Reliable Transaction Management in a Multidatabase System

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

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

Glenn R Thompson
Amoco Production Company
P O Box 3385
Tulsa, OK 74102

Abstract

A model of a multidatabase system is defined in which each local DBMS uses the two-phase locking protocol Locks are released by a global transaction only after the transaction commits or aborts at each local site Failures may occur during the processing of transactions We design a fault tolerant transaction management algorithm and recovery procedures that retain global database consistency We also show that our algorithms ensure freedom from global deadlocks of any kind

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 environment supports two types of transactions

- **local transactions**, those that are executed by local DBMS, outside of the MDBS system control
- **global transactions**, those that are executed under the MDBS system control

The MDBS is not aware of local transactions that are being executed by local DBMSs

Each DBMS integrated by the MDBS operates autonomously A global transaction executing at a local site cannot access DBMS control information that is not available to any local transaction (such as a wait-for-graph, a schedule, a DBMS log, etc) As a consequence, the MDBS system cannot access any local DBMS control information

Transaction management problems in a multidatabase environment were first discussed in [GLIG85]. Since then, the problem was extensively studied in two basic directions restricted autonomy of the local DBMS's ([ELMA87], [PU87], [SUGI87]) and a complete preservation of a local DBMS autonomy ([ALON87], [LITW89], [BREI89], [DU89])

Restricted autonomy ([ELMA87], [PU87]) implies that the local DBMSs 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 in the homogeneous distributed database environment with hierarchical organization of local sites This issue has been studied extensively in the literature and is fairly well understood

Several authors concentrated their effort on describing of the multidatabase concurrency control mechanism for a multidatabase model with complete preservation of local DBMSs autonomy ([BREI88], [LITW89], [DU89]). In [BREI88] we considered serializability as a correctness criteria of a multidatabase concurrency control mechanism and defined a multidatabase transaction management protocol that ensures both global serializability and freedom from global deadlocks, provided that each local DBMS ensures local serializability and freedom from local deadlocks. A multidatabase transaction manager based on that protocol was recently described in [BREI89]

Du and Elmagarmid [DU89] argued that a serializability might be too strong of a requirement in a multidatabase environment and as a result introduced the notion of *quasi serializability*. Every global serializable schedule is also quasi serializable but not vice versa In order to preserve global database consistency, quasi serializability requires that there be no data dependency between data items located at different local sites processed by the same global transaction. In many practical cases this may be too strong a restriction on global transactions

Litwin and Tirri [LITW89] proposed a new paradigm for a concurrency control that uses the concept of

† This material is based in part upon work supported by the Center for Manufacturing and Robotics of the University of Kentucky and by National Science Foundation under grants No IRI-8904932 and No RRI-8610671 and Commonwealth of Kentucky EPSCoR program

†† This material is based in part upon work supported by the Texas Advanced Research Program under Grant No 4355 and by NSF Grant IRI-8805215

value dates Their approach is applicable to a multidatabase concurrency control scheme that ensures global serializability and freedom from global deadlocks

In all the above previous work, it was assumed that no failures can occur in the MDBS system during the 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 introduce a model of a multidatabase environment in which each local DBMS uses the two-phase locking protocol [ESWA76] We design a fault tolerant transaction management algorithm and describe recovery procedures that retain global database consistency and ensure freedom from global deadlocks

In the next two sections we define a multidatabase transaction management model and describe global transaction processing In Section 4 we describe difficulties associated with multidatabase recovery management and discuss differences between distributed homogeneous and multidatabase recovery problems In Section 5 we describe the scheduler algorithm and prove its correctness In Section 6 we discuss global deadlock problems that may occur in our model In Section 7 we describe the recovery manager algorithm and prove its correctness Section 8 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 A global transaction is an execution of user's program on a global database Transactions considered in our model consist of the operations **begin , read, write, commit** and **abort** A transaction is initiated when the operation **begin** is encountered Upon receiving a **begin**, the MDBS creates a transaction identifier and assigns a timestamp to the transaction No two different transactions can have the same timestamp

To install permanently in the local databases the results of a global transaction, the operation **commit** is used If a user decides to abort a transaction and cancel all its changes in local databases, the operation **abort** is used The other two operations are **read** (denoted by r) and **write** (denoted by w) A **read** copies a global data item into the user address space and a **write** causes a new value of the data item to be written onto a local database 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**

Formally, a 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 define the notion of serializable global (local) schedules in the usual manner [BERN87] and use serializability as a correctness criterion for the MDBS and local DBMS concurrency control mechanisms

We assume that the MDBS software is centrally located It provides access to different DBMSs that are distributed among various local sites interconnected by a network The model discussed in this paper is based on the following assumptions

(1) No changes can be made to the local DBMS software. This means that local DBMSs cannot be modified in a manner that will provide the MDBS with local control information This assumption is adopted for purely practical reasons Any attempt on the part of the user to modify the DBMS software results in the vendor's dropping support of the product As a result the maintenance costs skyrocket and obliterate all advantages that the MDBS can bring to the user's organization

(2) A local DBMS is not able to distinguish between local and global transactions which are active at the local site This assumption ensures local user autonomy

(3) A local DBMS at one site is not able to communicate directly with local DBMSs at other sites to synchronize the execution of a global transaction active at several sites

(4) Each local DBMS uses the strict *two-phase locking* protocol [BERN87] (i.e , local locks are released only after a transaction aborts or commits), and has a mechanism for ensuring deadlock freedom Thus, each local schedule is serializable, and any local deadlocks are detected and recovered from

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

The MDBS system consists of the following four major components

- **Transaction manager** The transaction manager *TM* controls the execution of global transactions For each global operation to be executed, the TM selects a local site (or a set of sites) where the operation should be executed In each such site, the *TM* allocates a server, (one per transaction per site) and the operation is sent to the scheduler for scheduling and further execution at the selected site Once the *TM* allocates a server to the transac-

tion, it is not released until the transaction either aborts or commits In submitting transaction operations for execution, the *TM* uses the following restriction

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

- **Scheduler** The scheduler manages the order of execution of the various **read, write, commit,** and **abort** operations of different global transactions The scheduler receives the next entry from the *TM* and makes the determination whether the operation should be executed, whether the transaction issuing the operation should be aborted, or whether the transaction issuing the operation should wait until it can be executed The scheduler algorithm, which ensures global database consistency, is presented in Section 5
- **Recovery manager.** The recovery manager is responsible for restoring the global database to a consistent state in the case of failure In this paper, we consider software, communication, and site failures We do not consider here the case of non-volatile storage failure Since a major part of this paper deals with the recovery issue, a more extensive discussion concerning the nature of the recovery manager is presented in Section 4
- **Set of servers** A server is a process generated by the transaction manager to act as an agent for the global transaction at the local site The local DBMS treats each server as a local transaction Each server is responsible for translating global operations into the appropriate query language operations of the local DBMS, and submitting these operations for execution to the local DBMS Results of the operation execution by a local DBMS are reported to the recovery manager In addition, a server also needs to interact with the recovery manager after a failure has occurred (more on this in Section 4)

The general structure of the system is depicted in Figure 1

3. Global Transaction Processing

A multidatabase system is built on top of a number of existing DBMSs that are being integrated into a single MDBS Each of its constituent DBMSs assures the consistency of its local database (see assumption (4)) However, running of global transactions by the MDBS may create inconsistent global database states, since local databases do not coordinate the execution of the same global transaction

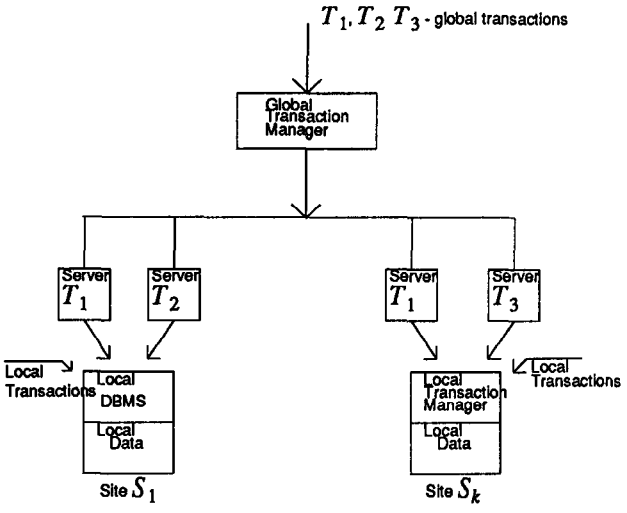

Figure 1 MDBS Architecture

Each local DBMS keeps a write-ahead log on stable storage that is used by the DBMS to restore the local database to a consistent state in the case of failure Before a global commit operation of transaction T_i can be processed, all its changes must appear in the stable log of the various local DBMSs in which it was active This log, however, is not available to the MDBS system and as a result it is not available to any of the transaction's servers Therefore, each server must also keep a write-ahead log 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 The server log is kept in stable storage

After the global transaction has successfully completed its execution at each local site, the MDBS should execute the **commit** operation

In order to ensure the atomicity of global transactions, the two-phase commit protocol [GRAY78] is used When the MDBS encounters the commit operation of transaction T_i, it sends to each server involved in this execution of T_i a *prepare-to-commit* message Each server receiving the message determines if it can commit transaction T_i If 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 The MDBS collects all responses from the servers If all responses are "*ready* T_i", then the transaction enters *ready commit* state and 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 deci-

sion, 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, and thus the appropriate local DBMS log records may not have yet appeared on stable storage of the local DBMS The transaction may still at this point in time abort locally (if the site fails prior to processing of the local **commit** of T_i) It is for this reason that we keep a separate log by each server The appropriate local DBMS log records appear on the local DBMS stable storage only after a server submits a **commit** operation to a local DBMS for execution and a transaction is *locally committed*

4. Recovery Management

There are many different failures that may occur in the multidatabase environment However, the major types of failures are· *transaction* failures, *system* failures, *server* failures, *site* failures, and *communication* failures The scheduler and recovery management algorithms described in this paper tolerate any failures of the described types

If a global transaction fails before it commits, then any of its local subtransactions must be undone by the appropriate local DBMSs As a consequence, global database consistency is also preserved, since the transaction did not make changes in any of its local databases

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 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 obtains the local write locks from the local DBMS to redo the T_i's **write** operations, 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 Let us illustrate these problems with the following example

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

$$T_1 \; r_1(a)\, w_1(a)\, w_1(b)$$

In addition to that, the following local transaction L_2 is submitted to the site S_1·

$$L_2 \; r_2(a)\, w_2(a)$$

Suppose that T_1 has completed its execution at both sites and it 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 lock on a Upon recovery of S_1, L_2 is granted the local lock on a, successfully completes its execution, and commits Following this, the server reexecutes transaction T_1 (as a new transaction, say T_3) This results in the following local schedule at site S_1

$$r_1(a)\, r_2(a)\, w_2(a)\, w_3(a)$$

However, the 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 non serializable schedule from the MDBS viewpoint

$$r_1(a)\, r_2(a)\, w_2(a)\, w_1(a) \quad \square$$

This 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 autonomy, it appears that it would be impossible to recover from global transaction failures during the commit phase of the transaction with no restrictions on global transactions One natural restriction is to partition the set of data items into two classes

- *globally updateable* -- those data items that can only be modified by global transactions, and
- *locally updateable* -- those data items that can only be modified by local transactions

For example, any replicated data item should be globally updateable and not locally updateable A data item, however, need not be replicated to be classified as globally updateable The subdivision of data items into these two mutually exclusive classes is purely administrative However, even with such a restriction, multidatabase recovery may still run into some problems as illustrated in our next example

Example 2: Consider a global database consisting of two sites S_1 and S_2, with data items a and b located at S_1, and c located at S_2. We will assume that a and c are globally updateable, whereas b is locally updateable. Consider the following global transaction T_1 that is submitted to the MDBS

$$T_1 \quad r_1(b)\, w_1(a)\, w_1(c)$$

In addition, let us consider the local transaction L_2 running at site S_1

$$L_2 \; r_2(a)\, r_2(b)\, w_2(b)$$

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. Thus, at site S_1 the local DBMS considers T_1 aborted and, as a consequence, releases its local locks on a and b. Following this, L_2 is immediately given local locks on a and b (before the MDBS is able to restore the value of data item a written by T_1 at site S_1). After L_2 has committed, the MDBS is able to obtain local locks for a and restores T_1 from its server's log. From the local DBMS viewpoint restoring T_1 from the server's log constitutes a new transaction, T_3 that consists of **write** operation on a. This results in the following local schedule at site S_1 as it is viewed by the local DBMS at S_1

$$r_1(b)\, r_2(a)\, r_2(b)\, w_2(b)\, w_3(a)$$

However, the 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 non serializable schedule from the MDBS viewpoint

$$r_1(b)\, r_2(a)\, r_2(b)\, w_2(b)\, w_1(a) \quad \square$$

The reason we have arrived at an inconsistent database state is that we allowed a global transaction to read data items that are locally updateable. The remedy is to require that transactions modifying globally updateable data items would not be allowed to read locally updateable data items. This restriction will be called the *global consistency requirement*.

If a global transaction, however, does not change the value of any data item (i.e., read-only global transactions), then it may read both globally and locally updateable data items. Local transactions may read any data item at the local site.

In [BREI88] we proved that the multidatabase consistency is retained if a local transaction is allowed to modify only nonreplicated data items. Since any replicated data item is always globally updateable, the global consistency requirement is stronger than the one from [BREI88]. Namely, local transactions may modify only those nonreplicated data items that are designated as locally updateable. Any data item that is designated as globally updateable cannot be modified by a local transaction.

5. Scheduler Algorithm and its Correctness

In this section we outline the algorithm used by the scheduler and prove that under the assumptions of our model global database consistency is assured. Obviously, the scheduler cannot allocate local locks, since they are maintained and allocated by local DBMSs. The scheduler, however, keeps track of global transactions requests for local locks through the use of a *global lock* mechanism. Each global data item has a global lock associated with it. A global transaction that needs only to read a data item requests a global read-lock. If two global transactions request conflicting global locks (locks are conflicting, if they are requested by two different transactions on the same data item and at least one of the requested locks is a write-lock) 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 a local lock on a data item at a local site, then it keeps a global lock on the same data item. The converse is not correct, that is, if a global transaction keeps a global lock on a data item, it is not necessarily keeps a local lock for this data item. It may still wait for a local lock at a local site.

The scheduler maintains a *global wait for graph* (*GWFG*) [KORT86]. The *GWFG* is a directed graph $G = (V, E)$, whose set of vertices V is a set of global transactions and 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 global locks that it was waiting for.

As we shall see later, we need to make sure that the *GWFG* is always acyclic. There are two methods for achieving this.

1. 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.
2. Use the *wait-die* scheme [ROSE78] to decide whether a transaction requesting a global lock should wait or abort. That is, if a transaction requesting a global lock on a data item has a smaller timestamp than any transaction holding a

lock on the data item, then it waits, otherwise it aborts Note that we can use the *wait-die* scheme, since each global transaction is assigned a unique timestamp upon entering the system

The scheduler does not release the global locks of a global transaction until the transaction either aborts or successfully commits This allows the MDBS to handle a failure that occurred during the transaction's **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 the local DBMSs at these sites On the other hand, the scheduler does not release global locks allocated to the transaction until the recovery manager successfully commits the transaction after redoing it at the failed site(s), protecting other global transactions from using data values that have not yet been committed

In [BREI88], we described an MDBS transaction management scheme that assures global database consistency under the assumption that no failures can occur during the global transaction commit process In that system, if the scheduler receives a **commit** operation it unconditionally passes it to servers for execution using the two phase commit protocol However, if failures can occur during the global transaction commit process, the **commit** operation can no longer be unconditionally passed to the recovery manager without possible loss of global database consistency, as the next example demonstrates

Example 3: 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

$$T_1 \;\; w_1(a)\, w_1(c)$$
$$T_2 \;\; w_2(b)\, w_2(d)$$

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

$$L_3 \;\; r_3(b)\, r_3(a)$$
$$L_4 \;\; r_4(d)\, r_4(c)$$

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 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 complete 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 $<T_1, L_3, T_2>$, while at site S_2, the local schedule is $<T_2, L_4, T_1>$ Therefore, the scheduler generated an non-serializable global schedule □

In the above 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 (i e , both operate on the same data item and one of them is write operation), 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

Example 3 demonstrates that the scheduler should use some protocol in scheduling various **commit** operations in order to assure global database consistency in the presence of failures We would like to formulate sufficient conditions under which global database consistency is assured in our model To do so we need to introduce a notion of a *commit graph*. A *commit graph* $CG = <TS, E>$ 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 has been scheduled for processing After the **commit** operation for T_i is completed, T_i is removed from the *commit* graph along with all edges incidental to T_i

Theorem 1: For a given system of global and local transactions, the global database consistency is assured if the *commit* graph does not contain any loops □

In order to decide when it is safe to process a global **commit** operation, the scheduler 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** of T_i can be scheduled

The scheduler uses the following algorithm for constructing the *WFCG* graph, and in scheduling a **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) The edges $T_i \rightarrow T_{i1}$, , $T_i \rightarrow T_{im}$ are inserted into the *WFCG* The set $\{T_{i1}, T_{i2}, , T_{im}\}$ consists of all the transactions which appear in the cycle which was created as a result of adding the new edges to the commit graph

   b) Remove the temporary edges from the commit graph

We note that transaction T_i, however, need not necessarily wait for the completion of every transaction T_{ik} such that $T_i \rightarrow T_{ik}$ It may be ready to be scheduled for a **commit** operation after some of transactions T_{il} such that $T_i \rightarrow T_{il}$ $(0 < l < r)$ successfully commit (and in some cases, a successful commit of only one such transaction would be sufficient to schedule the transaction's **commit**!) From the *WFCG* definition, it follows that the graph does not contain any loops

Theorem 2: The scheduler algorithm defined above assures global database consistency □

Example 4: Consider again the scenario depicted in Example 3. 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 schedule 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) Therefore, any local schedule is deadlock-free In such multidatabase environments, however, there is an additional problem of global deadlocks

Consider a set of global and local transactions that contains at least two or more global transactions If each transaction in this set 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, detecting *global deadlock* situations is not simple To illustrate this let us consider several examples

Example 5: In Example 4, we have placed transaction T_2 in 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 in the *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 6: 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

$$T_1 \; w_1(a)\, w_1(d)$$
$$T_2 \; w_2(c)\, w_2(b)$$

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

$$L_3 \; r_3(b)\, r_3(a)$$
$$L_4 \; r_4(d)\, r_4(c)$$

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 MDBS is not aware of L_3 and L_4, and since T_1 and T_2 do not access common variables, the 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 7: 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

$$T_1 \; w_1(a)\, w_1(c)$$
$$T_2 \; w_1(c)\, w_1(b)$$

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

$$L_3 \; r_3(b)\, r_3(a)$$

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 □

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

Lemma 1: Let GL be a set of global and local transactions that contain at least two global transactions Let GG be the union of all local *wait-for-graphs*, the $GWFG$, and the $WFCG$ A *global deadlock* exists if and only if GG contains a loop □

>From Lemma 1, it follows that in order to detect that a global deadlock occurred, it is necessary for the MDBS to have an access to the various local-wait-for-graphs Since this information is not available to the MDBS, it is necessary to devise a different method for approximating the union of the local wait-for-graphs This approximation may result in the detection of false deadlocks, but ensures that no global deadlock will be missed To achieve this, we introduce a new type of graph called a *potential-conflict-graph* (*PCG*) Before defining the graph, we must first introduce the notion of a transaction being *active* or *waiting* at a site

A transaction T_i *is active at site* S_j if it has a server at S_j and the server is either performing the operation of T_i at the site or has completed the current operation of T_i and is ready to receive the next operation of transaction T_i A transaction that is not active at site S_j is said to be *waiting at site* S_j, provided that it has a server at the site and at least one operation of the transaction was submitted to the site A transaction that is either *active* or *waiting* at a local site is called *executing* at the site

We assume that each global transaction can be in the *waiting* status at most at one site This restriction obviously holds for nonreplicated global databases It can also hold for a replicated global database as well In the latter case, if the transaction should execute 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 (via the **restart** operation) 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 2: Let XG be a graph that is formed by taking the union of PCG, $GWFG$, and $WFCG$ If XG is acyclic, then there is no possibility of a global deadlock (provided that each local wait-for-graph is acyclic, which we have assured in our model) □

7. Recovery Manager Algorithm

In this section we briefly outline the local lock request and commit procedures used by recovery manager (*RM*) to detect and break any *global deadlock*

As was stated in Section 2, each global transaction is assigned a unique timestamp upon entering the system The timestamp of a global transaction T_i is denoted by $ts(T_i)$

7.1. The Local Lock Request

Let T_i be a global transaction requesting a local lock at site S_j When the local lock request is made, T_i's status at site S_j is changed to *waiting* and the *PCG* is modified accordingly Following this, the *RM* sets a *timeout* period and waits for the response from the server at S_j If the timeout expires, then *XG*, which is the union of *PCG*, *GWFG*, and *WFCG*, is formed and a cycle detection algorithm is invoked If there is no loop in the graph, then a new timeout period is set, and the above procedure is repeated Otherwise, the set $\{T_{i1}, \ , T_{im}\}$ consisting of all global transactions that are *active* at S_j and appear in at least one loop with T_i, is formed If $ts(T_i) < \min(ts(T_{i1}), \ , ts(T_{im}))$, then a new timeout period is set, and the above procedure is repeated Otherwise, T_i is aborted at all sites After the server responded that the local lock has been obtained, the status of T_i is changed at S_j to *active* and the *PCG* is modified accordingly

7.2. The Commit Operation

Let *SS* be the set of sites at which transaction T_i was *executing* After the *prepare-to-commit* message was sent to every site in *SS*, the *RM* sets a *timeout* period and waits for the local sites to respond If at least one site responded with <*abort*, T_i>, or the timeout period has expired, then T_i is aborted at all the sites in *SS*. If all sites have responded with <*ready*, T_i>, then the record <*commit*, T_i> is added to the global log on stable storage, and the *commit* message is sent to each site in *SS* Following this the *RM* again sets a *timeout* period and waits for the *commit* completion message from each site from *SS* If the timeout period has expired, then for each site S_k that has not responded, if T_i at site S_k has not been restarted, then the transaction is restarted with a new timestamp, the status of T_i at S_k is changed to *waiting*, and the *PCG* is changed accordingly A record is then placed on the global transaction log indicating that T_i has been restarted at site S_k Following this, the graph *XG*, which is the union of *PCG*, *GWFG*, and *WFCG* is formed If *XG* contains a cycle then let $\{T_{i1}, \ , T_{im}\}$ be the set of all global transactions that are *active* at S_k and appear in at least one cycle with T_i The *RM* aborts transaction T_{it}, such that $ts(T_{it}) = \max(ts((T_{i1}), \ , ts(T_{im}))$, and T_{it} is not a restarted transaction If *XG* does not contain a cycle, then a new timeout period is set, and the above procedure is repeated

For each site S_p in *SS*, that responded with a *commit* completion message, the *RM* places a record in the global log specifying that the **commit** of T_i at site S_p has been completed After all the sites in *SS* responded with a *commit* completion message, T_i is removed from both *PCG* and *GWFG*, T_i's global locks are released, and the server of T_i at each site in *SS* is removed

The restarted transaction's local lock requests may create a deadlock situation with other global transactions which are either active or being restarted at the same time Thus the algorithm combines the potential conflict graph, the global wait-for-graph and the wait-for-commit graph to detect and break a potential global deadlock, if any should occur However, a restarted transaction can never be aborted if a deadlock has occurred (or may occur!), since the *RM* has already guaranteed that the new data value will be installed in the database

The server restart process is idempotent If the restarted server fails, then it can be restarted without creating an inconsistent global database state Eventually, the restarted server will be able to complete the restart process, and only then will the data that has been locked at the global level be unlocked for access by other global transactions

Let us consider the case where during the **commit** operation of transaction T_i, a failure has occurred at sites $S_{j1}, \ , S_{jr}$ In this case, the *RM* restarts T_i at these sites However, for each site $S_{i1}, \ , S_{jr}$, the *RM* will restart the transaction with a new timestamp This allows the *RM* to consider each restarted transaction at the local site as a new global transaction On the other hand, the *RM* also has a record specifying that all these restarted transactions are related to the failed global transaction This trick permits us to maintain the restriction formulated above, namely, that each global transaction may not be in the *waiting* state at more than one local site

7.3. Examples

To illustrate the *RM* algorithm, let us revisit Examples 5, 6, and 7

Example 5 (revisited): The transaction's T_1 server at site S_2 will not respond with a **commit** completed message, since the server failed The *RM* restarts the server with a new timestamp, the status of T_1 at site S_2 is changed to *waiting*, and the timeout is initialized to 0

At this point the union of the *WFCG* and the *PCG* contains a loop $C = T_1, T_2, T_1$ 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 com-

mit operation □

Example 6 (revisited): Let us assume that transaction T_1 times out at site S_2. Then the *Local Lock Request* procedure checks graph *XG* for a loop. In this case, the graph *PCG* contains a loop $C = T_1, T_2, T_1$. 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 7 (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 *XG* contains a loop $C = T_1, T_2, T_1$. At site S_2, transaction T_1 is *active* and transaction T_2 is *waiting*. After transaction T_2 has timed out at site S_2, the *Local Lock Request* procedure will abort transaction T_2, 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. □

8. Conclusion

We have presented a transaction management scheme for a multidatabase system that assures global database consistency and freedom from global deadlocks. The model discussed here assumes that every local database is using the strict two-phase locking protocol and a variant of the two-phase commit protocol.

It appears, however, that there is a price to pay for the advantages that a multidatabase environment provides—administrative restrictions on what users transactions are allowed to do. We believe that the restrictions outlined in this paper are administratively easy to maintain. In fact, these restrictions (namely, that each data item can be updated by only one method—either locally or globally) are already imposed in users organizations with which we are familiar. 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.

References

[ALON87] Alonso, R., H. Garcia-Molina, K. Salem "Concurrency Control and Recovery for Global Procedures in Federated Database Systems," IEEE Data Engineering, 1987

[BERN87] Bernstein, P., V. Hadzilacos, N. Goodman *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987

[BREI88] Breitbart, Y., A. Silberschatz "Multidatabase Update Issues," Proceedings of SIGMOD 1988

[BREI89] Breitbart, Y., A. Silberschatz, G. Thompson " Transaction Management in a Multidatabase System,"in *Integration of Information Systems Bridging Heterogeneous Databases*, ed. A. Gupta, IEEE Press, 1989

[DU89] Du, W., A. K. Elmagarmid "Quasi Serializability a Correctness Criterion for Global Concurrency Control in InterBase," Proceedings of the International Conference on Very Large Data Bases, 1989

[ELMA87] Elmagarmid, A., Y. Leu "An Optimistic Concurrency Control Algorithm for Heterogeneous Distributed Database Systems," IEEE Data Engineering, 1987

[ESWA76] Eswaran, K., J. Gray, R. Lorie, I. Traiger "The Notion of Consistency and Predicate Locks in a Database System, CACM, 19 11, 1976

[GLIG85] Gligor, V. and R. Popescu-Zeletin "Concurrency Control Issues in Distributed Heterogeneous Database Management," Distributed Data Sharing Systems, Eds. F. Schreiber, W. Litwin, North-Holland 1985

[GRAY78] Gray, J. N. "Notes on Database Operating Systems Operating Systems An Advanced Course," Lecture Notes in Computer Science 60, pp. 393-481, Springer-Verlag, Berlin, 1978

[KORT86] Korth, H., A. Silberschatz *Database System Concepts*, McGraw-Hill Book Co., 1986

[LITW89] Litwin, W., H. Tirri "Flexible Concurrency Control using Value Date," in *Integration of Information Systems Bridging Heterogeneous Databases*, ed. A. Gupta, IEEE Press, 1989

[PU87] Pu, C. "Superdatabases Transactions across Database Boundaries" IEEE Data Engineering, 1987

[ROSE78] Rosenkrantz, D., R. Stearns, P. Lewis "System Level Concurrency Control for Distributed Database Systems," ACM Transactions on Database Systems 3(2), 1978

[SUGI87] Sugihara, K. "Concurrency Control based on Cycle Detection," Proceeding of the International Conference on Data Engineering, 1987