VLDB Journal, 1, 1-39 (1992) Peter Scheuermann, Editor

Transaction Management Issues in a Failure-Prone Multidatabase System Environment

Yuri Breitbart, Avi Silberschatz, and Glenn R. Thompson

Received June 16, 1990; revised version received July 3, 1991; accepted October 3, 1991.

Abstract. This paper is concerned with the problem of integrating a number of existing, off-the-shelf local database systems into a multidatabase system that maintains consistency in the face of concurrency and failures. The major difficulties in designing such systems stem from the requirements that local transactions be allowed to execute outside the multidatabase system control, and that the various local database systems cannot participate in the execution of a global commit protocol. A scheme based on the assumption that the component local database systems use the strict two-phase locking protocol is developed. Two major problems are addressed: How to ensure global transaction atomicity without the provision of a commit protocol, and how to ensure freedom from global deadlocks.

Key Words. Algorithms, performance, reliability, serializibility, deadlock recovery, transaction log.

1. Introduction

A multidatabase system (MDBS) is a software package that allows user transactions to invoke retrieval and update commands against data located in heterogeneous hardware and software environments. A multidatabase system is built on top of a number of existing database management systems (DBMS's) that are being integrated into a single MDBS. A multidatabase environment supports two types of transactions:

- **local transactions**, those that are executed by a local DBMS outside of the MDBS control. Local transactions, by definition, can access data located at only a single site.

Yuri Breitbart, Ph.D., is Professor, Department of Computer Science, University of Kentucky, Lexington, KY 40506. Avi Silberschatz, Ph.D., is Endowed Professor, Department of Computer Science, University of Texas, Austin, TX 78712. Glenn R. Thompson, Ph.D., is Research Supervisor, Amoco Production Company, Tulsa, OK 74102. (Reprint requests to Dr. Y. Breitbart, Dept. of Computer Science, 915 Patterson Office Tower, Lexington, KY 40506-0027, USA.)

- **global transactions**, those that are executed under the MDBS control. Global transactions may access data located at several sites.

The MDBS is aware that local transactions are being executed by local DBMS's, but has no control over them.

Most commercially available DBMS's (IMS, IDMS, ADABAS, SQL/DS, DB2, etc.) were created as stand alone systems. As such, they do not provide mechanisms to allow for the smooth integration into an MDBS environment. Recently, however, a number of commercial products were introduced to support distributed transaction execution across a number of different DBMS's (Duquaine, 1989, 1990; Simonson, 1990; Sugihara, 1987; Bever, 1989). While interoperability is being addressed by several DBMS vendors, their approach seems to be either customized to specific systems or to impose various restrictions. Moreover, such approaches are not easily applicable to existing transaction programs without major rewriting. For purely economic reasons, however, it can hardly be expected that users will undertake a massive rewrite of their old applications in order to take advantage of new DBMS capabilities as they become available.

We assume in this paper that a global transaction executing at a local site is considered a regular user transaction as far as the DBMS is concerned, and as such it cannot access DBMS control information (such as a wait-for-graph, a schedule, a DBMS log, etc.). As a consequence, each DBMS integrated by the MDBS operates autonomously, without the knowledge of either other DBMS's, or the MDBS system. Thus, the various local DBMS's cannot exchange any control information that would allow them to coordinate the execution of a global transaction accessing data at different local sites. This implies that any local DBMS has complete control over all transactions (global and local) executing at its site. This form of autonomy, referred to as *execution autonomy*, means that a DBMS can abort a transaction at any point in time during its execution.

The execution autonomy property, in effect, precludes the participation of the various local DBMS's in the execution of an atomic commit protocol (e.g., the two phase commit protocol). Indeed, any atomic commit protocol requires that after a transaction executing at a local site has reached a decision to commit, the decision cannot be changed either by the transaction or by the local DBMS (Bernstein, 1987). On the other hand, the execution autonomy property states that a DBMS can abort a transaction at any time prior to the execution of the transaction's **commit** operation (this may happen even after a local transaction has submitted the commit operation to the local DBMS but before the DBMS has executed it).

The preservation of execution autonomy is crucial in an MDBS environment as this allows a local DBMS to have complete control over the execution of various transactions. Integration of local DBMS's into a MDBS system does not infringe on a local system's right to manage sites without consideration of an MDBS system.

For example, suppose that we tie together a bank DBMS and the stock market DBMS. We can now have a global transaction that can access information at both local DBMS's (for example, buy some stocks and move money from the bank to cover expenses). Execution autonomy in this environment implies that at any point, and for whatever reason (e.g., local locks held for too long), either one of these DBMS's can abort the execution of the global transaction.

Preservation of execution autonomy leads to a cardinal technical issue that we address in this paper. It is well known that any atomic commit protocol is subject to blocking in the case of some types of failure. This results in a situation where some transactions are blocked, so they can neither be committed nor aborted until the failure has been repaired. In this paper we investigate ways to ensure both global transaction atomicity as well as global serializability in an environment where blocked transactions can be aborted. Of course, this cannot be achieved without some compromises (as was very eloquently argued in Bernstein, 1987).

Transaction management problems in a multidatabase environment were first discussed in (Gligor, 1986). The authors pointed out that its inherent difficulties stem from the requirement that each local DBMS operate autonomously and that local transactions are permitted to execute outside of the MDBS system control. Since then, the problem was extensively studied in two basic directions: restricted autonomy of the local DBMS's (Elmagarmid, 1987; Pu, 1987; Sugihara, 1987) and a complete preservation of local DBMS autonomy (Alonso, 1987; Breitbart, 1987, 1988; Du, 1989).

Restricted autonomy (Elmagarmid, 1987; Pu, 1987) implies that the local DBMS's can share with the MDBS their local control information (for example, local schedules). This assumption, however, requires design changes in local DBMS's and as a result reduces the multidatabase transaction management problem to the same problem as in the homogeneous distributed database environment with hierarchical organization of local sites. This problem has been extensively studied and is by now well understood. On the other hand, the study of restricted autonomy is important in that it can determine the minimal information the local DBMS's need to share to ensure a correct execution of local and global transactions.

Research on complete preservation of a local DBMS autonomy has two important threads:

- No assumptions are made concerning the nature of the various local DBMS concurrency control mechanisms (i.e., whether a 2PL scheme or a timestamp

scheme is being used). This generality, however, has its downside–ensuring multidatabase consistency may result in a large number of global transaction aborts (Breitbart, 1988). That may become prohibitively expensive, especially in a geographically distributed multidatabase system.

- It was assumed that no failures can occur in the MDBS system during global transaction processing. We are not aware of any systematic treatment of fault-tolerant transaction management in a multidatabase environment that allows failures to occur during any stage of the global transaction processing.

In this paper, we focus our attention on the problem of global transaction management in a failure prone environment, where each local DBMS uses the strict two-phase locking protocol (Eswaran, 1976; Bernstein, 1987). We discuss the consequences of our requirement for execution autonomy of each local DBMS and restrictions on local and/or global transactions that still allow us to ensure global database consistency and freedom from deadlocks.

The remainder of the paper is organized as follows. In Section 2 we define the multidatabase transaction management model used in this paper. In Section 3 we describe how global transaction processing is performed. In Section 4 we discuss some of the difficulties associated with multidatabase recovery management and the differences between distributed homogeneous and multidatabase recovery problems. In Section 5 we address the question of when a global commit operation can be scheduled for execution, and provide an effective algorithm to do so. This algorithm is shown to ensure global database consistency. In Section 6 we discuss global deadlock problems that may occur in our model. Section 7 concludes the paper.

2. The MDBS Model

A global database is a collection of local databases distributed among different local sites interconnected by a communication network. We assume that the MDBS software is centrally located. It provides access to different DBMS's that are distributed among these nodes. The model discussed in this paper is based on the following assumptions:

1. No changes can be made to the local DBMS software. This assumption is adopted for purely practical reasons as discussed in the introduction. Consequently, while the MDBS is aware of the fact that local transactions may run at local sites, it is not aware of any specifics of the transactions and what data items they may access.

2. A local DBMS at one site is not able to communicate directly with local DBMS's at other sites to synchronize the execution of a global transaction active at several sites. This assumption is a direct consequence of our basic assumption: local DBMS's are not aware of each other and act under the illusion that each transaction is executed only at the site of the local DBMS.

3. Each local DBMS is responsible for ensuring the atomicity of the local transaction. This can be accomplished in a variety of ways, most commonly through the use of the write-ahead log scheme (Gray, 1978). Such a log, however, is not available to the MDBS system and as a result it is not available to any of the global transactions.

4. Each local DBMS uses the strict *two-phase locking* protocol (Bernstein, 1987) (i.e., local locks are released only after a transaction aborts or commits). In addition, a mechanism is available for ensuring deadlock freedom. Thus, each local schedule is serializable, and any local deadlocks are detected and recovered from.

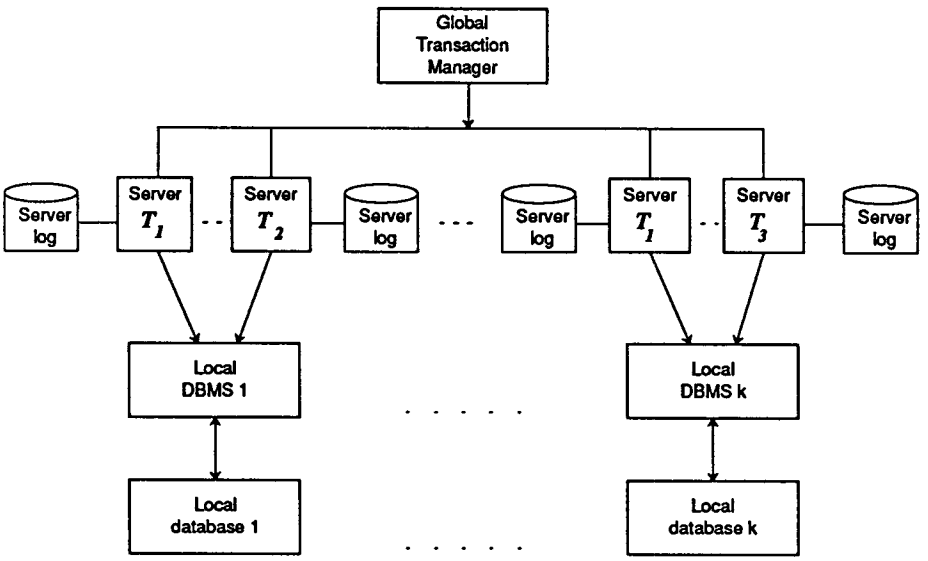
5. A local DBMS is not able to distinguish between local and global transactions which are active at a local site. This assumption ensures local user autonomy. Local and global transactions receive the same treatment at local sites. Therefore, global users cannot claim any advantage over local ones. This ensures greater acceptance of the MDBS system at local sites.

6. A local DBMS cannot participate in the execution of an atomic commit protocol (e.g., the two phase commit protocol). As discussed in the introduction, this assumption is made to ensure *execution* autonomy.

Thus, the MDBS is the only mechanism that is capable of coordinating executions of global transactions at different local sites. However, any such coordination must be conducted in the absence of any local DBMS control information. Hence, the MDBS must make the most pessimistic assumptions about the behavior of the local DBMS's in order to ensure global database consistency and freedom from global deadlocks.

2.1 System Structure. The MDBS system is responsible for the processing of the global transactions and for ensuring global serializability and freedom from global deadlocks. The system consists of two major components—the global transaction manager and set of servers. The general structure of the system is depicted in Figure 1.

Figure 1. MDBS Architecture



2.1.1 Global Transaction Manager. The global transaction manager (*GTM*) is responsible for the users' interactions with the MDBS system. The *GTM* uses the MDBS directory to generate (statically or dynamically) a global transaction, which is a sequence of **read, write, commit** or **abort** operations. For each global operation to be executed, the *GTM* selects a local site (or a set of sites) where the operation should be executed. In each such site, the *GTM* allocates a server, (one per transaction per site) which acts as a global transaction agent at that site. Once the *GTM* allocates a server to the transaction, it is not released until the transaction either aborts or commits. All transaction operations that are to be executed at the site are eventually sent to the server. In submitting transaction operations for execution, the *GTM* uses the following restriction:

> *No operation of the transaction (except the very first one) is submitted for execution until the GTM receives a response that the previous operation of the same transaction has completed.*

The *GTM* is also responsible for managing the order of execution of the various operations of different global transactions. An operation can be submitted to the appropriate server for execution as soon as it is generated, or it can be rejected (in that case the transaction issuing the operation should be aborted), or it can be delayed. This determination is carried out through the use of a *global lock* scheme that keeps track

of global transactions' requests for local locks (recall that local locks are maintained and allocated by local DBMS's). Each global data item has a global lock which can be either shared exclusive. If two global transactions request conflicting global locks the scheduler will prevent one of the transactions from proceeding because it knows that the two transactions will cause a conflict at the local site. The scheduler uses the strict two-phase locking protocol for allocating global locks to global transactions. It should be noted that if a global transaction has been granted a local lock on a data item at a local site, then it also has been granted a global lock on the same data item. The converse is not correct; that is, if a global transaction has been granted a global lock on a data item, it may still have to wait at some local site to obtain a local lock at some local site.

By introducing two types of locks—global and local, we implicitly assume that the granularity of a data item at both global and local levels is the same. This implies that a local lock conflict between two global transactions causes a global lock conflict between them. Thus, two global transactions cannot directly compete for a local lock at some local site. This assumption should not affect the generality of our transaction management scheme, since in a multidatabase environment a global transaction may only refer to a data item that is accessible by a user transaction (relation or a tuple, for example). In commercial DBMS's like SQL/DS, DB2, and INGRES, locking at the level of relations and tuples is available to users. On the other hand, almost no commercial DBMS's provide user transactions with the capability of locking at the page or a record level, since neither page numbers, nor records identifications are visible to user transactions. As a result, user transactions (global or local) can lock only at the relation or tuple level.

If a global transaction issues a tuple lock request, then the *GTM* passes the request to the appropriate local DBMS. The DBMS, in turn, acquires a tuple lock along with other locks (such as index lock, or TID lock, etc.) required by the transaction. Therefore, even if two global transactions request two different tuple locks they may in fact conflict on an index lock required to process the tuple. Thus, the *GTM* should be made aware about such possible conflicts by obtaining information not only on what relations are being accessed, but also on the primary and secondary indexes used for these relations. There are several different ways to implement such a locking scheme. One possible implementation is to employ a simple form of predicate locking in the following way:

1. If either a query, update, or insert is issued with primary key search predicates, then lock the primary keys of those tuples that are either being searched or updated.

2. If either a query, update, or delete is issued without primary key search predicates, then lock the entire relation.

2.1.2 Set of Servers. A server is a process generated by the *GTM* at some local site to act as an agent for the global transaction at that site. The local DBMS treats each server as a local transaction. Each time a global transaction operation is scheduled and is submitted for execution, it is eventually received by the server. Results of the operation execution by a local DBMS are reported to the *GTM*. In addition, a server also needs to interact with the *GTM* after a failure has occurred (more on this in Section 4).

2.2 Types of Failures The MDBS system must be able to cope with a variety of types of failures, including *transaction*, *system*, *server*, *site*, and *communication* failures. A *transaction* failure is indicated by an abort. When a global transaction is aborted by a local DBMS, the abort should be propagated to each site at which the transaction was active. A *system* failure is generally manifested by the corruption of the MDBS internal data structures and can be caused by a number of different things (for example, an operating system failure at the site where the MDBS system is located causes a multidatabase system failure). A system failure causes failure of all global transactions that are active at the time of failure. A *server* failure is indicated by the abort of the server process either by the operating system of the local site or by the DBMS of that site. The latter abort may occur at any time as was discussed in Introduction. A system failure causes the failure of all servers for all transactions at all local sites. However, the converse is not correct; the failure of a transaction server does not cause a system failure. A *site* failure in a multidatabase environment is similar to a site failure in a homogeneous distributed database environment. A local site failure may happen, for example, when a local database system fails, causing any global transaction that was active at the site to be aborted. After the site becomes operational again, the local DBMS ensures that all committed local transactions are redone, and that transactions that were aborted are undone. A *communication* failure means that a communication line between the MDBS software and some local site has failed. Site, communication, and server failures are manifested by the inability of global transactions to communicate with the MDBS system. We assume that the MDBS always detects *communication*, *site*, or *server* failures by requiring a server to acknowledge every received message from the *GTM*. The *GTM* assumes that a failure occurred if no acknowledgment and the transaction is aborted.

3. Global Transaction Processing

A global transaction is a sequence of operations $op_1, op_2, \ldots, op_k$, where op_1 is **begin**, op_k is either **commit** or **abort**, and each op_j $(1 < j < k)$, is either **read** or **write**. We assume that each data item can be read only once by the transaction and if a data item is read and written by the transaction, then a **read** occurs before a **write**.

Each global transaction is executed as a collection of subtransactions, each of which is executed as regular local transactions at some local DBMS. A local transaction has a structure similar to a global transaction in terms of the **begin, read, write, abort,** and **commit** operations.

A transaction is initiated when the operation **begin** is encountered by the *GTM*. After the global transaction has successfully completed the execution of its **read** and **write** operations at each local site, the *GTM* must submit the **commit** operation to all the sites in which that global transaction has executed. Since the various DBMS's cannot participate in the execution of a commit protocol, a situation may arise where a global transaction commits at one site and aborts at another site resulting in non-atomic transaction execution. Thus, to ensure the atomicity of each of the global transactions, we must employ some form of commit protocol at the server level. To do so, each server must keep a *log* consisting of the changes made by the global transaction that it is responsible for. Each time that the server updates a local database, it also updates its own log, which is kept in stable storage. This does not mean that each time a record is put into a server transaction log that a local input/output operation is required. The server may employ buffering techniques to minimize a number of external input/output operations. A server log, in contrast to the traditional write-ahead log (Gray, 1978) contains only the "redo" records (as will be seen in Section 4).

In this paper, we assume that the two-phase commit protocol (Gray, 1978) is used at the server level. When the MDBS encounters the commit operation of transaction T_i, it sends to each server involved in this execution a *prepare-to-commit* message. If a server determines it can commit, it forces all the log records for T_i to stable storage, including a record $<$*ready* $T_i >$. It then notifies the MDBS whether it is ready to commit, or T_i must be aborted. We assume here that a server at this point is not able to notify the local DBMS that it started the execution of the first phase of the **commit** operation. We believe that it is quite a realistic assumption, since we are not aware of any major commercially available DBMS (with the notable exception of Sybase!) that provides users with a **prepare-to-commit** operation that would enable a local DBMS to force transaction log records into the stable storage and to abort or commit the transaction only after the MDBS makes that decision. Even if in the future such a command should become available to users from a majority of DBMS vendors, it would be

enough to have only one DBMS that does not provide the **prepare-to-commit** operation in order to generate the transaction management problems discussed here. In any event, our assumption does not restrict a generality of our results. As we discussed in the Introduction, the preservation of execution autonomy implies that a local DBMS may abort a server even after the server has notified the DBMS that it has started a commit process.

The MDBS collects all responses from the servers. If all responses are "*ready* T_i", then the MDBS decides to commit T_i. If at least one server responds with "*abort* T_i", or at least one server fails to respond within a specified time out period, the MDBS decides to abort T_i. In either case, the decision is sent to all servers. Each server, upon receiving the decision, in turn, informs the local DBMS as to whether to abort or commit T_i at that local site.

Note that even though T_i is considered *globally committed* after the MDBS has sent a **commit** message to each transaction server, from the local DBMS's point of view the transaction is not committed yet. The transaction may still at this point in time abort locally (if the site or the server at the site fails prior to processing of the local **commit** of T_i). Since the local DBMS log is not available to the server, a separate log is kept by each server. The appropriate local DBMS log records appear in local DBMS stable storage only after a server submits a **commit** operation to a local DBMS for execution and a transaction is *locally committed.*

During a global transaction processing by the MDBS system, the transaction can be in one of the following states:

- *active*, if the **begin** operation has been processed by the MDBS, but neither a **commit** nor an **abort** has been processed thus far.
- *abort*, if an **abort** operation has been processed on behalf of the transaction.
- *ready-to-commit*, if the transaction completes all its **read** or **write** operations, and each server at the local site at which the transaction was active reports that all server log records associated with the transaction are in the stable storage.
- *commit*, if a commit message is sent to each local site at which the transaction was executing.
- *blocked*, if the transaction is waiting to obtain a global lock (see discussion in Section 6).

4. Global Transaction Recovery

If a global transaction fails before it commits, then any of its local subtransactions must be undone by the appropriate local DBMS's. As a consequence, global database consistency is also preserved, since the transaction did not make changes in any of its local databases. Thus, the MDBS does not need to use the **undo** operation to restore a multidatabase to a consistent database state.

The situation becomes more complicated if a failure occurs during the processing of a **commit** operation of a global transaction. Consider the case where the MDBS decides to commit transaction T_i. Suppose that a local site, S_k, in which T_i was active fails without having the appropriate local DBMS log records in stable storage and before the server for T_i at site S_k has received the commit message from the MDBS but after the server has notified the MDBS that it is ready to commit. If such a failure occurs, then upon recovery of S_k, T_i must be undone at S_k. However, the MDBS considers T_i to be committed and thus upon recovery of S_k, the MDBS must redo T_i at S_k. As far as the local DBMS is concerned, redoing the transaction constitutes a new transaction at that site, without any connection to the failed one. Thus, it is possible that between the time that the local DBMS at S_k recovers from the failure and the time that the restarted transaction T_i at site S_k is redone, the local DBMS may execute some other local transactions that, in turn, may lead to the loss of global database consistency. This situation creates serious recovery problems in the multidatabase environment, as illustrated below.

Example 1: Consider a global database consisting of two sites S_1 with data item a, and S_2 with data item b. Assume the following global transaction T_1 has been submitted to the MDBS:

$$T_1\text{: } r_1(a)\ w_1(a)\ w_1(b)$$

Suppose that T_1 has completed its execution at both sites and is in the *ready-to-commit* state. The MDBS now submits the **commit** operation to the servers at both sites. Further suppose that site S_2 has received the **commit** and has successfully executed it, while site S_1 fails before the **commit** operation arrives. Therefore, at site S_1 the local DBMS considers T_1 aborted and, as a consequence, releases its local locks. Upon recovery of S_1, a new local transaction L_2 is submitted at the site S_1:

$$L_2\text{: } r_2(a)\ w_2(a)$$

Transaction L_2 can now obtain the lock on a and execute to completion. Following this, the lock on a is released, and the server obtains a lock on a and redoes the trans-

action's T_1 write operation as a new transaction—T_3. The sole purpose of T_3 is to set the value of data item a to the original state it was in just after being written by T_1.

Transaction T_3 consists of all (and only) the write operations of the original transaction T_1 pertaining to the data item residing at site S_1. (We can always generate this transaction since we have all the relevant information in the server log.) Thus, in the case of transaction T_1, the new transaction is:

$$T_3 : w_3(a)$$

This new transaction is then submitted for execution. Thus, from the local DBMS's viewpoint, the committed projection of the local schedule generated at site S_1 is:

$$r_2(a)\ w_2(a)\ w_3(a)$$

However, T_3's **write** operation is the same as $w_1(a)$ as far as the MDBS is concerned. Thus, this execution results in the following nonserializable schedule from the MDBS viewpoint, where both T_1 and T_2 are committed in the schedule.

$$r_1(a)\ r_2(a)\ w_2(a)\ w_1(a)$$ □

The above example demonstrates one of the major difficulties in dealing with recovery in the multidatabase environment. A global transaction that fails for whatever reason at some local site is undone by the local DBMS, while it should be redone as far as the MDBS is concerned. Barring the elimination of local DBMS execution autonomy, it appears that it would be impossible to recover from global transaction failures during the commit phase of the transaction unless we place some restrictions on the way global and local transactions can interact.

Notice that the cause of our difficulties is the fact that between the time a global subtransaction was undone by the local DBMS and the time it is reexecuted as a new transaction, a number of local and global transactions can execute, potentially resulting in the system becoming inconsistent. Fortunately, we need to worry about only local transactions since the global lock scheme prevents two conflicting global transactions from committing in parallel at the same site.

Let us analyze the source of this difficulty. Let **read** (T_i) and **write** (T_i) denote the set of data items read and written respectively by transaction T_i at a local site. Let T_i be a transaction that needs to be reexecuted as a new transaction, and let $L_i = \{L_{i1}, L_{i2}, ..., L_{ih}\}$ be the set of local transactions that executed between the time a global subtransaction of T_i was undone by the local DBMS and the time it is reexecuted as a new transaction. To guarantee that the situation described in Example 1 would not occur, it suffices to ensure that either one of the following two conditions hold:

1. Condition 1

   - **write** $(T_i) \cap$ **read** $(L_{ij}) = \emptyset$ for all L_{ij} in L_i
   - **write** $(T_i) \cap$ **write** $(L_{ij}) = \emptyset$ for all L_{ij} in L_i

2. Condition 2

   - **read** $(T_i) \cap$ **write** $(L_{ij}) = \emptyset$ for all L_{ij} in L_i

Either of these two conditions guarantees that T_i and some L_{ij} in L_i cannot interact in an adverse way.

The next question to be addressed is how to ensure that either of these two conditions is met in our system. To do so, we first need to differentiate among the various types of data items used in an MDBS environment. Let a be a data item in a multidatabase. If a is replicated (i.e., a is located at several sites), then it is obvious that a local transaction cannot update a. Thus, in a multidatabase environment there are data items that can be updated only by global transactions. These data items are referred to as *globally updateable* data items. All other data items are referred to as *locally updateable* data items. We note that a locally updateable data item can be written by both global and local transactions.

The restriction on access to globally updateable data items, however, is not sufficient to guarantee that either one of the above two conditions is met in the system. To do so, we need to impose some additional restrictions on reading and writing access by the various transactions.

For Condition 1 to hold, we require that a local transaction be prohibited from reading and writing globally updateable data items, and that a global transaction be restricted to writing only globally updateable data items.

For Condition 2 to hold, we require that a global update transaction be restricted to reading only globally updateable data items.

If either of these conditions holds, then the MDBS system can maintain integrity constraints that involve both global and local data items. For example, suppose that data item a is globally updateable and data item b is locally updateable. Then integrity constraint $a > b$ can be maintained by the system. Indeed, a can be modified by a global transaction, since it can read b (Condition 1). It also can be maintained by a local transaction, since it can read both a and b and modify b (Condition 2). It cannot modify a, however. Thus, to modify a of the constraint, the global transaction should be used and to modify b the local transaction can be used.

5. Scheduling of the Global Commit Operation

In this section we address the question of when a global **commit** operation can be scheduled for execution. At first glance, one might think that a global **commit** operation can be submitted to the various servers for execution as soon as the global transaction completes its execution at all sites. Unfortunately, if a failure can occur during the processing of global **commit** operation it cannot be unconditionally passed to the servers without the possible loss of global database consistency, as the next example demonstrates.

Example 2: Consider a multidatabase that consists of data items a and b at site S_1 and c and d at site S_2. Let these data items be globally updateable. Consider the following global transactions submitted to the MDBS:

$$T_1\text{: } w_1(a)\, w_1(c)$$
$$T_2\text{: } w_2(b)\, w_2(d)$$

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

$$L_3\text{: } r_3(b)\, r_3(a)$$
$$L_4\text{: } r_4(d)\, r_4(c)$$

Consider now the following scenario. Operation $w_1(a)$ is sent to site S_1 and after it is executed, operation $w_1(c)$ is sent to S_2 and is also successfully executed. Following this, transaction L_3 at site S_1 successfully completes the execution of $r_3(b)$ and then must wait for T_1 to release the local write-lock on a. The **commit** operation for T_1 is now submitted for execution. T_1 successfully commits at S_1 and releases its local locks, but T_1's server at site S_2 fails before it receives a *commit* message and thus at site S_2, transaction T_1 is considered aborted by the local DBMS.

In the meantime, at site S_1, transaction L_3 completes its execution and successfully commits. Site S_2 now recovers and undoes T_1. The operations of transaction T_2 are now submitted and successfully completed at both sites, and the **commit** operation is submitted for execution and eventually is executed successfully. Finally, L_4 is successfully executed and committed at site S_2. Following this, transaction T_1 is resubmitted again for execution at site S_2 and successfully commits.

Thus, at site S_1 the local schedule is $\langle T_1;\ L_3;\ T_2 \rangle$, while at site S_2, the local schedule is $\langle T_2;\ L_4;\ T_1 \rangle$. Therefore, the scheduler generated a nonserializable global schedule. □

In the previous example, we arrived at an inconsistent global database state because the **commit** operation of transaction T_2 was processed before the **commit** operation of T_1 was successfully completed. Since both transactions have no conflicting operations, the scheduler does not have any reason to prevent the transactions from running concurrently. However, local transactions at both sites have conflicting operations with the global transactions, and in this case global inconsistency resulted without the scheduler being aware of it.

Let us assume for a moment that each local DBMS does have a **prepare-to-commit** operation available to user transactions. Then upon the server failure for transaction T_1 at site S_2 in Example 2, the local DBMS would not release local locks held by T_1 at site S_2. Thus, local transaction L_4 would have to wait to execute $r_4(c)$ until after the T_1's server is restarted and transaction T_1 completes its **commit** operation at S_2. However, since execution autonomy of each local DBMS is ensured in our system, the local DBMS at site S_2 may abort T_1 at any time and thereby T_1 will loose local locks. Thus, even if local DBMS's have **prepare-to-commit** statements, the situation described in Example 2 may still occur if a condition of execution autonomy is preserved.

It is interesting to observe that in the case of a homogeneous distributed database the situation of Example 2 cannot occur for the following reason. T_1's failure to commit at site S_2 is considered in such environment a site failure (in an MDBS environment, however, the server is managed by the local operating system and may fail independently of the site). After the site recovers from the failure, the local DBMS at site S_2 would consult either coordinator or site S_1 as to what decision was made concerning transaction T_1. It would be possible for site S_2 to do so, since it is aware that T_1 was also active at site S_1.

Example 2 demonstrates that the *GTM* should use some protocol in scheduling various **commit** operations in order to ensure global database consistency in the presence of failures. We first formulate criteria under which global database consistency is assured in our model.

Theorem 1: Let G be a set consisting of global transactions that have been committed by the MDBS. Global database consistency is retained if and only if there exists a total ordering of transactions from G such that T_i precedes T_j in this ordering if and only if for any local site C_k at which both transactions were executing, there is a schedule S' that is conflict equivalent to a local schedule S_k such that T_i commits before T_j in S'.

Proof: See Appendix A. □

Theorem 1 formulates necessary and sufficient conditions under which global database consistency is retained. However, it does not provide a clue on how to design a *GTM* that assures this. To do so, we need to introduce the concept of a *commit graph*.

A *commit graph* $CG = \langle TS, E \rangle$ is an undirected bipartite graph whose set of nodes TS consists of a set of global transactions (transaction nodes) and a set of local sites (site nodes). Edges from E may connect only transaction nodes with site nodes. An edge (T_i,S_j) is in E if and only if transaction T_i was executing at site S_j, and the **commit** operation for T_i is being processed by the transaction server at site S_j.

The commit graph scheme ensures that two global transactions may not process their **commit** operations concurrently at more than one local site. This is the key concept behind the commit graph scheme.

The idea behind the **commit** graph scheme is identical to the idea behind the site graph scheme introduced in (Thompson, 1987) and (Breitbart, 1988). There, we used the site graph to ensure that two global transactions are not processed concurrently at more than one site. It allowed us to retain global database consistency in the multidatabase environment with no restrictions on local concurrency control mechanisms.

Theorem 2: Let G be a set consisting of global transactions that have been committed by the MDBS. Global database consistency is retained if and only if the *commit graph* does not contain loops.

Proof: See Appendix A. □

In order to decide when it is safe to process a global **commit** operation, the *GTM* uses the *commit graph* defined above, and a *wait-for-commit graph*. A *wait-for-commit graph* (*WFCG*) is a directed graph $G = (V, E)$ whose set of vertices V consists of a set of global transactions. An edge $T_i \rightarrow T_j$ is in E if and only if T_i has finished its execution, but its **commit** operation is still pending and T_j is a transaction whose **commit** operation should be completed or aborted before the **commit** operation of T_i can start.

The *GTM* uses the following algorithm for scheduling the **commit** operation of transaction T_i:

1. For each site S_k in which T_i is executing, temporarily add the edge $T_i \rightarrow S_k$ to the commit graph.
2. If the augmented commit graph does not contain a cycle, then the global **commit** operation is submitted for processing, and the temporary edges become permanent.
3. If the augmented commit graph contains a cycle then:

(a) Let $\{T_{i1}, T_{i2}, ..., T_{im}\}$ be the set of all the transactions which appear in any cycle which was created as a result of adding the new edges to the commit graph. For each T_{ij} in that set, the edge $T_i \rightarrow T_{ij}$ is inserted into the *WFCG*.

(b) The temporary edges are removed from the commit graph.

The algorithm described above ensures that the commit graph will always contain no loops. Thus, from Theorem 2 it follows that our scheme ensures global database consistency.

Next, we need to examine the conditions under which vertices can be removed from the commit graph. It turns out that vertex T_i with all edges incidental to it cannot be removed from the commit graph as soon as the **commit** operation for T_i has been successfully executed at all relevant sites, since otherwise we can obtain an inconsistent global database. This situation is similar to one described in (Breitbart, 1989) in connection to the site graph scheme. However, if a set of transactions that have completed their **commit** operation constitutes a connected component in the commit graph, then all the vertices in this component can be safely removed from the commit graph without the violation of global database consistency.

After transaction T_i either commits or aborts, the *GTM* examines the *WFCG* to determine whether there is some other transaction from the *WFCG* whose **commit** operation can be submitted for execution. We note that transaction T_i, however, need not necessarily wait for the completion of every transaction T_{ik} such that $T_i \rightarrow T_{ik}$. The *GTM* may submit T_i's **commit** operation after some transactions T_{il} such that $T_i \rightarrow T_{il}$ $(0 < l < m)$ successfully commit (and in some cases, a successful commit of only one such transaction would be sufficient to start the transaction's **commit**!). From the *WFCG* definition, it follows that it does not contain any cycles.

Example 3: Consider again the scenario depicted in Example 2. This scenario cannot occur with the use of the algorithm described above since T_2 is placed on the *WFCG* to wait until transaction T_1 completes its **commit** operation. This follows from the fact that the attempt to start the **commit** for transaction T_2 creates a cycle in the commit graph. □

6. Multidatabase Deadlocks

In defining the MDBS model (see Section 2), we have assumed that each local DBMS is responsible for ensuring that no local deadlocks will occur (or if they do occur, they

are detected and recovered from). Unfortunately, this does not ensure that global deadlocks will not occur in the system.

Consider a set of global and local transactions that contains at least two or more global transactions. If each transaction in this set either waits for a local or global lock allocated to another transaction in the set, or waits for the completion of a **commit** operation of some transaction in the set to start its **commit** operation, then each transaction in the set is waiting. Therefore, no transaction from the set can release its global and local locks that 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, the detection of a *global deadlock* situation is not a simple task as will become evident shortly.

Our deadlock handling scheme is based upon the commonly used *wait-for-graph* concept (Korth, 1991). A wait-for-graph is a directed graph $G = (V, E)$, whose set of vertices V is a set of transactions and an edge $T_i \rightarrow T_j$ belongs to E if and only if transaction T_i waits for a lock allocated to transaction T_j. In order to reason about global deadlocks we assume (for argument sake only) that each site maintains a local wait-for-graph (*LWFG*), and that graph is available to the system.

The *GTM* maintains a *global wait for graph* (*GWFG*). The set V consists of the names of the global transactions. An edge $T_i \rightarrow T_j$ belongs to E if and only if global transaction T_i waits for a global lock allocated to global transaction T_j. If a global transaction waits for a global lock, then the transaction state becomes $blocked$ and the transaction is included in the *GWFG*. The transaction becomes active again only after it can obtain the global locks that it was waiting for. The *GWFG* is always acyclic. This can be ensured as follows. Whenever a new edge $T_i \rightarrow T_j$ is to be added to the graph, a cycle detection algorithm is invoked. If the inserted edge causes a cycle to occur, then some global transaction from the cycle is aborted.

The *GTM* does not release the global locks of a global transaction until the transaction either aborts or successfully commits. This allows the *GTM* to handle a failure that occurred during the transaction **commit** operation, when a global transaction successfully commits at one site and fails to commit at the other one. The local locks that the transaction keeps at both sites are released by local DBMS's at both sites. On the other hand, the *GTM* does not release global locks allocated the transaction until the transaction successfully commits at the failed site, preventing other global transactions from using data items that have not yet been committed.

After transaction T_i commits or aborts and its global locks are released, the *GTM* examines the *GWFG*. If transaction T_j in the *GWFG* was waiting only for global locks allocated to T_i, then T_j is unblocked and the next operation of T_j is submitted for execution.

Lemma 1: Let GL be a set of global and local transactions that contain at least two global transactions. Let GR be the union of all *LWFG*s, the *GWFG*, and the *WFCG*. A *global deadlock* exists if and only if GR contains a cycle.

Proof: Let us assume that GR contains a cycle $C = T_{i1} \rightarrow T_{i2}, ..., T_{ik} \rightarrow T_{i1}$. Each edge in the cycle is contributed either from a local wait-for-graph, or from the *GWFG*, or from the *WFCG*. Consider a set of global and local transactions that belong to C. We claim that C satisfies the definition of a set of global and local transactions in a *global deadlock*. Indeed, each global transaction in C is either waiting for a local or global lock or is waiting for a global transaction to complete its **commit** operation.

Let us assume now that there is a set GL of local and global transactions that is in a *global deadlock*. Then, selecting global transaction T_i and using a definition of the *global deadlock*, we construct a sequence of global and local transactions C such that each transaction in the sequence is either waiting for a local or a global lock allocated to the next transaction in the sequence, or is waiting for the next transaction in sequence to commit. Since set GL is finite, the constructed sequence should contain at least one transaction twice. Thus, graph GR contains a cycle. □

According to Lemma 1, the minimal number of graphs that can contribute to the cycle formation in graph GG is:

1. A union of two local-wait-for-graphs.
2. A union of a local-wait-for-graph and the *GWFG*.
3. A union of a local-wait-for-graph and the *WFCG*.
4. A union of the *GWFG* and the *WFCG*.

Examples 4, 5, and 6 below indicate that global deadlocks may indeed occur in the first three cases. Lemma 2 below indicates that in the fourth case a global deadlock situation is not possible.

Example 4: Consider again Examples 2 and 3. In Example 3, we have placed transaction T_2 onto the *WFCG* in order to assure global database consistency. While T_2 is on *WFCG*, it cannot release the local locks it is holding on data items b and d. Transaction T_2 can release these locks only after it commits at both sites. However, T_2 waits on *WFCG* for T_1 to commit. Transaction T_1, on the other hand, cannot restart at site S_2 (after it failed to commit there) before it obtains a local lock on c. Transaction L_4 holds a local lock on c and waits for a local lock on d that is being held by transaction T_2. The system now is in a *global deadlock*, since T_2 is waiting for T_1 on the *WFCG*

and T_1 is waiting for L_4 to release a local lock on c, and L_4 in turn waits for T_2 to release a local lock on d. □

Example 5: Consider a multidatabase that consists of globally updateable data items a and b at site S_1 and c and d at site S_2. Let us further assume that the following global transactions are submitted to the MDBS:

$$T_1\text{: } w_1(a)\ w_1(d)$$
$$T_2\text{: } w_2(c)\ w_2(b)$$

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

$$L_3\text{: } r_3(b)\ r_3(a)$$
$$L_4\text{: } r_4(d)\ r_4(c)$$

Consider a snapshot of the system where:

1. At site S_1, T_1 is holding a lock on a, and L_3 is holding a lock on b and waiting for T_1 to release the lock on a.

2. At site S_2, T_2 is holding a lock on c, and L_4 is holding a lock on d and waiting for T_2 to release the lock on c.

Since the scheduler is not aware of L_3 and L_4, and since T_1 and T_2 do not access common variables, operations $w_1(d)$ and $w_2(b)$ are submitted to the local sites. We are in a *global deadlock* since at site S_1, T_2 is waiting for L_3 which in turn waits for T_1, while at site S_2, T_1 is waiting for L_4 which in turn waits for T_2. □

Example 6: Consider a global database consisting of two sites S_1 and S_2, and having globally updateable data items a and b at site S_1, and a data item c at site S_2. Let us further assume that the following global transactions are submitted to the MDBS:

$$T_1\text{: } w_1(a)\ w_1(c)$$
$$T_2\text{: } w_1(c)\ w_1(b)$$

In addition to T_1 and T_2, at site S_1 the following local transaction is submitted:

$$L_3\text{: } r_3(b)\ r_3(a)$$

Let us assume that at site S_2, global transaction T_2 has a local write-lock on data item c and at site S_1, transaction T_1 has a local write-lock on a. Let us further assume that at site S_1, local transaction L_3 has a local read-lock on b. Transactions T_1 and T_2 also have global write locks on a and c, respectively. Transaction T_2 requests a global write-lock on b and obtains it, but it waits for a local write-lock on b at site S_1. L_3 requests a local read-lock on a and it waits for T_1. T_1 requests a global write lock on c and it waits for T_2 on the global wait for graph. We are again in a *global deadlock* situation. □

Lemma 2: A union of *WFCG* and *GWFG* cannot contain a cycle.

Proof: Let us assume to the contrary that there is a cycle $C = T_{i1}, ..., T_{ik}, T_{i1}$ in a union of *WFCG* and *GWFG*. Observe that any path in *WFCG* is of length 2. Indeed, if T_i waits for T_j to commit, then, by definition, T_j is included in the commit graph and thus, cannot wait for any other transaction to commit.

There are two cases to consider: C contains only edges contributed by *GWFG* and C contains edges from both *GWFG* and *WFCG*. The first case cannot occur, since we guarantee that *GWFG* is acyclic.

Let us consider the second case. Let us assume that in C, transaction T_{ij} is the last in C that is waiting for $T_{i(j+1)}$ to commit. Therefore, transaction $T_{i(j+1)}$ is being committed and as such has obtained all its global locks. But this contradicts the assumption that transaction $T_{i(j+1)}$ is waiting for a global lock allocated to transaction $T_{i(j+2)}$. The lemma is proven. □

We present now an additional example that demonstrates that a *global deadlock* situation may result if a failure occurs during the processing of the **commit** operation. In Example 4 we illustrated a *global deadlock* situation when one global transaction was waiting to commit for another transaction that failed to commit. In the example below, we illustrate a *global deadlock* between a restarted transaction waiting for a local lock and another global transaction waiting for a global lock that is being held by the restarted transaction.

Example 7: Consider a global database consisting of data items a at site S_1 and b and c at site S_2. All data items are globally updateable. Assume that the following global transactions are submitted to the MDBS:

$$T_1\text{: } w_1(c)\ w_1(a)$$
$$T_2\text{: } w_2(b)\ w_2(a)$$

In addition, the following local transaction is submitted at site S_2

$$L_3\text{: } r_3(b)\, r_3(c)$$

Consider the following snapshot of the transaction processing at both sites.

1. T_2 is in the *ready-to-commit* state.
2. L_3 waits at S_2 for T_2 to release the local lock on b.
3. T_1 finished $w_1(c)$ and is waiting for T_2 to release the global lock on a.

Assume that the commit message for T_2 is sent to both sites S_1 and S_2. At site S_1, the commit message for T_2 is received and successfully executed. On the other hand, at site S_2, T_2's server fails before the commit message for T_2 arrives. As a result, the local lock on a at S_1 is released, but since T_2 failed to commit at S_2, T_2 continues to hold a global lock on a.

At site S_2, T_2 loses a local lock on b, due to the server failure, and the local DBMS allocates it to L_3. Following this, L_3 issues $r_3(c)$ and waits for T_1 to release the local lock on c. The system now is in a *global deadlock*, since T_1 waits for T_2 at the *GWFG* while T_2 is waiting for a local lock on b that is held by L_3, which in turn waits for a local lock on c, that is being held by T_1. □

From Lemma 1, it follows that in order to detect that a global deadlock has occurred, 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 scheme for approximating the union of the local wait-for-graphs. As we shall see, our scheme may result in the detection of false deadlocks, but ensures that no global deadlock will be missed.

Our scheme is based on the ability to distinguish between an *active* and *waiting* state of a global transaction at some local site. To do so, we assume that the server is responsible for requesting the various local locks for the transaction it is responsible for. Given this, we say that that transaction T_i *is waiting at site* S_j if it has a server at S_j and has asked for a lock that has not been granted yet. A transaction that is not waiting at site S_j is said to be *active at site* S_j, provided that it has a server at the site 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 either *active* or *waiting* at a local site is called *executing* at the site.

For nonreplicated databases, each global operation refers to a data item that is located at exactly one local site. This means that if a global transaction is waiting for a local lock, it is waiting only at one local site. Since no operation of a global

transaction can be submitted for execution until the previous operation of the same transaction has been completed, in nonreplicated databases each global transaction can be in the *waiting* status only at one local site. In the sequel we assume that each global transaction can be in the *waiting* status at most at one site also for replicated databases. If a global transaction should execute the **write** operation on a replicated data item, 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 $G = (V, E)$ whose set of vertices V consists of a set of global transactions. An 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 *prepare-to-commit* message indicating the start of the transaction's **commit** operation, its status at all such sites is *active* and remains *active* until the transaction either commits or aborts, provided that no failures occurred during the **commit** operation. After the transaction commits or aborts, the transaction and all edges incidental to it are removed from the *PCG*. If a failure has occurred after the *commit* message was sent and before all sites received the message, the transaction needs to be restarted at the failed site.

The restarted transaction should request some local write locks to redo the transaction at the failed site. Thus, the transaction's status at the failed site is changed to *waiting* and remains such until the transaction receives local locks to redo its operations after the site becomes operational. After this takes place, the transaction status at the site becomes *active* again and remains such until the transaction is successfully redone at the site.

Lemma 3: Let *GG* be a graph that is a union of *PCG*, *GWFG*, and *WFCG*. If *GG* is acyclic, then there is no possibility of a global deadlock (provided that any local wait-for-graph is acyclic, which we have assumed in our model).

Proof: Suppose there is an edge $T_i \rightarrow T_j$ in *GG*. Then either transaction T_i is *waiting* for a local lock and transaction T_j is *active* at the same site, or T_i is waiting for a global lock allocated to transaction T_j, or T_i is waiting for transaction T_j to execute a **commit**. Let us now consider graph *GR* that is a union of local wait-for-graphs, *GWFG*,

and *WFCG*. By Lemma 1, a *global deadlock* exists if and only if there is a cycle in *GR*. Thus, to complete the proof of the lemma, we need to prove that *GR* does not contain any cycle.

Let us assume to the contrary that graph *GR* contains a cycle $C = T_{i1} \rightarrow \ldots \rightarrow T_{il} \rightarrow T_{i1}$, where each T_{ir} $(1 \leq r \leq l)$ is either a global or a local transaction.

This cycle is made up of paths contributed from local wait-for-graphs, the global wait-for-graph and the wait-for-commit graph (the latter contributes only separate edges).

Let P_1, P_2, ..., P_m be paths contributed either from local wait-for-graphs or from a global wait-for-graph, or from the wait-for-commit graph. Any two paths contributing to cycle C should have global transactions as end points, since local transactions may run only at one local site, by definition of a local transaction. Since every global transaction can wait for a local lock only at one local site, and no local deadlocks are allowed, each local wait-for-graph path P_{i1} starts with T_{ir} which is currently *waiting* at the local site, and ends with T_{it} which is *active* at the site.

Any two paths from C may have only global transactions in common with those that appear as end points of the path. Therefore, an assumption that there is a cycle, C, in *GR* allows construction of a cycle in the graph that is a union of *PCG*, *GWFG* and *WFCG* that is built from the global transactions that are end points of paths P_1, ..., P_m. This is a contradiction, and therefore the lemma is proven. □

We conclude this section by describing a global deadlock detection algorithm and proving its correctness. If the *GTM* requests a local lock and it does not receive a message from the local server that the lock has been granted, then the *GTM* can conclude that either a *global deadlock* has occurred, or it takes the local DBMS a long time to allocate a lock to the transaction server, or a failure has occurred. One strategy for dealing with global deadlocks is to use a *timeout* mechanism. If after some prespecified length of time, the *GTM* is not notified that the local lock request has been honored, it simply aborts the transaction making a pessimistic assumption that a global deadlock has occurred. Since the *GTM* is only guessing that the transaction is in a global deadlock, it may well abort a transaction that is not in a deadlock. Aborting a transaction that is not in a deadlock implies a performance penalty that is similar and in some cases is more severe than in a homogeneous distributed database environment. These complications are due to the desire to preserve the execution autonomy of each of the local DBMS's. Since each local DBMS has different performance characteristics, the selection of a proper timeout period also presents some difficulties.

Therefore, it would be advantageous for us to seek a more flexible deadlock handling scheme. Unfortunately, as Lemma 1 indicates, there is no way to avoid aborting

global transactions that are not in a global deadlock, since the *GTM* has no access to the various local wait-for-graphs. Thus, the best we can hope for in a multidatabase environment is to reduce the number of unnecessary aborts as compared with the simple timeout scheme outlined above. The algorithm given below is one such improvement.

Our global deadlock detection algorithm is a timestamp based scheme. We assume that when the *GTM* receives the **begin** operation of transaction T_i, it assigns a unique timestamp $ts(T_i)$ to the transaction (No two different transactions can have the same timestamp.) The global deadlock detection algorithm maintains the graph *GG* which is the union of *PCG*, *GWFG*, and *WFCG*. The algorithm is invoked by the *GTM* whenever some global transaction T_i requests a global lock, or local lock at some site S_j, or a **commit** operation of some transaction is placed on the *WFCG*. The algorithm consists of the following four steps:

1. Update *PCG*, *GWFG*, and *WFCG*, accordingly.
2. If there is a cycle in *GWFG*, then abort T_i and exit.
3. If there is a cycle in *GG*, then
   (a) 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.
   (b) If $ts(T_i) <$min $(ts(T_{i1}), ..., ts(T_{im}))$ then continue to wait, otherwise abort T_i at all sites, provided that T_i is not a transaction that is executing the **commit** operation.
4. If there is no cycle in *GG*, then continue to wait.

At the first step of the algorithm, depending whether transaction T_i requests either a global, or a local lock at some site, or needs to wait for the execution of its **commit** operation, some edges are included either in *GWFG*, or *PCG*, or *WFCG*, respectively. If new edges were included in the *GWFG* that created a cycle in the graph, then the cycle is broken by aborting transaction T_i at the second step of the algorithm. Thus, at the third step of the algorithm, we know that the *GWFG* does not contain a cycle. On the other hand, graph *GG* may or may not contain a cycle. If *GG* does not contain a cycle, then, by Lemma 2, a global deadlock may not occur. However, if *GG* does contain a cycle, a global deadlock may have occurred. At this time a decision needs to be made whether *GTM* should continue to wait for T_i to obtain the required lock, or T_i should be aborted (since T_i may well be in a global deadlock). We use the *wait-die* scheme (Rosenkrantz, 1978) to decide whether T_i should be aborted. We assume that in a multidatabase environment, the older the global transaction the more 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 and cannot obtain it. In that case, the *GTM* will abort the transaction and then transaction T_i will be able to obtain the requested local lock after one of the younger transactions in the cycle is aborted.

The key point behind our algorithm is that it is not possible to have a cycle in *GG* consisting of only transactions that are in the process of executing their **commit** operations. Indeed, none of these transactions can be either in *GWFG* or in *WFCG*. On the other hand, each transaction T_i that has failed in the *ready-to-commit* state at, say, two sites S_j and S_k, is restarted at these sites with two different new timestamps, say T_{i_1} and T_{i_2}. Thus, T_{i_1} and T_{i_2} are executing at exactly one site each and therefore no cycle in *PCG* can be generated from transactions that have failed in the *ready-to-commit* state.

To illustrate the global deadlock detection algorithm, let us revisit Examples 4, 5, 6, and 7.

Example 4 (revisited): Transaction T_1's server at site S_2 will not respond with a **commit** completed message, since the server failed. The *GTM* restarts the server with a new timestamp, the status of T_1 at site S_2 is changed to *waiting*.

At this point the union of the *WFCG* and the *PCG* contains a cycle $C = T_1, T_2, T_1$. T_2 is the only *active* transaction at site S_2. Therefore, T_2 gets aborted and thus, transaction T_1 will successfully complete its **commit** operation. □

Example 5 (revisited): Let us assume that the *GTM* calls the global deadlock detection algorithm while T_1 waits for a local lock at S_2. In this case, the graph *PCG* contains a cycle $C = T_1, T_2, T_1$. At site S_2, transaction T_1 is $active$, and since the timestamp of T_2 is larger then the timestamp of T_1, transaction T_2 is aborted. Transaction T_1 can then complete at both sites. □

Example 6 (revisited): Since transaction T_1 waits for a global lock that is allocated to transaction T_2 and the *PCG* contains a path consisting of T_2, T_1, graph *GG* contains a cycle $C = T_1, T_2, T_1$. At site S_2, transaction T_1 is *active* and transaction T_2 is *waiting*. Transaction T_2 gets aborted, since it has a larger timestamp than transaction T_1. Transaction T_1 will then obtain the global lock on c and consequently a local lock on c, and finally will complete its work and successfully commit. □

Example 7 (revisited): Since transaction T_1 waits for a global lock that is allocated to transaction T_2 (that has failed to commit at site S_2) and transaction T_2 implicitly waits for a local lock allocated to T_1, graph *GG* contains a cycle. Since transaction T_1 is *active* and T_2 is restarted at site S_2, and therefore cannot be aborted, the *GTM* aborts T_1. After that, transaction T_2 successfully completes its commit at S_2. □

We prove now that the global deadlock detection algorithm detects and breaks any global deadlock that occurs in the system. The algorithm in some cases assumes that there is a global deadlock, where in fact no deadlock is present. Such a situation is unavoidable in the multidatabase environment, since the *GTM* detects a potential global deadlock, and in the absence of any response from a local site will assume that a global deadlock exists. Since the *GTM* has no way to receive a local wait-for-graph, it cannot construct an union of local wait-for-graphs and *GWFG*, and *WFCG*. Thus, the recovery manager will have to make pessimistic assumptions about global deadlocks, that do not always reflect the reality of the situation.

Theorem 3: The deadlock detection algorithm detects and breaks any *global deadlock.*

Proof: See Appendix B. □

Performance evaluation results (Appleton, 1991) indicate that the deadlock detection method proposed here generates significantly less aborts of global transactions than a simple *timeout* mechanism.

7. Conclusions

The multidatabase transaction management scheme described in this paper ensures global database consistency and freedom from global deadlocks in a failure-prone multidatabase environment. We assumed that each local DBMS uses the strict two-phase locking protocol, which most commercial products use. As a consequence of the execution autonomy requirement, we argued that the various local DBMS's integrated by the MDBS system cannot participate in the execution of a global atomic commit protocol (even if each local DBMS provides a *prepare-to-commit* state available to user transactions!). Our main goal was to provide a scheme that preserves the execution autonomy of each of the constituent local DBMS's.

Our work was motivated by a major problem that exists in the modern industrial data processing environment—the maintenance of consistent data which is distributed among many different DBMS's. It is not reasonable to expect that one will be able to

offer to the industrial user community a single DBMS that will replace all currently existing DBMS's. It is also doubtful that users' organizations will agree to modify existing DBMS software to ensure their cooperation in a multidatabase environment, since this increases significantly the users' maintenance costs and negates the benefits of the multidatabase environment. Therefore, it is imperative to develop methods to support users' data in a consistent and reliable manner, similar to methods developed in the single DBMS environment. We believe that this paper serves as a basis for developing such methods. Indeed, the algorithms developed here are currently being implemented in the prototype version of the multidatabase system developed at the Amoco Production Company.

We have shown that if no restrictions are imposed on a set of local and global transactions then global database consistency cannot be guaranteed. (This follows implicitly from discussions in Bernstein, 1987.) Therefore, in order to guarantee global database consistency some restrictions have to be imposed on the database system. The restrictions discussed in this paper are formulated in terms of read and write sets of local and global transactions. Namely, we separate all data items into two mutually exclusive classes: globally and locally updateable data items, and impose some additional restrictions on reading access by the various transactions.

We believe that the separation of data items into globally updateable and locally updateable classes is reasonable and is administratively easy to maintain. In fact, this separation is necessary in the case of replicated data. Moreover, it is already imposed in some users' organizations. The payoff from imposed restrictions is quite significant. We guarantee consistent data update and data retrieval in the presence of system failures and, in addition, we assure local DBMS autonomy. All this is accomplished without requiring any modifications to local DBMS systems, thereby ensuring that user DBMS maintenance costs will not increase and local DBMS execution autonomy will be retained.

The restriction placed on reading and writing access by the local and global transactions may preclude the writing of some types of applications. This problem arises due to our desire to preserve the important properties of local autonomy, global transaction atomicity, and global serializability. To remove some of the proposed restrictions, some of the above properties must be sacrificed. We are currently investigating ways for achieving this. We note, however, that this may lead to other new restrictions that must be placed on the system.

Appendix A

In Appendices A and B we present the proofs of Theorems 1 through 3. To do so, we need first to define formally what is meant by global database consistency.

We define the notions of a schedule S and a *serial* schedule in the usual manner (Bernstein, 1987). Let op_i and op_j be two operations from S (with op_i preceding op_j in S) belonging to two different transactions T_i and T_j, respectively. We say that op_i and op_j are *conflicting* if they operate on the same data item and at least one of them is a **write** operation. Let S be an arbitrary schedule. We say that transactions T_i and T_j are *conflicting* in S if and only if there is a pair of conflicting operations op_i and op_j in S (with op_i preceding op_j in S) belonging to T_i and T_j, respectively. Two schedules S and S' are conflict equivalent if and only if they are defined on the same set of transactions and have the same set of conflicting transaction pairs. We say that a schedule is *serializable* if and only if it is conflict equivalent to a serial schedule. We use serializability as a correctness criterion for the *GTM* and local DBMS concurrency control mechanisms.

In dealing with schedules that contain **abort** operations we can come across a schedule that is not serializable but if one would consider only operations of committed transactions then the schedule becomes serializable. Since aborted transactions do not make any impact on the database values, instead of schedules as defined above we should deal with their *committed projections*. A *committed projection of schedule S* is a schedule that is obtained from S by deleting all operations that do not belong to transactions committed in S (Bernstein, 1987). We now modify the definition of global and local serializable schedules by considering only their committed projections.

A *serialization graph* of a committed projection of schedule S is a directed graph whose nodes are transactions, and an arc $T_i \rightarrow T_j$ exists in the graph if (T_i, T_j) is a conflicting pair in S and both T_i and T_j are committed in S. A serialization graph of schedule S is acyclic, if and only if S is serializable (Bernstein, 1987).

In order to prove in our model that global database consistency is assured, it is necessary to give a formal definition of global database consistency for a set of global transactions G (Breitbart, 1988). Let G be a set of global transactions. We say that G retains global database consistency if and only if for any set of local transactions L and local schedules $S_1, ..., S_n$ over set of transactions $G \cup L$ at sites $C_1, ..., C_n$, there is a total order of all global transactions in G such that for every two transactions T_i and T_j from G, T_i precedes T_j in this ordering if and only if for any local site C_k where they execute together, there is a serial local schedule S_k^1 *conflict equivalent* to S_k and T_i precedes T_j in S_k^1.

Let G and L be sets of global and local transactions, respectively. Let SG be a union of local serialization graphs. In (Breitbart, 1988) we proved that a set of global and local transactions retains global database consistency if SG does not contain a cycle (Breitbart, 1988).

In order to prove Theorems 1 and 2, we introduce a *commit* order defined by a schedule and prove two lemmas used in proofs of both theorems.

Let S be a committed projection of schedule S' and G be a set of transactions committed in S'. We define a *commit* order on the set G as follows:
$T_i <_c T_j$ if and only if T_i commits in S before T_j commits in S. We will call $<_c$ a *commit* order of schedule S.

Lemma 4: Let S be a committed projection of schedule S', generated by the strict two-phase locking protocol. There is a serial schedule S'' that is conflict equivalent to S such that the *commit* order of S is the same as the transaction order in S''.

Proof: Let S'' be a serial schedule defined by the commit order of schedule S. We prove that S'' is conflict equivalent to S. Let (T_i, T_j) be a pair of conflicting transactions in S'' with T_i preceding T_j in S''. Then in S'' there are conflicting operations $op_i(x)$ and $op_j(x)$ with $op_i(x)$ preceding $op_j(x)$ in S''. Let us assume that in S $op_i(x)$ follows $op_j(x)$. Since S is generated by the strict two-phase locking protocol, transaction T_j cannot release its lock on x until it commits. Therefore, in schedule S transaction T_j commits before transaction T_i commits, which contradicts that S'' is defined by the commit order of S.

Conversely, if (T_i, T_j) is a pair of conflicting transactions in S with T_i preceding T_j in S, then in S there are conflicting operations $op_i(x)$ and $op_j(x)$ with $op_i(x)$ preceding $op_j(x)$ in S. Since S is generated by the strict two-phase locking protocol, transaction T_i cannot release its lock on x until it commits. Therefore, in S transaction T_i commits before transaction T_j commits and consequently in S'' transaction T_i appears before transaction T_j and these transactions are conflicting in S''. The lemma is proven. □

To prove theorems 1 and 2 we need to show that for any set of local transactions and any combinations of failures the global schedule generated by the scheduler remains serializable. In other words, we need to prove that in the presence of global transaction restarts, the global serialization graph obtained as a union of local serialization graphs remains acyclic under the condition of our theorems.

Let us assume that during a transaction's commit process a failure occurs and the global transaction committed at one site but aborted at another. Let us further assume

that after the system recovers, the transaction that failed to commit is restarted at the site where it failed, and the resulting local schedule is such that the combination of schedules at all local sites violates global database consistency. Our first goal is to show that each local schedule remains serializable in the presence of global transaction restart operations at local sites.

Let LS be a local schedule that would be generated by the local DBMS, assuming that no failures occurred. Let LS' be a local schedule that is generated by the local DBMS after failures occurred, the system is restored, and all appropriate global transactions at the local site are redone. LS is serializable, since each local DBMS uses the strict two phase locking protocol. The schedule LS' differs from the schedule LS in that:

1. It has a different order of execution of write operations for failed global transactions after they were restarted by the MDBS.

2. It may include additional local transactions.

3. It contains read operations of global transactions that failed to commit and which local DBMS considered aborted.

The new order of write operations for the failed global transactions may create additional pairs of conflicting operations between transactions at the local site and as a consequence, cycles may appear in the local serialization graph.

If a global transaction fails at the local site and then is redone by the MDBS, the local DBMS considers the redone transaction as a completely new transaction T'' which consists of all write operations of the original transaction T, but which is not related to the original transaction T. Therefore, schedule LS' contains, instead of transaction T, two different transactions T' and T'', as far as the MDBS is concerned, where T' consists of all read operations of T and T'' consists of all write operations of T.

From its own viewpoint, the local DBMS generates a serializable local schedule considering T'' as a new transaction unrelated to T and assuming that T' is an aborted transaction. We will show that schedule LS' is also serializable from the MDBS viewpoint that considers T' and T'' as the same transaction. We will need the following lemma.

Lemma 5: If LS has an acyclic local serialization graph, then so does LS', as far as the MDBS is concerned (i. e. if T' and T'' are considered in LS' as the same transaction).

Proof: Let S be a schedule that includes transactions T_i and T_j. We say that $T_i < T_j$, if there is a path between T_i and T_j in a serialization graph of schedule S. Let us assume that $T_i < T_j$ in a serialization graph of LS and LS is a serializable schedule. Without a loss of generality, we may also assume that T_i is a global transaction that failed during the execution of a **commit** operation and had to be restarted resulting in schedule LS'. To prove the lemma, it will suffice to show that there is no cycle between T_i and T_j in LS' (considering T_i' and T_i'' in LS' as the same transaction).

Let us assume to the contrary that $T_i < T_j$ and $T_j < T_i$ are both in LS'. Since the local DBMS generates a local serializable schedule if T_i' is aborted and T_i'' is a transaction that is unrelated to T_i, the following three cases may occur in the local schedule LS'.

1. $T_i' < T_j$ and $T_j < T_i''$

   In this case we derive that there are sequences of local and global transactions in LS' such that the local serialization graph contains paths T_i', T_{i1}, ..., T_j and T_j, ..., T_{jk}, T_i''. Thus, T_i' and T_{i1} as well as T_{jk} and T_i'' contain conflicting operations. Since T_i' (T_i'') contains only read (write) operations, transaction T_{i1} (T_{jk}) contains a write (either read or write) operation on the same data item that is read (written) by T_i' (T_i'').

   If T_{i1} and T_{jk} are local transactions, then T_i reads a local data item and also updates a global data item. Thus, conditions 1 and 2 are violated. If, on the other hand, either of T_{i1} and T_{i1} is a global transaction, then the global scheduler allocated a global lock to either transaction T_{i1} or T_{jk} or both that is (are) in conflict with locks allocated to T_i, before transaction T_i released it. This contradicts the global strict two-phase locking policy of the global scheduler. Thus, case 1 cannot occur.

2. $T_i'' < T_j$ and $T_j < T_i'$

   In this case, as in the first one, we conclude that there are sequences of global and local transactions such that the local serialization graph contains paths T_j, ..., T_{jk}, T_i' and T_i'', T_{i1}, ..., T_j. Thus, T_i' and T_{jk} as well as T_{i1} and T_i'' contain conflicting operations. Since T_i' (T_i'') contains only read (write) operations, transaction T_{jk} (T_{i1}) contains a write (either read or write) operation on the same data item that is read (written) by T_i' (T_i'').

   Repeating the arguments of case 1, we derive that case 2 also cannot occur.

3. $T'_i < T_j$ and $T_j < T'_i$

   In this case, it is possible that T_i is not an updateable global transaction. Thus, it may read both global and local data items. In this case we derive that there are sequences of local and global transactions in LS' such that the local serialization graph contains paths T'_i, T_{i1}, ..., T_j and T_j, ..., T_{jk}, T'_i. Thus, T'_i and T_{i1} as well as T_{jk} and T'_i contain conflicting operations. Therefore, we obtain that at some point, a local DBMS violated the strict 2PL rule. Thus, this case is also impossible and the lemma is proven. □

From the lemma we conclude that in the presence of failures any local DBMS generates a local serializable schedule from the MDBS viewpoint.

Proof of Theorem 1: For each committed projection of a local schedule LS_i we build an acyclic local serialization graph that exists by virtue of Lemma 5. Let us consider a graph SG that is a union of local serialization graphs from each site. We will show that SG is acyclic if and only if a union of commit orders of local schedules generate a total order on the set of global transactions G.

(if) Let us assume that there is a total order on G generated by the union of commit orders of local schedules. Let us assume to the contrary that graph SG does contain a cycle. Let $C = T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_m \rightarrow T_1$ be a cycle in graph SG, where each T_i is either a local or a global transaction.

Since graph SG is a union of local serialization graphs, C is also a union of paths from local serialization graphs. Thus, C can be subdivided into path segments P_1, P_2, ..., P_k such that each P_i is a path of the local serialization graph from site S_i. By Lemma 5, the number of paths is more than 1, since otherwise we would have obtained a cycle in a local serialization graph.

Any two adjacent paths P_i and P_{i+1} have at least one global transaction in common, otherwise we would have a local transaction that ran at two local sites, which contradicts the definition of a local transaction.

Since each local DBMS uses the strict two phase locking protocol, by Lemma 4, a commit order at each local site generates a serializable order of global and local transactions. Thus, the total order on global transactions existing on G generates a serializable order of global transactions at each local site. Hence, at each local site there is a serial schedule equivalent to a committed projection of a local schedule such that global transactions at that site are committed in that order. Then, in the cycle C, global transactions would not be in that order. Therefore, by transitivity, a pair of global transactions can be found in some local schedule that violates the total order of global transactions. This is a contradiction!

(only if) Let us assume that global database consistency is assured. This means that the graph SG obtained as the union of local serialization graphs is acyclic. Let $S = T_{i1}, T_{i2}, ..., T_{in}$ be a sequence of global transactions obtained by a topological sort of SG. We claim that $T_{i1}, T_{i2}, ..., T_{in}$ is a total order satisfying the theorem's assertion. Let us assume that T_i precedes T_j in S. At any local site where both T_i and T_j are executed two cases may occur:

1. Neither $T_i < T_j$ nor $T_i > T_j$ exist in the committed projection of the local schedule SH. In this case there is a local schedule SH' conflict equivalent to a committed projection of the local schedule SH where T_i commits before T_j in SH'.

2. Either $T_i < T_j$ or $T_i > T_j$ exist in the committed projection of the local schedule SH. (Observe, that both $T_i < T_j$ and $T_j < T_i$ cannot exist in SH, since otherwise global database consistency would be violated). Since a local DBMS uses the strict two phase locking protocol, by Lemma 4, we obtain that transaction T_i commits before T_j in the committed projection of the schedule SH. The theorem is proven. □

Proof of Theorem 2: *(if)* Let us assume to the contrary, that the global database consistency is not assured. Therefore, graph SG that is a union of local serialization graphs does contain a cycle, whereas, by Lemma 5, each local serialization graph is acyclic. Let $C = T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_m \rightarrow T_1$ be the smallest cycle in G, where each T_i is either a global or local transaction. Since graph SG is a union of local serialization graphs, C is also a union of paths from local serialization graphs. Thus, C can be subdivided into path segments $P_1, P_2, ..., P_k$ such that each P_i is a path of the local serialization graph from site S_i, where $k > 1$ (by Lemma 5). Any two adjacent paths have at least one global transaction in common. Without loss of generality, we may assume that any two adjacent paths have exactly one global transaction in common and this transaction is the end point of the path from site S_i and the starting point of the path from site $S_{(i+1)}$, where $(1 \leq i < k)$ and an addition is taken by modulo k.

Let us select from C only those global transactions $T_1, T_{i1}, ..., T_{i(k-1)}$, that appear as starting points of paths $P_1, P_2, ..., P_k$, respectively. Since at each site a local DBMS uses the strict two-phase locking protocol, by Lemma 4, for each site S_r, $(1 \leq r < k)$ the **commit** operation for T_{ir} was completed before the **commit** operation for $T_{i(r+1)}$. Let us consider the **commit** operation for T_1. T_1 was executing at at least two sites—S_1 and S_k. Thus, at S_k, the **commit** for T_{ik} was completed before the **commit** for T_1. Therefore, the **commit** for T_{ik}, $T_{i(k-1)}$, ..., T_{i1} was not completed until the **commit** for T_1 was executing. Thus, *commit graph* CG contains $T_1, T_{i1}, ..., T_{ik}$ along with

edges incidental to them during the processing of the **commit** for T_1. Thus, we derive that CG contains the following cycle: T_1, S_1, T_{i1}, S_2, ..., $S_{(k-1)}$, $T_{i(k-1)}$, S_k, T_1. This is a contradiction!

(only if) Let us assume that G retains a global database consistency. We will prove that the *commit graph* CG does not contain cycles.

Let us assume to the contrary, that the *commit graph* does contain cycle $L = T_1, S_1, T_{i1}, S_2, ..., S_{k-1}, T_{i(k-1)}, S_k, T_1$. Since transactions $T_1, T_{i1}, ..., T_{i(k-1)}$ are in the *commit graph*, then these all these transactions are in the process of executing of their **commit** operation. To complete the proof we exhibit a system of local transactions and a set of failures such that the union of local serialization graphs would contain a cycle. The latter would contradict our assumption that G retains a global database consistency.

We generate the local transactions L_1, L_2, ..., L_k, at sites S_1, S_2, ..., S_k, respectively, such that the following relationships hold:

L_1 is conflicting with T_1, and T_{i1} at site S_1.
L_2 is conflicting with T_{i1}, and T_{i2} at site S_2.
. . . . . .
L_k is conflicting with $T_{i(k-1)}$, and T_1 at site S_k

Let us further assume that no errors occur during the commit process for transactions T_{i1}, ..., $T_{i(k-1)}$, but after transaction T_1 reported that it was ready to commit, T_1's server at site S_k failed.

We assume that at sites S_1, S_2, ..., S_{k-1} the following schedules are generated:

$T_1\ L_1\ T_{i1}$
$T_{i1}\ L_2\ T_{i2}$
. . . . . .
$T_{i(k-2)}\ L_{k-1}\ T_{i(k-1)}$

However, due to the failure of T_1 at site S_k, before T_1 is redone, transaction L_k is executed, and thus the schedule at site S_k would be as follows: $T_i(k-1)\ L_k\ T_1$. Therefore, the following cycle in the union of local serializations graph can be constructed: $T_1\ L_1\ T_{i1}\ L_2$ ... $T_{i(k-1)}\ L_k\ T_1$. This is a contradiction! Therefore, the theorem is proven. □

Appendix B

Proof of Theorem 3: Let us assume to the contrary that the system is in a *global deadlock* and the *global deadlock* was not detected by the *GTM*. Since each local DBMS detects and breaks any local deadlock, without loss of generality we assume that each local wait-for-graph is acyclic. In our model a *global deadlock* occurs if and only if every global transaction is either placed on the *GWFG*, or the *WFCG*, or is in a *waiting* status at some local site. Let us consider several cases.

1. No global transaction is in a *waiting* status at any local site.

   Every transaction in this case is either waiting on the *WFCG*, or the *GWFG*,or executing a **commit** operation, or executing locally **read (write)** operations. If a transaction is executing a **read (write)** operation at a local site, then it eventually completes the operation or the transaction gets aborted. Either outcome contradicts the assumption that the system is in a *global deadlock.*

   If a transaction is executing a **commit** operation and fails, then it either restarts and completes successfully or it is in the *waiting* status at the failed site. The latter contradicts the conditions of case 1. Thus, every transaction in the system that was in the process of performing some operation will eventually complete it. Assuming that a *global deadlock* exists, we derive (by Lemma 2) that a union of *PCG*, *GWFG*, and *WFCG* contains a cycle. Since every transaction at local sites is *active*, the *PCG* is empty. Consequently, there is a cycle in the union of *WFCG* and *GWFG*, which contradicts Lemma 2.

2. There are global transactions that are in a *waiting* status at some local sites.

   Every transaction that is waiting for local locks either gets aborted, or continues to wait after the completion of deadlock detection, or is restarted. If a transaction is aborted, then a deadlock is broken. Let us assume that the transaction continues to wait and a *global deadlock* persists. Then we obtain that a *waiting* transaction waits on some local wait-for-graph, or it waits on the *WFCG*, or it waits on the *GWFG*. The deadlock detection algorithm allows the transaction to wait for local locks only for two reasons: either it waits for a completion of the commit process, or it is the oldest transaction that is allowed to wait for local locks at a local site. The latter means that any other global transaction at the site that has received its local locks and appears in the cycle of the *PCG* is younger than the transaction waiting for the local lock at the same site. Let us consider these two cases.

(a) A transaction waits for a completion of the **commit** operation.

According to the **commit** process, no transaction that has decided to commit can be aborted by *GTM* (it can, however, be aborted by local DBMS as we discussed in Section 2). Thus, if transaction T_i cannot complete the commit and waits forever, then it was restarted as T_{i_1} at some site S_j and the restarted transaction is waiting there for local locks. Two global transactions in the *waiting* status at the same site may not be involved in a global deadlock, since, otherwise, we violate the condition that each global transaction cannot wait at more than one site (see proof of Lemma 3).

Hence, there is at least one *active* global transaction T_k at S_j that transaction T_{i_1} may be waiting for to obtain local locks. Transaction T_k cannot be another restarted transaction (that has already obtained its local locks), since otherwise T_{i_1} eventually will obtain local locks released by T_k. If transaction T_k is neither on *GWFG* nor on *WFCG*, then T_{i_1} is not in the deadlock, since T_{i_1} exists only at one site (recall, that T_{i_1} is a restarted transaction). Let us assume that T_k is either in *GWFG* or *WFCG*. Consider a graph combined from *PCG*, *GWFG*, and *WFCG*. If there is no cycle in the graph, then, by Lemma 3, there is no possibility of a global deadlock. If there is a cycle, then it will be broken, because there exists at least one *active* transaction at the same site as the restarted transaction and this transaction is not restarted.

(b) A transaction is not restarted and is in the *waiting* status at some site.

In this case, if there is no cycle in the combined graph then, by Lemma 3, there is no possibility of a global deadlock. If there is a cycle in the combined graph, then the cycle will be broken by the global deadlock detection algorithm. Therefore, the theorem is proven. □

Acknowledgments

This paper is based in part on (Breitbart, 1990) and in part on work supported by the Center for Manufacturing and Robotics of the University of Kentucky, NSF Grants IRI-8904932, IRI-8805215, IRI-9003341, and IRI-9106450. A number of people have helped with the content of the paper. Sharad Mehrotra and Rajeev Rastogi have noted that the edges of a commit graph cannot be removed as soon as a transaction commits. Randy Appleton and Atul Patankar implemented the prototype version of algorithms

described in the paper. Steve Lee has helped in prototyping the algorithms of the ADDS system at Amoco. Lastly, the anonymous referees have made numerous comments and suggestions that have helped considerably in improving our presentation.

References

Alonso, R., Garcia-Molina, H., and Salem, K. Concurrency control and recovery for global procedures in federated database systems, *IEEE Data Engineering,* 10:5-12, 1987.

Appleton, R. and Breitbart, Y. Deadlock detection in a multidatabase, *Technical Report 192-91,* Lexington, KY: University of Kentucky (1991).

Bernstein, P., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*, Reading, MA: Addison-Wesley (1987).

Bever, M., Feldhofer, M., and Pappe, S. OSI services for transaction processing, "Proceedings, Second International Workshop on High Performance Transaction Systems." *Lecture Notes in Computer Science,* 359:2-20, 1989.

Breitbart, Y., Silberschatz, A., and Thompson, G. An update mechanism for multidatabase systems, *IEEE Data Engineering,* 10:12-19, 1987.

Breitbart, Y. and Silberschatz, A. Multidatabase update issues, *Proceedings, ACM SIGMOD Conference,* Chicago, 1988.

Breitbart, Y., Silberschatz, A., and Thompson, G. Transaction management in a multidatabase environment. In: Gupta, A., ed. *Integration of Information Systems: Bridging Heterogeneous Databases*, New York: IEEE Press, 1989, pp 135-143.

Breitbart, Y., Silberschatz, A., and Thompson, G. Reliable transaction management in a multidatabase system, *Proceedings, ACM SIGMOD Conference,* Atlantic City, 1990.

Du, W. and Elmagarmid, A.K. Quasi serializability: A correctness criterion for global concurrency control in InterBase, *Proceedings, International Conference on Very Large Data Bases,* Amsterdam, 1989.

Duquaine, W. LU 6.2 as a network standard for transaction processing. "Proceedings, Second International Workshop on High Performance Transaction Systems," *Lecture Notes in Computer Science,* 359:20-39, 1989.

Duquaine, W. Mainframe DBMS connectivity via general client/server approach, *IEEE Data Engineering Bulletin*, 13:2, 1990.

Elmagarmid, A. and Leu, Y. An optimistic concurrency control algorithm for heterogeneous distributed database systems, *IEEE Data Engineering,* 10:26-33, 1987.

Eswaran, K., Gray, J., Lorie, R., and Traiger, I. The notion of consistency and predicate locks in a database system, *Communication of ACM,* 19:11, 1976.

Gligor, V. and Popescu-Zeletin, R. Transaction management in distributed heterogeneous database management systems, *Information Systems*, 11:4, 1986.

Gray, J. N. Notes on database operating systems: Operating systems, advanced course, *Lecture Notes in Computer Science*. Berlin: Springer Verlag, 1978, pp 393-481.

Korth, H., Silberschatz, A. *Database System Concepts*, second edition, New York: McGraw-Hill, 1991, pp 313-229.

Papadimitriou, C. *The Theory of Database Concurrency Control,* Rockville, MD: Computer Science Press, 1986, pp 1-229.

Pu, C. Superdatabases: Transactions across database boundaries, *IEEE Data Engineering,* 10:19-26, 1987.

Rosenkrantz, D., Stearns, R., and Lewis, P. System level concurrency control for distributed database systems, *ACM Transactions on Database Systems,* 3:2, 1978.

Simonson, D. and Benningfield, D. INGRES gateways: transparent heterogeneous SQL access, *IEEE Data Engineering Bulletin*, 13:2, 1990.

Sugihara, K. Concurrency control based on cycle detection, *Proceedings, International Conference on Data Engineering,* Los Angeles, 1987.

Sybase Open Server, Sybase Inc., Emerville, CA (1989).

Thompson, G.R. Multidatabase concurrency control. Ph.D. Dissertation, Department of Computing and Information Sciences, Oklahoma State University, Stillwater, OK, 1987.