A Transaction Model for Multidatabase Systems*

Sharad Mehrotra
Rajeev Rastogi
Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188 USA

Henry F. Korth

Matsushita Information Technology Laboratory
182 Nassau Street
Princeton, NJ 08542-7072

Abstract

A multidatabase system (MDBS), consists of a number of sites, each of which runs a distinct commercial database management system (DBMS). The goal of an MDBS is to integrate the various DBMSs to allow applications to access data residing in several DBMSs, without requiring modifications to the individual DBMSs. This implies that each site is allowed a high degree of local autonomy. This autonomy requirement makes the task of ensuring both the atomicity and isolation properties of transactions, in the presence of failures, difficult. In this paper, we develop a semantically rich transaction model for MDBS applications. We relax the atomicity requirement on transactions and propose a new suitable correctness criterion. We also develop new commit and concurrency control protocols that ensure correctness and do not violate the local autonomy of the various sites.

1 Introduction

The problem of transaction management in a multidatabase system (MDBS) has received considerable attention from the database community in recent years [BST90, ED90, GRS91, MRB+92b, MRB+92a]. The basic problem is to integrate a number of pre-existing local database management systems (DBMSs) located at different sites, into an MDBS environment that allows transactions to access data residing at multiple sites. Each local DBMS has a *local transaction manager* (LTM) which is responsible for ensuring local database consistency. The *global transaction manager* (GTM), built on top of the existing databases, is responsible for ensuring global database consistency.

Transactions in an MDBS are of two types:

- **Local transactions,** those transactions that execute at a single site, and outside the control of the GTM.
- **Global transactions,** those transactions that may execute at several sites, and under the control of the GTM.

A distinguishing feature of MDBSs is the requirement that the local autonomy of the local DBMSs be preserved. In this paper, local autonomy is defined to consist mainly of:

- **Design Autonomy.** The local sites are free to follow any concurrency control protocol in order to ensure local database consistency.
- **Execution Autonomy.** Each LTM has complete control over those transactions that are executing at its site. Thus, the LTM is free to abort a transaction as long as the transaction in question has not committed yet.

The preservation of the execution autonomy is essential in an MDBS environment since the local DBMSs may not, in general, permit transactions to hold onto resources, or execute, for an unbounded period of time. This requirement, however, has a serious impact on the way the atomicity of global transactions in an MDBS environment can be achieved [BST90]. For example, the two-phase commit (2PC) protocol, which is the standard protocol used to ensure the atomicity of global transactions [BHG87] cannot be used. The main problem with using the 2PC protocol, or any other atomic commit protocol (e.g., three phase commit protocol [Ske82]), in an MDBS environment

*Work partially supported by NSF grants IRI-8805215, IRI-9003341, grants from the IBM corporation, the HP corporation, and the NEC corporation.

is that such protocols require that in the presence of failures, a global subtransaction in the *prepared state* be allowed to hold onto resources for an unbounded period of time. This clearly violates the execution autonomy requirement. Furthermore, since the pre-existing local DBMSs may not provide for a prepared state, substantial changes may need to be made to the existing DBMS software to support an atomic commit protocol, thus resulting in the violation of design autonomy!

In the absence of an atomic commit protocol it is possible that certain subtransactions of a global transaction abort, whereas others commit, thereby violating the atomicity property. We refer to such a global transaction as a *partially committed transaction*. Partially committed transactions may result in the loss of consistency of the MDBS. The proposed approaches to deal with this problem can be characterized as being either *forward* or *backward* in nature. The forward approach was first suggested in [BST90], where the aborted subtransactions of a partially committed transaction are redone from logs maintained by the GTM. Since the GTM has no control over the execution of local transactions, certain local transactions may execute before the redo transaction of an aborted global subtransaction executes, resulting in a non-serializable execution [BST90, MRB+92b]. Thus, in order to ensure both atomicity and serializability, restrictions are placed on the data items read and updated by global and local transactions.

The backward approach was adopted in [LKS91a] in the context of a general distributed system to alleviate the problem of *blocking*. It utilizes compensating transactions to undo the effects of committed subtransactions of a partially committed transaction. Since compensation only guarantees a weaker form of atomicity, the authors recognize the need for, and propose a correctness criterion for executions in which compensation is used for recovery purposes.

In this paper, we develop a fault-tolerant transaction management scheme for an MDBS environment that combines both the forward and backward approaches. In contrast to the scheme developed in [BST90], where redo logs were used to restore database consistency, our forward approach is based upon retrying of the appropriate aborted subtransactions. Our combined recovery approach allows us to relax some of the restrictions imposed on data items that transactions can read and update [BST90]. We also present new commit and concurrency control protocols used by the GTM that ensure the correctness of executions and do not violate the local autonomy of sites. The correctness criterion we use in the paper is similar to the one in [LKS91a].

The remainder of the paper is organized as follows. In Section 2, we introduce the MDBS model and the global transaction model based on retriable and compensatable types of transactions. Section 3 discusses the GTM commit protocol. In Section 4, a correctness criterion for executions containing compensating transactions is proposed. In Section 5, we present the GTM concurrency control protocol and show that the protocol ensures resulting schedules are correct. Section 6 contains concluding remarks.

2 The MDBS Model

An MDBS consists of a set of autonomous pre-existing centralized local database systems located at sites $s_1, s_2, \ldots, s_m$. Transactions, in our model, are a sequence of read and write operations followed by either a commit operation or an abort operation. A local schedule consists of a sequence of operations resulting from the concurrent execution of transactions at a site. A global schedule is a *distributed schedule* [Pap86] consisting of operations belonging to transactions (global and local) with a partial order on them. The LTM at each local site s_i ensures the atomicity of transactions and the serializability of the local schedules at site s_i.

The execution of global transactions is co-ordinated by the GTM, which communicates with the LTMs by means of *server* processes that execute at each site on top of the local DBMSs. We assume that the interface between a server and an LTM provides for operations to be submitted by the server to the LTM, and the LTM to acknowledge the completion of operations to the server. The GTM does not schedule an operation belonging to a global transaction for execution unless it receives an acknowledgement from the server that the previous operation of the global transaction has been executed at the local site. We also assume that the GTM is centrally located, and the sites at which a global transaction executes are known to the GTM *a priori*. In addition, the LTM does not distinguish between local transactions and global subtransactions executing at its site.

In order to exploit the semantic recovery options of compensation and retrial we develop an extended transaction model. Each global transaction, in our model, consists of a number of subtransactions, each of which is one of the following:

- **Compensatable:** a subtransaction whose execution at a site can be undone, after it commits, by executing a compensating transaction. For example, a subtransaction that reserves a seat in an airline reservation system can be compensated for by a subtransaction that cancels the reservation.

- **Retriable:** a subtransaction that can be retried and eventually succeeds if retried a sufficient number of times. Cancellation of a seat in an airline reservation system, or crediting a bank account, are examples of retriable subtransactions.

- **Pivot:** a subtransaction that is neither retriable nor compensatable.

A global transaction has at most one pivot subtransaction. We further assume that no data dependencies exist between the subtransactions of a global transaction; that is, the execution of a global transaction at one site is independent of its execution at other sites.

Associated with each compensatable subtransaction is a *compensating transaction.* Compensating transactions are transactions that restore database consistency by semantically undoing committed transactions, without resorting to cascading aborts [LKS91a]. Let T_i be a global transaction and T_{ij} be a compensatable subtransaction of T_i that committed at site s_j. To undo the effects of T_{ij}, a compensating transaction for T_{ij}, denoted by CT_{ij}, is executed. CT_{ij} is a separate transaction from T_{ij} and is always serialized after T_{ij} in any schedule. Executing CT_{ij}, however, does not guarantee that all the effects of T_{ij} are undone and thus ensures only a weaker form of atomicity [LKS91a]. In our model, we further assume the following about compensating transactions:

- Since no data dependencies exist between subtransactions of a global transaction, CT_{ij} executes only at the site at which T_{ij} commits.

- CT_{ij} may itself be aborted by the local DBMSs, but if retried a sufficient number of times, it eventually succeeds.

- CT_{ij} is independent of the transactions that execute between T_{ij} and CT_{ij} in the schedule. It depends only on T_{ij}, and the integrity constraints of the database.

We now illustrate the expressive power of the above-developed transaction model by applying it to a banking enterprise. In such an environment, transfer of money between accounts, audits that return the current balance in accounts, deposits and withdrawals from accounts, constitute transactions that can be modeled using our scheme. For example, transactions that transfer money between accounts belonging to different sites can be modeled as global transactions with two subtransactions, one which credits a bank account, and another which debits a bank account. The credit subtransaction is retriable, while the debit subtransaction is compensatable (the compensating transaction for a debit transaction is a credit transaction, which can be assumed to succeed if retried a sufficient number of times[1]). Similarly, an audit transaction can be modeled as a global transaction, all of whose subtransactions are compensatable (the compensating transaction for a read-only transaction does nothing, since a read-only transaction has no effects on the execution of other transactions or the final database state). Transactions that transfer money between accounts belonging to the same site are local transactions as are those that deposit or withdraw money from an account.

3 The GTM Commit Protocol

The GTM commit protocol must ensure that either all subtransactions of a global transaction are committed (that is, they are either committed or retried), or all subtransactions are undone (that is, they are either aborted by the LTM or are compensated for). A commit protocol that ensures the above property of transactions is said to preserve *semantic atomicity* [LKS91a]. Since retriable subtransactions of a global transaction in our model may not be compensatable, and compensatable subtransactions may not be retriable, the GTM commit protocol must control the order in which the subtransactions are committed. The protocol consists of three phases, each of which deals with the commit of one of the subtransaction types. The commit protocol is invoked only after all the subtransactions of a global transaction have completed execution.

To describe the protocol, we need to introduce some terminology. The servers at sites at which a global transaction T_i executes are referred to as T_i's *cohorts.* The server process executing at the site at which T_i's pivot subtransaction executes is referred to as the *p-cohort.* Similarly, servers at sites on which compensatable and retriable subtransactions execute are referred to as *c-cohorts* and *r-cohorts* respectively.

We are now in a position to define the three phases in the GTM commit protocol, which are:

[1] In case the account is deleted, we assume that an exception is raised, and the money is mailed directly to the account-holder.

Phase 1:

- The GTM sends each c-cohort a $\langle commit, T_i \rangle$ message.
- When a c-cohort receives the $\langle commit, T_i \rangle$ message from the GTM, it submits the *commit* operation for T_i to the local DBMS. On receiving an acknowledgement from the local DBMS that the subtransaction has committed, the c-cohort sends a $\langle ack\_commit, T_i \rangle$ message to the GTM. If, however, the subtransaction is aborted by the local DBMS, it sends an $\langle ack\_abort, T_i \rangle$ message to the GTM.

Phase 2:

- When the GTM receives $\langle ack\_commit, T_i \rangle$ messages from all the c-cohorts, it sends a $\langle commit, T_i \rangle$ message to the p-cohort. If, however, it receives at least one $\langle ack\_abort, T_i \rangle$ message from any of the c-cohorts, it aborts the global transaction T_i, and sends all cohorts a $\langle abort, T_i \rangle$ message.
- When the p-cohort receives a $\langle commit, T_i \rangle$ message from the GTM, it submits the commit operation for T_i to the local DBMS. On receiving an acknowledgement from the local DBMS that the subtransaction has committed at the local DBMS, it sends a $\langle ack\_commit, T_i \rangle$ message to the GTM. If, however, the pivot subtransaction is aborted by the local DBMS, it sends a $\langle ack\_abort, T_i \rangle$ message to the GTM.

Phase 3:

- When the GTM receives a $\langle ack\_commit, T_i \rangle$ message from the p-cohort, it commits the global transaction and sends $\langle commit, T_i \rangle$ messages to each of the r-cohorts. If, however, it receives a $\langle ack\_abort, T_i \rangle$ message from the p-cohort, it aborts the global transaction, and sends all cohorts $\langle abort, T_i \rangle$ messages.
- When a r-cohort receives a $\langle commit, T_i \rangle$ message from the GTM, it submits the commit operation for T_i to the local DBMS. In case the local DBMS aborts the subtransaction, it retries the subtransaction until it is committed at the local DBMS. When the subtransaction finally commits at the local DBMS, it sends a $\langle ack\_commit, T_i \rangle$ message to the GTM.

The above protocol specifies the actions taken by a cohort when it receives the $\langle commit, T_i \rangle$ message from the GTM. It also specifies the actions the GTM takes when it receives either a $\langle ack\_commit, T_i \rangle$ or a $\langle ack\_abort, T_i \rangle$ message from each of the cohorts. We now specify the actions taken by the cohorts when they receive a $\langle abort, T_i \rangle$ message from the GTM.

- If the subtransaction has been aborted by the local DBMS, then the cohort sends a $\langle ack\_abort, T_i \rangle$ message to the GTM (if it has not already done so).
- If the subtransaction has neither been aborted nor committed by the local DBMS, the cohort submits the abort operation for T_i to the local DBMS. On receiving an acknowledgement from the local DBMS that the subtransaction has been aborted, it sends a $\langle ack\_abort, T_i \rangle$ message to the GTM.
- If a subtransaction T_{ij} has been committed by the local DBMS at site s_j (note that only c-cohorts can receive a $\langle abort, T_i \rangle$ message from the GTM after T_i has committed at the local DBMS), the cohort schedules CT_{ij}, the compensating transaction for T_{ij}, for execution. On commitment of CT_{ij} at the local DBMS, the cohort sends a $\langle ack\_abort, T_i \rangle$ message to the GTM.

In the above protocol, if any of the compensatable subtransactions or the pivot subtransaction aborts, the GTM aborts the global transaction. If the pivot subtransaction commits, then the GTM commits the global transaction. It should be noted that it is possible for the GTM to receive a $\langle ack\_commit, T_i \rangle$ message followed by a $\langle ack\_abort, T_i \rangle$ message from a c-cohort. However, from a p-cohort or a r-cohort, the GTM only receives either a $\langle ack\_commit, T_i \rangle$ or a $\langle ack\_abort, T_i \rangle$ message.

In the sequel, we say that transaction T_i has ***strongly terminated*** if the GTM receives either a $\langle ack\_commit, T_i \rangle$ message from every cohort, or a $\langle ack\_abort, T_i \rangle$ messages from every cohort. We also say that T_i has ***weakly terminated*** if the GTM receives either a $\langle ack\_commit, T_i \rangle$ or a $\langle ack\_abort, T_i \rangle$ message from every cohort. Thus, a strongly terminated transaction is also a weakly terminated transaction.

At various stages of the execution of the GTM commit protocol, the servers are required to wait for messages before progressing. This, in the presence of communication and site failures could potentially result in blocking. However, the problem is easily alleviated by using a timeout scheme. If a server is interrupted by a timeout while waiting for a message from the GTM, it assumes that the GTM has failed and submits an abort operation to the local DBMS, thereby releasing the resources held by the transaction. Note that a retriable subtransaction that has been aborted by the server on timeout may need to be retried in case the GTM commits the transaction. Thus, the GTM commit protocol does not cause blocking of local applications. Also, if there are no failures, the protocol requires $2n$ messages and 6 rounds as compared to $3n$ messages and 3 rounds needed by the standard 2PC protocol [BHG87], where n is the number of sites at which the transaction executes.

Theorem 1: The GTM commit protocol preserves semantic atomicity of transactions. □

4 Correctness of Non-atomic Schedules

Consider a global schedule S in which each of the global transactions has strongly terminated. Schedule S may contain transactions that are either ***atomic*** or ***non-atomic***. A transaction T_i in a schedule S is atomic if either one of the following two conditions hold:

- T_i is committed in S (in a distributed system, all subtransactions of T_i are committed in S).
- if T_i is aborted in S (in a distributed system, if any of T_i's subtransactions are aborted in S), then it does not have ***any*** effects on the execution of other transactions in S, or on the final database state.

A non-atomic transaction is one that does not satisfy both of the above conditions.

Consider a partially committed global transaction T_1 that commits at site s_1 but aborts at s_2. Depending upon the types of subtransactions T_{11} and T_{12}, either T_{12} is retried, or T_{11} is compensated for. Let S be the global schedule in which T_1 has strongly terminated. If T_{12} is retried, then T_1 is atomic (since all its subtransactions are committed). However, if T_{11} is compensated for, then T_1 is non-atomic (since it is committed at some sites but aborted at others). We refer to a schedule that contains non-atomic transactions as a ***non-atomic*** schedule. In [LKS91b], it was shown that even if a non-atomic schedule S were serializable, and the commit protocol were to ensure semantic atomicity, it is possible that certain transactions "see" an inconsistent state of the database. To illustrate this, consider the following example, from our banking enterprise domain.

Example 1: Let T_1 be a transaction that transfers money from an account A at site s_1 to an account B at site s_2, and consists of a debit subtransaction T_{11} and a credit subtransaction T_{12}. Consider a scenario in which T_{11} commits but T_{12} aborts. Let T_2 be an audit transaction that now executes (before T_{11} is compensated for by crediting account A); T_2 reads the balances in both accounts A and B, and sees a database state in which the sum of the balances of accounts A and B is less than the actual sum. This situation is clearly unacceptable. □

To prevent transactions from "seeing" an inconsistent database state, we must place restrictions on the concurrency permitted in the system. To develop these restrictions, we need to define the following notation. The set of sites at which a global transaction T_i executes is denoted by $exec(T_i)$. The sites at which T_i commits and aborts are denoted by $commit(T_i)$ and $abort(T_i)$, respectively (note that $exec(T_i) = commit(T_i) \cup abort(T_i)$). Furthermore, $S^{commit(T_i)}$ denotes the projection of schedule S on the data items at sites at which T_i commits.

Let $s_1, s_2, \ldots, s_r$ be the sites at which a non-atomic transaction T_i commits; thus $s_j \in commit(T_i)$, where $1 \leq j \leq r$. Let CT_{ij} be the compensating transaction that executes to undo semantically the effects of T_{ij}. Transactions $CT_{i1}, CT_{i2}, \ldots, CT_{ir}$ are considered as subtransactions of a global transaction CT_i, and CT_i is referred to as the compensating transaction corresponding to T_i.

Definition 1: Let S be a non-atomic schedule in which every global transaction is strongly terminated. Schedule S is ***serializable with respect to compensation*** (SRC) if all of the following conditions hold:

- For each non-atomic transaction T_i in S, there exists a compensating transaction CT_i that is committed in S.
- S is serializable.
- Let T_j be an arbitrary global transaction in S. For all non-atomic transactions T_i serialized before T_j in $S^{commit(T_j)}$, if CT_i is serialized after T_j in $S^{commit(T_j)}$, then $abort(T_i) \cap commit(T_j) = \emptyset$. □

Note that in Example 1, in which the audit transaction T_2 sees an inconsistent database state,

$commit(T_2) = \{s_1, s_2\}$. Further, since T_1 (the funds transfer transaction) aborts at s_2, we have $s_2 \in abort(T_1)$. Thus, as T_2 at site s_1 is serialized between T_1 and CT_1, and $abort(T_1) \cap commit(T_2) \neq \emptyset$, the resulting serializable schedule is not SRC. In an SRC schedule, the audit transaction would execute only after the debit subtransaction is compensated for, and thus would see a consistent database state.

It can be shown that that if the schedule S is SRC, then each transaction sees a consistent database state. The proof of this claim is substantial and is beyond the scope of this paper. We refer the interested reader to [MRKS92].

5 The GTM Concurrency Control Protocol

In this section, we present a GTM concurrency control protocol that irrespective of the concurrency control protocol followed by the local DBMSs, ensures that schedules in which every global transaction has strongly terminated are SRC. Our protocol involves insertion and deletion of edges from a transaction-site graph, to be defined below. We first develop an edge management scheme that ensures schedules consisting of global, local and compensating transactions are serializable. We then augment the edge management scheme developed in order to ensure that schedules are SRC.

5.1 The Transaction-Site Graph

Since local transactions execute outside the control of the GTM, the GTM is not aware of the possible indirect conflicts between global transactions at the local DBMSs due to local transactions. This may result in non-serializable executions. In order to prevent such non-serializable schedules, the GTM maintains a graph, called the *transaction-site graph* (TSG), which is similar to the commit graph presented in [BST90]. A TSG is an undirected bipartite graph consisting of nodes corresponding to local sites (site nodes) and global transactions (transaction nodes). Edges in the TSG may be present only between transaction nodes and site nodes. An edge between a transaction node T_i and a site node s_j indicates that $s_j \in exec(T_i)$, and is denoted by either (s_j, T_i) or (T_i, s_j). Edges (T_i, s_k) in the TSG, for all sites $s_k \in exec(T_i)$, are referred to as either T_i's edges, or edges incident on T_i.

Edges in the TSG are inserted as a result of the execution of certain *serialization events* [ED90] (also referred to as *serialization functions* in [MRB$^+$92a]). For a given concurrency control protocol, the serialization event, *ser*, is a function that maps every transaction to one of its operations such that, for any pair of transactions T_i and T_j in a schedule that results from the protocol, if T_i is serialized before T_j, then $ser(T_i)$ executes before $ser(T_j)$ in the schedule[2]. For example, for the timestamp ordering scheme, $ser(T_i)$ is the operation that results in transaction T_i being assigned a timestamp. Similarly, in case of the 2PL protocol, $ser(T_i)$ is the operation that results in T_i obtaining its last lock.

Serialization events may not exist for certain protocols (e.g., serialization graph testing). For such protocols, serialization events can be introduced by forcing conflicts between transactions [GRS91]. For example, we can require that every transaction update a particular data item, say, *ticket*. If some transaction T_i is serialized before another transaction T_j, then T_i must have updated *ticket* before T_j updated it. Thus, $ser(T_i)$ is the write operation of transaction T_i on *ticket*.

We require that all the edges associated with a global transaction T_i be inserted into the TSG before $ser(T_{ik})$ is submitted to the server at s_k, for any $s_k \in exec(T_i)$. Note that as the compensating transaction CT_i corresponding to a non-atomic transaction T_i is also considered as a global transaction, edges corresponding to CT_i must also be inserted into the TSG. Let CT_i consist of subtransactions $CT_{i1}, CT_{i2}, \ldots, CT_{ir}$, where $s_1, s_2, \ldots, s_r$ are the sites in $commit(T_i)$. Since the GTM does not control the execution of CT_{ij} (which are directly executed by the server on receipt of an $\langle abort, T_i \rangle$ message from the GTM), the GTM inserts edges corresponding to CT_i before dispatching any $\langle abort, T_i \rangle$ message to the servers at the sites in $commit(T_i)$.

5.2 The Edge Management Scheme

We now present the rules used by the GTM for deciding when edges can be safely inserted and deleted in the TSG. In contrast to the scheme used in [BST90], our scheme permits the TSG to contain cycles.

In order to describe the scheme we need to define the following terminology. If the GTM receives a $\langle ack\_commit, T_i \rangle$ message from the cohort at site s_j, then edge (T_i, s_j) is referred to as a *committed edge*. Similarly, if the GTM receives a $\langle ack\_abort, T_i \rangle$ message from the cohort at site s_j, edge (T_i, s_j) is referred

[2] For a given protocol, various functions may satisfy the property required of a serialization event. We assume that one of them is chosen to be *ser*.

to as an *aborted edge*. If the GTM has not received any message from the cohort at site s_j, then edge (T_i, s_j) is referred to as an *unmarked* edge. Thus, edge (T_i, s_j) is a committed edge only if T_{ij} has committed at site s_j. If edge (T_i, s_j) is an aborted edge but not a committed edge, then T_{ij} must have aborted at site s_j. Further, if edge (T_i, s_j) is both a committed and an aborted edge, then the compensating transaction for T_{ij} has committed at site s_j.

Edges of a transaction T_i are inserted into the TSG only if the insertion of T_i's edges does not violate the *edge insertion rule* described below. The GTM inserts edges belonging to only one transaction at a time.

Edge Insertion Rule: For every cycle that results due to the insertion of T_i's edges into the TSG at least one of the following condition holds:

1. The cycle contains an edge that is aborted but not committed.
2. There exist transactions T_j, T_k (different from T_i) and distinct sites s_q, s_r such that (T_j, s_q), (s_q, T_i), (T_i, s_r) and (s_r, T_k) are edges in the cycle, and both (T_j, s_q), (s_r, T_k) are committed edges. □

The edge insertion rule ensures that cycles in the TSG do not cause cycles in the *serialization graph* [BHG87]. To see this, consider a cycle in the TSG that satisfies condition 1. Since an aborted subtransaction does not conflict with any other transaction, such a cycle does not cause a cycle in the serialization graph. For a cycle in the TSG that satisfies condition 2, we see that, since $ser(T_{iq})$ executes after $ser(T_{jq})$ at site s_q, and $ser(T_{ir})$ executes after $ser(T_{kr})$ at site s_r, T_i is serialized after T_j and T_k at s_q and s_r respectively. Thus, such a cycle does not cause a cycle in the serialization graph.

Edge Deletion Rule: Let τ be a set of transactions such that for any pair of transactions T_i, T_j, if $T_i \in \tau$ and T_j is connected to T_i by a path consisting of either committed or unmarked edges, then $T_j \in \tau$. If every transaction in τ has weakly terminated, then edges incident on all transactions in τ are deleted from the TSG. □

It must be noted that if edges incident on any transaction in τ are deleted before all transactions in τ have weakly terminated, then non-serializable schedules may result.

Theorem 2: Consider an MDBS where every LTM ensures the serializability of local schedules. If the GTM follows the edge management scheme, then every global schedule is serializable. □

5.3 The Augmented Edge Management Scheme

The edge management scheme described above ensures that the resulting global schedules are serializable. However, it does not guarantee that such schedules are SRC. For example, the schedule in Example 1 is serializable but not SRC; it could be generated by the above scheme as follows. The funds transfer transaction T_1 executes first and edges corresponding to it are inserted. After the GTM receives a message that T_1 at site s_2 is aborted, edge (T_1, s_2) is an aborted edge. Thus, the above edge insertion rule will allow the audit transaction T_2 to execute since the cycle caused by it in the TSG contains an aborted edge, resulting in a non-SRC schedule.

In order to ensure that schedules are SRC, the edge insertion and deletion rules must be augmented as described below. An algorithm to implement the scheme efficiently can be found in [MRKS92].

Augmented Edge Insertion Rule: For every cycle that results due to the insertion of T_i's edges, the following two requirements must hold:

- The edge insertion rule of the edge management scheme, is satisfied.
- If there exists a transaction T_j (different from T_i) and distinct sites s_q and s_r such that (T_j, s_q), (s_q, T_i), (T_i, s_r), (s_r, T_j) are edges in the TSG, then either (T_j, s_q) and (s_r, T_j) are both committed edges, or (T_j, s_q) and (s_r, T_j) are both aborted edges. □

Augmented Edge Deletion Rule: Let τ be a set of transactions such that for any pair of transactions T_i, T_j, if $T_i \in \tau$ and T_j is connected to T_i by a path consisting of either committed or unmarked edges, then $T_j \in \tau$. If every transaction in τ has strongly terminated, then edges incident on all transactions in τ are deleted from the TSG. □

Since a strongly terminated transaction is also weakly terminated, if the augmented edge deletion rule holds, then the edge deletion rule also holds.

Theorem 3: Consider an MDBS where every LTM ensures the serializability of local schedules. If the

GTM commit protocol and the augmented edge management scheme are used, then every resulting global schedule S is SRC. □

6 Conclusion

We have proposed a transaction model for MDBS applications in which global subtransactions may be either *compensatable* or *retriable*. In our enriched transaction model, we use compensation and retrying for recovery purposes. However, since such executions may no longer consist of atomic transactions, it was necessary for us to develop a new correctness criterion that ensures that transactions see consistent database states, and database consistency is preserved.

We have developed a new commit protocol and a new concurrency control scheme that ensures that all generated schedules are correct. The new commit protocol eliminates the problem of blocking, which is characteristic of the standard 2PC protocol. The concurrency control protocol we presented can be used in any MDBS environment irrespective of the concurrency control protocol followed by the local DBMSs in order to ensure serializability.

Acknowledgements

We would like to thank Eliezer Levy for discussions that helped us develop a better understanding of compensation and its role in transaction management.

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[BST90] Y. Breitbart, A. Silberschatz, and G. R. Thompson. Reliable transaction management in a multidatabase system. In *Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data, Atlantic City, New Jersey*, pages 215–224, 1990.

[ED90] A.K. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the Sixth International Conference on Data Engineering*, 1990.

[GRS91] D. Georgakopolous, M. Rusinkiewicz, and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the Seventh International Conference on Data Engineering, Kobe, Japan*, 1991.

[LKS91a] E. Levy, H. F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data, Denver, Colorado*, pages 88–97, May 1991.

[LKS91b] E. Levy, H. F. Korth, and A. Silberschatz. A theory of relaxed atomicity. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1991.

[MRB$^+$92a] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. The concurrency control problem in multidatabases: Characteristics and solutions. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data, San Diego, California*, 1992.

[MRB$^+$92b] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. Ensuring transaction atomicity in multidatabase systems. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego*, 1992.

[MRKS92] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. A transaction model for multidatabse systems. Technical Report TR-92-14, Department of Computer Science, University of Texas at Austin, 1992.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.

[Ske82] D. Skeen. Non-blocking commit protocols. In *Proceedings of ACM-SIGMOD 1982 International Conference on Management of Data, Orlando*, pages 133–147, 1982.