transaction, broadcasting the corresponding actions to the servers. However, as the servers have the same value date, they may be allowed to abort the transaction autonomously, saving messages.

In this and similar cases, value dates are compared to the actual date — $A_d$ given by local clocks of local DBMS’s, instead of the single clock of the GTM. These clocks may drift. We assume that the drifts are small compared to the length of the transaction, (e.g., millisecond with respect to seconds or frequently minutes
for interactive transactions). The clocks are nowadays rather precise because of various other needs, email in particular. Also, cheap devices, available even for IBM-PCs, may synchronize a clock on the waves of an atomic one.

6. Conclusion

We have presented two approaches to deal with global deadlocks that could occur in multibase environment.

The value date approach does not require that the GTM will send one operation at a time to a site. Thus this approach provides more autonomy to sites, requires fewer messages to be exchanged, and is therefore a more efficient approach. Value dates also protect against a number of other causes of waiting that would not appear on the potential conflict graph.

On the other hand, the algorithm of Section 3 may detect a deadlock earlier than the value date technique. It would detect it indeed by the end of one of the blocked operations, while the VDA protocol would wait until the actual time reaches the closest value date of transactions involved. Furthermore, the VDA policy requires the GTM to maintain its own concurrency control on data in use by current transactions. This may be a table with the data names in use by the on-going global transactions. In contrast, the algorithm of Section 3 has to maintain the PCG that changes with each successful operation. Which method is more advantageous in this or that practical context from the point of view of these aspects is an open research issue.

Both methods also exhibit some similarity. The timestamp may indeed be seen as a value date, basically the same for all the transactions. The algorithm of Section 3 is called when the actual time reaches the value date. It eventually extends the value date, instead of automatically aborting the corresponding transaction, or confirms that an abort is necessary. This may be advantageous in some situations. It also seems to indicate that it may be interesting to combine both methods into a uniform framework. Again, the corresponding study is an open question.

References


