Efficient Global Transaction Management in Multidatabase Systems*

Sharad Mehrotra†  Rajeev Rastogi†  Yuri Breitbart†  Henry F. Korth§  Avi Silberschatz†

Abstract

Concurrency control schemes for ensuring global serializability in a multidatabase system (MDBS) environment are complicated due to the autonomy of local database management systems (DBMSs). In this paper, we develop concurrency control schemes, for the MDBS environment, that are simple, permit a high degree of concurrency and incur minimal overhead under two different assumptions on the nature of schedules produced by the local DBMSs: (1) each local DBMS produces rigorous schedules, (2) each local DBMS produces strongly recoverable schedules. Our interest in rigorous schedules stems from the fact that the majority of commercial DBMSs offer a concurrency control protocol that generates rigorous schedules. However, the class of rigorous schedules is quite restrictive in terms of the amount of concurrency permitted. We, therefore, also consider the class of strongly recoverable schedules which permits a higher degree of concurrency. The class of strongly recoverable schedules is the largest class of schedules for which transaction execution order coincides with their commit order. Therefore, the global transaction manager for multidatabase systems with strongly recoverable local DBMSs guarantees global serializability by simple control of global transaction commit execution without infringing on a local DBMS autonomy.

Our proposed schemes exploit the rigorousness and strong recoverability properties of local schedules to ensure global serializability with minimal overhead both, in presence, and absence of failures. Furthermore, our proposed schemes are simple and require only minimal information from the participating local DBMSs.

1 Introduction

A multidatabase system (MDBS) is a facility that allows access to data located in multiple pre-existing and autonomous database management systems (DBMSs). In such a system, a software package built on top of the local DBMSs, referred to as the global transaction manager (GTM), coordinates the execution of global transactions that execute at multiple sites. Independently, local transactions execute at a single site outside the control of the GTM. Each local DBMS is autonomous in the sense that no design or internal DBMS structure changes are permitted (design autonomy) to the local DBMS. Each local DBMS has complete control over all the transactions executing at its site, including the ability to abort any transaction at any time prior to its commitment (execution autonomy). Furthermore, each local DBMS may or may not communicate any information relevant for concurrency control to the GTM or to any others’ local DBMSs. Each local DBMS has complete control over all the transactions executing at its site, including the ability to abort any transaction at any time prior to its commitment (execution autonomy).

The concept of serializability has been the traditionally accepted notion of correctness for concurrent executions of multiple transactions. In an MDBS environment, however, ensuring global serializability even in a failure-free environment is difficult. This is due to the fact that the local transactions execute outside the control of the GTM, and the local DBMSs may follow different concurrency control schemes. Schemes for ensuring global serializability in a failure-free MDBS environment have been proposed in [5, 2, 6, 7].

In an MDBS, execution of each global transaction incurs overhead at two levels. First, overhead is incurred by the GTM to ensure global serializability in the presence of local transactions. Second, overhead is incurred by local DBMSs that execute global transactions operations. Therefore, to improve performance, it is highly desirable that the concurrency control scheme used by the GTM result in minimal overhead. In this paper, we develop such GTM schemes under two different assumptions on the nature of schedules produced by the local DBMSs:

- Each local DBMS generates rigorous schedules.
Each local DBMS generates strongly recoverable schedules.

Our interest in rigorous schedules stems from the fact that the majority of commercial DBMSs offer a concurrency control protocol that generates rigorous schedules. However, the class of rigorous schedules is quite restrictive in terms of the amount of concurrency permitted. We therefore, also consider the class of strongly recoverable schedules which permits a higher degree of concurrency. The class of strongly recoverable schedules is the largest class of schedules for which transaction execution order coincides with their commit order. Therefore, the global transaction manager for multidatabase systems in which local DBMSs generate strongly recoverable schedules can guarantee global serializability by simple control of global transactions' commit execution without infringing on a local DBMS autonomy. The class of rigorous and strongly recoverable schedules was originally introduced in [2]. A notion similar to strong recoverability, referred to as commit order (CO) was also introduced independently in [10] in which a detailed description of schemes that produce CO schedules was also provided.

Another issue in an MDBS environment is that of making the concurrency control schemes failure-resilient. The problem of developing failure-resilient schemes for MDBS environments has been addressed in [4, 12, 8, 9, 11]. Since global transactions in an MDBS are distributed transactions, that is, they execute at multiple sites, there is a need for a global commit protocol and recovery procedures to ensure the atomicity of such transactions in face of failures. The schemes for fault-tolerance in MDBSs differ widely depending upon whether or not each participating local DBMS supports a prepared state for the execution of atomic commit protocol. In case local DBMSs support a prepared state, an atomic commit protocol [1] (e.g., the 2PC protocol) can be used to ensure global transaction atomicity in an MDBS environment. However, if local DBMSs do not support a prepared state, ensuring global transaction atomicity is difficult. In this paper, we concentrate on concurrency control schemes that can be made failure-resilient even if local DBMSs do not support a prepared state.

The remainder of the paper is organized as follows. In the next section, we describe our MDBS model and formally define the notions of rigorousness and strong recoverability. In Section 3, we consider how global serializability can be ensured if the local DBMSs produce rigorous as well as strongly recoverable schedules. In Section 4, we discuss how our GTM scheme can be made failure-resilient in case local DBMSs do not support a prepared state for the execution of an atomic commit protocol. Finally, Section 5 concludes the paper.

2 MDBS Model

An MDBS consists of pre-existing and autonomous local DBMSs located at sites $s_1, s_2, \ldots, s_m$. A transaction $T_i$ is a sequence of read (denoted by $r_i$), and write (denoted by $w_i$) operations followed by either commit (denoted by $c_i$) or abort (denoted by $a_i$) operations, one per site at which $T_i$ executes. The local schedule at site $s_k$, denoted by $S_k$, is a sequence of operations belonging to local and global transactions resulting from their concurrent execution at site $s_k$. We say that transaction $T_i$ is in a $w$ ($w$, $w$) conflict with transaction $T_j$ in $S_k$ if and only if $S_k$ contains an operation $w_i (r_i, w_i)$ belonging to $T_i$, followed by $r_j (w_j, w_j)$ belonging to $T_j$, such that both $w_i (r_i, w_i)$ and $r_j (w_j, w_j)$ access the same data item at $s_k$ and $T_i$ does not abort before $r_j (w_j, w_j)$ is executed. If $T_i$ is in a $w$, $w$, or $w$ conflict with $T_j$, then $T_i$ is said to be in conflict with $T_j$. Transaction $T_i$ is said to be committed (aborted) in $S_k$ if $S_k$ contains a $c_i (a_i)$ operation. We denote by $T_{i,k}$ the subtransaction of global transaction $T_i$ at site $s_k$.

We assume that the GTM is centrally located and controls the submission of global transaction operations to the local DBMSs (local transactions submitted to local DBMSs are not subject to central GTM control.). For every global transaction operation, the GTM first selects a local DBMS where the operation should be executed, and then determines whether to submit the operation to the local DBMS, to delay it, or to abort the transaction. At each site, the GTM has an agent (server) that submits global transaction operations to the local DBMS. A local DBMS acknowledges the execution of an operation to the server, which, in turn, passes this information to the GTM. The GTM submits an operation of a transaction for execution to the server after it has received an acknowledgement for the execution of the previous operation of the transaction. Each global transaction thus consists of a set of subtransactions, each of which is executed as a regular local transaction at some local DBMS. Every local DBMS ensures the atomicity of transactions in local schedules and ensures that local schedules are conflict serializable. Furthermore, local DBMSs support mechanisms for detecting and resolving all local deadlocks.

Global serializability in the MDBS is said to be assured if the union of the conflict serialization graphs [1] of all the local schedules is acyclic. We next define the class of rigorous schedules and strongly recoverable schedules.

Definition 1: A schedule $S$ is said to be rigorous if, for all pairs of transactions $T_i, T_j$ in $S$, if $T_i$ is in conflict with $T_j$, then $T_j$ does not execute its conflicting operation

\footnote{Note that conflict is not a symmetric relation, that is, it is possible that $T_i$ is in conflict with $T_j$, but $T_j$ is not in conflict with $T_i$.}
Definition 2: A schedule \( S \) is said to be strongly recoverable if \( S \) is recoverable and further for all pairs of transactions \( T_i, T_j \) in \( S \), if \( T_i \) is in conflict with \( T_j \), then \( T_j \) does not commit before \( T_i \) commits or aborts. 

Every rigorous schedule is also strongly recoverable. The converse, however, is not true. For example, the local schedule \( S_1: r_1(x) w_1(x) c_1 c_2 \) is strongly recoverable but not rigorous.

3 Ensuring Global Serializability

In this section, we develop concurrency control protocols for ensuring global serializability. We will consider the problem of making our schemes failure-resilient in the next section.

Let us first consider the situation in which each local DBMS produces rigorous schedules. If each local DBMS produces rigorous schedules, then to ensure global serializability, it suffices to ensure that a global transaction \( Ti \) does not execute any database operations (that is, read and write operations) after its commit operation has executed at even one site. Since in our model, we have assumed that the local DBMSs acknowledge the execution of each operation, and further, since the commit operations of the transaction execute only after the acknowledgment of each database operation, to ensure global serializability in our system no global scheduling is required. That is, the GTM may submit the operations of the global transactions for execution to the various local DBMSs as soon as they arrive at the GTM. We refer to such a GTM in which the operations of the global transactions are submitted for execution to the various local DBMSs as soon as they arrive as a trivial GTM.

Theorem 1: Global serializability is ensured by a trivial GTM if each local DBMS generates rigorous schedules. 

Let us now turn our attention to ensuring global serializability if each local DBMS generates strongly recoverable schedules. First, we note that if each local DBMS only generates strongly recoverable schedules, then global serializability may not be ensured in a trivial MDBS, even if GTM uses the 2PC protocol to execute a commit operation. This is illustrated by the following example.

Example 1: Consider an MDBS consisting of two sites: site \( s_1 \) with data item \( z \) and site \( s_2 \) with data item \( y \). Let \( T_1 \) and \( T_2 \) be the following global transactions.

\[
T_1 : w_1(z) w_1(y) \\
T_2 : r_2(z) r_2(y)
\]

Since a trivial GTM does not coordinate the global transaction operations, the following execution may result.

\[
S_1 : r_2(z) w_1(z) c_2 c_1 \\
S_2 : w_1(y) c_1 r_2(y) c_2
\]

Note that even though both schedules \( S_1 \) and \( S_2 \) are strongly recoverable, global serializability is violated.

The above example suggests that if each local DBMS generates only strongly recoverable schedules, then to ensure global serializability, the GTM needs to control the order in which operations belonging to the global transactions execute. Before we develop the GTM concurrency control scheme, we examine certain properties of the strongly recoverable schedule which will guide our solution.

We first note that every strongly recoverable local schedule is serializable in an order consistent with the commit order of transactions. This leads us to the following theorem.

Theorem 2: Consider an MDBS where each local DBMS produces strongly recoverable schedules. Global serializability is assured if there exists a total order on global transactions such that at each site \( s_k \), for all pairs of global transactions \( Ti, Tj \) executing at \( s_k \), if \( Ti \) commits before \( Tj \) at \( s_k \), then \( Ti \) precedes \( Tj \) in the total order.

A consequence of Theorem 2 is that if the GTM executes global transactions serially and each of the local schedules is strongly recoverable, then global serializability is ensured. Further, in an MDBS where each local DBMS generates strongly recoverable schedules, it is sufficient to synchronize only the commit operations of the global transaction in order to ensure global serializability. Below, we discuss a scheme that utilizes the notion of a commit graph introduced in [4] to control the order in which commit operations of the global transactions execute. A commit graph is an undirected bipartite graph consisting of a set of nodes \( V \) corresponding to sites (site nodes) and transactions (transaction nodes). Edges in the commit graph are present only between transaction nodes and site nodes. An edge between a transaction node \( Ti \) and a site node \( s_k \) is present only if transaction \( Ti \) executes at site \( s_k \), and \( Ti \)'s commit operation is submitted to \( s_k \).

Algorithm TM1

1. When a global transaction \( Ti \) submits its commit operation to the GTM, the GTM temporarily adds to the commit graph, edges \((Ti, s_k)\) for each site \( s_k \) at which \( Ti \) executes.

2. If the augmented commit graph does not contain a cycle, then the global commit operation is submitted by the GTM to local DBMSs for execution and the temporary edges become permanent.
3. If the augmented commit graph does contain a cycle, then the GTM places global transaction $T_i$ on a *waiting queue*, and removes the temporary edges from the commit graph.

4. Upon completion of a global transaction $T_i$'s commit operation at the local DBMSs, the GTM performs the following: if there is no path in the commit graph from $T_i$ to any other transaction $T_j$ that has not completed its commit operation at some local DBMS, then delete from the commit graph $T_i$ and all transactions $T_j$ such that there is a path in the commit graph from $T_i$ to $T_j$, along with all edges incident on these nodes.

5. If a transaction is deleted from the commit graph, the GTM checks the commit graph (using steps 1 and 2) whether the commit operation of any transaction $T_i$ in the waiting queue can be submitted to the local DBMSs.

The GTM algorithm, TM1, presented above uses a commit graph in determining if a global commit operation can be submitted for execution to the local DBMSs. The only operations that are potentially delayed are global transaction commit operations. In essence, algorithm TM1 guarantees global serializability by preventing cycles in the commit graph.

**Theorem 3:** If each local DBMS generates strongly recoverable schedules, then algorithm TM1 ensures global serializability. □

## 4 Dealing with Failures

In this section we consider the problem of making our concurrency control scheme resilient to failures. Since an MDBS is a distributed database system, our scheme must be resilient to various kinds of failures (e.g., site, communication, transaction aborts). Presence of failures complicate the task of ensuring atomicity of transactions. Since, in our model, we assume that each local DBMS has pre-existing recovery procedures that ensure atomicity of local transactions and global subtransactions executing in its control, we can restrict ourselves to ensuring atomicity of global transactions in presence of failures.

It is well known that in homogeneous distributed database systems, the two-phase commit (2PC) protocol [1] can be used to ensure the atomicity of global transactions in the presence of failures. In the 2PC protocol, a coordinator, on completion of a transaction $T_i$, sends a VOTE-REQ message to all the local sites at which $T_i$ executed (referred to as participants). Upon receipt of a VOTE-REQ message, a local site responds by sending its vote: a YES (commit) or a NO (abort). If the local site at site $s_k$ votes to commit transaction $T_i$, it forces the log records corresponding to the writes done by subtransaction $T_{ik}$ onto stable storage before responding. If all the participating local sites respond with a YES, then the coordinator decides to commit transaction $T_i$ and sends a COMMIT message to each participating local site. Else, it sends an ABORT message to the local sites. The local site at site $s_k$ upon receiving the coordinator’s decision, commits or aborts $T_{ik}$.

Between the time site $s_k$ responds by sending its YES vote to the coordinator and the time $T_{ik}$ commits or aborts at $s_k$, $T_{ik}$ is in a special state referred to as the *prepared state*. Local sites cannot unilaterally commit or abort a global subtransaction in the prepared state.

The mechanism of ensuring atomicity of global transactions in MDBSs differs widely, depending upon whether or not each local DBMS supports a prepared state. If each local DBMS supports a prepared state, then atomicity can be ensured in a manner similar to that of homogeneous distributed database systems by employing the above discussed 2PC protocol. However, supporting a prepared state for transactions results in the violation of the execution autonomy of the participating local DBMSs (since the local DBMS cannot unilaterally abort a transaction in the prepared state). Since the preservation of local autonomy is crucial in many applications, some local sites, even if they support a prepared state for transactions, they may be unwilling to participate in an atomic commit protocol to commit global transactions.

If a local DBMS does not participate in an atomic commit protocol to commit a global transaction, it may unilaterally abort a global subtransaction at any time during its execution. Specifically, it may abort a subtransaction of a global transaction even after the GTM has decided to commit the global transaction. Consequently, it is possible that a global transaction is aborted at some local DBMSs and committed at some other DBMSs. In order to ensure global transaction atomicity, in such a case, the only recovery option may be to construct for each aborted subtransaction, a transaction, consisting of the updates made by the aborted subtransaction, and submit it for execution to the local DBMS.

We refer to such a transaction as the *redo* transaction and the above approach to recovery from subtransaction aborts as the *redo approach*. Before we describe a global commit protocol based on the redo approach to recovery, we first note that if the redo approach is used to recover from the subtransaction aborts, then a trivial GTM cannot be used for ensuring global serializability, even if each local DBMS produces rigorous schedules. This is illustrated in the following example.

**Example 2:** Consider an MDBS consisting of two sites: site $s_1$ with data item $x$ and site $s_2$ with data item $y$. Let $T_1$ and $T_2$ be the following global transactions:

$$T_1 : w_1(x) w_2(y)$$
Suppose that the GTM decides to commit $T_1$. Further suppose that $T_1$ successfully commits at $s_2$, but the local DBMS at $s_1$ aborts $T_1$. The server at $s_1$ constructs the following redo transaction $T_3$ and submits it for execution to the local DBMS.

$$T_3 : \nu_0(x)$$

However, before $T_3$ is executed, global transaction $T_2$ executes at both sites $s_1$ and $s_2$ and commits at both the sites, following which the redo transaction $T_2$ executes at $s_1$ and commits. This results in the following local schedules. In the schedule, the write operation of $T_1$ that aborted at $s_1$ is included in angular brackets for clarity and is undone by the local DBMS at $s_1$.

$$S_1 : \{w_1(x)\} a_1 w_2(x) c_2 w_3(x) c_3$$
$$S_2 : w_1(y) c_1 r_2(y) c_2$$

Note that since a trivial GTM does not control the order of execution of the global transaction operations, the above schedule would be permitted by a trivial GTM. Further, local schedules $S_1$ and $S_2$ are rigorous, illustrating that if the redo approach is combined with the trivial GTM, then even if local DBMSs are rigorous, global serializability may not be ensured.

In the remainder of the section, we will only consider the problem of how to combine the redo approach to recovery with our concurrency control protocol based on maintaining a commit graph to ensure correctness under the assumption that each local DBMS generates strongly recoverable schedules. Note that since each rigorous schedule is also strongly recoverable, the same solution can be used in case each local DBMS generates rigorous schedules. We begin by first describing a global commit protocol for an MDBS environment where the local DBMSs do not support a prepared state. The protocol, to ensure global transaction atomicity, uses the redo approach to recovery.

Upon the completion of a global transaction, the GTM initiates the 2PC protocol with the servers at sites at which the transaction executed. In this protocol, the GTM plays a role of a coordinator, while each server at sites at which the transaction is executed is considered as a participant. Each server maintains its own server log in which it records all changes made by the global transaction to the local database. The server, before responding to the GTM's VOTE.REQ message with a YES vote, forces the log records in the server log corresponding to writes done by the transaction onto stable storage. Note that the servers, and not the local DBMSs, participate in the global commit protocol. The server submits a commit operation to the local DBMS only after the GTM has reached a decision to commit the transaction.

We now discuss how recovery from failures is done in a system that uses the above commit protocol. We limit our discussion to recovery from global subtransaction aborts since recovery from other kinds of failures (e.g., local DBMS failure) are simple extensions to recovery from transaction aborts. Global subtransactions may abort for a variety of reasons. A subtransaction may be aborted by the local DBMS due to a logical error, it may be chosen as a victim by the local deadlock detection algorithm, or it may be aborted by the local DBMS due to a timeout constraint.

If a failure occurs before the server has voted to commit the transaction, then the commit protocol will result in the decision to abort the global transaction. Since we assume that the local DBMSs undo all the effects of an aborted transaction on local databases, global transaction atomicity will be preserved. If, however, a global subtransaction abort occurs after the server at the site has voted to commit the transaction, and a global decision has been reached to commit the transaction, then the server constructs a redo transaction consisting of all the updates performed by a transaction at the site, and submits the redo transaction for execution to the local DBMS. A redo transaction, in case it aborts, is repeatedly submitted to the local DBMSs until it commits. Note that since a redo transaction is a write-only transaction, it cannot fail logically.

Although the above protocol guarantees global transaction atomicity, it nevertheless, suffers from the problem that serializability of even the local schedules is not always maintained. The reason for this is that the redone global subtransaction is considered by the local DBMS as a completely different transaction unrelated to the transaction that has been aborted. However, from the MDBS viewpoint, the redone transaction is part of the same global subtransaction that aborted. To illustrate the difficulty, let us consider an example of a schedule that is not serializable from the MDBS viewpoint but could be generated if the above global commit protocol is used.

Example 3: Consider an MDBS consisting of two sites: site $s_1$ with data items $x$, $y$, and site $s_2$ with data item $z$. Let $T_1$ be the following global transaction.

$$T_1 : r_1(y) w_1(z) w_1(x)$$

Suppose that the GTM decides to commit $T_1$ (using the global commit protocol). Further suppose that $T_1$ successfully commits at $s_2$, but the local DBMS at $s_1$ aborts $T_1$. The server at $s_1$ constructs the following redo transaction $T_3$ from the server log and submits it to the local DBMS.

$$T_3 : \nu_0(x)$$

However, before $T_3$ is executed, a local transaction:
executes at $s_1$. This results in the following local schedule at $s_1$:

$S_1 : r_1(y) w_1(x) a_1 r_2(x) w_2(y) c_1 r_3(x) w_3(y) c_2$

Since the local DBMS at $s_1$ considers $T_1$ and $T_2$ to be separate transactions, schedule $S_1$ is serializable in the local view (that is, in the view of the local DBMS). However, from the global MDBS viewpoint, since the read done by $T_1$ and the write performed by $T_2$ are part of one transaction, $S_1$ is not serializable.

Since our concurrency control protocol ensures global serializability only under the assumption that each local schedule is serializable, we, therefore, must devise means to ensure serializability of each local schedule. Example 3 demonstrates that if the redo approach to recovery is used, it is impossible to recover from global subtransaction aborts without violation of global serializability unless some restrictions are placed on data items accessed by local and global transactions. These restrictions ensure that global and local transactions do not interact in adverse ways as illustrated in Example 3. To identify the required restrictions, as we proposed in [4], we first partition the set of data items at each site into:

- **global data items** - those data items that can only be modified by global transactions.
- **local data items** - all other data items.

An example of global data items are the set of replicated data item (local transactions do not write on replicated data items since that will violate the integrity constraints of the system). On the other hand, the set of data items which pre-existing local applications may modify are local data items. We impose the following restriction on data items accessed by global subtransactions:

**If a global subtransaction reads a local data item, then it does not write any data item at the site.**

The above restriction is minimal, in the sense that, if a global transaction that reads local data items were to write either local or global data items, then it is possible for schedules to be non-serializable from the MDBS viewpoint. In Example 3, since local transaction $T_3$ writes data item $y$, it must be the case that $y$ is a local data item. Thus, since the global subtransaction of $T_1$ at site $s_1$ reads local data item $y$ and writes data item $z$, it violates the above restriction resulting in a loss of serializability in the MDBS viewpoint. Though the above restriction on global transactions is necessary, it is, by itself not sufficient for ensuring serializability of the local schedules. Serializability of the local schedules can, however, be ensured if the MDBS follows the early commit (EC) protocol introduced in [8] and further certain modifications are made to the GTM concurrency control protocol. We discuss these in turn.

The EC protocol is similar to the global commit protocol developed earlier, except that the GTM, in the first phase of the protocol, instead of sending a VOTE-REQ message, sends a COMMIT message to the servers at all the sites at which the subtransaction is a read-only transaction. We refer to such servers as r-servers. To the remaining servers, referred to as w-servers, the GTM sends a VOTE-REQ message. An r-server, on receipt of a COMMIT message from the GTM submits a commit operation for the transaction to the local DBMS. On receipt of an acknowledgement from the local DBMS, the server sends a COMMIT-ACK message to the GTM; if the subtransaction is aborted by the local DBMS, then the server sends an ABORT-ACK message to the GTM. A w-server behaves as in the previous case and sends its vote: YES (commit) or a NO (abort). If the GTM receives a YES from each w-server and a COMMIT-ACK from each r-server, it decides to commit the transaction and sends a COMMIT message to each of the w-servers. If it receives a NO from some w-server or an ABORT-ACK from some r-server, it decides to abort the transaction and sends an ABORT message to each of the w-servers. A w-server, on receipt of a message containing the decision of the GTM, submits either a commit or an abort operation to the local DBMS (depending upon the decision). In the EC protocol, committing the read-only subtransactions of a global transaction early does not result in a violation of its atomicity since the read-only subtransactions do not cause any changes to the database.

We next discuss the modifications needed to the GTM concurrency control protocol such that using the redo approach to recovery does not result in a loss of serializability of the local schedules. To motivate the need for the modification, we first present an example in which loss of serializability of a local schedule results even though each global transaction satisfies the restrictions stated above and the GTM uses the EC protocol to commit global transactions.

**Example 4:** Consider an MDBS consisting of three sites that produce rigorous local schedules: site $s_1$ with global data items $x$, $y$, site $s_2$ with global data item $u$, and $s_3$ with global data item $v$. Let $T_1$ and $T_2$ be global transactions and $T_3$ be a local transaction that executes at site $s_1$.

$T_1 : r_1(x) w_1(y) w_1(u)$

$T_2 : w_2(x) w_2(v)$

$T_3 : r_3(x) r_3(y)$

Suppose the GTM decides to commit $T_1$ (using the EC protocol). Further, suppose that $T_1$ successfully commits
at \( s_3 \), but the local DBMS at \( s_1 \) aborts \( T_1 \). Transaction \( T_2 \) then executes at sites \( s_1 \) and \( s_2 \), and since there are no cycles in the commit graph, the GTM decides to commit \( T_2 \). Also, since the GTM considers \( T_1 \) to be committed, it executes the following redo transactions \( T_4 \) for \( T_1 \).

\[
T_4 : w_4(y)
\]

It is possible for transaction \( T_3 \) to execute at site \( s_1 \) in such a manner that the local schedule at site \( s_1 \) is as follows.

\[
S_1 : r_1(x) [w_1(y)] a_1 w_2(x) c_2 r_3(x) r_4(y) c_3 w_4(y) c_4
\]

Since \( T_1 \) and \( T_4 \) are the same transaction in the MDBS viewpoint, global serializability is lost. □

To ensure that each local schedule is serialisable and further global serializability is ensured, we need to modify the algorithm TM1 such that if two global transactions are executing their commit operations concurrently at a site, then they are not in a rw or wr conflict at the site. The modified algorithm TM2 is described below.

**Algorithm TM2**

1. When a global transaction \( T_i \) submits its commit operation to the GTM, the GTM temporarily adds to the commit graph, edges \((T_i, s_k)\) for each site \( s_k \) at which \( T_i \) executes.

2. If the augmented commit graph does contain a cycle, then the GTM places global transaction \( T_i \) on the waiting queue and deletes the temporary edges from the commit graph.

3. If the augmented commit graph does not contain a cycle, the GTM checks whether there is a global transaction \( T_j \) that the GTM has not decided to abort, such that at some site \( s_k \),

   - \( T_j \) is in a \( wr \) or \( rw \) conflict with \( T_{ik} \), and
   - \( T_j \) has not yet committed at site \( s_k \).

4. If such a transaction exists, the GTM places \( T_i \) on the waiting queue and deletes the temporary edges from the commit graph.

5. If no such transaction exists, then the GTM initiates the EC protocol for \( T_i \) and temporary edges in the commit graph become permanent.

6. If at some site \( s_k \), subtransaction \( T_{ik} \) is aborted by the local DBMS, after the GTM has decided to commit global transaction \( T_i \), then every global transaction \( T_j \) that reads data items written by \( T_{ik} \) is aborted. The server at site \( s_k \) then redoes \( T_{ik} \)'s write operations at site \( s_k \) until the redo transaction commits at \( s_k \). Furthermore, every time the redo transaction is aborted by site \( s_k \), every uncommitted global transaction \( T_j \) that reads data items written by \( T_{ik} \) is aborted.

7. Upon completion of a global transaction \( T_i \)'s commit operation at the local DBMSs, the GTM performs the following: if there is no path in the commit graph from \( T_i \) to any other transaction \( T_j \) that has not completed its commit operation at some local DBMS, then delete from the commit graph \( T_i \) and all transactions \( T_j \) such that there is a path in the commit graph from \( T_i \) to \( T_j \), along with all edges incident on these nodes.

8. If a transaction along with all incidental edges is deleted from the commit graph, the GTM checks the commit graph (using steps 1, 2 and 3) whether the commit operation of any transaction \( T_i \) in the waiting queue can be submitted to the local DBMSs.

In a failure-free environment, algorithm TM2 performs almost identically to algorithm TM1 with the only difference being that TM2 keeps track of the \( rw \) and \( wr \) conflicts between global transactions, while TM1 does not. However, if a global subtransaction \( T_{ik} \) at site \( s_k \) is aborted by a local DBMS after the GTM has decided to commit \( T_i \), then TM2 aborts all other global transactions \( T_j \) such that \( T_{ik} \) is in a \( wr \) conflict with \( T_{jk} \). In addition, any global transaction that submits an operation to read some data item written by the aborted subtransaction (until the redo transaction corresponding to the aborted subtransaction is successfully committed) is also aborted. We could have, instead, resorted to delaying such read operations until the failed subtransaction is redone.

**Theorem 4:** Consider an MDBS environment where the GTM follows the EC protocol and each local DBMS is strongly recoverable. If the GTM follows algorithm TM2, then global serializability is assured. □

In algorithm TM2, the additional checks performed in step 3 are essential, since otherwise, local schedules may not be serializable from the MDBS viewpoint (in the presence of global subtransaction aborts). This is illustrated by schedule \( S_k \) in Example 4, which is non-serializable since the GTM permitted the commit operation of global transaction \( T_2 \) to be processed even though \( T_1 \) was in a \( rw \) conflict with \( T_2 \). A similar example in which non-serializable executions may result if the GTM permitted the commit operation of a transaction in a \( wr \) conflict to execute can also be constructed.

In the above discussed algorithm TM2 the GTM only delays the commit operations of global transactions. In

\[2\] For this, the GTM would need to submit to the local DBMS, a read (write) operation that conflicts with a previously submitted write (read) operation, only after the acknowledgement for the execution of the previous operation has been received.
contrast, the schemes proposed in [4, 3] require not just the commit but also read and write operations of the global transactions to be delayed. Further, as opposed to TMs that only requires each local DBMS to be strongly recoverable, the scheme proposed in [4, 12] assumes that each local DBMS produces rigorous schedules.

5 Conclusions

A multidatabase system (MDBS) is a facility that allows access to data located in multiple pre-existing and autonomous database management systems (DBMSs). In such a system a software package built on top of the autonomous local DBMSs, referred to as the global transaction manager (GTM), coordinates the execution of global transactions that execute at multiple sites. Independently, local transactions execute at a single site outside the control of the GTM.

We studied, in detail, how global serializability can be ensured, both in presence and in the absence of failures, in an MDBS under two different assumptions on the nature of schedules produced by the participating local DBMSs: (1) Each local DBMS generates rigorous schedules, (2) Each local DBMS generates strongly recoverable schedules. Our interest in the rigorous schedules stems from the fact that the majority of commercial DBMSs offer a concurrency control protocol that generates rigorous schedules. However, the class of rigorous schedules is quite restrictive in terms of the amount of concurrency permitted. We, therefore, also considered the class of strongly recoverable schedules which permit a higher degree of concurrency as compared to the rigorous schedules. We proposed schemes that exploit the rigorousness and strong recoverability properties of the local schedules to ensure global serializability with minimal global overhead. We studied mechanisms to make our concurrency control technique failure-resilient under the scenario that each local DBMS supports a prepared state for the execution of the two-phase commit protocol as well as under the scenario that it does not.

References


